GCC 4.0.2 – The Implementation

GCC Version 4.0.2

Abhijat Vichare (amvichare@iitb.ac.in)
Sameera Deshpande (sameera@cse.iitb.ac.in)

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

This is edition 1.0 of "GCC 4.0.2 – The Implementation", 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 models in [The Conceptual Structure of GCC], page 46 and are succinctly captured in Figure 1.1 which is taken from that description. The figure also marks three useful time periods and introduces the notation for each. We also take support from the GCC Internals documentation ([GCC Internals (by Richard Stallman)], page 46) available for a few versions of GCC which describe in detail the uses of various macros and RTL objects in detail. This document bridges the gap between a conceptual view of GCC in [The Conceptual Structure of GCC], page 46 and the "programmer's manual" view in [GCC Internals (by Richard Stallman)], page 46. It uses the source layout structure described in [GCC – An Introduction], page 46.

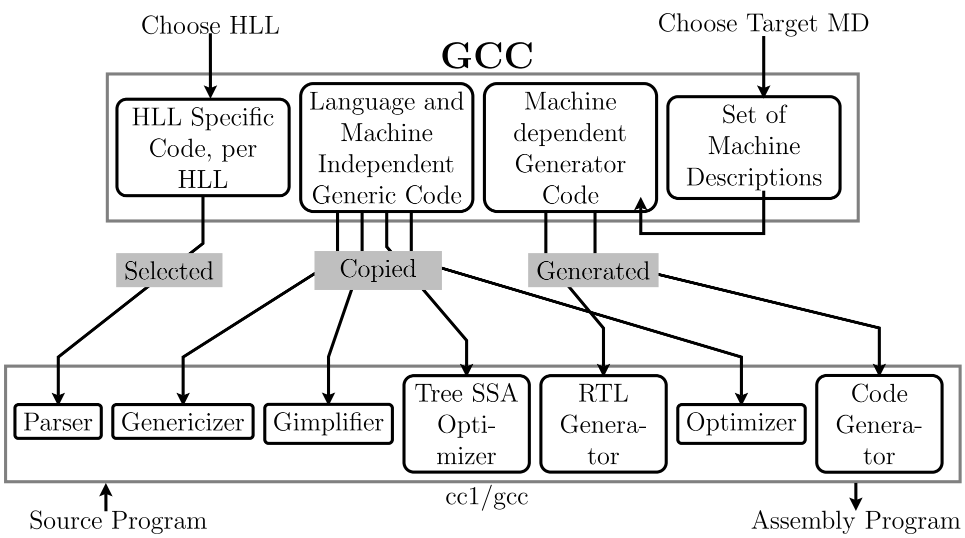

Figure 1.1: The GCC Compiler Generation Framework (CGF) and it's use to generate the target specific compiler (cc1/gcc) components. Some components of the compiler (cc1/gcc) are *selected* from the CGF, some are *copied* from the CGF and some are *generated* from the framework.

In general, the GCC source base makes a intensive use of source code level abstractions at $t_{develop}$. Most data structure manipulations are expressed via preprocessor macros. Repetitive coding is addressed by extracting the common patterns of code and data into single source objects that are then used when required. For example, the tree AST has node types common to both C and C++, node types of C, and a set of node types that augment these when C++ is supported. Thus repetitive node typing for C++ given that they already exist for C, has been avoided. To support retargetability, or multiple source languages, the central core of the compiler makes extensive use of function pointers at $t_{develop}$ that are "initialized" to the required function either at t_{build} or t_{run} . For instance, the central core merely makes a call to the "parser" function pointer. Before this call the function pointer is initialized to the parse function of the source language at input.

We avoid source code listings as it is available on the GNU web site. We recommend reading the source code along with this document to see the contexts clearly.

1.1 Document Scope

GCC is an industry strength implementation. It complies to a number of standards since it aims to support a few HLLs. Extensive error detection and reporting is implemented. As a *useful* compiler it also has code to support features useful for programming like debugging support (in various formats), timing of internal operations etc. All these aspects of GCC code is *not* detailed here. We focus on the Gimple and RTL IRs and in particular, retargetability of the back end machines. Most of the ignored part is handled conceptually when needed. A distinction must be made between a concept and the variety of ways in which it can be implemented. Thus, for example, although we conceptually describe the "selection" of the HLL specific parts, the implementation is actually in terms of defining HLL specific data structures (`lang_hooks`) that contain the HLL specific data. The actual "selection" thus occurs by chasing the information in these data structures.

1.2 Document Layout

In Chapter 2 [GCC Source Organization], page 3 we refresh some of the terms introduced in [GCC – An Introduction], page 46 and connect the compiler generation framework in Figure 1.1 to the source code structure. Chapter 3 [The Compilation Phases in GCC Source Code], page 5 gives a conceptual overview of the implementation structure of each part of the GCC phase sequence in Figure 1.1. The detailed description then begins. We have focused mainly on the Gimple and RTL phases of the GCC system and the other components are described in less detail and some have been skipped altogether as mentioned in [scope], page 2. However, the basic overall structure of the compiler is briefly described in Chapter 3 [The Compilation Phases in GCC Source Code], page 5. The next three sections give the implementation details of the three IRs of GCC: AST/Generic, Gimple and the RTL, at $t_{develop}$. Since most of Gimple is identical to the AST/Generic, we focus on the processing required at t_{build} time for Gimple in Chapter 5 [Gimple Implementation], page 22. The RTL is used at $t_{develop}$ as well as at t_{run} . We detail out the implementation issues of RTL for use at $t_{develop}$ and t_{run} including the transformations needed at t_{build} in Chapter 6 [RTL Implementation], page 27. The appendices provide some additional details like the phase sequence based file groups and the list of implemented targets as of GCC 4.0.2.

The AST/Generic is independent of the target machine but depends on the HLL selected at t_{build} . The Gimple is independent of the HLL as well the target. Hence the views at $t_{develop}$ and t_{run} match except for the Gimple $\rightarrow$ RTL translation. Further, in GCC 4.0.2 the Gimple is almost identical to the AST/Generic. Hence the Gimple details in Chapter 5 [Gimple Implementation], page 22 focus only on the differences relative to the AST.

2 GCC Source Organization

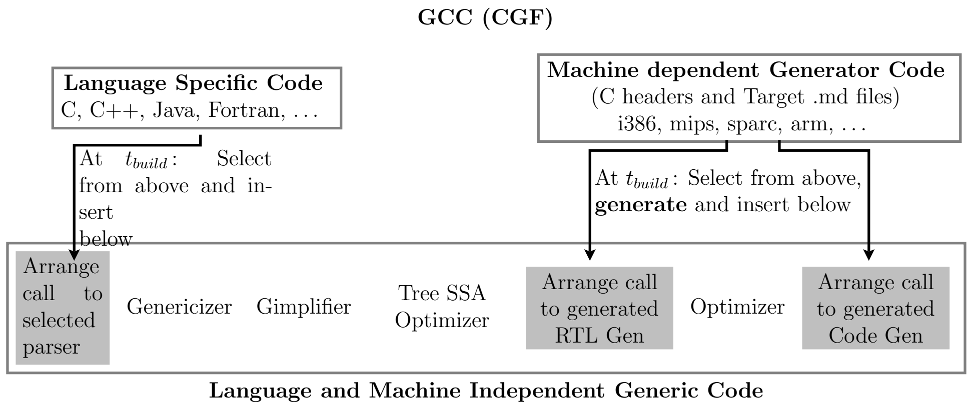

Figure 2.1: The source organization of the GCC Compiler Generation Framework (CGF). The arrows denote the points of insertion at t_{build} .

Given that a build system that adapts the GCC sources at $t_{develop}$ for the specific source language and the target system is required, we describe the organization of the source tree. This again, is a conceptual description that strives to build the intuition behind the structure that one obtains on unpacking the distribution. We emphasize that this is GCC 4.0.2 specific, and some variations exist across versions of GCC. We refer to the directory within which the GCC sources are unpacked as `$GCCHOME`.

Figure 2.1 describes the needs of the source organization at development time, $t_{develop}$. The HLL specific components (box labeled "Language Specific Code" in Figure 2.1), the back end components (box labeled "Machine dependent Generator Code" in Figure 2.1) and the actual compiler logic (box labeled "Language and Machine Independent Generic Code" in Figure 2.1) needs to be separated into distinct directories. A set of generator programs operate on these at build time, t_{build} to collect the components (e.g. parser, target specific RTL IR generator and the target specific code generator) for the chosen HLL and target pair. The the final HLL and target specific compiler sources (the lower half of Figure 1.1, labeled "cc1/gcc") are thus obtained and are subsequently compiled to obtain the binary that compiles programs in the chosen HLL to the chosen target at run time t_{run} . This strategy allows creating various kinds of compilers like native, cross or canadian cross.

The source and target independent parts of the compiler are within the `$GCCHOME/gcc` subdirectory of the main source trunk. It is in this directory that we find the code that

1. implements the complete *generic* compiler,
2. implements all the HLL and target independent manipulations, e.g. the optimization passes,
3. implements the HLL specific routines housed in a *separate* sub directory in `$GCCHOME/gcc/<HLL>` for each supported HLL, and
4. implements back end specific routines housed in a *separate* sub directory structure `$GCCHOME/gcc/config/<backend>`.

The currently supported front ends are: `C++`, Ada, Java, Fortran, Objective C and Treelang. Corresponding to each HLL, except C[1], is a subdirectory in `$GCCHOME/gcc` which all the code for processing that language exists. In particular this involves scanning the tokens of that language and creating the ASTs. If necessary, the basic AST tree node types need to be augmented with variations for this language. The main compiler calls these routines to handle input of that language. To isolate itself from the details of the source language, the main compiler uses a table of function pointers that are to be used to perform each required task. A language implementation needs to fill in such data structures of the main compiler code and build the language specific processing chain until the AST is obtained.

The back end specific code is organized as a list of directories corresponding to each supported back end system. This list of supported back ends is separately housed in `$GCCHOME/gcc/config` directory of the main trunk. The details of describing the back end target systems are in section Chapter 6 [RTL Implementation], page 27. Systematic development of these machine descriptions is in [Systematic Development of GCC Machine Descriptions], page 46.

Parts of the compiler that are common and find frequent usage have also been separated into a separate library called the `libiberty` and placed in a distinct subdirectory of `$GCCHOME`. This facilitates a one-time build of these common routines. We emphasize that these routines are common to the main compiler, the front end code and the back end code (e.g. regular expressions handling); the routines common to only the main compiler still reside in the main compiler directory, i.e. `$GCCHOME/gcc`. GCC also implements a garbage collection based memory management system for *it's* use during a run. This code is placed in the subdirectory `$GCCHOME/boehm-gc`. The main directory structure that results is shown in [GCC – An Introduction], page 46.

The details of the build process are in Chapter 7 [The GCC Build System Architecture], page 36 and the generated files are listed in [The Phase wise File Groups of GCC], page 46.

[1] GCC was originally aimed at being just a C compiler. Hence the C specific code is not well separated from the rest of the compiler.

3 The Compilation Phases in GCC Source Code

The implementation of the compiler proper – `cc1` for C – can be divided into the following operation time (t_{run}) phases:

1. Initialization.
2. Parsing and AST/Generic generation.
3. Gimplification, Gimple operations and Gimple → RTL conversion.
4. RTL expansion, RTL operations – in particular, operations that yield a strict RTL[1].
5. Assembly code generation.

Of the five phases above, the fourth and the fifth are target specific. The conversion to RTL part of the third phase is also target specific. For these parts and phases, the views at $t_{develop}$, t_{build} and t_{run} may differ and are so described in the rest of the document.

Start	`gcc/main.c`, `gcc/toplev.c`
Source Parser	`gcc/c-parse.y`
Parser–AST interface	`gcc/tree.def`, `gcc/c-tree.def`, `gcc/tree.h` and `gcc/tree.c`
AST → Gimple	`gimplify.c`
Gimple Optimizations	`gcc/tree-optimize.c`
Gimple → RTL	`stmt.c`, `expr.c`
RTL	`rtl.def`, `rtl.h`, `rtl.c`, `read-rtl.c`, `print-rtl.c`, `print-rtl1.c`, `optabs.c`
RTL Optimizations	`gcc/passes.c`
RTL → Target ASM	`gcc/final.c`

Table 3.1: Main GCC source files that implement main phases (all paths relative to `$GCCHOME`).

Table 3.1 lists out the main source files corresponding to the passes listed above[2]. In the following sections we describe the essential techniques used by GCC to implement each phase.

3.1 Initializations

The GCC implementation starts by initializing it's various subsystems. In particular the internationalization support and error reporting subsystems are initialized before processing the command line options. Initializations of the front end and the back end are done at this point. One particular initialization is the initialization of the garbage collector used by GCC in operation[3]. This phase may be considered to be over when the compiler calls the parser that starts accepting the input program for compilation.

---

[1] The RTL representation, when created from Gimple does not contain sufficient information to obtain the assembly code. Such RTL is called as non strict or incomplete. When the RTL passes finish computing all such information, the RTL can be converted to assembly code. Such RTL is called as strict or complete RTL.

[2] The passes are in currently `gcc/tree-optimize.c` and `gcc/passes.c`. In later versions, a full fledged passes manager has been implemented that includes the compilation and optimization phase sequence after parsing.

[3] Memory allocation is, therefore, separately done using `xmalloc()` allocator functions and no explicit free operations. Instead the garbage collector is used.

3.1.1 The "Initializations" Call Graph

The essential call graph is shown below along with the source file(s) that implement the functionality. The call graph is partial in that it shows the essential structure rather than the detail of every possible operation that needs to be done. Such detail is always available in the source file itself.

```
main ()                                     main.c
 toplev_main ()                             toplev.c
  general_init ()                           toplev.c
  decode_options ()                         toplev.c
  do_compile ()                             toplev.c
   compile_file()                           toplev.c
/* TO: Parsing */
```

3.2 The Parser

The bulk of the parsing code for C is in `$GCCHOME/gcc/c-parse.in` that gives `$GCCHOME/gcc/c-parse.y` This is a specification that `bison` reads in and generates the parser for C. The actions in some rules interface with the rest of the compiler. For instance, the action for a complete tree of a valid C function calls code to compile the generated tree based representation[4]. For other supported HLLs the parsers are in `$GCCHOME/gcc/<lang>/<lang-parse>.y`[5].

3.3 The AST/Generic

The AST/Generic is mainly composed of the passive data structure which is populated by the parser and consumed by the later compilation phases. The actual AST that will be formed at compilation time t_{run} is composed of tree fragments defined at $t_{develop}$ that are common across all the languages. The `gcc/tree.def` file defines the node types common to all languages. The `gcc/c-common.def` file is one component that collects nodes common to C and C++ into it. Any language specific addition to tree node types for languages other than C are added into the `gcc/<lang-dir>/<lang>-tree.def` file.

Having defined the various node types, organizing them (and other information) into tree structures is found in `gcc/tree.h`, while `gcc/tree.c` contains routines to use these data structures. The parser, for instance, uses routines or macros to instantiate tree nodes and populate them with the information extracted from the input. Chapter 4 [AST Implementation], page 14 describes all the details of the AST/Generic implementation.

3.3.1 The Parser + AST/Generic Call Graph

This continues from the initializations phase. Since the parse phase creates the AST representation we include that as a part of this call graph.

```
/* FROM: Initialisations */
  compile_file()                           toplev.c
```

[4] Later versions of GCC also support the "(compilation-)unit-at-time" mode in which the parser consumes the entire unit of compilation, i.e. a file, before calling the functions to perform the compilation.

[5] Later versions of GCC have hand coded recursive descent parsers, e.g. `$GCCHOME/gcc/c-parser.c` is the C parser code.

```
   lang_hooks.parse_file ()               toplev.c
    c_parse_file ()                     c-parser.c
     c_parser_translation_unit ()       c-parser.c
      c_parser_external_declaration ()  c-parser.c
       c_parser_declaration_or_fndef () c-parser.c
        finish_function ()                c-decl.c
/* TO: Gimplification */
```

3.4 The Gimple

In GCC 4.0.2 the Gimple IR is a subset of the AST/Generic tree nodes. The difference is that the Gimple uses only the sequencing and branching control flow constructs. All other control flow constructs are reduced to these two. Thus, a Gimple node is an instance of the `struct tree_common` (and other specialized tree structures) defined in `gcc/tree.h` ([The Conceptual Structure of GCC], page 46), except that the `code` field of any instance of that structure can have only the codes (in `gcc/tree.def`) for sequence and branch for control flow. Gimple in this form is said to be *unstructured*. Additionally, the Gimple phase also reduces complex expressions to simple ones by introducing any temporaries if required. This form of Gimple is said to be *structured*. The code that reduces the control flow is mainly in `gcc/gimplify.c`. The Gimple phase has three parts: conversion from AST/Generic to Gimple, Gimple based optimizations, and Gimple to RTL conversion. The critical part is the Gimple to RTL conversion. This is because while the Gimple part is target independent that RTL part is target specific and the problem is that while the target is known at build time t_{build} the conversion has to be implemented at development time $t_{develop}$.

The following insight is used to implement this translation. As described in [The Conceptual Structure of GCC], page 46, the conversion table for Gimple to RTL is divided into two parts: the target specific part and the target independent part. The target specific part is constructed at t_{build} and is *not* available at $t_{develop}$. A system of SPNs is used to semantically connect these two parts. The target independent part corresponds to associating the Gimple nodes (to be converted to target specific RTL) to corresponding SPNs. However, since this is a static activity, it has been hard coded into the conversion phase. The conversion code in `gcc/gimplify.c` (and the associated files), hence, implements the target independent part of the conversion table through a case analysis over Gimple nodes and arranges for invocation of the corresponding RTL routine indexed by the known SPN. The code need not contain direct references to the SPNs since using the index suffices. Thus, conceptually although the SPNs serve to separate the target specific and target independent parts of the conversion table, the implementation of the target independent parts is not required to directly refer to the SPNs.

3.4.1 The Gimple Call Graph

This continues from the AST/Generic phase.

```
/* FROM: Parsing */
c_genericize()                        c-gimplify.c
   gimplify_function_tree()             gimplify.c
      gimplify_body()                   gimplify.c
         gimplify_stmt()                gimplify.c
            gimplify_expr()             gimplify.c
```

```
lang_hooks.callgraph.expand_function()
 tree_rest_of_compilation()       tree-optimize.c
  tree_register_cfg_hooks()            cfghooks.c
   execute_pass_list()                   passes.c
/* TO: Gimple Optimizations passes */
```

The final call to the pass list runs the pass manager (see Section 3.6 [Optimizations], page 9). One of the passes in the pass manager is the RTL expander and the RTL passes sequencer.

3.5 The RTL

The RTL is used for two purposes in GCC: specifying target properties at $t_{develop}$ and representing a compilation internally at t_{run} . The RTL IR used at t_{run} is created during the build phase (t_{build}) by gleaning out information from the target MD files. At t_{build} the target specific part of the Gimple to RTL conversion table is created, as indicated in [The Conceptual Structure of GCC], page 46. C data structures for RTL are created in the build phase. To "join" the Gimple → RTL translation table separated at development time two main actions need to be taken at build time. First, the target independent part of the table must be "informed" of the exact "location" of the corresponding RTL pattern within the selected MD. This information is recorded in the `struct optab` data structure. Second, the target dependent part of the translation table must be created from the selected MD. Thus the actual RTL patterns must be recorded into a data structure, `struct insn_data`, that represents the target specific part of the translation table. Each pattern is recorded at that "location" in `struct insn_data` which corresponds to the information in `struct optab`. Thus `struct optab` supplies the index of the RTL pattern from `struct insn_data` that is to be used to represent the Gimple node in RTL. Note that the pattern names in the MD thus serve to identify the correspondences of the "locations" within the `optab` and `insn_data` arrays to be established. The `optab` table already "knows" the pattern names to seek. It merely records the occurrence within the MD.

Conceptually, the GCC build process lists out the patterns available in the target MD into a header file `insn-flags.h` by enumerating them via preprocessor defines that are non-zero if the pattern exists[6]. It then indexes the expansion patterns into an `enum` in `insn-codes.h`. These indexes are used to initialize the array of operations supported by the target – the `optabs` array, initialized in the `insn-opinit.c` file. The `optabs` structure is defined in `optabs.h` as:

```
struct optab
{
  enum rtx_code code;            /* enumerated from rtl.def */
  struct optab_handlers {
    enum insn_code insn_code;    /* in insn-codes.h */
    rtx libfunc;
  } handlers [NUM_MACHINE_MODES];
};
```

---

[6] C condition of patterns, if they exist, are used as the non-zero initializers, else the initialization value is 1.

```
typedef struct optab * optab;
```

Chapter 6 [RTL Implementation], page 27 describes all the details of implementing the MD-RTL and IR-RTL languages – usually referred to as simply "RTL" in the GCC source code.

3.5.1 The IR-RTL Call Graph

The call sequence for expanding Gimple IR to IR-RTL based IR is shown below. It is initiated by the handler of the `pass_expand` and is duplicated here for clarity.

```
/* FROM: Gimple optimization passes */
/* non strict RTL expander pass */
pass_expand_cfg                      cfgexpand.c
 expand_gimple_basic_block ()        cfgexpand.c
  expand_expr_stmt ()                     stmt.c
   expand_expr ()                         stmt.c
/* TO: non strict RTL passes: pass_rest_of_compilation */
```

3.6 Optimizations

GCC implements a large number of optimizations that are found in the literature. They are implemented at suitable places in the phase sequence. The nature of the optimization, e.g. control flow optimization, instruction scheduling etc., determines it's placement in the phase sequence. Since Gimple lowers control flow (see [The Conceptual Structure of GCC], page 46), once a compilation is represented in Gimple form, control flow optimizations can be implemented. Thus we find SSA being implemented after gimplification and instruction scheduling implemented after IR-RTL based representation of the compilation. Almost every optimization requires some analysis and corresponding computations before implementation. The GCC phase sequence thus has optimizing passes following the representation in a suitable IR.

GCC 4.0.2 implements a "pass manager" partially[7]. The pass manager resides in `$GCCHOME/gcc/tree-optimize.c`. Each pass is an instance of the `struct tree_opt_pass` (in `$GCCHOME/gcc/tree-pass.h`). One of the fields of this structure is the pass entry point function pointer. The instances of the implemented passes, i.e. the variables of type `struct tree_opt_pass`, are organized into a linear list in `$GCCHOME/gcc/tree-optimize.c`. Sub passes of a given pass in this list are organized as sub lists. The current implementation in GCC 4.0.2 organizes the Gimple based optimizations and IR-RTL conversion as a list of passes in a pass manager and the later versions also include the IR-RTL based passes. This list thus also represents the call structure. The complete pass list, with the sub passes indented and shown using the actual 72 `struct tree_opt_pass` variables, is:

```
pass_gimple
pass_remove_useless_stmts  /* Remove useless statements */
pass_mudflap_1  /* Mudflap pass 1 */
pass_lower_cf  /* Control flow lowering */
pass_lower_eh  /* Exception handling lowering */
pass_build_cfg  /* Build control flow graph */
```

[7] It has been completed in the later versions.

```
pass_pre_expand  /* Vector and Complex number expander */
pass_tree_profile  /* Tree profiler */
pass_init_datastructures  /* Initialize data structures */
pass_all_optimizations  /* List of all optimizations */
  pass_referenced_vars
  pass_build_ssa
  pass_may_alias
  pass_rename_ssa_copies
  pass_early_warn_uninitialized
  pass_dce  /* Dead code elimination */
  pass_dominator
  pass_redundant_phi
  pass_dce
  pass_merge_phi
  pass_forwprop  /* Forward propagation */
  pass_phiopt
  pass_may_alias
  pass_tail_recursion
  pass_ch  /* Loop header copying */
  pass_profile
  pass_sra
  pass_may_alias
  pass_rename_ssa_copies
  pass_dominator
  pass_redundant_phi
  pass_dce
  pass_dse  /* Dead store elimination */
  pass_may_alias
  pass_forwprop
  pass_phiopt
  pass_ccp  /* Conditional constant propagation */
  pass_redundant_phi
  pass_fold_builtins
  pass_may_alias
  pass_split_crit_edges
  pass_pre  /* Partial redundancy elimination */
  pass_loop
    pass_loop_init
    pass_lim  /* Loop invariant motion */
    pass_unswitch
    pass_record_bounds
    pass_linear_transform
    pass_iv_canon  /* Canonical induction variable creation */
    pass_if_conversion
    pass_vectorize
    pass_complete_unroll
    pass_iv_optimize
```

```
    pass_loop_done
  pass_dominator
  pass_redundant_phi
  pass_late_warn_uninitialized
  pass_cd_dce
  pass_dse
  pass_forwprop
  pass_phiopt
  pass_tail_calls
  pass_rename_ssa_copies
  pass_del_ssa
  pass_nrv  /* HLL independent return value optimization */
  pass_remove_useless_vars
  pass_mark_used_blocks
  pass_cleanup_cfg_post_optimizing
pass_warn_function_return
pass_mudflap_2
pass_free_datastructures
pass_expand  /* Expand to (incomplete) IR-RTL IR */
pass_rest_of_compilation  /* Do IR-RTL passes */
```

Once the compilation is in the IR-RTL based IR, the following functions may make passes over the IR depending on the conditionals that control the flow, and not shown here. Most of these functions are in `$GCCHOME/gcc/passes.c` and are entry points into the actual passes. The entire list is just the function calls in the body of the pass function of the `pass_rest_of_compilation` pass – the last pass – in the pass manager above.

```
remove_unnecessary_notes ()
init_function_for_compilation ()
rest_of_handle_jump ()
rest_of_handle_eh ()
emit_initial_value_sets ()
unshare_all_rtl ()
instantiate_virtual_regs ()
rest_of_handle_jump2 ()
rest_of_handle_cse ()
rest_of_handle_gcse ()
rest_of_handle_loop_optimize ()
rest_of_handle_jump_bypass ()
rest_of_handle_cfg ()
rtl_register_profile_hooks ()
rtl_register_value_prof_hooks ()
rest_of_handle_branch_prob ()
rest_of_handle_value_profile_transformations ()
count_or_remove_death_notes (NULL, 1)
rest_of_handle_if_conversion ()
rest_of_handle_tracer ()
rest_of_handle_loop2 ()
```

```
rest_of_handle_web ()
rest_of_handle_cse2 ()
rest_of_handle_life ()
rest_of_handle_combine ()
rest_of_handle_if_after_combine ()
rest_of_handle_partition_blocks ()
rest_of_handle_regmove ()
split_all_insns (1)
rest_of_handle_mode_switching ()
recompute_reg_usage ()
rest_of_handle_sms ()
rest_of_handle_sched ()
rest_of_handle_old_regalloc ()
rest_of_handle_postreload ()
rest_of_handle_gcse2 ()
rest_of_handle_flow2 ()
rest_of_handle_peephole2 ()
rest_of_handle_if_after_reload ()
rest_of_handle_regrename ()
rest_of_handle_reorder_blocks ()
rest_of_handle_branch_target_load_optimize ()
rest_of_handle_sched2 ()
rest_of_handle_stack_regs ()
compute_alignments ()
duplicate_computed_gotos ()
rest_of_handle_variable_tracking ()
free_bb_for_insn ()
rest_of_handle_machine_reorg ()
purge_line_number_notes (get_insns ())
cleanup_barriers ()
rest_of_handle_delay_slots ()
split_all_insns_noflow ()
convert_to_eh_region_ranges ()
rest_of_handle_shorten_branches ()
set_nothrow_function_flags ()
rest_of_handle_final ()
rest_of_clean_state ()
```

Perhaps the most critical and hairy function in this sequence is the register allocation pass. This pass computes the target specific hard registers to be used for the pseudo registers in the IR-RTL IR. The purpose of this pass is essentially to ensure that the IR-RTL representation of the compilation is complete enough so that each RTX corresponds to a unique target assembly string in the MD. It uses the so called reload pass that introduces the necessary load and store operations for instruction patterns that require hard registers. These register reloads are performed while satisfying the allocation constraints specified in the MD. This pass depends critically on sufficiently detailed specification of the data movement operations supported by the target.

3.7 Target assembly code emission

The register allocator pass in `rest_of_compilation ()` makes the assembly code generation by the `rest_of_handle_final ()` function a conceptually simple affair of substituting the concrete assembly syntax for the RTXs. At this point, the IR-RTL IR is to the assembly code as the AST is to the HLL code, and we regard the RTXs at this point as the "abstract syntax" of the final assembly code. The list of RTXs that represents the compilation as IR-RTL IR is simply traversed. For each RTX pattern encountered, the assembly string to be output is determined and emitted. The `insn-recog.c` file is generated from the machine description and contains a decision tree that compares a given instruction pattern in the IR-RTL IR of a program to the pattern specifications in the MD and determines the matching pattern, if any. If a match is found, then the corresponding assembly string is emitted. The basic algorithm is:

```
preprocess the MD to obtain the recognizer
for each instruction pattern in the IR-RTL IR of input program
   obtain the index from the recognizer
   use it to locate the assembly output in insn_data[]
```

Preprocessing the MD: At t_{build} the `genrecog` process scans the selected MD for occurrences of instruction patterns. The occurrence of the pattern expression in the MD file serves as the indexing integer into target specific arrays like `insn_data` that hold the actual detailed information. A given pattern is scanned for the various pieces of information that can be used for matching purposes. For instance, the machine mode of the operators and operands, or the "nature" of the operand (register, memory etc.). Corresponding match predicate expressions in C are constructed and emitted into the `insn-recog.c` file. This file yields the recognizer on compilation and contains the main entry point, `recog()`, of the recognizer.

3.7.1 The Assembly Emission Call Graph

The call sequence for emitting the assembly code from completed IR-RTL representation is shown below. It is initiated by the `rest_of_handle_final ()` function from the RTL passes in handler of the `pass_rest_of_compilation`.

```
/* FROM: RTL passes */
assemble_start_function ();       varasm.c
final_start_function ();           final.c
final ();                          final.c
final_end_function ();             final.c
assemble_end_function ();         varasm.c
```

4 AST Implementation

We describe the implementation of the AST IR in GCC. In general, this appears to be different from the conceptual presentation of the AST abstract machine. We first give the AST data structure that GCC actually uses. A list all the AST objects into their classes as defined by the GCC sources then follows. Conceptually, the AST/Generic views at development time and operation time are identical. Hence the t_{run} view is presented in terms of a partial call graph of the compiler.

4.1 The AST/Generic Data Structures

The AST is composed of a set of nodes. Some information is common to all nodes, and is collected in the `struct tree_common` structure in `tree.h`. The flags in this structure are documented in detail in `tree.h`. The "code" of the node, i.e. the kind of information that it contains, is expressed through a set of codes (documented in `tree.def`), is a field of the structure that is masked behind an accessor macro `TREE_CODE(NODE)` which simply returns this field. The `TREE_SET_CODE(NODE)` macro is the assignment macro that sets this field. This structure is included as a field of all nodes. The various tree nodes are:

Data Structure	**Information that the node contains**
`struct tree_int_cst`	integer constants.
`struct tree_real_cst`	real constants.
`struct tree_string`	string constants.
`struct tree_complex`	complex constants.
`struct tree_vector`	vector constants.
`struct tree_identifier`	Identifiers.
`struct tree_list`	Lists of tree nodes.
`struct tree_vec`	Vectors of tree nodes.
`struct tree_exp`	Expression node.
`struct tree_block`	Block definition node.
`struct tree_type`	Data type nodes.
`struct tree_decl`	Function Declaration.

The overall tree node is a union of all the various kinds of node structures listed above. It is given by the data structure (in `tree.h`) as:

```
union tree_node
{
  struct tree_common          common;
  struct tree_int_cst         int_cst;
  struct tree_real_cst        real_cst;
  struct tree_vector          vector;
  struct tree_string          string;
  struct tree_complex         complex;
  struct tree_identifier      identifier;
  struct tree_decl            decl;
  struct tree_type            type;
  struct tree_list            list;
```

```
  struct tree_vec            vec;
  struct tree_exp            exp;
  struct tree_block          block;
 };
```

Every member of this structure, except the `tree_common` structure has a field that points to the IR-RTL representation whose structure is presented in section, Section 6.1 [The RTX Data Structure], page 27. The compiler enumerates a number of data types corresponding explicitly to the source language types (for C, in our examples), and implicitly for internal purposes (e.g. mark errors, identify the main entry point etc.). The varieties of integers that the source (C) can represent and the corresponding integer type codes are exhaustively enumerated.

A retargetable architecture implies the possibility of a Canadian cross (see Chapter 7 [The GCC Build System Architecture], page 36). The differences in the characteristics of the build system, the host system and the target system have a few unusual consequences. Consider the situation when the word sizes on these three systems differ. The build system compiler has to build the compiler using the host system word size. The host system, further, has to build the target code using the target word size. Calculations involving word sizes, for instance pointer increment values, have to be calculated in the compiler sources depending on the run time faced by the object being built. A tree object is built on the host system while producing code for the target system, i.e. when the compiler runs to compile a file.

The `tree.def` file contains various node names defined using a C preprocessor macro – `DEFTREECODE`. The macro is used in different ways depending on the information required. We illustrate the use with an example. Consider the following macro that is represents the C `void` type:

```
DEFTREECODE (VOID_TYPE, "void_type", 't', 0)
```

Defining the `DEFTREECODE` macro as:

```
#define DEFTREECODE(arg1, arg2, arg3, arg4) arg2
```

yields the second argument which gives the name as a string. The definitions of the `DEFTREECODE` are changed as needed in the GCC sources. For instance, the various nodes are enumerated simply as:

```
#define DEFTREECODE(arg1, arg2, arg3, arg4)          arg1
enum tree_node_list {
#include "tree.def"
};
#undef DEFTREECODE
```

The technique is used at many places in the source, for example for the RTL definitions too.

The first argument of the `DEFTREECODE` macro is the symbolic name of the node typically used to create an enumerated data type of nodes. The second argument is the identifier used to refer to that node. The nodes in the GCC AST are of different kinds as listed in [kinds of AST nodes in GCC], page 16. These kinds are encoded via a set of character codes which are listed in `tree.def`. These codes are the third argument of the `DEFTREECODE` macro. The fourth argument in most node definitions is the number of operands of that node. For other nodes, the use of the fourth argument is dependent on the node being described. The collection of the `DEFTREECODE` macros define the database of nodes that GCC uses for it's AST.

4.2 AST/Generic Node types

Table 4.1–Table 4.12 list out all the node types for any C program. The list has been obtained by the common GCC tree node definitions data base in `tree.def`, and the node definitions for languages of the C family (C, objective C) in `c-common.def`. As is evident, the nodes listed below are a superset of the nodes required to represent any C program. Nodes for objects in other languages like Pascal or C++ also are a part of the `tree.def` file. The GCC code base classifies them into types as given by the 'code' value in the GCC tree node definition data bases. Codes have been defined for the following[1]:

1. Comparison expressions:
2. Unary arithmetic expressions:
3. Binary arithmetic expressions:
4. Lexical block:

   A symbol binding block. This captures the scoping rules into the intermediate representation of the program.
5. Constants:

   These node types represent constants of various types that can occur in the input program.
6. Declarations:

   All references to names are represented by nodes of this type.
7. Other kinds of expressions:
8. Storage referencing:

   Memory may be referenced in many ways in the source. It may be directly named, or may be referenced via a pointer, or an "offset" from a base of arrays or structures or unions, or bit fields
9. Expressions with inherent side effects:
10. Object types:

    These node types are used to represent each data type in the source language. Most C data types are represented. The `integer_type` also includes `char` in C. The `char_type` node denotes Pascal character type.
11. Miscellaneous:

[1] The GCC sources (`tree.def` etc.) describe each node type in great detail. We summarize.

`lt_expr`	$<$ operation, 2 operand
`le_expr`	$\leq$ operation, 2 operand
`gt_expr`	$>$ operation, 2 operand
`ge_expr`	$\geq$ operation, 2 operand
`eq_expr`	$=$ operation, 2 operand
`ne_expr`	$\neq$ operation, 2 operand
`unordered_expr`	Floating point unordered operations, 2 operand
`ordered_expr`	Floating point ordered operations, 2 operand
`unlt_expr`	Unordered $<$ operation, 2 operand
`unle_expr`	Unordered $\leq$ operation, 2 operand
`ungt_expr`	Unordered $>$ operation, 2 operand
`unge_expr`	Unordered $\geq$ operation, 2 operand
`uneq_expr`	Unordered $=$ operation, 2 operand

Table 4.1: GCC tree node types – Comparison operators.

`fix_trunc_expr`	Conversion of real to fixed point – truncate
`fix_ceil_expr`	Conversion of real to fixed point – ceil
`fix_floor_expr`	Conversion of real to fixed point – floor
`fix_round_expr`	Conversion of real to fixed point – round
`float_expr`	Conversion of integer to real
`negate_expr`	Unary negation
`abs_expr`	Absolute value
`ffs_expr`	?
`bit_not_expr`	Bit wise NOT
`convert_expr`	Conversion of a type of a value
`nop_expr`	Conversion does not require code to be generated
`non_lvalue_expr`	Guaranteed not an lvalue
`view_convert_expr`	View a thing of one type as being of other type
`sizeof_expr`	C `sizeof` operation
`alignof_expr`	?

Table 4.2: GCC tree node types – Unary arithmetic operators.

`plus_expr`	Addition
`minus_expr`	Subtraction
`mult_expr`	Multiplication
`trunc_div_expr`	Integer division (quotient rounded towards zero)
`ceil_div_expr`	Integer division (quotient rounded towards $+\infty$)
`floor_div_expr`	Integer division (quotient rounded towards $-\infty$)
`round_div_expr`	Integer division (quotient rounded towards nearest int)
`trunc_mod_expr`	Remainder – truncate
`ceil_mod_expr`	Remainder – ceil
`floor_mod_expr`	Remainder – floor
`round_mod_expr`	Remainder – round
`rdiv_expr`	Division for real result
`exact_div_expr`	Division not supposed to need rounding (C pointers)
`min_expr`	Minimum
`max_expr`	Maximum
`lshift_expr`	Shift left (logical on unsigned, arithmetic on signed)
`rshift_expr`	Shift right (logical on unsigned, arithmetic on signed)
`lrotate_expr`	Rotate left
`rrotate_expr`	Rotate right
`bit_ior_expr`	Bit wise inclusive OR
`bit_xor_expr`	Bit wise exclusive OR
`bit_and_expr`	Bit wise AND
`bit_andtc_expr`	Bit wise AND ? TC ?

Table 4.3: GCC tree node types – Binary arithmetic operators.

`block`	Symbol binding Lexical Block

Table 4.4: GCC tree node types – Lexical Block.

`integer_cst`	Integer constants
`real_cst`	Real constants
`string_cst`	String constants

Table 4.5: GCC tree node types – Constants.

`function_decl`	Function declaration
`label_decl`	Label declaration
`const_decl`	Constant declaration
`type_decl`	Type declaration
`var_decl`	Variable declaration
`parm_decl`	Parameters declaration
`result_decl`	Return value declaration
`field_decl`	Structure/Union field declaration

Table 4.6: GCC tree node types – Declarations.

`compound_expr`	Compute two expressions
`modify_expr`	Assignment expression
`init_expr`	Initialization expression (2 operand)
`target_expr`	Initialization (4 operand, with cleanup)
`cond_expr`	C ternary expression (_ ? _ : _)
`bind_expr`	Local variables (See GCC source code)
`call_expr`	Function call
`with_cleanup_expr`	Specify a value to compute, & it's cleanup
`cleanup_point_expr`	Specify a cleanup point
`with_record_expr`	Provide an expression referencing a record
`truth_andif_expr`	Logical short circuited AND
`truth_orif_expr`	Logical short circuited OR
`truth_and_expr`	Logical AND
`truth_or_expr`	Logical OR
`truth_xor_expr`	Logical XOR
`truth_not_expr`	Logical NOT
`save_expr`	Flag compute-once-use-many expressions
`unsave_expr`	Permit future re-evaluations of argument
`rtl_expr`	RTL already expanded expressions
`addr_expr`	C address-of operation
`reference_expr`	Non lvalue reference or pointer
`entry_value_expr`	?
`fdesc_expr`	?
`predecrement_expr`	Node type for `--` in C
`preincrement_expr`	Node type for `++` in C
`postdecrement_expr`	Node type for `--` in C
`postincrement_expr`	Node type for `++` in C
`va_arg_expr`	Used to implement `va_arg`
`goto_subroutine`	Used internally for cleanups
`labeled_block_expr`	A labeled block
`exit_block_expr`	Exit a labeled block
`expr_with_file_location`	Annotate node with source location info
`switch_expr`	Switch expression
`exc_ptr_expr`	Exception object from the run time

Table 4.7: GCC tree node types – Statements I.

`arrow_expr`	Arrow expression ?
`expr_stmt`	An expression statement
`compound_stmt`	A brace enclosed block
`decl_stmt`	Local declaration
`if_stmt`	'`if`' statement
`for_stmt`	'`for`' statement
`while_stmt`	'`while`' statement
`do_stmt`	'`do`' statement
`return_stmt`	'`return`' statement
`break_stmt`	'`break`' statement
`continue_stmt`	'`continue`' statement
`switch_stmt`	'`switch`' statement
`goto_stmt`	'`goto`' statement
`label_stmt`	'`label`' statement
`asm_stmt`	'`asm`' (inline assembly) statement
`scope_stmt`	Mark the beginning or end of a scope
`file_stmt`	Mark where a function changes files
`case_label`	'`case`' labels
`stmt_expr`	Statement expression
`compound_literal_expr`	C99 compound literal
`cleanup_stmt`	Mark the full construction of a declaration

Table 4.8: GCC tree node types – Statements II.

`component_ref`	Node is a structure or union component
`bit_field_ref`	Reference to a group of bits
`indirect_ref`	C unary '`*`'
`array_ref`	Array indexing, single index
`array_range_ref`	Array slicing, range of indices

Table 4.9: GCC tree node types – References to storage.

`label_expr`	Label definition encapsulated as a statement
`goto_expr`	GOTO expression
`return_expr`	RETURN expression
`exit_expr`	Conditional exit from innermost loop
`loop_expr`	A loop

Table 4.10: GCC tree node types – Expressions with inherent side effects.

`void_type`	C ‘`void`’ type
`integer_type`	Integer types (includes C ‘`char`’ type)
`real_type`	‘`float`’ and ‘`double`’ in C
`enumeral_type`	C ‘`enum`’
`pointer_type`	Pointer type
`offset_type`	Pointer relative to an object
`reference_type`	Pointer automatically coerced to the type of pointed object
`method_type`	Function that takes extra ’self’ argument
`array_type`	Types of arrays
`record_type`	‘`struct`’ in C or ‘`record`’ in Pascal
`union_type`	‘`union`’ in C
`qual_union_type`	Similar to ‘`union`’ (See GCC source code)
`function_type`	Type of functions
`lang_type`	Language specific type, determined by front end

Table 4.11: GCC tree node types – Type Object code.

`error_mark`	Mark an erroneous construct
`identifier_node`	Represent a name
`tree_list`	List of tree nodes
`tree_vec`	Array of tree nodes
`placeholder_expr`	Record to be supplied later
`srcloc`	Remember source position

Table 4.12: GCC tree node types – Exceptional code.

4.3 Program Representation in AST/Generic

At t_{run} the AST/Generic representation of some sample C program is shown in Figure 4.1.

```
int f(char *a)
{
  int n = 10; int i, g;

  i = 0;
  while (i < n) {
    a[i] = g * i + 3;
    i = i + 1;
  }
  return i;
}
```

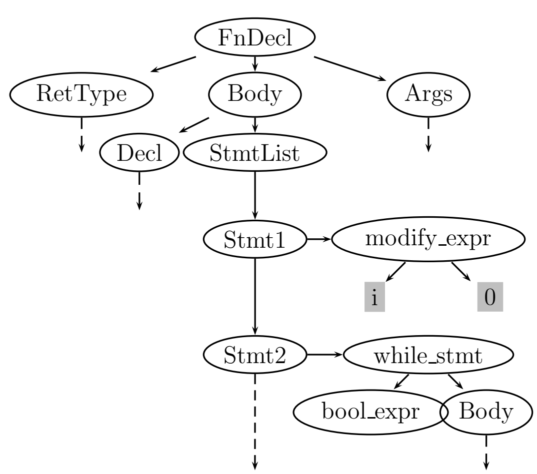

Figure 4.1: A simplified and partial AST/Generic representation of a C program.

5 Gimple Implementation

In GCC 4.0.2, the Gimple representation uses the same tree data structure as the AST/Generic. The only difference is that the AST/Generic control flow nodes listed in Table 5.1 must *not* exist in Gimple representation since Gimple lowers control flow. Following the creation of a Gimple representation, the pass manager (see Section 3.6 [Optimizations], page 9) runs a series of passes that eventually convert the Gimple representation to IR-RTL and run the IR-RTL passes. The Gimple → IR-RTL conversion is tricky to implement. With the concepts from [The Conceptual Structure of GCC], page 46 and implementation ideas of Section 3.4 [The Gimple], page 7 the details of are discussed below in Section 5.2 [Implementing the Gimple to IR-RTL Conversion], page 22.

5.1 GIMPLE Node types

`do_stmt`	`while_stmt`	`for_stmt`
`break_stmt`	`switch_stmt`	`continue_stmt`

Table 5.1: AST/Generic node types for C that are lowered during gimplification and hence cannot occur in a Gimple representation.

The AST/Generic node types listed above in Table 5.1 are lowered during the gimplification process and will *not* occur in a Gimple representation of a program being compiled. These nodes represent complex control flow constructs.

The nodes `do`, `while`, `for`, `break`, `switch`, `continue` from the AST/Generic representation are re-expressed using the `if` and `goto` statements during gimplification. Thus the Gimple node types are the same as AST/Generic node types (see Table 4.1–Table 4.12) **except** for those listed above in Table 5.1.

5.2 Implementing the Gimple → IR-RTL Conversion

5.2.1 Gimple → IR-RTL at $t_{develop}$

The Gimple → IR-RTL expander routine, `expand_expr()` in `expr.c`, contains a huge `switch-case` code. Corresponding to every Gimple node type case, the code switches to expand the standard pattern. In this way, the Gimple → IR-RTL conversion hard codes the standard names into the compiler. The IR-RTL expansion starts from the function declaration node at the top. A depth first (post order) traversal of the tree expands the child nodes (which contain operands, for example) before the root node of a given subtree. To “implement” expansion to IR-RTL of the root node, we use the following pseudo code:

```
INPUT: Gimple node type
ALGORITHM:
switch (Gimple node type) {
...
case NODE_TYPE_X: {
   get node operands, if any, from the tree structure
   use relevant information from the node, e.g. byte operation
   invoke RTX generator code for node and any sub nodes
}
```

```
case NEXT_NODE_TYPE:
...
}
```

It is possible that a given Gimple node expands to a sequence of IR-RTL expressions (RTXs). This depends on the RTX generator code, which in turn results from the specifications in the MD. If any child nodes, e.g. operands, are to be expanded, they are expanded in place, or via a recursive call to the main expander routine with the new node argument.

The "invoke RTX generator code" part of the expansion algorithm at $t_{develop}$ can only be realized at t_{build} since the actual target specific IR-RTL to use is determined at that time. Hence the idea of separating the Gimple and IR-RTL parts of the translation table is implemented at $t_{develop}$. The pattern names that can occur are enumerated in `$GCCHOME/gcc/optabs.h` and are used as indices into the array of `optab` structures defined in the same file. The contents of the array will conceptually be the occurrence of the *corresponding* pattern in the MD. The integer value of this occurrence will be generated at t_{build} by processing the MD. The program to scan the MD for occurrences of patterns and generate the indices is `gencodes.c` and is implemented at $t_{develop}$. This completes the implementation of the Gimple part of the translation table at development time. The IR-RTL part of the translation table is constructed at build time. However, the required processing of the MD is implemented at development time. The program `genoutput.c` implements this processing. Assuming that the build time processing and generation of the complete table is correct, the "invoke RTX generator code" can be implemented at $t_{develop}$ as:

```
INPUT: Gimple node type
KNOWN: pattern name corresponding to each node type
ALGORITHM:
use the pattern name to index into the ''optab'' array
get the contents at the index, which is another integer
use the integer obtained to index the ''insn_data'' array
the ''genfun'' field of the information stored in
   ''insn_data'' is the RTX generator
```

5.2.2 Gimple → IR-RTL at t_{build}

The program `genoutput.c` extracts the patterns from the MD at t_{build} and stores them in a data structure, `insn_data` array. Each pattern is stored in the sequence it occurs in the MD. As a result, the occurrence index stored corresponding to the pattern name in the `optabs` array can be used to locate the RTX generator code for the pattern that is stored in the `insn_data` array.

Figure 5.1 captures the separation of the Gimple → IR-RTL translation table at $t_{develop}$ and joining it back at t_{build} . The "movsi" pattern name is the semantic glue that connects the two separated tables at $t_{develop}$. Machine descriptions specify their own patterns for each pattern name (shown for the "movsi" pattern in the figure) at $t_{develop}$. At t_{build} the `gencodes` and `genoutput` programs operate on the selected MD and populate the `optab` and `insn_data` arrays respectively.

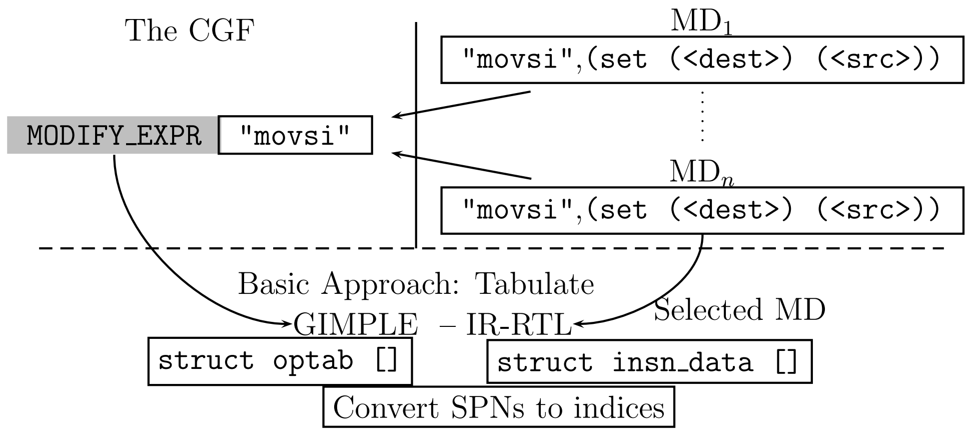

Figure 5.1: Joining the Gimple to IR-RTL translation finite function target independent LHS (`optab[]`) and target dependent RHS (`insn_data[]`) at build time, t_{build} . The contents of `insn_data[]` are from the selected machine description. Above the dashed line we have the GCC system as developed during $t_{develop}$. Below the dashed lines we have the situation at t_{build} .

5.2.3 Gimple → IR-RTL at t_{run}

Consider a concrete example of expanding a `PLUS_EXPR` (“`+`” expression) Gimple node to IR-RTL at t_{run} . Given that the expression tree in the input is located using a variable called `exp`, we extract the first operand using the macro call `TREE_OPERAND(exp, 0)` and the second operand using `TREE_OPERAND(exp, 1)` call. We need to analyze if the operands are pure constants or variables. In case they are variables, they are available either locally or globally, and either as pointers or actual variables. An activation record has been (at least conceptually) created since an expression is expected to occur within the context of some function, and the current `PLUS_EXPR` has been reached while expanding a `FUNCTION_EXPR`! Therefore, RTXs that locate the operand object are available. If the operands are not constants, the code recursively calls the main expansion routine to expand the operand node at hand. Eventually, we have an IR-RTL expansion that locates the memory area to be used for the addition operation. Skipping the details, we find that the actual RTX corresponding to the `PLUS_EXPR` Gimple node is done by a routine called `gen_rtx_PLUS()`. This expander of the “`+`” operation is *constructed* at build time from the target machine description as detailed in Section 6.4 [RTL at build time], page 33.

5.2.4 A Few Remarks About Pattern Names

Conventions have been evolved regarding the syntactic structure of pattern names. A pattern name may be an empty string “`""`” or may be a string of alphanumeric characters, or may begin with the “$\star$” character followed by an alphanumeric string. If the name is non empty and does *not* begin with the $\star$ character, then it is used during the Gimple → IR-RTL translation. The pattern name encodes two, and an optional third, pieces of information: the first substring denotes the actual operation, the second denotes the machine mode and the third optional one may be used to denote the number of operands or other purposes. Thus the “movsi” pattern name denotes the “mov” operation in “si” (Single Integer – SI) machine mode and there is no other information.

Some operations denoted by pattern names are designated as “standard” and include the machine mode. The `optab` array is actually a two dimensional array indexed using the operation part and the machine mode part of the pattern name. The “standard” operations are enumerated in `$GCCHOME/gcc/optabs.h`. The 37 “standard pattern names” are listed below. Giving one of the following names to an `insn` specification in the MD tells the IR-RTL generation pass that it can use the pattern to accomplish a certain task [GCC Internals (by Richard Stallman)], page 46.

`mov`*m*	moves data from operand 1 to operand 0, *m* is the machine mode.
`reload_in`*m*	Like `mov`*m*, but used when a *scratch* register is required to move the data from operand 0 to operand 1. Operand 2 describes the scratch register to be used.
`reload_out`*m*	Like `mov`*m*, but used when a *scratch* register is required to move the data from operand 0 to operand 1. Operand 2 describes the scratch register to be used.
`movstrict`*m*	Like `mov`*m*, but if the size of the destination of the assignment (i.e. operand 0) is *smaller*, i.e. it uses a *part* of the destination register, then this RTL instruction guarantees that the part of the register that is “outside” the destination is not altered.
`load_multiple`	Load consecutive memory locations starting from `ope-rand 1` to a set of consecutive registers starting from `operand 0`, with `operand 2` giving the number of consecutive registers – a constant.
`store_multiple`	Store to consecutive memory locations starting from `ope-rand 0` a set of consecutive registers starting from `operand 1`, with `operand 2` giving the number of consecutive registers – a constant.
`push`*m*	Output a `push` instruction. Operand 0 is the value to push.
`add`*m3*	Add operand 2 and operand 1 and store the result in operand 0; all operands of mode *m*.
`sub`*m3*	Subtract; rest similar to `add`.
`mul`*m3*	Multiply; rest similar to `add`.
`div`*m3*	Divide; rest similar to `add`.
`mod`*m3*	Modulo; rest similar to `add`.
`and`*m3*	Logical AND; rest similar to `add`.
`ior`*m3*	Logical Inclusive OR; rest similar to `add`.
`xor`*m3*	Logical Exclusive OR; rest similar to `add`.
`udiv`*m3*	unsigned Division; rest similar to `add`.
`umod`*m3*	Unsigned Modulo; rest similar to `add`.
`min`*m*3	Floating point `minimum_of` operation. If either both the ope-rands are zero or at least one of the operands is NaN, then which of them will be returned is unspecified.
`max`*m*3	Floating point `maximum_of` operation. If either both the ope-rands are zero or at least one of the operands is NaN, then which of them will be returned is unspecified.

`mulhisi3`
: Multiplication of two HI (Half Integer) mode operands, operands 1 and 2. The SI mode result is in operand 0. Since the HI (Half Integer) mode operands become SI (Single Integer) mode operands after the operation, the multiplication is characterized as *widening*.

`mulqihi3`
: Similar to `mulhisi3` except that two QI mode (Quarter Integer) mode operands yield a HI mode product.

`mulsidi3`
: Similar to `mulhisi3` except that two SI mode (Single Integer) mode operands yield a DI mode (Double Integer) product.

`umulhisi3`
: Unsigned Multiplication of two HI (Half Integer) mode ope-rands, operands 1 and 2. The SI mode result is in operand 0. Since the HI (Half Integer) mode operands become SI (Single Integer) mode operands after the operation, the multiplication is characterized as *widening*.

`umulqihi3`
: Similar to `umulhisi3` except that two QI mode (Quarter Integer) mode operands yield a HI mode product.

`umulsidi3`
: Similar to `umulhisi3` except that two SI mode (Single Integer) mode operands yield a DI mode (Double Integer) product.

`smul`*m*`3_highpart`
: Perform a signed multiplication of operands 1 and 2, which are of mode *m*, store the *most significant half* in operand 0, and discard the least significant half.

`umul`*m*`3_highpart`
: Perform an unsigned multiplication of operands 1 and 2, which are of mode *m*, store the *most significant half* in operand 0, and discard the least significant half.

`divmod`*m*`4`
: Signed division that produces the quotient **and** the remainder. Operand 1 is divided by operand 2 and the quotient is stored in operand 0 while the remainder goes in operand 3.

`udivmod`*m*`4`
: Unsigned division that produces the quotient **and** the remainder. Operand 1 is divided by operand 2 and the quotient is stored in operand 0 while the remainder goes in operand 3.

`ashl`*m*`3`
: Arithmetic Shift Left of operand 1 by the number of bits specified in operand 2 and store the result in operand 0.

`ashr`*m*`3`
: Arithmetic Shift Right of operand 1 by the number of bits specified in operand 2 and store the result in operand 0.

`lshl`*m*`3`
: Logical Shift Left of operand 1 by the number of bits specified in operand 2 and store the result in operand 0.

`lshr`*m*`3`
: Logical Shift Right of operand 1 by the number of bits specified in operand 2 and store the result in operand 0.

`rotl`*m*`3`
: Rotate Left of operand 1 by the number of bits specified in operand 2 and store the result in operand 0.

`rotr`*m*`3`
: Rotate Right of operand 1 by the number of bits specified in operand 2 and store the result in operand 0.

`neg`*m*`2`
: Negate operand 1 and store the result in operand 0.

`abs`*m*`2`
: Store the absolute value of operand 1 in operand 0.

6 RTL Implementation

RTL is used for two purposes in GCC: to specify target instruction semantics in MD at $t_{develop}$ and as an IR to represent a program being compiled. As pointed out in [The Conceptual Structure of GCC], page 46, these two uses of RTL are better described as two distinct languages: MD-RTL is a language used to specify target instruction semantics and IR-RTL is a language used to represent a program being compiled. The MD-RTL language is made up of MD constructs and (RTL) operators. The IR-RTL language is made up of IR constructs and (RTL) operators. The three objects – MD constructs, (RTL) operators and IR constructs – are together referred to as *RTL objects*. Thinking in terms of two distinct languages each suited for it's purpose helps in a more clear description of the processes that occur at $t_{develop}$, t_{build} and t_{run} . By definition, MD-RTL would be used at $t_{develop}$, and IR-RTL would be used at t_{run} ! At t_{build} , we would "generate" the IR-RTL version of the MD-RTL based specifications of the chosen target.

We first give the RTL data structure that GCC actually uses to represent any RTL object. We follow the GCC source code convention and list all the RTL objects according to their kinds and then further into their classes as defined by the GCC sources. The GCC code and documentation (see [GCC Internals (by Richard Stallman)], page 46) does **not** distinguish between MD-RTL and IR-RTL. Every RTL object is simply referred to as "RTL". As an aid to understand those documents, we have at times used the GCC terminology when the context makes it clear about which RTL language – MD-RTL or IR-RTL – is being discussed.

6.1 The RTX Data Structure

The `rtl.h` file contains the main data structure used to internally represent an RTL object. The file also contains preprocessor macros that access various fields for reading or writing values, and conditionally check the contents.

```
/* RTL expression ("rtx").  */
struct rtx_def
{
  ENUM_BITFIELD(rtx_code)     code          : 16;
  ENUM_BITFIELD(machine_mode) mode          : 8;
  unsigned int                jump          : 1;
  unsigned int                call          : 1;
  unsigned int                unchanging    : 1;
  unsigned int                volatil       : 1;
  unsigned int                in_struct     : 1;
  unsigned int                used          : 1;
  unsigned                    integrated    : 1;
  unsigned                    frame_related : 1;
  rtunion                     fld[1];
};
```

The generated file `config.h` defines `rtx` object as\\ `typedef struct rtx_def *rtx;`.

The `rtunion` is a union as below.

```
/* Common union for an element of an rtx.  */
```

```
union rtunion_def
{
  HOST_WIDE_INT              rtwint;
  int                        rtint;
  unsigned int               rtuint;
  const char                *rtstr;
  rtx                        rtx;
  rtvec                      rtvec;
  enum machine_mode          rttype;
  addr_diff_vec_flags        rt_addr_diff_vec_flags;
  struct cselib_val_struct  *rt_cselib; /* in cselib.h */
  struct bitmap_head_def    *rtbit;     /* in bitmap.h */
  tree                       rttree;
  struct basic_block_def    *bb;        /* in basic-block.h */
  mem_attrs                 *rtmem;
};
typedef union rtunion_def rtunion;
```

The `rtunion` union contains two `typedef`'d structures `addr_diff_vec_flags` and `mem_attrs` which are also defined in `rtl.h` as below:

```
typedef struct
{
  unsigned  min_align        : 8;
  unsigned  base_after_vec   : 1;
  unsigned  min_after_vec    : 1;
  unsigned  max_after_vec    : 1;
  unsigned  min_after_base   : 1;
  unsigned  max_after_base   : 1;
  unsigned  offset_unsigned  : 1;
  unsigned                   : 2;
  unsigned  scale            : 8;
} addr_diff_vec_flags;

typedef struct mem_attrs
{
  HOST_WIDE_INT   alias;
  tree            expr;
  rtx             offset;
  rtx             size;
  unsigned int    align;
} mem_attrs;
```

The `rtx` is the data structure into which the information from machine descriptions is scanned into.

6.2 Lists of all RTL objects

RTL Objects			
`const_int`	`const_double`	`const_string`	`const`
`pc`	`value`	`reg`	`scratch`
`concat`	`mem`	`label_ref`	`symbol_ref`
`cc`	`addressof`	`high`	`lo_sum`
`address`			
Comparison operators			
`ne`	`eq`	`ge`	`gt`
`le`	`lt`	`geu`	`gtu`
`leu`	`ltu`	`unordered`	`ordered`
`uneq`	`unge`	`ungt`	`unle`
`unlt`	`ltgt`		
Unary arithmetic			
`neg`	`not`	`sign_extend`	`zero_extend`
`truncate`	`float_extend`	`float_truncate`	`float`
`fix`	`unsigned_float`	`unsigned_fix`	`abs`
`sqrt`	`ffs`	`vec_duplicate`	`ss_truncate`
`us_truncate`			
Commutative binary operation			
`plus`	`mult`	`and`	`ior`
`xor`	`smin`	`smax`	`umin`
`umax`	`ss_plus`	`us_plus`	
Non-bitfield three input operation			
`if_then_else`	`vec_merge`		
Non-commutative binary operation			
`compare`	`minus`	`div`	`mod`
`udiv`	`umod`	`ashift`	`rotate`
`ashiftrt`	`lshiftrt`	`rotatert`	`vec_select`
`vec_concat`	`ss_minus`	`us_minus`	
Bit-field operation			
`sign_extract`	`zero_extract`		
Autoincrement addressing modes			
`pre_dec`	`pre_inc`	`post_dec`	`post_inc`
`pre_modify`	`post_modify`		

Table 6.1: RTL Operators I (with finer classification).

Side effects and misc.			
parallel	asm_input	asm_operands	addr_vec
addr_diff_vec	prefetch	set	use
clobber	call	return	trap_if
resx	const_vector	subreg	strict_low_part
queued	cond	range_info	range_reg
range_var	range_live	constant_p_rtx	call_placeholder
phi	nil	UnKnown	

Table 6.2: RTL Operators II (with finer classification).

insn	jump_insn	call_insn
code_label	barrier	note

Table 6.3: IR RTL types.

Pattern specification		
define_insn	define_peephole	define_split
define_insn_and_split	define_peephole	define_combine
define_expand	define_asm_attributes	define_cond_exec
Pipeline specification		
define_function_unit	define_delay	define_cpu_unit
define_query_cpu_unit	define_bypass	define_automaton
define_reservation	define_insn_ reservation	
Match specification		
match_operand	match_scratch	match_dup
match_operator	match_parallel	match_op_dup
match_par_dup	match_insn	
Attribute specification		
define_attr	attr	set_attr
set_attr_alternative	eq_attr	attr_flag
Miscellaneous		
include	expr_list	insn_list
automata_option	exclusion_set	presence_set
absence_set	cond_exec	sequence
unspec	unspec_volatile	

Table 6.4: MD RTL with finer classification.

Table 6.1 and Table 6.2 list the (RTL) operators from the set of all RTL objects listed in the `rtl.def` database. Table 6.3 lists the IR constructs and Table 6.4 lists the MD constructs. The MD constructs have further been separated according to functionality. For some RTL objects this corresponds to the class as given by the fourth argument of the `DEF_RTL_EXPR` macro.

All RTL objects have a Lisp like external syntax. We will refer to an expression made up of (RTL) operators as an *RTX*. Expressions that specify target semantics will be referred to as *MD-RTX*s.[1] Expressions that represent program (fragments) during a compilation will be referred to as *IR-RTX*s.[2] GCC refers to any RTL expression as “RTX” which we use when the context makes it clear as to which kind of expression is being referred to.

6.3 RTL at development time: Specifying MD (using MD-RTL)

Specifying target properties in GCC is extensive enough and two separate document [Writing GCC Machine Descriptions], page 46 and [Systematic Development of GCC Machine Descriptions], page 46 are fully devoted to their writing and systematic development. In this section we pose the basic problem and illustrate the technique of capturing target instruction semantics into an instruction pattern in MD. The example is demonstration oriented than being factual and continues to serve the illustrations for RTL operations at t_{build} and t_{run} .

Target CPUs for which assembly code is to be emitted vary in the number, complexity and detail semantics of instruction sets. RISC style architectures have lesser and simpler instructions than CISC style CPUs. Details can vary depending on a number of factors, some arbitrary. For instance, an architecture may insist on a certain constant value for an instruction and another architecture may insist on a quite different constant value for it’s corresponding instruction[3].

There are three main kinds of information within a MD. The first kind concerns the introduction and various manipulations of instructions, the second concerns the specification of other details of the semantics like the modes for the operands, and lastly the template of the concrete assembly code for the instruction. Other information like processor pipeline structure, instruction attributes like length may additionally be needed and specified.

Every instruction of the target that GCC may emit must be introduced to the compiler through an entry in the corresponding machine description. Additionally, a target may have more instructions that can be used to substitute an instruction for a sequence of instructions, or a expand an instruction into a sequence. The `define` family of MD constructs are used for such purposes. Target instructions may impose constraints of various kinds, size being a typical one, on the operands. MD constructs of the `match` family are used to suggest such constraints to GCC. Additional information about target properties is dependent on the target. Hence a given implementation may need an ability to define a target specific property, specify a range of it’s values and then characterize each instruction in terms of the defined attribute. MD constructs like `define_attr` and `set_attr` are used to define target instruction attributes. Finally, there are MD constructs that help implementation of the MD itself (e.g. `include` or `define_unspec`).

Armed with these MD structuring concepts, a new target machine is supported by implementing it’s MD. The basic approach is to specify instruction semantics using the RTL operators in tables Table 6.1 and Table 6.2. As described in Section 5.2 [Implementing the Gimple to IR-RTL Conversion], page 22 and illustrated in Figure 5.1, although the table columns are physically separated they are semantically connected by pattern names. A MD then uses pattern names and describes the target instruction that can implement

[1] MD-RTXs are made up of MD constructs and RTXs.

[2] IR-RTXs are made up of IR constructs and RTXs.

[3] The `trap` operation takes a constant value “5” on the i386 and a constant value “0” on the mips!

the semantics of the pattern name using the MD-RTL language. The RTL operators are used to construct the expression (RTX) that captures the semantics and the MD constructs are used to give various specification details – introduction of a new pattern, the operand matching criteria etc.

The `$GCCHOME/gcc/config/<target>/<target>.md` file implements the specification of instruction set semantics for the target "`<target>`". It is a collection of specification definitions of each target instruction that GCC supports (i.e. can emit) with optional constructs that ease the implementation.

6.3.1 Illustrative Example of RTL at $t_{develop}$:

Let a fictitious machine have a data movement operation which moves an integer argument into a register. The target syntax of the instruction is "`fictmove <source integer> <destination register>`". We use the "`movsi`" pattern to introduce this instruction to GCC via a MD-RTX as follows:

```
(define_insn "movsi"
   (set  (match_operand 0 "register_operand" "")
         (match_operand 1 "const_int_operand" ""))
   ""
   "fictmove  %1, %0"
)
```

The `define_insn` MD construct is used to introduce a new pattern to GCC. The first argument is the pattern name. It's second argument is the RTL expression (RTX) that captures the target instruction semantics. Here the RTL operator "`set`" captures the operational part of the target instruction – an assignment operation. The arguments of the "`set`" operator describe the nature of the operands. The first operand, operand 0, is the destination of the `set` and is required to be a register for this particular target. The second operand, operand 1, is the source of the `set` operation and is required to be a constant integer. Notice that the specification of the operand matching criteria are target specific, and the RTX captures the semantics of the target. Finally, the fourth argument of `define_insn` is the concrete assembly syntax of the target instruction with `%0` and `%1` being the place holders for the actual values of the operands during compilation.

The requirements of the operands that this particular target demands are specified as a match criteria using the `match_operand` MD construct. The `match_operand` expressions are the operands of the `set` operator. The third argument of `match_operand` is the name of a C boolean function that must be satisfied by the operand instance during compilation. This function implements the test. There are two functions, `register_operand ()` and `const_int_operand ()`, that are used here. The first one checks of the operand is a register and the second checks if the operand is a constant integer. Some test functions are provided by GCC, and the MD author may also write target specific ones (usually in `$GCCHOME/gcc/config/<target>/<target>.c`).

The RTX, i.e. the second operand of the `define_insn`, is written in a Lisp like form in the MD system. At build time, it is converted to C functions and data structures that would yield the internal representation at t_{run} . The `rtx` data structure in Section 6.1 [The RTX Data Structure], page 27 is used to represent RTL objects in internal form.

6.4 RTL at build time: Build Time Processing of MD

The program `gengenrtl.c` is the generator of the file `genrtl.c`. `genrtl.c` is a set of C functions that create IR-RTXs. These functions are invoked when the compiler is running to compile an input program. C preprocessor macros that are used in `genrtl.c` are found in `rtl.h`. The central data structure for RTL expressions is `struct rtx_def` in `rtl.h` that gets `typedef`'d to the pointer named `rtx`. The `code` field of this structure contains the RTX operation code. RTX emission code involves the following generic steps:

```
INPUT: RTX operation code to be instantiated

allocate compiler memory to instantiate an RTX
initialise it
set the 'code' field of the rtx struct to the input RTX code
set other fields if required
return the instantiated rtx
```

The available RTX codes are created by enumerating the RTL objects in `$GCCHOME/gcc/rtl.def`.

6.4.1 Illustrative Example of RTL at t_{build} :

The MD-RTX at $t_{develop}$

```
(define_insn "movsi"
   (set  (match_operand 0 "register_operand" "")
         (match_operand 1 "const_int_operand" ""))
   ""
   "fictmove  %1, %0"
)
```

is converted to a C function at t_{build}

```
rtx
gen_movsi  (rtx operand0, rtx operand1)
{
   ...
   emit_insn (gen_rtx_SET (VOIDmode, op0, op1));
   ...
}
```

The pattern name string "`movsi`" in the specification forms the suffix of the function name that starts with "`gen_`". The C function that implements the pattern is thus named "`gen_movsi`". Suppose this specification is the 23^{rd} MD-RTX in the MD file. Then the function `gen_movsi()` is stored as the 23^{rd} entry in the `insn_data` array. The "`gen_rtx_SET ()`" function will instantiate an instance at run time, t_{run} , of the `rtx` data structure (see Section 6.1 [The RTX Data Structure], page 27) whose "`code`" field has the integer code of the "SET" RTL operator and whose operand pointers will be set to "`op0`" and "`op1`". The operands would already be instantiated at run time since the Gimple → IR-RTL conversion would be performed by a depth first traversal at run time. The "`emit_insn`" function will chain the generated rtx into the linked list of RTXs at t_{run} that will represent the program being compiled as the RTL IR. It thus creates the IR-RTX by "embedding" the generated

RTX in suitable IR constructs. The ellipses denote the rest of the steps of the RTX emission algorithm.

6.5 RTL at t_{run} : Representing Programs (using IR-RTL)

Once the IR-RTX emission code generated at t_{build} is compiled, it can be used to emit IR-RTL representation at run time. Assuming that the input program requires a data movement operation that corresponds to the "`movsi`" pattern, we illustrate the instantiation of the RTL specification. The program in IR-RTL is a linear list of IR-RTXs. The RTL operators are used to construct the *instance* of the RTX that captures the instruction semantics in the MD at $t_{develop}$. The IR constructs usually encapsulate this particular instance with other information. Some such information is structural; for example the previous and the next IR-RTXs. Some other information is computed at various passes and propagated to later passes.

6.5.1 Illustrative Example of RTL at t_{run} :

The run time instance of the RTX in the illustrative example of sections Section 6.3 [RTL at development time], page 31 and Section 6.4 [RTL at build time], page 33 looks as:

```
(insn 24 22 25 1
  (set (reg:SI 58 [D.1283])
       (const_int 0 [0x0])
  )
  -1
  (nil)
  (nil)
)
```

The IR construct "`insn`" has many operands. The fifth operand in the above example is the RTX that is an *instance* of the RTX specified in the MD. Note that the first operand of the instantiated RTX is a register (number 58, in the example) and will satisfy the corresponding match criteria specified in the MD. Similarly the second operand is a constant integer 0 and will also satisfy the corresponding match criteria. The first three operands of the `insn` IR constrcut are mandatory. The complete syntax details may be ignored at the moment.[4]

The register number 58 in the run time instance of the RTX is a pseudoregister and the RTX as given is an *incomplete* (i.e. *non strict*) RTL. It cannot be used to generate the assembly code since the hardware register to be used is unknown! In general, an IR-RTX is *incomplete* if it lacks some information that is needed to emit the assembly code. The register allocator RTL pass would compute the actual hardware register to use for this pseudoregister. Suppose the register allocator determines that the pseudoregister 58 should correspond to hardware register named "eax" (a very i386 like name, but illustrative). Then the IR-RTX representing the program looks like:

```
(insn 24 22 25 1
  (set (reg:SI eax [D.1283])
       (const_int 0 [0x0])
```

[4] The exact syntax details of each RTL construct – MD RTL, RTL operators and IR RTL – are described in [GCC Internals (by Richard Stallman)], page 46

```
  )
  -1
  (nil)
  (nil)
)
```

The IR representation can now be converted to assembly code. The string to be used is specified in the corresponding MD-RTX (at $t_{develop}$) to be: "`fictmove %1, %0`". The value of "`(reg:SI eax ...)`" is "eax" and is used for "%0". The value of "`(const_int 0 ...)`" is "0" and is used for "%1". Hence the generated assembly instruction is:

```
fictmove 0, eax
```

7 The GCC Build System Architecture

GCC is a *generative* architecture in the sense that the build process first generates the source code of the target compiler and then builds this generated source into the target compilation system. This is a consequence of the retargetability feature of GCC. The motivations of such an architecture are discussed in [Writing GCC Machine Descriptions], page 46. Retargetability means the decision of target to be used to generate the assembly code for is decided at t_{build} . This means that at development time the target specific issues are specified for every target to be supported, and at build time a target from these set of specified targets is chosen and the information is incorporated into the compiler. Retargetabilty also facilitates generating various types of cross compilation systems [Cross Compilation and GCC], page 46.

To generate the target compiler, GCC uses a number of C programs typically prefixed by "`gen`". These programs scan the target machine descriptions (Section 6.3 [RTL at development time], page 31) and emit the data structures and code fragments that are required to obtain a complete target compilation system. The output files are collected into the build directory (see [GCC – An Introduction], page 46). Most of the source language specific parts are directly handled via suitable `Makefiles` or shell scripts. The build of the compiler is thus spread over source files generated in the build directory and the rest of the compiler in the original sources directory.

Retargetability makes it possible to create cross compilers which are a part of cross development tool chains. A cross compiler *runs* on a computer system but *generates* code for another system. In general, a cross compiler would be *built* on a *build system*, be *hosted* and *run* on a *host system*, and would *generate* code for a *target system*. These systems may have their own particular needs for proper operation; a target system may need to use it's own particular tools for correct operation of the compiler. These particulars must be known at build time t_{build} . However they must be specified on a per target basis at development time $t_{develop}$! The GCC build system must be designed so that these target specific fragments of information is collected at build time and used. Since the `make` program is used to build `gcc`, the '`Makefile`' to be used must be *composed* at t_{build} from '`Makefile`' fragments that contain such system specific information. Target specific files like '`t-TARGET`' and '`x-HOST`' contain such information, and are used by the `configure` script to create a complete '`Makefile`'.

7.1 GCC Build Overview

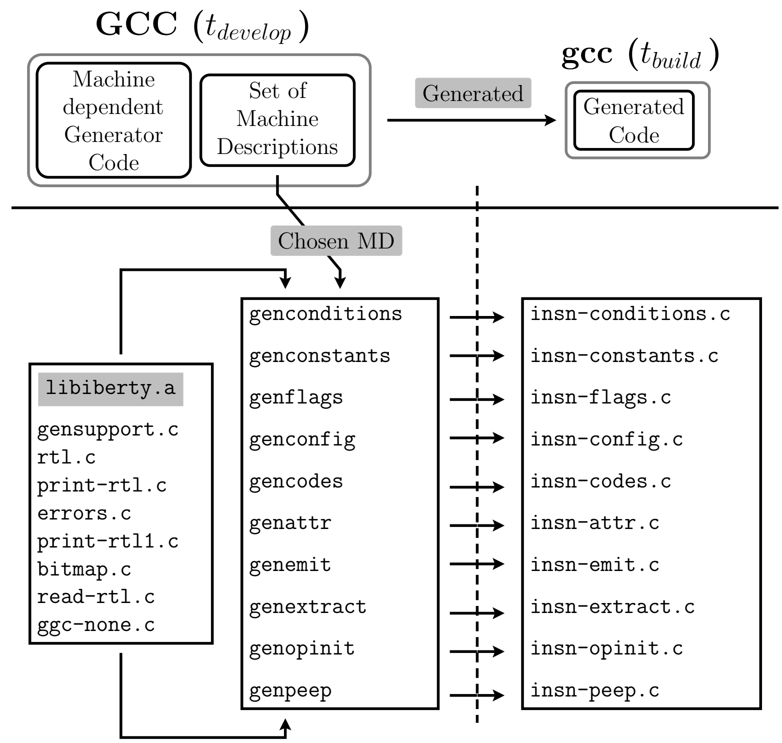

Figure 7.1: Generating target specific parts of the compiler. The solid horizontal line separates the conceptual view above from the implementation details below. The dashed vertical line separates the generators from the generated code.

Figure 7.1 details the generation part of Figure 1.1 (boxes labeled "Machine dependent Generator Code" and "Set of Machine Descriptions") of the target specific instance of the GCC sources. The figure re-orients the top-down view of Figure 1.1 to a left-right view and presents the operational details. The desired target is specified via the `configure` command (see [GCC – An Introduction], page 46). Once the desired target is known, a set of generators[1] operate on the chosen machine description and generate the target specific components of the compiler. The common functionalities like those required to read and print RTL code in machine descriptions, are compiled and archived into `libiberty.a`. Each generator uses these files and it's own main code to extract information from the target specific machine description. The figure shows a few generators and the target specific files they generate. The GCC sources at $t_{develop}$ are parametrised with "place holders" for target specific information (Figure 2.1). We emphasise that at this point we have generated a target specific version of the GCC sources which are yet to be compiled into a binary.

[1] The generator programs are obtained from the corresponding C source files at t_{build} .

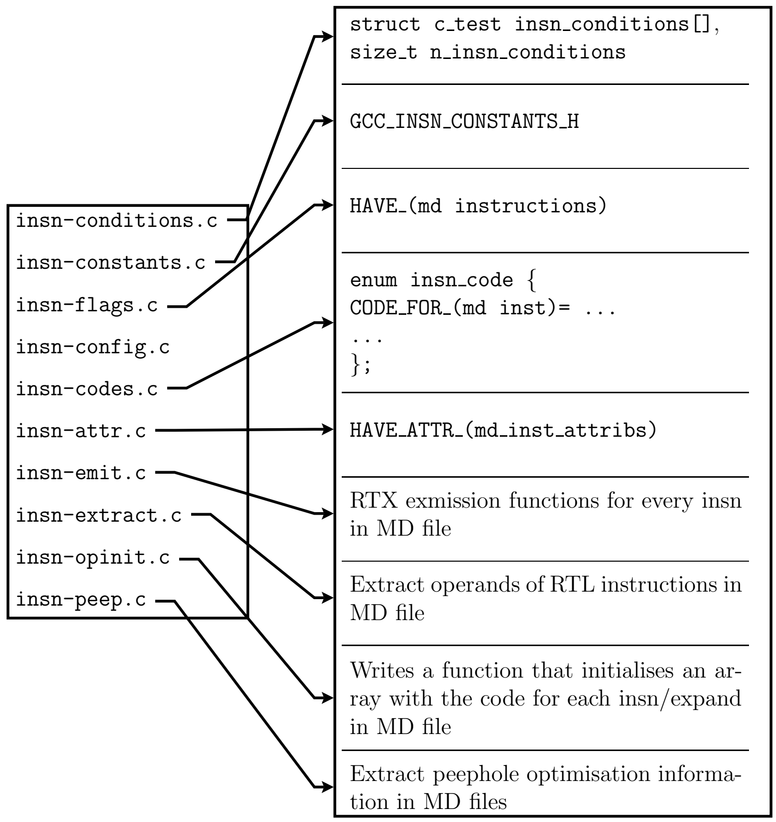

Figure 7.2: Target specific information in the generated files

The leftmost part of Figure 7.2 shows the target specific information contained in the generated files. This consists of the data structures used to represent target specific information.

The basic target specific source code generators are:

`gensupport.c`	Support routines for the various generation passes.
`genconditions.c`	Calculate constant conditions.
`genconstants.c`	Generate a series of `#define` statements, one for each constant named in a (define_constants ...) pattern.
`genflags.c`	Generate flags `HAVE_...` saying which instructions are available for this machine.
`genconfig.c`	Generate some `#define` configuration flags.
`gencodes.c`	Generate `CODE_FOR_...` macros giving the index value for each of the defined standard insn names.
`genpreds.c`	Generate prototype declarations for operand preds, and process new style predicate definitions[2]
`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 IR-RTL IR.
`genextract.c`	Generate code to operands extractor.
`genopinit.c`	Generate `optab` initializer code.
`genoutput.c`	Generate code to output assembler insns as recognized from IR-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.

The exact sequence of the generation and build of the target compiler can be obtained by looking at the sequence of the commands that are executed by a `make` on the sources after their configuration. The commands can be redirected to a file for such a study. An examination of these commands permits us to assign concpetual boundaries in the build process so that one can identify the initially the support routines are built and collected into the `libiberty.a` library, the generators are converted into process that extract the information from the machine descriptions and finally the compiler for the desired language is built. This technique captures the build for a given version of the compiler and cannot insulate us from architectural variations in future build techniques of the compiler. However, the idea is to try identifying the various logical components of a compiler build to create a foundation for understanding any future variations in the architecture.

A study of the `make` of the compiler roughly gives the following structure of compilations:

[2] This file changed in response to the introduction of new predicate definition syntax.

1.	`libiberty/`	Generates `libiberty.a`.
2.	`libiberty/testsuite/`	Nothing occurs here by default!
3.	`zlib/`	Generates `libz.a`.
4.	`fastjar/`	Generates a few JAR tools.
5.	`gcc/`	Generate some target files
6.	`gcc/intl/`	Internationalisation, if requested
7.	`gcc/`	Generate remaining target files
8.	`gcc/fixinc/`	Fix vendor include files, if needed.
9.	`gcc/`	Compiler compilation

The `libiberty` library contains general, multipurpose routines which are used in the other programs that build the final compiler. The common tasks handled are regular expressions manipulations, reading and printing RTLs, error handling and garbage collection used internally by GCC during operation.

The “`gen*.c`” files are compiled using the native compiler on the build system. This binary operates on the machine description, if necessary, to obtain the corresponding target compiler component source code. Once the target specific parts are generated, the build process continues to build the actual compiler. This is done in two phases. First, the front end independent parts are compiled and archived into the `libbackend.a` library. This is common to every front end, and multiple front ends may be requested by the user. The build system then builds a *separate* compiler for each desired front end.

A compiler for a given language, say C, is built using the front end processor files in the respective directories, a few common routines from `libiberty.a` and the backend library `libbackend.a`.

Source File	**Use**
c-parse.c	A bison parser made from c-parse.y
c-lang.c	Language Specific Hook implementations for C
c-pretty-print.c	Common C/C++ pretty printing routines
attribs.c	Functions dealing with attribute handling
c-errors.c	Various diagnostic routines
c-lex.c	Mainly the interface between cpplib and the C front ends
c-pragma.c	Handle `#pragma` SVR4 style
c-decl.c	Processes declarations and variables for C
c-typeck.c	Build expresions with type checking for C
c-convert.c	Language level data type conversion for C
c-aux-info.c	Generate information regarding function declarations and definitions based on information stored in GCC's tree structure
c-common.c	Routines common to all C variants
c-opts.c	C/C++/ObjC command line options processing
c-format.c	Check calls to formatted I/O functions (?)
c-semantics.c	Definitions and documentation for the common tree codes
c-objc-common.c	Some common code to C and ObjC front ends
c-dump.c	Tree dumping functionality for the C family
libcpp.a	
main.c	Defines `main()` for `cc1`, `cc1plus` etc.
libbackend.a	GNU CC internal collection of backend code
libiberty.a	GNU CC internal collection of useful routines

Table 7.1: Compiler files and their contents for `cc1`

For C, the files used are shown in table Table 7.1 and are used to generate the compiler `cc1`.

Finally, the compiler driver `gcc` can be, and is built (Section 7.2 [The Compilation Driver – gcc], page 41). This is actually built as `xgcc` to avoid possible name clash if `gcc` is available on the build system. This insulation from the possible availability of a `gcc` command is required during the boot strapping phases. For a native compiler, the build is performed in at least three boot strap stages. In the first stages the native `gcc` or the vendor C compiler is used to generate the compiler. The compiler `xgcc` generated in this stage is then used to build the complete compiler again in stage 2. To check the success of the second stage build, the `xgcc` from stage 2 is used to build the compiler again into stage 3. It is then expected that stages 2 and 3 give identical results. GCC, therefore, always builds the driver as `xgcc` and renames it as `gcc` during compiler *installation* time. This driver can be built once the compiler proper, namely `cc1` is built.

7.2 The `gcc` Compilation Driver

Conceptually, when a user requests the compilation of a file, say `myprog.c`, the system does the following:

- The shell forks (and execs) the GNU Compiler driver `gcc`.

- `gcc` "studies" the command line as discussed below. In particular, `gcc` sets up the commands with suitable options and invokes the desired compiler, assembler and linker in sequence.
- The compiler proper – `cc1` for C sources – compiles the given input, a single file, into an equivalent target assembly code.
- The assembler proper – `as` for assembler sources emitted by the compiler `cc1` – assembles, i.e. "compiles" the assembly source into object code.
- The linker – `ld` – is given the objects compiled along with suitable libraries to link into executable code.

The structure of the `gcc` driver code is as follows:

- Setup the program name.
- Do initialisation for internationalisation.
- Install signal handlers.
- Build multilib selection /* Libraries compiled multiple times */
- Setup options for `collect`.
- Setup machine specific environment variables.
- Make a table of what switches there are (switches, n_switches). Make a table of specified input files (infiles, n_infiles). Decode switches that are handled locally.
- Process driver self spec (?).
- The `default_compilers` array contains the command line specifications for invoking the compiler to be invoked based on the extension in the given input source.
- Read the `specs` file[3], if any, else fall to default.
- Now locate the required executables, i.e. the pre processor, the compiler, the assembler and the linker. Native system compilers have a standard location. Any standard libraries, e.g. `libc` for C programs, are added around this time.
- Locate the other support files, e.g. the startup and end code.
- Switches and specs done. Now set up the subdirectory based options.
- Unrecognised options, if any, are now responded to.
- Print out any user requested details of the information found until now.
- Bail out if no input file is given!
- Setup output file names. In particular, it **appears** that the file names of the entire tools chain are created here and setup in the array of input files. It is over this array that the next step operates. As a result of the lookup phase, the "compiler" that is found, is actually the binary that *transforms* the input file to the desired output. Thus a ".c" file has the `cc1` binary as the "compiler" that emits the ".s" assembly. This ".s" is also a part of the input files array. Hence in the next iteration, the lookup phase finds `as` as the "compiler" that emits a ".o" file. A ".o" file is *not* a part of the set of extensions recognised by the lookup phase and hence by default is passed on to the linker!
- Now start processing each input file:
  - Look up the compiler for the input file,

[3] The syntactic details of the `spec` file are detailed in the header comment of the `gcc.c` file.

- Find it's `spec` (assuming the compiler is found),
- If compiler not found, *assume* the input to be file for explicit linking (e.g. `.o` file),
- On errors, delete the delete-on-failure queue. If compilation successful, delete temporaries.
- We now have a set of files ready for linking. The linker is either `collect2` or `ld`. See `info gcc` for the similarities and differences between `collect2` and `ld`.
- Run the linkage processing phase.
- Delete temporaries. Cleanup based on any errors encountered or as specified on the command line by the user.

The central idea of the `gcc` driver architecture is a table driven approach to looking up the "compiler" binary based on the input file name extension and an associated standard command line which can be augmented with user specified command line. The architecture simply creates a sequence of intermediate file names that are the output of the current stage and then input of the next stage, and iterates through them. For each input file, the "compiler" is looked up, the actual command line created (this involves some parsing and instantiation of the corresponding specification from the `specs` file), and then a `fork ()` is issued. At the point of `fork ()`, the activation stack looks like (the stack grows upwards in the figure below):

1. `fork()`
2. `pexecute("cc1 path", "parent proc (gcc)", "input file", ...)`
3. `execute()`
4. a sequence of calls to `do_spec_1()` which ultimately instantiate the specification from the `specs` file
5. `do_spec()`: A point that is reached when the driver has found all the necessary information to initiate the execution sequence. That is the driver has found that an input file exists, it's "compiler" exists and the "standard" command line for the "compilation" exists
6. `main()`

8 Conclusion

We have described some of the implementation details of GCC 4.0.2 with the conceptual background of [The Conceptual Structure of GCC], page 46. The focus has been the implementation of the compilation concepts and not on the work required to extend it to being an industry strength compiler. Details like the implementation of standards adherence, error detection and reporting etc. have been omitted in this work. However, these details are necessary to help understanding the GCC source code. Some details were voluminous enough to merit a separate document. For instance the development of a machine description is separated and can be found in [Systematic Development of GCC Machine Descriptions], page 46. The syntactic details of almost all the concepts discussed can be found in [GCC Internals (by Richard Stallman)], page 46 and we include only the necessary parts (see, for example, Section 6.5 [RTL at run time], page 34).

The RTL is an interesting feature of GCC. It is a language that can capture the semantics of target instructions as well as represent the program internally during compilation. The use of RTL as an IR can be viewed as a "abstract syntax representation" of target assembly language; i.e. the concrete assembly syntax has been discarded and only the semantics captured. The (implicit, unstated) rule in GCC RTL phase is to ensure that the RTL IR is complete enough so that every RTX in the IR (ideally) maps to a unique assembly string. This suggests an attempt to perform compilation as much independent of the target syntax but with target as much target semantics as possible. This enables some traditionally target specific techniques like peephole optimization to be generically implemented.

8.1 Future Work

A number of possibilities exist for future work. The conceptual directions are already explored in [The Conceptual Structure of GCC], page 46. As far as implementation needs go, the official GCC site (`http://gcc.gnu.org/projects`) lists a number of projects that may be pursued, better documentation being one of them. Here we present our own additions/changes to that list.

- The present description of the internals of GCC need to be augmented with descriptions about issues that have been left out. A partial list is:
  - Regression testing
  - List of standards complied to
  - Standards implementation and compliance testing methods
  - Front end architecture[1]
  - Support libraries: concepts and implementation. Emulation libraries implement functionality that might not be available on a target, for example software floating point emulation. Some of these are part of `libgcc.a`. Compression libraries, HLL standard libraries (e.g. Java, but not C since the C standard library implementations – `glibc` or `newlib` – are separate GNU packages).
  - `autoconf` and `automake` details of the configuration and build process in GCC.
  - Garbage collection: concepts and implementation as *used* in GCC.

[1] Some descriptions are available on the Internet.

- The GCC machine description system can be improved. The current parameterisation is implemented using C preprocessor macros and RTL based target instruction semantics system. The conceptual components of machine descriptions are not well separated at the present. Given the current technology emphasis on embedded systems, mobile computing, DSP and SoC the processor architectures are changing fast and often include domain specific instructions as well as instruction level parallelism and complex addressing modes. The GCC machine description technology may need to be enhanced to support such systems.
- Improving the abstract machines to open up formal verification efforts of the architecture and implementation. This is a gargantuan task given the size and scale of the implementation.

References

(**Note:** In the URLs below: `$GCCINTDOCSHOME` is `http://www.cfdvs.iitb.ac.in/~amv/gcc-int-docs`)

1. Richard. M. Stallman.
   GCC Internals.
   (`http://gcc.gnu.org/onlinedocs/gccint`)
   2007.
2. Abhijat Vichare.
   GCC – An Introduction.
   (`$GCCINTDOCSHOME/html/gcc-basic-info.html`)
   2007.
3. Abhijat Vichare.
   Cross Compilation and GCC.
   (`$GCCINTDOCSHOME/html/gcc-cross-compilation.html`)
   2007.
4. Abhijat Vichare.
   Writing GCC Machine Descriptions.
   (`$GCCINTDOCSHOME/html/gcc-writing-md.html`)
   2007.
5. Abhijat Vichare.
   The Conceptual Structure of GCC.
   (`$GCCINTDOCSHOME/html/gcc-conceptual-structure.html`)
   2007.
6. Abhijat Vichare.
   The Phasewise File Groups of GCC.
   (`$GCCINTDOCSHOME/html/gcc-source-blocks.html`)
   2007.
7. Uday Khedker and Sameera Deshpande.
   Systematic Development of GCC Machine Descriptions.
   (`http://www.cse.iitb.ac.in/~uday/soft-copies/incrementalMD.pdf`)
   2007.

List of Figures

List of Tables

Appendix A Copyright

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

A.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this

License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ''GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.