Using and Porting GNU CC

Richard M. Stallman

last updated 21 February 1990

for version 1.37.1

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our 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. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, 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 a 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 tell them 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.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement 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 work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
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 General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual 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 General Public License.
   - 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.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.
4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying 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.
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.
8. 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 the 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 the license, you may choose any version ever published by the Free Software Foundation.
9. 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

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
11. 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 humanity, 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) 19yy  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 1, 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) 19yy 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 a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program ‘Gnomovision’ (a program to direct compilers to make passes
at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

That's all there is to it!

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 MCC wrote most of the description of the National Semiconductor 32000 series cpu. He also wrote the code for inline function integration and for the SPARC cpu and Motorola 88000 cpu and part of the Sun FPA support.
- 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.
- 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.

1 Protect Your Freedom—Fight "Look And Feel"

Ashton-Tate, 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 three 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, Apple and Ashton-Tate are permitted to make law through the courts, the precedent will hobble the software industry:

- Gratuitous incompatibilites 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.

Here are some things you can do to protect your freedom to write programs:

- Don't buy from Xerox, Lotus, Apple or Ashton-Tate. 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.
- Join the League for Programming Freedom. Phone (617) 492-0023 or write to:

```
League for Programming Freedom
1 Kendall Square #143
P.O. Box 9171
Cambridge, MA 02139       league@prep.ai.mit.edu
```

- Above all, don't work for the look-and-feel plaintiffs, and don't accept contracts from them.
- Write to or phone your elected representatives to show them how important this issue is.

```
Senator So and So       Representative So and So
United States Senate    House of Representatives
Washington, DC 20510    Washington, DC 20515
```

  You can phone senators and representatives at (202) 225-3121.

Express your opinion! You can make a difference.

2 GNU CC Command Options

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`'.

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. File names which end in '`.c`' are taken as C source to be preprocessed and compiled; file names ending in '`.i`' are taken as preprocessor output to be compiled; compiler output files plus any input files with names ending in '`.s`' are assembled; then the resulting object files, plus any other input files, are linked together to produce an executable.

Command 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 command 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; these are not documented here, but you rarely need to use any of them.

Here are the options to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

'`-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.

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

'`-c`' Compile or assemble the source files, but do not link. Produce object files with names made by replacing '`.c`' or '`.s`' with '`.o`' at the end of the input file names. Do nothing at all for object files specified as input.

'`-S`' Compile into assembler code but do not assemble. The assembler output file name is made by replacing '`.c`' with '`.s`' at the end of the input file name. Do nothing at all for assembler source files or object files specified as input.

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

'`-v`' Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler and linker. Some of these are directed to print their own version numbers.

'`-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.

'`-B`*`prefix`*'

Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are `cpp`, `cc1`, `as` and `ld`.

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.

The run-time support file `gnulib` is also 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. Most of the time, on most machines, you can do without it.

You can get a similar result from the environment variable; `GCC_EXEC_PREFIX` if it is defined, its value is used as a prefix in the same way. If both the '`-B`' option and the `GCC_EXEC_PREFIX` variable are present, the '`-B`' option is used first and the environment variable value second.

'`-b`*prefix*'
: The argument *prefix* is used as a second prefix for the compiler executables and libraries. This prefix is optional: the compiler tries each file first with it, then without it. This prefix follows the prefix specified with '`-B`' or the default prefixes.

  Thus, '`-bvax- -Bcc/`' in the presence of environment variable `GCC_EXEC_PREFIX` with definition `/u/foo/` causes GNU CC to try the following file names for the preprocessor executable:

```
cc/vax-cpp
cc/cpp
/u/foo/vax-cpp
/u/foo/cpp
/usr/local/lib/gcc-vax-cpp
/usr/local/lib/gcc-cpp
/usr/lib/gcc-vax-cpp
/usr/lib/gcc-cpp
```

These options control the details of C compilation itself.

'`-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.

  The alternate keywords `__asm__`, `__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`'.

  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.

'`-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.
- 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.
- 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.

'`-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.

Some of the '`-f`' options described below turn specific kinds of optimization on or off.

'`-g`' Produce debugging information in the operating system's native format (for DBX or SDB). GDB also can work with this debugging information.

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.

'`-gg`' Produce debugging information in the old GDB format. This is obsolete.

'`-w`' Inhibit all warning messages.

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

- 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 6.12 [Function Attributes], page 46.

- 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;
  }
  ```

  Spurious warnings can occur because GNU CC does not realize that certain functions (including `abort` and `longjmp`) will never return.

- An expression-statement contains no side effects.

In the future, other useful warnings may also be enabled by this option.

'`-Wimplicit`'
: Warn whenever a function 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).

'`-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 other '`-W...`' options below are not implied by '`-Wall`' because certain kinds of useful macros are almost impossible to write without causing those warnings.

'`-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 *`.

'`-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.

'`-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.

'`-l`*library*'
: Search a standard list of directories for a library named *library*, which is actually a file named `lib`*library*`.a`. The linker 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`' searches several directories.

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

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

'`-m`*machinespec*'
Machine-dependent option specifying something about the type of target machine. 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.

These are the '`-m`' options defined in the 68000 machine description:

'`-m68020`'
'`-mc68020`'
Generate output for a 68020 (rather than a 68000). This is the default if you use the unmodified sources.

'`-m68000`'

'`-mc68000`'
Generate output for a 68000 (rather than a 68020).

'`-m68881`' Generate output containing 68881 instructions for floating point. This is the default if you use the unmodified sources.

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

'`-msoft-float`'
Generate output containing library calls for floating point.

'`-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.

'`-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.

These '`-m`' options are defined in the Vax machine description:

'`-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.

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

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

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

These '`-m`' options are defined in the Convex machine description:

'`-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; this info is in the symbol table.)

'`-mnoargcount`'
Omit the argument count word. This is the default if you use the unmodified sources.

'`-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.

'`-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 GCC-compiled code and PCC-compiled code.

'`-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-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.

'`-fno-defer-pop`'
: Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

'`-fstrength-reduce`'
: Perform the optimizations of loop strength reduction and elimination of iteration variables.

'`-fcombine-regs`'
: Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to '`-O`'. I am interested in hearing about the difference this makes.

'`-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, 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 13.3 [Registers], page 122.

'`-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.

'`-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.

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

‘`-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.

‘`-fvolatile`’
: Consider all memory references through pointers to be volatile.

‘`-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.

‘`-funsigned-char`’
: Let the type `char` be the 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. (Actually, at present, the default is always signed.)

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

  Note that this is equivalent to ‘`-fno-signed-char`’, which is the negative form of ‘`-fsigned-char`’.

‘`-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`’.

‘`-fdelayed-branch`’
: If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

‘`-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.

'`-d`*letters*'
: Says to make debugging dumps at times specified by *letters*. Here are the possible letters:

  '`r`' Dump after RTL generation.

  '`j`' Dump after first jump optimization.

  '`s`' Dump after CSE (including the jump optimization that sometimes follows CSE).

  '`L`' Dump after loop optimization.

  '`f`' Dump after flow analysis.

  '`c`' Dump after instruction combination.

  '`l`' Dump after local register allocation.

  '`g`' Dump after global register allocation.

  '`d`' Dump after delayed branch scheduling.

  '`J`' Dump after last jump optimization.

  '`m`' Print statistics on memory usage, at the end of the run.

'`-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. There is no reason to *use* this option; it exists only to satisfy pedants.

'`-pedantic`' does not cause warning messages for use of the alternate keywords whose names begin and end with '`__`'. See Section 6.19 [Alternate Keywords], page 53.

'`-static`' On Suns running version 4, this prevents linking with the shared libraries. ('`-g`' has the same effect.)

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 C preprocessing. Some of these options make sense only together with '`-E`' because they request preprocessor output that is not suitable for actual compilation.

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

'`-I`*dir*' Search directory *dir* for include files.

'`-I-`' Any directories specified 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.

'`-i` *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 '`-i` *file*' is to make the macros defined in *file* available for use in the main input.

'`-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.

Between '`-nostdinc`' and '`-I-`', you can eliminate all directories from the search path except those you specify.

'`-M`' Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each source 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.

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

'`-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.

'`-MM`' implies '`-E`'.

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

'`-Dmacro=defn`'
Define macro *macro* as *defn*.

'`-Umacro`' Undefine macro *macro*.

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

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 Sun systems, 3b1 machines and HPUX. 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. Edit `Makefile`. If you are using HPUX, or any form of system V, you must make a few changes described in comments at the beginning of the file. Genix requires changes also, and so does the Pyramid.
2. On a Sequent system, go to the Berkeley universe.
3. Choose configuration files. The easy way to do this is to run the command file `config.gcc` with a single argument, which specifies the type of machine (and in some cases which operating system).

   Here is a list of the possible arguments:

   'vax' Vaxes running BSD.

   'vms' Vaxes running VMS.

   'vax-sysv'
   Vaxes running system V.

   'i386-sysv'
   Intel 386 PCs running system V.

   'i386-sysv-gas'
   Intel 386 PCs running system V, using the GNU assembler and GNU linker.

   'sequent-i386'
   Sequent with Intel 386 processors.

   'i386-aix'
   Intel 386 PCs or PS/2s running AIX.

   'sun2' Sun 2 running system version 2 or 3.

   'sun3' Sun 3 running system version 2 or 3, with 68881. Note there we do not provide a configuration file to use an FPA by default, because programs that establish signal handlers for floating point traps inherently cannot work with the FPA.

   'sun3-nfp'
   Sun 3 running system version 2 or 3, without 68881.

   'sun4' Sun 4 running system version 2 or 3. See Chapter 5 [Incompatibilities], page 37, for calling convention incompatibilities on the Sun 4 (sparc).

   'sun2-os4'
   Sun 2 running system version 4.

   'sun3-os4'
   Sun 3 running system version 4, with 68881.

‘`sun3-nfp-os4`’
Sun 3 running system version 4, without 68881.

‘`sun4-os4`’
Sun 4 running system version 4. See Chapter 5 [Incompatibilities], page 37, for calling convention incompatibilities on the Sun 4 (sparc).

‘`sun386`’ Sun 386 (“roadrunner”).

‘`alliant`’ Alliant FX/8 computer. Note that the standard installed C compiler in Concentrix 5.0 has a bug which prevent it from compiling GNU CC correctly. You can patch the compiler bug as follows:

```
cp /bin/pcc ./pcc
adb -w ./pcc - << EOF
15f6?w 6610
EOF
```

Then you must use the ‘`-ip12`’ option when compiling GNU CC with the patched compiler, as shown here:

```
make CC="./pcc -ip12" CFLAGS=-w
```

Note also that Alliant’s version of DBX does not manage to work with the output from GNU CC.

‘`tahoe`’ The tahoe computer (running BSD, and using DBX).

‘`decstation`’
The DEC 3100 Mips machine (“pmax”). Note that GNU CC cannot generate debugging information in the unusual format used on the Mips.

‘`mips-sysv`’
The Mips computer, RS series, with the System V environment as default. Note that GNU CC cannot generate debugging information in the unusual format used on the Mips.

‘`mips-bsd43`’
The Mips computer, RS series, with the BSD 4.3 environment as default. Note that GNU CC cannot generate debugging information in the unusual format used on the Mips.

‘`mips`’ The Mips computer, M series. Note that GNU CC cannot generate debugging information in the unusual format used on the Mips.

‘`iris`’ The Mips computer, as delivered by Iris. Note that GNU CC cannot generate debugging information in the unusual format used on the Mips.

‘`convex-c1`’
Convex C1 computer.

‘`convex-c2`’
Convex C2 computer.

‘`pyramid`’ Pyramid computer.

‘`hp9k320`’ HP 9000 series 300 using HPUX assembler. Note there is no support in GNU CC for HP’s debugger; thus, ‘`-g`’ is not available in this configuration.

'`hp9k320-gas`'
: HP 9000 series 300 using GNU assembler, linker and debugger. This requires the HP-adapt package, which is available along with the GNU linker as part of the "binutils" distribution. This is on the GNU CC distribution tape.

'`hp9k320-old`'
: HP 9000 series 300 using HPUX assembler, in operating system versions older than 6.5. Note there is no support in GNU CC for HP's debugger; thus, '`-g`' is not available in this configuration.

'`hp9k320-bsd`'
: HP 9000 series 300 running BSD.

'`isi68`' ISI 68000 or 68020 system with a 68881.

'`isi68-nfp`'
: ISI 68000 or 68020 system without a 68881.

'`news800`' Sony NEWS 68020 system.

'`next`' NeXT system.

'`altos`' Altos 3068. Note that 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`.

'`3b1`' AT&T 3b1, a.k.a. 7300 PC. Note that 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 32. You can bootstrap it more easily with previous versions of GNU CC if you have them.

'`3b1-gas`' AT&T 3b1 using the GNU assembler.

'`sequent-ns32k`'
: Sequent containing ns32000 processors.

'`encore`' Encore ns32000 system.

'`genix`' National Semiconductor ns32000 system.

'`88000`' Motorola 88000 processor. This port is not finished.

Here we spell out what files need to be set up:

- Make a symbolic link named `config.h` to the top-level config file for the machine you are using (see Chapter 14 [Config], page 157). This file is responsible for defining information about the host machine. It includes `tm.h`.

  The file is located in the subdirectory `config`. Its name should be `xm-`*`machine`*`.h`, with these exceptions:

  `xm-vms.h` for vaxen running VMS.

  `xm-vaxv.h`
  : for vaxen running system V.

  `xm-i386v.h`
  : for Intel 80386's running system V.

`xm-sun386i.h`
: for Sun roadrunner running any version of the operating system.

`xm-hp9k320.h`
: for the HP 9000 series 300.

`xm-genix.h`
: for the ns32000 running Genix

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.

- Make a symbolic link named `tm.h` to the machine-description macro file for your machine. It should be in the subdirectory `config` and its name should be `tm-`*`machine`*`.h`.

  If your system is a 68000, don't use the file `tm-m68k.h` directly. Instead, use one of these files:

  `tm-sun3.h`
  : for Sun 3 machines with 68881.

  `tm-sun3-nfp.h`
  : for Sun 3 machines with no hardware floating point.

  `tm-sun3os3.h`
  : for Sun 3 machines with 68881, running Sunos version 3.

  `tm-sun3os3nf.h`
  : for Sun 3 machines with no hardware floating point, running Sunos version 3.

  `tm-sun2.h`
  : for Sun 2 machines.

  `tm-3b1.h` for AT&T 3b1 (aka 7300 Unix PC).

  `tm-isi68.h`
  : for Integrated Solutions systems. This file assumes you use the GNU assembler.

  `tm-isi68-nfp.h`
  : for Integrated Solutions systems without a 68881. This file assumes you use the GNU assembler.

  `tm-news800.h`
  : for Sony NEWS systems.

  `tm-hp9k320.h`
  : for HPUX systems, if you are using GNU CC with the system's assembler and linker.

  `tm-hp9k320g.h`
  : for HPUX systems, if you are using the GNU assembler, linker and other utilities. Not all of the pieces of GNU software needed for this mode of operation are as yet in distribution; full instructions will appear here in the future.

For the vax, use `tm-vax.h` on BSD Unix, `tm-vaxv.h` on system V, or `tm-vms.h` on VMS.

For the Motorola 88000, use `tm-m88k.h`. The support for the 88000 does not currently work; it requires extensive changes which we hope to reconcile in version 2.

For the 80386, don't use `tm-i386.h` directly. Use `tm-i386v.h` if the target machine is running system V, `tm-i386gas.h` if it is running system V but you are using the GNU assembler and linker, `tm-seq386.h` for a Sequent 386 system, or `tm-compaq.h` for a Compaq, or `tm-sun386i.h` for a Sun 386 system.

For the Mips computer, there are five choices: `tm-mips.h` for the M series, `tm-mips-bsd.h` for the RS series with BSD, `tm-mips-sysv.h` for the RS series with System V, `tm-iris.h` for the Iris version of the machine, and `tm-decstatn.h` for the Decstation.

For the 32000, use `tm-sequent.h` if you are using a Sequent machine, or `tm-encore.h` for an Encore machine, or `tm-genix.h` if you are using Genix version 3; otherwise, perhaps `tm-ns32k.h` will work for you.

Note that Genix has bugs in `alloca` and `malloc`; you must get the compiled versions of these from GNU Emacs and edit GNU CC's `Makefile` to use them.

Note that Encore systems are supported only under BSD.

For Sparc (Sun 4) machines, use `tm-sparc.h` with operating system version 4, and `tm-sun4os3.h` with system version 3.

- Make a symbolic link named `md` 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.
- Make a symbolic link named `aux-output.c` to the output subroutine file for your machine. It should be in the `config` subdirectory and its name should be `out-`*`machine`*`.c`.

4. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output files `c-parse.tab.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.tab.c`.

5. Build the compiler. Just type '`make`' in the compiler directory.

   Ignore any warnings you may see about "statement not reached" in the `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 7 [Bugs], page 55).

   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.

6. If you are using COFF-encapsulation, you must convert `gnulib` to a GNU-format library at this point. See the file `README-ENCAP` in the directory containing the GNU binary file utilities, for directions.

7. 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`.

8. Recompile the compiler with itself, with this command:

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

   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"
   ```

9. If you wish to test the compiler by compiling it with itself one more time, do this (in C shell):

   ```
   make stage2
   make CC=stage2/gcc CFLAGS="-g -O -Bstage2/"
   foreach file (*.o)
   cmp $file stage2/$file
   end
   ```

   Aside from the '`-B`' option, the options should be the same as when you made stage 2.

   The `foreach` command (written in C shell) will notify you if any of these stage 3 object files differs from those of stage 2. On BSD systems, 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 7 [Bugs], page 55).

   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
   echo $file
   tail +10 $file > foo1
   tail +10 stage2/$file > foo2
   cmp foo1 foo2
   done
   ```

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

    ```
    make install
    ```

    This copies the files `cc1`, `cpp` and `gnulib` to files `gcc-cc1`, `gcc-cpp` and `gcc-gnulib` in directory `/usr/local/lib`, which is where the compiler driver program looks for them. 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.

    Warning: the GNU CPP may not work for `ioctl.h`, `ttychars.h` and other system header files unless the '`-traditional`' option is used. The bug is in the header files: at

least on some machines, they rely on behavior that is incompatible with ANSI C. This behavior consists of substituting for macro argument names when they appear inside of character constants. The '`-traditional`' option tells GNU CC to behave the way these headers expect.

Because of this problem, you might prefer to configure GNU CC to use the system's own C preprocessor. To do so, make the file `/usr/local/lib/gcc-cpp` a link to `/lib/cpp`.

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.

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 `gnulib`. 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. Go to that directory before running `config.gcc`:

   ```
   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.

2. Specify where to find `config.gcc` when you run it:

   ```
   ../gcc-1.36/config.gcc ...
   ```

3. Specify where to find the sources, as an argument to `config.gcc`:

   ```
   ../gcc-1.36/config.gcc -srcdir=../gcc-1.36 sun3
   ```

   The '`-srcdir=`*dir*' option is not needed when the source directory is the parent of the current directory, because `config.gcc` detects that case automatically.

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 `config.gcc` 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 `gnulib`. If this option were set to `f68881` when `gnulib` 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 `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.

If you have installed an earlier version of GCC, you can compile the newer version with that. However, you will run into trouble compiling `gnulib`, since that is normally compiled with CC. To solve the problem, uncomment this line in `Makefile`:

```
CCLIBFLAGS = -B/usr/local/lib/gcc- -tp -Wp,-traditional
```

3.4 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.

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 '`/verbose`' 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 `tm-vms.h` to `tm.h`, `xm-vms.h` to `config.h`, `vax.md` to `md.` and `out-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.
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`' to do recompile everything.

   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.

There is a known problem on VMS: `const` global variables don't work compatibly with the VMS C compiler; we don't know a way to get them to the linker properly.

Note that GNU CC on VMS does not generate debugging information to describe the program's symbols. It is not straightforward to implement this, and we have no time to spend on it, but we might consent to install a very modular implementation if you write it. You will probably have to modify GAS as well as GNU CC.

3.5 Installing GNU CC on HPUX

To install GNU CC on HPUX, you must start by editing the file `Makefile`. Search for the string '`HPUX`' to find comments saying what to change. You need to change some variable definitions and (if you are using GAS) some lines in the rule for the target '`gnulib`'.

To compile with the HPUX C compiler, you must specify get the file `alloca.c` from GNU Emacs. Then, when you run `make`, use this argument:

```
make ALLOCA=alloca.o
```

When recompiling GNU CC with itself, do not define `ALLOCA`. Instead, an '`-I`' option needs to be added to `CFLAGS` as follows:

```
make CC=stage1/gcc CFLAGS="-g -O -Bstage1/ -I../binutils/hp-include"
```

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 13.10 [Cross-compilation], page 142.
- 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.
- 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. I don't think it's worth being incompatible for.

5 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 older 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. 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 be 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 in a special, fixed register.

  PCC usually handles all sizes of structures and unions by returning the address of a block of static storage containing the value. This method is not used in GNU CC because it is slower and nonreentrant.

You can tell GNU CC to use the PCC convention with the option '`-fpcc-struct-return`'.

- On the Sparc, GNU CC uses an incompatible calling convention for structures. It passes them by including their contents in the argument list, whereas the standard compiler passes them effectively by reference.

  This really ought to be fixed, but such calling conventions are not yet supported in GNU CC, so it isn't straightforward to fix it.

  The convention for structure returning is also incompatible, and '`-fpcc-struct-return`' does not help.

6 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.

6.1 Statements and Declarations inside of 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 6.3 [Typeof], page 42) or type naming (see Section 6.2 [Naming Types], page 41).

6.2 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.

6.3 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 6.19 [Alternate Keywords], page 53.

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`.

6.4 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 valid. Taking the address of the cast is the same as taking the address without a cast, except for the type of the result. For example, these two expressions are equivalent (but the second may be valid when the type of `a` does not permit a cast to `int *`).

```
&(int *)a
(int **)&a
```

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 converter 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 *)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)a + 5))
```

6.5 Conditional Expressions with Omitted Middle-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.

6.6 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`.

6.7 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 that time 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);
}
```

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

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

The length of an array is computed on entry to the brace-level where the array is declared and is remembered for the scope of the array in case you access it with `sizeof`.

Jumping or breaking out of the scope of the array name will also deallocate the storage. Jumping into the scope is not allowed; you will 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`.)

6.8 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];
}
```

6.9 Arithmetic on `void`-Pointers 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.

6.10 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 };
  ...
}
```

6.11 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. The type must be a structure, union or array type.

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.

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

6.12 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 functions, such as `abort` and `exit`, cannot return. These functions should be declared `volatile`. For example,

```
extern volatile void abort ();
```

tells the compiler that it can assume that `abort` will not return. This makes slightly better code, but more importantly it helps avoid spurious warnings of uninitialized variables.

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 const void 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 must not be `const`.

Some people object to this feature, claiming that ANSI C's `#pragma` should be used instead. There are two reasons I did not do 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 *any* application whatever: as far as I can see, `#pragma` is never useful.

6.13 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 if you specify '`-traditional`'; they are not allowed if you specify '`-ansi`'. Whether they are allowed by default depends on the target machine; usually, they are not.

6.14 Inquiring about the Alignment of a Type or Variable

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.

6.15 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 6.19 [Alternate Keywords], page 53.)

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.

6.16 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 12.6 [Constraints], page 98).

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 the maximum number of operands in any instruction pattern in the machine description.

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.

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 12.6.4 [Modifiers], page 103.

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");
```

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");
```

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 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))
```

(However, an instruction without output operands will not be deleted or moved, regardless, unless it is unreachable.)

It is a natural idea to look for a way to give access to the condition code left by the assembler 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 6.19 [Alternate Keywords], page 53.

6.17 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.

6.18 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 6.16 [Extended Asm], page 48).

6.18.1 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 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.

6.18.2 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 6.16 [Extended Asm], page 48). 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 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.

6.19 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
```

7 Reporting Bugs

Your bug reports play an essential role in making GNU CC reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the important 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.

7.1 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 5 [Incompatibilities], page 37). 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.

7.2 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 '`info-gcc`', or to the newsgroup '`gnu.gcc`'. 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
545 Tech Sq
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 names of the files that you used for `tm.h` and `md` when you installed the compiler.

- The type of machine you are using, and the operating system name and version number.

- 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 test 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 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.

9 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 13 [Machine Macros], page 119).

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.

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 `gnulib`, 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 `gnulib` is not needed, but it is searched just in case.

Each arithmetic function is defined in `gnulib.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, `gnulib.c` does not work if compiled with GNU CC, because each arithmetic function would compile into a call to itself!

10 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, remains 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 6.15 [Inline], page 47).

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.

  C 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 associative-law simplifications are also done during this pass.

  The source files for parsing are `c-parse.y`, `c-decl.c`, `c-typeck.c`, `c-convert.c`, `stor-layout.c`, `fold-const.c`, and `tree.c`. The last three files are intended to be language-independent. There are also header files `c-parse.h`, `c-tree.h`, `tree.h` and `tree.def`. The last two define the format of the tree representation.

- 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 are `stmt.c`, `expr.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 files `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 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.)

  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`.

- 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 as well. Its source file is `loop.c`.

  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.

- 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.
- 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.

  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.
- 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 may be done at this point. The source file name is `dbranch.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.

- 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 and `symout.c` for GDB's own symbol table format.

Some additional files are used by all or many passes:

- Every pass uses `machmode.def`, which defines the machine modes.
- 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.

11 RTL Representation

Most of the work of the compiler is done on an intermediate representation called register transfer language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

11.1 RTL Object Types

RTL uses 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`, and a string is a `char *`. Within RTL code, strings appear only inside `symbol_ref` expressions, but they appear in other contexts in the RTL expressions that make up machine descriptions. 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.

A vector contains an arbitrary, specified 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)`.

11.2 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`'.

Two other format characters are used occasionally: '`u`' and '`0`'. '`u`' is equivalent to '`e`' except that it is printed differently in debugging dumps, and '`0`' means a slot whose contents do not fit any normal category. '`0`' slots are not printed at all in dumps, and are often used in special ways by small parts of the compiler.

There are macros to get the number of operands and the format of an expression code:

`GET_RTX_LENGTH (`*code*`)`
: Number of operands of an RTX of code *code*.

`GET_RTX_FORMAT (`*code*`)`
: The format of an RTX of code *code*, as a C string.

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.

11.3 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_USER_VAR_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 explicitly by the current function. (If it is a memory reference then it may 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.

`INSN_DELETED_P (insn)`
: In an insn, nonzero if the insn has been deleted. Stored in the `volatil` field and printed as '`/v`'.

`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`'.

These are the fields which the above macros refer to:

`used`
: 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 11.17 [Sharing], page 91).

`volatil`
: This flag is used in `mem` 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 `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`
: This flag is used 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 RTL dump, this flag is represented as '`/s`'.

`unchanging`
: This flag is used in `reg` and `mem` expressions. 1 means that the value of the expression never changes (at least within the current 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.

11.4 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.

`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.

`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.

`EPmode` "Entry Pointer" mode is intended to be used for function variables in Pascal and other block structured languages. Such values contain both a function address and a static chain pointer for access to automatic variables of outer levels. This mode is only partially implemented since C does not use it.

`CSImode`, ...
"Complex Single Integer" mode stands for a complex number represented as a pair of `SImode` integers. Any of the integer and floating modes may have '`C`'

prefixed to its name to obtain a complex number mode. For example, there are `CQImode`, `CSFmode`, and `CDFmode`. Since C does not support complex numbers, these machine modes are only partially implemented.

`BImode` This is the machine mode of a bit-field in a structure. It is used only in the syntax tree, never in RTL, and in the syntax tree it appears only in declaration nodes. In C, it appears only in `FIELD_DECL` nodes for structure fields defined with a bit size.

The machine description defines `Pmode` as a C macro which expands into the machine mode used for addresses. Normally this is `SImode`.

The only modes which a machine description *must* support are `QImode`, `SImode`, `SFmode` and `DFmode`. The compiler will attempt to use `DImode` for two-word structures and unions, but this can be prevented by overriding the definition of `MAX_FIXED_MODE_SIZE`. Likewise, you can arrange for the C type `short int` to avoid using `HImode`. In the long term it might be desirable to make the set of available machine modes machine-dependent and eliminate all assumptions about specific machine modes or their uses from the machine-independent code of the compiler.

To help begin this process, the machine modes are divided into mode classes. These are represented by the enumeration type `enum mode_class` defined in `rtl.h`. The possible mode classes are:

`MODE_INT` Integer modes. By default these are `QImode`, `HImode`, `SImode`, `DImode`, `TImode`, and also `BImode`.

`MODE_FLOAT`
Floating-point modes. By default these are `QFmode`, `HFmode`, `SFmode`, `DFmode` and `TFmode`, but the MC68881 also defines `XFmode` to be an 80-bit extended-precision floating-point mode.

`MODE_COMPLEX_INT`
Complex integer modes. By default these are `CQImode`, `CHImode`, `CSImode`, `CDImode` and `CTImode`.

`MODE_COMPLEX_FLOAT`
Complex floating-point modes. By default these are `CQFmode`, `CHFmode`, `CSFmode`, `CDFmode` and `CTFmode`,

`MODE_FUNCTION`
Algol or Pascal function variables including a static chain. (These are not currently implemented).

`MODE_RANDOM`
This is a catchall mode class for modes which don't fit into the above classes. Currently `VOIDmode`, `BLKmode` and `EPmode` 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_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_UNIT_SIZE (`*m*`)`
: Returns the size in bits of the subunits of a datum of mode *m*. This is the same as `GET_MODE_SIZE` except in the case of complex modes and `EPmode`. For them, the unit size is the size of the real or imaginary part, or the size of the function pointer or the context pointer.

11.5 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)`.

  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`. Any attempt to create an expression of code `const_int` and value zero or one will return `const0_rtx` or `const1_rtx` as appropriate.

`(const_double:`*m* *i0* *i1*`)`
: Represents a 64-bit constant of mode *m*. All floating point constants are represented in this way, and so are 64-bit `DImode` integer constants.

  The two integers *i0* and *i1* together contain the bits of the value. If the constant is floating point (either single or double precision), then they represent a `double`. To convert them to a `double`, do

```
union { double d; int i[2];} u;
u.i[0] = CONST_DOUBLE_LOW(x);
u.i[1] = CONST_DOUBLE_HIGH(x);
```

  and then refer to `u.d`.

  The global variables `dconst0_rtx` and `fconst0_rtx` hold `const_double` expressions with value 0, in modes `DFmode` and `SFmode`, respectively. The macro `CONST0_RTX (`*mode*`)` refers to a `const_double` expression with value 0 in mode *mode*. The mode *mode* must be of mode class `MODE_FLOAT`.

`(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*, 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 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.

11.6 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.

  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.

  Each pseudo register number used in a function's RTL code is represented by a unique `reg` expression.

  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.

  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.

Such multi-word hardware register `reg` expressions must not be live across the boundary of a basic block. The lifetime analysis pass does not know how to record properly that several consecutive registers are actually live there, and therefore register allocation would be confused. The CSE pass must go out of its way to make sure the situation does not arise.

`(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.

The other use of `subreg` is to extract the individual registers of a multi-register value. Machine modes such as `DImode` and `EPmode` indicate values longer than a word, values which usually require two 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 defined, 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 `subreg` which contains a `mem` instead of a `reg` as its first operand. The reload pass eliminates these cases by reloading the `mem` into a suitable 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.

`(cc0)` This refers to the machine's condition code register. It has no operands and may not have a machine mode. It 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`).

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`.

One special thing about the condition code register is that instructions can set it 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`). Only instructions whose sole purpose is to set the condition code, and instructions that use the condition code, need mention `(cc0)`.

`(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.

11.7 RTL Expressions for Arithmetic

`(plus:`*m* *x* *y*`)`
Represents the sum of the values represented by *x* and *y* carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for mode *m*.

`(minus:`*m* *x* *y*`)`
Like `plus` but represents subtraction.

`(compare` *x* *y*`)`
Represents the result of subtracting *y* from *x* for purposes of comparison. The absence of a machine mode in the `compare` expression indicates that 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 `(cc0)`. And that is the only way this kind of expression may validly be used: as a value to be stored in the condition codes.

`(neg:`*m* *x*`)` Represents the negation (subtraction from zero) of the value represented by *x*, carried out in mode *m*. *x* must be valid for mode *m*.

`(mult:`*m* *x* *y*`)`
Represents the signed product of the values represented by *x* and *y* carried out in machine mode *m*. If *x* and *y* are both valid for mode *m*, this is ordinary size-preserving multiplication. Alternatively, both *x* and *y* may be valid for a different, narrower mode. This represents the kind of multiplication that generates a product wider than the operands. Widening multiplication and same-size multiplication are completely distinct and supported by different machine instructions; machines may support one but not the other.

`mult` may be used for floating point multiplication as well. Then *m* is a floating point machine mode.

`(umult:`*m* *x* *y*`)`
: Like `mult` but represents unsigned multiplication. It may be used in both same-size and widening forms, like `mult`. `umult` is used only for fixed-point multiplication.

`(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. If *x* and *y* are both valid for mode *m*, this is ordinary size-preserving division. Some machines have division instructions in which the operands and quotient widths are not all the same; such instructions are represented by `div` expressions in which the machine modes are not all the same.

`(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.

`(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. *x* must be valid for mode *m*, which must be a fixed-point mode.

`(and:`*m* *x* *y*`)`
: Represents the bitwise logical-and of the values represented by *x* and *y*, carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for mode *m*, which must be a fixed-point mode.

`(ior:`*m* *x* *y*`)`
: Represents the bitwise inclusive-or of the values represented by *x* and *y*, carried out in machine mode *m*. This is valid only if *x* and *y* both are valid for 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*. This is valid only if *x* and *y* both are valid for mode *m*, which must be a fixed-point mode.

`(lshift:`*m* *x* *c*`)`
: Represents the result of logically shifting *x* left by *c* places. *x* must be valid for the mode *m*, a fixed-point machine mode. *c* must be valid for a fixed-point mode; 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*.

  On some machines, negative values of *c* may be meaningful; this is why logical left shift and arithmetic left shift are distinguished. For example, Vaxes have no right-shift instructions, and right shifts are represented as left-shift instructions whose counts happen to be negative constants or else computed (in a previous instruction) by negation.

`(ashift:`*m* *x* *c*`)`
: Like `lshift` but for arithmetic left shift.

`(lshiftrt:`*m* *x* *c*`)`
`(ashiftrt:`*m* *x* *c*`)`
: Like `lshift` and `ashift` but for right shift.

`(rotate:`*m* *x* *c*`)`
`(rotatert:`*m* *x* *c*`)`
: Similar but represent left and right rotate.

`(abs:`*m* *x*`)` Represents the absolute value of *x*, computed in mode *m*. *x* must be valid for *m*.

`(sqrt:`*m* *x*`)`
: Represents the square root of *x*, computed in mode *m*. *x* must be valid for *m*. Most often *m* will be a floating point mode.

`(ffs:`*m* *x*`)` Represents the 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.

11.8 Comparison Operations

Comparison operators test a relation on two operands and are considered to represent the value 1 if the relation holds, or zero if it does not. 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.

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.

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 above example stands for 1 if the condition codes were set to say "zero" or "equal", 0 otherwise. Although the same comparison operators are used for this as may be used in other contexts on actual data, no confusion can result since the machine description would never allow both kinds of uses in the same context.

`(eq` *x* *y*`)` 1 if the values represented by *x* and *y* are equal, otherwise 0.

`(ne` *x* *y*`)` 1 if the values represented by *x* and *y* are not equal, otherwise 0.

`(gt` *x* *y*`)` 1 if the *x* is greater than *y*. If they are fixed-point, the comparison is done in a signed sense.

`(gtu x y)` Like `gt` but does unsigned comparison, on fixed-point numbers only.

`(lt x y)`

`(ltu x y)` Like `gt` and `gtu` but test for "less than".

`(ge x y)`

`(geu x y)` Like `gt` and `gtu` but test for "greater than or equal".

`(le x y)`

`(leu 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.

11.9 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 a assignment, indicating insertion of a value into the specified bit field.

`(sign_extract:SI 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.

  Which machine modes are valid for *loc* depends on the machine, but typically *loc* should be a single byte when in memory or a full word in a register.

`(zero_extract:SI 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.

11.10 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.

`(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.

11.11 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 `subreg` expression.

  The presence of `strict_low_part` says that the part of the register which is meaningful in mode *n*, but is not part of mode *m*, is not to be altered. Normally,

an assignment to such a subreg is allowed to have undefined effects on the rest of the register when *m* is less than a word.

11.12 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 `SImode`, 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 have any mode. This represents a "test" or "compare" instruction.

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.

`(return)` 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.

`(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` 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.

x may also be null—a null C pointer, no expression at all. Such a `(clobber (null))` expression 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`.

When a `clobber` expression for a register appears inside a `parallel` with other side effects, GNU CC guarantees that the register is unoccupied both before and after that insn. Therefore, it is safe for the assembler code produced by the insn to use the register as a temporary. You can clobber either a specific hard register or a pseudo register; in the latter case, GNU CC will allocate a hard register that is available there for use as a temporary.

If you clobber a pseudo register in this way, use a pseudo register 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.

`(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 in 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` 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 never appears in an actual insn. 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.

Three expression codes appear in place of a side effect, as the body of an insn, though strictly speaking they do not describe side effects as such:

`(asm_input` *s*`)`
Represents literal assembler code as described by the string *s*.

`(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.

11.13 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.

Explicit popping of the stack could be represented with these embedded side effect operators, but that would not be safe; the instruction combination pass could move the popping past pushes, thus changing the meaning of the code.

An instruction that can be represented with an embedded side effect could also be represented using `parallel` containing an additional `set` to describe how the address register is altered. This is not done because machines that allow these operations at all typically allow them wherever a memory address is called for. Describing them as additional parallel stores would require doubling the number of entries in the machine description.

11.14 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.

11.15 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, 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 `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.

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. 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). `jump_insn` insns have the same extra fields as `insn` insns, accessed in the same way.

`call_insn`
: The expression code `call_insn` is used for instructions that may do function calls. It is important to distinguish these instructions because they imply that certain registers and memory locations may be altered unpredictably.

  `call_insn` insns have the same extra fields as `insn` insns, accessed in the same way.

`code_label`
: A `code_label` insn represents a label that a jump insn can jump to. It contains one special field of data in addition to the three standard ones. It 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 '`Ln`' where *n* is the label number.

`barrier`
: Barriers are placed in the instruction stream after unconditional jump instructions to indicate that the jumps are unconditional. 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_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.

`NOTE_INSN_LOOP_BEG`
: Appears at the place in a loop that `continue` statements jump to.

These codes are printed symbolically when they appear in debugging dumps.

The machine mode of an insn is normally zero (`VOIDmode`), but the reload pass sets it to `QImode` if the insn needs reloading.

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.

`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 is not used on an insn whose pattern consists of a single `use`, `clobber`, `asm`, `addr_vec` or `addr_diff_vec` expression.

`LOG_LINKS (`*i*`)`
: A list (chain of `insn_list` expressions) of previous "related" insns: insns which store into registers values that are used for the first time in this insn. (An additional constraint is that neither a jump nor a label may come between the related insns). This list is set up by the flow analysis pass; it is a null pointer until then.

`REG_NOTES (`*i*`)`
: A list (chain of `expr_list` expressions) giving information about the usage of registers in this insn. This list is set up by the flow analysis pass; it is a null pointer until then.

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.

The `REG_NOTES` field of an insn is a similar chain but of `expr_list` expressions instead of `insn_list`. There are several kinds of register notes, which are distinguished by the machine mode of the `expr_list`, which in a register note is really understood as being an `enum reg_note`. The first operand *op* of the `expr_list` is data whose meaning depends on the kind of note. Here are the kinds of register note:

`REG_DEAD` The register *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.

`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` RTX.

`REG_EQUIV`
: The register that is set by this insn will be equal to *op* at run time, 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.)

  The value which the insn explicitly copies into the register may look different from *op*, but they will be equal at run time.

  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.

`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).

  The RTX *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. It tells the CSE pass how to think of that value.

`REG_RETVAL`
: This insn copies the value of a library call, and *op* is the first insn that was generated to set up the arguments for the library call.

  Flow analysis uses this note to delete all of a library call whose result is dead.

`REG_WAS_0`
: The register *op* contained zero before this insn. You can rely on this note if it is present; its absence implies nothing.

`REG_LIBCALL`
: This is the inverse of `REG_RETVAL`: it is placed on the first insn of a library call, and it points to the last one.

  Loop optimization uses this note to move an entire library call out of a loop when its value is constant.

`REG_NONNEG`
: The register *op* is known to have nonnegative value when this insn is reached.

For convenience, the machine mode in an `insn_list` or `expr_list` is printed using these symbolic codes in debugging dumps.

The only difference between the expression codes `insn_list` and `expr_list` is that the first operand of an `insn_list` is assumed to be an insn and is printed in debugging dumps as the insn's unique id; the first operand of an `expr_list` is printed in the ordinary way as an expression.

11.16 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` RTX as shown above is the entire body of the insn.

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.

Immediately after RTL generation, if the value of the subroutine is actually used, this call insn is always followed closely by an insn which refers to the register *r*. This remains true through all the optimizer passes until cross jumping occurs.

The following insn has one of two forms. Either it copies the value into a pseudo-register, like this:

```
(set (reg:m p) (reg:m r))
```

or (in the case where the calling function will simply return whatever value the call produced, and no operation is needed to do this):

```
(use (reg:m r))
```

Between the call insn and this following insn there may intervene only a stack-adjustment insn (and perhaps some `note` insns).

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.

11.17 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, 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 zero, and only one with value one.
- There is only one `pc` expression.
- There is only one `cc0` expression.
- There is only one `const_double` expression with mode `SFmode` and value zero, and only one with mode `DFmode` and value zero.
- No `label_ref` 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` 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` RTX is made for each output operand. However, these all share the vector which contains the sequence of input operands. Because this sharing is used later on to test whether two `asm_operands` RTX's come from the same statement, the sharing must be guaranteed to be preserved.
- 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 with an insn can exist temporarily. However, the shared structure is copied before the combiner is finished with the insn. This is done by `copy_substitutions` in `combine.c`.

12 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.

12.1 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 12.3 [RTL Template], page 94) is a vector of incomplete RTL expressions which show what the instruction should look like. It is incomplete because it may contain `match_operand` and `match_dup` expressions that stand for operands of the instruction.

   If the vector has only one element, that element is what the instruction should look like. If the vector has multiple elements, then the instruction looks like a `parallel` expression containing that many elements as 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 12.4 [Output Template], page 97.

When simple substitution isn't general enough, you can specify a piece of C code to compute the output. See Section 12.5 [Output Statement], page 98.

5. Optionally, some *machine-specific information*. The meaning of this information is defined only by an individual machine description; typically it might say whether this insn alters the condition codes, or how many bytes of output it generates.

   This operand is written as a string containing a C initializer (complete with braces) for the structure type `INSN_MACHINE_INFO`, whose definition is up to you (see Section 13.11 [Misc], page 144).

12.2 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.

12.3 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* *pred* *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 *pred* 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.

pred 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 it returns zero, this instruction pattern fails to match. *pred* 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.

constraint controls reloading and the choice of the best register class to use for a value, as explained later (see Section 12.6 [Constraints], page 98).

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.

Most often, *pred* is `"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, *pred* 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, and it would prevent the register allocator from doing the best possible job.

For an operand that must be a constant, either *pred* should be `"immediate_operand"`, or the instruction pattern's extra condition should check for constants, 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_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 six 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 (code == PLUS || code == MULT || code == UMULT
          || code == AND || code == IOR || code == XOR);
}
```

Then the following pattern will match any RTL expression consisting of a commutative operator applied to two general operands:

```
(match_operator:SI 2 "commutative_operator"
  [(match_operand:SI 3 "general_operand" "g")
   (match_operand:SI 4 "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 3 and 4 of the insn. (This is done by the two instances of `match_operand`.) Operand 2 of the insn will be the entire commutative expression: use `GET_CODE (operands[2])` 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 2 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 3 and 4 of the gen-function. The subexpressions of argument 2 are not used; only its expression code matters.

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.

12.4 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.

12.5 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 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 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\");
  ")
```

12.6 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.

12.6.1 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).

When the constraint letter '`o`' is used, the reload pass may generate instructions which copy a nonoffsettable address into an index register. The idea is that the register can be used as a replacement offsettable address. But this method requires that there be patterns to copy any kind of address into a register. Auto-increment and auto-decrement addresses are an exception; there need not be an instruction that can copy such an address into a register, because reload handles these cases specially.

Most older machine designs have "load address" instructions which do just what is needed here. Some RISC machines do not advertise such instructions, but the possible addresses on these machines are very limited, so it is easy to fake them.

'`<`' 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`', ...
Other letters in the range '`I`' through '`M`' 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.

'`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.

'`n`' (a digit)
An operand that matches operand number *n* is allowed. If a digit is used together with letters, 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 matching constraint must appear in an input-only operand and the operand that it matches must be an output-only operand with a lower number. Thus, operand *n* must have '`=`' in its 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 accompanies by `address_operand` as the predicate in the `match_operand`.

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 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.

12.6.2 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.

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. When operands must be copied into registers, the compiler will never choose this alternative as the one to strive for.

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.). For example:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "r,m")
        (const_int 0))]
  ""
  "*
  return (which_alternative == 0
          ? \"clrreg %0\" : \"clrmem %0\");
  ")
```

12.6.3 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.

12.6.4 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")))]
  ...)
```

Note that in previous versions of GNU CC the '%' constraint modifier always applied to operands 1 and 2 regardless of which operand it was written in. The usual custom was to write it in operand 0. Now it must be in operand 1 if the operands to be exchanged are 1 and 2.

'#' 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.

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")))]
  ...)
```

12.6.5 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.

12.7 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.

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.

The constraints on a '`mov`*m*' must allow any hard register to be moved to any other hard register (provided that `HARD_REGNO_MODE_OK` permits mode *m* in both registers).

It is obligatory to support floating point '`mov`*m*' instructions into and out of any registers that can hold fixed point values, because unions and structures

(which have modes `SImode` or `DImode`) can be in those registers and they may have floating point members.

There may also be a need to support fixed point '`movm`' 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 '`movm`' instructions must be designed to avoid ever trying to reload into a floating point register.

'`movstrictm`'
: Like '`movm`' except that if operand 0 is a `subreg` with mode *m* of a register whose natural mode is wider, the '`movstrictm`' instruction is guaranteed not to alter any of the register except the part which belongs to mode *m*.

'`addm3`'
: 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.

'`subm3`', '`mulm3`', '`umulm3`', '`divm3`', '`udivm3`', '`modm3`', '`umodm3`', '`andm3`', '`iorm3`', '`xorm3`'
: Similar, for other arithmetic operations.

  There are special considerations for register classes for logical-and instructions, affecting also the macro `PREFERRED_RELOAD_CLASS`. They apply not only to the patterns with these standard names, but to any patterns that will match such an instruction. See Section 13.4 [Register Classes], page 127.

'`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.

'`divmodm4`'
: 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.

'`udivmodm4`'
: Similar, but does unsigned division.

'`ashlm3`'
: 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*.

'`ashrm3`', '`lshlm3`', '`lshrm3`', '`rotlm3`', '`rotrm3`'
: 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 '`ashlm3`' and leave '`lshlm3`' undefined.

There are special considerations for register classes for shift instructions, affecting also the macro `PREFERRED_RELOAD_CLASS`. They apply not only to the patterns with these standard names, but to any patterns that will match such an instruction. See Section 13.4 [Register Classes], page 127.

'`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 ...)))
```

Each such definition in the machine description, for integer mode *m*, must have a corresponding '`tst`*m*' pattern, because optimization can simplify the compare into a test when operand 1 is zero.

'`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 ...))
```

'`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.

'`cmpstr`*m*' Block compare instruction, with operands like '`movstr`*m*' except that the two memory blocks are compared byte by byte in lexicographic order. The effect of the instruction is to set the condition codes.

'`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 `Simode`. Operand 1 may have mode `QImode` or `SImode`; often `SImode` is allowed only for registers. Operands 2 and 3 must be valid for `SImode`.

  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 `SImode`) 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 `QImode` or `SImode`; often `SImode` is allowed only for registers. Operands 1 and 2 must be valid for `SImode`.

  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 must be omitted from the machine description. You must tell the compiler exactly which value is stored by defining the macro `STORE_FLAG_VALUE`.

'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*.

'`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.

'`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.

'`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.

'`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.

'`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 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.)
5. A label that precedes the table itself.

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 absolute address to jump to; otherwise, it is an offset which counts from the address of the table.

  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.

12.8 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.

12.9 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. These patterns 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.

The compiler does more with RTL than just create it from patterns and recognize the patterns: it can perform arithmetic expression codes when constant values for their operands can be determined. As a result, sometimes having one pattern can require other patterns.

For example, the Vax has no 'and' instruction, but it has 'and not' instructions. Here is the definition of one of them:

```
(define_insn "andcbsi2"
  [(set (match_operand:SI 0 "general_operand" "")
        (and:SI (match_dup 0)
                (not:SI (match_operand:SI
                          1 "general_operand" ""))))]
  ""
  "bicl2 %1,%0")
```

If operand 1 is an explicit integer constant, an instruction constructed using that pattern can be simplified into an 'and' like this:

```
(set (reg:SI 41)
     (and:SI (reg:SI 41)
             (const_int 0xffff7fff)))
```

(where the integer constant is the one's complement of what appeared in the original instruction).

To avoid a fatal error, the compiler must have a pattern that recognizes such an instruction. Here is what is used:

```
(define_insn ""
  [(set (match_operand:SI 0 "general_operand" "")
        (and:SI (match_dup 0)
                (match_operand:SI 1 "general_operand" "")))]
  "GET_CODE (operands[1]) == CONST_INT"
  "*
{ operands[1]
    = gen_rtx (CONST_INT, VOIDmode, ~INTVAL (operands[1]));
  return \"bicl2 %1,%0\";
}")
```

Whereas a pattern to match a general 'and' instruction is impossible to support on the Vax, this pattern is possible because it matches only a constant second argument: a special case that can be output as an 'and not' instruction.

A "compare" instruction whose RTL looks like this:

```
(set (cc0) (compare operand (const_int 0)))
```

may be simplified by optimization into a "test" like this:

```
(set (cc0) operand)
```

So in the machine description, each "compare" pattern for an integer mode must have a corresponding "test" pattern that will match the result of such simplification.

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.

When an instruction has the constraint letter '`o`', the reload pass may generate instructions which copy a nonoffsettable address into an index register. The idea is that the register can be used as a replacement offsettable address. In order for these generated instructions to work, there must be patterns to copy any kind of valid address into a register.

Most older machine designs have "load address" instructions which do just what is needed here. Some RISC machines do not advertise such instructions, but the possible addresses on these machines are very limited, so it is easy to fake them.

Auto-increment and auto-decrement addresses are an exception; there need not be an instruction that can copy such an address into a register, because reload handles these cases in a different manner.

12.10 Defining Jump Instruction Patterns

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_INSN (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, 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.

12.11 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"
  "machine-specific info")
```

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` 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_INSN` to find the preceding insns (but be careful to skip over any `note` insns that intervene).

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_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)))])
   ...]
  ...)
```

12.12 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 need 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 operations (addition, multiplication, shifting, etc.).

Do not emit any insns explicitly with `emit_insn` before failing.

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, 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`.

13 Machine Description Macros

The other half of the machine description is a C header file conventionally given the name `tm-`*machine*`.h`. The file `tm.h` should be a link to it. The header file `config.h` includes `tm.h` and most compiler source files include `config.h`.

13.1 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.

`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.

`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.

`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 `tm-`*`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}}
```

`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.

13.2 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 13.1 [Run-time Target], page 119.

`BITS_BIG_ENDIAN`
: Define this macro if the most significant bit in a byte has the lowest number. 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 if the most significant byte in a word has the lowest number.

`WORDS_BIG_ENDIAN`
: Define this macro 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.

`UNITS_PER_WORD`
: Number of storage units in a word; normally 4.

`POINTER_SIZE`
: Width of a pointer, in bits.

`POINTER_BOUNDARY`
: Alignment required for pointers stored in memory, 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.

`MAX_PARM_BOUNDARY`
: Largest alignment required for any stack parameters, in bits. If the data type of the parameter calls for more alignment than `PARM_BOUNDARY`, then it is given extra padding up to this limit.

  Don't define this macro if it would be equal to `PARM_BOUNDARY`; in other words, if the alignment of a stack parameter should not depend on its data type (as is the case on most machines).

`STACK_BOUNDARY`
: Define this macro if you wish to preserve a certain alignment for the stack pointer at all times. The definition is a C expression for the desired alignment (measured in bits).

`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.

`CONSTANT_ALIGNMENT (`*`code`*`, `*`typealign`*`)`
: A C expression to compute the alignment for a constant. The argument *typealign* is the alignment required for the constant's data type. *code* is the tree code of the constant itself.

  If this macro is not defined, the default is to use *typealign*. If you do define this macro, the value must be a multiple of *typealign*.

  The purpose of defining this macro is usually to cause string constants to be word aligned so that `dhrystone` can be made to run faster.

`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 a certain bizarre behavior pattern of some instances of PCC: a bit field whose declared type is `int` has the same effect on the size and alignment of a structure as an actual `int` would have.

  Just what effect that is in GNU CC depends on other parameters, but on most machines it would force the structure's alignment and size to a multiple of 32 or `BIGGEST_ALIGNMENT` bits.

`MAX_FIXED_MODE_SIZE`
: An integer expression for 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.

`CHECK_FLOAT_VALUE (`*`mode`*`, `*`value`*`)`
: A C statement to validate the value *value* (or 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.

13.3 Register Usage

`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.

`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.

`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.)

`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 num-

ber *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.

`HARD_REGNO_REGS (`*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.

GNU CC assumes that it can always move values between registers and (suitably addressed) memory locations. If it is impossible to move a value of a certain mode between memory and certain registers, then `HARD_REGNO_MODE_OK` must not allow this mode in those registers.

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.

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 insn'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.

`PC_REGNUM`

If the program counter has a register number, define this as that register number. Otherwise, do not define it.

`STACK_POINTER_REGNUM`

The register number of the stack pointer register, which must also be a fixed register according to `FIXED_REGISTERS`. On many 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.

`FRAME_POINTER_REQUIRED`

A C expression which is nonzero if a function must have and use a frame pointer. This expression is evaluated twice: at the beginning of generating RTL, and in the reload pass. If its value is nonzero at either time, then 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.

`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`.

`STATIC_CHAIN_REGNUM`

The register number used for passing a function's static chain pointer. This is needed for languages such as Pascal and Algol where functions defined within other functions can access the local variables of the outer functions; it is not currently used because C does not provide this feature, but you must define the macro.

The static chain register need not be a fixed register.

`STRUCT_VALUE_REGNUM`

When a function's value's mode is `BLKmode`, the value is not returned according to `FUNCTION_VALUE`. Instead, the caller passes the address of a block of memory in which the value should be stored.

If this value 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 a `mem` RTX, 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. If the value is a `mem`, the compiler assumes it is for an invisible first argument, and leaves space for it when finding the first real argument.

`REG_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).

If this macro is not defined, registers are used lowest numbered first (all else being equal).

One use of this macro is on the 360, where the highest numbered registers must always be saved and the save-multiple-registers instruction supports only sequences of consecutive registers. This macro is defined to cause the highest numbered allocatable registers to be used first.

13.4 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`.

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.

  In a shift instruction or a bitwise-and instruction, the mode of *x*, the value being reloaded, may not be the same as the mode of the instruction's operand. (They will both be fixed-point modes, however.) In such a case, *class* may not be a safe value to return. *class* is certainly valid for the instruction, but it may not be valid for reloading *x*. This problem can occur on machines such as the 68000 and 80386 where some registers can handle full-word values but cannot handle single-byte values.

  On such machines, this macro must examine the mode of *x* and return a subclass of *class* which can handle loads and stores of that mode. On the 68000, where address registers cannot handle `QImode`, if *x* has `QImode` then you must return

DATA_REGS. If *class* is ADDR_REGS, then there is no correct value to return; but the shift and bitwise-and instructions don't use ADDR_REGS, so this fatal case never arises.

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.

Two other special macros describe which constants 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 floating 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*.

13.5 Describing 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.

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.

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)
```

`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. (In the next version of GNU CC, the argument will be changed to the function data type rather than its declaration.)

`FIRST_PARM_CALLER_OFFSET (`*fundecl*`)`

Define this macro on machines where register parameters have shadow locations on the stack, at addresses below the nominal parameter. This matters because certain arguments cannot be passed on the stack. On these machines, such arguments must be stored into the shadow locations.

This macro should expand into a C expression whose value is the offset of the first parameter's shadow location from the nominal stack pointer value. (That value is itself computed by adding the value of `STACK_POINTER_OFFSET` to the stack pointer register.)

`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 actual allocation of such space would be done either by the call instruction or by the function prologue, or by defining FIRST_PARM_CALLER_OFFSET.

`STACK_ARGS_ADJUST (`*size*`)`

Define this macro if the machine requires padding on the stack for certain function calls. This is padding on a per-function-call basis, not padding for individual arguments.

The argument *size* will be a C variable of type `struct arg_data` which contains two fields, an integer named `constant` and an RTX named `var`. These together represent a size measured in bytes which is the sum of the integer and the RTX. Most of the time `var` is 0, which means that the size is simply the integer.

The definition should be a C statement or compound statement which alters the variable supplied in whatever way you wish.

Note that the value you leave in the variable `size` will ultimately be rounded up to a multiple of `STACK_BOUNDARY` bits.

This macro is not fully implemented for machines which have push instructions (i.e., on which `PUSH_ROUNDING` is defined).

`RETURN_POPS_ARGS (`*`funtype`*`)`

A C expression that should be 1 if a function pops its own arguments 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.

On the Vax, all functions always pop their arguments, so the definition of this macro is 1. 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.

`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_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.

`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.

`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.

`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.

`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 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.

`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.

`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.

`INIT_CUMULATIVE_ARGS (`*`cum`*`, `*`fntype`*`)`

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.

`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.

`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.

`FUNCTION_ARG_PADDING (`*`mode`*`, `*`size`*`)`

If defined, a C expression which determines whether, and in which direction, to pad out an argument with extra space. The value should be of type `enum direction`: either `upward` to pad above the argument, `downward` to pad below, or `none` to inhibit padding.

The argument *size* is an RTX which describes the size of the argument, in bytes. It should be used only if *mode* is `BLKmode`. Otherwise, *size* is 0.

This macro does not control the *amount* of padding; that is always just enough to reach the next multiple of `PARM_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_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*, but not if it is one of the call-used registers.

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.

On machines where arguments may be passed in registers, and not have stack space allocated, this macro must examine the variable `current_function_pretend_args_size`, and allocate that many bytes of uninitialized space on the stack just underneath the first argument arriving on the stack. (This may not be at the very end of the stack, 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 "pretend argument" space is allocated in functions that use the ANSI library `stdarg.h` to accept anonymous arguments of unspecified types; the last named argument is copied into the space, so that the anonymous arguments follow it consecutively.

`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.

`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.

`EXIT_IGNORES_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_IGNORES_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 instruc-

tions 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.

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 nonzero if the function should pop its own arguments. If so, use the variable `current_function_args_size` as the number of bytes to pop.

`FIX_FRAME_POINTER_ADDRESS (`*addr*`, `*depth*`)`
A C compound statement to alter a memory address that uses the frame pointer register so that it uses the stack pointer register instead. This must be done in the instructions that load parameter values into registers, when the reload pass determines that a frame pointer is not necessary for the function. *addr* will be a C variable name, and the updated address should be stored in that variable. *depth* will be the current depth of stack temporaries (number of bytes of arguments currently pushed). The change in offset between a frame-pointer-relative address and a stack-pointer-relative address must include *depth*.

Even if your machine description specifies there will always be a frame pointer in the frame pointer register, you must still define `FIX_FRAME_POINTER_ADDRESS`, but the definition will never be executed at run time, so it may be empty.

`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.

13.6 Library Subroutine Names

`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 `gnulib`.

`UMULSI3_LIBCALL`
A C string constant giving the name of the function to call for multiplication of one unsigned full-word by another. If you do not define this macro, the default name is used, which is `__umulsi3`, a function defined in `gnulib`.

`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 `gnulib`.

`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 `gnulib`.

`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 `gnulib`.

`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 `gnulib`.

`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`.

13.7 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 whose value is an integer. This includes integers whose values are not explicitly known, such as `symbol_ref` and `label_ref` expressions and `const` arithmetic expressions.

  On most machines, this can be defined as `CONSTANT_P (`*x*`)`, but a few machines are more restrictive in which constant addresses are supported.

`MAX_REGS_PER_ADDRESS`
: A number, the maximum number of registers that can appear in a valid memory address.

`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 as legitimate addresses.

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`.

`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 either *x* is a `const_double` or it satisfies `CONSTANT_P`, so you need not check these things. In fact, '`1`' is a suitable definition for this macro on machines where any `const_double` is valid and anything `CONSTANT_P` is valid.

13.8 Parameters for Delayed Branch Optimization

`HAVE_DELAYED_BRANCH`

Define this macro if the target machine has delayed branches, that is, a branch does not take effect immediately, and the actual branch instruction may be followed by one or more instructions that will be issued before the PC is actually changed.

If defined, this allows a special scheduling pass to be run after the second jump optimization to attempt to reorder instructions to exploit this. Defining this macro also requires the definition of certain other macros described below.

`DBR_SLOTS_AFTER (`*insn*`)`

This macro must be defined if `HAVE_DELAYED_BRANCH` is defined. Its definition should be a C expression returning the number of available delay slots following the instruction(s) output by the pattern for *insn*. The definition of "slot" is machine-dependent, and may denote instructions, bytes, or whatever.

`DBR_INSN_SLOTS (`*insn*`)`

This macro must be defined if `HAVE_DELAYED_BRANCH` is defined. It should be a C expression returning the number of slots (typically the number of machine instructions) consumed by *insn*.

You may assume that *insn* is truly an insn, not a note, label, barrier, dispatch table, `use`, or `clobber`.

`DBR_INSN_ELIGIBLE_P (`*insn*`, `*dinsn*`)`
A C expression whose value is non-zero if it is legitimate to put *insn* in the delay slot following *dinsn*.

You do not need to take account of data flow considerations in the definition of this macro, because the delayed branch optimizer always does that. This macro is needed only when certain insns may not be placed in certain delay slots for reasons not evident from the RTL expressions themselves. If there are no such problems, you don't need to define this macro.

You may assume that *insn* is truly an insn, not a note, label, barrier, dispatch table, `use`, or `clobber`. You may assume that *dinsn* is a jump insn with a delay slot.

`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.

13.9 Condition Code Information

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`.

`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.

`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)`.

If there are insn 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`.

13.10 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_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`.

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_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*.

13.11 Miscellaneous Parameters

`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.

`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.

`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`'.

`SCCS_DIRECTIVE`
: Define this if the preprocessor should ignore `#sccs` directives and print no error message.

`HAVE_VPRINTF`
: Define this if the library function `vprintf` is available on your system.

`MOVE_MAX` The maximum number of bytes that a single instruction can move quickly from memory to memory.

`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.

`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 slow (requires more than one instruction).

`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`.

`SHIFT_COUNT_TRUNCATED`
: Define this macro if shift instructions ignore all but the lowest few bits of the shift count. It implies that a sign-extend or zero-extend instruction for the shift count can be omitted.

`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.

`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.

`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.

`STORE_FLAG_VALUE`
: A C expression for the value stored by a store-flag instruction (`s`*`cond`*) when the condition is true. This is usually 1 or -1; it is required to be an odd number or a negative number.

  Do 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`.

`INSN_MACHINE_INFO`
: This macro should expand into a C structure type to use for the machine-dependent info field specified with the optional last argument in `define_insn` and `define_peephole` patterns. For example, it might expand into `struct machine_info`; then it would be up to you to define this structure in the `tm.h` file.

  You do not need to define this macro if you do not write the optional last argument in any of the patterns in the machine description.

`DEFAULT_MACHINE_INFO`
: This macro should expand into a C initializer to use to initialize the machine-dependent info for one insn pattern. It is used for patterns that do not specify the machine-dependent info.

  If you do not define this macro, zero is used.

`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`*`)`.

`DOLLARS_IN_IDENTIFIERS`
: Define this to be nonzero if the character '`$`' should be allowed by default in identifier names.

`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 macro, unlike most, describes the machine that the compiler is running on, rather than the one the compiler is compiling for. Therefore, it should be set in the `xm-`*`machine`*`.h` file rather than in the `tm-`*`machine`*`.h` file.

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.

13.12 Output of Assembler Code

`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 `tm-sun3.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 `LIBS_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`.

`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.

The prefix specified by the '`-B`' option, if any, is tried before the default prefix. After the default prefix, if the executable is not found that way, `/usr/lib/gcc-` is tried next; then the directories in your search path for shell commands are searched.

`STANDARD_STARTFILE_PREFIX`
: Define this macro as a C string constant if you wish to override the standard choice of `/usr/local/lib/` as the default prefix to try when searching for startup files such as `crt0.o`.

  In this search, all the prefixes tried for executable files are tried first. Then comes the default startfile prefix specified by this macro, followed by the prefixes `/lib/` and `/usr/lib/` as last resorts.

`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 `tm-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 `tm-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_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.

`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.

`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`.

`SELECT_SECTION (`*exp*`)`
: 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. Select the section by calling `text_section` or one of the alternatives for other sections.

  Do not define this macro if you use only the standard two sections and put all read-only variables and constants in 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 use only the standard two sections and put all constants in the text section.

`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.

`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.

`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 debugging output for SDB in response to the '`-g`' option.

`PUT_SDB_`*`op`*
: 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_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.

`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.

`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_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_LABELREF (`*stream*`, `*name*`)`
: A C statement to output to the stdio stream *stream* a reference in assembler syntax to a label named *name*. The character '`_`' should be added 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_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_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.

`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_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.

`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.

`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_DOUBLE (stream, value)`
A C statement to output to the stdio stream *stream* an assembler instruction to assemble a `double` constant whose value is *value*. *value* will be a C expression of type `double`.

`ASM_OUTPUT_FLOAT (stream, value)`
A C statement to output to the stdio stream *stream* an assembler instruction to assemble a `float` constant whose value is *value*. *value* will be a C expression of type `float`.

`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 a `int`, `short` or `char` constant whose value is *value*. The argument *exp* will be an RTL expression which represents a constant value. Use '`output_addr_const (exp)`' to output this value as an assembler expression.

`ASM_OUTPUT_DOUBLE_INT (stream, exp)`
A C statement to output to the stdio stream *stream* an assembler instruction to assemble a `long long` constant whose value is *exp*. The argument *exp* will be an RTL expression which represents a constant value. It may be a `const_double` RTX, or it may be an ordinary single-precision constant. In the latter case, you should zero-extend it.

`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_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_SKIP (stream, nbytes)`
A C statement to output to the stdio stream *stream* an assembler instruction to advance the location counter by *nbytes* bytes. *nbytes* will be a C expression of type `int`.

`ASM_OUTPUT_ALIGN (stream, power)`
A C statement to output to the stdio stream *stream* an assembler instruction to advance the location counter to a multiple of 2 to the *power* bytes. *power* will be a C expression of type `int`.

`ASM_OUTPUT_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_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_SOURCE_FILENAME (`*`stream`*`, `*`name`*`)`
: A C statment to output DBX or SDB 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 debugging information for the debugger in use is appropriate.

`ASM_OUTPUT_SOURCE_LINE (`*`stream`*`, `*`line`*`)`
: A C statment 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.

`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`'.

`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.

`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 ")"
```

14 The Configuration File

The configuration file `xm-machine.h` contains macro definitions that describe the machine and system on which the compiler is running. Most of the values in it 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:

`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.

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`.

Table of Contents