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).

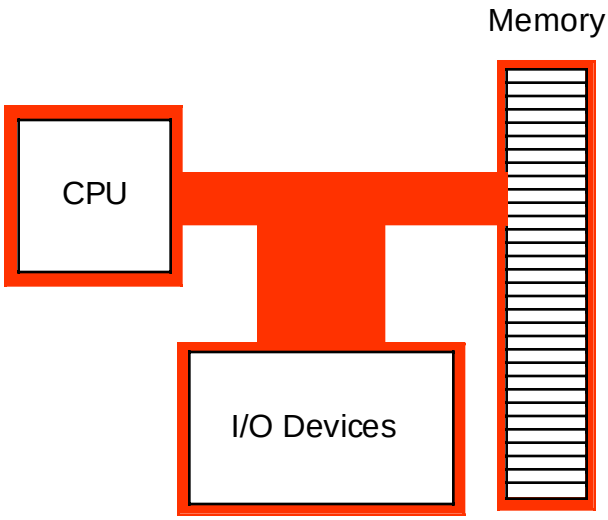

Figure 1.1 Typical Von Neumann Machine

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*[1]. This means each wire on a bus uses a standard voltage level to represent zero and one[2]. 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.

Every modern x86 CPU from the Pentium on up employs a 64-bit wide data bus. Some of the earlier processors used 8-bit, 16-bit, or 32-bit data busses, but such machines are sufficiently obsolete that we do not need to consider them here..

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.

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 minimum of either the number of data lines on the processor or the size of the largest general purpose integer register determines the processor size. The modern x86 CPUs all have 64-bit busses, but only provide 32-bit general purpose integer registers, so most people classify these devices as 32-bit processors.

Although the 80x86 family members with eight, 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 140).

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. Early x86 processors, for example, provided only 20 bit address busses. Therefore, they could only access up to 1,048,576 (or 2^{20}) memory locations. Larger address busses can access more memory.

Table 12: 80x86 Family Address Bus Sizes

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

Future 80x86 processors (e.g., the AMD "Hammer") 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.

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[3]. 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.

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

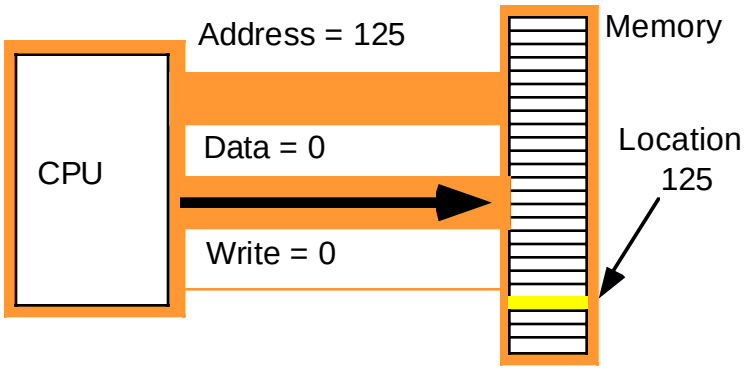

Figure 1.2 Memory Write Operation

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).

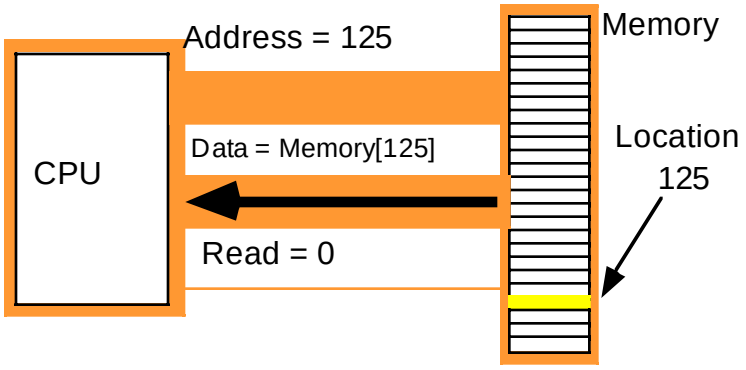

Figure 1.3 Memory Read Operation

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.

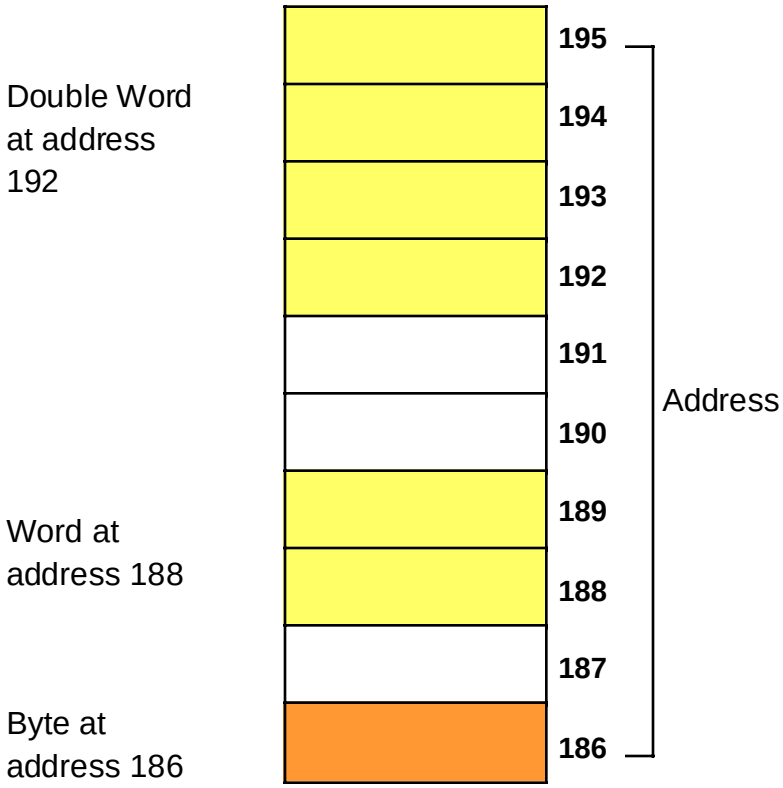

Figure 1.4 Byte, Word, and DWord Storage in Memory

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.

A processor with an eight-bit bus (like the old 8088 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.

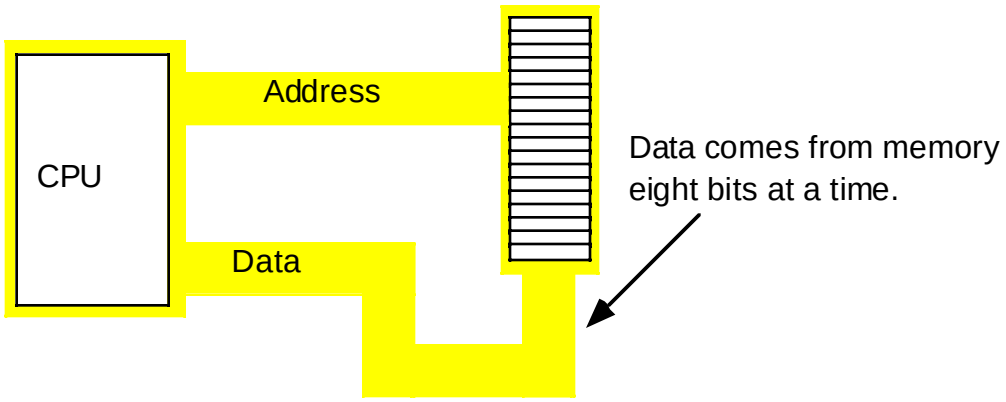

Figure 1.5 Eight-Bit CPU <-> Memory Interface

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.

CPUs with an eight-bit bus can manipulate word and double word values, even through their data bus is only eight bits wide. 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.

Some older x86 CPUs (e.g., the 8086 and 80286) 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):

	Even	Odd
Word 3	6	7
Word 2	4	5
Word 1	2	3
Word 0	0	1

Numbers in cells represent the byte addresses

Figure 1.6 Byte Addressing in Word Memory

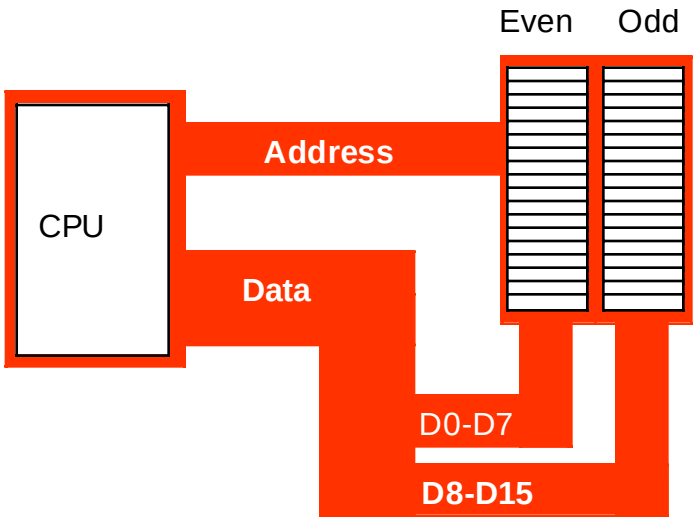

Figure 1.7 Sixteen-Bit Processor (8086, 80186, 80286, 80386sx) Memory Organization

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 on these CPUs.**

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, a 16-bit processor will require three memory operations to access the data.

The 80x86 processors with a 32-bit data bus (e.g., the 80386 and 80486) use four banks of memory connected to the 32 bit data bus (see Figure 1.8).

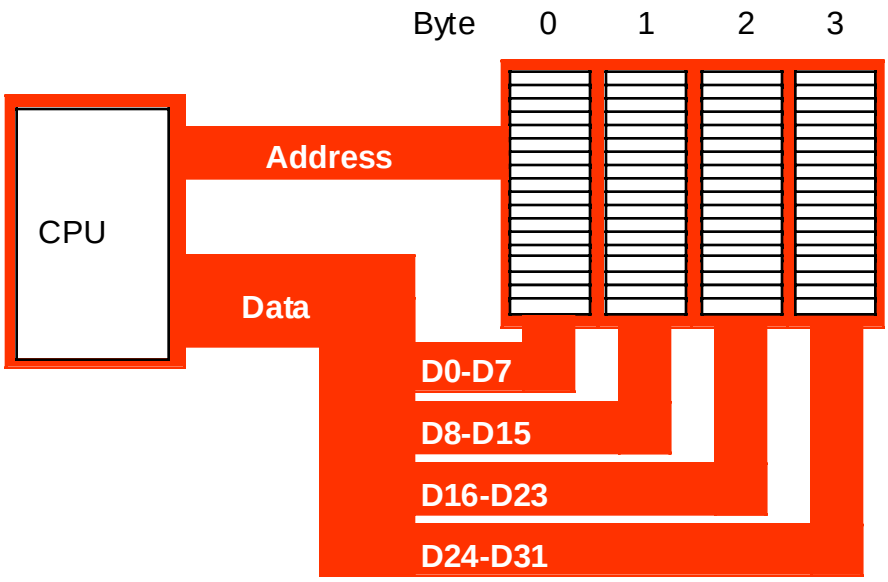

Figure 1.8 32-Bit Processor (80386, 80486, Pentium Overdrive) Memory Organization

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.

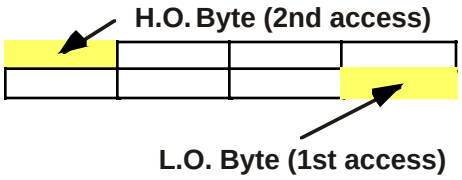

Figure 1.9 Accessing a Word at (Address mod 4) = 3.

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.

The Pentium and later processors provide a 64-bit bit data bus and special cache memory that reduces the impact of non-aligned data access. Although there may still be a penalty for accessing data at an inappropriate address, modern x86 CPUs suffer from the problem less frequently than the earlier CPUs. The discussion of cache memory in a later chapter will discuss the details.

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 video display card requires over eight megabytes of addressable 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 Windows, Linux, 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, 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.

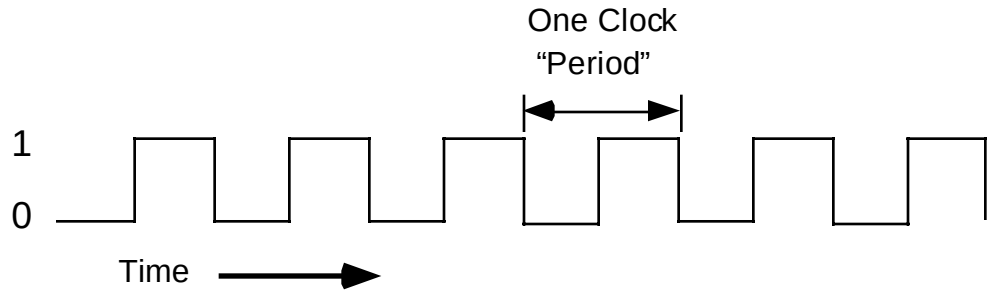

Figure 1.10 The System Clock

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 IV chip, circa 2002, runs at speeds of 2 billion cycles per second or faster. "Hertz" (Hz) is the technical term meaning one cycle per second. Therefore, the aforementioned Pentium chip runs at 2000 million hertz, or 2000 megahertz (MHz), also known as two 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 (1/1,000,000th 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 or some submultiple of 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..

Memory access time is the amount of time between a memory operation request (read or write) and the time the memory operation completes. Modern x86 CPUs are so much faster than memory that systems built around these CPUs often use a second clock, the bus clock, that is some sub-multiple of the CPU speed. For example, typical processors in the 100 MHz to 2 GHz range use 400MHz, 133MHz, 100MHz, or 66 MHz bus clocks (often, the bus speed is selectable on the CPU).

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 typical x86 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).

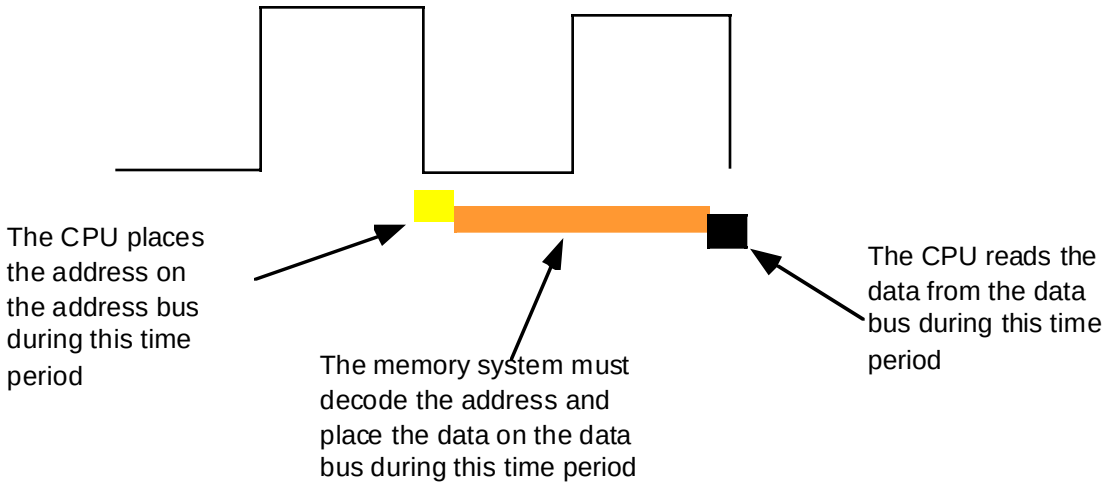

Figure 1.11 The 80x86 Memory Read Cycle

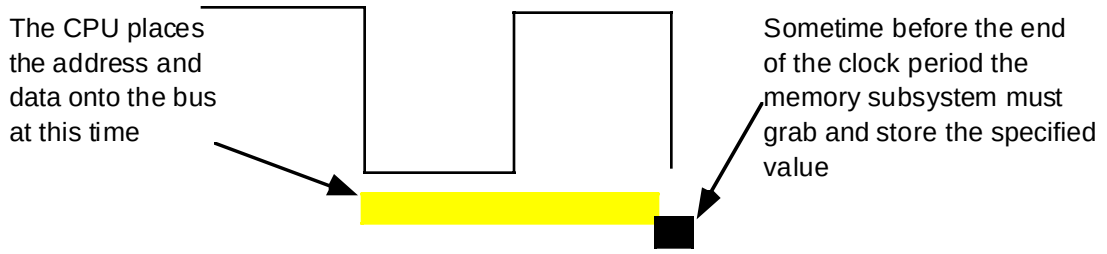

Figure 1.12 The 80x86 Memory Write Cycle

Note that the CPU doesn't wait for memory. The access time is specified by the bus 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 0.25-100 ns. You can buy bigger or faster devices, but they are much more expensive. A typical 2 GHz Pentium system uses 2.5 ns (400 MHz) memory devices.

Wait just a second here! At 2 GHz the clock period is roughly 0.5 ns. How can a system designer get away with using 2.5 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 100 MHz Penitum system has a 10 ns clock period. This implies that you need 10 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 10 ns, the memory must respond in less than 0 ns (which is impossible).

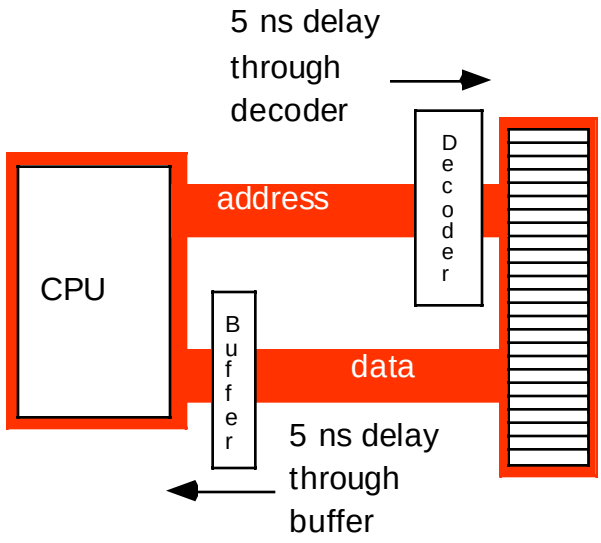

Figure 1.13 Decoding and Buffer Delays

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 2 GHz processor with a memory cycle time of 0.5 ns and you lose 2 ns to buffering and decoding, you'll need 2.5 ns memory. What if your system can only support 10 ns memory (i.e., a 100 MHz system bus)? Adding three wait states to extend the memory cycle to 10 ns (one bus clock cycle) will solve this problem.

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).

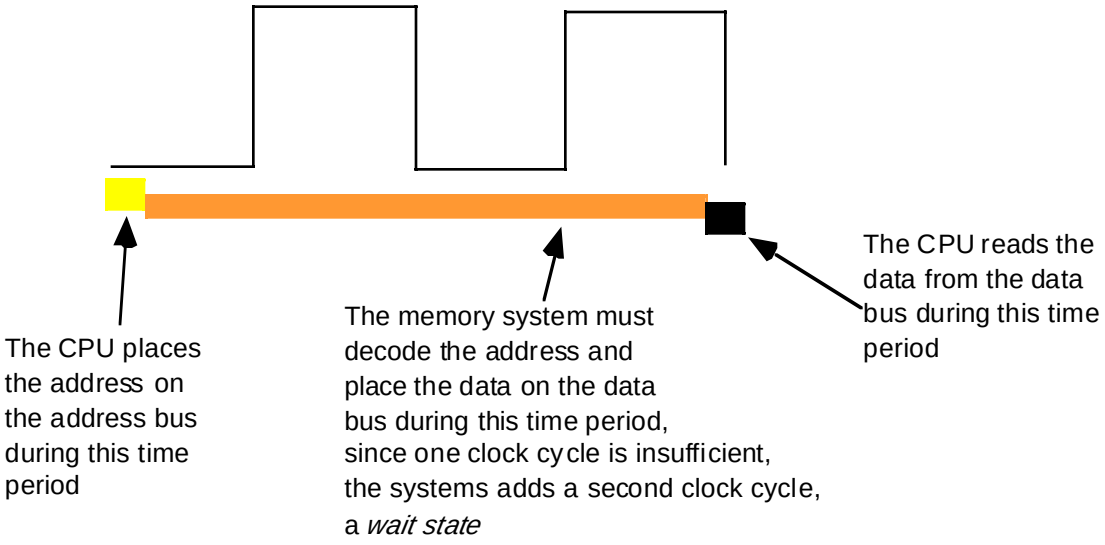

Figure 1.14 Inserting a Wait State into a Memory Read Operation

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 a typical 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 that tell the system to do the specified task also reside 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[4]. 80x86 CPUs, for example, read beween 16 and 64 bytes at a shot (depending upon the CPU) 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 in the average case.

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[5]. Altogether, the system will require 15 clock cycles to access five memory locations,

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

5. 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.

or three clock cycles per access, on the average. That's equivalent to two wait states added to every memory access. Doesn't sound as impressive, does it?

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 in our 80486 example – a substantial improvement. Alas, you can't pull an 80486 chip apart and solder more cache onto the chip. However, modern Pentium CPUs have a significantly larger cache than the 80486 and operates with fewer average 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).

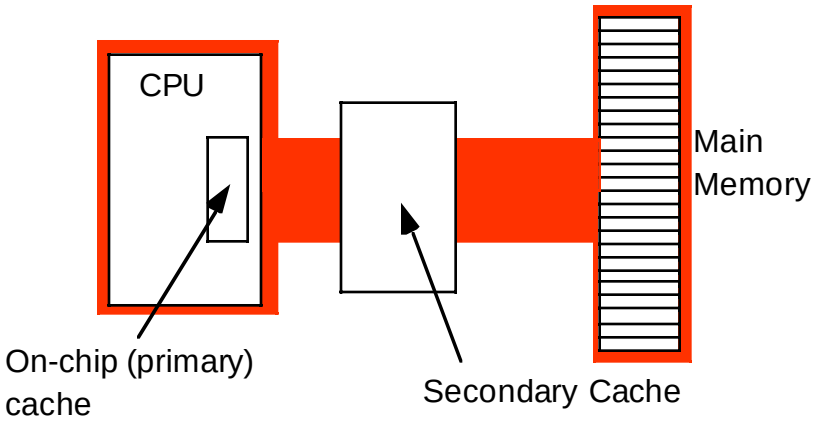

Figure 1.15 A Two Level Caching System

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.

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 fast memory 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 this volume will expand upon these comments to give you a complete overview of computer system organization.

