Using and Porting GNU CC

Richard M. Stallman

last updated 15 February 1992

for version 2.0
(preliminary draft, which will change)

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

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

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

8. 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.

9. 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.
10. 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.
11. 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

12. 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 DE-

FECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. 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) 19*yy* *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., 675 Mass Ave, Cambridge, MA 02139, 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) 19*yy* *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.

Contributors to GNU CC

In addition to Richard Stallman, several people have written parts of GNU CC.

- The idea of using RTL and some of the optimization ideas came from the U. of Arizona Portable Optimizer, written by Jack Davidson and Christopher Fraser. See "Register Allocation and Exhaustive Peephole Optimization", Software Practice and Experience 14 (9), Sept. 1984, 857-866.
- Paul Rubin wrote most of the preprocessor.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the Vax machine description.
- Ted Lemon wrote parts of the RTL reader and printer.
- Jim Wilson implemented loop strength reduction and some other loop optimizations.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Michael Tiemann of Cygnus Support wrote the front end for C++, as well as the support for inline functions and instruction scheduling. Also the descriptions of the National Semiconductor 32000 series cpu, the SPARC cpu and part of the Motorola 88000 cpu.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.
- Randy Smith finished the Sun FPA support.
- Robert Brown implemented the support for Encore 32000 systems.
- David Kashtan of SRI adapted GNU CC to the Vomit-Making System (VMS).
- Alex Crain provided changes for the 3b1.
- Greg Satz and Chris Hanson assisted in making GNU CC work on HP-UX for the 9000 series 300.
- William Schelter did most of the work on the Intel 80386 support.
- Christopher Smith did the port for Convex machines.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Alain Lichnewsky ported GNU CC to the Mips cpu.
- Devon Bowen, Dale Wiles and Kevin Zachmann ported GNU CC to the Tahoe.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Richard Kenner of New York University wrote the machine descriptions for the AMD 29000, 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.

- Richard Kenner and Michael Tiemann jointly developed reorg.c, the delay slot scheduler.
- Mike Meissner and Tom Wood of Data General finished the port to the Motorola 88000.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- NeXT, Inc. donated the front end that supports the Objective C language.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Mike Meissner at the Open Software Foundation finished the port to the MIPS cpu, including adding ECOFF debug support.
- 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.

1 Protect Your Freedom—Fight "Look And Feel"

This section is a political message from the League for Programming Freedom to the users of GNU CC. It is included here as an expression of support for the League on the part of the Free Software Foundation.

Apple, Lotus and Xerox are trying to create a new form of legal monopoly: a copyright on a class of user interfaces. These monopolies would cause serious problems for users and developers of computer software and systems.

Until a few years ago, the law seemed clear: no one could restrict others from using a user interface; programmers were free to implement any interface they chose. Imitating interfaces, sometimes with changes, was standard practice in the computer field. The interfaces we know evolved gradually in this way; for example, the Macintosh user interface drew ideas from the Xerox interface, which in turn drew on work done at Stanford and SRI. 1-2-3 imitated VisiCalc, and dBase imitated a database program from JPL.

Most computer companies, and nearly all computer users, were happy with this state of affairs. The companies that are suing say it does not offer "enough incentive" to develop their products, but they must have considered it "enough" when they made their decision to do so. It seems they are not satisfied with the opportunity to continue to compete in the marketplace—not even with a head start.

If Xerox, Lotus, and Apple are permitted to make law through the courts, the precedent will hobble the software industry:

- Gratuitous incompatibilities will burden users. Imagine if each car manufacturer had to arrange the pedals in a different order.
- Software will become and remain more expensive. Users will be "locked in" to proprietary interfaces, for which there is no real competition.
- Large companies have an unfair advantage wherever lawsuits become commonplace. Since they can easily afford to sue, they can intimidate small companies with threats even when they don't really have a case.
- User interface improvements will come slower, since incremental evolution through creative imitation will no longer be permitted.
- Even Apple, etc., will find it harder to make improvements if they can no longer adapt the good ideas that others introduce, for fear of weakening their own legal positions. Some users suggest that this stagnation may already have started.

- If you use GNU software, you might find it of some concern that user interface copyright will make it hard for the Free Software Foundation to develop programs compatible with the interfaces that you already know.

To protect our freedom from lawsuits like these, a group of programmers and users have formed a new grass-roots political organization, the League for Programming Freedom.

The purpose of the League is to oppose new monopolistic practices such as user-interface copyright and software patents; it calls for a return to the legal policies of the recent past, in which these practices were not allowed. The League is not concerned with free software as an issue, and not affiliated with the Free Software Foundation.

The League's membership rolls include John McCarthy, inventor of Lisp, Marvin Minsky, founder of the Artificial Intelligence lab, Guy L. Steele, Jr., author of well-known books on Lisp and C, as well as Richard Stallman, the developer of GNU CC. Please join and add your name to the list. Membership dues in the League are $42 per year for programmers, managers and professionals; $10.50 for students; $21 for others.

The League needs both activist members and members who only pay their dues.

To join, or for more information, phone (617) 492-0023 or write to:

League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139

You can also send electronic mail to `league@prep.ai.mit.edu`.

Here are some suggestions from the League for things you can do to protect your freedom to write programs:

- Don't buy from Xerox, Lotus or Apple. Buy from their competitors or from the defendants they are suing.
- Don't develop software to work with the systems made by these companies.
- Port your existing software to competing systems, so that you encourage users to switch.
- Write letters to company presidents to let them know their conduct is unacceptable.
- Tell your friends and colleagues about this issue and how it threatens to ruin the computer industry.

- Above all, don't work for the look-and-feel plaintiffs, and don't accept contracts from them.
- Write to Congress to explain the importance of this issue.

  House Subcommittee on Intellectual Property
  2137 Rayburn Bldg
  Washington, DC 20515

  Senate Subcommittee on Patents, Trademarks and Copyrights
  United States Senate
  Washington, DC 20510

  (These committees have received lots of mail already; let's give them even more.)

Express your opinion! You can make a difference.

2 GNU CC Command Options

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the '`-c`' option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

The GNU C compiler uses a command syntax much like the Unix C compiler. The `gcc` program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: '`-dr`' is very different from '`-d -r`'.

You can mix options and other arguments. For the most part, the order you use doesn't matter; `gcc` reorders the command-line options so that the choices specified by option flags are applied to all input files. Order does matter when you use several options of the same kind; for example, if you specify '`-L`' more than once, the directories are searched in the order specified.

Many options have long names starting with '`-f`' or with '`-W`'—for example, '`-fforce-mem`', '`-fstrength-reduce`', '`-Wformat`' and so on. Most of these have both positive and negative forms; the negative form of '`-ffoo`' would be '`-fno-foo`'. This manual documents only one of these two forms, whichever one is not the default.

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

Overall Options

See Section 2.1 [Options Controlling the Kind of Output], page 18.

`-c` `-S` `-E` `-o` *file* `-pipe` `-v` `-x` *language*

Language Options

See Section 2.2 [Options Controlling Dialect], page 19.

```
-ansi  -fbuiltin  -fcond-mismatch  -fno-asm
-fsigned-bitfields  -fsigned-char
-funsigned-bitfields  -funsigned-char  -fwritable-strings
-traditional  -traditional-cpp  -trigraphs
```

Warning Options

See Section 2.3 [Options to Request or Suppress Warnings], page 22.

`-fsyntax-only` `-pedantic` `-pedantic-errors`
`-w` `-W` `-Wall` `-Waggregate-return`
`-Wcast-align` `-Wcast-qual` `-Wcomment` `-Wconversion` `-Werror`
`-Wformat` `-Wid-clash-`*len* `-Wimplicit` `-Wmissing-prototypes`
`-Wno-parentheses` `-Wpointer-arith` `-Wreturn-type` `-Wshadow`
`-Wstrict-prototypes` `-Wswitch` `-Wtraditional` `-Wtrigraphs`
`-Wuninitialized` `-Wunused` `-Wwrite-strings` `-Wchar-subscripts`

Debugging Options

See Section 2.4 [Options for Debugging Your Program or GCC], page 27.

`-a` `-d`*letters* `-fpretend-float`
`-g` `-ggdb` `-gdwarf` `-gstabs` `-gstabs+` `-gcoff`
`-p` `-pg` `-save-temps`

Optimization Options

See Section 2.5 [Options that Control Optimization], page 29.

`-fcaller-saves` `-fcse-follow-jumps` `-fdelayed-branch`
`-fexpensive-optimizations` `-ffloat-store` `-fforce-addr` `-fforce-mem`
`-finline` `-finline-functions` `-fkeep-inline-functions`
`-fno-defer-pop` `-fno-function-cse` `-fomit-frame-pointer`
`-frerun-cse-after-loop` `-fschedule-insns` `-fschedule-insns2`
`-fstrength-reduce` `-fthread-jumps`
`-funroll-all-loops` `-funroll-loops`
`-O` `-O2`

Preprocessor Options

See Section 2.6 [Options Controlling the Preprocessor], page 33.

`-C` `-dD` `-dM` `-dN`
`-D`*macro*[`=`*defn*] `-E` `-H`
`-include` *file* `-imacros` *file*
`-M` `-MD` `-MM` `-MMD` `-nostdinc` `-P` `-trigraphs` `-U`*macro*

Linker Options

See Section 2.7 [Options for Linking], page 34.

object-file-name
`-l`*library* `-nostdlib` `-static`

Directory Options

See Section 2.8 [Options for Directory Search], page 36.

`-B`*prefix* `-I`*dir* `-I-` `-L`*dir*

Target Options

See Section 2.9 [Target Machine and Compiler Version], page 37.

`-b` *machine* `-V` *version*

Machine Dependent Options

See Section 2.10 [Hardware Models and Configurations], page 38.

M680x0 Options
`-m68000 -m68020 -m68881 -mbitfield -mc68000 -mc68020 -mfpa`
`-mnobitfield -mrtd -mshort -msoft-float`

VAX Options
`-mg -mgnu -munix`

SPARC Options
`-mfpu -mno-epilogue`

Convex Options
`-margcount -mc1 -mc2 -mnoargcount`

AMD29K Options
`-m29000 -m29050 -mbw -mdw -mkernel-registers -mlarge`
`-mnbw -mnodw -msmall -mstack-check -muser-registers`

M88K Options
`-m88000 -m88100 -m88110 -mbig-pic -mcheck-zero-division`
`-mhandle-large-shift -midentify-revision`
`-mno-check-zero-division -mno-ocs-debug-info`
`-mno-ocs-frame-position -mno-optimize-arg-area -mno-underscores`
`-mocs-debug-info -mocs-frame-position -moptimize-arg-area`
`-mshort-data-`*num* `-msvr3 -msvr4 -mtrap-large-shift`
`-muse-div-instruction -mversion-03.00 -mwarn-passed-structs`

RS/6000 Options
`-mfp-in-toc -mno-fop-in-toc`

RT Options
`-mcall-lib-mul -mfp-arg-in-fpregs -mfp-arg-in-gregs`
`-mfull-fp-blocks -mhc-struct-return -min-line-mul`
`-mminimum-fp-blocks -mnohc-struct-return`

MIPS Options
`-mcpu=`*cpu type* `-mips2 -mips3 -mint64 -mlong64 -mlonglong128`
`-mmips-as -mgas -mrnames -mno-rnames -mgpopt -mno-gpopt -mstats`
`-mno-stats -mmemcpy -mno-memcpy -mno-mips-tfile -mmips-tfile`
`-msoft-float -mhard-float -mabicalls -mno-abicalls -mhalf-pic`
`-mno-half-pic -G` *num*

Code Generation Options

See Section 2.11 [Options for Code Generation Conventions], page 47.

`-fcall-saved-`*reg* `-fcall-used-`*reg* `-ffixed-`*reg*
`-fno-common -fpcc-struct-return -fpic -fPIC -fshared-data`
`-fshort-enums -fshort-double -fvolatile`

2.1 Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

file`.c` C source code which must be preprocessed.

file`.i` C source code which should not be preprocessed.

file`.m` Objective-C source code

file`.h` C header file (not to be compiled or linked).

file`.cc`
file`.cxx`
file`.C` C++ source code which must be preprocessed.

file`.s` Assembler code.

file`.S` Assembler code which must be preprocessed.

other An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

You can specify the input language explicitly with the '`-x`' option:

`-x` *language*

Specify explicitly the *language* for the following input files (rather than choosing a default based on the file name suffix). This option applies to all following input files until the next '`-x`' option. Possible values of *language* are '`c`', '`objective-c`', '`c-header`', '`c++`', '`cpp-output`', '`assembler`', and '`assembler-with-cpp`'.

`-x none` Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if '`-x`' has not been used at all).

If you only want some of the stages of compilation, you can use '`-x`' (or filename suffixes) to tell `gcc` where to start, and one of the options '`-c`', '`-S`', or '`-E`' to say where `gcc` is to stop. Note that some combinations (for example, '`-x cpp-output -E`' instruct `gcc` to do nothing at all.

`-c` Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix '`.c`', '`.i`', '`.s`', etc., with '`.o`'.

Unrecognized input files, not requiring compilation or assembly, are ignored.

`-S` Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix '`.c`', '`.i`', etc., with '`.s`'.

Input files that don't require compilation are ignored.

`-E` Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files which don't require preprocessing are ignored.

`-o` *file* Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

Since only one output file can be specified, it does not make sense to use '`-o`' when compiling more than one input file, unless you are producing an executable file as output.

If '`-o`' is not specified, the default is to put an executable file in '`a.out`', the object file for '*source*`.`*suffix*' in '*source*`.o`', its assembler file in '*source*`.s`', and all preprocessed C source on standard output.

`-v` Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

`-pipe` Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

2.2 Options Controlling Dialect

The following options control the dialect of C that the compiler accepts:

`-ansi` Support all ANSI standard C programs.

This turns off certain features of GNU C that are incompatible with ANSI C, such as the `asm`, `inline` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature, and disallows '`$`' as part of identifiers.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite '`-ansi`'. You would not want to use them in an ANSI C program, of course, but it useful to put them in header files that might be included in compilations done with '`-ansi`'. Alternate predefined macros such as `__unix__` and `__vax__` are also available, with or without '`-ansi`'.

The '`-ansi`' option does not cause non-ANSI programs to be rejected gratuitously. For that, '`-pedantic`' is required in addition to '`-ansi`'. See Section 2.3 [Warning Options], page 22.

The macro `__STRICT_ANSI__` is predefined when the '`-ansi`' option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

`-fno-asm` Do not recognize `asm`, `inline` or `typeof` as a keyword. These words may then be used as identifiers. You can use `__asm__`, `__inline__` and `__typeof__` instead. '`-ansi`' implies '`-fno-asm`'.

`-fno-builtin`

Don't recognize non-ANSI built-in functions. '`-ansi`' also has this effect. Currently, the only function affected is `alloca`.

`-trigraphs`

Support ANSI C trigraphs. You don't want to know about this brain-damage. The '`-ansi`' option implies '`-trigraphs`'.

`-traditional`

Attempt to support some aspects of traditional C compilers. Specifically:

- All `extern` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
- The keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized. (You can still use the alternative keywords such as `__typeof__`, `__inline__`, and so on.)
- Comparisons between pointers and integers are always allowed.
- Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.
- Out-of-range floating point literals are not an error.
- String "constants" are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of '`-fwritable-strings`'.)
- All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.

- In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.
- The predefined macro `__STDC__` is not defined when you use '`-traditional`', but `__GNUC__` is (since the GNU extensions which `__GNUC__` indicates are not affected by '`-traditional`'). If you need to write header files that work differently depending on whether '`-traditional`' is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.

`-traditional-cpp`

Attempt to support some aspects of traditional C preprocessors. This includes the last three items in the table immediately above, but none of the other effects of '`-traditional`'.

`-fcond-mismatch`

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

`-funsigned-char`

Let the type `char` be unsigned, like `unsigned char`.

Each kind of machine has a default for what `char` should be. It is either like `unsigned char` by default or like `signed char` by default.

Ideally, a portable program should always use `signed char` or `unsigned char` when it depends on the signedness of an object. But many programs have been written to use plain `char` and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type `char` is always a distinct type from each of `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

`-fsigned-char`

Let the type `char` be signed, like `signed char`.

Note that this is equivalent to '`-fno-unsigned-char`', which is the negative form of '`-funsigned-char`'. Likewise, '`-fno-signed-char`' is equivalent to '`-funsigned-char`'.

`-fsigned-bitfields`
`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`

These options control whether a bitfield is signed or unsigned, when the declaration does not use either `signed` or `unsigned`. By default, such a bitfield is signed, because this is consistent: the basic integer types such as `int` are signed types.

However, when '`-traditional`' is used, bitfields are all unsigned no matter what.

`-fwritable-strings`

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. '`-traditional`' also has this effect.

Writing into string constants is a very bad idea; "constants" should be constant.

2.3 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning '`-W`', for example '`-Wimplicit`' to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning '`-Wno-`' to turn off warnings; for example, '`-Wno-implicit`'. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by GNU CC:

`-fsyntax-only`

Check the code for syntax errors, but don't emit any output.

`-w` Inhibit all warning messages.

`-pedantic`

Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require '`-ansi`'). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

'`-pedantic`' does not cause warning messages for use of the alternate keywords whose names begin and end with '`__`'. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them. See Section 7.28 [Alternate Keywords], page 102.

This option is not intended to be *useful*; it exists only to satisfy pedants who would otherwise claim that GNU CC fails to support the ANSI standard.

Some users try to use '`-pedantic`' to check programs for strict ANSI C conformance. They soon find that it does not do quite what they want: it finds some non-ANSI practices, but not all—only those for which ANSI C *requires* a diagnostic.

A feature to report any failure to conform to ANSI C might be useful in some instances, but would require considerable additional work and would be quite different from '`-pedantic`'. We recommend, rather, that users take advantage of the extensions of GNU C and disregard the limitations of other compilers. Aside from certain supercomputers and obsolete small machines, there is less and less reason ever to use any other C compiler other than for bootstrapping GNU CC.

`-pedantic-errors`

Like '`-pedantic`', except that errors are produced rather than warnings.

`-W` Print extra warning messages for these events:

- A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation.

  The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
  if (a > 0)
    return a;
}
```

- An expression-statement contains no side effects.
- An unsigned value is compared against zero with '`>`' or '`<=`'.

`-Wimplicit`

Warn whenever a function or parameter is implicitly declared.

`-Wreturn-type`

Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

`-Wunused` Warn whenever a local variable is unused aside from its declaration, whenever a function is declared static but never defined, and whenever a statement computes a result that is explicitly not used.

`-Wswitch` Warn whenever a `switch` statement has an index of enumeral type and lacks a `case` for one or more of the named codes of that enumeration. (The presence of a `default` label prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used.

`-Wcomment`

Warn whenever a comment-start sequence '`/*`' appears in a comment.

`-Wtrigraphs`

Warn if any trigraphs are encountered (assuming they are enabled).

`-Wformat` Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.

`-Wchar-subscripts`

Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

`-Wuninitialized`

An automatic variable is used without first being initialized.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify '`-O`', you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
```

```
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only if it is set.

Some spurious warnings can be avoided if you declare as `volatile` all the functions you use that never return. See Section 7.19 [Function Attributes], page 89.

`-Wall` All of the above '`-W`' options combined. These are all the options which pertain to usage that we recommend avoiding and that we believe is easy to avoid, even in conjunction with macros.

The remaining '`-W...`' options are not implied by '`-Wall`' because they warn about constructions that we consider reasonable to use, on occasion, in clean programs.

`-Wtraditional`

Warn about certain constructs that behave differently in traditional and ANSI C.

- Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.
- A function declared external in one block and then used after the end of the block.
- A `switch` statement has an operand of type `long`.

`-Wshadow` Warn whenever a local variable shadows another local variable.

`-Wid-clash-`*len*

Warn whenever two distinct identifiers match in the first *len* characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

`-Wpointer-arith`

Warn about anything that depends on the "size of" a function type or of `void`. GNU C assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.

`-Wcast-qual`

Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

`-Wcast-align`

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

`-Wwrite-strings`

Give string constants the type `const char[`*length*`]` so that copying the address of one into a non-`const char *` pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make '`-Wall`' request these warnings.

`-Wconversion`

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

`-Waggregate-return`

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

`-Wstrict-prototypes`

Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

`-Wmissing-prototypes`

Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

`-Wredundant-decls`

Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

`-Wnested-externs`

Warn if an `extern` declaration is encountered within an function.

`-Wno-parentheses`

Disable warnings that parentheses are suggested around an expression.

`-Werror` Make all warnings into errors.

2.4 Options for Debugging Your Program or GNU CC

GNU CC has various special options that are used for debugging either your program or GCC:

`-g` Produce debugging information in the operating system's native format (stabs or COFF or DWARF). GDB can work with this debugging information.

On most systems that use stabs format, '`-g`' enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but will probably make DBX crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use '`-gstabs+`' or '`-gstabs`' (see below).

Unlike most other C compilers, GNU CC allows you to use '`-g`' with '`-O`'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when GNU CC is generated with the capability for more than one debugging format.

`-ggdb` Produce debugging information in the native format (if that is supported), including GDB extensions if at all possible.

`-gstabs` Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems.

`-gstabs+` Produce debugging information in stabs format (if that is supported), using GDB extensions. The use of these extensions is likely to make DBX crash or refuse to read the program.

`-gcoff` Produce debugging information in COFF format (if that is supported). This is the format used by SDB on COFF systems.

`-gdwarf` Produce debugging information in DWARF format (if that is supported). This is the format used by SDB on systems that use DWARF.

`-g`*level*
`-ggdb`*level*
`-gstabs`*level*
`-gcoff`*level*
`-gdwarf`*level*

Request debugging information and also use *level* to specify how much information. The default level is 2.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, but no information about local variables and no line numbers.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use '`-g3`'.

`-p` Generate extra code to write profile information suitable for the analysis program `prof`.

`-pg` Generate extra code to write profile information suitable for the analysis program `gprof`.

`-a` Generate extra code to write profile information for basic blocks, which will record the number of times each basic block is executed. This data could be analyzed by a program like `tcov`. Note, however, that the format of the data is not what `tcov` expects. Eventually GNU `gprof` should be extended to process this data.

`-d`*letters* Says to make debugging dumps during compilation at times specified by *letters*. This is used for debugging the compiler. The file names for most of the dumps are made by appending a word to the source file name (e.g. '`foo.c.rtl`' or '`foo.c.jump`'). Here are the possible letters for use in *letters*, and their meanings:

- '`M`' Dump all macro definitions, at the end of preprocessing, and write no output.
- '`N`' Dump all macro names, at the end of preprocessing.
- '`D`' Dump all macro definitions, at the end of preprocessing, in addition to normal output.
- '`y`' Dump debugging information during parsing, to standard error.
- '`r`' Dump after RTL generation, to '*file*`.rtl`'.
- '`x`' Just generate RTL for a function instead of compiling it. Usually used with '`r`'.
- '`j`' Dump after first jump optimization, to '*file*`.jump`'.
- '`s`' Dump after CSE (including the jump optimization that sometimes follows CSE), to '*file*`.cse`'.
- '`L`' Dump after loop optimization, to '*file*`.loop`'.
- '`t`' Dump after the second CSE pass (including the jump optimization that sometimes follows CSE), to '*file*`.cse2`'.

‘`f`’ Dump after flow analysis, to ‘*file*`.flow`’.

‘`c`’ Dump after instruction combination, to ‘*file*`.combine`’.

‘`S`’ Dump after the first instruction scheduling pass, to ‘*file*`.sched`’.

‘`l`’ Dump after local register allocation, to ‘*file*`.lreg`’.

‘`g`’ Dump after global register allocation, to ‘*file*`.greg`’.

‘`R`’ Dump after the second instruction scheduling pass, to ‘*file*`.sched2`’.

‘`J`’ Dump after last jump optimization, to ‘*file*`.jump2`’.

‘`d`’ Dump after delayed branch scheduling, to ‘*file*`.dbr`’.

‘`k`’ Dump after conversion from registers to stack, to ‘*file*`.stack`’.

‘`a`’ Produce all the dumps listed above.

‘`m`’ Print statistics on memory usage, at the end of the run, to standard error.

‘`p`’ Annotate the assembler output with a comment indicating which pattern and alternative was used.

`-fpretend-float`

When running a cross-compiler, pretend that the target machine uses the same floating point format as the host machine. This causes incorrect output of the actual floating constants, but the actual instruction sequence will probably be the same as GNU CC would make when running on the target machine.

`-save-temps`

Store the usual “temporary” intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling ‘`foo.c`’ with ‘`-c -save-temps`’ would produce files ‘`foo.cpp`’ and ‘`foo.s`’, as well as ‘`foo.o`’.

2.5 Options That Control Optimization

These options control various sorts of optimizations:

`-O` Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without ‘`-O`’, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without ‘`-O`’, only variables declared `register` are allocated in registers. The resulting compiled code is a little worse than produced by PCC without ‘`-O`’.

With '`-O`', the compiler tries to reduce code size and execution time.

When '`-O`' is specified, '`-fthread-jumps`' and '`-fdelayed-branch`' are turned on. On some machines other flags may also be turned on.

`-O2` Highly optimize. All supported optimizations that do not involve a space-speed tradeoff are performed. As compared to '`-O`', this option will increase both compilation time and the performance of the generated code.

All '`-f`*flag*' options that control optimization are turned on when '`-O2`' is specified, except for '`-funroll-loops`' and '`-funroll-all-loops`'.

Options of the form '`-f`*flag*' specify machine-independent flags. Most flags have both positive and negative forms; the negative form of '`-ffoo`' would be '`-fno-foo`'. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing '`no-`' or adding it.

`-ffloat-store`

Do not store floating point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have.

For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '`-ffloat-store`' for such programs.

`-fno-defer-pop`

Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

`-fforce-mem`

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

`-fforce-addr`

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as '`-fforce-mem`' may. I am interested in hearing about the difference this makes.

`-fomit-frame-pointer`

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an

extra register available in many functions. **It also makes debugging impossible on some machines.**

On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The machine-description macro `FRAME_POINTER_REQUIRED` controls whether a target machine supports this flag. See Section 15.5 [Registers], page 229.

`-finline` Pay attention to the `inline` keyword. Normally the negation of this option '`-fno-inline`' is used to keep the compiler from expanding any functions inline. However, the opposite effect may be desirable when compiling without optimization, since inline expansion is turned off in that case.

`-finline-functions`

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

`-fcaller-saves`

Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

This option is enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

`-fkeep-inline-functions`

Even if all calls to a given function are integrated, and the function is declared `static`, nevertheless output a separate run-time callable version of the function.

`-fno-function-cse`

Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

The following options control specific optimizations. The '`-O2`' option turns on all of these optimizations except '`-funroll-loops`' and '`-funroll-all-loops`'. The '`-O`' option usually turns on the '`-fthread-jumps`' and '`-fdelayed-branch`' options, but specific machines may change the default optimizations.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

`-fstrength-reduce`

Perform the optimizations of loop strength reduction and elimination of iteration variables.

`-fthread-jumps`

Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

`-fcse-follow-jumps`

In common subexpression elimination, scan through jump instructions in certain cases. This is not as powerful as completely global CSE, but not as slow either.

`-frerun-cse-after-loop`

Re-run common subexpression elimination after loop optimizations has been performed.

`-fexpensive-optimizations`

Perform a number of minor optimizations that are relatively expensive.

`-fdelayed-branch`

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

`-fschedule-insns`

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

`-fschedule-insns2`

Similar to '`-fschedule-insns`', but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

`-funroll-loops`

Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. '`-funroll-loop`' implies '`-fstrength-reduce`' and '`-frerun-cse-after-loop`'.

`-funroll-all-loops`

Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. '`-funroll-all-loops`' implies '`-fstrength-reduce`' and '`-frerun-cse-after-loop`'.

`-fno-peephole`

Disable any machine-specific peephole optimizations.

2.6 Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the '`-E`' option, nothing is done except preprocessing. Some of these options make sense only together with '`-E`' because they cause the preprocessor output to be unsuitable for actual compilation.

`-include` *file*

Process *file* as input before processing the regular input file. In effect, the contents of *file* are compiled first. Any '`-D`' and '`-U`' options on the command line are always processed before '`-include` *file*', regardless of the order in which they are written. All the '`-include`' and '`-imacros`' options are processed in the order in which they are written.

`-imacros` *file*

Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of '`-imacros` *file*' is to make the macros defined in *file* available for use in the main input.

Any '`-D`' and '`-U`' options on the command line are always processed before '`-imacros` *file*', regardless of the order in which they are written. All the '`-include`' and '`-imacros`' options are processed in the order in which they are written.

`-nostdinc`

Do not search the standard system directories for header files. Only the directories you have specified with '`-I`' options (and the current directory, if appropriate) are searched. See Section 2.8 [Directory Options], page 36, for information on '`-I`'.

By using both '`-nostdinc`' and '`-I-`', you can limit the include-file search path to only those directories you specify explicitly.

`-undef` Do not predefine any nonstandard macros. (Including architecture flags).

`-E` Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.

`-C` Tell the preprocessor not to discard comments. Used with the '`-E`' option.

`-P` Tell the preprocessor not to generate '`#line`' commands. Used with the '`-E`' option.

`-M` Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each object file. For each source file, the preprocessor outputs one `make`-rule whose target is the object file name for that source file and whose dependencies are all the files '`#include`'d in it. This rule may be a single line or may be continued with '\'-newline

if it is long. The list of rules is printed on standard output instead of the preprocessed C program.

'`-M`' implies '`-E`'.

Another way to specify output of a `make` rule is by setting the environment variable `DEPENDENCIES_OUTPUT` (see Section 2.12 [Environment Variables], page 49).

`-MM` Like '`-M`' but the output mentions only the user header files included with '`#include "`*file*`"`'. System header files included with '`#include <`*file*`>`' are omitted.

`-MD` Like '`-M`' but the dependency information is written to files with names made by replacing '`.c`' with '`.d`' at the end of the input file names. This is in addition to compiling the file as specified—'`-MD`' does not inhibit ordinary compilation the way '`-M`' does.

The Mach utility '`md`' can be used to merge the '`.d`' files into a single dependency file suitable for using with the '`make`' command.

`-MMD` Like '`-MD`' except mention only user header files, not system header files.

`-H` Print the name of each header file used, in addition to other normal activities.

`-D`*macro* Define macro *macro* with the string '`1`' as its definition.

`-D`*macro*`=`*defn*

Define macro *macro* as *defn*. All instances of '`-D`' on the command line are processed before any '`-U`' options.

`-U`*macro* Undefine macro *macro*. '`-U`' options are evaluated after all '`-D`' options, but before any '`-include`' and '`-imacros`' options.

`-dM` Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the '`-E`' option.

`-dD` Tell the preprocessing to pass all macro definitions into the output, in their proper sequence in the rest of the output.

`-dN` Like '`-dD`' except that the macro arguments and contents are omitted. Only '`#define` *name*' is included in the output.

`-trigraphs`

Support ANSI C trigraphs. You don't want to know about this brain-damage. The '`-ansi`' option also has this effect.

2.7 Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

object-file-name

A file name that does not end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

`-c`
`-S`
`-E` If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See Section 2.1 [Overall Options], page 18.

`-l`*library* Search the library named *library* when linking.

It makes a difference where in the command you write this option; the linker searches processes libraries and object files in the order they are specified. Thus, '`foo.o -lz bar.o`' seaches library '`z`' after file '`foo.o`' but before '`bar.o`'. If '`bar.o`' refers to functions in '`z`', those functions may not be loaded.

The linker searches a standard list of directories for the library, which is actually a file named '`lib`*library*`.a`'. The linker then uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with '`-L`'.

Normally the files found this way are library files—archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an '`-l`' option and specifying a file name is that '`-l`' surrounds *library* with '`lib`' and '`.a`' and searches several directories.

`-nostdlib`

Don't use the standard system libraries and startup files when linking. Only the files you specify will be passed to the linker.

`-static` On systems that support dynamic linking, this prevents linking with the shared libraries. On other systems, this option has no effect.

`-dynamic` On systems that support dynamic linking, you can use this option to request it explicitly.

`-shared` Produce a shared object which can then be linked with other objects to form an executable. Only a few systems support this option.

`-symbolic`

Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option '`-Xlinker -z -Xlinker defs`'). Only a few systems support this option.

`-Xlinker` *option*

Pass *option* as an option to the linker. You can use this to supply system-specific linker options which GNU CC does not know how to recognize.

If you want to pass an option that takes an argument, you must use '`-Xlinker`' twice, once for the option and once for the argument. For example, to pass '`-assert definitions`', you must write '`-Xlinker -assert -Xlinker definitions`'. It does not work to write '`-Xlinker "-assert definitions"`', because this passes the entire string as a single argument, which is not what the linker expects.

2.8 Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

`-I`*dir* Append directory *dir* to the list of directories searched for include files.

`-I-` Any directories you specify with '`-I`' options before the '`-I-`' option are searched only for the case of '`#include "`*file*`"`'; they are not searched for '`#include <`*file*`>`'.

If additional directories are specified with '`-I`' options after the '`-I-`', these directories are searched for all '`#include`' directives. (Ordinarily *all* '`-I`' directories are used this way.)

In addition, the '`-I-`' option inhibits the use of the current directory (where the current input file came from) as the first search directory for '`#include "`*file*`"`'. There is no way to override this effect of '`-I-`'. With '`-I.`' you can specify searching the directory which was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory.

'`-I-`' does not inhibit the use of the standard system directories for header files. Thus, '`-I-`' and '`-nostdinc`' are independent.

`-L`*dir* Add directory *dir* to the list of directories to be searched for '`-l`'.

`-B`*prefix* This option specifies where to find the executables, libraries and data files of the compiler itself.

The compiler driver program runs one or more of the subprograms '`cpp`', '`cc1`', '`as`' and '`ld`'. It tries *prefix* as a prefix for each program it tries to run, both with and without '*machine*`/`*version*`/`' (see Section 2.9 [Target Options], page 37).

For each subprogram to be run, the compiler driver first tries the '`-B`' prefix, if any. If that name is not found, or if '`-B`' was not specified, the driver tries two standard prefixes, which are '`/usr/lib/gcc/`' and '`/usr/local/lib/gcc/`'. If neither of those

results in a file name that is found, the unmodified program name is searched for using the directories specified in your '`PATH`' environment variable.

'`-B`' prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into '`-L`' options for the linker.

The run-time support file '`libgcc.a`' can also be searched for using the '`-B`' prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means.

Another way to specify a prefix much like the '`-B`' prefix is to use the environment variable `GCC_EXEC_PREFIX`. See Section 2.12 [Environment Variables], page 49.

2.9 Specifying Target Machine and Compiler Version

By default, GNU CC compiles code for the same type of machine that you are using. However, it can also be installed as a cross-compiler, to compile for some other type of machine. In fact, several different configurations of GNU CC, for different target machines, can be installed side by side. Then you specify which one to use with the '`-b`' option.

In addition, older and newer versions of GNU CC can be installed side by side. One of them (probably the newest) will be the default, but you may sometimes wish to use another.

`-b` *machine*

The argument *machine* specifies the target machine for compilation. This is useful when you have installed GNU CC as a cross-compiler.

The value to use for *machine* is the same as was specified as the machine type when configuring GNU CC as a cross-compiler. For example, if a cross-compiler was configured with '`configure i386v`', meaning to compile for an 80386 running System V, then you would specify '`-b i386v`' to run that cross compiler.

When you do not specify '`-b`', it normally means to compile for the same type of machine that you are using.

`-V` *version* The argument *version* specifies which version of GNU CC to run. This is useful when multiple versions are installed. For example, *version* might be '`2.0`', meaning to run GNU CC version 2.0.

The default version, when you do not specify '`-V`', is controlled by the way GNU CC is installed. Normally, it will be a version that is recommended for general use.

The '`-b`' and '`-V`' options actually work by controlling part of the file name used for the executable files and libraries used for compilation. A given version of GNU CC, for a given target machine, is normally kept in the directory '`/usr/local/lib/gcc/`*machine*`/`*version*'.

It follows that sites can customize the effect of '`-b`' or '`-V`' either by changing the names of these directories or adding alternate names (or symbolic links). Thus, if '`/usr/local/lib/gcc/80386`' is a link to '`/usr/local/lib/gcc/i386v`', then '`-b 80386`' will be an alias for '`-b i386v`'.

In one respect, the '`-b`' or '`-V`' do not completely change to a different compiler: the top-level driver program `gcc` that you originally invoked continues to run and invoke the other executables (preprocessor, compiler per se, assembler and linker) that do the real work. However, since no real work is done in the driver program, it usually does not matter that the driver program in use is not the one for the specified target and version.

The only way that the driver program depends on the target machine is in the parsing and handling of special machine-specific options. However, this is controlled by a file which is found, along with the other executables, in the directory for the specified version and target machine. As a result, a single installed driver program adapts to any specified target machine and compiler version.

The driver program executable does control one significant thing, however: the default version and target machine. Therefore, you can install different instances of the driver program, compiled for different targets or versions, under different names.

For example, if the driver for version 2.0 is installed as `ogcc` and that for version 2.1 is installed as `gcc`, then the command `gcc` will use version 2.1 by default, while `ogcc` will use 2.0 by default. However, you can choose either version with either command with the '`-V`' option.

2.10 Specifying Hardware Models and Configurations

Earlier we discussed the standard option '`-b`' which chooses among different installed compilers for completely different target machines, such as Vax vs. 68000 vs. 80386.

In addition, each of these target machine types can have its own special options, starting with '`-m`', to choose among various hardware models or configurations—for example, 68010 vs 68020, floating coprocessor or none. A single installed version of the compiler can compile for any model or configuration, according to the options specified.

These options are defined by the macro `TARGET_SWITCHES` in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

2.10.1 M680x0 Options

These are the '`-m`' options defined for the 68000 series. The default values for these options depends on which style of 68000 was selected when the compiler was configured; the defaults for the most common choices are given below.

`-m68020`

`-mc68020` Generate output for a 68020 (rather than a 68000). This is the default when the compiler is configured for 68020-based systems.

`-m68000`

`-mc68000` Generate output for a 68000 (rather than a 68020). This is the default when the compiler is configured for a 68000-based systems.

`-m68881` Generate output containing 68881 instructions for floating point. This is the default for most 68020 systems unless '`-nfp`' was specified when the compiler was configured.

`-mfpa` Generate output containing Sun FPA instructions for floating point.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mshort` Consider type `int` to be 16 bits wide, like `short int`.

`-mnobitfield`

Do not use the bit-field instructions. '`-m68000`' implies '`-mnobitfield`'.

`-mbitfield`

Do use the bit-field instructions. '`-m68020`' implies '`-mbitfield`'. This is the default if you use the unmodified sources configured for a 68020.

`-mrtd` Use a different function-calling convention, in which functions that take a fixed number of arguments return with the `rtd` instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

The `rtd` instruction is supported by the 68010 and 68020 processors, but not by the 68000.

2.10.2 VAX Options

These '`-m`' options are defined for the Vax:

`-munix` Do not output certain jump instructions (`aobleq` and so on) that the Unix assembler for the Vax cannot handle across long ranges.

`-mgnu` Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

`-mg` Output code for g-format floating point numbers instead of d-format.

2.10.3 SPARC Options

These '`-m`' switches are supported on the Sparc:

`-mno-epilogue`

Generate separate return instructions for `return` statements. This has both advantages and disadvantages; I don't recall what they are.

2.10.4 Convex Options

These '`-m`' options are defined for the Convex:

`-mc1` Generate output for a C1. This is the default when the compiler is configured for a C1.

`-mc2` Generate output for a C2. This is the default when the compiler is configured for a C2.

`-margcount`

Generate code which puts an argument count in the word preceding each argument list. Some nonportable Convex and Vax programs need this word. (Debuggers don't, except for functions with variable-length argument lists; this info is in the symbol table.)

`-mnoargcount`

Omit the argument count word. This is the default if you use the unmodified sources.

2.10.5 AMD29K Options

These '`-m`' options are defined for the AMD Am29000:

`-mdw` Generate code that assumes the `DW` bit is set, i.e., that byte and halfword operations are directly supported by the hardware. This is the default.

`-mnodw` Generate code that assumes the `DW` bit is not set.

`-mbw` Generate code that assumes the system supports byte and halfword write operations. This is the default.

`-mnbw` Generate code that assumes the systems does not support byte and halfword write operations. '`-mnbw`' implies '`-mnodw`'.

`-msmall` Use a small memory model that assumes that all function addresses are either within a single 256 KB segment or at an absolute address of less than 256K. This allows the `call` instruction to be used instead of a `const`, `consth`, `calli` sequence.

`-mlarge` Do not assume that the `call` instruction can be used; this is the default.

`-m29050` Generate code for the Am29050.

`-m29000` Generate code for the Am29000. This is the default.

`-mkernel-registers`

Generate references to registers `gr64-gr95` instead of `gr96-gr127`. This option can be used when compiling kernel code that wants a set of global registers disjoint from that used by user-mode code.

Note that when this option is used, register names in '`-f`' flags must use the normal, user-mode, names.

`-muser-registers`

Use the normal set of global registers, `gr96-gr127`. This is the default.

`-mstack-check`

Insert a call to `__msp_check` after each stack adjustment. This is often used for kernel code.

2.10.6 M88K Options

These '`-m`' options are defined for Motorola 88K architectures:

`-m88000` Generate code that works well on both the m88100 and the m88110.

`-m88100` Generate code tha Generate code that works best for the m88100, but that also runs on the m88110.

`-m88110` Generate code that works best for the m88110, and may not run on the m88100.

`-midentify-revision`

Include an `ident` directive in the assembler output recording the source file name, compiler name and version, timestamp, and compilation flags used.

`-mno-underscores`

In assembler output, emit symbol names without adding an underscore character at the beginning of each name. The default is to use an underscore as prefix on each name.

`-mocs-debug-info`

`-mno-ocs-debug-info`

Include (or omit) additional debugging information (about registers used in each stack frame) as specified in the 88open Object Compatibility Standard, "OCS". This extra information allows debugging of code that has had the frame pointer eliminated. The default for DG/UX, SVr4, and Delta 88 SVr3.2 is to include this information; other 88k configurations omit this information by default.

`-mocs-frame-position`

When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the canonical frame address, which is the stack pointer (register 31) on entry to the function. The DG/UX, SVr4, Delta88 SVr3.2, and BCS configurations use '`-mocs-frame-position`'; other 88k configurations have the default '`-mno-ocs-frame-position`'.

`-mno-ocs-frame-position`

When emitting COFF debugging information for automatic variables and parameters stored on the stack, use the offset from the frame pointer register (register 30). When this option is in effect, the frame pointer is not eliminated when debugging information is selected by the -g switch.

`-moptimize-arg-area`

`-mno-optimize-arg-area`

Control how to store function arguments in stack frames. '`-moptimize-arg-area`' saves space, but was ruled illegal by 88open. '`-mno-optimize-arg-area`' conforms to the 88open standards. By default GNU CC does not optimize the argument area.

`-mshort-data-`*num*

Generate smaller data references by making them relative to `r0`, which allows loading a value using a single instruction (rather than the usual two). You control which data references are affected by specifying *num* with this option. For example, if you specify '`-mshort-data-512`', then the data references affected are those involving displacements of less than 512 bytes. '`-mshort-data-`*num*' is not effective for *num* greater than 64K.

`-msvr4`

`-msvr3` Turn on ('`-msvr4`') or off ('`-msvr3`') compiler extensions related to System V release 4 (SVr4). This controls the following:

1. Which variant of the assembler syntax to emit (which you can select independently using '`-mversion-03.00`').
2. '`-msvr4`' makes the C preprocessor recognize '`#pragma weak`' that is used on System V release 4.
3. '`-msvr4`' makes GNU CC issue additional declaration directives used in SVr4.

'`-msvr3`' is the default for all m88K configurations except the SVr4 configuration.

`-mversion-03.00`

In the DG/UX configuration, there are two flavors of SVr4. This option modifies '`-msvr4`' to select whether the hybrid-COFF or real-ELF flavor is used. All other configurations ignore this option.

`-mno-check-zero-division`

`-mcheck-zero-division`

Early models of the 88K architecture had problems with division by zero; in particular, many of them didn't trap. Use these options to avoid including (or to include explicitly) additional code to detect division by zero and signal an exception. All GNU CC configurations for the 88K use '`-mcheck-zero-division`' by default.

`-muse-div-instruction`

Do not emit code to check both the divisor and dividend when doing signed integer division to see if either is negative, and adjust the signs so the divide is done using non-negative numbers. Instead, rely on the operating system to calculate the correct value when the `div` instruction traps. This results in different behavior when the most negative number is divided by -1, but is useful when most or all signed integer divisions are done with positive numbers.

`-mtrap-large-shift`

`-mhandle-large-shift`

Include code to detect bit-shifts of more than 31 bits; respectively, trap such shifts or emit code to handle them properly. By default GNU CC makes no special provision for large bit shifts.

`-mwarn-passed-structs`

Warn when a function passes a struct as an argument or result. Structure-passing conventions have changed during the evolution of the C language, and are often the source of portability problems. By default, GNU CC issues no such warning.

2.10.7 IBM RS/6000 Options

Only one pair of '`-m`' options is defined for the IBM RS/6000:

`-mfp-in-toc`
`-mno-fp-in-toc`

Control whether or not floating-point constants go in the Table of Contents (TOC), a table of all global variable and function addresses. By default GNU CC puts floating-point constants there; if the TOC overflows, '`-mno-fp-in-toc`' will reduce the size of the TOC, which may avoid the overflow.

2.10.8 IBM RT Options

These '`-m`' options are defined for the IBM RT PC:

`-min-line-mul`

Use an in-line code sequence for integer multiplies. This is the default.

`-mcall-lib-mul`

Call `lmul$$` for integer multiples.

`-mfull-fp-blocks`

Generate full-size floating point data blocks, including the minimum amount of scratch space recommended by IBM. This is the default.

`-mminimum-fp-blocks`

Do not include extra scratch space in floating point data blocks. This results in smaller code, but slower execution, since scratch space must be allocated dynamically.

`-mfp-arg-in-fpregs`

Use a calling sequence incompatible with the IBM calling convention in which floating point arguments are passed in floating point registers. Note that `varargs.h` and `stdargs.h` will not work with floating point operands if this option is specified.

`-mfp-arg-in-gregs`

Use the normal calling convention for floating point arguments. This is the default.

`-mhc-struct-return`

Return structures of more than one word in memory, rather than in a register. This provides compatibility with the MetaWare HighC (hc) compiler. Use '`-fpcc-struct-return`' for compatibility with the Portable C Compiler (pcc).

`-mnohc-struct-return`

Return some structures of more than one word in registers, when convenient. This is the default. For compatibility with the IBM-supplied compilers, use either '`-fpcc-struct-return`' or '`-mhc-struct-return`'.

2.10.9 MIPS Options

These '`-m`' options are defined for the MIPS family of computers:

`-mcpu=`*cpu type*

Assume the defaults for the machine type *cpu type* when scheduling insturctions. The default *cpu type* is '`default`', which picks the longest cycles times for any of the machines, in order that the code run at reasonable rates on all MIPS cpu's. Other choices for *cpu type* are '`r2000`', '`r3000`', '`r4000`', and '`r6000`'. While picking a specific *cpu type* will schedule things appropriately for that particular chip, the compiler will not generate any code that does not meet level 1 of the MIPS ISA (instruction set architecture) without the '`-mips2`' or '`-mips3`' switches being used.

`-mips2` Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). The '`-mcpu=r4000`' or '`-mcpu=r6000`' switch must be used in conjuction with '`-mips2`'.

`-mips3` Issue instructions from level 3 of the MIPS ISA (64 bit instructions). You must use the '`-mcpu=r4000`' switch along with '`-mips3`'.

`-mint64`

`-mlong64`

`-mlonglong128`

These options don't work at present.

`-mmips-as`

Generate code for the MIPS assembler, and invoke '`mips-tfile`' to add normal debug information. This is the default for all platforms except for the OSF/1 reference platform, using the OSF/rose object format. If the either of the '`-gstabs`' or '`-gstabs+`' switches are used, the '`mips-tfile`' program will encapsulate the stabs within MIPS ECOFF.

`-mgas` Generate code for the GNU assembler. This is the default on the OSF/1 reference platform, using the OSF/rose object format.

`-mrnames`
`-mno-rnames`

The '`-mrnames`' switch says to output code using the MIPS software names for the registers, instead of the hardware names (ie, *a0* instead of *$4*). The GNU assembler does not support the '`-mrnames`' switch, and the MIPS assembler will be instructed to run the MIPS C preprocessor over the source file. The '`-mno-rnames`' switch is default.

`-mgpopt`
`-mno-gpopt`

The '`-mgpopt`' switch says to write all of the data declarations before the instructions in the text section, to all the MIPS assembler to generate one word memory references instead of using two words for short global or static data items. This is on by default if optimization is selected.

`-mstats`
`-mno-stats`

For each non-inline function processed, the '`-mstats`' switch causes the compiler to emit one line to the standard error file to print statistics about the program (number of registers saved, stack size, etc.).

`-mmemcpy`
`-mno-memcpy`

The '`-mmemcpy`' switch makes all block moves call the appropriate string function ('`memcpy`' or '`bcopy`') instead of possibly generating inline code.

`-mmips-tfile`
`-mno-mips-tfile`

The '`-mno-mips-tfile`' switch causes the compiler not postprocess the object file with the '`mips-tfile`' program, after the MIPS assembler has generated it to add debug support. If '`mips-tfile`' is not run, then no local variables will be available to the debugger. In addition, '`stage2`' and '`stage3`' objects will have the temporary file names passed to the assembler embedded in the object file, which means the objects will not compare the same.

`-msoft-float`

Generate output containing library calls for floating point. **Warning:** the requisite libraries are not part of GNU CC. Normally the facilities of the machine's usual C compiler are used, but this can't be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

`-mhard-float`

Generate output containing floating point instructions. This is the default if you use the unmodified sources.

`-mfp64` Assume that the *FR* bit in the status word is on, and that there are 32 64-bit floating point registers, instead of 32 32-bit floating point registers. You must also specify the '`-mcpu=r4000`' and '`-mips3`' switches.

`-mfp32` Assume that there are 32 32-bit floating point registers. This is the default.

`-mabicalls`
`-mno-abicalls`

Emit the '`.abicalls`', '`.cpload`', and '`.cprestore`' pseudo operations that some System V.4 ports use for position independent code.

`-mhalf-pic`
`-mno-half-pic`

Put pointers to extern references into the data section and load them up, rather than put the references in the text section. These options do not work at present.

`-G` *num* Put global and static items less than or equal to *num* bytes into the small data or bss sections instead of the normal data or bss section. This allows the assembler to emit one word memory reference instructions based on the global pointer (*gp* or *$28*), instead of the normal two words used. By default, *num* is 8 when the MIPS assembler is used, and 0 when the GNU assembler is used. The '`-G` *num*' switch is also passed to the assembler and linker. All modules should be compiled with the same '`-G` *num*' value.

These options are defined by the macro `TARGET_SWITCHES` in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

2.11 Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of '`-ffoo`' would be '`-fno-foo`'. In the table below, only one of the forms is listed—the one which is not the default. You can figure out the other form by either removing '`no-`' or adding it.

`-fpcc-struct-return`

Use the same convention for returning `struct` and `union` values that is used by the usual C compiler on your system. This convention is less efficient for small structures, and on many machines it fails to be reentrant; but it has the advantage of allowing intercallability between GNU CC-compiled code and PCC-compiled code.

`-fshort-enums`

Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type will be equivalent to the smallest integer type which has enough room.

`-fshort-double`

Use the same size for `double` as for `float`.

`-fshared-data`

Requests that the data and non-`const` variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

`-fno-common`

Allocate even uninitialized global variables in the bss section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without `extern`) in two different compilations, you will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

`-fno-ident`

Ignore the '`#ident`' directive.

`-fno-gnu-linker`

Don't output global initializations such as C++ constructors and destructors in the form used by the GNU linker (on systems where the GNU linker is the standard method of handling them). Use this option when you want to use a "collect" program and a non-GNU linker.

`-finhibit-size-directive`

Don't output a `.size` assembler directive, or anything else that would cause trouble if the function is split in the middle, and the two halves are placed at locations far apart in memory. This option is used when compiling '`crtstuff.c`'; you should not need to use it for anything else.

`-fvolatile`

Consider all memory references through pointers to be volatile.

`-fpic` If supported for the target machines, generate position-independent code, suitable for use in a shared library. All addresses will be accessed through a global offset table (GOT). If the GOT size for the linked executable exceeds a machine-specific maximum size, you will get an error message from the linker indicating that '`-fpic`' does not work; recompile with '`-fPIC`' instead. (These maximums are 16k on the m88k, 8k on the Sparc, and 32k on the m68k and RS/6000. The 386 has no such limit.)

Position-independent code requires special support, and therefore works only on certain machines. Code generated for the IBM RS/6000 is always position-independent.

`-fPIC` If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on the m68k, m88k and the Sparc.

Position-independent code requires special support, and therefore works only on certain machines.

`-ffixed-`*reg*

Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the `REGISTER_NAMES` macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-used-`*reg*

Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-saved-`*reg*

Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

2.12 Environment Variables Affecting GNU CC

This section describes several environment variables that affect how GNU CC operates. They work by specifying directories or prefixes to use when searching for various kinds of files.

Note that you can also specify places to search using options such as '`-B`', '`-I`' and '`-L`' (see Section 2.8 [Directory Options], page 36). These take precedence over places specified using envi-

ronment variables, which in turn take precedence over those specified by the configuration of GNU CC. See Section 15.1 [Driver], page 217.

`TMPDIR` If `TMPDIR` is set, it specifies the directory to use for temporary files. GNU CC uses temporary files to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

`GCC_EXEC_PREFIX`

If `GCC_EXEC_PREFIX` is set, it specifies a prefix to use in the names of the subprograms executed by the compiler. No slash is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish.

If GNU CC cannot find the subprogram using the specified prefix, it tries looking in the usual places for the subprogram.

Other prefixes specified with '`-B`' take precedence over this prefix.

This prefix is also used for finding files such as '`crt0.o`' that are used for linking.

In addition, the prefix is used in an unusual way in finding the directories to search for header files. For each of the standard directories whose name normally begins with '`/usr/local/lib/gcc`' (more precisely, with the value of `GCC_INCLUDE_DIR`), GNU CC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with '`-Bfoo/`', GNU CC will search '`foo/bar`' where it would normally search '`/usr/local/lib/bar`'. These alternate directories are searched first; the standard directories come next.

`COMPILER_PATH`

The value of `COMPILER_PATH` is a colon-separated list of directories, much like `PATH`. GNU CC tries the directories thus specified when searching for subprograms, if it can't find the subprograms using `GCC_EXEC_PREFIX`.

`LIBRARY_PATH`

The value of `LIBRARY_PATH` is a colon-separated list of directories, much like `PATH`. GNU CC tries the directories thus specified when searching for special linker files, if it can't find them using `GCC_EXEC_PREFIX`. Linking using GNU CC also uses these directories when searching for ordinary libraries for the '`-l`' option (but directories specified with '`-L`' come first).

`C_INCLUDE_PATH`
`C++_INCLUDE_PATH`
`OBJC_INCLUDE_PATH`

These environment variables pertain to particular languages. Each variable's value is a colon-separated list of directories, much like `PATH`. When GNU CC searches for header files, it tries the directories listed in the variable for the language you are using, after the directories specified with '`-I`' but before the standard header file directories.

`DEPENDENCIES_OUTPUT`

If this variable is set, its value specifies how to output dependencies for Make based on the header files processed by the compiler. This output looks much like the output from the '`-M`' option (see Section 2.6 [Preprocessor Options], page 33), but it goes to a separate file, and is in addition to the usual results of compilation.

The value of `DEPENDENCIES_OUTPUT` can be just a file name, in which case the Make rules are written to that file, guessing the target name from the source file name. Or the value can have the form '*file target*', in which case the rules are written to file *file* using *target* as the target name.

3 Installing GNU CC

Here is the procedure for installing GNU CC on a Unix system.

See below for VMS systems, and modified procedures needed on other systems including Sun, 3B1, SCO Unix and Unos. The following section says how to compile in a separate directory on Unix; here we assume you compile in the same directory that contains the source files.

1. If you have built GNU CC previously in the same directory for a different target machine, do '`make cleanconfig`' to delete all files that might be invalid.
2. On a Sequent system, go to the Berkeley universe.
3. On a System V release 4 system, make sure '`/usr/bin`' precedes '`/usr/ucb`' in `PATH`. The `cc` command in '`/usr/ucb`' uses libraries which have bugs.
4. Specify the host and target machine configurations. You do this by running the file '`configure`' with appropriate arguments.

   If you are building a compiler to produce code for the machine it runs on, specify just one machine type. To build a cross-compiler, specify two configurations, one for the *host machine* (which the compiler runs on), and one for the *target machine* (which the compiler produces code for). The command looks like this:

   ```
   configure --host=sun3-sunos3 --target=sparc-sun-sunos4.1
   ```

   A configuration name may be canonical or it may be more or less abbreviated.

   A canonical configuration name has three parts, separated by dashes. It looks like this: '*cpu-company-system*'. (The three parts may themselves contain dashes; '`configure`' can figure out which dashes serve which purpose.) For example, '`m68k-sun-sunos4.1`' specifies a Sun 3.

   You can also replace parts of the configuration by nicknames or aliases. For example, '`sun3`' stands for '`m68k-sun`', so '`sun3-sunos4.1`' is another way to specify a Sun 3. You can also use simply '`sun3-sunos`', since the version of Sunos is assumed by default to be version 4. '`sun3-bsd`' also works, since '`configure`' knows that the only BSD variant on a Sun 3 is Sunos.

   You can specify a version number after any of the system types, and some of the CPU types. In most cases, the version is irrelevant, and will be ignored. So you might as well specify the version if you know it.

   Here are the possible CPU types:

   > a29k, arm, c*n*, hppa, i386, i860, m68000, m68k, m88k, mips, ns32k, romp, rs6000, sparc, vax.

   Note that the type hppa currently works only with Berkeley systems, not with HP/UX.

   Here are the recognized company names. As you can see, customary abbreviations are used rather than the longer official names.

alliant, altos, apollo, att, convergent, convex, crds, dec, dg, encore, harris, hp, ibm, mips, motorola, ncr, next, ns, omron, sequent, sgi, sony, sun, tti, unicom.

The company name is meaningful only to disambiguate when the rest of the information supplied is insufficient. You can omit it, writing just '*cpu*-*system*', if it is not needed. For example, '`vax-ultrix4.2`' is equivalent to '`vax-dec-ultrix4.2`'.

Here is a list of system types:

bsd, sysv, mach, minix, genix, ultrix, vms, sco, esix, isc, aix, sunos, hpux, unos, luna, dgux, newsos, osfrose, osf, dynix, aos, ctix.

You can omit the system type; then '`configure`' guesses the operating system from the CPU and company.

Often a particular model of machine has a name. Many of these names are recognized as an alias for a CPU/company combination. The alias '`sun3`', mentioned above, is an example of this: it stands for '`m68k-sun`'. Sometimes we accept a company name as a machine name, when the name is popularly used for a particular machine. Here is a table of the known machine names:

3300, 3b1, 7300, altos3068, altos, apollo68, att-7300, balance, convex-c*n*, crds, decstation-3100, decstation-dec, decstation, delta, encore, gmicro, hp7*nn*, hp8*nn*, hp9k2*nn*, hp9k3*nn*, hp9k7*nn*, hp9k8*nn*, iris4d, iris, isi68, m3230, magnum, merlin, miniframe, mmax, news-3600, news800, news, next, pbd, pc532, pmax, ps2, risc-news, rtpc, sun2, sun386i, sun386, sun3, sun4, symmetry, tower-32, tower.

If you specify an impossible combination such as '`i860-dg-vms`', then you may get an error message from '`configure`', or it may ignore part of the information and do the best it can with the rest. '`configure`' always prints the canonical name for the alternative that it used.

On certain systems, you must specify whether you want GNU CC to work with the usual compilation tools or with the GNU compilation tools (including GAS). Use the '`--gas`' argument when you run '`configure`', if you want to use the GNU tools. The systems were this makes a difference are '`i386-`*anything*`-sysv`', '`i860-`*anything*`-bsd`', '`m68k-hp-hpux`', '`m68k-sony-bsd`', '`m68k-altos-sysv`', '`m68000-hp-hpux`', and '`m68000-att-sysv`'. On any other system, '`--gas`' has no effect.

On certain systems, you must specify whether the machine has a floating point unit. These systems are '`m68k-sun-sunos`*n*' and '`m68k-isi-bsd`'. On any other system, '`--nfp`' currently has no effect, though perhaps there are other systems where it could usefully make a difference.

If you want to install your own homemade configuration files, you can use '`local`' as the company name to access them. If you use configuration '*cpu*`-local`', the entire configuration name is used to form the configuration file names.

Thus, if you specify '`m68k-local`', then the files used are '`m68k-local.md`', '`m68k-local.h`', '`m68k-local.c`', '`xm-m68k-local.h`', '`t-m68k-local`', and '`x-m68k-local`'.

Here is a list of configurations that have special treatment:

‘`m68000-att`’

AT&T 3b1, a.k.a. 7300 PC. Special procedures are needed to compile GNU CC with this machine's standard C compiler, due to bugs in that compiler. See Section 3.3 [3b1 Install], page 60. You can bootstrap it more easily with previous versions of GNU CC if you have them.

‘`m68000-hp-bsd`’

HP 9000 series 200 running BSD. Note that the C compiler that comes with this system cannot compile GNU CC; contact `law@super.org` to get binaries of GNU CC for bootstrapping.

‘`m68k-altos`’

Altos 3068. You must use the GNU assembler, linker and debugger, with COFF-encapsulation. Also, you must fix a kernel bug. Details in the file ‘`ALTOS-README`’.

‘`m68k-hp-hpux`’

HP 9000 series 200 or 300 running HPUX. GNU CC does not support the special symbol table used by HP's debugger, but you can debug programs with GDB if you specify ‘`--gas`’ to use the GNU tools instead. In order to use the GNU tools, you must install a library conversion program called `hpxt`.

‘`m68k-sun`’

Sun 3. We do not provide a configuration file to use the Sun FPA by default, because programs that establish signal handlers for floating point traps inherently cannot work with the FPA.

‘`m88k-dgux`’

Motorola m88k running DG/UX. To build native or cross compilers on DG/UX, you must first change to the 88open BCS software development environment. This is done by issuing this command:

```
eval ‘sde-target m88kbcs‘
```

‘`ns32k-encore`’

Encore ns32000 system. Encore systems are supported only under BSD.

‘`ns32k-*-genix`’

National Semiconductor ns32000 system. Genix has bugs in `alloca` and `malloc`; you must get the compiled versions of these from GNU Emacs.

‘`ns32k-utek`’

UTEK ns32000 system (“merlin”). The C compiler that comes with this system cannot compile GNU CC; contact ‘`tektronix!reed!mason`’ to get binaries of GNU CC for bootstrapping.

‘`rs6000-ibm`’

IBM PowerStation/6000 machines. Due to the nonstandard debugging information required for this machine, ‘`-g`’ is not available in this configuration.

‘`vax-dec-ultrix`’

Don’t try compiling with Vax C (`vcc`). It produces incorrect code in some cases (for example, when `alloca` is used).

Meanwhile, compiling ‘`cp-parse.c`’ with pcc does not work because of an internal table size limitation in that compiler. To avoid this problem, compile just the GNU C compiler first, and use it to recompile building all the languages that you want to run.

Here we spell out what files will be set up by `configure`. Normally you need not be concerned with these files.

- A symbolic link named ‘`config.h`’ is made to the top-level config file for the machine you will run the compiler on (see Chapter 16 [Config], page 299). This file is responsible for defining information about the host machine. It includes ‘`tm.h`’.

  The top-level config file is located in the subdirectory ‘`config`’. Its name is always ‘`xm-`*something*`.h`’; usually ‘`xm-`*machine*`.h`’, but there are some exceptions.

  If your system does not support symbolic links, you might want to set up ‘`config.h`’ to contain a ‘`#include`’ command which refers to the appropriate file.

- A symbolic link named ‘`tconfig.h`’ is made to the top-level config file for your target machine. This is used for compiling certain programs to run on that machine.

- A symbolic link named ‘`tm.h`’ is made to the machine-description macro file for your target machine. It should be in the subdirectory ‘`config`’ and its name is often ‘*machine*`.h`’.

- A symbolic link named ‘`md`’ will be made to the machine description pattern file. It should be in the ‘`config`’ subdirectory and its name should be ‘*machine*`.md`’; but *machine* is often not the same as the name used in the ‘`tm.h`’ file because the ‘`md`’ files are more general.

- A symbolic link named ‘`aux-output.c`’ will be made to the output subroutine file for your machine. It should be in the ‘`config`’ subdirectory and its name should be ‘*machine*`.c`’.

- The command file ‘`configure`’ also constructs ‘`Makefile`’ by adding some text to the template file ‘`Makefile.in`’. The additional text comes from files in the ‘`config`’ directory, named ‘`t-`*target*’ and ‘`h-`*host*’. If these files do not exist, it means nothing needs to be added for a given target or host.

5. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output files ‘`c-parse.c`’ and ‘`cexp.c`’ are more recent than ‘`c-parse.y`’ and ‘`cexp.y`’ and you do not plan to change the ‘`.y`’ files.)

   Bison versions older than Sept 8, 1988 will produce incorrect output for ‘`c-parse.c`’.

6. Build the compiler. Just type ‘`make LANGUAGES=c`’ in the compiler directory.

   ‘`LANGUAGES=c`’ specifies that only the C compiler should be compiled. The makefile normally builds compilers for all the supported languages; currently, C, C++ and Objective C. However, C is the only language that is sure to work when you build with other non-GNU C compilers. In addition, building anything but C at this stage is a waste of time.

In general, you can specify the languages to build by typing the argument '`LANGUAGES="`*list*`"`', where *list* is one or more words from the list '`c`', '`c++`', and '`objective-c`'.

Ignore any warnings you may see about "statement not reached" in '`insn-emit.c`'; they are normal. Any other compilation errors may represent bugs in the port to your machine or operating system, and should be investigated and reported (see Chapter 8 [Bugs], page 103).

Some commercial compilers fail to compile GNU CC because they have bugs or limitations. For example, the Microsoft compiler is said to run out of macro space. Some Ultrix compilers run out of expression space; then you need to break up the statement where the problem happens.

7. If you are using COFF-encapsulation, you must convert '`libgcc.a`' to a GNU-format library at this point. See the file '`README-ENCAP`' in the directory containing the GNU binary file utilities, for directions.
8. Move the first-stage object files and executables into a subdirectory with this command:

   ```
   make stage1
   ```

   The files are moved into a subdirectory named '`stage1`'. Once installation is complete, you may wish to delete these files with `rm -r stage1`.
9. Recompile the compiler with itself, with this command:

   ```
   make CC=stage1/gcc CFLAGS="-g -O -Bstage1/"
   ```

   This is called making the stage 2 compiler.

   The command shown above builds compilers for all the supported languages. If you don't want them all, you can specify the languages to build by typing the argument '`LANGUAGES="`*list*`"`'. *list* should contain one or more words from the list '`c`', '`c++`', and '`objective-c`', separated by spaces.

   On a 68000 or 68020 system lacking floating point hardware, unless you have selected a '`tm.h`' file that expects by default that there is no such hardware, do this instead:

   ```
   make CC=stage1/gcc CFLAGS="-g -O -Bstage1/ -msoft-float"
   ```
10. If you wish to test the compiler by compiling it with itself one more time, do this:

    ```
    make stage2
    make CC=stage2/gcc CFLAGS="-g -O -Bstage2/"
    ```

    This is called making the stage 3 compiler. Aside from the '`-B`' option, the options should be the same as when you made the stage 2 compiler.

    Then compare the latest object files with the stage 2 object files—they ought to be identical, unless they contain time stamps. On systems where object files do not contain time stamps, you can do this (in Bourne shell):

    ```
    for file in *.o; do
    cmp $file stage2/$file
    done
    ```

    This will mention any object files that differ between stage 2 and stage 3. Any difference, no matter how innocuous, indicates that the stage 2 compiler has compiled GNU CC incorrectly,

and is therefore a potentially serious bug which you should investigate and report (see Chapter 8 [Bugs], page 103).

On systems that use COFF object files, bytes 5 to 8 will always be different, since it is a timestamp. On these systems, you can do the comparison as follows (in Bourne shell):

```
for file in *.o; do
tail +10c $file > foo1
tail +10c stage2/$file > foo2
cmp foo1 foo2 || echo $file
done
```

On MIPS machines, you need to use the shell script '`ecoff-cmp`' to compare two object files if you have built the compiler with the '`-mno-mips-tfile`' option. Thus, do this:

```
for file in *.o; do
ecoff-cmp $file stage2/$file
done
```

11. Install the compiler driver, the compiler's passes and run-time support. You can use the following command:

    ```
    make CC=stage2/gcc install
    ```

    (Use the same value for `CC` that you used when compiling the files that are being installed.)

    This copies the files '`cc1`', '`cpp`' and '`libgcc.a`' to files '`cc1`', '`cpp`' and '`libgcc.a`' in directory '`/usr/local/lib/gcc/`*target*`/`*version*', which is where the compiler driver program looks for them. Here *target* is the target machine type specified when you ran '`configure`', and *version* is the version number of GNU CC. This naming scheme permits various versions and/or cross-compilers to coexist.

    It also copies the driver program '`gcc`' into the directory '`/usr/local/bin`', so that it appears in typical execution search paths.

    Warning: there is a bug in `alloca` in the Sun library. To avoid this bug, install the binaries of GNU CC that were compiled by GNU CC. They use `alloca` as a built-in function and never the one in the library.

12. If you will be using C++ or Objective C, and your operating system does not handle constructors, then you must build and install the program `collect2`. Do this with the following command:

    ```
    make CC="stage2/gcc -O" install-collect2
    ```

    The systems that **do** handle constructors on their own include system V release 4, and system V release 3 on the Intel 386.

    Berkeley systems that use the "a.out" object file format handle constructors without `collect2` if you use the GNU linker. But if you don't use the GNU linker, then you need `collect2` on these systems.

13. Build and install `protoize` if you want it. Type

    ```
    make CC="stage2/gcc -O" install-proto
    ```

There is as yet no documentation for `protoize`. Sorry.

14. Correct errors in the header files on your machine.

    Various system header files often contain constructs which are incompatible with ANSI C, and they will not work when you compile programs with GNU CC. This behavior consists of substituting for macro argument names when they appear inside of character constants. The most common offender is '`ioctl.h`'.

    You can overcome this problem when you compile by specifying the '`-traditional`' option.

    Alternatively, on Sun systems and 4.3BSD at least, you can correct the include files by running the shell script '`fixincludes`'. This installs modified, corrected copies of the files '`ioctl.h`', '`ttychars.h`' and many others, in a special directory where only GNU CC will normally look for them. This script will work on various systems because it chooses the files by searching all the system headers for the problem cases that we know about.

    Use the following command to do this:

    ```
    make install-fixincludes
    ```

    If you selected a different directory for GNU CC installation when you installed it, by specifying the Make variable `prefix` or `libdir`, specify it the same way in this command.

    Note that some systems are starting to come with ANSI C system header files. On these systems, don't run '`fixincludes`'; it may not work, and is certainly not necessary.

If you cannot install the compiler's passes and run-time support in '`/usr/local/lib`', you can alternatively use the '`-B`' option to specify a prefix by which they may be found. The compiler concatenates the prefix with the names '`cpp`', '`cc1`' and '`libgcc.a`'. Thus, you can put the files in a directory '`/usr/foo/gcc`' and specify '`-B/usr/foo/gcc/`' when you run GNU CC.

Also, you can specify an alternative default directory for these files by setting the Make variable `libdir` when you make GNU CC.

3.1 Compilation in a Separate Directory

If you wish to build the object files and executables in a directory other than the one containing the source files, here is what you must do differently:

1. Make sure you have a version of Make that supports the `VPATH` feature. (GNU Make supports it, as do Make versions on most BSD systems.)
2. Go to that directory before running '`configure`':

   ```
   mkdir gcc-sun3
   cd gcc-sun3
   ```

On systems that do not support symbolic links, this directory must be on the same file system as the source code directory.

3. Specify where to find '`configure`' when you run it:

   ```
   ../gcc-2.00/configure ...
   ```

   This also tells `configure` where to find the compiler sources; `configure` takes the directory from the file name that was used to invoke it. But if you want to be sure, you can specify the source directory with the '`--srcdir`' option, like this:

   ```
   ../gcc-2.00/configure --srcdir=../gcc-2.00 sun3
   ```

   The directory you specify with '`--srcdir`' need not be the same as the one that `configure` is found in.

Now, you can run `make` in that directory. You need not repeat the configuration steps shown above, when ordinary source files change. You must, however, run `configure` again when the configuration files change, if your system does not support symbolic links.

3.2 Installing GNU CC on the Sun

Make sure the environment variable `FLOAT_OPTION` is not set when you compile '`libgcc.a`'. If this option were set to `f68881` when '`libgcc.a`' is compiled, the resulting code would demand to be linked with a special startup file and would not link properly without special pains.

There is a bug in `alloca` in certain versions of the Sun library. To avoid this bug, install the binaries of GNU CC that were compiled by GNU CC. They use `alloca` as a built-in function and never the one in the library.

Some versions of the Sun compiler crash when compiling GNU CC. The problem is a segmentation fault in cpp. This problem seems to be due to the bulk of data in the environment variables. You may be able to avoid it by using the following command to compile GNU CC with Sun CC:

```
make CC="TERMCAP=x OBJS=x LIBFUNCS=x STAGESTUFF=x cc"
```

3.3 Installing GNU CC on the 3b1

Installing GNU CC on the 3b1 is difficult if you do not already have GNU CC running, due to bugs in the installed C compiler. However, the following procedure might work. We are unable to test it.

1. Comment out the '`#include "config.h"`' line on line 37 of '`cccp.c`' and do '`make cpp`'. This makes a preliminary version of GNU cpp.
2. Save the old '`/lib/cpp`' and copy the preliminary GNU cpp to that file name.
3. Undo your change in '`cccp.c`', or reinstall the original version, and do '`make cpp`' again.
4. Copy this final version of GNU cpp into '`/lib/cpp`'.
5. Replace every occurrence of `obstack_free` in the file '`tree.c`' with `_obstack_free`.
6. Run `make` to get the first-stage GNU CC.
7. Reinstall the original version of '`/lib/cpp`'.
8. Now you can compile GNU CC with itself and install it in the normal fashion.

3.4 Installing GNU CC on SCO System V 3.2

The compiler that comes with this system does not work properly with '`-O`'. Therefore, you should redefine the Make variable `CCLIBFLAGS` not to use '`-O`'.

In addition, the compiler produces incorrect output when compiling parts of GNU CC; the resulting executable '`cc1`' does not work properly when it is used with '`-O`'.

Therefore, what you must do after building the first stage is use GNU CC to compile itself without optimization. Here is how:

```
make -k cc1 CC="./gcc -B./"
```

You can think of this as "stage 1.1" of the installation process. However, using this command has the effect of discarding the faulty stage 1 executable for '`cc1`' and replacing it with stage 1.1. You can then proceed with '`make stage1`' and the rest of installation.

On Xenix, the same thing is necessary; in addition, you may have to remove '`-g`' from the options used with `cc`, and you may have to simplify complicated statements in the sources of GNU CC to get them to compile.

3.5 Installing GNU CC on Unos

Use '`configure unos`' for building on Unos.

The Unos assembler is named `casm` instead of `as`. For some strange reason linking '`/bin/as`' to '`/bin/casm`' changes the behavior, and does not work. So, when installing GNU CC, you should install the following script as '`as`' in the subdirectory where the passes of GCC are installed:

```
#!/bin/sh
casm $*
```

The default Unos library is named '`libunos.a`' instead of '`libc.a`'. To allow GNU CC to function, either change all references to '`-lc`' in '`gcc.c`' to '`-lunos`' or link '`/lib/libc.a`' to '`/lib/libunos.a`'.

When compiling GNU CC with the standard compiler, to overcome bugs in the support of `alloca`, do not use '`-O`' when making stage 2. Then use the stage 2 compiler with '`-O`' to make the stage 3 compiler. This compiler will have the same characteristics as the usual stage 2 compiler on other systems. Use it to make a stage 4 compiler and compare that with stage 3 to verify proper compilation.

Unos uses memory segmentation instead of demand paging, so you will need a lot of memory. 5 Mb is barely enough if no other tasks are running. If linking '`cc1`' fails, try putting the object files into a library and linking from that library.

3.6 Installing GNU CC on VMS

The VMS version of GNU CC is distributed in a backup saveset containing both source code and precompiled binaries.

To install the '`gcc`' command so you can use the compiler easily, in the same manner as you use the VMS C compiler, you must install the VMS CLD file for GNU CC as follows:

1. Define the VMS logical names '`GNU_CC`' and '`GNU_CC_INCLUDE`' to point to the directories where the GNU CC executables ('`gcc-cpp`', '`gcc-cc1`', etc.) and the C include files are kept. This should be done with the commands:

   ```
   $ assign /super /system disk:[gcc.] gnu_cc
   $ assign /super /system disk:[gcc.include.] gnu_cc_include
   ```

   with the appropriate disk and directory names. These commands can be placed in your system startup file so they will be executed whenever the machine is rebooted. You may, if you choose, do this via the '`GCC_INSTALL.COM`' script in the '`[GCC]`' directory.
2. Install the '`GCC`' command with the command line:

```
$ set command /table=sys$library:dcltables gnu_cc:[000000]gcc
```

3. To install the help file, do the following:

   ```
   $ lib/help sys$library:helplib.hlb gcc.hlp
   ```

   Now you can invoke the compiler with a command like '`gcc /verbose file.c`', which is equivalent to the command '`gcc -v -c file.c`' in Unix.

If you wish to use GNU C++ you must first install GNU CC, and then perform the following steps:

1. Define the VMS logical name '`GNU_GXX_INCLUDE`' to point to the directory where the preprocessor will search for the C++ header files. This can be done with the command:

   ```
   $ assign /super /system disk:[gcc.gxx_include.] gnu_gxx_include
   ```

   with the appropriate disk and directory name. If you are going to be using libg++, you should place the libg++ header files in the directory that this logical name points to.
2. Obtain the file '`gcc-cc1plus.exe`', and place this in the same directory that '`gcc-cc1.exe`' is kept.
3. You will need several library functions which are used to call the constructors and destructors for global objects. These functions are part of the libg++ distribution, and you will automatically get them if you install libg++.

   If you are not planning to install libg++, you will need to obtain the files '`gxx-startup-1.mar`' and '`gstart.cc`' from the libg++ distribution, compile them, and supply them to the linker whenever you link a C++ program.

   The GNU C++ compiler can be invoked with a command like '`gcc /plus /verbose file.cc`', which is equivalent to the command '`g++ -v -c file.cc`' in Unix.

We try to put corresponding binaries and sources on the VMS distribution tape. But sometimes the binaries will be from an older version that the sources, because we don't always have time to update them. (Use the '`/version`' option to determine the version number of the binaries and compare it with the source file '`version.c`' to tell whether this is so.) In this case, you should use the binaries you get to recompile the sources. If you must recompile, here is how:

1. Copy the file '`vms.h`' to '`tm.h`', '`xm-vms.h`' to '`config.h`', '`vax.md`' to '`md.`' and '`vax.c`' to '`aux-output.c`'. The files to be copied are found in the subdirectory named '`config`'; they should be copied to the main directory of GNU CC. If you wish, you may use the command file '`config-gcc.com`' to perform these steps for you.
2. Setup the logical names and command tables as defined above. In addition, define the VMS logical name '`GNU_BISON`' to point at the to the directories where the Bison executable is kept. This should be done with the command:

```
$ assign /super /system disk:[bison.] gnu_bison
```

You may, if you choose, use the 'INSTALL_BISON.COM' script in the '[BISON]' directory.

3. Install the 'BISON' command with the command line:

```
$ set command /table=sys$library:dcltables gnu_bison:[000000]bison
```

4. Type '@make-gcc' to recompile everything (alternatively, you may submit the file 'make-gcc.com' to a batch queue). If you wish to build the GNU C++ compiler as well as the GNU CC compiler, you must first edit 'make-gcc.com' and follow the instructions that appear in the comments.

   If you are building GNU CC with a previous version of GNU CC, you also should check to see that you have the newest version of the assembler. In particular, GNU CC version 2 treats global constant variables slightly differently from GNU CC version 1, and GAS version 1.38.1 does not have the patches required to work with GCC version 2. If you use GAS 1.38.1, then `extern const` variables will not have the read-only bit set, and the linker will generate warning messages about mismatched psect attributes for these variables. These warning messages are merely a nuisance, and can safely be ignored.

   If you are compiling with a version of GNU CC older than 1.33, specify '/DEFINE=("inline=")' as an option in all the compilations. This requires editing all the `gcc` commands in 'make-cc1.com'. (The older versions had problems supporting `inline`.) Once you have a working 1.33 or newer GNU CC, you can change this file back.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked to the sharable 'VAXCRTL' library. Now this should work.

Even with this version, however, GNU CC itself should not be linked to the sharable 'VAXCRTL'. The `qsort` routine supplied with 'VAXCRTL' has a bug which can cause a compiler crash.

Similarly, the preprocessor should not be linked to the sharable 'VAXCRTL'. The `strncat` routine supplied with 'VAXCRTL' has a bug which can cause the preprocessor to go into an infinite loop.

If you attempt to link to the sharable 'VAXCRTL', the VMS linker will strongly resist any effort to force it to use the `qsort` and `strncat` routines from 'gcclib'. Until the bugs in 'VAXCRTL' have been fixed, linking any of the compiler components to the sharable VAXCRTL is not recommended. (These routines can be bypassed by placing duplicate copies of `qsort` and `strncat` in 'gcclib' under different names, and patching the compiler sources to use these routines). Both of the bugs in 'VAXCRTL' are still present in VMS version 5.4-1, which is the most recent version as of this writing.

The executables that are generated by 'make-cc1.com' and 'make-cccp.com' use the nonshared version of 'VAXCRTL' (and thus use the `qsort` and `strncat` routines from 'gcclib.olb').

4 Known Causes of Trouble with GNU CC

Here are some of the things that have caused trouble for people installing or using GNU CC.

- On certain systems, defining certain environment variables such as `CC` can interfere with the functioning of `make`.
- Cross compilation can run into trouble for certain machines because some target machines' assemblers require floating point numbers to be written as *integer* constants in certain contexts.

  The compiler writes these integer constants by examining the floating point value as an integer and printing that integer, because this is simple to write and independent of the details of the floating point representation. But this does not work if the compiler is running on a different machine with an incompatible floating point format, or even a different byte-ordering.

  In addition, correct constant folding of floating point values requires representing them in the target machine's format. (The C standard does not quite require this, but in practice it is the only way to win.)

  It is now possible to overcome these problems by defining macros such as `REAL_VALUE_TYPE`. But doing so is a substantial amount of work for each target machine. See Section 15.18 [Cross-compilation], page 290.
- Users often think it is a bug when GNU CC reports an error for code like this:

```
int foo (short);

int foo (x)
     short x;
{...}
```

  The error message is correct: this code really is erroneous, because the old-style non-prototype definition passes subword integers in their promoted types. In other words, the argument is really an `int`, not a `short`. The correct prototype is this:

```
int foo (int);
```
- Users often think it is a bug when GNU CC reports an error for code like this:

```
int foo (struct mumble *);

struct mumble { ... };

int foo (struct mumble *x)
{ ... }
```

  This code really is erroneous, because the scope of `struct mumble` the prototype is limited to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

  But in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

This behavior may seem silly, but it's what the ANSI standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype. It's not worth being incompatible with ANSI C just to avoid an error for the example shown above.

- Certain local variables aren't recognized by debuggers when you compile with optimization.

  This occurs because sometimes GNU CC optimizes the variable out of existence. There is no way to tell the debugger how to compute the value such a variable "would have had", and it is not clear that would be desirable anyway. So GNU CC simply does not mention the eliminated variable when it writes debugging information.

  You have to expect a certain amount of disagreement between the executable and your source code, when you use optimization.

- `-2147483648` is positive.

  This is because 2147483648 cannot fit in the type `int`, so (following the ANSI C rules) its data type is `unsigned long int`. Negating this value yields 2147483648 again.

- Sometimes on a Sun 4 you may observe a crash in the program `genflags` while building GCC. This is said to be due to a bug in `sh`. You can probably get around it by running `genflags` manually and then retrying the `make`.

- On some versions of Ultrix, the system supplied compiler cannot compile '`cp-parse.c`' because it cannot handle so many cases in a `switch` statement. You can work around this problem by compiling with GNU CC.

- On some BSD systems including some versions of Ultrix, use of profiling causes static variable destructors (currently used only in C++) not to be run.

- On the IBM RS/6000, compiling code of the form

  ```
  extern int foo;

  ... foo ...

  static int foo;
  ```

  will cause the linker to report an undefined symbol `foo`. Although this behavior differs from most other systems, it is not a bug because redefining an `extern` variable as `static` is undefined in ANSI C.

For additional common problems, see Chapter 6 [Incompatibilities], page 69.

5 How To Get Help with GNU CC

If you need help installing, using or changing GNU CC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `bug-gcc@prep.ai.mit.edu`, and if that brings no response, try `help-gcc@prep.ai.mit.edu`.
- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named '`SERVICE`' in the GNU CC distribution.

6 Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C. The '`-traditional`' option eliminates most of these incompatibilities, *but not all*, by telling GNU C to behave like the other C compilers.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string or input. This is because `sscanf` incorrectly tries to write into the string constant. Likewise `fscanf` and `scanf`.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the '`-fwritable-strings`' flag, which directs GNU CC to handle string constants the same way most C compilers do. '`-traditional`' also has this effect, among others.

- GNU CC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) "a"
  ```

  will produce output `"a"` regardless of what the argument *a* is.

  The '`-traditional`' option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo ()
  {
    int a, b;

    a = fun1 ();
    if (setjmp (j))
      return a;

    a = fun2 ();
    /* longjmp (j) may occur in fun3. */
    return a + fun3 ();
  }
  ```

Here `a` may or may not be restored to its first value when the `longjmp` occurs. If `a` is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

If you use the '`-W`' option with the '`-O`' option, you will get a warning when GNU CC thinks such a problem might be possible.

The '`-traditional`' option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call `setjmp`. This results in the behavior found in traditional C compilers.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, a `extern` declaration affects all the rest of the file even if it happens within a block.

  The '`-traditional`' option directs GNU C to treat all `extern` declarations as global, like traditional compilers.

- In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the '`-traditional`' flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.

- PCC allows whitespace in the middle of compound assignment operators such as '`+=`'. GNU CC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.

- GNU CC will flag unterminated character constants inside of preprocessor conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GNU CC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '`/*...*/`'. However, '`-traditional`' suppresses these error messages.

- When compiling functions that return `float`, PCC converts it to a double. GNU CC actually returns a `float`. If you are concerned with PCC compatibility, you should declare your functions to return `double`; you might as well say what you mean.

- When compiling functions that return structures or unions, GNU CC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GNU CC 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. GNU CC does not use this method because it is slower and nonreentrant.

  On some newer machines, PCC uses a reentrant convention for all structure and union returning. GNU CC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

  You can tell GNU CC to use a compatible convention for all structure and union returning with the option '`-fpcc-struct-return`'.

There are also system-specific incompatibilities.

- On the Alliant, the system's own convention for returning structures and unions is unusual, and is not compatible with GNU CC no matter what options are used.
- On the IBM RT PC, the MetaWare HighC compiler (hc) uses yet another convention for structure and union returning. Use '`-mhc-struct-return`' to tell GNU CC to use a convention compatible with it.
- On Ultrix, the Fortran compiler expects registers 2 through 5 to be saved by function calls. However, the C compiler uses conventions compatible with BSD Unix: registers 2 through 5 may be clobbered by function calls.

  GNU CC uses the same convention as the Ultrix C compiler. You can use these options to produce code compatible with the Fortran compiler:

  ```
  -fcall-saved-r2 -fcall-saved-r3 -fcall-saved-r4 -fcall-saved-r5
  ```

- DBX rejects some files produced by GNU CC, though it accepts similar constructs in output from PCC. Until someone can supply a coherent description of what is valid DBX input and what is not, there is nothing I can do about these problems. You are on your own.

7 GNU Extensions to the C Language

GNU C provides several language features not found in ANSI standard C. (The '`-pedantic`' option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro `__GNUC__`, which is always defined under GNU CC.

7.1 Statements and Declarations within Expressions

A compound statement in parentheses may appear inside an expression in GNU C. This allows you to declare variables within an expression. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either *a* or *b* twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let's assume `int`), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see Section 7.6 [Typeof], page 78) or type naming (see Section 7.5 [Naming Types], page 78).

7.2 Locally Declared Labels

Each statement expression is a scope in which *local labels* can be declared. A local label is simply an identifier; you can jump to it with an ordinary `goto` statement, but only from within the statement expression it belongs to.

A local label declaration looks like this:

`__label__` *label*;

or

`__label__` *label1*, *label2*, ...;

Local label declarations must come at the beginning of the statement expression, right after the '`({`', before any ordinary declarations.

The label declaration defines the label *name*, but does not define the label itself. You must do this in the usual way, with *label*`:`, within the statements of the statement expression.

The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a `goto` can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(array, target)                     \
({                                                \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
```

```
        { value = i; goto found; }              \
  value = -1;                                     \
 found:                                           \
  value;                                          \
})
```

7.3 Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator '`&&`'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement[1], `goto *`*exp*`;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds—array indexing in C never does that.

---

[1] The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use that rather than an array unless the problem does not fit a `switch` statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

7.4 Nested Functions

A *nested function* is a function defined inside another function. The nested function's name is local to the block where it is defined. For example, here we define a nested function named `square`, and call it twice:

```
foo (double a, double b)
{
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. For example, here we show a nested function which uses an inherited variable named `offset`:

```
bar (int *array, int offset, int size)
{
  int access (int *array, int index)
    { return array[index + offset]; }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
}
```

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
hack (int *array, int size)
{
  void store (int index, int value)
    { array[index] = value; }
```

```
  intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. But this technique works only so long as the containing function (`hack`, in this example) does not exit. If you try to call the nested function through its address after the containing function has exited, all hell will break loose.

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (see Section 7.2 [Local Labels], page 74). Such a jump returns instantly to the containing function, exiting the nested function which did the `goto` and any intermediate functions as well. Here is an example:

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
  ...
  return 0;

 /* Control comes here from access
    if it detects an error.  */
 failure:
  return -1;
}
```

A nested function always has internal linkage. Declaring one with `extern` is erroneous. If you need to declare the nested function before its definition, use `auto` (which is otherwise meaningless for function declarations).

```
bar (int *array, int offset, int size)
{
  __label__ failure;
```

```
  auto int access (int *, int);
  ...
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  ...
}
```

7.5 Naming an Expression's Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define *name* as a type name for the type of *exp*:

```
typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a,b) \
  ({typedef _ta = (a), _tb = (b);  \
    _ta _a = (a); _tb _b = (b);     \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

7.6 Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of `typeof`. See Section 7.28 [Alternate Keywords], page 102.

A `typeof`-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

  ```
  typeof (*x) y;
  ```

- This declares `y` as an array of such values.

  ```
  typeof (*x) y[4];
  ```

- This declares `y` as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

  It is equivalent to the following traditional C declaration:

  ```
  char *y[4];
  ```

  To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

  ```
  #define pointer(T)  typeof(T *)
  #define array(T, N) typeof(T [N])
  ```

  Now the declaration can be rewritten this way:

  ```
  array (pointer (char), 4) y;
  ```

  Thus, `array (pointer (char), 4)` is the type of arrays of 4 pointers to `char`.

7.7 Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is an lvalue. A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if `a` has type `char *`, the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)(int)5)
```

An assignment-with-arithmetic operation such as '`+=`' applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *)(int) ((int)a + 5))
```

You cannot take the address of an lvalue cast, because the use of its address would not work out coherently. Suppose that `&(int)f` were permitted, where `f` has type `float`. Then the following statement would try to store an integer bit-pattern where a floating point number belongs:

```
*&(int)f = 1;
```

This is quite different from what `(int)f = 1` would do—that would convert 1 to floating point and store it. Rather than cause this inconsistancy, we think it is better to prohibit use of '`&`' on a cast.

If you really do want an `int *` pointer with the address of `f`, you can simply write `(int *)&f`.

7.8 Conditional Expressions with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

7.9 Double-Word Integers

GNU C supports data types for integers that are twice as long as `long int`. Simply write `long long int` for a signed integer, or `unsigned long long int` for an unsigned integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bitwise boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports fullword-to-doubleword a widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use special library routines that come with GNU CC.

There may be pitfalls when you use `long long` types for function arguments, unless you declare function prototypes. If a function expects type `int` for its argument, and you pass a value of type `long long int`, confusion will result because the caller and the subroutine will disagree about the number of bytes for the argument. Likewise, if the function expects `long long int` and you pass `int`. The best way to avoid such problems is to use prototypes.

7.10 Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline = (struct line *)
    malloc (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

7.11 Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace-level is exited. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca`.)

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len][len])
{
  ...
}
```

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case you access it with `sizeof`.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list—another GNU extension.

```
struct entry
tester (int len; char data[len][len], int len)
{
  ...
}
```

The '`int len`' before the semicolon is a *parameter forward declaration*, and it serves the purpose of making the name `len` known when the declaration of `data` is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type.

7.12 Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary '`&`' operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {int a[4];};

struct foo f();

bar (int index)
{
  return f().a[index];
}
```

7.13 Arithmetic on `void`- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

The option '`-Wpointer-arith`' requests a warning if these extensions are used.

7.14 Non-Constant Initializers

The elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
}
```

7.15 Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer.

Usually, the specified type is a structure. Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a `struct foo` with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a `switch` statement, while the latter does the same thing an ordinary C initializer would do. Here is an example of subscripting an array constructor:

```
output = ((int[]) { 2, x, 28 }) [input];
```

Constructor expressions for scalar types and union types are is also allowed, but then the constructor expression is equivalent to a cast.

7.16 Labeled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In GNU C you can give the elements in any order, specifying the array indices or structure field names they apply to.

To specify an array index, write '`[`*index*`]`' before the element value. For example,

```
int a[6] = { [4] 29, [2] 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

In a structure initializer, specify the name of a field to initialize with '*fieldname*`:`' before the element value. For example, given the following structure,

```
struct point { int x, y; };
```

the following initialization

```
struct point p = { y: yvalue, x: xvalue };
```

is equivalent to

```
struct point p = { xvalue, yvalue };
```

You can also use an element label when initializing a union, to specify which element of the union should be used. For example,

```
union foo { int i; double d; };

union foo f = { d: 4 };
```

will convert 4 to a `double` to store it in the union using the second element. By contrast, casting 4 to type `union foo` would store it into the union as the integer `i`, since it is an integer. (See Section 7.18 [Cast to Union], page 88.)

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a label applies to the next consecutive element of the array or structure. For example,

```
int a[6] = { [1] v1, v2, [4] v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type. For example:

```
int whitespace[256]
  = { [' '] 1, ['\t'] 1, ['\h'] 1,
      ['\f'] 1, ['\n'] 1, ['\r'] 1 };
```

7.17 Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

`case` *low* `...` *high*:

This has the same effect as the proper number of individual `case` labels, one for each integer value from *low* to *high*, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the `...`, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

7.18 Cast to a Union Type

A cast to union type is like any other cast, except that the type specified is a union type. You can specify the type either with `union` *tag* or with a typedef name.

The types that may be cast to the union type are those of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

both `x` and `y` can be cast to type `union` foo.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
...
u = (union foo) x  ≡  u.i = x
u = (union foo) y  ≡  u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

7.19 Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls.

A few standard library functions, such as `abort` and `exit`, cannot return. GNU CC knows this automatically. Some programs define their own functions that never return. You can declare them `volatile` to tell the compiler this fact. For example,

```
extern void volatile fatal ();

void
fatal (...)
{
  ... /* Print error message. */ ...
  exit (1);
}
```

The `volatile` keyword tells the compiler to assume that `fatal` cannot return. This makes slightly better code, but more importantly it helps avoid spurious warnings of uninitialized variables.

It does not make sense for a `volatile` function to have a return type other than `void`.

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared `const`. For example,

```
extern int const square ();
```

says that the hypothetical function `square` is safe to call fewer times than the program says.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to return `void`.

We recommend placing the keyword `const` after the function's return type. It makes no difference in the example above, but when the return type is a pointer, it is the only way to make the function itself const. For example,

```
const char *mincp (int);
```

says that `mincp` returns `const char *`—a pointer to a const object. To declare `mincp` const, you must write this:

```
char * const mincp (int);
```

Some people object to this feature, suggesting that ANSI C's `#pragma` should be used instead. There are two reasons for not doing this.

1. It is impossible to generate `#pragma` commands from a macro.
2. The `#pragma` command is just as likely as these keywords to mean something else in another compiler.

These two reasons apply to almost any application that might be proposed for `#pragma`. It is basically a mistake to use `#pragma` for *anything*.

7.20 Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

Dollar signs are allowed on certain machines if you specify '`-traditional`'. On a few systems they are allowed by default, even if '`-traditional`' is not used. But they are never allowed if you specify '`-ansi`'.

There are certain ANSI C programs (obscure, to be sure) that would compile incorrectly if dollar signs were permitted in identifiers. For example:

```
#define foo(a) #a
#define lose(b) foo (b)
#define test$
lose (test)
```

7.21 The Character ESC in Constants

You can use the sequence '`\e`' in a string or character constant to stand for the ASCII character `ESC`.

7.22 Inquiring on Alignment of Types or Variables

The keyword `__alignof__` allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like `sizeof`.

For example, if the target machine requires a `double` value to be aligned on an 8-byte boundary, then `__alignof__ (double)` is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof__ (double)` is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, `__alignof__` reports the *recommended* alignment of a type.

When the operand of `__alignof__` is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof__ (foo1.y)` is probably 2 or 4, the same as `__alignof__ (int)`, even though the data type of `foo1.y` does not itself demand any alignment.

7.23 Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields. The only attributes currently defined are the `aligned` and `format` attributes.

The `aligned` attribute specifies the alignment of the variable or structure field. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On a 68000, this could be used in conjunction with an `asm` expression to access the `move16` instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed.

The `format` attribute specifies that a function takes `printf` or `scanf` style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The first parameter of the `format` attribute determines how the format string is interpreted, and should be either `printf` or `scanf`. The second parameter specifies the number of the format string argument (starting from 1). The third parameter specifies the number of the first argument which should be checked against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency.

In the example above, the format string (`my_format`) is the second argument to `my_print` and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions which take format strings as

arguments, so that GNU CC can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `vprintf`, `vfprintf` and `vsprintf` whenever such warnings are requested (using '`-Wformat`'), so there is no need to modify the header file '`stdio.h`'.

7.24 An Inline Function is As Fast As a Macro

By declaring a function `inline`, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write `__inline__` instead of `inline`. See Section 7.28 [Alternate Keywords], page 102.)

You can also make all "simple enough" functions inline with the option '`-finline-functions`'. Note that certain usages in a function definition can make it unsuitable for inline substitution.

When a function is both inline and `static`, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option '`-fkeep-inline-functions`'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

7.25 Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

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

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has '`"f"`' as its operand constraint, saying that a floating point register is required. The '`=`' in '`=f`' indicates that the operand is an output; all output operands' constraints must use '`=`'. The constraints use the same language used in the machine description (see Section 14.6 [Constraints], page 175).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions that the compiler itself does not know exist.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. Extended asm does not support input-output or read-write operands. For this reason, the constraint character '`+`', which indicates such an operand, may not be used.

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) '`combine`' instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint '`"0"`' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `foo`'s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the Vax:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

If you refer to a particular hardware register from the assembler code, then you will probably have to list the register after the third colon to tell the compiler that the register's value is modified. In many assemblers, the register names begin with '`%`'; to produce one '`%`' in the assembler code, you must write '`%%`' in the input.

You can put multiple assembler instructions together in a single `asm` template, separated either with newlines (written as '`\n`') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all Unix assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine `_foo` accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;movl %1,r10;call _foo"
     : /* no outputs */
     : "g" (from), "g" (to)
     : "r9", "r10");
```

Unless an output operand has the '`&`' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '`&`' for each output operand that may not overlap an input. See Section 14.6.4 [Modifiers], page 181.

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the `asm` construct, as follows:

```
asm ("clr %0;frob %1;beq 0f;mov #1,%0;0:"
     : "g" (result)
     : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most Unix assemblers do.

Usually the most convenient way to use these `asm` instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
   __value; })
```

Here the variable `__arg` is used to make sure that the instruction operates on a proper `double` value, and to accept only those arguments `x` which can convert automatically to a `double`.

Another way to make sure the instruction operates on the correct data type is to use a cast in the `asm`. This is different from using a variable `__arg` in that it converts more different types. For example, if the desired type were `int`, casting the argument to `int` would accept a pointer with no complaint, while assigning the argument to an `int` variable named `__arg` would warn about using a pointer unless the caller explicitly casts it.

If an `asm` has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an `asm` instruction from being deleted, moved significantly, or combined, by writing the keyword `volatile` after the `asm`. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Note that even a volatile `asm` instruction can be moved in ways that appear insignificant to the compiler, such as across jump instructions. You can't expect a sequence of volatile `asm` instructions to remain perfectly consecutive. If you want consecutive output, use a single `asm`.

It is a natural idea to look for a way to give access to the condition code left by the assem-

bler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write `__asm__` instead of `asm`. See Section 7.28 [Alternate Keywords], page 102.

7.26 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use `asm` in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
extern func () asm ("FUNC");

func (x, y)
     int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU CC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

7.27 Variables in Specified Registers

GNU C allows you to put a few global variables into specified hardware registers. You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses.

  These local variables are sometimes convenient for use with the extended `asm` feature (see Section 7.25 [Extended Asm], page 94), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the `asm`.)

7.27.1 Defining Global Register Variables

You can define a global register variable in GNU C like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register `a5` would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a "global" register that is not affected magically by the function call mechanism.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e. in a different source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. (If you are prepared to recompile `qsort` with the same global register variable, you can solve this problem.)

If you want to recompile `qsort` or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler option '`-ffixed-`*reg*'. You need not actually add a global register declaration to their source code.

A function which can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`. On some machines, however, `longjmp` will not change the value of global register variables. To be portable, the function that called `setjmp` should make other arrangements to save the values of the global register variables, and to restore them in a `longjmp`. This way, the same thing will happen regardless of what `longjmp` does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

On the Sparc, there are reports that g3 ... g7 are suitable registers, but certain library functions, such as `getwd`, as well as the subroutines for division and remainder, modify g3 and g4. g1 and g2 are local temporaries.

On the 68000, a2 ... a5 should be suitable, as should d2 ... d7. Of course, it will not do to use more than a few of those.

7.27.2 Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("a5");
```

Here `a5` is the name of the register which should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Naturally the register name is cpu-dependent, but this is not a problem, since specific registers are most often useful with explicit assembler instructions (see Section 7.25 [Extended Asm], page 94). Both of these things generally require that you conditionalize your program according to cpu type.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register `%a5`.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. However, these registers are made unavailable for use in the reload pass. I would not be surprised if excessive use of this feature leaves the compiler too few available registers to compile certain functions.

7.28 Alternate Keywords

The option '`-traditional`' disables certain keywords; '`-ansi`' disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords `asm`, `typeof` and `inline` cannot be used since they won't work in a program compiled with '`-ansi`', while the keywords `const`, `volatile`, `signed`, `typeof` and `inline` won't work in a program compiled with '`-traditional`'.

The way to solve these problems is to put '`__`' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, `__const__` instead of `const`, and `__inline__` instead of `inline`.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

'`-pedantic`' causes warnings for many GNU C extensions. You can prevent such warnings within one expression by writing `__extension__` before the expression. `__extension__` has no effect aside from this.

7.29 Incomplete `enum` Types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write `struct foo` without describing the elements. A later declaration which does specify the possible values completes the type.

You can't allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.

8 Reporting Bugs

Your bug reports play an essential role in making GNU CC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 4 [Trouble], page 65. Also look in Chapter 6 [Incompatibilities], page 69. If it isn't known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. (If it does not, look in the service directory; see Chapter 5 [Service], page 67.) In any case, the principal function of a bug report is to help the entire community by making the next version of GNU CC work better. Bug reports are your contribution to the maintenance of GNU CC.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

8.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have run into an incompatibility between GNU C and traditional C (see Chapter 6 [Incompatibilities], page 69). These incompatibilities might be considered bugs, but they are inescapable consequences of valuable features.

  Or you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C compiler.

  For example, in many nonoptimizing compilers, you can write '`x;`' at the end of a function instead of '`return x;`', with the same results. But the value of the function is undefined if `return` is omitted; it is not a bug when GNU CC produces different results.

Problems often result from expressions with two increment operators, as in `f (*p++, *p++)`. Your previous compiler might have interpreted that expression the way you intended; GNU CC might interpret it another way. Neither compiler is wrong. The bug is in your code.

After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.

- If the compiler produces an error message for valid input, that is a compiler bug.

  Note that the following is not valid input, and the error message for it is not a bug:

  ```
  int foo (char);

  int
  foo (x)
       char x;
  { ... }
  ```

  The prototype says to pass a `char`, while the definition says to pass an `int` and treat the value as a `char`. This is what the ANSI standard says, and it makes sense.

- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".

- If you are an experienced user of C compilers, your suggestions for improvement of GNU CC are welcome in any case.

8.2 How to Report Bugs

Send bug reports for GNU C to one of these addresses:

```
bug-gcc@prep.ai.mit.edu
{ucbvax|mit-eddie|uunet}!prep.ai.mit.edu!bug-gcc
```

Do not send bug reports to 'help-gcc', or to the newsgroup 'gnu.gcc.help'. Most users of GNU CC do not want to receive bug reports. Those that do, have asked to be on 'bug-gcc'.

The mailing list 'bug-gcc' has a newsgroup which serves as a repeater. The mailing list and the newsgroup carry exactly the same messages. Often people think of posting bug reports to the newsgroup instead of mailing them. This appears to work, but it has one problem which can be crucial: a newsgroup posting does not contain a mail path back to the sender. Thus, if I need to ask for more information, I may be unable to reach you. For this reason, it is better to send bug reports to the mailing list.

As a last resort, send bug reports on paper to:

```
GNU Compiler Bugs
Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
```

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give a specific, complete example. That is the easiest thing for you to do, and the most helpful.

Keep in mind that the purpose of a bug report is to enable me to fix the bug if it is not known. It isn't very important what happens if the bug is already known. Therefore, always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" Those bug reports are useless, and I urge everyone to *refuse to respond to them* except to chide the sender to report bugs properly.

To enable me to fix the bug, you should include all these things:

- The version of GNU CC. You can get this by running it with the '`-v`' option.

  Without this, I won't know whether there is any point in looking for the bug in the current version of GNU CC.

- A complete input file that will reproduce the bug. If the bug is in the C preprocessor, send me a source file and any header files that it requires. If the bug is in the compiler proper ('`cc1`'), run your source file through the C preprocessor by doing '`gcc -E` *sourcefile* `>` *outfile*', then include the contents of *outfile* in the bug report. (Any '`-I`', '`-D`' or '`-U`' options that you used in actual compilation should also be used when doing this.)

  A single statement is not enough of an example. In order to compile it, it must be embedded in a function definition; and the bug might depend on the details of how this is done.

Without a real example I can compile, all I can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function they happen in.

- The command arguments you gave GNU CC to compile that example and observe the bug. For example, did you use '`-O`'? To guarantee you won't omit something important, list them all.

  If I were to try to guess the arguments, I would probably guess wrong and then I would not encounter the bug.
- The type of machine you are using, and the operating system name and version number.
- The operands you gave to the `configure` command when you installed the compiler.
- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal," or, "There is an incorrect assembler instruction in the output."

  Of course, if the bug is that the compiler gets a fatal signal, then I will certainly notice it. But if the bug is incorrect output, I might not notice unless it is glaringly wrong. I won't study all the assembler code from a 50-line C program just on the off chance that it might be wrong.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the compiler is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and mine would not. If you *told* me to expect a crash, then when mine fails to crash, I would know that the bug was not happening for me. If you had not told me to expect a crash, then I would not be able to draw any conclusion from my observations.

  Often the observed symptom is incorrect output when your program is run. Sad to say, this is not enough information for me unless the program is short and simple. If you send me a large program, I don't have time to figure out how it would work if compiled correctly, much less which line of it was compiled wrong. So you will have to do that. Tell me which source line it is, and what incorrect result happens when that line is executed. A person who understands the program can find this as easily as a bug in the program itself.
- If you send me examples of output from GNU CC, please use '`-g`' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.
- If you wish to suggest changes to the GNU CC source, send me context diffs. If you even discuss something in the GNU CC source, refer to it by context, not by line number.

  The line numbers in my development sources don't match those in your sources. Your line numbers would convey no useful information to me.
- Additional information from a debugger might enable me to find a problem on a machine which I do not have available myself. However, you need to think when you collect this information if you want it to have any chance of being useful.

  For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about GNU CC because the compiler is largely data-

driven; the same functions are called over and over for different RTL insns, doing different things depending on the details of the insn.

Most of the arguments listed in the backtrace are useless because they are pointers to RTL list structure. The numeric values of the pointers, which the debugger prints in the backtrace, have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are other such pointers).

In addition, most compiler passes consist of one or more loops that scan the RTL insn sequence. The most vital piece of information about such a loop—which insn it has reached—is usually in a local variable, not in an argument.

What you need to provide in addition to a backtrace are the values of the local variables for several stack frames up. When a local variable or an argument is an RTX, first print its value and then use the GDB command `pr` to print the RTL expression that it points to. (If GDB doesn't run on your machine, use your debugger to call the function `debug_rtx` with the RTX as an argument.) In general, whenever a variable is a pointer, its value is no use without the data it points to.

In addition, include a debugging dump from just before the pass in which the crash happens. Most bugs involve a series of insns, not just one.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way I will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. I recommend that you save your time for something else.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience for me. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most GNU CC bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations if the crucial function depends on them. (Exception: inline functions may affect compilation of functions defined later in the file.)

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send me the entire test case you used.

- A patch for the bug.

  A patch for the bug does help me if it is a good one. But don't omit the necessary information, such as the test case, on the assumption that a patch is all I need. I might see problems with your patch and decide to fix the problem another way, or I might not understand it at all.

Sometimes with a program as complicated as GNU CC it is very hard to construct an example that will make the program follow a certain path through the code. If you don't send me the example, I won't be able to construct one, so I won't be able to verify that the bug is fixed.

And if I can't understand what bug you are trying to fix, or why your patch should be an improvement, I won't install it. A test case will help me to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even I can't guess right about such things without first using the debugger to find the facts.

8.3 Certain Changes We Don't Want to Make

This section lists changes that people frequently request, but which we do not make because we think GNU CC is better without them.

- Checking the number and type of arguments to a function which has an old-fashioned definition and no prototype.

  Such a feature would work only occasionally—only for calls that appear in the same file as the called function, following the definition. The only way to check all calls reliably is to add a prototype for the function. But adding a prototype will eliminate the need for this feature. So the feature is not worthwhile.

- Warning about using an expression whose type is signed as a shift count.

  Shift count operands are probably signed more often than unsigned. Warning about this would cause far more annoyance than good.

- Warning about assigning a signed value to an unsigned variable.

  Such assignments must be very common; warning about them would cause more annoyance than good.

- Making bitfields unsigned by default on particular machines where "the ABI standard" says to do so.

  The ANSI C standard leaves it up to the implementation whether a bitfield declared plain `int` is signed or not. This in effect creates two alternative dialects of C.

  The GNU C compiler supports both dialects; you can specify the dialect you want with the option '`-fsigned-bitfields`' or '`-funsigned-bitfields`'. However, this leaves open the question of which dialect to use by default.

  Currently, the preferred dialect makes plain bitfields signed, because this is simplest. Since `int` is the same as `signed int` in every other context, it is cleanest for them to be the same in bitfields as well.

Some computer manufacturers have published Application Binary Interface standards which specify that plain bitfields should be unsigned. It is a mistake, however, to say anything about this issue in an ABI. This is because the handling of plain bitfields distinguishes two dialects of C. Both dialects are meaningful on every type of machine. Whether a particular object file was compiled using signed bitfields or unsigned is of no concern to functions in any other object file, even if they access the same bitfields in the same data structures.

A given program is written in one or the other of these two dialects. The program stands a chance to work on most any machine if it is compiled with the proper dialect. It is unlikely to work at all if compiled with the wrong dialect.

Many users appreciate the GNU C compiler because it provides an environment that is uniform across machines. These users would be inconvenienced if the compiler treated plain bitfields differently on certain machines.

Occasionally users write programs intended only for a particular machine type. On these occasions, the users would benefit if the GNU C compiler were to support by default the same dialect as the other compilers on that machine. But such applications are rare. And users writing a program to run on more than one type of machine cannot possibly benefit from this kind of compatibility.

This is why GNU CC does and will treat plain bitfields in the same fashion on all types of machines (by default).

(Of course, users strongly concerned about portability should indicate explicitly in each bitfield whether it is signed or not.)

- Undefining `__STDC__` when '`-ansi`' is not used.

  Currently, GNU CC defines `__STDC__` as long as you don't use '`-traditional`'. This provides good results in practice.

  Programmers normally use conditionals on `__STDC__` to ask whether it is safe to use certain features of ANSI C, such as function prototypes or ANSI token concatenation. Since plain '`gcc`' supports all the features of ANSI C, the correct answer to these questions is "yes".

  Some users try to use `__STDC__` to check for the availability of certain library facilities. This is actually incorrect usage in an ANSI C program, because the ANSI C standard says that a conforming freestanding implementation should define `__STDC__` even though it does not have the library facilities. '`gcc -ansi -pedantic`' is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ANSI C library.

  Sometimes people say that defining `__STDC__` in a compiler that does not completely conform to the ANSI C standard somehow violates the standard. This is illogical. The standard is a standard for compilers that are supposed to conform. It says nothing about what any other compilers should do. Whatever the ANSI C standard says is relevant to the design of plain '`gcc`' without '`-ansi`' only for pragmatic reasons, not as a requirement.

- Undefining `__STDC__` in C++.

  Programs written to compile with C++-to-C translators get the value of `__STDC__` that goes with the C compiler that is subsequently used. These programs must test `__STDC__` to determine what kind of C preprocessor that compiler uses: whether they should concatenate tokens in the ANSI C fashion or in the traditional fashion.

  These programs work properly with GNU C++ if `__STDC__` is defined. They would not work otherwise.

  In addition, many header files are written to provide prototypes in ANSI C but not in traditional C. Many of these header files can work without change in C++ provided `__STDC__` is defined. If `__STDC__` is not defined, they will all fail, and will all need to be changed to test explicitly for C++ as well.

9 Using GNU CC on VMS

9.1 Include Files and VMS

Due to the differences between the filesystems of Unix and VMS, GNU CC attempts to translate file names in '`#include`' into names that VMS will understand. The basic strategy is to prepend a prefix to the specification of the include file, convert the whole filename to a VMS filename, and then try to open the file. GNU CC tries various prefixes one by one until one of them succeeds:

1. The first prefix is the '`GNU_CC_INCLUDE:`' logical name: this is where GNU C header files are traditionally stored. If you wish to store header files in non-standard locations, then you can assign the logical '`GNU_CC_INCLUDE`' to be a search list, where each element of the list is suitable for use with a rooted logical.
2. The next prefix tried is '`SYS$SYSROOT:[SYSLIB.]`'. This is where VAX-C header files are traditionally stored.
3. If the include file specification by itself is a valid VMS filename, the preprocessor then uses this name with no prefix in an attempt to open the include file.
4. If the file specification is not a valid VMS filename (i.e. does not contain a device or a directory specifier, and contains a '`/`' character), the preprocessor tries to convert it from Unix syntax to VMS syntax.

   Conversion works like this: the first directory name becomes a device, and the rest of the directories are converted into VMS-format directory names. For example, '`X11/foobar.h`' is translated to '`X11:[000000]foobar.h`' or '`X11:foobar.h`', whichever one can be opened. This strategy allows you to assign a logical name to point to the actual location of the header files.
5. If none of these strategies succeeds, the '`#include`' fails.

Include directives of the form:

```
#include foobar
```

are a common source of incompatibility between VAX-C and GNU CC. VAX-C treats this much like a standard `#include <foobar.h>` directive. That is incompatible with the ANSI C behavior implemented by GNU CC: to expand the name `foobar` as a macro. Macro expansion should eventually yield one of the two standard formats for `#include`:

```
#include "file"
```

```
#include <file>
```

If you have this problem, the best solution is to modify the source to convert the `#include` directives to one of the two standard forms. That will work with either compiler. If you want a quick and dirty fix, define the file names as macros with the proper expansion, like this:

```
#define stdio <stdio.h>
```

This will work, as long as the name doesn't conflict with anything else in the program.

Another source of incompatibility is that VAX-C assumes that:

```
#include "foobar"
```

is actually asking for the file '`foobar.h`'. GNU CC does not make this assumption, and instead takes what you ask for literally; it tries to read the file '`foobar`'. The best way to avoid this problem is to always specify the desired file extension in your include directives.

GNU CC for VMS is distributed with a set of include files that is sufficient to compile most general purpose programs. Even though the GNU CC distribution does not contain header files to define constants and structures for some VMS system-specific functions, there is no reason why you cannot use GNU CC with any of these functions. You first may have to generate or create header files, either by using the public domain utility `UNSDL` (which can be found on a DECUS tape), or by extracting the relevant modules from one of the system macro libraries, and using an editor to construct a C header file.

9.2 Global Declarations and VMS

GNU CC does not provide the `globalref`, `globaldef` and `globalvalue` keywords of VAX-C. You can get the same effect with an obscure feature of GAS, the GNU assembler. (This requires GAS version 1.39 or later.) The following macros allow you to use this feature in a fairly natural way:

```
#ifdef __GNUC__
#define GLOBALREF(NAME) \
        NAME asm("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME  )
#define GLOBALDEF(NAME,VALUE) \
        NAME asm("_$$PsectAttributes_GLOBALSYMBOL$$" #NAME  ) = VALUE
```

```
#define GLOBALVALUEREF(NAME) \
  const NAME [1] asm("_$$PsectAttributes_GLOBALVALUE$$" #NAME  )
#define GLOBALVALUEDEF(NAME,VALUE) \
  const NAME [1] asm("_$$PsectAttributes_GLOBALVALUE$$" #NAME  ) = {VALUE}
#else
#define GLOBALREF(NAME) globalref NAME
#define GLOBALDEF(NAME,VALUE) globaldef NAME = VALUE
#define GLOBALVALUEDEF(NAME,VALUE) globalvalue NAME = VALUE
#define GLOBALVALUEREF(NAME) globalvalue NAME
#endif
```

(The `_$$PsectAttributes_GLOBALSYMBOL` prefix at the start of the name is removed by the assembler, after it has modified the attributes of the symbol). These macros are provided in the VMS binaries distribution in a header file '`GNU_HACKS.H`'. An example of the usage is:

```
int GLOBALREF (ijk);
int GLOBALDEF (jkl, 0);
```

The macros `GLOBALREF` and `GLOBALDEF` cannot be used straightforwardly for arrays, since there is no way to insert the array dimension into the declaration at the right place. However, you can declare an array with these macros if you first define a typedef for the array type, like this:

```
typedef int intvector[10];
intvector GLOBALREF (foo);
```

Array and structure initializers will also break the macros; you can define the initializer to be a macro of its own, or you can expand the `GLOBALDEF` macro by hand. You may find a case where you wish to use the `GLOBALDEF` macro with a large array, but you are not interested in explicitly initializing each element of the array. In such cases you can use an initializer like: `{0,}`, which will initialize the entire array to `0`.

A shortcoming of this implementation is that a variable declared with `GLOBALVALUEREF` or `GLOBALVALUEDEF` is always an array. For example, the declaration:

```
int GLOBALVALUEREF(ijk);
```

declares the variable `ijk` as an array of type `int [1]`. This is done because a globalvalue is actually a constant; its "value" is what the linker would normally consider an address. That is not how an integer value works in C, but it is how an array works. So treating the symbol as an array name gives consistent results—with the exception that the value seems to have the wrong type. **Don't**

try to access an element of the array. It doesn't have any elements. The array "address" may not be the address of actual storage.

The fact that the symbol is an array may lead to warnings where the variable is used. Insert type casts to avoid the warnings. Here is an example; it takes advantage of the ANSI C feature allowing macros that expand to use the same name as the macro itself.

```
int GLOBALVALUEREF (ss$_normal);
int GLOBALVALUEDEF (xyzzy,123);
#ifdef __GNUC__
#define ss$_normal ((int) ss$_normal)
#define xyzzy ((int) xyzzy)
#endif
```

Don't use `globaldef` or `globalref` with a variable whose type is an enumeration type; this is not implemented. Instead, make the variable an integer, and use a `globalvaluedef` for each of the enumeration values. An example of this would be:

```
#ifdef __GNUC__
int GLOBALDEF (color, 0);
int GLOBALVALUEDEF (RED, 0);
int GLOBALVALUEDEF (BLUE, 1);
int GLOBALVALUEDEF (GREEN, 3);
#else
enum globaldef color {RED, BLUE, GREEN = 3};
#endif
```

9.3 Other VMS Issues

GNU CC automatically arranges for `main` to return 1 by default if you fail to specify an explicit return value. This will be interpreted by VMS as a status code indicating a normal successful completion. Version 1 of GNU CC did not provide this default.

GNU CC on VMS works only with the GNU assembler, GAS. You need version 1.37 or later of GAS in order to produce value debugging information for the VMS debugger. Use the ordinary VMS linker with the object files produced by GAS.

Under previous versions of GNU CC, the generated code would occasionally give strange results when linked to the sharable '`VAXCRTL`' library. Now this should work.

A caveat for use of `const` global variables: the `const` modifier must be specified in every external declaration of the variable in all of the source files that use that variable. Otherwise the linker will issue warnings about conflicting attributes for the variable. Your program will still work despite the warnings, but the variable will be placed in writable storage.

The VMS linker does not distinguish between upper and lower case letters in function and variable names. However, usual practice in C is to distinguish case. Normally GNU CC (by means of the assembler GAS) implements usual C behavior by augmenting each name that is not all lower-case. A name is augmented by truncating it to at most 23 characters and then adding more characters at the end which encode the case pattern the rest.

Name augmentation yields bad results for programs that use precompiled libraries (such as Xlib) which were generated by another compiler. You can use the compiler option '`/NOCASE_HACK`' to inhibit augmentation; it makes external C functions and variables case-independent as is usual on VMS. Alternatively, you could write all references to the functions and variables in such libraries using lower case; this will work on VMS, but is not portable to other systems.

Function and variable names are handled somewhat differently with GNU C++. The GNU C++ compiler performs *name mangling* on function names, which means that it adds information to the function name to describe the data types of the arguments that the function takes. One result of this is that the name of a function can become very long. Since the VMS linker only recognizes the first 31 characters in a name, special action is taken to ensure that each function and variable has a unique name that can be represented in 31 characters.

If the name (plus a name augmentation, if required) is less than 32 characters in length, then no special action is performed. If the name is longer than 31 characters, the assembler (GAS) will generate a hash string based upon the function name, truncate the function name to 23 characters, and append the hash string to the truncated name. If the '`/VERBOSE`' compiler option is used, the assembler will print both the full and truncated names of each symbol that is truncated.

The '`/NOCASE_HACK`' compiler option should not be used when you are compiling programs that use libg++. libg++ has several instances of objects (i.e. `Filebuf` and `filebuf`) which become indistinguishable in a case-insensitive environment. This leads to cases where you need to inhibit augmentation selectively (if you were using libg++ and Xlib in the same program, for example). There is no special feature for doing this, but you can get the result by defining a macro for each mixed case symbol for which you wish to inhibit augmentation. The macro should expand into the lower case equivalent of itself. For example:

```
#define StuDlyCapS studlycaps
```

These macro definitions can be placed in a header file to minimize the number of changes to your source code.

10 GNU CC and Portability

The main goal of GNU CC was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary.

GNU CC 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, I have not hesitated to define an ad-hoc parameter to the machine description. The purpose of portability is to reduce the total work needed on the compiler; it was not of interest for its own sake.

GNU CC 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 I have not tried to address all possible cases, but only the common ones or only the ones that I have 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.

11 Interfacing to GNU CC Output

GNU CC 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 15 [Machine Macros], page 217).

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 GNU CC, and vice versa. This does not cause trouble often because few Unix library routines return structures or unions.

GNU CC 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. (GNU CC 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 machine-description macros `STRUCT_VALUE` and `STRUCT_INCOMING_VALUE` tell GNU CC 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 GNU CC, 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, GNU CC 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.

GNU CC 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 GNU CC 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 GNU CC.

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

If you want a variable to be unaltered by `longjmp`, and you don't want to write `volatile` because old C compilers don't accept it, just take the address of the variable. If a variable's address is ever taken, even if just to compute it and ignore it, then the variable cannot go in a register:

```
{
  int careful;
  &careful;
  ...
}
```

Code compiled with GNU CC may call certain library routines. Most of them handle arithmetic for which there are no instructions. This includes multiply and divide on some machines, and floating point operations on any machine for which floating point support is disabled with '`-msoft-float`'. Some standard parts of the C library, such as `bcopy` or `memcpy`, are also called automatically. The usual function call interface is used for calling the library routines.

These library routines should be defined in the library '`libgcc.a`', which GNU CC automatically searches whenever it links a program. On machines that have multiply and divide instructions, if hardware floating point is in use, normally '`libgcc.a`' is not needed, but it is searched just in case.

Each arithmetic function is defined in '`libgcc1.c`' to use the corresponding C arithmetic operator. As long as the file is compiled with another C compiler, which supports all the C arithmetic operators, this file will work portably. However, '`libgcc1.c`' does not work if compiled with GNU CC, because each arithmetic function would compile into a call to itself!

12 Passes and Files of the Compiler

The overall control structure of the compiler is in '`toplev.c`'. This file is responsible for initialization, decoding arguments, opening and closing files, and sequencing the passes.

The parsing pass is invoked only once, to parse the entire input. The RTL intermediate code for a function is generated as the function is parsed, a statement at a time. Each statement is read in as a syntax tree and then converted to RTL; then the storage for the tree for the statement is reclaimed. Storage for types (and the expressions for their sizes), declarations, and a representation of the binding contours and how they nest, remain until the function is finished being compiled; these are all needed to output the debugging information.

Each time the parsing pass reads a complete function definition or top-level declaration, it calls the function `rest_of_compilation` or `rest_of_decl_compilation` in '`toplev.c`', which are responsible for all further processing necessary, ending with output of the assembler language. All other compiler passes run, in sequence, within `rest_of_compilation`. When that function returns from compiling a function definition, the storage used for that function definition's compilation is entirely freed, unless it is an inline function (see Section 7.24 [Inline], page 93).

Here is a list of all the passes of the compiler and their source files. Also included is a description of where debugging dumps can be requested with '`-d`' options.

- Parsing. This pass reads the entire text of a function definition, constructing partial syntax trees. This and RTL generation are no longer truly separate passes (formerly they were), but it is easier to think of them as separate.

  The tree representation does not entirely follow C syntax, because it is intended to support other languages as well.

  Language-specific data type analysis is also done in this pass, and every tree node that represents an expression has a data type attached. Variables are represented as declaration nodes.

  Constant folding and some arithmetic simplifications are also done during this pass.

  The language-independent source files for parsing are '`stor-layout.c`', '`fold-const.c`', and '`tree.c`'. There are also header files '`tree.h`' and '`tree.def`' which define the format of the tree representation.

  The source files for parsing C are '`c-parse.y`', '`c-decl.c`', '`c-typeck.c`', '`c-convert.c`', '`c-lang.c`', and '`c-aux-info.c`' along with header files '`c-lex.h`', and '`c-tree.h`'.

  The source files for parsing C++ are '`cp-parse.y`', '`cp-class.c`', '`cp-cvt.c`', '`cp-decl.c`', '`cp-decl.c`', '`cp-decl2.c`', '`cp-dem.c`', '`cp-except.c`', '`cp-expr.c`', '`cp-init.c`', '`cp-lex.c`', '`cp-method.c`', '`cp-ptree.c`',

'`cp-search.c`', '`cp-tree.c`', '`cp-type2.c`', and '`cp-typeck.c`', along with header files '`cp-tree.def`', '`cp-tree.h`', and '`cp-decl.h`'.

The special source files for parsing Objective C are '`objc-parse.y`', '`objc-actions.c`', '`objc-tree.def`', and '`objc-actions.h`'. Certain C-specific files are used for this as well.

The file '`c-common.c`' is also used for all of the above languages.

- RTL generation. This is the conversion of syntax tree into RTL code. It is actually done statement-by-statement during parsing, but for most purposes it can be thought of as a separate pass.

  This is where the bulk of target-parameter-dependent code is found, since often it is necessary for strategies to apply only when certain standard kinds of instructions are available. The purpose of named instruction patterns is to provide this information to the RTL generation pass.

  Optimization is done in this pass for `if`-conditions that are comparisons, boolean operations or conditional expressions. Tail recursion is detected at this time also. Decisions are made about how best to arrange loops and how to output `switch` statements.

  The source files for RTL generation include '`stmt.c`', '`function.c`', '`expr.c`', '`calls.c`', '`explow.c`', '`expmed.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.

  Aside from debugging information output, none of the following passes refers to the tree structure representation of the function (only part of which is saved).

  The decision of whether the function can and should be expanded inline in its subsequent callers is made at the end of rtl generation. The function must meet certain criteria, currently related to the size of the function and the types and number of parameters it has. Note that this function may contain loops, recursive calls to itself (tail-recursive functions can be inlined!), gotos, in short, all constructs supported by GNU CC. The file '`integrate.c`' contains the code to save a function's rtl for later inlining and to inline that rtl when the function is called. The header file '`integrate.h`' is also used for this purpose.

  The option '`-dr`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.rtl`' to the input file name.

- Jump optimization. This pass simplifies jumps to the following instruction, jumps across jumps, and jumps to jumps. It deletes unreferenced labels and unreachable code, except that unreachable code that contains a loop is not recognized as unreachable in this pass. (Such loops are deleted later in the basic block analysis.) It also converts some code originally written with jumps into sequences of instructions that directly set values from the results of comparisons, if the machine has such instructions.

Jump optimization is performed two or three times. The first time is immediately following RTL generation. The second time is after CSE, but only if CSE says repeated jump optimization is needed. The last time is right before the final pass. That time, cross-jumping and deletion of no-op move instructions are done together with the optimizations described above.

The source file of this pass is '`jump.c`'.

The option '`-dj`' causes a debugging dump of the RTL code after this pass is run for the first time. This dump file's name is made by appending '`.jump`' to the input file name.

- Register scan. This pass finds the first and last use of each register, as a guide for common subexpression elimination. Its source is in '`regclass.c`'.
- Jump threading. This pass detects a condition jump that branches to an identical or inverse test. Such jumps can be '`threaded`' through the second conditional test. The source code for this pass is in '`jump.c`'. This optimization is only performed if '`-fthread-jumps`' is enabled.
- Common subexpression elimination. This pass also does constant propagation. Its source file is '`cse.c`'. If constant propagation causes conditional jumps to become unconditional or to become no-ops, jump optimization is run again when CSE is finished.

  The option '`-ds`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.cse`' to the input file name.
- Loop optimization. This pass moves constant expressions out of loops, and optionally does strength-reduction and loop unrolling as well. Its source files are '`loop.c`' and '`unroll.c`', plus the header '`loop.h`' used for communication between them. Loop unrolling uses some functions in '`integrate.c`' and the header '`integrate.h`'.

  The option '`-dL`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.loop`' to the input file name.
- If '`-frerun-cse-after-loop`' was enabled, a second common subexpression elimination pass is performed after the loop optimization pass. Jump threading is also done again at this time if it was specified.

  The option '`-dt`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.cse2`' to the input file name.
- Stupid register allocation is performed at this point in a nonoptimizing compilation. It does a little data flow analysis as well. When stupid register allocation is in use, the next pass executed is the reloading pass; the others in between are skipped. The source file is '`stupid.c`'.
- Data flow analysis ('`flow.c`'). This pass divides the program into basic blocks (and in the process deletes unreachable loops); then it 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.

  This pass also deletes computations whose results are never used, and combines memory references with add or subtract instructions to make autoincrement or autodecrement addressing.

The option '`-df`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.flow`' to the input file name. If stupid register allocation is in use, this dump file reflects the full results of such allocation.

- Instruction combination ('`combine.c`'). 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 option '`-dc`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.combine`' to the input file name.
- Instruction scheduling ('`sched.c`'). 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.

  Instruction scheduling is performed twice. The first time is immediately after instruction combination and the second is immediately after reload.

  The option '`-dS`' causes a debugging dump of the RTL code after this pass is run for the first time. The dump file's name is made by appending '`.sched`' to the input file name.
- 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 ('`local-alloc.c`'). 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 very good job.

  The option '`-dl`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.lreg`' to the input file name.
- Global register allocation ('`global-alloc.c`'). This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block).
- 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.

  The option '`-dg`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.greg`' to the input file name.

- Instruction scheduling is repeated here to try to avoid pipeline stalls due to memory loads generated for spilled pseudo registers.

  The option '`-dR`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.sched2`' to the input file name.

- Jump optimization is repeated, this time including cross-jumping and deletion of no-op move instructions.

  The option '`-dJ`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.jump2`' to the input file name.

- 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`'.

  The option '`-dd`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.dbr`' to the input file name.

- 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`'.

  The options '`-dk`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.stack`' to the input file name.

- Final. This pass outputs the assembler code for the function. It is also responsible for identifying spurious test and compare instructions. Machine-specific peephole optimizations are performed at the same time. The function entry and exit sequences are generated directly as assembler code in this pass; they never exist as RTL.

  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.

- 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, and '`dwarfout.c`' for DWARF symbol table format.

Some additional files are used by all or many passes:

- Every pass uses '`machmode.def`' and '`machmode.h`' which define the machine modes.
- Several passes use '`real.h`', which defines the default representation of floating point constants and how to operate on them.
- All the passes that work with RTL use the header files '`rtl.h`' and '`rtl.def`', and subroutines in file '`rtl.c`'. The tools `gen*` also use these files to read and work with the machine description RTL.

- Several passes refer to the header file '`insn-config.h`' which contains a few parameters (C macro definitions) generated automatically from the machine description RTL by the tool `genconfig`.
- Several passes use the instruction recognizer, which consists of '`recog.c`' and '`recog.h`', plus the files '`insn-recog.c`' and '`insn-extract.c`' that are generated automatically from the machine description by the tools '`genrecog`' and '`genextract`'.
- Several passes use the header files '`regs.h`' which defines the information recorded about pseudo register usage, and '`basic-block.h`' which defines the information recorded about basic blocks.
- '`hard-reg-set.h`' defines the type `HARD_REG_SET`, a bit-vector with a bit for each hard register, and some macros to manipulate it. This type is just `int` if the machine has few enough hard registers; otherwise it is an array of `int` and some of the macros expand into loops.
- Several passes use instruction attributes. A definition of the attributes defined for a particular machine is in file '`insn-attr.h`', which is generated from the machine description by the program '`genattr`'. The file '`insn-attrtab.c`' contains subroutines to obtain the attribute values for insns. It is generated from the machine description by the program '`genattrtab`'.

13 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.

13.1 RTL Object Types

RTL uses four kinds of objects: expressions, 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 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.

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 upper case) 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 lower case, but when they appear in C code they are written in upper case. 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)`.

13.2 Access to Operands

For each expression type '`rtl.def`' specifies the number of contained objects and their kinds, with four possibilities: '`e`' for expression (actually a pointer to an expression), '`i`' for integer, '`s`' for string, and '`E`' for vector of expressions. The sequence of letters for an expression code is called its *format*. Thus, the format of `subreg` is '`ei`'.

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.

`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, the format, and the class 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.

`GET_RTX_CLASS (`*code*`)`

A single character representing the type of RTX operation that code *code* performs.

The following classes are defined:

`o` An RTX code that represents an actual object, such as `reg` or `mem`. `subreg` is not in this class.

`<` An RTX code for a comparison. The codes in this class are `NE`, `EQ`, `LE`, `LT`, `GE`, `GT`, `LEU`, `LTU`, `GEU`, `GTU`.

`1` An RTX code for a unary arithmetic operation, such as `neg`.

`c` An RTX code for a commutative binary operation, other than `NE` and `EQ` (which have class '`<`').

`2` An RTX code for a noncommutative binary operation, such as `MINUS`.

`b` An RTX code for a bitfield operation (`ZERO_EXTRACT` and `SIGN_EXTRACT`).

`3` An RTX code for other three input operations, such as `IF_THEN_ELSE`.

`i` An RTX code for a machine insn (`INSN`, `JUMP_INSN`, and `CALL_INSN`).

`m` An RTX code for something that matches in insns, such as `MATCH_DUP`.

`x` All other RTX codes.

Operands of expressions are accessed using the macros `XEXP`, `XINT` 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.

13.3 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:

`MEM_VOLATILE_P` (*x*)

In `mem` expressions, nonzero for volatile memory references. 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. Stored in the `in_struct` field and printed as '`/s`'.

`REG_LOOP_TEST_P`

In `reg` expressions, nonzero if this register's entire life is contained in the exit test code for some loop. Stored in the `in_struct` field and printed as '`/s`'.

`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`'.

`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`'.

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_UNCHANGING_P` (*x*)

Nonzero in a `reg` or `mem` if the value is not changed. (This flag is not set for memory references via pointers to constants. Such pointers only guarantee that the object will not be changed explicitly by the current function. The object might be changed by other functions or by aliasing.) Stored in the `unchanging` field and printed as '`/u`'.

`RTX_INTEGRATED_P` (*insn*)

Nonzero in an insn if it resulted from an in-line function call. Stored in the `integrated` field and printed as '`/i`'. This may be deleted; nothing currently depends on it.

`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_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`'.

`LABEL_OUTSIDE_LOOP_P`

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`'.

`INSN_DELETED_P (`*insn*`)`

In an insn, nonzero if the insn has been deleted. Stored in the `volatil` field and printed as '`/v`'.

`INSN_ANNULLED_BRANCH_P (`*insn*`)`

In an `insn` in the delay slot of a branch insn, indicates that an annulling branch should be used. See the discussion under `sequence` below. Stored in the `unchanging` field and printed as '`/u`'.

`INSN_FROM_TARGET_P (`*insn*`)`

In an `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 should only be executed if the branch is taken. For annulled branches with this bit clear, the insn should be executed only if the branch is not taken. Stored in the `in_struct` field and printed as '`/s`'.

`CONSTANT_POOL_ADDRESS_P (`*x*`)`

Nonzero in a `symbol_ref` if it refers to part of the current function's "constants pool". These are addresses close to the beginning of the function, and GNU CC assumes they can be addressed directly (perhaps with the help of base registers). Stored in the `unchanging` field and printed as '`/u`'.

`CONST_CALL_P (`*x*`)`

In a `call_insn`, indicates that the insn represents a call to a const function. Stored in the `unchanging` field and printed as '`/u`'.

`LABEL_PRESERVE_P (`*x*`)`

In a `code_label`, indicates that the label can never be deleted. Labels referenced by a a non-local goto will have this bit set. Stored in the `in_struct` field and printed as '`/s`'.

`SCHED_GROUP_P (`*insn*`)`

During instruction scheduling, in an 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`'.

These are the fields which the above macros refer to:

`used` Normally, this flag is used only momentarily, 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 13.17 [Sharing], page 165).

In a `symbol_ref`, it indicates that an external declaration for the symbol has already been written.

In a `reg`, it is used by the leaf register renumbering code to ensure that each register is only renumbered once.

`volatil` This flag is used in `mem`,`symbol_ref` and `reg` expressions and in insns. In RTL dump files, it is printed as '`/v`'.

In a `mem` 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_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 an insn in the delay slot of a branch, 1 means that this insn is from the target of the branch.

During instruction scheduling, in an insn, 1 means that this insn must be scheduled as part of a group together with the previous insn.

In `reg` expressions, it is 1 if the register has its entire life contained within the test expression of some loopl.

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.

In an RTL dump, this flag is represented as '`/s`'.

`unchanging`

In `reg` and `mem` expressions, 1 means that the value of the expression never changes.

In an insn, 1 means that this is an annulling branch.

In a `symbol_ref` expression, 1 means that this symbol addresses something in the per-function constants pool.

In a `call_insn`, 1 means that this instruction is a call to a const function.

In an RTL dump, this flag is represented as '`/u`'.

`integrated`

In some kinds of expressions, including insns, this flag means the rtl was produced by procedure integration.

In a `reg` expression, this flag indicates the register containing 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.

13.4 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 15.3 [Storage Layout], page 223).

`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.

`SFmode` "Single Floating" mode represents a single-precision (four byte) floating point number.

`DFmode` "Double Floating" mode represents a double-precision (eight byte) floating point number.

`XFmode` "Extended Floating" mode represents a triple-precision (twelve byte) floating point number. This mode is used for IEEE extended floating point.

`TFmode` "Tetra Floating" mode represents a quadruple-precision (sixteen byte) floating point number.

`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 15.12 [Condition Code], page 267).

`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.

`SCmode, DCmode, XCmode, TCmode`

These modes stand for a complex number represented as a pair of floating point values. The values are in `SFmode`, `DFmode`, `XFmode`, and `TFmode`, respectively. Since C does not support complex numbers, these machine modes are only partially implemented.

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 `QImode`, `HImode`, `SImode`, `DImode`, and `TImode`.

`MODE_PARTIAL_INT`

The "partial integer" modes, `PSImode` and `PDImode`.

`MODE_FLOAT`

floating point modes. By default these are `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 `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 modes listed in the `EXTRA_CC_MODES` macro. See Section 14.10 [Jump Patterns], page 192, also see Section 15.12 [Condition Code], page 267.

`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. E.g., `GET_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 `BITS_PER_UNIT` or `BITS_PER_WORD`, respectively. On 32-bit machines, these are `QImode` and `SImode`, respectively.

13.5 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 `XINT (`*exp*`, 0)`.

Keep in mind that 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. In many cases, for proper results, you must carefully disregard the values of those bits.

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 that is too large to fit into `HOST_BITS_PER_INT` bits but small enough to fit within twice that number of bits (GNU CC does not provide a mechanism to represent even larger constants). In the latter case, *m* will be `VOIDmode`.

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 bit 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 (either single or double precision), then the number of integers used to store the value depends on the size of `REAL_VALUE_TYPE` (see Section 15.18 [Cross-compilation], page 290). The integers represent a `double`. To convert them to a `double`, do

```
union real_extract u;
bcopy (&CONST_DOUBLE_LOW (x), &u, sizeof u);
```

and then refer to `u.d`.

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`. Otherwise, it returns a `CONST_DOUBLE` 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`.

`(const_string` *str*`)`

Represents a constant string with value *str*. Currently this is used only for insn attributes (see Section 14.15 [Insn Attributes], page 204) since constant strings in C are placed in memory.

`(symbol_ref` *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 '`_`'.

`(label_ref` *label*`)`

Represents the value of an assembler label for code. It contains one operand, an expression, which must be a `code_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.

`(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`.

13.6 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* (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_DOWNWARDS` is defined, this points to immediately above the first variable on the stack. Otherwise, it points to the first variable on the stack.

It 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.

It 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`).

It is replaced by the sum of the register given by `STACK_POINTER_REGNUM` and the value `STACK_POINTER_OFFSET`.

`(subreg:`*m reg wordnum*`)`

`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-word `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, *wordnum* 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*. However, 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 *wordnum* that says which register.

The compilation parameter `WORDS_BIG_ENDIAN`, if set to 1, says that word number zero is the most significant part; otherwise, it is the least significant part.

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) 1)` 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_WORD` 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 13.12 [Side Effects], page 150).

`(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 15.12 [Condition Code], page 267. 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*`)`

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.

13.7 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*`)`

Represents the sum of the values represented by *x* and *y* carried out in machine mode *m*.

`(lo_sum:`*m x y*`)`

Like `plus`, except that it represents that sum of *x* and the low-order bits of *y*. 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 the `high` code (see Section 13.5 [Constants], page 137).

m should be `Pmode`.

`(minus:`*m x y*`)`

Like `plus` but represents subtraction.

`(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.

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 15.12 [Condition Code], page 267.

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.

Write patterns for unsigned widening multiplication similarly using `zero_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 integers in mode *m*.

`(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*.

`(lshift:`*m x c*`)`

Like `lshift` but for arithmetic left shift. `ashift` and `lshift` are identical operations; we customarily use `ashift` for both.

`(lshiftrt:`*m x c*`)`
`(ashiftrt:`*m x c*`)`

Like `lshift` and `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.

13.8 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 15.19 [Misc], page 293) if the relation holds, or zero if it does not. 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`. If the comparison operation is producing data to be stored in some variable, the mode must be in class `MODE_INT`. All comparison operations producing data must use the same mode, which is machine-specific.

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 instructing 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*`)`

1 if the values represented by *x* and *y* are equal, otherwise 0.

`(ne:`*m x y*`)`

1 if the values represented by *x* and *y* are not equal, otherwise 0.

`(gt:`*m x y*`)`

1 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 non-zero test, or *default* if none of the tests are non-zero expressions.

This is currently not valid for instruction patterns and is supported only for insn attributes. See Section 14.15 [Insn Attributes], page 204.

13.9 Bit Fields

Special expression codes exist to represent bit-field instructions. These types of expressions are lvalues in RTL; they may appear on the left side of an assignment, indicating insertion of a value into the specified bit field.

`(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 14.7 [Standard Names], page 183) and is usually a full-word integer mode.

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.

`(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.

13.10 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*.

`(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.

13.11 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: operand 0 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.

13.12 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` or `strict_low_part`), `mem`, `pc` 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 `(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 `(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 Vaxes. 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` 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))`, it means that all memory locations must be presumed clobbered.

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 14.3 [RTL Template], page 169) 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 14.6.4 [Modifiers], page 181). You can clobber either a specific hard register, a pseudo register, or a `scratch` expression; in the latter two cases, GNU CC 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.

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.

`(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 14.15.6 [Delay Slots], page 212.

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 [`*operands ...*`]` *index*`)`

Represents a machine-specific operation on *operands*. *index* selects between multiple macine-specific operations. `unspec_volatile` is used for volatile operations and operations that may trap; `unspec` is used for other operations.

These codes may appear themselves 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 ...*`])`

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.

13.13 Embedded Side-Effects on Addresses

Four 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.

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.

13.14 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.

13.15 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*`))` is *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 contains 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.

Return insns count as jumps, but since they do not refer to any labels, they have zero in the `JUMP_LABEL` field.

`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.

A `call_insn` insn may be preceeded by insns that contain a single `use` expression and be followed by insns the contain a single `clobber` expression. If so, these `use` and `clobber` expressions are treated as being part of the function call. There must not even be a `note` between the `call_insn` and the `use` or `clobber` insns for this special treatment to take place. This is somewhat of a kludge and will be removed in a later version of GNU CC.

`call_insn` insns have the same extra fields as `insn` insns, accessed in the same way.

`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 '`L`*n*' 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.

The field `LABEL_NUSES` is only defined once the jump optimization phase is completed and contains the number of times this label is referenced in the current function.

`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_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_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_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.

`NOTE_INSN_SETJMP`

Appears following each call to `setjmp` or a related function.

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; for example, the reload pass sets it to `HImode` if the insn needs reloading but not register elimination and `QImode` if both are required. 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.

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`, or `parallel`. 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 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.

This does not necessarily mean that the register *op* has no useful value after this insn since it may also be an output of the insn. In such a case, however, a `REG_DEAD` note would be redundant and is usually not present until after the reload pass, but no code relies on this fact.

`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 contains a pattern named '`decrement_and_branch_until_zero`'.

`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`, but is not a `jump_insn`. The presence of this note allows jump optimization to be aware that *op* is, in fact, being used.

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.

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 there 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.

`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_WAS_0`

The single output of this insn contained zero before this insn. *op* is the insn that set it to zero. You can rely on this note if it is present and *op* has not been deleted or turned into a `note`; its absence implies nothing.

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.

`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.

`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).

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.

13.16 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 it 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 15.5.1 [Register Basics], page 229) 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.

13.17 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.
- There is only one `const_int` expression with value 0, only one with value 1, and only one with value −1. Some other integer values are also stored uniquely.
- 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.
- 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`.

14 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.

14.1 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.
2. The *RTL template* (see Section 14.3 [RTL Template], page 169) 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`.

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 14.4 [Output Template], page 172.

   When simple substitution isn't general enough, you can specify a piece of C code to compute the output. See Section 14.5 [Output Statement], page 173.

5. Optionally, a vector containing the values of attributes for insns matching this pattern. See Section 14.15 [Insn Attributes], page 204.

14.2 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 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.

14.3 RTL Template for Generating and Recognizing Insns

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.

predicate is a string that is the name of a C function that accepts two arguments, an expression and a machine mode. 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 14.6 [Constraints], page 175).

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.

On CISC machines, *predicate* is most often `"general_operand"`. This function checks that the putative operand is either a constant, a register or a memory reference, and that it is valid for mode *m*.

For an operand that must be a register, *predicate* should be `"register_operand"`. It would be valid to use `"general_operand"`, since the reload pass would copy any non-register operands through registers, but this would make GNU CC do extra work, it would prevent invariant operands (such as constant) from being removed from loops, and it would prevent the register allocator from doing the best possible job. On RISC machines, it is usually most efficient to allow *predicate* to accept only objects that the constraints allow.

For an operand that must be a constant, either use `"immediate_operand"` for *predicate*, or make the instruction pattern's extra condition require a constant, or both. You cannot expect the constraints to do this work! If the constraints allow only constants, but the predicate allows something else, the compiler will crash when that case arises.

`(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 completely 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 them when necessary. See Section 13.12 [Side Effects], page 150.

`(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` behaves exactly 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.

`(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_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) == 'c' || 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.

`(address (match_operand:`*m* *n* `"address_operand" ""))`

This complex of expressions is a placeholder for an operand number *n* in a "load address" instruction: an operand which specifies a memory location in the usual way, but for which the actual operand value used is the address of the location, not the contents of the location.

`address` expressions never appear in RTL code, only in machine descriptions. And they are used only in machine descriptions that do not use the operand constraint feature. When operand constraints are in use, the letter '`p`' in the constraint serves this purpose.

m is the machine mode of the *memory location being addressed*, not the machine mode of the address itself. That mode is always the same on a given target machine (it is `Pmode`, which normally is `SImode`), so there is no point in mentioning it; thus, no machine mode is written in the `address` expression. If some day support is added for machines in which addresses of different kinds of objects appear differently or are used differently (such as the PDP-10), different formats would perhaps need different machine modes and these modes might be written in the `address` expression.

14.4 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.

‘%’ 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.

14.5 C Statements for Generating 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 14.6.2 [Multi-Alternative], page 180). 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 %1,%0
   addm %1,%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 '\'.

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")
```

14.6 Operand Constraints

Each `match_operand` in an instruction pattern can specify a constraint for the type of operands allowed. 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.

14.6.1 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:

'`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.

'd', 'a', 'f', ...

Other letters can be defined in machine-dependent fashion to stand for particular classes of registers. 'd', 'a' and 'f' are defined on the 68000/68020 to stand for data, address and floating point registers.

'i' An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.

'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`) 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 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 machines an add instruction really has only two operands, one of them an input-output operand.

Matching constraints work only in circumstances like that add insn. 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.

'`Q`', '`R`', '`S`', ... '`U`'

Letters in the range '`Q`' through '`U`' may be defined in a machine-dependent fashion to stand for arbitrary operand types. The machine description macro `EXTRA_CONSTRAINT`

is passed the operand as its first argument and the constraint letter as its second operand.

A typical use for this would be to distinguish certain types of memory references that affect other insn operands.

Do not define these constraint letters to accept register references (`reg`); the reload pass does not expect this and would not handle it properly.

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.

14.6.2 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 14.6.3 [Class Preferences], page 181).

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 14.5 [Output Statement], page 173.

14.6.3 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.

14.6.4 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.

'`&`' Means (in a particular alternative) that this operand is written 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.

'&' 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")))]
  ...)
```

'#' 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")))]
  ...)
```

14.6.5 Not Using Constraints

Some machines are so clean that operand constraints are not required. For example, on the Vax, an operand valid in one context is valid in any other context. On such a machine, every operand constraint would be 'g', excepting only operands of "load address" instructions which are written as if they referred to a memory location's contents but actual refer to its address. They would have constraint 'p'.

For such machines, instead of writing 'g' and 'p' for all the constraints, you can choose to write a description with empty constraints. Then you write '""' for the constraint in every `match_operand`. Address operands are identified by writing an `address` expression around the `match_operand`, not by their constraints.

When the machine description has just empty constraints, certain parts of compilation are skipped, making the compiler faster. However, few machines actually do not need constraints; all machine descriptions now in existence use constraints.

14.7 Standard Names for Patterns Used in 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 in to accomplish a certain task.

'`mov`*m*' Here *m* stands for a two-letter machine mode name, in lower case. 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*. The effect on the rest of the register is undefined.

This class of patterns is special in several ways. First of all, each of these names *must* be defined, because there is no other way to copy a datum from one place to another.

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. Look in '`spur.md`' to see how the requirement can be satisfied.

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 reload. But this is impossible during and after reload. If there are cases needing scratch registers 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 15.6 [Register Classes], page 235.

The constraints on a '`move`*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 '`move`*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 '`move`*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 '`move`*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 15.6 [Register Classes], page 235.

'`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*.

'`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’
‘smin*m*3’, ‘smax*m*3’, ‘umin*m*3’, ‘umax*m*3’
‘and*m*3’, ‘ior*m*3’, ‘xor*m*3’

Similar, for other arithmetic operations.

‘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.

‘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. Operand 2 has mode `SImode`, not mode *m*.

‘ashr*m*3’, ‘lshl*m*3’, ‘lshr*m*3’, ‘rotl*m*3’, ‘rotr*m*3’

Other shift and rotate instructions.

Logical and arithmetic left shift are the same. Machines that do not allow negative shift counts often have only one instruction for shifting left. On such machines, you should define a pattern named ‘ashl*m*3’ and leave ‘lshl*m*3’ undefined.

‘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.

'`ffs`*m*`2`' 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.

'`one_cmpl`*m*`2`'

Store the bitwise-complement of operand 1 into operand 0.

'`cmp`*m*' 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 ...)))
```

'`tst`*m*' Compare operand 0 against zero, and set the condition codes. The RTL pattern should look like this:

```
(set (cc0) (match_operand:m 0 ...))
```

'`tst`*m*' 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 '`cmp`*m*' patterns should be used instead.

'`movstr`*m*' Block move instruction. The addresses of the destination and source strings are the first two operands, and both are in mode `Pmode`. The number of bytes to move is the third operand, in mode *m*.

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.

These patterns need not give special consideration to the possibility that the source and destination strings might overlap.

'`cmpstr`*m*' Block compare instruction, with five operands. Operand 0 is the output; it has mode *m*. The remaining four operands are like the operands of '`movstr`*m*'. The two memory blocks specified are compared byte by byte in lexicographic order. The effect of the instruction is to store a value in operand 0 whose sign indicates the result of the comparison.

'`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.

'`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*' 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*'

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*'

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.

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.

'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 15.19 [Misc], page 293). 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 14.13 [Expander Definitions], page 199) for the conditional branch operations. All calls to expand 'v*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 14.10 [Jump Patterns], page 192

The above discussion also applies to 's*cond*' patterns.

'`call`' Subroutine call instruction returning no value. Operand 0 is the function to call; operand 1 is the number of bytes of arguments pushed (in mode `SImode`, except it is normally 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 14.13 [Expander Definitions], page 199) 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 non-zero. 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 non-zero, the use of these patterns increases the number of functions for which the frame pointer can be eliminated, if desired.

'`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 non-zero 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.

'`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. (If the machine-description macro `CASE_DROPS_THROUGH` is defined, then an out-of-bounds index drops through to the code following the jump table instead of jumping to this label. In that case, this label is not actually used by the '`casesi`' instruction, but it is always provided as an operand.)

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` is defined then the first operand is an offset which counts from the address of the table; otherwise, it is an absolute address to jump to.

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.

14.8 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.

14.9 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. You must ensure that the predicate permits all objects allowed by the constraints to prevent the compiler from crashing.

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.

14.10 Defining Jump Instruction Patterns

For most machines, GNU CC 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 GNU CC 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 15.12 [Condition Code], page 267.) 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 the macro `EXTRA_CC_MODES` to list the additional modes required (see Section 15.12 [Condition Code], page 267). Also define `EXTRA_CC_NAMES` to list the names of those modes and `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`.

14.11 Canonicalization of Instructions

There are often cases where multiple RTL expressions could represent an operation peformed 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 these operators, if only one operand is a `neg`, `not`, `mult`, `plus`, or `minus` expression, it will be the first operand.
- For the `compare` operator, a constant is always the second operand on machines where `cc0` is used (see Section 14.10 [Jump Patterns], page 192). 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.

14.12 Defining 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.

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_nonnote_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)))])
   ...]
  ...)
```

14.13 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 just like the RTL template for a `define_peephole` in that it is a vector of RTL expressions each being one insn.
- 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 GNU CC is run. This is just like the condition of a `define_insn` that has a standard name.
- 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. GNU CC 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 bitfield (`extv`, `extzv`, and `insv`) operations.

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_rtx (CONST_INT,
                                   VOIDmode, 65535)); ")
```

Note: If the `define_expand` is used to serve a standard binary or unary arithmetic operation or a bitfield 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.

14.14 Splitting Instructions into Multiple Instructions

On machines that have instructions requiring delay slots (see Section 14.15.6 [Delay Slots],

page 212) or that have instructions whose output is not available for multiple cycles (see Section 14.15.7 [Function Units], page 213), the compiler phases that optimize these cases need to be able to move insns into one-cycle delay slots. However, some insns may generate more than one machine instruction. These insns would be unable to be placed into a delay slot.

It is often possible to write 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 `define_split` definition tells the compiler how to split a complex insn into several simpler insns. This spilling will be performed if there is a reason to believe that it might improve instruction or delay slot scheduling. The definition 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`. Any insn matched by a `define_split` must also be matched by a `define_insn` in case it does not need to be split.

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 specified for `define_expand` (see Section 14.13 [Expander Definitions], page 199) and are executed before the new RTL is generated to prepare for the generated code or emit some insns whose pattern is not fixed.

As a simple case, consider the following example from the AMD 29000 machine description, 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]); }")
```

14.15 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.

14.15.1 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 14.15.4 [Attr Example], page 209, for more information on the handling of defaults.

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 enumeral 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 upper case.

- 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`.

14.15.2 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), and `set_attr` (see Section 14.15.3 [Tagging Insns], page 207) expressions.

`(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 non-zero 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 non-zero 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`, `lshift`, `ashift`, `lshiftrt`, and `ashiftrt` expressions.

`const_int` and `symbol_ref` are always valid terms (see Section 14.15.5 [Insn Lengths], page 210,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 14.5 [Output Statement], page 173) 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.

14.15.3 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 14.15.2 [Expressions], page 205) 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 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 `define_insn` and `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. To compute the length of an `asm` insn, the length specified in the `define_asm_attributes` expression is multiplied 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.

14.15.4 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.

14.15.5 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 will be computed as the number of vectors multiplied by the size of each vector.

The following macros can be used to refine the length computation:

`FIRST_INSN_ADDRESS`

When the `length` insn attribute is used, this macro specifies the value to be assigned to the address of the first insn in a function. If not specified, 0 is used.

`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. If updating is required, *insn* must not be a varying-length 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 the value of the `length` attribute, `get_attr_value`, 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)))])
```

14.15.6 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)])
```

14.15.7 Specifying Function Units

On most RISC machines, there are instructions whose results are not available for a specific number of cycles. Common cases are instructions that load data from memory. On many machines, a pipeline stall will result if the data is referenced too soon after the load instruction.

In addition, many newer microprocessors have multiple function units, usually one for integer and one for floating point, and often will incur pipeline stalls when a result that is needed is not yet ready.

The descriptions in this section allow the specification of how much time must elapse between the execution of an instruction and the time when its result is used. It also allows specification of when the execution of an instruction will delay execution of similar instructions due to function unit conflicts.

For the purposes of the specifications in this section, a machine is divided into *function units*, each of which execute a specific class of instructions. Function units that accept one instruction each cycle and allow a result to be used in the succeeding instruction (usually via forwarding) need not be specified. Classic RISC microprocessors will normally have a single function unit, which we can call '`memory`'. The newer "superscalar" processors will often have function units for floating point operations, usually at least a floating point adder and multiplier.

Each usage of a function units by a class of insns is specified with a `define_function_unit` expression, which looks like this:

```
(define_function_unit name multiplicity simultaneity
      test ready-delay busy-delay
     [conflict-list])
```

name is a string giving the name of the function unit.

multiplicity is an integer specifying the number of identical units in the processor. If more than one unit is specified, they will be scheduled independently. Only truly independent units should be counted; a pipelined unit should be specified as a single unit. (The only common example of a machine that has multiple function units for a single instruction class that are truly independent and not pipelined are the two multiply and two increment units of the CDC 6600.)

simultaneity specifies the maximum number of insns that can be executing in each instance of the function unit simultaneously or zero if the unit is pipelined and has no limit.

All `define_function_unit` definitions referring to function unit *name* must have the same name and values for *multiplicity* and *simultaneity*.

test is an attribute test that selects the insns we are describing in this definition. Note that an insn may use more than one function unit and a function unit may be specified in more than one `define_function_unit`.

ready-delay is an integer that specifies the number of cycles after which the result of the instruction can be used without introducing any stalls.

busy-delay is an integer that represents the default cost if an insn is scheduled for this unit while the unit is active with another insn. If *simultaneity* is zero, this specification is ignored. Otherwise, a zero value indicates that these insns execute on *name* in a fully pipelined fashion, even if *simultaneity* is non-zero. A non-zero value indicates that scheduling a new insn on this unit while another is active will incur a cost. A cost of two indicates a single cycle delay. For a normal non-pipelined function unit, *busy-delay* will be twice *ready-delay*.

conflict-list is an optional list giving detailed conflict costs for this unit. If specified, it is a list of condition test expressions which are applied to insns already executing in *name*. For each insn that is in the list, *busy-delay* will be used for the conflict cost, while a value of zero will be used for insns not in the list.

Typical uses of this vector are where a floating point function unit can pipeline either single-

or double-precision operations, but not both, or where a memory unit can pipeline loads, but not stores, etc.

As an example, consider a classic RISC machine where the result of a load instruction is not available for two cycles (a single "delay" instruction is required) and where only one load instruction can be executed simultaneously. This would be specified as:

```
(define_function_unit "memory" 1 1 (eq_attr "type" "load") 2 4)
```

For the case of a floating point function unit that can pipeline either single or double precision, but not both, the following could be specified:

```
(define_function_unit
   "fp" 1 1 (eq_attr "type" "sp_fp") 4 8 (eq_attr "type" "dp_fp")]
(define_function_unit
   "fp" 1 1 (eq_attr "type" "dp_fp") 4 8 (eq_attr "type" "sp_fp")]
```

Note: No code currently exists to avoid function unit conflicts, only data conflicts. Hence *multiplicity*, *simultaneity*, *busy-cost*, and *conflict-list* are currently ignored. When such code is written, it is possible that the specifications for these values may be changed. It has recently come to our attention that these specifications may not allow modeling of some of the newer "superscalar" processors that have insns using multiple pipelined units. These insns will cause a potential conflict for the second unit used during their execution and there is no way of representing that conflict. We welcome any examples of how function unit conflicts work in such processors and suggestions for their representation.

15 Machine Description Macros

In addition to the file '*machine*`.md`', a machine description includes a C header file conventionally given the name '*machine*`.h`'. This 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`'.

15.1 Controlling the Compilation Driver, '`gcc`'

`SWITCH_TAKES_ARG (`*char*`)`

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 to handle the standard options properly. You need not define it unless you wish to add additional options which take arguments.

`WORD_SWITCH_TAKES_ARG (`*name*`)`

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 to handle the standard options properly. You need not define it unless you wish to add additional options which take arguments.

`SWITCHES_NEED_SPACES`

A string-valued C expression which is nonempty if the linker needs a space between the '`-L`' or '`-o`' option and its argument.

If this macro is not defined, the default value is 0.

`CPP_SPEC` A C string constant that tells the GNU CC driver program options to pass to CPP. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the CPP.

Do not define this macro if it does not need to do anything.

`SIGNED_CHAR_SPEC`

A C string constant that tells the GNU CC driver program options to pass to CPP. By default, this macro is defined to pass the option '`-D__CHAR_UNSIGNED__`' to CPP if `char` will be treated as `unsigned char` by `cc1`.

Do not define this macro unless you need to override the default definition.

`CC1_SPEC` A C string constant that tells the GNU CC driver program options to pass to `cc1`. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the `cc1`.

Do not define this macro if it does not need to do anything.

`CC1PLUS_SPEC`

A C string constant that tells the GNU CC driver program options to pass to `cc1plus`. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the `cc1plus`.

Do not define this macro if it does not need to do anything.

`ASM_SPEC` A C string constant that tells the GNU CC driver program options to pass to the assembler. It can also specify how to translate options you give to GNU CC into options for GNU CC 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`

A C string constant that tells the GNU CC 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.

`LINK_SPEC`

A C string constant that tells the GNU CC driver program options to pass to the linker. It can also specify how to translate options you give to GNU CC into options for GNU CC to pass to the linker.

Do not define this macro if it does not need to do anything.

`LIB_SPEC` 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`'.

`STARTFILE_SPEC`

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`

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.

`LINK_LIBGCC_SPECIAL`

Define this macro meaning that `gcc` should find the library '`libgcc.a`' by hand, rather than passing the argument '`-lgcc`' to tell the linker to do the search.

`RELATIVE_PREFIX_NOT_LINKDIR`

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.

`STANDARD_EXEC_PREFIX`

Define this macro as a C string constant if you wish to override the standard choice of '`/usr/local/lib/gcc/`' as the default prefix to try when searching for the executable files of the compiler.

`MD_EXEC_PREFIX`

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.

`STANDARD_STARTFILE_PREFIX`

Define this macro as a C string constant if you wish to override the standard choice of '`/usr/local/lib/gcc/`' as the default prefix to try when searching for startup files such as '`crt0.o`'.

`MD_STARTFILE_PREFIX`

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 the compiler is built as a cross compiler.

`LOCAL_INCLUDE_DIR`

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 use this macro and do not search either '`/usr/local/include`' or its replacement.

`SYSTEM_INCLUDE_DIR`

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`

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 do not use this macro and do not search either '`/usr/include`' or its replacement.

`INCLUDE_DEFAULTS`

Define this macro if you wish to override the entire default search path for include files. The default search path includes `GPLUSPLUS_INCLUDE_DIR`, `GCC_INCLUDE_DIR`, `LOCAL_INCLUDE_DIR`, `SYSTEM_INCLUDE_DIR`, and `STANDARD_INCLUDE_DIR`. In addition, the macros `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 two elements: the directory name (a string constant) and a flag for C++-only directories. Mark the end of the array with a null element. For example, here is the definition used for VMS:

```
#define INCLUDE_DEFAULTS \
{                                       \
  { "GNU_GXX_INCLUDE:", 1},             \
  { "GNU_CC_INCLUDE:", 0},              \
  { "SYS$SYSROOT:[SYSLIB.]", 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`.
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/`'.

15.2 Run-time Target Specification

`CPP_PREDEFINES`

Define this to be a string constant containing '`-D`' options to define the predefined macros that identify this machine and system. These macros will be predefined unless the '`-ansi`' option is specified.

In addition, a parallel set of macros are predefined, whose names are made by appending '`__`' at the beginning and at the end. These '`__`' macros are permitted by the ANSI standard, so they are predefined regardless of whether '`-ansi`' is specified.

For example, on the Sun, one can use the following value:

```
"-Dmc68000 -Dsun -Dunix"
```

The result is to define the macros `__mc68000__`, `__sun__` and `__unix__` unconditionally, and the macros `mc68000`, `sun` and `unix` provided '`-ansi`' is not specified.

`STDC_VALUE`

Define the value to be assigned to the built-in macro `__STDC__`. The default is the value '`1`'.

`extern int target_flags;`

This declaration should be present.

`TARGET_...`

This series of macros is to allow compiler command arguments to enable or disable the use of optional features of the target machine. For example, one machine description serves both the 68000 and the 68020; a command argument tells the compiler whether it should use 68020-only instructions or not. This command argument works by means of a macro `TARGET_68020` that tests a bit in `target_flags`.

Define a macro `TARGET_`*featurename* for each such option. Its definition should test a bit in `target_flags`; for example:

```
#define TARGET_68020 (target_flags & 1)
```

One place where these macros are used is in the condition-expressions of instruction patterns. Note how `TARGET_68020` appears frequently in the 68000 machine description file, '`m68k.md`'. Another place they are used is in the definitions of the other macros in the '*machine*`.h`' file.

`TARGET_SWITCHES`

This macro defines names of command options to set and clear bits in `target_flags`. Its definition is an initializer with a subgrouping for each command option.

Each subgrouping contains a string constant, that defines the option name, and a number, which contains the bits to set in `target_flags`. A negative number says to clear bits instead; the negative of the number is which bits to clear. The actual option name is made by appending '`-m`' to the specified name.

One of the subgroupings should have a null string. The number in this grouping is the default value for `target_flags`. Any target options act starting with that value.

Here is an example which defines '`-m68000`' and '`-m68020`' with opposite meanings, and picks the latter as the default:

```
#define TARGET_SWITCHES \
  { { "68020", 1},      \
    { "68000", -1},     \
    { "", 1}}
```

`TARGET_OPTIONS`

This macro is similar to `TARGET_SWITCHES` but defines names of command options that have values. Its definition is an initializer with a subgrouping for each command option.

Each subgrouping contains a string constant, that defines the fixed part of the option name, and the address of a variable. The variable, type `char *`, is set to the variable part of the given option if the fixed part matches. The actual option name is made by appending '`-m`' to the specified name.

Here is an example which defines '`-mshort-data-`*number*'. If the given option is '`-mshort-data-512`', the variable `m88k_short_data` will be set to the string `"512"`.

```
extern char *m88k_short_data;
#define TARGET_OPTIONS { { "short-data-", &m88k_short_data } }
```

`TARGET_VERSION`

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`

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*`)`

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.

Do not examine `write_symbols` in this macro! The debugging options are not supposed to alter the generated code.

15.3 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 15.2 [Run-time Target], page 221.

`BITS_BIG_ENDIAN`

Define this macro to be the value 1 if the most significant bit in a byte has the lowest number; otherwise define it to be the value zero. This means that bit-field instructions count from the most significant bit. If the machine has no bit-field instructions, this macro is irrelevant.

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`

Define this macro to be 1 if the most significant byte in a word has the lowest number.

`WORDS_BIG_ENDIAN`

Define this macro to be 1 if, in a multiword object, the most significant word has the lowest number.

`BITS_PER_UNIT`

Number of bits in an addressable storage unit (byte); normally 8.

`BITS_PER_WORD`

Number of bits in a word; normally 32.

`MAX_BITS_PER_WORD`

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`

Number of storage units in a word; normally 4.

`POINTER_SIZE`

Width of a pointer, in bits.

`PARM_BOUNDARY`

Normal alignment required for function parameters on the stack, in bits. All stack parameters receive least this much alignment regardless of data type. On most machines, this is the same as the size of an integer.

`STACK_BOUNDARY`

Define this macro if you wish to preserve a certain alignment for the stack pointer. The definition is a C expression for the desired alignment (measured in bits).

If `PUSH_ROUNDING` is not defined, the stack will always be aligned to the specified boundary. If `PUSH_ROUNDING` is defined and specifies a less strict alignment than `STACK_BOUNDARY`, the stack may be momentarily unaligned while pushing arguments.

`FUNCTION_BOUNDARY`

Alignment required for a function entry point, in bits.

`BIGGEST_ALIGNMENT`

Biggest alignment that any data type can require on this machine, in bits.

`BIGGEST_FIELD_ALIGNMENT`

Biggest alignment that any structure field can require on this machine, in bits.

`MAX_OFILE_ALIGNMENT`

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*`)`

If defined, a C expression to compute the alignment for a static variable. *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*`)`

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.

`EMPTY_FIELD_BOUNDARY`

Alignment in bits to be given to a structure bit field that follows an empty field such as `int : 0;`.

`STRUCTURE_SIZE_BOUNDARY`

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`

Define this if instructions will fail to work if given data not on the nominal alignment. If instructions will merely go slower in that case, do not define this macro.

`PCC_BITFIELD_TYPE_MATTERS`

Define this if you wish to imitate the way many other C compilers handle alignment of bitfields and the structures that contain them.

The behavior is that the type written for a bitfield (`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 bitfield 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 bitfield 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.)

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 bitfields 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 bitfields work is to define `STRUCTURE_SIZE_BOUNDARY` as large as `BIGGEST_ALIGNMENT`. Then every structure can be accessed with fullwords.

Unless the machine has bitfield instructions or you define `STRUCTURE_SIZE_BOUNDARY` that way, you must define `PCC_BITFIELD_TYPE_MATTERS` to have a nonzero value.

`BITFIELD_NBYTES_LIMITED`

Like PCC_BITFIELD_TYPE_MATTERS except that its effect is limited to aligning a bitfield within the structure.

`ROUND_TYPE_SIZE (`*struct*`, `*size*`, `*align*`)`

Define this macro as an expression for the overall size of a structure (given by *struct* as a tree node) when the size computed from the fields is *size* and the alignment is *align*.

The default is to round *size* up to a multiple of *align*.

`ROUND_TYPE_ALIGN (`*struct*`, `*computed*`, `*specified*`)`

Define this macro as an expression for the alignment of a structure (given by *struct* 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`

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.

`CHECK_FLOAT_VALUE (`*mode*`, `*value*`)`

A C statement to validate the value *value* (of type `double`) for mode *mode*. This means that you check whether *value* fits within the possible range of values for mode *mode* on this target machine. The mode *mode* is always `SFmode` or `DFmode`.

If *value* is not valid, you should call `error` to print an error message and then assign some valid value to *value*. Allowing an invalid value to go through the compiler can produce incorrect assembler code which may even cause Unix assemblers to crash.

This macro need not be defined if there is no work for it to do.

`TARGET_FLOAT_FORMAT`

A code distinguishing the floating point format of the target machine. There are three defined values:

`IEEE_FLOAT_FORMAT`

This code indicates IEEE floating point. It is the default; there is no need to define this macro when the format is IEEE.

`VAX_FLOAT_FORMAT`

This code indicates the peculiar format used on the Vax.

`UNKNOWN_FLOAT_FORMAT`

This code indicates any other format.

The value of this macro is compared with `HOST_FLOAT_FORMAT` (see Chapter 16 [Config], page 299) to determine whether the target machine has the same format as the host machine. If any other formats are actually in use on supported machines, new codes should be defined for them.

15.4 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`

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`

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`

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.

`LONG_LONG_TYPE_SIZE`

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.

`CHAR_TYPE_SIZE`

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 one quarter of a word. (If this would be less than one storage unit, it is rounded up to one unit.)

`FLOAT_TYPE_SIZE`

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`

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`

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.

`DEFAULT_SIGNED_CHAR`

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`'.

`DEFAULT_SHORT_ENUMS`

A C expression to determine whether to give an `enum` type only as many bytes as it takes to represent the range of possible values of that type. A nonzero value means to do that; a zero value means all `enum` types should be allocated like `int`.

If you don't define the macro, the default is 0.

`SIZE_TYPE`

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`

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`

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`

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`.

`OBJC_INT_SELECTORS`

Define this macro if the type of Objective C selectors should be `int`.

If this macro is not defined, then selectors should have the type `struct objc_selector *`.

`OBJC_NONUNIQUE_SELECTORS`

Define this macro if Objective C selector-references will be made unique by the linker (this is the default). In this case, each selector-reference will be given a separate

assembler label. Otherwise, the selector-references will be gathered into an array with a single assembler label.

`MULTIBYTE_CHARS`

Define this macro to enable support for multibyte characters in the input to GNU CC. This requires that the host system support the ANSI C library functions for converting multibyte characters to wide characters.

`TARGET_BELL`

A C constant expression for the integer value for escape sequence '`\a`'.

`TARGET_BS`
`TARGET_TAB`
`TARGET_NEWLINE`

C constant expressions for the integer values for escape sequences '`\b`', '`\t`' and '`\n`'.

`TARGET_VT`
`TARGET_FF`
`TARGET_CR`

C constant expressions for the integer values for escape sequences '`\v`', '`\f`' and '`\r`'.

15.5 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 15.6 [Register Classes], page 235. For information on using registers to access a stack frame, see Section 15.7.2 [Frame Registers], page 242. For passing values in registers, see Section 15.7.5 [Register Arguments], page 247. For returning values in registers, see Section 15.7.6 [Scalar Return], page 249.

15.5.1 Basic Characteristics of Registers

`FIRST_PSEUDO_REGISTER`

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`

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`

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.

`CONDITIONAL_REGISTER_USAGE`

Zero or more C statements that may conditionally modify two variables `fixed_regs` and `call_used_regs` (both of type `char []`) after they have been initialized from the two preceding macros.

This is necessary in case the fixed or call-clobbered registers depend on target flags.

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` 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.)

`NON_SAVING_SETJMP`

If this macro is defined and has a nonzero value, it means that `setjmp` and related functions fail to save the registers, or that `longjmp` fails to restore them. To compensate, the compiler avoids putting variables in registers in functions that use `setjmp`.

15.5.2 Order of Allocation of Registers

`REG_ALLOC_ORDER`

If defined, an initializer for a vector of integers, containing the numbers of hard registers in the order in which GNU CC 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 allocatable register first.

`ORDER_REGS_FOR_LOCAL_ALLOC`

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 order of registers 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.

15.5.3 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*`)`

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))
```

`HARD_REGNO_MODE_OK (`*regno*`, `*mode*`)`

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
```

It is not necessary for this macro 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. The way to implement that is to define 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 any other hard register for which the mode is OK; and that moving a value into the register and back out not alter it.

Since the same instruction used to move `SImode` 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 and this macro should 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.

`MODES_TIEABLE_P (`*mode1*`, `*mode2*`)`

A C expression that is nonzero if it is desirable to choose register allocation so as to avoid move instructions between a value of mode *mode1* and a value of mode *mode2*.

If `HARD_REGNO_MODE_OK (`*r*`, `*mode1*`)` and `HARD_REGNO_MODE_OK (`*r*`, `*mode2*`)` are ever different for any *r*, then `MODES_TIEABLE_P (`*mode1*`, `*mode2*`)` must be zero.

15.5.4 Handling Leaf Functions

On some machines, a leaf function (i.e., one which make 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. Also, the leaf function may use only those registers for its own variables and temporaries.

GNU CC 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`

A C initializer for a 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 GNU CC 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*`)`

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.

`REG_LEAF_ALLOC_ORDER`

If defined, an initializer for a vector of integers, containing the numbers of hard registers in the order in which the GNU CC should prefer to use them (from most preferred to least) in a leaf function. If this macro is not defined, REG_ALLOC_ORDER is used for both non-leaf and leaf-functions.

Normally, it is necessary for `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE` to treat leaf functions specially. The C variable `leaf_function` is nonzero for such a function.

15.5.5 Registers That Form a Stack

There are special features to handle computers where some of the "registers" form a stack, as in the 80387 coprocessor for the 80386. Stack registers are normally written by pushing onto the stack, and are numbered relative to the top of the stack.

Currently, GNU CC can only handle one group of stack-like registers, and they must be consecutively numbered.

`STACK_REGS`

Define this if the machine has any stack-like registers.

`FIRST_STACK_REG`

The number of the first stack-like register. This one is the top of the stack.

`LAST_STACK_REG`

The number of the last stack-like register. This one is the bottom of the stack.

15.5.6 Obsolete Macros for Controlling Register Usage

These features do not work very well. They exist because they used to be required to generate correct code for the 80387 coprocessor of the 80386. They are no longer used by that machine description and may be removed in a later version of the compiler. Don't use them!

`OVERLAPPING_REGNO_P (`*regno*`)`

If defined, this is a C expression whose value is nonzero if hard register number *regno* is an overlapping register. This means a hard register which overlaps a hard register with a different number. (Such overlap is undesirable, but occasionally it allows a machine to be supported which otherwise could not be.) This macro must return nonzero for *all* the registers which overlap each other. GNU CC can use an overlapping register only in certain limited ways. It can be used for allocation within a basic block, and may be spilled for reloading; that is all.

If this macro is not defined, it means that none of the hard registers overlap each other. This is the usual situation.

`INSN_CLOBBERS_REGNO_P (`*insn*`, `*regno*`)`

If defined, this is a C expression whose value should be nonzero if the insn *insn* has the effect of mysteriously clobbering the contents of hard register number *regno*. By "mysterious" we mean that the insn's RTL expression doesn't describe such an effect.

If this macro is not defined, it means that no insn clobbers registers mysteriously. This is the usual situation; all else being equal, it is best for the RTL expression to show all the activity.

`PRESERVE_DEATH_INFO_REGNO_P (`*regno*`)`

If defined, this is a C expression whose value is nonzero if accurate `REG_DEAD` notes are needed for hard register number *regno* at the time of outputting the assembler code. When this is so, a few optimizations that take place after register allocation and could invalidate the death notes are not done when this register is involved.

You would arrange to preserve death info for a register when some of the code in the machine description which is executed to write the assembler code looks at the death notes. This is necessary only when the actual hardware feature which GNU CC thinks of as a register is not actually a register of the usual sort. (It might, for example, be a hardware stack.)

If this macro is not defined, it means that no death notes need to be preserved. This is the usual situation.

15.6 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`

> An enumeral type that must be defined with all the register class names as enumeral values. `NO_REGS` must be first. `ALL_REGS` must be the last register class, followed by one more enumeral 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`

> The number of distinct register classes, defined as follows:

```
#define N_REG_CLASSES (int) LIM_REG_CLASSES
```

`REG_CLASS_NAMES`

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`

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`'.

`REGNO_REG_CLASS (`*regno*`)`

A C expression whose value is a register class containing hard register *regno*. In general there is more that one such class; choose a class which is *minimal*, meaning that no smaller class also contains the register.

`BASE_REG_CLASS`

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.

`INDEX_REG_CLASS`

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).

`REG_CLASS_FROM_LETTER (`*char*`)`

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`.

`REGNO_OK_FOR_BASE_P (`*num*`)`

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_OK_FOR_INDEX_P (`*num*`)`

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*`)`

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 definition

```
#define PREFERRED_RELOAD_CLASS(X,CLASS) CLASS
```

is safe.

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.

If *x* is a `const_double`, by returning `NO_REGS` you can force *x* into a memory constant. This is useful on certain machines where immediate floating values cannot be loaded into certain kinds of registers.

`LIMIT_RELOAD_CLASS (`*mode*`, `*class*`)`

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*`)`
`SECONDARY_INPUT_RELOAD_CLASS (`*class*`, `*mode*`, `*x*`)`
`SECONDARY_OUTPUT_RELOAD_CLASS (`*class*`, `*mode*`, `*x*`)`

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, you should define `SECONDARY_OUTPUT_RELOAD_CLASS` 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 14.7 [Standard Names], page 183. 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 a intermediate storage. This case often occurs between floating-point and general registers.

`SMALL_REGISTER_CLASSES`

Normally the compiler will avoid choosing spill registers from registers that have been explicitly mentioned in the rtl (these registers are normally those used to pass parameters and return values). However, some machines have so few registers of certain classes that there would not be enough registers to use as spill registers if this were done.

On those machines, you should define `SMALL_REGISTER_CLASSES`. When it is defined, the compiler allows registers explicitly used in the rtl to be used as spill registers but prevents the compiler from extending the lifetime of these registers.

Defining this macro is always safe, but unnecessarily defining this macro will reduce the amount of optimizations that can be performed in some cases. If this macro is not

defined but needs to be, the compiler will run out of reload registers and print a fatal error message.

For most machines, this macro should not be defined.

`CLASS_MAX_NREGS (`*class*`,` *mode*`)`

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.

Three other special macros describe which operands fit which constraint letters.

`CONST_OK_FOR_LETTER_P (`*value*`,` *c*`)`

A C expression that defines the machine-dependent operand constraint letters 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_DOUBLE_OK_FOR_LETTER_P (`*value*`,` *c*`)`

A C expression that defines the machine-dependent operand constraint letters that specify particular ranges of `const_double` values.

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.

`EXTRA_CONSTRAINT (`*value*`,` *c*`)`

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. 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.

15.7 Describing Stack Layout and Calling Conventions

15.7.1 Basic Stack Layout

`STACK_GROWS_DOWNWARD`

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.

`FRAME_GROWS_DOWNWARD`

Define this macro if the addresses of local variable slots are at negative offsets from the frame pointer.

`ARGS_GROW_DOWNWARD`

Define this macro if successive arguments to a function occupy decreasing addresses on the stack.

`STARTING_FRAME_OFFSET`

Offset from the frame pointer to the first local variable slot to be allocated.

If `FRAME_GROWS_DOWNWARD`, the next slot's offset is found by subtracting the length of the first slot from `STARTING_FRAME_OFFSET`. Otherwise, it is found by adding the length of the first slot to the value `STARTING_FRAME_OFFSET`.

`STACK_POINTER_OFFSET`

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*)

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*)

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.

`DYNAMIC_CHAIN_ADDRESS` (*frameaddr*)

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.

15.7.2 Registers That Address the Stack Frame

`STACK_POINTER_REGNUM`

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`

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.

`ARG_POINTER_REGNUM`

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 15.7.3 [Elimination], page 243).

`STATIC_CHAIN_REGNUM`
`STATIC_CHAIN_INCOMING_REGNUM`

Register numbers used for passing a function's static chain pointer. If register windows are used, `STATIC_CHAIN_INCOMING_REGNUM` is the register number as seen by the called function, while `STATIC_CHAIN_REGNUM` is the register number as seen by the calling

function. 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`
`STATIC_CHAIN_INCOMING`

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.

15.7.3 Eliminating Frame Pointer and Arg Pointer

`FRAME_POINTER_REQUIRED`

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.

This macro is ignored and need not be defined if `ELIMINABLE_REGS` is defined.

`INITIAL_FRAME_POINTER_OFFSET (`*depth-var*`)`

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`

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*`)`

A C expression that returns non-zero 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*`)`

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.

`LONGJMP_RESTORE_FROM_STACK`

Define this macro if the `longjmp` function restores registers from the stack frames,

rather than from those saved specifically by `setjmp`. Certain quantities must not be kept in registers across a call to `setjmp` on such machines.

15.7.4 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.

`PROMOTE_PROTOTYPES`

Define this macro if an argument declared as `char` or `short` in a prototype 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.

`PUSH_ROUNDING (`*npushed*`)`

A C expression that is the number of bytes actually pushed onto the stack when an instruction attempts to push *npushed* bytes.

If the target machine does not have a push instruction, do not define this macro. That directs GNU CC to use an alternate strategy: to allocate the entire argument block and then store the arguments into it.

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`

If defined, 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.

It is not proper to define both `PUSH_ROUNDING` and `ACCUMULATE_OUTGOING_ARGS`.

`REG_PARM_STACK_SPACE`

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.

This space can either 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`

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`

Define this macro if `REG_PARM_STACK_SPACE` is defined but stack parameters don't skip the area specified by `REG_PARM_STACK_SPACE`.

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 (`*funtype*`,` *stack-size*`)`

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.

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, *funtype* 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.

15.7.5 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*`)`

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.

The value of the expression should either be 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 usual way to make the ANSI 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 macro `MUST_PASS_IN_STACK (`*mode*`, `*type*`)` 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 non-zero 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.

`FUNCTION_INCOMING_ARG (`*cum*`, `*mode*`, `*type*`, `*named*`)`

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.

`FUNCTION_ARG_PARTIAL_NREGS (`*cum*`, `*mode*`, `*type*`, `*named*`)`

A C expression for the number of words, at the beginning of an argument, 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 *n* 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 of the words 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.

`FUNCTION_ARG_PASS_BY_REFERENCE (`*cum*`, `*mode*`, `*type*`, `*named*`)`

A C expression that indicates when an argument must be passed by reference. If nonzero for an argument, 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.

On machines where `REG_PARM_STACK_SPACE` is not defined, a suitable definition of this macro might be

```
#define FUNCTION_ARG_PASS_BY_REFERENCE(CUM, MODE, TYPE, NAMED)  \
  MUST_PASS_IN_STACK (MODE, TYPE)
```

`CUMULATIVE_ARGS`

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*`)`

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.

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_INCOMING_ARGS (`*cum*`, `*fntype*`, `*libname*`)`

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 argument *libname* exists for symmetry with `INIT_CUMULATIVE_ARGS`. The value passed for *libname* is always 0, since library routines with special calling conventions are never compiled with GNU CC.

`FUNCTION_ARG_ADVANCE (`*cum*`, `*mode*`, `*type*`, `*named*`)`

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*`)`

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.

This macro does not control the *amount* of padding; that is always just enough to reach the next multiple of `FUNCTION_ARG_BOUNDARY`.

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.

`FUNCTION_ARG_BOUNDARY (`*mode*`, `*type*`)`

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*`)`

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.

15.7.6 How Scalar Function Values Are Returned

This section discusses the macros that control returning scalars as values—values that can fit in registers.

`TRADITIONAL_RETURN_FLOAT`

Define this macro if '`-traditional`' should not cause functions declared to return `float` to convert the value to `double`.

`FUNCTION_VALUE (`*valtype*`, `*func*`)`

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).

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 `STRUCT_VALUE_REGNUM` and related macros, below.

`FUNCTION_OUTGOING_VALUE (`*valtype*`, `*func*`)`

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, and `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 `STRUCT_VALUE_REGNUM` and related macros, below.

`LIBCALL_VALUE (`*mode*`)`

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*`)`

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.

15.7.7 How Large Values Are Returnd

When a function value's mode is `BLKmode` (and in some other cases), the value is not returned according to `FUNCTION_VALUE` (see Section 15.7.6 [Scalar Return], page 249). 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.

`RETURN_IN_MEMORY (`*type*`)`

A C expression which can inhibit the returning of certain function values in registers, based on the type of value. A nonzero value says to return the function value in memory, just as large structures are always returned. Here *type* will be a C expression of type `tree`, representing the data type of the value.

Note that values of mode `BLKmode` are returned in memory regardless of this macro. Also, the option '`-fpcc-struct-return`' takes effect regardless of this macro. On most systems, it is possible to leave the macro undefined; this causes a default definition to be used, whose value is the constant 0.

`STRUCT_VALUE_REGNUM`

If the structure value address is passed in a register, then `STRUCT_VALUE_REGNUM` should be the number of that register.

`STRUCT_VALUE`

If the structure value address is not passed in a register, define `STRUCT_VALUE` as an expression returning an RTX for the place where the address is passed. If it returns 0, the address is passed as an "invisible" first argument.

`STRUCT_VALUE_INCOMING_REGNUM`

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.

If the incoming location of the structure value address is in a register, define this macro as the register number.

`STRUCT_VALUE_INCOMING`

If the incoming location is not a register, define `STRUCT_VALUE_INCOMING` as an expression for an RTX for where the called function should find the value. If it should find the value on the stack, define this to create a `mem` which refers to the frame pointer. A definition of 0 means that the address is passed as an "invisible" first argument.

`PCC_STATIC_STRUCT_RETURN`

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. GNU CC does not normally use this convention, even if it is the usual one, but does use it if '`-fpcc-struct-value`' is specified.

Do not define this if the usual system convention is for the caller to pass an address to the subroutine.

15.7.8 Caller-Saves Register Allocation

If you enable it, GNU CC can save registers around function calls. This makes it possible to use call-clobbered registers to hold variables that must live across calls.

`DEFAULT_CALLER_SAVES`

Define this macro if function calls on the target machine do not preserve any registers; in other words, if `CALL_USED_REGISTERS` has 1 for all registers. This macro enables '`-fcaller-saves`' by default. Eventually that option will be enabled by default on all machines and both the option and this macro will be eliminated.

`CALLER_SAVE_PROFITABLE` (*refs*, *calls*)

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*.

15.7.9 Function Entry and Exit

This section describes the macros that output function entry (*prologue*) and exit (*epilogue*) code.

`FUNCTION_PROLOGUE (`*file*`, `*size*`)`

A C compound statement 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. (`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 15.7.3 [Elimination], page 243.

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.

- 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 '`varargs.h`' and '`stdargs.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, in the case that `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 15.7.4 [Stack Arguments], page 245.

Normally, it is necessary for `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE` to treat leaf functions specially. The C variable `leaf_function` is nonzero for such a function.

`EXIT_IGNORE_STACK`

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.

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`.

`FUNCTION_EPILOGUE (`*file*`,` *size*`)`

A C compound statement 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 `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 `FUNCTION_EPILOGUE` at all.

Do not define a pattern named '`return`' if you want the `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 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.

Normally, it is necessary for `FUNCTION_PROLOGUE` and `FUNCTION_EPILOGUE` to treat leaf functions specially. The C variable `leaf_function` is nonzero for such a function. See Section 15.5.4 [Leaf Functions], page 233.

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. `FUNCTION_EPILOGUE` needs to know what was decided. The variable `current_function_pops_args` is the number of bytes of its arguments that a function should pop. See Section 15.7.6 [Scalar Return], page 249.

`DELAY_SLOTS_FOR_EPILOGUE`

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*`)`

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 `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`.

15.7.10 Generating Code for Profiling

`FUNCTION_PROFILER (`*file*`, `*labelno*`)`

A C statement or compound statement to output to *file* some assembler code to call the profiling subroutine `mcount`. Before calling, the assembler code must load the address of a counter variable into a register where `mcount` expects to find the address. The name of this variable is '`LP`' followed by the number *labelno*, so you would generate the name using '`LP%d`' in a `fprintf`.

The details of how the address should be passed to `mcount` are determined by your operating system environment, not by GNU CC. 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.

`PROFILE_BEFORE_PROLOGUE`

Define this macro if the code for function profiling should come before the function prologue. Normally, the profiling code comes after.

`FUNCTION_BLOCK_PROFILER (`*file*`, `*labelno*`)`

A C statement or compound statement to output to *file* some assembler code to initialize basic-block profiling for the current object module. This code should call the subroutine `__bb_init_func` once per object module, passing it as its sole argument the address of a block allocated in the object module.

The name of the block is a local symbol made with this statement:

```
ASM_GENERATE_INTERNAL_LABEL (buffer, "LPBX", 0);
```

Of course, since you are writing the definition of `ASM_GENERATE_INTERNAL_LABEL` as well as that of this macro, you can take a short cut in the definition of this macro and use the name that you know will result.

The first word of this block is a flag which will be nonzero if the object module has already been initialized. So test this word first, and do not call `__bb_init_func` if the flag is nonzero.

`BLOCK_PROFILER (`*file*`, `*blockno*`)`

A C statement or compound statement to increment the count associated with the basic block number *blockno*. Basic blocks are numbered separately from zero within each compilation. The count associated with block number *blockno* is at index *blockno* in a vector of words; the name of this array is a local symbol made with this statement:

```
ASM_GENERATE_INTERNAL_LABEL (buffer, "LPBX", 2);
```

Of course, since you are writing the definition of `ASM_GENERATE_INTERNAL_LABEL` as well as that of this macro, you can take a short cut in the definition of this macro and use the name that you know will result.

15.8 Implementing the Varargs Macros

GNU CC 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.

ANSI '`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 ANSI implementation takes an additional first argument, which is the last named argument of the function. However, it 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 ()`

Use this built-in function to save the argument registers in memory so that the varargs mechanism can access them. Both ANSI and traditional versions of `va_start` must use `__builtin_saveregs`, unless you use `SETUP_INCOMING_VARARGS` (see below) instead.

On some machines, `__builtin_saveregs` is open-coded under the control of the macro `EXPAND_BUILTIN_SAVEREGS`. On other machines, it calls a routine written in assembler language, found in '`libgcc2.c`'.

Regardless of what code is generated for the call to `__builtin_saveregs`, it appears at the beginning of the function, not where the call to `__builtin_saveregs` is written. This is because the registers must be saved before the function starts to use them for its own purposes.

`__builtin_args_info (`*category*`)`

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 ()`

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.

`__builtin_classify_type (`*object*`)`

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:

`EXPAND_BUILTIN_SAVEREGS (`*args*`)`

If defined, is a C expression that 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`.

The argument *args* is a `tree_list` containing the arguments that were passed to `__builtin_saveregs`.

If this macro is not defined, the compiler will output an ordinary call to the library function '`__builtin_saveregs`'.

`SETUP_INCOMING_VARARGS (`*args_so_far*`,` *mode*`,` *type*`,` *pretend_args_size*`,` *second_time*`)`

This macro offers an alternative to using `__builtin_saveregs` and defining the macro `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* is the `CUMULATIVE_ARGS` data structure, containing the values that obtain after processing of 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 macro implementation 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 whose name is supplied as the argument *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, `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 macro `SETUP_INCOMING_VARARGS` should not generate any instructions in this case.

15.9 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 GNU CC 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*`)`

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.

`TRAMPOLINE_SIZE`

A C expression for the size in bytes of the trampoline, as an integer.

`TRAMPOLINE_ALIGNMENT`

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*`)`

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.

`ALLOCATE_TRAMPOLINE (`*fp*`)`

A C expression to allocate run-time space for a trampoline. The expression value should be an RTX representing a memory reference to the space for the trampoline.

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 `FUNCTION_PROLOGUE` and `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. Normally (when `ALLOCATE_TRAMPOLINE` is not defined), 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 macros which describe the shape of the cache.

`INSN_CACHE_SIZE`

The total size in bytes of the cache.

`INSN_CACHE_LINE_WIDTH`

The length in bytes of each cache line. The cache is divided into cache lines which are disjoint slots, each holding a contiguous chunk of data fetched from memory. Each time data is brought into the cache, an entire line is read at once. The data loaded into a cache line is always aligned on a boundary equal to the line size.

`INSN_CACHE_DEPTH`

The number of alternative cache lines that can hold any particular memory location.

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`

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 GNU CC. 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.

15.10 Implicit Calls to Library Routines

`MULSI3_LIBCALL`

A C string constant giving the name of the function to call for multiplication of one signed full-word by another. If you do not define this macro, the default name is used, which is `__mulsi3`, a function defined in '`libgcc.a`'.

`DIVSI3_LIBCALL`

A C string constant giving the name of the function to call for division of one signed full-word by another. If you do not define this macro, the default name is used, which is `__divsi3`, a function defined in '`libgcc.a`'.

`UDIVSI3_LIBCALL`

A C string constant giving the name of the function to call for division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__udivsi3`, a function defined in '`libgcc.a`'.

`MODSI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one signed full-word by another. If you do not define this macro, the default name is used, which is `__modsi3`, a function defined in '`libgcc.a`'.

`UMODSI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__umodsi3`, a function defined in '`libgcc.a`'.

`MULDI3_LIBCALL`

A C string constant giving the name of the function to call for multiplication of one signed double-word by another. If you do not define this macro, the default name is used, which is `__muldi3`, a function defined in '`libgcc.a`'.

`DIVDI3_LIBCALL`

A C string constant giving the name of the function to call for division of one signed double-word by another. If you do not define this macro, the default name is used, which is `__divdi3`, a function defined in '`libgcc.a`'.

`UDIVDI3_LIBCALL`

A C string constant giving the name of the function to call for division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__udivdi3`, a function defined in '`libgcc.a`'.

`MODDI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one signed double-word by another. If you do not define this macro, the default name is used, which is `__moddi3`, a function defined in '`libgcc.a`'.

`UMODDI3_LIBCALL`

A C string constant giving the name of the function to call for the remainder in division of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__umoddi3`, a function defined in '`libgcc.a`'.

`TARGET_MEM_FUNCTIONS`

Define this macro if GNU CC should generate calls to the System V (and ANSI C) library functions `memcpy` and `memset` rather than the BSD functions `bcopy` and `bzero`.

`LIBGCC_NEEDS_DOUBLE`

Define this macro if only `float` arguments cannot be passed to library routines (so they must be converted to `double`). This macro affects both how library calls are generated and how the library routines in '`libgcc1.c`' accept their arguments. It is useful on machines where floating and fixed point arguments are passed differently, such as the i860.

`FLOAT_ARG_TYPE`

Define this macro to override the type used by the library routines to pick up arguments of type `float`. (By default, they use a union of `float` and `int`.)

The obvious choice would be `float`—but that won't work with traditional C compilers that expect all arguments declared as `float` to arrive as `double`. To avoid this conversion, the library routines ask for the value as some other type and then treat it as a `float`.

On some systems, no other type will work for this. For these systems, you must use `LIBGCC_NEEDS_DOUBLE` instead, to force conversion of the values `double` before they are passed.

`FLOATIFY (`*passed-value*`)`

Define this macro to override the way library routines redesignate a `float` argument as a `float` instead of the type it was passed as. The default is an expression which takes the `float` field of the union.

`FLOAT_VALUE_TYPE`

Define this macro to override the type used by the library routines to return values that ought to have type `float`. (By default, they use `int`.)

The obvious choice would be `float`—but that won't work with traditional C compilers gratuitously convert values declared as `float` into `double`.

`INTIFY (`*float-value*`)`

Define this macro to override the way the value of a `float`-returning library routine should be packaged in order to return it. These functions are actually declared to return type `FLOAT_VALUE_TYPE` (normally `int`).

These values can't be returned as type `float` because traditional C compilers would gratuitously convert the value to a `double`.

A local variable named `intify` is always available when the macro `INTIFY` is used. It is a union of a `float` field named `f` and a field named `i` whose type is `FLOAT_VALUE_TYPE` or `int`.

If you don't define this macro, the default definition works by copying the value through that union.

`SItype` Define this macro as the name of the data type corresponding to `SImode` in the system's own C compiler.

You need not define this macro if that type is `int`, as it usually is.

`perform_...`

Define these macros to supply explicit C statements to carry out various arithmetic operations on types `float` and `double` in the library routines in '`libgcc1.c`'. See that file for a full list of these macros and their arguments.

On most machines, you don't need to define any of these macros, because the C compiler that comes with the system takes care of doing them.

`NEXT_OBJC_RUNTIME`

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.

15.11 Addressing Modes

`HAVE_POST_INCREMENT`

Define this macro if the machine supports post-increment addressing.

`HAVE_PRE_INCREMENT`
`HAVE_POST_DECREMENT`
`HAVE_PRE_DECREMENT`

Similar for other kinds of addressing.

`CONSTANT_ADDRESS_P (`*x*`)`

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` 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`

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*`)`

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.

Typically among the subroutines used to define `GO_IF_LEGITIMATE_ADDRESS` are subroutines to check for acceptable registers for various purposes (one for base registers, one for index registers, and so on). 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 macro `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 15.16 [Assembler Format], page 274.

The best way to modify the name string is by adding text to the beginning, with suitable punctuation to prevent any ambiguity. Allocate the new name in `saveable_obstack`. You will have to modify `ASM_OUTPUT_LABELREF` to remove and decode the added text and output the name accordingly.

You can check the information stored here into the `symbol_ref` in the definitions of `GO_IF_LEGITIMATE_ADDRESS` and `PRINT_OPERAND_ADDRESS`.

`REG_OK_FOR_BASE_P (`*x*`)`

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_OK_FOR_INDEX_P (`*x*`)`

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.

`LEGITIMIZE_ADDRESS (`*x*`, `*oldx*`, `*mode*`, `*win*`)`

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 for this macro to do nothing. But often a machine-dependent strategy can generate better code.

`GO_IF_MODE_DEPENDENT_ADDRESS (`*addr*`, `*label*`)`

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.

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*`)`

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.

`LEGITIMATE_PIC_OPERAND_P (x)`

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.

15.12 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`

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`

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*`)`

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 14.15 [Insn Attributes], page 204) named, for example, '`cc`'. This avoids having detailed information about patterns in two places, the '`md`' file and in `NOTICE_UPDATE_CC`.

`EXTRA_CC_MODES`

A list of names to be used for additional modes for condition code values in registers (see Section 14.10 [Jump Patterns], page 192). These names are added to `enum machine_mode` and all have class `MODE_CC`. By convention, they should start with '`CC`' and end with '`mode`'.

You should only define this macro if your machine does not use `cc0` and only if additional modes are required.

`EXTRA_CC_NAMES`

A list of C strings giving the names for the modes listed in `EXTRA_CC_MODES`. For example, the Sparc defines this macro and `EXTRA_CC_MODES` as

```
#define EXTRA_CC_MODES CC_NOOVmode, CCFPmode
#define EXTRA_CC_NAMES "CC_NOOV", "CCFP"
```

This macro is not required if `EXTRA_CC_MODES` is not defined.

`SELECT_CC_MODE (`*op*`,` *x*`)`

Returns a mode from class `MODE_CC` to be used when comparison operation code *op* is applied to rtx *x*. For example, on the Sparc, `SELECT_CC_MODE` is defined as (see see Section 14.10 [Jump Patterns], page 192 for a description of the reason for this definition)

```
#define SELECT_CC_MODE(OP,X) \
  (GET_MODE_CLASS (GET_MODE (X)) == MODE_FLOAT ? CCFPmode    \
   : (GET_CODE (X) == PLUS || GET_CODE (X) == MINUS      \
      || GET_CODE (X) == NEG)       \
   ? CC_NOOVmode : CCmode)
```

This macro is not required if `EXTRA_CC_MODES` is not defined.

15.13 Describing Relative Costs of Operations

These macros let you describe the relative speed of various operations on the target machine.

`CONST_COSTS (`*x*`, `*code*`)`

A part of a C `switch` statement that describes the relative costs of constant RTL expressions. It must contain `case` labels for expression codes `const_int`, `const`, `symbol_ref`, `label_ref` and `const_double`. Each case must ultimately reach a `return` statement to return the relative cost of the use of that kind of constant value in an expression. The cost may depend on the precise value of the constant, which is available for examination in *x*.

code is the expression code—redundant, since it can be obtained with `GET_CODE (`*x*`)`.

`RTX_COSTS (`*x*`, `*code*`)`

Like `CONST_COSTS` but applies to nonconstant RTL expressions. This can be used, for example, to indicate how costly a multiply instruction is. In writing this macro, you can use the construct `COSTS_N_INSNS (`*n*`)` to specify a cost equal to *n* fast instructions.

This macro is optional; do not define it if the default cost assumptions are adequate for the target machine.

`ADDRESS_COST (`*address*`)`

An expression giving the cost of an addressing mode that contains *address*. If not defined, the cost is computed from the *address* expression and the `CONST_COSTS` values.

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.

Similar use of this macro is made in strength reduction of loops.

address need not be valid as an address. In such a case, the cost is not relevant and can be any value; invalid addresses need not be assigned a different cost.

On machines where an address involving more than one register is as cheap as an address computation involving only one register, defining `ADDRESS_COST` to reflect this can cause two registers to be live over a region of code where only one would have been if `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.

This macro will normally either not be defined or be defined as a constant.

`REGISTER_MOVE_COST (`*from*`, `*to*`)`

A C expression for the cost of moving data 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 '`mov`*m*' pattern's constraints do not allow such copying.

`MEMORY_MOVE_COST (`*m*`)`

A C expression for the cost of moving data of mode *m* between a register and memory. A value of 2 is the default; 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.

`BRANCH_COST`

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 GNU CC would ordinarily expect.

`SLOW_BYTE_ACCESS`

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_ZERO_EXTEND`

Define this macro if zero-extension (of a `char` or `short` to an `int`) can be done faster if the destination is a register that is known to be zero.

If you define this macro, you must have instruction patterns that recognize RTL structures like this:

```
(set (strict_low_part (subreg:QI (reg:SI ...) 0)) ...)
```

and likewise for `HImode`.

`SLOW_UNALIGNED_ACCESS`

Define this macro if unaligned accesses have a cost many times greater than aligned accesses, for example if they are emulated in a trap handler.

When this macro is defined, the compiler will act as if `STRICT_ALIGNMENT` were defined when generating code for block moves. This can cause significantly more instructions to be produced. Therefore, do not define this macro if unaligned accesses only add a cycle or two to the time for a memory access.

`DONT_REDUCE_ADDR`

Define this macro to inhibit strength reduction of memory addresses. (On some machines, such strength reduction seems to do harm rather than good.)

`MOVE_RATIO`

The number of scalar move insns which 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.

If you don't define this, a reasonable default is used.

`NO_FUNCTION_CSE`

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.

`NO_RECURSIVE_FUNCTION_CSE`

Define this macro if it is as good or better for a function to call itself with an explicit address than to call an address kept in a register.

15.14 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`

A C string constant for the assembler operation that should precede instructions and read-only data. Normally `".text"` is right.

`DATA_SECTION_ASM_OP`

A C string constant for the assembler operation to identify the following data as writable initialized data. Normally `".data"` is right.

`SHARED_SECTION_ASM_OP`

If defined, a C string constant for the assembler operation to identify the following data as shared data. If not defined, `DATA_SECTION_ASM_OP` will be used.

`INIT_SECTION_ASM_OP`

If defined, a C string constant for the assembler operation to identify the following data as initialization code. If not defined, GNU CC will assume such a section does not exist.

`EXTRA_SECTIONS`

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`

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`.

`READONLY_DATA_SECTION`

On most machines, read-only variables, constants, and jump tables are placed in the text section. If this is not the case on your machine, this macro should be defined to be the name of a function (either `data_section` or a function defined in `EXTRA_SECTIONS`) that switches to the section to be used for read-only items.

If these items should be placed in the text section, this macro should not be defined.

`SELECT_SECTION (`*exp*`,` *reloc*`)`

A C statement or statements to switch 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. Select the section by calling `text_section` or one of the alternatives for other sections.

Do not define this macro if you put all read-only variables and constants in the read-only data section (usually the text section).

`SELECT_RTX_SECTION (`*mode*`, `*rtx*`)`

A C statement or statements to switch to the appropriate section for output of *rtx* in mode *mode*. You can assume that *rtx* 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 `text_section` or one of the alternatives for other sections.

Do not define this macro if you put all constants in the read-only data section.

`JUMP_TABLES_IN_TEXT_SECTION`

Define this macro 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.

`ENCODE_SECTION_INFO (`*decl*`)`

Define this macro if references to a symbol must be treated differently depending on something about the variable or function named by the symbol (such as what section it is in).

The macro definition, if any, is executed immediately after the rtl for *decl* has been created and stored in `DECL_RTL (`*decl*`)`. The value of the rtl will be a `mem` whose address is a `symbol_ref`.

The usual thing for this macro to do is to record a flag in the `symbol_ref` (such as `SYMBOL_REF_FLAG`) or to store a modified name string in the `symbol_ref` (if one bit is not enough information).

15.15 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 `LEGITIMIZE_ADDRESS`, and `PRINT_OPERAND_ADDRESS` as well. 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`

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).

`FINALIZE_PIC`

By generating position-independent code, when two different programs (A and B) share a common library (libC.a), the text of the library can be shared whether or not the library is linked at the same address for both programs. In some of these environments, position-independent code requires not only the use of different addressing modes, but also special code to enable the use of these addressing modes.

The `FINALIZE_PIC` macro serves as a hook to emit these special codes once the function is being compiled into assembly code, but not before. (It is not done before, because in the case of compiling an inline function, it would lead to multiple PIC prologues being included in functions which used inline functions and were compiled to assembly language.)

15.16 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.

15.16.1 The Overall Framework of an Assembler File

`ASM_FILE_START (`*stream*`)`

A C expression which outputs to the stdio stream *stream* some appropriate text to go at the start of an assembler file.

Normally this macro is defined to output a line containing '`#NO_APP`', which is a comment that has no effect on most assemblers but tells the GNU assembler that it can save time by not checking for certain assembler constructs.

On systems that use SDB, it is necessary to output certain commands; see '`attasm.h`'.

`ASM_FILE_END (`*stream*`)`

A C expression which outputs to the stdio stream *stream* some appropriate text to go at the end of an assembler file.

If this macro is not defined, the default is to output nothing special at the end of the file. Most systems don't require any definition.

On systems that use SDB, it is necessary to output certain commands; see '`attasm.h`'.

`ASM_IDENTIFY_GCC (`*file*`)`

A C statement to output assembler commands which will identify the object file as having been compiled with GNU CC (or another GNU compiler).

If you don't define this macro, the string '`gcc_compiled.:`' is output. This string is calculated to define a symbol which, on BSD systems, will never be defined for any other reason. GDB checks for the presence of this symbol when reading the symbol table of an executable.

On non-BSD systems, you must arrange communication with GDB in some other fashion. If GDB is not used on your system, you can define this macro with an empty body.

`ASM_COMMENT_START`

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`

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`

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*`)`

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.

`ASM_OUTPUT_SOURCE_LINE (`*stream*`, `*line*`)`

A C statement to output DBX or SDB debugging information before code for line number *line* of the current source file to the stdio stream *stream*.

This macro need not be defined if the standard form of debugging information for the debugger in use is appropriate.

`ASM_OUTPUT_IDENT (`*stream*`, `*string*`)`

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.

`OBJC_PROLOGUE`

A C statement to output any assembler statements which are required to precede any Objective C object definitions or message sending. The statement is executed only when compiling an Objective C program.

15.16.2 Output of Data

`ASM_OUTPUT_LONG_DOUBLE` (*stream*, *value*)

`ASM_OUTPUT_DOUBLE` (*stream*, *value*)

`ASM_OUTPUT_FLOAT` (*stream*, *value*)

A C statement to output to the stdio stream *stream* an assembler instruction to assemble a floating-point constant of `TFmode`, `DFmode` or `SFmode`, respectively, whose value is *value*. *value* will be a C expression of type `REAL_VALUE__TYPE`, usually `double`.

`ASM_OUTPUT_QUADRUPLE_INT` (*stream*, *exp*)

`ASM_OUTPUT_DOUBLE_INT` (*stream*, *exp*)

`ASM_OUTPUT_INT` (*stream*, *exp*)

`ASM_OUTPUT_SHORT` (*stream*, *exp*)

`ASM_OUTPUT_CHAR` (*stream*, *exp*)

A C statement to output to the stdio stream *stream* an assembler instruction to assemble an integer of 16, 8, 4, 2 or 1 bytes, respectively, whose value is *value*. The argument *exp* will be an RTL expression which represents a constant value. Use '`output_addr_const` (*stream*, *exp*)' to output this value as an assembler expression.

For sizes larger than `UNITS_PER_WORD`, if the action of a macro would be identical to repeatedly calling the macro corresponding to a size of `UNITS_PER_WORD`, once for each word, you need not define the macro.

`ASM_OUTPUT_BYTE` (*stream*, *value*)

A C statement to output to the stdio stream *stream* an assembler instruction to assemble a single byte containing the number *value*.

`ASM_BYTE_OP`

A C string constant giving the pseudo-op to use for a sequence of single-byte constants. If this macro is not defined, the default is `"byte"`.

`ASM_OUTPUT_ASCII` (*stream*, *ptr*, *len*)

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_POOL_PROLOGUE` (*file funname fundecl size*)

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*)

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:

```
ASM_OUTPUT_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_OPEN_PAREN`

`ASM_CLOSE_PAREN`

These macros are defined as C string constant, describing the syntax in the assembler for grouping arithmetic expressions. The following definitions are correct for most assemblers:

```
#define ASM_OPEN_PAREN "("
#define ASM_CLOSE_PAREN ")"
```

15.16.3 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*`)`

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 global variables are output.

`ASM_OUTPUT_ALIGNED_COMMON (`*stream*`, `*name*`, `*size*`, `*alignment*`)`

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.

`ASM_OUTPUT_SHARED_COMMON (`*stream*`, `*name*`, `*size*`, `*rounded*`)`

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_LOCAL (`*stream*`, `*name*`, `*size*`, `*rounded*`)`

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*`)`

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.

`ASM_OUTPUT_SHARED_LOCAL (`*stream*`, `*name*`, `*size*`, `*rounded*`)`

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.

15.16.4 Output and Generation of Labels

`ASM_OUTPUT_LABEL (`*stream*`, `*name*`)`

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.

`ASM_DECLARE_FUNCTION_NAME (`*stream*`, `*name*`, `*decl*`)`

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`).

`ASM_DECLARE_FUNCTION_SIZE (`*stream*`, `*name*`, `*decl*`)`

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.

`ASM_DECLARE_OBJECT_NAME (`*stream*`, `*name*`, `*decl*`)`

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`).

`ASM_GLOBALIZE_LABEL (`*stream*`, `*name*`)`

A C statement (sans semicolon) to output to the stdio stream *stream* some commands that will make the label *name* global; that is, available for reference from other files. 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 global, and a newline.

`ASM_OUTPUT_EXTERNAL (`*stream*`, `*decl*`, `*name*`)`

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.

`ASM_OUTPUT_EXTERNAL_LIBCALL (`*stream*`,` *symref*`)`

A C statement (sans semicolon) to output on *stream* an assembler pseudo-op to declare a library function name external. The name of the library function is given by *symref*, which has type `rtx` and is a `symbol_ref`.

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.

`ASM_OUTPUT_LABELREF (`*stream*`,` *name*`)`

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_LABELREF_AS_INT (`*file*`,` *label*`)`

Define this macro for systems that use the program `collect2`. The definition should be a C statement to output a word containing a reference to the label *label*.

`ASM_GENERATE_INTERNAL_LABEL (`*string*`,` *prefix*`,` *num*`)`

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 `ASM_OUTPUT_LABELREF`, should produce the same output that `ASM_OUTPUT_INTERNAL_LABEL` would produce with the same *prefix* and *num*.

`ASM_OUTPUT_INTERNAL_LABEL (`*stream*`,` *prefix*`,` *num*`)`

A C statement to output to the stdio stream *stream* a label whose name is made from the string *prefix* and the number *num*. These labels are used for internal purposes, and there is no reason for them to appear in the symbol table of the object file. On many systems, the letter '`L`' at the beginning of a label has this effect. The usual definition of this macro is as follows:

```
fprintf (stream, "L%s%d:\n", prefix, num)
```

`ASM_FORMAT_PRIVATE_NAME (`*outvar*`,` *name*`,` *number*`)`

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.

This string will be used as the 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.

`OBJC_GEN_METHOD_LABEL (`*buf*`, `*is_inst*`, `*class_name*`, `*cat_name*`, `*sel_name*`)`

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 a buffer in which to store the name (256 chars max); *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.

15.16.5 Output of Initialization Routines

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 what assembler code is needed to do this.

Here are the two macros you should define if necessary:

`ASM_OUTPUT_CONSTRUCTOR (`*stream*`, `*name*`)`

Define this macro as a C statement to output on the stream *stream* the assembler code to arrange to call the function named *name* at initialization time.

Assume that *name* is the name of a C function generated automatically by the compiler. This function takes no arguments. Use the function `assemble_name` to output the name *name*; this performs any system-specific syntactic transformations such as adding an underscore.

If you don't define this macro, nothing special is output to arrange to call the function. This is correct when the function will be called in some other manner—for example, by means of the `collect` program, which looks through the symbol table to find these functions by their names. If you want to use `collect`, then you need to arrange for it to be built and installed and used on your system.

`ASM_OUTPUT_DESTRUCTOR` (*stream*, *name*)

This is like `ASM_OUTPUT_CONSTRUCTOR` but used for termination functions rather than initialization functions.

15.16.6 Output of Assembler Instructions

`REGISTER_NAMES`

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`

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*)

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_operand`.

If the macro definition does nothing, the instruction is output in the usual way.

`FINAL_PRESCAN_INSN` (*insn*, *opvec*, *noperands*)

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*)

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*)

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*`)`

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 macro `ENCODE_SECTION_INFO` to store the information into the `symbol_ref`, and then check for it here. See Section 15.16 [Assembler Format], page 274.

`DBR_OUTPUT_SEQEND(`*file*`)`

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`
`LOCAL_LABEL_PREFIX`
`USER_LABEL_PREFIX`
`IMMEDIATE_PREFIX`

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_OUTPUT_REG_PUSH (`*stream*`, `*regno*`)`

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*`)`

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.

15.16.7 Output of Dispatch Tables

`ASM_OUTPUT_ADDR_DIFF_ELT` (*stream*, *value*, *rel*)

This macro should be provided on machines where the addresses in a dispatch table are relative to the table's own address.

The definition should be 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 `ASM_OUTPUT_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)
```

`ASM_OUTPUT_ADDR_VEC_ELT` (*stream*, *value*)

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 `ASM_OUTPUT_INTERNAL_LABEL`. For example,

```
fprintf (stream, "\t.word L%d\n", value)
```

`ASM_OUTPUT_CASE_LABEL` (*stream*, *prefix*, *num*, *table*)

Define this if the label before a jump-table needs to be output specially. The first three arguments are the same as for `ASM_OUTPUT_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 `ASM_OUTPUT_INTERNAL_LABEL`.

`ASM_OUTPUT_CASE_END` (*stream*, *num*, *table*)

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.

15.16.8 Assembler Commands for Alignment

`ASM_OUTPUT_ALIGN_CODE` (*file*)

A C expression to output text to align the location counter in the way that is desirable at a point in the code that is reached only by jumping.

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.

`ASM_OUTPUT_LOOP_ALIGN (`*file*`)`

A C expression to output text to align the location counter in the way that is desirable at the beginning of a loop.

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.

`ASM_OUTPUT_SKIP (`*stream*`,` *nbytes*`)`

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`

Define this macro if `ASM_OUTPUT_SKIP` should not be used in the text section because it fails 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*`)`

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`.

15.17 Controlling Debugging Information Format

`DBX_REGISTER_NUMBER (`*regno*`)`

A C expression that returns the DBX register number for the compiler register number *regno*. In simple cases, the value of this expression may 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 GNU CC, 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.

`DBX_DEBUGGING_INFO`

Define this macro if GNU CC should produce debugging output for DBX in response to the '`-g`' option.

`SDB_DEBUGGING_INFO`

Define this macro if GNU CC should produce COFF-style debugging output for SDB in response to the '`-g`' option.

`DWARF_DEBUGGING_INFO`

Define this macro if GNU CC should produce dwarf format debugging output in response to the '`-g`' option.

`DEFAULT_GDB_EXTENSIONS`

Define this macro to control whether GNU CC 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.

`DEBUG_SYMS_TEXT`

Define this macro if all `.stabs` commands should be output while in the text section.

`DEBUGGER_AUTO_OFFSET (`*x*`)`

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*`)`

A C expression that returns the integer offset value for an argument having address *x* (an RTL expression). The nominal offset is *offset*.

`ASM_STABS_OP`

A C string constant naming the assembler pseudo op to use instead of `.stabs` to define an ordinary debugging symbol. If you don't define this macro, `.stabs` is used. This macro applies only to DBX debugging information format.

`ASM_STABD_OP`

A C string constant naming the assembler pseudo op to use instead of `.stabd` to define a debugging symbol whose value is the current location. If you don't define this macro, `.stabd` is used. This macro applies only to DBX debugging information format.

`ASM_STABN_OP`

A C string constant naming the assembler pseudo op to use instead of `.stabn` to define a debugging symbol with no name. If you don't define this macro, `.stabn` is used. This macro applies only to DBX debugging information format.

`PUT_SDB_...`

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`

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_GENERATE_FAKE`

Define this macro to override the usual method of constructing a dummy name for anonymous structure and union types. See '`sdbout.c`' for more information.

`SDB_ALLOW_UNKNOWN_REFERENCES`

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`

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.

`DBX_NO_XREFS`

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`

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`

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`

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_LBRAC_FIRST`

Define this macro if the `N_LBRAC` symbol for a block should precede the debugging information for variables and functions defined in that block. Normally, in DBX format, the `N_LBRAC` symbol comes first.

`DBX_FUNCTION_FIRST`

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_OUTPUT_FUNCTION_END (`*stream*`,` *function*`)`

Define this macro if the target machine requires special output at the end of the debugging information for a function. The definition should be a C statement (sans semicolon) to output the appropriate information to *stream*. *function* is the `FUNCTION_DECL` node for the function.

`DBX_OUTPUT_STANDARD_TYPES (`*syms*`)`

Define this macro if you need to control the order of output of the standard data types at the beginning of compilation. The argument *syms* is a `tree` which is a chain of all the predefined global symbols, including names of data types.

Normally, DBX output starts with definitions of the types for integers and characters, followed by all the other predefined types of the particular language in no particular order.

On some machines, it is necessary to output different particular types first. To do this, define `DBX_OUTPUT_STANDARD_TYPES` to output those symbols in the necessary order. Any predefined types that you don't explicitly output will be output afterward in no particular order.

Be careful not to define this macro so that it works only for C. There are no global variables to access most of the built-in types, because another language may have another set of types. The way to output a particular type is to look through *syms* to see if you can find it. Here is an example:

```
{
  tree decl;
  for (decl = syms; decl; decl = TREE_CHAIN (decl))
    if (!strcmp (IDENTIFIER_POINTER (DECL_NAME (decl)), "long int"))
      dbxout_symbol (decl);
  ...
}
```

This does nothing if the expected type does not exist.

See the function `init_decl_processing` in source file '`c-decl.c`' to find the names to use for all the built-in C types.

`DBX_OUTPUT_MAIN_SOURCE_FILENAME (`*stream*`,` *name*`)`

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.

`DBX_OUTPUT_MAIN_SOURCE_DIRECTORY (`*stream*`,` *name*`)`

A C statement to output DBX debugging information to the stdio stream *stream* which indicates that the current directory during compilation is named *name*.

This macro need not be defined if the standard form of output for DBX debugging information is appropriate.

`DBX_OUTPUT_MAIN_SOURCE_FILE_END (`*stream*`,` *name*`)`

A C statement to output DBX debugging information at the end of compilation of the main source file *name*.

If you don't define this macro, nothing special is output at the end of compilation, which is correct for most machines.

`DBX_OUTPUT_SOURCE_FILENAME (`*stream*`,` *name*`)`

A C statement to output DBX debugging information to the stdio stream *stream* which indicates that file *name* is the current source file. This output is generated each time input shifts to a different source file as a result of '`#include`', the end of an included file, or a '`#line`' command.

This macro need not be defined if the standard form of output for DBX debugging information is appropriate.

15.18 Cross Compilation and Floating Point Format

While all modern machines use 2's 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, the cross compiler cannot safely use the host machine's floating point arithmetic. Therefore, floating point constants must be represented in the target machine's format. This means that the cross

compiler cannot use `atof` to parse a floating point constant; it must have its own special routine to use instead. Also, constant folding must emulate the target machine's arithmetic (or must not be done at all).

The macros in the following table should be defined only if you are cross compiling between different floating point formats.

Otherwise, don't define them. Then default definitions will be set up which use `double` as the data type, `==` to test for equality, etc.

You don't need to worry about how many times you use an operand of any of these macros. The compiler never uses operands which have side effects.

`REAL_VALUE_TYPE`

A macro for the C data type to be used to hold a floating point value in the target machine's format. Typically this would be a `struct` containing an array of `int`.

`REAL_VALUES_EQUAL (`*x*`,` *y*`)`

A macro for a C expression which compares for equality the two values, *x* and *y*, both of type `REAL_VALUE_TYPE`.

`REAL_VALUES_LESS (`*x*`,` *y*`)`

A macro for a C expression which tests whether *x* is less than *y*, both values being of type `REAL_VALUE_TYPE` and interpreted as floating point numbers in the target machine's representation.

`REAL_VALUE_LDEXP (`*x*`,` *scale*`)`

A macro for a C expression which performs the standard library function `ldexp`, but using the target machine's floating point representation. Both *x* and the value of the expression have type `REAL_VALUE_TYPE`. The second argument, *scale*, is an integer.

`REAL_VALUE_FIX (`*x*`)`

A macro whose definition is a C expression to convert the target-machine floating point value *x* to a signed integer. *x* has type `REAL_VALUE_TYPE`.

`REAL_VALUE_UNSIGNED_FIX (`*x*`)`

A macro whose definition is a C expression to convert the target-machine floating point value *x* to an unsigned integer. *x* has type `REAL_VALUE_TYPE`.

`REAL_VALUE_FIX_TRUNCATE (`*x*`)`

A macro whose definition is a C expression to convert the target-machine floating point value *x* to a signed integer, rounding toward 0. *x* has type `REAL_VALUE_TYPE`.

`REAL_VALUE_UNSIGNED_FIX_TRUNCATE (`*x*`)`

A macro whose definition is a C expression to convert the target-machine floating point value *x* to an unsigned integer, rounding toward 0. *x* has type `REAL_VALUE_TYPE`.

`REAL_VALUE_ATOF (`*string*`)`

A macro for a C expression which converts *string*, an expression of type `char *`, into a floating point number in the target machine's representation. The value has type `REAL_VALUE_TYPE`.

`REAL_INFINITY`

Define this macro if infinity is a possible floating point value, and therefore division by 0 is legitimate.

`REAL_VALUE_ISINF (`*x*`)`

A macro for a C expression which determines whether *x*, a floating point value, is infinity. The value has type `int`. By default, this is defined to call `isinf`.

`REAL_VALUE_ISNAN (`*x*`)`

A macro for a C expression which determines whether *x*, a floating point value, is a "nan" (not-a-number). The value has type `int`. By default, this is defined to call `isnan`.

Define the following additional macros if you want to make floating point constant folding work while cross compiling. If you don't define them, cross compilation is still possible, but constant folding will not happen for floating point values.

`REAL_ARITHMETIC (`*output*`, `*code*`, `*x*`, `*y*`)`

A macro for a C statement which calculates an arithmetic operation of the two floating point values *x* and *y*, both of type `REAL_VALUE_TYPE` in the target machine's representation, to produce a result of the same type and representation which is stored in *output* (which will be a variable).

The operation to be performed is specified by *code*, a tree code which will always be one of the following: `PLUS_EXPR`, `MINUS_EXPR`, `MULT_EXPR`, `RDIV_EXPR`, `MAX_EXPR`, `MIN_EXPR`.

The expansion of this macro is responsible for checking for overflow. If overflow happens, the macro expansion should execute the statement `return 0;`, which indicates the inability to perform the arithmetic operation requested.

`REAL_VALUE_NEGATE (`*x*`)`

A macro for a C expression which returns the negative of the floating point value *x*. Both *x* and the value of the expression have type `REAL_VALUE_TYPE` and are in the target machine's floating point representation.

There is no way for this macro to report overflow, since overflow can't happen in the negation operation.

`REAL_VALUE_TRUNCATE (x)`

A macro for a C expression which converts the double-precision floating point value *x* to single-precision.

Both *x* and the value of the expression have type `REAL_VALUE_TYPE` and are in the target machine's floating point representation. However, the value should have an appropriate bit pattern to be output properly as a single-precision floating constant.

There is no way for this macro to report overflow.

`REAL_VALUE_TO_INT (low, high, x)`

A macro for a C expression which converts a floating point value *x* into a double-precision integer which is then stored into *low* and *high*, two variables of type *int*.

`REAL_VALUE_FROM_INT (x, low, high)`

A macro for a C expression which converts a double-precision integer found in *low* and *high*, two variables of type *int*, into a floating point value which is then stored into *x*.

15.19 Miscellaneous Parameters

`PREDICATE_CODES`

Optionally define this if you have added predicates to '*machine*`.c`'. This macro is called within an initializer of an array of structures. The first field in the structure is the name of a predicate and the second field is an arrary of rtl codes. For each predicate, list all rtl codes that can be in expressions matched by the predicate. The list should have a trailing comma. Here is an example of two entries in the list for a typical RISC machine:

```
#define PREDICATE_CODES \
  {"gen_reg_rtx_operand", {SUBREG, REG}},  \
  {"reg_or_short_cint_operand", {SUBREG, REG, CONST_INT}},
```

Defining this macro does not affect the generated code (however, incorrect definitions that omit an rtl code that may be matched by the predicate can cause the compiler to malfunction). Instead, it allows the table built by '`genrecog`' to be more compact and efficient, thus speeding up the compiler. The most important predicates to include in the list specified by this macro are thoses used in the most insn patterns.

`CASE_VECTOR_MODE`

An alias for a machine mode name. This is the machine mode that elements of a jump-table should have.

`CASE_VECTOR_PC_RELATIVE`

Define this macro if jump-tables should contain relative addresses.

`CASE_DROPS_THROUGH`

Define this if control falls through a `case` insn when the index value is out of range. This means the specified default-label is actually ignored by the `case` insn proper.

`BYTE_LOADS_ZERO_EXTEND`

Define this macro if an instruction to load a value narrower than a word from memory into a register also zero-extends the value to the whole register.

`IMPLICIT_FIX_EXPR`

An alias for a tree code that should be used by default for conversion of floating point values to fixed point. Normally, `FIX_ROUND_EXPR` is used.

`FIXUNS_TRUNC_LIKE_FIX_TRUNC`

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.

`EASY_DIV_EXPR`

An alias for a tree code that is the easiest kind of division to compile code for in the general case. It may be `TRUNC_DIV_EXPR`, `FLOOR_DIV_EXPR`, `CEIL_DIV_EXPR` or `ROUND_DIV_EXPR`. These four division operators differ in how they round the result to an integer. `EASY_DIV_EXPR` is used when it is permissible to use any of those kinds of division and the choice should be made on the basis of efficiency.

`MOVE_MAX` The maximum number of bytes that a single instruction can move quickly from memory to memory.

`SHIFT_COUNT_TRUNCATED`

Defining this macro causes the compiler to omit a sign-extend, zero-extend, or bitwise 'and' instruction that truncates the count of a shift operation to a width equal to the number of bits needed to represent the size of the object being shifted. On machines that have instructions that act on bitfields at variable positions, including 'bit test' instructions, defining `SHIFT_COUNT_TRUNCATED` also causes truncation not to be applied to these instructions.

If both types of instructions truncate the count (for shifts) and position (for bitfield operations), or if no variable-position bitfield instructions exist, you should define this macro.

However, on some machines, such as the 80386, truncation only applies to shift operations and not bitfield operations. Do not define `SHIFT_COUNT_TRUNCATED` on such machines. Instead, add patterns to the '`md`' file that include the implied truncation of the shift instructions.

`TRULY_NOOP_TRUNCATION (`*outprec*`, `*inprec*`)`

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.

It is reported that suboptimal code can result when `TRULY_NOOP_TRUNCATION` returns 1 for a pair of sizes for modes for which `MODES_TIEABLE_P` is 0. If this is the case, making `TRULY_NOOP_TRUNCATION` return 0 in such cases may improve things.

`STORE_FLAG_VALUE`

A C expression describing the value returned by a comparison operator and stored by a store-flag instruction ('s*cond*') when the condition is true. This description must apply to *all* the 's*cond*' patterns and all the comparison operators.

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 's*cond*' 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 GNU CC 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 (see Section 8.2 [Bug Reporting], page 104).

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 efficent 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`'.

You need not define `STORE_FLAG_VALUE` if the machine has no store-flag instructions.

`Pmode` An alias for the machine mode for pointers. Normally the definition can be

```
#define Pmode SImode
```

`FUNCTION_MODE`

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`.

`INTEGRATE_THRESHOLD (`*decl*`)`

A C expression for the maximum number of instructions above which the function *decl* should not be inlined. *decl* is a `FUNCTION_DECL` node.

The default definition of this macro is 64 plus 8 times the number of arguments that the function accepts. Some people think a larger threshold should be used on RISC machines.

`SCCS_DIRECTIVE`

Define this if the preprocessor should ignore `#sccs` directives and print no error message.

`HANDLE_PRAGMA (`*stream*`)`

Define this macro if you want to implement any pragmas. If defined, it should be a C statement to be executed when `#pragma` is seen. The argument *stream* is the stdio input stream from which the source text can be read.

It is generally a bad idea to implement new uses of `#pragma`. The only reason to define this macro is for compatibility with other compilers that do support `#pragma` for the sake of any user programs which already use it.

`HAVE_VPRINTF`

Define this if the library function `vprintf` is available on your system.

`DOLLARS_IN_IDENTIFIERS`

Define this macro to control use of the character '`$`' in identifier names. The value should be 0, 1, or 2. 0 means '`$`' is not allowed by default; 1 means it is allowed by default if '`-traditional`' is used; 2 means it is allowed by default provided '`-ansi`' is not used. 1 is the default; there is no need to define this macro in that case.

`DEFAULT_MAIN_RETURN`

Define this macro if the target system expects every program's `main` function to return a standard "success" value by default (if no other value is explicitly returned).

The definition should be a C statement (sans semicolon) to generate the appropriate rtl instructions. It is used only when compiling the end of `main`.

`HAVE_ATEXIT`

Define this if the target system supports the function `atexit` from the ANSI C standard. If this is not defined, and `INIT_SECTION_ASM_OP` is not defined, a default `exit` function will be provided to support C++.

`EXIT_BODY`

Define this if your `exit` function needs to do something besides calling an external function `_cleanup` before terminating with `_exit`. The `EXIT_BODY` macro is only needed if netiher `HAVE_ATEXIT` nor `INIT_SECTION_ASM_OP` are defined.

16 The Configuration File

The configuration file '`xm-`*machine*`.h`' contains macro definitions that describe the machine and system on which the compiler is running, unlike the definitions in '*machine*`.h`', which describe the machine for which the compiler is producing output. Most of the values in '`xm-`*machine*`.h`' are actually the same on all machines that GNU CC runs on, so large parts of all configuration files are identical. But there are some macros that vary:

`USG` Define this macro if the host system is System V.

`VMS` Define this macro if the host system is VMS.

`FAILURE_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits after serious errors.

`SUCCESS_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits without serious errors.

`HOST_WORDS_BIG_ENDIAN`

Defined if the host machine stores words of multi-word values in big-endian order. (GNU CC does not depend on the host byte ordering within a word.)

`HOST_FLOAT_FORMAT`

A numeric code distinguishing the floating point format for the host machine. See `TARGET_FLOAT_FORMAT` in Section 15.3 [Storage Layout], page 223 for the alternatives and default.

`HOST_BITS_PER_CHAR`

A C expression for the number of bits in `char` on the host machine.

`HOST_BITS_PER_SHORT`

A C expression for the number of bits in `short` on the host machine.

`HOST_BITS_PER_INT`

A C expression for the number of bits in `int` on the host machine.

`HOST_BITS_PER_LONG`

A C expression for the number of bits in `long` on the host machine.

`ONLY_INT_FIELDS`

Define this macro to indicate that the host compiler only supports `int` bit fields, rather than other integral types, including `enum`, as do most C compilers.

`EXECUTABLE_SUFFIX`

Define this macro if the host system uses a naming convention for executable files that involves a common suffix (such as, in some systems, '`.exe`') that must be mentioned explicitly when you run the program.

`OBSTACK_CHUNK_SIZE`

A C expression for the size of ordinary obstack chunks. If you don't define this, a usually-reasonable default is used.

`OBSTACK_CHUNK_ALLOC`

The function used to allocate obstack chunks. If you don't define this, `xmalloc` is used.

`OBSTACK_CHUNK_FREE`

The function used to free obstack chunks. If you don't define this, `free` is used.

`USE_C_ALLOCA`

Define this macro to indicate that the compiler is running with the `alloca` implemented in C. This version of `alloca` can be found in the file '`alloca.c`'; to use it, you must also alter the '`Makefile`' variable `ALLOCA`. (This is done automatically for the systems on which we know it is needed.)

If you do define this macro, you should probably do it as follows:

```
#ifndef __GNUC__
#define USE_C_ALLOCA
#else
#define alloca __builtin_alloca
#endif
```

so that when the compiler is compiled with GNU CC it uses the more efficient built-in `alloca` function.

`FUNCTION_CONVERSION_BUG`

Define this macro to indicate that the host compiler does not properly handle converting a function value to a pointer-to-function when it is used in an expression.

In addition, configuration files for system V define `bcopy`, `bzero` and `bcmp` as aliases. Some files define `alloca` as a macro when compiled with GNU CC, in order to take advantage of the benefit of GNU CC's built-in `alloca`.

Index

!

'!' in constraint 180

#

'#' in constraint 182
#pragma 296
#pragma, reason for not using 90

$

$ 90

%

'%' in constraint 182
'%' in template 172

&

'&' in constraint 181

,

' 70

(

(nil) 128

*

'*' in constraint 182
* in template 174

/

'/i' in RTL dump 131
'/s' in RTL dump 131, 132
'/u' in RTL dump 131
'/v' in RTL dump 130

=

'=' in constraint 181

?

'?' in constraint 180
?: extensions 79, 81
?: side effect 81

_

'_' in variables in macros 78
__bb_init_func 256
__builtin_args_info 257
__builtin_classify_type 258
__builtin_next_arg 257
__builtin_saveregs 257

+

'+' in constraint 181

>

'>' in constraint 176

\

\ 173

<

'<' in constraint 176

0

'0' in constraint 177

3

3b1 installation 60

A

abort 117
abs 145
abs and attributes 206
abs*m*2 instruction pattern 185
absolute value 145
access to operands 128
accessors 128
ACCUMULATE_OUTGOING_ARGS 245
ACCUMULATE_OUTGOING_ARGS and stack frames 254

ADDITIONAL_REGISTER_NAMES 282
add*m*3 instruction pattern 184
addr_diff_vec 154
addr_diff_vec, length of 210
addr_vec 154
addr_vec, length of 210
address 172
address constraints 177
address of a label 75
ADDRESS_COST 269
address_operand 177
addressing modes 264
ADJUST_INSN_LENGTH 211
aggregates as return values 251
alignment 91
ALL_REGS 235
Alliant 71
alloca and SunOs 58
alloca vs variable-length arrays 83
alloca, for SunOs 60
alloca, for Unos 62
ALLOCATE_TRAMPOLINE 260
alternate keywords 102
AMD29K options 41
analysis, data flow 123
and 145
and and attributes 206
and, canonicalization of 195
and*m*3 instruction pattern 184
ANSI support 19
apostrophes 70
ARG_POINTER_REGNUM 242
ARG_POINTER_REGNUM and virtual registers 140
arg_pointer_rtx 243
ARGS_GROW_DOWNWARD 241
argument passing 119
arguments in frame (88k) 42
arguments in registers 247
arguments on stack 245
arithmetic libraries 120
arithmetic shift 145
arithmetic simplifications 121
arithmetic, in RTL 143
arrays of length zero 82
arrays of variable length 82
arrays, non-lvalue 84
ashift 145
ashift and attributes 206
ashiftrt 145
ashiftrt and attributes 206
ashl*m*3 instruction pattern 185
ashr*m*3 instruction pattern 185
asm expressions 94
ASM_APP_OFF 275
ASM_APP_ON 275
ASM_BYTE_OP 276
ASM_CLOSE_PAREN 277
ASM_COMMENT_START 275
ASM_DECLARE_FUNCTION_NAME 279
ASM_DECLARE_FUNCTION_SIZE 279
ASM_DECLARE_OBJECT_NAME 279
ASM_FILE_END 274
ASM_FILE_START 274
ASM_FINAL_SPEC 218
ASM_FORMAT_PRIVATE_NAME 280
asm_fprintf 284
ASM_GENERATE_INTERNAL_LABEL 280
ASM_GLOBALIZE_LABEL 279
ASM_IDENTIFY_GCC 274
asm_input 154
ASM_NO_SKIP_IN_TEXT 286
asm_noperands 160
ASM_OPEN_PAREN 277
asm_operands, RTL sharing 166
asm_operands, usage 155
ASM_OUTPUT_ADDR_DIFF_ELT 285
ASM_OUTPUT_ADDR_VEC_ELT 285
ASM_OUTPUT_ALIGN 286
ASM_OUTPUT_ALIGN_CODE 285
ASM_OUTPUT_ALIGNED_COMMON 278
ASM_OUTPUT_ALIGNED_LOCAL 278
ASM_OUTPUT_ASCII 276
ASM_OUTPUT_BYTE 276
ASM_OUTPUT_CASE_END 285
ASM_OUTPUT_CASE_LABEL 285
ASM_OUTPUT_CHAR 276

ASM_OUTPUT_COMMON ... 277
ASM_OUTPUT_CONSTRUCTOR ... 281
ASM_OUTPUT_DESTRUCTOR ... 282
ASM_OUTPUT_DOUBLE ... 276
ASM_OUTPUT_DOUBLE_INT ... 276
ASM_OUTPUT_EXTERNAL ... 279
ASM_OUTPUT_EXTERNAL_LIBCALL ... 279
ASM_OUTPUT_FLOAT ... 276
ASM_OUTPUT_IDENT ... 275
ASM_OUTPUT_INT ... 276
ASM_OUTPUT_INTERNAL_LABEL ... 280
ASM_OUTPUT_LABEL ... 278
ASM_OUTPUT_LABELREF ... 280
ASM_OUTPUT_LABELREF_AS_INT ... 280
ASM_OUTPUT_LOCAL ... 278
ASM_OUTPUT_LONG_DOUBLE ... 276
ASM_OUTPUT_LOOP_ALIGN ... 286
ASM_OUTPUT_OPCODE ... 282
ASM_OUTPUT_POOL_PROLOGUE ... 276
ASM_OUTPUT_QUADRUPLE_INT ... 276
ASM_OUTPUT_REG_POP ... 284
ASM_OUTPUT_REG_PUSH ... 284
ASM_OUTPUT_SHARED_COMMON ... 278
ASM_OUTPUT_SHARED_LOCAL ... 278
ASM_OUTPUT_SHORT ... 276
ASM_OUTPUT_SKIP ... 286
ASM_OUTPUT_SOURCE_FILENAME ... 275
ASM_OUTPUT_SOURCE_LINE ... 275
ASM_OUTPUT_SPECIAL_POOL_ENTRY ... 277
ASM_SPEC ... 218
ASM_STABD_OP ... 287
ASM_STABN_OP ... 287
ASM_STABS_OP ... 287
assemble_name ... 278
assembler format ... 274
assembler instructions ... 94
assembler instructions in RTL ... 155
assembler names for identifiers ... 98
assembler syntax, 88k ... 43
assembly code, invalid ... 103
assigning attribute values to insns ... 207
asterisk in template ... 174
atof ... 290
attr ... 208
attribute expressions ... 205
attribute of variables ... 91
attribute specifications ... 209
attribute specifications example ... 209
attributes, defining ... 204
autoincrement addressing, availability ... 117
autoincrement/decrement addressing ... 176
autoincrement/decrement analysis ... 123

B

backslash ... 173
backtrace for bug reports ... 106
barrier ... 158
BASE_REG_CLASS ... 237
basic blocks ... 123
bcmp ... 300
b*cond* instruction pattern ... 188
bcopy, implicit usage ... 262
BIGGEST_ALIGNMENT ... 224
BIGGEST_FIELD_ALIGNMENT ... 224
Bison parser generator ... 56
bit fields ... 148
bit shift overflow (88k) ... 43
BITFIELD_NBYTES_LIMITED ... 225
BITS_BIG_ENDIAN ... 223
BITS_BIG_ENDIAN, effect on sign_extract ... 148
BITS_PER_UNIT ... 223
BITS_PER_WORD ... 223
bitwise complement ... 145
bitwise exclusive-or ... 145
bitwise inclusive-or ... 145
bitwise logical-and ... 145
BLKmode ... 135
BLKmode, and function return values ... 165
BLOCK_PROFILER ... 256
BRANCH_COST ... 270
break_out_memory_refs ... 266
bug criteria ... 103
bug reports ... 104
bugs ... 103
bugs, known ... 65
BYTE_LOADS_ZERO_EXTEND ... 294

byte_mode ... 137
BYTES_BIG_ENDIAN ... 223
bzero ... 300
bzero, implicit usage ... 262

C

C language extensions ... 73
C language, traditional ... 20
C statements for assembler output ... 173
C_INCLUDE_PATH ... 50
C++_INCLUDE_PATH ... 50
call ... 151
call instruction pattern ... 188
call usage ... 164
call-clobbered register ... 230
call-saved register ... 230
call-used register ... 230
call_insn ... 158
call_insn and '/u' ... 132
call_pop instruction pattern ... 189
CALL_USED_REGISTERS ... 230
call_used_regs ... 230
call_value instruction pattern ... 189
call_value_pop instruction pattern ... 189
CALLER_SAVE_PROFITABLE ... 252
calling conventions ... 241
calling functions in RTL ... 164
CAN_ELIMINATE ... 244
canonicalization of instructions ... 195
case labels in initializers ... 86
case ranges ... 87
case sensitivity and VMS ... 115
CASE_DROPS_THROUGH ... 294
CASE_VECTOR_MODE ... 293
CASE_VECTOR_PC_RELATIVE ... 293
casesi instruction pattern ... 190
cast to a union ... 88
casts as lvalues ... 79
cc_status ... 267
CC_STATUS_MDEP ... 267
CC_STATUS_MDEP_INIT ... 267
cc0 ... 142
cc0, RTL sharing ... 166
cc0_rtx ... 142
CC1_SPEC ... 217
CC1PLUS_SPEC ... 218
CCmode ... 134
change_address ... 183
CHAR_TYPE_SIZE ... 227
CHECK_FLOAT_VALUE ... 226
class definitions, register ... 235
class preference constraints ... 181
CLASS_MAX_NREGS ... 240
classes of RTX codes ... 129
clobber ... 151
cmp*m* instruction pattern ... 186
cmpstr*m* instruction pattern ... 186
code generation conventions ... 47
code generation RTL sequences ... 199
code motion ... 123
code_label ... 158
code_label and '/i' ... 132
CODE_LABEL_NUMBER ... 158
codes, RTL expression ... 127
combiner pass ... 141
command options ... 15
common subexpression elimination ... 123
compare ... 143
compare, canonicalization of ... 195
compilation in a separate directory ... 59
compiler bugs, reporting ... 104
compiler passes and files ... 121
compiler version, specifying ... 37
COMPILER_PATH ... 50
complement, bitwise ... 145
compound expressions as lvalues ... 79
computed gotos ... 75
computing the length of an insn ... 210
cond ... 147
cond and attributes ... 205
condition code register ... 142
condition code status ... 267
condition codes ... 146
conditional expressions as lvalues ... 79
conditional expressions, extensions ... 81
CONDITIONAL_REGISTER_USAGE ... 230

conditions, in patterns ... 168
configuration file ... 299
conflicting types ... 65
const applied to function ... 89
CONST_CALL_P ... 132
CONST_COSTS ... 269
const_double ... 137
const_double, RTL sharing ... 166
CONST_DOUBLE_CHAIN ... 138
CONST_DOUBLE_LOW ... 138
CONST_DOUBLE_MEM ... 138
CONST_DOUBLE_OK_FOR_LETTER_P ... 240
const_int ... 137
const_int and attribute tests ... 206
const_int and attributes ... 205
const_int, RTL sharing ... 166
CONST_OK_FOR_LETTER_P ... 240
const_string ... 138
const_string and attributes ... 205
const_true_rtx ... 137
const0_rtx ... 137
CONST0_RTX ... 138
const1_rtx ... 137
CONST1_RTX ... 138
const2_rtx ... 137
CONST2_RTX ... 138
constant folding ... 121
constant folding and floating point ... 292
constant propagation ... 123
CONSTANT_ADDRESS_P ... 264
CONSTANT_ALIGNMENT ... 224
CONSTANT_P ... 264
CONSTANT_POOL_ADDRESS_P ... 132
constants in constraints ... 176
constm1_rtx ... 137
constraint modifier characters ... 181
constraint, matching ... 177
constraints ... 175
constructor expressions ... 85
constructors, output of ... 281
contributors ... 9
controlling register usage ... 230
controlling the compilation driver ... 217
conventions, run-time ... 119
conversions ... 148
Convex options ... 40
copy_rtx_if_shared ... 166
core dump ... 103
costs of instructions ... 269
COSTS_N_INSNS ... 269
CPP_PREDEFINES ... 221
CPP_SPEC ... 217
cross compilation and floating point ... 290
cross compiling ... 37
cross-jumping ... 125
CUMULATIVE_ARGS ... 248
current_function_epilogue_delay_list ... 255
current_function_outgoing_args_size ... 245
current_function_pops_args ... 255
current_function_pretend_args_size ... 253

D

'd' in constraint ... 176
data flow analysis ... 123
DATA_ALIGNMENT ... 224
data_section ... 272
DATA_SECTION_ASM_OP ... 272
DBR_OUTPUT_SEQEND ... 284
dbr_sequence_length ... 284
DBX ... 71
DBX_CONTIN_CHAR ... 288
DBX_CONTIN_LENGTH ... 288
DBX_DEBUGGING_INFO ... 286
DBX_FUNCTION_FIRST ... 289
DBX_LBRAC_FIRST ... 289
DBX_NO_XREFS ... 288
DBX_OUTPUT_FUNCTION_END ... 289
DBX_OUTPUT_MAIN_SOURCE_DIRECTORY ... 290
DBX_OUTPUT_MAIN_SOURCE_FILE_END ... 290
DBX_OUTPUT_MAIN_SOURCE_FILENAME ... 290
DBX_OUTPUT_SOURCE_FILENAME ... 290
DBX_OUTPUT_STANDARD_TYPES ... 289
DBX_REGISTER_NUMBER ... 286
DBX_STATIC_STAB_DATA_SECTION ... 288
DCmode ... 135
De Morgan's law ... 195

dead code ... 122
dead_or_set_p ... 197
deallocating variable length arrays ... 82
death notes ... 235
debug_rtx ... 107
DEBUG_SYMS_TEXT ... 287
DEBUGGER_ARG_OFFSET ... 287
DEBUGGER_AUTO_OFFSET ... 287
debugging information generation ... 125
debugging information options ... 27
debugging, 88k OCS ... 42
declaration scope ... 70
declarations inside expressions ... 73
declarations, RTL ... 150
declaring attributes of functions ... 89
DEFAULT_CALLER_SAVES ... 252
DEFAULT_GDB_EXTENSIONS ... 287
DEFAULT_MAIN_RETURN ... 296
DEFAULT_SHORT_ENUMS ... 228
DEFAULT_SIGNED_CHAR ... 227
define_asm_attributes ... 208
define_attr ... 204
define_delay ... 212
define_expand ... 200
define_function_unit ... 213
define_insn ... 167
define_insn example ... 168
define_peephole ... 200
defining attributes and their values ... 204
defining jump instruction patterns ... 192
defining peephole optimizers ... 196
defining RTL sequences for code generation ... 199
delay slots, defining ... 212
DELAY_SLOTS_FOR_EPILOGUE ... 255
delayed branch scheduling ... 125
dependencies for make as output ... 51
dependencies, make ... 33
DEPENDENCIES_OUTPUT ... 51
Dependent Patterns ... 191
destructors, output of ... 281
DFmode ... 134
dialect options ... 19
digits in constraint ... 177
DImode ... 134
directory options ... 36
disabling certain registers ... 230
dispatch table ... 285
div ... 144
div and attributes ... 206
DIVDI3_LIBCALL ... 262
divide instruction, 88k ... 43
division ... 144
div*m*3 instruction pattern ... 184
divmod*m*4 instruction pattern ... 185
DIVSI3_LIBCALL ... 261
dollar signs in identifier names ... 90
DOLLARS_IN_IDENTIFIERS ... 296
DONE ... 201
DONT_REDUCE_ADDR ... 271
double-word arithmetic ... 81
DOUBLE_TYPE_SIZE ... 227
downward funargs ... 76
driver ... 217
DWARF_DEBUGGING_INFO ... 287
DYNAMIC_CHAIN_ADDRESS ... 242

E

'E' in constraint ... 176
EASY_DIV_EXPR ... 294
ELIGIBLE_FOR_EPILOGUE_DELAY ... 255
ELIMINABLE_REGS ... 244
empty constraints ... 182
EMPTY_FIELD_BOUNDARY ... 225
ENCODE_SECTION_INFO ... 273
ENCODE_SECTION_INFO and address validation ... 265
ENCODE_SECTION_INFO usage ... 284
ENDFILE_SPEC ... 218
endianness ... 117
enum machine_mode ... 134
enum reg_class ... 236
environment variables ... 49
epilogue ... 253
eq ... 147
eq and attributes ... 206
eq_attr ... 206
equal ... 147

`error` ... 226
exclamation point ... 180
exclusive-or, bitwise ... 145
`EXECUTABLE_SUFFIX` ... 299
exit status and VMS ... 114
`EXIT_BODY` ... 297
`EXIT_IGNORE_STACK` ... 254
`EXPAND_BUILTIN_SAVEREGS` ... 258
expander definitions ... 199
explicit register variables ... 99
`expr_list` ... 164
expression codes ... 127
expressions containing statements ... 73
expressions, compound, as lvalues ... 79
expressions, conditional, as lvalues ... 79
expressions, constructor ... 85
extended `asm` ... 94
`extend`*mn* instruction pattern ... 187
extensible constraints ... 177
extensions, `?:` ... 79, 81
extensions, C language ... 73
`extern int target_flags` ... 221
external declaration scope ... 70
`EXTRA_CC_MODES` ... 268
`EXTRA_CC_NAMES` ... 268
`EXTRA_CONSTRAINT` ... 240
`EXTRA_SECTION_FUNCTIONS` ... 272
`EXTRA_SECTIONS` ... 272
`extv` instruction pattern ... 187
`extzv` instruction pattern ... 187

F

'`F`' in constraint ... 176
`FAIL` ... 201
`FAILURE_EXIT_CODE` ... 299
fatal signal ... 103
features, optional, in system conventions ... 221
`ffs` ... 146
`ffs`*m*2 instruction pattern ... 185
file name suffix ... 18
file names ... 34
files and passes of the compiler ... 121
final pass ... 125
`FINAL_PRESCAN_INSN` ... 283
`final_scan_insn` ... 255
`final_sequence` ... 284
`FINALIZE_PIC` ... 274
`FIRST_INSN_ADDRESS` ... 211
`FIRST_PARM_OFFSET` ... 241
`FIRST_PARM_OFFSET` and virtual registers ... 140
`FIRST_PSEUDO_REGISTER` ... 229
`FIRST_STACK_REG` ... 234
`FIRST_VIRTUAL_REGISTER` ... 140
`fix` ... 149
`fix_trunc`*mn*2 instruction pattern ... 187
fixed register ... 230
`FIXED_REGISTERS` ... 230
`fixed_regs` ... 230
`fix`*mn*2 instruction pattern ... 186
`FIXUNS_TRUNC_LIKE_FIX_TRUNC` ... 294
`fixuns_trunc`*mn*2 instruction pattern ... 187
`fixuns`*mn*2 instruction pattern ... 186
flags in RTL expression ... 130
`float` ... 149
`float` as function value type ... 70
`FLOAT_ARG_TYPE` ... 262
`float_extend` ... 149
`float_truncate` ... 149
`FLOAT_TYPE_SIZE` ... 227
`FLOAT_VALUE_TYPE` ... 263
`FLOATIFY` ... 263
floating point format and cross compilation ... 290
`float`*mn*2 instruction pattern ... 186
`floatuns`*mn*2 instruction pattern ... 186
`force_reg` ... 183
frame layout ... 241
`FRAME_GROWS_DOWNWARD` ... 241
`FRAME_GROWS_DOWNWARD` and virtual registers ... 140
`frame_pointer_needed` ... 253
`FRAME_POINTER_REGNUM` ... 242
`FRAME_POINTER_REGNUM` and virtual registers ... 140
`FRAME_POINTER_REQUIRED` ... 243
`frame_pointer_rtx` ... 243
`fscanf`, and constant strings ... 69
`ftrunc`*m*2 instruction pattern ... 187
function attributes ... 89

function call conventions ... 119
function entry and exit ... 253
function pointers, arithmetic ... 84
function units, for scheduling ... 213
function, size of pointer to ... 84
function-call insns ... 164
FUNCTION_ARG ... 247
FUNCTION_ARG_ADVANCE ... 249
FUNCTION_ARG_BOUNDARY ... 249
FUNCTION_ARG_PADDING ... 249
FUNCTION_ARG_PARTIAL_NREGS ... 247
FUNCTION_ARG_PASS_BY_REFERENCE ... 248
FUNCTION_ARG_REGNO_P ... 249
FUNCTION_BLOCK_PROFILER ... 256
FUNCTION_BOUNDARY ... 224
FUNCTION_CONVERSION_BUG ... 300
FUNCTION_EPILOGUE ... 254
FUNCTION_EPILOGUE and trampolines ... 260
FUNCTION_INCOMING_ARG ... 247
FUNCTION_MODE ... 296
FUNCTION_OUTGOING_VALUE ... 250
FUNCTION_PROFILER ... 255
FUNCTION_PROLOGUE ... 253
FUNCTION_PROLOGUE and trampolines ... 260
FUNCTION_VALUE ... 250
FUNCTION_VALUE_REGNO_P ... 250
functions that have no side effects ... 89
functions that never return ... 89
functions, leaf ... 233

G

'g' in constraint ... 177
'G' in constraint ... 176
GCC_EXEC_PREFIX ... 50
ge ... 147
ge and attributes ... 206
gencodes ... 122
genconfig ... 125
general_operand ... 170
GENERAL_REGS ... 235
generalized lvalues ... 79
generating assembler output ... 173
generating insns ... 169
genflags ... 122
genflags, crash on Sun 4 ... 66
get_attr ... 206
get_attr_value ... 211
GET_CLASS_NARROWEST_MODE ... 137
GET_CODE ... 127
get_frame_size ... 243
get_insns ... 157
get_last_insn ... 157
GET_MODE ... 136
GET_MODE_ALIGNMENT ... 136
GET_MODE_BITSIZE ... 136
GET_MODE_CLASS ... 136
GET_MODE_MASK ... 136
GET_MODE_NAME ... 136
GET_MODE_NUNITS ... 137
GET_MODE_SIZE ... 136
GET_MODE_UNIT_SIZE ... 137
GET_MODE_WIDER_MODE ... 136
GET_RTX_CLASS ... 129
GET_RTX_FORMAT ... 129
GET_RTX_LENGTH ... 129
geu ... 147
geu and attributes ... 206
global offset table ... 48
global register after longjmp ... 100
global register allocation ... 124
global register variables ... 99
GLOBALDEF ... 112
GLOBALREF ... 112
GLOBALVALUEDEF ... 112
GLOBALVALUEREF ... 112
GNU CC and portability ... 117
GNU CC command options ... 15
GNU extensions to the C language ... 73
GO_IF_LEGITIMATE_ADDRESS ... 264
GO_IF_MODE_DEPENDENT_ADDRESS ... 266
goto with computed label ... 75
gp-relative references (MIPS) ... 47
gprof ... 28
greater than ... 147
grouping options ... 15
gt ... 147

gt and attributes ... 206
gtu ... 147
gtu and attributes ... 206

H

'H' in constraint ... 176
HANDLE_PRAGMA ... 296
hard registers ... 139
HARD_REGNO_MODE_OK ... 231
HARD_REGNO_NREGS ... 231
hardware models and configurations, specifying ... 38
HAVE_ATEXIT ... 297
HAVE_POST_DECREMENT ... 264
HAVE_POST_INCREMENT ... 264
HAVE_PRE_DECREMENT ... 264
HAVE_PRE_INCREMENT ... 264
HAVE_VPRINTF ... 296
header files and VMS ... 111
high ... 139
HImode ... 134
HImode, in insn ... 159
HOST_BITS_PER_CHAR ... 299
HOST_BITS_PER_INT ... 299
HOST_BITS_PER_LONG ... 299
HOST_BITS_PER_SHORT ... 299
HOST_FLOAT_FORMAT ... 299
HOST_WORDS_BIG_ENDIAN ... 299

I

'i' in constraint ... 176
'I' in constraint ... 176
IBM RS/6000 Options ... 44
IBM RT options ... 44
IBM RT PC ... 71
identifier names, dollar signs in ... 90
identifiers, names in assembler code ... 98
identifying source, compiler (88k) ... 42
IEEE_FLOAT_FORMAT ... 226
if_then_else ... 147
if_then_else and attributes ... 205
if_then_else usage ... 151
immediate_operand ... 170
IMMEDIATE_PREFIX ... 284
IMPLICIT_FIX_EXPR ... 294
in_data ... 272
in_struct ... 133
in_struct, in code_label ... 132
in_struct, in insn ... 132
in_struct, in label_ref ... 131
in_struct, in mem ... 131
in_struct, in reg ... 131
in_text ... 272
include files and VMS ... 111
INCLUDE_DEFAULTS ... 219
inclusive-or, bitwise ... 145
incompatibilities of GNU CC ... 69
increment operators ... 103
INDEX_REG_CLASS ... 237
indirect_jump instruction pattern ... 190
INIT_CUMULATIVE_ARGS ... 248
INIT_CUMULATIVE_INCOMING_ARGS ... 248
INIT_SECTION_ASM_OP ... 272
INITIAL_ELIMINATION_OFFSET ... 244
INITIAL_FRAME_POINTER_OFFSET ... 243
initialization routines ... 281
initializations in expressions ... 85
INITIALIZE_TRAMPOLINE ... 260
initializers with labeled elements ... 86
initializers, non-constant ... 84
inline functions ... 93
inline functions, omission of ... 93
inline, automatic ... 122
insn ... 157
insn and '/i' ... 132
insn and '/s' ... 132
insn and '/u' ... 132
insn attributes ... 204
insn canonicalization ... 195
insn lengths, computing ... 210
insn splitting ... 202
insn-attr.h ... 204
INSN_ANNULLED_BRANCH_P ... 132
INSN_CACHE_DEPTH ... 261
INSN_CACHE_LINE_WIDTH ... 260
INSN_CACHE_SIZE ... 260
INSN_CLOBBERS_REGNO_P ... 234

INSN_CODE ... 160
INSN_DELETED_P ... 131
INSN_FROM_TARGET_P ... 132
insn_list ... 164
INSN_UID ... 156
insns ... 156
insns, generating ... 169
insns, recognizing ... 169
installation on SCO systems ... 61
installation trouble ... 65
installing GNU CC ... 53
installing GNU CC on the 3b1 ... 60
installing GNU CC on the Sun ... 60
installing GNU CC on Unos ... 61
installing GNU CC on VMS ... 62
instruction attributes ... 204
instruction combination ... 124
instruction patterns ... 167
instruction recognizer ... 126
instruction scheduling ... 124
instruction splitting ... 202
insv instruction pattern ... 187
INT_TYPE_SIZE ... 227
INTEGRATE_THRESHOLD ... 296
integrated ... 133
integrated, in insn ... 131
integrated, in reg ... 131
integrating function code ... 93
Interdependence of Patterns ... 191
interfacing to GNU CC output ... 119
INTIFY ... 263
invalid assembly code ... 103
invalid input ... 104
ior ... 145
ior and attributes ... 206
ior, canonicalization of ... 195
ior*m*3 instruction pattern ... 184
isinf ... 292
isnan ... 292

J

jump instruction patterns ... 192
jump instructions and set ... 151
jump optimization ... 122
jump threading ... 123
jump_insn ... 157
JUMP_LABEL ... 158
JUMP_TABLES_IN_TEXT_SECTION ... 273

K

keywords, alternate ... 102
known causes of trouble ... 65

L

LABEL_NUSES ... 158
LABEL_OUTSIDE_LOOP_P ... 131
LABEL_PRESERVE_P ... 132
label_ref ... 138
label_ref and '/s' ... 131
label_ref, RTL sharing ... 166
labeled elements in initializers ... 86
labels as values ... 75
language dialect options ... 19
large bit shifts (88k) ... 43
large return values ... 251
LAST_STACK_REG ... 234
LAST_VIRTUAL_REGISTER ... 140
ldexp ... 291
le ... 147
le and attributes ... 206
leaf functions ... 233
leaf_function ... 233
leaf_function_p ... 189
LEAF_REG_REMAP ... 233
LEAF_REGISTERS ... 233
left rotate ... 145
left shift ... 145
LEGITIMATE_CONSTANT_P ... 266
LEGITIMATE_PIC_OPERAND_P ... 266
LEGITIMIZE_ADDRESS ... 266
length-zero arrays ... 82
less than ... 147
less than or equal ... 147
leu ... 147
leu and attributes ... 206
LIB_SPEC ... 218

LIBCALL_VALUE ... 250
'libgcc.a' ... 261
LIBGCC_NEEDS_DOUBLE ... 262
Libraries ... 35
library subroutine names ... 261
LIBRARY_PATH ... 50
LIMIT_RELOAD_CLASS ... 238
link options ... 34
LINK_LIBGCC_SPECIAL ... 218
LINK_SPEC ... 218
lo_sum ... 143
load address instruction ... 177
local labels ... 74
local register allocation ... 124
local variables in macros ... 78
local variables, specifying registers ... 101
LOCAL_INCLUDE_DIR ... 219
LOCAL_LABEL_PREFIX ... 284
LOG_LINKS ... 160
logical shift ... 145
logical-and, bitwise ... 145
long long data types ... 81
LONG_DOUBLE_TYPE_SIZE ... 227
LONG_LONG_TYPE_SIZE ... 227
LONG_TYPE_SIZE ... 227
longjmp ... 100
longjmp and automatic variables ... 20, 119
longjmp incompatibilities ... 69
longjmp warnings ... 23
LONGJMP_RESTORE_FROM_STACK ... 244
loop optimization ... 123
lshift ... 145
lshift and attributes ... 206
lshiftrt ... 145
lshiftrt and attributes ... 206
lshl*m*3 instruction pattern ... 185
lshr*m*3 instruction pattern ... 185
lt ... 147
lt and attributes ... 206
ltu ... 147
lvalues, generalized ... 79

M

'm' in constraint ... 175
M680x0 options ... 39
M88k options ... 42
machine dependent options ... 38
machine description macros ... 217
machine descriptions ... 167
machine mode conversions ... 148
machine modes ... 134
macros containing asm ... 96
macros, inline alternative ... 93
macros, local labels ... 74
macros, local variables in ... 78
macros, machine description ... 217
macros, statements in expressions ... 73
macros, types of arguments ... 78
main and the exit status ... 114
make ... 33
make_safe_from ... 202
match_dup ... 170
match_dup and attributes ... 210
match_operand ... 169
match_operand and attributes ... 206
match_operator ... 170
match_scratch ... 170
matching constraint ... 177
matching operands ... 173
math libraries ... 120
math, in RTL ... 143
MAX_BITS_PER_WORD ... 223
MAX_FIXED_MODE_SIZE ... 226
MAX_OFILE_ALIGNMENT ... 224
MAX_REGS_PER_ADDRESS ... 264
max*m*3 instruction pattern ... 184
mcount ... 256
MD_EXEC_PREFIX ... 219
MD_STARTFILE_PREFIX ... 219
mem ... 143
mem and '/s' ... 131
mem and '/u' ... 131
mem and '/v' ... 130
mem, RTL sharing ... 166
MEM_IN_STRUCT_P ... 131

MEM_VOLATILE_P 130
memcpy, implicit usage 262
memory reference, nonoffsettable 179
memory references in constraints 175
MEMORY_MOVE_COST 270
memset, implicit usage 262
messages, warning 22
middle-operands, omitted 81
min*m*3 instruction pattern 184
minus 143
minus and attributes 206
minus, canonicalization of 195
MIPS options 45
mktemp, and constant strings 69
mod 144
mod and attributes 206
MODDI3_LIBCALL 262
mode classes 135
MODE_CC 136
MODE_COMPLEX_FLOAT 135
MODE_COMPLEX_INT 135
MODE_FLOAT 135
MODE_FUNCTION 136
MODE_INT 135
MODE_PARTIAL_INT 135
MODE_RANDOM 136
MODES_TIEABLE_P 232
modifiers in constraints 181
mod*m*3 instruction pattern 184
MODSI3_LIBCALL 261
MOVE_MAX 294
MOVE_RATIO 271
mov*m* instruction pattern 183
movstrict*m* instruction pattern 184
movstr*m* instruction pattern 186
MULDI3_LIBCALL 262
mulhisi3 instruction pattern 185
mul*m*3 instruction pattern 184
mulqihi3 instruction pattern 185
MULSI3_LIBCALL 261
mulsidi3 instruction pattern 185
mult 144
mult and attributes 206
mult, canonicalization of 195
MULTIBYTE_CHARS 229
multiple alternative constraints 180
multiplication 144
multiprecision arithmetic 81
MUST_PASS_IN_STACK, and FUNCTION_ARG 247

N

'n' in constraint 176
N_REG_CLASSES 236
name augmentation 115
named patterns and conditions 168
names used in assembler code 98
names, pattern 183
naming types 78
ne 147
ne and attributes 206
neg 144
neg and attributes 206
neg, canonicalization of 195
neg*m*2 instruction pattern 185
nested functions 76
nested functions, trampolines for 259
next_cc0_user 194
NEXT_INSN 157
NEXT_OBJC_RUNTIME 264
nil 128
no constraints 182
no-op move instructions 125
NO_FUNCTION_CSE 271
NO_RECURSIVE_FUNCTION_CSE 271
NO_REGS 235
non-constant initializers 84
non-static inline function 93
NON_SAVING_SETJMP 230
nonoffsettable memory reference 179
nop instruction pattern 189
not 145
not and attributes 206
not equal 147
not using constraints 182
not, canonicalization of 195
note 158

NOTE_INSN_BLOCK_BEG ... 159
NOTE_INSN_BLOCK_END ... 159
NOTE_INSN_DELETED ... 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_INSN_SETJMP ... 159
NOTE_LINE_NUMBER ... 158
NOTE_SOURCE_FILE ... 158
NOTICE_UPDATE_CC ... 267
NUM_MACHINE_MODES ... 136

O

'o' in constraint ... 175
OBJC_GEN_METHOD_LABEL ... 281
OBJC_INCLUDE_PATH ... 50
OBJC_INT_SELECTORS ... 228
OBJC_NONUNIQUE_SELECTORS ... 228
OBJC_PROLOGUE ... 275
OBSTACK_CHUNK_ALLOC ... 300
OBSTACK_CHUNK_FREE ... 300
OBSTACK_CHUNK_SIZE ... 300
obstack_free ... 61
OCS (88k) ... 42
offsettable address ... 175
omitted middle-operands ... 81
one_cmpl*m*2 instruction pattern ... 186
ONLY_INT_FIELDS ... 299
open coding ... 93
operand access ... 128
operand constraints ... 175
operand substitution ... 172
operands ... 168
OPTIMIZATION_OPTIONS ... 223
optimize options ... 29
optional hardware or system features ... 221
options to control warnings ... 22
options, code generation ... 47
options, debugging ... 27
options, dialect ... 19
options, directory search ... 36
options, GNU CC command ... 15
options, grouping ... 15
options, linking ... 34
options, optimization ... 29
options, order ... 15
options, preprocessor ... 33
order of options ... 15
order of register allocation ... 231
ORDER_REGS_FOR_LOCAL_ALLOC ... 231
Ordering of Patterns ... 190
other directory, compilation in ... 59
OUTGOING_REG_PARM_STACK_SPACE ... 245
output file option ... 19
output of assembler code ... 274
output statements ... 173
output templates ... 172
output_addr_const ... 276
output_asm_insn ... 174
overflow while constant folding ... 292
OVERLAPPING_REGNO_P ... 234
OVERRIDE_OPTIONS ... 222

P

'p' in constraint ... 177
parallel ... 152
parameter forward declaration ... 83
parameters, miscellaneous ... 293
PARM_BOUNDARY ... 224
parser generator, Bison ... 56
parsing pass ... 121
passes and files of the compiler ... 121
passing arguments ... 119
PATTERN ... 160
pattern conditions ... 168
pattern names ... 183
Pattern Ordering ... 190
patterns ... 167
pc ... 143
pc and attributes ... 210
pc, RTL sharing ... 166
pc_rtx ... 143
PCC_BITFIELD_TYPE_MATTERS ... 225
PCC_STATIC_STRUCT_RETURN ... 252

PDImode ... 134
peephole optimization ... 125
peephole optimization, RTL representation ... 153
peephole optimizer definitions ... 196
percent sign ... 172
perform_ ... 263
PIC ... 273
PIC_OFFSET_TABLE_REGNUM ... 273
plus ... 143
plus and attributes ... 206
plus, canonicalization of ... 195
Pmode ... 296
pointer arguments ... 89
POINTER_SIZE ... 224
portability ... 117
position independent code ... 273
post_dec ... 155
post_inc ... 155
pragma ... 296
pragma, reason for not using ... 90
pre_dec ... 154
pre_inc ... 155
predefined macros ... 221
PREDICATE_CODES ... 293
PREFERRED_RELOAD_CLASS ... 238
preprocessor options ... 33
PRESERVE_DEATH_INFO_REGNO_P ... 235
prev_cc0_setter ... 194
PREV_INSN ... 156
prev_nonnote_insn ... 197
PRINT_OPERAND ... 283
PRINT_OPERAND_ADDRESS ... 283
PRINT_OPERAND_PUNCT_VALID_P ... 283
product ... 144
prof ... 28
PROFILE_BEFORE_PROLOGUE ... 256
profiling, code generation ... 255
program counter ... 143
prologue ... 253
PROMOTE_PROTOTYPES ... 245
promotion of formal parameters ... 65
pseudo registers ... 139
PSImode ... 134
PTRDIFF_TYPE ... 228
push address instruction ... 177
PUSH_ROUNDING ... 245
PUSH_ROUNDING, interaction with STACK_BOUNDARY ... 224
PUT_CODE ... 127
PUT_MODE ... 136
PUT_REG_NOTE_KIND ... 161
PUT_SDB_ ... 287

Q

'Q', in constraint ... 177
QImode ... 134
QImode, in insn ... 159
qsort, and global register variables ... 100
question mark ... 180
quotient ... 144

R

'r' in constraint ... 176
r0-relative references (88k) ... 43
ranges in case statements ... 87
read-only strings ... 69
READONLY_DATA_SECTION ... 272
REAL_ARITHMETIC ... 292
REAL_INFINITY ... 292
REAL_VALUE_ATOF ... 292
REAL_VALUE_FIX ... 291
REAL_VALUE_FIX_TRUNCATE ... 291
REAL_VALUE_FROM_INT ... 293
REAL_VALUE_ISINF ... 292
REAL_VALUE_ISNAN ... 292
REAL_VALUE_LDEXP ... 291
REAL_VALUE_NEGATE ... 292
REAL_VALUE_TO_INT ... 293
REAL_VALUE_TRUNCATE ... 293
REAL_VALUE_TYPE ... 291
REAL_VALUE_UNSIGNED_FIX ... 291
REAL_VALUE_UNSIGNED_FIX_TRUNCATE ... 291
REAL_VALUES_EQUAL ... 291
REAL_VALUES_LESS ... 291
recog_operand ... 282
recognizing insns ... 169
reg ... 139

`reg` and '`/i`' ... 131
`reg` and '`/s`' ... 131
`reg` and '`/u`' ... 131
`reg` and '`/v`' ... 131
`reg`, RTL sharing ... 166
`REG_ALLOC_ORDER` ... 231
`REG_CC_SETTER` ... 163
`REG_CC_USER` ... 163
`REG_CLASS_CONTENTS` ... 237
`REG_CLASS_FROM_LETTER` ... 237
`REG_CLASS_NAMES` ... 237
`REG_DEAD` ... 161
`REG_DEP_ANTI` ... 164
`REG_DEP_OUTPUT` ... 164
`REG_EQUAL` ... 162
`REG_EQUIV` ... 162
`REG_FUNCTION_VALUE_P` ... 131
`REG_INC` ... 161
`REG_LABEL` ... 162
`REG_LEAF_ALLOC_ORDER` ... 233
`REG_LIBCALL` ... 163
`REG_LOOP_TEST_P` ... 131
`reg_names` ... 283
`REG_NO_CONFLICT` ... 161
`REG_NONNEG` ... 161
`REG_NOTE_KIND` ... 161
`REG_NOTES` ... 160
`REG_OK_FOR_BASE_P` ... 265
`REG_OK_FOR_INDEX_P` ... 265
`REG_OK_STRICT` ... 265
`REG_PARM_STACK_SPACE` ... 245
`REG_PARM_STACK_SPACE`, and `FUNCTION_ARG` ... 247
`REG_RETVAL` ... 163
`REG_UNUSED` ... 163
`REG_USERVAR_P` ... 131
`REG_WAS_0` ... 163
register allocation ... 124
register allocation order ... 231
register allocation, stupid ... 123
register class definitions ... 235
register class preference constraints ... 181
register class preference pass ... 124
register pairs ... 232
register positions in frame (88k) ... 42
Register Transfer Language (RTL) ... 127
register usage ... 229
register use analysis ... 123
register variable after `longjmp` ... 100
register-to-stack conversion ... 125
`REGISTER_MOVE_COST` ... 270
`REGISTER_NAMES` ... 282
`register_operand` ... 170
`REGISTER_PREFIX` ... 284
registers ... 94
registers arguments ... 247
registers for local variables ... 101
registers in constraints ... 176
registers, global allocation ... 99
registers, global variables in ... 99
`REGNO_OK_FOR_BASE_P` ... 237
`REGNO_OK_FOR_INDEX_P` ... 237
`REGNO_REG_CLASS` ... 237
`regs_ever_live` ... 253
relative costs ... 269
`RELATIVE_PREFIX_NOT_LINKDIR` ... 219
reload pass ... 141
`reload_completed` ... 189
`reload_in` ... 184
`reload_in_progress` ... 183
`reload_out` ... 184
reloading ... 124
remainder ... 144
reporting bugs ... 103
representation of RTL ... 127
`rest_of_compilation` ... 121
`rest_of_decl_compilation` ... 121
`return` ... 151
`return` instruction pattern ... 189
return value of `main` ... 114
return values in registers ... 249
`RETURN_IN_MEMORY` ... 251
`RETURN_POPS_ARGS` ... 246
returning aggregate values ... 251
returning structures and unions ... 119
right rotate ... 145
right shift ... 145

`rotate` ... 145
`rotatert` ... 145
`rotl`*m*`3` instruction pattern ... 185
`rotr`*m*`3` instruction pattern ... 185
`ROUND_TYPE_ALIGN` ... 226
`ROUND_TYPE_SIZE` ... 226
RS/6000 Options ... 44
RT options ... 44
RT PC ... 71
RTL addition ... 143
RTL comparison ... 143
RTL comparison operations ... 146
RTL constant expression types ... 137
RTL constants ... 137
RTL declarations ... 150
RTL difference ... 143
RTL expression ... 127
RTL expressions for arithmetic ... 143
RTL format ... 128
RTL format characters ... 128
RTL function-call insns ... 164
RTL generation ... 122
RTL insn template ... 169
RTL integers ... 127
RTL memory expressions ... 139
RTL object types ... 127
RTL postdecrement ... 154
RTL postincrement ... 154
RTL predecrement ... 154
RTL preincrement ... 154
RTL register expressions ... 139
RTL representation ... 127
RTL side effect expressions ... 150
RTL strings ... 127
RTL structure sharing assumptions ... 165
RTL subtraction ... 143
RTL sum ... 143
RTL vectors ... 127
RTX (See RTL) ... 127
`RTX_COSTS` ... 269
`RTX_INTEGRATED_P` ... 131
`RTX_UNCHANGING_P` ... 131
run-time conventions ... 119
run-time options ... 47
run-time target specification ... 221

S

'`s`' in constraint ... 176
`saveable_obstack` ... 265
scalars, returned as values ... 249
`scanf`, and constant strings ... 69
`SCCS_DIRECTIVE` ... 296
`SCHED_GROUP_P` ... 132
scheduling, delayed branch ... 125
scheduling, instruction ... 124
`SCmode` ... 135
SCO installation ... 61
s*cond* instruction pattern ... 187
scope of a variable length array ... 82
scope of declaration ... 65
scope of external declarations ... 70
`scratch` ... 142
scratch operands ... 142
`scratch`, RTL sharing ... 166
`SDB_ALLOW_FORWARD_REFERENCES` ... 288
`SDB_ALLOW_UNKNOWN_REFERENCES` ... 288
`SDB_DEBUGGING_INFO` ... 287
`SDB_DELIM` ... 288
`SDB_GENERATE_FAKE` ... 288
search path ... 36
`SECONDARY_INPUT_RELOAD_CLASS` ... 238
`SECONDARY_OUTPUT_RELOAD_CLASS` ... 238
`SECONDARY_RELOAD_CLASS` ... 238
`SELECT_CC_MODE` ... 268
`SELECT_RTX_SECTION` ... 273
`SELECT_SECTION` ... 272
separate directory, compilation in ... 59
`sequence` ... 153
`set` ... 150
`set_attr` ... 208
`set_attr_alternative` ... 208
`SET_DEST` ... 151
`SET_SRC` ... 151
`setjmp` ... 100
`setjmp` incompatibilities ... 69
`SETUP_INCOMING_VARARGS` ... 258

SFmode ... 134
shared strings ... 69
shared VMS run time system ... 114
SHARED_SECTION_ASM_OP ... 272
sharing of RTL components ... 165
shift ... 145
SHIFT_COUNT_TRUNCATED ... 294
SHORT_TYPE_SIZE ... 227
side effect in ?: ... 81
side effects, macro argument ... 73
sign_extend ... 149
sign_extract ... 148
sign_extract, canonicalization of ... 196
signed division ... 144
signed maximum ... 145
signed minimum ... 145
SIGNED_CHAR_SPEC ... 217
SImode ... 134
simple constraints ... 175
simplifications, arithmetic ... 121
SItype ... 263
SIZE_TYPE ... 228
sizeof ... 78
SLOW_BYTE_ACCESS ... 270
SLOW_UNALIGNED_ACCESS ... 271
SLOW_ZERO_EXTEND ... 271
SMALL_REGISTER_CLASSES ... 239
smaller data references (88k) ... 43
smaller data references (MIPS) ... 47
smax ... 145
smin ... 145
SPARC options ... 40
specified registers ... 99
specifying compiler version and target machine ... 37
specifying hardware config ... 38
specifying machine version ... 37
specifying registers for local variables ... 101
speed of instructions ... 269
splitting instructions ... 202
sqrt ... 146
sqrt*m*2 instruction pattern ... 185
square root ... 146
sscanf, and constant strings ... 69
stack arguments ... 245
stack frame layout ... 241
STACK_BOUNDARY ... 224
STACK_DYNAMIC_OFFSET ... 241
STACK_DYNAMIC_OFFSET and virtual registers ... 140
STACK_GROWS_DOWNWARD ... 241
STACK_PARMS_IN_REG_PARM_AREA ... 246
STACK_POINTER_OFFSET ... 241
STACK_POINTER_OFFSET and virtual registers ... 141
STACK_POINTER_REGNUM ... 242
STACK_POINTER_REGNUM and virtual registers ... 140
stack_pointer_rtx ... 243
STACK_REGS ... 234
stage1 ... 57
standard pattern names ... 183
STANDARD_EXEC_PREFIX ... 219
STANDARD_INCLUDE_DIR ... 219
STANDARD_STARTFILE_PREFIX ... 219
STARTFILE_SPEC ... 218
STARTING_FRAME_OFFSET ... 241
STARTING_FRAME_OFFSET and virtual registers ... 140
statements inside expressions ... 73
STATIC_CHAIN ... 243
STATIC_CHAIN_INCOMING ... 243
STATIC_CHAIN_INCOMING_REGNUM ... 242
STATIC_CHAIN_REGNUM ... 242
'stdarg.h' and register arguments ... 247
'stdarg.h' and RT PC ... 44
STDC_VALUE ... 221
storage layout ... 223
STORE_FLAG_VALUE ... 295
strcpy ... 224
strength-reduction ... 123
STRICT_ALIGNMENT ... 225
strict_low_part ... 150
string constants ... 69
STRUCT_VALUE ... 251
STRUCT_VALUE_INCOMING ... 252
STRUCT_VALUE_INCOMING_REGNUM ... 251
STRUCT_VALUE_REGNUM ... 251
structure passing (88k) ... 44
structure value address ... 251
STRUCTURE_SIZE_BOUNDARY ... 225

structures ... 70
structures, constructor expression ... 85
structures, returning ... 119
stupid register allocation ... 123
`sub`*m*3 instruction pattern ... 184
submodel options ... 38
`subreg` ... 141
`subreg`, in `strict_low_part` ... 150
`subreg`, special reload handling ... 141
`SUBREG_REG` ... 141
`SUBREG_WORD` ... 141
subscripting ... 84
subscripting and function values ... 84
`SUCCESS_EXIT_CODE` ... 299
Sun installation ... 60
suppressing warnings ... 22
SVr4 ... 43
`SWITCH_TAKES_ARG` ... 217
`SWITCHES_NEED_SPACES` ... 217
`symbol_ref` ... 138
`symbol_ref` and '`/u`' ... 132
`symbol_ref` and '`/v`' ... 131
`symbol_ref`, RTL sharing ... 166
`SYMBOL_REF_FLAG` ... 131
`SYMBOL_REF_FLAG`, in `ENCODE_SECTION_INFO` ... 273
`SYMBOL_REF_USED` ... 131
symbolic label ... 166
syntax checking ... 22
`SYSTEM_INCLUDE_DIR` ... 219

T

`tablejump` instruction pattern ... 190
tagging insns ... 207
tail recursion optimization ... 122
target machine, specifying ... 37
target options ... 37
target specifications ... 221
target-parameter-dependent code ... 122
`TARGET_BELL` ... 229
`TARGET_BS` ... 229
`TARGET_CR` ... 229
`TARGET_FF` ... 229
`TARGET_FLOAT_FORMAT` ... 226
`TARGET_MEM_FUNCTIONS` ... 262
`TARGET_NEWLINE` ... 229
`TARGET_OPTIONS` ... 222
`TARGET_SWITCHES` ... 221
`TARGET_TAB` ... 229
`TARGET_VERSION` ... 222
`TARGET_VT` ... 229
`TCmode` ... 135
`tcov` ... 28
termination routines ... 281
`text_section` ... 272
`TEXT_SECTION_ASM_OP` ... 272
`TFmode` ... 134
thunks ... 76
`TImode` ... 134
'`tm.h`' macros ... 217
`TMPDIR` ... 50
top level of compiler ... 121
traditional C language ... 20
`TRADITIONAL_RETURN_FLOAT` ... 249
`TRAMPOLINE_ALIGNMENT` ... 259
`TRAMPOLINE_SIZE` ... 259
`TRAMPOLINE_TEMPLATE` ... 259
trampolines for nested functions ... 259
`TRANSFER_FROM_TRAMPOLINE` ... 261
`TRULY_NOOP_TRUNCATION` ... 294
`truncate` ... 149
`trunc`*mn* instruction pattern ... 187
`tst`*m* instruction pattern ... 186
type alignment ... 91
typedef names as function parameters ... 70
`typeof` ... 78

U

`udiv` ... 144
`UDIVDI3_LIBCALL` ... 262
`udiv`*m*3 instruction pattern ... 184
`udivmod`*m*4 instruction pattern ... 185
`UDIVSI3_LIBCALL` ... 261
Ultrix calling convention ... 71
`umax` ... 145
`umax`*m*3 instruction pattern ... 184
`umin` ... 145

`uminm3` instruction pattern 184
`umod` 144
`UMODDI3_LIBCALL` 262
`umodm3` instruction pattern 184
`UMODSI3_LIBCALL` 262
`umulhisi3` instruction pattern 185
`umulqihi3` instruction pattern 185
`umulsidi3` instruction pattern 185
`unchanging` 133
`unchanging`, in `call_insn` 132
`unchanging`, in `insn` 132
`unchanging`, in `reg` and `mem` 131
`unchanging`, in `symbol_ref` 132
undefined behavior 103
undefined function value 103
underscores in variables in macros 78
underscores, avoiding (88k) 42
union, casting to a 88
unions 70
unions, returning 119
`UNITS_PER_WORD` 224
`UNKNOWN_FLOAT_FORMAT` 226
Unos installation 61
unreachable code 122
`unshare_all_rtl` 166
unsigned division 144
unsigned greater than 147
unsigned less than 147
unsigned minimum and maximum 145
`unsigned_fix` 149
`unsigned_float` 149
`unspec` 154
`unspec_volatile` 154
`use` 152
`USE_C_ALLOCA` 300
`used` 132
`used`, in `symbol_ref` 131
`USER_LABEL_PREFIX` 284
`USG` 299

V

'`V`' in constraint 176
value after `longjmp` 100
values, returned by functions 249
varargs implementation 257
'`varargs.h`' and RT PC 44
variable alignment 91
variable attributes 91
variable-length array scope 82
variable-length arrays 82
variables in specified registers 99
variables, local, in macros 78
Vax calling convention 71
VAX options 40
`VAX_FLOAT_FORMAT` 226
'`VAXCRTL`' 114
`VIRTUAL_INCOMING_ARGS_REGNUM` 140
`VIRTUAL_OUTGOING_ARGS_REGNUM` 140
`VIRTUAL_STACK_DYNAMIC_REGNUM` 140
`VIRTUAL_STACK_VARS_REGNUM` 140
`VMS` 299
VMS and case sensitivity 115
VMS and include files 111
VMS installation 62
void pointers, arithmetic 84
void, size of pointer to 84
`VOIDmode` 135
`volatil` 133
`volatil`, in `insn` 131
`volatil`, in `mem` 130
`volatil`, in `reg` 131
`volatil`, in `symbol_ref` 131
`volatile` applied to function 89
volatile memory references 133
voting between constraint alternatives 181
`vprintf` 296

W

warning messages 22
`WCHAR_TYPE` 228
`WCHAR_TYPE_SIZE` 228
`which_alternative` 174
whitespace 70
`word_mode` 137
`WORD_SWITCH_TAKES_ARG` 217
`WORDS_BIG_ENDIAN` 223

`WORDS_BIG_ENDIAN`, effect on `subreg` 141

X

'`X`' in constraint 177
`XCmode` 135
`XEXP` 129
`XFmode` 134
`XINT` 129
'`xm-`*machine*`.h`' 299
`xor` 145
`xor`, canonicalization of 196
`xor`*m*`3` instruction pattern 184
`XSTR` 129
`XVEC` 130
`XVECEXP` 130
`XVECLEN` 130

Z

zero division on 88k 43
zero-length arrays 82
`zero_extend` 149
`zero_extend`*mn* instruction pattern 187
`zero_extract` 148
`zero_extract`, canonicalization of 196

Table of Contents

GNU GENERAL PUBLIC LICENSE **1**

Preamble 1
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION 2
Appendix: How to Apply These Terms to Your New Programs 7

Contributors to GNU CC **9**

1 Protect Your Freedom—Fight "Look And Feel" . . **11**

2 GNU CC Command Options **15**

2.1 Options Controlling the Kind of Output 18
2.2 Options Controlling Dialect 19
2.3 Options to Request or Suppress Warnings 22
2.4 Options for Debugging Your Program or GNU CC 27
2.5 Options That Control Optimization 29
2.6 Options Controlling the Preprocessor 33
2.7 Options for Linking 34
2.8 Options for Directory Search 36
2.9 Specifying Target Machine and Compiler Version 37
2.10 Specifying Hardware Models and Configurations 38
2.10.1 M680x0 Options 39
2.10.2 VAX Options 40
2.10.3 SPARC Options 40
2.10.4 Convex Options 40
2.10.5 AMD29K Options 41
2.10.6 M88K Options 42
2.10.7 IBM RS/6000 Options 44
2.10.8 IBM RT Options 44
2.10.9 MIPS Options 45
2.11 Options for Code Generation Conventions 47
2.12 Environment Variables Affecting GNU CC 49

3 Installing GNU CC **53**

3.1 Compilation in a Separate Directory 59
3.2 Installing GNU CC on the Sun 60
3.3 Installing GNU CC on the 3b1 60

3.4 Installing GNU CC on SCO System V 3.2 61
3.5 Installing GNU CC on Unos 61
3.6 Installing GNU CC on VMS 62

4 Known Causes of Trouble with GNU CC 65

5 How To Get Help with GNU CC 67

6 Incompatibilities of GNU CC 69

7 GNU Extensions to the C Language 73

7.1 Statements and Declarations within Expressions 73
7.2 Locally Declared Labels 74
7.3 Labels as Values 75
7.4 Nested Functions 76
7.5 Naming an Expression's Type 78
7.6 Referring to a Type with `typeof` 78
7.7 Generalized Lvalues 79
7.8 Conditional Expressions with Omitted Operands 81
7.9 Double-Word Integers 81
7.10 Arrays of Length Zero 82
7.11 Arrays of Variable Length 82
7.12 Non-Lvalue Arrays May Have Subscripts 84
7.13 Arithmetic on `void`- and Function-Pointers 84
7.14 Non-Constant Initializers 84
7.15 Constructor Expressions 85
7.16 Labeled Elements in Initializers 86
7.17 Case Ranges 87
7.18 Cast to a Union Type 88
7.19 Declaring Attributes of Functions 89
7.20 Dollar Signs in Identifier Names 90
7.21 The Character `ESC` in Constants 91
7.22 Inquiring on Alignment of Types or Variables 91
7.23 Specifying Attributes of Variables 91
7.24 An Inline Function is As Fast As a Macro 93
7.25 Assembler Instructions with C Expression Operands 94
7.26 Controlling Names Used in Assembler Code 98
7.27 Variables in Specified Registers 99
7.27.1 Defining Global Register Variables 99
7.27.2 Specifying Registers for Local Variables 101
7.28 Alternate Keywords 102
7.29 Incomplete `enum` Types 102

8 Reporting Bugs 103
8.1 Have You Found a Bug? 103
8.2 How to Report Bugs 104
8.3 Certain Changes We Don't Want to Make 108

9 Using GNU CC on VMS 111
9.1 Include Files and VMS 111
9.2 Global Declarations and VMS 112
9.3 Other VMS Issues 114

10 GNU CC and Portability 117

11 Interfacing to GNU CC Output 119

12 Passes and Files of the Compiler 121

13 RTL Representation 127
13.1 RTL Object Types 127
13.2 Access to Operands 128
13.3 Flags in an RTL Expression 130
13.4 Machine Modes 134
13.5 Constant Expression Types 137
13.6 Registers and Memory 139
13.7 RTL Expressions for Arithmetic 143
13.8 Comparison Operations 146
13.9 Bit Fields 148
13.10 Conversions 148
13.11 Declarations 150
13.12 Side Effect Expressions 150
13.13 Embedded Side-Effects on Addresses 154
13.14 Assembler Instructions as Expressions 155
13.15 Insns 156
13.16 RTL Representation of Function-Call Insns 164
13.17 Structure Sharing Assumptions 165

14 Machine Descriptions 167
14.1 Everything about Instruction Patterns 167
14.2 Example of `define_insn` 168
14.3 RTL Template for Generating and Recognizing Insns 169
14.4 Output Templates and Operand Substitution 172
14.5 C Statements for Generating Assembler Output 173

14.6 Operand Constraints 175
14.6.1 Simple Constraints 175
14.6.2 Multiple Alternative Constraints 180
14.6.3 Register Class Preferences 181
14.6.4 Constraint Modifier Characters 181
14.6.5 Not Using Constraints 182
14.7 Standard Names for Patterns Used in Generation 183
14.8 When the Order of Patterns Matters 190
14.9 Interdependence of Patterns 191
14.10 Defining Jump Instruction Patterns 192
14.11 Canonicalization of Instructions 195
14.12 Defining Machine-Specific Peephole Optimizers 196
14.13 Defining RTL Sequences for Code Generation 199
14.14 Splitting Instructions into Multiple Instructions 202
14.15 Instruction Attributes 204
14.15.1 Defining Attributes and their Values 204
14.15.2 Attribute Expressions 205
14.15.3 Assigning Attribute Values to Insns 207
14.15.4 Example of Attribute Specifications 209
14.15.5 Computing the Length of an Insn 210
14.15.6 Delay Slot Scheduling 212
14.15.7 Specifying Function Units 213

15 Machine Description Macros 217

15.1 Controlling the Compilation Driver, 'gcc' 217
15.2 Run-time Target Specification 221
15.3 Storage Layout 223
15.4 Layout of Source Language Data Types 227
15.5 Register Usage 229
15.5.1 Basic Characteristics of Registers 229
15.5.2 Order of Allocation of Registers 231
15.5.3 How Values Fit in Registers 231
15.5.4 Handling Leaf Functions 233
15.5.5 Registers That Form a Stack 234
15.5.6 Obsolete Macros for Controlling Register Usage 234
15.6 Register Classes 235
15.7 Describing Stack Layout and Calling Conventions 241
15.7.1 Basic Stack Layout 241
15.7.2 Registers That Address the Stack Frame 242
15.7.3 Eliminating Frame Pointer and Arg Pointer 243
15.7.4 Passing Function Arguments on the Stack 245
15.7.5 Passing Arguments in Registers 247

15.7.6 How Scalar Function Values Are Returned 249
15.7.7 How Large Values Are Returnd 251
15.7.8 Caller-Saves Register Allocation 252
15.7.9 Function Entry and Exit 253
15.7.10 Generating Code for Profiling 255
15.8 Implementing the Varargs Macros 257
15.9 Trampolines for Nested Functions 259
15.10 Implicit Calls to Library Routines 261
15.11 Addressing Modes 264
15.12 Condition Code Status 267
15.13 Describing Relative Costs of Operations 269
15.14 Dividing the Output into Sections (Texts, Data, ...) 271
15.15 Position Independent Code 273
15.16 Defining the Output Assembler Language 274
15.16.1 The Overall Framework of an Assembler File 274
15.16.2 Output of Data 276
15.16.3 Output of Uninitialized Variables 277
15.16.4 Output and Generation of Labels 278
15.16.5 Output of Initialization Routines 281
15.16.6 Output of Assembler Instructions 282
15.16.7 Output of Dispatch Tables 284
15.16.8 Assembler Commands for Alignment 285
15.17 Controlling Debugging Information Format 286
15.18 Cross Compilation and Floating Point Format 290
15.19 Miscellaneous Parameters 293

16 The Configuration File 299

Index 301