IBM

developerWorks®

Inline assembly for x86 in Linux

Putting the pieces together

Bharata Rao (rbharata@in.ibm.com)
IBM Linux Technology Center, IBM Software Labs, India

01 March 2001

Bharata B. Rao offers a guide to the overall use and structure of inline assembly for x86 on the Linux platform. He covers the basics of inline assembly and its various usages, gives some basic inline assembly coding guidelines, and explains the instances of inline assembly code in the Linux kernel.

If you're a Linux kernel developer, you probably find yourself coding highly architecture-dependent functions or optimizing a code path pretty often. And you probably do this by inserting assembly language instructions into the middle of C statements (a method otherwise known as inline assembly). Let's take a look at the specific usage of inline assembly in Linux. (We'll limit our discussion to the IA32 assembly.)

GNU assembler syntax in brief

Let's first look at the basic assembler syntax used in Linux. GCC, the GNU C Compiler for Linux, uses AT&T assembly syntax. Some of the basic rules of this syntax are listed below. (The list is by no means complete; I've included only those rules pertinent to inline assembly.)

Register naming

Register names are prefixed by %. That is, if eax has to be used, it should be used as %eax.

Source and destination ordering

In any instruction, source comes first and destination follows. This differs from Intel syntax, where source comes after destination.

```
    mov %eax, %ebx, transfers the contents of eax to ebx.
```

Size of operand

The instructions are suffixed by b, w, or l, depending on whether the operand is a byte, word, or long. This is not mandatory; GCC tries provide the appropriate suffix by reading the operands. But specifying the suffixes manually improves the code readability and eliminates the possibility of the compilers guessing incorrectly.

```
   movb %al, %bl -- Byte move
movw %ax, %bx -- Word move
movl %eax, %ebx -- Longword move
```

Immediate operand

An immediate operand is specified by using $.

```
   movl $0xffff, %eax -- will move the value of 0xffff into eax register.
```

Indirect memory reference

Any indirect references to memory are done by using ( ).

```
movb (%esi), %al -- will transfer the byte in the memory
                    pointed by esi into al  register
```

Inline assembly

GCC provides the special construct "asm" for inline assembly, which has the following format:

```
asm ( assembler template
    : output operands     (optional)
    : input operands     (optional)
    : list of clobbered registers     (optional)
    );
```

In this example, the assembler template consists of assembly instructions. The input operands are the C expressions that serve as input operands to the instructions. The output operands are the C expressions on which the output of the assembly instructions will be performed.

```
   asm ("movl %%cr3, %0\n" :"=r"(cr3val));
```

```
 a %eax
 b  %ebx
 c  %ecx
 d  %edx
 S %esi
 D %edi
```

Memory operand constraint(m)

When the operands are in the memory, any operations performed on them will occur directly in the memory location, as opposed to register constraints, which first store the value in a register to be modified and then write it back to the memory location. But register constraints are usually used only when they are absolutely necessary for an instruction or they significantly speed up the process. Memory constraints can be used most efficiently in cases where a C variable needs to be updated inside "asm" and you really don't want to use a register to hold its value. For example, the value of idtr is stored in the memory location loc:

```
    ("sidt %0\n" : :"m"(loc));
```

Matching(Digit) constraints

In some cases, a single variable may serve as both the input and the output operand. Such cases may be specified in "asm" by using matching constraints.

```
    asm ("incl %0" :"=a"(var):"0"(var));
```

In our example for matching constraints, the register %eax is used as both the input and the output variable. var input is read to %eax and updated %eax is stored in var again after increment. "0" here specifies the same constraint as the 0th output variable. That is, it specifies that the output instance of var should be stored in %eax only. This constraint can be used:

- In cases where input is read from a variable or the variable is modified and modification is written back to the same variable
- In cases where separate instances of input and output operands are not necessary

The most important effect of using matching restraints is that they lead to the efficient use of available registers.

Examples of common inline assembly usage

The following examples illustrate usage through different operand constraints. There are too many constraints to give examples for each one, but these are the most frequently used constraint types.

"asm" and the register constraint "r"
Let's first take a look at "asm" with the register constraint 'r'. Our example shows how GCC allocates registers, and how it updates the value of output variables.

```
int main(void)
{
 int x = 10, y;

 asm ("movl %1, %%eax;
      "movl %%eax, %0;"
  :"=r"(y) /* y is output operand */
  :"r"(x)  /* x is input operand */
  :"%eax"); /* %eax is clobbered register */
}
```

In this example, the value of x is copied to y inside "asm". x and y are passed to "asm" by being stored in registers. The assembly code generated for this example looks like this:

```
    main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl $10,-4(%ebp)
        movl -4(%ebp),%edx /* x=10 is stored in %edx */
#APP /* asm starts here */
        movl %edx, %eax  /* x is moved to %eax */
        movl %eax, %edx  /* y is allocated in edx and updated */

#NO_APP /* asm ends here */
        movl %edx,-8(%ebp) /* value of y in stack is updated with
       the value in %edx */
```

GCC is free here to allocate any register when the "r" constraint is used. In our example it chose %edx for storing x. After reading the value of x in %edx, it allocated the same register for y.

Since y is specified in the output operand section, the updated value in %edx is stored in -8(%ebp), the location of y on stack. If y were specified in the input section, the value of y on stack would not be updated, even though it does get updated in the temporary register storage of y(%edx).

And since %eax is specified in the clobbered list, GCC doesn't use it anywhere else to store data.

Both input x and output y were allocated in the same %edx register, assuming that inputs are consumed before outputs are produced. Note that if you have a lot of instructions, this may not be the case. To make sure that input and output are allocated in different registers, we can specify the & constraint modifier. Here is our example with the constraint modifier added.

```
 int main(void)
{
 int x = 10, y;

 asm ("movl %1, %%eax;
      "movl %%eax, %0;"
  :"=&r"(y) /* y is output operand, note the
       & constraint modifier. */
  :"r"(x)  /* x is input operand */
  :"%eax"); /* %eax is clobbered register */
}
```

And here is the assembly code generated for this example, from which it is evident that x and y have been stored in different registers across "asm".

```
    main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl $10,-4(%ebp)
        movl -4(%ebp),%ecx /* x, the input is in %ecx */
#APP
 movl %ecx, %eax
 movl %eax, %edx  /* y, the output is in %edx */

#NO_APP
        movl %edx,-8(%ebp)
```

Use of specific register constraints

Now let's take a look at how to specify individual registers as constraints for the operands. In the following example, the cpuid instruction takes the input in the %eax register and gives output in four registers: %eax, %ebx, %ecx, %edx. The input to cpuid (the variable "op") is passed to "asm" in the eax register, as cpuid expects it to. The a, b, c, and d constraints are used in the output to collect the values in the four registers, respectively.

```
    asm ("cpuid"
                : "=a" (_eax),
                  "=b" (_ebx),
                  "=c" (_ecx),
                  "=d" (_edx)
                : "a" (op));
```

And below you can see the generated assembly code for this (assuming the _eax, _ebx, etc.... variables are stored on stack):

```
    movl -20(%ebp),%eax /* store 'op' in %eax -- input */
#APP
        cpuid
#NO_APP
        movl %eax,-4(%ebp) /* store %eax in _eax -- output */
        movl %ebx,-8(%ebp) /* store other registers in
        movl %ecx,-12(%ebp)    respective output variables */
        movl %edx,-16(%ebp)
```

The strcpy function can be implemented using the "S" and "D" constraints in the following manner:

```
    asm ("cld\n
       rep\n
       movsb"
       : /* no input */
       :"S"(src), "D"(dst), "c"(count));
```

The source pointer src is put into %esi by using the "S" constraint, and the destination pointer dst is put into %edi using the "D" constraint. The count value is put into %ecx as it is needed by rep prefix.

And here you can see another constraint that uses the two registers %eax and %edx to combine two 32-bit values and generate a 64-bit value:

```
#define rdtscll(val) \
     __asm__ __volatile__ ("rdtsc" : "=A" (val))

The generated assembly looks like this (if val has a 64 bit memory space).

#APP
        rdtsc
#NO_APP
        movl %eax,-8(%ebp) /* As a result of A constraint
        movl %edx,-4(%ebp)    %eax and %edx serve as outputs */

Note here that the values in %edx:%eax serve as 64 bit output.
```

Using matching constraints

Here you can see the code for the system call, with four parameters:

```
#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
          "d" ((long)(arg3)),"S" ((long)(arg4))); \
__syscall_return(type,__res); \
}
```

In the above example, four arguments to the system call are put into %ebx, %ecx, %edx, and %esi by using the constraints b, c, d, and S. Note that the "=a" constraint is used in the output so that the return value of the system call, which is in %eax, is put into the variable __res. By using the matching constraint "0" as the first operand constraint in the input section, the syscall number __NR_##name is put into %eax and serves as the input to the system call. Thus %eax serves here as both input and output register. No separate registers are used for this purpose. Note also that the input (syscall number) is consumed (used) before the output (the return value of syscall) is produced.

Use of memory operand constraint

Consider the following atomic decrement operation:

```
    __asm__ __volatile__(
                "lock; decl %0"
                :"=m" (counter)
                :"m" (counter));
```

The generated assembly for this would look something like this:

```
    #APP
 lock
 decl -24(%ebp) /* counter is modified on its memory location */
#NO_APP.
```

You might think of using the register constraint here for the counter. If you do, the value of the counter must first be copied on to a register, decremented, and then updated to its memory. But then you lose the whole purpose of locking and atomicity, which clearly shows the necessity of using the memory constraint.

Using clobbered registers

Consider an elementary implementation of memory copy.

```
 asm ("movl $count, %%ecx;
       up: lodsl;
       stosl;
       loop up;"
  :    /* no output */
  :"S"(src), "D"(dst) /* input */
  :"%ecx", "%eax" ); /* clobbered list */
```

While lodsl modifies %eax, the lodsl and stosl instructions use it implicitly. And the %ecx register explicitly loads the count. But GCC won't know this unless we inform it, which is exactly what we do by including %eax and %ecx in the clobbered register set. Unless this is done, GCC assumes that %eax and %ecx are free, and it may decide to use them for storing other data. Note here that %esi and %edi are used by "asm", and are not in the clobbered list. This is because it has been declared that "asm" will use them in the input operand list. The bottom line here is that if a register is used inside "asm" (implicitly or explicitly), and it is not present in either the input or output operand list, you must list it as a clobbered register.

Conclusion

On the whole, inline assembly is huge and provides a lot of features that we did not even touch on here. But with a basic grasp of the material in this article, you should be able to start coding inline assembly on your own.

Resources

- Refer to the Using and Porting the GNU Compiler Collection (GCC) manual.
- Refer to the GNU Assembler (GAS) manual.
- Check out Brennan's Guide to Inline Assembly.

About the author

Bharata Rao

Bharata B. Rao has a bachelor of Engineering in Electronics and Communication from Mysore University, India. He has been working for IBM Global Services, India since 1999. He is a member of the IBM Linux Technology Center, where he concentrates primarily on Linux RAS (Reliability, Availability, and Serviceability). Other areas of interest are operating system internals and processor architecture. He can be reached at rbharata@in.ibm.com.