**Chapter Two**

**Inside Intel 8086**

1. **Inside the Intel 8086**
   1. **Revision on computer architecture and organization**

***Computer Architecture*** is the study of ***blocks*** or components that make up computer system and how they are interconnected.

Two famous computer architectures are the Von Neumann (or Stored Program) and Harvard Architectures.

* + 1. **Von Neumann architecture**

The so-called **von Neumann architecture** is a model for a computing machine that uses a single storage structure to hold both the set of instructions on how to perform the computation and the data required or generated by the computation. Such machines are also known as **stored-program computers**. The separation of storage from the processing unit is implicit in this model.

The architecture is named after mathematician John von Neumann who provided an early written account of a general purpose stored-program computing machine.

By treating the instructions in the same way as the data, a stored-program machine can easily change the instructions. In other words the machine is reprogrammable. One important motivation for such a facility was the need for a program to increment or otherwise modify the address portion of instructions. This became less important when index registers and indirect addressing became customary features of machine architecture.

The Von Neumann bottleneck: CPU and Memory separation is implicit. Trends in high improvement on CPU performance but lagging performance in memory degrade overall performance.

* + 1. **Harvard Architecture**

The term **Harvard architecture** originally referred to computer architectures that used physically separate storage and signal pathways for their instructions and data (in contrast to the *von Neumann architecture*). The term originated from the Harvard Mark I relay-based computer, which stored instructions on punched tape and data in relay latches.

In a computer with a von Neumann architecture, the CPU can be either reading an instruction or reading/writing data from/to the memory. Both cannot occur at the same time since the instructions and data use the same signal pathways and memory. In a computer with Harvard architecture, the CPU can read both an instruction and data from memory at the same time. A computer with Harvard architecture can be faster because it is able to fetch the next instruction at the same time it completes the current instruction. Speed is gained at the expense of more complex electrical circuitry.

In recent years the speed of the CPU has grown many times in comparison to the access speed of the main memory. Care needs to be taken to reduce the number of times main memory is accessed in order to maintain performance. If, for instance, every instruction run in the CPU requires an access to memory, the computer gains nothing for increased CPU speed - a problem referred to as being *memory bound*.

Memory can be made much faster, but only at high cost. The solution then is to provide a small amount of very fast memory known as a cache. As long as the data the CPU needs is in the cache, the performance hit is very much less than it is if the cache then has to turn around and get the data from the main memory. Tuning the cache is an important aspect of computer design.

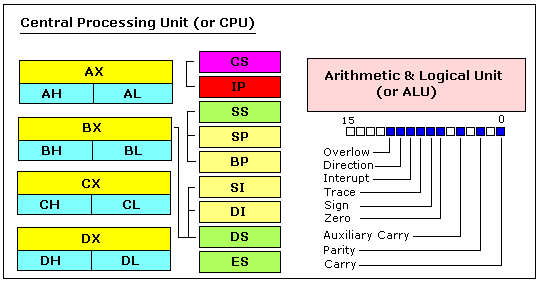
*Modern high performance CPU chip designs incorporate aspects of both Harvard and von Neumann architecture. On chip cache memory is divided into an instruction cache and a data cache. Harvard architecture is used as the CPU accesses the cache. In the case of a cache miss, however, the data is retrieved from the main memory, which is not divided into separate instruction and data sections. Thus a von Neumann architecture is used for off chip memory access.*

**Computer Organization:** is a study that is concerned with the *implementation* of computer architecture.

**Computer Engineering:** is a field of study that concerned with the actual construction of a system. Examples are length of wires, size of circuits, level of integration, cooling, etc.

* 1. **Intel 8086**

The 8086 was launched in 1978 as a fully 16-bit extension of Intel's 8-bit based 8080 microprocessor and also introduced segmentation to overcome the 16-bit addressing barrier of such designs.



* + 1. **General Purpose Registers**

The Intel 8086 CPU has 8 general purpose registers, each register has its own name:

* **AX** - the accumulator register (divided into **AH / AL**).
* **BX** - the base address register (divided into **BH / BL**).
* **CX** - the count register (divided into **CH / CL**).
* **DX** - the data register (divided into **DH / DL**).
* **SI** - source index register.
* **DI** - destination index register.
* **BP** - base pointer.
* **SP** - stack pointer.

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The main purpose of a register is to keep a number (variable). The size of the above registers is 16 bit, it's something like: **0011000000111001b** (in binary form), or **1234** in hexadecimal form.

Four general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= **0011000000111001b**, then AH=**00110000b** and AL=**00111001b**. Therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. The same is for other 3 registers, "H" is for high and "L" is for low part.

Because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep values in the registers. Register sets are very small and most registers have special purposes which limit their use as storage for values, but they are still an excellent place to store temporary data of calculations.

* + 1. **Segment Registers**
* **CS** - points at the memory segment containing the current program.
* **DS** - generally points at the memory segment where variables are defined.
* **ES** - extra segment register, it's up to a coder to define its usage (usually I/O operations).
* **SS** - points at the memory segment containing the stack.

Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.

Segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, since this way we can access much more memory than with a single register that is limited to 16 bit values.  
CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h \* 10h + 45h = 12345h):

C:\Users\Tayea\Desktop\CS302\4. Help_from_EMU8086(run_Index)-for-Intel8086-Instructions\effective_address.gif  
The address formed with 2 registers is called an **effective address**.   
By default **BX, SI** and **DI** registers work with **DS** segment register;  
**BP** and **SP** work with **SS** segment register.  
Other general purpose registers cannot form an effective address!   
Also, although **BX** can form an effective address, **BH** and **BL** cannot!

* + 1. **Special Purpose Registers**
* **IP** - the instruction pointer.
* **Flags Register** - determines the current state of the processor.

**IP** register always works together with **CS** segment register and it points to currently executing instruction.

**Flags Register** is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program. Generally you cannot access these registers directly.

* 1. **More on computer architecture and organization**

Although Computers are different in their architectural designs, the over all system concept remains roughly the same. All general-purpose computers require the following hardware components:

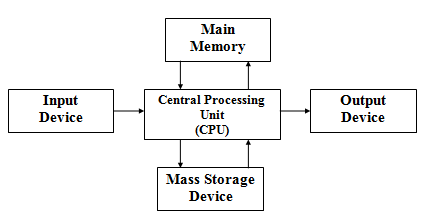
*Memory:* Enables a computer to store, at least temporarily, data and programs.

*Mass storage device:* Allows a computer to permanently retain large amounts of data. Common mass storage devices include disk drives and tape drives.

*Input device:* Usually a keyboard and mouse, the input device is the conduit through which data and instructions enter a computer.

*Output device:* A display screen, printer, or other device that lets you see what the computer has accomplished.

*Central processing unit (CPU):* The heart of the computer, this is the component that actually executes instructions.



* + 1. **Von Neumann architecture**

The so-called **von Neumann architecture** is a model for a computing machine that uses a single storage structure to hold both the set of instructions on how to perform the computation and the data required or generated by the computation. Such machines are also known as **stored-program computers**. The separation of storage from the processing unit is implicit in this model.

The architecture is named after mathematician John von Neumann who provided an early written account of a general purpose stored-program computing machine.

By treating the instructions in the same way as the data, a stored-program machine can easily change the instructions. In other words the machine is reprogrammable. One important motivation for such a facility was the need for a program to increment or otherwise modify the address portion of instructions. This became less important when index registers and indirect addressing became customary features of machine architecture.

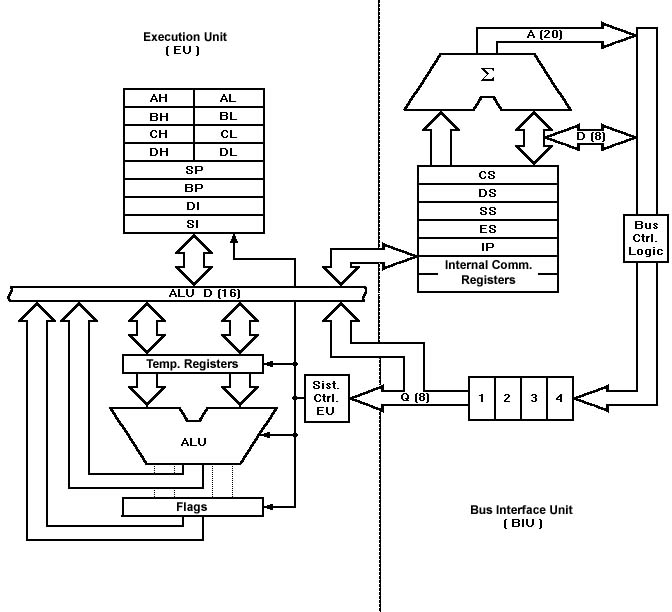
* + 1. **8086/8088 CPU Architecture**

The microprocessors functions as the CPU in the stored program model of the digital computer. Its job is to generate all system-timing signals and synchronize the transfer of data between memory, I/O, and itself. It accomplishes this task via the three-bus system architecture.

The microprocessor must recognize, decode, and execute program instructions fetched from the memory unit. This requires an Arithmetic-Logic Unit (ALU) within the CPU to perform arithmetic and logical (AND, OR, NOT, compare, etc) functions.

The 8086 CPU is organized as two separate processors, called the Bus Interface Unit (BIU) and the Execution Unit (EU). The BIU provides H/W functions, including generation of the memory and I/O addresses for the transfer of data between the outside world -outside the CPU, that is- and the EU.

The EU receives program instruction codes and data from the BIU, executes these instructions, and store the results in the general registers. By passing the data back to the BIU, data can also be stored in a memory location or written to an output device. Note that the EU has no connection to the system buses. It receives and outputs all its data thru the BIU.

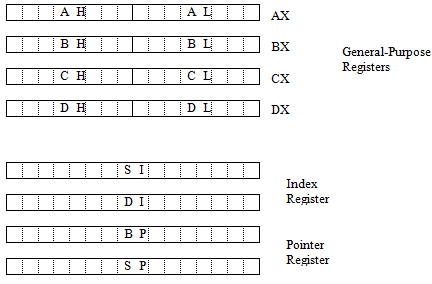


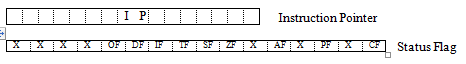
The only difference between an 8088 microprocessor and an 8086 microprocessor is the BIU. In the 8088, the BIU data bus path is 8 bits wide versus the 8086's 16-bit data bus. Another difference is that the 8088-instruction queue is four bytes long instead of six.

The important point to note, however, is that because the EU is the same for each processor, the programming instructions are exactly the same for each. Programs written for the 8086 can be run on the 8088 without any changes.

* + 1. **8086/8088 CPU Registers**

The 8086/8088 contains four general purpose registers, two index registers, two pointer registers, four segment registers, an instruction pointer register and flag register.





The first four general-purpose registers are separable and can be treated as 8 bits at a time. Those registers can be regarded as if they were four 16-bit registers or eight 8-bit registers or any combination of 16-bit and 8-bit registers.

CF (0) - Carry Flag: Set on high-order bit carry or borrow; cleared otherwise

PF (2) - parity Flag: Set if low-order 8 bits of result contain an even number of 1-bits; cleared otherwise

AF (4) - Auxiliary Flag: Set on carry from or borrow to the low-order 4 bits of AL; cleared otherwise

ZF (6) - Zero Flag: Set if result is zero; cleared otherwise

SF (7) - Sign Flag: Set equal to high-order bit of result (0 is positive, 1 if negative)

TF (8) - Single step Flag: Once set, a single-step interrupt occurs after the next instruction executes; TF is cleared by the single-step interrupt.

IF (9) - Interrupt-enable Flag: When set, maskable interrupt will cause the CPUto transfer control to an interrupt vector specified location.

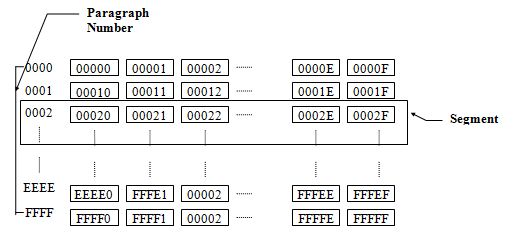
DF (10)- Direction Flag: Causes string instructions to auto decrement the appropriate index register when set; Clearing DF causes auto increment.

OF (11)- Overflow Flag: Set if the signed result cannot be expressed within the number of bits in the destination operand; cleared otherwise.

* + 1. **8086/8088 Memory Address Space**

The 8086/8088 is designed with 1 megabyte of memory address space (MAS). Since its address is 20 bits wide, the 8086/8088 can specify 220 (1,048,576) different address. When the address of a location in MAS is referred to in terms of its sequential position in MAS, that address is called an absolute address. Absolute addresses are typically represented as five-digit hexadecimal numbers. At the extreme bottom of MAS is the byte with absolute address $00000, followed by the byte with absolute address $00001, and then absolute address $00002, and so forth. At the other extreme is the byte with the highest address in MAS, at absolute address $FFFFF (1,048,575 in decimal).

The 8086/8088 organizes its 1,048,576 bytes of MAS into 16-byte by 65,536-byte matrix.



Each row is called a paragraph. Each paragraph is identified by a sequentially assigned paragraph number. Paragraphs are numbered starting from $0000 through $FFFF. A block of memory that begins on a paragraph boundary and extends for 65,536($10000) bytes is called a physical segment of memory address space. Every physical segment has a segment number, a base and a range. The segment number of a physical segment is the paragraph number of the paragraph at which it begins. The base of a physical segment is the absolute address of the first byte in that paragraph. The range of a physical segment extends from its base to the highest absolute address within it. The base of a physical segment is its segment number times $10. The range of a physical segment extends from its base through the byte at the absolute address equal to its base plus $FFFF.

Representing absolute addresses from $00000 through $FFFFF requires 20 bits, but the 8086/8088 is a 16-bit processor. It doesn’t have any accessible registers capable of holding a 20-bit number or, any number greater than $FFFF. The 8086/8088 manages its oversized address space by using segment-relative format. A segment-relative address consists of two components, a segment number and an offset. Each component of a segment relative address must be in the range from $0000 through $FFFF, and the value of each component can be stored in a 16-bit register. Specifying a segment number identifies one of the 65,536 paragraph boundaries in MAS. Specifying an offset identifies one of the 65,536 absolute address within that segment. By convention, segment-relative address are expressed as a segment number and an offset, both given in hexadecimal notation and separated by a colon (ex 0020:E000). The CPU would find the absolute address of the segment-related address by adding the offset to the absolute address of the byte at the base of the segment.

Example:

The absolute address of 0020:E000 is

0 0 2 0 0 **(Absolute address of the byte at the base of segment $0200)**

+ E 0 0 0 **(Offset of the byte at 0020:E000)**

0 E 0 0 0 **(Absolute address of the byte at** **0020:E000)**

The 8086/8088 has four segment registers - the CS, SS, DS, and ES registers. It can process a reference to an address in memory only if that address is expressed in segment-relative format and only if the segment -number component of the address is stored in a segment register.

* 1. **Detail Concepts of This Chapter (Optional Part)**

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 80486 systems. Likewise, programming techniques that provide big performance boosts on an

80486 chip may not help at all on an 80286. Fortunately, some programming techniques work well whatever microprocessor you’re using. This section discusses the effect hardware has on the performance of computer software.

* + 1. **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 section 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 that operate and 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.

Students might argue that this section 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. 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 80486 systems. Likewise, programming techniques that provide big performance boosts on an 80486 chip may not help at all on an 80286. Fortunately, some programming techniques work well whatever microprocessor you’re using. This chapter discusses the effect hardware has on the performance of computer software.

To understand how to improve system performance, it’s time to explore the internal operation of the CPU. Unfortunately, the processors in the 80x86 family are complex beasts. Discussing their internal operation would probably cause more confusion than enlightenment. There are also simplified processors like 886, 8286, 8486, and 8686 processors (the ‘x86’ processors).

This section is extreme simplifications of various members of the 80x86 family. They highlight the important architectural features of the 80x86

* + 1. **The 80x86 CPUs: A Programmer’s View**

Now it’s time to discuss some real processors: the 8088/8086, 80188/80186, 80286, and

80386/80486/80586/Pentium. While these hardware components affect the way you should write software, there is more to a CPU than bus cycles and pipelines. It’s time to look at those components of the CPU which are most visible to you, the assembly language programmer. The most visible component of the CPU is the register set. Like our hypothetical processors, the 80x86 chips have a set of on-board registers. The register set for each processor in the 80x86 family is a superset of those in the preceding CPUs. The best place to start is with the register set for the 8088, 8086, 80188, and 80186 since these four processors have the same registers. In the discussion which follows, the term ‘8086’ will imply any of these four CPUs.

Intel’s designers have classified the registers on the 8086 into three categories: general purpose registers, segment registers, and miscellaneous registers. The general purpose registers are those which may appear as operands of the arithmetic, logical, and related instructions. Although these registers are ‘general purpose’, everyone has its own special purpose. Intel uses the term ‘general purpose’ loosely. The 8086 uses the segment registers to access blocks of memory called, surprisingly enough, segments. See ‘Segments on the 80x86’ on page 151 for more details on the exact nature of the segment registers. The final classes of 8086 registers are the miscellaneous registers. There are two special registers in this group which we’ll discuss shortly.

* + - 1. **8086 General Purpose Registers**

There are eight 16 bit general purpose registers on the 8086: ax , bx , cx , dx , si , di , bp , and sp . While you can use many of these registers interchangeably in a computation, many instructions work more efficiently or absolutely require a specified register from this group.

So much for general purpose. The ax register (Accumulator) is where most arithmetic and logical computations take place. Although you can do most arithmetic and logical operations in other registers, it is often more efficient to use the ax register for such computations. The bx register (Base) has some special purposes as well. It is commonly used to hold indirect addresses, much like the bx register on the x86 processors. The cx register (Count), as its name implies, counts things. You often use it to count off the number of iterations in a loop or specify the number of characters in a string. The dx register (Data) has two special purposes: it holds the overflow from certain arithmetic operations, and it holds I/O addresses when accessing data on the 80x86 I/O bus.

The si and di registers (Source Index and Destination Index) have some special purposes as well. You may use these registers as pointers (much like the bx register) to indirectly access memory. You’ll also use these registers with the 8086 string instructions when processing character strings. The bp register (Base Pointer) is similar to the bx register. You’ll generally use this register to access parameters and local variables in a procedure. The sp register (Stack Pointer) has a very special purpose - it maintains the program stack Normally, you would not use this register for arithmetic computations. The proper operation of most programs depends upon the careful use of this register. Besides the eight 16 bit registers, the 8086 CPUs also have eight 8 bit registers.

Intel calls these registers al, ah, bl, bh, cl, ch, dl, and dh. You’ve probably noticed a similarity between these names and the names of some 16 bit registers (ax, bx, cx, and dx, to be exact). The eight bit registers are not independent. al stands for ‘ax ‘s L.O. byte.’ ah stands for ‘ax ‘s H.O. byte.’ The names of the other eight bit registers mean the same thing with respect to bx , cx and dx. Figure 2.1 shows the general purpose register set.

Note that the eight bit registers do not form an independent register set. Modifying al will change the value of ax; so will modifying ah. The value of al exactly corresponds to bits zero through seven of ax. The value of ah corresponds to bits eight through fifteen of ax. Therefore any modification to al or ah will modify the value of ax. Likewise, modifying ax will change both al and ah. Note, however, that changing al will not affect the value of ah, and vice versa. This statement applies to bx/bl/bh, cx/cl/ch, and dx/dl/dh as well.

The si, di, bp, and sp registers are only 16 bits. There is no way to directly access the individual bytes of these registers as you can the low and high order bytes of ax, bx, cx, and dx

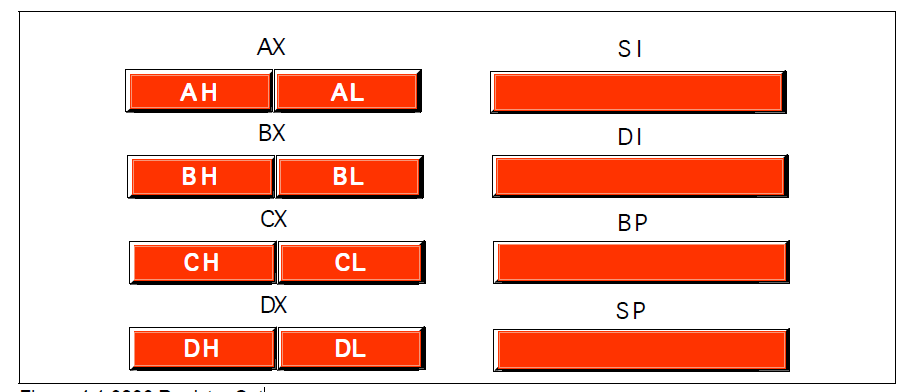
. 

Figure 2.1 8086 Register Set

* + - 1. **8086 Segment Registers**

The 8086 has four special segment registers: cs, ds, es, and ss. These stand for Code Segment, Data Segment, Extra Segment, and Stack Segment, respectively. These registers are all 16 bits wide. They deal with selecting blocks (segments) of main memory. A segment register (e.g., cs ) points at the beginning of a segment in memory. Segments of memory on the 8086 can be no larger than 65,536 bytes long.

This infamous ‘64K segment limitation’ has disturbed many a programmer. We’ll see some problems with this 64K limitation, and some solutions to those problems, later. The cs register points at the segment containing the currently executing machine instructions.

Note that, despite the 64K segment limitation, 8086 programs can be longer than 64K. You simply need multiple code segments in memory. Since you can change the value of the cs register, you can switch to a new code segment when you want to execute the code located there.

The data segment register, ds, generally points at global variables for the program. Again, you’re limited to 65,536 bytes of data in the data segment; but you can always change the value of the ds register to access additional data in other segments.

The extra segment register, es, is exactly that - an extra segment register. 8086 programs often use this segment register to gain access to segments when it is difficult or impossible to modify the other segment registers.

The ss register points at the segment containing the 8086 stack. The stack is where the

8086 stores important machine state information, subroutine return addresses, procedure parameters, and local variables. In general, you do not modify the stack segment register because too many things in the system depend upon it.

Although it is theoretically possible to store data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory. Any attempt to use the registers for any other purpose may result in considerable grief, especially if you intend to move up to a better CPU like the 80386.

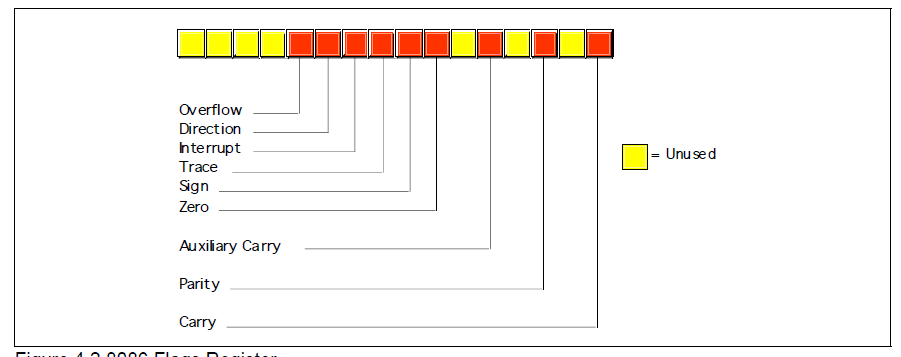


Figure 2.2 8086 Flags Register

* + - 1. **8086 Special Purpose Registers**

There are two special purpose registers on the 8086 CPU: the instruction pointer (ip) and the flags register. You do not access these registers the same way you access the other 8086 registers. Instead, the CPU generally manipulates these registers directly.

The ip register is the equivalent of the ip register on the x86 processors - it contains the address of the currently executing instruction. This is a 16 bit register which provides a pointer into the current code segment (16 bits lets you select any one of 65,536 different memory locations). We’ll come back to this register when we discuss the control transfer instructions later.

The flags register is unlike the other registers on the 8086. The other registers hold eight or 16 bit values. The flags register is simply an eclectic collection of one bit values which help determine the current state of the processor. Although the flags register is 16 bits wide, the 8086 uses only nine of those bits. Of these flags, four flags you use all the time: zero, carry, sign, and overflow. These flags are the 8086 condition codes. The flags register appears in Figure 2.2.

* + - 1. **80x86 Physical Memory Organization**

Chapter Three discussed the basic organization of a Von Neumann Architecture (VNA) computer system. In a typical VNA machine, the CPU connects to memory via the bus. The 80x86 selects some particular memory element using a binary number on the address bus. Another way to view memory is as an array of bytes. A Pascal data structure that roughly corresponds to memory would be:

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

The value on the address bus corresponds to the index supplied to this array. E.g., writing

data to memory is equivalent to

Memory [address]:= Value\_to\_Write;

Reading data from memory is equivalent to

Value\_Read:= Memory [address];

Different 80x86 CPUs have different address busses that control the maximum number of elements in the memory array). However, regardless of the number of address lines on the bus, most computer systems do not have one byte of memory for each addressable location. For example, 80386 processors have 32 address lines allowing up to four gigabytes of memory. Very few 80386 systems actually have four gigabytes. Usually, you’ll find one to 256 megabytes in an 80x86 based system.

The first megabyte of memory, from address zero to 0FFFFFh is special on the 80x86.

This corresponds to the entire address space of the 8088, 8086, 80186, and 80188 microprocessors. Most DOS programs limit their program and data addresses to locations in this range. Addresses limited to this range are named real addresses after the 80x86 real mode.

* + - 1. **Segments on the 80x86**

You cannot discuss memory addressing on the 80x86 processor family without Þrst discussing segmentation. Among other things, segmentation provides a powerful memory management mechanism. It allows programmers to partition their programs into modules that operate independently of one another. Segments provide a way to easily implement object-oriented programs. Segments allow two processes to easily share data.

All in all, segmentation is a really neat feature. On the other hand, if you ask ten programmers what they think of segmentation, at least nine of the ten will claim it’s terrible. Why such a response?

What segmentation is all about? Consider the current view of memory: it looks like a linear array of bytes. A single index (address) selects some particular byte from that array. Let’s call this type of addressing linear or flat addressing. Segmented addressing uses two components to specify a memory location: a segment value and an offset within that segment. Ideally, the segment and offset values are independent of one another. The best way to describe

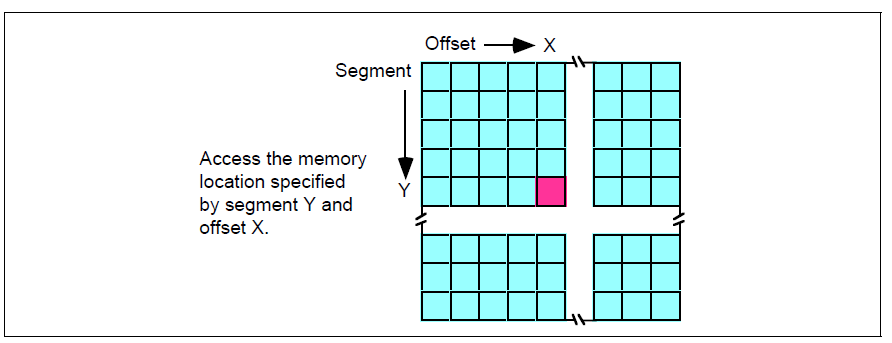


Figure 2.4 Segmented Addressing as a Two-Dimensional Process

segmented addressing is with a two-dimensional array. The segment provides one of the indices into the array, the offset provides the other (see Figure 2.4).

Now you may be wondering, ‘Why make this process more complex?’ Linear addresses seem to work fine, why bother with this two dimensional addressing scheme?

Well, let’s consider the way you typically write a program. If you were to write, say, a SIN(X) routine and you needed some temporary variables, you probably would not use global variables. Instead, you would use local variables inside the SIN(X) function. In a broad sense, this is one of the features that segmentation offers - the ability to attach blocks of variables (a segment) to a particular piece of code. You could, for example, have a segment containing local variables for SIN, a segment for SQRT, a segment for DRAWWindow, etc. Since the variables for SIN appear in the segment for SIN, it’s less likely your SIN routine will affect the variables belonging to the SQRT routine. Indeed, on the 80286 and later operating in protected mode, the CPU can prevent one routine from accidentally modifying the variables in a different segment.

A full segmented address contains a segment component and an offset component. This text will write segmented addresses as segment:offset. On the 8086 through the 80286, these two values are 16 bit constants. On the 80386 and later, the offset can be a 16 bit constant or a 32 bit constant.

The size of the offset limits the maximum size of a segment. On the 8086 with 16 bit offsets, a segment may be no longer than 64K; it could be smaller (and most segments are), but never larger. The 80386 and later processors allow 32 bit offsets with segments as large as four gigabytes.

The segment portion is 16 bits on all 80x86 processors. This lets a single program have up to 65,536 different segments in the program. Most programs have less than 16 segments (or thereabouts) so this isn’t a practical limitation.

Of course, despite the fact that the 80x86 family uses segmented addressing, the actual (physical) memory connected to the CPU is still a linear array of bytes. There is a function that converts the segment value to a physical memory address. The processor then adds the offset to this physical address to obtain the actual address of the data in memory. This text will refer to addresses in your programs as segmented addresses or logical addresses.

The actual linear address that appears on the address bus is the physical address (see Figure2.4). On the 8086, 8088, 80186, and 80188 (and other processors operating in real mode), the function that maps a segment to a physical address is very simple. The CPU multiplies the segment value by sixteen (10h) and adds the offset portion. For example, consider the segmented address2: 1000:1F00. To convert this to a physical address you multiply the seg-

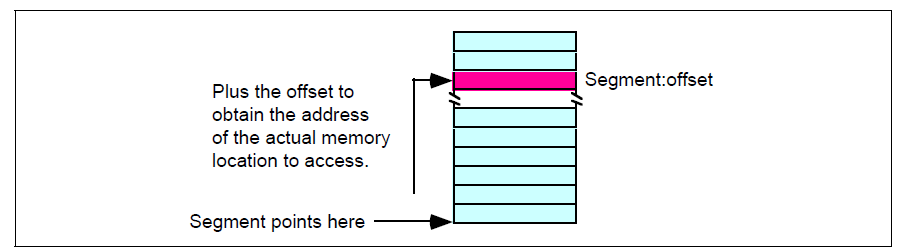


Figure 2.5 Segmented Addressing in Physical Memory

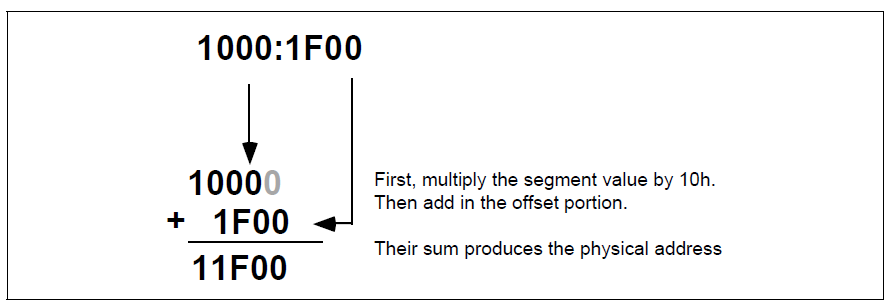


Figure 2.6 Converting a Logical Address to a Physical Address

ment value (1000h) by sixteen. Multiplying by the radix is very easy. Just append a zero to the end of the number. Appending a zero to 1000h produces 10000h. Add 1F00h to this to obtain 11F00h. So 11F00h is the physical address that corresponds to the segmented address 1000:1F00 (see Figure 2.4).

**Warning**: A very common mistake people make when performing this computation is to forget they are working in hexadecimal, not decimal. It is surprising to see how many people add 9+1 and get 10h rather than the correct answer 0Ah. Intel, when designing the 80286 and later processors, did not extend the addressing by adding more bits to the segment registers. Instead, they changed the function the CPU uses to convert a logical address to a physical address. If you write code that depends on the ‘multiply by sixteen and add in the offset’ function, your program will only work on an 80x86 processor operating in real mode, and you will be limited to one megabyte of memory3.

In the 80286 and later processors, Intel introduced protected mode segments. Among other changes, Intel completely revamped the algorithm for mapping segments to the linear address space. Rather than using a function (such as multiplying the segment value by 10h), the protected mode processors use a look up table to compute the physical address. In protected mode, the 80286 and later processors use the segment value as the index into an array. The contents of the selected array element provide (among other things) the starting address for the segment. The CPU adds this value to the offset to obtain the physical address (see Figure 2.4).

Note that your applications cannot directly modify the segment descriptor table (the lookup table). The protected mode operating system (UNIX, Linux, Windows, OS/2, etc.) handles that operation. Figure 4.5 Segmented Addressing in Physical Memory Segment:offset Segment points here Plus the offset to obtain the address of the actual memory location to access.

First, multiply the segment value by 10h. Then add in the offset portion. Their sum produces the physical address

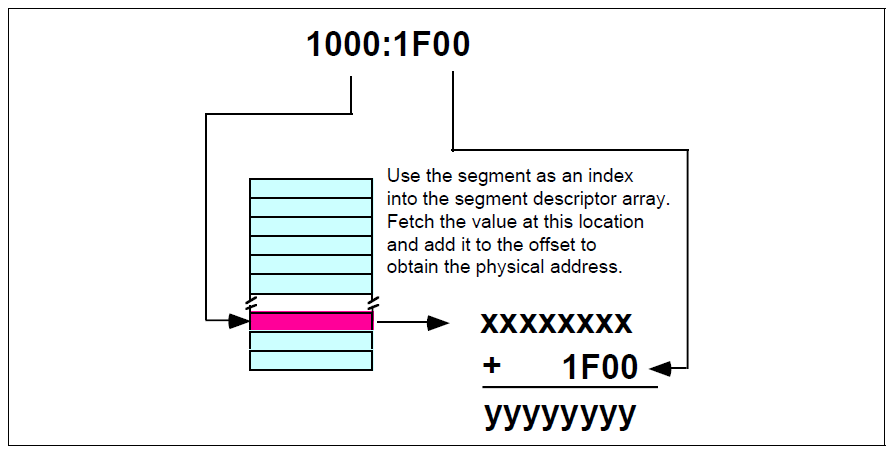


Figure 4.7 Converting a Logical Address to a Physical Address in Protected Mode

The best programs never assume that a segment is located at a particular spot in memory. You should leave it up to the operating system to place your programs into memory and not generate any segment addresses on your own.

* + - 1. **Segment Registers on the 80x86**

When Intel designed the 8086 in 1976, memory was a precious commodity. They designed their instruction set so that each instruction would use as few bytes as possible.

This made their programs smaller so computer systems employing Intel processors would use less memory. As such, those computer systems cost less to produce. Of course, the cost of memory has plummeted to the point where this is no longer a concern but it was a concern back then4. One thing Intel wanted to avoid was appending a 32 bit address (segment: offset) to the end of instructions that reference memory. They were able to reduce this to 16 bits (offset only) by making certain assumptions about which segments in memory an instruction could access.

The 8086 through 80286 processors have four segment registers: cs, ds, ss and es. The 80386 and later processors have these segment registers plus fs and gs. The cs (code segment) register points at the segment containing the currently executing code. The CPU always fetches instructions from the address given by cs:ip. By default, the CPU expects to access most variables in the data segment. Certain variables and other operations occur in the stack segment. When accessing data in these speciÞc areas, no segment value is necessary.

To access data in one of the extra segments (es, fs, or gs), only a single byte is necessary to choose the appropriate segment register. Only a few control transfer instructions allow you to specify a full 32 bit segmented address.

Now, this might seem rather limiting. After all, with only four segment registers on the 8086 you can address a maximum of 256 Kilobytes (64K per segment), not the full megabyte promised. However, you can change the segment registers under program control, so it is possible to address any byte by changing the value in a segment register.

Of course, it takes a couple of instructions to change the value of one of the 80x86’s segment registers. These instructions consume memory and take time to execute. So saving two bytes per memory access would not pay off if you are accessing data in different segments all the time. Fortunately, most consecutive memory accesses occur in the same segment. Hence, loading segment registers isn’t something you do very often.

* + 1. **The 80x86 Addressing Modes**

Like the x86 processors described in the previous chapter, the 80x86 processors let you access memory in many different ways. 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 Þrst 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; the new modes are just an added bonus. If you need to write code that works on 80286 and earlier processors, you will not be able to take advantage of these new modes.

However, if you intend to run your code on 80386sx or higher processors, you can use these new modes. Since many programmers still need to write programs that run on 80286 and earlier machines5, it’s important to separate the discussion of these two sets of addressing modes to avoid confusing them.

* + - 1. **8086 Register Addressing Modes**

Most 8086 instructions can operate on the 8086’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 8086 mov (move) instruction:

mov destination, source

This instruction copies the data from the source operand to the destination operand. The eight and 16 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 8086 mov instructions:

mov ax, bx ;Copies the value from BX into AX

mov dl, al ;Copies the value from AL into DL

mov si, dx ;Copies the value from DX into SI

mov sp, bp ;Copies the value from BP into SP

mov dh, cl ;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 8086’s general purpose registers.

In addition to the general purpose registers, many 8086 instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, you may not specify cs as the destination operand, second, only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of cs to ds, you’d have to use some sequence like:

mov ax, cs

mov ds, ax

You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses. But more on that, later. Throughout this text you’ll see the abbreviated operand *sreg* used wherever segment register operands are allowed (or required).

* + - 1. **8086 Memory Addressing Modes**

The 8086 provides 17 different ways to access memory. This may seem like quite a bit at first, but fortunately most of the address 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 8086 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed.

Variations on these five forms provide the 17 different addressing modes on the 8086. See, from 17 down to five. It’s not so bad after all!

* + - 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 16 bit constant that specifies the address of the target location. The instruction mov al, ds:[8088h] loads the al register with a copy of the byte at memory loca-

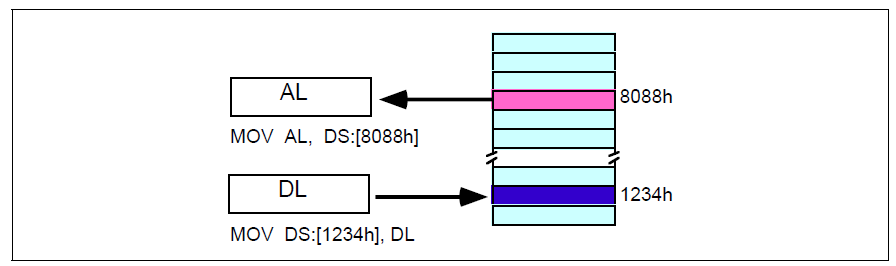


Figure 2.8 Displacement Only (Direct) Addressing Mode

tion 8088h7. Likewise, the instruction mov ds:[1234h], dl stores the value in the dl register to memory location 1234h (see Figure 2.8)

The displacement-only addressing mode is perfect for accessing simple variables. Of course, you’d probably prefer using names like ‘I’ or ‘J’ rather than ‘DS:[1234h]’ or ‘DS:[8088h]’. Well, fear not, you’ll soon see it’s possible to do just that.

Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the mov opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors (see the previous chapter). There are some minor differences, however. First of all, a displacement is exactly that- some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). Don’t worry if this doesn’t make a lot of sense right now. You’ll get an opportunity to study segments to your heart’s content a little later in this chapter. For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this chapter will typically access bytes in memory. Don’t forget, however, that you can also access words on the 8086 processors.

By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a segment override prefix before your address. For example, to access location 1234h in the extra segment (es) you would use an instruction of the form mov ax,es:[1234h]. Likewise, to access this location in the code segment you would use the instruction mov ax, cs:[1234h]. The ds: prefix in the previous examples is not a segment override. The CPU uses the data segment register by default. These specific examples require ds: because of MASM’s syntactical limitations.

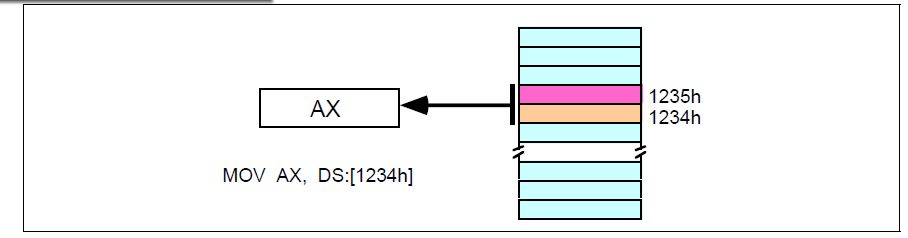


Figure 2.9 Accessing a Word

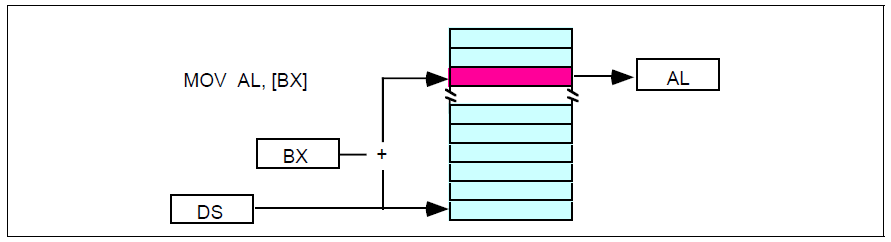


Figure 2.10 [BX] Addressing Mode

* + - 1. **The Register Indirect Addressing Modes**

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

mov al, [bx]

mov al, [bp]

mov al, [si]

mov al, [di]

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the ds segment by default. The [bp] addressing mode uses the stack segment (ss) by default.

You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

mov al, cs:[bx]

mov al, ds:[bp]

mov al, ss:[si]

mov al, es:[di]

Intel refers to [bx] and [bp] as base addressing modes and bx and bp as base registers (in fact, bp stands for base pointer). Intel refers to the [si] and [di] addressing modes as indexed addressing modes (si stands for source index, di stands for destination index). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.

Note: the [si] and [di] addressing modes work exactly the same way, just substitute si and di for bx above.

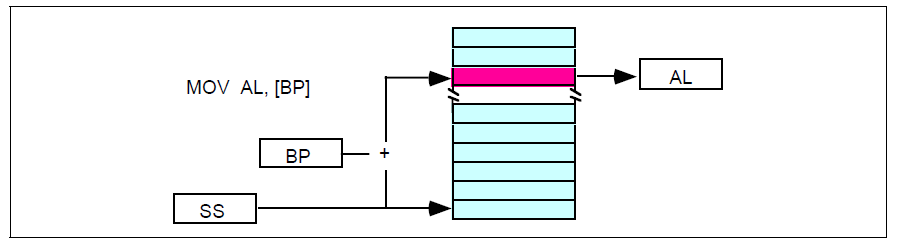


Figure 2.11 [BP] Addressing Mode

* + - 1. **Indexed Addressing Modes**

The indexed addressing modes use the following syntax:

mov al, disp[bx]

mov al, disp[bp]

mov al, disp[si]

mov al, disp[di]

If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the speciÞed register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

mov al, ss:disp[bx]

mov al, es:disp[bp]

mov al, cs:disp[si]

mov al, ss:disp[di]

You may substitute si or di in Figure 2.12 to obtain the [si+disp] and [di+disp] addressing modes. Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel’s literature does not differentiate between these modes with or without the constant. If you look at how the hardware works, this is a reasonable definition. From the programmer’s point of view, however, these addressing modes are useful for entirely

Based vs. Indexed Addressing

There is actually a subtle difference between the based and indexed addressing modes. Both addressing modes consist of a displacement added together with a register. The major difference between the two is the relative sizes of the displacement and register values. In the indexed addressing mode, the constant typically provides the address of the specific data structure and the register provides an offset from that address. In the based addressing mode, the register contains the address of the data structure and the constant displacement supplies the index from that point.

Since addition is commutative, the two views are essentially equivalent. However, since Intel supports one and two byte displacements. It made more sense for them to call it the based addressing mode. In actual use, however, you’ll wind up using it as an indexed addressing mode more often than as a based addressing mode, hence the name change.

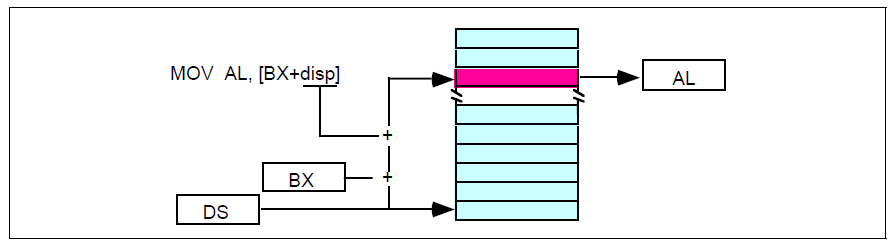


Figure 2.12 [BX+disp] Addressing Mode

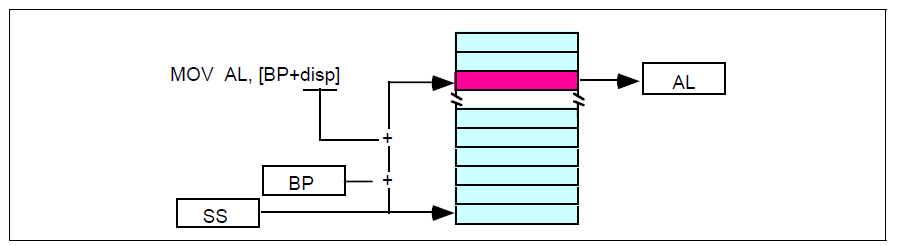


Figure 2.13 [BP+disp] Addressing Mode

different things. Which is why this handout uses different terms to describe them. Unfortunately, there is very little consensus on the use of these terms in the 80x86 world.

* + - 1. **Based Indexed Addressing Modes**

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

mov al, [bx][si]

mov al, [bx][di]

mov al, [bp][si]

mov al, [bp][di]

Suppose that bx contains 1000h and si contains 880h. Then the instruction

mov al,[bx][si]

would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004,

mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D.

The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.

You substitute di in Figure 2.12 to obtain the [bx+di] addressing mode. You substitute di in Figure 2.12 for the [bp+di] addressing mode.

* + - 1. **Based Indexed Plus Displacement Addressing Mode**

These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes (see Figure 2.12 and Figure 2.12).

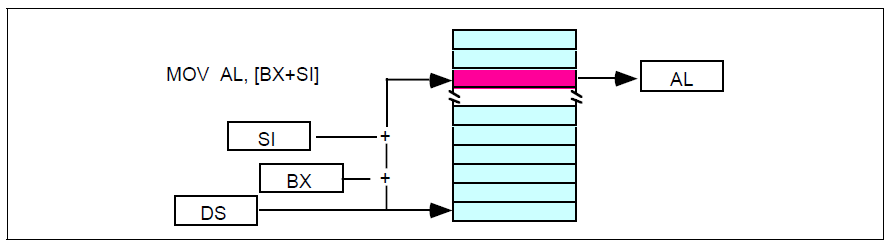


Figure 2.14 [BX+SI] Addressing Mode

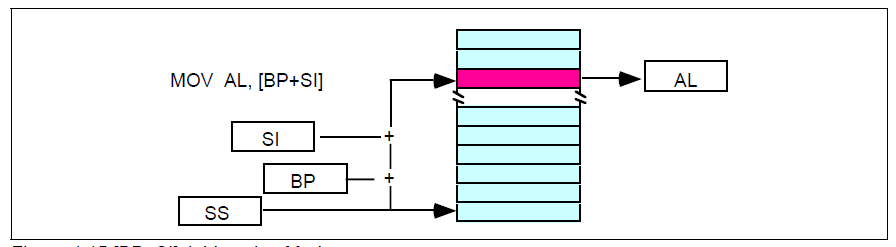


Figure 2.15 [BP+SI] Addressing Mode

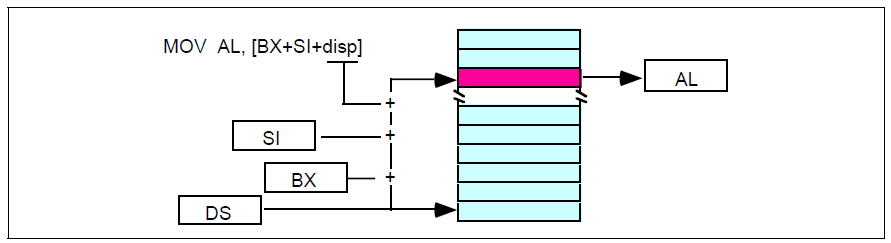


Figure 2.16 [BX + SI + disp] Addressing Mode

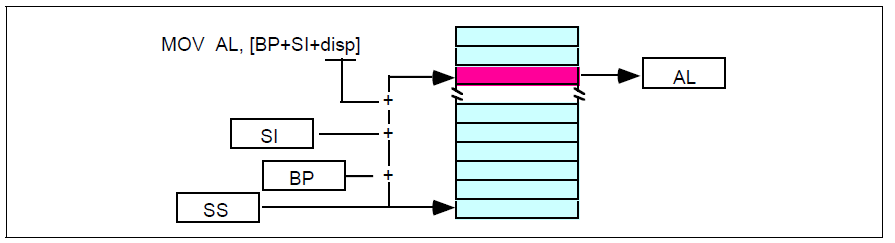


Figure 2.17 [BP + SI + disp] Addressing Mode

mov al, disp[bx][si]

mov al, disp[bx+di]

mov al, [bp+si+disp]

mov al, [bp][di][disp]

You may substitute di in Figure 2.12 to produce the [bx+di+disp] addressing mode. You may

substitute di in Figure 2.12 to produce the [bp+di+disp] addressing mode.

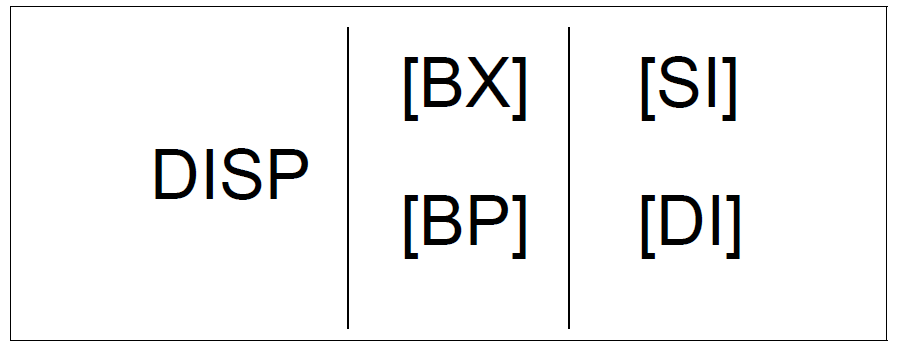


Figure 2.18 Table to Generate Valid 8086 Addressing Modes

Suppose bp contains 1000h, bx contains 2000h, si contains 120h, and di contains 5. Then mov al,10h[bx+si] loads al from address DS:2130; mov ch,125h[bp+di] loads ch from location SS:112A; and mov bx,cs:2[bx][di] loads bx from location CS:2007.

* + - 1. **An Easy Way to Remember the 8086 Memory Addressing Modes**

There are a total of 17 different legal memory addressing modes on the 8086: disp, [bx], [bp], [si], [di], disp[bx], disp[bp], disp[si], disp[di], [bx][si], [bx][di], [bp][si], [bp][di], disp[bx][si], disp [bx][di], disp[bp][si], and disp[bp][di]9. You could memorize all these forms so that you know which are valid (and, by omission, which forms are invalid).

However, there is an easier way besides memorizing these 17 forms. Consider the chart in Figure 2.12.

If you choose zero or one items from each of the columns and wind up with at least one item, you’ve got a valid 8086 memory addressing mode. Some examples:

* Choose disp from column one, nothing from column two, [di] from column 3, you get disp[di].
* Choose disp, [bx], and [di]. You get disp[bx][di].
* Skip column one & two, choose [si]. You get [si]
* Skip column one, choose [bx], then choose [di]. You get [bx][di]

Likewise, if you have an addressing mode that you *cannot* construct from this table, then it is not legal. For example, disp[dx][si] is illegal because you cannot obtain [dx] from any of the columns above.

* + - 1. **Some Final Comments About 8086 Addressing Modes**

The *effective address* is the final offset produced by an addressing mode computation. For example, if bx contains 10h, the effective address for 10h[bx] is 20h. You will see the term effective address in almost any discussion of the 8086’s addressing mode. There is even a special instruction *load effective address* (lea) that computes effective addresses.

Not all addressing modes are created equal! Different addressing modes may take differing amounts of time to compute the effective address. The exact difference varies from processor to processor. Generally, though, the more complex an addressing mode is, the longer it takes to compute the effective address. Complexity of an addressing mode is directly related to the number of terms in the addressing mode. For example, disp[bx][si] is 9. That’s not even counting the syntactical variations! more complex than [bx]. See the instruction set reference in the appendices for information regarding the cycle times of various addressing modes on the different 80x86 processors.

The displacement field in all addressing modes *except* displacement-only can be a signed eight bit constant or a signed 16 bit constant. If your offset is in the range -128É+127 the instruction will be shorter (and therefore faster) than an instruction with a displacement outside that range. The size of the value in the register does not affect the execution time or size. So if you can arrange to put a large number in the register(s) and use a small displacement, that is preferable over a large constant and small values in the register(s).

If the effective address calculation produces a value greater than 0FFFFh, the CPU ignores the overflow and the result *wraps around* back to zero. For example, if bx contains 10h, then the instruction mov al,0FFFFh[bx] will load the al register from location ds:0Fh, not from location ds:1000Fh.

In this discussion you’ve seen how these addressing modes operate. The preceding discussion didn’t explain *what you use them for*. That will come a little later. As long as you know how each addressing mode performs its effective address calculation, you’ll be fine.

* + 1. **The 80x86 MOV Instruction**

The examples throughout this chapter will make extensive use of the 80x86 mov (move) instruction. Furthermore, the mov instruction is the most common 80x86 machine instruction. Therefore, it’s worthwhile to spend a few moments discussing the operation of this instruction. Like it’s x86 counterpart, the mov instruction is very simple. It takes the form:

mov Dest,Source

Mov makes a copy of *Source* and stores this value into *Dest*. This instruction does not affect the original contents of *Source.* It overwrites the previous value in *Dest*. For the most part, the operation of this instruction is completely described by the C++ statement:

Dest = Source;

This instruction has many limitations. You’ll get ample opportunity to deal with them throughout your study of 80x86 assembly language. To understand why these limitations exist, you’re going to have to take a look at the machine code for the various forms of this instruction. One word of warning, they don’t call the 80386 a CISC (Complex Instruction Set Computer) for nothing. The encoding for the mov instruction is probably the most complex in the instruction set. Nonetheless, without studying the machine code for this instruction you will not be able to appreciate it, nor will you have a good understanding of how to write optimal code using this instruction. You’ll see why you worked with the x86 processors in the previous chapters rather than using actual 80x86 instructions.

There are several versions of the mov instruction. The mnemonic (Mnemonic means memory aid. This term describes the English names for instructions like MOV, ADD, SUB,

etc., which are much easier to remember than the hexadecimal encodings for the machine instructions.) mov describes over a dozen different instructions on the 80386. The most commonly used form of the mov instruction has the following binary encoding shown in Figure 2.19.

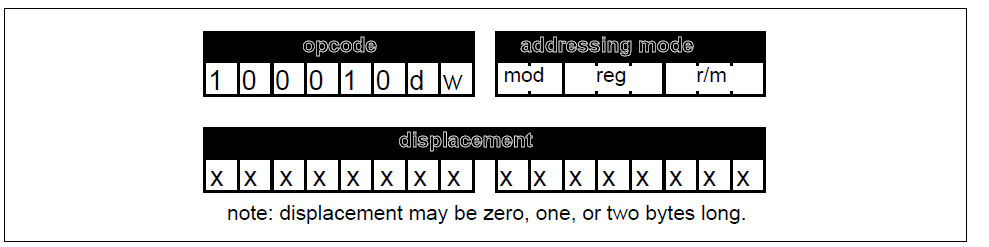
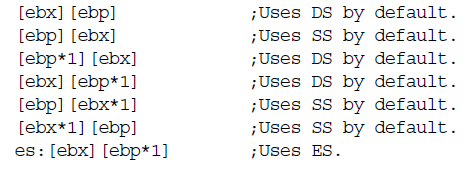


Figure 2.19 Generic MOV Instruction



The opcode is the first eight bits of the instruction. Bits zero and one define the *width* of the instruction (8, 16, or 32 bits) and the *direction* of the transfer. When discussing specific instructions this text will always Þll in the values of *d* and *w* for you. They appear here only because almost every other text on this subject requires that *you* Þll in these values.

Following the opcode is the addressing mode byte, affectionately called the ‘mod-reg-r/m’ byte by most programmers. This byte chooses which of 256 different posisible operand combinations the generic mov instruction allows. The generic mov instruction takes three different assembly language forms:

mov reg, memory

mov memory, reg

mov reg, reg

Note that at least one of the operands is always a general purpose register. The *reg* field in the mod/reg/rm byte speciÞes that register (or one of the registers if using the third form above). The *d* (direction) bit in the opcode decides whether the instruction stores data into the register (d=1) or into memory (d=0).

**Some Final Comments on the MOV Instructions**

There are several important facts you should always remember about the mov instruction.

First of all, *there are no memory to memory moves*. For some reason, newcomers to assembly language have a hard time grasping this point. While there are a couple of instructions that perform memory to memory moves, loading a register and then storing that register is almost always more efficient. Another important fact to remember about the mov instruction is that there are many different mov instructions that accomplish the same thing. Likewise, there are several different addressing modes you can use to access the same memory location. If you are interested in writing the shortest and fastest possible programs in assembly language, you must be constantly aware of the trade-offs between equivalent instructions.

The discussion in this chapter deals mainly with the generic mov instruction so you can see how the 80x86 processors encode the memory and register addressing modes into the mov instruction. Other forms of the mov instruction let you transfer data between 16-bit general purpose registers and the 80x86 segment registers. Others let you load a register or memory location with a constant. These variants of the mov instruction use a different opcode

There are several additional mov instructions on the 80386 that let you load the 80386 special purpose registers. This text will not consider them. There are also some string instructions on the 80x86 that perform memory to memory operations. They are not a good substitute for the mov instruction.