

Today we will learn about how the 68000 views memory, and about the syntax of the assembly language for the 68000. This material is all covered in Chapter 2 of the textbook. Here is an outline of the material for today.

Data Types  
Memory model  
Big-endian vs. Little-endian  
Memory maps  
Compile-Link-Load vs. Assemble-Load  
Source code notations and format  
Types of Instructions

**Sizes of data:**

The 68000 can operate on data in sizes of 32 bits, 16 bits, 8 bits and 1 bit.

- A 32 bit value is called a long or long-word. 32 bits can represent  $2^{32}$  different values, or roughly 4 billion. In the assembly language, long-word operations have a .L suffix.
- A 16 bit value is called a word. 16 bits can represent  $2^{16}$  different values, or 64K. The actual limits for unsigned 16 bit values are 0 to 65,535. For signed values, the limits are -32,768 to 32,767. In the assembly language, word operations have a .W suffix.
- An 8 bit value is called a byte. 8 bits can represent  $2^8$  different values or 256. The range of values goes from 0 to 255. In the
- A 1 bit value is simply called a bit. Within a byte, word, or long-word, each bit has an index. On the 68000, the bit in the one's place, or the low order bit, is bit 0. The high bit in a byte, is bit number 8, in a word is bit number 15, and in a long is bit number 31.

**A note on binary values:**

$2^0 = 1$	$2^4 = 16$
$2^5 = 32$	$2^8 = 256$
$2^{10} = 1K (1024)$	
$2^{15} = 32K (32768)$	$2^{16} = 64K (65536)$
$2^{20} = 1M (1,048,576)$	$2^{24} = 16M (16,777,216)$
$2^{25} = 32M (33,554,432)$	
$2^{30} = 1G (1,073,741,824)$	$2^{32} = 4G (4,294,967,296)$

The table above shows different binary numbers together with their short hand and actual decimal equivalents. The column on the left shows a pattern that provides an easy way to remember the powers of 2. Note that 2 to multiples of 10 correspond to the powers of 10 that we normally use for counting (1K, 1M, 1G). The in-between values of 2 raised to a multiple of 5 are always 32 times the previous 2 raised to a multiple of 10. Thus the

pattern is 1-32-1-32-1-32-1 ... Can you guess what  $2^{35}$  is? Memorize this pattern!!!  
The column on the left shows values of significance on the 68000. Data comes in 8, 16, or 32 bits. Addresses are 24 bits.

### Addressing in the 68000

- Data is stored in memory. Each location in memory that can hold data has an address. Think of it as being like the addresses of houses on a street, or perhaps of rooms along a hallway. Addresses are numbered consecutively from 0. Data in memory is referenced by giving its address on the address lines of the bus. The data that is stored at that address is then accessed by the data lines of the bus.
- The 68000 uses byte addressing. That means that each address is the address of a byte, and adding 1 to that address gives the address of the next byte.
- The 68000 uses 24 bit addresses. That means it can address up to  $2^{24}$  different bytes in memory, or 16M bytes.
- Addresses are normally written in Hex or base 16. In C and C++, Hex values are preceded by 0x as in 0x000008. In 68000 assembly, Hex values are preceded by a dollar sign, as in \$000008. Note that each hex digit takes 4 bits, so a 24 bit address uses 6 hex digits.

On the 68000, the address of a word is the address of the high order byte. See figure 2.5 on page 26 in the book. If \$000008 is the address of a word, \$000009 is the address of its low order byte and \$00000A is the address of the next word. If \$000008 is the address of a long, \$000008 is the address of its high order byte, \$00000B is the address of its low order byte, and \$00000C is the address of the next long. This kind of addressing, where the address of the value is the same as the address of its high order byte, is called big-endian. This may seem confusing. But it is actually less confusing than the other choice, little-endian. Little-endian is used by the Intel x86 processors.

### The Memory Map:

- Not every address in memory needs to have actual memory with real data. Large ranges of the available memory addresses can be un-allocated. Each memory chip is hardwired to a region of the address space. That means it has a given starting address, and uses offsets from that address to access the data that it holds. If the starting address of a 64K byte memory chip is \$400000, then \$400000 is the address of its first byte and  $\$400000 + 64\text{K}$  (or  $2^{16}$ ) is the address beyond its end.

\$64K is  $2 * 32\text{K}$ . Remember from the above pattern that  $2^{15}$  is 32K. So 64K is  $2^{16}$ . Now to convert to Hex, look at the actual binary pattern.  $2^{16}$  is 1 followed by 16 zeroes. Since each 4 bits is a hex digit, the 16 zeroes will be 4 Hex zeroes. The Hex value for  $2^{16}$ , or %1000000000000000, is \$10000.  $\$400000 + \$10000 = \$410000$ . Since \$410000 is just beyond the end of the chip, its last address is 1 less, or \$40FFFF. Please study this method of computing addresses carefully. We will be using it throughout the class. We will practice it more. The first quiz will have several questions like this.

- Figure 2.7 on page 27 in the book shows how different ranges of addresses were used on the first Macintosh. This is called a memory map. Here is a Polish web site that reprints the sidebar from the 1984 Byte magazine article that explains the

Macintosh memory map in more detail – including what VIA, IWM, and SCC are all about.

<http://www.aci.com.pl/mwichary/computerhistory/articles/macintoshbytepreview/memorymap>

In the lecture, I may also talk about different kinds of memory (i.e., core, RAM, ROM, PROM, DRAM, SRAM, Flash, and MRAM).

### **Assembly Language Programming:**

When we program in most languages, we think of the programming process as having four steps: Edit, Compile, Link, and Execute. A compiler reads the code in a process called “parsing” and then translates it into an equivalent, machine understandable form. Actually it translates it into a nearly machine understandable form with annotations. The program is then combined with other code from library files to create an executable file. The executable is loaded, from disk storage, into the computer’s memory. Finally, the computer is told to start execution at the program’s starting address.

Assembly programming works the same, except the whole process involves code that is much closer to what the machine understands. Instead of a compiler, which may do some major translation, we use an assembler, which only needs to do a very shallow translation to arrive at an executable form without the linking annotation. We then load the program into memory. In general, each line of assembly describes exactly one machine instruction. Here is an example:

```
MOV.L    D1,D2
```

This instruction copies the entire 32 bit contents from the D1 data register into the D2 data register. MOV is short for Move, but the semantics is actually to copy, since the contents of the D1 register are not changed.

Syntax refers to the order, and kinds of words and symbols in a valid line.

Semantics refers to what it means, in other words, what it will actually do. Compilers and assemblers can detect syntax errors – lines that are not well formed according to the rules of the language. But they cannot detect semantic errors – whether or not it does the right thing.

Each line starts with a space or tab, followed by the name of an operation, followed by another space or tab, followed by 0, 1, or 2 arguments (called operands). Most instructions take 2 operands separated by a comma.

The 68000 is a 2 operand machine. In an operation that takes two values as input, one of the result will be written over the second operand. In other words, the second operand is both an input and an output. For example, the instruction:

```
ADD.W    D2,D5
```

The low order word in the D2 data register is added to the low order word in the D5 register. The result is then written to the low order word of the D5 register, replacing its original contents.

Table 2.1 on page 29 gives a quick look at the types of instructions on the 68000.