Module 0215: 386 instruction addressing modes

Tak Auyeung, Ph.D.
February 2, 2010

1 About this module

- Prerequisites:
- Objectives: This module enumerates addressing modes of the 386 processor.

2 The “mov” instruction

Before we can discuss addressing modes, it is important to start with an instruction. It is natural to choose the mov instruction because it does not involve any computation, and yet it has a source and a destination operand.

In general, a mov instruction looks like the following:

\texttt{mov[b|w|l] \ <src>, \ <dest>}

This is a typical syntax description that is also used to describe the syntax of many other languages. Here is a quick explanation:

- Regular text like mov are required verbatim.
- Anything enclosed by square brackets ([ ]) is optional.
- Alternatives are separated by the vertical bar (|).
- Place holders to be replaced by something else are enclosed by angle brackets (< >).

As a result, in this case, the instruction itself can be mov, movb, movw or movl. It must have two operands, < src > and < dest >. The two operands must be separated by a comma. An operand of an instruction is like arguments of a function. Operands supply extra information that is needed for an instruction to execute.

Note that the mov mnemonic is actually a misnomer. It should have been cpy for copy. This is because memory content is not moved, but rather copied. In other words, the < src > still retains the same value after a mov instruction.

3 Operand categories

There are three main types of operands. This section takes a superficial look at the operand categories.

3.1 Immediate

An immediate operand, also called a constant operand in some contexts, is a bit pattern that is embedded into an instruction in machine code. As such, immediate operands do not use data memory at all, as all instructions should be in code memory.

Furthermore, the value of an immediate operand cannot be changed in most situations, as it is very poor practice to modify instructions (in machine code) when a program executes. Self modifying programs exist for very specialized purposes, such as obfuscation.

Immediate operands are only used as a source operand. This is because an immediate operand cannot be changed (since it is a part of an instruction).

The access speed of immediate operands is fast, as the value of the operand is a part of the binary bit pattern of an instruction.
3.2 Register

A register operand is one that refers to the value of a register. Most processors, such as the 386, has a small number of registers to store values to be accessed frequently in the near future (the next few instructions).

The reason why there are few registers is two folded. First, a modern processor uses general purpose registers, which means all registers can be used as the source and/or destination operands of an instruction. This means that a switch must be implemented in silicon gates to let an instruction select any one register. This kind of switch is expensive in silicon.

Second, a register must keep up with the speed of the ALUs (arithmetic and logic units) of a processor. ALUs use combinatorial logic to implement operations like add, subtract, divide and etc. ALUs of a modern processor can perform one operation per CPU clock, which is extremely fast.

In order for registers to keep up with this speed, they must be physically close to the ALUs. However, only so many registers can be placed “close enough” to the ALUs to keep up with the speed of the ALUs. As a result, the number of registers is restricted.

A register operand can be a source operand (to use the value of a register as a source) or a destination operand (to update the value of a register).

3.3 Memory

A memory operand is one that refers to the value of a memory location. Of the three types, memory operands are the slowest. However, a memory operand can specify any one of the $2^{32}$ memory locations, which makes it very flexible.

Memory operands can serve as source and/or destination operands in an instruction.

The 386 instruction set offers many different ways to compute the address of the memory location to access. We will explore these later in this module. A RISC (reduced instruction set computer) typically offers far fewer ways to specify the address of memory location.

4 Other assembly language constructs

In order to illustrate how to use the different addressing modes, it is necessary to introduce a few assembly language constructs.

4.1 Memory sections

A memory section is a contiguous area when a program is loaded as a process. The concept of memory sections is necessary for several reasons.

Many processors, including the i386, has the concept of segments. Not counting historical reasons from the 8086 days, there are still two main segments that are important. A code segment/section is used to store instructions, and a data segment/section is used to store data and the system stack.

This differentiation is needed to implement certain security features. For example, newer Pentium and AMD equivalent processors can inhibit code execution in the data segment/section. This stops buffer overflow and stack overflow exploits completely (as both need to execute exploit code overwritten in data memory).

The assembler allows a program to define any number of sections. However, two sections are predefined. The section that is for code (instruction) is called the “text” section, whereas the section that is for data is called the “data” section.

To switch to the text section, use the .text assembly language directive on a line. To switch to the data section, use the .data assembly language directive on a line.

Let us take a look at the code in listing 4.1.

```
.data
  ... # some data definitions
.text
  ... # some code definitions
.data
  ... # more data definitions
```

Even though the two .data portions are interleaved by a .text portion in the source code, the assembler actually make the two .data portions contiguous in the data section.
Instructions should always be placed in the text section. However, data definitions can go to the data section (for normal variables that change) or the text section (for const variables).

4.2 Memory allocation
A program can allocate memory from any section/segment and initialize the memory locations at the same time. This is quite useful for initialized variables in a program.

The following is a summary of the main types of memory allocation and initialization directives:

- `.int <v1>{, <v>}*`
  This notation means that the `.int` directive has one or more integer values, separated by commas. The curly braces `{}` is a grouping mechanism, while the asterisk `*` means any number of the previous item. Each value of an `.int` directive is encoded as a 32-bit integer.

- `.word <v1>{, <v>}*`
  Similar to `.int`, except each value is encoded as a 16-bit (word) integer.

- `.byte <v1>{, <v>}*`
  Similar to `.int`, except each value is encoded as a 8-bit (byte) integer.

- `.fill <count>[, <size>[, <value>]]`
  `.fill` lets you specify a flexible repeating pattern. `<count>` specifies how many items. `<size>` specifies the number of bytes used by each item, and `<value>` is the value of each item. In the absence of `<value>`, 0 is assumed. In the absence of `<size>`, 1 is assumed.

- `.ascii <string>`
  `.ascii` allocates and initializes memory locations for a string, represented in ASCII (one byte per character). A string is quoted by the single quote symbol.

- `.asciz <string>`
  `.asciz` is similar to `.ascii`, except a trailing byte of 0 is included for a null-terminated string.

4.3 Labels
One of the convenient features of an assembly language is the use of labels. A label is a symbolic name (think C identifier) that is associated with a specific numerical value. This allows the use of symbolic names instead of literal numerical values throughout a program.

A label is defined as follows:

```
<labelname> = <value>
```

`<labelname>` follows almost the same rules as C identifiers. If in doubt, just follow all the rules of C identifiers. `<value>` is an expression that has to evaluate to a numerical value.

A simple example of a label definition is as follows:

```
three = 3
```

This equates the symbolic name `three` with the value of 3.

You can also use any arithmetic expressions:

```
six = three + three
```

It is important to understand that label definitions do not take up any space at run-time. They are symbolic items that are used only in assemble time and link time. As a result, you should use as many symbols as necessary to make your program easy to understand and flexible.
4.4 The current (section) location

As you define data entries (using .int, for example) and specify instructions, a section (text or data) will be filled
with content. Each section has a “current location” counter to keep track of the number of byte locations used up
so far. The “current location” counter is represented by a single period (.).

For example, if you want to allocate 20 bytes without specifying what to initialize it with, you can use the
following directive:

. = . + 20

This directive says that the current location should be incremented by 20, which essentially takes up 20 bytes of
the current section (either text or data).

More often, we use the current location to define symbolic labels:

prompt = .
.asciz 'Enter a value'

In this example, prompt is defined to be a particular location (where the current location was) before the null-
terminated string is allocated. This means that prompt is actually defined as the location of the first byte (E) of the
null-terminated string.

Defining a symbol as the current location is a frequent operation. That is why it has a short form. The following
code is equivalent to the previous code:

prompt: .asciz 'Enter a value'

Note that .asciz can start on the following line, but this form is more compact.

5 i386 specific addressing modes

This section discusses i386 specific addressing modes, along with the AT&T syntax. There are two main types of
x86 assemblers, one uses the Intel notation, the other uses the AT&T notation. The assembler as uses the latter.

5.1 Immediate

An immediate operand is an general expression that starts with the dollar ($) symbol. The value of an immediate
operand is the value of the expression.

Since the mov instruction requires a destination operand, but an immediate operand cannot be a destination, we
cannot use an example that uses only an immediate operand.

However, if you insist, here is one instruction using an immediate operand (everything to the right of the pound
(#) symbol is comment):

pushl $0 # push a 32-bit zero on the stack

5.2 Register

A register operand is the symbolic name of register prefixed by the percent % symbol.

Due to historical reasons, the i386 has some interesting combinations of registers. Let us start with 8-bit registers.
The eight 8-bit registers are ax, ah, bl, bh, cl, ch, dl, and dh. The “l” stands for “low”, and the “h” stands for
“high”.

There are four general purpose 16-bit registers. Each one is a concatenation of the matching 8-bit registers. For
example, ax (“x” stands for extended) is a 16-bit register in which al makes up the least significant 8 bits, and
ah makes up the most significant 8 bits. The other three 16-bit registers are bx, cx and dx. There are other 16-bit
registers, but they are generally not useful when a modern processor is in 32-bit mode.

Last, but not least, there are 8 32-bit registers. The first four general purpose registers are extensions of ax to
dx. eax is the “enhanced” (hence “e”) version of ax. This means that ax is the least significant 16 bits of eax.
There is no register that specifies only the most significant 16 bits of eax. Likewise, ebx, ecx and edx are enhanced versions of bx, cx and dx.

The other four 32-bit registers are useful as pointers.

- esi is the source index register. It is used implicitly by some CISC instructions.
- edi is the destination index register. It is used implicitly by some CISC instructions.
- esp is the stack pointer. We’ll talk about this extensively in some other modules.
- ebp is the frame pointer (aka base pointer). It is closely related to esp.

The following is an example of a mov instruction that copies the value of eax to ebx:

```assembly
mov %eax, %ebx
```

### 5.3 Direct addressing mode

The is our first memory operand. A direct operand specifies an address, and the value at that address is the value of the operand. In a direct operand, the address cannot change. However, the value at that address can change over time.

A direct operand is specified by an expression that evaluates to the address. Let us take a look at the following confusing example:

```assembly
mov $0, 0
```

What does it do?

The source operand is $0, which means it is an immediate operand that specifies a value of 0. The destination operand is 0, which is just an expression. The destination operand is a direct operand that specifies location 0! In other words, this instruction copies a value of zero to location zero in memory.

Since location 0 is usually not allocated to a process, the execution of this instruction is likely to result in a GPF (general protection fault) or segmentation fault (aka segfault). A segfault essentially means that a process attempt to access ammeory location that it has no permission to access.

In general, we do not use a numerical constant for direct operands. Instead, we use labels that are defined as the first location of some allocated memory. Let us take a look at listing 5.3.

```assembly
.data
label1: .int 0x5
.text
... # some code
mov $23, label1
... # some more code
```

Let us take a look at the mov $23, label1 instruction. The source operand is obvious, it is a constant of 23. What about the destination operand? The lack of a dollar sign means it is a direct operand. The expression consists of label1 itself. How was label1 defined? It is defined as the first byte of the 32-bit integer with an initial value of 5. Consequently, this instruction overwrite the location (that was initialized to 5) with a new value of 23.

### 5.4 Digression: Endianness and Instruction width

Hold on a second, there was some confusion with the previous section. When we specify the instruction mov $23, label1, how do we know the width of operands? How do we know what bytes are copied?
5.4.1 Operand width

The “width” of an operand is the number of bits. It can be 8, 16 or 32 (in 32-bit mode). The instruction 
\texttt{mov $23, label1} does not indicate the width, at all. Note that the value 23 needs at least 5 bits to represent,
but it can also be represented by more bits (just more leading zeros).

In other words, is the source operand \texttt{0b00010111}, \texttt{0b00000000 00010111}, or \texttt{0b00000000 00000000 00010111}?

In the absence of a width indicator (\texttt{b} for 8 bits, \texttt{w} for 16 bits or \texttt{l} for 32 bits), the default is \texttt{l} (32 bits).

As a general rule, always use a width designator. Using an operator with the wrong width can cause interesting
but obscure problems. We will explore that in other modules.

5.4.2 Endianness

Now that we know how to specify the width of operands, how will the bytes be organized? After all, the binary
number \texttt{0b00000000 00010111} needs two bytes to represent, which byte has the lowest address?

Well, the processor industry cannot agree, either. Some manufacturers such as Motorola decides to put the most
significant byte (MSB) first, at the lowest address. Others, such as Intel, decides to put the least significant byte
(LSB) first, at the lowest address.

Little Endian refers to the Intel method, which is summarized as “least significant byte at the lowest address”.
Big Endian, of course, means the other way around, “most significant byte at the lowest address”.

This can lead to some confusion to people who are just getting into this, mostly because when we write a number
(in any base), we put the most significant digits on the left hand side. At the same time, when we list byte values
using a debugger or the \texttt{.byte} directive, we list the bytes in increasing address.

As a test, see if you can understand why the following directives define the same byte sequence:

- \texttt{.int 0xa852d9b2}
- \texttt{.byte 0xb2, 0xd9 0x52, 0xa8}
- \texttt{.word 0xd9b2, 0xa852}

5.5 Indirect addressing mode

Many RISC architectures do not implement the direct addressing mode. Instead, the only addressing mode support
is the indirect addressing mode.

The value of an indirect operand is the content of a memory location, and the address of the memory location is
specified by one of the registers. The syntax of an indirect operand is a register (has to be a 32-bit one) enclosed in
round parentheses.

For example, the following instruction initializes the byte “pointed to by eax” to zero:

\texttt{movb $0, (%eax)}

Just from looking at this instruction, we cannot determine which location will be overwritten with zero. This is
because the value of \texttt{eax} is unknown.

Recall that I said that most RISC architectures do not implement direct operands. The instruction in listing 5.3
can be implemented by the following two instructions:

\texttt{movl $label1, %eax}
\texttt{mov $0, (%eax)}

The first instruction, “\texttt{movl $label1, %eax}”, specifies an immediate operand as the source. The source operand
is simply the value of \texttt{label1}. \texttt{label1} is defined as the address of the first location of the 32-bit integer initialized
to 5. Consequently, the instruction copies the address of the integer to \texttt{eax}.

The second instruction, “\texttt{mov $0, (%eax)}” is an implied 32-bit copy. It copies the constant 0 to the 32-bit integer
pointed to by \texttt{eax}.
5.6 Another digression: CISC vs. RISC

Why do we want to design RISC processors? After all, some single CISC instructions (in combination with operands) expands to multiple RISC instructions. Isn’t this slower?

Because of the fewer addressing modes and instructions, RISC processors require fewer gates compared to a comparable CISC processor. This means that if both are without optimization and other speed enhancing techniques, a RISC processor core is much smaller than that of a comparable CISC processor due to the reduction of transistors.

There are two immediate advantages. First, a RISC processor core is less expensive to manufacture because its surface area is smaller. More RISC processor cores can fit on a silicon wafer than a comparable CISC processor. Second, a RISC processor consumes less power because the number of transistors is reduced.

But what about processing speed? It is true that a CISC core is usually slightly faster than a comparable RISC core when instructions are examined locally. In other words, one can easily argue that since a CISC instruction like this requires \( n \) RISC instructions to accomplish the same thing, the total CPU cycles needed by a RISC core is more than the total CPU cycles needed by a CISC core.

However, this argument does not always translate to matching performance differential in real life. This is because real life performance also depends on over all code generation and compiler optimization. A RISC core that has the same size (number of transistors and area) as a CISC core usually has more registers. The increased number of registers makes it possible for a compiler to further optimize code to reuse a register whenever possible.

Even more importantly, the latest bottleneck of data centers is power. In other words, many data centers are already consuming power at the limit. This shifts the focus from performance per chip to performance per watt. Some studies have pointed out that even cell phone processors (all are RISC) have very good performance per watt ratios, and may be suitable for data center applications.

5.7 Other addressing modes

The i386 architecture supports a variety of addressing modes in addition to “immediate”, “register”, “direct” and “indirect”. However, these addressing modes can be emulated by the other addressing modes, combined with arithmatic instructions. Keep this in mind when you read the following addressing modes.

5.8 Based addressing mode

An based operand is specified by a “displacement” or “offset” and a register. The register is known as a base register in this type of addressing. The sum of the displacement and the base register value becomes an “effective address”, and the content at that address becomes the value of the operand.

For example, let us take a look at the following instruction:

\[
\text{movb } $0,2(%eax)
\]

The destination operand, \( 2(%eax) \) means that the destination of the \text{movb} \ instruction is two byte higher than the address stored in register \text{eax}.

This addressing mode becomes very useful when we need to access items that are at a fixed displacement from a base address. We will use this addressing mode very often when we get to parameters and local variables of subroutines.

5.9 Based indexed addressing mode

A based-indexed operand is like a based operand, except it specifies one additional register called an index register. The sum of the displacement, the value of the base register and the value of the index register becomes an effective address. The value of the effective address is the operand.

The following instruction is an example:

\[
\text{movl \ 2(%ebx,%edx), \ %ecx}
\]

The source operand in this example is the value located at an effective address that is the sume of 2, the value of \text{ebx} and the value of \text{edx}.
5.10 Based scaled indexed addressing mode

This is similar to a based-indexed operand, except a scale is also specified. The scale is an integer that is 1, 2 or 4. The operand is the value at the effective address computed as a sum of a displacement, a base register and the product of the scale and an index register.

Observe the following example:

\texttt{movw} \ $25,-4(%ebp,%eax,2)

In this example, the effective address of the destination is four bytes before the address specified by the sum of \texttt{ebp} and the product of 2 and \texttt{eax}.