16.1 Why Study the 80x86?

Any introduction to processor architecture should be followed by an investigation of the architecture of a specific processor. The choice then becomes which processor to examine. There are so many. Some approaches use a virtual processor, i.e., one that exists only on paper or as a simulator. This method simplifies the learning process by concealing the complexities and idiosyncrasies of a real processor.

At the other extreme, we could examine a modern processor such as the Intel® Pentium® 4 Processor Extreme Edition with its Hyper-Threading Technology™, Hyper-Pipelined Technology™, enhanced branch prediction, three levels of 8-way cache including a split L1 cache, and multiple ALUs. Or we could look at the Apple® PowerPC® G5 with its 64-bit architecture, two double-precision floating point units, and twelve functional units. If you are a student who has just been introduced to processor architecture, this can be like trying to swallow an elephant. Too many new concepts must be explained before even a minimal understanding of the processor can be had.

A third method is to examine the simplest processor from a family of existing processors. This particular processor should provide the closest match to the processor architecture discussed in Chapter 15 while providing a link to the most modern processor of the family. It eliminates the need for a discussion of advanced computer architecture concepts while giving the student a real processor that they can program.

The processor we present here is the original 16-bit Intel processor, the 80186, the root of the Intel processor family that is commonly referred to as the 80x86 family. The 'x' in 80x86 represents the generation of the processor, 1, 2, 3, and so on. Table 16-1 presents a summary of the bus characteristics of some of the 80x86 processors.

The 80186 has 16 data lines allowing it to perform operations on unsigned integers from 0 to $2^{16} – 1 = 65,535$ and signed integers from –32,768 to 32767. It has 20 address lines providing access to a memory space of $2^{20} = 1$ Meg.

Table 16-1 Summary of Intel 80x86 Bus Characteristics

Processor	Data bus width	Address bus width	Size of address space
80186	16	20	2^{20} = 1 Meg
80286	16	24	2^{24} = 16 Meg
80386SX	16	24	2^{24} = 16 Meg
80386DX	32	32	2^{32} = 4 Gig
80486	32	32	2^{32} = 4 Gig
80586 "Pentium"	64	32	2^{32} = 4 Gig

16.2 Execution Unit

The 80x86 processor is divided into two main components: the execution unit and the bus interface unit. The ***execution unit*** (EU) is the 80x86's CPU as discussed in Chapter 15. It is controlled by the EU control system which serves a dual purpose: it acts as the control unit and also as a portion of the instruction decoder. The EU also contains the ALU, the processor flags, and the general purpose and address registers. Figure 16-1 presents a block diagram of the EU.

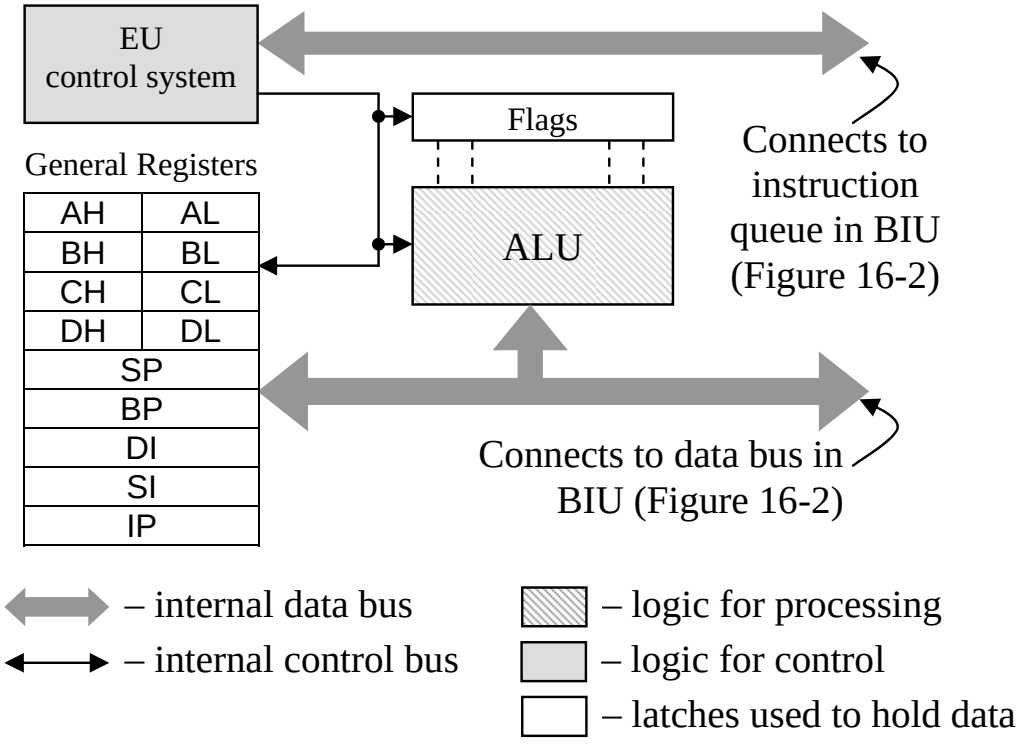


Figure 16-1 Block Diagram of 80x86 Execution Unit (EU)

16.2.1 General Purpose Registers

The registers of the 80x86 are grouped into two categories: general purpose and address. The general purpose registers are for manipulating or transferring data; the address registers contain memory addresses and are used to point to the locations in memory where data will be retrieved or stored.

Figure 16-1 shows that there are eight general purpose registers: AH, AL, BH, BL, CH, CL, DH, and DL. Each of these registers is eight bits. Earlier we said that the 80186 is a 16-bit processor. How can this be since we only have 8-bit registers?

The 80186 processor creates a 16-bit register by combining two 8-bit registers. AH and AL, for example, can function as a pair. This larger register is referred to as AX. The 8-bit registers are combined by linking them together so that the 8 bits of AH are the 8 most significant bits of AX and AL are the 8 least significant bits of AX. For example, if AH contains $10110000_2 = B0_{16}$ and AL contains $01011111_2 = 5F_{16}$, then the virtual register AX contains $1011000001011111_2 = B05F_{16}$.

Example

If CX contains the binary value 0110110101101011_2, what value does CH have?

Solution

Since the register CH provides the most significant 8 bits of CX, then the upper eight bits of CX is CH, i.e., CH contains 01101101_2.

Each of the general purpose registers is named according to their default purpose. For the most part, these purposes are not set in stone. The programmer still has some flexibility in how the registers are used. The following discussion presents their suggested use.

AX is called the ***accumulator register***, and it is used mostly for arithmetic, logic, and the general transfer of data. Many of the assembly language instructions for higher level mathematical operations such as multiply and divide don't even let the programmer specify a register other than AX to be used.

BX is called the ***base register***, and it is used as a base address or pointer to things like data arrays. We will find out later that there are a number of other registers that are used as pointers, but those are special purpose pointers. BX tends to be more of a general purpose pointer.

CX is called the ***counter register***. When a programmer uses a for-loop, the index for that loop is usually stored in CX. Intel designed a number of special purpose instructions that use CX in order to get better performance out of loops.

DX is called the ***data register***. This register is used with AX for special arithmetic functions allowing for things such as storing the upper half of a 32-bit result of a 16-bit multiply or holding the remainder after an integer division.

16.2.2 Address Registers

Below the general purpose registers in Figure 16-1 are the address registers: SP, BP, DI, SI, and IP. These are 16-bit registers meant to contain addresses with which to point to locations in memory. At this point, do not worry about how a 16-bit register can reference something in a memory space that uses a 20-bit address bus. The process involves using the segment registers of the BIU. We will address the mechanics behind the use of the segment registers later in this chapter.

These address registers are classified into two groups: the pointer registers, SP, BP, and IP, and the index registers, DI and SI. Although they all operate in the same manner, i.e., pointing to addresses in memory, each address register has a specific purpose.

SP is the ***stack pointer*** and it points to the address of the last piece of data stored to the stack. To store something to the stack, the stack pointer is decremented by the size of the value to be stored, i.e., SP is decremented by 2 for a word or 4 for a double word. The value is then stored at the new address pointed to by the stack pointer. To retrieve a value from the stack, the value is read from the address pointed to by the stack pointer, then the stack pointer is incremented accordingly.

BP is the ***base pointer*** and its primary use is to point to the parameters that are passed to a function during a function call. For example, if the function myfunc(var1, var2) is called, the values for var1 and var2 are placed in the temporary memory of the stack. BP contains the address in the stack where the list of variables begins.

IP is the ***instruction pointer***. As we discussed in Chapter 15, the CPU goes step-by-step through memory loading, interpreting, and then executing machine code. It uses the memory address contained in IP as a marker pointing to where to retrieve the next instruction. Each time it retrieves an instruction, it increments IP so that it points to the next instruction to retrieve. In some cases, the instruction decoder needs to

increment IP multiple times to account for data or operands that might follow an element of machine code.

SI, the ***source index***, and ***DI***, the ***destination index***, also contain addresses that point to memory. They are used for string operations where strings may be copied, searched, or otherwise manipulated. SI points to memory locations from which characters are to be retrieved while DI points to memory locations where characters will be stored.

16.2.3 Flags

The flags of the 80x86 processor are contained in a 16-bit register. Not all 16 bits are used, and it isn't important to remember the exact bit positions of each of the flags inside the register. The important thing is to understand the purpose of each of the flags.

Remember from Chapter 15 that the flags indicate the current status of the processor. Of these, the majority report the results of the last executed instruction to affect the flags. (Not all instructions affect all the flags.) These flags are then used by a set of instructions that test their state and alter the flow of the software based on the result.

The flags of the 80x86 processor are divided into two categories: control flags and status flags. The control flags are modified by the software to change how the processor operates. There are three of them: trap, direction, and interrupt.

The ***trap flag (TF)*** is used for debugging purposes and allows code to be executed one instruction at a time. This allows the programmer to step through code address-by-address so that the results of each instruction can be inspected for proper operation.

The ***direction flag (DF)*** is associated with string operations. In particular, DF dictates whether a string is to be examined by incrementing through the characters or decrementing. This flag is used by the 80x86 instructions that automate string operations.

Chapter 15 introduced us to the concept of interrupts by showing how devices that need the processor's attention can send a signal interrupting the processor's operation in order to avoid missing critical data. The ***interrupt flag (IF)*** is used to enable or disable this function. When this flag contains a one, any interrupt that occurs is serviced by the processor. When this flag contains a zero, the maskable interrupts are ignored by the processor, their requests for service remaining in a queue waiting for the flag to return to a one.

The IF flag is cleared and set by software using two different assembly language commands: **STI** for setting and **CLI** for clearing. Some interrupts known as non-maskable interrupts cannot be disabled. Either their purpose is considered to be a priority over all other processor functions or the software itself calls the interrupt.

The remaining flags are the status flags. These are set or cleared based on the result of the last executed instruction. There are six of them: overflow, sign, zero, auxiliary carry, parity, and carry. The following describes the operation of each of these bits.

- ***Overflow flag (OF)*** – indicates when an overflow has occurred in a mathematical operation.
- ***Sign flag (SF)*** – follows the sign bit of a mathematical or logical result, i.e., it is cleared to 0 when the result is positive and set to 1 when the result is negative.
- ***Zero flag (ZF)*** – is set to 1 when the result of a mathematical or logical function is zero. The flag is cleared to 0 otherwise.
- ***Auxiliary carry flag (AF)*** – equals the carry from the bit 3 column of an addition into the bit 4 column. If you recall the section on BCD addition from Chapter 3, a carry out of a nibble is one indication that error correction must be taken. This flag represents the carry out of the least significant nibble.
- ***Parity flag (PF)*** – is set to 1 if the result contains an even number of ones and cleared to 0 otherwise.
- ***Carry flag (CF)*** – represents the carry out of the most significant bit position. Some shift operations also use the carry to hold the bit that was last shifted out of a register.

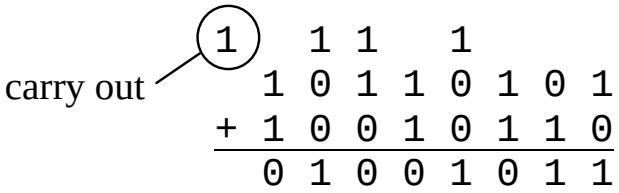
Example

How would the status flags be set after the processor performed the 8-bit addition of 10110101_2 and 10010110_2?

Solution

This problem assumes that the addition affects all of the flags. This is not true for all assembly language instructions, i.e., a logical OR does not affect AF.

Let's begin by adding the two numbers to see what the result is.



Now go through each of the flags to see how it is affected.

OF=1 – There was an overflow, i.e., adding two negative numbers resulted in a positive number.
SF=0 – The result is positive.
ZF=0 – The result does not equal zero.
AF=0 – No carry occurred from the fourth column (bit 3) to the fifth column (bit 4).
PF=1 – The result contains four ones which is an even number.
CF=1 – There was a carry.

16.2.4 Internal Buses

There are two internal buses in the EU that are used to pass information between the components. The first is used to exchange data and addressing information between the registers and the ALU. This same bus is also used to transfer data to and from memory by way of the bus interface unit. Each assembly language instruction that uses operands must move those operands from their source to a destination. These transfers occur along the data bus.

The second bus has one purpose: to transfer instructions that have been obtained by the bus interface unit to the instruction decoder contained in the EU control system.

The next section discusses how the bus interface unit performs data transactions with the memory space.

16.3 Bus Interface Unit

The ***bus interface unit*** (BIU) controls the transfer of information between the processor and the external devices such as memory, I/O ports, and storage devices. Basically, it acts as the bridge between the EU and the external bus. A portion of the instruction decoder as defined in Chapter 15 is located in the BIU. The instruction queue acts as a buffer allowing instructions to be queued up as they wait for their turn in the EU. Figure 16-2 presents the block diagram of the BIU.

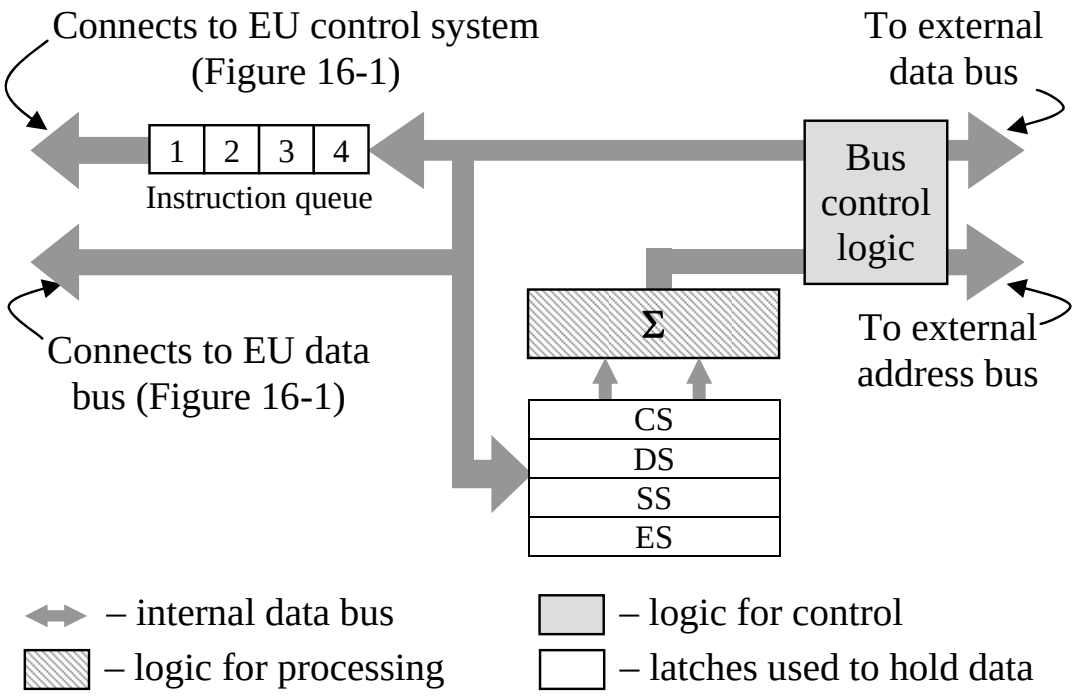


Figure 16-2 Block Diagram of 80x86 Bus Interface Unit (BIU)

The main purpose of the BIU is to take the 16-bit pointers of the EU and modify them so that they can point to data in the 20-bit address space. This is done using the four registers CS, DS, SS, and ES. These are the segment registers.

16.3.1 Segment Addressing

In the center of the BIU block diagram is a set of segment registers labeled CS, DS, SS, and ES. These four 16-bit registers are used in conjunction with the pointer and index registers to store and retrieve items from the memory space.

So how does the processor combine a 16-bit address register with a 16-bit segment register to create a 20-bit address? Well, it is all done in the address summing block located directly above the segment registers in the block diagram of the BIU in Figure 16-2. Every time the processor goes out to its memory space to read or write data, this 20-bit address must be calculated based on different combinations of address and segment registers.

Next time your Intel-based operating system throws up an execution error, look to see if it gives you the address where the error occurred. If it does, you should see some hexadecimal numbers in a format similar to the one shown below:

3241:A34E

This number is a special representation of the segment register (the number to the left of the colon) and the pointer or index register (the number to the right of the colon). Remember that a 4-digit hexadecimal number represents a 16-bit binary number. It is the combination of these two 16-bit registers that creates the 20-bit address.

The process works like this. First take the value in the segment register and shift if left four places. This has the effect of adding a zero to the right side of the hexadecimal number or four zeros to the right side of the binary number. In our example above, the segment is 3241_{16} = 0011 0010 0100 0001_2. Adding a zero nibble to the right side of the segment gives us 32410_{16} = 0011 0010 0100 0001 0000_2.

The pointer or index register is then added to this 20-bit segment address. Continuing our example gives us:

```
 0011 0010 0100 0001 0000          32410₁₆
+     1010 0011 0100 1110   or   + A34E₁₆
-------------------------------   ---------
 0011 1100 0111 0101 1110          3C75E₁₆
```

For the rest of this book, we will use the following terminology to represent these three values.

- The 20-bit value created by shifting the value in a segment register four places to the left will be referred to as the ***segment address***. It points to the lowest address to which a segment:pointer combination can point. This address may also be referred to as the ***base address*** of the segment.
- The 16-bit value stored in a pointer or index register will be referred to as the ***offset address***. It represents an offset from the segment address to the address in memory that the processor needs to communicate with.
- The resulting 20-bit value that comes out of the address summing block points to a specific address in the processor's memory space.

This address will be referred to as the ***physical address***, and it is the address that is placed on the address lines of the memory bus.

If we look at the function of the segment and pointer registers from the perspective of the memory space, the segment register adjusted with four binary zeros filled in from the right points to an address somewhere in the full memory space. Because the least significant four bits are always zero, this value can only point to memory in 16-byte increments. The 16-bit offset address from the pointer register is then added to the segment address pointing to an address within the 2^{16} = 65,535 (64K) locations above where the segment register is pointing. This is the physical address. Figure 16-3 shows how the segment and pointer addresses relate to each other when pointing to a specific address within the memory space.

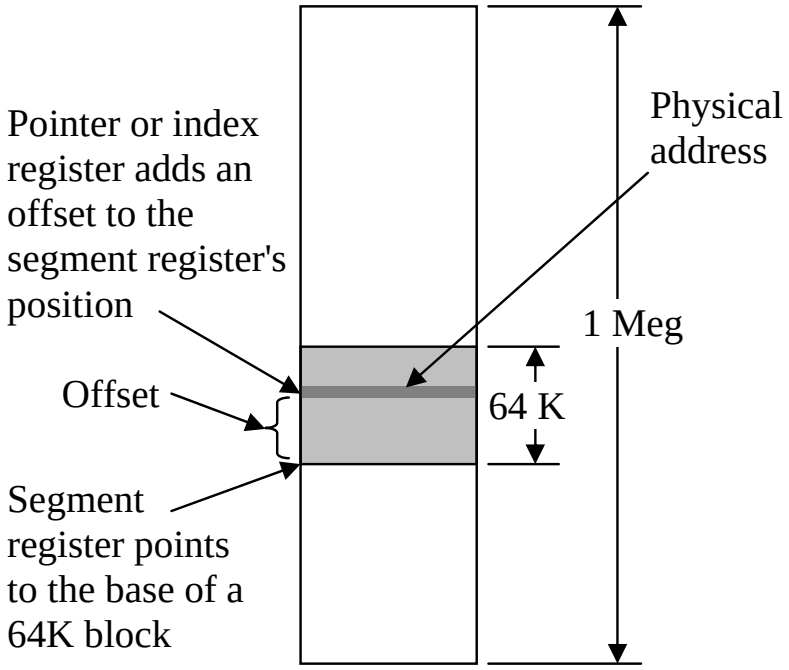


Figure 16-3 Segment/Pointer Relation in the 80x86 Memory Map

There is a second purpose for this segment:pointer addressing method beyond allowing the 80x86 processor to control 20 address lines using 16-bit registers. This second reason is actually of greater importance as it allows for greater functionality of the operating system.

By assigning the responsibility of maintaining the segment registers to the operating system while allowing the application to control the address and pointer registers, applications can be placed anywhere in memory without affecting their operation. When the operating system loads an application to be executed, it selects a 64 K block of memory called a ***segment*** and uses the lowest address of that block as the base address for that particular application. During execution, the application modifies only the pointer registers keeping its scope within the 64K block of its segment.

As long as the application modifies only the address registers, then the program remains in the 64 K segment it was assigned to. By using this process, the operating system is free to place an application wherever it wants to in memory. It also allows the operating system to maintain several concurrent applications in memory by keeping track of which application is assigned to which segment.

Although the programmer may force a segment register to be used for a different purpose, each segment register has an assigned purpose. The following describes the uses of the four segment registers, CS, DS, SS, and ES.

- ***Code Segment (CS)*** – This register contains the base address of the segment assigned to contain the code of an application. It is paired with the Instruction Pointer (IP) to point to the next instruction to load into the instruction decoder for execution.
- ***Data Segment (DS)*** – This register contains the base address of the segment assigned to contain the data used by an application. It is typically associated with the SI register.
- ***Stack Segment (SS)*** – This register contains the base address of the stack segment. Remember that there are two pointer registers that use the stack. The first is the stack pointer, and the combination of SS and SP points to the last value stored in this temporary memory. The other register is the base pointer which is used to point to the block of data elements passed to a function.
- ***Extra Segment (ES)*** – Like DS, this register points to the data segment assigned to an application. Where DS is associated with the SI register, ES is associated with the DI register.

Example

If CS contains $A487_{16}$ and IP contains 1436_{16}, then what is the physical address of the next instruction in memory to be executed?

Solution

The physical address is found by shifting $A487_{16}$ left four bits and adding 1436_{16} to the result.

$$\begin{array}{r} A4870_{16} \\ +\ 1436_{16} \\ \hline A5CA6_{16} \end{array} \quad \text{or} \quad \begin{array}{r} 1010\ 0100\ 1000\ 0111\ 0000 \\ +\quad\ 0001\ 0100\ 0011\ 0110 \\ \hline 1010\ 0101\ 1100\ 1010\ 0110 \end{array}$$

Therefore, the physical address pointed to by A487:1436 is $A5CA6_{16}$.

16.3.2 Instruction Queue

As discussed in Chapter 15, there are times during the execution of an instruction when different portions of the processor are idle. In the case of the 80x86 processor for example, while the BIU is retrieving the next instruction to be executed from memory, the EU control system and the ALU are standing by waiting for the instruction.

The 80186 divides the process of executing an instruction into three cycles: fetch, decode, and execute. These cycles are described below:

- ***Fetch*** – Retrieve the next instruction to execute from its location in memory. This is taken care of by the BIU.
- ***Decode*** – Determine which circuits to energize in order to execute the fetched instruction. This function is performed by the instruction decoding circuitry in the EU control system.
- ***Execute*** – Perform the operation dictated by the instruction using the ALU, registers, and data transfer mechanisms.

The purpose of the instruction queue of the BIU is to maintain a sequence of fetched instructions for the EU to execute. In some cases, branches or returns from functions can disrupt the sequence of instructions and require a change in the anticipated order of execution. An advanced instruction queue can handle this by loading both paths of execution allowing the EU to determine which one it will need after executing the previous instructions.

16.4 Memory versus I/O Ports

In order to communicate with external hardware devices without taking up space in the 1 Meg memory space of the 80x86 processor, two additional control lines are added to the bus that effectively turn it into two buses, one for data and one for I/O. This second bus uses the same address and data lines that are used by the memory bus. The difference is that the I/O devices use different read and write control lines.

To read data from memory, the 80x86 processor uses the active-low signal MRDC. When MRDC is low, the addressed memory device on the bus knows to pass the appropriate data back to the processor.

To write data to memory, the 80x86 processor uses the active-low signal MWTC. When MWTC is low, the addressed memory device on the bus knows that the processor will be sending data to it. Once the memory device receives this data, it knows to store it in the appropriate memory location.

If both MRDC and MWTC are high, then the memory devices remain inactive. By adding a second pair of read and write control lines, the processor can communicate with a new set of devices on the same set of address and data lines. These new devices are called ***I/O ports***, and they connect the processor to the external environment. By placing an address on the address lines, an I/O port is selected in the same way that a memory chip is selected using chip select circuitry.

The read control for the I/O ports is called IORC, and it too is an active low signal. When IORC equals zero, the selected I/O port places data on the data lines for the processor to read. This data might be the value of a key press, the digital value of an analog input, the status of a printer, or anything else that the processor needs to input from the external devices.

The write control for the I/O ports is called IOWC. This active low signal goes low when the processor wants to send data to an external device. This data might be the characters of a document to be printed, a command to the video system, or any other value that the processor needs to send to the external devices.

Table 16-2 summarizes the settings of these four read and write control signals based on their functions.

Table 16-2 Summary of the 80x86 Read and Write Control Signals

Function	MRDC	MWTC	IORC	IOWC
Reading from memory	0	1	1	1
Writing to memory	1	0	1	1
Reading from an I/O device	1	1	0	1
Writing to an I/O device	1	1	1	0

Even though they use the same address and data lines, there are slight differences between the use of memory and the use of I/O ports. First, regardless of the generation of the 80x86 processor, only the lowest 16 address lines are used for I/O ports. This means that even if the memory space of an 80x86 processor goes to 4 Gig, the I/O port address space will always be 2^{16} = 65,536 = 64K. This is not a problem as the demand on the number of external devices that a processor needs to communicate with has not grown nearly at the rate of demand on memory space.

The second difference between the memory space and the I/O port address space is the requirement placed on the programmer. Although we have not yet discussed the 80x86 assembly language instruction set, the assembly language commands for transferring data between the registers and memory are of the form MOV. This command cannot be used for input or output to the I/O ports because it uses MRDC and MWTC for bus commands. To send data to the I/O ports, the assembly language commands OUT and OUTS are used while the commands for reading data from the I/O ports are IN and INS.

16.5 What's Next?

Now that you have a general idea of the architecture of the 80x86, we can begin programming with it. In Chapter 17, we will present some of the instructions from the 80x86 assembly language along with the format of the typical assembly language program. In addition, the syntax used to differentiate between registers, memory, and constants in 80x86 assembly language code will be presented. This information will then be used to take you though some sample programs.

Problems

Answer problems 1 though 7 using the following settings of the 80x86 processor registers.

$AX = 1234_{16}$	$BP = 1212_{16}$	$CS = A101_{16}$
$BX = 8721_{16}$	$SP = 3434_{16}$	$DS = B101_{16}$
$CX = 5678_{16}$	$DI = 5656_{16}$	$SS = C101_{16}$
$DX = 8765_{16}$	$IP = 7878_{16}$	$ES = D101_{16}$

1. What is the value in the register AL?
2. What is the value in the register CH?
3. What is the physical address pointed to by ES:DI?
4. What is the physical address of the next instruction to be executed in memory?
5. What is the physical address of the last data item to be stored in the stack?
6. Assuming a function has been called and the appropriate address and segment registers have been set, what is the physical address of the location of the function parameters in the stack?
7. What would the settings of the flags OF, SF, ZF, AF, PF, and CF be after the addition of BH to AL?
8. True or false: Every 80x86 assembly language instruction modifies the flags.
9. What is the purpose of the internal bus that connects the instruction queue in the BIU with the EU control system?
10. List the two benefits of segmented addressing.
11. What are the values of MRDC, MWTC, IORC, and IOWC when the processor is storing data to memory?
12. What are the values of MRDC, MWTC, IORC, and IOWC when the processor is reading data from a device on the I/O port bus?
13. What 80x86 assembly language commands are used to write data to a memory device on the I/O port bus?

14. On an 80486 processor with its 32 address lines, what is the maximum number of I/O ports it can address?