The Phasewise File Groups of GCC

GCC Version 4.0.2

Abhijat Vichare (amvichare@iitb.ac.in)

Indian Institute of Technology, Bombay
(http://www.iitb.ac.in)

This is edition 1.0 of "The Phasewise File Groups of GCC", last updated on January 7, 2008., and is based on GCC version 4.0.2.

Short Contents

Table of Contents

1 Introduction

In this document we note the details of the GCC 4.0.2 implementation given the background of the architecture described in [The Conceptual Structure of GCC], page 10, and the corresponding implementation described in [The Implementation of GCC], page 10. Figure 1.1 that appears below succinctly captures the core concepts in GCC. We also take support from the GCC Internals documentation (see [GCC Internals (by Richard Stallman)], page 10) available for a few versions of GCC which describe in detail the uses of various macros and RTL objects in detail. The source layout structure described in [GCC – An Introduction], page 10 is used.

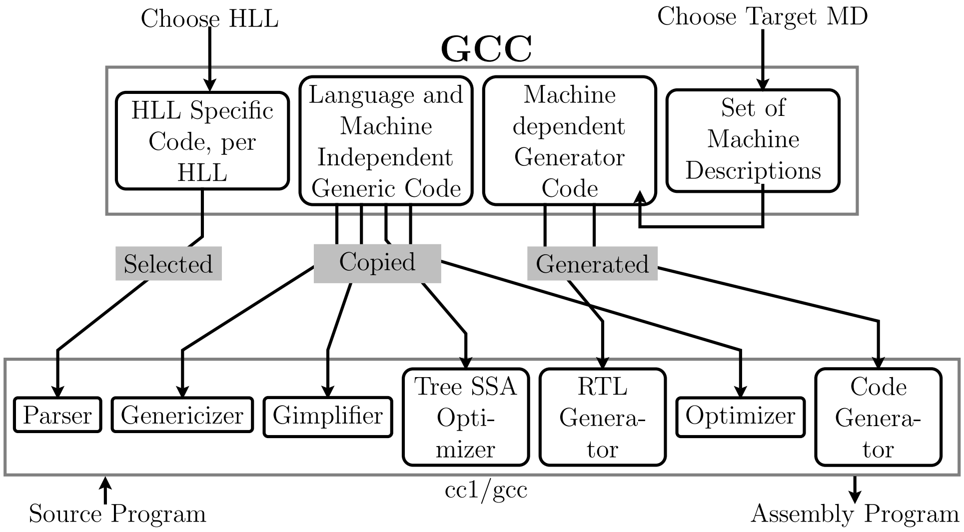

Figure 1.1: The GCC Compiler Generation Framework (CGF).

The various details follow. Some more details can also be added as reference in the future. For instance, systematic grouping and description of accessor macros that are used to access and manipulate internal data structures like the AST/Generic trees, RTL objects and details of target characteristics etc. can be added here. In some cases, the description has been derived from the comments in the source files themselves.

2 Pristine File Groups

See section "Source Organization" in *GCC – An Introduction*.

3 The Machine Description Processing Programs

This section summarises a few "generator" programs that process the information in the selected MD at t_{build} . Most of the synopses and descriptions are extracted from the commentary in the source files themselves. These form the first of the details of the target specific code generation activity at build time. The main goal is to generate the target specific RTL part of the Gimple → translation table (see [The Implementation of GCC], page 10, [The Conceptual Structure of GCC], page 10). The code number of an insn is simply its position in the machine description. They are assigned sequentially to entries in the description, starting with code number 0.

`gensupport.c`	Support routines for the various generation passes.
`genconditions.c`	Calculate constant conditions.
`genconstants.c`	Generate a series of `#define`s, one for each constant named in a (`define_constants` ...) pattern.
`genflags.c`	Generate flags `HAVE_...` saying which standard instructions are available for this machine.
`genconfig.c`	Generate some `#define` configuration flags.
`gencodes.c`	Generate some macros `CODE_FOR_...` giving the *insn_code_number* value for each of the defined standard insn names.
`genpreds.c`	Generate some macros `CODE_FOR_...` giving the *insn_code_number* value for each of the defined standard insn names.
`genattr.c`	Generate attribute information (insn-attr.h).
`genattrtab.c`	Generate code to compute values of attributes.
`genemit.c`	Generate code to emit insns as rtl.
`genextract.c`	Generate code to extract operands from insn
`genopinit.c`	Generate code to initialize optabs from machine description.
`genoutput.c`	Generate code to output assembler insns as recognized from RTL.
`genpeep.c`	Generate code to perform peephole optimizations.
`genrecog.c`	Generate code to recognize rtl as insns.
`gencheck.c`	Generate check macros for tree codes.
`gengenrtl.c`	Generate code to allocate RTL structures.
`genrtl.c`	Generated automatically by gengenrtl from `rtl.def.`
`gengtype.c`	Process source files and output type information.
`genautomata.c`	Pipeline hazard description translator.
`gengtype-lex.c`	A lexical scanner generated by flex
`gengtype-yacc.c`	A Bison parser, made from gengtype-yacc.y.
`gen-protos.c`	Massages a list of prototypes, for use by fixproto.

Table 3.1: A brief description of the various `gen` files. These files are compiled to the programs that process the chosen machine description to convert the information for internal use.

3.1 gensupport

Synopsis: Support routines for the various generation passes.

This file has a number of functions that are useful at various points of the target compiler generation. In particular, `init_md_reader` and `read_md_rtx` are used to setup the reading of a machine description file and reading a single rtx in it. The function `maybe_eval_c_test` takes a string representing a C test expression, looks it up in the condition table and reports whether or not its value is known at compile time.

3.2 genconditions

Synopsis: Calculate constant conditions.

Generates: `insn-conditions.c`

In a machine description, all of the insn patterns - `define_insn`, `define_expand`, `define_split`, `define_peephole`, `define_peephole2` - contain an optional C expression which makes the final decision about whether or not this pattern is usable. That expression may turn out to be always false when the compiler is built. If it is, most of the programs that generate code from the machine description can simply ignore the entire pattern.

3.3 genconstants

Synopsis: Generate a series of `#define` statements, one for each constant named in a (`define_constants` ...) pattern.

Generates: `insn-constants.h`

This program does not use gensupport.c because it does looks only at the `define_constants`.

3.4 genflags

Synopsis: Generate flags `HAVE_...` saying which simple standard instructions are available for this machine.

Generates: `insn-flags.h`

We scan the `define_insn`'s and `define_expand`'s in the machine description and look at "instructions" with names that are either not NULL or begin with any other character except a *. In other words, the so-called "standard instructions" are accepted, the rest are ignored. Thus we create a list of those "standard instructions" that the given processor "knows". An instruction in the MD file could have an associated condition expressed in C. This is the second "field" of the description of the instruction. The `genconditions` program would have already looked at each of these and memoized the compile time constants. The instruction pattern is practically non existent if the condition is false. We therefore, list out only those instruction patterns for which the condition is known to be true or it's value is not known at compile time. If the condition is known to be true, we define an "existence" macro. If the condition is not known at compile time, then we define the macro to be the condition itself. Note that the `genconditions` program is concerned with the conditions in all the RTL constructs, while we focus only on the "instructions" constructs, i.e. `define_insn` and `define_expand`. However, since the `genconditions` program has already looked at all the condition expressions and memoized them, we directly use the table that it constructs.

3.5 genconfig

Synopsis: Generate some `#define` configuration flags.

Generates: `insn-config.h`

e.g. (am i sure that what follows is an example of the comment above ?) flags to determine output of machine description dependent #define's.

3.6 gencodes

Synopsis: Generate some macros `CODE_FOR_...` giving the insn_code_number value for each of the defined standard insn names.

Generates: `insn-codes.h`

3.7 genpreds

Synopsis: Generate some macros `CODE_FOR_...` giving the insn_code_number value for each of the defined standard insn names.

3.8 genattr

Synopsis: Generate attribute information (insn-attr.h).

Generates: `insn-attr.h`

3.9 genattrtab

Synopsis: Generate code to compute values of attributes.

Generates: `insn-attrtab.c`

Uses: `genautomata.c` (for pipeline hazard description system in MD files)

This program handles insn attributes and the `define_delay` and `define_-function_unit` definitions.

It produces a series of functions named 'get_attr_....', one for each insn attribute. Each of these is given the rtx for an insn and returns a member of the enum for the attribute.

These subroutines have the form of a 'switch' on the INSN_CODE (via 'recog_-memoized'). Each case either returns a constant attribute value or a value that depends on tests on other attributes, the form of operands, or some random C expression (encoded with a SYMBOL_REF expression).

If the attribute 'alternative', or a random C expression is present, 'constrain_ope-rands' is called. If either of these cases of a reference to an operand is found, 'extract_insn' is called.

The special attribute 'length' is also recognized. For this operand, expressions involving the address of an operand or the current insn, (address (pc)), are valid. In this case, an initial pass is made to set all lengths that do not depend on address. Those that do are set to the maximum length. Then each insn that depends on an address is checked and possibly has its length changed. The process repeats until no further changed are made. The resulting lengths are saved for use by 'get_attr_length'.

A special form of `define_attr`, where the expression for default value is a CONST expression, indicates an attribute that is constant for a given run of the compiler. The subroutine generated for these attributes has no parameters as it does not depend on any particular insn. Constant attributes are typically used to specify which variety of processor is used.

Internal attributes are defined to handle `define_delay` and `define_function_unit`. Special routines are output for these cases.

This program works by keeping a list of possible values for each attribute. These include the basic attribute choices, default values for attribute, and all derived quantities.

As the description file is read, the definition for each insn is saved in a 'struct insn_def'. When the file reading is complete, a 'struct insn_ent' is created for each insn and chained to the corresponding attribute value, either that specified, or the default.

An optimization phase is then run. This simplifies expressions for each insn. EQ_ATTR tests are resolved, whenever possible, to a test that indicates when the attribute has the specified value for the insn. This avoids recursive calls during compilation.

The strategy used when processing `define_delay` and `define_function_unit` definitions is to create arbitrarily complex expressions and have the optimization simplify them.

Once optimization is complete, any required routines and definitions will be written.

An optimization that is not yet implemented is to hoist the constant expressions entirely out of the routines and definitions that are written. A way to do this is to iterate over all possible combinations of values for constant attributes and generate a set of functions for that given combination. An initialization function would be written that evaluates the attributes and installs the corresponding set of routines and definitions (each would be accessed through a pointer).

We use the flags in an RTX as follows:

'unchanging' (ATTR_IND_SIMPLIFIED_P): This rtx is fully simplified independent of the insn code.

'in_struct' (ATTR_CURR_SIMPLIFIED_P): This rtx is fully simplified for the insn code currently being processed (see optimize_attrs).

'integrated' (ATTR_PERMANENT_P): This rtx is permanent and unique (see attr_rtx).

'volatil' (ATTR_EQ_ATTR_P): During simplify_by_exploding the value of an EQ_ATTR rtx is true if !volatil and false if volatil.

3.10 genemit

Synopsis: Generate code to emit insns as rtl.

3.11 genextract

Synopsis: Generate code to extract operands from insn as rtl.

3.12 genopinit

Synopsis: Generate code to initialize optabs from machine description.

Generates: `insn-opinit.c`

Many parts of GCC use arrays that are indexed by machine mode and contain the insn codes for pattern in the MD file that perform a given operation on operands of that mode.

These patterns are present in the MD file with names that contain the mode(s) used and the name of the operation. This program writes a function 'init_all_op-tabs' that initializes the optabs with all the insn codes of the relevant patterns present in the MD file.

This array contains a list of optabs that need to be initialized. Within each string, the name of the pattern to be matched against is delimited with $(and $). In the string, $a and $b are used to match a short mode name (the part of the mode name not including 'mode' and converted to lower-case). When writing out the initializer, the entire string is used. $A and $B are replaced with the full name of the mode; $a and $b are replaced with the short form of the name, as above.

If $N is present in the pattern, it means the two modes must be consecutive widths in the same mode class (e.g, QImode and HImode). $I means that only full integer modes should be considered for the next mode, and $F means that only float modes should be considered. $P means that both full and partial integer modes should be considered.

$V means to emit 'v' if the first mode is a MODE_FLOAT mode.

For some optabs, we store the operation by RTL codes. These are only used for comparisons. In that case, $c and $C are the lower-case and upper-case forms of the comparison, respectively.

3.13 genoutput

Synopsis: Generate code to output assembler insns as recognized from rtl.

Generates: `insn-output.c`

This program reads the machine description for the compiler target machine and produces a file containing these things:

1. An array of 'struct insn_data', which is indexed by insn code number, which contains:
    1. 'name' is the name for that pattern. Nameless patterns are given a name.
    2. 'output' hold either the output template, an array of output templates, or an output function.
    3. 'genfun' is the function to generate a body for that pattern, given operands as arguments.
    4. 'n_operands' is the number of distinct operands in the pattern for that insn,
    5. 'n_dups' is the number of match_dup's that appear in the insn's pattern. This says how many elements of 'recog_data.dup_loc' are significant after an insn has been recognized.
    6. 'n_alternatives' is the number of alternatives in the constraints of each pattern.
    7. 'output_format' tells what type of thing 'output' is.
    8. 'operand' is the base of an array of operand data for the insn.
2. An array of 'struct insn_operand data', used by 'operand' above.
    1. 'predicate', an int-valued function, is the match_operand predicate for this operand.
    2. 'constraint' is the constraint for this operand. This exists only if register constraints appear in match_ope-rand rtx's.
    3. 'address_p' indicates that the operand appears within ADDRESS rtx's. This exists only if there are *no* register constraints in the match_operand rtx's.

4. 'mode' is the machine mode that that operand is supposed to have.
5. 'strict_low', is nonzero for operands contained in a STRICT_LOW_-PART.
6. 'eliminable', is nonzero for operands that are matched normally by MATCH_OPERAND; it is zero for operands that should not be changed during register elimination such as MATCH_OPERATORs.

Since the code number of an insn is simply its position in the machine description, the following entry in the machine description

```
(define_insn "clrdf"
  [(set (match_operand:DF 0 "general_operand" "")
        (const_int 0))]
  ""
  "clrd %0")
```

assuming it is the 25^{th} entry present, would cause `insn_data[24].template` to be `"clrd %0"`, and `insn_data[24].n_operands` to be `1`.

3.14 genpeep

Synopsis: Generate code to perform peephole optimizations.

3.15 genrecog

Synopsis: Generate code to recognize rtl as insns.

Generates: `insn-recog.c`

This program is used to produce insn-recog.c, which contains a function called 'recog' plus its subroutines. These functions contain a decision tree that recognizes whether an rtx, the argument given to recog, is a valid instruction.

recog returns -1 if the rtx is not valid. If the rtx is valid, recog returns a nonnegative number which is the insn code number for the pattern that matched. This is the same as the order in the machine description of the entry that matched. This number can be used as an index into various insn_* tables, such as insn_template, insn_outfun, and insn_n_operands (found in insn-output.c).

The third argument to recog is an optional pointer to an int. If present, recog will accept a pattern if it matches except for missing CLOBBER expressions at the end. In that case, the value pointed to by the optional pointer will be set to the number of CLOBBERs that need to be added (it should be initialized to zero by the caller). If it is set nonzero, the caller should allocate a PARALLEL of the appropriate size, copy the initial entries, and call add_clobbers (found in insn-emit.c) to fill in the CLOBBERs.

This program also generates the function 'split_insns', which returns 0 if the rtl could not be split, or it returns the split rtl as an INSN list.

This program also generates the function 'peephole2_insns', which returns 0 if the rtl could not be matched. If there was a match, the new rtl is returned in an INSN list, and LAST_INSN will point to the last recognized insn in the old sequence.

3.16 gencheck

Synopsis: Generate check macros for tree codes.

3.17 gengenrtl

Synopsis: Generate code to allocate RTL structures.

3.18 genrtl

Synopsis: Generated automatically by gengenrtl from `rtl.def`.

3.19 gengtype

Synopsis: Process source files and output type information.

3.20 genautomata

Synopsis: Pipeline hazard description translator.

3.21 gen-protos

Synopsis: A lexical scanner generated by flex

3.22 gengtype-lex

Synopsis: A Bison parser, made from gengtype-yacc.y.

3.23 gengtype-yacc

Synopsis: Massages a list of prototypes, for use by fixproto.

References

1. Abhijat Vichare.
   GCC – An Introduction
   (http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs/html/gcc-basic-info.html)

   2007.
2. Abhijat Vichare.
   The Implementation of GCC
   (http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs/html/gcc-implementation-details.html)

   2007.
3. Abhijat Vichare.
   The Conceptual Structure of GCC
   (http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs/conceptual-structure)
   2007.
4. Richard. M. Stallman.
   GCC Internals
   (http://gcc.gnu.org/onlinedocs/gccint)
   2007.

List of Figures

Appendix A Copyright

A.1 GNU Free Documentation License