

## **Volume Two: An Introduction to Machine Architecture**

- Chapter One:    System Organization  
                  A gentle introduction to the components that make up a typical PC.
- Chapter Two:    Memory Access and Organization  
                  A discussion of the 80x86 memory addressing modes and how HLA organizes your data in memory.
- Chapter Three:   Introduction to Digital Design  
                  A low-level description of how computer designers build CPUs and other system components.
- Chapter Four:    CPU Architecture  
                  A look at the internal operation of the CPU.
- Chapter Five:    Instruction Set Architecture  
                  This chapter describes how Intel's engineers designed the 80x86 instruction set. It also explains many of their design decisions, good and bad.
- Chapter Six:     Memory Architecture  
                  How memory is organized for high performance computing systems.
- Chapter Seven:   The I/O Subsystem  
                  Input and output are two of the most important functions on a PC. This chapter describes how input and output occurs on a typical 80x86 system.
- Chapter Eight:   Questions, Projects, and Laboratory Exercises  
                  See what you've learned in this topic!

This topic, as its title suggests, is primarily targeted towards a machine organization course. Those who wish to study assembly language programming should at least read Chapter Two and possibly Chapter One. Chapter Three is a low-level discussion of digital logic. This information is important to those who are interested in design-

**Volume Two:**

**Machine Architecture**

ing CPUs and other system components. Those individuals who are main interested in programming can safely skip this chapter. Chapters Four, Five, and Six provide a more in-depth look at computer systems' architecture. Those wanting to know how things work "under the hood" will want to read these chapters. However, programmers who just want to learn assembly language programming can safely skip these chapters. Chapter Seven discusses I/O on the 80x86. Under Win32 you will not be able to utilize much of this information unless you are writing device drivers. However, those interested in learning how low-level I/O takes place in assembly language will want to read this chapter.

---

# System Organization

# Chapter One

To write even a modest 80x86 assembly language program requires considerable familiarity with the 80x86 family. To write *good* assembly language programs requires a strong knowledge of the underlying hardware. Unfortunately, the underlying hardware is not consistent. Techniques that are crucial for 8088 programs may not be useful on Pentium systems. Likewise, programming techniques that provide big performance boosts on the Pentium chip may not help at all on an 80486. Fortunately, some programming techniques work well no matter which microprocessor you're using. This chapter discusses the effect hardware has on the performance of computer software.

---

## 1.1 Chapter Overview

This chapter describes the basic components that make up a computer system: the CPU, memory, I/O, and the bus that connects them. Although you can write software that is ignorant of these concepts, high performance software requires a complete understanding of this material. This chapter also discusses the 80x86 memory addressing modes and how you access memory data from your programs.

This chapter begins by discussing bus organization and memory organization. These two hardware components will probably have a bigger performance impact on your software than the CPU's speed. Understanding the organization of the system bus will allow you to design data structures and algorithms that operate at maximum speed. Similarly, knowing about memory performance characteristics, data locality, and cache operation can help you design software that runs as fast as possible. Of course, if you're not interested in writing code that runs as fast as possible, you can skip this discussion; however, most people do care about speed at one point or another, so learning this information is useful.

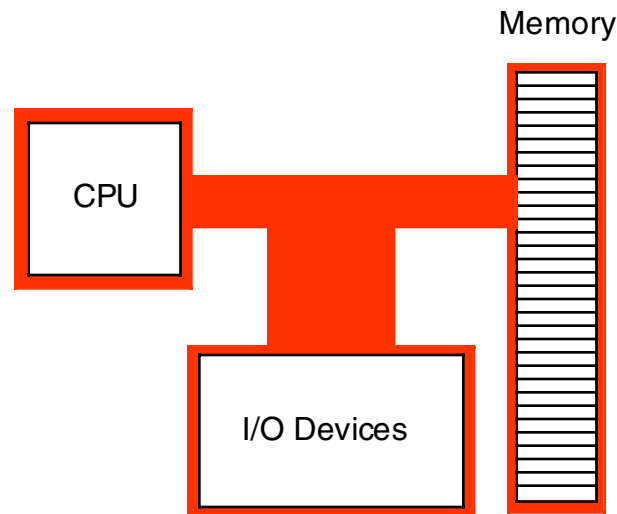
With the generic hardware issues out of the way, this chapter then discusses the program-visible components of the memory architecture - specifically the 80x86 addressing modes and how a program can access memory. In addition to the addressing modes, this chapter introduces several new 80x86 instructions that are quite useful for manipulating memory. This chapter also presents several new HLA Standard Library calls you can use to allocate and deallocate memory.

Some might argue that this chapter gets too involved with computer architecture. They feel such material should appear in an architectural book, not an assembly language programming book. This couldn't be farther from the truth! Writing *good* assembly language programs requires a strong knowledge of the architecture. Hence the emphasis on computer architecture in this chapter.

---

## 1.2 The Basic System Components

The basic operational design of a computer system is called its *architecture*. John Von Neumann, a pioneer in computer design, is given credit for the architecture of most computers in use today. For example, the 80x86 family uses the *Von Neumann architecture* (VNA). A typical Von Neumann system has three major components: the *central processing unit* (or *CPU*), *memory*, and *input/output* (or *I/O*). The way a system designer combines these components impacts system performance (See Figure 1.1).



**Figure 1.1 Typical Von Neumann Machine**

In VNA machines, like the 80x86 family, the CPU is where all the action takes place. All computations occur inside the CPU. Data and machine instructions reside in memory until required by the CPU. To the CPU, most I/O devices look like memory because the CPU can store data to an output device and read data from an input device. The major difference between memory and I/O locations is the fact that I/O locations are generally associated with external devices in the outside world.

## 1.2.1 The System Bus

The *system bus* connects the various components of a VNA machine. The 80x86 family has three major busses: the *address bus*, the *data bus*, and the *control bus*. A bus is a collection of wires on which electrical signals pass between components in the system. These busses vary from processor to processor. However, each bus carries comparable information on all processors; e.g., the data bus may have a different implementation on the 80386 than on the 8088, but both carry data between the processor, I/O, and memory.

A typical 80x86 system component uses *standard TTL logic levels*<sup>1</sup>. This means each wire on a bus uses a standard voltage level to represent zero and one<sup>2</sup>. We will always specify zero and one rather than the electrical levels because these levels vary on different processors (especially laptops).

### 1.2.1.1 The Data Bus

The 80x86 processors use the *data bus* to shuffle data between the various components in a computer system. The size of this bus varies widely in the 80x86 family. Indeed, this bus defines the “size” of the processor.

On typical 80x86 systems, the data bus contains eight, 16, 32, or 64 lines. The 8088 and 80188 microprocessors have an eight bit data bus (eight data lines). The 8086, 80186, 80286, and 80386SX processors have a 16 bit data bus. The 80386DX, 80486, and Pentium Overdrive™ processors have a 32 bit data bus.

1. Actually, newer members of the family tend to use lower voltage signals, but these remain compatible with TTL signals.

2. TTL logic represents the value zero with a voltage in the range 0.0-0.8v. It represents a one with a voltage in the range 2.4-5v. If the signal on a bus line is between 0.8v and 2.4v, it's value is indeterminate. Such a condition should only exist when a bus line is changing from one state to the other.

The Pentium™, Pentium Pro, Pentium II, Pentium III, and Pentium IV processors have a 64 bit data bus. Future versions of the chip may have a larger bus.

Having an eight bit data bus does not limit the processor to eight bit data types. It simply means that the processor can only access one byte of data per memory cycle (see “The Memory Subsystem” on page 133 for a description of memory cycles). Therefore, the eight bit bus on an 8088 can only transmit half the information per unit time (memory cycle) as the 16 bit bus on the 8086. Therefore, processors with a 16 bit bus are naturally faster than processors with an eight bit bus. Likewise, processors with a 32 bit bus are faster than those with a 16 or eight bit data bus. The size of the data bus affects the performance of the system more than the size of any other bus.

You’ll often hear a processor called an *eight, 16, 32, or 64 bit processor*. While there is a mild controversy concerning the size of a processor, most people now agree that the maximum of either the number of data lines on the processor or the size of the largest general purpose integer register determines the processor size. Since the 80x86 family busses are eight, 16, 32, or 64 bits wide, most data accesses are also eight, 16, 32, or 64 bits. Therefore, the Pentium CPUs are still 32-bit CPUs even though they have a 64-bit data bus.

Although it is possible to process 12 bit data with an 80x86, most programmers process 16 bits since the processor will fetch and manipulate 16 bits anyway. This is because the processor always fetches data in groups of eight bits. To fetch 12 bits requires two eight bit memory operations. Since the processor fetches 16 bits rather than 12, most programmers use all 16 bits. In general, manipulating data which is eight, 16, 32, or 64 bits in length is the most efficient.

Although the 80x86 family members with 16, 32, and 64 bit data busses *can* process data up to the width of the bus, they can also access smaller memory units of eight, 16, or 32 bits. Therefore, anything you can do with a small data bus can be done with a larger data bus as well; the larger data bus, however, may access memory faster and can access larger chunks of data in one memory operation. You’ll read about the exact nature of these memory accesses a little later (see “The Memory Subsystem” on page 133).

**Table 12: 80x86 Processor Data Bus Sizes**

Processor	Data Bus Size
8088	8
80188	8
8086	16
80186	16
80286	16
80386sx	16
80386dx	32
80486	32
80586 class / Pentium family	64

### 1.2.1.2 The Address Bus

The data bus on an 80x86 family processor transfers information between a particular memory location or I/O device and the CPU. The only question is, “Which memory location or I/O device?” The address bus answers that question. To differentiate memory locations and I/O devices, the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device, it places the corresponding address on the address bus. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to

read the data from or place data on to the data bus. In either case, all other memory locations ignore the request. Only the device whose address matches the value on the address bus responds.

With a single address line, a processor could create exactly two unique addresses: zero and one. With  $n$  address lines, the processor can provide  $2^n$  unique addresses (since there are  $2^n$  unique values in an  $n$ -bit binary number). Therefore, the number of bits on the address bus will determine the *maximum* number of addressable memory and I/O locations. The 8088 and 8086, for example, have 20 bit address busses. Therefore, they can access up to 1,048,576 (or  $2^{20}$ ) memory locations. Larger address busses can access more memory. The 8088 and 8086 suffer from an anemic address space<sup>3</sup> – their address bus is too small. Intel corrected this in later processors.

**Table 13: 80x86 Family Address Bus Sizes**

Processor	Address Bus Size	Max Addressable Memory	In English!
8088	20	1,048,576	One Megabyte
8086	20	1,048,576	One Megabyte
80188	20	1,048,576	One Megabyte
80186	20	1,048,576	One Megabyte
80286	24	16,777,216	Sixteen Megabytes
80386sx	24	16,777,216	Sixteen Megabytes
80386dx	32	4,294,976,296	Four Gigabytes
80486	32	4,294,976,296	Four Gigabytes
80586 / Pentium	32	4,294,976,296	Four Gigabytes
Pentium Pro	36	68,719,476,736	64 Gigabytes
Pentium II	36	68,719,476,736	64 Gigabytes
Pentium III	36	68,719,476,736	64 Gigabytes

Future 80x86 processors will probably support 40, 48, and 64-bit address busses. The time is coming when most programmers will consider four gigabytes of storage to be too small, much like they consider one megabyte insufficient today. (There was a time when one megabyte was considered far more than anyone would ever need!).

---

### 1.2.1.3 The Control Bus

The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. Consider for a moment the data bus. The CPU sends data to memory and receives data from memory on the data bus. This prompts the question, “Is it sending or receiving?” There are two lines on the control bus, *read* and *write*, which specify the direction of data flow. Other signals include system clocks, interrupt lines, status lines, and so on. The exact make up of the control bus varies among processors in the 80x86 family. However, some control lines are common to all processors and are worth a brief mention.

The *read* and *write* control lines control the direction of data on the data bus. When both contain a logic one, the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero), the CPU is reading data from memory (that is, the system is transferring data from memory to the CPU). If the write line is low, the system transfers data from the CPU to memory.

---

3. The address space is the set of all addressable memory locations.

The *byte enable lines* are another set of important control lines. These control lines allow 16, 32, and 64 bit processors to deal with smaller chunks of data. Additional details appear in the next section.

The 80x86 family, unlike many other processors, provides two distinct address spaces: one for memory and one for I/O. While the memory address busses on various 80x86 processors vary in size, the I/O address bus on all 80x86 CPUs is 16 bits wide. This allows the processor to address up to 65,536 different I/O *locations*. As it turns out, most devices (like the keyboard, printer, disk drives, etc.) require more than one I/O location. Nonetheless, 65,536 I/O locations are more than sufficient for most applications. The original IBM PC design only allowed the use of 1,024 of these.

Although the 80x86 family supports two address spaces, it does not have two address busses (for I/O and memory). Instead, the system shares the address bus for both I/O and memory addresses. Additional control lines decide whether the address is intended for memory or I/O. When such signals are active, the I/O devices use the address on the L.O. 16 bits of the address bus. When inactive, the I/O devices ignore the signals on the address bus (the memory subsystem takes over at that point).

## 1.2.2 The Memory Subsystem

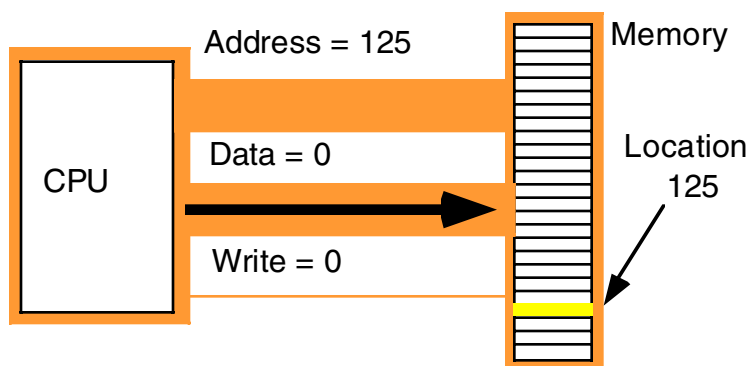
A typical 80x86 processor addresses a maximum of  $2^n$  different memory locations, where  $n$  is the number of bits on the address bus<sup>4</sup>. As you've seen already, 80x86 processors have 20, 24, 32, and 36 bit address busses (with 64 bits on the way).

Of course, the first question you should ask is, "What exactly is a memory location?" The 80x86 supports *byte addressable memory*. Therefore, the basic memory unit is a byte. So with 20, 24, 32, and 36 address lines, the 80x86 processors can address one megabyte, 16 megabytes, four gigabytes, and 64 gigabytes of memory, respectively.

Think of memory as a linear array of bytes. The address of the first byte is zero and the address of the last byte is  $2^n - 1$ . For an 8088 with a 20 bit address bus, the following pseudo-Pascal array declaration is a good approximation of memory:

Memory: array [0..1048575] of byte;

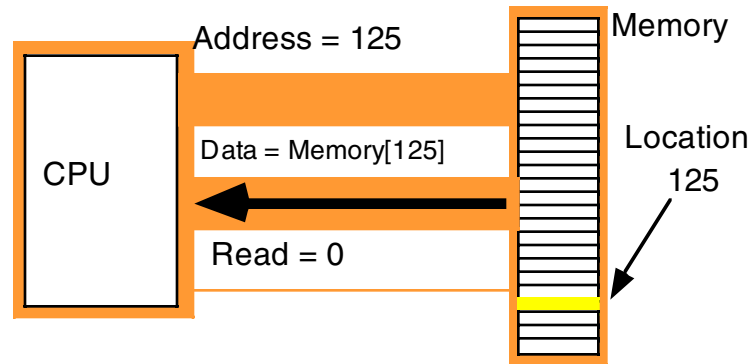
To execute the equivalent of the Pascal statement "Memory [125] := 0;" the CPU places the value zero on the data bus, the address 125 on the address bus, and asserts the write line (since the CPU is writing data to memory), see Figure 1.2.



**Figure 1.2** Memory Write Operation

4. This is the *maximum*. Most computer systems built around 80x86 family do not include the maximum addressable amount of memory.

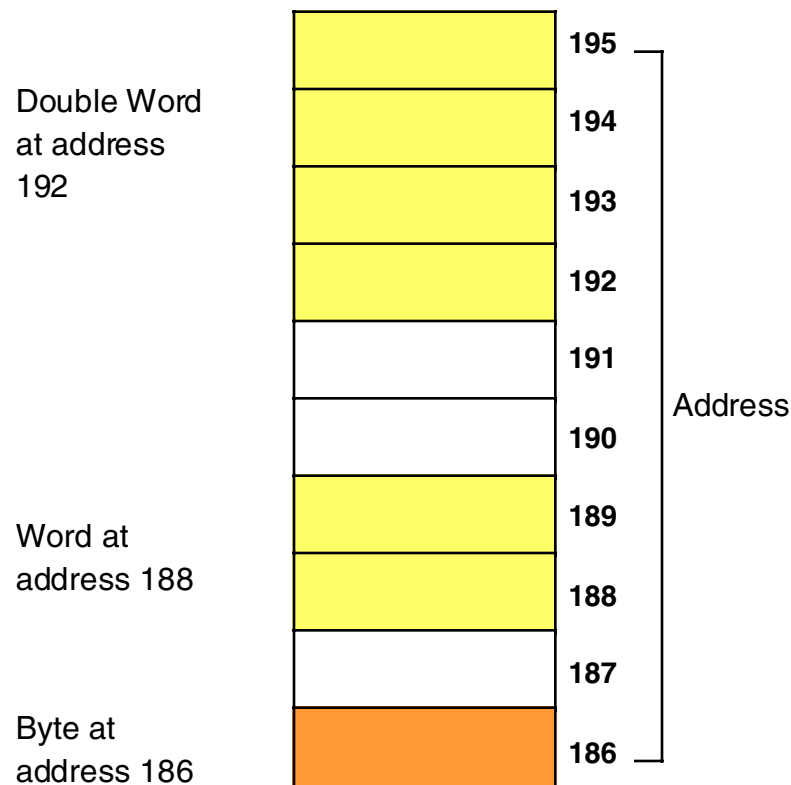
To execute the equivalent of “CPU := Memory [125];” the CPU places the address 125 on the address bus, asserts the read line (since the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 1.3).



**Figure 1.3**      **Memory Read Operation**

The above discussion applies *only* when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Since memory consists of an array of bytes, how can we possibly deal with values larger than eight bits?

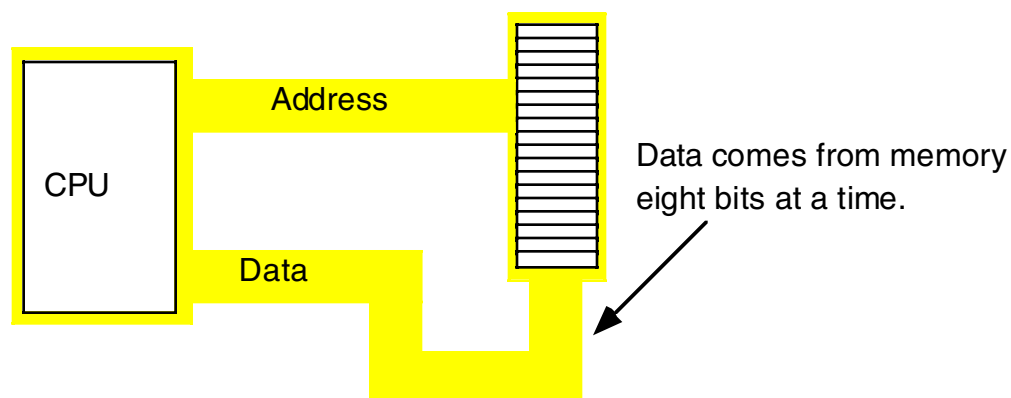
Different computer systems have different solutions to this problem. The 80x86 family deals with this problem by storing the L.O. byte of a word at the address specified and the H.O. byte at the next location. Therefore, a word consumes two consecutive memory addresses (as you would expect, since a word consists of two bytes). Similarly, a double word consumes four consecutive memory locations. The address for the double word is the address of its L.O. byte. The remaining three bytes follow this L.O. byte, with the H.O. byte appearing at the address of the double word *plus three* (see Figure 1.4). Bytes, words, and double words may begin at *any* valid address in memory. We will soon see, however, that starting larger objects at an arbitrary address is not a good idea.



**Figure 1.4** Byte, Word, and DWord Storage in Memory

Note that it is quite possible for byte, word, and double word values to overlap in memory. For example, in Figure 1.4 you could have a word variable beginning at address 193, a byte variable at address 194, and a double word value beginning at address 192. These variables would all overlap.

The 8088 and 80188 microprocessors have an eight bit data bus. This means that the CPU can transfer eight bits of data at a time. Since each memory address corresponds to an eight bit byte, this turns out to be the most convenient arrangement (from the hardware perspective), see Figure 1.5.



**Figure 1.5** Eight-Bit CPU <-> Memory Interface

The term “byte addressable memory array” means that the CPU can address memory in chunks as small as a single byte. It also means that this is the *smallest* unit of memory you can access at once with the processor. That is, if the processor wants to access a four bit value, it must read eight bits and then ignore the extra four bits. Also realize that byte addressability does not imply that the CPU can access eight bits on any arbitrary bit boundary. When you specify address 125 in memory, you get the entire eight bits at that address, nothing less, nothing more. Addresses are integers; you cannot, for example, specify address 125.5 to fetch fewer than eight bits.

The 8088 and 80188 can manipulate word and double word values, even with their eight bit data bus. However, this requires multiple memory operations because these processors can only move eight bits of data at once. To load a word requires two memory operations; to load a double word requires four memory operations.

The 8086, 80186, 80286, and 80386sx processors have a 16 bit data bus. This allows these processors to access twice as much memory in the same amount of time as their eight bit brethren. These processors organize memory into two *banks*: an “even” bank and an “odd” bank (see Figure 1.6). Figure 1.7 illustrates the connection to the CPU (D0-D7 denotes the L.O. byte of the data bus, D8-D15 denotes the H.O. byte of the data bus):

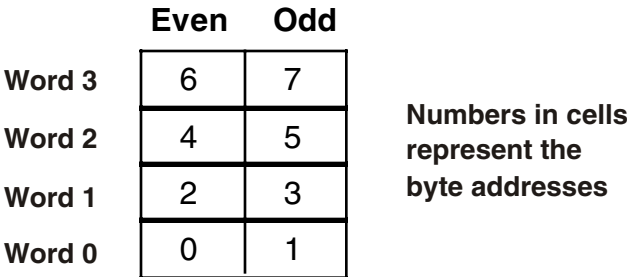


Figure 1.6      Byte Addressing in Word Memory

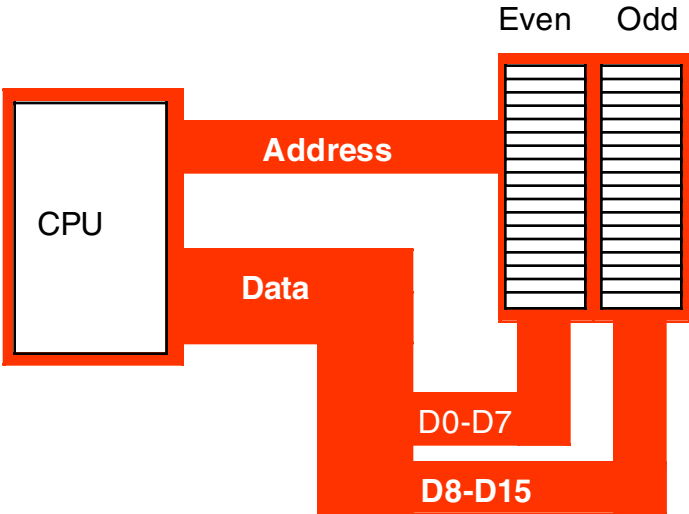


Figure 1.7      Sixteen-Bit Processor (8086, 80186, 80286, 80386sx) Memory Organization

The 16 bit members of the 80x86 family can load a word from any arbitrary address. As mentioned earlier, the processor fetches the L.O. byte of the value from the address specified and the H.O. byte from the next consecutive address. This creates a subtle problem if you look closely at the diagram above. What happens when you access a word on an odd address? Suppose you want to read a word from location 125. Okay, the L.O. byte of the word comes from location 125 and the H.O. word comes from location 126. What's the big deal? It turns out that there are two problems with this approach.

First, look again at Figure 1.7. Data bus lines eight through 15 (the H.O. byte) connect to the odd bank, and data bus lines zero through seven (the L.O. byte) connect to the even bank. Accessing memory location 125 will transfer data to the CPU on the H.O. byte of the data bus; yet we want this data in the L.O. byte! Fortunately, the 80x86 CPUs recognize this situation and automatically transfer the data on D8-D15 to the L.O. byte.

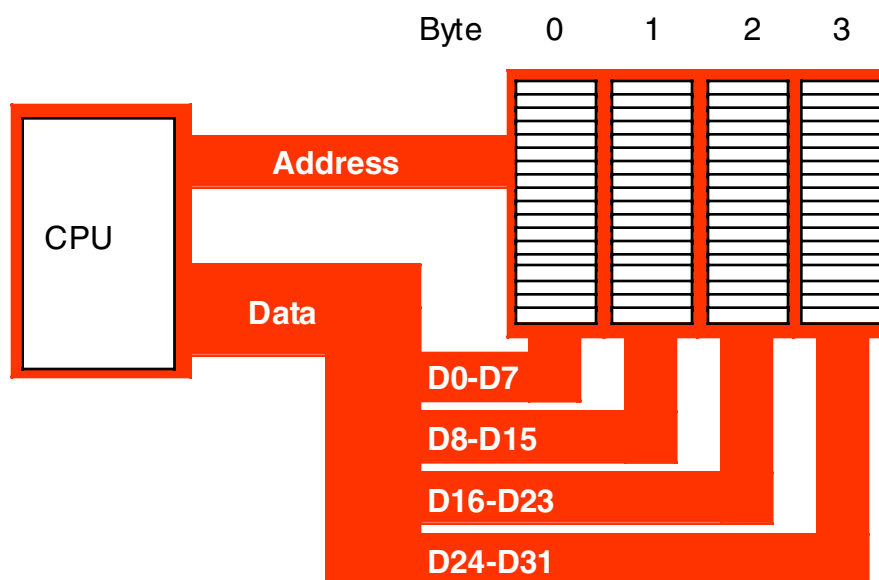
The second problem is even more obscure. When accessing words, we're really accessing two separate bytes, each of which has its own byte address. So the question arises, "What address appears on the address bus?" The 16 bit 80x86 CPUs always place even addresses on the bus. Even bytes always appear on data lines D0-D7 and the odd bytes always appear on data lines D8-D15. If you access a word at an even address, the CPU can bring in the entire 16 bit chunk in one memory operation. Likewise, if you access a single byte, the CPU activates the appropriate bank (using a "byte enable" control line). If the byte appeared at an odd address, the CPU will automatically move it from the H.O. byte on the bus to the L.O. byte.

So what happens when the CPU accesses a *word* at an odd address, like the example given earlier? Well, the CPU cannot place the address 125 onto the address bus and read the 16 bits from memory. There are no odd addresses coming out of a 16 bit 80x86 CPU. The addresses are always even. So if you try to put 125 on the address bus, this will put 124 on to the address bus. Were you to read the 16 bits at this address, you would get the word at addresses 124 (L.O. byte) and 125 (H.O. byte) – not what you'd expect. Accessing a word at an odd address requires two memory operations. First the CPU must read the byte at address 125, then it needs to read the byte at address 126. Finally, it needs to swap the positions of these bytes internally since both entered the CPU on the wrong half of the data bus.

Fortunately, the 16 bit 80x86 CPUs hide these details from you. Your programs can access words at *any* address and the CPU will properly access and swap (if necessary) the data in memory. However, to access a word at an odd address requires two memory operations (just like the 8088/80188). Therefore, accessing words at odd addresses on a 16 bit processor is slower than accessing words at even addresses. **By carefully arranging how you use memory, you can improve the speed of your program.**

Accessing 32 bit quantities always takes at least two memory operations on the 16 bit processors. If you access a 32 bit quantity at an odd address, the processor will require three memory operations to access the data.

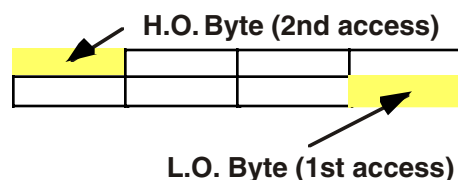
The 32 bit 80x86 processors (the 80386, 80486, and Pentium Overdrive) use four banks of memory connected to the 32 bit data bus (see Figure 1.8).



**Figure 1.8 32-Bit Processor (80386, 80486, Pentium Overdrive) Memory Organization**

The address placed on the address bus is always some multiple of four. Using various “byte enable” lines, the CPU can select which of the four bytes at that address the software wants to access. As with the 16 bit processor, the CPU will automatically rearrange bytes as necessary.

With a 32 bit memory interface, the 80x86 CPU can access any byte with one memory operation. If (address MOD 4) does not equal three, then a 32 bit CPU can access a word at that address using a single memory operation. However, if the remainder is three, then it will take two memory operations to access that word (see Figure 1.9). This is the same problem encountered with the 16 bit processor, except it occurs half as often.



**Figure 1.9 Accessing a Word at (Address mod 4) = 3.**

A 32 bit CPU can access a double word in a single memory operation *if* the address of that value is evenly divisible by four. If not, the CPU will require two memory operations.

Once again, the CPU handles all of this automatically. In terms of loading correct data the CPU handles everything for you. However, there is a performance benefit to proper data alignment. As a general rule you should always place word values at even addresses and double word values at addresses which are evenly divisible by four. This will speed up your program.

### 1.2.3 The I/O Subsystem

Besides the 20, 24, or 32 address lines which access memory, the 80x86 family provides a 16 bit I/O address bus. This gives the 80x86 CPUs two separate address spaces: one for memory and one for I/O operations. Lines on the control bus differentiate between memory and I/O addresses. Other than separate control lines and a smaller bus, I/O addressing behaves exactly like memory addressing. Memory and I/O devices both share the same data bus and the L.O. 16 lines on the address bus.

There are three limitations to the I/O subsystem on the PC: first, the 80x86 CPUs require special instructions to access I/O devices; second, the designers of the PC used the “best” I/O locations for their own purposes, forcing third party developers to use less accessible locations; third, 80x86 systems can address no more than 65,536 ( $2^{16}$ ) I/O addresses. When you consider that a typical VGA display card requires over 128,000 different locations, you can see a problem with the size of I/O bus.

Fortunately, hardware designers can map their I/O devices into the memory address space as easily as they can the I/O address space. So by using the appropriate circuitry, they can make their I/O devices look just like memory. This is how, for example, display adapters on the PC work.

## 1.3 HLA Support for Data Alignment

In order to write the fastest running programs, you need to ensure that your data objects are properly aligned in memory. Data becomes misaligned whenever you allocate storage for different sized objects in adjacent memory locations. Since it is nearly impossible to write a (large) program that uses objects that are all the same size, some other facility is necessary in order to realign data that would normally be unaligned in memory.

Consider the following HLA variable declarations:

```
static
  dw:    dword;
  b:     byte;
  w:     word;
  dw2:   dword;
  w2:    word;
  b2:    byte;
  dw3:   dword;
```

The first static declaration in a program (running under Win32 and most 32-bit operating systems) places its variables at an address that is an even multiple of 4096 bytes. Since 4096 is a power of two, whatever variable first appears in the static declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of the sizes of all the preceding variables plus the starting address. Therefore, assuming the above variables are allocated at a starting address of 4096, then each variable will be allocated at the following addresses:

		// Start Adrs	Length
dw:	dword;	// 4096	4
b:	byte;	// 4100	1
w:	word;	// 4101	2
dw2:	dword;	// 4103	4
w2:	word;	// 4107	2
b2:	byte;	// 4109	1
dw3:	dword;	// 4110	4

With the exception of the first variable (which is aligned on a 4K boundary) and the byte variables (whose alignment doesn't matter), all of these variables are misaligned in memory. The *w*, *w2*, and *dw2* variables are aligned on odd addresses and the *dw3* variable is aligned on an even address that is not an even multiple of four.

An easy way to guarantee that your variables are aligned on an appropriate address is to put all the dword variables first, the word variables second, and the byte variables last in the declaration:

```
static
dw:    dword;
dw2:   dword;
dw3:   dword;
w:     word;
w2:    word;
b:     byte;
b2:    byte;
```

This organization produces the following addresses in memory (again, assuming the first variable is allocated at address 4096):

		// Start Adrs	Length
dw:	dword;	// 4096	4
dw2:	dword;	// 4100	4
dw3:	dword;	// 4104	4
w:	word;	// 4108	2
w2:	word;	// 4110	2
b:	byte;	// 4112	1
b2:	byte;	// 4113	1

As you can see, these variables are all aligned at reasonable addresses.

Unfortunately, it is rarely possible for you to arrange your variables in this manner. While there are lots of technical reasons that make this alignment impossible, a good practical reason for not doing this is because it doesn't let you organize your variable declarations by logical function (that is, you probably want to keep related variables next to one another regardless of their size).

To resolve this problem, HLA provides two solutions. The first is an alignment option whenever you encounter a *static* section. If you follow the static keyword by an integer constant inside parentheses, HLA will align the very next variable declaration at an address that is an even multiple of the specified constant, e.g.,

```
static( 4 )
dw:    dword;
b:     byte;
w:     word;
dw2:   dword;
w2:    word;
b2:    byte;
dw3:   dword;
```

Of course, if you have only a single *static* section in your entire program, this declaration doesn't buy you much because the first declaration in the section is already aligned on a 4096 byte boundary. However, HLA does allow you to put multiple *static* sections into your program, so you can specify an alignment constant for each *static* section:

```
static( 4 )
dw:    dword;
b:     byte;

static( 2 )
w:     word;

static( 4 )
dw2:   dword;
w2:    word;
b2:    byte;
```

```
static( 4 )
    dw3:    dword;
```

This particular sequence guarantees that all double word variables are aligned on addresses that are multiples of four and all word variables are aligned on even addresses (note that a special section was not created for `w2` since its address is going to be an even multiple of four).

While the alignment parameter to the *static* directive is useful on occasion, there are two problems with it: The first problem is that inserting so many *static* directives into the middle of your variable declarations tends to disrupt the readability of your variable declarations. Part of this problem can be overcome by simply placing a *static* directive before every variable declaration:

```
static( 4 )    dw:    dword;
static( 1 )    b:     byte;
static( 2 )    w:     word;
static( 4 )    dw2:   dword;
static( 2 )    w2:    word;
static( 1 )    b2:    byte;
static( 4 )    dw3:   dword;
```

While this approach can, arguably, make a program easier to read, it certainly involves more typing and it doesn't address the second problem: variables appearing in separate *static* sections are not guaranteed to be allocated in adjacent memory locations. Once in a while it is very important to ensure that two variables are allocated in adjacent memory cells and most programmers assume that variables declared next to one another in the source code are allocated in adjacent memory cells. The mechanism above does not guarantee this.

The second facility HLA provides to help align adjacent memory locations is the *align* directive. The *align* directive uses the following syntax:

```
align( integer_constant );
```

The integer constant must be one of the following small unsigned integer values: 1, 2, 4, 8, 10, or 16. If HLA encounters the *align* directive in a *static* section, it will align the very next variable on an address that is an even multiple of the specified alignment constant. The previous example could be rewritten, using the *align* directive, as follows:

```
static( 4 )
    dw:    dword;
    b:     byte;
    align( 2 );
    w:     word;
    align( 4 );
    dw2:   dword;
    w2:    word;
    b2:    byte;
    align( 4 );
    dw3:   dword;
```

If you're wondering how the *align* directive works, it's really quite simple. If HLA determines that the current address is not an even multiple of the specified value, HLA will quietly emit extra bytes of padding after the previous variable declaration until the current address in the *static* section is an even multiple of the specified value. This has the effect of making your program slightly larger (by a few bytes) in exchange for faster access to your data; Given that your program will only grow by a small number of bytes when you use this feature, this is a good trade off.

## 1.4 System Timing

Although modern computers are quite fast and getting faster all the time, they still require a finite amount of time to accomplish even the smallest tasks. On Von Neumann machines like the 80x86, most operations are *serialized*. This means that the computer executes commands in a prescribed order. It wouldn't do, for example, to execute the statement  $I := I * 5 + 2$ ; before  $I := J$ ; in the following sequence:

```
I := J;
I := I * 5 + 2;
```

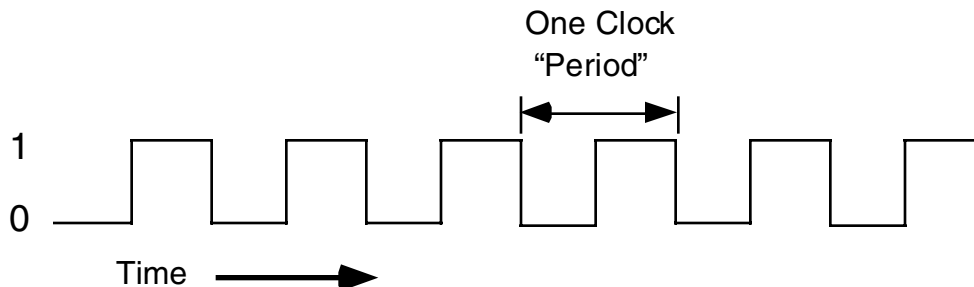
Clearly we need some way to control which statement executes first and which executes second.

Of course, on real computer systems, operations do not occur instantaneously. Moving a copy of J into I takes a certain amount of time. Likewise, multiplying I by five and then adding two and storing the result back into I takes time. As you might expect, the second Pascal statement above takes quite a bit longer to execute than the first. For those interested in writing fast software, a natural question to ask is, "How does the processor execute statements, and how do we measure how long they take to execute?"

The CPU is a very complex piece of circuitry. Without going into too many details, let us just say that operations inside the CPU must be very carefully coordinated or the CPU will produce erroneous results. To ensure that all operations occur at just the right moment, the 80x86 CPUs use an alternating signal called the *system clock*.

### 1.4.1 The System Clock

At the most basic level, the *system clock* handles all synchronization within a computer system. The system clock is an electrical signal on the control bus which alternates between zero and one at a periodic rate (see Figure 1.10). All activity within the CPU is synchronized with the edges (rising or falling) of this clock signal.



**Figure 1.10 The System Clock**

The frequency with which the system clock alternates between zero and one is the *system clock frequency*. The time it takes for the system clock to switch from zero to one and back to zero is the *clock period*. One full period is also called a *clock cycle*. On most modern systems, the system clock switches between zero and one at rates exceeding several hundred million times per second to several billion times per second. The clock frequency is simply the number of clock cycles which occur each second. A typical Pentium III chip, circa 2000, runs at speeds of 1 billion cycles per second or faster. "Hertz" (Hz) is the technical term meaning one cycle per second. Therefore, the aforementioned Pentium chip runs at 1000 million hertz, or 1000 megahertz (MHz), also known as one gigahertz. Typical frequencies for 80x86 parts range from 5 MHz up to several Gigahertz (GHz, or billions of cycles per second) and beyond. Note that one clock period (the amount of time for one complete clock cycle) is the reciprocal of the clock frequency. For example, a 1 MHz clock would have a clock period of one microsecond ( $1/1,000,000^{\text{th}}$  of a second). Likewise, a 10 MHz clock would have a clock period of 100 nanoseconds (100 billionths of a second). A CPU running at 1 GHz would

have a clock period of one nanosecond. Note that we usually express clock periods in millionths or billionths of a second.

To ensure synchronization, most CPUs start an operation on either the *falling edge* (when the clock goes from one to zero) or the *rising edge* (when the clock goes from zero to one). The system clock spends most of its time at either zero or one and very little time switching between the two. Therefore clock edge is the perfect synchronization point.

Since all CPU operations are synchronized around the clock, the CPU cannot perform tasks any faster than the clock. However, just because a CPU is running at some clock frequency doesn't mean that it is executing that many operations each second. Many operations take multiple clock cycles to complete so the CPU often performs operations at a significantly lower rate.

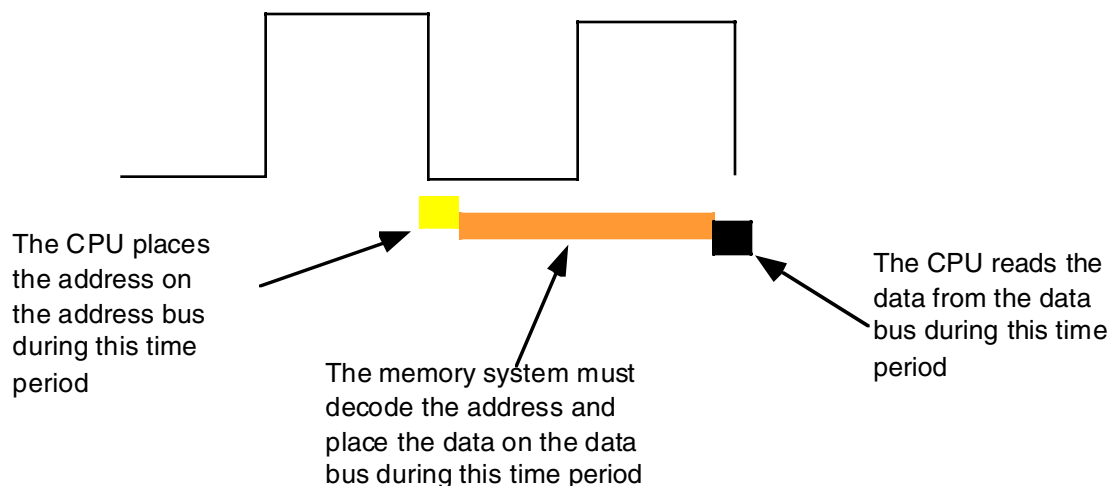
## 1.4.2 Memory Access and the System Clock

Memory access is one of the most common CPU activities. Memory access is definitely an operation synchronized around the system clock. That is, reading a value from memory or writing a value to memory occurs no more often than once every clock cycle. Indeed, on many 80x86 processors, it takes several clock cycles to access a memory location. The *memory access time* is the number of clock cycles the system requires to access a memory location; this is an important value since longer memory access times result in lower performance.

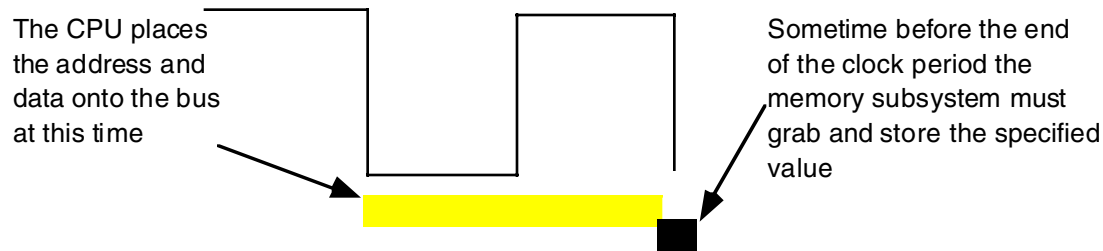
Different 80x86 processors have different memory access times ranging from one to four clock cycles. For example, the 8088 and 8086 CPUs require *four* clock cycles to access memory; the 80486 and later CPUs require only one. Therefore, the 80486 will execute programs that access memory faster than an 8086, even when running at the same clock frequency.

Memory access time is the amount of time between a memory operation request (read or write) and the time the memory operation completes. On a 5 MHz 8088/8086 CPU the memory access time is roughly 800 ns (nanoseconds). On a 50 MHz 80486, the memory access time is slightly less than 20 ns. Note that the memory access time for the 80486 is 40 times faster than the 8088/8086. This is because the 80486's clock frequency is ten times faster and it uses one-fourth the clock cycles to access memory.

When reading from memory, the memory access time is the amount of time from the point that the CPU places an address on the address bus and the CPU takes the data off the data bus. On an 80486 CPU with a one cycle memory access time, a read looks something like shown in Figure 1.11. Writing data to memory is similar (see Figure 1.12).



**Figure 1.11 The 80x86 Memory Read Cycle**



**Figure 1.12 The 80x86 Memory Write Cycle**

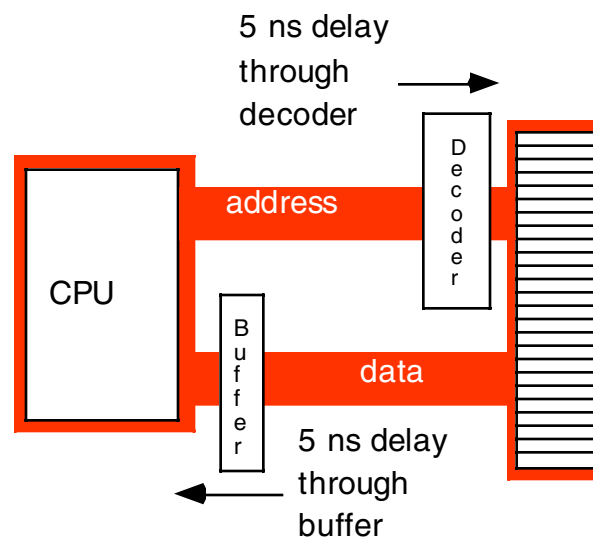
Note that the CPU doesn't wait for memory. The access time is specified by the clock frequency. If the memory subsystem doesn't work fast enough, the CPU will read garbage data on a memory read operation and will not properly store the data on a memory write operation. This will surely cause the system to fail.

Memory devices have various ratings, but the two major ones are capacity and speed (access time). Typical dynamic RAM (random access memory) devices have capacities of 512 (or more) megabytes and speeds of 5-100 ns. You can buy bigger or faster devices, but they are much more expensive. A typical 500 MHz Pentium system uses 10 ns memory devices.

Wait just a second here! At 500 MHz the clock period is roughly 2 ns. How can a system designer get away with using 10 ns memory? The answer is *wait states*.

### 1.4.3 Wait States

A wait state is nothing more than an extra clock cycle to give some device time to complete an operation. For example, a 50 MHz 80486 system has a 20 ns clock period. This implies that you need 20 ns memory. In fact, the situation is worse than this. In most computer systems there is additional circuitry between the CPU and memory: decoding and buffering logic. This additional circuitry introduces additional delays into the system (see Figure 1.13). In this diagram, the system loses 10ns to buffering and decoding. So if the CPU needs the data back in 20 ns, the memory must respond in less than 10 ns.

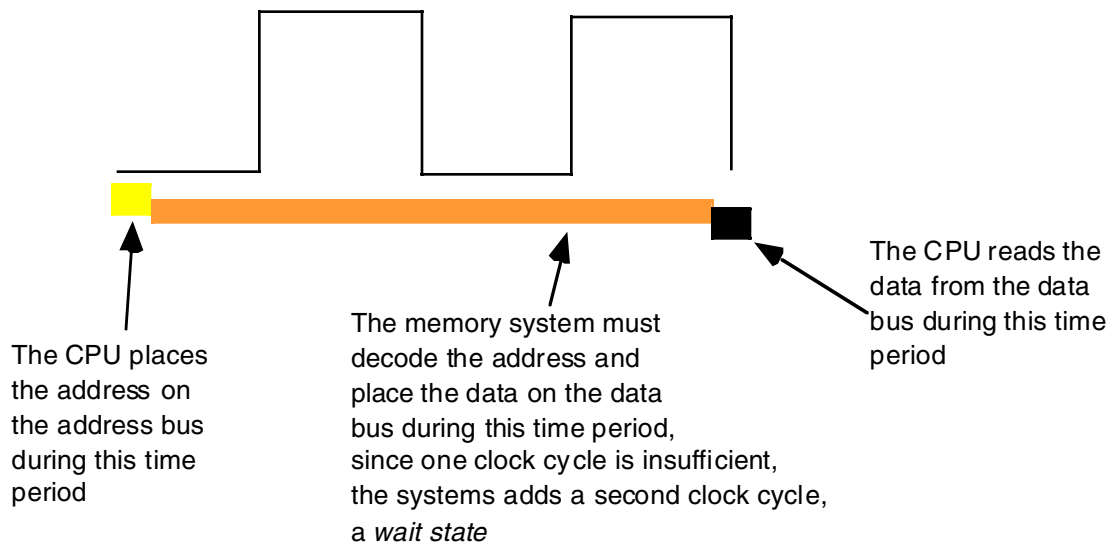


**Figure 1.13 Decoding and Buffer Delays**

You can actually buy 10ns static memory. However, it is very expensive, bulky, consumes a lot of power, and generates a lot of heat. These are bad attributes. Supercomputers use this type of memory. However, supercomputers also cost millions of dollars, take up entire rooms, require special cooling, and have giant power supplies. Not the kind of stuff you want sitting on your desk.

If cost-effective memory won't work with a fast processor, how do companies manage to sell fast PCs? One part of the answer is the wait state. For example, if you have a 20 MHz processor with a memory cycle time of 50 ns and you lose 10 ns to buffering and decoding, you'll need 40 ns memory. What if you can only afford 80 ns memory in a 20 MHz system? Adding a wait state to extend the memory cycle to 100 ns (two clock cycles) will solve this problem. Subtracting 10ns for the decoding and buffering leaves 90 ns. Therefore, 80 ns memory will respond well before the CPU requires the data.

Almost every general purpose CPU in existence provides a signal on the control bus to allow the insertion of wait states. Generally, the decoding circuitry asserts this line to delay one additional clock period, if necessary. This gives the memory sufficient access time, and the system works properly (see Figure 1.14).



**Figure 1.14** Inserting a Wait State into a Memory Read Operation

Sometimes a single wait state is not sufficient. Consider the 80486 running at 50 MHz. The normal memory cycle time is less than 20 ns. Therefore, less than 10 ns are available after subtracting decoding and buffering time. If you are using 60 ns memory in the system, adding a single wait state will not do the trick. Each wait state gives you 20 ns, so with a single wait state you would need 30 ns memory. To work with 60 ns memory you would need to add *three* wait states (zero wait states = 10 ns, one wait state = 30 ns, two wait states = 50 ns, and three wait states = 70 ns).

Needless to say, from the system performance point of view, wait states are *not* a good thing. While the CPU is waiting for data from memory it cannot operate on that data. Adding a single wait state to a memory cycle on an 80486 CPU *doubles* the amount of time required to access the data. This, in turn, *halves* the speed of the memory access. Running with a wait state on every memory access is almost like cutting the processor clock frequency in half. You're going to get a lot less work done in the same amount of time.

However, we're not doomed to slow execution because of added wait states. There are several tricks hardware designers can play to achieve zero wait states *most* of the time. The most common of these is the use of *cache* (pronounced "cash") memory.

### 1.4.4 Cache Memory

If you look at a typical program (as many researchers have), you'll discover that it tends to access the same memory locations repeatedly. Furthermore, you also discover that a program often accesses adjacent memory locations. The technical names given to this phenomenon are *temporal locality of reference* and *spatial locality of reference*. When exhibiting spatial locality, a program accesses neighboring memory locations. When displaying temporal locality of reference a program repeatedly accesses the same memory location during a short time period. Both forms of locality occur in the following Pascal code segment:

```
for i := 0 to 10 do
  A [i] := 0;
```

There are two occurrences each of spatial and temporal locality of reference within this loop. Let's consider the obvious ones first.

In the Pascal code above, the program references the variable *i* several times. The for loop compares *i* against 10 to see if the loop is complete. It also increments *i* by one at the bottom of the loop. The assignment statement also uses *i* as an array index. This shows temporal locality of reference in action since the CPU accesses *i* at three points in a short time period.

This program also exhibits spatial locality of reference. The loop itself zeros out the elements of array *A* by writing a zero to the first location in *A*, then to the second location in *A*, and so on. Assuming that Pascal stores the elements of *A* into consecutive memory locations, each loop iteration accesses adjacent memory locations.

There is an additional example of temporal and spatial locality of reference in the Pascal example above, although it is not so obvious. Computer instructions which tell the system to do the specified task also appear in memory. These instructions appear sequentially in memory – the spatial locality part. The computer also executes these instructions repeatedly, once for each loop iteration – the temporal locality part.

If you look at the execution profile of a typical program, you'd discover that the program typically executes less than half the statements. Generally, a typical program might only use 10-20% of the memory allotted to it. At any one given time, a one megabyte program might only access four to eight kilobytes of data and code. So if you paid an outrageous sum of money for expensive zero wait state RAM, you wouldn't be using most of it at any one given time! Wouldn't it be nice if you could buy a small amount of fast RAM and dynamically reassign its address(es) as the program executes?

This is exactly what cache memory does for you. Cache memory sits between the CPU and main memory. It is a small amount of very fast (zero wait state) memory. Unlike normal memory, the bytes appearing within a cache do not have fixed addresses. Instead, cache memory can reassign the address of a data object. This allows the system to keep recently accessed values in the cache. Addresses that the CPU has never accessed or hasn't accessed in some time remain in main (slow) memory. Since most memory accesses are to recently accessed variables (or to locations near a recently accessed location), the data generally appears in cache memory.

Cache memory is not perfect. Although a program may spend considerable time executing code in one place, eventually it will call a procedure or wander off to some section of code outside cache memory. In such an event the CPU has to go to main memory to fetch the data. Since main memory is slow, this will require the insertion of wait states.

A cache *hit* occurs whenever the CPU accesses memory and finds the data in the cache. In such a case the CPU can usually access data with zero wait states. A cache *miss* occurs if the CPU accesses memory and the data is not present in cache. Then the CPU has to read the data from main memory, incurring a performance loss. To take advantage of locality of reference, the CPU copies data into the cache whenever it accesses an address not present in the cache. Since it is likely the system will access that same location shortly, the system will save wait states by having that data in the cache.

As described above, cache memory handles the temporal aspects of memory access, but not the spatial aspects. Caching memory locations *when you access them* won't speed up the program if you constantly access consecutive locations (spatial locality of reference). To solve this problem, most caching systems read several consecutive bytes from memory when a cache miss occurs<sup>5</sup>. The 80486, for example, reads 16 bytes at a shot upon a cache miss. If you read 16 bytes, why read them in blocks rather than as you need them? As

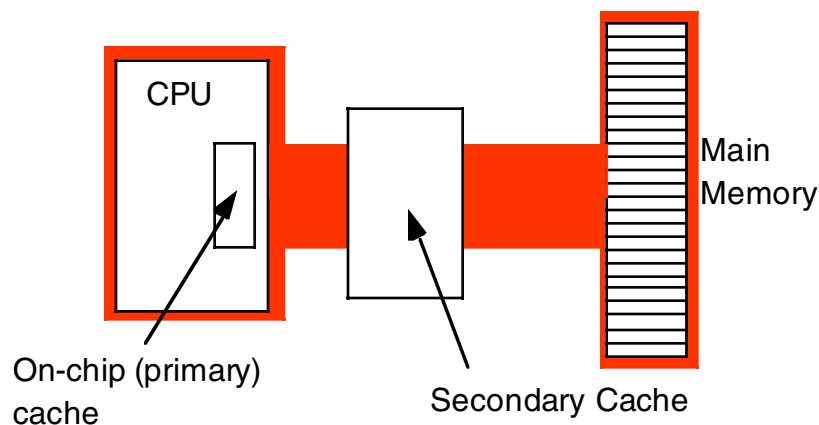
it turns out, most memory chips available today have special modes which let you quickly access several consecutive memory locations on the chip. The cache exploits this capability to reduce the average number of wait states needed to access memory.

If you write a program that randomly accesses memory, using a cache might actually slow you down. Reading 16 bytes on each cache miss is expensive if you only access a few bytes in the corresponding cache line. Nonetheless, cache memory systems work quite well.

It should come as no surprise that the ratio of cache hits to misses increases with the size (in bytes) of the cache memory subsystem. The 80486 chip, for example, has 8,192 bytes of on-chip cache. Intel claims to get an 80-95% hit rate with this cache (meaning 80-95% of the time the CPU finds the data in the cache). This sounds very impressive. However, if you play around with the numbers a little bit, you'll discover it's not all *that* impressive. Suppose we pick the 80% figure. Then one out of every five memory accesses, on the average, will not be in the cache. If you have a 50 MHz processor and a 90 ns memory access time, four out of five memory accesses require only one clock cycle (since they are in the cache) and the fifth will require about 10 wait states<sup>6</sup>. Altogether, the system will require 15 clock cycles to access five memory locations, or three clock cycles per access, on the average. That's equivalent to two wait states added to every memory access. Now do you believe that your machine runs at zero wait states?

There are a couple of ways to improve the situation. First, you can add more cache memory. This improves the cache hit ratio, reducing the number of wait states. For example, increasing the hit ratio from 80% to 90% lets you access 10 memory locations in 20 cycles. This reduces the average number of wait states per memory access to one wait state – a substantial improvement. Alas, you can't pull an 80486 chip apart and solder more cache onto the chip. However, the 80586/Pentium CPU has a significantly larger cache than the 80486 and operates with fewer wait states.

Another way to improve performance is to build a *two-level* caching system. Many 80486 systems work in this fashion. The first level is the on-chip 8,192 byte cache. The next level, between the on-chip cache and main memory, is a secondary cache built on the computer system circuit board (see Figure 1.15). Pentiums and later chips typically move the secondary cache onto the same chip carrier as the CPU (that is, Intel's designers have included the secondary cache as part of the CPU module).



**Figure 1.15 A Two Level Caching System**

A typical secondary cache contains anywhere from 32,768 bytes to one megabyte of memory. Common sizes on PC subsystems are 256K, 512K, and 1024 Kbytes (1 MB) of cache.

5. Engineers call this block of data a *cache line*.

6. Ten wait states were computed as follows: five clock cycles to read the first four bytes (10+20+20+20+20=90). However, the cache always reads 16 consecutive bytes. Most memory subsystems let you read consecutive addresses in about 40 ns after accessing the first location. Therefore, the 80486 will require an additional six clock cycles to read the remaining three double words. The total is 11 clock cycles or 10 wait states.

You might ask, “Why bother with a two-level cache? Why not use a 262,144 byte cache to begin with?” Well, the secondary cache generally does not operate at zero wait states. The circuitry to support 262,144 bytes of 10 ns memory (20 ns total access time) would be *very* expensive. So most system designers use slower memory which requires one or two wait states. This is still *much* faster than main memory. Combined with the on-chip cache, you can get better performance from the system.

Consider the previous example with an 80% hit ratio. If the secondary cache requires two cycles for each memory access and three cycles for the first access, then a cache miss on the on-chip cache will require a total of six clock cycles. All told, the average system performance will be two clocks per memory access. Quite a bit faster than the three required by the system without the secondary cache. Furthermore, the secondary cache can update its values in parallel with the CPU. So the number of cache misses (which affect CPU performance) goes way down.

You’re probably thinking, “So far this all sounds interesting, but what does it have to do with programming?” Quite a bit, actually. By writing your program carefully to take advantage of the way the cache memory system works, you can improve your program’s performance. By co-locating variables you commonly use together in the same cache line, you can force the cache system to load these variables as a group, saving extra wait states on each access.

If you organize your program so that it tends to execute the same sequence of instructions repeatedly, it will have a high degree of temporal locality of reference and will, therefore, execute faster.

---

## 1.5 Putting It All Together

This chapter has provided a quick overview of the components that make up a typical computer system. The remaining chapters in Topic Two will expand upon these comments to give you a complete overview of computer system organization.

# Memory Access and Organization

## Chapter Two

### 2.1 Chapter Overview

In earlier chapters you saw how to declare and access simple variables in an assembly language program. In this chapter you will learn how the 80x86 CPUs actually access memory (e.g., variables). You will also learn how to efficiently organize your variable declarations so the CPU can access them faster. In this chapter you will also learn about the 80x86 stack and how to manipulate data on the stack with some 80x86 instructions this chapter introduces. Finally, you will learn about dynamic memory allocation and the chapter concludes by discussing the HLA Standard Library Console module.

### 2.2 The 80x86 Addressing Modes

The 80x86 processors let you access memory in many different ways. Until now, you've only seen a single way to access a variable, the so-called *displacement-only* addressing mode that you use to access scalar variables. Now it's time to look at the many different ways that you can access memory on the 80x86.

The 80x86 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 80x86 addressing modes is the first step towards mastering 80x86 assembly language.

When Intel designed the original 8086 processor, they provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. Note that the 80386 retained all the modes of the previous processors. However, in 32-bit environments like Win32 and Linux, these earlier addressing modes are not very useful; indeed, HLA doesn't even support the use of these older, 16-bit only, addressing modes. Fortunately, anything you can do with the older addressing modes can be done with the new addressing modes as well (even better, as a matter of fact). Therefore, you won't need to bother learning the old 16-bit addressing modes on today's high-performance processors. Do keep in mind, however, that if you intend to work under MS-DOS or some other 16-bit operating system, you will need to study up on those old addressing modes.

#### 2.2.1 80x86 Register Addressing Modes

Most 80x86 instructions can operate on the 80x86's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 80x86 MOV (move) instruction:

```
mov( source, destination );
```

This instruction copies the data from the *source* operand to the *destination* operand. The eight-bit, 16-bit, and 32-bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 80x86 MOV instructions:

```
mov( bx, ax );    // Copies the value from BX into AX
mov( al, dl );    // Copies the value from AL into DL
mov( edx, esi );  // Copies the value from EDX into ESI
mov( bp, sp );    // Copies the value from BP into SP
mov( cl, dh );    // Copies the value from CL into DH
mov( ax, ax );    // Yes, this is legal!
```

Remember, the registers are the best place to keep often used variables. As you'll see a little later, instructions using the registers are shorter and faster than those that access memory. Throughout this chapter you'll see the abbreviated operands *reg* and *r/m* (register/memory) used wherever you may use one of the 80x86's general purpose registers.

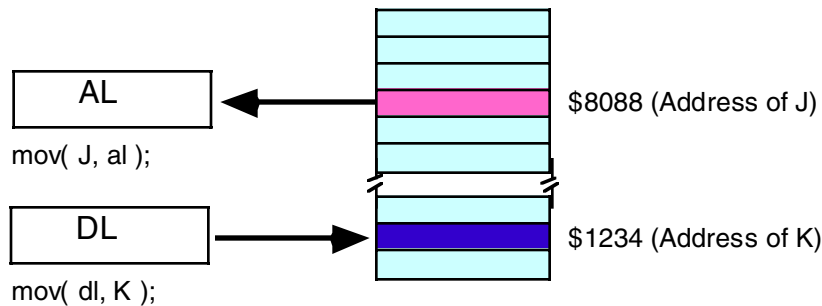
## 2.2.2 80x86 32-bit Memory Addressing Modes

The 80x86 provides hundreds of different ways to access memory. This may seem like quite a bit at first, but fortunately most of the addressing modes are simple variants of one another so they're very easy to learn. And learn them you should! The key to good assembly language programming is the proper use of memory addressing modes.

The addressing modes provided by the 80x86 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed. Variations on these five forms provide the many different addressing modes on the 80x86. See, from 256 down to five. It's not so bad after all!

### 2.2.2.1 The Displacement Only Addressing Mode

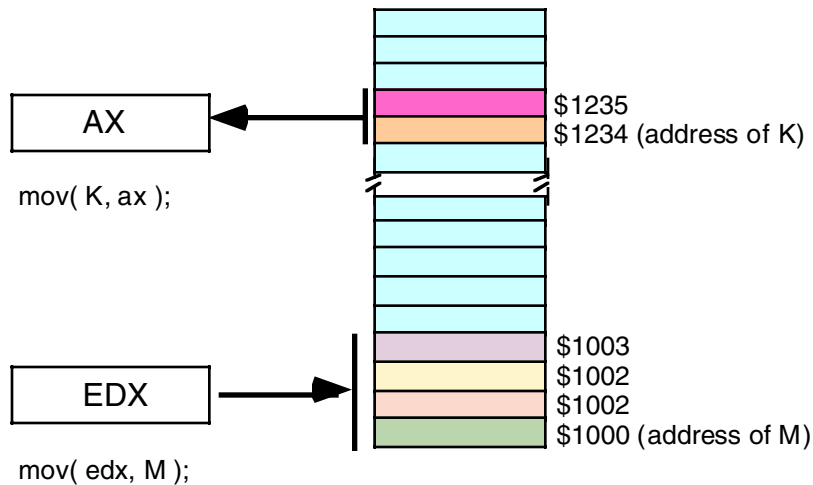
The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. The displacement-only addressing mode consists of a 32 bit constant that specifies the address of the target location. Assuming that variable *J* is an *int8* variable allocated at address \$8088, the instruction “`mov( J, al );`” loads the AL register with a copy of the byte at memory location \$8088. Likewise, if *int8* variable *K* is at address \$1234 in memory, then the instruction “`mov( dl, K );`” stores the value in the DL register to memory location \$1234 (see Figure 2.1).



**Figure 2.1 Displacement Only (Direct) Addressing Mode**

The displacement-only addressing mode is perfect for accessing simple scalar variables.

Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the MOV opcode in memory. On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address zero). The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words and double words on the 80x86 processors (see Figure 2.2).



**Figure 2.2 Accessing a Word or DWord Using the Displacement Only Addressing Mode**

### 2.2.2.2 The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. The term indirect means that the operand is not the actual address, but rather, the operand's value specifies the memory address to use. In the case of the register indirect addressing modes, the register's value is the memory location to access. For example, the instruction "mov( eax, [ebx] );" tells the CPU to store EAX's value at the location whose address is in EBX (the square brackets around EBX tell HLA to use the register indirect addressing mode).

There are eight forms of this addressing mode on the 80x86, best demonstrated by the following instructions:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

These eight addressing modes reference the memory location at the offset found in the register enclosed by brackets (EAX, EBX, ECX, EDX, EDI, ESI, EBP, or ESP, respectively).

Note that the register indirect addressing modes require a 32-bit register. You cannot specify a 16-bit or eight-bit register when using an indirect addressing mode<sup>1</sup>. Technically, you could load a 32-bit register with an arbitrary numeric value and access that location indirectly using the register indirect addressing mode:

```
mov( $1234_5678, ebx );
mov( [ebx], al );           // Attempts to access location $1234_5678.
```

Unfortunately (or fortunately, depending on how you look at it), this will probably cause Windows to generate a protection fault since it's not always legal to access arbitrary memory locations.

1. Actually, the 80x86 does support addressing modes involving certain 16-bit registers, as mentioned earlier. However, HLA does not support these modes and they are not particularly useful under Win32.

The register indirect addressing mode has lots of uses. You can use it to access data referenced by a pointer, you can use it to step through array data, and, in general, you can use it whenever you need to modify the address of a variable while your program is running.

The register indirect addressing mode provides an example of a *anonymous* variable. When using the register indirect addressing mode you refer to the value of a variable by its numeric memory address (e.g., the value you load into a register) rather than by the name of the variable. Hence the phrase anonymous variable.

HLA provides a simple operator that you can use to take the address of a `STATIC` variable and put this address into a 32-bit register. This is the “&” (address of) operator (note that this is the same symbol that C/C++ uses for the address-of operator). The following example loads the address of variable *J* into `EBX` and then stores the value in `EAX` into *J* using the register indirect addressing mode:

```
mov( &J, ebx );           // Load address of J into EBX.
mov( eax, [ebx] );        // Store EAX into J.
```

Of course, it would have been simpler to store the value in `EAX` directly into *J* rather than using two instructions to do this indirectly. However, you can easily imagine a code sequence where the program loads one of several different addresses into `EBX` prior to the execution of the “`mov( eax, [ebx]);`” statement, thus storing `EAX` into one of several different locations depending on the execution path of the program.

**Warning:** the “&” (address-of) operator is not a general address-of operator like the “&” operator in C/C++. You may only apply this operator to static variables<sup>2</sup>. It cannot be applied to generic address expressions or other types of variables. For more information on taking the address of such objects, see “Obtaining the Address of a Memory Object” on page 184.

### 2.2.2.3 Indexed Addressing Modes

The indexed addressing modes use the following syntax:

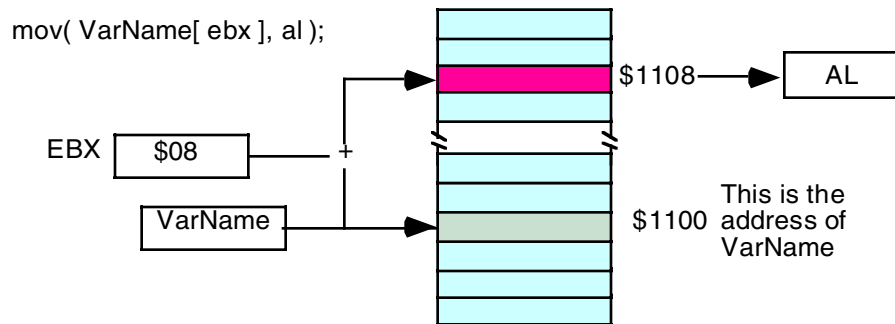
```
mov( VarName[ eax ], al );
mov( VarName[ ebx ], al );
mov( VarName[ ecx ], al );
mov( VarName[ edx ], al );
mov( VarName[ edi ], al );
mov( VarName[ esi ], al );
mov( VarName[ ebp ], al );
mov( VarName[ esp ], al );
```

*VarName* is the name of some variable in your program.

The indexed addressing mode computes an *effective address*<sup>3</sup> by adding the address of the specified variable to the value of the 32-bit register appearing inside the square brackets. This sum is the actual address in memory that the instruction will access. So if *VarName* is at address \$1100 in memory and `EBX` contains eight, then “`mov( VarName[ ebx ], al );`” loads the byte at address \$1108 into the `AL` register (see Figure 2.3).

2. Note: the term “static” here indicates a `STATIC`, `READONLY`, `STORAGE`, or `DATA` object.

3. The effective address is the ultimate address in memory that an instruction will access, once all the address calculations are complete.



**Figure 2.3 Indexed Addressing Mode**

The indexed addressing mode is really handy for accessing elements of arrays. You will see how to use this addressing mode for that purpose a little later in this text. A little later in this chapter you will see how to use the indexed addressing mode to step through data values in a table.

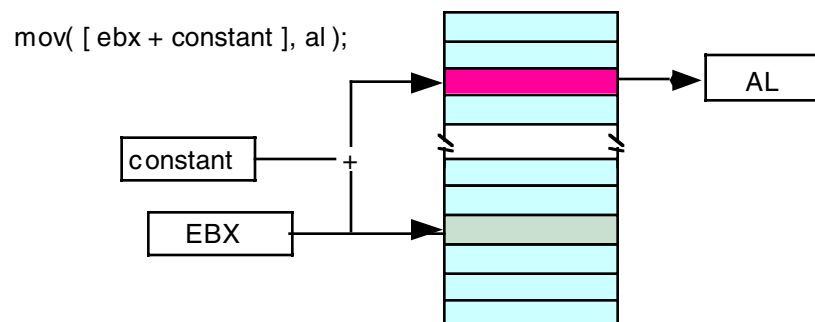
#### 2.2.2.4 Variations on the Indexed Addressing Mode

There are two important syntactical variations of the indexed addressing mode. Both forms generate the same basic machine instructions, but their syntax suggests other uses for these variants.

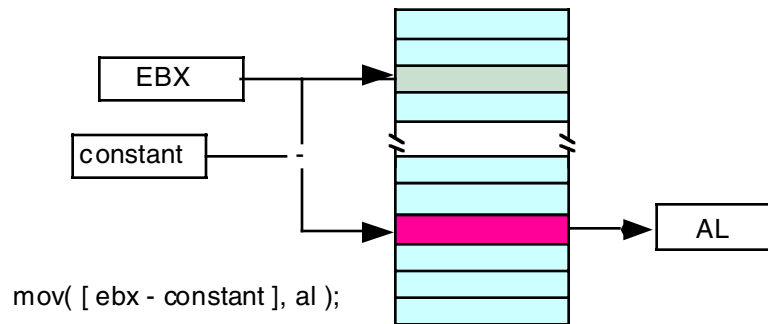
The first variant uses the following syntax:

```
mov( [ ebx + constant ], al );
mov( [ ebx - constant ], al );
```

These examples use only the EBX register. However, you can use any of the other 32-bit general purpose registers in place of EBX. This addressing mode computes its effective address by adding the value in EBX to the specified constant, or subtracting the specified constant from EBX (See Figure 2.4 and Figure 2.5).



**Figure 2.4 Indexed Addressing Mode Using a Register Plus a Constant**



**Figure 2.5 Indexed Addressing Mode Using a Register Minus a Constant**

This particular variant of the addressing mode is useful if a 32-bit register contains the *base address* of a multi-byte object and you wish to access a memory location some number of bytes before or after that location. One important use of this addressing mode is accessing fields of a record (or structure) when you have a pointer to the record data. You'll see a little later in this text that this addressing mode is also invaluable for accessing automatic (local) variables in procedures.

The second variant of the indexed addressing mode is actually a combination of the previous two forms. The syntax for this version is the following:

```
mov( VarName[ ebx + constant ], al );
mov( VarName[ ebx - constant ], al );
```

Once again, this example uses only the EBX register. You may, however, substitute any of the 32-bit general purpose registers in place of EBX in these two examples. This particular form is quite useful when accessing elements of an array of records (structures) in an assembly language program (more on that in a few chapters).

These instructions compute their effective address by adding or subtracting the *constant* value from *VarName* and then adding the value in EBX to this result. Note that HLA, not the CPU, computes the sum or difference of *VarName* and *constant*. The actual machine instructions above contain a single constant value that the instructions add to the value in EBX at run-time. Since HLA substitutes a constant for *VarName*, it can reduce an instruction of the form

```
mov( VarName[ ebx + constant ], al );
```

to an instruction of the form:

```
mov( constant1[ ebx + constant2 ], al );
```

Because of the way these addressing modes work, this is semantically equivalent to

```
mov( [ebx + (constant1 + constant2)], al );
```

HLA will add the two constants together at compile time, effectively producing the following instruction:

```
mov( [ebx + constant_sum], al );
```

So, HLA converts the first addressing mode of this sequence to the last in this sequence.

Of course, there is nothing special about subtraction. You can easily convert the addressing mode involving subtraction to addition by simply taking the two's complement of the 32-bit constant and then adding this complemented value (rather than subtracting the uncomplemented value). Other transformations are equally possible and legal. The end result is that these three variations on the indexed addressing mode are indeed equivalent.

### 2.2.2.5 Scaled Indexed Addressing Modes

The scaled indexed addressing modes are similar to the indexed addressing modes with two differences: (1) the scaled indexed addressing modes allow you to combine two registers plus a displacement, and (2) the scaled indexed addressing modes let you multiply the index register by a (scaling) factor of one, two, four, or eight. The allowable forms for these addressing modes are

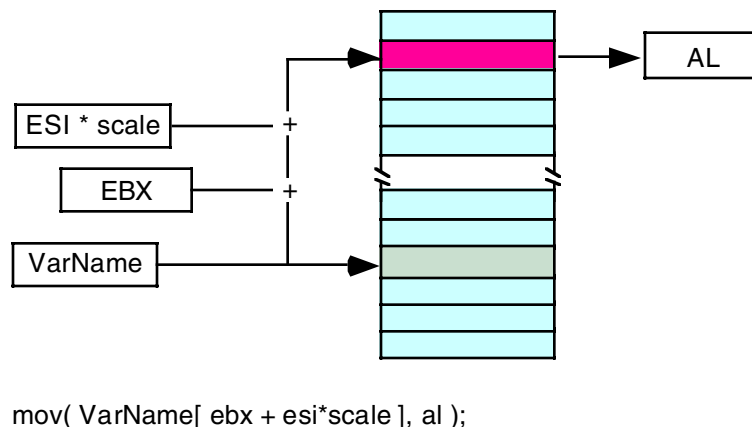
```
VarName[ IndexReg32*scale ]
VarName[ IndexReg32*scale + displacement ]
VarName[ IndexReg32*scale - displacement ]

[ BaseReg32 + IndexReg32*scale ]
[ BaseReg32 + IndexReg32*scale + displacement ]
[ BaseReg32 + IndexReg32*scale - displacement ]

VarName[ BaseReg32 + IndexReg32*scale ]
VarName[ BaseReg32 + IndexReg32*scale + displacement ]
VarName[ BaseReg32 + IndexReg32*scale - displacement ]
```

In these examples, *BaseReg<sub>32</sub>* represents any general purpose 32-bit register, *IndexReg<sub>32</sub>* represents any general purpose 32-bit register except ESP, and *scale* must be one of the constants: 1, 2, 4, or 8.

The primary difference between the scaled indexed addressing mode and the indexed addressing mode is the inclusion of the *IndexReg<sub>32</sub>\*scale* component. The effective address computation is extended by adding in the value of this new register after it has been multiplied by the specified scaling factor (see Figure 2.6 for an example involving EBX as the base register and ESI as the index register).



**Figure 2.6 The Scaled Indexed Addressing Mode**

In Figure 2.6, suppose that `EBX` contains \$100, `ESI` contains \$20, and `VarName` is at base address \$2000 in memory, then the following instruction:

```
mov( VarName[ ebx + esi*4 + 4 ], al );
```

will move the byte at address \$2184 ( $\$1000 + \$100 + \$20*4 + 4$ ) into the `AL` register.

The scaled indexed addressing mode is typically used to access elements of arrays whose elements are two, four, or eight bytes each. This addressing mode is also useful for access elements of an array when you have a pointer to the beginning of the array.

**Warning:** although this addressing mode contains to variable components (the base and index registers), don't get the impression that you use this addressing mode to access elements of a two-dimensional array by loading the two array indices into the two registers. Two-dimensional array access is quite a bit

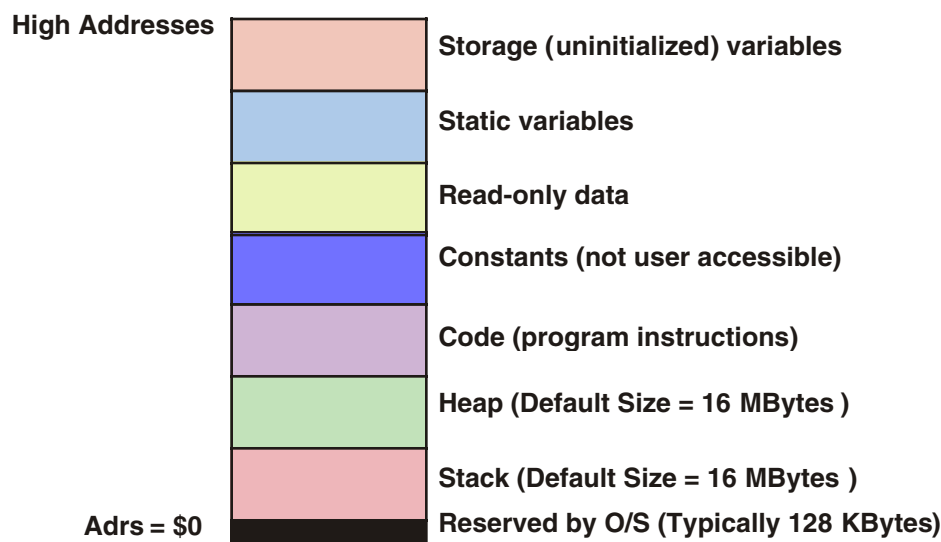
more complicated than this. A later chapter in this text will consider multi-dimensional array access and discuss how to do this.

### 2.2.2.6 Addressing Mode Wrap-up

Well, believe it or not, you've just learned several hundred addressing modes! That wasn't hard now, was it? If you're wondering where all these modes came from, just consider the fact that the register indirect addressing mode isn't a single addressing mode, but eight different addressing modes (involving the eight different registers). Combinations of registers, constant sizes, and other factors multiply the number of possible addressing modes on the system. In fact, you only need to memorize less than two dozen forms and you've got it made. In practice, you'll use less than half the available addressing modes in any given program (and many addressing modes you may never use at all). So learning all these addressing modes is actually much easier than it sounds.

## 2.3 Run-Time Memory Organization

An operating system like Windows NT tends to put different types of data into different sections (or segments) of main memory. Although it is possible to reconfigure memory to your choice by running the Microsoft Linker and specify various parameters, by default an HLA program loads into memory using the following basic organization:



**Figure 2.7 Win32 Typical Run-Time Memory Organization**

The lowest memory addresses are reserved by the operating system. Generally, your application is not allowed to access data (or execute instructions) at the lowest addresses in memory. One reason the O/S reserves this space is to help trap NULL pointer references. If you attempt to access memory location zero, the operating system will generate a “general protection fault” meaning you’ve accessed a memory location that doesn’t contain valid data. Since programmers often initialize pointers to NULL (zero) to indicate that the pointer is not pointing anywhere, an access of location zero typically means that the programmer has made a mistake and has not properly initialized a pointer to a legal (non-NULL) value. Also note that if you attempt to use one of the 80x86 sixteen-bit addressing modes (HLA doesn’t allow this, but were you to

encode the instruction yourself and execute it...) the address will always be in the range 0..\$1FFFE<sup>4</sup>. This will also access a location in the reserved area, generating a fault.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the stack section, the heap section, the code section, the READONLY section, the STATIC section, the DATA section, and the STORAGE section. Each of these memory sections correspond to some type of data you can create in your HLA programs. The following sections discuss each of these sections in detail.

---

### 2.3.1 The Code Section

The code section contains the machine instructions that appear in an HLA program. HLA translates each machine instruction you write into a sequence of one or more byte values. The CPU interprets these byte values as machine instructions during program execution.

By default, when HLA links your program it tells the system that your program can execute instructions out of the code segment and you can read data from the code segment. Note, specifically, that you cannot write data to the code segment. The Windows operating system will generate a general protection fault if you attempt to store any data into the code segment.

Remember, machine instructions are nothing more than data bytes. In theory, you could write a program that stores data values into memory and then transfers control to the data it just wrote, thereby producing a program that writes itself as it executes. This possibility produces romantic visions of Artificially Intelligent programs that modify themselves to produce some desired result. In real life, the effect is somewhat less glamorous.

Prior to the popularity of *protected mode operating systems*, like Windows NT, a program could overwrite the machine instructions during execution. Most of the time this was caused by defects in a program, not by some super-smart artificial intelligence program. A program would begin writing data to some array and fail to stop once it reached the end of the array, eventually overwriting the executing instructions that make up the program. Far from improving the quality of the code, such a defect usually causes the program to fail spectacularly.

Of course, if a feature is available, someone is bound to take advantage of it. Some programmers have discovered that in some special cases, using *self-modifying code*, that is, a program that modifies its machine instructions during execution, can produce slightly faster or slightly smaller programs. Unfortunately, self-modifying code is very difficult to test and debug. Given the speed of modern processors combined with their instruction set and wide variety of addressing modes, there is almost no reason to use self-modifying code in a modern program. Indeed, protected mode operating systems like Windows make it difficult for you to write self modifying code.

HLA automatically stores the data associated with your machine code into the code section. In addition to machine instructions, you can also store data into the code section by using the following pseudo-opcodes:

- byte
- word
- dword
- uns8
- uns16
- uns32
- int8
- int16
- in32
- boolean
- char

---

4. It's \$1FFFE, not \$FFFF because you could use the indexed addressing mode with a displacement of \$FFFF along with the value \$FFFF in a 16-bit register.

The syntax for each of these *pseudo-opcodes*<sup>5</sup> is exemplified by the following BYTE statement:

```
byte comma_separated_list_of_byte_constants ;
```

Here are some examples:

```
boolean    true;
char       'A';
byte       0,1,2;
byte       "Hello", 0
word       0,2;
int8       -5;
uns32      356789, 0;
```

If more than one value appears in the list of values after the pseudo-opcode, HLA emits each successive value to the code stream. So the first *byte* statement above emits three bytes to the code stream, the values zero, one, and two. If a string appears within a byte statement, HLA emits one byte of data for each character in the string. Therefore, the second byte statement above emits six bytes: the characters 'H', 'e', 'l', 'l', 'o', and 'o', followed by a zero byte.

Keep in mind that the CPU will attempt to treat data you emit to the code stream as machine instructions unless you take special care not to allow the execution of the data. For example, if you write something like the following:

```
mov( 0, ax );
byte 0,1,2,3;
add( bx, cx );
```

Your program will attempt to execute the 0, 1, 2, and 3 byte values as a machine instruction after executing the MOV. Unless you know the machine code for a particular instruction sequence, sticking such data values into the middle of your code will almost always produce unexpected results. More often than not, this will crash your program. Therefore, you should never insert arbitrary data bytes into the middle of an instruction stream unless you know exactly what executing those data values will do in your program<sup>6</sup>.

In the chapter on intermediate procedures, we will take another look at embedding data into the code stream. This is a convenient way to pass certain types of parameters to various procedures. In the chapter on advanced control structures, you will see other reasons for embedding data into the code stream. For now, just keep in mind that it is possible to do this but that you should avoid embedding data into the code stream.

### 2.3.2 The Read-Only Data Section

The READONLY data section holds constants, tables, and other data that your program must not change during program execution. You can place read only objects in your program by declaring them in the READONLY declaration section. The READONLY data declaration section is very similar to the STATIC section with three primary differences:

- The READONLY section begins with the reserved word READONLY rather than STATIC,
- All declarations in the READONLY section must have an initializer, and
- You are not allowed to store data into a READONLY object while the program is running.

Example:

```
readonly
pi:      real32 := 3.14159;
```

5. A pseudo-opcode is a data declaration statement that emits data to the code section, but isn't a true machine instruction (e.g., BYTE is a pseudo-opcode, MOV is a machine instruction).

6. The main reason for encoding machine code using a data directive like *byte* is to implement machine instructions that HLA does not support (for example, to implement machine instructions added after HLA was written but before HLA could be updated for the new instruction(s)).

```
e:      real32 := 2.71;
MaxU16: uns16  := 65_535;
MaxI16:  int16  := 32_767;
```

All READONLY object declarations must have an initializer because you cannot initialize the value under program control (since you are not allowed to write data into a READONLY object). The operating system will generate an exception and abort your program if you attempt to write a value to a READONLY object. For all intents and purposes, READONLY objects can be thought of as constants. However, these constants to consume memory and other than the fact that you cannot write data to READONLY objects, they behave like, and can be used like, STATIC variables.

The READONLY reserved word allows an alignment parameter, just like the STATIC keyword. You may also place the ALIGN directive in the READONLY section in order to align individual objects on a specific boundary. The following example demonstrates both of these features in the READONLY section:

```
readonly( 8 )
  pi:    real64 := 3.14159265359;
  aChar: char   := 'a';
  align(4);
  d:      dword := 4;
```

---

### 2.3.3 The Storage Section

The READONLY section requires that you initialize all objects you declare. The STATIC section lets you optionally initialize objects (or leave them uninitialized, in which case they have the default initial value of zero). The STORAGE section completes the initialization coverage: you use it to declare variables that are always uninitialized when the program begins running. The STORAGE section begins with the “storage” reserved word and then contains variable declarations that are identical to those appearing in the STATIC section except that you are not allowed to initialize the object. Here is an example:

```
storage
  UninitUns32:    uns32;
  i:              int32;
  character:      char;
  b:              byte;
```

Variables you declare in the STORAGE section may consume less disk space in the executable file for the program. This is because HLA writes out initial values for READONLY and STATIC objects to the executable file, but uses a compact representation for uninitialized variables you declare in the STORAGE section.

Like the STATIC and READONLY sections, you can supply an alignment parameter after the STORAGE keyword and the ALIGN directive may appear within the STORAGE section. Of course, aligning your data can produce faster access to that data at the expense of a slightly larger STORAGE section. The following example demonstrates the use of these two features in the STORAGE section:

```
storage( 4 )
  d:      dword;
  b:      byte;
  align(2);
  w:      word;
```

---

### 2.3.4 The Data and Static Sections

In addition to declaring static variables, you can also embed lists of data into the STATIC memory segment. You use the same technique to embed data into your STATIC section that you use to embed data into

the code section: you use the *byte*, *word*, *dword*, *uns32*, etc., pseudo-opcodes. Consider the following example:

```
static
  b: byte := 0;
    byte 1,2,3;

  u: uns32 := 1;
    uns32 5,2,10;

  c: char;
    char 'a', 'b', 'c', 'd', 'e', 'f';

  bn: boolean;
    boolean true;
```

Data that HLA writes to the STATIC memory segment using these pseudo-opcodes is written to the segment after the preceding variables. For example, the byte values one, two, and three are emitted to the STATIC section after *b*'s zero byte in the example above. Since there aren't any labels associated with these values, you do not have direct access to these values in your program. The section on address expressions, later in this chapter, will discuss how to access these extra values.

In the examples above, note that the *c* and *bn* variables do not have an (explicit) initial value. However, HLA always initializes variables in the STATIC section to all zero bits, so HLA assigns the NULL character (ASCII code zero) to *c* as its initial value. Likewise, HLA assigns false as the initial value for *bn*. In particular, you should note that your variable declarations in the STATIC section always consume memory, even if you haven't assigned them an initial value. Any data you declare in a pseudo-opcode like BYTE will always follow the actual data associated with the variable declaration.

The DATA declaration section lets you declare variables without actually allocating storage for the variable. Consider the following example:

```
data
  d: dword;
    dword 1,2,3,4;
```

In this example, HLA creates a double word variable *d* but does not allocate any storage for it. Therefore, whenever you access the variable *d* in your program you will actually access the first double word value immediately following *d* (which would be the value one in the example above). Assuming you don't want HLA to automatically allocate storage for a variable and you want to rely, instead, on the values immediately following the declaration, then the DATA section will do the job for you.

Since HLA does not allocate storage for variables you declare in the DATA section, you can create *aliases*<sup>7</sup> by declaring two or more variables in a row followed by a data initialization pseudo-opcode:

```
data
  ViewAsBytes: byte;
  ViewAsWords: word;
  ViewAsDWords: dword
    dword 550_000, 66_000, 7700;
```

In this example, an instruction of the form “mov( ViewAsBytes, AL );” will load the first byte of the data appearing in the *dword* pseudo-opcode (\$70, which is the L.O. byte of 550,000) into the AL register. This is because *ViewAsBytes*, *ViewAsWords*, and *ViewAsDWords* all refer to the same memory location. Separate data is not allocated for each declaration, only for the *dword* pseudo-opcode.

This last example demonstrates another reason you might want to use the DATA section to declare variables- it lets you declare a variable as one type and initialize it with data of some other type. For example, consider the following:

```
data
```

---

7. Two variables are aliases if they use different names to refer to the same memory location.

```
d: dword;
   byte  ' ', 'a', 'b', 'c';
```

This example demonstrates how you can create a double word variable and initialize the four bytes of the double word variable with four character values.

Of course, the DATA section supports the alignment parameter and the ALIGN keyword, just like the STATIC section. The following example demonstrates how to align data objects in the DATA section using these features:

```
data( 16 )
b0:  byte;
w0:  word;
d:   dword;
     byte  0;

b1:  byte;
     byte  1;

b2:  byte;
w1:  word;
     byte  2;

b3:  byte;
     byte  3;

chr:  char;
      char 'a';

align( 4 );
d2:  dword;
     dword 0;
```

HLA allocates both STATIC and DATA variables in the static memory segment. HLA does not place them in separate memory segments as it does with the READONLY and STORAGE variables.

Whenever you access one of the variables in a DATA section, the CPU only access the object immediately following the data declaration. Consider the following example:

```
data
variable: byte;
          byte 0, 10, 100, 1, 11, 101, 2, 12, 102, 3, 13, 103;
          .
          .
          .
mov( variable, al );
```

This program loads the value zero into AL since the first byte after the declaration for *variable* contains a zero. This instruction ignores the remaining eleven bytes following the initial zero byte in memory.

If you want to access the other eleven values in this list of values then you must explicitly specify the address of each of these bytes in the instruction that needs to access them. Unfortunately, only the first byte has a label (variable name) associated with it, so you cannot directly refer to the other bytes in this list using only a label. However, since HLA always allocates objects in a list in contiguous memory locations, we know that the value 10 follows the zero byte in memory, that value 100 immediately follows 10 (and, therefore, is two bytes beyond the zero byte), etc. The 80x86 indexed addressing modes let us modify an address by adding the value in a 32-bit register to some variable's base address, so we can use this addressing mode to access these additional bytes in the list. The following example code demonstrates how to step through the items in this list and print their values:

```
mov( 0, ebx );
while( ebx < 12 ) do
```

```

mov( variable[ebx], al );
stdout.put( "value ", ebx, " = ", al, nl );
add( 1, ebx );

endwhile;

```

Each iteration of this loop accesses successive elements in the list. On the first iteration, EBX contains zero. Therefore, “variable[ebx]” accesses memory location *variable* + 0 (i.e., the zero byte in the list above). At the end of the loop, the code adds one to EBX so on the next iteration “variable[ebx]” refers to the next address in memory, the address of the byte containing 10. This process repeats for all 12 items in the list with each successive iteration fetching the next sequential byte in memory (e.g., the next byte from the list).

You must take care when stepping through the lists like this when using the 80x86 addressing modes. Don’t forget that an addressing mode of the form “variable[ebx]” simply computes an address in memory by adding together the address of *variable* and the current value in EBX. Despite the syntax similarities, this is not, in general, exactly the same thing as accessing elements of an array. For example, suppose you declare the *variable* list as follows:

```

data
variable: word;
          word 0, 10, 100, 1, 11, 101, 2, 12, 102, 3, 13, 103;

```

The following WHILE loop will not properly display the values in this list:

```

mov( 0, ebx );
while( ebx < 12 ) do

    mov( variable[ebx], ax );
    stdout.put( "value ", ebx, " = ", ax, nl );
    add( 1, ebx );

endwhile;

```

The problem with this example is that “variable[ebx]” computes the address of some memory location by adding the value in EBX with *variable*’s address in memory. As it turns out, this does not properly compute the address of the *i*<sup>th</sup> item in the list (assuming EBX contains *i*). Here’s why: remember, each word in the variable list consumes two consecutive bytes in memory. Therefore, the first item in the list requires two zero bytes, the second item has \$0A in the L.O. byte and \$00 in the H.O. byte (since \$000A is the 16-bit representation for ten), etc. When you run the code above, the first thing it prints is the value 0000 since the first word contains zero. The second time through the loop, this program fetches the word at address *variable*+1. This does not fetch the value 10. Instead, it fetches zero as the L.O. byte and \$0A as the H.O. byte since the word at location *variable*+1 consists of the H.O. byte of the first list element and the L.O. byte of the second list element. As a result, this program displays “0A00” (2,560) during the second iteration of the loop. On the third iteration of the loop (EBX=2), this code fragment displays the word at location *variable*+2. Two bytes beyond the first word we have the second word in the list, so the program prints “000A” which is the value we should have printed on the second iteration. This process continues until the loop prints 12 values. It will properly print the first half of the values in the list above and the remaining six output values will be scrambled versions of the data above (using the H.O. byte of one entry as the L.O. byte and the L.O. byte of the next entry in the table as the H.O. byte of the data it prints).

The correct way to step through the list of words above is to adjust our index by two on each iteration of the loop. One simple way to do this is to add two to EBX on each iteration of the loop (and adjust the loop termination condition accordingly):

```

mov( 0, ebx );
while( ebx < 24 ) do

    mov( variable[ebx], ax );

```

```

        stdout.put( "value ", ebx, " = ", ax, nl );
        add( 2, ebx );

    endwhile;

```

A better way to handle this problem is to use one of the 80x86's scaled index addressing modes. The following example demonstrates how to do this using the EBX\*2 scaled index addressing mode:

```

    mov( 0, ebx );
    while( ebx < 12 ) do

        mov( variable[ebx*2], ax );
        stdout.put( "value ", ebx, " = ", ax, nl );
        add( 1, ebx );

    endwhile;

```

If your list of values is a list of double word objects, you will need to bump up the index by four on each iteration of the loop (or use the EBX\*4 scaled index addressing mode). For other object sizes, you need to increase the index by the size of an individual object in the list on each iteration. We'll take another look at this process when we consider arrays in a later chapter of this text.

### 2.3.5 The NOSTORAGE Attribute

The DATA section allows you to create aliases of objects (that is, two objects share the same memory locations) and initialize variables with arbitrarily typed data. The NOSTORAGE attribute lets you achieve this same effect in the other static data declaration sections (i.e., STATIC, READONLY, and STORAGE). The NOSTORAGE option tells HLA to assign the current address in a data declaration section to a variable but not allocate any storage for the object. Therefore, that variable will share the same memory address as the next object appearing in the variable declaration section. Here is the syntax for the NOSTORAGE option:

```
variableName: varType : NOSTORAGE;
```

Note that you follow the type name with “:nostorage” rather than some initial value or just a semicolon. The following code sequence provides an example of using the NOSTORAGE option in the READONLY section:

```

readonly
    abcd: dword: nostorage;
           byte 'a', 'b', 'c', 'd';

```

In this example, *abcd* is a double word whose L.O. byte contains 97 ('a'), byte #1 contains 98 ('b'), byte #2 contains 99 ('c'), and the H.O. byte contains 100 ('d'). HLA does not reserve storage for the *abcd* variable, so HLA associates the following four bytes in memory (allocated by the BYTE directive) with *abcd*. This behavior is nearly identical to declarations appearing in the DATA section except, of course, values in the READONLY section are read-only and will raise an exception if you attempt to write to them.

Note that the NOSTORAGE attribute is only legal in the STATIC, STORAGE, and READONLY sections. It would be irrelevant in the DATA section and HLA does not allow its use in the VAR section.

### 2.3.6 The Var Section

HLA provides another variable declaration section, the VAR section, that you can use to create *automatic* variables. Your program will allocate storage for automatic variables whenever a program unit (i.e., main program or procedure) begins execution, and it will deallocate storage for automatic variables when

that program unit returns to its caller. Of course, any automatic variables you declare in your main program have the same *lifetime*<sup>8</sup> as all the STATIC, DATA, READONLY, and STORAGE objects, so the automatic allocation feature of the VAR section is wasted on the main program. In general, you should only use automatic objects in procedures (see the chapter on procedures for details). HLA allows them in your main program's declaration section as a generalization.

Since variables you declare in the VAR section are created at run-time, HLA does not allow initializers on variables you declare in this section. So the syntax for the VAR section is nearly identical to that for the STORAGE section; the only real difference in the syntax between the two is the use of the VAR reserved word rather than the STORAGE reserved word. The following example illustrates this:

```
var
  vInt:   int32;
  vChar:  char;
```

HLA allocates variables you declare in the VAR section in the stack segment. HLA does not allocate VAR objects at fixed locations within the stack segment; instead, it allocates these variables in an *activation record* associated with the current program unit. The chapter on intermediate procedures will discuss activation records in greater detail, for now it is important only to realize that HLA programs use the EBP register as a pointer to the current activation record. Therefore, anytime you access a var object, HLA automatically replaces the variable name with “[EBP+displacement]”. Displacement is the offset of the object in the activation record. This means that you cannot use the full scaled indexed addressing mode (a base register plus a scaled index register) with VAR objects because VAR objects already use the EBP register as their base register. Although you will not directly use the two register addressing modes often, the fact that the VAR section has this limitation is a good reason to avoid using the VAR section in your main program.

The VAR section supports the align parameter and the ALIGN directive, like the other declaration sections, however, these align directives only guarantee that the alignment within the activation record is on the boundary you specify. If the activation record is not aligned on a reasonable boundary (unlikely, but possible) then the actual variable alignment won't be correct.

---

## 2.3.7 Organization of Declaration Sections Within Your Programs

The STATIC, READONLY, STORAGE, DATA, and VAR sections may appear zero or more times between the PROGRAM header and the associated BEGIN for the main program. Between these two points in your program, the declaration sections may appear in any order as the following example demonstrates:

```
program demoDeclarations;

static
  i_static:  int32;

var
  i_auto:    int32;

storage
  i_uninit:  int32;

data
  i_data:    int32;
             byte 0, 1, 2, 3;

readonly
  i_readonly: int32 := 5;
```

---

8. The lifetime of a variable is the point from which memory is first allocated to the point the memory is deallocated for that variable.

```

static
    j:          uns32;

var
    k:char;

readonly
    i2:uns8 := 9;

storage
    c:char;

storage
    d:dword;

begin demoDeclarations;

    << code goes here >>

end demoDeclarations;

```

In addition to demonstrating that the sections may appear in an arbitrary order, this section also demonstrates that a given declaration section may appear more than once in your program. When multiple declaration sections of the same type (e.g., the three `STORAGE` sections above) appear in a declaration section of your program, HLA combines them into a single section<sup>9</sup>.

---

## 2.4 Address Expressions

In the section on addressing modes (see “The 80x86 Addressing Modes” on page 149) this chapter pointed out that addressing modes take a couple generic forms, including:

```

VarName[ Reg32 ]
VarName[ Reg32 + offset ]
VarName[ RegNotESP32*Scale ]
VarName[ Reg32 + RegNotESP32*Scale ]
VarName[ RegNotESP32*Scale + offset ]
and
VarName[ Reg32 + RegNotESP32*Scale + offset ]

```

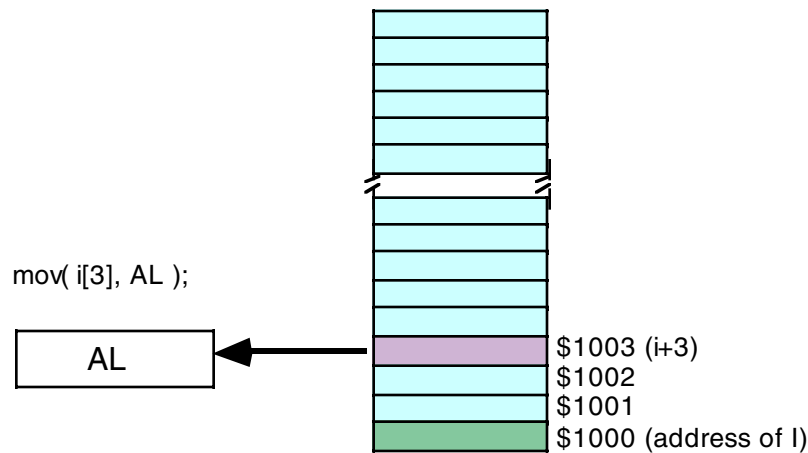
Another legal form, which isn’t actually a new addressing mode but simply an extension of the displacement-only addressing mode is

```
VarName[ offset ]
```

This latter example computes its effective address by adding the (constant) offset within the brackets to the specified variable address. For example, the instruction “`MOV(Address[3], AL);`” loads the AL register with the byte in memory that is three bytes beyond the *Address* object.

---

9. Remember, though, that HLA combines *static* and *data* declarations into the same memory segment.



**Figure 2.8 Using an Address Expression to Access Data Beyond a Variable**

It is extremely important to remember that the *offset* value in these examples must be a constant. If *Index* is an *int32* variable, then “Variable[Index]” is not a legal specification. If you wish to specify an index that varies at run-time, then you must use one of the indexed or scaled indexed addressing modes; that is, any index that changes at run-time must be held in a general purpose 32-bit register.

Another important thing to remember is that the offset in “Address[offset]” is a byte offset. Despite the fact that this syntax is reminiscent of array indexing in a high level language like C/C++ or Pascal, this does not properly index into an array of objects unless *Address* is an array of bytes.

This text will consider an *address expression* to be any legal 80x86 addressing mode that includes a displacement (i.e., variable name) or an offset. In addition to the above forms, the following are also address expressions:

$$\begin{aligned} & [ \text{Reg}_{32} + \text{offset} ] \\ & [ \text{Reg}_{32} + \text{RegNotESP}_{32} * \text{Scale} + \text{offset} ] \end{aligned}$$

This text will *not* consider the following to be address expressions since they do not involve a displacement or offset component:

$$\begin{aligned} & [ \text{Reg}_{32} ] \\ & [ \text{Reg}_{32} + \text{RegNotESP}_{32} * \text{Scale} ] \end{aligned}$$

Address expressions are special because those instructions containing an address expression always encode a displacement constant as part of the machine instruction. That is, the machine instruction contains some number of bits (usually eight or thirty-two) that hold a numeric constant. That constant is the sum of the displacement (i.e., the address or offset of the variable) plus the offset supplied in the addressing mode. Note that HLA automatically adds these two values together for you (or subtracts the offset if you use the “-” rather than “+” operator in the addressing mode).

Until this point, the offset in all the addressing mode examples has always been a single numeric constant. However, HLA also allows a *constant expression* anywhere an offset is legal. A constant expression consists of one or more constant terms manipulated by operators such as addition, subtraction, multiplication, division, modulo, and a wide variety of other operators. Most address expressions, however, will only involve addition, subtraction, multiplication, and sometimes, division. Consider the following example:

```
mov( X[ 2*4+1 ], al );
```

This instruction will move the byte at address  $X+9$  into the AL register.

The value of an address expression is always computed at compile-time, never while the program is running. When HLA encounters the instruction above, it calculates  $2*4+1$  on the spot and adds this result to

the base address of *X* in memory. HLA encodes this single sum (base address of *X* plus nine) as part of the instruction; HLA does not emit extra instructions to compute this sum for you at run-time (which is good, doing so would be less efficient). Since HLA computes the value of address expressions at compile-time, all components of the expression must be constants since HLA cannot know what the value of a variable will be at run-time while it is compiling the program.

Address expressions are very useful for accessing addition bytes in memory beyond a variable, particularly when you've used the *byte*, *word*, *dword*, etc., statements in a *STATIC*, *DATA*, or *READONLY* section to tack on additional bytes after a data declaration. For example, consider the following program:

---

---

```

program adrsExpressions;
#include( "stdlib.hhf" );

data
    i:      int8;
           byte 0, 1, 2, 3;

begin adrsExpressions;

    stdout.put
    (
        "i[0] = ", i[0], nl,
        "i[1] = ", i[1], nl,
        "i[2] = ", i[2], nl,
        "i[3] = ", i[3], nl
    );

end adrsExpressions;

```

---

---

### Program 3.1 Demonstration of Address Expressions

---

---

Throughout this chapter and those that follow you will see several additional uses of address expressions.

## 2.5 Type Coercion

Although HLA is fairly loose when it comes to type checking, HLA does ensure that you specify appropriate operand sizes to an instruction. For example, consider the following (incorrect) program:

```

program hasErrors;
static
    i8:      int8;
    i16:     int16;
    i32:     int32;
begin hasErrors;

    mov( i8,  eax );
    mov( i16, al );
    mov( i32,  ax );

end hasErrors;

```

HLA will generate errors for the three MOV instructions appearing in this program. This is because the operand sizes do not agree. The first instruction attempts to move a byte into EAX, the second instruction

attempts to move a word into AL and the third instruction attempts to move a dword into AX. The MOV instruction, of course, requires that its two operands both be the same size.

While this is a good feature in HLA<sup>10</sup>, there are times when it gets in the way of the task at hand. For example, consider the following data declaration:

```
data
    byte_values: byte;
                byte    0, 1;

    ...

    mov( byte_values, ax );
```

In this example let's assume that the programmer really wants to load the word starting at address *byte\_values* in memory into the AX register because they want to load AL with zero and AH with one using a single instruction. HLA will refuse, claiming there is a type mismatch error (since *byte\_values* is a *byte* object and AX is a *word* object). The programmer could break this into two instructions, one to load AL with the byte at address *byte\_values* and the other to load AH with the byte at address *byte\_values[1]*. Unfortunately, this decomposition makes the program slightly less efficient (which was probably the reason for using the single MOV instruction in the first place). Somehow, it would be nice if we could tell HLA that we know what we're doing and we want to treat the *byte\_values* variable as a *word* object. HLA's *type coercion* facilities provide this capability.

Type coercion<sup>11</sup> is the process of telling HLA that you want to treat an object as an explicitly specified type, regardless of its declared type. To coerce the type of a variable, you use the following syntax:

```
(type newTypeName addressingMode)
```

The *newTypeName* component is the new type you wish HLA to apply to the memory location specified by *addressingMode*. You may use this coercion operator anywhere a memory address is legal. To correct the previous example, so HLA doesn't complain about type mismatches, you would use the following statement:

```
mov( (type word byte_values), ax );
```

This instruction tells HLA to load the AX register with the word starting at address *byte\_values* in memory. Assuming *byte\_values* still contains its initial values, this instruction will load zero into AL and one into AH.

Type coercion is necessary when you specify an anonymous variable as the operand to an instruction that modifies memory directly (e.g., NEG, SHL, NOT, etc.). Consider the following statement:

```
not( [ebx] );
```

HLA will generate an error on this instruction because it cannot determine the size of the memory operand. That is, the instruction does not supply sufficient information to determine whether the program should invert the bits in the byte pointed at by EBX, the word pointed at by EBX, or the double word pointed at by EBX. You must use type coercion to explicitly tell HLA the size of the memory operand when using anonymous variables with these types of instructions:

```
not( (type byte [ebx]) );
not( (type word [ebx]) );
not( (type dword [ebx]) );
```

**Warning:** do not use the type coercion operator unless you know exactly what you are doing and the effect that it has on your program. Beginning assembly language programmers often use type coercion as a tool to quiet the compiler when it complains about type mismatches without solving the underlying problem. For example, consider the following statement:

```
mov( eax, (type dword byteVar) );
```

10. After all, if the two operand sizes are different this usually indicates an error in the program.

11. Also called type casting in some languages.

Without the type coercion operator, HLA probably complains about this instruction because it attempts to store a 32-bit register into an eight-bit memory location (assuming *byteVar* is a byte variable). A beginning programmer, wanting their program to compile, may take a short cut and use the type coercion operator as shown in this instruction; this certainly quiets the compiler - it will no longer complain about a type mismatch. So the beginning programmer is happy. But the program is still incorrect, the only difference is that HLA no longer warns you about your error. The type coercion operator does not fix the problem of attempting to store a 32-bit value into an eight-bit memory location - it simply allows the instruction to store a 32-bit value *starting at the address* specified by the eight-bit variable. The program still stores away four bytes, overwriting the three bytes following *byteVar* in memory. This often produces unexpected results including the phantom modification of variables in your program<sup>12</sup>. Another, rarer, possibility is for the program to abort with a general protection fault. This can occur if the three bytes following *byteVar* are not allocated actual memory or if those bytes just happen to fall in a read-only segment in memory. The important thing to remember about the type coercion operator is this: "If you can't exactly state the affect this operator has, don't use it."

Also keep in mind that the type coercion operator does not perform any translation of the data in memory. It simply tells the compiler to treat the bits in memory as a different type. It will not automatically sign extend an eight-bit value to 32 bits nor will it convert an integer to a floating point value. It simply tells the compiler to treat the bit pattern that exists in memory as a different type.

---

## 2.6 Register Type Coercion

You can also cast a register as a specific type using the type coercion operator. By default, the eight-bit registers are of type *byte*, the 16-bit registers are of type *word*, and the 32-bit registers are of type *dword*. With type coercion, you can cast a register as a different type *as long as the size of the new type agrees with the size of the register*. This is an important restriction that does not apply when applying type coercion to a memory variable.

Most of the type you do not need to coerce a register to a different type. After all, as *byte*, *word*, and *dword* objects, they are already compatible with all one, two, and four byte objects. However, there are a few instances where register type coercion is handy, if not downright necessary. Two examples include boolean expressions in HLA high level language statements (e.g., IF and WHILE) and register I/O in the *stdout.put* and *stdin.get* (and related) statements.

In boolean expressions, *byte*, *word*, and *dword* objects are always treated as unsigned values. Therefore, without type coercion register objects are always treated as unsigned values so the boolean expression in the following IF statement is always false (since there is no unsigned value less than zero):

```
if( eax < 0 ) then

    stdout.put( "EAX is negative!", nl );

endif;
```

You can overcome this limitation by casting EAX as an *int32* value:

```
if( (type int32 eax) < 0 ) then

    stdout.put( "EAX is negative!", nl );

endif;
```

In a similar vein, the HLA Standard Library *stdout.put* routine always outputs *byte*, *word*, and *dword* values as hexadecimal numbers. Therefore, if you attempt to print a register, the *stdout.put* routine will print

---

12. If you have a variable immediately following *byteVar* in this example, the MOV instruction will surely overwrite the value of that variable, whether or not you intend this to happen.

it as a hex value. If you would like to print the value as some other type, you can use register type coercion to achieve this:

```
stdout.put( "AL printed as a char = '", (type char al), "'", nl );
```

The same is true for the *stdin.get* routine. It will always read a hexadecimal value for a register unless you coerce its type to something other than *byte*, *word*, or *dword*.

You will see some additional uses for register type coercion in the next chapter.

## 2.7 The Stack Segment and the Push and Pop Instructions

This chapter mentioned that all variables you declare in the VAR section wind up in the stack memory segment (see “The Var Section” on page 163). However, VAR objects are not the only things that wind up in the stack segment in memory; your programs manipulate data in the stack segment in many different ways. This section introduces a set of instructions, the PUSH and POP instructions, that also manipulate data in the stack segment.

The stack segment in memory is where the 80x86 maintains the *stack*. The stack is a dynamic data structure that grows and shrinks according to certain memory needs of the program. The stack also stores important information about program including local variables, subroutine information, and temporary data.

The 80x86 stack is controlled by the ESP (stack pointer) register. When your program begins execution, the operating system initializes ESP with the address of the last memory location in the stack memory segment. Data is written to the stack segment by “pushing” data onto the stack and “popping” or “pulling” data off of the stack. Whenever you push data onto the stack, the 80x86 decrements the stack pointer by the size of the data you are pushing and then it copies the data to memory where ESP is then pointing. As a concrete example, consider the 80x86 PUSH instruction:

```
push( reg16 );
push( reg32 );
push( memory16 );
push( memory32 );
pushw( constant );
pushd( constant );
```

These six forms allow you to push *word* or *dword* registers, memory locations, and constants. You should specifically note that you cannot push *byte* typed objects onto the stack.

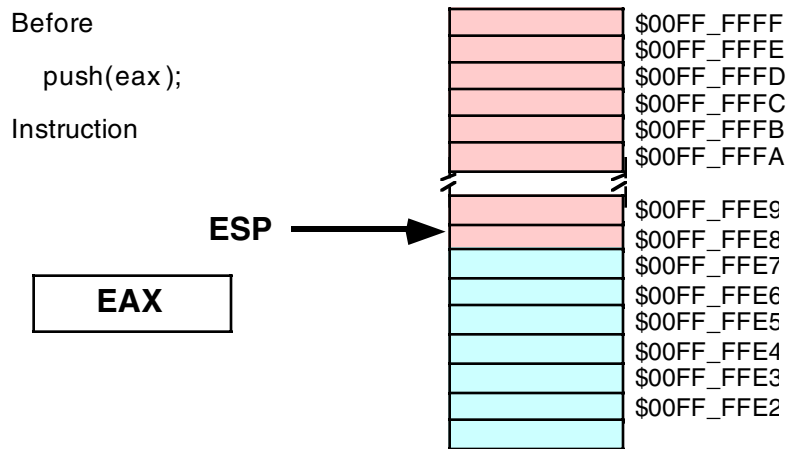
### 2.7.1 The Basic PUSH Instruction

The PUSH instruction does the following:

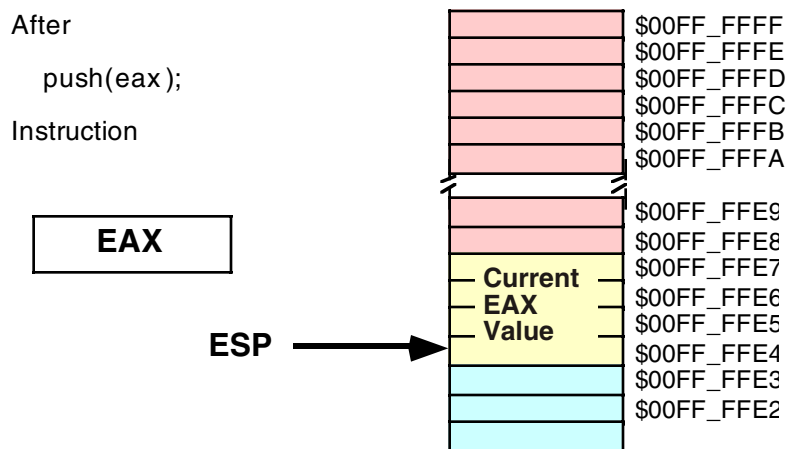
```
ESP := ESP - Size_of_Register_or_Memory_Operand (2 or 4)
[ESP] := Operand's_Value
```

The PUSHW and PUSHHD operand sizes are always two or four bytes, respectively.

Assuming that ESP contains \$00FF\_FFE8, then the instruction “PUSH( EAX );” will set ESP to \$00FF\_FFE4 and store the current value of EAX into memory location \$00FF\_FFE4 as shown in Figure 2.9 and Figure 2.10:



**Figure 2.9** Stack Segment Before “PUSH( EAX );” Operation



**Figure 2.10** Stack Segment After “PUSH( EAX );” Operation

Note that the “PUSH( EAX );” instruction does not affect the value in the EAX register.

Although the 80x86 supports 16-bit push operations, these are intended primarily for use in 16-bit environments such as DOS. For maximum performance, the stack pointer should always be an even multiple of four; indeed, your program may malfunction under Win32 if ESP contains a value that is not a multiple of four and you make an HLA Standard Library or Win32 API call. The only reason for push less than four bytes at a time on the stack is because you’re building up a double via two successive word pushes.

## 2.7.2 The Basic POP Instruction

To retrieve data you’ve pushed onto the stack, you use the POP instruction. The basic POP instruction allows the following different forms:

```
pop( reg16 );
pop( reg32 );
pop( memory16 );
```

```
pop( memory32 );
```

Like the PUSH instruction, the POP instruction only supports 16-bit and 32-bit operands; you cannot pop an eight-bit value from the stack. Also like the PUSH instruction, you should avoid popping 16-bit values (unless you do two 16-bit pops in a row) because 16-bit pops may leave the ESP register containing a value that is not an even multiple of four. One major difference between PUSH and POP is that you cannot POP a constant value (which makes sense, because the operand for PUSH is a source operand while the operand for POP is a destination operand).

Formally, here’s what the POP instruction does:

```
Operand := [ESP]
ESP := ESP + Size_of_Operand (2 or 4)
```

As you can see, the POP operation is the converse of the PUSH operation. Note that the POP instruction copies the data from memory location [ESP] before adjusting the value in ESP. See Figure 2.11 and Figure 2.12 for details on this operation:

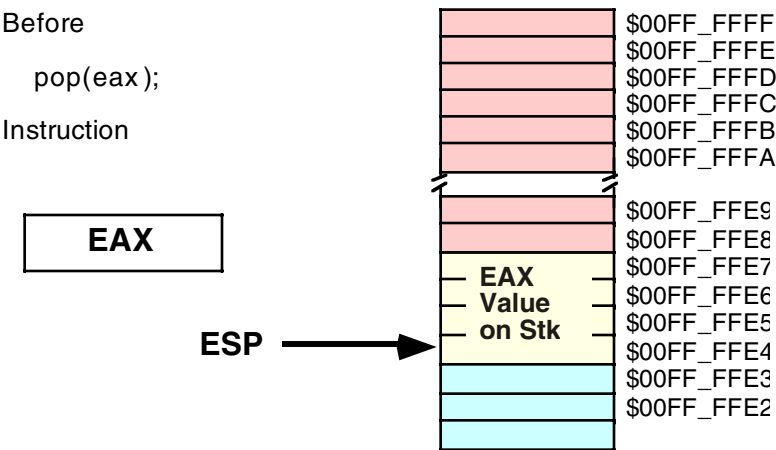


Figure 2.11 Memory Before a “POP( EAX );” Operation

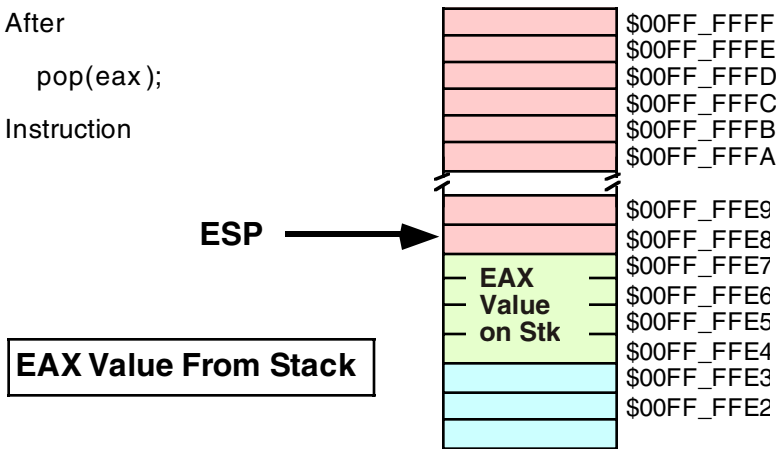


Figure 2.12 Memory After the “POP( EAX );” Instruction

Note that the value popped from the stack is still present in memory. Popping a value does not erase the value in memory, it just adjusts the stack pointer so that it points at the next value above the popped value. However, you should never attempt to access a value you've popped off the stack. The next time something is pushed onto the stack, the popped value will be obliterated. Since your code isn't the only thing that uses the stack (i.e., the operating system uses the stack as do other subroutines), you cannot rely on data remaining in stack memory once you've popped it off the stack.

---

### 2.7.3 Preserving Registers With the PUSH and POP Instructions

Perhaps the most common use of the PUSH and POP instructions is to save register values during intermediate calculations. A problem with the 80x86 architecture is that it provides very few general purpose registers. Since registers are the best place to hold temporary values, and registers are also needed for the various addressing modes, it is very easy to run out of registers when writing code that performs complex calculations. The PUSH and POP instructions can come to your rescue when this happens.

Consider the following program outline:

```
<< Some sequence of instructions that use the EAX register >>

<< Some sequence of instructions that need to use EAX, for a
    different purpose than the above instructions >>

<< Some sequence of instructions that need the original value in EAX >>
```

The PUSH and POP instructions are perfect for this situation. By inserting a PUSH instruction before the middle sequence and a POP instruction after the middle sequence above, you can preserve the value in EAX across those calculations:

```
<< Some sequence of instructions that use the EAX register >>
push( eax );
<< Some sequence of instructions that need to use EAX, for a
    different purpose than the above instructions >>
pop( eax );
<< Some sequence of instructions that need the original value in EAX >>
```

The PUSH instruction above copies the data computed in the first sequence of instructions onto the stack. Now the middle sequence of instructions can use EAX for any purpose it chooses. After the middle sequence of instructions finishes, the POP instruction restores the value in EAX so the last sequence of instructions can use the original value in EAX.

---

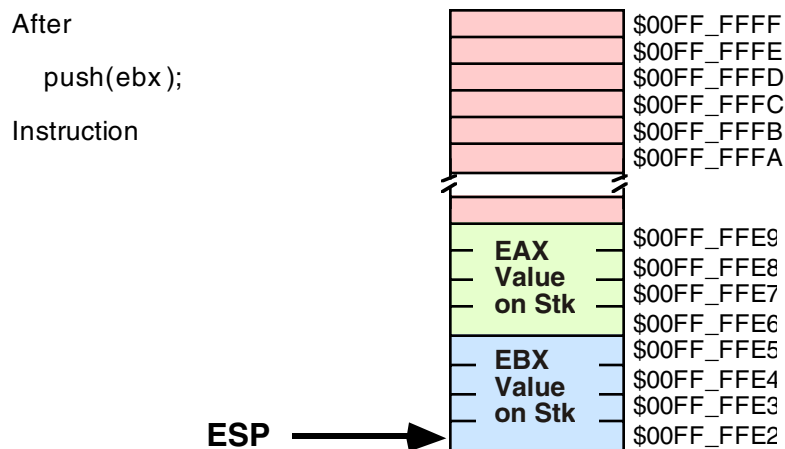
### 2.7.4 The Stack is a LIFO Data Structure

Of course, you can push more than one value onto the stack without first popping previous values off the stack. However, the stack is a last-in, first-out (LIFO) data structure, so you must be careful how you push and pop multiple values. For example, suppose you want to preserve EAX and EBX across some block of instructions, the following code demonstrates the obvious way to handle this:

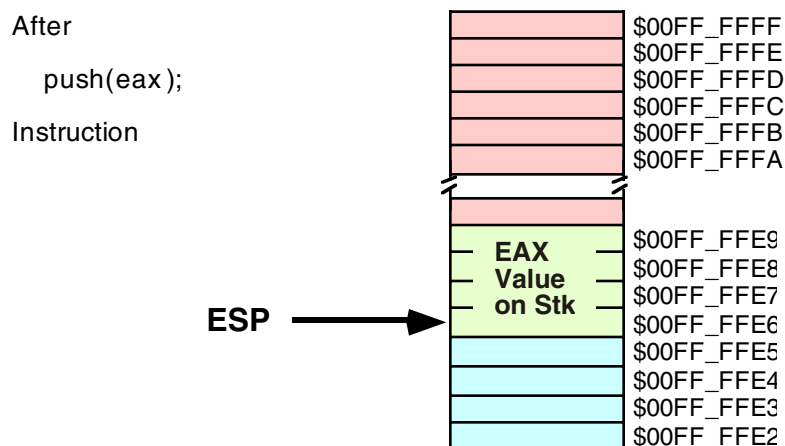
```
push( eax );
push( ebx );
<< Code that uses EAX and EBX goes here >>
pop( eax );
pop( ebx );
```

Unfortunately, this code will not work properly! Figures 2.13, 2.14, 2.15, and 2.16 show the problem. Since this code pushes EAX first and EBX second, the stack pointer is left pointing at the value in EBX pushed

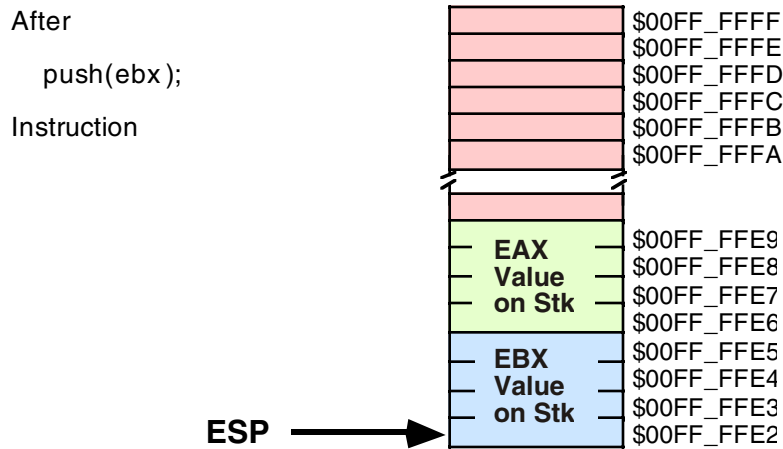
onto the stack. When the POP( EAX ) instruction comes along, it removes the value that was originally in EBX from the stack and places it in EAX! Likewise, the POP( EBX ) instruction pops the value that was originally in EAX into the EBX register. The end result is that this code has managed to swap the values in the registers by popping them in the same order that it pushed them.



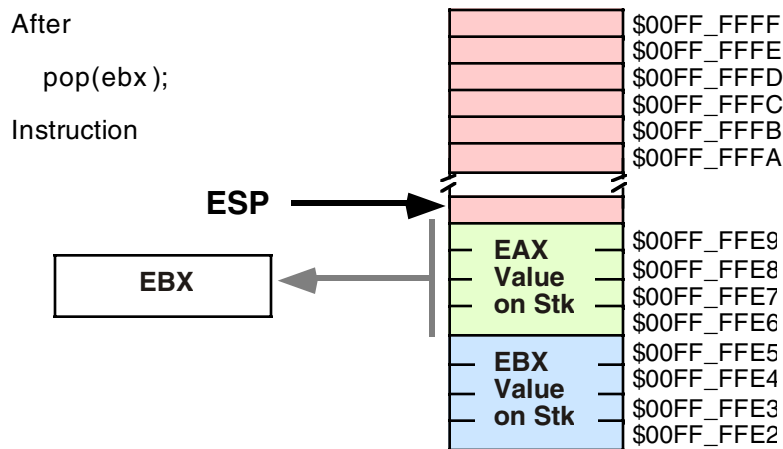
**Figure 2.13** Stack After Pushing EAX



**Figure 2.14** Stack After Pushing EBX



**Figure 2.15** Stack After Popping EAX



**Figure 2.16** Stack After Popping EBX

To rectify this problem, you need to note that the stack is a last-in, first-out data structure, so the first thing you must pop is the last thing you've pushed onto the stack. To do this, you must always observe the following maxim:

- ❑ **Always pop values in the reverse order that you push them.**

The correction to the previous code is

```
push( eax );
push( ebx );
<< Code that uses EAX and EBX goes here >>
pop( ebx );
pop( eax );
```

Another important maxim to remember is

- ❑ **Always pop exactly the same number of bytes that you push.**

This generally means that the number of pushes and pops must exactly agree. If you have too few pops, you will leave data on the stack which may confuse the running program<sup>13</sup>; If you have too many pops, you will accidentally remove previously pushed data, often with disastrous results.

A corollary to the maxim above is “Be careful when pushing and popping data within a loop.” Often it is quite easy to put the pushes in a loop and leave the pops outside the loop (or vice versa), creating an inconsistent stack. Remember, it is the execution of the PUSH and POP instructions that matters, not the number of PUSH and POP instructions that appear in your program. At run-time, the number (and order) of the PUSH instructions the program executes must match the number (and reverse order) of the POP instructions.

---

### 2.7.5 Other PUSH and POP Instructions

The 80x86 provides several additional PUSH and POP instructions in addition to the basic instructions described in the previous sections. These instructions include the following:

- PUSHA
- PUSHAD
- PUSHF
- PUSHFD
- POPA
- POPAD
- POPF
- POPFD

The PUSHA instruction pushes all the general-purpose 16-bit registers onto the stack. This instruction is primarily intended for older 16-bit operating systems like DOS. In general, you will have very little need for this instruction. The PUSHAD instruction pushes the registers onto the stack in the following order:

ax  
cx  
dx  
bx  
sp  
bp  
si  
di

The PUSHAD instruction pushes all the 32-bit (dword) registers onto the stack. It pushes the registers onto the stack in the following order:

eax  
ecx  
edx  
ebx  
esp  
ebp  
esi  
edi

Since the SP/ESP register is inherently modified by the PUSHAD and PUSHAD instructions, you may wonder why Intel bothered to push it at all. It was probably easier in the hardware to go ahead and push SP/ESP rather than make a special case out of it. In any case, these instructions do push SP or ESP so don't worry about it too much - there is nothing you can do about it.

---

13. You'll see why when we cover procedures.

The POPA and POPAD instructions provide the corresponding “pop all” operation to the PUSHHA and PUSHAD instructions. This will pop the registers pushed by PUSHHA or PUSHAD in the appropriate order (that is, POPA and POPAD will properly restore the register values by popping them in the reverse order that PUSHHA or PUSHAD pushed them).

Although the PUSHHA/POPA and PUSHAD/POPAD sequences are short and convenient, they are actually slower than the corresponding sequence of PUSH/POP instructions, this is especially true when you consider that you rarely need to push a majority, much less all the registers<sup>14</sup>. So if you’re looking for the maximum amount of speed, you should carefully consider whether to use the PUSHHA(D)/POPA(D) instructions. This text generally opts for convenience and readability; so it will use the PUSHAD and POPAD instructions without worrying about lost efficiency.

The PUSHF, PUSHFD, POPF, and POPFD instructions push and pop the (E)FLAGS register. These instructions allow you to preserve condition code and other flag settings across the execution of some sequence of instructions. Unfortunately, unless you go to a lot of trouble, it is difficult to preserve individual flags. When using the PUSHF(D) and POPF(D) instructions it’s an all or nothing proposition - you preserve all the flags when you push them, you restore all the flags when you pop them.

Like the PUSHHA and POPA instructions, you should really use the PUSHFD and POPFD instructions to push the full 32-bit version of the EFLAGS register. Although the extra 16-bits you push and pop are essentially ignored when writing applications, you still want to keep the stack aligned by pushing and popping only double words.

---

## 2.7.6 Removing Data From the Stack Without Popping It

Once in a while you may discover that you’ve pushed data onto the stack that you no longer need. Although you could pop the data into an unused register or memory location, there is an easier way to remove unwanted data from the stack - simply adjust the value in the ESP register to skip over the unwanted data on the stack.

Consider the following dilemma:

```
push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into EAX and EBX >>

if( Calculation_was_performed ) then

    // Whoops, we don't want to pop EAX and EBX!
    // What to do here?

else

    // No calculation, so restore EAX, EBX.

    pop( ebx );
    pop( eax );

endif;
```

Within the THEN section of the IF statement, this code wants to remove the old values of EAX and EBX without otherwise affecting any registers or memory locations. How to do this?

Since the ESP register simply contains the memory address of the item on the top of the stack, we can remove the item from the top of stack by adding the size of that item to the ESP register. In the example

---

14. For example, it is extremely rare for you to need to push and pop the ESP register with the PUSHAD/POPAD instruction sequence.

above, we want to remove two double word items from the top of stack, so we can easily accomplish this by adding eight to the stack pointer:

```

push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into EAX and EBX >>

if( Calculation_was_performed ) then

    add( 8, ESP );    // Remove unneeded EAX and EBX values from the stack.

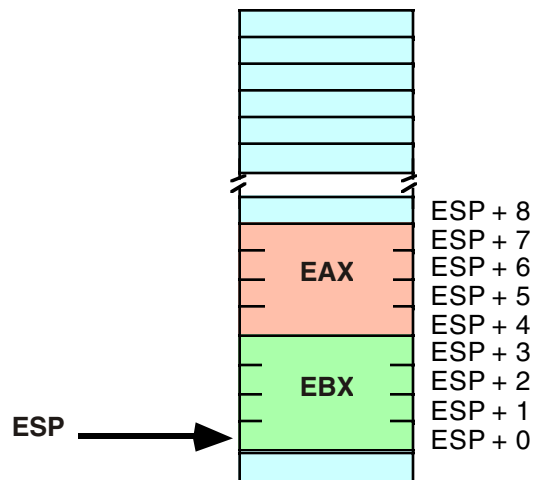
else

    // No calculation, so restore EAX, EBX.

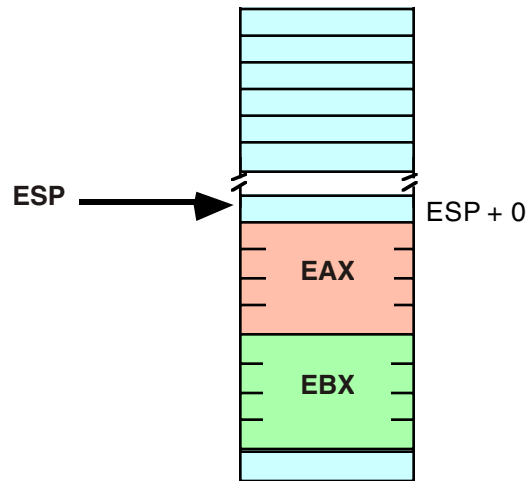
    pop( ebx );
    pop( eax );

endif;

```



**Figure 2.17** Removing Data from the Stack, Before ADD( 8, ESP )



**Figure 2.18 Removing Data from the Stack, After `ADD( 8, ESP );`**

Effectively, this code pops the data off the stack without moving it anywhere. Also note that this code is more efficient than two dummy POP instructions because it can remove any number of bytes from the stack with a single ADD instruction.

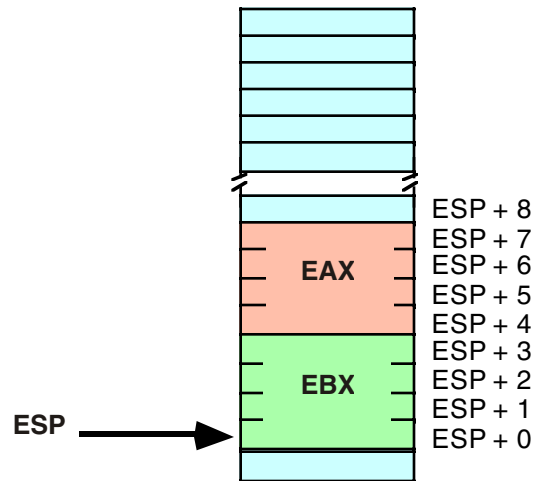
**Warning:** remember to keep the stack aligned on a double word boundary. Therefore, you should always add a constant that is an even multiple of four to ESP when removing data from the stack.

### 2.7.7 Accessing Data You’ve Pushed on the Stack Without Popping It

Once in a while you will push data onto the stack and you will want to get a copy of that data’s value, or perhaps you will want to change that data’s value, without actually popping the data off the stack (that is, you wish to pop the data off the stack at a later time). The 80x86 “[reg<sub>32</sub> + offset]” addressing mode provides the mechanism for this.

Consider the stack after the execution of the following two instructions (see Figure 2.19):

```
push( eax );
push( ebx );
```



**Figure 2.19** Stack After Pushing EAX and EBX

If you wanted to access the original EBX value without removing it from the stack, you could cheat and pop the value and then immediately push it again. Suppose, however, that you wish to access EAX's old value; or some other value even farther up on the stack. Popping all the intermediate values and then pushing them back onto the stack is problematic at best, impossible at worst. However, as you will notice from Figure 2.19, each of the values pushed on the stack is at some offset from the ESP register in memory. Therefore, we can use the "[ESP + offset]" addressing mode to gain direct access to the value we are interested in. In the example above, you can reload EAX with its original value by using the single instruction:

```
mov( [esp+4], eax );
```

This code copies the four bytes starting at memory address ESP+4 into the EAX register. This value just happens to be the value of EAX that was earlier pushed onto the stack. This same technique can be used to access other data values you've pushed onto the stack.

**Warning:** Don't forget that the offsets of values from ESP into the stack change every time you push or pop data. Abusing this feature can create code that is hard to modify; if you use this feature throughout your code, it will make it difficult to push and pop other data items between the point you first push data onto the stack and the point you decide to access that data again using the "[ESP + offset]" memory addressing mode.

In the previous section it pointed out how to remove data from the stack by adding a constant to the ESP register. That code example could probably be written more safely as:

```
push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into EAX and EBX >>

if( Calculation_was_performed ) then

    // Overwrite saved values on stack with new EAX/EBX values.
    // (so the pops that follow won't change the values in EAX/EBX.)
    mov( eax, [esp+4] );
    mov( ebx, [esp] );

endif;
pop( ebx );
pop( eax );
```

In this code sequence, the calculated result was stored over the top of the values saved on the stack. Later on, when the values are popped off the stack, the program loads these calculated values into EAX and EBX.

## 2.8 Dynamic Memory Allocation and the Heap Segment

Although static and automatic variables are all a simple program may need, more sophisticated programs need the ability to allocate and deallocate storage dynamically (at run-time) under program control. In the C language, you would use the *malloc* and *free* functions for this purpose. C++ provides the *new* and *delete* operators. Pascal uses *new* and *dispose*. Other languages provide comparable routines. These memory allocation routines share a couple of things in common: they let the programmer request how many bytes of storage to allocate, they return a *pointer* to the newly allocated storage, and they provide a facility for returning the storage to the system so the system can reuse it in a future allocation call. As you've probably guessed, HLA also provides a set of routines in the HLA Standard Library that handles memory allocation and deallocation.

The HLA Standard Library *malloc* and *free* routines handle the memory allocation and deallocation chores (respectively)<sup>15</sup>. The *malloc* routine uses the following calling sequence:

```
malloc( Number_of_Bytes_Requested );
```

The single parameter is a *dword* value (an unsigned constant) specifying the number of bytes of storage you are requesting. This procedure calls the Windows API to allocate storage in the *heap* segment in memory. Windows locates an unused block of memory of the specified size in the heap segment and marks the block as “in use” so that future calls to *malloc* will not reallocate this same storage. After marking the block as “in use” the *malloc* routine returns a pointer to the first byte of this storage in the EAX register.

For many objects, you will know the number of bytes you need to represent that object in memory. For example, if you wish to allocate storage for an *uns32* variable, you could use the following call to the *malloc* routine:

```
malloc( 4 );
```

Although you can specify a literal constant as this example suggests, it's generally a poor idea to do so when allocating storage for a specific data type. Instead, use the HLA built-in compile-time function *@size* to compute the size of some data type. The *@size* function uses the following syntax:

```
@size( variable_or_type_name )
```

The *@size* function returns an unsigned integer constant that specifies the size of its parameter in bytes. So you should rewrite the previous call to *malloc* as follows:

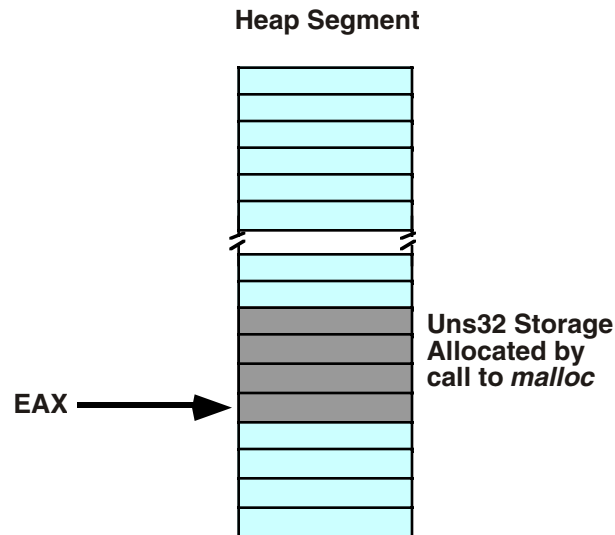
```
malloc( @size( uns32 ) );
```

This call will properly allocate a sufficient amount of storage for the specified object, regardless of its type. While it is unlikely that the number of bytes required by an *uns32* object will ever change, this is not necessarily true for other data types; so you should always use *@size* rather than a literal constant in these calls.

Upon return from the *malloc* routine, the EAX register contains the address of the storage you have requested (see Figure 2.20):

---

15. HLA provides some other memory allocation and deallocation routines as well. See the HLA Standard Library documentation for more details.



**Figure 2.20 Call to Malloc Returns a Pointer in the EAX Register**

To access the storage *malloc* allocates you must use a register indirect addressing mode. The following code sequence demonstrates how to assign the value 1234 to the *uns32* variable *malloc* creates:

```
malloc( @size( uns32 ) );
mov( 1234, (type uns32 [eax]));
```

Note the use of the type coercion operation. This is necessary in this example because anonymous variables don't have a type associated with them and the constant 1234 could be a *word* or *dword* value. The type coercion operator eliminates the ambiguity.

A call to the *malloc* routine is not guaranteed to succeed. By default, windows only reserves about a megabyte for the heap; HLA modifies this default to about 16 megabytes. If there isn't a single contiguous block of free memory in the heap segment that is large enough to satisfy the request, then the *malloc* routine will raise an *ex.MemoryAllocationFailure* exception. If you do not provide a TRY..EXCEPTION..ENDTRY handler to deal with this situation, a memory allocation failure will cause your program to abort execution. Since most programs do not allocate massive amounts of dynamic storage using *malloc*, this exception rarely occurs. However, you should never assume that the memory allocation will always occur without error.

When you are done using a value that *malloc* allocates on the heap, you can release the storage (that is, mark it as "no longer in use") by calling the *free* procedure. The *free* routine requires a single parameter that must be an address previously returned by the *malloc* routine that you have not already freed. The following code fragment demonstrates the nature of the *malloc/free* pairing:

```
malloc( @size( uns32 ) );

<< use the storage pointed at by EAX >>
<< Note: this code must not modify EAX >>

free( eax );
```

This code demonstrates a very important point - in order to properly free the storage that *malloc* allocates, you must preserve the value that *malloc* returns. There are several ways to do this if you need to use EAX for some other purpose; you could save the pointer value on the stack using PUSH and POP instructions or you could save EAX's value in a variable until you need to free it.

Storage you release is available for reuse by future calls to the *malloc* routine. Like automatic variables you declare in the VAR section, the ability to allocate storage while you need it and then free the storage for other use when you are done with it improves the memory efficiency of your program. By deallocating storage once you are finished with it, your program can reuse that storage for other purposes allowing your program to operate with less memory than it would if you statically allocated storage for the individual objects.

There are several problems that can occur when you use pointers. You should be aware of a few common errors that beginning programmers make when using dynamic storage allocation routines like *malloc* and *free*:

- Mistake #1: Continuing to refer to storage after you free it. Once you return storage to the system via the call to *free*, you should no longer access the data allocated by the call to *malloc*. Doing so may cause a protection fault or, worse yet, corrupt other data in your program without indicating an error.
- Mistake #2: Calling *free* twice to release a single block of storage. Doing so may accidentally free some other storage that you did not intend to release or, worse yet, it may corrupt the system memory management tables.

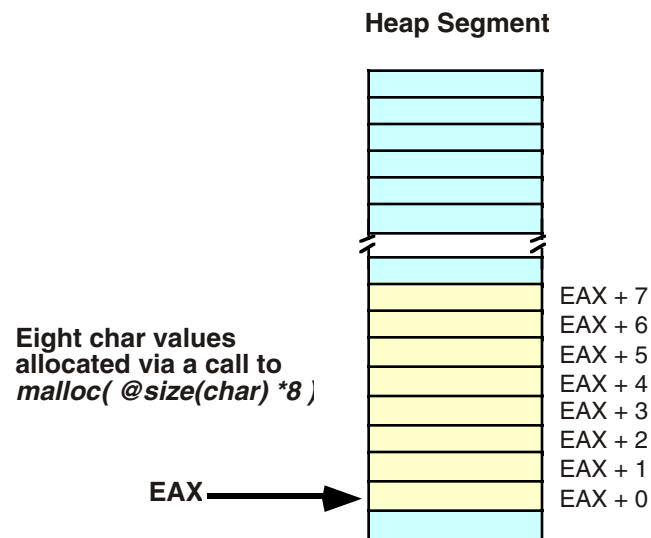
A later chapter will discuss some additional problems you will typically encounter when dealing with dynamically allocated storage.

The examples thus far in this section have all allocated storage for a single unsigned 32-bit object. Obviously you can allocate storage for any data type using a call to *malloc* by simply specifying the size of that object as *malloc*'s parameter. It is also possible to allocate storage for a sequence of contiguous objects in memory when calling *malloc*. For example, the following code will allocate storage for a sequence of 8 characters:

```
malloc( @size( char ) * 8 );
```

Note the use of the constant expression to compute the number of bytes required by an eight-character sequence. Since “@size(char)” always returns a constant value (one in this case), the compiler can compute the value of the expression “@size(char) \* 8” without generating any extra machine instructions.

Calls to *malloc* always allocate multiple bytes of storage in contiguous memory locations. Hence the former call to *malloc* produces the sequence appearing in Figure 2.21:



**Figure 2.21 Allocating a Sequence of Eight Character Objects Using Malloc**

To access these extra character values you use an offset from the base address (contained in EAX upon return from *malloc*). For example, “MOV( CH, [EAX + 2] );” stores the character in the CH register into the

third byte that *malloc* allocates. You can also use an addressing mode like “[EAX + EBX]” to step through each of the allocated objects under program control. For example, the following code will set all the characters in a block of 128 bytes to the NULL character (#0):

```
malloc( 128 );
mov( 0, ebx );
while( ebx < 128 ) do

    mov( 0, (type byte [eax+ebx]) );
    add( 1, ebx );

endwhile;
```

The chapter on arrays, later in this text, discusses additional ways to deal with blocks of memory.

---

## 2.9 The INC and DEC Instructions

As the example in the last section indicates, indeed, as several examples up to this point have indicated, adding or subtracting one from a register or memory location is a very common operation. In fact, this operation is so common that Intel’s engineer’s included a pair of instructions to perform these specific operations: the INC (increment) and DEC (decrement) instructions.

The INC and DEC instructions use the following syntax:

```
inc( mem/reg );
dec( mem/reg );
```

The single operand can be any legal eight-bit, 16-bit, or 32-bit register or memory operand. The INC instruction will add one to the specified operand, the DEC instruction will subtract one from the specified operand.

These two instructions are slightly more efficient (they are smaller) than the corresponding ADD or SUB instructions. There is also one slight difference between these two instructions and the corresponding ADD or SUB instructions: they do not affect the carry flag.

As an example of the INC instruction, consider the example from the previous section, recoded to use INC rather than ADD:

```
malloc( 128 );
mov( 0, ebx );
while( ebx < 128 ) do

    mov( 0, (type byte [eax+ebx]) );
    inc( ebx );

endwhile;
```

---

## 2.10 Obtaining the Address of a Memory Object

In the section “The Register Indirect Addressing Modes” on page 151 this chapter discusses how to use the address-of operator, “&”, to take the address of a static variable<sup>16</sup>. Unfortunately, you cannot use the address-of operator to take the address of an automatic variable (one you declare in the VAR section), you cannot use it to compute the address of an anonymous variable, nor can you use this operator to take the address of a memory reference that uses an indexed or scaled indexed addressing mode (even if a static variable is part of the address expression). You may only use the address-of operator to take the address of a static variable that uses the displacement-only memory addressing mode. Often, you will need to take the

---

16. A static variable is one that you declare in the *static*, *readonly*, *storage*, or *data* sections of your program.

address of other memory objects as well; fortunately, the 80x86 provides the load effective address instruction, LEA, to give you this capability.

The LEA instruction uses the following syntax:

```
lea( reg32, Memory_operand );
```

The first operand must be a 32-bit register, the second operand can be any legal memory reference using any valid memory addressing mode. This instruction will load the address of the specified memory location into the register. This instruction does not modify the value of the memory operand in any way, nor does it reference that value in memory.

Once you load the effective address of a memory location into a 32-bit general purpose register, you can use the register indirect, indexed, or scaled indexed addressing modes to access the data at the specified memory address. For example, consider the following code:

```
data
    b:byte;
        byte 7, 0, 6, 1, 5, 2, 4, 3;
        .
        .
        .
    lea( ebx, b );
    mov( 0, ecx );
    while( ecx < 8 ) do

        stdout.put( "[ebx+ecx]=", (type byte [ebx+ecx]), nl );
        inc( ecx );

    endwhile;
```

This code steps through each of the eight bytes following the *b* label in the DATA section and prints their values. Note the use of the “[ebx+ecx]” addressing mode. The EBX register holds the base address of the list (that is, the address of the first item in the list) and ECX contains the byte index into the list.

---

## 2.11 Bonus Section: The HLA Standard Library CONSOLE Module

The HLA Standard Library contains a module that lets you control output to the *console* device. The console device is the virtual text/video display of the command window. The procedures in the console module let you clear the screen, position the cursor, output text to a specific cursor position in the window, adjust the window size, control the color of the output characters, handle mouse events, and do other console-related operations. The judicious use of the console module lets you transform a drab, boring text-based application into a visually appealing text-based application. The sample programs in this section demonstrate some of the capabilities of the HLA Standard Library console module.

Note: to use the console module routines in your program you must include one (or both) of the following statements in your HLA program:

```
#include( "stdlib.hhf" );
#include( "console.hhf" );
```

---

### 2.11.1 Clearing the Screen

Perhaps the most important routine in the console module, based on HLA user requests, is the *console.cls()* procedure. This routine clears the screen and positions the cursor to coordinate (0,0)<sup>17</sup>. The following sample application demonstrates the use of this routine.

---

17. In console coordinates, location (0,0) is the upper left hand corner of the screen. The X coordinates increase as you progress from left to right and the Y coordinates increase as you progress from top to bottom on the screen.

---



---

```

program testCls;
#include( "stdlib.hhf" );
begin testCls;

    // Throw up some text to prove that
    // this program really clears the screen:

    stdout.put
    (
        nl,
        "HLA console.cls() Test Routine", nl
        "-----", nl
        nl
        "This routine will clear the screen and move the cursor to (0,0)", nl
        "then it will print a short message and quit", nl
        nl
        "Press the Enter key to continue:"
    );

    // Make the user hit Enter to continue. This is so that they
    // can see that the screen is not blank.

    stdin.ReadLn();

    // Okay, clear the screen and print a simple message:

    console.cls();
    stdout.put( "The screen was cleared.", nl );

end testCls;

```

---



---

### Program 3.2 The console.cls() Routine

---



---

## 2.11.2 Positioning the Cursor

After clearing the screen, the most often requested console capability is cursor positioning. The HLA Standard Library *console.gotoxy* procedure handles this task. The *console.gotoxy* call uses the following syntax:

```
console.gotoxy( RowPosition, ColumnPosition );
```

Note that *RowPosition* and *ColumnPosition* must be 16-bit values (constants, variables, or registers).

The astute reader will notice that the first parameter, the *RowPosition*, is actually the Y coordinate and the second parameter, *ColumnPosition*, is the X coordinate. This coordinate ordering may seem counter-intuitive given the name of the procedure (*gotoxy*, with X appearing in the name before Y). However, in actual practice most people find it more intuitive to specify the Y coordinate first and the X coordinate second. The name “gotoxy” sounds better than “gotoyx” so HLA uses “gotoxy” despite the minor inconsistency between the name and the parameter ordering.

The following program demonstrates the *console.gotoxy* procedure:

---



---

```

program testGotoxy;
#include( "stdlib.hhf" );

```

```

var
  x:int16;
  y:int16;

begin testGotoxy;

  // Throw up some text to prove that
  // this program really clears the screen:

  stdout.put
  (
    nl,
    "HLA console.gotoxy() Test Routine", nl,
    "-----", nl,
    nl,
    "This routine will clear the screen then demonstrate the use", nl,
    "of the gotoxy routine to position the cursor at various", nl,
    "points on the screen.",nl,
    nl,
    "Press the Enter key to continue:"
  );

  // Make the user hit Enter to continue. This is so that they
  // can control when they see the effect of console.gotoxy.

  stdin.ReadLn();

  // Okay, clear the screen:

  console.cls();

  // Now demonstrate the gotoxy routine:

  console.gotoxy( 5,10 );
  stdout.put( "(5,10)" );

  console.gotoxy( 10, 5 );
  stdout.put( "(10,5)" );

  mov( 20, x );
  for( mov( 0,y ); y<20; inc(y)) do

    console.gotoxy( y, x );
    stdout.put( "(", x, ",", y, ")" );
    inc( x );

  endfor;

end testGotoxy;

```

---

**Program 3.3**    The console.gotoxy(row,column) Routine

---

### 2.11.3 Locating the Cursor

In addition to letting you specify a new cursor position, the HLA console module provides routines that let you determine the current cursor position. The *console.getX()* and *console.getY()* routines return the X

and Y coordinates (respectively) of the current cursor position in the EAX register. The following program demonstrates the use of these two functions.

---

---

```

program testGetxy;
#include( "stdlib.hhf" );

var
  x:uns32;
  y:uns32;

begin testGetxy;

  // Begin by getting the current cursor position

  console.getX();
  mov( eax, x );

  console.getY();
  mov( eax, y );

  // Clear the screen and print a banner message:

  console.cls();

  stdout.put
  (
    nl,
    "HLA console.GetX() and console.GetY() Test Routine", nl,
    "-----", nl,
    nl,
    "This routine will clear the screen then demonstrate the use", nl,
    "of the GetX and GetY routines to reposition the cursor", nl,
    "to its original location on the screen.",nl,
    nl,
    "Press the Enter key to continue:"
  );

  // Make the user hit Enter to continue. This is so that they
  // can control when they see the effect of console.gotoxy.

  stdin.ReadLn();

  // Now demonstrate the GetX and GetY routines by calling
  // the gotoxy routine to move the cursor back to its original
  // position.

  console.gotoxy( (type uns16 y), (type uns16 x) );
  stdout.put( "*<- Cursor was originally here.", nl );

end testGetxy;

```

---

---

#### Program 3.4 The console.GetX() and console.GetY() Routines

## 2.11.4 Text Attributes

The HLA console module lets you specify the color of the text you print to the console window. You may specify one of sixteen different foreground or background colors for each character you print. The foreground color is the color of the dots that make up the actual character on the display; the background color is the color of the other pixels (dots) in the character cell. The console module supports any of the following available foreground and background colors:

```
win.bgnd_Black
win.bgnd_Blue
win.bgnd_Green
win.bgnd_Cyan
win.bgnd_Red
win.bgnd_Magenta
win.bgnd_Brown
win.bgnd_LightGray
win.bgnd_DarkGray
win.bgnd_LightBlue
win.bgnd_LightGreen
win.bgnd_LightCyan
win.bgnd_LightRed
win.bgnd_LightMagenta
win.bgnd_Yellow
win.bgnd_White
```

```
win.fgnd_Black
win.fgnd_Blue
win.fgnd_Green
win.fgnd_Cyan
win.fgnd_Red
win.fgnd_Magenta
win.fgnd_Brown
win.fgnd_LightGray
win.fgnd_DarkGray
win.fgnd_LightBlue
win.fgnd_LightGreen
win.fgnd_LightCyan
win.fgnd_LightRed
win.fgnd_LightMagenta
win.fgnd_Yellow
win.fgnd_White
```

The “win32.hhf” header file defines the symbolic constants for these colors. Therefore, you must include one of the following statements in your program to have access to these colors:

```
#include( "stdlib.hhf" );
#include( "win32.hhf" );
```

The first routine to take advantage of these color attributes is the *console.setOutputAttr* routine. A call to this procedure uses the following syntax:

```
console.setOutputAttr( ColorValues );
```

The single parameter to this routine is a single foreground or background color, or a pair of colors (one background and one foreground) combined with the “|” operator<sup>18</sup>. E.g.,

```
console.setOutputAttr( win.fgnd_Yellow );
console.setOutputAttr( win.bgnd_White );
```

---

18. This is the bitwise OR operator.

```
console.setOutputAttr( win.fgnd_Yellow | win.bgnd_Blue );
```

If you do not specify both colors, the default for the missing color is black. Therefore, the first call above sets the foreground color to yellow and the background color to black. Likewise, the second call above sets the foreground color to black and the background color to white.

The *console.setOutputAttr* routine does not automatically change the color of all characters on the screen. Instead, it only affects the color of the characters output after the call. Therefore, you can switch between various colors on a character-by-character basis, as necessary. The following sample program demonstrates the use of *console.setOutputAttr* routine.

```
program testSetOutputAttr;
#include( "stdlib.hhf" );

var
  x:uns32;
  y:uns32;

begin testSetOutputAttr;

  // Clear the screen and print a banner message:

  console.cls();

  console.setOutputAttr( win.fgnd_LightRed | win.bgnd_Black );
  stdout.put
  (
    nl,
    "HLA console.setOutputAttr Test Routine", nl,
    "-----", nl,
    nl,
    "Press the Enter key to continue:"
  );

  // Make the user hit Enter to continue.

  stdin.ReadLn();

  // Display the following text in a different color.

  console.setOutputAttr( win.fgnd_Yellow | win.bgnd_Blue );
  stdout.put
  (
    "                                ", nl
    " In blue and yellow          ", nl,
    "                                ", nl,
    " Press Enter to continue ", nl
    "                                ", nl
    nl
  );
  stdin.ReadLn();

  // Note: set the attributes back to black and white when
  // the program exits so the console window doesn't continue
  // displaying text in Blue and Yellow.

  console.setOutputAttr( win.fgnd_White | win.bgnd_Black );
```

```
end testSetOutputAttr;
```

---

### Program 3.5 The console.setOutputAttr Routine

---

## 2.11.5 Filling a Rectangular Section of the Screen

The *console.fillRect* procedure gives you the ability to fill a rectangular portion of the screen with a single character and a set of text attributes. The call to this routine uses the following syntax:

```
console.fillRect( ULrow, ULcol, LRow, LCol, character, attr );
```

The *ULrow* and *ULcol* parameters must be 16-bit values that specify the row and column number of the upper left hand corner of the rectangle to draw. Likewise, the *LRow* and *LCol* parameters are 16-bit values that specify the lower right hand corner of the rectangle to draw. The *character* parameter is the character you wish to draw throughout the rectangular block. This is normally a space if you want to produce a simple rectangle. The *attr* parameter is a text attribute parameter, identical to the parameter for the *console.setOutputAttr* routine that the previous section describes. The following sample program demonstrates the use of the *console.fillRect* procedure.

---

```
program testFillRect;
#include( "stdlib.hhf" );

var
  x:uns32;
  y:uns32;

begin testFillRect;

  console.setOutputAttr( win.fgnd_LightRed | win.bgnd_Black );
  stdout.put
  (
    nl,
    "HLA console.fillRect Test Routine", nl,
    "-----", nl,
    nl,
    "Press the Enter key to continue:"
  );

  // Make the user hit Enter to continue.

  stdin.ReadLn();
  console.cls();

  // Test outputting rectangular blocks of color.
  // Note that the blocks are always filled with spaces,
  // so there is no need to specify a foreground color.

  console.fillRect( 2, 50, 5, 55, ' ', win.bgnd_Black );
  console.fillRect( 6, 50, 9, 55, ' ', win.bgnd_Green );
  console.fillRect( 10, 50, 13, 55, ' ', win.bgnd_Cyan );
  console.fillRect( 14, 50, 17, 55, ' ', win.bgnd_Red );
  console.fillRect( 18, 50, 21, 55, ' ', win.bgnd_Magenta );

  console.fillRect( 2, 60, 5, 65, ' ', win.bgnd_Brown );
  console.fillRect( 6, 60, 9, 65, ' ', win.bgnd_LightGray );
  console.fillRect( 10, 60, 13, 65, ' ', win.bgnd_DarkGray );
```

```

console.fillRect( 14, 60, 17, 65, ' ', win.bgnd_LightBlue );
console.fillRect( 18, 60, 21, 65, ' ', win.bgnd_LightGreen );

console.fillRect( 2, 70, 5, 75, ' ', win.bgnd_LightCyan );
console.fillRect( 6, 70, 9, 75, ' ', win.bgnd_LightRed );
console.fillRect( 10, 70, 13, 75, ' ', win.bgnd_LightMagenta );
console.fillRect( 14, 70, 17, 75, ' ', win.bgnd_Yellow );
console.fillRect( 18, 70, 21, 75, ' ', win.bgnd_White );

// Note: set the attributes back to black and white when
// the program exits so the console window doesn't continue
// displaying text in Blue and Yellow.

console.setOutputAttr( win.fgnd_White | win.bgnd_Black );

end testFillRect;

```

---

### Program 3.6 The console.fillRect Procedure

---

## 2.11.6 Console Direct String Output

Although you can use the standard output routines (e.g., *stdout.put*) to write text to the console window, the console module provides a couple of convenient routines that output strings to the display. These routines combine the standard library *stdout.puts* routine with *console.gotoxy* and *console.setOutputAttr*. Two common console output routines are

```

console.puts( Row, Col, StringToPrint );
console.putsx( Row, Col, Color, MaxChars, StringToPrint );

```

The *Row* and *Col* parameters specify the coordinate of the first output character. *StringToPrint* is the string to display at the specified coordinate. The *console.putsx* routine supports two additional parameters; *Color*, that specifies the output foreground and background colors for the text, and *MaxChars* that specifies the maximum number of characters to print from *StringToPrint*<sup>19</sup>. The following sample program demonstrates these two routines

---

```

program testPutsx;
#include( "stdlib.hhf" );

var
  x:uns32;
  y:uns32;

begin testPutsx;

  // Clear the screen and print a banner message:

  console.cls();

  // Note that console.puts always defaults to black and white text.
  // The following setOutputAttr call proves this.

```

---

19. If *StringToPrint* is a constant, then *MaxChars* should specify the exact length of the string. When you learn about string variables in the next chapter you will see the purpose of the *MaxChars* parameter; it lets you ensure that the text you output fits within a certain range of cells on the screen.

```

console.setOutputAttr( win.fgnd_LightRed | win.bgnd_Black );

// Display the text in black and white:

console.puts
(
    10,
    10,
    "HLA console.setOutputAttr Test Routine"
);
console.puts
(
    11,
    10,
    "-----"
);
console.puts
(
    13,
    10,
    "Press the Enter key to continue:"
);

// Make the user hit Enter to continue.

stdin.ReadLn();

// Demonstrate the console.putsx routine.
// Note that the colors set by putsx are
// "local" to this call. Hence, the current
// output attribute colors will not be affected
// by this call.

console.putsx
(
    15,
    15,
    win.bgnd_White | win.fgnd_Blue,
    35,
    "Putsx at (15, 15) of length 35....."
);

console.putsx
(
    16,
    15,
    win.bgnd_White | win.fgnd_Red,
    40,
    "1234567890123456789012345678901234567890"
);

// Since the following is a stdout call, the text
// will use the current output attribute, which
// is the red/black attributes set at the beginning
// of this program.

console.gotoxy( 23, 0 );
stdout.put( "Press enter to continue:" );
stdin.ReadLn();

```

```
// Note: set the attributes back to black and white when
// the program exits.

console.setOutputAttr( win.fgnd_White | win.bgnd_Black );
console.cls();

end testPutsx;
```

---

**Program 3.7** Demonstration of console.puts and console.putsx

---

---

### 2.11.7 Other Console Module Routines

The sample programs in this chapter have really only touched on the capabilities of the HLA Standard Library Console Module. In addition to the routines this section demonstrates, the HLA Standard Library provides procedures to scroll the window, to resize the window, to read characters off the screen, to clear selected portions of the screen, to grab and restore data on the screen, and so forth. Space limitations preclude the further demonstration of the console module in this text. However, if you are interested you should read the HLA Standard Library documentation to learn more about the console module.

---

## 2.12 Putting It All Together

This chapter discussed the 80x86 address modes and other related topics. It began by discussing the 80x86's register, displacement-only (direct), register indirect, and indexed addressing modes. A good knowledge of these addressing modes and their uses is essential if you want to write good assembly language programs. Although this chapter does not delve deeply into the use of each of these addressing modes, it does present their syntax and a few simple examples of each (later chapters will expand on how you use each of these addressing modes).

After discussing addressing modes, this chapter described how HLA and Windows organize your code and data in memory. At this point this chapter also discussed the HLA `STATIC`, `DATA`, `READONLY`, `STORAGE`, and `VAR` data declaration sections. The alignment of data in memory can affect the performance of your programs; therefore, when discussing this topic, this chapter also described how to properly align objects in memory to obtain the fastest executing code.

One special section of memory is the 80x86 stack. In addition to briefly discussing the stack, this chapter also described how to use the stack to save temporary values using the `PUSH` and `POP` instructions (and several variations on these instructions).

To a running program, a variable is really nothing more than a simple address in memory. In an HLA source file, however, you may specify the address and type of an object in memory using powerful address expressions and type coercion operators. These chapter discusses the syntax for these expressions and operators and gives several examples of why you would want to use them.

This chapter concludes by discussing two modules in the HLA Standard Library: the dynamic memory allocation routines (*malloc* and *free*) and the console module. The console module is interesting because it lets you write more interesting programs by varying the text display.

Logic circuits are the basis for modern digital computer systems. To appreciate how computer systems operate you will need to understand digital logic and boolean algebra.

This Chapter provides only a basic introduction to boolean algebra. That subject alone is often the subject of an entire textbook. This Chapter will concentrate on those subjects that support other chapters in this text.

### 3.1 Chapter Overview

Boolean logic forms the basis for computation in modern binary computer systems. You can represent any algorithm, or any electronic computer circuit, using a system of boolean equations. This chapter provides a brief introduction to boolean algebra, truth tables, canonical representation, of boolean functions, boolean function simplification, logic design, and combinatorial and sequential circuits.

This material is especially important to those who want to design electronic circuits or write software that controls electronic circuits. Even if you never plan to design hardware or write software than controls hardware, the introduction to boolean algebra this chapter provides is still important since you can use such knowledge to optimize certain complex conditional expressions within IF, WHILE, and other conditional statements.

The section on minimizing (optimizing) logic functions uses *Veitch Diagrams* or *Karnaugh Maps*. The optimizing techniques this chapter uses reduce the number of *terms* in a boolean function. You should realize that many people consider this optimization technique obsolete because reducing the number of terms in an equation is not as important as it once was. This chapter uses the mapping method as an example of boolean function optimization, not as a technique one would regularly employ. If you are interested in circuit design and optimization, you will need to consult a text on logic design for better techniques.

### 3.2 Boolean Algebra

Boolean algebra is a deductive mathematical system closed over the values zero and one (false and true). A *binary operator* “ $\circ$ ” defined over this set of values accepts a pair of boolean inputs and produces a single boolean value. For example, the boolean AND operator accepts two boolean inputs and produces a single boolean output (the logical AND of the two inputs).

For any given algebra system, there are some initial assumptions, or *postulates*, that the system follows. You can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems often employ the following postulates:

- *Closure*. The boolean system is *closed* with respect to a binary operator if for every pair of boolean values, it produces a boolean result. For example, logical AND is closed in the boolean system because it accepts only boolean operands and produces only boolean results.
- *Commutativity*. A binary operator “ $\circ$ ” is said to be commutative if  $A \circ B = B \circ A$  for all possible boolean values  $A$  and  $B$ .
- *Associativity*. A binary operator “ $\circ$ ” is said to be associative if

$$(A \circ B) \circ C = A \circ (B \circ C)$$

for all boolean values  $A$ ,  $B$ , and  $C$ .

- *Distribution*. Two binary operators “ $\circ$ ” and “ $\%$ ” are distributive if

$$A \circ (B \% C) = (A \circ B) \% (A \circ C)$$

for all boolean values  $A$ ,  $B$ , and  $C$ .

- *Identity.* A boolean value  $I$  is said to be the *identity element* with respect to some binary operator “ $\circ$ ” if  $A \circ I = A$ .
- *Inverse.* A boolean value  $I$  is said to be the *inverse element* with respect to some binary operator “ $\circ$ ” if  $A \circ I = B$  and  $B \neq A$  (i.e.,  $B$  is the opposite value of  $A$  in a boolean system).

For our purposes, we will base boolean algebra on the following set of operators and values:

The two possible values in the boolean system are zero and one. Often we will call these values false and true (respectively).

The symbol “ $\bullet$ ” represents the logical AND operation; e.g.,  $A \bullet B$  is the result of logically ANDing the boolean values  $A$  and  $B$ . When using single letter variable names, this text will drop the “ $\bullet$ ” symbol; Therefore,  $AB$  also represents the logical AND of the variables  $A$  and  $B$  (we will also call this the product of  $A$  and  $B$ ).

The symbol “ $+$ ” represents the logical OR operation; e.g.,  $A + B$  is the result of logically ORing the boolean values  $A$  and  $B$ . (We will also call this the sum of  $A$  and  $B$ .)

Logical complement, negation, or not, is a unary operator. This text will use the (') symbol to denote logical negation. For example,  $A'$  denotes the logical NOT of  $A$ .

If several different operators appear in a single boolean expression, the result of the expression depends on the *precedence* of the operators. We'll use the following precedences (from highest to lowest) for the boolean operators: parenthesis, logical NOT, logical AND, then logical OR. The logical AND and OR operators are *left associative*. If two operators with the same precedence are adjacent, you must evaluate them from left to right. The logical NOT operation is right associative, although it would produce the same result using left or right associativity since it is a unary operator.

We will also use the following set of postulates:

- P1 Boolean algebra is closed under the AND, OR, and NOT operations.
- P2 The identity element with respect to  $\bullet$  is one and  $+$  is zero. There is no identity element with respect to logical NOT.
- P3 The  $\bullet$  and  $+$  operators are commutative.
- P4  $\bullet$  and  $+$  are distributive with respect to one another. That is,  $A \bullet (B + C) = (A \bullet B) + (A \bullet C)$  and  $A + (B \bullet C) = (A + B) \bullet (A + C)$ .
- P5 For every value  $A$  there exists a value  $A'$  such that  $A \bullet A' = 0$  and  $A + A' = 1$ . This value is the logical complement (or NOT) of  $A$ .
- P6  $\bullet$  and  $+$  are both associative. That is,  $(A \bullet B) \bullet C = A \bullet (B \bullet C)$  and  $(A + B) + C = A + (B + C)$ .

You can prove all other theorems in boolean algebra using these postulates. This text will not go into the formal proofs of these theorems, however, it is a good idea to familiarize yourself with some important theorems in boolean algebra. A sampling includes:

- Th1:  $A + A = A$
- Th2:  $A \bullet A = A$
- Th3:  $A + 0 = A$
- Th4:  $A \bullet 1 = A$
- Th5:  $A \bullet 0 = 0$
- Th6:  $A + 1 = 1$
- Th7:  $(A + B)' = A' \bullet B'$
- Th8:  $(A \bullet B)' = A' + B'$
- Th9:  $A + A \bullet B = A$
- Th10:  $A \bullet (A + B) = A$
- Th11:  $A + A'B = A + B$

$$\text{Th12: } A' \cdot (A + B') = A'B'$$

$$\text{Th13: } AB + AB' = A$$

$$\text{Th14: } (A' + B') \cdot (A' + B) = A'$$

$$\text{Th15: } A + A' = 1$$

$$\text{Th16: } A \cdot A' = 0$$

Theorems seven and eight above are known as *DeMorgan's Theorems* after the mathematician who discovered them.

The theorems above appear in pairs. Each pair (e.g., Th1 & Th2, Th3 & Th4, etc.) form a *dual*. An important principle in the boolean algebra system is that of *duality*. Any valid expression you can create using the postulates and theorems of boolean algebra remains valid if you interchange the operators and constants appearing in the expression. Specifically, if you exchange the  $\cdot$  and  $+$  operators and swap the 0 and 1 values in an expression, you will wind up with an expression that obeys all the rules of boolean algebra. *This does not mean the dual expression computes the same values*, it only means that both expressions are legal in the boolean algebra system. Therefore, this is an easy way to generate a second theorem for any fact you prove in the boolean algebra system.

Although we will not be proving any theorems for the sake of boolean algebra in this text, we will use these theorems to show that two boolean equations are identical. This is an important operation when attempting to produce *canonical representations* of a boolean expression or when simplifying a boolean expression.

### 3.3 Boolean Functions and Truth Tables

A boolean *expression* is a sequence of zeros, ones, and *literals* separated by boolean operators. A literal is a primed (negated) or unprimed variable name. For our purposes, all variable names will be a single alphabetic character. A boolean function is a specific boolean expression; we will generally give boolean functions the name  $F$  with a possible subscript. For example, consider the following boolean:

$$F_0 = AB + C$$

This function computes the logical AND of  $A$  and  $B$  and then logically ORs this result with  $C$ . If  $A=1$ ,  $B=0$ , and  $C=1$ , then  $F_0$  returns the value one ( $1 \cdot 0 + 1 = 1$ ).

Another way to represent a boolean function is via a *truth table*. A previous chapter (see “Logical Operations on Bits” on page 55) used truth tables to represent the AND and OR functions. Those truth tables took the forms:

**Table 14: AND Truth Table**

AND	0	1
0	0	0
1	0	1

**Table 15: OR Truth Table**

OR	0	1
0	0	1
1	1	1

For binary operators and two input variables, this form of a truth table is very natural and convenient. However, reconsider the boolean function  $F_0$  above. That function has *three* input variables, not two. Therefore, one cannot use the truth table format given above. Fortunately, it is still very easy to construct truth tables for three or more variables. The following example shows one way to do this for functions of three or four variables:

**Table 16: Truth Table for a Function with Three Variables**

$F = AB + C$		BA			
		00	01	10	11
C	0	0	0	0	1
	1	1	1	1	1

**Table 17: Truth Table for a Function with Four Variables**

$F = AB + CD$		BA			
		00	01	10	11
DC	00	0	0	0	1
	01	0	0	0	1
	10	0	0	0	1
	11	1	1	1	1

In the truth tables above, the four columns represent the four possible combinations of zeros and ones for  $A$  &  $B$  ( $B$  is the H.O. or leftmost bit,  $A$  is the L.O. or rightmost bit). Likewise the four rows in the second truth table above represent the four possible combinations of zeros and ones for the  $C$  and  $D$  variables. As before,  $D$  is the H.O. bit and  $C$  is the L.O. bit.

Table 18 shows another way to represent truth tables. This form has two advantages over the forms above – it is easier to fill in the table and it provides a compact representation for two or more functions.

**Table 18: Another Format for Truth Tables**

C	B	A	F = ABC	F = AB + C	F = A+BC
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

Note that the truth table above provides the values for three separate functions of three variables.

Although you can create an infinite variety of boolean functions, they are not all unique. For example,  $F=A$  and  $F=AA$  are two different functions. By theorem two, however, it is easy to show that these two functions are equivalent, that is, they produce exactly the same outputs for all input combinations. If you fix the number of input variables, there are a finite number of unique boolean functions possible. For example, there are only 16 unique boolean functions with two inputs and there are only 256 possible boolean functions of three input variables. Given  $n$  input variables, there are  $2^{2^n}$  (two raised to the two raised to the  $n^{\text{th}}$  power)<sup>1</sup> unique boolean functions of those  $n$  input values. For two input variables,  $2^{2^2} = 2^4$  or 16 different functions. With three input variables there are  $2^{2^3} = 2^8$  or 256 possible functions. Four input variables create  $2^{2^4}$  or  $2^{16}$ , or 65,536 different unique boolean functions.

When dealing with only 16 boolean functions, it's easy enough to name each function. The following table lists the 16 possible boolean functions of two input variables along with some common names for those functions:

**Table 19: The 16 Possible Boolean Functions of Two Variables**

Function #	Description
0	Zero or Clear. Always returns zero regardless of A and B input values.
1	Logical NOR ( $\text{NOT } (A \text{ OR } B)) = (A+B)'$
2	Inhibition = $AB'$ (A, not B). Also equivalent to $A > B$ or $B < A$ .
3	NOT B. Ignores A and returns B'.
4	Inhibition = $BA'$ (B, not A). Also equivalent to $B > A$ or $A < B$ .
5	NOT A. Returns A' and ignores B
6	Exclusive-or (XOR) = $A \oplus B$ . Also equivalent to $A \neq B$ .
7	Logical NAND ( $\text{NOT } (A \text{ AND } B)) = (A \cdot B)'$
8	Logical AND = $A \cdot B$ . Returns A AND B.

1. In this context, the operator “\*\*” means exponentiation.

Table 19: The 16 Possible Boolean Functions of Two Variables

Function #	Description
9	Equivalence = (A = B). Also known as exclusive-NOR (not exclusive-or).
10	Copy A. Returns the value of A and ignores B's value.
11	Implication, B implies A, or A + B'. (if B then A). Also equivalent to B >= A.
12	Copy B. Returns the value of B and ignores A's value.
13	Implication, A implies B, or B + A' (if A then B). Also equivalent to A >= B.
14	Logical OR = A+B. Returns A OR B.
15	One or Set. Always returns one regardless of A and B input values.

Beyond two input variables there are too many functions to provide specific names. Therefore, we will refer to the function's number rather than the function's name. For example,  $F_8$  denotes the logical AND of  $A$  and  $B$  for a two-input function and  $F_{14}$  is the logical OR operation. Of course, the only problem is to determine a function's number. For example, given the function of three variables  $F=AB+C$ , what is the corresponding function number? This number is easy to compute by looking at the truth table for the function (see Table 22 on page 203). If we treat the values for  $A$ ,  $B$ , and  $C$  as bits in a binary number with  $C$  being the H.O. bit and  $A$  being the L.O. bit, they produce the binary numbers in the range zero through seven. Associated with each of these binary strings is a zero or one function result. If we construct a binary value by placing the function result in the bit position specified by  $A$ ,  $B$ , and  $C$ , the resulting binary number is that function's number. Consider the truth table for  $F=AB+C$ :

CBA:	7	6	5	4	3	2	1	0
F=AB+C :	1	1	1	1	1	0	0	0

If we treat the function values for  $F$  as a binary number, this produces the value  $F8_{16}$  or  $248_{10}$ . We will usually denote function numbers in decimal.

This also provides the insight into why there are  $2^{2^n}$  different functions of  $n$  variables: if you have  $n$  input variables, there are  $2^n$  bits in function's number. If you have  $m$  bits, there are  $2^m$  different values. Therefore, for  $n$  input variables there are  $m=2^n$  possible bits and  $2^m$  or  $2^{2^n}$  possible functions.

3.4 Algebraic Manipulation of Boolean Expressions

You can transform one boolean expression into an equivalent expression by applying the postulates the theorems of boolean algebra. This is important if you want to convert a given expression to a *canonical form* (a standardized form) or if you want to minimize the number of literals (primed or unprimed variables) or terms in an expression. Minimizing terms and expressions can be important because electrical circuits often consist of individual components that implement each term or literal for a given expression. Minimizing the expression allows the designer to use fewer electrical components and, therefore, can reduce the cost of the system.

Unfortunately, there are no fixed rules you can apply to optimize a given expression. Much like constructing mathematical proofs, an individual's ability to easily do these transformations is usually a function of experience. Nevertheless, a few examples can show the possibilities:

ab + ab' + a'b

= a(b+b') + a'b

= a•1 + a'b

= a + a'b

= a + b

By P4

By P5

By Th4

By Th11

$$\begin{aligned}
(a'b + a'b' + b')' &= (a'(b+b') + b')' && \text{By P4} \\
&= (a' \bullet 1 + b')' && \text{By P5} \\
&= (a' + b') && \text{By Th4} \\
&= ((ab)')' && \text{By Th8} \\
&= ab && \text{By definition of not}
\end{aligned}$$

$$\begin{aligned}
b(a+c) + ab' + bc' + c &= ba + bc + ab' + bc' + c && \text{By P4} \\
&= a(b+b') + b(c + c') + c && \text{By P4} \\
&= a \bullet 1 + b \bullet 1 + c && \text{By P5} \\
&= a + b + c && \text{By Th4}
\end{aligned}$$

Although these examples all use algebraic transformations to simplify a boolean expression, we can also use algebraic operations for other purposes. For example, the next section describes a canonical form for boolean expressions. We can use algebraic manipulation to produce canonical forms even though the canonical forms are rarely optimal.

### 3.5 Canonical Forms

Since there are a finite number of boolean functions of  $n$  input variables, yet an infinite number of possible logic expressions you can construct with those  $n$  input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given the same inputs). To help eliminate possible confusion, logic designers generally specify a boolean function using a *canonical*, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called *sum of minterms* and the second is the *product of maxterms*. Using the duality principle, it is very easy to convert between these two.

A *term* is a variable or a product (logical AND) of several different literals. For example, if you have two variables,  $A$  and  $B$ , there are eight possible terms:  $A$ ,  $B$ ,  $A'$ ,  $B'$ ,  $A'B'$ ,  $A'B$ ,  $AB'$ , and  $AB$ . For three variables we have 26 different terms:  $A$ ,  $B$ ,  $C$ ,  $A'$ ,  $B'$ ,  $C'$ ,  $A'B'$ ,  $A'B$ ,  $AB'$ ,  $AB$ ,  $A'C'$ ,  $A'C$ ,  $AC'$ ,  $AC$ ,  $B'C'$ ,  $B'C$ ,  $BC'$ ,  $BC$ ,  $A'B'C'$ ,  $AB'C'$ ,  $A'BC'$ ,  $ABC'$ ,  $A'B'BC$ ,  $AB'BC$ ,  $A'BC$ , and  $ABC$ . As you can see, as the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing exactly  $n$  literals. For example, the minterms for two variables are  $A'B'$ ,  $AB'$ ,  $A'B$ , and  $AB$ . Likewise, the minterms for three variables  $A$ ,  $B$ , and  $C$  are  $A'B'C'$ ,  $AB'C'$ ,  $A'BC'$ ,  $ABC'$ ,  $A'B'BC$ ,  $AB'BC$ ,  $A'BC$ , and  $ABC$ . In general, there are  $2^n$  minterms for  $n$  variables. The set of possible minterms is very easy to generate since they correspond to the sequence of binary numbers:

**Table 20: Minterms for Three Input Variables**

Binary Equivalent (CBA)	Minterm
000	$A'B'C'$
001	$AB'C'$
010	$A'BC'$
011	$ABC'$
100	$A'B'C$
101	$AB'C$
110	$A'BC$
111	$ABC$

We can specify *any* boolean function using a sum (logical OR) of minterms. Given  $F_{248}=AB+C$  the equivalent canonical form is  $ABC+A'BC+AB'C+A'B'C+ABC'$ . Algebraically, we can show that these two are equivalent as follows:

$$\begin{aligned}
 ABC+A'BC+AB'C+A'B'C+ABC' &= BC(A+A') + B'C(A+A') + ABC' && \text{By P4} \\
 &= BC \bullet 1 + B'C \bullet 1 + ABC' && \text{By Th15} \\
 &= C(B+B') + ABC' && \text{By P4} \\
 &= C + ABC' && \text{By Th15 \& Th4} \\
 &= C + AB && \text{By Th11}
 \end{aligned}$$

Obviously, the canonical form is not the optimal form. On the other hand, there is a big advantage to the sum of minterms canonical form: it is very easy to generate the truth table for a function from this canonical form. Furthermore, it is also very easy to generate the logic equation from the truth table.

To build the truth table from the canonical form, simply convert each minterm into a binary value by substituting a “1” for unprimed variables and a “0” for primed variables. Then place a “1” in the corresponding position (specified by the binary minterm value) in the truth table:

1) Convert minterms to binary equivalents:

$$\begin{aligned}
 F_{248} &= CBA + CBA' + CB'A + CB'A' + C'BA \\
 &= 111 + 110 + 101 + 100 + 011
 \end{aligned}$$

2) Substitute a one in the truth table for each entry above:

**Table 21: Creating a Truth Table from Minterms, Step One**

C	B	A	F = AB+C
0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Finally, put zeros in all the entries that you did not fill with ones in the first step above:

**Table 22: Creating a Truth Table from Minterms, Step Two**

C	B	A	F = AB+C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Going in the other direction, generating a logic function from a truth table, is almost as easy. First, locate all the entries in the truth table with a one. In the table above, these are the last five entries. The number of table entries containing ones determines the number of minterms in the canonical equation. To generate the individual minterms, substitute  $A$ ,  $B$ , or  $C$  for ones and  $A'$ ,  $B'$ , or  $C'$  for zeros in the truth table above. Then compute the sum of these items. In the example above,  $F_{248}$  contains one for  $CBA = 111$ ,  $110$ ,  $101$ ,  $100$ , and  $011$ . Therefore,  $F_{248} = CBA + CBA' + CB'A + CB'A' + C'AB$ . The first term,  $CBA$ , comes from the last entry in the table above.  $C$ ,  $B$ , and  $A$  all contain ones so we generate the minterm  $CBA$  (or  $ABC$ , if you prefer). The second to last entry contains  $110$  for  $CBA$ , so we generate the minterm  $CBA'$ . Likewise,  $101$  produces  $CB'A$ ;  $100$  produces  $CB'A'$ , and  $011$  produces  $C'BA$ . Of course, the logical OR and logical AND operations are both commutative, so we can rearrange the terms within the minterms as we please and we can rearrange the minterms within the sum as we see fit. This process works equally well for any number of variables. Consider the function  $F_{53504} = ABCD + A'BCD + A'B'CD + A'B'C'D$ . Placing ones in the appropriate positions in the truth table generates the following:

**Table 23: Creating a Truth Table with Four Variables from Minterms**

D	C	B	A	$F = ABCD + A'B'CD + A'B'CD + A'B'C'D$
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	1
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	1

The remaining elements in this truth table all contain zero.

Perhaps the easiest way to generate the canonical form of a boolean function is to first generate the truth table for that function and then build the canonical form from the truth table. We'll use this technique, for example, when converting between the two canonical forms this chapter presents. However, it is also a simple matter to generate the sum of minterms form algebraically. By using the distributive law and theorem 15 ( $A + A' = 1$ ) makes this task easy. Consider  $F_{248} = AB + C$ . This function contains two terms,  $AB$  and  $C$ , but they are not minterms. Minterms contain each of the possible variables in a primed or unprimed form. We can convert the first term to a sum of minterms as follows:

$$\begin{aligned}
 AB &= AB \bullet 1 && \text{By Th4} \\
 &= AB \bullet (C + C') && \text{By Th 15} \\
 &= ABC + ABC' && \text{By distributive law} \\
 &= CBA + C'BA && \text{By associative law}
 \end{aligned}$$

Similarly, we can convert the second term in  $F_{248}$  to a sum of minterms as follows:

$$\begin{aligned}
 C &= C \bullet 1 && \text{By Th4} \\
 &= C \bullet (A + A') && \text{By Th15} \\
 &= CA + CA' && \text{By distributive law} \\
 &= CA \bullet 1 + CA' \bullet 1 && \text{By Th4} \\
 &= CA \bullet (B + B') + CA' \bullet (B + B') && \text{By Th15} \\
 &= CAB + CAB' + CA'B + CA'B' && \text{By distributive law} \\
 &= CBA + CBA' + CB'A + CB'A' && \text{By associative law}
 \end{aligned}$$

The last step (rearranging the terms) in these two conversions is optional. To obtain the final canonical form for  $F_{248}$  we need only sum the results from these two conversions:

$$\begin{aligned} F_{248} &= (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A') \\ &= CBA + CBA' + CB'A + CB'A' + C'BA \end{aligned}$$

Another way to generate a canonical form is to use *products of maxterms*. A maxterm is the sum (logical OR) of all input variables, primed or unprimed. For example, consider the following logic function  $G$  of three variables:

$$G = (A+B+C) \cdot (A'+B+C) \cdot (A+B'+C).$$

Like the sum of minterms form, there is exactly one product of maxterms for each possible logic function. Of course, for every product of maxterms there is an equivalent sum of minterms form. In fact, the function  $G$ , above, is equivalent to

$$F_{248} = CBA + CBA' + CB'A + CB'A' + C'BA = AB + C.$$

Generating a truth table from the product of maxterms is no more difficult than building it from the sum of minterms. You use the duality principle to accomplish this. Remember, the duality principle says to swap AND for OR and zeros for ones (and vice versa). Therefore, to build the truth table, you would first swap primed and non-primed literals. In  $G$  above, this would yield:

$$G = (A' + B' + C') \cdot (A + B' + C') \cdot (A' + B + C')$$

The next step is to swap the logical OR and logical AND operators. This produces

$$G = A'B'C' + AB'C' + A'BC'$$

Finally, you need to swap all zeros and ones. This means that you store *zeros* into the truth table for each of the above entries and then fill in the rest of the truth table with ones. This will place a zero in entries zero, one, and two in the truth table. Filling the remaining entries with ones produces  $F_{248}$ .

You can easily convert between these two canonical forms by generating the truth table for one form and working backwards from the truth table to produce the other form. For example, consider the function of two variables,  $F_7 = A + B$ . The sum of minterms form is  $F_7 = A'B + AB' + AB$ . The truth table takes the form:

**Table 24:  $F_7$  (OR) Truth Table for Two Variables**

$F_7$	A	B
0	0	0
0	1	0
1	0	1
1	1	1

Working backwards to get the product of maxterms, we locate all entries that have a zero result. This is the entry with  $A$  and  $B$  equal to zero. This gives us the first step of  $G=A'B'$ . However, we still need to invert all the variables to obtain  $G=AB$ . By the duality principle we need to swap the logical OR and logical AND operators obtaining  $G=A+B$ . This is the canonical *product of maxterms* form.

Since working with the product of maxterms is a little messier than working with sums of minterms, this text will generally use the sum of minterms form. Furthermore, the sum of minterms form is more common in boolean logic work. However, you will encounter both forms when studying logic design.

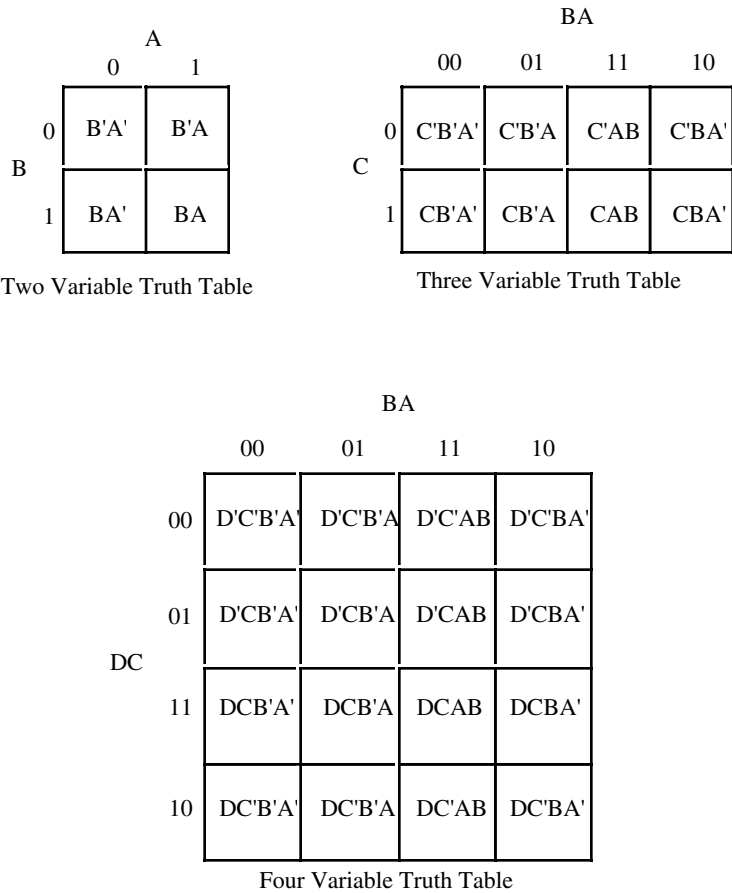
### 3.6 Simplification of Boolean Functions

Since there are an infinite variety of boolean functions of  $n$  variables, but only a finite number of unique boolean functions of those  $n$  variables, you might wonder if there is some method that will simplify a given boolean function to produce the optimal form. Of course, you can always use algebraic transformations to produce the optimal form, but using heuristics does not guarantee an optimal transformation. There are, however, two methods that *will* reduce a given boolean function to its optimal form: the map method and the prime implicants method. In this text we will only cover the mapping method, see any text on logic design for other methods.

Since for any logic function some optimal form must exist, you may wonder why we don't use the optimal form for the canonical form. There are two reasons. First, there may be several optimal forms. They are not guaranteed to be unique. Second, it is easy to convert between the canonical and truth table forms.

Using the map method to optimize boolean functions is practical only for functions of two, three, or four variables. With care, you can use it for functions of five or six variables, but the map method is cumbersome to use at that point. For more than six variables, attempting map simplifications by hand would not be wise<sup>2</sup>.

The first step in using the map method is to build a two-dimensional truth table for the function (see Figure 3.1)



BA

0001011110

00	D'C'B'A'	D'C'B'A	D'C'AB	D'C'BA'
01	D'CB'A'	D'CB'A	D'CAB	D'CBA'
11	DCB'A'	DCB'A	DCAB	DCBA'
10	DC'B'A'	DC'B'A	DC'AB	DC'BA'

DC

Four Variable Truth Table

Figure 3.1 Two, Three, and Four Dimensional Truth Tables

2. However, it's probably quite reasonable to write a *program* that uses the map method for seven or more variables.

**Warning:** Take a careful look at these truth tables. They do not use the same forms appearing earlier in this chapter. In particular, the progression of the values is 00, 01, 11, 10, not 00, 01, 10, 11. This is very important! If you organize the truth tables in a binary sequence, the mapping optimization method will not work properly. We will call this a *truth map* to distinguish it from the standard truth table.

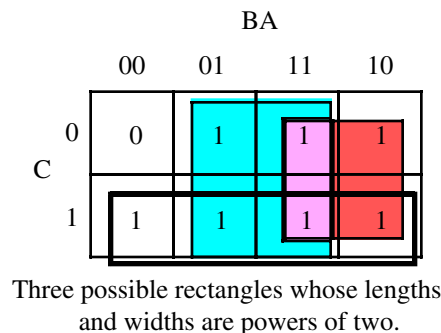
Assuming your boolean function is in canonical form (sum of minterms), insert ones for each of the truth map entries corresponding to a minterm in the function. Place zeros everywhere else. For example, consider the function of three variables  $F = C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA$ . Figure 3.2 shows the truth map for this function.

		BA			
		00	01	11	10
C	0	0	1	1	1
	1	1	1	1	1

$$F = C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA.$$

**Figure 3.2 A Simple Truth Map**

The next step is to draw rectangles around rectangular groups of ones. The rectangles you enclose must have sides whose lengths are powers of two. For functions of three variables, the rectangles can have sides whose lengths are one, two, and four. The set of rectangles you draw must surround all cells containing ones in the truth map. The trick is to draw all possible rectangles unless a rectangle would be completely enclosed within another. Note that the rectangles may overlap if one does not enclose the other. In the truth map in Figure 3.3 there are three such rectangles (see Figure 3.3)



**Figure 3.3 Surrounding Rectangular Groups of Ones in a Truth Map**

Each rectangle represents a term in the simplified boolean function. Therefore, the simplified boolean function will contain only three terms. You build each term using the process of elimination. You eliminate any variables whose primed and unprimed form both appear within the rectangle. Consider the long skinny rectangle above that is sitting in the row where  $C=1$ . This rectangle contains both  $A$  and  $B$  in primed and unprimed form. Therefore, we can eliminate  $A$  and  $B$  from the term. Since the rectangle sits in the  $C=1$  region, this rectangle represents the single literal  $C$ .

Now consider the blue square above. This rectangle includes  $C$ ,  $C'$ ,  $B$ ,  $B'$  and  $A$ . Therefore, it represents the single term  $A$ . Likewise, the red square above contains  $C$ ,  $C'$ ,  $A$ ,  $A'$  and  $B$ . Therefore, it represents the single term  $B$ .

The final, optimal, function is the sum (logical OR) of the terms represented by the three squares. Therefore,  $F = A + B + C$ . You do not have to consider the remaining squares containing zeros.

When enclosing groups of ones in the truth map, you must consider the fact that a truth map forms a *torus* (i.e., a doughnut shape). The right edge of the map *wraps around* to the left edge (and vice-versa). Likewise, the top edge *wraps around* to the bottom edge. This introduces additional possibilities when surrounding groups of ones in a map. Consider the boolean function  $F = C'B'A' + C'BA' + CB'A' + CBA'$ . Figure 3.4 shows the truth map for this function.

		BA			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

$F = C'B'A' + C'BA' + CB'A' + CBA'$

**Figure 3.4 Truth Map for  $F = C'B'A' + C'BA' + CB'A' + CBA'$**

At first glance, you would think that there are two possible rectangles here as Figure 3.5 shows.

		BA			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

**Figure 3.5 First Attempt at Surrounding Rectangles Formed by Ones**

However, because the truth map is a continuous object with the right side and left sides connected, we can form a single, square rectangle, as Figure 3.6 shows.

		BA			
		00	01	11	10
C	0	1	0	0	1
	1	1	0	0	1

**Figure 3.6 Correct Rectangle for the Function**

So what? Why do we care if we have one rectangle or two in the truth map? The answer is because the larger the rectangles are, the more terms they will eliminate. The fewer rectangles that we have, the fewer terms will appear in the final boolean function. For example, the former example with two rectangles generates a function with two terms. The first rectangle (on the left) eliminates the  $C$  variable, leaving  $A'B'$  as its term. The second rectangle, on the right, also eliminates the  $C$  variable, leaving the term  $BA'$ . Therefore, this truth map would produce the equation  $F = A'B' + A'B$ . We know this is not optimal, see Th 13. Now consider the second truth map above. Here we have a single rectangle so our boolean function will only have a single term. Obviously this is more optimal than an equation with two terms. Since this rectangle includes both  $C$  and  $C'$  and also  $B$  and  $B'$ , the only term left is  $A'$ . This boolean function, therefore, reduces to  $F = A'$ .

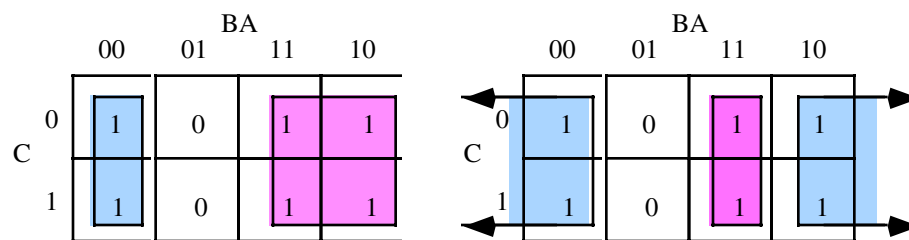
There are only two cases that the truth map method cannot handle properly: a truth map that contains all zeros or a truth map that contains all ones. These two cases correspond to the boolean functions  $F=0$  and  $F=1$  (that is, the function number is  $2^{n-1}$ ), respectively. These functions are easy to generate by inspection of the truth map.

An important thing you must keep in mind when optimizing boolean functions using the mapping method is that you always want to pick the largest rectangles whose sides' lengths are a power of two. You must do this even for overlapping rectangles (unless one rectangle encloses another). Consider the boolean function  $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$ . This produces the truth map appearing in Figure 3.7.

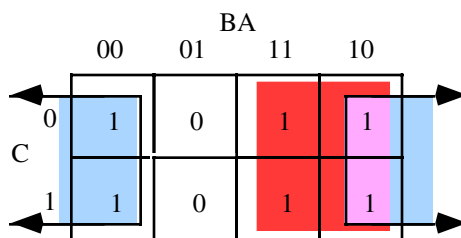
		BA			
		00	01	11	10
C	0	1	0	1	1
	1	1	0	1	1

**Figure 3.7 Truth Map for  $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$**

The initial temptation is to create one of the sets of rectangles found in Figure 3.8. However, the correct mapping appears in Figure 3.9



**Figure 3.8 Obvious Choices for Rectangles**



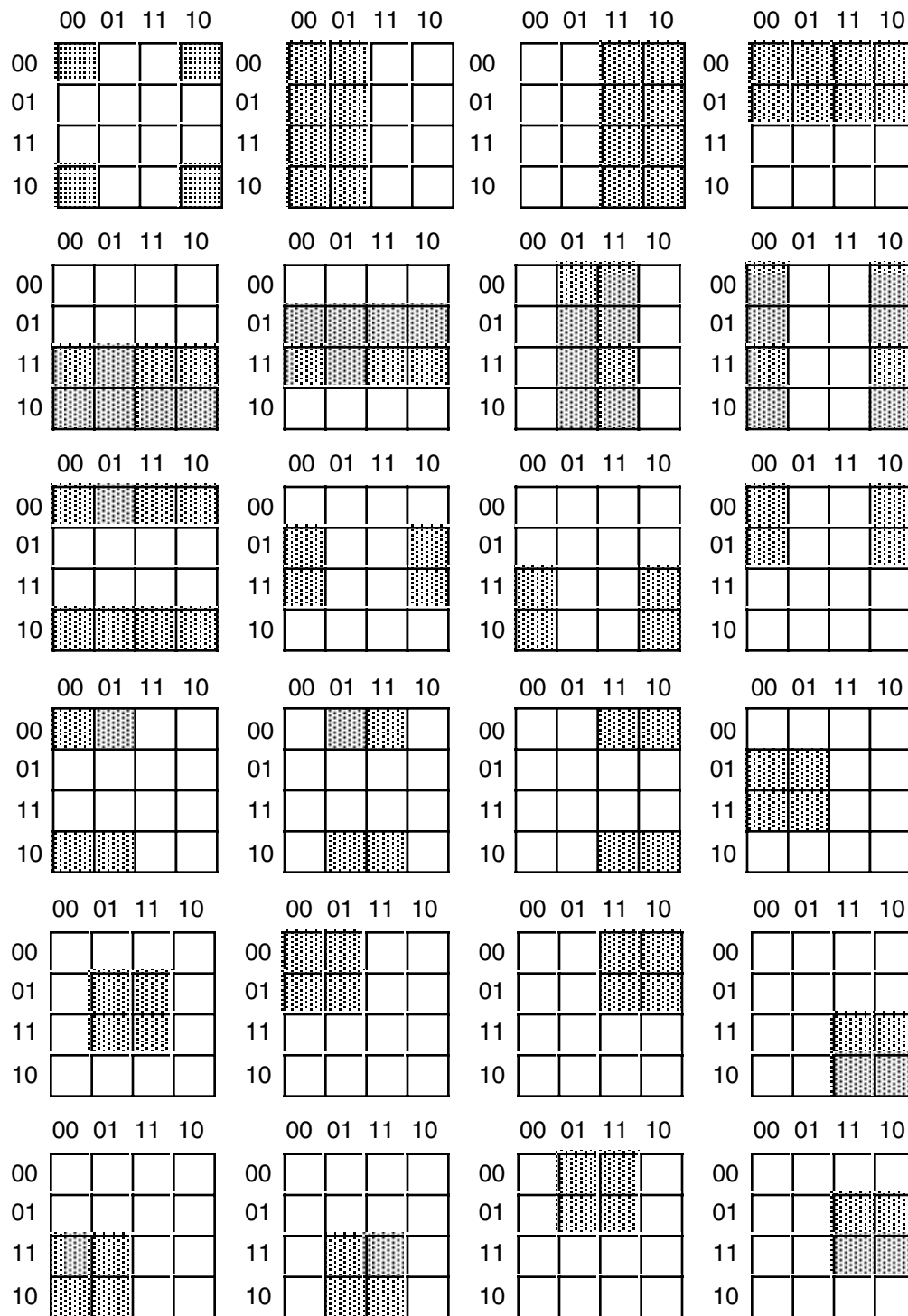
**Figure 3.9** Correct Set of Rectangles for  $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$

All three mappings will produce a boolean function with two terms. However, the first two will produce the expressions  $F = B + A'B'$  and  $F = AB + A'$ . The third form produces  $F = B + A'$ . Obviously, this last form is more optimal than the other two forms (see theorems 11 and 12).

For functions of three variables, the size of the rectangle determines the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have three literals (assuming we're working with functions of three variables).
- A rectangle surrounding two squares containing ones represents a term containing two literals.
- A rectangle surrounding four squares containing ones represents a term containing a single literal.
- A rectangle surrounding eight squares represents the function  $F = 1$ .

Truth maps you create for functions of four variables are even trickier. This is because there are lots of places rectangles can hide from you along the edges. Figure 3.10 shows some possible places rectangles can hide.



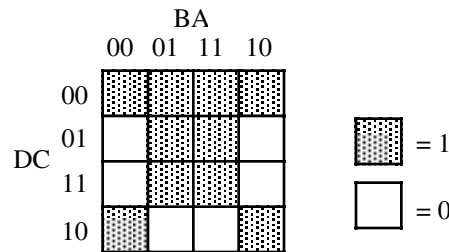
**Figure 3.10** Partial Pattern List for 4x4 Truth Map

This list of patterns doesn't even begin to cover all of them! For example, these diagrams show none of the 1x2 rectangles. You must exercise care when working with four variable maps to ensure you select the largest possible rectangles, especially when overlap occurs. This is particularly important with you have a rectangle next to an edge of the truth map.

As with functions of three variables, the size of the rectangle in a four variable truth map controls the number of terms it represents:

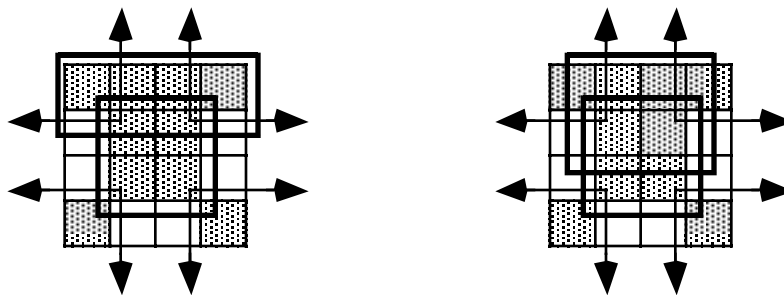
- A rectangle enclosing a single square represents a minterm. The associated term will have four literals.
- A rectangle surrounding two squares containing ones represents a term containing three literals.
- A rectangle surrounding four squares containing ones represents a term containing two literals.
- A rectangle surrounding eight squares containing ones represents a term containing a single literal.
- A rectangle surrounding sixteen squares represents the function  $F=1$ .

This last example demonstrates an optimization of a function containing four variables. The function is  $F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA'$ , the truth map appears in Figure 3.11.



**Figure 3.11 Truth Map for  $F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA'$**

Here are two possible sets of maximal rectangles for this function, each producing three terms (see Figure 3.12). Both functions are equivalent; both are as optimal as you can get<sup>3</sup>. Either will suffice for our purposes.



**Figure 3.12 Two Combinations of Surrounded Values Yielding Three Terms**

First, let's consider the term represented by the rectangle formed by the four corners. This rectangle contains  $B, B', D$ , and  $D'$ ; so we can eliminate those terms. The remaining terms contained within these rectangles are  $C'$  and  $A'$ , so this rectangle represents the term  $C'A'$ .

The second rectangle, common to both maps in Figure 3.12, is the rectangle formed by the middle four squares. This rectangle includes the terms  $A, B, B', C, D$ , and  $D'$ . Eliminating  $B, B', D$ , and  $D'$  (since both primed and unprimed terms exist), we obtain  $CA$  as the term for this rectangle.

3. Remember, there is no guarantee that there is a unique optimal solution.

The map on the left in Figure 3.12 has a third term represented by the top row. This term includes the variables  $A, A', B, B', C'$  and  $D'$ . Since it contains  $A, A', B$ , and  $B'$ , we can eliminate these terms. This leaves the term  $C'D'$ . Therefore, the function represented by the map on the left is  $F = C'A' + CA + C'D'$ .

The map on the right in Figure 3.12 has a third term represented by the top/middle four squares. This rectangle subsumes the variables  $A, B, B', C, C'$ , and  $D'$ . We can eliminate  $B, B', C$ , and  $C'$  since both primed and unprimed versions appear, this leaves the term  $AD$ . Therefore, the function represented by the function on the right is  $F = C'A' + CA + AD$ .

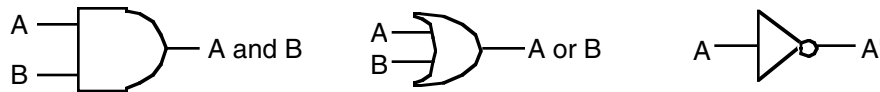
Since both expressions are equivalent, contain the same number of terms, and the same number of operators, either form is equivalent. Unless there is another reason for choosing one over the other, you can use either form.

## 3.7 What Does This Have To Do With Computers, Anyway?

Although there is a tenuous relationship between boolean functions and boolean expressions in programming languages like C or Pascal, it is fair to wonder why we're spending so much time on this material. However, the relationship between boolean logic and computer systems is much stronger than it first appears. There is a one-to-one relationship between boolean functions and electronic circuits. Electrical engineers who design CPUs and other computer related circuits need to be intimately familiar with this stuff. Even if you never intend to design your own electronic circuits, understanding this relationship is important if you want to make the most of any computer system.

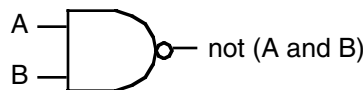
### 3.7.1 Correspondence Between Electronic Circuits and Boolean Functions

There is a one-to-one correspondence between an electrical circuits and boolean functions. For any boolean function you can design an electronic circuit and vice versa. Since boolean functions only require the AND, OR, and NOT boolean operators<sup>4</sup>, we can construct any electronic circuit using these operations exclusively. The boolean AND, OR, and NOT functions correspond to the following electronic circuits, the AND, OR, and inverter (NOT) gates (see Figure 3.13).



**Figure 3.13** AND, OR, and Inverter (NOT) Gates

One interesting fact is that you only need a single gate type to implement *any* electronic circuit. This gate is the NAND gate, shown in Figure 3.14.

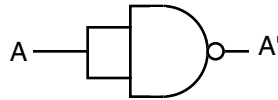


**Figure 3.14** The NAND Gate

To prove that we can construct any boolean function using only NAND gates, we need only show how to build an inverter (NOT), an AND gate, and an OR gate from a NAND (since we can create any boolean func-

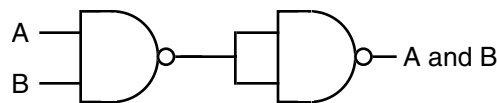
4. We know this is true because these are the only operators that appear within canonical forms.

tion using only AND, NOT, and OR). Building an inverter is easy, just connect the two inputs together (see Figure 3.15).



**Figure 3.15 Inverter Built from a NAND Gate**

Once we can build an inverter, building an AND gate is easy – just invert the output of a NAND gate. After all, NOT (NOT (A AND B)) is equivalent to A AND B (see Figure 3.16). Of course, this takes two NAND gates to construct a single AND gate, but no one said that circuits constructed only with NAND gates would be optimal, only that it is possible.

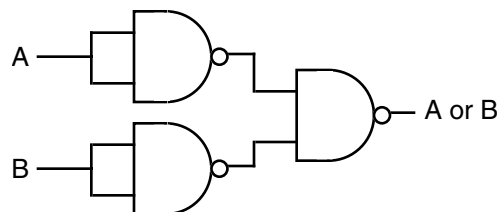


**Figure 3.16 Constructing an AND Gate From Two NAND Gates**

The remaining gate we need to synthesize is the logical-OR gate. We can easily construct an OR gate from NAND gates by applying DeMorgan's theorems.

$(A \text{ or } B)'$	$= A' \text{ and } B'$	DeMorgan's Theorem.
$A \text{ or } B$	$= (A' \text{ and } B')'$	Invert both sides of the equation.
$A \text{ or } B$	$= A' \text{ nand } B'$	Definition of NAND operation.

By applying these transformations, you get the circuit in Figure 3.17.



**Figure 3.17 Constructing an OR Gate from NAND Gates**

Now you might be wondering why we would even bother with this. After all, why not just use logical AND, OR, and inverter gates directly? There are two reasons for this. First, NAND gates are generally less expensive to build than other gates. Second, it is also much easier to build up complex integrated circuits from the same basic building blocks than it is to construct an integrated circuit using different basic gates.

Note, by the way, that it is possible to construct any logic circuit using only NOR gates<sup>5</sup>. The correspondence between NAND and NOR logic is orthogonal to the correspondence between the two canonical forms appearing in this chapter (sum of minterms vs. product of maxterms). While NOR logic is useful for many circuits, most electronic designs use NAND logic. See the exercises for more examples.

5. NOR is NOT (A OR B).

### 3.7.2 Combinatorial Circuits

A combinatorial circuit is a system containing basic boolean operations (AND, OR, NOT), some inputs, and a set of outputs. Since each output corresponds to an individual logic function, a combinatorial circuit often implements several different boolean functions. It is very important that you remember this fact – each output represents a different boolean function.

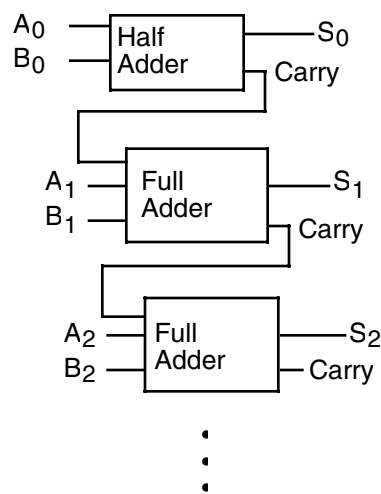
A computer's CPU is built up from various combinatorial circuits. For example, you can implement an addition circuit using boolean functions. Suppose you have two one-bit numbers,  $A$  and  $B$ . You can produce the one-bit sum and the one-bit carry of this addition using the two boolean functions:

$$\begin{array}{ll} S &= AB' + A'B && \text{Sum of } A \text{ and } B. \\ C &= AB && \text{Carry from addition of } A \text{ and } B. \end{array}$$

These two boolean functions implement a *half-adder*. Electrical engineers call it a half adder because it adds two bits together but cannot add in a carry from a previous operation. A *full adder* adds three one-bit inputs (two bits plus a carry from a previous addition) and produces two outputs: the sum and the carry. The two logic equations for a full adder are

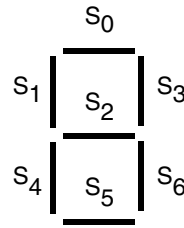
$$\begin{array}{ll} S &= A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in} \\ C_{out} &= AB + AC_{in} + BC_{in} \end{array}$$

Although these logic equations only produce a single bit result (ignoring the carry), it is easy to construct an  $n$ -bit sum by combining adder circuits (see Figure 3.18). So, as this example clearly illustrates, we can use logic functions to implement arithmetic and boolean operations.

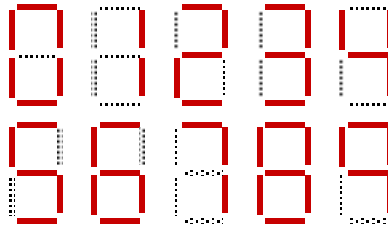


**Figure 3.18 Building an N-Bit Adder Using Half and Full Adders**

Another common combinatorial circuit is the *seven-segment decoder*. This is a combinatorial circuit that accepts four inputs and determines which of the segments on a seven-segment LED display should be on (logic one) or off (logic zero). Since a seven segment display contains seven output values (one for each segment), there will be seven logic functions associated with the display (segment zero through segment six). See Figure 3.19 for the segment assignments. Figure 3.20 shows the segment assignments for each of the ten decimal values.



**Figure 3.19** Seven Segment Display



**Figure 3.20** Seven Segment Values for “0” Through “9”

The four inputs to each of these seven boolean functions are the four bits from a binary number in the range 0..9. Let  $D$  be the H.O. bit of this number and  $A$  be the L.O. bit of this number. Each logic function should produce a one (segment on) for a given input if that particular segment should be illuminated. For example  $S_4$  (segment four) should be on for binary values 0000, 0010, 0110, and 1000. For each value that illuminates a segment, you will have one minterm in the logic equation:

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'.$$

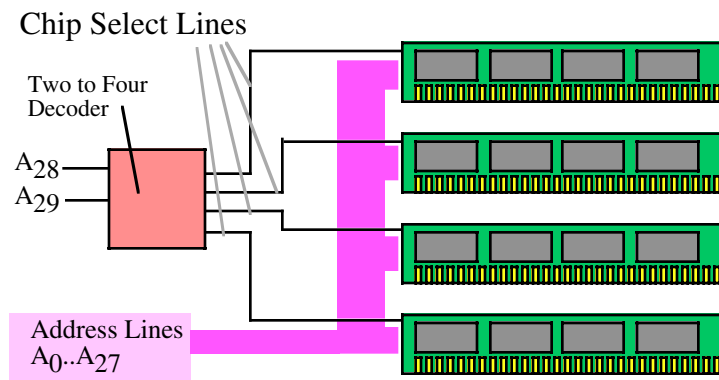
$S_0$ , as a second example, is on for values zero, two, three, five, six, seven, eight, and nine. Therefore, the logic function for  $S_0$  is

$$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A$$

You can generate the other five logic functions in a similar fashion (see the exercises).

Decoder circuits are among the more important circuits in computer system design. They provide the ability to recognize (or ‘decode’) a string of bits. One very common use for a decoder is memory expansion. For example, suppose a system designer wishes to install four (identical) 256 MByte memory modules in a system to bring the total to one gigabyte of RAM. These 256 MByte memory modules have 28 address lines assuming each memory modules is eight bits wide ( $2^{28} \times 8$  bits is 256 MBytes)<sup>6</sup>. Unfortunately, if the system designer hooked up those four memory modules to the CPU’s address bus they would all respond to the same addresses on the bus. Pandemonium would result. To correct this problem, we need to select each memory module when a different set of addresses appear on the address bus. By adding a chip enable line to each of the memory modules and using a two-input, four-output decoder circuit, we can easily do this. See Figure 3.21 for the details.

6. Actually, most memory modules are wider than eight bits, so a real 256 MByte memory module will have fewer than 28 address lines, but we will ignore this technicality in this example.



**Figure 3.21 Adding Four 256 MByte Memory Modules to a System**

The two-line to four-line decoder circuit in Figure 3.21 actually incorporates four different logic functions, one function for each of the outputs. Assume the inputs are  $A$  and  $B$  ( $A=A_{28}$  and  $B=A_{29}$ ) then the four output functions have the following (simple) equations:

$$Q_0 = A' B'$$

$$Q_1 = A B'$$

$$Q_2 = A' B$$

$$Q_3 = A B$$

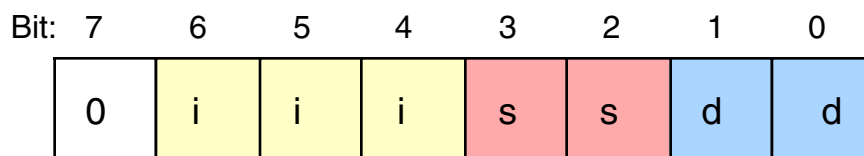
Following standard electronic circuit notation, these equations use “Q” to denote an output (electronic designers use “Q” for output rather than “O” because “Q” looks somewhat like an “O” and is more easily differentiated from zero). Also note that most circuit designers use *active low logic* for decoders and chip enables. This means that they enable a circuit with a low input value (zero) and disable the circuit with a high input value (one). Likewise, the output lines of a decoder chip are normally high and go low when the inputs select a given output line. This means that the equations above really need to be inverted for real-world examples. We’ll ignore this issue here and use positive (or active high) logic<sup>7</sup>.

Another big use for decoding circuits is to decode a byte in memory that represents a machine instruction in order to activate the corresponding circuitry to perform whatever tasks the instruction requires. We’ll cover this subject in much greater depth in a later chapter, but a simple example at this point will provide another solid example for using decoders.

Most modern (Von Neumann) computer systems represent machine instructions via values in memory. To execute an instruction the CPU fetches a value from memory, decodes that value, and then does the appropriate activity the instruction specifies. Obviously, the CPU uses decoding circuitry to decode the instruction. To see how this is done, let’s create a very simple CPU with a very simple instruction set. Figure 3.22 provides the instruction format (that is, it specifies all the numeric codes) for our simple CPU.

7. Electronic circuits often use active low logic because the circuits that employ them typically require fewer transistors to implement.

## Instruction (opcode) Format:



iii

000 = MOV  
 001 = ADD  
 010 = SUB  
 011 = MUL  
 100 = DIV  
 101 = AND  
 110 = OR  
 111 = XOR

ss &amp; dd

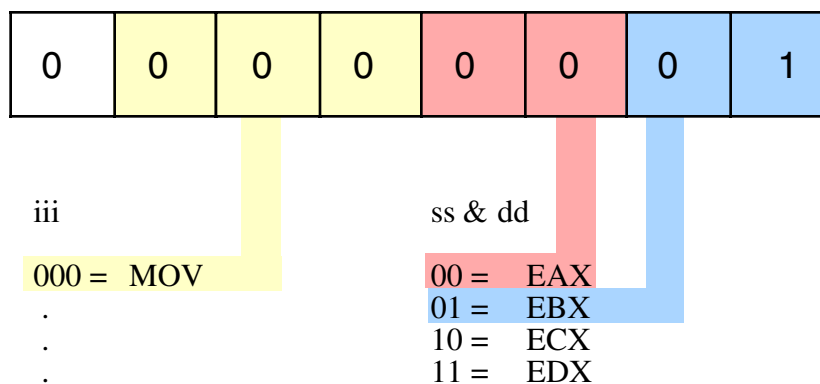
00 = EAX  
 01 = EBX  
 10 = ECX  
 11 = EDX

**Figure 3.22 Instruction (opcode) Format for a Very Simple CPU**

To determine the eight-bit operation code (opcode) for a given instruction, the first thing you do is choose the instruction you want to encode. Let's pick "MOV( EAX, EBX);" as our simple example. To convert this instruction to its numeric equivalent we must first look up the value for MOV in the *iii* table above; the corresponding value is 000. Therefore, we must substitute 000 for *iii* in the opcode byte.

Second, we consider our source operand. The source operand is EAX, whose encoding in the source operand table (*ss & dd*) is 00. Therefore, we substitute 00 for *ss* in the instruction opcode.

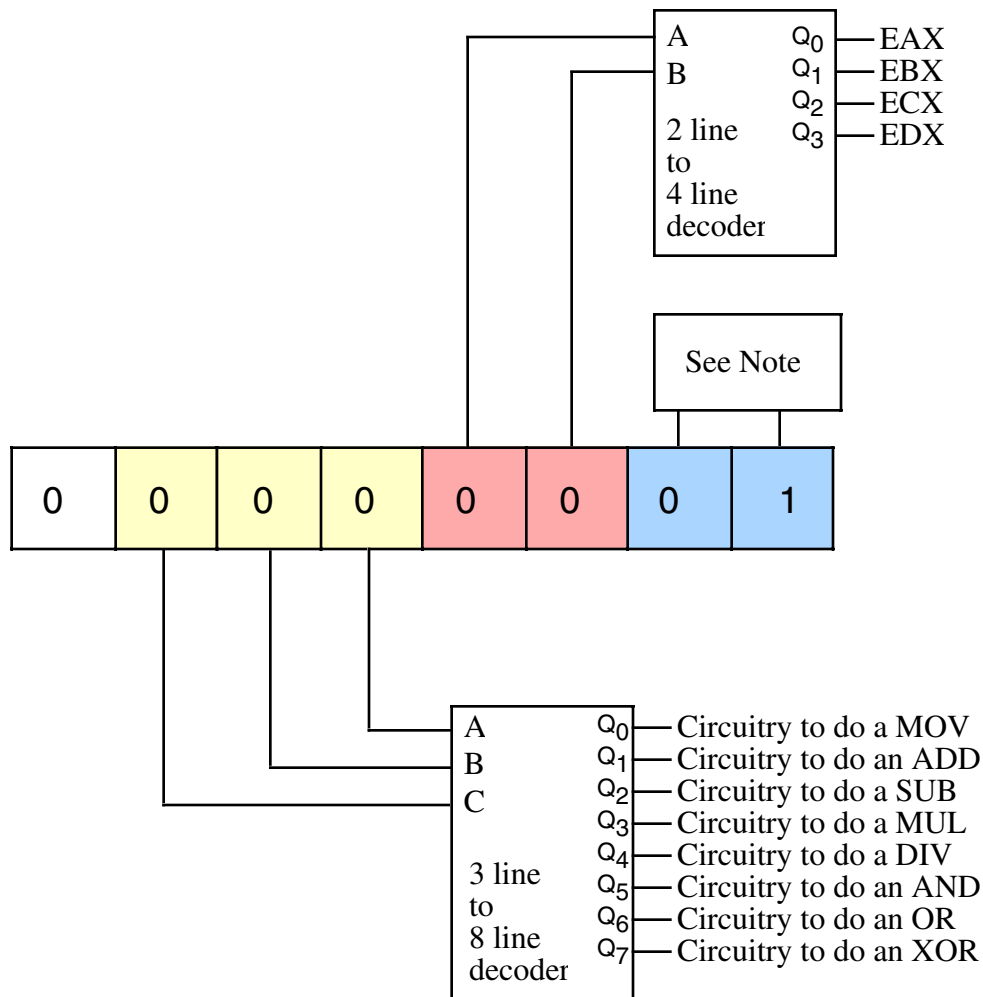
Next, we need to convert the destination operand to its numeric equivalent. Once again, we look up the value for this operand in the *ss & dd* table. The destination operand is EBX and its value is 01. So we substitute 01 for *dd* in our opcode byte. Assembling these three fields into the opcode byte (a packed data type), we obtain the following bit value: %00000001. Therefore, the numeric value \$1 is the value for the "MOV( EAX, EBX);" instruction (see Figure 3.23).

**Figure 3.23 Encoding the MOV( EAX, EBX ); Instruction**

As another example, consider the “AND( EDX, ECX);” instruction. For this instruction the *iii* field is %101, the *ss* field is %11, and the *dd* field is %10. This yields the opcode %01011110 or \$5E. You may easily create other opcodes for our simple instruction set using this same technique.

**Warning:** please do not come to the conclusion that these encodings apply to the 80x86 instruction set. The encodings in this examples are highly simplified in order to demonstrate instruction decoding. They do not correspond to any real-life CPU, and the especially don’t apply to the x86 family.

In these past few examples we were actually *encoding* the instructions. Of course, the real purpose of this exercise is to discover how the CPU can use a decoder circuit to decode these instructions and execute them at run time. A typical set of decoder circuits for this might look like that in Figure 3.24:



Note: the circuitry attached to the destination register bits is identical to the circuitry for the source register bits.

**Figure 3.24** Decoding Simple Machine Instructions

Notice how this circuit uses three separate decoders to decode the individual fields of the opcode. This is much less complex than creating a seven-line to 128-line decoder to decode each individual opcode. Of course, all that the circuit above will do is tell you which instruction and what operands a given opcode spec-

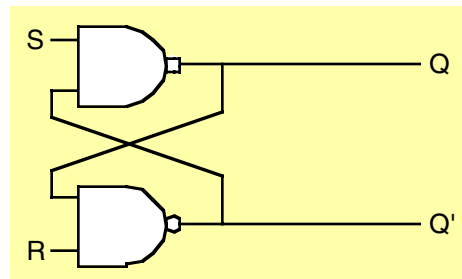
ifies. To actually execute this instruction you must supply additional circuitry to select the source and destination operands from an array of registers and act accordingly upon those operands. Such circuitry is beyond the scope of this chapter, so we'll save the juicy details for later.

Combinatorial circuits are the basis for many components of a basic computer system. You can construct circuits for addition, subtraction, comparison, multiplication, division, and many other operations using combinatorial logic.

### 3.7.3 Sequential and Clocked Logic

One major problem with combinatorial logic is that it is *memoryless*. In theory, all logic function outputs depend only on the current inputs. Any change in the input values is immediately reflected in the outputs<sup>8</sup>. Unfortunately, computers need the ability to *remember* the results of past computations. This is the domain of sequential or clocked logic.

A *memory cell* is an electronic circuit that remembers an input value after the removal of that input value. The most basic memory unit is the *set/reset flip-flop*. You can construct an *SR flip-flop* using two NAND gates, as shown in Figure 3.25.



**Figure 3.25 Set/Reset Flip Flop Constructed from NAND Gates**

The  $S$  and  $R$  inputs are normally high. If you *temporarily* set the  $S$  input to zero and then bring it back to one (*toggle* the  $S$  input), this forces the  $Q$  output to one. Likewise, if you toggle the  $R$  input from one to zero back to one, this sets the  $Q$  output to zero. The  $Q'$  input is generally the inverse of the  $Q$  output.

Note that if both  $S$  and  $R$  are one, then the  $Q$  output depends upon  $Q$ . That is, whatever  $Q$  happens to be, the top NAND gate continues to output that value. If  $Q$  was originally one, then there are two ones as inputs to the bottom flip-flop ( $Q$  and  $R$ ). This produces an output of zero ( $Q'$ ). Therefore, the two inputs to the top NAND gate are zero and one. This produces the value one as an output (matching the original value for  $Q$ ).

If the original value for  $Q$  was zero, then the inputs to the bottom NAND gate are  $Q=0$  and  $R=1$ . Therefore, the output of this NAND gate is one. The inputs to the top NAND gate, therefore, are  $S=1$  and  $Q'=1$ . This produces a zero output, the original value of  $Q$ .

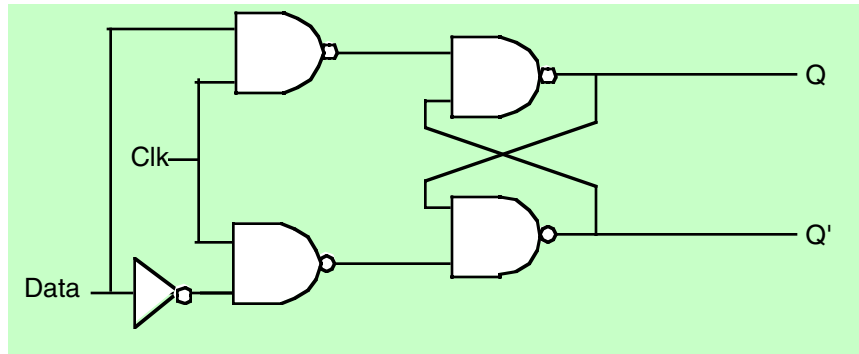
Suppose  $Q$  is zero,  $S$  is zero and  $R$  is one. This sets the two inputs to the top flip-flop to one and zero, forcing the output ( $Q$ ) to one. Returning  $S$  to the high state does not change the output at all. You can obtain this same result if  $Q$  is one,  $S$  is zero, and  $R$  is one. Again, this produces an output value of one. This value remains one even when  $S$  switches from zero to one. Therefore, toggling the  $S$  input from one to zero and then back to one produces a one on the output (i.e., *sets* the flip-flop). The same idea applies to the  $R$  input, except it forces the  $Q$  output to zero rather than to one.

There is one catch to this circuit. It does not operate properly if you set both the  $S$  and  $R$  inputs to zero simultaneously. This forces both the  $Q$  and  $Q'$  outputs to one (which is logically inconsistent). Whichever

8. In practice, there is a short *propagation delay* between a change in the inputs and the corresponding outputs in any electronic implementation of a boolean function.

input remains zero the longest determines the final state of the flip-flop. A flip-flop operating in this mode is said to be *unstable*.

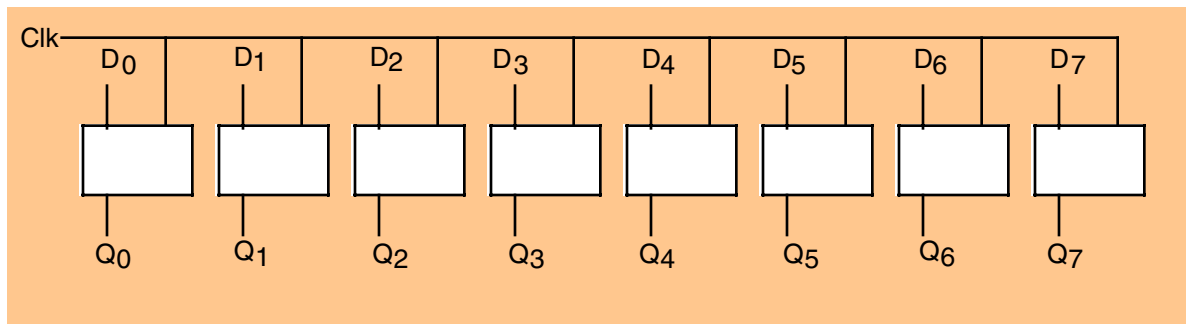
The only problem with the S/R flip-flop is that you must use separate inputs to remember a zero or a one value. A memory cell would be more valuable to us if we could specify the data value to remember on one input and provide a *clock input* to *latch* the input value. This type of flip-flop, the D flip-flop (for *data*) uses the circuit in Figure 3.26.



**Figure 3.26** Implementing a D flip-flop with NAND Gates

Assuming you fix the  $Q$  and  $Q'$  outputs to either 0/1 or 1/0, sending a *clock pulse* that goes from zero to one back to zero will copy the  $D$  input to the  $Q$  output. It will also copy  $D'$  to  $Q'$ . The exercises at the end of this topic section will expect you to describe this operation in detail, so study this diagram carefully.

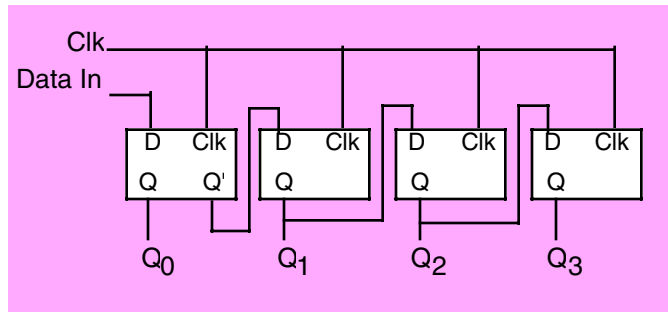
Although remembering a single bit is often important, in most computer systems you will want to remember a group of bits. You can remember a sequence of bits by combining several D flip-flops in parallel. Concatenating flip-flops to store an  $n$ -bit value forms a *register*. The electronic schematic in Figure 3.27 shows how to build an eight-bit register from a set of D flip-flops.



**Figure 3.27** An Eight-bit Register Implemented with Eight D Flip-flops

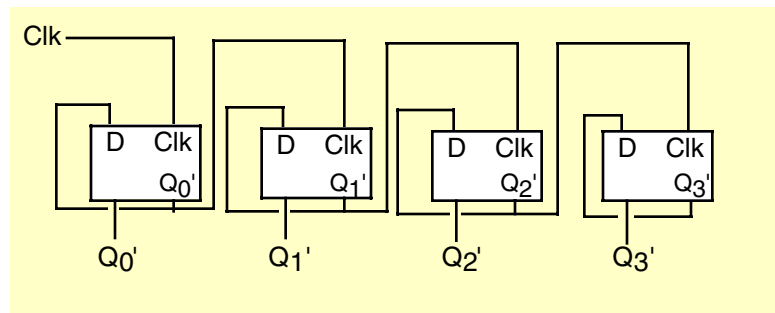
Note that the eight D flip-flops use a common clock line. This diagram does not show the  $Q'$  outputs on the flip-flops since they are rarely required in a register.

D flip-flops are useful for building many sequential circuits above and beyond simple registers. For example, you can build a *shift register* that shifts the bits one position to the left on each clock pulse. A four-bit shift register appears in Figure 3.28.



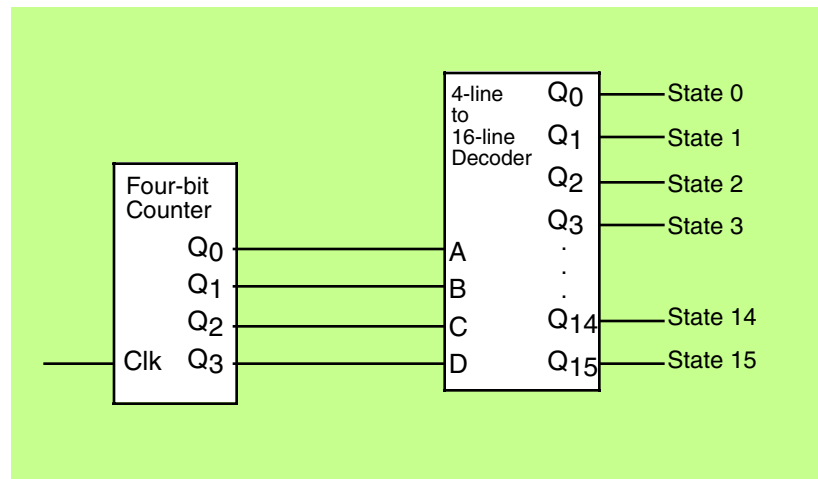
**Figure 3.28 A Four-bit Shift Register Built from D Flip-flops**

You can even build a *counter*, that counts the number of times the clock toggles from one to zero and back to one using flip-flops. The circuit in Figure 3.29 implements a four bit counter using D flip-flops.



**Figure 3.29 Four-bit Counter Built from D Flip-flops**

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits beyond these. For example, you can build a simple state machine known as a sequencer by combining a counter and a decoder as shown in Figure 3.30. For each cycle of the clock this sequencer activates one of its output lines. Those lines, in turn, may control other circuitry. By “firing” these circuits on each of the 16 output lines of the decoder, we can control the order in which these 16 different circuits accomplish their tasks. This is a fundamental need in a CPU since we often need to control the sequence of various operations (for example, it wouldn’t be a good thing if the “ADD( EAX, EBX);” instruction stored the result into EBX before fetching the source operand from EAX (or EBX). A simple sequencer such as this one can tell the CPU when to fetch the first operand, when to fetch the second operand, when to add them together, and when to store the result away. But we’re getting a little ahead of ourselves, we’ll discuss this in greater detail in the next chapter.



**Figure 3.30** A Simple 16-State Sequencer

### 3.8 Okay, What Does It Have To Do With Programming, Then?

Once you have registers, counters, and shift registers, you can build *state machines*. The implementation of an algorithm in hardware using state machines is well beyond the scope of this text. However, one important point must be made with respect to such circuitry – *any algorithm you can implement in software you can also implement directly in hardware*. This suggests that boolean logic is the basis for computation on all modern computer systems. Any program you can write, you can specify as a sequence of boolean equations.

Of course, it is much easier to specify a solution to a programming problem using languages like Pascal, C, or even assembly language than it is to specify the solution using boolean equations. Therefore, it is unlikely that you would ever implement an entire program using a set of state machines and other logic circuitry. Nevertheless, there are times when a hardware implementation is better. A hardware solution can be one, two, three, or more *orders of magnitude* faster than an equivalent software solution. Therefore, some time critical operations may require a hardware solution.

A more interesting fact is that the converse of the above statement is also true. Not only can you implement all software functions in hardware, but it is also possible to *implement all hardware functions in software*. This is an important revelation because many operations you would normally implement in hardware are *much cheaper* to implement using software on a microprocessor. Indeed, this is a primary use of *assembly language* in modern systems – to inexpensively replace a complex electronic circuit. It is often possible to replace many tens or hundreds of dollars of electronic components with a single \$5 microcomputer chip. The whole field of *embedded systems* deals with this very problem. Embedded systems are computer systems embedded in other products. For example, most microwave ovens, TV sets, video games, CD players, and other consumer devices contain one or more complete computer systems whose sole purpose is to replace a complex hardware design. Engineers use computers for this purpose because they are *less expensive* and *easier to design with* than traditional electronic circuitry.

You can easily design software that reads switches (input variables) and turns on motors, LEDs or lights, locks or unlocks a door, etc. (output functions). To write such software, you will need an understanding of boolean functions and how to implement such functions in software.

Of course, there is one other reason for studying boolean functions, even if you never intend to write software intended for an embedded system or write software that manipulates real-world devices. Many high level languages process boolean expressions (e.g., those expressions that control an IF statement or WHILE loop). By applying transformations like DeMorgan's theorems or a mapping optimization it is often possible

to improve the performance of high level language code. Therefore, studying boolean functions *is* important even if you never intend to design an electronic circuit. It can help you write better code in a traditional programming language.

For example, suppose you have the following statement in Pascal:

```
if ((x=y) and (a <> b)) or ((x=y) and (c <= d)) then SomeStmt;
```

You can use the distributive law to simplify this to:

```
if ((x=y) and ((a <> b) or (c <= d))) then SomeStmt;
```

Likewise, we can use DeMorgan's theorem to reduce

```
while (not((a=b) and (c=d))) do Something;
```

to

```
while (a <> b) or (c <> d) do Something;
```

So as you can see, understanding a little boolean algebra can actually help you write better software.

---

### 3.9 Putting It All Together

A good understanding of boolean algebra and digital design is absolutely necessary for anyone who wants to understand the internal operation of a CPU. As an added bonus, programmers who understand digital design can write better assembly language (and high level language) programs. This chapter provides a basic introduction to boolean algebra and digital circuit design. Although a detailed knowledge of this material isn't necessary if you simply want to write assembly language programs, this knowledge will help explain why Intel chose to implement instructions in certain ways; questions that will undoubtedly arise as we begin to look at the low-level implementation of the CPU.

This chapter is not, by any means, a complete treatment of this subject. If you're interested in learning more about boolean algebra and digital circuit design, there are dozens and dozens of texts on this subject available. Since this is a text on assembly language programming, we cannot afford to spend additional time on this subject; please see one of these other texts for more information.

# CPU Architecture

## Chapter Four

### 4.1 Chapter Overview

This chapter discusses history of the 80x86 CPU family and the major improvements occurring along the line. The historical background will help you better understand the design compromises they made as well as understand the legacy issues surrounding the CPU's design. This chapter also discusses the major advances in computer architecture that Intel employed while improving the x86<sup>1</sup>.

### 4.2 The History of the 80x86 CPU Family

Intel developed and delivered the first commercially viable microprocessor way back in the early 1970's: the 4004 and 4040 devices. These four-bit microprocessors, intended for use in calculators, had very little power. Nevertheless, they demonstrated the future potential of the microprocessor – an entire CPU on a single piece of silicon<sup>2</sup>. Intel rapidly followed their four-bit offerings with their 8008 and 8080 eight-bit CPUs. A small outfit in Santa Fe, New Mexico, incorporated the 8080 CPU into a box they called the Altair 8800. Although this was not the world's first "personal computer" (there were some limited distribution machines built around the 8008 prior to this), the Altair was the device that sparked the imaginations of hobbyists the world over and the personal computer revolution was born.

Intel soon had competition from Motorola, MOS Technology, and an upstart company formed by disgruntled Intel employees, Zilog. To compete, Intel produced the 8085 microprocessor. To the software engineer, the 8085 was essentially the same as the 8080. However, the 8085 had lots of hardware improvements that made it easier to design into a circuit. Unfortunately, from a software perspective the other manufacturer's offerings were better. Motorola's 6800 series was easier to program, MOS Technologies' 65xx family was easier to program and very inexpensive, and Zilog's Z80 chip was upwards compatible with the 8080 with lots of additional instructions and other features. By 1978 most personal computers were using the 6502 or Z80 chips, not the Intel offerings.

Sometime between 1976 and 1978 Intel decided that they needed to leap-frog the competition and produce a 16-bit microprocessor that offered substantially more power than their competitor's eight-bit offerings. This initiative led to the design of the 8086 microprocessor. The 8086 microprocessor was not the world's first (there were some oddball 16-bit microprocessors prior to this point) but it was certainly the highest performance single-chip 16-bit microprocessor when it was first introduced.

During the design timeframe of the 8086 memory was very expensive. Sixteen Kilobytes of RAM was selling above \$200 at the time. One problem with a 16-bit CPU is that programs tend to consume more memory than their counterparts on an eight-bit CPU. Intel, ever cognizant of the fact that designers would reject their CPU if the total system cost was too high, made a special effort to design an instruction set that had a high memory density (that is, packed as many instructions into as little RAM as possible). Intel achieved their design goal and programs written for the 8086 were comparable in size to code running on eight-bit microprocessors. However, those design decisions still haunt us today as you'll soon see.

At the time Intel designed the 8086 CPU the average lifetime of a CPU was only a couple of years. Their experiences with the 4004, 4040, 8008, 8080, and 8085 taught them that designers would quickly ditch the old technology in favor of the new technology as long as the new stuff was radically better. So Intel designed the 8086 assuming that whatever compromises they made in order to achieve a high instruction density would be fixed in newer chips. Based on their experience, this was a reasonable assumption.

Intel's competitors were not standing still. Zilog created their own 16-bit processor that they called the Z8000, Motorola created the 68000, their own 16-bit processor, and National Semiconductor introduced the

---

1. Note that Intel wasn't the inventor of most of these new technological advances. They simply duplicated research long since commercially employed by mainframe designers.

2. Prior to this point, commercial computer systems used multiple semiconductor devices to implement the CPU.

16032 device (later to be renamed the 32016). The designers of these chips had different design goals than Intel. Primarily, they were more interested in providing a reasonable instruction set for programmers even if their code density wasn't anywhere near as high as the 8086. The Motorola and National offers even provided 32-bit integer registers, making programming the chips even easier. All in all, these chips were much better (from a software development standpoint) than the Intel chip.

Intel wasn't resting on its laurels with the 8086. Immediately after the release of the 8086 they created an eight-bit version, the 8088. The purpose of this chip was to reduce system cost (since a minimal system could get by with half the memory chips and cheaper peripherals since the 8088 had an eight-bit data bus). In the very early 1980's, Intel also began work on their intended successor to the 8086 – the iAPX432 CPU. Intel fully expected the 8086 and 8088 to die away and that system designers who were creating general purpose computer systems would choose the '432 chip instead.

Then a major event occurred that would forever change history: in 1980 a small group at IBM got the go-ahead to create a "personal computer" along the likes of the Apple II and TRS-80 computers (the most popular PCs at the time). IBM's engineers probably evaluated lots of different CPUs and system designs. Ultimately, they settled on the 8088 chip. Most likely they chose this chip because they could create a minimal system with only 16 Kilobytes of RAM and a set of cheap eight-bit peripheral devices. So Intel's design goals of creating CPUs that worked well in low-cost systems landed them a very big "design win" from IBM.

Intel was still hard at work on the (ill-fated) iAPX432 project, but a funny thing happened – IBM PCs started selling far better than anyone had ever dreamed. As the popularity of the IBM PCs increased (and as people began "cloning" the PC), lots of software developers began writing software for the 8088 (and 8086) CPU, mostly in assembly language. In the meantime, Intel was pushing their iAPX432 with the Ada programming language (which was supposed to be the next big thing after Pascal, a popular language at the time). Unfortunately for Intel, no one was interested in the '432. Their PC software, written mostly in assembly language wouldn't run on the '432 and the '432 was notoriously slow. It took a while, but the iAPX432 project eventually died off completely and remains a black spot on Intel's record to this day.

Intel wasn't sitting pretty on the 8086 and 8088 CPUs, however. In the late 1970's and early 1980's they developed the 80186 and 80188 CPUs. These CPUs, unlike their previous CPU offerings, were fully upwards compatible with the 8086 and 8088 CPUs. In the past, whenever Intel produced a new CPU it did not necessarily run the programs written for the previous processors. For example, the 8086 did not run 8080 software and the 8080 did not run 4040 software. Intel, recognizing that there was a tremendous investment in 8086 software, decided to create an upgrade to the 8086 that was superior (both in terms of hardware capability and with respect to the software it would execute). Although the 80186 did not find its way into many PCs, it was a very popular chip in embedded applications (i.e., non-computer devices that use a CPU to control their functions). Indeed, variants of the 80186 are in common use even today.

The unexpected popularity of the IBM PC created a problem for Intel. This popularity obliterated the assumption that designers would be willing to switch to a better chip when such a chip arrived, even if it meant rewriting their software. Unfortunately, IBM and tens of thousands of software developers weren't willing to do this to make life easy for Intel. They wanted to stick with the 8086 software they'd written but they also wanted something a little better than the 8086. If they were going to be forced into jumping ship to a new CPU, the Motorola, Zilog, and National offerings were starting to look pretty good. So Intel did something that saved their bacon and has infuriated computer architects ever since: they started creating upwards compatible CPUs that continued to execute programs written for previous members of their growing CPU family while adding new features.

As noted earlier, memory was very expensive when Intel first designed the 8086 CPU. At that time, computer systems with a megabyte of memory usually cost megabucks. Intel was expecting a typical computer system employing the 8086 to have somewhere between 4 Kilobytes and 64 Kilobytes of memory. So when they designed in a one megabyte limitation, they figured no one would ever install that much memory in a system. Of course, by 1983 people were still using 8086 and 8088 CPUs in their systems and memory prices had dropped to the point where it was very common to install 640 Kilobytes of memory on a PC (the IBM PC design effectively limited the amount of RAM to 640 Kilobytes even though the 8086 was capable of addressing one megabyte). By this time software developers were starting to write more sophisticated programs and users were starting to use these programs in more sophisticated ways. The bottom line was that everyone was bumping up against the one megabyte limit of the 8086. Despite the investment in exist-

ing software, Intel was about to lose their cash cow if they didn't do something about the memory addressing limitations of their 8086 family (the 68000 and 32016 CPUs could address up to 16 Megabytes at the time and many system designers [e.g., Apple] were defecting to these other chips). So Intel introduced the 80286 which was a bit improvement over the previous CPUs. The 80286 added lots of new instructions to make programming a whole lot easier and they added a new "protected" mode of operation that allowed access to as much as 16 megabytes of memory. They also improved the internal operation of the CPU and bumped up the clock frequency so that the 80286 ran about 10 times faster than the 8088 in IBM PC systems.

IBM introduced the 80286 in their IBM PC/AT (AT = "advanced technology"). This change proved enormously popular. PC/AT clones based on the 80286 started appearing everywhere and Intel's financial future was assured.

Realizing that the 80x86 (x = "", "1", or "2") family was a big money maker, Intel immediately began the process of designing new chips that continued to execute the old code while improving performance and adding new features. Intel was still playing catch-up with their competitors in the CPU arena with respect to features, but they were definitely the king of the hill with respect to CPUs installed in PCs. One significant difference between Intel's chips and many of their competitors was that their competitors (notably Motorola and National) had a 32-bit internal architecture while the 80x86 family was stuck at 16-bits. Again, concerned that people would eventually switch to the 32-bit devices their competitors offered, Intel upgraded the 80x86 family to 32 bits by adding the 80386 to the product line.

The 80386 was truly a remarkable chip. It maintained almost complete compatibility with the previous 16-bit CPUs while fixing most of the real complaints people had with those older chips. In addition to supporting 32-bit computing, the 80386 also bumped up the maximum addressability to four gigabytes as well as solving some problems with the "segmented" organization of the previous chips (a big complaint by software developers at the time). The 80386 also represented the most radical change to ever occur in the 80x86 family. Intel more than doubled the total number of instructions, added new memory management facilities, added hardware debugging support for software, and introduced many other features. Continuing the trend they set with the 80286, the 80386 executed instructions faster than previous generation chips, even when running at the same clock speed plus the new chip ran at a higher clock speed than the previous generation chips. Therefore, it ran existing 8088 and 80286 programs faster than on these older chips. Unfortunately, while people adopted the new chip for its higher performance, they didn't write new software to take advantage of the chip's new features. But more on that in a moment.

Although the 80386 represented the most radical change in the 80x86 architecture from the programmer's view, Intel wasn't done wringing all the performance out of the x86 family. By the time the 80386 appeared, computer architects were making a big noise about the so-called RISC (Reduced Instruction Set Computer) CPUs. While there were several advantages to these new RISC chips, a important advantage of these chips is that they purported to execute one instruction every clock cycle. The 80386 instructions required a wildly varying number of cycles to execute ranging from a few cycles per instruction to well over a hundred. Although comparing RISC processors directly with the 80386 was dangerous (because many 80386 instructions actually did the work of two or more RISC instructions), there was a general perception that, at the same clock speed, the 80386 was slower since it executed fewer instructions in a given amount of time.

The 80486 CPU introduced two major advances in the x86 design. First, the 80486 integrated the floating point unit (or FPU) directly onto the CPU die. Prior to this point Intel supplied a separate, external, chip to provide floating point calculations (these were the 8087, 80287, and 80387 devices). By incorporating the FPU with the CPU, Intel was able to speed up floating point operations and provide this capability at a lower cost (at least on systems that required floating point arithmetic). The second major architectural advance was the use of *pipelined instruction execution*. This feature (which we will discuss in detail a little later in this chapter) allowed Intel to overlap the execution of two or more instructions. The end result of pipelining is that they effectively reduced the number of cycles each instruction required for execution. With pipelining, many of the simpler instructions had an aggregate throughput of one instruction per clock cycle (under ideal conditions) so the 80486 was able to compete with RISC chips in terms of clocks per instruction cycle.

While Intel was busy adding pipelining to their x86 family, the companies building RISC CPUs weren't standing still. To create ever faster and faster CPU offerings, RISC designers began creating *superscalar* CPUs that could actually execute more than one instruction per clock cycle. Once again, Intel's CPUs were perceived as following the leaders in terms of CPU performance. Another problem with Intel's CPU is that

the integrated FPU, though faster than the earlier models, was significantly slower than the FPUs on the RISC chips. As a result, those designing high-end engineering workstations (that typically require good floating point hardware support) began using the RISC chips because they were faster than Intel's offerings.

From the programmer's perspective, there was very little difference between an 80386 with an 80387 FPU and an 80486 CPU. There were only a handful of new instructions (most of which had very little utility in standard applications) and not much in the way of other architectural features that software could use. The 80486, from the software engineer's point of view, was just a really fast 80386/80387 combination.

So Intel went back to their CAD<sup>3</sup> tools and began work on their next CPU. This new CPU featured a superscalar design with vastly improved floating point performance. Finally, Intel was closing in on the performance of the RISC chips. Like the 80486 before it, this new CPU added only a small number of new instructions and most of those were intended for use by operating systems, not application software.

Intel did not designate this new chip the 80586. Instead, they called it the *Pentium™ Processor*<sup>4</sup>. The reason they discontinued referring to processors by number and started naming them was because of confusion in the marketplace. Intel was not the only company producing x86 compatible CPUs. AMD, Cyrix, and a host of others were also building and selling these chips in direct competition with Intel. Until the 80486 came along, the internal design of the CPUs were relatively simple and even small companies could faithfully reproduce the functionality of Intel's CPUs. The 80486 was a different story altogether. This chip was quite complex and taxed the design capabilities of the smaller companies. Some companies, like AMD, actually licensed Intel's design and they were able to produce chips that were compatible with Intel's (since they were, effectively, Intel's chips). Other companies attempted to create their own version of the 80486 and fell short of the goal. Perhaps they didn't integrate an FPU or the new instructions on the 80486. Many didn't support pipelining. Some chips lacked other features found on the 80486. In fact, most of the (non-Intel) chips were really 80386 devices with some very slight improvements. Nevertheless, they called these chips 80486 CPUs.

This created massive confusion in the marketplace. Prior to this, if you'd purchased a computer with an 80386 chip you knew the capabilities of the CPU. All 80386 chips were equivalent. However, when the 80486 came along and you purchased a computer system with an 80486, you didn't know if you were getting an actual 80486 or a remarked 80386 CPU. To counter this, Intel began their enormously successful "Intel Inside" campaign to let people know that there was a difference between Intel CPUs and CPUs from other vendors. This marketing campaign was so successful that people began specifying Intel CPUs even though some other vendor's chips (i.e., AMD) were completely compatible.

Not wanting to repeat this problem with the 80586 generation, Intel ditched the numeric designation of their chips. They created the term "Pentium Processor" to describe their new CPU so they could trademark the name and prevent other manufacturers from using the same designation for their chip. Initially, of course, savvy computer users griped about Intel's strong-arm tactics but the average user benefited quite a bit from Intel's marketing strategy. Other manufacturers release their own 80586 chips (some even used the "586" designation), but they couldn't use the Pentium Processor name on their parts so when someone purchased a system with a Pentium in it, they knew it was going to have all the capabilities of Intel's chip since it had to be Intel's chip. This was a good thing because most of the other '586 class chips that people produced at that time were not as powerful as the Pentium.

The Pentium cemented Intel's position as champ of the personal computer. It had near RISC performance and ran tons of existing software. Only the Apple Macintosh and high-end UNIX workstations and servers went the RISC route. Together, these machines comprised less than 10% of the total desktop computer market.

Intel still was not satisfied. They wanted to control the server market as well. So they developed the Pentium Pro CPU. The Pentium Pro had a couple of features that made it ideal for servers. Intel improved the 32-bit performance of the CPU (at the expense of its 16-bit performance), they added better support for multiprocessing to allow multiple CPUs in a system (high-end servers usually have two or more processors), and they added a handful of new instructions to improve the performance of certain instruction sequences on the pipelined architecture. Unfortunately, most application software written at the time of the Pentium Pro's

---

3. Computer aided design.

4. Pentium Processor is a registered trademark of Intel Corporation. For legal reasons Intel could not trademark the name Pentium by itself, hence the full name of the CPU is the "Pentium Processor".

release was 16-bit software which actually ran slower on the Pentium Pro than it did on a Pentium at equivalent clock frequencies. So although the Pentium Pro did wind up in a few server machines, it was never as popular as the other chips in the Intel line.

The Pentium Pro had another big strike against it: shortly after the introduction of the Pentium Pro, Intel's engineers introduced an upgrade to the standard Pentium chip, the MMX (multimedia extension) instruction set. These new instructions (nearly 60 in all) gave the Pentium additional power to handle computer video and audio applications. These extensions became popular overnight, putting the last nail in the Pentium Pro's coffin. The Pentium Pro was slower than the standard Pentium chip and slower than high-end RISC chips, so it didn't see much use.

Intel corrected the 16-bit performance in the Pentium Pro, added the MMX extensions and called the result the Pentium II<sup>5</sup>. The Pentium II demonstrated an interesting point. Computers had reached a point where they were powerful enough for most people's everyday activities. Prior to the introduction of the Pentium II, Intel (and most industry pundits) had assumed that people would always want more power out of their computer systems. Even if they didn't need the machines to run faster, surely the software developers would write larger (and slower) systems requiring more and more CPU power. The Pentium II proved this idea wrong. The average user needed email, word processing, Internet access, multimedia support, simple graphics editing capabilities, and a spreadsheet now and then. Most of these applications, at least as home users employed them, were fast enough on existing CPUs. The applications that were slow (e.g., Internet access) were generally beyond the control of the CPU (i.e., the modem was the bottleneck not the CPU). As a result, when Intel introduced their pricey Pentium II CPUs, they discovered that system manufacturers started buying other people's x86 chips because they were far less expensive and quite suitable for their customer's applications. This nearly stunned Intel since it contradicted their experience up to that point.

Realizing that the competition was capturing the low-end market and stealing sales away, Intel devised a low-cost (lower performance) version of the Pentium II that they named *Celeron*<sup>6</sup>. The initial Celerons consisted of a Pentium II CPU without the on-board level two cache. Without the cache, the chip ran only a little bit better than half the speed of the Pentium II part. Nevertheless, the performance was comparable to other low-cost parts so Intel's fortunes improved once more.

While designing the low-end Celeron, Intel had not lost sight of the fact that they wanted to capture a chunk of the high-end workstation and server market as well. So they created a third version of the Pentium II, the Xeon Processor with improved cache and the capability of multiprocessor more than two CPUs. The Pentium II supports a two CPU multiprocessor system but it isn't easy to expand it beyond this number; the Xeon processor corrected this limitation. With the introduction of the Xeon processor (plus special versions of Unix and Windows NT), Intel finally started to make some serious inroads into the server and high-end workstation markets.

You can probably imagine what followed the Pentium II. Yep, the Pentium III. The Pentium III introduced the SIMD (pronounced SIM-DEE) extensions to the instruction set. These new instructions provided high performance floating point operations for certain types of computations that allow the Pentium III to compete with high-end RISC CPUs. The Pentium III also introduced another handful of integer instructions to aid certain applications.

With the introduction of the Pentium III, nearly all serious claims about RISC chips offering better performance were fading away. In fact, for most applications, the Intel chips were actually faster than the RISC chips available at the time. As this is being written, Intel was just introducing the Pentium IV chip and was slating it to run at 1.4 GHz, a much higher clock frequency than its RISC contemporaries. One would think that Intel would soon own it all. Surely by the time of the Pentium V, the RISC competition wouldn't be a factor anymore.

There is one problem with this theory: even Intel is admitting that they've pushed the x86 architecture about as far as they can. For nearly 20 years, computer architects have blasted Intel's architecture as being gross and bloated having to support code written for the 8086 processor way back in 1978. Indeed, Intel's design decisions (like high instruction density) that seemed so important in 1978 are holding back the CPU today. So-called "clean" designs, that don't have to support legacy applications, allow CPU designers to cre-

---

5. Interestingly enough, by the time the Pentium II appeared, the 16-bit efficiency was no longer a factor since most software was written as 32-bit code.

6. The term "Celeron Processor" is also an Intel trademark.

ate high-performance CPUs with far less effort than Intel's. Worse, those decisions Intel made in the 1976-1978 time frame are beginning to catch up with them and will eventually stall further development of the CPU. Computer architects have been warning everyone about this problem for twenty years; it is a testament to Intel's design effort (and willingness to put money into R&D) that they've taken the CPU as far as they have.

The biggest problem on the horizon is that most RISC manufacturers are now extending their architectures to 64-bits. This has two important impacts on computer systems. First, arithmetic calculations will be somewhat faster as will many internal operations and second, the CPUs will be able to directly address more than four gigabytes of main memory. This last factor is probably the most important for server and workstation systems. Already, high-end servers have more than four gigabytes installed. In the future, the ability to address more than four gigabytes of physical RAM will become essential for servers and high-end workstations. As the price of a gigabyte or more of memory drops below \$100, you'll see low-end personal computers with more than four gigabytes installed. To effectively handle this kind of memory, Intel will need a 64-bit processor to compete with the RISC chips.

Perhaps Intel has seen the light and decided it's time to give up on the x86 architecture. Towards the middle to end of the 1990's Intel announced that they were going to create a partnership with Hewlett-Packard to create a new 64-bit processor based around HP's PA-RISC architecture. This new 64-bit chip would execute x86 code in a special "emulation" mode and run native 64-bit code using a new instruction set. It's too early to tell if Intel will be successful with this strategy, but there are some major risks (pardon the pun) with this approach. First, the first such CPUs (just becoming available as this is being written) run 32-bit code far slower than the Pentium IV chip. Not only does the emulation of the x86 instruction set slow things down, but the clock speeds of the early CPUs are half the speed of the Pentium IVs. This is roughly the same situation Intel had with the Pentium Pro running 16-bit code slower than the Pentium. Second, the 64-bit CPUs (the IA64 family) rely heavily on compiler technology and are using a commercially untested architecture. This is similar to the situation with the iAPX432 project that failed quite miserably. Hopefully Intel knows what they're doing and ten years from now we'll all be using IA64 processors and wondering why anyone ever stuck with the IA32. On the other hand, hopefully Intel has a back-up plan in case the IA64 initiative fails.

Intel is betting that people will move to the IA64 when they need 64-bit computing capabilities. AMD, on the other hand, is betting that people would rather have a 64-bit x86 processor. Although the details are sketchy, AMD has announced that they will extend the x86 architecture to 64 bits in much the same way that Intel extend the 8086 and 80286 to 32-bits with the introduction of the the 80386 microprocessor. Only time will tell if Intel or AMD (or both) are successful with their visions.

**Table 25: 80x86 CPU Family**

Processor	Date of Introduction	Transistors on Chip	Maximum MIPS at Introduction <sup>a</sup>	Maximum Clock Frequency at Introduction <sup>b</sup>	On-chip Cache Memory	Maximum Addressable Memory
8086	1978	29K	0.8	8 MHz		1 MB
80286	1982	134K	2.7	12.5 MHz		16 MB
80386	1985	275K	6	20 MHz		4 GB
80486	1989	1.2M	20	25 MHz <sup>c</sup>	8K Level 1	4 GB
Pentium	1993	3.1M	100	60MHz	16K Level 1	4 GB
Pentium Pro	1995	5.5M	440	200 MHz	16K Level 1, 256K/512K Level 2	64 GB

**Table 25: 80x86 CPU Family**

Processor	Date of Introduction	Transistors on Chip	Maximum MIPS at Introduction <sup>a</sup>	Maximum Clock Frequency at Introduction <sup>b</sup>	On-chip Cache Memory	Maximum Addressable Memory
Pentium II	1997	7M	466	266 MHz	32K Level 1, 256/512K Level 2	64 GB
Pentium III	1999	8.2M	1,000	500 MHz	32K Level 1, 512K Level 2	64 GB

a. By the introduction of the next generation this value was usually higher.

b. Maximum clock frequency at introduction was very limited sampling. Usually, the chips were available at the next lower clock frequency in Intel's scale. Also note that by the introduction of the next generation this value was usually much higher.

c. Shortly after the introduction of the 25MHz 80486, Intel began using "Clock doubling" techniques to run the CPU twice as fast internally as the external clock. Hence, a 50 MHz 80486 DX2 chip was really running at 25 MHz externally and 50 MHz internally. Most chips after the 80486 employ a different internal clock frequency compared to the external (or "bus") frequency.

### 4.3 A History of Software Development for the x86

A section on the history of software development may seem unusual in a chapter on CPU Architecture. However, the 80x86's architecture is inexorably tied to the development of the software for this platform. Many architectural design decisions were a direct result of ensuring compatibility with existing software. So to fully understand the architecture, you must know a little bit about the history of the software that runs on the chip.

From the date of the very first working sample of the 8086 microprocessor to the latest and greatest IA-64 CPU, Intel has had an important goal: as much as possible, ensure compatibility with software written for previous generations of the processor. This mantra existed even on the first 8086, before there was a previous generation of the family. For the very first member of the family, Intel chose to include a modicum of compatibility with their previous eight-bit microprocessor, the 8085. The 8086 was not capable of running 8085 software, but Intel designed the 8086 instruction set to provide almost a one for one mapping of 8085 instructions to 8086 instructions. This allowed 8085 software developers to easily translate their existing assembly language programs to the 8086 with very little effort (in fact, software translators were available that did about 85% of the work for these developers).

Intel did not provide *object code compatibility*<sup>7</sup> with the 8085 instruction set because the design of the 8085 instruction set did not allow the expansion Intel needed for the 8086. Since there was very little software running on the 8085 that needed to run on the 8086, Intel felt that making the software developers responsible for this translation was a reasonable thing to do.

When Intel introduced the 8086 in 1978, the majority of the world's 8085 (and Z80) software was written in Microsoft's BASIC running under Digital Research's CP/M operating system. Therefore, to "port" the majority of business software (such that it existed at the time) to the 8086 really only required two things: porting the CP/M operating system (which was less than eight kilobytes long) and Microsoft's BASIC (most versions were around 16 kilobytes at the time). Porting such small programs may have seemed like a bit of work to developers of that era, but such porting is trivial compared with the situation that exists today. Anyway, as Intel expected, both Microsoft and Digital Research ported their products to the 8086 in short

7. That is, the ability to run 8085 machine code directly.

order so it was possible for a large percentage of the 8085 software to run on 8086 within about a year of the 8086's introduction.

Unfortunately, there was no great rush by computer hobbyists (the computer users of that era) to switch to the 8086. About this time the Radio Shack TRS-80 and the Apple II microcomputer systems were battling for supremacy of the home computer market and no one was really making computer systems utilizing the 8086 that appealed to the mass market. Intel wasn't doing poorly with the 8086; its market share, when you compared it with the other microprocessors, was probably better than most. However, the situation certainly wasn't like it is today (circa 2001) where the 80x86 CPU family owns 85% of the general purpose computer market.

The 8086 CPU, and its smaller sibling, the eight-bit 8088, was happily raking in its portion of the microprocessor market and Intel naturally assumed that it was time to start working on a 32-bit processor to replace the 8086 in much the same way that the 8086 replaced the eight-bit 8085. As noted earlier, this new processor was the ill-fated iAPX 432 system. The iAPX 432 was such a dismal failure that Intel might not have survived had it not been for a big stroke of luck – IBM decided to use the 8088 microprocessor in their personal computer system.

To most computer historians, there were two watershed events in the history of the personal computer. The first was the introduction of the Visicalc spreadsheet program on the Apple II personal computer system. This single program demonstrated that there was a real reason for owning a computer beyond the nerdy "gee, I've got my own computer" excuse. Visicalc quickly (and, alas, briefly) made Apple Computer the largest PC company around. The second big event in the history of personal computers was, of course, the introduction of the IBM PC. The fact that IBM, a "real" computer company, would begin building PCs legitimized the market. Up to that point, businesses tended to ignore PCs and treated them as toys that nerdy engineers liked to play with. The introduction of the IBM PC caused a lot of businesses to take notice of these new devices. Not only did they take notice, but they liked what they saw. Although IBM cannot make the claim that they started the PC revolution, they certainly can take credit for giving it a big jumpstart early on in its life.

Once people began buying lots of PCs, it was only natural that people would start writing and selling software for these machines. The introduction of the IBM PC greatly expanded the marketplace for computer systems. Keep in mind that at the time of the IBM PC's introduction, most computer systems had only sold tens of thousands of units. The more popular models, like the TRS-80 and Apple II had only sold hundreds of thousands of units. Indeed, it wasn't until a couple of years after the introduction of the IBM PC that the first computer system sold one million units; and that was a Commodore 64 system, not the IBM PC.

For a brief period, the introduction of the IBM PC was a godsend to most of the other computer manufacturers. The original IBM PC was underpowered and quite a bit more expensive than its counterparts. For example, a dual-floppy disk drive PC with 64 Kilobytes of memory and a monochrome display sold for \$3,000. A comparable Apple II system with a color display sold for under \$2,000. The original IBM PC with its 4.77 MHz 8088 processor (that's four-point-seven-seven, not four hundred seventy-seven!) was only about two to three times as fast as the Apple II with its paltry 1 MHz eight-bit 6502 processor. The fact that most Apple II software was written by expert assembly language programmers while most (early) IBM software was written in a high level language (often interpreted) or by inexperienced 8086 assembly language programmers narrowed the gap even more.

Nonetheless, software development on PCs accelerated. The wide range of different (and incompatible) systems made software development somewhat risky. Those who did not have an emotional attachment to one particular company (and didn't have the resources to develop for more than one platform) generally decided to go with IBM's PC when developing their software.

One problem with the 8086's architecture was beginning to show through by 1983 (remember, this is five years after Intel introduced the 8086). The *segmented memory architecture* that allowed them to extend their 16-bit addressing scheme to 20 bits (allowing the 8086 to address a megabyte of memory) was being attacked on two fronts. First, this segmented addressing scheme was difficult to use in a program, especially if a program needed to access more than 64 kilobytes of data or, worse yet, needed to access a single data structure that was larger than 64K long. By 1983 software had reached the level of sophistication that most programs were using this much memory and many needed large data structures. The software community as a whole began to grumble and complain about this segmented memory architecture and what a stupid thing it was.

The second problem with Intel's segmented architecture is that it only supported a maximum of a one megabyte address space. Worse, the design of the IBM PC effectively limited the amount of RAM the system could have to 640 kilobytes. This limitation was also beginning to create problems for more sophisticated programs running on the PC. Once again, the software development community grumbled and complained about Intel's segmented architecture and the limitations it imposed upon their software.

About the time people began complaining about Intel's architecture, Intel began running an ad campaign bragging about how great their chip was. They quoted top executives at companies like Visicorp (the outfit selling Visicalc) who claimed that the segmented architecture was great. They also made a big deal about the fact that over a billion dollars worth of software had been written for their chip. This was all marketing hype, of course. Their chip was not particularly special. Indeed, the 8086's contemporaries (Z8000, 68000, and 16032) were architecturally superior. However, Intel was quite right about one thing – people had written a lot of software for the 8086 and most of the really good stuff was written in 8086 assembly language and could not be easily ported to the other processors. Worse, the software that people were writing for the 8086 was starting to get large; making it even more difficult to port it to the other chips. As a result, software developers were becoming locked into using the 8086 CPU.

About this time Intel undoubtedly realized that they were getting locked into the 80x86 architecture, as well. The iAPX 432 project was on its death bed. People were no more interested in the iAPX 432 than they were the other processors (in fact, they were less interested). So Intel decided to do the only reasonable thing – extend the 8086 family so they could continue to make more money off their cash cow.

The first real extension to the 8086 family that found its way into general purpose PCs was the 80286 that appeared in 1982. This CPU answered the second complaint by adding the ability to address up to 16 MBytes of RAM (a formidable amount in 1982). Unfortunately, it did not extend the segment size beyond 64 kilobytes. In 1985 Intel introduced the 80386 microprocessor. This chip answered most of the complaints about the x86 family, and then some, but people still complained about these problems for nearly ten years after the introduction of the 80386.

Intel was suffering at the hands of Microsoft and the installed base of existing PCs. When IBM introduced the floppy disk drive for the IBM PC they didn't choose an operating system to ship with it. Instead, they offered their customers a choice of the widely available operating systems at the time. Of course, Digital Research had ported CP/M to the PC, UCSD/Softech had ported UCSD Pascal (a very popular language/operating system at the time) to the PC, and Microsoft had quickly purchased a CP/M knock-off named QD DOS (for Quick and Dirty DOS) from Seattle Microsystems, relabelled it "MS-DOS", and offered this as well. CP/M-86 cost somewhere in the vicinity of \$595. UCSD Pascal was selling for something like \$795. MS-DOS was selling for \$50. Guess which one sold more copies! Within a year, almost no one ran CP/M or UCSD Pascal on PCs. Microsoft and MS-DOS (also called IBM DOS) ruled the PC.

MS-DOS v1.0 lived up to its "quick and dirty" heritage. Working furiously, Microsoft's engineers added lots of new features (many taken from the UNIX operating system and shell program) and MS-DOS v2.0 appeared shortly thereafter. Although still crude, MS-DOS v2.0 was a substantial improvement and people started writing tons of software for it.

Unfortunately, MS-DOS, even in its final version, wasn't the best operating system design. In particular, it left all but rudimentary control of the hardware to the application programmer. It provided a file system so application writers didn't have to deal with the disk drive and it provided mediocre support for keyboard input and character display. It provided nearly useless support for other devices. As a result, most application programmers (and most high level languages) bypassed MS-DOS' device control and used MS-DOS primarily as a file system module.

In addition to poor device management, MS-DOS provided nearly non-existent memory management. For all intents and purposes, once MS-DOS started a program running, it was that program's responsibility to manage the system's resources. Not only did this create extra work for application programmers, but it was one of the main reasons most software could not take advantage of the new features Intel was adding to their microprocessors.

When Intel introduced the 80286 and, later, the 80386, the only way to take advantage of their extra addressing capabilities and the larger segments of the 80386 was to operate in a so-called *protected mode*. Unfortunately, neither MS-DOS nor most applications (that managed memory themselves) were capable of operating in protected mode without substantial change (actually, it would have been easy to modify

MS-DOS to use protected mode, but it would have broken all the existing software that ran under MS-DOS; Microsoft, like Intel, couldn't afford to alienate the software developers in this manner).

Even if Microsoft could magically make MS-DOS run under protected mode, they couldn't afford to do so. When Intel introduced the 80386 microprocessor it was a very expensive device (the chip itself cost over \$1,000 at initial introduction). Although the 80286 had been out for three years, systems built around the 8088 were still extremely popular (since they were much lower cost than systems using the 80386). Software developers had a choice: they could solve their memory addressing problems and use the new features of the 80386 chip but limit their market to the few who had 80386 systems, or they could continue to suffer with the 64K segment limitation imposed by the 8088 and MS-DOS and be able to sell their software to millions of users who owned one of the earlier machines. The marketing departments of these companies ruled the day, all software was written to run on plain 8088 boxes so that it had a larger market. It wasn't until 1995, when Microsoft introduced Windows 95 that people finally felt they could abandon processors earlier than the 80386. The end result was the people were still complaining about the Intel architecture and its 64K segment limitation ten years after Intel had corrected the problem. The concept of upwards compatibility was clearly a double-edged sword in this case.

Segmentation had developed such a bad name over the years that Microsoft abandoned the use of segments in their 32-bit versions of Windows (95, 98, NT, 2000, ME, etc.). In a couple of respects, this was a real shame because Intel finally did segmentation right (or, at least, pretty good) in the 80386 and later processors. By not allowing the use of segmentation in Win32 programs Microsoft limited the use of this powerful feature. They also limited their users to a maximum address space of 4GB (the Pentium Pro and later processors were capable of addressing 64GB of physical memory). Considering that many applications are starting to push the 4GB barrier, this limitation on Microsoft's part was ill-considered. Nevertheless, the "flat" memory model that Microsoft employs is easier to write software for, undoubtedly a big part of their decision not to use segmentation.

The introduction of Windows NT, that actually ran on CPUs other than Intel's, must have given Intel a major scare. Fortunately for Intel, NT was an asbysmal failure on non-Intel architectures like the Alpha and the PowerPC. On the other hand, the new Windows architecture does make it easier to move existing applications to 64-bit processors like the IA-64; so maybe WinNT's flexibility will work to Intel's advantage after all.

The 8086 software legacy has both advanced and retarded the 80x86 architecture. On the one hand, had software developers not written so much software for the 80x86, Intel would have abandoned the family in favor of something better a long time ago (not an altogether bad thing, in many people's opinions). On the other hand, however, the general acceptance of the 80386 and later processors was greatly delayed by the fact that software developers were writing software for the installed base of processors.

Around 1996, two types of software actually accelerated the design and acceptance of Intel's newer processors: multimedia software and games. When Intel introduced the MMX extensions to the 80x86 instruction set, software developers ignored the installed base and immediately began writing software to take advantage of these new instructions. This change of heart took place because the MMX instructions allowed developers to do things they hadn't been able to do before - not simply run faster, but run fast enough to display actual video and quick render 3D images. Combined with a change in pricing policy by Intel on new processor technology, the public quickly accepted these new systems.

Hard-core gamers, multimedia artists, and others quickly grabbed new machines and software as it became available. More often than not, each new generation of software would only run on the latest hardware, forcing these individuals to upgrade their equipment far more rapidly than ever before.

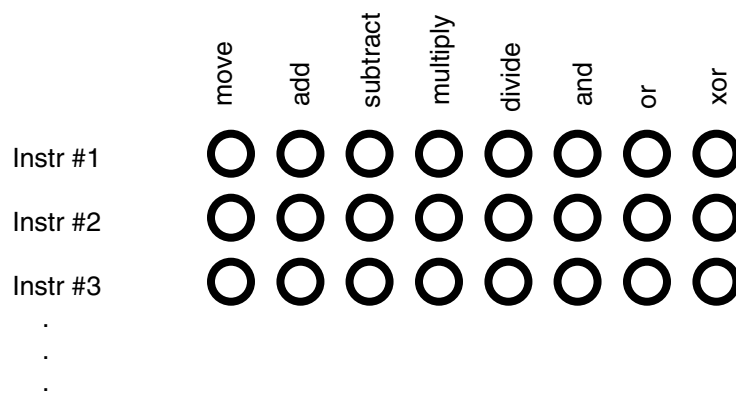
Intel, sensing an opportunity here, began developing CPUs with additional instruction targetted at specific applications. For example, the Pentium III introduced the SIMD (pronounced SIM-DEE) instructions that did for floating point calculations what the MMX instructions did for integer applications. Intel also hired lots of software engineers and began funding research into topic areas like speech recognition and (visual) pattern recognition in order to drive the new technologies that would require the new instructions their Pentium IV and later processors would offer. As this is being written, Intel is busy developing new uses for their specialized instructions so that system designers and software developers continue to use the 80x86 (and, perhaps, IA-64) family chips.

However, this discussion of fancy instruction sets is getting way ahead of the game. Let's take a long step back to the original 8086 chip and take a look at how system designers put a CPU together.

## 4.4 Basic CPU Design

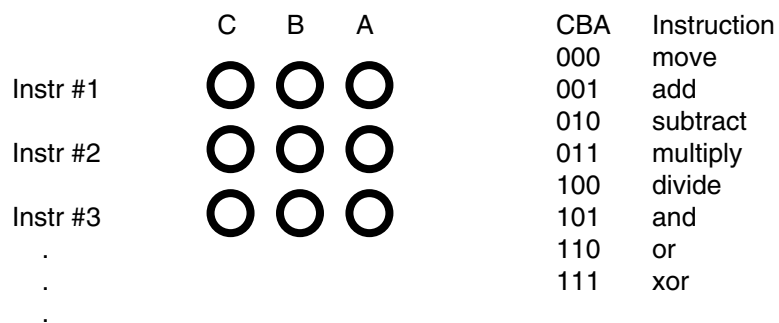
A fair question to ask at this point is “How exactly does a CPU perform assigned chores?” This is accomplished by giving the CPU a fixed set of commands, or instructions, to work on. Keep in mind that CPU designers construct these processors using logic gates to execute these instructions. To keep the number of logic gates to a reasonably small set CPU designers must necessarily restrict the number and complexity of the commands the CPU recognizes. This small set of commands is the CPU’s instruction set.

Programs in early (pre-Von Neumann) computer systems were often “hard-wired” into the circuitry. That is, the computer’s wiring determined what problem the computer would solve. One had to rewire the circuitry in order to change the program. A very difficult task. The next advance in computer design was the programmable computer system, one that allowed a computer programmer to easily “rewire” the computer system using a sequence of sockets and plug wires. A computer program consisted of a set of rows of holes (sockets), each row representing one operation during the execution of the program. The programmer could select one of several instructions by plugging a wire into the particular socket for the desired instruction (see Figure 4.1).



**Figure 4.1 Patch Panel Programming**

Of course, a major difficulty with this scheme is that the number of possible instructions is severely limited by the number of sockets one could physically place on each row. However, CPU designers quickly discovered that with a small amount of additional logic circuitry, they could reduce the number of sockets required from  $n$  holes for  $n$  instructions to  $\log_2(n)$  holes for  $n$  instructions. They did this by assigning a numeric code to each instruction and then encode that instruction as a binary number using  $\log_2(n)$  holes (see Figure 4.2).



**Figure 4.2 Encoding Instructions**

This addition requires eight logic functions to decode the A, B, and C bits from the patch panel, but the extra circuitry is well worth the cost because it reduces the number of sockets that must be repeated for each instruction (this circuitry, by the way, is nothing more than a single three-line to eight-line decoder).

Of course, many CPU instructions are not stand-alone. For example, the move instruction is a command that moves data from one location in the computer to another (e.g., from one register to another). Therefore, the move instruction requires two operands: a source operand and a destination operand. The CPU’s designer usually encodes these source and destination operands as part of the machine instruction, certain sockets correspond to the source operand and certain sockets correspond to the destination operand. Figure 4.3 shows one possible combination of sockets to handle this. The move instruction would move data from the source register to the destination register, the add instruction would add the value of the source register to the destination register, etc.

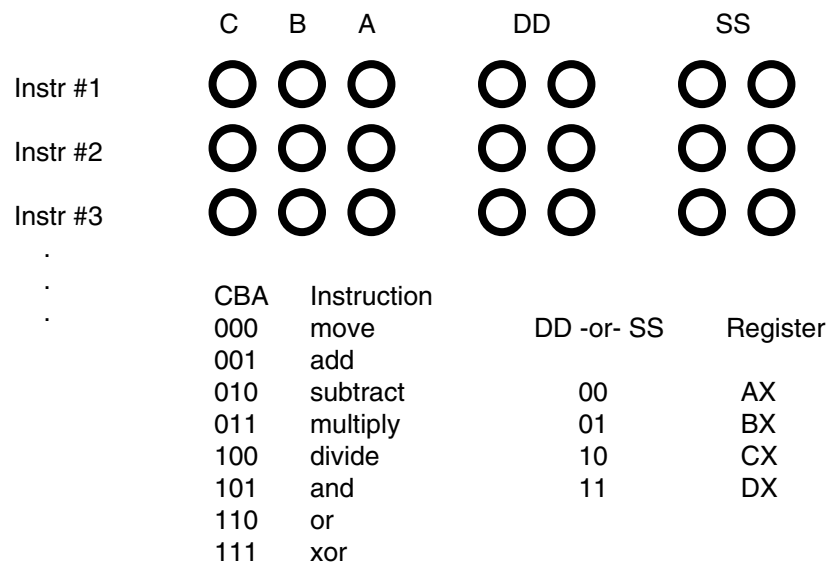


Figure 4.3      Encoding Instructions with Source and Destination Fields

One of the primary advances in computer design that the VNA provides is the concept of a stored program. One big problem with the patch panel programming method is that the number of program steps (machine instructions) is limited by the number of rows of sockets available on the machine. John Von Neumann and others recognized a relationship between the sockets on the patch panel and bits in memory; they figured they could store the binary equivalents of a machine program in main memory and fetch each program from memory, load it into a special decoding register that connected directly to the instruction decoding circuitry of the CPU.

The trick, of course, was to add yet more circuitry to the CPU. This circuitry, the control unit (CU), fetches instruction codes (also known as operation codes or opcodes) from memory and moves them to the instruction decoding register. The control unit contains a special register, the instruction pointer that contains the address of an executable instruction. The control unit fetches this instruction’s opcode from memory and places it in the decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution, and so on.

When designing an instruction set, the CPU’s designers generally choose opcodes that are a multiple of eight bits long so the CPU can easily fetch complete instructions from memory. The goal of the CPU’s designer is to assign an appropriate number of bits to the instruction class field (move, add, subtract, etc.) and to the operand fields. Choosing more bits for the instruction field lets you have more instructions, choosing additional bits for the operand fields lets you select a larger number of operands (e.g., memory locations or registers). There are additional complications. Some instructions have only one operand or, perhaps, they don’t have any operands at all. Rather than waste the bits associated with these fields, the CPU designers

often reuse these fields to encode additional opcodes, once again with some additional circuitry. The Intel 80x86 CPU family takes this to an extreme with instructions ranging from one to almost 15 bytes long<sup>8</sup>.

## 4.5 Decoding and Executing Instructions: Random Logic Versus Microcode

Once the control unit fetches an instruction from memory, you may wonder "exactly how does the CPU execute this instruction?" In traditional CPU design there have been two common approaches: hardwired logic and emulation. The 80x86 family uses both of these techniques.

A hardwired, or *random logic*<sup>9</sup>, approach uses decoders, latches, counters, and other logic devices to move data around and operate on that data. The microcode approach uses a very fast but simple internal processor that uses the CPU's opcodes as an index into a table of operations (the *microcode*) and executes a sequence of *microinstructions* that do the work of the *macroinstruction* (i.e., the CPU instruction) they are emulating.

The random logic approach has the advantage that it is possible to devise faster CPUs if typical CPU speeds are faster than typical memory speeds (a situation that has been true for quite some time). The drawback to random logic is that it is difficult to design CPUs with large and complex instruction sets using a random logic approach. The logic to execute the instructions winds up requiring large percentage of the chip's real estate and it becomes difficult to properly lay out the logic so that related circuits are close to one another in the two-dimensional space of the chip,

CPUs based on microcode contain a small, very fast, execution unit that fetches instructions from the microcode bank (which is really nothing more than fast ROM on the CPU chip). This microcode executes one microinstruction per clock cycle and a sequence of microinstructions decode the instruction, fetch its operands, move the operands to appropriate functional units that do whatever calculations are necessary, store away necessary results, and then update appropriate registers and flags in anticipation of the next instruction.

The microcode approach may appear to be substantially slower than the random logic approach because of all the steps involved. Actually, this isn't necessarily true. Keep in mind that with a random logic approach to instruction execution, part of the random logic is often a sequencer that steps through several states (one state per clock cycle). Whether you use your clock cycles executing microinstructions or stepping through a random logic state machine, you're still burning up clock cycles.

One advantage of microcode is that it makes better reuse of existing silicon on the CPU. Many CPU instructions (macroinstructions) execute some of the same microinstructions as many other instructions. This allows the CPU designer to use microcode subroutines to implement many common operations, thus saving silicon on the CPU. While it is certainly possible to share circuitry in a random logic device, this is often difficult if two circuits could otherwise share some logic but are across the chip from one another.

Another advantage of microcode is that it lets you create some very complex instructions that consist of several different operations. This provides programmers (especially assembly language programmers) with the ability to do more work with fewer instructions in their programs. In theory, this lets them write faster programs since they now execute half as many instructions, each doing twice the work of a simpler instruction set (the 80x86 MMX instruction set extension is a good example of this theory in action, although the MMX instructions do not use a microcode implementation).

Microcode does suffer from one disadvantage compared to random logic: the speed of the processor is tied to the speed of the internal microcode execution unit. Although the "microengine" itself is usually quite fast, the microengine must fetch its instruction from the microcode ROM. Therefore, if memory technology is slower than the execution logic, the microcode ROM will slow the microengine down because the system will have to introduce wait states into the microcode ROM access. Actually, microengines generally don't support the use of wait states, so this means that the microengine will have to run at the same speed as the

8. Though this is, by no means, the most complex instruction set. The VAX, for example, has instructions up to 150 bytes long!

9. There is actually nothing random about this logic at all. This design technique gets its name from the fact that if you view a photomicrograph of a CPU die that uses microcode, the microcode section looks very regular; the same photograph of a CPU that utilizes random logic contains no such easily discernable patterns.

microcode ROM. This effectively limits the speed at which the microengine, and therefore the CPU, can run.

Which approach is better for CPU design? That depends entirely on the current state of memory technology. If memory technology is faster than CPU technology, then the microcode approach tends to make more sense. If memory technology is slower than CPU technology, then random logic tends to produce the faster CPUs.

When Intel first began designing the 8086 CPU sometime between 1976 and 1978, memory technology was faster so they used microcode. Today, CPU technology is much faster than memory technology, so random logic CPUs tend to be faster. Most modern (non-x86) processors use random logic. The 80x86 family uses a combination of these technologies to improve performance while maintaining compatibility with the complex instruction set that relied on microcode way back in 1978.

---

## 4.6 RISC vs. CISC vs. VLIW

In the 1970's, CPU designers were busy extending their instruction sets to make their chips easier to program. It was very common to find a CPU designer poring over the assembly output of some high level language compiler searching for common two and three instruction sequences the compiler would emit. The designer would then create a single instruction that did the work of this two or three instruction sequence, the compiler writer would modify the compiler to use this new instruction, and a recompilation of the program would, presumably, produce a faster and shorter program than before.

Digital Equipment Corporation (now part of Compaq Computer) raised this process to a new level in their VAX minicomputer series. It is not surprising, therefore, that many research papers appearing in the 1980's would commonly use the VAX as an example of what not to do.

The problem is, these designers lost track of what they were trying to do, or to use the old cliché, they couldn't see the forest for the trees. They assumed that there were making their processors faster by executing a single instruction that previously required two or more. They also assumed that they were making the programs smaller, for exactly the same reason. They also assumed that they were making the processors easier to program because programmers (or compilers) could write a single instruction instead of using multiple instructions. In many cases, they assumed wrong.

In the early 80's, researchers at IBM and several institutions like Stanford and UC Berkeley challenged the assumptions of these designers. They wrote several papers showing how complex instructions on the VAX minicomputer could actually be done faster (and sometimes in less space) using a sequence of simpler instructions. As a result, most compiler writers did not use the fancy new instructions on the VAX (nor did assembly language programmers). Some might argue that having an unused instruction doesn't hurt anything, but these researchers argued otherwise. They claimed that any unnecessary instructions required additional logic to implement and as the complexity of the logic grows it becomes more and more difficult to produce a high clock speed CPU.

This research led to the development of the RISC, or Reduced Instruction Set Computer, CPU. The basic idea behind RISC was to go in the opposite direction of the VAX. Decide what the smallest reasonable instruction set could be and implement that. By throwing out all the complex instructions, RISC CPU designers could use random logic rather than microcode (by this time, CPU speeds were outpacing memory speeds). Rather than making an individual instruction more complex, they could move the complexity to the system level and add many on-chip features to improve the overall system performance (like caches, pipelines, and other advanced mainframe features of the time). Thus, the great "RISC vs. CISC<sup>10</sup>" debate was born.

Before commenting further on the result of this debate, you should realize that RISC actually means "(Reduced Instruction) Set Computer," not "Reduced (Instruction Set) Computer." That is, the goal of RISC was to reduce the complexity of individual instructions, not necessarily reduce the number of instructions a RISC CPU supports. It was often the case that RISC CPUs had fewer instructions than their CISC counter-

---

10. CISC stands for Complex Instruction Set Computer and defines those CPUs that were popular at the time like the VAX and the 80x86.

parts, but this was not a precondition for calling a CPU a RISC device. Many RISC CPUs had more instructions than some of their CISC contemporaries, depending on how you count instructions.

First, there is no debate about one thing: if you have two CPUs, one RISC and one CISC and they both run at the same clock frequency and they execute the same average number of instructions per clock cycle, CISC is the clear winner. Since CISC processors do more work with each instruction, if the two CPUs execute the same number of instructions in the same amount of time, the CISC processor usually gets more work done.

However, RISC performance claims were based on the fact that RISC's simpler design would allow the CPU designers to reduce the overall complexity of the chip, thereby allowing it to run at a higher clock frequency. Further, with a little added complexity, they could easily execute more instructions per clock cycle, on the average, than their CISC contemporaries.

One drawback to RISC CPUs is that their code density was much lower than CISC CPUs. Although memory devices were dropping in price and the need to keep programs small was decreasing, low code density requires larger caches to maintain the same number of instructions in the cache. Further, since memory speeds were not keeping up with CPU speeds, the larger instruction sizes found in the RISC CPUs meant that the system spent more time bringing in those instructions from memory to cache since they could transfer fewer instructions per bus transaction. For many years, CPU architects argued to and fro about whether RISC or CISC was the better approach. With one big footnote, the RISC approach has generally won the argument. Most of the popular CISC systems, e.g., the VAX, the Z8000, the 16032/32016, and the 68000, have quietly faded away to be replaced by the likes of the PowerPC, the MIPS CPUs, the Alpha, and the SPARC. The one footnote here is, of course, the 80x86 family. Intel has proven that if you really want to keep extending a CISC architecture, and you're willing to throw a lot of money at it, you can extend it far greater than anyone ever expected. As of late 2000/early 2001 the 80x86 is the raw performance leader. The CPU runs at a higher clock frequency than the competing RISC chips; it executes fairly close to the same number of instructions per clock cycle as the competing RISC chips; it has about the same "average instruction size to cache size" ratio as the RISC chips; and it is a CISC, so many of the instructions do more work than their RISC equivalents. So overall, the 80x86 is, on the average, faster than contemporary RISC chips<sup>11</sup>.

To achieve this raw performance advantage, the 80x86 has borrowed heavily from RISC research. Intel has divided the instruction set into a set of simple instructions that Intel calls the "RISC core" and the remaining, complex instructions. The complex instructions do not execute as rapidly as the RISC core instructions. In fact, it is often the case that the task of a complex instruction can be accomplished faster using multiple RISC core instructions. Intel supports the complex instructions to provide full compatibility with older software, but compiler writers and assembly language programmers tend to avoid the use of these instructions. Note that Intel moves instructions between these two sets over time. As Intel improves the processor they tend to speed up some of the slower, complex, instructions. Therefore, it is not possible to give a static list of instructions you should avoid; instead, you will have to refer to Intel's documentation for the specific processor you use.

Later Pentium processors do not use an interpretive engine and microcode like the earlier 80x86 processors. Instead, the Pentium core processors execute a set of "micro-operations" (or "micro-ops"). The Pentium processors translate the 80x86 instruction set into a sequence of micro-ops on the fly. The RISC core instructions typically generate a single micro-op while the CISC instructions generate a sequence of two or more micro-ops. For the purposes of determining the performance of a section of code, we can treat each micro-op as a single instruction. Therefore, the CISC instructions are really nothing more than "macro-instructions" that the CPU automatically translates into a sequence of simpler instructions. This is the reason the complex instructions take longer to execute.

Unfortunately, as the x86 nears its 25<sup>th</sup> birthday, it's clear (to Intel, at least) that it's been pushed to its limits. This is why Intel is working with HP to base their IA-64 architecture on the PA-RISC instruction set. The IA-64 architecture is an interesting blend. On the one hand, it (supposedly) supports object-code compatibility with the previous generation x86 family (though at reduced performance levels). Obviously, it's a

---

11. Note, by the way, that this doesn't imply that 80x86 systems are faster than computer systems built around RISC chips. Many RISC systems gain additional speed by supporting multiple processors better than the x86 or by having faster bus throughput. This is one reason, for example, why Internet companies select Sun equipment for their web servers.

RISC architecture since it was originally based on Hewlett-Packard's PA-RISC (PA=Precision Architecture) design. However, Intel and HP have extended on the RISC design by using another technology: Very Long Instruction Word (VLIW) computing. The idea behind VLIW computing is to use a very long opcode that handle multiple operations in parallel. In some respects, this is similar to CISC computing since a single VLIW "instruction" can do some very complex things. However, unlike CISC instructions, a VLIW instruction word can actually complete several independent tasks simultaneously. Effectively, this allows the CPU to execute some number of instructions in parallel.

Intel's VLIW approach is risky. To succeed, they are depending on compiler technology that doesn't yet exist. They made this same mistake with the iAPX 432. It remains to be seen whether history is about to repeat itself or if Intel has a winner on their hands.

---

## 4.7 Instruction Execution, Step-By-Step

To understand the problems with developing an efficient CPU, let's consider four representative 80x86 instructions: MOV, ADD, LOOP, and JNZ (jump if not zero). These instructions will allow us to explore many of the issues facing the x86 CPU designer.

You've seen the MOV and ADD instructions in previous chapters so there is no need to review them here. The LOOP and JNZ instructions are new, so it's probably a good idea to explain what they do before proceeding. Both of these instructions are *conditional jump* instructions. A conditional jump instruction tests some condition and jumps to some other instruction in memory if the condition is true and they fall through to the next instruction if the condition is false. This is basically the opposite of HLA's IF statement (which falls through if the condition is true and jumps to the else section if the condition is false). The JNZ (jump if not zero) instruction tests the CPU's zero flag and transfers control to some target location if the zero flag contains zero; JNZ falls through to the next instruction if the zero flag contains one. The program specifies the target instruction to jump to by specifying the distance from the JNZ instruction to the target instruction as a small signed integer (for our purposes here, we'll assume that the distance is within the range  $\pm 128$  bytes so the instruction uses a single byte to specify the distance to the target location).

The last instruction of interest to us here is the LOOP instruction. The LOOP instruction decrements the value of the ECX register and transfers control to a target instruction within  $\pm 128$  bytes if ECX does not contain zero (after the decrement). This is a good example of a CISC instruction since it does multiple operations: (1) it subtracts one from ECX and then it (2) does a conditional jump if ECX does not contain zero. That is, LOOP is equivalent to the following two 80x86 instructions<sup>12</sup>:

```
loop SomeLabel;
```

-is roughly equivalent to-

```
dec( ecx );
jnz SomeLabel;
```

Note that *SomeLabel* specifies the address of the target instruction that must be within about  $\pm 128$  bytes of the LOOP or JNZ instructions above. The LOOP instruction is a good example of a complex (vs. RISC core) instruction on the Pentium processors. It is actually faster to execute a DEC and a JNZ instruction<sup>13</sup> than it is to execute a LOOP instruction. In this section we will not concern ourselves with this issue; we will assume that the LOOP instruction operates as though it were a RISC core instruction.

The 80x86 CPUs do not execute instructions in a single clock cycle. For example, the MOV instruction (which is relatively simple) could use the following execution steps<sup>14</sup>:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.

---

12. This sequence is not exactly equivalent to LOOP since this sequence affects the flags while LOOP does not.

13. Actually, you'll see a little later that there is a *decrement* instruction you can use to subtract one from ECX. The decrement instruction is better because it is shorter.

14. It is not possible to state exactly what steps each CPU requires since many CPUs are different from one another.

- If required, fetch a 16-bit instruction operand from memory.
- If required, update EIP to point beyond the operand.
- If required, compute the address of the operand (e.g., EBX+disp) .
- Fetch the operand.
- Store the fetched value into the destination register

If we allocate one clock cycle for each of the above steps, an instruction could take as many as eight clock cycles to complete (note that three of the steps above are optional, depending on the MOV instruction's addressing mode, so a simple MOV instruction could complete in as few as five clock cycles).

The ADD instruction is a little more complex. Here's a typical set of operations the ADD( reg, reg) instruction must complete:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Get the value of the source operand and send it to the ALU.
- Fetch the value of the destination operand (a register) and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the first register operand.
- Update the flags register with the result of the addition operation.

If the source operand is a memory location, the operation is slightly more complicated:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- If required, fetch a displacement for use in the effective address calculation
- If required, update EIP to point beyond the displacement value.
- Get the value of the source operand from memory and send it to the ALU.
- Fetch the value of the destination operand (a register) and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the register operand.
- Update the flags register with the result of the addition operation.

ADD( const, memory) is the messiest of all, this code sequence looks something like the following:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- If required, fetch a displacement for use in the effective address calculation
- If required, update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Update EIP to point beyond the constant's value (at the next instruction in memory).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand.
- Update the flags register with the result of the addition operation.

Note that there are other forms of the ADD instruction requiring their own special processing. These are just representative examples. As you see in these examples, the ADD instruction could take as many as ten steps (or cycles) to complete. Note that this is one advantage of a RISC design. Most RISC design have only one or two forms of the ADD instruction (that add registers together and, perhaps, that add constants to registers). Since register to register adds are often the fastest (and constant to register adds are probably the second fastest), the RISC CPUs force you to use the fastest forms of these instructions.

The JNZ instruction might use the following sequence of steps:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.

- Fetch a displacement byte to determine the jump distance send this to the ALU
- Update EIP to point at the next byte.
- Test the zero flag to see if it is clear.
- If the zero flag was clear, copy the EIP register to the ALU.
- If the zero flag was clear, instruct the ALU to add the displacement and EIP register values.
- If the zero flag was clear, copy the result of the addition above back to the EIP register.

Notice how the JNZ instruction requires fewer steps if the jump is not taken. This is very typical for conditional jump instructions. If each step above corresponds to one clock cycle, the JNZ instruction would take six or nine clock cycles, depending on whether the branch is taken. Because the 80x86 JNZ instruction does not allow different types of operands, there is only one sequence of steps needed for this application.

The 80x86 LOOP instruction might use an execution sequence like the following:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch the value of the ECX register and send it to the ALU.
- Instruct the ALU to decrement the value.
- Send the result back to the ECX register. Set a special internal flag if this value is non-zero.
- Fetch a displacement byte to determine the jump distance send this to the ALU
- Update EIP to point at the next byte.
- Test the special flag to see if ECX was non-zero.
- If the flag was set, copy the EIP register to the ALU.
- If the flag was set, instruct the ALU to add the displacement and EIP register values.
- If the flag was set, copy the result of the addition above back to the EIP register.

Although a given 80x86 CPU might not execute the steps for the instructions above, they all execute some sequence of operations. Each operation requires a finite amount of time to execute (generally, one clock cycle per operation or *stage* as we usually refer to each of the above steps). Obviously, the more steps needed for an instruction, the slower it will run. This is why complex instructions generally run slower than simple instructions, complex instructions usually have lots of execution stages.

---

## 4.8 Parallelism – the Key to Faster Processors

An early goal of the RISC processors was to execute one instruction per clock cycle, on the average. However, even if a RISC instruction is simplified, the actual execution of the instruction still requires multiple steps. So how could they achieve this goal? And how do later members the 80x86 family with their complex instruction sets also achieve this goal? The answer is parallelism.

Consider the following steps for a MOV( reg, reg) instruction:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- Fetch the source register.
- Store the fetched value into the destination register

There are five stages in the execution of this instruction with certain dependencies between each stage. For example, the CPU must fetch the instruction byte from memory before it updates EIP to point at the next byte in memory. Likewise, the CPU must decode the instruction before it can fetch the source register (since it doesn't know it needs to fetch a source register until it decodes the instruction). As a final example, the CPU must fetch the source register before it can store the fetched value in the destination register.

Most of the stages in the execution of this MOV instruction are *serial*. That is, the CPU must execute one stage before proceeding to the next. The one exception is the "Update EIP" step. Although this stage must follow the first stage, none of the following stages in the instruction depend upon this step. Therefore, this could be the third, fourth, or fifth step in the calculation and it wouldn't affect the outcome of the instruc-

tion. Further, we could execute this step concurrently with any of the other steps and it wouldn't affect the operation of the MOV instruction, e.g.,

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does.
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

By doing two of the stages in parallel, we can reduce the execution time of this instruction by one clock cycle. Although the remaining stages in the "mov( reg, reg );" instruction must remain serialized (that is, they must take place in exactly this order), other forms of the MOV instruction offer similar opportunities to overlapped portions of their execution to save some cycles. For example, consider the "mov( [ebx+disp], eax );" instruction:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- Fetch a displacement operand from memory.
- Update EIP to point beyond the displacement.
- Compute the address of the operand (e.g., EBX+disp) .
- Fetch the operand.
- Store the fetched value into the destination register

Once again there is the opportunity to overlap the execution of several stages in this instruction, for example:

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does and update the EIP register to point at the next byte.
- Fetch a displacement operand from memory.
- Compute the address of the operand (e.g., EBX+disp) and update EIP to point beyond the displacement..
- Fetch the operand.
- Store the fetched value into the destination register

In this example, we reduced the number of execution steps from eight to six by overlapping the update of EIP with two other operations.

As a last example, consider the "add( const, [ebx+disp] );" instruction (the instruction with the largest number of steps we've considered thus far). It's non-overlapped execution looks like this:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Compute the address of the memory operand (EBX+disp).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand.
- Update the flags register with the result of the addition operation.
- Update EIP to point beyond the constant's value (at the next instruction in memory).

We can overlap at least three steps in this instruction by noting that certain stages don't depend on the result of their immediate predecessor

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Compute the address of the memory operand (EBX+disp).

- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

Note that we could not merge one of the "Update EIP" operations because the previous stage and following stages of the instruction both use the value of EIP before and after the update.

Unlike the MOV instruction, the steps in the ADD instruction above are not all dependent upon the previous stage in the instruction's execution. For example, the sequence above fetches the constant from memory and then computes the effective address (EBX+disp) of the memory operand. Neither operation depends upon the other, so we could easily swap their positions above to yield the following:

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Compute the address of the memory operand (EBX+disp).
- Fetch the constant value from memory and send it to the ALU.
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

This doesn't save any steps, but it does reduce some dependencies between certain stages and their immediate predecessors, allowing additional parallel operation. For example, we can now merge the "Update EIP" operation with the effective address calculation:

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Compute the address of the memory operand (EBX+disp) and update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

Although it might seem possible to fetch the constant and the memory operand in the same step (since their values do not depend upon one another), the CPU can't actually do this (yet!) because it has only a single data bus. Since both of these values are coming from memory, we can't bring them into the CPU during the same step because the CPU uses the data bus to fetch these two values. In the next section you'll see how we can overcome this problem.

By overlapping various stages in the execution of these instructions we've been able to substantially reduce the number of steps (i.e., clock cycles) that the instructions need to complete execution. This process of executing various steps of the instruction in parallel with other steps is a major key to improving CPU performance without cranking up the clock speed on the chip. In this section we've seen how to speed up the execution of an instruction by doing many of the internal operations of that instruction in parallel. However, there's only so much to be gained from this approach. In this approach, the instructions themselves are still serialized (one instruction completes before the next instruction begins execution). Starting with the next section we'll start to see how to overlap the execution of adjacent instructions in order to save additional cycles.

### 4.8.1 The Prefetch Queue – Using Unused Bus Cycles

The key to improving the speed of a processor is to perform operations in parallel. If we were able to do two operations on each clock cycle, the CPU would execute instructions twice as fast when running at the same clock speed. However, simply deciding to execute two operations per clock cycle is not so easy. Many steps in the execution of an instruction share *functional units* in the CPU (functional units are groups of logic that perform a common operation, e.g., the ALU and the CU). A functional unit is only capable of one operation at a time. Therefore, you cannot do two operations that use the same functional unit concurrently (e.g., incrementing the EIP register and adding two values together). Another difficulty with doing certain operations concurrently is that one operation may depend on the other's result. For example, the two steps of the ADD instruction that involve adding two values and then storing their sum. You cannot store the sum into a register until after you've computed the sum. There are also some other resources the CPU cannot share between steps in an instruction. For example, there is only one data bus; the CPU cannot fetch an instruction opcode at the same time it is trying to store some data to memory. The trick in designing a CPU that executes several steps in parallel is to arrange those steps to reduce conflicts or add additional logic so the two (or more) operations can occur simultaneously by executing in different functional units.

Consider again the steps the MOV( mem/reg, reg ) instruction requires:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a displacement operand from memory.
- If required, update EIP to point beyond the displacement.
- Compute the address of the operand, if required (i.e., EBX+xxxx) .
- Fetch the operand.
- Store the fetched value into the destination register

The first operation uses the value of the EIP register (so we cannot overlap incrementing EIP with it) and it uses the bus to fetch the instruction opcode from memory. Every step that follows this one depends upon the opcode it fetches from memory, so it is unlikely we will be able to overlap the execution of this step with any other.

The second and third operations do not share any functional units, nor does decoding an opcode depend upon the value of the EIP register. Therefore, we can easily modify the control unit so that it increments the EIP register at the same time it decodes the instruction. This will shave one cycle off the execution of the MOV instruction.

The third and fourth operations above (decoding and optionally fetching the displacement operand) do not look like they can be done in parallel since you must decode the instruction to determine if it the CPU needs to fetch an operand from memory. However, we could design the CPU to go ahead and fetch the operand anyway, so that it's available if we need it. There is one problem with this idea, though, we must have the address of the operand to fetch (the value in the EIP register) and if we must wait until we are done incrementing the EIP register before fetching this operand. If we are incrementing EIP at the same time we're decoding the instruction, we will have to wait until the next cycle to fetch this operand.

Since the next three steps are optional, there are several possible instruction sequences at this point:

- #1 (step 4, step 5, step 6, and step 7) – e.g., MOV( [ebx+1000], eax )
- #2 (step 4, step 5, and step 7) – e.g., MOV( disp, eax ) -- assume disp's address is 1000
- #3 (step 6 and step 7) – e.g., MOV( [ebx], eax )
- #4 (step 7) – e.g., MOV( ebx, eax )

In the sequences above, step seven always relies on the previous steps in the sequence. Therefore, step seven cannot execute in parallel with any of the other steps. Step six also relies upon step four. Step five cannot execute in parallel with step four since step four uses the value in the EIP register, however, step five can execute in parallel with any other step. Therefore, we can shave one cycle off the first two sequences above as follows:

- #1 (step 4, step 5/6, and step 7)
- #2 (step 4, step 5/7)
- #3 (step 6 and step 7)
- #4 (step 7)

Of course, there is no way to overlap the execution of steps seven and eight in the MOV instruction since it must surely fetch the value before storing it away. By combining these steps, we obtain the following steps for the MOV instruction:

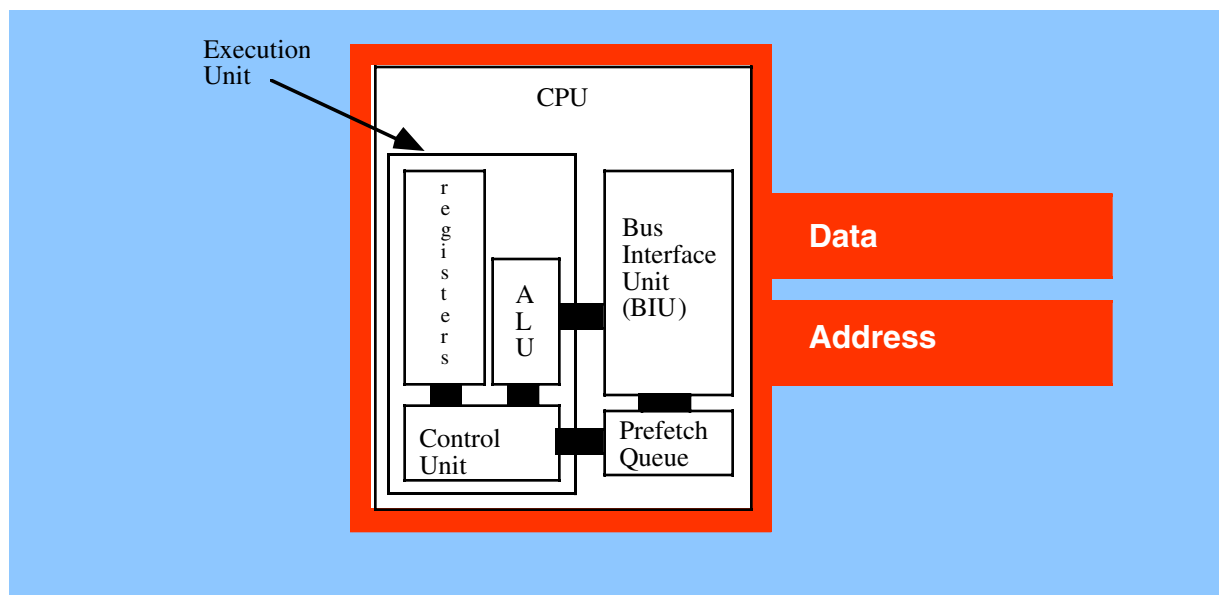
- Fetch the instruction byte from memory.
- Decode the instruction and update ip
- If required, fetch a displacement operand from memory.
- Compute the address of the operand, if required (i.e.,  $ebx+xxxx$ ).
- Fetch the operand, if required update EIP to point beyond  $xxxx$ .
- Store the fetched value into the destination register

By adding a small amount of logic to the CPU, we've shaved one or two cycles off the execution of the MOV instruction. This simple optimization works with most of the other instructions as well.

Consider what happens with the MOV instruction above executes on a CPU with a 32-bit data bus. If the MOV instruction fetches an eight-bit displacement from memory, the CPU may actually wind up fetching the following three bytes after the displacement along with the displacement value (since the 32-bit data bus lets us fetch four bytes in a single bus cycle). The second byte on the data bus is actually the opcode of the next instruction. If we could save this opcode until the execution of the next instruction, we could shave a cycle of its execution time since it would not have to fetch the opcode byte. Furthermore, since the instruction decoder is idle while the CPU is executing the MOV instruction, we can actually decode the next instruction while the current instruction is executing, thereby shaving yet another cycle off the execution of the next instruction. This, effectively, overlaps a portion of the MOV instruction with the beginning of the execution of the next instruction, allowing additional parallelism.

Can we improve on this? The answer is yes. Note that during the execution of the MOV instruction the CPU is not accessing memory on every clock cycle. For example, while storing the data into the destination register the bus is idle. During time periods when the bus is idle we can pre-fetch instruction opcodes and operands and save these values for executing the next instruction.

The hardware to do this is the prefetch queue. Figure 4.4 shows the internal organization of a CPU with a prefetch queue. The Bus Interface Unit, as its name implies, is responsible for controlling access to the address and data busses. Whenever some component inside the CPU wishes to access main memory, it sends this request to the bus interface unit (or BIU) that acts as a "traffic cop" and handles simultaneous requests for bus access by different modules (e.g., the execution unit and the prefetch queue).



**Figure 4.4 CPU Design with a Prefetch Queue**

Whenever the execution unit is not using the Bus Interface Unit, the BIU can fetch additional bytes from the instruction stream. Whenever the CPU needs an instruction or operand byte, it grabs the next available byte from the prefetch queue. Since the BIU grabs four bytes at a time from memory (assuming a 32-bit data bus) and it generally consumes fewer than four bytes per clock cycle, any bytes the CPU would normally fetch from the instruction stream will already be sitting in the prefetch queue.

Note, however, that we're not guaranteed that all instructions and operands will be sitting in the prefetch queue when we need them. For example, consider the "JNZ Label;" instruction, if it transfers control to *Label*, will invalidate the contents of the prefetch queue. If this instruction appears at locations 400 and 401 in memory (it is a two-byte instruction), the prefetch queue will contain the bytes at addresses 402, 403, 404, 405, 406, 407, etc. If the target address of the JNZ instruction is 480, the bytes at addresses 402, 403, 404, etc., won't do us any good. So the system has to pause for a moment to fetch the double word at address 480 before it can go on.

Another improvement we can make is to overlap instruction decoding with the last step of the previous instruction. After the CPU processes the operand, the next available byte in the prefetch queue is an opcode, and the CPU can decode it in anticipation of its execution. Of course, if the current instruction modifies the EIP register then any time spent decoding the next instruction goes to waste, but since this occurs in parallel with other operations, it does not slow down the system (though it does require extra circuitry to do this).

The instruction execution sequence now assumes that the following events occur in the background:  
CPU Prefetch Events:

- If the prefetch queue is not full (generally it can hold between eight and thirty-two bytes, depending on the processor) and the BIU is idle on the current clock cycle, fetch the next double word from memory at the address in EIP at the beginning of the clock cycle<sup>15</sup>.
- If the instruction decoder is idle and the current instruction does not require an instruction operand, begin decoding the opcode at the front of the prefetch queue (if present), otherwise begin decoding the byte beyond the current operand in the prefetch queue (if present). If the desired byte is not in the prefetch queue, do not execute this event.

15. This operation fetches only a byte if ip contains an odd value.

Now let's reconsider our "mov( reg, reg );" instruction from the previous section. With the addition of the prefetch queue and the bus interface unit, fetching and decoding opcode bytes, as well as updating the EIP register, takes place in parallel with the previous instruction. Without the BIU and the prefetch queue, the "mov( reg, reg );" requires the following steps:

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does.
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

However, now that we can overlap the instruction fetch and decode with the previous instruction, we now get the following steps:

- Fetch and Decode Instruction - overlapped with previous instruction
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

The instruction execution timings make a few optimistic assumptions, namely that the opcode is already present in the prefetch queue and that the CPU has already decoded it. If either cause is not true, additional cycles will be necessary so the system can fetch the opcode from memory and/or decode the instruction.

Because they invalidate the prefetch queue, jump and conditional jump instructions (when actually taken) are much slower than other instructions. This is because the CPU cannot overlap fetching and decoding the opcode for the next instruction with the execution of the jump instruction since the opcode is (probably) not in the prefetch queue. Therefore, it may take several cycles after the execution of one of these instructions for the prefetch queue to recover and the CPU is decoding opcodes in parallel with the execution of previous instructions. This has one very important implication to your programs: if you want to write fast code, make sure to avoid jumping around in your program as much as possible.

Note that the conditional jump instructions only invalidate the prefetch queue if they actually make the jump. If the condition is false, they fall through to the next instruction and continue to use the values in the prefetch queue as well as any pre-decoded instruction opcodes. Therefore, if you can determine, while writing the program, which condition is most likely (e.g., less than vs. not less than), you should arrange your program so that the most common case falls through and conditional jump rather than take the branch.

Instruction size (in bytes) can also affect the performance of the prefetch queue. The longer the instruction, the faster the CPU will empty the prefetch queue. Instructions involving constants and memory operands tend to be the largest. If you place a string of these in a row, the CPU may wind up having to wait because it is removing instructions from the prefetch queue faster than the BIU is copying data to the prefetch queue. Therefore, you should attempt to use shorter instructions whenever possible since they will improve the performance of the prefetch queue.

Usually, including the prefetch queue improves performance. That's why Intel provides the prefetch queue on every model of the 80x86, from the 8088 on up. On these processors, the BIU is constantly fetching data for the prefetch queue whenever the program is not actively reading or writing data.

Prefetch queues work best when you have a wide data bus. The 8086 processor runs much faster than the 8088 because it can keep the prefetch queue full with fewer bus accesses. Don't forget, the CPU needs to use the bus for other purposes than fetching opcodes, displacements, and immediate constants. Instructions that access memory compete with the prefetch queue for access to the bus (and, therefore, have priority). If you have a sequence of instructions that all access memory, the prefetch queue may become empty if there are only a few bus cycles available for filling the prefetch queue during the execution of these instructions. Of course, once the prefetch queue is empty, the CPU must wait for the BIU to fetch new opcodes from memory, slowing the program.

A wider data bus allows the BIU to pull in more prefetch queue data in the few bus cycles available for this purpose, so it is less likely the prefetch queue will ever empty out with a wider data bus. Executing shorter instructions also helps keep the prefetch queue full. The reason is that the prefetch queue has time to refill itself with the shorter instructions. Moral of the story: when programming a processor with a prefetch queue, always use the shortest instructions possible to accomplish a given task.

## 4.8.2 Pipelining – Overlapping the Execution of Multiple Instructions

Executing instructions in parallel using a bus interface unit and an execution unit is a special case of pipelining. The 80x86 family, starting with the 80486, incorporates pipelining to improve performance. With just a few exceptions, we'll see that pipelining allows us to execute one instruction per clock cycle.

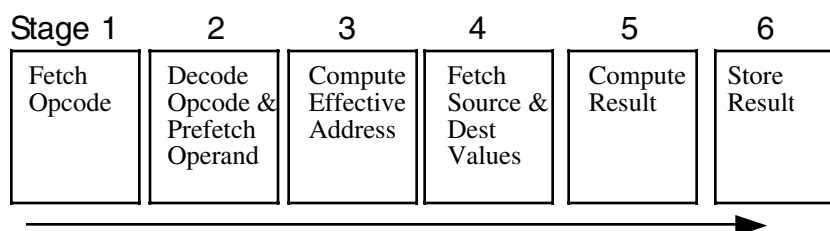
The advantage of the prefetch queue was that it let the CPU overlap instruction fetching and decoding with instruction execution. That is, while one instruction is executing, the BIU is fetching and decoding the next instruction. Assuming you're willing to add hardware, you can execute almost all operations in parallel. That is the idea behind pipelining.

### 4.8.2.1 A Typical Pipeline

Consider the steps necessary to do a generic operation:

- Fetch opcode.
- Decode opcode and (in parallel) prefetch a possible displacement or constant operand (or both)
- Compute complex addressing mode (e.g., [ebx+xxxx]), if applicable.
- Fetch the source value from memory (if a memory operand) and the destination register value (if applicable).
- Compute the result.
- Store result into destination register.

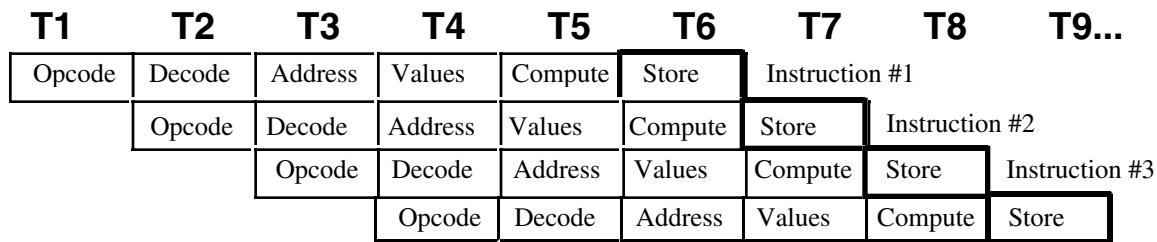
Assuming you're willing to pay for some extra silicon, you can build a little “mini-processor” to handle each of the above steps. The organization would look something like Figure 4.5.



**Figure 4.5 A Pipelined Implementation of Instruction Execution**

Note how we've combined some stages from the previous section. For example, in stage four of Figure 4.5 the CPU fetches the source and destination operands in the same step. You can do this by putting multiple data paths inside the CPU (e.g., from the registers to the ALU) and ensuring that no two operands ever compete for simultaneous use of the data bus (i.e., no memory-to-memory operations).

If you design a separate piece of hardware for each stage in the pipeline above, almost all these steps can take place in parallel. Of course, you cannot fetch and decode the opcode for more than one instruction at the same time, but you can fetch one opcode while decoding the previous instruction. If you have an n-stage pipeline, you will usually have n instructions executing concurrently.



**Figure 4.6 Instruction Execution in a Pipeline**

Figure 4.6 shows pipelining in operation. T1, T2, T3, etc., represent consecutive “ticks” of the system clock. At T=T1 the CPU fetches the opcode byte for the first instruction.

At T=T2, the CPU begins decoding the opcode for the first instruction. In parallel, it fetches a block of bytes from the prefetch queue in the event the instruction has an operand. Since the first instruction no longer needs the opcode fetching circuitry, the CPU instructs it to fetch the opcode of the second instruction in parallel with the decoding of the first instruction. Note there is a minor conflict here. The CPU is attempting to fetch the next byte from the prefetch queue for use as an operand, at the same time it is fetching operand data from the prefetch queue for use as an opcode. How can it do both at once? You’ll see the solution in a few moments.

At T=T3 the CPU computes an operand address for the first instruction, if any. The CPU does nothing on the first instruction if it does not use an addressing mode requiring such computation. During T3, the CPU also decodes the opcode of the second instruction and fetches any necessary operand. Finally the CPU also fetches the opcode for the third instruction. With each advancing tick of the clock, another step in the execution of each instruction in the pipeline completes, and the CPU fetches yet another instruction from memory.

This process continues until at T=T6 the CPU completes the execution of the first instruction, computes the result for the second, etc., and, finally, fetches the opcode for the sixth instruction in the pipeline. The important thing to see is that after T=T5 the CPU completes an instruction on every clock cycle. Once the CPU fills the pipeline, it completes one instruction on each cycle. Note that this is true even if there are complex addressing modes to be computed, memory operands to fetch, or other operations which use cycles on a non-pipelined processor. All you need to do is add more stages to the pipeline, and you can still effectively process each instruction in one clock cycle.

A bit earlier you saw a small conflict in the pipeline organization. At T=T2, for example, the CPU is attempting to prefetch a block of bytes for an operand and at the same time it is trying to fetch the next opcode byte. Until the CPU decodes the first instruction it doesn’t know how many operands the instruction requires nor does it know their length. However, the CPU needs to know this information to determine the length of the instruction so it knows what byte to fetch as the opcode of the next instruction. So how can the pipeline fetch an instruction opcode in parallel with an address operand?

One solution is to disallow this. If an instruction is an address or constant operand, we simply delay the start of the next instruction (this is known as a *hazard* as you shall soon see). Unfortunately, many instructions have these additional operands, so this approach will have a substantial negative impact on the execution speed of the CPU.

The second solution is to throw (a lot) more hardware at the problem. Operand and constant sizes usually come in one, two, and four-byte lengths. Therefore, if we actually fetch three bytes from memory, at offsets one, three, and five, beyond the current opcode we are decoding, we know that one of these bytes will probably contain the opcode of the next instruction. Once we are through decoding the current instruction we know how long it will be and, therefore, we know the offset of the next opcode. We can use a simple data selector circuit to choose which of the three opcode bytes we want to use.

In actual practice, we have to select the next opcode byte from more than three candidates because 80x86 instructions take many different lengths. For example, an instruction that moves a 32-bit constant to a memory location can be ten or more bytes long. And there are instruction lengths for nearly every value

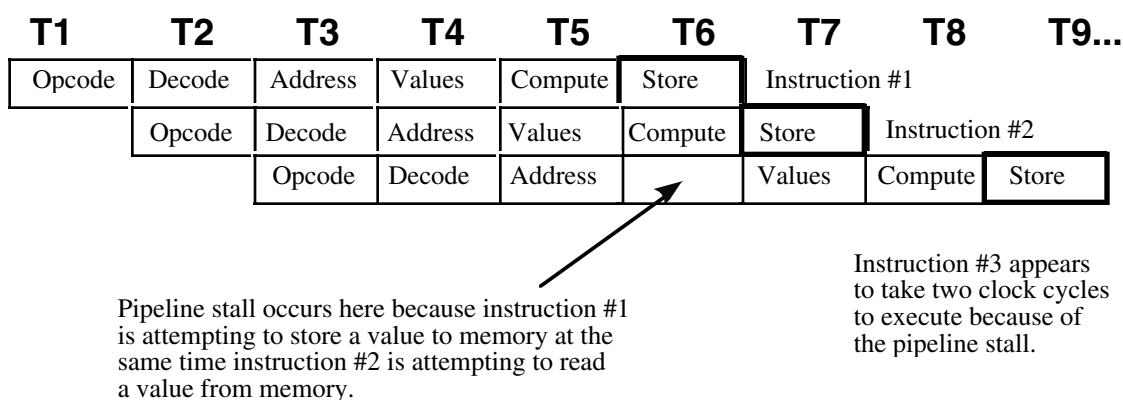
between one and fifteen bytes. Also, some opcodes on the 80x86 are longer than one byte, so the CPU may have to fetch multiple bytes in order to properly decode the current instruction. Nevertheless, by throwing more hardware at the problem we can decode the current opcode at the same time we're fetching the next.

### 4.8.2.2 Stalls in a Pipeline

Unfortunately, the scenario presented in the previous section is a little too simplistic. There are two drawbacks to that simple pipeline: bus contention among instructions and non-sequential program execution. Both problems may increase the average execution time of the instructions in the pipeline.

Bus contention occurs whenever an instruction needs to access some item in memory. For example, if a "mov( reg, mem);" instruction needs to store data in memory and a "mov( mem, reg);" instruction is reading data from memory, contention for the address and data bus may develop since the CPU will be trying to simultaneously fetch data and write data in memory.

One simplistic way to handle bus contention is through a *pipeline stall*. The CPU, when faced with contention for the bus, gives priority to the instruction furthest along in the pipeline. The CPU suspends fetching opcodes until the current instruction fetches (or stores) its operand. This causes the new instruction in the pipeline to take two cycles to execute rather than one (see Figure 4.7).



**Figure 4.7 A Pipeline Stall**

This example is but one case of bus contention. There are many others. For example, as noted earlier, fetching instruction operands requires access to the prefetch queue at the same time the CPU needs to fetch an opcode. Given the simple scheme above, it's unlikely that most instructions would execute at one clock per instruction (CPI).

Fortunately, the intelligent use of a cache system can eliminate many pipeline stalls like the ones discussed above. The next section on caching will describe how this is done. However, it is not always possible, even with a cache, to avoid stalling the pipeline. What you cannot fix in hardware, you can take care of with software. If you avoid using memory, you can reduce bus contention and your programs will execute faster. Likewise, using shorter instructions also reduces bus contention and the possibility of a pipeline stall.

What happens when an instruction *modifies* the EIP register? This, of course, implies that the next set of instructions to execute do not immediately follow the instruction that modifies EIP. By the time the instruction

```
JNZ    Label;
```

completes execution (assuming the zero flag is clear so the branch is taken), we've already started five other instructions and we're only one clock cycle away from the completion of the first of these. Obviously, the CPU must not execute those instructions or it will compute improper results.

The only reasonable solution is to *flush* the entire pipeline and begin fetching opcodes anew. However, doing so causes a severe execution time penalty. It will take six clock cycles (the length of the pipeline in our examples) before the next instruction completes execution. Clearly, you should avoid the use of instructions which interrupt the sequential execution of a program. This also shows another problem – pipeline length. The longer the pipeline is, the more you can accomplish per cycle in the system. However, lengthening a pipeline may slow a program if it jumps around quite a bit. Unfortunately, you cannot control the number of stages in the pipeline<sup>16</sup>. You can, however, control the number of transfer instructions which appear in your programs. Obviously you should keep these to a minimum in a pipelined system.

---

### 4.8.3 Instruction Caches – Providing Multiple Paths to Memory

System designers can resolve many problems with bus contention through the intelligent use of the prefetch queue and the cache memory subsystem. They can design the prefetch queue to buffer up data from the instruction stream, and they can design the cache with separate data and code areas. Both techniques can improve system performance by eliminating some conflicts for the bus.

The prefetch queue simply acts as a buffer between the instruction stream in memory and the opcode fetching circuitry. The prefetch queue works well when the CPU isn't constantly accessing memory. When the CPU isn't accessing memory, the BIU can fetch additional instruction opcodes for the prefetch queue. Alas, the pipelined 80x86 CPUs are constantly accessing memory since they fetch an opcode byte on every clock cycle. Therefore, the prefetch queue cannot take advantage of any "dead" bus cycles to fetch additional opcode bytes – there aren't any "dead" bus cycles. However, the prefetch queue is still valuable for a very simple reason: the BIU fetches multiple bytes on each memory access and most instructions are shorter. Without the prefetch queue, the system would have to explicitly fetch each opcode, even if the BIU had already "accidentally" fetched the opcode along with the previous instruction. With the prefetch queue, however, the system will not refetch any opcodes. It fetches them once and saves them for use by the opcode fetch unit.

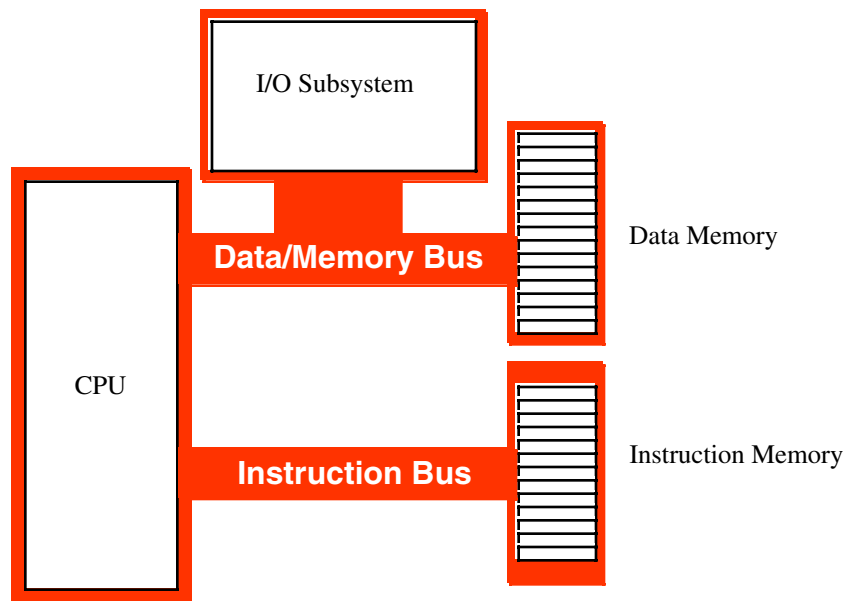
For example, if you execute two one-byte instructions in a row, the BIU can fetch both opcodes in one memory cycle, freeing up the bus for other operations. The CPU can use these available bus cycles to fetch additional opcodes or to deal with other memory accesses.

Of course, not all instructions are one byte long. The 80x86 has a large number of different instruction sizes. If you execute several large instructions in a row, you're going to run slower. Once again we return to that same rule: *the fastest programs are the ones which use the shortest instructions*. If you can use shorter instructions to accomplish some task, do so.

Suppose, for a moment, that the CPU has two separate memory spaces, one for instructions and one for data, each with their own bus. This is called the *Harvard Architecture* since the first such machine was built at Harvard. On a Harvard machine there would be no contention for the bus. The BIU could continue to fetch opcodes on the instruction bus while accessing memory on the data/memory bus (see Figure 4.8),

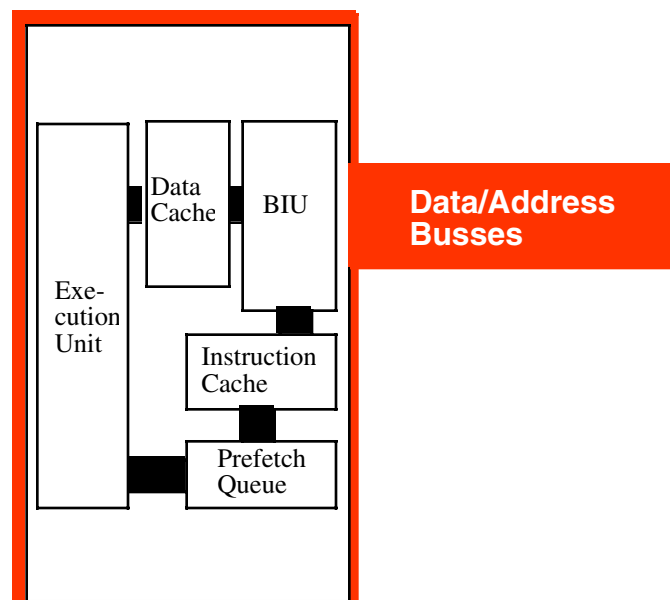
---

16. Note, by the way, that the number of stages in an instruction pipeline varies among CPUs.



**Figure 4.8 A Typical Harvard Machine**

In the real world, there are very few true Harvard machines. The extra pins needed on the processor to support two physically separate busses increase the cost of the processor and introduce many other engineering problems. However, microprocessor designers have discovered that they can obtain many benefits of the Harvard architecture with few of the disadvantages by using separate on-chip caches for data and instructions. Advanced CPUs use an internal Harvard architecture and an external Von Neumann architecture. Figure 4.9 shows the structure of the 80x86 with separate data and instruction caches.



**Figure 4.9 Using Separate Code and Data Caches**

Each path inside the CPU represents an independent bus. Data can flow on all paths concurrently. This means that the prefetch queue can be pulling instruction opcodes from the instruction cache while the execution unit is writing data to the data cache. Now the BIU only fetches opcodes from memory whenever it cannot locate them in the instruction cache. Likewise, the data cache buffers memory. The CPU uses the data/address bus only when reading a value which is not in the cache or when flushing data back to main memory.

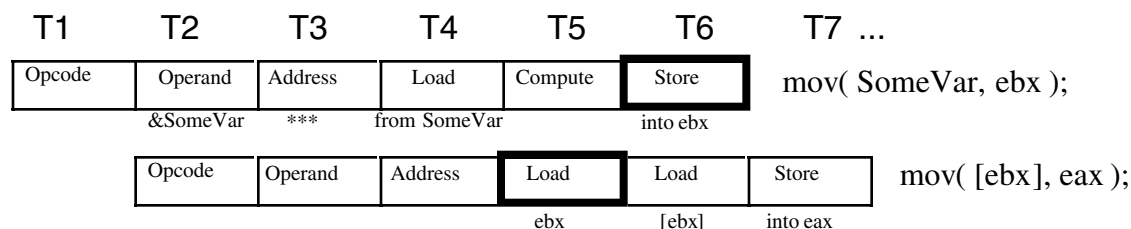
Although you cannot control the presence, size, or type of cache on a CPU, as an assembly language programmer you must be aware of how the cache operates to write the best programs. On-chip level one instruction caches are generally quite small (8,192 bytes on the 80486, for example). Therefore, the shorter your instructions, the more of them will fit in the cache (getting tired of “shorter instructions” yet?). The more instructions you have in the cache, the less often bus contention will occur. Likewise, using registers to hold temporary results places less strain on the data cache so it doesn’t need to flush data to memory or retrieve data from memory quite so often. *Use the registers wherever possible!*

#### 4.8.4 Hazards

There is another problem with using a pipeline: the data hazard. Let’s look at the execution profile for the following instruction sequence:

```
mov( SomeVar, ebx );
mov( [ebx], eax );
```

When these two instructions execute, the pipeline will look something like shown in Figure 4.10:

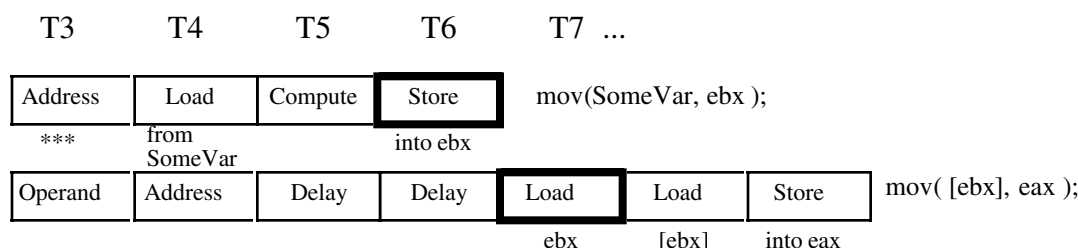


**Figure 4.10 A Data Hazard**

Note a major problem here. These two instructions fetch the 32 bit value whose address appears at location &SomeVar in memory. *But this sequence of instructions won’t work properly!* Unfortunately, the second instruction has already used the value in EBX before the first instruction loads the contents of memory location &SomeVar (T4 & T6 in the diagram above).

CISC processors, like the 80x86, handle hazards automatically<sup>17</sup>. However, they will stall the pipeline to synchronize the two instructions. The actual execution would look something like shown in Figure 4.11.

17. Some RISC chips do not. If you tried this sequence on certain RISC chips you would get an incorrect answer.



**Figure 4.11 How the 80x86 Handles a Data Hazard**

By delaying the second instruction two clock cycles, the 8486 guarantees that the load instruction will load EAX from the proper address. Unfortunately, the second load instruction now executes in three clock cycles rather than one. However, requiring two extra clock cycles is better than producing incorrect results. Fortunately, you can reduce the impact of hazards on execution speed within your software.

Note that the data hazard occurs when the source operand of one instruction was a destination operand of a previous instruction. There is nothing wrong with loading EBX from SomeVar and then loading EAX from [EBX], *unless they occur one right after the other*. Suppose the code sequence had been:

```
mov( 2000, ecx );
mov( SomeVar, ebx );
mov( [ebx], eax );
```

We could reduce the effect of the hazard that exists in this code sequence by simply *rearranging the instructions*. Let's do that and obtain the following:

```
mov( SomeVar, ebx );
mov( 2000, ecx );
mov( [ebx], eax );
```

Now the "mov( [ebx], eax);" instruction requires only one additional clock cycle rather than two. By inserting yet another instruction between the "mov( SomeVar, ebx);" and the "mov( [ebx], eax);" instructions you can eliminate the effects of the hazard altogether<sup>18</sup>.

On a pipelined processor, the order of instructions in a program may dramatically affect the performance of that program. Always look for possible hazards in your instruction sequences. Eliminate them wherever possible by rearranging the instructions.

In addition to data hazards, there are also *control hazards*. We've actually discussed control hazards already, although we did not refer to them by that name. A control hazard occurs whenever the CPU branches to some new location in memory and the CPU has to flush the following instructions following the branch that are in various stages of execution.

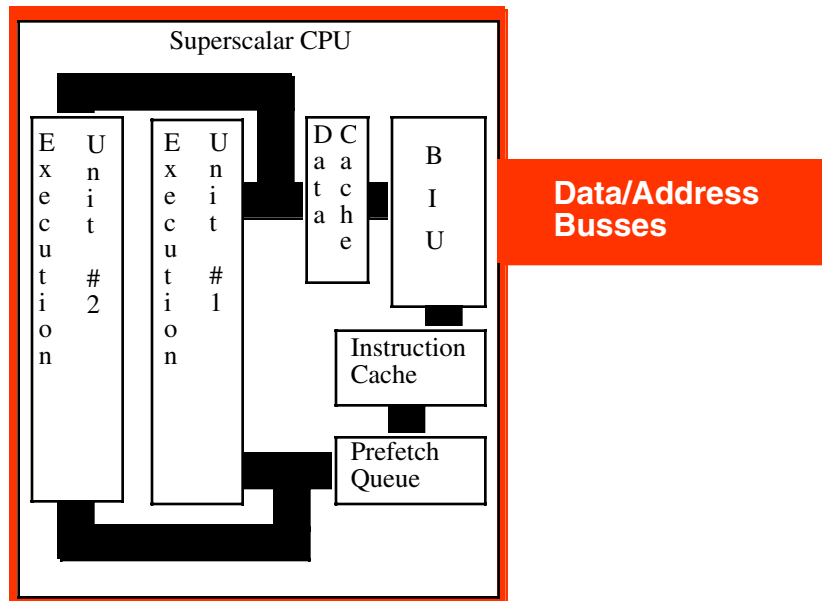
## 4.8.5 Superscalar Operation— Executing Instructions in Parallel

With the pipelined architecture we could achieve, at best, execution times of one CPI (clock per instruction). Is it possible to execute instructions faster than this? At first glance you might think, "Of course not, we can do at most one operation per clock cycle. So there is no way we can execute more than one instruction per clock cycle." Keep in mind however, that a single instruction is *not* a single operation. In the examples presented earlier each instruction has taken between six and eight operations to complete. By adding seven or eight separate units to the CPU, we could effectively execute these eight operations in one clock

18. Of course, any instruction you insert at this point must *not* modify the values in the eax and ebx registers. Also note that the examples in this section are contrived to demonstrate pipeline stalls. Actual 80x86 CPUs have additional circuitry to help reduce the effect of pipeline stalls on the execution time.

cycle, yielding one CPI. If we add more hardware and execute, say, 16 operations at once, can we achieve 0.5 CPI? The answer is a qualified “yes.” A CPU including this additional hardware is a *superscalar* CPU and can execute more than one instruction during a single clock cycle. The 80x86 family began supporting superscalar execution with the introduction of the Pentium processor.

A superscalar CPU has, essentially, several execution units (see Figure 4.12). If it encounters two or more instructions in the instruction stream (i.e., the prefetch queue) which can execute independently, it will do so.



**Figure 4.12 A CPU that Supports Superscalar Operation**

There are a couple of advantages to going superscalar. Suppose you have the following instructions in the instruction stream:

```
mov( 1000, eax );
mov( 2000, ebx );
```

If there are no other problems or hazards in the surrounding code, and all six bytes for these two instructions are currently in the prefetch queue, there is no reason why the CPU cannot fetch and execute both instructions in parallel. All it takes is extra silicon on the CPU chip to implement two execution units.

Besides speeding up independent instructions, a superscalar CPU can also speed up program sequences which have hazards. One limitation of superscalar CPU is that once a hazard occurs, the offending instruction will completely stall the pipeline. Every instruction which follows will also have to wait for the CPU to synchronize the execution of the instructions. With a superscalar CPU, however, instructions following the hazard may continue execution through the pipeline as long as they don't have hazards of their own. This alleviates (though does not eliminate) some of the need for careful instruction scheduling.

As an assembly language programmer, the way you write software for a superscalar CPU can dramatically affect its performance. First and foremost is that rule you're probably sick of by now: *use short instructions*. The shorter your instructions are, the more instructions the CPU can fetch in a single operation and, therefore, the more likely the CPU will execute faster than one CPI. Most superscalar CPUs do not completely duplicate the execution unit. There might be multiple ALUs, floating point units, etc. This means that certain instruction sequences can execute very quickly while others won't. You have to study the exact composition of your CPU to decide which instruction sequences produce the best performance.

## 4.8.6 Out of Order Execution

In a standard superscalar CPU it is the programmer's (or compiler's) responsibility to schedule (arrange) the instructions to avoid hazards and pipeline stalls. Fancier CPUs can actually remove some of this burden and improve performance by automatically rescheduling instructions while the program executes. To understand how this is possible, consider the following instruction sequence:

```
mov( SomeVar, ebx );
mov( [ebx], eax );
mov( 2000, ecx );
```

A data hazard exists between the first and second instructions above. The second instruction must delay until the first instruction completes execution. This introduces a pipeline stall and increases the running time of the program. Typically, the stall affects every instruction that follows. However, note that the third instruction's execution does not depend on the result from either of the first two instructions. Therefore, there is no reason to stall the execution of the "mov( 2000, ecx );" instruction. It may continue executing while the second instruction waits for the first to complete. This technique, appearing in later members of the Pentium line, is called "out of order execution" because the CPU completes the execution of some instruction prior to the execution of previous instructions appearing in the code stream.

Clearly, the CPU may only execute instruction out of sequence if doing so produces exactly the same results as in-order execution. While there are a lot of little technical issues that make this problem a little more difficult than it seems, with enough engineering effort it is quite possible to implement this feature.

Although you might think that this extra effort is not worth it (why not make it the programmer's or compiler's responsibility to schedule the instructions) there are some situations where out of order execution will improve performance that static scheduling could not handle.

## 4.8.7 Register Renaming

One problem that hampers the effectiveness of superscalar operation on the 80x86 CPU is the 80x86's limited number of general purpose registers. Suppose, for example, that the CPU had four different pipelines and, therefore, was capable of executing four instructions simultaneously. Actually achieving four instructions per clock cycle would be very difficult because most instructions (that can execute simultaneously with other instructions) operate on two register operands. For four instructions to execute concurrently, you'd need four separate destination registers and four source registers (and the two sets of registers must be disjoint, that is, a destination register for one instruction cannot be the source of another). CPUs that have lots of registers can handle this task quite easily, but the limited register set of the 80x86 makes this difficult. Fortunately, there is a way to alleviate part of the problem: through *register renaming*.

Register renaming is a sneaky way to give a CPU more registers than it actually has. Programmers will not have direct access to these extra registers, but the CPU can use these additional registers to prevent hazards in certain cases. For example, consider the following short instruction sequence:

```
mov( 0, eax );
mov( eax, i );
mov( 50, eax );
mov( eax, j );
```

Clearly a data hazard exists between the first and second instructions and, likewise, a data hazard exists between the third and fourth instructions in this sequence. Out of order execution in a superscalar CPU would normally allow the first and third instructions to execute concurrently and then the second and fourth instructions could also execute concurrently. However, a data hazard, of sorts, also exists between the first and third instructions since they use the same register. The programmer could have easily solved this problem by using a different register (say EBX) for the third and fourth instructions. However, let's assume that the programmer was unable to do this because the other registers are all holding important values. Is this sequence doomed to executing in four cycles on a superscalar CPU that should only require two?

One advanced trick a CPU can employ is to create a bank of registers for each of the general purpose registers on the CPU. That is, rather than having a single EAX register, the CPU could support an array of EAX registers; let's call these registers EAX[0], EAX[1], EAX[2], etc. Similarly, you could have an array of each of the registers, so we could also have EBX[0]..EBX[n], ECX[0]..ECX[n], etc. Now the instruction set does not give the programmer the ability to select one of these specific register array elements for a given instruction, but the CPU can automatically choose a different register array element if doing so would not change the overall computation and doing so could speed up the execution of the program. For example, consider the following sequence (with register array elements automatically chosen by the CPU):

```
mov( 0, eax[0] );
mov( eax[0], i );
mov( 50, eax[1] );
mov( eax[1], j );
```

Since EAX[0] and EAX[1] are different registers, the CPU can execute the first and third instructions concurrently. Likewise, the CPU can execute the second and fourth instructions concurrently.

The code above provides an example of *register renaming*. Dynamically, the CPU automatically selects one of several different elements from a register array in order to prevent data hazards. Although this is a simple example, and different CPUs implement register renaming in many different ways, this example does demonstrate how the CPU can improve performance in certain instances through the use of this technique.

#### 4.8.8 Very Long Instruction Word Architecture (VLIW)

Superscalar operation attempts to schedule, in hardware, the execution of multiple instructions simultaneously. Another technique that Intel is using in their IA-64 architecture is the use of very long instruction words, or VLIW. In a VLIW computer system, the CPU fetches a large block of bytes (41 in the case of the IA-64 Itanium CPU) and decodes and executes this block all at once. This block of bytes usually contains two or more instructions (three in the case of the IA-64). VLIW computing requires the programmer or compiler to properly schedule the instructions in each block (so there are no hazards or other conflicts), but if properly scheduled, the CPU can execute three or more instructions per clock cycle.

The Intel IA-64 Architecture is not the only computer system to employ a VLIW architecture. Transmeta's Crusoe processor family also uses a VLIW architecture. The Crusoe processor is different than the IA-64 architecture insofar as it does not support native execution of IA-32 instructions. Instead, the Crusoe processor dynamically translates 80x86 instructions to Crusoe's VLIW instructions. This "code morphing" technology results in code running about 50% slower than native code, though the Crusoe processor has other advantages.

We will not consider VLIW computing any further since the IA-32 architecture does not support it. But keep this architectural advance in mind if you move towards the IA-64 family or the Crusoe family.

#### 4.8.9 Parallel Processing

Most of the techniques for improving CPU performance via architectural advances involve the parallel (overlapped) execution of instructions. Most of the techniques of this chapter are transparent to the programmer. That is, the programmer does not have to do anything special to take minimal advantage of the parallel operation of pipelines and superscalar operations. True, if programmers are aware of the underlying architecture they can write code that runs even faster, but these architectural advances often improve performance even if programmers do not write special code to take advantage of them.

The only problem with this approach (attempting to dynamically parallelize an inherently sequential program) is that there is only so much you can do to parallelize a program that requires sequential execution for proper operation (which covers most programs). To truly produce a parallel program, the programmer must specifically write parallel code; of course, this does require architectural support from the CPU. This section and the next touches on the types of support a CPU can provide.

Typical CPUs use what is known as the SISD model: *Single Instruction, Single Data*. This means that the CPU executes one instruction at a time that operates on a single piece of data<sup>19</sup>. Two common parallel models are the so-called SIMD (*Single Instruction, Multiple Data*) and MIMD (*Multiple Instruction, Multiple Data*) models. As it turns out, x86 systems can support both of these parallel execution models.

In the SIMD model, the CPU executes a single instruction stream, just like the standard SISD model. However, the CPU executes the specified operation on multiple pieces of data concurrently rather than a single data object. For example, consider the 80x86 ADD instruction. This is a SISD instruction that operates on (that is, produces) a single piece of data; true, the instruction fetches values from two source operands and stores a sum into a destination operand but the end result is that the ADD instruction will only produce a single sum. An SIMD version of ADD, on the other hand, would compute the sum of several values simultaneously. The Pentium III's MMX and SIMD instruction extensions operate in exactly this fashion. With an MMX instruction, for example, you can add up to eight separate pairs of values with the execution of a single instruction. The aptly named SIMD instruction extensions operate in a similar fashion.

Note that SIMD instructions are only useful in specialized situations. Unless you have an algorithm that can take advantage of SIMD instructions, they're not that useful. Fortunately, high-speed 3-D graphics and multimedia applications benefit greatly from these SIMD (and MMX) instructions, so their inclusion in the 80x86 CPU offers a huge performance boost for these important applications.

The MIMD model uses multiple instructions, operating on multiple pieces of data (usually one instruction per data object, though one of these instructions could also operate on multiple data items). These multiple instructions execute independently of one another. Therefore, it's very rare that a single program (or, more specifically, a single thread of execution) would use the MIMD model. However, if you have a multi-programming environment with multiple programs attempting to execute concurrently in memory, the MIMD model does allow each of those programs to execute their own code stream concurrently. This type of parallel system is usually called a multiprocessor system. Multiprocessor systems are the subject of the next section.

The common computation models are SISD, SIMD, and MIMD. If you're wondering if there is a MISD model (Multiple Instruction, Single Data) the answer is no. Such an architecture doesn't really make sense.

## 4.8.10 Multiprocessing

Pipelining, superscalar operation, out of order execution, and VLIW design are techniques CPU designers use in order to execute several operations in parallel. These techniques support *fine-grained parallelism*<sup>20</sup> and are useful for speeding up adjacent instructions in a computer system. If adding more functional units increases parallelism (and, therefore, speeds up the system), you might wonder what would happen if you added two CPUs to the system. This technique, known as *multiprocessing*, can improve system performance, though not as uniformly as other techniques. As noted in the previous section, a multiprocessor system uses the MIMD parallel execution model.

The techniques we've considered to this point don't require special programming to realize a performance increase. True, if you do pay attention you will get better performance; but no special programming is necessary to activate these features. Multiprocessing, on the other hand, doesn't help a program one bit unless that program was specifically written to use multiprocessor (or runs under an O/S specifically written to support multiprocessing). If you build a system with two CPUs, those CPUs cannot trade off executing alternate instructions within a program. In fact, it is very expensive (timewise) to switch the execution of a program from one processor to another. Therefore, multiprocessor systems are really only effective in a system that execute multiple programs concurrently (i.e., a multitasking system)<sup>21</sup>. To differentiate this type of parallelism from that afforded by pipelining and superscalar operation, we'll call this kind of parallelism *coarse-grained parallelism*.

19. We will ignore the parallelism provided by pipelining and superscalar operation in this discussion.

20. For our purposes, fine-grained parallelism means that we are executing adjacent program instructions in parallel.

21. Technically, it only needs to execute multiple threads concurrently, but we'll ignore this distinction here since the technical distinction between threads and programs/processes is beyond the scope of this chapter.

Adding multiple processors to a system is not as simple as wiring the processor to the motherboard. A big problem with multiple processors is the *cache coherency* problem. To understand this problem, consider two separate programs running on separate processors in a multiprocessor system. Suppose also that these two processors communicate with one another by writing to a block of shared physical memory. Unfortunately, when CPU #1 writes to this block of addresses the CPU caches the data up and might not actually write the data to physical memory for some time. Simultaneously, CPU #2 might be attempting to read this block of shared memory but winds up reading the data out of its local cache rather than the data that CPU #1 wrote to the block of shared memory (assuming the data made it out of CPU #1's local cache). In order for these two functions to operate properly, the two CPUs must communicate writes to common memory addresses in cache between themselves. This is a very complex and involved process.

Currently, the Pentium III and IV processors directly support cache updates between two CPUs in a system. Intel also builds a more expensive processor, the XEON, that supports more than two CPUs in a system. However, one area where the RISC CPUs have a big advantage over Intel is in the support for multiple processors in a system. While Intel systems reach a point of diminishing returns at about 16 processors, Sun SPARC and other RISC processors easily support 64-CPU systems (with more arriving, it seems, every day). This is why large databases and large web server systems tend to use expensive UNIX-based RISC systems rather than x86 systems.

---

## 4.9 Putting It All Together

The performance of modern CPUs is intrinsically tied to the architecture of that CPU. Over the past half century there have been many major advances in CPU design that have dramatically improved performance. Although the clock frequency has improved by over four orders of magnitude during this time period, other improvements have added one or two orders of magnitude improvement as well. Over the 80x86's lifetime, performance has improved 10,000-fold.

Unfortunately, the 80x86 family has just about pushed the limits of what it can achieve by extending the architecture. Perhaps another order of magnitude is possible, but Intel is reaching the point of diminishing returns. Having realized this, Intel has chosen to implement a new architecture using VLIW for their IA-64 family. Only time will prove whether their approach is the correct one, but most people believe that the IA-32 has reached the end of its lifetime. On the other hand, people have been announcing the death of the IA-32 for the past decade, so we'll see what happens now.

---

# Instruction Set Architecture

# Chapter Five

---

## 5.1 Chapter Overview

This chapter discusses the low-level implementation of the 80x86 instruction set. It describes how the Intel engineers decided to encode the instructions in a numeric format (suitable for storage in memory) and it discusses the trade-offs they had to make when designing the CPU. This chapter also presents a historical background of the design effort so you can better understand the compromises they had to make.

---

## 5.2 The Importance of the Design of the Instruction Set

In this chapter we will be exploring one of the most interesting and important aspects of CPU design: the design of the CPU's instruction set. The instruction set architecture (or ISA) is one of the most important design issues that a CPU designer must get right from the start. Features like caches, pipelining, superscalar implementation, etc., can all be grafted on to a CPU design long after the original design is obsolete. However, it is very difficult to change the instructions a CPU executes once the CPU is in production and people are writing software that uses those instructions. Therefore, one must carefully choose the instructions for a CPU.

You might be tempted to take the "kitchen sink" approach to instruction set design<sup>1</sup> and include as many instructions as you can dream up in your instruction set. This approach fails for several reasons we'll discuss in the following paragraphs. Instruction set design is the epitome of compromise management. Good CPU design is the process of selecting what to throw out rather than what to leave in. It's easy enough to say "let's include everything." The hard part is deciding what to leave out once you realize you can't put everything on the chip.

**Nasty reality #1: Silicon real estate.** The first problem with "putting it all on the chip" is that each feature requires some number of transistors on the CPU's silicon die. CPU designers work with a "silicon budget" and are given a finite number of transistors to work with. This means that there aren't enough transistors to support "putting all the features" on a CPU. The original 8086 processor, for example, had a transistor budget of less than 30,000 transistors. The Pentium III processor had a budget of over eight million transistors. These two budgets reflect the differences in semiconductor technology in 1978 vs. 1998.

**Nasty reality #2: Cost.** Although it is possible to use millions of transistors on a CPU today, the more transistors you use the more expensive the CPU. Pentium III processors, for example, cost hundreds of dollars (circa 2000). A CPU with only 30,000 transistors (also circa 2000) would cost only a few dollars. For low-cost systems it may be more important to shave some features and use fewer transistors, thus lowering the CPU's cost.

**Nasty reality #3: Expandability.** One problem with the "kitchen sink" approach is that it's very difficult to anticipate all the features people will want. For example, Intel's MMX and SIMD instruction enhancements were added to make multimedia programming more practical on the Pentium processor. Back in 1978 very few people could have possibly anticipated the need for these instructions.

**Nasty reality #4: Legacy Support.** This is almost the opposite of expandability. Often it is the case that an instruction the CPU designer feels is important turns out to be less useful than anticipated. For example, the LOOP instruction on the 80x86 CPU sees very little use in modern high-performance programs. The 80x86 ENTER instruction is another good example. When designing a CPU using the "kitchen sink" approach, it is often common to discover that programs almost never use some of the available instructions. Unfortunately, you cannot easily remove instructions in later versions of a processor because this will break some existing programs that use those instructions. Generally, once you add an instruction you have to support it forever in the instruction set. Unless very few programs use the instruction (and you're willing to let

---

1. As in "Everything, including the kitchen sink."

them break) or you can automatically simulate the instruction in software, removing instructions is a very difficult thing to do.

**Nasty reality #4: Complexity.** The popularity of a new processor is easily measured by how much software people write for that processor. Most CPU designs die a quick death because no one writes software specific to that CPU. Therefore, a CPU designer must consider the assembly programmers and compiler writers who will be using the chip upon introduction. While a "kitchen sink" approach might seem to appeal to such programmers, the truth is no one wants to learn an overly complex system. If your CPU does everything under the sun, this might appeal to someone who is already familiar with the CPU. However, pity the poor soul who doesn't know the chip and has to learn it all at once.

These problems with the "kitchen sink" approach all have a common solution: design a simple instruction set to begin with and leave room for later expansion. This is one of the main reasons the 80x86 has proven to be so popular and long-lived. Intel started with a relatively simple CPU and figured out how to extend the instruction set over the years to accommodate new features.

---

### 5.3 Basic Instruction Design Goals

In a typical Von Neumann architecture CPU, the computer encodes CPU instructions as numeric values and stores these numeric values in memory. The encoding of these instructions is one of the major tasks in instruction set design and requires very careful thought.

To encode an instruction we must pick a unique numeric opcode value for each instruction (clearly, two different instructions cannot share the same numeric value or the CPU will not be able to differentiate them when it attempts to decode the opcode value). With an  $n$ -bit number, there are  $2^n$  different possible opcodes, so to encode  $m$  instructions you will need an opcode that is at least  $\log_2(m)$  bits long.

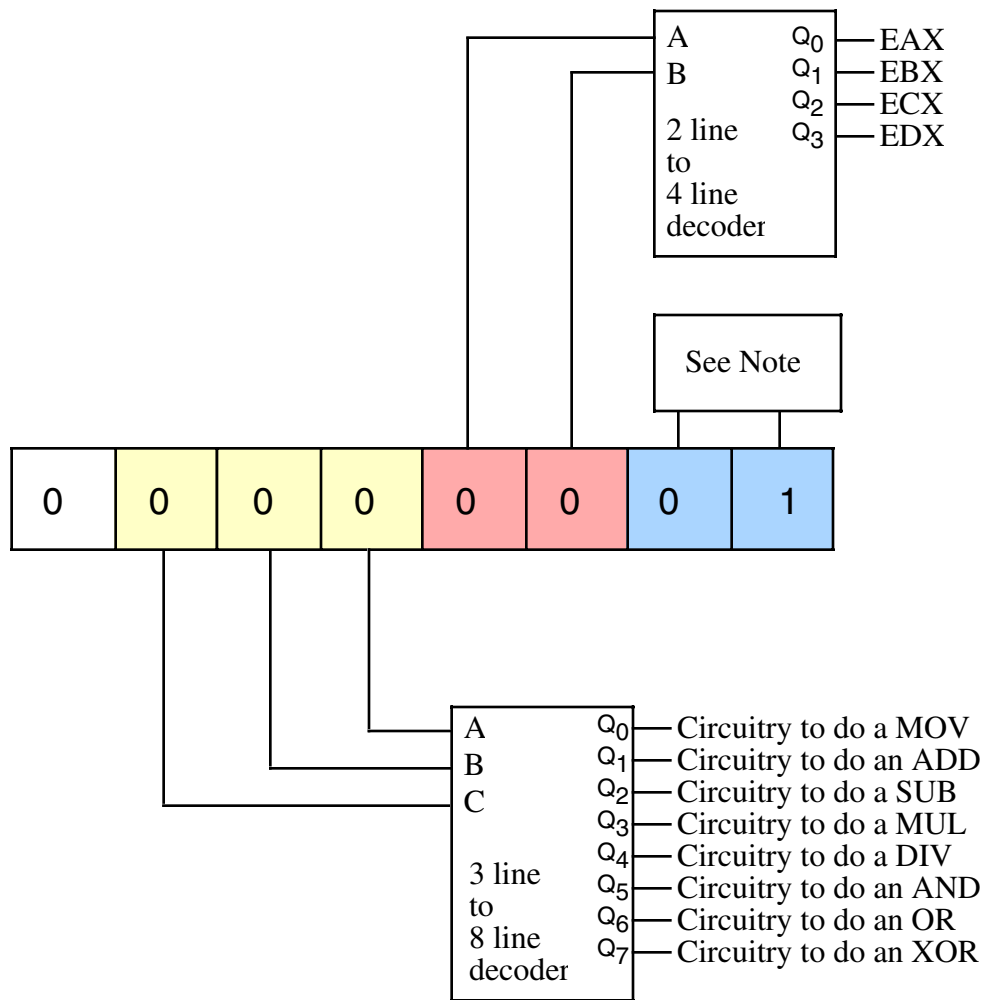
Encoding opcodes is a little more involved than assigning a unique numeric value to each instruction. Remember, we have to use actual hardware (i.e., decoder circuits) to figure out what each instruction does and command the rest of the hardware to do the specified task. Suppose you have a seven-bit opcode. With an opcode of this size we could encode 128 different instructions. To decode each instruction individually requires a seven-line to 128-line decode – an expensive piece of circuitry. Assuming our instructions contain certain patterns, we can reduce the hardware by replacing this large decoder with three smaller decoders.

If you have 128 truly unique instructions, there's little you can do other than to decode each instruction individually. However, in most architectures the instructions are not completely independent of one another. For example, on the 80x86 CPUs the opcodes for "mov( eax, ebx );" and "mov( ecx, edx );" are different (because these are different instructions) but these instructions are not unrelated. They both move data from one register to another. In fact, the only difference between them is the source and destination operands. This suggests that we could encode instructions like MOV with a sub-opcode and encode the operands using other strings of bits within the opcode.

For example, if we really have only eight instructions, each instruction has two operands, and each operand can be one of four different values, then we can encode the opcode as three packed fields containing three, two, and two bits (see Figure 5.1). This encoding only requires the use of three simple decoders to completely determine what instruction the CPU should execute. While this is a bit of a trivial case, it does demonstrate one very important facet of instruction set design – it is important to make opcodes easy to decode and the easiest way to do this is to break up the opcode into several different bit fields, each field contributing part of the information necessary to execute the full instruction. The smaller these bit fields, the easier it will be for the hardware to decode and execute them<sup>2</sup>.

---

2. Not to mention faster and less expensive.



Note: the circuitry attached to the destination register bits is identical to the circuitry for the source register bits.

**Figure 5.1 Separating an Opcode into Separate Fields to Ease Decoding**

Although Intel probably went a little overboard with the design of the original 8086 instruction set, an important design goal is to keep instruction sizes within a reasonable range. CPUs with unnecessarily long instructions consume extra memory for their programs. This tends to create more cache misses and, therefore, hurts the overall performance of the CPU. Therefore, we would like our instructions to be as compact as possible so our programs' code uses as little memory as possible.

It would seem that if we are encoding  $2^n$  different instructions using  $n$  bits, there would be very little leeway in choosing the size of the instruction. It's going to take  $n$  bits to encode those  $2^n$  instructions, you can't do it with any fewer. You may, of course, use more than  $n$  bits. And believe it or not, that's the secret to reducing the size of a typical program on the CPU.

Before discussing how to use longer instructions to generate shorter programs, a short digression is necessary. The first thing to note is that we generally cannot choose an arbitrary number of bits for our opcode length. Assuming that our CPU is capable of reading bytes from memory, the opcode will probably have to be some even multiple of eight bits long. If the CPU is not capable of reading bytes from memory (e.g., most RISC CPUs only read memory in 32 or 64 bit chunks) then the opcode is going to be the same size as

the smallest object the CPU can read from memory at one time (e.g., 32 bits on a typical RISC chip). Any attempt to shrink the opcode size below this data bus enforced lower limit is futile. Since we're discussing the 80x86 architecture in this text, we'll work with opcodes that must be an even multiple of eight bits long.

Another point to consider here is the size of an instruction's operands. Some CPU designers (specifically, RISC designers) include all operands in their opcode. Other CPU designers (typically CISC designers) do not count operands like immediate constants or address displacements as part of the opcode (though they do usually count register operand encodings as part of the opcode). We will take the CISC approach here and not count immediate constant or address displacement values as part of the actual opcode.

With an eight-bit opcode you can only encode 256 different instructions. Even if we don't count the instruction's operands as part of the opcode, having only 256 different instructions is somewhat limiting. It's not that you can't build a CPU with an eight-bit opcode, most of the eight-bit processors predating the 8086 had eight-bit opcodes, it's just that modern processors tend to have far more than 256 different instructions. The next step up is a two-byte opcode. With a two-byte opcode we can have up to 65,536 different instructions (which is probably enough) but our instructions have doubled in size (not counting the operands, of course).

If reducing the instruction size is an important design goal<sup>3</sup> we can employ some techniques from data compression theory to reduce the average size of our instructions. The basic idea is this: first we analyze programs written for our CPU (or a CPU similar to ours if no one has written any programs for our CPU) and count the number of occurrences of each opcode in a large number of typical applications. We then create a sorted list of these opcodes from most-frequently-used to least-frequently-used. Then we attempt to design our instruction set using one-byte opcodes for the most-frequently-used instructions, two-byte opcodes for the next set of most-frequently-used instructions, and three (or more) byte opcodes for the rarely used instructions. Although our maximum instruction size is now three or more bytes, most of the actual instructions appearing in a program will use one or two byte opcodes, so the average opcode length will be somewhere between one and two bytes (let's call it 1.5 bytes) and a typical program will be shorter than had we chosen a two byte opcode for all instructions (see Figure 5.2).

---

3. To many CPU designers it is not; however, since this was a design goal for the 8086 we'll follow this path.

0	1	X	X	X	X	X	X
1	0	X	X	X	X	X	X
1	1	X	X	X	X	X	X

If the H.O. two bits of the first opcode byte are not both zero, then the whole opcode is one byte long and the remaining six bits let us encode 64 one-byte instructions. Since there are a total of three opcode bytes of these form, we can encode up to 192 different one-byte instructions.

0	0	1	X	X	X	X	X
X	X	X	X	X	X	X	X

If the H.O. three bits of our first opcode byte contain %001, then the opcode is two bytes long and the remaining 13 bits let us encode 8192 different instructions.

0	0	0	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

If the H.O. three bits of our first opcode byte contain all zeros, then the opcode is three bytes long and the remaining 21 bits let us encode two million ( $2^{21}$ ) different instructions.

**Figure 5.2 Encoding Instructions Using a Variable-Length Opcode**

Although using variable-length instructions allows us to create smaller programs, it comes at a price. First of all, decoding the instructions is a bit more complicated. Before decoding an instruction field, the CPU must first decode the instruction's size. This extra step consumes time and may affect the overall performance of the CPU (by introducing delays in the decoding step and, thereby, limiting the maximum clock

frequency of the CPU). Another problem with variable length instructions is that it makes decoding multiple instructions in a pipeline quite difficult (since we cannot trivially determine the instruction boundaries in the prefetch queue). These reasons, along with some others, is why most popular RISC architectures avoid variable-sized instructions. However, for our purpose, we'll go with a variable length approach since saving memory is an admirable goal.

Before actually choosing the instructions you want to implement in your CPU, now would be a good time to plan for the future. Undoubtedly, you will discover the need for new instructions at some point in the future, so reserving some opcodes specifically for that purpose is a real good idea. If you were using the instruction encoding appearing in Figure 5.2 for your opcode format, it might not be a bad idea to reserve one block of 64 one-byte opcodes, half (4,096) of the two-byte instructions, and half (1,048,576) of the three-byte opcodes for future use. In particular, giving up 64 of the very valuable one-byte opcodes may seem extravagant, but history suggests that such foresight is rewarded.

The next step is to choose the instructions you want to implement. Note that although we've reserved nearly half the instructions for future expansion, we don't actually have to implement instructions for all the remaining opcodes. We can choose to leave a good number of these instructions unimplemented (and effectively reserve them for the future as well). The right approach is not to see how quickly we can use up all the opcodes, but rather to ensure that we have a consistent and complete instruction set given the compromises we have to live with (e.g., silicon limitations). The main point to keep in mind here is that it's much easier to add an instruction later than it is to remove an instruction later. So for the first go-around, it's generally better to go with a simpler design rather than a more complex design.

The first step is to choose some generic instruction types. For a first attempt, you should limit the instructions to some well-known and common instructions. The best place to look for help in choosing these instructions is the instruction sets of other processors. For example, most processors you find will have instructions like the following:

- Data movement instructions (e.g., MOV)
- Arithmetic and logical instructions (e.g., ADD, SUB, AND, OR, NOT)
- Comparison instructions
- A set of conditional jump instructions (generally used after the compare instructions)
- Input/Output instructions
- Other miscellaneous instructions

Your goal as the designer of the CPU's initial instruction set is to choose a reasonable set of instructions that will allow programmers to efficiently write programs (using as few instructions as possible) without adding so many instructions you exceed your silicon budget or violate other system compromises. This is a very strategic decision, one that CPU designers should base on careful research, experimentation, and simulation. The job of the CPU designer is not to create the best instruction set, but to create an instruction set that is optimal given all the constraints.

Once you've decided which instructions you want to include in your (initial) instruction set, the next step is to assign opcodes for them. The first step is to group your instructions into sets by common characteristics of those instructions. For example, an ADD instruction is probably going to support the exact same set of operands as the SUB instruction. So it makes sense to put these two instructions into the same group. On the other hand, the NOT instruction generally requires only a single operand<sup>4</sup> as does a NEG instruction. So you'd probably put these two instructions in the same group but a different group than ADD and SUB.

Once you've grouped all your instructions, the next step is to encode them. A typical encoding will use some bits to select the group the instruction falls into, it will use some bits to select a particular instruction from that group, and it will use some bits to determine the types of operands the instruction allows (e.g., registers, memory locations, and constants). The number of bits needed to encode all this information may have a direct impact on the instruction's size, regardless of the frequency of the instruction. For example, if you need two bits to select a group, four bits to select an instruction within that group, and six bits to specify the instruction's operand types, you're not going to fit this instruction into an eight-bit opcode. On the other hand, if all you really want to do is push one of eight different registers onto the stack, you can use four bits

---

4. Assuming this operation treats its single operand as both a source and destination operand, a common way of handling this instruction.

to select the PUSH instruction and three bits to select the register (assuming the encoding in Figure 5.2 the eighth and H.O. bit would have to contain zero).

Encoding operands is always a problem because many instructions allow a large number of operands. For example, the generic 80x86 MOV instruction requires a two-byte opcode<sup>5</sup>. However, Intel noticed that the "mov( disp, eax );" and "mov( eax, disp );" instructions occurred very frequently. So they created a special one byte version of this instruction to reduce its size and, therefore, the size of those programs that use this instruction frequently. Note that Intel did not remove the two-byte versions of these instructions. They have two different instructions that will store EAX into memory or load EAX from memory. A compiler or assembler would always emit the shorter of the two instructions when given an option of two or more instructions that wind up doing exactly the same thing.

Notice an important trade-off Intel made with the MOV instruction. They gave up an extra opcode in order to provide a shorter version of one of the MOV instructions. Actually, Intel used this trick all over the place to create shorter and easier to decode instructions. Back in 1978 this was a good compromise (reducing the total number of possible instructions while also reducing the program size). Today, a CPU designer would probably want to use those redundant opcodes for a different purpose, however, Intel's decision was reasonable at the time (given the high cost of memory in 1978).

To further this discussion, we need to work with an example. So the next section will go through the process of designing a very simple instruction set as a means of demonstrating this process.

---

## 5.4 The Y86 Hypothetical Processor

Because of enhancements made to the 80x86 processor family over the years, Intel's design goals in 1978, and advances in computer architecture occurring over the years, the encoding of 80x86 instructions is very complex and somewhat illogical. Therefore, the 80x86 is not a good candidate for an example architecture when discussing how to design and encode an instruction set. However, since this is a text about 80x86 assembly language programming, attempting to present the encoding for some simpler real-world processor doesn't make sense. Therefore, we will discuss instruction set design in two stages: first, we will develop a simple (trivial) instruction set for a hypothetical processor that is a small subset of the 80x86, then we will expand our discussion to the full 80x86 instruction set. Our hypothetical processor is not a true 80x86 CPU, so we will call it the Y86 processor to avoid any accidental association with the Intel x86 family.

The Y86 processor is a *very* stripped down version of the x86 CPUs. First of all, the Y86 only supports one operand size – 16 bits. This simplification frees us from having to encode the size of the operand as part of the opcode (thereby reducing the total number of opcodes we will need). Another simplification is that the Y86 processor only supports four 16-bit registers: AX, BX, CX, and DX. This lets us encode register operands with only two bits (versus the three bits the 80x86 family requires to encode eight registers). Finally, the Y86 processors only support a 16-bit address bus with a maximum of 65,536 bytes of addressable memory. These simplifications, plus a very limited instruction set will allow us to encode all Y86 instructions using a single byte opcode and a two-byte displacement/offset (if needed).

The Y86 CPU provides 20 instructions. Seven of these instructions have two operands, eight of these instructions have a single operand, and five instructions have no operands at all. The instructions are MOV (two forms), ADD, SUB, CMP, AND, OR, NOT, JE, JNE, JB, JBE, JA, JAE, JMP, BRK, IRET, HALT, GET, and PUT. The following paragraphs describe how each of these work.

The MOV instruction is actually two instruction classes merged into the same instruction. The two forms of the mov instruction take the following forms:

```
mov( reg/memory/constant, reg );
mov( reg, memory );
```

where *reg* is any of AX, BX, CX, or DX; *constant* is a numeric constant (using hexadecimal notation), and *memory* is an operand specifying a memory location. The next section describes the possible forms the

---

5. Actually, Intel claims it's a one byte opcode plus a one-byte "mod-reg-r/m" byte. For our purposes, we'll treat the mod-reg-r/m byte as part of the opcode.

memory operand can take. The “reg/memory/constant” operand tells you that this particular operand may be a register, memory location, or a constant.

The arithmetic and logical instructions take the following forms:

```
add( reg/memory/constant, reg );
sub( reg/memory/constant, reg );
cmp( reg/memory/constant, reg );
and( reg/memory/constant, reg );
or( reg/memory/constant, reg );

not( reg/memory );
```

Note: the NOT instruction appears separately because it is in a different class than the other arithmetic instructions (since it supports only a single operand).

The ADD instruction adds the value of the first operand to the second (register) operand, leaving the sum in the second (register) operand. The SUB instruction subtracts the value of the first operand from the second, leaving the difference in the second operand. The CMP instruction compares the first operand against the second and saves the result of this comparison for use with one of the conditional jump instructions (described in a moment). The AND and OR instructions compute the corresponding bitwise logical operation on the two operands and store the result into the first operand. The NOT instruction inverts the bits in the single memory or register operand.

The *control transfer instructions* interrupt the sequential execution of instructions in memory and transfer control to some other point in memory either unconditionally, or after testing the result of the previous CMP instruction. These instructions include the following:

```
ja  dest;  -- Jump if above (i.e., greater than)
jae dest;  -- Jump if above or equal (i.e., greater than or equal)
jb  dest;  -- Jump if below (i.e., less than)
jbe dest;  -- Jump if below or equal (i.e., less than or equal)
je  dest;  -- Jump if equal
jne dest;  -- Jump if not equal

jmp dest;  -- Unconditional jump

iret;      -- Return from an interrupt
```

The first six instructions let you check the result of the previous CMP instruction for greater than, greater or equal, less than, less or equal, equality, or inequality<sup>6</sup>. For example, if you compare the AX and BX registers with a “cmp( ax, bx );” instruction and execute the JA instruction, the Y86 CPU will jump to the specified destination location if AX was greater than BX. If AX was not greater than BX, control will fall through to the next instruction in the program.

The JMP instruction unconditionally transfers control to the instruction at the destination address. The IRET instruction returns control from an interrupt service routine, which we will discuss later.

The GET and PUT instructions let you read and write integer values. GET will stop and prompt the user for a hexadecimal value and then store that value into the AX register. PUT displays (in hexadecimal) the value of the AX register.

The remaining instructions do not require any operands, they are HALT and BRK. HALT terminates program execution and BRK stops the program in a state that it can be restarted.

The Y86 processors require a unique opcode for every different instruction, not just the instruction classes. Although “mov( bx, ax );” and “mov( cx, ax );” are both in the same class, they must have different opcodes if the CPU is to differentiate them. However, before looking at all the possible opcodes, perhaps it would be a good idea to learn about all the possible operands for these instructions.

---

6. The Y86 processor only performs *unsigned* comparisons.

### 5.4.1 Addressing Modes on the Y86

The Y86 instructions use five different operand types: registers, constants, and three memory addressing schemes. Each form is called an *addressing mode*. The Y86 processor supports the *register* addressing mode<sup>7</sup>, the *immediate* addressing mode, the *indirect* addressing mode, the *indexed* addressing mode, and the *direct* addressing mode. The following paragraphs explain each of these modes.

Register operands are the easiest to understand. Consider the following forms of the MOV instruction:

```
mov( ax, ax );
mov( bx, ax );
mov( cx, ax );
mov( dx, ax );
```

The first instruction accomplishes absolutely nothing. It copies the value from the AX register back into the AX register. The remaining three instructions copy the values of BX, CX and DX into AX. Note that these instructions leave BX, CX, and DX unchanged. The second operand (the destination) is not limited to AX; you can move values to any of these registers.

Constants are also pretty easy to deal with. Consider the following instructions:

```
mov( 25, ax );
mov( 195, bx );
mov( 2056, cx );
mov( 1000, dx );
```

These instructions are all pretty straightforward; they load their respective registers with the specified hexadecimal constant<sup>8</sup>.

There are three addressing modes which deal with accessing data in memory. The following instructions demonstrate the use of these addressing modes:

```
mov( [1000], ax );
mov( [bx], ax );
mov( [1000+bx], ax );
```

The first instruction above uses the direct addressing mode to load AX with the 16 bit value stored in memory starting at location \$1000.

The "mov( [bx], ax );" instruction loads AX from the memory location specified by the contents of the bx register. This is an indirect addressing mode. Rather than using the value in BX, this instruction accesses to the memory location whose address appears in BX. Note that the following two instructions:

```
mov( 1000, bx );
mov( [bx], ax );
```

are equivalent to the single instruction:

```
mov( [1000], ax );
```

Of course, the second sequence is preferable. However, there are many cases where the use of indirection is faster, shorter, and better. We'll see some examples of this a little later.

The last memory addressing mode is the *indexed* addressing mode. An example of this memory addressing mode is

```
mov( [1000+bx], ax );
```

This instruction adds the contents of BX with \$1000 to produce the address of the memory value to fetch. This instruction is useful for accessing elements of arrays, records, and other data structures.

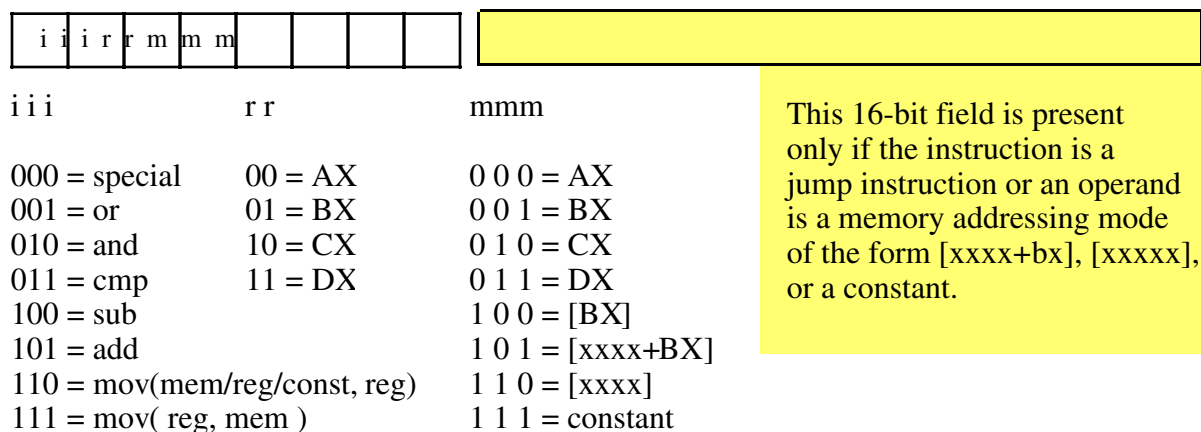
7. Technically, registers do not have an address, but we apply the term *addressing mode* to registers nonetheless.

8. All numeric constants in Y86 assembly language are given in hexadecimal. The "\$" prefix is not necessary.

### 5.4.2 Encoding Y86 Instructions

Although we could arbitrarily assign opcodes to each of the Y86 instructions, keep in mind that a real CPU uses logic circuitry to decode the opcodes and act appropriately on them. A typical CPU opcode uses a certain number of bits in the opcode to denote the instruction class (e.g., MOV, ADD, SUB), and a certain number of bits to encode each of the operands.

A typical Y86 instruction takes the form shown in Figure 5.3. The basic instruction is either one or three bytes long. The instruction opcode consists of a single byte that contains three fields. The first field, the H.O. three bits, defines the instruction. This provides eight combinations. As you may recall, there are 20 different instructions; we cannot encode 20 instructions with three bits, so we'll have to pull some tricks to handle the other instructions. As you can see in Figure 5.3, the basic opcode encodes the MOV instructions (two instructions, one where the *rr* field specifies the destination, one where the *mmm* field specifies the destination), and the ADD, SUB, CMP, AND, and OR instructions. There is one additional instruction field: *special*. The special instruction class provides a mechanism that allows us to expand the number of available instruction classes, we will return to this expansion opcode shortly.

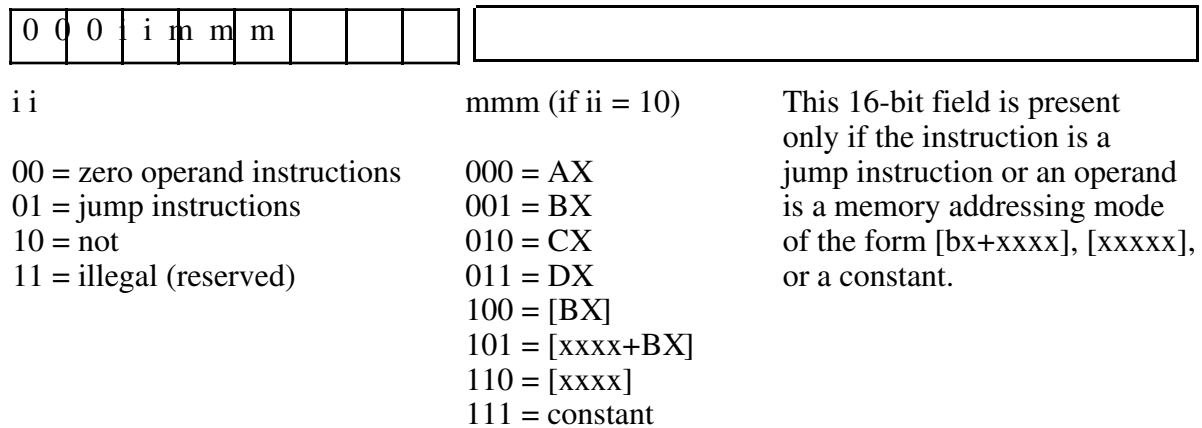


### Figure 5.3 Basic Y86 Instruction Encoding

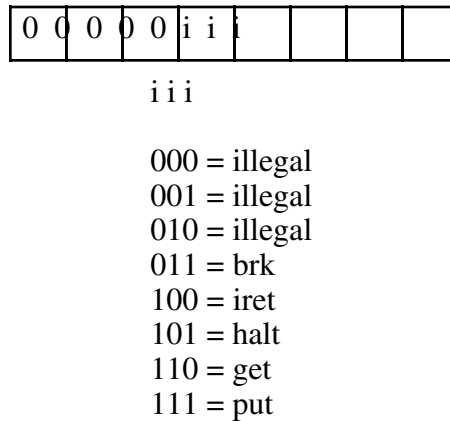
To determine a particular instruction's opcode, you need only select the appropriate bits for the *iii*, *rr*, and *mmm* fields. The *rr* field contains the destination register (except for the MOV instruction whose *iii* field is %111) and the *mmm* field encodes the source operand. For example, to encode the "mov( bx, ax );" instruction you would select *iii*=110 ("mov( reg, reg );"), *rr*=00 (AX), and *mmm*=001 (BX). This produces the one-byte instruction %11000001 or \$C0.

Some Y86 instructions require more than one byte. For example, the instruction "mov( [1000], ax );" loads the AX register from memory location \$1000. The encoding for the opcode is %11000110 or \$C6. However, the encoding for the "mov( [2000], ax );" instruction's opcode is also \$C6. Clearly these two instructions do different things, one loads the AX register from memory location \$1000 while the other loads the AX register from memory location \$2000. To encode an address for the [xxxx] or [xxxx+bx] addressing modes, or to encode the constant for the immediate addressing mode, you must follow the opcode with the 16-bit address or constant, with the L.O. byte immediately following the opcode in memory and the H.O. byte after that. So the three byte encoding for "mov( [1000], ax );" would be \$C6, \$00, \$10 and the three byte encoding for "mov( [2000], ax );" would be \$C6, \$00, \$20.

The *special* opcode allows the x86 CPU to expand the set of available instructions. This opcode handles several zero and one-operand instructions as shown in Figure 5.4 and Figure 5.5.

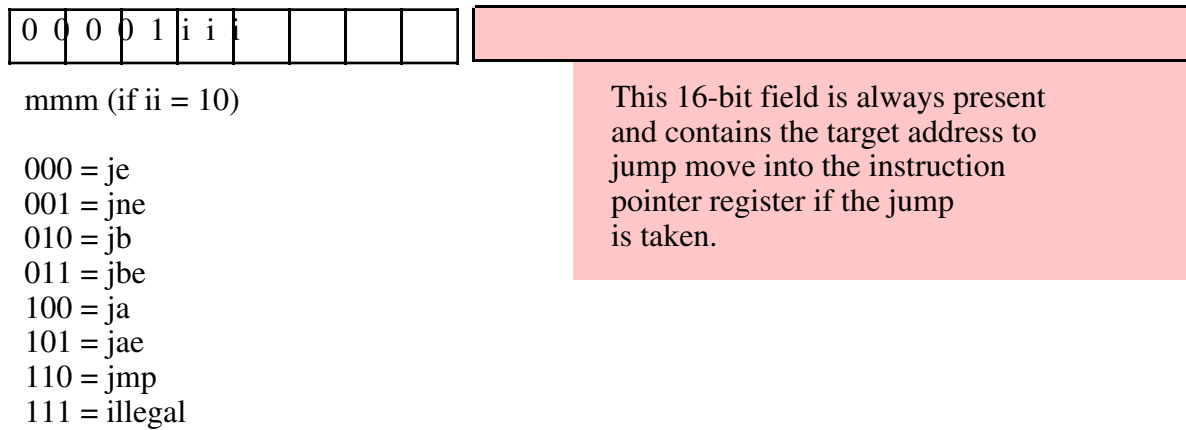


**Figure 5.4 Single Operand Instruction Encodings**



**Figure 5.5 Zero Operand Instruction Encodings**

There are four one-operand instruction classes. The first encoding (00) further expands the instruction set with a set of zero-operand instructions (see Figure 5.5). The second opcode is also an expansion opcode that provides all the Y86 *jump* instructions (see Figure 5.6). The third opcode is the NOT instruction. This is the bitwise logical not operation that inverts all the bits in the destination register or memory operand. The fourth single-operand opcode is currently unassigned. Any attempt to execute this opcode will halt the processor with an illegal instruction error. CPU designers often reserve unassigned opcodes like this one to extend the instruction set at a future date (as Intel did when moving from the 80286 processor to the 80386).



**Figure 5.6**      **Jump Instruction Encodings**

There are seven jump instructions in the x86 instruction set. They all take the following form:

`jxx      address;`

The JMP instruction copies the 16-bit value (address) following the opcode into the IP register. Therefore, the CPU will fetch the next instruction from this target address; effectively, the program “jumps” from the point of the JMP instruction to the instruction at the target address.

The JMP instruction is an example of an unconditional jump instruction. It always transfers control to the target address. The remaining six instructions are conditional jump instructions. They test some condition and jump if the condition is true; they fall through to the next instruction if the condition is false. These six instructions, JA, JAE, JB, JBE, JE, and JNE let you test for greater than, greater than or equal, less than, less than or equal, equality, and inequality. You would normally execute these instructions immediately after a CMP instruction since it sets the less than and equality flags that the conditional jump instructions test. Note that there are eight possible jump opcodes, but the x86 uses only seven of them. The eighth opcode is another illegal opcode.

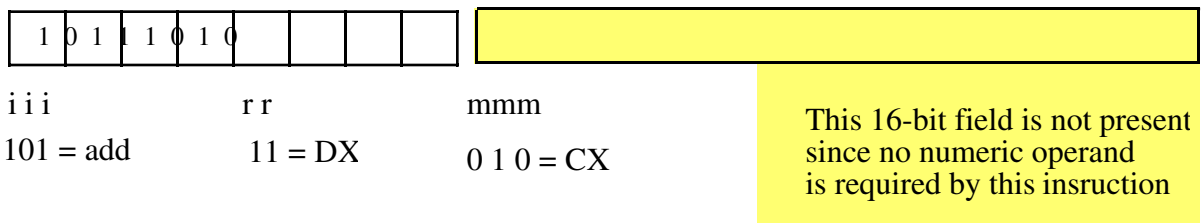
The last group of instructions, the zero operand instructions, appear in Figure 5.5. Three of these instructions are illegal instruction opcodes. The BRK (break) instruction pauses the CPU until the user manually restarts it. This is useful for pausing a program during execution to observe results. The IRET (interrupt return) instruction returns control from an interrupt service routine. We will discuss interrupt service routines later. The HALT program terminates program execution. The GET instruction reads a hexadecimal value from the user and returns this value in the AX register; the PUT instruction outputs the value in the AX register.

### 5.4.3 Hand Encoding Instructions

Keep in mind that the Y86 processor fetches instructions as bit patterns from memory. It decodes and executes those bit patterns. The processor does not execute instructions of the form “mov( ax, bx );” (that is, a string of characters that are readable by humans. Instead, it executes the bit pattern \$C1 from memory. Instructions like “mov( ax, bx );” and “add( 5, cx );” are human-readable representations of these instructions that we must first convert into *machine code* (that is, the binary representation of the instruction that the machine actually executes). In this section we will explore how to manually accomplish this task.

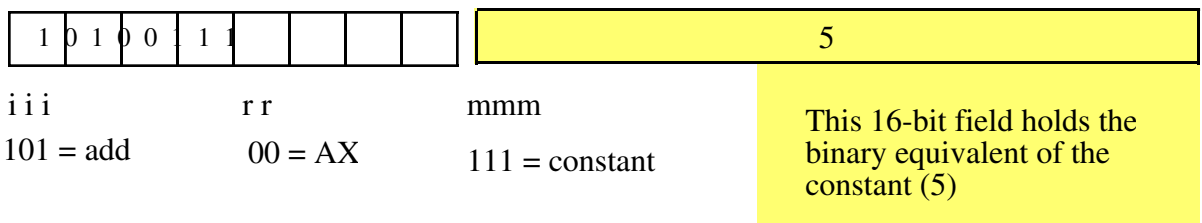
The first step is to choose an instruction to convert into machine code. We’ll start with a very simple example, the “add( cx, dx );” instruction. Once you’ve chosen the instruction, you look up the instruction in one of the figures of the previous section. The ADD instruction is in the first group (see Figure 5.3) and has

an *iii* field of %101. The source operand is CX, so the *mmm* field is %010 and the destination operand is DX so the *rr* field is %11. Merging these bits produces the opcode %10111010 or \$BA.



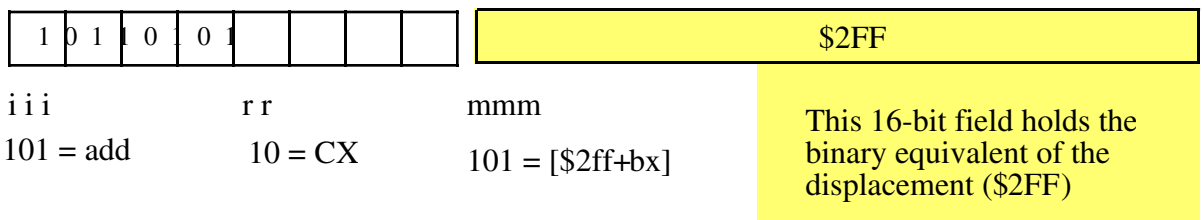
**Figure 5.7** Encoding ADD( *cx*, *dx* );

Now consider the "add( 5, ax );" instruction. Since this instruction has an immediate source operand, the *mmm* field will be %111. The destination register operand is AX (%00) so the full opcode becomes \$10100111 or \$A7. Note, however, that this does not complete the encoding of the instruction. We also have to include the 16-bit constant \$0005 as part of the instruction. The binary encoding of the constant must immediately follow the opcode in memory, so the sequence of bytes in memory (from lowest address to highest address) is \$A7, \$05, \$00. Note that the L.O. byte of the constant follows the opcode and the H.O. byte of the constant follows the L.O. byte. This sequence appears backwards because the bytes are arranged in order of increasing memory address and the H.O. byte of a constant always appears in the highest memory address.



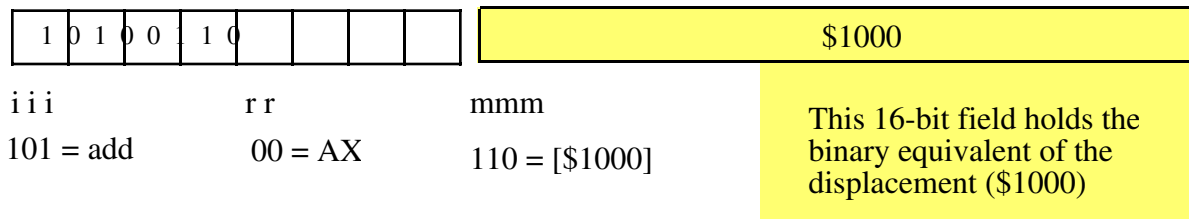
**Figure 5.8** Encoding ADD( 5, *ax* );

The "add( [2ff+bx], *cx* );" instruction also contains a 16-bit constant associated with the instruction's encoding – the displacement portion of the indexed addressing mode. To encode this instruction we use the following field values: *iii*=%101, *rr*=%10, and *mmm*=%101. This produces the opcode byte %10110101 or \$B5. The complete instruction also requires the constant \$2FF so the full instruction is the three-byte sequence \$B5, \$FF, \$02.



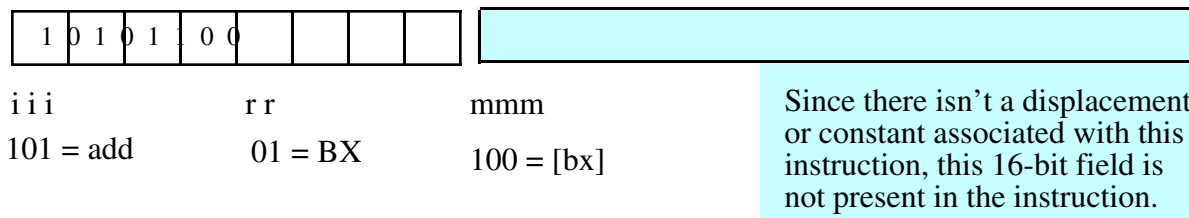
**Figure 5.9** Encoding ADD( [2ff+bx], *cx* );

Now consider the "add([1000], ax);" instruction. This instruction adds the 16-bit contents of memory locations \$1000 and \$1001 to the value in the AX register. Once again, *iii*=%101 for the ADD instruction. The destination register is AX so *rr*=%00. Finally, the addressing mode is the displacement-only addressing mode, so *mmm*=%110. This forms the opcode %10100110 or \$A6. The instruction is three bytes long since it must encode the displacement (address) of the memory location in the two bytes following the opcode. Therefore, the complete three-byte sequence is \$A6, \$00, \$10.



**Figure 5.10**      **Encoding `ADD( [1000], ax );`**

The last addressing mode to consider is the register indirect addressing mode, [bx]. The "add([bx], bx);" instruction uses the following encoded values: *mmm*=%101, *rr*=%01 (bx), and *mmm*=%100 ([bx]). Since the value in the BX register completely specifies the memory address, there is no need for a displacement field. Hence, this instruction is only one byte long.



### Figure 5.11 Encoding the ADD( [bx], bx ); Instruction

You use a similar approach to encode the SUB, CMP, AND, and OR instructions as you do the ADD instruction. The only difference is that you use different values for the *iii* field in the opcode.

The MOV instruction is special because there are two forms of the MOV instruction. You encode the first form (*iii=%110*) exactly as you do the ADD instruction. This form copies a constant or data from memory or a register (the *mmm* field) into a destination register (the *rr* field).

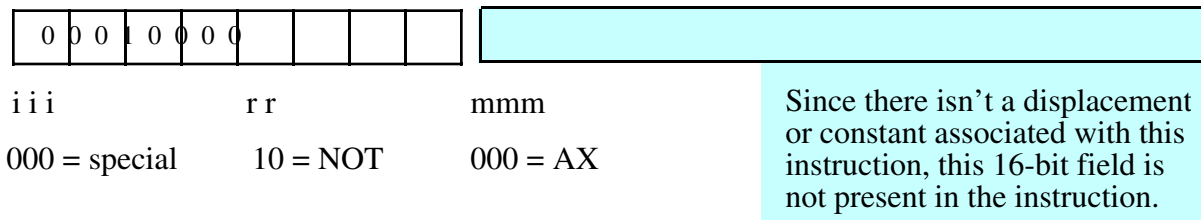
The second form of the MOV instruction (*iii=%111*) copies data from a source register (*rr*) to a destination memory location (that the *mmm* field specifies). In this form of the MOV instruction, the source/destination meanings of the *rr* and *mmm* fields are reversed so that *rr* is the source field and *mmm* is the destination field. Another difference is that the *mmm* field may only contain the values %100 ([bx]), %101 ([disp+bx]), and %110 ([disp]). The destination values cannot be %000..%011 (registers) or %111 (constant). These latter five encodings are illegal (the register destination instructions are handled by the other MOV instruction and storing data into a constant doesn't make any sense).

The Y86 processor supports a single instruction with a single memory/register operand – the NOT instruction. The NOT instruction has the syntax: "not( reg );" or "not( mem );" where mem represents one of the memory addressing modes ([bx], [disp+bx], or [disp]). Note that you may not specify a constant as the operand of the NOT instruction.

Since the NOT instruction has only a single operand, it only uses the mmm field to encode this operand. The *rr* field, combined with the *iii* field, selects the NOT instruction (*iii*=%000 and *rr*=%10). Whenever the

*iii* field contains zero this tells the CPU that special decoding is necessary for the instruction. In this case, the *rr* field specifies whether we have the NOT instruction or one of the other specially decoded instructions.

To encode an instruction like "not( ax );" you would simply specify %000 for *iii* and %10 for the *rr* fields. Then you would encode the *mmm* field the same way you would encode this field for the ADD instruction. Since *mmm*=%000 for AX, the encoding of "not( ax );" would be %00010000 or \$10.



**Figure 5.12    Encoding the NOT( ax ); Instruction**

The NOT instruction does not allow an immediate (constant) operand, hence the opcode %00010111 (\$17) is an illegal opcode.

The Y86 conditional jump instructions also use a special encoding. These instructions are always three bytes long. The first byte (the opcode) specifies which conditional jump instruction to execute and the next two bytes specify where the CPU transfers if the condition is met. There are seven different Y86 jump instructions, six conditional jumps and one unconditional jump. These instructions set *mmm*=%000, *rr*=%01, and use the *mmm* field to select one of the seven possible jumps; the eighth possible opcode is an illegal opcode (see Figure 5.6). Encoding these instructions is relatively straight-forward. Once you pick the instruction you want to encode, you've determined the opcode (since there is a single opcode for each instruction). The opcode values fall in the range \$08..\$0E (\$0F is the illegal opcode).

The only field that requires some thought is the 16-bit operand that follows the opcode. This field holds the address of the target instruction to which the (un)conditional jump transfers if the condition is true (e.g., JE transfers control to this address if the previous CMP instruction found that its two operands were equal). To properly encode this field you must know the address of the opcode byte of the target instruction. If you've already converted the instruction to binary form and stored it into memory, this isn't a problem; just specify the address of that instruction as the operand of the condition jump. On the other hand, if you haven't yet written, converted, and placed that instruction into memory, knowing its address would seem to require a bit of divination. Fortunately, you can figure out the target address by computing the lengths of all the instructions between the current jump instruction you're encoding and the target instruction. Unfortunately, this is an arduous task. The best solution is to write all your instructions down on paper, compute their lengths (which is easy, all instructions are one or three bytes long depending on the presence of a 16-bit operand), and then assign an appropriate address to each instruction. Once you've done this (and, assuming you haven't made any mistakes) you'll know the starting address for each instruction and you can fill in target address operands in your (un)conditional jump instructions as you encode them. Fortunately, there is a better way to do this, as you'll see in the next section.

The last group of instructions, the zero operand instructions, are the easiest to encode. Since they have no operands they are always one byte long and the instruction uniquely specifies the opcode for the instruction. These instructions always have *iii*=%000, *rr*=%00, and *mmm* specifies the particular instruction opcode (see Figure 5.5). Note that the Y86 CPU leaves three of these instructions undefined (so we can use these opcodes for future expansion).

#### 5.4.4    Using an Assembler to Encode Instructions

Of course, hand coding machine language programs as demonstrated in the previous section is impractical for all but the smallest programs. Certainly you haven't had to do anything like this when writing HLA

programs. The HLA compiler lets you create a text file containing human readable forms of the instructions. You might wonder why we can write such code for the 80x86 but not for the Y86. The answer is to use an assembler or compiler for the Y86. The job of an assembler/compiler is to read a text file containing human readable text and translate that text into the binary encoded representation for the corresponding machine language program.

An assembler or compiler is nothing special. It's just another program that executes on your computer system. The only thing special about an assembler or compiler is that it translates programs from one form (source code) to another (machine code). A typical Y86 assembler, for example, would read lines of text with each line containing a Y86 instruction, it would *parse*<sup>9</sup> each statement and then write the binary equivalent of each instruction to memory or to a file for later execution. In the laboratory exercises associated with this chapter, you'll get the opportunity to use a simple Y86 assembler to translate Y86 source programs into Y86 machine code.

Assemblers have two big advantages over coding in machine code. First, they automatically translate strings like "ADD( ax, bx );" and "MOV( ax, [1000]);" to their corresponding binary form. Second, and probably even more important, assemblers let you attach labels to statements and refer to those labels within jump instructions; this means that you don't have to know the target address of an instruction in order to specify that instruction as the target of a jump or conditional jump instruction. The laboratory exercises associated with this chapter provide a very simple Y86 assembler that lets you specify up to 26 labels in a program (using the symbols 'A'..'Z'). To attach a label to a statement, you simply preface the instruction with the label and a colon, e.g.,

```
L: mov( 0, ax );
```

To transfer control to a statement with a label attached to it, you simply specify the label name as the operand of the jump instruction, e.g.,

```
jmp L;
```

The assembler will compute the address of the label and fill in the address for you whenever you specify the label as the operand of a jump or conditional jump instruction. The assembler can do this even if it hasn't yet encountered the label in the program's source file (i.e., the label is attached to a later instruction in the source file). Most assemblers accomplish this magic by making two passes over the source file. During the first pass the assembler determines the starting address of each symbol and stores this information in a simple database called the *symbol table*. The assembler does not emit any machine code during this first pass. Then the assembler makes a second pass over the source file and actually emits the machine code. During this second pass it looks up all label references in the symbol table and uses the information it retrieves from this database to fill in the operand fields of the instructions that refer to some symbol.

---

### 5.4.5 Extending the Y86 Instruction Set

The Y86 CPU is a trivial CPU, suitable only for demonstrating how to encode machine instructions. However, like any good CPU the Y86 design does provide the capability for expansion. So if you wanted to improve the CPU by adding new instructions, the ability to accomplish this exists in the instruction set.

There are two standard ways to increase the number of instructions in a CPU's instruction set. Both mechanisms require the presence of undefined (or illegal) opcodes on the CPU. Since the Y86 CPU has several of these, we can expand the instruction set.

The first method is to directly use the undefined opcodes to define new instructions. This works best when there are undefined bit patterns within an opcode group and the new instruction you want to add falls into that same group. For example, the opcode %00011mmm falls into the same group as the NOT instruction. If you decided that you really needed a NEG (negate, take the two's complement) instruction, using this particular opcode for this purpose makes a lot of sense because you'd probably expect the NEG instruction to use the same syntax (and, therefore, decoding) as the NOT instruction.

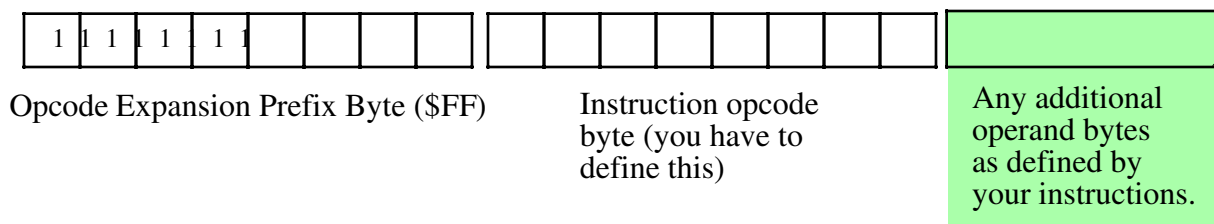
---

9. "Parse" means to figure out the meaning of the statement.

Likewise, if you want to add a zero-operand instruction to the instruction set, there are three undefined zero-operand instructions that you could use for this purpose. You'd just appropriate one of these opcodes and assign your instruction to it.

Unfortunately, the Y86 CPU doesn't have that many illegal opcodes open. For example, if you wanted to add the SHL, SHR, ROL, and ROR instructions (shift and rotate left and right) as single-operand instructions, there is insufficient space in the single operand instruction opcodes to add these instructions (there is currently only one open opcode you could use). Likewise, there are no two-operand opcodes open, so if you wanted to add an XOR instruction or some other two-operand instruction, you'd be out of luck.

A common way to handle this dilemma (one the Intel designers have employed) is to use a prefix opcode byte. This opcode expansion scheme uses one of the undefined opcodes as an opcode prefix byte. Whenever the CPU encounters a prefix byte in memory, it reads and decodes the next byte in memory as the actual opcode. However, it does not treat this second byte as it would any other opcode. Instead, this second opcode byte uses a completely different encoding scheme and, therefore, lets you specify as many new instructions as you can encode in that byte (or bytes, if you prefer). For example, the opcode \$FF is illegal (it corresponds to a "mov( dx, const );" instruction) so we can use this byte as a special prefix byte to further expand the instruction set<sup>10</sup>.



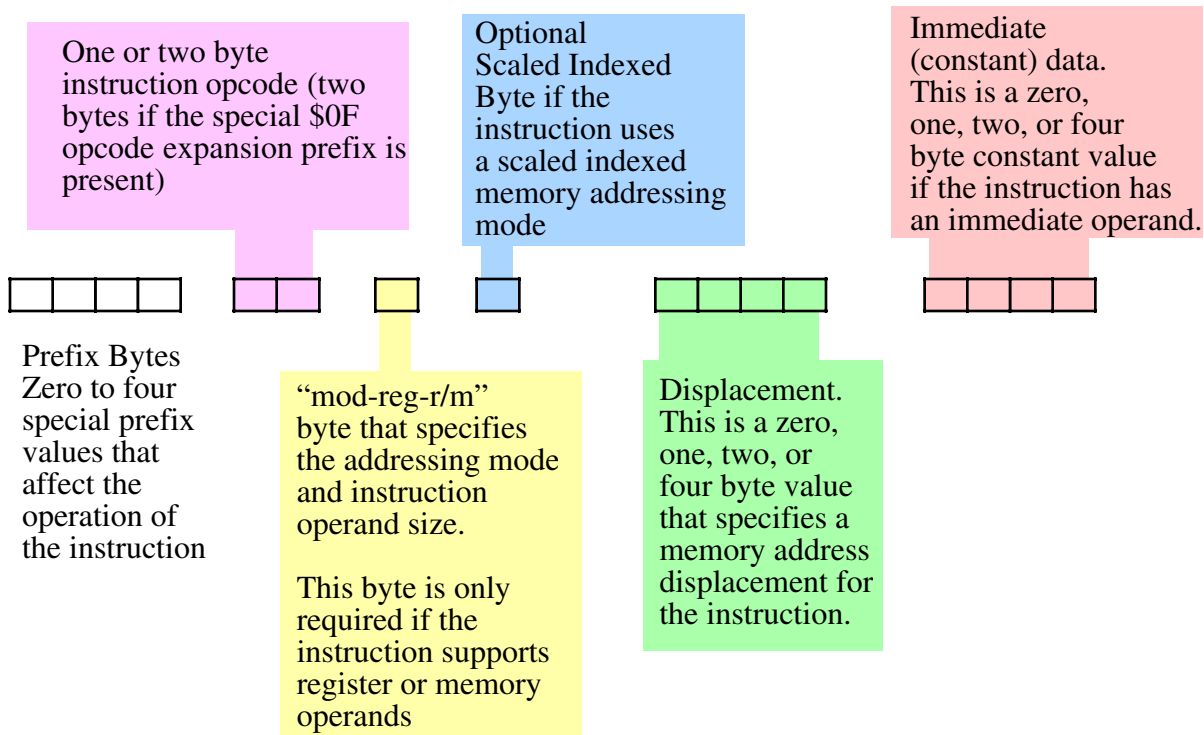
### Figure 5.13 Using a Prefix Byte to Extend the Instruction Set

## 5.5 Encoding 80x86 Instructions

The Y86 processor is simple to understand, easy to hand encode instructions for it, and a great vehicle for learning how to assign opcodes. It's also a purely hypothetical device intended only as a teaching tool. Therefore, you can now forget all about the Y86, it's served its purpose. Now it's time to take a look at the actual machine instruction format for the 80x86 CPU family.

They don't call the 80x86 CPU a *Complex* Instruction Set Computer for nothing. Although more complex instruction encodings do exist, no one is going to challenge the assertion that the 80x86 has a complex instruction encoding. The generic 80x86 instruction takes the form shown in Figure 5.14. Although this diagram seems to imply that instructions can be up to 16 bytes long, in actuality the 80x86 will not allow instructions greater than 15 bytes in length.

10. We could also have used values \$F7, \$EF, and \$E7 since they also correspond to an attempt to store a register into a constant. However, \$FF is easier to decode. On the other hand, if you need even more prefix bytes for instruction expansion, you can use these three values as well.



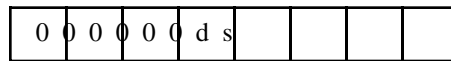
**Figure 5.14 80x86 Instruction Encoding**

The prefix bytes are not the "opcode expansion prefix" that the previous sections in this chapter discussed. Instead, these are special bytes to modify the behavior of existing instructions (rather than define new instructions). We'll take a look at a couple of these prefix bytes in a little bit, others we'll leave for discussion in later chapters. The 80x86 certainly supports more than four prefix values, however, an instruction may have a maximum of four prefix bytes attached to it. Also note that the behavior of many prefix bytes are mutually exclusive and the results are undefined if you put a pair of mutually exclusive prefix bytes in front of an instruction.

The 80x86 supports two basic opcode sizes: a standard one-byte opcode and a two-byte opcode consisting of a \$0F opcode expansion prefix byte and a second byte specifying the actual instruction. One way to view these opcode bytes is as an eight-bit extension of the *iii* field in the Y86 encoding. This provides for up to 512 different instruction classes (although the 80x86 does not yet use them all). In reality, various instruction classes use certain bits in this opcode for decidedly non-instruction-class purposes. For example, consider the ADD instruction opcode. It takes the form shown in Figure 5.15.

Note that bit number zero specifies the size of the operands the ADD instruction operates upon. If this field contains zero then the operands are eight bit registers and memory locations. If this bit contains one then the operands are either 16-bits or 32-bits. Under Win32 the default is 32-bit operands if this field contains a one. To specify a 16-bit operand (under Win32) you must insert a special "operand-size prefix byte" in front of the instruction.

Bit number one specifies the direction of the transfer. If this bit is zero, then the destination operand is a memory location (e.g., "add( al, [ebx]);" If this bit is one, then the destination operand is a register (e.g., "add( [ebx], al );" You'll soon see that this direction bit creates a problem that results in one instruction have two different possible opcodes.



ADD opcode.

d = 0 if adding from register to memory.

d = 1 if adding from memory to register.

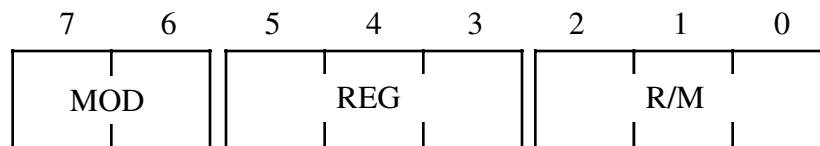
s = 0 if adding eight-bit operands.

s = 1 if adding 16-bit or 32-bit operands

**Figure 5.15 80x86 ADD Opcode**

### 5.5.1 Encoding Instruction Operands

The "mod-reg-r/m" byte (in Figure 5.14) specifies a basic addressing mode. This byte contains the following fields:



**Figure 5.16 MOD-REG-R/M Byte**

The REG field specifies an 80x86 register. Depending on the instruction, this can be either the source or the destination operand. Many instructions have the "d" (direction) field in their opcode to choose whether this operand is the source (d=0) or the destination (d=1) operand. This field is encoded using the bit patterns found in Table 26

**Table 26 REG Field Encoding**

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
%000	al	ax	eax
%001	cl	cx	ecx
%010	dl	dx	edx
%011	bl	bx	ebx
%100	ah	sp	esp
%101	ch	bp	ebp
%110	dh	si	esi

**Table 26 REG Field Encoding**

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
%111	bh	di	edi

For certain (single operand) instructions, the REG field may contain an opcode extension rather than a register value (the R/M field will specify the operand in this case).

The MOD and R/M fields combine to specify the other operand in a two-operand instruction (or the only operand in a single-operand instruction like NOT or NEG). Remember, the "d" bit in the opcode determines which operand is the source and which is the destination. The MOD and R/M fields together specify the following addressing modes:

**Table 27 Meaning of the MOD Field**

MOD	Meaning
%00	Register indirect addressing mode or SIB with no displacement (when R/M=%100) or Displacement only addressing mode (when R/M=%101).
%01	One-byte signed displacement follows addressing mode byte(s).
%10	Four-byte signed displacement follows addressing mode byte(s).
%11	Register addressing mode.

**Table 28 MOD and R/M Addressing Modes**

MOD	R/M	Addressing Mode
%00	%000	[eax]
%01	%000	[eax+disp <sub>8</sub> ]
%10	%000	[eax+disp <sub>32</sub> ]
%11	%000	register (al/ax/eax) <sup>a</sup>
%00	%001	[ecx]
%01	%001	[ecx+disp <sub>8</sub> ]

**Table 28 MOD and R/M Addressing Modes**

MOD	R/M	Addressing Mode
%10	%001	[ecx+disp <sub>32</sub> ]
%11	%001	register (cl/cx/ecx)
%00	%010	[edx]
%01	%010	[edx+disp <sub>8</sub> ]
%10	%010	[edx+disp <sub>32</sub> ]
%11	%010	register (dl/dx/edx)
%00	%011	[ebx]
%01	%011	[ebx+disp <sub>8</sub> ]
%10	%011	[ebx+disp <sub>32</sub> ]
%11	%011	register (bl/bx/ebx)
%00	%100	SIB Mode
%01	%100	SIB + disp <sub>8</sub> Mode
%10	%100	SIB + disp <sub>32</sub> Mode
%11	%100	register (ah/sp/esp)
%00	%101	Displacement Only Mode (32-bit displacement)
%01	%101	[ebp+disp <sub>8</sub> ]
%10	%101	[ebp+disp <sub>32</sub> ]
%11	%101	register (ch/bp/ebp)
%00	%110	[esi]
%01	%110	[esi+disp <sub>8</sub> ]
%10	%110	[esi+disp <sub>32</sub> ]
%11	%110	register (dh/si/esi)
%00	%111	[edi]
%01	%111	[edi+disp <sub>8</sub> ]
%10	%111	[edi+disp <sub>32</sub> ]
%11	%111	register (bh/di/edi)

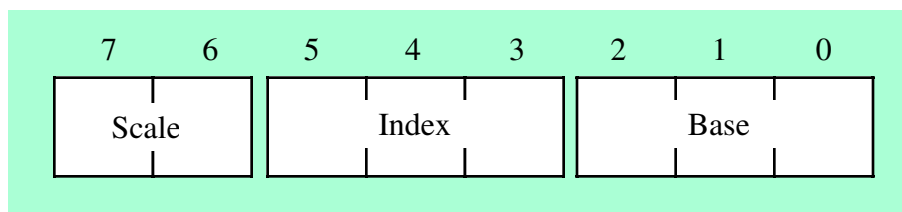
a. The size bit in the opcode specifies eight or 32-bit register size. To select a 16-bit register requires a prefix byte.

There are a couple of interesting things to note about this table. First of all, note that there are two forms of the [reg+disp] addressing modes: one form with an eight-bit displacement and one form with a 32-bit displacement. Addressing modes whose displacement falls in the range -128..+127 require only a single byte displacement after the opcode; hence these instructions will be shorter (and sometimes faster) than

instructions whose displacement value is outside this range. It turns out that many offsets are within this range, so the assembler/compiler can generate shorter instructions for a large percentage of the instructions.

The second thing to note is that there is no [ebp] addressing mode. If you look in the table above where this addressing mode logically belongs, you'll find that it's slot is occupied by the 32-bit displacement only addressing mode. The basic encoding scheme for addressing modes didn't allow for a displacement only addressing mode, so Intel "stole" the encoding for [ebp] and used that for the displacement only mode. Fortunately, anything you can do with the [ebp] addressing mode you can do with the [ebp+disp<sub>8</sub>] addressing mode by setting the eight-bit displacement to zero. True, the instruction is a bit longer, but the capabilities are still there. Intel (wisely) chose to replace this addressing mode because they anticipated that programmers would use this addressing mode less often than the other register indirect addressing modes (for reasons you'll discover in a later chapter).

Another thing you'll notice missing from this table are addressing modes of the form [ebx+edx\*4], the so-called scaled indexed addressing modes. You'll also notice that the table is missing addressing modes of the form [esp], [esp+disp<sub>8</sub>], and [esp+disp<sub>32</sub>]. In the slots where you would normally expect these addressing modes you'll find the SIB (scaled index byte) modes. If these values appear in the MOD and R/M fields then the addressing mode is a scaled indexed addressing mode with a second byte (the SIB byte) following the MOD-REG-R/M byte that specifies the registers to use (note that the MOD field still specifies the displacement size of zero, one, or four bytes). The following diagram shows the layout of this SIB byte and the following tables explain the values for each field.



**Figure 5.17 SIB (Scaled Index Byte) Layout**

**Table 29 SIB Scale Field Values**

Scale Value	Index*Scale Value
%00	Index*1
%01	Index*2
%10	Index*4
%11	Index*8

**Table 30 Index Field Values**

Index	Register
%000	EAX
%001	ECX
%010	EDX

**Table 30 Index Field Values**

Index	Register
%011	EBX
%100	Illegal
%101	EBP
%110	ESI
%111	EDI

**Table 31 Base Field Values**

Base	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
%100	ESP
%101	Displacement-only if MOD = %00, EBP if MOD = %01 or %10
%110	ESI
%111	EDI

The MOD-REG-R/M and SIB bytes are complex and convoluted, no question about that. The reason these addressing mode bytes are so convoluted is because Intel reused their 16-bit addressing circuitry in the 32-bit mode rather than simply abandoning the 16-bit format in the 32-bit mode. There are good hardware reasons for this, but the end result is a complex scheme for specifying addressing modes.

Part of the reason the addressing scheme is so convoluted is because of the special cases for the SIB and displacement-only modes. You will note that the intuitive encoding of the MOD-REG-R/M byte does not allow for a displacement-only mode. Intel added a quick kludge to the addressing scheme replacing the [EBP] addressing mode with the displacement-only mode. Programmers who actually want to use the [EBP] addressing mode have to use [EBP+0] instead. Semantically, this mode produces the same result but the instruction is one byte longer since it requires a displacement byte containing zero.

You will also note that if the REG field of the MOD-REG-R/M byte contains %100 and MOD does not contain %11 then the addressing mode is an "SIB" mode rather than the expected [ESP], [ESP+disp<sub>8</sub>], or [ESP+disp<sub>32</sub>] mode. The SIB mode is used when an addressing mode uses one of the scaled indexed registers, i.e., one of the following addressing modes:

[reg <sub>32</sub> +eax*n]	MOD = %00
[reg <sub>32</sub> +ebx*n]	Note: n = 1, 2, 4, or 8.
[reg <sub>32</sub> +ecx*n]	
[reg <sub>32</sub> +edx*n]	
[reg <sub>32</sub> +ebp*n]	
[reg <sub>32</sub> +esi*n]	

```

[reg32+edi*n]

[disp+reg8+eax*n]      MOD = %01
[disp+reg8+ebx*n]
[disp+reg8+ecx*n]
[disp+reg8+edx*n]
[disp+reg8+ebp*n]
[disp+reg8+esi*n]
[disp+reg8+edi*n]

[disp+reg32+eax*n]      MOD = %10
[disp+reg32+ebx*n]
[disp+reg32+ecx*n]
[disp+reg32+edx*n]
[disp+reg32+ebp*n]
[disp+reg32+esi*n]
[disp+reg32+edi*n]

[disp+eax*n]            MOD = %00 and BASE field contains %101
[disp+ebx*n]
[disp+ecx*n]
[disp+edx*n]
[disp+ebp*n]
[disp+esi*n]
[disp+edi*n]

```

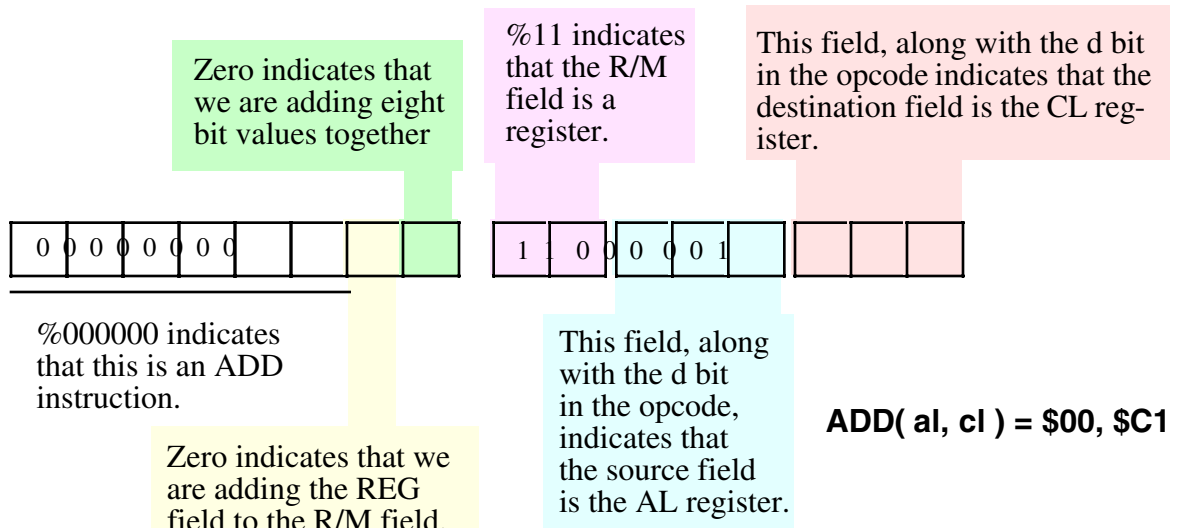
In each of these addressing modes, the MOD field of the MOD-REG-R/M byte specifies the size of the displacement (zero, one, or four bytes). This is indicated in Table 28 via the modes "SIB Mode," "SIB + disp<sub>8</sub> Mode," and "SIB + disp<sub>32</sub> Mode." The Base and Index fields of the SIB byte select the base and index registers, respectively. Note that this addressing mode does not allow the use of the ESP register as an index register. Presumably, Intel left this particular mode undefined to provide the ability to extend the addressing modes in a future version of the CPU (although extending the addressing mode sequence to three bytes seems a bit extreme).

Like the MOD-REG-R/M encoding, the SIB format redefines the [EBP+index\*scale] mode as a displacement plus index mode. Once again, if you really need this addressing mode, you will have to use a single byte displacement value containing zero to achieve the same result.

---

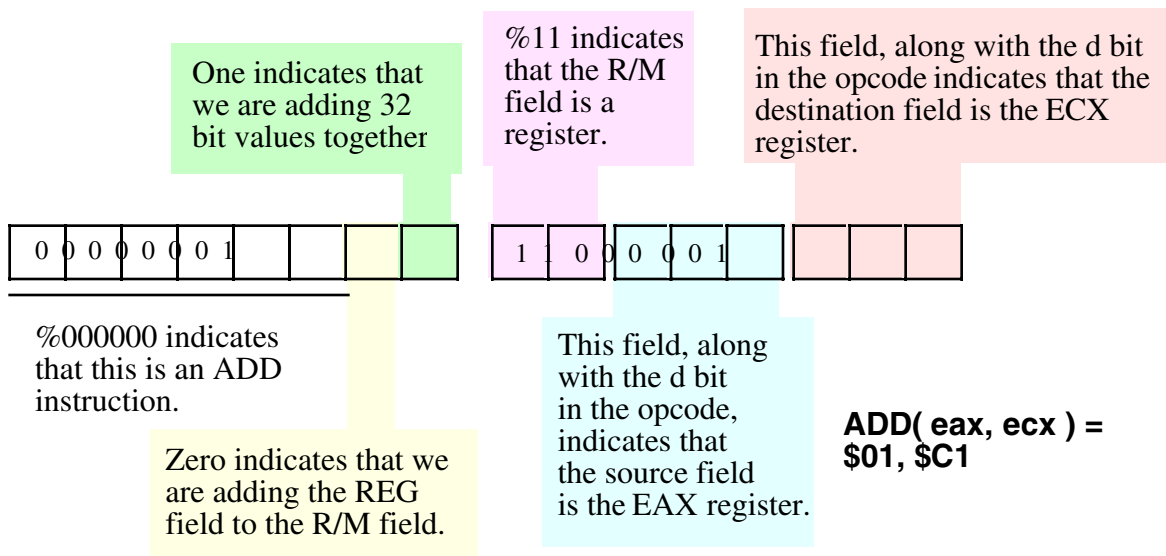
### 5.5.2 Encoding the ADD Instruction: Some Examples

To figure out how to encode an instruction using this complex scheme, some examples are warranted. So let's take a look at how to encode the 80x86 ADD instruction using various addressing modes. The ADD opcode is \$00, \$01, \$02, or \$03, depending on the direction and size bits in the opcode (see Figure 5.15). The following figures each describe how to encode various forms of the ADD instruction using different addressing modes.



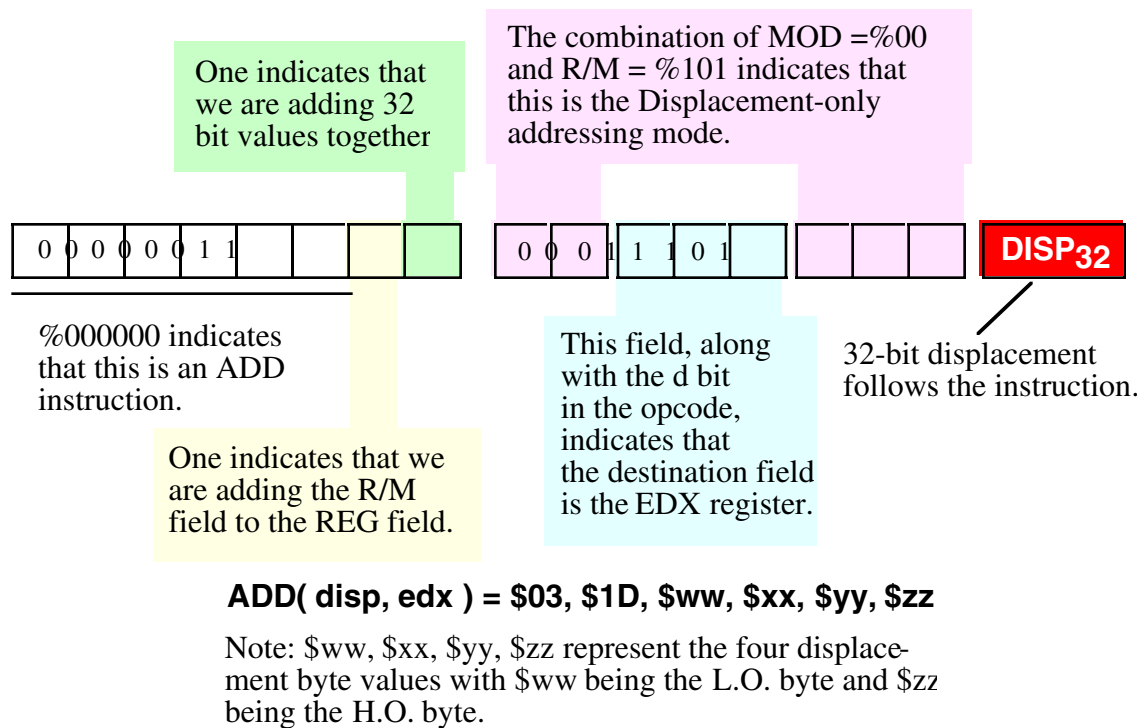
**Figure 5.18** Encoding the **ADD( al, cl );** Instruction

There is an interesting side effect of the operation of the direction bit and the MOD-REG-R/M organization: some instructions have two different opcodes (and both are legal). For example, we could encode the "add( al, cl );" instruction from Figure 5.18 as \$02, \$C8 by reversing the AL and CL registers in the REG and R/M fields and then setting the *d* bit in the opcode (bit #1). This issue applies to instructions with two register operands.

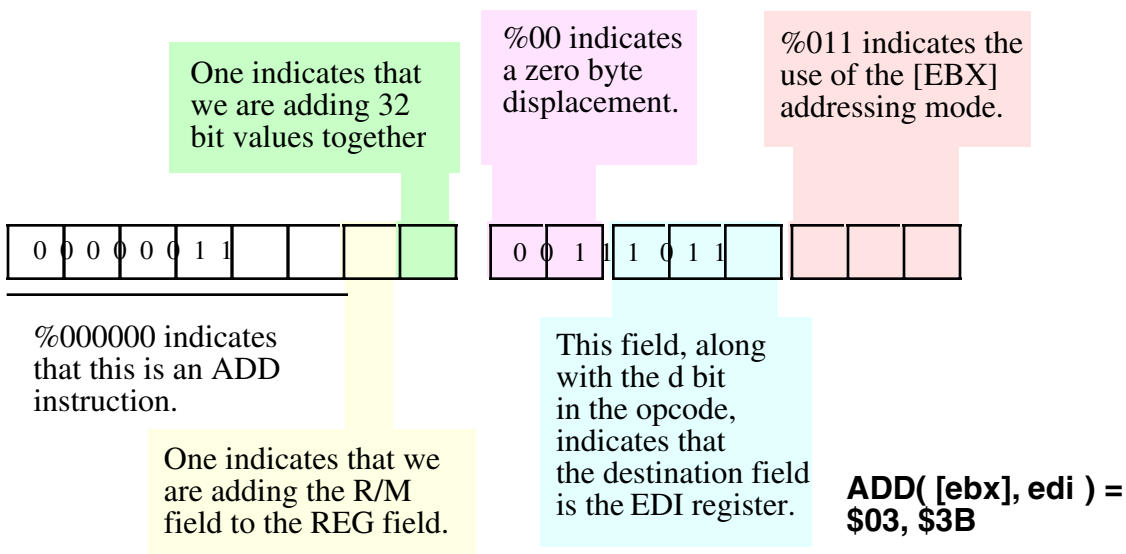


**Figure 5.19** Encoding the **ADD( eax, ecx );** instruction

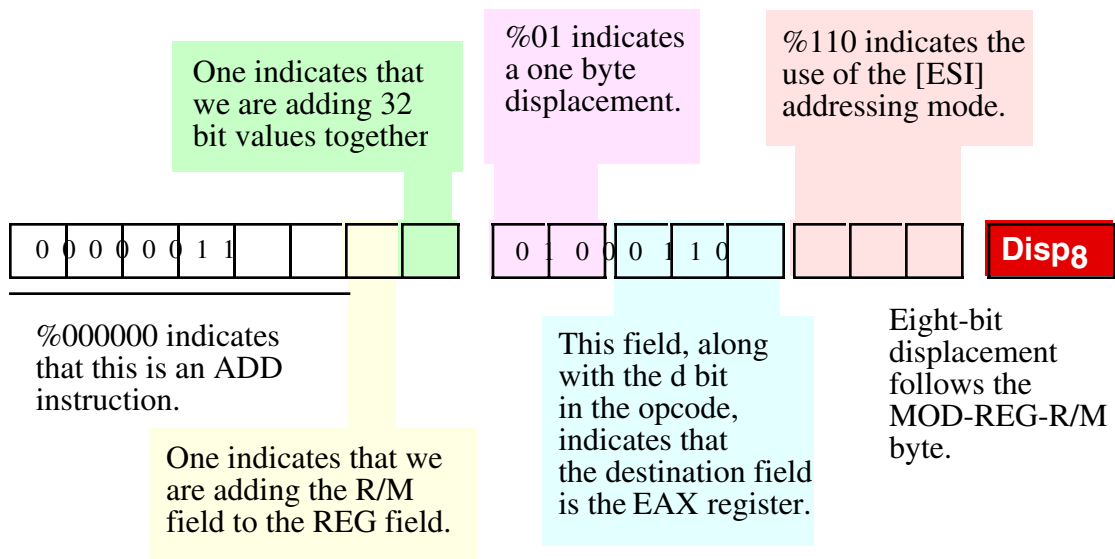
Note that we can also encode "add( eax, ecx );" using the bytes \$03, \$C8.



**Figure 5.20** Encoding the ADD( disp, edx ); Instruction

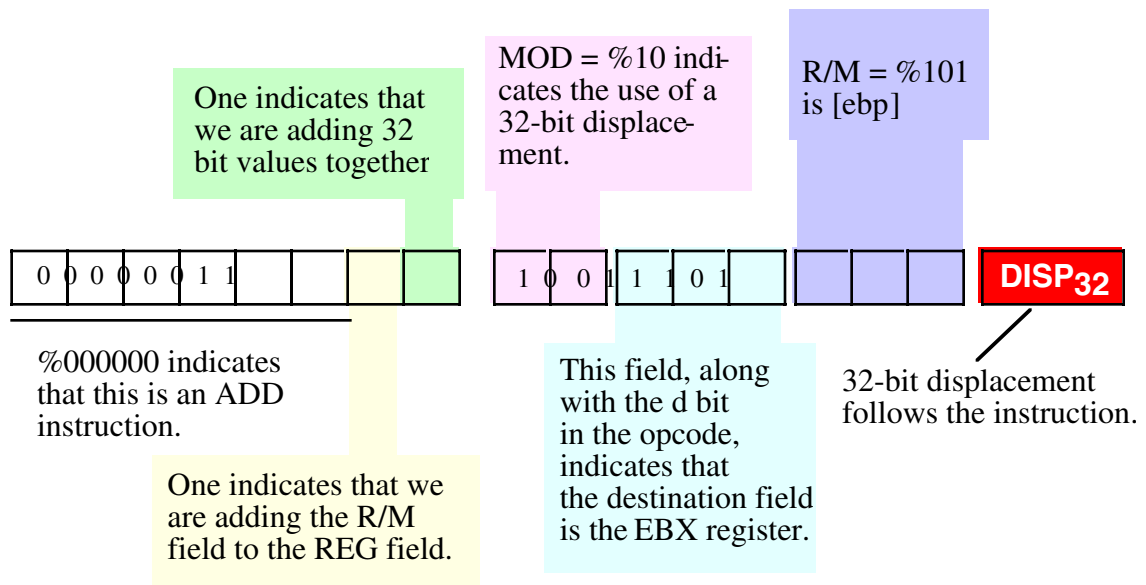


**Figure 5.21** Encoding the ADD( [ebx], edi ); Instruction



**ADD ( [esi + disp8], eax ) = \$03, \$46, \$xx**

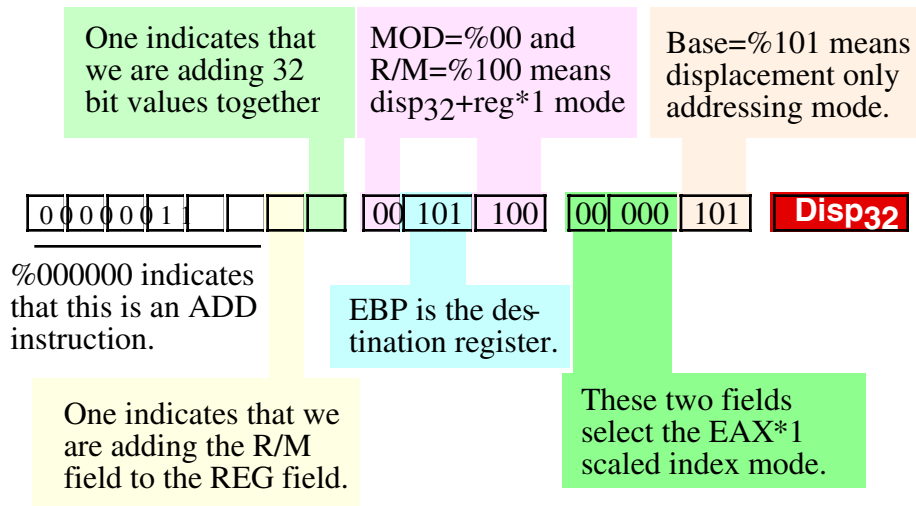
**Figure 5.22** Encoding the ADD( [esi+disp8], eax ); Instruction



**ADD( [ebp+disp32], ebx ) = \$03, \$9D, \$ww, \$xx, \$yy, \$zz**

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values with \$ww being the L.O. byte and \$zz being the H.O. byte.

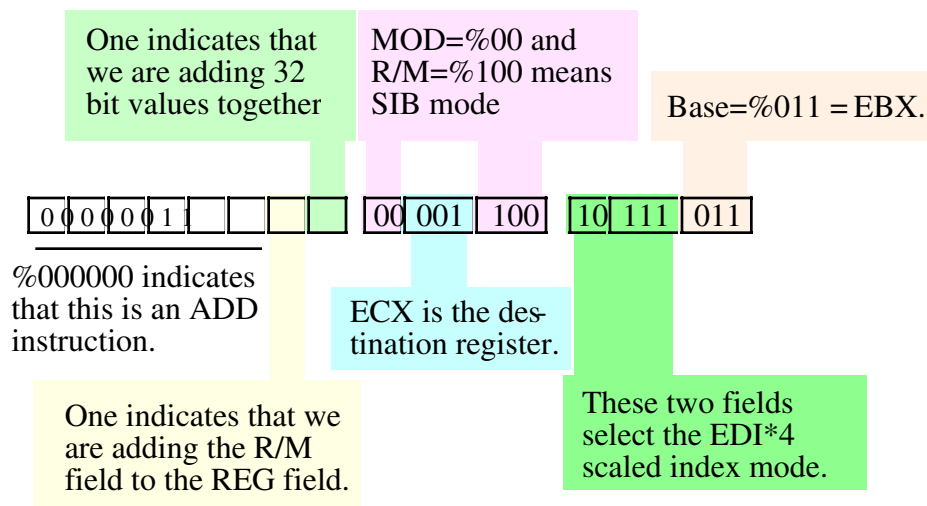
**Figure 5.23** Encoding the ADD ( [ebp+disp32], ebx); Instruction



**ADD ( [disp<sub>32</sub> + eax\*1], ebp ) = \$03, \$2C, \$05, \$ww, \$xx, \$yy, \$zz**

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values with \$ww being the L.O. byte and \$zz being the H.O. byte.

**Figure 5.24** Encoding the ADD( [disp<sub>32</sub> +eax\*1], ebp ); Instruction

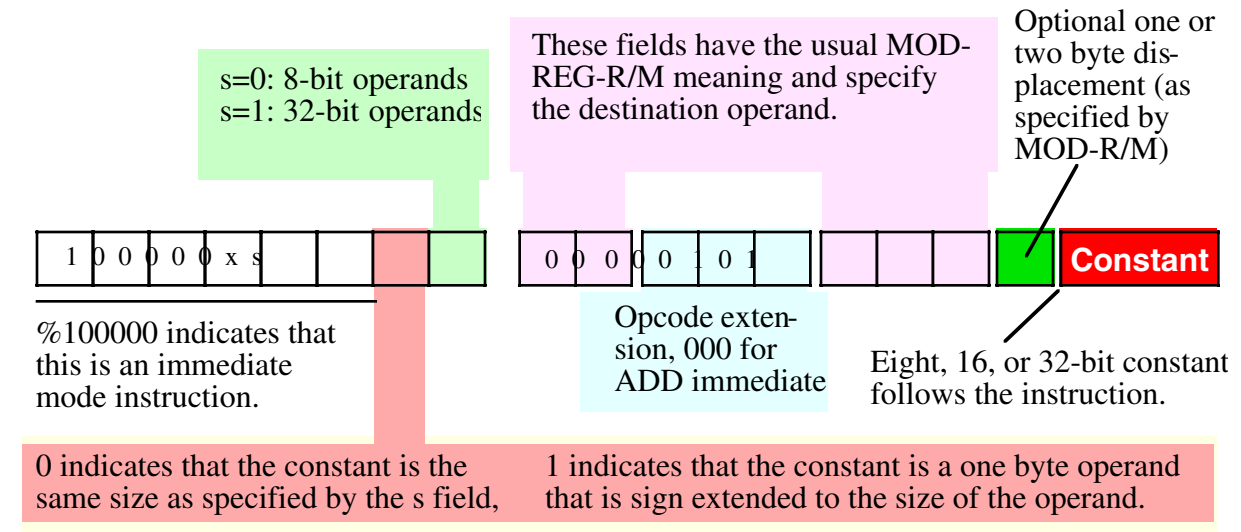


**ADD ( [ebx+ edi\*4], ecx ) = \$03, \$0C, \$BB**

**Figure 5.25** Encoding the ADD( [ebx + edi \* 4], ecx ); Instruction

### 5.5.3 Encoding Immediate Operands

You may have noticed that the MOD-REG-R/M and SIB bytes don't contain any bit combinations you can use to specify an immediate operand. The 80x86 uses a completely different opcode to specify an immediate operand. Figure 5.26 shows the basic encoding for an ADD immediate instruction.



**Figure 5.26** Encoding an ADD Immediate Instruction

There are three major differences between the encoding of the ADD immediate and the standard ADD instruction. First, and most important, the opcode has a one in the H.O. bit position. This tells the CPU that the instruction has an immediate constant. This individual change, however, does not tell the CPU that it must execute an ADD instruction, as you'll see momentarily.

The second difference is that there is no direction bit in the opcode. This makes sense because you cannot specify a constant as a destination operand. Therefore, the destination operand is always the location the MOD and R/M bits specify in the MOD-REG-R/M field.

In place of the direction bit, the opcode has a sign extension (*x*) bit. For eight-bit operands, the CPU ignores this bit. For 16-bit and 32-bit operands, this bit specifies the size of the constant following the ADD instruction. If this bit contains zero then the constant is the same size as the operand (i.e., 16 or 32 bits). If this bit contains one then the constant is a signed eight-bit value and the CPU sign extends this value to the appropriate size before adding it to the operand. This little trick often makes programs quite a bit shorter because one commonly adds small valued constants to 16 or 32 bit operands.

The third difference between the ADD immediate and the standard ADD instruction is the meaning of the REG field in the MOD-REG-R/M byte. Since the instruction implies that the source operand is a constant and the MOD-R/M fields specify the destination operand, the instruction does not need to use the REG field to specify an operand. Instead, the 80x86 CPU uses these three bits as an opcode extension. For the ADD immediate instruction these three bits must contain zero (other bit patterns would correspond to a different instruction).

Note that when adding a constant to a memory location, the displacement (if any) associated with the memory location immediately precedes the immediate (constant) data in the opcode sequence.

---

### 5.5.4 Encoding Eight, Sixteen, and Thirty-Two Bit Operands

When Intel designed the 8086 they used one bit (*s*) to select between eight and sixteen bit integer operand sizes in the opcode. Later, when they extended the 80x86 architecture to 32 bits with the introduction of the 80386, they had a problem, with this single bit they could only encode two sizes but they needed to encode three (8, 16, and 32 bits). To solve this problem, they used a *operand size prefix byte*.

Intel studied their instruction set and came to the conclusion that in a 32-bit environment, programs were more likely to use eight-bit and 32-bit operands far more often than 16-bit operands. So Intel decided to let the size bit (*s*) in the opcode select between eight and thirty-two bit operands, as the previous sections describe. Although modern 32-bit programs don't use 16-bit operands that often, they do need them now and then. To allow for 16-bit operands, Intel lets you prefix a 32-bit instruction with the operand size prefix byte, whose value is \$66. This prefix byte tells the CPU to operand on 16-bit data rather than 32-bit data.

You do not have to explicitly put an operand size prefix byte in front of your 16-bit instructions; the assembler will take care of this automatically for you whenever you use a 16-bit operand in an instruction. However, do keep in mind that whenever you use a 16-bit operand in a 32-bit program, the instruction is longer (by one byte) because of the prefix value. Therefore, you should be careful about using 16-bit instructions if size (and to a lesser extent, speed) are important because these instructions are longer (and may be slower because of their effect on the cache).

---

### 5.5.5 Alternate Encodings for Instructions

As noted earlier in this chapter, one of Intel's primary design goals for the 80x86 was to all programmers to write very short programs in order to save precious (at the time) memory. One way they did this was to create alternate encodings of some very commonly used instructions. These alternate instructions were shorter than the standard counterparts and Intel hoped that programmers would make extensive use of these instructions, thus creating shorter programs.

A good example of these alternate instructions are the "add( constant, accumulator );" instructions (the accumulator is AL, AX, or EAX). The 80x86 provides a single byte opcode for "add( constant, al );" and "add( constant, eax );" (the opcodes are \$04 and \$05, respectively). With a one-byte opcode and no MOD-REG-R/M byte, these instructions are one byte shorter than their standard ADD immediate counterparts. Note that the "add( constant, ax );" instruction requires an operand size prefix (as does the standard "add( constant, ax );" instruction, so it's opcode is effectively two bytes if you count the prefix byte. This, however, is still one byte shorter than the corresponding standard ADD immediate.

You do not have to specify anything special to use these instructions. Any decent assembler will automatically choose the shortest possible instruction it can use when translating your source code into machine code. However, you should note that Intel only provides alternate encodings for the accumulator registers. Therefore, if you have a choice of several instructions to use and the accumulator registers are among these choices, the AL/AX/EAX registers almost always make the best bet. This is a good reason why you should take some time and scan through the encodings of the 80x86 instructions some time. By familiarizing yourself with the instruction encodings, you'll know which instructions have special (and, therefore, shorter) encodings.

---

## 5.6 Putting It All Together

Designing an instruction set that can stand the test of time is a true intellectual challenge. An engineer must balance several compromises when choosing an instruction set and assigning opcodes for the instructions. The Intel 80x86 instruction set is a classic example of a kludge that people are currently using for purposes the original designers never intended. However, the 80x86 is also a marvelous testament to the ingenuity of Intel's engineers who were faced with the difficult task of extending the CPU in ways it was never intended. The end result, though functional, is extremely complex. Clearly, no one designing a CPU (from scratch) today would choose the encoding that Intel's engineers are using. Nevertheless, the 80x86

CPU does demonstrate that careful planning (or just plain luck) does give the designer the ability to extend the CPU far beyond it's original design.

Historically, an important fact we've learned from the 80x86 family is that it's very poor planning to assume that your CPU will last only a short time period and that users will replace the chip and their software when something better comes along. Software developers usually don't have a problem adapting to a new architecture when they write new software (assuming financial incentive to do so), but they are very resistant to moving existing software from one platform to another. This is the primary reason the Intel 80x86 platform remains popular to this day.

Choosing which instructions you want to incorporate into the initial design of a new CPU is a difficult task. You must balance the desire to provide lots of useful instructions with the silicon budget and you must also be careful not to include lots of irrelevant instructions that programmers wind up ignoring for one reason or another. Remember, all future versions of the CPU will probably have to support all the instructions in the initial instruction set, so it's better to err on the side of supplying too few instructions rather than too many. Remember, you can always expand the instruction set in a later version of the chip.

Hand in hand with selecting the optimal instruction set is allowing for easy future expansion of the chip. You must leave some undefined opcodes available so you can easily expand the instruction set later on. However, you must balance the number of undefined opcodes with the number of initial instructions and the size of your opcodes. For efficiency reasons, we want the opcodes to be as short as possible. We also need a reasonable set of instructions in the initial instruction set. A reasonable instruction set may consume most of the legal bit patterns in small opcode. So a hard decision has to be made: reduce the number of instructions in the initial instruction set, increase the size of the opcode, or rely on an opcode prefix byte (which makes the newer instructions (you add later) longer. There is no easy answer to this problem, as the CPU designer, you must carefully weigh these choices during the initial CPU design. Unfortunately, you can't easily change your mind later on.

Most CPUs (Von Neumann architecture) use a binary encoding of instructions and fetch these instructions from memory. This chapter introduces the concept of binary instruction encoding via the hypothetical "Y86" processor. This is a trivial (and not very practical) CPU design that makes it easy to demonstrate how to choose opcodes for a simple instruction set, encode operands, and leave room for future expansion. Some of the more interesting features the Y86 demonstrates includes the fact that an opcode often contains sub-fields and we usually group instructions by the number of types of operands they support. The Y86 encoding also demonstrates how to use special opcodes to differentiate one group of instructions from another and to provide undefined (illegal) opcodes that we can use for future expansion.

The Y86 CPU is purely hypothetical and useful only as an educational tool. After exploring the design of a simple instruction set with the Y86, this chapter began to discuss the encoding of instructions on the 80x86 platform. While the full 80x86 instruction set is far too complex to discuss this early in this text (i.e., there are lots of instructions we still have to discuss later in this text), this chapter was able to discuss basic instruction encoding using the ADD instruction as an example. Note that this chapter only touches on the 80x86 instruction encoding scheme. For a full discussion of 80x86 encoding, see the appendices in this text and the Intel 80x86 documentation.



# Memory Architecture

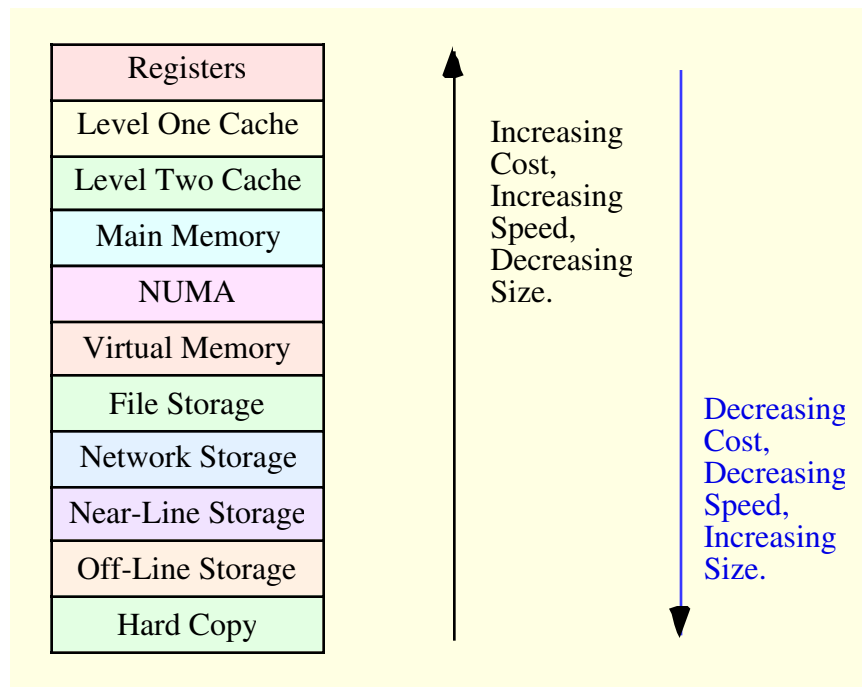
## Chapter Six

### 6.1 Chapter Overview

This chapter discusses the memory hierarchy – the different types and performance levels of memory found on a typical 80x86 computer system. Many programmers tend to view memory as this big nebulous block of storage that holds values for future use. From a semantic point of view, this is a reasonable view. However, from a performance point of view there are many different kinds of memory and using the wrong one or using one form improperly can have a dramatically negative impact on the performance of a program. This chapter discusses the memory hierarchy and how to best use it within your programs.

### 6.2 The Memory Hierarchy

Most modern programs can benefit greatly from a large amount of very fast memory. A physical reality, however, is that as a memory device gets larger, it tends to get slower. For example, cache memories (see “Cache Memory” on page 146) are very fast but are also small and expensive. Main memory is inexpensive and large, but is slow (requiring wait states, see “Wait States” on page 144). The memory hierarchy is a mechanism of comparing the cost and performance of the various places we can store data and instructions. Figure 6.1 provides a look at one possible form of the memory hierarchy.



**Figure 6.1** The Memory Hierarchy

At the top level of the memory hierarchy are the CPU’s general purpose registers. The registers provide the fastest access to data possible on the 80x86 CPU. The register file is also the smallest memory object in the memory hierarchy (with just eight general purpose registers available). By virtue of the fact that it is virtually impossible to add more registers to the 80x86, registers are also the most expensive memory locations. Note that we can include FPU, MMX, SIMD, and other CPU registers in this class as well. These additional

registers do not change the fact that there are a very limited number of registers and the cost per byte is quite high (figuring the cost of the CPU divided by the number of bytes of register available).

Working our way down, the Level One Cache system is the next highest performance subsystem in the memory hierarchy. On the 80x86 CPUs, the Level One Cache is provided on-chip by Intel and cannot be expanded. The size is usually quite small (typically between 4Kbytes and 32Kbytes), though much larger than the registers available on the CPU chip. Although the Level One Cache size is fixed on the CPU and you cannot expand it, the cost per byte of cache memory is much lower than that of the registers because the cache contains far more storage than is available in all the combined registers.

The Level Two Cache is present on some CPUs, on other CPUs it is the system designer's task to incorporate this cache (if it is present at all). For example, most Pentium II, III, and IV CPUs have a level two cache as part of the CPU package, but many of Intel's Celeron chips do not<sup>1</sup>. The Level Two Cache is generally much larger than the level one cache (e.g., 256 or 512KBytes versus 16 Kilobytes). On CPUs where Intel includes the Level Two Cache as part of the CPU package, the cache is not expandable. It is still lower cost than the Level One Cache because we amortize the cost of the CPU across all the bytes in the Level Two Cache. On systems where the Level Two Cache is external, many system designers let the end user select the cache size and upgrade the size. For economic reasons, external caches are actually more expensive than caches that are part of the CPU package, but the cost per bit at the transistor level is still equivalent to the in-package caches.

Below the Level Two Cache system in the memory hierarchy falls the main memory subsystem. This is the general-purpose, relatively low-cost memory found in most computer systems. Typically, this is DRAM or some similar inexpensive memory technology.

Below main memory is the NUMA category. NUMA, which stands for NonUniform Memory Access is a bit of a misnomer here. NUMA means that different types of memory have different access times. Therefore, the term NUMA is fairly descriptive of the entire memory hierarchy. In Figure 6.1a, however, we'll use the term NUMA to describe blocks of memory that are electronically similar to main memory but for one reason or another operate significantly slower than main memory. A good example is the memory on a video display card. Access to memory on video display cards is often an order of magnitude slower than access to main memory. Other peripheral devices that provide a block of shared memory between the CPU and the peripheral probably have similar access times as this video card example. Another example of NUMA includes certain slower memory technologies like Flash Memory that have significant slower access and transfers times than standard semiconductor RAM. We'll use the term NUMA in this chapter to describe these blocks of memory that look like main memory but run at slower speeds.

Most modern computer systems implement a Virtual Memory scheme that lets them simulate main memory using storage on a disk drive. While disks are significantly slower than main memory, the cost per bit is also significantly lower. Therefore, it is far less expensive (by three orders of magnitude) to keep some data on magnetic storage rather than in main memory. A Virtual Memory subsystem is responsible for transparently copying data between the disk and main memory as needed by a program.

File Storage also uses disk media to store program data. However, it is the program's responsibility to store and retrieve file data. In many instances, this is a bit slower than using Virtual Memory, hence the lower position in the memory hierarchy<sup>2</sup>.

Below File Storage in the memory hierarchy comes Network Storage. At this level a program is keeping data on a different system that connects the program's system via a network. With Network Storage you can implement Virtual Memory, File Storage, and a system known as Distributed Shared Memory (where processes running on different computer systems share data in a common block of memory and communicate changes to that block across the network).

Virtual Memory, File Storage, and Network Storage are examples of so-called *on-line memory subsystems*. Memory access via these mechanism is slower than main memory access, but when a program requests data from one of these memory devices, the device is ready and able to respond to the request as quickly as is physically possible. This is not true for the remaining levels in the memory hierarchy.

---

1. Note, by the way, that the level two cache on the Pentium CPUs is typically not on the same chip as the CPU. Instead, Intel packages a separate chip inside the box housing the Pentium CPU and wires this second chip (containing the level two cache) directly to the Pentium CPU inside the package.

2. Note, however, that in some degenerate cases Virtual Memory can be much slower than file access.

The Near-Line and Off-Line Storage subsystems are not immediately ready to respond to a program's request for data. An Off-Line Storage system keeps its data in electronic form (usually magnetic or optical) but on media that is not (necessarily) connected to the computer system while the program that needs the data is running. Examples of Off-Line Storage include magnetic tapes, disk cartridges, optical disks, and floppy diskettes. When a program needs data from an off-line medium, the program must stop and wait for a someone or something to mount the appropriate media on the computer system. This delay can be quite long (perhaps the computer operator decided to take a coffee break?). Near-Line Storage uses the same media as Off-Line Storage, the difference is that the system holds the media in a special robotic jukebox device that can automatically mount the desired media when some program requests it. Tapes and removable media are among the most inexpensive electronic data storage formats available. Hence, these media are great for storing large amounts of data for long time periods.

Hard Copy storage is simply a print-out (in one form or another) of some data. If a program requests some data and that data is present only in hard copy form, someone will have to manually enter the data into the computer. Paper (or other hard copy media) is probably the least expensive form of memory, at least for certain data types.

---

### 6.3 How the Memory Hierarchy Operates

The whole point of the memory hierarchy is to allow reasonably fast access to a large amount of memory. If only a little memory was necessary, we'd use fast static RAM (i.e., the stuff they make cache memory out of) for everything. If speed wasn't necessary, we'd just use low-cost dynamic RAM for everything. The whole idea of the memory hierarchy is that we can take advantage of the principle of locality of reference (see "Cache Memory" on page 146) to move often-referenced data into fast memory and leave less-used data in slower memory. Unfortunately, the selection of often-used versus lesser-used data varies over the execution of any given program. Therefore, we cannot simply place our data at various levels in the memory hierarchy and leave the data alone throughout the execution of the program. Instead, the memory subsystems need to be able to move data between themselves dynamically to adjust for changes in locality of reference during the program's execution.

Moving data between the registers and the rest of the memory hierarchy is strictly a program function. The program, of course, loads data into registers and stores register data into memory using instructions like MOV. It is strictly the programmer's or compiler's responsibility to select an instruction sequence that keeps heavily referenced data in the registers as long as possible.

The program is largely unaware of the memory hierarchy. In fact, the program only explicitly controls access to main memory and those components of the memory hierarchy at the file storage level and below (since manipulating files is a program-specific operation). In particular, cache access and virtual memory operation are generally transparent to the program. That is, access to these levels of the memory hierarchy usually take place without any intervention on the program's part. The program just accesses main memory and the hardware (and operating system) take care of the rest.

Of course, if the program really accessed main memory on each access, the program would run quite slowly since modern DRAM main memory subsystems are much slower than the CPU. The job of the cache memory subsystems (and the cache controller) is to move data between main memory and the cache so that the CPU can quickly access data in the cache. Likewise, if data is not available in main memory, but is available in slower virtual memory, the virtual memory subsystem is responsible for moving the data from hard disk to main memory (and then the caching subsystem may move the data from main memory to cache for even faster access by the CPU).

With few exceptions, most transparent memory subsystem accesses always take place between one level of the memory hierarchy and the level immediately below or above it. For example, the CPU rarely accesses main memory directly. Instead, when the CPU requests data from memory, the Level One Cache subsystem takes over. If the requested data is in the cache, then the Level One Cache subsystem returns the data and that's the end of the memory access. On the other hand if the data is not present in the level one cache, then it passes the request on down to the Level Two Cache subsystem. If the Level Two Cache subsystem has the data, it returns this data to the Level One Cache, which then returns the data to the CPU. Note that requests

for this same data in the near future will come from the Level One Cache rather than the Level Two Cache since the Level One Cache now has a copy of the data.

If neither the Level One nor Level Two Cache subsystems have a copy of the data, then the memory subsystem goes to main memory to get the data. If found in main memory, then the memory subsystems copy this data to the Level Two Cache which passes it to the Level One Cache which gives it to the CPU. Once again, the data is now in the Level One Cache, so any references to this data in the near future will come from the Level One Cache.

If the data is not present in main memory, but is present in Virtual Memory on some storage device, the operating system takes over, reads the data from disk (or other devices, such as a network storage server) and places this data in main memory. Main memory then passes this data through the caches to the CPU.

Because of locality of reference, the largest percentage of memory accesses take place in the Level One Cache system. The next largest percentage of accesses occur in the Level Two Cache subsystems. The most infrequent accesses take place in Virtual Memory.

---

## 6.4 Relative Performance of Memory Subsystems

If you take another look at Figure 6.1 you'll notice that the speed of the various levels increases at the higher levels of the memory hierarchy. A good question to ask, and one we'll hope to answer in this section, is "how much faster is each successive level in the memory hierarchy?" It actually ranges from "almost no difference" to "four orders of magnitude" as you'll seem momentarily.

Registers are, unquestionably, the best place to store data you need to access quickly. Accessing a register never requires any extra time<sup>3</sup>. Further, instructions that access data can almost always access that data in a register. Such instructions already encode the register "address" as part of the MOD-REG-R/M byte (see "Encoding Instruction Operands" on page 279). Therefore, it never takes any extra bits in an instruction to use a register. Instructions that access memory often require extra bytes (i.e., displacement bytes) as part of the instruction encoding. This makes the instruction longer which means fewer of them can sit in the cache or in a prefetch queue. Hence, the program may run slower if it uses memory operands more often than register operands simply due to the instruction size difference.

If you read Intel's instruction timing tables, you'll see that they claim that an instruction like "mov( someVar, ecx );" is supposed to run as fast as an instruction of the form "mov( ebx, ecx );" However, if you read the fine print, you'll find that they make several assumptions about the former instruction. First, they assume that *someVar*'s value is present in the level one cache memory. If it is not, then the cache controller needs to look in the level two cache, in main memory, or worse, on disk in the virtual memory subsystem. All of a sudden, this instruction that should execute in one cycle (e.g., one nanosecond on a one gigahertz processor) requires several milliseconds to execution. That's over six orders of magnitude difference, if you're counting. Now granted, locality of reference suggests that future accesses to this variable will take place in one cycle. However, if you access *someVar*'s value one million times immediately thereafter, the average access time of each instruction will be two cycles because of the large amount of time needed to access *someVar* the very first time (when it was on a disk in the virtual memory system). Now granted, the likelihood that some variable will be on disk in the virtual memory subsystem is quite low. But there is a three orders of magnitude difference in performance between the level one cache subsystem and the main memory subsystem. So if the program has to bring in the data from main memory, 999 accesses later you're still paying an average cost of two cycles for the instruction that Intel's documentation claims should execute in one cycle. Note that register accesses never suffer from this problem. Hence, register accesses are much faster.

The difference between the level one and level two cache systems is not so dramatic. Usually, a level two caching subsystem introduces between one and eight wait states (see "Wait States" on page 144). The difference is usually much greater, though, if the secondary cache is not packaged together with the CPU.

On a one gigahertz processor the level one cache must respond within one nanosecond if the cache operates with zero wait states (note that some processors actually introduce wait states in accesses to the level

---

3. Okay, strictly speaking this is not true. However, we'll ignore data hazards in this discussion and assume that the programmer or compiler has scheduled their instructions properly to avoid pipeline stalls due to data hazards with register data.

one cache, but system designers try not to do this). Accessing data in the level two cache is always slower than in the level one cache and there is always the equivalent of at least one wait state, perhaps more, when accessing data in the level two cache. The reason is quite simple – it takes the CPU time to determine that the data it is seeking is not in the L1 (level one) cache; by the time it determines that the data is not present, the memory access cycle is nearly complete and there is no time to access the data in the L2 (level two) cache.

It may also be that the L2 cache is slower than the L1 cache. This is usually done in order to make the L2 cache less expensive. Also, larger memory subsystems tend to be slower than smaller ones, and L2 caches are usually 16 to 64 times larger than the L1 cache, hence they are usually slower as well. Finally, because L2 caches are not usually on the same silicon chip as the CPU, there are some delays associated with getting data in and out of the cache. All this adds up to additional wait states when accessing data in the L2 cache. As noted above, the L2 cache can be as much as an order of magnitude slower than the L1 cache.

Another difference between the L1 and L2 caches is the amount of data the system fetches when there is an L1 cache miss. When the CPU fetches data from the L1 cache, it generally fetches (or writes) only the data requested. If you execute a "mov( al, memory);" instruction, the CPU writes only a single byte to the cache. Likewise, if you execute "mov( mem32, eax );" then the CPU reads 32 bits from the L1 cache. Access to memory subsystems below the L1 cache, however, do not work in small chunks like this. Usually, memory subsystems read blocks (or *cache lines*) of data whenever accessing lower levels of the memory hierarchy. For example, if you execute the "mov( mem32, eax );" instruction and *mem32*'s value is not in the L1 cache, the cache controller doesn't simply read *mem32*'s value from the L2 cache (assuming it's present there). Instead, the cache controller will actually read a block of bytes (generally 16, 32, or 64 bytes, this depends on the particular processor) from the lower memory levels. The hope is that spatial locality exists and reading a block of bytes will speed up accesses to adjacent objects in memory<sup>4</sup>. The bad news, however, is that the "mov( mem32, eax );" instruction doesn't complete until the L1 cache reads the entire cache line (of 16, 32, 64, etc., bytes) from the L2 cache. Although the program may amortize the cost of reading this block of bytes over future accesses to adjacent memory locations, there is a large passage of time between the request for *mem32* and the actual completion of the "mov( mem32, eax );" instruction. This excess time is known as latency. As noted, the hope is that extra time will be worth the cost when future accesses to adjacent memory locations occur; however, if the program does not access memory objects adjacent to *mem32*, this latency is lost time.

A similar performance gulf separates the L2 cache and main memory. Main memory is typically an order of magnitude slower than the L2 cache. Again the L2 cache reads data from main memory in blocks (cache lines) to speed up access to adjacent memory elements.

There is a three to four order of magnitude difference in performance between standard DRAM and disk storage. To overcome this difference, there is usually a two to three orders of magnitude difference in size between the L2 cache and the main memory. In other words, the idea is "if the access time difference between main memory and virtual memory is two orders of magnitude greater than the difference between the L2 cache and main memory, then we'd better make sure we have two orders of magnitude more main memory than we have L2 cache." This keeps the performance loss to a reasonable level since we access virtual memory on disk two orders of magnitude less often.

We will not consider the performance of the other memory hierarchy subsystems since they are more or less under programmer control (their access is not automatic by the CPU or operating system). Hence, very little can be said about how frequently a program will access them.

---

## 6.5 Cache Architecture

Up to this point, cache has been this magical place that automatically stores data when we need it, perhaps fetching new data as the CPU requires it. However, a good question is "how exactly does the cache do this?" Another might be "what happens when the cache is full and the CPU is requesting additional data not

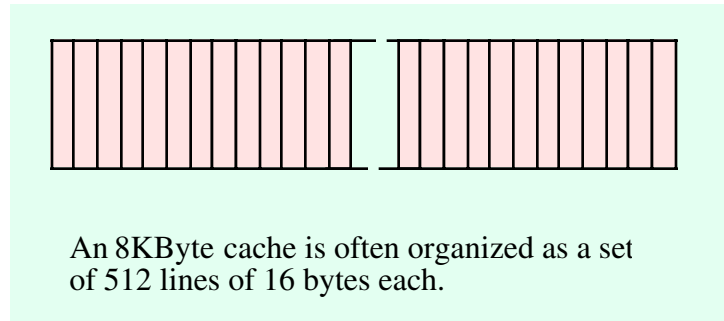
---

4. Note that reading a block of *n* bytes is much faster than *n* reads of one byte. So this scheme is many times faster if spatial locality does occur in the program. For information about spatial locality, see "Cache Memory" on page 146.

in the cache?" In this section, we'll take a look at the internal cache organization and try to answer these questions along with a few others.

The basic idea behind a cache is that a program only access a small amount of data at a given time. If the cache is the same size as the typical amount of data the program access at any one given time, then we can put that data into the cache and access most of the data at a very high speed. Unfortunately, the data rarely sits in contiguous memory locations; usually, there's a few bytes here, a few bytes there, and some bytes somewhere else. In general, the data is spread out all over the address space. Therefore, the cache design has got to accommodate the fact that it must map data objects at widely varying addresses in memory.

As noted in the previous section, cache memory is not organized as a group of bytes. Instead, cache organization is usually in blocks of cache lines with each line containing some number of bytes (typically a small number that is a power of two like 16, 32, or 64), see Figure 6.2.

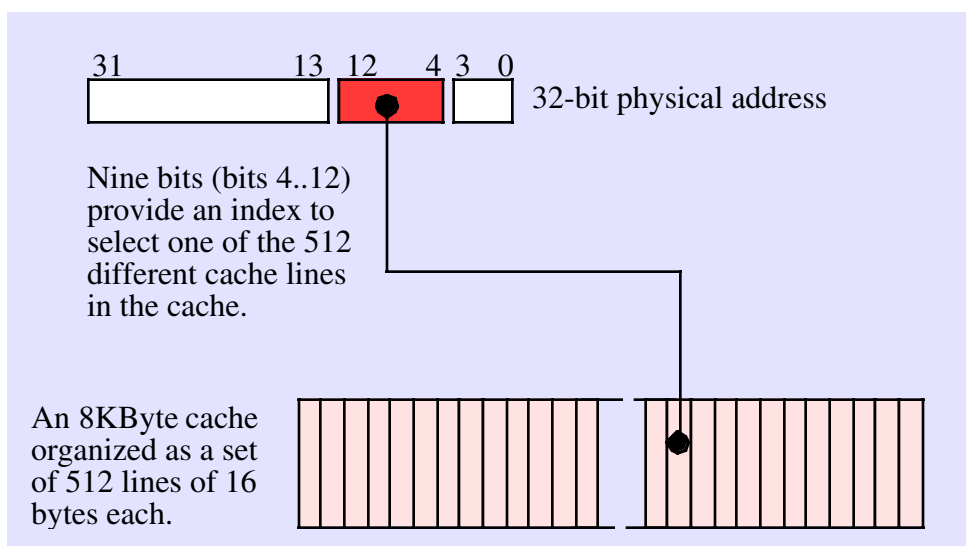


**Figure 6.2** Possible Organization of an 8 Kilobyte Cache

The idea of a cache system is that we can attach a different (non-contiguous) address to each of the cache lines. So cache line #0 might correspond to addresses \$10000..\$1000F and cache line #1 might correspond to addresses \$21400..\$2140F. Generally, if a cache line is  $n$  bytes long ( $n$  is usually some power of two) then that cache line will hold  $n$  bytes from main memory that fall on an  $n$ -byte boundary. In this example, the cache lines are 16 bytes long, so a cache line holds blocks of 16 bytes whose addresses fall on 16-byte boundaries in main memory (i.e., the L.O. four bits of the address of the first byte in the cache line are always zero).

When the cache controller reads a cache line from a lower level in the memory hierarchy, a good question is "where does the data go in the cache?" The most flexible cache system is the *fully associative cache*. In a fully associative cache subsystem, the caching controller can place a block of bytes in any one of the cache lines present in the cache memory. While this is a very flexible system, the flexibility is not without cost. The extra circuitry to achieve full associativity is expensive and, worse, can slow down the memory subsystem. Most L1 and L2 caches are not fully associative for this reason.

At the other extreme is the *direct mapped cache* (also known as the *one-way set associative cache*). In a direct mapped cache, a block of main memory is always loaded into the same cache line in the cache. Generally, some number of bits in the main memory address select the cache line. For example, Figure 6.3 shows how the cache controller could select a cache line for an 8 Kilobyte cache with 16-byte cache lines and a 32-bit main memory address. Since there are 512 cache lines, this example uses bits four through twelve to select one of the cache lines (bits zero through three select a particular byte within the 16-byte cache line). The direct-mapped cache scheme is very easy to implement. Extracting nine (or some other number of) bits from the address and using this as an index into the array of cache lines is trivial and fast. However, direct-mapped caches to suffer from some other problems.

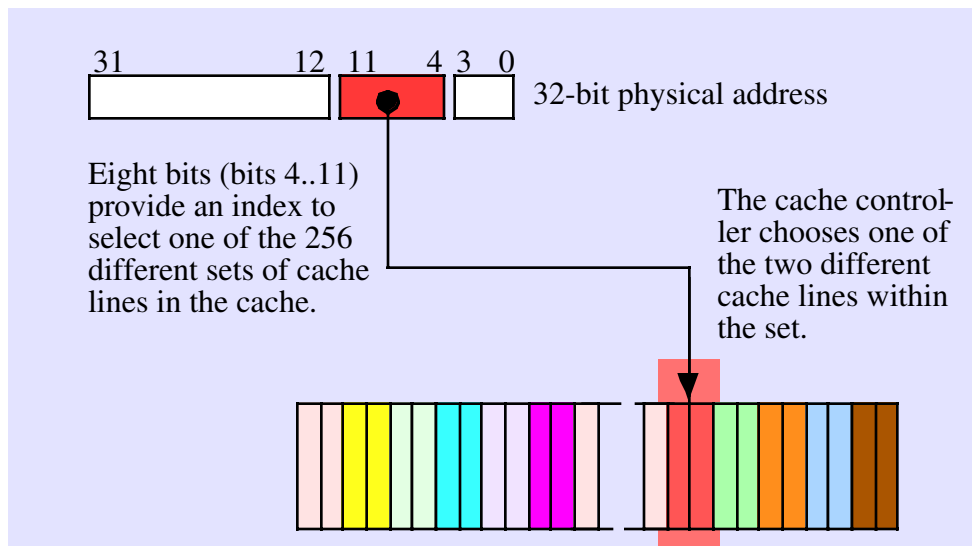


**Figure 6.3** Selecting a Cache Line in a Direct-mapped Cache

Perhaps the biggest problem with a direct-mapped cache is that it may not make effective use of all the cache memory. For example, the cache scheme in Figure 6.3 maps address zero to cache line #0. It also maps address \$2000 (8K), \$4000 (16K), \$6000 (24K), \$8000 (32K), and, in fact, every address that is an even multiple of eight kilobytes to cache line #0. This means that if a program is constantly accessing data at addresses that are even multiples of 8K and not accessing any other locations, the system will only use cache line #0, leaving all the other cache lines unused. Each time the CPU requests data at an address that is not at an address within cache line #0, the CPU will have to go down to a lower level in the memory hierarchy to access the data. In this pathological case, the cache is effectively limited to the size of one cache line. Had we used a fully associative cache organization, each access (up to 512 cache lines' worth) could have their own cache line, thus improving performance.

If a fully associative cache organization is too complex, expensive, and slow to implement, but a direct-mapped cache organization isn't as good as we'd like, one might ask if there is a compromise that gives us more capability than a direct-mapped approach without all the complexity of a fully associative cache. The answer is yes, we can create an *n-way set associative cache* which is a compromise between these two extremes. The idea here is to break up the cache into sets of cache lines. The CPU selects a particular set using some subset of the address bits, just as for direct-mapping. Within each set there are *n* cache lines. The caching controller uses a fully associative mapping algorithm to select one of the *n* cache lines within the set.

As an example, an 8 kilobyte two-way set associative cache subsystem with 16-byte cache lines organizes the cache as a set of 256 sets with each set containing two cache lines ("two-way" means each set contains two cache lines). Eight bits from the memory address select one of these 256 different sets. Then the cache controller can map the block of bytes to either cache line within the set (see Figure 6.4). The advantage of a two-way set associative cache over a direct mapped cache is that you can have two accesses on 8 Kilobyte boundaries (using the current example) and still get different cache lines for both accesses. However, once you attempt to access a third memory location at an address that is an even multiple of eight kilobytes you will have a conflict.



**Figure 6.4 A Two-Way Set Associative Cache**

A two-way set associative cache is much better than a direct-mapped cache and considerably less complex than a fully associative cache. However, if you're still getting too many conflicts, you might consider using a four-way set associative cache. A four-way set associative cache puts four associative cache lines in each block. In the current 8K cache example, a four-way set associative example would have 128 sets with each set containing four cache lines. This would allow up to four accesses to an address that is an even multiple of eight kilobytes before a conflict would occur.

Obviously, we can create an arbitrary  $m$ -way set associative cache (well,  $m$  does have to be a power of two). However, if  $m$  is equal to  $n$ , where  $n$  is the number of cache lines, then you've got a fully associative cache with all the attendant problems (complexity and speed). Most cache designs are direct-mapped, two-way set associative, or four-way set associative. The 80x86 family CPUs use all three (depending on the CPU and cache).

Although this section has made direct-mapped cache look bad, they are, in fact, very effective for many types of data. In particular, they are very good for data that you access in a sequential rather than random fashion. Since the CPU typically executes instructions in a sequential fashion, instructions are a good thing to put into a direct-mapped cache. Data access is probably a bit more random access, so a two-way or four-way set associative cache probably makes a better choice.

Because access to data and instructions is different, many CPU designers will use separate caches for instructions and data. For example, the CPU designer could choose to implement an 8K instruction cache and an 8K data cache rather than a 16K unified cache. The advantage is that the CPU designer could choose a more appropriate caching scheme for instructions versus data. The drawback is that the two caches are now each half the size of a unified cache and you may get fewer cache misses from a unified cache. The choice of an appropriate cache organization is a difficult one and can only be made after analyzing lots of running programs on the target processor. How to choose an appropriate cache format is beyond the scope of this text, just be aware that it's not an easy choice you can make by reading some textbook.

Thus far, we've answered the question "where do we put a block of data when we read it into the cache?" An equally important question we ignored until now is "what happens if a cache line isn't available when we need to read data from memory?" Clearly, if all the lines in a set of cache lines contain data, we're going to have to replace one of these lines with the new data. The question is, "how do we choose the cache line to replace?"

For a direct-mapped (one-way set associative) cache architecture, the answer is trivial. We replace exactly the block that the memory data maps to in the cache. The cache controller replaces whatever data

was formerly in the cache line with the new data. Any reference to the old data will result in a cache miss and the cache controller will have to bring that data into the cache replacing whatever data is in that block at that time.

For a two-way set associative cache, the replacement algorithm is a bit more complex. Whenever the CPU references a memory location, the cache controller uses some number of the address bits to select the set that should contain the cache line. Using some fancy circuitry, the caching controller determines if the data is already present in one of the two cache lines in the set. If not, then the CPU has to bring the data in from memory. Since the main memory data can go into either cache line, somehow the controller has to pick one or the other. If either (or both) cache lines are currently unused, the selection is trivial: pick an unused cache line. If both cache lines are currently in use, then the cache controller must pick one of the cache lines and replace its data with the new data. Ideally, we'd like to keep the cache line that will be referenced first (that is, we want to replace the one whose next reference is later in time). Unfortunately, neither the cache controller nor the CPU is omniscient, they cannot predict which is the best one to replace. However, remember the principle of temporal locality (see "Cache Memory" on page 146): if a memory location has been referenced recently, it is likely to be referenced again in the very near future. A corollary to this is "if a memory location has not been accessed in a while, it is likely to be a long time before the CPU accesses it again." Therefore, a good replacement policy that many caching controllers use is the "least recently used" or LRU algorithm. The idea is to pick the cache line that was not most frequently accessed and replace that cache line with the new data. An LRU policy is fairly easy to implement in a two-way set associative cache system. All you need is a bit that is set to zero whenever the CPU accessing one cache line and set it to one when you access the other cache line. This bit will indicate which cache line to replace when a replacement is necessary. For four-way (and greater) set associative caches, maintaining the LRU information is a bit more difficult, which is one of the reasons the circuitry for such caches is more complex. Other possible replacement policies include First-in, First-out<sup>5</sup> (FIFO) and random. These are easier to implement than LRU, but they have their own little problems.

The replacement policies for four-way and n-way set associative caches are roughly the same as for two-way set associative caches. The major difference is in the complexity of the circuit needed to implement the replacement policy (see the comments on LRU in the previous paragraph).

Another problem we've overlooked in this discussion on caches is "what happens when the CPU writes data to memory?" The simple answer is trivial, the CPU writes the data to the cache. However, what happens when the cache line containing this data is replaced by incoming data? If the contents of the cache line is not written back to main memory, then the data that was written will be lost. The next time the CPU reads that data, it will fetch the original data values from main memory and the value written is lost.

Clearly any data written to the cache must ultimately be written to main memory as well. There are two common write policies that caches use: write-back and write-through. Interestingly enough, it is sometimes possible to set the write policy under software control; these aren't hardwired into the cache controller like most of the rest of the cache design. However, don't get your hopes up. Generally the CPU only allows the BIOS or operating system to set the cache write policy, your applications don't get to mess with this. However, if you're the one writing the operating system...

The write-through policy states that any time data is written to the cache, the cache immediately turns around and writes a copy of that cache line to main memory. Note that the CPU does not have to halt while the cache controller writes the data to memory. So unless the CPU needs to access main memory shortly after the write occurs, this writing takes place in parallel with the execution of the program. Still, writing a cache line to memory takes some time and it is likely that the CPU (or some CPU in a multiprocessor system) will want to access main memory during this time, so the write-through policy may not be a high performance solution to the problem. Worse, suppose the CPU reads and writes the value in a memory location several times in succession. With a write-through policy in place the CPU will saturate the bus with cache line writes and this will have a very negative impact on the program's performance. On the positive side, the write-through policy does update main memory with the new value as rapidly as possible. So if two different CPUs are communicating through the use of shared memory, the write-through policy is probably better because the second CPU will see the change to memory as rapidly as possible when using this policy.

---

5. This policy does exhibit some anomalies. These problems are beyond the scope of this chapter, but a good text on architecture or operating systems will discuss the problems with the FIFO replacement policy.

The second common cache write policy is the write-back policy. In this mode, writes to the cache are not immediately written to main memory; instead, the cache controller updates memory at a later time. This scheme tends to be higher performance because several writes to the same variable (or cache line) only update the cache line, they do not generate multiple writes to main memory.

Of course, at some point the cache controller must write the data in cache to memory. To determine which cache lines must be written back to main memory, the cache controller usually maintains a *dirty bit* with each cache line. The cache system sets this bit whenever it writes data to the cache. At some later time the cache controller checks this dirty bit to determine if it must write the cache line to memory. Of course, whenever the cache controller replaces a cache line with other data from memory, it must first write that cache line to memory if the dirty bit is set. Note that this increases the latency time when replacing a cache line. If the cache controller were able to write dirty cache lines to main memory while no other bus access was occurring, the system could reduce this latency during cache line replacement.

A cache subsystem is not a panacea for slow memory access. In order for a cache system to be effective the software must exhibit locality of reference. If a program accesses memory in a random fashion (or in a fashion guaranteed to exploit the caching controller's weaknesses) then the caching subsystem will actually cause a big performance drop. Fortunately, real-world programs do exhibit locality of reference, so most programs will benefit from the presence of a cache in the memory subsystem.

---

## 6.6 Virtual Memory, Protection, and Paging

In a modern operating system such as Windows, it is very common to have several different programs running concurrently in memory. This presents several problems. First, how do you keep the programs from interfering with one another? Second, if one program expects to load into memory at address \$1000 and a second program also expects to load into memory at address \$1000, how can you load and execute both programs at the same time? One last question we might ask is what happens if our computer has 64 megabytes of memory and we decide to load and execute three different applications, two of which require 32 megabytes and one that requires 16 megabytes (not to mention the memory the operating system requires for its own purposes)? The answer to all these questions lies in the virtual memory subsystem the 80x86 processors support<sup>6</sup>.

Virtual memory on the 80x86 gives each process its own 32-bit address space<sup>7</sup>. This means that address \$1000 in one program is physically different than address \$1000 in a separate program. The 80x86 achieves this sleight of hand by using *paging* to remap *virtual addresses* within one program to different *physical addresses* in memory. A virtual address is the memory address that the program uses. A physical address is the bit pattern that actually appears on the CPU's address bus. The two don't have to be the same (and usually, they aren't). For example, program #1's virtual address \$1000 might actually correspond to physical address \$215000 while program #2's virtual address \$1000 might correspond to physical memory address \$300000. How can the CPU do this? Easy, by using *paging*.

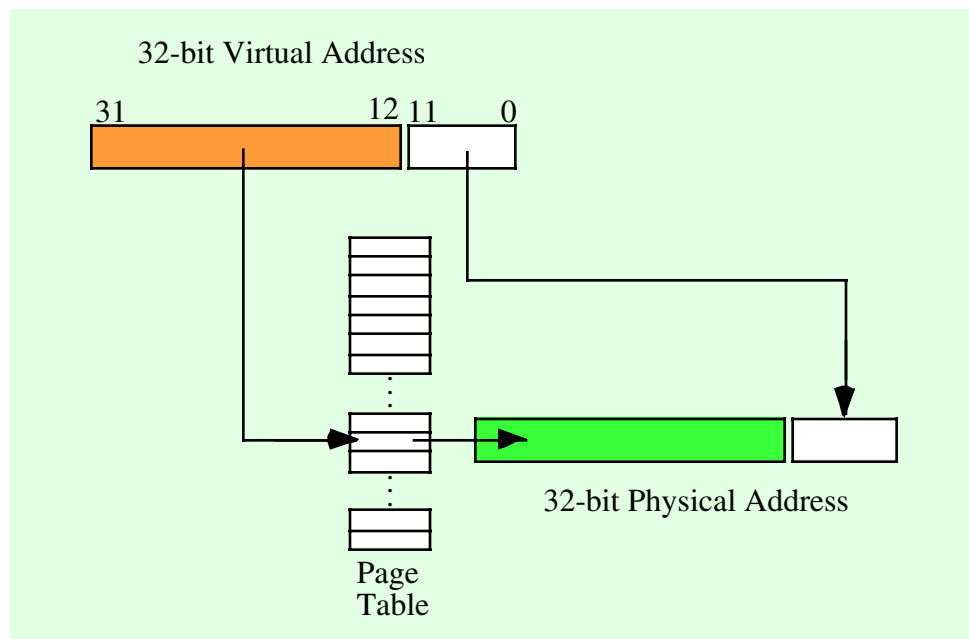
The concept behind paging is quite simple. First, you break up memory into blocks of bytes called pages. A page in main memory is comparable to a cache line in a cache subsystem, although pages are usually much larger than cache lines. For example, the 80x86 CPUs use a page size of 4,096 bytes.

After breaking up memory into pages, you use a lookup table to translate the H.O. bits of a virtual address to select a page; you use the L.O. bits of the virtual address as an index into the page. For example, with a 4,096-byte page, you'd use the L.O. 12 bits of the virtual address as the offset within the page in physical memory. The upper 20 bits of the address you would use as an index into a lookup table that returns the actual upper 20 bits of the physical address (see Figure 6.5).

---

6. Actually, virtual memory is really only supported by the 80386 and later processors. We'll ignore this issue here since most people have an 80386 or later processor.

7. Strictly speaking, you actually get a 36-bit address space on Pentium Pro and later processors, but Windows limits you to 32-bits so we'll use that limitation here.



**Figure 6.5** Translating a Virtual Address to a Physical Address

Of course, a 20-bit index into the page table would require over one million entries in the page table. If each entry is 32 bits (20 bits for the offset plus 12 bits for other purposes), then the page table would be four megabytes long. This would be larger than most of the programs that would run in memory! However, using what is known as a multi-level page table, it is very easy to create a page table that is only 8 kilobytes long for most small programs. The details are unimportant here, just rest assured that you don't need a four megabyte page table unless your program consumes the entire four gigabyte address space.

If you study Figure 6.5 for a few moments, you'll probably discover one problem with using a page table – it requires two memory accesses in order to access an address in memory: one access to fetch a value from the page table and one access to read or write the desired memory location. To prevent cluttering the data (or instruction) cache with page table entries (thus increasing the number of cache misses), the page table uses its own cache known as the *Translation Lookaside Buffer*, or TLB. This cache typically has 32 entries on a Pentium family processor. This provides a sufficient lookup capability to handle 128 kilobytes of memory (32 pages) without a miss. Since a program typically works with less data than this at any given time, most page table accesses come from the cache rather than main memory.

As noted, each entry in the page table is 32 bits even though the system really only needs 20 bits to remap the addresses. Intel uses some of the remaining 12 bits to provide some memory protection information. For example, one bit marks whether a page is read/write or read-only. Another bit determines if you can execute code on that page. Some bits determine if the application can access that page or if only the operating system can do so. Some bits determine if the page is "dirty" (that is, if the CPU has written to the page) and whether the CPU has accessed the page recently (these bits have the same meaning as for cache lines). Another bit determines whether the page is actually present in physical memory or if it's stored on secondary storage somewhere. Note that your applications do not have access to the page table, and therefore they cannot modify these bits. However, Windows does provide some functions you can call if you want to change certain bits in the page table (e.g., Windows will allow you to set a page to read-only if you want to do so).

Beyond remapping memory so multiple programs can coexist in memory even though they access the same virtual addresses, paging also provides a mechanism whereby the operating system can move infrequently used pages to secondary storage (i.e., a disk drive). Just as locality of reference applies to cache lines, it applies to pages in memory as well. At any one given time a program will only access a small per-

centage of the pages in memory that contain data and code (this set of pages is known as the *working set*). While this working set of pages varies (slowly) over time, for a reasonable time period the working set remains constant. Therefore, there is little need to have the remainder of the program in memory consuming valuable physical memory that some other process could be using. If the operating system can save those (currently unused) pages to disk, the physical memory they consume would be available for other programs that need it.

Of course, the problem with moving data out of physical memory is that sooner or later the program might actually need it. If you attempt to access a page of memory and the page table bit tells the MMU (memory management unit) that this page is not present in physical memory, then the CPU interrupts the program and passes control to the operating system. The operating system analyzes the memory access request and reads the corresponding page of data from the disk drive to some available page in memory. The process is nearly identical to that used by a fully associative cache subsystem except, of course, accessing the disk is much slower than main memory. In fact, you can think of main memory as a fully associative write-back cache with 4,096 byte cache lines that caches the data on the disk drive. Placement and replacement policies and other issues are very similar to those we've discussed for caches. Discussing how the virtual memory subsystem works beyond equating it to a cache is well beyond the scope of this text. If you're interested, any decent text on operating system design will explain how a virtual memory subsystem swaps pages between main memory and the disk. Our main goal here is to realize that this process takes place in operating systems like Windows and that accessing the disk is very slow.

One important issue resulting from the fact that each program has a separate page table and the programs themselves don't have access to the page table is that programs cannot interfere with the operation of other programs by overwriting those other program's data (assuming, of course, that the operating system is properly written). Further, if your program crashes by overwriting itself, it cannot crash other programs at the same time. This is a big benefit of a paging memory system.

Note that if two programs want to cooperate and share data, they can do so. All they've got to do is to tell the operating system that they want to share some blocks of memory. The operating system will map their corresponding virtual addresses (of the shared memory area) to the same physical addresses in memory. Under Windows, you can achieve this use *memory mapped files*; see the Windows documentation for more details.

---

## 6.7 Thrashing

Thrashing is a degenerate case that occurs when there is insufficient memory at one level in the memory hierarchy to properly contain the working set required by the upper levels of the memory hierarchy. This can result in the overall performance of the system dropping to the speed of a lower level in the memory hierarchy. Therefore, thrashing can quickly reduce the performance of the system to the speed of main memory or, worse yet, the speed of the disk drive.

There are two primary causes of thrashing: (1) insufficient memory at a given level in the memory hierarchy, and (2) the program does not exhibit locality of reference. If there is insufficient memory to hold a working set of pages or cache lines, then the memory system is constantly replacing one block (cache line or page) with another. As a result, the system winds up operating at the speed of the slower memory in the hierarchy. A common example occurs with virtual memory. A user may have several applications running at the same time and the sum total of these programs' working sets is greater than all of physical memory available to the program. As a result, as the operating system switches between the applications it has to copy each application's data to and from disk and it may also have to copy the code from disk to memory. Since a context switch between programs is often much faster than retrieving data from the disk, this slows the programs down by a tremendous factor since thrashing slows the context switch down to the speed of swapping the applications to and from disk.

If the program does not exhibit locality of reference and the lower memory subsystems are not fully associative, then thrashing can occur even if there is free memory at the current level in the memory hierarchy. For example, suppose an eight kilobyte L1 caching system uses a direct-mapped cache with 16-byte cache lines (i.e., 512 cache lines). If a program references data objects 8K apart on each access then the sys-

tem will have to replace the same line in the cache over and over again with each access. This occurs even though the other 511 cache lines are currently unused.

If insufficient memory is the cause of thrashing, an easy solution is to add more memory (if possible, it is rather hard to add more L1 cache when the cache is on the same chip as the processor). Another alternative is to run fewer processes concurrently or modify the program so that it references less memory over a given time period. If lack of locality of reference is causing the problem, then you should restructure your program and its data structures to make references local to one another.

---

## 6.8 NUMA and Peripheral Devices

Although most of the RAM memory in a system is based on high-speed DRAM interfaced directly to the processor's bus, not all memory is connected to the CPU in this manner. Sometimes a large block of RAM is part of a peripheral device and you communicate with that device by writing data to the RAM on the peripheral. Video display cards are probably the most common example, but some network interface cards and USB controllers also work this way (as well as other peripherals). Unfortunately, the access time to the RAM on these peripheral devices is often much slower than access to normal memory. We'll call such access NUMA<sup>8</sup> access to indicate that access to such memory isn't uniform (that is, not all memory locations have the same access times). In this section we'll use the video card as an example, although NUMA performance applies to other devices and memory technologies as well.

A typical video card interfaces to the CPU via the PCI (or much worse, ISA) bus inside the computer system. The PCI bus nominally runs at 33 MHz and is capable of transferring four bytes per bus cycle. In burst mode, a video controller card, therefore, is capable of transferring 132 megabytes per second (though few would ever come close to achieving this for technical reasons). Now compare this with main memory access. Main memory usually connects directly to the CPU's bus and modern CPUs have a 100 MHz 64-bit wide bus. Technically (if memory were fast enough), the CPU's bus could transfer 800 MBytes/sec. between memory and the CPU. This is six times faster than transferring data across the PCI bus. Game programmers long ago discovered that it's much faster to manipulate a copy of the screen data in main memory and only copy that data to the video display memory when a vertical retrace occurs (about 60 times/sec.). This mechanism is much faster than writing directly to the video memory every time you want to make a change.

Unlike caches and the virtual memory subsystem that operate in a transparent fashion, programs that write to NUMA devices must be aware of this and minimize the accesses whenever possible (e.g., by using an off-screen bitmap to hold temporary results). If you're actually storing and retrieving data on a NUMA device, like a Flash memory card, then you must explicitly cache the data yourself. Later in this text you'll learn about hash tables and searching. Those techniques will help you create your own caching system for NUMA devices.

---

## 6.9 Segmentation

Segmentation is another memory management scheme, like paging, that provides memory protection and virtual memory capabilities. Windows (Win32) does not support the use of segments, nor does HLA provide any instructions that let you manipulate segment registers or use segment override prefixes on an instruction. The Win32 operating systems employ the flat memory model that, essentially, ignores segments on the 80x86. Furthermore, the remainder of this text also ignores segmentation. What this means is that you don't really need to know anything about segmentation in order to write assembly language programs that run under Windows. However, it's unthinkable to write a book on 80x86 assembly language programming that doesn't at least mention segmentation. Hence this section.

The basic idea behind the segmentation model is that memory is managed using a set of segments. Each segment is, essentially, its own address space. A segment consists of two components: a base address that contains the address of some physical memory location and a length value that specifies the length of the

---

8. Remember, NUMA stands for NonUniform Memory Access.

segment. A segmented address also consists of two components: a segment selector and an offset into the segment. The segment selector specifies the segment to use (that is, the base address and length values) while the offset component specifies the offset from the base address for the actual memory access. The physical address of the actual memory location is the sum of the offset and the base address values. If the offset exceeds the length of the segment, the system generates a protection violation.

Segmentation on the 80x86 got a (deservedly) bad name back in the days of the 8086, 8088, and 80286 processors. The problem back then is that the offset into the segment was only a 16-bit value, effectively limiting segments to 64 kilobytes in length. By creating multiple segments in memory it was possible to address more than 64K within a single program; however, it was a major pain to do so, especially if a single data object exceeded 64 kilobytes in length. With the advent of the 80386, Intel solved this problem (and others) with their segmentation model. By then, however, the damage had been done; segmentation had developed a really bad name that it still bears to this day.

Segments are an especially powerful memory management system when a program needs to manipulate different variable sized objects and the program cannot determine the size of the objects before run time. For example, suppose you want to manipulate several different files using Windows' memory mapped file scheme. Under Win32, which doesn't support segmentation, you have to specify the maximum size of the file before you map it into memory. If you don't do this, then Windows can't leave sufficient space at the end of the first file in memory before the second file starts. On the other hand, if Windows supported segmentation, it could easily return segmented pointers to these two memory mapped files, each in their own logical address space. This would allow the files to grow to the size of the maximum offset within a segment (or the maximum file size, whichever is smaller). Likewise, if two programs wanted to share some common data, a segmented system could allow the two programs to put the shared data in a segment. This would allow both programs to reference objects in the shared area using like-valued pointer (offset) values. This makes it easier to pass pointer data (within the shared segment) between the two programs, a very difficult thing to do when using a flat memory model without segmentation as Windows currently does.

One of the more interesting features of the 80386 and later processors is the fact that Intel combined both segmentation and paging in the same memory management unit. Prior to the 80386 most real-world CPUs used paging or segmentation but not both. The 80386 processor merged both of these memory management mechanisms into the same chip, offering the advantages of both systems on a single chip. Unfortunately, most 32-bit operating systems (e.g., Linux and Windows) fail to take advantage of segmentation so this feature goes wasted on the chip.

---

## 6.10 Segments and HLA

Although HLA creates programs use the flat memory model under Win32, HLA does provide limited support for segments in your code. However, HLA's (and Window's) segments are not the same thing as 80x86 segments; HLA segments are a logical organization of memory that has very little to do with segmentation on the 80x86. HLA's segments provide a simple way to organize variables and other objects in memory.

Logically, a segment is a block of memory where you place related objects. By default, HLA supports five different segments: a segment that holds machine instructions, a read-only segment that holds constant objects that HLA creates, a readonly segment that holds values you declare in the `READONLY` section, a data segment that holds variables and other objects you declare in the `STATIC` and `DATA` sections, and a "BSS" section that holds uninitialized variables you declare in the `STORAGE` section<sup>9</sup>.

Normally you are completely unaware of the fact that HLA creates these segments in memory. The use of these segments is automatic and generally transparent to your HLA programs. In a few cases, however, you may need access to this segment information. For example, when linking your HLA programs with high level languages like C/C++ or Delphi you may need to tell HLA to use different names for the five segments it create (as imposed by the high level language). By default, HLA uses the following segment names for its five segments:

---

9. In theory, there is also a stack and a heap segment. However, the linker, not HLA, defines and allocates these two segments. You cannot explicitly declare static objects in these two segments during compilation.

- `_TEXT` for the code segment (corresponds to the `".code"` segment).
- `_DATA` for the STATIC and DATA sections (corresponds to the `".data"` segment).
- `_BSS` for the STORAGE section (corresponds to the `".bss"` segment).
- `"CONST"` for the HLA constant segment (corresponds to the `".edata"` segment).
- `"readonly"` for the HLA READONLY segment (this is not a standardized segment name).

The `"_TEXT"`, `"_DATA"`, `"_BSS"`, and `"CONST"` segment names are quite standard under Win32. Most common compilers that generate Win32 code use these segment names for the code, data, uninitialized, and constant data sections. There does not seem to be a common segment that high level language compilers use for read-only data (other than `CONST`), so HLA creates a separate specifically for this purpose: the `"readonly"` segment where HLA puts the objects you declare in the `READONLY` section.

Examples of objects HLA puts in the `"CONST"` segment include string literal constants for string variables, constants HLA emits for extended syntax forms of the `mul`, `imul`, `div`, `idiv`, `bounds`, and other instructions, floating point constants that HLA automatically emits (e.g., for the `"fld( 1.234 );"` instruction) and so on. Generally, you do not explicitly declare values that wind up in this section (other than through the use of one of the aforementioned instructions).

HLA provides special directives that let you change the default names for these segments. Although `"_TEXT"`, `"_DATA"`, `"_BSS"` and `"CONST"` are very standard names, some compilers may use different names and expect HLA to put its code and data in those different segments. The `"readonly"` segment is definitely non-standard, some compilers may not allow you to use it (indeed, some compilers may not even allow read-only segments in memory). Should you encounter a language that wants different segment names or doesn't allow read-only segments, you can tell HLA to use a different segment name or to map the read-only segments to the static data segment. Here are the directives to achieve this:

```
#code( "codeSegmentName", "alignment", "class" )
#static( "dataSegmentName", "alignment", "class" )
#storage( "bssSegmentName", "alignment", "class" )
#readonly( "readOnlySegmentName", "alignment", "class" )
#const( "constSegmentName", "alignment", "class" )
```

The `#code` directive tells HLA to rename the code segment (`"_TEXT"`) or use different alignment or classification options. The `#static` directive renames the data segment (`"_DATA"`, the segment the `STATIC` and `DATA` sections use). The `#storage` directive renames the uninitialized data segment (`"_BSS"`, the segment the `STORAGE` section uses). The `#readonly` directive renames the `"readonly"` segment (where HLA places data you declare in the `READONLY` section). Finally, the `#const` directive renames HLA's `"CONST"` segments (where HLA places constants that it emits internally).

Each of these directives contains three string expression operands. The first string operand specifies the name of the segment. The second string specifies the segment alignment; we'll return to a discussion of this operand in a moment. The third operand is the segment class; the linker uses this name to combine segments that have different names into a single memory block. Generally, the class name is some variant of the segment name, but this is not necessarily the case (e.g., the standard class name for the `"_TEXT"` segment is `"CODE"`).

The alignment operand must be a string that contains one of the following identifiers: `"byte"`, `"word"`, `"dword"`, `"para"`, or `"page"`. HLA will only allow a string constant containing one of these five strings. The alignment option specifies the boundary on which the linker will start a segment. This option is only meaningful if you combine two different segments by using the same string for the class parameter. The linker combines two segments by concatenating them in memory. When the linker combines the segments, it makes sure that the concatenated segments start on the boundary the alignment operand specifies. A `"byte"` alignment means that the segment can start at an arbitrary byte boundary. The `"word"` and `"dword"` alignment options tell the linker that the segment must start on a word or double word boundary (respectively). The `"para"` alignment option tells the linker to start the segment on a paragraph (16-byte) boundary. The `"page"` option tells the linker to align the segment on a 256-byte page boundary (this has nothing to do with 4K pages). Most systems expect paragraph alignment, so the most common option here is `"para"`<sup>10</sup>.

---

10. In fact, MASM requires `PARA` alignment for the standard segment names. You may only change the alignment if you specify different segment names.

By default, the linker will start each segment in memory on a 4K MMU page boundary. Therefore, if each segment in an HLA program uses only one byte, that program will consume at least 20K because each segment in memory will start on a different 4K boundary <sup>11</sup>. This is why a simple "Hello World" application consumes so much memory – the five default HLA segments each consume 4K of the memory space whether or not the segments actually have 4K of data. The program isn't really 20K long, it's just spread out over the 20K. As you add more code to the "Hello World" program, you'll notice that the executable file doesn't grow in size until you reach some magic point. Then the program jumps in size by increments of 4K (each time a segment's length crosses a 4K boundary, the program grows in length by 4K). If you want the shortest possible executable file, you can tell HLA to combine all the segments into a single segment. However, saving 8K, 12K, or even 16K of data is hardly useful on modern computer systems. Combining segments only saves a significant percentage of the program's size on very tiny programs, so it's not worth the effort for most real applications.

To combine two segments you use the same name for the third parameter in the `#code`, `#data`, `#static`, `#readonly`, and `#const` directives. For example, if you want to combine the "CONST" and "readonly" segments into a single memory segment, you can do so with the following two statements (this is actually the default definition):

```
#readonly( "readonly", "para", "CONST" )
#const( "CONST", "para", "CONST" )
```

By using the same class names but different segment names you tell the linker to combine these two segments in memory. Of course, you can also combine the two segments by giving them the same segment name, e.g.,

```
#readonly( "readonly", "para", "readonly" )
#const( "readonly", "para", "readonly" )    // This is a bad idea, see below.
```

Generally, though, you should not give the `#const` ("CONST") segment the same name as other segments. This can create code generation problems. If you want to combine the "CONST" segment with a different segment, use the class parameter to achieve this. If you give the `#const` segment the same name as one of the other segments, HLA may interleave data objects between the other segment and the `#const` segment; HLA generally assumes this is not the case, so the combination of segments may introduce some subtle bugs in your programs.

If the particular language you are using doesn't support read-only segments, you should map the "readonly" and "CONST" segments to the `__TEXT` (or equivalent) segment using the "CODE" combine class parameter. Do not map the rename these segments to `__TEXT` because this will interleave data with code and that will most likely crash your program.

The segment renaming directives do not check the syntax of the strings you specify for the segment name and class fields. These should be legal MASM identifiers and should not be MASM keywords. Generally, legal HLA identifiers work just fine here (unless, of course, you just happen to pick a MASM reserved word). If you specify a syntactically incorrect segment name or class name, HLA will not complain until it attempts to assemble its output file with MASM.

You may only rename the HLA segments once and these directives must appear before the `UNIT` or `PROGRAM` statements in an HLA source file. HLA does not allow you to change the name of one of these segments after it has emitted any code for a specific segment. Since HLA emits segment declarations in response to a `UNIT` or `PROGRAM` statement, you must do any segment renaming prior to these statements in an HLA source file; i.e., these directives will typically be the very first statements in a source file.

Here are the default segment names, alignments, and class values that HLA uses:

```
#code( "__TEXT", "para", "CODE" )
#static( "__DATA", "para", "DATA" )
#storage( "__BSS", "para", "BSS" )
#const( "CONST", "para", "CONST" )
#readonly( "readonly", "para", "CONST" )
```

---

11. Actually, the smallest HLA program will probably be at least 24K or 28K because the Win32 libraries also consume a few pages in memory.

If you use the MASM-defined names `"_TEXT"`, `"_DATA"`, `"_BSS"`, or `"CONST"` you must provide the alignment and class parameters given above or MASM will complain when it compiles HLA's output.

## 6.11 User Defined Segments in HLA

In addition to the five standard segments, HLA lets you declare your own segments in your assembly programs. Like the five standard segments, you should not confuse HLA segments with 80x86 segments. You do not use the 80x86 segment registers to access data in user-defined segments. Instead, user segments exist as a logical entity to group a set of related objects into the same physical block of memory. In this section we'll take a look at why you would want to use segments and how you declare them in HLA.

It should come as no surprise that when you declare two variables in adjacent statements in a declaration section (e.g., `STATIC`) that HLA allocates those objects in adjacent memory locations. What may be surprising is that HLA will probably not allocate two variables adjacently in memory if you declare those variables in two adjacent declaration selections. E.g., HLA will allocate *i* and *j* below in adjacent memory locations, but it probably will not allocate *j* and *k* in adjacent memory locations:

```
static
    i:uns32;
    j:int32;

storage
    k:dword;
```

The reason *k* does not immediately follow *j* in memory is because *k* is in the `"_BSS"` segment while *i* and *j* are in the `"_DATA"` segment. Since segments typically start on 4K boundaries, there may be a huge gap between *j* and *k*, assuming that the `"_BSS"` segment follows the `"_DATA"` segment in memory (and it may not).

Another somewhat surprising result is that HLA (and MASM and the linker) will combine declarations from declaration sections with the same segment name, even if those declarations are not adjacent. Consider the following code:

```
static
    i:uns32;
    j:int32;

storage
    k:dword;

static
    m:real32;
```

Although *j* and *k* probably won't occupy adjacent memory locations, nor will *k* and *m*, it is quite possible for *j* and *m* to occupy adjacent memory locations since HLA places both declarations in the `"_DATA"` segment. There is no requirement for HLA to allocate *m* immediately after *j*, but HLA will allocate both objects in the same block of physical memory. If you need allocate two variables in adjacent memory locations, or one variable must appear at a lower address than another in memory, you must allocate both objects in the same (physical) declaration sequence. I.e., *i* and *j* (in the declarations above) will be allocated in adjacent memory locations with *i* at the lower address. HLA allocates *m* in the same segment as *i* and *j*, but there's no guarantee that *m* will appear at a higher or lower address than *i* and *j*.

In addition to the five standard segments, HLA lets you define your own memory segments. You use the `SEGMENT` declaration statement to accomplish this. The `SEGMENT` statement takes the following form:

```
segment segName( "alignment", "class" );
    << Declarations >>
```

You would use this declaration anywhere a `STATIC`, `READONLY`, `STORAGE`, or `DATA` declaration section is legal<sup>12</sup>. Anything legal after a `STATIC` keyword is legal after the `SEGMENT` declaration.

The *segName* field in the declaration above is the name you're going to give this segment. You should choose a unique name and it probably shouldn't be *\_TEXT*, *\_BSS*, *\_DATA*, *readonly*, or *CONST* (HLA doesn't prevent the use of these segment names; however, there is little purpose to using most of them since you can create objects in most of these segments using the standard declaration sections). This segment name will automatically be a public name, so you should use an identifier that doesn't conflict with any MASM keywords or other global symbols.

The alignment field must be one of the following strings: *"byte"*, *"word"*, *"dword"*, *"para"*, or *"page"*. This alignment directive has the same meaning as the corresponding string in the segment renaming directives.

The *"class"* operand specifies the combine class. This field has the same meaning as the combine class operand in the segment renaming directives. Note that, like those directives, this operand must be a string constant. HLA does not check the syntax of this string. It should be a legal identifier that doesn't conflict with any MASM reserved words (just like the segment renaming directives' class field).

Segment names are global, even if you define a segment within a procedure. You may use the same segment name in several different segment declaration sections throughout your program; if you do, HLA (and MASM and the linker) will physically combine all the objects you declare in such a segment.

One nice thing about using different segments for variable declarations is that you physically separate the objects in memory. This reduces the impact of errant programs on data unrelated to the task at hand. For example, if you put each procedure's static variables in their own separate segment, this will reduce the likelihood that one procedure will accidentally overwrite another procedure's data if it oversteps an array bounds by a few bytes. Of course, the procedure can still wipe out its own variables by doing this, however, keeping the values in their own segment localizes the impact and makes it easier to track down this defect in your code. One bad thing about using separate segments for each procedure is that each segment consumes a minimum of 4K of memory; so your program's executable will contain a lot of empty data if you have a large number of these segments and you don't declare 4K of data in each procedure.

## 6.12 Controlling the Placement and Attributes of Segments in Memory

Whenever you compile and HLA program, HLA produces two output files: an *".ASM"* file that HLA assembles via MASM, and a *".LINK"* file that contains information for the linker. The *".LINK"* file controls the placement of segments within memory (when the program actually executes) and it also controls other attributes of segments (such as whether they may contain executable code, whether the segment is writable, etc.). When HLA compiles a program to an executable, it first calls a program named *"HLAPARSE.EXE"* which is actually responsible for translating the HLA source code to a MASM-compatible *".ASM"* file. Then HLA calls the *LINK* program to link the *OBJ* files that MASM produces with various library files to produce an executable file<sup>13</sup>. In addition to passing in the list of *OBJ* and *LIB* filenames, HLA also provides the linker with other useful information about segment placement. In this section we'll explore some of the linker options so you can run the linker separately should you want to exercise explicit control over the placement of segments in memory.

To get a (partial) list of the linker options, run the *link.exe* program with the *"/?"* command line option. The linker will respond with a list that looks something like the following:

```
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
usage: LINK [options] [files] [@commandfile]
```

```
options:
```

```
/ALIGN:#
```

12. Well, not really. segment declarations may not appear in classes or namespaces. See the appropriate sections later in this text for a discussion of classes and namespaces.

13. If you've got any resource files, HLA will also call the resource compiler, *rc.exe*, to compile these files. Resource files are beyond the scope of this chapter, so we will ignore them here.

```

/BASE:{address|@filename,key}
/COMMENT:comment
/DEBUG
/DEBUGTYPE:{CV|COFF}
/DEF:filename
/DEFAULTLIB:library
/DELAY:{NOBIND|UNLOAD}
/DELAYLOAD:dll
/DLL
/DRIVER[:{UPONLY|WDM}]
/ENTRY:symbol
/EXETYPE:DYNAMIC
/EXPORT:symbol
/FIXED[:NO]
/FORCE[:{MULTIPLE|UNRESOLVED}]
/GPSIZE:#
/HEAP:reserve[,commit]
/IMPLIB:filename
/INCLUDE:symbol
/INCREMENTAL:{YES|NO}
/LARGEADDRESSAWARE[:NO]
/LIBPATH:dir
/LINK50COMPAT
/MACHINE:{ALPHA|ARM|IX86|MIPS|MIPS16|MIPSR41XX|PPC|SH3|SH4}
/MAP[:filename]
/MAPINFO:{EXPORTS|FIXUPS|LINES}
/MERGE:from=to
/NODEFAULTLIB[:library]
/NOENTRY
/NOLOGO
/OPT:{ICF[,iterations]|NOICF|NOREF|NOWIN98|REF|WIN98}
/ORDER:@filename
/OUT:filename
/PDB:{filename|NONE}
/PDBTYPE:{CON[SOLIDATE]|SEPT[YPE]}
/PROFILE
/RELEASE
/SECTION:name,[E][R][W][S][D][K][L][P][X]
/STACK:reserve[,commit]
/STUB:filename
/SUBSYSTEM:{NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX}[,#[.##]]
/SWAPRUN:{CD|NET}
/VERBOSE[:LIB]
/VERSION:#[.##]
/VXD
/WARN[:warninglevel]
/WINDOWSCE:{CONVERT|EMULATION}
/WS:AGGRESSIVE

```

Most of these options are very advanced, or of little use to us right now. However, a good number of them are useful on occasion so we'll discuss them here.

**/ALIGN: *number*** The number value must be a decimal number and it must be a power of two<sup>14</sup>. The default (which HLA uses) is 4096. This specifies the default alignment for each segment in the program. You should normally leave this at 4K, but if you write a lot of very short assembly programs you can shrink the size of the executable image by setting this to a smaller value. Note that this number should be at least as large as the largest alignment option (byte, word, dword, para, or page) that you specify for you segments.

---

14. You can use a hexadecimal value if you specify the number using C/C++ syntax, e.g., "0x123ABC".

By default, HLA normally uses PAGE alignment for its segments, so unless you rename all the segments (to change the alignment, if nothing else), the `/ALIGN` value should be at least 256.

The `/BASE:address` option lets you specify the starting address of the code segment ("`_TEXT`"). The linker defaults this address to `0x4000000` (i.e., `$400_0000`). HLA typically uses a default value of `0x3000000` (`$300_0000`). This leaves room for a 16 Mbyte unused block, a 16 Mbyte stack segment, and a 16 Mbyte heap segment below the code segment in memory (which is where the linker normally puts the stack and heap). If you want a larger heap or stack segment, you should specify a higher starting address with the `/BASE` linker option.

The `/ENTRY:name` options specifies the name of the main program. This is the location where program execution begins when Windows first executes the program. For HLA console window applications, the name of the main program is `"?HLAMain"`. Unless you're linking HLA code with a main program written in another language, or you completely understand the HLA start up sequence, you should always use this identifier to specify the entry point of an HLA main program. Note that if you circumvent this entry point, HLA does not properly set up the exception handling facilities and other features of the language. So change this name at your own risk.

`/HEAP:reserve,commit` This option specifies the amount of memory that the system reserves for the heap. The first numeric value indicates the amount of heap space to reserve, the second parameter specifies the amount of that heap space to actual map into the address space. By default, HLA supplies `0x1000000` (`$100_0000`, or 16 Mbytes) for both values. This sets aside room for a 16 Mbyte heap and makes all of it available to your program. This is a rather large value for the heap, especially if you write short programs that don't allocate much memory dynamically. For most small applications you may want to set this to a more reasonable (smaller) value. The Windows default is one megabyte (`0x100000` or `$10_0000`). If you don't do much dynamic memory allocation, your code will probably coexists with other applications better if you set this value to 128K (`0x20000` or `$2_0000`). As a general rule, you should set both operands to the same value.

The `/MACHINE:IX86` option tells the linker that you're creating code for an 80x86 CPU. You should not attempt to specify a different CPU when using HLA.

`/MAP` and `/MAP:filename`. These options tell the linker to produce a map *file*. The first form, without the optional filename, causes the linker to produce a map file with the same base name as the output file and a suffix of `".map"`. The second form lets you specify the name of the map file. The map file is a text file that contains several bits of information about the object file. You should produce a map file something and view this information with a text editor to see the kind of information the linker produces. None of this information is usually essential, but it is handy to have now and then. By default, HLA does not produce a map file.

`/MERGE:from=to`. This option merges the segment (section) named *from* to *to*. This will cause the linker to concatenate the two segments in memory. This is roughly equivalent to using the same combine class string in the segment declaration. For example, `"/MERGE:readonly=.edata"` merges the `readonly` segment with the `CONST` segment by concatenating the two.

`/OUT:filename`. This option specifies the output (executable) filename. By default, HLA appends `".EXE"` to the base name of your program and uses that as the executable name. If you would prefer a different name, then use this option to specify the executable file name that LINK produces.

`/SECTION:name,options`. This option lets you specify the ordering of segments in memory as well as apply attributes to those segments. The `".LINK"` file that HLA produces contains a list of `/SECTION` commands to feed to the linker that specifies the ordering of the segments (by their appearance in the `".LINK"` file) and the attributes of those segments. The *name* field is the segment name. This is a case sensitive field, so the case of *name* must exactly match the original segment declaration. The *options* field is a string of one or more characters that specifies the characteristics of that segment in memory. Here are some of the more common options:

- **E** Allows the execution of code in this segment
- **R** Allows the program to read data in this segment
- **W** Allows the program to write data in this segment
- **S** Shared. Allows multiple copies of this program to share this data.
- **K** Marks the page as non-cachable (generally for multiprocessing applications).
- **P** Marks the page as non-pageable (i.e., it must always be in real memory).

Most of the other options are either very advanced, uninteresting, or not applicable to HLA programs. Most segments will have at least one of the E, R, or W options. HLA's default segments generally use the following section options:

```
/SECTION: .text,ER      -- Note: .text = _TEXT
/SECTION: .edata,R      -- Note: .edata = CONST
/SECTION: readonly,R
/SECTION: .data,RW      -- Note: .data = _DATA
/SECTION: .bss,RW       -- Note: .bss = _BSS
```

*/STACK:reserve,commit.* This option is similar to the */HEAP* option except it reserves space for the program's stack segment rather than the heap segment. Like the HEAP segment, HLA defaults the stack size to 16 Mbytes (0x4000000 or \$400\_0000). If you write shorter applications that don't use a lot of local variable space or heavy recursion, you may want to consider setting this value to one megabyte or less, e.g., */STACK:0x100000,0x100000*.

*/SUBSYSTEM:system.* You must supply a subsystem option when you create an executable program. For HLA programs you would normally use */SUBSYSTEM:CONSOLE* when writing a standard console application. You can use HLA to write GUI applications, if you do this, then you will need to use the */SUBSYSTEM:WINDOWS* linker option. By default, HLA links your code with the */SUBSYSTEM:CONSOLE* option. If you use the HLA *-w* command line option, then HLA will invoke the linker with the */SUBSYSTEM:WINDOWS* option. Of course, if you explicitly run the linker yourself, you will have to supply one of these two options.

The preceding paragraphs explain most of the command line options you'll use when linking programs written in HLA. For more information about the linker, see the Microsoft on-line documentation that accompanies the linker.

If you get tired of typing really long linker command lines every time you compile and link an HLA program, you can gather all the (non-changing) command line options into a linker command file and tell the linker to grab those options and filenames from the command file rather than from the command line. The ".LINK" file that the HLA compiler produces is a good example of a linker command file. The ".LINK" file contains the */SECTION* options for the default and user-defined segments found in an HLA program. Rather than manually supplying these options on each call to the linker, you can use a command line like the following:

```
link @filename.link other_options file_names
```

The at-sign ("@") tells the linker to read a list of commands from the specified command file. Note that you can have several different command files, so if you're compiling and linking several different HLA source files, you can specify the ".link" file for each compilation on the command line.

The filenames you specify on the linker command line should be the names of OBJ and LIB files that you wish to link together. In addition to the OBJ files you've created with HLA, you'll probably want to specify the following library files:

- kernel32.lib      Contains definitions for the base Windows API (e.g., console stuff)
- user32.lib        Contains the definition of the MessageBox dialog (used for exceptions).
- hlalib.lib        The HLA Standard Library

If you don't call any HLA Standard Library routines (unlikely, but possible) then you obviously don't need to specify the hlalib.lib file. Note that it doesn't hurt to specify the name of a library whose members you don't use. The linker will not include any object code from a library unless the program actually uses code or data from that library.

If you're manually linking code that you compile with HLA, you will probably want to create one linker command file containing all the static commands and include that and any appropriate HLA ".LINK" files on the linker command line. Here's a typical example of a static link file (i.e., a file that doesn't get rewritten each time you compile the HLA program):

```
/heap:0x20000, 0x20000
/stack:0x2000, 0x20000
/base:0x1000000
```

```
/machine:IX86
/entry:?HLAMain
/out:mypgm.exe
kernel32.lib
user32.lib
hlalib.lib
```

Generally, you'd use the /SECTION commands from the HLA ".LINK" file unless you wanted to explicitly set the segment ordering or change the attributes of the memory segments.

To run the linker manually, you'd normal tell HLA to perform a compile (and assemble) only operation. This is done using the HLA "-c" command line option. That is, a command like "hla -c myfile.hla" will compile "myfile.hla" to "myfile.asm" and then run MASM to assemble this to "myfile.obj". HLA will not run the linker when you specify the "-c" option. If you prefer, you can run MASM separately by using the "-s" command line option as follows:

```
hla -s myfile.hla
ml -c -Cp -COFF myfile.asm
```

However, there is very little benefit to running the assembler yourself (run "MASM /?" to see the available MASM command line options).

Once you've compiled all necessary source files, you can link them by using the Microsoft LINK.EXE program with the command line (or command file) options this section discusses. Note that this section discusses options specific to the LINK.EXE v6.0 product. These features may change in a future version of the linker. Please see the Microsoft documentation if you have any questions about how the linker operates or if you're using a different version of the linker.

---

## 6.13 Putting it All Together

CPU architects divide memory into several different types depending on cost, capacity, and speed. They call this the memory hierarchy. Many of the levels in the memory hierarchy are transparent to the programmer. That is, the system automatically moves data between levels in the memory hierarchy without intervention on the programmer's part. However, if you are aware of the effects of the memory hierarchy on program performance, you can write faster programs by organizing your data and code so that it conforms to the expectations of the caching and virtual memory subsystems in the memory hierarchy.

# The I/O Subsystem

## Chapter Seven

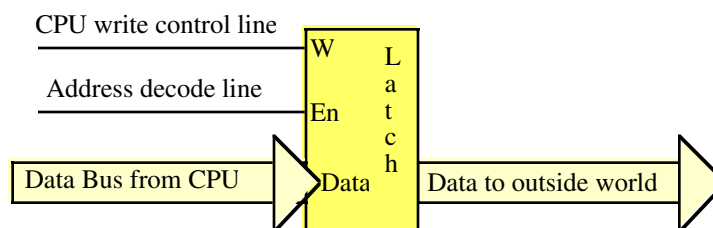
### 7.1 Chapter Overview

A typical program does three basic activities: input, computation, and output. In this section we will discuss the other two activities beyond computation: input and output or I/O. This chapter concentrates on low-level CPU I/O rather than high level file or character I/O. This chapter discusses how the CPU transfers bytes of data to and from the outside world. This chapter discusses the mechanisms and performance issues behind the I/O.

### 7.2 Connecting a CPU to the Outside World

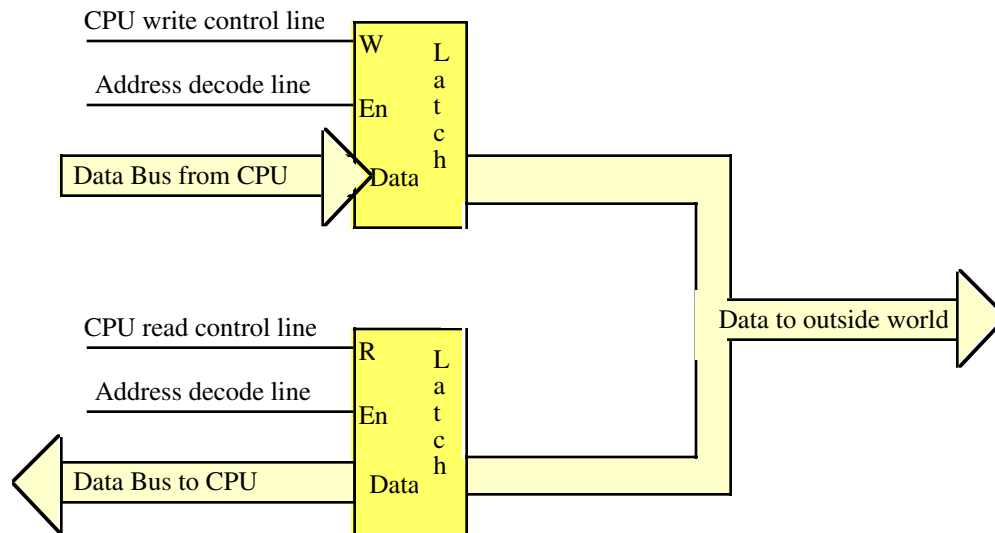
Most I/O devices interface to the CPU in a fashion quite similar to memory. Indeed, many devices appear to the CPU as though they were memory devices. To output data to the outside world the CPU simply stores data into a "memory" location and the data magically appears on some connectors external to the computer. Similarly, to input data from some external device, the CPU simply transfers data from a "memory" location into the CPU; this "memory" location holds the value found on the pins of some external connector.

An output port is a device that looks like a memory cell to the computer but contains connections to the outside world. An I/O port typically uses a latch rather than a flip-flop to implement the memory cell. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system (see Figure 7.1). Note that output ports can be write-only, or read/write. The port in Figure 7.1, for example, is a write-only port. Since the outputs on the latch do not loop back to the CPU's data bus, the CPU cannot read the data the latch contains. Both the address decode and write control lines must be active for the latch to operate; when reading from the latch's address the decode line is active, but the write control line is not.



**Figure 7.1 A Typical Output Port**

Figure 7.2 shows how to create a read/write input/output port. The data written to the output port loops back to a transparent latch. Whenever the CPU reads the decoded address the read and decode lines are active and this activates the lower latch. This places the data previously written to the output port on the CPU's data bus, allowing the CPU to read that data. A read-only (input) port is simply the lower half of Figure 7.2; the system ignores any data written to an input port.



**Figure 7.2 An Output Port that Supports Read/Write Access**

Note that the port in Figure 7.2 is not an input port. Although the CPU can read this data, this port organization simply lets the CPU read the data it previously wrote to the port. The data appearing on an external connector is an output port (only). One could create a (read-only) input port by using the lower half of the circuit in Figure 7.2. The input to the latch would appear on the CPU's data bus whenever the CPU reads the latch data.

A perfect example of an output port is a parallel printer port. The CPU typically writes an ASCII character to a byte-wide output port that connects to the DB-25F connector on the back of the computer's case. A cable transmits this data to the printer where an input port (to the printer) receives the data. A processor inside the printer typically converts this ASCII character to a sequence of dots it prints on the paper.

Generally, a given peripheral device will use more than a single I/O port. A typical PC parallel printer interface, for example, uses three ports: a read/write port, an input port, and an output port. The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port). The input port returns control signals from the printer; these signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc. The output port transmits control information to the printer such as whether data is available to print.

The first thing to learn about the input/output subsystem is that I/O in a typical computer system is radically different than I/O in a typical high level programming language. In a real computer system you will rarely find machine instructions that behave like *writeln*, *cout*, *printf*, or even the HLA *stdin* and *stdout* statements. In fact, most input/output instructions behave exactly like the 80x86's MOV instruction. To send data to an output device, the CPU simply moves that data to a special memory location. To read data from an input device, the CPU simply moves data from the address of that device into the CPU. Other than there are usually more wait states associated with a typical peripheral device than actual memory, the input or output operation looks very similar to a memory read or write operation.

### 7.3 Read-Only, Write-Only, Read/Write, and Dual I/O Ports

We can classify input/output ports into four categories based on the CPU's ability to read and write data at a given port address. These four categories are read-only ports, write-only ports, read/write ports, and dual I/O ports.

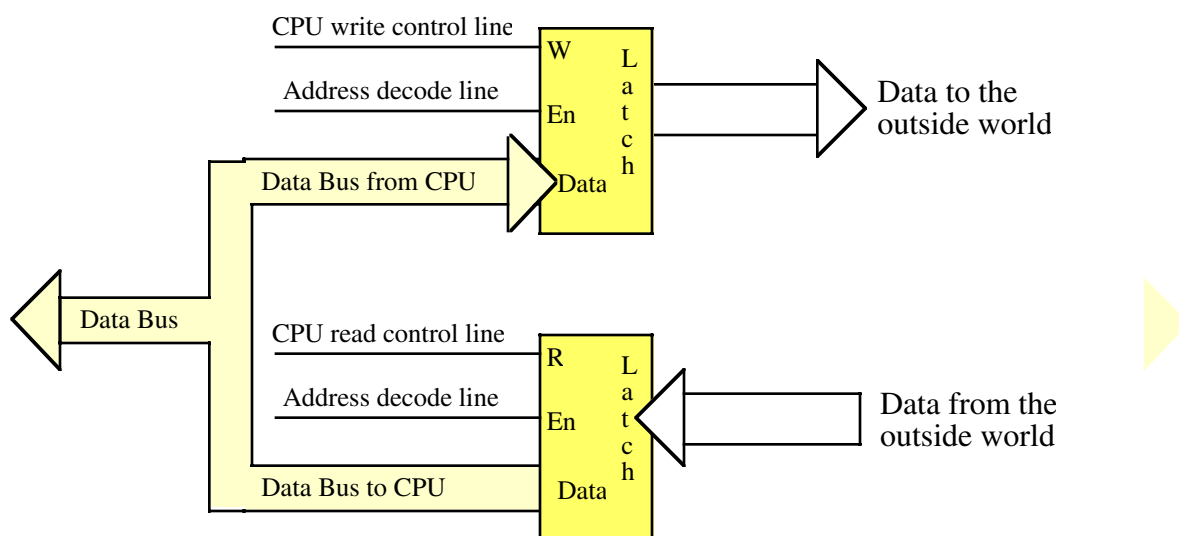
A read-only port is (obviously) an input port. If the CPU can only read the data from the port, then that port is providing data appearing on lines external to the CPU. The system typically ignores any attempt to

write data to a read-only port<sup>1</sup>. A good example of a read-only port is the status port on a PC's parallel printer interface. Reading data from this port lets you test the current condition of the printer. The system ignores any data written to this port.

A write-only port is always an output port. Writing data to such a port presents the data for use by an external device. Attempting to read data from a write-only port generally returns garbage (i.e., whatever values that just happen to be on the data bus at that time). You generally cannot depend on the meaning of any value read from a write-only port.

A read/write port is an output port as far as the outside world is concerned. However, the CPU can read as well as write data to such a port. Whenever the CPU reads data from a read/write port, it reads the data that was last written to the port. Reading the port does not affect the data the external peripheral device sees, reading the port is a simple convenience for the programmer so that s/he doesn't have to save the value last written to the port should they want to retrieve the value.

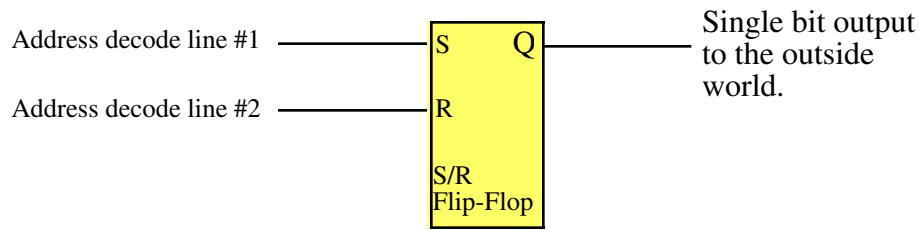
A dual I/O port is also a read/write port, but reading the port reads data from some external device while writing data to the port transmits data to a different external device. Figure 7.3 shows how you could interface such a device to the system. Note that the input and output ports are actually a read-only and a write-only port that share the same address. Reading the address accesses one port while writing to the address accesses the other port. Essentially, this port arrangement uses the R/W control line(s) as an extra address bit when selecting these ports.



**Figure 7.3 An Input and an Output Device That Share the Same Address (a Dual I/O Port)**

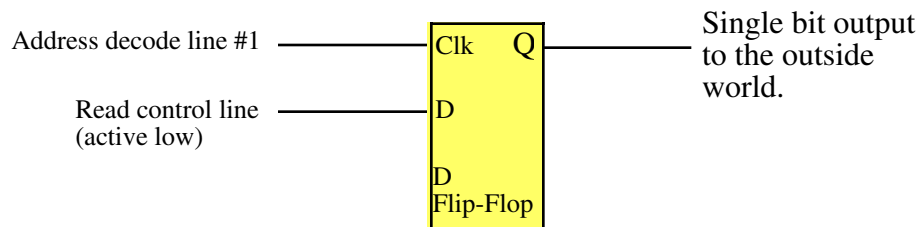
These examples may leave you with the impression that the CPU always reads and writes data to peripheral devices using data on the data bus (that is, whatever data the CPU places on the data bus when it writes to an output port is the data actually written to that output port). While this is generally true for input ports (that is, the CPU transfers input data across the data bus when reading data from the input port), this isn't necessarily true for output ports. In fact, a very common output mechanism is simply accessing a port. Figure 7.4 provides a very simple example. In this circuit, an address decoder decodes two separate addresses. Any access (read or write) to the first address sets the output line high; any read or write of the second address clears the output line. Note that this circuit ignores the data on the CPU's data lines. It is not important whether the CPU reads or writes data to these addresses, nor is the data written of any consequence. The only thing that matters is that the CPU access one of these two addresses.

1. Note, however, that some devices may fail if you attempt to write to their corresponding input ports, so it's never a good idea to write data to a read-only port.



**Figure 7.4**      **Outputting Data to a Port by Simply Accessing That Port**

Another possible way to connect an output port to the CPU is to use a D flip-flop and connect the read/write status lines to the D input on the flip-flop. Figure 7.5 shows how you could design such a device. In this diagram any read of the selected port sets the output bit to zero while a write to this output port sets the output bit to one.



**Figure 7.5**      **Outputting Data Using the Read/Write Control as the Data to Output**

There are a wide variety of ways you can connect external devices to the CPU. This section only provides a few examples as a sampling of what is possible. In the real world, there are an amazing number of different ways that engineers connect external devices to the CPU. Unless otherwise noted, the rest of this chapter will assume that the CPU reads and writes data to an external device using the data bus. This is not to imply that this is the only type of I/O that one could use in a given example.

## 7.4 I/O (Input/Output) Mechanisms

There are three basic forms of input and output that a typical computer system will use: I/O-mapped I/O, memory-mapped I/O, and direct memory access (DMA). I/O-mapped input/output uses special instructions to transfer data between the computer system and the outside world; memory-mapped I/O uses special memory locations in the normal address space of the CPU to communicate with real-world devices; DMA is a special form of memory-mapped I/O where the peripheral device reads and writes data in memory without going through the CPU. Each I/O mechanism has its own set of advantages and disadvantages, we will discuss these in this section.

### 7.4.1 Memory Mapped Input/Output

A memory mapped peripheral device is connected to the CPU's address and data lines exactly like memory, so whenever the CPU reads or writes the address associated with the peripheral device, the CPU transfers data to or from the device. This mechanism has several benefits and only a few disadvantages.

The principle advantage of a memory-mapped I/O subsystem is that the CPU can use any instruction that accesses memory to transfer data between the CPU and a memory-mapped I/O device. The MOV instruction is the one most commonly used to send and receive data from a memory-mapped I/O device, but any instruction that reads or writes data in memory is also legal. For example, if you have an I/O port that is read/write, you can use the ADD instruction to read the port, add data to the value read, and then write data back to the port.

Of course, this feature is only usable if the port is a read/write port (or the port is readable and you've specified the port address as the source operand of your ADD instruction). If the port is read-only or write-only, an instruction that reads memory, modifies the value, and then writes the modified value back to memory will be of little use. You should use such read/modify/write instructions only with read/write ports (or dual I/O ports if such an operation makes sense).

Nevertheless, the fact that you can use any instruction that accesses memory to manipulate port data is often a big advantage since you can operate on the data with a single instruction rather than first moving the data into the CPU, manipulating the data, and then writing the data back to the I/O port.

The big disadvantage of memory-mapped I/O devices is that they consume addresses in the memory map. Generally, the minimum amount of space you can allocate to a peripheral (or block of related peripherals) is a four kilobyte page. Therefore, a few independent peripheral peripherals can wind up consuming a fair amount of the physical address space. Fortunately, a typical PC has only a couple dozen such devices, so this isn't much of a problem. However, some devices, like video cards, consume a large chunk of the address space (e.g., some video cards have 32 megabytes of on-board memory that they map into the memory address space).

---

## 7.4.2 I/O Mapped Input/Output

I/O-mapped input/output uses special instructions to access I/O ports. Many CPUs do not provide this type of I/O, though the 80x86 does. The Intel 80x86 family uses the IN and OUT instructions to provide I/O-mapped input/output capabilities. The 80x86 IN and OUT instructions behave somewhat like the MOV instruction except they transmit their data to and from a special I/O address space that is distinct from the memory address space. The IN and OUT instructions use the following syntax:

```
in( port, al ); // ... or AX or EAX, port is a constant in the range
out( al, port ); // 0..255.

in( dx, al ); // Or AX or EAX.
out( al, dx );
```

The 80x86 family uses a separate address bus for I/O transfers<sup>2</sup>. This bus is only 16-bits wide, so the 80x86 can access a maximum of 65,536 different bytes in the I/O space. The first two instructions encode the port address as an eight-bit constant, so they're actually limited to accessing only the first 256 I/O addresses in this address space. This makes the instruction shorter (two bytes instead of three). Unfortunately, most of the interesting peripheral devices are at addresses above 255, so the first pair of instructions above are only useful for accessing certain on-board peripherals in a PC system.

To access I/O ports at addresses beyond 255 you must use the latter two forms of the IN and OUT instructions above. These forms require that you load the 16-bit I/O address into the DX register and use DX as a pointer to the specified I/O address. For example, to write a byte to the I/O address \$378<sup>3</sup> you would use an instruction sequence like the following:

```
mov( $378, dx );
out( al, dx );
```

---

2. Physically, the I/O address bus is the same as the memory address bus, but additional control lines determine whether the address on the bus is accessing memory or an I/O device.

3. This is typically the address of the data port on the parallel printer port.

The advantage of an I/O address space is that peripheral devices mapped to this area do not consume space in the memory address space. This allows you to fully expand the memory address space with RAM or other memory. On the other hand, you cannot use arbitrary memory instructions to access peripherals in the I/O address space, you can only use the IN and OUT instructions.

Another disadvantage to the 80x86 I/O address space is that it is quite small. Although most peripheral devices only use a couple of I/O address (and most use fewer than 16 I/O addresses), a few devices, like video display cards, can occupy millions of different I/O locations (e.g., three bytes for each pixel on the screen). As noted earlier, some video display cards have 32 megabytes of dual-ported RAM on board. Clearly we cannot easily map this many locations into the 64K I/O address space.

---

### 7.4.3 Direct Memory Access

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory. For this reason, we often call these two forms of input/output programmed I/O. For example, to input a sequence of ten bytes from an input port and store these bytes into memory the CPU must read each value and store it into memory. For very high-speed I/O devices the CPU may be too slow when processing this data a byte (or word or double word) at a time. Such devices generally have an interface to the CPU's bus so they can directly read and write memory. This is known as direct memory access since the peripheral device accesses memory directly, without using the CPU as an intermediary. This often allows the I/O operation to proceed in parallel with other CPU operations, thereby increasing the overall speed of the system. Note, however, that the CPU and DMA device cannot both use the address and data busses at the same time. Therefore, concurrent processing only occurs if the CPU has a cache and is executing code and accessing data found in the cache (so the bus is free). Nevertheless, even if the CPU must halt and wait for the DMA operation to complete, the I/O is still much faster since many of the bus operations during I/O or memory-mapped input/output consist of instruction fetches or I/O port accesses which are not present during DMA operations.

A typical DMA controller consists of a pair of counters and other circuitry that interfaces with memory and the peripheral device. One of the counters serves as an address register. This counter supplies an address on the address bus for each transfer. The second counter specifies the number of transfers to complete. Each time the peripheral device wants to transfer data to or from memory, it sends a signal to the DMA controller. The DMA controller places the value of the address counter on the address bus. At the same time, the peripheral device places data on the data bus (if this is an input operation) or reads data from the data bus (if this is an output operation). After a successful data transfer, the DMA controller increments its address register and decrements the transfer counter. This process repeats until the transfer counter decrements to zero.

---

## 7.5 I/O Speed Hierarchy

Different devices have different data transfer rates. Some devices, like keyboards, are extremely slow (comparing their speed to CPU speeds). Other devices like disk drives can actually transfer data faster than the CPU can read it. The mechanisms for data transfer differ greatly based on the transfer speed of the device. Therefore, it makes sense to create some terminology to describe the different transfer rates of peripheral devices.

*Low-speed devices* are those that produce or consume data at a rate much slower than the CPU is capable of processing. For the purposes of discussion, we'll claim that low-speed devices operate at speeds that are two to three orders of magnitude (or more) slower than the CPU. *Medium-speed devices* are those that transfer data at approximately the same rate (within an order of magnitude slower, but never faster) than the CPU. *High-speed devices* are those that transfer data faster than the CPU is capable of moving data between the device and the CPU. Clearly, high-speed devices must use DMA since the CPU is incapable of transferring the data between the CPU and memory.

With typical bus architectures, modern day PCs are capable of one transfer per microsecond or better. Therefore, high-speed devices are those that transfer data more rapidly than once per microsecond.

Medium-speed transfers are those that involve a data transfer every one to 100 microseconds. Low-speed devices usually transfer data less often than once every 100 microseconds. The difference between these speeds will decide the mechanism we use for the I/O operation (e.g., high-speed transfers require the use of DMA or other techniques).

Note that one transfer per microsecond is not the same thing as a one megabyte per second data transfer rate. A peripheral device can actually transfer more than one byte per data transfer operation. For example, when using the "in( dx, eax );" instruction, the peripheral device can transfer four bytes in one transfer. Therefore, if the device is reaching one transfer per microsecond, then the device can transfer four megabytes per second. Likewise, a DMA device on a Pentium processor can transfer 64 bits at a time, so if the device completes one transfer per microsecond it will achieve an eight megabyte per second data transfer rate.

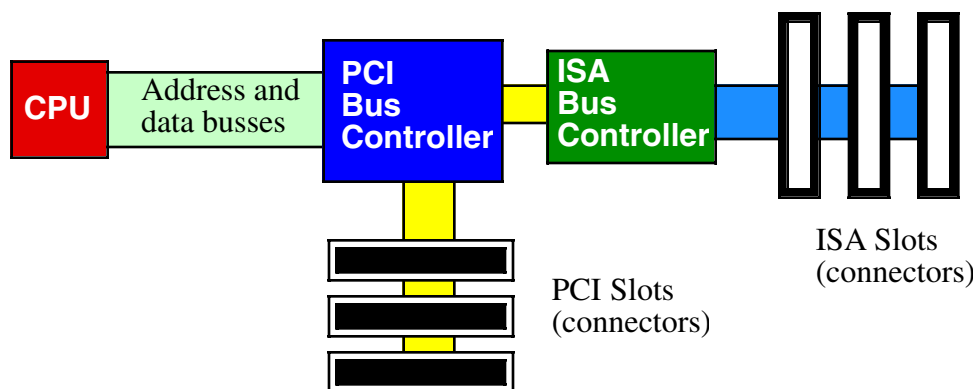
## 7.6 System Busses and Data Transfer Rates

In Chapter One of this Volume (see "The System Bus" on page 130) you saw that the CPU communicates to memory and I/O devices using the system bus. In that chapter you saw that a typical Von Neumann Architecture machine has three different busses: the address bus, the data bus, and the control bus. If you've ever opened up a computer and looked inside or read the specifications for a system, you've probably heard terms like *PCI*, *ISA*, *EISA*, or even *NuBus* mentioned when discussing the computer's bus. If you're familiar with these terms, you may wonder what their relationship is with the CPU's bus. In this section we'll discuss this relationship and describe how these different busses affect the performance of a system.

Computer system busses like PCI (Peripheral Connection Interface) and ISA (Industry Standard Architecture) are definitions for physical connectors inside a computer system. These definitions describe a set of signals, physical dimensions (i.e., connector layouts and distances from one another), and a data transfer protocol for connecting different electronic devices. These busses are related to the CPU's bus only insofar as many of the signals on one of the peripheral busses also appear on the CPU's bus. For example, all of the aforementioned busses provide lines for address, data, and control functions.

Peripheral interconnection busses do not necessarily mirror the CPU's bus. All of these busses contain several additional lines that are not present on the CPU's bus. These additional lines let peripheral devices communicate with one other directly (without having to go through the CPU or memory). For example, most busses provide a common set of interrupt control signals that let various I/O devices communicate directly with the system's interrupt controller (which is also a peripheral device). Nor does the peripheral bus always include all the signals found on the CPU's bus. For example, the ISA bus only supports 24 address lines whereas the Pentium II supports 36 address lines. Therefore, peripherals on the ISA bus only have access to 16 megabytes of the Pentium II's 64 gigabyte address range.

A typical modern-day PC supports the PCI bus (although some older systems also provide ISA connectors). The organization of the PCI and ISA busses in a typical computer system appears in Figure 7.6.



**Figure 7.6** Connection of the PCI and ISA Busses in a Typical PC

Notice how the CPU's address and data busses connect to a PCI Bus Controller device (which is, itself, a peripheral of sorts). The actual PCI bus is connected to this chip. Note that the CPU does not connect directly to the PCI bus. Instead, the PCI Bus Controller acts as an intermediary, rerouting all data transfer requests between the CPU and the PCI bus.

Another interesting thing to note is that the ISA Bus Controller is not directly connected to the CPU. Instead, it is connected to the PCI Bus Controller. There is no logical reason why the ISA Controller couldn't be connected directly to the CPU's bus, however, in most modern PCs the ISA and PCI controllers appear on the same chip and the manufacturer of this chip has chosen to interface the ISA bus through the PCI controller for cost or performance reasons.

The CPU's bus (often called the local bus) usually runs at some submultiple of the CPU's frequency. Typical local bus frequencies include 66 MHz, 100 MHz, 200 MHz, 400 MHz, and, possibly, beyond<sup>4</sup>. Usually, only memory and a few selected peripherals (e.g., the PCI Bus Controller) sit on the CPU's bus and operate at this high frequency. Since the CPU's bus is typically 64 bits wide (for Pentium and later processors) and it is theoretically possible to achieve one data transfer per cycle, the CPU's bus has a maximum possible data transfer rate (or maximum bandwidth) of eight times the clock frequency (e.g., 800 megabytes/second for a 100 Mhz bus). In practice, CPU's rarely achieve the maximum data transfer rate, but they do achieve some percentage of this, so the faster the bus, the more data can move in and out of the CPU (and caches) in a given amount of time.

The PCI bus comes in several configurations. The base configuration has a 32-bit wide data bus operating at 33 MHz. Like the CPU's local bus, the PCI is theoretically capable of transferring data on each clock cycle. This provides a theoretical maximum of 132 MBytes/second data transfer rate (33 MHz times four bytes). In practice, the PCI bus doesn't come anywhere near this level of performance except in short bursts. Whenever the CPU wishes to access a peripheral on the PCI bus, it must negotiate with other peripheral devices for the right to use the bus. This negotiation can take several clock cycles before the PCI controller grants the CPU the bus. If a CPU writes a sequence of values to a peripheral a double word per bus request, then the negotiation takes the majority of the time and the data transfer rate drops dramatically. The only way to achieve anywhere near the maximum theoretical bandwidth on the bus is to use a DMA controller and move blocks of data. In this block mode the DMA controller can negotiate just once for the bus and transfer a fair sized block of data without giving up the bus between each transfer. This "burst mode" allows the device to move lots of data quickly.

There are a couple of enhancements to the PCI bus that improve performance. Some PCI busses support a 64-bit wide data path. This, obviously, doubles the maximum theoretical data transfer rate. Another enhancement is to run the bus at 66 MHz, which also doubles the throughput. In theory, you could have a 64-bit wide 66 MHz bus that quadruples the data transfer rate (over the performance of the baseline configuration). Few systems or peripherals currently support anything other than the base configuration, but these optional enhancements to the PCI bus allow it to grow with the CPU as CPUs increase their performance.

The ISA bus is a carry over from the original PC/AT computer system. This bus is 16 bits wide and operates at 8 MHz. It requires four cycles for each bus cycle. This this and other reasons, the ISA bus is capable of about only one data transmission per microsecond. With a 16-bit wide bus, data transfer is limited to about two megabytes per second. Much slower than the CPU's local bus and the PCI bus. Generally, you would only attach low-speed devices like an RS-232 communications device, a modem, or a parallel printer to the ISA bus. Most other devices (disks, scanners, network cards, etc.) are too fast for the ISA bus. The ISA bus is really only capable of supporting low-speed and medium speed devices.

Note that accessing the ISA bus on most systems involves first negotiating for the PCI bus. The PCI bus is so much faster than the ISA bus that this has very little impact on the performance of peripherals on the ISA bus. Therefore, there is very little difference to be gained by connecting the ISA controller directly to the CPU's local bus.

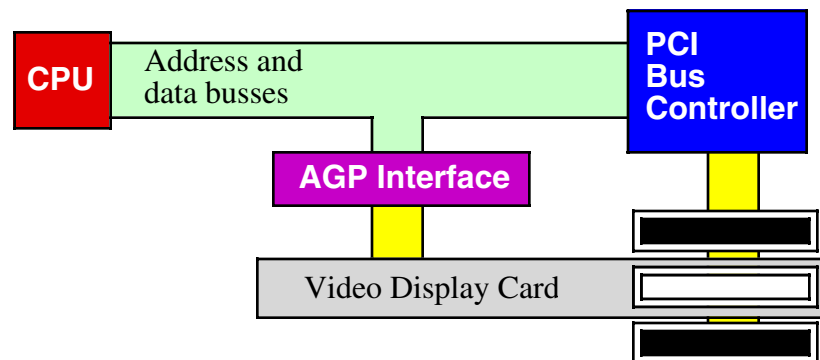
---

4. 400 MHz was the maximum CPU bus frequency as this was being written.

## 7.7 The AGP Bus

Video display cards are a very special peripheral that need the maximum possible amount of bus bandwidth to ensure quick screen updates and fast graphic operations. Unfortunately, if the CPU has to constantly negotiate with other peripherals for the use of the PCI bus, graphics performance can suffer. To overcome this problem, video card designers created the AGP (Advanced Graphics Port) interface between the CPU and the video display card.

The AGP is a secondary bus interface that a video card uses in addition to the PCI bus. The AGP connection lets the CPU quickly move data to and from the video display RAM. The PCI bus provides a connection to the other I/O ports on the video display card (see Figure 7.7). Since there is only one AGP port per system, only one card can use the AGP and the system never has to negotiate for access to the AGP bus.



**Figure 7.7 AGP Bus Interface**

## 7.8 Buffering

If a particular I/O device produces or consumes data faster than the system is capable of transferring data to that device, the system designer has two choices: provide a faster connection between the CPU and the device or slow down the rate of transfer between the two.

Creating a faster connection is possible if the peripheral device is already connected to a slow bus like ISA. Another possibility is going to a wider bus (e.g., to the 64-bit PCI bus) to increase bandwidth, or to use a bus with a higher frequency (e.g., a 66 MHz bus rather than a 33 MHz bus). Systems designers can sometimes create a faster interface to the bus; the AGP connection is a good example. However, once you're using the fastest bus available on the system, improving system performance by selecting a faster connection to the computer can be very expensive.

The other alternative is to slow down the transfer rate between the peripheral and the computer system. This isn't always as bad as it seems. Most high-speed devices don't transfer data at a constant rate to the system. Instead, devices typically transfer a block of data rapidly and then sit idle for some period of time. Although the burst rate is high (and faster than the CPU or system can handle), the average data transfer rate is usually lower than what the CPU/system can handle. If you could average out the peaks and transfer some of the data when the peripheral is inactive, you could easily move data between the peripheral and the computer system without resorting to an expensive, high-bandwidth, solution.

The trick is to use memory to buffer the data on the peripheral side. The peripheral can rapidly fill this buffer with data (or extract data from the buffer). Once the buffer is empty (or full) and the peripheral device is inactive, the system can refill (or empty) the buffer at a sustainable rate. As long as the average data rate of the peripheral device is below the maximum bandwidth the system will support, and the buffer is large enough to hold bursts of data to/from the peripheral, this scheme lets the peripheral communicate with the system at a lower data transfer rate than the device requires during burst operation.

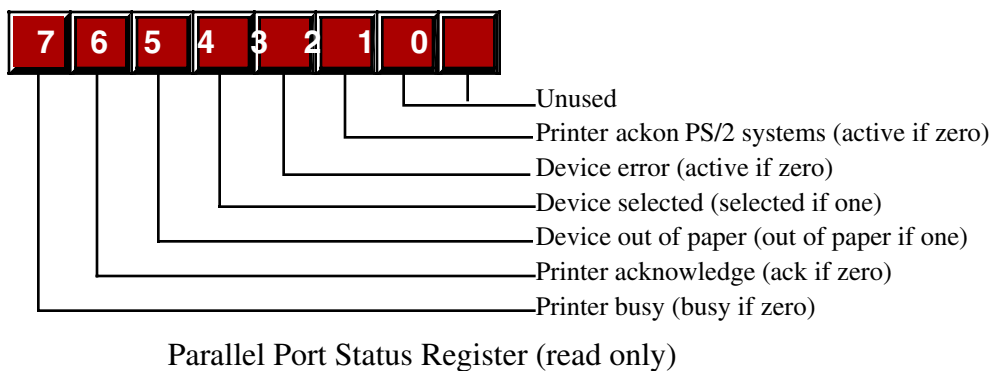
## 7.9 Handshaking

Many I/O devices cannot accept data at an arbitrary rate. For example, a Pentium based PC is capable of sending several hundred million characters a second to a printer, but that printer is (probably) unable to print that many characters each second. Likewise, an input device like a keyboard is unable to provide several million keystrokes per second (since it operates at human speeds, not computer speeds). The CPU needs some mechanism to coordinate data transfer between the computer system and its peripheral devices.

One common way to coordinate data transfer is to provide some status bits in a secondary input port. For example, a one in a single bit in an I/O port can tell the CPU that a printer is ready to accept more data, a zero would indicate that the printer is busy and the CPU should not send new data to the printer. Likewise, a one bit in a different port could tell the CPU that a keystroke from the keyboard is available at the keyboard data port, a zero in that same bit could indicate that no keystroke is available. The CPU can test these bits prior to reading a key from the keyboard or writing a character to the printer.

Using status bits to indicate that a device is ready to accept or transmit data is known as handshaking. It gets this name because the protocol is similar to two people agreeing on some method of transfer by a hand shake.

Figure 7.8 shows the layout of the parallel printer port's status register. For the LPT1: printer interface, this port appears at I/O address \$379. As you can see from this diagram, bit seven determines if the printer is capable of receiving data from the system; this bit will contain a one when the printer is capable of receiving data.



**Figure 7.8 The Parallel Port Status Port**

The following short program segment will continuously loop while the H.O. bit of the printer status register contains zero and will exit once the printer is ready to accept data:

```
mov( $379, dx );
repeat

    in( dx, al );
    and( $80, al );    // Clears Z flag if bit seven is set.

until( @nz );

// Okay to write another byte to the printer data port here.
```

The code above begins by setting DX to \$379 since this is the I/O address of the printer status port. Within the loop the code reads a byte from the status port (the IN instruction) and then tests the H.O. bit of the port using the AND instruction. Note that logically ANDing the AL register with \$80 will produce zero if the H.O. bit of AL was zero (that is, if the byte read from the input port was zero). Similarly, logically and-ing AL with \$80 will produce \$80 (a non-zero result) if the H.O. bit of the printer status port was set. The

80x86 zero flag reflects the result of the AND instruction; therefore, the zero flag will be set if AND produces a zero result, it will be reset otherwise. The REPEAT..UNTIL loop repeats this test until the AND instruction produces a non-zero result (meaning the H.O. bit of the status port is set).

One problem with using the AND instruction to test bits as the code above is that you might want to test other bits in AL once the code leaves the loop. Unfortunately, the "and( \$80, al );" instruction destroys the values of the other bits in AL while testing the H.O. bit. To overcome this problem, the 80x86 supports another form of the AND instruction – TEST. The TEST instruction works just like AND except it only updates the flags; it does not store the result of the logical AND operation back into the destination register (AL in this case). One other advantage to TEST is that it only reads its operands, so there are less problems with data hazards when using this instruction (versus AND). Also, you can safely use the TEST instruction directly on read-only memory-mapped I/O ports since it does not write data back to the port. As an example, let's recode the previous loop using the TEST instruction:

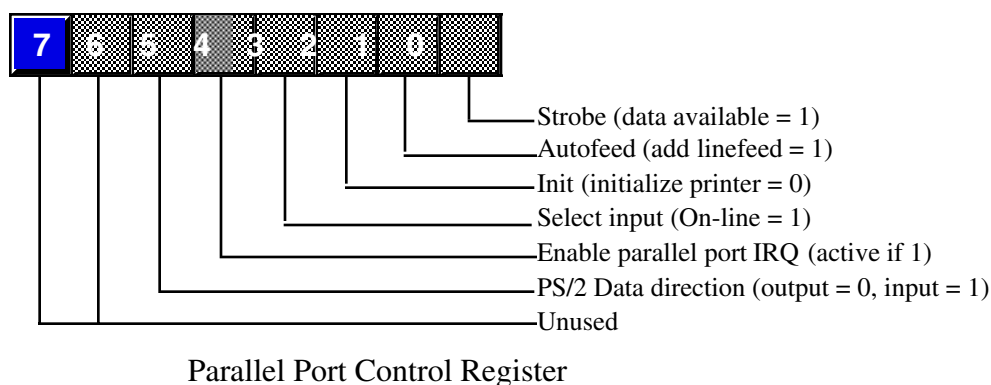
```
mov( $379, dx );
repeat

    in( dx, al );
    test( $80, al ); // Clears Z flag if bit seven is set.

until( @nz );

// Okay to write another byte to the printer data port here.
```

Once the H.O. bit of the printer status port is set, it's okay to transmit another byte to the printer. The computer can make a byte available by storing the byte data into I/O address \$378 (for LPT1:). However, simply storing data to this port does not inform the printer that it can take the byte. The system must complete the other half of the handshake operation and send the printer a signal to indicate that a byte is available.



**Figure 7.9 The Parallel Port Command Register**

Bit zero (the strobe line) must be set to one and then back to zero when the CPU makes data available for the printer (the term "strobe" suggests that the system pulses this line in the command port). In order to pulse this bit without affecting the other control lines, the CPU must first read this port, OR a one into the L.O. bit, write the data to the port, then mask out the L.O. bit using an AND instruction, and write the final result back to the control port again. Therefore, it takes three accesses (a read and two writes) to send the strobe to the printer. The following code handles this transmission:

```
mov( $378, dx ); // Data port address
mov( Data2Xmit, al ); // Send the data to the printer.
out( al, dx );

mov( $37a, dx ); // Point DX at the control port.
```

```

in( dx, al );           // Get the current port setting.
or( 1, al );           // Set the L.O. bit.
out( al, dx );          // Set the strobe line high.
and( $fe, al );         // Clear the L.O. bit.
out( al, dx );          // Set the strobe line low.

```

The code above would normally follow the REPEAT..UNTIL loop in the previous example. To transmit a second byte to the printer you would jump back to the REPEAT..UNTIL loop and wait for the printer to consume the current byte.

Note that it takes a minimum of five I/O port accesses to transmit a byte to the printer use the code above (minimum one IN instruction in the REPEAT..UNTIL loop plus four instructions to send the byte and strobe). If the parallel port is connected to the ISA bus, this means it takes a minimum of five microseconds to transmit a single byte; that works out to less than 200,000 bytes per second. If you are sending ASCII characters to the printer, this is far faster than the printer can print the characters. However, if you are send a bitmap or a Postscript file to the printer, the printer port bandwidth limitation will become the bottleneck since it takes considerable data to print a page of graphics. For this reason, most graphic printers use a different technique than the above to transmit data to the printer (some parallel ports support DMA in order to get the data transfer rate up to a reasonable level).

---

## 7.10 Time-outs on an I/O Port

One problem with the REPEAT..UNTIL loop in the previous section is that it could spin indefinitely waiting for the printer to become ready to accept additional input. If someone turns the printer off or the printer cable becomes disconnected, the program could freeze up, forever waiting for the printer to become available. Usually, it's a good idea to indicate to the user that something has gone wrong rather than simply freezing up the system. A typical way to handle this problem is using a time-out period to determine that something is wrong with the peripheral device.

With most peripheral devices you can expect some sort of response within a reasonable amount of time. For example, most printers will be ready to accept additional character data within a few seconds of the last transmission (worst case). Therefore, if 30 seconds or more have passed since the printer was last willing to accept a character, this is probably an indication that something is wrong. If the program could detect this, then it could ask the user to check the printer and tell the program to resume printing once the problem is resolved.

Choosing a good time-out period is not an easy task. You must carefully balance the irritation of having the program constantly ask you what's wrong when there is nothing wrong with the printer (or other device) with the program locking up for long periods of time when there is something wrong. Both situations are equally annoying to the end user.

Any easy way to create a time-out period is to count the number of times the program loops while waiting for a handshake signal from a peripheral. Consider the following modification to the REPEAT..UNTIL loop of the previous section:

```

mov( $379, dx );
mov( 30_000_000, ecx );
repeat

    dec( ecx );           // Count down to see if the time-out has expired.
    breakif( @z );        // Leave this loop if ecx counted down to zero.

    in( dx, al );
    test( $80, al );      // Clears Z flag if bit seven is set.

until( @nz );

if( ecx = 0 ) then

    // We had a time-out error.

```

```

else

    // Okay to write another byte to the printer data port here.

endif;

```

The code above will exit once the printer is ready to accept data or when approximately 30 seconds have expired. You may question the 30 second figure. After all, a software based loop (counting down ECX to zero) should run at different speeds on different processors. However, don't miss the fact that there is an IN instruction inside this loop. The IN instruction reads a port on the ISA bus and that means this instruction will take approximately one microsecond to execute (about the fastest operation on the ISA bus). Hence, every one million times through the loop will take about a second ( $\pm 50\%$ , but close enough for our purposes). This is true regardless of the CPU frequency.

The 80x86 provides a couple of instructions that are quite useful for implementing time-outs in a polling loop: LOOPZ and LOOPNZ. We'll consider the LOOPZ instruction here since it's perfect for the loop above. The LOOPZ instruction decrements the ECX register by one and falls through to the next instruction if ECX contains zero. If ECX does not contain zero, then this instruction checks the zero flag setting prior to decrementing ECX; if the zero flag was set, then LOOPZ transfers control to a label specified as LOOPZ's operand. Consider the implementation of the previous REPEAT..UNTIL loop using LOOPZ:

```

mov( $379, dx );
mov( 30_000_000, ecx );
PollingLoop:

    in( dx, al );
    test( $80, al );    // Clears Z flag if bit seven is set.

loopz PollingLoop;    // Repeat while zero and ECX<>0.

if( ecx = 0 ) then

    // We had a time-out error.

else

    // Okay to write another byte to the printer data port here.

endif;

```

Notice how this code doesn't need to explicitly decrement ECX and check to see if it became zero.

**Warning:** the LOOPZ instruction can only transfer control to a label with  $\pm 127$  bytes of the LOOPZ instruction. Due to a design problem, HLA cannot detect this problem. If the branch range exceeds 127 bytes HLA will not report an error. Instead, MASM will report the error when it assembles HLA's output. Since it's somewhat difficult to track down these problems in the MASM listing, the best solution is to never use the LOOPZ instruction to jump more than a few instructions in your code. It's perfect for short polling loops like the one above, it's not suitable for branching large distances.

---

## 7.11 Interrupts and Polled I/O

*Polling* is constantly testing a port to see if data is available. That is, the CPU polls (asks) the port if it has data available or if it is capable of accepting data. The REPEAT..UNTIL loop in the previous section is a good example of polling. The CPU continually polls the port to see if the printer is ready to accept data. Polled I/O is inherently inefficient. Consider what happens in the previous section if the printer takes ten seconds to accept another byte of data – the CPU spins in a loop doing nothing (other than testing the printer status port) for those ten seconds.

In early personal computer systems, this is exactly how a program would behave; when it wanted to read a key from the keyboard it would poll the keyboard status port until a key was available. Such computers could not do other operations while waiting for the keyboard.

The solution to this problem is to provide an *interrupt mechanism*. An interrupt is an external hardware event (such as the printer becoming ready to accept another byte) that causes the CPU to interrupt the current instruction sequence and call a special interrupt service routine. (ISR). An interrupt service routine typically saves all the registers and flags (so that it doesn't disturb the computation it interrupts), does whatever operation is necessary to handle the source of the interrupt, it restores the registers and flags, and then it resumes execution of the code it interrupted. In many computer systems (e.g., the PC), many I/O devices generate an interrupt whenever they have data available or are able to accept data from the CPU. The ISR quickly processes the request in the background, allowing some other computation to proceed normally in the foreground.

An interrupt is essentially a procedure call that the hardware makes (rather than explicit call to some procedure, like a call to the *stdout.put* routine). The most important thing to remember about an interrupt is that it can pause the execution of some program at any point between two instructions when an interrupt occurs. Therefore, you typically have no guarantee that one instruction always executes immediately after another in the program because an interrupt could occur between the two instructions. If an interrupt occurs in the middle of the execution of some instruction, then the CPU finishes that instruction before transferring control to the appropriate interrupt service routine. However, the interrupt generally interrupts execution before the start of the next instruction<sup>5</sup>. Suppose, for example, that an interrupt occurs between the execution of the following two instructions:

```
add( i, eax );
               <---- Interrupt occurs here.
mov( eax, j );
```

When the interrupt occurs, control transfers to the appropriate ISR that handles the hardware event. When that ISR completes and executes the IRET (interrupt return) instruction, control returns back to the point of interruption and execution of the original code continues with the instruction immediately after the point of interrupt (e.g., the MOV instruction above). Imagine an interrupt service routine that executes the following code:

```
mov( 0, eax );
iret;
```

If this ISR executes in response to the interrupt above, then the main program will not produce a correct result. Specifically, the main program should compute "j := eax + i;" Instead, it computes "j := 0;" (in this particular case) because the interrupt service routine sets EAX to zero, wiping out the sum of *i* and the previous value of EAX. This highlights a very important fact about ISRs: **ISRs must preserve all registers and flags whose values they modify**. If an ISR does not preserve some register or flag value, this will definitely affect the correctness of the programs running when an interrupt occurs. Usually, the ISR mechanism itself preserves the flags (e.g., the interrupt pushes the flags onto the stack and the IRET instruction restores those flags). However, the ISR itself is responsible for preserving any registers that it modifies.

Although the preceding discussion makes it clear that ISRs must preserve registers and the flags, your ISRs must exercise similar care when manipulating any other resources the ISR shares with other processes. This includes variables, I/O ports, etc. Note that preserving the values of such objects isn't always the correct solution. Many ISRs communicate their results to the foreground program using shared variables. However, as you will see, the ISR and the foreground program must coordinate access to shared resources or they may produce incorrect results. Writing code that correctly works with shared resources is a difficult challenge; the possibility of subtle bugs creeping into the program is very great. We'll consider some of these issues a little later in this chapter; the messy details will have to wait for a later volume of this text.

---

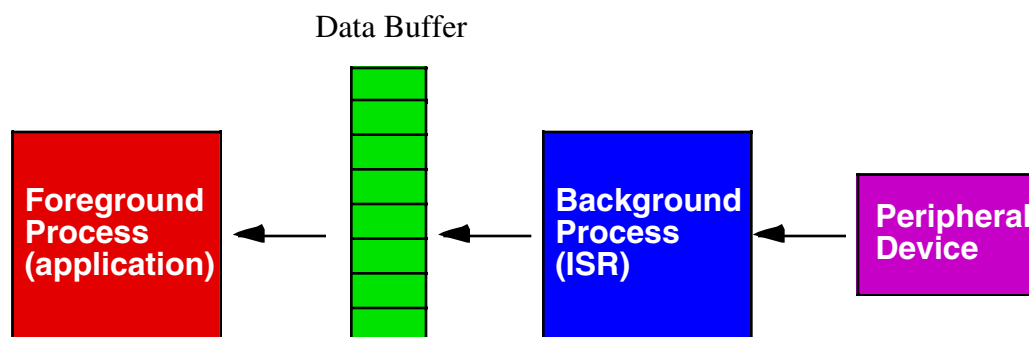
5. The situation is somewhat fuzzy if you have pipelines and superscalar operation. Exactly what instruction does an interrupt precede if there are multiple instructions executing simultaneously? The answer is somewhat irrelevant, however, since the interrupt does take place between the execution of some pair of instructions; in reality, the interrupt may occur immediately after the last instruction to enter the pipeline when the interrupt occurs. Nevertheless, the system does interrupt the execution of the foreground process after the execution of some instruction.

CPUs that support interrupts must provide some mechanism that allows the programmer to specify the address of the ISR to execute when an interrupt occurs. Typically, an interrupt vector is a special memory location that contains the address of the ISR to execute when an interrupt occurs. PCs typically support up to 16 different interrupts.

After an ISR completes its operation, it generally returns control to the foreground task with a special “return from interrupt” instruction. On the Y86 hypothetical processor, for example, the IRET (interrupt return) instruction handles this task. This same instruction does a similar task on the 80x86. An ISR should always end with this instruction so the ISR can return control to the program it interrupted.

## 7.12 Using a Circular Queue to Buffer Input Data from an ISR

A typical interrupt-driven input system uses the ISR to read data from an input port and buffer it up whenever data becomes available. The foreground program can read that data from the buffer at its leisure without losing any data from the port. A typical foreground/ISR arrangement appears in Figure 7.10. In this diagram the ISR reads a value from the peripheral device and then stores the data into a common buffer that the ISR shares with the foreground application. Sometime later, the foreground process removes the data from the buffer. If (during a burst of input) the device and ISR produce data faster than the foreground application reads data from the buffer, the ISR will store up multiple unread data values in the buffer. As long as the average consumption rate of the foreground process matches the average production rate of the ISR, and the buffer is large enough to hold bursts of data, there will be no lost data.



The background process produces data (by reading it from the device) and places it in the buffer. The foreground process consumes data by removing it from the buffer.

**Figure 7.10** Interrupt Service Routine as a Data Produce/Application as a Data Consumer

If the foreground process in Figure 7.10 consumes data faster than the ISR produces it, sooner or later the buffer will become empty. When this happens the foreground process will have to wait for the background process to produce more data. Typically the foreground process would poll the data buffer (or, in a more advanced system, block execution) until additional data arrives. Then the foreground process can easily extract the new data from the buffer and continue execution.

There is nothing special about the data buffer. It is just a block of contiguous bytes in memory and a few additional pieces of information to maintain the list of data in the buffer. While there are lots of ways to maintain data in a buffer such as this one, probably the most popular technique is to use a *circular buffer*. A typical circular buffer implementation contains three objects: an array that holds the actual data, a pointer to the next available data object in the buffer, and a length value that specifies how many objects are currently in the buffer.

In the next volume of this series you will see how to declare and use arrays. However, in Chapter Two of this volume you saw how to allocate a block of data in the DATA or STATIC sections (see “The Data and

Static Sections” on page 159) or how to use malloc to allocate a block of bytes (see “Dynamic Memory Allocation and the Heap Segment” on page 181). For our purposes, declaring a block of bytes in the STATIC section is just fine; the following code shows one way to set aside 16 bytes for a buffer:

```
static
    buffer:    byte := 0;                // Reserves one byte.
              byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 // 15 additional bytes.
```

Of course, this technique would not be useful if you wanted to set aside storage for a really large buffer, but it works fine for small buffers (like our example above). See the discussion of arrays in the next volume if you need to allocate storage for a larger buffer.

In addition to the buffer data itself, a circular buffer also needs at least two other values: an index into the buffer that specifies where the next available data object appears and a count of valid items in the buffer. Given that the 80x86’s addressing modes all use 32-bit registers, we’ll find it most convenient to use a 32-bit unsigned integer for this purpose even though the index and count values never exceed 16. The declaration for these values might be the following:

```
static
    index: uns32 := 0; // Start with first element of the array.
    count: uns32 := 0; // Initially, there is no data in the array.
```

The data producer (the ISR in our example) inserts data into the buffer by following these steps:

- Check the count. If the count is equal to the buffer size, then the buffer is full and some corrective action is necessary.
- Store the new data object at location  $((\text{index} + \text{count}) \bmod \text{buffer\_size})$ .
- Increment the count variable.

Suppose that the producer wishes to add a character to the initially empty buffer. The *count* is zero so we don’t have to deal with a buffer overflow. The *index* value is also zero, so  $((\text{index} + \text{count}) \bmod 16)$  is zero and we store our first data byte at index zero in the array. Finally, we increment *count* by one so that the producer will put the next byte at offset one in the array of bytes.

If the consumer never removes any bytes and the producer keeps producing bytes, sooner or later the buffer will fill up and *count* will hit 16. Any attempt to insert additional data into the buffer is an error condition. The producer needs to decide what to do at that point. Some simple routines may simply ignore any additional data (that is, any additional incoming data from the device will be lost). Some routines may signal an exception and leave it up to the main application to deal with the error. Some other routines may attempt to expand the buffer size to allow additional data in the buffer. The correction action is application-specific. In our examples we’ll assume the program either ignores the extra data or immediately stops the program if a buffer overflow occurs.

You’ll notice that the producer stores the data at location  $((\text{index} + \text{count}) \bmod \text{buffer\_size})$  in the array. This calculation, as you’ll soon see, is how the circular buffer obtains its name. HLA does provide a MOD instruction that will compute the remainder after the division of two values, however, most buffer routines don’t compute remainder using the MOD instruction. Instead, most buffer routines rely on a cute little trick to compute this value much more efficiently than with the MOD instruction. The trick is this: if a buffer’s size is a power of two (16 in our case), you can compute  $(x \bmod \text{buffer\_size})$  by logically ANDing *x* with *buffer\_size* - 1. In our case, this means that the following instruction sequence computes  $((\text{index} + \text{count}) \bmod 16)$  in the EBX register:

```
mov( index, ebx );
add( count, ebx );
and( 15, ebx );
```

Remember, this trick only works if the buffer size is an integral power of two. If you look at most programs that use a circular buffer for their data, you’ll discover that they commonly use a buffer size that is an integral power of two. The value is not arbitrary; they do this so they can use the AND trick to efficiently compute the remainder.

To remove data from the buffer, the consumer half of the program follows these steps:

- The consumer first checks to the count to see if there is any data in the buffer. If not, the consumer waits until data is available.
- If (or when) data is available, the consumer fetches the value at the location *index* specifies within the buffer.
- The consumer then decrements the *count* and computes  $index := (index + 1) \text{ MOD } buffer\_size$ .

To remove a byte from the circular buffer in our current example, you'd use code like the following:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count );             // Note that we've removed a character.
inc( ebx );               // Index := Index + 1;
and( 15, ebx );           // Index := (index + 1) mod 16;
mov( ebx, index );        // Save away the new index value.
```

As the consumer removes data from the circular queue, it advances the index into the array. If you're wondering what happens at the end of the array, well that's the purpose of the MOD calculation. If index starts at zero and increments with each character, you'd expect the sequence 0, 1, 2, ... At some point or another the index will exceed the bounds of the buffer (i.e., when *index* increments to 16). However, the MOD operation resets this value back to zero (since 16 MOD 16 is zero). Therefore, the consumer, after that point, will begin removing data from the beginning of the buffer.

Take a close look at the REPEAT..UNTIL loop in the previous code. At first blush you may be tempted to think that this is an infinite loop if *count* initially contains zero. After all, there is no code in the body of the loop that modifies *count*'s value. So if *count* contains zero upon initial entry, how does it ever change? Well, that's the job of the ISR. When an interrupt comes along the ISR suspends the execution of this loop at some arbitrary point. Then the ISR reads a byte from the device, puts the byte into the buffer, and updates the *count* variable (from zero to one). Then the ISR returns and the consumer code above resumes where it left off. On the next loop iteration, however, *count*'s value is no longer zero, so the loop falls through to the following code. This is a classic example of how an ISR communicates with a foreground process – by writing a value to some shared variable.

There is a subtle problem with the producer/consumer code in this section. It will fail if the producer is attempting to insert data into the buffer at exactly the same time the consumer is removing data. Consider the following sequence of instructions:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count );             // Note that we've removed a character.

*** Assume the interrupt occurs here, so we begin executing
*** the data insertion sequence:

mov( index, ebx );
add( count, ebx );
and( 15, ebx );
mov( al, buffer[ebx] );
inc( count );
```

\*\*\* now the ISR returns to the consumer code (assume we've preserved EBX):

```
inc( ebx );           // Index := Index + 1;
and( 15, ebx );       // Index := (index + 1) mod 16;
mov( ebx, index );    // Save away the new index value.
```

The problem with this code, which is very subtle, is that the consumer has decremented the count variable and an interrupt occurs before the consumer can update the index variable as well. Therefore, upon arrival into the ISR, the count value and the index value are inconsistent. That is, `index+count` now points at the last value placed in the buffer rather than the next available location. Therefore, the ISR will overwrite the last byte in the buffer rather than properly placing this byte after the (current) last byte. Worse, once the ISR returns to the consumer code, the consumer will update the *index* value and effectively add a byte of garbage to the end of the circular buffer. The end result is that we wipe out the next to last value in the buffer and add a garbage byte to the end of the buffer.

Note that this problem doesn't occur all the time, or even frequently for that matter. In fact, it only occurs in the very special case where the interrupt occurs between the `"dec( count );"` and `"mov(ebx, index);"` instructions in this code. If this code executes a very tiny percentage of the time, the likelihood of encountering this error is quite small. This may seem good, but this is actually worse than having the problem occur all the time; the fact that the problem rarely occurs just means that it's going to be really hard to find and correct this problem when you finally do detect that something has gone wrong. ISRs and concurrent programs are among the most difficult programs in the world to test and debug. The best solution is to carefully consider the interaction between foreground and background tasks when writing ISRs and other concurrent programs. In a later volume, this text will consider the issues in concurrent programming, for now, be very careful about using shared objects in an ISR.

There are two ways to correct the problem that occurs in this example. One way is to use a pair of (somewhat) independent variables to manipulate the queue. The original PC's type ahead keyboard buffer, for example, used two index variables rather than an index and a count to maintain the queue. The ISR would use one index to insert data and the foreground process would use the second index to remove data from the buffer. The only sharing of the two pointers was a comparison for equality, which worked okay even in an interrupt environment. Here's how the code worked:

```
// Declarations

static
  buffer: byte := 0; byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
  Ins: uns32 := 0; // Insert bytes starting here.
  Rmv: uns32 := 0; // Remove bytes starting here.

// Insert a byte into the queue (the ISR ):

  mov( Ins, ebx );
  inc( ebx );
  and( 15, ebx );
  if( ebx <> Rmv ) then

    mov( al, buffer[ ebx ] );
    mov( ebx, Ins );

  else

    // Buffer overflow error.
    // Note that we don't update INS in this case.

  endif;

// Remove a byte from the queue (the consumer process).
```

```

mov( Rmv, ebx );
repeat

    // Wait for data to arrive

until( ebx <> Ins );
mov( buffer[ ebx ], al );
inc( ebx );
and( 15, ebx );
mov( ebx, Rmv );

```

If you study this code (very) carefully, you'll discover that the two code sequences don't interfere with one another. The difference between this code and the previous code is that the foreground and background processes don't write to a (control) variable that the other routine uses. The ISR only writes to *Ins* while the foreground process only writes to *Rmv*. In general, this is not a sufficient guarantee that the two code sequences won't interfere with one another, but it does work in this instance.

One drawback to this code is that it doesn't fully utilize the buffer. Specifically, this code sequence can only hold 15 characters in the buffer; one byte must go unused because this code determines that the buffer is full when the value of *Ins* is one less than *Rmv* (MOD 16). When the two indices are equal, the buffer is empty. Since we need to test for both these conditions, we can't use one of the bytes in the buffer.

A second solution, that many people prefer, is to protect that section of code in the foreground process that could fail if an interrupt comes along. There are lots of ways to protect this *critical section*<sup>6</sup> of code. Alas, most of the mechanisms are beyond the scope of this chapter and will have to wait for a later volume in this text. However, one simple way to protect a critical section is to simply disable interrupts during the execution of that code. The 80x86 family provides two instructions, CLI and STI that let you enable and disable interrupts. The CLI instruction (clear interrupt enable flag) disables interrupts by clearing the "I" bit in the flags register (this is the interrupt enable flag). Similarly, the STI instruction enables interrupts by setting this flag. These two instructions use the following syntax:

```

cli();    // Disables interrupts from this point forward...
.
.
.
sti();    // Enables interrupts from this point forward...

```

You can surround a critical section in your program with these two instructions to protect that section from interrupts. The original consumer code could be safely written as follows:

```

// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

cli();                                // Protect the following critical section.
mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count );             // Note that we've removed a character.
inc( ebx );               // Index := Index + 1;
and( 15, ebx );           // Index := (index + 1) mod 16;
mov( ebx, index );        // Save away the new index value.
sti();                   // Critical section is done, restore interrupts.

```

---

6. A critical section is a region of code during which certain resources have to be protected from other processes. For example, the consumer code that fetches data from the buffer needs to be protected from the ISR.

Perhaps a better sequence to use is to push the EFLAGS register (that contains the I flag) and turn off the interrupts. Then, rather than blindly turning interrupts back on, you can restore the original I flag setting using a POPFD instruction:

```
// Remove the character from the buffer.

pushfd();           // Preserve current I flag value.
cli();              // Protect the following critical section.
mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count );       // Note that we've removed a character.
inc( ebx );         // Index := Index + 1;
and( 15, ebx );     // Index := (index + 1) mod 16;
mov( ebx, index );  // Save away the new index value.
popfd();            // Restore original I flag value
```

This mechanism is arguably safer since it doesn't turn the interrupts on even if they were already off before executing this sequence.

In our simple example (with a single producer and a single consumer) there is no need to protect the code in the ISR. However, if it were possible for two different ISRs to insert data into the buffer, and one ISR could interrupt another, then you would have to protect the code inside the ISR as well.

You must be very careful about turning the interrupts on and off. If you turn the interrupts off and forget to turn them back on, the next time you enter a loop like one of the REPEAT..UNTIL loops in this section the program will lock up because the loop control variable (*count*) will never change if an ISR cannot execute and update its value. This situation is called *deadlock* and you must take special care to avoid it.

---

## 7.13 Using a Circular Queue to Buffer Output Data for an ISR

You can also use a circular buffer to hold data waiting for transmission. For example, a program can buffer up data in bursts while an output device is busy and then the output device can empty the buffer at a steady state. The queuing and dequeuing routines are very similar to those found in the previous section with one major difference: output devices don't automatically initiate the transfer like input devices. This problem is a source of many bugs in output buffering routines and one that we'll pay careful attention to in this section.

As noted above, one advantage of an input device is that it automatically interrupts the system whenever data is available. This activates the corresponding ISR that reads the data from the device and places the data in the buffer. No special processing is necessary to prepare the interrupt system for the next input value.

There is a subtle difference between the interrupts an input device generates and the interrupts an output device generates. An input device generates an interrupt when data is available, output devices generate an interrupt when they are ready to accept more data. For example, a keyboard device generates an interrupt when the user presses a key and the system has to read the character from the keyboard. A printer device, on the other hand, generates an interrupt once it is done with the last character transmitted to it and it's ready to accept another character. Whenever the user presses a keyboard for the very first time, the system will generate an interrupt in response to that event. However, the printer does not generate an interrupt when the system first powers up to tell the system that it's ready to accept a character. Even if it did, the system would ignore the interrupt since it (probably) doesn't have any data to transmit to the printer at that point. Later, when the system puts data in the printer's buffer for transmission, there is no interrupt that activates the ISR to remove a character from the buffer and send it to the printer. The printer device only sends interrupts when it is done processing a character; if it isn't processing any characters, it won't generate any interrupts.

This creates a bit of a problem. If the foreground process places characters in the queue and the background process (the ISR, which is the consumer in this case) only removes those characters when an interrupt occurs, the system will never activate the ISR since the device isn't currently processing anything. To correct this problem, the producer code (the foreground process) must maintain a flag that indicates whether the output device is currently processing a character; if so, then the producer can simply queue up the character in the buffer. If the device is not currently processing any data, then the producer should send the data

directly to the device rather than queue up the data in the buffer<sup>7</sup>. Old time programmers refer to this as "priming the pump" since we have to put data in the transmission pipeline in order to get the process working properly.

Once the producer "primes the pump" the process continues automatically as long as there is data in the buffer. After the output device processes the current byte it generates an interrupt. The ISR removes a byte from the buffer and transmits this data to the device. When that byte completes transmission the device generates another interrupt and the process repeats. This process repeats automatically as long as there is data in the buffer to transmit to the output device.

When the ISR transmits the last character from the buffer, the output device still generates an interrupt at the end of the transmission. The ISR, upon noting that the buffer is empty, returns without sending any new data to the output device. Since there is no pending data transmission to the output device, there will be no new interrupts to activate the ISR when new data appears in the buffer. Once again the foreground process (producer) will have to prime the pump to get the process going when it attempts to put data in the buffer.

Perhaps the easiest way to handle this process is to use a boolean variable to indicate whether the output device is currently transmitting data (and will generate an interrupt to process the next byte). If the flag is set, the foreground process can simply enqueue the data; if the flag is clear, the foreground process must transmit the data directly to the device (or call the code that does this). In this latter case, the foreground process must also set the flag to denote a transmission in progress.

Here is some code that can implement this functionality:

```
static
OutBuf: byte := 0; byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
Index: uns32 := 0;
Count: uns32 := 0;
Xmitting: boolean := false; // Flag to denote transmission in progress.
.
.
.
// Code to enqueue a byte (foreground process executes this)

if( Count = 16 ) then

    // Error, buffer is full. Do whatever processing is necessary to
    // deal with this problem.
    .
    .
    .

elseif( Xmitting ) then

    // If we're currently transmitting data, just add the byte to the queue.

    pushfd();           // Critical region! Turn off the interrupts.
    cli();
    mov( Index, ebx ); // Store the new byte at address (Index+Count) mod 16
    add( Count, ebx );
    and( 15, ebx );
    mov( al, OutBuf[ ebx ] );
    inc( Count );
    popfd();           // Restore the interrupt flag's value.

else

    // The buffer is empty and there is no character in transmission.
```

---

7. Another possibility is to go ahead and queue up the data and then manually activate the code that dequeues the data and sends it to the output device.

```

// Do whatever is necessary to transmit the character to the output
// device.
.
.
.

// Be sure to set the Xmitting flag since a character is now being
// transmitted to the output device.

mov( true, Xmitting );

endif;
.
.
.
// Here's the code that would appear inside the ISR to remove a character
// from the buffer and send it to the output device. The system calls this
// ISR whenever the device finishes processing some character.
// (Presumably, the ISR preserves all registers this code sequence modifies)

if( Count > 0 ) then

    // Okay, there are characters in the buffer. Remove the first one
    // and transmit it to the device:

    mov( Index, ebx );
    mov( OutBuf[ ebx ], al ); // Get the next character to output
    inc( ebx );              // Point Index at the next byte in the
    and( 15, ebx );          // circular buffer.
    mov( ebx, Index );
    dec( Count );            // Decrement count since we removed a char.

    << At this point, do whatever needs to be done in order to
        transmit a character to the output device >>
    .
    .
    .

else

    // At this point, the ISR was called but the buffer is empty.
    // Simply clear the Xmitting flag and return. (this will force the
    // next buffer insertion operation to transmit the data directly to the
    // device.)

    mov( true, Xmitting );

endif;

```

---

## 7.14 I/O and the Cache

It goes without saying that the CPU cannot cache values for memory-mapped I/O ports. If a port is an input port, caching the data from that port would always return the first value read; subsequent reads would read the value in the cache rather than the possible (volatile) data at the input port. Similarly, with a write-back cache mechanism, writes to an output port may never reach that port (i.e., the CPU may save up several writes in the cache and send the last such write to the actual I/O port). Therefore, there must be some mechanism to tell the CPU not to cache up accesses to certain memory locations.

The solution is in the virtual memory subsystem of the 80x86. The 80x86's page table entries contain information that the CPU can use to determine whether it is okay to map data from a page in memory to cache. If this flag is set one way, then the cache operates normally; if the flag is set the other way, then the CPU does not cache up accesses to that page.

Unfortunately, the granularity (that is, the minimum size) of this access is the 4K page. So if you need to map 16 device registers into memory somewhere and cannot cache them, you must actually consume 4K of the address space to hold these 16 locations. Fortunately, there is a lot of room in the 4 GByte virtual address space and there aren't that many peripheral devices that need to be mapped into the memory address space. So assigning these device addresses sparsely in the memory map will not present too many problems.

---

## 7.15 Win32 and Protected Mode Operation

Windows NT, 2000, and later versions of the operating system employ the 80x86's protected mode of operation. In this mode of operation, direct access to devices is restricted to the operating system and certain privileged programs. Standard applications, even those written in assembly language, are not so privileged. If you write a simple program that attempts to send data to an I/O port via an IN or an OUT instruction, the system will generate an illegal access exception and halt your program. Unless you're willing to write a device driver for Windows, you'll probably not be able to access the I/O devices directly.

Not all versions of Windows deny access to the peripherals. Windows 95 and 98, for example, don't prevent I/O access. So if you're using one of these operating systems, you can write assembly code that accesses the ports directly. However, the days of being able to access I/O devices directly from an application are clearly over. Future versions of Windows will restrict this activity.

This chapter has provided an introduction to I/O in a very general, architectural sense. It hasn't spent too much time discussing the particular peripheral devices present in a typical PC. This is an intended omission; there is no need to confuse readers with information they can't use. Furthermore, as manufacturers introduce new PCs they are removing many of the common peripherals like parallel and serial ports that are relatively easy to program in assembly language. They are replacing these devices with complex peripherals like USB and Firewire. Unfortunately, programming these newer peripheral devices is well beyond the scope of this text (Microsoft's USB code, for example, is well over 100 pages of C++ code).

Those who are interested in additional information about programming standard PC peripherals may want to consult one of the many excellent hardware references available for the PC or take a look at the DOS/16-bit version of this text.

IN and OUT aren't the only instructions that you cannot execute in an application running under protected mode. The system considers many instructions to be "privileged" and will abort your program if you attempt to use these instructions. The CLI and STI instructions are good examples. If you attempt to execute either of these instructions, the system will stop your program.

Some instructions will execute in an application, but behave differently than they do when the operating system executes them. The PUSHFD and POPFD instructions are good examples. These instructions push and pop the interrupt enable flag (among others). Therefore, you could use PUSHFD to push the flags on the stack, pop this double word off the stack and clear the bit associated with the interrupt flag, push the value back onto the stack and then use POPFD to restore the flags (and, in the process, clear the interrupt flag). This would seem like a sneaky way around clearing the interrupt flag. The CPU must allow applications to push and pop the flags for other reasons. However, for various security reasons the CPU cannot allow applications to manipulate the interrupt disable flag. Therefore, the POPFD instruction behaves a little differently in an application than it does when the operating system executes it. In an application, the CPU ignores the interrupt flag bit it pops off the stack. In operating system ("kernel") mode, popping the flags register does restore the interrupt flag.

---

## 7.16 Device Drivers

If Win32 doesn't allow direct access to peripheral devices, how does a program communicate with these devices? Clearly this can be done since applications interact with real-world devices all the time. If you

reread the previous section carefully, you'll note that it doesn't claim that programs can't access the devices, it only states that application programs are denied such access. Specially written modules, known as device drivers, are able to access I/O ports by special permission from the operating system. Writing device drivers is well beyond the scope of this chapter (though it will make an excellent subject for a later volume in this text). Nevertheless, an understanding of how device drivers work may help you understand the possibilities and limitations of I/O under a Win32 operating system.

A device driver is a special type of program that connects to the Windows operating system. The device driver must follow some special protocols and it must make some special calls to Windows that are not available to standard applications. Further, in order to install a device driver in your system you must have administrator privileges (device drivers create all kinds of security and resource allocation problems; you can't have every hacker in the world taking advantage of rogue device drivers running on your system). Therefore, "whipping out a device driver" is not a trivial process and application programs cannot load and unload arbitrary drivers at will.

Fortunately, there are only a limited number of devices you'd typically find on a PC, therefore you only need a limited number of device drivers. You would typically install a device driver in the operating system the same time you install the device (or when you install the operating system if the device is built into the PC). About the only time you'd really need to write your own device driver is when you build your own device or in some special instance when you need to take advantage of some devices capabilities that the standard device drivers don't allow for.

One big advantage to the device driver mechanism is that the operating system (or device vendors) must provide a reasonable set of device drivers or the system will never become popular (one of the reasons Microsoft and IBM's OS/2 operating system was never successful was the dearth of device drivers). This means that applications can easily manipulate lots of devices without the application programmer having to know much about the device itself; the real work has been taken care of by the operating system.

The device driver model does have a few drawbacks, however. The device driver model is great for low-speed devices, where the OS and device driver can respond to the device much more quickly than the device requires. The device driver model is also great for medium and high-speed devices where the system transmits large blocks of data in one direction at a time; in such a situation the application can pass a large block of data to the operating system and the OS can transmit this data to the device (or conversely, read a large block of data from the device and place it in an application-supplied buffer). One problem with the device driver model is that it does not support medium and high-speed data transfers that require a high degree of interaction between the device and the application.

The problem is that calling the operating system is an expensive process. Whenever an application makes a call to the OS to transmit data to the device it could actually take hundreds of microseconds, if not milliseconds, before the device driver actually sees the data. If the interaction between the device and the application requires a constant flurry of bytes moving back and forth, there will be a big delay if each transfer has to go through the operating system. For such applications you will need to write a special device driver to handle the transactions directly in the driver rather than continually returning to the application.

---

## 7.17 Putting It All Together

Although the CPU is where all the computation takes place in a computer system, that computation would be for naught if there was no way to get information into and out of the computer system. This is the responsibility of the I/O subsystem. I/O at the machine level is considerably different than the interface high level languages and I/O subroutine libraries (like *stdout.put*) provide. At the machine level, I/O transfers consist of moving bytes (or other data units) between the CPU and device registers or memory.

The 80x86 family supports two types of *programmed I/O*: memory-mapped input/output and I/O-mapped I/O. PCs also provide a third form of I/O that is mostly independent of the CPU: direct memory access or DMA. Memory-mapped input/output uses standard instructions that access memory to move data between the system and the peripheral devices. I/O-mapped input/output uses special instructions, IN and OUT, to move data between the CPU and peripheral devices. I/O-mapped devices have the advantage that they do not consume memory addresses normally intended for system memory. However, the only access to devices using this scheme is through the IN and OUT instructions; you cannot use arbitrary instructions that

manipulate memory to control such peripherals. Devices that use DMA have special hardware that let them transmit data to and from system memory without going through the CPU. Devices that use DMA tend to be very high performance, but this I/O mechanism is really only useful for devices that transmit large blocks of data at high speeds.

I/O devices have many different operating speeds. Some devices are far slower than the CPU while other devices can actually produce or consume data faster than the CPU. For devices that are slower than the CPU, some sort of handshaking mechanism is necessary in order to coordinate the data transfer between the CPU and the device. High-speed devices require a DMA controller or buffering since the CPU cannot handle the data rates of these devices. In all cases, some mechanism is necessary to tell the CPU that the I/O operation is complete so the CPU can go about other business.

In modern 32-bit operating systems like Win32, applications programs do not have direct access to the peripheral devices. The operating system coordinates all I/O via the use of device drivers. The good thing about device drivers is that you (usually) don't have to write them – the operating system provides them for you. The bad thing about writing device drivers is if you have to write one, they are very complex. A later volume in this text will discuss how to do this.

Because HLA programs usually run as applications under Win32, you will not be able to use most of the coding techniques this chapter discusses within your HLA applications. Nevertheless, understanding how device I/O works can help you write better applications. Of course, if you ever have to write a device driver for some device, then the basic knowledge this chapter presents is a good foundation for learning how to write such code.



# Questions, Projects, and Labs

## Chapter Eight

### 8.1 Questions

1. What three components make up Von Neumann Machines?
2. What is the purpose of
  - a) The system bus
  - b) The address bus
  - c) The data bus
  - d) The control bus
3. Which bus defines the “size” of the processor?
4. Which bus controls how much memory you can have?
5. Does the size of the data bus control the maximum value the CPU can process? Explain.
6. What are the data bus sizes of:
 

a) 8088	b) 8086	c) 80286	d) 80386sx
e) 80386	f) 80486	g) Pentium	h) Pentium II
7. What are the address bus sizes of the above processors?
8. How many “banks” of memory do each of the above processors have?
9. Explain how to store a word in byte addressable memory (that is, at what addresses). Explain how to store a double word.
10. How many memory operations will it take to read a word from the following addresses on the following processors?

**Table 32: Memory Cycles for Word Accesses**

	\$100	\$101	\$102	\$103	\$104	\$105
8088						
80286						
80386						

11. Repeat the above for double words

**Table 33: Memory Cycles for Doubleword Accesses**

	\$100	\$101	\$102	\$103	\$104	\$105
8088						

**Table 33: Memory Cycles for Doubleword Accesses**

	\$100	\$101	\$102	\$103	\$104	\$105
80286						
80386						

12. Explain which addresses are best for byte, word, and doubleword variables on an 8088, 80286, and 80386 processor.
13. Given the system bus size, what address boundary is best for a *real64* object in memory?
14. What is the purpose of the system clock?
15. What is a clock cycle?
16. What is the relationship between clock frequency and the clock period?
17. Explain why 10ns memory should not work on a 500 MHz Pentium III processor? Explain why it does.
18. What does the term “memory access time” mean?
19. What is a *wait state*?
20. If you are running an 80486 at the following clock speeds, how many wait states are required if you are using 80ns RAM (assuming no other delays)?
  - a) 20 MHz                      b) 25 MHz                      c) 33 MHz                      d) 50 MHz                      e) 100 MHz
21. If your CPU runs at 50 MHz, 20ns RAM probably won’t be fast enough to operate at zero wait states. Explain why.
22. Since sub-10ns RAM is available, why aren’t most systems zero wait state systems?
23. Explain how the cache operates to save some wait states.
24. What is the difference between spatial and temporal locality of reference?
25. Explain where temporal and spatial locality of reference occur in the following Pascal code:
 

```
while i < 10 do begin
    x := x * i;
    i := i + 1;
end;
```
26. How does cache memory improve the performance of a section of code exhibiting spatial locality of reference?
27. Under what circumstances is a cache not going to save you any wait states?
28. What is the effective (average) number of wait states the following systems will operate under?
  - a) 80% cache hit ratio, 10 wait states (WS) for memory, 0 WS for cache.
  - b) 90% cache hit ratio; 7 WS for memory; 0 WS for cache.
  - c) 95% cache hit ratio; 10 WS memory; 1 WS cache.
  - d) 50% cache hit ratio; 2 WS memory; 0 WS cache.
29. What is the purpose of a two level caching system? What does it save?
30. What is the effective number of wait states for the following systems?
  - a) 80% primary cache hit ratio (HR) zero WS; 95% secondary cache HR with 2 WS; 10 WS for main memory access.

- b) 50% primary cache HR, zero WS; 98% secondary cache HR, one WS; five WS for main memory access.
- c) 95% primary cache HR, one WS; 98% secondary cache HR, 4 WS; 10 WS for main memory access.

32. In what HLA declaration section would you declare initialized values that must not be changed during program execution?
33. In what HLA declaration section would you declare uninitialized variables?
34. In what HLA declaration section would you declare automatic variables?
35. Explain how you allocate and deallocate dynamic memory using the HLA Standard Library.
36. Provide two ways to take the address of a variable you declare in the STATIC section of your program.
37. What is the difference between the DATA and STATIC sections of your program?
38. Suppose you have a word variable, “w”, and you wish to load the L.O. byte of “w” into the AH register. What MOV instruction could you use to achieve this?
39. Suppose you have a word variable, “w”, and you wish to load the H.O. byte of “w” into the AL register. What MOV instruction could you use to achieve this?
40. What is the difference between “add( 1, [eax]);” and “add( 0, [eax+1]);”?
41. By default, the “stdout.put( eax );” statement will print EAX as an eight-digit hexadecimal value. Explain how to tell *stdout.put* to print EAX as an unsigned 32-bit integer; as a signed 32-bit integer. Provide the actual instructions to accomplish this.
42. Explain, step by step, what the “PUSH( EAX );” instruction does.
43. Explain, step by step, what the “POP( EAX );” instruction does.
44. What is the purpose of the PUSHW and PUSHHD instructions? What kind of data do they push? Why are there no POPW and POPD instructions?
45. What is the purpose of the “PUSHAD();” instruction? In what order does it push its data onto the stack?
46. Suppose you execute the following three instructions:

```
push( eax );
pop( bx );
pop( cx );
```

What value will be left in BX after this sequence? What value will be left in CX?

47. Suppose you needed to save the value of the carry flag across the execution of several instructions. Explain how you could do this (and provide the code).
48. Suppose you’ve executed the following two instructions to push EAX and EBX onto the stack:

```
push( eax );
push( ebx );
```

Without popping any data off the stack, explain how you can reload EAX’s value that was pushed on the stack. Provide a single instruction that will do this.

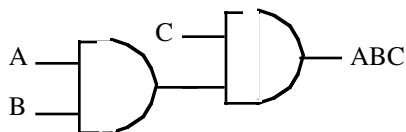
49. What is the difference between the “DEC( EAX );” instruction and the “SUB( 1, EAX );” instruction?
50. How can you check for unsigned arithmetic overflow immediately after an “INC( EAX );” instruction?
51. What is the identity element with respect to
  - a) AND            b) OR            c) XOR            d) NOT            e) NAND            f) NOR
52. Provide truth tables for the following functions of two input variables:
  - a) AND            b) OR            c) XOR            d) NAND            e) NOR
  - f) Equivalence   g)  $A < B$             h)  $A > B$             i)  $A$  implies  $B$
53. Provide the truth tables for the following functions of three input variables:

a)  $ABC$  (and) b)  $A+B+C$  (OR) c)  $(ABC)'$  (NAND) d)  $(A+B+C)'$  (NOR)

e) Equivalence  $(ABC) + (A'B'C')$  f) XOR  $(ABC + A'B'C')$

54. Provide schematics (electrical circuit diagrams) showing how to implement each of the functions in question three using only two-input gates and inverters. E.g.,

A)  $ABC =$



55. Provide implementations of an AND gate, OR gate, and inverter gate using one or more NOR gates.
56. What is the principle of duality? What does it do for us?
57. Build a single truth table that provides the outputs for the following three boolean functions of three variables:
- $$F_x = A + BC$$
- $$F_y = AB + C'B$$
- $$F_z = A'B'C' + ABC + C'B'A$$
58. Provide the function numbers for the three functions in question seven above.
59. How many possible (unique) boolean functions are there if the function has
- a) one input b) two inputs c) three inputs d) four inputs e) five inputs
60. Simplify the following boolean functions using algebraic transformations. Show your work.
- a)  $F = AB + AB'$  b)  $F = ABC + BC' + AC + ABC'$
- c)  $F = A'B'C'D' + A'B'C'D + A'B'CD + A'B'CD'$
- d)  $F = A'BC + ABC' + A'BC' + AB'C' + ABC + AB'C$
61. Simplify the boolean functions in question 60 using the mapping method.
62. Provide the logic equations in canonical form for the boolean functions  $S_0..S_6$  for the seven segment display (see “Combinatorial Circuits” on page 215).
63. Provide the truth tables for each of the functions in question 62
64. Minimize each of the functions in question 62 using the map method.
65. The logic equation for a half-adder (in canonical form) is
- $$\text{Sum} = AB' + A'B \quad \text{Carry} = AB$$
- a) Provide an electronic circuit diagram for a half-adder using AND, OR, and Inverter gates
- b) Provide the circuit using only NAND gates
66. The canonical equations for a full adder take the form:
- $$\text{Sum} = A'B'C + A'BC' + AB'C' + ABC$$
- $$\text{Carry} = ABC + ABC' + AB'C + A'BC$$
- a) Provide the schematic for these circuits using AND, OR, and inverter gates.
- b) Optimize these equations using the map method.
- c) Provide the electronic circuit for the optimized version (using AND, OR, and inverter gates).
67. Assume you have a D flip-flop (use this definition in this text) whose outputs currently are  $Q=1$  and  $Q'=0$ . Describe, in minute detail, exactly what happens when the clock line goes
- a) from low to high with  $D=0$

- b) from high to low with  $D=0$
68. Rewrite the following Pascal statements to make them more efficient:
    - a) if (x or (not x and y)) then write('1');
    - b) while(not x and not y) do somefunc(x,y);
    - c) if not((x <> y) and (a = b)) then Something;
  69. Provide canonical forms (sum of minterms) for each of the following:
    - a)  $F(A,B,C) = A'BC + AB + BCb$   $F(A,B,C,D) = A + B + CD' + D$
    - c)  $F(A,B,C) = A'B + B'Ad$   $F(A,B,C,D) = A + BD'$
    - e)  $F(A,B,C,D) = A'B'C'D + AB'C'D' + CD + A'BCD'$
  70. Convert the sum of minterms forms in question 69 to the product of maxterms forms.
  71. What is the difference between the Harvard and the Von Neumann Architectures?
  72. Explain how encoding instructions in binary saves space in an opcode (see “Basic CPU Design” on page 235).
  73. What is the difference between Random Logic and Microcode?
  74. What do the following acronyms stand for? CISC, RISC, VLIW.
  75. Is the LOOP instruction a “RISC Core” or a “Complex” instruction? Explain.
  76. What is the difference between an 80x86 “RISC Core” and a “Complex” instruction?
  77. What sequence of instructions is the LOOP instruction equivalent to?
  78. Explain, step-by-step, how a MOV instruction might work.
  79. Explain how CPU designers use parallelism to increase the CPU’s throughput (# of instrs/second).
  80. What is a prefetch queue?
  81. What is pipelining?
  82. What is a pipeline stall?
  83. How can creating separate instruction and data caches improve performance?
  84. Why do separate instruction and data caches often operate at a greater miss rate than a unified cache?
  85. What is a data hazard?
  86. What is a superscalar CPU?
  87. Explain how “out of order” execution works. Under what circumstances can “out of order” execution improve performance?
  88. What is “register renaming?” How can it improve performance?
  89. Explain the following terms: SISD, SIMD, MIMD.
  90. Are MMX instructions SISD, SIMD, or MIMD?
  91. Provide four reasons why we can’t/shouldn’t design a CPU with as many instructions as possible.
  92. Explain why, when decoding an instruction, it is better to use several smaller decoders rather than one big decoder.
  93. Explain how to use an opcode prefix byte to extend an instruction set.
  94. What is the value of the opcode prefix byte on the 80x86 CPU?
  95. What is the maximum length of an 80x86 instruction?
  96. What is the purpose of the MOD-REG-R/M byte on the 80x86?
  97. What is the purpose of the SIB byte on the 80x86?
  98. How long (in bytes) is the 80x86 opcode?

99. When encoding an instruction with a memory addressing mode that has a displacement, where does the 80x86 expect the displacement value to appear?
100. On the 80x86, how do you encode immediate constant within the instruction?
101. What are the displacement sizes that the 80x86 supports for memory operands (under Win32)? Hint: this one is a little tricky.
102. How do you select the displacement size in the instruction encoding?
103. Based on the values in the MOD-REG-R/M byte, explain why the 80x86 instructions don't allow memory to memory operations.
104. Explain why there is no "[EBP]" addressing mode on the 80x86.
105. The "[ESP]" address mode requires an SIB byte even though this instruction doesn't have an index register associated with it. Explain.
106. 80x86 opcodes only have one bit to specify the operand size. Explain how the 80x86 encodes three different operand sizes.
107. Under Win32, what are the two default operand sizes?
108. What is an *alternate instruction encoding*?
109. What is the *memory hierarchy*?
110. What is the difference between a direct-mapped cache and a fully associative cache?
111. What is the difference between a two-way set associative cache and a four-way set associative cache?
112. Why would a direct-mapped cache offer better performance than a fully associative cache? Why would a fully associative cache offer better performance than a direct-mapped cache? Give the circumstances for both.
113. What is the difference between the write-through and the write-back cache policies? Which is higher performance? Which is best in a multiprocessor system?
114. What is paging?
115. What is virtual memory?
116. What is thrashing?
117. What does NUMA stand for? How can it affect the performance of your programs?
118. What is the difference between the read-only, write-only, read/write, and dual I/O ports?
119. If the CPU sees a port as read/write, how does the external world see that port (input, output, both)?
120. What is the difference between I/O-mapped input/output and memory-mapped input/output?
121. What is DMA? Explain, don't just spell out the acronym.
122. What is polling?
123. What are the relative speed differences between low, medium, and high speed I/O devices?
124. How are busses like PCI and ISA different than the CPU's address and data busses?
125. Which bus is higher performance, PCI or ISA? Give performance numbers to back up your claim.
126. What is the AGP bus and what advantage does it have over the PCI bus?
127. What is I/O buffering? How can it keep a system from losing data?
128. What is the purpose of handshaking?
129. What is a time-out? What are some convenient instructions for checking for a time-out while waiting on a bit in an I/O port?
130. What is the difference between interrupt driven I/O and polled I/O?

## 8.2 Programming Projects

1. Write a program that requests a number from the user and then allows the user to enter the specified number of characters. Display the characters in the reverse order the user enters them. If the user enters fewer than the specified number of characters, fill the remainder of your buffer with spaces. Repeat until the user enters an empty line of text. (Idea: Use the HLA Standard Library `stdin.eoln()` function, that returns true (1) or false (0) depending on whether the character just read by `stdin.getc()` was the last character on the line). Be sure to free up any dynamically allocated storage you use after displaying the user output.
2. Write a program that requests a count from the user and allocates storage for that many 32-bit integers (don't forget to multiply the count by four since `int32` objects are four bytes long; use `SHL` to multiply by four). The program should then read the specified number of integers from the user and store them away in the allocated buffer. Finally, the program should sum up each value in the buffer, display the sum, and then free the allocated storage and quit.
3. Modify program (2) above so that it reports the largest (maximum) and smallest (minimum) integer values in the buffer.
4. Write a program that allocates a buffer containing 32 bytes of storage, reads an integer value from the user, and then copies each bit from the integer into successive bytes of memory; that is, bit zero winds up in the first byte, bit 1 winds up in the second byte, etc. When your program is done copying the bits, the 32 bytes should all contain a zero or a one depending upon the setting of the corresponding bit in the input integer. Write each of these bits to the display. Sum the 32 bits up and print the sum (which tells you how many one-bits were in the number). Be sure to free the storage before your program quits.
5. Write a program that allocates storage for two buffers. A single user input should determine the size of these two buffers: the first buffer will contain the number of bytes specified by the user and the second buffer will contain four times this many bytes. Read a sequence of `int8` values from the user and store them into the first buffer. Then copy the values from first buffer to consecutive dwords in the second buffer, using sign extension to convert the values. Display both buffers using hexadecimal notation once the conversion is complete.
6. Using HLA Standard Library `console.fillRect` procedure, write a program that generates a checkerboard pattern on the screen. The user should be able to specify the number of squares on each side of the checkerboard as well as the two colors to use for the checkerboard display. Your program should verify that the checkerboard will fit on the display before drawing it and report an error if it does not fit.
7. Write a two-player tic-tac-toe game. Use the HLA Standard Library `console` module to clear the screen and draw the board between each move. Let the users alternate moves until one of them decides to quit (don't bother trying to have your program determine if the game is over, just use a special input value to end the game). Be sure to draw a second game board listing numbers or characters by which each player can choose the square into which they want to move. Use different colors for each player's symbols. If you want to get real fancy, you can draw the X's and O's by printing blocks of spaces on the screen. If you really want to be impressive, read the HLA console documentation and learn how to use the mouse. Then use mouse clicks to make the moves (this is optional!).
8. Write a program that inputs an (x,y) coordinate from the user and a single unsigned integer value. Your program should draw a rectangle on the screen using the PC's line drawing graphic characters (see Appendix B). The upper left hand corner of the rectangle should be at the coordinate specified by the user. The integer value specifies the width and height of the rectangle on the screen. Before drawing the rectangle, verify that it will fit on the screen and print an error message if it won't. Use the `console.gotoxy` procedure to position the cursor before drawing each character; do not disturb any other characters on the screen except those character positions where you actually draw one of the line drawing characters.
9. The HLA "`console.getRect( top, left, bottom, right, buffer )`" function copies the data in the specified rectangle on the screen to the buffer passed as the last parameter. E.g., "`console.getRect( 10, 10, 20, 20, (type byte [eax]))`;" copies the 10x10 matrix of characters into the block of memory pointed at by `EAX`. Similarly, the "`console.putRect( top, left, bottom, right, buffer)`;" call will copy the data from the specified buffer to the screen at the specified coordinates. The buffer must contain twice the number of bytes as there are characters in the rectangular region. Allocate a sufficient amount of storage using `malloc` and

then use these two routines to temporarily save a portion of the screen while you write something else to it. Then, upon prompt from the program's user, restore the original rectangle.

- 10) Write a program that reads four values from the user, I, J, K, and L, and plugs these values into a truth table with  $B'A' = I$ ,  $B'A = J$ ,  $BA' = K$ , and  $BA = L$ . Ensure that these input values are only zero or one. Then input a series of pairs of zeros or ones from the user and plug them into the truth table. Display the result for each computation.
- 11) Write a program that, given a 4-bit logic function number, displays the truth table for that function of two variables.
- 12) Write a program that, given an 8-bit logic function number, displays the truth table for that function of three variables.
- 13) Write a program that, given a 16-bit logic function number, displays the truth table for that function of four variables.
- 14) Write a program that, given a 16-bit logic function number, displays the canonical equation for that function (hint: build the truth table).

## 8.3 Chapters One and Two Laboratory Exercises

Accompanying this text is a significant amount of software. This software can be found in the AOA\_Software directory. Inside this directory are a set of subdirectories with names like *Volume2* and *Volume3*. Inside those directories are files with names like *Ch02* and *Ch03* with the names obviously corresponding to chapters in this text. The code for these laboratory exercises appears in the Volume2\Ch08 sub-directory. Please see this directory for more details.

### 8.3.1 Memory Organization Exercises

The following program (Program 8.1) demonstrates the memory layout of an HLA program. It does this by declaring variables or other symbols in each of the various run-time memory segments. This program uses the LEA instruction to take the address of each memory object and then displays the address using hexadecimal notation. Run this program and compare the addresses of each of the objects.

For your lab report, compare the output addresses against the memory layout presented in this chapter. Since each memory segment begins on a 4096 byte (\$1000) boundary, what clues do the output addresses give us concerning the grouping of variables in the memory segments? Given the output of this program, and the address of some other variable declared in this program, describe how you could determine which segment that other variable is in given no other information than the variable's location.

---

---

```
// Sample program for Memory Organization Laboratory Exercise
// in "The Art of Assembly Language Programming"
// (HLA Edition).

program MemOrg;
#include( "stdlib.hhf" );

// Declare a set of variables in each of the
// different memory sections so we can compare
// their addresses in memory.

var
    AutoVar:    int32;

static
    StaticVar:  int32;

data
    DataVar:    int32;
               dword 0;

readonly
    ROVar:      int32 := 0;

storage
    StorageVar: int32;

begin MemOrg;

    // Take the address of each of the variables (plus allocate
    // storage for a dynamic variable) and display the addresses.

    lea( eax, AutoVar );
    stdout.put( "AutoVar address in memory is: $", eax, nl );
```

```

lea( eax, StaticVar );
stdout.put( "StaticVar address in memory is: $", eax, nl );

lea( eax, DataVar );
stdout.put( "DataVar address in memory is: $", eax, nl );

lea( eax, ROVar );
stdout.put( "ROVar address in memory is: $", eax, nl );

lea( eax, StorageVar );
stdout.put( "StorageVar address in memory is: $", eax, nl );

// Dynamically allocate a variable on the heap.

malloc( 4 );
stdout.put( "Dynamic variable address in memory is: $", eax, nl );
free( eax );

// The following code computes the address of this instruction
// in memory:

CodeAdrs: lea( eax, CodeAdrs );
stdout.put( "CodeAdrs label's address in memory is: $", eax, nl );

// Just to put things into perspective, display the value of the
// stack pointer as well:

stdout.put( "Stack Pointer contains the address: $", esp, nl );

end MemOrg;

```

---

### Program 8.1 Demonstration of Memory Sections

---

## 8.3.2 Data Alignment Exercises

The following program lets you test the effect of data alignment on your program. It allocates a block of 1,000,000 dword values and then runs through two loops that access these values on an address that is an even multiple of four and at an address that is not an even multiple of four. By measuring the time these two code fragments take to execute, you can compare the difference between aligned and non-aligned memory access.

Note: keep in mind that Windows is a multi-tasking operating system. Therefore, the difference in execution time will not be as great as you might expect because a large percentage of the execution time you measure with this program will be spent in another process. You can assume that approximately the same amount of time is spent outside this process in both halves of the program.

To avoid optimizations by the cache hardware, this program accesses data at over four million different addresses. The fact that most memory accesses will experience a cache miss will also skew the timings (and make them more similar). Finally, different processors (e.g., Pentium vs. Pentium II vs. K6 vs. Athlon) have different penalties for misaligned data. Hence, you should expect to see only a small percentage difference between the two halves of the following program.

This program also demonstrates the use of the align directive in the code segment. If used immediately before a loop that executes a large number of times (as it is in this code), the align directive can improve the performance of the loop slightly.

Run this program and time the two halves using a watch. Record the difference in your lab report and provide an explanation for the different running times.

---

```
// Sample program for Data Alignment Laboratory Exercise
// in "The Art of Assembly Language Programming"
// (HLA Edition).

program DataAlign;
#include( "stdlib.hhf" );

begin DataAlign;

    console.cls();
    stdout.put
    (
        "Memory Alignment Exercise",nl,
        nl,
        "Using a watch (preferably a stopwatch), time the execution of", nl
        "the following code to determine how many seconds it takes to", nl
        "execute.", nl
        nl
        "Press Enter to begin timing the code:"
    );

    // Allocate enough dynamic memory to ensure that it does not
    // all fit inside the cache. Note: the machine had better have
    // at least four megabytes free or virtual memory will kick in
    // and invalidate the timing.

    malloc( 4_000_000 );

    // Zero out the memory (this loop really exists just to
    // ensure that all memory is mapped in by the OS).

    mov( 1_000_000, ecx );
    repeat

        dec( ecx );
        mov( 0, (type dword [eax+ecx*4]));

    until( !ecx ); // Repeat until ECX = 0.

    // Okay, wait for the user to press the Enter key.

    stdin.ReadLn();

    // Note: as processors get faster and faster, you may
    // want to increase the size of the following constant.
    // Execution time for this loop should be approximately
    // 10-30 seconds.

    mov( 1000, edx );
    inc( eax ); // Force misalignment of data.

    repeat

        mov( 999_999, ecx );
        align( 16 );
```

```

repeat

    dec( ecx );
    mov( [eax+ecx*4], ebx );

until( !ecx );
dec( edx );

until( !edx ); // Repeat until EAX is zero.

stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );

// Okay, time the aligned access.

stdout.put
(
    "Press Enter again to begin timing access to aligned variable:"
);
stdin.ReadLn();

// Note: if you change the constant above, be sure to change
// this one, too!

mov( 1000, edx );
dec( eax );    // Align the data.
repeat

    mov( 999_999, ecx );
    align( 16 );
    repeat

        dec( ecx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );
    dec( edx );

until( !edx ); // Repeat until EAX is zero.

stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );
free( eax );

end DataAlign;

```

---

### Program 8.2 Data Alignment Exercise

---

## 8.3.3 Readonly Segment Exercises

The following exercise demonstrates that you cannot write data to a readonly variable. The variable *i*, appearing in the READONLY section, generates an *ex.AccessViolation* exception if you attempt to write to it. Run this program and document what happens in your lab report. Explain how you might use READONLY variables in your program to protect against errors in your code.

---

```
// Sample program for Read Only Data Laboratory Exercise 3.17.3
```

```
// in "The Art of Assembly Language Programming"
// (HLA Edition).

program ReadOnlyDemo;
#include( "stdlib.hhf" );

readonly
    i:int32 := 10;

begin ReadOnlyDemo;

    stdout.put( "i = ", i, nl );
    try

        mov( 0, i );
        stdout.put( "Now i contains ", i, nl );

        exception( ex.AccessViolation )

            stdout.put( "Error attempting to write to variable 'i'", nl );

    endtry;

end ReadOnlyDemo;
```

---



---

### Program 8.3 READONLY Variable Demonstration

---



---

## 8.3.4 Type Coercion Exercises

The HLA type coercion operators are actually useful for many things besides letting you load a portion of a variable into a smaller register (the primary example this chapter has given thus far). Many HLA instructions and Standard Library routines use the type information associated with a variable to determine how to use that variable. For example, the *stdout.put* routine in the Standard Library determines how to display a value based on its type. For example, if you have an *int32* variable, *i32*, and you decide to print it with *stdout.put*, it will use the type information to determine that it must print this value as a signed 32-bit integer. This laboratory exercise demonstrates that you can use the type coercion operators to display a value as a different type.

Compile and run the following program that demonstrates how to display an *int32* variable (*testVar*) using hexadecimal notation:

---



---

```
// Sample program for Coercion Laboratory Exercise
// in "The Art of Assembly Language Programming"
// (HLA Edition).

program Coercion;
#include( "stdlib.hhf" );

static
    testVar:int32;

begin Coercion;

    stdout.put( "Enter a 32-bit signed integer value: " );
    stdin.get( testVar );
```

```

        stdout.put
        (
            nl,
            "In decimal: ", testVar, nl
            "In hexadecimal: $", (type dword testVar), nl
        );
    end Coercion;

```

---

### Program 8.4    Type Coercion Exercise

---

After running this program and noting the results, modify the type of *testVar* so that it is a *real32* variable. Rerun the program. Based on what you've learned from Chapter Two, explain the output when you enter 1.0 as the real value. Note that you can also use type coercion in the *stdin.get* routine. Type cast the *testVar* parameter in *stdin.get* so that it is a *real32* object. Rerun the program and explain the results in your lab report.

---

## 8.3.5 Dynamic Memory Allocation Exercises

The following short program demonstrates how to use the *malloc* procedure to allocate storage for 10 integer values. Note that this program doesn't declare any variables at all. All memory accesses occur in the dynamically allocated block of memory. Compile and run this program. Describe the results in your lab report.

---

```

// Sample program for Dynamic Allocation Laboratory Exercise
// in "The Art of Assembly Language Programming"
// (HLA Edition).

program DynMem;
#include( "stdlib.hhf" );

begin DynMem;

    stdout.put( "Allocating storage for 10 integers", nl );

    // Note: we must allocate storage for 40 bytes since
    // each integer consumes four bytes. This function
    // returns a pointer to the data in EAX.

    malloc( 40 );

    // Move pointer to storage to the ESI register so we
    // can use EAX for other purposes.

    mov( eax, esi );

    // Prompt the user to enter a value:

    stdout.put( "Enter a integer to initialize the data with: " );
    stdin.geti32();

    // Use the input value as a starting value with which to
    // initialize the storage we've just created:

```

```

stdout.newln();
for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do

    mov( eax, [esi+ebx*4] );
    inc( eax );

endfor;

// Okay, display these values from the allocated storage:

for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do

    mov( [esi+ebx*4], eax );
    stdout.put
    (
        "Value[",
        (type uns32 ebx),
        "] = ",
        (type int32 eax),
        nl
    );

endfor;

// Free up the storage allocated above.

stdout.put( nl, "Freeing storage:", nl );
free( esi );

end DynMem;

```

---

### Program 8.5    MALLOC and FREE Exercise

---

Modify this program to read and display 20 characters rather than 10 integers. Describe the necessary changes in your lab report and include a copy of the new program with your lab report.

---

## 8.4    Chapter Three Laboratory Exercises

This laboratory uses several Windows programs to manipulate truth tables and logic expressions, optimize logic equations, and simulate logic equations. These programs will help you understand the relationship between logic equations and truth tables as well as gain a fuller understanding of logic systems.

The *WLOGIC.EXE* program simulates logic circuitry. WLOGIC stores several logic equations that describe an electronic circuit and then it simulates that circuit using “switches” as inputs and “LEDs” as outputs.

---

### 8.4.1    Truth Tables and Logic Equations Exercises

In this laboratory exercise you will create several different truth tables of two, three, and four variables. The TRUTHBL.EXE program (found in the Volume2\Ch08 subdirectory) will automatically convert the truth tables you input into logic equations in the sum of minterms canonical form.

The TRUTHBL.EXE program provides three buttons that let you choose a two variable, three variable, or four variable truth table. Pressing one of these buttons rearranges the truth table in an appropriate fashion. By default, the TRUTHBL program assumes you want to work with a four variable truth table. Try pressing the *Two Variables*, *Three Variables*, and *Four Variables* buttons and observe the results. Describe what happens in your lab report.

To change the truth table entries, all you need do is click on the square associated with the truth table value you want to change. Clicking on one of these boxes toggles (inverts) that value in that square. For example, try clicking on the DCBA square several times and observe the results.

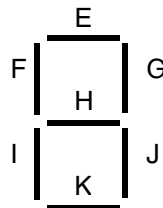
Note that as you click on different truth table entries, the TRUTHBL program automatically recomputes the sum of minterms canonical logic equation and displays it at the bottom of the window. What equation does the program display if you set all squares in the truth table to zero?<sup>1</sup>

Set up the TRUTHBL program to work with four variables. Set the DCBA square to one. Now press the *Two Variables* button. Press the *Four Variables* button and set *all* the squares to one. Now press the *Two Variables* button again. Finally, press the *Four Variables* button and examine the results. What does the TRUTHBL program do when you switch between different sized truth tables? Feel free to try additional experiments to verify your hypothesis. Describe your results in your lab report.

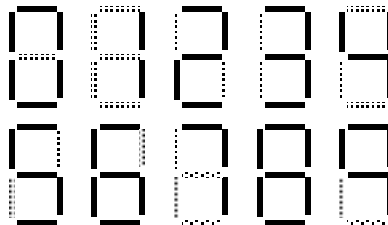
Switch to two variable mode. Input the truth tables for the logical AND, OR, XOR, and NAND truth tables. Verify the correctness of the resulting logic equations. Write up the results in your lab report. Note: if there is a Windows-compatible printer attached to your computer, you can print each truth table you create by pressing the Print button in the window. This makes it very easy to include the truth table and corresponding logic equation in your lab report. **For additional credit:** input the truth tables for all 16 functions of two variables. In your lab report, present the results for these 16 functions.

Design several two, three, and four variable truth tables by hand. Manually determine their logic equations in sum of minterms canonical form. Input the truth tables and verify the correctness of your logic equations. Include your hand designed truth tables and logic equations as well as the machine produced versions in your lab report.

Consider the following layout for a seven-segment display:



Here are the segments to light for the binary values DCBA = 0000..1001:



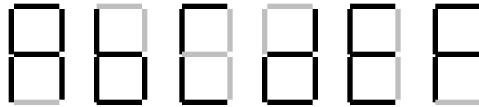
$$\begin{aligned}
 E &= D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A \\
 F &= D'C'B'A' + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A \\
 G &= D'C'B'A' + D'C'B'A + D'C'BA' + D'C'BA + D'CB'A' + D'CBA + DC'B'A' + DC'B'A \\
 H &= D'C'BA' + D'C'BA + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
 \end{aligned}$$

1. Note: On initial entry to the program, TRUTHBL does not display a logic equation. Therefore, you will need to set at least one square to one and then back to zero to see this equation.

$$\begin{aligned} I &= D' C' B' A' + D' C' B A' + D' C B A' + D C' B' A' \\ J &= D' C' B' A' + D' C' B' A + D' C' B A + D' C B' A' + D' C B' A + D' C B A' + D' C B A + D C' B' A' + D C' B' A' \\ K &= D' C' B' A' + D' C' B A' + D' C' B A + D' C B' A + D' C B A' + D C' B' A' \end{aligned}$$

Convert each of these logic equations to a truth table by setting all entries in the table to zero and then clicking on each square corresponding to each minterm in the equation. Verify by observing the equation that TRUTH TBL produces that you've successfully converted each equation to a truth table. Describe the results and provide the truth tables in your lab report.

**For Additional Credit:** Modify the equations above to include the following hexadecimal characters. Determine the new truth tables and use the TRUTH TBL program to verify that your truth tables and logic equations are correct.



## 8.4.2 Canonical Logic Equations Exercises

In this laboratory you will enter several different logic equations and compute their canonical forms as well as generate their truth table. In a sense, this exercise is the opposite of the previous exercise where you generated a canonical logic equation from a truth table.

This exercise uses the CANON.EXE program found in the Volume2\Ch08 subdirectory. Run this program from Windows by double clicking on its icon. This program displays a text box, a truth table, and several buttons. Unlike the TRUTH TBL.EXE program from the previous exercise, you cannot modify the truth table in the CANON.EXE program; it is a display-only table. In this program you will enter logic equations in the text entry box and then press the “Compute” button to see the resulting truth table. This program also produces the sum of minterms canonical form for the logic equation you enter (hence this program’s name).

Valid logic equations take the following form:

- A *term* is either a variable (A, B, C, or D) or a logic expression surrounded by parentheses.
- A *factor* is either a term, or a factor followed by the prime symbol (an apostrophe, i.e., “’”). The prime symbol logically negates the factor immediately preceding it.
- A *product* is either a factor, or a factor concatenated with a product. The concatenation denotes logical AND operation.
- An expression is either a product or a product followed by a “+” (denoting logical OR) and followed by another expression.

Note that logical OR has the lowest precedence, logical AND has an intermediate precedence, and logical NOT has the highest precedence of these three operators. You can use parentheses to override operator precedence. The logical NOT operator, since its precedence is so high, applies only to a variable or a parenthesized expression. The following are all examples of legal expressions:

$$\begin{aligned} &AB'C + D(B' + C') \\ &AB(C+D)' + A'B'(C+D) \\ &A'B'C'D' + ABCD + A(B+C) \\ &(A+B)' + A'B' \end{aligned}$$

For this set of exercises, you should create several logic expression and feed them through CANON.EXE. Include the truth tables and canonical logic forms in your lab report. Also verify that the theorems appearing in this chapter (See “Boolean Algebra” on page 195.) are valid by entering each side of the theorem and verifying that they both produce the same truth table (e.g.,  $(AB)' = A' + B'$ ). For additional credit, create several complex logic equations and generate their truth tables and canonical forms by hand. Then input them into the CANON.EXE program to verify your work.

### 8.4.3 Optimization Exercises

In this set of laboratory exercises, the OPTIMZP.EXE program (found in the Volume2\Ch08 subdirectory) will guide you through the steps of logic function optimization. The OPTIMZP.EXE program uses the Karnaugh Map technique to produce an equation with the minimal number of terms.

Run the OPTIMZP.EXE program by clicking on its icon or running the OPTIMZP.EXE program using the program manager's File>Run menu option. This program lets you enter an arbitrary logic equation using the same syntax as the CANON.EXE program in the previous exercise.

After entering an equation press the "Optimize" button in the OPTIMZP.EXE window. This will construct the truth table, canonical equation, and an optimized form of the logic equation you enter. Once you have optimized the equation, OPTIMZP.EXE enables the "Step" button. Pressing this button walks you through the optimization process step-by-step.

For this exercise you should enter the seven equations for the seven-segment display. Generate and record the optimize versions of these equations for your lab report and the next set of exercises. Single step through each of the equations to make sure you understand how OPTIMZP.EXE produces the optimal expressions.

**For additional credit:** OPTIMZP.EXE generates a single optimal expression for any given logic function. Other optimal functions may exist. Using the Karnaugh mapping technique, see if you can determine if other, equivalent, optimal expressions exist. Feed the optimal equations OPTIMZP.EXE produces and your optimal expressions into the CANON.EXE program to verify that their canonical forms are identical (and, hence, the functions are equivalent).

### 8.4.4 Logic Evaluation Exercises

In this set of laboratory exercises you will use the LOGIC.EXE program to enter, edit, initialize, and evaluation logic expressions. This program lets you enter up to 22 distinct logic equations involving as many as 26 variables plus a clock value. LOGIC.EXE provides four input variables and 11 output variables (four simulated LEDs and a simulated seven-segment display). Note: this program requires that you install two files in your WINDOWS\SYSTEM directory. Please see the README.TXT file in the Volume2\Ch08 subdirectory for more details.

Execute the LOGIC.EXE program by double-clicking on its icon or using the program manager's "File>Run" menu option. This program consists of three main parts: an equation editor, an initialization screen, and an execution module. LOGIC.EXE uses a set of *tabbed notebook screens* to switch between these three modules. By clicking on the "Create", *Initialize*, and *Execute* tabs at the top of the screen with your mouse, you can select the specific module you want to use. Typically, you would first create a set of equations on the *Create* page and then execute those functions on the *Execute* page. Optionally, you can initialize any necessary logic variables (D-Z) on the *Initialize* page. At any time you can easily switch between modules by pressing on the appropriate notebook tab. For example, you could create a set of equations, execute them, and then go back and modify the equations (e.g., to correct any mistakes) by pressing on the *Create* tab.

The Create page lets you add, edit, and delete logic equations. Logic equations may use the variables A-Z plus the "#" symbol ("#" denotes the clock). The equations use a syntax that is very similar to the logic expressions you've used in previous exercises in this chapter. In fact, there are only two major differences between the functions LOGIC.EXE allows and the functions that the other programs allow. First, LOGIC.EXE lets you use the variables A-Z and "#" (the other programs only let you enter functions of four variables using A-D). The second difference is that LOGIC.EXE functions must take the form:

$$\text{variable} = \text{expression}$$

where *variable* is a single alphabetic character E-Z<sup>2</sup> and *expression* is a logic expression using the variables A-Z and #. An expression may use a maximum of four different variables (A-Z) plus the clock value (#).

---

2. A-D are read-only values that you read from a set of switches. Therefore, you cannot store a value into these variables.

© 2000, By Randall Hyde

Page 359

The *Create* page contains three important buttons: *Add*, *Edit*, and *Delete*. When you press the *Add* button LOGIC.EXE opens a dialog box that lets you enter an equation. Type your equation (or edit the default equation) and press the *Okay* button. If there is a problem with the equation you enter, LOGIC.EXE will report the error and make you fix the problem, otherwise, LOGIC.EXE will attempt to add this equation to the system you are building. If a function already exists that has the same destination variable as the equation you've just added, LOGIC.EXE will ask you if you really want to replace that function before proceeding with the replacement. Once LOGIC.EXE adds your equation to its list, it also displays the truth table for that

equation. You can add up to 22 equations to the system (since there are 22 possible destination variables, E-Z). LOGIC.EXE displays those functions in the list box on the right hand side of the window.

Once you've entered two or more logic functions, you can view the truth table for a given logic function by simply clicking on that function with the mouse in the function list box.

If you make a mistake in a logic function you can delete that function by selecting with the mouse and pressing the *delete* button, or you can edit it by selecting it with the mouse and pressing the *edit* button. You can also edit a function by double-clicking on the function in the expression list.

The *Initialize* page displays boxes for each of the 26 possible variables. It lets you view the current values for these 26 variables and change the values of the E-Z variables (remember, A-D are read-only). As a general rule, you will not need to initialize any of the variables, so you can skip this page if you don't need to initialize any variables.

The *Execute* page contains five buttons of importance: *A-D* and *Pulse*. The *A-D* toggle switches let you set the input values for the A-D variables. The *Pulse* switch toggles the clock value from zero to one and then back to zero, evaluating the system of logic functions while the clock is in each state.

In addition to the input buttons, there are several outputs on the *Execute* page. First, of course, are the four LEDs (W, X, Y, and Z) as well as the seven-segment display (output variables E-K as noted above). In addition to the LEDs, there is an *Instability* annunciator that turns red if LOGIC.EXE detects an instability in the system. There is also a small panel that displays the current values of all the system variables at the bottom of the window.

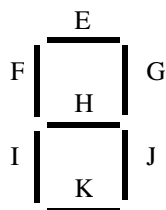
To execute the system of equations simply change one of the input values (A-D) or press the *Pulse* button. LOGIC.EXE will automatically reevaluate the system of equations whenever A-D or # changes.

To familiarize yourself with the LOGIC.EXE program, enter the following equations into the equation editor:

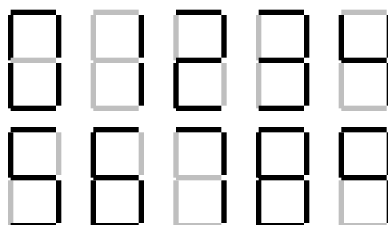
$W = AB$	A and B
$X = A + B$	A or B
$Y = A'B + AB'$	A xor B
$Z = A'$	not A

After entering these equations, go to the execute page and enter the four values 00, 01, 10, and 11 for BA. Note the values for W, X, Y, and Z for your lab report.

The LOGIC.EXE program simulates a seven segment display. Variables E-K light the individual segments as follows:



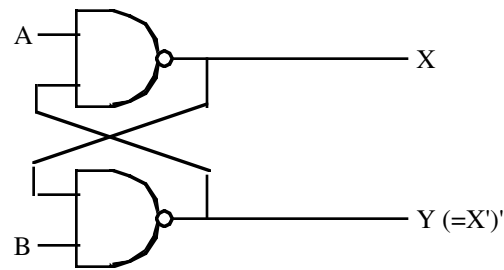
Here are the segments to light for the binary values DCBA = 0000 - 1001:



Enter the seven equations for these segments into LOGIC.EXE and try out each of the patterns (0000 through 1111). **Hint:** use the optimized equations you developed earlier. **Optional, for additional credit:**

enter the equations for the 16 hexadecimal values and cycle through those 16 values. Include the results in your lab manual.

A simple sequential circuit. For this exercise you will enter the logic equations for a simple set / reset flip-flop. The circuit diagram is



A Set/Reset Flip-Flop

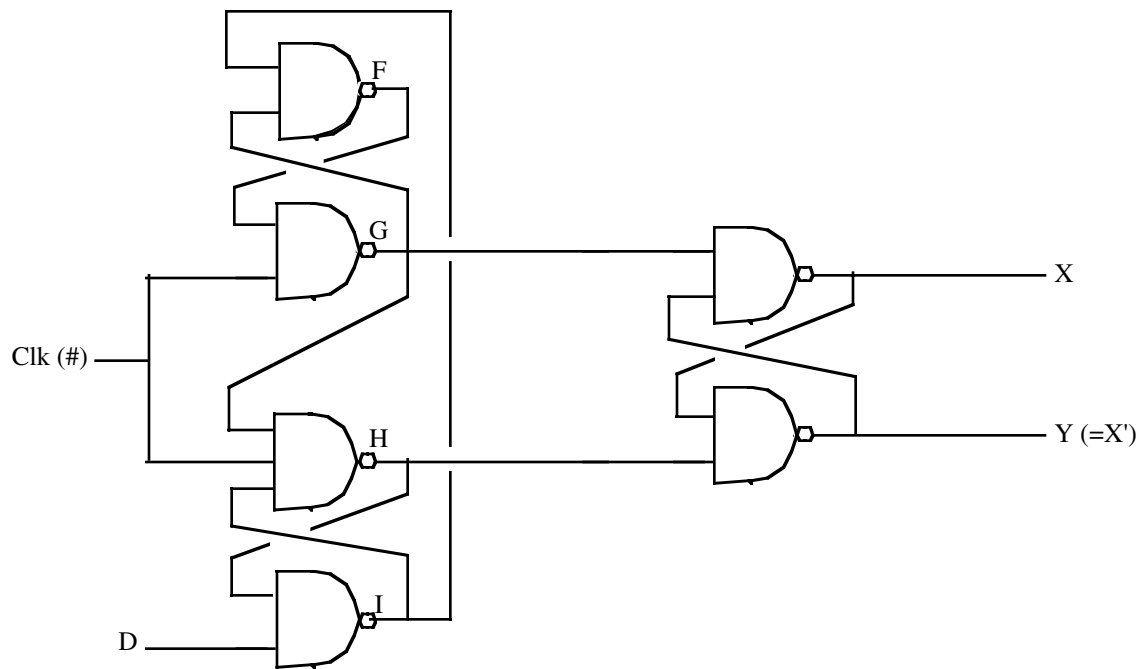
Since there are two outputs, this circuit has two corresponding logic equations. They are

$$\begin{aligned} X &= (AY)' \\ Y &= (BX)' \end{aligned}$$

These two equations form a *sequential circuit* since they both use variables that are function outputs. In particular, Y uses the previous value for X and X uses the previous value for Y when computing new values for X and Y.

Enter these two equations into LOGIC.EXE. Set the A and B inputs to one (the normal or *quiescent* state) and run the logic simulation. Try setting the A switch to zero and determine what happens. Press the *Pulse* button several times with A still at zero to see what happens. Then switch A back to one and repeat this process. Now try this experiment again, this time setting B to zero. Finally, try setting *both* A and B to zero and then press the *Pulse* key several times while they are zero. Then set A back to one. Try setting both to zero and then set B back to one. **For your lab report:** provide diagrams for the switch settings and resultant LED values for each time you toggle one of the buttons.

A true D flip-flop only latches the data on the D input during a clock transition from low to high. In this exercise you will simulate a D flip-flop. The circuit diagram for a true D flip-flop is

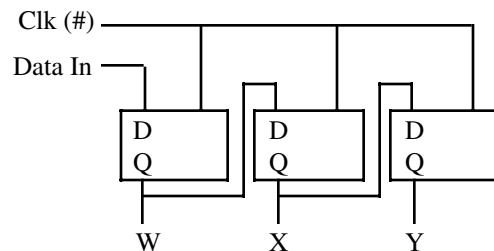


A True D flip-flop

$$\begin{aligned}
 F &= (IG)' \\
 G &= (\#F)' \\
 H &= (G\#I)' \\
 I &= (DH)' \\
 X &= (GY)' \\
 Y &= (HX)'
 \end{aligned}$$

Enter this set of equations and then test your flip-flop by entering different values on the D input switch and pressing the clock pulse button. Explain your results in your lab report.

In this exercise you will build a three-bit shift register using the logic equations for a true D flip-flop. To construct a shift register, you connect the outputs from each flip-flop to the input of the next flip-flop. The data input line provides the input to the first flip-flop, the last output line is the “carry out” of the circuit. Using a simple rectangle to represent a flip-flop and ignoring the Q' output (since we don't use it), the schematic for a four-bit shift register looks something like the following:



A Three-bit Shift Register Built from D Flip-flops

In the previous exercise you used six boolean expressions to define the D flip-flop. Therefore, we will need a total of 18 boolean expressions to implement a three-bit flip-flop. These expressions are

Flip-Flop #1:

$$\begin{aligned}
 W &= (GR)' \\
 F &= (IG)'
 \end{aligned}$$

```

G = (F#) '
H = (G#I) '
I = (DH) '
R = (HW) '

```

Flip-Flop #2:

```

X = (KS) '
J = (MK) '
K = (J#) '
L = (K#M) '
M = (WL) '
S = (LX) '

```

Flip-Flop #3:

```

Y = (OT) '
N = (QO) '
O = (N#) '
P = (O#Q) '
Q = (XP) '
T = (PY) '

```

Enter these equations into LOGIC.EXE. Initialize W, X, and Y to zero. Set D to one and press the *Pulse* button once to shift a one into W. Now set D to zero and press the pulse button several times to shift that single bit through each of the output bits. **For your lab report:** try shifting several bit patterns through the shift register. Describe the step-by-step operation in your lab report.

**For additional credit:** Describe how to create a *recirculating shift register*. One whose output from bit four feeds back into bit zero. What would be the logic equations for such a shift register? How could you initialize it (since you cannot use the D input) when using LOGIC.EXE?

**Post-lab, for additional credit:** Design a two-bit full adder that computes the sum of BA and DC and stores the binary result to the WXY LEDs. Include the equations and sample results in your lab report.

## 8.5 Laboratory Exercises for Chapters Four, Five, Six, and Seven

In this laboratory you will use the “SIMY86.EXE” program found in the Volume\Ch08 subdirectory. This program contains a built-in assembler (compiler), debugger, and interrupter for the Y86 hypothetical CPUs. You will learn how to write basic Y86 assembly language programs, assemble (compile) them, modify the contents of memory, and execute your Y86 programs. You will also experiment with memory-mapped I/O, I/O-mapped input/output, DMA, and polled as well as interrupt-driven I/O systems.

### 8.5.1 The SIMY86 Program – Some Simple Y86 Programs

To run the SIMY86 program double click on its icon or choose run from the Windows file menu and enter the pathname for SIMY86. The SIMY86 program consists of three main screen that you can select by clicking on the *Editor*, *Memory*, or *Emulator* notebook tabs in the window. By default, SIMY86 opens the Editor screen. From the Editor screen you can edit and assemble Y86 programs; from Memory screen you can view and modify the contents of memory; from the Emulator screen you execute Y86 programs and view Y86 programs in memory.

The SIMY86 program contains two menu items: File and Edit. These are standard Windows menus so there is little need to describe their operation except for two points. First, the New, Open, Save, and Save As items under the file menu manipulate the data in the text editor box on the Editor screen, they do not affect anything on the other screens. Second, the Print menu item in the File menu prints the source code appearing in the text editor if the Editor screen is active, it prints the entire form if the Memory or Emulator screens are active.

To see how the SIMY86 program operates, switch to the Editor screen (if you are not already there). Select “Open” from the File menu and choose “EX1.Y86” from the Volume2\Ch08 subdirectory. That file should look like the following:

```

mov( [1000], ax );
mov( [1002], bx );
add( bx, ax );
sub( 1, ax );
mov( ax, bx );
add( ax, bx );
add( bx, ax );
halt;

```

This short code sequence adds the two values at location 1000 and 1002, subtracts one from their sum, and multiplies the result by three  $((ax + ax) + ax) = ax * 3$ , leaving the result in AX and then it halts.

On the Editor screen you will see three objects: the editor window itself, a box that holds the “Starting Address,” and an “Assemble” button. The “Starting Address” box holds a hexadecimal number that specifies where the assembler will store the machine code for the Y86 program you write with the editor. By default, this address is zero. About the only time you should change this is when writing interrupt service routines since the default reset address is zero. The “Assemble” button directs the SIMY86 program to convert your assembly language source code into Y86 machine code and store the result beginning at the Starting Address in memory. Go ahead and press the “Assemble” button at this time to assemble this program to memory.

Now press the “Memory” tab to select the memory screen. On this screen you will see a set of 64 boxes arranged as eight rows of eight boxes. To the left of these eight rows you will see a set of eight (hexadecimal) memory addresses (by default, these are 0000, 0008, 0010, 0018, 0020, 0028, 0030, and 0038). This tells you that the first eight boxes at the top of the screen correspond to memory locations 0, 1, 2, 3, 4, 5, 6, and 7; the second row of eight boxes correspond to locations 8, 9, A, B, C, D, E, and F; and so on. At this point you should be able to see the machine codes for the program you just assembled in memory locations 0000 through 000D. The rest of memory will contain zeros.

The memory screen lets you look at and possibly modify 64 bytes of the total 64K memory provided for the Y86 processors. If you want to look at some memory locations other than 0000 through 003F, all you need do is edit the first address (the one that currently contains zero). At this time you should change the starting address of the memory display to 1000 so you can modify the values at addresses 1000 and 1002 (remember, the program adds these two values together). Type the following values into the corresponding cells: at address 1000 enter the value 34, at location 1001 the value 12, at location 1002 the value 01, and at location 1003 the value 02. Note that if you type an illegal hexadecimal value, the system will turn that cell red and beep at you.

By typing an address in the memory display starting address cell, you can look at or modify locations almost anywhere in memory. Note that if you enter a hexadecimal address that is not an even multiple of eight, the SIMY86 program disable up to seven cells on the first row. For example, if you enter the starting address 1002, SIMY86 will disable the first two cells since they correspond to addresses 1000 and 1001. The first active cell is 1002. Note the SIMY86 reserves memory locations FFF0 through FFFF for memory-mapped I/O. Therefore, it will not allow you to edit these locations. Addresses FFF0 through FFF7 correspond to read-only input ports (and you will be able to see the input values even though SIMY86 disables these cells). Locations FFF8 through FFFF are write-only output ports, SIMY86 displays garbage values if you look at these locations.

On the Memory page along with the memory value display/edit cells, there are two other entry cells and a button. The “Clear Memory” button clears memory by writing zeros throughout. Since your program’s object code and initial values are currently in memory, you should not press this button. If you do, you will need to reassemble your code and reenter the values for locations 1000 through 1003.

The other two items on the Memory screen let you set the interrupt vector address and the reset vector address. By default, the reset vector address contains zero. This means that the SIMY86 begins program execution at address zero whenever you reset the emulator. Since your program is currently sitting at location zero in memory, you should not change the default reset address.

The “Interrupt Vector” value is FFFF by default. FFFF is a special value that tells SIMY86 “there is no interrupt service routine present in the system, so ignore all interrupts.” Any other value must be the address of an ISR that SIMY86 will call whenever an interrupt occurs. Since the program you assembled does not have an interrupt service routine, you should leave the interrupt vector cell containing the value FFFF.

Finally, press the “Emulator” tab to look at the emulator screen. This screen is much busier than the other two. In the upper left hand corner of the screen is a data entry box with the label IP. This box holds the current value of the Y86 *instruction pointer* register. Whenever SIMY86 runs a program, it begins execution with the instruction at this address. Whenever you press the reset button (or enter SIMY86 for the first time), the IP register contains the value found in the reset vector. If this register does not contain zero at this point, press the reset button on the Emulator screen to reset the system.

Immediately below the IP value, the Emulator page *disassembles* the instruction found at the address in the IP register. This is the very next instruction that SIMY86 will execute when you press the “Run” or “Step” buttons. Note that SIMY86 does not obtain this instruction from the source code window on the Editor screen. Instead, it decodes the opcode in memory (at the address found in IP) and generates this string itself. Therefore, there may be minor differences between the instruction you wrote and the instruction SIMY86 displays on this page. Note that a disassembled instruction contains several numeric values in front of the actual instruction. The first (four-digit) value is the memory address of that instruction. The next pair of digits (or the next three pairs of digits) are the opcodes and possible instruction operand values. For example, the “mov( [1000], ax );” instruction’s machine code is C6 00 10 since these are the three sets of digits appearing at this point.

Below the current disassembled instruction, SIMY86 displays 15 instructions it disassembles. The starting address for this disassembly is *not* the value in the IP register. Instead, the value in the lower right hand corner of the screen specifies the starting disassembly address. The two little arrows next to the disassembly starting address let you quickly increment or decrement the disassembly starting address. Assuming the starting address is zero (change it to zero if it is not), press the down arrow. Note that this increments the starting address by one. Now look back at the disassembled listing. As you can see, pressing the down arrow has produced an interesting result. The first instruction (at address 0001) is “\*\*\*\*\*”. The four asterisks indicate that this particular opcode is an illegal instruction opcode. The second instruction, at address 0002, is “not( ax );”. Since the program you assembled did not contain an illegal opcode or a “not( ax );” instruction, you may be wondering where these instructions came from. However, note the starting address of the first instruction: 0001. This is the second byte of the first instruction in your program. In fact, the illegal instruction (opcode=00) and the “not( ax );” instruction (opcode=10) are actually a disassembly of the “mov( [1000], ax );” instruction’s two-byte operand. This should clearly demonstrate a problem with disassembly – it is possible to get “out of phase” by specifying a starting address that is in the middle of a multi-byte instruction. You will need to consider this when disassembling code.

In the middle of the Emulator screen there are several buttons: Run, Step, Halt, Interrupt, and Reset (the “Running” box is an annunciator, not a button). Pressing the Run button will cause the SIMY86 program to run the program (starting at the address in the IP register) at “full” speed. Pressing the Step button instructs SIMY86 to execute only the instruction that IP points at and then stop. The Halt button, which is only active while a program is running, will stop execution. Pressing the Interrupt button generates an interrupt and pressing the Reset button resets the system (and halts execution if a program is currently running). Note that pressing the Reset button clears the Y86 registers to zero and loads the ip register with the value in the reset vector.

The “Running” annunciator is gray if SIMY86 is not currently running a program. It turns red when a program is actually running. You can use this annunciator as an easy way to tell if a program is running if the program is busy computing something (or is in an infinite loop) and there is no I/O to indicate program execution.

The boxes with the AX, BX, CX, and DX labels let you modify the values of these registers while a program is not running (the entry cells are not enabled while a program is actually running). These cells also display the current values of the registers whenever a program stops or between instructions when you are stepping through a program. Note that while a program is running the values in these cells are static and do not reflect their current values.

The “Less” and “Equal” check boxes denote the values of the less than and equal flags. The Y86 CMP instruction sets these flags depending on the result of the comparison. You can view these values while the program is not running. You can also initialize them to true or false by clicking on the appropriate box with the mouse (while the program is not running).

In the middle section of the Emulator screen there are four “LEDs” and four “toggle switches.” Above each of these objects is a hexadecimal address denoting their memory-mapped I/O addresses. Writing a zero

to a corresponding LED address turns that LED “off” (turns it white). Writing a one to a corresponding LED address turns that LED “on” (turns it red). Note that the LEDs only respond to bit zero of their port addresses. These output devices ignore all other bits in the value written to these addresses.

The toggle switches provide four memory-mapped input devices. If you read the address above each switch SIMY86 will return a zero if the switch is off. SIMY86 will return a one if the switch is in the on position. You can toggle a switch by clicking on it with the mouse. Note that a little rectangle representing the switch turns red if the switch is in the “on” position.

The two columns on the right side of the Emulate screen (“Input” and “Output”) display input values read with the GET instruction and output values the PUT instruction prints.

For this first exercise, you will use the Step button to single step through each of the instructions in the EX1.Y86 program. First, begin by pressing the Reset button<sup>3</sup>. Next, press the Step button once. Note that the values in the IP and AX registers change. The IP register value changes to 0003 since that is the address of the next instruction in memory, AX’S value changed to 1234 since that’s the value you placed at location 1000 when operating on the Memory screen. Single step through the remaining instructions (by repeatedly pressing Step) until you get the “Halt Encountered” dialog box.

**For your lab report:** explain the results obtained after the execution of each instruction. Note that single-stepping through a program as you’ve done here is an excellent way to ensure that you fully understand how the program operates. As a general rule, you should always single-step through every program you write when testing it.

## 8.5.2 Simple I/O-Mapped Input/Output Operations

Go to the Editor screen and load the EX2.Y86 file into the editor. This program introduces some new concepts, so take a moment to study this code:

```

    mov( 1000, bx );
a:   get;
    mov( ax, [bx] );
    add( 2, bx );
    cmp( ax, 0 );
    jne a;

    mov( bx, cx );
    mov( 1000, bx )
    mov( 0, ax );
b:   add( [bx], ax );
    add( 2, bx );
    cmp( bx, cx );
    jb b;

    put;
    halt;
```

The first thing to note are the two strings “a:” and “b:” appearing in column one of the listing. The SIMY86 assembler lets you specify up to 26 statement *labels* by specifying a single alphabetic character followed by a colon. Labels are generally the operand of a jump instruction of some sort. Therefore, the “jne a;” instruction above really says “jump if not equal to the statement prefaced with the ‘a:’ label” rather than saying “jump if not equal to location ten (0Ah) in memory.”

Using labels is much more convenient than figuring out the address of a target instruction manually, especially if the target instruction appears later in the code. The SIMY86 assembler computes the address of these labels and substitutes the correct address for the operands of the jump instructions. Note that you *can* specify a numeric address in the operand field of a jump instruction. However, all numeric addresses must begin with a decimal digit (even though they are hexadecimal values). If your target address would normally

3. It is a good idea to get in the habit of pressing the Reset button before running or stepping through any program.

begin with a value in the range A through F, simply prepend a zero to the number. For example, if “jne a;” was supposed to mean “jump if not equal to location 0Ah” you would write the instruction as “jne 0a;”.

This program contains two loops. In the first loop, the program reads a sequence of values from the user until the user enters the value zero. This loop stores each word into successive memory locations starting at address 1000h. Remember, each word read by the user requires two bytes; this is why the loop adds two to bx on each iteration.

The second loop in this program scans through the input values and computes their sum. At the end of the loop, the code prints the sum to the output window using the PUT instruction.

**For your lab report:** single-step through this program and describe how each instruction works. Reset the Y86 and run this program at full speed. Enter several values and describe the result. Discuss the GET and PUT instructions. Describe why they do the equivalent of I/O-mapped input/output operations rather than memory-mapped input/output operations.

### 8.5.3 Memory Mapped I/O

From the Editor screen, load the EX3.Y86 program file. That program takes the following form (the comments were added here to make the operation of this program clearer):

```
a:    mov( [fff0], ax );
      mov( [fff2], bx );

      mov( ax, cx );    // Compute Sw0 and Sw1
      and( bx, cx );
      mov( cx, [fff8] );

      mov( ax, cx );    // Computes Sw0 OR Sw1
      or( bx, cx );
      mov( cx, [fffa] );

      mov( ax, cx );    // Computes Sw0 xor Sw1
      mov( bx, dx );    // XOR = AB' + A'B
      not( cx );
      not( dx );
      and( bx, cx );
      and( ax, dx );
      or( dx, cx );
      mov( cx, [fffc] );

      not( cx );        // Computes Sw0 = Sw1
      mov( cx, [fffe] ); // Note: equals is not xor

      mov( [fff4], ax ); // Read the third switch.
      cmp( ax, 0 );      // See if it's on.
      je a;              // Repeat this program while it's on.
      halt;
```

Locations \$FFF0, \$FFF2, and \$FFF4 correspond to the first three toggle switches on the Execution page. These are memory-mapped I/O devices that put a zero or one into the corresponding memory locations depending upon whether the toggle switch is in the on or off state. Locations \$FFF8, \$FFFA, \$FFFC, and \$FFFE correspond to the four LEDs. Writing a zero to these locations turns the corresponding LED off, writing a one turns it on.

This program computes the logical AND, OR, XOR, and XNOR (not XOR) functions for the values read from the first two toggle switches. This program displays the results of these functions on the four output LEDs. This program reads the value of the third toggle switch to determine when to quit. When the third toggle switch is in the on position, the program will stop.

**For your lab report:** run this program and cycle through the four possible combinations of on and off for the first two switches. Include the results in your lab report.

### 8.5.4 DMA Exercises

In this exercise you will start a program running (EX4.Y86) that examines and operates on values found in memory. Then you will switch to the Memory screen and modify values in memory (that is, you will directly access memory while the program continues to run), thus simulating a peripheral device that uses DMA.

The EX4.Y86 program begins by setting memory location \$1000 to zero. Then it loops until one of two conditions is met – either the user toggles the FFF0 switch or the user changes the value in memory location \$1000. Toggling the FFF0 switch terminates the program. Changing the value in memory location \$1000 transfers control to a section of the program that adds together  $n$  words, where  $n$  is the new value in memory location \$1000. The program sums the words appearing in contiguous memory locations starting at address \$1002. The actual program looks like the following:

```
d:    mov( 0, cx );           // Clear location $1000 before we begin testing it.
      mov( cx, [1000] );

// The following loop checks to see if memory location $1000 changes or if
// the FFF0 switch is in the on position.

a:    mov( [1000], cx );     // Check to see if location $1000
      cmp( cx, 0 );          // Changes. Jump to the section that
      jne c;                 // sums the values if it does.

      mov( [fff0], ax );     // If location $1000 still contains zero,
      cmp( ax, 0 );          // read the FFF0 switch and see if it is
      je a;                  // of. If so, loop back. If the switch
      halt;                  // is on, quit the program.

// The following code sums up the "cx" contiguous words of memory starting at
// memory location $1002. After it sums up these values, it prints their sum.

c:    mov( 1002, bx );       // Initialize BX to point at data array.
      mov( 0, ax );          // Initialize the sum.
b:    add( [bx], ax );        // Sum in the next array value.
      add( 2, bx );          // Point BX at the next item in the array.
      sub( 1, cx );          // Decrement the element count.
      cmp( cx, 0 );          // Test to see if we've added up all the
      jne b;                 // values in the array.

      put;                   // Print the sum and start over.
      jmp d;
```

Load this program into SIMY86 and assemble it. Switch to the Emulate screen, press the Reset button, make sure the FFF0 switch is in the off position, and then run the program. Once the program is running switch to the memory screen by pressing the Memory tab. Change the starting display address to \$1000. Change the value at location \$1000 to 5. Switch back to the emulator screen. Assuming memory locations \$1002 through \$100B all contain zero, the program should display a zero in the output column.

Switch back to the memory page. What does location \$1000 now contain? Change the L.O. bytes of the words at address \$1002, \$1004, and \$1006 to 1, 2, and 3, respectively. Change the value in location \$1000 to three. Switch to the Emulator page. Describe the output in your lab report. Try entering other values into memory. Toggle the FFF0 switch when you want to quit running this program.

**For your lab report:** explain how this program uses DMA to provide program input. Run several tests with different values in location \$1000 and different values in the data array starting at location \$1002. Include the results in your report.

**For additional credit:** Store the value \$12 into memory location \$1000. Explain why the program prints *two* values instead of just one value.

## 8.5.5 Interrupt Driven I/O Exercises

In this exercise you will load *two* programs into memory: a main program and an interrupt service routine. This exercise demonstrates the use of interrupts and an interrupt service routine.

The main program (EX5a.Y86) will constantly compare memory locations \$1000 and \$1002. If they are not equal, the main program will print the value of location \$1000 and then copy this value to location \$1002 and repeat this process. The main program repeats this loop until the user toggles switch FFF0 to the on position. The code for the main program is the following:

```
a:    mov( [1000], ax );    // Fetch the data at location $1000 and
        cmp( ax, [1002] );    // see if it is the same as location
        je b;                // $1002. If so, check the FFF0 switch.
        put;                // If the two values are different, print
        mov( ax, [1002] );    // $1000's value and make them the same.

b:    mov( [fff0], ax );    // Test the FFF0 switch to see if we
        cmp( ax, 0 );        // should quit this program.
        je a;
        halt;
```

The interrupt service routine (EX5b.Y86) sits at location \$100 in memory. Whenever an interrupt occurs, this ISR simply increments the value at location \$1000 by loading this value into AX, adding one to the value in AX, and then storing this value back to location \$1000. After these instructions, the ISR returns to the main program. The interrupt service routine contains the following code:

```
mov( ax, [1004] );        // The ISR must preserve any register it uses!
mov( [1000], ax );        // Increment the value at location $1000 by one
add( 1, ax );             // and return to the interrupted code.
mov( ax, [1000] );
mov( [1004], ax );        // Restore AX's original value.
iret;                     // Return from the interrupt.
```

You must load and assemble both files before attempting to run the main program. Begin by loading the main program (EX5a.Y86) into memory and assemble it at address zero. Then load the ISR (EX5b.Y86) into memory, set the Starting Address to 100, and then assemble your code. **Warning:** if you forget to change the starting address you will wipe out your main program when you assemble the ISR. If this happens, you will need to repeat this procedure from the beginning.

After assembling the code, the next step is to set the interrupt vector so that it contains the address of the ISR. To do this, switch to the Memory screen. The interrupt vector cell should currently contain \$FFFF (this value indicates that interrupts are disabled). Change this to \$100 so that it contains the address of the interrupt service routine. This also enables the interrupt system.

Finally, switch to the Emulator screen, make sure the FFF0 toggle switch is in the off position, reset the program, and start it running. Normally, nothing will happen. Now press the interrupt button and observe the results.

**For your lab report:** describe the output of the program whenever you press the interrupt button. Explain all the steps you would need to follow to place the interrupt service routine at address \$2000 rather than \$100.

**For additional credit:** write your own interrupt service routine that does something simple. Run the main program and press the interrupt button to test your code. Verify that your ISR works properly.

---

## 8.5.6 Machine Language Programming & Instruction Encoding Exercises

To this point you have been creating machine language programs with SIMY86's built-in assembler. An assembler is a program that translates an ASCII source file containing textual representations of a program into the actual machine code. The assembler program saves you a considerable amount of work by translat-

ing human readable instructions into machine code. Although tedious, you can perform this translation yourself. In this exercise you will create some very short *machine language* programs by encoding the instructions and entering their hexadecimal opcodes into memory on the memory screen.

Using the instruction encodings found in Figure 5.3, Figure 5.4, Figure 5.5, and Figure 5.6, write the hexadecimal values for the opcodes beside each of the following instructions:

	Binary Opcode	Hex Operand	
mov( 0, cx );	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>
a: get;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
put;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
add( ax, ax );	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
put;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
add( ax, ax );	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
put;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
add( ax, ax );	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
put;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		
add( 1, cx );	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>
cmp( cx, 4 );	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>
jb a;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>	<div style="border: 1px solid black; width: 60px; height: 15px;"></div>
halt;	<div style="border: 1px solid black; width: 100px; height: 15px; background-color: #cccccc;"></div>		

**Figure 8.1 A Simple Program to Convert to Machine Code**

You can assume that the program starts at address zero and, therefore, label *a* will be at address \$0003 since the “mov( 0, cx );” instruction is three bytes long.

**For your lab report:** enter the hexadecimal opcodes and operands into memory starting at location zero using the Memory editor screen. Dump these values and include them in your lab report. Switch to the Emulator screen and disassemble the code starting at address zero. Verify that this code is the same as the assembly code above. Print a copy of the disassembled code and include it in your lab report. Run the program and verify that it works properly.

## 8.5.7 Self Modifying Code Exercises

In the previous laboratory exercise, you discovered that the system doesn't really differentiate data and instructions in memory. You were able to enter hexadecimal data and the Y86 processor treats it as a sequence of executable instructions. It is also possible for a program to store data into memory and then execute it. A program is *self-modifying* if it creates or modifies some of the instructions it executes.

Consider the following Y86 program (EX6.Y86):

```

        sub( ax, ax );          // Trick:  sets AX to zero.
        mov( ax, [100] );

a:      mov( [100], ax );
        cmp( ax, 0 );
        je b;
        halt;

b:      mov( c6, ax );
        mov( ax, [100] );
        mov( 710, ax );
        mov( ax, [102] );
        mov( a6a0, ax );
        mov( ax, [104] );
        mov( 1000, ax );
        mov( ax, [106] );
        mov( 8007, ax );
        mov( ax, [108] );
        mov( e6, ax );
        mov( ax, [10a] );
        mov( e10, ax );
        mov( ax, [10c] );
        mov( 4, ax );
        mov( ax, [10e] );
        jmp 100;

```

This program writes the following code to location \$100 and then executes it:

```

        mov( [1000], ax );
        put;
        add( ax, ax );
        add( [1000], ax );
        put;
        sub( ax, ax );
        mov( ax, [1000] );
        put;
        sub( ax, ax );
        mov( ax, [1000] );
        jmp 0004;              // $0004 is the address of the a: label.

```

**For your lab report:** execute the EX7.Y86 program and verify that it generates the above code at location 100.

Although this program demonstrates the principle of self-modifying code, it hardly does anything useful. As a general rule, one would not use self-modifying code in the manner above, where one segment writes some sequence of instructions and then executes them. Instead, most programs that use self-modifying code only modify existing instructions and often only the operands of those instructions.

Self-modifying code is rarely found in modern assembly language programs. Programs that are self-modifying are hard to read and understand, difficult to debug, and often unstable. Programmers often resort to self-modifying code when the CPU's architecture lacks sufficient power to achieve a desired goal. The later Intel 80x86 processors do not lack for instructions or addressing modes, so it is very rare to find

80x86 programs that use self-modifying code<sup>4</sup>. The Y86 processors, however, have a very weak instruction set, so there are actually a couple of instances where self-modifying code may prove useful.

A good example of an architectural deficiency where the Y86 is lacking is with respect to subroutines. The Y86 instruction set does not provide any (direct) way to call and return from a subroutine. However, you can easily simulate a call and return using the JMP instruction and self-modifying code. Consider the following Y86 “subroutine” that sits at location \$100 in memory:

```
// Integer to Binary converter.
// Expects an unsigned integer value in AX.
// Converts this to a string of zeros and ones storing this string of
// values into memory starting at location $1000.

        mov( 1000, ax );      // Starting address of string.
        mov( 10, cx );       // 16 ($10) digits in a word.
a:       mov( 0, dx );        // Assume current bit is zero.
        cmp( ax, 8000 );     // See if AX's H.O. bit is zero or one.
        jnb b;              // Branch if AX's H.O. bit is zero.
        mov( 1, dx );       // AX's H.O. bit is one, set that here.
b:       mov( dx, [bx] );    // Store zero or one to next location.
        add( 1, bx );       // Bump BX to point at next byte in memory.
        add( ax, ax );      // AX = AX *2 (shift left operation).
        sub( 1, cx );       // Count off 16 bits.
        cmp( cx, 0 );       // Repeat 16 times.
        ja a;
        jmp 0;              // Return to caller via self-modifying code.
```

The only instruction that a program will modify in this subroutine is the very last JMP instruction. This jump instruction must transfer control to the first instruction beyond the JMP in the calling code that transfers control to this subroutine; that is, the caller must store the return address into the operand of the JMP instruction in the code above. As it turns out, the JMP instruction is at address \$120 (assuming the code above starts at location \$100). Therefore, the caller must store the return address into location \$121 (the operand of the JMP instruction). The following sample “main” program makes three calls to the “subroutine” above:

```
mov( c, ax );      // Address of the BRK instruction below.
mov( ax, [121] );  // Store into JMP as return address.
mov( 1234, ax );   // Convert $1234 to binary.
jmp 100;           // "Call" the subroutine above.
brk;              // Pause to let the user example bytes at $1000.

mov( 19, ax );     // Address of the BRK instruction below.
mov( ax, [121] );  // Store into JMP as return address.
mov( fdeb, ax );   // Convert $FDEB to binary.
jmp 100;           // "Call" the subroutine above.
brk;              // Pause to let the user example bytes at $1000.

mov( 16, ax );     // Address of the BRK instruction below.
mov( ax, [121] );  // Store into JMP as return address.
mov( 2345, ax );   // Convert $2345 to binary.
jmp 100;           // "Call" the subroutine above.
brk;              // Pause to let the user example bytes at $1000.

halt;
```

Load the subroutine (EX7s.Y86) into SIMY86 and assemble it starting at location \$100. Next, load the main program (EX7m.Y86) into memory and assemble it starting at location zero. Switch to the Emulator screen and verify that all the return addresses (\$c, \$19, and \$26) are correct. Also verify that the return

---

4. Many viruses and copy protection programs use self modifying code to make it difficult to detect or bypass them.

address needs to be written to location \$121. Next, run the program. The program will execute a BRK instruction after each of the first two calls. The BRK instruction pauses the program. At this point you can switch to the memory screen and look at locations \$1000..100F in memory. They should contain the pseudo-binary conversion of the value passed to the subroutine. Once you verify that the conversion is correct, switch back to the Emulator screen and press the Run button to continue program execution after the BRK.

**For your lab report:** describe how self-modifying code works and explain in detail how this code uses self-modifying code to simulate call and return instructions. Explain the modifications you would need to make to move the main program to address \$800 and the subroutine to location \$900.

**For additional credit:** Actually change the program and subroutine so that they work properly at the addresses above (\$800 and \$900).

---

## 8.5.8 Virtual Memory Exercise

The SIMY86 emulator treats the two 4K blocks of memory starting at addresses \$D000 and \$E000 specially. These blocks use virtual memory for their actual implementation. Only one block at a time can be in memory. If you access an address in the range \$D000..\$DFFF and that block is not currently in memory, the SIMY86 program will read the data for this block from the disk. Ditto for \$E000..\$EFFF. However, since only one of the two blocks can be in memory at a time, any attempt to access a block that is not in memory replaces other other block. If the block is “dirty” when the system needs to replace it (i.e., you’ve written data to the block) then the system first writes the data to the “paging file” before reading the other block from memory. In this laboratory exercise you will experiment with the performance of virtual memory on the Y86 hypothetical processor.

In the first exercise you will measure the amount of time a program takes to execute that exhibits spatial locality of reference. The first version of this program (EX8a.Y86) reads and writes data to locations \$B000..\$CFFF. This is our control case; we’ll run this program and time its execution to obtain a baseline to compare with our other experiments. Here’s the code for the control case:

```

        mov( 80, dx );           // Repeat the outer loop this many times (128).
a:      mov( 0, bx );           // Starting index for block one.
        mov( 0, ax );           // Write zeros to page $b000.
b:      mov( ax, [b000+bx] );
        mov( ax, [c000+bx] ); // Write zeros to page $c000 too.
        add( 2, bx );           // Move on to the next word in these pages.
        cmp( bx, 1000 );        // See if we’re done yet.
        jb b;
        sub( 1, dx );           // Repeat this whole process DX times.
        cmp( dx, 0 );
        ja a;
        halt;

```

This program will repeatedly clear locations \$D000..\$EFFF in a sequential fashion, starting at location \$D000. Note that this code writes two adjacent words in memory and bumps the index by four on each iteration of the loop. This ensures that we access all the locations in page \$D000 in a sequential fashion and then access all the locations in page \$E000 in a sequential fashion. This keeps these two pages in (physical) memory for the greatest length of time. Here’s the program (EX8b.Y86) that does the job:

```

        mov( 80, dx );           // Repeat the outer loop this many times (128).
a:      mov( 0, bx );           // Starting index for block one.
        mov( 0, ax );
b:      mov( ax, [d000+bx] ); // Write zeros to pages $d000-$e000
        mov( ax, [d002+bx] ); // Write alternate words.
        add( 4, bx );           // Move on to the next word in these pages.
        cmp( bx, 2000 );        // See if we’re done yet.
        jb b;

```

```

sub( 1, dx );          // Repeat this whole process DX times.
cmp( dx, 0 );
ja a;
halt;

```

Now load the following program into memory (EX8c.Y86) and repeat the timing of this code. This code also accesses pages \$D000 and \$E000 but the access pattern is different. Rather than accessing all the locations in page \$D000 and then accessing all the locations in \$E000, this code “ping-pongs” between the two pages, accessing a word in one page and then accessing a word in the second page. This forces the virtual memory subsystem to continuously reload the two pages on each access (i.e., thrashing occurs). Measure the amount of time it takes to execute. Record the time for your lab report. Note a major difference between this program and the previous two: the previous programs executed the outer loop 128 times while the following program only executes eight times. Be sure to multiply the running time of the following program by 16 to obtain a fair comparison of the running time of this program.

```

mov( 8, dx );          // Repeat the outer loop this many times (8).
a:  mov( 0, bx );        // Starting index for block one.
    mov( 0, ax );        // Write zeros to page $d000.
b:  mov( ax, [d000+bx] );
    mov( ax, [e000++bx] ); // Write zeros to page $e000 too.
    add( 2, bx );        // Move on to the next word in these pages.
    cmp( bx, 1000 );     // See if we're done yet.
    jb b;
    sub( 1, dx );        // Repeat this whole process DX times.
    cmp( dx, 0 );
    ja a;
    halt;

```

**For your lab report:** Measure the execution time of these three programs. Present the results in your lab report. In light of this experiment, describe how you might restructure a real program running in virtual memory to obtain the best performance.

**For additional credit:** Explain, based on your knowledge of the hardware needed to implement paging, why there is a difference in execution time between the first and second programs in this experiment.