The Conceptual Structure of GCC

GCC Version 4.0.2

Abbhijat Vichare (`amvichare@iitb.ac.in`)

Indian Institute of Technology, Bombay

(`http://www.iitb.ac.in`)

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

Short Contents

Table of Contents

1 Introduction

The GNU Compiler Collection (GCC) is a standard system compiler for GNU systems, including GNU/Linux. It is often an alternate compiler on many platforms due to it's wide spread availability. The FSF model of development has reduced the costs and GCC is also a strong candidate for new systems, like embedded systems, that require a software development system. However, porting GCC to new platforms is an involved task. And then, there is no suitable description of the internals that can serve porting endeavors. The GCC internals document found on the GCC site (see [GCC Internals (by Richard Stallman)], page 34), has more of a "reference" flavor than a conceptual description of the GCC architecture. Most porting exercises rely on descriptions of personal experiences in GCC porting (see [Porting GCC for Dunces (by Hans-Peter Nilsson)], page 34), or as parts of other works (see [GCC Internals (by Deigo Novillo)], page 34 and [GCC – Yesterday Today and Tomorrow (by Deigo Novillo)], page 34). An attempt has also been made (see [GCC Wiki Book], page 34) to describe the internals more completely. It is a collective effort but focuses on the implementation details. The lack of a description of the internals at a sufficiently abstract level makes porting attempts an expensive affair unless the people involved are well versed with compiler internals. The availability of such a description is useful for general study of compilation techniques, and in particular, can help GCC become a system of choice for researchers in languages and compilation, especially those who wish to deal with real life engineering issues.

This document views GCC version 4.0.2 as a chain of lowering operations in terms of the "abstraction lowering" phases that are required to lower a C like HLL to a typical 32 bit target. We show how these lowering operations connect with the implementation, and thus serve to conceptually understand the otherwise complex implementation. In particular, we demonstrate the utility and implementation strategies for the GCC machine description system that is a consequence of the retargetability requirements of GCC.

The document contributes to bridging the gap between the known concepts and a complex implementation. In particular, we hope that the identification of the sequence of lowering operations encourages a more formal effort in GCC internals. The work is organized as follows. First, we identify the basic abstraction mechanisms in programming languages. These are used to motivate an intuitive concept of an abstraction gap between the source language and target language of a compiler. Having identified this gap, this section ends with an empirical prescription to obtain the phase sequence of a compiler for lowering the HLL abstractions. Such a phase sequence of operations would be followed by a compiler in operation and the goal of design and implementation is to specify the desired phase sequence. With the backdrop of these abstraction mechanisms and the need for retargetability, we identify three parts of the GCC system: the compiler generation framework architecture, the build system and the compiler operation architecture. The views are cleanly separated from each other by identifying the time durations of work for each. In fact, these time durations are used throughout all our documents and serve as "hooks" from which the different views of GCC may be obtained. For the compiler operation view we identify the IRs of GCC. One of the IR languages, the RTL, is also used in at development time to specify machine descriptions, and is the connecting link between the development time and compiler operation time of GCC. The identification of these three parts of the GCC system, the architecture of the compiler generation framework, the build system and the compiler operation architecture, are the central contributions of this work. We mention the front end processing IR for completeness but our main focus is the rest of the system. The C language is used as an illustrative HLL; GCC was born as a C compiler. We also ignore optimisations that GCC performs, partly because they are large in number and mainly because they do not contribute to the translation process. We conclude with a note on future directions.

In short, we focus on the compilation phase sequence and the backend architecture using C as an illustrative HLL. The following are a few issues that we ignore:

1. front end architecture,
2. standards compliance,
3. optimisations, and
4. supporting code like memory management and error detection and reporting.

2 Language Abstractions

There are four main kinds of abstractions that programming languages use: data abstraction, control flow abstraction, procedural abstraction and name space abstractions.

Data abstractions involve identification of useful data types and their definitions in terms of available ones. Languages support data abstraction by providing a set of data types, and mechanisms to compose newer ones from the given ones, or the ones already composed. Given a pair of programming languages it is possible to intuitively grasp their respective levels of data abstraction. A typical machine language fares poorly in relation to an HLL like C. C provides richer data types and data structuring mechanisms than typical machine languages.

Control flow abstractions, minimally, are sequencing and unrestricted branching. Higher order flow constructs include loops, function calls (in the sense of current state preservation, loading new state, passing control to it, and returning back to previous state through state restoration), exception handling, coroutines and continuations. Most higher order control flow constructs are not provided by typical machine languages.

Procedural abstractions isolate computations from the state. Having named the computations, they can be reused by supplying the state under which the evaluation is to be performed. Function calls are procedural abstractions in that named computations are invoked for various values. Higher order functions, nested functions etc. are procedural abstractions.

Name space abstractions emerge because computation objects are named. However, since there are a large number of objects, program specifications are under a "naming pressure" to create unique names that can be bound to unique addresses on target hardware. Naming pressure is alleviated by defining the lifetime and access rules for names of the objects. Scope rules, name privacy, mangling techniques are examples of name space abstractions that serve to manage the name pressure.

2.1 The Abstraction Gap

Given the various abstractions in programming language design, it is possible to intuitively grasp the abstraction gap between a pair of languages. For the purposes of compilation, we consider a $\langle$ HLL, machine language $\rangle$ pair. For concreteness, we illustrate using C as the HLL and the i386 as the machine language.

For C, we have

- Data Abstraction: Character, unsigned integers and string data types.
- Control Abstraction: If-then-else branching, finite and infinite iteration loops and function calls.
- Procedural Abstraction: Function calls.
- Name Space Abstraction: statically scoped variables, file scoping.

For the i386, we have

- Data Abstraction: Unsigned integers between 0 and a `MAX_INT`.
- Control Abstraction: Auto-incrementing program counter to support sequencing, and an unrestricted branch.

- Procedural Abstraction: None (the `call` instruction does not perform a true procedure call).
- Name Space Abstraction: None.

Despite a partial listing, the enumeration of the abstractions clearly points out to an abstraction gap. A compiler is required to bridge this abstraction gap. We suggest an empirical rule for bridging this abstraction gap: handle largest gap first. Thus the compiler should start bridging either the procedural abstraction or the name space abstraction first. Note that the suggestion does not imply completing the handling of a given abstraction before starting to deal with the next. Also, since the gap is intuitive in nature, we do not have a unique lowering sequence.

The phase sequence of a compiler is the sequence of lowering these abstractions from the HLL to the target. The GCC lowering sequence (for C) is: procedural $\rightarrow$ name space $\rightarrow$ control flow $\rightarrow$ data.

3 The GCC System

This section describes the main ideas of the three views that are useful to understand the GCC system at three distinct time periods described in Section 3.1 [The Impact of Retargetability], page 5 and depicted in Figure 3.1. Figure 3.2 succinctly captures the structure of the entire GCC system. It's components are described in the various subsections of this section. The figure can be vertically divided into three parts each corresponding to the various time periods. The top box of the figure is described in Section 3.2 [The GCC Compiler Generation Architecture], page 6. The middle part is described in Section 3.4 [The GCC Build System Architecture], page 9 and the bottom part is in Section 3.3 [The GCC Compiler Architecture], page 8. Finally, Section 3.5 [The GCC IRs], page 10 describes the concepts that go in realising these ideas using the intermediate representations.

3.1 The Impact of Retargetability

From the GCC project point of view retargetability is a desirable feature for the project to be useful to a broad community of users. GCC is also resourcable.[1] Retargetability implies that the target specific information is not available at implementation time and must be incorporated at a later point of time, namely the *build time*, denoted by t_{build} . The build time is a critical time period in GCC. It separates the GCC system into two different views on it's either sides as shown below in Figure 3.1.

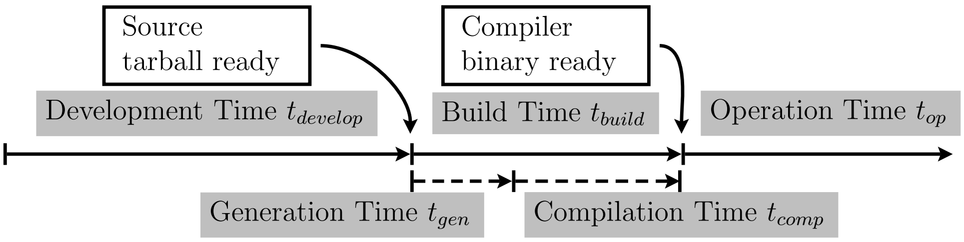

Figure 3.1: Important time durations in GCC. $t_{develop}$ is the time during which a given version of GCC is being developed. t_{build} is the time during which GCC is built, and t_{run} is the time when the binary is used by programmers to compile their programs. t_{build} may be further sharpened into t_{gen} and t_{comp} . During t_{gen} the GCC sources are processed to generate the target specific parts of the (retargetable) compiler system created at $t_{develop}$. During t_{comp} the compilation of GCC starts to yield a binary.

Before t_{build} , the compiler is being developed and the architecture described is used by the GCC developer. Implementing retargetability in GCC at development time, denoted by $t_{develop}$, results in three logically distinct parts of the compiler namely

1. the compiler code parameterized with respect to retargetability issues,
2. a set of per target specifications of the parameter "values" listed for each supported target, and

[1] Resourcability implies being able to use the compiler to compile some other HLL. This term is not very common, but we use it for precision.

3. the "generator" code that will be used to incorporate the target information selected at t_{build} into the parameterized compiler code.

At the end of $t_{develop}$, the compiler generation framework is ready in source form and is distributed as versioned tarball from the official GCC website. When the compiler is built from the sources at t_{build} the specifications of the chosen target are incorporated into the parameterised code resulting in a compiler source that would yield a compiler for the chosen target. Thus, at $t_{develop}$, GCC exists as a parameterised part and set of target specifications. The conceptual view that is useful to understand GCC at $t_{develop}$ is described in Section 3.2 [The GCC Compiler Generation Architecture], page 6.

At the end of t_{build} we have a complete target specific compiler program that programmers can use to compile their programs. These times after t_{build} when the binary is used will be denoted by t_{run} . The architectural view that is useful to understand the operation of the compiler at t_{run} is described in Section 3.3 [The GCC Compiler Architecture], page 8. Occasionally it may be necessary to distinguish the system at t_{build} into the time when the target specific parts of the compiler are being generated, denoted by t_{gen} and the subsequent compilation time, denoted by t_{comp} , where the final compiler binary is created. The views at $t_{develop}$ and t_{run} come together at build time t_{build} . The compiler generation framework implemented during $t_{develop}$ is used to obtain the target specific compiler (operation) phase sequence at t_{run} at this time. The operations that are required to occur at t_{build} forms the build system described in Section 3.4 [The GCC Build System Architecture], page 9. Figure 3.1 shows these time period labels.

3.2 The GCC Compiler Generation Architecture

Prior to t_{build} the GCC system is the system that a GCC developer actually works with. The code base is made up of the parameterised compiler and the set of target specifications of the parameters. We refer to this as the GCC Compiler Generation Framework, or CGF for short, or simply GCC[2]. The top half of Figure 3.2 shows the major components of this part.

[2] Note the upper case.

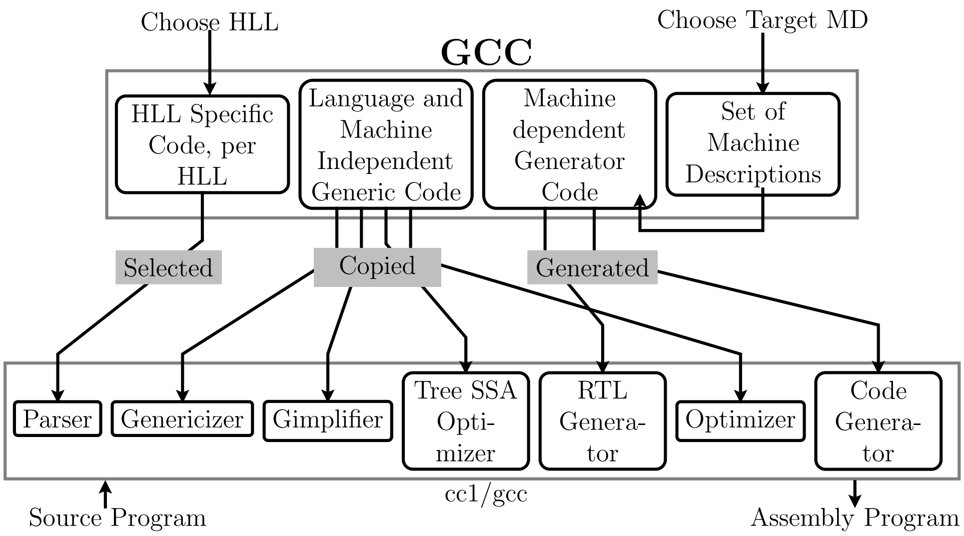

Figure 3.2: The GCC Compiler Generation Framework (CGF) and it's use to generate the target specific compiler (cc1/gcc) components. Some components of the compiler (cc1/gcc) are *selected* from the CGF, some are *copied* from the CGF and some are *generated* from the framework.

Since a number of different source languages are supported by GCC, the parsers and generisers for each supported front end form the details of the "language specific code" component in this block. At build time the processing code of the chosen front end language is *selected* from amongst the available ones and included. Thus the front end developer view of the GCC system is just addition of the processing code which finally produces the AST/Generic representation.

The CGF also contains code that is independent of the characteristics of the target system or the front end language. For instance, the gimplifier converts the generic representation into the gimple IR (described in Chapter 4 [The Gimple IR], page 12). The gimple IR is independent of the target properties too, and many machine independent optimisations that can be performed on this IR are implemented in the CGF. Such code is simply *copied*[3] into the compiler created at build time and is shown as the second box in the "GCC" block of Figure 3.2. The RTL expresses the Gimple code in a target specific manner. However, the code that works on the RTL IR can be generically expressed and is a component of this box. This part forms the bulk of the compiler code.

Perhaps the most interesting part of the CGF is the code that is dependent on target properties, but has been "parameterized". For instance, the target code generator of the compiler in operation needs to emit the target assembly code in the syntax of the target assembly language! Thus details like the layout structure of the assembly program must be known. Parameterization is a consequence of retargetability and the CGF can then

- *name* the various target properties in the main compiler code, and

[3] The code is only *conceptually* copied. In practice, this code is simple picked up from the `$GCCHOME` (see [GCC – An Introduction], page 34) rather than a physically copying it into the `$BUILDDIR`.

- separately *list* out their target specific values for each target.

This is the main emphasis at development time. The parameter values are separately listed at development time for each supported target. Most parameter values can be captured through simple C preprocessor expressions. However some "values" like expressing target instruction semantics need a better mechanism than that provided by the C preprocessing system. But employing such a different mechanism invites developing a processing system to incorporate the information into compiler! Simply put, the target machine descriptions must be processed at build time to *generate* the target specific parts of the compiler. This processing system must be developed at development time. This is indicated in the third box.

The set of machine descriptions that GCC supports, i.e. for which the values of the parameterised entities of the compiler have been specified, form the contents of the fourth box. GCC uses the RTL language to capture the target instruction semantics. The RTL also serves as the language for IR at operation time t_{run} . Using RTL for both these purposes makes it easy since the data structures and operations are the same for both purposes. At t_{build} one of the targets from this set is chosen and presented to the processing system to generate the target specific parts of the compiler.

3.3 The GCC Compiler Architecture

Once the target specific parts of the compiler are generated from the CGF, we have a complete compiler source code for the chosen front end HLL and the chosen target machine. The structure and operation of this compiler, which we refer to as "gcc"[4] in Figure 3.2, follows the approach outlined in Chapter 2 [Language Abstractions], page 3, especially the sequence pointed out at the end of Section 2.1 [Abstraction Gap], page 3. Figure 3.3 shows the relationship between the lowering of abstractions and the phase sequence of the "gcc" compiler. Additionally, it also shows the various optimisation opportunities that open up at each phase. We have ignored the processing of the front end and hence the parsing phase that starts lowering the procedure (functions in C) is not shown[5].

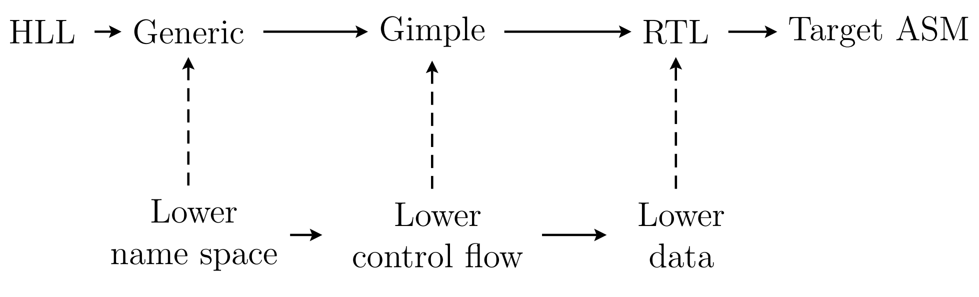

Figure 3.3: The relationship between the GCC phase sequence and sequence of language abstract lowerings done one at a time.

[4] Note the lower case.

[5] Recent versions of GCC can also start lowering a file of C functions rather than just one function as indicated here.

The RTL is the connecting link between the CGF view and the operation view of the compiler. In the CGF it is used to *specify* target instruction semantics. In operation, it is used to *express* computations in a target specific manner. We observe that the RTL is the last IR in the phase sequence. The gcc compiler attempts to perform as many target independent operations as possible before becoming target specific[6]. The input program is then represented in RTL form that contains target specific information. In particular, the input data representations, i.e. data structures and data objects, are lowered. The implicit goal in this phase is to perform all machine dependent operations in a *generic* manner so that every RTL computation construct maps to at least one target syntax. One, therefore, expects that at the end of this phase all that remains is converting the RTL representation to target assembly syntax.

The phase sequence of `gcc` is captured in the bottom part of Figure 3.2. We detail out the conceptual structure of the Gimple and RTL IRs in Section 3.5 [The GCC IRs], page 10.

3.4 The GCC Build System Architecture

The CGF at $t_{develop}$ in the top half of Figure 3.2, needs to be transformed into the compiler architecture that is used at t_{run} and shown in the bottom half of Figure 3.2. As discussed in Section 3.1 [The Impact of Retargetability], page 5 the two views differ. At $t_{develop}$ there is no notion of which target the compiler is to generate output for. Once the target is chosen at t_{build} these target specific parts need to be generated from the specifications at $t_{develop}$. In this section we examine the architecture of this part of GCC.

The above two sections each point to the need of using languages for their specific purposes. The CGF requires a language to specify the semantics of the target instructions. The compiler that runs requires a language to represent a compilation internally. It is best to consider that conceptually these are two separate languages. We refer to the language used to specify target instructions as the *"Machine Description RTL"* or MD-RTL for short. The language used to represent a program being compiled by the compiler will be referred to as the *"Intermediate Representation RTL"* or IR-RTL for short. At t_{build}, then, we need to convert the MD-RTL based information at $t_{develop}$ into data structures and operations that would yield IR-RTL at t_{run}. The MD-RTL based information is shown in the box labeled "Set of Machine Descriptions" (fourth box) in the top half of Figure 3.2. The box labeled "RTL Generator" (fifth box) in the bottom half of Figure 3.2 stands for the internal representation in terms of IR-RTL. The arrows labeled "Generated" denote the conversion at t_{build} from MD-RTL to IR-RTL.

The implementation of the conversion is eased by choosing some part of the two languages, MD-RTL and IR-RTL, to be common. This part is used to specify (at $t_{develop}$) or represent (at t_{run}) the semantics of the target instructions. The current[7] GCC documentation does not distinguish between these two separate languages. Instead both the languages are simply referred to as "RTL" often leading to confusion. Chapter 5 [The RTL], page 19 describes these two languages in detail.

---

[6] This was not cleanly evident in earlier versions which did not use the Gimple IR. After the parser lowered the input to an AST representation it was directly converted to RTL representation until GCC version 3. The target independent operations were mostly performed on the RTL representation resulting in a complex code. Dealing with this code base required a knowledge of what parts of the code were machine dependent and what parts were not – something that is not *a priori* evident.

[7] As of 2007.

At t_{build} , the MD-RTL based specifications are *selected* according to the chosen target and transformed to *generate* the IR-RTL based target specific parts of the final compiler. It is because of the generation of target specific source code from the target properties specifications at build time that we refer to GCC prior to t_{build} as a Compiler Generation Framework.

Thus two crucial steps must be taken at t_{build} given the chosen target:

1. The parameterised information in the header files of the chosen target must be included into the source tree, and
2. the MD-RTL based target specifications must be processed to obtain a IR-RTL based representation system.

On completing these steps we have a compiler source code that has front end processing code for the "selected" HLL, the "copied" main body of the compiler and the "generated" target specific code for the chosen target as shown in Figure 3.2. This source is then compiled to obtain the compiler binary.

3.5 The GCC IRs

GCC 4.0.2 uses three IRs: The AST/Generic, the Gimple and the IR-RTL. The AST is the output of the front end parser and depends on the particular language features. Since the parser starts the lowering process the moment a function is found in the input, we say that GCC handles procedural abstractions first. Generic is intended as the common IR to which front end ASTs are reduced to. However the AST and Generic have not been well separated as yet, and we consider them as a single IR called AST/Generic. At this level, GCC starts handling the name space abstractions of C. The local scope variables are marked so that the later phases can lower them. The Gimple is a front end and back end independent IR that lowers the control flow.

The most interesting IR in GCC is the IR-RTL – a syntactically Lisp like IR. GCC attempts to capture all target specific information in the IR-RTL representation. This is required since the final lowering to target specific code requires a knowledge of the target properties. IR-RTL lowers the data abstractions, and the target specific lowering of the other abstractions. In particular, it unfolds the data structures into bits, bytes, words etc. sequences and creates the activation record[8]. Lifetime rules of local variables are handled by placing them on the activation record under a stack discipline. The activation record also handles the implementation of the procedural abstraction.

An implicit goal of the IR-RTL IR is to do all target specific manipulations as completely as possible so that at the end of the IR-RTL passes, the program representation contains all the information of the assembly language version except for the syntactic details.

Thus the overall gcc phase sequence shown in Figure 3.3 can be written as: (parser : procedural abstraction) → (Generic : name space abstraction) → (Gimple : control flow abstraction) → (IR-RTL : data abstraction), where "(parser : procedural abstraction)" reads as "parser deals with procedural abstraction", and "→" reads as "lowers to". Each phase deals with lowering a particular HLL abstraction and has it's own IR designed for

[8] In earlier versions of GCC, the activation record used to be constructed during the ASM emission time. The trend is to have the IR-RTL IR do as much target specific work as possible and in recent versions, the responsibility of creating the activation record is with the RTL subsystem.

that purpose.[9] The compilation process occursduring the chaining of these phases where a transition is made from the representation in a given phase to the representation in the next phase. Each transition can be realized as a translation table.

The next section details these ideas for the Gimple IR. The IR-RTL is used to construct the IR at t_{run} and is detailed as a part of the RTL in section Chapter 5 [The RTL], page 19.

[9] The introduction of Gimple tuples in GCC 4.3 branch actually aggrandises this view that the IR is designed for the purpose. As of GCC 4.0.2 the Gimple IR is conceptually distinct from the AST/Generic, but uses the same data structures as the AST/Generic.

4 The Gimple IR

Section Section 3.5 [The GCC IRs], page 10 mentioned the IRs that GCC uses and indicated that the compilation is effected by translating the representation in one IR to the next through translation tables for each. In this section, we expand the concept for Gimple. Ignoring the front end, gcc will have a Generic to Gimple translation table and a Gimple to IR-RTL translation table. Note that the former is a target independent table and the latter is a target dependent table. This implies that the former table can be expressed in the generic part of GCC at $t_{develop}$ but the latter table cannot be. For the Gimple to IR-RTL translation table, the Gimple part is known at $t_{develop}$ but the IR-RTL part is not known at $t_{develop}$. At $t_{develop}$ the Gimple to IR-RTL table is incomplete. It is completed at t_{build} time when the IR-RTL part of the table is generated from the specifications of the target.

The Generic to Gimple table is detailed in section Chapter 4 [The Gimple IR], page 12 and the Gimple to IR-RTL table requires two different views: the one at $t_{develop}$ is described in section Section 4.1 [Development time Gimple to IR-RTL conversion issues], page 15 and the one at t_{build} is in section Section 4.2 [Build time Gimple to IR-RTL conversion issues], page 17. The RTL subsystem of GCC (section Chapter 5 [The RTL], page 19) requires a more detailed treatment.

In Gimple, the data objects retain their form in the AST/Generic machine. The code objects `do`, `while`, `for`, `break`, `switch`, `continue` from the AST/Generic representation are re-expressed using the `if` and `goto` statements. In other words, representation in terms of the Gimple IR lowers the control flow abstractions. For all other purposes the Gimple machine is identical to the C machine. Until about GCC version 4.3, the data structures for Gimple representation have been identical to the Generic tree structures. Recently they have been changed to Gimple tuples. We discuss the tree based structures. The Gimple representation also simplifies complex expressions to a set of simple expressions by introducing any additional temporaries that may be required. Lowering only the control flow enables implementing control flow based machine independent optimizations through representation in static single assignment (SSA) form. The GCC community often refers to Gimple representation as Tree-SSA form. The Gimple is thus a target and source language independent IR, and hence the views at development time and operation time are identical except for the details of the conversion to the IR-RTL representation. Since the Gimple is target independent and the IR-RTL is target dependent, the design, build and operation of the conversion is complex process that is described in sections~Section 4.1 [Development time Gimple to IR-RTL conversion issues], page 15 and Section 4.2 [Build time Gimple to IR-RTL conversion issues], page 17

The Gimple nodes are a subset of the AST/Generic nodes. All the AST/Generic nodes are listed in the '`$GCCHOME/gcc/tree.def`' file in the GCC code base at $t_{develop}$, where `$GCCHOME` is the location on the file system where the pristine GCC (4.0.2) sources have been extracted. Table Table 4.1 lists the Gimple node types for the C front end.

lt_expr	le_expr	gt_expr
ge_expr	eq_expr	ne_expr
unordered_expr	ordered_expr	unlt_expr
unle_expr	ungt_expr	unge_expr
uneq_expr	fix_trunc_expr	fix_ceil_expr
fix_floor_expr	fix_round_expr	float_expr
negate_expr	abs_expr	ffs_expr
bit_not_expr	convert_expr	nop_expr
non_lvalue_expr	view_convert_expr	conj_expr
realpart_expr	imagpart_expr	sizeof_expr
alignof_expr	plus_expr	minus_expr
mult_expr	trunc_div_expr	ceil_div_expr
floor_div_expr	round_div_expr	trunc_mod_expr
ceil_mod_expr	floor_mod_expr	round_mod_expr
rdiv_expr	exact_div_expr	min_expr
max_expr	lshift_expr	rshift_expr
lrotate_expr	rrotate_expr	bit_ior_expr
bit_xor_expr	bit_and_expr	bit_andtc_expr
complex_expr	block	integer_cst
real_cst	complex_cst	vector_cst
string_cst	function_decl	label_decl
const_decl	type_decl	var_decl
parm_decl	result_decl	field_decl
namespace_decl	compound_expr	modify_expr
init_expr	target_expr	cond_expr
bind_expr	call_expr	with_cleanup_expr
cleanup_point_expr	with_record_expr	truth_andif_expr
truth_orif_expr	truth_and_expr	truth_or_expr
truth_xor_expr	truth_not_expr	save_expr
unsave_expr	rtl_expr	addr_expr
reference_expr	entry_value_expr	fdesc_expr
predecrement_expr	preincrement_expr	postdecrement_expr
postincrement_expr	va_arg_expr	goto_subroutine
labeled_block_expr	expr_with_file_ location	exit_block_expr
arrow_expr	expr_stmt	compound_stmt
decl_stmt	if_stmt	return_stmt
goto_stmt	label_stmt	asm_stmt
scope_stmt	file_stmt	case_label
stmt_expr	compound_literal_expr	cleanup_stmt
component_ref	bit_field_ref	indirect_ref
array_ref	array_range_ref	label_expr
goto_expr	return_expr	exit_expr
void_type	integer_type	real_type
enumeral_type	pointer_type	offset_type
reference_type	array_type	record_type
union_type	qual_union_type	function_type
error_mark	identifier_node	tree_list
tree_vec	placeholder_expr	srcloc

Table 4.1: The Gimple nodes.

A Gimple representation of input function is a “tree” of nodes. It corresponds to a linearised control flow representation of the AST/Generic tree and hence actually is more of a linear list of “tree” nodes. A Gimple node is represented using the following (annotated) structure in the file ‘$GCCHOME/gcc/tree.h’:

```
struct tree_common
{
  tree chain;                  /* Chaining ptr */
  tree type;                   /* Expression type ptr */
  union tree_ann_d *ann;

  ENUM_BITFIELD(tree_code) code : 8; /* Node type (tree.def) */

  /* Various flags */
  unsigned side_effects_flag : 1;
  unsigned constant_flag : 1;
  unsigned addressable_flag : 1;
  unsigned volatile_flag : 1;
  unsigned readonly_flag : 1;
  unsigned unsigned_flag : 1;
  unsigned asm_written_flag: 1;
  unsigned nowarning_flag : 1;

  unsigned used_flag : 1;
  unsigned nothrow_flag : 1;
  unsigned static_flag : 1;
  unsigned public_flag : 1;
  unsigned private_flag : 1;
  unsigned protected_flag : 1;
  unsigned deprecated_flag : 1;
  unsigned invariant_flag : 1;

  unsigned lang_flag_0 : 1;
  unsigned lang_flag_1 : 1;
  unsigned lang_flag_2 : 1;
  unsigned lang_flag_3 : 1;
  unsigned lang_flag_4 : 1;
  unsigned lang_flag_5 : 1;
  unsigned lang_flag_6 : 1;
  unsigned visited : 1;
};
```

This structure represents information common to every Gimple node type. More specific information of each node type resides in the corresponding structure declarations in ‘$GCCHOME/gcc/tree.h’.

Lowering the control flow is done by replacing control flow nodes for loop constructs (`for`, `while` etc.) and complex branch constructs (`switch`) with simple conditional control flow, `if`, and the unrestricted branch, `goto`. The AST/Generic to Gimple translation table is directly implemented as a case analysis over the code nodes above in '`$GCCHOME/gcc/gimple.c`'. The translation is initiated by the function `gimplify_function_tree ()` with the translation table in the function `gimplify_expr ()`.

4.1 Gimple → IR-RTL Conversion Issues at $t_{develop}$

One critical part of GCC is the implementation of the Gimple → IR-RTL translation required at t_{run} . This is critical since the Gimple is target independent while the IR-RTL is target specific. The problem then is to design and implement the translation at $t_{develop}$ given that the actual target would be known only at t_{build} as shown in Figure 4.1.

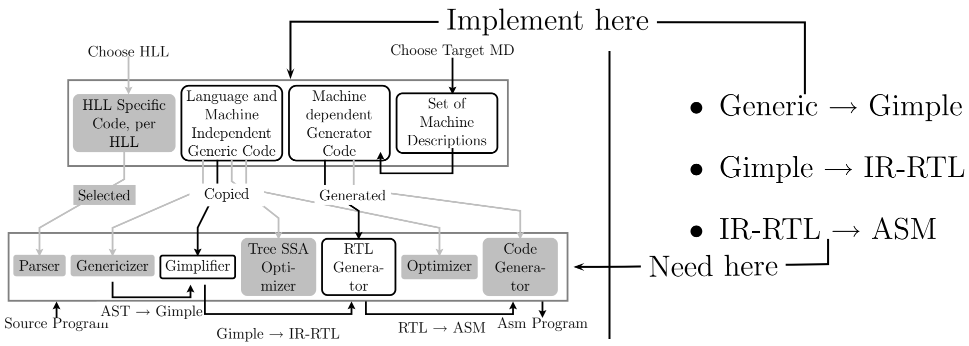

Figure 4.1: The Gimple → IR-RTL translation is required at t_{run} . It must be implemented at $t_{develop}$ and converted at t_{build} .

The GCC technique is to separate the two parts – the target independent Gimple and the target specific IR-RTL – of the table at $t_{develop}$, and join them at t_{build} . The separation at $t_{develop}$ only separates the table data structure. However, the use of the data structure in performing the translation can still be implemented at $t_{develop}$. As shown in Figure 4.2, at $t_{develop}$, therefore, GCC does the following:

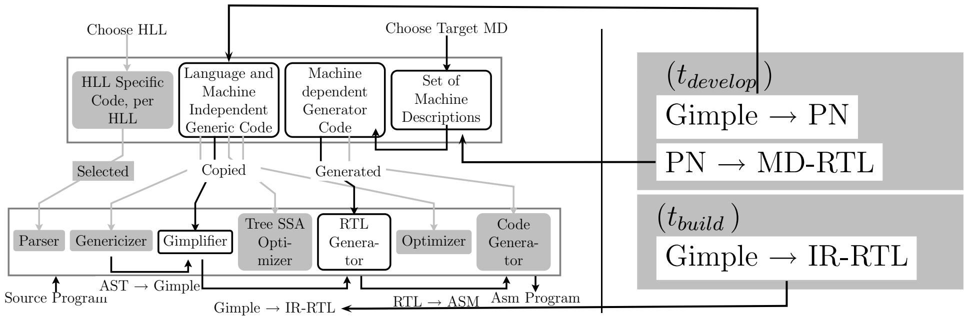

Figure 4.2: At $t_{develop}$, the Gimple → IR-RTL translation is separated into two tables connected by "Pattern Names" ("PN"). The "Gimple → PN" part is implemented in the language independent code base and the "PN → MD-RTL" part is implemented in the machine descriptions. At t_{build} , the "PN → MD-RTL" part is converted to obtain the IR-RTL based "Gimple → IR-RTL" table. The "generation" code required for the conversion is implemented at $t_{develop}$.

- implements the Gimple part of the table,

- implements the target specifications in the RTL language, and

- implements the *generation* code that would be used at t_{build} to generate the IR-RTL part of the table using the target specifications.

To facilitate the joining, GCC needs some mechanism to specify at development time $t_{develop}$ the semantic identity between the Gimple part and the IR-RTL part of the translation table. In other words, it needs to find out which IR-RTL pattern can cover a given Gimple subtree. GCC does this statically by defining arbitrary strings, called "pattern names" (PN), some of which are "standard" (SPN). Conceptually, the PNs give rise to two distinct tables: the Gimple to PN and the PN to IR-RTL. The first table can be implemented at $t_{develop}$ while the second can be implemented using IR-RTL based specification system at $t_{develop}$ and converted to the IR-RTL table at t_{build} . This is pictorially depicted in Figure 4.3

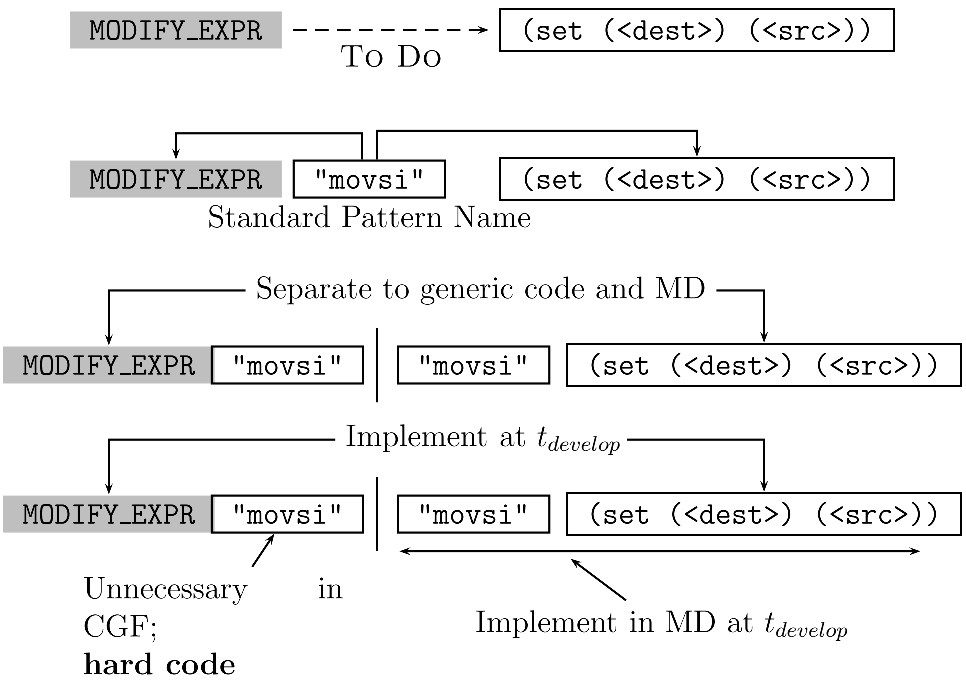

Figure 4.3: Separating the Gimple to IR-RTL translation finite function into target independent LHS and target dependent RHS at design time. The MODIFY_EXPR is a Gimple construct that represents an assignment operation. The (set (<dest> (<src>))) is an IR-RTL construct that represents the same operation in a target specific way. The " movsi" is an SPN that logically glues the two sides. (CGF: Compiler Generation Framework; see text.)

which describes the concepts behind the separation of the desired table (at the top) into the two tables (at the bottom). The syntactic issues of the implementation in the machine description will be discussed in section Section 5.3.1 [MD-RTL - Describing Target Machines], page 24.

4.2 Gimple → IR-RTL Conversion Issues at t_{build}

Having separated the Gimple → IR-RTL translation table at development time and implemented code to join them back at build time, we now conceptually expose the events that occur during t_{build} . Figure 4.4 explains them. The figure is divided into

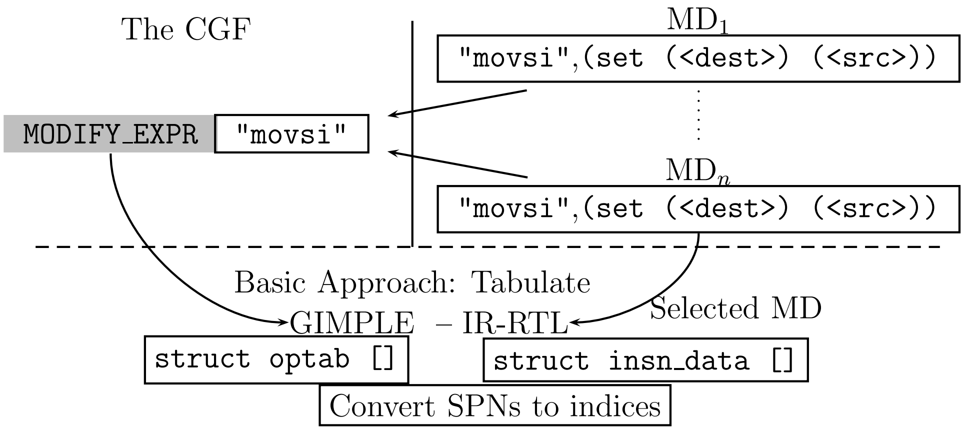

Figure 4.4: Joining the Gimple to IR-RTL translation finite function target independent LHS (`optab[]`) and target dependent RHS (`insn_data[]`) at build time, t_{build} . The contents of `insn_data[]` are from the selected machine description. Above the dashed line we have the GCC system as developed during $t_{develop}$. Below the dashed lines we have the situation at t_{build} .

two parts by the dashed line. Above the line we depict the situation at $t_{develop}$ and below the line we have the situation at t_{build} . At $t_{develop}$, GCC is the implementation of the target independent part of the Gimple → IR-RTL table, and a collection of machine descriptions (MD_1 . . . MD_n) that each implement the target specific (the IR-RTL) part of that table. One of the machine descriptions is chosen at t_{build} . The MD-RTL specifications are converted to C code and the RTL expression of PN is stored in the array `struct insn_data []`. The indices into this array are stored in the `struct optab []` array which is itself arranged according to the PNs. Since the PN for a given Gimple node is known, the `optab` array is can be used to obtain the index of the corresponding IR-RTL pattern. It is in this sense that the PNs are converted to "indices" and are used to join the separated parts of the Gimple → IR-RTL table. Note that the `insn_data []` array is filled in the sequence written in the machine description by the machine description author.

4.3 Gimple → IR-RTL Conversion at t_{run}

At the end of t_{build} the Gimple → IR-RTL table for the chosen target is complete and can be used at t_{run} to perform the translation. The Gimple representation is traversed in a depth first manner. The child nodes that (usually) represent the operands are thus first expanded to RTXs representing the operands – typically RTXs that represent pseudoregisters, but could also be home locations. Given a Gimple node, the `optab` table is consulted to obtain the location of the corresponding IR-RTL in the `insn_data` table. Using this index to lookup the `insn_data` table yields the *function* that emits the IR-RTX of the Gimple node. This function is called to emit the IR-RTX with the generated operand RTXs passed as arguments. The complete program is thus a linear (doubly) linked list of IR-RTXs.

5 The RTL

The main point of section Chapter 3 [The GCC System], page 5 is that three different time periods, $t_{develop}$, t_{build} and t_{run} , need different mental views to understand their structure. The GCC system at $t_{develop}$ and the gcc system at t_{run} are connected through the RTL representation that undergoes conversion at t_{build} from form at $t_{develop}$ to the form to be used at t_{run} . This section describes the purpose of the RTL system and the consequent two languages for each purpose in Section 5.2 [The Two Languages - MD-RTL and IR-RTL], page 19.

5.1 Some RTL Concepts

The RTL subsystem abstracts out the essential characteristics of typical hardware. It conceptually uses two semi-infinite discrete memories that we call as the RTL core memory and the RTL pseudoregisters. These are conceptual objects that mimic the primary memory and register sets respectively of the supported CPUs.

One memory, called the RTL core memory, represents the layout in virtual memory of the real target, and is linear and byte addressable. The RTL memory model is close to most target hardware and the data abstraction gap is quite small. The mapping of RTL core memory to target memory is done on a per target basis in the GCC machine description system through a set of C preprocessor macros. The information is incorporated into the compiler at compiler build time. Conceptually, the Gimple objects will be mapped to RTL memory, and then be lowered to target memory. Note that as a result of mapping Gimple objects to a linear addressable memory, the named objects now can have computable addresses. In other words, the name to address association can now be determined. The RTL provides data types that are fractions and multiples of the word size similar to typical target hardware.

The other type of unbounded memory that RTL has is a set of pseudo registers. The RTL register tape is used to allocate pseudo registers. During the conversion from Gimple representation to RTL representation, objects are allocated fresh pseudo registers from this semi infinite pool. In later phases, when register allocation is performed, the program objects get the hard registers or are spilled on to the RTL core memory. Different targets have different number of registers. The RTL register memory designates the first N registers to correspond to the N hardware registers of the target. Thus the actual pseudo registers start from the N^{th} location in the RTL register tape since the registers are indexed starting from zero.

5.2 The Two Languages: MD-RTL and IR-RTL

The RTL subsystem of GCC is used for two distinct purposes in GCC:

1. specification of target instruction semantics at $t_{develop}$, and
2. representation of a program being compiled at t_{run} .

At $t_{develop}$ the RTL is in human readable MD-RTL form while at t_{run} it is in machine readable IR-RTL form.[1] The IR-RTL form is the result of compiling a C representation that is generated at t_{build} from the MD-RTL specifications.

[1] IR-RTL is *dumped* in a human readable Lisp like syntax.

5.2.1 MD-RTL: Capturing Target Instruction Semantics

To *specify* the semantics of a target instruction, we need to:

- *capture* the semantics of the instruction, and
- *describe* it's properties that are relevant at t_{run} .

The specification process is analogous to the emulation of another processor, say a MIPS, on a given processor, say an i386. The operational part of a target (say MIPS) instruction can be captured using the operations of a given processor (i386). Next, the operands have to be mapped. Some information of this part can be captured at $t_{develop}$, for instance the sizes. The rest, like the actual values, cannot be as they are run time dependent. However the nature of the values, integer or floating point and such, can be specified at $t_{develop}$. The MD-RTL language is made up of a few constructs used to describe the target instruction properties and operators of a fictitious processor to capture target instruction semantics as shown in Figure 5.1.

Figure 5.1: The MD-RTL language. The MD constructs describe properties of target instructions and the operators (of the fictitious processor) capture the target instruction semantics.

5.2.2 IR-RTL: Expressing a Compilation

To *express* a program being compiled in an IR-RTL based target specific (linear) representation, we need to:

- *express* the target specific semantics of the instruction in the sequence, and
- *describe* it's control flow properties and layout that are relevant at t_{run} .

This expression process is analogous to writing the program in target assembly code, but in a target independent, IR-RTL based syntax. The operational part of a target instruction can be expressed using the same operators used in the MD-RTL specification language. Thus the operators used in the IR-RTL are the same as those used in MD-RTL. Next, the layout can be described by augmenting the expression with the linking information about the predecessor and successor expressions. In case the expression, i.e. the operational part, has branching semantics, an explicit `jump` or `return` for example, then such control flow effects are separately described. The IR-RTL language is shown in Figure 5.2 and is made up of the same operators as the MD-RTL and additional constructs that describe the other aspects of intermediate representation at t_{run} .

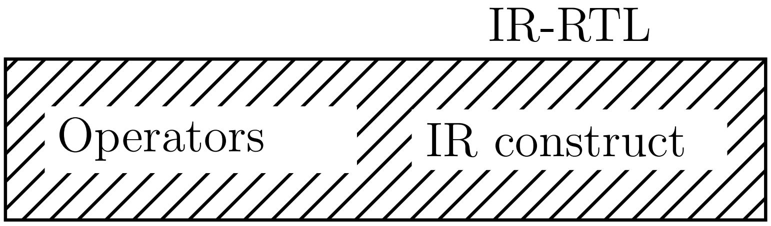

Figure 5.2: The IR-RTL language. The the operators (of the fictitious processor) capture the target instruction semantics and the IR constructs describe the control flow and layout properties of target instruction sequence.

An intermediate representation may also be required to propagate information computed in various passes. Rather than using any explicit constructs, GCC adds such information as the operands of the IR-RTL objects – both, the objects that express and the objects that describe. In terms of the "processor emulation" analogy, the IR-RTL represents the program being compiled in terms of the "emulated processor"!

5.2.3 GCC Implementation of MD-RTL and IR-RTL

Although we have viewed the MD-RTL and IR-RTL as two distinct languages, GCC implements them as one whole RTL subsystem, as shown in Figure 5.3. To see the correspondence between the implementation and the two languages, we divide the RTL objects listed in '`$GCCHOME/gcc/rtl.def`' into three different kinds as shown in Figure 5.3 and then described below:

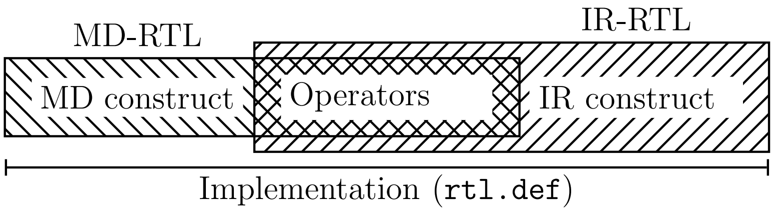

Figure 5.3: The MD-RTL and IR-RTL languages, and their implementation in GCC using three kinds of RTL objects. The MD constructs appear only in the MD-RTL language. The IR constructs appear only in the IR-RTL language. The operators are common to both the languages. The implementation lists *all* the RTL objects together in '`$GCCHOME/gcc/rtl.def`' and does *not* distinguish them (except in some comments in the code).

1. A set of RTL objects called *operators* that by themselves are basic elementary computation operations and are used to form RTL expressions (RTX, for short) that express the target instruction semantics in both – the MD-RTL and the IR-RTL – languages. Table (Table 5.1) lists the operators. We emphasize: the target semantics are captured in the RTXs and RTXs are constructed using operators.

Note that, unlike the current[2] GCC documentation, by "RTX" we mean only the expressions created using operators to express target semantics.

2. A set of RTL objects that are used to describe the control flow layout of a sequence of RTXs that represent the input program being compiled during t_{run} . We refer to these as *IR (Intermediate Representation) constructs*, or *Flow Representation constructs* and are listed in table (Table 5.2). Each RTX in the sequence is embedded in a suitable IR construct object that expresses the nature of the control flow (sequence or branch) that the RTX is a part of.

   Additionally, we also include RTL constructs that are used for book keeping purposes during a compilation operation as a part of this set of constructs. Constructs like `note` are useful to record some information computed in a pass for use at later times, for instance.[3]

   We will refer to IR-RTL expressions as *IR-RTX*s. The IR-RTXs contain the RTXs that express the target instruction semantics. Thus a program being compiled is represented as a list of IR-RTXs with the RTX within each IR-RTX capturing the semantics of a target instruction.

3. A set of constructs that specify the semantic patterns of target instructions in terms of the operators. We refer to these as *MD (Machine Description) constructs* and are listed in table (Table 5.3). Each instruction semantics pattern, or simply instruction pattern (as the GCC documentation calls it) is enveloped by a suitable MD construct object that expresses the nature of the specification.

   We will refer to MD-RTL expressions as *MD-RTX*s. The MD-RTXs contain the RTXs that express the target instruction semantics. Thus a machine description of target instruction semantics is a sequence of MD-RTXs with the RTX within each MD-RTX capturing the semantics of a target instruction.

---

[2] As of 2007.

[3] This does not mean that whenever there is a need to propagate information across passes, the `note` construct is used. Global variables within the code are also used, as are extra fields in the IR RTL expressions. We ignore these details.

abs	addr_diff_vec	addressof
addr_vec	and	ashift
ashiftrt	asm_input	asm_operands
call	call_placeholder	cc0
clobber	compare	concat
cond	cond_exec	const
constant_p_rtx	const_double	const_int
const_string	const_vector	div
eq	expr_list	ffs
fix	float	float_extend
float_truncate	ge	geu
gt	gtu	high
if_then_else	include	insn_list
ior	label_ref	le
leu	lo_sum	lshiftrt
lt	ltgt	ltu
mem	minus	mod
mult	ne	neg
nil	not	ordered
parallel	pc	phi
plus	post_dec	post_inc
post_modify	pre_dec	prefetch
pre_inc	pre_modify	queued
range_info	range_live	range_reg
range_var	reg	resx
return	rotate	rotatert
scratch	set	sign_extend
sign_extract	smax	smin
sqrt	ss_minus	ss_plus
ss_truncate	strict_low_part	subreg
symbol_ref	trap_if	truncate
udiv	umax	umin
umod	uneq	unge
ungt	unknown	unle
unlt	unordered	unsigned_fix
unsigned_float	unspec	unspec_volatile
use	us_minus	us_plus
us_truncate	value	vec_concat
vec_duplicate	vec_merge	vec_select
xor	zero_extend	zero_extract

Table 5.1: The operators in MD-RTL and IR-RTL.

insn	call_insn	note
barrier	jump_insn	code_label

Table 5.2: The IR RTLs.

`absence_set`	`address`	`attr`
`attr_flag`	`automata_option`	`define_asm_attributes`
`define_attr`	`define_automaton`	`define_bypass`
`define_combine`	`define_cond_exec`	`define_cpu_unit`
`define_delay`	`define_expand`	`define_function_unit`
`define_insn`	`define_insn_and_split`	`define_insn_reservation`
`define_peephole`	`define_peephole2`	`define_query_cpu_unit`
`define_reservation`	`define_split`	`eq_attr`
`exclusion_set`	`match_dup`	`match_insn`
`match_op_dup`	`match_operand`	`match_operator`
`match_parallel`	`match_par_dup`	`match_scratch`
`presence_set`	`sequence`	`set_attr`
`set_attr_alternative`		

Table 5.3: The MD RTLs.

These three sets are distinct and disjoint. The operators and MD constructs define the MD-RTL language and are used to describe target instruction semantics at $t_{develop}$. The IR constructs and operators define the IR-RTL language and are used to construct the intermediate representation of a compilation run at t_{run} . The current[4] GCC documentation does not distinguish between the different languages, nor the different RTL objects and RTL expressions and heavily depends on the context for the distinction.

5.3 How the RTL Works

In this section we describe how the MD-RTL and the IR-RTL machinery described above is employed by GCC. We use the time durations to fix the details with respect to Figure 3.2.

5.3.1 MD-RTL: Describing Target Machines ($t_{develop}$)

MD-RTL as a specification system is made up of operators and MD constructs. The operators are used to construct RTXs that capture target instruction semantics and are called as *instruction patterns*. Two main issues arise at specification time:

1. At $t_{develop}$, the goal is to create target specific part of the Gimple to IR-RTL translation table. The MD-RTL is to be used to capture and express the target semantics.
2. During that actual translation from Gimple to IR-RTL at t_{run} not all information about the target is available. For instance, the target register to be used as an operand to an instruction. However, it is possible to specify the *properties* required by the register object. The MD-RTL should be able to express such run time scenarios.

The RTX within a MD-RTX addresses the first issue by capturing target instruction semantics. The other MD construct within a MD-RTX address the second issue through various constructs (e.g. `match_operand`). Additionally, we may need to capture some target properties that can provide useful information to the compiler. For example, the pipeline characteristics of the target, if any. The specification system thus has three main goals.

[4] As of 2007.

1. To create the target specific part of the Gimple to IR-RTL translation table. This is useful during Gimple to IR-RTL transformation.
2. To provide target specific information that can be used by the compiler to improve the code. This results in transforming one IR-RTL sequence to another, possibly a better one.
3. To ensure that at the end of the compilation every IR-RTL object maps to at least one target instruction.

The last goal is implicit in the GCC (and gcc) system. To facilitate the construction of the Gimple to IR-RTL table at t_{build} , GCC standardises a few pattern semantics called Standard Pattern Names (SPNs). Target specifications are expected to specify instruction patterns corresponding to the semantics of these pattern names. It is possible that for some targets the semantics of a pattern name may correspond to a single instruction while for others a sequence of RTL expressions may be required. It may also be possible that a pattern may correspond to different RTXs depending on some target properties which are known only at t_{run} . In general, the specification system has to be designed with the possible scenarios that might occur at t_{run} . For such purposes the specification system needs constructs to capture the exact situation at t_{run} that might occur. The simplest case is that the semantics of a SPN correspond to a single target instruction through a single RTX. This is usually the case and the MD construct called `define_insn` is used to associate a SPN to the RTX that implements the semantics. Additionally, it also associates the same RTX that implements the semantics to the corresponding target instruction in assembly syntax. In case the semantics of an SPN require different RTXs depending on target properties that are known at runtime (for instance the properties of the register operand), then the MD construct `define_expand` is used in the specification.

Example: As an example consider the SPN `addsi3` that has the addition semantics of three operands each of size SI (single RTL integer). For concreteness, we assume a fictitious target that has registers with the same width as SI mode of RTL, and a `sum` instruction that adds the integers in the first two register operands and deposits the result in the third register operand. There is an implicit assignment operation in the instruction. The operator that expresses addition is `plus`, and the operator that expresses assignment is `set`. Since this is a situation in which the semantics are implemented by a single target instruction, we use the `define_insn` construct to specify a MD-RTX as:

```
;;------------------------------------------------------------
;; MD-RTX defining the addition operation on the target
(define_insn  ; MD construct, at least 4 arguments
; 1. The name of the SPN
   "addsi3"
; 2. RTX capturing the semantics of the equivalent (to SPN)
;     target instruction (the "sum" instruction)
   ; RTL assignment operator
   (set
      ; Describe third operand, required to be a register
      (match_operand 2 "register_operand" "r")
      ; RTL addition operator
      (plus
         ; Describe first operand, required to be a register
```

```
            (match_operand 0 "register_operand" "r")
            ; Describe second operand, required to be a register
            (match_operand 1 "register_operand" "r")
         )
      ) ; END of RTX implementing the semantics of SPN
   ; 3. Unused third operand (in this case) of define_insn
      ""
   ; 4. The actual target instruction in target assembly syntax
   ;     %1, %2, %3 are replaced by the actual target registers.
      "sum  %1 %2 %3"
   ; 5. Unspecified fifth operand that is optional anyway.
   )
   ;;------------------------------------------------------------
```

The `define_insn` construct in the MD-RTX above requires four operands and may take an optional fifth operand (not shown above). The first operand is the name of the SPN – `addsi3` in the above example. The second operand is an RTX that implements the specification of the SPN in a target specific manner. In the above case, the RTX ensured that all the three operands are register operands by describing the operand characteristics using the `match_operand` MD construct. This need was particular to the target. Some other targets could also work with some operands in core memory rather than registers, and in which case the operand matching specifications would be accordingly written. This is what makes the RTXs target specific; they capture the target semantics. The third operand is irrelevant at the moment. The fourth operand is the string that expresses the same semantics in target specific assembly syntax. In particular, the addition operation that is semantically identical to `addsi3` is written out with the actual hardware registers expressed as parameters. The values of these parameters, i.e. the actual registers, will be filled in with pseudoregisters that satisfy the properties expressed by the `register_operand` expression in the `match_operand` construct.

Note that the RTX within the MD-RTX has been constructed using operators and has been embedded within a suitable specification RTL construct using the MD constructs. We also observe that the SPN is coarse enough to correspond to some fine detail of the Gimple representation, and it can be fine tuned to target specific expressions with a little effort. Also, the specification is in human readable Lisp like syntax. This is converted to C code at build time. Finally, the syntactic and operational details of each MD construct can be found on the online GCC internals manual [GCC Internals (by Richard Stallman)], page 34.

5.3.2 From MD-RTL to IR-RTL (t_{build})

At t_{build}, the machine description is chosen, the RTXs are extracted from the MD and processed as described in section Section 4.2 [Build time Gimple to IR-RTL conversion issues], page 17. Extraction of the RTXs implies converting each object in the MD-RTL form to an internal representation using the RTL structure in the '`$GCCHOME/gcc/rtl.h`' file. The RTL structure is (some comments from the original source have been removed):

```
/* Common union for an element of an rtx.  */
union rtunion_def
{
  int                          rt_int;
```

```
  unsigned int               rt_uint;
  const char                *rt_str;
  rtx                        rt_rtx;
  rtvec                      rt_rtvec;
  enum machine_mode          rt_type;
  addr_diff_vec_flags        rt_addr_diff_vec_flags;
  struct cselib_val_struct  *rt_cselib;
  struct bitmap_head_def    *rt_bit;
  tree                       rt_tree;
  struct basic_block_def    *rt_bb;
  mem_attrs                 *rt_mem;
  reg_attrs                 *rt_reg;
};
typedef union rtunion_def rtunion;

/* RTL expression ("rtx").  */
struct rtx_def
{
  /* The kind of expression this is.  */
  ENUM_BITFIELD(rtx_code) code: 16;

  /* The kind of value the expression has.  */
  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     frame_related  : 1;
  unsigned     return_val     : 1;
  union u {
    rtunion       fld[1];
    HOST_WIDE_INT  hwint[1];
  };
};
```

Each RTL object, particulary the operators and IR constructs, internally corresponds to an instance of this structure. The operation is identified using the `code` field whose numerical value in the code field is obtained by simply enumerating all the RTL objects defined in '`$GCCHOME/gcc/rtl.def`'. Most other details are in the union defined by `rtunion`. The Lisp like syntax of MD-RTL makes this an easy internal representation with the expression syntax defining the structure of the internal lists-of-RTL-objects. To *use* the RTX in the specification to represent a computation in a compilation of input program at t_{run} , the GCC build process *generates* (i.e. emits) C code that:

1. Task 1: Identify the particular operator code required.

2. Task 2: Generate a C function to emit an instance of the RTL structure `rtx_def` structure with that code.
3. Task 3: Generate a C function that will chain the instantiated RTL object into the RTL based IR of the input program.

Example: The `addsi3` example specification (from the example above)

```
;;------------------------------------------------------------
(define_insn "addsi3"
   (set (match_operand 2 "register_operand" "r")
        (plus  (match_operand 0 "register_operand" "r")
               (match_operand 1 "register_operand" "r")))
   ""
   "sum  %1 %2 %3"
)
;;------------------------------------------------------------
```

is converted to the equivalent C code below at generation time t_{gen} during t_{build} :

```
/*---------------------------------------------------*/
rtx
gen_addsi3 (rtx operand0, rtx operand1, rtx operand2)
{
   ...
   emit_insn (gen_rtx_set (SImode, op0, op1, op2));
   ...
}
/*---------------------------------------------------*/
```

The generated C functions start with "`gen_`" prefix and the name of the pattern ("`addsi3`" in this example) as the suffix. The specified RTX starts with the `set` operation. The operands of the `set` are themselves other RTXs (e.g. `plus`) and may be initialised by their equivalent C code (not shown) before being passed to the `gen_rtx_set()` function shown above. The `gen_rtx_set ()` function (task [step2], page 28) is the code to create an instance of the RTL structure above with the `code` field initialised to the (enumerated) value of the `SET` operator (task [step1], page 27).

Since the internal representation of the RTL object would actually be a part of a (doubly linked) list while representing an input program at t_{run} , the `gen_addsi3 ()` function calls the `emit_insn()` function to chain the output of `gen_rtx_set ()` into the chain (task [step3], page 28).

All of these steps are bundled into (usually) a single C function that conceptually now "emits" the entire IR-RTX corresponding to the SPN. This function[5] is stored in the `insn_data []` array in Figure 4.4. The index at which this function is stored in `insn_data []` is stored in `optab` array at the offset indexed using the SPN. All the specified RTXs in the machine description are thus processed at t_{build} and collected into the C data structure called `insn_data`. The "SPN" corresponding to a given Gimple construct is known at $t_{develop}$ and can be used to index the `optab` structure. This returns the index into the `insn_data` structure to yield the function pointer that would instantiate the RTX pattern

[5] It's function pointer to be precise.

specified in the MD. Note that because the MD-RTXs captured target semantics (in it's RTX), the RTXs that represent the compilation of a program are also, therefore, target specific.

At the end of t_{gen} all the required C functions are generated. During t_{comp} they will be compiled into the object code which when executed at t_{run} would emit a IR-RTX as a list of objects of the RTL data structure at t_{run} .

This completes the task of generating the target specific code. The complete target specific Gimple to IR-RTL translation table is thus implemented through a target independent data structure that corresponds to the Gimple part and a target specific data structure that corresponds to the IR-RTL part. Both these are connected using the pattern names.

5.3.3 IR-RTL: Representating a Compilation (t_{run})

The IR-RTL representation of the input program being compiled is generated by a depth first traversal of the Gimple representation at t_{run} . During this process the data abstractions are lowered. The program is linear list of IR-RTXs. Each IR-RTX is is obtained from the IR-RTX emitting function in `insn_data[]`. The operands are typically pseudoregisters, but may also be the home locations, and are already available when the IR-RTX emitting function runs (due to the depth first traversal). The operands hopefully satisfy the target specific matching criteria given in the specification in the machine description (e.g. `match_operand` says that the operand should be a "`register_operand`"). The operand matching criteria are not checked at this stage. Thus the RTX in the IR-RTX has the instantiated operands rather than their specifications. For example, the instantiated RTX corresponding to the `define_insn` example above would look like:

```
;;--------------------------------------------------------------
(set
   ; Selected destination (pseudo) register
   (reg 20)
   ; Selected source (pseudo) registers
   (plus (reg 29) (reg 8))
)
;;--------------------------------------------------------------
```

This chain of IR-RTXs is expressed using the IR constructs like `insn`, `jump_insn` etc. to envelope the instantiated RTX as required. The actual operations like data movement, arithmetic etc. are expressed using the corresponding operators `mov`, `plus` etc. within the RTX. Thus the IR-RTL representation of an input program being compiled is made up of chaining information expressed using IR RTXs. The `insn` construct is used to express linear chaining, while the `jump_insn` is used to denote that the containing RTX is a branch operation to some other location thus disrupting the linear control flow chain. It is because of such constructs that we suggest referring to IR constructs as "(control) flow RTLs".

Example: As an example, the enveloping of the instantiated RTX above by `insn` construct would look like:

```
;;-----------------------------------------------------------
(insn
   ; the current node number in the linear chain
   101
```

```
        ; the previous node number in the linear chain
        100
        ; the next node number in the linear chain
        102
        ; the current RTX
        (set
           ; Selected destination (pseudo) register
           (reg 20)
           ; Selected source (pseudo) registers
           (plus (reg 29) (reg 8))
        )
        ...
     )
     ;;----------------------------------------------------------
```

Expression texts of the above type are dumped by `gcc` when requested (See: [GCC – An Introduction], page 34) to dump the IR-RTL IR at various stages of a compilation run. The IR-RTL representation is obtained from the Gimple representation by translating the operational semantics of the Gimple nodes using the corresponding IR-RTXs (as determined by the `optab` and `insn_data` data structures). However, not all operand information is available at this point and the initial conversion to IR-RTL remains incomplete. As the processing in RTL phase of the compiler proceeds, the required information is obtained (e.g. the register allocator fixes the actual hardware registers to be used in place of the pseudo registers). Instruction selection in GCC is partial and is completed over a set of phases.

6 Summary

In summary, we look at the essentials of the GCC architecture in the sequence of the three time durations of section Section 3.1 [The Impact of Retargetability], page 5 that are a consequence of retargetability. We ignore the AST/Generic generation and target assembly generation. The AST/Generic is the output of the parsing process which is well understood and hence need not be addressed to in an architectural description. The assembly code generation is simplified because at the end of the RTL passes, the IR-RTL representation is complete enough to merely require expression in target assembly syntax. Hence, we look at Gimple and RTL at $t_{develop}$, t_{build} and t_{run} times.

1. $t_{develop}$
   1. Gimple
      1. Separate target independent and target specific part of the Gimple → IR-RTL translation,
      2. Prepare to join the separated parts by
         1. defining SPNs, and
         2. writing the generator code for use at t_{build}
   2. RTL
      1. Use MD construct and operators to construct MD-RTXs that *specify* the semantics of the target instructions in the corresponding machine description,
      2. define the RTL data structure in C that will be used to represent the RTXs in internal form
2. t_{build}
   1. Gimple
      1. Run the generator programs on the chosen MD to list out the pattern names of each defined pattern, and
      2. use the indices obtained to fill the `optab` table with the offset of the corresponding pattern expression in the `insn_data` table.
   2. RTL
      1. Run the generator programs on the chosen MD to list out the patterns in MD-RTL, and convert them to C functions that would emit the IR-RTL form, and
      2. fill in the `insn_data` array with the function pointer to the generated C function.
3. t_{run}
   1. Gimple: Index into the `optab` array to obtain the offset to be used into the `insn_data` array.
   2. RTL: Use the index obtained from the `optab` arrat into the target specific `insn_data` array and obtain the function pointer that points to the function to emit the IR-RTL representation.

Note that because the same Lisp like syntax of RTXs[1] is used during target specific compiler build and a later run of the built compiler, no interconversion (between the specification and runtime internal representation) is required. The MD constructs used for

[1] RTXs are constructed from operators!

specification purposes, and the IR constructs, used for representation purposes are well separated. This particular feature of the RTL system is not clearly brought out by the current GCC documentation.[2]

[2] As of 2007.

7 Conclusion and Future Work

Amongst the many concepts that have gone in into the construction of the GCC system, three main ones are useful to understand the overall picture. Firstly, the abstraction gap is useful in obtaining the phase sequence of lowering-one-gap-at-a-time operations. Second, we identify three time points $t_{develop}$, t_{build} , t_{run} (and t_{gen} and t_{comp} as a part of t_{run}) that serve to delimit the three different conceptual views:

1. the architecture of the Compiler Generation Framework,
2. the architecture of target specific Compiler operation phase sequence, and
3. the architecture of the build system.

These three views are useful to understand GCC. Retargetability implies postponing target specific decisions to build time rather than at design time. Hence at design time, the source is expressed "parameterically" with respect to target properties and the target properties are separately specified on a per target basis. At build time, the target specific parts of the compiler are generated from the specifications of the chosen target. The complete compiler then operates on an input program lowering one abstraction gap at a time for each IR. Thirdly, it is useful to see the implemented RTL system as emerging from the two languages: MD-RTL and IR-RTL. The first is used for specification purposes at $t_{develop}$ and the second is used to represent a program being compiled at t_{run} . They share the same operators. Hence the common implementation of all RTL objects in '`$GCCHOME/gcc/rtl.def`' can be seen as composed of three distinct kinds of objects: the MD constructs, the operators and the IR constructs. The MD constructs and the operators define the MD-RTL language. The IR constructs and the operators define the IR-RTL language. The operators connect the target specific semantics expressed at development time to the representations at the operation time. This connection is explicitly made at build time when the build system operates.

7.1 Future Work

The identification of three views that describe GCC at various time durations can be employed for directing future work. A number of possibilities exist, and we list a few interesting ones.

- Compilation theory:
  - Quantifying the abstraction gap and determination of factors that determine the phase sequence.
  - Formalising the IRs as abstract machines and studying their properties.
  - Formally modeling compilation phases.
  - Study the notion of retargetability.
- Future GCC directions:
  - Implementation changes to push choice of target at operation time rather than build time.
  - Changes in the phase sequence to implement new challenges like auto parallelisation.
  - Changes in the implementation to do better instruction selection.

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. Hans-Peter Nilsson.
   Porting GCC for Dunces.
   (`ftp://ftp.axis.se/pub/users/hp/pgccfd/`)
   May 2000.
3. Deigo Novillo.
   GCC Internals.,
   In International Symposium on Code Generation and Optimization (CGO), San Jose, California,
   2007.
4. Deigo Novillo.
   GCC – Yesterday, Today and Tomorrow.
   In *2nd HiPEAC GCC Tutorial*, Ghent, Belgium,,
   2007.
5. –
   GCC Wiki Book.
   (`http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals`)
   2007.
6. Abhijat Vichare.
   GCC – An Introduction.
   (`$GCCINTDOCSHOME/html/gcc-basic-info.html`)
   2007.

List of Figures

List of Tables

Appendix A Copyright

A.1 GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or 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.