First Look at ia32 Assembly Language

In this chapter, we will take a first look at the assembly language and machine language of the ia32. Rather than start from scratch, we are going to ask gcc to be our tutor. What we will do is to write some very simple C programs, and then we will ask gcc to show us the assembler code that it generates for these C programs. Then the task will be to understand why these assembly instructions that are generated do in fact result in the right behavior given the original C program.

For a first example, we will use the following C code

```
unsigned a = 1;
unsigned b = 2;
unsigned c = 3;
void t () {
  a = b + c;
  if (a == 4)
    b = 3;
  else
    c = a & b;
  while (a > 0) a--;
  }
```

For the moment, we avoid the use of signed integers, and we avoid either passing arguments to functions or trying to return results to functions. Right, so let's ask gcc to compile this, and instead of generating machine language, let's ask gcc to show us the assembly language. Normally gcc generates this assembly language in a temporary file, assembles it using the assembler into machine language, and then deletes the temporary file, but by using –S instead of –c, we ask gcc to simply generate the assembly language (into a file called name.s where the C program was name.c), and then we can look at this assembly language. The exact command we use to compile, assuming that the above example is stored in a file called t.c, is

```
gcc –S t.c –fomit-frame-pointer –masm=intel
```

Here, the switch –S asks for assembly language to be generated, as discussed above. The switch **–fomit-frame-pointer** asks gcc not to use a frame pointer. We don't know yet what a frame pointer is, and that's the point. We don't want to worry about frame pointers, so this option gets rid of them for now. The switch **–masm=intel** asks gcc to use Intel syntax for the assembly language. There are two quite different syntaxes in use for ia32 assembly language. The Intel syntax is the one that Intel originally devised for this architecture. The AT&T syntax is typically used on Unix, and is more similar to the assembly language used by other processors. There is no particular reason technically to prefer one over the other. We choose to use the Intel syntax simply because most text books on assembly language for this machine use this syntax, so if you are using some auxiliary reference materials, life will most likely be easier using the Intel syntax.

With this command line, the output of gcc is stored in file **t.s** and looks like:

```
     .file     "t.c"
     .intel_syntax
.globl _a
     .data
     .align 4
_a:
     .long     1
.globl _b
     .align 4
_b:
     .long     2
.globl _c
     .align 4
_c:
     .long     3
     .text
.globl _t
     .def _t;  .scl 2;   .type     32;  .endef
_t:
     mov  eax, DWORD PTR _c
     add  eax, DWORD PTR _b
     mov  DWORD PTR _a, eax
     cmp  DWORD PTR _a, 4
     jne  L2
     mov  DWORD PTR _b, 3
     jmp  L3
L2:
     mov  eax, DWORD PTR _b
     and  eax, DWORD PTR _a
     mov  DWORD PTR _c, eax
L3:
L4:
     cmp  DWORD PTR _a, 0
     je   L5
     dec  DWORD PTR _a
     jmp  L4
L5:
     ret
```

So now let's get busy understanding this, line by line. A general note here is that the lines that start with a period are directions to the assembler, and are typically not part of the actual program. It's as though we wrote down a speech for a policitian, and at the start we had a direction saying "remember to smile and don't snear". We don't expect the politician to read these words at the start of the speech (though you never know these days ☺) The dot lines are similar, typically they are not part of the program proper, but rather they are directions to the assembler.

```
        .file     "t.c"
```

The **.file** line simply records the name of the original C file for informational purposes. This is not part of the program, but can be useful for both humans and other computer tools in keeping track of where things came from.

```
        .intel_syntax
```

As we discussed above, there are two different syntaxes for ia32 assembly language. The default is AT&T syntax. This directive tells the assembler that the rest of the file will use the Intel syntax.

```
.globl _a
```

This line is a note to the assembler that the symbol _a can possibly be referenced from other files. The assembler will notify the linker so that the proper inter-file connections can be made. There is no effect on the actual code generated for the program. Note that all symbols in the original C program have an underscore appended. This avoids name clashes with some existing symbols (at least that was historically the reason for this decision, though probably it is no longer really necessary).

```
        .data
```

A program is generally divided into data and code. Generally these two sections should not be mixed up. You don't want to execute your data as code, and you don't want to treat your code as data. The **.data** directive tells the assembler that the following lines generate data rather than code. The assembler and linker will between them arrange to place data and code in separate sections of memory, so that they are kept apart.

```
        .align 4
```

On the ia32, there is no requirement for data alignment. A program will work correctly with four-byte integers regardless of where they are located. For example, a four byte integer could be located at addresses 1,2,3,4. However, the machine executes much more efficiently if, for example, four byte integers are on a four byte boundary, so a better choice of starting address for a four byte integer is an address that is a multiple of 4. The .align directive tells the assembler to bump the location counter (the location of the next data to be generated) to the next four byte boundary. This may or may not waste space depending on the current value. Typically the data from a given file always starts on a four byte boundary, so most likely the alignment directive has no effect in this particular case, but it is certainly harmless, and in the general case it may improve efficiency by ensuring that the value about to be generated after the labal is optimally aligned for the most efficient execution.

```
_a:
```

This is a label. It causes the symbol **_a** to be assigned to the address of the next data or code to be generated. Later on we can reference this address by using this label name.

```
     .long     1
```

This is the first line in the assembler file that actually generates something. The **.long** directive causes four bytes (a long word) of data to be generated, initialized to the given value. Since this is a little-endian machine, the four bytes generated will contain 1, 0, 0, 0 in sequence.

```
.globl _b
     .align 4
_b:
     .long     2
```

Similar declarations for the variable b, initialized to 2.

```
.globl _c
     .align 4
_c:
     .long     3
     .text
```

Similar declarations for the variable c, initialized to 3**.**

```
.globl _t
     .def _t;  .scl 2;   .type     32;  .endef
```

Now we have the start of function t. The **.globl** declaration as before informs the assembler that this function is accessible from other files. The **.def** line has some information about the function. The **.scl** value says that the symbol is an external gobal symbol (which is redundant information, given the **.globl** directive, and is in fact ignored). The .type value indicates a function returning void. This is information that could be used by other tools, e.g. type checking tools. In fact it is typically not used at all, and can be omitted. It is there simply because the official rules for the assembly language syntax say it should be there.

```
_t:
```

This is the label for the function. We will be able to use this label to refer to the function, e.g. in a call instruction as we will see in the next chapter. As we see, both instructions and data can have labels. The distinction between code and data lies in the way we reference this data. You do not jump to data, and you do not load instructions, at least not in a correct program, unless something very tricky is going on.

```
     mov  eax, DWORD PTR _c
```

Finally, a machine instruction. Typical instructions take two operands, the left operand is the destination and the right operand is the source. The **mov** instruction simply moves data

from the right hand operand, which in this case is a double word (four byte integer) stored starting at label **_c**, to the register **eax**. The data in memory is not changed, so this is very like an assignment statement in a typical high level language:

```
eax := c;
```

This pattern of loading values into registers and then operating on the values in rthe registers, and then finally storing the results is a typical pattern in machine language programming since access to registers is fast, and access to memory is relatively slow.

```
add  eax, DWORD PTR _b
```

For binary operations like **add**, the left and right operands are the operands for the binary operation, and the result is stored back in the left operand, so this instruction has an effect like the following assignment in a high level language:

```
eax := eax + b;
```

The **add** instruction also sets the **CF** and **ZF** flags as follows. **ZF** is set (i.e. will contain a 1 bit) only if the result is all zero bits (which can happen if both operands are zero, or if the result exactly. If the result is non-zero, then **ZF** is clear (i.e. will contain zero). **CF** is set if the addition causes a carry, and is clear otherwise. The add instruction always sets these two flags (it also sets some other flags, but we are not discussing that at this stage). However, it is often the case, as here, that no one cares how they are set, and the flag values are ignored.

```
mov  DWORD PTR _a, eax
```

This move instruction moves the result back from **eax** into variable a, completing the translation of the first line of C in the function (**a = b + c**). So one line of C expanded into three lines of assembly language. This kind of expansion is typical, and is one of the reasons why we prefer to program in a high level language if we can. Note by the way that the flags are still set from the add instruction, since **mov** instruction does not affect the flags. However, we still don't care how they are set.

```
cmp  DWORD PTR _a, 4
```

The **cmp** instruction is used to compare two operands, in this case the variable _a on the left and the constant 4 on the right, corresponding to the test in the if statement in the C program. What **cmp** does is actually to subtract the right operand from the left operand and then throw away the result, but the **CF** and **ZF** flags are set. ZF is set if the result of the subtraction is all zero bits, and **CF** is set if a borrow occurs, which happens if the right operand is greater than the left operand. Let's make a little chart showing how the flags are set after a compare instruction (cmp A, B):

```
A < B          CF = 1          ZF = 0
A = B          CF = 0          ZF = 1
A > B          CF = 0          ZF = 0
```

Since the flags can distinguish the three possible results of the compare, we can test the flags to determine the outcome of the comparison. There are several ways to test the flags, but the most usual one is a conditional jump, as in:

```
jne  L2
```

The **jne** instruction is called a conditional jump instruction. It tests a condition (in this case the condition is that the ZF flag is not set), and if this condition is met, it copies the given label address into the **EIP** register so that the next instruction will be taken from the jump target. The effect is similar to the C statements

```
if (ZF = 0) goto L2;
```

You may well have been taught never to use goto statements. In fact you may have learned programming using a language that does not have goto statements at all. If so, you had better forget the rule, since gotos are the only way of changing the flow of control in machine/assembly language. One of the jobs of a compiler for a nice high level language is to translate nice control structures like conditionals and loops into equivalent sequences of goto (or as they are called in assembly language, jump) instructions.

Note by the way that since this instruction tests the **ZF** flag, it would seem more natural to have a name like **jnz** (jump if ZF not set). In fact there is such an instruction, and furthermore, **jne** is simply a synonym for **jnz**. The **jne** synonym is provided for the benefit of human readers. It is easier to think of the compare and jump as a unit, even though the machine executes them separately, and read as a unit, **jne** is clearer.

If we look at the C code, this instruction makes perfect sense. If a is not equal to 4, we jump to the else section, otherwise we fall through to the then section of the if statement and execute the instruction:

```
mov  DWORD PTR _b, 3
```

which copies the value 3 into variable b which is indeed the content of the then branch of the if statement. Once this is completed, we want to skip past the else section, which is achieved with the instruction:

```
jmp  L3
```

which is an unconditional jump. This instruction unconditionally puts the address that is associated with **L3** into the **EIP** register, so that execution continues with **L3**, the label just past the if statement.

```
L2:
```

This is the label on the else part. Control passes here from the **jne** instruction if the condition of the **if** test was false, and we then execute the **else** part.

```
mov  eax, DWORD PTR _b
and  eax, DWORD PTR _a
mov  DWORD PTR _c, eax
```

These three statements are just like the addition operation earlier on, except the actual operation is **and** rather than **add**, corresponding to the C operator &. Note that **CF** and **ZF** are also set by the **and** instruction. **CF** will always be reset to zero (since the **and** operation cannot cause an overflow), **ZF** will be set if the result is all zero bits. Again we don't actually test the flags from this operation. There are also instructions **or** and **xor** corresponding to the logical or operation (the | operator in C) and the logical xor operation (the ^ operator in C).

```
L3:
```

That's the label marking the end of the if statement. It was used at the end of the then part, to transfer control past the if statement

```
L4:
```

Now, we come to the while loop. The label **L4** is the label at the top of the loop. Note that we could have used **L3** for this, but the compiler tends to do one thing at a time, so it translates the **if** statement as a unit, and then it translates the **while** statement as a unit. In any case there is no harm in having two labels here, they have the same value, and are interchangeable, but the label itself does not generate any code, so piling lots of unnecessary labels into a program is ugly looking, but completely harmless.

```
cmp  DWORD PTR _a, 0
je   L5
```

These two instructions implement the while test. If the **while** test is false, then the **je** instruction (**je** is a synonym for **jz**, which tests if the zero flag is set, as it will be when variable a is zero) sets the **EIP** register to the address **L5**, which is a label just past the while statement, so it has the effect of exiting from the loop. If the condition is false, control falls through to the next instruction, which is the body of the loop

```
dec  DWORD PTR _a
```

The **dec** instruction decrements its operand by one. It is almost exactly equivalent to subtracting one, as in the following almost equivalent instruction:

```
sub  DWORD PTR _a, 1
```

The difference is that **dec** never affects the value of the **CF** flag (though it does set or reset **ZF** depending on whether the result is zero. In this case we don't care how the flags are set so the use of **dec** is just fine. The explanation of why **dec** does not set the **CF** flag (there is a similar instruction **inc** that increments by one, and also leaves **CF** unchanged) is a bit subtle, and we will explain the motivation later on.

Now the body of the loop is completed, and so we need to jump back to the start

jmp L4

and the above unconditional jump achieves that goal, since **L4** was the label that was output at the start of the loop, just before the while test. Control will repeatedly flow through the loop and back to the while test, until the condition tested in the while is finally false, and at that point, the while test at the top of the loop will exit the loop by transferring to the label:

L5:

which marks the end of the loop. Now the code of the function is completed, with the results being left behind in the global variables and all we have to do is to return to the caller of the function.

ret

And that's the return instruction that achieves this goal. The effect of **ret** is to cause instruction execution to continue past the point of the call in the calling code that called this function. That's easy enough to say, but you may want to ask how **ret** works. For now, we do something that we try to avoid. We invoke magic, somewhat in the style of a parent who in response to a child's question *why*, answers *because*.

We don't really like to invoke magic, but in this case, the true explanation is quite complex and there is quite enough on the plate already. We won't have to wait too long. The next chapter is entirely devoted to the issue of how the **call** and **ret** instructions work.

Code Optimization

If you look carefully again at the sequence of code that is generated for this function, it's a bit stupid. It is not hard at all to reduce the number of instructions needed for this particular function, and in particular to reduce the number of dreaded memory loads and stores, which are so much slower than doing things in registers. So you may conclude that the compiler is rather stupid.

While it is true that no compiler can do as well as the most skilled human when it comes to generating efficient assembly code, we are not being fair to gcc if we leave things like this. Our default command for compilation:

```
gcc –S t.c –fomit-frame-pointer –masm=intel
```

did not specify an *optimization level*. As a result, gcc defaulted to the so called –O0 (oh zero) mode which says "compile as quickly as you can, I don't care if you generate code that is large and slow. If, as is often the case, we prefer small quick code, we can tell gcc to optimize the code by setting the appropriate optimization flag. There are several settings, corresponding to increasing efforts at improving the code in return for spending more time on the compilation. For real life programs, it is often the case that we will run the same

program many times, but compile it only once. In this case the trade off will favor slow compilation and fast/small code. On the other hand, if we are developing and testing our code, we often prefer the default no optimization mode, to keep compilation times short. Let's try out the effect of optimization on our example. The –O1 mode sets the next level of optimization.

```
gcc –S t.c –fomit-frame-pointer –masm=intel –O1
```

Now lets first recall the result without optimization (-O0) from before (looking just at the assembly code for the function itself):

```
_t:
     mov  eax, DWORD PTR _c
     add  eax, DWORD PTR _b
     mov  DWORD PTR _a, eax
     cmp  DWORD PTR _a, 4
     jne  L2
     mov  DWORD PTR _b, 3
     jmp  L3
L2:
     mov  eax, DWORD PTR _b
     and  eax, DWORD PTR _a
     mov  DWORD PTR _c, eax
L3:
L4:
     cmp  DWORD PTR _a, 0
     je   L5
     dec  DWORD PTR _a
     jmp  L4
L5:
     ret
```

That's fifteen instructions, and ten memory references. The number of instructions is not too bad, but that's a lot of memory accesses, so the run-time efficiency may be low.

For comparison here is the result in –O1 (oh one) mode:

```
_t:
     mov  eax, DWORD PTR _b
     add  eax, DWORD PTR _c
     mov  DWORD PTR _a, eax
     cmp  eax, 4
     jne  L2
     mov  DWORD PTR _b, 3
     jmp  L3
L2:
     mov  eax, DWORD PTR _a
```

```
     and  eax, DWORD PTR _b
     mov  DWORD PTR _c, eax
L3:
     cmp  DWORD PTR _a, 0
     je   L8
     mov  eax, DWORD PTR _a
L6:
     dec  eax
     jne  L6
     mov  DWORD PTR _a, eax
L8:
     ret
```

The optimized code is 17 instructions (two more than before), and there are still ten memory references. Well that's a disappointment, things seem to have got worse. But wait, let's take a closer look. First of all, notice the difference in dealing with the first assignment. It still takes three memory references, but the result has been left in **eax**, and the immediately following test uses the value in **eax**, saving a memory reference. So that's good. How come we didn't save memory references in the rest of the code? Well, look again, what matters is not the number of memory references in the whole program, but the number of memory reference instructions that actually get executed. Let's just focus our attention on the loop itself. In the unoptimized code, the loop is:

```
L4:
     cmp  DWORD PTR _a, 0
     je   L5
     dec  DWORD PTR _a
     jmp  L4
```

Compare that to the optimized code loop

```
L6:
     dec  eax
     jne  L6
```

Wow! That's a huge difference, each iteration of the loop is down from 4 instructions and two memory references to 2 instructions and no memory references. What the compiler did was to realize that the memory references are all referring to the variable a, so instead of continually referencing the variable in memory, the new improved code first loads the value of a into eax, then does the loop using this register, and then after the loop is over dumps the value back into memory. The total number of memory references doesn't change since we still have to load and store a, but the number of loads and stores executed dramatically decreases. Programs generally spend all their time in loops, so if we can speed up loops like this, we will get a big speed up.

Instruction Review

We studied one simple example in this chapter, but we learned quite a few instructions from this one example so let's review them.

Move Instruction

```
mov  a, b
```

The **mov** instruction copies the second operand into the first operand, like an assignment statement. The value of **b** is unchanged. The first operand, **a**, can be a four byte memory location, or one of the 32-bit registers. The second operand, **b**, can be a four byte memory location, or one of the 32-bit registers, or an immediate constant value, except that there is no single instruction for moving from one memory location to another (such an operation must be achieved by using an intermediate register and two separate move instructions). Here are examples of the various **mov** instructions:

```
mov  eax, ebx            register to register
mov  eax, dword ptr x    memory to register
mov  dword ptr y, esi    register to memory
mov  eax, 3              immediate to register
mov  dword ptr z, 15     immediate to memory
```

Binary Operation Instructions

```
add  a, b
sub  a, b
and  a, b
or   a, b
xor  a, b
```

These instructions compute the corresponding binary operation (addition, subtraction, logical and, logical or, logical exclusive or). The two operands are the two operands of the binary operation, and the result is stored back into the left operand. Again there are multiple forms, everything except memory to memory is allowed. Here are examples:

```
add  eax, ebx            register to register
sub  esi, dword ptr q    register and memory
and  dword ptr x, ebp    register to memory
or   ecx, 3              immediate to register
xor  dword ptr z, 1      immediate to memory
```

All these instructions set CF and ZF. CF is set to 1 if a carry or borrow occurs (only possible or add and sub), and CF is set to 0 if there is no carry (it is always 0 after any of the logical instructions. ZF is set if the result is all zero bits, and is otherwise 0. These instructions have all sorts of uses. Here are three ways to set a register to zero.

```
sub  esi, esi
and  esi, 0
```

```
xor  esi, esi
```

Here is an interesting way to exchange two registers:

```
xor  esi, edi
xor  edi, esi
xor  esi, edi
```

That's quite a non-obvious trick. Try some sample values and see if you can figure out how this works.

Increment/Decrement Instructions

```
inc  a
dec  a
```

These instructions increment or decrement the operand, which can be a register or a four byte memory location, by one. Some examples are

```
dec  edi
inc  dword ptr x
```

A bit oddly (to be explained later) these instructions never affect the value in the **CF** flag, but the **ZF** flag is set if the result is zero (happens when we add 1 to the largest number, or subtract 1 from 1), and cleared otherwise. Apart from this little difference in the handling of the **CF** flag, these instructions are virtually identical to the corresponding **add** and **sub** instructions with a second argument being the constant 1, but they are shorter instructions so we prefer to use them when we can.

Comparison Instruction

```
cmp  a, b
```

The comparison instruction is identical to a subtract (sub) instruction with one small detail that differs, namely the comparison instruction does not write the result back to the first operand. So what use is it? Well it sets the ZF and CF flags. The ZF flag is set if and only if the result is all zero bits (which happens only if a and b are equal). The CF flag is set if and only if the subtraction causes a borrow (which happens only if b is greater than a). For now we assume that all operands are unsigned 32-bit integers.

Like the sub instruction, cmp exists in five different forms:

```
cmp  eax, ebx            compare two registers
cmp  esi, dword ptr q    compare register to memory
cmp  dword ptr x, ebp    compare memory to register
cmp  ecx, 3              compare register to constant
cmp  dword ptr z, 1      compare memory to constant
```

Equal/Not Equal Conditional Jump Instructions

```
jne  labl
jnz  labl
je   labl
jz   labl
```

These instructions test the setting of the **ZF** flag and jump if the associated condition is true. For **jne** and **jnz** (which are simply two different ways of spelling the same instruction), the condition is that **ZF** is 0. For **je** and **jz**, which again are different spellings of the same instruction, the condition is that **ZF** is 1.

These instructions can be used to test the result of an operation, in which case we would usually use the **jz**/**jnz** forms:

```
sub  edx, 1
jz   L3         # jump if count down to zero

and  word ptr a, 1
jnz  L4         # jump if word stored at a is odd
```

Or they can be used after a comparison instruction to see if the result of the comparison was equal or not equal, in which case we usually use the **je**/**jne** forms.

```
cmp  edx, 1
je   L3         # jump if count is one

cmp  word ptr a, 0
jne  L4         # jump if word stored at a is non-zero
```

By the way we have snuck in one more detail here. The compiler is not in the business of generating comments, but if humans write assembly language programs by hand, we are in rather desperate need of comments. The character # signals a comment, and can either appear on the same line as the instruction (as in the above examples), or we can have comments all on their own line as in

```
#  Start of function f which does exciting stuff
```

Comments are really important in assembly language programs. are common about whether high level languages like C or COBOL are self documenting, but everyone agrees that assembly language is not self documenting and comments are essential. A style in which virtually every line of code has a comment is not at all inappropriate.

The Return Instruction

```
ret
```

This is the notorious return instruction, which magically causes control to pass back to the caller. Notorious because we have not explained how it works, but we will fix that right now. The very next chapter explains the **call** and **ret** instructions in (somewhat excruciating) detail.

We have only learned a few instructions, but that's actually enough to write some quite complex programs reasonably easily. We will learn a few more instructions, but not that many more. One of the interesting discoveries of computer design was that not only is it feasible to limit a processor to a small set of instructions, but it is actually generally desirable to do so. That's what RISC (reduced instruction set) computing is about, and we will look at these details later.

The ia32 architecture design preceded the RISC enlightenment, and so this processor has dozens, even hundreds of instructions, some of which are so complicated that they take pages of complex text to describe. But don't worry, we don't need to learn most of these instructions, and furthermore, typical compilers won't use them anyway.