9

Inline Assembly Code

TODAY, FEW PROGRAMMERS USE ASSEMBLY LANGUAGE. Higher-level languages such as C and C++ run on nearly all architectures and yield higher productivity when writing and maintaining code. For occasions when programmers need to use assembly instructions in their programs, the GNU Compiler Collection permits programmers to add architecture-dependent assembly language instructions to their programs.

GCC's inline assembly statements should not be used indiscriminately. Assembly language instructions are architecture-dependent, so, for example, programs using x86 instructions cannot be compiled on PowerPC computers. To use them, you'll require a facility in the assembly language for your architecture. However, inline assembly statements permit you to access hardware directly and can also yield faster code.

An `asm` instruction allows you to insert assembly instructions into C and C++ programs. For example, this instruction

```
asm ("fsin" : "=t" (answer) : "0" (angle));
```

is an x86-specific way of coding this C statement:[1]

```
answer = sin (angle);
```

1. The expression `sin (angle)` is usually implemented as a function call into the math library, but if you specify the `-O1` or higher optimization flag, GCC is smart enough to replace the function call with a single `fsin` assembly instruction.

Observe that unlike ordinary assembly code instructions, `asm` statements permit you to specify input and output operands using C syntax.

To read more about the x86 instruction set, which we will use in this chapter, see `http://developer.intel.com/design/pentiumii/manuals/` and `http://www.x86-64.org/documentation`.

9.1 When to Use Assembly Code

Although `asm` statements can be abused, they allow your programs to access the computer hardware directly, and they can produce programs that execute quickly. You can use them when writing operating system code that directly needs to interact with hardware. For example, `/usr/include/asm/io.h` contains assembly instructions to access input/output ports directly. The Linux source code file `/usr/src/linux/arch/i386/kernel/process.s` provides another example, using `hlt` in idle loop code. See other Linux source code files in `/usr/src/linux/arch/` and `/usr/src/linux/drivers/`.

Assembly instructions can also speed the innermost loop of computer programs. For example, if the majority of a program's running time is computing the sine and cosine of the same angles, this innermost loop could be recoded using the `fsincos` x86 instruction.[2] See, for example, `/usr/include/bits/mathinline.h`, which wraps up into macros some inline assembly sequences that speed transcendental function computation.

You should use inline assembly to speed up code only as a last resort. Current compilers are quite sophisticated and know a lot about the details of the processors for which they generate code. Therefore, compilers can often choose code sequences that may seem unintuitive or roundabout but that actually execute faster than other instruction sequences. Unless you understand the instruction set and scheduling attributes of your target processor very well, you're probably better off letting the compiler's optimizers generate assembly code for you for most operations.

Occasionally, one or two assembly instructions can replace several lines of higher-level language code. For example, determining the position of the most significant nonzero bit of a nonzero integer using the C programming languages requires a loop or floating-point computations. Many architectures, including the x86, have a single assembly instruction (`bsr`) to compute this bit position. We'll demonstrate the use of one of these in Section 9.4, "Example."

2. Algorithmic or data structure changes may be more effective in reducing a program's running time than using assembly instructions.

9.2 Simple Inline Assembly

Here we introduce the syntax of `asm` assembler instructions with an x86 example to shift a value 8 bits to the right:

```
asm ("shrl $8, %0" : "=r" (answer) : "r" (operand) : "cc");
```

The keyword `asm` is followed by a parenthetic expression consisting of sections separated by colons. The first section contains an assembler instruction and its operands. In this example, `shrl` right-shifts the bits in its first operand. Its first operand is represented by `%0`. Its second operand is the immediate constant `$8`.

The second section specifies the outputs. The instruction's one output will be placed in the C variable `answer`, which must be an lvalue. The string `"=r"` contains an equals sign indicating an output operand and an `r` indicating that `answer` is stored in a register.

The third section specifies the inputs. The C variable `operand` specifies the value to shift. The string `"r"` indicates that it is stored in a register but omits an equals sign because it is an input operand, not an output operand.

The fourth section indicates that the instruction changes the value in the condition code `cc` register.

9.2.1 Converting an *asm* to Assembly Instructions

GCC's treatment of `asm` statements is very simple. It produces assembly instructions to deal with the `asm`'s operands, and it replaces the `asm` statement with the instruction that you specify. It does not analyze the instruction in any way.

For example, GCC converts this program fragment

```
double foo, bar;
asm ("mycool_asm %1, %0" : "=r" (bar) : "r" (foo));
```

to these x86 assembly instructions:

```
        movl -8(%ebp),%edx
        movl -4(%ebp),%ecx
#APP
        mycool_asm %edx, %edx
#NO_APP
        movl %edx,-16(%ebp)
        movl %ecx,-12(%ebp)
```

Remember that `foo` and `bar` each require two words of stack storage on a 32-bit x86 architecture. The register `ebp` points to data on the stack.

The first two instructions copy `foo` into registers `EDX` and `ECX` on which `mycool_asm` operates. The compiler decides to use the same registers to store the answer, which is copied into `bar` by the final two instructions. It chooses appropriate registers, even reusing the same registers, and copies operands to and from the proper locations automatically.

9.3 Extended Assembly Syntax

In the subsections that follow, we describe the syntax rules for `asm` statements. Their sections are separated by colons.

We will refer to this illustrative `asm` statement, which computes the Boolean expression `x > y`:

```
asm ("fucomip %%st(1), %%st; seta %%al" :
     "=a" (result) : "u" (y), "t" (x) : "cc", "st");
```

First, `fucomip` compares its two operands `x` and `y`, and stores values indicating the result into the condition code register. Then `seta` converts these values into a 0 or 1 result.

9.3.1 Assembler Instructions

The first section contains the assembler instructions, enclosed in quotation marks. The example `asm` contains two assembly instructions, `fucomip` and `seta`, separated by semicolons. If the assembler does not permit semicolons, use newline characters (`\n`) to separate instructions.

The compiler ignores the contents of this first section, except that one level of percentage signs is removed, so `%%` changes to `%`. The meaning of `%%st(1)` and other such terms is architecture-dependent.

GCC will complain if you specify the `-traditional` option or the `-ansi` option when compiling a program containing `asm` statements. To avoid producing these errors, such as in header files, use the alternative keyword `__asm__`.

9.3.2 Outputs

The second section specifies the instructions' output operands using C syntax. Each operand is specified by an operand constraint string followed by a C expression in parentheses. For output operands, which must be lvalues, the constraint string should begin with an equals sign. The compiler checks that the C expression for each output operand is in fact an lvalue.

Letters specifying registers for a particular architecture can be found in the GCC source code, in the `REG_CLASS_FROM_LETTER` macro. For example, the `gcc/config/i386/i386.h` configuration file in GCC lists the register letters for the x86 architecture.[3] Table 9.1 summarizes these.

3. You'll need to have some familiarity with GCC's internals to make sense of this file.

Table 9.1 **Register Letters for the Intel x86 Architecture**

Register Letter	Registers That GCC May Use
R	General register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	General register for data (EAX, EBX, ECX, EDX)
f	Floating-point register
t	Top floating-point register
u	Second-from-top floating-point register
a	EAX register
b	EBX register
c	ECX register
d	EDX register
x	SSE register (Streaming SIMD Extension register)
y	MMX multimedia registers
A	An 8-byte value formed from EAX and EDX
D	Destination pointer for string operations (EDI)
S	Source pointer for string operations (ESI)

Multiple operands in an `asm` statement, each specified by a constraint string and a C expression, are separated by commas, as illustrated in the example `asm`'s input section. You may specify up to 10 operands, denoted `%0`, `%1`, …, `%9`, in the output and input sections. If there are no output operands but there are input operands or clobbered registers, leave the output section empty or mark it with a comment like `/* no outputs */`.

9.3.3 Inputs

The third section specifies the input operands for the assembler instructions. The constraint string for an input operand should not have an equals sign, which indicates an lvalue. Otherwise, an input operand's syntax is the same as for output operands.

To indicate that a register is both read from and written to in the same `asm`, use an input constraint string of the output operand's number. For example, to indicate that an input register is the same as the first output register number, use `0`. Output operands are numbered left to right, starting with 0. Merely specifying the same C expression for an output operand and an input operand does not guarantee that the two values will be placed in the same register.

This input section can be omitted if there are no input operands and the subsequent clobber section is empty.

9.3.4 Clobbers

If an instruction modifies the values of one or more registers as a side effect, specify the clobbered registers in the `asm`'s fourth section. For example, the `fucomip` instruction modifies the condition code register, which is denoted `cc`. Separate strings representing clobbered registers with commas. If the instruction can modify an arbitrary memory location, specify `memory`. Using the clobber information, the compiler determines which values must be reloaded after the `asm` executes. If you don't specify this information correctly, GCC may assume incorrectly that registers still contain values that have, in fact, been overwritten, which will affect your program's correctness.

9.4 Example

The x86 architecture includes instructions that determine the positions of the least significant set bit and the most significant set bit in a word. The processor can execute these instructions quite efficiently. In contrast, implementing the same operation in C requires a loop and a bit shift.

For example, the `bsrl` assembly instruction computes the position of the most significant bit set in its first operand, and places the bit position (counting from 0, the least significant bit) into its second operand. To place the bit position for `number` into `position`, we could use this `asm` statement:

```
asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
```

One way you could implement the same operation in C is using this loop:

```
long i;
for (i = (number >> 1), position = 0; i != 0; ++position)
  i >>= 1;
```

To test the relative speeds of these two versions, we'll place them in a loop that computes the bit positions for a large number of values. Listing 9.1 does this using the C loop implementation. The program loops over integers, from 1 up to the value specified on the command line. For each value of `number`, it computes the most significant bit that is set. Listing 9.2 does the same thing using the inline assembly instruction. Note that in both versions, we assign the computed bit position to a volatile variable `result`. This is to coerce the compiler's optimizer so that it does not eliminate the entire bit position computation; if the result is not used or stored in memory, the optimizer eliminates the computation as "dead code."

Listing 9.1 **(*bit-pos-loop.c*) Find Bit Position Using a Loop**

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
  long max = atoi (argv[1]);
  long number;
```

```
  long i;
  unsigned position;
  volatile unsigned result;

  /* Repeat the operation for a large number of values.  */
  for (number = 1; number <= max; ++number) {
    /* Repeatedly shift the number to the right, until the result is
       zero.  Keep count of the number of shifts this requires.  */
    for (i = (number >> 1), position = 0; i != 0; ++position)
      i >>= 1;
    /* The position of the most significant set bit is the number of
       shifts we needed after the first one.  */
    result = position;
  }

  return 0;
}
```

Listing 9.2 (*bit-pos-asm.c*) **Find Bit Position Using *bsrl***

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv[])
{
  long max = atoi (argv[1]);
  long number;
  unsigned position;
  volatile unsigned result;

  /* Repeat the operation for a large number of values.  */
  for (number = 1; number <= max; ++number) {
    /* Compute the position of the most significant set bit using the
       bsrl assembly instruction.  */
    asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
    result = position;
  }

  return 0;
}
```

We'll compile both versions with full optimization:

```
% cc -O2 -o bit-pos-loop bit-pos-loop.c
% cc -O2 -o bit-pos-asm bit-pos-asm.c
```

Now let's run each using the `time` command to measure execution time. We'll specify a large value as the command-line argument, to make sure that each version takes at least a few seconds to run.

```
% time ./bit-pos-loop 250000000
19.51user 0.00system 0:20.40elapsed 95%CPU (0avgtext+0avgdata
0maxresident)k0inputs+0outputs (73major+11minor)pagefaults 0swaps
% time ./bit-pos-asm 250000000
3.19user 0.00system 0:03.32elapsed 95%CPU (0avgtext+0avgdata
0maxresident)k0inputs+0outputs (73major+11minor)pagefaults 0swaps
```

Notice that the version that uses inline assembly executes a great deal faster (your results for this example may vary).

9.5 Optimization Issues

GCC's optimizer attempts to rearrange and rewrite programs' code to minimize execution time even in the presence of `asm` expressions. If the optimizer determines that an `asm`'s output values are not used, the instruction will be omitted unless the keyword `volatile` occurs between `asm` and its arguments. (As a special case, GCC will not move an `asm` without any output operands outside a loop.) Any `asm` can be moved in ways that are difficult to predict, even across jumps. The only way to guarantee a particular assembly instruction ordering is to include all the instructions in the same `asm`.

Using `asm`s can restrict the optimizer's effectiveness because the compiler does not know the asms' semantics. GCC is forced to make conservative guesses that may prevent some optimizations. *Caveat emptor!*

9.6 Maintenance and Portability Issues

If you decide to use nonportable, architecture-dependent `asm` statements, encapsulating these statements within macros or functions can aid in maintenance and porting. Placing all these macros in one file and documenting them will ease porting to a different architecture, something that occurs with surprising frequency even for "throwaway" programs. Thus, the programmer will need to rewrite only one file for the different architecture.

For example, most `asm` statements in the Linux source code are grouped into `/usr/src/linux/include/asm` and `/usr/src/linux/include/asm-i386` header files, and `/usr/src/linux/arch/i386/` and `/usr/src/linux/drivers/` source files.