

More on Intel Arithmetic and Logic Operations

Norm Matloff

February 16, 2006

©2005-2007, N.S. Matloff

Contents

1	Instructions for Multiplication and Division	2
1.1	Multiplication	2
1.1.1	The IMUL Instruction	2
1.1.2	Issues of Sign	2
1.2	Division	2
1.2.1	The IDIV Instruction	2
1.2.2	Issues of Sign	3
1.3	Example	3
2	More on Carry and Overflow, and More Jump Instructions	4
3	Logical Instructions	6
4	Floating-Point	8


```
idivl src
```

We place the dividend in EDX:EAX and the divisor in **src**, again with the latter being specified in register addressing mode. The quotient then comes out in EAX, and the remainder in EDX. If the quotient is too big for 32-bit storage, then an **exception**, i.e. an execution error will occur.

1.2.2 Issues of Sign

As noted for multiplication above, you must be careful with 64-bit quantities. Even if your dividend fits into 32 bits, it will be treated as a 64-bit number, with the high bits being in EDX—whether you put them there or not.

So, if your dividend is nonnegative, make sure to put 0 in EDX; otherwise put -1 there.

A compact and fast way to do this is to use the CWD instruction.

1.3 Example

Here is an example of IMUL/IDIV:

```
1 .text
2 .globl _start
3 _start:
4     movl $2, %ecx
5     movl $-1, %eax
6     imull %ecx
7     idivl %ecx
8 done: movl %ebx, %ebx
```

Let's use GDB to explore the behavior of these instructions:

```
1 gdb) r
2 Starting program: /www/matloff/public_html/50/PLN/a.out
3
4 Breakpoint 1, _start () at g.s:4
5 4          imull %ecx
6 Current language: auto; currently asm
7 (gdb) p/x $eax
8 $1 = 0xffffffff
9 (gdb) p/x $edx
10 $2 = 0x0
11 (gdb) si
12 8          idivl %ecx
13 (gdb) p/x $eax
14 $3 = 0xffffffffe
15 (gdb) p/x $edx
16 $4 = 0xfffffffff
17 (gdb) si
18 9          done: movl %ebx, %ebx
19 (gdb) p/x $eax
20 $5 = 0xfffffffff
21 (gdb) p/x $edx
22 $6 = 0x0
```

We started with the 32-bit version of -1 in EAX, and 0 in EDX. Multiplying by 2 gives us -2, but IMUL puts its product in 64-bit form in the register pair EDX:EAX. GDB verified this for us. Then we divided by 2, getting -1, but since IDIV puts its result in 32-bit form in the single register EAX, EDX becomes 0.

Or, if we want to view the numbers as unsigned—remember, the nice thing about 2s complement is that we have our choice of interpretation—then our multiplying by 2 changed $2^{32} - 1$ (a number fitting in 32 bits) to $2^{33} - 2$ (a 64-bit number).

2 More on Carry and Overflow, and More Jump Instructions

In our earlier unit, we saw how to use instructions like JS, JNZ, etc. to do “if-then-else” tests. This works for simple cases with small or moderate-sized numbers. Unfortunately, it is more complicated than this, though, due to problems with the difference between signed and unsigned numbers. To see why, suppose for convenience that we are working with 1-byte quantities (other than convenience, there is nothing special about using only one byte here), and consider what happens with

```
cmpb %a1, %b1
```

when $c(AL) = 0x50$ and $c(BL) = 0xfb$. Viewed as signed numbers, $c(BL)$ is smaller, since $80 > -5$, but viewed as unsigned numbers, $c(AL)$ is smaller, since $80 < 251$.

Recall that the hardware doesn’t know whether we are considering our numbers to be signed or unsigned. So, for example, there is nothing in the hardware for the CMP instruction to distinguish between the signed and unsigned cases, nor is there such a capability in the ADD and SUB instructions.² However, the hardware does store relevant information in the Carry and Overflow flags, and this will allow us as programmers to handle the signed and unsigned cases differently on our own, as we will see below.

Before continuing, let’s review at what the hardware does when adding two numbers. It will in any addition simply add the two numbers bit by bit sequentially from right to left, with carries, in grade-school fashion. The hardware is basically treating the numbers as unsigned, but remember, that’s OK even if we are considering the numbers to be signed, because the nature of 2s complement storage makes it all work out.

When performing an ADD instruction, the hardware will set or clear the Carry flag (CF), according to whether a carry does occur out of the most significant bit. As noted in our earlier unit, we can programmatically check this by using the JC or JNC instructions. Similarly, the hardware will set or clear the Overflow flag (OF), under certain conditions which we will discuss shortly; our program can check this flag via the JO and JNO instructions.

The term **overflow**—as opposed to the Overflow flag—informally means that the result of an operation is too large to fit into the number of bits allocated to it. The definition of “too large” in turn depends on whether we are treating the numbers as signed or unsigned.

Again taking the 8-bit case as our example, suppose $c(AL) = 0x50$ and $c(BL) = 0x50$, i.e. both registers contain decimal 80. Their sum, 160, does fit into 8 bits if we are considering our numbers to be unsigned, but if we are thinking of them as signed, then “160” is really -96. So, in this example an overflow will not occur if the numbers are considered unsigned, but will occur if they are considered signed.

Clearly, if we are considering our numbers to be unsigned, overflow is indicated by the Carry Flag. But what about the signed case? The Carry Flag will not tell us (at least not directly) whether overflow has occurred. In other words, we’ll need another flag to check overflow in the signed case, and that’s the purpose of the Overflow Flag.


²The MIPS chip, by contrast, does have, for instance, both ADD and ADDU instructions for addition. ADD causes an **exception**, i.e. execution error, upon overflow, while ADDU does nothing to report the problem. C/C++ compilers will use ADDU.

So, how did the Intel engineers design the circuitry which manages the Overflow Flag? Well, let's first note that there won't be an overflow problem if we are adding a nonnegative number to a negative one. The nonnegative number will be in the range 0 to +127, while the negative one will be between -128 and -1. Then the sum must necessarily be between -128 and +127, thus no overflow.

But a problem can occur if we are adding two positive numbers. They are basically 7 bits each (not counting the sign bit), so overflow will occur if they have an 8-bit sum. In such a case, there will be a carry out of the 7th bit, thus placing a 1 in the 8th bit—thus producing a negative number if we are considering everything signed. If you step through this reasoning in the case of adding two negative numbers, you'll see the problem arises if their sum is positive. In other words:

In signed addition, overflow occurs if and only if the sum of two positive numbers is negative or the sum of two negative numbers is positive.

Knowing this, the developers of the Intel hardware designed the circuitry so that the Overflow flag would be set if it sees that there is a sign change of the nature described above, and clears the flag otherwise.

Note once again that the hardware doesn't know whether we are considering our numbers to be signed or unsigned. The developers of the hardware assumed that most people's typical usage would be signed, and thus they designed the Overflow flag accordingly. If we really are treating our numbers as unsigned, then we would have our program check the Overflow flag. But if we are treating them as unsigned, we would check the Carry flag. 

What about subtraction? Recall that it is implemented via addition, i.e. $x-y$ is effected by adding the 2s complement representation of $-y$ to x , so we are really back to the addition case. But if we are using subtraction to set up a conditional jump—typically by using a CMP instruction—we need to worry about the sign flag too, as follows.

- If the Overflow flag is 0, we have nothing to worry about, and can simply look at the Sign flag to determine whether the subtraction resulted in a negative number.
- If the Overflow is 1, things are a little less simple. Recall that when overflow occurs in an addition (which, recall, is performed for the subtraction process), the sign of the sum is opposite to that of the two addends. In other words, an overflow addition changes signs. So, if the subtraction had resulted in a negative difference, which ordinarily would set the Sign flag, the overflow will make the Sign flag 0 instead. Conversely, if the difference were not negative, the overflow will make the Sign flag 1, falsely making the difference look negative.

From these considerations, we see that:

In a subtraction of signed numbers $x-y$, y will be the larger of the two if and only if the $OF \neq SF$.

For this reason, the developers of the Intel chip including the JL (“jump if less than”) which jumps if $OF \neq SF$. This is then the safe way to do less-than comparisons on signed numbers; JS is not safe unless you know your numbers are small enough that overflow won't occur. There are also others. Here is an incomplete list (here “dst” and “src” refer to the destination and source done, for example, in an immediately-previous CMP):

mnemonic	action	flags checked
JL	jump if dst < src	SF ≠ OF
JLE	jump if dst ≤ src	SF = 1 or SF ≠ OF
JG	jump if dst > src	ZF = 0 and SF = OF
JGE	jump if dst ≥ src	SF = OF

There are corresponding instructions for the unsigned case: JB (“jump if below”), JBE (“jump if below or equal”), JA (“jump if above”), and JAE (“jump if above or equal”),

For **imull**, if the product cannot fit into 32 bits and the high bits go into EDX, both CF and OF are set. (SF and ZF are undefined.) With **idivl**, all of these flags are undefined.

3 Logical Instructions

This section will illustrate one of the most important tools available to the assembly language programmer—data operations at the bit level. Keep these in mind in all your assembly language programming.

In our introductory unit on Linux assembly language, we saw how to access 16-bit and 8-bit operands. But what can we do in smaller cases, for example 1-bit operands? The answer is that we cannot do this in a manner directly analogous to the 16-bit and 8-bit cases. The reason for this is that although individual bytes have addresses, individual bits do not. Thus for example there is no 1-bit MOV instruction comparable to **movl** and **movb**.

However, we can access individual bits, or groups of bits, in an indirect manner by using the instructions introduced in this section.

A logical AND operation does the following: At each bit position in the two operands, the result is 1 if both operand bits are 1, and 0 otherwise.

For example,

```

      1011
AND   1101
-----
      1001

```

In the leftmost bit position of the two operands here, we see two 1s, so there is a 1 in the result. In the second-to-the left position, there is a 0 in the first operand and 1 in the second, so there is a 0 in the result, etc.

The **andl** (“AND long”) instruction performs an AND operation on the source and destination, placing the result back in the destination. It is important to note that that means that for every 1 bit in the source operand, every 0 in the source will cause the corresponding bit in the destination to become 0. In other words, **andl** is used to place 0s in desired spots within the destination while keeping the other destination bits unchanged.

Suppose for instance that we wish to put a 0 in bit position 1 (second from the right) in EAX, but keep all the other bits in EAX unchanged. We would use the instruction

```
andl $0xffffffff, %eax
```

because 0xffffffff has a 0 at bit position 2 and all 1s elsewhere.

Often an AND operation is used as a **mask**, which is a mechanism by which we can test for certain bits to have certain values. For example, suppose we wish to jump to a location **w** if bit positions 15 and 4 have the

values 1 and 0, respectively, in ECX. We could use the following code to do this:

```
andl $0x00008004, %ecx
cmpl $0x00008000, %ecx
jz w
```

The point is that after the **andl** is executed, c(ECX) will equal 0x00008000 if and only if the original value in ECX had 1 and 0 in bit positions 15 and 4.

The **&** operator in C (the binary operator, not the “address of” operator) performs an AND operation, and in fact the compiler will probably translate a statement containing **&** into an **andl** instruction.

In an OR operation, if at least one of the two corresponding bits in the two operands is a 1, then the bit in the result is a 1, while that bit is a 0 otherwise.

The **orl** (“OR long”) instruction performs an OR operation on the source and destination, placing the result back in the destination. Again, it is important to note that that means that for every bit in the source operand which is a 0, the corresponding bit in the destination remains unchanged, while every 1 in the source will cause the corresponding bit in the destination to become 1. In other words, **orl** is used to place 1s in desired spots within the destination while keeping the other destination bits unchanged.

So for example to set bit 2 to 1 in EAX, we would use the instruction

```
orl $0x00000004, %eax
```

The **|** operator in C performs a logical-or operation, and the compiler will usually translate it to **orl** or other instruction from the OR family.

Logical operations are quite prominent in the programming of device drivers. For example, the built-in speaker on a PC has a 1-byte **port** at address 0x61. The least-significant 2 bits must be set to 11 for the speaker to be on, 00 for off; the other bits contain other important information and must not be changed. I/O ports, as we will see later, are accessed on Intel machines via Intel’s IN and OUT instructions.³ The following code turns the PC speaker off and on:

```
1 # to turn speaker on:
2 # copy speaker port to AL register
3 inb $0x61,%al
4 # place 11 in last 2 bits of AL
5 orb $3,%al
6 # copy back to speaker port
7 outb %al,$0x61
8
9 # to turn speaker off:
10 # copy speaker port to AL register
11 inb $0x61,%al
12 # place 0s in last 2 bits of AL
13 andb $0xfc,%al
14 # copy back to speaker port
15 outb %al,$0x61
```

The NOT instruction (e.g. **notl**) changes all 1 bits to 0s and all 0 bits to 1s.

Each bit position in the result of the XOR (“exclusive or”) instruction is equal to 1 if and only if exactly one of the operands has a 1 at the same bit position.⁴ It is a classic way of zeroing-out a register using a very short instruction. For example,

³Accessing them requires status as privileged user.

⁴You can also think of it as adding two bits and taking the sum modulo 2.

```
xorl %ecx, %ecx
```

will make all bits of ECX equal to 0.

There are also **shift** operations, which move all bits in the operand to the left or right. The number of bit positions to shift can be specified either in an immediate constant, or in the CL register.⁵

For example,

```
shll $4, %eax
```

would move all bits in EAX leftward by 4 positions. Note that those which move past the left end are lost, and the bit positions vacated near the right end become 0s. So, for instance, if $c(\text{EAX})$ had been `0xd5ffff3`, after the shift it would be `0x5ffff30`.

The **shrl** instruction does the same thing, but rightward.

Note that if we are considering the contents of the operand to be an unsigned integer, a left shift by 4 bits is equivalent to multiplication by $2^4 = 16$, and a right shift by 4 bits is equivalent to division by 16.

If on the other hand we are considering the operand to be a signed integer, we would use **sall** and **sarl**, since these preserve signs. It is not an issue for multiplication (except for overflow), but it makes a difference for division.

This difference is summarized in the comments in the following code:

```
1 # the bit string 0x80000000 represents 2,147,483,648 if viewed as an
2 # unsigned number, and -2,147,483,647 if viewed as signed
3 movl $0x80000000, %eax
4 movl %eax, %ebx
5 shrl $1, %eax # c(EAX) becomes 0x40000000, i.e. 1,073,741,824
6 sarl $1, %ebx # c(EBX) becomes 0xc0000000, i.e. -1,073,741,823
```

In C/C++, left and right shift operations are performed via `<<` and `>>`. In the latter case, the compiler will translate to **shrl** if the operand is **unsigned**, and **sall** in the signed case.

4 Floating-Point

Modern Intel CPUs have a built-in Floating-Point unit, which is hardware to perform floating-point add, subtract and so on. Without this hardware, the operations must be performed in software. For example, consider the code

```
float u, v, w;
...
w = u + v;
```

With an Intel CPU, the compiler would implement the addition by generating an FADD (“floating-point add”) instruction. If there were no such instruction, the compiler would have to generate a rather length amount of code in which one would first decompose **u** and **v** into their mantissa and exponent portions,

⁵Recall that this is the lower 8 bits of ECX.

would then change one of the addends in order to match exponent values, would then add the mantissas, etc. The resulting code would run much more slowly would a single FADD instruction.

Intel CPUs have special floating-point registers, each 80 bits wide, thus giving more accuracy than the standard 32-bit size in which C/C++ **float** variables are stored. (The variables are still stored in that 32-bit format, but the actual arithmetic is done in 80 bits.) In ATT&T syntax, the registers are named **%st(0)** (also called simply **%st**), **%st(1)**, **%st(2)** and so on. The “st” here stands for “stack,” as the registers are indeed arranged as a **stack**.

We will learn about stacks in detail in our unit on subroutines, but the main point is that a stack is a “last-in, first-out” data structure, meaning that items are removed from the stack in reverse order from which they are inserted. For instance, say the stack is initially empty, and then we **push**, i.e. insert, the numbers 1.8, 302.5 and -29.5222 onto the stack. We say that -29.5222 is now at the **top** of the stack, followed by 302.5 and then 1.8. If we now **pop** the stack, that means removing the top element, -29.5222, so that the new top is 302.5, with 1.8 next. If we pop again, the stack consists only of 1.8. If we now push 168.8888, then that latter value is on top, with 1.8 next.

Here is some sample code:

```
1  .data
2  x:      .float 1.625
3  y:      .float 0.0
4  point25:
5         .float 0.25
6  .text
7  .globl _start
8  _start:
9      flds x # push x onto f.p. stack; x now on top
10     flds point25 # push 0.25; 0.25 now on top, followed by x
11     faddp # add the top 2 elts. of the stack, replace 2nd by sum,
12         # then pop once; x+0.25 now on top
13     fstps y # pop the stack and store the popped value in y
```

Here we see a new assembler directive, **.float**. This tells the assembler to arrange things so that when this program is run, the initial value at **x** in memory will be the (32-bit) representation of the number 1.625. From our unit on data representation, we know that this is the bit pattern 0x3fd00000. In fact, we would get the same result from

```
x:      .long 0x3fd00000
```

and would even save the assembler some work in the process, since it would be spared the effort of converting 1.625.⁶

There are two **flds** instructions. They are from the FLD (“floating-point load”) Intel family; the ‘s’ here means “stack,” just as ‘l’ means “long” in **addl** from the ADD family.

Read the comments in the code before continuing.

OK, let’s use GDB to learn more about this code:

```
(gdb) l
1      .data
```


⁶That effort would be the same as what **scanf()** would go through with **%f** format: It would count the digits after the decimal point, finding the number to be 3. It would then convert the characters ‘1’, ‘6’ etc. to 1, 6 and so on, and thus form the numbers 1 and 625. Finally it would do something like $a = 1 + 625.0/1000.0$.

```

2      x:      .float 1.625
3      y:      .float 0.0
4      point25:
5          .float 0.25
6      .text
7      .globl _start
8      _start:
9          flds x # push x onto f.p. stack
10         flds point25 # push 0.25
(gdb)
11         faddp
12         fstps y
(gdb) b 12
Breakpoint 1 at 0x8048082: file fp.s, line 12.
(gdb) r
Starting program:
/fandrhome/matloff/public_html/matloff/public_html/50/PLN/fp

Breakpoint 1, _start () at fp.s:12
12         fstps y
Current language: auto; currently asm
(gdb) nexti
0x08048088 in ?? ()
(gdb) p/f y
$1 = 1.875
(gdb) p/x y
$2 = 0x3ff00000

```

Nothing new here, except that we've used GDB's **p** ("print") command with **f** format. Obviously, the latter is similar to **%f** for **printf()**. We see that **y** comes out to 1.875, as expected. We also print out the value of **y** as a bit string, which turns out to be 0x3ff00000. You should check that that is indeed the IEEE format representation of 1.875. 

Of course, there are also the obvious instruction families such as FSUB for subtraction, FMUL for multiplication and so on. But there are also instruction families that implement transcendental function operations, such as FSIN for sine, FSQRT for square root, etc. Again, by implementing these operations in hardware, we can get large speedups in C/C++ programs that make heavy use of **float** variables, such as programs that do graphics, games, scientific computation and so on.

The Intel hardware also gives the programmer the ability to have fine control over issues such as rounding, via special instructions that control **floating-point status registers**.