GNU Compiler Collection Internals (GCC)

GNU Compiler Collection Internals

by Richard M. Stallman and the GCC Developer Community

For GCC Version 4.1.1

Copyright © 1988, 1989, 1992, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being "GNU General Public License" and "Funding Free Software", the Front-Cover texts being (a) (see below), and with the Back-Cover Texts being (b) (see below). A copy of the license is included in the section entitled "GNU Free Documentation License".

(a) The FSF's Front-Cover Text is:

A GNU Manual

(b) The FSF's Back-Cover Text is:

You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.

Short Contents

Introduction 1
1 Contributing to GCC Development 3
2 GCC and Portability 5
3 Interfacing to GCC Output 7
4 The GCC low-level runtime library 9
5 Language Front Ends in GCC 17
6 Source Tree Structure and Build System 19
7 Option specification files 45
8 Passes and Files of the Compiler 49
9 Trees: The intermediate representation used by the C and C++ front ends 61
10 Analysis and Optimization of GIMPLE Trees 97
11 RTL Representation 121
12 Control Flow Graph 169
13 Machine Descriptions 179
14 Target Description Macros and Functions 267
15 Host Configuration 405
16 Makefile Fragments 409
17 `collect2` 413
18 Standard Header File Directories 415
19 Memory Management and Type Information 417
Funding Free Software 423
The GNU Project and GNU/Linux 425
GNU GENERAL PUBLIC LICENSE 427
GNU Free Documentation License 433
Contributors to GCC 441
Option Index 457
Concept Index 459

Table of Contents

Introduction 1

1 Contributing to GCC Development 3

2 GCC and Portability 5

3 Interfacing to GCC Output 7

4 The GCC low-level runtime library 9

4.1 Routines for integer arithmetic 9
4.1.1 Arithmetic functions 9
4.1.2 Comparison functions 10
4.1.3 Trapping arithmetic functions 11
4.1.4 Bit operations 11
4.2 Routines for floating point emulation 12
4.2.1 Arithmetic functions 12
4.2.2 Conversion functions 13
4.2.3 Comparison functions 14
4.3 Language-independent routines for exception handling 15
4.4 Miscellaneous runtime library routines 16
4.4.1 Cache control functions 16

5 Language Front Ends in GCC 17

6 Source Tree Structure and Build System . . . 19

6.1 Configure Terms and History 19
6.2 Top Level Source Directory 19
6.3 The ‘`gcc`’ Subdirectory 20
6.3.1 Subdirectories of ‘`gcc`’ 21
6.3.2 Configuration in the ‘`gcc`’ Directory 21
6.3.2.1 Scripts Used by ‘`configure`’ 22
6.3.2.2 The ‘`config.build`’; ‘`config.host`’; and ‘`config.gcc`’ Files 22
6.3.2.3 Files Created by `configure` 22
6.3.3 Build System in the ‘`gcc`’ Directory 23
6.3.4 Makefile Targets 23
6.3.5 Library Source Files and Headers under the ‘`gcc`’ Directory 25
6.3.6 Headers Installed by GCC 25
6.3.7 Building Documentation 26
6.3.7.1 Texinfo Manuals 26

6.3.7.2 Man Page Generation ... 27
6.3.7.3 Miscellaneous Documentation ... 27
6.3.8 Anatomy of a Language Front End ... 28
6.3.8.1 The Front End '*language*' Directory ... 29
6.3.8.2 The Front End '`config-lang.in`' File ... 31
6.3.9 Anatomy of a Target Back End ... 32
6.4 Testsuites ... 33
6.4.1 Idioms Used in Testsuite Code ... 33
6.4.2 Directives used within DejaGnu tests ... 35
6.4.3 Ada Language Testsuites ... 38
6.4.4 C Language Testsuites ... 39
6.4.5 The Java library testsuites. ... 40
6.4.6 Support for testing `gcov` ... 41
6.4.7 Support for testing profile-directed optimizations ... 42
6.4.8 Support for testing binary compatibility ... 42

7 Option specification files ... 45
7.1 Option file format ... 45
7.2 Option properties ... 45

8 Passes and Files of the Compiler ... 49
8.1 Parsing pass ... 49
8.2 Gimplification pass ... 50
8.3 Pass manager ... 50
8.4 Tree-SSA passes ... 51
8.5 RTL passes ... 55

9 Trees: The intermediate representation used by the C and C++ front ends ... 61
9.1 Deficiencies ... 61
9.2 Overview ... 61
9.2.1 Trees ... 62
9.2.2 Identifiers ... 62
9.2.3 Containers ... 63
9.3 Types ... 63
9.4 Scopes ... 68
9.4.1 Namespaces ... 68
9.4.2 Classes ... 69
9.5 Declarations ... 71
9.5.1 Working with declarations ... 71
9.5.2 Internal structure ... 73
9.5.2.1 Current structure hierarchy ... 73
9.5.2.2 Adding new DECL node types ... 74
9.6 Functions ... 76
9.6.1 Function Basics ... 76
9.6.2 Function Bodies ... 79
9.6.2.1 Statements ... 80

9.7 Attributes in trees 84
9.8 Expressions 84

10 Analysis and Optimization of GIMPLE Trees 97

10.1 GENERIC 97
10.2 GIMPLE 97
10.2.1 Interfaces 98
10.2.2 Temporaries 98
10.2.3 Expressions 99
10.2.3.1 Compound Expressions 99
10.2.3.2 Compound Lvalues 99
10.2.3.3 Conditional Expressions 99
10.2.3.4 Logical Operators 100
10.2.4 Statements 100
10.2.4.1 Blocks 100
10.2.4.2 Statement Sequences 101
10.2.4.3 Empty Statements 101
10.2.4.4 Loops 101
10.2.4.5 Selection Statements 101
10.2.4.6 Jumps 101
10.2.4.7 Cleanups 101
10.2.4.8 Exception Handling 102
10.2.5 GIMPLE Example 102
10.2.6 Rough GIMPLE Grammar 104
10.3 Annotations 106
10.4 Statement Operands 107
10.4.1 Operand Iterators And Access Routines 108
10.4.2 Immediate Uses 111
10.5 Static Single Assignment 112
10.5.1 Preserving the SSA form 113
10.5.2 Examining `SSA_NAME` nodes 115
10.5.3 Walking use-def chains 115
10.5.4 Walking the dominator tree 115
10.6 Alias analysis 116

11 RTL Representation 121

11.1 RTL Object Types 121
11.2 RTL Classes and Formats 122
11.3 Access to Operands 124
11.4 Access to Special Operands 125
11.5 Flags in an RTL Expression 127
11.6 Machine Modes 132
11.7 Constant Expression Types 136
11.8 Registers and Memory 138
11.9 RTL Expressions for Arithmetic 142
11.10 Comparison Operations 145

11.11 Bit-Fields ... 147
11.12 Vector Operations ... 147
11.13 Conversions ... 148
11.14 Declarations ... 149
11.15 Side Effect Expressions ... 149
11.16 Embedded Side-Effects on Addresses ... 154
11.17 Assembler Instructions as Expressions ... 156
11.18 Insns ... 156
11.19 RTL Representation of Function-Call Insns ... 165
11.20 Structure Sharing Assumptions ... 165
11.21 Reading RTL ... 166

12 Control Flow Graph ... 169

12.1 Basic Blocks ... 169
12.2 Edges ... 170
12.3 Profile information ... 173
12.4 Maintaining the CFG ... 174
12.5 Liveness information ... 176

13 Machine Descriptions ... 179

13.1 Overview of How the Machine Description is Used ... 179
13.2 Everything about Instruction Patterns ... 179
13.3 Example of `define_insn` ... 180
13.4 RTL Template ... 181
13.5 Output Templates and Operand Substitution ... 184
13.6 C Statements for Assembler Output ... 186
13.7 Predicates ... 187
13.7.1 Machine-Independent Predicates ... 188
13.7.2 Defining Machine-Specific Predicates ... 190
13.8 Operand Constraints ... 191
13.8.1 Simple Constraints ... 191
13.8.2 Multiple Alternative Constraints ... 195
13.8.3 Register Class Preferences ... 196
13.8.4 Constraint Modifier Characters ... 197
13.8.5 Constraints for Particular Machines ... 198
13.9 Standard Pattern Names For Generation ... 212
13.10 When the Order of Patterns Matters ... 232
13.11 Interdependence of Patterns ... 232
13.12 Defining Jump Instruction Patterns ... 233
13.13 Defining Looping Instruction Patterns ... 234
13.14 Canonicalization of Instructions ... 236
13.15 Defining RTL Sequences for Code Generation ... 237
13.16 Defining How to Split Instructions ... 240
13.17 Including Patterns in Machine Descriptions. ... 243
13.17.1 RTL Generation Tool Options for Directory Search ... 244
13.18 Machine-Specific Peephole Optimizers ... 244
13.18.1 RTL to Text Peephole Optimizers ... 245
13.18.2 RTL to RTL Peephole Optimizers ... 247

13.19 Instruction Attributes 248
13.19.1 Defining Attributes and their Values 248
13.19.2 Attribute Expressions 249
13.19.3 Assigning Attribute Values to Insns 251
13.19.4 Example of Attribute Specifications 253
13.19.5 Computing the Length of an Insn 254
13.19.6 Constant Attributes 255
13.19.7 Delay Slot Scheduling 255
13.19.8 Specifying processor pipeline description 256
13.20 Conditional Execution 262
13.21 Constant Definitions 263
13.22 Macros 263
13.22.1 Mode Macros 263
13.22.1.1 Defining Mode Macros 264
13.22.1.2 Substitution in Mode Macros 264
13.22.1.3 Mode Macro Examples 265
13.22.2 Code Macros 265

14 Target Description Macros and Functions 267

14.1 The Global `targetm` Variable 267
14.2 Controlling the Compilation Driver, '`gcc`' 267
14.3 Run-time Target Specification 275
14.4 Defining data structures for per-function information. 277
14.5 Storage Layout 278
14.6 Layout of Source Language Data Types 286
14.7 Register Usage 290
14.7.1 Basic Characteristics of Registers 290
14.7.2 Order of Allocation of Registers 292
14.7.3 How Values Fit in Registers 293
14.7.4 Handling Leaf Functions 294
14.7.5 Registers That Form a Stack 295
14.8 Register Classes 296
14.9 Stack Layout and Calling Conventions 304
14.9.1 Basic Stack Layout 304
14.9.2 Exception Handling Support 307
14.9.3 Specifying How Stack Checking is Done 309
14.9.4 Registers That Address the Stack Frame 310
14.9.5 Eliminating Frame Pointer and Arg Pointer 312
14.9.6 Passing Function Arguments on the Stack 314
14.9.7 Passing Arguments in Registers 316
14.9.8 How Scalar Function Values Are Returned 320
14.9.9 How Large Values Are Returned 322
14.9.10 Caller-Saves Register Allocation 323
14.9.11 Function Entry and Exit 323
14.9.12 Generating Code for Profiling 327
14.9.13 Permitting tail calls 327
14.9.14 Stack smashing protection 328

14.10 Implementing the Varargs Macros 328
14.11 Trampolines for Nested Functions 330
14.12 Implicit Calls to Library Routines 333
14.13 Addressing Modes 334
14.14 Condition Code Status 338
14.15 Describing Relative Costs of Operations 341
14.16 Adjusting the Instruction Scheduler 345
14.17 Dividing the Output into Sections (Texts, Data, . . .) 349
14.18 Position Independent Code 353
14.19 Defining the Output Assembler Language 353
14.19.1 The Overall Framework of an Assembler File 353
14.19.2 Output of Data 355
14.19.3 Output of Uninitialized Variables 358
14.19.4 Output and Generation of Labels 359
14.19.5 How Initialization Functions Are Handled 366
14.19.6 Macros Controlling Initialization Routines 368
14.19.7 Output of Assembler Instructions 370
14.19.8 Output of Dispatch Tables 373
14.19.9 Assembler Commands for Exception Regions 374
14.19.10 Assembler Commands for Alignment 376
14.20 Controlling Debugging Information Format 378
14.20.1 Macros Affecting All Debugging Formats 378
14.20.2 Specific Options for DBX Output 379
14.20.3 Open-Ended Hooks for DBX Format 381
14.20.4 File Names in DBX Format 381
14.20.5 Macros for SDB and DWARF Output 382
14.20.6 Macros for VMS Debug Format 384
14.21 Cross Compilation and Floating Point 384
14.22 Mode Switching Instructions 386
14.23 Defining target-specific uses of `__attribute__` 387
14.24 Defining coprocessor specifics for MIPS targets. 388
14.25 Parameters for Precompiled Header Validity Checking 389
14.26 C++ ABI parameters 389
14.27 Miscellaneous Parameters 390

15 Host Configuration 405
15.1 Host Common 405
15.2 Host Filesystem 406
15.3 Host Misc 407

16 Makefile Fragments 409
16.1 Target Makefile Fragments 409
16.2 Host Makefile Fragments 411

17 `collect2` 413

18 Standard Header File Directories 415

19 Memory Management and Type Information 417

19.1 The Inside of a `GTY(())` 417
19.2 Marking Roots for the Garbage Collector 421
19.3 Source Files Containing Type Information 421

Funding Free Software 423

The GNU Project and GNU/Linux 425

GNU GENERAL PUBLIC LICENSE 427

Preamble 427
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 428
Appendix: How to Apply These Terms to Your New Programs 432

GNU Free Documentation License 433

ADDENDUM: How to use this License for your documents 439

Contributors to GCC 441

Option Index 457

Concept Index 459

Introduction

This manual documents the internals of the GNU compilers, including how to port them to new targets and some information about how to write front ends for new languages. It corresponds to GCC version 4.1.1. The use of the GNU compilers is documented in a separate manual. See section "Introduction" in *Using the GNU Compiler Collection (GCC)*.

This manual is mainly a reference manual rather than a tutorial. It discusses how to contribute to GCC (see Chapter 1 [Contributing], page 3), the characteristics of the machines supported by GCC as hosts and targets (see Chapter 2 [Portability], page 5), how GCC relates to the ABIs on such systems (see Chapter 3 [Interface], page 7), and the characteristics of the languages for which GCC front ends are written (see Chapter 5 [Languages], page 17). It then describes the GCC source tree structure and build system, some of the interfaces to GCC front ends, and how support for a target system is implemented in GCC.

Additional tutorial information is linked to from `http://gcc.gnu.org/readings.html`.

1 Contributing to GCC Development

If you would like to help pretest GCC releases to assure they work well, current development sources are available by CVS (see `http://gcc.gnu.org/cvs.html`). Source and binary snapshots are also available for FTP; see `http://gcc.gnu.org/snapshots.html`.

If you would like to work on improvements to GCC, please read the advice at these URLs:

```
http://gcc.gnu.org/contribute.html
http://gcc.gnu.org/contributewhy.html
```

for information on how to make useful contributions and avoid duplication of effort. Suggested projects are listed at `http://gcc.gnu.org/projects/`.

2 GCC and Portability

GCC itself aims to be portable to any machine where `int` is at least a 32-bit type. It aims to target machines with a flat (non-segmented) byte addressed data address space (the code address space can be separate). Target ABIs may have 8, 16, 32 or 64-bit `int` type. `char` can be wider than 8 bits.

GCC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, ad-hoc parameters have been defined for machine descriptions. The purpose of portability is to reduce the total work needed on the compiler; it was not of interest for its own sake.

GCC does not contain machine dependent code, but it does contain code that depends on machine parameters such as endianness (whether the most significant byte has the highest or lowest address of the bytes in a word) and the availability of autoincrement addressing. In the RTL-generation pass, it is often necessary to have multiple strategies for generating code for a particular kind of syntax tree, strategies that are usable for different combinations of parameters. Often, not all possible cases have been addressed, but only the common ones or only the ones that have been encountered. As a result, a new target may require additional strategies. You will know if this happens because the compiler will call `abort`. Fortunately, the new strategies can be added in a machine-independent fashion, and will affect only the target machines that need them.

3 Interfacing to GCC Output

GCC is normally configured to use the same function calling convention normally in use on the target system. This is done with the machine-description macros described (see Chapter 14 [Target Macros], page 267).

However, returning of structure and union values is done differently on some target machines. As a result, functions compiled with PCC returning such types cannot be called from code compiled with GCC, and vice versa. This does not cause trouble often because few Unix library routines return structures or unions.

GCC code returns structures and unions that are 1, 2, 4 or 8 bytes long in the same registers used for `int` or `double` return values. (GCC typically allocates variables of such types in registers also.) Structures and unions of other sizes are returned by storing them into an address passed by the caller (usually in a register). The target hook `TARGET_STRUCT_VALUE_RTX` tells GCC where to pass this address.

By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. This is slower than the method used by GCC, and fails to be reentrant.

On some target machines, such as RISC machines and the 80386, the standard system convention is to pass to the subroutine the address of where to return the value. On these machines, GCC has been configured to be compatible with the standard compiler, when this method is used. It may not be compatible for structures of 1, 2, 4 or 8 bytes.

GCC uses the system's standard convention for passing arguments. On some machines, the first few arguments are passed in registers; in others, all are passed on the stack. It would be possible to use registers for argument passing on any machine, and this would probably result in a significant speedup. But the result would be complete incompatibility with code that follows the standard convention. So this change is practical only if you are switching to GCC as the sole C compiler for the system. We may implement register argument passing on certain machines once we have a complete GNU system so that we can compile the libraries with GCC.

On some machines (particularly the SPARC), certain types of arguments are passed "by invisible reference". This means that the value is stored in memory, and the address of the memory location is passed to the subroutine.

If you use `longjmp`, beware of automatic variables. ISO C says that automatic variables that are not declared `volatile` have undefined values after a `longjmp`. And this is all GCC promises to do, because it is very difficult to restore register variables correctly, and one of GCC's features is that it can put variables in registers without your asking it to.

4 The GCC low-level runtime library

GCC provides a low-level runtime library, '`libgcc.a`' or '`libgcc_s.so.1`' on some platforms. GCC generates calls to routines in this library automatically, whenever it needs to perform some operation that is too complicated to emit inline code for.

Most of the routines in `libgcc` handle arithmetic operations that the target processor cannot perform directly. This includes integer multiply and divide on some machines, and all floating-point operations on other machines. `libgcc` also includes routines for exception handling, and a handful of miscellaneous operations.

Some of these routines can be defined in mostly machine-independent C. Others must be hand-written in assembly language for each processor that needs them.

GCC will also generate calls to C library routines, such as `memcpy` and `memset`, in some cases. The set of routines that GCC may possibly use is documented in section "Other Builtins" in *Using the GNU Compiler Collection (GCC)*.

These routines take arguments and return values of a specific machine mode, not a specific C type. See Section 11.6 [Machine Modes], page 132, for an explanation of this concept. For illustrative purposes, in this chapter the floating point type `float` is assumed to correspond to `SFmode`; `double` to `DFmode`; and `long double` to both `TFmode` and `XFmode`. Similarly, the integer types `int` and `unsigned int` correspond to `SImode`; `long` and `unsigned long` to `DImode`; and `long long` and `unsigned long long` to `TImode`.

4.1 Routines for integer arithmetic

The integer arithmetic routines are used on platforms that don't provide hardware support for arithmetic operations on some modes.

4.1.1 Arithmetic functions

`int __ashlsi3 (int a, int b)` [Runtime Function]
`long __ashldi3 (long a, int b)` [Runtime Function]
`long long __ashlti3 (long long a, int b)` [Runtime Function]
These functions return the result of shifting *a* left by *b* bits.

`int __ashrsi3 (int a, int b)` [Runtime Function]
`long __ashrdi3 (long a, int b)` [Runtime Function]
`long long __ashrti3 (long long a, int b)` [Runtime Function]
These functions return the result of arithmetically shifting *a* right by *b* bits.

`int __divsi3 (int a, int b)` [Runtime Function]
`long __divdi3 (long a, long b)` [Runtime Function]
`long long __divti3 (long long a, long long b)` [Runtime Function]
These functions return the quotient of the signed division of *a* and *b*.

`int __lshrsi3 (int a, int b)` [Runtime Function]
`long __lshrdi3 (long a, int b)` [Runtime Function]
`long long __lshrti3 (long long a, int b)` [Runtime Function]
These functions return the result of logically shifting *a* right by *b* bits.

`int __modsi3 (int a, int b)` [Runtime Function]
`long __moddi3 (long a, long b)` [Runtime Function]
`long long __modti3 (long long a, long long b)` [Runtime Function]
These functions return the remainder of the signed division of *a* and *b*.

`int __mulsi3 (int a, int b)` [Runtime Function]
`long __muldi3 (long a, long b)` [Runtime Function]
`long long __multi3 (long long a, long long b)` [Runtime Function]
These functions return the product of *a* and *b*.

`long __negdi2 (long a)` [Runtime Function]
`long long __negti2 (long long a)` [Runtime Function]
These functions return the negation of *a*.

`unsigned int __udivsi3 (unsigned int a, unsigned int b)` [Runtime Function]
`unsigned long __udivdi3 (unsigned long a, unsigned long b)` [Runtime Function]
`unsigned long long __udivti3 (unsigned long long a, unsigned long long b)` [Runtime Function]
These functions return the quotient of the unsigned division of *a* and *b*.

`unsigned long __udivmoddi3 (unsigned long a, unsigned long b, unsigned long *c)` [Runtime Function]
`unsigned long long __udivti3 (unsigned long long a, unsigned long long b, unsigned long long *c)` [Runtime Function]
These functions calculate both the quotient and remainder of the unsigned division of *a* and *b*. The return value is the quotient, and the remainder is placed in variable pointed to by *c*.

`unsigned int __umodsi3 (unsigned int a, unsigned int b)` [Runtime Function]
`unsigned long __umoddi3 (unsigned long a, unsigned long b)` [Runtime Function]
`unsigned long long __umodti3 (unsigned long long a, unsigned long long b)` [Runtime Function]
These functions return the remainder of the unsigned division of *a* and *b*.

4.1.2 Comparison functions

The following functions implement integral comparisons. These functions implement a low-level compare, upon which the higher level comparison operators (such as less than and greater than or equal to) can be constructed. The returned values lie in the range zero to two, to allow the high-level operators to be implemented by testing the returned result using either signed or unsigned comparison.

`int __cmpdi2 (long a, long b)` [Runtime Function]
`int __cmpti2 (long long a, long long b)` [Runtime Function]
These functions perform a signed comparison of *a* and *b*. If *a* is less than *b*, they return 0; if *a* is greater than *b*, they return 2; and if *a* and *b* are equal they return 1.

`int __ucmpdi2 (unsigned long a, unsigned long b)` [Runtime Function]
`int __ucmpti2 (unsigned long long a, unsigned long long b)` [Runtime Function]
These functions perform an unsigned comparison of *a* and *b*. If *a* is less than *b*, they return 0; if *a* is greater than *b*, they return 2; and if *a* and *b* are equal they return 1.

4.1.3 Trapping arithmetic functions

The following functions implement trapping arithmetic. These functions call the libc function `abort` upon signed arithmetic overflow.

`int __absvsi2 (int a)` [Runtime Function]
`long __absvdi2 (long a)` [Runtime Function]
These functions return the absolute value of *a*.

`int __addvsi3 (int a, int b)` [Runtime Function]
`long __addvdi3 (long a, long b)` [Runtime Function]
These functions return the sum of *a* and *b*; that is `a + b`.

`int __mulvsi3 (int a, int b)` [Runtime Function]
`long __mulvdi3 (long a, long b)` [Runtime Function]
The functions return the product of *a* and *b*; that is `a * b`.

`int __negvsi2 (int a)` [Runtime Function]
`long __negvdi2 (long a)` [Runtime Function]
These functions return the negation of *a*; that is `-a`.

`int __subvsi3 (int a, int b)` [Runtime Function]
`long __subvdi3 (long a, long b)` [Runtime Function]
These functions return the difference between *b* and *a*; that is `a - b`.

4.1.4 Bit operations

`int __clzsi2 (int a)` [Runtime Function]
`int __clzdi2 (long a)` [Runtime Function]
`int __clzti2 (long long a)` [Runtime Function]
These functions return the number of leading 0-bits in *a*, starting at the most significant bit position. If *a* is zero, the result is undefined.

`int __ctzsi2 (int a)` [Runtime Function]
`int __ctzdi2 (long a)` [Runtime Function]
`int __ctzti2 (long long a)` [Runtime Function]
These functions return the number of trailing 0-bits in *a*, starting at the least significant bit position. If *a* is zero, the result is undefined.

`int __ffsdi2 (long a)` [Runtime Function]
`int __ffsti2 (long long a)` [Runtime Function]
These functions return the index of the least significant 1-bit in *a*, or the value zero if *a* is zero. The least significant bit is index one.

`int __paritysi2 (int a)` [Runtime Function]
`int __paritydi2 (long a)` [Runtime Function]
`int __parityti2 (long long a)` [Runtime Function]
These functions return the value zero if the number of bits set in *a* is even, and the value one otherwise.

`int __popcountsi2 (`*int* `a)` [Runtime Function]
`int __popcountdi2 (`*long* `a)` [Runtime Function]
`int __popcountti2 (`*long long* `a)` [Runtime Function]
These functions return the number of bits set in *a*.

4.2 Routines for floating point emulation

The software floating point library is used on machines which do not have hardware support for floating point. It is also used whenever '`-msoft-float`' is used to disable generation of floating point instructions. (Not all targets support this switch.)

For compatibility with other compilers, the floating point emulation routines can be renamed with the `DECLARE_LIBRARY_RENAMES` macro (see Section 14.12 [Library Calls], page 333). In this section, the default names are used.

Presently the library does not support `XFmode`, which is used for `long double` on some architectures.

4.2.1 Arithmetic functions

`float __addsf3 (`*float* `a`, *float* `b)` [Runtime Function]
`double __adddf3 (`*double* `a`, *double* `b)` [Runtime Function]
`long double __addtf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
`long double __addxf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
These functions return the sum of *a* and *b*.

`float __subsf3 (`*float* `a`, *float* `b)` [Runtime Function]
`double __subdf3 (`*double* `a`, *double* `b)` [Runtime Function]
`long double __subtf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
`long double __subxf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
These functions return the difference between *b* and *a*; that is, $a - b$.

`float __mulsf3 (`*float* `a`, *float* `b)` [Runtime Function]
`double __muldf3 (`*double* `a`, *double* `b)` [Runtime Function]
`long double __multf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
`long double __mulxf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
These functions return the product of *a* and *b*.

`float __divsf3 (`*float* `a`, *float* `b)` [Runtime Function]
`double __divdf3 (`*double* `a`, *double* `b)` [Runtime Function]
`long double __divtf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
`long double __divxf3 (`*long double* `a`, *long double* `b)` [Runtime Function]
These functions return the quotient of *a* and *b*; that is, a/b.

`float __negsf2 (`*float* `a)` [Runtime Function]
`double __negdf2 (`*double* `a)` [Runtime Function]
`long double __negtf2 (`*long double* `a)` [Runtime Function]
`long double __negxf2 (`*long double* `a)` [Runtime Function]
These functions return the negation of *a*. They simply flip the sign bit, so they can produce negative zero and negative NaN.

4.2.2 Conversion functions

`double __extendsfdf2 (float a)` [Runtime Function]
`long double __extendsftf2 (float a)` [Runtime Function]
`long double __extendsfxf2 (float a)` [Runtime Function]
`long double __extenddftf2 (double a)` [Runtime Function]
`long double __extenddfxf2 (double a)` [Runtime Function]
These functions extend *a* to the wider mode of their return type.

`double __truncxfdf2 (long double a)` [Runtime Function]
`double __trunctfdf2 (long double a)` [Runtime Function]
`float __truncxfsf2 (long double a)` [Runtime Function]
`float __trunctfsf2 (long double a)` [Runtime Function]
`float __truncdfsf2 (double a)` [Runtime Function]
These functions truncate *a* to the narrower mode of their return type, rounding toward zero.

`int __fixsfsi (float a)` [Runtime Function]
`int __fixdfsi (double a)` [Runtime Function]
`int __fixtfsi (long double a)` [Runtime Function]
`int __fixxfsi (long double a)` [Runtime Function]
These functions convert *a* to a signed integer, rounding toward zero.

`long __fixsfdi (float a)` [Runtime Function]
`long __fixdfdi (double a)` [Runtime Function]
`long __fixtfdi (long double a)` [Runtime Function]
`long __fixxfdi (long double a)` [Runtime Function]
These functions convert *a* to a signed long, rounding toward zero.

`long long __fixsfti (float a)` [Runtime Function]
`long long __fixdfti (double a)` [Runtime Function]
`long long __fixtfti (long double a)` [Runtime Function]
`long long __fixxfti (long double a)` [Runtime Function]
These functions convert *a* to a signed long long, rounding toward zero.

`unsigned int __fixunssfsi (float a)` [Runtime Function]
`unsigned int __fixunsdfsi (double a)` [Runtime Function]
`unsigned int __fixunstfsi (long double a)` [Runtime Function]
`unsigned int __fixunsxfsi (long double a)` [Runtime Function]
These functions convert *a* to an unsigned integer, rounding toward zero. Negative values all become zero.

`unsigned long __fixunssfdi (float a)` [Runtime Function]
`unsigned long __fixunsdfdi (double a)` [Runtime Function]
`unsigned long __fixunstfdi (long double a)` [Runtime Function]
`unsigned long __fixunsxfdi (long double a)` [Runtime Function]
These functions convert *a* to an unsigned long, rounding toward zero. Negative values all become zero.

`unsigned long long __fixunssfti (float a)` [Runtime Function]
`unsigned long long __fixunsdfti (double a)` [Runtime Function]
`unsigned long long __fixunstfti (long double a)` [Runtime Function]
`unsigned long long __fixunsxfti (long double a)` [Runtime Function]
These functions convert *a* to an unsigned long long, rounding toward zero. Negative values all become zero.

`float __floatsisf (int i)` [Runtime Function]
`double __floatsidf (int i)` [Runtime Function]
`long double __floatsitf (int i)` [Runtime Function]
`long double __floatsixf (int i)` [Runtime Function]
These functions convert *i*, a signed integer, to floating point.

`float __floatdisf (long i)` [Runtime Function]
`double __floatdidf (long i)` [Runtime Function]
`long double __floatditf (long i)` [Runtime Function]
`long double __floatdixf (long i)` [Runtime Function]
These functions convert *i*, a signed long, to floating point.

`float __floattisf (long long i)` [Runtime Function]
`double __floattidf (long long i)` [Runtime Function]
`long double __floattitf (long long i)` [Runtime Function]
`long double __floattixf (long long i)` [Runtime Function]
These functions convert *i*, a signed long long, to floating point.

4.2.3 Comparison functions

There are two sets of basic comparison functions.

`int __cmpsf2 (float a, float b)` [Runtime Function]
`int __cmpdf2 (double a, double b)` [Runtime Function]
`int __cmptf2 (long double a, long double b)` [Runtime Function]
These functions calculate $a <=> b$. That is, if a is less than b, they return -1; if a is greater than b, they return 1; and if a and b are equal they return 0. If either argument is NaN they return 1, but you should not rely on this; if NaN is a possibility, use one of the higher-level comparison functions.

`int __unordsf2 (float a, float b)` [Runtime Function]
`int __unorddf2 (double a, double b)` [Runtime Function]
`int __unordtf2 (long double a, long double b)` [Runtime Function]
These functions return a nonzero value if either argument is NaN, otherwise 0.

There is also a complete group of higher level functions which correspond directly to comparison operators. They implement the ISO C semantics for floating-point comparisons, taking NaN into account. Pay careful attention to the return values defined for each set. Under the hood, all of these routines are implemented as

```
if (__unordXf2 (a, b))
  return E;
return __cmpXf2 (a, b);
```

where E is a constant chosen to give the proper behavior for NaN. Thus, the meaning of the return value is different for each set. Do not rely on this implementation; only the semantics documented below are guaranteed.

`int __eqsf2 (`*float* `a`, *float* `b`)` [Runtime Function]
`int __eqdf2 (`*double* `a`, *double* `b`)` [Runtime Function]
`int __eqtf2 (`*long double* `a`, *long double* `b`)` [Runtime Function]
These functions return zero if neither argument is NaN, and a and b are equal.

`int __nesf2 (`*float* `a`, *float* `b`)` [Runtime Function]
`int __nedf2 (`*double* `a`, *double* `b`)` [Runtime Function]
`int __netf2 (`*long double* `a`, *long double* `b`)` [Runtime Function]
These functions return a nonzero value if either argument is NaN, or if a and b are unequal.

`int __gesf2 (`*float* `a`, *float* `b`)` [Runtime Function]
`int __gedf2 (`*double* `a`, *double* `b`)` [Runtime Function]
`int __getf2 (`*long double* `a`, *long double* `b`)` [Runtime Function]
These functions return a value greater than or equal to zero if neither argument is NaN, and a is greater than or equal to b.

`int __ltsf2 (`*float* `a`, *float* `b`)` [Runtime Function]
`int __ltdf2 (`*double* `a`, *double* `b`)` [Runtime Function]
`int __lttf2 (`*long double* `a`, *long double* `b`)` [Runtime Function]
These functions return a value less than zero if neither argument is NaN, and a is strictly less than b.

`int __lesf2 (`*float* `a`, *float* `b`)` [Runtime Function]
`int __ledf2 (`*double* `a`, *double* `b`)` [Runtime Function]
`int __letf2 (`*long double* `a`, *long double* `b`)` [Runtime Function]
These functions return a value less than or equal to zero if neither argument is NaN, and a is less than or equal to b.

`int __gtsf2 (`*float* `a`, *float* `b`)` [Runtime Function]
`int __gtdf2 (`*double* `a`, *double* `b`)` [Runtime Function]
`int __gttf2 (`*long double* `a`, *long double* `b`)` [Runtime Function]
These functions return a value greater than zero if neither argument is NaN, and a is strictly greater than b.

4.3 Language-independent routines for exception handling

document me!

```
_Unwind_DeleteException
_Unwind_Find_FDE
_Unwind_ForcedUnwind
_Unwind_GetGR
_Unwind_GetIP
_Unwind_GetLanguageSpecificData
_Unwind_GetRegionStart
_Unwind_GetTextRelBase
_Unwind_GetDataRelBase
```

```
_Unwind_RaiseException
_Unwind_Resume
_Unwind_SetGR
_Unwind_SetIP
_Unwind_FindEnclosingFunction
_Unwind_SjLj_Register
_Unwind_SjLj_Unregister
_Unwind_SjLj_RaiseException
_Unwind_SjLj_ForcedUnwind
_Unwind_SjLj_Resume
__deregister_frame
__deregister_frame_info
__deregister_frame_info_bases
__register_frame
__register_frame_info
__register_frame_info_bases
__register_frame_info_table
__register_frame_info_table_bases
__register_frame_table
```

4.4 Miscellaneous runtime library routines

4.4.1 Cache control functions

`void __clear_cache` (*char* ***beg***, *char* ***end***) [Runtime Function]

This function clears the instruction cache between *beg* and *end*.

5 Language Front Ends in GCC

The interface to front ends for languages in GCC, and in particular the `tree` structure (see Chapter 9 [Trees], page 61), was initially designed for C, and many aspects of it are still somewhat biased towards C and C-like languages. It is, however, reasonably well suited to other procedural languages, and front ends for many such languages have been written for GCC.

Writing a compiler as a front end for GCC, rather than compiling directly to assembler or generating C code which is then compiled by GCC, has several advantages:

- GCC front ends benefit from the support for many different target machines already present in GCC.
- GCC front ends benefit from all the optimizations in GCC. Some of these, such as alias analysis, may work better when GCC is compiling directly from source code then when it is compiling from generated C code.
- Better debugging information is generated when compiling directly from source code than when going via intermediate generated C code.

Because of the advantages of writing a compiler as a GCC front end, GCC front ends have also been created for languages very different from those for which GCC was designed, such as the declarative logic/functional language Mercury. For these reasons, it may also be useful to implement compilers created for specialized purposes (for example, as part of a research project) as GCC front ends.

6 Source Tree Structure and Build System

This chapter describes the structure of the GCC source tree, and how GCC is built. The user documentation for building and installing GCC is in a separate manual (`http://gcc.gnu.org/install/`), with which it is presumed that you are familiar.

6.1 Configure Terms and History

The configure and build process has a long and colorful history, and can be confusing to anyone who doesn't know why things are the way they are. While there are other documents which describe the configuration process in detail, here are a few things that everyone working on GCC should know.

There are three system names that the build knows about: the machine you are building on (*build*), the machine that you are building for (*host*), and the machine that GCC will produce code for (*target*). When you configure GCC, you specify these with '`--build=`', '`--host=`', and '`--target=`'.

Specifying the host without specifying the build should be avoided, as `configure` may (and once did) assume that the host you specify is also the build, which may not be true.

If build, host, and target are all the same, this is called a *native*. If build and host are the same but target is different, this is called a *cross*. If build, host, and target are all different this is called a *canadian* (for obscure reasons dealing with Canada's political party and the background of the person working on the build at that time). If host and target are the same, but build is different, you are using a cross-compiler to build a native for a different system. Some people call this a *host-x-host*, *crossed native*, or *cross-built native*. If build and target are the same, but host is different, you are using a cross compiler to build a cross compiler that produces code for the machine you're building on. This is rare, so there is no common way of describing it. There is a proposal to call this a *crossback*.

If build and host are the same, the GCC you are building will also be used to build the target libraries (like `libstdc++`). If build and host are different, you must have already build and installed a cross compiler that will be used to build the target libraries (if you configured with '`--target=foo-bar`', this compiler will be called `foo-bar-gcc`).

In the case of target libraries, the machine you're building for is the machine you specified with '`--target`'. So, build is the machine you're building on (no change there), host is the machine you're building for (the target libraries are built for the target, so host is the target you specified), and target doesn't apply (because you're not building a compiler, you're building libraries). The configure/make process will adjust these variables as needed. It also sets `$with_cross_host` to the original '`--host`' value in case you need it.

The `libiberty` support library is built up to three times: once for the host, once for the target (even if they are the same), and once for the build if build and host are different. This allows it to be used by all programs which are generated in the course of the build process.

6.2 Top Level Source Directory

The top level source directory in a GCC distribution contains several files and directories that are shared with other software distributions such as that of GNU Binutils. It also contains several subdirectories that contain parts of GCC and its runtime libraries:

'`boehm-gc`'
: The Boehm conservative garbage collector, used as part of the Java runtime library.

'`contrib`'
: Contributed scripts that may be found useful in conjunction with GCC. One of these, '`contrib/texi2pod.pl`', is used to generate man pages from Texinfo manuals as part of the GCC build process.

'`fastjar`'
: An implementation of the `jar` command, used with the Java front end.

'`gcc`'
: The main sources of GCC itself (except for runtime libraries), including optimizers, support for different target architectures, language front ends, and testsuites. See Section 6.3 [The '`gcc`' Subdirectory], page 20, for details.

'`include`'
: Headers for the `libiberty` library.

'`libada`'
: The Ada runtime library.

'`libcpp`'
: The C preprocessor library.

'`libgfortran`'
: The Fortran runtime library.

'`libffi`'
: The `libffi` library, used as part of the Java runtime library.

'`libiberty`'
: The `libiberty` library, used for portability and for some generally useful data structures and algorithms. See section "Introduction" in GNU *libiberty*, for more information about this library.

'`libjava`'
: The Java runtime library.

'`libmudflap`'
: The `libmudflap` library, used for instrumenting pointer and array dereferencing operations.

'`libobjc`'
: The Objective-C and Objective-C++ runtime library.

'`libstdc++-v3`'
: The C++ runtime library.

'`maintainer-scripts`'
: Scripts used by the `gccadmin` account on `gcc.gnu.org`.

'`zlib`'
: The `zlib` compression library, used by the Java front end and as part of the Java runtime library.

The build system in the top level directory, including how recursion into subdirectories works and how building runtime libraries for multilibs is handled, is documented in a separate manual, included with GNU Binutils. See section "GNU configure and build system" in *The GNU configure and build system*, for details.

6.3 The '`gcc`' Subdirectory

The '`gcc`' directory contains many files that are part of the C sources of GCC, other files used as part of the configuration and build process, and subdirectories including documentation and a testsuite. The files that are sources of GCC are documented in a separate chapter. See Chapter 8 [Passes and Files of the Compiler], page 49.

6.3.1 Subdirectories of 'gcc'

The 'gcc' directory contains the following subdirectories:

'*language*'
: Subdirectories for various languages. Directories containing a file 'config-lang.in' are language subdirectories. The contents of the subdirectories 'cp' (for C++), 'objc' (for Objective-C) and 'objcp' (for Objective-C++) are documented in this manual (see Chapter 8 [Passes and Files of the Compiler], page 49); those for other languages are not. See Section 6.3.8 [Anatomy of a Language Front End], page 28, for details of the files in these directories.

'config'
: Configuration files for supported architectures and operating systems. See Section 6.3.9 [Anatomy of a Target Back End], page 32, for details of the files in this directory.

'doc'
: Texinfo documentation for GCC, together with automatically generated man pages and support for converting the installation manual to HTML. See Section 6.3.7 [Documentation], page 26.

'fixinc'
: The support for fixing system headers to work with GCC. See 'fixinc/README' for more information. The headers fixed by this mechanism are installed in '*libsubdir*/include'. Along with those headers, 'README-fixinc' is also installed, as '*libsubdir*/include/README'.

'ginclude'
: System headers installed by GCC, mainly those required by the C standard of freestanding implementations. See Section 6.3.6 [Headers Installed by GCC], page 25, for details of when these and other headers are installed.

'intl'
: GNU libintl, from GNU gettext, for systems which do not include it in libc. Properly, this directory should be at top level, parallel to the 'gcc' directory.

'po'
: Message catalogs with translations of messages produced by GCC into various languages, '*language*.po'. This directory also contains 'gcc.pot', the template for these message catalogues, 'exgettext', a wrapper around gettext to extract the messages from the GCC sources and create 'gcc.pot', which is run by 'make gcc.pot', and 'EXCLUDES', a list of files from which messages should not be extracted.

'testsuite'
: The GCC testsuites (except for those for runtime libraries). See Section 6.4 [Testsuites], page 33.

6.3.2 Configuration in the 'gcc' Directory

The 'gcc' directory is configured with an Autoconf-generated script 'configure'. The 'configure' script is generated from 'configure.ac' and 'aclocal.m4'. From the files 'configure.ac' and 'acconfig.h', Autoheader generates the file 'config.in'. The file 'cstamp-h.in' is used as a timestamp.

6.3.2.1 Scripts Used by 'configure'

'configure' uses some other scripts to help in its work:

- The standard GNU 'config.sub' and 'config.guess' files, kept in the top level directory, are used. FIXME: when is the 'config.guess' file in the 'gcc' directory (that just calls the top level one) used?
- The file 'config.gcc' is used to handle configuration specific to the particular target machine. The file 'config.build' is used to handle configuration specific to the particular build machine. The file 'config.host' is used to handle configuration specific to the particular host machine. (In general, these should only be used for features that cannot reasonably be tested in Autoconf feature tests.) See Section 6.3.2.2 [The 'config.build'; 'config.host'; and 'config.gcc' Files], page 22, for details of the contents of these files.
- Each language subdirectory has a file '*language*/config-lang.in' that is used for front-end-specific configuration. See Section 6.3.8.2 [The Front End 'config-lang.in' File], page 31, for details of this file.
- A helper script 'configure.frag' is used as part of creating the output of 'configure'.

6.3.2.2 The 'config.build'; 'config.host'; and 'config.gcc' Files

The 'config.build' file contains specific rules for particular systems which GCC is built on. This should be used as rarely as possible, as the behavior of the build system can always be detected by autoconf.

The 'config.host' file contains specific rules for particular systems which GCC will run on. This is rarely needed.

The 'config.gcc' file contains specific rules for particular systems which GCC will generate code for. This is usually needed.

Each file has a list of the shell variables it sets, with descriptions, at the top of the file.

FIXME: document the contents of these files, and what variables should be set to control build, host and target configuration.

6.3.2.3 Files Created by configure

Here we spell out what files will be set up by 'configure' in the 'gcc' directory. Some other files are created as temporary files in the configuration process, and are not used in the subsequent build; these are not documented.

- 'Makefile' is constructed from 'Makefile.in', together with the host and target fragments (see Chapter 16 [Makefile Fragments], page 409) 't-*target*' and 'x-*host*' from 'config', if any, and language Makefile fragments '*language*/Make-lang.in'.
- 'auto-host.h' contains information about the host machine determined by 'configure'. If the host machine is different from the build machine, then 'auto-build.h' is also created, containing such information about the build machine.
- 'config.status' is a script that may be run to recreate the current configuration.
- 'configargs.h' is a header containing details of the arguments passed to 'configure' to configure GCC, and of the thread model used.
- 'cstamp-h' is used as a timestamp.

- '`fixinc/Makefile`' is constructed from '`fixinc/Makefile.in`'.
- '`gccbug`', a script for reporting bugs in GCC, is constructed from '`gccbug.in`'.
- '`intl/Makefile`' is constructed from '`intl/Makefile.in`'.
- '`mklibgcc`', a shell script to create a Makefile to build libgcc, is constructed from '`mklibgcc.in`'.
- If a language '`config-lang.in`' file (see Section 6.3.8.2 [The Front End '`config-lang.in`' File], page 31) sets `outputs`, then the files listed in `outputs` there are also generated.

The following configuration headers are created from the Makefile, using '`mkconfig.sh`', rather than directly by '`configure`'. '`config.h`', '`bconfig.h`' and '`tconfig.h`' all contain the '`xm-`*`machine`*`.h`' header, if any, appropriate to the host, build and target machines respectively, the configuration headers for the target, and some definitions; for the host and build machines, these include the autoconfigured headers generated by '`configure`'. The other configuration headers are determined by '`config.gcc`'. They also contain the typedefs for `rtx`, `rtvec` and `tree`.

- '`config.h`', for use in programs that run on the host machine.
- '`bconfig.h`', for use in programs that run on the build machine.
- '`tconfig.h`', for use in programs and libraries for the target machine.
- '`tm_p.h`', which includes the header '*`machine`*`-protos.h`' that contains prototypes for functions in the target '`.c`' file. FIXME: why is such a separate header necessary?

6.3.3 Build System in the 'gcc' Directory

FIXME: describe the build system, including what is built in what stages. Also list the various source files that are used in the build process but aren't source files of GCC itself and so aren't documented below (see Chapter 8 [Passes], page 49).

6.3.4 Makefile Targets

`all` This is the default target. Depending on what your build/host/target configuration is, it coordinates all the things that need to be built.

`doc` Produce info-formatted documentation and man pages. Essentially it calls '`make man`' and '`make info`'.

`dvi` Produce DVI-formatted documentation.

`html` Produce HTML-formatted documentation.

`man` Generate man pages.

`info` Generate info-formatted pages.

`mostlyclean`
Delete the files made while building the compiler.

`clean` That, and all the other files built by '`make all`'.

`distclean`
That, and all the files created by `configure`.

`maintainer-clean`
: Distclean plus any file that can be generated from other files. Note that additional tools may be required beyond what is normally needed to build gcc.

`srcextra` Generates files in the source directory that do not exist in CVS but should go into a release tarball. One example is '`gcc/java/parse.c`' which is generated from the CVS source file '`gcc/java/parse.y`'.

`srcinfo`
`srcman` Copies the info-formatted and manpage documentation into the source directory usually for the purpose of generating a release tarball.

`install` Installs gcc.

`uninstall`
: Deletes installed files.

`check` Run the testsuite. This creates a '`testsuite`' subdirectory that has various '`.sum`' and '`.log`' files containing the results of the testing. You can run subsets with, for example, '`make check-gcc`'. You can specify specific tests by setting RUNTESTFLAGS to be the name of the '`.exp`' file, optionally followed by (for some tests) an equals and a file wildcard, like:

```
make check-gcc RUNTESTFLAGS="execute.exp=19980413-*"
```

Note that running the testsuite may require additional tools be installed, such as TCL or dejagnu.

`bootstrap`
: Builds GCC three times—once with the native compiler, once with the native-built compiler it just built, and once with the compiler it built the second time. In theory, the last two should produce the same results, which '`make compare`' can check. Each step of this process is called a "stage", and the results of each stage N ($N = 1 \ldots 3$) are copied to a subdirectory '`stage`*N*`/`'.

`bootstrap-lean`
: Like `bootstrap`, except that the various stages are removed once they're no longer needed. This saves disk space.

`bubblestrap`
: This incrementally rebuilds each of the three stages, one at a time. It does this by "bubbling" the stages up from their subdirectories (if they had been built previously), rebuilding them, and copying them back to their subdirectories. This will allow you to, for example, continue a bootstrap after fixing a bug which causes the stage2 build to crash.

`quickstrap`
: Rebuilds the most recently built stage. Since each stage requires special invocation, using this target means you don't have to keep track of which stage you're on or what invocation that stage needs.

`cleanstrap`
: Removed everything ('`make clean`') and rebuilds ('`make bootstrap`').

`restrap` Like `cleanstrap`, except that the process starts from the first stage build, not from scratch.

`stage`*N* (*N* = `1...4`)
: For each stage, moves the appropriate files to the '`stage`*N*' subdirectory.

`unstage`*N* (*N* = `1...4`)
: Undoes the corresponding `stage`*N*.

`restage`*N* (*N* = `1...4`)
: Undoes the corresponding `stage`*N* and rebuilds it with the appropriate flags.

`compare`
: Compares the results of stages 2 and 3. This ensures that the compiler is running properly, since it should produce the same object files regardless of how it itself was compiled.

`profiledbootstrap`
: Builds a compiler with profiling feedback information. For more information, see section "Building with profile feedback" in *Installing GCC*. This is actually a target in the top-level directory, which then recurses into the '`gcc`' subdirectory multiple times.

6.3.5 Library Source Files and Headers under the '`gcc`' Directory

FIXME: list here, with explanation, all the C source files and headers under the '`gcc`' directory that aren't built into the GCC executable but rather are part of runtime libraries and object files, such as '`crtstuff.c`' and '`unwind-dw2.c`'. See Section 6.3.6 [Headers Installed by GCC], page 25, for more information about the '`ginclude`' directory.

6.3.6 Headers Installed by GCC

In general, GCC expects the system C library to provide most of the headers to be used with it. However, GCC will fix those headers if necessary to make them work with GCC, and will install some headers required of freestanding implementations. These headers are installed in '*`libsubdir`*`/include`'. Headers for non-C runtime libraries are also installed by GCC; these are not documented here. (FIXME: document them somewhere.)

Several of the headers GCC installs are in the '`ginclude`' directory. These headers, '`iso646.h`', '`stdarg.h`', '`stdbool.h`', and '`stddef.h`', are installed in '*`libsubdir`*`/include`', unless the target Makefile fragment (see Section 16.1 [Target Fragment], page 409) overrides this by setting `USER_H`.

In addition to these headers and those generated by fixing system headers to work with GCC, some other headers may also be installed in '*`libsubdir`*`/include`'. '`config.gcc`' may set `extra_headers`; this specifies additional headers under '`config`' to be installed on some systems.

GCC installs its own version of `<float.h>`, from '`ginclude/float.h`'. This is done to cope with command-line options that change the representation of floating point numbers.

GCC also installs its own version of `<limits.h>`; this is generated from '`glimits.h`', together with '`limitx.h`' and '`limity.h`' if the system also has its own version of `<limits.h>`. (GCC provides its own header because it is required of ISO C freestanding implementations, but needs to include the system header from its own header as well because other standards such as POSIX specify additional values to be defined in `<limits.h>`.) The system's `<limits.h>` header is used via '*`libsubdir`*`/include/syslimits.h`', which is copied from '`gsyslimits.h`' if it does not need fixing to work with GCC; if it needs fixing, '`syslimits.h`' is the fixed copy.

6.3.7 Building Documentation

The main GCC documentation is in the form of manuals in Texinfo format. These are installed in Info format, and DVI versions may be generated by '`make dvi`' and HTML versions may be generated by `make html`. In addition, some man pages are generated from the Texinfo manuals, there are some other text files with miscellaneous documentation, and runtime libraries have their own documentation outside the '`gcc`' directory. FIXME: document the documentation for runtime libraries somewhere.

6.3.7.1 Texinfo Manuals

The manuals for GCC as a whole, and the C and C++ front ends, are in files '`doc/*.texi`'. Other front ends have their own manuals in files '*`language`*`/*.texi`'. Common files '`doc/include/*.texi`' are provided which may be included in multiple manuals; the following files are in '`doc/include`':

'`fdl.texi`'
: The GNU Free Documentation License.

'`funding.texi`'
: The section "Funding Free Software".

'`gcc-common.texi`'
: Common definitions for manuals.

'`gpl.texi`'
: The GNU General Public License.

'`texinfo.tex`'
: A copy of '`texinfo.tex`' known to work with the GCC manuals.

DVI formatted manuals are generated by '`make dvi`', which uses `texi2dvi` (via the Makefile macro `$(TEXI2DVI)`). HTML formatted manuals are generated by `make html`. Info manuals are generated by '`make info`' (which is run as part of a bootstrap); this generates the manuals in the source directory, using `makeinfo` via the Makefile macro `$(MAKEINFO)`, and they are included in release distributions.

Manuals are also provided on the GCC web site, in both HTML and PostScript forms. This is done via the script '`maintainer-scripts/update_web_docs`'. Each manual to be provided online must be listed in the definition of `MANUALS` in that file; a file '*`name`*`.texi`' must only appear once in the source tree, and the output manual must have the same name as the source file. (However, other Texinfo files, included in manuals but not themselves the root files of manuals, may have names that appear more than once in the source tree.) The manual file '*`name`*`.texi`' should only include other files in its own directory or in '`doc/include`'. HTML manuals will be generated by '`makeinfo --html`' and PostScript manuals by `texi2dvi` and `dvips`. All Texinfo files that are parts of manuals must be checked into CVS, even if they are generated files, for the generation of online manuals to work.

The installation manual, '`doc/install.texi`', is also provided on the GCC web site. The HTML version is generated by the script '`doc/install.texi2html`'.

6.3.7.2 Man Page Generation

Because of user demand, in addition to full Texinfo manuals, man pages are provided which contain extracts from those manuals. These man pages are generated from the Texinfo manuals using '`contrib/texi2pod.pl`' and `pod2man`. (The man page for `g++`, '`cp/g++.1`', just contains a '`.so`' reference to '`gcc.1`', but all the other man pages are generated from Texinfo manuals.)

Because many systems may not have the necessary tools installed to generate the man pages, they are only generated if the '`configure`' script detects that recent enough tools are installed, and the Makefiles allow generating man pages to fail without aborting the build. Man pages are also included in release distributions. They are generated in the source directory.

Magic comments in Texinfo files starting '`@c man`' control what parts of a Texinfo file go into a man page. Only a subset of Texinfo is supported by '`texi2pod.pl`', and it may be necessary to add support for more Texinfo features to this script when generating new man pages. To improve the man page output, some special Texinfo macros are provided in '`doc/include/gcc-common.texi`' which '`texi2pod.pl`' understands:

`@gcctabopt`
: Use in the form '`@table @gcctabopt`' for tables of options, where for printed output the effect of '`@code`' is better than that of '`@option`' but for man page output a different effect is wanted.

`@gccoptlist`
: Use for summary lists of options in manuals.

`@gol`
: Use at the end of each line inside '`@gccoptlist`'. This is necessary to avoid problems with differences in how the '`@gccoptlist`' macro is handled by different Texinfo formatters.

FIXME: describe the '`texi2pod.pl`' input language and magic comments in more detail.

6.3.7.3 Miscellaneous Documentation

In addition to the formal documentation that is installed by GCC, there are several other text files with miscellaneous documentation:

'`ABOUT-GCC-NLS`'
: Notes on GCC's Native Language Support. FIXME: this should be part of this manual rather than a separate file.

'`ABOUT-NLS`'
: Notes on the Free Translation Project.

'`COPYING`' The GNU General Public License.

'`COPYING.LIB`'
: The GNU Lesser General Public License.

'`*ChangeLog*`'
'`*/ChangeLog*`'
: Change log files for various parts of GCC.

'`LANGUAGES`'
: Details of a few changes to the GCC front-end interface. FIXME: the information in this file should be part of general documentation of the front-end interface in this manual.

'`ONEWS`'
: Information about new features in old versions of GCC. (For recent versions, the information is on the GCC web site.)

'`README.Portability`'
: Information about portability issues when writing code in GCC. FIXME: why isn't this part of this manual or of the GCC Coding Conventions?

'`SERVICE`'
: A pointer to the GNU Service Directory.

FIXME: document such files in subdirectories, at least '`config`', '`cp`', '`objc`', '`testsuite`'.

6.3.8 Anatomy of a Language Front End

A front end for a language in GCC has the following parts:

- A directory '*`language`*' under '`gcc`' containing source files for that front end. See Section 6.3.8.1 [The Front End '*`language`*' Directory], page 29, for details.
- A mention of the language in the list of supported languages in '`gcc/doc/install.texi`'.
- A mention of the name under which the language's runtime library is recognized by '`--enable-shared=`*`package`*' in the documentation of that option in '`gcc/doc/install.texi`'.
- A mention of any special prerequisites for building the front end in the documentation of prerequisites in '`gcc/doc/install.texi`'.
- Details of contributors to that front end in '`gcc/doc/contrib.texi`'. If the details are in that front end's own manual then there should be a link to that manual's list in '`contrib.texi`'.
- Information about support for that language in '`gcc/doc/frontends.texi`'.
- Information about standards for that language, and the front end's support for them, in '`gcc/doc/standards.texi`'. This may be a link to such information in the front end's own manual.
- Details of source file suffixes for that language and '`-x` *`lang`*' options supported, in '`gcc/doc/invoke.texi`'.
- Entries in `default_compilers` in '`gcc.c`' for source file suffixes for that language.
- Preferably testsuites, which may be under '`gcc/testsuite`' or runtime library directories. FIXME: document somewhere how to write testsuite harnesses.
- Probably a runtime library for the language, outside the '`gcc`' directory. FIXME: document this further.
- Details of the directories of any runtime libraries in '`gcc/doc/sourcebuild.texi`'.

If the front end is added to the official GCC CVS repository, the following are also necessary:

- At least one Bugzilla component for bugs in that front end and runtime libraries. This category needs to be mentioned in '`gcc/gccbug.in`', as well as being added to the Bugzilla database.

- Normally, one or more maintainers of that front end listed in '`MAINTAINERS`'.
- Mentions on the GCC web site in '`index.html`' and '`frontends.html`', with any relevant links on '`readings.html`'. (Front ends that are not an official part of GCC may also be listed on '`frontends.html`', with relevant links.)
- A news item on '`index.html`', and possibly an announcement on the `gcc-announce@gcc.gnu.org` mailing list.
- The front end's manuals should be mentioned in '`maintainer-scripts/update_web_docs`' (see Section 6.3.7.1 [Texinfo Manuals], page 26) and the online manuals should be linked to from '`onlinedocs/index.html`'.
- Any old releases or CVS repositories of the front end, before its inclusion in GCC, should be made available on the GCC FTP site `ftp://gcc.gnu.org/pub/gcc/old-releases/`.
- The release and snapshot script '`maintainer-scripts/gcc_release`' should be updated to generate appropriate tarballs for this front end. The associated '`maintainer-scripts/snapshot-README`' and '`maintainer-scripts/snapshot-index.html`' files should be updated to list the tarballs and diffs for this front end.
- If this front end includes its own version files that include the current date, '`maintainer-scripts/update_version`' should be updated accordingly.
- '`CVSROOT/modules`' in the GCC CVS repository should be updated.

6.3.8.1 The Front End '*language*' Directory

A front end '*language*' directory contains the source files of that front end (but not of any runtime libraries, which should be outside the '`gcc`' directory). This includes documentation, and possibly some subsidiary programs build alongside the front end. Certain files are special and other parts of the compiler depend on their names:

'`config-lang.in`'
: This file is required in all language subdirectories. See Section 6.3.8.2 [The Front End '`config-lang.in`' File], page 31, for details of its contents

'`Make-lang.in`'
: This file is required in all language subdirectories. It contains targets *`lang.hook`* (where *`lang`* is the setting of `language` in '`config-lang.in`') for the following values of *`hook`*, and any other Makefile rules required to build those targets (which may if necessary use other Makefiles specified in `outputs` in '`config-lang.in`', although this is deprecated). Some hooks are defined by using a double-colon rule for *`hook`*, rather than by using a target of form *`lang.hook`*. These hooks are called "double-colon hooks" below. It also adds any testsuite targets that can use the standard rule in '`gcc/Makefile.in`' to the variable `lang_checks`.

  `all.build`
  `all.cross`
  `start.encap`
  `rest.encap`
  FIXME: exactly what goes in each of these targets?

`tags` Build an `etags` '`TAGS`' file in the language subdirectory in the source tree.

`info` Build info documentation for the front end, in the build directory. This target is only called by '`make bootstrap`' if a suitable version of `makeinfo` is available, so does not need to check for this, and should fail if an error occurs.

`dvi` Build DVI documentation for the front end, in the build directory. This should be done using `$(TEXI2DVI)`, with appropriate '`-I`' arguments pointing to directories of included files. This hook is a double-colon hook.

`html` Build HTML documentation for the front end, in the build directory.

`man` Build generated man pages for the front end from Texinfo manuals (see Section 6.3.7.2 [Man Page Generation], page 27), in the build directory. This target is only called if the necessary tools are available, but should ignore errors so as not to stop the build if errors occur; man pages are optional and the tools involved may be installed in a broken way.

`install-normal`
FIXME: what is this target for?

`install-common`
Install everything that is part of the front end, apart from the compiler executables listed in `compilers` in '`config-lang.in`'.

`install-info`
Install info documentation for the front end, if it is present in the source directory. This target should have dependencies on info files that should be installed. This hook is a double-colon hook.

`install-man`
Install man pages for the front end. This target should ignore errors.

`srcextra` Copies its dependencies into the source directory. This generally should be used for generated files such as Bison output files which are not present in CVS, but should be included in any release tarballs. This target will be executed during a bootstrap if '`--enable-generated-files-in-srcdir`' was specified as a '`configure`' option.

`srcinfo`
`srcman` Copies its dependencies into the source directory. These targets will be executed during a bootstrap if '`--enable-generated-files-in-srcdir`' was specified as a '`configure`' option.

`uninstall`
: Uninstall files installed by installing the compiler. This is currently documented not to be supported, so the hook need not do anything.

`mostlyclean`
`clean`
`distclean`
`maintainer-clean`
: The language parts of the standard GNU '`*clean`' targets. See section "Standard Targets for Users" in *GNU Coding Standards*, for details of the standard targets. For GCC, `maintainer-clean` should delete all generated files in the source directory that are not checked into CVS, but should not delete anything checked into CVS.

`stage1`
`stage2`
`stage3`
`stage4`
`stageprofile`
`stagefeedback`
: Move to the stage directory files not included in `stagestuff` in '`config-lang.in`' or otherwise moved by the main '`Makefile`'.

'`lang.opt`'
: This file registers the set of switches that the front end accepts on the command line, and their '`--help`' text. See Chapter 7 [Options], page 45.

'`lang-specs.h`'
: This file provides entries for `default_compilers` in '`gcc.c`' which override the default of giving an error that a compiler for that language is not installed.

'*`language`*`-tree.def`'
: This file, which need not exist, defines any language-specific tree codes.

6.3.8.2 The Front End '`config-lang.in`' File

Each language subdirectory contains a '`config-lang.in`' file. In addition the main directory contains '`c-config-lang.in`', which contains limited information for the C language. This file is a shell script that may define some variables describing the language:

`language`
: This definition must be present, and gives the name of the language for some purposes such as arguments to '`--enable-languages`'.

`lang_requires`
: If defined, this variable lists (space-separated) language front ends other than C that this front end requires to be enabled (with the names given being their `language` settings). For example, the Java front end depends on the C++ front end, so sets '`lang_requires=c++`'.

`target_libs`
: If defined, this variable lists (space-separated) targets in the top level '`Makefile`' to build the runtime libraries for this language, such as `target-libobjc`.

`lang_dirs`
: If defined, this variable lists (space-separated) top level directories (parallel to '`gcc`'), apart from the runtime libraries, that should not be configured if this front end is not built.

`build_by_default`
: If defined to '`no`', this language front end is not built unless enabled in a '`--enable-languages`' argument. Otherwise, front ends are built by default, subject to any special logic in '`configure.ac`' (as is present to disable the Ada front end if the Ada compiler is not already installed).

`boot_language`
: If defined to '`yes`', this front end is built in stage 1 of the bootstrap. This is only relevant to front ends written in their own languages.

`compilers`
: If defined, a space-separated list of compiler executables that will be run by the driver. The names here will each end with '`\$(exeext)`'.

`stagestuff`
: If defined, a space-separated list of files that should be moved to the '`stage`*n*' directories in each stage of bootstrap.

`outputs`
: If defined, a space-separated list of files that should be generated by '`configure`' substituting values in them. This mechanism can be used to create a file '*language*`/Makefile`' from '*language*`/Makefile.in`', but this is deprecated, building everything from the single '`gcc/Makefile`' is preferred.

`gtfiles`
: If defined, a space-separated list of files that should be scanned by gengtype.c to generate the garbage collection tables and routines for this language. This excludes the files that are common to all front ends. See Chapter 19 [Type Information], page 417.

`need_gmp`
: If defined to '`yes`', this frontend requires the GMP library. Enables configure tests for GMP, which set `GMPLIBS` and `GMPINC` appropriately.

6.3.9 Anatomy of a Target Back End

A back end for a target architecture in GCC has the following parts:

- A directory '*machine*' under '`gcc/config`', containing a machine description '*machine*`.md`' file (see Chapter 13 [Machine Descriptions], page 179), header files '*machine*`.h`' and '*machine*`-protos.h`' and a source file '*machine*`.c`' (see Chapter 14 [Target Description Macros and Functions], page 267), possibly a target Makefile fragment '`t-`*machine*' (see Section 16.1 [The Target Makefile Fragment], page 409), and maybe some other files. The names of these files may be changed from the defaults given by explicit specifications in '`config.gcc`'.
- If necessary, a file '*machine*`-modes.def`' in the '*machine*' directory, containing additional machine modes to represent condition codes. See Section 14.14 [Condition Code], page 338, for further details.
- An optional '*machine*`.opt`' file in the '*machine*' directory, containing a list of target-specific options. You can also add other option files using the `extra_options` variable in '`config.gcc`'. See Chapter 7 [Options], page 45.

- Entries in '`config.gcc`' (see Section 6.3.2.2 [The '`config.gcc`' File], page 22) for the systems with this target architecture.
- Documentation in '`gcc/doc/invoke.texi`' for any command-line options supported by this target (see Section 14.3 [Run-time Target Specification], page 275). This means both entries in the summary table of options and details of the individual options.
- Documentation in '`gcc/doc/extend.texi`' for any target-specific attributes supported (see Section 14.23 [Defining target-specific uses of `__attribute__`], page 387), including where the same attribute is already supported on some targets, which are enumerated in the manual.
- Documentation in '`gcc/doc/extend.texi`' for any target-specific pragmas supported.
- Documentation in '`gcc/doc/extend.texi`' of any target-specific built-in functions supported.
- Documentation in '`gcc/doc/extend.texi`' of any target-specific format checking styles supported.
- Documentation in '`gcc/doc/md.texi`' of any target-specific constraint letters (see Section 13.8.5 [Constraints for Particular Machines], page 198).
- A note in '`gcc/doc/contrib.texi`' under the person or people who contributed the target support.
- Entries in '`gcc/doc/install.texi`' for all target triplets supported with this target architecture, giving details of any special notes about installation for this target, or saying that there are no special notes if there are none.
- Possibly other support outside the '`gcc`' directory for runtime libraries. FIXME: reference docs for this. The libstdc++ porting manual needs to be installed as info for this to work, or to be a chapter of this manual.

If the back end is added to the official GCC CVS repository, the following are also necessary:

- An entry for the target architecture in '`readings.html`' on the GCC web site, with any relevant links.
- Details of the properties of the back end and target architecture in '`backends.html`' on the GCC web site.
- A news item about the contribution of support for that target architecture, in '`index.html`' on the GCC web site.
- Normally, one or more maintainers of that target listed in '`MAINTAINERS`'. Some existing architectures may be unmaintained, but it would be unusual to add support for a target that does not have a maintainer when support is added.

6.4 Testsuites

GCC contains several testsuites to help maintain compiler quality. Most of the runtime libraries and language front ends in GCC have testsuites. Currently only the C language testsuites are documented here; FIXME: document the others.

6.4.1 Idioms Used in Testsuite Code

In general C testcases have a trailing '`-n.c`', starting with '`-1.c`', in case other testcases with similar names are added later. If the test is a test of some well-defined feature, it

should have a name referring to that feature such as '*feature*`-1.c`'. If it does not test a well-defined feature but just happens to exercise a bug somewhere in the compiler, and a bug report has been filed for this bug in the GCC bug database, '`pr`*bug-number*`-1.c`' is the appropriate form of name. Otherwise (for miscellaneous bugs not filed in the GCC bug database), and previously more generally, test cases are named after the date on which they were added. This allows people to tell at a glance whether a test failure is because of a recently found bug that has not yet been fixed, or whether it may be a regression, but does not give any other information about the bug or where discussion of it may be found. Some other language testsuites follow similar conventions.

In the '`gcc.dg`' testsuite, it is often necessary to test that an error is indeed a hard error and not just a warning—for example, where it is a constraint violation in the C standard, which must become an error with '`-pedantic-errors`'. The following idiom, where the first line shown is line *line* of the file and the line that generates the error, is used for this:

```
/* { dg-bogus "warning" "warning in place of error" } */
/* { dg-error "regexp" "message" { target *-*-* } line } */
```

It may be necessary to check that an expression is an integer constant expression and has a certain value. To check that *E* has value *V*, an idiom similar to the following is used:

```
char x[((E) == (V) ? 1 : -1)];
```

In '`gcc.dg`' tests, `__typeof__` is sometimes used to make assertions about the types of expressions. See, for example, '`gcc.dg/c99-condexpr-1.c`'. The more subtle uses depend on the exact rules for the types of conditional expressions in the C standard; see, for example, '`gcc.dg/c99-intconst-1.c`'.

It is useful to be able to test that optimizations are being made properly. This cannot be done in all cases, but it can be done where the optimization will lead to code being optimized away (for example, where flow analysis or alias analysis should show that certain code cannot be called) or to functions not being called because they have been expanded as built-in functions. Such tests go in '`gcc.c-torture/execute`'. Where code should be optimized away, a call to a nonexistent function such as `link_failure ()` may be inserted; a definition

```
#ifndef __OPTIMIZE__
void
link_failure (void)
{
  abort ();
}
#endif
```

will also be needed so that linking still succeeds when the test is run without optimization. When all calls to a built-in function should have been optimized and no calls to the non-built-in version of the function should remain, that function may be defined as `static` to call `abort ()` (although redeclaring a function as static may not work on all targets).

All testcases must be portable. Target-specific testcases must have appropriate code to avoid causing failures on unsupported systems; unfortunately, the mechanisms for this differ by directory.

FIXME: discuss non-C testsuites here.

6.4.2 Directives used within DejaGnu tests

Test directives appear within comments in a test source file and begin with `dg-`. Some of these are defined within DegaGnu and others are local to the GCC testsuite.

The order in which test directives appear in a test can be important: directives local to GCC sometimes override information used by the DejaGnu directives, which know nothing about the GCC directives, so the DejaGnu directives must precede GCC directives.

Several test directives include selectors which are usually preceded by the keyword `target` or `xfail`. A selector is: one or more target triplets, possibly including wildcard characters; a single effective-target keyword; or a logical expression. Depending on the context, the selector specifies whether a test is skipped and reported as unsupported or is expected to fail. Use '`*-*-*`' to match any target. Effective-target keywords are defined in '`target-supports.exp`' in the GCC testsuite.

A selector expression appears within curly braces and uses a single logical operator: one of '`!`', '`&&`', or '`||`'. An operand is another selector expression, an effective-target keyword, a single target triplet, or a list of target triplets within quotes or curly braces. For example:

```
{ target { ! "hppa*-*-* ia64*-*-*" } }
{ target { powerpc*-*-* && lp64 } }
{ xfail { lp64 || vect_no_align } }
```

`{ dg-do` *`do-what-keyword`* `[{ target/xfail` *`selector`* `}] }`

do-what-keyword specifies how the test is compiled and whether it is executed. It is one of:

`preprocess`
: Compile with '`-E`' to run only the preprocessor.

`assemble`
: Compile with '`-S`' to produce an assembly code file.

`compile`
: Compile with '`-c`' to produce a relocatable object file.

`link`
: Compile, assemble, and link to produce an executable file.

`run`
: Produce and run an executable file, which is expected to return an exit code of 0.

The default is `compile`. That can be overridden for a set of tests by redefining `dg-do-what-default` within the `.exp` file for those tests.

If the directive includes the optional '`{ target` *`selector`* `}`' then the test is skipped unless the target system is included in the list of target triplets or matches the effective-target keyword.

If the directive includes the optional '`{ xfail` *`selector`* `}`' and the selector is met then the test is expected to fail. For `dg-do run`, execution is expected to fail but compilation is expected to pass.

`{ dg-options` *`options`* `[{ target` *`selector`* `}] }`

This DejaGnu directive provides a list of compiler options, to be used if the target system matches *selector*, that replace the default options used for this set of tests.

`{ dg-skip-if` *`comment`* `{` *`selector`* `} {` *`include-opts`* `} {` *`exclude-opts`* `} }`

Skip the test if the test system is included in *selector* and if each of the options in *include-opts* is in the set of options with which the test would be compiled

and if none of the options in *exclude-opts* is in the set of options with which the test would be compiled.

Use '`"*"`' for an empty *include-opts* list and '`""`' for an empty *exclude-opts* list.

`{ dg-xfail-if` *comment* `{` *selector* `} { include-opts } { exclude-opts } }`
Expect the test to fail if the conditions (which are the same as for `dg-skip-if`) are met.

`{ dg-require-`*support* `args }`
Skip the test if the target does not provide the required support; see '`gcc-dg.exp`' in the GCC testsuite for the actual directives. These directives must appear after any `dg-do` directive in the test. They require at least one argument, which can be an empty string if the specific procedure does not examine the argument.

`{ dg-require-effective-target` *keyword* `}`
Skip the test if the test target, including current multilib flags, is not covered by the effective-target keyword. This directive must appear after any `dg-do` directive in the test.

`{ dg-error` *regexp* `[`*comment* `[{ target/xfail` *selector* `} [`*line*`] }]] }`
This DejaGnu directive appears on a source line that is expected to get an error message, or else specifies the source line associated with the message. If there is no message for that line or if the text of that message is not matched by *regexp* then the check fails and *comment* is included in the `FAIL` message. The check does not look for the string '`"error"`' unless it is part of *regexp*.

`{ dg-warning` *regexp* `[`*comment* `[{ target/xfail` *selector* `} [`*line*`] }]] }`
This DejaGnu directive appears on a source line that is expected to get a warning message, or else specifies the source line associated with the message. If there is no message for that line or if the text of that message is not matched by *regexp* then the check fails and *comment* is included in the `FAIL` message. The check does not look for the string '`"warning"`' unless it is part of *regexp*.

`{ dg-bogus` *regexp* `[`*comment* `[{ target/xfail` *selector* `} [`*line*`] }]] }`
This DejaGnu directive appears on a source line that should not get a message matching *regexp*, or else specifies the source line associated with the bogus message. It is usually used with '`xfail`' to indicate that the message is a known problem for a particular set of targets.

`{ dg-excess-errors` *comment* `[{ target/xfail` *selector* `}] }`
This DejaGnu directive indicates that the test is expected to fail due to compiler messages that are not handled by '`dg-error`', '`dg-warning`' or '`dg-bogus`'.

`{ dg-output` *regexp* `[{ target/xfail` *selector* `}] }`
This DejaGnu directive compares *regexp* to the combined output that the test executable writes to '`stdout`' and '`stderr`'.

`{ dg-prune-output` *regexp* `}`
Prune messages matching *regexp* from test output.

`{ dg-additional-files "`*`filelist`*`" }`
: Specify additional files, other than source files, that must be copied to the system where the compiler runs.

`{ dg-additional-sources "`*`filelist`*`" }`
: Specify additional source files to appear in the compile line following the main test file.

`{ dg-final { `*`local-directive`*` } }`
: This DejaGnu directive is placed within a comment anywhere in the source file and is processed after the test has been compiled and run. Multiple '`dg-final`' commands are processed in the order in which they appear in the source file.

  The GCC testsuite defines the following directives to be used within `dg-final`.

  `cleanup-coverage-files`
  : Removes coverage data files generated for this test.

  `cleanup-repo-files`
  : Removes files generated for this test for '`-frepo`'.

  `cleanup-rtl-dump `*`suffix`*
  : Removes RTL dump files generated for this test.

  `cleanup-tree-dump `*`suffix`*
  : Removes tree dump files matching *suffix* which were generated for this test.

  `cleanup-saved-temps`
  : Removes files for the current test which were kept for '`--save-temps`'.

  `scan-file `*`filename regexp`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *regexp* matches text in *filename*.

  `scan-file-not `*`filename regexp`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *regexp* does not match text in *filename*.

  `scan-hidden `*`symbol`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *symbol* is defined as a hidden symbol in the test's assembly output.

  `scan-not-hidden `*`symbol`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *symbol* is not defined as a hidden symbol in the test's assembly output.

  `scan-assembler-times `*`regex num`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *regex* is matched exactly *num* times in the test's assembler output.

  `scan-assembler `*`regex`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *regex* matches text in the test's assembler output.

  `scan-assembler-not `*`regex`*` [{ target/xfail `*`selector`*` }]`
  : Passes if *regex* does not match text in the test's assembler output.

`scan-assembler-dem` *regex* `[{ target/xfail` *selector* `}]`
: Passes if *regex* matches text in the test's demangled assembler output.

`scan-assembler-dem-not` *regex* `[{ target/xfail` *selector* `}]`
: Passes if *regex* does not match text in the test's demangled assembler output.

`scan-tree-dump-times` *regex num suffix* `[{ target/xfail` *selector* `}]`
: Passes if *regex* is found exactly *num* times in the dump file with suffix *suffix*.

`scan-tree-dump` *regex suffix* `[{ target/xfail` *selector* `}]`
: Passes if *regex* matches text in the dump file with suffix *suffix*.

`scan-tree-dump-not` *regex suffix* `[{ target/xfail` *selector* `}]`
: Passes if *regex* does not match text in the dump file with suffix *suffix*.

`scan-tree-dump-dem` *regex suffix* `[{ target/xfail` *selector* `}]`
: Passes if *regex* matches demangled text in the dump file with suffix *suffix*.

`scan-tree-dump-dem-not` *regex suffix* `[{ target/xfail` *selector* `}]`
: Passes if *regex* does not match demangled text in the dump file with suffix *suffix*.

`run-gcov` *sourcefile*
: Check line counts in `gcov` tests.

`run-gcov [branches] [calls] {` *opts sourcefile* `}`
: Check branch and/or call counts, in addition to line counts, in `gcov` tests.

6.4.3 Ada Language Testsuites

The Ada testsuite includes executable tests from the ACATS 2.5 testsuite, publicly available at `http://www.adaic.org/compilers/acats/2.5`

These tests are integrated in the GCC testsuite in the '`gcc/testsuite/ada/acats`' directory, and enabled automatically when running `make check`, assuming the Ada language has been enabled when configuring GCC.

You can also run the Ada testsuite independently, using `make check-ada`, or run a subset of the tests by specifying which chapter to run, e.g.:

```
$ make check-ada CHAPTERS="c3 c9"
```

The tests are organized by directory, each directory corresponding to a chapter of the Ada Reference Manual. So for example, c9 corresponds to chapter 9, which deals with tasking features of the language.

There is also an extra chapter called '`gcc`' containing a template for creating new executable tests.

The tests are run using two `sh` scripts: '`run_acats`' and '`run_all.sh`'. To run the tests using a simulator or a cross target, see the small customization section at the top of '`run_all.sh`'.

These tests are run using the build tree: they can be run without doing a `make install`.

6.4.4 C Language Testsuites

GCC contains the following C language testsuites, in the '`gcc/testsuite`' directory:

'`gcc.dg`' This contains tests of particular features of the C compiler, using the more modern '`dg`' harness. Correctness tests for various compiler features should go here if possible.

Magic comments determine whether the file is preprocessed, compiled, linked or run. In these tests, error and warning message texts are compared against expected texts or regular expressions given in comments. These tests are run with the options '`-ansi -pedantic`' unless other options are given in the test. Except as noted below they are not run with multiple optimization options.

'`gcc.dg/compat`'
This subdirectory contains tests for binary compatibility using '`compat.exp`', which in turn uses the language-independent support (see Section 6.4.8 [Support for testing binary compatibility], page 42).

'`gcc.dg/cpp`'
This subdirectory contains tests of the preprocessor.

'`gcc.dg/debug`'
This subdirectory contains tests for debug formats. Tests in this subdirectory are run for each debug format that the compiler supports.

'`gcc.dg/format`'
This subdirectory contains tests of the '`-Wformat`' format checking. Tests in this directory are run with and without '`-DWIDE`'.

'`gcc.dg/noncompile`'
This subdirectory contains tests of code that should not compile and does not need any special compilation options. They are run with multiple optimization options, since sometimes invalid code crashes the compiler with optimization.

'`gcc.dg/special`'
FIXME: describe this.

'`gcc.c-torture`'
This contains particular code fragments which have historically broken easily. These tests are run with multiple optimization options, so tests for features which only break at some optimization levels belong here. This also contains tests to check that certain optimizations occur. It might be worthwhile to separate the correctness tests cleanly from the code quality tests, but it hasn't been done yet.

'`gcc.c-torture/compat`'
FIXME: describe this.

This directory should probably not be used for new tests.

'`gcc.c-torture/compile`'
This testsuite contains test cases that should compile, but do not need to link or run. These test cases are compiled with several different combinations of

optimization options. All warnings are disabled for these test cases, so this directory is not suitable if you wish to test for the presence or absence of compiler warnings. While special options can be set, and tests disabled on specific platforms, by the use of '`.x`' files, mostly these test cases should not contain platform dependencies. FIXME: discuss how defines such as `NO_LABEL_VALUES` and `STACK_SIZE` are used.

'`gcc.c-torture/execute`'
: This testsuite contains test cases that should compile, link and run; otherwise the same comments as for '`gcc.c-torture/compile`' apply.

'`gcc.c-torture/execute/ieee`'
: This contains tests which are specific to IEEE floating point.

'`gcc.c-torture/unsorted`'
: FIXME: describe this.

  This directory should probably not be used for new tests.

'`gcc.c-torture/misc-tests`'
: This directory contains C tests that require special handling. Some of these tests have individual expect files, and others share special-purpose expect files:

  '`bprob*.c`'
  : Test '`-fbranch-probabilities`' using '`bprob.exp`', which in turn uses the generic, language-independent framework (see Section 6.4.7 [Support for testing profile-directed optimizations], page 42).

  '`dg-*.c`' Test the testsuite itself using '`dg-test.exp`'.

  '`gcov*.c`' Test `gcov` output using '`gcov.exp`', which in turn uses the language-independent support (see Section 6.4.6 [Support for testing gcov], page 41).

  '`i386-pf-*.c`'
  : Test i386-specific support for data prefetch using '`i386-prefetch.exp`'.

FIXME: merge in '`testsuite/README.gcc`' and discuss the format of test cases and magic comments more.

6.4.5 The Java library testsuites.

Runtime tests are executed via '`make check`' in the '*`target`*`/libjava/testsuite`' directory in the build tree. Additional runtime tests can be checked into this testsuite.

Regression testing of the core packages in libgcj is also covered by the Mauve testsuite. The Mauve Project develops tests for the Java Class Libraries. These tests are run as part of libgcj testing by placing the Mauve tree within the libjava testsuite sources at '`libjava/testsuite/libjava.mauve/mauve`', or by specifying the location of that tree when invoking '`make`', as in '`make MAUVEDIR=~/mauve check`'.

To detect regressions, a mechanism in '`mauve.exp`' compares the failures for a test run against the list of expected failures in '`libjava/testsuite/libjava.mauve/xfails`' from

the source hierarchy. Update this file when adding new failing tests to Mauve, or when fixing bugs in libgcj that had caused Mauve test failures.

The Jacks project provides a testsuite for Java compilers that can be used to test changes that affect the GCJ front end. This testsuite is run as part of Java testing by placing the Jacks tree within the libjava testsuite sources at '`libjava/testsuite/libjava.jacks/jacks`'.

We encourage developers to contribute test cases to Mauve and Jacks.

6.4.6 Support for testing `gcov`

Language-independent support for testing `gcov`, and for checking that branch profiling produces expected values, is provided by the expect file '`gcov.exp`'. `gcov` tests also rely on procedures in '`gcc.dg.exp`' to compile and run the test program. A typical `gcov` test contains the following DejaGnu commands within comments:

```
{ dg-options "-fprofile-arcs -ftest-coverage" }
{ dg-do run { target native } }
{ dg-final { run-gcov sourcefile } }
```

Checks of `gcov` output can include line counts, branch percentages, and call return percentages. All of these checks are requested via commands that appear in comments in the test's source file. Commands to check line counts are processed by default. Commands to check branch percentages and call return percentages are processed if the `run-gcov` command has arguments `branches` or `calls`, respectively. For example, the following specifies checking both, as well as passing '`-b`' to `gcov`:

```
{ dg-final { run-gcov branches calls { -b sourcefile } } }
```

A line count command appears within a comment on the source line that is expected to get the specified count and has the form `count(`*`cnt`*`)`. A test should only check line counts for lines that will get the same count for any architecture.

Commands to check branch percentages (`branch`) and call return percentages (`returns`) are very similar to each other. A beginning command appears on or before the first of a range of lines that will report the percentage, and the ending command follows that range of lines. The beginning command can include a list of percentages, all of which are expected to be found within the range. A range is terminated by the next command of the same kind. A command `branch(end)` or `returns(end)` marks the end of a range without starting a new one. For example:

```
if (i > 10 && j > i && j < 20)  /* branch(27 50 75) */
                                /* branch(end) */
  foo (i, j);
```

For a call return percentage, the value specified is the percentage of calls reported to return. For a branch percentage, the value is either the expected percentage or 100 minus that value, since the direction of a branch can differ depending on the target or the optimization level.

Not all branches and calls need to be checked. A test should not check for branches that might be optimized away or replaced with predicated instructions. Don't check for calls inserted by the compiler or ones that might be inlined or optimized away.

A single test can check for combinations of line counts, branch percentages, and call return percentages. The command to check a line count must appear on the line that will

report that count, but commands to check branch percentages and call return percentages can bracket the lines that report them.

6.4.7 Support for testing profile-directed optimizations

The file '`profopt.exp`' provides language-independent support for checking correct execution of a test built with profile-directed optimization. This testing requires that a test program be built and executed twice. The first time it is compiled to generate profile data, and the second time it is compiled to use the data that was generated during the first execution. The second execution is to verify that the test produces the expected results.

To check that the optimization actually generated better code, a test can be built and run a third time with normal optimizations to verify that the performance is better with the profile-directed optimizations. '`profopt.exp`' has the beginnings of this kind of support.

'`profopt.exp`' provides generic support for profile-directed optimizations. Each set of tests that uses it provides information about a specific optimization:

`tool` tool being tested, e.g., `gcc`

`profile_option`
: options used to generate profile data

`feedback_option`
: options used to optimize using that profile data

`prof_ext` suffix of profile data files

`PROFOPT_OPTIONS`
: list of options with which to run each test, similar to the lists for torture tests

6.4.8 Support for testing binary compatibility

The file '`compat.exp`' provides language-independent support for binary compatibility testing. It supports testing interoperability of two compilers that follow the same ABI, or of multiple sets of compiler options that should not affect binary compatibility. It is intended to be used for testsuites that complement ABI testsuites.

A test supported by this framework has three parts, each in a separate source file: a main program and two pieces that interact with each other to split up the functionality being tested.

'*`testname`*`_main.`*`suffix`*'
: Contains the main program, which calls a function in file '*`testname`*`_x.`*`suffix`*'.

'*`testname`*`_x.`*`suffix`*'
: Contains at least one call to a function in '*`testname`*`_y.`*`suffix`*'.

'*`testname`*`_y.`*`suffix`*'
: Shares data with, or gets arguments from, '*`testname`*`_x.`*`suffix`*'.

Within each test, the main program and one functional piece are compiled by the GCC under test. The other piece can be compiled by an alternate compiler. If no alternate compiler is specified, then all three source files are all compiled by the GCC under test. You can specify pairs of sets of compiler options. The first element of such a pair specifies

options used with the GCC under test, and the second element of the pair specifies options used with the alternate compiler. Each test is compiled with each pair of options.

'`compat.exp`' defines default pairs of compiler options. These can be overridden by defining the environment variable `COMPAT_OPTIONS` as:

```
COMPAT_OPTIONS="[list [list {tst1} {alt1}]
  ...[list {tstn} {altn}]]"
```

where *tsti* and *alti* are lists of options, with *tsti* used by the compiler under test and *alti* used by the alternate compiler. For example, with `[list [list {-g -O0} {-O3}] [list {-fpic} {-fPIC -O2}]]`, the test is first built with '`-g -O0`' by the compiler under test and with '`-O3`' by the alternate compiler. The test is built a second time using '`-fpic`' by the compiler under test and '`-fPIC -O2`' by the alternate compiler.

An alternate compiler is specified by defining an environment variable to be the full pathname of an installed compiler; for C define `ALT_CC_UNDER_TEST`, and for C++ define `ALT_CXX_UNDER_TEST`. These will be written to the '`site.exp`' file used by DejaGnu. The default is to build each test with the compiler under test using the first of each pair of compiler options from `COMPAT_OPTIONS`. When `ALT_CC_UNDER_TEST` or `ALT_CXX_UNDER_TEST` is `same`, each test is built using the compiler under test but with combinations of the options from `COMPAT_OPTIONS`.

To run only the C++ compatibility suite using the compiler under test and another version of GCC using specific compiler options, do the following from '*objdir*`/gcc`':

```
rm site.exp
make -k \
  ALT_CXX_UNDER_TEST=${alt_prefix}/bin/g++ \
  COMPAT_OPTIONS="lists as shown above" \
  check-c++ \
  RUNTESTFLAGS="compat.exp"
```

A test that fails when the source files are compiled with different compilers, but passes when the files are compiled with the same compiler, demonstrates incompatibility of the generated code or runtime support. A test that fails for the alternate compiler but passes for the compiler under test probably tests for a bug that was fixed in the compiler under test but is present in the alternate compiler.

The binary compatibility tests support a small number of test framework commands that appear within comments in a test file.

`dg-require-*`
: These commands can be used in '*testname*`_main.`*suffix*' to skip the test if specific support is not available on the target.

`dg-options`
: The specified options are used for compiling this particular source file, appended to the options from `COMPAT_OPTIONS`. When this command appears in '*testname*`_main.`*suffix*' the options are also used to link the test program.

`dg-xfail-if`
: This command can be used in a secondary source file to specify that compilation is expected to fail for particular options on particular targets.

7 Option specification files

Most GCC command-line options are described by special option definition files, the names of which conventionally end in `.opt`. This chapter describes the format of these files.

7.1 Option file format

Option files are a simple list of records in which each field occupies its own line and in which the records themselves are separated by blank lines. Comments may appear on their own line anywhere within the file and are preceded by semicolons. Whitespace is allowed before the semicolon.

The files can contain the following types of record:

- A language definition record. These records have two fields: the string '`Language`' and the name of the language. Once a language has been declared in this way, it can be used as an option property. See Section 7.2 [Option properties], page 45.
- An option definition record. These records have the following fields:
  1. the name of the option, with the leading "-" removed
  2. a space-separated list of option properties (see Section 7.2 [Option properties], page 45)
  3. the help text to use for '`--help`' (omitted if the second field contains the `Undocumented` property).

  By default, all options beginning with "f", "W" or "m" are implicitly assumed to take a "no-" form. This form should not be listed separately. If an option beginning with one of these letters does not have a "no-" form, you can use the `RejectNegative` property to reject it.

  The help text is automatically line-wrapped before being displayed. Normally the name of the option is printed on the left-hand side of the output and the help text is printed on the right. However, if the help text contains a tab character, the text to the left of the tab is used instead of the option's name and the text to the right of the tab forms the help text. This allows you to elaborate on what type of argument the option takes.
- A target mask record. These records have one field of the form '`Mask(x)`'. The options-processing script will automatically allocate a bit in `target_flags` (see Section 14.3 [Run-time Target], page 275) for each mask name *x* and set the macro `MASK_x` to the appropriate bitmask. It will also declare a `TARGET_x` macro that has the value 1 when bit `MASK_x` is set and 0 otherwise.

  They are primarily intended to declare target masks that are not associated with user options, either because these masks represent internal switches or because the options are not available on all configurations and yet the masks always need to be defined.

7.2 Option properties

The second field of an option record can specify the following properties:

`Common` The option is available for all languages and targets.

`Target` The option is available for all languages but is target-specific.

language The option is available when compiling for the given language.

It is possible to specify several different languages for the same option. Each *language* must have been declared by an earlier `Language` record. See Section 7.1 [Option file format], page 45.

`RejectNegative`
The option does not have a "no-" form. All options beginning with "f", "W" or "m" are assumed to have a "no-" form unless this property is used.

`Joined`
`Separate` The option takes a mandatory argument. `Joined` indicates that the option and argument can be included in the same `argv` entry (as with `-mflush-func=`*name*, for example). `Separate` indicates that the option and argument can be separate `argv` entries (as with `-o`). An option is allowed to have both of these properties.

`JoinedOrMissing`
The option takes an optional argument. If the argument is given, it will be part of the same `argv` entry as the option itself.

This property cannot be used alongside `Joined` or `Separate`.

`UInteger` The option's argument is a non-negative integer. The option parser will check and convert the argument before passing it to the relevant option handler.

`Var(`*var*`)` The state of this option should be stored in variable *var*. The way that the state is stored depends on the type of option:

- If the option uses the `Mask` or `InverseMask` properties, *var* is the integer variable that contains the mask.
- If the option is a normal on/off switch, *var* is an integer variable that is nonzero when the option is enabled. The options parser will set the variable to 1 when the positive form of the option is used and 0 when the "no-" form is used.
- If the option takes an argument and has the `UInteger` property, *var* is an integer variable that stores the value of the argument.
- Otherwise, if the option takes an argument, *var* is a pointer to the argument string. The pointer will be null if the argument is optional and wasn't given.

The option-processing script will usually declare *var* in '`options.c`' and leave it to be zero-initialized at start-up time. You can modify this behavior using `VarExists` and `Init`.

`Var(`*var*`, `*set*`)`
The option controls an integer variable *var* and is active when *var* equals *set*. The option parser will set *var* to *set* when the positive form of the option is used and `!`*set* when the "no-" form is used.

var is declared in the same way as for the single-argument form described above.

`VarExists`
The variable specified by the `Var` property already exists. No definition should be added to '`options.c`' in response to this option record.

You should use this property if an earlier option has already declared the variable or if the variable is declared outside '`options.c`'.

`Init(value)`
: The variable specified by the `Var` property should be statically initialized to *value*.

`Mask(name)`
: The option is associated with a bit in the `target_flags` variable (see Section 14.3 [Run-time Target], page 275) and is active when that bit is set. You may also specify `Var` to select a variable other than `target_flags`.

  The options-processing script will automatically allocate a unique bit for the option. If the option is attached to '`target_flags`', the script will set the macro `MASK_name` to the appropriate bitmask. It will also declare a `TARGET_name` macro that has the value 1 when the option is active and 0 otherwise. If you use `Var` to attach the option to a different variable, the associated macros are called `OPTION_MASK_name` and `OPTION_name` respectively.

  You can disable automatic bit allocation using `MaskExists`.

`InverseMask(othername)`
`InverseMask(othername, thisname)`
: The option is the inverse of another option that has the `Mask(othername)` property. If *thisname* is given, the options-processing script will declare a `TARGET_thisname` macro that is 1 when the option is active and 0 otherwise.

`MaskExists`
: The mask specified by the `Mask` property already exists. No `MASK` or `TARGET` definitions should be added to '`options.h`' in response to this option record.

  The main purpose of this property is to support synonymous options. The first option should use '`Mask(name)`' and the others should use '`Mask(name) MaskExists`'.

`Report`
: The state of the option should be printed by '`-fverbose-asm`'.

`Undocumented`
: The option is deliberately missing documentation and should not be included in the '`--help`' output.

`Condition(cond)`
: The option should only be accepted if preprocessor condition *cond* is true. Note that any C declarations associated with the option will be present even if *cond* is false; *cond* simply controls whether the option is accepted and whether it is printed in the '`--help`' output.

8 Passes and Files of the Compiler

This chapter is dedicated to giving an overview of the optimization and code generation passes of the compiler. In the process, it describes some of the language front end interface, though this description is no where near complete.

8.1 Parsing pass

The language front end is invoked only once, via `lang_hooks.parse_file`, to parse the entire input. The language front end may use any intermediate language representation deemed appropriate. The C front end uses GENERIC trees (CROSSREF), plus a double handful of language specific tree codes defined in '`c-common.def`'. The Fortran front end uses a completely different private representation.

At some point the front end must translate the representation used in the front end to a representation understood by the language-independent portions of the compiler. Current practice takes one of two forms. The C front end manually invokes the gimplifier (CROSSREF) on each function, and uses the gimplifier callbacks to convert the language-specific tree nodes directly to GIMPLE (CROSSREF) before passing the function off to be compiled. The Fortran front end converts from a private representation to GENERIC, which is later lowered to GIMPLE when the function is compiled. Which route to choose probably depends on how well GENERIC (plus extensions) can be made to match up with the source language and necessary parsing data structures.

BUG: Gimplification must occur before nested function lowering, and nested function lowering must be done by the front end before passing the data off to cgraph.

TODO: Cgraph should control nested function lowering. It would only be invoked when it is certain that the outer-most function is used.

TODO: Cgraph needs a gimplify_function callback. It should be invoked when (1) it is certain that the function is used, (2) warning flags specified by the user require some amount of compilation in order to honor, (3) the language indicates that semantic analysis is not complete until gimplification occurs. Hum... this sounds overly complicated. Perhaps we should just have the front end gimplify always; in most cases it's only one function call.

The front end needs to pass all function definitions and top level declarations off to the middle-end so that they can be compiled and emitted to the object file. For a simple procedural language, it is usually most convenient to do this as each top level declaration or definition is seen. There is also a distinction to be made between generating functional code and generating complete debug information. The only thing that is absolutely required for functional code is that function and data *definitions* be passed to the middle-end. For complete debug information, function, data and type declarations should all be passed as well.

In any case, the front end needs each complete top-level function or data declaration, and each data definition should be passed to `rest_of_decl_compilation`. Each complete type definition should be passed to `rest_of_type_compilation`. Each function definition should be passed to `cgraph_finalize_function`.

TODO: I know rest_of_compilation currently has all sorts of rtl-generation semantics. I plan to move all code generation bits (both tree and rtl) to compile_function. Should we hide cgraph from the front ends and move back to rest_of_compilation as the official

interface? Possibly we should rename all three interfaces such that the names match in some meaningful way and that is more descriptive than "rest_of".

The middle-end will, at its option, emit the function and data definitions immediately or queue them for later processing.

8.2 Gimplification pass

Gimplification is a whimsical term for the process of converting the intermediate representation of a function into the GIMPLE language (CROSSREF). The term stuck, and so words like "gimplification", "gimplify", "gimplifier" and the like are sprinkled throughout this section of code.

While a front end may certainly choose to generate GIMPLE directly if it chooses, this can be a moderately complex process unless the intermediate language used by the front end is already fairly simple. Usually it is easier to generate GENERIC trees plus extensions and let the language-independent gimplifier do most of the work.

The main entry point to this pass is `gimplify_function_tree` located in '`gimplify.c`'. From here we process the entire function gimplifying each statement in turn. The main workhorse for this pass is `gimplify_expr`. Approximately everything passes through here at least once, and it is from here that we invoke the `lang_hooks.gimplify_expr` callback.

The callback should examine the expression in question and return `GS_UNHANDLED` if the expression is not a language specific construct that requires attention. Otherwise it should alter the expression in some way to such that forward progress is made toward producing valid GIMPLE. If the callback is certain that the transformation is complete and the expression is valid GIMPLE, it should return `GS_ALL_DONE`. Otherwise it should return `GS_OK`, which will cause the expression to be processed again. If the callback encounters an error during the transformation (because the front end is relying on the gimplification process to finish semantic checks), it should return `GS_ERROR`.

8.3 Pass manager

The pass manager is located in '`passes.c`', '`tree-optimize.c`' and '`tree-pass.h`'. Its job is to run all of the individual passes in the correct order, and take care of standard bookkeeping that applies to every pass.

The theory of operation is that each pass defines a structure that represents everything we need to know about that pass—when it should be run, how it should be run, what intermediate language form or on-the-side data structures it needs. We register the pass to be run in some particular order, and the pass manager arranges for everything to happen in the correct order.

The actuality doesn't completely live up to the theory at present. Command-line switches and `timevar_id_t` enumerations must still be defined elsewhere. The pass manager validates constraints but does not attempt to (re-)generate data structures or lower intermediate language form based on the requirements of the next pass. Nevertheless, what is present is useful, and a far sight better than nothing at all.

TODO: describe the global variables set up by the pass manager, and a brief description of how a new pass should use it. I need to look at what info rtl passes use first...

8.4 Tree-SSA passes

The following briefly describes the tree optimization passes that are run after gimplification and what source files they are located in.

- Remove useless statements

  This pass is an extremely simple sweep across the gimple code in which we identify obviously dead code and remove it. Here we do things like simplify `if` statements with constant conditions, remove exception handling constructs surrounding code that obviously cannot throw, remove lexical bindings that contain no variables, and other assorted simplistic cleanups. The idea is to get rid of the obvious stuff quickly rather than wait until later when it's more work to get rid of it. This pass is located in '`tree-cfg.c`' and described by `pass_remove_useless_stmts`.

- Mudflap declaration registration

  If mudflap (see section "-fmudflap -fmudflapth -fmudflapir" in *Using the GNU Compiler Collection (GCC)*) is enabled, we generate code to register some variable declarations with the mudflap runtime. Specifically, the runtime tracks the lifetimes of those variable declarations that have their addresses taken, or whose bounds are unknown at compile time (`extern`). This pass generates new exception handling constructs (`try/finally`), and so must run before those are lowered. In addition, the pass enqueues declarations of static variables whose lifetimes extend to the entire program. The pass is located in '`tree-mudflap.c`' and is described by `pass_mudflap_1`.

- Lower control flow

  This pass flattens `if` statements (`COND_EXPR`) and and moves lexical bindings (`BIND_EXPR`) out of line. After this pass, all `if` statements will have exactly two `goto` statements in its `then` and `else` arms. Lexical binding information for each statement will be found in `TREE_BLOCK` rather than being inferred from its position under a `BIND_EXPR`. This pass is found in '`gimple-low.c`' and is described by `pass_lower_cf`.

- Lower exception handling control flow

  This pass decomposes high-level exception handling constructs (`TRY_FINALLY_EXPR` and `TRY_CATCH_EXPR`) into a form that explicitly represents the control flow involved. After this pass, `lookup_stmt_eh_region` will return a non-negative number for any statement that may have EH control flow semantics; examine `tree_can_throw_internal` or `tree_can_throw_external` for exact semantics. Exact control flow may be extracted from `foreach_reachable_handler`. The EH region nesting tree is defined in '`except.h`' and built in '`except.c`'. The lowering pass itself is in '`tree-eh.c`' and is described by `pass_lower_eh`.

- Build the control flow graph

  This pass decomposes a function into basic blocks and creates all of the edges that connect them. It is located in '`tree-cfg.c`' and is described by `pass_build_cfg`.

- Find all referenced variables

  This pass walks the entire function and collects an array of all variables referenced in the function, `referenced_vars`. The index at which a variable is found in the array is used as a UID for the variable within this function. This data is needed by the SSA rewriting routines. The pass is located in '`tree-dfa.c`' and is described by `pass_referenced_vars`.

- Enter static single assignment form

  This pass rewrites the function such that it is in SSA form. After this pass, all `is_gimple_reg` variables will be referenced by `SSA_NAME`, and all occurrences of other variables will be annotated with `VDEFS` and `VUSES`; phi nodes will have been inserted as necessary for each basic block. This pass is located in '`tree-ssa.c`' and is described by `pass_build_ssa`.

- Warn for uninitialized variables

  This pass scans the function for uses of `SSA_NAME`s that are fed by default definition. For non-parameter variables, such uses are uninitialized. The pass is run twice, before and after optimization. In the first pass we only warn for uses that are positively uninitialized; in the second pass we warn for uses that are possibly uninitialized. The pass is located in '`tree-ssa.c`' and is defined by `pass_early_warn_uninitialized` and `pass_late_warn_uninitialized`.

- Dead code elimination

  This pass scans the function for statements without side effects whose result is unused. It does not do memory life analysis, so any value that is stored in memory is considered used. The pass is run multiple times throughout the optimization process. It is located in '`tree-ssa-dce.c`' and is described by `pass_dce`.

- Dominator optimizations

  This pass performs trivial dominator-based copy and constant propagation, expression simplification, and jump threading. It is run multiple times throughout the optimization process. It it located in '`tree-ssa-dom.c`' and is described by `pass_dominator`.

- Redundant phi elimination

  This pass removes phi nodes for which all of the arguments are the same value, excluding feedback. Such degenerate forms are typically created by removing unreachable code. The pass is run multiple times throughout the optimization process. It is located in '`tree-ssa.c`' and is described by `pass_redundant_phi`.o

- Forward propagation of single-use variables

  This pass attempts to remove redundant computation by substituting variables that are used once into the expression that uses them and seeing if the result can be simplified. It is located in '`tree-ssa-forwprop.c`' and is described by `pass_forwprop`.

- Copy Renaming

  This pass attempts to change the name of compiler temporaries involved in copy operations such that SSA->normal can coalesce the copy away. When compiler temporaries are copies of user variables, it also renames the compiler temporary to the user variable resulting in better use of user symbols. It is located in '`tree-ssa-copyrename.c`' and is described by `pass_copyrename`.

- PHI node optimizations

  This pass recognizes forms of phi inputs that can be represented as conditional expressions and rewrites them into straight line code. It is located in '`tree-ssa-phiopt.c`' and is described by `pass_phiopt`.

- May-alias optimization

  This pass performs a flow sensitive SSA-based points-to analysis. The resulting may-alias, must-alias, and escape analysis information is used to promote variables from

in-memory addressable objects to non-aliased variables that can be renamed into SSA form. We also update the `VDEF/VUSE` memory tags for non-renamable aggregates so that we get fewer false kills. The pass is located in '`tree-ssa-alias.c`' and is described by `pass_may_alias`.

- Profiling

  This pass rewrites the function in order to collect runtime block and value profiling data. Such data may be fed back into the compiler on a subsequent run so as to allow optimization based on expected execution frequencies. The pass is located in '`predict.c`' and is described by `pass_profile`.

- Lower complex arithmetic

  This pass rewrites complex arithmetic operations into their component scalar arithmetic operations. The pass is located in '`tree-complex.c`' and is described by `pass_lower_complex`.

- Scalar replacement of aggregates

  This pass rewrites suitable non-aliased local aggregate variables into a set of scalar variables. The resulting scalar variables are rewritten into SSA form, which allows subsequent optimization passes to do a significantly better job with them. The pass is located in '`tree-sra.c`' and is described by `pass_sra`.

- Dead store elimination

  This pass eliminates stores to memory that are subsequently overwritten by another store, without any intervening loads. The pass is located in '`tree-ssa-dse.c`' and is described by `pass_dse`.

- Tail recursion elimination

  This pass transforms tail recursion into a loop. It is located in '`tree-tailcall.c`' and is described by `pass_tail_recursion`.

- Forward store motion

  This pass sinks stores and assignments down the flowgraph closer to it's use point. The pass is located in '`tree-ssa-sink.c`' and is described by `pass_sink_code`.

- Partial redundancy elimination

  This pass eliminates partially redundant computations, as well as performing load motion. The pass is located in '`tree-ssa-pre.c`' and is described by `pass_pre`.

  Just before partial redundancy elimination, if '`-funsafe-math-optimizations`' is on, GCC tries to convert divisions to multiplications by the reciprocal. The pass is located in '`tree-ssa-math-opts.c`' and is described by `pass_cse_reciprocal`.

- Loop optimization

  The main driver of the pass is placed in '`tree-ssa-loop.c`' and described by `pass_loop`.

  The optimizations performed by this pass are:

  Loop invariant motion. This pass moves only invariants that would be hard to handle on rtl level (function calls, operations that expand to nontrivial sequences of insns). With '`-funswitch-loops`' it also moves operands of conditions that are invariant out of the loop, so that we can use just trivial invariantness analysis in loop unswitching. The pass also includes store motion. The pass is implemented in '`tree-ssa-loop-im.c`'.

Canonical induction variable creation. This pass creates a simple counter for number of iterations of the loop and replaces the exit condition of the loop using it, in case when a complicated analysis is necessary to determine the number of iterations. Later optimizations then may determine the number easily. The pass is implemented in '`tree-ssa-loop-ivcanon.c`'.

Induction variable optimizations. This pass performs standard induction variable optimizations, including strength reduction, induction variable merging and induction variable elimination. The pass is implemented in '`tree-ssa-loop-ivopts.c`'.

Loop unswitching. This pass moves the conditional jumps that are invariant out of the loops. To achieve this, a duplicate of the loop is created for each possible outcome of conditional jump(s). The pass is implemented in '`tree-ssa-loop-unswitch.c`'. This pass should eventually replace the rtl-level loop unswitching in '`loop-unswitch.c`', but currently the rtl-level pass is not completely redundant yet due to deficiencies in tree level alias analysis.

The optimizations also use various utility functions contained in '`tree-ssa-loop-manip.c`', '`cfgloop.c`', '`cfgloopanal.c`' and '`cfgloopmanip.c`'.

Vectorization. This pass transforms loops to operate on vector types instead of scalar types. Data parallelism across loop iterations is exploited to group data elements from consecutive iterations into a vector and operate on them in parallel. Depending on available target support the loop is conceptually unrolled by a factor `VF` (vectorization factor), which is the number of elements operated upon in parallel in each iteration, and the `VF` copies of each scalar operation are fused to form a vector operation. Additional loop transformations such as peeling and versioning may take place to align the number of iterations, and to align the memory accesses in the loop. The pass is implemented in '`tree-vectorizer.c`' (the main driver and general utilities), '`tree-vect-analyze.c`' and '`tree-vect-transform.c`'. Analysis of data references is in '`tree-data-ref.c`'.

- Tree level if-conversion for vectorizer

  This pass applies if-conversion to simple loops to help vectorizer. We identify if convertable loops, if-convert statements and merge basic blocks in one big block. The idea is to present loop in such form so that vectorizer can have one to one mapping between statements and available vector operations. This patch re-introduces COND_EXPR at GIMPLE level. This pass is located in '`tree-if-conv.c`'.

- Conditional constant propagation

  This pass relaxes a lattice of values in order to identify those that must be constant even in the presence of conditional branches. The pass is located in '`tree-ssa-ccp.c`' and is described by `pass_ccp`.

- Folding builtin functions

  This pass simplifies builtin functions, as applicable, with constant arguments or with inferrable string lengths. It is located in '`tree-ssa-ccp.c`' and is described by `pass_fold_builtins`.

- Split critical edges

  This pass identifies critical edges and inserts empty basic blocks such that the edge is no longer critical. The pass is located in '`tree-cfg.c`' and is described by `pass_split_crit_edges`.

- Partial redundancy elimination

  This pass answers the question "given a hypothetical temporary variable, what expressions could we eliminate?" It is located in 'tree-ssa-pre.c' and is described by pass_pre.

- Control dependence dead code elimination

  This pass is a stronger form of dead code elimination that can eliminate unnecessary control flow statements. It is located in 'tree-ssa-dce.c' and is described by pass_cd_dce.

- Tail call elimination

  This pass identifies function calls that may be rewritten into jumps. No code transformation is actually applied here, but the data and control flow problem is solved. The code transformation requires target support, and so is delayed until RTL. In the meantime CALL_EXPR_TAILCALL is set indicating the possibility. The pass is located in 'tree-tailcall.c' and is described by pass_tail_calls. The RTL transformation is handled by fixup_tail_calls in 'calls.c'.

- Warn for function return without value

  For non-void functions, this pass locates return statements that do not specify a value and issues a warning. Such a statement may have been injected by falling off the end of the function. This pass is run last so that we have as much time as possible to prove that the statement is not reachable. It is located in 'tree-cfg.c' and is described by pass_warn_function_return.

- Mudflap statement annotation

  If mudflap is enabled, we rewrite some memory accesses with code to validate that the memory access is correct. In particular, expressions involving pointer dereferences (INDIRECT_REF, ARRAY_REF, etc.) are replaced by code that checks the selected address range against the mudflap runtime's database of valid regions. This check includes an inline lookup into a direct-mapped cache, based on shift/mask operations of the pointer value, with a fallback function call into the runtime. The pass is located in 'tree-mudflap.c' and is described by pass_mudflap_2.

- Leave static single assignment form

  This pass rewrites the function such that it is in normal form. At the same time, we eliminate as many single-use temporaries as possible, so the intermediate language is no longer GIMPLE, but GENERIC. The pass is located in 'tree-ssa.c' and is described by pass_del_ssa.

8.5 RTL passes

The following briefly describes the rtl generation and optimization passes that are run after tree optimization.

- RTL generation

  The source files for RTL generation include 'stmt.c', 'calls.c', 'expr.c', 'explow.c', 'expmed.c', 'function.c', 'optabs.c' and 'emit-rtl.c'. Also, the file 'insn-emit.c', generated from the machine description by the program genemit, is used in this pass. The header file 'expr.h' is used for communication within this pass.

The header files '`insn-flags.h`' and '`insn-codes.h`', generated from the machine description by the programs `genflags` and `gencodes`, tell this pass which standard names are available for use and which patterns correspond to them.

- Generate exception handling landing pads

  This pass generates the glue that handles communication between the exception handling library routines and the exception handlers within the function. Entry points in the function that are invoked by the exception handling library are called *landing pads*. The code for this pass is located within '`except.c`'.

- Cleanup control flow graph

  This pass removes unreachable code, simplifies jumps to next, jumps to jump, jumps across jumps, etc. The pass is run multiple times. For historical reasons, it is occasionally referred to as the "jump optimization pass". The bulk of the code for this pass is in '`cfgcleanup.c`', and there are support routines in '`cfgrtl.c`' and '`jump.c`'.

- Common subexpression elimination

  This pass removes redundant computation within basic blocks, and optimizes addressing modes based on cost. The pass is run twice. The source is located in '`cse.c`'.

- Global common subexpression elimination.

  This pass performs two different types of GCSE depending on whether you are optimizing for size or not (LCM based GCSE tends to increase code size for a gain in speed, while Morel-Renvoise based GCSE does not). When optimizing for size, GCSE is done using Morel-Renvoise Partial Redundancy Elimination, with the exception that it does not try to move invariants out of loops—that is left to the loop optimization pass. If MR PRE GCSE is done, code hoisting (aka unification) is also done, as well as load motion. If you are optimizing for speed, LCM (lazy code motion) based GCSE is done. LCM is based on the work of Knoop, Ruthing, and Steffen. LCM based GCSE also does loop invariant code motion. We also perform load and store motion when optimizing for speed. Regardless of which type of GCSE is used, the GCSE pass also performs global constant and copy propagation. The source file for this pass is '`gcse.c`', and the LCM routines are in '`lcm.c`'.

- Loop optimization

  This pass moves constant expressions out of loops, and optionally does strength-reduction as well. The pass is located in '`loop.c`'. Loop dependency analysis routines are contained in '`dependence.c`'. This pass is seriously out-of-date and is supposed to be replaced by a new one described below in near future.

  A second loop optimization pass takes care of basic block level optimizations—unrolling, peeling and unswitching loops. The source files are '`cfgloopanal.c`' and '`cfgloopmanip.c`' containing generic loop analysis and manipulation code, '`loop-init.c`' with initialization and finalization code, '`loop-unswitch.c`' for loop unswitching and '`loop-unroll.c`' for loop unrolling and peeling. It also contains a separate loop invariant motion pass implemented in '`loop-invariant.c`'.

- Jump bypassing

  This pass is an aggressive form of GCSE that transforms the control flow graph of a function by propagating constants into conditional branch instructions. The source file for this pass is '`gcse.c`'.

- If conversion

  This pass attempts to replace conditional branches and surrounding assignments with arithmetic, boolean value producing comparison instructions, and conditional move instructions. In the very last invocation after reload, it will generate predicated instructions when supported by the target. The pass is located in '`ifcvt.c`'.

- Web construction

  This pass splits independent uses of each pseudo-register. This can improve effect of the other transformation, such as CSE or register allocation. Its source files are '`web.c`'.

- Life analysis

  This pass computes which pseudo-registers are live at each point in the program, and makes the first instruction that uses a value point at the instruction that computed the value. It then deletes computations whose results are never used, and combines memory references with add or subtract instructions to make autoincrement or autodecrement addressing. The pass is located in '`flow.c`'.

- Instruction combination

  This pass attempts to combine groups of two or three instructions that are related by data flow into single instructions. It combines the RTL expressions for the instructions by substitution, simplifies the result using algebra, and then attempts to match the result against the machine description. The pass is located in '`combine.c`'.

- Register movement

  This pass looks for cases where matching constraints would force an instruction to need a reload, and this reload would be a register-to-register move. It then attempts to change the registers used by the instruction to avoid the move instruction. The pass is located in '`regmove.c`'.

- Optimize mode switching

  This pass looks for instructions that require the processor to be in a specific "mode" and minimizes the number of mode changes required to satisfy all users. What these modes are, and what they apply to are completely target-specific. The source is located in '`mode-switching.c`'.

- Modulo scheduling

  This pass looks at innermost loops and reorders their instructions by overlapping different iterations. Modulo scheduling is performed immediately before instruction scheduling. The pass is located in ('`modulo-sched.c`').

- Instruction scheduling

  This pass looks for instructions whose output will not be available by the time that it is used in subsequent instructions. Memory loads and floating point instructions often have this behavior on RISC machines. It re-orders instructions within a basic block to try to separate the definition and use of items that otherwise would cause pipeline stalls. This pass is performed twice, before and after register allocation. The pass is located in '`haifa-sched.c`', '`sched-deps.c`', '`sched-ebb.c`', '`sched-rgn.c`' and '`sched-vis.c`'.

- Register allocation

  These passes make sure that all occurrences of pseudo registers are eliminated, either by allocating them to a hard register, replacing them by an equivalent expression (e.g. a constant) or by placing them on the stack. This is done in several subpasses:

  - Register class preferencing. The RTL code is scanned to find out which register class is best for each pseudo register. The source file is '`regclass.c`'.
  - Local register allocation. This pass allocates hard registers to pseudo registers that are used only within one basic block. Because the basic block is linear, it can use fast and powerful techniques to do a decent job. The source is located in '`local-alloc.c`'.
  - Global register allocation. This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block). The pass is located in '`global.c`'.
  - Reloading. This pass renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Then it finds instructions that are invalid because a value has failed to end up in a register, or has ended up in a register of the wrong kind. It fixes up these instructions by reloading the problematical values temporarily into registers. Additional instructions are generated to do the copying.

    The reload pass also optionally eliminates the frame pointer and inserts instructions to save and restore call-clobbered registers around calls.

    Source files are '`reload.c`' and '`reload1.c`', plus the header '`reload.h`' used for communication between them.

- Basic block reordering

  This pass implements profile guided code positioning. If profile information is not available, various types of static analysis are performed to make the predictions normally coming from the profile feedback (IE execution frequency, branch probability, etc). It is implemented in the file '`bb-reorder.c`', and the various prediction routines are in '`predict.c`'.

- Variable tracking

  This pass computes where the variables are stored at each position in code and generates notes describing the variable locations to RTL code. The location lists are then generated according to these notes to debug information if the debugging information format supports location lists.

- Delayed branch scheduling

  This optional pass attempts to find instructions that can go into the delay slots of other instructions, usually jumps and calls. The source file name is '`reorg.c`'.

- Branch shortening

  On many RISC machines, branch instructions have a limited range. Thus, longer sequences of instructions must be used for long branches. In this pass, the compiler figures out what how far each instruction will be from each other instruction, and therefore whether the usual instructions, or the longer sequences, must be used for each branch.

- Register-to-stack conversion

  Conversion from usage of some hard registers to usage of a register stack may be done at this point. Currently, this is supported only for the floating-point registers of the Intel 80387 coprocessor. The source file name is '`reg-stack.c`'.

- Final

  This pass outputs the assembler code for the function. The source files are '`final.c`' plus '`insn-output.c`'; the latter is generated automatically from the machine description by the tool '`genoutput`'. The header file '`conditions.h`' is used for communication between these files. If mudflap is enabled, the queue of deferred declarations and any addressed constants (e.g., string literals) is processed by `mudflap_finish_file` into a synthetic constructor function containing calls into the mudflap runtime.

- Debugging information output

  This is run after final because it must output the stack slot offsets for pseudo registers that did not get hard registers. Source files are '`dbxout.c`' for DBX symbol table format, '`sdbout.c`' for SDB symbol table format, '`dwarfout.c`' for DWARF symbol table format, files '`dwarf2out.c`' and '`dwarf2asm.c`' for DWARF2 symbol table format, and '`vmsdbgout.c`' for VMS debug symbol table format.

9 Trees: The intermediate representation used by the C and C++ front ends

This chapter documents the internal representation used by GCC to represent C and C++ source programs. When presented with a C or C++ source program, GCC parses the program, performs semantic analysis (including the generation of error messages), and then produces the internal representation described here. This representation contains a complete representation for the entire translation unit provided as input to the front end. This representation is then typically processed by a code-generator in order to produce machine code, but could also be used in the creation of source browsers, intelligent editors, automatic documentation generators, interpreters, and any other programs needing the ability to process C or C++ code.

This chapter explains the internal representation. In particular, it documents the internal representation for C and C++ source constructs, and the macros, functions, and variables that can be used to access these constructs. The C++ representation is largely a superset of the representation used in the C front end. There is only one construct used in C that does not appear in the C++ front end and that is the GNU "nested function" extension. Many of the macros documented here do not apply in C because the corresponding language constructs do not appear in C.

If you are developing a "back end", be it is a code-generator or some other tool, that uses this representation, you may occasionally find that you need to ask questions not easily answered by the functions and macros available here. If that situation occurs, it is quite likely that GCC already supports the functionality you desire, but that the interface is simply not documented here. In that case, you should ask the GCC maintainers (via mail to `gcc@gcc.gnu.org`) about documenting the functionality you require. Similarly, if you find yourself writing functions that do not deal directly with your back end, but instead might be useful to other people using the GCC front end, you should submit your patches for inclusion in GCC.

9.1 Deficiencies

There are many places in which this document is incomplet and incorrekt. It is, as of yet, only *preliminary* documentation.

9.2 Overview

The central data structure used by the internal representation is the `tree`. These nodes, while all of the C type `tree`, are of many varieties. A `tree` is a pointer type, but the object to which it points may be of a variety of types. From this point forward, we will refer to trees in ordinary type, rather than in `this font`, except when talking about the actual C type `tree`.

You can tell what kind of node a particular tree is by using the `TREE_CODE` macro. Many, many macros take trees as input and return trees as output. However, most macros require a certain kind of tree node as input. In other words, there is a type-system for trees, but it is not reflected in the C type-system.

For safety, it is useful to configure GCC with '`--enable-checking`'. Although this results in a significant performance penalty (since all tree types are checked at run-time), and is

therefore inappropriate in a release version, it is extremely helpful during the development process.

Many macros behave as predicates. Many, although not all, of these predicates end in '`_P`'. Do not rely on the result type of these macros being of any particular type. You may, however, rely on the fact that the type can be compared to `0`, so that statements like

```
if (TEST_P (t) && !TEST_P (y))
  x = 1;
```

and

```
int i = (TEST_P (t) != 0);
```

are legal. Macros that return `int` values now may be changed to return `tree` values, or other pointers in the future. Even those that continue to return `int` may return multiple nonzero codes where previously they returned only zero and one. Therefore, you should not write code like

```
if (TEST_P (t) == 1)
```

as this code is not guaranteed to work correctly in the future.

You should not take the address of values returned by the macros or functions described here. In particular, no guarantee is given that the values are lvalues.

In general, the names of macros are all in uppercase, while the names of functions are entirely in lowercase. There are rare exceptions to this rule. You should assume that any macro or function whose name is made up entirely of uppercase letters may evaluate its arguments more than once. You may assume that a macro or function whose name is made up entirely of lowercase letters will evaluate its arguments only once.

The `error_mark_node` is a special tree. Its tree code is `ERROR_MARK`, but since there is only ever one node with that code, the usual practice is to compare the tree against `error_mark_node`. (This test is just a test for pointer equality.) If an error has occurred during front-end processing the flag `errorcount` will be set. If the front end has encountered code it cannot handle, it will issue a message to the user and set `sorrycount`. When these flags are set, any macro or function which normally returns a tree of a particular kind may instead return the `error_mark_node`. Thus, if you intend to do any processing of erroneous code, you must be prepared to deal with the `error_mark_node`.

Occasionally, a particular tree slot (like an operand to an expression, or a particular field in a declaration) will be referred to as "reserved for the back end". These slots are used to store RTL when the tree is converted to RTL for use by the GCC back end. However, if that process is not taking place (e.g., if the front end is being hooked up to an intelligent editor), then those slots may be used by the back end presently in use.

If you encounter situations that do not match this documentation, such as tree nodes of types not mentioned here, or macros documented to return entities of a particular kind that instead return entities of some different kind, you have found a bug, either in the front end or in the documentation. Please report these bugs as you would any other bug.

9.2.1 Trees

This section is not here yet.

9.2.2 Identifiers

An `IDENTIFIER_NODE` represents a slightly more general concept that the standard C or C++ concept of identifier. In particular, an `IDENTIFIER_NODE` may contain a '`$`', or other extraordinary characters.

There are never two distinct IDENTIFIER_NODEs representing the same identifier. Therefore, you may use pointer equality to compare IDENTIFIER_NODEs, rather than using a routine like strcmp.

You can use the following macros to access identifiers:

IDENTIFIER_POINTER
: The string represented by the identifier, represented as a char*. This string is always NUL-terminated, and contains no embedded NUL characters.

IDENTIFIER_LENGTH
: The length of the string returned by IDENTIFIER_POINTER, not including the trailing NUL. This value of IDENTIFIER_LENGTH (x) is always the same as strlen (IDENTIFIER_POINTER (x)).

IDENTIFIER_OPNAME_P
: This predicate holds if the identifier represents the name of an overloaded operator. In this case, you should not depend on the contents of either the IDENTIFIER_POINTER or the IDENTIFIER_LENGTH.

IDENTIFIER_TYPENAME_P
: This predicate holds if the identifier represents the name of a user-defined conversion operator. In this case, the TREE_TYPE of the IDENTIFIER_NODE holds the type to which the conversion operator converts.

9.2.3 Containers

Two common container data structures can be represented directly with tree nodes. A TREE_LIST is a singly linked list containing two trees per node. These are the TREE_PURPOSE and TREE_VALUE of each node. (Often, the TREE_PURPOSE contains some kind of tag, or additional information, while the TREE_VALUE contains the majority of the payload. In other cases, the TREE_PURPOSE is simply NULL_TREE, while in still others both the TREE_PURPOSE and TREE_VALUE are of equal stature.) Given one TREE_LIST node, the next node is found by following the TREE_CHAIN. If the TREE_CHAIN is NULL_TREE, then you have reached the end of the list.

A TREE_VEC is a simple vector. The TREE_VEC_LENGTH is an integer (not a tree) giving the number of nodes in the vector. The nodes themselves are accessed using the TREE_VEC_ELT macro, which takes two arguments. The first is the TREE_VEC in question; the second is an integer indicating which element in the vector is desired. The elements are indexed from zero.

9.3 Types

All types have corresponding tree nodes. However, you should not assume that there is exactly one tree node corresponding to each type. There are often several nodes each of which correspond to the same type.

For the most part, different kinds of types have different tree codes. (For example, pointer types use a POINTER_TYPE code while arrays use an ARRAY_TYPE code.) However, pointers to member functions use the RECORD_TYPE code. Therefore, when writing a switch statement that depends on the code associated with a particular type, you should take care to handle pointers to member functions under the RECORD_TYPE case label.

In C++, an array type is not qualified; rather the type of the array elements is qualified. This situation is reflected in the intermediate representation. The macros described here will always examine the qualification of the underlying element type when applied to an array type. (If the element type is itself an array, then the recursion continues until a non-array type is found, and the qualification of this type is examined.) So, for example, `CP_TYPE_CONST_P` will hold of the type `const int ()[7]`, denoting an array of seven `int`s.

The following functions and macros deal with cv-qualification of types:

`CP_TYPE_QUALS`
: This macro returns the set of type qualifiers applied to this type. This value is `TYPE_UNQUALIFIED` if no qualifiers have been applied. The `TYPE_QUAL_CONST` bit is set if the type is `const`-qualified. The `TYPE_QUAL_VOLATILE` bit is set if the type is `volatile`-qualified. The `TYPE_QUAL_RESTRICT` bit is set if the type is `restrict`-qualified.

`CP_TYPE_CONST_P`
: This macro holds if the type is `const`-qualified.

`CP_TYPE_VOLATILE_P`
: This macro holds if the type is `volatile`-qualified.

`CP_TYPE_RESTRICT_P`
: This macro holds if the type is `restrict`-qualified.

`CP_TYPE_CONST_NON_VOLATILE_P`
: This predicate holds for a type that is `const`-qualified, but *not* `volatile`-qualified; other cv-qualifiers are ignored as well: only the `const`-ness is tested.

`TYPE_MAIN_VARIANT`
: This macro returns the unqualified version of a type. It may be applied to an unqualified type, but it is not always the identity function in that case.

A few other macros and functions are usable with all types:

`TYPE_SIZE`
: The number of bits required to represent the type, represented as an `INTEGER_CST`. For an incomplete type, `TYPE_SIZE` will be `NULL_TREE`.

`TYPE_ALIGN`
: The alignment of the type, in bits, represented as an `int`.

`TYPE_NAME`
: This macro returns a declaration (in the form of a `TYPE_DECL`) for the type. (Note this macro does *not* return a `IDENTIFIER_NODE`, as you might expect, given its name!) You can look at the `DECL_NAME` of the `TYPE_DECL` to obtain the actual name of the type. The `TYPE_NAME` will be `NULL_TREE` for a type that is not a built-in type, the result of a typedef, or a named class type.

`CP_INTEGRAL_TYPE`
: This predicate holds if the type is an integral type. Notice that in C++, enumerations are *not* integral types.

`ARITHMETIC_TYPE_P`
: This predicate holds if the type is an integral type (in the C++ sense) or a floating point type.

`CLASS_TYPE_P`
: This predicate holds for a class-type.

`TYPE_BUILT_IN`
: This predicate holds for a built-in type.

`TYPE_PTRMEM_P`
: This predicate holds if the type is a pointer to data member.

`TYPE_PTR_P`
: This predicate holds if the type is a pointer type, and the pointee is not a data member.

`TYPE_PTRFN_P`
: This predicate holds for a pointer to function type.

`TYPE_PTROB_P`
: This predicate holds for a pointer to object type. Note however that it does not hold for the generic pointer to object type `void *`. You may use `TYPE_PTROBV_P` to test for a pointer to object type as well as `void *`.

`same_type_p`
: This predicate takes two types as input, and holds if they are the same type. For example, if one type is a `typedef` for the other, or both are `typedef`s for the same type. This predicate also holds if the two trees given as input are simply copies of one another; i.e., there is no difference between them at the source level, but, for whatever reason, a duplicate has been made in the representation. You should never use `==` (pointer equality) to compare types; always use `same_type_p` instead.

Detailed below are the various kinds of types, and the macros that can be used to access them. Although other kinds of types are used elsewhere in G++, the types described here are the only ones that you will encounter while examining the intermediate representation.

`VOID_TYPE`
: Used to represent the `void` type.

`INTEGER_TYPE`
: Used to represent the various integral types, including `char`, `short`, `int`, `long`, and `long long`. This code is not used for enumeration types, nor for the `bool` type. Note that GCC's `CHAR_TYPE` node is *not* used to represent `char`. The `TYPE_PRECISION` is the number of bits used in the representation, represented as an `unsigned int`. (Note that in the general case this is not the same value as `TYPE_SIZE`; suppose that there were a 24-bit integer type, but that alignment requirements for the ABI required 32-bit alignment. Then, `TYPE_SIZE` would be an `INTEGER_CST` for 32, while `TYPE_PRECISION` would be 24.) The integer type is unsigned if `TYPE_UNSIGNED` holds; otherwise, it is signed.

The TYPE_MIN_VALUE is an INTEGER_CST for the smallest integer that may be represented by this type. Similarly, the TYPE_MAX_VALUE is an INTEGER_CST for the largest integer that may be represented by this type.

REAL_TYPE
: Used to represent the float, double, and long double types. The number of bits in the floating-point representation is given by TYPE_PRECISION, as in the INTEGER_TYPE case.

COMPLEX_TYPE
: Used to represent GCC built-in __complex__ data types. The TREE_TYPE is the type of the real and imaginary parts.

ENUMERAL_TYPE
: Used to represent an enumeration type. The TYPE_PRECISION gives (as an int), the number of bits used to represent the type. If there are no negative enumeration constants, TYPE_UNSIGNED will hold. The minimum and maximum enumeration constants may be obtained with TYPE_MIN_VALUE and TYPE_MAX_VALUE, respectively; each of these macros returns an INTEGER_CST.

  The actual enumeration constants themselves may be obtained by looking at the TYPE_VALUES. This macro will return a TREE_LIST, containing the constants. The TREE_PURPOSE of each node will be an IDENTIFIER_NODE giving the name of the constant; the TREE_VALUE will be an INTEGER_CST giving the value assigned to that constant. These constants will appear in the order in which they were declared. The TREE_TYPE of each of these constants will be the type of enumeration type itself.

BOOLEAN_TYPE
: Used to represent the bool type.

POINTER_TYPE
: Used to represent pointer types, and pointer to data member types. The TREE_TYPE gives the type to which this type points. If the type is a pointer to data member type, then TYPE_PTRMEM_P will hold. For a pointer to data member type of the form 'T X::*', TYPE_PTRMEM_CLASS_TYPE will be the type X, while TYPE_PTRMEM_POINTED_TO_TYPE will be the type T.

REFERENCE_TYPE
: Used to represent reference types. The TREE_TYPE gives the type to which this type refers.

FUNCTION_TYPE
: Used to represent the type of non-member functions and of static member functions. The TREE_TYPE gives the return type of the function. The TYPE_ARG_TYPES are a TREE_LIST of the argument types. The TREE_VALUE of each node in this list is the type of the corresponding argument; the TREE_PURPOSE is an expression for the default argument value, if any. If the last node in the list is void_list_node (a TREE_LIST node whose TREE_VALUE is the void_type_node), then functions of this type do not take variable arguments. Otherwise, they do take a variable number of arguments.

Note that in C (but not in C++) a function declared like `void f()` is an unprototyped function taking a variable number of arguments; the `TYPE_ARG_TYPES` of such a function will be `NULL`.

`METHOD_TYPE`
: Used to represent the type of a non-static member function. Like a `FUNCTION_TYPE`, the return type is given by the `TREE_TYPE`. The type of `*this`, i.e., the class of which functions of this type are a member, is given by the `TYPE_METHOD_BASETYPE`. The `TYPE_ARG_TYPES` is the parameter list, as for a `FUNCTION_TYPE`, and includes the `this` argument.

`ARRAY_TYPE`
: Used to represent array types. The `TREE_TYPE` gives the type of the elements in the array. If the array-bound is present in the type, the `TYPE_DOMAIN` is an `INTEGER_TYPE` whose `TYPE_MIN_VALUE` and `TYPE_MAX_VALUE` will be the lower and upper bounds of the array, respectively. The `TYPE_MIN_VALUE` will always be an `INTEGER_CST` for zero, while the `TYPE_MAX_VALUE` will be one less than the number of elements in the array, i.e., the highest value which may be used to index an element in the array.

`RECORD_TYPE`
: Used to represent `struct` and `class` types, as well as pointers to member functions and similar constructs in other languages. `TYPE_FIELDS` contains the items contained in this type, each of which can be a `FIELD_DECL`, `VAR_DECL`, `CONST_DECL`, or `TYPE_DECL`. You may not make any assumptions about the ordering of the fields in the type or whether one or more of them overlap. If `TYPE_PTRMEMFUNC_P` holds, then this type is a pointer-to-member type. In that case, the `TYPE_PTRMEMFUNC_FN_TYPE` is a `POINTER_TYPE` pointing to a `METHOD_TYPE`. The `METHOD_TYPE` is the type of a function pointed to by the pointer-to-member function. If `TYPE_PTRMEMFUNC_P` does not hold, this type is a class type. For more information, see see Section 9.4.2 [Classes], page 69.

`UNION_TYPE`
: Used to represent `union` types. Similar to `RECORD_TYPE` except that all `FIELD_DECL` nodes in `TYPE_FIELD` start at bit position zero.

`QUAL_UNION_TYPE`
: Used to represent part of a variant record in Ada. Similar to `UNION_TYPE` except that each `FIELD_DECL` has a `DECL_QUALIFIER` field, which contains a boolean expression that indicates whether the field is present in the object. The type will only have one field, so each field's `DECL_QUALIFIER` is only evaluated if none of the expressions in the previous fields in `TYPE_FIELDS` are nonzero. Normally these expressions will reference a field in the outer object using a `PLACEHOLDER_EXPR`.

`UNKNOWN_TYPE`
: This node is used to represent a type the knowledge of which is insufficient for a sound processing.

`OFFSET_TYPE`
: This node is used to represent a pointer-to-data member. For a data member `X::m` the `TYPE_OFFSET_BASETYPE` is `X` and the `TREE_TYPE` is the type of `m`.

`TYPENAME_TYPE`
: Used to represent a construct of the form `typename T::A`. The `TYPE_CONTEXT` is `T`; the `TYPE_NAME` is an `IDENTIFIER_NODE` for `A`. If the type is specified via a template-id, then `TYPENAME_TYPE_FULLNAME` yields a `TEMPLATE_ID_EXPR`. The `TREE_TYPE` is non-`NULL` if the node is implicitly generated in support for the implicit typename extension; in which case the `TREE_TYPE` is a type node for the base-class.

`TYPEOF_TYPE`
: Used to represent the `__typeof__` extension. The `TYPE_FIELDS` is the expression the type of which is being represented.

There are variables whose values represent some of the basic types. These include:

`void_type_node`
: A node for `void`.

`integer_type_node`
: A node for `int`.

`unsigned_type_node.`
: A node for `unsigned int`.

`char_type_node.`
: A node for `char`.

It may sometimes be useful to compare one of these variables with a type in hand, using `same_type_p`.

9.4 Scopes

The root of the entire intermediate representation is the variable `global_namespace`. This is the namespace specified with `::` in C++ source code. All other namespaces, types, variables, functions, and so forth can be found starting with this namespace.

Besides namespaces, the other high-level scoping construct in C++ is the class. (Throughout this manual the term *class* is used to mean the types referred to in the ANSI/ISO C++ Standard as classes; these include types defined with the `class`, `struct`, and `union` keywords.)

9.4.1 Namespaces

A namespace is represented by a `NAMESPACE_DECL` node.

However, except for the fact that it is distinguished as the root of the representation, the global namespace is no different from any other namespace. Thus, in what follows, we describe namespaces generally, rather than the global namespace in particular.

The following macros and functions can be used on a `NAMESPACE_DECL`:

`DECL_NAME`
: This macro is used to obtain the `IDENTIFIER_NODE` corresponding to the unqualified name of the name of the namespace (see Section 9.2.2 [Identifiers], page 62). The name of the global namespace is '`::`', even though in C++ the global namespace is unnamed. However, you should use comparison with `global_namespace`, rather than `DECL_NAME` to determine whether or not a namespace is the global one. An unnamed namespace will have a `DECL_NAME` equal to `anonymous_namespace_name`. Within a single translation unit, all unnamed namespaces will have the same name.

`DECL_CONTEXT`
: This macro returns the enclosing namespace. The `DECL_CONTEXT` for the `global_namespace` is `NULL_TREE`.

`DECL_NAMESPACE_ALIAS`
: If this declaration is for a namespace alias, then `DECL_NAMESPACE_ALIAS` is the namespace for which this one is an alias.

  Do not attempt to use `cp_namespace_decls` for a namespace which is an alias. Instead, follow `DECL_NAMESPACE_ALIAS` links until you reach an ordinary, non-alias, namespace, and call `cp_namespace_decls` there.

`DECL_NAMESPACE_STD_P`
: This predicate holds if the namespace is the special `::std` namespace.

`cp_namespace_decls`
: This function will return the declarations contained in the namespace, including types, overloaded functions, other namespaces, and so forth. If there are no declarations, this function will return `NULL_TREE`. The declarations are connected through their `TREE_CHAIN` fields.

  Although most entries on this list will be declarations, `TREE_LIST` nodes may also appear. In this case, the `TREE_VALUE` will be an `OVERLOAD`. The value of the `TREE_PURPOSE` is unspecified; back ends should ignore this value. As with the other kinds of declarations returned by `cp_namespace_decls`, the `TREE_CHAIN` will point to the next declaration in this list.

  For more information on the kinds of declarations that can occur on this list, See Section 9.5 [Declarations], page 71. Some declarations will not appear on this list. In particular, no `FIELD_DECL`, `LABEL_DECL`, or `PARM_DECL` nodes will appear here.

  This function cannot be used with namespaces that have `DECL_NAMESPACE_ALIAS` set.

9.4.2 Classes

A class type is represented by either a `RECORD_TYPE` or a `UNION_TYPE`. A class declared with the `union` tag is represented by a `UNION_TYPE`, while classes declared with either the `struct` or the `class` tag are represented by `RECORD_TYPE`s. You can use the `CLASSTYPE_DECLARED_CLASS` macro to discern whether or not a particular type is a `class` as opposed to a `struct`. This macro will be true only for classes declared with the `class` tag.

Almost all non-function members are available on the `TYPE_FIELDS` list. Given one member, the next can be found by following the `TREE_CHAIN`. You should not depend in any

way on the order in which fields appear on this list. All nodes on this list will be 'DECL' nodes. A FIELD_DECL is used to represent a non-static data member, a VAR_DECL is used to represent a static data member, and a TYPE_DECL is used to represent a type. Note that the CONST_DECL for an enumeration constant will appear on this list, if the enumeration type was declared in the class. (Of course, the TYPE_DECL for the enumeration type will appear here as well.) There are no entries for base classes on this list. In particular, there is no FIELD_DECL for the "base-class portion" of an object.

The TYPE_VFIELD is a compiler-generated field used to point to virtual function tables. It may or may not appear on the TYPE_FIELDS list. However, back ends should handle the TYPE_VFIELD just like all the entries on the TYPE_FIELDS list.

The function members are available on the TYPE_METHODS list. Again, subsequent members are found by following the TREE_CHAIN field. If a function is overloaded, each of the overloaded functions appears; no OVERLOAD nodes appear on the TYPE_METHODS list. Implicitly declared functions (including default constructors, copy constructors, assignment operators, and destructors) will appear on this list as well.

Every class has an associated *binfo*, which can be obtained with TYPE_BINFO. Binfos are used to represent base-classes. The binfo given by TYPE_BINFO is the degenerate case, whereby every class is considered to be its own base-class. The base binfos for a particular binfo are held in a vector, whose length is obtained with BINFO_N_BASE_BINFOS. The base binfos themselves are obtained with BINFO_BASE_BINFO and BINFO_BASE_ITERATE. To add a new binfo, use BINFO_BASE_APPEND. The vector of base binfos can be obtained with BINFO_BASE_BINFOS, but normally you do not need to use that. The class type associated with a binfo is given by BINFO_TYPE. It is not always the case that BINFO_TYPE (TYPE_BINFO (x)), because of typedefs and qualified types. Neither is it the case that TYPE_BINFO (BINFO_TYPE (y)) is the same binfo as y. The reason is that if y is a binfo representing a base-class B of a derived class D, then BINFO_TYPE (y) will be B, and TYPE_BINFO (BINFO_TYPE (y)) will be B as its own base-class, rather than as a base-class of D.

The access to a base type can be found with BINFO_BASE_ACCESS. This will produce access_public_node, access_private_node or access_protected_node. If bases are always public, BINFO_BASE_ACCESSES may be NULL.

BINFO_VIRTUAL_P is used to specify whether the binfo is inherited virtually or not. The other flags, BINFO_MARKED_P and BINFO_FLAG_1 to BINFO_FLAG_6 can be used for language specific use.

The following macros can be used on a tree node representing a class-type.

LOCAL_CLASS_P
: This predicate holds if the class is local class *i.e.* declared inside a function body.

TYPE_POLYMORPHIC_P
: This predicate holds if the class has at least one virtual function (declared or inherited).

TYPE_HAS_DEFAULT_CONSTRUCTOR
: This predicate holds whenever its argument represents a class-type with default constructor.

`CLASSTYPE_HAS_MUTABLE`
`TYPE_HAS_MUTABLE_P`
: These predicates hold for a class-type having a mutable data member.

`CLASSTYPE_NON_POD_P`
: This predicate holds only for class-types that are not PODs.

`TYPE_HAS_NEW_OPERATOR`
: This predicate holds for a class-type that defines `operator new`.

`TYPE_HAS_ARRAY_NEW_OPERATOR`
: This predicate holds for a class-type for which `operator new[]` is defined.

`TYPE_OVERLOADS_CALL_EXPR`
: This predicate holds for class-type for which the function call `operator()` is overloaded.

`TYPE_OVERLOADS_ARRAY_REF`
: This predicate holds for a class-type that overloads `operator[]`

`TYPE_OVERLOADS_ARROW`
: This predicate holds for a class-type for which `operator->` is overloaded.

9.5 Declarations

This section covers the various kinds of declarations that appear in the internal representation, except for declarations of functions (represented by `FUNCTION_DECL` nodes), which are described in Section 9.6 [Functions], page 76.

9.5.1 Working with declarations

Some macros can be used with any kind of declaration. These include:

`DECL_NAME`
: This macro returns an `IDENTIFIER_NODE` giving the name of the entity.

`TREE_TYPE`
: This macro returns the type of the entity declared.

`TREE_FILENAME`
: This macro returns the name of the file in which the entity was declared, as a `char*`. For an entity declared implicitly by the compiler (like `__builtin_memcpy`), this will be the string `"<internal>"`.

`TREE_LINENO`
: This macro returns the line number at which the entity was declared, as an `int`.

`DECL_ARTIFICIAL`
: This predicate holds if the declaration was implicitly generated by the compiler. For example, this predicate will hold of an implicitly declared member function, or of the `TYPE_DECL` implicitly generated for a class type. Recall that in C++ code like:

```
struct S {};
```

  is roughly equivalent to C code like:

```
struct S {};
typedef struct S S;
```

The implicitly generated `typedef` declaration is represented by a `TYPE_DECL` for which `DECL_ARTIFICIAL` holds.

`DECL_NAMESPACE_SCOPE_P`
: This predicate holds if the entity was declared at a namespace scope.

`DECL_CLASS_SCOPE_P`
: This predicate holds if the entity was declared at a class scope.

`DECL_FUNCTION_SCOPE_P`
: This predicate holds if the entity was declared inside a function body.

The various kinds of declarations include:

`LABEL_DECL`
: These nodes are used to represent labels in function bodies. For more information, see Section 9.6 [Functions], page 76. These nodes only appear in block scopes.

`CONST_DECL`
: These nodes are used to represent enumeration constants. The value of the constant is given by `DECL_INITIAL` which will be an `INTEGER_CST` with the same type as the `TREE_TYPE` of the `CONST_DECL`, i.e., an `ENUMERAL_TYPE`.

`RESULT_DECL`
: These nodes represent the value returned by a function. When a value is assigned to a `RESULT_DECL`, that indicates that the value should be returned, via bitwise copy, by the function. You can use `DECL_SIZE` and `DECL_ALIGN` on a `RESULT_DECL`, just as with a `VAR_DECL`.

`TYPE_DECL`
: These nodes represent `typedef` declarations. The `TREE_TYPE` is the type declared to have the name given by `DECL_NAME`. In some cases, there is no associated name.

`VAR_DECL`
: These nodes represent variables with namespace or block scope, as well as static data members. The `DECL_SIZE` and `DECL_ALIGN` are analogous to `TYPE_SIZE` and `TYPE_ALIGN`. For a declaration, you should always use the `DECL_SIZE` and `DECL_ALIGN` rather than the `TYPE_SIZE` and `TYPE_ALIGN` given by the `TREE_TYPE`, since special attributes may have been applied to the variable to give it a particular size and alignment. You may use the predicates `DECL_THIS_STATIC` or `DECL_THIS_EXTERN` to test whether the storage class specifiers `static` or `extern` were used to declare a variable.

  If this variable is initialized (but does not require a constructor), the `DECL_INITIAL` will be an expression for the initializer. The initializer should be evaluated, and a bitwise copy into the variable performed. If the `DECL_INITIAL` is the `error_mark_node`, there is an initializer, but it is given by an explicit statement later in the code; no bitwise copy is required.

  GCC provides an extension that allows either automatic variables, or global variables, to be placed in particular registers. This extension is being used for

a particular VAR_DECL if DECL_REGISTER holds for the VAR_DECL, and if DECL_ASSEMBLER_NAME is not equal to DECL_NAME. In that case, DECL_ASSEMBLER_NAME is the name of the register into which the variable will be placed.

PARM_DECL
: Used to represent a parameter to a function. Treat these nodes similarly to VAR_DECL nodes. These nodes only appear in the DECL_ARGUMENTS for a FUNCTION_DECL.

  The DECL_ARG_TYPE for a PARM_DECL is the type that will actually be used when a value is passed to this function. It may be a wider type than the TREE_TYPE of the parameter; for example, the ordinary type might be short while the DECL_ARG_TYPE is int.

FIELD_DECL
: These nodes represent non-static data members. The DECL_SIZE and DECL_ALIGN behave as for VAR_DECL nodes. The DECL_FIELD_BITPOS gives the first bit used for this field, as an INTEGER_CST. These values are indexed from zero, where zero indicates the first bit in the object.

  If DECL_C_BIT_FIELD holds, this field is a bit-field.

NAMESPACE_DECL
: See Section 9.4.1 [Namespaces], page 68.

TEMPLATE_DECL
: These nodes are used to represent class, function, and variable (static data member) templates. The DECL_TEMPLATE_SPECIALIZATIONS are a TREE_LIST. The TREE_VALUE of each node in the list is a TEMPLATE_DECLs or FUNCTION_DECLs representing specializations (including instantiations) of this template. Back ends can safely ignore TEMPLATE_DECLs, but should examine FUNCTION_DECL nodes on the specializations list just as they would ordinary FUNCTION_DECL nodes.

  For a class template, the DECL_TEMPLATE_INSTANTIATIONS list contains the instantiations. The TREE_VALUE of each node is an instantiation of the class. The DECL_TEMPLATE_SPECIALIZATIONS contains partial specializations of the class.

USING_DECL
: Back ends can safely ignore these nodes.

9.5.2 Internal structure

DECL nodes are represented internally as a hierarchy of structures.

9.5.2.1 Current structure hierarchy

struct tree_decl_minimal
: This is the minimal structure to inherit from in order for common DECL macros to work. The fields it contains are a unique ID, source location, context, and name.

`struct tree_decl_common`
: This structure inherits from `struct tree_decl_minimal`. It contains fields that most `DECL` nodes need, such as a field to store alignment, machine mode, size, and attributes.

`struct tree_field_decl`
: This structure inherits from `struct tree_decl_common`. It is used to represent `FIELD_DECL`.

`struct tree_label_decl`
: This structure inherits from `struct tree_decl_common`. It is used to represent `LABEL_DECL`.

`struct tree_translation_unit_decl`
: This structure inherits from `struct tree_decl_common`. It is used to represent `TRANSLATION_UNIT_DECL`.

`struct tree_decl_with_rtl`
: This structure inherits from `struct tree_decl_common`. It contains a field to store the low-level RTL associated with a `DECL` node.

`struct tree_result_decl`
: This structure inherits from `struct tree_decl_with_rtl`. It is used to represent `RESULT_DECL`.

`struct tree_const_decl`
: This structure inherits from `struct tree_decl_with_rtl`. It is used to represent `CONST_DECL`.

`struct tree_parm_decl`
: This structure inherits from `struct tree_decl_with_rtl`. It is used to represent `PARM_DECL`.

`struct tree_decl_with_vis`
: This structure inherits from `struct tree_decl_with_rtl`. It contains fields necessary to store visibility information, as well as a section name and assembler name.

`struct tree_var_decl`
: This structure inherits from `struct tree_decl_with_vis`. It is used to represent `VAR_DECL`.

`struct tree_function_decl`
: This structure inherits from `struct tree_decl_with_vis`. It is used to represent `FUNCTION_DECL`.

9.5.2.2 Adding new DECL node types

Adding a new `DECL` tree consists of the following steps

Add a new tree code for the `DECL` node
: For language specific `DECL` nodes, there is a '`.def`' file in each frontend directory where the tree code should be added. For `DECL` nodes that are part of the middle-end, the code should be added to '`tree.def`'.

Create a new structure type for the `DECL` node

These structures should inherit from one of the existing structures in the language hierarchy by using that structure as the first member.

```
struct tree_foo_decl
{
   struct tree_decl_with_vis common;
}
```

Would create a structure name `tree_foo_decl` that inherits from `struct tree_decl_with_vis`.

For language specific `DECL` nodes, this new structure type should go in the appropriate '`.h`' file. For `DECL` nodes that are part of the middle-end, the structure type should go in '`tree.h`'.

Add a member to the tree structure enumerator for the node

For garbage collection and dynamic checking purposes, each `DECL` node structure type is required to have a unique enumerator value specified with it. For language specific `DECL` nodes, this new enumerator value should go in the appropriate '`.def`' file. For `DECL` nodes that are part of the middle-end, the enumerator values are specified in '`treestruct.def`'.

Update `union tree_node`

In order to make your new structure type usable, it must be added to `union tree_node`. For language specific `DECL` nodes, a new entry should be added to the appropriate '`.h`' file of the form

```
struct tree_foo_decl GTY ((tag ("TS_VAR_DECL"))) foo_decl;
```

For `DECL` nodes that are part of the middle-end, the additional member goes directly into `union tree_node` in '`tree.h`'.

Update dynamic checking info

In order to be able to check whether accessing a named portion of `union tree_node` is legal, and whether a certain `DECL` node contains one of the enumerated `DECL` node structures in the hierarchy, a simple lookup table is used. This lookup table needs to be kept up to date with the tree structure hierarchy, or else checking and containment macros will fail inappropriately.

For language specific `DECL` nodes, their is an `init_ts` function in an appropriate '`.c`' file, which initializes the lookup table. Code setting up the table for new `DECL` nodes should be added there. For each `DECL` tree code and enumerator value representing a member of the inheritance hierarchy, the table should contain 1 if that tree code inherits (directly or indirectly) from that member. Thus, a `FOO_DECL` node derived from `struct decl_with_rtl`, and enumerator value `TS_FOO_DECL`, would be set up as follows

```
tree_contains_struct[FOO_DECL][TS_FOO_DECL] = 1;
tree_contains_struct[FOO_DECL][TS_DECL_WRTL] = 1;
tree_contains_struct[FOO_DECL][TS_DECL_COMMON] = 1;
tree_contains_struct[FOO_DECL][TS_DECL_MINIMAL] = 1;
```

For `DECL` nodes that are part of the middle-end, the setup code goes into '`tree.c`'.

Add macros to access any new fields and flags

Each added field or flag should have a macro that is used to access it, that performs appropriate checking to ensure only the right type of `DECL` nodes access the field.

These macros generally take the following form

```
#define FOO_DECL_FIELDNAME(NODE) FOO_DECL_CHECK(NODE)->foo_decl.fieldname
```

However, if the structure is simply a base class for further structures, something like the following should be used

```
#define BASE_STRUCT_CHECK(T) CONTAINS_STRUCT_CHECK(T, TS_BASE_STRUCT)
#define BASE_STRUCT_FIELDNAME(NODE) \
   (BASE_STRUCT_CHECK(NODE)->base_struct.fieldname
```

9.6 Functions

A function is represented by a FUNCTION_DECL node. A set of overloaded functions is sometimes represented by a OVERLOAD node.

An OVERLOAD node is not a declaration, so none of the 'DECL_' macros should be used on an OVERLOAD. An OVERLOAD node is similar to a TREE_LIST. Use OVL_CURRENT to get the function associated with an OVERLOAD node; use OVL_NEXT to get the next OVERLOAD node in the list of overloaded functions. The macros OVL_CURRENT and OVL_NEXT are actually polymorphic; you can use them to work with FUNCTION_DECL nodes as well as with overloads. In the case of a FUNCTION_DECL, OVL_CURRENT will always return the function itself, and OVL_NEXT will always be NULL_TREE.

To determine the scope of a function, you can use the DECL_CONTEXT macro. This macro will return the class (either a RECORD_TYPE or a UNION_TYPE) or namespace (a NAMESPACE_DECL) of which the function is a member. For a virtual function, this macro returns the class in which the function was actually defined, not the base class in which the virtual declaration occurred.

If a friend function is defined in a class scope, the DECL_FRIEND_CONTEXT macro can be used to determine the class in which it was defined. For example, in

```
class C { friend void f() {} };
```

the DECL_CONTEXT for f will be the global_namespace, but the DECL_FRIEND_CONTEXT will be the RECORD_TYPE for C.

In C, the DECL_CONTEXT for a function maybe another function. This representation indicates that the GNU nested function extension is in use. For details on the semantics of nested functions, see the GCC Manual. The nested function can refer to local variables in its containing function. Such references are not explicitly marked in the tree structure; back ends must look at the DECL_CONTEXT for the referenced VAR_DECL. If the DECL_CONTEXT for the referenced VAR_DECL is not the same as the function currently being processed, and neither DECL_EXTERNAL nor DECL_STATIC hold, then the reference is to a local variable in a containing function, and the back end must take appropriate action.

9.6.1 Function Basics

The following macros and functions can be used on a FUNCTION_DECL:

DECL_MAIN_P
: This predicate holds for a function that is the program entry point ::code.

DECL_NAME
: This macro returns the unqualified name of the function, as an IDENTIFIER_NODE. For an instantiation of a function template, the DECL_NAME is the unqualified name of the template, not something like f<int>. The value of DECL_NAME

is undefined when used on a constructor, destructor, overloaded operator, or type-conversion operator, or any function that is implicitly generated by the compiler. See below for macros that can be used to distinguish these cases.

`DECL_ASSEMBLER_NAME`
: This macro returns the mangled name of the function, also an `IDENTIFIER_NODE`. This name does not contain leading underscores on systems that prefix all identifiers with underscores. The mangled name is computed in the same way on all platforms; if special processing is required to deal with the object file format used on a particular platform, it is the responsibility of the back end to perform those modifications. (Of course, the back end should not modify `DECL_ASSEMBLER_NAME` itself.)

  Using `DECL_ASSEMBLER_NAME` will cause additional memory to be allocated (for the mangled name of the entity) so it should be used only when emitting assembly code. It should not be used within the optimizers to determine whether or not two declarations are the same, even though some of the existing optimizers do use it in that way. These uses will be removed over time.

`DECL_EXTERNAL`
: This predicate holds if the function is undefined.

`TREE_PUBLIC`
: This predicate holds if the function has external linkage.

`DECL_LOCAL_FUNCTION_P`
: This predicate holds if the function was declared at block scope, even though it has a global scope.

`DECL_ANTICIPATED`
: This predicate holds if the function is a built-in function but its prototype is not yet explicitly declared.

`DECL_EXTERN_C_FUNCTION_P`
: This predicate holds if the function is declared as an '`extern "C"`' function.

`DECL_LINKONCE_P`
: This macro holds if multiple copies of this function may be emitted in various translation units. It is the responsibility of the linker to merge the various copies. Template instantiations are the most common example of functions for which `DECL_LINKONCE_P` holds; G++ instantiates needed templates in all translation units which require them, and then relies on the linker to remove duplicate instantiations.

  FIXME: This macro is not yet implemented.

`DECL_FUNCTION_MEMBER_P`
: This macro holds if the function is a member of a class, rather than a member of a namespace.

`DECL_STATIC_FUNCTION_P`
: This predicate holds if the function a static member function.

`DECL_NONSTATIC_MEMBER_FUNCTION_P`
: This macro holds for a non-static member function.

`DECL_CONST_MEMFUNC_P`
: This predicate holds for a `const`-member function.

`DECL_VOLATILE_MEMFUNC_P`
: This predicate holds for a `volatile`-member function.

`DECL_CONSTRUCTOR_P`
: This macro holds if the function is a constructor.

`DECL_NONCONVERTING_P`
: This predicate holds if the constructor is a non-converting constructor.

`DECL_COMPLETE_CONSTRUCTOR_P`
: This predicate holds for a function which is a constructor for an object of a complete type.

`DECL_BASE_CONSTRUCTOR_P`
: This predicate holds for a function which is a constructor for a base class sub-object.

`DECL_COPY_CONSTRUCTOR_P`
: This predicate holds for a function which is a copy-constructor.

`DECL_DESTRUCTOR_P`
: This macro holds if the function is a destructor.

`DECL_COMPLETE_DESTRUCTOR_P`
: This predicate holds if the function is the destructor for an object a complete type.

`DECL_OVERLOADED_OPERATOR_P`
: This macro holds if the function is an overloaded operator.

`DECL_CONV_FN_P`
: This macro holds if the function is a type-conversion operator.

`DECL_GLOBAL_CTOR_P`
: This predicate holds if the function is a file-scope initialization function.

`DECL_GLOBAL_DTOR_P`
: This predicate holds if the function is a file-scope finalization function.

`DECL_THUNK_P`
: This predicate holds if the function is a thunk.

  These functions represent stub code that adjusts the `this` pointer and then jumps to another function. When the jumped-to function returns, control is transferred directly to the caller, without returning to the thunk. The first parameter to the thunk is always the `this` pointer; the thunk should add `THUNK_DELTA` to this value. (The `THUNK_DELTA` is an `int`, not an `INTEGER_CST`.)

  Then, if `THUNK_VCALL_OFFSET` (an `INTEGER_CST`) is nonzero the adjusted `this` pointer must be adjusted again. The complete calculation is given by the following pseudo-code:

```
this += THUNK_DELTA
if (THUNK_VCALL_OFFSET)
  this += (*((ptrdiff_t **) this))[THUNK_VCALL_OFFSET]
```

Finally, the thunk should jump to the location given by DECL_INITIAL; this will always be an expression for the address of a function.

DECL_NON_THUNK_FUNCTION_P
: This predicate holds if the function is *not* a thunk function.

GLOBAL_INIT_PRIORITY
: If either DECL_GLOBAL_CTOR_P or DECL_GLOBAL_DTOR_P holds, then this gives the initialization priority for the function. The linker will arrange that all functions for which DECL_GLOBAL_CTOR_P holds are run in increasing order of priority before main is called. When the program exits, all functions for which DECL_GLOBAL_DTOR_P holds are run in the reverse order.

DECL_ARTIFICIAL
: This macro holds if the function was implicitly generated by the compiler, rather than explicitly declared. In addition to implicitly generated class member functions, this macro holds for the special functions created to implement static initialization and destruction, to compute run-time type information, and so forth.

DECL_ARGUMENTS
: This macro returns the PARM_DECL for the first argument to the function. Subsequent PARM_DECL nodes can be obtained by following the TREE_CHAIN links.

DECL_RESULT
: This macro returns the RESULT_DECL for the function.

TREE_TYPE
: This macro returns the FUNCTION_TYPE or METHOD_TYPE for the function.

TYPE_RAISES_EXCEPTIONS
: This macro returns the list of exceptions that a (member-)function can raise. The returned list, if non NULL, is comprised of nodes whose TREE_VALUE represents a type.

TYPE_NOTHROW_P
: This predicate holds when the exception-specification of its arguments if of the form '()'.

DECL_ARRAY_DELETE_OPERATOR_P
: This predicate holds if the function an overloaded operator delete[].

9.6.2 Function Bodies

A function that has a definition in the current translation unit will have a non-NULL DECL_INITIAL. However, back ends should not make use of the particular value given by DECL_INITIAL.

The DECL_SAVED_TREE macro will give the complete body of the function.

9.6.2.1 Statements

There are tree nodes corresponding to all of the source-level statement constructs, used within the C and C++ frontends. These are enumerated here, together with a list of the various macros that can be used to obtain information about them. There are a few macros that can be used with all statements:

`STMT_IS_FULL_EXPR_P`

> In C++, statements normally constitute "full expressions"; temporaries created during a statement are destroyed when the statement is complete. However, G++ sometimes represents expressions by statements; these statements will not have `STMT_IS_FULL_EXPR_P` set. Temporaries created during such statements should be destroyed when the innermost enclosing statement with `STMT_IS_FULL_EXPR_P` set is exited.

Here is the list of the various statement nodes, and the macros used to access them. This documentation describes the use of these nodes in non-template functions (including instantiations of template functions). In template functions, the same nodes are used, but sometimes in slightly different ways.

Many of the statements have substatements. For example, a `while` loop will have a body, which is itself a statement. If the substatement is `NULL_TREE`, it is considered equivalent to a statement consisting of a single `;`, i.e., an expression statement in which the expression has been omitted. A substatement may in fact be a list of statements, connected via their `TREE_CHAIN`s. So, you should always process the statement tree by looping over substatements, like this:

```
void process_stmt (stmt)
     tree stmt;
{
  while (stmt)
    {
      switch (TREE_CODE (stmt))
        {
        case IF_STMT:
          process_stmt (THEN_CLAUSE (stmt));
          /* More processing here.  */
          break;

        ...
        }

      stmt = TREE_CHAIN (stmt);
    }
}
```

In other words, while the `then` clause of an `if` statement in C++ can be only one statement (although that one statement may be a compound statement), the intermediate representation will sometimes use several statements chained together.

`ASM_EXPR`

> Used to represent an inline assembly statement. For an inline assembly statement like:
>
> ```
> asm ("mov x, y");
> ```
>
> The `ASM_STRING` macro will return a `STRING_CST` node for `"mov x, y"`. If the original statement made use of the extended-assembly syntax, then `ASM_OUTPUTS`, `ASM_INPUTS`, and `ASM_CLOBBERS` will be the outputs, inputs, and

clobbers for the statement, represented as STRING_CST nodes. The extended-assembly syntax looks like:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

The first string is the ASM_STRING, containing the instruction template. The next two strings are the output and inputs, respectively; this statement has no clobbers. As this example indicates, "plain" assembly statements are merely a special case of extended assembly statements; they have no cv-qualifiers, outputs, inputs, or clobbers. All of the strings will be NUL-terminated, and will contain no embedded NUL-characters.

If the assembly statement is declared volatile, or if the statement was not an extended assembly statement, and is therefore implicitly volatile, then the predicate ASM_VOLATILE_P will hold of the ASM_EXPR.

BREAK_STMT

Used to represent a break statement. There are no additional fields.

CASE_LABEL_EXPR

Use to represent a case label, range of case labels, or a default label. If CASE_LOW is NULL_TREE, then this is a default label. Otherwise, if CASE_HIGH is NULL_TREE, then this is an ordinary case label. In this case, CASE_LOW is an expression giving the value of the label. Both CASE_LOW and CASE_HIGH are INTEGER_CST nodes. These values will have the same type as the condition expression in the switch statement.

Otherwise, if both CASE_LOW and CASE_HIGH are defined, the statement is a range of case labels. Such statements originate with the extension that allows users to write things of the form:

```
case 2 ... 5:
```

The first value will be CASE_LOW, while the second will be CASE_HIGH.

CLEANUP_STMT

Used to represent an action that should take place upon exit from the enclosing scope. Typically, these actions are calls to destructors for local objects, but back ends cannot rely on this fact. If these nodes are in fact representing such destructors, CLEANUP_DECL will be the VAR_DECL destroyed. Otherwise, CLEANUP_DECL will be NULL_TREE. In any case, the CLEANUP_EXPR is the expression to execute. The cleanups executed on exit from a scope should be run in the reverse order of the order in which the associated CLEANUP_STMTs were encountered.

CONTINUE_STMT

Used to represent a continue statement. There are no additional fields.

CTOR_STMT

Used to mark the beginning (if CTOR_BEGIN_P holds) or end (if CTOR_END_P holds of the main body of a constructor. See also SUBOBJECT for more information on how to use these nodes.

DECL_STMT

Used to represent a local declaration. The DECL_STMT_DECL macro can be used to obtain the entity declared. This declaration may be a LABEL_DECL,

indicating that the label declared is a local label. (As an extension, GCC allows the declaration of labels with scope.) In C, this declaration may be a `FUNCTION_DECL`, indicating the use of the GCC nested function extension. For more information, see Section 9.6 [Functions], page 76.

`DO_STMT`

Used to represent a `do` loop. The body of the loop is given by `DO_BODY` while the termination condition for the loop is given by `DO_COND`. The condition for a `do`-statement is always an expression.

`EMPTY_CLASS_EXPR`

Used to represent a temporary object of a class with no data whose address is never taken. (All such objects are interchangeable.) The `TREE_TYPE` represents the type of the object.

`EXPR_STMT`

Used to represent an expression statement. Use `EXPR_STMT_EXPR` to obtain the expression.

`FOR_STMT`

Used to represent a `for` statement. The `FOR_INIT_STMT` is the initialization statement for the loop. The `FOR_COND` is the termination condition. The `FOR_EXPR` is the expression executed right before the `FOR_COND` on each loop iteration; often, this expression increments a counter. The body of the loop is given by `FOR_BODY`. Note that `FOR_INIT_STMT` and `FOR_BODY` return statements, while `FOR_COND` and `FOR_EXPR` return expressions.

`GOTO_EXPR`

Used to represent a `goto` statement. The `GOTO_DESTINATION` will usually be a `LABEL_DECL`. However, if the "computed goto" extension has been used, the `GOTO_DESTINATION` will be an arbitrary expression indicating the destination. This expression will always have pointer type.

`HANDLER`

Used to represent a C++ `catch` block. The `HANDLER_TYPE` is the type of exception that will be caught by this handler; it is equal (by pointer equality) to `NULL` if this handler is for all types. `HANDLER_PARMS` is the `DECL_STMT` for the catch parameter, and `HANDLER_BODY` is the code for the block itself.

`IF_STMT`

Used to represent an `if` statement. The `IF_COND` is the expression.

If the condition is a `TREE_LIST`, then the `TREE_PURPOSE` is a statement (usually a `DECL_STMT`). Each time the condition is evaluated, the statement should be executed. Then, the `TREE_VALUE` should be used as the conditional expression itself. This representation is used to handle C++ code like this:

```
if (int i = 7) ...
```

where there is a new local variable (or variables) declared within the condition.

The `THEN_CLAUSE` represents the statement given by the `then` condition, while the `ELSE_CLAUSE` represents the statement given by the `else` condition.

LABEL_EXPR
: Used to represent a label. The LABEL_DECL declared by this statement can be obtained with the LABEL_EXPR_LABEL macro. The IDENTIFIER_NODE giving the name of the label can be obtained from the LABEL_DECL with DECL_NAME.

RETURN_INIT
: If the function uses the G++ "named return value" extension, meaning that the function has been defined like:

```
S f(int) return s {...}
```

then there will be a RETURN_INIT. There is never a named returned value for a constructor. The first argument to the RETURN_INIT is the name of the object returned; the second argument is the initializer for the object. The object is initialized when the RETURN_INIT is encountered. The object referred to is the actual object returned; this extension is a manual way of doing the "return-value optimization". Therefore, the object must actually be constructed in the place where the object will be returned.

RETURN_STMT
: Used to represent a `return` statement. The RETURN_EXPR is the expression returned; it will be NULL_TREE if the statement was just

```
return;
```

SUBOBJECT
: In a constructor, these nodes are used to mark the point at which a subobject of `this` is fully constructed. If, after this point, an exception is thrown before a CTOR_STMT with CTOR_END_P set is encountered, the SUBOBJECT_CLEANUP must be executed. The cleanups must be executed in the reverse order in which they appear.

SWITCH_STMT
: Used to represent a `switch` statement. The SWITCH_STMT_COND is the expression on which the switch is occurring. See the documentation for an IF_STMT for more information on the representation used for the condition. The SWITCH_STMT_BODY is the body of the switch statement. The SWITCH_STMT_TYPE is the original type of switch expression as given in the source, before any compiler conversions.

TRY_BLOCK
: Used to represent a `try` block. The body of the try block is given by TRY_STMTS. Each of the catch blocks is a HANDLER node. The first handler is given by TRY_HANDLERS. Subsequent handlers are obtained by following the TREE_CHAIN link from one handler to the next. The body of the handler is given by HANDLER_BODY.

  If CLEANUP_P holds of the TRY_BLOCK, then the TRY_HANDLERS will not be a HANDLER node. Instead, it will be an expression that should be executed if an exception is thrown in the try block. It must rethrow the exception after executing that code. And, if an exception is thrown while the expression is executing, `terminate` must be called.

`USING_STMT`
: Used to represent a `using` directive. The namespace is given by `USING_STMT_NAMESPACE`, which will be a NAMESPACE_DECL. This node is needed inside template functions, to implement using directives during instantiation.

`WHILE_STMT`
: Used to represent a `while` loop. The `WHILE_COND` is the termination condition for the loop. See the documentation for an `IF_STMT` for more information on the representation used for the condition.

  The `WHILE_BODY` is the body of the loop.

9.7 Attributes in trees

Attributes, as specified using the `__attribute__` keyword, are represented internally as a `TREE_LIST`. The `TREE_PURPOSE` is the name of the attribute, as an `IDENTIFIER_NODE`. The `TREE_VALUE` is a `TREE_LIST` of the arguments of the attribute, if any, or `NULL_TREE` if there are no arguments; the arguments are stored as the `TREE_VALUE` of successive entries in the list, and may be identifiers or expressions. The `TREE_CHAIN` of the attribute is the next attribute in a list of attributes applying to the same declaration or type, or `NULL_TREE` if there are no further attributes in the list.

Attributes may be attached to declarations and to types; these attributes may be accessed with the following macros. All attributes are stored in this way, and many also cause other changes to the declaration or type or to other internal compiler data structures.

`tree DECL_ATTRIBUTES (`*tree* *`decl`*`)` [Tree Macro]
: This macro returns the attributes on the declaration *decl*.

`tree TYPE_ATTRIBUTES (`*tree* *`type`*`)` [Tree Macro]
: This macro returns the attributes on the type *type*.

9.8 Expressions

The internal representation for expressions is for the most part quite straightforward. However, there are a few facts that one must bear in mind. In particular, the expression "tree" is actually a directed acyclic graph. (For example there may be many references to the integer constant zero throughout the source program; many of these will be represented by the same expression node.) You should not rely on certain kinds of node being shared, nor should rely on certain kinds of nodes being unshared.

The following macros can be used with all expression nodes:

`TREE_TYPE`
: Returns the type of the expression. This value may not be precisely the same type that would be given the expression in the original program.

In what follows, some nodes that one might expect to always have type `bool` are documented to have either integral or boolean type. At some point in the future, the C front end may also make use of this same intermediate representation, and at this point these nodes will certainly have integral type. The previous sentence is not meant to imply that the C++ front end does not or will not give these nodes integral type.

Below, we list the various kinds of expression nodes. Except where noted otherwise, the operands to an expression are accessed using the `TREE_OPERAND` macro. For example, to access the first operand to a binary plus expression `expr`, use:

```
TREE_OPERAND (expr, 0)
```

As this example indicates, the operands are zero-indexed.

The table below begins with constants, moves on to unary expressions, then proceeds to binary expressions, and concludes with various other kinds of expressions:

`INTEGER_CST`

These nodes represent integer constants. Note that the type of these constants is obtained with `TREE_TYPE`; they are not always of type `int`. In particular, `char` constants are represented with `INTEGER_CST` nodes. The value of the integer constant `e` is given by

```
((TREE_INT_CST_HIGH (e) << HOST_BITS_PER_WIDE_INT)
+ TREE_INST_CST_LOW (e))
```

HOST_BITS_PER_WIDE_INT is at least thirty-two on all platforms. Both `TREE_INT_CST_HIGH` and `TREE_INT_CST_LOW` return a `HOST_WIDE_INT`. The value of an `INTEGER_CST` is interpreted as a signed or unsigned quantity depending on the type of the constant. In general, the expression given above will overflow, so it should not be used to calculate the value of the constant.

The variable `integer_zero_node` is an integer constant with value zero. Similarly, `integer_one_node` is an integer constant with value one. The `size_zero_node` and `size_one_node` variables are analogous, but have type `size_t` rather than `int`.

The function `tree_int_cst_lt` is a predicate which holds if its first argument is less than its second. Both constants are assumed to have the same signedness (i.e., either both should be signed or both should be unsigned.) The full width of the constant is used when doing the comparison; the usual rules about promotions and conversions are ignored. Similarly, `tree_int_cst_equal` holds if the two constants are equal. The `tree_int_cst_sgn` function returns the sign of a constant. The value is `1`, `0`, or `-1` according on whether the constant is greater than, equal to, or less than zero. Again, the signedness of the constant's type is taken into account; an unsigned constant is never less than zero, no matter what its bit-pattern.

`REAL_CST`

FIXME: Talk about how to obtain representations of this constant, do comparisons, and so forth.

`COMPLEX_CST`

These nodes are used to represent complex number constants, that is a `__complex__` whose parts are constant nodes. The `TREE_REALPART` and `TREE_IMAGPART` return the real and the imaginary parts respectively.

`VECTOR_CST`

These nodes are used to represent vector constants, whose parts are constant nodes. Each individual constant node is either an integer or a double constant node. The first operand is a `TREE_LIST` of the constant nodes and is accessed through `TREE_VECTOR_CST_ELTS`.

`STRING_CST`

These nodes represent string-constants. The `TREE_STRING_LENGTH` returns the length of the string, as an `int`. The `TREE_STRING_POINTER` is a `char*` containing the string itself. The string may not be `NUL`-terminated, and it may contain embedded `NUL` characters. Therefore, the `TREE_STRING_LENGTH` includes the trailing `NUL` if it is present.

For wide string constants, the `TREE_STRING_LENGTH` is the number of bytes in the string, and the `TREE_STRING_POINTER` points to an array of the bytes of the string, as represented on the target system (that is, as integers in the target endianness). Wide and non-wide string constants are distinguished only by the `TREE_TYPE` of the `STRING_CST`.

FIXME: The formats of string constants are not well-defined when the target system bytes are not the same width as host system bytes.

`PTRMEM_CST`

These nodes are used to represent pointer-to-member constants. The `PTRMEM_CST_CLASS` is the class type (either a `RECORD_TYPE` or `UNION_TYPE` within which the pointer points), and the `PTRMEM_CST_MEMBER` is the declaration for the pointed to object. Note that the `DECL_CONTEXT` for the `PTRMEM_CST_MEMBER` is in general different from the `PTRMEM_CST_CLASS`. For example, given:

```
struct B { int i; };
struct D : public B {};
int D::*dp = &D::i;
```

The `PTRMEM_CST_CLASS` for `&D::i` is `D`, even though the `DECL_CONTEXT` for the `PTRMEM_CST_MEMBER` is `B`, since `B::i` is a member of `B`, not `D`.

`VAR_DECL`

These nodes represent variables, including static data members. For more information, see Section 9.5 [Declarations], page 71.

`NEGATE_EXPR`

These nodes represent unary negation of the single operand, for both integer and floating-point types. The type of negation can be determined by looking at the type of the expression.

The behavior of this operation on signed arithmetic overflow is controlled by the `flag_wrapv` and `flag_trapv` variables.

`ABS_EXPR` These nodes represent the absolute value of the single operand, for both integer and floating-point types. This is typically used to implement the `abs`, `labs` and `llabs` builtins for integer types, and the `fabs`, `fabsf` and `fabsl` builtins for floating point types. The type of abs operation can be determined by looking at the type of the expression.

This node is not used for complex types. To represent the modulus or complex abs of a complex value, use the `BUILT_IN_CABS`, `BUILT_IN_CABSF` or `BUILT_IN_CABSL` builtins, as used to implement the C99 `cabs`, `cabsf` and `cabsl` built-in functions.

`BIT_NOT_EXPR`

These nodes represent bitwise complement, and will always have integral type. The only operand is the value to be complemented.

`TRUTH_NOT_EXPR`

These nodes represent logical negation, and will always have integral (or boolean) type. The operand is the value being negated. The type of the operand and that of the result are always of `BOOLEAN_TYPE` or `INTEGER_TYPE`.

`PREDECREMENT_EXPR`
`PREINCREMENT_EXPR`
`POSTDECREMENT_EXPR`
`POSTINCREMENT_EXPR`

These nodes represent increment and decrement expressions. The value of the single operand is computed, and the operand incremented or decremented. In the case of `PREDECREMENT_EXPR` and `PREINCREMENT_EXPR`, the value of the expression is the value resulting after the increment or decrement; in the case of `POSTDECREMENT_EXPR` and `POSTINCREMENT_EXPR` is the value before the increment or decrement occurs. The type of the operand, like that of the result, will be either integral, boolean, or floating-point.

`ADDR_EXPR`

These nodes are used to represent the address of an object. (These expressions will always have pointer or reference type.) The operand may be another expression, or it may be a declaration.

As an extension, GCC allows users to take the address of a label. In this case, the operand of the `ADDR_EXPR` will be a `LABEL_DECL`. The type of such an expression is `void*`.

If the object addressed is not an lvalue, a temporary is created, and the address of the temporary is used.

`INDIRECT_REF`

These nodes are used to represent the object pointed to by a pointer. The operand is the pointer being dereferenced; it will always have pointer or reference type.

`FIX_TRUNC_EXPR`

These nodes represent conversion of a floating-point value to an integer. The single operand will have a floating-point type, while the the complete expression will have an integral (or boolean) type. The operand is rounded towards zero.

`FLOAT_EXPR`

These nodes represent conversion of an integral (or boolean) value to a floating-point value. The single operand will have integral type, while the complete expression will have a floating-point type.

FIXME: How is the operand supposed to be rounded? Is this dependent on '`-mieee`'?

`COMPLEX_EXPR`

These nodes are used to represent complex numbers constructed from two expressions of the same (integer or real) type. The first operand is the real part and the second operand is the imaginary part.

`CONJ_EXPR`

These nodes represent the conjugate of their operand.

`REALPART_EXPR`
`IMAGPART_EXPR`
: These nodes represent respectively the real and the imaginary parts of complex numbers (their sole argument).

`NON_LVALUE_EXPR`
: These nodes indicate that their one and only operand is not an lvalue. A back end can treat these identically to the single operand.

`NOP_EXPR` These nodes are used to represent conversions that do not require any code-generation. For example, conversion of a `char*` to an `int*` does not require any code be generated; such a conversion is represented by a `NOP_EXPR`. The single operand is the expression to be converted. The conversion from a pointer to a reference is also represented with a `NOP_EXPR`.

`CONVERT_EXPR`
: These nodes are similar to `NOP_EXPR`s, but are used in those situations where code may need to be generated. For example, if an `int*` is converted to an `int` code may need to be generated on some platforms. These nodes are never used for C++-specific conversions, like conversions between pointers to different classes in an inheritance hierarchy. Any adjustments that need to be made in such cases are always indicated explicitly. Similarly, a user-defined conversion is never represented by a `CONVERT_EXPR`; instead, the function calls are made explicit.

`THROW_EXPR`
: These nodes represent `throw` expressions. The single operand is an expression for the code that should be executed to throw the exception. However, there is one implicit action not represented in that expression; namely the call to `__throw`. This function takes no arguments. If `setjmp/longjmp` exceptions are used, the function `__sjthrow` is called instead. The normal GCC back end uses the function `emit_throw` to generate this code; you can examine this function to see what needs to be done.

`LSHIFT_EXPR`
`RSHIFT_EXPR`
: These nodes represent left and right shifts, respectively. The first operand is the value to shift; it will always be of integral type. The second operand is an expression for the number of bits by which to shift. Right shift should be treated as arithmetic, i.e., the high-order bits should be zero-filled when the expression has unsigned type and filled with the sign bit when the expression has signed type. Note that the result is undefined if the second operand is larger than or equal to the first operand's type size.

`BIT_IOR_EXPR`
`BIT_XOR_EXPR`
`BIT_AND_EXPR`
: These nodes represent bitwise inclusive or, bitwise exclusive or, and bitwise and, respectively. Both operands will always have integral type.

`TRUTH_ANDIF_EXPR`
`TRUTH_ORIF_EXPR`

These nodes represent logical and and logical or, respectively. These operators are not strict; i.e., the second operand is evaluated only if the value of the expression is not determined by evaluation of the first operand. The type of the operands and that of the result are always of `BOOLEAN_TYPE` or `INTEGER_TYPE`.

`TRUTH_AND_EXPR`
`TRUTH_OR_EXPR`
`TRUTH_XOR_EXPR`

These nodes represent logical and, logical or, and logical exclusive or. They are strict; both arguments are always evaluated. There are no corresponding operators in C or C++, but the front end will sometimes generate these expressions anyhow, if it can tell that strictness does not matter. The type of the operands and that of the result are always of `BOOLEAN_TYPE` or `INTEGER_TYPE`.

`PLUS_EXPR`
`MINUS_EXPR`
`MULT_EXPR`

These nodes represent various binary arithmetic operations. Respectively, these operations are addition, subtraction (of the second operand from the first) and multiplication. Their operands may have either integral or floating type, but there will never be case in which one operand is of floating type and the other is of integral type.

The behavior of these operations on signed arithmetic overflow is controlled by the `flag_wrapv` and `flag_trapv` variables.

`RDIV_EXPR`

This node represents a floating point division operation.

`TRUNC_DIV_EXPR`
`FLOOR_DIV_EXPR`
`CEIL_DIV_EXPR`
`ROUND_DIV_EXPR`

These nodes represent integer division operations that return an integer result. `TRUNC_DIV_EXPR` rounds towards zero, `FLOOR_DIV_EXPR` rounds towards negative infinity, `CEIL_DIV_EXPR` rounds towards positive infinity and `ROUND_DIV_EXPR` rounds to the closest integer. Integer division in C and C++ is truncating, i.e. `TRUNC_DIV_EXPR`.

The behavior of these operations on signed arithmetic overflow, when dividing the minimum signed integer by minus one, is controlled by the `flag_wrapv` and `flag_trapv` variables.

`TRUNC_MOD_EXPR`
`FLOOR_MOD_EXPR`
`CEIL_MOD_EXPR`
`ROUND_MOD_EXPR`

These nodes represent the integer remainder or modulus operation. The integer modulus of two operands `a` and `b` is defined as `a - (a/b)*b` where the division

calculated using the corresponding division operator. Hence for `TRUNC_MOD_EXPR` this definition assumes division using truncation towards zero, i.e. `TRUNC_DIV_EXPR`. Integer remainder in C and C++ uses truncating division, i.e. `TRUNC_MOD_EXPR`.

`EXACT_DIV_EXPR`
: The `EXACT_DIV_EXPR` code is used to represent integer divisions where the numerator is known to be an exact multiple of the denominator. This allows the backend to choose between the faster of `TRUNC_DIV_EXPR`, `CEIL_DIV_EXPR` and `FLOOR_DIV_EXPR` for the current target.

`ARRAY_REF`
: These nodes represent array accesses. The first operand is the array; the second is the index. To calculate the address of the memory accessed, you must scale the index by the size of the type of the array elements. The type of these expressions must be the type of a component of the array. The third and fourth operands are used after gimplification to represent the lower bound and component size but should not be used directly; call `array_ref_low_bound` and `array_ref_element_size` instead.

`ARRAY_RANGE_REF`
: These nodes represent access to a range (or "slice") of an array. The operands are the same as that for `ARRAY_REF` and have the same meanings. The type of these expressions must be an array whose component type is the same as that of the first operand. The range of that array type determines the amount of data these expressions access.

`TARGET_MEM_REF`
: These nodes represent memory accesses whose address directly map to an addressing mode of the target architecture. The first argument is `TMR_SYMBOL` and must be a `VAR_DECL` of an object with a fixed address. The second argument is `TMR_BASE` and the third one is `TMR_INDEX`. The fourth argument is `TMR_STEP` and must be an `INTEGER_CST`. The fifth argument is `TMR_OFFSET` and must be an `INTEGER_CST`. Any of the arguments may be NULL if the appropriate component does not appear in the address. Address of the `TARGET_MEM_REF` is determined in the following way.

```
&TMR_SYMBOL + TMR_BASE + TMR_INDEX * TMR_STEP + TMR_OFFSET
```

The sixth argument is the reference to the original memory access, which is preserved for the purposes of the RTL alias analysis. The seventh argument is a tag representing the results of tree level alias analysis.

`LT_EXPR`
`LE_EXPR`
`GT_EXPR`
`GE_EXPR`
`EQ_EXPR`
`NE_EXPR`
: These nodes represent the less than, less than or equal to, greater than, greater than or equal to, equal, and not equal comparison operators. The first and second operand with either be both of integral type or both of floating type.

The result type of these expressions will always be of integral or boolean type. These operations return the result type's zero value for false, and the result type's one value for true.

For floating point comparisons, if we honor IEEE NaNs and either operand is NaN, then `NE_EXPR` always returns true and the remaining operators always return false. On some targets, comparisons against an IEEE NaN, other than equality and inequality, may generate a floating point exception.

`ORDERED_EXPR`
`UNORDERED_EXPR`

These nodes represent non-trapping ordered and unordered comparison operators. These operations take two floating point operands and determine whether they are ordered or unordered relative to each other. If either operand is an IEEE NaN, their comparison is defined to be unordered, otherwise the comparison is defined to be ordered. The result type of these expressions will always be of integral or boolean type. These operations return the result type's zero value for false, and the result type's one value for true.

`UNLT_EXPR`
`UNLE_EXPR`
`UNGT_EXPR`
`UNGE_EXPR`
`UNEQ_EXPR`
`LTGT_EXPR`

These nodes represent the unordered comparison operators. These operations take two floating point operands and determine whether the operands are unordered or are less than, less than or equal to, greater than, greater than or equal to, or equal respectively. For example, `UNLT_EXPR` returns true if either operand is an IEEE NaN or the first operand is less than the second. With the possible exception of `LTGT_EXPR`, all of these operations are guaranteed not to generate a floating point exception. The result type of these expressions will always be of integral or boolean type. These operations return the result type's zero value for false, and the result type's one value for true.

`MODIFY_EXPR`

These nodes represent assignment. The left-hand side is the first operand; the right-hand side is the second operand. The left-hand side will be a `VAR_DECL`, `INDIRECT_REF`, `COMPONENT_REF`, or other lvalue.

These nodes are used to represent not only assignment with '`=`' but also compound assignments (like '`+=`'), by reduction to '`=`' assignment. In other words, the representation for '`i += 3`' looks just like that for '`i = i + 3`'.

`INIT_EXPR`

These nodes are just like `MODIFY_EXPR`, but are used only when a variable is initialized, rather than assigned to subsequently. This means that we can assume that the target of the initialization is not used in computing its own value; any reference to the lhs in computing the rhs is undefined.

COMPONENT_REF
: These nodes represent non-static data member accesses. The first operand is the object (rather than a pointer to it); the second operand is the FIELD_DECL for the data member. The third operand represents the byte offset of the field, but should not be used directly; call component_ref_field_offset instead.

COMPOUND_EXPR
: These nodes represent comma-expressions. The first operand is an expression whose value is computed and thrown away prior to the evaluation of the second operand. The value of the entire expression is the value of the second operand.

COND_EXPR
: These nodes represent ?: expressions. The first operand is of boolean or integral type. If it evaluates to a nonzero value, the second operand should be evaluated, and returned as the value of the expression. Otherwise, the third operand is evaluated, and returned as the value of the expression.

  The second operand must have the same type as the entire expression, unless it unconditionally throws an exception or calls a noreturn function, in which case it should have void type. The same constraints apply to the third operand. This allows array bounds checks to be represented conveniently as (i >= 0 && i < 10) ? i : abort().

  As a GNU extension, the C language front-ends allow the second operand of the ?: operator may be omitted in the source. For example, x ? : 3 is equivalent to x ? x : 3, assuming that x is an expression without side-effects. In the tree representation, however, the second operand is always present, possibly protected by SAVE_EXPR if the first argument does cause side-effects.

CALL_EXPR
: These nodes are used to represent calls to functions, including non-static member functions. The first operand is a pointer to the function to call; it is always an expression whose type is a POINTER_TYPE. The second argument is a TREE_LIST. The arguments to the call appear left-to-right in the list. The TREE_VALUE of each list node contains the expression corresponding to that argument. (The value of TREE_PURPOSE for these nodes is unspecified, and should be ignored.) For non-static member functions, there will be an operand corresponding to the this pointer. There will always be expressions corresponding to all of the arguments, even if the function is declared with default arguments and some arguments are not explicitly provided at the call sites.

STMT_EXPR
: These nodes are used to represent GCC's statement-expression extension. The statement-expression extension allows code like this:

  ```
  int f() { return ({ int j; j = 3; j + 7; }); }
  ```

  In other words, an sequence of statements may occur where a single expression would normally appear. The STMT_EXPR node represents such an expression. The STMT_EXPR_STMT gives the statement contained in the expression. The value of the expression is the value of the last sub-statement in the body. More precisely, the value is the value computed by the last statement nested inside BIND_EXPR, TRY_FINALLY_EXPR, or TRY_CATCH_EXPR. For example, in:

  ```
  ({ 3; })
  ```

the value is 3 while in:

```
({ if (x) { 3; } })
```

there is no value. If the STMT_EXPR does not yield a value, it's type will be void.

BIND_EXPR

These nodes represent local blocks. The first operand is a list of variables, connected via their TREE_CHAIN field. These will never require cleanups. The scope of these variables is just the body of the BIND_EXPR. The body of the BIND_EXPR is the second operand.

LOOP_EXPR

These nodes represent "infinite" loops. The LOOP_EXPR_BODY represents the body of the loop. It should be executed forever, unless an EXIT_EXPR is encountered.

EXIT_EXPR

These nodes represent conditional exits from the nearest enclosing LOOP_EXPR. The single operand is the condition; if it is nonzero, then the loop should be exited. An EXIT_EXPR will only appear within a LOOP_EXPR.

CLEANUP_POINT_EXPR

These nodes represent full-expressions. The single operand is an expression to evaluate. Any destructor calls engendered by the creation of temporaries during the evaluation of that expression should be performed immediately after the expression is evaluated.

CONSTRUCTOR

These nodes represent the brace-enclosed initializers for a structure or array. The first operand is reserved for use by the back end. The second operand is a TREE_LIST. If the TREE_TYPE of the CONSTRUCTOR is a RECORD_TYPE or UNION_TYPE, then the TREE_PURPOSE of each node in the TREE_LIST will be a FIELD_DECL and the TREE_VALUE of each node will be the expression used to initialize that field.

If the TREE_TYPE of the CONSTRUCTOR is an ARRAY_TYPE, then the TREE_PURPOSE of each element in the TREE_LIST will be an INTEGER_CST or a RANGE_EXPR of two INTEGER_CSTs. A single INTEGER_CST indicates which element of the array (indexed from zero) is being assigned to. A RANGE_EXPR indicates an inclusive range of elements to initialize. In both cases the TREE_VALUE is the corresponding initializer. It is re-evaluated for each element of a RANGE_EXPR. If the TREE_PURPOSE is NULL_TREE, then the initializer is for the next available array element.

In the front end, you should not depend on the fields appearing in any particular order. However, in the middle end, fields must appear in declaration order. You should not assume that all fields will be represented. Unrepresented fields will be set to zero.

COMPOUND_LITERAL_EXPR

These nodes represent ISO C99 compound literals. The COMPOUND_LITERAL_EXPR_DECL_STMT is a DECL_STMT containing an anonymous VAR_DECL for the

unnamed object represented by the compound literal; the DECL_INITIAL of that VAR_DECL is a CONSTRUCTOR representing the brace-enclosed list of initializers in the compound literal. That anonymous VAR_DECL can also be accessed directly by the COMPOUND_LITERAL_EXPR_DECL macro.

SAVE_EXPR

A SAVE_EXPR represents an expression (possibly involving side-effects) that is used more than once. The side-effects should occur only the first time the expression is evaluated. Subsequent uses should just reuse the computed value. The first operand to the SAVE_EXPR is the expression to evaluate. The side-effects should be executed where the SAVE_EXPR is first encountered in a depth-first preorder traversal of the expression tree.

TARGET_EXPR

A TARGET_EXPR represents a temporary object. The first operand is a VAR_DECL for the temporary variable. The second operand is the initializer for the temporary. The initializer is evaluated and, if non-void, copied (bitwise) into the temporary. If the initializer is void, that means that it will perform the initialization itself.

Often, a TARGET_EXPR occurs on the right-hand side of an assignment, or as the second operand to a comma-expression which is itself the right-hand side of an assignment, etc. In this case, we say that the TARGET_EXPR is "normal"; otherwise, we say it is "orphaned". For a normal TARGET_EXPR the temporary variable should be treated as an alias for the left-hand side of the assignment, rather than as a new temporary variable.

The third operand to the TARGET_EXPR, if present, is a cleanup-expression (i.e., destructor call) for the temporary. If this expression is orphaned, then this expression must be executed when the statement containing this expression is complete. These cleanups must always be executed in the order opposite to that in which they were encountered. Note that if a temporary is created on one branch of a conditional operator (i.e., in the second or third operand to a COND_EXPR), the cleanup must be run only if that branch is actually executed.

See STMT_IS_FULL_EXPR_P for more information about running these cleanups.

AGGR_INIT_EXPR

An AGGR_INIT_EXPR represents the initialization as the return value of a function call, or as the result of a constructor. An AGGR_INIT_EXPR will only appear as a full-expression, or as the second operand of a TARGET_EXPR. The first operand to the AGGR_INIT_EXPR is the address of a function to call, just as in a CALL_EXPR. The second operand are the arguments to pass that function, as a TREE_LIST, again in a manner similar to that of a CALL_EXPR.

If AGGR_INIT_VIA_CTOR_P holds of the AGGR_INIT_EXPR, then the initialization is via a constructor call. The address of the third operand of the AGGR_INIT_EXPR, which is always a VAR_DECL, is taken, and this value replaces the first argument in the argument list.

In either case, the expression is void.

`VA_ARG_EXPR`

This node is used to implement support for the C/C++ variable argument-list mechanism. It represents expressions like `va_arg (ap, type)`. Its `TREE_TYPE` yields the tree representation for `type` and its sole argument yields the representation for `ap`.

10 Analysis and Optimization of GIMPLE Trees

GCC uses three main intermediate languages to represent the program during compilation: GENERIC, GIMPLE and RTL. GENERIC is a language-independent representation generated by each front end. It is used to serve as an interface between the parser and optimizer. GENERIC is a common representation that is able to represent programs written in all the languages supported by GCC.

GIMPLE and RTL are used to optimize the program. GIMPLE is used for target and language independent optimizations (e.g., inlining, constant propagation, tail call elimination, redundancy elimination, etc). Much like GENERIC, GIMPLE is a language independent, tree based representation. However, it differs from GENERIC in that the GIMPLE grammar is more restrictive: expressions contain no more than 3 operands (except function calls), it has no control flow structures and expressions with side-effects are only allowed on the right hand side of assignments. See the chapter describing GENERIC and GIMPLE for more details.

This chapter describes the data structures and functions used in the GIMPLE optimizers (also known as "tree optimizers" or "middle end"). In particular, it focuses on all the macros, data structures, functions and programming constructs needed to implement optimization passes for GIMPLE.

10.1 GENERIC

The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees. To this end, it was necessary to add a few new tree codes to the back end, but most everything was already there. If you can express it with the codes in `gcc/tree.def`, it's GENERIC.

Early on, there was a great deal of debate about how to think about statements in a tree IL. In GENERIC, a statement is defined as any expression whose value, if any, is ignored. A statement will always have `TREE_SIDE_EFFECTS` set (or it will be discarded), but a non-statement expression may also have side effects. A `CALL_EXPR`, for instance.

It would be possible for some local optimizations to work on the GENERIC form of a function; indeed, the adapted tree inliner works fine on GENERIC, but the current compiler performs inlining after lowering to GIMPLE (a restricted form described in the next section). Indeed, currently the frontends perform this lowering before handing off to `tree_rest_of_compilation`, but this seems inelegant.

If necessary, a front end can use some language-dependent tree codes in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE and doesn't expect them to work with any (hypothetical) optimizers that run before the conversion to GIMPLE. The intermediate representation used while parsing C and C++ looks very little like GENERIC, but the C and C++ gimplifier hooks are perfectly happy to take it as input and spit out GIMPLE.

10.2 GIMPLE

GIMPLE is a simplified subset of GENERIC for use in optimization. The particular subset chosen (and the name) was heavily influenced by the SIMPLE IL used by the McCAT

compiler project at McGill University, though we have made some different choices. For one thing, SIMPLE doesn't support `goto`; a production compiler can't afford that kind of restriction.

GIMPLE retains much of the structure of the parse trees: lexical scopes are represented as containers, rather than markers. However, expressions are broken down into a 3-address form, using temporary variables to hold intermediate values. Also, control structures are lowered to gotos.

In GIMPLE no container node is ever used for its value; if a `COND_EXPR` or `BIND_EXPR` has a value, it is stored into a temporary within the controlled blocks, and that temporary is used in place of the container.

The compiler pass which lowers GENERIC to GIMPLE is referred to as the '`gimplifier`'. The gimplifier works recursively, replacing complex statements with sequences of simple statements.

10.2.1 Interfaces

The tree representation of a function is stored in `DECL_SAVED_TREE`. It is lowered to GIMPLE by a call to `gimplify_function_tree`.

If a front end wants to include language-specific tree codes in the tree representation which it provides to the back end, it must provide a definition of `LANG_HOOKS_GIMPLIFY_EXPR` which knows how to convert the front end trees to GIMPLE. Usually such a hook will involve much of the same code for expanding front end trees to RTL. This function can return fully lowered GIMPLE, or it can return GENERIC trees and let the main gimplifier lower them the rest of the way; this is often simpler.

The C and C++ front ends currently convert directly from front end trees to GIMPLE, and hand that off to the back end rather than first converting to GENERIC. Their gimplifier hooks know about all the `_STMT` nodes and how to convert them to GENERIC forms. There was some work done on a genericization pass which would run first, but the existence of `STMT_EXPR` meant that in order to convert all of the C statements into GENERIC equivalents would involve walking the entire tree anyway, so it was simpler to lower all the way. This might change in the future if someone writes an optimization pass which would work better with higher-level trees, but currently the optimizers all expect GIMPLE.

A front end which wants to use the tree optimizers (and already has some sort of whole-function tree representation) only needs to provide a definition of `LANG_HOOKS_GIMPLIFY_EXPR`, call `gimplify_function_tree` to lower to GIMPLE, and then hand off to `tree_rest_of_compilation` to compile and output the function.

You can tell the compiler to dump a C-like representation of the GIMPLE form with the flag '`-fdump-tree-gimple`'.

10.2.2 Temporaries

When gimplification encounters a subexpression which is too complex, it creates a new temporary variable to hold the value of the subexpression, and adds a new statement to initialize it before the current statement. These special temporaries are known as '`expression temporaries`', and are allocated using `get_formal_tmp_var`. The compiler tries to always evaluate identical expressions into the same temporary, to simplify elimination of redundant calculations.

We can only use expression temporaries when we know that it will not be reevaluated before its value is used, and that it will not be otherwise modified[1]. Other temporaries can be allocated using `get_initialized_tmp_var` or `create_tmp_var`.

Currently, an expression like `a = b + 5` is not reduced any further. We tried converting it to something like

```
T1 = b + 5;
a = T1;
```

but this bloated the representation for minimal benefit. However, a variable which must live in memory cannot appear in an expression; its value is explicitly loaded into a temporary first. Similarly, storing the value of an expression to a memory variable goes through a temporary.

10.2.3 Expressions

In general, expressions in GIMPLE consist of an operation and the appropriate number of simple operands; these operands must either be a GIMPLE rvalue (`is_gimple_val`), i.e. a constant or a register variable. More complex operands are factored out into temporaries, so that

```
a = b + c + d
```

becomes

```
T1 = b + c;
a = T1 + d;
```

The same rule holds for arguments to a `CALL_EXPR`.

The target of an assignment is usually a variable, but can also be an `INDIRECT_REF` or a compound lvalue as described below.

10.2.3.1 Compound Expressions

The left-hand side of a C comma expression is simply moved into a separate statement.

10.2.3.2 Compound Lvalues

Currently compound lvalues involving array and structure field references are not broken down; an expression like `a.b[2] = 42` is not reduced any further (though complex array subscripts are). This restriction is a workaround for limitations in later optimizers; if we were to convert this to

```
T1 = &a.b;
T1[2] = 42;
```

alias analysis would not remember that the reference to `T1[2]` came by way of `a.b`, so it would think that the assignment could alias another member of `a`; this broke `struct-alias-1.c`. Future optimizer improvements may make this limitation unnecessary.

10.2.3.3 Conditional Expressions

A C `?:` expression is converted into an `if` statement with each branch assigning to the same temporary. So,

```
a = b ? c : d;
```

becomes

[1] These restrictions are derived from those in Morgan 4.8.

```
if (b)
  T1 = c;
else
  T1 = d;
a = T1;
```

Tree level if-conversion pass re-introduces `?:` expression, if appropriate. It is used to vectorize loops with conditions using vector conditional operations.

Note that in GIMPLE, `if` statements are also represented using `COND_EXPR`, as described below.

10.2.3.4 Logical Operators

Except when they appear in the condition operand of a `COND_EXPR`, logical 'and' and 'or' operators are simplified as follows: `a = b && c` becomes

```
T1 = (bool)b;
if (T1)
  T1 = (bool)c;
a = T1;
```

Note that `T1` in this example cannot be an expression temporary, because it has two different assignments.

10.2.4 Statements

Most statements will be assignment statements, represented by `MODIFY_EXPR`. A `CALL_EXPR` whose value is ignored can also be a statement. No other C expressions can appear at statement level; a reference to a volatile object is converted into a `MODIFY_EXPR`. In GIMPLE form, type of `MODIFY_EXPR` is not meaningful. Instead, use type of LHS or RHS.

There are also several varieties of complex statements.

10.2.4.1 Blocks

Block scopes and the variables they declare in GENERIC and GIMPLE are expressed using the `BIND_EXPR` code, which in previous versions of GCC was primarily used for the C statement-expression extension.

Variables in a block are collected into `BIND_EXPR_VARS` in declaration order. Any runtime initialization is moved out of `DECL_INITIAL` and into a statement in the controlled block. When gimplifying from C or C++, this initialization replaces the `DECL_STMT`.

Variable-length arrays (VLAs) complicate this process, as their size often refers to variables initialized earlier in the block. To handle this, we currently split the block at that point, and move the VLA into a new, inner `BIND_EXPR`. This strategy may change in the future.

`DECL_SAVED_TREE` for a GIMPLE function will always be a `BIND_EXPR` which contains declarations for the temporary variables used in the function.

A C++ program will usually contain more `BIND_EXPR`s than there are syntactic blocks in the source code, since several C++ constructs have implicit scopes associated with them. On the other hand, although the C++ front end uses pseudo-scopes to handle cleanups for objects with destructors, these don't translate into the GIMPLE form; multiple declarations at the same level use the same `BIND_EXPR`.

10.2.4.2 Statement Sequences

Multiple statements at the same nesting level are collected into a `STATEMENT_LIST`. Statement lists are modified and traversed using the interface in '`tree-iterator.h`'.

10.2.4.3 Empty Statements

Whenever possible, statements with no effect are discarded. But if they are nested within another construct which cannot be discarded for some reason, they are instead replaced with an empty statement, generated by `build_empty_stmt`. Initially, all empty statements were shared, after the pattern of the Java front end, but this caused a lot of trouble in practice.

An empty statement is represented as `(void)0`.

10.2.4.4 Loops

At one time loops were expressed in GIMPLE using `LOOP_EXPR`, but now they are lowered to explicit gotos.

10.2.4.5 Selection Statements

A simple selection statement, such as the C `if` statement, is expressed in GIMPLE using a void `COND_EXPR`. If only one branch is used, the other is filled with an empty statement.

Normally, the condition expression is reduced to a simple comparison. If it is a shortcut (`&&` or `||`) expression, however, we try to break up the `if` into multiple `if`s so that the implied shortcut is taken directly, much like the transformation done by `do_jump` in the RTL expander.

A `SWITCH_EXPR` in GIMPLE contains the condition and a `TREE_VEC` of `CASE_LABEL_EXPR`s describing the case values and corresponding `LABEL_DECL`s to jump to. The body of the `switch` is moved after the `SWITCH_EXPR`.

10.2.4.6 Jumps

Other jumps are expressed by either `GOTO_EXPR` or `RETURN_EXPR`.

The operand of a `GOTO_EXPR` must be either a label or a variable containing the address to jump to.

The operand of a `RETURN_EXPR` is either `NULL_TREE` or a `MODIFY_EXPR` which sets the return value. It would be nice to move the `MODIFY_EXPR` into a separate statement, but the special return semantics in `expand_return` make that difficult. It may still happen in the future, perhaps by moving most of that logic into `expand_assignment`.

10.2.4.7 Cleanups

Destructors for local C++ objects and similar dynamic cleanups are represented in GIMPLE by a `TRY_FINALLY_EXPR`. `TRY_FINALLY_EXPR` has two operands, both of which are a sequence of statements to execute. The first sequence is executed. When it completes the second sequence is executed.

The first sequence may complete in the following ways:

1. Execute the last statement in the sequence and fall off the end.
2. Execute a goto statement (`GOTO_EXPR`) to an ordinary label outside the sequence.

3. Execute a return statement (`RETURN_EXPR`).
4. Throw an exception. This is currently not explicitly represented in GIMPLE.

The second sequence is not executed if the first sequence completes by calling `setjmp` or `exit` or any other function that does not return. The second sequence is also not executed if the first sequence completes via a non-local goto or a computed goto (in general the compiler does not know whether such a goto statement exits the first sequence or not, so we assume that it doesn't).

After the second sequence is executed, if it completes normally by falling off the end, execution continues wherever the first sequence would have continued, by falling off the end, or doing a goto, etc.

`TRY_FINALLY_EXPR` complicates the flow graph, since the cleanup needs to appear on every edge out of the controlled block; this reduces the freedom to move code across these edges. Therefore, the EH lowering pass which runs before most of the optimization passes eliminates these expressions by explicitly adding the cleanup to each edge. Rethrowing the exception is represented using `RESX_EXPR`.

10.2.4.8 Exception Handling

Other exception handling constructs are represented using `TRY_CATCH_EXPR`. `TRY_CATCH_EXPR` has two operands. The first operand is a sequence of statements to execute. If executing these statements does not throw an exception, then the second operand is ignored. Otherwise, if an exception is thrown, then the second operand of the `TRY_CATCH_EXPR` is checked. The second operand may have the following forms:

1. A sequence of statements to execute. When an exception occurs, these statements are executed, and then the exception is rethrown.
2. A sequence of `CATCH_EXPR` expressions. Each `CATCH_EXPR` has a list of applicable exception types and handler code. If the thrown exception matches one of the caught types, the associated handler code is executed. If the handler code falls off the bottom, execution continues after the original `TRY_CATCH_EXPR`.
3. An `EH_FILTER_EXPR` expression. This has a list of permitted exception types, and code to handle a match failure. If the thrown exception does not match one of the allowed types, the associated match failure code is executed. If the thrown exception does match, it continues unwinding the stack looking for the next handler.

Currently throwing an exception is not directly represented in GIMPLE, since it is implemented by calling a function. At some point in the future we will want to add some way to express that the call will throw an exception of a known type.

Just before running the optimizers, the compiler lowers the high-level EH constructs above into a set of '`goto`'s, magic labels, and EH regions. Continuing to unwind at the end of a cleanup is represented with a `RESX_EXPR`.

10.2.5 GIMPLE Example

```
struct A { A(); ~A(); };

int i;
int g();
void f()
```

```
{
  A a;
  int j = (--i, i ? 0 : 1);

  for (int x = 42; x > 0; --x)
    {
      i += g()*4 + 32;
    }
}
```

becomes

```
void f()
{
  int i.0;
  int T.1;
  int iftmp.2;
  int T.3;
  int T.4;
  int T.5;
  int T.6;

  {
    struct A a;
    int j;

    __comp_ctor (&a);
    try
      {
        i.0 = i;
        T.1 = i.0 - 1;
        i = T.1;
        i.0 = i;
        if (i.0 == 0)
          iftmp.2 = 1;
        else
          iftmp.2 = 0;
        j = iftmp.2;
        {
          int x;

          x = 42;
          goto test;
          loop:;

          T.3 = g ();
          T.4 = T.3 * 4;
          i.0 = i;
          T.5 = T.4 + i.0;
          T.6 = T.5 + 32;
          i = T.6;
          x = x - 1;

          test:;
          if (x > 0)
            goto loop;
          else
            goto break_;
          break_:;
        }
```

```
    }
   finally
    {
      __comp_dtor (&a);
    }
  }
}
```

10.2.6 Rough GIMPLE Grammar

```
function     : FUNCTION_DECL
                      DECL_SAVED_TREE -> compound-stmt

compound-stmt: STATEMENT_LIST
                      members -> stmt

stmt         : block
             | if-stmt
             | switch-stmt
             | goto-stmt
             | return-stmt
             | resx-stmt
             | label-stmt
             | try-stmt
             | modify-stmt
             | call-stmt

block        : BIND_EXPR
                      BIND_EXPR_VARS -> chain of DECLs
                      BIND_EXPR_BLOCK -> BLOCK
                      BIND_EXPR_BODY -> compound-stmt

if-stmt      : COND_EXPR
                      op0 -> condition
                      op1 -> compound-stmt
                      op2 -> compound-stmt

switch-stmt  : SWITCH_EXPR
                      op0 -> val
                      op1 -> NULL
                      op2 -> TREE_VEC of CASE_LABEL_EXPRs
                          The CASE_LABEL_EXPRs are sorted by CASE_LOW,
                          and default is last.

goto-stmt    : GOTO_EXPR
                      op0 -> LABEL_DECL | val

return-stmt  : RETURN_EXPR
                      op0 -> return-value

return-value : NULL
             | RESULT_DECL
             | MODIFY_EXPR
                      op0 -> RESULT_DECL
                      op1 -> lhs

resx-stmt    : RESX_EXPR

label-stmt   : LABEL_EXPR
```

```
                    op0 -> LABEL_DECL

try-stmt     : TRY_CATCH_EXPR
                    op0 -> compound-stmt
                    op1 -> handler
             | TRY_FINALLY_EXPR
                    op0 -> compound-stmt
                    op1 -> compound-stmt

handler      : catch-seq
             | EH_FILTER_EXPR
             | compound-stmt

catch-seq    : STATEMENT_LIST
                    members -> CATCH_EXPR

modify-stmt  : MODIFY_EXPR
                    op0 -> lhs
                    op1 -> rhs

call-stmt    : CALL_EXPR
                    op0 -> val | OBJ_TYPE_REF
                    op1 -> call-arg-list

call-arg-list: TREE_LIST
                    members -> lhs | CONST

addr-expr-arg: ID
             | compref

addressable  : addr-expr-arg
             | indirectref

with-size-arg: addressable
             | call-stmt

indirectref  : INDIRECT_REF
                    op0 -> val

lhs          : addressable
             | bitfieldref
             | WITH_SIZE_EXPR
                    op0 -> with-size-arg
                    op1 -> val

min-lval     : ID
             | indirectref

bitfieldref  : BIT_FIELD_REF
                    op0 -> inner-compref
                    op1 -> CONST
                    op2 -> var

compref      : inner-compref
             | TARGET_MEM_REF
                    op0 -> ID
                    op1 -> val
                    op2 -> val
```

```
                        op3 -> CONST
                        op4 -> CONST
                | REALPART_EXPR
                        op0 -> inner-compref
                | IMAGPART_EXPR
                        op0 -> inner-compref

    inner-compref: min-lval
                | COMPONENT_REF
                        op0 -> inner-compref
                        op1 -> FIELD_DECL
                        op2 -> val
                | ARRAY_REF
                        op0 -> inner-compref
                        op1 -> val
                        op2 -> val
                        op3 -> val
                | ARRAY_RANGE_REF
                        op0 -> inner-compref
                        op1 -> val
                        op2 -> val
                        op3 -> val
                | VIEW_CONVERT_EXPR
                        op0 -> inner-compref

    condition   : val
                | RELOP
                        op0 -> val
                        op1 -> val

    val         : ID
                | CONST

    rhs         : lhs
                | CONST
                | call-stmt
                | ADDR_EXPR
                        op0 -> addr-expr-arg
                | UNOP
                        op0 -> val
                | BINOP
                        op0 -> val
                        op1 -> val
                | RELOP
                        op0 -> val
                        op1 -> val
| COND_EXPR
op0 -> condition
op1 -> val
op2 -> val
```

10.3 Annotations

The optimizers need to associate attributes with statements and variables during the optimization process. For instance, we need to know what basic block a statement belongs to or whether a variable has aliases. All these attributes are stored in data structures called annotations which are then linked to the field `ann` in `struct tree_common`.

Presently, we define annotations for statements (`stmt_ann_t`), variables (`var_ann_t`) and SSA names (`ssa_name_ann_t`). Annotations are defined and documented in '`tree-flow.h`'.

10.4 Statement Operands

Almost every GIMPLE statement will contain a reference to a variable or memory location. Since statements come in different shapes and sizes, their operands are going to be located at various spots inside the statement's tree. To facilitate access to the statement's operands, they are organized into lists associated inside each statement's annotation. Each element in an operand list is a pointer to a `VAR_DECL`, `PARM_DECL` or `SSA_NAME` tree node. This provides a very convenient way of examining and replacing operands.

Data flow analysis and optimization is done on all tree nodes representing variables. Any node for which `SSA_VAR_P` returns nonzero is considered when scanning statement operands. However, not all `SSA_VAR_P` variables are processed in the same way. For the purposes of optimization, we need to distinguish between references to local scalar variables and references to globals, statics, structures, arrays, aliased variables, etc. The reason is simple, the compiler can gather complete data flow information for a local scalar. On the other hand, a global variable may be modified by a function call, it may not be possible to keep track of all the elements of an array or the fields of a structure, etc.

The operand scanner gathers two kinds of operands: *real* and *virtual*. An operand for which `is_gimple_reg` returns true is considered real, otherwise it is a virtual operand. We also distinguish between uses and definitions. An operand is used if its value is loaded by the statement (e.g., the operand at the RHS of an assignment). If the statement assigns a new value to the operand, the operand is considered a definition (e.g., the operand at the LHS of an assignment).

Virtual and real operands also have very different data flow properties. Real operands are unambiguous references to the full object that they represent. For instance, given

```
{
  int a, b;
  a = b
}
```

Since `a` and `b` are non-aliased locals, the statement `a = b` will have one real definition and one real use because variable `b` is completely modified with the contents of variable `a`. Real definition are also known as *killing definitions*. Similarly, the use of `a` reads all its bits.

In contrast, virtual operands are used with variables that can have a partial or ambiguous reference. This includes structures, arrays, globals, and aliased variables. In these cases, we have two types of definitions. For globals, structures, and arrays, we can determine from a statement whether a variable of these types has a killing definition. If the variable does, then the statement is marked as having a *must definition* of that variable. However, if a statement is only defining a part of the variable (i.e. a field in a structure), or if we know that a statement might define the variable but we cannot say for sure, then we mark that statement as having a *may definition*. For instance, given

```
{
  int a, b, *p;

  if (...)
    p = &a;
  else
```

```
    p = &b;
  *p = 5;
  return *p;
}
```

The assignment `*p = 5` may be a definition of `a` or `b`. If we cannot determine statically where `p` is pointing to at the time of the store operation, we create virtual definitions to mark that statement as a potential definition site for `a` and `b`. Memory loads are similarly marked with virtual use operands. Virtual operands are shown in tree dumps right before the statement that contains them. To request a tree dump with virtual operands, use the '`-vops`' option to '`-fdump-tree`':

```
{
  int a, b, *p;

  if (...)
    p = &a;
  else
    p = &b;
  # a = V_MAY_DEF <a>
  # b = V_MAY_DEF <b>
  *p = 5;

  # VUSE <a>
  # VUSE <b>
  return *p;
}
```

Notice that `V_MAY_DEF` operands have two copies of the referenced variable. This indicates that this is not a killing definition of that variable. In this case we refer to it as a *may definition* or *aliased store*. The presence of the second copy of the variable in the `V_MAY_DEF` operand will become important when the function is converted into SSA form. This will be used to link all the non-killing definitions to prevent optimizations from making incorrect assumptions about them.

Operands are updated as soon as the statement is finished via a call to `update_stmt`. If statement elements are changed via `SET_USE` or `SET_DEF`, then no further action is required (ie, those macros take care of updating the statement). If changes are made by manipulating the statement's tree directly, then a call must be made to `update_stmt` when complete. Calling one of the `bsi_insert` routines or `bsi_replace` performs an implicit call to `update_stmt`.

10.4.1 Operand Iterators And Access Routines

Operands are collected by '`tree-ssa-operands.c`'. They are stored inside each statement's annotation and can be accessed through either the operand iterators or an access routine.

The following access routines are available for examining operands:

1. `SINGLE_SSA_{USE,DEF,TREE}_OPERAND`: These accessors will return NULL unless there is exactly one operand matching the specified flags. If there is exactly one operand, the operand is returned as either a `tree`, `def_operand_p`, or `use_operand_p`.

   ```
   tree t = SINGLE_SSA_TREE_OPERAND (stmt, flags);
   use_operand_p u = SINGLE_SSA_USE_OPERAND (stmt, SSA_ALL_VIRTUAL_USES);
   def_operand_p d = SINGLE_SSA_DEF_OPERAND (stmt, SSA_OP_ALL_DEFS);
   ```

2. `ZERO_SSA_OPERANDS`: This macro returns true if there are no operands matching the specified flags.

```
if (ZERO_SSA_OPERANDS (stmt, SSA_OP_ALL_VIRTUALS))
  return;
```

3. NUM_SSA_OPERANDS: This macro Returns the number of operands matching 'flags'. This actually executes a loop to perform the count, so only use this if it is really needed.

```
int count = NUM_SSA_OPERANDS (stmt, flags)
```

If you wish to iterate over some or all operands, use the FOR_EACH_SSA_{USE,DEF,TREE}_OPERAND iterator. For example, to print all the operands for a statement:

```
void
print_ops (tree stmt)
{
  ssa_op_iter;
  tree var;

  FOR_EACH_SSA_TREE_OPERAND (var, stmt, iter, SSA_OP_ALL_OPERANDS)
    print_generic_expr (stderr, var, TDF_SLIM);
}
```

How to choose the appropriate iterator:

1. Determine whether you are need to see the operand pointers, or just the trees, and choose the appropriate macro:

```
Need            Macro:
----            -------
use_operand_p   FOR_EACH_SSA_USE_OPERAND
def_operand_p   FOR_EACH_SSA_DEF_OPERAND
tree            FOR_EACH_SSA_TREE_OPERAND
```

2. You need to declare a variable of the type you are interested in, and an ssa_op_iter structure which serves as the loop controlling variable.
3. Determine which operands you wish to use, and specify the flags of those you are interested in. They are documented in 'tree-ssa-operands.h':

```
#define SSA_OP_USE              0x01    /* Real USE operands.  */
#define SSA_OP_DEF              0x02    /* Real DEF operands.  */
#define SSA_OP_VUSE             0x04    /* VUSE operands.  */
#define SSA_OP_VMAYUSE          0x08    /* USE portion of V_MAY_DEFS.  */
#define SSA_OP_VMAYDEF          0x10    /* DEF portion of V_MAY_DEFS.  */
#define SSA_OP_VMUSTDEF         0x20    /* V_MUST_DEF definitions.  */

/* These are commonly grouped operand flags.  */
#define SSA_OP_VIRTUAL_USES     (SSA_OP_VUSE | SSA_OP_VMAYUSE)
#define SSA_OP_VIRTUAL_DEFS     (SSA_OP_VMAYDEF | SSA_OP_VMUSTDEF)
#define SSA_OP_ALL_USES         (SSA_OP_VIRTUAL_USES | SSA_OP_USE)
#define SSA_OP_ALL_DEFS         (SSA_OP_VIRTUAL_DEFS | SSA_OP_DEF)
#define SSA_OP_ALL_OPERANDS     (SSA_OP_ALL_USES | SSA_OP_ALL_DEFS)
```

So if you want to look at the use pointers for all the USE and VUSE operands, you would do something like:

```
use_operand_p use_p;
ssa_op_iter iter;

FOR_EACH_SSA_USE_OPERAND (use_p, stmt, iter, (SSA_OP_USE | SSA_OP_VUSE))
  {
    process_use_ptr (use_p);
  }
```

The TREE macro is basically the same as the USE and DEF macros, only with the use or def dereferenced via USE_FROM_PTR (use_p) and DEF_FROM_PTR (def_p). Since we aren't using operand pointers, use and defs flags can be mixed.

```
tree var;
ssa_op_iter iter;

FOR_EACH_SSA_TREE_OPERAND (var, stmt, iter, SSA_OP_VUSE | SSA_OP_VMUSTDEF)
  {
     print_generic_expr (stderr, var, TDF_SLIM);
  }
```

V_MAY_DEFs are broken into two flags, one for the DEF portion (SSA_OP_VMAYDEF) and one for the USE portion (SSA_OP_VMAYUSE). If all you want to look at are the V_MAY_DEFs together, there is a fourth iterator macro for this, which returns both a def_operand_p and a use_operand_p for each V_MAY_DEF in the statement. Note that you don't need any flags for this one.

```
use_operand_p use_p;
def_operand_p def_p;
ssa_op_iter iter;

FOR_EACH_SSA_MAYDEF_OPERAND (def_p, use_p, stmt, iter)
  {
    my_code;
  }
```

V_MUST_DEFs are broken into two flags, one for the DEF portion (SSA_OP_VMUSTDEF) and one for the kill portion (SSA_OP_VMUSTKILL). If all you want to look at are the V_MUST_DEFs together, there is a fourth iterator macro for this, which returns both a def_operand_p and a use_operand_p for each V_MUST_DEF in the statement. Note that you don't need any flags for this one.

```
use_operand_p kill_p;
def_operand_p def_p;
ssa_op_iter iter;

FOR_EACH_SSA_MUSTDEF_OPERAND (def_p, kill_p, stmt, iter)
  {
    my_code;
  }
```

There are many examples in the code as well, as well as the documentation in 'tree-ssa-operands.h'.

There are also a couple of variants on the stmt iterators regarding PHI nodes.

FOR_EACH_PHI_ARG Works exactly like FOR_EACH_SSA_USE_OPERAND, except it works over PHI arguments instead of statement operands.

```
/* Look at every virtual PHI use.  */
FOR_EACH_PHI_ARG (use_p, phi_stmt, iter, SSA_OP_VIRTUAL_USES)
{
   my_code;
}

/* Look at every real PHI use.  */
FOR_EACH_PHI_ARG (use_p, phi_stmt, iter, SSA_OP_USES)
  my_code;

/* Look at every every PHI use.  */
```

```
FOR_EACH_PHI_ARG (use_p, phi_stmt, iter, SSA_OP_ALL_USES)
  my_code;
```

FOR_EACH_PHI_OR_STMT_{USE,DEF} works exactly like FOR_EACH_SSA_{USE,DEF}_OPERAND, except it will function on either a statement or a PHI node. These should be used when it is appropriate but they are not quite as efficient as the individual FOR_EACH_PHI and FOR_EACH_SSA routines.

```
FOR_EACH_PHI_OR_STMT_USE (use_operand_p, stmt, iter, flags)
  {
     my_code;
  }

FOR_EACH_PHI_OR_STMT_DEF (def_operand_p, phi, iter, flags)
  {
     my_code;
  }
```

10.4.2 Immediate Uses

Immediate use information is now always available. Using the immediate use iterators, you may examine every use of any SSA_NAME. For instance, to change each use of ssa_var to ssa_var2:

```
use_operand_p imm_use_p;
imm_use_iterator iterator;
tree ssa_var

FOR_EACH_IMM_USE_SAFE (imm_use_p, iterator, ssa_var)
  SET_USE (imm_use_p, ssa_var_2);
```

There are 2 iterators which can be used. FOR_EACH_IMM_USE_FAST is used when the immediate uses are not changed, ie. you are looking at the uses, but not setting them.

If they do get changed, then care must be taken that things are not changed under the iterators, so use the FOR_EACH_IMM_USE_SAFE iterator. It attempts to preserve the sanity of the use list by moving an iterator element through the use list, preventing insertions and deletions in the list from resulting in invalid pointers. This is a little slower since it adds a placeholder element and moves it through the list. This element must be also be removed if the loop is terminated early. A macro (BREAK_FROM SAFE_IMM_USE) is provided for this:

```
FOR_EACH_IMM_USE_SAFE (use_p, iter, var)
  {
    if (var == last_var)
      BREAK_FROM_SAFE_IMM_USE (iter);
    else
      SET_USE (use_p, var2);
  }
```

There are checks in verify_ssa which verify that the immediate use list is up to date, as well as checking that an optimization didn't break from the loop without using this macro. It is safe to simply 'break'; from a FOR_EACH_IMM_USE_FAST traverse.

Some useful functions and macros:

1. has_zero_uses (ssa_var) : Returns true if there are no uses of ssa_var.
2. has_single_use (ssa_var) : Returns true if there is only a single use of ssa_var.
3. single_imm_use (ssa_var, use_operand_p *ptr, tree *stmt) : Returns true if there is only a single use of ssa_var, and also returns the use pointer and statement it occurs in in the second and third parameters.

4. `num_imm_uses (ssa_var)` : Returns the number of immediate uses of `ssa_var`. It is better not to use this if possible since it simply utilizes a loop to count the uses.
5. `PHI_ARG_INDEX_FROM_USE (use_p)` : Given a use within a `PHI` node, return the index number for the use. An assert is triggered if the use isn't located in a `PHI` node.
6. `USE_STMT (use_p)` : Return the statement a use occurs in.

Note that uses are not put into an immediate use list until their statement is actually inserted into the instruction stream via a `bsi_*` routine.

It is also still possible to utilize lazy updating of statements, but this should be used only when absolutely required. Both alias analysis and the dominator optimizations currently do this.

When lazy updating is being used, the immediate use information is out of date and cannot be used reliably. Lazy updating is achieved by simply marking statements modified via calls to `mark_stmt_modified` instead of `update_stmt`. When lazy updating is no longer required, all the modified statements must have `update_stmt` called in order to bring them up to date. This must be done before the optimization is finished, or `verify_ssa` will trigger an abort.

This is done with a simple loop over the instruction stream:

```
block_stmt_iterator bsi;
basic_block bb;
FOR_EACH_BB (bb)
  {
    for (bsi = bsi_start (bb); !bsi_end_p (bsi); bsi_next (&bsi))
      update_stmt_if_modified (bsi_stmt (bsi));
  }
```

10.5 Static Single Assignment

Most of the tree optimizers rely on the data flow information provided by the Static Single Assignment (SSA) form. We implement the SSA form as described in *R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, 13(4):451-490, October 1991.*

The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable. Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment.

We represent variable versions using `SSA_NAME` nodes. The renaming process in '`tree-ssa.c`' wraps every real and virtual operand with an `SSA_NAME` node which contains the version number and the statement that created the `SSA_NAME`. Only definitions and virtual definitions may create new `SSA_NAME` nodes.

Sometimes, flow of control makes it impossible to determine what is the most recent version of a variable. In these cases, the compiler inserts an artificial definition for that

variable called *PHI function* or *PHI node*. This new definition merges all the incoming versions of the variable to create a new name for it. For instance,

```
if (...)
  a_1 = 5;
else if (...)
  a_2 = 2;
else
  a_3 = 13;

# a_4 = PHI <a_1, a_2, a_3>
return a_4;
```

Since it is not possible to determine which of the three branches will be taken at runtime, we don't know which of `a_1`, `a_2` or `a_3` to use at the return statement. So, the SSA renamer creates a new version `a_4` which is assigned the result of "merging" `a_1`, `a_2` and `a_3`. Hence, PHI nodes mean "one of these operands. I don't know which".

The following macros can be used to examine PHI nodes

`PHI_RESULT (`*phi*`)` [Macro]
Returns the `SSA_NAME` created by PHI node *phi* (i.e., *phi*'s LHS).

`PHI_NUM_ARGS (`*phi*`)` [Macro]
Returns the number of arguments in *phi*. This number is exactly the number of incoming edges to the basic block holding *phi*.

`PHI_ARG_ELT (`*phi*, *i*`)` [Macro]
Returns a tuple representing the *i*th argument of *phi*. Each element of this tuple contains an `SSA_NAME` *var* and the incoming edge through which *var* flows.

`PHI_ARG_EDGE (`*phi*, *i*`)` [Macro]
Returns the incoming edge for the *i*th argument of *phi*.

`PHI_ARG_DEF (`*phi*, *i*`)` [Macro]
Returns the `SSA_NAME` for the *i*th argument of *phi*.

10.5.1 Preserving the SSA form

Some optimization passes make changes to the function that invalidate the SSA property. This can happen when a pass has added new symbols or changed the program so that variables that were previously aliased aren't anymore. Whenever something like this happens, the affected symbols must be renamed into SSA form again. Transformations that emit new code or replicate existing statements will also need to update the SSA form.

Since GCC implements two different SSA forms for register and virtual variables, keeping the SSA form up to date depends on whether you are updating register or virtual names. In both cases, the general idea behind incremental SSA updates is similar: when new SSA names are created, they typically are meant to replace other existing names in the program.

For instance, given the following code:

```
1 L0:
2 x_1 = PHI (0, x_5)
3 if (x_1 < 10)
4   if (x_1 > 7)
5     y_2 = 0
```

```
6   else
7     y_3 = x_1 + x_7
8   endif
9   x_5 = x_1 + 1
10   goto L0;
11 endif
```

Suppose that we insert new names `x_10` and `x_11` (lines `4` and `8`).

```
1 L0:
2 x_1 = PHI (0, x_5)
3 if (x_1 < 10)
4   x_10 = ...
5   if (x_1 > 7)
6     y_2 = 0
7   else
8     x_11 = ...
9     y_3 = x_1 + x_7
10   endif
11   x_5 = x_1 + 1
12   goto L0;
13 endif
```

We want to replace all the uses of `x_1` with the new definitions of `x_10` and `x_11`. Note that the only uses that should be replaced are those at lines `5`, `9` and `11`. Also, the use of `x_7` at line `9` should *not* be replaced (this is why we cannot just mark symbol `x` for renaming).

Additionally, we may need to insert a PHI node at line `11` because that is a merge point for `x_10` and `x_11`. So the use of `x_1` at line `11` will be replaced with the new PHI node. The insertion of PHI nodes is optional. They are not strictly necessary to preserve the SSA form, and depending on what the caller inserted, they may not even be useful for the optimizers.

Updating the SSA form is a two step process. First, the pass has to identify which names need to be updated and/or which symbols need to be renamed into SSA form for the first time. When new names are introduced to replace existing names in the program, the mapping between the old and the new names are registered by calling `register_new_name_mapping` (note that if your pass creates new code by duplicating basic blocks, the call to `tree_duplicate_bb` will set up the necessary mappings automatically). On the other hand, if your pass exposes a new symbol that should be put in SSA form for the first time, the new symbol should be registered with `mark_sym_for_renaming`.

After the replacement mappings have been registered and new symbols marked for renaming, a call to `update_ssa` makes the registered changes. This can be done with an explicit call or by creating `TODO` flags in the `tree_opt_pass` structure for your pass. There are several `TODO` flags that control the behavior of `update_ssa`:

- `TODO_update_ssa`. Update the SSA form inserting PHI nodes for newly exposed symbols and virtual names marked for updating. When updating real names, only insert PHI nodes for a real name `O_j` in blocks reached by all the new and old definitions for `O_j`. If the iterated dominance frontier for `O_j` is not pruned, we may end up inserting PHI nodes in blocks that have one or more edges with no incoming definition for `O_j`. This would lead to uninitialized warnings for `O_j`'s symbol.
- `TODO_update_ssa_no_phi`. Update the SSA form without inserting any new PHI nodes at all. This is used by passes that have either inserted all the PHI nodes themselves or passes that need only to patch use-def and def-def chains for virtuals (e.g., DCE).

- `TODO_update_ssa_full_phi`. Insert PHI nodes everywhere they are needed. No pruning of the IDF is done. This is used by passes that need the PHI nodes for `O_j` even if it means that some arguments will come from the default definition of `O_j`'s symbol (e.g., `pass_linear_transform`).

  WARNING: If you need to use this flag, chances are that your pass may be doing something wrong. Inserting PHI nodes for an old name where not all edges carry a new replacement may lead to silent codegen errors or spurious uninitialized warnings.
- `TODO_update_ssa_only_virtuals`. Passes that update the SSA form on their own may want to delegate the updating of virtual names to the generic updater. Since FUD chains are easier to maintain, this simplifies the work they need to do. NOTE: If this flag is used, any OLD->NEW mappings for real names are explicitly destroyed and only the symbols marked for renaming are processed.

10.5.2 Examining `SSA_NAME` nodes

The following macros can be used to examine `SSA_NAME` nodes

`SSA_NAME_DEF_STMT (var)` [Macro]
Returns the statement *s* that creates the `SSA_NAME` *var*. If *s* is an empty statement (i.e., `IS_EMPTY_STMT (s)` returns `true`), it means that the first reference to this variable is a USE or a VUSE.

`SSA_NAME_VERSION (var)` [Macro]
Returns the version number of the `SSA_NAME` object *var*.

10.5.3 Walking use-def chains

`void walk_use_def_chains (var, fn, data)` [Tree SSA function]
Walks use-def chains starting at the `SSA_NAME` node *var*. Calls function *fn* at each reaching definition found. Function *FN* takes three arguments: *var*, its defining statement (*def_stmt*) and a generic pointer to whatever state information that *fn* may want to maintain (*data*). Function *fn* is able to stop the walk by returning `true`, otherwise in order to continue the walk, *fn* should return `false`.

Note, that if *def_stmt* is a `PHI` node, the semantics are slightly different. For each argument *arg* of the PHI node, this function will:

1. Walk the use-def chains for *arg*.
2. Call `FN (arg, phi, data)`.

Note how the first argument to *fn* is no longer the original variable *var*, but the PHI argument currently being examined. If *fn* wants to get at *var*, it should call `PHI_RESULT` (*phi*).

10.5.4 Walking the dominator tree

`void walk_dominator_tree (walk_data, bb)` [Tree SSA function]
This function walks the dominator tree for the current CFG calling a set of callback functions defined in *struct dom_walk_data* in '`domwalk.h`'. The call back functions you need to define give you hooks to execute custom code at various points during traversal:

1. Once to initialize any local data needed while processing *bb* and its children. This local data is pushed into an internal stack which is automatically pushed and popped as the walker traverses the dominator tree.
2. Once before traversing all the statements in the *bb*.
3. Once for every statement inside *bb*.
4. Once after traversing all the statements and before recursing into *bb*'s dominator children.
5. It then recurses into all the dominator children of *bb*.
6. After recursing into all the dominator children of *bb* it can, optionally, traverse every statement in *bb* again (i.e., repeating steps 2 and 3).
7. Once after walking the statements in *bb* and *bb*'s dominator children. At this stage, the block local data stack is popped.

10.6 Alias analysis

Alias analysis proceeds in 4 main phases:

1. Structural alias analysis.

   This phase walks the types for structure variables, and determines which of the fields can overlap using offset and size of each field. For each field, a "subvariable" called a "Structure field tag" (SFT) is created, which represents that field as a separate variable. All accesses that could possibly overlap with a given field will have virtual operands for the SFT of that field.

   ```
   struct foo
   {
     int a;
     int b;
   }
   struct foo temp;
   int bar (void)
   {
     int tmp1, tmp2, tmp3;
     SFT.0_2 = V_MUST_DEF <SFT.0_1>
     temp.a = 5;
     SFT.1_4 = V_MUST_DEF <SFT.1_3>
     temp.b = 6;

     VUSE <SFT.1_4>
     tmp1_5 = temp.b;
     VUSE <SFT.0_2>
     tmp2_6 = temp.a;

     tmp3_7 = tmp1_5 + tmp2_6;
     return tmp3_7;
   }
   ```

   If you copy the type tag for a variable for some reason, you probably also want to copy the subvariables for that variable.

2. Points-to and escape analysis.

   This phase walks the use-def chains in the SSA web looking for three things:

   - Assignments of the form `P_i = &VAR`

- Assignments of the form P_i = malloc()
- Pointers and ADDR_EXPR that escape the current function.

The concept of 'escaping' is the same one used in the Java world. When a pointer or an ADDR_EXPR escapes, it means that it has been exposed outside of the current function. So, assignment to global variables, function arguments and returning a pointer are all escape sites.

This is where we are currently limited. Since not everything is renamed into SSA, we lose track of escape properties when a pointer is stashed inside a field in a structure, for instance. In those cases, we are assuming that the pointer does escape.

We use escape analysis to determine whether a variable is call-clobbered. Simply put, if an ADDR_EXPR escapes, then the variable is call-clobbered. If a pointer P_i escapes, then all the variables pointed-to by P_i (and its memory tag) also escape.

3. Compute flow-sensitive aliases

   We have two classes of memory tags. Memory tags associated with the pointed-to data type of the pointers in the program. These tags are called "type memory tag" (TMT). The other class are those associated with SSA_NAMEs, called "name memory tag" (NMT). The basic idea is that when adding operands for an INDIRECT_REF *P_i, we will first check whether P_i has a name tag, if it does we use it, because that will have more precise aliasing information. Otherwise, we use the standard type tag.

   In this phase, we go through all the pointers we found in points-to analysis and create alias sets for the name memory tags associated with each pointer P_i. If P_i escapes, we mark call-clobbered the variables it points to and its tag.

4. Compute flow-insensitive aliases

   This pass will compare the alias set of every type memory tag and every addressable variable found in the program. Given a type memory tag TMT and an addressable variable V. If the alias sets of TMT and V conflict (as computed by may_alias_p), then V is marked as an alias tag and added to the alias set of TMT.

For instance, consider the following function:

```
foo (int i)
{
  int *p, *q, a, b;

  if (i > 10)
    p = &a;
  else
    q = &b;

  *p = 3;
  *q = 5;
  a = b + 2;
  return *p;
}
```

After aliasing analysis has finished, the type memory tag for pointer `p` will have two aliases, namely variables `a` and `b`. Every time pointer `p` is dereferenced, we want to mark the operation as a potential reference to `a` and `b`.

```
foo (int i)
{
```

```
  int *p, a, b;

  if (i_2 > 10)
    p_4 = &a;
  else
    p_6 = &b;
  # p_1 = PHI <p_4(1), p_6(2)>;

  # a_7 = V_MAY_DEF <a_3>;
  # b_8 = V_MAY_DEF <b_5>;
  *p_1 = 3;

  # a_9 = V_MAY_DEF <a_7>
  # VUSE <b_8>
  a_9 = b_8 + 2;

  # VUSE <a_9>;
  # VUSE <b_8>;
  return *p_1;
}
```

In certain cases, the list of may aliases for a pointer may grow too large. This may cause an explosion in the number of virtual operands inserted in the code. Resulting in increased memory consumption and compilation time.

When the number of virtual operands needed to represent aliased loads and stores grows too large (configurable with '`--param max-aliased-vops`'), alias sets are grouped to avoid severe compile-time slow downs and memory consumption. The alias grouping heuristic proceeds as follows:

1. Sort the list of pointers in decreasing number of contributed virtual operands.
2. Take the first pointer from the list and reverse the role of the memory tag and its aliases. Usually, whenever an aliased variable Vi is found to alias with a memory tag T, we add Vi to the may-aliases set for T. Meaning that after alias analysis, we will have:

   ```
   may-aliases(T) = { V1, V2, V3, ..., Vn }
   ```

   This means that every statement that references T, will get `n` virtual operands for each of the Vi tags. But, when alias grouping is enabled, we make T an alias tag and add it to the alias set of all the Vi variables:

   ```
   may-aliases(V1) = { T }
   may-aliases(V2) = { T }
   ...
   may-aliases(Vn) = { T }
   ```

   This has two effects: (a) statements referencing T will only get a single virtual operand, and, (b) all the variables Vi will now appear to alias each other. So, we lose alias precision to improve compile time. But, in theory, a program with such a high level of aliasing should not be very optimizable in the first place.
3. Since variables may be in the alias set of more than one memory tag, the grouping done in step (2) needs to be extended to all the memory tags that have a non-empty intersection with the may-aliases set of tag T. For instance, if we originally had these may-aliases sets:

   ```
   may-aliases(T) = { V1, V2, V3 }
   may-aliases(R) = { V2, V4 }
   ```

In step (2) we would have reverted the aliases for T as:

```
may-aliases(V1) = { T }
may-aliases(V2) = { T }
may-aliases(V3) = { T }
```

But note that now V2 is no longer aliased with R. We could add R to may-aliases(V2), but we are in the process of grouping aliases to reduce virtual operands so what we do is add V4 to the grouping to obtain:

```
may-aliases(V1) = { T }
may-aliases(V2) = { T }
may-aliases(V3) = { T }
may-aliases(V4) = { T }
```

4. If the total number of virtual operands due to aliasing is still above the threshold set by max-alias-vops, go back to (2).

11 RTL Representation

Most of the work of the compiler is done on an intermediate representation called register transfer language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

11.1 RTL Object Types

RTL uses five kinds of objects: expressions, integers, wide integers, strings and vectors. Expressions are the most important ones. An RTL expression ("RTX", for short) is a C structure, but it is usually referred to with a pointer; a type that is given the typedef name `rtx`.

An integer is simply an `int`; their written form uses decimal digits. A wide integer is an integral object whose type is `HOST_WIDE_INT`; their written form uses decimal digits.

A string is a sequence of characters. In core it is represented as a `char *` in usual C fashion, and it is written in C syntax as well. However, strings in RTL may never be null. If you write an empty string in a machine description, it is represented in core as a null pointer rather than as a pointer to a null character. In certain contexts, these null pointers instead of strings are valid. Within RTL code, strings are most commonly found inside `symbol_ref` expressions, but they appear in other contexts in the RTL expressions that make up machine descriptions.

In a machine description, strings are normally written with double quotes, as you would in C. However, strings in machine descriptions may extend over many lines, which is invalid C, and adjacent string constants are not concatenated as they are in C. Any string constant may be surrounded with a single set of parentheses. Sometimes this makes the machine description easier to read.

There is also a special syntax for strings, which can be useful when C code is embedded in a machine description. Wherever a string can appear, it is also valid to write a C-style brace block. The entire brace block, including the outermost pair of braces, is considered to be the string constant. Double quote characters inside the braces are not special. Therefore, if you write string constants in the C code, you need not escape each quote character with a backslash.

A vector contains an arbitrary number of pointers to expressions. The number of elements in the vector is explicitly present in the vector. The written form of a vector consists of square brackets ('`[...]`') surrounding the elements, in sequence and with whitespace separating them. Vectors of length zero are not created; null pointers are used instead.

Expressions are classified by *expression codes* (also called RTX codes). The expression code is a name defined in '`rtl.def`', which is also (in uppercase) a C enumeration constant. The possible expression codes and their meanings are machine-independent. The code of an RTX can be extracted with the macro `GET_CODE (x)` and altered with `PUT_CODE (x, newcode)`.

The expression code determines how many operands the expression contains, and what kinds of objects they are. In RTL, unlike Lisp, you cannot tell by looking at an operand what kind of object it is. Instead, you must know from its context—from the expression code of the containing expression. For example, in an expression of code `subreg`, the first operand is to be regarded as an expression and the second operand as an integer. In an expression of code `plus`, there are two operands, both of which are to be regarded as expressions. In a `symbol_ref` expression, there is one operand, which is to be regarded as a string.

Expressions are written as parentheses containing the name of the expression type, its flags and machine mode if any, and then the operands of the expression (separated by spaces).

Expression code names in the '`md`' file are written in lowercase, but when they appear in C code they are written in uppercase. In this manual, they are shown as follows: `const_int`.

In a few contexts a null pointer is valid where an expression is normally wanted. The written form of this is `(nil)`.

11.2 RTL Classes and Formats

The various expression codes are divided into several *classes*, which are represented by single characters. You can determine the class of an RTX code with the macro `GET_RTX_CLASS (`*`code`*`)`. Currently, '`rtl.def`' defines these classes:

`RTX_OBJ` An RTX code that represents an actual object, such as a register (`REG`) or a memory location (`MEM`, `SYMBOL_REF`). `LO_SUM`) is also included; instead, `SUBREG` and `STRICT_LOW_PART` are not in this class, but in class `x`.

`RTX_CONST_OBJ`
An RTX code that represents a constant object. `HIGH` is also included in this class.

`RTX_COMPARE`
An RTX code for a non-symmetric comparison, such as `GEU` or `LT`.

`RTX_COMM_COMPARE`
An RTX code for a symmetric (commutative) comparison, such as `EQ` or `ORDERED`.

`RTX_UNARY`
An RTX code for a unary arithmetic operation, such as `NEG`, `NOT`, or `ABS`. This category also includes value extension (sign or zero) and conversions between integer and floating point.

`RTX_COMM_ARITH`
An RTX code for a commutative binary operation, such as `PLUS` or `AND`. `NE` and `EQ` are comparisons, so they have class `<`.

`RTX_BIN_ARITH`
An RTX code for a non-commutative binary operation, such as `MINUS`, `DIV`, or `ASHIFTRT`.

`RTX_BITFIELD_OPS`
: An RTX code for a bit-field operation. Currently only `ZERO_EXTRACT` and `SIGN_EXTRACT`. These have three inputs and are lvalues (so they can be used for insertion as well). See Section 11.11 [Bit-Fields], page 147.

`RTX_TERNARY`
: An RTX code for other three input operations. Currently only `IF_THEN_ELSE` and `VEC_MERGE`.

`RTX_INSN`
: An RTX code for an entire instruction: `INSN`, `JUMP_INSN`, and `CALL_INSN`. See Section 11.18 [Insns], page 156.

`RTX_MATCH`
: An RTX code for something that matches in insns, such as `MATCH_DUP`. These only occur in machine descriptions.

`RTX_AUTOINC`
: An RTX code for an auto-increment addressing mode, such as `POST_INC`.

`RTX_EXTRA`
: All other RTX codes. This category includes the remaining codes used only in machine descriptions (`DEFINE_*`, etc.). It also includes all the codes describing side effects (`SET`, `USE`, `CLOBBER`, etc.) and the non-insns that may appear on an insn chain, such as `NOTE`, `BARRIER`, and `CODE_LABEL`. `SUBREG` is also part of this class.

For each expression code, '`rtl.def`' specifies the number of contained objects and their kinds using a sequence of characters called the *format* of the expression code. For example, the format of `subreg` is '`ei`'.

These are the most commonly used format characters:

`e`
: An expression (actually a pointer to an expression).

`i`
: An integer.

`w`
: A wide integer.

`s`
: A string.

`E`
: A vector of expressions.

A few other format characters are used occasionally:

`u`
: '`u`' is equivalent to '`e`' except that it is printed differently in debugging dumps. It is used for pointers to insns.

`n`
: '`n`' is equivalent to '`i`' except that it is printed differently in debugging dumps. It is used for the line number or code number of a `note` insn.

`S`
: '`S`' indicates a string which is optional. In the RTL objects in core, '`S`' is equivalent to '`s`', but when the object is read, from an '`md`' file, the string value of this operand may be omitted. An omitted string is taken to be the null string.

`V` ‘`V`’ indicates a vector which is optional. In the RTL objects in core, ‘`V`’ is equivalent to ‘`E`’, but when the object is read from an ‘`md`’ file, the vector value of this operand may be omitted. An omitted vector is effectively the same as a vector of no elements.

`B` ‘`B`’ indicates a pointer to basic block structure.

`0` ‘`0`’ means a slot whose contents do not fit any normal category. ‘`0`’ slots are not printed at all in dumps, and are often used in special ways by small parts of the compiler.

There are macros to get the number of operands and the format of an expression code:

`GET_RTX_LENGTH (`*code*`)`
Number of operands of an RTX of code *code*.

`GET_RTX_FORMAT (`*code*`)`
The format of an RTX of code *code*, as a C string.

Some classes of RTX codes always have the same format. For example, it is safe to assume that all comparison operations have format `ee`.

`1` All codes of this class have format `e`.

`<`
`c`
`2` All codes of these classes have format `ee`.

`b`
`3` All codes of these classes have format `eee`.

`i` All codes of this class have formats that begin with `iuueiee`. See Section 11.18 [Insns], page 156. Note that not all RTL objects linked onto an insn chain are of class `i`.

`o`
`m`
`x` You can make no assumptions about the format of these codes.

11.3 Access to Operands

Operands of expressions are accessed using the macros `XEXP`, `XINT`, `XWINT` and `XSTR`. Each of these macros takes two arguments: an expression-pointer (RTX) and an operand number (counting from zero). Thus,

```
XEXP (x, 2)
```

accesses operand 2 of expression *x*, as an expression.

```
XINT (x, 2)
```

accesses the same operand as an integer. `XSTR`, used in the same fashion, would access it as a string.

Any operand can be accessed as an integer, as an expression or as a string. You must choose the correct method of access for the kind of value actually stored in the operand. You would do this based on the expression code of the containing expression. That is also how you would know how many operands there are.

For example, if *x* is a `subreg` expression, you know that it has two operands which can be correctly accessed as `XEXP (x, 0)` and `XINT (x, 1)`. If you did `XINT (x, 0)`, you would get the address of the expression operand but cast as an integer; that might occasionally be useful, but it would be cleaner to write `(int) XEXP (x, 0)`. `XEXP (x, 1)` would also compile without error, and would return the second, integer operand cast as an expression pointer, which would probably result in a crash when accessed. Nothing stops you from writing `XEXP (x, 28)` either, but this will access memory past the end of the expression with unpredictable results.

Access to operands which are vectors is more complicated. You can use the macro `XVEC` to get the vector-pointer itself, or the macros `XVECEXP` and `XVECLEN` to access the elements and length of a vector.

`XVEC (exp, idx)`
: Access the vector-pointer which is operand number *idx* in *exp*.

`XVECLEN (exp, idx)`
: Access the length (number of elements) in the vector which is in operand number *idx* in *exp*. This value is an `int`.

`XVECEXP (exp, idx, eltnum)`
: Access element number *eltnum* in the vector which is in operand number *idx* in *exp*. This value is an RTX.

  It is up to you to make sure that *eltnum* is not negative and is less than `XVECLEN (exp, idx)`.

All the macros defined in this section expand into lvalues and therefore can be used to assign the operands, lengths and vector elements as well as to access them.

11.4 Access to Special Operands

Some RTL nodes have special annotations associated with them.

`MEM`

`MEM_ALIAS_SET (x)`
: If 0, *x* is not in any alias set, and may alias anything. Otherwise, *x* can only alias `MEM`s in a conflicting alias set. This value is set in a language-dependent manner in the front-end, and should not be altered in the back-end. In some front-ends, these numbers may correspond in some way to types, or other language-level entities, but they need not, and the back-end makes no such assumptions. These set numbers are tested with `alias_sets_conflict_p`.

`MEM_EXPR (x)`
: If this register is known to hold the value of some user-level declaration, this is that tree node. It may also be a `COMPONENT_REF`, in which case this is some field reference, and `TREE_OPERAND (x, 0)` contains the declaration, or another `COMPONENT_REF`, or null if there is no compile-time object associated with the reference.

`MEM_OFFSET (x)`
: The offset from the start of `MEM_EXPR` as a `CONST_INT` rtx.

`MEM_SIZE (x)`
: The size in bytes of the memory reference as a `CONST_INT` rtx. This is mostly relevant for `BLKmode` references as otherwise the size is implied by the mode.

`MEM_ALIGN (x)`
: The known alignment in bits of the memory reference.

`REG`

`ORIGINAL_REGNO (x)`
: This field holds the number the register "originally" had; for a pseudo register turned into a hard reg this will hold the old pseudo register number.

`REG_EXPR (x)`
: If this register is known to hold the value of some user-level declaration, this is that tree node.

`REG_OFFSET (x)`
: If this register is known to hold the value of some user-level declaration, this is the offset into that logical storage.

`SYMBOL_REF`

`SYMBOL_REF_DECL (x)`
: If the `symbol_ref` x was created for a `VAR_DECL` or a `FUNCTION_DECL`, that tree is recorded here. If this value is null, then x was created by back end code generation routines, and there is no associated front end symbol table entry.

  `SYMBOL_REF_DECL` may also point to a tree of class `'c'`, that is, some sort of constant. In this case, the `symbol_ref` is an entry in the per-file constant pool; again, there is no associated front end symbol table entry.

`SYMBOL_REF_FLAGS (x)`
: In a `symbol_ref`, this is used to communicate various predicates about the symbol. Some of these are common enough to be computed by common code, some are specific to the target. The common bits are:

  `SYMBOL_FLAG_FUNCTION`
  : Set if the symbol refers to a function.

  `SYMBOL_FLAG_LOCAL`
  : Set if the symbol is local to this "module". See `TARGET_BINDS_LOCAL_P`.

  `SYMBOL_FLAG_EXTERNAL`
  : Set if this symbol is not defined in this translation unit. Note that this is not the inverse of `SYMBOL_FLAG_LOCAL`.

`SYMBOL_FLAG_SMALL`
: Set if the symbol is located in the small data section. See `TARGET_IN_SMALL_DATA_P`.

`SYMBOL_REF_TLS_MODEL (x)`
: This is a multi-bit field accessor that returns the `tls_model` to be used for a thread-local storage symbol. It returns zero for non-thread-local symbols.

Bits beginning with `SYMBOL_FLAG_MACH_DEP` are available for the target's use.

11.5 Flags in an RTL Expression

RTL expressions contain several flags (one-bit bit-fields) that are used in certain types of expression. Most often they are accessed with the following macros, which expand into lvalues.

`CONSTANT_POOL_ADDRESS_P (x)`
: Nonzero in a `symbol_ref` if it refers to part of the current function's constant pool. For most targets these addresses are in a `.rodata` section entirely separate from the function, but for some targets the addresses are close to the beginning of the function. In either case GCC assumes these addresses can be addressed directly, perhaps with the help of base registers. Stored in the `unchanging` field and printed as '`/u`'.

`CONST_OR_PURE_CALL_P (x)`
: In a `call_insn`, `note`, or an `expr_list` for notes, indicates that the insn represents a call to a const or pure function. Stored in the `unchanging` field and printed as '`/u`'.

`INSN_ANNULLED_BRANCH_P (x)`
: In a `jump_insn`, `call_insn`, or `insn` indicates that the branch is an annulling one. See the discussion under `sequence` below. Stored in the `unchanging` field and printed as '`/u`'.

`INSN_DELETED_P (x)`
: In an `insn`, `call_insn`, `jump_insn`, `code_label`, `barrier`, or `note`, nonzero if the insn has been deleted. Stored in the `volatil` field and printed as '`/v`'.

`INSN_FROM_TARGET_P (x)`
: In an `insn` or `jump_insn` or `call_insn` in a delay slot of a branch, indicates that the insn is from the target of the branch. If the branch insn has `INSN_ANNULLED_BRANCH_P` set, this insn will only be executed if the branch is taken. For annulled branches with `INSN_FROM_TARGET_P` clear, the insn will be executed only if the branch is not taken. When `INSN_ANNULLED_BRANCH_P` is not set, this insn will always be executed. Stored in the `in_struct` field and printed as '`/s`'.

`LABEL_OUTSIDE_LOOP_P (x)`
: In `label_ref` expressions, nonzero if this is a reference to a label that is outside the innermost loop containing the reference to the label. Stored in the `in_struct` field and printed as '`/s`'.

`LABEL_PRESERVE_P (x)`

In a `code_label` or `note`, indicates that the label is referenced by code or data not visible to the RTL of a given function. Labels referenced by a non-local goto will have this bit set. Stored in the `in_struct` field and printed as '`/s`'.

`LABEL_REF_NONLOCAL_P (x)`

In `label_ref` and `reg_label` expressions, nonzero if this is a reference to a non-local label. Stored in the `volatil` field and printed as '`/v`'.

`MEM_IN_STRUCT_P (x)`

In `mem` expressions, nonzero for reference to an entire structure, union or array, or to a component of one. Zero for references to a scalar variable or through a pointer to a scalar. If both this flag and `MEM_SCALAR_P` are clear, then we don't know whether this `mem` is in a structure or not. Both flags should never be simultaneously set. Stored in the `in_struct` field and printed as '`/s`'.

`MEM_KEEP_ALIAS_SET_P (x)`

In `mem` expressions, 1 if we should keep the alias set for this mem unchanged when we access a component. Set to 1, for example, when we are already in a non-addressable component of an aggregate. Stored in the `jump` field and printed as '`/j`'.

`MEM_SCALAR_P (x)`

In `mem` expressions, nonzero for reference to a scalar known not to be a member of a structure, union, or array. Zero for such references and for indirections through pointers, even pointers pointing to scalar types. If both this flag and `MEM_IN_STRUCT_P` are clear, then we don't know whether this `mem` is in a structure or not. Both flags should never be simultaneously set. Stored in the `frame_related` field and printed as '`/f`'.

`MEM_VOLATILE_P (x)`

In `mem`, `asm_operands`, and `asm_input` expressions, nonzero for volatile memory references. Stored in the `volatil` field and printed as '`/v`'.

`MEM_NOTRAP_P (x)`

In `mem`, nonzero for memory references that will not trap. Stored in the `call` field and printed as '`/c`'.

`REG_FUNCTION_VALUE_P (x)`

Nonzero in a `reg` if it is the place in which this function's value is going to be returned. (This happens only in a hard register.) Stored in the `integrated` field and printed as '`/i`'.

`REG_POINTER (x)`

Nonzero in a `reg` if the register holds a pointer. Stored in the `frame_related` field and printed as '`/f`'.

`REG_USERVAR_P (x)`

In a `reg`, nonzero if it corresponds to a variable present in the user's source code. Zero for temporaries generated internally by the compiler. Stored in the `volatil` field and printed as '`/v`'.

The same hard register may be used also for collecting the values of functions called by this one, but `REG_FUNCTION_VALUE_P` is zero in this kind of use.

`RTX_FRAME_RELATED_P (x)`

Nonzero in an `insn`, `call_insn`, `jump_insn`, `barrier`, or `set` which is part of a function prologue and sets the stack pointer, sets the frame pointer, or saves a register. This flag should also be set on an instruction that sets up a temporary register to use in place of the frame pointer. Stored in the `frame_related` field and printed as '`/f`'.

In particular, on RISC targets where there are limits on the sizes of immediate constants, it is sometimes impossible to reach the register save area directly from the stack pointer. In that case, a temporary register is used that is near enough to the register save area, and the Canonical Frame Address, i.e., DWARF2's logical frame pointer, register must (temporarily) be changed to be this temporary register. So, the instruction that sets this temporary register must be marked as `RTX_FRAME_RELATED_P`.

If the marked instruction is overly complex (defined in terms of what `dwarf2out_frame_debug_expr` can handle), you will also have to create a `REG_FRAME_RELATED_EXPR` note and attach it to the instruction. This note should contain a simple expression of the computation performed by this instruction, i.e., one that `dwarf2out_frame_debug_expr` can handle.

This flag is required for exception handling support on targets with RTL prologues.

`code_label`, `insn_list`, `const`, or `note` if it resulted from an in-line function call. Stored in the `integrated` field and printed as '`/i`'.

`MEM_READONLY_P (x)`

Nonzero in a `mem`, if the memory is statically allocated and read-only.

Read-only in this context means never modified during the lifetime of the program, not necessarily in ROM or in write-disabled pages. A common example of the later is a shared library's global offset table. This table is initialized by the runtime loader, so the memory is technically writable, but after control is transfered from the runtime loader to the application, this memory will never be subsequently modified.

Stored in the `unchanging` field and printed as '`/u`'.

`SCHED_GROUP_P (x)`

During instruction scheduling, in an `insn`, `call_insn` or `jump_insn`, indicates that the previous insn must be scheduled together with this insn. This is used to ensure that certain groups of instructions will not be split up by the instruction scheduling pass, for example, `use` insns before a `call_insn` may not be separated from the `call_insn`. Stored in the `in_struct` field and printed as '`/s`'.

`SET_IS_RETURN_P (x)`

For a `set`, nonzero if it is for a return. Stored in the `jump` field and printed as '`/j`'.

`SIBLING_CALL_P (x)`

For a `call_insn`, nonzero if the insn is a sibling call. Stored in the `jump` field and printed as '`/j`'.

`STRING_POOL_ADDRESS_P (x)`
: For a `symbol_ref` expression, nonzero if it addresses this function's string constant pool. Stored in the `frame_related` field and printed as '`/f`'.

`SUBREG_PROMOTED_UNSIGNED_P (x)`
: Returns a value greater then zero for a `subreg` that has `SUBREG_PROMOTED_VAR_P` nonzero if the object being referenced is kept zero-extended, zero if it is kept sign-extended, and less then zero if it is extended some other way via the `ptr_extend` instruction. Stored in the `unchanging` field and `volatil` field, printed as '`/u`' and '`/v`'. This macro may only be used to get the value it may not be used to change the value. Use `SUBREG_PROMOTED_UNSIGNED_SET` to change the value.

`SUBREG_PROMOTED_UNSIGNED_SET (x)`
: Set the `unchanging` and `volatil` fields in a `subreg` to reflect zero, sign, or other extension. If `volatil` is zero, then `unchanging` as nonzero means zero extension and as zero means sign extension. If `volatil` is nonzero then some other type of extension was done via the `ptr_extend` instruction.

`SUBREG_PROMOTED_VAR_P (x)`
: Nonzero in a `subreg` if it was made when accessing an object that was promoted to a wider mode in accord with the `PROMOTED_MODE` machine description macro (see Section 14.5 [Storage Layout], page 278). In this case, the mode of the `subreg` is the declared mode of the object and the mode of `SUBREG_REG` is the mode of the register that holds the object. Promoted variables are always either sign- or zero-extended to the wider mode on every assignment. Stored in the `in_struct` field and printed as '`/s`'.

`SYMBOL_REF_USED (x)`
: In a `symbol_ref`, indicates that x has been used. This is normally only used to ensure that x is only declared external once. Stored in the `used` field.

`SYMBOL_REF_WEAK (x)`
: In a `symbol_ref`, indicates that x has been declared weak. Stored in the `integrated` field and printed as '`/i`'.

`SYMBOL_REF_FLAG (x)`
: In a `symbol_ref`, this is used as a flag for machine-specific purposes. Stored in the `volatil` field and printed as '`/v`'.

  Most uses of `SYMBOL_REF_FLAG` are historic and may be subsumed by `SYMBOL_REF_FLAGS`. Certainly use of `SYMBOL_REF_FLAGS` is mandatory if the target requires more than one bit of storage.

These are the fields to which the above macros refer:

`call`
: In a `mem`, 1 means that the memory reference will not trap.

  In an RTL dump, this flag is represented as '`/c`'.

`frame_related`
: In an `insn` or `set` expression, 1 means that it is part of a function prologue and sets the stack pointer, sets the frame pointer, saves a register, or sets up a temporary register to use in place of the frame pointer.

In `reg` expressions, 1 means that the register holds a pointer.

In `symbol_ref` expressions, 1 means that the reference addresses this function's string constant pool.

In `mem` expressions, 1 means that the reference is to a scalar.

In an RTL dump, this flag is represented as '`/f`'.

`in_struct`

In `mem` expressions, it is 1 if the memory datum referred to is all or part of a structure or array; 0 if it is (or might be) a scalar variable. A reference through a C pointer has 0 because the pointer might point to a scalar variable. This information allows the compiler to determine something about possible cases of aliasing.

In `reg` expressions, it is 1 if the register has its entire life contained within the test expression of some loop.

In `subreg` expressions, 1 means that the `subreg` is accessing an object that has had its mode promoted from a wider mode.

In `label_ref` expressions, 1 means that the referenced label is outside the innermost loop containing the insn in which the `label_ref` was found.

In `code_label` expressions, it is 1 if the label may never be deleted. This is used for labels which are the target of non-local gotos. Such a label that would have been deleted is replaced with a `note` of type `NOTE_INSN_DELETED_LABEL`.

In an `insn` during dead-code elimination, 1 means that the insn is dead code.

In an `insn` or `jump_insn` during reorg for an insn in the delay slot of a branch, 1 means that this insn is from the target of the branch.

In an `insn` during instruction scheduling, 1 means that this insn must be scheduled as part of a group together with the previous insn.

In an RTL dump, this flag is represented as '`/s`'.

`integrated`

In an `insn`, `insn_list`, or `const`, 1 means the RTL was produced by procedure integration.

In `reg` expressions, 1 means the register contains the value to be returned by the current function. On machines that pass parameters in registers, the same register number may be used for parameters as well, but this flag is not set on such uses.

In `symbol_ref` expressions, 1 means the referenced symbol is weak.

In an RTL dump, this flag is represented as '`/i`'.

`jump` In a `mem` expression, 1 means we should keep the alias set for this mem unchanged when we access a component.

In a `set`, 1 means it is for a return.

In a `call_insn`, 1 means it is a sibling call.

In an RTL dump, this flag is represented as '`/j`'.

`unchanging`

In `reg` and `mem` expressions, 1 means that the value of the expression never changes.

In `subreg` expressions, it is 1 if the `subreg` references an unsigned object whose mode has been promoted to a wider mode.

In an `insn` or `jump_insn` in the delay slot of a branch instruction, 1 means an annulling branch should be used.

In a `symbol_ref` expression, 1 means that this symbol addresses something in the per-function constant pool.

In a `call_insn`, `note`, or an `expr_list` of notes, 1 means that this instruction is a call to a const or pure function.

In an RTL dump, this flag is represented as '`/u`'.

`used` This flag is used directly (without an access macro) at the end of RTL generation for a function, to count the number of times an expression appears in insns. Expressions that appear more than once are copied, according to the rules for shared structure (see Section 11.20 [Sharing], page 165).

For a `reg`, it is used directly (without an access macro) by the leaf register renumbering code to ensure that each register is only renumbered once.

In a `symbol_ref`, it indicates that an external declaration for the symbol has already been written.

`volatil` In a `mem`, `asm_operands`, or `asm_input` expression, it is 1 if the memory reference is volatile. Volatile memory references may not be deleted, reordered or combined.

In a `symbol_ref` expression, it is used for machine-specific purposes.

In a `reg` expression, it is 1 if the value is a user-level variable. 0 indicates an internal compiler temporary.

In an `insn`, 1 means the insn has been deleted.

In `label_ref` and `reg_label` expressions, 1 means a reference to a non-local label.

In an RTL dump, this flag is represented as '`/v`'.

11.6 Machine Modes

A machine mode describes a size of data object and the representation used for it. In the C code, machine modes are represented by an enumeration type, `enum machine_mode`, defined in '`machmode.def`'. Each RTL expression has room for a machine mode and so do certain kinds of tree expressions (declarations and types, to be precise).

In debugging dumps and machine descriptions, the machine mode of an RTL expression is written after the expression code with a colon to separate them. The letters '`mode`' which appear at the end of each machine mode name are omitted. For example, `(reg:SI 38)` is a `reg` expression with machine mode `SImode`. If the mode is `VOIDmode`, it is not written at all.

Here is a table of machine modes. The term "byte" below refers to an object of `BITS_PER_UNIT` bits (see Section 14.5 [Storage Layout], page 278).

`BImode` "Bit" mode represents a single bit, for predicate registers.

`QImode` "Quarter-Integer" mode represents a single byte treated as an integer.

`HImode` "Half-Integer" mode represents a two-byte integer.

`PSImode` "Partial Single Integer" mode represents an integer which occupies four bytes but which doesn't really use all four. On some machines, this is the right mode to use for pointers.

`SImode` "Single Integer" mode represents a four-byte integer.

`PDImode` "Partial Double Integer" mode represents an integer which occupies eight bytes but which doesn't really use all eight. On some machines, this is the right mode to use for certain pointers.

`DImode` "Double Integer" mode represents an eight-byte integer.

`TImode` "Tetra Integer" (?) mode represents a sixteen-byte integer.

`OImode` "Octa Integer" (?) mode represents a thirty-two-byte integer.

`QFmode` "Quarter-Floating" mode represents a quarter-precision (single byte) floating point number.

`HFmode` "Half-Floating" mode represents a half-precision (two byte) floating point number.

`TQFmode` "Three-Quarter-Floating" (?) mode represents a three-quarter-precision (three byte) floating point number.

`SFmode` "Single Floating" mode represents a four byte floating point number. In the common case, of a processor with IEEE arithmetic and 8-bit bytes, this is a single-precision IEEE floating point number; it can also be used for double-precision (on processors with 16-bit bytes) and single-precision VAX and IBM types.

`DFmode` "Double Floating" mode represents an eight byte floating point number. In the common case, of a processor with IEEE arithmetic and 8-bit bytes, this is a double-precision IEEE floating point number.

`XFmode` "Extended Floating" mode represents an IEEE extended floating point number. This mode only has 80 meaningful bits (ten bytes). Some processors require such numbers to be padded to twelve bytes, others to sixteen; this mode is used for either.

`TFmode` "Tetra Floating" mode represents a sixteen byte floating point number all 128 of whose bits are meaningful. One common use is the IEEE quad-precision format.

`CCmode` "Condition Code" mode represents the value of a condition code, which is a machine-specific set of bits used to represent the result of a comparison operation. Other machine-specific modes may also be used for the condition code. These modes are not used on machines that use `cc0` (see see Section 14.14 [Condition Code], page 338).

`BLKmode` "Block" mode represents values that are aggregates to which none of the other modes apply. In RTL, only memory references can have this mode, and only if they appear in string-move or vector instructions. On machines which have no such instructions, `BLKmode` will not appear in RTL.

`VOIDmode` Void mode means the absence of a mode or an unspecified mode. For example, RTL expressions of code `const_int` have mode `VOIDmode` because they can be taken to have whatever mode the context requires. In debugging dumps of RTL, `VOIDmode` is expressed by the absence of any mode.

`QCmode, HCmode, SCmode, DCmode, XCmode, TCmode`
These modes stand for a complex number represented as a pair of floating point values. The floating point values are in `QFmode`, `HFmode`, `SFmode`, `DFmode`, `XFmode`, and `TFmode`, respectively.

`CQImode, CHImode, CSImode, CDImode, CTImode, COImode`
These modes stand for a complex number represented as a pair of integer values. The integer values are in `QImode`, `HImode`, `SImode`, `DImode`, `TImode`, and `OImode`, respectively.

The machine description defines `Pmode` as a C macro which expands into the machine mode used for addresses. Normally this is the mode whose size is `BITS_PER_WORD`, `SImode` on 32-bit machines.

The only modes which a machine description *must* support are `QImode`, and the modes corresponding to `BITS_PER_WORD`, `FLOAT_TYPE_SIZE` and `DOUBLE_TYPE_SIZE`. The compiler will attempt to use `DImode` for 8-byte structures and unions, but this can be prevented by overriding the definition of `MAX_FIXED_MODE_SIZE`. Alternatively, you can have the compiler use `TImode` for 16-byte structures and unions. Likewise, you can arrange for the C type `short int` to avoid using `HImode`.

Very few explicit references to machine modes remain in the compiler and these few references will soon be removed. Instead, the machine modes are divided into mode classes. These are represented by the enumeration type `enum mode_class` defined in '`machmode.h`'. The possible mode classes are:

`MODE_INT` Integer modes. By default these are `BImode`, `QImode`, `HImode`, `SImode`, `DImode`, `TImode`, and `OImode`.

`MODE_PARTIAL_INT`
The "partial integer" modes, `PQImode`, `PHImode`, `PSImode` and `PDImode`.

`MODE_FLOAT`
Floating point modes. By default these are `QFmode`, `HFmode`, `TQFmode`, `SFmode`, `DFmode`, `XFmode` and `TFmode`.

`MODE_COMPLEX_INT`
Complex integer modes. (These are not currently implemented).

`MODE_COMPLEX_FLOAT`
Complex floating point modes. By default these are `QCmode`, `HCmode`, `SCmode`, `DCmode`, `XCmode`, and `TCmode`.

`MODE_FUNCTION`
: Algol or Pascal function variables including a static chain. (These are not currently implemented).

`MODE_CC`
: Modes representing condition code values. These are `CCmode` plus any `CC_MODE` modes listed in the '*machine*`-modes.def`'. See Section 13.12 [Jump Patterns], page 233, also see Section 14.14 [Condition Code], page 338.

`MODE_RANDOM`
: This is a catchall mode class for modes which don't fit into the above classes. Currently `VOIDmode` and `BLKmode` are in `MODE_RANDOM`.

Here are some C macros that relate to machine modes:

`GET_MODE (`*x*`)`
: Returns the machine mode of the RTX *x*.

`PUT_MODE (`*x*`, `*newmode*`)`
: Alters the machine mode of the RTX *x* to be *newmode*.

`NUM_MACHINE_MODES`
: Stands for the number of machine modes available on the target machine. This is one greater than the largest numeric value of any machine mode.

`GET_MODE_NAME (`*m*`)`
: Returns the name of mode *m* as a string.

`GET_MODE_CLASS (`*m*`)`
: Returns the mode class of mode *m*.

`GET_MODE_WIDER_MODE (`*m*`)`
: Returns the next wider natural mode. For example, the expression `GET_MODE_WIDER_MODE (QImode)` returns `HImode`.

`GET_MODE_SIZE (`*m*`)`
: Returns the size in bytes of a datum of mode *m*.

`GET_MODE_BITSIZE (`*m*`)`
: Returns the size in bits of a datum of mode *m*.

`GET_MODE_MASK (`*m*`)`
: Returns a bitmask containing 1 for all bits in a word that fit within mode *m*. This macro can only be used for modes whose bitsize is less than or equal to `HOST_BITS_PER_INT`.

`GET_MODE_ALIGNMENT (`*m*`)`
: Return the required alignment, in bits, for an object of mode *m*.

`GET_MODE_UNIT_SIZE (`*m*`)`
: Returns the size in bytes of the subunits of a datum of mode *m*. This is the same as `GET_MODE_SIZE` except in the case of complex modes. For them, the unit size is the size of the real or imaginary part.

`GET_MODE_NUNITS (`*m*`)`
: Returns the number of units contained in a mode, i.e., `GET_MODE_SIZE` divided by `GET_MODE_UNIT_SIZE`.

`GET_CLASS_NARROWEST_MODE (c)`
: Returns the narrowest mode in mode class *c*.

The global variables `byte_mode` and `word_mode` contain modes whose classes are `MODE_INT` and whose bitsizes are either `BITS_PER_UNIT` or `BITS_PER_WORD`, respectively. On 32-bit machines, these are `QImode` and `SImode`, respectively.

11.7 Constant Expression Types

The simplest RTL expressions are those that represent constant values.

`(const_int i)`
: This type of expression represents the integer value *i*. *i* is customarily accessed with the macro `INTVAL` as in `INTVAL (exp)`, which is equivalent to `XWINT (exp, 0)`.

  Constants generated for modes with fewer bits than `HOST_WIDE_INT` must be sign extended to full width (e.g., with `gen_int_mode`).

  There is only one expression object for the integer value zero; it is the value of the variable `const0_rtx`. Likewise, the only expression for integer value one is found in `const1_rtx`, the only expression for integer value two is found in `const2_rtx`, and the only expression for integer value negative one is found in `constm1_rtx`. Any attempt to create an expression of code `const_int` and value zero, one, two or negative one will return `const0_rtx`, `const1_rtx`, `const2_rtx` or `constm1_rtx` as appropriate.

  Similarly, there is only one object for the integer whose value is `STORE_FLAG_VALUE`. It is found in `const_true_rtx`. If `STORE_FLAG_VALUE` is one, `const_true_rtx` and `const1_rtx` will point to the same object. If `STORE_FLAG_VALUE` is -1, `const_true_rtx` and `constm1_rtx` will point to the same object.

`(const_double:m addr i0 i1 ...)`
: Represents either a floating-point constant of mode *m* or an integer constant too large to fit into `HOST_BITS_PER_WIDE_INT` bits but small enough to fit within twice that number of bits (GCC does not provide a mechanism to represent even larger constants). In the latter case, *m* will be `VOIDmode`.

`(const_vector:m [x0 x1 ...])`
: Represents a vector constant. The square brackets stand for the vector containing the constant elements. *x0*, *x1* and so on are the `const_int` or `const_double` elements.

  The number of units in a `const_vector` is obtained with the macro `CONST_VECTOR_NUNITS` as in `CONST_VECTOR_NUNITS (v)`.

  Individual elements in a vector constant are accessed with the macro `CONST_VECTOR_ELT` as in `CONST_VECTOR_ELT (v, n)` where *v* is the vector constant and *n* is the element desired.

  addr is used to contain the `mem` expression that corresponds to the location in memory that at which the constant can be found. If it has not been allocated a memory location, but is on the chain of all `const_double` expressions in this compilation (maintained using an undisplayed field), *addr* contains `const0_rtx`. If it is not on the chain, *addr* contains `cc0_rtx`. *addr* is customarily

accessed with the macro `CONST_DOUBLE_MEM` and the chain field via `CONST_DOUBLE_CHAIN`.

If *m* is `VOIDmode`, the bits of the value are stored in *i0* and *i1*. *i0* is customarily accessed with the macro `CONST_DOUBLE_LOW` and *i1* with `CONST_DOUBLE_HIGH`.

If the constant is floating point (regardless of its precision), then the number of integers used to store the value depends on the size of `REAL_VALUE_TYPE` (see Section 14.21 [Floating Point], page 384). The integers represent a floating point number, but not precisely in the target machine's or host machine's floating point format. To convert them to the precise bit pattern used by the target machine, use the macro `REAL_VALUE_TO_TARGET_DOUBLE` and friends (see Section 14.19.2 [Data Output], page 355).

The macro `CONST0_RTX (`*mode*`)` refers to an expression with value 0 in mode *mode*. If mode *mode* is of mode class `MODE_INT`, it returns `const0_rtx`. If mode *mode* is of mode class `MODE_FLOAT`, it returns a `CONST_DOUBLE` expression in mode *mode*. Otherwise, it returns a `CONST_VECTOR` expression in mode *mode*. Similarly, the macro `CONST1_RTX (`*mode*`)` refers to an expression with value 1 in mode *mode* and similarly for `CONST2_RTX`. The `CONST1_RTX` and `CONST2_RTX` macros are undefined for vector modes.

`(const_string` *str*`)`

Represents a constant string with value *str*. Currently this is used only for insn attributes (see Section 13.19 [Insn Attributes], page 248) since constant strings in C are placed in memory.

`(symbol_ref:`*mode symbol*`)`

Represents the value of an assembler label for data. *symbol* is a string that describes the name of the assembler label. If it starts with a '`*`', the label is the rest of *symbol* not including the '`*`'. Otherwise, the label is *symbol*, usually prefixed with '`_`'.

The `symbol_ref` contains a mode, which is usually `Pmode`. Usually that is the only mode for which a symbol is directly valid.

`(label_ref:`*mode label*`)`

Represents the value of an assembler label for code. It contains one operand, an expression, which must be a `code_label` or a `note` of type `NOTE_INSN_DELETED_LABEL` that appears in the instruction sequence to identify the place where the label should go.

The reason for using a distinct expression type for code label references is so that jump optimization can distinguish them.

The `label_ref` contains a mode, which is usually `Pmode`. Usually that is the only mode for which a label is directly valid.

`(const:`*m exp*`)`

Represents a constant that is the result of an assembly-time arithmetic computation. The operand, *exp*, is an expression that contains only constants (`const_int`, `symbol_ref` and `label_ref` expressions) combined with `plus` and `minus`. However, not all combinations are valid, since the assembler cannot do arbitrary arithmetic on relocatable symbols.

m should be `Pmode`.

`(high:`*m exp*`)`

Represents the high-order bits of *exp*, usually a `symbol_ref`. The number of bits is machine-dependent and is normally the number of bits specified in an instruction that initializes the high order bits of a register. It is used with `lo_sum` to represent the typical two-instruction sequence used in RISC machines to reference a global memory location.

m should be `Pmode`.

11.8 Registers and Memory

Here are the RTL expression types for describing access to machine registers and to main memory.

`(reg:`*m n*`)`

For small values of the integer *n* (those that are less than `FIRST_PSEUDO_REGISTER`), this stands for a reference to machine register number *n*: a *hard register*. For larger values of *n*, it stands for a temporary value or *pseudo register*. The compiler's strategy is to generate code assuming an unlimited number of such pseudo registers, and later convert them into hard registers or into memory references.

m is the machine mode of the reference. It is necessary because machines can generally refer to each register in more than one mode. For example, a register may contain a full word but there may be instructions to refer to it as a half word or as a single byte, as well as instructions to refer to it as a floating point number of various precisions.

Even for a register that the machine can access in only one mode, the mode must always be specified.

The symbol `FIRST_PSEUDO_REGISTER` is defined by the machine description, since the number of hard registers on the machine is an invariant characteristic of the machine. Note, however, that not all of the machine registers must be general registers. All the machine registers that can be used for storage of data are given hard register numbers, even those that can be used only in certain instructions or can hold only certain types of data.

A hard register may be accessed in various modes throughout one function, but each pseudo register is given a natural mode and is accessed only in that mode. When it is necessary to describe an access to a pseudo register using a nonnatural mode, a `subreg` expression is used.

A `reg` expression with a machine mode that specifies more than one word of data may actually stand for several consecutive registers. If in addition the register number specifies a hardware register, then it actually represents several consecutive hardware registers starting with the specified one.

Each pseudo register number used in a function's RTL code is represented by a unique `reg` expression.

Some pseudo register numbers, those within the range of `FIRST_VIRTUAL_REGISTER` to `LAST_VIRTUAL_REGISTER` only appear during the RTL generation

phase and are eliminated before the optimization phases. These represent locations in the stack frame that cannot be determined until RTL generation for the function has been completed. The following virtual register numbers are defined:

`VIRTUAL_INCOMING_ARGS_REGNUM`
: This points to the first word of the incoming arguments passed on the stack. Normally these arguments are placed there by the caller, but the callee may have pushed some arguments that were previously passed in registers.

  When RTL generation is complete, this virtual register is replaced by the sum of the register given by `ARG_POINTER_REGNUM` and the value of `FIRST_PARM_OFFSET`.

`VIRTUAL_STACK_VARS_REGNUM`
: If `FRAME_GROWS_DOWNWARD` is defined to a nonzero value, this points to immediately above the first variable on the stack. Otherwise, it points to the first variable on the stack.

  `VIRTUAL_STACK_VARS_REGNUM` is replaced with the sum of the register given by `FRAME_POINTER_REGNUM` and the value `STARTING_FRAME_OFFSET`.

`VIRTUAL_STACK_DYNAMIC_REGNUM`
: This points to the location of dynamically allocated memory on the stack immediately after the stack pointer has been adjusted by the amount of memory desired.

  This virtual register is replaced by the sum of the register given by `STACK_POINTER_REGNUM` and the value `STACK_DYNAMIC_OFFSET`.

`VIRTUAL_OUTGOING_ARGS_REGNUM`
: This points to the location in the stack at which outgoing arguments should be written when the stack is pre-pushed (arguments pushed using push insns should always use `STACK_POINTER_REGNUM`).

  This virtual register is replaced by the sum of the register given by `STACK_POINTER_REGNUM` and the value `STACK_POINTER_OFFSET`.

`(subreg:`*m reg bytenum*`)`
: `subreg` expressions are used to refer to a register in a machine mode other than its natural one, or to refer to one register of a multi-part `reg` that actually refers to several registers.

  Each pseudo-register has a natural mode. If it is necessary to operate on it in a different mode—for example, to perform a fullword move instruction on a pseudo-register that contains a single byte—the pseudo-register must be enclosed in a `subreg`. In such a case, *bytenum* is zero.

  Usually *m* is at least as narrow as the mode of *reg*, in which case it is restricting consideration to only the bits of *reg* that are in *m*.

  Sometimes *m* is wider than the mode of *reg*. These `subreg` expressions are often called *paradoxical*. They are used in cases where we want to refer to an

object in a wider mode but do not care what value the additional bits have. The reload pass ensures that paradoxical references are only made to hard registers.

The other use of `subreg` is to extract the individual registers of a multi-register value. Machine modes such as `DImode` and `TImode` can indicate values longer than a word, values which usually require two or more consecutive registers. To access one of the registers, use a `subreg` with mode `SImode` and a *bytenum* offset that says which register.

Storing in a non-paradoxical `subreg` has undefined results for bits belonging to the same word as the `subreg`. This laxity makes it easier to generate efficient code for such instructions. To represent an instruction that preserves all the bits outside of those in the `subreg`, use `strict_low_part` around the `subreg`.

The compilation parameter `WORDS_BIG_ENDIAN`, if set to 1, says that byte number zero is part of the most significant word; otherwise, it is part of the least significant word.

The compilation parameter `BYTES_BIG_ENDIAN`, if set to 1, says that byte number zero is the most significant byte within a word; otherwise, it is the least significant byte within a word.

On a few targets, `FLOAT_WORDS_BIG_ENDIAN` disagrees with `WORDS_BIG_ENDIAN`. However, most parts of the compiler treat floating point values as if they had the same endianness as integer values. This works because they handle them solely as a collection of integer values, with no particular numerical value. Only real.c and the runtime libraries care about `FLOAT_WORDS_BIG_ENDIAN`.

Between the combiner pass and the reload pass, it is possible to have a paradoxical `subreg` which contains a `mem` instead of a `reg` as its first operand. After the reload pass, it is also possible to have a non-paradoxical `subreg` which contains a `mem`; this usually occurs when the `mem` is a stack slot which replaced a pseudo register.

Note that it is not valid to access a `DFmode` value in `SFmode` using a `subreg`. On some machines the most significant part of a `DFmode` value does not have the same format as a single-precision floating value.

It is also not valid to access a single word of a multi-word value in a hard register when less registers can hold the value than would be expected from its size. For example, some 32-bit machines have floating-point registers that can hold an entire `DFmode` value. If register 10 were such a register `(subreg:SI (reg:DF 10) 4)` would be invalid because there is no way to convert that reference to a single machine register. The reload pass prevents `subreg` expressions such as these from being formed.

The first operand of a `subreg` expression is customarily accessed with the `SUBREG_REG` macro and the second operand is customarily accessed with the `SUBREG_BYTE` macro.

`(scratch:`*m*`)`

This represents a scratch register that will be required for the execution of a single instruction and not used subsequently. It is converted into a `reg` by either the local register allocator or the reload pass.

`scratch` is usually present inside a `clobber` operation (see Section 11.15 [Side Effects], page 149).

`(cc0)` This refers to the machine's condition code register. It has no operands and may not have a machine mode. There are two ways to use it:

- To stand for a complete set of condition code flags. This is best on most machines, where each comparison sets the entire series of flags.

  With this technique, `(cc0)` may be validly used in only two contexts: as the destination of an assignment (in test and compare instructions) and in comparison operators comparing against zero (`const_int` with value zero; that is to say, `const0_rtx`).

- To stand for a single flag that is the result of a single condition. This is useful on machines that have only a single flag bit, and in which comparison instructions must specify the condition to test.

  With this technique, `(cc0)` may be validly used in only two contexts: as the destination of an assignment (in test and compare instructions) where the source is a comparison operator, and as the first operand of `if_then_else` (in a conditional branch).

There is only one expression object of code `cc0`; it is the value of the variable `cc0_rtx`. Any attempt to create an expression of code `cc0` will return `cc0_rtx`.

Instructions can set the condition code implicitly. On many machines, nearly all instructions set the condition code based on the value that they compute or store. It is not necessary to record these actions explicitly in the RTL because the machine description includes a prescription for recognizing the instructions that do so (by means of the macro `NOTICE_UPDATE_CC`). See Section 14.14 [Condition Code], page 338. Only instructions whose sole purpose is to set the condition code, and instructions that use the condition code, need mention `(cc0)`.

On some machines, the condition code register is given a register number and a `reg` is used instead of `(cc0)`. This is usually the preferable approach if only a small subset of instructions modify the condition code. Other machines store condition codes in general registers; in such cases a pseudo register should be used.

Some machines, such as the SPARC and RS/6000, have two sets of arithmetic instructions, one that sets and one that does not set the condition code. This is best handled by normally generating the instruction that does not set the condition code, and making a pattern that both performs the arithmetic and sets the condition code register (which would not be `(cc0)` in this case). For examples, search for '`addcc`' and '`andcc`' in '`sparc.md`'.

`(pc)` This represents the machine's program counter. It has no operands and may not have a machine mode. `(pc)` may be validly used only in certain specific contexts in jump instructions.

There is only one expression object of code `pc`; it is the value of the variable `pc_rtx`. Any attempt to create an expression of code `pc` will return `pc_rtx`.

All instructions that do not jump alter the program counter implicitly by incrementing it, but there is no need to mention this in the RTL.

`(mem:`*m addr alias*`)`
: This RTX represents a reference to main memory at an address represented by the expression *addr*. *m* specifies how large a unit of memory is accessed. *alias* specifies an alias set for the reference. In general two items are in different alias sets if they cannot reference the same memory address.

  The construct `(mem:BLK (scratch))` is considered to alias all other memories. Thus it may be used as a memory barrier in epilogue stack deallocation patterns.

`(addressof:`*m reg*`)`
: This RTX represents a request for the address of register *reg*. Its mode is always `Pmode`. If there are any `addressof` expressions left in the function after CSE, *reg* is forced into the stack and the `addressof` expression is replaced with a `plus` expression for the address of its stack slot.

11.9 RTL Expressions for Arithmetic

Unless otherwise specified, all the operands of arithmetic expressions must be valid for mode *m*. An operand is valid for mode *m* if it has mode *m*, or if it is a `const_int` or `const_double` and *m* is a mode of class `MODE_INT`.

For commutative binary operations, constants should be placed in the second operand.

`(plus:`*m x y*`)`
`(ss_plus:`*m x y*`)`
`(us_plus:`*m x y*`)`
: These three expressions all represent the sum of the values represented by *x* and *y* carried out in machine mode *m*. They differ in their behavior on overflow of integer modes. `plus` wraps round modulo the width of *m*; `ss_plus` saturates at the maximum signed value representable in *m*; `us_plus` saturates at the maximum unsigned value.

`(lo_sum:`*m x y*`)`
: This expression represents the sum of *x* and the low-order bits of *y*. It is used with `high` (see Section 11.7 [Constants], page 136) to represent the typical two-instruction sequence used in RISC machines to reference a global memory location.

  The number of low order bits is machine-dependent but is normally the number of bits in a `Pmode` item minus the number of bits set by `high`.

  m should be `Pmode`.

`(minus:`*m x y*`)`
`(ss_minus:`*m x y*`)`
`(us_minus:`*m x y*`)`
: These three expressions represent the result of subtracting *y* from *x*, carried out in mode *M*. Behavior on overflow is the same as for the three variants of `plus` (see above).

`(compare:`*m* *x* *y*`)`

Represents the result of subtracting *y* from *x* for purposes of comparison. The result is computed without overflow, as if with infinite precision.

Of course, machines can't really subtract with infinite precision. However, they can pretend to do so when only the sign of the result will be used, which is the case when the result is stored in the condition code. And that is the *only* way this kind of expression may validly be used: as a value to be stored in the condition codes, either `(cc0)` or a register. See Section 11.10 [Comparisons], page 145.

The mode *m* is not related to the modes of *x* and *y*, but instead is the mode of the condition code value. If `(cc0)` is used, it is `VOIDmode`. Otherwise it is some mode in class `MODE_CC`, often `CCmode`. See Section 14.14 [Condition Code], page 338. If *m* is `VOIDmode` or `CCmode`, the operation returns sufficient information (in an unspecified format) so that any comparison operator can be applied to the result of the `COMPARE` operation. For other modes in class `MODE_CC`, the operation only returns a subset of this information.

Normally, *x* and *y* must have the same mode. Otherwise, `compare` is valid only if the mode of *x* is in class `MODE_INT` and *y* is a `const_int` or `const_double` with mode `VOIDmode`. The mode of *x* determines what mode the comparison is to be done in; thus it must not be `VOIDmode`.

If one of the operands is a constant, it should be placed in the second operand and the comparison code adjusted as appropriate.

A `compare` specifying two `VOIDmode` constants is not valid since there is no way to know in what mode the comparison is to be performed; the comparison must either be folded during the compilation or the first operand must be loaded into a register while its mode is still known.

`(neg:`*m* *x*`)`

Represents the negation (subtraction from zero) of the value represented by *x*, carried out in mode *m*.

`(mult:`*m* *x* *y*`)`

Represents the signed product of the values represented by *x* and *y* carried out in machine mode *m*.

Some machines support a multiplication that generates a product wider than the operands. Write the pattern for this as

```
(mult:m (sign_extend:m x) (sign_extend:m y))
```

where *m* is wider than the modes of *x* and *y*, which need not be the same.

For unsigned widening multiplication, use the same idiom, but with `zero_extend` instead of `sign_extend`.

`(div:`*m* *x* *y*`)`

Represents the quotient in signed division of *x* by *y*, carried out in machine mode *m*. If *m* is a floating point mode, it represents the exact quotient; otherwise, the integerized quotient.

Some machines have division instructions in which the operands and quotient widths are not all the same; you should represent such instructions using `truncate` and `sign_extend` as in,

```
(truncate:m1 (div:m2 x (sign_extend:m2 y)))
```

`(udiv:`*m* *x* *y*`)`
: Like `div` but represents unsigned division.

`(mod:`*m* *x* *y*`)`
`(umod:`*m* *x* *y*`)`
: Like `div` and `udiv` but represent the remainder instead of the quotient.

`(smin:`*m* *x* *y*`)`
`(smax:`*m* *x* *y*`)`
: Represents the smaller (for `smin`) or larger (for `smax`) of *x* and *y*, interpreted as signed values in mode *m*. When used with floating point, if both operands are zeros, or if either operand is `NaN`, then it is unspecified which of the two operands is returned as the result.

`(umin:`*m* *x* *y*`)`
`(umax:`*m* *x* *y*`)`
: Like `smin` and `smax`, but the values are interpreted as unsigned integers.

`(not:`*m* *x*`)`
: Represents the bitwise complement of the value represented by *x*, carried out in mode *m*, which must be a fixed-point machine mode.

`(and:`*m* *x* *y*`)`
: Represents the bitwise logical-and of the values represented by *x* and *y*, carried out in machine mode *m*, which must be a fixed-point machine mode.

`(ior:`*m* *x* *y*`)`
: Represents the bitwise inclusive-or of the values represented by *x* and *y*, carried out in machine mode *m*, which must be a fixed-point mode.

`(xor:`*m* *x* *y*`)`
: Represents the bitwise exclusive-or of the values represented by *x* and *y*, carried out in machine mode *m*, which must be a fixed-point mode.

`(ashift:`*m* *x* *c*`)`
: Represents the result of arithmetically shifting *x* left by *c* places. *x* have mode *m*, a fixed-point machine mode. *c* be a fixed-point mode or be a constant with mode `VOIDmode`; which mode is determined by the mode called for in the machine description entry for the left-shift instruction. For example, on the VAX, the mode of *c* is `QImode` regardless of *m*.

`(lshiftrt:`*m* *x* *c*`)`
`(ashiftrt:`*m* *x* *c*`)`
: Like `ashift` but for right shift. Unlike the case for left shift, these two operations are distinct.

`(rotate:`*m* *x* *c*`)`
`(rotatert:`*m* *x* *c*`)`
: Similar but represent left and right rotate. If *c* is a constant, use `rotate`.

`(abs:`*m* *x*`)`
: Represents the absolute value of *x*, computed in mode *m*.

`(sqrt:`*m* *x*`)`
: Represents the square root of *x*, computed in mode *m*. Most often *m* will be a floating point mode.

`(ffs:`*m* *x*`)`
: Represents one plus the index of the least significant 1-bit in *x*, represented as an integer of mode *m*. (The value is zero if *x* is zero.) The mode of *x* need not be *m*; depending on the target machine, various mode combinations may be valid.

`(clz:`*m* *x*`)`
: Represents the number of leading 0-bits in *x*, represented as an integer of mode *m*, starting at the most significant bit position. If *x* is zero, the value is determined by `CLZ_DEFINED_VALUE_AT_ZERO`. Note that this is one of the few expressions that is not invariant under widening. The mode of *x* will usually be an integer mode.

`(ctz:`*m* *x*`)`
: Represents the number of trailing 0-bits in *x*, represented as an integer of mode *m*, starting at the least significant bit position. If *x* is zero, the value is determined by `CTZ_DEFINED_VALUE_AT_ZERO`. Except for this case, `ctz(x)` is equivalent to `ffs(x) - 1`. The mode of *x* will usually be an integer mode.

`(popcount:`*m* *x*`)`
: Represents the number of 1-bits in *x*, represented as an integer of mode *m*. The mode of *x* will usually be an integer mode.

`(parity:`*m* *x*`)`
: Represents the number of 1-bits modulo 2 in *x*, represented as an integer of mode *m*. The mode of *x* will usually be an integer mode.

11.10 Comparison Operations

Comparison operators test a relation on two operands and are considered to represent a machine-dependent nonzero value described by, but not necessarily equal to, `STORE_FLAG_VALUE` (see Section 14.27 [Misc], page 390) if the relation holds, or zero if it does not, for comparison operators whose results have a 'MODE_INT' mode, `FLOAT_STORE_FLAG_VALUE` (see Section 14.27 [Misc], page 390) if the relation holds, or zero if it does not, for comparison operators that return floating-point values, and a vector of either `VECTOR_STORE_FLAG_VALUE` (see Section 14.27 [Misc], page 390) if the relation holds, or of zeros if it does not, for comparison operators that return vector results. The mode of the comparison operation is independent of the mode of the data being compared. If the comparison operation is being tested (e.g., the first operand of an `if_then_else`), the mode must be `VOIDmode`.

There are two ways that comparison operations may be used. The comparison operators may be used to compare the condition codes `(cc0)` against zero, as in `(eq (cc0) (const_int 0))`. Such a construct actually refers to the result of the preceding instruction in which the condition codes were set. The instruction setting the condition code must be adjacent to the instruction using the condition code; only `note` insns may separate them.

Alternatively, a comparison operation may directly compare two data objects. The mode of the comparison is determined by the operands; they must both be valid for a common machine mode. A comparison with both operands constant would be invalid as the machine mode could not be deduced from it, but such a comparison should never exist in RTL due to constant folding.

In the example above, if `(cc0)` were last set to `(compare x y)`, the comparison operation is identical to `(eq x y)`. Usually only one style of comparisons is supported on a particular machine, but the combine pass will try to merge the operations to produce the `eq` shown in case it exists in the context of the particular insn involved.

Inequality comparisons come in two flavors, signed and unsigned. Thus, there are distinct expression codes `gt` and `gtu` for signed and unsigned greater-than. These can produce different results for the same pair of integer values: for example, 1 is signed greater-than −1 but not unsigned greater-than, because −1 when regarded as unsigned is actually `0xffffffff` which is greater than 1.

The signed comparisons are also used for floating point values. Floating point comparisons are distinguished by the machine modes of the operands.

`(eq:`*m* *x* *y*`)`
: `STORE_FLAG_VALUE` if the values represented by *x* and *y* are equal, otherwise 0.

`(ne:`*m* *x* *y*`)`
: `STORE_FLAG_VALUE` if the values represented by *x* and *y* are not equal, otherwise 0.

`(gt:`*m* *x* *y*`)`
: `STORE_FLAG_VALUE` if the *x* is greater than *y*. If they are fixed-point, the comparison is done in a signed sense.

`(gtu:`*m* *x* *y*`)`
: Like `gt` but does unsigned comparison, on fixed-point numbers only.

`(lt:`*m* *x* *y*`)`
`(ltu:`*m* *x* *y*`)`
: Like `gt` and `gtu` but test for "less than".

`(ge:`*m* *x* *y*`)`
`(geu:`*m* *x* *y*`)`
: Like `gt` and `gtu` but test for "greater than or equal".

`(le:`*m* *x* *y*`)`
`(leu:`*m* *x* *y*`)`
: Like `gt` and `gtu` but test for "less than or equal".

`(if_then_else` *cond then else*`)`
: This is not a comparison operation but is listed here because it is always used in conjunction with a comparison operation. To be precise, *cond* is a comparison expression. This expression represents a choice, according to *cond*, between the value represented by *then* and the one represented by *else*.

  On most machines, `if_then_else` expressions are valid only to express conditional jumps.

`(cond [test1 value1 test2 value2 ...] default)`

Similar to `if_then_else`, but more general. Each of *test1*, *test2*, ... is performed in turn. The result of this expression is the *value* corresponding to the first nonzero test, or *default* if none of the tests are nonzero expressions.

This is currently not valid for instruction patterns and is supported only for insn attributes. See Section 13.19 [Insn Attributes], page 248.

11.11 Bit-Fields

Special expression codes exist to represent bit-field instructions.

`(sign_extract:m loc size pos)`

This represents a reference to a sign-extended bit-field contained or starting in *loc* (a memory or register reference). The bit-field is *size* bits wide and starts at bit *pos*. The compilation option `BITS_BIG_ENDIAN` says which end of the memory unit *pos* counts from.

If *loc* is in memory, its mode must be a single-byte integer mode. If *loc* is in a register, the mode to use is specified by the operand of the `insv` or `extv` pattern (see Section 13.9 [Standard Names], page 212) and is usually a full-word integer mode, which is the default if none is specified.

The mode of *pos* is machine-specific and is also specified in the `insv` or `extv` pattern.

The mode *m* is the same as the mode that would be used for *loc* if it were a register.

A `sign_extract` can not appear as an lvalue, or part thereof, in RTL.

`(zero_extract:m loc size pos)`

Like `sign_extract` but refers to an unsigned or zero-extended bit-field. The same sequence of bits are extracted, but they are filled to an entire word with zeros instead of by sign-extension.

Unlike `sign_extract`, this type of expressions can be lvalues in RTL; they may appear on the left side of an assignment, indicating insertion of a value into the specified bit-field.

11.12 Vector Operations

All normal RTL expressions can be used with vector modes; they are interpreted as operating on each part of the vector independently. Additionally, there are a few new expressions to describe specific vector operations.

`(vec_merge:m vec1 vec2 items)`

This describes a merge operation between two vectors. The result is a vector of mode *m*; its elements are selected from either *vec1* or *vec2*. Which elements are selected is described by *items*, which is a bit mask represented by a `const_int`; a zero bit indicates the corresponding element in the result vector is taken from *vec2* while a set bit indicates it is taken from *vec1*.

`(vec_select:m vec1 selection)`

This describes an operation that selects parts of a vector. *vec1* is the source vector, *selection* is a `parallel` that contains a `const_int` for each of the sub-

parts of the result vector, giving the number of the source subpart that should be stored into it.

`(vec_concat:`*m vec1 vec2*`)`
Describes a vector concat operation. The result is a concatenation of the vectors *vec1* and *vec2*; its length is the sum of the lengths of the two inputs.

`(vec_duplicate:`*m vec*`)`
This operation converts a small vector into a larger one by duplicating the input values. The output vector mode must have the same submodes as the input vector mode, and the number of output parts must be an integer multiple of the number of input parts.

11.13 Conversions

All conversions between machine modes must be represented by explicit conversion operations. For example, an expression which is the sum of a byte and a full word cannot be written as `(plus:SI (reg:QI 34) (reg:SI 80))` because the `plus` operation requires two operands of the same machine mode. Therefore, the byte-sized operand is enclosed in a conversion operation, as in

```
(plus:SI (sign_extend:SI (reg:QI 34)) (reg:SI 80))
```

The conversion operation is not a mere placeholder, because there may be more than one way of converting from a given starting mode to the desired final mode. The conversion operation code says how to do it.

For all conversion operations, *x* must not be `VOIDmode` because the mode in which to do the conversion would not be known. The conversion must either be done at compile-time or *x* must be placed into a register.

`(sign_extend:`*m x*`)`
Represents the result of sign-extending the value *x* to machine mode *m*. *m* must be a fixed-point mode and *x* a fixed-point value of a mode narrower than *m*.

`(zero_extend:`*m x*`)`
Represents the result of zero-extending the value *x* to machine mode *m*. *m* must be a fixed-point mode and *x* a fixed-point value of a mode narrower than *m*.

`(float_extend:`*m x*`)`
Represents the result of extending the value *x* to machine mode *m*. *m* must be a floating point mode and *x* a floating point value of a mode narrower than *m*.

`(truncate:`*m x*`)`
Represents the result of truncating the value *x* to machine mode *m*. *m* must be a fixed-point mode and *x* a fixed-point value of a mode wider than *m*.

`(ss_truncate:`*m x*`)`
Represents the result of truncating the value *x* to machine mode *m*, using signed saturation in the case of overflow. Both *m* and the mode of *x* must be fixed-point modes.

`(us_truncate:`*m* *x*`)`
: Represents the result of truncating the value *x* to machine mode *m*, using unsigned saturation in the case of overflow. Both *m* and the mode of *x* must be fixed-point modes.

`(float_truncate:`*m* *x*`)`
: Represents the result of truncating the value *x* to machine mode *m*. *m* must be a floating point mode and *x* a floating point value of a mode wider than *m*.

`(float:`*m* *x*`)`
: Represents the result of converting fixed point value *x*, regarded as signed, to floating point mode *m*.

`(unsigned_float:`*m* *x*`)`
: Represents the result of converting fixed point value *x*, regarded as unsigned, to floating point mode *m*.

`(fix:`*m* *x*`)`
: When *m* is a fixed point mode, represents the result of converting floating point value *x* to mode *m*, regarded as signed. How rounding is done is not specified, so this operation may be used validly in compiling C code only for integer-valued operands.

`(unsigned_fix:`*m* *x*`)`
: Represents the result of converting floating point value *x* to fixed point mode *m*, regarded as unsigned. How rounding is done is not specified.

`(fix:`*m* *x*`)`
: When *m* is a floating point mode, represents the result of converting floating point value *x* (valid for mode *m*) to an integer, still represented in floating point mode *m*, by rounding towards zero.

11.14 Declarations

Declaration expression codes do not represent arithmetic operations but rather state assertions about their operands.

`(strict_low_part (subreg:`*m* `(reg:`*n* *r*`) 0))`
: This expression code is used in only one context: as the destination operand of a `set` expression. In addition, the operand of this expression must be a non-paradoxical `subreg` expression.

  The presence of `strict_low_part` says that the part of the register which is meaningful in mode *n*, but is not part of mode *m*, is not to be altered. Normally, an assignment to such a subreg is allowed to have undefined effects on the rest of the register when *m* is less than a word.

11.15 Side Effect Expressions

The expression codes described so far represent values, not actions. But machine instructions never produce values; they are meaningful only for their side effects on the state of the machine. Special expression codes are used to represent side effects.

The body of an instruction is always one of these side effect codes; the codes described above, which represent values, appear only as the operands of these.

`(set lval x)`

Represents the action of storing the value of *x* into the place represented by *lval*. *lval* must be an expression representing a place that can be stored in: `reg` (or `subreg`, `strict_low_part` or `zero_extract`), `mem`, `pc`, `parallel`, or `cc0`.

If *lval* is a `reg`, `subreg` or `mem`, it has a machine mode; then *x* must be valid for that mode.

If *lval* is a `reg` whose machine mode is less than the full width of the register, then it means that the part of the register specified by the machine mode is given the specified value and the rest of the register receives an undefined value. Likewise, if *lval* is a `subreg` whose machine mode is narrower than the mode of the register, the rest of the register can be changed in an undefined way.

If *lval* is a `strict_low_part` of a subreg, then the part of the register specified by the machine mode of the `subreg` is given the value *x* and the rest of the register is not changed.

If *lval* is a `zero_extract`, then the referenced part of the bit-field (a memory or register reference) specified by the `zero_extract` is given the value *x* and the rest of the bit-field is not changed. Note that `sign_extract` can not appear in *lval*.

If *lval* is `(cc0)`, it has no machine mode, and *x* may be either a `compare` expression or a value that may have any mode. The latter case represents a "test" instruction. The expression `(set (cc0) (reg:m n))` is equivalent to `(set (cc0) (compare (reg:m n) (const_int 0)))`. Use the former expression to save space during the compilation.

If *lval* is a `parallel`, it is used to represent the case of a function returning a structure in multiple registers. Each element of the `parallel` is an `expr_list` whose first operand is a `reg` and whose second operand is a `const_int` representing the offset (in bytes) into the structure at which the data in that register corresponds. The first element may be null to indicate that the structure is also passed partly in memory.

If *lval* is `(pc)`, we have a jump instruction, and the possibilities for *x* are very limited. It may be a `label_ref` expression (unconditional jump). It may be an `if_then_else` (conditional jump), in which case either the second or the third operand must be `(pc)` (for the case which does not jump) and the other of the two must be a `label_ref` (for the case which does jump). *x* may also be a `mem` or `(plus:SI (pc) y)`, where *y* may be a `reg` or a `mem`; these unusual patterns are used to represent jumps through branch tables.

If *lval* is neither `(cc0)` nor `(pc)`, the mode of *lval* must not be `VOIDmode` and the mode of *x* must be valid for the mode of *lval*.

lval is customarily accessed with the `SET_DEST` macro and *x* with the `SET_SRC` macro.

`(return)` As the sole expression in a pattern, represents a return from the current function, on machines where this can be done with one instruction, such as VAXen.

On machines where a multi-instruction "epilogue" must be executed in order to return from the function, returning is done by jumping to a label which precedes the epilogue, and the `return` expression code is never used.

Inside an `if_then_else` expression, represents the value to be placed in `pc` to return to the caller.

Note that an insn pattern of `(return)` is logically equivalent to `(set (pc) (return))`, but the latter form is never used.

`(call` *function nargs*`)`

Represents a function call. *function* is a `mem` expression whose address is the address of the function to be called. *nargs* is an expression which can be used for two purposes: on some machines it represents the number of bytes of stack argument; on others, it represents the number of argument registers.

Each machine has a standard machine mode which *function* must have. The machine description defines macro `FUNCTION_MODE` to expand into the requisite mode name. The purpose of this mode is to specify what kind of addressing is allowed, on machines where the allowed kinds of addressing depend on the machine mode being addressed.

`(clobber` *x*`)`

Represents the storing or possible storing of an unpredictable, undescribed value into *x*, which must be a `reg`, `scratch`, `parallel` or `mem` expression.

One place this is used is in string instructions that store standard values into particular hard registers. It may not be worth the trouble to describe the values that are stored, but it is essential to inform the compiler that the registers will be altered, lest it attempt to keep data in them across the string instruction.

If *x* is `(mem:BLK (const_int 0))` or `(mem:BLK (scratch))`, it means that all memory locations must be presumed clobbered. If *x* is a `parallel`, it has the same meaning as a `parallel` in a `set` expression.

Note that the machine description classifies certain hard registers as "call-clobbered". All function call instructions are assumed by default to clobber these registers, so there is no need to use `clobber` expressions to indicate this fact. Also, each function call is assumed to have the potential to alter any memory location, unless the function is declared `const`.

If the last group of expressions in a `parallel` are each a `clobber` expression whose arguments are `reg` or `match_scratch` (see Section 13.4 [RTL Template], page 181) expressions, the combiner phase can add the appropriate `clobber` expressions to an insn it has constructed when doing so will cause a pattern to be matched.

This feature can be used, for example, on a machine that whose multiply and add instructions don't use an MQ register but which has an add-accumulate instruction that does clobber the MQ register. Similarly, a combined instruction might require a temporary register while the constituent instructions might not.

When a `clobber` expression for a register appears inside a `parallel` with other side effects, the register allocator guarantees that the register is unoccupied both before and after that insn. However, the reload phase may allocate a

register used for one of the inputs unless the '`&`' constraint is specified for the selected alternative (see Section 13.8.4 [Modifiers], page 197). You can clobber either a specific hard register, a pseudo register, or a `scratch` expression; in the latter two cases, GCC will allocate a hard register that is available there for use as a temporary.

For instructions that require a temporary register, you should use `scratch` instead of a pseudo-register because this will allow the combiner phase to add the `clobber` when required. You do this by coding (`clobber` (`match_scratch` ...)). If you do clobber a pseudo register, use one which appears nowhere else—generate a new one each time. Otherwise, you may confuse CSE.

There is one other known use for clobbering a pseudo register in a `parallel`: when one of the input operands of the insn is also clobbered by the insn. In this case, using the same pseudo register in the clobber and elsewhere in the insn produces the expected results.

`(use` *x*`)` Represents the use of the value of *x*. It indicates that the value in *x* at this point in the program is needed, even though it may not be apparent why this is so. Therefore, the compiler will not attempt to delete previous instructions whose only effect is to store a value in *x*. *x* must be a `reg` expression.

In some situations, it may be tempting to add a `use` of a register in a `parallel` to describe a situation where the value of a special register will modify the behavior of the instruction. An hypothetical example might be a pattern for an addition that can either wrap around or use saturating addition depending on the value of a special control register:

```
(parallel [(set (reg:SI 2) (unspec:SI [(reg:SI 3)
                                       (reg:SI 4)] 0))
           (use (reg:SI 1))])
```

This will not work, several of the optimizers only look at expressions locally; it is very likely that if you have multiple insns with identical inputs to the `unspec`, they will be optimized away even if register 1 changes in between.

This means that `use` can *only* be used to describe that the register is live. You should think twice before adding `use` statements, more often you will want to use `unspec` instead. The `use` RTX is most commonly useful to describe that a fixed register is implicitly used in an insn. It is also safe to use in patterns where the compiler knows for other reasons that the result of the whole pattern is variable, such as '`movmem`*m*' or '`call`' patterns.

During the reload phase, an insn that has a `use` as pattern can carry a reg_equal note. These `use` insns will be deleted before the reload phase exits.

During the delayed branch scheduling phase, *x* may be an insn. This indicates that *x* previously was located at this place in the code and its data dependencies need to be taken into account. These `use` insns will be deleted before the delayed branch scheduling phase exits.

`(parallel [`*x0 x1 ...*`])`
Represents several side effects performed in parallel. The square brackets stand for a vector; the operand of `parallel` is a vector of expressions. *x0*, *x1* and so

on are individual side effect expressions—expressions of code `set`, `call`, `return`, `clobber` or `use`.

"In parallel" means that first all the values used in the individual side-effects are computed, and second all the actual side-effects are performed. For example,

```
(parallel [(set (reg:SI 1) (mem:SI (reg:SI 1)))
           (set (mem:SI (reg:SI 1)) (reg:SI 1))])
```

says unambiguously that the values of hard register 1 and the memory location addressed by it are interchanged. In both places where `(reg:SI 1)` appears as a memory address it refers to the value in register 1 *before* the execution of the insn.

It follows that it is *incorrect* to use `parallel` and expect the result of one `set` to be available for the next one. For example, people sometimes attempt to represent a jump-if-zero instruction this way:

```
(parallel [(set (cc0) (reg:SI 34))
           (set (pc) (if_then_else
                       (eq (cc0) (const_int 0))
                       (label_ref ...)
                       (pc)))])
```

But this is incorrect, because it says that the jump condition depends on the condition code value *before* this instruction, not on the new value that is set by this instruction.

Peephole optimization, which takes place together with final assembly code output, can produce insns whose patterns consist of a `parallel` whose elements are the operands needed to output the resulting assembler code—often `reg`, `mem` or constant expressions. This would not be well-formed RTL at any other stage in compilation, but it is ok then because no further optimization remains to be done. However, the definition of the macro `NOTICE_UPDATE_CC`, if any, must deal with such insns if you define any peephole optimizations.

`(cond_exec [`*cond expr*`])`

Represents a conditionally executed expression. The *expr* is executed only if the *cond* is nonzero. The *cond* expression must not have side-effects, but the *expr* may very well have side-effects.

`(sequence [`*insns* `...])`

Represents a sequence of insns. Each of the *insns* that appears in the vector is suitable for appearing in the chain of insns, so it must be an `insn`, `jump_insn`, `call_insn`, `code_label`, `barrier` or `note`.

A `sequence` RTX is never placed in an actual insn during RTL generation. It represents the sequence of insns that result from a `define_expand` *before* those insns are passed to `emit_insn` to insert them in the chain of insns. When actually inserted, the individual sub-insns are separated out and the `sequence` is forgotten.

After delay-slot scheduling is completed, an insn and all the insns that reside in its delay slots are grouped together into a `sequence`. The insn requiring the delay slot is the first insn in the vector; subsequent insns are to be placed in the delay slot.

`INSN_ANNULLED_BRANCH_P` is set on an insn in a delay slot to indicate that a branch insn should be used that will conditionally annul the effect of the insns in the delay slots. In such a case, `INSN_FROM_TARGET_P` indicates that the insn is from the target of the branch and should be executed only if the branch is taken; otherwise the insn should be executed only if the branch is not taken. See Section 13.19.7 [Delay Slots], page 255.

These expression codes appear in place of a side effect, as the body of an insn, though strictly speaking they do not always describe side effects as such:

`(asm_input` *s*`)`
: Represents literal assembler code as described by the string *s*.

`(unspec [`*operands ...*`]` *index*`)`
`(unspec_volatile [`*operands ...*`]` *index*`)`
: Represents a machine-specific operation on *operands*. *index* selects between multiple machine-specific operations. `unspec_volatile` is used for volatile operations and operations that may trap; `unspec` is used for other operations.

  These codes may appear inside a `pattern` of an insn, inside a `parallel`, or inside an expression.

`(addr_vec:`*m* `[`*lr0 lr1 ...*`])`
: Represents a table of jump addresses. The vector elements *lr0*, etc., are `label_ref` expressions. The mode *m* specifies how much space is given to each address; normally *m* would be `Pmode`.

`(addr_diff_vec:`*m base* `[`*lr0 lr1 ...*`]` *min max flags*`)`
: Represents a table of jump addresses expressed as offsets from *base*. The vector elements *lr0*, etc., are `label_ref` expressions and so is *base*. The mode *m* specifies how much space is given to each address-difference. *min* and *max* are set up by branch shortening and hold a label with a minimum and a maximum address, respectively. *flags* indicates the relative position of *base*, *min* and *max* to the containing insn and of *min* and *max* to *base*. See rtl.def for details.

`(prefetch:`*m addr rw locality*`)`
: Represents prefetch of memory at address *addr*. Operand *rw* is 1 if the prefetch is for data to be written, 0 otherwise; targets that do not support write prefetches should treat this as a normal prefetch. Operand *locality* specifies the amount of temporal locality; 0 if there is none or 1, 2, or 3 for increasing levels of temporal locality; targets that do not support locality hints should ignore this.

  This insn is used to minimize cache-miss latency by moving data into a cache before it is accessed. It should use only non-faulting data prefetch instructions.

11.16 Embedded Side-Effects on Addresses

Six special side-effect expression codes appear as memory addresses.

`(pre_dec:`*m x*`)`
: Represents the side effect of decrementing *x* by a standard amount and represents also the value that *x* has after being decremented. *x* must be a `reg` or

`mem`, but most machines allow only a `reg`. *m* must be the machine mode for pointers on the machine in use. The amount *x* is decremented by is the length in bytes of the machine mode of the containing memory reference of which this expression serves as the address. Here is an example of its use:

```
(mem:DF (pre_dec:SI (reg:SI 39)))
```

This says to decrement pseudo register 39 by the length of a `DFmode` value and use the result to address a `DFmode` value.

`(pre_inc:`*m x*`)`
: Similar, but specifies incrementing *x* instead of decrementing it.

`(post_dec:`*m x*`)`
: Represents the same side effect as `pre_dec` but a different value. The value represented here is the value *x* has *before* being decremented.

`(post_inc:`*m x*`)`
: Similar, but specifies incrementing *x* instead of decrementing it.

`(post_modify:`*m x y*`)`
: Represents the side effect of setting *x* to *y* and represents *x* before *x* is modified. *x* must be a `reg` or `mem`, but most machines allow only a `reg`. *m* must be the machine mode for pointers on the machine in use.

  The expression *y* must be one of three forms:

  `(plus:`*m x z*`)`, `(minus:`*m x z*`)`, or `(plus:`*m x i*`)`,

  where *z* is an index register and *i* is a constant.

  Here is an example of its use:

```
(mem:SF (post_modify:SI (reg:SI 42) (plus (reg:SI 42)
                                          (reg:SI 48))))
```

  This says to modify pseudo register 42 by adding the contents of pseudo register 48 to it, after the use of what ever 42 points to.

`(pre_modify:`*m x expr*`)`
: Similar except side effects happen before the use.

These embedded side effect expressions must be used with care. Instruction patterns may not use them. Until the '`flow`' pass of the compiler, they may occur only to represent pushes onto the stack. The '`flow`' pass finds cases where registers are incremented or decremented in one instruction and used as an address shortly before or after; these cases are then transformed to use pre- or post-increment or -decrement.

If a register used as the operand of these expressions is used in another address in an insn, the original value of the register is used. Uses of the register outside of an address are not permitted within the same insn as a use in an embedded side effect expression because such insns behave differently on different machines and hence must be treated as ambiguous and disallowed.

An instruction that can be represented with an embedded side effect could also be represented using `parallel` containing an additional `set` to describe how the address register is altered. This is not done because machines that allow these operations at all typically allow them wherever a memory address is called for. Describing them as additional parallel stores would require doubling the number of entries in the machine description.

11.17 Assembler Instructions as Expressions

The RTX code `asm_operands` represents a value produced by a user-specified assembler instruction. It is used to represent an `asm` statement with arguments. An `asm` statement with a single output operand, like this:

```
asm ("foo %1,%2,%0" : "=a" (outputvar) : "g" (x + y), "di" (*z));
```

is represented using a single `asm_operands` RTX which represents the value that is stored in `outputvar`:

```
(set rtx-for-outputvar
     (asm_operands "foo %1,%2,%0" "a" 0
                   [rtx-for-addition-result rtx-for-*z]
                   [(asm_input:m1 "g")
                    (asm_input:m2 "di")]))
```

Here the operands of the `asm_operands` RTX are the assembler template string, the output-operand's constraint, the index-number of the output operand among the output operands specified, a vector of input operand RTX's, and a vector of input-operand modes and constraints. The mode *m1* is the mode of the sum `x+y`; *m2* is that of `*z`.

When an `asm` statement has multiple output values, its insn has several such `set` RTX's inside of a `parallel`. Each `set` contains a `asm_operands`; all of these share the same assembler template and vectors, but each contains the constraint for the respective output operand. They are also distinguished by the output-operand index number, which is 0, 1, ... for successive output operands.

11.18 Insns

The RTL representation of the code for a function is a doubly-linked chain of objects called *insns*. Insns are expressions with special codes that are used for no other purpose. Some insns are actual instructions; others represent dispatch tables for `switch` statements; others represent labels to jump to or various sorts of declarative information.

In addition to its own specific data, each insn must have a unique id-number that distinguishes it from all other insns in the current function (after delayed branch scheduling, copies of an insn with the same id-number may be present in multiple places in a function, but these copies will always be identical and will only appear inside a `sequence`), and chain pointers to the preceding and following insns. These three fields occupy the same position in every insn, independent of the expression code of the insn. They could be accessed with `XEXP` and `XINT`, but instead three special macros are always used:

`INSN_UID (i)`
: Accesses the unique id of insn *i*.

`PREV_INSN (i)`
: Accesses the chain pointer to the insn preceding *i*. If *i* is the first insn, this is a null pointer.

`NEXT_INSN (i)`
: Accesses the chain pointer to the insn following *i*. If *i* is the last insn, this is a null pointer.

The first insn in the chain is obtained by calling `get_insns`; the last insn is the result of calling `get_last_insn`. Within the chain delimited by these insns, the `NEXT_INSN` and `PREV_INSN` pointers must always correspond: if *insn* is not the first insn,

```
NEXT_INSN (PREV_INSN (insn)) == insn
```

is always true and if *insn* is not the last insn,

```
PREV_INSN (NEXT_INSN (insn)) == insn
```

is always true.

After delay slot scheduling, some of the insns in the chain might be `sequence` expressions, which contain a vector of insns. The value of `NEXT_INSN` in all but the last of these insns is the next insn in the vector; the value of `NEXT_INSN` of the last insn in the vector is the same as the value of `NEXT_INSN` for the `sequence` in which it is contained. Similar rules apply for `PREV_INSN`.

This means that the above invariants are not necessarily true for insns inside `sequence` expressions. Specifically, if *insn* is the first insn in a `sequence`, `NEXT_INSN (PREV_INSN (insn))` is the insn containing the `sequence` expression, as is the value of `PREV_INSN (NEXT_INSN (insn))` if *insn* is the last insn in the `sequence` expression. You can use these expressions to find the containing `sequence` expression.

Every insn has one of the following six expression codes:

`insn`
: The expression code `insn` is used for instructions that do not jump and do not do function calls. `sequence` expressions are always contained in insns with code `insn` even if one of those insns should jump or do function calls.

  Insns with code `insn` have four additional fields beyond the three mandatory ones listed above. These four are described in a table below.

`jump_insn`
: The expression code `jump_insn` is used for instructions that may jump (or, more generally, may contain `label_ref` expressions). If there is an instruction to return from the current function, it is recorded as a `jump_insn`.

  `jump_insn` insns have the same extra fields as `insn` insns, accessed in the same way and in addition contain a field `JUMP_LABEL` which is defined once jump optimization has completed.

  For simple conditional and unconditional jumps, this field contains the `code_label` to which this insn will (possibly conditionally) branch. In a more complex jump, `JUMP_LABEL` records one of the labels that the insn refers to; the only way to find the others is to scan the entire body of the insn. In an `addr_vec`, `JUMP_LABEL` is `NULL_RTX`.

  Return insns count as jumps, but since they do not refer to any labels, their `JUMP_LABEL` is `NULL_RTX`.

`call_insn`
: The expression code `call_insn` is used for instructions that may do function calls. It is important to distinguish these instructions because they imply that certain registers and memory locations may be altered unpredictably.

  `call_insn` insns have the same extra fields as `insn` insns, accessed in the same way and in addition contain a field `CALL_INSN_FUNCTION_USAGE`, which contains a list (chain of `expr_list` expressions) containing `use` and `clobber` expressions that denote hard registers and `MEM`s used or clobbered by the called function.

  A `MEM` generally points to a stack slots in which arguments passed to the libcall by reference (see Section 14.9.7 [Register Arguments], page 316) are

stored. If the argument is caller-copied (see Section 14.9.7 [Register Arguments], page 316), the stack slot will be mentioned in `CLOBBER` and `USE` entries; if it's callee-copied, only a `USE` will appear, and the `MEM` may point to addresses that are not stack slots. These `MEM`s are used only in libcalls, because, unlike regular function calls, `CONST_CALL`s (which libcalls generally are, see Section 11.5 [Flags], page 127) aren't assumed to read and write all memory, so flow would consider the stores dead and remove them. Note that, since a libcall must never return values in memory (see Section 14.9.9 [Aggregate Return], page 322), there will never be a `CLOBBER` for a memory address holding a return value.

`CLOBBER`ed registers in this list augment registers specified in `CALL_USED_REGISTERS` (see Section 14.7.1 [Register Basics], page 290).

`code_label`

A `code_label` insn represents a label that a jump insn can jump to. It contains two special fields of data in addition to the three standard ones. `CODE_LABEL_NUMBER` is used to hold the *label number*, a number that identifies this label uniquely among all the labels in the compilation (not just in the current function). Ultimately, the label is represented in the assembler output as an assembler label, usually of the form '`Ln`' where *n* is the label number.

When a `code_label` appears in an RTL expression, it normally appears within a `label_ref` which represents the address of the label, as a number.

Besides as a `code_label`, a label can also be represented as a `note` of type `NOTE_INSN_DELETED_LABEL`.

The field `LABEL_NUSES` is only defined once the jump optimization phase is completed. It contains the number of times this label is referenced in the current function.

The field `LABEL_KIND` differentiates four different types of labels: `LABEL_NORMAL`, `LABEL_STATIC_ENTRY`, `LABEL_GLOBAL_ENTRY`, and `LABEL_WEAK_ENTRY`. The only labels that do not have type `LABEL_NORMAL` are *alternate entry points* to the current function. These may be static (visible only in the containing translation unit), global (exposed to all translation units), or weak (global, but can be overridden by another symbol with the same name).

Much of the compiler treats all four kinds of label identically. Some of it needs to know whether or not a label is an alternate entry point; for this purpose, the macro `LABEL_ALT_ENTRY_P` is provided. It is equivalent to testing whether '`LABEL_KIND (label) == LABEL_NORMAL`'. The only place that cares about the distinction between static, global, and weak alternate entry points, besides the front-end code that creates them, is the function `output_alternate_entry_point`, in '`final.c`'.

To set the kind of a label, use the `SET_LABEL_KIND` macro.

`barrier` Barriers are placed in the instruction stream when control cannot flow past them. They are placed after unconditional jump instructions to indicate that the jumps are unconditional and after calls to `volatile` functions, which do not return (e.g., `exit`). They contain no information beyond the three standard fields.

`note` `note` insns are used to represent additional debugging and declarative information. They contain two nonstandard fields, an integer which is accessed with the macro `NOTE_LINE_NUMBER` and a string accessed with `NOTE_SOURCE_FILE`.

If `NOTE_LINE_NUMBER` is positive, the note represents the position of a source line and `NOTE_SOURCE_FILE` is the source file name that the line came from. These notes control generation of line number data in the assembler output.

Otherwise, `NOTE_LINE_NUMBER` is not really a line number but a code with one of the following values (and `NOTE_SOURCE_FILE` must contain a null pointer):

`NOTE_INSN_DELETED`
: Such a note is completely ignorable. Some passes of the compiler delete insns by altering them into notes of this kind.

`NOTE_INSN_DELETED_LABEL`
: This marks what used to be a `code_label`, but was not used for other purposes than taking its address and was transformed to mark that no code jumps to it.

`NOTE_INSN_BLOCK_BEG`
`NOTE_INSN_BLOCK_END`
: These types of notes indicate the position of the beginning and end of a level of scoping of variable names. They control the output of debugging information.

`NOTE_INSN_EH_REGION_BEG`
`NOTE_INSN_EH_REGION_END`
: These types of notes indicate the position of the beginning and end of a level of scoping for exception handling. `NOTE_BLOCK_NUMBER` identifies which `CODE_LABEL` or `note` of type `NOTE_INSN_DELETED_LABEL` is associated with the given region.

`NOTE_INSN_LOOP_BEG`
`NOTE_INSN_LOOP_END`
: These types of notes indicate the position of the beginning and end of a `while` or `for` loop. They enable the loop optimizer to find loops quickly.

`NOTE_INSN_LOOP_CONT`
: Appears at the place in a loop that `continue` statements jump to.

`NOTE_INSN_LOOP_VTOP`
: This note indicates the place in a loop where the exit test begins for those loops in which the exit test has been duplicated. This position becomes another virtual start of the loop when considering loop invariants.

`NOTE_INSN_FUNCTION_BEG`
: Appears at the start of the function body, after the function prologue.

`NOTE_INSN_FUNCTION_END`
: Appears near the end of the function body, just before the label that `return` statements jump to (on machine where a single instruction does not suffice for returning). This note may be deleted by jump optimization.

These codes are printed symbolically when they appear in debugging dumps.

The machine mode of an insn is normally `VOIDmode`, but some phases use the mode for various purposes.

The common subexpression elimination pass sets the mode of an insn to `QImode` when it is the first insn in a block that has already been processed.

The second Haifa scheduling pass, for targets that can multiple issue, sets the mode of an insn to `TImode` when it is believed that the instruction begins an issue group. That is, when the instruction cannot issue simultaneously with the previous. This may be relied on by later passes, in particular machine-dependent reorg.

Here is a table of the extra fields of `insn`, `jump_insn` and `call_insn` insns:

`PATTERN (`*i*`)`
: An expression for the side effect performed by this insn. This must be one of the following codes: `set`, `call`, `use`, `clobber`, `return`, `asm_input`, `asm_output`, `addr_vec`, `addr_diff_vec`, `trap_if`, `unspec`, `unspec_volatile`, `parallel`, `cond_exec`, or `sequence`. If it is a `parallel`, each element of the `parallel` must be one these codes, except that `parallel` expressions cannot be nested and `addr_vec` and `addr_diff_vec` are not permitted inside a `parallel` expression.

`INSN_CODE (`*i*`)`
: An integer that says which pattern in the machine description matches this insn, or -1 if the matching has not yet been attempted.

  Such matching is never attempted and this field remains -1 on an insn whose pattern consists of a single `use`, `clobber`, `asm_input`, `addr_vec` or `addr_diff_vec` expression.

  Matching is also never attempted on insns that result from an `asm` statement. These contain at least one `asm_operands` expression. The function `asm_noperands` returns a non-negative value for such insns.

  In the debugging output, this field is printed as a number followed by a symbolic representation that locates the pattern in the '`md`' file as some small positive or negative offset from a named pattern.

`LOG_LINKS (`*i*`)`
: A list (chain of `insn_list` expressions) giving information about dependencies between instructions within a basic block. Neither a jump nor a label may come between the related insns.

`REG_NOTES (`*i*`)`
: A list (chain of `expr_list` and `insn_list` expressions) giving miscellaneous information about the insn. It is often information pertaining to the registers used in this insn.

The `LOG_LINKS` field of an insn is a chain of `insn_list` expressions. Each of these has two operands: the first is an insn, and the second is another `insn_list` expression (the next one in the chain). The last `insn_list` in the chain has a null pointer as second operand. The significant thing about the chain is which insns appear in it (as first operands of `insn_list` expressions). Their order is not significant.

This list is originally set up by the flow analysis pass; it is a null pointer until then. Flow only adds links for those data dependencies which can be used for instruction combination. For each insn, the flow analysis pass adds a link to insns which store into registers values that are used for the first time in this insn. The instruction scheduling pass adds extra links so that every dependence will be represented. Links represent data dependencies, antidependencies and output dependencies; the machine mode of the link distinguishes these three types: antidependencies have mode `REG_DEP_ANTI`, output dependencies have mode `REG_DEP_OUTPUT`, and data dependencies have mode `VOIDmode`.

The `REG_NOTES` field of an insn is a chain similar to the `LOG_LINKS` field but it includes `expr_list` expressions in addition to `insn_list` expressions. There are several kinds of register notes, which are distinguished by the machine mode, which in a register note is really understood as being an `enum reg_note`. The first operand *op* of the note is data whose meaning depends on the kind of note.

The macro `REG_NOTE_KIND (x)` returns the kind of register note. Its counterpart, the macro `PUT_REG_NOTE_KIND (x, newkind)` sets the register note type of *x* to be *newkind*.

Register notes are of three classes: They may say something about an input to an insn, they may say something about an output of an insn, or they may create a linkage between two insns. There are also a set of values that are only used in `LOG_LINKS`.

These register notes annotate inputs to an insn:

`REG_DEAD` The value in *op* dies in this insn; that is to say, altering the value immediately after this insn would not affect the future behavior of the program.

It does not follow that the register *op* has no useful value after this insn since *op* is not necessarily modified by this insn. Rather, no subsequent instruction uses the contents of *op*.

`REG_UNUSED`
The register *op* being set by this insn will not be used in a subsequent insn. This differs from a `REG_DEAD` note, which indicates that the value in an input will not be used subsequently. These two notes are independent; both may be present for the same register.

`REG_INC` The register *op* is incremented (or decremented; at this level there is no distinction) by an embedded side effect inside this insn. This means it appears in a `post_inc`, `pre_inc`, `post_dec` or `pre_dec` expression.

`REG_NONNEG`
The register *op* is known to have a nonnegative value when this insn is reached. This is used so that decrement and branch until zero instructions, such as the m68k dbra, can be matched.

The `REG_NONNEG` note is added to insns only if the machine description has a '`decrement_and_branch_until_zero`' pattern.

`REG_NO_CONFLICT`
: This insn does not cause a conflict between *op* and the item being set by this insn even though it might appear that it does. In other words, if the destination register and *op* could otherwise be assigned the same register, this insn does not prevent that assignment.

  Insns with this note are usually part of a block that begins with a `clobber` insn specifying a multi-word pseudo register (which will be the output of the block), a group of insns that each set one word of the value and have the `REG_NO_CONFLICT` note attached, and a final insn that copies the output to itself with an attached `REG_EQUAL` note giving the expression being computed. This block is encapsulated with `REG_LIBCALL` and `REG_RETVAL` notes on the first and last insns, respectively.

`REG_LABEL`
: This insn uses *op*, a `code_label` or a `note` of type `NOTE_INSN_DELETED_LABEL`, but is not a `jump_insn`, or it is a `jump_insn` that required the label to be held in a register. The presence of this note allows jump optimization to be aware that *op* is, in fact, being used, and flow optimization to build an accurate flow graph.

`REG_CROSSING_JUMP`
: This insn is an branching instruction (either an unconditional jump or an indirect jump) which crosses between hot and cold sections, which could potentially be very far apart in the executable. The presence of this note indicates to other optimizations that this this branching instruction should not be "collapsed" into a simpler branching construct. It is used when the optimization to partition basic blocks into hot and cold sections is turned on.

`REG_SETJMP`
: Appears attached to each `CALL_INSN` to `setjmp` or a related function.

The following notes describe attributes of outputs of an insn:

`REG_EQUIV`
`REG_EQUAL`
: This note is only valid on an insn that sets only one register and indicates that that register will be equal to *op* at run time; the scope of this equivalence differs between the two types of notes. The value which the insn explicitly copies into the register may look different from *op*, but they will be equal at run time. If the output of the single `set` is a `strict_low_part` expression, the note refers to the register that is contained in `SUBREG_REG` of the `subreg` expression.

  For `REG_EQUIV`, the register is equivalent to *op* throughout the entire function, and could validly be replaced in all its occurrences by *op*. ("Validly" here refers to the data flow of the program; simple replacement may make some insns invalid.) For example, when a constant is loaded into a register that is never assigned any other value, this kind of note is used.

  When a parameter is copied into a pseudo-register at entry to a function, a note of this kind records that the register is equivalent to the stack slot where the parameter was passed. Although in this case the register may be set by other

insns, it is still valid to replace the register by the stack slot throughout the function.

A `REG_EQUIV` note is also used on an instruction which copies a register parameter into a pseudo-register at entry to a function, if there is a stack slot where that parameter could be stored. Although other insns may set the pseudo-register, it is valid for the compiler to replace the pseudo-register by stack slot throughout the function, provided the compiler ensures that the stack slot is properly initialized by making the replacement in the initial copy instruction as well. This is used on machines for which the calling convention allocates stack space for register parameters. See `REG_PARM_STACK_SPACE` in Section 14.9.6 [Stack Arguments], page 314.

In the case of `REG_EQUAL`, the register that is set by this insn will be equal to *op* at run time at the end of this insn but not necessarily elsewhere in the function. In this case, *op* is typically an arithmetic expression. For example, when a sequence of insns such as a library call is used to perform an arithmetic operation, this kind of note is attached to the insn that produces or copies the final value.

These two notes are used in different ways by the compiler passes. `REG_EQUAL` is used by passes prior to register allocation (such as common subexpression elimination and loop optimization) to tell them how to think of that value. `REG_EQUIV` notes are used by register allocation to indicate that there is an available substitute expression (either a constant or a `mem` expression for the location of a parameter on the stack) that may be used in place of a register if insufficient registers are available.

Except for stack homes for parameters, which are indicated by a `REG_EQUIV` note and are not useful to the early optimization passes and pseudo registers that are equivalent to a memory location throughout their entire life, which is not detected until later in the compilation, all equivalences are initially indicated by an attached `REG_EQUAL` note. In the early stages of register allocation, a `REG_EQUAL` note is changed into a `REG_EQUIV` note if *op* is a constant and the insn represents the only set of its destination register.

Thus, compiler passes prior to register allocation need only check for `REG_EQUAL` notes and passes subsequent to register allocation need only check for `REG_EQUIV` notes.

These notes describe linkages between insns. They occur in pairs: one insn has one of a pair of notes that points to a second insn, which has the inverse note pointing back to the first insn.

`REG_RETVAL`

This insn copies the value of a multi-insn sequence (for example, a library call), and *op* is the first insn of the sequence (for a library call, the first insn that was generated to set up the arguments for the library call).

Loop optimization uses this note to treat such a sequence as a single operation for code motion purposes and flow analysis uses this note to delete such sequences whose results are dead.

A `REG_EQUAL` note will also usually be attached to this insn to provide the expression being computed by the sequence.

These notes will be deleted after reload, since they are no longer accurate or useful.

`REG_LIBCALL`
: This is the inverse of `REG_RETVAL`: it is placed on the first insn of a multi-insn sequence, and it points to the last one.

  These notes are deleted after reload, since they are no longer useful or accurate.

`REG_CC_SETTER`
`REG_CC_USER`
: On machines that use `cc0`, the insns which set and use `cc0` set and use `cc0` are adjacent. However, when branch delay slot filling is done, this may no longer be true. In this case a `REG_CC_USER` note will be placed on the insn setting `cc0` to point to the insn using `cc0` and a `REG_CC_SETTER` note will be placed on the insn using `cc0` to point to the insn setting `cc0`.

These values are only used in the `LOG_LINKS` field, and indicate the type of dependency that each link represents. Links which indicate a data dependence (a read after write dependence) do not use any code, they simply have mode `VOIDmode`, and are printed without any descriptive text.

`REG_DEP_ANTI`
: This indicates an anti dependence (a write after read dependence).

`REG_DEP_OUTPUT`
: This indicates an output dependence (a write after write dependence).

These notes describe information gathered from gcov profile data. They are stored in the `REG_NOTES` field of an insn as an `expr_list`.

`REG_BR_PROB`
: This is used to specify the ratio of branches to non-branches of a branch insn according to the profile data. The value is stored as a value between 0 and REG_BR_PROB_BASE; larger values indicate a higher probability that the branch will be taken.

`REG_BR_PRED`
: These notes are found in JUMP insns after delayed branch scheduling has taken place. They indicate both the direction and the likelihood of the JUMP. The format is a bitmask of ATTR_FLAG_* values.

`REG_FRAME_RELATED_EXPR`
: This is used on an RTX_FRAME_RELATED_P insn wherein the attached expression is used in place of the actual insn pattern. This is done in cases where the pattern is either complex or misleading.

For convenience, the machine mode in an `insn_list` or `expr_list` is printed using these symbolic codes in debugging dumps.

The only difference between the expression codes `insn_list` and `expr_list` is that the first operand of an `insn_list` is assumed to be an insn and is printed in debugging dumps

as the insn's unique id; the first operand of an `expr_list` is printed in the ordinary way as an expression.

11.19 RTL Representation of Function-Call Insns

Insns that call subroutines have the RTL expression code `call_insn`. These insns must satisfy special rules, and their bodies must use a special RTL expression code, `call`.

A `call` expression has two operands, as follows:

```
(call (mem:fm addr) nbytes)
```

Here *nbytes* is an operand that represents the number of bytes of argument data being passed to the subroutine, *fm* is a machine mode (which must equal as the definition of the `FUNCTION_MODE` macro in the machine description) and *addr* represents the address of the subroutine.

For a subroutine that returns no value, the `call` expression as shown above is the entire body of the insn, except that the insn might also contain `use` or `clobber` expressions.

For a subroutine that returns a value whose mode is not `BLKmode`, the value is returned in a hard register. If this register's number is *r*, then the body of the call insn looks like this:

```
(set (reg:m r)
     (call (mem:fm addr) nbytes))
```

This RTL expression makes it clear (to the optimizer passes) that the appropriate register receives a useful value in this insn.

When a subroutine returns a `BLKmode` value, it is handled by passing to the subroutine the address of a place to store the value. So the call insn itself does not "return" any value, and it has the same RTL form as a call that returns nothing.

On some machines, the call instruction itself clobbers some register, for example to contain the return address. `call_insn` insns on these machines should have a body which is a `parallel` that contains both the `call` expression and `clobber` expressions that indicate which registers are destroyed. Similarly, if the call instruction requires some register other than the stack pointer that is not explicitly mentioned in its RTL, a `use` subexpression should mention that register.

Functions that are called are assumed to modify all registers listed in the configuration macro `CALL_USED_REGISTERS` (see Section 14.7.1 [Register Basics], page 290) and, with the exception of `const` functions and library calls, to modify all of memory.

Insns containing just `use` expressions directly precede the `call_insn` insn to indicate which registers contain inputs to the function. Similarly, if registers other than those in `CALL_USED_REGISTERS` are clobbered by the called function, insns containing a single `clobber` follow immediately after the call to indicate which registers.

11.20 Structure Sharing Assumptions

The compiler assumes that certain kinds of RTL expressions are unique; there do not exist two distinct objects representing the same value. In other cases, it makes an opposite assumption: that no RTL expression object of a certain kind appears in more than one place in the containing structure.

These assumptions refer to a single function; except for the RTL objects that describe global variables and external functions, and a few standard objects such as small integer constants, no RTL objects are common to two functions.

- Each pseudo-register has only a single `reg` object to represent it, and therefore only a single machine mode.
- For any symbolic label, there is only one `symbol_ref` object referring to it.
- All `const_int` expressions with equal values are shared.
- There is only one `pc` expression.
- There is only one `cc0` expression.
- There is only one `const_double` expression with value 0 for each floating point mode. Likewise for values 1 and 2.
- There is only one `const_vector` expression with value 0 for each vector mode, be it an integer or a double constant vector.
- No `label_ref` or `scratch` appears in more than one place in the RTL structure; in other words, it is safe to do a tree-walk of all the insns in the function and assume that each time a `label_ref` or `scratch` is seen it is distinct from all others that are seen.
- Only one `mem` object is normally created for each static variable or stack slot, so these objects are frequently shared in all the places they appear. However, separate but equal objects for these variables are occasionally made.
- When a single `asm` statement has multiple output operands, a distinct `asm_operands` expression is made for each output operand. However, these all share the vector which contains the sequence of input operands. This sharing is used later on to test whether two `asm_operands` expressions come from the same statement, so all optimizations must carefully preserve the sharing if they copy the vector at all.
- No RTL object appears in more than one place in the RTL structure except as described above. Many passes of the compiler rely on this by assuming that they can modify RTL objects in place without unwanted side-effects on other insns.
- During initial RTL generation, shared structure is freely introduced. After all the RTL for a function has been generated, all shared structure is copied by `unshare_all_rtl` in '`emit-rtl.c`', after which the above rules are guaranteed to be followed.
- During the combiner pass, shared structure within an insn can exist temporarily. However, the shared structure is copied before the combiner is finished with the insn. This is done by calling `copy_rtx_if_shared`, which is a subroutine of `unshare_all_rtl`.

11.21 Reading RTL

To read an RTL object from a file, call `read_rtx`. It takes one argument, a stdio stream, and returns a single RTL object. This routine is defined in '`read-rtl.c`'. It is not available in the compiler itself, only the various programs that generate the compiler back end from the machine description.

People frequently have the idea of using RTL stored as text in a file as an interface between a language front end and the bulk of GCC. This idea is not feasible.

GCC was designed to use RTL internally only. Correct RTL for a given program is very dependent on the particular target machine. And the RTL does not contain all the information about the program.

The proper way to interface GCC to a new language front end is with the "tree" data structure, described in the files '`tree.h`' and '`tree.def`'. The documentation for this structure (see Chapter 9 [Trees], page 61) is incomplete.

12 Control Flow Graph

A control flow graph (CFG) is a data structure built on top of the intermediate code representation (the RTL or `tree` instruction stream) abstracting the control flow behavior of a function that is being compiled. The CFG is a directed graph where the vertices represent basic blocks and edges represent possible transfer of control flow from one basic block to another. The data structures used to represent the control flow graph are defined in '`basic-block.h`'.

12.1 Basic Blocks

A basic block is a straight-line sequence of code with only one entry point and only one exit. In GCC, basic blocks are represented using the `basic_block` data type.

Two pointer members of the `basic_block` structure are the pointers `next_bb` and `prev_bb`. These are used to keep doubly linked chain of basic blocks in the same order as the underlying instruction stream. The chain of basic blocks is updated transparently by the provided API for manipulating the CFG. The macro `FOR_EACH_BB` can be used to visit all the basic blocks in lexicographical order. Dominator traversals are also possible using `walk_dominator_tree`. Given two basic blocks A and B, block A dominates block B if A is *always* executed before B.

The `BASIC_BLOCK` array contains all basic blocks in an unspecified order. Each `basic_block` structure has a field that holds a unique integer identifier `index` that is the index of the block in the `BASIC_BLOCK` array. The total number of basic blocks in the function is `n_basic_blocks`. Both the basic block indices and the total number of basic blocks may vary during the compilation process, as passes reorder, create, duplicate, and destroy basic blocks. The index for any block should never be greater than `last_basic_block`.

Special basic blocks represent possible entry and exit points of a function. These blocks are called `ENTRY_BLOCK_PTR` and `EXIT_BLOCK_PTR`. These blocks do not contain any code, and are not elements of the `BASIC_BLOCK` array. Therefore they have been assigned unique, negative index numbers.

Each `basic_block` also contains pointers to the first instruction (the *head*) and the last instruction (the *tail*) or *end* of the instruction stream contained in a basic block. In fact, since the `basic_block` data type is used to represent blocks in both major intermediate representations of GCC (`tree` and RTL), there are pointers to the head and end of a basic block for both representations.

For RTL, these pointers are `rtx head, end`. In the RTL function representation, the head pointer always points either to a `NOTE_INSN_BASIC_BLOCK` or to a `CODE_LABEL`, if present. In the RTL representation of a function, the instruction stream contains not only the "real" instructions, but also *notes*. Any function that moves or duplicates the basic blocks needs to take care of updating of these notes. Many of these notes expect that the instruction stream consists of linear regions, making such updates difficult. The `NOTE_INSN_BASIC_BLOCK` note is the only kind of note that may appear in the instruction stream contained in a basic block. The instruction stream of a basic block always follows a `NOTE_INSN_BASIC_BLOCK`, but zero or more `CODE_LABEL` nodes can precede the block note. A basic block ends by control flow instruction or last instruction before following `CODE_LABEL` or `NOTE_INSN_BASIC_BLOCK`. A `CODE_LABEL` cannot appear in the instruction stream of a basic block.

In addition to notes, the jump table vectors are also represented as "pseudo-instructions" inside the insn stream. These vectors never appear in the basic block and should always be placed just after the table jump instructions referencing them. After removing the table-jump it is often difficult to eliminate the code computing the address and referencing the vector, so cleaning up these vectors is postponed until after liveness analysis. Thus the jump table vectors may appear in the insn stream unreferenced and without any purpose. Before any edge is made *fall-thru*, the existence of such construct in the way needs to be checked by calling `can_fallthru` function.

For the `tree` representation, the head and end of the basic block are being pointed to by the `stmt_list` field, but this special `tree` should never be referenced directly. Instead, at the tree level abstract containers and iterators are used to access statements and expressions in basic blocks. These iterators are called *block statement iterators* (BSIs). Grep for `^bsi` in the various '`tree-*`' files. The following snippet will pretty-print all the statements of the program in the GIMPLE representation.

```
FOR_EACH_BB (bb)
  {
     block_stmt_iterator si;

     for (si = bsi_start (bb); !bsi_end_p (si); bsi_next (&si))
       {
          tree stmt = bsi_stmt (si);
          print_generic_stmt (stderr, stmt, 0);
       }
  }
```

12.2 Edges

Edges represent possible control flow transfers from the end of some basic block A to the head of another basic block B. We say that A is a predecessor of B, and B is a successor of A. Edges are represented in GCC with the `edge` data type. Each `edge` acts as a link between two basic blocks: the `src` member of an edge points to the predecessor basic block of the `dest` basic block. The members `preds` and `succs` of the `basic_block` data type point to type-safe vectors of edges to the predecessors and successors of the block.

When walking the edges in an edge vector, *edge iterators* should be used. Edge iterators are constructed using the `edge_iterator` data structure and several methods are available to operate on them:

`ei_start` This function initializes an `edge_iterator` that points to the first edge in a vector of edges.

`ei_last` This function initializes an `edge_iterator` that points to the last edge in a vector of edges.

`ei_end_p` This predicate is `true` if an `edge_iterator` represents the last edge in an edge vector.

`ei_one_before_end_p`
This predicate is `true` if an `edge_iterator` represents the second last edge in an edge vector.

`ei_next` This function takes a pointer to an `edge_iterator` and makes it point to the next edge in the sequence.

`ei_prev` This function takes a pointer to an `edge_iterator` and makes it point to the previous edge in the sequence.

`ei_edge` This function returns the `edge` currently pointed to by an `edge_iterator`.

`ei_safe_safe`
This function returns the `edge` currently pointed to by an `edge_iterator`, but returns `NULL` if the iterator is pointing at the end of the sequence. This function has been provided for existing code makes the assumption that a `NULL` edge indicates the end of the sequence.

The convenience macro `FOR_EACH_EDGE` can be used to visit all of the edges in a sequence of predecessor or successor edges. It must not be used when an element might be removed during the traversal, otherwise elements will be missed. Here is an example of how to use the macro:

```
edge e;
edge_iterator ei;

FOR_EACH_EDGE (e, ei, bb->succs)
  {
     if (e->flags & EDGE_FALLTHRU)
       break;
  }
```

There are various reasons why control flow may transfer from one block to another. One possibility is that some instruction, for example a `CODE_LABEL`, in a linearized instruction stream just always starts a new basic block. In this case a *fall-thru* edge links the basic block to the first following basic block. But there are several other reasons why edges may be created. The `flags` field of the `edge` data type is used to store information about the type of edge we are dealing with. Each edge is of one of the following types:

jump No type flags are set for edges corresponding to jump instructions. These edges are used for unconditional or conditional jumps and in RTL also for table jumps. They are the easiest to manipulate as they may be freely redirected when the flow graph is not in SSA form.

fall-thru Fall-thru edges are present in case where the basic block may continue execution to the following one without branching. These edges have the `EDGE_FALLTHRU` flag set. Unlike other types of edges, these edges must come into the basic block immediately following in the instruction stream. The function `force_nonfallthru` is available to insert an unconditional jump in the case that redirection is needed. Note that this may require creation of a new basic block.

exception handling
Exception handling edges represent possible control transfers from a trapping instruction to an exception handler. The definition of "trapping" varies. In C++, only function calls can throw, but for Java, exceptions like division by zero or segmentation fault are defined and thus each instruction possibly throwing this kind of exception needs to be handled as control flow instruction. Exception edges have the `EDGE_ABNORMAL` and `EDGE_EH` flags set.

When updating the instruction stream it is easy to change possibly trapping instruction to non-trapping, by simply removing the exception edge. The opposite conversion is difficult, but should not happen anyway. The edges can be eliminated via `purge_dead_edges` call.

In the RTL representation, the destination of an exception edge is specified by `REG_EH_REGION` note attached to the insn. In case of a trapping call the `EDGE_ABNORMAL_CALL` flag is set too. In the `tree` representation, this extra flag is not set.

In the RTL representation, the predicate `may_trap_p` may be used to check whether instruction still may trap or not. For the tree representation, the `tree_could_trap_p` predicate is available, but this predicate only checks for possible memory traps, as in dereferencing an invalid pointer location.

sibling calls

Sibling calls or tail calls terminate the function in a non-standard way and thus an edge to the exit must be present. `EDGE_SIBCALL` and `EDGE_ABNORMAL` are set in such case. These edges only exist in the RTL representation.

computed jumps

Computed jumps contain edges to all labels in the function referenced from the code. All those edges have `EDGE_ABNORMAL` flag set. The edges used to represent computed jumps often cause compile time performance problems, since functions consisting of many taken labels and many computed jumps may have *very* dense flow graphs, so these edges need to be handled with special care. During the earlier stages of the compilation process, GCC tries to avoid such dense flow graphs by factoring computed jumps. For example, given the following series of jumps,

```
goto *x;
[ ... ]

goto *x;
[ ... ]

goto *x;
[ ... ]
```

factoring the computed jumps results in the following code sequence which has a much simpler flow graph:

```
goto y;
[ ... ]

goto y;
[ ... ]

goto y;
[ ... ]

y:
  goto *x;
```

However, the classic problem with this transformation is that it has a runtime cost in there resulting code: An extra jump. Therefore, the computed jumps are un-factored in the later passes of the compiler. Be aware of that when

you work on passes in that area. There have been numerous examples already where the compile time for code with unfactored computed jumps caused some serious headaches.

nonlocal goto handlers
: GCC allows nested functions to return into caller using a `goto` to a label passed to as an argument to the callee. The labels passed to nested functions contain special code to cleanup after function call. Such sections of code are referred to as "nonlocal goto receivers". If a function contains such nonlocal goto receivers, an edge from the call to the label is created with the `EDGE_ABNORMAL` and `EDGE_ABNORMAL_CALL` flags set.

function entry points
: By definition, execution of function starts at basic block 0, so there is always an edge from the `ENTRY_BLOCK_PTR` to basic block 0. There is no `tree` representation for alternate entry points at this moment. In RTL, alternate entry points are specified by `CODE_LABEL` with `LABEL_ALTERNATE_NAME` defined. This feature is currently used for multiple entry point prologues and is limited to post-reload passes only. This can be used by back-ends to emit alternate prologues for functions called from different contexts. In future full support for multiple entry functions defined by Fortran 90 needs to be implemented.

function exits
: In the pre-reload representation a function terminates after the last instruction in the insn chain and no explicit return instructions are used. This corresponds to the fall-thru edge into exit block. After reload, optimal RTL epilogues are used that use explicit (conditional) return instructions that are represented by edges with no flags set.

12.3 Profile information

In many cases a compiler must make a choice whether to trade speed in one part of code for speed in another, or to trade code size for code speed. In such cases it is useful to know information about how often some given block will be executed. That is the purpose for maintaining profile within the flow graph. GCC can handle profile information obtained through *profile feedback*, but it can also estimate branch probabilities based on statics and heuristics.

The feedback based profile is produced by compiling the program with instrumentation, executing it on a train run and reading the numbers of executions of basic blocks and edges back to the compiler while re-compiling the program to produce the final executable. This method provides very accurate information about where a program spends most of its time on the train run. Whether it matches the average run of course depends on the choice of train data set, but several studies have shown that the behavior of a program usually changes just marginally over different data sets.

When profile feedback is not available, the compiler may be asked to attempt to predict the behavior of each branch in the program using a set of heuristics (see '`predict.def`' for details) and compute estimated frequencies of each basic block by propagating the probabilities over the graph.

Each `basic_block` contains two integer fields to represent profile information: `frequency` and `count`. The `frequency` is an estimation how often is basic block executed within a function. It is represented as an integer scaled in the range from 0 to `BB_FREQ_BASE`. The most frequently executed basic block in function is initially set to `BB_FREQ_BASE` and the rest of frequencies are scaled accordingly. During optimization, the frequency of the most frequent basic block can both decrease (for instance by loop unrolling) or grow (for instance by cross-jumping optimization), so scaling sometimes has to be performed multiple times.

The `count` contains hard-counted numbers of execution measured during training runs and is nonzero only when profile feedback is available. This value is represented as the host's widest integer (typically a 64 bit integer) of the special type `gcov_type`.

Most optimization passes can use only the frequency information of a basic block, but a few passes may want to know hard execution counts. The frequencies should always match the counts after scaling, however during updating of the profile information numerical error may accumulate into quite large errors.

Each edge also contains a branch probability field: an integer in the range from 0 to `REG_BR_PROB_BASE`. It represents probability of passing control from the end of the `src` basic block to the `dest` basic block, i.e. the probability that control will flow along this edge. The `EDGE_FREQUENCY` macro is available to compute how frequently a given edge is taken. There is a `count` field for each edge as well, representing same information as for a basic block.

The basic block frequencies are not represented in the instruction stream, but in the RTL representation the edge frequencies are represented for conditional jumps (via the `REG_BR_PROB` macro) since they are used when instructions are output to the assembly file and the flow graph is no longer maintained.

The probability that control flow arrives via a given edge to its destination basic block is called *reverse probability* and is not directly represented, but it may be easily computed from frequencies of basic blocks.

Updating profile information is a delicate task that can unfortunately not be easily integrated with the CFG manipulation API. Many of the functions and hooks to modify the CFG, such as `redirect_edge_and_branch`, do not have enough information to easily update the profile, so updating it is in the majority of cases left up to the caller. It is difficult to uncover bugs in the profile updating code, because they manifest themselves only by producing worse code, and checking profile consistency is not possible because of numeric error accumulation. Hence special attention needs to be given to this issue in each pass that modifies the CFG.

It is important to point out that `REG_BR_PROB_BASE` and `BB_FREQ_BASE` are both set low enough to be possible to compute second power of any frequency or probability in the flow graph, it is not possible to even square the `count` field, as modern CPUs are fast enough to execute 2^32 operations quickly.

12.4 Maintaining the CFG

An important task of each compiler pass is to keep both the control flow graph and all profile information up-to-date. Reconstruction of the control flow graph after each pass is not an option, since it may be very expensive and lost profile information cannot be reconstructed at all.

GCC has two major intermediate representations, and both use the `basic_block` and `edge` data types to represent control flow. Both representations share as much of the CFG maintenance code as possible. For each representation, a set of *hooks* is defined so that each representation can provide its own implementation of CFG manipulation routines when necessary. These hooks are defined in '`cfghooks.h`'. There are hooks for almost all common CFG manipulations, including block splitting and merging, edge redirection and creating and deleting basic blocks. These hooks should provide everything you need to maintain and manipulate the CFG in both the RTL and `tree` representation.

At the moment, the basic block boundaries are maintained transparently when modifying instructions, so there rarely is a need to move them manually (such as in case someone wants to output instruction outside basic block explicitly). Often the CFG may be better viewed as integral part of instruction chain, than structure built on the top of it. However, in principle the control flow graph for the `tree` representation is *not* an integral part of the representation, in that a function tree may be expanded without first building a flow graph for the `tree` representation at all. This happens when compiling without any `tree` optimization enabled. When the `tree` optimizations are enabled and the instruction stream is rewritten in SSA form, the CFG is very tightly coupled with the instruction stream. In particular, statement insertion and removal has to be done with care. In fact, the whole `tree` representation can not be easily used or maintained without proper maintenance of the CFG simultaneously.

In the RTL representation, each instruction has a `BLOCK_FOR_INSN` value that represents pointer to the basic block that contains the instruction. In the `tree` representation, the function `bb_for_stmt` returns a pointer to the basic block containing the queried statement.

When changes need to be applied to a function in its `tree` representation, *block statement iterators* should be used. These iterators provide an integrated abstraction of the flow graph and the instruction stream. Block statement iterators iterators are constructed using the `block_stmt_iterator` data structure and several modifier are available, including the following:

`bsi_start`
: This function initializes a `block_stmt_iterator` that points to the first non-empty statement in a basic block.

`bsi_last`
: This function initializes a `block_stmt_iterator` that points to the last statement in a basic block.

`bsi_end_p`
: This predicate is `true` if a `block_stmt_iterator` represents the end of a basic block.

`bsi_next`
: This function takes a `block_stmt_iterator` and makes it point to its successor.

`bsi_prev`
: This function takes a `block_stmt_iterator` and makes it point to its predecessor.

`bsi_insert_after`
: This function inserts a statement after the `block_stmt_iterator` passed in. The final parameter determines whether the statement iterator is updated to point to the newly inserted statement, or left pointing to the original statement.

`bsi_insert_before`
: This function inserts a statement before the `block_stmt_iterator` passed in. The final parameter determines whether the statement iterator is updated to point to the newly inserted statement, or left pointing to the original statement.

`bsi_remove`
: This function removes the `block_stmt_iterator` passed in and rechains the remaining statements in a basic block, if any.

In the RTL representation, the macros `BB_HEAD` and `BB_END` may be used to get the head and end `rtx` of a basic block. No abstract iterators are defined for traversing the insn chain, but you can just use `NEXT_INSN` and `PREV_INSN` instead. See See Section 11.18 [Insns], page 156.

Usually a code manipulating pass simplifies the instruction stream and the flow of control, possibly eliminating some edges. This may for example happen when a conditional jump is replaced with an unconditional jump, but also when simplifying possibly trapping instruction to non-trapping while compiling Java. Updating of edges is not transparent and each optimization pass is required to do so manually. However only few cases occur in practice. The pass may call `purge_dead_edges` on a given basic block to remove superfluous edges, if any.

Another common scenario is redirection of branch instructions, but this is best modeled as redirection of edges in the control flow graph and thus use of `redirect_edge_and_branch` is preferred over more low level functions, such as `redirect_jump` that operate on RTL chain only. The CFG hooks defined in '`cfghooks.h`' should provide the complete API required for manipulating and maintaining the CFG.

It is also possible that a pass has to insert control flow instruction into the middle of a basic block, thus creating an entry point in the middle of the basic block, which is impossible by definition: The block must be split to make sure it only has one entry point, i.e. the head of the basic block. The CFG hook `split_block` may be used when an instruction in the middle of a basic block has to become the target of a jump or branch instruction.

For a global optimizer, a common operation is to split edges in the flow graph and insert instructions on them. In the RTL representation, this can be easily done using the `insert_insn_on_edge` function that emits an instruction "on the edge", caching it for a later `commit_edge_insertions` call that will take care of moving the inserted instructions off the edge into the instruction stream contained in a basic block. This includes the creation of new basic blocks where needed. In the `tree` representation, the equivalent functions are `bsi_insert_on_edge` which inserts a block statement iterator on an edge, and `bsi_commit_edge_inserts` which flushes the instruction to actual instruction stream.

While debugging the optimization pass, an `verify_flow_info` function may be useful to find bugs in the control flow graph updating code.

Note that at present, the representation of control flow in the `tree` representation is discarded before expanding to RTL. Long term the CFG should be maintained and "expanded" to the RTL representation along with the function `tree` itself.

12.5 Liveness information

Liveness information is useful to determine whether some register is "live" at given point of program, i.e. that it contains a value that may be used at a later point in the program.

This information is used, for instance, during register allocation, as the pseudo registers only need to be assigned to a unique hard register or to a stack slot if they are live. The hard registers and stack slots may be freely reused for other values when a register is dead.

The liveness information is stored partly in the RTL instruction stream and partly in the flow graph. Local information is stored in the instruction stream: Each instruction may contain `REG_DEAD` notes representing that the value of a given register is no longer needed, or `REG_UNUSED` notes representing that the value computed by the instruction is never used. The second is useful for instructions computing multiple values at once.

Global liveness information is stored in the control flow graph. Each basic block contains two bitmaps, `global_live_at_start` and `global_live_at_end` representing liveness of each register at the entry and exit of the basic block. The file `flow.c` contains functions to compute liveness of each register at any given place in the instruction stream using this information.

Liveness is expensive to compute and thus it is desirable to keep it up to date during code modifying passes. This can be easily accomplished using the `flags` field of a basic block. Functions modifying the instruction stream automatically set the `BB_DIRTY` flag of a modifies basic block, so the pass may simply use`clear_bb_flags` before doing any modifications and then ask the data flow module to have liveness updated via the `update_life_info_in_dirty_blocks` function.

This scheme works reliably as long as no control flow graph transformations are done. The task of updating liveness after control flow graph changes is more difficult as normal iterative data flow analysis may produce invalid results or get into an infinite cycle when the initial solution is not below the desired one. Only simple transformations, like splitting basic blocks or inserting on edges, are safe, as functions to implement them already know how to update liveness information locally.

13 Machine Descriptions

A machine description has two parts: a file of instruction patterns (‘.md’ file) and a C header file of macro definitions.

The ‘.md’ file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about). It may also contain comments. A semicolon causes the rest of the line to be a comment, unless the semicolon is inside a quoted string.

See the next chapter for information on the C header file.

13.1 Overview of How the Machine Description is Used

There are three main conversions that happen in the compiler:

1. The front end reads the source code and builds a parse tree.
2. The parse tree is used to generate an RTL insn list based on named instruction patterns.
3. The insn list is matched against the RTL templates to produce assembler code.

For the generate pass, only the names of the insns matter, from either a named `define_insn` or a `define_expand`. The compiler will choose the pattern with the right name and apply the operands according to the documentation later in this chapter, without regard for the RTL template or operand constraints. Note that the names the compiler looks for are hard-coded in the compiler—it will ignore unnamed patterns and patterns with names it doesn't know about, but if you don't provide a named pattern it needs, it will abort.

If a `define_insn` is used, the template given is inserted into the insn list. If a `define_expand` is used, one of three things happens, based on the condition logic. The condition logic may manually create new insns for the insn list, say via `emit_insn()`, and invoke `DONE`. For certain named patterns, it may invoke `FAIL` to tell the compiler to use an alternate way of performing that task. If it invokes neither `DONE` nor `FAIL`, the template given in the pattern is inserted, as if the `define_expand` were a `define_insn`.

Once the insn list is generated, various optimization passes convert, replace, and rearrange the insns in the insn list. This is where the `define_split` and `define_peephole` patterns get used, for example.

Finally, the insn list's RTL is matched up with the RTL templates in the `define_insn` patterns, and those patterns are used to emit the final assembly code. For this purpose, each named `define_insn` acts like it's unnamed, since the names are ignored.

13.2 Everything about Instruction Patterns

Each instruction pattern contains an incomplete RTL expression, with pieces to be filled in later, operand constraints that restrict how the pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a `define_insn` expression.

A `define_insn` is an RTL expression containing four or five operands:

1. An optional name. The presence of a name indicate that this instruction pattern can perform a certain standard job for the RTL-generation pass of the compiler. This pass knows certain names and will use the instruction patterns with those names, if the names are defined in the machine description.

The absence of a name is indicated by writing an empty string where the name should go. Nameless instruction patterns are never used for generating RTL code, but they may permit several simpler insns to be combined later on.

Names that are not thus known and used in RTL-generation have no effect; they are equivalent to no name at all.

For the purpose of debugging the compiler, you may also specify a name beginning with the '`*`' character. Such a name is used only for identifying the instruction in RTL dumps; it is entirely equivalent to having a nameless pattern for all other purposes.

2. The *RTL template* (see Section 13.4 [RTL Template], page 181) is a vector of incomplete RTL expressions which show what the instruction should look like. It is incomplete because it may contain `match_operand`, `match_operator`, and `match_dup` expressions that stand for operands of the instruction.

   If the vector has only one element, that element is the template for the instruction pattern. If the vector has multiple elements, then the instruction pattern is a `parallel` expression containing the elements described.

3. A condition. This is a string which contains a C expression that is the final test to decide whether an insn body matches this pattern.

   For a named pattern, the condition (if present) may not depend on the data in the insn being matched, but only the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.

   For nameless patterns, the condition is applied only when matching an individual insn, and only after the insn has matched the pattern's recognition template. The insn's operands may be found in the vector `operands`. For an insn where the condition has once matched, it can't be used to control register allocation, for example by excluding certain hard registers or hard register combinations.

4. The *output template*: a string that says how to output matching insns as assembler code. '`%`' in this string specifies where to substitute the value of an operand. See Section 13.5 [Output Template], page 184.

   When simple substitution isn't general enough, you can specify a piece of C code to compute the output. See Section 13.6 [Output Statement], page 186.

5. Optionally, a vector containing the values of attributes for insns matching this pattern. See Section 13.19 [Insn Attributes], page 248.

13.3 Example of `define_insn`

Here is an actual example of an instruction pattern, for the 68000/68020.

```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
  "*
{
  if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
    return \"tstl %0\";
  return \"cmpl #0,%0\";
}")
```

This can also be written using braced strings:

```
(define_insn "tstsi"
  [(set (cc0)
        (match_operand:SI 0 "general_operand" "rm"))]
  ""
{
  if (TARGET_68020 || ! ADDRESS_REG_P (operands[0]))
    return "tstl %0";
  return "cmpl #0,%0";
})
```

This is an instruction that sets the condition codes based on the value of a general operand. It has no condition, so any insn whose RTL description has the form shown may be handled according to this pattern. The name '`tstsi`' means "test a `SImode` value" and tells the RTL generation pass that, when it is necessary to test such a value, an insn to do so can be constructed using this pattern.

The output control string is a piece of C code which chooses which output template to return based on the kind of operand and the specific type of CPU for which code is being generated.

'`"rm"`' is an operand constraint. Its meaning is explained below.

13.4 RTL Template

The RTL template is used to define which insns match the particular pattern and how to find their operands. For named patterns, the RTL template also says how to construct an insn from specified operands.

Construction involves substituting specified operands into a copy of the template. Matching involves determining the values that serve as the operands in the insn being matched. Both of these activities are controlled by special expression types that direct matching and substitution of the operands.

`(match_operand:`*m* *n predicate constraint*`)`

This expression is a placeholder for operand number *n* of the insn. When constructing an insn, operand number *n* will be substituted at this point. When matching an insn, whatever appears at this position in the insn will be taken as operand number *n*; but it must satisfy *predicate* or this instruction pattern will not match at all.

Operand numbers must be chosen consecutively counting from zero in each instruction pattern. There may be only one `match_operand` expression in the pattern for each operand number. Usually operands are numbered in the order of appearance in `match_operand` expressions. In the case of a `define_expand`, any operand numbers used only in `match_dup` expressions have higher values than all other operand numbers.

predicate is a string that is the name of a function that accepts two arguments, an expression and a machine mode. See Section 13.7 [Predicates], page 187. During matching, the function will be called with the putative operand as the expression and *m* as the mode argument (if *m* is not specified, `VOIDmode` will be used, which normally causes *predicate* to accept any mode). If it returns zero, this instruction pattern fails to match. *predicate* may be an empty string; then

it means no test is to be done on the operand, so anything which occurs in this position is valid.

Most of the time, *predicate* will reject modes other than *m*—but not always. For example, the predicate `address_operand` uses *m* as the mode of memory ref that the address should be valid for. Many predicates accept `const_int` nodes even though their mode is `VOIDmode`.

constraint controls reloading and the choice of the best register class to use for a value, as explained later (see Section 13.8 [Constraints], page 191). If the constraint would be an empty string, it can be omitted.

People are often unclear on the difference between the constraint and the predicate. The predicate helps decide whether a given insn matches the pattern. The constraint plays no role in this decision; instead, it controls various decisions in the case of an insn which does match.

`(match_scratch:`*m* *n* *constraint*`)`

This expression is also a placeholder for operand number *n* and indicates that operand must be a `scratch` or `reg` expression.

When matching patterns, this is equivalent to

```
(match_operand:m n "scratch_operand" pred)
```

but, when generating RTL, it produces a (`scratch`:*m*) expression.

If the last few expressions in a `parallel` are `clobber` expressions whose operands are either a hard register or `match_scratch`, the combiner can add or delete them when necessary. See Section 11.15 [Side Effects], page 149.

`(match_dup` *n*`)`

This expression is also a placeholder for operand number *n*. It is used when the operand needs to appear more than once in the insn.

In construction, `match_dup` acts just like `match_operand`: the operand is substituted into the insn being constructed. But in matching, `match_dup` behaves differently. It assumes that operand number *n* has already been determined by a `match_operand` appearing earlier in the recognition template, and it matches only an identical-looking expression.

Note that `match_dup` should not be used to tell the compiler that a particular register is being used for two operands (example: `add` that adds one register to another; the second register is both an input operand and the output operand). Use a matching constraint (see Section 13.8.1 [Simple Constraints], page 191) for those. `match_dup` is for the cases where one operand is used in two places in the template, such as an instruction that computes both a quotient and a remainder, where the opcode takes two input operands but the RTL template has to refer to each of those twice; once for the quotient pattern and once for the remainder pattern.

`(match_operator:`*m* *n* *predicate* `[`*operands*`...])`

This pattern is a kind of placeholder for a variable RTL expression code.

When constructing an insn, it stands for an RTL expression whose expression code is taken from that of operand *n*, and whose operands are constructed from the patterns *operands*.

When matching an expression, it matches an expression if the function *predicate* returns nonzero on that expression *and* the patterns *operands* match the operands of the expression.

Suppose that the function `commutative_operator` is defined as follows, to match any expression whose operator is one of the commutative arithmetic operators of RTL and whose mode is *mode*:

```
int
commutative_integer_operator (x, mode)
     rtx x;
     enum machine_mode mode;
{
  enum rtx_code code = GET_CODE (x);
  if (GET_MODE (x) != mode)
    return 0;
  return (GET_RTX_CLASS (code) == RTX_COMM_ARITH
          || code == EQ || code == NE);
}
```

Then the following pattern will match any RTL expression consisting of a commutative operator applied to two general operands:

```
(match_operator:SI 3 "commutative_operator"
  [(match_operand:SI 1 "general_operand" "g")
   (match_operand:SI 2 "general_operand" "g")])
```

Here the vector `[`*`operands`*`...]` contains two patterns because the expressions to be matched all contain two operands.

When this pattern does match, the two operands of the commutative operator are recorded as operands 1 and 2 of the insn. (This is done by the two instances of `match_operand`.) Operand 3 of the insn will be the entire commutative expression: use `GET_CODE (operands[3])` to see which commutative operator was used.

The machine mode *m* of `match_operator` works like that of `match_operand`: it is passed as the second argument to the predicate function, and that function is solely responsible for deciding whether the expression to be matched "has" that mode.

When constructing an insn, argument 3 of the gen-function will specify the operation (i.e. the expression code) for the expression to be made. It should be an RTL expression, whose expression code is copied into a new expression whose operands are arguments 1 and 2 of the gen-function. The subexpressions of argument 3 are not used; only its expression code matters.

When `match_operator` is used in a pattern for matching an insn, it usually best if the operand number of the `match_operator` is higher than that of the actual operands of the insn. This improves register allocation because the register allocator often looks at operands 1 and 2 of insns to see if it can do register tying.

There is no way to specify constraints in `match_operator`. The operand of the insn which corresponds to the `match_operator` never has any constraints because it is never reloaded as a whole. However, if parts of its *operands* are matched by `match_operand` patterns, those parts may have constraints of their own.

`(match_op_dup:`*m* *n* `[`*operands*`...])`

Like `match_dup`, except that it applies to operators instead of operands. When constructing an insn, operand number *n* will be substituted at this point. But in matching, `match_op_dup` behaves differently. It assumes that operand number *n* has already been determined by a `match_operator` appearing earlier in the recognition template, and it matches only an identical-looking expression.

`(match_parallel` *n predicate* `[`*subpat*`...])`

This pattern is a placeholder for an insn that consists of a `parallel` expression with a variable number of elements. This expression should only appear at the top level of an insn pattern.

When constructing an insn, operand number *n* will be substituted at this point. When matching an insn, it matches if the body of the insn is a `parallel` expression with at least as many elements as the vector of *subpat* expressions in the `match_parallel`, if each *subpat* matches the corresponding element of the `parallel`, *and* the function *predicate* returns nonzero on the `parallel` that is the body of the insn. It is the responsibility of the predicate to validate elements of the `parallel` beyond those listed in the `match_parallel`.

A typical use of `match_parallel` is to match load and store multiple expressions, which can contain a variable number of elements in a `parallel`. For example,

```
(define_insn ""
  [(match_parallel 0 "load_multiple_operation"
     [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
           (match_operand:SI 2 "memory_operand" "m"))
      (use (reg:SI 179))
      (clobber (reg:SI 179))])]
  ""
  "loadm 0,0,%1,%2")
```

This example comes from '`a29k.md`'. The function `load_multiple_operation` is defined in '`a29k.c`' and checks that subsequent elements in the `parallel` are the same as the `set` in the pattern, except that they are referencing subsequent registers and memory locations.

An insn that matches this pattern might look like:

```
(parallel
 [(set (reg:SI 20) (mem:SI (reg:SI 100)))
  (use (reg:SI 179))
  (clobber (reg:SI 179))
  (set (reg:SI 21)
       (mem:SI (plus:SI (reg:SI 100)
                        (const_int 4))))
  (set (reg:SI 22)
       (mem:SI (plus:SI (reg:SI 100)
                        (const_int 8))))])
```

`(match_par_dup` *n* `[`*subpat*`...])`

Like `match_op_dup`, but for `match_parallel` instead of `match_operator`.

13.5 Output Templates and Operand Substitution

The *output template* is a string which specifies how to output the assembler code for an instruction pattern. Most of the template is a fixed string which is output literally. The character '`%`' is used to specify where to substitute an operand; it can also be used to identify places where different variants of the assembler require different syntax.

In the simplest case, a '`%`' followed by a digit *n* says to output operand *n* at that point in the string.

'`%`' followed by a letter and a digit says to output an operand in an alternate fashion. Four letters have standard, built-in meanings described below. The machine description macro `PRINT_OPERAND` can define additional letters with nonstandard meanings.

'`%c`*digit*' can be used to substitute an operand that is a constant value without the syntax that normally indicates an immediate operand.

'`%n`*digit*' is like '`%c`*digit*' except that the value of the constant is negated before printing.

'`%a`*digit*' can be used to substitute an operand as if it were a memory reference, with the actual operand treated as the address. This may be useful when outputting a "load address" instruction, because often the assembler syntax for such an instruction requires you to write the operand as if it were a memory reference.

'`%l`*digit*' is used to substitute a `label_ref` into a jump instruction.

'`%=`' outputs a number which is unique to each instruction in the entire compilation. This is useful for making local labels to be referred to more than once in a single template that generates multiple assembler instructions.

'`%`' followed by a punctuation character specifies a substitution that does not use an operand. Only one case is standard: '`%%`' outputs a '`%`' into the assembler code. Other nonstandard cases can be defined in the `PRINT_OPERAND` macro. You must also define which punctuation characters are valid with the `PRINT_OPERAND_PUNCT_VALID_P` macro.

The template may generate multiple assembler instructions. Write the text for the instructions, with '`\;`' between them.

When the RTL contains two operands which are required by constraint to match each other, the output template must refer only to the lower-numbered operand. Matching operands are not always identical, and the rest of the compiler arranges to put the proper RTL expression for printing into the lower-numbered operand.

One use of nonstandard letters or punctuation following '`%`' is to distinguish between different assembler languages for the same machine; for example, Motorola syntax versus MIT syntax for the 68000. Motorola syntax requires periods in most opcode names, while MIT syntax does not. For example, the opcode '`movel`' in MIT syntax is '`move.l`' in Motorola syntax. The same file of patterns is used for both kinds of output syntax, but the character sequence '`%.`' is used in each place where Motorola syntax wants a period. The `PRINT_OPERAND` macro for Motorola syntax defines the sequence to output a period; the macro for MIT syntax defines it to do nothing.

As a special case, a template consisting of the single character `#` instructs the compiler to first split the insn, and then output the resulting instructions separately. This helps eliminate redundancy in the output templates. If you have a `define_insn` that needs to emit multiple assembler instructions, and there is an matching `define_split` already

defined, then you can simply use `#` as the output template instead of writing an output template that emits the multiple assembler instructions.

If the macro `ASSEMBLER_DIALECT` is defined, you can use construct of the form '`{option0|option1|option2}`' in the templates. These describe multiple variants of assembler language syntax. See Section 14.19.7 [Instruction Output], page 370.

13.6 C Statements for Assembler Output

Often a single fixed template string cannot produce correct and efficient assembler code for all the cases that are recognized by a single instruction pattern. For example, the opcodes may depend on the kinds of operands; or some unfortunate combinations of operands may require extra machine instructions.

If the output control string starts with a '`@`', then it is actually a series of templates, each on a separate line. (Blank lines and leading spaces and tabs are ignored.) The templates correspond to the pattern's constraint alternatives (see Section 13.8.2 [Multi-Alternative], page 195). For example, if a target machine has a two-address add instruction '`addr`' to add into a register and another '`addm`' to add a register to memory, you might write this pattern:

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_operand" "=r,m")
        (plus:SI (match_operand:SI 1 "general_operand" "0,0")
                 (match_operand:SI 2 "general_operand" "g,r")))]
  ""
  "@
   addr %2,%0
   addm %2,%0")
```

If the output control string starts with a '`*`', then it is not an output template but rather a piece of C program that should compute a template. It should execute a `return` statement to return the template-string you want. Most such templates use C string literals, which require doublequote characters to delimit them. To include these doublequote characters in the string, prefix each one with '\'.

If the output control string is written as a brace block instead of a double-quoted string, it is automatically assumed to be C code. In that case, it is not necessary to put in a leading asterisk, or to escape the doublequotes surrounding C string literals.

The operands may be found in the array `operands`, whose C data type is `rtx []`.

It is very common to select different ways of generating assembler code based on whether an immediate operand is within a certain range. Be careful when doing this, because the result of `INTVAL` is an integer on the host machine. If the host machine has more bits in an `int` than the target machine has in the mode in which the constant will be used, then some of the bits you get from `INTVAL` will be superfluous. For proper results, you must carefully disregard the values of those bits.

It is possible to output an assembler instruction and then go on to output or compute more of them, using the subroutine `output_asm_insn`. This receives two arguments: a template-string and a vector of operands. The vector may be `operands`, or it may be another array of `rtx` that you declare locally and initialize yourself.

When an insn pattern has multiple alternatives in its constraints, often the appearance of the assembler code is determined mostly by which alternative was matched. When this

is so, the C code can test the variable `which_alternative`, which is the ordinal number of the alternative that was actually satisfied (0 for the first, 1 for the second alternative, etc.).

For example, suppose there are two opcodes for storing zero, '`clrreg`' for registers and '`clrmem`' for memory locations. Here is how a pattern could use `which_alternative` to choose between them:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r,m")
        (const_int 0))]
  ""
  {
  return (which_alternative == 0
          ? "clrreg %0" : "clrmem %0");
  })
```

The example above, where the assembler code to generate was *solely* determined by the alternative, could also have been specified as follows, having the output control string start with a '`@`':

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r,m")
        (const_int 0))]
  ""
  "@
   clrreg %0
   clrmem %0")
```

13.7 Predicates

A predicate determines whether a `match_operand` or `match_operator` expression matches, and therefore whether the surrounding instruction pattern will be used for that combination of operands. GCC has a number of machine-independent predicates, and you can define machine-specific predicates as needed. By convention, predicates used with `match_operand` have names that end in '`_operand`', and those used with `match_operator` have names that end in '`_operator`'.

All predicates are Boolean functions (in the mathematical sense) of two arguments: the RTL expression that is being considered at that position in the instruction pattern, and the machine mode that the `match_operand` or `match_operator` specifies. In this section, the first argument is called *op* and the second argument *mode*. Predicates can be called from C as ordinary two-argument functions; this can be useful in output templates or other machine-specific code.

Operand predicates can allow operands that are not actually acceptable to the hardware, as long as the constraints give reload the ability to fix them up (see Section 13.8 [Constraints], page 191). However, GCC will usually generate better code if the predicates specify the requirements of the machine instructions as closely as possible. Reload cannot fix up operands that must be constants ("immediate operands"); you must use a predicate that allows only constants, or else enforce the requirement in the extra condition.

Most predicates handle their *mode* argument in a uniform manner. If *mode* is `VOIDmode` (unspecified), then *op* can have any mode. If *mode* is anything else, then *op* must have the same mode, unless *op* is a `CONST_INT` or integer `CONST_DOUBLE`. These RTL expressions always have `VOIDmode`, so it would be counterproductive to check that their mode matches.

Instead, predicates that accept `CONST_INT` and/or integer `CONST_DOUBLE` check that the value stored in the constant will fit in the requested mode.

Predicates with this behavior are called *normal*. `genrecog` can optimize the instruction recognizer based on knowledge of how normal predicates treat modes. It can also diagnose certain kinds of common errors in the use of normal predicates; for instance, it is almost always an error to use a normal predicate without specifying a mode.

Predicates that do something different with their *mode* argument are called *special*. The generic predicates `address_operand` and `pmode_register_operand` are special predicates. `genrecog` does not do any optimizations or diagnosis when special predicates are used.

13.7.1 Machine-Independent Predicates

These are the generic predicates available to all back ends. They are defined in '`recog.c`'. The first category of predicates allow only constant, or *immediate*, operands.

`immediate_operand` [Function]
: This predicate allows any sort of constant that fits in *mode*. It is an appropriate choice for instructions that take operands that must be constant.

`const_int_operand` [Function]
: This predicate allows any `CONST_INT` expression that fits in *mode*. It is an appropriate choice for an immediate operand that does not allow a symbol or label.

`const_double_operand` [Function]
: This predicate accepts any `CONST_DOUBLE` expression that has exactly *mode*. If *mode* is `VOIDmode`, it will also accept `CONST_INT`. It is intended for immediate floating point constants.

The second category of predicates allow only some kind of machine register.

`register_operand` [Function]
: This predicate allows any `REG` or `SUBREG` expression that is valid for *mode*. It is often suitable for arithmetic instruction operands on a RISC machine.

`pmode_register_operand` [Function]
: This is a slight variant on `register_operand` which works around a limitation in the machine-description reader.

```
(match_operand n "pmode_register_operand" constraint)
```

means exactly what

```
(match_operand:P n "register_operand" constraint)
```

would mean, if the machine-description reader accepted '`:P`' mode suffixes. Unfortunately, it cannot, because `Pmode` is an alias for some other mode, and might vary with machine-specific options. See Section 14.27 [Misc], page 390.

`scratch_operand` [Function]
: This predicate allows hard registers and `SCRATCH` expressions, but not pseudo-registers. It is used internally by `match_scratch`; it should not be used directly.

The third category of predicates allow only some kind of memory reference.

`memory_operand` [Function]
: This predicate allows any valid reference to a quantity of mode *mode* in memory, as determined by the weak form of `GO_IF_LEGITIMATE_ADDRESS` (see Section 14.13 [Addressing Modes], page 334).

`address_operand` [Function]
: This predicate is a little unusual; it allows any operand that is a valid expression for the *address* of a quantity of mode *mode*, again determined by the weak form of `GO_IF_LEGITIMATE_ADDRESS`. To first order, if '`(mem:`*mode* `(`*exp*`))`' is acceptable to `memory_operand`, then *exp* is acceptable to `address_operand`. Note that *exp* does not necessarily have the mode *mode*.

`indirect_operand` [Function]
: This is a stricter form of `memory_operand` which allows only memory references with a `general_operand` as the address expression. New uses of this predicate are discouraged, because `general_operand` is very permissive, so it's hard to tell what an `indirect_operand` does or does not allow. If a target has different requirements for memory operands for different instructions, it is better to define target-specific predicates which enforce the hardware's requirements explicitly.

`push_operand` [Function]
: This predicate allows a memory reference suitable for pushing a value onto the stack. This will be a `MEM` which refers to `stack_pointer_rtx`, with a side-effect in its address expression (see Section 11.16 [Incdec], page 154); which one is determined by the `STACK_PUSH_CODE` macro (see Section 14.9.1 [Frame Layout], page 304).

`pop_operand` [Function]
: This predicate allows a memory reference suitable for popping a value off the stack. Again, this will be a `MEM` referring to `stack_pointer_rtx`, with a side-effect in its address expression. However, this time `STACK_POP_CODE` is expected.

The fourth category of predicates allow some combination of the above operands.

`nonmemory_operand` [Function]
: This predicate allows any immediate or register operand valid for *mode*.

`nonimmediate_operand` [Function]
: This predicate allows any register or memory operand valid for *mode*.

`general_operand` [Function]
: This predicate allows any immediate, register, or memory operand valid for *mode*.

Finally, there is one generic operator predicate.

`comparison_operator` [Function]
: This predicate matches any expression which performs an arithmetic comparison in *mode*; that is, `COMPARISON_P` is true for the expression code.

13.7.2 Defining Machine-Specific Predicates

Many machines have requirements for their operands that cannot be expressed precisely using the generic predicates. You can define additional predicates using `define_predicate` and `define_special_predicate` expressions. These expressions have three operands:

- The name of the predicate, as it will be referred to in `match_operand` or `match_operator` expressions.
- An RTL expression which evaluates to true if the predicate allows the operand *op*, false if it does not. This expression can only use the following RTL codes:

  `MATCH_OPERAND`
  : When written inside a predicate expression, a `MATCH_OPERAND` expression evaluates to true if the predicate it names would allow *op*. The operand number and constraint are ignored. Due to limitations in `genrecog`, you can only refer to generic predicates and predicates that have already been defined.

  `MATCH_CODE`
  : This expression has one operand, a string constant containing a comma-separated list of RTX code names (in lower case). It evaluates to true if *op* has any of the listed codes.

  `MATCH_TEST`
  : This expression has one operand, a string constant containing a C expression. The predicate's arguments, *op* and *mode*, are available with those names in the C expression. The `MATCH_TEST` evaluates to true if the C expression evaluates to a nonzero value. `MATCH_TEST` expressions must not have side effects.

  `AND`
  `IOR`
  `NOT`
  `IF_THEN_ELSE`
  : The basic '`MATCH_`' expressions can be combined using these logical operators, which have the semantics of the C operators '`&&`', '`||`', '`!`', and '`? :`' respectively.

- An optional block of C code, which should execute '`return true`' if the predicate is found to match and '`return false`' if it does not. It must not have any side effects. The predicate arguments, *op* and *mode*, are available with those names.

  If a code block is present in a predicate definition, then the RTL expression must evaluate to true *and* the code block must execute '`return true`' for the predicate to allow the operand. The RTL expression is evaluated first; do not re-check anything in the code block that was checked in the RTL expression.

The program `genrecog` scans `define_predicate` and `define_special_predicate` expressions to determine which RTX codes are possibly allowed. You should always make this explicit in the RTL predicate expression, using `MATCH_OPERAND` and `MATCH_CODE`.

Here is an example of a simple predicate definition, from the IA64 machine description:

```
;; True if op is a SYMBOL_REF which refers to the sdata section.
(define_predicate "small_addr_symbolic_operand"
  (and (match_code "symbol_ref")
       (match_test "SYMBOL_REF_SMALL_ADDR_P (op)")))
```

And here is another, showing the use of the C block.

```
;; True if op is a register operand that is (or could be) a GR reg.
(define_predicate "gr_register_operand"
  (match_operand 0 "register_operand")
{
  unsigned int regno;
  if (GET_CODE (op) == SUBREG)
    op = SUBREG_REG (op);

  regno = REGNO (op);
  return (regno >= FIRST_PSEUDO_REGISTER || GENERAL_REGNO_P (regno));
})
```

Predicates written with `define_predicate` automatically include a test that *mode* is `VOIDmode`, or *op* has the same mode as *mode*, or *op* is a `CONST_INT` or `CONST_DOUBLE`. They do *not* check specifically for integer `CONST_DOUBLE`, nor do they test that the value of either kind of constant fits in the requested mode. This is because target-specific predicates that take constants usually have to do more stringent value checks anyway. If you need the exact same treatment of `CONST_INT` or `CONST_DOUBLE` that the generic predicates provide, use a `MATCH_OPERAND` subexpression to call `const_int_operand`, `const_double_operand`, or `immediate_operand`.

Predicates written with `define_special_predicate` do not get any automatic mode checks, and are treated as having special mode handling by `genrecog`.

The program `genpreds` is responsible for generating code to test predicates. It also writes a header file containing function declarations for all machine-specific predicates. It is not necessary to declare these predicates in '*cpu*`-protos.h`'.

13.8 Operand Constraints

Each `match_operand` in an instruction pattern can specify constraints for the operands allowed. The constraints allow you to fine-tune matching within the set of operands allowed by the predicate.

Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match.

13.8.1 Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

whitespace

> Whitespace characters are ignored and can be inserted at any position except the first. This enables each alternative for different operands to be visually aligned in the machine description even if they have different number of constraints and modifiers.

‘m’ A memory operand is allowed, with any kind of address that the machine supports in general.

‘o’ A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter ‘o’ is valid only when accompanied by both ‘<’ (if the target machine has predecrement addressing) and ‘>’ (if the target machine has preincrement addressing).

‘V’ A memory operand that is not offsettable. In other words, anything that would fit the ‘m’ constraint but not the ‘o’ constraint.

‘<’ A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed.

‘>’ A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed.

‘r’ A register operand is allowed provided that it is in a general register.

‘i’ An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time or later.

‘n’ An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use ‘n’ rather than ‘i’.

‘I’, ‘J’, ‘K’, . . . ‘P’
Other letters in the range ‘I’ through ‘P’ may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, ‘I’ is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

‘E’ An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

‘F’ An immediate floating operand (expression code `const_double` or `const_vector`) is allowed.

‘G’, ‘H’ ‘G’ and ‘H’ may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

'`s`' An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use '`s`' instead of '`i`'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between −128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a '`moveq`' instruction. We arrange for this to happen by defining the letter '`K`' to mean "any integer outside the range −128 to 127", and then specifying '`Ks`' in the operand constraints.

'`g`' Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

'`X`' Any operand whatsoever is allowed, even if it does not satisfy `general_operand`. This is normally used in the constraint of a `match_scratch` when certain alternatives will not actually require a scratch register.

'`0`', '`1`', '`2`', . . . '`9`'
An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This number is allowed to be more than a single digit. If multiple digits are encountered consecutively, they are interpreted as a single decimal integer. There is scant chance for ambiguity, since to-date it has never been desirable that '`10`' be interpreted as matching either operand 1 *or* operand 0. Should this be desired, one can use multiple alternatives instead.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles considered separate in the RTL insn. For example, an add insn has two input operands and one output operand in the RTL, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

For operands to match in a particular case usually means that they are identical-looking RTL expressions. But in a few special cases specific kinds of dissimilarity are allowed. For example, `*x` as an input operand will match `*x++` as an output operand. For proper results in such cases, the output template should always use the output-operand's number when printing the operand.

'`p`' An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

'`p`' in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the

`match_operand` as the mode of the memory reference for which the address would be valid.

other-letters
: Other letters can be defined in machine-dependent fashion to stand for particular classes of registers or other arbitrary operand types. '`d`', '`a`' and '`f`' are defined on the 68000/68020 to stand for data, address and floating point registers.

  The machine description macro `REG_CLASS_FROM_LETTER` has first cut at the otherwise unused letters. If it evaluates to `NO_REGS`, then `EXTRA_CONSTRAINT` is evaluated.

  A typical use for `EXTRA_CONSTRAINT` would be to distinguish certain types of memory references that affect other insn operands.

In order to have valid assembler code, each operand must satisfy its constraint. But a failure to do so does not prevent the pattern from applying to an insn. Instead, it directs the compiler to modify the code so that the constraint will be satisfied. Usually this is done by copying an operand into a register.

Contrast, therefore, the two instruction patterns that follow:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r")
        (plus:SI (match_dup 0)
                 (match_operand:SI 1 "general_operand" "r")))]
  ""
  "...")
```

which has two operands, one of which must appear in two places, and

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r")
        (plus:SI (match_operand:SI 1 "general_operand" "0")
                 (match_operand:SI 2 "general_operand" "r")))]
  ""
  "...")
```

which has three operands, two of which are required by a constraint to be identical. If we are considering an insn of the form

```
(insn n prev next
  (set (reg:SI 3)
       (plus:SI (reg:SI 6) (reg:SI 109)))
  ...)
```

the first pattern would not apply at all, because this insn does not contain two identical subexpressions in the right place. The pattern would say, "That does not look like an add instruction; try other patterns". The second pattern would say, "Yes, that's an add instruction, but there is something wrong with it". It would direct the reload pass of the compiler to generate additional insns to make the constraint true. The results might look like this:

```
(insn n2 prev n
  (set (reg:SI 3) (reg:SI 6))
  ...)

(insn n n2 next
  (set (reg:SI 3)
```

```
        (plus:SI (reg:SI 3) (reg:SI 109)))
    ...)
```

It is up to you to make sure that each operand, in each pattern, has constraints that can handle any RTL expression that could be present for that operand. (When multiple alternatives are in use, each pattern must, for each possible combination of operand expressions, have at least one alternative which can handle that combination of operands.) The constraints don't need to *allow* any possible operand—when this is the case, they do not constrain—but they must at least point the way to reloading any possible operand so that it will fit.

- If the constraint accepts whatever operands the predicate permits, there is no problem: reloading is never necessary for this operand.

  For example, an operand whose constraints permit everything except registers is safe provided its predicate rejects registers.

  An operand whose predicate accepts only constant values is safe provided its constraints include the letter '`i`'. If any possible constant value is accepted, then nothing less than '`i`' will do; if the predicate is more selective, then the constraints may also be more selective.

- Any operand expression can be reloaded by copying it into a register. So if an operand's constraints allow some kind of register, it is certain to be safe. It need not permit all classes of registers; the compiler knows how to copy a register into another register of the proper class in order to make an instruction valid.

- A nonoffsettable memory reference can be reloaded by copying the address into a register. So if the constraint uses the letter '`o`', all memory references are taken care of.

- A constant operand can be reloaded by allocating space in memory to hold it as preinitialized data. Then the memory reference can be used in place of the constant. So if the constraint uses the letters '`o`' or '`m`', constant operands are not a problem.

- If the constraint permits a constant and a pseudo register used in an insn was not allocated to a hard register and is equivalent to a constant, the register will be replaced with the constant. If the predicate does not permit a constant and the insn is re-recognized for some reason, the compiler will crash. Thus the predicate must always recognize any objects allowed by the constraint.

If the operand's predicate can recognize registers, but the constraint does not permit them, it can make the compiler crash. When this operand happens to be a register, the reload pass will be stymied, because it does not know how to copy a register temporarily into memory.

If the predicate accepts a unary operator, the constraint applies to the operand. For example, the MIPS processor at ISA level 3 supports an instruction which adds two registers in `SImode` to produce a `DImode` result, but only if the registers are correctly sign extended. This predicate for the input operands accepts a `sign_extend` of an `SImode` register. Write the constraint to indicate the type of register that is required for the operand of the `sign_extend`.

13.8.2 Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative. Here is how it is done for fullword logical-or on the 68000:

```
(define_insn "iorsi3"
  [(set (match_operand:SI 0 "general_operand" "=m,d")
        (ior:SI (match_operand:SI 1 "general_operand" "%0,0")
                (match_operand:SI 2 "general_operand" "dKs,dmKs")))]
  ...)
```

The first alternative has '`m`' (memory) for operand 0, '`0`' for operand 1 (meaning it must match operand 0), and '`dKs`' for operand 2. The second alternative has '`d`' (data register) for operand 0, '`0`' for operand 1, and '`dmKs`' for operand 2. The '`=`' and '`%`' in the constraints apply to all the alternatives; their meaning is explained in the next section (see Section 13.8.3 [Class Preferences], page 196).

If all the operands fit any one alternative, the instruction is valid. Otherwise, for each alternative, the compiler counts how many instructions must be added to copy the operands so that that alternative applies. The alternative requiring the least copying is chosen. If two alternatives need the same amount of copying, the one that comes first is chosen. These choices can be altered with the '`?`' and '`!`' characters:

`?` Disparage slightly the alternative that the '`?`' appears in, as a choice when no alternative applies exactly. The compiler regards this alternative as one unit more costly for each '`?`' that appears in it.

`!` Disparage severely the alternative that the '`!`' appears in. This alternative can still be used if it fits without reloading, but if reloading is needed, some other alternative will be used.

When an insn pattern has multiple alternatives in its constraints, often the appearance of the assembler code is determined mostly by which alternative was matched. When this is so, the C code for writing the assembler code can use the variable `which_alternative`, which is the ordinal number of the alternative that was actually satisfied (0 for the first, 1 for the second alternative, etc.). See Section 13.6 [Output Statement], page 186.

13.8.3 Register Class Preferences

The operand constraints have another function: they enable the compiler to decide which kind of hardware register a pseudo register is best allocated to. The compiler examines the constraints that apply to the insns that use the pseudo register, looking for the machine-dependent letters such as '`d`' and '`a`' that specify classes of registers. The pseudo register is put in whichever class gets the most "votes". The constraint letters '`g`' and '`r`' also vote: they vote in favor of a general register. The machine description says which registers are considered general.

Of course, on some machines all registers are equivalent, and no register classes are defined. Then none of this complexity is relevant.

13.8.4 Constraint Modifier Characters

Here are constraint modifier characters.

'=' Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

'+' Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are inputs to the instruction and which are outputs from it. '=' identifies an output; '+' identifies an operand that is both input and output; all other operands are assumed to be input only.

If you specify '=' or '+' in a constraint, you put it in the first character of the constraint string.

'&' Means (in a particular alternative) that this operand is an *earlyclobber* operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.

'&' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires '&' while others do not. See, for example, the '`movdf`' insn of the 68000.

An input operand can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the inputs can be affected by the earlyclobber. See, for example, the '`mulsi3`' insn of the ARM.

'&' does not obviate the need to write '='.

'%' Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints. This is often used in patterns for addition instructions that really have only two operands: the result must go in one of the arguments. Here for example, is how the 68000 halfword-add instruction is defined:

```
(define_insn "addhi3"
  [(set (match_operand:HI 0 "general_operand" "=m,r")
     (plus:HI (match_operand:HI 1 "general_operand" "%0,0")
              (match_operand:HI 2 "general_operand" "di,g")))]
  ...)
```

GCC can only handle one commutative pair in an asm; if you use more, the compiler may fail. Note that you need not use the modifier if the two alternatives are strictly identical; this would only waste time in the reload pass. The modifier is not operational after register allocation, so the result of `define_peephole2` and `define_split`s performed after reload cannot rely on '%' to make the intended insn match.

'#' Says that all following characters, up to the next comma, are to be ignored as a constraint. They are significant only for choosing register preferences.

'*' Says that the following character should be ignored when choosing register preferences. '*' has no effect on the meaning of the constraint as a constraint, and no effect on reloading.

Here is an example: the 68000 has an instruction to sign-extend a halfword in a data register, and can also sign-extend a value by copying it into an address register. While either kind of register is acceptable, the constraints on an address-register destination are less strict, so it is best if register allocation makes an address register its goal. Therefore, '*' is used so that the 'd' constraint letter (for data register) is ignored when computing register preferences.

```
(define_insn "extendhisi2"
  [(set (match_operand:SI 0 "general_operand" "=*d,a")
        (sign_extend:SI
         (match_operand:HI 1 "general_operand" "0,g")))]
  ...)
```

13.8.5 Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in `asm` arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures. The most commonly used constraints are 'm' and 'r' (for memory and general-purpose registers respectively; see Section 13.8.1 [Simple Constraints], page 191), and 'I', usually the letter indicating the most common immediate-constant format.

For each machine architecture, the 'config/*machine*/*machine*.h' file defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for `asm` statements; therefore, some of the constraints are not particularly interesting for `asm`. The constraints are defined through these macros:

`REG_CLASS_FROM_LETTER`
: Register class constraints (usually lowercase).

`CONST_OK_FOR_LETTER_P`
: Immediate constant constraints, for non-floating point constants of word size or smaller precision (usually uppercase).

`CONST_DOUBLE_OK_FOR_LETTER_P`
: Immediate constant constraints, for all floating point constants and for constants of greater than word size precision (usually uppercase).

`EXTRA_CONSTRAINT`
: Special cases of registers or memory. This macro is not required, and is only defined for some machines.

Inspecting these macro definitions in the compiler source for your machine is the best way to be certain you have the right constraints. However, here is a summary of the machine-dependent constraints available on some particular machines.

ARM family—'`arm.h`'

`f`	Floating-point register
`w`	VFP floating-point register

`F` One of the floating-point constants 0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0 or 10.0

`G` Floating-point constant that would satisfy the constraint '`F`' if it were negated

`I` Integer that is valid as an immediate operand in a data processing instruction. That is, an integer in the range 0 to 255 rotated by a multiple of 2

`J` Integer in the range −4095 to 4095

`K` Integer that satisfies constraint '`I`' when inverted (ones complement)

`L` Integer that satisfies constraint '`I`' when negated (twos complement)

`M` Integer in the range 0 to 32

`Q` A memory reference where the exact address is in a single register ("`m`" is preferable for `asm` statements)

`R` An item in the constant pool

`S` A symbol in the text segment of the current file

`Uv` A memory reference suitable for VFP load/store insns (reg+constant offset)

`Uy` A memory reference suitable for iWMMXt load/store instructions.

`Uq` A memory reference suitable for the ARMv4 ldrsb instruction.

AVR family—'`avr.h`'

`l` Registers from r0 to r15

`a` Registers from r16 to r23

`d` Registers from r16 to r31

`w` Registers from r24 to r31. These registers can be used in '`adiw`' command

`e` Pointer register (r26–r31)

`b` Base pointer register (r28–r31)

`q` Stack pointer register (SPH:SPL)

`t` Temporary register r0

`x` Register pair X (r27:r26)

`y` Register pair Y (r29:r28)

`z` Register pair Z (r31:r30)

`I` Constant greater than −1, less than 64

`J` Constant greater than −64, less than 1

`K`	Constant integer 2
`L`	Constant integer 0
`M`	Constant that fits in 8 bits
`N`	Constant integer −1
`O`	Constant integer 8, 16, or 24
`P`	Constant integer 1
`G`	A floating point constant 0.0

CRX Architecture—'`crx.h`'

`b`	Registers from r0 to r14 (registers without stack pointer)
`l`	Register r16 (64-bit accumulator lo register)
`h`	Register r17 (64-bit accumulator hi register)
`k`	Register pair r16-r17. (64-bit accumulator lo-hi pair)
`I`	Constant that fits in 3 bits
`J`	Constant that fits in 4 bits
`K`	Constant that fits in 5 bits
`L`	Constant that is one of -1, 4, -4, 7, 8, 12, 16, 20, 32, 48
`G`	Floating point constant that is legal for store immediate

PowerPC and IBM RS6000—'`rs6000.h`'

`b`	Address base register
`f`	Floating point register
`v`	Vector register
`h`	'`MQ`', '`CTR`', or '`LINK`' register
`q`	'`MQ`' register
`c`	'`CTR`' register
`l`	'`LINK`' register
`x`	'`CR`' register (condition register) number 0
`y`	'`CR`' register (condition register)
`z`	'`FPMEM`' stack memory for FPR-GPR transfers
`I`	Signed 16-bit constant
`J`	Unsigned 16-bit constant shifted left 16 bits (use '`L`' instead for `SImode` constants)
`K`	Unsigned 16-bit constant
`L`	Signed 16-bit constant shifted left 16 bits

`M` Constant larger than 31

`N` Exact power of 2

`O` Zero

`P` Constant whose negation is a signed 16-bit constant

`G` Floating point constant that can be loaded into a register with one instruction per word

`Q` Memory operand that is an offset from a register ('`m`' is preferable for `asm` statements)

`R` AIX TOC entry

`S` Constant suitable as a 64-bit mask operand

`T` Constant suitable as a 32-bit mask operand

`U` System V Release 4 small data area reference

MorphoTech family—'`mt.h`'

`I` Constant for an arithmetic insn (16-bit signed integer).

`J` The constant 0.

`K` Constant for a logical insn (16-bit zero-extended integer).

`L` A constant that can be loaded with `lui` (i.e. the bottom 16 bits are zero).

`M` A constant that takes two words to load (i.e. not matched by `I`, `K`, or `L`).

`N` Negative 16-bit constants other than -65536.

`O` A 15-bit signed integer constant.

`P` A positive 16-bit constant.

Intel 386—'`i386.h`'

`q` '`a`', `b`, `c`, or `d` register for the i386. For x86-64 it is equivalent to '`r`' class (for 8-bit instructions that do not use upper halves).

`Q` '`a`', `b`, `c`, or `d` register (for 8-bit instructions, that do use upper halves).

`R` Legacy register—equivalent to `r` class in i386 mode. (for non-8-bit registers used together with 8-bit upper halves in a single instruction)

`A` Specifies the '`a`' or '`d`' registers. This is primarily useful for 64-bit integer values (when in 32-bit mode) intended to be returned with the '`d`' register holding the most significant bits and the '`a`' register holding the least significant bits.

`f` Floating point register

`t`	First (top of stack) floating point register
`u`	Second floating point register
`a`	'`a`' register
`b`	'`b`' register
`c`	'`c`' register
`C`	Specifies constant that can be easily constructed in SSE register without loading it from memory.
`d`	'`d`' register
`D`	'`di`' register
`S`	'`si`' register
`x`	'`xmm`' SSE register
`y`	MMX register
`I`	Constant in range 0 to 31 (for 32-bit shifts)
`J`	Constant in range 0 to 63 (for 64-bit shifts)
`K`	'`0xff`'
`L`	'`0xffff`'
`M`	0, 1, 2, or 3 (shifts for `lea` instruction)
`N`	Constant in range 0 to 255 (for `out` instruction)
`Z`	Constant in range 0 to `0xffffffff` or symbolic reference known to fit specified range. (for using immediates in zero extending 32-bit to 64-bit x86-64 instructions)
`e`	Constant in range −2147483648 to 2147483647 or symbolic reference known to fit specified range. (for using immediates in 64-bit x86-64 instructions)
`G`	Standard 80387 floating point constant

Intel IA-64—'`ia64.h`'

`a`	General register `r0` to `r3` for `addl` instruction
`b`	Branch register
`c`	Predicate register ('`c`' as in "conditional")
`d`	Application register residing in M-unit
`e`	Application register residing in I-unit
`f`	Floating-point register
`m`	Memory operand. Remember that '`m`' allows postincrement and postdecrement which require printing with '`%Pn`' on IA-64. Use '`S`' to disallow postincrement and postdecrement.

`G` Floating-point constant 0.0 or 1.0

`I` 14-bit signed integer constant

`J` 22-bit signed integer constant

`K` 8-bit signed integer constant for logical instructions

`L` 8-bit adjusted signed integer constant for compare pseudo-ops

`M` 6-bit unsigned integer constant for shift counts

`N` 9-bit signed integer constant for load and store postincrements

`O` The constant zero

`P` 0 or −1 for `dep` instruction

`Q` Non-volatile memory for floating-point loads and stores

`R` Integer constant in the range 1 to 4 for `shladd` instruction

`S` Memory operand except postincrement and postdecrement

FRV—'`frv.h`'

`a` Register in the class `ACC_REGS` (`acc0` to `acc7`).

`b` Register in the class `EVEN_ACC_REGS` (`acc0` to `acc7`).

`c` Register in the class `CC_REGS` (`fcc0` to `fcc3` and `icc0` to `icc3`).

`d` Register in the class `GPR_REGS` (`gr0` to `gr63`).

`e` Register in the class `EVEN_REGS` (`gr0` to `gr63`). Odd registers are excluded not in the class but through the use of a machine mode larger than 4 bytes.

`f` Register in the class `FPR_REGS` (`fr0` to `fr63`).

`h` Register in the class `FEVEN_REGS` (`fr0` to `fr63`). Odd registers are excluded not in the class but through the use of a machine mode larger than 4 bytes.

`l` Register in the class `LR_REG` (the `lr` register).

`q` Register in the class `QUAD_REGS` (`gr2` to `gr63`). Register numbers not divisible by 4 are excluded not in the class but through the use of a machine mode larger than 8 bytes.

`t` Register in the class `ICC_REGS` (`icc0` to `icc3`).

`u` Register in the class `FCC_REGS` (`fcc0` to `fcc3`).

`v` Register in the class `ICR_REGS` (`cc4` to `cc7`).

`w` Register in the class `FCR_REGS` (`cc0` to `cc3`).

`x` Register in the class `QUAD_FPR_REGS` (`fr0` to `fr63`). Register numbers not divisible by 4 are excluded not in the class but through the use of a machine mode larger than 8 bytes.

`z`	Register in the class `SPR_REGS` (`lcr` and `lr`).
`A`	Register in the class `QUAD_ACC_REGS` (`acc0` to `acc7`).
`B`	Register in the class `ACCG_REGS` (`accg0` to `accg7`).
`C`	Register in the class `CR_REGS` (`cc0` to `cc7`).
`G`	Floating point constant zero
`I`	6-bit signed integer constant
`J`	10-bit signed integer constant
`L`	16-bit signed integer constant
`M`	16-bit unsigned integer constant
`N`	12-bit signed integer constant that is negative—i.e. in the range of −2048 to −1
`O`	Constant zero
`P`	12-bit signed integer constant that is greater than zero—i.e. in the range of 1 to 2047.

Blackfin family—'`bfin.h`'

`a`	P register
`d`	D register
`z`	A call clobbered P register.
`D`	Even-numbered D register
`W`	Odd-numbered D register
`e`	Accumulator register.
`A`	Even-numbered accumulator register.
`B`	Odd-numbered accumulator register.
`b`	I register
`B`	B register
`f`	M register
`c`	Registers used for circular buffering, i.e. I, B, or L registers.
`C`	The CC register.
`x`	Any D, P, B, M, I or L register.
`y`	Additional registers typically used only in prologues and epilogues: RETS, RETN, RETI, RETX, RETE, ASTAT, SEQSTAT and USP.
`w`	Any register except accumulators or CC.
`Ksh`	Signed 16 bit integer (in the range -32768 to 32767)
`Kuh`	Unsigned 16 bit integer (in the range 0 to 65535)

`Ks7` Signed 7 bit integer (in the range -64 to 63)

`Ku7` Unsigned 7 bit integer (in the range 0 to 127)

`Ku5` Unsigned 5 bit integer (in the range 0 to 31)

`Ks4` Signed 4 bit integer (in the range -8 to 7)

`Ks3` Signed 3 bit integer (in the range -3 to 4)

`Ku3` Unsigned 3 bit integer (in the range 0 to 7)

`Pn` Constant *n*, where *n* is a single-digit constant in the range 0 to 4.

`M1` Constant 255.

`M2` Constant 65535.

`J` An integer constant with exactly a single bit set.

`L` An integer constant with all bits set except exactly one.

`H`

`Q` Any SYMBOL_REF.

M32C—'`m32c.c`'

Rsp
Rfb
Rsb '`$sp`', '`$fb`', '`$sb`'.

Rcr Any control register, when they're 16 bits wide (nothing if control registers are 24 bits wide)

Rcl Any control register, when they're 24 bits wide.

R0w
R1w
R2w
R3w $r0, $r1, $r2, $r3.

R02 $r0 or $r2, or $r2r0 for 32 bit values.

R13 $r1 or $r3, or $r3r1 for 32 bit values.

Rdi A register that can hold a 64 bit value.

Rhl $r0 or $r1 (registers with addressable high/low bytes)

R23 $r2 or $r3

Raa Address registers

Raw Address registers when they're 16 bits wide.

Ral Address registers when they're 24 bits wide.

Rqi Registers that can hold QI values.

Rad Registers that can be used with displacements ($a0, $a1, $sb).

Rsi Registers that can hold 32 bit values.

Rhi Registers that can hold 16 bit values.

Rhc Registers chat can hold 16 bit values, including all control registers.

Rra $r0 through R1, plus $a0 and $a1.

Rfl The flags register.

Rmm The memory-based pseudo-registers $mem0 through $mem15.

Rpi Registers that can hold pointers (16 bit registers for r8c, m16c; 24 bit registers for m32cm, m32c).

Rpa Matches multiple registers in a PARALLEL to form a larger register. Used to match function return values.

Is3 -8 ... 7

IS1 -128 ... 127

IS2 -32768 ... 32767

IU2 0 ... 65535

In4 -8 ... -1 or 1 ... 8

In5 -16 ... -1 or 1 ... 16

In4 -8 ... -1 or 1 ... 8

IM2 -65536 ... -1

Ilb An 8 bit value with exactly one bit set.

Ilw A 16 bit value with exactly one bit set.

Sd The common src/dest memory addressing modes.

Sa Memory addressed using $a0 or $a1.

Si Memory addressed with immediate addresses.

Ss Memory addressed using the stack pointer ($sp).

Sf Memory addressed using the frame base register ($fb).

Ss Memory addressed using the small base register ($sb).

S1 $r1h

MIPS—‘`mips.h`’

`d`	General-purpose integer register
`f`	Floating-point register (if available)
`h`	‘`Hi`’ register
`l`	‘`Lo`’ register
`x`	‘`Hi`’ or ‘`Lo`’ register
`y`	General-purpose integer register

`z` Floating-point status register

`I` Signed 16-bit constant (for arithmetic instructions)

`J` Zero

`K` Zero-extended 16-bit constant (for logic instructions)

`L` Constant with low 16 bits zero (can be loaded with `lui`)

`M` 32-bit constant which requires two instructions to load (a constant which is not '`I`', '`K`', or '`L`')

`N` Negative 16-bit constant

`O` Exact power of two

`P` Positive 16-bit constant

`G` Floating point zero

`Q` Memory reference that can be loaded with more than one instruction ('`m`' is preferable for `asm` statements)

`R` Memory reference that can be loaded with one instruction ('`m`' is preferable for `asm` statements)

`S` Memory reference in external OSF/rose PIC format ('`m`' is preferable for `asm` statements)

Motorola 680x0—'`m68k.h`'

`a` Address register

`d` Data register

`f` 68881 floating-point register, if available

`I` Integer in the range 1 to 8

`J` 16-bit signed number

`K` Signed number whose magnitude is greater than 0x80

`L` Integer in the range −8 to −1

`M` Signed number whose magnitude is greater than 0x100

`G` Floating point constant that is not a 68881 constant

Motorola 68HC11 & 68HC12 families—'`m68hc11.h`'

`a` Register 'a'

`b` Register 'b'

`d` Register 'd'

`q` An 8-bit register

`t` Temporary soft register _.tmp

`u` A soft register _.d1 to _.d31

`w`	Stack pointer register
`x`	Register ‘x’
`y`	Register ‘y’
`z`	Pseudo register ‘z’ (replaced by ‘x’ or ‘y’ at the end)
`A`	An address register: x, y or z
`B`	An address register: x or y
`D`	Register pair (x:d) to form a 32-bit value
`L`	Constants in the range −65536 to 65535
`M`	Constants whose 16-bit low part is zero
`N`	Constant integer 1 or −1
`O`	Constant integer 16
`P`	Constants in the range −8 to 2

SPARC—‘`sparc.h`’

`f`	Floating-point register on the SPARC-V8 architecture and lower floating-point register on the SPARC-V9 architecture.
`e`	Floating-point register. It is equivalent to ‘`f`’ on the SPARC-V8 architecture and contains both lower and upper floating-point registers on the SPARC-V9 architecture.
`c`	Floating-point condition code register.
`d`	Lower floating-point register. It is only valid on the SPARC-V9 architecture when the Visual Instruction Set is available.
`b`	Floating-point register. It is only valid on the SPARC-V9 architecture when the Visual Instruction Set is available.
`h`	64-bit global or out register for the SPARC-V8+ architecture.
`I`	Signed 13-bit constant
`J`	Zero
`K`	32-bit constant with the low 12 bits clear (a constant that can be loaded with the `sethi` instruction)
`L`	A constant in the range supported by `movcc` instructions
`M`	A constant in the range supported by `movrcc` instructions
`N`	Same as ‘`K`’, except that it verifies that bits that are not in the lower 32-bit range are all zero. Must be used instead of ‘`K`’ for modes wider than `SImode`
`O`	The constant 4096
`G`	Floating-point zero

`H` Signed 13-bit constant, sign-extended to 32 or 64 bits

`Q` Floating-point constant whose integral representation can be moved into an integer register using a single sethi instruction

`R` Floating-point constant whose integral representation can be moved into an integer register using a single mov instruction

`S` Floating-point constant whose integral representation can be moved into an integer register using a high/lo_sum instruction sequence

`T` Memory address aligned to an 8-byte boundary

`U` Even register

`W` Memory address for '`e`' constraint registers

`Y` Vector zero

TMS320C3x/C4x—'`c4x.h`'

`a` Auxiliary (address) register (ar0-ar7)

`b` Stack pointer register (sp)

`c` Standard (32-bit) precision integer register

`f` Extended (40-bit) precision register (r0-r11)

`k` Block count register (bk)

`q` Extended (40-bit) precision low register (r0-r7)

`t` Extended (40-bit) precision register (r0-r1)

`u` Extended (40-bit) precision register (r2-r3)

`v` Repeat count register (rc)

`x` Index register (ir0-ir1)

`y` Status (condition code) register (st)

`z` Data page register (dp)

`G` Floating-point zero

`H` Immediate 16-bit floating-point constant

`I` Signed 16-bit constant

`J` Signed 8-bit constant

`K` Signed 5-bit constant

`L` Unsigned 16-bit constant

`M` Unsigned 8-bit constant

`N` Ones complement of unsigned 16-bit constant

`O` High 16-bit constant (32-bit constant with 16 LSBs zero)

`Q` Indirect memory reference with signed 8-bit or index register displacement

`R` Indirect memory reference with unsigned 5-bit displacement

`S` Indirect memory reference with 1 bit or index register displacement

`T` Direct memory reference

`U` Symbolic address

S/390 and zSeries—'`s390.h`'

`a` Address register (general purpose register except r0)

`c` Condition code register

`d` Data register (arbitrary general purpose register)

`f` Floating-point register

`I` Unsigned 8-bit constant (0–255)

`J` Unsigned 12-bit constant (0–4095)

`K` Signed 16-bit constant (−32768–32767)

`L` Value appropriate as displacement.

`(0..4095)`
: for short displacement

`(-524288..524287)`
: for long displacement

`M` Constant integer with a value of 0x7fffffff.

`N` Multiple letter constraint followed by 4 parameter letters.

`0..9:` number of the part counting from most to least significant

`H,Q:` mode of the part

`D,S,H:` mode of the containing operand

`0,F:` value of the other parts (F—all bits set)

The constraint matches if the specified part of a constant has a value different from it's other parts.

`Q` Memory reference without index register and with short displacement.

`R` Memory reference with index register and short displacement.

`S` Memory reference without index register but with long displacement.

`T` Memory reference with index register and long displacement.

`U` Pointer with short displacement.

`W` Pointer with long displacement.

`Y` Shift count operand.

Xstormy16—'`stormy16.h`'

`a` Register r0.

`b` Register r1.

`c` Register r2.

`d` Register r8.

`e` Registers r0 through r7.

`t` Registers r0 and r1.

`y` The carry register.

`z` Registers r8 and r9.

`I` A constant between 0 and 3 inclusive.

`J` A constant that has exactly one bit set.

`K` A constant that has exactly one bit clear.

`L` A constant between 0 and 255 inclusive.

`M` A constant between −255 and 0 inclusive.

`N` A constant between −3 and 0 inclusive.

`O` A constant between 1 and 4 inclusive.

`P` A constant between −4 and −1 inclusive.

`Q` A memory reference that is a stack push.

`R` A memory reference that is a stack pop.

`S` A memory reference that refers to a constant address of known value.

`T` The register indicated by Rx (not implemented yet).

`U` A constant that is not between 2 and 15 inclusive.

`Z` The constant 0.

Xtensa—'`xtensa.h`'

`a` General-purpose 32-bit register

`b` One-bit boolean register

`A` MAC16 40-bit accumulator register

`I` Signed 12-bit integer constant, for use in MOVI instructions

`J` Signed 8-bit integer constant, for use in ADDI instructions

`K` Integer constant valid for BccI instructions

`L` Unsigned constant valid for BccUI instructions

13.9 Standard Pattern Names For Generation

Here is a table of the instruction names that are meaningful in the RTL generation pass of the compiler. Giving one of these names to an instruction pattern tells the RTL generation pass that it can use the pattern to accomplish a certain task.

'`movm`' Here *m* stands for a two-letter machine mode name, in lowercase. This instruction pattern moves data with that machine mode from operand 1 to operand 0. For example, '`movsi`' moves full-word data.

If operand 0 is a `subreg` with mode *m* of a register whose own mode is wider than *m*, the effect of this instruction is to store the specified value in the part of the register that corresponds to mode *m*. Bits outside of *m*, but which are within the same target word as the `subreg` are undefined. Bits which are outside the target word are left unchanged.

This class of patterns is special in several ways. First of all, each of these names up to and including full word size *must* be defined, because there is no other way to copy a datum from one place to another. If there are patterns accepting operands in larger modes, '`movm`' must be defined for integer modes of those sizes.

Second, these patterns are not used solely in the RTL generation pass. Even the reload pass can generate move insns to copy values from stack slots into temporary registers. When it does so, one of the operands is a hard register and the other is an operand that can need to be reloaded into a register.

Therefore, when given such a pair of operands, the pattern must generate RTL which needs no reloading and needs no temporary registers—no registers other than the operands. For example, if you support the pattern with a `define_expand`, then in such a case the `define_expand` mustn't call `force_reg` or any other such function which might generate new pseudo registers.

This requirement exists even for subword modes on a RISC machine where fetching those modes from memory normally requires several insns and some temporary registers.

During reload a memory reference with an invalid address may be passed as an operand. Such an address will be replaced with a valid address later in the reload pass. In this case, nothing may be done with the address except to use it as it stands. If it is copied, it will not be replaced with a valid address. No attempt should be made to make such an address into a valid address and no routine (such as `change_address`) that will do so may be called. Note that `general_operand` will fail when applied to such an address.

The global variable `reload_in_progress` (which must be explicitly declared if required) can be used to determine whether such special handling is required.

The variety of operands that have reloads depends on the rest of the machine description, but typically on a RISC machine these can only be pseudo registers that did not get hard registers, while on other machines explicit memory references will get optional reloads.

If a scratch register is required to move an object to or from memory, it can be allocated using `gen_reg_rtx` prior to life analysis.

If there are cases which need scratch registers during or after reload, you must define `SECONDARY_INPUT_RELOAD_CLASS` and/or `SECONDARY_OUTPUT_RELOAD_CLASS` to detect them, and provide patterns '`reload_in`*m*' or '`reload_out`*m*' to handle them. See Section 14.8 [Register Classes], page 296.

The global variable `no_new_pseudos` can be used to determine if it is unsafe to create new pseudo registers. If this variable is nonzero, then it is unsafe to call `gen_reg_rtx` to allocate a new pseudo.

The constraints on a '`mov`*m*' must permit moving any hard register to any other hard register provided that `HARD_REGNO_MODE_OK` permits mode *m* in both registers and `REGISTER_MOVE_COST` applied to their classes returns a value of 2.

It is obligatory to support floating point '`mov`*m*' instructions into and out of any registers that can hold fixed point values, because unions and structures (which have modes `SImode` or `DImode`) can be in those registers and they may have floating point members.

There may also be a need to support fixed point '`mov`*m*' instructions in and out of floating point registers. Unfortunately, I have forgotten why this was so, and I don't know whether it is still true. If `HARD_REGNO_MODE_OK` rejects fixed point values in floating point registers, then the constraints of the fixed point '`mov`*m*' instructions must be designed to avoid ever trying to reload into a floating point register.

'`reload_in`*m*'
'`reload_out`*m*'
Like '`mov`*m*', but used when a scratch register is required to move between operand 0 and operand 1. Operand 2 describes the scratch register. See the discussion of the `SECONDARY_RELOAD_CLASS` macro in see Section 14.8 [Register Classes], page 296.

There are special restrictions on the form of the `match_operand`s used in these patterns. First, only the predicate for the reload operand is examined, i.e., `reload_in` examines operand 1, but not the predicates for operand 0 or 2. Second, there may be only one alternative in the constraints. Third, only a single register class letter may be used for the constraint; subsequent constraint letters are ignored. As a special exception, an empty constraint string matches the `ALL_REGS` register class. This may relieve ports of the burden of defining an `ALL_REGS` constraint letter just for these patterns.

'`movstrict`*m*'
Like '`mov`*m*' except that if operand 0 is a `subreg` with mode *m* of a register whose natural mode is wider, the '`movstrict`*m*' instruction is guaranteed not to alter any of the register except the part which belongs to mode *m*.

'`movmisalign`*m*'
This variant of a move pattern is designed to load or store a value from a memory address that is not naturally aligned for its mode. For a store, the memory will be in operand 0; for a load, the memory will be in operand 1. The other operand is guaranteed not to be a memory, so that it's easy to tell whether this is a load or store.

This pattern is used by the autovectorizer, and when expanding a `MISALIGNED_INDIRECT_REF` expression.

'`load_multiple`'
Load several consecutive memory locations into consecutive registers. Operand 0 is the first of the consecutive registers, operand 1 is the first memory location, and operand 2 is a constant: the number of consecutive registers.

Define this only if the target machine really has such an instruction; do not define this if the most efficient way of loading consecutive registers from memory is to do them one at a time.

On some machines, there are restrictions as to which consecutive registers can be stored into memory, such as particular starting or ending register numbers or only a range of valid counts. For those machines, use a `define_expand` (see Section 13.15 [Expander Definitions], page 237) and make the pattern fail if the restrictions are not met.

Write the generated insn as a `parallel` with elements being a `set` of one register from the appropriate memory location (you may also need `use` or `clobber` elements). Use a `match_parallel` (see Section 13.4 [RTL Template], page 181) to recognize the insn. See '`rs6000.md`' for examples of the use of this insn pattern.

'`store_multiple`'
Similar to '`load_multiple`', but store several consecutive registers into consecutive memory locations. Operand 0 is the first of the consecutive memory locations, operand 1 is the first register, and operand 2 is a constant: the number of consecutive registers.

'`vec_set`*m*'
Set given field in the vector value. Operand 0 is the vector to modify, operand 1 is new value of field and operand 2 specify the field index.

'`vec_extract`*m*'
Extract given field from the vector value. Operand 1 is the vector, operand 2 specify field index and operand 0 place to store value into.

'`vec_init`*m*'
Initialize the vector to given values. Operand 0 is the vector to initialize and operand 1 is parallel containing values for individual fields.

'`push`*m*`1`' Output a push instruction. Operand 0 is value to push. Used only when `PUSH_ROUNDING` is defined. For historical reason, this pattern may be missing and in such case an `mov` expander is used instead, with a `MEM` expression forming the push operation. The `mov` expander method is deprecated.

'`add`*m*`3`' Add operand 2 and operand 1, storing the result in operand 0. All operands must have mode *m*. This can be used even on two-address machines, by means of constraints requiring operands 1 and 0 to be the same location.

‘sub*m*3’, ‘mul*m*3’
‘div*m*3’, ‘udiv*m*3’
‘mod*m*3’, ‘umod*m*3’
‘umin*m*3’, ‘umax*m*3’
‘and*m*3’, ‘ior*m*3’, ‘xor*m*3’
Similar, for other arithmetic operations.

‘smin*m*3’, ‘smax*m*3’
Signed minimum and maximum operations. When used with floating point, if both operands are zeros, or if either operand is `NaN`, then it is unspecified which of the two operands is returned as the result.

‘reduc_smin_*m*’, ‘reduc_smax_*m*’
Find the signed minimum/maximum of the elements of a vector. The vector is operand 1, and the scalar result is stored in the least significant bits of operand 0 (also a vector). The output and input vector should have the same modes.

‘reduc_umin_*m*’, ‘reduc_umax_*m*’
Find the unsigned minimum/maximum of the elements of a vector. The vector is operand 1, and the scalar result is stored in the least significant bits of operand 0 (also a vector). The output and input vector should have the same modes.

‘reduc_splus_*m*’
Compute the sum of the signed elements of a vector. The vector is operand 1, and the scalar result is stored in the least significant bits of operand 0 (also a vector). The output and input vector should have the same modes.

‘reduc_uplus_*m*’
Compute the sum of the unsigned elements of a vector. The vector is operand 1, and the scalar result is stored in the least significant bits of operand 0 (also a vector). The output and input vector should have the same modes.

‘vec_shl_*m*’, ‘vec_shr_*m*’
Whole vector left/right shift in bits. Operand 1 is a vector to be shifted. Operand 2 is an integer shift amount in bits. Operand 0 is where the resulting shifted vector is stored. The output and input vectors should have the same modes.

‘mulhisi3’
Multiply operands 1 and 2, which have mode `HImode`, and store a `SImode` product in operand 0.

‘mulqihi3’, ‘mulsidi3’
Similar widening-multiplication instructions of other widths.

‘umulqihi3’, ‘umulhisi3’, ‘umulsidi3’
Similar widening-multiplication instructions that do unsigned multiplication.

‘smul*m*3_highpart’
Perform a signed multiplication of operands 1 and 2, which have mode *m*, and store the most significant half of the product in operand 0. The least significant half of the product is discarded.

‘umul*m*3_highpart’
: Similar, but the multiplication is unsigned.

‘divmod*m*4’
: Signed division that produces both a quotient and a remainder. Operand 1 is divided by operand 2 to produce a quotient stored in operand 0 and a remainder stored in operand 3.

  For machines with an instruction that produces both a quotient and a remainder, provide a pattern for ‘divmod*m*4’ but do not provide patterns for ‘div*m*3’ and ‘mod*m*3’. This allows optimization in the relatively common case when both the quotient and remainder are computed.

  If an instruction that just produces a quotient or just a remainder exists and is more efficient than the instruction that produces both, write the output routine of ‘divmod*m*4’ to call `find_reg_note` and look for a `REG_UNUSED` note on the quotient or remainder and generate the appropriate instruction.

‘udivmod*m*4’
: Similar, but does unsigned division.

‘ashl*m*3’
: Arithmetic-shift operand 1 left by a number of bits specified by operand 2, and store the result in operand 0. Here *m* is the mode of operand 0 and operand 1; operand 2's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction. The meaning of out-of-range shift counts can optionally be specified by `TARGET_SHIFT_TRUNCATION_MASK`. See [TARGET_SHIFT_TRUNCATION_MASK], page 393.

‘ashr*m*3’, ‘lshr*m*3’, ‘rotl*m*3’, ‘rotr*m*3’
: Other shift and rotate instructions, analogous to the ashl*m*3 instructions.

‘neg*m*2’
: Negate operand 1 and store the result in operand 0.

‘abs*m*2’
: Store the absolute value of operand 1 into operand 0.

‘sqrt*m*2’
: Store the square root of operand 1 into operand 0.

  The `sqrt` built-in function of C always uses the mode which corresponds to the C data type `double` and the `sqrtf` built-in function uses the mode which corresponds to the C data type `float`.

‘cos*m*2’
: Store the cosine of operand 1 into operand 0.

  The `cos` built-in function of C always uses the mode which corresponds to the C data type `double` and the `cosf` built-in function uses the mode which corresponds to the C data type `float`.

‘sin*m*2’
: Store the sine of operand 1 into operand 0.

  The `sin` built-in function of C always uses the mode which corresponds to the C data type `double` and the `sinf` built-in function uses the mode which corresponds to the C data type `float`.

‘exp*m*2’
: Store the exponential of operand 1 into operand 0.

  The `exp` built-in function of C always uses the mode which corresponds to the C data type `double` and the `expf` built-in function uses the mode which corresponds to the C data type `float`.

‘`log`*m*`2`’ Store the natural logarithm of operand 1 into operand 0.

The `log` built-in function of C always uses the mode which corresponds to the C data type `double` and the `logf` built-in function uses the mode which corresponds to the C data type `float`.

‘`pow`*m*`3`’ Store the value of operand 1 raised to the exponent operand 2 into operand 0.

The `pow` built-in function of C always uses the mode which corresponds to the C data type `double` and the `powf` built-in function uses the mode which corresponds to the C data type `float`.

‘`atan2`*m*`3`’ Store the arc tangent (inverse tangent) of operand 1 divided by operand 2 into operand 0, using the signs of both arguments to determine the quadrant of the result.

The `atan2` built-in function of C always uses the mode which corresponds to the C data type `double` and the `atan2f` built-in function uses the mode which corresponds to the C data type `float`.

‘`floor`*m*`2`’ Store the largest integral value not greater than argument.

The `floor` built-in function of C always uses the mode which corresponds to the C data type `double` and the `floorf` built-in function uses the mode which corresponds to the C data type `float`.

‘`btrunc`*m*`2`’
Store the argument rounded to integer towards zero.

The `trunc` built-in function of C always uses the mode which corresponds to the C data type `double` and the `truncf` built-in function uses the mode which corresponds to the C data type `float`.

‘`round`*m*`2`’ Store the argument rounded to integer away from zero.

The `round` built-in function of C always uses the mode which corresponds to the C data type `double` and the `roundf` built-in function uses the mode which corresponds to the C data type `float`.

‘`ceil`*m*`2`’ Store the argument rounded to integer away from zero.

The `ceil` built-in function of C always uses the mode which corresponds to the C data type `double` and the `ceilf` built-in function uses the mode which corresponds to the C data type `float`.

‘`nearbyint`*m*`2`’
Store the argument rounded according to the default rounding mode

The `nearbyint` built-in function of C always uses the mode which corresponds to the C data type `double` and the `nearbyintf` built-in function uses the mode which corresponds to the C data type `float`.

‘`rint`*m*`2`’ Store the argument rounded according to the default rounding mode and raise the inexact exception when the result differs in value from the argument

The `rint` built-in function of C always uses the mode which corresponds to the C data type `double` and the `rintf` built-in function uses the mode which corresponds to the C data type `float`.

‘`copysignm3`’
: Store a value with the magnitude of operand 1 and the sign of operand 2 into operand 0.

  The `copysign` built-in function of C always uses the mode which corresponds to the C data type `double` and the `copysignf` built-in function uses the mode which corresponds to the C data type `float`.

‘`ffsm2`’
: Store into operand 0 one plus the index of the least significant 1-bit of operand 1. If operand 1 is zero, store zero. *m* is the mode of operand 0; operand 1's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

  The `ffs` built-in function of C always uses the mode which corresponds to the C data type `int`.

‘`clzm2`’
: Store into operand 0 the number of leading 0-bits in *x*, starting at the most significant bit position. If *x* is 0, the result is undefined. *m* is the mode of operand 0; operand 1's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

‘`ctzm2`’
: Store into operand 0 the number of trailing 0-bits in *x*, starting at the least significant bit position. If *x* is 0, the result is undefined. *m* is the mode of operand 0; operand 1's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

‘`popcountm2`’
: Store into operand 0 the number of 1-bits in *x*. *m* is the mode of operand 0; operand 1's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

‘`paritym2`’
: Store into operand 0 the parity of *x*, i.e. the number of 1-bits in *x* modulo 2. *m* is the mode of operand 0; operand 1's mode is specified by the instruction pattern, and the compiler will convert the operand to that mode before generating the instruction.

‘`one_cmplm2`’
: Store the bitwise-complement of operand 1 into operand 0.

‘`cmpm`’
: Compare operand 0 and operand 1, and set the condition codes. The RTL pattern should look like this:

```
(set (cc0) (compare (match_operand:m 0 ...)
                    (match_operand:m 1 ...)))
```

‘`tstm`’
: Compare operand 0 against zero, and set the condition codes. The RTL pattern should look like this:

```
(set (cc0) (match_operand:m 0 ...))
```

  ‘`tstm`’ patterns should not be defined for machines that do not use `(cc0)`. Doing so would confuse the optimizer since it would no longer be clear which `set` operations were comparisons. The ‘`cmpm`’ patterns should be used instead.

‘`movmemm`’
: Block move instruction. The destination and source blocks of memory are the first two operands, and both are `mem:BLK`s with an address in mode `Pmode`.

The number of bytes to move is the third operand, in mode *m*. Usually, you specify `word_mode` for *m*. However, if you can generate better code knowing the range of valid lengths is smaller than those representable in a full word, you should provide a pattern with a mode corresponding to the range of values you can handle efficiently (e.g., `QImode` for values in the range 0–127; note we avoid numbers that appear negative) and also a pattern with `word_mode`.

The fourth operand is the known shared alignment of the source and destination, in the form of a `const_int` rtx. Thus, if the compiler knows that both source and destination are word-aligned, it may provide the value 4 for this operand.

Descriptions of multiple `movmem`*m* patterns can only be beneficial if the patterns for smaller modes have fewer restrictions on their first, second and fourth operands. Note that the mode *m* in `movmem`*m* does not impose any restriction on the mode of individually moved data units in the block.

These patterns need not give special consideration to the possibility that the source and destination strings might overlap.

'`movstr`' String copy instruction, with `stpcpy` semantics. Operand 0 is an output operand in mode `Pmode`. The addresses of the destination and source strings are operands 1 and 2, and both are `mem:BLK`s with addresses in mode `Pmode`. The execution of the expansion of this pattern should store in operand 0 the address in which the `NUL` terminator was stored in the destination string.

'`setmem`*m*' Block set instruction. The destination string is the first operand, given as a `mem:BLK` whose address is in mode `Pmode`. The number of bytes to set is the second operand, in mode *m*. The value to initialize the memory with is the third operand. Targets that only support the clearing of memory should reject any value that is not the constant 0. See '`movmem`*m*' for a discussion of the choice of mode.

The fourth operand is the known alignment of the destination, in the form of a `const_int` rtx. Thus, if the compiler knows that the destination is word-aligned, it may provide the value 4 for this operand.

The use for multiple `setmem`*m* is as for `movmem`*m*.

'`cmpstrn`*m*'
String compare instruction, with five operands. Operand 0 is the output; it has mode *m*. The remaining four operands are like the operands of '`movmem`*m*'. The two memory blocks specified are compared byte by byte in lexicographic order starting at the beginning of each string. The instruction is not allowed to prefetch more than one byte at a time since either string may end in the first byte and reading past that may access an invalid page or segment and cause a fault. The effect of the instruction is to store a value in operand 0 whose sign indicates the result of the comparison.

'`cmpstr`*m*' String compare instruction, without known maximum length. Operand 0 is the output; it has mode *m*. The second and third operand are the blocks of memory to be compared; both are `mem:BLK` with an address in mode `Pmode`.

The fourth operand is the known shared alignment of the source and destination, in the form of a `const_int` rtx. Thus, if the compiler knows that both source and destination are word-aligned, it may provide the value 4 for this operand.

The two memory blocks specified are compared byte by byte in lexicographic order starting at the beginning of each string. The instruction is not allowed to prefetch more than one byte at a time since either string may end in the first byte and reading past that may access an invalid page or segment and cause a fault. The effect of the instruction is to store a value in operand 0 whose sign indicates the result of the comparison.

'`cmpmem`*m*' Block compare instruction, with five operands like the operands of '`cmpstr`*m*'. The two memory blocks specified are compared byte by byte in lexicographic order starting at the beginning of each block. Unlike '`cmpstr`*m*' the instruction can prefetch any bytes in the two memory blocks. The effect of the instruction is to store a value in operand 0 whose sign indicates the result of the comparison.

'`strlen`*m*' Compute the length of a string, with three operands. Operand 0 is the result (of mode *m*), operand 1 is a `mem` referring to the first character of the string, operand 2 is the character to search for (normally zero), and operand 3 is a constant describing the known alignment of the beginning of the string.

'`float`*mn*`2`'
Convert signed integer operand 1 (valid for fixed point mode *m*) to floating point mode *n* and store in operand 0 (which has mode *n*).

'`floatuns`*mn*`2`'
Convert unsigned integer operand 1 (valid for fixed point mode *m*) to floating point mode *n* and store in operand 0 (which has mode *n*).

'`fix`*mn*`2`' Convert operand 1 (valid for floating point mode *m*) to fixed point mode *n* as a signed number and store in operand 0 (which has mode *n*). This instruction's result is defined only when the value of operand 1 is an integer.

If the machine description defines this pattern, it also needs to define the `ftrunc` pattern.

'`fixuns`*mn*`2`'
Convert operand 1 (valid for floating point mode *m*) to fixed point mode *n* as an unsigned number and store in operand 0 (which has mode *n*). This instruction's result is defined only when the value of operand 1 is an integer.

'`ftrunc`*m*`2`'
Convert operand 1 (valid for floating point mode *m*) to an integer value, still represented in floating point mode *m*, and store it in operand 0 (valid for floating point mode *m*).

'`fix_trunc`*mn*`2`'
Like '`fix`*mn*`2`' but works for any floating point value of mode *m* by converting the value to an integer.

'`fixuns_trunc`*mn*`2`'
Like '`fixuns`*mn*`2`' but works for any floating point value of mode *m* by converting the value to an integer.

'`trunc`*mn*`2`'
: Truncate operand 1 (valid for mode *m*) to mode *n* and store in operand 0 (which has mode *n*). Both modes must be fixed point or both floating point.

'`extend`*mn*`2`'
: Sign-extend operand 1 (valid for mode *m*) to mode *n* and store in operand 0 (which has mode *n*). Both modes must be fixed point or both floating point.

'`zero_extend`*mn*`2`'
: Zero-extend operand 1 (valid for mode *m*) to mode *n* and store in operand 0 (which has mode *n*). Both modes must be fixed point.

'`extv`'
: Extract a bit-field from operand 1 (a register or memory operand), where operand 2 specifies the width in bits and operand 3 the starting bit, and store it in operand 0. Operand 0 must have mode `word_mode`. Operand 1 may have mode `byte_mode` or `word_mode`; often `word_mode` is allowed only for registers. Operands 2 and 3 must be valid for `word_mode`.

  The RTL generation pass generates this instruction only with constants for operands 2 and 3 and the constant is never zero for operand 2.

  The bit-field value is sign-extended to a full word integer before it is stored in operand 0.

'`extzv`'
: Like '`extv`' except that the bit-field value is zero-extended.

'`insv`'
: Store operand 3 (which must be valid for `word_mode`) into a bit-field in operand 0, where operand 1 specifies the width in bits and operand 2 the starting bit. Operand 0 may have mode `byte_mode` or `word_mode`; often `word_mode` is allowed only for registers. Operands 1 and 2 must be valid for `word_mode`.

  The RTL generation pass generates this instruction only with constants for operands 1 and 2 and the constant is never zero for operand 1.

'`mov`*mode*`cc`'
: Conditionally move operand 2 or operand 3 into operand 0 according to the comparison in operand 1. If the comparison is true, operand 2 is moved into operand 0, otherwise operand 3 is moved.

  The mode of the operands being compared need not be the same as the operands being moved. Some machines, sparc64 for example, have instructions that conditionally move an integer value based on the floating point condition codes and vice versa.

  If the machine does not have conditional move instructions, do not define these patterns.

'`add`*mode*`cc`'
: Similar to '`mov`*mode*`cc`' but for conditional addition. Conditionally move operand 2 or (operands 2 `+` operand 3) into operand 0 according to the comparison in operand 1. If the comparison is true, operand 2 is moved into operand 0, otherwise (operand 2 `+` operand 3) is moved.

'`s`*cond*'
: Store zero or nonzero in the operand according to the condition codes. Value stored is nonzero iff the condition *cond* is true. *cond* is the name of a comparison operation expression code, such as `eq`, `lt` or `leu`.

You specify the mode that the operand must have when you write the `match_operand` expression. The compiler automatically sees which mode you have used and supplies an operand of that mode.

The value stored for a true condition must have 1 as its low bit, or else must be negative. Otherwise the instruction is not suitable and you should omit it from the machine description. You describe to the compiler exactly which value is stored by defining the macro `STORE_FLAG_VALUE` (see Section 14.27 [Misc], page 390). If a description cannot be found that can be used for all the '`s`*cond*' patterns, you should omit those operations from the machine description.

These operations may fail, but should do so only in relatively uncommon cases; if they would fail for common cases involving integer comparisons, it is best to omit these patterns.

If these operations are omitted, the compiler will usually generate code that copies the constant one to the target and branches around an assignment of zero to the target. If this code is more efficient than the potential instructions used for the '`s`*cond*' pattern followed by those required to convert the result into a 1 or a zero in `SImode`, you should omit the '`s`*cond*' operations from the machine description.

'`b`*cond*' Conditional branch instruction. Operand 0 is a `label_ref` that refers to the label to jump to. Jump if the condition codes meet condition *cond*.

Some machines do not follow the model assumed here where a comparison instruction is followed by a conditional branch instruction. In that case, the '`cmp`*m*' (and '`tst`*m*') patterns should simply store the operands away and generate all the required insns in a `define_expand` (see Section 13.15 [Expander Definitions], page 237) for the conditional branch operations. All calls to expand '`b`*cond*' patterns are immediately preceded by calls to expand either a '`cmp`*m*' pattern or a '`tst`*m*' pattern.

Machines that use a pseudo register for the condition code value, or where the mode used for the comparison depends on the condition being tested, should also use the above mechanism. See Section 13.12 [Jump Patterns], page 233.

The above discussion also applies to the '`mov`*mode*`cc`' and '`s`*cond*' patterns.

'`cbranch`*mode*`4`'
Conditional branch instruction combined with a compare instruction. Operand 0 is a comparison operator. Operand 1 and operand 2 are the first and second operands of the comparison, respectively. Operand 3 is a `label_ref` that refers to the label to jump to.

'`jump`' A jump inside a function; an unconditional branch. Operand 0 is the `label_ref` of the label to jump to. This pattern name is mandatory on all machines.

'`call`' Subroutine call instruction returning no value. Operand 0 is the function to call; operand 1 is the number of bytes of arguments pushed as a `const_int`; operand 2 is the number of registers used as operands.

On most machines, operand 2 is not actually stored into the RTL pattern. It is supplied for the sake of some RISC machines which need to put this information into the assembler code; they can put it in the RTL instead of operand 1.

Operand 0 should be a `mem` RTX whose address is the address of the function. Note, however, that this address can be a `symbol_ref` expression even if it would not be a legitimate memory address on the target machine. If it is also not a valid argument for a call instruction, the pattern for this operation should be a `define_expand` (see Section 13.15 [Expander Definitions], page 237) that places the address into a register and uses that register in the call instruction.

'`call_value`'
Subroutine call instruction returning a value. Operand 0 is the hard register in which the value is returned. There are three more operands, the same as the three operands of the '`call`' instruction (but with numbers increased by one).

Subroutines that return `BLKmode` objects use the '`call`' insn.

'`call_pop`', '`call_value_pop`'
Similar to '`call`' and '`call_value`', except used if defined and if `RETURN_POPS_ARGS` is nonzero. They should emit a `parallel` that contains both the function call and a `set` to indicate the adjustment made to the frame pointer.

For machines where `RETURN_POPS_ARGS` can be nonzero, the use of these patterns increases the number of functions for which the frame pointer can be eliminated, if desired.

'`untyped_call`'
Subroutine call instruction returning a value of any type. Operand 0 is the function to call; operand 1 is a memory location where the result of calling the function is to be stored; operand 2 is a `parallel` expression where each element is a `set` expression that indicates the saving of a function return value into the result block.

This instruction pattern should be defined to support `__builtin_apply` on machines where special instructions are needed to call a subroutine with arbitrary arguments or to save the value returned. This instruction pattern is required on machines that have multiple registers that can hold a return value (i.e. `FUNCTION_VALUE_REGNO_P` is true for more than one register).

'`return`' Subroutine return instruction. This instruction pattern name should be defined only if a single instruction can do all the work of returning from a function.

Like the '`mov`*m*' patterns, this pattern is also used after the RTL generation phase. In this case it is to support machines where multiple instructions are usually needed to return from a function, but some class of functions only requires one instruction to implement a return. Normally, the applicable functions are those which do not need to save any registers or allocate stack space.

For such machines, the condition specified in this pattern should only be true when `reload_completed` is nonzero and the function's epilogue would only be a single instruction. For machines with register windows, the routine `leaf_function_p` may be used to determine if a register window push is required.

Machines that have conditional return instructions should define patterns such as

```
(define_insn ""
  [(set (pc)
```

```
      (if_then_else (match_operator
                      0 "comparison_operator"
                      [(cc0) (const_int 0)])
                    (return)
                    (pc)))]
  "condition"
  "...")
```

where *condition* would normally be the same condition specified on the named '`return`' pattern.

'`untyped_return`'
: Untyped subroutine return instruction. This instruction pattern should be defined to support `__builtin_return` on machines where special instructions are needed to return a value of any type.

  Operand 0 is a memory location where the result of calling a function with `__builtin_apply` is stored; operand 1 is a `parallel` expression where each element is a `set` expression that indicates the restoring of a function return value from the result block.

'`nop`'
: No-op instruction. This instruction pattern name should always be defined to output a no-op in assembler code. `(const_int 0)` will do as an RTL pattern.

'`indirect_jump`'
: An instruction to jump to an address which is operand zero. This pattern name is mandatory on all machines.

'`casesi`'
: Instruction to jump through a dispatch table, including bounds checking. This instruction takes five operands:

  1. The index to dispatch on, which has mode `SImode`.
  2. The lower bound for indices in the table, an integer constant.
  3. The total range of indices in the table—the largest index minus the smallest one (both inclusive).
  4. A label that precedes the table itself.
  5. A label to jump to if the index has a value outside the bounds.

  The table is a `addr_vec` or `addr_diff_vec` inside of a `jump_insn`. The number of elements in the table is one plus the difference between the upper bound and the lower bound.

'`tablejump`'
: Instruction to jump to a variable address. This is a low-level capability which can be used to implement a dispatch table when there is no '`casesi`' pattern.

  This pattern requires two operands: the address or offset, and a label which should immediately precede the jump table. If the macro `CASE_VECTOR_PC_RELATIVE` evaluates to a nonzero value then the first operand is an offset which counts from the address of the table; otherwise, it is an absolute address to jump to. In either case, the first operand has mode `Pmode`.

  The '`tablejump`' insn is always the last insn before the jump table it uses. Its assembler code normally has no need to use the second operand, but you should

incorporate it in the RTL pattern so that the jump optimizer will not delete the table as unreachable code.

‘`decrement_and_branch_until_zero`’
Conditional branch instruction that decrements a register and jumps if the register is nonzero. Operand 0 is the register to decrement and test; operand 1 is the label to jump to if the register is nonzero. See Section 13.13 [Looping Patterns], page 234.

This optional instruction pattern is only used by the combiner, typically for loops reversed by the loop optimizer when strength reduction is enabled.

‘`doloop_end`’
Conditional branch instruction that decrements a register and jumps if the register is nonzero. This instruction takes five operands: Operand 0 is the register to decrement and test; operand 1 is the number of loop iterations as a `const_int` or `const0_rtx` if this cannot be determined until run-time; operand 2 is the actual or estimated maximum number of iterations as a `const_int`; operand 3 is the number of enclosed loops as a `const_int` (an innermost loop has a value of 1); operand 4 is the label to jump to if the register is nonzero. See Section 13.13 [Looping Patterns], page 234.

This optional instruction pattern should be defined for machines with low-overhead looping instructions as the loop optimizer will try to modify suitable loops to utilize it. If nested low-overhead looping is not supported, use a `define_expand` (see Section 13.15 [Expander Definitions], page 237) and make the pattern fail if operand 3 is not `const1_rtx`. Similarly, if the actual or estimated maximum number of iterations is too large for this instruction, make it fail.

‘`doloop_begin`’
Companion instruction to `doloop_end` required for machines that need to perform some initialization, such as loading special registers used by a low-overhead looping instruction. If initialization insns do not always need to be emitted, use a `define_expand` (see Section 13.15 [Expander Definitions], page 237) and make it fail.

‘`canonicalize_funcptr_for_compare`’
Canonicalize the function pointer in operand 1 and store the result into operand 0.

Operand 0 is always a `reg` and has mode `Pmode`; operand 1 may be a `reg`, `mem`, `symbol_ref`, `const_int`, etc and also has mode `Pmode`.

Canonicalization of a function pointer usually involves computing the address of the function which would be called if the function pointer were used in an indirect call.

Only define this pattern if function pointers on the target machine can have different values but still call the same function when used in an indirect call.

'save_stack_block'
'save_stack_function'
'save_stack_nonlocal'
'restore_stack_block'
'restore_stack_function'
'restore_stack_nonlocal'

Most machines save and restore the stack pointer by copying it to or from an object of mode `Pmode`. Do not define these patterns on such machines.

Some machines require special handling for stack pointer saves and restores. On those machines, define the patterns corresponding to the non-standard cases by using a `define_expand` (see Section 13.15 [Expander Definitions], page 237) that produces the required insns. The three types of saves and restores are:

1. '`save_stack_block`' saves the stack pointer at the start of a block that allocates a variable-sized object, and '`restore_stack_block`' restores the stack pointer when the block is exited.
2. '`save_stack_function`' and '`restore_stack_function`' do a similar job for the outermost block of a function and are used when the function allocates variable-sized objects or calls `alloca`. Only the epilogue uses the restored stack pointer, allowing a simpler save or restore sequence on some machines.
3. '`save_stack_nonlocal`' is used in functions that contain labels branched to by nested functions. It saves the stack pointer in such a way that the inner function can use '`restore_stack_nonlocal`' to restore the stack pointer. The compiler generates code to restore the frame and argument pointer registers, but some machines require saving and restoring additional data such as register window information or stack backchains. Place insns in these patterns to save and restore any such required data.

When saving the stack pointer, operand 0 is the save area and operand 1 is the stack pointer. The mode used to allocate the save area defaults to `Pmode` but you can override that choice by defining the `STACK_SAVEAREA_MODE` macro (see Section 14.5 [Storage Layout], page 278). You must specify an integral mode, or `VOIDmode` if no save area is needed for a particular type of save (either because no save is needed or because a machine-specific save area can be used). Operand 0 is the stack pointer and operand 1 is the save area for restore operations. If '`save_stack_block`' is defined, operand 0 must not be `VOIDmode` since these saves can be arbitrarily nested.

A save area is a `mem` that is at a constant offset from `virtual_stack_vars_rtx` when the stack pointer is saved for use by nonlocal gotos and a `reg` in the other two cases.

'allocate_stack'

Subtract (or add if `STACK_GROWS_DOWNWARD` is undefined) operand 1 from the stack pointer to create space for dynamically allocated data.

Store the resultant pointer to this space into operand 0. If you are allocating space from the main stack, do this by emitting a move insn to copy `virtual_stack_dynamic_rtx` to operand 0. If you are allocating the space elsewhere,

generate code to copy the location of the space to operand 0. In the latter case, you must ensure this space gets freed when the corresponding space on the main stack is free.

Do not define this pattern if all that must be done is the subtraction. Some machines require other operations such as stack probes or maintaining the back chain. Define this pattern to emit those operations in addition to updating the stack pointer.

'`check_stack`'
: If stack checking cannot be done on your system by probing the stack with a load or store instruction (see Section 14.9.3 [Stack Checking], page 309), define this pattern to perform the needed check and signaling an error if the stack has overflowed. The single operand is the location in the stack furthest from the current stack pointer that you need to validate. Normally, on machines where this pattern is needed, you would obtain the stack limit from a global or thread-specific variable or register.

'`nonlocal_goto`'
: Emit code to generate a non-local goto, e.g., a jump from one function to a label in an outer function. This pattern has four arguments, each representing a value to be used in the jump. The first argument is to be loaded into the frame pointer, the second is the address to branch to (code to dispatch to the actual label), the third is the address of a location where the stack is saved, and the last is the address of the label, to be placed in the location for the incoming static chain.

  On most machines you need not define this pattern, since GCC will already generate the correct code, which is to load the frame pointer and static chain, restore the stack (using the '`restore_stack_nonlocal`' pattern, if defined), and jump indirectly to the dispatcher. You need only define this pattern if this code will not work on your machine.

'`nonlocal_goto_receiver`'
: This pattern, if defined, contains code needed at the target of a nonlocal goto after the code already generated by GCC. You will not normally need to define this pattern. A typical reason why you might need this pattern is if some value, such as a pointer to a global table, must be restored when the frame pointer is restored. Note that a nonlocal goto only occurs within a unit-of-translation, so a global table pointer that is shared by all functions of a given module need not be restored. There are no arguments.

'`exception_receiver`'
: This pattern, if defined, contains code needed at the site of an exception handler that isn't needed at the site of a nonlocal goto. You will not normally need to define this pattern. A typical reason why you might need this pattern is if some value, such as a pointer to a global table, must be restored after control flow is branched to the handler of an exception. There are no arguments.

'`builtin_setjmp_setup`'
: This pattern, if defined, contains additional code needed to initialize the `jmp_buf`. You will not normally need to define this pattern. A typical reason why

you might need this pattern is if some value, such as a pointer to a global table, must be restored. Though it is preferred that the pointer value be recalculated if possible (given the address of a label for instance). The single argument is a pointer to the `jmp_buf`. Note that the buffer is five words long and that the first three are normally used by the generic mechanism.

'`builtin_setjmp_receiver`'
This pattern, if defined, contains code needed at the site of an built-in setjmp that isn't needed at the site of a nonlocal goto. You will not normally need to define this pattern. A typical reason why you might need this pattern is if some value, such as a pointer to a global table, must be restored. It takes one argument, which is the label to which builtin_longjmp transfered control; this pattern may be emitted at a small offset from that label.

'`builtin_longjmp`'
This pattern, if defined, performs the entire action of the longjmp. You will not normally need to define this pattern unless you also define `builtin_setjmp_setup`. The single argument is a pointer to the `jmp_buf`.

'`eh_return`'
This pattern, if defined, affects the way `__builtin_eh_return`, and thence the call frame exception handling library routines, are built. It is intended to handle non-trivial actions needed along the abnormal return path.

The address of the exception handler to which the function should return is passed as operand to this pattern. It will normally need to copied by the pattern to some special register or memory location. If the pattern needs to determine the location of the target call frame in order to do so, it may use `EH_RETURN_STACKADJ_RTX`, if defined; it will have already been assigned.

If this pattern is not defined, the default action will be to simply copy the return address to `EH_RETURN_HANDLER_RTX`. Either that macro or this pattern needs to be defined if call frame exception handling is to be used.

'`prologue`'
This pattern, if defined, emits RTL for entry to a function. The function entry is responsible for setting up the stack frame, initializing the frame pointer register, saving callee saved registers, etc.

Using a prologue pattern is generally preferred over defining `TARGET_ASM_FUNCTION_PROLOGUE` to emit assembly code for the prologue.

The `prologue` pattern is particularly useful for targets which perform instruction scheduling.

'`epilogue`'
This pattern emits RTL for exit from a function. The function exit is responsible for deallocating the stack frame, restoring callee saved registers and emitting the return instruction.

Using an epilogue pattern is generally preferred over defining `TARGET_ASM_FUNCTION_EPILOGUE` to emit assembly code for the epilogue.

The `epilogue` pattern is particularly useful for targets which perform instruction scheduling or which have delay slots for their return instruction.

‘`sibcall_epilogue`’
: This pattern, if defined, emits RTL for exit from a function without the final branch back to the calling function. This pattern will be emitted before any sibling call (aka tail call) sites.

  The `sibcall_epilogue` pattern must not clobber any arguments used for parameter passing or any stack slots for arguments passed to the current function.

‘`trap`’
: This pattern, if defined, signals an error, typically by causing some kind of signal to be raised. Among other places, it is used by the Java front end to signal ‘invalid array index’ exceptions.

‘`conditional_trap`’
: Conditional trap instruction. Operand 0 is a piece of RTL which performs a comparison. Operand 1 is the trap code, an integer.

  A typical `conditional_trap` pattern looks like

```
(define_insn "conditional_trap"
  [(trap_if (match_operator 0 "trap_operator"
             [(cc0) (const_int 0)])
            (match_operand 1 "const_int_operand" "i"))]
  ""
  "...")
```

‘`prefetch`’
: This pattern, if defined, emits code for a non-faulting data prefetch instruction. Operand 0 is the address of the memory to prefetch. Operand 1 is a constant 1 if the prefetch is preparing for a write to the memory address, or a constant 0 otherwise. Operand 2 is the expected degree of temporal locality of the data and is a value between 0 and 3, inclusive; 0 means that the data has no temporal locality, so it need not be left in the cache after the access; 3 means that the data has a high degree of temporal locality and should be left in all levels of cache possible; 1 and 2 mean, respectively, a low or moderate degree of temporal locality.

  Targets that do not support write prefetches or locality hints can ignore the values of operands 1 and 2.

‘`memory_barrier`’
: If the target memory model is not fully synchronous, then this pattern should be defined to an instruction that orders both loads and stores before the instruction with respect to loads and stores after the instruction. This pattern has no operands.

‘`sync_compare_and_swap`*`mode`*’
: This pattern, if defined, emits code for an atomic compare-and-swap operation. Operand 1 is the memory on which the atomic operation is performed. Operand 2 is the “old” value to be compared against the current contents of the memory location. Operand 3 is the “new” value to store in the memory if the compare succeeds. Operand 0 is the result of the operation; it should contain the contents of the memory before the operation. If the compare succeeds, this should obviously be a copy of operand 2.

  This pattern must show that both operand 0 and operand 1 are modified.

This pattern must issue any memory barrier instructions such that all memory operations before the atomic operation occur before the atomic operation and all memory operations after the atomic operation occur after the atomic operation.

'`sync_compare_and_swap_cc`*mode*'
: This pattern is just like `sync_compare_and_swap`*mode*, except it should act as if compare part of the compare-and-swap were issued via `cmp`*m*. This comparison will only be used with `EQ` and `NE` branches and `setcc` operations.

  Some targets do expose the success or failure of the compare-and-swap operation via the status flags. Ideally we wouldn't need a separate named pattern in order to take advantage of this, but the combine pass does not handle patterns with multiple sets, which is required by definition for `sync_compare_and_swap`*mode*.

'`sync_add`*mode*', '`sync_sub`*mode*'
'`sync_ior`*mode*', '`sync_and`*mode*'
'`sync_xor`*mode*', '`sync_nand`*mode*'
: These patterns emit code for an atomic operation on memory. Operand 0 is the memory on which the atomic operation is performed. Operand 1 is the second operand to the binary operator.

  The "nand" operation is `~op0 & op1`.

  This pattern must issue any memory barrier instructions such that all memory operations before the atomic operation occur before the atomic operation and all memory operations after the atomic operation occur after the atomic operation.

  If these patterns are not defined, the operation will be constructed from a compare-and-swap operation, if defined.

'`sync_old_add`*mode*', '`sync_old_sub`*mode*'
'`sync_old_ior`*mode*', '`sync_old_and`*mode*'
'`sync_old_xor`*mode*', '`sync_old_nand`*mode*'
: These patterns are emit code for an atomic operation on memory, and return the value that the memory contained before the operation. Operand 0 is the result value, operand 1 is the memory on which the atomic operation is performed, and operand 2 is the second operand to the binary operator.

  This pattern must issue any memory barrier instructions such that all memory operations before the atomic operation occur before the atomic operation and all memory operations after the atomic operation occur after the atomic operation.

  If these patterns are not defined, the operation will be constructed from a compare-and-swap operation, if defined.

'`sync_new_add`*mode*', '`sync_new_sub`*mode*'
'`sync_new_ior`*mode*', '`sync_new_and`*mode*'
'`sync_new_xor`*mode*', '`sync_new_nand`*mode*'
: These patterns are like their `sync_old_`*op* counterparts, except that they return the value that exists in the memory location after the operation, rather than before the operation.

‘`sync_lock_test_and_set`*mode*’
: This pattern takes two forms, based on the capabilities of the target. In either case, operand 0 is the result of the operand, operand 1 is the memory on which the atomic operation is performed, and operand 2 is the value to set in the lock.

  In the ideal case, this operation is an atomic exchange operation, in which the previous value in memory operand is copied into the result operand, and the value operand is stored in the memory operand.

  For less capable targets, any value operand that is not the constant 1 should be rejected with `FAIL`. In this case the target may use an atomic test-and-set bit operation. The result operand should contain 1 if the bit was previously set and 0 if the bit was previously clear. The true contents of the memory operand are implementation defined.

  This pattern must issue any memory barrier instructions such that the pattern as a whole acts as an acquire barrier, that is all memory operations after the pattern do not occur until the lock is acquired.

  If this pattern is not defined, the operation will be constructed from a compare-and-swap operation, if defined.

‘`sync_lock_release`*mode*’
: This pattern, if defined, releases a lock set by `sync_lock_test_and_set`*mode*. Operand 0 is the memory that contains the lock; operand 1 is the value to store in the lock.

  If the target doesn't implement full semantics for `sync_lock_test_and_set`*mode*, any value operand which is not the constant 0 should be rejected with `FAIL`, and the true contents of the memory operand are implementation defined.

  This pattern must issue any memory barrier instructions such that the pattern as a whole acts as a release barrier, that is the lock is released only after all previous memory operations have completed.

  If this pattern is not defined, then a `memory_barrier` pattern will be emitted, followed by a store of the value to the memory operand.

‘`stack_protect_set`’
: This pattern, if defined, moves a `Pmode` value from the memory in operand 1 to the memory in operand 0 without leaving the value in a register afterward. This is to avoid leaking the value some place that an attacker might use to rewrite the stack guard slot after having clobbered it.

  If this pattern is not defined, then a plain move pattern is generated.

‘`stack_protect_test`’
: This pattern, if defined, compares a `Pmode` value from the memory in operand 1 with the memory in operand 0 without leaving the value in a register afterward and branches to operand 2 if the values weren't equal.

  If this pattern is not defined, then a plain compare pattern and conditional branch pattern is used.

13.10 When the Order of Patterns Matters

Sometimes an insn can match more than one instruction pattern. Then the pattern that appears first in the machine description is the one used. Therefore, more specific patterns (patterns that will match fewer things) and faster instructions (those that will produce better code when they do match) should usually go first in the description.

In some cases the effect of ordering the patterns can be used to hide a pattern when it is not valid. For example, the 68000 has an instruction for converting a fullword to floating point and another for converting a byte to floating point. An instruction converting an integer to floating point could match either one. We put the pattern to convert the fullword first to make sure that one will be used rather than the other. (Otherwise a large integer might be generated as a single-byte immediate quantity, which would not work.) Instead of using this pattern ordering it would be possible to make the pattern for convert-a-byte smart enough to deal properly with any constant value.

13.11 Interdependence of Patterns

Every machine description must have a named pattern for each of the conditional branch names '`b`*cond*'. The recognition template must always have the form

```
(set (pc)
     (if_then_else (cond (cc0) (const_int 0))
                   (label_ref (match_operand 0 "" ""))
                   (pc)))
```

In addition, every machine description must have an anonymous pattern for each of the possible reverse-conditional branches. Their templates look like

```
(set (pc)
     (if_then_else (cond (cc0) (const_int 0))
                   (pc)
                   (label_ref (match_operand 0 "" ""))))
```

They are necessary because jump optimization can turn direct-conditional branches into reverse-conditional branches.

It is often convenient to use the `match_operator` construct to reduce the number of patterns that must be specified for branches. For example,

```
(define_insn ""
  [(set (pc)
        (if_then_else (match_operator 0 "comparison_operator"
                                      [(cc0) (const_int 0)])
                      (pc)
                      (label_ref (match_operand 1 "" ""))))]
  "condition"
  "...")
```

In some cases machines support instructions identical except for the machine mode of one or more operands. For example, there may be "sign-extend halfword" and "sign-extend byte" instructions whose patterns are

```
(set (match_operand:SI 0 ...)
     (extend:SI (match_operand:HI 1 ...)))

(set (match_operand:SI 0 ...)
     (extend:SI (match_operand:QI 1 ...)))
```

Constant integers do not specify a machine mode, so an instruction to extend a constant value could match either pattern. The pattern it actually will match is the one that appears first in the file. For correct results, this must be the one for the widest possible mode (`HImode`, here). If the pattern matches the `QImode` instruction, the results will be incorrect if the constant value does not actually fit that mode.

Such instructions to extend constants are rarely generated because they are optimized away, but they do occasionally happen in nonoptimized compilations.

If a constraint in a pattern allows a constant, the reload pass may replace a register with a constant permitted by the constraint in some cases. Similarly for memory references. Because of this substitution, you should not provide separate patterns for increment and decrement instructions. Instead, they should be generated from the same pattern that supports register-register add insns by examining the operands and generating the appropriate machine instruction.

13.12 Defining Jump Instruction Patterns

For most machines, GCC assumes that the machine has a condition code. A comparison insn sets the condition code, recording the results of both signed and unsigned comparison of the given operands. A separate branch insn tests the condition code and branches or not according its value. The branch insns come in distinct signed and unsigned flavors. Many common machines, such as the VAX, the 68000 and the 32000, work this way.

Some machines have distinct signed and unsigned compare instructions, and only one set of conditional branch instructions. The easiest way to handle these machines is to treat them just like the others until the final stage where assembly code is written. At this time, when outputting code for the compare instruction, peek ahead at the following branch using `next_cc0_user (insn)`. (The variable `insn` refers to the insn being output, in the output-writing code in an instruction pattern.) If the RTL says that is an unsigned branch, output an unsigned compare; otherwise output a signed compare. When the branch itself is output, you can treat signed and unsigned branches identically.

The reason you can do this is that GCC always generates a pair of consecutive RTL insns, possibly separated by `note` insns, one to set the condition code and one to test it, and keeps the pair inviolate until the end.

To go with this technique, you must define the machine-description macro `NOTICE_UPDATE_CC` to do `CC_STATUS_INIT`; in other words, no compare instruction is superfluous.

Some machines have compare-and-branch instructions and no condition code. A similar technique works for them. When it is time to "output" a compare instruction, record its operands in two static variables. When outputting the branch-on-condition-code instruction that follows, actually output a compare-and-branch instruction that uses the remembered operands.

It also works to define patterns for compare-and-branch instructions. In optimizing compilation, the pair of compare and branch instructions will be combined according to these patterns. But this does not happen if optimization is not requested. So you must use one of the solutions above in addition to any special patterns you define.

In many RISC machines, most instructions do not affect the condition code and there may not even be a separate condition code register. On these machines, the restriction

that the definition and use of the condition code be adjacent insns is not necessary and can prevent important optimizations. For example, on the IBM RS/6000, there is a delay for taken branches unless the condition code register is set three instructions earlier than the conditional branch. The instruction scheduler cannot perform this optimization if it is not permitted to separate the definition and use of the condition code register.

On these machines, do not use `(cc0)`, but instead use a register to represent the condition code. If there is a specific condition code register in the machine, use a hard register. If the condition code or comparison result can be placed in any general register, or if there are multiple condition registers, use a pseudo register.

On some machines, the type of branch instruction generated may depend on the way the condition code was produced; for example, on the 68k and SPARC, setting the condition code directly from an add or subtract instruction does not clear the overflow bit the way that a test instruction does, so a different branch instruction must be used for some conditional branches. For machines that use `(cc0)`, the set and use of the condition code must be adjacent (separated only by `note` insns) allowing flags in `cc_status` to be used. (See Section 14.14 [Condition Code], page 338.) Also, the comparison and branch insns can be located from each other by using the functions `prev_cc0_setter` and `next_cc0_user`.

However, this is not true on machines that do not use `(cc0)`. On those machines, no assumptions can be made about the adjacency of the compare and branch insns and the above methods cannot be used. Instead, we use the machine mode of the condition code register to record different formats of the condition code register.

Registers used to store the condition code value should have a mode that is in class `MODE_CC`. Normally, it will be `CCmode`. If additional modes are required (as for the add example mentioned above in the SPARC), define them in '*machine*`-modes.def`' (see Section 14.14 [Condition Code], page 338). Also define `SELECT_CC_MODE` to choose a mode given an operand of a compare.

If it is known during RTL generation that a different mode will be required (for example, if the machine has separate compare instructions for signed and unsigned quantities, like most IBM processors), they can be specified at that time.

If the cases that require different modes would be made by instruction combination, the macro `SELECT_CC_MODE` determines which machine mode should be used for the comparison result. The patterns should be written using that mode. To support the case of the add on the SPARC discussed above, we have the pattern

```
(define_insn ""
  [(set (reg:CC_NOOV 0)
        (compare:CC_NOOV
          (plus:SI (match_operand:SI 0 "register_operand" "%r")
                   (match_operand:SI 1 "arith_operand" "rI"))
          (const_int 0)))]
  ""
  "...")
```

The `SELECT_CC_MODE` macro on the SPARC returns `CC_NOOVmode` for comparisons whose argument is a `plus`.

13.13 Defining Looping Instruction Patterns

Some machines have special jump instructions that can be utilized to make loops more efficient. A common example is the 68000 '`dbra`' instruction which performs a decrement of a register and a branch if the result was greater than zero. Other machines, in particular digital signal processors (DSPs), have special block repeat instructions to provide low-overhead loop support. For example, the TI TMS320C3x/C4x DSPs have a block repeat instruction that loads special registers to mark the top and end of a loop and to count the number of loop iterations. This avoids the need for fetching and executing a '`dbra`'-like instruction and avoids pipeline stalls associated with the jump.

GCC has three special named patterns to support low overhead looping. They are '`decrement_and_branch_until_zero`', '`doloop_begin`', and '`doloop_end`'. The first pattern, '`decrement_and_branch_until_zero`', is not emitted during RTL generation but may be emitted during the instruction combination phase. This requires the assistance of the loop optimizer, using information collected during strength reduction, to reverse a loop to count down to zero. Some targets also require the loop optimizer to add a `REG_NONNEG` note to indicate that the iteration count is always positive. This is needed if the target performs a signed loop termination test. For example, the 68000 uses a pattern similar to the following for its `dbra` instruction:

```
(define_insn "decrement_and_branch_until_zero"
  [(set (pc)
(if_then_else
  (ge (plus:SI (match_operand:SI 0 "general_operand" "+d*am")
       (const_int -1))
      (const_int 0))
  (label_ref (match_operand 1 "" ""))
  (pc)))
   (set (match_dup 0)
(plus:SI (match_dup 0)
 (const_int -1)))]
  "find_reg_note (insn, REG_NONNEG, 0)"
  "...")
```

Note that since the insn is both a jump insn and has an output, it must deal with its own reloads, hence the 'm' constraints. Also note that since this insn is generated by the instruction combination phase combining two sequential insns together into an implicit parallel insn, the iteration counter needs to be biased by the same amount as the decrement operation, in this case -1. Note that the following similar pattern will not be matched by the combiner.

```
(define_insn "decrement_and_branch_until_zero"
  [(set (pc)
(if_then_else
  (ge (match_operand:SI 0 "general_operand" "+d*am")
      (const_int 1))
  (label_ref (match_operand 1 "" ""))
  (pc)))
   (set (match_dup 0)
(plus:SI (match_dup 0)
 (const_int -1)))]
  "find_reg_note (insn, REG_NONNEG, 0)"
  "...")
```

The other two special looping patterns, '`doloop_begin`' and '`doloop_end`', are emitted by the loop optimizer for certain well-behaved loops with a finite number of loop iterations using information collected during strength reduction.

The '`doloop_end`' pattern describes the actual looping instruction (or the implicit looping operation) and the '`doloop_begin`' pattern is an optional companion pattern that can be used for initialization needed for some low-overhead looping instructions.

Note that some machines require the actual looping instruction to be emitted at the top of the loop (e.g., the TMS320C3x/C4x DSPs). Emitting the true RTL for a looping instruction at the top of the loop can cause problems with flow analysis. So instead, a dummy `doloop` insn is emitted at the end of the loop. The machine dependent reorg pass checks for the presence of this `doloop` insn and then searches back to the top of the loop, where it inserts the true looping insn (provided there are no instructions in the loop which would cause problems). Any additional labels can be emitted at this point. In addition, if the desired special iteration counter register was not allocated, this machine dependent reorg pass could emit a traditional compare and jump instruction pair.

The essential difference between the '`decrement_and_branch_until_zero`' and the '`doloop_end`' patterns is that the loop optimizer allocates an additional pseudo register for the latter as an iteration counter. This pseudo register cannot be used within the loop (i.e., general induction variables cannot be derived from it), however, in many cases the loop induction variable may become redundant and removed by the flow pass.

13.14 Canonicalization of Instructions

There are often cases where multiple RTL expressions could represent an operation performed by a single machine instruction. This situation is most commonly encountered with logical, branch, and multiply-accumulate instructions. In such cases, the compiler attempts to convert these multiple RTL expressions into a single canonical form to reduce the number of insn patterns required.

In addition to algebraic simplifications, following canonicalizations are performed:

- For commutative and comparison operators, a constant is always made the second operand. If a machine only supports a constant as the second operand, only patterns that match a constant in the second operand need be supplied.
- For associative operators, a sequence of operators will always chain to the left; for instance, only the left operand of an integer `plus` can itself be a `plus`. `and`, `ior`, `xor`, `plus`, `mult`, `smin`, `smax`, `umin`, and `umax` are associative when applied to integers, and sometimes to floating-point.
- For these operators, if only one operand is a `neg`, `not`, `mult`, `plus`, or `minus` expression, it will be the first operand.
- In combinations of `neg`, `mult`, `plus`, and `minus`, the `neg` operations (if any) will be moved inside the operations as far as possible. For instance, `(neg (mult A B))` is canonicalized as `(mult (neg A) B)`, but `(plus (mult (neg A) B) C)` is canonicalized as `(minus A (mult B C))`.
- For the `compare` operator, a constant is always the second operand on machines where `cc0` is used (see Section 13.12 [Jump Patterns], page 233). On other machines, there are rare cases where the compiler might want to construct a `compare` with a constant as the

first operand. However, these cases are not common enough for it to be worthwhile to provide a pattern matching a constant as the first operand unless the machine actually has such an instruction.

An operand of `neg`, `not`, `mult`, `plus`, or `minus` is made the first operand under the same conditions as above.

- `(minus x (const_int n))` is converted to `(plus x (const_int -n))`.
- Within address computations (i.e., inside `mem`), a left shift is converted into the appropriate multiplication by a power of two.
- De Morgan's Law is used to move bitwise negation inside a bitwise logical-and or logical-or operation. If this results in only one operand being a `not` expression, it will be the first one.

  A machine that has an instruction that performs a bitwise logical-and of one operand with the bitwise negation of the other should specify the pattern for that instruction as

```
(define_insn ""
  [(set (match_operand:m 0 ...)
        (and:m (not:m (match_operand:m 1 ...))
               (match_operand:m 2 ...)))]
  "..."
  "...")
```

  Similarly, a pattern for a "NAND" instruction should be written

```
(define_insn ""
  [(set (match_operand:m 0 ...)
        (ior:m (not:m (match_operand:m 1 ...))
               (not:m (match_operand:m 2 ...))))]
  "..."
  "...")
```

  In both cases, it is not necessary to include patterns for the many logically equivalent RTL expressions.
- The only possible RTL expressions involving both bitwise exclusive-or and bitwise negation are `(xor:m x y)` and `(not:m (xor:m x y))`.
- The sum of three items, one of which is a constant, will only appear in the form

```
(plus:m (plus:m x y) constant)
```

- On machines that do not use `cc0`, `(compare x (const_int 0))` will be converted to *x*.
- Equality comparisons of a group of bits (usually a single bit) with zero will be written using `zero_extract` rather than the equivalent `and` or `sign_extract` operations.

13.15 Defining RTL Sequences for Code Generation

On some target machines, some standard pattern names for RTL generation cannot be handled with single insn, but a sequence of RTL insns can represent them. For these target machines, you can write a `define_expand` to specify how to generate the sequence of RTL.

A `define_expand` is an RTL expression that looks almost like a `define_insn`; but, unlike the latter, a `define_expand` is used only for RTL generation and it can produce more than one RTL insn.

A `define_expand` RTX has four operands:

- The name. Each `define_expand` must have a name, since the only use for it is to refer to it by name.
- The RTL template. This is a vector of RTL expressions representing a sequence of separate instructions. Unlike `define_insn`, there is no implicit surrounding `PARALLEL`.
- The condition, a string containing a C expression. This expression is used to express how the availability of this pattern depends on subclasses of target machine, selected by command-line options when GCC is run. This is just like the condition of a `define_insn` that has a standard name. Therefore, the condition (if present) may not depend on the data in the insn being matched, but only the target-machine-type flags. The compiler needs to test these conditions during initialization in order to learn exactly which named instructions are available in a particular run.
- The preparation statements, a string containing zero or more C statements which are to be executed before RTL code is generated from the RTL template.

  Usually these statements prepare temporary registers for use as internal operands in the RTL template, but they can also generate RTL insns directly by calling routines such as `emit_insn`, etc. Any such insns precede the ones that come from the RTL template.

Every RTL insn emitted by a `define_expand` must match some `define_insn` in the machine description. Otherwise, the compiler will crash when trying to generate code for the insn or trying to optimize it.

The RTL template, in addition to controlling generation of RTL insns, also describes the operands that need to be specified when this pattern is used. In particular, it gives a predicate for each operand.

A true operand, which needs to be specified in order to generate RTL from the pattern, should be described with a `match_operand` in its first occurrence in the RTL template. This enters information on the operand's predicate into the tables that record such things. GCC uses the information to preload the operand into a register if that is required for valid RTL code. If the operand is referred to more than once, subsequent references should use `match_dup`.

The RTL template may also refer to internal "operands" which are temporary registers or labels used only within the sequence made by the `define_expand`. Internal operands are substituted into the RTL template with `match_dup`, never with `match_operand`. The values of the internal operands are not passed in as arguments by the compiler when it requests use of this pattern. Instead, they are computed within the pattern, in the preparation statements. These statements compute the values and store them into the appropriate elements of `operands` so that `match_dup` can find them.

There are two special macros defined for use in the preparation statements: `DONE` and `FAIL`. Use them with a following semicolon, as a statement.

`DONE` Use the `DONE` macro to end RTL generation for the pattern. The only RTL insns resulting from the pattern on this occasion will be those already emitted by explicit calls to `emit_insn` within the preparation statements; the RTL template will not be generated.

`FAIL` Make the pattern fail on this occasion. When a pattern fails, it means that the pattern was not truly available. The calling routines in the compiler will try other strategies for code generation using other patterns.

Failure is currently supported only for binary (addition, multiplication, shifting, etc.) and bit-field (`extv`, `extzv`, and `insv`) operations.

If the preparation falls through (invokes neither `DONE` nor `FAIL`), then the `define_expand` acts like a `define_insn` in that the RTL template is used to generate the insn.

The RTL template is not used for matching, only for generating the initial insn list. If the preparation statement always invokes `DONE` or `FAIL`, the RTL template may be reduced to a simple list of operands, such as this example:

```
(define_expand "addsi3"
  [(match_operand:SI 0 "register_operand" "")
   (match_operand:SI 1 "register_operand" "")
   (match_operand:SI 2 "register_operand" "")]
  ""
  "
{
  handle_add (operands[0], operands[1], operands[2]);
  DONE;
}")
```

Here is an example, the definition of left-shift for the SPUR chip:

```
(define_expand "ashlsi3"
  [(set (match_operand:SI 0 "register_operand" "")
        (ashift:SI
          (match_operand:SI 1 "register_operand" "")
          (match_operand:SI 2 "nonmemory_operand" "")))]
  ""
  "

{
  if (GET_CODE (operands[2]) != CONST_INT
      || (unsigned) INTVAL (operands[2]) > 3)
    FAIL;
}")
```

This example uses `define_expand` so that it can generate an RTL insn for shifting when the shift-count is in the supported range of 0 to 3 but fail in other cases where machine insns aren't available. When it fails, the compiler tries another strategy using different patterns (such as, a library call).

If the compiler were able to handle nontrivial condition-strings in patterns with names, then it would be possible to use a `define_insn` in that case. Here is another case (zero-extension on the 68000) which makes more use of the power of `define_expand`:

```
(define_expand "zero_extendhisi2"
  [(set (match_operand:SI 0 "general_operand" "")
        (const_int 0))
   (set (strict_low_part
          (subreg:HI
            (match_dup 0)
            0))
        (match_operand:HI 1 "general_operand" ""))]
  ""
  "operands[1] = make_safe_from (operands[1], operands[0]);")
```

Here two RTL insns are generated, one to clear the entire output operand and the other to copy the input operand into its low half. This sequence is incorrect if the input operand refers to [the old value of] the output operand, so the preparation statement makes sure this isn't so. The function `make_safe_from` copies the `operands[1]` into a temporary register if it refers to `operands[0]`. It does this by emitting another RTL insn.

Finally, a third example shows the use of an internal operand. Zero-extension on the SPUR chip is done by `and`-ing the result against a halfword mask. But this mask cannot be represented by a `const_int` because the constant value is too large to be legitimate on this machine. So it must be copied into a register with `force_reg` and then the register used in the `and`.

```
(define_expand "zero_extendhisi2"
  [(set (match_operand:SI 0 "register_operand" "")
        (and:SI (subreg:SI
                  (match_operand:HI 1 "register_operand" "")
                  0)
                (match_dup 2)))]
  ""
  "operands[2]
     = force_reg (SImode, GEN_INT (65535)); ")
```

Note: If the `define_expand` is used to serve a standard binary or unary arithmetic operation or a bit-field operation, then the last insn it generates must not be a `code_label`, `barrier` or `note`. It must be an `insn`, `jump_insn` or `call_insn`. If you don't need a real insn at the end, emit an insn to copy the result of the operation into itself. Such an insn will generate no code, but it can avoid problems in the compiler.

13.16 Defining How to Split Instructions

There are two cases where you should specify how to split a pattern into multiple insns. On machines that have instructions requiring delay slots (see Section 13.19.7 [Delay Slots], page 255) or that have instructions whose output is not available for multiple cycles (see Section 13.19.8 [Processor pipeline description], page 256), the compiler phases that optimize these cases need to be able to move insns into one-instruction delay slots. However, some insns may generate more than one machine instruction. These insns cannot be placed into a delay slot.

Often you can rewrite the single insn as a list of individual insns, each corresponding to one machine instruction. The disadvantage of doing so is that it will cause the compilation to be slower and require more space. If the resulting insns are too complex, it may also suppress some optimizations. The compiler splits the insn if there is a reason to believe that it might improve instruction or delay slot scheduling.

The insn combiner phase also splits putative insns. If three insns are merged into one insn with a complex expression that cannot be matched by some `define_insn` pattern, the combiner phase attempts to split the complex pattern into two insns that are recognized. Usually it can break the complex pattern into two patterns by splitting out some subexpression. However, in some other cases, such as performing an addition of a large constant in two insns on a RISC machine, the way to split the addition into two insns is machine-dependent.

The `define_split` definition tells the compiler how to split a complex insn into several simpler insns. It looks like this:

```
(define_split
  [insn-pattern]
  "condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation-statements")
```

insn-pattern is a pattern that needs to be split and *condition* is the final condition to be tested, as in a `define_insn`. When an insn matching *insn-pattern* and satisfying *condition* is found, it is replaced in the insn list with the insns given by *new-insn-pattern-1*, *new-insn-pattern-2*, etc.

The *preparation-statements* are similar to those statements that are specified for `define_expand` (see Section 13.15 [Expander Definitions], page 237) and are executed before the new RTL is generated to prepare for the generated code or emit some insns whose pattern is not fixed. Unlike those in `define_expand`, however, these statements must not generate any new pseudo-registers. Once reload has completed, they also must not allocate any space in the stack frame.

Patterns are matched against *insn-pattern* in two different circumstances. If an insn needs to be split for delay slot scheduling or insn scheduling, the insn is already known to be valid, which means that it must have been matched by some `define_insn` and, if `reload_completed` is nonzero, is known to satisfy the constraints of that `define_insn`. In that case, the new insn patterns must also be insns that are matched by some `define_insn` and, if `reload_completed` is nonzero, must also satisfy the constraints of those definitions.

As an example of this usage of `define_split`, consider the following example from '`a29k.md`', which splits a `sign_extend` from `HImode` to `SImode` into a pair of shift insns:

```
(define_split
  [(set (match_operand:SI 0 "gen_reg_operand" "")
        (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))]
  ""
  [(set (match_dup 0)
        (ashift:SI (match_dup 1)
                   (const_int 16)))
   (set (match_dup 0)
        (ashiftrt:SI (match_dup 0)
                     (const_int 16)))]
  "
{ operands[1] = gen_lowpart (SImode, operands[1]); }")
```

When the combiner phase tries to split an insn pattern, it is always the case that the pattern is *not* matched by any `define_insn`. The combiner pass first tries to split a single `set` expression and then the same `set` expression inside a `parallel`, but followed by a `clobber` of a pseudo-reg to use as a scratch register. In these cases, the combiner expects exactly two new insn patterns to be generated. It will verify that these patterns match some `define_insn` definitions, so you need not do this test in the `define_split` (of course, there is no point in writing a `define_split` that will never produce insns that match).

Here is an example of this use of `define_split`, taken from '`rs6000.md`':

```
(define_split
  [(set (match_operand:SI 0 "gen_reg_operand" "")
        (plus:SI (match_operand:SI 1 "gen_reg_operand" "")
                 (match_operand:SI 2 "non_add_cint_operand" "")))]
  ""
```

```
  [(set (match_dup 0) (plus:SI (match_dup 1) (match_dup 3)))
   (set (match_dup 0) (plus:SI (match_dup 0) (match_dup 4)))]
"
{
  int low = INTVAL (operands[2]) & 0xffff;
  int high = (unsigned) INTVAL (operands[2]) >> 16;

  if (low & 0x8000)
    high++, low |= 0xffff0000;

  operands[3] = GEN_INT (high << 16);
  operands[4] = GEN_INT (low);
}")
```

Here the predicate `non_add_cint_operand` matches any `const_int` that is *not* a valid operand of a single add insn. The add with the smaller displacement is written so that it can be substituted into the address of a subsequent operation.

An example that uses a scratch register, from the same file, generates an equality comparison of a register and a large constant:

```
(define_split
  [(set (match_operand:CC 0 "cc_reg_operand" "")
        (compare:CC (match_operand:SI 1 "gen_reg_operand" "")
                    (match_operand:SI 2 "non_short_cint_operand" "")))
   (clobber (match_operand:SI 3 "gen_reg_operand" ""))]
  "find_single_use (operands[0], insn, 0)
   && (GET_CODE (*find_single_use (operands[0], insn, 0)) == EQ
       || GET_CODE (*find_single_use (operands[0], insn, 0)) == NE)"
  [(set (match_dup 3) (xor:SI (match_dup 1) (match_dup 4)))
   (set (match_dup 0) (compare:CC (match_dup 3) (match_dup 5)))]
  "
{
  /* Get the constant we are comparing against, C, and see what it
     looks like sign-extended to 16 bits.  Then see what constant
     could be XOR'ed with C to get the sign-extended value.  */

  int c = INTVAL (operands[2]);
  int sextc = (c << 16) >> 16;
  int xorv = c ^ sextc;

  operands[4] = GEN_INT (xorv);
  operands[5] = GEN_INT (sextc);
}")
```

To avoid confusion, don't write a single `define_split` that accepts some insns that match some `define_insn` as well as some insns that don't. Instead, write two separate `define_split` definitions, one for the insns that are valid and one for the insns that are not valid.

The splitter is allowed to split jump instructions into sequence of jumps or create new jumps in while splitting non-jump instructions. As the central flowgraph and branch prediction information needs to be updated, several restriction apply.

Splitting of jump instruction into sequence that over by another jump instruction is always valid, as compiler expect identical behavior of new jump. When new sequence contains multiple jump instructions or new labels, more assistance is needed. Splitter is required to create only unconditional jumps, or simple conditional jump instructions. Additionally

it must attach a `REG_BR_PROB` note to each conditional jump. A global variable `split_branch_probability` holds the probability of the original branch in case it was an simple conditional jump, −1 otherwise. To simplify recomputing of edge frequencies, the new sequence is required to have only forward jumps to the newly created labels.

For the common case where the pattern of a define_split exactly matches the pattern of a define_insn, use `define_insn_and_split`. It looks like this:

```
(define_insn_and_split
  [insn-pattern]
  "condition"
  "output-template"
  "split-condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation-statements"
  [insn-attributes])
```

insn-pattern, *condition*, *output-template*, and *insn-attributes* are used as in `define_insn`. The *new-insn-pattern* vector and the *preparation-statements* are used as in a `define_split`. The *split-condition* is also used as in `define_split`, with the additional behavior that if the condition starts with '`&&`', the condition used for the split will be the constructed as a logical "and" of the split condition with the insn condition. For example, from i386.md:

```
(define_insn_and_split "zero_extendhisi2_and"
  [(set (match_operand:SI 0 "register_operand" "=r")
     (zero_extend:SI (match_operand:HI 1 "register_operand" "0")))
   (clobber (reg:CC 17))]
  "TARGET_ZERO_EXTEND_WITH_AND && !optimize_size"
  "#"
  "&& reload_completed"
  [(parallel [(set (match_dup 0)
                   (and:SI (match_dup 0) (const_int 65535)))
      (clobber (reg:CC 17))])]
  ""
  [(set_attr "type" "alu1")])
```

In this case, the actual split condition will be '`TARGET_ZERO_EXTEND_WITH_AND && !optimize_size && reload_completed`'.

The `define_insn_and_split` construction provides exactly the same functionality as two separate `define_insn` and `define_split` patterns. It exists for compactness, and as a maintenance tool to prevent having to ensure the two patterns' templates match.

13.17 Including Patterns in Machine Descriptions.

The `include` pattern tells the compiler tools where to look for patterns that are in files other than in the file '`.md`'. This is used only at build time and there is no preprocessing allowed.

It looks like:

```
(include
  pathname)
```

For example:

```
(include "filestuff")
```

Where *pathname* is a string that specifies the location of the file, specifies the include file to be in '`gcc/config/target/filestuff`'. The directory '`gcc/config/target`' is regarded as the default directory.

Machine descriptions may be split up into smaller more manageable subsections and placed into subdirectories.

By specifying:

```
(include "BOGUS/filestuff")
```

the include file is specified to be in '`gcc/config/`*target*`/BOGUS/filestuff`'.

Specifying an absolute path for the include file such as;

```
(include "/u2/BOGUS/filestuff")
```

is permitted but is not encouraged.

13.17.1 RTL Generation Tool Options for Directory Search

The '`-I`*dir*' option specifies directories to search for machine descriptions. For example:

```
genrecog -I/p1/abc/proc1 -I/p2/abcd/pro2 target.md
```

Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system machine definition file, substituting your own version, since these directories are searched before the default machine description file directories. If you use more than one '`-I`' option, the directories are scanned in left-to-right order; the standard default directory come after.

13.18 Machine-Specific Peephole Optimizers

In addition to instruction patterns the '`md`' file may contain definitions of machine-specific peephole optimizations.

The combiner does not notice certain peephole optimizations when the data flow in the program does not suggest that it should try them. For example, sometimes two consecutive insns related in purpose can be combined even though the second one does not appear to use a register computed in the first one. A machine-specific peephole optimizer can detect such opportunities.

There are two forms of peephole definitions that may be used. The original `define_peephole` is run at assembly output time to match insns and substitute assembly text. Use of `define_peephole` is deprecated.

A newer `define_peephole2` matches insns and substitutes new insns. The `peephole2` pass is run after register allocation but before scheduling, which may result in much better code for targets that do scheduling.

13.18.1 RTL to Text Peephole Optimizers

A definition looks like this:

```
(define_peephole
  [insn-pattern-1
   insn-pattern-2
   ...]
  "condition"
  "template"
  "optional-insn-attributes")
```

The last string operand may be omitted if you are not using any machine-specific information in this machine description. If present, it must obey the same rules as in a `define_insn`.

In this skeleton, *insn-pattern-1* and so on are patterns to match consecutive insns. The optimization applies to a sequence of insns when *insn-pattern-1* matches the first one, *insn-pattern-2* matches the next, and so on.

Each of the insns matched by a peephole must also match a `define_insn`. Peepholes are checked only at the last stage just before code generation, and only optionally. Therefore, any insn which would match a peephole but no `define_insn` will cause a crash in code generation in an unoptimized compilation, or at various optimization stages.

The operands of the insns are matched with `match_operands`, `match_operator`, and `match_dup`, as usual. What is not usual is that the operand numbers apply to all the insn patterns in the definition. So, you can check for identical operands in two insns by using `match_operand` in one insn and `match_dup` in the other.

The operand constraints used in `match_operand` patterns do not have any direct effect on the applicability of the peephole, but they will be validated afterward, so make sure your constraints are general enough to apply whenever the peephole matches. If the peephole matches but the constraints are not satisfied, the compiler will crash.

It is safe to omit constraints in all the operands of the peephole; or you can write constraints which serve as a double-check on the criteria previously tested.

Once a sequence of insns matches the patterns, the *condition* is checked. This is a C expression which makes the final decision whether to perform the optimization (we do so if the expression is nonzero). If *condition* is omitted (in other words, the string is empty) then the optimization is applied to every sequence of insns that matches the patterns.

The defined peephole optimizations are applied after register allocation is complete. Therefore, the peephole definition can check which operands have ended up in which kinds of registers, just by looking at the operands.

The way to refer to the operands in *condition* is to write `operands[i]` for operand number *i* (as matched by `(match_operand i ...)`). Use the variable `insn` to refer to the last of the insns being matched; use `prev_active_insn` to find the preceding insns.

When optimizing computations with intermediate results, you can use *condition* to match only when the intermediate results are not used elsewhere. Use the C expression `dead_or_set_p (insn, op)`, where *insn* is the insn in which you expect the value to be used for the last time (from the value of `insn`, together with use of `prev_nonnote_insn`), and *op* is the intermediate value (from `operands[i]`).

Applying the optimization means replacing the sequence of insns with one new insn. The *template* controls ultimate output of assembler code for this combined insn. It works exactly

like the template of a `define_insn`. Operand numbers in this template are the same ones used in matching the original sequence of insns.

The result of a defined peephole optimizer does not need to match any of the insn patterns in the machine description; it does not even have an opportunity to match them. The peephole optimizer definition itself serves as the insn pattern to control how the insn is output.

Defined peephole optimizers are run as assembler code is being output, so the insns they produce are never combined or rearranged in any way.

Here is an example, taken from the 68000 machine description:

```
(define_peephole
  [(set (reg:SI 15) (plus:SI (reg:SI 15) (const_int 4)))
   (set (match_operand:DF 0 "register_operand" "=f")
        (match_operand:DF 1 "register_operand" "ad"))]
  "FP_REG_P (operands[0]) && ! FP_REG_P (operands[1])"
{
  rtx xoperands[2];
  xoperands[1] = gen_rtx_REG (SImode, REGNO (operands[1]) + 1);
#ifdef MOTOROLA
  output_asm_insn ("move.l %1,(sp)", xoperands);
  output_asm_insn ("move.l %1,-(sp)", operands);
  return "fmove.d (sp)+,%0";
#else
  output_asm_insn ("movel %1,sp@", xoperands);
  output_asm_insn ("movel %1,sp@-", operands);
  return "fmoved sp@+,%0";
#endif
})
```

The effect of this optimization is to change

```
jbsr _foobar
addql #4,sp
movel d1,sp@-
movel d0,sp@-
fmoved sp@+,fp0
```

into

```
jbsr _foobar
movel d1,sp@
movel d0,sp@-
fmoved sp@+,fp0
```

insn-pattern-1 and so on look *almost* like the second operand of `define_insn`. There is one important difference: the second operand of `define_insn` consists of one or more RTX's enclosed in square brackets. Usually, there is only one: then the same action can be written as an element of a `define_peephole`. But when there are multiple actions in a `define_insn`, they are implicitly enclosed in a `parallel`. Then you must explicitly write the `parallel`, and the square brackets within it, in the `define_peephole`. Thus, if an insn pattern looks like this,

```
(define_insn "divmodsi4"
  [(set (match_operand:SI 0 "general_operand" "=d")
        (div:SI (match_operand:SI 1 "general_operand" "0")
                (match_operand:SI 2 "general_operand" "dmsK")))
   (set (match_operand:SI 3 "general_operand" "=d")
        (mod:SI (match_dup 1) (match_dup 2)))]
  "TARGET_68020"
```

```
  "divsl%.l %2,%3:%0")
```

then the way to mention this insn in a peephole is as follows:

```
(define_peephole
  [...
   (parallel
    [(set (match_operand:SI 0 "general_operand" "=d")
          (div:SI (match_operand:SI 1 "general_operand" "0")
                  (match_operand:SI 2 "general_operand" "dmsK")))
     (set (match_operand:SI 3 "general_operand" "=d")
          (mod:SI (match_dup 1) (match_dup 2)))])
   ...]
  ...)
```

13.18.2 RTL to RTL Peephole Optimizers

The `define_peephole2` definition tells the compiler how to substitute one sequence of instructions for another sequence, what additional scratch registers may be needed and what their lifetimes must be.

```
(define_peephole2
  [insn-pattern-1
   insn-pattern-2
   ...]
  "condition"
  [new-insn-pattern-1
   new-insn-pattern-2
   ...]
  "preparation-statements")
```

The definition is almost identical to `define_split` (see Section 13.16 [Insn Splitting], page 240) except that the pattern to match is not a single instruction, but a sequence of instructions.

It is possible to request additional scratch registers for use in the output template. If appropriate registers are not free, the pattern will simply not match.

Scratch registers are requested with a `match_scratch` pattern at the top level of the input pattern. The allocated register (initially) will be dead at the point requested within the original sequence. If the scratch is used at more than a single point, a `match_dup` pattern at the top level of the input pattern marks the last position in the input sequence at which the register must be available.

Here is an example from the IA-32 machine description:

```
(define_peephole2
  [(match_scratch:SI 2 "r")
   (parallel [(set (match_operand:SI 0 "register_operand" "")
                   (match_operator:SI 3 "arith_or_logical_operator"
                     [(match_dup 0)
                      (match_operand:SI 1 "memory_operand" "")]))
              (clobber (reg:CC 17))])]
  "! optimize_size && ! TARGET_READ_MODIFY"
  [(set (match_dup 2) (match_dup 1))
   (parallel [(set (match_dup 0)
                   (match_op_dup 3 [(match_dup 0) (match_dup 2)]))
              (clobber (reg:CC 17))])]
  "")
```

This pattern tries to split a load from its use in the hopes that we'll be able to schedule around the memory load latency. It allocates a single `SImode` register of class `GENERAL_REGS` (`"r"`) that needs to be live only at the point just before the arithmetic.

A real example requiring extended scratch lifetimes is harder to come by, so here's a silly made-up example:

```
(define_peephole2
  [(match_scratch:SI 4 "r")
   (set (match_operand:SI 0 "" "") (match_operand:SI 1 "" ""))
   (set (match_operand:SI 2 "" "") (match_dup 1))
   (match_dup 4)
   (set (match_operand:SI 3 "" "") (match_dup 1))]
  "/* determine 1 does not overlap 0 and 2 */"
  [(set (match_dup 4) (match_dup 1))
   (set (match_dup 0) (match_dup 4))
   (set (match_dup 2) (match_dup 4))]
   (set (match_dup 3) (match_dup 4))]
  "")
```

If we had not added the `(match_dup 4)` in the middle of the input sequence, it might have been the case that the register we chose at the beginning of the sequence is killed by the first or second `set`.

13.19 Instruction Attributes

In addition to describing the instruction supported by the target machine, the '`md`' file also defines a group of *attributes* and a set of values for each. Every generated insn is assigned a value for each attribute. One possible attribute would be the effect that the insn has on the machine's condition code. This attribute can then be used by `NOTICE_UPDATE_CC` to track the condition codes.

13.19.1 Defining Attributes and their Values

The `define_attr` expression is used to define each attribute required by the target machine. It looks like:

```
(define_attr name list-of-values default)
```

name is a string specifying the name of the attribute being defined.

list-of-values is either a string that specifies a comma-separated list of values that can be assigned to the attribute, or a null string to indicate that the attribute takes numeric values.

default is an attribute expression that gives the value of this attribute for insns that match patterns whose definition does not include an explicit value for this attribute. See Section 13.19.4 [Attr Example], page 253, for more information on the handling of defaults. See Section 13.19.6 [Constant Attributes], page 255, for information on attributes that do not depend on any particular insn.

For each defined attribute, a number of definitions are written to the '`insn-attr.h`' file. For cases where an explicit set of values is specified for an attribute, the following are defined:

- A '`#define`' is written for the symbol '`HAVE_ATTR_`*name*'.

- An enumerated class is defined for '`attr_name`' with elements of the form '*`upper-name_upper-value`*' where the attribute name and value are first converted to uppercase.
- A function '`get_attr_name`' is defined that is passed an insn and returns the attribute value for that insn.

For example, if the following is present in the '`md`' file:

```
(define_attr "type" "branch,fp,load,store,arith" ...)
```

the following lines will be written to the file '`insn-attr.h`'.

```
#define HAVE_ATTR_type
enum attr_type {TYPE_BRANCH, TYPE_FP, TYPE_LOAD,
                TYPE_STORE, TYPE_ARITH};
extern enum attr_type get_attr_type ();
```

If the attribute takes numeric values, no `enum` type will be defined and the function to obtain the attribute's value will return `int`.

13.19.2 Attribute Expressions

RTL expressions used to define attributes use the codes described above plus a few specific to attribute definitions, to be discussed below. Attribute value expressions must have one of the following forms:

`(const_int` *`i`*`)`
: The integer *i* specifies the value of a numeric attribute. *i* must be non-negative.

  The value of a numeric attribute can be specified either with a `const_int`, or as an integer represented as a string in `const_string`, `eq_attr` (see below), `attr`, `symbol_ref`, simple arithmetic expressions, and `set_attr` overrides on specific instructions (see Section 13.19.3 [Tagging Insns], page 251).

`(const_string` *`value`*`)`
: The string *value* specifies a constant attribute value. If *value* is specified as '`"*"`', it means that the default value of the attribute is to be used for the insn containing this expression. '`"*"`' obviously cannot be used in the *default* expression of a `define_attr`.

  If the attribute whose value is being specified is numeric, *value* must be a string containing a non-negative integer (normally `const_int` would be used in this case). Otherwise, it must contain one of the valid values for the attribute.

`(if_then_else` *`test true-value false-value`*`)`
: *test* specifies an attribute test, whose format is defined below. The value of this expression is *true-value* if *test* is true, otherwise it is *false-value*.

`(cond [`*`test1 value1 ...`*`]` *`default`*`)`
: The first operand of this expression is a vector containing an even number of expressions and consisting of pairs of *test* and *value* expressions. The value of the `cond` expression is that of the *value* corresponding to the first true *test* expression. If none of the *test* expressions are true, the value of the `cond` expression is that of the *default* expression.

test expressions can have one of the following forms:

`(const_int i)`
: This test is true if *i* is nonzero and false otherwise.

`(not test)`
`(ior test1 test2)`
`(and test1 test2)`
: These tests are true if the indicated logical function is true.

`(match_operand:m n pred constraints)`
: This test is true if operand *n* of the insn whose attribute value is being determined has mode *m* (this part of the test is ignored if *m* is `VOIDmode`) and the function specified by the string *pred* returns a nonzero value when passed operand *n* and mode *m* (this part of the test is ignored if *pred* is the null string).

  The *constraints* operand is ignored and should be the null string.

`(le arith1 arith2)`
`(leu arith1 arith2)`
`(lt arith1 arith2)`
`(ltu arith1 arith2)`
`(gt arith1 arith2)`
`(gtu arith1 arith2)`
`(ge arith1 arith2)`
`(geu arith1 arith2)`
`(ne arith1 arith2)`
`(eq arith1 arith2)`
: These tests are true if the indicated comparison of the two arithmetic expressions is true. Arithmetic expressions are formed with `plus`, `minus`, `mult`, `div`, `mod`, `abs`, `neg`, `and`, `ior`, `xor`, `not`, `ashift`, `lshiftrt`, and `ashiftrt` expressions.

  `const_int` and `symbol_ref` are always valid terms (see Section 13.19.5 [Insn Lengths], page 254,for additional forms). `symbol_ref` is a string denoting a C expression that yields an `int` when evaluated by the '`get_attr_...`' routine. It should normally be a global variable.

`(eq_attr name value)`
: *name* is a string specifying the name of an attribute.

  value is a string that is either a valid value for attribute *name*, a comma-separated list of values, or '`!`' followed by a value or list. If *value* does not begin with a '`!`', this test is true if the value of the *name* attribute of the current insn is in the list specified by *value*. If *value* begins with a '`!`', this test is true if the attribute's value is *not* in the specified list.

  For example,

  ```
  (eq_attr "type" "load,store")
  ```

  is equivalent to

  ```
  (ior (eq_attr "type" "load") (eq_attr "type" "store"))
  ```

  If *name* specifies an attribute of '`alternative`', it refers to the value of the compiler variable `which_alternative` (see Section 13.6 [Output Statement], page 186) and the values must be small integers. For example,

```
(eq_attr "alternative" "2,3")
```

is equivalent to

```
(ior (eq (symbol_ref "which_alternative") (const_int 2))
     (eq (symbol_ref "which_alternative") (const_int 3)))
```

Note that, for most attributes, an `eq_attr` test is simplified in cases where the value of the attribute being tested is known for all insns matching a particular pattern. This is by far the most common case.

`(attr_flag` *name*`)`

The value of an `attr_flag` expression is true if the flag specified by *name* is true for the `insn` currently being scheduled.

name is a string specifying one of a fixed set of flags to test. Test the flags `forward` and `backward` to determine the direction of a conditional branch. Test the flags `very_likely`, `likely`, `very_unlikely`, and `unlikely` to determine if a conditional branch is expected to be taken.

If the `very_likely` flag is true, then the `likely` flag is also true. Likewise for the `very_unlikely` and `unlikely` flags.

This example describes a conditional branch delay slot which can be nullified for forward branches that are taken (annul-true) or for backward branches which are not taken (annul-false).

```
(define_delay (eq_attr "type" "cbranch")
  [(eq_attr "in_branch_delay" "true")
   (and (eq_attr "in_branch_delay" "true")
        (attr_flag "forward"))
   (and (eq_attr "in_branch_delay" "true")
        (attr_flag "backward"))])
```

The `forward` and `backward` flags are false if the current `insn` being scheduled is not a conditional branch.

The `very_likely` and `likely` flags are true if the `insn` being scheduled is not a conditional branch. The `very_unlikely` and `unlikely` flags are false if the `insn` being scheduled is not a conditional branch.

`attr_flag` is only used during delay slot scheduling and has no meaning to other passes of the compiler.

`(attr` *name*`)`

The value of another attribute is returned. This is most useful for numeric attributes, as `eq_attr` and `attr_flag` produce more efficient code for non-numeric attributes.

13.19.3 Assigning Attribute Values to Insns

The value assigned to an attribute of an insn is primarily determined by which pattern is matched by that insn (or which `define_peephole` generated it). Every `define_insn` and `define_peephole` can have an optional last argument to specify the values of attributes for matching insns. The value of any attribute not specified in a particular insn is set to the default value for that attribute, as specified in its `define_attr`. Extensive use of default values for attributes permits the specification of the values for only one or two attributes in the definition of most insn patterns, as seen in the example in the next section.

The optional last argument of `define_insn` and `define_peephole` is a vector of expressions, each of which defines the value for a single attribute. The most general way of assigning an attribute's value is to use a `set` expression whose first operand is an `attr` expression giving the name of the attribute being set. The second operand of the `set` is an attribute expression (see Section 13.19.2 [Expressions], page 249) giving the value of the attribute.

When the attribute value depends on the '`alternative`' attribute (i.e., which is the applicable alternative in the constraint of the insn), the `set_attr_alternative` expression can be used. It allows the specification of a vector of attribute expressions, one for each alternative.

When the generality of arbitrary attribute expressions is not required, the simpler `set_attr` expression can be used, which allows specifying a string giving either a single attribute value or a list of attribute values, one for each alternative.

The form of each of the above specifications is shown below. In each case, *name* is a string specifying the attribute to be set.

`(set_attr` *name value-string*`)`
: *value-string* is either a string giving the desired attribute value, or a string containing a comma-separated list giving the values for succeeding alternatives. The number of elements must match the number of alternatives in the constraint of the insn pattern.

  Note that it may be useful to specify '`*`' for some alternative, in which case the attribute will assume its default value for insns matching that alternative.

`(set_attr_alternative` *name* `[`*value1 value2* `...])`
: Depending on the alternative of the insn, the value will be one of the specified values. This is a shorthand for using a `cond` with tests on the '`alternative`' attribute.

`(set (attr` *name*`)` *value*`)`
: The first operand of this `set` must be the special RTL expression `attr`, whose sole operand is a string giving the name of the attribute being set. *value* is the value of the attribute.

The following shows three different ways of representing the same attribute value specification:

```
(set_attr "type" "load,store,arith")

(set_attr_alternative "type"
                      [(const_string "load") (const_string "store")
                       (const_string "arith")])

(set (attr "type")
     (cond [(eq_attr "alternative" "1") (const_string "load")
            (eq_attr "alternative" "2") (const_string "store")]
           (const_string "arith")))
```

The `define_asm_attributes` expression provides a mechanism to specify the attributes assigned to insns produced from an `asm` statement. It has the form:

```
(define_asm_attributes [attr-sets])
```

where *attr-sets* is specified the same as for both the `define_insn` and the `define_peephole` expressions.

These values will typically be the "worst case" attribute values. For example, they might indicate that the condition code will be clobbered.

A specification for a `length` attribute is handled specially. The way to compute the length of an `asm` insn is to multiply the length specified in the expression `define_asm_attributes` by the number of machine instructions specified in the `asm` statement, determined by counting the number of semicolons and newlines in the string. Therefore, the value of the `length` attribute specified in a `define_asm_attributes` should be the maximum possible length of a single machine instruction.

13.19.4 Example of Attribute Specifications

The judicious use of defaulting is important in the efficient use of insn attributes. Typically, insns are divided into *types* and an attribute, customarily called `type`, is used to represent this value. This attribute is normally used only to define the default value for other attributes. An example will clarify this usage.

Assume we have a RISC machine with a condition code and in which only full-word operations are performed in registers. Let us assume that we can divide all insns into loads, stores, (integer) arithmetic operations, floating point operations, and branches.

Here we will concern ourselves with determining the effect of an insn on the condition code and will limit ourselves to the following possible effects: The condition code can be set unpredictably (clobbered), not be changed, be set to agree with the results of the operation, or only changed if the item previously set into the condition code has been modified.

Here is part of a sample '`md`' file for such a machine:

```
(define_attr "type" "load,store,arith,fp,branch" (const_string "arith"))

(define_attr "cc" "clobber,unchanged,set,change0"
             (cond [(eq_attr "type" "load")
                        (const_string "change0")
                    (eq_attr "type" "store,branch")
                        (const_string "unchanged")
                    (eq_attr "type" "arith")
                        (if_then_else (match_operand:SI 0 "" "")
                                      (const_string "set")
                                      (const_string "clobber"))]
                   (const_string "clobber")))

(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r,r,m")
        (match_operand:SI 1 "general_operand" "r,m,r"))]
  ""
  "@
   move %0,%1
   load %0,%1
   store %0,%1"
  [(set_attr "type" "arith,load,store")])
```

Note that we assume in the above example that arithmetic operations performed on quantities smaller than a machine word clobber the condition code since they will set the condition code to a value corresponding to the full-word result.

13.19.5 Computing the Length of an Insn

For many machines, multiple types of branch instructions are provided, each for different length branch displacements. In most cases, the assembler will choose the correct instruction to use. However, when the assembler cannot do so, GCC can when a special attribute, the `length` attribute, is defined. This attribute must be defined to have numeric values by specifying a null string in its `define_attr`.

In the case of the `length` attribute, two additional forms of arithmetic terms are allowed in test expressions:

`(match_dup` *n*`)`
: This refers to the address of operand *n* of the current insn, which must be a `label_ref`.

`(pc)`
: This refers to the address of the *current* insn. It might have been more consistent with other usage to make this the address of the *next* insn but this would be confusing because the length of the current insn is to be computed.

For normal insns, the length will be determined by value of the `length` attribute. In the case of `addr_vec` and `addr_diff_vec` insn patterns, the length is computed as the number of vectors multiplied by the size of each vector.

Lengths are measured in addressable storage units (bytes).

The following macros can be used to refine the length computation:

`ADJUST_INSN_LENGTH (`*insn*`,` *length*`)`
: If defined, modifies the length assigned to instruction *insn* as a function of the context in which it is used. *length* is an lvalue that contains the initially computed length of the insn and should be updated with the correct length of the insn.

  This macro will normally not be required. A case in which it is required is the ROMP. On this machine, the size of an `addr_vec` insn must be increased by two to compensate for the fact that alignment may be required.

The routine that returns `get_attr_length` (the value of the `length` attribute) can be used by the output routine to determine the form of the branch instruction to be written, as the example below illustrates.

As an example of the specification of variable-length branches, consider the IBM 360. If we adopt the convention that a register will be set to the starting address of a function, we can jump to labels within 4k of the start using a four-byte instruction. Otherwise, we need a six-byte sequence to load the address from memory and then branch to it.

On such a machine, a pattern for a branch instruction might be specified as follows:

```
(define_insn "jump"
  [(set (pc)
        (label_ref (match_operand 0 "" "")))]
  ""
{
```

```
   return (get_attr_length (insn) == 4
           ? "b %l0" : "l r15,=a(%l0); br r15");
}
  [(set (attr "length")
        (if_then_else (lt (match_dup 0) (const_int 4096))
                      (const_int 4)
                      (const_int 6)))])
```

13.19.6 Constant Attributes

A special form of `define_attr`, where the expression for the default value is a `const` expression, indicates an attribute that is constant for a given run of the compiler. Constant attributes may be used to specify which variety of processor is used. For example,

```
(define_attr "cpu" "m88100,m88110,m88000"
 (const
  (cond [(symbol_ref "TARGET_88100") (const_string "m88100")
         (symbol_ref "TARGET_88110") (const_string "m88110")]
        (const_string "m88000"))))

(define_attr "memory" "fast,slow"
 (const
  (if_then_else (symbol_ref "TARGET_FAST_MEM")
                (const_string "fast")
                (const_string "slow"))))
```

The routine generated for constant attributes has no parameters as it does not depend on any particular insn. RTL expressions used to define the value of a constant attribute may use the `symbol_ref` form, but may not use either the `match_operand` form or `eq_attr` forms involving insn attributes.

13.19.7 Delay Slot Scheduling

The insn attribute mechanism can be used to specify the requirements for delay slots, if any, on a target machine. An instruction is said to require a *delay slot* if some instructions that are physically after the instruction are executed as if they were located before it. Classic examples are branch and call instructions, which often execute the following instruction before the branch or call is performed.

On some machines, conditional branch instructions can optionally *annul* instructions in the delay slot. This means that the instruction will not be executed for certain branch outcomes. Both instructions that annul if the branch is true and instructions that annul if the branch is false are supported.

Delay slot scheduling differs from instruction scheduling in that determining whether an instruction needs a delay slot is dependent only on the type of instruction being generated, not on data flow between the instructions. See the next section for a discussion of data-dependent instruction scheduling.

The requirement of an insn needing one or more delay slots is indicated via the `define_delay` expression. It has the following form:

```
(define_delay test
              [delay-1 annul-true-1 annul-false-1
               delay-2 annul-true-2 annul-false-2
               ...])
```

test is an attribute test that indicates whether this `define_delay` applies to a particular insn. If so, the number of required delay slots is determined by the length of the vector

specified as the second argument. An insn placed in delay slot *n* must satisfy attribute test *delay-n*. *annul-true-n* is an attribute test that specifies which insns may be annulled if the branch is true. Similarly, *annul-false-n* specifies which insns in the delay slot may be annulled if the branch is false. If annulling is not supported for that delay slot, `(nil)` should be coded.

For example, in the common case where branch and call insns require a single delay slot, which may contain any insn other than a branch or call, the following would be placed in the 'md' file:

```
(define_delay (eq_attr "type" "branch,call")
              [(eq_attr "type" "!branch,call") (nil) (nil)])
```

Multiple `define_delay` expressions may be specified. In this case, each such expression specifies different delay slot requirements and there must be no insn for which tests in two `define_delay` expressions are both true.

For example, if we have a machine that requires one delay slot for branches but two for calls, no delay slot can contain a branch or call insn, and any valid insn in the delay slot for the branch can be annulled if the branch is true, we might represent this as follows:

```
(define_delay (eq_attr "type" "branch")
   [(eq_attr "type" "!branch,call")
    (eq_attr "type" "!branch,call")
    (nil)])

(define_delay (eq_attr "type" "call")
              [(eq_attr "type" "!branch,call") (nil) (nil)
               (eq_attr "type" "!branch,call") (nil) (nil)])
```

13.19.8 Specifying processor pipeline description

To achieve better performance, most modern processors (super-pipelined, superscalar RISC, and VLIW processors) have many *functional units* on which several instructions can be executed simultaneously. An instruction starts execution if its issue conditions are satisfied. If not, the instruction is stalled until its conditions are satisfied. Such *interlock (pipeline) delay* causes interruption of the fetching of successor instructions (or demands nop instructions, e.g. for some MIPS processors).

There are two major kinds of interlock delays in modern processors. The first one is a data dependence delay determining *instruction latency time*. The instruction execution is not started until all source data have been evaluated by prior instructions (there are more complex cases when the instruction execution starts even when the data are not available but will be ready in given time after the instruction execution start). Taking the data dependence delays into account is simple. The data dependence (true, output, and anti-dependence) delay between two instructions is given by a constant. In most cases this approach is adequate. The second kind of interlock delays is a reservation delay. The reservation delay means that two instructions under execution will be in need of shared processors resources, i.e. buses, internal registers, and/or functional units, which are reserved for some time. Taking this kind of delay into account is complex especially for modern RISC processors.

The task of exploiting more processor parallelism is solved by an instruction scheduler. For a better solution to this problem, the instruction scheduler has to have an adequate description of the processor parallelism (or *pipeline description*). GCC machine descriptions

describe processor parallelism and functional unit reservations for groups of instructions with the aid of *regular expressions*.

The GCC instruction scheduler uses a *pipeline hazard recognizer* to figure out the possibility of the instruction issue by the processor on a given simulated processor cycle. The pipeline hazard recognizer is automatically generated from the processor pipeline description. The pipeline hazard recognizer generated from the machine description is based on a deterministic finite state automaton (DFA): the instruction issue is possible if there is a transition from one automaton state to another one. This algorithm is very fast, and furthermore, its speed is not dependent on processor complexity[1].

The rest of this section describes the directives that constitute an automaton-based processor pipeline description. The order of these constructions within the machine description file is not important.

The following optional construction describes names of automata generated and used for the pipeline hazards recognition. Sometimes the generated finite state automaton used by the pipeline hazard recognizer is large. If we use more than one automaton and bind functional units to the automata, the total size of the automata is usually less than the size of the single automaton. If there is no one such construction, only one finite state automaton is generated.

```
(define_automaton automata-names)
```

automata-names is a string giving names of the automata. The names are separated by commas. All the automata should have unique names. The automaton name is used in the constructions `define_cpu_unit` and `define_query_cpu_unit`.

Each processor functional unit used in the description of instruction reservations should be described by the following construction.

```
(define_cpu_unit unit-names [automaton-name])
```

unit-names is a string giving the names of the functional units separated by commas. Don't use name '`nothing`', it is reserved for other goals.

automaton-name is a string giving the name of the automaton with which the unit is bound. The automaton should be described in construction `define_automaton`. You should give *automaton-name*, if there is a defined automaton.

The assignment of units to automata are constrained by the uses of the units in insn reservations. The most important constraint is: if a unit reservation is present on a particular cycle of an alternative for an insn reservation, then some unit from the same automaton must be present on the same cycle for the other alternatives of the insn reservation. The rest of the constraints are mentioned in the description of the subsequent constructions.

The following construction describes CPU functional units analogously to `define_cpu_unit`. The reservation of such units can be queried for an automaton state. The instruction scheduler never queries reservation of functional units for given automaton state. So as a rule, you don't need this construction. This construction could be used for future code generation goals (e.g. to generate VLIW insn templates).

[1] However, the size of the automaton depends on processor complexity. To limit this effect, machine descriptions can split orthogonal parts of the machine description among several automata: but then, since each of these must be stepped independently, this does cause a small decrease in the algorithm's performance.

```
(define_query_cpu_unit unit-names [automaton-name])
```

unit-names is a string giving names of the functional units separated by commas.

automaton-name is a string giving the name of the automaton with which the unit is bound.

The following construction is the major one to describe pipeline characteristics of an instruction.

```
(define_insn_reservation insn-name default_latency
                         condition regexp)
```

default_latency is a number giving latency time of the instruction. There is an important difference between the old description and the automaton based pipeline description. The latency time is used for all dependencies when we use the old description. In the automaton based pipeline description, the given latency time is only used for true dependencies. The cost of anti-dependencies is always zero and the cost of output dependencies is the difference between latency times of the producing and consuming insns (if the difference is negative, the cost is considered to be zero). You can always change the default costs for any description by using the target hook `TARGET_SCHED_ADJUST_COST` (see Section 14.16 [Scheduling], page 345).

insn-name is a string giving the internal name of the insn. The internal names are used in constructions `define_bypass` and in the automaton description file generated for debugging. The internal name has nothing in common with the names in `define_insn`. It is a good practice to use insn classes described in the processor manual.

condition defines what RTL insns are described by this construction. You should remember that you will be in trouble if *condition* for two or more different `define_insn_reservation` constructions is TRUE for an insn. In this case what reservation will be used for the insn is not defined. Such cases are not checked during generation of the pipeline hazards recognizer because in general recognizing that two conditions may have the same value is quite difficult (especially if the conditions contain `symbol_ref`). It is also not checked during the pipeline hazard recognizer work because it would slow down the recognizer considerably.

regexp is a string describing the reservation of the cpu's functional units by the instruction. The reservations are described by a regular expression according to the following syntax:

```
regexp = regexp "," oneof
       | oneof

oneof = oneof "|" allof
      | allof

allof = allof "+" repeat
      | repeat

repeat = element "*" number
       | element

element = cpu_function_unit_name
        | reservation_name
        | result_name
        | "nothing"
        | "(" regexp ")"
```

- '`,`' is used for describing the start of the next cycle in the reservation.
- '`|`' is used for describing a reservation described by the first regular expression **or** a reservation described by the second regular expression **or** etc.
- '`+`' is used for describing a reservation described by the first regular expression **and** a reservation described by the second regular expression **and** etc.
- '`*`' is used for convenience and simply means a sequence in which the regular expression are repeated *number* times with cycle advancing (see '`,`').
- '`cpu_function_unit_name`' denotes reservation of the named functional unit.
- '`reservation_name`' — see description of construction '`define_reservation`'.
- '`nothing`' denotes no unit reservations.

Sometimes unit reservations for different insns contain common parts. In such case, you can simplify the pipeline description by describing the common part by the following construction

```
(define_reservation reservation-name regexp)
```

reservation-name is a string giving name of *regexp*. Functional unit names and reservation names are in the same name space. So the reservation names should be different from the functional unit names and can not be the reserved name '`nothing`'.

The following construction is used to describe exceptions in the latency time for given instruction pair. This is so called bypasses.

```
(define_bypass number out_insn_names in_insn_names
               [guard])
```

number defines when the result generated by the instructions given in string *out_insn_names* will be ready for the instructions given in string *in_insn_names*. The instructions in the string are separated by commas.

guard is an optional string giving the name of a C function which defines an additional guard for the bypass. The function will get the two insns as parameters. If the function returns zero the bypass will be ignored for this case. The additional guard is necessary to recognize complicated bypasses, e.g. when the consumer is only an address of insn '`store`' (not a stored value).

The following five constructions are usually used to describe VLIW processors, or more precisely, to describe a placement of small instructions into VLIW instruction slots. They can be used for RISC processors, too.

```
(exclusion_set unit-names unit-names)
(presence_set unit-names patterns)
(final_presence_set unit-names patterns)
(absence_set unit-names patterns)
(final_absence_set unit-names patterns)
```

unit-names is a string giving names of functional units separated by commas.

patterns is a string giving patterns of functional units separated by comma. Currently pattern is one unit or units separated by white-spaces.

The first construction ('`exclusion_set`') means that each functional unit in the first string can not be reserved simultaneously with a unit whose name is in the second string and vice versa. For example, the construction is useful for describing processors (e.g. some SPARC processors) with a fully pipelined floating point functional unit which can execute simultaneously only single floating point insns or only double floating point insns.

The second construction ('`presence_set`') means that each functional unit in the first string can not be reserved unless at least one of pattern of units whose names are in the second string is reserved. This is an asymmetric relation. For example, it is useful for description that VLIW '`slot1`' is reserved after '`slot0`' reservation. We could describe it by the following construction

```
(presence_set "slot1" "slot0")
```

Or '`slot1`' is reserved only after '`slot0`' and unit '`b0`' reservation. In this case we could write

```
(presence_set "slot1" "slot0 b0")
```

The third construction ('`final_presence_set`') is analogous to '`presence_set`'. The difference between them is when checking is done. When an instruction is issued in given automaton state reflecting all current and planned unit reservations, the automaton state is changed. The first state is a source state, the second one is a result state. Checking for '`presence_set`' is done on the source state reservation, checking for '`final_presence_set`' is done on the result reservation. This construction is useful to describe a reservation which is actually two subsequent reservations. For example, if we use

```
(presence_set "slot1" "slot0")
```

the following insn will be never issued (because '`slot1`' requires '`slot0`' which is absent in the source state).

```
(define_reservation "insn_and_nop" "slot0 + slot1")
```

but it can be issued if we use analogous '`final_presence_set`'.

The forth construction ('`absence_set`') means that each functional unit in the first string can be reserved only if each pattern of units whose names are in the second string is not reserved. This is an asymmetric relation (actually '`exclusion_set`' is analogous to this one but it is symmetric). For example, it is useful for description that VLIW '`slot0`' can not be reserved after '`slot1`' or '`slot2`' reservation. We could describe it by the following construction

```
(absence_set "slot2" "slot0, slot1")
```

Or '`slot2`' can not be reserved if '`slot0`' and unit '`b0`' are reserved or '`slot1`' and unit '`b1`' are reserved. In this case we could write

```
(absence_set "slot2" "slot0 b0, slot1 b1")
```

All functional units mentioned in a set should belong to the same automaton.

The last construction ('`final_absence_set`') is analogous to '`absence_set`' but checking is done on the result (state) reservation. See comments for '`final_presence_set`'.

You can control the generator of the pipeline hazard recognizer with the following construction.

```
(automata_option options)
```

options is a string giving options which affect the generated code. Currently there are the following options:

- *no-minimization* makes no minimization of the automaton. This is only worth to do when we are debugging the description and need to look more accurately at reservations of states.
- *time* means printing additional time statistics about generation of automata.

- *v* means a generation of the file describing the result automata. The file has suffix '`.dfa`' and can be used for the description verification and debugging.
- *w* means a generation of warning instead of error for non-critical errors.
- *ndfa* makes nondeterministic finite state automata. This affects the treatment of operator '`|`' in the regular expressions. The usual treatment of the operator is to try the first alternative and, if the reservation is not possible, the second alternative. The nondeterministic treatment means trying all alternatives, some of them may be rejected by reservations in the subsequent insns.
- *progress* means output of a progress bar showing how many states were generated so far for automaton being processed. This is useful during debugging a DFA description. If you see too many generated states, you could interrupt the generator of the pipeline hazard recognizer and try to figure out a reason for generation of the huge automaton.

As an example, consider a superscalar RISC machine which can issue three insns (two integer insns and one floating point insn) on the cycle but can finish only two insns. To describe this, we define the following functional units.

```
(define_cpu_unit "i0_pipeline, i1_pipeline, f_pipeline")
(define_cpu_unit "port0, port1")
```

All simple integer insns can be executed in any integer pipeline and their result is ready in two cycles. The simple integer insns are issued into the first pipeline unless it is reserved, otherwise they are issued into the second pipeline. Integer division and multiplication insns can be executed only in the second integer pipeline and their results are ready correspondingly in 8 and 4 cycles. The integer division is not pipelined, i.e. the subsequent integer division insn can not be issued until the current division insn finished. Floating point insns are fully pipelined and their results are ready in 3 cycles. Where the result of a floating point insn is used by an integer insn, an additional delay of one cycle is incurred. To describe all of this we could specify

```
(define_cpu_unit "div")

(define_insn_reservation "simple" 2 (eq_attr "type" "int")
                         "(i0_pipeline | i1_pipeline), (port0 | port1)")

(define_insn_reservation "mult" 4 (eq_attr "type" "mult")
                         "i1_pipeline, nothing*2, (port0 | port1)")

(define_insn_reservation "div" 8 (eq_attr "type" "div")
                         "i1_pipeline, div*7, div + (port0 | port1)")

(define_insn_reservation "float" 3 (eq_attr "type" "float")
                         "f_pipeline, nothing, (port0 | port1))

(define_bypass 4 "float" "simple,mult,div")
```

To simplify the description we could describe the following reservation

```
(define_reservation "finish" "port0|port1")
```

and use it in all `define_insn_reservation` as in the following construction

```
(define_insn_reservation "simple" 2 (eq_attr "type" "int")
                         "(i0_pipeline | i1_pipeline), finish")
```

13.20 Conditional Execution

A number of architectures provide for some form of conditional execution, or predication. The hallmark of this feature is the ability to nullify most of the instructions in the instruction set. When the instruction set is large and not entirely symmetric, it can be quite tedious to describe these forms directly in the '`.md`' file. An alternative is the `define_cond_exec` template.

```
(define_cond_exec
  [predicate-pattern]
  "condition"
  "output-template")
```

predicate-pattern is the condition that must be true for the insn to be executed at runtime and should match a relational operator. One can use `match_operator` to match several relational operators at once. Any `match_operand` operands must have no more than one alternative.

condition is a C expression that must be true for the generated pattern to match.

output-template is a string similar to the `define_insn` output template (see Section 13.5 [Output Template], page 184), except that the '`*`' and '`@`' special cases do not apply. This is only useful if the assembly text for the predicate is a simple prefix to the main insn. In order to handle the general case, there is a global variable `current_insn_predicate` that will contain the entire predicate if the current insn is predicated, and will otherwise be `NULL`.

When `define_cond_exec` is used, an implicit reference to the `predicable` instruction attribute is made. See Section 13.19 [Insn Attributes], page 248. This attribute must be boolean (i.e. have exactly two elements in its *list-of-values*). Further, it must not be used with complex expressions. That is, the default and all uses in the insns must be a simple constant, not dependent on the alternative or anything else.

For each `define_insn` for which the `predicable` attribute is true, a new `define_insn` pattern will be generated that matches a predicated version of the instruction. For example,

```
(define_insn "addsi"
  [(set (match_operand:SI 0 "register_operand" "r")
        (plus:SI (match_operand:SI 1 "register_operand" "r")
                 (match_operand:SI 2 "register_operand" "r")))]
  "test1"
  "add %2,%1,%0")

(define_cond_exec
  [(ne (match_operand:CC 0 "register_operand" "c")
       (const_int 0))]
  "test2"
  "(%0)")
```

generates a new pattern

```
(define_insn ""
  [(cond_exec
     (ne (match_operand:CC 3 "register_operand" "c") (const_int 0))
     (set (match_operand:SI 0 "register_operand" "r")
          (plus:SI (match_operand:SI 1 "register_operand" "r")
                   (match_operand:SI 2 "register_operand" "r"))))]
  "(test2) && (test1)"
  "(%3) add %2,%1,%0")
```

13.21 Constant Definitions

Using literal constants inside instruction patterns reduces legibility and can be a maintenance problem.

To overcome this problem, you may use the `define_constants` expression. It contains a vector of name-value pairs. From that point on, wherever any of the names appears in the MD file, it is as if the corresponding value had been written instead. You may use `define_constants` multiple times; each appearance adds more constants to the table. It is an error to redefine a constant with a different value.

To come back to the a29k load multiple example, instead of

```
(define_insn ""
  [(match_parallel 0 "load_multiple_operation"
     [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
           (match_operand:SI 2 "memory_operand" "m"))
      (use (reg:SI 179))
      (clobber (reg:SI 179))])]
  ""
  "loadm 0,0,%1,%2")
```

You could write:

```
(define_constants [
    (R_BP 177)
    (R_FC 178)
    (R_CR 179)
    (R_Q  180)
])

(define_insn ""
  [(match_parallel 0 "load_multiple_operation"
     [(set (match_operand:SI 1 "gpc_reg_operand" "=r")
           (match_operand:SI 2 "memory_operand" "m"))
      (use (reg:SI R_CR))
      (clobber (reg:SI R_CR))])]
  ""
  "loadm 0,0,%1,%2")
```

The constants that are defined with a define_constant are also output in the insn-codes.h header file as #defines.

13.22 Macros

Ports often need to define similar patterns for more than one machine mode or for more than one rtx code. GCC provides some simple macro facilities to make this process easier.

13.22.1 Mode Macros

Ports often need to define similar patterns for two or more different modes. For example:

- If a processor has hardware support for both single and double floating-point arithmetic, the `SFmode` patterns tend to be very similar to the `DFmode` ones.
- If a port uses `SImode` pointers in one configuration and `DImode` pointers in another, it will usually have very similar `SImode` and `DImode` patterns for manipulating pointers.

Mode macros allow several patterns to be instantiated from one '`.md`' file template. They can be used with any type of rtx-based construct, such as a `define_insn`, `define_split`, or `define_peephole2`.

13.22.1.1 Defining Mode Macros

The syntax for defining a mode macro is:

```
(define_mode_macro name [(mode1 "cond1") ... (moden "condn")])
```

This allows subsequent '`.md`' file constructs to use the mode suffix `:name`. Every construct that does so will be expanded *n* times, once with every use of `:name` replaced by `:mode1`, once with every use replaced by `:mode2`, and so on. In the expansion for a particular *modei*, every C condition will also require that *condi* be true.

For example:

```
(define_mode_macro P [(SI "Pmode == SImode") (DI "Pmode == DImode")])
```

defines a new mode suffix `:P`. Every construct that uses `:P` will be expanded twice, once with every `:P` replaced by `:SI` and once with every `:P` replaced by `:DI`. The `:SI` version will only apply if `Pmode == SImode` and the `:DI` version will only apply if `Pmode == DImode`.

As with other '`.md`' conditions, an empty string is treated as "always true". `(mode "")` can also be abbreviated to `mode`. For example:

```
(define_mode_macro GPR [SI (DI "TARGET_64BIT")])
```

means that the `:DI` expansion only applies if `TARGET_64BIT` but that the `:SI` expansion has no such constraint.

Macros are applied in the order they are defined. This can be significant if two macros are used in a construct that requires substitutions. See Section 13.22.1.2 [Substitutions], page 264.

13.22.1.2 Substitution in Mode Macros

If an '`.md`' file construct uses mode macros, each version of the construct will often need slightly different strings or modes. For example:

- When a `define_expand` defines several `addm3` patterns (see Section 13.9 [Standard Names], page 212), each expander will need to use the appropriate mode name for *m*.
- When a `define_insn` defines several instruction patterns, each instruction will often use a different assembler mnemonic.
- When a `define_insn` requires operands with different modes, using a macro for one of the operand modes usually requires a specific mode for the other operand(s).

GCC supports such variations through a system of "mode attributes". There are two standard attributes: `mode`, which is the name of the mode in lower case, and `MODE`, which is the same thing in upper case. You can define other attributes using:

```
(define_mode_attr name [(mode1 "value1") ... (moden "valuen")])
```

where *name* is the name of the attribute and *valuei* is the value associated with *modei*.

When GCC replaces some *:macro* with *:mode*, it will scan each string and mode in the pattern for sequences of the form `<macro:attr>`, where *attr* is the name of a mode attribute. If the attribute is defined for *mode*, the whole `<...>` sequence will be replaced by the appropriate attribute value.

For example, suppose an '`.md`' file has:

```
(define_mode_macro P [(SI "Pmode == SImode") (DI "Pmode == DImode")])
(define_mode_attr load [(SI "lw") (DI "ld")])
```

If one of the patterns that uses `:P` contains the string `"<P:load>\t%0,%1"`, the `SI` version of that pattern will use `"lw\t%0,%1"` and the `DI` version will use `"ld\t%0,%1"`.

Here is an example of using an attribute for a mode:

```
(define_mode_macro LONG [SI DI])
(define_mode_attr SHORT [(SI "HI") (DI "SI")])
(define_insn ...
  (sign_extend:LONG (match_operand:<LONG:SHORT> ...)) ...)
```

The *macro*`:` prefix may be omitted, in which case the substitution will be attempted for every macro expansion.

13.22.1.3 Mode Macro Examples

Here is an example from the MIPS port. It defines the following modes and attributes (among others):

```
(define_mode_macro GPR [SI (DI "TARGET_64BIT")])
(define_mode_attr d [(SI "") (DI "d")])
```

and uses the following template to define both `subsi3` and `subdi3`:

```
(define_insn "sub<mode>3"
  [(set (match_operand:GPR 0 "register_operand" "=d")
        (minus:GPR (match_operand:GPR 1 "register_operand" "d")
                   (match_operand:GPR 2 "register_operand" "d")))]
  ""
  "<d>subu\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "<MODE>")])
```

This is exactly equivalent to:

```
(define_insn "subsi3"
  [(set (match_operand:SI 0 "register_operand" "=d")
        (minus:SI (match_operand:SI 1 "register_operand" "d")
                  (match_operand:SI 2 "register_operand" "d")))]
  ""
  "subu\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")])

(define_insn "subdi3"
  [(set (match_operand:DI 0 "register_operand" "=d")
        (minus:DI (match_operand:DI 1 "register_operand" "d")
                  (match_operand:DI 2 "register_operand" "d")))]
  ""
  "dsubu\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "DI")])
```

13.22.2 Code Macros

Code macros operate in a similar way to mode macros. See Section 13.22.1 [Mode Macros], page 263.

The construct:

```
(define_code_macro name [(code1 "cond1") ... (coden "condn")])
```

defines a pseudo rtx code *name* that can be instantiated as *codei* if condition *condi* is true. Each *codei* must have the same rtx format. See Section 11.2 [RTL Classes], page 122.

As with mode macros, each pattern that uses *name* will be expanded *n* times, once with all uses of *name* replaced by *code1*, once with all uses replaced by *code2*, and so on. See Section 13.22.1.1 [Defining Mode Macros], page 264.

It is possible to define attributes for codes as well as for modes. There are two standard code attributes: `code`, the name of the code in lower case, and `CODE`, the name of the code in upper case. Other attributes are defined using:

```
(define_code_attr name [(code1 "value1") ... (coden "valuen")])
```

Here's an example of code macros in action, taken from the MIPS port:

```
(define_code_macro any_cond [unordered ordered unlt unge uneq ltgt unle ungt
                             eq ne gt ge lt le gtu geu ltu leu])

(define_expand "b<code>"
  [(set (pc)
        (if_then_else (any_cond:CC (cc0)
                                   (const_int 0))
                      (label_ref (match_operand 0 ""))
                      (pc)))]
  ""
{
  gen_conditional_branch (operands, <CODE>);
  DONE;
})
```

This is equivalent to:

```
(define_expand "bunordered"
  [(set (pc)
        (if_then_else (unordered:CC (cc0)
                                    (const_int 0))
                      (label_ref (match_operand 0 ""))
                      (pc)))]
  ""
{
  gen_conditional_branch (operands, UNORDERED);
  DONE;
})

(define_expand "bordered"
  [(set (pc)
        (if_then_else (ordered:CC (cc0)
                                  (const_int 0))
                      (label_ref (match_operand 0 ""))
                      (pc)))]
  ""
{
  gen_conditional_branch (operands, ORDERED);
  DONE;
})

...
```

14 Target Description Macros and Functions

In addition to the file '*machine*.md', a machine description includes a C header file conventionally given the name '*machine*.h' and a C source file named '*machine*.c'. The header file defines numerous macros that convey the information about the target machine that does not fit into the scheme of the '.md' file. The file 'tm.h' should be a link to '*machine*.h'. The header file 'config.h' includes 'tm.h' and most compiler source files include 'config.h'. The source file defines a variable targetm, which is a structure containing pointers to functions and data relating to the target machine. '*machine*.c' should also contain their definitions, if they are not defined elsewhere in GCC, and other functions called through the macros defined in the '.h' file.

14.1 The Global targetm Variable

struct gcc_target targetm [Variable]

The target '.c' file must define the global targetm variable which contains pointers to functions and data relating to the target machine. The variable is declared in 'target.h'; 'target-def.h' defines the macro TARGET_INITIALIZER which is used to initialize the variable, and macros for the default initializers for elements of the structure. The '.c' file should override those macros for which the default definition is inappropriate. For example:

```
#include "target.h"
#include "target-def.h"

/* Initialize the GCC target structure.  */

#undef TARGET_COMP_TYPE_ATTRIBUTES
#define TARGET_COMP_TYPE_ATTRIBUTES machine_comp_type_attributes

struct gcc_target targetm = TARGET_INITIALIZER;
```

Where a macro should be defined in the '.c' file in this manner to form part of the targetm structure, it is documented below as a "Target Hook" with a prototype. Many macros will change in future from being defined in the '.h' file to being part of the targetm structure.

14.2 Controlling the Compilation Driver, 'gcc'

You can control the compilation driver.

SWITCH_TAKES_ARG (*char*) [Macro]

A C expression which determines whether the option '-*char*' takes arguments. The value should be the number of arguments that option takes–zero, for many options.

By default, this macro is defined as DEFAULT_SWITCH_TAKES_ARG, which handles the standard options properly. You need not define SWITCH_TAKES_ARG unless you wish to add additional options which take arguments. Any redefinition should call DEFAULT_SWITCH_TAKES_ARG and then check for additional options.

WORD_SWITCH_TAKES_ARG (*name*) [Macro]

A C expression which determines whether the option '-*name*' takes arguments. The value should be the number of arguments that option takes–zero, for many options. This macro rather than SWITCH_TAKES_ARG is used for multi-character option names.

By default, this macro is defined as `DEFAULT_WORD_SWITCH_TAKES_ARG`, which handles the standard options properly. You need not define `WORD_SWITCH_TAKES_ARG` unless you wish to add additional options which take arguments. Any redefinition should call `DEFAULT_WORD_SWITCH_TAKES_ARG` and then check for additional options.

`SWITCH_CURTAILS_COMPILATION (`*`char`*`)` [Macro]

A C expression which determines whether the option '`-`*`char`*' stops compilation before the generation of an executable. The value is boolean, nonzero if the option does stop an executable from being generated, zero otherwise.

By default, this macro is defined as `DEFAULT_SWITCH_CURTAILS_COMPILATION`, which handles the standard options properly. You need not define `SWITCH_CURTAILS_COMPILATION` unless you wish to add additional options which affect the generation of an executable. Any redefinition should call `DEFAULT_SWITCH_CURTAILS_COMPILATION` and then check for additional options.

`SWITCHES_NEED_SPACES` [Macro]

A string-valued C expression which enumerates the options for which the linker needs a space between the option and its argument.

If this macro is not defined, the default value is `""`.

`TARGET_OPTION_TRANSLATE_TABLE` [Macro]

If defined, a list of pairs of strings, the first of which is a potential command line target to the '`gcc`' driver program, and the second of which is a space-separated (tabs and other whitespace are not supported) list of options with which to replace the first option. The target defining this list is responsible for assuring that the results are valid. Replacement options may not be the `--opt` style, they must be the `-opt` style. It is the intention of this macro to provide a mechanism for substitution that affects the multilibs chosen, such as one option that enables many options, some of which select multilibs. Example nonsensical definition, where '`-malt-abi`', '`-EB`', and '`-mspoo`' cause different multilibs to be chosen:

```
#define TARGET_OPTION_TRANSLATE_TABLE \
{ "-fast",   "-march=fast-foo -malt-abi -I/usr/fast-foo" }, \
{ "-compat", "-EB -malign=4 -mspoo" }
```

`DRIVER_SELF_SPECS` [Macro]

A list of specs for the driver itself. It should be a suitable initializer for an array of strings, with no surrounding braces.

The driver applies these specs to its own command line between loading default '`specs`' files (but not command-line specified ones) and choosing the multilib directory or running any subcommands. It applies them in the order given, so each spec can depend on the options added by earlier ones. It is also possible to remove options using '`%<`*`option`*' in the usual way.

This macro can be useful when a port has several interdependent target options. It provides a way of standardizing the command line so that the other specs are easier to write.

Do not define this macro if it does not need to do anything.

`OPTION_DEFAULT_SPECS` [Macro]

A list of specs used to support configure-time default options (i.e. '`--with`' options) in the driver. It should be a suitable initializer for an array of structures, each containing two strings, without the outermost pair of surrounding braces.

The first item in the pair is the name of the default. This must match the code in '`config.gcc`' for the target. The second item is a spec to apply if a default with this name was specified. The string '`%(VALUE)`' in the spec will be replaced by the value of the default everywhere it occurs.

The driver will apply these specs to its own command line between loading default '`specs`' files and processing `DRIVER_SELF_SPECS`, using the same mechanism as `DRIVER_SELF_SPECS`.

Do not define this macro if it does not need to do anything.

`CPP_SPEC` [Macro]

A C string constant that tells the GCC driver program options to pass to CPP. It can also specify how to translate options you give to GCC into options for GCC to pass to the CPP.

Do not define this macro if it does not need to do anything.

`CPLUSPLUS_CPP_SPEC` [Macro]

This macro is just like `CPP_SPEC`, but is used for C++, rather than C. If you do not define this macro, then the value of `CPP_SPEC` (if any) will be used instead.

`CC1_SPEC` [Macro]

A C string constant that tells the GCC driver program options to pass to `cc1`, `cc1plus`, `f771`, and the other language front ends. It can also specify how to translate options you give to GCC into options for GCC to pass to front ends.

Do not define this macro if it does not need to do anything.

`CC1PLUS_SPEC` [Macro]

A C string constant that tells the GCC driver program options to pass to `cc1plus`. It can also specify how to translate options you give to GCC into options for GCC to pass to the `cc1plus`.

Do not define this macro if it does not need to do anything. Note that everything defined in CC1_SPEC is already passed to `cc1plus` so there is no need to duplicate the contents of CC1_SPEC in CC1PLUS_SPEC.

`ASM_SPEC` [Macro]

A C string constant that tells the GCC driver program options to pass to the assembler. It can also specify how to translate options you give to GCC into options for GCC to pass to the assembler. See the file '`sun3.h`' for an example of this.

Do not define this macro if it does not need to do anything.

`ASM_FINAL_SPEC` [Macro]

A C string constant that tells the GCC driver program how to run any programs which cleanup after the normal assembler. Normally, this is not needed. See the file '`mips.h`' for an example of this.

Do not define this macro if it does not need to do anything.

`AS_NEEDS_DASH_FOR_PIPED_INPUT` [Macro]

Define this macro, with no value, if the driver should give the assembler an argument consisting of a single dash, '`-`', to instruct it to read from its standard input (which will be a pipe connected to the output of the compiler proper). This argument is given after any '`-o`' option specifying the name of the output file.

If you do not define this macro, the assembler is assumed to read its standard input if given no non-option arguments. If your assembler cannot read standard input at all, use a '`%{pipe:%e}`' construct; see '`mips.h`' for instance.

`LINK_SPEC` [Macro]

A C string constant that tells the GCC driver program options to pass to the linker. It can also specify how to translate options you give to GCC into options for GCC to pass to the linker.

Do not define this macro if it does not need to do anything.

`LIB_SPEC` [Macro]

Another C string constant used much like `LINK_SPEC`. The difference between the two is that `LIB_SPEC` is used at the end of the command given to the linker.

If this macro is not defined, a default is provided that loads the standard C library from the usual place. See '`gcc.c`'.

`LIBGCC_SPEC` [Macro]

Another C string constant that tells the GCC driver program how and when to place a reference to '`libgcc.a`' into the linker command line. This constant is placed both before and after the value of `LIB_SPEC`.

If this macro is not defined, the GCC driver provides a default that passes the string '`-lgcc`' to the linker.

`REAL_LIBGCC_SPEC` [Macro]

By default, if `ENABLE_SHARED_LIBGCC` is defined, the `LIBGCC_SPEC` is not directly used by the driver program but is instead modified to refer to different versions of '`libgcc.a`' depending on the values of the command line flags '`-static`', '`-shared`', '`-static-libgcc`', and '`-shared-libgcc`'. On targets where these modifications are inappropriate, define `REAL_LIBGCC_SPEC` instead. `REAL_LIBGCC_SPEC` tells the driver how to place a reference to '`libgcc`' on the link command line, but, unlike `LIBGCC_SPEC`, it is used unmodified.

`USE_LD_AS_NEEDED` [Macro]

A macro that controls the modifications to `LIBGCC_SPEC` mentioned in `REAL_LIBGCC_SPEC`. If nonzero, a spec will be generated that uses –as-needed and the shared libgcc in place of the static exception handler library, when linking without any of `-static`, `-static-libgcc`, or `-shared-libgcc`.

`LINK_EH_SPEC` [Macro]

If defined, this C string constant is added to `LINK_SPEC`. When `USE_LD_AS_NEEDED` is zero or undefined, it also affects the modifications to `LIBGCC_SPEC` mentioned in `REAL_LIBGCC_SPEC`.

`STARTFILE_SPEC` [Macro]

Another C string constant used much like `LINK_SPEC`. The difference between the two is that `STARTFILE_SPEC` is used at the very beginning of the command given to the linker.

If this macro is not defined, a default is provided that loads the standard C startup file from the usual place. See '`gcc.c`'.

`ENDFILE_SPEC` [Macro]

Another C string constant used much like `LINK_SPEC`. The difference between the two is that `ENDFILE_SPEC` is used at the very end of the command given to the linker.

Do not define this macro if it does not need to do anything.

`THREAD_MODEL_SPEC` [Macro]

GCC `-v` will print the thread model GCC was configured to use. However, this doesn't work on platforms that are multilibbed on thread models, such as AIX 4.3. On such platforms, define `THREAD_MODEL_SPEC` such that it evaluates to a string without blanks that names one of the recognized thread models. `%*`, the default value of this macro, will expand to the value of `thread_file` set in '`config.gcc`'.

`SYSROOT_SUFFIX_SPEC` [Macro]

Define this macro to add a suffix to the target sysroot when GCC is configured with a sysroot. This will cause GCC to search for usr/lib, et al, within sysroot+suffix.

`SYSROOT_HEADERS_SUFFIX_SPEC` [Macro]

Define this macro to add a headers_suffix to the target sysroot when GCC is configured with a sysroot. This will cause GCC to pass the updated sysroot+headers_suffix to CPP, causing it to search for usr/include, et al, within sysroot+headers_suffix.

`EXTRA_SPECS` [Macro]

Define this macro to provide additional specifications to put in the '`specs`' file that can be used in various specifications like `CC1_SPEC`.

The definition should be an initializer for an array of structures, containing a string constant, that defines the specification name, and a string constant that provides the specification.

Do not define this macro if it does not need to do anything.

`EXTRA_SPECS` is useful when an architecture contains several related targets, which have various `..._SPECS` which are similar to each other, and the maintainer would like one central place to keep these definitions.

For example, the PowerPC System V.4 targets use `EXTRA_SPECS` to define either `_CALL_SYSV` when the System V calling sequence is used or `_CALL_AIX` when the older AIX-based calling sequence is used.

The '`config/rs6000/rs6000.h`' target file defines:

```
#define EXTRA_SPECS \
  { "cpp_sysv_default", CPP_SYSV_DEFAULT },

#define CPP_SYS_DEFAULT ""
```

The '`config/rs6000/sysv.h`' target file defines:

```
#undef CPP_SPEC
#define CPP_SPEC \
"%{posix: -D_POSIX_SOURCE } \
%{mcall-sysv: -D_CALL_SYSV } \
%{!mcall-sysv: %(cpp_sysv_default) } \
%{msoft-float: -D_SOFT_FLOAT} %{mcpu=403: -D_SOFT_FLOAT}"

#undef CPP_SYSV_DEFAULT
#define CPP_SYSV_DEFAULT "-D_CALL_SYSV"
```

while the 'config/rs6000/eabiaix.h' target file defines CPP_SYSV_DEFAULT as:

```
#undef CPP_SYSV_DEFAULT
#define CPP_SYSV_DEFAULT "-D_CALL_AIX"
```

LINK_LIBGCC_SPECIAL_1 [Macro]

Define this macro if the driver program should find the library 'libgcc.a'. If you do not define this macro, the driver program will pass the argument '-lgcc' to tell the linker to do the search.

LINK_GCC_C_SEQUENCE_SPEC [Macro]

The sequence in which libgcc and libc are specified to the linker. By default this is %G %L %G.

LINK_COMMAND_SPEC [Macro]

A C string constant giving the complete command line need to execute the linker. When you do this, you will need to update your port each time a change is made to the link command line within 'gcc.c'. Therefore, define this macro only if you need to completely redefine the command line for invoking the linker and there is no other way to accomplish the effect you need. Overriding this macro may be avoidable by overriding LINK_GCC_C_SEQUENCE_SPEC instead.

LINK_ELIMINATE_DUPLICATE_LDIRECTORIES [Macro]

A nonzero value causes collect2 to remove duplicate '-L*directory*' search directories from linking commands. Do not give it a nonzero value if removing duplicate search directories changes the linker's semantics.

MULTILIB_DEFAULTS [Macro]

Define this macro as a C expression for the initializer of an array of string to tell the driver program which options are defaults for this target and thus do not need to be handled specially when using MULTILIB_OPTIONS.

Do not define this macro if MULTILIB_OPTIONS is not defined in the target makefile fragment or if none of the options listed in MULTILIB_OPTIONS are set by default. See Section 16.1 [Target Fragment], page 409.

RELATIVE_PREFIX_NOT_LINKDIR [Macro]

Define this macro to tell gcc that it should only translate a '-B' prefix into a '-L' linker option if the prefix indicates an absolute file name.

MD_EXEC_PREFIX [Macro]

If defined, this macro is an additional prefix to try after STANDARD_EXEC_PREFIX. MD_EXEC_PREFIX is not searched when the '-b' option is used, or the compiler is built as a cross compiler. If you define MD_EXEC_PREFIX, then be sure to add it to the list of directories used to find the assembler in 'configure.in'.

`STANDARD_STARTFILE_PREFIX` [Macro]
Define this macro as a C string constant if you wish to override the standard choice of `libdir` as the default prefix to try when searching for startup files such as '`crt0.o`'. `STANDARD_STARTFILE_PREFIX` is not searched when the compiler is built as a cross compiler.

`STANDARD_STARTFILE_PREFIX_1` [Macro]
Define this macro as a C string constant if you wish to override the standard choice of `/lib` as a prefix to try after the default prefix when searching for startup files such as '`crt0.o`'. `STANDARD_STARTFILE_PREFIX_1` is not searched when the compiler is built as a cross compiler.

`STANDARD_STARTFILE_PREFIX_2` [Macro]
Define this macro as a C string constant if you wish to override the standard choice of `/lib` as yet another prefix to try after the default prefix when searching for startup files such as '`crt0.o`'. `STANDARD_STARTFILE_PREFIX_2` is not searched when the compiler is built as a cross compiler.

`MD_STARTFILE_PREFIX` [Macro]
If defined, this macro supplies an additional prefix to try after the standard prefixes. `MD_EXEC_PREFIX` is not searched when the '`-b`' option is used, or when the compiler is built as a cross compiler.

`MD_STARTFILE_PREFIX_1` [Macro]
If defined, this macro supplies yet another prefix to try after the standard prefixes. It is not searched when the '`-b`' option is used, or when the compiler is built as a cross compiler.

`INIT_ENVIRONMENT` [Macro]
Define this macro as a C string constant if you wish to set environment variables for programs called by the driver, such as the assembler and loader. The driver passes the value of this macro to `putenv` to initialize the necessary environment variables.

`LOCAL_INCLUDE_DIR` [Macro]
Define this macro as a C string constant if you wish to override the standard choice of '`/usr/local/include`' as the default prefix to try when searching for local header files. `LOCAL_INCLUDE_DIR` comes before `SYSTEM_INCLUDE_DIR` in the search order.

Cross compilers do not search either '`/usr/local/include`' or its replacement.

`MODIFY_TARGET_NAME` [Macro]
Define this macro if you wish to define command-line switches that modify the default target name.

For each switch, you can include a string to be appended to the first part of the configuration name or a string to be deleted from the configuration name, if present. The definition should be an initializer for an array of structures. Each array element should have three elements: the switch name (a string constant, including the initial dash), one of the enumeration codes `ADD` or `DELETE` to indicate whether the string should be inserted or deleted, and the string to be inserted or deleted (a string constant).

For example, on a machine where '64' at the end of the configuration name denotes a 64-bit target and you want the '-32' and '-64' switches to select between 32- and 64-bit targets, you would code

```
#define MODIFY_TARGET_NAME \
  { { "-32", DELETE, "64"}, \
    {"-64", ADD, "64"}}
```

SYSTEM_INCLUDE_DIR [Macro]

Define this macro as a C string constant if you wish to specify a system-specific directory to search for header files before the standard directory. SYSTEM_INCLUDE_DIR comes before STANDARD_INCLUDE_DIR in the search order.

Cross compilers do not use this macro and do not search the directory specified.

STANDARD_INCLUDE_DIR [Macro]

Define this macro as a C string constant if you wish to override the standard choice of '/usr/include' as the default prefix to try when searching for header files.

Cross compilers ignore this macro and do not search either '/usr/include' or its replacement.

STANDARD_INCLUDE_COMPONENT [Macro]

The "component" corresponding to STANDARD_INCLUDE_DIR. See INCLUDE_DEFAULTS, below, for the description of components. If you do not define this macro, no component is used.

INCLUDE_DEFAULTS [Macro]

Define this macro if you wish to override the entire default search path for include files. For a native compiler, the default search path usually consists of GCC_INCLUDE_DIR, LOCAL_INCLUDE_DIR, SYSTEM_INCLUDE_DIR, GPLUSPLUS_INCLUDE_DIR, and STANDARD_INCLUDE_DIR. In addition, GPLUSPLUS_INCLUDE_DIR and GCC_INCLUDE_DIR are defined automatically by 'Makefile', and specify private search areas for GCC. The directory GPLUSPLUS_INCLUDE_DIR is used only for C++ programs.

The definition should be an initializer for an array of structures. Each array element should have four elements: the directory name (a string constant), the component name (also a string constant), a flag for C++-only directories, and a flag showing that the includes in the directory don't need to be wrapped in `extern 'C'` when compiling C++. Mark the end of the array with a null element.

The component name denotes what GNU package the include file is part of, if any, in all uppercase letters. For example, it might be 'GCC' or 'BINUTILS'. If the package is part of a vendor-supplied operating system, code the component name as '0'.

For example, here is the definition used for VAX/VMS:

```
#define INCLUDE_DEFAULTS \
{                                       \
  { "GNU_GXX_INCLUDE:", "G++", 1, 1},   \
  { "GNU_CC_INCLUDE:", "GCC", 0, 0},    \
  { "SYS$SYSROOT:[SYSLIB.]", 0, 0, 0},  \
  { ".", 0, 0, 0},                      \
  { 0, 0, 0, 0}                         \
}
```

Here is the order of prefixes tried for exec files:

1. Any prefixes specified by the user with '`-B`'.
2. The environment variable `GCC_EXEC_PREFIX`, if any.
3. The directories specified by the environment variable `COMPILER_PATH`.
4. The macro `STANDARD_EXEC_PREFIX`.
5. '`/usr/lib/gcc/`'.
6. The macro `MD_EXEC_PREFIX`, if any.

Here is the order of prefixes tried for startfiles:

1. Any prefixes specified by the user with '`-B`'.
2. The environment variable `GCC_EXEC_PREFIX`, if any.
3. The directories specified by the environment variable `LIBRARY_PATH` (or port-specific name; native only, cross compilers do not use this).
4. The macro `STANDARD_EXEC_PREFIX`.
5. '`/usr/lib/gcc/`'.
6. The macro `MD_EXEC_PREFIX`, if any.
7. The macro `MD_STARTFILE_PREFIX`, if any.
8. The macro `STANDARD_STARTFILE_PREFIX`.
9. '`/lib/`'.
10. '`/usr/lib/`'.

14.3 Run-time Target Specification

Here are run-time target specifications.

`TARGET_CPU_CPP_BUILTINS ()` [Macro]

This function-like macro expands to a block of code that defines built-in preprocessor macros and assertions for the target cpu, using the functions `builtin_define`, `builtin_define_std` and `builtin_assert`. When the front end calls this macro it provides a trailing semicolon, and since it has finished command line option processing your code can use those results freely.

`builtin_assert` takes a string in the form you pass to the command-line option '`-A`', such as `cpu=mips`, and creates the assertion. `builtin_define` takes a string in the form accepted by option '`-D`' and unconditionally defines the macro.

`builtin_define_std` takes a string representing the name of an object-like macro. If it doesn't lie in the user's namespace, `builtin_define_std` defines it unconditionally. Otherwise, it defines a version with two leading underscores, and another version with two leading and trailing underscores, and defines the original only if an ISO standard was not requested on the command line. For example, passing `unix` defines `__unix`, `__unix__` and possibly `unix`; passing `_mips` defines `__mips`, `__mips__` and possibly `_mips`, and passing `_ABI64` defines only `_ABI64`.

You can also test for the C dialect being compiled. The variable `c_language` is set to one of `clk_c`, `clk_cplusplus` or `clk_objective_c`. Note that if we are preprocessing

assembler, this variable will be `clk_c` but the function-like macro `preprocessing_asm_p()` will return true, so you might want to check for that first. If you need to check for strict ANSI, the variable `flag_iso` can be used. The function-like macro `preprocessing_trad_p()` can be used to check for traditional preprocessing.

`TARGET_OS_CPP_BUILTINS ()` [Macro]
Similarly to `TARGET_CPU_CPP_BUILTINS` but this macro is optional and is used for the target operating system instead.

`TARGET_OBJFMT_CPP_BUILTINS ()` [Macro]
Similarly to `TARGET_CPU_CPP_BUILTINS` but this macro is optional and is used for the target object format. '`elfos.h`' uses this macro to define `__ELF__`, so you probably do not need to define it yourself.

`extern int target_flags` [Variable]
This variable is declared in '`options.h`', which is included before any target-specific headers.

`Target Hook int` *TARGET_DEFAULT_TARGET_FLAGS* [Variable]
This variable specifies the initial value of `target_flags`. Its default setting is 0.

`bool TARGET_HANDLE_OPTION (`*size_t code*, *const char *arg*, *int value*`)` [Target Hook]
This hook is called whenever the user specifies one of the target-specific options described by the '`.opt`' definition files (see Chapter 7 [Options], page 45). It has the opportunity to do some option-specific processing and should return true if the option is valid. The default definition does nothing but return true.

code specifies the `OPT_`*name* enumeration value associated with the selected option; *name* is just a rendering of the option name in which non-alphanumeric characters are replaced by underscores. *arg* specifies the string argument and is null if no argument was given. If the option is flagged as a `UInteger` (see Section 7.2 [Option properties], page 45), *value* is the numeric value of the argument. Otherwise *value* is 1 if the positive form of the option was used and 0 if the "no-" form was.

`TARGET_VERSION` [Macro]
This macro is a C statement to print on `stderr` a string describing the particular machine description choice. Every machine description should define `TARGET_VERSION`. For example:

```
#ifdef MOTOROLA
#define TARGET_VERSION \
  fprintf (stderr, " (68k, Motorola syntax)");
#else
#define TARGET_VERSION \
  fprintf (stderr, " (68k, MIT syntax)");
#endif
```

`OVERRIDE_OPTIONS` [Macro]
Sometimes certain combinations of command options do not make sense on a particular target machine. You can define a macro `OVERRIDE_OPTIONS` to take account of this. This macro, if defined, is executed once just after all the command options have been parsed.

Don't use this macro to turn on various extra optimizations for '`-O`'. That is what `OPTIMIZATION_OPTIONS` is for.

`OPTIMIZATION_OPTIONS (`*`level`*`, `*`size`*`)` [Macro]

Some machines may desire to change what optimizations are performed for various optimization levels. This macro, if defined, is executed once just after the optimization level is determined and before the remainder of the command options have been parsed. Values set in this macro are used as the default values for the other command line options.

level is the optimization level specified; 2 if '`-O2`' is specified, 1 if '`-O`' is specified, and 0 if neither is specified.

size is nonzero if '`-Os`' is specified and zero otherwise.

You should not use this macro to change options that are not machine-specific. These should uniformly selected by the same optimization level on all supported machines. Use this macro to enable machine-specific optimizations.

Do not examine `write_symbols` in this macro! The debugging options are not supposed to alter the generated code.

`CAN_DEBUG_WITHOUT_FP` [Macro]

Define this macro if debugging can be performed even without a frame pointer. If this macro is defined, GCC will turn on the '`-fomit-frame-pointer`' option whenever '`-O`' is specified.

14.4 Defining data structures for per-function information.

If the target needs to store information on a per-function basis, GCC provides a macro and a couple of variables to allow this. Note, just using statics to store the information is a bad idea, since GCC supports nested functions, so you can be halfway through encoding one function when another one comes along.

GCC defines a data structure called `struct function` which contains all of the data specific to an individual function. This structure contains a field called `machine` whose type is `struct machine_function *`, which can be used by targets to point to their own specific data.

If a target needs per-function specific data it should define the type `struct machine_function` and also the macro `INIT_EXPANDERS`. This macro should be used to initialize the function pointer `init_machine_status`. This pointer is explained below.

One typical use of per-function, target specific data is to create an RTX to hold the register containing the function's return address. This RTX can then be used to implement the `__builtin_return_address` function, for level 0.

Note—earlier implementations of GCC used a single data area to hold all of the per-function information. Thus when processing of a nested function began the old per-function data had to be pushed onto a stack, and when the processing was finished, it had to be popped off the stack. GCC used to provide function pointers called `save_machine_status` and `restore_machine_status` to handle the saving and restoring of the target specific information. Since the single data area approach is no longer used, these pointers are no longer supported.

`INIT_EXPANDERS` [Macro]
Macro called to initialize any target specific information. This macro is called once per function, before generation of any RTL has begun. The intention of this macro is to allow the initialization of the function pointer `init_machine_status`.

`void (*)(struct function *) init_machine_status` [Variable]
If this function pointer is non-`NULL` it will be called once per function, before function compilation starts, in order to allow the target to perform any target specific initialization of the `struct function` structure. It is intended that this would be used to initialize the `machine` of that structure.

`struct machine_function` structures are expected to be freed by GC. Generally, any memory that they reference must be allocated by using `ggc_alloc`, including the structure itself.

14.5 Storage Layout

Note that the definitions of the macros in this table which are sizes or alignments measured in bits do not need to be constant. They can be C expressions that refer to static variables, such as the `target_flags`. See Section 14.3 [Run-time Target], page 275.

`BITS_BIG_ENDIAN` [Macro]
Define this macro to have the value 1 if the most significant bit in a byte has the lowest number; otherwise define it to have the value zero. This means that bit-field instructions count from the most significant bit. If the machine has no bit-field instructions, then this must still be defined, but it doesn't matter which value it is defined to. This macro need not be a constant.

This macro does not affect the way structure fields are packed into bytes or words; that is controlled by `BYTES_BIG_ENDIAN`.

`BYTES_BIG_ENDIAN` [Macro]
Define this macro to have the value 1 if the most significant byte in a word has the lowest number. This macro need not be a constant.

`WORDS_BIG_ENDIAN` [Macro]
Define this macro to have the value 1 if, in a multiword object, the most significant word has the lowest number. This applies to both memory locations and registers; GCC fundamentally assumes that the order of words in memory is the same as the order in registers. This macro need not be a constant.

`LIBGCC2_WORDS_BIG_ENDIAN` [Macro]
Define this macro if `WORDS_BIG_ENDIAN` is not constant. This must be a constant value with the same meaning as `WORDS_BIG_ENDIAN`, which will be used only when compiling '`libgcc2.c`'. Typically the value will be set based on preprocessor defines.

`FLOAT_WORDS_BIG_ENDIAN` [Macro]
Define this macro to have the value 1 if `DFmode`, `XFmode` or `TFmode` floating point numbers are stored in memory with the word containing the sign bit at the lowest address; otherwise define it to have the value 0. This macro need not be a constant.

You need not define this macro if the ordering is the same as for multi-word integers.

`BITS_PER_UNIT` [Macro]
Define this macro to be the number of bits in an addressable storage unit (byte). If you do not define this macro the default is 8.

`BITS_PER_WORD` [Macro]
Number of bits in a word. If you do not define this macro, the default is `BITS_PER_UNIT * UNITS_PER_WORD`.

`MAX_BITS_PER_WORD` [Macro]
Maximum number of bits in a word. If this is undefined, the default is `BITS_PER_WORD`. Otherwise, it is the constant value that is the largest value that `BITS_PER_WORD` can have at run-time.

`UNITS_PER_WORD` [Macro]
Number of storage units in a word; normally the size of a general-purpose register, a power of two from 1 or 8.

`MIN_UNITS_PER_WORD` [Macro]
Minimum number of units in a word. If this is undefined, the default is `UNITS_PER_WORD`. Otherwise, it is the constant value that is the smallest value that `UNITS_PER_WORD` can have at run-time.

`UNITS_PER_SIMD_WORD` [Macro]
Number of units in the vectors that the vectorizer can produce. The default is equal to `UNITS_PER_WORD`, because the vectorizer can do some transformations even in absence of specialized SIMD hardware.

`POINTER_SIZE` [Macro]
Width of a pointer, in bits. You must specify a value no wider than the width of `Pmode`. If it is not equal to the width of `Pmode`, you must define `POINTERS_EXTEND_UNSIGNED`. If you do not specify a value the default is `BITS_PER_WORD`.

`POINTERS_EXTEND_UNSIGNED` [Macro]
A C expression whose value is greater than zero if pointers that need to be extended from being `POINTER_SIZE` bits wide to `Pmode` are to be zero-extended and zero if they are to be sign-extended. If the value is less then zero then there must be an "ptr_extend" instruction that extends a pointer from `POINTER_SIZE` to `Pmode`.

You need not define this macro if the `POINTER_SIZE` is equal to the width of `Pmode`.

`PROMOTE_MODE` (*m*, *unsignedp*, *type*) [Macro]
A macro to update *m* and *unsignedp* when an object whose type is *type* and which has the specified mode and signedness is to be stored in a register. This macro is only called when *type* is a scalar type.

On most RISC machines, which only have operations that operate on a full register, define this macro to set *m* to `word_mode` if *m* is an integer mode narrower than `BITS_PER_WORD`. In most cases, only integer modes should be widened because wider-precision floating-point operations are usually more expensive than their narrower counterparts.

For most machines, the macro definition does not change *unsignedp*. However, some machines, have instructions that preferentially handle either signed or unsigned quantities of certain modes. For example, on the DEC Alpha, 32-bit loads from memory and 32-bit add instructions sign-extend the result to 64 bits. On such machines, set *unsignedp* according to which kind of extension is more efficient.

Do not define this macro if it would never modify *m*.

`PROMOTE_FUNCTION_MODE` [Macro]

Like `PROMOTE_MODE`, but is applied to outgoing function arguments or function return values, as specified by `TARGET_PROMOTE_FUNCTION_ARGS` and `TARGET_PROMOTE_FUNCTION_RETURN`, respectively.

The default is `PROMOTE_MODE`.

`bool TARGET_PROMOTE_FUNCTION_ARGS (`*tree* ***fntype***`)` [Target Hook]

This target hook should return `true` if the promotion described by `PROMOTE_FUNCTION_MODE` should be done for outgoing function arguments.

`bool TARGET_PROMOTE_FUNCTION_RETURN (`*tree* ***fntype***`)` [Target Hook]

This target hook should return `true` if the promotion described by `PROMOTE_FUNCTION_MODE` should be done for the return value of functions.

If this target hook returns `true`, `FUNCTION_VALUE` must perform the same promotions done by `PROMOTE_FUNCTION_MODE`.

`PARM_BOUNDARY` [Macro]

Normal alignment required for function parameters on the stack, in bits. All stack parameters receive at least this much alignment regardless of data type. On most machines, this is the same as the size of an integer.

`STACK_BOUNDARY` [Macro]

Define this macro to the minimum alignment enforced by hardware for the stack pointer on this machine. The definition is a C expression for the desired alignment (measured in bits). This value is used as a default if `PREFERRED_STACK_BOUNDARY` is not defined. On most machines, this should be the same as `PARM_BOUNDARY`.

`PREFERRED_STACK_BOUNDARY` [Macro]

Define this macro if you wish to preserve a certain alignment for the stack pointer, greater than what the hardware enforces. The definition is a C expression for the desired alignment (measured in bits). This macro must evaluate to a value equal to or larger than `STACK_BOUNDARY`.

`FUNCTION_BOUNDARY` [Macro]

Alignment required for a function entry point, in bits.

`BIGGEST_ALIGNMENT` [Macro]

Biggest alignment that any data type can require on this machine, in bits.

`MINIMUM_ATOMIC_ALIGNMENT` [Macro]

If defined, the smallest alignment, in bits, that can be given to an object that can be referenced in one operation, without disturbing any nearby object. Normally, this is `BITS_PER_UNIT`, but may be larger on machines that don't have byte or half-word store operations.

`BIGGEST_FIELD_ALIGNMENT` [Macro]

Biggest alignment that any structure or union field can require on this machine, in bits. If defined, this overrides `BIGGEST_ALIGNMENT` for structure and union fields only, unless the field alignment has been set by the `__attribute__ ((aligned (n)))` construct.

`ADJUST_FIELD_ALIGN (field, computed)` [Macro]

An expression for the alignment of a structure field *field* if the alignment computed in the usual way (including applying of `BIGGEST_ALIGNMENT` and `BIGGEST_FIELD_ALIGNMENT` to the alignment) is *computed*. It overrides alignment only if the field alignment has not been set by the `__attribute__ ((aligned (n)))` construct.

`MAX_OFILE_ALIGNMENT` [Macro]

Biggest alignment supported by the object file format of this machine. Use this macro to limit the alignment which can be specified using the `__attribute__ ((aligned (n)))` construct. If not defined, the default value is `BIGGEST_ALIGNMENT`.

`DATA_ALIGNMENT (type, basic-align)` [Macro]

If defined, a C expression to compute the alignment for a variable in the static store. *type* is the data type, and *basic-align* is the alignment that the object would ordinarily have. The value of this macro is used instead of that alignment to align the object.

If this macro is not defined, then *basic-align* is used.

One use of this macro is to increase alignment of medium-size data to make it all fit in fewer cache lines. Another is to cause character arrays to be word-aligned so that `strcpy` calls that copy constants to character arrays can be done inline.

`CONSTANT_ALIGNMENT (constant, basic-align)` [Macro]

If defined, a C expression to compute the alignment given to a constant that is being placed in memory. *constant* is the constant and *basic-align* is the alignment that the object would ordinarily have. The value of this macro is used instead of that alignment to align the object.

If this macro is not defined, then *basic-align* is used.

The typical use of this macro is to increase alignment for string constants to be word aligned so that `strcpy` calls that copy constants can be done inline.

`LOCAL_ALIGNMENT (type, basic-align)` [Macro]

If defined, a C expression to compute the alignment for a variable in the local store. *type* is the data type, and *basic-align* is the alignment that the object would ordinarily have. The value of this macro is used instead of that alignment to align the object.

If this macro is not defined, then *basic-align* is used.

One use of this macro is to increase alignment of medium-size data to make it all fit in fewer cache lines.

`EMPTY_FIELD_BOUNDARY` [Macro]

Alignment in bits to be given to a structure bit-field that follows an empty field such as `int : 0;`.

If `PCC_BITFIELD_TYPE_MATTERS` is true, it overrides this macro.

`STRUCTURE_SIZE_BOUNDARY` [Macro]

Number of bits which any structure or union's size must be a multiple of. Each structure or union's size is rounded up to a multiple of this.

If you do not define this macro, the default is the same as `BITS_PER_UNIT`.

`STRICT_ALIGNMENT` [Macro]

Define this macro to be the value 1 if instructions will fail to work if given data not on the nominal alignment. If instructions will merely go slower in that case, define this macro as 0.

`PCC_BITFIELD_TYPE_MATTERS` [Macro]

Define this if you wish to imitate the way many other C compilers handle alignment of bit-fields and the structures that contain them.

The behavior is that the type written for a named bit-field (`int`, `short`, or other integer type) imposes an alignment for the entire structure, as if the structure really did contain an ordinary field of that type. In addition, the bit-field is placed within the structure so that it would fit within such a field, not crossing a boundary for it.

Thus, on most machines, a named bit-field whose type is written as `int` would not cross a four-byte boundary, and would force four-byte alignment for the whole structure. (The alignment used may not be four bytes; it is controlled by the other alignment parameters.)

An unnamed bit-field will not affect the alignment of the containing structure.

If the macro is defined, its definition should be a C expression; a nonzero value for the expression enables this behavior.

Note that if this macro is not defined, or its value is zero, some bit-fields may cross more than one alignment boundary. The compiler can support such references if there are '`insv`', '`extv`', and '`extzv`' insns that can directly reference memory.

The other known way of making bit-fields work is to define `STRUCTURE_SIZE_BOUNDARY` as large as `BIGGEST_ALIGNMENT`. Then every structure can be accessed with fullwords.

Unless the machine has bit-field instructions or you define `STRUCTURE_SIZE_BOUNDARY` that way, you must define `PCC_BITFIELD_TYPE_MATTERS` to have a nonzero value.

If your aim is to make GCC use the same conventions for laying out bit-fields as are used by another compiler, here is how to investigate what the other compiler does. Compile and run this program:

```
struct foo1
{
  char x;
  char :0;
  char y;
};

struct foo2
{
  char x;
  int :0;
  char y;
};
```

```
main ()
{
  printf ("Size of foo1 is %d\n",
          sizeof (struct foo1));
  printf ("Size of foo2 is %d\n",
          sizeof (struct foo2));
  exit (0);
}
```

If this prints 2 and 5, then the compiler's behavior is what you would get from `PCC_BITFIELD_TYPE_MATTERS`.

`BITFIELD_NBYTES_LIMITED` [Macro]

Like `PCC_BITFIELD_TYPE_MATTERS` except that its effect is limited to aligning a bitfield within the structure.

`bool TARGET_ALIGN_ANON_BITFIELDS (`*void*`)` [Target Hook]

When `PCC_BITFIELD_TYPE_MATTERS` is true this hook will determine whether unnamed bitfields affect the alignment of the containing structure. The hook should return true if the structure should inherit the alignment requirements of an unnamed bitfield's type.

`MEMBER_TYPE_FORCES_BLK (`*field*, *mode*`)` [Macro]

Return 1 if a structure or array containing *field* should be accessed using `BLKMODE`.

If *field* is the only field in the structure, *mode* is its mode, otherwise *mode* is VOIDmode. *mode* is provided in the case where structures of one field would require the structure's mode to retain the field's mode.

Normally, this is not needed. See the file '`c4x.h`' for an example of how to use this macro to prevent a structure having a floating point field from being accessed in an integer mode.

`ROUND_TYPE_ALIGN (`*type*, *computed*, *specified*`)` [Macro]

Define this macro as an expression for the alignment of a type (given by *type* as a tree node) if the alignment computed in the usual way is *computed* and the alignment explicitly specified was *specified*.

The default is to use *specified* if it is larger; otherwise, use the smaller of *computed* and `BIGGEST_ALIGNMENT`

`MAX_FIXED_MODE_SIZE` [Macro]

An integer expression for the size in bits of the largest integer machine mode that should actually be used. All integer machine modes of this size or smaller can be used for structures and unions with the appropriate sizes. If this macro is undefined, `GET_MODE_BITSIZE (DImode)` is assumed.

`STACK_SAVEAREA_MODE (`*save_level*`)` [Macro]

If defined, an expression of type `enum machine_mode` that specifies the mode of the save area operand of a `save_stack_`*level* named pattern (see Section 13.9 [Standard Names], page 212). *save_level* is one of `SAVE_BLOCK`, `SAVE_FUNCTION`, or `SAVE_NONLOCAL` and selects which of the three named patterns is having its mode specified.

You need not define this macro if it always returns `Pmode`. You would most commonly define this macro if the `save_stack_level` patterns need to support both a 32- and a 64-bit mode.

`STACK_SIZE_MODE` [Macro]

If defined, an expression of type `enum machine_mode` that specifies the mode of the size increment operand of an `allocate_stack` named pattern (see Section 13.9 [Standard Names], page 212).

You need not define this macro if it always returns `word_mode`. You would most commonly define this macro if the `allocate_stack` pattern needs to support both a 32- and a 64-bit mode.

`TARGET_FLOAT_FORMAT` [Macro]

A code distinguishing the floating point format of the target machine. There are four defined values:

`IEEE_FLOAT_FORMAT`
: This code indicates IEEE floating point. It is the default; there is no need to define `TARGET_FLOAT_FORMAT` when the format is IEEE.

`VAX_FLOAT_FORMAT`
: This code indicates the "F float" (for `float`) and "D float" or "G float" formats (for `double`) used on the VAX and PDP-11.

`IBM_FLOAT_FORMAT`
: This code indicates the format used on the IBM System/370.

`C4X_FLOAT_FORMAT`
: This code indicates the format used on the TMS320C3x/C4x.

If your target uses a floating point format other than these, you must define a new *name*_FLOAT_FORMAT code for it, and add support for it to '`real.c`'.

The ordering of the component words of floating point values stored in memory is controlled by `FLOAT_WORDS_BIG_ENDIAN`.

`MODE_HAS_NANS (`*mode*`)` [Macro]

When defined, this macro should be true if *mode* has a NaN representation. The compiler assumes that NaNs are not equal to anything (including themselves) and that addition, subtraction, multiplication and division all return NaNs when one operand is NaN.

By default, this macro is true if *mode* is a floating-point mode and the target floating-point format is IEEE.

`MODE_HAS_INFINITIES (`*mode*`)` [Macro]

This macro should be true if *mode* can represent infinity. At present, the compiler uses this macro to decide whether '`x - x`' is always defined. By default, the macro is true when *mode* is a floating-point mode and the target format is IEEE.

`MODE_HAS_SIGNED_ZEROS (`*mode*`)` [Macro]

True if *mode* distinguishes between positive and negative zero. The rules are expected to follow the IEEE standard:

- '`x + x`' has the same sign as '`x`'.
- If the sum of two values with opposite sign is zero, the result is positive for all rounding modes expect towards −infinity, for which it is negative.
- The sign of a product or quotient is negative when exactly one of the operands is negative.

The default definition is true if *mode* is a floating-point mode and the target format is IEEE.

`MODE_HAS_SIGN_DEPENDENT_ROUNDING (mode)` [Macro]

If defined, this macro should be true for *mode* if it has at least one rounding mode in which '`x`' and '`-x`' can be rounded to numbers of different magnitude. Two such modes are towards −infinity and towards +infinity.

The default definition of this macro is true if *mode* is a floating-point mode and the target format is IEEE.

`ROUND_TOWARDS_ZERO` [Macro]

If defined, this macro should be true if the prevailing rounding mode is towards zero. A true value has the following effects:

- `MODE_HAS_SIGN_DEPENDENT_ROUNDING` will be false for all modes.
- '`libgcc.a`''s floating-point emulator will round towards zero rather than towards nearest.
- The compiler's floating-point emulator will round towards zero after doing arithmetic, and when converting from the internal float format to the target format.

The macro does not affect the parsing of string literals. When the primary rounding mode is towards zero, library functions like `strtod` might still round towards nearest, and the compiler's parser should behave like the target's `strtod` where possible.

Not defining this macro is equivalent to returning zero.

`LARGEST_EXPONENT_IS_NORMAL (size)` [Macro]

This macro should return true if floats with *size* bits do not have a NaN or infinity representation, but use the largest exponent for normal numbers instead.

Defining this macro to true for *size* causes `MODE_HAS_NANS` and `MODE_HAS_INFINITIES` to be false for *size*-bit modes. It also affects the way '`libgcc.a`' and '`real.c`' emulate floating-point arithmetic.

The default definition of this macro returns false for all sizes.

`bool TARGET_VECTOR_OPAQUE_P (`*tree* *type*`)` [Target Hook]

This target hook should return `true` a vector is opaque. That is, if no cast is needed when copying a vector value of type *type* into another vector lvalue of the same size. Vector opaque types cannot be initialized. The default is that there are no such types.

`bool TARGET_MS_BITFIELD_LAYOUT_P (`*tree* *record_type*`)` [Target Hook]

This target hook returns `true` if bit-fields in the given *record_type* are to be laid out following the rules of Microsoft Visual C/C++, namely: (i) a bit-field won't share the same storage unit with the previous bit-field if their underlying types have different sizes, and the bit-field will be aligned to the highest alignment of the underlying types

of itself and of the previous bit-field; (ii) a zero-sized bit-field will affect the alignment of the whole enclosing structure, even if it is unnamed; except that (iii) a zero-sized bit-field will be disregarded unless it follows another bit-field of nonzero size. If this hook returns `true`, other macros that control bit-field layout are ignored.

When a bit-field is inserted into a packed record, the whole size of the underlying type is used by one or more same-size adjacent bit-fields (that is, if its long:3, 32 bits is used in the record, and any additional adjacent long bit-fields are packed into the same chunk of 32 bits. However, if the size changes, a new field of that size is allocated). In an unpacked record, this is the same as using alignment, but not equivalent when packing.

If both MS bit-fields and '`__attribute__((packed))`' are used, the latter will take precedence. If '`__attribute__((packed))`' is used on a single field when MS bit-fields are in use, it will take precedence for that field, but the alignment of the rest of the structure may affect its placement.

`const char * TARGET_MANGLE_FUNDAMENTAL_TYPE (`*tree* *type*`)` [Target Hook]

If your target defines any fundamental types, define this hook to return the appropriate encoding for these types as part of a C++ mangled name. The *type* argument is the tree structure representing the type to be mangled. The hook may be applied to trees which are not target-specific fundamental types; it should return `NULL` for all such types, as well as arguments it does not recognize. If the return value is not `NULL`, it must point to a statically-allocated string constant.

Target-specific fundamental types might be new fundamental types or qualified versions of ordinary fundamental types. Encode new fundamental types as '`u` *n* *name*', where *name* is the name used for the type in source code, and *n* is the length of *name* in decimal. Encode qualified versions of ordinary types as '`U` *n* *name* *code*', where *name* is the name used for the type qualifier in source code, *n* is the length of *name* as above, and *code* is the code used to represent the unqualified version of this type. (See `write_builtin_type` in '`cp/mangle.c`' for the list of codes.) In both cases the spaces are for clarity; do not include any spaces in your string.

The default version of this hook always returns `NULL`, which is appropriate for a target that does not define any new fundamental types.

14.6 Layout of Source Language Data Types

These macros define the sizes and other characteristics of the standard basic data types used in programs being compiled. Unlike the macros in the previous section, these apply to specific features of C and related languages, rather than to fundamental aspects of storage layout.

`INT_TYPE_SIZE` [Macro]

A C expression for the size in bits of the type `int` on the target machine. If you don't define this, the default is one word.

`SHORT_TYPE_SIZE` [Macro]

A C expression for the size in bits of the type `short` on the target machine. If you don't define this, the default is half a word. (If this would be less than one storage unit, it is rounded up to one unit.)

`LONG_TYPE_SIZE` [Macro]
A C expression for the size in bits of the type `long` on the target machine. If you don't define this, the default is one word.

`ADA_LONG_TYPE_SIZE` [Macro]
On some machines, the size used for the Ada equivalent of the type `long` by a native Ada compiler differs from that used by C. In that situation, define this macro to be a C expression to be used for the size of that type. If you don't define this, the default is the value of `LONG_TYPE_SIZE`.

`LONG_LONG_TYPE_SIZE` [Macro]
A C expression for the size in bits of the type `long long` on the target machine. If you don't define this, the default is two words. If you want to support GNU Ada on your machine, the value of this macro must be at least 64.

`CHAR_TYPE_SIZE` [Macro]
A C expression for the size in bits of the type `char` on the target machine. If you don't define this, the default is `BITS_PER_UNIT`.

`BOOL_TYPE_SIZE` [Macro]
A C expression for the size in bits of the C++ type `bool` and C99 type `_Bool` on the target machine. If you don't define this, and you probably shouldn't, the default is `CHAR_TYPE_SIZE`.

`FLOAT_TYPE_SIZE` [Macro]
A C expression for the size in bits of the type `float` on the target machine. If you don't define this, the default is one word.

`DOUBLE_TYPE_SIZE` [Macro]
A C expression for the size in bits of the type `double` on the target machine. If you don't define this, the default is two words.

`LONG_DOUBLE_TYPE_SIZE` [Macro]
A C expression for the size in bits of the type `long double` on the target machine. If you don't define this, the default is two words.

`LIBGCC2_LONG_DOUBLE_TYPE_SIZE` [Macro]
Define this macro if `LONG_DOUBLE_TYPE_SIZE` is not constant or if you want routines in '`libgcc2.a`' for a size other than `LONG_DOUBLE_TYPE_SIZE`. If you don't define this, the default is `LONG_DOUBLE_TYPE_SIZE`.

`LIBGCC2_HAS_DF_MODE` [Macro]
Define this macro if neither `LIBGCC2_DOUBLE_TYPE_SIZE` nor `LIBGCC2_LONG_DOUBLE_TYPE_SIZE` is `DFmode` but you want `DFmode` routines in '`libgcc2.a`' anyway. If you don't define this and either `LIBGCC2_DOUBLE_TYPE_SIZE` or `LIBGCC2_LONG_DOUBLE_TYPE_SIZE` is 64 then the default is 1, otherwise it is 0.

`LIBGCC2_HAS_XF_MODE` [Macro]
Define this macro if `LIBGCC2_LONG_DOUBLE_TYPE_SIZE` is not `XFmode` but you want `XFmode` routines in '`libgcc2.a`' anyway. If you don't define this and `LIBGCC2_LONG_DOUBLE_TYPE_SIZE` is 80 then the default is 1, otherwise it is 0.

`LIBGCC2_HAS_TF_MODE` [Macro]

Define this macro if `LIBGCC2_LONG_DOUBLE_TYPE_SIZE` is not `TFmode` but you want `TFmode` routines in '`libgcc2.a`' anyway. If you don't define this and `LIBGCC2_LONG_DOUBLE_TYPE_SIZE` is 128 then the default is 1, otherwise it is 0.

`TARGET_FLT_EVAL_METHOD` [Macro]

A C expression for the value for `FLT_EVAL_METHOD` in '`float.h`', assuming, if applicable, that the floating-point control word is in its default state. If you do not define this macro the value of `FLT_EVAL_METHOD` will be zero.

`WIDEST_HARDWARE_FP_SIZE` [Macro]

A C expression for the size in bits of the widest floating-point format supported by the hardware. If you define this macro, you must specify a value less than or equal to the value of `LONG_DOUBLE_TYPE_SIZE`. If you do not define this macro, the value of `LONG_DOUBLE_TYPE_SIZE` is the default.

`DEFAULT_SIGNED_CHAR` [Macro]

An expression whose value is 1 or 0, according to whether the type `char` should be signed or unsigned by default. The user can always override this default with the options '`-fsigned-char`' and '`-funsigned-char`'.

`bool TARGET_DEFAULT_SHORT_ENUMS (`*void*`)` [Target Hook]

This target hook should return true if the compiler should give an `enum` type only as many bytes as it takes to represent the range of possible values of that type. It should return false if all `enum` types should be allocated like `int`.

The default is to return false.

`SIZE_TYPE` [Macro]

A C expression for a string describing the name of the data type to use for size values. The typedef name `size_t` is defined using the contents of the string.

The string can contain more than one keyword. If so, separate them with spaces, and write first any length keyword, then `unsigned` if appropriate, and finally `int`. The string must exactly match one of the data type names defined in the function `init_decl_processing` in the file '`c-decl.c`'. You may not omit `int` or change the order—that would cause the compiler to crash on startup.

If you don't define this macro, the default is `"long unsigned int"`.

`PTRDIFF_TYPE` [Macro]

A C expression for a string describing the name of the data type to use for the result of subtracting two pointers. The typedef name `ptrdiff_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is `"long int"`.

`WCHAR_TYPE` [Macro]

A C expression for a string describing the name of the data type to use for wide characters. The typedef name `wchar_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is `"int"`.

`WCHAR_TYPE_SIZE` [Macro]

A C expression for the size in bits of the data type for wide characters. This is used in `cpp`, which cannot make use of `WCHAR_TYPE`.

`WINT_TYPE` [Macro]

A C expression for a string describing the name of the data type to use for wide characters passed to `printf` and returned from `getwc`. The typedef name `wint_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is `"unsigned int"`.

`INTMAX_TYPE` [Macro]

A C expression for a string describing the name of the data type that can represent any value of any standard or extended signed integer type. The typedef name `intmax_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is the first of `"int"`, `"long int"`, or `"long long int"` that has as much precision as `long long int`.

`UINTMAX_TYPE` [Macro]

A C expression for a string describing the name of the data type that can represent any value of any standard or extended unsigned integer type. The typedef name `uintmax_t` is defined using the contents of the string. See `SIZE_TYPE` above for more information.

If you don't define this macro, the default is the first of `"unsigned int"`, `"long unsigned int"`, or `"long long unsigned int"` that has as much precision as `long long unsigned int`.

`TARGET_PTRMEMFUNC_VBIT_LOCATION` [Macro]

The C++ compiler represents a pointer-to-member-function with a struct that looks like:

```
struct {
  union {
    void (*fn)();
    ptrdiff_t vtable_index;
  };
  ptrdiff_t delta;
};
```

The C++ compiler must use one bit to indicate whether the function that will be called through a pointer-to-member-function is virtual. Normally, we assume that the low-order bit of a function pointer must always be zero. Then, by ensuring that the vtable_index is odd, we can distinguish which variant of the union is in use. But, on some platforms function pointers can be odd, and so this doesn't work. In that case, we use the low-order bit of the `delta` field, and shift the remainder of the `delta` field to the left.

GCC will automatically make the right selection about where to store this bit using the `FUNCTION_BOUNDARY` setting for your platform. However, some platforms such as ARM/Thumb have `FUNCTION_BOUNDARY` set such that functions always start at even addresses, but the lowest bit of pointers to functions indicate whether the function at that address is in ARM or Thumb mode. If this is the case of your architecture, you should define this macro to `ptrmemfunc_vbit_in_delta`.

In general, you should not have to define this macro. On architectures in which function addresses are always even, according to `FUNCTION_BOUNDARY`, GCC will automatically define this macro to `ptrmemfunc_vbit_in_pfn`.

`TARGET_VTABLE_USES_DESCRIPTORS` [Macro]
Normally, the C++ compiler uses function pointers in vtables. This macro allows the target to change to use "function descriptors" instead. Function descriptors are found on targets for whom a function pointer is actually a small data structure. Normally the data structure consists of the actual code address plus a data pointer to which the function's data is relative.

If vtables are used, the value of this macro should be the number of words that the function descriptor occupies.

`TARGET_VTABLE_ENTRY_ALIGN` [Macro]
By default, the vtable entries are void pointers, the so the alignment is the same as pointer alignment. The value of this macro specifies the alignment of the vtable entry in bits. It should be defined only when special alignment is necessary. */

`TARGET_VTABLE_DATA_ENTRY_DISTANCE` [Macro]
There are a few non-descriptor entries in the vtable at offsets below zero. If these entries must be padded (say, to preserve the alignment specified by `TARGET_VTABLE_ENTRY_ALIGN`), set this to the number of words in each data entry.

14.7 Register Usage

This section explains how to describe what registers the target machine has, and how (in general) they can be used.

The description of which registers a specific instruction can use is done with register classes; see Section 14.8 [Register Classes], page 296. For information on using registers to access a stack frame, see Section 14.9.4 [Frame Registers], page 310. For passing values in registers, see Section 14.9.7 [Register Arguments], page 316. For returning values in registers, see Section 14.9.8 [Scalar Return], page 320.

14.7.1 Basic Characteristics of Registers

Registers have various characteristics.

`FIRST_PSEUDO_REGISTER` [Macro]
Number of hardware registers known to the compiler. They receive numbers 0 through `FIRST_PSEUDO_REGISTER-1`; thus, the first pseudo register's number really is assigned the number `FIRST_PSEUDO_REGISTER`.

`FIXED_REGISTERS` [Macro]
An initializer that says which registers are used for fixed purposes all throughout the compiled code and are therefore not available for general allocation. These would include the stack pointer, the frame pointer (except on machines where that can be used as a general register when no frame pointer is needed), the program counter on machines where that is considered one of the addressable registers, and any other numbered register with a standard use.

This information is expressed as a sequence of numbers, separated by commas and surrounded by braces. The *n*th number is 1 if register *n* is fixed, 0 otherwise.

The table initialized from this macro, and the table initialized by the following one, may be overridden at run time either automatically, by the actions of the macro `CONDITIONAL_REGISTER_USAGE`, or by the user with the command options '`-ffixed-`*reg*', '`-fcall-used-`*reg*' and '`-fcall-saved-`*reg*'.

`CALL_USED_REGISTERS` [Macro]

Like `FIXED_REGISTERS` but has 1 for each register that is clobbered (in general) by function calls as well as for fixed registers. This macro therefore identifies the registers that are not available for general allocation of values that must live across function calls.

If a register has 0 in `CALL_USED_REGISTERS`, the compiler automatically saves it on function entry and restores it on function exit, if the register is used within the function.

`CALL_REALLY_USED_REGISTERS` [Macro]

Like `CALL_USED_REGISTERS` except this macro doesn't require that the entire set of `FIXED_REGISTERS` be included. (`CALL_USED_REGISTERS` must be a superset of `FIXED_REGISTERS`). This macro is optional. If not specified, it defaults to the value of `CALL_USED_REGISTERS`.

`HARD_REGNO_CALL_PART_CLOBBERED` (*regno*, *mode*) [Macro]

A C expression that is nonzero if it is not permissible to store a value of mode *mode* in hard register number *regno* across a call without some part of it being clobbered. For most machines this macro need not be defined. It is only required for machines that do not preserve the entire contents of a register across a call.

`CONDITIONAL_REGISTER_USAGE` [Macro]

Zero or more C statements that may conditionally modify five variables `fixed_regs`, `call_used_regs`, `global_regs`, `reg_names`, and `reg_class_contents`, to take into account any dependence of these register sets on target flags. The first three of these are of type `char []` (interpreted as Boolean vectors). `global_regs` is a `const char *[]`, and `reg_class_contents` is a `HARD_REG_SET`. Before the macro is called, `fixed_regs`, `call_used_regs`, `reg_class_contents`, and `reg_names` have been initialized from `FIXED_REGISTERS`, `CALL_USED_REGISTERS`, `REG_CLASS_CONTENTS`, and `REGISTER_NAMES`, respectively. `global_regs` has been cleared, and any '`-ffixed-`*reg*', '`-fcall-used-`*reg*' and '`-fcall-saved-`*reg*' command options have been applied.

You need not define this macro if it has no work to do.

If the usage of an entire class of registers depends on the target flags, you may indicate this to GCC by using this macro to modify `fixed_regs` and `call_used_regs` to 1 for each of the registers in the classes which should not be used by GCC. Also define the macro `REG_CLASS_FROM_LETTER` / `REG_CLASS_FROM_CONSTRAINT` to return `NO_REGS` if it is called with a letter for a class that shouldn't be used.

(However, if this class is not included in `GENERAL_REGS` and all of the insn patterns whose constraints permit this class are controlled by target switches, then GCC will

automatically avoid using these registers when the target switches are opposed to them.)

`INCOMING_REGNO (out)` [Macro]
Define this macro if the target machine has register windows. This C expression returns the register number as seen by the called function corresponding to the register number *out* as seen by the calling function. Return *out* if register number *out* is not an outbound register.

`OUTGOING_REGNO (in)` [Macro]
Define this macro if the target machine has register windows. This C expression returns the register number as seen by the calling function corresponding to the register number *in* as seen by the called function. Return *in* if register number *in* is not an inbound register.

`LOCAL_REGNO (regno)` [Macro]
Define this macro if the target machine has register windows. This C expression returns true if the register is call-saved but is in the register window. Unlike most call-saved registers, such registers need not be explicitly restored on function exit or during non-local gotos.

`PC_REGNUM` [Macro]
If the program counter has a register number, define this as that register number. Otherwise, do not define it.

14.7.2 Order of Allocation of Registers

Registers are allocated in order.

`REG_ALLOC_ORDER` [Macro]
If defined, an initializer for a vector of integers, containing the numbers of hard registers in the order in which GCC should prefer to use them (from most preferred to least).

If this macro is not defined, registers are used lowest numbered first (all else being equal).

One use of this macro is on machines where the highest numbered registers must always be saved and the save-multiple-registers instruction supports only sequences of consecutive registers. On such machines, define `REG_ALLOC_ORDER` to be an initializer that lists the highest numbered allocable register first.

`ORDER_REGS_FOR_LOCAL_ALLOC` [Macro]
A C statement (sans semicolon) to choose the order in which to allocate hard registers for pseudo-registers local to a basic block.

Store the desired register order in the array `reg_alloc_order`. Element 0 should be the register to allocate first; element 1, the next register; and so on.

The macro body should not assume anything about the contents of `reg_alloc_order` before execution of the macro.

On most machines, it is not necessary to define this macro.

14.7.3 How Values Fit in Registers

This section discusses the macros that describe which kinds of values (specifically, which machine modes) each register can hold, and how many consecutive registers are needed for a given mode.

`HARD_REGNO_NREGS (`*`regno`*`, `*`mode`*`)` [Macro]

A C expression for the number of consecutive hard registers, starting at register number *regno*, required to hold a value of mode *mode*.

On a machine where all registers are exactly one word, a suitable definition of this macro is

```
#define HARD_REGNO_NREGS(REGNO, MODE)            \
   ((GET_MODE_SIZE (MODE) + UNITS_PER_WORD - 1)  \
    / UNITS_PER_WORD)
```

`REGMODE_NATURAL_SIZE (`*`mode`*`)` [Macro]

Define this macro if the natural size of registers that hold values of mode *mode* is not the word size. It is a C expression that should give the natural size in bytes for the specified mode. It is used by the register allocator to try to optimize its results. This happens for example on SPARC 64-bit where the natural size of floating-point registers is still 32-bit.

`HARD_REGNO_MODE_OK (`*`regno`*`, `*`mode`*`)` [Macro]

A C expression that is nonzero if it is permissible to store a value of mode *mode* in hard register number *regno* (or in several registers starting with that one). For a machine where all registers are equivalent, a suitable definition is

```
#define HARD_REGNO_MODE_OK(REGNO, MODE) 1
```

You need not include code to check for the numbers of fixed registers, because the allocation mechanism considers them to be always occupied.

On some machines, double-precision values must be kept in even/odd register pairs. You can implement that by defining this macro to reject odd register numbers for such modes.

The minimum requirement for a mode to be OK in a register is that the '`mov`*`mode`*' instruction pattern support moves between the register and other hard register in the same class and that moving a value into the register and back out not alter it.

Since the same instruction used to move `word_mode` will work for all narrower integer modes, it is not necessary on any machine for `HARD_REGNO_MODE_OK` to distinguish between these modes, provided you define patterns '`movhi`', etc., to take advantage of this. This is useful because of the interaction between `HARD_REGNO_MODE_OK` and `MODES_TIEABLE_P`; it is very desirable for all integer modes to be tieable.

Many machines have special registers for floating point arithmetic. Often people assume that floating point machine modes are allowed only in floating point registers. This is not true. Any registers that can hold integers can safely *hold* a floating point machine mode, whether or not floating arithmetic can be done on it in those registers. Integer move instructions can be used to move the values.

On some machines, though, the converse is true: fixed-point machine modes may not go in floating registers. This is true if the floating registers normalize any value

stored in them, because storing a non-floating value there would garble it. In this case, `HARD_REGNO_MODE_OK` should reject fixed-point machine modes in floating registers. But if the floating registers do not automatically normalize, if you can store any bit pattern in one and retrieve it unchanged without a trap, then any machine mode may go in a floating register, so you can define this macro to say so.

The primary significance of special floating registers is rather that they are the registers acceptable in floating point arithmetic instructions. However, this is of no concern to `HARD_REGNO_MODE_OK`. You handle it by writing the proper constraints for those instructions.

On some machines, the floating registers are especially slow to access, so that it is better to store a value in a stack frame than in such a register if floating point arithmetic is not being done. As long as the floating registers are not in class `GENERAL_REGS`, they will not be used unless some pattern's constraint asks for one.

`HARD_REGNO_RENAME_OK (`*`from`*`, `*`to`*`)` [Macro]

A C expression that is nonzero if it is OK to rename a hard register *from* to another hard register *to*.

One common use of this macro is to prevent renaming of a register to another register that is not saved by a prologue in an interrupt handler.

The default is always nonzero.

`MODES_TIEABLE_P (`*`mode1`*`, `*`mode2`*`)` [Macro]

A C expression that is nonzero if a value of mode *mode1* is accessible in mode *mode2* without copying.

If `HARD_REGNO_MODE_OK (`*`r`*`, `*`mode1`*`)` and `HARD_REGNO_MODE_OK (`*`r`*`, `*`mode2`*`)` are always the same for any *r*, then `MODES_TIEABLE_P (`*`mode1`*`, `*`mode2`*`)` should be nonzero. If they differ for any *r*, you should define this macro to return zero unless some other mechanism ensures the accessibility of the value in a narrower mode.

You should define this macro to return nonzero in as many cases as possible since doing so will allow GCC to perform better register allocation.

`AVOID_CCMODE_COPIES` [Macro]

Define this macro if the compiler should avoid copies to/from `CCmode` registers. You should only define this macro if support for copying to/from `CCmode` is incomplete.

14.7.4 Handling Leaf Functions

On some machines, a leaf function (i.e., one which makes no calls) can run more efficiently if it does not make its own register window. Often this means it is required to receive its arguments in the registers where they are passed by the caller, instead of the registers where they would normally arrive.

The special treatment for leaf functions generally applies only when other conditions are met; for example, often they may use only those registers for its own variables and temporaries. We use the term "leaf function" to mean a function that is suitable for this special handling, so that functions with no calls are not necessarily "leaf functions".

GCC assigns register numbers before it knows whether the function is suitable for leaf function treatment. So it needs to renumber the registers in order to output a leaf function. The following macros accomplish this.

`LEAF_REGISTERS` [Macro]

Name of a char vector, indexed by hard register number, which contains 1 for a register that is allowable in a candidate for leaf function treatment.

If leaf function treatment involves renumbering the registers, then the registers marked here should be the ones before renumbering—those that GCC would ordinarily allocate. The registers which will actually be used in the assembler code, after renumbering, should not be marked with 1 in this vector.

Define this macro only if the target machine offers a way to optimize the treatment of leaf functions.

`LEAF_REG_REMAP (`*regno*`)` [Macro]

A C expression whose value is the register number to which *regno* should be renumbered, when a function is treated as a leaf function.

If *regno* is a register number which should not appear in a leaf function before renumbering, then the expression should yield −1, which will cause the compiler to abort.

Define this macro only if the target machine offers a way to optimize the treatment of leaf functions, and registers need to be renumbered to do this.

`TARGET_ASM_FUNCTION_PROLOGUE` and `TARGET_ASM_FUNCTION_EPILOGUE` must usually treat leaf functions specially. They can test the C variable `current_function_is_leaf` which is nonzero for leaf functions. `current_function_is_leaf` is set prior to local register allocation and is valid for the remaining compiler passes. They can also test the C variable `current_function_uses_only_leaf_regs` which is nonzero for leaf functions which only use leaf registers. `current_function_uses_only_leaf_regs` is valid after all passes that modify the instructions have been run and is only useful if `LEAF_REGISTERS` is defined.

14.7.5 Registers That Form a Stack

There are special features to handle computers where some of the "registers" form a stack. Stack registers are normally written by pushing onto the stack, and are numbered relative to the top of the stack.

Currently, GCC can only handle one group of stack-like registers, and they must be consecutively numbered. Furthermore, the existing support for stack-like registers is specific to the 80387 floating point coprocessor. If you have a new architecture that uses stack-like registers, you will need to do substantial work on '`reg-stack.c`' and write your machine description to cooperate with it, as well as defining these macros.

`STACK_REGS` [Macro]

Define this if the machine has any stack-like registers.

`FIRST_STACK_REG` [Macro]

The number of the first stack-like register. This one is the top of the stack.

`LAST_STACK_REG` [Macro]

The number of the last stack-like register. This one is the bottom of the stack.

14.8 Register Classes

On many machines, the numbered registers are not all equivalent. For example, certain registers may not be allowed for indexed addressing; certain registers may not be allowed in some instructions. These machine restrictions are described to the compiler using *register classes*.

You define a number of register classes, giving each one a name and saying which of the registers belong to it. Then you can specify register classes that are allowed as operands to particular instruction patterns.

In general, each register will belong to several classes. In fact, one class must be named `ALL_REGS` and contain all the registers. Another class must be named `NO_REGS` and contain no registers. Often the union of two classes will be another class; however, this is not required.

One of the classes must be named `GENERAL_REGS`. There is nothing terribly special about the name, but the operand constraint letters '`r`' and '`g`' specify this class. If `GENERAL_REGS` is the same as `ALL_REGS`, just define it as a macro which expands to `ALL_REGS`.

Order the classes so that if class x is contained in class y then x has a lower class number than y.

The way classes other than `GENERAL_REGS` are specified in operand constraints is through machine-dependent operand constraint letters. You can define such letters to correspond to various classes, then use them in operand constraints.

You should define a class for the union of two classes whenever some instruction allows both classes. For example, if an instruction allows either a floating point (coprocessor) register or a general register for a certain operand, you should define a class `FLOAT_OR_GENERAL_REGS` which includes both of them. Otherwise you will get suboptimal code.

You must also specify certain redundant information about the register classes: for each class, which classes contain it and which ones are contained in it; for each pair of classes, the largest class contained in their union.

When a value occupying several consecutive registers is expected in a certain class, all the registers used must belong to that class. Therefore, register classes cannot be used to enforce a requirement for a register pair to start with an even-numbered register. The way to specify this requirement is with `HARD_REGNO_MODE_OK`.

Register classes used for input-operands of bitwise-and or shift instructions have a special requirement: each such class must have, for each fixed-point machine mode, a subclass whose registers can transfer that mode to or from memory. For example, on some machines, the operations for single-byte values (`QImode`) are limited to certain registers. When this is so, each register class that is used in a bitwise-and or shift instruction must have a subclass consisting of registers from which single-byte values can be loaded or stored. This is so that `PREFERRED_RELOAD_CLASS` can always have a possible value to return.

`enum reg_class` [Data type]

An enumerated type that must be defined with all the register class names as enumerated values. `NO_REGS` must be first. `ALL_REGS` must be the last register class, followed by one more enumerated value, `LIM_REG_CLASSES`, which is not a register class but rather tells how many classes there are.

Each register class has a number, which is the value of casting the class name to type `int`. The number serves as an index in many of the tables described below.

`N_REG_CLASSES` [Macro]

The number of distinct register classes, defined as follows:

```
#define N_REG_CLASSES (int) LIM_REG_CLASSES
```

`REG_CLASS_NAMES` [Macro]

An initializer containing the names of the register classes as C string constants. These names are used in writing some of the debugging dumps.

`REG_CLASS_CONTENTS` [Macro]

An initializer containing the contents of the register classes, as integers which are bit masks. The *n*th integer specifies the contents of class *n*. The way the integer *mask* is interpreted is that register *r* is in the class if `mask & (1 << r)` is 1.

When the machine has more than 32 registers, an integer does not suffice. Then the integers are replaced by sub-initializers, braced groupings containing several integers. Each sub-initializer must be suitable as an initializer for the type `HARD_REG_SET` which is defined in '`hard-reg-set.h`'. In this situation, the first integer in each sub-initializer corresponds to registers 0 through 31, the second integer to registers 32 through 63, and so on.

`REGNO_REG_CLASS (`*regno*`)` [Macro]

A C expression whose value is a register class containing hard register *regno*. In general there is more than one such class; choose a class which is *minimal*, meaning that no smaller class also contains the register.

`BASE_REG_CLASS` [Macro]

A macro whose definition is the name of the class to which a valid base register must belong. A base register is one used in an address which is the register value plus a displacement.

`MODE_BASE_REG_CLASS (`*mode*`)` [Macro]

This is a variation of the `BASE_REG_CLASS` macro which allows the selection of a base register in a mode dependent manner. If *mode* is VOIDmode then it should return the same value as `BASE_REG_CLASS`.

`MODE_BASE_REG_REG_CLASS (`*mode*`)` [Macro]

A C expression whose value is the register class to which a valid base register must belong in order to be used in a base plus index register address. You should define this macro if base plus index addresses have different requirements than other base register uses.

`INDEX_REG_CLASS` [Macro]

A macro whose definition is the name of the class to which a valid index register must belong. An index register is one used in an address where its value is either multiplied by a scale factor or added to another register (as well as added to a displacement).

`CONSTRAINT_LEN (char, str)` [Macro]

For the constraint at the start of *str*, which starts with the letter *c*, return the length. This allows you to have register class / constant / extra constraints that are longer than a single letter; you don't need to define this macro if you can do with single-letter constraints only. The definition of this macro should use DEFAULT_CONSTRAINT_LEN for all the characters that you don't want to handle specially. There are some sanity checks in genoutput.c that check the constraint lengths for the md file, so you can also use this macro to help you while you are transitioning from a byzantine single-letter-constraint scheme: when you return a negative length for a constraint you want to re-use, genoutput will complain about every instance where it is used in the md file.

`REG_CLASS_FROM_LETTER (char)` [Macro]

A C expression which defines the machine-dependent operand constraint letters for register classes. If *char* is such a letter, the value should be the register class corresponding to it. Otherwise, the value should be `NO_REGS`. The register letter '`r`', corresponding to class `GENERAL_REGS`, will not be passed to this macro; you do not need to handle it.

`REG_CLASS_FROM_CONSTRAINT (char, str)` [Macro]

Like `REG_CLASS_FROM_LETTER`, but you also get the constraint string passed in *str*, so that you can use suffixes to distinguish between different variants.

`REGNO_OK_FOR_BASE_P (num)` [Macro]

A C expression which is nonzero if register number *num* is suitable for use as a base register in operand addresses. It may be either a suitable hard register or a pseudo register that has been allocated such a hard register.

`REGNO_MODE_OK_FOR_BASE_P (num, mode)` [Macro]

A C expression that is just like `REGNO_OK_FOR_BASE_P`, except that that expression may examine the mode of the memory reference in *mode*. You should define this macro if the mode of the memory reference affects whether a register may be used as a base register. If you define this macro, the compiler will use it instead of `REGNO_OK_FOR_BASE_P`.

`REGNO_MODE_OK_FOR_REG_BASE_P (num, mode)` [Macro]

A C expression which is nonzero if register number *num* is suitable for use as a base register in base plus index operand addresses, accessing memory in mode *mode*. It may be either a suitable hard register or a pseudo register that has been allocated such a hard register. You should define this macro if base plus index addresses have different requirements than other base register uses.

`REGNO_OK_FOR_INDEX_P (num)` [Macro]

A C expression which is nonzero if register number *num* is suitable for use as an index register in operand addresses. It may be either a suitable hard register or a pseudo register that has been allocated such a hard register.

The difference between an index register and a base register is that the index register may be scaled. If an address involves the sum of two registers, neither one of them scaled, then either one may be labeled the "base" and the other the "index"; but

whichever labeling is used must fit the machine's constraints of which registers may serve in each capacity. The compiler will try both labelings, looking for one that is valid, and will reload one or both registers only if neither labeling works.

`PREFERRED_RELOAD_CLASS (x, class)` [Macro]

A C expression that places additional restrictions on the register class to use when it is necessary to copy value *x* into a register in class *class*. The value is a register class; perhaps *class*, or perhaps another, smaller class. On many machines, the following definition is safe:

```
#define PREFERRED_RELOAD_CLASS(X,CLASS) CLASS
```

Sometimes returning a more restrictive class makes better code. For example, on the 68000, when *x* is an integer constant that is in range for a '`moveq`' instruction, the value of this macro is always `DATA_REGS` as long as *class* includes the data registers. Requiring a data register guarantees that a '`moveq`' will be used.

One case where `PREFERRED_RELOAD_CLASS` must not return *class* is if *x* is a legitimate constant which cannot be loaded into some register class. By returning `NO_REGS` you can force *x* into a memory location. For example, rs6000 can load immediate values into general-purpose registers, but does not have an instruction for loading an immediate value into a floating-point register, so `PREFERRED_RELOAD_CLASS` returns `NO_REGS` when *x* is a floating-point constant. If the constant can't be loaded into any kind of register, code generation will be better if `LEGITIMATE_CONSTANT_P` makes the constant illegitimate instead of using `PREFERRED_RELOAD_CLASS`.

`PREFERRED_OUTPUT_RELOAD_CLASS (x, class)` [Macro]

Like `PREFERRED_RELOAD_CLASS`, but for output reloads instead of input reloads. If you don't define this macro, the default is to use *class*, unchanged.

`LIMIT_RELOAD_CLASS (mode, class)` [Macro]

A C expression that places additional restrictions on the register class to use when it is necessary to be able to hold a value of mode *mode* in a reload register for which class *class* would ordinarily be used.

Unlike `PREFERRED_RELOAD_CLASS`, this macro should be used when there are certain modes that simply can't go in certain reload classes.

The value is a register class; perhaps *class*, or perhaps another, smaller class.

Don't define this macro unless the target machine has limitations which require the macro to do something nontrivial.

`SECONDARY_RELOAD_CLASS (class, mode, x)` [Macro]
`SECONDARY_INPUT_RELOAD_CLASS (class, mode, x)` [Macro]
`SECONDARY_OUTPUT_RELOAD_CLASS (class, mode, x)` [Macro]

Many machines have some registers that cannot be copied directly to or from memory or even from other types of registers. An example is the '`MQ`' register, which on most machines, can only be copied to or from general registers, but not memory. Some machines allow copying all registers to and from memory, but require a scratch register for stores to some memory locations (e.g., those with symbolic address on the RT, and those with certain symbolic address on the SPARC when compiling PIC). In some cases, both an intermediate and a scratch register are required.

You should define these macros to indicate to the reload phase that it may need to allocate at least one register for a reload in addition to the register to contain the data. Specifically, if copying *x* to a register *class* in *mode* requires an intermediate register, you should define `SECONDARY_INPUT_RELOAD_CLASS` to return the largest register class all of whose registers can be used as intermediate registers or scratch registers.

If copying a register *class* in *mode* to *x* requires an intermediate or scratch register, `SECONDARY_OUTPUT_RELOAD_CLASS` should be defined to return the largest register class required. If the requirements for input and output reloads are the same, the macro `SECONDARY_RELOAD_CLASS` should be used instead of defining both macros identically.

The values returned by these macros are often `GENERAL_REGS`. Return `NO_REGS` if no spare register is needed; i.e., if *x* can be directly copied to or from a register of *class* in *mode* without requiring a scratch register. Do not define this macro if it would always return `NO_REGS`.

If a scratch register is required (either with or without an intermediate register), you should define patterns for '`reload_in`*m*' or '`reload_out`*m*', as required (see Section 13.9 [Standard Names], page 212. These patterns, which will normally be implemented with a `define_expand`, should be similar to the '`mov`*m*' patterns, except that operand 2 is the scratch register.

Define constraints for the reload register and scratch register that contain a single register class. If the original reload register (whose class is *class*) can meet the constraint given in the pattern, the value returned by these macros is used for the class of the scratch register. Otherwise, two additional reload registers are required. Their classes are obtained from the constraints in the insn pattern.

x might be a pseudo-register or a `subreg` of a pseudo-register, which could either be in a hard register or in memory. Use `true_regnum` to find out; it will return -1 if the pseudo is in memory and the hard register number if it is in a register.

These macros should not be used in the case where a particular class of registers can only be copied to memory and not to another class of registers. In that case, secondary reload registers are not needed and would not be helpful. Instead, a stack location must be used to perform the copy and the `mov`*m* pattern should use memory as an intermediate storage. This case often occurs between floating-point and general registers.

`SECONDARY_MEMORY_NEEDED (`*class1*, *class2*, *m*`)` [Macro]

Certain machines have the property that some registers cannot be copied to some other registers without using memory. Define this macro on those machines to be a C expression that is nonzero if objects of mode *m* in registers of *class1* can only be copied to registers of class *class2* by storing a register of *class1* into memory and loading that memory location into a register of *class2*.

Do not define this macro if its value would always be zero.

`SECONDARY_MEMORY_NEEDED_RTX (`*mode*`)` [Macro]

Normally when `SECONDARY_MEMORY_NEEDED` is defined, the compiler allocates a stack slot for a memory location needed for register copies. If this macro is defined, the compiler instead uses the memory location defined by this macro.

Do not define this macro if you do not define `SECONDARY_MEMORY_NEEDED`.

`SECONDARY_MEMORY_NEEDED_MODE (`*`mode`*`)` [Macro]

When the compiler needs a secondary memory location to copy between two registers of mode *mode*, it normally allocates sufficient memory to hold a quantity of `BITS_PER_WORD` bits and performs the store and load operations in a mode that many bits wide and whose class is the same as that of *mode*.

This is right thing to do on most machines because it ensures that all bits of the register are copied and prevents accesses to the registers in a narrower mode, which some machines prohibit for floating-point registers.

However, this default behavior is not correct on some machines, such as the DEC Alpha, that store short integers in floating-point registers differently than in integer registers. On those machines, the default widening will not work correctly and you must define this macro to suppress that widening in some cases. See the file '`alpha.h`' for details.

Do not define this macro if you do not define `SECONDARY_MEMORY_NEEDED` or if widening *mode* to a mode that is `BITS_PER_WORD` bits wide is correct for your machine.

`SMALL_REGISTER_CLASSES` [Macro]

On some machines, it is risky to let hard registers live across arbitrary insns. Typically, these machines have instructions that require values to be in specific registers (like an accumulator), and reload will fail if the required hard register is used for another purpose across such an insn.

Define `SMALL_REGISTER_CLASSES` to be an expression with a nonzero value on these machines. When this macro has a nonzero value, the compiler will try to minimize the lifetime of hard registers.

It is always safe to define this macro with a nonzero value, but if you unnecessarily define it, you will reduce the amount of optimizations that can be performed in some cases. If you do not define this macro with a nonzero value when it is required, the compiler will run out of spill registers and print a fatal error message. For most machines, you should not define this macro at all.

`CLASS_LIKELY_SPILLED_P (`*`class`*`)` [Macro]

A C expression whose value is nonzero if pseudos that have been assigned to registers of class *class* would likely be spilled because registers of *class* are needed for spill registers.

The default value of this macro returns 1 if *class* has exactly one register and zero otherwise. On most machines, this default should be used. Only define this macro to some other expression if pseudos allocated by '`local-alloc.c`' end up in memory because their hard registers were needed for spill registers. If this macro returns nonzero for those classes, those pseudos will only be allocated by '`global.c`', which knows how to reallocate the pseudo to another register. If there would not be another register available for reallocation, you should not change the definition of this macro since the only effect of such a definition would be to slow down register allocation.

`CLASS_MAX_NREGS (`*`class`*`, `*`mode`*`)` [Macro]

A C expression for the maximum number of consecutive registers of class *class* needed to hold a value of mode *mode*.

This is closely related to the macro `HARD_REGNO_NREGS`. In fact, the value of the macro `CLASS_MAX_NREGS (`*`class`*`, `*`mode`*`)` should be the maximum value of `HARD_REGNO_NREGS (`*`regno`*`, `*`mode`*`)` for all *regno* values in the class *class*.

This macro helps control the handling of multiple-word values in the reload pass.

`CANNOT_CHANGE_MODE_CLASS (`*`from`*`, `*`to`*`, `*`class`*`)` [Macro]

If defined, a C expression that returns nonzero for a *class* for which a change from mode *from* to mode *to* is invalid.

For the example, loading 32-bit integer or floating-point objects into floating-point registers on the Alpha extends them to 64 bits. Therefore loading a 64-bit object and then storing it as a 32-bit object does not store the low-order 32 bits, as would be the case for a normal register. Therefore, '`alpha.h`' defines `CANNOT_CHANGE_MODE_CLASS` as below:

```
#define CANNOT_CHANGE_MODE_CLASS(FROM, TO, CLASS) \
  (GET_MODE_SIZE (FROM) != GET_MODE_SIZE (TO) \
   ? reg_classes_intersect_p (FLOAT_REGS, (CLASS)) : 0)
```

Three other special macros describe which operands fit which constraint letters.

`CONST_OK_FOR_LETTER_P (`*`value`*`, `*`c`*`)` [Macro]

A C expression that defines the machine-dependent operand constraint letters ('`I`', '`J`', '`K`', ... '`P`') that specify particular ranges of integer values. If *c* is one of those letters, the expression should check that *value*, an integer, is in the appropriate range and return 1 if so, 0 otherwise. If *c* is not one of those letters, the value should be 0 regardless of *value*.

`CONST_OK_FOR_CONSTRAINT_P (`*`value`*`, `*`c`*`, `*`str`*`)` [Macro]

Like `CONST_OK_FOR_LETTER_P`, but you also get the constraint string passed in *str*, so that you can use suffixes to distinguish between different variants.

`CONST_DOUBLE_OK_FOR_LETTER_P (`*`value`*`, `*`c`*`)` [Macro]

A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of `const_double` values ('`G`' or '`H`').

If *c* is one of those letters, the expression should check that *value*, an RTX of code `const_double`, is in the appropriate range and return 1 if so, 0 otherwise. If *c* is not one of those letters, the value should be 0 regardless of *value*.

`const_double` is used for all floating-point constants and for `DImode` fixed-point constants. A given letter can accept either or both kinds of values. It can use `GET_MODE` to distinguish between these kinds.

`CONST_DOUBLE_OK_FOR_CONSTRAINT_P (`*`value`*`, `*`c`*`, `*`str`*`)` [Macro]

Like `CONST_DOUBLE_OK_FOR_LETTER_P`, but you also get the constraint string passed in *str*, so that you can use suffixes to distinguish between different variants.

`EXTRA_CONSTRAINT (`*`value`*`, `*`c`*`)` [Macro]

A C expression that defines the optional machine-dependent constraint letters that can be used to segregate specific types of operands, usually memory references, for the target machine. Any letter that is not elsewhere defined and not matched by

`REG_CLASS_FROM_LETTER` / `REG_CLASS_FROM_CONSTRAINT` may be used. Normally this macro will not be defined.

If it is required for a particular target machine, it should return 1 if *value* corresponds to the operand type represented by the constraint letter *c*. If *c* is not defined as an extra constraint, the value returned should be 0 regardless of *value*.

For example, on the ROMP, load instructions cannot have their output in r0 if the memory reference contains a symbolic address. Constraint letter '`Q`' is defined as representing a memory address that does *not* contain a symbolic address. An alternative is specified with a '`Q`' constraint on the input and '`r`' on the output. The next alternative specifies '`m`' on the input and a register class that does not include r0 on the output.

`EXTRA_CONSTRAINT_STR (`*value*, *c*, *str*`)` [Macro]

Like `EXTRA_CONSTRAINT`, but you also get the constraint string passed in *str*, so that you can use suffixes to distinguish between different variants.

`EXTRA_MEMORY_CONSTRAINT (`*c*, *str*`)` [Macro]

A C expression that defines the optional machine-dependent constraint letters, amongst those accepted by `EXTRA_CONSTRAINT`, that should be treated like memory constraints by the reload pass.

It should return 1 if the operand type represented by the constraint at the start of *str*, the first letter of which is the letter *c*, comprises a subset of all memory references including all those whose address is simply a base register. This allows the reload pass to reload an operand, if it does not directly correspond to the operand type of *c*, by copying its address into a base register.

For example, on the S/390, some instructions do not accept arbitrary memory references, but only those that do not make use of an index register. The constraint letter '`Q`' is defined via `EXTRA_CONSTRAINT` as representing a memory address of this type. If the letter '`Q`' is marked as `EXTRA_MEMORY_CONSTRAINT`, a '`Q`' constraint can handle any memory operand, because the reload pass knows it can be reloaded by copying the memory address into a base register if required. This is analogous to the way a '`o`' constraint can handle any memory operand.

`EXTRA_ADDRESS_CONSTRAINT (`*c*, *str*`)` [Macro]

A C expression that defines the optional machine-dependent constraint letters, amongst those accepted by `EXTRA_CONSTRAINT` / `EXTRA_CONSTRAINT_STR`, that should be treated like address constraints by the reload pass.

It should return 1 if the operand type represented by the constraint at the start of *str*, which starts with the letter *c*, comprises a subset of all memory addresses including all those that consist of just a base register. This allows the reload pass to reload an operand, if it does not directly correspond to the operand type of *str*, by copying it into a base register.

Any constraint marked as `EXTRA_ADDRESS_CONSTRAINT` can only be used with the `address_operand` predicate. It is treated analogously to the '`p`' constraint.

14.9 Stack Layout and Calling Conventions

This describes the stack layout and calling conventions.

14.9.1 Basic Stack Layout

Here is the basic stack layout.

`STACK_GROWS_DOWNWARD` [Macro]

Define this macro if pushing a word onto the stack moves the stack pointer to a smaller address.

When we say, "define this macro if . . .", it means that the compiler checks this macro only with `#ifdef` so the precise definition used does not matter.

`STACK_PUSH_CODE` [Macro]

This macro defines the operation used when something is pushed on the stack. In RTL, a push operation will be `(set (mem (STACK_PUSH_CODE (reg sp))) ...)`

The choices are `PRE_DEC`, `POST_DEC`, `PRE_INC`, and `POST_INC`. Which of these is correct depends on the stack direction and on whether the stack pointer points to the last item on the stack or whether it points to the space for the next item on the stack.

The default is `PRE_DEC` when `STACK_GROWS_DOWNWARD` is defined, which is almost always right, and `PRE_INC` otherwise, which is often wrong.

`FRAME_GROWS_DOWNWARD` [Macro]

Define this macro to nonzero value if the addresses of local variable slots are at negative offsets from the frame pointer.

`ARGS_GROW_DOWNWARD` [Macro]

Define this macro if successive arguments to a function occupy decreasing addresses on the stack.

`STARTING_FRAME_OFFSET` [Macro]

Offset from the frame pointer to the first local variable slot to be allocated.

If `FRAME_GROWS_DOWNWARD`, find the next slot's offset by subtracting the first slot's length from `STARTING_FRAME_OFFSET`. Otherwise, it is found by adding the length of the first slot to the value `STARTING_FRAME_OFFSET`.

`STACK_ALIGNMENT_NEEDED` [Macro]

Define to zero to disable final alignment of the stack during reload. The nonzero default for this macro is suitable for most ports.

On ports where `STARTING_FRAME_OFFSET` is nonzero or where there is a register save block following the local block that doesn't require alignment to `STACK_BOUNDARY`, it may be beneficial to disable stack alignment and do it in the backend.

`STACK_POINTER_OFFSET` [Macro]

Offset from the stack pointer register to the first location at which outgoing arguments are placed. If not specified, the default value of zero is used. This is the proper value for most machines.

If `ARGS_GROW_DOWNWARD`, this is the offset to the location above the first location at which outgoing arguments are placed.

`FIRST_PARM_OFFSET (fundecl)` [Macro]

Offset from the argument pointer register to the first argument's address. On some machines it may depend on the data type of the function.

If `ARGS_GROW_DOWNWARD`, this is the offset to the location above the first argument's address.

`STACK_DYNAMIC_OFFSET (fundecl)` [Macro]

Offset from the stack pointer register to an item dynamically allocated on the stack, e.g., by `alloca`.

The default value for this macro is `STACK_POINTER_OFFSET` plus the length of the outgoing arguments. The default is correct for most machines. See '`function.c`' for details.

`INITIAL_FRAME_ADDRESS_RTX` [Macro]

A C expression whose value is RTL representing the address of the initial stack frame. This address is passed to `RETURN_ADDR_RTX` and `DYNAMIC_CHAIN_ADDRESS`. If you don't define this macro, a reasonable default value will be used. Define this macro in order to make frame pointer elimination work in the presence of `__builtin_frame_address (count)` and `__builtin_return_address (count)` for `count` not equal to zero.

`DYNAMIC_CHAIN_ADDRESS (frameaddr)` [Macro]

A C expression whose value is RTL representing the address in a stack frame where the pointer to the caller's frame is stored. Assume that *frameaddr* is an RTL expression for the address of the stack frame itself.

If you don't define this macro, the default is to return the value of *frameaddr*—that is, the stack frame address is also the address of the stack word that points to the previous frame.

`SETUP_FRAME_ADDRESSES` [Macro]

If defined, a C expression that produces the machine-specific code to setup the stack so that arbitrary frames can be accessed. For example, on the SPARC, we must flush all of the register windows to the stack before we can access arbitrary stack frames. You will seldom need to define this macro.

`bool TARGET_BUILTIN_SETJMP_FRAME_VALUE ()` [Target Hook]

This target hook should return an rtx that is used to store the address of the current frame into the built in `setjmp` buffer. The default value, `virtual_stack_vars_rtx`, is correct for most machines. One reason you may need to define this target hook is if `hard_frame_pointer_rtx` is the appropriate value on your machine.

`RETURN_ADDR_RTX (count, frameaddr)` [Macro]

A C expression whose value is RTL representing the value of the return address for the frame *count* steps up from the current frame, after the prologue. *frameaddr* is the frame pointer of the *count* frame, or the frame pointer of the *count* − 1 frame if `RETURN_ADDR_IN_PREVIOUS_FRAME` is defined.

The value of the expression must always be the correct address when *count* is zero, but may be `NULL_RTX` if there is not way to determine the return address of other frames.

`RETURN_ADDR_IN_PREVIOUS_FRAME` [Macro]

Define this if the return address of a particular stack frame is accessed from the frame pointer of the previous stack frame.

`INCOMING_RETURN_ADDR_RTX` [Macro]

A C expression whose value is RTL representing the location of the incoming return address at the beginning of any function, before the prologue. This RTL is either a `REG`, indicating that the return value is saved in '`REG`', or a `MEM` representing a location in the stack.

You only need to define this macro if you want to support call frame debugging information like that provided by DWARF 2.

If this RTL is a `REG`, you should also define `DWARF_FRAME_RETURN_COLUMN` to `DWARF_FRAME_REGNUM (REGNO)`.

`DWARF_ALT_FRAME_RETURN_COLUMN` [Macro]

A C expression whose value is an integer giving a DWARF 2 column number that may be used as an alternate return column. This should be defined only if `DWARF_FRAME_RETURN_COLUMN` is set to a general register, but an alternate column needs to be used for signal frames.

`DWARF_ZERO_REG` [Macro]

A C expression whose value is an integer giving a DWARF 2 register number that is considered to always have the value zero. This should only be defined if the target has an architected zero register, and someone decided it was a good idea to use that register number to terminate the stack backtrace. New ports should avoid this.

`void TARGET_DWARF_HANDLE_FRAME_UNSPEC (`*const char* *`label`*`,` [Target Hook]
rtx `pattern`*, int* `index`*`)`

This target hook allows the backend to emit frame-related insns that contain UNSPECs or UNSPEC_VOLATILEs. The DWARF 2 call frame debugging info engine will invoke it on insns of the form

```
(set (reg) (unspec [...] UNSPEC_INDEX))
```

and

```
(set (reg) (unspec_volatile [...] UNSPECV_INDEX)).
```

to let the backend emit the call frame instructions. *label* is the CFI label attached to the insn, *pattern* is the pattern of the insn and *index* is `UNSPEC_INDEX` or `UNSPECV_INDEX`.

`INCOMING_FRAME_SP_OFFSET` [Macro]

A C expression whose value is an integer giving the offset, in bytes, from the value of the stack pointer register to the top of the stack frame at the beginning of any function, before the prologue. The top of the frame is defined to be the value of the stack pointer in the previous frame, just before the call instruction.

You only need to define this macro if you want to support call frame debugging information like that provided by DWARF 2.

`ARG_POINTER_CFA_OFFSET (`*`fundecl`*`)` [Macro]

A C expression whose value is an integer giving the offset, in bytes, from the argument pointer to the canonical frame address (cfa). The final value should coincide with that

calculated by `INCOMING_FRAME_SP_OFFSET`. Which is unfortunately not usable during virtual register instantiation.

The default value for this macro is `FIRST_PARM_OFFSET (fundecl)`, which is correct for most machines; in general, the arguments are found immediately before the stack frame. Note that this is not the case on some targets that save registers into the caller's frame, such as SPARC and rs6000, and so such targets need to define this macro.

You only need to define this macro if the default is incorrect, and you want to support call frame debugging information like that provided by DWARF 2.

`FRAME_POINTER_CFA_OFFSET (`*`fundecl`*`)` [Macro]

If defined, a C expression whose value is an integer giving the offset in bytes from the frame pointer to the canonical frame address (cfa). The final value should conincide with that calculated by `INCOMING_FRAME_SP_OFFSET`.

Normally the CFA is calculated as an offset from the argument pointer, via `ARG_POINTER_CFA_OFFSET`, but if the argument pointer is variable due to the ABI, this may not be possible. If this macro is defined, it imples that the virtual register instantiation should be based on the frame pointer instead of the argument pointer. Only one of `FRAME_POINTER_CFA_OFFSET` and `ARG_POINTER_CFA_OFFSET` should be defined.

14.9.2 Exception Handling Support

`EH_RETURN_DATA_REGNO (`*`N`*`)` [Macro]

A C expression whose value is the *N*th register number used for data by exception handlers, or `INVALID_REGNUM` if fewer than *N* registers are usable.

The exception handling library routines communicate with the exception handlers via a set of agreed upon registers. Ideally these registers should be call-clobbered; it is possible to use call-saved registers, but may negatively impact code size. The target must support at least 2 data registers, but should define 4 if there are enough free registers.

You must define this macro if you want to support call frame exception handling like that provided by DWARF 2.

`EH_RETURN_STACKADJ_RTX` [Macro]

A C expression whose value is RTL representing a location in which to store a stack adjustment to be applied before function return. This is used to unwind the stack to an exception handler's call frame. It will be assigned zero on code paths that return normally.

Typically this is a call-clobbered hard register that is otherwise untouched by the epilogue, but could also be a stack slot.

Do not define this macro if the stack pointer is saved and restored by the regular prolog and epilog code in the call frame itself; in this case, the exception handling library routines will update the stack location to be restored in place. Otherwise, you must define this macro if you want to support call frame exception handling like that provided by DWARF 2.

`EH_RETURN_HANDLER_RTX` [Macro]

A C expression whose value is RTL representing a location in which to store the address of an exception handler to which we should return. It will not be assigned on code paths that return normally.

Typically this is the location in the call frame at which the normal return address is stored. For targets that return by popping an address off the stack, this might be a memory address just below the *target* call frame rather than inside the current call frame. If defined, `EH_RETURN_STACKADJ_RTX` will have already been assigned, so it may be used to calculate the location of the target call frame.

Some targets have more complex requirements than storing to an address calculable during initial code generation. In that case the `eh_return` instruction pattern should be used instead.

If you want to support call frame exception handling, you must define either this macro or the `eh_return` instruction pattern.

`RETURN_ADDR_OFFSET` [Macro]

If defined, an integer-valued C expression for which rtl will be generated to add it to the exception handler address before it is searched in the exception handling tables, and to subtract it again from the address before using it to return to the exception handler.

`ASM_PREFERRED_EH_DATA_FORMAT (`*`code`*`, `*`global`*`)` [Macro]

This macro chooses the encoding of pointers embedded in the exception handling sections. If at all possible, this should be defined such that the exception handling section will not require dynamic relocations, and so may be read-only.

code is 0 for data, 1 for code labels, 2 for function pointers. *global* is true if the symbol may be affected by dynamic relocations. The macro should return a combination of the `DW_EH_PE_*` defines as found in '`dwarf2.h`'.

If this macro is not defined, pointers will not be encoded but represented directly.

`ASM_MAYBE_OUTPUT_ENCODED_ADDR_RTX (`*`file`*`, `*`encoding`*`, `*`size`*`, `*`addr`*`,` [Macro]
`done``)`

This macro allows the target to emit whatever special magic is required to represent the encoding chosen by `ASM_PREFERRED_EH_DATA_FORMAT`. Generic code takes care of pc-relative and indirect encodings; this must be defined if the target uses text-relative or data-relative encodings.

This is a C statement that branches to *done* if the format was handled. *encoding* is the format chosen, *size* is the number of bytes that the format occupies, *addr* is the `SYMBOL_REF` to be emitted.

`MD_UNWIND_SUPPORT` [Macro]

A string specifying a file to be #include'd in unwind-dw2.c. The file so included typically defines `MD_FALLBACK_FRAME_STATE_FOR`.

`MD_FALLBACK_FRAME_STATE_FOR (`*`context`*`, `*`fs`*`)` [Macro]

This macro allows the target to add cpu and operating system specific code to the call-frame unwinder for use when there is no unwind data available. The most common reason to implement this macro is to unwind through signal frames.

This macro is called from `uw_frame_state_for` in '`unwind-dw2.c`' and '`unwind-ia64.c`'. *context* is an `_Unwind_Context`; *fs* is an `_Unwind_FrameState`. Examine `context->ra` for the address of the code being executed and `context->cfa` for the stack pointer value. If the frame can be decoded, the register save addresses should be updated in *fs* and the macro should evaluate to `_URC_NO_REASON`. If the frame cannot be decoded, the macro should evaluate to `_URC_END_OF_STACK`.

For proper signal handling in Java this macro is accompanied by `MAKE_THROW_FRAME`, defined in '`libjava/include/*-signal.h`' headers.

`MD_HANDLE_UNWABI (context, fs)` [Macro]

This macro allows the target to add operating system specific code to the call-frame unwinder to handle the IA-64 `.unwabi` unwinding directive, usually used for signal or interrupt frames.

This macro is called from `uw_update_context` in '`unwind-ia64.c`'. *context* is an `_Unwind_Context`; *fs* is an `_Unwind_FrameState`. Examine `fs->unwabi` for the abi and context in the `.unwabi` directive. If the `.unwabi` directive can be handled, the register save addresses should be updated in *fs*.

`TARGET_USES_WEAK_UNWIND_INFO` [Macro]

A C expression that evaluates to true if the target requires unwind info to be given comdat linkage. Define it to be `1` if comdat linkage is necessary. The default is `0`.

14.9.3 Specifying How Stack Checking is Done

GCC will check that stack references are within the boundaries of the stack, if the '`-fstack-check`' is specified, in one of three ways:

1. If the value of the `STACK_CHECK_BUILTIN` macro is nonzero, GCC will assume that you have arranged for stack checking to be done at appropriate places in the configuration files, e.g., in `TARGET_ASM_FUNCTION_PROLOGUE`. GCC will do not other special processing.
2. If `STACK_CHECK_BUILTIN` is zero and you defined a named pattern called `check_stack` in your '`md`' file, GCC will call that pattern with one argument which is the address to compare the stack value against. You must arrange for this pattern to report an error if the stack pointer is out of range.
3. If neither of the above are true, GCC will generate code to periodically "probe" the stack pointer using the values of the macros defined below.

Normally, you will use the default values of these macros, so GCC will use the third approach.

`STACK_CHECK_BUILTIN` [Macro]

A nonzero value if stack checking is done by the configuration files in a machine-dependent manner. You should define this macro if stack checking is require by the ABI of your machine or if you would like to have to stack checking in some more efficient way than GCC's portable approach. The default value of this macro is zero.

`STACK_CHECK_PROBE_INTERVAL` [Macro]

An integer representing the interval at which GCC must generate stack probe instructions. You will normally define this macro to be no larger than the size of the

"guard pages" at the end of a stack area. The default value of 4096 is suitable for most systems.

`STACK_CHECK_PROBE_LOAD` [Macro]
A integer which is nonzero if GCC should perform the stack probe as a load instruction and zero if GCC should use a store instruction. The default is zero, which is the most efficient choice on most systems.

`STACK_CHECK_PROTECT` [Macro]
The number of bytes of stack needed to recover from a stack overflow, for languages where such a recovery is supported. The default value of 75 words should be adequate for most machines.

`STACK_CHECK_MAX_FRAME_SIZE` [Macro]
The maximum size of a stack frame, in bytes. GCC will generate probe instructions in non-leaf functions to ensure at least this many bytes of stack are available. If a stack frame is larger than this size, stack checking will not be reliable and GCC will issue a warning. The default is chosen so that GCC only generates one instruction on most systems. You should normally not change the default value of this macro.

`STACK_CHECK_FIXED_FRAME_SIZE` [Macro]
GCC uses this value to generate the above warning message. It represents the amount of fixed frame used by a function, not including space for any callee-saved registers, temporaries and user variables. You need only specify an upper bound for this amount and will normally use the default of four words.

`STACK_CHECK_MAX_VAR_SIZE` [Macro]
The maximum size, in bytes, of an object that GCC will place in the fixed area of the stack frame when the user specifies '`-fstack-check`'. GCC computed the default from the values of the above macros and you will normally not need to override that default.

14.9.4 Registers That Address the Stack Frame

This discusses registers that address the stack frame.

`STACK_POINTER_REGNUM` [Macro]
The register number of the stack pointer register, which must also be a fixed register according to `FIXED_REGISTERS`. On most machines, the hardware determines which register this is.

`FRAME_POINTER_REGNUM` [Macro]
The register number of the frame pointer register, which is used to access automatic variables in the stack frame. On some machines, the hardware determines which register this is. On other machines, you can choose any register you wish for this purpose.

`HARD_FRAME_POINTER_REGNUM` [Macro]
On some machines the offset between the frame pointer and starting offset of the automatic variables is not known until after register allocation has been done (for

example, because the saved registers are between these two locations). On those machines, define `FRAME_POINTER_REGNUM` the number of a special, fixed register to be used internally until the offset is known, and define `HARD_FRAME_POINTER_REGNUM` to be the actual hard register number used for the frame pointer.

You should define this macro only in the very rare circumstances when it is not possible to calculate the offset between the frame pointer and the automatic variables until after register allocation has been completed. When this macro is defined, you must also indicate in your definition of `ELIMINABLE_REGS` how to eliminate `FRAME_POINTER_REGNUM` into either `HARD_FRAME_POINTER_REGNUM` or `STACK_POINTER_REGNUM`.

Do not define this macro if it would be the same as `FRAME_POINTER_REGNUM`.

`ARG_POINTER_REGNUM` [Macro]

The register number of the arg pointer register, which is used to access the function's argument list. On some machines, this is the same as the frame pointer register. On some machines, the hardware determines which register this is. On other machines, you can choose any register you wish for this purpose. If this is not the same register as the frame pointer register, then you must mark it as a fixed register according to `FIXED_REGISTERS`, or arrange to be able to eliminate it (see Section 14.9.5 [Elimination], page 312).

`RETURN_ADDRESS_POINTER_REGNUM` [Macro]

The register number of the return address pointer register, which is used to access the current function's return address from the stack. On some machines, the return address is not at a fixed offset from the frame pointer or stack pointer or argument pointer. This register can be defined to point to the return address on the stack, and then be converted by `ELIMINABLE_REGS` into either the frame pointer or stack pointer.

Do not define this macro unless there is no other way to get the return address from the stack.

`STATIC_CHAIN_REGNUM` [Macro]

`STATIC_CHAIN_INCOMING_REGNUM` [Macro]

Register numbers used for passing a function's static chain pointer. If register windows are used, the register number as seen by the called function is `STATIC_CHAIN_INCOMING_REGNUM`, while the register number as seen by the calling function is `STATIC_CHAIN_REGNUM`. If these registers are the same, `STATIC_CHAIN_INCOMING_REGNUM` need not be defined.

The static chain register need not be a fixed register.

If the static chain is passed in memory, these macros should not be defined; instead, the next two macros should be defined.

`STATIC_CHAIN` [Macro]

`STATIC_CHAIN_INCOMING` [Macro]

If the static chain is passed in memory, these macros provide rtx giving `mem` expressions that denote where they are stored. `STATIC_CHAIN` and `STATIC_CHAIN_INCOMING` give the locations as seen by the calling and called functions, respectively. Often the former will be at an offset from the stack pointer and the latter at an offset from the frame pointer.

The variables `stack_pointer_rtx`, `frame_pointer_rtx`, and `arg_pointer_rtx` will have been initialized prior to the use of these macros and should be used to refer to those items.

If the static chain is passed in a register, the two previous macros should be defined instead.

`DWARF_FRAME_REGISTERS` [Macro]

This macro specifies the maximum number of hard registers that can be saved in a call frame. This is used to size data structures used in DWARF2 exception handling.

Prior to GCC 3.0, this macro was needed in order to establish a stable exception handling ABI in the face of adding new hard registers for ISA extensions. In GCC 3.0 and later, the EH ABI is insulated from changes in the number of hard registers. Nevertheless, this macro can still be used to reduce the runtime memory requirements of the exception handling routines, which can be substantial if the ISA contains a lot of registers that are not call-saved.

If this macro is not defined, it defaults to `FIRST_PSEUDO_REGISTER`.

`PRE_GCC3_DWARF_FRAME_REGISTERS` [Macro]

This macro is similar to `DWARF_FRAME_REGISTERS`, but is provided for backward compatibility in pre GCC 3.0 compiled code.

If this macro is not defined, it defaults to `DWARF_FRAME_REGISTERS`.

`DWARF_REG_TO_UNWIND_COLUMN (`*`regno`*`)` [Macro]

Define this macro if the target's representation for dwarf registers is different than the internal representation for unwind column. Given a dwarf register, this macro should return the internal unwind column number to use instead.

See the PowerPC's SPE target for an example.

`DWARF_FRAME_REGNUM (`*`regno`*`)` [Macro]

Define this macro if the target's representation for dwarf registers used in .eh_frame or .debug_frame is different from that used in other debug info sections. Given a GCC hard register number, this macro should return the .eh_frame register number. The default is `DBX_REGISTER_NUMBER (`*`regno`*`)`.

`DWARF2_FRAME_REG_OUT (`*`regno`*`, `*`for_eh`*`)` [Macro]

Define this macro to map register numbers held in the call frame info that GCC has collected using `DWARF_FRAME_REGNUM` to those that should be output in .debug_frame (*`for_eh`* is zero) and .eh_frame (*`for_eh`* is nonzero). The default is to return *`regno`*.

14.9.5 Eliminating Frame Pointer and Arg Pointer

This is about eliminating the frame pointer and arg pointer.

`FRAME_POINTER_REQUIRED` [Macro]

A C expression which is nonzero if a function must have and use a frame pointer. This expression is evaluated in the reload pass. If its value is nonzero the function will have a frame pointer.

The expression can in principle examine the current function and decide according to the facts, but on most machines the constant 0 or the constant 1 suffices. Use 0 when

the machine allows code to be generated with no frame pointer, and doing so saves some time or space. Use 1 when there is no possible advantage to avoiding a frame pointer.

In certain cases, the compiler does not know how to produce valid code without a frame pointer. The compiler recognizes those cases and automatically gives the function a frame pointer regardless of what `FRAME_POINTER_REQUIRED` says. You don't need to worry about them.

In a function that does not require a frame pointer, the frame pointer register can be allocated for ordinary usage, unless you mark it as a fixed register. See `FIXED_REGISTERS` for more information.

`INITIAL_FRAME_POINTER_OFFSET (`*depth-var*`)` [Macro]

A C statement to store in the variable *depth-var* the difference between the frame pointer and the stack pointer values immediately after the function prologue. The value would be computed from information such as the result of `get_frame_size ()` and the tables of registers `regs_ever_live` and `call_used_regs`.

If `ELIMINABLE_REGS` is defined, this macro will be not be used and need not be defined. Otherwise, it must be defined even if `FRAME_POINTER_REQUIRED` is defined to always be true; in that case, you may set *depth-var* to anything.

`ELIMINABLE_REGS` [Macro]

If defined, this macro specifies a table of register pairs used to eliminate unneeded registers that point into the stack frame. If it is not defined, the only elimination attempted by the compiler is to replace references to the frame pointer with references to the stack pointer.

The definition of this macro is a list of structure initializations, each of which specifies an original and replacement register.

On some machines, the position of the argument pointer is not known until the compilation is completed. In such a case, a separate hard register must be used for the argument pointer. This register can be eliminated by replacing it with either the frame pointer or the argument pointer, depending on whether or not the frame pointer has been eliminated.

In this case, you might specify:

```
#define ELIMINABLE_REGS  \
{{ARG_POINTER_REGNUM, STACK_POINTER_REGNUM}, \
 {ARG_POINTER_REGNUM, FRAME_POINTER_REGNUM}, \
 {FRAME_POINTER_REGNUM, STACK_POINTER_REGNUM}}
```

Note that the elimination of the argument pointer with the stack pointer is specified first since that is the preferred elimination.

`CAN_ELIMINATE (`*from-reg*, *to-reg*`)` [Macro]

A C expression that returns nonzero if the compiler is allowed to try to replace register number *from-reg* with register number *to-reg*. This macro need only be defined if `ELIMINABLE_REGS` is defined, and will usually be the constant 1, since most of the cases preventing register elimination are things that the compiler already knows about.

`INITIAL_ELIMINATION_OFFSET (from-reg, to-reg, offset-var)` [Macro]
This macro is similar to `INITIAL_FRAME_POINTER_OFFSET`. It specifies the initial difference between the specified pair of registers. This macro must be defined if `ELIMINABLE_REGS` is defined.

14.9.6 Passing Function Arguments on the Stack

The macros in this section control how arguments are passed on the stack. See the following section for other macros that control passing certain arguments in registers.

`bool TARGET_PROMOTE_PROTOTYPES (tree fntype)` [Target Hook]
This target hook returns `true` if an argument declared in a prototype as an integral type smaller than `int` should actually be passed as an `int`. In addition to avoiding errors in certain cases of mismatch, it also makes for better code on certain machines. The default is to not promote prototypes.

`PUSH_ARGS` [Macro]
A C expression. If nonzero, push insns will be used to pass outgoing arguments. If the target machine does not have a push instruction, set it to zero. That directs GCC to use an alternate strategy: to allocate the entire argument block and then store the arguments into it. When `PUSH_ARGS` is nonzero, `PUSH_ROUNDING` must be defined too.

`PUSH_ARGS_REVERSED` [Macro]
A C expression. If nonzero, function arguments will be evaluated from last to first, rather than from first to last. If this macro is not defined, it defaults to `PUSH_ARGS` on targets where the stack and args grow in opposite directions, and 0 otherwise.

`PUSH_ROUNDING (npushed)` [Macro]
A C expression that is the number of bytes actually pushed onto the stack when an instruction attempts to push *npushed* bytes.

On some machines, the definition

```
#define PUSH_ROUNDING(BYTES) (BYTES)
```

will suffice. But on other machines, instructions that appear to push one byte actually push two bytes in an attempt to maintain alignment. Then the definition should be

```
#define PUSH_ROUNDING(BYTES) (((BYTES) + 1) & ~1)
```

`ACCUMULATE_OUTGOING_ARGS` [Macro]
A C expression. If nonzero, the maximum amount of space required for outgoing arguments will be computed and placed into the variable `current_function_outgoing_args_size`. No space will be pushed onto the stack for each call; instead, the function prologue should increase the stack frame size by this amount.

Setting both `PUSH_ARGS` and `ACCUMULATE_OUTGOING_ARGS` is not proper.

`REG_PARM_STACK_SPACE (fndecl)` [Macro]
Define this macro if functions should assume that stack space has been allocated for arguments even when their values are passed in registers.

The value of this macro is the size, in bytes, of the area reserved for arguments passed in registers for the function represented by *fndecl*, which can be zero if GCC is calling a library function.

This space can be allocated by the caller, or be a part of the machine-dependent stack frame: `OUTGOING_REG_PARM_STACK_SPACE` says which.

`OUTGOING_REG_PARM_STACK_SPACE` [Macro]

Define this if it is the responsibility of the caller to allocate the area reserved for arguments passed in registers.

If `ACCUMULATE_OUTGOING_ARGS` is defined, this macro controls whether the space for these arguments counts in the value of `current_function_outgoing_args_size`.

`STACK_PARMS_IN_REG_PARM_AREA` [Macro]

Define this macro if `REG_PARM_STACK_SPACE` is defined, but the stack parameters don't skip the area specified by it.

Normally, when a parameter is not passed in registers, it is placed on the stack beyond the `REG_PARM_STACK_SPACE` area. Defining this macro suppresses this behavior and causes the parameter to be passed on the stack in its natural location.

`RETURN_POPS_ARGS (`*`fundecl`*`, `*`funtype`*`, `*`stack-size`*`)` [Macro]

A C expression that should indicate the number of bytes of its own arguments that a function pops on returning, or 0 if the function pops no arguments and the caller must therefore pop them all after the function returns.

fundecl is a C variable whose value is a tree node that describes the function in question. Normally it is a node of type `FUNCTION_DECL` that describes the declaration of the function. From this you can obtain the `DECL_ATTRIBUTES` of the function.

funtype is a C variable whose value is a tree node that describes the function in question. Normally it is a node of type `FUNCTION_TYPE` that describes the data type of the function. From this it is possible to obtain the data types of the value and arguments (if known).

When a call to a library function is being considered, *fundecl* will contain an identifier node for the library function. Thus, if you need to distinguish among various library functions, you can do so by their names. Note that "library function" in this context means a function used to perform arithmetic, whose name is known specially in the compiler and was not mentioned in the C code being compiled.

stack-size is the number of bytes of arguments passed on the stack. If a variable number of bytes is passed, it is zero, and argument popping will always be the responsibility of the calling function.

On the VAX, all functions always pop their arguments, so the definition of this macro is *stack-size*. On the 68000, using the standard calling convention, no functions pop their arguments, so the value of the macro is always 0 in this case. But an alternative calling convention is available in which functions that take a fixed number of arguments pop them but other functions (such as `printf`) pop nothing (the caller pops all). When this convention is in use, *funtype* is examined to determine whether a function takes a fixed number of arguments.

`CALL_POPS_ARGS (`*`cum`*`)` [Macro]

A C expression that should indicate the number of bytes a call sequence pops off the stack. It is added to the value of `RETURN_POPS_ARGS` when compiling a function call.

cum is the variable in which all arguments to the called function have been accumulated.

On certain architectures, such as the SH5, a call trampoline is used that pops certain registers off the stack, depending on the arguments that have been passed to the function. Since this is a property of the call site, not of the called function, `RETURN_POPS_ARGS` is not appropriate.

14.9.7 Passing Arguments in Registers

This section describes the macros which let you control how various types of arguments are passed in registers or how they are arranged in the stack.

`FUNCTION_ARG (`*cum*`, `*mode*`, `*type*`, `*named*`)` [Macro]

A C expression that controls whether a function argument is passed in a register, and which register.

The arguments are *cum*, which summarizes all the previous arguments; *mode*, the machine mode of the argument; *type*, the data type of the argument as a tree node or 0 if that is not known (which happens for C support library functions); and *named*, which is 1 for an ordinary argument and 0 for nameless arguments that correspond to '`...`' in the called function's prototype. *type* can be an incomplete type if a syntax error has previously occurred.

The value of the expression is usually either a `reg` RTX for the hard register in which to pass the argument, or zero to pass the argument on the stack.

For machines like the VAX and 68000, where normally all arguments are pushed, zero suffices as a definition.

The value of the expression can also be a `parallel` RTX. This is used when an argument is passed in multiple locations. The mode of the `parallel` should be the mode of the entire argument. The `parallel` holds any number of `expr_list` pairs; each one describes where part of the argument is passed. In each `expr_list` the first operand must be a `reg` RTX for the hard register in which to pass this part of the argument, and the mode of the register RTX indicates how large this part of the argument is. The second operand of the `expr_list` is a `const_int` which gives the offset in bytes into the entire argument of where this part starts. As a special exception the first `expr_list` in the `parallel` RTX may have a first operand of zero. This indicates that the entire argument is also stored on the stack.

The last time this macro is called, it is called with `MODE == VOIDmode`, and its result is passed to the `call` or `call_value` pattern as operands 2 and 3 respectively.

The usual way to make the ISO library '`stdarg.h`' work on a machine where some arguments are usually passed in registers, is to cause nameless arguments to be passed on the stack instead. This is done by making `FUNCTION_ARG` return 0 whenever *named* is 0.

You may use the hook `targetm.calls.must_pass_in_stack` in the definition of this macro to determine if this argument is of a type that must be passed in the stack. If `REG_PARM_STACK_SPACE` is not defined and `FUNCTION_ARG` returns nonzero for such an argument, the compiler will abort. If `REG_PARM_STACK_SPACE` is defined, the argument will be computed in the stack and then loaded into a register.

`bool TARGET_MUST_PASS_IN_STACK (`*enum machine_mode* **`mode`**`,` *tree* **`type`**`)` [Target Hook]

This target hook should return `true` if we should not pass *type* solely in registers. The file '`expr.h`' defines a definition that is usually appropriate, refer to '`expr.h`' for additional documentation.

`FUNCTION_INCOMING_ARG (`***`cum`***`,` ***`mode`***`,` ***`type`***`,` ***`named`***`)` [Macro]

Define this macro if the target machine has "register windows", so that the register in which a function sees an arguments is not necessarily the same as the one in which the caller passed the argument.

For such machines, `FUNCTION_ARG` computes the register in which the caller passes the value, and `FUNCTION_INCOMING_ARG` should be defined in a similar fashion to tell the function being called where the arguments will arrive.

If `FUNCTION_INCOMING_ARG` is not defined, `FUNCTION_ARG` serves both purposes.

`int TARGET_ARG_PARTIAL_BYTES (`*CUMULATIVE_ARGS* ***`*cum`***`,` *enum machine_mode* **`mode`**`,` *tree* **`type`**`,` *bool* **`named`**`)` [Target Hook]

This target hook returns the number of bytes at the beginning of an argument that must be put in registers. The value must be zero for arguments that are passed entirely in registers or that are entirely pushed on the stack.

On some machines, certain arguments must be passed partially in registers and partially in memory. On these machines, typically the first few words of arguments are passed in registers, and the rest on the stack. If a multi-word argument (a `double` or a structure) crosses that boundary, its first few words must be passed in registers and the rest must be pushed. This macro tells the compiler when this occurs, and how many bytes should go in registers.

`FUNCTION_ARG` for these arguments should return the first register to be used by the caller for this argument; likewise `FUNCTION_INCOMING_ARG`, for the called function.

`bool TARGET_PASS_BY_REFERENCE (`*CUMULATIVE_ARGS* ***`*cum`***`,` *enum machine_mode* **`mode`**`,` *tree* **`type`**`,` *bool* **`named`**`)` [Target Hook]

This target hook should return `true` if an argument at the position indicated by *cum* should be passed by reference. This predicate is queried after target independent reasons for being passed by reference, such as `TREE_ADDRESSABLE (type)`.

If the hook returns true, a copy of that argument is made in memory and a pointer to the argument is passed instead of the argument itself. The pointer is passed in whatever way is appropriate for passing a pointer to that type.

`bool TARGET_CALLEE_COPIES (`*CUMULATIVE_ARGS* ***`*cum`***`,` *enum machine_mode* **`mode`**`,` *tree* **`type`**`,` *bool* **`named`**`)` [Target Hook]

The function argument described by the parameters to this hook is known to be passed by reference. The hook should return true if the function argument should be copied by the callee instead of copied by the caller.

For any argument for which the hook returns true, if it can be determined that the argument is not modified, then a copy need not be generated.

The default version of this hook always returns false.

`CUMULATIVE_ARGS` [Macro]

A C type for declaring a variable that is used as the first argument of `FUNCTION_ARG` and other related values. For some target machines, the type `int` suffices and can hold the number of bytes of argument so far.

There is no need to record in `CUMULATIVE_ARGS` anything about the arguments that have been passed on the stack. The compiler has other variables to keep track of that. For target machines on which all arguments are passed on the stack, there is no need to store anything in `CUMULATIVE_ARGS`; however, the data structure must exist and should not be empty, so use `int`.

`INIT_CUMULATIVE_ARGS (`*`cum`*`, `*`fntype`*`, `*`libname`*`, `*`fndecl`*`,` [Macro]
`  `*`n_named_args`*`)`

A C statement (sans semicolon) for initializing the variable *cum* for the state at the beginning of the argument list. The variable has type `CUMULATIVE_ARGS`. The value of *fntype* is the tree node for the data type of the function which will receive the args, or 0 if the args are to a compiler support library function. For direct calls that are not libcalls, *fndecl* contain the declaration node of the function. *fndecl* is also set when `INIT_CUMULATIVE_ARGS` is used to find arguments for the function being compiled. *n_named_args* is set to the number of named arguments, including a structure return address if it is passed as a parameter, when making a call. When processing incoming arguments, *n_named_args* is set to -1.

When processing a call to a compiler support library function, *libname* identifies which one. It is a `symbol_ref` rtx which contains the name of the function, as a string. *libname* is 0 when an ordinary C function call is being processed. Thus, each time this macro is called, either *libname* or *fntype* is nonzero, but never both of them at once.

`INIT_CUMULATIVE_LIBCALL_ARGS (`*`cum`*`, `*`mode`*`, `*`libname`*`)` [Macro]

Like `INIT_CUMULATIVE_ARGS` but only used for outgoing libcalls, it gets a `MODE` argument instead of *fntype*, that would be `NULL`. *indirect* would always be zero, too. If this macro is not defined, `INIT_CUMULATIVE_ARGS (cum, NULL_RTX, libname, 0)` is used instead.

`INIT_CUMULATIVE_INCOMING_ARGS (`*`cum`*`, `*`fntype`*`, `*`libname`*`)` [Macro]

Like `INIT_CUMULATIVE_ARGS` but overrides it for the purposes of finding the arguments for the function being compiled. If this macro is undefined, `INIT_CUMULATIVE_ARGS` is used instead.

The value passed for *libname* is always 0, since library routines with special calling conventions are never compiled with GCC. The argument *libname* exists for symmetry with `INIT_CUMULATIVE_ARGS`.

`FUNCTION_ARG_ADVANCE (`*`cum`*`, `*`mode`*`, `*`type`*`, `*`named`*`)` [Macro]

A C statement (sans semicolon) to update the summarizer variable *cum* to advance past an argument in the argument list. The values *mode*, *type* and *named* describe that argument. Once this is done, the variable *cum* is suitable for analyzing the *following* argument with `FUNCTION_ARG`, etc.

This macro need not do anything if the argument in question was passed on the stack. The compiler knows how to track the amount of stack space used for arguments without any special help.

`FUNCTION_ARG_PADDING` (*mode*, *type*) [Macro]
If defined, a C expression which determines whether, and in which direction, to pad out an argument with extra space. The value should be of type `enum direction`: either `upward` to pad above the argument, `downward` to pad below, or `none` to inhibit padding.

The *amount* of padding is always just enough to reach the next multiple of `FUNCTION_ARG_BOUNDARY`; this macro does not control it.

This macro has a default definition which is right for most systems. For little-endian machines, the default is to pad upward. For big-endian machines, the default is to pad downward for an argument of constant size shorter than an `int`, and upward otherwise.

`PAD_VARARGS_DOWN` [Macro]
If defined, a C expression which determines whether the default implementation of va_arg will attempt to pad down before reading the next argument, if that argument is smaller than its aligned space as controlled by `PARM_BOUNDARY`. If this macro is not defined, all such arguments are padded down if `BYTES_BIG_ENDIAN` is true.

`BLOCK_REG_PADDING` (*mode*, *type*, *first*) [Macro]
Specify padding for the last element of a block move between registers and memory. *first* is nonzero if this is the only element. Defining this macro allows better control of register function parameters on big-endian machines, without using `PARALLEL` rtl. In particular, `MUST_PASS_IN_STACK` need not test padding and mode of types in registers, as there is no longer a "wrong" part of a register; For example, a three byte aggregate may be passed in the high part of a register if so required.

`FUNCTION_ARG_BOUNDARY` (*mode*, *type*) [Macro]
If defined, a C expression that gives the alignment boundary, in bits, of an argument with the specified mode and type. If it is not defined, `PARM_BOUNDARY` is used for all arguments.

`FUNCTION_ARG_REGNO_P` (*regno*) [Macro]
A C expression that is nonzero if *regno* is the number of a hard register in which function arguments are sometimes passed. This does *not* include implicit arguments such as the static chain and the structure-value address. On many machines, no registers can be used for this purpose since all function arguments are pushed on the stack.

`bool TARGET_SPLIT_COMPLEX_ARG` (*tree type*) [Target Hook]
This hook should return true if parameter of type *type* are passed as two scalar parameters. By default, GCC will attempt to pack complex arguments into the target's word size. Some ABIs require complex arguments to be split and treated as their individual components. For example, on AIX64, complex floats should be passed in a pair of floating point registers, even though a complex float would fit in one 64-bit floating point register.

The default value of this hook is `NULL`, which is treated as always false.

`tree TARGET_BUILD_BUILTIN_VA_LIST (void)` [Target Hook]
This hook returns a type node for `va_list` for the target. The default version of the hook returns `void*`.

`tree TARGET_GIMPLIFY_VA_ARG_EXPR (tree valist, tree type, tree *pre_p, tree *post_p)` [Target Hook]
This hook performs target-specific gimplification of `VA_ARG_EXPR`. The first two parameters correspond to the arguments to `va_arg`; the latter two are as in `gimplify.c:gimplify_expr`.

`bool TARGET_VALID_POINTER_MODE (enum machine_mode mode)` [Target Hook]
Define this to return nonzero if the port can handle pointers with machine mode *mode*. The default version of this hook returns true for both `ptr_mode` and `Pmode`.

`bool TARGET_SCALAR_MODE_SUPPORTED_P (enum machine_mode mode)` [Target Hook]
Define this to return nonzero if the port is prepared to handle insns involving scalar mode *mode*. For a scalar mode to be considered supported, all the basic arithmetic and comparisons must work.

The default version of this hook returns true for any mode required to handle the basic C types (as defined by the port). Included here are the double-word arithmetic supported by the code in '`optabs.c`'.

`bool TARGET_VECTOR_MODE_SUPPORTED_P (enum machine_mode mode)` [Target Hook]
Define this to return nonzero if the port is prepared to handle insns involving vector mode *mode*. At the very least, it must have move patterns for this mode.

14.9.8 How Scalar Function Values Are Returned

This section discusses the macros that control returning scalars as values—values that can fit in registers.

`FUNCTION_VALUE (valtype, func)` [Macro]
A C expression to create an RTX representing the place where a function returns a value of data type *valtype*. *valtype* is a tree node representing a data type. Write `TYPE_MODE (valtype)` to get the machine mode used to represent that type. On many machines, only the mode is relevant. (Actually, on most machines, scalar values are returned in the same place regardless of mode).

The value of the expression is usually a `reg` RTX for the hard register where the return value is stored. The value can also be a `parallel` RTX, if the return value is in multiple places. See `FUNCTION_ARG` for an explanation of the `parallel` form.

If `TARGET_PROMOTE_FUNCTION_RETURN` returns true, you must apply the same promotion rules specified in `PROMOTE_MODE` if *valtype* is a scalar type.

If the precise function being called is known, *func* is a tree node (`FUNCTION_DECL`) for it; otherwise, *func* is a null pointer. This makes it possible to use a different value-returning convention for specific functions when all their calls are known.

`FUNCTION_VALUE` is not used for return vales with aggregate data types, because these are returned in another way. See `TARGET_STRUCT_VALUE_RTX` and related macros, below.

`FUNCTION_OUTGOING_VALUE (`*`valtype`*`, `*`func`*`)` [Macro]

Define this macro if the target machine has "register windows" so that the register in which a function returns its value is not the same as the one in which the caller sees the value.

For such machines, `FUNCTION_VALUE` computes the register in which the caller will see the value. `FUNCTION_OUTGOING_VALUE` should be defined in a similar fashion to tell the function where to put the value.

If `FUNCTION_OUTGOING_VALUE` is not defined, `FUNCTION_VALUE` serves both purposes.

`FUNCTION_OUTGOING_VALUE` is not used for return vales with aggregate data types, because these are returned in another way. See `TARGET_STRUCT_VALUE_RTX` and related macros, below.

`LIBCALL_VALUE (`*`mode`*`)` [Macro]

A C expression to create an RTX representing the place where a library function returns a value of mode *mode*. If the precise function being called is known, *func* is a tree node (`FUNCTION_DECL`) for it; otherwise, *func* is a null pointer. This makes it possible to use a different value-returning convention for specific functions when all their calls are known.

Note that "library function" in this context means a compiler support routine, used to perform arithmetic, whose name is known specially by the compiler and was not mentioned in the C code being compiled.

The definition of `LIBRARY_VALUE` need not be concerned aggregate data types, because none of the library functions returns such types.

`FUNCTION_VALUE_REGNO_P (`*`regno`*`)` [Macro]

A C expression that is nonzero if *regno* is the number of a hard register in which the values of called function may come back.

A register whose use for returning values is limited to serving as the second of a pair (for a value of type `double`, say) need not be recognized by this macro. So for most machines, this definition suffices:

```
#define FUNCTION_VALUE_REGNO_P(N) ((N) == 0)
```

If the machine has register windows, so that the caller and the called function use different registers for the return value, this macro should recognize only the caller's register numbers.

`APPLY_RESULT_SIZE` [Macro]

Define this macro if '`untyped_call`' and '`untyped_return`' need more space than is implied by `FUNCTION_VALUE_REGNO_P` for saving and restoring an arbitrary return value.

`bool TARGET_RETURN_IN_MSB (`*`tree type`*`)` [Target Hook]

This hook should return true if values of type *type* are returned at the most significant end of a register (in other words, if they are padded at the least significant end). You can assume that *type* is returned in a register; the caller is required to check this.

Note that the register provided by `FUNCTION_VALUE` must be able to hold the complete return value. For example, if a 1-, 2- or 3-byte structure is returned at the most significant end of a 4-byte register, `FUNCTION_VALUE` should provide an `SImode` rtx.

14.9.9 How Large Values Are Returned

When a function value's mode is `BLKmode` (and in some other cases), the value is not returned according to `FUNCTION_VALUE` (see Section 14.9.8 [Scalar Return], page 320). Instead, the caller passes the address of a block of memory in which the value should be stored. This address is called the *structure value address*.

This section describes how to control returning structure values in memory.

`bool TARGET_RETURN_IN_MEMORY (`*tree* ***type***, *tree* ***fntype***`)` [Target Hook]
This target hook should return a nonzero value to say to return the function value in memory, just as large structures are always returned. Here *type* will be the data type of the value, and *fntype* will be the type of the function doing the returning, or `NULL` for libcalls.

Note that values of mode `BLKmode` must be explicitly handled by this function. Also, the option '`-fpcc-struct-return`' takes effect regardless of this macro. On most systems, it is possible to leave the hook undefined; this causes a default definition to be used, whose value is the constant 1 for `BLKmode` values, and 0 otherwise.

Do not use this hook to indicate that structures and unions should always be returned in memory. You should instead use `DEFAULT_PCC_STRUCT_RETURN` to indicate this.

`DEFAULT_PCC_STRUCT_RETURN` [Macro]
Define this macro to be 1 if all structure and union return values must be in memory. Since this results in slower code, this should be defined only if needed for compatibility with other compilers or with an ABI. If you define this macro to be 0, then the conventions used for structure and union return values are decided by the `TARGET_RETURN_IN_MEMORY` target hook.

If not defined, this defaults to the value 1.

`rtx TARGET_STRUCT_VALUE_RTX (`*tree* ***fndecl***, *int* ***incoming***`)` [Target Hook]
This target hook should return the location of the structure value address (normally a `mem` or `reg`), or 0 if the address is passed as an "invisible" first argument. Note that *fndecl* may be `NULL`, for libcalls. You do not need to define this target hook if the address is always passed as an "invisible" first argument.

On some architectures the place where the structure value address is found by the called function is not the same place that the caller put it. This can be due to register windows, or it could be because the function prologue moves it to a different place. *incoming* is `true` when the location is needed in the context of the called function, and `false` in the context of the caller.

If *incoming* is `true` and the address is to be found on the stack, return a `mem` which refers to the frame pointer.

`PCC_STATIC_STRUCT_RETURN` [Macro]
Define this macro if the usual system convention on the target machine for returning structures and unions is for the called function to return the address of a static variable containing the value.

Do not define this if the usual system convention is for the caller to pass an address to the subroutine.

This macro has effect in '`-fpcc-struct-return`' mode, but it does nothing when you use '`-freg-struct-return`' mode.

14.9.10 Caller-Saves Register Allocation

If you enable it, GCC can save registers around function calls. This makes it possible to use call-clobbered registers to hold variables that must live across calls.

`CALLER_SAVE_PROFITABLE (`*`refs`*`, `*`calls`*`)` [Macro]

A C expression to determine whether it is worthwhile to consider placing a pseudo-register in a call-clobbered hard register and saving and restoring it around each function call. The expression should be 1 when this is worth doing, and 0 otherwise.

If you don't define this macro, a default is used which is good on most machines: `4 * calls < refs`.

`HARD_REGNO_CALLER_SAVE_MODE (`*`regno`*`, `*`nregs`*`)` [Macro]

A C expression specifying which mode is required for saving *nregs* of a pseudo-register in call-clobbered hard register *regno*. If *regno* is unsuitable for caller save, `VOIDmode` should be returned. For most machines this macro need not be defined since GCC will select the smallest suitable mode.

14.9.11 Function Entry and Exit

This section describes the macros that output function entry (*prologue*) and exit (*epilogue*) code.

`void TARGET_ASM_FUNCTION_PROLOGUE (`*FILE *file*`,` [Target Hook]
HOST_WIDE_INT size`)`

If defined, a function that outputs the assembler code for entry to a function. The prologue is responsible for setting up the stack frame, initializing the frame pointer register, saving registers that must be saved, and allocating *size* additional bytes of storage for the local variables. *size* is an integer. *file* is a stdio stream to which the assembler code should be output.

The label for the beginning of the function need not be output by this macro. That has already been done when the macro is run.

To determine which registers to save, the macro can refer to the array `regs_ever_live`: element *r* is nonzero if hard register *r* is used anywhere within the function. This implies the function prologue should save register *r*, provided it is not one of the call-used registers. (`TARGET_ASM_FUNCTION_EPILOGUE` must likewise use `regs_ever_live`.)

On machines that have "register windows", the function entry code does not save on the stack the registers that are in the windows, even if they are supposed to be preserved by function calls; instead it takes appropriate steps to "push" the register stack, if any non-call-used registers are used in the function.

On machines where functions may or may not have frame-pointers, the function entry code must vary accordingly; it must set up the frame pointer if one is wanted, and not

otherwise. To determine whether a frame pointer is in wanted, the macro can refer to the variable `frame_pointer_needed`. The variable's value will be 1 at run time in a function that needs a frame pointer. See Section 14.9.5 [Elimination], page 312.

The function entry code is responsible for allocating any stack space required for the function. This stack space consists of the regions listed below. In most cases, these regions are allocated in the order listed, with the last listed region closest to the top of the stack (the lowest address if `STACK_GROWS_DOWNWARD` is defined, and the highest address if it is not defined). You can use a different order for a machine if doing so is more convenient or required for compatibility reasons. Except in cases where required by standard or by a debugger, there is no reason why the stack layout used by GCC need agree with that used by other compilers for a machine.

`void TARGET_ASM_FUNCTION_END_PROLOGUE (`*FILE *`file`*`)` [Target Hook]
If defined, a function that outputs assembler code at the end of a prologue. This should be used when the function prologue is being emitted as RTL, and you have some extra assembler that needs to be emitted. See [prologue instruction pattern], page 228.

`void TARGET_ASM_FUNCTION_BEGIN_EPILOGUE (`*FILE *`file`*`)` [Target Hook]
If defined, a function that outputs assembler code at the start of an epilogue. This should be used when the function epilogue is being emitted as RTL, and you have some extra assembler that needs to be emitted. See [epilogue instruction pattern], page 228.

`void TARGET_ASM_FUNCTION_EPILOGUE (`*FILE *`file`*, *HOST_WIDE_INT `size`*`)` [Target Hook]
If defined, a function that outputs the assembler code for exit from a function. The epilogue is responsible for restoring the saved registers and stack pointer to their values when the function was called, and returning control to the caller. This macro takes the same arguments as the macro `TARGET_ASM_FUNCTION_PROLOGUE`, and the registers to restore are determined from `regs_ever_live` and `CALL_USED_REGISTERS` in the same way.

On some machines, there is a single instruction that does all the work of returning from the function. On these machines, give that instruction the name '`return`' and do not define the macro `TARGET_ASM_FUNCTION_EPILOGUE` at all.

Do not define a pattern named '`return`' if you want the `TARGET_ASM_FUNCTION_EPILOGUE` to be used. If you want the target switches to control whether return instructions or epilogues are used, define a '`return`' pattern with a validity condition that tests the target switches appropriately. If the '`return`' pattern's validity condition is false, epilogues will be used.

On machines where functions may or may not have frame-pointers, the function exit code must vary accordingly. Sometimes the code for these two cases is completely different. To determine whether a frame pointer is wanted, the macro can refer to the variable `frame_pointer_needed`. The variable's value will be 1 when compiling a function that needs a frame pointer.

Normally, `TARGET_ASM_FUNCTION_PROLOGUE` and `TARGET_ASM_FUNCTION_EPILOGUE` must treat leaf functions specially. The C variable `current_function_is_leaf` is nonzero for such a function. See Section 14.7.4 [Leaf Functions], page 294.

On some machines, some functions pop their arguments on exit while others leave that for the caller to do. For example, the 68020 when given '`-mrtd`' pops arguments in functions that take a fixed number of arguments.

Your definition of the macro `RETURN_POPS_ARGS` decides which functions pop their own arguments. `TARGET_ASM_FUNCTION_EPILOGUE` needs to know what was decided. The variable that is called `current_function_pops_args` is the number of bytes of its arguments that a function should pop. See Section 14.9.8 [Scalar Return], page 320.

- A region of `current_function_pretend_args_size` bytes of uninitialized space just underneath the first argument arriving on the stack. (This may not be at the very start of the allocated stack region if the calling sequence has pushed anything else since pushing the stack arguments. But usually, on such machines, nothing else has been pushed yet, because the function prologue itself does all the pushing.) This region is used on machines where an argument may be passed partly in registers and partly in memory, and, in some cases to support the features in `<stdarg.h>`.
- An area of memory used to save certain registers used by the function. The size of this area, which may also include space for such things as the return address and pointers to previous stack frames, is machine-specific and usually depends on which registers have been used in the function. Machines with register windows often do not require a save area.
- A region of at least *size* bytes, possibly rounded up to an allocation boundary, to contain the local variables of the function. On some machines, this region and the save area may occur in the opposite order, with the save area closer to the top of the stack.
- Optionally, when `ACCUMULATE_OUTGOING_ARGS` is defined, a region of `current_function_outgoing_args_size` bytes to be used for outgoing argument lists of the function. See Section 14.9.6 [Stack Arguments], page 314.

`EXIT_IGNORE_STACK` [Macro]

Define this macro as a C expression that is nonzero if the return instruction or the function epilogue ignores the value of the stack pointer; in other words, if it is safe to delete an instruction to adjust the stack pointer before a return from the function. The default is 0.

Note that this macro's value is relevant only for functions for which frame pointers are maintained. It is never safe to delete a final stack adjustment in a function that has no frame pointer, and the compiler knows this regardless of `EXIT_IGNORE_STACK`.

`EPILOGUE_USES` (*regno*) [Macro]

Define this macro as a C expression that is nonzero for registers that are used by the epilogue or the '`return`' pattern. The stack and frame pointer registers are already be assumed to be used as needed.

`EH_USES` (*regno*) [Macro]

Define this macro as a C expression that is nonzero for registers that are used by the exception handling mechanism, and so should be considered live on entry to an exception edge.

`DELAY_SLOTS_FOR_EPILOGUE` [Macro]

Define this macro if the function epilogue contains delay slots to which instructions from the rest of the function can be "moved". The definition should be a C expression whose value is an integer representing the number of delay slots there.

`ELIGIBLE_FOR_EPILOGUE_DELAY (`*`insn`*`, n)` [Macro]

A C expression that returns 1 if *insn* can be placed in delay slot number *n* of the epilogue.

The argument *n* is an integer which identifies the delay slot now being considered (since different slots may have different rules of eligibility). It is never negative and is always less than the number of epilogue delay slots (what `DELAY_SLOTS_FOR_EPILOGUE` returns). If you reject a particular insn for a given delay slot, in principle, it may be reconsidered for a subsequent delay slot. Also, other insns may (at least in principle) be considered for the so far unfilled delay slot.

The insns accepted to fill the epilogue delay slots are put in an RTL list made with `insn_list` objects, stored in the variable `current_function_epilogue_delay_list`. The insn for the first delay slot comes first in the list. Your definition of the macro `TARGET_ASM_FUNCTION_EPILOGUE` should fill the delay slots by outputting the insns in this list, usually by calling `final_scan_insn`.

You need not define this macro if you did not define `DELAY_SLOTS_FOR_EPILOGUE`.

`void TARGET_ASM_OUTPUT_MI_THUNK (`*FILE *`file`*`, tree thunk_fndecl, HOST_WIDE_INT delta, HOST_WIDE_INT vcall_offset, tree function`*`)` [Target Hook]

A function that outputs the assembler code for a thunk function, used to implement C++ virtual function calls with multiple inheritance. The thunk acts as a wrapper around a virtual function, adjusting the implicit object parameter before handing control off to the real function.

First, emit code to add the integer *delta* to the location that contains the incoming first argument. Assume that this argument contains a pointer, and is the one used to pass the `this` pointer in C++. This is the incoming argument *before* the function prologue, e.g. '`%o0`' on a sparc. The addition must preserve the values of all other incoming arguments.

Then, if *vcall_offset* is nonzero, an additional adjustment should be made after adding `delta`. In particular, if *p* is the adjusted pointer, the following adjustment should be made:

```
p += (*((ptrdiff_t **)p))[vcall_offset/sizeof(ptrdiff_t)]
```

After the additions, emit code to jump to *function*, which is a `FUNCTION_DECL`. This is a direct pure jump, not a call, and does not touch the return address. Hence returning from *FUNCTION* will return to whoever called the current '`thunk`'.

The effect must be as if *function* had been called directly with the adjusted first argument. This macro is responsible for emitting all of the code for a thunk function; `TARGET_ASM_FUNCTION_PROLOGUE` and `TARGET_ASM_FUNCTION_EPILOGUE` are not invoked.

The *thunk_fndecl* is redundant. (*delta* and *function* have already been extracted from it.) It might possibly be useful on some targets, but probably not.

If you do not define this macro, the target-independent code in the C++ front end will generate a less efficient heavyweight thunk that calls *function* instead of jumping to it. The generic approach does not support varargs.

`bool TARGET_ASM_CAN_OUTPUT_MI_THUNK (`*tree* `thunk_fndecl`, *HOST_WIDE_INT* `delta`, *HOST_WIDE_INT* `vcall_offset`, *tree* `function``)` [Target Hook]

A function that returns true if TARGET_ASM_OUTPUT_MI_THUNK would be able to output the assembler code for the thunk function specified by the arguments it is passed, and false otherwise. In the latter case, the generic approach will be used by the C++ front end, with the limitations previously exposed.

14.9.12 Generating Code for Profiling

These macros will help you generate code for profiling.

`FUNCTION_PROFILER (`*file*, *labelno*`)` [Macro]

A C statement or compound statement to output to *file* some assembler code to call the profiling subroutine `mcount`.

The details of how `mcount` expects to be called are determined by your operating system environment, not by GCC. To figure them out, compile a small program for profiling using the system's installed C compiler and look at the assembler code that results.

Older implementations of `mcount` expect the address of a counter variable to be loaded into some register. The name of this variable is '`LP`' followed by the number *labelno*, so you would generate the name using '`LP%d`' in a `fprintf`.

`PROFILE_HOOK` [Macro]

A C statement or compound statement to output to *file* some assembly code to call the profiling subroutine `mcount` even the target does not support profiling.

`NO_PROFILE_COUNTERS` [Macro]

Define this macro if the `mcount` subroutine on your system does not need a counter variable allocated for each function. This is true for almost all modern implementations. If you define this macro, you must not use the *labelno* argument to `FUNCTION_PROFILER`.

`PROFILE_BEFORE_PROLOGUE` [Macro]

Define this macro if the code for function profiling should come before the function prologue. Normally, the profiling code comes after.

14.9.13 Permitting tail calls

`bool TARGET_FUNCTION_OK_FOR_SIBCALL (`*tree* `decl`, *tree* `exp``)` [Target Hook]

True if it is ok to do sibling call optimization for the specified call expression *exp*. *decl* will be the called function, or `NULL` if this is an indirect call.

It is not uncommon for limitations of calling conventions to prevent tail calls to functions outside the current unit of translation, or during PIC compilation. The hook is used to enforce these restrictions, as the `sibcall` md pattern can not fail, or

fall over to a "normal" call. The criteria for successful sibling call optimization may vary greatly between different architectures.

14.9.14 Stack smashing protection

`tree TARGET_STACK_PROTECT_GUARD (void)` [Target Hook]

This hook returns a `DECL` node for the external variable to use for the stack protection guard. This variable is initialized by the runtime to some random value and is used to initialize the guard value that is placed at the top of the local stack frame. The type of this variable must be `ptr_type_node`.

The default version of this hook creates a variable called '`__stack_chk_guard`', which is normally defined in '`libgcc2.c`'.

`tree TARGET_STACK_PROTECT_FAIL (void)` [Target Hook]

This hook returns a tree expression that alerts the runtime that the stack protect guard variable has been modified. This expression should involve a call to a `noreturn` function.

The default version of this hook invokes a function called '`__stack_chk_fail`', taking no arguments. This function is normally defined in '`libgcc2.c`'.

14.10 Implementing the Varargs Macros

GCC comes with an implementation of `<varargs.h>` and `<stdarg.h>` that work without change on machines that pass arguments on the stack. Other machines require their own implementations of varargs, and the two machine independent header files must have conditionals to include it.

ISO `<stdarg.h>` differs from traditional `<varargs.h>` mainly in the calling convention for `va_start`. The traditional implementation takes just one argument, which is the variable in which to store the argument pointer. The ISO implementation of `va_start` takes an additional second argument. The user is supposed to write the last named argument of the function here.

However, `va_start` should not use this argument. The way to find the end of the named arguments is with the built-in functions described below.

`__builtin_saveregs ()` [Macro]

Use this built-in function to save the argument registers in memory so that the varargs mechanism can access them. Both ISO and traditional versions of `va_start` must use `__builtin_saveregs`, unless you use `TARGET_SETUP_INCOMING_VARARGS` (see below) instead.

On some machines, `__builtin_saveregs` is open-coded under the control of the target hook `TARGET_EXPAND_BUILTIN_SAVEREGS`. On other machines, it calls a routine written in assembler language, found in '`libgcc2.c`'.

Code generated for the call to `__builtin_saveregs` appears at the beginning of the function, as opposed to where the call to `__builtin_saveregs` is written, regardless of what the code is. This is because the registers must be saved before the function starts to use them for its own purposes.

`__builtin_args_info (category)` [Macro]
Use this built-in function to find the first anonymous arguments in registers.

In general, a machine may have several categories of registers used for arguments, each for a particular category of data types. (For example, on some machines, floating-point registers are used for floating-point arguments while other arguments are passed in the general registers.) To make non-varargs functions use the proper calling convention, you have defined the `CUMULATIVE_ARGS` data type to record how many registers in each category have been used so far

`__builtin_args_info` accesses the same data structure of type `CUMULATIVE_ARGS` after the ordinary argument layout is finished with it, with *category* specifying which word to access. Thus, the value indicates the first unused register in a given category.

Normally, you would use `__builtin_args_info` in the implementation of `va_start`, accessing each category just once and storing the value in the `va_list` object. This is because `va_list` will have to update the values, and there is no way to alter the values accessed by `__builtin_args_info`.

`__builtin_next_arg (lastarg)` [Macro]
This is the equivalent of `__builtin_args_info`, for stack arguments. It returns the address of the first anonymous stack argument, as type `void *`. If `ARGS_GROW_DOWNWARD`, it returns the address of the location above the first anonymous stack argument. Use it in `va_start` to initialize the pointer for fetching arguments from the stack. Also use it in `va_start` to verify that the second parameter *lastarg* is the last named argument of the current function.

`__builtin_classify_type (object)` [Macro]
Since each machine has its own conventions for which data types are passed in which kind of register, your implementation of `va_arg` has to embody these conventions. The easiest way to categorize the specified data type is to use `__builtin_classify_type` together with `sizeof` and `__alignof__`.

`__builtin_classify_type` ignores the value of *object*, considering only its data type. It returns an integer describing what kind of type that is—integer, floating, pointer, structure, and so on.

The file '`typeclass.h`' defines an enumeration that you can use to interpret the values of `__builtin_classify_type`.

These machine description macros help implement varargs:

`rtx TARGET_EXPAND_BUILTIN_SAVEREGS (void)` [Target Hook]
If defined, this hook produces the machine-specific code for a call to `__builtin_saveregs`. This code will be moved to the very beginning of the function, before any parameter access are made. The return value of this function should be an RTX that contains the value to use as the return of `__builtin_saveregs`.

`void TARGET_SETUP_INCOMING_VARARGS (CUMULATIVE_ARGS *args_so_far, enum machine_mode mode, tree type, int *pretend_args_size, int second_time)` [Target Hook]
This target hook offers an alternative to using `__builtin_saveregs` and defining the hook `TARGET_EXPAND_BUILTIN_SAVEREGS`. Use it to store the anonymous register

arguments into the stack so that all the arguments appear to have been passed consecutively on the stack. Once this is done, you can use the standard implementation of varargs that works for machines that pass all their arguments on the stack.

The argument *args_so_far* points to the `CUMULATIVE_ARGS` data structure, containing the values that are obtained after processing the named arguments. The arguments *mode* and *type* describe the last named argument—its machine mode and its data type as a tree node.

The target hook should do two things: first, push onto the stack all the argument registers *not* used for the named arguments, and second, store the size of the data thus pushed into the `int`-valued variable pointed to by *pretend_args_size*. The value that you store here will serve as additional offset for setting up the stack frame.

Because you must generate code to push the anonymous arguments at compile time without knowing their data types, `TARGET_SETUP_INCOMING_VARARGS` is only useful on machines that have just a single category of argument register and use it uniformly for all data types.

If the argument *second_time* is nonzero, it means that the arguments of the function are being analyzed for the second time. This happens for an inline function, which is not actually compiled until the end of the source file. The hook `TARGET_SETUP_INCOMING_VARARGS` should not generate any instructions in this case.

`bool TARGET_STRICT_ARGUMENT_NAMING (`*CUMULATIVE_ARGS* `*ca)` [Target Hook]

Define this hook to return `true` if the location where a function argument is passed depends on whether or not it is a named argument.

This hook controls how the *named* argument to `FUNCTION_ARG` is set for varargs and stdarg functions. If this hook returns `true`, the *named* argument is always true for named arguments, and false for unnamed arguments. If it returns `false`, but `TARGET_PRETEND_OUTGOING_VARARGS_NAMED` returns `true`, then all arguments are treated as named. Otherwise, all named arguments except the last are treated as named.

You need not define this hook if it always returns zero.

`bool TARGET_PRETEND_OUTGOING_VARARGS_NAMED` [Target Hook]

If you need to conditionally change ABIs so that one works with `TARGET_SETUP_INCOMING_VARARGS`, but the other works like neither `TARGET_SETUP_INCOMING_VARARGS` nor `TARGET_STRICT_ARGUMENT_NAMING` was defined, then define this hook to return `true` if `TARGET_SETUP_INCOMING_VARARGS` is used, `false` otherwise. Otherwise, you should not define this hook.

14.11 Trampolines for Nested Functions

A *trampoline* is a small piece of code that is created at run time when the address of a nested function is taken. It normally resides on the stack, in the stack frame of the containing function. These macros tell GCC how to generate code to allocate and initialize a trampoline.

The instructions in the trampoline must do two things: load a constant address into the static chain register, and jump to the real address of the nested function. On CISC

machines such as the m68k, this requires two instructions, a move immediate and a jump. Then the two addresses exist in the trampoline as word-long immediate operands. On RISC machines, it is often necessary to load each address into a register in two parts. Then pieces of each address form separate immediate operands.

The code generated to initialize the trampoline must store the variable parts—the static chain value and the function address—into the immediate operands of the instructions. On a CISC machine, this is simply a matter of copying each address to a memory reference at the proper offset from the start of the trampoline. On a RISC machine, it may be necessary to take out pieces of the address and store them separately.

`TRAMPOLINE_TEMPLATE (`*`file`*`)` [Macro]

A C statement to output, on the stream *file*, assembler code for a block of data that contains the constant parts of a trampoline. This code should not include a label—the label is taken care of automatically.

If you do not define this macro, it means no template is needed for the target. Do not define this macro on systems where the block move code to copy the trampoline into place would be larger than the code to generate it on the spot.

`TRAMPOLINE_SECTION` [Macro]

The name of a subroutine to switch to the section in which the trampoline template is to be placed (see Section 14.17 [Sections], page 349). The default is a value of '`readonly_data_section`', which places the trampoline in the section containing read-only data.

`TRAMPOLINE_SIZE` [Macro]

A C expression for the size in bytes of the trampoline, as an integer.

`TRAMPOLINE_ALIGNMENT` [Macro]

Alignment required for trampolines, in bits.

If you don't define this macro, the value of `BIGGEST_ALIGNMENT` is used for aligning trampolines.

`INITIALIZE_TRAMPOLINE (`*`addr`*`, `*`fnaddr`*`, `*`static_chain`*`)` [Macro]

A C statement to initialize the variable parts of a trampoline. *addr* is an RTX for the address of the trampoline; *fnaddr* is an RTX for the address of the nested function; *static_chain* is an RTX for the static chain value that should be passed to the function when it is called.

`TRAMPOLINE_ADJUST_ADDRESS (`*`addr`*`)` [Macro]

A C statement that should perform any machine-specific adjustment in the address of the trampoline. Its argument contains the address that was passed to `INITIALIZE_TRAMPOLINE`. In case the address to be used for a function call should be different from the address in which the template was stored, the different address should be assigned to *addr*. If this macro is not defined, *addr* will be used for function calls.

If this macro is not defined, by default the trampoline is allocated as a stack slot. This default is right for most machines. The exceptions are machines where it is impossible to execute instructions in the stack area. On such machines, you may

have to implement a separate stack, using this macro in conjunction with `TARGET_ASM_FUNCTION_PROLOGUE` and `TARGET_ASM_FUNCTION_EPILOGUE`.

fp points to a data structure, a `struct function`, which describes the compilation status of the immediate containing function of the function which the trampoline is for. The stack slot for the trampoline is in the stack frame of this containing function. Other allocation strategies probably must do something analogous with this information.

Implementing trampolines is difficult on many machines because they have separate instruction and data caches. Writing into a stack location fails to clear the memory in the instruction cache, so when the program jumps to that location, it executes the old contents.

Here are two possible solutions. One is to clear the relevant parts of the instruction cache whenever a trampoline is set up. The other is to make all trampolines identical, by having them jump to a standard subroutine. The former technique makes trampoline execution faster; the latter makes initialization faster.

To clear the instruction cache when a trampoline is initialized, define the following macro.

`CLEAR_INSN_CACHE (`*beg*`, `*end*`)` [Macro]
If defined, expands to a C expression clearing the *instruction cache* in the specified interval. The definition of this macro would typically be a series of `asm` statements. Both *beg* and *end* are both pointer expressions.

The operating system may also require the stack to be made executable before calling the trampoline. To implement this requirement, define the following macro.

`ENABLE_EXECUTE_STACK` [Macro]
Define this macro if certain operations must be performed before executing code located on the stack. The macro should expand to a series of C file-scope constructs (e.g. functions) and provide a unique entry point named `__enable_execute_stack`. The target is responsible for emitting calls to the entry point in the code, for example from the `INITIALIZE_TRAMPOLINE` macro.

To use a standard subroutine, define the following macro. In addition, you must make sure that the instructions in a trampoline fill an entire cache line with identical instructions, or else ensure that the beginning of the trampoline code is always aligned at the same point in its cache line. Look in '`m68k.h`' as a guide.

`TRANSFER_FROM_TRAMPOLINE` [Macro]
Define this macro if trampolines need a special subroutine to do their work. The macro should expand to a series of `asm` statements which will be compiled with GCC. They go in a library function named `__transfer_from_trampoline`.

If you need to avoid executing the ordinary prologue code of a compiled C function when you jump to the subroutine, you can do so by placing a special label of your own in the assembler code. Use one `asm` statement to generate an assembler label, and another to make the label global. Then trampolines can use that label to jump directly to your special assembler code.

14.12 Implicit Calls to Library Routines

Here is an explanation of implicit calls to library routines.

`DECLARE_LIBRARY_RENAMES` [Macro]

This macro, if defined, should expand to a piece of C code that will get expanded when compiling functions for libgcc.a. It can be used to provide alternate names for GCC's internal library functions if there are ABI-mandated names that the compiler should provide.

`void TARGET_INIT_LIBFUNCS (`*void*`)` [Target Hook]

This hook should declare additional library routines or rename existing ones, using the functions `set_optab_libfunc` and `init_one_libfunc` defined in '`optabs.c`'. `init_optabs` calls this macro after initializing all the normal library routines.

The default is to do nothing. Most ports don't need to define this hook.

`FLOAT_LIB_COMPARE_RETURNS_BOOL (`*mode*`, `*comparison*`)` [Macro]

This macro should return `true` if the library routine that implements the floating point comparison operator *comparison* in mode *mode* will return a boolean, and *false* if it will return a tristate.

GCC's own floating point libraries return tristates from the comparison operators, so the default returns false always. Most ports don't need to define this macro.

`TARGET_LIB_INT_CMP_BIASED` [Macro]

This macro should evaluate to `true` if the integer comparison functions (like `__cmpdi2`) return 0 to indicate that the first operand is smaller than the second, 1 to indicate that they are equal, and 2 to indicate that the first operand is greater than the second. If this macro evaluates to `false` the comparison functions return −1, 0, and 1 instead of 0, 1, and 2. If the target uses the routines in '`libgcc.a`', you do not need to define this macro.

`US_SOFTWARE_GOFAST` [Macro]

Define this macro if your system C library uses the US Software GOFAST library to provide floating point emulation.

In addition to defining this macro, your architecture must set `TARGET_INIT_LIBFUNCS` to `gofast_maybe_init_libfuncs`, or else call that function from its version of that hook. It is defined in '`config/gofast.h`', which must be included by your architecture's '*cpu*`.c`' file. See '`sparc/sparc.c`' for an example.

If this macro is defined, the `TARGET_FLOAT_LIB_COMPARE_RETURNS_BOOL` target hook must return false for `SFmode` and `DFmode` comparisons.

`TARGET_EDOM` [Macro]

The value of `EDOM` on the target machine, as a C integer constant expression. If you don't define this macro, GCC does not attempt to deposit the value of `EDOM` into `errno` directly. Look in '`/usr/include/errno.h`' to find the value of `EDOM` on your system.

If you do not define `TARGET_EDOM`, then compiled code reports domain errors by calling the library function and letting it report the error. If mathematical functions on your system use `matherr` when there is an error, then you should leave `TARGET_EDOM` undefined so that `matherr` is used normally.

`GEN_ERRNO_RTX` [Macro]
Define this macro as a C expression to create an rtl expression that refers to the global "variable" `errno`. (On certain systems, `errno` may not actually be a variable.) If you don't define this macro, a reasonable default is used.

`TARGET_C99_FUNCTIONS` [Macro]
When this macro is nonzero, GCC will implicitly optimize `sin` calls into `sinf` and similarly for other functions defined by C99 standard. The default is nonzero that should be proper value for most modern systems, however number of existing systems lacks support for these functions in the runtime so they needs this macro to be redefined to 0.

`NEXT_OBJC_RUNTIME` [Macro]
Define this macro to generate code for Objective-C message sending using the calling convention of the NeXT system. This calling convention involves passing the object, the selector and the method arguments all at once to the method-lookup library function.

The default calling convention passes just the object and the selector to the lookup function, which returns a pointer to the method.

14.13 Addressing Modes

This is about addressing modes.

`HAVE_PRE_INCREMENT` [Macro]
`HAVE_PRE_DECREMENT` [Macro]
`HAVE_POST_INCREMENT` [Macro]
`HAVE_POST_DECREMENT` [Macro]
A C expression that is nonzero if the machine supports pre-increment, pre-decrement, post-increment, or post-decrement addressing respectively.

`HAVE_PRE_MODIFY_DISP` [Macro]
`HAVE_POST_MODIFY_DISP` [Macro]
A C expression that is nonzero if the machine supports pre- or post-address side-effect generation involving constants other than the size of the memory operand.

`HAVE_PRE_MODIFY_REG` [Macro]
`HAVE_POST_MODIFY_REG` [Macro]
A C expression that is nonzero if the machine supports pre- or post-address side-effect generation involving a register displacement.

`CONSTANT_ADDRESS_P (`*x*`)` [Macro]
A C expression that is 1 if the RTX *x* is a constant which is a valid address. On most machines, this can be defined as `CONSTANT_P (x)`, but a few machines are more restrictive in which constant addresses are supported.

`CONSTANT_P (`*x*`)` [Macro]
`CONSTANT_P`, which is defined by target-independent code, accepts integer-values expressions whose values are not explicitly known, such as `symbol_ref`, `label_ref`, and `high` expressions and `const` arithmetic expressions, in addition to `const_int` and `const_double` expressions.

`MAX_REGS_PER_ADDRESS` [Macro]

A number, the maximum number of registers that can appear in a valid memory address. Note that it is up to you to specify a value equal to the maximum number that `GO_IF_LEGITIMATE_ADDRESS` would ever accept.

`GO_IF_LEGITIMATE_ADDRESS (`*mode*, *x*, *label*`)` [Macro]

A C compound statement with a conditional `goto` *label*`;` executed if *x* (an RTX) is a legitimate memory address on the target machine for a memory operand of mode *mode*.

It usually pays to define several simpler macros to serve as subroutines for this one. Otherwise it may be too complicated to understand.

This macro must exist in two variants: a strict variant and a non-strict one. The strict variant is used in the reload pass. It must be defined so that any pseudo-register that has not been allocated a hard register is considered a memory reference. In contexts where some kind of register is required, a pseudo-register with no hard register must be rejected.

The non-strict variant is used in other passes. It must be defined to accept all pseudo-registers in every context where some kind of register is required.

Compiler source files that want to use the strict variant of this macro define the macro `REG_OK_STRICT`. You should use an `#ifdef REG_OK_STRICT` conditional to define the strict variant in that case and the non-strict variant otherwise.

Subroutines to check for acceptable registers for various purposes (one for base registers, one for index registers, and so on) are typically among the subroutines used to define `GO_IF_LEGITIMATE_ADDRESS`. Then only these subroutine macros need have two variants; the higher levels of macros may be the same whether strict or not.

Normally, constant addresses which are the sum of a `symbol_ref` and an integer are stored inside a `const` RTX to mark them as constant. Therefore, there is no need to recognize such sums specifically as legitimate addresses. Normally you would simply recognize any `const` as legitimate.

Usually `PRINT_OPERAND_ADDRESS` is not prepared to handle constant sums that are not marked with `const`. It assumes that a naked `plus` indicates indexing. If so, then you *must* reject such naked constant sums as illegitimate addresses, so that none of them will be given to `PRINT_OPERAND_ADDRESS`.

On some machines, whether a symbolic address is legitimate depends on the section that the address refers to. On these machines, define the target hook `TARGET_ENCODE_SECTION_INFO` to store the information into the `symbol_ref`, and then check for it here. When you see a `const`, you will have to look inside it to find the `symbol_ref` in order to determine the section. See Section 14.19 [Assembler Format], page 353.

`REG_OK_FOR_BASE_P (`*x*`)` [Macro]

A C expression that is nonzero if *x* (assumed to be a `reg` RTX) is valid for use as a base register. For hard registers, it should always accept those which the hardware permits and reject the others. Whether the macro accepts or rejects pseudo registers must be controlled by `REG_OK_STRICT` as described above. This usually requires two variant definitions, of which `REG_OK_STRICT` controls the one actually used.

`REG_MODE_OK_FOR_BASE_P (x, mode)` [Macro]

A C expression that is just like `REG_OK_FOR_BASE_P`, except that that expression may examine the mode of the memory reference in *mode*. You should define this macro if the mode of the memory reference affects whether a register may be used as a base register. If you define this macro, the compiler will use it instead of `REG_OK_FOR_BASE_P`.

`REG_MODE_OK_FOR_REG_BASE_P (x, mode)` [Macro]

A C expression which is nonzero if *x* (assumed to be a `reg` RTX) is suitable for use as a base register in base plus index operand addresses, accessing memory in mode *mode*. It may be either a suitable hard register or a pseudo register that has been allocated such a hard register. You should define this macro if base plus index addresses have different requirements than other base register uses.

`REG_OK_FOR_INDEX_P (x)` [Macro]

A C expression that is nonzero if *x* (assumed to be a `reg` RTX) is valid for use as an index register.

The difference between an index register and a base register is that the index register may be scaled. If an address involves the sum of two registers, neither one of them scaled, then either one may be labeled the "base" and the other the "index"; but whichever labeling is used must fit the machine's constraints of which registers may serve in each capacity. The compiler will try both labelings, looking for one that is valid, and will reload one or both registers only if neither labeling works.

`FIND_BASE_TERM (x)` [Macro]

A C expression to determine the base term of address *x*. This macro is used in only one place: 'find_base_term' in alias.c.

It is always safe for this macro to not be defined. It exists so that alias analysis can understand machine-dependent addresses.

The typical use of this macro is to handle addresses containing a label_ref or symbol_ref within an UNSPEC.

`LEGITIMIZE_ADDRESS (x, oldx, mode, win)` [Macro]

A C compound statement that attempts to replace *x* with a valid memory address for an operand of mode *mode*. *win* will be a C statement label elsewhere in the code; the macro definition may use

```
GO_IF_LEGITIMATE_ADDRESS (mode, x, win);
```

to avoid further processing if the address has become legitimate.

x will always be the result of a call to `break_out_memory_refs`, and *oldx* will be the operand that was given to that function to produce *x*.

The code generated by this macro should not alter the substructure of *x*. If it transforms *x* into a more legitimate form, it should assign *x* (which will always be a C variable) a new value.

It is not necessary for this macro to come up with a legitimate address. The compiler has standard ways of doing so in all cases. In fact, it is safe to omit this macro. But often a machine-dependent strategy can generate better code.

`LEGITIMIZE_RELOAD_ADDRESS (x, mode, opnum, type, ind_levels, win)` [Macro]

A C compound statement that attempts to replace *x*, which is an address that needs reloading, with a valid memory address for an operand of mode *mode*. *win* will be a C statement label elsewhere in the code. It is not necessary to define this macro, but it might be useful for performance reasons.

For example, on the i386, it is sometimes possible to use a single reload register instead of two by reloading a sum of two pseudo registers into a register. On the other hand, for number of RISC processors offsets are limited so that often an intermediate address needs to be generated in order to address a stack slot. By defining `LEGITIMIZE_RELOAD_ADDRESS` appropriately, the intermediate addresses generated for adjacent some stack slots can be made identical, and thus be shared.

Note: This macro should be used with caution. It is necessary to know something of how reload works in order to effectively use this, and it is quite easy to produce macros that build in too much knowledge of reload internals.

Note: This macro must be able to reload an address created by a previous invocation of this macro. If it fails to handle such addresses then the compiler may generate incorrect code or abort.

The macro definition should use `push_reload` to indicate parts that need reloading; *opnum*, *type* and *ind_levels* are usually suitable to be passed unaltered to `push_reload`.

The code generated by this macro must not alter the substructure of *x*. If it transforms *x* into a more legitimate form, it should assign *x* (which will always be a C variable) a new value. This also applies to parts that you change indirectly by calling `push_reload`.

The macro definition may use `strict_memory_address_p` to test if the address has become legitimate.

If you want to change only a part of *x*, one standard way of doing this is to use `copy_rtx`. Note, however, that is unshares only a single level of rtl. Thus, if the part to be changed is not at the top level, you'll need to replace first the top level. It is not necessary for this macro to come up with a legitimate address; but often a machine-dependent strategy can generate better code.

`GO_IF_MODE_DEPENDENT_ADDRESS (addr, label)` [Macro]

A C statement or compound statement with a conditional `goto label;` executed if memory address *x* (an RTX) can have different meanings depending on the machine mode of the memory reference it is used for or if the address is valid for some modes but not others.

Autoincrement and autodecrement addresses typically have mode-dependent effects because the amount of the increment or decrement is the size of the operand being addressed. Some machines have other mode-dependent addresses. Many RISC machines have no mode-dependent addresses.

You may assume that *addr* is a valid address for the machine.

`LEGITIMATE_CONSTANT_P (x)` [Macro]

A C expression that is nonzero if *x* is a legitimate constant for an immediate operand on the target machine. You can assume that *x* satisfies `CONSTANT_P`, so you need

not check this. In fact, '1' is a suitable definition for this macro on machines where anything CONSTANT_P is valid.

rtx TARGET_DELEGITIMIZE_ADDRESS (*rtx x*) [Target Hook]
This hook is used to undo the possibly obfuscating effects of the LEGITIMIZE_ADDRESS and LEGITIMIZE_RELOAD_ADDRESS target macros. Some backend implementations of these macros wrap symbol references inside an UNSPEC rtx to represent PIC or similar addressing modes. This target hook allows GCC's optimizers to understand the semantics of these opaque UNSPECs by converting them back into their original form.

bool TARGET_CANNOT_FORCE_CONST_MEM (*rtx x*) [Target Hook]
This hook should return true if *x* is of a form that cannot (or should not) be spilled to the constant pool. The default version of this hook returns false.

The primary reason to define this hook is to prevent reload from deciding that a non-legitimate constant would be better reloaded from the constant pool instead of spilling and reloading a register holding the constant. This restriction is often true of addresses of TLS symbols for various targets.

tree TARGET_VECTORIZE_BUILTIN_MASK_FOR_LOAD (*void*) [Target Hook]
This hook should return the DECL of a function *f* that given an address *addr* as an argument returns a mask *m* that can be used to extract from two vectors the relevant data that resides in *addr* in case *addr* is not properly aligned.

The autovectrizer, when vectorizing a load operation from an address *addr* that may be unaligned, will generate two vector loads from the two aligned addresses around *addr*. It then generates a REALIGN_LOAD operation to extract the relevant data from the two loaded vectors. The first two arguments to REALIGN_LOAD, *v1* and *v2*, are the two vectors, each of size *VS*, and the third argument, *OFF*, defines how the data will be extracted from these two vectors: if *OFF* is 0, then the returned vector is *v2*; otherwise, the returned vector is composed from the last *VS*-*OFF* elements of *v1* concatenated to the first *OFF* elements of *v2*.

If this hook is defined, the autovectorizer will generate a call to *f* (using the DECL tree that this hook returns) and will use the return value of *f* as the argument *OFF* to REALIGN_LOAD. Therefore, the mask *m* returned by *f* should comply with the semantics expected by REALIGN_LOAD described above. If this hook is not defined, then *addr* will be used as the argument *OFF* to REALIGN_LOAD, in which case the low log2(*VS*)-1 bits of *addr* will be considered.

14.14 Condition Code Status

This describes the condition code status.

The file 'conditions.h' defines a variable cc_status to describe how the condition code was computed (in case the interpretation of the condition code depends on the instruction that it was set by). This variable contains the RTL expressions on which the condition code is currently based, and several standard flags.

Sometimes additional machine-specific flags must be defined in the machine description header file. It can also add additional machine-specific information by defining CC_STATUS_MDEP.

`CC_STATUS_MDEP` [Macro]

C code for a data type which is used for declaring the `mdep` component of `cc_status`. It defaults to `int`.

This macro is not used on machines that do not use `cc0`.

`CC_STATUS_MDEP_INIT` [Macro]

A C expression to initialize the `mdep` field to "empty". The default definition does nothing, since most machines don't use the field anyway. If you want to use the field, you should probably define this macro to initialize it.

This macro is not used on machines that do not use `cc0`.

`NOTICE_UPDATE_CC (`*exp*, *insn*`)` [Macro]

A C compound statement to set the components of `cc_status` appropriately for an insn *insn* whose body is *exp*. It is this macro's responsibility to recognize insns that set the condition code as a byproduct of other activity as well as those that explicitly set `(cc0)`.

This macro is not used on machines that do not use `cc0`.

If there are insns that do not set the condition code but do alter other machine registers, this macro must check to see whether they invalidate the expressions that the condition code is recorded as reflecting. For example, on the 68000, insns that store in address registers do not set the condition code, which means that usually `NOTICE_UPDATE_CC` can leave `cc_status` unaltered for such insns. But suppose that the previous insn set the condition code based on location '`a4@(102)`' and the current insn stores a new value in '`a4`'. Although the condition code is not changed by this, it will no longer be true that it reflects the contents of '`a4@(102)`'. Therefore, `NOTICE_UPDATE_CC` must alter `cc_status` in this case to say that nothing is known about the condition code value.

The definition of `NOTICE_UPDATE_CC` must be prepared to deal with the results of peephole optimization: insns whose patterns are `parallel` RTXs containing various `reg`, `mem` or constants which are just the operands. The RTL structure of these insns is not sufficient to indicate what the insns actually do. What `NOTICE_UPDATE_CC` should do when it sees one is just to run `CC_STATUS_INIT`.

A possible definition of `NOTICE_UPDATE_CC` is to call a function that looks at an attribute (see Section 13.19 [Insn Attributes], page 248) named, for example, '`cc`'. This avoids having detailed information about patterns in two places, the '`md`' file and in `NOTICE_UPDATE_CC`.

`SELECT_CC_MODE (`*op*, *x*, *y*`)` [Macro]

Returns a mode from class `MODE_CC` to be used when comparison operation code *op* is applied to rtx *x* and *y*. For example, on the SPARC, `SELECT_CC_MODE` is defined as (see see Section 13.12 [Jump Patterns], page 233 for a description of the reason for this definition)

```
#define SELECT_CC_MODE(OP,X,Y) \
  (GET_MODE_CLASS (GET_MODE (X)) == MODE_FLOAT          \
   ? ((OP == EQ || OP == NE) ? CCFPmode : CCFPEmode)    \
   : ((GET_CODE (X) == PLUS || GET_CODE (X) == MINUS    \
       || GET_CODE (X) == NEG) \
```

```
? CC_NOOVmode : CCmode))
```

You should define this macro if and only if you define extra CC modes in '*machine*-modes.def'.

CANONICALIZE_COMPARISON (*code*, *op0*, *op1*) [Macro]

On some machines not all possible comparisons are defined, but you can convert an invalid comparison into a valid one. For example, the Alpha does not have a GT comparison, but you can use an LT comparison instead and swap the order of the operands.

On such machines, define this macro to be a C statement to do any required conversions. *code* is the initial comparison code and *op0* and *op1* are the left and right operands of the comparison, respectively. You should modify *code*, *op0*, and *op1* as required.

GCC will not assume that the comparison resulting from this macro is valid but will see if the resulting insn matches a pattern in the 'md' file.

You need not define this macro if it would never change the comparison code or operands.

REVERSIBLE_CC_MODE (*mode*) [Macro]

A C expression whose value is one if it is always safe to reverse a comparison whose mode is *mode*. If SELECT_CC_MODE can ever return *mode* for a floating-point inequality comparison, then REVERSIBLE_CC_MODE (*mode*) must be zero.

You need not define this macro if it would always returns zero or if the floating-point format is anything other than IEEE_FLOAT_FORMAT. For example, here is the definition used on the SPARC, where floating-point inequality comparisons are always given CCFPEmode:

```
#define REVERSIBLE_CC_MODE(MODE)  ((MODE) != CCFPEmode)
```

REVERSE_CONDITION (*code*, *mode*) [Macro]

A C expression whose value is reversed condition code of the *code* for comparison done in CC_MODE *mode*. The macro is used only in case REVERSIBLE_CC_MODE (*mode*) is nonzero. Define this macro in case machine has some non-standard way how to reverse certain conditionals. For instance in case all floating point conditions are non-trapping, compiler may freely convert unordered compares to ordered one. Then definition may look like:

```
#define REVERSE_CONDITION(CODE, MODE) \
   ((MODE) != CCFPmode ? reverse_condition (CODE) \
    : reverse_condition_maybe_unordered (CODE))
```

REVERSE_CONDEXEC_PREDICATES_P (*op1*, *op2*) [Macro]

A C expression that returns true if the conditional execution predicate *op1*, a comparison operation, is the inverse of *op2* and vice versa. Define this to return 0 if the target has conditional execution predicates that cannot be reversed safely. There is no need to validate that the arguments of op1 and op2 are the same, this is done separately. If no expansion is specified, this macro is defined as follows:

```
#define REVERSE_CONDEXEC_PREDICATES_P (x, y) \
   (GET_CODE ((x)) == reversed_comparison_code ((y), NULL))
```

`bool TARGET_FIXED_CONDITION_CODE_REGS (`*unsigned int *, unsigned int **`)` [Target Hook]

On targets which do not use `(cc0)`, and which use a hard register rather than a pseudo-register to hold condition codes, the regular CSE passes are often not able to identify cases in which the hard register is set to a common value. Use this hook to enable a small pass which optimizes such cases. This hook should return true to enable this pass, and it should set the integers to which its arguments point to the hard register numbers used for condition codes. When there is only one such register, as is true on most systems, the integer pointed to by the second argument should be set to `INVALID_REGNUM`.

The default version of this hook returns false.

`enum machine_mode` *TARGET_CC_MODES_COMPATIBLE* (*enum machine_mode, enum machine_mode*) [Target Hook]

On targets which use multiple condition code modes in class `MODE_CC`, it is sometimes the case that a comparison can be validly done in more than one mode. On such a system, define this target hook to take two mode arguments and to return a mode in which both comparisons may be validly done. If there is no such mode, return `VOIDmode`.

The default version of this hook checks whether the modes are the same. If they are, it returns that mode. If they are different, it returns `VOIDmode`.

14.15 Describing Relative Costs of Operations

These macros let you describe the relative speed of various operations on the target machine.

`REGISTER_MOVE_COST (`*mode*, *from*, *to*`)` [Macro]

A C expression for the cost of moving data of mode *mode* from a register in class *from* to one in class *to*. The classes are expressed using the enumeration values such as `GENERAL_REGS`. A value of 2 is the default; other values are interpreted relative to that.

It is not required that the cost always equal 2 when *from* is the same as *to*; on some machines it is expensive to move between registers if they are not general registers.

If reload sees an insn consisting of a single `set` between two hard registers, and if `REGISTER_MOVE_COST` applied to their classes returns a value of 2, reload does not check to ensure that the constraints of the insn are met. Setting a cost of other than 2 will allow reload to verify that the constraints are met. You should do this if the '`movm`' pattern's constraints do not allow such copying.

`MEMORY_MOVE_COST (`*mode*, *class*, *in*`)` [Macro]

A C expression for the cost of moving data of mode *mode* between a register of class *class* and memory; *in* is zero if the value is to be written to memory, nonzero if it is to be read in. This cost is relative to those in `REGISTER_MOVE_COST`. If moving between registers and memory is more expensive than between two registers, you should define this macro to express the relative cost.

If you do not define this macro, GCC uses a default cost of 4 plus the cost of copying via a secondary reload register, if one is needed. If your machine requires a secondary

reload register to copy between memory and a register of *class* but the reload mechanism is more complex than copying via an intermediate, define this macro to reflect the actual cost of the move.

GCC defines the function `memory_move_secondary_cost` if secondary reloads are needed. It computes the costs due to copying via a secondary register. If your machine copies from memory using a secondary register in the conventional way but the default base value of 4 is not correct for your machine, define this macro to add some other value to the result of that function. The arguments to that function are the same as to this macro.

`BRANCH_COST` [Macro]

A C expression for the cost of a branch instruction. A value of 1 is the default; other values are interpreted relative to that.

Here are additional macros which do not specify precise relative costs, but only that certain actions are more expensive than GCC would ordinarily expect.

`SLOW_BYTE_ACCESS` [Macro]

Define this macro as a C expression which is nonzero if accessing less than a word of memory (i.e. a `char` or a `short`) is no faster than accessing a word of memory, i.e., if such access require more than one instruction or if there is no difference in cost between byte and (aligned) word loads.

When this macro is not defined, the compiler will access a field by finding the smallest containing object; when it is defined, a fullword load will be used if alignment permits. Unless bytes accesses are faster than word accesses, using word accesses is preferable since it may eliminate subsequent memory access if subsequent accesses occur to other fields in the same word of the structure, but to different bytes.

`SLOW_UNALIGNED_ACCESS` (*mode*, *alignment*) [Macro]

Define this macro to be the value 1 if memory accesses described by the *mode* and *alignment* parameters have a cost many times greater than aligned accesses, for example if they are emulated in a trap handler.

When this macro is nonzero, the compiler will act as if `STRICT_ALIGNMENT` were nonzero when generating code for block moves. This can cause significantly more instructions to be produced. Therefore, do not set this macro nonzero if unaligned accesses only add a cycle or two to the time for a memory access.

If the value of this macro is always zero, it need not be defined. If this macro is defined, it should produce a nonzero value when `STRICT_ALIGNMENT` is nonzero.

`MOVE_RATIO` [Macro]

The threshold of number of scalar memory-to-memory move insns, *below* which a sequence of insns should be generated instead of a string move insn or a library call. Increasing the value will always make code faster, but eventually incurs high cost in increased code size.

Note that on machines where the corresponding move insn is a `define_expand` that emits a sequence of insns, this macro counts the number of such sequences.

If you don't define this, a reasonable default is used.

`MOVE_BY_PIECES_P (size, alignment)` [Macro]
A C expression used to determine whether `move_by_pieces` will be used to copy a chunk of memory, or whether some other block move mechanism will be used. Defaults to 1 if `move_by_pieces_ninsns` returns less than `MOVE_RATIO`.

`MOVE_MAX_PIECES` [Macro]
A C expression used by `move_by_pieces` to determine the largest unit a load or store used to copy memory is. Defaults to `MOVE_MAX`.

`CLEAR_RATIO` [Macro]
The threshold of number of scalar move insns, *below* which a sequence of insns should be generated to clear memory instead of a string clear insn or a library call. Increasing the value will always make code faster, but eventually incurs high cost in increased code size.

If you don't define this, a reasonable default is used.

`CLEAR_BY_PIECES_P (size, alignment)` [Macro]
A C expression used to determine whether `clear_by_pieces` will be used to clear a chunk of memory, or whether some other block clear mechanism will be used. Defaults to 1 if `move_by_pieces_ninsns` returns less than `CLEAR_RATIO`.

`STORE_BY_PIECES_P (size, alignment)` [Macro]
A C expression used to determine whether `store_by_pieces` will be used to set a chunk of memory to a constant value, or whether some other mechanism will be used. Used by `__builtin_memset` when storing values other than constant zero and by `__builtin_strcpy` when when called with a constant source string. Defaults to 1 if `move_by_pieces_ninsns` returns less than `MOVE_RATIO`.

`USE_LOAD_POST_INCREMENT (mode)` [Macro]
A C expression used to determine whether a load postincrement is a good thing to use for a given mode. Defaults to the value of `HAVE_POST_INCREMENT`.

`USE_LOAD_POST_DECREMENT (mode)` [Macro]
A C expression used to determine whether a load postdecrement is a good thing to use for a given mode. Defaults to the value of `HAVE_POST_DECREMENT`.

`USE_LOAD_PRE_INCREMENT (mode)` [Macro]
A C expression used to determine whether a load preincrement is a good thing to use for a given mode. Defaults to the value of `HAVE_PRE_INCREMENT`.

`USE_LOAD_PRE_DECREMENT (mode)` [Macro]
A C expression used to determine whether a load predecrement is a good thing to use for a given mode. Defaults to the value of `HAVE_PRE_DECREMENT`.

`USE_STORE_POST_INCREMENT (mode)` [Macro]
A C expression used to determine whether a store postincrement is a good thing to use for a given mode. Defaults to the value of `HAVE_POST_INCREMENT`.

`USE_STORE_POST_DECREMENT (mode)` [Macro]
A C expression used to determine whether a store postdecrement is a good thing to use for a given mode. Defaults to the value of `HAVE_POST_DECREMENT`.

`USE_STORE_PRE_INCREMENT (mode)` [Macro]
This macro is used to determine whether a store preincrement is a good thing to use for a given mode. Defaults to the value of `HAVE_PRE_INCREMENT`.

`USE_STORE_PRE_DECREMENT (mode)` [Macro]
This macro is used to determine whether a store predecrement is a good thing to use for a given mode. Defaults to the value of `HAVE_PRE_DECREMENT`.

`NO_FUNCTION_CSE` [Macro]
Define this macro if it is as good or better to call a constant function address than to call an address kept in a register.

`RANGE_TEST_NON_SHORT_CIRCUIT` [Macro]
Define this macro if a non-short-circuit operation produced by '`fold_range_test ()`' is optimal. This macro defaults to true if `BRANCH_COST` is greater than or equal to the value 2.

`bool TARGET_RTX_COSTS (`*rtx* `x`, *int* `code`, *int* `outer_code`, *int* `*total``)` [Target Hook]
This target hook describes the relative costs of RTL expressions.

The cost may depend on the precise form of the expression, which is available for examination in *x*, and the rtx code of the expression in which it is contained, found in *outer_code*. *code* is the expression code—redundant, since it can be obtained with `GET_CODE (x)`.

In implementing this hook, you can use the construct `COSTS_N_INSNS (n)` to specify a cost equal to *n* fast instructions.

On entry to the hook, `*total` contains a default estimate for the cost of the expression. The hook should modify this value as necessary. Traditionally, the default costs are `COSTS_N_INSNS (5)` for multiplications, `COSTS_N_INSNS (7)` for division and modulus operations, and `COSTS_N_INSNS (1)` for all other operations.

When optimizing for code size, i.e. when `optimize_size` is nonzero, this target hook should be used to estimate the relative size cost of an expression, again relative to `COSTS_N_INSNS`.

The hook returns true when all subexpressions of *x* have been processed, and false when `rtx_cost` should recurse.

`int TARGET_ADDRESS_COST (`*rtx* `address`)` [Target Hook]
This hook computes the cost of an addressing mode that contains *address*. If not defined, the cost is computed from the *address* expression and the `TARGET_RTX_COST` hook.

For most CISC machines, the default cost is a good approximation of the true cost of the addressing mode. However, on RISC machines, all instructions normally have the same length and execution time. Hence all addresses will have equal costs.

In cases where more than one form of an address is known, the form with the lowest cost will be used. If multiple forms have the same, lowest, cost, the one that is the most complex will be used.

For example, suppose an address that is equal to the sum of a register and a constant is used twice in the same basic block. When this macro is not defined, the address will be computed in a register and memory references will be indirect through that register. On machines where the cost of the addressing mode containing the sum is no higher than that of a simple indirect reference, this will produce an additional instruction and possibly require an additional register. Proper specification of this macro eliminates this overhead for such machines.

This hook is never called with an invalid address.

On machines where an address involving more than one register is as cheap as an address computation involving only one register, defining `TARGET_ADDRESS_COST` to reflect this can cause two registers to be live over a region of code where only one would have been if `TARGET_ADDRESS_COST` were not defined in that manner. This effect should be considered in the definition of this macro. Equivalent costs should probably only be given to addresses with different numbers of registers on machines with lots of registers.

14.16 Adjusting the Instruction Scheduler

The instruction scheduler may need a fair amount of machine-specific adjustment in order to produce good code. GCC provides several target hooks for this purpose. It is usually enough to define just a few of them: try the first ones in this list first.

`int TARGET_SCHED_ISSUE_RATE (`*void*`)` [Target Hook]

This hook returns the maximum number of instructions that can ever issue at the same time on the target machine. The default is one. Although the insn scheduler can define itself the possibility of issue an insn on the same cycle, the value can serve as an additional constraint to issue insns on the same simulated processor cycle (see hooks '`TARGET_SCHED_REORDER`' and '`TARGET_SCHED_REORDER2`'). This value must be constant over the entire compilation. If you need it to vary depending on what the instructions are, you must use '`TARGET_SCHED_VARIABLE_ISSUE`'.

You could define this hook to return the value of the macro `MAX_DFA_ISSUE_RATE`.

`int TARGET_SCHED_VARIABLE_ISSUE (`*FILE *file*, *int* `verbose`, *rtx* `insn`, *int* `more``)` [Target Hook]

This hook is executed by the scheduler after it has scheduled an insn from the ready list. It should return the number of insns which can still be issued in the current cycle. The default is '`more - 1`' for insns other than `CLOBBER` and `USE`, which normally are not counted against the issue rate. You should define this hook if some insns take more machine resources than others, so that fewer insns can follow them in the same cycle. *file* is either a null pointer, or a stdio stream to write any debug output to. *verbose* is the verbose level provided by '`-fsched-verbose-n`'. *insn* is the instruction that was scheduled.

`int TARGET_SCHED_ADJUST_COST (`*rtx* `insn`, *rtx* `link`, *rtx* `dep_insn`, *int* `cost``)` [Target Hook]

This function corrects the value of *cost* based on the relationship between *insn* and *dep_insn* through the dependence *link*. It should return the new value. The default is

to make no adjustment to *cost*. This can be used for example to specify to the scheduler using the traditional pipeline description that an output- or anti-dependence does not incur the same cost as a data-dependence. If the scheduler using the automaton based pipeline description, the cost of anti-dependence is zero and the cost of output-dependence is maximum of one and the difference of latency times of the first and the second insns. If these values are not acceptable, you could use the hook to modify them too. See also see Section 13.19.8 [Processor pipeline description], page 256.

`int TARGET_SCHED_ADJUST_PRIORITY (`*rtx* `insn`, *int* `priority``)` [Target Hook]
This hook adjusts the integer scheduling priority *priority* of *insn*. It should return the new priority. Reduce the priority to execute *insn* earlier, increase the priority to execute *insn* later. Do not define this hook if you do not need to adjust the scheduling priorities of insns.

`int TARGET_SCHED_REORDER (`*FILE* `*file`, *int* `verbose`, *rtx* `*ready`, *int* `*n_readyp`, *int* `clock``)` [Target Hook]
This hook is executed by the scheduler after it has scheduled the ready list, to allow the machine description to reorder it (for example to combine two small instructions together on 'VLIW' machines). *file* is either a null pointer, or a stdio stream to write any debug output to. *verbose* is the verbose level provided by '`-fsched-verbose-n`'. *ready* is a pointer to the ready list of instructions that are ready to be scheduled. *n_readyp* is a pointer to the number of elements in the ready list. The scheduler reads the ready list in reverse order, starting with *ready*[**n_readyp*-1] and going to *ready*[0]. *clock* is the timer tick of the scheduler. You may modify the ready list and the number of ready insns. The return value is the number of insns that can issue this cycle; normally this is just `issue_rate`. See also '`TARGET_SCHED_REORDER2`'.

`int TARGET_SCHED_REORDER2 (`*FILE* `*file`, *int* `verbose`, *rtx* `*ready`, *int* `*n_ready`, `clock`)` [Target Hook]
Like '`TARGET_SCHED_REORDER`', but called at a different time. That function is called whenever the scheduler starts a new cycle. This one is called once per iteration over a cycle, immediately after '`TARGET_SCHED_VARIABLE_ISSUE`'; it can reorder the ready list and return the number of insns to be scheduled in the same cycle. Defining this hook can be useful if there are frequent situations where scheduling one insn causes other insns to become ready in the same cycle. These other insns can then be taken into account properly.

`void TARGET_SCHED_DEPENDENCIES_EVALUATION_HOOK (`*rtx* `head`, *rtx* `tail`)` [Target Hook]
This hook is called after evaluation forward dependencies of insns in chain given by two parameter values (*head* and *tail* correspondingly) but before insns scheduling of the insn chain. For example, it can be used for better insn classification if it requires analysis of dependencies. This hook can use backward and forward dependencies of the insn scheduler because they are already calculated.

`void TARGET_SCHED_INIT (`*FILE* `*file`, *int* `verbose`, *int* `max_ready`)` [Target Hook]
This hook is executed by the scheduler at the beginning of each block of instructions that are to be scheduled. *file* is either a null pointer, or a stdio stream to write

any debug output to. *verbose* is the verbose level provided by '`-fsched-verbose-n`'. *max_ready* is the maximum number of insns in the current scheduling region that can be live at the same time. This can be used to allocate scratch space if it is needed, e.g. by '`TARGET_SCHED_REORDER`'.

`void TARGET_SCHED_FINISH (`*FILE *`file`, *int* `verbose``)` [Target Hook]
This hook is executed by the scheduler at the end of each block of instructions that are to be scheduled. It can be used to perform cleanup of any actions done by the other scheduling hooks. *file* is either a null pointer, or a stdio stream to write any debug output to. *verbose* is the verbose level provided by '`-fsched-verbose-n`'.

`void TARGET_SCHED_INIT_GLOBAL (`*FILE *`file`, *int* `verbose`, *int* `old_max_uid``)` [Target Hook]
This hook is executed by the scheduler after function level initializations. *file* is either a null pointer, or a stdio stream to write any debug output to. *verbose* is the verbose level provided by '`-fsched-verbose-n`'. *old_max_uid* is the maximum insn uid when scheduling begins.

`void TARGET_SCHED_FINISH_GLOBAL (`*FILE *`file`, *int* `verbose``)` [Target Hook]
This is the cleanup hook corresponding to `TARGET_SCHED_INIT_GLOBAL`. *file* is either a null pointer, or a stdio stream to write any debug output to. *verbose* is the verbose level provided by '`-fsched-verbose-n`'.

`int TARGET_SCHED_DFA_PRE_CYCLE_INSN (`*void*`)` [Target Hook]
The hook returns an RTL insn. The automaton state used in the pipeline hazard recognizer is changed as if the insn were scheduled when the new simulated processor cycle starts. Usage of the hook may simplify the automaton pipeline description for some VLIW processors. If the hook is defined, it is used only for the automaton based pipeline description. The default is not to change the state when the new simulated processor cycle starts.

`void TARGET_SCHED_INIT_DFA_PRE_CYCLE_INSN (`*void*`)` [Target Hook]
The hook can be used to initialize data used by the previous hook.

`int TARGET_SCHED_DFA_POST_CYCLE_INSN (`*void*`)` [Target Hook]
The hook is analogous to '`TARGET_SCHED_DFA_PRE_CYCLE_INSN`' but used to changed the state as if the insn were scheduled when the new simulated processor cycle finishes.

`void TARGET_SCHED_INIT_DFA_POST_CYCLE_INSN (`*void*`)` [Target Hook]
The hook is analogous to '`TARGET_SCHED_INIT_DFA_PRE_CYCLE_INSN`' but used to initialize data used by the previous hook.

`int TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD (`*void*`)` [Target Hook]
This hook controls better choosing an insn from the ready insn queue for the DFA-based insn scheduler. Usually the scheduler chooses the first insn from the queue. If the hook returns a positive value, an additional scheduler code tries all permutations of '`TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD ()`' subsequent ready insns to choose an insn whose issue will result in maximal number of issued

insns on the same cycle. For the VLIW processor, the code could actually solve the problem of packing simple insns into the VLIW insn. Of course, if the rules of VLIW packing are described in the automaton.

This code also could be used for superscalar RISC processors. Let us consider a superscalar RISC processor with 3 pipelines. Some insns can be executed in pipelines A or B, some insns can be executed only in pipelines B or C, and one insn can be executed in pipeline B. The processor may issue the 1st insn into A and the 2nd one into B. In this case, the 3rd insn will wait for freeing B until the next cycle. If the scheduler issues the 3rd insn the first, the processor could issue all 3 insns per cycle.

Actually this code demonstrates advantages of the automaton based pipeline hazard recognizer. We try quickly and easy many insn schedules to choose the best one.

The default is no multipass scheduling.

`int` [Target Hook]
`TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD_GUARD` (*rtx*)

This hook controls what insns from the ready insn queue will be considered for the multipass insn scheduling. If the hook returns zero for insn passed as the parameter, the insn will be not chosen to be issued.

The default is that any ready insns can be chosen to be issued.

`int TARGET_SCHED_DFA_NEW_CYCLE` (*FILE *, int, rtx, int, int, int **) [Target Hook]

This hook is called by the insn scheduler before issuing insn passed as the third parameter on given cycle. If the hook returns nonzero, the insn is not issued on given processors cycle. Instead of that, the processor cycle is advanced. If the value passed through the last parameter is zero, the insn ready queue is not sorted on the new cycle start as usually. The first parameter passes file for debugging output. The second one passes the scheduler verbose level of the debugging output. The forth and the fifth parameter values are correspondingly processor cycle on which the previous insn has been issued and the current processor cycle.

`bool TARGET_SCHED_IS_COSTLY_DEPENDENCE` (*rtx* ***insn1***, *rtx* ***insn2***, *rtx* ***dep_link***, *int* ***dep_cost***, *int* ***distance***) [Target Hook]

This hook is used to define which dependences are considered costly by the target, so costly that it is not advisable to schedule the insns that are involved in the dependence too close to one another. The parameters to this hook are as follows: The second parameter *insn2* is dependent upon the first parameter *insn1*. The dependence between *insn1* and *insn2* is represented by the third parameter *dep_link*. The fourth parameter *cost* is the cost of the dependence, and the fifth parameter *distance* is the distance in cycles between the two insns. The hook returns `true` if considering the distance between the two insns the dependence between them is considered costly by the target, and `false` otherwise.

Defining this hook can be useful in multiple-issue out-of-order machines, where (a) it's practically hopeless to predict the actual data/resource delays, however: (b) there's a better chance to predict the actual grouping that will be formed, and (c) correctly emulating the grouping can be very important. In such targets one may want to allow issuing dependent insns closer to one another—i.e., closer than the dependence

distance; however, not in cases of "costly dependences", which this hooks allows to define.

Macros in the following table are generated by the program 'genattr' and can be useful for writing the hooks.

MAX_DFA_ISSUE_RATE [Macro]
The macro definition is generated in the automaton based pipeline description interface. Its value is calculated from the automaton based pipeline description and is equal to maximal number of all insns described in constructions 'define_insn_reservation' which can be issued on the same processor cycle.

14.17 Dividing the Output into Sections (Texts, Data, . . .)

An object file is divided into sections containing different types of data. In the most common case, there are three sections: the *text section*, which holds instructions and read-only data; the *data section*, which holds initialized writable data; and the *bss section*, which holds uninitialized data. Some systems have other kinds of sections.

The compiler must tell the assembler when to switch sections. These macros control what commands to output to tell the assembler this. You can also define additional sections.

TEXT_SECTION_ASM_OP [Macro]
A C expression whose value is a string, including spacing, containing the assembler operation that should precede instructions and read-only data. Normally "\t.text" is right.

HOT_TEXT_SECTION_NAME [Macro]
If defined, a C string constant for the name of the section containing most frequently executed functions of the program. If not defined, GCC will provide a default definition if the target supports named sections.

UNLIKELY_EXECUTED_TEXT_SECTION_NAME [Macro]
If defined, a C string constant for the name of the section containing unlikely executed functions in the program.

DATA_SECTION_ASM_OP [Macro]
A C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as writable initialized data. Normally "\t.data" is right.

READONLY_DATA_SECTION_ASM_OP [Macro]
A C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as read-only initialized data.

READONLY_DATA_SECTION [Macro]
A macro naming a function to call to switch to the proper section for read-only data. The default is to use READONLY_DATA_SECTION_ASM_OP if defined, else fall back to text_section.

The most common definition will be data_section, if the target does not have a special read-only data section, and does not put data in the text section.

`BSS_SECTION_ASM_OP` [Macro]
If defined, a C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as uninitialized global data. If not defined, and neither `ASM_OUTPUT_BSS` nor `ASM_OUTPUT_ALIGNED_BSS` are defined, uninitialized global data will be output in the data section if '`-fno-common`' is passed, otherwise `ASM_OUTPUT_COMMON` will be used.

`INIT_SECTION_ASM_OP` [Macro]
If defined, a C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as initialization code. If not defined, GCC will assume such a section does not exist.

`FINI_SECTION_ASM_OP` [Macro]
If defined, a C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as finalization code. If not defined, GCC will assume such a section does not exist.

`INIT_ARRAY_SECTION_ASM_OP` [Macro]
If defined, a C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as part of the `.init_array` (or equivalent) section. If not defined, GCC will assume such a section does not exist. Do not define both this macro and `INIT_SECTION_ASM_OP`.

`FINI_ARRAY_SECTION_ASM_OP` [Macro]
If defined, a C expression whose value is a string, including spacing, containing the assembler operation to identify the following data as part of the `.fini_array` (or equivalent) section. If not defined, GCC will assume such a section does not exist. Do not define both this macro and `FINI_SECTION_ASM_OP`.

`CRT_CALL_STATIC_FUNCTION (`*`section_op`*`, `*`function`*`)` [Macro]
If defined, an ASM statement that switches to a different section via *section_op*, calls *function*, and switches back to the text section. This is used in '`crtstuff.c`' if `INIT_SECTION_ASM_OP` or `FINI_SECTION_ASM_OP` to calls to initialization and finalization functions from the init and fini sections. By default, this macro uses a simple function call. Some ports need hand-crafted assembly code to avoid dependencies on registers initialized in the function prologue or to ensure that constant pools don't end up too far way in the text section.

`FORCE_CODE_SECTION_ALIGN` [Macro]
If defined, an ASM statement that aligns a code section to some arbitrary boundary. This is used to force all fragments of the `.init` and `.fini` sections to have to same alignment and thus prevent the linker from having to add any padding.

`EXTRA_SECTIONS` [Macro]
A list of names for sections other than the standard two, which are `in_text` and `in_data`. You need not define this macro on a system with no other sections (that GCC needs to use).

`EXTRA_SECTION_FUNCTIONS` [Macro]

One or more functions to be defined in '`varasm.c`'. These functions should do jobs analogous to those of `text_section` and `data_section`, for your additional sections. Do not define this macro if you do not define `EXTRA_SECTIONS`.

`JUMP_TABLES_IN_TEXT_SECTION` [Macro]

Define this macro to be an expression with a nonzero value if jump tables (for `tablejump` insns) should be output in the text section, along with the assembler instructions. Otherwise, the readonly data section is used.

This macro is irrelevant if there is no separate readonly data section.

`void TARGET_ASM_SELECT_SECTION (`*tree* `exp`, *int* `reloc`, *unsigned HOST_WIDE_INT* `align``)` [Target Hook]

Switches to the appropriate section for output of *exp*. You can assume that *exp* is either a `VAR_DECL` node or a constant of some sort. *reloc* indicates whether the initial value of *exp* requires link-time relocations. Bit 0 is set when variable contains local relocations only, while bit 1 is set for global relocations. Select the section by calling `data_section` or one of the alternatives for other sections. *align* is the constant alignment in bits.

The default version of this function takes care of putting read-only variables in `readonly_data_section`.

See also *USE_SELECT_SECTION_FOR_FUNCTIONS*.

`USE_SELECT_SECTION_FOR_FUNCTIONS` [Macro]

Define this macro if you wish TARGET_ASM_SELECT_SECTION to be called for `FUNCTION_DECL`s as well as for variables and constants.

In the case of a `FUNCTION_DECL`, *reloc* will be zero if the function has been determined to be likely to be called, and nonzero if it is unlikely to be called.

`void TARGET_ASM_UNIQUE_SECTION (`*tree* `decl`, *int* `reloc``)` [Target Hook]

Build up a unique section name, expressed as a `STRING_CST` node, and assign it to '`DECL_SECTION_NAME (decl)`'. As with `TARGET_ASM_SELECT_SECTION`, *reloc* indicates whether the initial value of *exp* requires link-time relocations.

The default version of this function appends the symbol name to the ELF section name that would normally be used for the symbol. For example, the function `foo` would be placed in `.text.foo`. Whatever the actual target object format, this is often good enough.

`void TARGET_ASM_FUNCTION_RODATA_SECTION (`*tree* `decl``)` [Target Hook]

Switches to a readonly data section associated with '`DECL_SECTION_NAME (decl)`'. The default version of this function switches to `.gnu.linkonce.r.name` section if function's section is `.gnu.linkonce.t.name`, to `.rodata.name` if function is in `.text.name` section and otherwise switches to the normal readonly data section.

`void TARGET_ASM_SELECT_RTX_SECTION (`*enum machine_mode* `mode`, *rtx* `x`, *unsigned HOST_WIDE_INT* `align``)` [Target Hook]

Switches to the appropriate section for output of constant pool entry *x* in *mode*. You can assume that *x* is some kind of constant in RTL. The argument *mode* is redundant

except in the case of a `const_int` rtx. Select the section by calling `readonly_data_section` or one of the alternatives for other sections. *align* is the constant alignment in bits.

The default version of this function takes care of putting symbolic constants in `flag_pic` mode in `data_section` and everything else in `readonly_data_section`.

`void TARGET_ENCODE_SECTION_INFO (`*tree* `decl`*, rtx* `rtl`*, int* `new_decl_p`*)* [Target Hook]

Define this hook if references to a symbol or a constant must be treated differently depending on something about the variable or function named by the symbol (such as what section it is in).

The hook is executed immediately after rtl has been created for *decl*, which may be a variable or function declaration or an entry in the constant pool. In either case, *rtl* is the rtl in question. Do *not* use `DECL_RTL (`*decl*`)` in this hook; that field may not have been initialized yet.

In the case of a constant, it is safe to assume that the rtl is a `mem` whose address is a `symbol_ref`. Most decls will also have this form, but that is not guaranteed. Global register variables, for instance, will have a `reg` for their rtl. (Normally the right thing to do with such unusual rtl is leave it alone.)

The *new_decl_p* argument will be true if this is the first time that `TARGET_ENCODE_SECTION_INFO` has been invoked on this decl. It will be false for subsequent invocations, which will happen for duplicate declarations. Whether or not anything must be done for the duplicate declaration depends on whether the hook examines `DECL_ATTRIBUTES`. *new_decl_p* is always true when the hook is called for a constant.

The usual thing for this hook to do is to record flags in the `symbol_ref`, using `SYMBOL_REF_FLAG` or `SYMBOL_REF_FLAGS`. Historically, the name string was modified if it was necessary to encode more than one bit of information, but this practice is now discouraged; use `SYMBOL_REF_FLAGS`.

The default definition of this hook, `default_encode_section_info` in '`varasm.c`', sets a number of commonly-useful bits in `SYMBOL_REF_FLAGS`. Check whether the default does what you need before overriding it.

`const char *`*TARGET_STRIP_NAME_ENCODING* `(`*const char* **name*`)` [Target Hook]

Decode *name* and return the real name part, sans the characters that `TARGET_ENCODE_SECTION_INFO` may have added.

`bool TARGET_IN_SMALL_DATA_P (`*tree* `exp`*)* [Target Hook]

Returns true if *exp* should be placed into a "small data" section. The default version of this hook always returns false.

`Target Hook bool` *TARGET_HAVE_SRODATA_SECTION* [Variable]

Contains the value true if the target places read-only "small data" into a separate section. The default value is false.

`bool TARGET_BINDS_LOCAL_P (`*tree* `exp`*)* [Target Hook]

Returns true if *exp* names an object for which name resolution rules must resolve to the current "module" (dynamic shared library or executable image).

The default version of this hook implements the name resolution rules for ELF, which has a looser model of global name binding than other currently supported object file formats.

`Target Hook bool` *TARGET_HAVE_TLS* [Variable]
Contains the value true if the target supports thread-local storage. The default value is false.

14.18 Position Independent Code

This section describes macros that help implement generation of position independent code. Simply defining these macros is not enough to generate valid PIC; you must also add support to the macros `GO_IF_LEGITIMATE_ADDRESS` and `PRINT_OPERAND_ADDRESS`, as well as `LEGITIMIZE_ADDRESS`. You must modify the definition of '`movsi`' to do something appropriate when the source operand contains a symbolic address. You may also need to alter the handling of switch statements so that they use relative addresses.

`PIC_OFFSET_TABLE_REGNUM` [Macro]
The register number of the register used to address a table of static data addresses in memory. In some cases this register is defined by a processor's "application binary interface" (ABI). When this macro is defined, RTL is generated for this register once, as with the stack pointer and frame pointer registers. If this macro is not defined, it is up to the machine-dependent files to allocate such a register (if necessary). Note that this register must be fixed when in use (e.g. when `flag_pic` is true).

`PIC_OFFSET_TABLE_REG_CALL_CLOBBERED` [Macro]
Define this macro if the register defined by `PIC_OFFSET_TABLE_REGNUM` is clobbered by calls. Do not define this macro if `PIC_OFFSET_TABLE_REGNUM` is not defined.

`LEGITIMATE_PIC_OPERAND_P (`*x*`)` [Macro]
A C expression that is nonzero if *x* is a legitimate immediate operand on the target machine when generating position independent code. You can assume that *x* satisfies `CONSTANT_P`, so you need not check this. You can also assume *flag_pic* is true, so you need not check it either. You need not define this macro if all constants (including `SYMBOL_REF`) can be immediate operands when generating position independent code.

14.19 Defining the Output Assembler Language

This section describes macros whose principal purpose is to describe how to write instructions in assembler language—rather than what the instructions do.

14.19.1 The Overall Framework of an Assembler File

This describes the overall framework of an assembly file.

`void TARGET_ASM_FILE_START ()` [Target Hook]
Output to `asm_out_file` any text which the assembler expects to find at the beginning of a file. The default behavior is controlled by two flags, documented below. Unless your target's assembler is quite unusual, if you override the default, you should call `default_file_start` at some point in your target hook. This lets other target files rely on these variables.

`bool TARGET_ASM_FILE_START_APP_OFF` [Target Hook]
If this flag is true, the text of the macro `ASM_APP_OFF` will be printed as the very first line in the assembly file, unless '`-fverbose-asm`' is in effect. (If that macro has been defined to the empty string, this variable has no effect.) With the normal definition of `ASM_APP_OFF`, the effect is to notify the GNU assembler that it need not bother stripping comments or extra whitespace from its input. This allows it to work a bit faster.

The default is false. You should not set it to true unless you have verified that your port does not generate any extra whitespace or comments that will cause GAS to issue errors in NO_APP mode.

`bool TARGET_ASM_FILE_START_FILE_DIRECTIVE` [Target Hook]
If this flag is true, `output_file_directive` will be called for the primary source file, immediately after printing `ASM_APP_OFF` (if that is enabled). Most ELF assemblers expect this to be done. The default is false.

`void TARGET_ASM_FILE_END ()` [Target Hook]
Output to `asm_out_file` any text which the assembler expects to find at the end of a file. The default is to output nothing.

`void file_end_indicate_exec_stack ()` [Function]
Some systems use a common convention, the '`.note.GNU-stack`' special section, to indicate whether or not an object file relies on the stack being executable. If your system uses this convention, you should define `TARGET_ASM_FILE_END` to this function. If you need to do other things in that hook, have your hook function call this function.

`ASM_COMMENT_START` [Macro]
A C string constant describing how to begin a comment in the target assembler language. The compiler assumes that the comment will end at the end of the line.

`ASM_APP_ON` [Macro]
A C string constant for text to be output before each `asm` statement or group of consecutive ones. Normally this is `"#APP"`, which is a comment that has no effect on most assemblers but tells the GNU assembler that it must check the lines that follow for all valid assembler constructs.

`ASM_APP_OFF` [Macro]
A C string constant for text to be output after each `asm` statement or group of consecutive ones. Normally this is `"#NO_APP"`, which tells the GNU assembler to resume making the time-saving assumptions that are valid for ordinary compiler output.

`ASM_OUTPUT_SOURCE_FILENAME (`*stream*, *name*`)` [Macro]
A C statement to output COFF information or DWARF debugging information which indicates that filename *name* is the current source file to the stdio stream *stream*.

This macro need not be defined if the standard form of output for the file format in use is appropriate.

`OUTPUT_QUOTED_STRING (`*stream*, *string*`)` [Macro]
A C statement to output the string *string* to the stdio stream *stream*. If you do not call the function `output_quoted_string` in your config files, GCC will only call it

to output filenames to the assembler source. So you can use it to canonicalize the format of the filename using this macro.

`ASM_OUTPUT_IDENT` (*stream*, *string*) [Macro]
A C statement to output something to the assembler file to handle a '`#ident`' directive containing the text *string*. If this macro is not defined, nothing is output for a '`#ident`' directive.

`void TARGET_ASM_NAMED_SECTION` (*const char *name*, *unsigned int flags*, *unsigned int* `align`) [Target Hook]
Output assembly directives to switch to section *name*. The section should have attributes as specified by *flags*, which is a bit mask of the `SECTION_*` flags defined in '`output.h`'. If *align* is nonzero, it contains an alignment in bytes to be used for the section, otherwise some target default should be used. Only targets that must specify an alignment within the section directive need pay attention to *align* – we will still use `ASM_OUTPUT_ALIGN`.

`bool TARGET_HAVE_NAMED_SECTIONS` [Target Hook]
This flag is true if the target supports `TARGET_ASM_NAMED_SECTION`.

`unsigned int TARGET_SECTION_TYPE_FLAGS` (*tree* *decl*, *const char *name*, *int reloc*) [Target Hook]
Choose a set of section attributes for use by `TARGET_ASM_NAMED_SECTION` based on a variable or function decl, a section name, and whether or not the declaration's initializer may contain runtime relocations. *decl* may be null, in which case read-write data should be assumed.

The default version if this function handles choosing code vs data, read-only vs read-write data, and `flag_pic`. You should only need to override this if your target has special flags that might be set via `__attribute__`.

14.19.2 Output of Data

`const char * TARGET_ASM_BYTE_OP` [Target Hook]
`const char * TARGET_ASM_ALIGNED_HI_OP` [Target Hook]
`const char * TARGET_ASM_ALIGNED_SI_OP` [Target Hook]
`const char * TARGET_ASM_ALIGNED_DI_OP` [Target Hook]
`const char * TARGET_ASM_ALIGNED_TI_OP` [Target Hook]
`const char * TARGET_ASM_UNALIGNED_HI_OP` [Target Hook]
`const char * TARGET_ASM_UNALIGNED_SI_OP` [Target Hook]
`const char * TARGET_ASM_UNALIGNED_DI_OP` [Target Hook]
`const char * TARGET_ASM_UNALIGNED_TI_OP` [Target Hook]
These hooks specify assembly directives for creating certain kinds of integer object. The `TARGET_ASM_BYTE_OP` directive creates a byte-sized object, the `TARGET_ASM_ALIGNED_HI_OP` one creates an aligned two-byte object, and so on. Any of the hooks may be `NULL`, indicating that no suitable directive is available.

The compiler will print these strings at the start of a new line, followed immediately by the object's initial value. In most cases, the string should contain a tab, a pseudo-op, and then another tab.

`bool TARGET_ASM_INTEGER (`*rtx* `x`*, unsigned int* `size`*, int* `aligned_p``)` [Target Hook]

The `assemble_integer` function uses this hook to output an integer object. *x* is the object's value, *size* is its size in bytes and *aligned_p* indicates whether it is aligned. The function should return `true` if it was able to output the object. If it returns false, `assemble_integer` will try to split the object into smaller parts.

The default implementation of this hook will use the `TARGET_ASM_BYTE_OP` family of strings, returning `false` when the relevant string is `NULL`.

`OUTPUT_ADDR_CONST_EXTRA (`*stream*, *x*, *fail*`)` [Macro]

A C statement to recognize *rtx* patterns that `output_addr_const` can't deal with, and output assembly code to *stream* corresponding to the pattern *x*. This may be used to allow machine-dependent `UNSPEC`s to appear within constants.

If `OUTPUT_ADDR_CONST_EXTRA` fails to recognize a pattern, it must `goto fail`, so that a standard error message is printed. If it prints an error message itself, by calling, for example, `output_operand_lossage`, it may just complete normally.

`ASM_OUTPUT_ASCII (`*stream*, *ptr*, *len*`)` [Macro]

A C statement to output to the stdio stream *stream* an assembler instruction to assemble a string constant containing the *len* bytes at *ptr*. *ptr* will be a C expression of type `char *` and *len* a C expression of type `int`.

If the assembler has a `.ascii` pseudo-op as found in the Berkeley Unix assembler, do not define the macro `ASM_OUTPUT_ASCII`.

`ASM_OUTPUT_FDESC (`*stream*, *decl*, *n*`)` [Macro]

A C statement to output word *n* of a function descriptor for *decl*. This must be defined if `TARGET_VTABLE_USES_DESCRIPTORS` is defined, and is otherwise unused.

`CONSTANT_POOL_BEFORE_FUNCTION` [Macro]

You may define this macro as a C expression. You should define the expression to have a nonzero value if GCC should output the constant pool for a function before the code for the function, or a zero value if GCC should output the constant pool after the function. If you do not define this macro, the usual case, GCC will output the constant pool before the function.

`ASM_OUTPUT_POOL_PROLOGUE (`*file*, *funname*, *fundecl*, *size*`)` [Macro]

A C statement to output assembler commands to define the start of the constant pool for a function. *funname* is a string giving the name of the function. Should the return type of the function be required, it can be obtained via *fundecl*. *size* is the size, in bytes, of the constant pool that will be written immediately after this call.

If no constant-pool prefix is required, the usual case, this macro need not be defined.

`ASM_OUTPUT_SPECIAL_POOL_ENTRY (`*file*, *x*, *mode*, *align*, *labelno*, *jumpto*`)` [Macro]

A C statement (with or without semicolon) to output a constant in the constant pool, if it needs special treatment. (This macro need not do anything for RTL expressions that can be output normally.)

The argument *file* is the standard I/O stream to output the assembler code on. *x* is the RTL expression for the constant to output, and *mode* is the machine mode (in case *x* is a '`const_int`'). *align* is the required alignment for the value *x*; you should output an assembler directive to force this much alignment.

The argument *labelno* is a number to use in an internal label for the address of this pool entry. The definition of this macro is responsible for outputting the label definition at the proper place. Here is how to do this:

```
(*targetm.asm_out.internal_label) (file, "LC", labelno);
```

When you output a pool entry specially, you should end with a `goto` to the label *jumpto*. This will prevent the same pool entry from being output a second time in the usual manner.

You need not define this macro if it would do nothing.

`ASM_OUTPUT_POOL_EPILOGUE (file funname fundecl size)` [Macro]

A C statement to output assembler commands to at the end of the constant pool for a function. *funname* is a string giving the name of the function. Should the return type of the function be required, you can obtain it via *fundecl*. *size* is the size, in bytes, of the constant pool that GCC wrote immediately before this call.

If no constant-pool epilogue is required, the usual case, you need not define this macro.

`IS_ASM_LOGICAL_LINE_SEPARATOR (C)` [Macro]

Define this macro as a C expression which is nonzero if *C* is used as a logical line separator by the assembler.

If you do not define this macro, the default is that only the character '`;`' is treated as a logical line separator.

`const char * TARGET_ASM_OPEN_PAREN` [Target Hook]
`const char * TARGET_ASM_CLOSE_PAREN` [Target Hook]

These target hooks are C string constants, describing the syntax in the assembler for grouping arithmetic expressions. If not overridden, they default to normal parentheses, which is correct for most assemblers.

These macros are provided by '`real.h`' for writing the definitions of `ASM_OUTPUT_DOUBLE` and the like:

`REAL_VALUE_TO_TARGET_SINGLE (x, l)` [Macro]
`REAL_VALUE_TO_TARGET_DOUBLE (x, l)` [Macro]
`REAL_VALUE_TO_TARGET_LONG_DOUBLE (x, l)` [Macro]

These translate *x*, of type `REAL_VALUE_TYPE`, to the target's floating point representation, and store its bit pattern in the variable *l*. For `REAL_VALUE_TO_TARGET_SINGLE`, this variable should be a simple `long int`. For the others, it should be an array of `long int`. The number of elements in this array is determined by the size of the desired target floating point data type: 32 bits of it go in each `long int` array element. Each array element holds 32 bits of the result, even if `long int` is wider than 32 bits on the host machine.

The array element values are designed so that you can print them out using `fprintf` in the order they should appear in the target machine's memory.

14.19.3 Output of Uninitialized Variables

Each of the macros in this section is used to do the whole job of outputting a single uninitialized variable.

`ASM_OUTPUT_COMMON (`*`stream`*`, `*`name`*`, `*`size`*`, `*`rounded`*`)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* the assembler definition of a common-label named *name* whose size is *size* bytes. The variable *rounded* is the size rounded up to whatever alignment the caller wants.

Use the expression `assemble_name (`*`stream`*`, `*`name`*`)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline.

This macro controls how the assembler definitions of uninitialized common global variables are output.

`ASM_OUTPUT_ALIGNED_COMMON (`*`stream`*`, `*`name`*`, `*`size`*`, `*`alignment`*`)` [Macro]
Like `ASM_OUTPUT_COMMON` except takes the required alignment as a separate, explicit argument. If you define this macro, it is used in place of `ASM_OUTPUT_COMMON`, and gives you more flexibility in handling the required alignment of the variable. The alignment is specified as the number of bits.

`ASM_OUTPUT_ALIGNED_DECL_COMMON (`*`stream`*`, `*`decl`*`, `*`name`*`, `*`size`*`,` [Macro]
`alignment``)`
Like `ASM_OUTPUT_ALIGNED_COMMON` except that *decl* of the variable to be output, if there is one, or `NULL_TREE` if there is no corresponding variable. If you define this macro, GCC will use it in place of both `ASM_OUTPUT_COMMON` and `ASM_OUTPUT_ALIGNED_COMMON`. Define this macro when you need to see the variable's decl in order to chose what to output.

`ASM_OUTPUT_SHARED_COMMON (`*`stream`*`, `*`name`*`, `*`size`*`, `*`rounded`*`)` [Macro]
If defined, it is similar to `ASM_OUTPUT_COMMON`, except that it is used when *name* is shared. If not defined, `ASM_OUTPUT_COMMON` will be used.

`ASM_OUTPUT_BSS (`*`stream`*`, `*`decl`*`, `*`name`*`, `*`size`*`, `*`rounded`*`)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* the assembler definition of uninitialized global *decl* named *name* whose size is *size* bytes. The variable *rounded* is the size rounded up to whatever alignment the caller wants.

Try to use function `asm_output_bss` defined in 'varasm.c' when defining this macro. If unable, use the expression `assemble_name (`*`stream`*`, `*`name`*`)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline.

This macro controls how the assembler definitions of uninitialized global variables are output. This macro exists to properly support languages like C++ which do not have `common` data. However, this macro currently is not defined for all targets. If this macro and `ASM_OUTPUT_ALIGNED_BSS` are not defined then `ASM_OUTPUT_COMMON` or `ASM_OUTPUT_ALIGNED_COMMON` or `ASM_OUTPUT_ALIGNED_DECL_COMMON` is used.

`ASM_OUTPUT_ALIGNED_BSS (stream, decl, name, size, alignment)` [Macro]
Like `ASM_OUTPUT_BSS` except takes the required alignment as a separate, explicit argument. If you define this macro, it is used in place of `ASM_OUTPUT_BSS`, and gives you more flexibility in handling the required alignment of the variable. The alignment is specified as the number of bits.

Try to use function `asm_output_aligned_bss` defined in file 'varasm.c' when defining this macro.

`ASM_OUTPUT_SHARED_BSS (stream, decl, name, size, rounded)` [Macro]
If defined, it is similar to `ASM_OUTPUT_BSS`, except that it is used when *name* is shared. If not defined, `ASM_OUTPUT_BSS` will be used.

`ASM_OUTPUT_LOCAL (stream, name, size, rounded)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* the assembler definition of a local-common-label named *name* whose size is *size* bytes. The variable *rounded* is the size rounded up to whatever alignment the caller wants.

Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline.

This macro controls how the assembler definitions of uninitialized static variables are output.

`ASM_OUTPUT_ALIGNED_LOCAL (stream, name, size, alignment)` [Macro]
Like `ASM_OUTPUT_LOCAL` except takes the required alignment as a separate, explicit argument. If you define this macro, it is used in place of `ASM_OUTPUT_LOCAL`, and gives you more flexibility in handling the required alignment of the variable. The alignment is specified as the number of bits.

`ASM_OUTPUT_ALIGNED_DECL_LOCAL (stream, decl, name, size, alignment)` [Macro]
Like `ASM_OUTPUT_ALIGNED_DECL` except that *decl* of the variable to be output, if there is one, or `NULL_TREE` if there is no corresponding variable. If you define this macro, GCC will use it in place of both `ASM_OUTPUT_DECL` and `ASM_OUTPUT_ALIGNED_DECL`. Define this macro when you need to see the variable's decl in order to chose what to output.

`ASM_OUTPUT_SHARED_LOCAL (stream, name, size, rounded)` [Macro]
If defined, it is similar to `ASM_OUTPUT_LOCAL`, except that it is used when *name* is shared. If not defined, `ASM_OUTPUT_LOCAL` will be used.

14.19.4 Output and Generation of Labels

This is about outputting labels.

`ASM_OUTPUT_LABEL (stream, name)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* the assembler definition of a label named *name*. Use the expression `assemble_name (stream, name)` to output the name itself; before and after that, output the additional assembler syntax for defining the name, and a newline. A default definition of this macro is provided which is correct for most systems.

`ASM_OUTPUT_INTERNAL_LABEL (`*stream*, *name*`)` [Macro]
Identical to `ASM_OUTPUT_LABEL`, except that *name* is known to refer to a compiler-generated label. The default definition uses `assemble_name_raw`, which is like `assemble_name` except that it is more efficient.

`SIZE_ASM_OP` [Macro]
A C string containing the appropriate assembler directive to specify the size of a symbol, without any arguments. On systems that use ELF, the default (in '`config/elfos.h`') is '`"\t.size\t"`'; on other systems, the default is not to define this macro.

Define this macro only if it is correct to use the default definitions of `ASM_OUTPUT_SIZE_DIRECTIVE` and `ASM_OUTPUT_MEASURED_SIZE` for your system. If you need your own custom definitions of those macros, or if you do not need explicit symbol sizes at all, do not define this macro.

`ASM_OUTPUT_SIZE_DIRECTIVE (`*stream*, *name*, *size*`)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* a directive telling the assembler that the size of the symbol *name* is *size*. *size* is a `HOST_WIDE_INT`. If you define `SIZE_ASM_OP`, a default definition of this macro is provided.

`ASM_OUTPUT_MEASURED_SIZE (`*stream*, *name*`)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* a directive telling the assembler to calculate the size of the symbol *name* by subtracting its address from the current address.

If you define `SIZE_ASM_OP`, a default definition of this macro is provided. The default assumes that the assembler recognizes a special '`.`' symbol as referring to the current address, and can calculate the difference between this and another symbol. If your assembler does not recognize '`.`' or cannot do calculations with it, you will need to redefine `ASM_OUTPUT_MEASURED_SIZE` to use some other technique.

`TYPE_ASM_OP` [Macro]
A C string containing the appropriate assembler directive to specify the type of a symbol, without any arguments. On systems that use ELF, the default (in '`config/elfos.h`') is '`"\t.type\t"`'; on other systems, the default is not to define this macro.

Define this macro only if it is correct to use the default definition of `ASM_OUTPUT_TYPE_DIRECTIVE` for your system. If you need your own custom definition of this macro, or if you do not need explicit symbol types at all, do not define this macro.

`TYPE_OPERAND_FMT` [Macro]
A C string which specifies (using `printf` syntax) the format of the second operand to `TYPE_ASM_OP`. On systems that use ELF, the default (in '`config/elfos.h`') is '`"@%s"`'; on other systems, the default is not to define this macro.

Define this macro only if it is correct to use the default definition of `ASM_OUTPUT_TYPE_DIRECTIVE` for your system. If you need your own custom definition of this macro, or if you do not need explicit symbol types at all, do not define this macro.

`ASM_OUTPUT_TYPE_DIRECTIVE (stream, type)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* a directive telling the assembler that the type of the symbol *name* is *type*. *type* is a C string; currently, that string is always either '`"function"`' or '`"object"`', but you should not count on this.

If you define `TYPE_ASM_OP` and `TYPE_OPERAND_FMT`, a default definition of this macro is provided.

`ASM_DECLARE_FUNCTION_NAME (stream, name, decl)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* any text necessary for declaring the name *name* of a function which is being defined. This macro is responsible for outputting the label definition (perhaps using `ASM_OUTPUT_LABEL`). The argument *decl* is the `FUNCTION_DECL` tree node representing the function.

If this macro is not defined, then the function name is defined in the usual manner as a label (by means of `ASM_OUTPUT_LABEL`).

You may wish to use `ASM_OUTPUT_TYPE_DIRECTIVE` in the definition of this macro.

`ASM_DECLARE_FUNCTION_SIZE (stream, name, decl)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* any text necessary for declaring the size of a function which is being defined. The argument *name* is the name of the function. The argument *decl* is the `FUNCTION_DECL` tree node representing the function.

If this macro is not defined, then the function size is not defined.

You may wish to use `ASM_OUTPUT_MEASURED_SIZE` in the definition of this macro.

`ASM_DECLARE_OBJECT_NAME (stream, name, decl)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* any text necessary for declaring the name *name* of an initialized variable which is being defined. This macro must output the label definition (perhaps using `ASM_OUTPUT_LABEL`). The argument *decl* is the `VAR_DECL` tree node representing the variable.

If this macro is not defined, then the variable name is defined in the usual manner as a label (by means of `ASM_OUTPUT_LABEL`).

You may wish to use `ASM_OUTPUT_TYPE_DIRECTIVE` and/or `ASM_OUTPUT_SIZE_DIRECTIVE` in the definition of this macro.

`ASM_DECLARE_CONSTANT_NAME (stream, name, exp, size)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* any text necessary for declaring the name *name* of a constant which is being defined. This macro is responsible for outputting the label definition (perhaps using `ASM_OUTPUT_LABEL`). The argument *exp* is the value of the constant, and *size* is the size of the constant in bytes. *name* will be an internal label.

If this macro is not defined, then the *name* is defined in the usual manner as a label (by means of `ASM_OUTPUT_LABEL`).

You may wish to use `ASM_OUTPUT_TYPE_DIRECTIVE` in the definition of this macro.

ASM_DECLARE_REGISTER_GLOBAL (*stream*, *decl*, *regno*, *name*) [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* any text necessary for claiming a register *regno* for a global variable *decl* with name *name*.

If you don't define this macro, that is equivalent to defining it to do nothing.

ASM_FINISH_DECLARE_OBJECT (*stream*, *decl*, *toplevel*, *atend*) [Macro]
A C statement (sans semicolon) to finish up declaring a variable name once the compiler has processed its initializer fully and thus has had a chance to determine the size of an array when controlled by an initializer. This is used on systems where it's necessary to declare something about the size of the object.

If you don't define this macro, that is equivalent to defining it to do nothing.

You may wish to use ASM_OUTPUT_SIZE_DIRECTIVE and/or ASM_OUTPUT_MEASURED_SIZE in the definition of this macro.

void TARGET_ASM_GLOBALIZE_LABEL (*FILE *stream*, *const char *name*) [Target Hook]
This target hook is a function to output to the stdio stream *stream* some commands that will make the label *name* global; that is, available for reference from other files.

The default implementation relies on a proper definition of GLOBAL_ASM_OP.

ASM_WEAKEN_LABEL (*stream*, *name*) [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* some commands that will make the label *name* weak; that is, available for reference from other files but only used if no other definition is available. Use the expression assemble_name (*stream*, *name*) to output the name itself; before and after that, output the additional assembler syntax for making that name weak, and a newline.

If you don't define this macro or ASM_WEAKEN_DECL, GCC will not support weak symbols and you should not define the SUPPORTS_WEAK macro.

ASM_WEAKEN_DECL (*stream*, *decl*, *name*, *value*) [Macro]
Combines (and replaces) the function of ASM_WEAKEN_LABEL and ASM_OUTPUT_WEAK_ALIAS, allowing access to the associated function or variable decl. If *value* is not NULL, this C statement should output to the stdio stream *stream* assembler code which defines (equates) the weak symbol *name* to have the value *value*. If *value* is NULL, it should output commands to make *name* weak.

ASM_OUTPUT_WEAKREF (*stream*, *decl*, *name*, *value*) [Macro]
Outputs a directive that enables *name* to be used to refer to symbol *value* with weak-symbol semantics. decl is the declaration of name.

SUPPORTS_WEAK [Macro]
A C expression which evaluates to true if the target supports weak symbols.

If you don't define this macro, 'defaults.h' provides a default definition. If either ASM_WEAKEN_LABEL or ASM_WEAKEN_DECL is defined, the default definition is '1'; otherwise, it is '0'. Define this macro if you want to control weak symbol support with a compiler flag such as '-melf'.

`MAKE_DECL_ONE_ONLY (decl)` [Macro]

A C statement (sans semicolon) to mark *decl* to be emitted as a public symbol such that extra copies in multiple translation units will be discarded by the linker. Define this macro if your object file format provides support for this concept, such as the '`COMDAT`' section flags in the Microsoft Windows PE/COFF format, and this support requires changes to *decl*, such as putting it in a separate section.

`SUPPORTS_ONE_ONLY` [Macro]

A C expression which evaluates to true if the target supports one-only semantics.

If you don't define this macro, '`varasm.c`' provides a default definition. If `MAKE_DECL_ONE_ONLY` is defined, the default definition is '`1`'; otherwise, it is '`0`'. Define this macro if you want to control one-only symbol support with a compiler flag, or if setting the `DECL_ONE_ONLY` flag is enough to mark a declaration to be emitted as one-only.

`void TARGET_ASM_ASSEMBLE_VISIBILITY (`*tree* `decl`*, const char* `*visibility`*`)` [Target Hook]

This target hook is a function to output to *asm_out_file* some commands that will make the symbol(s) associated with *decl* have hidden, protected or internal visibility as specified by *visibility*.

`TARGET_WEAK_NOT_IN_ARCHIVE_TOC` [Macro]

A C expression that evaluates to true if the target's linker expects that weak symbols do not appear in a static archive's table of contents. The default is `0`.

Leaving weak symbols out of an archive's table of contents means that, if a symbol will only have a definition in one translation unit and will have undefined references from other translation units, that symbol should not be weak. Defining this macro to be nonzero will thus have the effect that certain symbols that would normally be weak (explicit template instantiations, and vtables for polymorphic classes with noninline key methods) will instead be nonweak.

The C++ ABI requires this macro to be zero. Define this macro for targets where full C++ ABI compliance is impossible and where linker restrictions require weak symbols to be left out of a static archive's table of contents.

`ASM_OUTPUT_EXTERNAL (`*stream*`, `*decl*`, `*name*`)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* any text necessary for declaring the name of an external symbol named *name* which is referenced in this compilation but not defined. The value of *decl* is the tree node for the declaration.

This macro need not be defined if it does not need to output anything. The GNU assembler and most Unix assemblers don't require anything.

`void TARGET_ASM_EXTERNAL_LIBCALL (`*rtx* `symref`*`)` [Target Hook]

This target hook is a function to output to *asm_out_file* an assembler pseudo-op to declare a library function name external. The name of the library function is given by *symref*, which is a `symbol_ref`.

`void TARGET_ASM_MARK_DECL_PRESERVED (`*tree* `decl`*`)` [Target Hook]

This target hook is a function to output to *asm_out_file* an assembler directive to annotate used symbol. Darwin target use .no_dead_code_strip directive.

`ASM_OUTPUT_LABELREF (`*stream*, *name*`)` [Macro]
A C statement (sans semicolon) to output to the stdio stream *stream* a reference in assembler syntax to a label named *name*. This should add '_' to the front of the name, if that is customary on your operating system, as it is in most Berkeley Unix systems. This macro is used in `assemble_name`.

`ASM_OUTPUT_SYMBOL_REF (`*stream*, *sym*`)` [Macro]
A C statement (sans semicolon) to output a reference to `SYMBOL_REF` *sym*. If not defined, `assemble_name` will be used to output the name of the symbol. This macro may be used to modify the way a symbol is referenced depending on information encoded by `TARGET_ENCODE_SECTION_INFO`.

`ASM_OUTPUT_LABEL_REF (`*stream*, *buf*`)` [Macro]
A C statement (sans semicolon) to output a reference to *buf*, the result of `ASM_GENERATE_INTERNAL_LABEL`. If not defined, `assemble_name` will be used to output the name of the symbol. This macro is not used by `output_asm_label`, or the `%l` specifier that calls it; the intention is that this macro should be set when it is necessary to output a label differently when its address is being taken.

`void TARGET_ASM_INTERNAL_LABEL (`*FILE* **stream*, *const char* **prefix*, *unsigned long* *labelno*`)` [Target Hook]
A function to output to the stdio stream *stream* a label whose name is made from the string *prefix* and the number *labelno*.

It is absolutely essential that these labels be distinct from the labels used for user-level functions and variables. Otherwise, certain programs will have name conflicts with internal labels.

It is desirable to exclude internal labels from the symbol table of the object file. Most assemblers have a naming convention for labels that should be excluded; on many systems, the letter 'L' at the beginning of a label has this effect. You should find out what convention your system uses, and follow it.

The default version of this function utilizes `ASM_GENERATE_INTERNAL_LABEL`.

`ASM_OUTPUT_DEBUG_LABEL (`*stream*, *prefix*, *num*`)` [Macro]
A C statement to output to the stdio stream *stream* a debug info label whose name is made from the string *prefix* and the number *num*. This is useful for VLIW targets, where debug info labels may need to be treated differently than branch target labels. On some systems, branch target labels must be at the beginning of instruction bundles, but debug info labels can occur in the middle of instruction bundles.

If this macro is not defined, then `(*targetm.asm_out.internal_label)` will be used.

`ASM_GENERATE_INTERNAL_LABEL (`*string*, *prefix*, *num*`)` [Macro]
A C statement to store into the string *string* a label whose name is made from the string *prefix* and the number *num*.

This string, when output subsequently by `assemble_name`, should produce the output that `(*targetm.asm_out.internal_label)` would produce with the same *prefix* and *num*.

If the string begins with '*', then `assemble_name` will output the rest of the string unchanged. It is often convenient for `ASM_GENERATE_INTERNAL_LABEL` to use '*' in

this way. If the string doesn't start with '*', then ASM_OUTPUT_LABELREF gets to output the string, and may change it. (Of course, ASM_OUTPUT_LABELREF is also part of your machine description, so you should know what it does on your machine.)

ASM_FORMAT_PRIVATE_NAME (*outvar*, *name*, *number*) [Macro]

A C expression to assign to *outvar* (which is a variable of type char *) a newly allocated string made from the string *name* and the number *number*, with some suitable punctuation added. Use alloca to get space for the string.

The string will be used as an argument to ASM_OUTPUT_LABELREF to produce an assembler label for an internal static variable whose name is *name*. Therefore, the string must be such as to result in valid assembler code. The argument *number* is different each time this macro is executed; it prevents conflicts between similarly-named internal static variables in different scopes.

Ideally this string should not be a valid C identifier, to prevent any conflict with the user's own symbols. Most assemblers allow periods or percent signs in assembler symbols; putting at least one of these between the name and the number will suffice.

If this macro is not defined, a default definition will be provided which is correct for most systems.

ASM_OUTPUT_DEF (*stream*, *name*, *value*) [Macro]

A C statement to output to the stdio stream *stream* assembler code which defines (equates) the symbol *name* to have the value *value*.

If SET_ASM_OP is defined, a default definition is provided which is correct for most systems.

ASM_OUTPUT_DEF_FROM_DECLS (*stream*, *decl_of_name*, *decl_of_value*) [Macro]

A C statement to output to the stdio stream *stream* assembler code which defines (equates) the symbol whose tree node is *decl_of_name* to have the value of the tree node *decl_of_value*. This macro will be used in preference to 'ASM_OUTPUT_DEF' if it is defined and if the tree nodes are available.

If SET_ASM_OP is defined, a default definition is provided which is correct for most systems.

TARGET_DEFERRED_OUTPUT_DEFS (*decl_of_name*, *decl_of_value*) [Macro]

A C statement that evaluates to true if the assembler code which defines (equates) the symbol whose tree node is *decl_of_name* to have the value of the tree node *decl_of_value* should be emitted near the end of the current compilation unit. The default is to not defer output of defines. This macro affects defines output by 'ASM_OUTPUT_DEF' and 'ASM_OUTPUT_DEF_FROM_DECLS'.

ASM_OUTPUT_WEAK_ALIAS (*stream*, *name*, *value*) [Macro]

A C statement to output to the stdio stream *stream* assembler code which defines (equates) the weak symbol *name* to have the value *value*. If *value* is NULL, it defines *name* as an undefined weak symbol.

Define this macro if the target only supports weak aliases; define ASM_OUTPUT_DEF instead if possible.

`OBJC_GEN_METHOD_LABEL (`*buf*`, `*is_inst*`, `*class_name*`, `*cat_name*`, `*sel_name*`)` [Macro]

Define this macro to override the default assembler names used for Objective-C methods.

The default name is a unique method number followed by the name of the class (e.g. '`_1_Foo`'). For methods in categories, the name of the category is also included in the assembler name (e.g. '`_1_Foo_Bar`').

These names are safe on most systems, but make debugging difficult since the method's selector is not present in the name. Therefore, particular systems define other ways of computing names.

buf is an expression of type `char *` which gives you a buffer in which to store the name; its length is as long as *class_name*, *cat_name* and *sel_name* put together, plus 50 characters extra.

The argument *is_inst* specifies whether the method is an instance method or a class method; *class_name* is the name of the class; *cat_name* is the name of the category (or `NULL` if the method is not in a category); and *sel_name* is the name of the selector.

On systems where the assembler can handle quoted names, you can use this macro to provide more human-readable names.

`ASM_DECLARE_CLASS_REFERENCE (`*stream*`, `*name*`)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* commands to declare that the label *name* is an Objective-C class reference. This is only needed for targets whose linkers have special support for NeXT-style runtimes.

`ASM_DECLARE_UNRESOLVED_REFERENCE (`*stream*`, `*name*`)` [Macro]

A C statement (sans semicolon) to output to the stdio stream *stream* commands to declare that the label *name* is an unresolved Objective-C class reference. This is only needed for targets whose linkers have special support for NeXT-style runtimes.

14.19.5 How Initialization Functions Are Handled

The compiled code for certain languages includes *constructors* (also called *initialization routines*)—functions to initialize data in the program when the program is started. These functions need to be called before the program is "started"—that is to say, before `main` is called.

Compiling some languages generates *destructors* (also called *termination routines*) that should be called when the program terminates.

To make the initialization and termination functions work, the compiler must output something in the assembler code to cause those functions to be called at the appropriate time. When you port the compiler to a new system, you need to specify how to do this.

There are two major ways that GCC currently supports the execution of initialization and termination functions. Each way has two variants. Much of the structure is common to all four variations.

The linker must build two lists of these functions—a list of initialization functions, called `__CTOR_LIST__`, and a list of termination functions, called `__DTOR_LIST__`.

Each list always begins with an ignored function pointer (which may hold 0, −1, or a count of the function pointers after it, depending on the environment). This is followed

by a series of zero or more function pointers to constructors (or destructors), followed by a function pointer containing zero.

Depending on the operating system and its executable file format, either '`crtstuff.c`' or '`libgcc2.c`' traverses these lists at startup time and exit time. Constructors are called in reverse order of the list; destructors in forward order.

The best way to handle static constructors works only for object file formats which provide arbitrarily-named sections. A section is set aside for a list of constructors, and another for a list of destructors. Traditionally these are called '`.ctors`' and '`.dtors`'. Each object file that defines an initialization function also puts a word in the constructor section to point to that function. The linker accumulates all these words into one contiguous '`.ctors`' section. Termination functions are handled similarly.

This method will be chosen as the default by '`target-def.h`' if `TARGET_ASM_NAMED_SECTION` is defined. A target that does not support arbitrary sections, but does support special designated constructor and destructor sections may define `CTORS_SECTION_ASM_OP` and `DTORS_SECTION_ASM_OP` to achieve the same effect.

When arbitrary sections are available, there are two variants, depending upon how the code in '`crtstuff.c`' is called. On systems that support a *.init* section which is executed at program startup, parts of '`crtstuff.c`' are compiled into that section. The program is linked by the `gcc` driver like this:

```
ld -o output_file crti.o crtbegin.o ... -lgcc crtend.o crtn.o
```

The prologue of a function (`__init`) appears in the `.init` section of '`crti.o`'; the epilogue appears in '`crtn.o`'. Likewise for the function `__fini` in the *.fini* section. Normally these files are provided by the operating system or by the GNU C library, but are provided by GCC for a few targets.

The objects '`crtbegin.o`' and '`crtend.o`' are (for most targets) compiled from '`crtstuff.c`'. They contain, among other things, code fragments within the `.init` and `.fini` sections that branch to routines in the `.text` section. The linker will pull all parts of a section together, which results in a complete `__init` function that invokes the routines we need at startup.

To use this variant, you must define the `INIT_SECTION_ASM_OP` macro properly.

If no init section is available, when GCC compiles any function called `main` (or more accurately, any function designated as a program entry point by the language front end calling `expand_main_function`), it inserts a procedure call to `__main` as the first executable code after the function prologue. The `__main` function is defined in '`libgcc2.c`' and runs the global constructors.

In file formats that don't support arbitrary sections, there are again two variants. In the simplest variant, the GNU linker (GNU `ld`) and an 'a.out' format must be used. In this case, `TARGET_ASM_CONSTRUCTOR` is defined to produce a `.stabs` entry of type '`N_SETT`', referencing the name `__CTOR_LIST__`, and with the address of the void function containing the initialization code as its value. The GNU linker recognizes this as a request to add the value to a *set*; the values are accumulated, and are eventually placed in the executable as a vector in the format described above, with a leading (ignored) count and a trailing zero element. `TARGET_ASM_DESTRUCTOR` is handled similarly. Since no init section is available, the absence of `INIT_SECTION_ASM_OP` causes the compilation of `main` to call `__main` as above, starting the initialization process.

The last variant uses neither arbitrary sections nor the GNU linker. This is preferable when you want to do dynamic linking and when using file formats which the GNU linker does not support, such as 'ECOFF'. In this case, `TARGET_HAVE_CTORS_DTORS` is false, initialization and termination functions are recognized simply by their names. This requires an extra program in the linkage step, called `collect2`. This program pretends to be the linker, for use with GCC; it does its job by running the ordinary linker, but also arranges to include the vectors of initialization and termination functions. These functions are called via `__main` as described above. In order to use this method, `use_collect2` must be defined in the target in '`config.gcc`'.

14.19.6 Macros Controlling Initialization Routines

Here are the macros that control how the compiler handles initialization and termination functions:

`INIT_SECTION_ASM_OP` [Macro]
: If defined, a C string constant, including spacing, for the assembler operation to identify the following data as initialization code. If not defined, GCC will assume such a section does not exist. When you are using special sections for initialization and termination functions, this macro also controls how '`crtstuff.c`' and '`libgcc2.c`' arrange to run the initialization functions.

`HAS_INIT_SECTION` [Macro]
: If defined, `main` will not call `__main` as described above. This macro should be defined for systems that control start-up code on a symbol-by-symbol basis, such as OSF/1, and should not be defined explicitly for systems that support `INIT_SECTION_ASM_OP`.

`LD_INIT_SWITCH` [Macro]
: If defined, a C string constant for a switch that tells the linker that the following symbol is an initialization routine.

`LD_FINI_SWITCH` [Macro]
: If defined, a C string constant for a switch that tells the linker that the following symbol is a finalization routine.

`COLLECT_SHARED_INIT_FUNC (`*stream*`, `*func*`)` [Macro]
: If defined, a C statement that will write a function that can be automatically called when a shared library is loaded. The function should call *func*, which takes no arguments. If not defined, and the object format requires an explicit initialization function, then a function called `_GLOBAL__DI` will be generated.

  This function and the following one are used by collect2 when linking a shared library that needs constructors or destructors, or has DWARF2 exception tables embedded in the code.

`COLLECT_SHARED_FINI_FUNC (`*stream*`, `*func*`)` [Macro]
: If defined, a C statement that will write a function that can be automatically called when a shared library is unloaded. The function should call *func*, which takes no arguments. If not defined, and the object format requires an explicit finalization function, then a function called `_GLOBAL__DD` will be generated.

`INVOKE__main` [Macro]
If defined, `main` will call `__main` despite the presence of `INIT_SECTION_ASM_OP`. This macro should be defined for systems where the init section is not actually run automatically, but is still useful for collecting the lists of constructors and destructors.

`SUPPORTS_INIT_PRIORITY` [Macro]
If nonzero, the C++ `init_priority` attribute is supported and the compiler should emit instructions to control the order of initialization of objects. If zero, the compiler will issue an error message upon encountering an `init_priority` attribute.

`bool TARGET_HAVE_CTORS_DTORS` [Target Hook]
This value is true if the target supports some "native" method of collecting constructors and destructors to be run at startup and exit. It is false if we must use `collect2`.

`void TARGET_ASM_CONSTRUCTOR (`*rtx symbol*, *int priority*`)` [Target Hook]
If defined, a function that outputs assembler code to arrange to call the function referenced by *symbol* at initialization time.

Assume that *symbol* is a `SYMBOL_REF` for a function taking no arguments and with no return value. If the target supports initialization priorities, *priority* is a value between 0 and `MAX_INIT_PRIORITY`; otherwise it must be `DEFAULT_INIT_PRIORITY`.

If this macro is not defined by the target, a suitable default will be chosen if (1) the target supports arbitrary section names, (2) the target defines `CTORS_SECTION_ASM_OP`, or (3) `USE_COLLECT2` is not defined.

`void TARGET_ASM_DESTRUCTOR (`*rtx symbol*, *int priority*`)` [Target Hook]
This is like `TARGET_ASM_CONSTRUCTOR` but used for termination functions rather than initialization functions.

If `TARGET_HAVE_CTORS_DTORS` is true, the initialization routine generated for the generated object file will have static linkage.

If your system uses `collect2` as the means of processing constructors, then that program normally uses `nm` to scan an object file for constructor functions to be called.

On certain kinds of systems, you can define this macro to make `collect2` work faster (and, in some cases, make it work at all):

`OBJECT_FORMAT_COFF` [Macro]
Define this macro if the system uses COFF (Common Object File Format) object files, so that `collect2` can assume this format and scan object files directly for dynamic constructor/destructor functions.

This macro is effective only in a native compiler; `collect2` as part of a cross compiler always uses `nm` for the target machine.

`REAL_NM_FILE_NAME` [Macro]
Define this macro as a C string constant containing the file name to use to execute `nm`. The default is to search the path normally for `nm`.

If your system supports shared libraries and has a program to list the dynamic dependencies of a given library or executable, you can define these macros to enable support for running initialization and termination functions in shared libraries:

`LDD_SUFFIX` [Macro]
Define this macro to a C string constant containing the name of the program which lists dynamic dependencies, like `"ldd"` under SunOS 4.

`PARSE_LDD_OUTPUT (ptr)` [Macro]
Define this macro to be C code that extracts filenames from the output of the program denoted by `LDD_SUFFIX`. *ptr* is a variable of type `char *` that points to the beginning of a line of output from `LDD_SUFFIX`. If the line lists a dynamic dependency, the code must advance *ptr* to the beginning of the filename on that line. Otherwise, it must set *ptr* to `NULL`.

14.19.7 Output of Assembler Instructions

This describes assembler instruction output.

`REGISTER_NAMES` [Macro]
A C initializer containing the assembler's names for the machine registers, each one as a C string constant. This is what translates register numbers in the compiler into assembler language.

`ADDITIONAL_REGISTER_NAMES` [Macro]
If defined, a C initializer for an array of structures containing a name and a register number. This macro defines additional names for hard registers, thus allowing the `asm` option in declarations to refer to registers using alternate names.

`ASM_OUTPUT_OPCODE (stream, ptr)` [Macro]
Define this macro if you are using an unusual assembler that requires different names for the machine instructions.

The definition is a C statement or statements which output an assembler instruction opcode to the stdio stream *stream*. The macro-operand *ptr* is a variable of type `char *` which points to the opcode name in its "internal" form—the form that is written in the machine description. The definition should output the opcode name to *stream*, performing any translation you desire, and increment the variable *ptr* to point at the end of the opcode so that it will not be output twice.

In fact, your macro definition may process less than the entire opcode name, or more than the opcode name; but if you want to process text that includes '`%`'-sequences to substitute operands, you must take care of the substitution yourself. Just be sure to increment *ptr* over whatever text should not be output normally.

If you need to look at the operand values, they can be found as the elements of `recog_data.operand`.

If the macro definition does nothing, the instruction is output in the usual way.

`FINAL_PRESCAN_INSN (insn, opvec, noperands)` [Macro]
If defined, a C statement to be executed just prior to the output of assembler code for *insn*, to modify the extracted operands so they will be output differently.

Here the argument *opvec* is the vector containing the operands extracted from *insn*, and *noperands* is the number of elements of the vector which contain meaningful data for this insn. The contents of this vector are what will be used to convert the insn

template into assembler code, so you can change the assembler output by changing the contents of the vector.

This macro is useful when various assembler syntaxes share a single file of instruction patterns; by defining this macro differently, you can cause a large class of instructions to be output differently (such as with rearranged operands). Naturally, variations in assembler syntax affecting individual insn patterns ought to be handled by writing conditional output routines in those patterns.

If this macro is not defined, it is equivalent to a null statement.

`PRINT_OPERAND (`*`stream`*`, `*`x`*`, `*`code`*`)` [Macro]

A C compound statement to output to stdio stream *stream* the assembler syntax for an instruction operand *x*. *x* is an RTL expression.

code is a value that can be used to specify one of several ways of printing the operand. It is used when identical operands must be printed differently depending on the context. *code* comes from the '`%`' specification that was used to request printing of the operand. If the specification was just '`%`*`digit`*' then *code* is 0; if the specification was '`%`*`ltr digit`*' then *code* is the ASCII code for *ltr*.

If *x* is a register, this macro should print the register's name. The names can be found in an array `reg_names` whose type is `char *[]`. `reg_names` is initialized from `REGISTER_NAMES`.

When the machine description has a specification '`%`*`punct`*' (a '`%`' followed by a punctuation character), this macro is called with a null pointer for *x* and the punctuation character for *code*.

`PRINT_OPERAND_PUNCT_VALID_P (`*`code`*`)` [Macro]

A C expression which evaluates to true if *code* is a valid punctuation character for use in the `PRINT_OPERAND` macro. If `PRINT_OPERAND_PUNCT_VALID_P` is not defined, it means that no punctuation characters (except for the standard one, '`%`') are used in this way.

`PRINT_OPERAND_ADDRESS (`*`stream`*`, `*`x`*`)` [Macro]

A C compound statement to output to stdio stream *stream* the assembler syntax for an instruction operand that is a memory reference whose address is *x*. *x* is an RTL expression.

On some machines, the syntax for a symbolic address depends on the section that the address refers to. On these machines, define the hook `TARGET_ENCODE_SECTION_INFO` to store the information into the `symbol_ref`, and then check for it here. See Section 14.19 [Assembler Format], page 353.

`DBR_OUTPUT_SEQEND (`*`file`*`)` [Macro]

A C statement, to be executed after all slot-filler instructions have been output. If necessary, call `dbr_sequence_length` to determine the number of slots filled in a sequence (zero if not currently outputting a sequence), to decide how many no-ops to output, or whatever.

Don't define this macro if it has nothing to do, but it is helpful in reading assembly output if the extent of the delay sequence is made explicit (e.g. with white space).

Note that output routines for instructions with delay slots must be prepared to deal with not being output as part of a sequence (i.e. when the scheduling pass is not run, or when no slot fillers could be found.) The variable `final_sequence` is null when not processing a sequence, otherwise it contains the `sequence` rtx being output.

`REGISTER_PREFIX` [Macro]
`LOCAL_LABEL_PREFIX` [Macro]
`USER_LABEL_PREFIX` [Macro]
`IMMEDIATE_PREFIX` [Macro]

If defined, C string expressions to be used for the '`%R`', '`%L`', '`%U`', and '`%I`' options of `asm_fprintf` (see '`final.c`'). These are useful when a single '`md`' file must support multiple assembler formats. In that case, the various '`tm.h`' files can define these macros differently.

`ASM_FPRINTF_EXTENSIONS` (*file*, *argptr*, *format*) [Macro]

If defined this macro should expand to a series of `case` statements which will be parsed inside the `switch` statement of the `asm_fprintf` function. This allows targets to define extra printf formats which may useful when generating their assembler statements. Note that uppercase letters are reserved for future generic extensions to asm_fprintf, and so are not available to target specific code. The output file is given by the parameter *file*. The varargs input pointer is *argptr* and the rest of the format string, starting the character after the one that is being switched upon, is pointed to by *format*.

`ASSEMBLER_DIALECT` [Macro]

If your target supports multiple dialects of assembler language (such as different opcodes), define this macro as a C expression that gives the numeric index of the assembler language dialect to use, with zero as the first variant.

If this macro is defined, you may use constructs of the form

```
'{option0|option1|option2...}'
```

in the output templates of patterns (see Section 13.5 [Output Template], page 184) or in the first argument of `asm_fprintf`. This construct outputs '`option0`', '`option1`', '`option2`', etc., if the value of `ASSEMBLER_DIALECT` is zero, one, two, etc. Any special characters within these strings retain their usual meaning. If there are fewer alternatives within the braces than the value of `ASSEMBLER_DIALECT`, the construct outputs nothing.

If you do not define this macro, the characters '`{`', '`|`' and '`}`' do not have any special meaning when used in templates or operands to `asm_fprintf`.

Define the macros `REGISTER_PREFIX`, `LOCAL_LABEL_PREFIX`, `USER_LABEL_PREFIX` and `IMMEDIATE_PREFIX` if you can express the variations in assembler language syntax with that mechanism. Define `ASSEMBLER_DIALECT` and use the '`{option0|option1}`' syntax if the syntax variant are larger and involve such things as different opcodes or operand order.

`ASM_OUTPUT_REG_PUSH` (*stream*, *regno*) [Macro]

A C expression to output to *stream* some assembler code which will push hard register number *regno* onto the stack. The code need not be optimal, since this macro is used only when profiling.

`ASM_OUTPUT_REG_POP (stream, regno)` [Macro]

A C expression to output to *stream* some assembler code which will pop hard register number *regno* off of the stack. The code need not be optimal, since this macro is used only when profiling.

14.19.8 Output of Dispatch Tables

This concerns dispatch tables.

`ASM_OUTPUT_ADDR_DIFF_ELT (stream, body, value, rel)` [Macro]

A C statement to output to the stdio stream *stream* an assembler pseudo-instruction to generate a difference between two labels. *value* and *rel* are the numbers of two internal labels. The definitions of these labels are output using `(*targetm.asm_out.internal_label)`, and they must be printed in the same way here. For example,

```
fprintf (stream, "\t.word L%d-L%d\n",
         value, rel)
```

You must provide this macro on machines where the addresses in a dispatch table are relative to the table's own address. If defined, GCC will also use this macro on all machines when producing PIC. *body* is the body of the `ADDR_DIFF_VEC`; it is provided so that the mode and flags can be read.

`ASM_OUTPUT_ADDR_VEC_ELT (stream, value)` [Macro]

This macro should be provided on machines where the addresses in a dispatch table are absolute.

The definition should be a C statement to output to the stdio stream *stream* an assembler pseudo-instruction to generate a reference to a label. *value* is the number of an internal label whose definition is output using `(*targetm.asm_out.internal_label)`. For example,

```
fprintf (stream, "\t.word L%d\n", value)
```

`ASM_OUTPUT_CASE_LABEL (stream, prefix, num, table)` [Macro]

Define this if the label before a jump-table needs to be output specially. The first three arguments are the same as for `(*targetm.asm_out.internal_label)`; the fourth argument is the jump-table which follows (a `jump_insn` containing an `addr_vec` or `addr_diff_vec`).

This feature is used on system V to output a `swbeg` statement for the table.

If this macro is not defined, these labels are output with `(*targetm.asm_out.internal_label)`.

`ASM_OUTPUT_CASE_END (stream, num, table)` [Macro]

Define this if something special must be output at the end of a jump-table. The definition should be a C statement to be executed after the assembler code for the table is written. It should write the appropriate code to stdio stream *stream*. The argument *table* is the jump-table insn, and *num* is the label-number of the preceding label.

If this macro is not defined, nothing special is output at the end of the jump-table.

`void TARGET_ASM_EMIT_UNWIND_LABEL (stream, decl, for_eh, empty)` [Target Hook]

This target hook emits a label at the beginning of each FDE. It should be defined on targets where FDEs need special labels, and it should write the appropriate label, for the FDE associated with the function declaration *decl*, to the stdio stream *stream*. The third argument, *for_eh*, is a boolean: true if this is for an exception table. The fourth argument, *empty*, is a boolean: true if this is a placeholder label for an omitted FDE.

The default is that FDEs are not given nonlocal labels.

`void TARGET_UNWIND_EMIT (FILE * stream, rtx insn)` [Taget Hook]

This target hook emits and assembly directives required to unwind the given instruction. This is only used when TARGET_UNWIND_INFO is set.

14.19.9 Assembler Commands for Exception Regions

This describes commands marking the start and the end of an exception region.

`EH_FRAME_SECTION_NAME` [Macro]

If defined, a C string constant for the name of the section containing exception handling frame unwind information. If not defined, GCC will provide a default definition if the target supports named sections. '`crtstuff.c`' uses this macro to switch to the appropriate section.

You should define this symbol if your target supports DWARF 2 frame unwind information and the default definition does not work.

`EH_FRAME_IN_DATA_SECTION` [Macro]

If defined, DWARF 2 frame unwind information will be placed in the data section even though the target supports named sections. This might be necessary, for instance, if the system linker does garbage collection and sections cannot be marked as not to be collected.

Do not define this macro unless `TARGET_ASM_NAMED_SECTION` is also defined.

`EH_TABLES_CAN_BE_READ_ONLY` [Macro]

Define this macro to 1 if your target is such that no frame unwind information encoding used with non-PIC code will ever require a runtime relocation, but the linker may not support merging read-only and read-write sections into a single read-write section.

`MASK_RETURN_ADDR` [Macro]

An rtx used to mask the return address found via `RETURN_ADDR_RTX`, so that it does not contain any extraneous set bits in it.

`DWARF2_UNWIND_INFO` [Macro]

Define this macro to 0 if your target supports DWARF 2 frame unwind information, but it does not yet work with exception handling. Otherwise, if your target supports this information (if it defines '`INCOMING_RETURN_ADDR_RTX`' and either '`UNALIGNED_INT_ASM_OP`' or '`OBJECT_FORMAT_ELF`'), GCC will provide a default definition of 1.

If `TARGET_UNWIND_INFO` is defined, the target specific unwinder will be used in all cases. Defining this macro will enable the generation of DWARF 2 frame debugging information.

If `TARGET_UNWIND_INFO` is not defined, and this macro is defined to 1, the DWARF 2 unwinder will be the default exception handling mechanism; otherwise, `setjmp/longjmp` will be used by default.

`TARGET_UNWIND_INFO` [Macro]
Define this macro if your target has ABI specified unwind tables. Usually these will be output by `TARGET_UNWIND_EMIT`.

`Target Hook bool` *TARGET_UNWID_TABLES_DEFAULT* [Variable]
This variable should be set to `true` if the target ABI requires unwinding tables even when exceptions are not used.

`MUST_USE_SJLJ_EXCEPTIONS` [Macro]
This macro need only be defined if `DWARF2_UNWIND_INFO` is runtime-variable. In that case, '`except.h`' cannot correctly determine the corresponding definition of `MUST_USE_SJLJ_EXCEPTIONS`, so the target must provide it directly.

`DWARF_CIE_DATA_ALIGNMENT` [Macro]
This macro need only be defined if the target might save registers in the function prologue at an offset to the stack pointer that is not aligned to `UNITS_PER_WORD`. The definition should be the negative minimum alignment if `STACK_GROWS_DOWNWARD` is defined, and the positive minimum alignment otherwise. See Section 14.20.5 [SDB and DWARF], page 382. Only applicable if the target supports DWARF 2 frame unwind information.

`void TARGET_ASM_EXCEPTION_SECTION ()` [Target Hook]
If defined, a function that switches to the section in which the main exception table is to be placed (see Section 14.17 [Sections], page 349). The default is a function that switches to a section named `.gcc_except_table` on machines that support named sections via `TARGET_ASM_NAMED_SECTION`, otherwise if '`-fpic`' or '`-fPIC`' is in effect, the `data_section`, otherwise the `readonly_data_section`.

`void TARGET_ASM_EH_FRAME_SECTION ()` [Target Hook]
If defined, a function that switches to the section in which the DWARF 2 frame unwind information to be placed (see Section 14.17 [Sections], page 349). The default is a function that outputs a standard GAS section directive, if `EH_FRAME_SECTION_NAME` is defined, or else a data section directive followed by a synthetic label.

`Target Hook bool` *TARGET_TERMINATE_DW2_EH_FRAME_INFO* [Variable]
Contains the value true if the target should add a zero word onto the end of a Dwarf-2 frame info section when used for exception handling. Default value is false if `EH_FRAME_SECTION_NAME` is defined, and true otherwise.

`rtx TARGET_DWARF_REGISTER_SPAN (`*rtx* *reg*`)` [Target Hook]
Given a register, this hook should return a parallel of registers to represent where to find the register pieces. Define this hook if the register and its mode are represented

in Dwarf in non-contiguous locations, or if the register should be represented in more than one register in Dwarf. Otherwise, this hook should return `NULL_RTX`. If not defined, the default is to return `NULL_RTX`.

`bool TARGET_ASM_TTYPE (rtx sym)` [Target Hook]
This hook is used to output a reference from a frame unwinding table to the type_info object identified by *sym*. It should return `true` if the reference was output. Returning `false` will cause the reference to be output using the normal Dwarf2 routines.

`bool TARGET_ARM_EABI_UNWINDER` [Target Hook]
This hook should be set to `true` on targets that use an ARM EABI based unwinding library, and `false` on other targets. This effects the format of unwinding tables, and how the unwinder in entered after running a cleanup. The default is `false`.

14.19.10 Assembler Commands for Alignment

This describes commands for alignment.

`JUMP_ALIGN (label)` [Macro]
The alignment (log base 2) to put in front of *label*, which is a common destination of jumps and has no fallthru incoming edge.

This macro need not be defined if you don't want any special alignment to be done at such a time. Most machine descriptions do not currently define the macro.

Unless it's necessary to inspect the *label* parameter, it is better to set the variable *align_jumps* in the target's `OVERRIDE_OPTIONS`. Otherwise, you should try to honor the user's selection in *align_jumps* in a `JUMP_ALIGN` implementation.

`LABEL_ALIGN_AFTER_BARRIER (label)` [Macro]
The alignment (log base 2) to put in front of *label*, which follows a `BARRIER`.

This macro need not be defined if you don't want any special alignment to be done at such a time. Most machine descriptions do not currently define the macro.

`LABEL_ALIGN_AFTER_BARRIER_MAX_SKIP` [Macro]
The maximum number of bytes to skip when applying `LABEL_ALIGN_AFTER_BARRIER`. This works only if `ASM_OUTPUT_MAX_SKIP_ALIGN` is defined.

`LOOP_ALIGN (label)` [Macro]
The alignment (log base 2) to put in front of *label*, which follows a `NOTE_INSN_LOOP_BEG` note.

This macro need not be defined if you don't want any special alignment to be done at such a time. Most machine descriptions do not currently define the macro.

Unless it's necessary to inspect the *label* parameter, it is better to set the variable `align_loops` in the target's `OVERRIDE_OPTIONS`. Otherwise, you should try to honor the user's selection in `align_loops` in a `LOOP_ALIGN` implementation.

`LOOP_ALIGN_MAX_SKIP` [Macro]
The maximum number of bytes to skip when applying `LOOP_ALIGN`. This works only if `ASM_OUTPUT_MAX_SKIP_ALIGN` is defined.

`LABEL_ALIGN (label)` [Macro]
The alignment (log base 2) to put in front of *label*. If `LABEL_ALIGN_AFTER_BARRIER` / `LOOP_ALIGN` specify a different alignment, the maximum of the specified values is used.

Unless it's necessary to inspect the *label* parameter, it is better to set the variable `align_labels` in the target's `OVERRIDE_OPTIONS`. Otherwise, you should try to honor the user's selection in `align_labels` in a `LABEL_ALIGN` implementation.

`LABEL_ALIGN_MAX_SKIP` [Macro]
The maximum number of bytes to skip when applying `LABEL_ALIGN`. This works only if `ASM_OUTPUT_MAX_SKIP_ALIGN` is defined.

`ASM_OUTPUT_SKIP (stream, nbytes)` [Macro]
A C statement to output to the stdio stream *stream* an assembler instruction to advance the location counter by *nbytes* bytes. Those bytes should be zero when loaded. *nbytes* will be a C expression of type `int`.

`ASM_NO_SKIP_IN_TEXT` [Macro]
Define this macro if `ASM_OUTPUT_SKIP` should not be used in the text section because it fails to put zeros in the bytes that are skipped. This is true on many Unix systems, where the pseudo–op to skip bytes produces no-op instructions rather than zeros when used in the text section.

`ASM_OUTPUT_ALIGN (stream, power)` [Macro]
A C statement to output to the stdio stream *stream* an assembler command to advance the location counter to a multiple of 2 to the *power* bytes. *power* will be a C expression of type `int`.

`ASM_OUTPUT_ALIGN_WITH_NOP (stream, power)` [Macro]
Like `ASM_OUTPUT_ALIGN`, except that the "nop" instruction is used for padding, if necessary.

`ASM_OUTPUT_MAX_SKIP_ALIGN (stream, power, max_skip)` [Macro]
A C statement to output to the stdio stream *stream* an assembler command to advance the location counter to a multiple of 2 to the *power* bytes, but only if *max_skip* or fewer bytes are needed to satisfy the alignment request. *power* and *max_skip* will be a C expression of type `int`.

14.20 Controlling Debugging Information Format

This describes how to specify debugging information.

14.20.1 Macros Affecting All Debugging Formats

These macros affect all debugging formats.

`DBX_REGISTER_NUMBER (regno)` [Macro]

A C expression that returns the DBX register number for the compiler register number *regno*. In the default macro provided, the value of this expression will be *regno* itself. But sometimes there are some registers that the compiler knows about and DBX does not, or vice versa. In such cases, some register may need to have one number in the compiler and another for DBX.

If two registers have consecutive numbers inside GCC, and they can be used as a pair to hold a multiword value, then they *must* have consecutive numbers after renumbering with `DBX_REGISTER_NUMBER`. Otherwise, debuggers will be unable to access such a pair, because they expect register pairs to be consecutive in their own numbering scheme.

If you find yourself defining `DBX_REGISTER_NUMBER` in way that does not preserve register pairs, then what you must do instead is redefine the actual register numbering scheme.

`DEBUGGER_AUTO_OFFSET (x)` [Macro]

A C expression that returns the integer offset value for an automatic variable having address *x* (an RTL expression). The default computation assumes that *x* is based on the frame-pointer and gives the offset from the frame-pointer. This is required for targets that produce debugging output for DBX or COFF-style debugging output for SDB and allow the frame-pointer to be eliminated when the '`-g`' options is used.

`DEBUGGER_ARG_OFFSET (offset, x)` [Macro]

A C expression that returns the integer offset value for an argument having address *x* (an RTL expression). The nominal offset is *offset*.

`PREFERRED_DEBUGGING_TYPE` [Macro]

A C expression that returns the type of debugging output GCC should produce when the user specifies just '`-g`'. Define this if you have arranged for GCC to support more than one format of debugging output. Currently, the allowable values are `DBX_DEBUG`, `SDB_DEBUG`, `DWARF_DEBUG`, `DWARF2_DEBUG`, `XCOFF_DEBUG`, `VMS_DEBUG`, and `VMS_AND_DWARF2_DEBUG`.

When the user specifies '`-ggdb`', GCC normally also uses the value of this macro to select the debugging output format, but with two exceptions. If `DWARF2_DEBUGGING_INFO` is defined, GCC uses the value `DWARF2_DEBUG`. Otherwise, if `DBX_DEBUGGING_INFO` is defined, GCC uses `DBX_DEBUG`.

The value of this macro only affects the default debugging output; the user can always get a specific type of output by using '`-gstabs`', '`-gcoff`', '`-gdwarf-2`', '`-gxcoff`', or '`-gvms`'.

14.20.2 Specific Options for DBX Output

These are specific options for DBX output.

`DBX_DEBUGGING_INFO` [Macro]
Define this macro if GCC should produce debugging output for DBX in response to the '`-g`' option.

`XCOFF_DEBUGGING_INFO` [Macro]
Define this macro if GCC should produce XCOFF format debugging output in response to the '`-g`' option. This is a variant of DBX format.

`DEFAULT_GDB_EXTENSIONS` [Macro]
Define this macro to control whether GCC should by default generate GDB's extended version of DBX debugging information (assuming DBX-format debugging information is enabled at all). If you don't define the macro, the default is 1: always generate the extended information if there is any occasion to.

`DEBUG_SYMS_TEXT` [Macro]
Define this macro if all `.stabs` commands should be output while in the text section.

`ASM_STABS_OP` [Macro]
A C string constant, including spacing, naming the assembler pseudo op to use instead of `"\t.stabs\t"` to define an ordinary debugging symbol. If you don't define this macro, `"\t.stabs\t"` is used. This macro applies only to DBX debugging information format.

`ASM_STABD_OP` [Macro]
A C string constant, including spacing, naming the assembler pseudo op to use instead of `"\t.stabd\t"` to define a debugging symbol whose value is the current location. If you don't define this macro, `"\t.stabd\t"` is used. This macro applies only to DBX debugging information format.

`ASM_STABN_OP` [Macro]
A C string constant, including spacing, naming the assembler pseudo op to use instead of `"\t.stabn\t"` to define a debugging symbol with no name. If you don't define this macro, `"\t.stabn\t"` is used. This macro applies only to DBX debugging information format.

`DBX_NO_XREFS` [Macro]
Define this macro if DBX on your system does not support the construct '`xs`*`tagname`*'. On some systems, this construct is used to describe a forward reference to a structure named *tagname*. On other systems, this construct is not supported at all.

`DBX_CONTIN_LENGTH` [Macro]
A symbol name in DBX-format debugging information is normally continued (split into two separate `.stabs` directives) when it exceeds a certain length (by default, 80 characters). On some operating systems, DBX requires this splitting; on others, splitting must not be done. You can inhibit splitting by defining this macro with the value zero. You can override the default splitting-length by defining this macro as an expression for the length you desire.

`DBX_CONTIN_CHAR` [Macro]
Normally continuation is indicated by adding a '\' character to the end of a `.stabs` string when a continuation follows. To use a different character instead, define this macro as a character constant for the character you want to use. Do not define this macro if backslash is correct for your system.

`DBX_STATIC_STAB_DATA_SECTION` [Macro]
Define this macro if it is necessary to go to the data section before outputting the '`.stabs`' pseudo-op for a non-global static variable.

`DBX_TYPE_DECL_STABS_CODE` [Macro]
The value to use in the "code" field of the `.stabs` directive for a typedef. The default is `N_LSYM`.

`DBX_STATIC_CONST_VAR_CODE` [Macro]
The value to use in the "code" field of the `.stabs` directive for a static variable located in the text section. DBX format does not provide any "right" way to do this. The default is `N_FUN`.

`DBX_REGPARM_STABS_CODE` [Macro]
The value to use in the "code" field of the `.stabs` directive for a parameter passed in registers. DBX format does not provide any "right" way to do this. The default is `N_RSYM`.

`DBX_REGPARM_STABS_LETTER` [Macro]
The letter to use in DBX symbol data to identify a symbol as a parameter passed in registers. DBX format does not customarily provide any way to do this. The default is `'P'`.

`DBX_FUNCTION_FIRST` [Macro]
Define this macro if the DBX information for a function and its arguments should precede the assembler code for the function. Normally, in DBX format, the debugging information entirely follows the assembler code.

`DBX_BLOCKS_FUNCTION_RELATIVE` [Macro]
Define this macro, with value 1, if the value of a symbol describing the scope of a block (`N_LBRAC` or `N_RBRAC`) should be relative to the start of the enclosing function. Normally, GCC uses an absolute address.

`DBX_LINES_FUNCTION_RELATIVE` [Macro]
Define this macro, with value 1, if the value of a symbol indicating the current line number (`N_SLINE`) should be relative to the start of the enclosing function. Normally, GCC uses an absolute address.

`DBX_USE_BINCL` [Macro]
Define this macro if GCC should generate `N_BINCL` and `N_EINCL` stabs for included header files, as on Sun systems. This macro also directs GCC to output a type number as a pair of a file number and a type number within the file. Normally, GCC does not generate `N_BINCL` or `N_EINCL` stabs, and it outputs a single number for a type number.

14.20.3 Open-Ended Hooks for DBX Format

These are hooks for DBX format.

`DBX_OUTPUT_LBRAC (stream, name)` [Macro]
Define this macro to say how to output to *stream* the debugging information for the start of a scope level for variable names. The argument *name* is the name of an assembler symbol (for use with `assemble_name`) whose value is the address where the scope begins.

`DBX_OUTPUT_RBRAC (stream, name)` [Macro]
Like `DBX_OUTPUT_LBRAC`, but for the end of a scope level.

`DBX_OUTPUT_NFUN (stream, lscope_label, decl)` [Macro]
Define this macro if the target machine requires special handling to output an `N_FUN` entry for the function *decl*.

`DBX_OUTPUT_SOURCE_LINE (stream, line, counter)` [Macro]
A C statement to output DBX debugging information before code for line number *line* of the current source file to the stdio stream *stream*. *counter* is the number of time the macro was invoked, including the current invocation; it is intended to generate unique labels in the assembly output.

This macro should not be defined if the default output is correct, or if it can be made correct by defining `DBX_LINES_FUNCTION_RELATIVE`.

`NO_DBX_FUNCTION_END` [Macro]
Some stabs encapsulation formats (in particular ECOFF), cannot handle the `.stabs "",N_FUN,,0,0,Lscope-function-1` gdb dbx extension construct. On those machines, define this macro to turn this feature off without disturbing the rest of the gdb extensions.

`NO_DBX_BNSYM_ENSYM` [Macro]
Some assemblers cannot handle the `.stabd BNSYM/ENSYM,0,0` gdb dbx extension construct. On those machines, define this macro to turn this feature off without disturbing the rest of the gdb extensions.

14.20.4 File Names in DBX Format

This describes file names in DBX format.

`DBX_OUTPUT_MAIN_SOURCE_FILENAME (stream, name)` [Macro]
A C statement to output DBX debugging information to the stdio stream *stream*, which indicates that file *name* is the main source file—the file specified as the input file for compilation. This macro is called only once, at the beginning of compilation.

This macro need not be defined if the standard form of output for DBX debugging information is appropriate.

It may be necessary to refer to a label equal to the beginning of the text section. You can use '`assemble_name (stream, ltext_label_name)`' to do so. If you do this, you must also set the variable *used_ltext_label_name* to `true`.

`NO_DBX_MAIN_SOURCE_DIRECTORY` [Macro]
Define this macro, with value 1, if GCC should not emit an indication of the current directory for compilation and current source language at the beginning of the file.

`NO_DBX_GCC_MARKER` [Macro]
Define this macro, with value 1, if GCC should not emit an indication that this object file was compiled by GCC. The default is to emit an `N_OPT` stab at the beginning of every source file, with '`gcc2_compiled.`' for the string and value 0.

`DBX_OUTPUT_MAIN_SOURCE_FILE_END (`*stream*, *name*`)` [Macro]
A C statement to output DBX debugging information at the end of compilation of the main source file *name*. Output should be written to the stdio stream *stream*.

If you don't define this macro, nothing special is output at the end of compilation, which is correct for most machines.

`DBX_OUTPUT_NULL_N_SO_AT_MAIN_SOURCE_FILE_END` [Macro]
Define this macro *instead of* defining `DBX_OUTPUT_MAIN_SOURCE_FILE_END`, if what needs to be output at the end of compilation is a `N_SO` stab with an empty string, whose value is the highest absolute text address in the file.

14.20.5 Macros for SDB and DWARF Output

Here are macros for SDB and DWARF output.

`SDB_DEBUGGING_INFO` [Macro]
Define this macro if GCC should produce COFF-style debugging output for SDB in response to the '`-g`' option.

`DWARF2_DEBUGGING_INFO` [Macro]
Define this macro if GCC should produce dwarf version 2 format debugging output in response to the '`-g`' option.

`int TARGET_DWARF_CALLING_CONVENTION (`*tree function*`)` [Target Hook]
Define this to enable the dwarf attribute `DW_AT_calling_convention` to be emitted for each function. Instead of an integer return the enum value for the `DW_CC_` tag.

To support optional call frame debugging information, you must also define `INCOMING_RETURN_ADDR_RTX` and either set `RTX_FRAME_RELATED_P` on the prologue insns if you use RTL for the prologue, or call `dwarf2out_def_cfa` and `dwarf2out_reg_save` as appropriate from `TARGET_ASM_FUNCTION_PROLOGUE` if you don't.

`DWARF2_FRAME_INFO` [Macro]
Define this macro to a nonzero value if GCC should always output Dwarf 2 frame information. If `DWARF2_UNWIND_INFO` (see Section 14.19.9 [Exception Region Output], page 374 is nonzero, GCC will output this information not matter how you define `DWARF2_FRAME_INFO`.

`DWARF2_ASM_LINE_DEBUG_INFO` [Macro]
Define this macro to be a nonzero value if the assembler can generate Dwarf 2 line debug info sections. This will result in much more compact line number tables, and hence is desirable if it works.

`ASM_OUTPUT_DWARF_DELTA (`*`stream`*`, `*`size`*`, `*`label1`*`, `*`label2`*`)` [Macro]
A C statement to issue assembly directives that create a difference between the two given labels, using an integer of the given size.

`ASM_OUTPUT_DWARF_OFFSET (`*`stream`*`, `*`size`*`, `*`label`*`)` [Macro]
A C statement to issue assembly directives that create a section-relative reference to the given label, using an integer of the given size.

`ASM_OUTPUT_DWARF_PCREL (`*`stream`*`, `*`size`*`, `*`label`*`)` [Macro]
A C statement to issue assembly directives that create a self-relative reference to the given label, using an integer of the given size.

`void TARGET_ASM_OUTPUT_DWARF_DTPREL (`*FILE *FILE*, *int* *`size`*, *rtx* *`x`*`)` [Target Hook]
If defined, this target hook is a function which outputs a DTP-relative reference to the given TLS symbol of the specified size.

`PUT_SDB_...` [Macro]
Define these macros to override the assembler syntax for the special SDB assembler directives. See '`sdbout.c`' for a list of these macros and their arguments. If the standard syntax is used, you need not define them yourself.

`SDB_DELIM` [Macro]
Some assemblers do not support a semicolon as a delimiter, even between SDB assembler directives. In that case, define this macro to be the delimiter to use (usually '`\n`'). It is not necessary to define a new set of `PUT_SDB_`*`op`* macros if this is the only change required.

`SDB_ALLOW_UNKNOWN_REFERENCES` [Macro]
Define this macro to allow references to unknown structure, union, or enumeration tags to be emitted. Standard COFF does not allow handling of unknown references, MIPS ECOFF has support for it.

`SDB_ALLOW_FORWARD_REFERENCES` [Macro]
Define this macro to allow references to structure, union, or enumeration tags that have not yet been seen to be handled. Some assemblers choke if forward tags are used, while some require it.

`SDB_OUTPUT_SOURCE_LINE (`*`stream`*`, `*`line`*`)` [Macro]
A C statement to output SDB debugging information before code for line number *line* of the current source file to the stdio stream *stream*. The default is to emit an `.ln` directive.

14.20.6 Macros for VMS Debug Format

Here are macros for VMS debug format.

`VMS_DEBUGGING_INFO` [Macro]
Define this macro if GCC should produce debugging output for VMS in response to the '`-g`' option. The default behavior for VMS is to generate minimal debug info for a traceback in the absence of '`-g`' unless explicitly overridden with '`-g0`'. This behavior is controlled by `OPTIMIZATION_OPTIONS` and `OVERRIDE_OPTIONS`.

14.21 Cross Compilation and Floating Point

While all modern machines use twos-complement representation for integers, there are a variety of representations for floating point numbers. This means that in a cross-compiler the representation of floating point numbers in the compiled program may be different from that used in the machine doing the compilation.

Because different representation systems may offer different amounts of range and precision, all floating point constants must be represented in the target machine's format. Therefore, the cross compiler cannot safely use the host machine's floating point arithmetic; it must emulate the target's arithmetic. To ensure consistency, GCC always uses emulation to work with floating point values, even when the host and target floating point formats are identical.

The following macros are provided by '`real.h`' for the compiler to use. All parts of the compiler which generate or optimize floating-point calculations must use these macros. They may evaluate their operands more than once, so operands must not have side effects.

`REAL_VALUE_TYPE` [Macro]
The C data type to be used to hold a floating point value in the target machine's format. Typically this is a `struct` containing an array of `HOST_WIDE_INT`, but all code should treat it as an opaque quantity.

`int REAL_VALUES_EQUAL (`*REAL_VALUE_TYPE* `x`, *REAL_VALUE_TYPE* `y``)` [Macro]
Compares for equality the two values, *x* and *y*. If the target floating point format supports negative zeroes and/or NaNs, '`REAL_VALUES_EQUAL (-0.0, 0.0)`' is true, and '`REAL_VALUES_EQUAL (NaN, NaN)`' is false.

`int REAL_VALUES_LESS (`*REAL_VALUE_TYPE* `x`, *REAL_VALUE_TYPE* `y``)` [Macro]
Tests whether *x* is less than *y*.

`HOST_WIDE_INT REAL_VALUE_FIX (`*REAL_VALUE_TYPE* `x``)` [Macro]
Truncates *x* to a signed integer, rounding toward zero.

`unsigned HOST_WIDE_INT REAL_VALUE_UNSIGNED_FIX (`*REAL_VALUE_TYPE* `x``)` [Macro]
Truncates *x* to an unsigned integer, rounding toward zero. If *x* is negative, returns zero.

`REAL_VALUE_TYPE REAL_VALUE_ATOF (`*const char *`string`, enum machine_mode* `mode``)` [Macro]
Converts *string* into a floating point number in the target machine's representation for mode *mode*. This routine can handle both decimal and hexadecimal floating point constants, using the syntax defined by the C language for both.

`int REAL_VALUE_NEGATIVE (`*REAL_VALUE_TYPE* `x``)` [Macro]
Returns 1 if *x* is negative (including negative zero), 0 otherwise.

`int REAL_VALUE_ISINF (`*REAL_VALUE_TYPE* `x``)` [Macro]
Determines whether *x* represents infinity (positive or negative).

`int REAL_VALUE_ISNAN (`*REAL_VALUE_TYPE* `x``)` [Macro]
Determines whether *x* represents a "NaN" (not-a-number).

`void REAL_ARITHMETIC (`*REAL_VALUE_TYPE* `output`, *enum tree_code* `code`, *REAL_VALUE_TYPE* `x`, *REAL_VALUE_TYPE* `y``)` [Macro]
Calculates an arithmetic operation on the two floating point values *x* and *y*, storing the result in *output* (which must be a variable).

The operation to be performed is specified by *code*. Only the following codes are supported: `PLUS_EXPR`, `MINUS_EXPR`, `MULT_EXPR`, `RDIV_EXPR`, `MAX_EXPR`, `MIN_EXPR`.

If `REAL_ARITHMETIC` is asked to evaluate division by zero and the target's floating point format cannot represent infinity, it will call `abort`. Callers should check for this situation first, using `MODE_HAS_INFINITIES`. See Section 14.5 [Storage Layout], page 278.

`REAL_VALUE_TYPE REAL_VALUE_NEGATE (`*REAL_VALUE_TYPE* `x``)` [Macro]
Returns the negative of the floating point value *x*.

`REAL_VALUE_TYPE REAL_VALUE_ABS (`*REAL_VALUE_TYPE* `x``)` [Macro]
Returns the absolute value of *x*.

`REAL_VALUE_TYPE REAL_VALUE_TRUNCATE (`*REAL_VALUE_TYPE* `mode`, *enum machine_mode* `x``)` [Macro]
Truncates the floating point value *x* to fit in *mode*. The return value is still a full-size `REAL_VALUE_TYPE`, but it has an appropriate bit pattern to be output asa floating constant whose precision accords with mode *mode*.

`void REAL_VALUE_TO_INT (`*HOST_WIDE_INT* `low`, *HOST_WIDE_INT* `high`, *REAL_VALUE_TYPE* `x``)` [Macro]
Converts a floating point value *x* into a double-precision integer which is then stored into *low* and *high*. If the value is not integral, it is truncated.

`void REAL_VALUE_FROM_INT (`*REAL_VALUE_TYPE* `x`, *HOST_WIDE_INT* `low`, *HOST_WIDE_INT* `high`, *enum machine_mode* `mode``)` [Macro]
Converts a double-precision integer found in *low* and *high*, into a floating point value which is then stored into *x*. The value is truncated to fit in mode *mode*.

14.22 Mode Switching Instructions

The following macros control mode switching optimizations:

`OPTIMIZE_MODE_SWITCHING (`*entity*`)` [Macro]

Define this macro if the port needs extra instructions inserted for mode switching in an optimizing compilation.

For an example, the SH4 can perform both single and double precision floating point operations, but to perform a single precision operation, the FPSCR PR bit has to be cleared, while for a double precision operation, this bit has to be set. Changing the PR bit requires a general purpose register as a scratch register, hence these FPSCR sets have to be inserted before reload, i.e. you can't put this into instruction emitting or `TARGET_MACHINE_DEPENDENT_REORG`.

You can have multiple entities that are mode-switched, and select at run time which entities actually need it. `OPTIMIZE_MODE_SWITCHING` should return nonzero for any *entity* that needs mode-switching. If you define this macro, you also have to define `NUM_MODES_FOR_MODE_SWITCHING`, `MODE_NEEDED`, `MODE_PRIORITY_TO_MODE` and `EMIT_MODE_SET`. `MODE_AFTER`, `MODE_ENTRY`, and `MODE_EXIT` are optional.

`NUM_MODES_FOR_MODE_SWITCHING` [Macro]

If you define `OPTIMIZE_MODE_SWITCHING`, you have to define this as initializer for an array of integers. Each initializer element N refers to an entity that needs mode switching, and specifies the number of different modes that might need to be set for this entity. The position of the initializer in the initializer—starting counting at zero—determines the integer that is used to refer to the mode-switched entity in question. In macros that take mode arguments / yield a mode result, modes are represented as numbers $0 \ldots N - 1$. N is used to specify that no mode switch is needed / supplied.

`MODE_NEEDED (`*entity*`, `*insn*`)` [Macro]

entity is an integer specifying a mode-switched entity. If `OPTIMIZE_MODE_SWITCHING` is defined, you must define this macro to return an integer value not larger than the corresponding element in `NUM_MODES_FOR_MODE_SWITCHING`, to denote the mode that *entity* must be switched into prior to the execution of *insn*.

`MODE_AFTER (`*mode*`, `*insn*`)` [Macro]

If this macro is defined, it is evaluated for every *insn* during mode switching. It determines the mode that an insn results in (if different from the incoming mode).

`MODE_ENTRY (`*entity*`)` [Macro]

If this macro is defined, it is evaluated for every *entity* that needs mode switching. It should evaluate to an integer, which is a mode that *entity* is assumed to be switched to at function entry. If `MODE_ENTRY` is defined then `MODE_EXIT` must be defined.

`MODE_EXIT (`*entity*`)` [Macro]

If this macro is defined, it is evaluated for every *entity* that needs mode switching. It should evaluate to an integer, which is a mode that *entity* is assumed to be switched to at function exit. If `MODE_EXIT` is defined then `MODE_ENTRY` must be defined.

`MODE_PRIORITY_TO_MODE (entity, n)` [Macro]
This macro specifies the order in which modes for *entity* are processed. 0 is the highest priority, `NUM_MODES_FOR_MODE_SWITCHING[entity] - 1` the lowest. The value of the macro should be an integer designating a mode for *entity*. For any fixed *entity*, `mode_priority_to_mode` (*entity*, *n*) shall be a bijection in 0 ... `num_modes_for_mode_switching[entity] - 1`.

`EMIT_MODE_SET (entity, mode, hard_regs_live)` [Macro]
Generate one or more insns to set *entity* to *mode*. *hard_reg_live* is the set of hard registers live at the point where the insn(s) are to be inserted.

14.23 Defining target-specific uses of `__attribute__`

Target-specific attributes may be defined for functions, data and types. These are described using the following target hooks; they also need to be documented in '`extend.texi`'.

`const struct attribute_spec * TARGET_ATTRIBUTE_TABLE` [Target Hook]
If defined, this target hook points to an array of '`struct attribute_spec`' (defined in '`tree.h`') specifying the machine specific attributes for this target and some of the restrictions on the entities to which these attributes are applied and the arguments they take.

`int TARGET_COMP_TYPE_ATTRIBUTES (tree type1, tree type2)` [Target Hook]
If defined, this target hook is a function which returns zero if the attributes on *type1* and *type2* are incompatible, one if they are compatible, and two if they are nearly compatible (which causes a warning to be generated). If this is not defined, machine-specific attributes are supposed always to be compatible.

`void TARGET_SET_DEFAULT_TYPE_ATTRIBUTES (tree type)` [Target Hook]
If defined, this target hook is a function which assigns default attributes to newly defined *type*.

`tree TARGET_MERGE_TYPE_ATTRIBUTES (tree type1, tree type2)` [Target Hook]
Define this target hook if the merging of type attributes needs special handling. If defined, the result is a list of the combined `TYPE_ATTRIBUTES` of *type1* and *type2*. It is assumed that `comptypes` has already been called and returned 1. This function may call `merge_attributes` to handle machine-independent merging.

`tree TARGET_MERGE_DECL_ATTRIBUTES (tree olddecl, tree newdecl)` [Target Hook]
Define this target hook if the merging of decl attributes needs special handling. If defined, the result is a list of the combined `DECL_ATTRIBUTES` of *olddecl* and *newdecl*. *newdecl* is a duplicate declaration of *olddecl*. Examples of when this is needed are when one attribute overrides another, or when an attribute is nullified by a subsequent definition. This function may call `merge_attributes` to handle machine-independent merging.

If the only target-specific handling you require is '`dllimport`' for Microsoft Windows targets, you should define the macro `TARGET_DLLIMPORT_DECL_ATTRIBUTES` to `1`. The compiler will then define a function called `merge_dllimport_decl_attributes`

which can then be defined as the expansion of `TARGET_MERGE_DECL_ATTRIBUTES`. You can also add `handle_dll_attribute` in the attribute table for your port to perform initial processing of the '`dllimport`' and '`dllexport`' attributes. This is done in '`i386/cygwin.h`' and '`i386/i386.c`', for example.

`bool TARGET_VALID_DLLIMPORT_ATTRIBUTE_P (`*tree decl*`)` [Target Hook]
decl is a variable or function with `__attribute__((dllimport))` specified. Use this hook if the target needs to add extra validation checks to `handle_dll_attribute`.

`TARGET_DECLSPEC` [Macro]
Define this macro to a nonzero value if you want to treat `__declspec(X)` as equivalent to `__attribute((X))`. By default, this behavior is enabled only for targets that define `TARGET_DLLIMPORT_DECL_ATTRIBUTES`. The current implementation of `__declspec` is via a built-in macro, but you should not rely on this implementation detail.

`void TARGET_INSERT_ATTRIBUTES (`*tree node*, *tree *attr_ptr*`)` [Target Hook]
Define this target hook if you want to be able to add attributes to a decl when it is being created. This is normally useful for back ends which wish to implement a pragma by using the attributes which correspond to the pragma's effect. The *node* argument is the decl which is being created. The *attr_ptr* argument is a pointer to the attribute list for this decl. The list itself should not be modified, since it may be shared with other decls, but attributes may be chained on the head of the list and `*attr_ptr` modified to point to the new attributes, or a copy of the list may be made if further changes are needed.

`bool TARGET_FUNCTION_ATTRIBUTE_INLINABLE_P (`*tree fndecl*`)` [Target Hook]
This target hook returns `true` if it is ok to inline *fndecl* into the current function, despite its having target-specific attributes, `false` otherwise. By default, if a function has a target specific attribute attached to it, it will not be inlined.

14.24 Defining coprocessor specifics for MIPS targets.

The MIPS specification allows MIPS implementations to have as many as 4 coprocessors, each with as many as 32 private registers. GCC supports accessing these registers and transferring values between the registers and memory using asm-ized variables. For example:

```
register unsigned int cp0count asm ("c0r1");
unsigned int d;

d = cp0count + 3;
```

("c0r1" is the default name of register 1 in coprocessor 0; alternate names may be added as described below, or the default names may be overridden entirely in `SUBTARGET_CONDITIONAL_REGISTER_USAGE`.)

Coprocessor registers are assumed to be epilogue-used; sets to them will be preserved even if it does not appear that the register is used again later in the function.

Another note: according to the MIPS spec, coprocessor 1 (if present) is the FPU. One accesses COP1 registers through standard mips floating-point support; they are not included in this mechanism.

There is one macro used in defining the MIPS coprocessor interface which you may want to override in subtargets; it is described below.

ALL_COP_ADDITIONAL_REGISTER_NAMES [Macro]
A comma-separated list (with leading comma) of pairs describing the alternate names of coprocessor registers. The format of each entry should be

```
{ alternatename, register_number}
```

Default: empty.

14.25 Parameters for Precompiled Header Validity Checking

void *TARGET_GET_PCH_VALIDITY (*size_t *sz*) [Target Hook]
This hook returns the data needed by TARGET_PCH_VALID_P and sets '*sz' to the size of the data in bytes.

const char *TARGET_PCH_VALID_P (*const void *data*, *size_t sz*) [Target Hook]
This hook checks whether the options used to create a PCH file are compatible with the current settings. It returns NULL if so and a suitable error message if not. Error messages will be presented to the user and must be localized using '_(*msg*)'.

data is the data that was returned by TARGET_GET_PCH_VALIDITY when the PCH file was created and *sz* is the size of that data in bytes. It's safe to assume that the data was created by the same version of the compiler, so no format checking is needed.

The default definition of default_pch_valid_p should be suitable for most targets.

const char *TARGET_CHECK_PCH_TARGET_FLAGS (*int pch_flags*) [Target Hook]
If this hook is nonnull, the default implementation of TARGET_PCH_VALID_P will use it to check for compatible values of target_flags. *pch_flags* specifies the value that target_flags had when the PCH file was created. The return value is the same as for TARGET_PCH_VALID_P.

14.26 C++ ABI parameters

tree TARGET_CXX_GUARD_TYPE (*void*) [Target Hook]
Define this hook to override the integer type used for guard variables. These are used to implement one-time construction of static objects. The default is long_long_integer_type_node.

bool TARGET_CXX_GUARD_MASK_BIT (*void*) [Target Hook]
This hook determines how guard variables are used. It should return false (the default) if first byte should be used. A return value of true indicates the least significant bit should be used.

tree TARGET_CXX_GET_COOKIE_SIZE (*tree type*) [Target Hook]
This hook returns the size of the cookie to use when allocating an array whose elements have the indicated *type*. Assumes that it is already known that a cookie is needed. The default is max(sizeof (size_t), alignof(type)), as defined in section 2.7 of the IA64/Generic C++ ABI.

bool TARGET_CXX_COOKIE_HAS_SIZE (*void*) [Target Hook]
This hook should return true if the element size should be stored in array cookies. The default is to return false.

`int TARGET_CXX_IMPORT_EXPORT_CLASS (`*tree* *type*, *int* *import_export*`)` [Target Hook]

If defined by a backend this hook allows the decision made to export class *type* to be overruled. Upon entry *import_export* will contain 1 if the class is going to be exported, −1 if it is going to be imported and 0 otherwise. This function should return the modified value and perform any other actions necessary to support the backend's targeted operating system.

`bool TARGET_CXX_CDTOR_RETURNS_THIS (`*void*`)` [Target Hook]

This hook should return `true` if constructors and destructors return the address of the object created/destroyed. The default is to return `false`.

`bool TARGET_CXX_KEY_METHOD_MAY_BE_INLINE (`*void*`)` [Target Hook]

This hook returns true if the key method for a class (i.e., the method which, if defined in the current translation unit, causes the virtual table to be emitted) may be an inline function. Under the standard Itanium C++ ABI the key method may be an inline function so long as the function is not declared inline in the class definition. Under some variants of the ABI, an inline function can never be the key method. The default is to return `true`.

`void TARGET_CXX_DETERMINE_CLASS_DATA_VISIBILITY (`*tree* *decl*`)` [Target Hook]

decl is a virtual table, virtual table table, typeinfo object, or other similar implicit class data object that will be emitted with external linkage in this translation unit. No ELF visibility has been explicitly specified. If the target needs to specify a visibility other than that of the containing class, use this hook to set `DECL_VISIBILITY` and `DECL_VISIBILITY_SPECIFIED`.

`bool TARGET_CXX_CLASS_DATA_ALWAYS_COMDAT (`*void*`)` [Target Hook]

This hook returns true (the default) if virtual tables and other similar implicit class data objects are always COMDAT if they have external linkage. If this hook returns false, then class data for classes whose virtual table will be emitted in only one translation unit will not be COMDAT.

`bool TARGET_CXX_USE_AEABI_ATEXIT (`*void*`)` [Target Hook]

This hook returns true if `__aeabi_atexit` (as defined by the ARM EABI) should be used to register static destructors when '`-fuse-cxa-atexit`' is in effect. The default is to return false to use `__cxa_atexit`.

`void TARGET_CXX_ADJUST_CLASS_AT_DEFINITION (`*tree* *type*`)` [Target Hook]

type is a C++ class (i.e., RECORD_TYPE or UNION_TYPE) that has just been defined. Use this hook to make adjustments to the class (eg, tweak visibility or perform any other required target modifications).

14.27 Miscellaneous Parameters

Here are several miscellaneous parameters.

`HAS_LONG_COND_BRANCH` [Macro]

Define this boolean macro to indicate whether or not your architecture has conditional branches that can span all of memory. It is used in conjunction with an optimization

that partitions hot and cold basic blocks into separate sections of the executable. If this macro is set to false, gcc will convert any conditional branches that attempt to cross between sections into unconditional branches or indirect jumps.

`HAS_LONG_UNCOND_BRANCH` [Macro]
Define this boolean macro to indicate whether or not your architecture has unconditional branches that can span all of memory. It is used in conjunction with an optimization that partitions hot and cold basic blocks into separate sections of the executable. If this macro is set to false, gcc will convert any unconditional branches that attempt to cross between sections into indirect jumps.

`CASE_VECTOR_MODE` [Macro]
An alias for a machine mode name. This is the machine mode that elements of a jump-table should have.

`CASE_VECTOR_SHORTEN_MODE (`*`min_offset`*`,` *`max_offset`*`,` *`body`*`)` [Macro]
Optional: return the preferred mode for an `addr_diff_vec` when the minimum and maximum offset are known. If you define this, it enables extra code in branch shortening to deal with `addr_diff_vec`. To make this work, you also have to define `INSN_ALIGN` and make the alignment for `addr_diff_vec` explicit. The *body* argument is provided so that the offset_unsigned and scale flags can be updated.

`CASE_VECTOR_PC_RELATIVE` [Macro]
Define this macro to be a C expression to indicate when jump-tables should contain relative addresses. You need not define this macro if jump-tables never contain relative addresses, or jump-tables should contain relative addresses only when '`-fPIC`' or '`-fPIC`' is in effect.

`CASE_VALUES_THRESHOLD` [Macro]
Define this to be the smallest number of different values for which it is best to use a jump-table instead of a tree of conditional branches. The default is four for machines with a `casesi` instruction and five otherwise. This is best for most machines.

`CASE_USE_BIT_TESTS` [Macro]
Define this macro to be a C expression to indicate whether C switch statements may be implemented by a sequence of bit tests. This is advantageous on processors that can efficiently implement left shift of 1 by the number of bits held in a register, but inappropriate on targets that would require a loop. By default, this macro returns `true` if the target defines an `ashlsi3` pattern, and `false` otherwise.

`WORD_REGISTER_OPERATIONS` [Macro]
Define this macro if operations between registers with integral mode smaller than a word are always performed on the entire register. Most RISC machines have this property and most CISC machines do not.

`LOAD_EXTEND_OP (`*`mem_mode`*`)` [Macro]
Define this macro to be a C expression indicating when insns that read memory in *mem_mode*, an integral mode narrower than a word, set the bits outside of *mem_mode*

to be either the sign-extension or the zero-extension of the data read. Return `SIGN_EXTEND` for values of *mem_mode* for which the insn sign-extends, `ZERO_EXTEND` for which it zero-extends, and `UNKNOWN` for other modes.

This macro is not called with *mem_mode* non-integral or with a width greater than or equal to `BITS_PER_WORD`, so you may return any value in this case. Do not define this macro if it would always return `UNKNOWN`. On machines where this macro is defined, you will normally define it as the constant `SIGN_EXTEND` or `ZERO_EXTEND`.

You may return a non-`UNKNOWN` value even if for some hard registers the sign extension is not performed, if for the `REGNO_REG_CLASS` of these hard registers `CANNOT_CHANGE_MODE_CLASS` returns nonzero when the *from* mode is *mem_mode* and the *to* mode is any integral mode larger than this but not larger than `word_mode`.

You must return `UNKNOWN` if for some hard registers that allow this mode, `CANNOT_CHANGE_MODE_CLASS` says that they cannot change to `word_mode`, but that they can change to another integral mode that is larger then *mem_mode* but still smaller than `word_mode`.

`SHORT_IMMEDIATES_SIGN_EXTEND` [Macro]

Define this macro if loading short immediate values into registers sign extends.

`FIXUNS_TRUNC_LIKE_FIX_TRUNC` [Macro]

Define this macro if the same instructions that convert a floating point number to a signed fixed point number also convert validly to an unsigned one.

`int TARGET_MIN_DIVISIONS_FOR_RECIP_MUL (`*enum machine_mode mode*`)` [Target Hook]

When '`-ffast-math`' is in effect, GCC tries to optimize divisions by the same divisor, by turning them into multiplications by the reciprocal. This target hook specifies the minimum number of divisions that should be there for GCC to perform the optimization for a variable of mode *mode*. The default implementation returns 3 if the machine has an instruction for the division, and 2 if it does not.

`MOVE_MAX` [Macro]

The maximum number of bytes that a single instruction can move quickly between memory and registers or between two memory locations.

`MAX_MOVE_MAX` [Macro]

The maximum number of bytes that a single instruction can move quickly between memory and registers or between two memory locations. If this is undefined, the default is `MOVE_MAX`. Otherwise, it is the constant value that is the largest value that `MOVE_MAX` can have at run-time.

`SHIFT_COUNT_TRUNCATED` [Macro]

A C expression that is nonzero if on this machine the number of bits actually used for the count of a shift operation is equal to the number of bits needed to represent the size of the object being shifted. When this macro is nonzero, the compiler will assume that it is safe to omit a sign-extend, zero-extend, and certain bitwise 'and' instructions that truncates the count of a shift operation. On machines that have instructions that act on bit-fields at variable positions, which may include 'bit test'

instructions, a nonzero `SHIFT_COUNT_TRUNCATED` also enables deletion of truncations of the values that serve as arguments to bit-field instructions.

If both types of instructions truncate the count (for shifts) and position (for bit-field operations), or if no variable-position bit-field instructions exist, you should define this macro.

However, on some machines, such as the 80386 and the 680x0, truncation only applies to shift operations and not the (real or pretended) bit-field operations. Define `SHIFT_COUNT_TRUNCATED` to be zero on such machines. Instead, add patterns to the '`md`' file that include the implied truncation of the shift instructions.

You need not define this macro if it would always have the value of zero.

`int TARGET_SHIFT_TRUNCATION_MASK (`*enum machine_mode* `mode)` [Target Hook]
This function describes how the standard shift patterns for *mode* deal with shifts by negative amounts or by more than the width of the mode. See [shift patterns], page 216.

On many machines, the shift patterns will apply a mask m to the shift count, meaning that a fixed-width shift of x by y is equivalent to an arbitrary-width shift of x by y & m. If this is true for mode *mode*, the function should return m, otherwise it should return 0. A return value of 0 indicates that no particular behavior is guaranteed.

Note that, unlike `SHIFT_COUNT_TRUNCATED`, this function does *not* apply to general shift rtxes; it applies only to instructions that are generated by the named shift patterns.

The default implementation of this function returns `GET_MODE_BITSIZE (`*mode*`) - 1` if `SHIFT_COUNT_TRUNCATED` and 0 otherwise. This definition is always safe, but if `SHIFT_COUNT_TRUNCATED` is false, and some shift patterns nevertheless truncate the shift count, you may get better code by overriding it.

`TRULY_NOOP_TRUNCATION (`*outprec*, *inprec*`)` [Macro]
A C expression which is nonzero if on this machine it is safe to "convert" an integer of *inprec* bits to one of *outprec* bits (where *outprec* is smaller than *inprec*) by merely operating on it as if it had only *outprec* bits.

On many machines, this expression can be 1.

When `TRULY_NOOP_TRUNCATION` returns 1 for a pair of sizes for modes for which `MODES_TIEABLE_P` is 0, suboptimal code can result. If this is the case, making `TRULY_NOOP_TRUNCATION` return 0 in such cases may improve things.

`STORE_FLAG_VALUE` [Macro]
A C expression describing the value returned by a comparison operator with an integral mode and stored by a store-flag instruction ('`scond`') when the condition is true. This description must apply to *all* the '`scond`' patterns and all the comparison operators whose results have a `MODE_INT` mode.

A value of 1 or -1 means that the instruction implementing the comparison operator returns exactly 1 or -1 when the comparison is true and 0 when the comparison is false. Otherwise, the value indicates which bits of the result are guaranteed to be 1 when the comparison is true. This value is interpreted in the mode of the comparison

operation, which is given by the mode of the first operand in the '*scond*' pattern. Either the low bit or the sign bit of STORE_FLAG_VALUE be on. Presently, only those bits are used by the compiler.

If STORE_FLAG_VALUE is neither 1 or −1, the compiler will generate code that depends only on the specified bits. It can also replace comparison operators with equivalent operations if they cause the required bits to be set, even if the remaining bits are undefined. For example, on a machine whose comparison operators return an SImode value and where STORE_FLAG_VALUE is defined as '0x80000000', saying that just the sign bit is relevant, the expression

```
(ne:SI (and:SI x (const_int power-of-2)) (const_int 0))
```

can be converted to

```
(ashift:SI x (const_int n))
```

where *n* is the appropriate shift count to move the bit being tested into the sign bit.

There is no way to describe a machine that always sets the low-order bit for a true value, but does not guarantee the value of any other bits, but we do not know of any machine that has such an instruction. If you are trying to port GCC to such a machine, include an instruction to perform a logical-and of the result with 1 in the pattern for the comparison operators and let us know at gcc@gcc.gnu.org.

Often, a machine will have multiple instructions that obtain a value from a comparison (or the condition codes). Here are rules to guide the choice of value for STORE_FLAG_VALUE, and hence the instructions to be used:

- Use the shortest sequence that yields a valid definition for STORE_FLAG_VALUE. It is more efficient for the compiler to "normalize" the value (convert it to, e.g., 1 or 0) than for the comparison operators to do so because there may be opportunities to combine the normalization with other operations.
- For equal-length sequences, use a value of 1 or −1, with −1 being slightly preferred on machines with expensive jumps and 1 preferred on other machines.
- As a second choice, choose a value of '0x80000001' if instructions exist that set both the sign and low-order bits but do not define the others.
- Otherwise, use a value of '0x80000000'.

Many machines can produce both the value chosen for STORE_FLAG_VALUE and its negation in the same number of instructions. On those machines, you should also define a pattern for those cases, e.g., one matching

```
(set A (neg:m (ne:m B C)))
```

Some machines can also perform and or plus operations on condition code values with less instructions than the corresponding '*scond*' insn followed by and or plus. On those machines, define the appropriate patterns. Use the names incscc and decscc, respectively, for the patterns which perform plus or minus operations on condition code values. See 'rs6000.md' for some examples. The GNU Superoptizer can be used to find such instruction sequences on other machines.

If this macro is not defined, the default value, 1, is used. You need not define STORE_FLAG_VALUE if the machine has no store-flag instructions, or if the value generated by these instructions is 1.

`FLOAT_STORE_FLAG_VALUE (mode)` [Macro]

A C expression that gives a nonzero `REAL_VALUE_TYPE` value that is returned when comparison operators with floating-point results are true. Define this macro on machines that have comparison operations that return floating-point values. If there are no such operations, do not define this macro.

`VECTOR_STORE_FLAG_VALUE (mode)` [Macro]

A C expression that gives a rtx representing the nonzero true element for vector comparisons. The returned rtx should be valid for the inner mode of *mode* which is guaranteed to be a vector mode. Define this macro on machines that have vector comparison operations that return a vector result. If there are no such operations, do not define this macro. Typically, this macro is defined as `const1_rtx` or `constm1_rtx`. This macro may return `NULL_RTX` to prevent the compiler optimizing such vector comparison operations for the given mode.

`CLZ_DEFINED_VALUE_AT_ZERO (mode, value)` [Macro]
`CTZ_DEFINED_VALUE_AT_ZERO (mode, value)` [Macro]

A C expression that evaluates to true if the architecture defines a value for `clz` or `ctz` with a zero operand. If so, *value* should be set to this value. If this macro is not defined, the value of `clz` or `ctz` is assumed to be undefined.

This macro must be defined if the target's expansion for `ffs` relies on a particular value to get correct results. Otherwise it is not necessary, though it may be used to optimize some corner cases.

Note that regardless of this macro the "definedness" of `clz` and `ctz` at zero do *not* extend to the builtin functions visible to the user. Thus one may be free to adjust the value at will to match the target expansion of these operations without fear of breaking the API.

`Pmode` [Macro]

An alias for the machine mode for pointers. On most machines, define this to be the integer mode corresponding to the width of a hardware pointer; `SImode` on 32-bit machine or `DImode` on 64-bit machines. On some machines you must define this to be one of the partial integer modes, such as `PSImode`.

The width of `Pmode` must be at least as large as the value of `POINTER_SIZE`. If it is not equal, you must define the macro `POINTERS_EXTEND_UNSIGNED` to specify how pointers are extended to `Pmode`.

`FUNCTION_MODE` [Macro]

An alias for the machine mode used for memory references to functions being called, in `call` RTL expressions. On most machines this should be `QImode`.

`STDC_0_IN_SYSTEM_HEADERS` [Macro]

In normal operation, the preprocessor expands `__STDC__` to the constant 1, to signify that GCC conforms to ISO Standard C. On some hosts, like Solaris, the system compiler uses a different convention, where `__STDC__` is normally 0, but is 1 if the user specifies strict conformance to the C Standard.

Defining `STDC_0_IN_SYSTEM_HEADERS` makes GNU CPP follows the host convention when processing system header files, but when processing user files `__STDC__` will always expand to 1.

`NO_IMPLICIT_EXTERN_C` [Macro]

Define this macro if the system header files support C++ as well as C. This macro inhibits the usual method of using system header files in C++, which is to pretend that the file's contents are enclosed in '`extern "C" {...}`'.

`REGISTER_TARGET_PRAGMAS ()` [Macro]

Define this macro if you want to implement any target-specific pragmas. If defined, it is a C expression which makes a series of calls to `c_register_pragma` or `c_register_pragma_with_expansion` for each pragma. The macro may also do any setup required for the pragmas.

The primary reason to define this macro is to provide compatibility with other compilers for the same target. In general, we discourage definition of target-specific pragmas for GCC.

If the pragma can be implemented by attributes then you should consider defining the target hook '`TARGET_INSERT_ATTRIBUTES`' as well.

Preprocessor macros that appear on pragma lines are not expanded. All '`#pragma`' directives that do not match any registered pragma are silently ignored, unless the user specifies '`-Wunknown-pragmas`'.

`void c_register_pragma (`*const char *`**`space`*, *const char *`**`name`*, *void (*`**`callback`*) (*struct cpp_reader **)) [Function]

`void c_register_pragma_with_expansion (`*const char *`**`space`*, *const char *`**`name`*, *void (*`**`callback`*) (*struct cpp_reader **)) [Function]

Each call to `c_register_pragma` or `c_register_pragma_with_expansion` establishes one pragma. The *callback* routine will be called when the preprocessor encounters a pragma of the form

```
#pragma [space] name ...
```

space is the case-sensitive namespace of the pragma, or `NULL` to put the pragma in the global namespace. The callback routine receives *pfile* as its first argument, which can be passed on to cpplib's functions if necessary. You can lex tokens after the *name* by calling `c_lex`. Tokens that are not read by the callback will be silently ignored. The end of the line is indicated by a token of type `CPP_EOF`. Macro expansion occurs on the arguments of pragmas registered with `c_register_pragma_with_expansion` but not on the arguments of pragmas registered with `c_register_pragma`.

For an example use of this routine, see '`c4x.h`' and the callback routines defined in '`c4x-c.c`'.

Note that the use of `c_lex` is specific to the C and C++ compilers. It will not work in the Java or Fortran compilers, or any other language compilers for that matter. Thus if `c_lex` is going to be called from target-specific code, it must only be done so when building the C and C++ compilers. This can be done by defining the variables `c_target_objs` and `cxx_target_objs` in the target entry in the '`config.gcc`' file. These variables should name the target-specific, language-specific object file which contains the code that uses `c_lex`. Note it will also be necessary to add a rule to the makefile fragment pointed to by `tmake_file` that shows how to build this object file.

`HANDLE_SYSV_PRAGMA` [Macro]

Define this macro (to a value of 1) if you want the System V style pragmas '`#pragma pack(<n>)`' and '`#pragma weak <name> [=<value>]`' to be supported by gcc.

The pack pragma specifies the maximum alignment (in bytes) of fields within a structure, in much the same way as the '`__aligned__`' and '`__packed__`' `__attribute__`s do. A pack value of zero resets the behavior to the default.

A subtlety for Microsoft Visual C/C++ style bit-field packing (e.g. -mms-bitfields) for targets that support it: When a bit-field is inserted into a packed record, the whole size of the underlying type is used by one or more same-size adjacent bit-fields (that is, if its long:3, 32 bits is used in the record, and any additional adjacent long bit-fields are packed into the same chunk of 32 bits. However, if the size changes, a new field of that size is allocated).

If both MS bit-fields and '`__attribute__((packed))`' are used, the latter will take precedence. If '`__attribute__((packed))`' is used on a single field when MS bit-fields are in use, it will take precedence for that field, but the alignment of the rest of the structure may affect its placement.

The weak pragma only works if `SUPPORTS_WEAK` and `ASM_WEAKEN_LABEL` are defined. If enabled it allows the creation of specifically named weak labels, optionally with a value.

`HANDLE_PRAGMA_PACK_PUSH_POP` [Macro]

Define this macro (to a value of 1) if you want to support the Win32 style pragmas '`#pragma pack(push[,n])`' and '`#pragma pack(pop)`'. The '`pack(push,[n])`' pragma specifies the maximum alignment (in bytes) of fields within a structure, in much the same way as the '`__aligned__`' and '`__packed__`' `__attribute__`s do. A pack value of zero resets the behavior to the default. Successive invocations of this pragma cause the previous values to be stacked, so that invocations of '`#pragma pack(pop)`' will return to the previous value.

`HANDLE_PRAGMA_PACK_WITH_EXPANSION` [Macro]

Define this macro, as well as `HANDLE_SYSV_PRAGMA`, if macros should be expanded in the arguments of '`#pragma pack`'.

`TARGET_DEFAULT_PACK_STRUCT` [Macro]

If your target requires a structure packing default other than 0 (meaning the machine default), define this macro to the necessary value (in bytes). This must be a value that would also valid to be used with '`#pragma pack()`' (that is, a small power of two).

`DOLLARS_IN_IDENTIFIERS` [Macro]

Define this macro to control use of the character '`$`' in identifier names for the C family of languages. 0 means '`$`' is not allowed by default; 1 means it is allowed. 1 is the default; there is no need to define this macro in that case.

`NO_DOLLAR_IN_LABEL` [Macro]

Define this macro if the assembler does not accept the character '`$`' in label names. By default constructors and destructors in G++ have '`$`' in the identifiers. If this macro is defined, '`.`' is used instead.

`NO_DOT_IN_LABEL` [Macro]

Define this macro if the assembler does not accept the character '`.`' in label names. By default constructors and destructors in G++ have names that use '`.`'. If this macro is defined, these names are rewritten to avoid '`.`'.

`INSN_SETS_ARE_DELAYED (`*`insn`*`)` [Macro]

Define this macro as a C expression that is nonzero if it is safe for the delay slot scheduler to place instructions in the delay slot of *insn*, even if they appear to use a resource set or clobbered in *insn*. *insn* is always a `jump_insn` or an `insn`; GCC knows that every `call_insn` has this behavior. On machines where some `insn` or `jump_insn` is really a function call and hence has this behavior, you should define this macro.

You need not define this macro if it would always return zero.

`INSN_REFERENCES_ARE_DELAYED (`*`insn`*`)` [Macro]

Define this macro as a C expression that is nonzero if it is safe for the delay slot scheduler to place instructions in the delay slot of *insn*, even if they appear to set or clobber a resource referenced in *insn*. *insn* is always a `jump_insn` or an `insn`. On machines where some `insn` or `jump_insn` is really a function call and its operands are registers whose use is actually in the subroutine it calls, you should define this macro. Doing so allows the delay slot scheduler to move instructions which copy arguments into the argument registers into the delay slot of *insn*.

You need not define this macro if it would always return zero.

`MULTIPLE_SYMBOL_SPACES` [Macro]

Define this macro as a C expression that is nonzero if, in some cases, global symbols from one translation unit may not be bound to undefined symbols in another translation unit without user intervention. For instance, under Microsoft Windows symbols must be explicitly imported from shared libraries (DLLs).

You need not define this macro if it would always evaluate to zero.

`tree TARGET_MD_ASM_CLOBBERS (`*tree* *`outputs`*`,` *tree* *`inputs`*`,` *tree* *`clobbers`*`)` [Target Hook]

This target hook should add to *clobbers* `STRING_CST` trees for any hard regs the port wishes to automatically clobber for an asm. It should return the result of the last `tree_cons` used to add a clobber. The *outputs*, *inputs* and *clobber* lists are the corresponding parameters to the asm and may be inspected to avoid clobbering a register that is an input or output of the asm. You can use `tree_overlaps_hard_reg_set`, declared in '`tree.h`', to test for overlap with regards to asm-declared registers.

`MATH_LIBRARY` [Macro]

Define this macro as a C string constant for the linker argument to link in the system math library, or '`""`' if the target does not have a separate math library.

You need only define this macro if the default of '`"-lm"`' is wrong.

`LIBRARY_PATH_ENV` [Macro]

Define this macro as a C string constant for the environment variable that specifies where the linker should look for libraries.

You need only define this macro if the default of '`"LIBRARY_PATH"`' is wrong.

`TARGET_POSIX_IO` [Macro]

Define this macro if the target supports the following POSIX file functions, access, mkdir and file locking with fcntl / F_SETLKW. Defining `TARGET_POSIX_IO` will enable the test coverage code to use file locking when exiting a program, which avoids race conditions if the program has forked. It will also create directories at run-time for cross-profiling.

`MAX_CONDITIONAL_EXECUTE` [Macro]

A C expression for the maximum number of instructions to execute via conditional execution instructions instead of a branch. A value of `BRANCH_COST+1` is the default if the machine does not use cc0, and 1 if it does use cc0.

`IFCVT_MODIFY_TESTS (`*`ce_info`*`, `*`true_expr`*`, `*`false_expr`*`)` [Macro]

Used if the target needs to perform machine-dependent modifications on the conditionals used for turning basic blocks into conditionally executed code. *ce_info* points to a data structure, `struct ce_if_block`, which contains information about the currently processed blocks. *true_expr* and *false_expr* are the tests that are used for converting the then-block and the else-block, respectively. Set either *true_expr* or *false_expr* to a null pointer if the tests cannot be converted.

`IFCVT_MODIFY_MULTIPLE_TESTS (`*`ce_info`*`, `*`bb`*`, `*`true_expr`*`, `*`false_expr`*`)` [Macro]

Like `IFCVT_MODIFY_TESTS`, but used when converting more complicated if-statements into conditions combined by `and` and `or` operations. *bb* contains the basic block that contains the test that is currently being processed and about to be turned into a condition.

`IFCVT_MODIFY_INSN (`*`ce_info`*`, `*`pattern`*`, `*`insn`*`)` [Macro]

A C expression to modify the *PATTERN* of an *INSN* that is to be converted to conditional execution format. *ce_info* points to a data structure, `struct ce_if_block`, which contains information about the currently processed blocks.

`IFCVT_MODIFY_FINAL (`*`ce_info`*`)` [Macro]

A C expression to perform any final machine dependent modifications in converting code to conditional execution. The involved basic blocks can be found in the `struct ce_if_block` structure that is pointed to by *ce_info*.

`IFCVT_MODIFY_CANCEL (`*`ce_info`*`)` [Macro]

A C expression to cancel any machine dependent modifications in converting code to conditional execution. The involved basic blocks can be found in the `struct ce_if_block` structure that is pointed to by *ce_info*.

`IFCVT_INIT_EXTRA_FIELDS (`*`ce_info`*`)` [Macro]

A C expression to initialize any extra fields in a `struct ce_if_block` structure, which are defined by the `IFCVT_EXTRA_FIELDS` macro.

`IFCVT_EXTRA_FIELDS` [Macro]

If defined, it should expand to a set of field declarations that will be added to the `struct ce_if_block` structure. These should be initialized by the `IFCVT_INIT_EXTRA_FIELDS` macro.

`void TARGET_MACHINE_DEPENDENT_REORG ()` [Target Hook]

If non-null, this hook performs a target-specific pass over the instruction stream. The compiler will run it at all optimization levels, just before the point at which it normally does delayed-branch scheduling.

The exact purpose of the hook varies from target to target. Some use it to do transformations that are necessary for correctness, such as laying out in-function constant pools or avoiding hardware hazards. Others use it as an opportunity to do some machine-dependent optimizations.

You need not implement the hook if it has nothing to do. The default definition is null.

`void TARGET_INIT_BUILTINS ()` [Target Hook]

Define this hook if you have any machine-specific built-in functions that need to be defined. It should be a function that performs the necessary setup.

Machine specific built-in functions can be useful to expand special machine instructions that would otherwise not normally be generated because they have no equivalent in the source language (for example, SIMD vector instructions or prefetch instructions).

To create a built-in function, call the function `lang_hooks.builtin_function` which is defined by the language front end. You can use any type nodes set up by `build_common_tree_nodes` and `build_common_tree_nodes_2`; only language front ends that use those two functions will call '`TARGET_INIT_BUILTINS`'.

`rtx TARGET_EXPAND_BUILTIN (`*tree* ***exp***, *rtx* ***target***, *rtx* ***subtarget***, *enum machine_mode* ***mode***, *int* ***ignore***`)` [Target Hook]

Expand a call to a machine specific built-in function that was set up by '`TARGET_INIT_BUILTINS`'. *exp* is the expression for the function call; the result should go to *target* if that is convenient, and have mode *mode* if that is convenient. *subtarget* may be used as the target for computing one of *exp*'s operands. *ignore* is nonzero if the value is to be ignored. This function should return the result of the call to the built-in function.

`tree TARGET_RESOLVE_OVERLOADED_BUILTIN (`*tree* ***fndecl***, *tree* ***arglist***`)` [Target Hook]

Select a replacement for a machine specific built-in function that was set up by '`TARGET_INIT_BUILTINS`'. This is done *before* regular type checking, and so allows the target to implement a crude form of function overloading. *fndecl* is the declaration of the built-in function. *arglist* is the list of arguments passed to the built-in function. The result is a complete expression that implements the operation, usually another `CALL_EXPR`.

`tree TARGET_FOLD_BUILTIN (`*tree* `fndecl`, *tree* `arglist`, *bool* `ignore``)` [Target Hook]

Fold a call to a machine specific built-in function that was set up by '`TARGET_INIT_BUILTINS`'. *fndecl* is the declaration of the built-in function. *arglist* is the list of arguments passed to the built-in function. The result is another tree containing a simplified expression for the call's result. If *ignore* is true the value will be ignored.

`const char *` *TARGET_INVALID_WITHIN_DOLOOP* `(`*rtx* `insn``)` [Target Hook]

Take an instruction in *insn* and return NULL if it is valid within a low-overhead loop, otherwise return a string why doloop could not be applied.

Many targets use special registers for low-overhead looping. For any instruction that clobbers these this function should return a string indicating the reason why the doloop could not be applied. By default, the RTL loop optimizer does not use a present doloop pattern for loops containing function calls or branch on table instructions.

`MD_CAN_REDIRECT_BRANCH (`*branch1*, *branch2*`)` [Macro]

Take a branch insn in *branch1* and another in *branch2*. Return true if redirecting *branch1* to the destination of *branch2* is possible.

On some targets, branches may have a limited range. Optimizing the filling of delay slots can result in branches being redirected, and this may in turn cause a branch offset to overflow.

`bool TARGET_COMMUTATIVE_P (`*rtx* `x`, *outer_code*`)` [Target Hook]

This target hook returns `true` if *x* is considered to be commutative. Usually, this is just COMMUTATIVE_P (*x*), but the HP PA doesn't consider PLUS to be commutative inside a MEM. *outer_code* is the rtx code of the enclosing rtl, if known, otherwise it is UNKNOWN.

`rtx TARGET_ALLOCATE_INITIAL_VALUE (`*rtx* `hard_reg``)` [Target Hook]

When the initial value of a hard register has been copied in a pseudo register, it is often not necessary to actually allocate another register to this pseudo register, because the original hard register or a stack slot it has been saved into can be used. `TARGET_ALLOCATE_INITIAL_VALUE` is called at the start of register allocation once for each hard register that had its initial value copied by using `get_func_hard_reg_initial_val` or `get_hard_reg_initial_val`. Possible values are `NULL_RTX`, if you don't want to do any special allocation, a `REG` rtx—that would typically be the hard register itself, if it is known not to be clobbered—or a `MEM`. If you are returning a `MEM`, this is only a hint for the allocator; it might decide to use another register anyways. You may use `current_function_leaf_function` in the hook, functions that use `REG_N_SETS`, to determine if the hard register in question will not be clobbered. The default value of this hook is `NULL`, which disables any special allocation.

`TARGET_OBJECT_SUFFIX` [Macro]

Define this macro to be a C string representing the suffix for object files on your target machine. If you do not define this macro, GCC will use '`.o`' as the suffix for object files.

`TARGET_EXECUTABLE_SUFFIX` [Macro]
Define this macro to be a C string representing the suffix to be automatically added to executable files on your target machine. If you do not define this macro, GCC will use the null string as the suffix for executable files.

`COLLECT_EXPORT_LIST` [Macro]
If defined, `collect2` will scan the individual object files specified on its command line and create an export list for the linker. Define this macro for systems like AIX, where the linker discards object files that are not referenced from `main` and uses export lists.

`MODIFY_JNI_METHOD_CALL (`*`mdecl`*`)` [Macro]
Define this macro to a C expression representing a variant of the method call *mdecl*, if Java Native Interface (JNI) methods must be invoked differently from other methods on your target. For example, on 32-bit Microsoft Windows, JNI methods must be invoked using the `stdcall` calling convention and this macro is then defined as this expression:

```
build_type_attribute_variant (mdecl,
                              build_tree_list
                              (get_identifier ("stdcall"),
                               NULL))
```

`bool TARGET_CANNOT_MODIFY_JUMPS_P (`*void*`)` [Target Hook]
This target hook returns `true` past the point in which new jump instructions could be created. On machines that require a register for every jump such as the SHmedia ISA of SH5, this point would typically be reload, so this target hook should be defined to a function such as:

```
static bool
cannot_modify_jumps_past_reload_p ()
{
  return (reload_completed || reload_in_progress);
}
```

`int TARGET_BRANCH_TARGET_REGISTER_CLASS (`*void*`)` [Target Hook]
This target hook returns a register class for which branch target register optimizations should be applied. All registers in this class should be usable interchangeably. After reload, registers in this class will be re-allocated and loads will be hoisted out of loops and be subjected to inter-block scheduling.

`bool TARGET_BRANCH_TARGET_REGISTER_CALLEE_SAVED (`*bool after_prologue_epilogue_gen*`)` [Target Hook]
Branch target register optimization will by default exclude callee-saved registers that are not already live during the current function; if this target hook returns true, they will be included. The target code must than make sure that all target registers in the class returned by '`TARGET_BRANCH_TARGET_REGISTER_CLASS`' that might need saving are saved. *after_prologue_epilogue_gen* indicates if prologues and epilogues have already been generated. Note, even if you only return true when *after_prologue_epilogue_gen* is false, you still are likely to have to make special provisions in `INITIAL_ELIMINATION_OFFSET` to reserve space for caller-saved target registers.

`POWI_MAX_MULTS` [Macro]

If defined, this macro is interpreted as a signed integer C expression that specifies the maximum number of floating point multiplications that should be emitted when expanding exponentiation by an integer constant inline. When this value is defined, exponentiation requiring more than this number of multiplications is implemented by calling the system library's `pow`, `powf` or `powl` routines. The default value places no upper bound on the multiplication count.

`void TARGET_EXTRA_INCLUDES (`*const char* ***`sysroot`**`, const char *`***`iprefix`**`, int` **`stdinc`**`)` [Macro]

This target hook should register any extra include files for the target. The parameter *stdinc* indicates if normal include files are present. The parameter *sysroot* is the system root directory. The parameter *iprefix* is the prefix for the gcc directory.

`void TARGET_EXTRA_PRE_INCLUDES (`*const char* ***`sysroot`**`, const char *`***`iprefix`**`, int` **`stdinc`**`)` [Macro]

This target hook should register any extra include files for the target before any standard headers. The parameter *stdinc* indicates if normal include files are present. The parameter *sysroot* is the system root directory. The parameter *iprefix* is the prefix for the gcc directory.

`void TARGET_OPTF (`*char* ***`path`**`)` [Macro]

This target hook should register special include paths for the target. The parameter *path* is the include to register. On Darwin systems, this is used for Framework includes, which have semantics that are different from '`-I`'.

`bool TARGET_USE_LOCAL_THUNK_ALIAS_P (`*tree* ***`fndecl`**`)` [Target Hook]

This target hook returns `true` if it is safe to use a local alias for a virtual function *fndecl* when constructing thunks, `false` otherwise. By default, the hook returns `true` for all functions, if a target supports aliases (i.e. defines `ASM_OUTPUT_DEF`), `false` otherwise,

`TARGET_FORMAT_TYPES` [Macro]

If defined, this macro is the name of a global variable containing target-specific format checking information for the '`-Wformat`' option. The default is to have no target-specific format checks.

`TARGET_N_FORMAT_TYPES` [Macro]

If defined, this macro is the number of entries in `TARGET_FORMAT_TYPES`.

`bool TARGET_RELAXED_ORDERING` [Target Hook]

If set to `true`, means that the target's memory model does not guarantee that loads which do not depend on one another will access main memory in the order of the instruction stream; if ordering is important, an explicit memory barrier must be used. This is true of many recent processors which implement a policy of "relaxed," "weak," or "release" memory consistency, such as Alpha, PowerPC, and ia64. The default is `false`.

`const char` [Target Hook]
**TARGET_INVALID_ARG_FOR_UNPROTOTYPED_FN* (*tree* *typelist*, *tree* *funcdecl*, *tree* *val*)

If defined, this macro returns the diagnostic message when it is illegal to pass argument *val* to function *funcdecl* with prototype *typelist*.

`const char * TARGET_INVALID_CONVERSION` (*tree* *fromtype*, *tree* *totype*) [Target Hook]

If defined, this macro returns the diagnostic message when it is invalid to convert from *fromtype* to *totype*, or `NULL` if validity should be determined by the front end.

`const char * TARGET_INVALID_UNARY_OP` (*int* *op*, *tree* *type*) [Target Hook]

If defined, this macro returns the diagnostic message when it is invalid to apply operation *op* (where unary plus is denoted by `CONVERT_EXPR`) to an operand of type *type*, or `NULL` if validity should be determined by the front end.

`const char * TARGET_INVALID_BINARY_OP` (*int* *op*, *tree* *type1*, *tree* *type2*) [Target Hook]

If defined, this macro returns the diagnostic message when it is invalid to apply operation *op* to operands of types *type1* and *type2*, or `NULL` if validity should be determined by the front end.

`TARGET_USE_JCR_SECTION` [Macro]

This macro determines whether to use the JCR section to register Java classes. By default, TARGET_USE_JCR_SECTION is defined to 1 if both SUPPORTS_WEAK and TARGET_HAVE_NAMED_SECTIONS are true, else 0.

15 Host Configuration

Most details about the machine and system on which the compiler is actually running are detected by the `configure` script. Some things are impossible for `configure` to detect; these are described in two ways, either by macros defined in a file named '`xm-machine.h`' or by hook functions in the file specified by the *out_host_hook_obj* variable in '`config.gcc`'. (The intention is that very few hosts will need a header file but nearly every fully supported host will need to override some hooks.)

If you need to define only a few macros, and they have simple definitions, consider using the `xm_defines` variable in your '`config.gcc`' entry instead of creating a host configuration header. See Section 6.3.2.2 [System Config], page 22.

15.1 Host Common

Some things are just not portable, even between similar operating systems, and are too difficult for autoconf to detect. They get implemented using hook functions in the file specified by the *host_hook_obj* variable in '`config.gcc`'.

`void HOST_HOOKS_EXTRA_SIGNALS (`*void*`)` [Host Hook]
This host hook is used to set up handling for extra signals. The most common thing to do in this hook is to detect stack overflow.

`void * `*HOST_HOOKS_GT_PCH_GET_ADDRESS* `(`*size_t* `size`, *int* `fd``)` [Host Hook]
This host hook returns the address of some space that is likely to be free in some subsequent invocation of the compiler. We intend to load the PCH data at this address such that the data need not be relocated. The area should be able to hold *size* bytes. If the host uses `mmap`, *fd* is an open file descriptor that can be used for probing.

`int HOST_HOOKS_GT_PCH_USE_ADDRESS (`*void * `address`, size_t* `size`, *int* `fd`, *size_t* `offset``)` [Host Hook]
This host hook is called when a PCH file is about to be loaded. We want to load *size* bytes from *fd* at *offset* into memory at *address*. The given address will be the result of a previous invocation of `HOST_HOOKS_GT_PCH_GET_ADDRESS`. Return −1 if we couldn't allocate *size* bytes at *address*. Return 0 if the memory is allocated but the data is not loaded. Return 1 if the hook has performed everything.

If the implementation uses reserved address space, free any reserved space beyond *size*, regardless of the return value. If no PCH will be loaded, this hook may be called with *size* zero, in which case all reserved address space should be freed.

Do not try to handle values of *address* that could not have been returned by this executable; just return −1. Such values usually indicate an out-of-date PCH file (built by some other GCC executable), and such a PCH file won't work.

`size_t HOST_HOOKS_GT_PCH_ALLOC_GRANULARITY (`*void*`);` [Host Hook]
This host hook returns the alignment required for allocating virtual memory. Usually this is the same as getpagesize, but on some hosts the alignment for reserving memory differs from the pagesize for committing memory.

15.2 Host Filesystem

GCC needs to know a number of things about the semantics of the host machine's filesystem. Filesystems with Unix and MS-DOS semantics are automatically detected. For other systems, you can define the following macros in '`xm-`*`machine`*`.h`'.

`HAVE_DOS_BASED_FILE_SYSTEM`
: This macro is automatically defined by '`system.h`' if the host file system obeys the semantics defined by MS-DOS instead of Unix. DOS file systems are case insensitive, file specifications may begin with a drive letter, and both forward slash and backslash ('`/`' and '`\`') are directory separators.

`DIR_SEPARATOR`
`DIR_SEPARATOR_2`
: If defined, these macros expand to character constants specifying separators for directory names within a file specification. '`system.h`' will automatically give them appropriate values on Unix and MS-DOS file systems. If your file system is neither of these, define one or both appropriately in '`xm-`*`machine`*`.h`'.

  However, operating systems like VMS, where constructing a pathname is more complicated than just stringing together directory names separated by a special character, should not define either of these macros.

`PATH_SEPARATOR`
: If defined, this macro should expand to a character constant specifying the separator for elements of search paths. The default value is a colon ('`:`'). DOS-based systems usually, but not always, use semicolon ('`;`').

`VMS` Define this macro if the host system is VMS.

`HOST_OBJECT_SUFFIX`
: Define this macro to be a C string representing the suffix for object files on your host machine. If you do not define this macro, GCC will use '`.o`' as the suffix for object files.

`HOST_EXECUTABLE_SUFFIX`
: Define this macro to be a C string representing the suffix for executable files on your host machine. If you do not define this macro, GCC will use the null string as the suffix for executable files.

`HOST_BIT_BUCKET`
: A pathname defined by the host operating system, which can be opened as a file and written to, but all the information written is discarded. This is commonly known as a *bit bucket* or *null device*. If you do not define this macro, GCC will use '`/dev/null`' as the bit bucket. If the host does not support a bit bucket, define this macro to an invalid filename.

`UPDATE_PATH_HOST_CANONICALIZE (`*`path`*`)`
: If defined, a C statement (sans semicolon) that performs host-dependent canonicalization when a path used in a compilation driver or preprocessor is canonicalized. *path* is a malloc-ed path to be canonicalized. If the C statement does canonicalize *path* into a different buffer, the old path should be freed and the new buffer should have been allocated with malloc.

`DUMPFILE_FORMAT`

Define this macro to be a C string representing the format to use for constructing the index part of debugging dump file names. The resultant string must fit in fifteen bytes. The full filename will be the concatenation of: the prefix of the assembler file name, the string resulting from applying this format to an index number, and a string unique to each dump file kind, e.g. '`rtl`'.

If you do not define this macro, GCC will use '`.%02d.`'. You should define this macro if using the default will create an invalid file name.

`DELETE_IF_ORDINARY`

Define this macro to be a C statement (sans semicolon) that performs host-dependent removal of ordinary temp files in the compilation driver.

If you do not define this macro, GCC will use the default version. You should define this macro if the default version does not reliably remove the temp file as, for example, on VMS which allows multiple versions of a file.

`HOST_LACKS_INODE_NUMBERS`

Define this macro if the host filesystem does not report meaningful inode numbers in struct stat.

15.3 Host Misc

`FATAL_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits after serious errors. The default is the system-provided macro '`EXIT_FAILURE`', or '`1`' if the system doesn't define that macro. Define this macro only if these defaults are incorrect.

`SUCCESS_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits without serious errors. (Warnings are not serious errors.) The default is the system-provided macro '`EXIT_SUCCESS`', or '`0`' if the system doesn't define that macro. Define this macro only if these defaults are incorrect.

`USE_C_ALLOCA`

Define this macro if GCC should use the C implementation of `alloca` provided by '`libiberty.a`'. This only affects how some parts of the compiler itself allocate memory. It does not change code generation.

When GCC is built with a compiler other than itself, the C `alloca` is always used. This is because most other implementations have serious bugs. You should define this macro only on a system where no stack-based `alloca` can possibly work. For instance, if a system has a small limit on the size of the stack, GCC's builtin `alloca` will not work reliably.

`COLLECT2_HOST_INITIALIZATION`

If defined, a C statement (sans semicolon) that performs host-dependent initialization when `collect2` is being initialized.

`GCC_DRIVER_HOST_INITIALIZATION`

If defined, a C statement (sans semicolon) that performs host-dependent initialization when a compilation driver is being initialized.

`SMALL_ARG_MAX`
: Define this macro if the host system has a small limit on the total size of an argument vector. This causes the driver to take more care not to pass unnecessary arguments to subprocesses.

`HOST_LONG_LONG_FORMAT`
: If defined, the string used to indicate an argument of type `long long` to functions like `printf`. The default value is `"ll"`.

In addition, if `configure` generates an incorrect definition of any of the macros in '`auto-host.h`', you can override that definition in a host configuration header. If you need to do this, first see if it is possible to fix `configure`.

16 Makefile Fragments

When you configure GCC using the 'configure' script, it will construct the file 'Makefile' from the template file 'Makefile.in'. When it does this, it can incorporate makefile fragments from the 'config' directory. These are used to set Makefile parameters that are not amenable to being calculated by autoconf. The list of fragments to incorporate is set by 'config.gcc' (and occasionally 'config.build' and 'config.host'); See Section 6.3.2.2 [System Config], page 22.

Fragments are named either 't-*target*' or 'x-*host*', depending on whether they are relevant to configuring GCC to produce code for a particular target, or to configuring GCC to run on a particular host. Here *target* and *host* are mnemonics which usually have some relationship to the canonical system name, but no formal connection.

If these files do not exist, it means nothing needs to be added for a given target or host. Most targets need a few 't-*target*' fragments, but needing 'x-*host*' fragments is rare.

16.1 Target Makefile Fragments

Target makefile fragments can set these Makefile variables.

LIBGCC2_CFLAGS
: Compiler flags to use when compiling 'libgcc2.c'.

LIB2FUNCS_EXTRA
: A list of source file names to be compiled or assembled and inserted into 'libgcc.a'.

Floating Point Emulation
: To have GCC include software floating point libraries in 'libgcc.a' define FPBIT and DPBIT along with a few rules as follows:

```
# We want fine grained libraries, so use the new code
# to build the floating point emulation libraries.
FPBIT = fp-bit.c
DPBIT = dp-bit.c


fp-bit.c: $(srcdir)/config/fp-bit.c
        echo '#define FLOAT' > fp-bit.c
        cat $(srcdir)/config/fp-bit.c >> fp-bit.c

dp-bit.c: $(srcdir)/config/fp-bit.c
        cat $(srcdir)/config/fp-bit.c > dp-bit.c
```

: You may need to provide additional #defines at the beginning of 'fp-bit.c' and 'dp-bit.c' to control target endianness and other options.

CRTSTUFF_T_CFLAGS
: Special flags used when compiling 'crtstuff.c'. See Section 14.19.5 [Initialization], page 366.

CRTSTUFF_T_CFLAGS_S
: Special flags used when compiling 'crtstuff.c' for shared linking. Used if you use 'crtbeginS.o' and 'crtendS.o' in EXTRA-PARTS. See Section 14.19.5 [Initialization], page 366.

`MULTILIB_OPTIONS`

For some targets, invoking GCC in different ways produces objects that can not be linked together. For example, for some targets GCC produces both big and little endian code. For these targets, you must arrange for multiple versions of '`libgcc.a`' to be compiled, one for each set of incompatible options. When GCC invokes the linker, it arranges to link in the right version of '`libgcc.a`', based on the command line options used.

The `MULTILIB_OPTIONS` macro lists the set of options for which special versions of '`libgcc.a`' must be built. Write options that are mutually incompatible side by side, separated by a slash. Write options that may be used together separated by a space. The build procedure will build all combinations of compatible options.

For example, if you set `MULTILIB_OPTIONS` to '`m68000/m68020 msoft-float`', '`Makefile`' will build special versions of '`libgcc.a`' using the following sets of options: '`-m68000`', '`-m68020`', '`-msoft-float`', '`-m68000 -msoft-float`', and '`-m68020 -msoft-float`'.

`MULTILIB_DIRNAMES`

If `MULTILIB_OPTIONS` is used, this variable specifies the directory names that should be used to hold the various libraries. Write one element in `MULTILIB_DIRNAMES` for each element in `MULTILIB_OPTIONS`. If `MULTILIB_DIRNAMES` is not used, the default value will be `MULTILIB_OPTIONS`, with all slashes treated as spaces.

For example, if `MULTILIB_OPTIONS` is set to '`m68000/m68020 msoft-float`', then the default value of `MULTILIB_DIRNAMES` is '`m68000 m68020 msoft-float`'. You may specify a different value if you desire a different set of directory names.

`MULTILIB_MATCHES`

Sometimes the same option may be written in two different ways. If an option is listed in `MULTILIB_OPTIONS`, GCC needs to know about any synonyms. In that case, set `MULTILIB_MATCHES` to a list of items of the form '`option=option`' to describe all relevant synonyms. For example, '`m68000=mc68000 m68020=mc68020`'.

`MULTILIB_EXCEPTIONS`

Sometimes when there are multiple sets of `MULTILIB_OPTIONS` being specified, there are combinations that should not be built. In that case, set `MULTILIB_EXCEPTIONS` to be all of the switch exceptions in shell case syntax that should not be built.

For example the ARM processor cannot execute both hardware floating point instructions and the reduced size THUMB instructions at the same time, so there is no need to build libraries with both of these options enabled. Therefore `MULTILIB_EXCEPTIONS` is set to:

```
*mthumb/*mhard-float*
```

`MULTILIB_EXTRA_OPTS`

Sometimes it is desirable that when building multiple versions of '`libgcc.a`' certain options should always be passed on to the compiler. In that case, set `MULTILIB_EXTRA_OPTS` to be the list of options to be used for all builds. If you set this, you should probably set `CRTSTUFF_T_CFLAGS` to a dash followed by it.

`NATIVE_SYSTEM_HEADER_DIR`
: If the default location for system headers is not '`/usr/include`', you must set this to the directory containing the headers. This value should match the value of the `SYSTEM_INCLUDE_DIR` macro.

`SPECS`
: Unfortunately, setting `MULTILIB_EXTRA_OPTS` is not enough, since it does not affect the build of target libraries, at least not the build of the default multilib. One possible work-around is to use `DRIVER_SELF_SPECS` to bring options from the '`specs`' file as if they had been passed in the compiler driver command line. However, you don't want to be adding these options after the toolchain is installed, so you can instead tweak the '`specs`' file that will be used during the toolchain build, while you still install the original, built-in '`specs`'. The trick is to set `SPECS` to some other filename (say '`specs.install`'), that will then be created out of the built-in specs, and introduce a '`Makefile`' rule to generate the '`specs`' file that's going to be used at build time out of your '`specs.install`'.

16.2 Host Makefile Fragments

The use of '`x-`*`host`*' fragments is discouraged. You should do so only if there is no other mechanism to get the behavior desired. Host fragments should never forcibly override variables set by the configure script, as they may have been adjusted by the user.

Variables provided for host fragments to set include:

`X_CFLAGS`
`X_CPPFLAGS`
: These are extra flags to pass to the C compiler and preprocessor, respectively. They are used both when building GCC, and when compiling things with the just-built GCC.

`XCFLAGS`
: These are extra flags to use when building the compiler. They are not used when compiling '`libgcc.a`'. However, they *are* used when recompiling the compiler with itself in later stages of a bootstrap.

`BOOT_LDFLAGS`
: Flags to be passed to the linker when recompiling the compiler with itself in later stages of a bootstrap. You might need to use this if, for instance, one of the front ends needs more text space than the linker provides by default.

`EXTRA_PROGRAMS`
: A list of additional programs required to use the compiler on this host, which should be compiled with GCC and installed alongside the front ends. If you set this variable, you must also provide rules to build the extra programs.

17 collect2

GCC uses a utility called `collect2` on nearly all systems to arrange to call various initialization functions at start time.

The program `collect2` works by linking the program once and looking through the linker output file for symbols with particular names indicating they are constructor functions. If it finds any, it creates a new temporary '`.c`' file containing a table of them, compiles it, and links the program a second time including that file.

The actual calls to the constructors are carried out by a subroutine called `__main`, which is called (automatically) at the beginning of the body of `main` (provided `main` was compiled with GNU CC). Calling `__main` is necessary, even when compiling C code, to allow linking C and C++ object code together. (If you use '`-nostdlib`', you get an unresolved reference to `__main`, since it's defined in the standard GCC library. Include '`-lgcc`' at the end of your compiler command line to resolve this reference.)

The program `collect2` is installed as `ld` in the directory where the passes of the compiler are installed. When `collect2` needs to find the *real* `ld`, it tries the following file names:

- '`real-ld`' in the directories listed in the compiler's search directories.
- '`real-ld`' in the directories listed in the environment variable `PATH`.
- The file specified in the `REAL_LD_FILE_NAME` configuration macro, if specified.
- '`ld`' in the compiler's search directories, except that `collect2` will not execute itself recursively.
- '`ld`' in `PATH`.

"The compiler's search directories" means all the directories where `gcc` searches for passes of the compiler. This includes directories that you specify with '`-B`'.

Cross-compilers search a little differently:

- '`real-ld`' in the compiler's search directories.
- '*target*`-real-ld`' in `PATH`.
- The file specified in the `REAL_LD_FILE_NAME` configuration macro, if specified.
- '`ld`' in the compiler's search directories.
- '*target*`-ld`' in `PATH`.

`collect2` explicitly avoids running `ld` using the file name under which `collect2` itself was invoked. In fact, it remembers up a list of such names—in case one copy of `collect2` finds another copy (or version) of `collect2` installed as `ld` in a second place in the search path.

`collect2` searches for the utilities `nm` and `strip` using the same algorithm as above for `ld`.

18 Standard Header File Directories

`GCC_INCLUDE_DIR` means the same thing for native and cross. It is where GCC stores its private include files, and also where GCC stores the fixed include files. A cross compiled GCC runs `fixincludes` on the header files in '`$(tooldir)/include`'. (If the cross compilation header files need to be fixed, they must be installed before GCC is built. If the cross compilation header files are already suitable for GCC, nothing special need be done).

`GPLUSPLUS_INCLUDE_DIR` means the same thing for native and cross. It is where `g++` looks first for header files. The C++ library installs only target independent header files in that directory.

`LOCAL_INCLUDE_DIR` is used only by native compilers. GCC doesn't install anything there. It is normally '`/usr/local/include`'. This is where local additions to a packaged system should place header files.

`CROSS_INCLUDE_DIR` is used only by cross compilers. GCC doesn't install anything there.

`TOOL_INCLUDE_DIR` is used for both native and cross compilers. It is the place for other packages to install header files that GCC will use. For a cross-compiler, this is the equivalent of '`/usr/include`'. When you build a cross-compiler, `fixincludes` processes any header files in this directory.

19 Memory Management and Type Information

GCC uses some fairly sophisticated memory management techniques, which involve determining information about GCC's data structures from GCC's source code and using this information to perform garbage collection and implement precompiled headers.

A full C parser would be too complicated for this task, so a limited subset of C is interpreted and special markers are used to determine what parts of the source to look at. All `struct` and `union` declarations that define data structures that are allocated under control of the garbage collector must be marked. All global variables that hold pointers to garbage-collected memory must also be marked. Finally, all global variables that need to be saved and restored by a precompiled header must be marked. (The precompiled header mechanism can only save static variables if they're scalar. Complex data structures must be allocated in garbage-collected memory to be saved in a precompiled header.)

The full format of a marker is

```
GTY (([option] [(param)], [option] [(param)] ...))
```

but in most cases no options are needed. The outer double parentheses are still necessary, though: `GTY(())`. Markers can appear:

- In a structure definition, before the open brace;
- In a global variable declaration, after the keyword `static` or `extern`; and
- In a structure field definition, before the name of the field.

Here are some examples of marking simple data structures and globals.

```
struct tag GTY(())
{
  fields...
};

typedef struct tag GTY(())
{
  fields...
} *typename;

static GTY(()) struct tag *list;   /* points to GC memory */
static GTY(()) int counter;        /* save counter in a PCH */
```

The parser understands simple typedefs such as `typedef struct tag *name;` and `typedef int name;`. These don't need to be marked.

19.1 The Inside of a `GTY(())`

Sometimes the C code is not enough to fully describe the type structure. Extra information can be provided with `GTY` options and additional markers. Some options take a parameter, which may be either a string or a type name, depending on the parameter. If an option takes no parameter, it is acceptable either to omit the parameter entirely, or to provide an empty string as a parameter. For example, `GTY ((skip))` and `GTY ((skip (""))` are equivalent.

When the parameter is a string, often it is a fragment of C code. Four special escapes may be used in these strings, to refer to pieces of the data structure being marked:

`%h` The current structure.

`%1` The structure that immediately contains the current structure.

`%0` The outermost structure that contains the current structure.

`%a` A partial expression of the form `[i1][i2]...` that indexes the array item currently being marked.

For instance, suppose that you have a structure of the form

```
struct A {
  ...
};
struct B {
  struct A foo[12];
};
```

and `b` is a variable of type `struct B`. When marking '`b.foo[11]`', `%h` would expand to '`b.foo[11]`', `%0` and `%1` would both expand to '`b`', and `%a` would expand to '`[11]`'.

As in ordinary C, adjacent strings will be concatenated; this is helpful when you have a complicated expression.

```
GTY ((chain_next ("TREE_CODE (&%h.generic) == INTEGER_TYPE"
                  " ? TYPE_NEXT_VARIANT (&%h.generic)"
                  " : TREE_CHAIN (&%h.generic)")))
```

The available options are:

`length ("`*expression*`")`

There are two places the type machinery will need to be explicitly told the length of an array. The first case is when a structure ends in a variable-length array, like this:

```
struct rtvec_def GTY(()) {
  int num_elem; /* number of elements */
  rtx GTY ((length ("%h.num_elem"))) elem[1];
};
```

In this case, the `length` option is used to override the specified array length (which should usually be `1`). The parameter of the option is a fragment of C code that calculates the length.

The second case is when a structure or a global variable contains a pointer to an array, like this:

```
tree *
  GTY ((length ("%h.regno_pointer_align_length"))) regno_decl;
```

In this case, `regno_decl` has been allocated by writing something like

```
x->regno_decl =
  ggc_alloc (x->regno_pointer_align_length * sizeof (tree));
```

and the `length` provides the length of the field.

This second use of `length` also works on global variables, like:

```
static GTY((length ("reg_base_value_size")))
  rtx *reg_base_value;
```

`skip`

If `skip` is applied to a field, the type machinery will ignore it. This is somewhat dangerous; the only safe use is in a union when one field really isn't ever used.

`desc ("`*expression*`")`

`tag ("`*constant*`")`

`default`

The type machinery needs to be told which field of a `union` is currently active. This is done by giving each field a constant `tag` value, and then specifying a discriminator using `desc`. The value of the expression given by `desc` is compared against each `tag` value, each of which should be different. If no `tag` is matched, the field marked with `default` is used if there is one, otherwise no field in the union will be marked.

In the `desc` option, the "current structure" is the union that it discriminates. Use `%1` to mean the structure containing it. There are no escapes available to the `tag` option, since it is a constant.

For example,

```
struct tree_binding GTY(())
{
  struct tree_common common;
  union tree_binding_u {
    tree GTY ((tag ("0"))) scope;
    struct cp_binding_level * GTY ((tag ("1"))) level;
  } GTY ((desc ("BINDING_HAS_LEVEL_P ((tree)&%0)"))) xscope;
  tree value;
};
```

In this example, the value of BINDING_HAS_LEVEL_P when applied to a `struct tree_binding *` is presumed to be 0 or 1. If 1, the type mechanism will treat the field `level` as being present and if 0, will treat the field `scope` as being present.

`param_is (`*type*`)`
`use_param`

Sometimes it's convenient to define some data structure to work on generic pointers (that is, `PTR`) and then use it with a specific type. `param_is` specifies the real type pointed to, and `use_param` says where in the generic data structure that type should be put.

For instance, to have a `htab_t` that points to trees, one would write the definition of `htab_t` like this:

```
typedef struct GTY(()) {
  ...
  void ** GTY ((use_param, ...)) entries;
  ...
} htab_t;
```

and then declare variables like this:

```
static htab_t GTY ((param_is (union tree_node))) ict;
```

`paramn_is (`*type*`)`
`use_paramn`

In more complicated cases, the data structure might need to work on several different types, which might not necessarily all be pointers. For this, `param1_is` through `param9_is` may be used to specify the real type of a field identified by `use_param1` through `use_param9`.

`use_params`

When a structure contains another structure that is parameterized, there's no need to do anything special, the inner structure inherits the parameters of the outer one. When a structure contains a pointer to a parameterized structure,

the type machinery won't automatically detect this (it could, it just doesn't yet), so it's necessary to tell it that the pointed-to structure should use the same parameters as the outer structure. This is done by marking the pointer with the `use_params` option.

`deletable`

`deletable`, when applied to a global variable, indicates that when garbage collection runs, there's no need to mark anything pointed to by this variable, it can just be set to `NULL` instead. This is used to keep a list of free structures around for re-use.

`if_marked ("`*expression*`")`

Suppose you want some kinds of object to be unique, and so you put them in a hash table. If garbage collection marks the hash table, these objects will never be freed, even if the last other reference to them goes away. GGC has special handling to deal with this: if you use the `if_marked` option on a global hash table, GGC will call the routine whose name is the parameter to the option on each hash table entry. If the routine returns nonzero, the hash table entry will be marked as usual. If the routine returns zero, the hash table entry will be deleted.

The routine `ggc_marked_p` can be used to determine if an element has been marked already; in fact, the usual case is to use `if_marked ("ggc_marked_p")`.

`maybe_undef`

When applied to a field, `maybe_undef` indicates that it's OK if the structure that this fields points to is never defined, so long as this field is always `NULL`. This is used to avoid requiring backends to define certain optional structures. It doesn't work with language frontends.

`nested_ptr (`*type*`, "`*to expression*`", "`*from expression*`")`

The type machinery expects all pointers to point to the start of an object. Sometimes for abstraction purposes it's convenient to have a pointer which points inside an object. So long as it's possible to convert the original object to and from the pointer, such pointers can still be used. *type* is the type of the original object, the *to expression* returns the pointer given the original object, and the *from expression* returns the original object given the pointer. The pointer will be available using the `%h` escape.

`chain_next ("`*expression*`")`

`chain_prev ("`*expression*`")`

It's helpful for the type machinery to know if objects are often chained together in long lists; this lets it generate code that uses less stack space by iterating along the list instead of recursing down it. `chain_next` is an expression for the next item in the list, `chain_prev` is an expression for the previous item. For singly linked lists, use only `chain_next`; for doubly linked lists, use both. The machinery requires that taking the next item of the previous item gives the original item.

`reorder ("`*function name*`")`

Some data structures depend on the relative ordering of pointers. If the precompiled header machinery needs to change that ordering, it

will call the function referenced by the `reorder` option, before changing the pointers in the object that's pointed to by the field the option applies to. The function must take four arguments, with the signature '`void *, void *, gt_pointer_operator, void *`'. The first parameter is a pointer to the structure that contains the object being updated, or the object itself if there is no containing structure. The second parameter is a cookie that should be ignored. The third parameter is a routine that, given a pointer, will update it to its correct new value. The fourth parameter is a cookie that must be passed to the second parameter.

PCH cannot handle data structures that depend on the absolute values of pointers. `reorder` functions can be expensive. When possible, it is better to depend on properties of the data, like an ID number or the hash of a string instead.

`special ("`*name*`")`
The `special` option is used to mark types that have to be dealt with by special case machinery. The parameter is the name of the special case. See '`gengtype.c`' for further details. Avoid adding new special cases unless there is no other alternative.

19.2 Marking Roots for the Garbage Collector

In addition to keeping track of types, the type machinery also locates the global variables (*roots*) that the garbage collector starts at. Roots must be declared using one of the following syntaxes:

- `extern GTY(([`*options*`])) `*type name*`;`
- `static GTY(([`*options*`])) `*type name*`;`

The syntax

- `GTY(([`*options*`])) `*type name*`;`

is *not* accepted. There should be an `extern` declaration of such a variable in a header somewhere—mark that, not the definition. Or, if the variable is only used in one file, make it `static`.

19.3 Source Files Containing Type Information

Whenever you add `GTY` markers to a source file that previously had none, or create a new source file containing `GTY` markers, there are three things you need to do:

1. You need to add the file to the list of source files the type machinery scans. There are four cases:
   a. For a back-end file, this is usually done automatically; if not, you should add it to `target_gtfiles` in the appropriate port's entries in '`config.gcc`'.
   b. For files shared by all front ends, add the filename to the `GTFILES` variable in '`Makefile.in`'.
   c. For files that are part of one front end, add the filename to the `gtfiles` variable defined in the appropriate '`config-lang.in`'. For C, the file is '`c-config-lang.in`'.
   d. For files that are part of some but not all front ends, add the filename to the `gtfiles` variable of *all* the front ends that use it.

2. If the file was a header file, you'll need to check that it's included in the right place to be visible to the generated files. For a back-end header file, this should be done automatically. For a front-end header file, it needs to be included by the same file that includes '`gtype-lang.h`'. For other header files, it needs to be included in '`gtype-desc.c`', which is a generated file, so add it to `ifiles` in `open_base_file` in '`gengtype.c`'.

   For source files that aren't header files, the machinery will generate a header file that should be included in the source file you just changed. The file will be called '`gt-path.h`' where *path* is the pathname relative to the '`gcc`' directory with slashes replaced by `-`, so for example the header file to be included in '`cp/parser.c`' is called '`gt-cp-parser.c`'. The generated header file should be included after everything else in the source file. Don't forget to mention this file as a dependency in the '`Makefile`'!

3. If a new '`gt-path.h`' file is needed, you need to arrange to add a '`Makefile`' rule that will ensure this file can be built. This is done by making it a dependency of `s-gtype`, like this:

   ```
   gt-path.h : s-gtype ; @true
   ```

For language frontends, there is another file that needs to be included somewhere. It will be called '`gtype-lang.h`', where *lang* is the name of the subdirectory the language is contained in. It will need '`Makefile`' rules just like the other generated files.

Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Copyright © 1994 Free Software Foundation, Inc.
Verbatim copying and redistribution of this section is permitted
without royalty; alteration is not permitted.

The GNU Project and GNU/Linux

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. (GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced "guh-NEW".) Variants of the GNU operating system, which use the kernel Linux, are now widely used; though these systems are often referred to as "Linux", they are more accurately called GNU/Linux systems.

For more information, see:

`http://www.gnu.org/`
`http://www.gnu.org/gnu/linux-and-gnu.html`

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘`show w`’ and ‘`show c`’; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and

that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called

an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Contributors to GCC

The GCC project would like to thank its many contributors. Without them the project would not have been nearly as successful as it has been. Any omissions in this list are accidental. Feel free to contact `law@redhat.com` or `gerald@pfeifer.com` if you have been left out or some of your contributions are not listed. Please keep this list in alphabetical order.

- Analog Devices helped implement the support for complex data types and iterators.
- John David Anglin for threading-related fixes and improvements to libstdc++-v3, and the HP-UX port.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Abramo and Roberto Bagnara for the SysV68 Motorola 3300 Delta Series port.
- Alasdair Baird for various bug fixes.
- Giovanni Bajo for analyzing lots of complicated C++ problem reports.
- Peter Barada for his work to improve code generation for new ColdFire cores.
- Gerald Baumgartner added the signature extension to the C++ front end.
- Godmar Back for his Java improvements and encouragement.
- Scott Bambrough for help porting the Java compiler.
- Wolfgang Bangerth for processing tons of bug reports.
- Jon Beniston for his Microsoft Windows port of Java.
- Daniel Berlin for better DWARF2 support, faster/better optimizations, improved alias analysis, plus migrating GCC to Bugzilla.
- Geoff Berry for his Java object serialization work and various patches.
- Eric Blake for helping to make GCJ and libgcj conform to the specifications.
- Janne Blomqvist for contributions to gfortran.
- Segher Boessenkool for various fixes.
- Hans-J. Boehm for his garbage collector, IA-64 libffi port, and other Java work.
- Neil Booth for work on cpplib, lang hooks, debug hooks and other miscellaneous clean-ups.
- Steven Bosscher for integrating the gfortran front end into GCC and for contributing to the tree-ssa branch.
- Eric Botcazou for fixing middle- and backend bugs left and right.
- Per Bothner for his direction via the steering committee and various improvements to the infrastructure for supporting new languages. Chill front end implementation. Initial implementations of cpplib, fix-header, config.guess, libio, and past C++ library (libg++) maintainer. Dreaming up, designing and implementing much of GCJ.
- Devon Bowen helped port GCC to the Tahoe.
- Don Bowman for mips-vxworks contributions.
- Dave Brolley for work on cpplib and Chill.
- Paul Brook for work on the ARM architecture and maintaining gfortran.

- Robert Brown implemented the support for Encore 32000 systems.
- Christian Bruel for improvements to local store elimination.
- Herman A.J. ten Brugge for various fixes.
- Joerg Brunsmann for Java compiler hacking and help with the GCJ FAQ.
- Joe Buck for his direction via the steering committee.
- Craig Burley for leadership of the G77 Fortran effort.
- Stephan Buys for contributing Doxygen notes for libstdc++.
- Paolo Carlini for libstdc++ work: lots of efficiency improvements to the C++ strings, streambufs and formatted I/O, hard detective work on the frustrating localization issues, and keeping up with the problem reports.
- John Carr for his alias work, SPARC hacking, infrastructure improvements, previous contributions to the steering committee, loop optimizations, etc.
- Stephane Carrez for 68HC11 and 68HC12 ports.
- Steve Chamberlain for support for the Renesas SH and H8 processors and the PicoJava processor, and for GCJ config fixes.
- Glenn Chambers for help with the GCJ FAQ.
- John-Marc Chandonia for various libgcj patches.
- Scott Christley for his Objective-C contributions.
- Eric Christopher for his Java porting help and clean-ups.
- Branko Cibej for more warning contributions.
- The GNU Classpath project for all of their merged runtime code.
- Nick Clifton for arm, mcore, fr30, v850, m32r work, '`--help`', and other random hacking.
- Michael Cook for libstdc++ cleanup patches to reduce warnings.
- R. Kelley Cook for making GCC buildable from a read-only directory as well as other miscellaneous build process and documentation clean-ups.
- Ralf Corsepius for SH testing and minor bugfixing.
- Stan Cox for care and feeding of the x86 port and lots of behind the scenes hacking.
- Alex Crain provided changes for the 3b1.
- Ian Dall for major improvements to the NS32k port.
- Paul Dale for his work to add uClinux platform support to the m68k backend.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Russell Davidson for fstream and stringstream fixes in libstdc++.
- Bud Davis for work on the G77 and gfortran compilers.
- Mo DeJong for GCJ and libgcj bug fixes.
- DJ Delorie for the DJGPP port, build and libiberty maintenance, various bug fixes, and the M32C port.
- Arnaud Desitter for helping to debug gfortran.

- Gabriel Dos Reis for contributions to G++, contributions and maintenance of GCC diagnostics infrastructure, libstdc++-v3, including `valarray<>`, `complex<>`, maintaining the numerics library (including that pesky `<limits>` :-) and keeping up-to-date anything to do with numbers.
- Ulrich Drepper for his work on glibc, testing of GCC using glibc, ISO C99 support, CFG dumping support, etc., plus support of the C++ runtime libraries including for all kinds of C interface issues, contributing and maintaining `complex<>`, sanity checking and disbursement, configuration architecture, libio maintenance, and early math work.
- Zdenek Dvorak for a new loop unroller and various fixes.
- Richard Earnshaw for his ongoing work with the ARM.
- David Edelsohn for his direction via the steering committee, ongoing work with the RS6000/PowerPC port, help cleaning up Haifa loop changes, doing the entire AIX port of libstdc++ with his bare hands, and for ensuring GCC properly keeps working on AIX.
- Kevin Ediger for the floating point formatting of num_put::do_put in libstdc++.
- Phil Edwards for libstdc++ work including configuration hackery, documentation maintainer, chief breaker of the web pages, the occasional iostream bug fix, and work on shared library symbol versioning.
- Paul Eggert for random hacking all over GCC.
- Mark Elbrecht for various DJGPP improvements, and for libstdc++ configuration support for locales and fstream-related fixes.
- Vadim Egorov for libstdc++ fixes in strings, streambufs, and iostreams.
- Christian Ehrhardt for dealing with bug reports.
- Ben Elliston for his work to move the Objective-C runtime into its own subdirectory and for his work on autoconf.
- Marc Espie for OpenBSD support.
- Doug Evans for much of the global optimization framework, arc, m32r, and SPARC work.
- Christopher Faylor for his work on the Cygwin port and for caring and feeding the gcc.gnu.org box and saving its users tons of spam.
- Fred Fish for BeOS support and Ada fixes.
- Ivan Fontes Garcia for the Portuguese translation of the GCJ FAQ.
- Peter Gerwinski for various bug fixes and the Pascal front end.
- Kaveh R. Ghazi for his direction via the steering committee, amazing work to make '`-W -Wall -W* -Werror`' useful, and continuously testing GCC on a plethora of platforms. Kaveh extends his gratitude to the CAIP Center at Rutgers University for providing him with computing resources to work on Free Software since the late 1980s.
- John Gilmore for a donation to the FSF earmarked improving GNU Java.
- Judy Goldberg for c++ contributions.
- Torbjorn Granlund for various fixes and the c-torture testsuite, multiply- and divide-by-constant optimization, improved long long support, improved leaf function register allocation, and his direction via the steering committee.

- Anthony Green for his '`-Os`' contributions and Java front end work.
- Stu Grossman for gdb hacking, allowing GCJ developers to debug Java code.
- Michael K. Gschwind contributed the port to the PDP-11.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for Dwarf symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Mostafa Hagog for Swing Modulo Scheduling (SMS) and post reload GCSE.
- Bruno Haible for improvements in the runtime overhead for EH, new warnings and assorted bug fixes.
- Andrew Haley for his amazing Java compiler and library efforts.
- Chris Hanson assisted in making GCC work on HP-UX for the 9000 series 300.
- Michael Hayes for various thankless work he's done trying to get the c30/c40 ports functional. Lots of loop and unroll improvements and fixes.
- Dara Hazeghi for wading through myriads of target-specific bug reports.
- Kate Hedstrom for staking the G77 folks with an initial testsuite.
- Richard Henderson for his ongoing SPARC, alpha, ia32, and ia64 work, loop opts, and generally fixing lots of old problems we've ignored for years, flow rewrite and lots of further stuff, including reviewing tons of patches.
- Aldy Hernandez for working on the PowerPC port, SIMD support, and various fixes.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Kazu Hirata for caring and feeding the Renesas H8/300 port and various fixes.
- Katherine Holcomb for work on gfortran.
- Manfred Hollstein for his ongoing work to keep the m88k alive, lots of testing and bug fixing, particularly of GCC configury code.
- Steve Holmgren for MachTen patches.
- Jan Hubicka for his x86 port improvements.
- Falk Hueffner for working on C and optimization bug reports.
- Bernardo Innocenti for his m68k work, including merging of ColdFire improvements and uClinux support.
- Christian Iseli for various bug fixes.
- Kamil Iskra for general m68k hacking.
- Lee Iverson for random fixes and MIPS testing.
- Andreas Jaeger for testing and benchmarking of GCC and various bug fixes.
- Jakub Jelinek for his SPARC work and sibling call optimizations as well as lots of bug fixes and test cases, and for improving the Java build system.
- Janis Johnson for ia64 testing and fixes, her quality improvement sidetracks, and web page maintenance.
- Kean Johnston for SCO OpenServer support and various fixes.
- Tim Josling for the sample language treelang based originally on Richard Kenner's "toy" language.

- Nicolai Josuttis for additional libstdc++ documentation.
- Klaus Kaempf for his ongoing work to make alpha-vms a viable target.
- Steven G. Kargl for work on gfortran.
- David Kashtan of SRI adapted GCC to VMS.
- Ryszard Kabatek for many, many libstdc++ bug fixes and optimizations of strings, especially member functions, and for auto_ptr fixes.
- Geoffrey Keating for his ongoing work to make the PPC work for GNU/Linux and his automatic regression tester.
- Brendan Kehoe for his ongoing work with G++ and for a lot of early work in just about every part of libstdc++.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination and delay slot scheduling. Richard Kenner was also the head maintainer of GCC for several years.
- Mumit Khan for various contributions to the Cygwin and Mingw32 ports and maintaining binary releases for Microsoft Windows hosts, and for massive libstdc++ porting work to Cygwin/Mingw32.
- Robin Kirkham for cpu32 support.
- Mark Klein for PA improvements.
- Thomas Koenig for various bug fixes.
- Bruce Korb for the new and improved fixincludes code.
- Benjamin Kosnik for his G++ work and for leading the libstdc++-v3 effort.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Asher Langton and Mike Kumbera for contributing Cray pointer support to gfortran, and for other gfortran improvements.
- Jeff Law for his direction via the steering committee, coordinating the entire egcs project and GCC 2.95, rolling out snapshots and releases, handling merges from GCC2, reviewing tons of patches that might have fallen through the cracks else, and random but extensive hacking.
- Marc Lehmann for his direction via the steering committee and helping with analysis and improvements of x86 performance.
- Victor Leikehman for work on gfortran.
- Ted Lemon wrote parts of the RTL reader and printer.
- Kriang Lerdsuwanakij for C++ improvements including template as template parameter support, and many C++ fixes.
- Warren Levy for tremendous work on libgcj (Java Runtime Library) and random work on the Java front end.
- Alain Lichnewsky ported GCC to the MIPS CPU.

- Oskar Liljeblad for hacking on AWT and his many Java bug reports and patches.
- Robert Lipe for OpenServer support, new testsuites, testing, etc.
- Weiwen Liu for testing and various bug fixes.
- Dave Love for his ongoing work with the Fortran front end and runtime libraries.
- Martin von Löwis for internal consistency checking infrastructure, various C++ improvements including namespace support, and tons of assistance with libstdc++/compiler merges.
- H.J. Lu for his previous contributions to the steering committee, many x86 bug reports, prototype patches, and keeping the GNU/Linux ports working.
- Greg McGary for random fixes and (someday) bounded pointers.
- Andrew MacLeod for his ongoing work in building a real EH system, various code generation improvements, work on the global optimizer, etc.
- Vladimir Makarov for hacking some ugly i960 problems, PowerPC hacking improvements to compile-time performance, overall knowledge and direction in the area of instruction scheduling, and design and implementation of the automaton based instruction scheduler.
- Bob Manson for his behind the scenes work on dejagnu.
- Philip Martin for lots of libstdc++ string and vector iterator fixes and improvements, and string clean up and testsuites.
- All of the Mauve project contributors, for Java test code.
- Bryce McKinlay for numerous GCJ and libgcj fixes and improvements.
- Adam Megacz for his work on the Microsoft Windows port of GCJ.
- Michael Meissner for LRS framework, ia32, m32r, v850, m88k, MIPS, powerpc, haifa, ECOFF debug support, and other assorted hacking.
- Jason Merrill for his direction via the steering committee and leading the G++ effort.
- David Miller for his direction via the steering committee, lots of SPARC work, improvements in jump.c and interfacing with the Linux kernel developers.
- Gary Miller ported GCC to Charles River Data Systems machines.
- Alfred Minarik for libstdc++ string and ios bug fixes, and turning the entire libstdc++ testsuite namespace-compatible.
- Mark Mitchell for his direction via the steering committee, mountains of C++ work, load/store hoisting out of loops, alias analysis improvements, ISO C `restrict` support, and serving as release manager for GCC 3.x.
- Alan Modra for various GNU/Linux bits and testing.
- Toon Moene for his direction via the steering committee, Fortran maintenance, and his ongoing work to make us make Fortran run fast.
- Jason Molenda for major help in the care and feeding of all the services on the gcc.gnu.org (formerly egcs.cygnus.com) machine—mail, web services, ftp services, etc etc. Doing all this work on scrap paper and the backs of envelopes would have been... difficult.
- Catherine Moore for fixing various ugly problems we have sent her way, including the haifa bug which was killing the Alpha & PowerPC Linux kernels.

- Mike Moreton for his various Java patches.
- David Mosberger-Tang for various Alpha improvements, and for the initial IA-64 port.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits and for ISO C99 support.
- Bill Moyer for his behind the scenes work on various issues.
- Philippe De Muyter for his work on the m68k port.
- Joseph S. Myers for his work on the PDP-11 port, format checking and ISO C99 support, and continuous emphasis on (and contributions to) documentation.
- Nathan Myers for his work on libstdc++-v3: architecture and authorship through the first three snapshots, including implementation of locale infrastructure, string, shadow C headers, and the initial project documentation (DESIGN, CHECKLIST, and so forth). Later, more work on MT-safe string and shadow headers.
- Felix Natter for documentation on porting libstdc++.
- Nathanael Nerode for cleaning up the configuration/build process.
- NeXT, Inc. donated the front end that supports the Objective-C language.
- Hans-Peter Nilsson for the CRIS and MMIX ports, improvements to the search engine setup, various documentation fixes and other small fixes.
- Geoff Noer for his work on getting cygwin native builds working.
- Diego Novillo for his SPEC performance tracking web pages and assorted fixes in the middle end and various back ends.
- David O'Brien for the FreeBSD/alpha, FreeBSD/AMD x86-64, FreeBSD/ARM, FreeBSD/PowerPC, and FreeBSD/SPARC64 ports and related infrastructure improvements.
- Alexandre Oliva for various build infrastructure improvements, scripts and amazing testing work, including keeping libtool issues sane and happy.
- Stefan Olsson for work on mt_alloc.
- Melissa O'Neill for various NeXT fixes.
- Rainer Orth for random MIPS work, including improvements to GCC's o32 ABI support, improvements to dejagnu's MIPS support, Java configuration clean-ups and porting work, etc.
- Hartmut Penner for work on the s390 port.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Alexandre Petit-Bianco for implementing much of the Java compiler and continued Java maintainership.
- Matthias Pfaller for major improvements to the NS32k port.
- Gerald Pfeifer for his direction via the steering committee, pointing out lots of problems we need to solve, maintenance of the web pages, and taking care of documentation maintenance in general.
- Andrew Pinski for processing bug reports by the dozen.
- Ovidiu Predescu for his work on the Objective-C front end and runtime libraries.

- Jerry Quinn for major performance improvements in C++ formatted I/O.
- Ken Raeburn for various improvements to checker, MIPS ports and various cleanups in the compiler.
- Rolf W. Rasmussen for hacking on AWT.
- David Reese of Sun Microsystems contributed to the Solaris on PowerPC port.
- Volker Reichelt for keeping up with the problem reports.
- Joern Rennecke for maintaining the sh port, loop, regmove & reload hacking.
- Loren J. Rittle for improvements to libstdc++-v3 including the FreeBSD port, threading fixes, thread-related configury changes, critical threading documentation, and solutions to really tricky I/O problems, as well as keeping GCC properly working on FreeBSD and continuous testing.
- Craig Rodrigues for processing tons of bug reports.
- Ola Rönnerup for work on mt_alloc.
- Gavin Romig-Koch for lots of behind the scenes MIPS work.
- David Ronis inspired and encouraged Craig to rewrite the G77 documentation in texinfo format by contributing a first pass at a translation of the old '`g77-0.5.16/f/DOC`' file.
- Ken Rose for fixes to GCC's delay slot filling code.
- Paul Rubin wrote most of the preprocessor.
- Pétur Runólfsson for major performance improvements in C++ formatted I/O and large file support in C++ filebuf.
- Chip Salzenberg for libstdc++ patches and improvements to locales, traits, Makefiles, libio, libtool hackery, and "long long" support.
- Juha Sarlin for improvements to the H8 code generator.
- Greg Satz assisted in making GCC work on HP-UX for the 9000 series 300.
- Roger Sayle for improvements to constant folding and GCC's RTL optimizers as well as for fixing numerous bugs.
- Bradley Schatz for his work on the GCJ FAQ.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- William Schelter did most of the work on the Intel 80386 support.
- Tobias Schlüter for work on gfortran.
- Bernd Schmidt for various code generation improvements and major work in the reload pass as well a serving as release manager for GCC 2.95.3.
- Peter Schmid for constant testing of libstdc++—especially application testing, going above and beyond what was requested for the release criteria—and libstdc++ header file tweaks.
- Jason Schroeder for jcf-dump patches.
- Andreas Schwab for his work on the m68k port.
- Lars Segerlund for work on gfortran.
- Joel Sherrill for his direction via the steering committee, RTEMS contributions and RTEMS testing.
- Nathan Sidwell for many C++ fixes/improvements.

- Jeffrey Siegal for helping RMS with the original design of GCC, some code which handles the parse tree and RTL data structures, constant folding and help with the original VAX & m68k ports.
- Kenny Simpson for prompting libstdc++ fixes due to defect reports from the LWG (thereby keeping GCC in line with updates from the ISO).
- Franz Sirl for his ongoing work with making the PPC port stable for GNU/Linux.
- Andrey Slepuhin for assorted AIX hacking.
- Christopher Smith did the port for Convex machines.
- Danny Smith for his major efforts on the Mingw (and Cygwin) ports.
- Randy Smith finished the Sun FPA support.
- Scott Snyder for queue, iterator, istream, and string fixes and libstdc++ testsuite entries. Also for providing the patch to G77 to add rudimentary support for `INTEGER*1`, `INTEGER*2`, and `LOGICAL*1`.
- Brad Spencer for contributions to the GLIBCPP_FORCE_NEW technique.
- Richard Stallman, for writing the original GCC and launching the GNU project.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Nigel Stephens for various mips16 related fixes/improvements.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Graham Stott for various infrastructure improvements.
- John Stracke for his Java HTTP protocol fixes.
- Mike Stump for his Elxsi port, G++ contributions over the years and more recently his vxworks contributions
- Jeff Sturm for Java porting help, bug fixes, and encouragement.
- Shigeya Suzuki for this fixes for the bsdi platforms.
- Ian Lance Taylor for his mips16 work, general configury hacking, fixincludes, etc.
- Holger Teutsch provided the support for the Clipper CPU.
- Gary Thomas for his ongoing work to make the PPC work for GNU/Linux.
- Philipp Thomas for random bug fixes throughout the compiler
- Jason Thorpe for thread support in libstdc++ on NetBSD.
- Kresten Krab Thorup wrote the run time support for the Objective-C language and the fantastic Java bytecode interpreter.
- Michael Tiemann for random bug fixes, the first instruction scheduler, initial C++ support, function integration, NS32k, SPARC and M88k machine description work, delay slot scheduling.
- Andreas Tobler for his work porting libgcj to Darwin.
- Teemu Torma for thread safe exception handling support.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the VAX machine description.
- Tom Tromey for internationalization support and for his many Java contributions and libgcj maintainership.

- Lassi Tuura for improvements to config.guess to determine HP processor types.
- Petter Urkedal for libstdc++ CXXFLAGS, math, and algorithms fixes.
- Andy Vaught for the design and initial implementation of the gfortran front end.
- Brent Verner for work with the libstdc++ cshadow files and their associated configure steps.
- Todd Vierling for contributions for NetBSD ports.
- Jonathan Wakely for contributing libstdc++ Doxygen notes and XHTML guidance.
- Dean Wakerley for converting the install documentation from HTML to texinfo in time for GCC 3.0.
- Krister Walfridsson for random bug fixes.
- Feng Wang for contributions to gfortran.
- Stephen M. Webb for time and effort on making libstdc++ shadow files work with the tricky Solaris 8+ headers, and for pushing the build-time header tree.
- John Wehle for various improvements for the x86 code generator, related infrastructure improvements to help x86 code generation, value range propagation and other work, WE32k port.
- Ulrich Weigand for work on the s390 port.
- Zack Weinberg for major work on cpplib and various other bug fixes.
- Matt Welsh for help with Linux Threads support in GCJ.
- Urban Widmark for help fixing java.io.
- Mark Wielaard for new Java library code and his work integrating with Classpath.
- Dale Wiles helped port GCC to the Tahoe.
- Bob Wilson from Tensilica, Inc. for the Xtensa port.
- Jim Wilson for his direction via the steering committee, tackling hard problems in various places that nobody else wanted to work on, strength reduction and other loop optimizations.
- Paul Woegerer and Tal Agmon for the CRX port.
- Carlo Wood for various fixes.
- Tom Wood for work on the m88k port.
- Canqun Yang for work on gfortran.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- Kevin Zachmann helped port GCC to the Tahoe.
- Ayal Zaks for Swing Modulo Scheduling (SMS).
- Xiaoqiang Zhang for work on gfortran.
- Gilles Zunino for help porting Java to Irix.

The following people are recognized for their contributions to GNAT, the Ada front end of GCC:

- Bernard Banner
- Romain Berrendonner

- Geert Bosch
- Emmanuel Briot
- Joel Brobecker
- Ben Brosgol
- Vincent Celier
- Arnaud Charlet
- Chien Chieng
- Cyrille Comar
- Cyrille Crozes
- Robert Dewar
- Gary Dismukes
- Robert Duff
- Ed Falis
- Ramon Fernandez
- Sam Figueroa
- Vasiliy Fofanov
- Michael Friess
- Franco Gasperoni
- Ted Giering
- Matthew Gingell
- Laurent Guerby
- Jerome Guitton
- Olivier Hainque
- Jerome Hugues
- Hristian Kirtchev
- Jerome Lambourg
- Bruno Leclerc
- Albert Lee
- Sean McNeil
- Javier Miranda
- Laurent Nana
- Pascal Obry
- Dong-Ik Oh
- Laurent Pautet
- Brett Porter
- Thomas Quinot
- Nicolas Roche
- Pat Rogers
- Jose Ruiz

- Douglas Rupp
- Sergey Rybin
- Gail Schenker
- Ed Schonberg
- Nicolas Setton
- Samuel Tardieu

The following people are recognized for their contributions of new features, bug reports, testing and integration of classpath/libgcj for GCC version 4.1:

- Lillian Angel for `JTree` implementation and lots Free Swing additions and bugfixes.
- Wolfgang Baer for `GapContent` bugfixes.
- Anthony Balkissoon for `JList`, Free Swing 1.5 updates and mouse event fixes, lots of Free Swing work including `JTable` editing.
- Stuart Ballard for RMI constant fixes.
- Goffredo Baroncelli for `HTTPURLConnection` fixes.
- Gary Benson for `MessageFormat` fixes.
- Daniel Bonniot for `Serialization` fixes.
- Chris Burdess for lots of gnu.xml and http protocol fixes, `StAX` and `DOM` `xml:id` support.
- Ka-Hing Cheung for `TreePath` and `TreeSelection` fixes.
- Archie Cobbs for build fixes, VM interface updates, `URLClassLoader` updates.
- Kelley Cook for build fixes.
- Martin Cordova for Suggestions for better `SocketTimeoutException`.
- David Daney for `BitSet` bugfixes, `HttpURLConnection` rewrite and improvements.
- Thomas Fitzsimmons for lots of upgrades to the gtk+ AWT and Cairo 2D support. Lots of imageio framework additions, lots of AWT and Free Swing bugfixes.
- Jeroen Frijters for `ClassLoader` and nio cleanups, serialization fixes, better `Proxy` support, bugfixes and IKVM integration.
- Santiago Gala for `AccessControlContext` fixes.
- Nicolas Geoffray for `VMClassLoader` and `AccessController` improvements.
- David Gilbert for `basic` and `metal` icon and plaf support and lots of documenting, Lots of Free Swing and metal theme additions. `MetalIconFactory` implementation.
- Anthony Green for `MIDI` framework, `ALSA` and `DSSI` providers.
- Andrew Haley for `Serialization` and `URLClassLoader` fixes, gcj build speedups.
- Kim Ho for `JFileChooser` implementation.
- Andrew John Hughes for `Locale` and net fixes, URI RFC2986 updates, `Serialization` fixes, `Properties` XML support and generic branch work, VMIntegration guide update.
- Bastiaan Huisman for `TimeZone` bugfixing.
- Andreas Jaeger for mprec updates.
- Paul Jenner for better '`-Werror`' support.
- Ito Kazumitsu for `NetworkInterface` implementation and updates.

- Roman Kennke for `BoxLayout`, `GrayFilter` and `SplitPane`, plus bugfixes all over. Lots of Free Swing work including styled text.
- Simon Kitching for `String` cleanups and optimization suggestions.
- Michael Koch for configuration fixes, `Locale` updates, bug and build fixes.
- Guilhem Lavaux for configuration, thread and channel fixes and Kaffe integration. JCL native `Pointer` updates. Logger bugfixes.
- David Lichteblau for JCL support library global/local reference cleanups.
- Aaron Luchko for JDWP updates and documentation fixes.
- Ziga Mahkovec for `Graphics2D` upgraded to Cairo 0.5 and new regex features.
- Sven de Marothy for BMP imageio support, CSS and `TextLayout` fixes. `GtkImage` rewrite, 2D, awt, free swing and date/time fixes and implementing the Qt4 peers.
- Casey Marshall for crypto algorithm fixes, `FileChannel` lock, `SystemLogger` and `FileHandler` rotate implementations, NIO `FileChannel.map` support, security and policy updates.
- Bryce McKinlay for RMI work.
- Audrius Meskauskas for lots of Free Corba, RMI and HTML work plus testing and documenting.
- Kalle Olavi Niemitalo for build fixes.
- Rainer Orth for build fixes.
- Andrew Overholt for `File` locking fixes.
- Ingo Proetel for `Image`, `Logger` and `URLClassLoader` updates.
- Olga Rodimina for `MenuSelectionManager` implemenation.
- Jan Roehrich for `BasicTreeUI` and `JTree` fixes.
- Julian Scheid for documentation updates and gjdoc support.
- Christian Schlichtherle for zip fixes and cleanups.
- Robert Schuster for documentation updates and beans fixes, `TreeNode` enumerations and `ActionCommand` and various fixes, XML and URL, AWT and Free Swing bugfixes.
- Keith Seitz for lots of JDWP work.
- Christian Thalinger for 64-bit cleanups, Configuration and VM interface fixes and `CACAO` integration, `fdlibm` updates.
- Gael Thomas for `VMClassLoader` boot packages support sugestions.
- Andreas Tobler for Darwin and Solaris testing and fixing, `Qt4` support for Darwin/OS X, `Graphics2D` support, `gtk+` updates.
- Dalibor Topic for better `DEBUG` support, build cleanups and Kaffe integration. `Qt4` build infrastructure, `SHA1PRNG` and `GdkPixbugDecoder` updates.
- Tom Tromey for Eclipse integration, generics work, lots of bugfixes and gcj integration including coordinating The Big Merge.
- Mark Wielaard for bugfixes, packaging and release management, `Clipboard` implementation, system call interrupts and network timeouts and `GdkPixpufDecoder` fixes.

In addition to the above, all of which also contributed time and energy in testing GCC, we would like to thank the following for their contributions to testing:

- Michael Abd-El-Malek
- Thomas Arend
- Bonzo Armstrong
- Steven Ashe
- Chris Baldwin
- David Billinghurst
- Jim Blandy
- Stephane Bortzmeyer
- Horst von Brand
- Frank Braun
- Rodney Brown
- Sidney Cadot
- Bradford Castalia
- Jonathan Corbet
- Ralph Doncaster
- Richard Emberson
- Levente Farkas
- Graham Fawcett
- Mark Fernyhough
- Robert A. French
- Jörgen Freyh
- Mark K. Gardner
- Charles-Antoine Gauthier
- Yung Shing Gene
- David Gilbert
- Simon Gornall
- Fred Gray
- John Griffin
- Patrik Hagglund
- Phil Hargett
- Amancio Hasty
- Takafumi Hayashi
- Bryan W. Headley
- Kevin B. Hendricks
- Joep Jansen
- Christian Joensson
- Michel Kern
- David Kidd
- Tobias Kuipers

- Anand Krishnaswamy
- A. O. V. Le Blanc
- llewelly
- Damon Love
- Brad Lucier
- Matthias Klose
- Martin Knoblauch
- Rick Lutowski
- Jesse Macnish
- Stefan Morrell
- Anon A. Mous
- Matthias Mueller
- Pekka Nikander
- Rick Niles
- Jon Olson
- Magnus Persson
- Chris Pollard
- Richard Polton
- Derk Reefman
- David Rees
- Paul Reilly
- Tom Reilly
- Torsten Rueger
- Danny Sadinoff
- Marc Schifer
- Erik Schnetter
- Wayne K. Schroll
- David Schuler
- Vin Shelton
- Tim Souder
- Adam Sulmicki
- Bill Thorson
- George Talbot
- Pedro A. M. Vazquez
- Gregory Warnes
- Ian Watson
- David E. Young
- And many others

And finally we'd like to thank everyone who uses the compiler, submits bug reports and generally reminds us why we're doing this work in the first place.

Option Index

GCC's command line options are indexed here without any initial '`-`' or '`--`'. Where an option has both positive and negative forms (such as '`-f`*option*' and '`-fno-`*option*'), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

`msoft-float` 12

Concept Index

!

'!' in constraint 196

#

'#' in constraint 197
in template 185
#pragma 396, 397

%

'%' in constraint 197
% in GTY option 417
'%' in template 185

&

'&' in constraint 197

(

(nil) 122

*

* 405
'*' in constraint 197
* in template 186
*TARGET_GET_PCH_VALIDITY 389

+

'+' in constraint 197

/

'/c' in RTL dump 130
'/f' in RTL dump 130
'/i' in RTL dump 131
'/j' in RTL dump 131
'/s' in RTL dump 131
'/u' in RTL dump 131
'/v' in RTL dump 132

<

'<' in constraint 192

=

'=' in constraint 197

>

'>' in constraint 192

?

'?' in constraint 196

_

__absvdi2 11
__absvsi2 11
__adddf3 12
__addsf3 12
__addtf3 12
__addvdi3 11
__addvsi3 11
__addxf3 12
__ashldi3 9
__ashlsi3 9
__ashlti3 9
__ashrdi3 9
__ashrsi3 9
__ashrti3 9
__builtin_args_info 329
__builtin_classify_type 329
__builtin_next_arg 329
__builtin_saveregs 328
__clear_cache 16
__clzdi2 11
__clzsi2 11
__clzti2 11
__cmpdf2 14
__cmpdi2 10
__cmpsf2 14
__cmptf2 14
__cmpti2 10
__CTOR_LIST__ 366
__ctzdi2 11
__ctzsi2 11
__ctzti2 11
__divdf3 12
__divdi3 9
__divsf3 12
__divsi3 9
__divtf3 12
__divti3 9
__divxf3 12
__DTOR_LIST__ 366
__eqdf2 15
__eqsf2 15
__eqtf2 15
__extenddftf2 13
__extenddfxf2 13
__extendsfdf2 13

__extendsftf2 13
__extendsfxf2 13
__ffsdi2 11
__ffsti2 11
__fixdfdi 13
__fixdfsi 13
__fixdfti 13
__fixsfdi 13
__fixsfsi 13
__fixsfti 13
__fixtfdi 13
__fixtfsi 13
__fixtfti 13
__fixunsdfdi 13
__fixunsdfsi 13
__fixunsdfti 14
__fixunssfdi 13
__fixunssfsi 13
__fixunssfti 14
__fixunstfdi 13
__fixunstfsi 13
__fixunstfti 14
__fixunsxfdi 13
__fixunsxfsi 13
__fixunsxfti 14
__fixxfdi 13
__fixxfsi 13
__fixxfti 13
__floatdidf 14
__floatdisf 14
__floatditf 14
__floatdixf 14
__floatsidf 14
__floatsisf 14
__floatsitf 14
__floatsixf 14
__floattidf 14
__floattisf 14
__floattitf 14
__floattixf 14
__gedf2 15
__gesf2 15
__getf2 15
__gtdf2 15
__gtsf2 15
__gttf2 15
__ledf2 15
__lesf2 15
__letf2 15
__lshrdi3 9
__lshrsi3 9
__lshrti3 9
__ltdf2 15
__ltsf2 15
__lttf2 15
__main 413
__moddi3 10
__modsi3 10
__modti3 10
__muldf3 12
__muldi3 10
__mulsf3 12
__mulsi3 10
__multf3 12
__multi3 10
__mulvdi3 11
__mulvsi3 11
__mulxf3 12
__nedf2 15
__negdf2 12
__negdi2 10
__negsf2 12
__negtf2 12
__negti2 10
__negvdi2 11
__negvsi2 11
__negxf2 12
__nesf2 15
__netf2 15
__paritydi2 11
__paritysi2 11
__parityti2 11
__popcountdi2 12
__popcountsi2 12
__popcountti2 12
__subdf3 12
__subsf3 12
__subtf3 12
__subvdi3 11
__subvsi3 11
__subxf3 12
__truncdfsf2 13
__trunctfdf2 13
__trunctfsf2 13
__truncxfdf2 13
__truncxfsf2 13
__ucmpdi2 10
__ucmpti2 10
__udivdi3 10
__udivmoddi3 10
__udivsi3 10
__udivti3 10
__umoddi3 10
__umodsi3 10
__umodti3 10
__unorddf2 14
__unordsf2 14
__unordtf2 14

\

\ 185

0

‘0’ in constraint 193

A

abort ... 5
abs ... 144
abs and attributes ... 250
ABS_EXPR ... 84
absence_set ... 259
abs*m*2 instruction pattern ... 216
absolute value ... 144
access to operands ... 124
access to special operands ... 125
accessors ... 124
ACCUMULATE_OUTGOING_ARGS ... 314
ACCUMULATE_OUTGOING_ARGS and stack frames ... 325
ADA_LONG_TYPE_SIZE ... 287
ADDITIONAL_REGISTER_NAMES ... 370
add*m*3 instruction pattern ... 214
add*mode*cc instruction pattern ... 221
addr_diff_vec ... 154
addr_diff_vec, length of ... 254
ADDR_EXPR ... 84
addr_vec ... 154
addr_vec, length of ... 254
address constraints ... 193
address_operand ... 189, 193
addressing modes ... 334
addressof ... 142
ADJUST_FIELD_ALIGN ... 281
ADJUST_INSN_LENGTH ... 254
AGGR_INIT_EXPR ... 84
aggregates as return values ... 322
alias ... 116
ALL_COP_ADDITIONAL_REGISTER_NAMES ... 389
ALL_REGS ... 296
allocate_stack instruction pattern ... 226
alternate entry points ... 158
and ... 144
and and attributes ... 250
and, canonicalization of ... 237
and*m*3 instruction pattern ... 214
annotations ... 106
APPLY_RESULT_SIZE ... 321
ARG_POINTER_CFA_OFFSET ... 306
ARG_POINTER_REGNUM ... 311
ARG_POINTER_REGNUM and virtual registers ... 139
arg_pointer_rtx ... 311
ARGS_GROW_DOWNWARD ... 304
argument passing ... 7
arguments in registers ... 316
arguments on stack ... 314
arithmetic library ... 12
arithmetic shift ... 144
arithmetic, in RTL ... 142
ARITHMETIC_TYPE_P ... 65
array ... 63
ARRAY_RANGE_REF ... 84
ARRAY_REF ... 84
ARRAY_TYPE ... 63
AS_NEEDS_DASH_FOR_PIPED_INPUT ... 270
ashift ... 144
ashift and attributes ... 250
ashiftrt ... 144
ashiftrt and attributes ... 250
ashl*m*3 instruction pattern ... 216
ashr*m*3 instruction pattern ... 216
ASM_APP_OFF ... 354
ASM_APP_ON ... 354
ASM_COMMENT_START ... 354
ASM_DECLARE_CLASS_REFERENCE ... 366
ASM_DECLARE_CONSTANT_NAME ... 361
ASM_DECLARE_FUNCTION_NAME ... 361
ASM_DECLARE_FUNCTION_SIZE ... 361
ASM_DECLARE_OBJECT_NAME ... 361
ASM_DECLARE_REGISTER_GLOBAL ... 362
ASM_DECLARE_UNRESOLVED_REFERENCE ... 366
ASM_FINAL_SPEC ... 269
ASM_FINISH_DECLARE_OBJECT ... 362
ASM_FORMAT_PRIVATE_NAME ... 365
asm_fprintf ... 372
ASM_FPRINTF_EXTENSIONS ... 372
ASM_GENERATE_INTERNAL_LABEL ... 364
asm_input ... 154
asm_input and '/v' ... 128
ASM_MAYBE_OUTPUT_ENCODED_ADDR_RTX ... 308
ASM_NO_SKIP_IN_TEXT ... 377
asm_noperands ... 160
asm_operands and '/v' ... 128
asm_operands, RTL sharing ... 166
asm_operands, usage ... 156
ASM_OUTPUT_ADDR_DIFF_ELT ... 373
ASM_OUTPUT_ADDR_VEC_ELT ... 373
ASM_OUTPUT_ALIGN ... 377
ASM_OUTPUT_ALIGN_WITH_NOP ... 377
ASM_OUTPUT_ALIGNED_BSS ... 359
ASM_OUTPUT_ALIGNED_COMMON ... 358
ASM_OUTPUT_ALIGNED_DECL_COMMON ... 358
ASM_OUTPUT_ALIGNED_DECL_LOCAL ... 359
ASM_OUTPUT_ALIGNED_LOCAL ... 359
ASM_OUTPUT_ASCII ... 356
ASM_OUTPUT_BSS ... 358
ASM_OUTPUT_CASE_END ... 373
ASM_OUTPUT_CASE_LABEL ... 373
ASM_OUTPUT_COMMON ... 358
ASM_OUTPUT_DEBUG_LABEL ... 364
ASM_OUTPUT_DEF ... 365
ASM_OUTPUT_DEF_FROM_DECLS ... 365
ASM_OUTPUT_DWARF_DELTA ... 383
ASM_OUTPUT_DWARF_OFFSET ... 383
ASM_OUTPUT_DWARF_PCREL ... 383
ASM_OUTPUT_EXTERNAL ... 363
ASM_OUTPUT_FDESC ... 356
ASM_OUTPUT_IDENT ... 355
ASM_OUTPUT_INTERNAL_LABEL ... 360
ASM_OUTPUT_LABEL ... 359
ASM_OUTPUT_LABEL_REF ... 364
ASM_OUTPUT_LABELREF ... 364

ASM_OUTPUT_LOCAL ... 359
ASM_OUTPUT_MAX_SKIP_ALIGN ... 377
ASM_OUTPUT_MEASURED_SIZE ... 360
ASM_OUTPUT_OPCODE ... 370
ASM_OUTPUT_POOL_EPILOGUE ... 357
ASM_OUTPUT_POOL_PROLOGUE ... 356
ASM_OUTPUT_REG_POP ... 373
ASM_OUTPUT_REG_PUSH ... 372
ASM_OUTPUT_SHARED_BSS ... 359
ASM_OUTPUT_SHARED_COMMON ... 358
ASM_OUTPUT_SHARED_LOCAL ... 359
ASM_OUTPUT_SIZE_DIRECTIVE ... 360
ASM_OUTPUT_SKIP ... 377
ASM_OUTPUT_SOURCE_FILENAME ... 354
ASM_OUTPUT_SPECIAL_POOL_ENTRY ... 356
ASM_OUTPUT_SYMBOL_REF ... 364
ASM_OUTPUT_TYPE_DIRECTIVE ... 361
ASM_OUTPUT_WEAK_ALIAS ... 365
ASM_OUTPUT_WEAKREF ... 362
ASM_PREFERRED_EH_DATA_FORMAT ... 308
ASM_SPEC ... 269
ASM_STABD_OP ... 379
ASM_STABN_OP ... 379
ASM_STABS_OP ... 379
ASM_WEAKEN_DECL ... 362
ASM_WEAKEN_LABEL ... 362
assemble_name ... 359
assemble_name_raw ... 360
assembler format ... 353
assembler instructions in RTL ... 156
ASSEMBLER_DIALECT ... 372
assigning attribute values to insns ... 251
assignment operator ... 76
asterisk in template ... 186
atan2*m*3 instruction pattern ... 217
attr ... 251, 252
attr_flag ... 251
attribute expressions ... 249
attribute specifications ... 253
attribute specifications example ... 253
attributes ... 84
attributes, defining ... 248
attributes, target-specific ... 387
autoincrement addressing, availability ... 5
autoincrement/decrement addressing ... 192
automata_option ... 260
automaton based pipeline description ... 256, 257
automaton based scheduler ... 256
AVOID_CCMODE_COPIES ... 294

B

backslash ... 185
barrier ... 158
barrier and '/f' ... 128
barrier and '/i' ... 129
barrier and '/v' ... 127
BASE_REG_CLASS ... 297
basic block ... 169
basic-block.h ... 169
basic_block ... 169
BASIC_BLOCK ... 169
BB_DIRTY, clear_bb_flags, update_life_info_in_dirty_blocks ... 177
BB_HEAD, BB_END ... 176
b*cond* instruction pattern ... 222
BIGGEST_ALIGNMENT ... 280
BIGGEST_FIELD_ALIGNMENT ... 281
BImode ... 133
BIND_EXPR ... 84
BINFO_TYPE ... 69
bit-fields ... 147
BIT_AND_EXPR ... 84
BIT_IOR_EXPR ... 84
BIT_NOT_EXPR ... 84
BIT_XOR_EXPR ... 84
BITFIELD_NBYTES_LIMITED ... 283
BITS_BIG_ENDIAN ... 278
BITS_BIG_ENDIAN, effect on sign_extract ... 147
BITS_PER_UNIT ... 279
BITS_PER_WORD ... 279
bitwise complement ... 144
bitwise exclusive-or ... 144
bitwise inclusive-or ... 144
bitwise logical-and ... 144
BLKmode ... 133
BLKmode, and function return values ... 165
block statement iterators ... 170, 175
BLOCK_FOR_INSN, bb_for_stmt ... 175
BLOCK_REG_PADDING ... 319
Blocks ... 100
bool ... 352, 353, 375
BOOL_TYPE_SIZE ... 287
BOOLEAN_TYPE ... 63
branch prediction ... 173
BRANCH_COST ... 342
break_out_memory_refs ... 336
BREAK_STMT ... 79
bsi_commit_edge_inserts ... 176
bsi_end_p ... 175
bsi_insert_after ... 175
bsi_insert_before ... 176
bsi_insert_on_edge ... 176
bsi_last ... 175
bsi_next ... 175
bsi_prev ... 175
bsi_remove ... 176
bsi_start ... 175
BSS_SECTION_ASM_OP ... 350
btrunc*m*2 instruction pattern ... 217
builtin_longjmp instruction pattern ... 228
builtin_setjmp_receiver instruction pattern ... 228
builtin_setjmp_setup instruction pattern ... 227
byte_mode ... 136
BYTES_BIG_ENDIAN ... 278

BYTES_BIG_ENDIAN, effect on subreg 140

C

C statements for assembler output 186
C/C++ Internal Representation 61
c_register_pragma 396
c_register_pragma_with_expansion 396
C4X_FLOAT_FORMAT 284
C99 math functions, implicit usage 334
call 130, 151
call instruction pattern 222
call usage 165
call, in mem 128
call-clobbered register 291
call-saved register 291
call-used register 291
CALL_EXPR 84
call_insn 157
call_insn and '/f' 128
call_insn and '/i' 129
call_insn and '/j' 129
call_insn and '/s' 127, 129
call_insn and '/u' 127
call_insn and '/v' 127
CALL_INSN_FUNCTION_USAGE 157
call_pop instruction pattern 223
CALL_POPS_ARGS 315
CALL_REALLY_USED_REGISTERS 291
CALL_USED_REGISTERS 291
call_used_regs 291
call_value instruction pattern 223
call_value_pop instruction pattern 223
CALLER_SAVE_PROFITABLE 323
calling conventions 304
calling functions in RTL 165
CAN_DEBUG_WITHOUT_FP 277
CAN_ELIMINATE 313
can_fallthru 169
canadian 19
CANNOT_CHANGE_MODE_CLASS 302
canonicalization of instructions 236
CANONICALIZE_COMPARISON 340
canonicalize_funcptr_for_compare instruction pattern 225
CASE_USE_BIT_TESTS 391
CASE_VALUES_THRESHOLD 391
CASE_VECTOR_MODE 391
CASE_VECTOR_PC_RELATIVE 391
CASE_VECTOR_SHORTEN_MODE 391
casesi instruction pattern 224
cbranch*mode*4 instruction pattern 222
cc_status 338
CC_STATUS_MDEP 339
CC_STATUS_MDEP_INIT 339
cc0 141
cc0, RTL sharing 166
cc0_rtx 141
CC1_SPEC 269
CC1PLUS_SPEC 269
CCmode 133
CDImode 134
CEIL_DIV_EXPR 84
CEIL_MOD_EXPR 84
ceil*m*2 instruction pattern 217
CFG, Control Flow Graph 169
cfghooks.h 174
cgraph_finalize_function 49
chain_next 420
chain_prev 420
change_address 212
char 352, 389, 401, 404
CHAR_TYPE_SIZE 287
check_stack instruction pattern 227
CHImode 134
class 69
class definitions, register 296
class preference constraints 196
CLASS_LIKELY_SPILLED_P 301
CLASS_MAX_NREGS 301
CLASS_TYPE_P 65
classes of RTX codes 122
CLASSTYPE_DECLARED_CLASS 69
CLASSTYPE_HAS_MUTABLE 71
CLASSTYPE_NON_POD_P 71
CLEANUP_DECL 79
CLEANUP_EXPR 79
CLEANUP_POINT_EXPR 84
CLEANUP_STMT 79
Cleanups 101
CLEAR_BY_PIECES_P 343
CLEAR_INSN_CACHE 332
CLEAR_RATIO 343
clobber 151
clz 145
CLZ_DEFINED_VALUE_AT_ZERO 395
clz*m*2 instruction pattern 218
cmp*m* instruction pattern 218
cmpmem*m* instruction pattern 220
cmpstr*m* instruction pattern 219
cmpstrn*m* instruction pattern 219
code generation RTL sequences 237
code macros in '.md' files 265
code_label 158
code_label and '/i' 127, 129
code_label and '/v' 127
CODE_LABEL_NUMBER 158
codes, RTL expression 121
COImode 134
COLLECT_EXPORT_LIST 402
COLLECT_SHARED_FINI_FUNC 368
COLLECT_SHARED_INIT_FUNC 368
COLLECT2_HOST_INITIALIZATION 407
combiner pass 140
commit_edge_insertions 176
compare 142

`compare`, canonicalization of ... 236
`comparison_operator` ... 189
compiler passes and files ... 49
complement, bitwise ... 144
`COMPLEX_CST` ... 84
`COMPLEX_EXPR` ... 84
`COMPLEX_TYPE` ... 63
`COMPONENT_REF` ... 84
Compound Expressions ... 99
Compound Lvalues ... 99
`COMPOUND_EXPR` ... 84
`COMPOUND_LITERAL_EXPR` ... 84
`COMPOUND_LITERAL_EXPR_DECL` ... 93
`COMPOUND_LITERAL_EXPR_DECL_STMT` ... 93
computed jump ... 172
computing the length of an insn ... 254
`cond` ... 146
`cond` and attributes ... 249
`cond_exec` ... 153
`COND_EXPR` ... 84
condition code register ... 141
condition code status ... 338
condition codes ... 145
conditional execution ... 262
Conditional Expressions ... 99
`CONDITIONAL_REGISTER_USAGE` ... 291
`conditional_trap` instruction pattern ... 229
conditions, in patterns ... 180
configuration file ... 406, 407
configure terms ... 19
`CONJ_EXPR` ... 84
`const` and '`/i`' ... 129
`CONST_DECL` ... 71
`const_double` ... 136
`const_double`, RTL sharing ... 166
`CONST_DOUBLE_CHAIN` ... 136
`CONST_DOUBLE_LOW` ... 137
`CONST_DOUBLE_MEM` ... 136
`CONST_DOUBLE_OK_FOR_CONSTRAINT_P` ... 302
`CONST_DOUBLE_OK_FOR_LETTER_P` ... 302
`const_double_operand` ... 188
`const_int` ... 136
`const_int` and attribute tests ... 250
`const_int` and attributes ... 249
`const_int`, RTL sharing ... 166
`const_int_operand` ... 188
`CONST_OK_FOR_CONSTRAINT_P` ... 302
`CONST_OK_FOR_LETTER_P` ... 302
`CONST_OR_PURE_CALL_P` ... 127
`const_string` ... 137
`const_string` and attributes ... 249
`const_true_rtx` ... 136
`const_vector` ... 136
`const_vector`, RTL sharing ... 166
`const0_rtx` ... 136
`CONST0_RTX` ... 137
`const1_rtx` ... 136
`CONST1_RTX` ... 137
`const2_rtx` ... 136
`CONST2_RTX` ... 137
constant attributes ... 255
constant definitions ... 263
`CONSTANT_ADDRESS_P` ... 334
`CONSTANT_ALIGNMENT` ... 281
`CONSTANT_P` ... 334
`CONSTANT_POOL_ADDRESS_P` ... 127
`CONSTANT_POOL_BEFORE_FUNCTION` ... 356
constants in constraints ... 192
`constm1_rtx` ... 136
constraint modifier characters ... 197
constraint, matching ... 193
`CONSTRAINT_LEN` ... 298
constraints ... 191
constraints, machine specific ... 198
constructor ... 76
`CONSTRUCTOR` ... 84
constructors, automatic calls ... 413
constructors, output of ... 366
container ... 63
`CONTINUE_STMT` ... 79
contributors ... 441
controlling register usage ... 291
controlling the compilation driver ... 267
conventions, run-time ... 7
conversions ... 148
`CONVERT_EXPR` ... 84
copy constructor ... 76
`copy_rtx` ... 337
`copy_rtx_if_shared` ... 166
`copysign`*m*`3` instruction pattern ... 217
`cos`*m*`2` instruction pattern ... 216
costs of instructions ... 341
`CP_INTEGRAL_TYPE` ... 64
`cp_namespace_decls` ... 69
`CP_TYPE_CONST_NON_VOLATILE_P` ... 64
`CP_TYPE_CONST_P` ... 64
`CP_TYPE_QUALS` ... 63, 64
`CP_TYPE_RESTRICT_P` ... 64
`CP_TYPE_VOLATILE_P` ... 64
`CPLUSPLUS_CPP_SPEC` ... 269
`CPP_SPEC` ... 269
`CQImode` ... 134
cross compilation and floating point ... 384
`CRT_CALL_STATIC_FUNCTION` ... 350
`CRTSTUFF_T_CFLAGS` ... 409
`CRTSTUFF_T_CFLAGS_S` ... 409
`CSImode` ... 134
`CTImode` ... 134
`ctz` ... 145
`CTZ_DEFINED_VALUE_AT_ZERO` ... 395
`ctz`*m*`2` instruction pattern ... 218
`CUMULATIVE_ARGS` ... 318
`current_function_epilogue_delay_list` ... 326
`current_function_is_leaf` ... 295
`current_function_outgoing_args_size` ... 314
`current_function_pops_args` ... 325

current_function_pretend_args_size 325
current_function_uses_only_leaf_regs 295
current_insn_predicate 262

D

data bypass 258, 259
data dependence delays 256
data structures 277
DATA_ALIGNMENT 281
data_section 351
DATA_SECTION_ASM_OP 349
DBR_OUTPUT_SEQEND 371
dbr_sequence_length 371
DBX_BLOCKS_FUNCTION_RELATIVE 380
DBX_CONTIN_CHAR 380
DBX_CONTIN_LENGTH 379
DBX_DEBUGGING_INFO 379
DBX_FUNCTION_FIRST 380
DBX_LINES_FUNCTION_RELATIVE 380
DBX_NO_XREFS 379
DBX_OUTPUT_LBRAC 381
DBX_OUTPUT_MAIN_SOURCE_FILE_END 382
DBX_OUTPUT_MAIN_SOURCE_FILENAME 381
DBX_OUTPUT_NFUN 381
DBX_OUTPUT_NULL_N_SO_AT_MAIN_SOURCE_FILE_END 382
DBX_OUTPUT_RBRAC 381
DBX_OUTPUT_SOURCE_LINE 381
DBX_REGISTER_NUMBER 378
DBX_REGPARM_STABS_CODE 380
DBX_REGPARM_STABS_LETTER 380
DBX_STATIC_CONST_VAR_CODE 380
DBX_STATIC_STAB_DATA_SECTION 380
DBX_TYPE_DECL_STABS_CODE 380
DBX_USE_BINCL 380
DCmode 134
De Morgan's law 237
dead_or_set_p 245
DEBUG_SYMS_TEXT 379
DEBUGGER_ARG_OFFSET 378
DEBUGGER_AUTO_OFFSET 378
DECL_ALIGN 71
DECL_ANTICIPATED 77
DECL_ARGUMENTS 79
DECL_ARRAY_DELETE_OPERATOR_P 79
DECL_ARTIFICIAL 71, 76, 79
DECL_ASSEMBLER_NAME 76, 77
DECL_ATTRIBUTES 84
DECL_BASE_CONSTRUCTOR_P 78
DECL_CLASS_SCOPE_P 72
DECL_COMPLETE_CONSTRUCTOR_P 78
DECL_COMPLETE_DESTRUCTOR_P 78
DECL_CONST_MEMFUNC_P 78
DECL_CONSTRUCTOR_P 76, 78
DECL_CONTEXT 69
DECL_CONV_FN_P 76, 78
DECL_COPY_CONSTRUCTOR_P 78
DECL_DESTRUCTOR_P 76, 78
DECL_EXTERN_C_FUNCTION_P 77
DECL_EXTERNAL 71, 77
DECL_FUNCTION_MEMBER_P 76, 77
DECL_FUNCTION_SCOPE_P 72
DECL_GLOBAL_CTOR_P 76, 78
DECL_GLOBAL_DTOR_P 76, 78
DECL_INITIAL 71
DECL_LINKONCE_P 76, 77
DECL_LOCAL_FUNCTION_P 77
DECL_MAIN_P 76
DECL_NAME 69, 71, 76
DECL_NAMESPACE_ALIAS 69
DECL_NAMESPACE_SCOPE_P 72
DECL_NAMESPACE_STD_P 69
DECL_NON_THUNK_FUNCTION_P 79
DECL_NONCONVERTING_P 78
DECL_NONSTATIC_MEMBER_FUNCTION_P 77
DECL_OVERLOADED_OPERATOR_P 76, 78
DECL_RESULT 79
DECL_SIZE 71
DECL_STATIC_FUNCTION_P 77
DECL_STMT 79
DECL_STMT_DECL 79
DECL_THUNK_P 78
DECL_VOLATILE_MEMFUNC_P 78
declaration 71
declarations, RTL 149
DECLARE_LIBRARY_RENAMES 333
decrement_and_branch_until_zero instruction pattern 225
default 418
default_file_start 353
DEFAULT_GDB_EXTENSIONS 379
DEFAULT_PCC_STRUCT_RETURN 322
DEFAULT_SIGNED_CHAR 288
define_asm_attributes 253
define_attr 248
define_automaton 257
define_bypass 259
define_code_attr 265
define_code_macro 265
define_cond_exec 262
define_constants 263
define_cpu_unit 257
define_delay 255
define_expand 237
define_insn 179
define_insn example 180
define_insn_and_split 243
define_insn_reservation 258
define_mode_attr 264
define_mode_macro 264
define_peephole 245
define_peephole2 247
define_predicate 190
define_query_cpu_unit 257
define_reservation 259

`define_special_predicate` ... 190
`define_split` ... 240
defining attributes and their values ... 248
defining jump instruction patterns ... 233
defining looping instruction patterns ... 235
defining peephole optimizers ... 244
defining predicates ... 190
defining RTL sequences for code generation ... 237
delay slots, defining ... 255
`DELAY_SLOTS_FOR_EPILOGUE` ... 326
`deletable` ... 420
`DELETE_IF_ORDINARY` ... 407
Dependent Patterns ... 232
`desc` ... 418
destructor ... 76
destructors, output of ... 366
deterministic finite state automaton ... 256, 260
`DFmode` ... 133
digits in constraint ... 193
`DImode` ... 133
`DIR_SEPARATOR` ... 406
`DIR_SEPARATOR_2` ... 406
directory options .md ... 244
disabling certain registers ... 291
dispatch table ... 373
`div` ... 143
`div` and attributes ... 250
division ... 143, 144
`div`*m*`3` instruction pattern ... 214
`divmod`*m*`4` instruction pattern ... 216
`DO_BODY` ... 79
`DO_COND` ... 79
`DO_STMT` ... 79
`DOLLARS_IN_IDENTIFIERS` ... 397
`doloop_begin` instruction pattern ... 225
`doloop_end` instruction pattern ... 225
`DONE` ... 238
`DOUBLE_TYPE_SIZE` ... 287
driver ... 267
`DRIVER_SELF_SPECS` ... 268
`DUMPFILE_FORMAT` ... 407
`DWARF_ALT_FRAME_RETURN_COLUMN` ... 306
`DWARF_CIE_DATA_ALIGNMENT` ... 375
`DWARF_FRAME_REGISTERS` ... 312
`DWARF_FRAME_REGNUM` ... 312
`DWARF_REG_TO_UNWIND_COLUMN` ... 312
`DWARF_ZERO_REG` ... 306
`DWARF2_ASM_LINE_DEBUG_INFO` ... 383
`DWARF2_DEBUGGING_INFO` ... 382
`DWARF2_FRAME_INFO` ... 382
`DWARF2_FRAME_REG_OUT` ... 312
`DWARF2_UNWIND_INFO` ... 374
`DYNAMIC_CHAIN_ADDRESS` ... 305

E

'`E`' in constraint ... 192
earlyclobber operand ... 197
`edge` ... 170
edge in the flow graph ... 170
edge iterators ... 170
edge splitting ... 176
`EDGE_ABNORMAL` ... 172
`EDGE_ABNORMAL, EDGE_ABNORMAL_CALL` ... 173
`EDGE_ABNORMAL, EDGE_EH` ... 171
`EDGE_ABNORMAL, EDGE_SIBCALL` ... 172
`EDGE_FALLTHRU, force_nonfallthru` ... 171
`EDOM`, implicit usage ... 333
`EH_FRAME_IN_DATA_SECTION` ... 374
`EH_FRAME_SECTION_NAME` ... 374
`eh_return` instruction pattern ... 228
`EH_RETURN_DATA_REGNO` ... 307
`EH_RETURN_HANDLER_RTX` ... 308
`EH_RETURN_STACKADJ_RTX` ... 307
`EH_TABLES_CAN_BE_READ_ONLY` ... 374
`EH_USES` ... 325
`ei_edge` ... 171
`ei_end_p` ... 170
`ei_last` ... 170
`ei_next` ... 170
`ei_one_before_end_p` ... 170
`ei_prev` ... 171
`ei_safe_safe` ... 171
`ei_start` ... 170
`ELIGIBLE_FOR_EPILOGUE_DELAY` ... 326
`ELIMINABLE_REGS` ... 313
`ELSE_CLAUSE` ... 79
`EMIT_MODE_SET` ... 387
Empty Statements ... 101
`EMPTY_CLASS_EXPR` ... 79
`EMPTY_FIELD_BOUNDARY` ... 281
`ENABLE_EXECUTE_STACK` ... 332
`ENDFILE_SPEC` ... 271
endianness ... 5
`ENTRY_BLOCK_PTR, EXIT_BLOCK_PTR` ... 169
`enum machine_mode` ... 132
`enum reg_class` ... 296
`ENUMERAL_TYPE` ... 63
epilogue ... 323
`epilogue` instruction pattern ... 228
`EPILOGUE_USES` ... 325
`eq` ... 146
`eq` and attributes ... 250
`eq_attr` ... 250
`EQ_EXPR` ... 84
equal ... 146
`errno`, implicit usage ... 334
`EXACT_DIV_EXPR` ... 84
examining SSA_NAMEs ... 115
exception handling ... 171, 307
`exception_receiver` instruction pattern ... 227
exclamation point ... 196
`exclusion_set` ... 259
exclusive-or, bitwise ... 144
`EXIT_EXPR` ... 84
`EXIT_IGNORE_STACK` ... 325

expander definitions ... 237
`exp`*m*`2` instruction pattern ... 216
`expr_list` ... 164
`EXPR_STMT` ... 79
`EXPR_STMT_EXPR` ... 79
expression ... 84
expression codes ... 121
`extend`*mn*`2` instruction pattern ... 221
extensible constraints ... 194
`EXTRA_ADDRESS_CONSTRAINT` ... 303
`EXTRA_CONSTRAINT` ... 302
`EXTRA_CONSTRAINT_STR` ... 303
`EXTRA_MEMORY_CONSTRAINT` ... 303
`EXTRA_SECTION_FUNCTIONS` ... 351
`EXTRA_SECTIONS` ... 350
`EXTRA_SPECS` ... 271
`extv` instruction pattern ... 221
`extzv` instruction pattern ... 221

F

'`F`' in constraint ... 192
`FAIL` ... 238
`fall-thru` ... 171
`FATAL_EXIT_CODE` ... 407
FDL, GNU Free Documentation License ... 433
features, optional, in system conventions ... 276
`ffs` ... 145
`ffs`*m*`2` instruction pattern ... 218
`FIELD_DECL` ... 71
`file_end_indicate_exec_stack` ... 354
files and passes of the compiler ... 49
files, generated ... 421
`final_absence_set` ... 259
`FINAL_PRESCAN_INSN` ... 370
`final_presence_set` ... 259
`final_scan_insn` ... 326
`final_sequence` ... 372
`FIND_BASE_TERM` ... 336
`FINI_ARRAY_SECTION_ASM_OP` ... 350
`FINI_SECTION_ASM_OP` ... 350
finite state automaton minimization ... 260
`FIRST_PARM_OFFSET` ... 305
`FIRST_PARM_OFFSET` and virtual registers ... 139
`FIRST_PSEUDO_REGISTER` ... 290
`FIRST_STACK_REG` ... 295
`FIRST_VIRTUAL_REGISTER` ... 138
`fix` ... 149
`FIX_TRUNC_EXPR` ... 84
`fix_trunc`*mn*`2` instruction pattern ... 220
fixed register ... 290
`FIXED_REGISTERS` ... 290
`fixed_regs` ... 291
`fix`*mn*`2` instruction pattern ... 220
`FIXUNS_TRUNC_LIKE_FIX_TRUNC` ... 392
`fixuns_trunc`*mn*`2` instruction pattern ... 220
`fixuns`*mn*`2` instruction pattern ... 220
flags in RTL expression ... 127
`float` ... 149
`FLOAT_EXPR` ... 84
`float_extend` ... 148
`FLOAT_LIB_COMPARE_RETURNS_BOOL` ... 333
`FLOAT_STORE_FLAG_VALUE` ... 395
`float_truncate` ... 149
`FLOAT_TYPE_SIZE` ... 287
`FLOAT_WORDS_BIG_ENDIAN` ... 278
`FLOAT_WORDS_BIG_ENDIAN`, (lack of) effect on `subreg` ... 140
floating point and cross compilation ... 384
`Floating Point Emulation` ... 409
floating point emulation library, US Software GOFAST ... 333
`float`*mn*`2` instruction pattern ... 220
`floatuns`*mn*`2` instruction pattern ... 220
`FLOOR_DIV_EXPR` ... 84
`FLOOR_MOD_EXPR` ... 84
`floor`*m*`2` instruction pattern ... 217
flow-insensitive alias analysis ... 116
flow-sensitive alias analysis ... 116
`FOR_BODY` ... 79
`FOR_COND` ... 79
`FOR_EXPR` ... 79
`FOR_INIT_STMT` ... 79
`FOR_STMT` ... 79
`FORCE_CODE_SECTION_ALIGN` ... 350
`force_reg` ... 212
frame layout ... 304
`FRAME_GROWS_DOWNWARD` ... 304
`FRAME_GROWS_DOWNWARD` and virtual registers ... 139
`FRAME_POINTER_CFA_OFFSET` ... 307
`frame_pointer_needed` ... 323
`FRAME_POINTER_REGNUM` ... 310
`FRAME_POINTER_REGNUM` and virtual registers ... 139
`FRAME_POINTER_REQUIRED` ... 312
`frame_pointer_rtx` ... 311
`frame_related` ... 130
`frame_related`, in `insn`, `call_insn`, `jump_insn`, `barrier`, and `set` ... 128
`frame_related`, in `mem` ... 128
`frame_related`, in `reg` ... 128
`frame_related`, in `symbol_ref` ... 129
`frequency`, `count`, `BB_FREQ_BASE` ... 173
`ftrunc`*m*`2` instruction pattern ... 220
function ... 76
function body ... 79
function call conventions ... 7
function entry and exit ... 323
function entry point, alternate function entry point ... 173
function-call insns ... 165
`FUNCTION_ARG` ... 316
`FUNCTION_ARG_ADVANCE` ... 318
`FUNCTION_ARG_BOUNDARY` ... 319
`FUNCTION_ARG_PADDING` ... 319
`FUNCTION_ARG_REGNO_P` ... 319
`FUNCTION_BOUNDARY` ... 280

FUNCTION_DECL 76
FUNCTION_INCOMING_ARG 317
FUNCTION_MODE 395
FUNCTION_OUTGOING_VALUE 321
FUNCTION_PROFILER 327
FUNCTION_TYPE 63
FUNCTION_VALUE 320
FUNCTION_VALUE_REGNO_P 321
functions, leaf 294
fundamental type 63

G

'g' in constraint 193
'G' in constraint 192
GCC and portability 5
GCC_DRIVER_HOST_INITIALIZATION 407
gcov_type 174
ge 146
ge and attributes 250
GE_EXPR 84
GEN_ERRNO_RTX 334
gencodes 55
general_operand 189
GENERAL_REGS 296
generated files 421
generating assembler output 186
generating insns 181
GENERIC 49, 50, 97
generic predicates 188
genflags 55
get_attr 250
get_attr_length 254
GET_CLASS_NARROWEST_MODE 135
GET_CODE 121
get_frame_size 313
get_insns 156
get_last_insn 156
GET_MODE 135
GET_MODE_ALIGNMENT 135
GET_MODE_BITSIZE 135
GET_MODE_CLASS 135
GET_MODE_MASK 135
GET_MODE_NAME 135
GET_MODE_NUNITS 135
GET_MODE_SIZE 135
GET_MODE_UNIT_SIZE 135
GET_MODE_WIDER_MODE 135
GET_RTX_CLASS 122
GET_RTX_FORMAT 124
GET_RTX_LENGTH 124
geu 146
geu and attributes 250
GGC 417
GIMPLE 49, 50, 97
GIMPLE Example 102
GIMPLE Exception Handling 102
GIMPLE Expressions 99
gimplification 49, 50, 98
gimplifier 49
gimplify_expr 50
gimplify_function_tree 50
GLOBAL_INIT_PRIORITY 76, 79
global_live_at_start, global_live_at_end 177
global_regs 291
GO_IF_LEGITIMATE_ADDRESS 335
GO_IF_MODE_DEPENDENT_ADDRESS 337
GOFAST, floating point emulation library 333
gofast_maybe_init_libfuncs 333
greater than 146
gt 146
gt and attributes 250
GT_EXPR 84
gtu 146
gtu and attributes 250
GTY 417

H

'H' in constraint 192
HANDLE_PRAGMA_PACK_PUSH_POP 397
HANDLE_PRAGMA_PACK_WITH_EXPANSION 397
HANDLE_SYSV_PRAGMA 397
HANDLER 79
HANDLER_BODY 79
HANDLER_PARMS 79
hard registers 138
HARD_FRAME_POINTER_REGNUM 310
HARD_REGNO_CALL_PART_CLOBBERED 291
HARD_REGNO_CALLER_SAVE_MODE 323
HARD_REGNO_MODE_OK 293
HARD_REGNO_NREGS 293
HARD_REGNO_RENAME_OK 294
HAS_INIT_SECTION 368
HAS_LONG_COND_BRANCH 390
HAS_LONG_UNCOND_BRANCH 391
HAVE_DOS_BASED_FILE_SYSTEM 406
HAVE_POST_DECREMENT 334
HAVE_POST_INCREMENT 334
HAVE_POST_MODIFY_DISP 334
HAVE_POST_MODIFY_REG 334
HAVE_PRE_DECREMENT 334
HAVE_PRE_INCREMENT 334
HAVE_PRE_MODIFY_DISP 334
HAVE_PRE_MODIFY_REG 334
HCmode 134
HFmode 133
high 138
HImode 133
HImode, in insn 160
host configuration 405
host functions 405
host hooks 405
host makefile fragment 411
HOST_BIT_BUCKET 406

HOST_EXECUTABLE_SUFFIX 406
HOST_HOOKS_EXTRA_SIGNALS 405
HOST_HOOKS_GT_PCH_ALLOC_GRANULARITY 405
HOST_HOOKS_GT_PCH_USE_ADDRESS 405
HOST_LACKS_INODE_NUMBERS 407
HOST_LONG_LONG_FORMAT 408
HOST_OBJECT_SUFFIX 406
HOT_TEXT_SECTION_NAME 349

I

'i' in constraint 192
'I' in constraint 192
IBM_FLOAT_FORMAT 284
identifier 62
IDENTIFIER_LENGTH 63
IDENTIFIER_NODE 62
IDENTIFIER_OPNAME_P 63
IDENTIFIER_POINTER 63
IDENTIFIER_TYPENAME_P 63
IEEE_FLOAT_FORMAT 284
IF_COND 79
if_marked 420
IF_STMT 79
if_then_else 146
if_then_else and attributes 249
if_then_else usage 150
IFCVT_EXTRA_FIELDS 400
IFCVT_INIT_EXTRA_FIELDS 399
IFCVT_MODIFY_CANCEL 399
IFCVT_MODIFY_FINAL 399
IFCVT_MODIFY_INSN 399
IFCVT_MODIFY_MULTIPLE_TESTS 399
IFCVT_MODIFY_TESTS 399
IMAGPART_EXPR 84
Immediate Uses 111
immediate_operand 188
IMMEDIATE_PREFIX 372
in_data 350
in_struct 131
in_struct, in code_label and note 127
in_struct, in insn and jump_insn and call_insn 127
in_struct, in insn, jump_insn and call_insn 129
in_struct, in label_ref 127
in_struct, in mem 128
in_struct, in subreg 130
in_text 350
include 243
INCLUDE_DEFAULTS 274
inclusive-or, bitwise 144
INCOMING_FRAME_SP_OFFSET 306
INCOMING_REGNO 292
INCOMING_RETURN_ADDR_RTX 306
INDEX_REG_CLASS 297
indirect_jump instruction pattern 224
indirect_operand 189
INDIRECT_REF 84
INIT_ARRAY_SECTION_ASM_OP 350
INIT_CUMULATIVE_ARGS 318
INIT_CUMULATIVE_INCOMING_ARGS 318
INIT_CUMULATIVE_LIBCALL_ARGS 318
INIT_ENVIRONMENT 273
INIT_EXPANDERS 278
INIT_EXPR 84
init_machine_status 278
init_one_libfunc 333
INIT_SECTION_ASM_OP 350, 368
INITIAL_ELIMINATION_OFFSET 314
INITIAL_FRAME_ADDRESS_RTX 305
INITIAL_FRAME_POINTER_OFFSET 313
initialization routines 366
INITIALIZE_TRAMPOLINE 331
inlining 388
insert_insn_on_edge 176
insn 157
insn and '/f' 128
insn and '/i' 129
insn and '/j' 129
insn and '/s' 127, 129
insn and '/u' 127
insn and '/v' 127
insn attributes 248
insn canonicalization 236
insn includes 243
insn lengths, computing 254
insn splitting 240
insn-attr.h 248
INSN_ANNULLED_BRANCH_P 127
INSN_CODE 160
INSN_DELETED_P 127
INSN_FROM_TARGET_P 127
insn_list 164
insn_list and '/i' 129
INSN_REFERENCES_ARE_DELAYED 398
INSN_SETS_ARE_DELAYED 398
INSN_UID 156
insns 156
insns, generating 181
insns, recognizing 181
instruction attributes 248
instruction latency time 256, 258, 259
instruction patterns 179
instruction splitting 240
insv instruction pattern 221
int 276
INT_TYPE_SIZE 286
INTEGER_CST 84
INTEGER_TYPE 63
integrated 131
integrated, in insn, call_insn, jump_insn, barrier, code_label, insn_list, const, and note 129
integrated, in reg 128
integrated, in symbol_ref 130

Interdependence of Patterns ... 232
interfacing to GCC output ... 7
interlock delays ... 256
intermediate representation lowering ... 49
INTMAX_TYPE ... 289
introduction ... 1
INVOKE__main ... 369
ior ... 144
ior and attributes ... 250
ior, canonicalization of ... 237
iorm3 instruction pattern ... 214
IS_ASM_LOGICAL_LINE_SEPARATOR ... 357

J

jump ... 131
jump instruction pattern ... 222
jump instruction patterns ... 233
jump instructions and set ... 150
jump, in call_insn ... 129
jump, in insn ... 129
jump, in mem ... 128
JUMP_ALIGN ... 376
jump_insn ... 157
jump_insn and '/f' ... 128
jump_insn and '/i' ... 129
jump_insn and '/s' ... 127, 129
jump_insn and '/u' ... 127
jump_insn and '/v' ... 127
JUMP_LABEL ... 157
JUMP_TABLES_IN_TEXT_SECTION ... 351
Jumps ... 101

L

LABEL_ALIGN ... 377
LABEL_ALIGN_AFTER_BARRIER ... 376
LABEL_ALIGN_AFTER_BARRIER_MAX_SKIP ... 376
LABEL_ALIGN_MAX_SKIP ... 377
LABEL_ALT_ENTRY_P ... 158
LABEL_ALTERNATE_NAME ... 173
LABEL_DECL ... 71
LABEL_KIND ... 158
LABEL_NUSES ... 158
LABEL_OUTSIDE_LOOP_P ... 127
LABEL_PRESERVE_P ... 127
label_ref ... 137
label_ref and '/s' ... 127
label_ref and '/v' ... 128
label_ref, RTL sharing ... 166
LABEL_REF_NONLOCAL_P ... 128
lang_hooks.gimplify_expr ... 50
lang_hooks.parse_file ... 49
language-independent intermediate representation ... 49
large return values ... 322
LARGEST_EXPONENT_IS_NORMAL ... 285
LAST_STACK_REG ... 295
LAST_VIRTUAL_REGISTER ... 138
LD_FINI_SWITCH ... 368
LD_INIT_SWITCH ... 368
LDD_SUFFIX ... 370
le ... 146
le and attributes ... 250
LE_EXPR ... 84
leaf functions ... 294
leaf_function_p ... 223
LEAF_REG_REMAP ... 295
LEAF_REGISTERS ... 295
left rotate ... 144
left shift ... 144
LEGITIMATE_CONSTANT_P ... 337
LEGITIMATE_PIC_OPERAND_P ... 353
LEGITIMIZE_ADDRESS ... 336
LEGITIMIZE_RELOAD_ADDRESS ... 337
length ... 418
less than ... 146
less than or equal ... 146
leu ... 146
leu and attributes ... 250
LIB_SPEC ... 270
LIB2FUNCS_EXTRA ... 409
LIBCALL_VALUE ... 321
'libgcc.a' ... 333
LIBGCC_SPEC ... 270
LIBGCC2_CFLAGS ... 409
LIBGCC2_HAS_DF_MODE ... 287
LIBGCC2_HAS_TF_MODE ... 288
LIBGCC2_HAS_XF_MODE ... 287
LIBGCC2_LONG_DOUBLE_TYPE_SIZE ... 287
LIBGCC2_WORDS_BIG_ENDIAN ... 278
library subroutine names ... 333
LIBRARY_PATH_ENV ... 398
LIMIT_RELOAD_CLASS ... 299
LINK_COMMAND_SPEC ... 272
LINK_EH_SPEC ... 270
LINK_ELIMINATE_DUPLICATE_LDIRECTORIES ... 272
LINK_GCC_C_SEQUENCE_SPEC ... 272
LINK_LIBGCC_SPECIAL_1 ... 272
LINK_SPEC ... 270
linkage ... 76
list ... 63
Liveness representation ... 176
lo_sum ... 142
load address instruction ... 193
LOAD_EXTEND_OP ... 391
load_multiple instruction pattern ... 214
LOCAL_ALIGNMENT ... 281
LOCAL_CLASS_P ... 70
LOCAL_INCLUDE_DIR ... 273
LOCAL_LABEL_PREFIX ... 372
LOCAL_REGNO ... 292
LOG_LINKS ... 160
Logical Operators ... 100
logical-and, bitwise ... 144
logm2 instruction pattern ... 216

LONG_DOUBLE_TYPE_SIZE ... 287
LONG_LONG_TYPE_SIZE ... 287
LONG_TYPE_SIZE ... 287
longjmp and automatic variables ... 7
LOOP_ALIGN ... 376
LOOP_ALIGN_MAX_SKIP ... 376
LOOP_EXPR ... 84
looping instruction patterns ... 235
Loops ... 101
lowering, language-dependent intermediate representation ... 49
LSHIFT_EXPR ... 84
lshiftrt ... 144
lshiftrt and attributes ... 250
lshr*m*3 instruction pattern ... 216
lt ... 146
lt and attributes ... 250
LT_EXPR ... 84
LTGT_EXPR ... 84
ltu ... 146

M

'm' in constraint ... 191
machine attributes ... 387
machine description macros ... 267
machine descriptions ... 179
machine mode conversions ... 148
machine modes ... 132
machine specific constraints ... 198
machine-independent predicates ... 188
machine_mode ... 341
macros in '.md' files ... 263
macros, target description ... 267
MAKE_DECL_ONE_ONLY ... 363
make_safe_from ... 240
makefile fragment ... 409
makefile targets ... 23
marking roots ... 421
MASK_RETURN_ADDR ... 374
match_dup ... 182, 247
match_dup and attributes ... 254
match_op_dup ... 183
match_operand ... 181
match_operand and attributes ... 250
match_operator ... 182
match_par_dup ... 184
match_parallel ... 184
match_scratch ... 182, 247
matching constraint ... 193
matching operands ... 185
math library ... 12
math, in RTL ... 142
MATH_LIBRARY ... 398
matherr ... 333
MAX_BITS_PER_WORD ... 279
MAX_CONDITIONAL_EXECUTE ... 399
MAX_DFA_ISSUE_RATE ... 349
MAX_FIXED_MODE_SIZE ... 283
MAX_MOVE_MAX ... 392
MAX_OFILE_ALIGNMENT ... 281
MAX_REGS_PER_ADDRESS ... 335
max*m*3 instruction pattern ... 215
may_trap_p, tree_could_trap_p ... 172
maybe_undef ... 420
mcount ... 327
MD_CAN_REDIRECT_BRANCH ... 401
MD_EXEC_PREFIX ... 272
MD_FALLBACK_FRAME_STATE_FOR ... 308
MD_HANDLE_UNWABI ... 309
MD_STARTFILE_PREFIX ... 273
MD_STARTFILE_PREFIX_1 ... 273
MD_UNWIND_SUPPORT ... 308
mem ... 142
mem and '/c' ... 128
mem and '/f' ... 128
mem and '/j' ... 128
mem and '/s' ... 128
mem and '/u' ... 129
mem and '/v' ... 128
mem, RTL sharing ... 166
MEM_ALIAS_SET ... 125
MEM_ALIGN ... 126
MEM_EXPR ... 125
MEM_IN_STRUCT_P ... 128
MEM_KEEP_ALIAS_SET_P ... 128
MEM_NOTRAP_P ... 128
MEM_OFFSET ... 125
MEM_READONLY_P ... 129
MEM_SCALAR_P ... 128
MEM_SIZE ... 125
MEM_VOLATILE_P ... 128
MEMBER_TYPE_FORCES_BLK ... 283
memory reference, nonoffsettable ... 195
memory references in constraints ... 191
memory_barrier instruction pattern ... 229
MEMORY_MOVE_COST ... 341
memory_operand ... 189
METHOD_TYPE ... 63
MIN_UNITS_PER_WORD ... 279
MINIMUM_ATOMIC_ALIGNMENT ... 280
min*m*3 instruction pattern ... 215
minus ... 142
minus and attributes ... 250
minus, canonicalization of ... 236
MINUS_EXPR ... 84
MIPS coprocessor-definition macros ... 388
mod ... 144
mod and attributes ... 250
mode classes ... 134
mode macros in '.md' files ... 263
mode switching ... 386
MODE_AFTER ... 386
MODE_BASE_REG_CLASS ... 297
MODE_BASE_REG_REG_CLASS ... 297
MODE_CC ... 135

MODE_COMPLEX_FLOAT ... 134
MODE_COMPLEX_INT ... 134
MODE_ENTRY ... 386
MODE_EXIT ... 386
MODE_FLOAT ... 134
MODE_FUNCTION ... 134
MODE_HAS_INFINITIES ... 284
MODE_HAS_NANS ... 284
MODE_HAS_SIGN_DEPENDENT_ROUNDING ... 285
MODE_HAS_SIGNED_ZEROS ... 284
MODE_INT ... 134
MODE_NEEDED ... 386
MODE_PARTIAL_INT ... 134
MODE_PRIORITY_TO_MODE ... 387
MODE_RANDOM ... 135
MODES_TIEABLE_P ... 294
modifiers in constraints ... 197
MODIFY_EXPR ... 84
MODIFY_JNI_METHOD_CALL ... 402
MODIFY_TARGET_NAME ... 273
mod*m*3 instruction pattern ... 214
modulo scheduling ... 57
MOVE_BY_PIECES_P ... 343
MOVE_MAX ... 392
MOVE_MAX_PIECES ... 343
MOVE_RATIO ... 342
mov*m* instruction pattern ... 212
movmem*m* instruction pattern ... 218
movmisalign*m* instruction pattern ... 213
mov*mode*cc instruction pattern ... 221
movstr instruction pattern ... 219
movstrict*m* instruction pattern ... 213
mulhisi3 instruction pattern ... 215
mul*m*3 instruction pattern ... 214
mulqihi3 instruction pattern ... 215
mulsidi3 instruction pattern ... 215
mult ... 143
mult and attributes ... 250
mult, canonicalization of ... 236
MULT_EXPR ... 84
MULTILIB_DEFAULTS ... 272
MULTILIB_DIRNAMES ... 410
MULTILIB_EXCEPTIONS ... 410
MULTILIB_EXTRA_OPTS ... 410
MULTILIB_MATCHES ... 410
MULTILIB_OPTIONS ... 409
multiple alternative constraints ... 196
MULTIPLE_SYMBOL_SPACES ... 398
multiplication ... 143
MUST_USE_SJLJ_EXCEPTIONS ... 375

N

'n' in constraint ... 192
N_REG_CLASSES ... 297
name ... 62
named patterns and conditions ... 180
names, pattern ... 212
namespace ... 68
namespace, class, scope ... 68
NAMESPACE_DECL ... 68, 71
NATIVE_SYSTEM_HEADER_DIR ... 410
ne ... 146
ne and attributes ... 250
NE_EXPR ... 84
nearbyint*m*2 instruction pattern ... 217
neg ... 143
neg and attributes ... 250
neg, canonicalization of ... 236
NEGATE_EXPR ... 84
neg*m*2 instruction pattern ... 216
nested functions, trampolines for ... 330
nested_ptr ... 420
next_bb, prev_bb, FOR_EACH_BB ... 169
next_cc0_user ... 234
NEXT_INSN ... 156
NEXT_OBJC_RUNTIME ... 334
nil ... 122
NO_DBX_BNSYM_ENSYM ... 381
NO_DBX_FUNCTION_END ... 381
NO_DBX_GCC_MARKER ... 382
NO_DBX_MAIN_SOURCE_DIRECTORY ... 382
NO_DOLLAR_IN_LABEL ... 397
NO_DOT_IN_LABEL ... 398
NO_FUNCTION_CSE ... 344
NO_IMPLICIT_EXTERN_C ... 396
no_new_pseudos ... 213
NO_PROFILE_COUNTERS ... 327
NO_REGS ... 296
NON_LVALUE_EXPR ... 84
nondeterministic finite state automaton ... 260
nonimmediate_operand ... 189
nonlocal goto handler ... 173
nonlocal_goto instruction pattern ... 227
nonlocal_goto_receiver instruction pattern ... 227
nonmemory_operand ... 189
nonoffsettable memory reference ... 195
nop instruction pattern ... 224
NOP_EXPR ... 84
normal predicates ... 187
not ... 144
not and attributes ... 250
not equal ... 146
not, canonicalization of ... 236
note ... 158
note and '/i' ... 127, 129
note and '/v' ... 127
NOTE_INSN_BASIC_BLOCK, CODE_LABEL, notes ... 169
NOTE_INSN_BLOCK_BEG ... 159
NOTE_INSN_BLOCK_END ... 159
NOTE_INSN_DELETED ... 159
NOTE_INSN_DELETED_LABEL ... 159
NOTE_INSN_EH_REGION_BEG ... 159
NOTE_INSN_EH_REGION_END ... 159

NOTE_INSN_FUNCTION_BEG ... 159
NOTE_INSN_FUNCTION_END ... 159
NOTE_INSN_LOOP_BEG ... 159
NOTE_INSN_LOOP_CONT ... 159
NOTE_INSN_LOOP_END ... 159
NOTE_INSN_LOOP_VTOP ... 159
NOTE_LINE_NUMBER ... 158
NOTE_SOURCE_FILE ... 158
NOTICE_UPDATE_CC ... 339
NUM_MACHINE_MODES ... 135
NUM_MODES_FOR_MODE_SWITCHING ... 386

O

'o' in constraint ... 192
OBJC_GEN_METHOD_LABEL ... 366
OBJECT_FORMAT_COFF ... 369
OFFSET_TYPE ... 63
offsettable address ... 192
OImode ... 133
one_cmpl*m*2 instruction pattern ... 218
operand access ... 124
Operand Access Routines ... 108
operand constraints ... 191
Operand Iterators ... 108
operand predicates ... 187
operand substitution ... 185
operands ... 107
operands ... 180
operator predicates ... 187
Optimization infrastructure for GIMPLE ... 97
OPTIMIZATION_OPTIONS ... 277
OPTIMIZE_MODE_SWITCHING ... 386
option specification files ... 45
OPTION_DEFAULT_SPECS ... 269
optional hardware or system features ... 276
options, directory search ... 244
'opts.sh' ... 45
order of register allocation ... 292
ORDER_REGS_FOR_LOCAL_ALLOC ... 292
ORDERED_EXPR ... 84
Ordering of Patterns ... 232
ORIGINAL_REGNO ... 126
other register constraints ... 194
OUTGOING_REG_PARM_STACK_SPACE ... 315
OUTGOING_REGNO ... 292
output of assembler code ... 353
output statements ... 186
output templates ... 185
OUTPUT_ADDR_CONST_EXTRA ... 356
output_asm_insn ... 186
OUTPUT_QUOTED_STRING ... 354
OVERLOAD ... 76
OVERRIDE_OPTIONS ... 276
OVL_CURRENT ... 76
OVL_NEXT ... 76

P

'p' in constraint ... 193
PAD_VARARGS_DOWN ... 319
parallel ... 152
param_is ... 419
parameters, c++ abi ... 389
parameters, miscellaneous ... 390
parameters, precompiled headers ... 389
param*n*_is ... 419
parity ... 145
parity*m*2 instruction pattern ... 218
PARM_BOUNDARY ... 280
PARM_DECL ... 71
PARSE_LDD_OUTPUT ... 370
passes and files of the compiler ... 49
passing arguments ... 7
PATH_SEPARATOR ... 406
PATTERN ... 160
pattern conditions ... 180
pattern names ... 212
Pattern Ordering ... 232
patterns ... 179
pc ... 141
pc and attributes ... 254
pc, RTL sharing ... 166
PC_REGNUM ... 292
pc_rtx ... 141
PCC_BITFIELD_TYPE_MATTERS ... 282
PCC_STATIC_STRUCT_RETURN ... 322
PDImode ... 133
peephole optimization, RTL representation ... 153
peephole optimizer definitions ... 244
per-function data ... 277
percent sign ... 185
PHI_ARG_DEF ... 113
PHI_ARG_EDGE ... 113
PHI_ARG_ELT ... 113
PHI_NUM_ARGS ... 113
PHI_RESULT ... 113
PIC ... 353
PIC_OFFSET_TABLE_REG_CALL_CLOBBERED ... 353
PIC_OFFSET_TABLE_REGNUM ... 353
pipeline hazard recognizer ... 256, 257
plus ... 142
plus and attributes ... 250
plus, canonicalization of ... 236
PLUS_EXPR ... 84
Pmode ... 395
pmode_register_operand ... 188
pointer ... 63
POINTER_SIZE ... 279
POINTER_TYPE ... 63
POINTERS_EXTEND_UNSIGNED ... 279
pop_operand ... 189
popcount ... 145
popcount*m*2 instruction pattern ... 218
portability ... 5
position independent code ... 353

post_dec ... 155
post_inc ... 155
post_modify ... 155
POSTDECREMENT_EXPR ... 84
POSTINCREMENT_EXPR ... 84
POWI_MAX_MULTS ... 403
pow*m*3 instruction pattern ... 217
pragma ... 396, 397
pre_dec ... 154
PRE_GCC3_DWARF_FRAME_REGISTERS ... 312
pre_inc ... 155
pre_modify ... 155
PREDECREMENT_EXPR ... 84
predefined macros ... 275
predicates ... 187
predicates and machine modes ... 187
predication ... 262
predict.def ... 173
PREFERRED_DEBUGGING_TYPE ... 378
PREFERRED_OUTPUT_RELOAD_CLASS ... 299
PREFERRED_RELOAD_CLASS ... 299
PREFERRED_STACK_BOUNDARY ... 280
prefetch ... 154
prefetch instruction pattern ... 229
PREINCREMENT_EXPR ... 84
presence_set ... 259
preserving SSA form ... 113
prev_active_insn ... 245
prev_cc0_setter ... 234
PREV_INSN ... 156
PRINT_OPERAND ... 371
PRINT_OPERAND_ADDRESS ... 371
PRINT_OPERAND_PUNCT_VALID_P ... 371
processor functional units ... 256, 257
processor pipeline description ... 256
product ... 143
profile feedback ... 173
profile representation ... 173
PROFILE_BEFORE_PROLOGUE ... 327
PROFILE_HOOK ... 327
profiling, code generation ... 327
program counter ... 141
prologue ... 323
prologue instruction pattern ... 228
PROMOTE_FUNCTION_MODE ... 280
PROMOTE_MODE ... 279
pseudo registers ... 138
PSImode ... 133
PTRDIFF_TYPE ... 288
PTRMEM_CST ... 84
PTRMEM_CST_CLASS ... 84
PTRMEM_CST_MEMBER ... 84
purge_dead_edges ... 171, 176
push address instruction ... 193
PUSH_ARGS ... 314
PUSH_ARGS_REVERSED ... 314
push_operand ... 189
push_reload ... 337
PUSH_ROUNDING ... 314
push*m*1 instruction pattern ... 214
PUT_CODE ... 121
PUT_MODE ... 135
PUT_REG_NOTE_KIND ... 161
PUT_SDB_... 383

Q

QCmode ... 134
QFmode ... 133
QImode ... 133
QImode, in insn ... 160
qualified type ... 63
querying function unit reservations ... 257
question mark ... 196
quotient ... 143

R

'r' in constraint ... 192
RANGE_TEST_NON_SHORT_CIRCUIT ... 344
RDIV_EXPR ... 84
READONLY_DATA_SECTION ... 349
READONLY_DATA_SECTION_ASM_OP ... 349
real operands ... 107
REAL_ARITHMETIC ... 385
REAL_CST ... 84
REAL_LIBGCC_SPEC ... 270
REAL_NM_FILE_NAME ... 369
REAL_TYPE ... 63
REAL_VALUE_ABS ... 385
REAL_VALUE_ATOF ... 385
REAL_VALUE_FIX ... 384
REAL_VALUE_FROM_INT ... 385
REAL_VALUE_ISINF ... 385
REAL_VALUE_ISNAN ... 385
REAL_VALUE_NEGATE ... 385
REAL_VALUE_NEGATIVE ... 385
REAL_VALUE_TO_INT ... 385
REAL_VALUE_TO_TARGET_DOUBLE ... 357
REAL_VALUE_TO_TARGET_LONG_DOUBLE ... 357
REAL_VALUE_TO_TARGET_SINGLE ... 357
REAL_VALUE_TRUNCATE ... 385
REAL_VALUE_TYPE ... 384
REAL_VALUE_UNSIGNED_FIX ... 384
REAL_VALUES_EQUAL ... 384
REAL_VALUES_LESS ... 384
REALPART_EXPR ... 84
recog_data.operand ... 370
recognizing insns ... 181
RECORD_TYPE ... 63, 69
redirect_edge_and_branch ... 174
redirect_edge_and_branch, redirect_jump ... 176
reduc_smax_*m* instruction pattern ... 215
reduc_smin_*m* instruction pattern ... 215
reduc_splus_*m* instruction pattern ... 215

`reduc_umax_m` instruction pattern 215
`reduc_umin_m` instruction pattern 215
`reduc_uplus_m` instruction pattern 215
reference 63
`REFERENCE_TYPE` 63
`reg` 138
`reg` and '`/f`' 128
`reg` and '`/i`' 128
`reg` and '`/v`' 128
`reg`, RTL sharing 166
`REG_ALLOC_ORDER` 292
`REG_BR_PRED` 164
`REG_BR_PROB` 164
`REG_BR_PROB_BASE`, `BB_FREQ_BASE`, `count` 174
`REG_BR_PROB_BASE`, `EDGE_FREQUENCY` 174
`REG_CC_SETTER` 164
`REG_CC_USER` 164
`reg_class_contents` 291
`REG_CLASS_CONTENTS` 297
`REG_CLASS_FROM_CONSTRAINT` 298
`REG_CLASS_FROM_LETTER` 298
`REG_CLASS_NAMES` 297
`REG_CROSSING_JUMP` 162
`REG_DEAD` 161
`REG_DEAD`, `REG_UNUSED` 177
`REG_DEP_ANTI` 164
`REG_DEP_OUTPUT` 164
`REG_EH_REGION`, `EDGE_ABNORMAL_CALL` 172
`REG_EQUAL` 162
`REG_EQUIV` 162
`REG_EXPR` 126
`REG_FRAME_RELATED_EXPR` 164
`REG_FUNCTION_VALUE_P` 128
`REG_INC` 161
`REG_LABEL` 162
`reg_label` and '`/v`' 128
`REG_LIBCALL` 164
`REG_MODE_OK_FOR_BASE_P` 336
`REG_MODE_OK_FOR_REG_BASE_P` 336
`reg_names` 291, 371
`REG_NO_CONFLICT` 161
`REG_NONNEG` 161
`REG_NOTE_KIND` 161
`REG_NOTES` 160
`REG_OFFSET` 126
`REG_OK_FOR_BASE_P` 335
`REG_OK_FOR_INDEX_P` 336
`REG_OK_STRICT` 335
`REG_PARM_STACK_SPACE` 314
`REG_PARM_STACK_SPACE`, and `FUNCTION_ARG` ... 316
`REG_POINTER` 128
`REG_RETVAL` 163
`REG_SETJMP` 162
`REG_UNUSED` 161
`REG_USERVAR_P` 128
register allocation order 292
register class definitions 296
register class preference constraints 196
register pairs 293
Register Transfer Language (RTL) 121
register usage 290
`REGISTER_MOVE_COST` 341
`REGISTER_NAMES` 370
`register_operand` 188
`REGISTER_PREFIX` 372
`REGISTER_TARGET_PRAGMAS` 396
registers arguments 316
registers in constraints 192
`REGMODE_NATURAL_SIZE` 293
`REGNO_MODE_OK_FOR_BASE_P` 298
`REGNO_MODE_OK_FOR_REG_BASE_P` 298
`REGNO_OK_FOR_BASE_P` 298
`REGNO_OK_FOR_INDEX_P` 298
`REGNO_REG_CLASS` 297
`regs_ever_live` 323
regular expressions 256, 258
relative costs 341
`RELATIVE_PREFIX_NOT_LINKDIR` 272
reload pass 140
`reload_completed` 223
`reload_in` instruction pattern 213
`reload_in_progress` 212
`reload_out` instruction pattern 213
reloading 58
remainder 144
`reorder` 420
representation of RTL 121
reservation delays 256
`rest_of_decl_compilation` 49
`rest_of_type_compilation` 49
`restore_stack_block` instruction pattern 225
`restore_stack_function` instruction pattern 225
`restore_stack_nonlocal` instruction pattern 225
`RESULT_DECL` 71
`return` 150
`return` instruction pattern 223
return values in registers 320
`RETURN_ADDR_IN_PREVIOUS_FRAME` 306
`RETURN_ADDR_OFFSET` 308
`RETURN_ADDR_RTX` 305
`RETURN_ADDRESS_POINTER_REGNUM` 311
`RETURN_EXPR` 79
`RETURN_INIT` 79
`RETURN_POPS_ARGS` 315
`RETURN_STMT` 79
returning aggregate values 322
returning structures and unions 7
reverse probability 174
`REVERSE_CONDEXEC_PREDICATES_P` 340
`REVERSE_CONDITION` 340
`REVERSIBLE_CC_MODE` 340
right rotate 144
right shift 144
`rintm2` instruction pattern 217

RISC ... 256, 259
roots, marking ... 421
rotate ... 144
rotatert ... 144
rotlm3 instruction pattern ... 216
rotrm3 instruction pattern ... 216
Rough GIMPLE Grammar ... 104
ROUND_DIV_EXPR ... 84
ROUND_MOD_EXPR ... 84
ROUND_TOWARDS_ZERO ... 285
ROUND_TYPE_ALIGN ... 283
roundm2 instruction pattern ... 217
RSHIFT_EXPR ... 84
RTL addition ... 142
RTL addition with signed saturation ... 142
RTL addition with unsigned saturation ... 142
RTL classes ... 122
RTL comparison ... 142
RTL comparison operations ... 145
RTL constant expression types ... 136
RTL constants ... 136
RTL declarations ... 149
RTL difference ... 142
RTL expression ... 121
RTL expressions for arithmetic ... 142
RTL format ... 123
RTL format characters ... 123
RTL function-call insns ... 165
RTL insn template ... 181
RTL integers ... 121
RTL memory expressions ... 138
RTL object types ... 121
RTL postdecrement ... 154
RTL postincrement ... 154
RTL predecrement ... 154
RTL preincrement ... 154
RTL register expressions ... 138
RTL representation ... 121
RTL side effect expressions ... 149
RTL strings ... 121
RTL structure sharing assumptions ... 165
RTL subtraction ... 142
RTL subtraction with signed saturation ... 142
RTL subtraction with unsigned saturation ... 142
RTL sum ... 142
RTL vectors ... 121
RTX (See RTL) ... 121
RTX codes, classes of ... 122
RTX_FRAME_RELATED_P ... 128
run-time conventions ... 7
run-time target specification ... 275

S

's' in constraint ... 192
same_type_p ... 65
SAVE_EXPR ... 84
save_stack_block instruction pattern ... 225
save_stack_function instruction pattern ... 225
save_stack_nonlocal instruction pattern ... 225
scalars, returned as values ... 320
SCHED_GROUP_P ... 129
SCmode ... 134
scond instruction pattern ... 221
scratch ... 140
scratch operands ... 140
scratch, RTL sharing ... 166
scratch_operand ... 188
SDB_ALLOW_FORWARD_REFERENCES ... 383
SDB_ALLOW_UNKNOWN_REFERENCES ... 383
SDB_DEBUGGING_INFO ... 382
SDB_DELIM ... 383
SDB_OUTPUT_SOURCE_LINE ... 383
search options ... 244
SECONDARY_INPUT_RELOAD_CLASS ... 299
SECONDARY_MEMORY_NEEDED ... 300
SECONDARY_MEMORY_NEEDED_MODE ... 301
SECONDARY_MEMORY_NEEDED_RTX ... 300
SECONDARY_OUTPUT_RELOAD_CLASS ... 299
SECONDARY_RELOAD_CLASS ... 299
SELECT_CC_MODE ... 339
Selection Statements ... 101
sequence ... 153
set ... 150
set and '/f' ... 128
SET_ASM_OP ... 365
set_attr ... 252
set_attr_alternative ... 252
SET_DEST ... 150
SET_IS_RETURN_P ... 129
SET_LABEL_KIND ... 158
set_optab_libfunc ... 333
SET_SRC ... 150
setmemm instruction pattern ... 219
SETUP_FRAME_ADDRESSES ... 305
SFmode ... 133
sharing of RTL components ... 165
shift ... 144
SHIFT_COUNT_TRUNCATED ... 392
SHORT_IMMEDIATES_SIGN_EXTEND ... 392
SHORT_TYPE_SIZE ... 286
sibcall_epilogue instruction pattern ... 228
sibling call ... 172
SIBLING_CALL_P ... 129
sign_extend ... 148
sign_extract ... 147
sign_extract, canonicalization of ... 237
signed division ... 143
signed maximum ... 144
signed minimum ... 144
SImode ... 133
simple constraints ... 191
sinm2 instruction pattern ... 216
SIZE_ASM_OP ... 360
SIZE_TYPE ... 288
skip ... 418

SLOW_BYTE_ACCESS ... 342
SLOW_UNALIGNED_ACCESS ... 342
SMALL_ARG_MAX ... 408
SMALL_REGISTER_CLASSES ... 301
smax ... 144
smin ... 144
sms, swing, software pipelining ... 57
smulm3_highpart instruction pattern ... 215
soft float library ... 12
special ... 421
special predicates ... 187
SPECS ... 411
speed of instructions ... 341
split_block ... 176
splitting instructions ... 240
sqrt ... 145
sqrtm2 instruction pattern ... 216
square root ... 145
ss_minus ... 142
ss_plus ... 142
ss_truncate ... 148
SSA ... 112
SSA_NAME_DEF_STMT ... 115
SSA_NAME_VERSION ... 115
stack arguments ... 314
stack frame layout ... 304
stack smashing protection ... 328
STACK_ALIGNMENT_NEEDED ... 304
STACK_BOUNDARY ... 280
STACK_CHECK_BUILTIN ... 309
STACK_CHECK_FIXED_FRAME_SIZE ... 310
STACK_CHECK_MAX_FRAME_SIZE ... 310
STACK_CHECK_MAX_VAR_SIZE ... 310
STACK_CHECK_PROBE_INTERVAL ... 309
STACK_CHECK_PROBE_LOAD ... 310
STACK_CHECK_PROTECT ... 310
STACK_DYNAMIC_OFFSET ... 305
STACK_DYNAMIC_OFFSET and virtual registers ... 139
STACK_GROWS_DOWNWARD ... 304
STACK_PARMS_IN_REG_PARM_AREA ... 315
STACK_POINTER_OFFSET ... 304
STACK_POINTER_OFFSET and virtual registers ... 139
STACK_POINTER_REGNUM ... 310
STACK_POINTER_REGNUM and virtual registers ... 139
stack_pointer_rtx ... 311
stack_protect_set instruction pattern ... 231
stack_protect_test instruction pattern ... 231
STACK_PUSH_CODE ... 304
STACK_REGS ... 295
STACK_SAVEAREA_MODE ... 283
STACK_SIZE_MODE ... 284
standard pattern names ... 212
STANDARD_INCLUDE_COMPONENT ... 274
STANDARD_INCLUDE_DIR ... 274
STANDARD_STARTFILE_PREFIX ... 273
STANDARD_STARTFILE_PREFIX_1 ... 273
STANDARD_STARTFILE_PREFIX_2 ... 273
STARTFILE_SPEC ... 271
STARTING_FRAME_OFFSET ... 304
STARTING_FRAME_OFFSET and virtual registers ... 139
Statement Sequences ... 101
statements ... 79
Statements ... 100
Static profile estimation ... 173
static single assignment ... 112
STATIC_CHAIN ... 311
STATIC_CHAIN_INCOMING ... 311
STATIC_CHAIN_INCOMING_REGNUM ... 311
STATIC_CHAIN_REGNUM ... 311
'stdarg.h' and register arguments ... 316
STDC_0_IN_SYSTEM_HEADERS ... 395
STMT_EXPR ... 84
STMT_IS_FULL_EXPR_P ... 80
storage layout ... 278
STORE_BY_PIECES_P ... 343
STORE_FLAG_VALUE ... 393
'store_multiple' instruction pattern ... 214
strcpy ... 281
STRICT_ALIGNMENT ... 282
strict_low_part ... 149
strict_memory_address_p ... 337
STRING_CST ... 84
STRING_POOL_ADDRESS_P ... 129
strlenm instruction pattern ... 220
structure value address ... 322
STRUCTURE_SIZE_BOUNDARY ... 282
structures, returning ... 7
subm3 instruction pattern ... 214
SUBOBJECT ... 79
SUBOBJECT_CLEANUP ... 79
subreg ... 139
subreg and '/s' ... 130
subreg and '/u' ... 130
subreg and '/u' and '/v' ... 130
subreg, in strict_low_part ... 149
subreg, special reload handling ... 140
SUBREG_BYTE ... 140
SUBREG_PROMOTED_UNSIGNED_P ... 130
SUBREG_PROMOTED_UNSIGNED_SET ... 130
SUBREG_PROMOTED_VAR_P ... 130
SUBREG_REG ... 140
SUCCESS_EXIT_CODE ... 407
SUPPORTS_INIT_PRIORITY ... 369
SUPPORTS_ONE_ONLY ... 363
SUPPORTS_WEAK ... 362
SWITCH_BODY ... 79
SWITCH_COND ... 79
SWITCH_CURTAILS_COMPILATION ... 268
SWITCH_STMT ... 79
SWITCH_TAKES_ARG ... 267
SWITCHES_NEED_SPACES ... 268
SYMBOL_FLAG_EXTERNAL ... 126
SYMBOL_FLAG_FUNCTION ... 126
SYMBOL_FLAG_LOCAL ... 126
SYMBOL_FLAG_SMALL ... 126

SYMBOL_FLAG_TLS_SHIFT 127
symbol_ref 137
symbol_ref and '/f' 129
symbol_ref and '/i' 130
symbol_ref and '/u' 127
symbol_ref and '/v' 130
symbol_ref, RTL sharing 166
SYMBOL_REF_DECL 126
SYMBOL_REF_EXTERNAL_P 126
SYMBOL_REF_FLAG 130
SYMBOL_REF_FLAG, in TARGET_ENCODE_SECTION_INFO 352
SYMBOL_REF_FLAGS 126
SYMBOL_REF_FUNCTION_P 126
SYMBOL_REF_LOCAL_P 126
SYMBOL_REF_SMALL_P 126
SYMBOL_REF_TLS_MODEL 127
SYMBOL_REF_USED 130
SYMBOL_REF_WEAK 130
symbolic label 166
sync_add*mode* instruction pattern 230
sync_and*mode* instruction pattern 230
sync_compare_and_swap_cc*mode* instruction pattern 230
sync_compare_and_swap*mode* instruction pattern 229
sync_ior*mode* instruction pattern 230
sync_lock_release*mode* instruction pattern .. 231
sync_lock_test_and_set*mode* instruction pattern 230
sync_nand*mode* instruction pattern 230
sync_new_add*mode* instruction pattern 230
sync_new_and*mode* instruction pattern 230
sync_new_ior*mode* instruction pattern 230
sync_new_nand*mode* instruction pattern 230
sync_new_sub*mode* instruction pattern 230
sync_new_xor*mode* instruction pattern 230
sync_old_add*mode* instruction pattern 230
sync_old_and*mode* instruction pattern 230
sync_old_ior*mode* instruction pattern 230
sync_old_nand*mode* instruction pattern 230
sync_old_sub*mode* instruction pattern 230
sync_old_xor*mode* instruction pattern 230
sync_sub*mode* instruction pattern 230
sync_xor*mode* instruction pattern 230
SYSROOT_HEADERS_SUFFIX_SPEC 271
SYSROOT_SUFFIX_SPEC 271
SYSTEM_INCLUDE_DIR 274

T

't-*target*' 409
table jump 169
tablejump instruction pattern 224
tag 418
tagging insns 251
tail calls 327
target attributes 387
target description macros 267
target functions 267
target hooks 267
target makefile fragment 409
target specifications 275
TARGET_ADDRESS_COST 344
TARGET_ALIGN_ANON_BITFIELDS 283
TARGET_ALLOCATE_INITIAL_VALUE 401
TARGET_ARG_PARTIAL_BYTES 317
TARGET_ARM_EABI_UNWINDER 376
TARGET_ASM_ALIGNED_DI_OP 355
TARGET_ASM_ALIGNED_HI_OP 355
TARGET_ASM_ALIGNED_SI_OP 355
TARGET_ASM_ALIGNED_TI_OP 355
TARGET_ASM_ASSEMBLE_VISIBILITY 363
TARGET_ASM_BYTE_OP 355
TARGET_ASM_CAN_OUTPUT_MI_THUNK 327
TARGET_ASM_CLOSE_PAREN 357
TARGET_ASM_CONSTRUCTOR 369
TARGET_ASM_DESTRUCTOR 369
TARGET_ASM_EH_FRAME_SECTION 375
TARGET_ASM_EMIT_UNWIND_LABEL 374
TARGET_ASM_EXCEPTION_SECTION 375
TARGET_ASM_EXTERNAL_LIBCALL 363
TARGET_ASM_FILE_END 354
TARGET_ASM_FILE_START 353
TARGET_ASM_FILE_START_APP_OFF 354
TARGET_ASM_FILE_START_FILE_DIRECTIVE 354
TARGET_ASM_FUNCTION_BEGIN_EPILOGUE 324
TARGET_ASM_FUNCTION_END_PROLOGUE 324
TARGET_ASM_FUNCTION_EPILOGUE 324
TARGET_ASM_FUNCTION_EPILOGUE and trampolines 331
TARGET_ASM_FUNCTION_PROLOGUE 323
TARGET_ASM_FUNCTION_PROLOGUE and trampolines 331
TARGET_ASM_FUNCTION_RODATA_SECTION 351
TARGET_ASM_GLOBALIZE_LABEL 362
TARGET_ASM_INTEGER 356
TARGET_ASM_INTERNAL_LABEL 364
TARGET_ASM_MARK_DECL_PRESERVED 363
TARGET_ASM_NAMED_SECTION 355
TARGET_ASM_OPEN_PAREN 357
TARGET_ASM_OUTPUT_DWARF_DTPREL 383
TARGET_ASM_OUTPUT_MI_THUNK 326
TARGET_ASM_SELECT_RTX_SECTION 351
TARGET_ASM_SELECT_SECTION 351
TARGET_ASM_TTYPE 376
TARGET_ASM_UNALIGNED_DI_OP 355
TARGET_ASM_UNALIGNED_HI_OP 355
TARGET_ASM_UNALIGNED_SI_OP 355
TARGET_ASM_UNALIGNED_TI_OP 355
TARGET_ASM_UNIQUE_SECTION 351
TARGET_ATTRIBUTE_TABLE 387
TARGET_BINDS_LOCAL_P 352
TARGET_BRANCH_TARGET_REGISTER_CALLEE_SAVED 402
TARGET_BRANCH_TARGET_REGISTER_CLASS 402

TARGET_BUILD_BUILTIN_VA_LIST 320
TARGET_BUILTIN_SETJMP_FRAME_VALUE 305
TARGET_C99_FUNCTIONS 334
TARGET_CALLEE_COPIES 317
TARGET_CANNOT_FORCE_CONST_MEM 338
TARGET_CANNOT_MODIFY_JUMPS_P 402
TARGET_COMMUTATIVE_P 401
TARGET_COMP_TYPE_ATTRIBUTES 387
TARGET_CPU_CPP_BUILTINS 275
TARGET_CXX_ADJUST_CLASS_AT_DEFINITION ... 390
TARGET_CXX_CDTOR_RETURNS_THIS 390
TARGET_CXX_CLASS_DATA_ALWAYS_COMDAT 390
TARGET_CXX_COOKIE_HAS_SIZE 389
TARGET_CXX_DETERMINE_CLASS_DATA_VISIBILITY 390
TARGET_CXX_GET_COOKIE_SIZE 389
TARGET_CXX_GUARD_MASK_BIT 389
TARGET_CXX_GUARD_TYPE 389
TARGET_CXX_IMPORT_EXPORT_CLASS 390
TARGET_CXX_KEY_METHOD_MAY_BE_INLINE 390
TARGET_CXX_USE_AEABI_ATEXIT 390
TARGET_DECLSPEC 388
TARGET_DEFAULT_PACK_STRUCT 397
TARGET_DEFAULT_SHORT_ENUMS 288
TARGET_DEFERRED_OUTPUT_DEFS 365
TARGET_DELEGITIMIZE_ADDRESS 338
TARGET_DLLIMPORT_DECL_ATTRIBUTES 387
TARGET_DWARF_CALLING_CONVENTION 382
TARGET_DWARF_HANDLE_FRAME_UNSPEC 306
TARGET_DWARF_REGISTER_SPAN 375
TARGET_EDOM 333
TARGET_ENCODE_SECTION_INFO 352
TARGET_ENCODE_SECTION_INFO and address validation 335
TARGET_ENCODE_SECTION_INFO usage 371
TARGET_EXECUTABLE_SUFFIX 402
TARGET_EXPAND_BUILTIN 400
TARGET_EXPAND_BUILTIN_SAVEREGS 329
TARGET_EXPR 84
TARGET_EXTRA_INCLUDES 403
TARGET_EXTRA_PRE_INCLUDES 403
TARGET_FIXED_CONDITION_CODE_REGS 341
target_flags 276
TARGET_FLOAT_FORMAT 284
TARGET_FLT_EVAL_METHOD 288
TARGET_FOLD_BUILTIN 401
TARGET_FORMAT_TYPES 403
TARGET_FUNCTION_ATTRIBUTE_INLINABLE_P ... 388
TARGET_FUNCTION_OK_FOR_SIBCALL 327
TARGET_GIMPLIFY_VA_ARG_EXPR 320
TARGET_HANDLE_OPTION 276
TARGET_HAVE_CTORS_DTORS 369
TARGET_HAVE_NAMED_SECTIONS 355
TARGET_IN_SMALL_DATA_P 352
TARGET_INIT_BUILTINS 400
TARGET_INIT_LIBFUNCS 333
TARGET_INSERT_ATTRIBUTES 388
TARGET_INVALID_BINARY_OP 404
TARGET_INVALID_CONVERSION 404
TARGET_INVALID_UNARY_OP 404
TARGET_LIB_INT_CMP_BIASED 333
TARGET_MACHINE_DEPENDENT_REORG 400
TARGET_MANGLE_FUNDAMENTAL_TYPE 286
TARGET_MD_ASM_CLOBBERS 398
TARGET_MEM_REF 84
TARGET_MERGE_DECL_ATTRIBUTES 387
TARGET_MERGE_TYPE_ATTRIBUTES 387
TARGET_MIN_DIVISIONS_FOR_RECIP_MUL 392
TARGET_MS_BITFIELD_LAYOUT_P 285
TARGET_MUST_PASS_IN_STACK 317
TARGET_MUST_PASS_IN_STACK, and FUNCTION_ARG 316
TARGET_N_FORMAT_TYPES 403
TARGET_OBJECT_SUFFIX 401
TARGET_OBJFMT_CPP_BUILTINS 276
TARGET_OPTF 403
TARGET_OPTION_TRANSLATE_TABLE 268
TARGET_OS_CPP_BUILTINS 276
TARGET_PASS_BY_REFERENCE 317
TARGET_POSIX_IO 399
TARGET_PRETEND_OUTGOING_VARARGS_NAMED ... 330
TARGET_PROMOTE_FUNCTION_ARGS 280
TARGET_PROMOTE_FUNCTION_RETURN 280
TARGET_PROMOTE_PROTOTYPES 314
TARGET_PTRMEMFUNC_VBIT_LOCATION 289
TARGET_RELAXED_ORDERING 403
TARGET_RESOLVE_OVERLOADED_BUILTIN 400
TARGET_RETURN_IN_MEMORY 322
TARGET_RETURN_IN_MSB 321
TARGET_RTX_COSTS 344
TARGET_SCALAR_MODE_SUPPORTED_P 320
TARGET_SCHED_ADJUST_COST 345
TARGET_SCHED_ADJUST_PRIORITY 346
TARGET_SCHED_DEPENDENCIES_EVALUATION_HOOK 346
TARGET_SCHED_DFA_NEW_CYCLE 348
TARGET_SCHED_DFA_POST_CYCLE_INSN 347
TARGET_SCHED_DFA_PRE_CYCLE_INSN 347
TARGET_SCHED_FINISH 347
TARGET_SCHED_FINISH_GLOBAL 347
TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD 347
TARGET_SCHED_FIRST_CYCLE_MULTIPASS_DFA_LOOKAHEAD_GUARD 348
TARGET_SCHED_INIT 346
TARGET_SCHED_INIT_DFA_POST_CYCLE_INSN ... 347
TARGET_SCHED_INIT_DFA_PRE_CYCLE_INSN 347
TARGET_SCHED_INIT_GLOBAL 347
TARGET_SCHED_IS_COSTLY_DEPENDENCE 348
TARGET_SCHED_ISSUE_RATE 345
TARGET_SCHED_REORDER 346
TARGET_SCHED_REORDER2 346
TARGET_SCHED_VARIABLE_ISSUE 345
TARGET_SECTION_TYPE_FLAGS 355
TARGET_SET_DEFAULT_TYPE_ATTRIBUTES 387
TARGET_SETUP_INCOMING_VARARGS 329

TARGET_SHIFT_TRUNCATION_MASK ... 393
TARGET_SPLIT_COMPLEX_ARG ... 319
TARGET_STACK_PROTECT_FAIL ... 328
TARGET_STACK_PROTECT_GUARD ... 328
TARGET_STRICT_ARGUMENT_NAMING ... 330
TARGET_STRUCT_VALUE_RTX ... 322
TARGET_UNWIND_EMIT ... 374
TARGET_UNWIND_INFO ... 375
TARGET_USE_JCR_SECTION ... 404
TARGET_USE_LOCAL_THUNK_ALIAS_P ... 403
TARGET_USES_WEAK_UNWIND_INFO ... 309
TARGET_VALID_DLLIMPORT_ATTRIBUTE_P ... 388
TARGET_VALID_POINTER_MODE ... 320
TARGET_VECTOR_MODE_SUPPORTED_P ... 320
TARGET_VECTOR_OPAQUE_P ... 285
TARGET_VECTORIZE_BUILTIN_MASK_FOR_LOAD ... 338
TARGET_VERSION ... 276
TARGET_VTABLE_DATA_ENTRY_DISTANCE ... 290
TARGET_VTABLE_ENTRY_ALIGN ... 290
TARGET_VTABLE_USES_DESCRIPTORS ... 290
TARGET_WEAK_NOT_IN_ARCHIVE_TOC ... 363
targetm ... 267
targets, makefile ... 23
TCmode ... 134
TEMPLATE_DECL ... 71
Temporaries ... 98
termination routines ... 366
text_section ... 351
TEXT_SECTION_ASM_OP ... 349
TFmode ... 133
THEN_CLAUSE ... 79
THREAD_MODEL_SPEC ... 271
THROW_EXPR ... 84
THUNK_DECL ... 71
THUNK_DELTA ... 71
TImode ... 133
TImode, in insn ... 160
'tm.h' macros ... 267
TQFmode ... 133
TRAMPOLINE_ADJUST_ADDRESS ... 331
TRAMPOLINE_ALIGNMENT ... 331
TRAMPOLINE_SECTION ... 331
TRAMPOLINE_SIZE ... 331
TRAMPOLINE_TEMPLATE ... 331
trampolines for nested functions ... 330
TRANSFER_FROM_TRAMPOLINE ... 332
trap instruction pattern ... 229
tree ... 61, 62
Tree SSA ... 97
TREE_CODE ... 61
TREE_FILENAME ... 71
tree_int_cst_equal ... 84
TREE_INT_CST_HIGH ... 84
TREE_INT_CST_LOW ... 84
tree_int_cst_lt ... 84
TREE_LINENO ... 71
TREE_LIST ... 63
TREE_OPERAND ... 84
TREE_PUBLIC ... 76, 77
TREE_PURPOSE ... 63
TREE_STRING_LENGTH ... 84
TREE_STRING_POINTER ... 84
TREE_TYPE ... 63, 71, 79, 84
TREE_VALUE ... 63
TREE_VEC ... 63
TREE_VEC_ELT ... 63
TREE_VEC_LENGTH ... 63
Trees ... 61
TRULY_NOOP_TRUNCATION ... 393
TRUNC_DIV_EXPR ... 84
TRUNC_MOD_EXPR ... 84
truncate ... 148
trunc*mn*2 instruction pattern ... 220
TRUTH_AND_EXPR ... 84
TRUTH_ANDIF_EXPR ... 84
TRUTH_NOT_EXPR ... 84
TRUTH_OR_EXPR ... 84
TRUTH_ORIF_EXPR ... 84
TRUTH_XOR_EXPR ... 84
TRY_BLOCK ... 79
TRY_HANDLERS ... 79
TRY_STMTS ... 79
tst*m* instruction pattern ... 218
type ... 63
type declaration ... 71
TYPE_ALIGN ... 63, 64
TYPE_ARG_TYPES ... 63
TYPE_ASM_OP ... 360
TYPE_ATTRIBUTES ... 84
TYPE_BINFO ... 69
TYPE_BUILT_IN ... 65
TYPE_CONTEXT ... 63
TYPE_DECL ... 71
TYPE_FIELDS ... 63, 69
TYPE_HAS_ARRAY_NEW_OPERATOR ... 71
TYPE_HAS_DEFAULT_CONSTRUCTOR ... 70
TYPE_HAS_MUTABLE_P ... 71
TYPE_HAS_NEW_OPERATOR ... 71
TYPE_MAIN_VARIANT ... 63, 64
TYPE_MAX_VALUE ... 63
TYPE_METHOD_BASETYPE ... 63
TYPE_METHODS ... 69
TYPE_MIN_VALUE ... 63
TYPE_NAME ... 63, 64
TYPE_NOTHROW_P ... 79
TYPE_OFFSET_BASETYPE ... 63
TYPE_OPERAND_FMT ... 360
TYPE_OVERLOADS_ARRAY_REF ... 71
TYPE_OVERLOADS_ARROW ... 71
TYPE_OVERLOADS_CALL_EXPR ... 71
TYPE_POLYMORPHIC_P ... 70
TYPE_PRECISION ... 63
TYPE_PTR_P ... 65
TYPE_PTRFN_P ... 65
TYPE_PTRMEM_P ... 63, 65
TYPE_PTROB_P ... 65

TYPE_PTROBV_P ... 63
TYPE_QUAL_CONST ... 63
TYPE_QUAL_RESTRICT ... 63
TYPE_QUAL_VOLATILE ... 63
TYPE_RAISES_EXCEPTIONS ... 79
TYPE_SIZE ... 63, 64
TYPE_UNQUALIFIED ... 63
TYPE_VFIELD ... 69
TYPENAME_TYPE ... 63
TYPENAME_TYPE_FULLNAME ... 63
TYPEOF_TYPE ... 63

U

udiv ... 144
udiv*m*3 instruction pattern ... 214
udivmod*m*4 instruction pattern ... 216
UINTMAX_TYPE ... 289
umax ... 144
umax*m*3 instruction pattern ... 214
umin ... 144
umin*m*3 instruction pattern ... 214
umod ... 144
umod*m*3 instruction pattern ... 214
umulhisi3 instruction pattern ... 215
umul*m*3_highpart instruction pattern ... 215
umulqihi3 instruction pattern ... 215
umulsidi3 instruction pattern ... 215
unchanging ... 131
unchanging, in call_insn ... 127
unchanging, in jump_insn, call_insn and insn ... 127
unchanging, in mem ... 129
unchanging, in subreg ... 130
unchanging, in symbol_ref ... 127
UNEQ_EXPR ... 84
UNGE_EXPR ... 84
UNGT_EXPR ... 84
UNION_TYPE ... 63, 69
unions, returning ... 7
UNITS_PER_SIMD_WORD ... 279
UNITS_PER_WORD ... 279
UNKNOWN_TYPE ... 63
UNLE_EXPR ... 84
UNLIKELY_EXECUTED_TEXT_SECTION_NAME ... 349
UNLT_EXPR ... 84
UNORDERED_EXPR ... 84
unshare_all_rtl ... 166
unsigned division ... 144
unsigned greater than ... 146
unsigned less than ... 146
unsigned minimum and maximum ... 144
unsigned_fix ... 149
unsigned_float ... 149
unspec ... 154
unspec_volatile ... 154
untyped_call instruction pattern ... 223
untyped_return instruction pattern ... 224
UPDATE_PATH_HOST_CANONICALIZE (*path*) ... 406
update_ssa ... 113
update_stmt ... 107
US Software GOFAST, floating point emulation library ... 333
us_minus ... 142
us_plus ... 142
US_SOFTWARE_GOFAST ... 333
us_truncate ... 148
use ... 152
USE_C_ALLOCA ... 407
USE_LD_AS_NEEDED ... 270
USE_LOAD_POST_DECREMENT ... 343
USE_LOAD_POST_INCREMENT ... 343
USE_LOAD_PRE_DECREMENT ... 343
USE_LOAD_PRE_INCREMENT ... 343
use_param ... 419
use_param*n* ... 419
use_params ... 419
USE_SELECT_SECTION_FOR_FUNCTIONS ... 351
USE_STORE_POST_DECREMENT ... 343
USE_STORE_POST_INCREMENT ... 343
USE_STORE_PRE_DECREMENT ... 344
USE_STORE_PRE_INCREMENT ... 344
used ... 132
used, in symbol_ref ... 130
USER_LABEL_PREFIX ... 372
USING_DECL ... 71
USING_STMT ... 79

V

'V' in constraint ... 192
VA_ARG_EXPR ... 84
values, returned by functions ... 320
VAR_DECL ... 71, 84
varargs implementation ... 328
variable ... 71
VAX_FLOAT_FORMAT ... 284
vec_concat ... 148
vec_duplicate ... 148
vec_extract*m* instruction pattern ... 214
vec_init*m* instruction pattern ... 214
vec_merge ... 147
vec_select ... 147
vec_set*m* instruction pattern ... 214
vec_shl_*m* instruction pattern ... 215
vec_shr_*m* instruction pattern ... 215
vector ... 63
vector operations ... 147
VECTOR_CST ... 84
VECTOR_STORE_FLAG_VALUE ... 395
virtual operands ... 107
VIRTUAL_INCOMING_ARGS_REGNUM ... 139
VIRTUAL_OUTGOING_ARGS_REGNUM ... 139
VIRTUAL_STACK_DYNAMIC_REGNUM ... 139
VIRTUAL_STACK_VARS_REGNUM ... 139
VLIW ... 256, 259

VMS 406
VMS_DEBUGGING_INFO 384
VOID_TYPE 63
VOIDmode 134
volatil 132
volatil, in insn, call_insn, jump_insn, code_label, barrier, and note 127
volatil, in label_ref and reg_label 128
volatil, in mem, asm_operands, and asm_input 128
volatil, in reg 128
volatil, in subreg 130
volatil, in symbol_ref 130
volatile memory references 132
voting between constraint alternatives 196

W

walk_dominator_tree 115
walk_use_def_chains 115
WCHAR_TYPE 288
WCHAR_TYPE_SIZE 289
which_alternative 186
WHILE_BODY 79
WHILE_COND 79
WHILE_STMT 79
WIDEST_HARDWARE_FP_SIZE 288
WINT_TYPE 289
word_mode 136
WORD_REGISTER_OPERATIONS 391
WORD_SWITCH_TAKES_ARG 267
WORDS_BIG_ENDIAN 278
WORDS_BIG_ENDIAN, effect on subreg 140

X

'X' in constraint 193
'x-*host*' 411
XCmode 134
XCOFF_DEBUGGING_INFO 379
XEXP 124
XFmode 133
XINT 124
'xm-*machine*.h' 406, 407
xor 144
xor, canonicalization of 237
xor*m*3 instruction pattern 214
XSTR 124
XVEC 125
XVECEXP 125
XVECLEN 125
XWINT 124

Z

zero_extend 148
zero_extend*mn*2 instruction pattern 221
zero_extract 147
zero_extract, canonicalization of 237