Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

# Retargeting the GNU C Compiler

An inside look on how to deploy the GNU C compiler to your platform of choice.

June 01, 2002
URL:http://www.drdobbs.com/retargeting-the-gnu-c-compiler/184401529

---

## Introduction

The GCC (GNU C compiler) is a highly optimizing and retargetable compiler from the FSF (Free Software Foundation). Richard Stallman originally developed the compiler with the source code made freely available via the FSF's copyleft agreement [1].

Today GCC can generate code for approximately 41 different microprocessors. The ability to generate code for many different processor architectures is known as "retargetability." GCC also supports different languages including C, C++, Java, Ada, Pascal, and Fortran.

## Compiler Overview

As shown in Figure 1, a compiler is typically divided into two parts, a front end and a back end. The front end is responsible for reading and understanding the input language, issuing diagnostic messages, and translating the program into an IL (intermediate language) suitable for the back end. The GCC front end converts the input language into a tree-based IL. The GCC front end eventually converts the tree-based IL into a more machine-specific IL known as RTL (register transfer language). A compiler's back end is typically responsible for code optimizations and code generation for a particular microprocessor. The GCC back end includes most state-of-the-art optimizations including instruction scheduling, various loop optimizations, and common sub-expression elimination. The GNU compiler's optimizations work directly on the RTL with machine-specific details provided by the machine description files. This means that once a retarget to a new processor is complete all of GCC's state-of-the-art optimizations already work on the new processor.

## The RTL IL

The GCC back end performs all of its operations on the RTL IL. RTL is a low-level language that models the registers, memory, and instructions of a microprocessor. RTL is written and modeled in a "lisp-like" form, which affords it tremendous power and flexibility. Unlike other ILs, such as tuples, RTL imposes no arbitrary limits on constructs such as the number of operands except where necessary to enforce correctness.

## Machine Modes

The first important idea in RTL is a machine mode. A machine mode represents the data type of an object such as a register or memory. It is a machine-independent way of describing a machine-specific constraint, namely the size of an object.

Table 1 illustrates some commonly used machine modes in GCC and their machine-independent meaning.

Based on information in the target machine macro file (described later), GCC maps the machine mode to specific bit sizes on the target machine. For example, on an 8-bit byte machine like the 386, **QImode** would be equal to 8 bits, **HImode** would be 16 bits, and **SImode** would be 32 bits. However, on a port that I completed to a 16-bit DSP processor, **QImode** was equal to 16 bits, **HImode** was equal to 32 bits, and **SImode** was equal to 64 bits. Based on this description, obviously the only information you can infer about modes is their sizes relative to each other. For example, you can always assume that **HImode** is twice the size of **QImode**.

## RTL Object Types

RTL has five types of objects: expressions, integers, wide integers, strings, and vectors. For our purposes, expressions and vectors are the most important and commonly used. RTL expressions represent hardware concepts like registers, memory, and instruction operations. They are distinguished by a predefined expression code, such as a **plus** operation. By examining the expression code, it is possible to determine how many operands an expression has and what kind of operands they are (i.e., their type, such as integer or floating point). Expressions are written as parentheses containing the name of the expression type, its flags and machine mode if any, and then the operands of the expression (separated by spaces). Below is an example of an arithmetic expression in RTL:

```
(plus:SI (reg:SI 2)
                (const_int 10))
```

This expression is interpreted as follows: add register **2** with machine mode **SImode** and the integer constant **10**. The addition is carried out in **SImode**. Note that integer constants have no machine mode and that the expression does not specify a destination for the result of the addition. Finally, note the similarity to lisp code and the heavy emphasis on parenthesis.

A vector is an arbitrary number of expressions. The number of expressions is embedded within the vector itself. The vector is a very powerful way of expressing parallelism among a set of expressions or more realistically instructions, as you will see later.

#### Constant Type Expressions

Table 2 shows some of the commonly used constant RTL expression types.

#### Register and Memory Types

Table 3 shows some of the commonly used register and memory RTL types.

#### Arithmetic RTL Expression Types

Table 4 shows some commonly used arithmetic expressions.

### The Retargeting Process

The first step in the retargeting process is to understand the architecture of the target microprocessor. Key items to understand are the pipeline model of the processor (e.g., are there hazards or stalls?), the register file (e.g., is it orthogonal or are there special purpose registers?), and the instruction set.

The second step in the retargeting process is to define the ABI (application binary interface). The ABI defines how arguments are passed to functions, the layout of the stack frame, and register usage conventions.

The third and final step in the retargeting process is to define three key files that will describe the microprocessor and its operating environment to GCC. These three files are the target machine macro file (**machine.h**), the machine description file (**machine.md**), and a third file containing helper functions for the previous two files (**machine.c**) [2].

The manual, *Using and Porting the GNU Compiler Collection (GCC)* [3], is the bible on retargeting a GCC port and should be read and understood before beginning a port. Additionally, if possible, finding and learning from other GCC ports that have a processor with a similar architecture is very helpful.

### The Target Description Macro File

The target description macro file (**machine.h**) is used to parameterize portions of GCC that would otherwise have to be modified for each target processor. My typical approach to writing the target description macro file is to develop it by section as outlined in *Using and Porting the GNU Compiler Collection (GCC)* [3]. There are literally hundreds of macros documented in the manual, some of which need to be defined, others that do not, and some that have reasonable defaults. The best advice I can give with regard to these macros is to consult other GCC ports to see which macros they have defined; the most common macros are defined for almost every port. Below I will outline some of the more important macros you will need to define. Note that in the target description macro file these are C preprocessor macros defined with **#define**.

### Storage Layout

- **BYTES_BIG_ENDIAN** — On a big endian processor, this variable would be defined as **1** and as **0** on a little endian processor.
- **BITS_PER_UNIT** — On an 8-bit byte machine, such as the x86 family, this would be defined as **8**. For a word machine, such as the DSP1600 port I completed, this would be the size of a word on the target machine. In the case of the DSP1600 family, this has the value **16**. As you can see, GCC is not locked into viewing a processor as an 8-bit byte machine as is commonly assumed by many compilers I have encountered.
- **BITS_PER_WORD** — This macro is the number of bits in a word. For the x86 family, this would equal the value **32**. Note that by using the formula **BITS_PER_WORD/BITS_PER_UNIT**, GCC is able to determine the number of bytes in a word. Note, a byte to GCC is **BITS_PER_UNIT**. For the DSP1600 family, this has the value **16**, which means that there is one byte (16 bits) in a word (16 bits).
- **POINTER_SIZE** — This macro is the number of bits in a pointer. This must be a multiple of **BITS_PER_UNIT**. For the x86 processor family, this has the value **32**.

### Layout of Source Language Data Types

- **INT_TYPE_SIZE** — This is a C expression for the number of bits for the type **int**. Default is **BITS_PER_WORD**.
- **FLOAT_TYPE_SIZE** — This is a C expression for the number of bits for the type **float**. Default is **BITS_PER_WORD**.

### Basic Characteristics of Registers

- **FIRST_PSEUDO_REGISTER** — The number of hardware registers known to the compiler. These registers receive number **0** thru (**FIRST_PSEUDO_REGISTER - 1**). The compiler creates pseudo registers for early optimization passes. Register allocation then binds these pseudo registers to actual hardware registers. Let's say a hypothetical new processor contained eight general purposes named **r0**-**r7**. In this case, you would define **FIRST_PSEUDO_REGISTER** as **8**.
- **FIXED_REGISTERS** — Used to initialize an array that indicates by the value **1** that registers are unavailable to the register allocator. The value **0** indicates that the register is available for allocation. The stack pointer is a typical fixed register.
- **CALL_USED_REGISTERS** — Used to initialize an array like **FIXED_REGISTERS**, but has the value **1** for any register that is overwritten by a function call as well as for fixed registers. The value **0** indicates to GCC that the register must be saved and restored in the prologue and epilogue code of a function.

### Register Class Information

The register class macros in this section are very important in determining the quality of register allocation that GCC is able to perform. There is one very important variable in this section defined as:

```
enum reg_class { ..... }
```

where the various register classes are specified as C enumerations.

Quoting from the GCC manual:

> On many machines, the numbered registers are not all equivalent. For example, certain registers may not be allowed for indexed addressing; certain registers may not be allowed in some instructions. These machine restrictions are described to the compiler using register classes. You define a number of register classes, giving each one a name and saying which of the registers belong to it. Then you can specify register classes that are allowed as operands to particular instruction patterns.

There are several register classes that must always be defined: the set of all registers called **ALL_REGS**, the set of registers containing no registers called **NO_REGS**, and a set of registers the user must define as **GENERAL_REGS**. Most GCC ports define **GENERAL_REGS** and **ALL_REGS** to be equivalent.

You should also define the union of two register classes whenever an instruction may need them. For example, if you define one class of registers called address registers and another class of registers called data registers, more than likely you want a class of registers that represents their union. The reason the union of two registers' classes should be defined is to give the register allocator flexibility in choosing registers and also for better register allocation. Let me make all this practical with an example. Say you have a machine that has eight data registers (**d0**-**d7**), address registers (**a0**-**a7**), and floating-point registers. In this case, register classes for data, address, and floating-point registers should be defined. A class that represents the union of data and address registers should also be defined. Finally you may want a class that represents the union of the data and floating-point registers if the data registers can contain floating-point values temporarily. This is how you would represent this information to GCC:

```
enum reg_class {NO_REGS, DATA_REGS,
  ADDR_REGS, DATA_ADDR_REGS, FP_REGS,
  DATA_FP_REGS,  ALL_REGS}
```

You may be wondering how you connect register numbers to the above register classes? To understand the connection, assume that the data registers are numbered **0**-**7**, the address registers are numbered **8**-**15**, and the floating-point registers are numbered **16**-**23**. You then define another macro in this section call **REG_CLASS_CONTENTS**, which defines the bit contents of each register class:

```
#define REG_CLASS_CONTENTS  {0x00000000,
  0x000000ff, 0x0000ff00, 0x0000ffff,
  0x00ff0000, 0x00ff00ff,0x00ffffff}
```

The second initializer in the table indicates that the data registers **d0**-**d7**, with values **0**-**7**, are represented as the value **0x000000ff**. The other values in the table are derived accordingly. The important point to note about this table is that the register allocator uses it when it must allocate a register from a particular class. If the register allocator needs to allocate a floating-point register, it would use the enumeration constant **FP_REG** as an index into the array initialized from **REG_CLASS_CONTENTS**. It would get the value **0x00ff0000**, which represents the set of all floating-point registers. Note, there is a one-to-one correlation between the enumeration **enum reg_class** and the macro **REG_CLASS_CONTENTS**.

The final macro I will cover maps the register class names to letters of the alphabet. These letters are shortcuts for the register class names and are used in the machine description file. For example the 68000 GCC port uses the letter **"a"** to represent address registers, the letter **"d"** to represent data registers, and the letter **"f"** to represent floating-point registers. This would be represented as follows:

```
#define REG_CLASS_FROM_LETTER(c)
  ((c) == 'a' ? ADDR_REGS :
   (c) == 'd' ? DATA_REGS :
   (c) == 'f' ? FP_REG : NO_REGS)
```

Now when the compiler sees the letter **"a"** in the machine description file, as I will describe later, it will know that an address register must be allocated using the procedure I described above.

## Machine Description File

An overview of GCC's use of the machine description file follows. Quoting from *Using and Compiling the GNU Compiler Collection (GCC)*, there are three main conversions that happen in the compiler:

1. The front end reads the source code and builds a parse tree.
2. The parse tree is used to generate an RTL instruction list based on named instruction patterns.
3. The instruction list is matched against the RTL templates to produce assembler code.

Steps 2 and 3 use the machine description file heavily.

When GCC must generate code for an expression in the source language, it looks for a predefined named pattern in the machine description file. This predefined pattern then generates the appropriate RTL IL statements. These predefined patterns are defined using either a **define_insn** or **define_expand** in the machine description file. So, for example, if GCC needs to generate RTL code for a 32-bit on an x86 machine, GCC will look for a named pattern in the machine description file called **addsi3**. It is called **addsi3** because it is an addition instruction, using machine mode **SImode**, and it takes three operands. Remember that **SImode** is 32-bits on the x86, but it may not be the case on another processor. GCC has a predefined set of patterns, such as **addsi3**, which it uses to generate code for source language constructs. These patterns are called "standard names" in the GCC manual. One note about nomenclature, GCC uses the abbreviation **insn** to refer to an RTL instruction.

## Defining Machine Instructions

Listing 1 shows how to define a 32-bit add instruction for a hypothetical processor. There is a lot of information in this pattern, so I will explain it one step at a time. First, the front end uses this pattern when it wants to generate code for a 32-bit (**SImode** in this case) addition. A function call **gen_addsi3** will be created when GCC is built and will be called by the front end to generate the appropriate RTL. In this case, the RTL generated would look like:

```
(set (reg:SI 150) (plus:SI (reg:SI 151) (reg:SI 152)))
```

The numbers **150**-**152** in the register RTL expression are pseudo registers whose numbers I chose at random. Note how **(reg:150)** corresponds to operand **0** and so forth for the other operands.

Second, this pattern is used for matching the RTL generated by the front end or other phases of the compiler to patterns in the machine description file. This is important: any RTL **insn** that is generated by any phase of the compiler must eventually match a legal RTL template in the machine description file. The way matching works in this case is that GCC would look at the entire RTL **insn** and make sure that every operand, operator, and machine mode match. So, in Listing 1, GCC would first check that the entire operation was a set with a plus in **SImode**, with three registers all with mode **SImode**. GCC calls the **register_operand** construct a predicate. The predicate is a C function that checks the validity of the operator or operand passed to it. In this case, a predefined C function called **register_operand** will be called three times (once for each operand of the **plus** operation). Its job will be to insure that its operand is a RTL register. If it is an RTL register, the function returns **1**. If the test fails, it will return zero. Additionally, by the time register allocation is finished, GCC will have substituted actual hardware registers for the pseudo register above. So the final RTL **insn** looks like:

```
(set (reg:SI 0) (plus:SI
     (reg:SI 1) (reg:SI 2)))
```

The registers I have chosen correspond to registers **d0**-**d2** as discussed above. The one-letter characters following the predicate are called the constraint. The constraint is used to "squeeze" the already legal predicate (in this case a register) into a particular hardware resource (in this case a specific set of registers). The constraint **r** says to use any general-purpose register. This corresponds to the register class **GENERAL_REGS** as discussed above. From this set of registers, GCC is free to pick any register it chooses.

The final component of this template is the assembly language output string. This string is used once the pattern has been matched and the operands substituted as shown in the code above. The assembly language pattern says to output an instruction substituting the operands into the output string. So in the example above, you would end up with the assembly language instruction **"add d0, d1, d2"**. The construct **%n** says to substitute the actual operand from the template into the output string.

The machine description file should contain template patterns for all the common operations your microprocessor supports, including addition, subtraction, compare, jump, call, etc. In fact, the more patterns you add to the machine description file, the better the final code GCC can generate. This is because, if a pattern that GCC wishes to generate code for is missing, the compiler assumes the instruction does not exist on the processor. The compiler will then generate a function call to a predefined name. This is not what you want if a native instruction exists on your microprocessor. This default behavior is fine for operations like floating point if your chip only supports fixed-point instructions. As a final note, if GCC has to create function calls for operations that do not exist, it is up to you as the compiler developer to provide the missing functionality as a library routine. The official GCC comes with a set of common missing functionality, such as IEEE floating-point math.

## Conclusion

The GCC is a highly retargetable compiler that implements state-of-the art optimizations. By describing the architecture and instruction set of your target processor, a compiler can be developed in a relatively short period of time.

## Notes and References

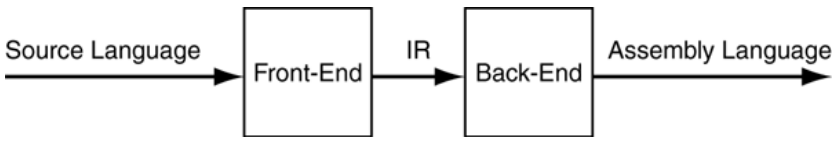
[1] The GNU General Public License, <www.fsf.org/licenses/licenses.html>.

[2] The source code for GCC may be downloaded from <http://gcc.gnu.org>.

[3] *Using and Compiling the GNU Compiler Collection (GCC)*, <http://gcc.gnu.org/onlinedocs)>.

***Michael Collison*** *has been involved in compiler development since 1988. He has developed several C compilers based on the GNU C compiler for several processor architectures. He is official GNU C compiler developer and maintainer for the Lucent DSP1600 DSP chip. This port can be found in the official GNU C compiler distribution.*

**Figure 1: Compiler overview**




**Listing 1: Defining a 32-bit add instruction for a hypothetical processor**

```
(define_insn "addsi3" [(set (match_operand:SI 0 "register_operand" "=r"
                                                (plus:SI
(match_operand:SI 1 "register_operand" "%r")

(match_operand:SI 2 "register_operand" "r")))]
""
"add %0, %1, %2")
— End of Listing —
```

**Table 1: Commonly used machine modes in GCC**

| | |
|---|---|
| **QImode** | **Quarter integer mode. Represents a single byte treated as an integer.** |
| **HImode** | **Half integer mode. Represents a two-byte integer.** |
| **SImode** | **Single integer mode. Represents a four-byte integer.** |
| **SFmode** | **Single floating-point mode. Represents a four-byte floating number.** |

**DFmode Double floating-point mode. Represents an eight-byte floating number.**

**Table 2: Constant RTL expression types**

| | |
|---|---|
| **(const_int i)** | **Represents the integer value of i.** |
| **(symbol_ref:m name)** | **Represents the value of a label for data. name is a string that describes the name of the label.** |
| **(label_ref label)** | **Represents the value of a label for code. label indentifies where the branch or call instruction will transfer control to.** |

**Table 3: Register and memory RTL types**

| | |
|---|---|
| **(reg:m n)** | Represents a physical or pseudo register with mode **m** and register number **n**. |
| **(mem:m addr)** | Represents a memory address with mode **m** and **addr** representing a target-specific address expression. |
| **(pc)** | Represents the program counter. Typically used in jump and call expressions. |

**Table 4: Arithmetic RTL expression types**

| | |
|---|---|
| **(plus:m x y)** | Represents the sum of the values **x** and **y** carried out in machine mode **m**. |
| **(mult:m x y)** | Represents the signed product of **x** and **y** in machine mode **m**. |
| **(sign_extend:m x)** | Represents the result of sign extend the value **x** to machine mode **m**. Mode **m** must be an integer mode and **x** must have an integer mode narrower than mode **m**. |