9

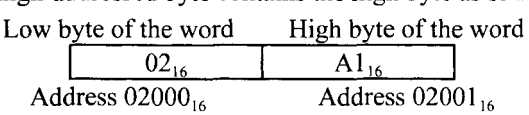
INTEL 8086

This chapter covers the Intel 8086 in detail. Intel's 32-bit microprocessors are based on the Intel 8086. Therefore, the 8086 provides an excellent educational tool for understanding Intel 32- and 64-bit microprocessors. Because the 8086 and its peripheral chips are inexpensive, the implementation costs of 8086-based systems are low. This makes the 8086 appropriate for thorough coverage in a first course on microprocessors. Thus, the 8086 is covered in detail in this chapter.

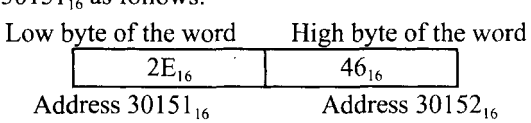
9.1 Introduction

The 8086 was Intel's first 16-bit microprocessor. This means that the 8086 has a 16-bit ALU. The 8086 contains 20 address pins. Therefore, it has a main (directly addressable) memory of one megabyte (2^{20} bytes).

The memory of an 8086-based microcomputer is organized as bytes. Each byte is uniquely addressed with 20-bit addresses of 00000_{16}, 00001_{16}, ... $FFFFF_{16}$. An 8086 word in memory consists of any two consecutive bytes; the low-addressed byte is the low byte of the word and the high-addressed byte contains the high byte as follows:



The 16-bit word at the even address 02000_{16} is $A102_{16}$. Next, consider a word stored at an address 30151_{16} as follows:



The 16-bit word stored at the odd address 30151_{16} is $462E_{16}$.

The 8086 always reads a 16-bit word from memory. This means that a word instruction accessing a word starting at an even address can perform its function with one memory read. A word instruction starting at an odd address, however, must perform two memory accesses to two consecutive memory even addresses, discarding the unwanted bytes of each. For byte read starting at odd address N, the byte at the previous even address N - 1 is also accessed but discarded. Similarly, for byte read starting at even address N, the byte with odd address N + 1 is also accessed but discarded.

For the 8086, register names followed by the letters X, H, or L in an instruction for data transfer between register and memory specify whether the transfer is 16-bit or 8-bit. For example, consider `MOV AX, [START]`. If the 20-bit address `START` is an even number such as 02212_{16}, then this instruction loads the low (AL) and high (AH) bytes of

the 8086 16-bit register AX with the contents of memory locations 02212_{16} and 02213_{16}, respectively, in a single access. Now, if START is an odd number such as 02213_{16}, then the MOV AX, [START] instruction loads AL and AH with the contents of memory locations 02213_{16} and 02214_{16}, respectively, in two accesses. The 8086 also accesses memory locations 02212_{16} and 02215_{16} but ignores their contents.

Next, consider MOV AL, [START]. If START is an even number such as 30156_{16}, then this instruction accesses both addresses, 30156_{16} and 30157_{16}, but loads AL with the contents of 30156_{16} and ignores the contents of 30157_{16}. However, if START is an odd number such as 30157_{16}, then MOV AL, [START] loads AL with the contents of 30157_{16}. In this case the 8086 also reads the contents of 30156_{16} but discards it.

The 8086 is packaged in a 40-pin chip. A single +5 V power supply is required. The clock input signal is generated by the 8284 clock generator/driver chip. Instruction execution times vary between 2 and 30 clock cycles.

There are four versions of the 8086. They are 8086, 8086-1, 8086-2, and 8086-4. There is no difference between the four versions other than the maximum allowed clock speeds. The 8086 can be operated from a maximum clock frequency of 5 MHz. The maximum clock frequencies of the 8086-1, 8086-2 and 8086-4 are 10 MHz, 8 MHz and 4 MHz, respectively.

The 8086 family consists of two types of 16-bit microprocessors, the 8086 and 8088. The main difference is how the processors communicate with the outside world. The 8088 has an 8-bit external data path to memory and I/O; the 8086 has a 16-bit external data path. This means that the 8088 will have to do two READ operations to read a 16-bit word from memory. Similarly, two write operations are required to write a 16-bit word into memory. In most other respects, the processors are identical. Note that the 8088 accesses memory in bytes. No alterations are needed to run software written for one microprocessor on the other. Because of similarities, only the 8086 will be considered here. The 8088 was used in designing IBM's first personal computer.

An 8086 can be configured as a small uniprocessor (minimum mode when the MN/$\overline{\text{MX}}$ pin is tied to HIGH) or as a multiprocessor system (maximum mode when the MN/$\overline{\text{MX}}$ pin is tied to LOW). In a given system, the MN/$\overline{\text{MX}}$ pin is permanently tied to either HIGH or LOW. Some of the 8086 pins have dual functions depending on the selection of the MN/$\overline{\text{MX}}$ pin level.

In the minimum mode (MN/$\overline{\text{MX}}$ pin HIGH), these pins transfer control signals directly to memory and I/O devices; in the maximum mode (MN/$\overline{\text{MX}}$ pin LOW), these same pins have different functions that facilitate multiprocessor systems. In the maximum mode, the control functions normally present in minimum mode are assumed by a support chip, the 8288 bus controller.

Due to technological advances, Intel introduced the high-performance 80186 and 80188, which are enhanced versions of the 8086 and 8088, respectively. The 8-MHz 80186/80188 provides two times greater throughput than the standard 5-MHz 8086/8088. Both have integrated several new peripheral functional units, such as a DMA controller, a 16-bit timer unit, and an interrupt controller unit, into a single chip. Just like the 8086 and 8088, the 80186 has a 16-bit data bus and the 80188 has an 8-bit data bus; otherwise, the architecture and instruction set of the 80186 and 80188 are identical. The 80186/80188 has an on-chip clock generator so that only an external crystal is required to generate the clock. The 80186/80188 can operate at either a 6- or an 8-MHz internal clock frequency. The crystal frequency is divided by 2 internally. In other words, external crystals of 12 or 16 MHz must be connected to generate the 6- or 8-MHz internal clock frequency. The 80186/80188

is fabricated in a 68-pin package. Both processors have on-chip priority interrupt controller circuits to provide five interrupt pins. Like the 8086/8088, the 80186/80188 can directly address one megabyte of memory. The 80186/80188 is provided with 10 new instructions beyond the 8086/8088 instruction set. Examples of these instructions include INS and OUTS for inputting and outputting a string byte or string word.

The 80286, on the other hand, has added memory protection and management capabilities to the basic 8086 architecture. An 8-MHz 80286 provides up to 6 times greater throughput than the 5-MHz 8086. The 80286 is fabricated in a 68-pin package. The 80286 can be operated at a clock frequency of 4, 6, or 8 MHz. An external 82284 clock generator chip is required to generate the clock. The 82284 divides the external clock by 2 to generate the internal clock. The 80286 can be operated in two modes, real address and protected virtual address. Real address mode emulates a very high-performance 8086. In this mode, the 80286 can directly address one megabyte of memory. In virtual address mode, the 80286 can directly address 16 megabytes of memory. Virtual address mode provides (in addition to the real address mode capabilities) virtual memory management as well as task management and protection. The programmer can select one of these modes by loading appropriate data in the 16-bit machine status word (MSW) register by using the load instruction (`LMSW`).

The 80286 was used as the microprocessor of the IBM PC/AT personal computer. An enhanced version of the 80286 is the 32-bit 80386 microprocessor. The 80386 was used as the microprocessor in the IBM 386PC. The 80486 is another 32-bit microprocessor. It is based on the Intel 80386 and includes on-chip floating-point circuitry. IBM's 486 PC contains the 80486 chip. Other 32-bit and 64-bit Intel microprocessors include Pentium, Pentium Pro, Pentium II, Celeron, Pentium III, Pentium 4 and Merced.

Although the 8086 seems to be obsolete, it is expected to be around for some time from second sources. Therefore, a detailed coverage of the 8086 is included. A summary of the 32- and 64-bit microprocessors is then provided.

9.2 8086 Main Memory

The 8086 uses a segmented memory. There are some advantages to working with the segmented memory. First, after initializing the 16-bit segment registers, the 8086 has to deal with only 16-bit effective addresses. That is, the 8086 has to manipulate and store 16-bit address components. Second, because of memory segmentation, the 8086 can be effectively used in time-shared systems. For example, in a time-shared system, several users may share one 8086. Suppose that the 8086 works with one user's program for, say, 5 milliseconds. After spending 5 milliseconds with one of the other users, the 8086 returns to execute the first user's program. Each time the 8086 switches from one user's program to the next, it must execute a new section of code and new sections of data. Segmentation makes it easy to switch from one user program to another.

The 8086's main memory can be divided into 16 segments of 64K bytes each (16 × 64 KB = 1 MB). A segment may contain codes or data. The 8086 uses 16-bit registers to address segments. For example, in order to address codes, the code segment register must be initialized in some manner (to be discussed later): A 16-bit 8086 register called the "instruction pointer" (IP), which is similar to the program counter of a typical microprocessor, linearly addresses each location in a code segment. Because the size of the IP is 16 bits, the segment size is 64K bytes (2^{16}). Similarly, a 16-bit data segment register must be initialized to hold the segment value of a data segment. The contents of

certain 16-bit registers are designed to hold a 16-bit address in a 64-Kbyte data segment. One of these address registers can be used to linearly address each location once the data segment is initialized by an instruction. Finally, in order to access the stack segment, the 8086 16-bit stack segment (SS) register must be initialized; the 64-Kbyte stack is addressed linearly by a 16-bit stack pointer register. Note that the stack memory must be a read/write (RAM) memory. Whenever the programmer reads from or writes to the 8086 memory or stack, two components of a memory address must be considered: a segment value and, an address or an offset or a displacement value. The 8086 assembly language program works with these two components while accessing memory. These two 16-bit components (the contents of a 16-bit segment register and a 16-bit offset or IP) form a logical address. The programmer writes programs using these logical addresses in assembly language programming.

The 8086 includes on-chip hardware to map or translate these two 16-bit components of a memory address into a 20-bit address called a "physical address" by shifting the contents of a segment register four times to left and then adding the contents of IP or offset. Note that the 8086 contains 20 address pins, so the physical address size is 20 bits wide.

Consider, for example, a logical address with the 16-bit code segment register contents of 2050_{16} and the 16-bit 8086 instruction pointer containing a value of 0004_{16}. Suppose that the programmer writes an 8086 assembly language program using this logical address. The programmer assembles this program and obtains the object or machine code. When the 8086 executes this program and encounters the logical address, it will generate the 20-bit physical address as follows: If 16-bit contents of IP = 0004_{16}, 16-bit contents of code segment = 2050_{16}, 16-bit contents of code segment value after shifting logically 4 times to the left = 20500_{16}, then the 20-bit physical address generated by the 8086 on its 20-pin address is 20504_{16} . Note that the 8086 assigns the low address to the low byte of a 16-bit register and the high address to the high byte of the 16-bit register for 16-bit transfers between the 8086 and main memory. This is called *Little-endian byte ordering.*

9.3 8086 Registers

As mentioned in Chapter 6, the 8086 is divided internally into two independent units: the bus interface unit (BIU) and the execution unit (EU). The BIU reads (fetches) instructions, reads operands, and writes results. The EU executes instructions already fetched by the BIU. The 8086 prefetches up to 6 instruction bytes from external memory into a FIFO (first-in–first-out) memory in the BIU and queues them in order to speed up instruction execution. The BIU contains a dedicated adder to produce the 20-bit address. The bus control logic of the BIU generates all the bus control signals, such as the READ and WRITE signals, for memory and I/O. The BIU also has four 16-bit segment registers: the code segment (CS), data segment (DS), stack segment (SS), and extra segment (ES) registers.

All program instructions must be located in main memory, pointed to by the 16-bit CS register with a 16-bit offset contained in the 16-bit instruction pointer (IP). Note that immediate data are considered as part of the code segment. The SS register points to the current stack. The 20-bit physical stack address is calculated from the SS and SP (stack pointer) for stack instructions such as PUSH and POP. The programmer can create a programmer's stack with the BP (base pointer) instead of the SP for accessing the stack using the based addressing mode. In this case, the 20-bit physical stack address is calculated

from the BP and SS. The DS register points to the current data segment; operands for most instructions are fetched from this segment. The 16-bit contents of a register such as the SI (source index) or DI (destination index) or a 16-bit displacement are used as offsets for computing the 20-bit physical address.
The ES register points to the extra segment in which data (in excess of 64 KB pointed to by the DS) is stored. String instructions always use the ES and DI to determine the 20-bit physical address for the destination.

The segments can be contiguous, partially overlapped, fully overlapped, or disjointed. An example of how five segments (SEGMENT 0 through SEGMENT 4), may be stored in physical memory is shown in Figure 9.1. In this example, SEGMENTs 0 and 1 are contiguous (adjacent), SEGMENTs 1 and 2 are partially overlapped, SEGMENTs 2 and 3 are fully overlapped, and SEGMENTs 2 and 4 are disjointed.

Every segment must start on 16-byte memory boundaries. Typical examples of values of segments should then be selected based on physical addresses starting at 00000_{16}, 00010_{16}, 00020_{16}, 00030_{16}, ..., $FFFF0_{16}$. A physical memory location may be mapped into (contained in) one or more logical segments. Many applications can be written to simply initialize the segment registers and then forget them.

A segment can be pointed to by more than one segment register. For example, the DS and ES may point to the same segment in memory if a string located in that segment is used as a source segment in one string instruction and a destination segment in another string instruction. Note that, for string instructions, a destination segment must be pointed to by the ES. One example of four currently addressable segments is shown in Figure 9.2.

The EU decodes and executes instructions. It has a 16-bit ALU for performing arithmetic and logic operations. The EU has nine 16-bit registers: AX, BX, CX, DX, SP,

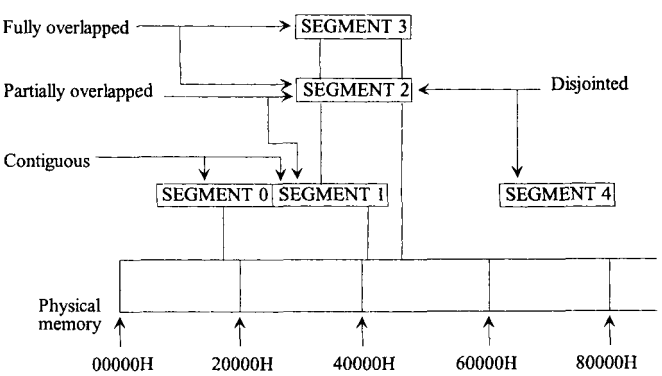


FIGURE 9.1 An Example of 8086 Memory Segments

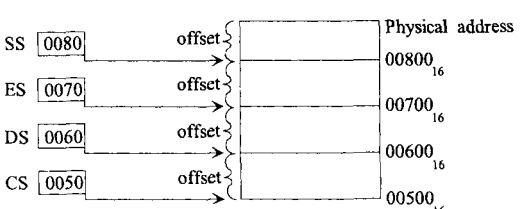


FIGURE 9.2 Four currently addressable 8086 segments

BP, SI, and DI, and the flag register. The 16-bit general registers AX, BX, CX, and DX can be used as two 8-bit registers (AH, AL; BH, BL; CH, CL; DH, DL). For example, the 16-bit register DX can be considered as two 8-bit registers DH (high byte of DX) and DL (low byte of DX). The general-purpose registers AX, BX, CX, and DX perform the following functions:

- The AX register is 16 bit wide whereas AH and AL are 8 bit wide. The use of AX and AL registers is assumed by some instructions. The I/O (IN or OUT) instructions always use the AX or AL for inputting/outputting 16- or 8-bit data to or from an I/O port. Multiplication and division instructions also use the AX or AL.
- The BX register is called the "base register." This is the only general-purpose register whose contents can be used for addressing 8086 memory. All memory references utilizing this register content for addressing use the DS as the default segment register.
- The CX register is known as the *counter* register because some instructions, such as `SHIFT`, `ROTATE`, and `LOOP`, use the contents of CX as a counter, For example, the instruction `LOOP START` will automatically decrement CX by 1 without affecting flags and will check to see if (Cx) = 0. If it is zero, the 8086 executes the next instruction; otherwise, the 8086 branches to the label `START`.
- The DX register, or data register, is used to hold the high 16-bit result (data) (LOW 16-bit data is contained in AX) after 16 × 16 multiplication or the high 16-bit dividend (data) before a 32 ÷ 16 division and the 16-bit remainder after the division (16-bit quotient is contained in AX).
- The two pointer registers, SP (stack pointer) and BP (base pointer), are used to access data in the stack segment. The SP is used as an offset from the current SS during execution of instructions that involve the stack segment in external memory. The SP contents are automatically updated (incremented or decremented) due to execution of a POP or PUSH instruction. The BP contains an offset address in the current SS. This offset is used by instructions utilizing the based addressing mode.
- The two index registers, SI (source index) and DI (destination index), are used in indexed addressing. Note that instructions that process data strings use the SI and DI index registers together with the DS and ES, respectively, in order to distinguish between the source and destination addresses.
- The flag register in the EU holds the status flags, typically after an ALU operation. The

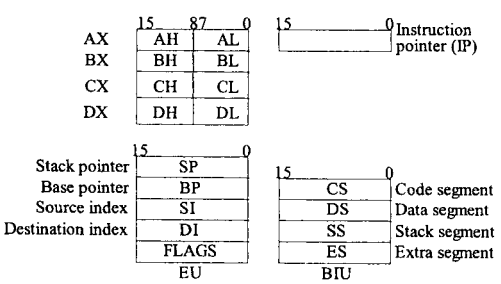


FIGURE 9.3 8086 Registers

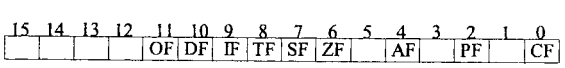


FIGURE 9.4 8086 Flag Register

EU sets or resets these flags to reflect the results of arithmetic and logic operations.

Figure 9.3 depicts the 8086 registers. It shows the nine 16-bit registers in the EU. As described earlier, each one of the AX, BX, CX, and DX registers can be used as two 8-bit registers or as one 16-bit register. The other registers can be accessed as 16-bit registers. Also shown are the four 16-bit segment registers and the 16-bit IP in the BIU. The IP is similar to the program counter. The CS register points to the current code segment from which instructions are fetched. The effective address is derived from the CS and IP. The SS register points to the current stack. The effective address is obtained from the SS and SP. The DS register points to the current data segment. The ES register points to the current extra segment where data is usually stored.

Figure 9.4 shows the 8086 flag register. The 8086 has six one-bit status flags. Let us now explain these flags.

- **AF** (auxiliary carry flag) is set if there is a carry due to addition of the low nibble into the high nibble or a borrow due to the subtraction of the low nibble from the high nibble of a number.
  This flag is used by BCD arithmetic instructions; otherwise, AF is zero.
- **CF** (carry flag) is set if there is a carry from addition or a borrow from subtraction.
- **OF** (overflow flag) is set if there is an arithmetic overflow (i.e., if the size of the result exceeds the capacity of the destination location). An interrupt on overflow instruction is available to generate an interrupt in this situation; otherwise, it is zero.
- **SF** (sign flag) is set if the most significant bit of the result is one; otherwise, it is zero.
- **PF** (parity flag) is set if the result has even parity; PF is zero for odd parity of the result.
- **ZF** (zero flag) is set if the result is zero; ZF is zero for a nonzero result.

The 8086 has three control bits in the flag register that can be set or cleared by the programmer:

1. Setting DF (direction flag) causes string instructions to auto-decrement; clearing DF causes string instructions to auto-increment.
2. Setting IF (interrupt flag) causes the 8086 to recognize external maskable interrupts; clearing IF disables these interrupts.
3. Setting TF (trap flag) puts the 8086 in the single-step mode. In this mode, the 8086 generates an internal interrupt after execution of each instruction. The user can write a service routine at the interrupt address vector to display the desired registers and memory locations. The user can thus debug a program.

9.4 8086 Addressing Modes

The 8086 provides various addressing modes to access instruction operands. Operands may be contained in registers, within the instruction op-code, in memory, or in I/O ports. The 8086 has 12 addressing modes, which can be classified into five groups:

1. Register and immediate modes (two modes)
2. Memory addressing modes (six modes)
3. Port addressing mode (two modes)
4. Relative addressing mode (one mode)
5. Implied addressing mode (one mode)

Note that in the following, symbol ( ) is used to indicate the contents of an 8086 register or a memory location.

9.4.1 Register and Immediate Modes

Register mode. The addressing modes are illustrated utilizing 8086 instructions with directives of a typical assembler. In register mode, source operands, destination operands, or both may be contained in registers. For example, `MOV AX,BX` moves the 16-bit contents of BX into AX. On the other hand, `MOV AH,BL` moves the 8-bit contents of BL into AH.

Immediate mode. In immediate mode, 8- or 16 bit data can be specified as part of the instruction. For example, `MOV CX,5062H` moves the 16-bit data 5062_{16} into register CX.

9.4.2 Memory Addressing Modes

The EU has direct access to all registers and data for register and immediate modes. However, the EU cannot directly access the memory operands. It must use the BIU to access memory operands. For example, when the EU needs a memory operand, it sends an offset value to the BIU. As mentioned before, this offset is added to the contents of a segment register after shifting it four times to the left, generating a 20-bit physical address. For example, suppose that the contents of a segment register is 2052_{16} and the offset is 0020_{16}. Now, in order to generate the 20-bit physical address, the EU passes this offset to the BIU. The BIU then shifts the segment register four times to the left, obtains 20520_{16} and then adds the 0020_{16} offset to provide the 20-bit physical address 20540_{16}.

Note that the 8086 must use a segment register whenever it accesses the memory. Also, every memory addressing mode has a standard default segment register. However, a segment override instruction can be placed before most of the memory operand instructions whose default segment register is to be overridden. For example, `INC BYTE PTR [START]` will increment the 8-bit contents of a memory location in DS with offset `START` by 1. However, segment DS can be overridden by ES as follows: `INC ES: BYTE PTR [START]`. Segments cannot be overridden for stack reference instructions (such as `PUSH` and `POP`). The destination segment of a string segment, which must be ES (if a prefix is used with a string instruction, only the source segment DS can be overridden) cannot be overridden. The code segment (CS) register used in program memory addressing cannot be overridden. The EU calculates an offset from the instruction for a memory operand. This offset is called the operand's *effective address,* or EA. It is a 16-bit number that represents the operand's distance in bytes from the start of the segment in which it resides.

The various memory addressing modes will now be described.

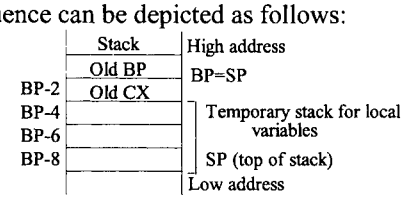
1. ***Memory Direct Addressing.*** In this mode, the effective address is taken directly from the displacement field of the instruction. No registers are involved. For example, `MOV BX,[START]`, or `MOV BX, OFFSET START` moves the contents of the 20-bit address computed from DS and START to BX. Some assemblers use square brackets around START to indicate that the contents of the memory location(s) are at a displacement START from the segment DS. If square brackets are not used, then the programmer may define START as a 16-bit offset by using the assembler directive, `OFFSET`.
2. ***Register Indirect Addressing.*** The effective address of a memory operand may be taken directly from one of the base or index registers (BX, BP, SI, DI). For example, consider `MOV CX,[BX]`. If (DS) = 2000_{16}, (BX) = 0004_{16}, and (20004_{16}) = 0224_{16}, then, after `MOV CX,[BX]`, the contents of CX are 0224_{16}. Note that the segment register used in `MOV CX,[BX]` can be overridden, such as `MOV CX,ES:[BX]`. Now, the `MOV` instruction will use ES instead of DS. If (ES) = 1000_{16} and (10004_{16})

$= 0002_{16}$, then, after `MOV CX,ES:[BX]` , the register CX will contain 0002_{16}. Note that in the above, symbol ( ) is used to indicate the contents of an 8086 register or a memory location.

3. ***Based Addressing.*** In this mode, the effective address is the sum of a displacement value (signed 8-bit or unsigned 16-bit) and the contents of register BX or BP. For example, `MOV AX,4[BX]` moves the contents of the 20-bit address computed from a segment register and BX + 4 into AX. The segment register is DS or SS. The content of BX is unchanged. The displacement (4 in this case) can be unsigned 16-bit or signed 8-bit. This means that if the displacement is 8-bit, then the 8086 sign extends this to 16-bit. Segment register SS is used when the stack is accessed; otherwise, this mode uses segment register DS. When memory is accessed, the 20-bit physical address is computed from BX and DS. On the other hand, when the stack is accessed, the 20-bit physical address is computed from BP and SS. Note that BP may be considered as the user stack pointer while SP is the system stack pointer. This is because SP is used by some 8086 instructions (such as CALL subroutine) automatically.
   The based addressing mode with BP is a very convenient way to access stack data. BP can be used as a stack pointer in SS to access local variables. Consider the following instruction sequence (arbitrarily chosen to illustrate the use of BP for stack):

```
PUSH    BP              ;    Save BP
MOV     BP,SP           ;    Establish BP
PUSH    CX              ;    Save CX
SUB     SP, 6           ;    Allocate 3 words of
                        ;    stack for local variables
MOV     -4[BP], BX      ;    Push BX onto stack using BP
MOV     -6[BP], AX      ;    Push AX onto stack using BP
MOV     -8[BP], DX      ;    Push DX onto stack using BP
ADD     SP, 6           ;    Deallocate stack
POP     CX              ;    Restore CX
POP     BP              ;    Restore BP
```

   This instruction sequence can be depicted as follows:



4. ***Indexed Addressing.*** In this mode, the effective address is calculated from the sum of a displacement value and the contents of register SI or DI. For example, `MOV AX, VALUE [SI]` moves the contents of the 20-bit address computed from VALUE, SI and the segment register into AX. The segment register is DS. The content of SI is unchanged. The displacement (VALUE in this case) can be unsigned 16-bit or signed 8-bit. The indexed mode can be used to access a table.
5. ***Based Indexed Addressing.*** In this mode, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI), and a displacement. For example, `MOV AX, 4[BX][SI]` moves the contents of the 20-bit address computed from the segment register and (BX) + (SI) + 4 into AX. The segment register is DS. The displacement can be unsigned 16-bit or signed 8-bit. This mode can be used to access two-dimensional arrays such as matrices.

6. ***String Addressing.*** This mode uses index registers. SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination when a string instruction is executed. The SI or DI is automatically incremented or decremented to point to the next byte or word depending on DF. The default segment register for source is DS, and it may be overridden; the segment register used for the destination must be ES, and can not be overridden. An example is `MOVS WORD`. If (DF) = 0, (DS) = 3000_{16}, (SI) = 0020_{16}, (ES) 5000_{16}, (DI) = 0040_{16}, (30020) = 30_{16}, (30021) = 05_{16}, (50040) = 06_{16}, and (50041) = 20_{16}, then, after this `MOVS`, (50040) = 30_{16}, (50041) = 05_{16}, (SI) = 0022_{16}, and (DI) = 0042_{16}.

9.4.3 Port Addressing

Two I/O port addressing modes can be used: direct port and indirect port. In either case, 8- or 16-bit I/O transfers must take place via AL or AX respectively.In *direct port mode,* the port number is an 8-bit immediate operand to access 256 ports. For example. `IN AL, 02` moves the contents of port 02 to AL. In *indirect port mode,* the port number is taken from DX, allowing 64K bytes or 32K words of ports. For example, suppose (DX) = 0020, (port 0020) = 02_{16}, and (port 0021) = 03_{16}, then, after `IN AX, DX`, register AX contains 0302_{16}. On the other hand, after `IN AL, DX`, register AL contains 02_{16}.

9.4.4 Relative Addressing Mode

Instructions using this mode specify the operand as a signed 8-bit displacement relative to IP. An example is `JNC START`. This instruction means that if carry = 0, then IP is loaded with the current IP contents plus the 8-bit signed value of START; otherwise, the next instruction is executed.

An advantage of relative mode is that the destination address is specified relative to the address of the instruction after the conditional Jump instruction. Since the 8086 conditional Jump instructions do not contain an absolute address, the program can be placed anywhere in memory which can still be executed properly by the 8086. A program which can be placed anywhere in memory, and can still run correctly is called a "relocatable" program. It is a good practice to write relocatable programs.

9.4.5 Implied Addressing Mode

Instructions using this mode have no operands. An example is `CLC`, which clears the carry flag to zero.

9.5 8086 Instruction Set

The 8086 has approximately 117 different instructions with about 300 op-codes. The 8086 instruction set contains no-operand, single-operand, and two-operand instructions. Except for string instructions that involve array operations, 8086 instructions do not permit memory-to-memory operations. Appendices F and H provide 8086 instruction reference data and the instruction set (alphabetical order), respectively. The 8086 instructions can be classified into eight groups:

1. Data Transfer Instructions
2. Arithmetic Instructions
3. Bit Manipulation Instructions
4. String Instructions
5. Unconditional Transfer Instructions
6. Conditional Branch Instructions
7. Interrupt Instructions
8. Processor Control Instructions

Let us now explain some of the 8086 instructions with numerical examples. Note that

TABLE 9.1 8086 Data Transfer Instructions

	General Purpose
MOV d, s	[d] ← [s] MOV byte or word
PUSH d	PUSH word into stack
POP d	POP word off stack
XCHG mem/reg, mem/reg	[mem/reg] ↔ [mem/reg]; No mem to mem.
XLAT	AL ← [20 bit address computed from AL, BX, and DS]
	Input / Output
IN A, DX or Port	Input byte or word
OUT DX or Port, A	Output byte or word
	Address Object
LEA reg, mem	LOAD Effictive Address
LDS reg, mem	LOAD pointer using DS
LES reg, mem	LOAD pointer using ES
	Flag Transfer
LAHF	LOAD AH register from flags
SAHF	STORE AH register in flags
PUSHF	PUSH flags onto stack
POPF	POP flags off stack

d = "mem" or "reg" or "segreg," s = "data" or " mem" or "reg" or "segreg," A = AX or AL

in the following examples , symbol ( ) is used to indicate the contents of a register or a memory location.

9.5.1 Data Transfer Instructions

Table 9.1 lists the data transfer instructions. Note that LEA is used to load 16-bit offset to a specified register; LDS and LES are similar to LEA except that they load specified register as well as DS or ES. As an example, LEA BX, 3000H has the same meaning as MOV BX,3000H. On the other hand, if (SI)=2000H, then LEA BX,4[SI] will load 2004H into BX while MOV BX,4[SI] will initialize BX with the contents of memory locations computed from 2004H and DS. The LEA instruction can be useful when memory computation is desirable.

In Table 9.1, there are 14 data transfer instructions. These instructions move single bytes and words between a register, a memory location, or an I/O port. Let us explain some of the instructions in Table 9.1.

- `MOV CX,DX` copies the 16-bit contents of DX into CX. `MOV AX,2025H` moves immediate data 2025H into the 16-bit register AX. `MOV CH,[BX]` moves the 8-bit contents of a memory location addressed by BX in segment register DS into CH. If (BX) = 0050H, (DS) = 2000H, and (20050H) = 08H, then, after `MOV CH,[BX]`, the contents of CH will be 08H. `MOV START [BP],CX` moves the 16-bit (CL to first location and then CH) contents of CX into two memory locations addressed by the sum of the displacement START and BP in segment register SS. For example, if (CX) = 5009H, (BP)=0030H, (SS) = 3000H, and START = 06H, then, after `MOV START [BP],CX`, (30036H) = 09H and (30037H) = 50H.
- `LDS SI,[0010H]` loads SI and DS from memory. For example, if (DS) = 2000H, (20010) = 0200H, and (20012) = 0100H, then, after `LDS SI,[0010H]`, SI and DS will contain 0200H and 0100H, respectively.
- In the 8086, the SP is decremented by 2 for PUSH and incremented by 2 for POP. For

example, consider PUSH [BX]. If (DS) = 2000_{16}, (BX) = 0200_{16}, (SP) = 3000_{16}, (SS) = 4000_{16}, and (20200) = 0120_{16}, then, after execution of `PUSH [BX]`, memory locations 42FFF and 42FFE will contain 01_{16} and 20_{16}, respectively, and the contents of SP will be $2FFE_{16}$.

- `XCHG` has three variations: `XCHG` reg, reg and `XCHG` mem, reg or `XCHG` reg, mem. For example, `XCHG AX, BX` exchanges the contents of 16-bit register BX with the contents of AX. `XCHG` mem, reg exchanges 8- or 16-bit data in mem with 8-or 16-bit reg.
- `XLAT` can be used to employ an index in a table or for code conversion. This instruction utilizes BX to hold the starting address of the table in memory consisting of 8-bit data elements. The index in the table is assumed to be in the AL register. For example, if (BX) = 0200_{16}, (AL) = 04_{16}, and (DS) = 3000_{16}, then, after `XLAT`, the contents of location 30204_{16} will be loaded into AL. Note that the XLAT instruction is the same as `MOV AL,[AL][BX]`. As mentioned before, XLAT instruction can be used to convert from one code to another. For example, consider an 8086-based microcomputer with an ASCII keyboard connected to Port A and an EBCDIC printer connected to Port B. Suppose that it is desired to enter numerical data via the ASCII keyboard, and then print them on the EBCDIC printer. Note that numerical data entered into this computer via the keyboard will be in ASCII code. Since the printer only understands EBCDIC code, an ASCII to EBCDIC code conversion program is required. The ASCII codes for numbers 0 through 9 are 30H through 39H while the EBCDIC codes for numbers 0 to 9 are F0H to F9H (Table 2.6). The EBCDIC codes for the numbers 0 to 9 can be stored in a table starting at an offset 2030H , data can be input from the keyboard using IN AL, PORTA, convert this ASCII data to EBCDIC using XLAT instruction, and then output to Port B using OUT PORTB, AL. The instruction sequence for the code conversion program is provided below:

```
MOV   BX,2000H     ;Initialize BX
IN    AL,PORTA     ;Input ASCII data
XLAT               ;Obtain EBCDIC code from table below
OUT   PORTB,AL     ;Output to EBCDIC Printer
ORG   2030H
DB 0F0,0F1,0F2,0F3,0F4,0F5,0F6,0F7,0F8,0F9
```

- Consider fixed port addressing, in which the 8-bit port address is directly specified as part of the instruction. `IN AL, 38H` inputs 8-bit data from port 38H into AL. `IN AX, 38H` inputs 16-bit data from ports 38H and 39H into AX. `OUT 38H, AL` outputs the contents of AL to port 38H. `OUT 38H, AX`, on the other hand, outputs the 16-bit contents of AX to ports 38H and 39H.
- For variable port addressing, the port address is 16-bit and is specified in the DX register. Assume (DX) = 3124_{16} in all the following examples.

  `IN AL,DX` inputs 8-bit data from 8-bit port 3124_{16} into AL.

  `IN AX,DX` inputs 16-bit data from ports 3124_{16} and 3125_{16} into AX.

  `OUT DX,AL` outputs 8-bit data from AL into port 3124_{16}.

  `OUT DX,AX` outputs 16-bit data from AX into ports 3124_{16} and 3125_{16}.

  Variable port addressing allows up to 65,536 ports with addresses from 0000H to FFFFH. The port addresses in variable port addressing can be calculated dynamically in a program. For example, assume that an 8086-based microcomputer is connected to three printers via three separate ports. Now, in order to output to each one of the printers, separate programs are required if fixed port addressing is used. However,

with variable port addressing, one can write a general subroutine to output to the printers and then supply the address of the port for a particular printer in which data output is desired to register DX in the subroutine.

9.5.2 Arithmetic Instructions

Table 9.2 shows the 8086 arithmetic instructions. These operations can be performed on four types of numbers: unsigned binary, signed binary, unsigned packed decimal, and signed packed decimal numbers. Binary numbers can be 8 or 16 bits wide. Decimal numbers are stored in bytes; two digits per byte for packed decimal and one digit per byte for unpacked decimal with the high 4 bits filled with zeros.

Let us explain some of the instructions in Table 9.2.

- Consider `ADC` mem/reg , mem/reg. This instruction adds source and destination data along with the carry flag, and stores the result in destination. There is no ADC mem , mem instruction. All flags in the low byte of the Flag register are affected. For example, if (AX) = 0020_{16}, (BX) = 0300_{16}, CF = 1, (DS) = 2020_{16}, and (20500) = 0100_{16}, then, after `ADC AX, [BX]`, the contents of register AX = 0020 + 0100 + 1 = 0121_{16}; CF = 0, PF = 0 (Result with odd Parity), AF = 0, ZF = 0 (Nonzero Result), SF = 0 (Most Significant bit of the result is zero), and OF = 0.
- Consider SBB mem/reg , mem/reg. This instruction subtracts source data and the carry flag from destination data, and stores the result in destination. There is no SBB mem , mem instruction. All flags in the low byte of the Flag register are affected. For example, if (CH) = 03_{16}, (DL) = 02_{16}, and CF = 1, then, after SBB CH,DL, the contents of register CH = 03 - 02 - 1 = 00_{16}.

```
                                        1111 111← Intermediate Carries
Using two's complement subtraction, (CH) =  0000 0011 (+3)
Add two's complement of 3 (DL plus CF)  = + 1111 1101 (-3)
                                        ----------------------
                          Final Carry →1    0000 0000
```

Final carry is one's complemented after subtraction to reflect the correct borrow. Hence, CF = 0. Also, PF = 1 (Even parity; number of 1's in the result is 0 and 0 is an even number), AF = 1, ZF = 1 (Zero Result), SF = 0 (Most Significant bit of the result is zero), and OF = $C_f \oplus C_p = 1 \oplus 1 = 0$.

- The Compare (CMP) instruction subtracts source from destination providing no result of subtraction; all status flags are affected based on the result. Note that the SUBTRACT instruction provides the result and also affects the status flags. Consider `CMP DH,BL`. If prior to execution of the instruction, (DH) = 40H and (BL) = 30H then after execution of `CMP DH, BL`, the flags are: CF = 0, PF = 0, AF = 0, ZF = 0, SF = 0, and OF = 0; result 10H is not provided. Suppose it is desired to find the number of matches for an 8-bit number in an 8086 register such as DL in a data array of 50 bytes in memory pointed to by BX in DS. The following instruction sequence with `CMP DL, [BX]` rather than `SUB DL, [BX]` can be used :

```
        MOV     AL,0      ;  Clear AL to 0, AL to hold number of
                          ;  matches
        MOV     CX,50     ;  Initialize array count
START:  CMP     DL,[BX]   ;  Compare the number to be matched in DL
        JZ      MATCH     ;  with a data byte in the array.If there is
                          ;  a match, ZF=1. Branch to label MATCH.
        JMP     DOWN      ;  Unconditional jump to label DOWN.
MATCH:  INC     AL        ;  increment AL to hold number of matches.
```

TABLE 9.2 8086 Arithmetic Instructions

	Addition	
ADD a, b	Add byte or word	
ADC a, b	Add byte or word with carry	
INC reg/mem	Increment byte or word by one	
AAA	ASCII adjust for addition	
DAA	Decimal adjust [AL], to be used after ADD or ADC	
	Subtraction	
SUB a, b	Subtract byte or word	
SBB a, b	Subtract byte or word with borrow	
DEC reg/mem	Decrement byte or word by one	
NEG reg/mem	Negate byte or word	
CMP a, b	Compare byte or word	
AAS	ASCII adjust for subtraction	
DAS	Decimal adjust [AL] after SUB or SBB	
MUL reg/mem	Multiply byte or word unsigned	for byte
IMUL reg/mem	Integer multiply byte or word (signed)	[AX] ← [AL] · [mem/reg]
		for word
		[DX][AX] ← [AX] · [mem/reg]
	Division	
DIV reg/mem	Divide byte or word unsigned	16 ÷ 8 bit; [AX] ← $\frac{[AX]}{[mem/reg]}$
IDIV reg/mem	Integer divide byte or word (signed)	[AH] ← remainder
		[AL] ← quotient
		32÷16 bit; [DX:AX]← $\frac{[DX:AX]}{[mem/reg]}$
		[DX] ← remainder
		[AX] ← quotient
AAD	ASCII adjust for division	
CBW	Convert byte to word	
CWD	Convert word to double word	

a = "reg" or "mem," b = "reg" or "mem" or "data."

```
DOWN:     INC     BX      ; Increment BX to point to next data byte.
          LOOP    START   ; Decrement CX by 1, go back to START if
                          ; CX ≠ 0.If CX = 0, go to the next
                          ; instruction
                          ; AL contains the number of matches
```

In the above, if `SUB DL, [BX]` were used instead of `CMP DL, [BX]`, then the number to be matched needed to be loaded after each subtraction because the contents of DL would have been lost after each SUB. Since we are only interested in the match rather than the result, `CMP DL, [BX]` instead of `SUB DL, [BX]` should be used in the above.

- Numerical data received by an 8086-based microcomputer from a terminal is usually in ASCII code. The ASCII codes for numbers 0 to 9 are 30H through 39H. Two 8-bit data items can be entered into an 8086-based microcomputer via a keyboard. The ASCII codes for these data items (with 3 as the upper nibble for each type) can be added. AAA instruction can then be used to provide the correct unpacked BCD. Suppose that ASCII codes for 2 (32_{16}) and 5 (35_{16}) are entered into an 8086-based microcomputer via a keyboard. These ASCII codes can be added and then the result can be adjusted to provide the correct unpacked BCD using the AAA instruction as follows:

```
ADD     CL, DL    ;    (CL) = 32H = ACSII for 2
                  ;    (DL) = 35H = ASCII for 5
                  ;    Result (CL) = 67H
MOV     AL, CL    ;    Move ASCII result
                  ;    into AL because AAA
                  ;    adjusts only (AL)
AAA               ;    (AL) = 07, unpacked
                  ;    BCD for 7
```

  Note that, in order to print the unpacked BCD result 07_{16} on an ASCII printer, (AL) = 07 can be ORed with 30H to provide 37H, the ASCII code for 7.
  In case of an invalid BCD digit after addition, AAA instruction can be used to obtain correct unpacked BCD as follows:

```
ADD     BH, DL    ;    (BH) = 38H = ACSII for 8
                  ;    (DL) = 37H = ASCII for 7
                  ;    Result (BH) = 6FH
MOV     AL, BH    ;    Move ASCII result
                  ;    into AL because AAA gets rid of 6 in
                  ;    the upper 4 bits of AL, and adds 6 to
                  ;    F for BCD correction to provide the
AAA               ;    correct unpacked BCD for 5, (AL) = 05,
                  ;    with  CF=1 so that correct result is
                  ;    15 decimal
```

- DAA is used to adjust the result of adding two packed BCD numbers in AL to provide a valid BCD number. If, after the addition, the low 4 bits of the result in AL is greater than 9 (or if AF = 1), then the DAA adds 6 to the low 4 bits of AL. On the other hand. if the high 4 bits of the result in AL are greater than 9 (or if CF = 1), then DAA adds 60H to AL.
- DAS may be used to adjust the result of subtraction in AL of two packed BCD numbers to provide the correct packed BCD. While performing these subtractions, any borrows from low and high nibbles are ignored, For example, consider subtracting packed BCD 55 in DL from packed BCD 94 in AL:

  Packed BCD 55 = 0101 0101_2 and Packed BCD 94 = 1001 0100_2.

```
                          Packed BCD 94 = 1001 0100
          Add Two's complement of 0101 0101 = 1010 1011
                                              ---------
                  Ignore Carry → 1  0011 1111= 3FH
```

The invalid BCD digit (F) in the low 4 bits of the result can be corrected by subtracting 6 from F:

```
Low Nibble = FH = 1111
             -6  = 1010
             ---------------
Ignore Carry →1   1001   This will provide the correct BCD result of 39.
```

The following 8086 instruction sequence will accomplish this:

```
SUB    AL,DL ;[AL] = 3FH
DAS          ;[AL] = 39
```

- For 8-bit by 8-bit signed or unsigned multiplication between the contents of a memory location and AL, assembler directive BYTE PTR can be used. Example: IMUL BYTE PTR[BX]. On the other hand, for 16-bit by 16-bit signed or unsigned multiplication between the 16-bit contents of a memory location and register AX, assembler directive WORD PTR can be used. Example: MUL WORD PTR[SI].
- Consider 16 × 16 unsigned multiplication, `MUL WORD PTR [BX]`. If (BX) = 0050H, (DS) = 3000H, (30050H) = 0002H, and (AX) = 0006H, then, after `MUL WORD PTR [BX]`, (DX) = 0000H and (AX) = 000CH.
- `MUL` mem/reg provides unsigned 8 × 8 or unsigned 16 × 16 multiplication. Consider `MUL BL`. If (AL) = 20_{16} and (BL) = 02_{16}, then, after `MUL BL`, register AX will contain 0040_{16}.
- `IMUL` mem/reg provides signed 8 × 8 or signed 16 × 16 multiplication. As an example, if (CL) = FDH = -3_{10} and (AL) = FEH = -2_{10}, then, after `IMUL CL`, register AX contains 0006H.
- Consider `IMUL DH`. If (AL) = FF_{16} = -1_{10} and (DH) = 02_{16}, then, after `IMUL DH`, register AX will contain $FFFE_{16}$ (-2_{10}) .
- `DIV` mem/reg performs unsigned division and divides (AX) or (DX:AX) registers by reg or mem. For example, if (AX) = 0005_{16} and (CL) = 02_{16}, then, after `DIV CL`, (AH) = 01_{16} = Remainder and (AL) = 02_{16} = Quotient.
- Consider `DIV BL`. If (AX) = 0009H and (BL) = 02H, then, after `DIV BL`,
  (AH) = remainder = 01H
  (AL) = quotient = 04H
- `IDIV` mem/reg performs signed division and divides 16-bit contents of AX by an 8-bit number in a register or a memory location, or 32-bit contents of DX:AX registers by a 16-bit number in a register or a memory location. Consider IDIV CX. If (CX) = 2 and (DXAX) = -5_{10} = $FFFFFFFB_{16}$, then, after this `IDIV`, registers DX and AX will contain:

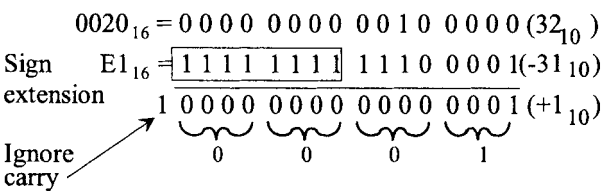
DX	AX
FFFF	FFFE
16-bit remainder = -1_{10}	16-bit quotient = -2_{10}

Note that in the 8086, after `IDIV`, the sign of remainder is always the same as the dividend unless the remainder is equal to zero. Therefore, in this example, because the dividend is negative (-5_{10}), the remainder is negative (-1_{10}).

- For 16-bit by 8-bit signed or unsigned division of the 16-bit contents of AX by 8-bit contents of a memory location, assembler directive BYTE PTR can be used. Example: IDIV BYTE PTR[BX]. On the other hand, for 32-bit by 16-bit signed or unsigned division of the 32-bit contents of DXAX by 16-bit contents of a memory location , assembler directive WORD PTR can be used. Example: MUL WORD PTR[SI].
- Consider `IDIV WORD PTR [BX]`. If (BX) = 0020H, (DS) = 2000H, (20020H) = 0004H, and (DX) (AX) = 00000011H, then, after `IDIV WORD PTR [BX]`,

  (DX) = remainder = 0001H

  (AX) = quotient = 0004H
- Consider `CBW`. This instruction extends the sign from the AL register to the AH register. For example, if AL = $F1_{16}$, then, after execution of `CBW`, register AH will contain FF_{16} because the most significant bit of $F1_{16}$ is 1. Note that the sign extension is very useful when one wants to perform an arithmetic operation on two signed numbers of different lengths. For example. the 16-bit signed number 0020_{16} can be added with the 8-bit signed number $E1_{16}$ by sign-extending E1 as follows:



- Another example of sign extension is that, to multiply a signed 8-bit number by a signed 16-bit number, one must first sign-extend the signed 8-bit into a signed 16-bit number and then the instruction `IMUL` can be used for 16 × 16 signed multiplication. For unsigned multiplication of a 16-bit number by an 8-bit number, the 8-bit number must be zero extended to 16 bits using logical instruction such as AND before using the `MUL` instruction.
- `CWD` sign-extends the AX register into the DX register. That is, if the most significant bit of AX is 1, then $FFFF_{16}$ is stored into DX.
- `AAD` converts two unpacked BCD digits in AH and AL to an equivalent binary number in AL after converting them to packed BCD. `AAD` must be used before dividing two unpacked BCD digits in AX by an unpacked BCD byte. For example, consider dividing (AX) = unpacked BCD 0508 (58 Packed BCD) by (DH) = 07H. (AX) must first be converted to binary by using `AAD`. The register AX will then contain 003AH = 58 Packed BCD. After `DIV DH`, (AL) = quotient = 08 (unpacked BCD), and (AH) = remainder 02 (unpacked BCD).
- `AAM` adjusts the product of two unpacked BCD digits in AX. If (AL) = 03H (unpacked BCD for 3) = 00000011_2 and (CH) = 08H (unpacked BCD for 8) = $0000\ 1000_2$, then, after `MUL CH`, (AX) = 0000000000011000_2 = 0018H, and, after using `AAM`, (AX) = 0000010000000100_2 = unpacked 0204. The following instruction sequence accomplishes this:

```
MUL CH
AAM
```

  Note that the 8086 does not allow multiplication of two ASCII codes. Therefore, before multiplying two ASCII bytes received from a terminal, one must make the upper 4 bits of each one of these bytes zero, multiply them as two unpacked BCD digits, and then use `AAM` for adjustment to convert the unpacked BCD product back to

ASCII by ORing the product with 3030H. The result in decimal can then be printed on an ASCII printer.

9.5.3 Bit Manipulation Instructions

The 8086 provides three groups of bit manipulation instructions. These are logicals, shifts, and rotates, as shown in Table 9.3. The operand to be shifted or rotated can be either 8- or 16-bit. Let us explain some of the instructions in Table 9.3

- Consider `AND BH,8FH`. If prior to execution of this instruction, (BH) = 72H, then after execution of `AND BH,8FH`, the following result is obtained:

```
(BH) =  72H = 0111 0010
  AND   8FH = 1000 1111
              ---------
(BH) = 0000 0010
```

ZF = 0 (Result is nonzero), SF = 0 (Most Significant Bit of the result is 0), PF = 0 (Result has odd parity). CF, AF, and OF are always cleared to 0 after logic operation. The status flags are similarly affected after execution of other logic instructions such as OR, XOR, NOT, and TEST.

The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the word can be logically ANDed with appropriate data to accomplish this. For example, the bit value at bit 2 of an 8-bit number 0100 1Y10 (where unknown bit value of Y is to be determined) can be obtained as follows:

```
        0 1 0 0 1 Y 1 0  -- 8-bit number
AND     0 0 0 0 0 1 0 0  -- Masking data
        -----------------
        0 0 0 0 0 Y 0 0  -- Result
```

If the bit value Y at bit 2 is 1, then the result is nonzero (Flag Z=0); otherwise, the result is zero (Flag Z=1). The Z flag can be tested using typical conditional JUMP instructions such as JZ (Jump if Z=1) or JNZ (Jump if Z=0) to determine whether Y

TABLE 9.3 8086 Bit Manipulation Instructions

	Logicals
`NOT` mem/reg	NOT byte or word
`AND` a, b	AND byte or word
`OR` a, b	OR byte or word
`XOR` a, b	Exclusive OR byte or word
`TEST` a, b	Test byte or word
	Shifts
`SHL/SAL` mem/reg, `CNT`	Shift logical/arithmetic left byte or word
`SHR/SAR` mem/reg, `CNT`	Shift logical/arithmetic right byte or word
	Rotates
`ROL` mem/reg, `CNT`	Rotate left byte or word
`ROR` mem/reg, `CNT`	Rotate right byte or word
`RCL` mem/reg, `CNT`	Rotate through carry left byte or word
`RCR` mem/reg, `CNT`	Rotate through carry right byte or word

a = "reg" or "mem," b = "reg" or "mem" or "data," `CNT` = number of times to be shifted.
If `CNT` > 1, then `CNT` is contained in CL. Zero or negative shifts and rotates are illegal.
If `CNT` = 1 then `CNT` is immediate data. Up to 255 shifts are allowed.

is 0 or 1. This is called masking operation. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the Least Significant bit (LSB) of the number (LSB=0 for even and LSB=1 for odd).

- Consider `OR DL,AH`. If prior to execution of this instruction, [DL] = A2H and [AH] = 5DH, then after exection of `OR DL,AH`, the contents of DL are FFH. The flags are affected similar to the AND instruction. The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits. For example, a 1 can be inserted using the OR instruction at bit number 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits as follows:

```
        0 1 1 1 0 0 1 1 -- 8-bit number
OR      0 0 0 0 1 0 0 0 -- data for inserting a 1 at bit number 3
        ---------------
        0 1 1 1 1 0 1 1 -- Result
```

- Consider `XOR CX,2`. If prior to execution of this instruction, `(CX)` = 2342H, then after execution of `XOR CX,2`, the 16-bit contents of CX will be 2340H. All flags are affected in the same manner as the AND instruction. The Exclusive-OR instruction can be used to find the ones complement of a binary number by XORing the number with all 1's as follows:

```
          0 1 0 1 1 1 0 0 -- 8-bit number
XOR       1 1 1 1 1 1 1 1 -- data
          -----------------------
          1 0 1 0 0 0 1 1 -- Result ( Ones Complement of the
                             8-bit number 0 1 0 1 1 1 0 0 )
```

- `TEST CL,05H` logically ANDs (CL) with 00000101_2 but does not store the result in CL. All flags are affected.
- Consider `SHR` mem/reg, `CNT` or `SHL` mem/reg, `CNT`. These instructions are logical right or left shifts, respectively. The CL register contains the number of shifts if the shift is greater than 1. If `CNT` = 1, the shift count is immediate data. In both cases, the last bit shifted out goes to CF (carry flag) and 0 is the last bit shifted in. For example, SHL BL,1 logically shifts the contents of BL one bit to the left. Note that the shift count '1' is immediate data. Now prior to execution of this instruction, if (BL) = $A1_{16}$ and CF = 0, then after SHL Bl,1, the contents of BL are 42_{16} and CF = 1.
- Consider the 8086 instruction sequence,

```
MOV CL,2  ; shift count 2 is moved into CL
SHR  DX,CL; Logically shifts (DX) twice to right
```

Prior to execution of the above instruction sequence, if (DX) = 97_{16} and CF = 0, then after execution
of the above instruction sequence, (DX) = 25_{16} and CF = 1.

- Figure 9.5 shows `SAR` mem/reg, `CNT` or `SAL` mem/reg, `CNT`. Note that a true arithmetic left shift does not exist in 8086 because the sign bit is not retained after execution of `SAL`. Also, `SAL` and `SHL` perform the same operation except that `SAL` sets OF to 1 if the sign bit of the number shifted changes during or after shifting. This will allow one to multiply a signed number by 2^n by shifting the number n times to left; the result is correct if OF = 0 while the result is incorrect if OF = 1. Since the execution time of the multiplication instruction is longer, multiplication by shifting may be more efficient when multiplication of a signed number by 2^n is desired.

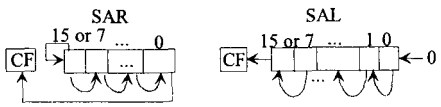


FIGURE 9.5 8086 SAR and SAL instructions

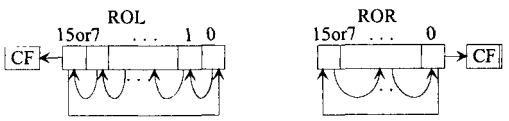


FIGURE 9.6 8086 ROR and ROL instructions

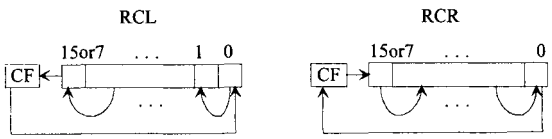


FIGURE 9.7 8086 RCL and RCR instructions

- ROL mem/reg, CNT rotates [mem/reg] left by the specified number of bits (Figure 9.6). The number of bits to be rotated is either 1 or contained in CL. For example, if CF = 0, (BX) = 0010_{16}, and (CL) = 03_{16} then, after ROL BX, CL, register BX will contain 0080_{16} and CF = 0. On the other hand, ROL BL, 1 rotates the 8-bit contents of BL 1 bit to the left. ROR mem/reg, CNT is similar to ROL except that the rotation is to the right (Figure 9.6).
- Figure 9.7 shows RCL mem/reg, CNT and RCR mem/reg, CNT.

9.5.4 String Instructions

The word "string" means that an array of data bytes or words is stored in consecutive memory locations. String instructions are available to MOVE, COMPARE, or SCAN for a value as well as to move string elements to and from AL or AX. The instructions, listed in Table 9.4, contain "repeat" prefixes that cause these instructions to be repeated in hardware, allowing long strings to be processed much faster than if done in a software loop.

Let us explain some of the instructions in Table 9.4.

- MOVS WORD or BYTE moves 8- or 16-bit data from the memory location addressed by SI in DS to the memory location addressed by DI in ES. SI and DI are incremented or decremented depending on the DF flag. For example, if (DF) = 0, (DS) = 1000_{16}, (ES) = 3000_{16}, (SI) = 0002_{16}, (DI) = 0004_{16}, and (10002) = 1234_{16}, then, after MOVS WORD, (30004) = 1234_{16}, (SI) = 0004_{16}, and (DI) =

TABLE 9.4 8086 String Instructions

REP	Repeat MOVS or STOS until CX = 0
REPE/REPZ.	Repeat CMPS or SCAS until ZF = 1 or CX = 0
REPNE/REPNZ	Repeat CMPS or SCAS until ZF = 0 or CX = 0
MOVS BYTE/WORD	Move byte or word string
CMPS BYTE/WORD	Compare byte or word string
SCAS BYTE/WORD	Scan byte or word string
LODS BYTE/WORD	Load from memory into AL or AX
STOS BYTE/WORD	Store AL or AX into memory

0006_{16}. Assuming $(10002_{16}) = 1234_{16}$, the following 8086 instruction sequence will accomplish the above:

```
CLD                 ;DF = 0
MOV   AX,1000H      ;DS = 1000H
MOV   DS,AX
MOV   BX,3000H      ;ES = 3000H
MOV   ES,BX
MOV   SI,0002H      ;Initialize SI to 0002₁₆
MOV   DI,0004H      ;Initialize DI to 0004₁₆
MOVS  WORD
```

Note that DS (source segment) in the MOVS instruction can be overridden while the destination segment, ES is fixed, cannot be overridden. For example, the instruction ES: MOVS WORD will override the source segment, DS by ES while the destination segment remains at ES so that data will be moved in the same extra segment, ES.

- REP repeats the instruction that follows until the CX register is decremented to 0. For example, the following instruction sequence uses LOOP instruction for moving 50 bytes from source to destination:

```
MOV     CX,50      ;  Initialize CX to 50
BACK:   MOVSB      ;  Move a byte from source array to destination
        LOOP BACK  ;  array in the direction based on DF. LOOP
                   ;  decrements CX by 1
                   ;  and goes to label BACK if CX ≠ 0. If CX =
                   ;  0,goes to the next instruction. Thus, 50 bytes
                   ;  are moved
```

The above instruction sequence can be replaced using REP prefix as follows:

```
MOV     CX,50      ;  Initialize CX to 50
REPMOVSB           ;  Move a byte from source array to destination
                   ;  array in the direction based on DF. REP
                   ;  decrements CX by 1
                   ;  and executes MOVSB 50 times.
                   ;  Thus, 50 bytes are moved.
```

- A REPE/REPZ or REPNE/REPNZ prefix can be used with CMPS or SCAS to cause one of these instructions to continue executing until ZF = 0 (for the REPNE/REPNZ prefix) or CX = 0. REPE and REPZ also provide a similar purpose. If CMPS is prefixed with REPE or REPZ, the operation is interpreted as "compare while not end-of-string (CX ≠ 0) or strings are equal (ZF = 1)." If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX ≠ 0) or strings not equal (ZF = 0)." Thus, repeated CMPS instructions can be used to find matching or differing string elements.
- If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX ≠ 0) or string-element = scan-value (ZF = 1)" This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX ≠ 0) or string-element is not equal to scan-value (ZF = 0)." This form may be used to locate a value in a string.
- Consider SCAS WORD or BYTE. This compares the memory with AL or AX. If $(DI) = 0000_{16}$, $(ES) = 2000_{16}$, (DF) = 0, $(20000) = 05_{16}$, and $(AL) = 03_{16}$, then, after

SCAS BYTE, DI will contain 0001_{16} because (DF) = 0 and all flags are affected based on the operation (AL) - (20000).

- CMPS WORD or BYTE subtracts without any result (affects flags accordingly) 8- or 16-bit data in the source memory location addressed by SI in DS from the destination memory location addressed by DI in ES. SI and DI are incremented or decremented depending on the DF flag. For example, if (DF) = 0, (DS) = 1000_{16}, (ES) = 3000_{16}, (SI) = 0002_{16}, (DI) = 0004_{16}, (10002) = 1234_{16}, and (30004) = 1234_{16} then, after CMPS WORD, CF = 0, PF = 1, AF = 1, ZF = 1, SF = 0, OF = 0, (10002) = 1234_{16}, and (30004) = 1234_{16}, (SI) = 0004_{16}, and (DI) = 0006_{16}.
- LODS BYTE or WORD loads a byte into AL or a word into AX respectively from a string in memory addressed by SI in DS ; SI is then automatically incremented or decremented by 1for a byte or by 2 for a word based on DF. For example, prior to execution of LODS BYTE, if (SI)= 0020H, (DS) = 3000H, (30020H) = 05H, DF = 0, then after execution of LODS BYTE, 05H is loaded into AL; SI is then automatically incremented to 0021H since DF = 0. STOS BYTE or WORD, on the other hand, stores a byte in AL or a word in AX respectively into a string addressed by DI in ES. DI is then automatically incremented or decremented by 1 for a byte or by 2 for a word based on DF.

9.5.5 Unconditional Transfer Instructions

Unconditional transfer instructions transfer control to a location either in the current executing memory segment (intrasegment) or in a different code segment (intersegment). Table 9.5 lists the unconditional transfer instructions.

The 8086 CALL instructions provide the mechanism to call a subroutine into operation while the RET instruction placed at the end of the subroutine transfers control back to the main program. There are two types of 8086 CALL instruction. These are intrasegment CALL (IP changes, CS is fixed), and intersegment CALL (both IP and CS are changed). Intrasegment or Intersegment CALL is defined by the various operands of the CALL instruction. For example, the three operands NEAR PROC, mem16, and reg16 define intrasegment CALLs to a subroutine. Upon execution of the intrasegment CALL with any of the three operands, the 8086 pushes the current contents of IP onto the stack; the SP is then decremented by 2. The saved IP value is the offset that contains the next instruction to be executed in the main program. The 8086 then places a new 16-bit value (Offset of the first instruction in the subroutine) into IP. The three types of operands of the intrasegment CALL will be discussed next.

Consider CALL NEAR PROC. The assembler directive NEAR specifies the CALL instruction with relative addressing mode. This means that NEAR determines a 16-bit displacement, and the offset is computed relative to the address of the CALL instruction. With 16-bit displacement, the range of the CALL instruction is limited to -32766 to + 32765 (0 being positive). As an example, consider the following 8086 instruction sequence:

```
CODE        SEGMENT
            ASSUME      CS:CODE, DS:DATA, SS:STACK
```

TABLE 9.5 8086 Unconditional Transfers

CALL reg/mem/disp 16	Call subroutine
RET or RET disp 16	Return from subroutine
JMP disp8/disp 16 /reg16/mem16	Unconditional jump

```
            ------------
            ------------
            CALL  MULTI
            ------------
            ------------
            HLT
MULTI       PROC        NEAR
            ------
            ------
            RET
MULTI       ENDP
CODE        ENDS
```

In the above, the main program is located in a segment named CODE. A subroutine called MULTI is also resident in the same code segment named CODE. Since this subroutine is in the same code segment as the main program containing the CALL instruction, the contents of CS are not altered to access it. Use of the assembler directive NEAR in the statement MULTI PROC NEAR tells the 8086 assembler that the main program and the subroutine are located in the same code segment.

The instructions CALL mem16 and CALL reg16 specify a memory location or a 16-bit register such as BX to hold the offset to be loaded into IP. Thus, these two CALL instructions use indirect addressing mode. An example of CALL mem16 is CALL [BX] which loads the 16-bit value stored in the memory location pointed to by BX into IP. The physical address of the offset is calculated from the current DS and the contents of BX. The first instruction of the subroutine is contained in the address computed from new IP value and current CS. Next, typical examples of CALL reg16 are CALL BX and CALL BP; these instructions load the 16-bit contents of BX or BP into IP. The starting address (physical address) of the subroutine is computed from the new value of IP and the current CS contents. Note that intrasegment CALL instructions are used when the main program and the subroutine are located in the same code segment.

Intersegment CALL instructions are used when the main program and the subroutine are located in two different code segments. The two intersegment CALL instructions are CALL FAR PROC and CALL mem32. These instructions define a new offset for IP and a new value for CS. Upon execution of these two instructions, the 8086 pushes the current contents of IP and CS onto the stack, the new values of IP and CS are then loaded. For example consider CALL FAR PROC which loads the new value of IP from the next two bytes, and the new value of CS from the following two bytes. As an example, consider the following 8086 instruction sequence:

```
CODE        SEGMENT
            ASSUME      CS:CODE, DS:DATA, SS:STACK
            ------------
            ------------
            CALL  MULTI
            ------------
            ------------
            HLT
CODE        ENDS
SUBR        SEGMENT
MULTI       PROC        FAR
```

```
                ASSUME      CS:SUBR
                ------
                ------
                RET
MULTI           ENDP
SUBR            ENDS
```

In the above, the main program is located in a segment named CODE. A subroutine called MULTI is in a segment named SUBR. Since this subroutine is in a different code segment from the CALL instruction, the contents of CS must be altered to access it. Use of the assembler directive FAR in the statement MULTI PROC FAR tells the 8086 assembler that the main program and the subroutine are located in different code segments. When the assembler translates the CALL instruction, it will assign the value of SUBR to CS, and will place the offset of the first instruction of the subroutine in SUBR as the IP value in the instruction.

CALL FAR [SI] stores the pointer for the subroutine as four bytes in data memory. The location of the first byte of the four-byte pointer is specified indirectly by one of the 8086 registers (SI in this case). In this example, the 20-bit physical address of the first byte of the four-byte pointer is computed from DS and SI. Finally, CALL FAR [BX] pushes CS and IP onto stack and loads IP and CS with the contents of four consecutive bytes pointed to by BX.

`RET` instruction is usually placed at the end of a subroutine which pops IP (pushed onto the stack by the intrasegment `CALL` instruction) or both IP and CS (pushed onto the stack by the intersegment `CALL` instruction), and returns control to the main program. `RET` disp 16, on the other hand, adds 16-bit value (disp 16) to SP after placing the return address into IP (for intrasegment `CALL`) or into IP and CS ((for intersegment `CALL`). The main objective of inclusion of the 16-bit displacement operand with the `RET` instruction is to discard the parameters that were saved onto the stack before execution of the subroutine `CALL` instruction.

Similar to the `CALL` instruction, the jump instruction in Table 9.5 can be either intrasegment `JMP` (Jump within the current code segment; only IP changes) or intersegment `JMP` (Jump from one code segment to another code segment; both CS and IP contents are modified). Intrasegment Jump can have an operand with a short label, near label, reg16 or mem16. For example, the short label and near label operands use relative addressing mode. This means that the Jump is performed relative to the address of the `JMP` instruction. For jumps with short label, IP changes and CS is fixed. `JMP disp8` adds the second object code byte (signed 8-bit displacement) to (IP + 2), and (CS) is unchanged. With an 8-bit signed displacement, jump with a short label operand is allowed in the range from -128 to +127 (0 being positive) from the address of the `JMP` instruction. Near label operand allows a `JMP` instruction to have a signed 16-bit displacement with a range -32K to +32K bytes from the address of the `JMP` instruction. An example of `JMP` short label or near label is `JMP START`. The 8086 assembler automatically computes the value of the displacement `START` at assembly time. The programmer does not have to worry about it. Based upon the displacement size of `START` (in this case), the assembler determines whether the `JMP` is to be performed with short or near label.

`JMP` reg16 or `JMP` mem16 specifies the JUMP address respectively by the 16-bit contents of of a register or a memory location. The range for this `JMP` is from -32K to +32K bytes from the address of the `JMP`. An example of `JMP` reg16 is `JMP SI` which

copies the contents of SI into IP. SI contains the 16-bit displacement. The 8086 computes the physical address from the current CS value and the new IP value. An example of JMP mem16 is JMP [DI] which uses the contents of DI as the address of the memory location containing the offset. This offset is placed into IP. The physical address is computed from this IP value and the current code segment value.

The intersegment JMP instruction includes operands with far label and mem32. Jump with far label uses a 32-bit immediate operand ; the first 16 bits are loaded into IP while the next 16 bits are loaded into CS. An example of JMP with far label is JMP FAR BEGIN (or some 8086 assemblers use JMP FAR PTR BEGIN) which unconditionally branches to a label BEGIN in a different code segment.

Finally, JMP mem32 indirectly specifies the offset and the code segment values. IP and CS are loaded from the 32-bit contents of four consecutive memory locations; each memory location contains a byte. As an example, JMP FAR [SI] loads IP and CS with the contents of four consecutive bytes pointed to by SI in DS.

9.5.6 Conditional Branch Instructions

All 8086 conditional branch instructions use 8-bit signed displacement. That is, the displacement covers a branch range of -128 to +127, with 0 being positive. The structure of a typical conditional branch instruction is as follows:

If condition is true,
then IP ← IP + disp8,
otherwise IP ← IP + 2 and execute next instruction.

There are two types of conditional branch instructions. In one type, the various relationships that exist between two numbers such as equal, above, below, less than, or greater than can be determined by the appropriate conditional branch instruction after a COMPARE instruction. These instructions can be used for both signed and unsigned numbers. When comparing signed numbers, terms such as "less than" and "greater than" are used. On the other hand, when comparing unsigned numbers, terms such as "below zero" or "above zero" are used.

Table 9.6 lists the 8086 signed and unsigned conditional branch instructions. Note that in Table 9.6 the instructions for checking which two numbers are "equal" or

TABLE 9.6 8086 Signed and Unsigned Conditional Branch Instructions

Signed		*Unsigned*	
Name	*Alternate Name*	*Name*	*Alternate Name*
JE disp 8 (JUMP if equal)	JZ disp8 (JUMP if result zero)	JE disp8 (JUMP if equal)	JZ disp8 (JUMP if zero)
JNE disp8 (JUMP if not equal)	JNZ disp 8 (JUMP if not zero)	JNE disp8 (JUMP if not equal)	JNZ disp8 (JUMP if not zero)
JG disp8 (JUMP if greater)	JNLE disp8 (JUMP if not less or equal)	JA disp8 (JUMP if above)	JNBE disp8 (JUMP if not below or equal)
JGE disp8 (JUMP if greater or equal)	JNL disp8 (JUMP if not less)	JAE disp8 (JUMP if above or equal)	JNB disp8 (JUMP if not below)
JL disp8 (JUMP if less than)	JNGE disp8 (JUMP if not greater or equal)	JB disp8 (JUMP if below)	JNAE disp8 (JUMP if not above or equal)
JLE disp8 (JUMP if less or equal)	JNG disp8 (JUMP if not greater)	JBE disp8 (JUMP if below or equal)	JNA disp8 (JUMP if not above)

TABLE 9.7 8086 Conditional Branch Instructions Affecting Individual Flags

JC disp8	JUMP if carry, i.e., CF = 1
JNC disp8	JUMP if no carry, i.e., CF = 0
JP disp8	JUMP if parity, i.e., PF = 1
JNP disp8	JUMP if no parity. i.e., PF = 0
JO disp8	JUMP if overflow, i.e., OF = 1
JNO disp8	JUMP if no overflow, i.e., OF = 0
JS disp8	JUMP if sign, i.e., SF = 1
JNS disp8	JUMP if no sign, i.e.. SF = 0
JZ disp8	JUMP if result zero, i.e.. ZF = 1
JNZ disp8	JUMP if result not zero, i.e., ZF = 0

TABLE 9.8 8086 Instructions To Be Used after `CMP A, B;` a and b are data in the following.

Signed "a" and "b"		*Unsigned "a" and "b"*	
JGE disp8	if a ≥ b	JAE disp8	if a ≥ b
JL disp8	if a < b	JB disp8	if a < b
JG disp8	if a > b	JA disp8	if a > b
JLE disp8	if a ≤ b	JBE disp8	if a ≤ b

"not equal" are the same for both signed and unsigned numbers. This is because when two numbers are compared for equality, irrespective of whether they are signed or unsigned, they will provide a zero result (ZF = 1) if they are equal and a nonzero result (ZF = 0) if they are not equal. Therefore, the same instructions apply for both signed and unsigned numbers for "equal to" or "not equal to" conditions. The second type of conditional branch instructions is concerned with the setting of flags rather than the relationship between two numbers. Table 9.7 lists these instructions.

Now, in order to check whether the result of an arithmetic or logic operation is zero, nonzero, positive or negative, did or did not produce a carry, did or did not produce parity, or did or did not cause overflow, the following instructions should be used: JZ, JNZ, JS, JNS, JC, JNC, JP, JNP, JO, JNO. However, in order to compare two signed or unsigned numbers (*a* in address A or *b* in address B) for various conditions, we use `CMP A, B`, which will form *a* - *b*. and then one of the instructions in Table 9.8.

Now let us illustrate the concept of using the preceding signed or unsigned instructions by an example. Consider clearing a section of memory word starting at B up to and including A, where (A) = 3000_{16} and (B) = 2000_{16} in DS = 1000_{16}, using the following instruction sequence:

```
        MOV   AX, 1000H
        MOV   DS, AX              ;Initialize DS
        MOV   BX, 2000H
        MOV   CX, 3000H
AGAIN:  MOV   WORD PTR[BX], 0000H
        INC   BX
        INC   BX
        CMP   CX, BX
        JGE   AGAIN
```

JGE treats CMP operands as twos complement numbers. The loop will terminate when BX = 3002H. Now, suppose that the contents of A and B are as follows: (A) = 8500_{16} , (B) = 0500_{16}

In this case, after `CMP CX, BX` is first executed,

$$\begin{aligned}(CX) - (BX) &= 8500 - 0500\\ &= 8000_{16}\\ &= 1000\ 0000\ 0000\ 0000\\ &\quad\uparrow\\ &\quad SF = 1, \text{ i.e., a negative number}\end{aligned}$$

Because 8000_{16} is a negative number, the loop terminates.

The correct approach is to use a branch instruction that treats operands as unsigned numbers (positive numbers) and uses the following instruction sequence:

```
        MOV   AX,1000H
        MOV   DS,AX              ;   initialize DS
        MOV   BX,0500H
        MOV   CX,8500H
AGAIN:  MOV   WORD PTR[BX],0000H
        INC   BX
        INC   BX
        CMP   CX,BX
        JAE   AGAIN
```

JAE will work regardless of the values of A and B.

Also, note that addresses are always positive numbers (unsigned). Hence, unsigned conditional jump instruction must be used to obtain the correct answer. The above examples are shown for illustrative purposes.

9.5.7 Iteration Control Instructions

Table 9.9 lists *iteration control instructions.* All these instructions have relative addressing modes.

LOOP disp8 decrements the CX register by 1 without affecting the flags and then acts in the same way as the JMP dsp8 instruction except that if CX ≠ 0, then the JMP is performed: otherwise, the next instruction is executed.

LOOPE (Loop while equal) / LOOPZ (Loop while zero), on the other hand, decrements CX by 1 without affecting the flags. The contents of CX are then checked for zero, and also the zero flag (ZF), that results from execution of previous instruction, is checked for one. If CX ≠ 0 and ZF = 1, the loop continues. If either CX = 0 or ZF = 0, the next instruction after the LOOPE or LOOPZ is executed. The following 8086 instruction sequence compares an array of 50 bytes with data byte 00H. As soon as a match is not found or end of array is reached, the loop exits. LOOPE instruction can be used for this purpose. The following 8086 instruction sequence illustrates this:

```
MOV   SI, START        ; Intialize SI with the starting
                       ; offset of the array
```

TABLE 9.9 8086 Iteration Control Instructions

LOOP disp8	Decrement CX by 1 without affecting flags and branch to label if CX ≠ 0; otherwise, go to the next instruction.
LOOPE/LOOPZ disp8	Decrement CX by 1 without affecting flags and branch to label if CX ≠ 0 and ZF = 1; otherwise (CX=0 or ZF=0), go to the next instruction.
LOOPNE/LOOPNZ disp8	Decrement CX by 1 without affecting flags and branch to label if CX ≠ 0 and ZF = 0; otherwise (CX=0 or ZF=1), go to the next instruction.
JCXZ disp8	JMP if register CX =0.

```
      DEC   SI
      MOV   CX,50            ; Initialize CX with array count
BACK: INC   SI               ; Update pointer
      CMP   BYTE PTR[SI],00H ; Compare array element with 00H
      LOOPE BACK
```

LOOPNE (LOOP while not equal) / LOOPNZ (Loop while not zero) is similar to LOOPE / LOOPZ except that the loop continues if CX ≠ 0 and ZF = 0. On the other hand, If CX = 0 or ZF = 1, the next instruction is executed. The following 8086 instruction sequence compares an array of 50 bytes with data byte 00H for a match. As soon as a match is found or end of array is reached, the loop exits. LOOPNE instruction can be used for this purpose. CX=0 and ZF=0 upon execution of the CMP instruction 50 times in the following would imply that data byte 00H was not found in the array. The following 8086 instruction illustrates this:

```
       MOV     SI, START        ; Intialize SI with the starting offset of
                                ; the array
       DEC     SI
       MOV     CX,50            ; Initialize CX with array count
BACK:  INC     SI               ; Update pointer
       CMP     BYTE PTR[SI],00H ; Compare array element with 00H
       LOOPNE  BACK
```

JCXZ START jumps to label START if CX = 0. This is normally used to skip a loop (instruction sequence arbitrarily chosen inside the loop) as follows:

```
          ---------------------
          ---------------------
          JCXZ      DOWN             ; If CX is already 0, skip
                                     ; the loop
BACK:     SUB       WORD PTR[SI], 4  ; Subtract 4 from the
                                     ; 16-bit contents of
                                     ; addressed by SI
          ADD       SI, 2            ; Update SI to point to
                                     ; next value
          LOOPBACK                   ; Decrement CX by 1 and
                                     ; Loop until
                                     ; CX = 0
DOWN:     ---------------------
          ---------------------
```

9.5.8 Interrupt Instructions

Table 9.10 shows the *interrupt instructions.* INT n is a software interrupt instruction. Execution of INT n causes the 8086 to push current CS, IP , and Flags onto the stack, and loads CS and IP with new values based on interrupt type n; an interrupt service routine is written at this new address. IRET at the end of the service routine transfers control to the main program by popping old CS, IP, and flags from the stack.

The interrupt on overflow is a type 4 (n = 4) interrupt. This interrupt occurs if the overflow flag (OF) is set and the INTO instruction is executed. The overflow flag

TABLE 9.10 8086 Interrupt Instructions

INT n	Software interrupt instructions
(n can be $0\text{-}255_{10}$)	(INT $32_{10} - 255_{10}$ available to the user.)
INTO	Interrupt on overflow
IRET	Interrupt return

is affected, for example, after execution of a signed arithmetic (such as IMUL, signed multiplication) instruction. The user can execute an INTO instruction after the IMUL. If there is an overflow, an error service routine written by the user at the type 4 interrupt address vector is executed.

Interrupt instructions are discussed in detail later in this Chapter.

9.5.9 Processor Control Instructions

Table 9.11 shows the *processor control functions.* Let us explain some of the instructions in Table 9.11.

- ESC mem places the contents of the specified memory location on the data bus at the time when the 8086 ready pin is asserted by the addressed memory device. This instruction is used to pass instructions to a coprocessor such as the 8087 math coprocessor which shares the address and data bus with the 8086.
- LOCK prefix allows the 8086 to ensure that another processor does not take control of the system bus while it is executing an instruction which uses the system bus. LOCK prefix is placed in front of an instruction so that when the instruction with the LOCK prefix is executed, the 8086 outputs a LOW on the LOCK pin of the 8086 for the duration of the next instruction. This Lock signal is connected to an external bus controller which prevents any other processor from taking over the system bus. Thus the LOCK prefix is used in multiprocessing.
- WAIT causes the 8086 to enter an idle state if the signal on the $\overline{TEST}$ input pin is not asserted. This means that the 8086 will remain in the idle state until its $\overline{TEST}$ pin is asserted. The WAIT instruction can be used to synchronize the 8086 with other external hardware such as the 8087 (Math coprocessor).

9.6 8086 Assembler-Dependent Instructions

Some 8086 instructions do not define whether an 8-bit or a 16-bit operation is to be executed. Instructions with one of the 8086 registers as an operand typically define the operation as 8-bit or 16-bit based on the register size. An example is MOV CL, [BX], which moves an 8-bit number with the offset defined by [BX] in DS into register CL; MOV CX, [BX], on the other hand, moves a 16-bit number from offsets (BX) and (BX + 1) in DS into CX

Instructions with a single-memory operand may define an 8-bit or a 16-bit operation by adding B for byte or W for word with the mnemonic. Typical examples are

TABLE 9.11 8086 Processor Control Instructions

STC	Set carry CF ← 1
CLC	Clear carry CF ← 0
CMC	Complement carry, CF ← $\overline{CF}$
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt enable flag
CLI	Clear interrupt enable flag
NOP	No operation
HLT	Halt
WAIT	Wait for $\overline{TEST}$ pin active
ESC mem	Escape to external processor
LOCK	Lock bus during next instruction

MULB [BX] and IDIVW [ADDR]. The string instructions may define this in two ways. Typical examples are MOVSB or MOVS BYTE for 8-bit and MOVSW or MOVS WORD for 16-bit. Memory offsets can also be specified by including BYTE PTR for 8-bit and WORD PTR for 16-bit with the instruction. Typical examples are INC BYTE PTR [BX] and INC WORD PTR [BX].

9.7 Typical 8086 Assembler Pseudo-Instructions or Directives

One of the requirements of typical 8086 assemblers such as MASM (discussed later) is that a variable's type must be declared as a byte (8-bit), word (16-bit), or double word (4 bytes or 2 words) before using the variable in a program. Some examples are as follows:

```
BEGIN DB 0        ;BEGIN is declared as a byte offset with contents zero.
START DW 25F1H    ;START is declared as a word offset with contents 25F1H.
PROG  DD 0        ;PROG is declared as a double word (4 bytes) offset with
                   zero contents.
```

Note that the directive DD is not used by all assemblers. In that case, one should use the directive DW twice to declare a 32-bit offset.

The EQU directive can be used to assign a name to constants. For example, the statement NUMB EQU 21H directs the assembler to assign the value 21H every time it finds NUMB in the program. This means that the assembler reads the statement MOV BH, NUMB as MOV BH, 21H. As mentioned before, DB, DW, and DD are the directives used to assign names and specific data types for variables in a program. For example, after execution of the statement ADDR DW 2050H the assembler assigns 50H to the offset name ADDR and 20H to the offset name ADDR + 1. This means that the program can use the instruction MOV BX, [ADDR] to load the 16-bit contents of memory starting at the offset ADDR in DS into BX. The DW sets aside storage for a word in memory and gives the starting address of this word the name ADDR.

As an example, consider 16 × 16 multiplication. The size of the product should be 32 bits and must be initialized to zero. The following will accomplish this:

```
Multiplicand    DW 2A05H
Multiplier      DW 052AH
Product         DD 0
```

Some versions of MASM assembler such as version 5.10 use directive AT to assign a value to an 8086 segment.

The 8086 addressing mode examples for the typical assemblers are given next:

`MOV AH, BL`	Both source and destination are in register mode.
`MOV CH, 8`	Source is in immediate mode and destination is in register mode.
`MOV AX, [START]`	Source is in memory direct mode and destination is in register mode.
`MOV CH, [BX]`	Source is in register indirect mode and destination is in register mode.
`MOV [SI], AL`	Source is in register mode and destination is in register indirect mode.
`MOV [DI], BH`	Source is in register mode and destination is in register indirect mode.

`MOV BH, VALUE [DI]`	Source is in register indirect with displacement mode and destination is in register mode. VALUE is typically defined by the `EQU` directive prior to this instruction.
`MOV AX, 4[DI]`	Source is in indexed with displacement mode and destination is in register mode.
`MOV SI, 2[BP] [DI]`	Source is in based indexed with displacement mode and destination is in register mode.
`OUT 30H, AL`	Source is in register mode and destination is in direct port mode.
`IN AX, DX`	Source is in indirect port mode and destination is in register mode.

In the following paragraphs, more assembler directives such as `SEGMENT`, `ENDS`, `ASSUME`, and `DUP` will be discussed.

9.7.1 SEGMENT and ENDS Directives

A section of a an 8086 program or a data array can be defined by the *SEGMENT and ENDS directives* as follows:

```
START    SEGMENT
X1       DB     0F1H
X2       DB     50H
X3       DB     25H
START    ENDS
```

The segment name is START (arbitrarily chosen). The assembler will assign a numeric value to START corresponding to the base value of the data segment. The programmer must use the 8086 instructions to load START into DS as follows:

```
MOV BX, START
MOV DS, BX
```

Note that all segment registers except CS must be loaded via a 16-bit general purpose register such as BX. A data array or an instruction sequence between the `SEGMENT` and `ENDS` directives is called a *logical segment.* These two directives are used to set up a logical segment with a specific name. A typical assembler allows one to use up to 31 characters for the name without any spaces. An underscore is sometimes used to separate words in a name, for example, `PROGRAM_BEGIN`.

9.7.2 ASSUME Directive

As mentioned before, at any time the 8086 can directly address four physical segments, which include a code segment, a data segment, a stack segment, and an extra segment. The 8086 may contain a number of logical segments containing codes, data, and stack. The `ASSUME` *directive* assigns a logical segment to a physical segment at any given time. That is, the `ASSUME` directive tells the assembler what addresses will be in the segment registers at execution time.

For example, the statement `ASSUME CS: PROGRAM_1,DS: DATA_1,SS: STACK_1` directs the assembler to use the logical code segment PROGRAM _1 as CS, containing the instructions, the logical data segment DATA_1 as DS, containing data, and the logical stack segment STACK _1 as SS, containing the stack.

9.7.3 DUP, LABEL, and Other Directives

The DUP *directive* can be used to initialize several locations to zero. For example, the statement START DW 4 DUP (0) reserves four words starting at the offset START in DS and initializes them to zero. The DUP directive can also be used to reserve several locations that need not be initialized. A question mark must be used with DUP in this case. For example, the statement BEGIN DB 100 DUP (?) reserves 100 bytes of uninitialized data space to an offset BEGIN in DS. Note that BEGIN should be typed in the label field, DB in the OP code field, and 100 DUP (?) in the operand field.

A typical example illustrating the use of these directives is given next:

```
DATA_1          SEGMENT
ADDR_1          DW 3005H
ADDR_2          DW 2003H
DATA_1          ENDS
STACK_1         SEGMENT
                DW 60 DUP (0)                ;     Assign 60 words
                                             ;     of stack with zeros.
STACK_TOP       LABEL WORD                   ;     Define stack
                                             ;     as 16-bit
STACK_1         ENDS                         ;     words.
CODE_1          SEGMENT
                ASSUME CS: CODE_1, DS: DATA_1,  SS: STACK_1
                MOV AX, STACK_1
                MOV SS, AX
                LEA SP, STACK_TOP
                MOV AX, DATA_1
                MOV DS, AX
                LEA SI, ADDR_1
                LEA DI, ADDR_2
                        -       ←        Main program
                        -       ←        body
CODE_1          ENDS    -
```

Note that LABEL is a directive used to the allocate stack from the next location after the top of the stack. The statement STACK_TOP LABEL WORD allocates the stack for local variables from the next address after STACK_TOP. In this example, 60 words are set aside for the stack. The WORD in this statement indicates that PUSH into and POP from the stack are done as words.

Also note that in the above, ASSUME directive tells the assembler to use the logical segment names CODE_1, DATA_1, and STACK_1 as the code segment, data segment, and stack segment, respectively. The extra segment can be assigned a name in a similar manner. When the instructions are executed, the displacements in the instructions along with the segment register contents are used by the assembler to generate the 20-bit physical addresses. The segment register, other than the code segment, must be initialized before it is used to access data. The code segment is typically initialized upon hardware reset or by using ORG.

When the assembler translates an assembly language program, it computes the displacement, or offset, of each instruction code byte from the start of a logical segment that contains it. For example, in the preceding program, the CS: CODE_1 in the ASSUME statement directs the assembler to compute the offsets or displacements by the following instructions from the start of the logical segment CODE_1. This means that when the program is run, the CS will contain the 16-bit value where the logical segment CODE_1 is located in memory. The assembler keeps track of the instruction byte displacements, which are loaded into IP. The 20-bit physical address generated from CS and IP are used

to fetch each instruction. Some versions of MASM use directive AT to assign a segment value.

Note that typical 8086 assemblers such as Microsoft and Hewlett-Packard HP64000 use the ORG directive to load CS and IP. For example, CS and IP can be initialized with 2000H and 0300H as follows:

For Microsoft 8086 Assembler (some versions) `ORG 20000300H`
For HP64000 8086 Assembler `ORG 2000H:0300H`

9.7.4 8086 Stack

Each 8086 stack segment is 64K bytes long and is organized as 32K 16-bit words. The lowest byte (valid data) of the stack is pointed to by the 20-bit physical address computed from current SP and SS. This is the lowest memory location in the stack (Top of the Stack) where data is pushed. The 8086 PUSH and POP instructions always utilize 16-bit words. Therefore, stack locations should be configured at even addrsesses in order to minimize the number of memory cycles for efficient stack operations. The 8086 can have several stack segments; however, only one stack segment is active at a time.

Since the 8086 uses 16-bit data for PUSH and POP operations from the top of the stack, the 8086 PUSH instruction first decrements SP by 2 and then the 16-bit data is written onto the stack. Therefore, the 8086 stack grows from high to low memory addresses of the stack. On the other hand, when a 16-bit data is popped from the top of the stack using the 8086 POP instruction , the 8086 reads 16-bit data from the stack into the specified register or memory, the 8086 then increments the SP by 2. Note that the 20-bit physical address computed from SP and SS always points to the last data pushed onto the stack. One can save and restore flags in the 8086 using PUSHF and POPF instructions. Memory locations can also be saved and restored using PUSH and POP instructions without using any 8086 registers. Finally, One must POP registers in the reverse order in which they are PUSHed. For example, if the registers BX, DX, and SI are PUSHed using

```
PUSH    BX
PUSH    DX
PUSH    SI
```

then the registers must be popped using

```
POP     SI
POP     DX
POP     BX
```

9.8 8086 Delay routine

Typical 8086 software delay loops can be written using MOV and LOOP instructions. For example, the following instruction sequence can be used for a delay loop of 20 millisecond:

```
        MOV     CX,count
DELAY:  LOOP    DELAY
```

The initial loop counter value of "count" can be calculated using the cycles required to execute the following 8086 instructions (Appendix F):

```
MOV     reg/imm (4 cycles)
LOOP    label (17/5 cycles)
```

Note that the 8086 LOOP instruction requires two different execution times. LOOP requires 17 cycles when the 8086 branches if the CX is not equal to zero after

autodecrementing CX by 1. However, the 8086 goes to the next instruction and does not branch when CX = 0 after autodecrementing CX by 1, and this requires 5 cycles. This means that the DELAY loop will require 17 cycles for (count - 1) times, and the last iteration will take 5 cycles.

For 2-MHz 8086 clock, each cycle is 500ns. For 20 ms, total cycles = $\frac{20\ m\,\text{sec}}{500\ n\,\text{sec}} =$ 40,000. The loop will require 17 cycles for (count - 1) times when CX $\neq$ 0 and 5 cycles will be required when no branch is taken (CX = 0). Thus, total cycles including the MOV = 4+17×(count - 1) + 5= 40,000. Hence, count $\cong 2353_{10} = 0931_{16}$. Therefore, CX must be loaded with 2353_{10} or 0931_{16}.

Now, in order to obtain delay of 20 seconds, the above DELAY loop of 20 millisecond can be used with an external counter. Counter value = (20 sec) / (20 msec) = 1000. The following instruction sequence will provide an approximate delay of 20 seconds:

```
        MOV   DX,1000   ;Initialize counter for 20 second delay
BACK:   MOV   CX,2353
DELAY:  LOOP  DELAY     ;20msec delay
        DEC   DX
        JNE   BACK
```

Next, the delay time provided by the above instruction sequence can be calculated. From Appendix F, the cycles required to execute the following 8086 instructions:

```
MOV reg / imm   (4 cycles)
DEC reg16       (2 cycles)
JNE             (16/4 cycles)
```

As before, assuming 4-MHz 8086 clock, each cycle is 250ns. Total time from the above instruction sequence for 20-second delay = Execution time for MOV DX + 1000 * (20 msec delay) + 1000 * (Execution time for DEC) + 999* (Execution time for JNE for Z = 0 when DX $\neq$ 0) + (Execution time for JNE for Z = 1 when DX = 0) = 4 * 250ns + 1000 * 20msec + 1000 * 2 * 250ns + 999 * 16 * 250ns + 4 * 250ns $\cong$ 20.0045 seconds which is approximately 20 seconds discarding the execution times of MOV DX, DEC, and JNE.

Example 9.1

(a) Determine the effect of each of the following 8086 instructions:
`i). DIV CH  ii). CBW  iii). MOVSW` Assume the following data prior to execution of each of these instructions independently (assume that all numbers are in hexadecimal): (DS) = 2000H, (ES) = 4000H, (CX) = 0300H, (AX) = 0091H, (20300H) = 05H, (20301H) = 02H, (40200H) = 06H, (40201H) = 07H, (SI) = 0300H, (DI) = 0200H, DF = 0.

(b) Write an 8086 assembly language program for each of the following C language program structures:

i). if (x >= y)
    x = x + 10;
  else y = y - 12;

Assume x and y are addresses of two 16-bit signed integers.

ii). sum = 0;
  for (i=0; i<=9; i=i+1)
  sum = sum + a[i];

Assume sum is the address of the 16-bit result.

Solution

(a)

i). Before unsigned division, CH contains 03_{10} and AX contains 145_{10}. Therefore, after `DIV CH`, (AH) = remainder = 01H and (AL) = quotient = 48_{10} = 30H.

ii). `CBW` sign-extends the AL register into the AH register. Because the content of AL is 91H, the sign bit is 1. Therefore, after `CBW`, (AX) = FF91H

iii). Before `MOVSW`,

Source String	Destination String
(SI) = 0300H, (DS) = 2000H	(DI) = 0200H, (ES) = 4000H
Physical address = 20300H	Physical address = 40200H

After `MOVSW`, (40200H) = 05H, (40201H) = 02H. Because DF = 0, (SI) = 0302H, (DI) = 0202H

(b)

i). Assume addresses x and y are initialized with the contents of the 8086 memory locations addressed by offsets BX and SI in segment register , DS:

```
        MOV AX,[BX]             ; Move [x] into AX
        CMP AX,[SI]             ; Compare [x] with [y]
        JGE TEN
        SUB WORD PTR[SI],12     ; Execute else part
        JMP FINISH
TEN:  ADD WORD PTR[BX],10       ; execute then part
FINISH: HLT                     ; Halt
```

ii). Assume register `SI` holds the address of the first element of the array while BX contains the offset of sum :

```
        MOV CX,10               ;initialize CX
        MOV WORD PTR [BX],0     ;sum = 0
AGAIN:  MOV AX,[SI]
        ADD  [BX],AX
        ADD  SI,2
        LOOP AGAIN
        HLT
```

Example 9.2

(a) Write an 8086 assembly program to find $(X^2)/255$ where X is an 8-bit signed number stored in CH. Store the 16-bit result onto the stack. Initialize SS and SP to 1000H and 2000H respectively.

(b) What are the remainder, quotient, and registers containing them after execution of the following 8086 instruction sequence?

```
        MOV     AH, 0FFH
        MOV     AL, 0FFH
        MOV     CX, 2
        IDIV    CL
```

Solution

(a)

```
CODE    SEGMENT
        ASSUME      CS:CODE, SS:STACK
        MOV         AX, 1000H           ; Initialize SS
        MOV         SS, AX              ; to 1000H
        MOV         SP, 2000;           ; Initialize SP to 2000H
        MOV         AL, CH              ; Move X into AL
        IMUL        CH                  ; Compute X² and store in AX
        MOV         CL, 255             ; Since X² and 255 are both positve, use
        DIV         CL                  ; unsigned division. Remainder in AH
        PUSH        AX                  ; and quotient in AL. Push AX to stack
        HLT
```

```
CODE   ENDS
STACK  SEGMENT
STACK  ENDS
```

(b)

```
        MOV     AH, 0FFH        ; AH = FFH
        MOV     AL, 0FFH        ; AL = FFH, hence AX = FFFFH = -1
        MOV     CX, 2           ; AX / CL = -1/2
        IDIV    CL
```

AH	AL
FFH	00H
8-bit remainder $= -1_{10}$	8-bit quotient = 0

Example 9.3

Write an 8086 assembly language program to add two 16-bit numbers in CX and DX and store the result in location 0500H addressed by DI.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              10/25/04 23:54:48
ex93.asm                                                 Page 1 - 1

 0000                           DATA    SEGMENT
 0000                           DATA    ENDS
 0000                           CODE    SEGMENT
                                ASSUME  CS:CODE,DS:DATA
 0000  B8 ---- R                        MOV     AX,DATA      ;Initialize DS
 0003  8E D8                            MOV     DS,AX
 0005  BF 0500                          MOV     DI,0500H
 0008  03 CA                            ADD     CX,DX ;Add
 000A  89 0D                            MOV     [DI],CX      ;Store
 000C  F4                               HLT
 000D                           CODE    ENDS
                                        END
Microsoft (R) Macro Assembler Version 6.11              10/25/04 23:54:48
ex93.asm                                                 Symbols 2 - 1
Segments and Groups:

                N a m e                 Size     Length   Align   Combine
Class

CODE . . . . . .                        16 Bit   000D     Para    Private
DATA . . . . . .                        16 Bit   0000     Para    Private

          0 Warnings
          0 Errors
```

Example 9.4

Write an 8086 assembly language program to add two 64-bit numbers. Assume SI and DI contain the starting offsets of the numbers. Store the result in memory pointed to by DI.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/08/04 23:20:22
ex94.asm                                                 Page 1 - 1

 0000                   PROG_CODE       SEGMENT
                                        ASSUME  CS:PROG_CODE, DS:DATA_ARRAY
 0000  B8 ---- R                        MOV     AX,DATA_ARRAY
```

```
0003  8E D8                      MOV     DS,AX       ;Initialize DS
0005  BA 0004                    MOV     DX,4        ;Load 4 into DX
0008  BE 0000                    MOV     SI,0000H    ;Initialize SI
000B  BF 0008                    MOV     DI,0008H    ;Initialize DI
000E  F8                         CLC                 ;Clear Carry
000F  8B 04           START:     MOV     AX,[SI]     ;Load DATA1
0011  11 05                      ADC     [DI],AX     ;Add with carry
0013  46                         INC     SI          ;Update pointers
0014  46                         INC     SI          ;by 2 for WORD
0015  47                         INC     DI          ;Update pointers
0016  47                         INC     DI          ;by 2 for WORD
0017  4A                         DEC     DX          ;decrement
0018  75 F5                      JNZ     START       ;branch
001A  F4                         HLT
001B                  PROG_CODE  ENDS
0000                  DATA_ARRAY SEGMENT
0000  0A71           DATA1      DW      0A71H       ;DATA1 low
0002  F218                      DW      0F218H
0004  2F17                      DW      2F17H       ;DATA1 high
0006  6200                      DW      6200H
0008  7A24           DATA2      DW      7A24H       ;DATA2 low
000A  1601                      DW      1601H
000C  152A                      DW      152AH       ;DATA2 high
000E  671F                      DW      671FH
0010                  DATA_ARRAY ENDS
                                END
Microsoft (R) Macro Assembler Version 6.11 11/08/04 23:20:22
ex94.asm                                              Symbols 2 - 1
Segments and Groups:
                N a m e          Size       Length      Align   Combine
                                                                  Class
DATA_ARRAY . . . . . .           16 Bit      0010       Para     Private
PROG_CODE  . . . . . .           16 Bit      001B       Para     Private
Symbols:
                N a m e          Type       Value       Attr
DATA1  . . . . . .               Word        0000       DATA_ARRAY
DATA2  . . . . . .               Word        0008       DATA_ARRAY
START  . . . . . .               L Near      000F       PROG_CODE

          0 Warnings
          0 Errors
```

Example 9.5

Write an 8086 assembly language program to multiply two 16-bit unsigned numbers to provide a 32-bit result. Assume that the two numbers are stored in CX and DX.

Solution

```
Microsoft (R) Macro Assembler Version 6.11      11/03/04 16:18:45
ex95.asm                                                 Page 1 - 1

 0000          CODE_SEG       SEGMENT
                              ASSUME CS:CODE_SEG
 0000  8B C2                  MOV AX,DX       ;Move first data
 0002  F7 E1                  MUL CX          ;[DX][AX]<--[AX]*[CX]
 0004  F4                     HLT
 0005          CODE_SEG       ENDS
                              END
Microsoft (R) Macro Assembler Version 6.11      11/03/04 16:18:45
ex95.asm                                                 Symbols 2 - 1
```

```
Segments and Groups:
                N a m e           Size  Length  Align   Combine
                                                                Class
CODE_SEG . . . . . . . . . . . .      16 Bit   0005     Para    Private

          0 Warnings
          0 Errors
```

Example 9.6

Write an 8086 assembly language program to clear 50_{10} consecutive bytes starting at offset 1000H. Assume DS is already initialized.

Solution

```
Microsoft (R) Macro Assembler Version 6.11   11/03/04 01:32:04
ex9_6.asm                                                 Page 1 - 1

 0000            CODE_SEG   SEGMENT
                            ASSUME   CS:CODE_SEG,DS:DATA_SEG
 0000  BB 1000              MOV      BX,1000H       ;initialize BX
 0003  B9 0032              MOV      CX,50          ;initialize loop count
 0006  C6 07 00 START:      MOV      BYTE PTR[BX],00H  ;clear memory byte
 0009  43                   INC      BX             ;update pointer
 000A  E2 FA                LOOP     START          ;decrement CX and loop
 000C  F4                   HLT                     ;halt
 000D            CODE_SEG   ENDS
 0000            DATA_SEG   SEGMENT
 0000            DATA_SEG   ENDS
                            END
Microsoft (R) Macro Assembler Version 6.11                    11/03/04
01:32:04
ex9_6.asm
Symbols 2 - 1
Segments and Groups:

                N a m e              Size   Length   Align   Combine Class

CODE_SEG . . . . . . . . . . . .      16 Bit   000D     Para    Private
DATA_SEG . . . . . . . . . . . .      16 Bit   0000     Para    Private
Symbols:
 Name                                  Type     Value    Attr

START  . . . . . . . . . . . . .      L Near   0006     CODE_SEG
          0 Warnings
          0 Errors
```

Example 9.7

Write an 8086 assembly program to implement the following C language program loop:

```
sum = 0;
for (i = 0; i <=99; i = i + 1)
sum =  sum + x[i] * y[i];
```

The assembly language program will compute $\sum x_i y_i$ where x_i and y_i are signed 8-bit numbers stored at offsets 4000H and 5000H respectively. Initialize DS to 2000H. Store 16-bit result in DX. Assume no overflow.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/03/04 13:44:38
ex97.asm                                                 Page 1 - 1

 0000             CODE    SEGMENT
```

```
                        ASSUME   CS:CODE,DS:DATA
0000  B8 2000           MOV      AX,2000H      ;Initialize
0003  8E D8             MOV      DS,AX         ;Data Segment
0005  B9 0064           MOV      CX,100        ;Initialize loop count
0008  BB 4000           MOV      BX,4000H      ;Initialize pointer of Xi
000B  BE 5000           MOV      SI,5000H      ;Initialize pointer of Yi
000E  BA 0000           MOV      DX,0000H      ;Initialize sum to 0
0011  8A 07      START: MOV      AL,[BX]       ;Load data into AL
0013  F6 2C             IMUL     BYTE PTR [SI] ;Signed 8x8 multiplication
0015  03 D0             ADD      DX,AX         ;Sum XiYi
0017  43                INC      BX            ;Update pointer
0018  46                INC      SI            ;Update pointer
0019  E2 F6             LOOP     START         ;Decrement CX & loop
001B  F4                HLT
001C            CODE    ENDS
0000            DATA    SEGMENT
0000            DATA    ENDS
                        END                    ;End program

Microsoft (R) Macro Assembler Version 6.11      11/03/04 13:44:38
ex97.asm                                         Symbols 2 - 1
Segments and Groups:
 N a m e                         Size   Length   Align   Combine Class

CODE . . . . . .                 16 Bit  001C     Para    Private
DATA . . . . . .                 16 Bit  0000     Para    Private
Symbols:
                N a m e           Type   Value    Attr
START  . . . . . .               L Near  0011     CODE
          0 Warnings
          0 Errors
```

Example 9.8

Write an 8086 assembly language program to add two words; each contains two ASCII digits. The first word is stored in two consecutive locations with the low byte pointed to by SI at offset 0300H, while the second word is stored in two consecutive locations with the low byte pointed to by DI at offset 0700H. Store the unpacked BCD result in memory location pointed to by DI. Assume that each unpacked BCD result of addition is less than or equal to 09H.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/09/04 12:00:57
9-8.asm                                                  Page 1 - 1


0000                    CODE    SEGMENT
                                ASSUME  CS:CODE,DS:DATA
0000  B8 2000                   MOV     AX,2000H        ;initialize
                                                        ;data segment
0003  8E D8                     MOV     DS,AX           ;at 2000H
0005  B9 0002                   MOV     CX,2            ;initialize
                                                        loop count
0008  BE 0300                   MOV     SI,0300H        ;initialize SI
000B. BF 0700                   MOV     DI,0700H        ;initialize DI
000E  8A 04             START:  MOV     AL,[SI]         ;load data into
                                                        ;AL
0010  02 05                     ADD     AL,[DI]         ;perform addition
0012  37                        AAA                     ;ASCII adjust
0013  88 05                     MOV     [DI],AL         ;store result
```

```
 0015  46                      INC     SI              ;update pointer
 0016  47                      INC     DI              ;update pointer
 0017  E2 F5                   LOOP    START           ;decrement CX &
                                                       ;loop
 0019  F4                      HLT                     ;halt
 001A                  CODE    ENDS
 0000                  DATA    SEGMENT
 0000                  DATA    ENDS
                               END
Microsoft (R) Macro Assembler Version 6.11              11/09/04 12:00:57
9-8.asm                                                  Symbols 2 - 1
Segments and Groups:
  N a m e                              Size  Length   Align Combine Class
CODE . . . . . . . . . . . . . .      16 Bit  001A     Para    Private
DATA . . . . . . . . . . . . . .      16 Bit  0000     Para    Private
Symbols:
 N a m e                              Type    Value    Attr
START  . . . . . . . . . . . . .      L Near  000E     CODE
          0 Warnings
          0 Errors
```

Example 9.9

Write an 8086 assembly language program to compare a source string of 50_{10} words pointed to by an offset 1000H in the data segment at 2000H with a destination string pointed to by an offset 3000H in the extra segment at 4000H. The program should be halted as soon as a match is found or the end of the string is reached.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/06/04 15:09:33
E9_9.ASM                                                 Page 1 - 1

 0000           CODE    SEGMENT
                        ASSUME CS:CODE,DS:DATA,ES:DATA1
 0000  B8 2000          MOV AX,2000H    ;Initialize
 0003  8E D8            MOV DS,AX       ;Data Segment at 2000H
 0005  B8 4000          MOV AX,4000H    ;Initialize
 0008  8E C0            MOV ES,AX       ;ES at 4000H
 000A  BE 1000          MOV SI,1000H    ;Initialize SI at 1000H FOR DS
 000D  BF 3000          MOV DI,3000H    ;Initialize DI AT 3000H FOR ES
 0010  B9 0032          MOV CX,50       ;Initialize CX
 0013  FC               CLD             ;Clear DF SO THAT
                                        ;SI and DI will
                                        ;autoincrement
                                        ;after compare
 0014  F2/ A7           REPNE   CMPSW   ;Repeat CMPSW until CX=0 or
                                        ;until compared words are equal
 0016  F4               HLT     ;Halt
 0017           CODE    ENDS
 0000           DATA1   SEGMENT
 0000           DATA1   ENDS
 0000           DATA    SEGMENT
 0000           DATA    ENDS
                        END     ;End program

Microsoft (R) Macro Assembler Version 6.11              11/06/04 15:09:33
E9_9.ASM                                                 Symbols 2 - 1
Segments and Groups:
  N a m e                              Size  Length   Align  Combine Class
CODE . . . . . . . . . . . . . .      16 Bit  0017     Para    Private
```

```
DATA1  . . . . . .  16 Bit  0000  Para  Private
DATA . . . . . .  16 Bit  0000  Para  Private
       0 Warnings
       0 Errors
```

Example 9.10

Write a subroutine in 8086 assembly language which can be called by a main program in the same code segment. The subroutine will multiply a signed 16-bit number in CX by a signed 8-bit number in AL. The main program will perform initializations (DS to 5000H, SS to 6000H, SP to 0020H and BX to 2000H), call this subroutine, store the result in two consecutive memory words, and stop. Assume SI and DI contain pointers to the signed 8-bit and 16-bit data respectively. Store 32-bit result in a memory location pointed to by BX.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/09/04 12:31:12
9-10.asm                                                 Page 1 - 1

 0000            CODE   SEGMENT
                        ASSUME  CS:CODE, DS:DATA,SS:STACK
 0000  B8 5000          MOV     AX, 5000H       ; Initialize Data Segment at
 0003  8E D8            MOV     DS, AX          ; 5000H
 0005  B8 6000          MOV     AX, 6000H       ; Initialize SS at
 0008  8E D0            MOV     SS, AX          ; 6000H
 000A  BC 0020          MOV     SP, 0020H       ; Initialize SP at 0020H
 000D  BB 2000          MOV     BX, 2000H       ; Initialize BX at 2000H
 0010  BE 0000          MOV     SI, 0000H       ; Initialize SI
 0013  BF 0004          MOV     DI, 0004H       ; Initialize DI
 0016  8A 04            MOV     AL, [SI]        ; Move 8-bit data
 0018  8B 0D            MOV     CX, [DI]        ; Move 16-bit data
 001A  E8 0006          CALL    MULTI           ; Call MULTI subroutine
 001D  89 17            MOV     [BX], DX        ; Store high word of result
 001F  89 47 02         MOV     [BX+2], AX      ; Store low word of result
 0022  F4               HLT                     ; Halt
 0023          MULTI    PROC    NEAR            ; Must be called from
 0023  98               CBW                     ; Sign extend AL
 0024  F7 E9            IMUL    CX              ; [DX] [AX] < - - [AX]*[CX]
 0026  C3               RET                     ; Return
 0027          MULTI    ENDP                    ; End of procedure
 0027          CODE     ENDS
 0000          DATA     SEGMENT
 0000          DATA     ENDS
 0000          STACK    SEGMENT
 0000          STACK    ENDS
                        END
Àicrosoft (R) Macro Assembler Version 6.11              11/09/04 12:31:12
9-10.asm                                                 Symbols 2 - 1
Segments and Groups:
   N a m e                         Size     Length   Align   Combine
Class
CODE . . . . . .  16 Bit  0027  Para  Private
DATA . . . . . .  16 Bit  0000  Para  Private
STACK  . . . . . .  16 Bit  0000  Para  Private
Procedures,  parameters and locals:
   N a m e                 Type  Value   Attr
MULTI  . . . . . .P Near 0023   CODE  Length= 0004 Private
         0 Warnings
         0 Errors
```

Example 9.11

Write an 8086 assembly program that converts a temperature (signed) from Fahrenheit degrees stored at an offset contained in SI to Celsius degrees. The program stores the 8-bit integer part of the result at an offset contained in DI. Assume that the temperature can be represented by one byte and, DS is already initialized. The source byte is assumed to reside at offset 2000H in the data segment, and the destination byte at an offset of 3000H in the same data segment. Use the formula: C = (F-32)/9 x 5

Solution

```
Microsoft (R) Macro Assembler Version 6.11                  11/10/04 14:28:58
9-11.asm                                                     Page 1 - 1
 0000                CODE    SEGMENT
                             ASSUME CS:CODE,DS:DATA
 0000  BE 2000               MOV     SI,2000H        ; Initialize source pointer
 0003  BF 3000               MOV     DI,3000H        ; Unit. destination pointer
 0006  8A 04                 MOV     AL,[SI]         ; Get degrees F
 0008  98                    CBW                     ; Sign extend
 0009  83 E8 20              SUB     AX,32           ; Subtract 32
 000C  B9 0005               MOV     CX,5            ; Get multiplier
 000F  F7 E9                 IMUL    CX              ; Multiply by 5
 0011  B9 0009               MOV     CX,9            ; Get divisor
 0014  F7 F9                 IDIV     CX             ; Divide by 9 to get
                                                     ; Celsius
 0016  88 05                 MOV     [DI],AL         ; Put result in destination
 0018  F4                    HLT                     ; Stop
 0019                CODE    ENDS                    ; End segment
 0000                DATA    SEGMENT
 0000                DATA    ENDS
                              END
Microsoft (R) Macro Assembler Version 6.11                  11/10/04 14:28:58
9-11.asm                                                     Symbols 2 - 1
Segments and Groups:
 N a m e                                 Size     Length   Align   Combine
Class
CODE . . . . . . . . . . . . . .         16 Bit   0019     Para    Private
DATA . . . . . . . . . . . . . .         16 Bit   0000     Para    Private
0 Warnings
          0 Errors
```

Example 9.12

Write an 8086 assembly language program to multiply two 8 bit signed numbers stored in the same register; AH holds one number and AL holds the other number. Store the 16-bit result in DX.

Solution

```
Microsoft (R) Macro Assembler Version 6.11                  10/24/04 13:19:45
EX10_12.ASM                                                  Page 1 - 1


 0000                PROG_CODE       SEGMENT
                                     ASSUME  CS:PROG_CODE,DS
 0000  F6 EC                         IMUL    AH              ;(AH)*(AL)-->(AX)
 0002  8B D0                         MOV     DX,AX           ;Store result in DX
 0004  F4                            HLT
 0005                PROG_CODE       ENDS
                                     END

Microsoft (R) Macro Assembler Version 6.11                  10/24/04 13:19:45
```

```
EX10_12.ASM                                              Symbols 2 - 1
Segments and Groups:
   N a m e                          Size   Length   Align  Combine Class
PROG_CODE  . . . . . . . . . . .    16 Bit   000A    Para    Private
              0 Warnings
              0 Errors
```

Example 9.13

Write an 8086 assembly language program to move a block of 16-bit data of length 100_{10} from the source block starting at offset 0200H to the destination block starting at offset 0300H from low to high addresses.

Solution

```
Microsoft (R) Macro Assembler Version 6.11                11/16/04 16:31:36
EX913.ASM                                                  Page 1 - 1

 0000                CODE    SEGMENT
                             ASSUME CS:CODE, DS:DATA, ES:DATA1
 0000  B8 1000               MOV     AX, 1000H       ;INITIALIZE DS
 0003  8E D8                 MOV     DS, AX
 0005  BB 2000               MOV     BX, 2000H       ;INITIALIZE ES
 0008  8E C3                 MOV     ES, BX
 000A  BE 0200               MOV     SI, 0200H       ;INITIALIZE SOURCE
 000D  BF 0300               MOV     DI, 0300H       ;INITIALIZE DESTINATION
                                                     POINTERS
 0010  B9 0064               MOV     CX, 100         ;INITIALIZE LOOP COUNTER
 0013  FC                    CLD                     ;CLEAR DF FOR LOW
                                                     ;TO HIGH ADDRESS
 0014  F3/ A5                REP MOVSW               ;MOVE STRING WORD
 0016  F4                    HLT
 0017                CODE    ENDS
 0000                DATA    SEGMENT
 0000                DATA    ENDS
 0000                DATA1   SEGMENT
 0000                DATA1   ENDS
                             END

Microsoft (R) Macro Assembler Version 6.11                11/16/04 16:31:36
EX913.ASM                                                  Symbols 2 - 1

Segments and Groups:

                N a m e                 Size   Length   Align Combine Class

CODE . . . . . . . . . . . . . .        16 Bit   0017    Para    Private
DATA1  . . . . . . . . . . . . .        16 Bit   0000    Para    Private
DATA . . . . . . . . . . . . . .        16 Bit   0000    Para    Private

          0 Warnings
          0 Errors
```

Example 9.14

Write an 8086 assembly language program that will perform : $5 \times X + 6 \times Y + (Y/8) \rightarrow$ (BP)(BX) where X is an unsigned 8-bit number stored at offset 0100H and Y is a 16-bit signed number stored at offsets 0200H and 0201H. Neglect the remainder of Y/8. Store the result in registers BX and BP. BX holds the low 16-bit of the 32-bit result and BP holds the high 16-bit of the 32-bit result.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/16/04 15:36:15
9-14.asm                                                 Page 1 - 1

 0000                CODE    SEGMENT
                             ASSUME CS:CODE, DS:DATA
 0000  B8 1000               MOV     AX, 1000H       ;Initialize DS
 0003  8E D8                 MOV     DS, AX
 0005  BE 0100               MOV     SI, 0100H       ;Pointer to X
 0008  BF 0200               MOV     DI, 0200H       ;Pointer to Y
 000B  8A 04                 MOV     AL, [SI]        ;Move X to AL
 000D  BB 0000               MOV     BX, 0           ;Clear 16-bit sum to zero
 0010  B1 05                 MOV     CL, 5
 0012  F6 E1                 MUL     CL              ;Unsigned MUL
                                                     ;[AX] = 5*X
 0014  03 D8                 ADD     BX, AX          ;Sum 5*X with BX
 0016  BD 0000               MOV     BP, 0           ;Convert 5*X to unsigned
                                                     ;32-bit
 0019  8B 05                 MOV     AX, [DI]        ;Move Y to AX
 001B  B1 03                 MOV     CL, 3
 001D  D3 F8                 SAR     AX, CL          ;Divide by 8
 001F  99                    CWD                     ;Convert Y/8 into 32-
                                                     ;bit in [DX][AX]
 0020  03 D8                 ADD     BX, AX          ;Sum 5*X and Y/8
 0022  13 EA                 ADC     BP, DX          ;in BP Bx
 0024  8B 05                 MOV     AX, [DI]        ;Move Y to AX
 0026  B9 0006               MOV     CX, 6
 0029  F7 E9                 IMUL    CX              ;[DX][AX] <- 6*Y
 002B  03 D8                 ADD     BX, AX          ;32-bit result
 002D  13 EA                 ADC     BP, DX          ;in BP BX
 002F  F4                    HLT                     ;Halt
 0030                CODE    ENDS
 0000                DATA    SEGMENT
 0000                DATA    ENDS
                             END


Àicrosoft (R) Macro Assembler Version 6.11              11/16/04 15:36:15
9-14.asm                                                 Symbols 2 - 1

Segments and Groups:
   N a m e                         Size   Length   Align Combine Class
CODE . . . . . . . . . . . . . .   16 Bit  0030     Para    Private
DATA . . . . . . . . . . . . . .   16 Bit  0000     Para    Private
                0 Warnings
                0 Errors
```

Example 9.15

Write an 8086 assembly language program to add four 16-bit numbers stored in consecutive locations starting at offset 5000H. Store the 16-bit result onto the stack. Use ADC instruction for addition.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/10/04 16:14:38
9-15.asm                                                 Page 1 - 1

 0000                CODE    SEGMENT
                             ASSUME CS:CODE, DS:DATA, SS:STACK
 0000  B8 ---- R             MOV     AX, DATA        ; Initialize AX
```

```
0003  8E D8           MOV     DS, AX          ; Initialize DS
0005  B8 0000         MOV     AX, 0000H       ; Initialize AX
0008  8E D0           MOV     SS, AX          ; Initialize SS at 0000H
000A  BC 2000         MOV     SP, 2000H       ; Initialize SP at 2000H
000D  BB 5000         MOV     BX, 5000H       ; Initialize BX at 5000H
0010  B9 0004         MOV     CX, 4           ; Initialize loop count
0013  F8              CLC                     ; clear carry
0014  13 07   START:  ADC     AX, [BX]        ; Add
0016  43              INC     BX              ; Update pointer. INC does not
                                              ; affect CF
0017  43              INC     BX              ; Update pointer
0018  E2 FA           LOOP    START           ; Decrement CX & loop
001A  50              PUSH    AX              ; Storing 16-bit result onto
                                              ; the stack
001B  F4              HLT                     ; Stop
001C          CODE    ENDS                    ; End segment
0000          DATA    SEGMENT
0000          DATA    ENDS
0000          STACK   SEGMENT
0000          STACK   ENDS
                      END
Microsoft (R) Macro Assembler Version 6.11              11/10/04 16:14:38
9-15.asm                                                 Symbols 2 - 1
Segments and Groups:
 N a m e                         Size   Length   Align Combine Class
CODE . . . . . .                 16 Bit  001C     Para    Private
DATA . . . . . .                 16 Bit  0000     Para    Private
STACK  . . . . . .               16 Bit  0000     Para    Private
Symbols:
   N a m e                      Type     Value    Attr
START  . . . . . .               L Near   0014     CODE
          0 Warnings
          0 Errors
```

Example 9.16

Write a subroutine in 8086 assembly language in the same code segment as the main program to implement the C language assignment statement: p = p + q; where addresses p and q hold two 16-digit (64-bit) packed BCD numbers (N1 and N2). The main program will initialize addresses p and q to DS:2000H and DS:3000H respectively. Address DS:2007H will hold the lowest byte of N1 with the highest byte at address DS:2000H while address DS:3007H will hold the lowest byte of N2 with the highest byte at address DS:3000H. Also, write the main program at offset 7000H which will perform all initializations including DS to 2000H, SS to 6000H, SP to 0020H, SI to 2000H, DI to 3000H, loop count to 8 and, then call the subroutine.

Solution

```
Microsoft (R) Macro Assembler Version 6.11              11/29/04 00:37:06
ex916.asm                                                Page 1 - 1

0000          CODE    SEGMENT
                      ASSUME CS:CODE,DS:DATA,SS:STACK
0000  B8 2000         MOV     AX,2000H        ;Initialize Data segment at
                                              ;2000H
0003  8E D8           MOV     DS,AX
0005  B8 6000         MOV     AX,6000H        ;Initialize Stack segment at
                                              ;6000H
0008  8E D0           MOV     SS,AX
000A  BC 0020         MOV     SP,0020H        ;Initialize SP at 0020H
```

```
000D  B9 0008          MOV     CX,8            ;Initialize Count
0010  BE 2000          MOV     SI,2000H        ;Initialize pointer to N1 -> q
0013  BF 3000          MOV     DI,3000H        ;Initialize pointer to N2 -> p
0016  B8 0000          MOV     AX,0000H        ;Clear AX
0019  E8 0001          CALL    PBCD            ;Call PBCD subroutine
001C  F4               HLT
001D            PBCD    PROC NEAR
001D  F8               CLC                     ;Clear Carry
001E  8A 04    START:  MOV     AL,[SI]         ;Move Data to AL
0020  8A 1D            MOV     BL,[DI]         ;Move Data to AL
0022  12 C3            ADC     AL,BL           ;Add ASCII into AL
0024  27               DAA                     ;BCD adjust [AL]
0025  88 05            MOV     [DI],AL         ;Store result in [DI]
0027  46               INC     SI              ;Update pointers
0028  47               INC     DI              ;Update pointers
0029  E2 F3            LOOP    START
002B  C3               RET                     ;Return
002C            PBCD    ENDP
002C            CODE    ENDS
0000            DATA    SEGMENT
0000            DATA    ENDS
0000            STACK   SEGMENT
0000            STACK   ENDS
                        END
Microsoft (R) Macro Assembler Version 6.11              11/29/04 00:37:06
ex916.asm                                                Symbols 2 - 1

Segments and Groups:
                N a m e                 Size    Length   Align Combine Class
CODE . . . . . . . . . . . . . .        16 Bit  002C     Para    Private
DATA . . . . . . . . . . . . . .        16 Bit  0000     Para    Private
STACK  . . . . . . . . . . . . .        16 Bit  0000     Para    Private

Procedures,  parameters and locals:

                N a m e         Type    Value    Attr
PBCD . . . . . . . . . . . .    P Near  001D     CODE  Length= 000F Private

Symbols:
                N a m e         Type    Value   Attr
START  . . . . . . . . . . .    L Near  001E     CODE

          0 Warnings
          0 Errors
```

Example 9.17

Write an 8086 assembly language program to move the 8-bit contents of a memory location addressed by the contents of AL and BX into AL. Use XLAT instruction. This program will illustrate that XLAT is equivalent to MOV AL, [AL][BX].

Solution

```
0000            CODE    SEGMENT
                        ASSUME CS:CODE,DS:DATA
0000  B8 2030           MOV     AX, 2030H       ;Initialize
0003  8E D8             MOV     DS, AX          ;Data segment register
0005  B0 31             MOV     AL, 31H         ;Overwrite low byte of
                                                ;AX with 31H
0007  BB 2000           MOV     BX, 2000H       ;Store value 2000 in hex
```

```
                                                ;into BX
000A  D7                XLAT                   ;[AL] <- [AL] + [BX]
000B  F4                HLT                    ;Halt
000C              CODE   ENDS
0000              DATA   SEGMENT
0000              DATA   ENDS
                         END
Microsoft (R) Macro Assembler Version 6.11                11/03/04 13:16:50
9-17.asm                                                   Symbols 2 - 1
Segments and Groups:
N a m e                                  Size  Length   Align Combine Class
CODE . . . . . . . . . . . . . .      16 Bit  000C     Para    Private
DATA . . . . . . . . . . . . . .      16 Bit  0000     Para    Private
                 0 Warnings
                 0 Errors
```

Example 9.18

Write a subroutine in 8086 assembly language which can be called by a main program in a different code segment. The subroutine will compute $\sum X_i^2 / N$. Assume the X_i's are 16-bit signed integers, N = 100 and, $\sum X_i^2$ is 32-bit wide. The numbers are stored in consecutive locations. Assume SI points to the X_i's. The subroutine will start at an offset 7000H, and will initialize SI to 4000H, compute $\sum X_i^2 / N$, and store 32-bit result in DX:AX (16-bit remainder in DX and 16-bit quotient in AX). Also, write the main program which will initialize DS to 2000H, SS to 6000H, SP to 0040H, call the subroutine, and stop.

Solution

```
Microsoft (R) Macro Assembler Version 6.11                11/29/04 00:05:33
ex918.asm                                                  Page 1 - 1

 0000             CODE    SEGMENT
                          ASSUME CS:CODE,DS:DATA,SS:STACK
 0000  B8 2000            MOV     AX,2000H        ;Initialize Data segment at
                                                   ;2000H
 0003  8E D8              MOV     DS,AX
 0005  B8 6000            MOV     AX,6000H        ;Initialize Stack segment at
                                                   ;6000H
 0008  8E D0              MOV     SS, AX
 000A  BC 0040            MOV     SP,0040H
 000D  9A ---- 7000 R     CALL    FAR PTR SQRDIV ;Call SQRDIV subroutine
 0012  F4                 HLT
 0013             CODE    ENDS
 0000             SUBR    SEGMENT
                          ORG     7000H
                          ASSUME CS:SUBR
 7000             SQRDIV  PROC FAR
 7000  B9 0064            MOV     CX,100          ;Initialize CX to 100
 7003  BB 0000            MOV     BX,0000H        ;Clear low 16-bits sum to zero
 7006  BE 4000            MOV     SI,4000H        ;Initialize pointer of Xi
 7009  BF 3000            MOV     DI,3000H        ;High 16-bits sum
 700C  C7 05 0000         MOV     [DI],0000H      ;Clear contents of DI to zero
 7010  8B 04    START:     MOV     AX,[SI]         ;Load data into AX
 7012  F7 2C              IMUL    WORD PTR [SI]   ;Signed multiplication Xi*Xi
 7014  F8                CLC                     ;Clear Carry Flag
 7015  13 D8              ADC     BX,AX           ;Add low 16-bits to sum
 7017  11 15              ADC     [DI],DX         ;Add high 16-bits to sum
 7019  46                INC     SI              ;Update pointer
 701A  46                INC     SI              ;Twice for WORD
```

```
 701B  E2 F3           LOOP    START          ;Jump and decrement CX
 701D  8B 15           MOV     DX,[DI]        ;Place high 16-bits of sum
                                              ;to DX
 701F  8B C3           MOV     AX,BX          ;Place low 16-bits of sum
                                              ;to AX
 7021  B9 0064         MOV     CX,100         ;Load 100 into CX
 7024  F7 F1           DIV     CX             ;unsigned division DX:AX / CX
 7026  CB              RET                    ;Return
 7027          SQRDIV  ENDP
 7027          SUBR    ENDS
 0000          DATA    SEGMENT
 0000          DATA    ENDS
 0000          STACK   SEGMENT
 0000          STACK   ENDS
                       END
Microsoft (R) Macro Assembler Version 6.11              11/29/04 00:05:33
ex918.asm                                                Symbols 2 - 1

Segments and Groups:

                N a m e                 Size   Length   Align  Combine Class
CODE . . . . . . . . . . . . . .        16 Bit  0013     Para   Private
DATA . . . . . . . . . . . . . .        16 Bit  0000     Para   Private
STACK  . . . . . . . . . . . . .        16 Bit  0000     Para   Private
SUBR . . . . . . . . . . . . . .        16 Bit  7027     Para   Private


Procedures,  parameters and locals:

                N a m e         Type    Value   Attr
SQRDIV . . . . . . . . . . . . P Far   7000    SUBR  Length= 0027 Private

Symbols:
                N a m e         Type    Value   Attr
START  . . . . . . . . . . . . L Near  7010    SUBR

          0 Warnings
          0 Errors
```

Note: In the above, DIV is used for computing sum (X_i**2)/N since both SUM (X_i**2) and N are unsigned (positive). Also, in order to execute the above program, values for X_i must be stored in memory using 8086 assembler directive, DW.

9.9 System Design Using the 8086

This section covers the basic concepts associated with interfacing the 8086 with its support chips such as memory and I/O. Topics such as timing diagrams and 8086 pins and signals will also be included. Appendix E provides data sheets for Intel 8086 and support chips.

9.9.1 8086 Pins and Signals

The 8086 pins and signals are shown in Figure 9.8. As mentioned before, the 8086 can operate in two modes. These are the minimum (uniprocessor systems with a single 8086) and maximum mode (multiprocessor system with more than one 8086). MN/$\overline{MX}$ is an input pin used to select one of these modes.

When MN/$\overline{MX}$ is HIGH, the 8086 operates in the minimum mode. In this mode, the 8086

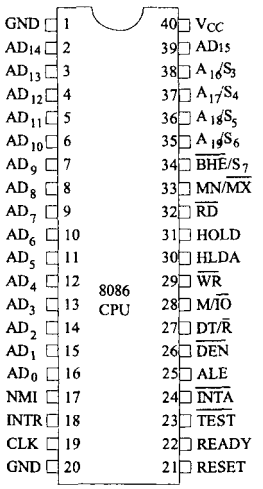


FIGURE 9.8 8086 Pin Diagram

is configured (that is, pins are defined) to support small single-processor systems using a few devices that use the system bus. When MN/$\overline{MX}$ is low, the 8086 is configured (that is, some of the pins are redefined in maximum mode) to support multiprocessor systems. In this case, the Intel 8288 bus controller is added to the 8086 to provide bus control and compatibility with the multibus architecture. Note that, in a particular application, MN/$\overline{MX}$ must be tied to either HIGH or LOW.

The AD_0–AD_{15} lines are a 16-bit multiplexed address/data bus. During the first clock cycle, AD_0–AD_{15} are the low-order 16-bit address. The 8086 has a total of 20 address lines. The upper four lines, A_{16}/S_3, A_{17}/S_4, A_{18}/S_5, and A_{19}/S_6, are multiplexed with the status signals for the 8086. During the first clock period of a bus cycle (read or write cycle), the entire 20-bit address is available on these lines. During all other cycles for memory and I/O, AD_0–AD_{15} lines contain the 16-bit data, and the multiplexed address / status lines become S_3, S_4, S_5, and S_6. S_3 and S_4 are decoded as follows:

A_{17}/S_4	A_{16}/S_3	Function
0	0	Extra segment
0	1	Stack segment
1	0	Code or no segment
1	1	Data segment

Therefore, after the first clock cycle of an instruction execution, the A_{17}/S_4 and A_{16}/S_3 pins specify which segment register generates the segment portion of the 8086 address. Thus, by decoding these pins and then using the decoder outputs as chip selects for memory chips, up to four megabytes (one megabyte per segment) can be included. This provides a degree of protection by preventing erroneous write operations to one segment from overlapping onto another segment and destroying the information in that segment. A_{18}/S_5 and A_{19}/S_6 are used as A_{18} and A_{19}, respectively, during the first clock cycle of an instruction execution. If an I/O instruction is executed, they stay LOW for the first clock period. During all other cycles, A_{18}/S_5 indicates the status of the 8086 interrupt enable flag

and A_{19}/S_6 becomes S_6; a LOW S_6 pin indicates that the 8086 is on the bus. During a hold acknowledge clock period, the 8086 tristates the A_{19}/S_6 pin and this allows another bus master to take control of the system bus. The 8086 tristates AD_0–AD_{15} during interrupt acknowledge or hold acknowledge cycles.

$\overline{BHE}/S_7$ is used as $\overline{BHE}$ (bus high enable) during the first clock cycle of an instruction execution. The 8086 outputs a LOW on this pin during the read, write, and interrupt acknowledge cycles in which data are to be transferred in a high-order byte (AD_{15}–AD_8) of the data bus. $\overline{BHE}$ can be used in conjunction with AD_0 to select memory banks. A thorough discussion is provided later. During all other cycles, $\overline{BHE}/S_7$ is used as S_7 and the 8086 maintains the output level ($\overline{BHE}$) of the first clock cycle on this pin. S_7 is the same as $\overline{BHE}$ and does not have any special meaning.

$\overline{TEST}$ is an input pin and is only used by the WAIT instruction. The 8086 enters a wait state after execution of the WAIT instruction until a low is seen on the $\overline{TEST}$ pin. This input is synchronized internally during each clock cycle on the leading edge of the clock.

INTR is the maskable interrupt input. This line is not latched, so INTR must be held at a HIGH level until it is recognized to generate an interrupt.

NMI is the nonmaskable interrupt pin input activated by a positive edge.

RESET is the system reset input signal. This signal must be HIGH for at least four clock cycles to be recognized, except on power-on, which requires a 50-μsec reset pulse. It causes the 8086 to initialize registers DS, ES, SS, IP, and flags to zeros. It also initializes CS to FFFFH. Upon removal of the RESET signal from the RESET pin, the 8086 will fetch its next instruction from a 20-bit physical address FFFF0H (CS = FFFFH, IP = 0000H). When the 8086 detects a positive edge of a pulse on RESET, it stops all activities until the signal goes LOW. Upon hardware reset, the 8086 initializes the system as follows:

8086 Components	Content
Flags	Clear
IP	0000H
CS	FFFFH
DS	0000H
SS	0000H
ES	0000H
Queue	Empty

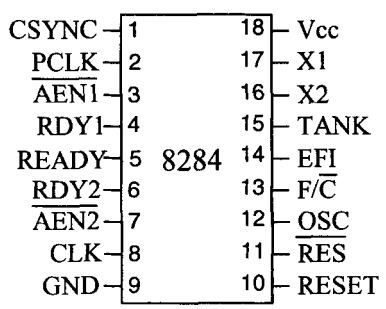
As mentioned before, the 8086 can be configured in either minimum or maximum mode using the MN/$\overline{MX}$ input pin. In minimum mode, the 8086 itself generates all bus control signals. These signals are as follows:

- DT/$\overline{R}$ (data transmit/receive) is an output signal required in a minimum system that uses an 8286/8287 data bus transceiver. It is used to control direction of data flow through the transceiver.
- $\overline{DEN}$ (data enable) is provided as an output enable for the 8286/8287 in a minimum system that uses the transceiver. $\overline{DEN}$ is active LOW during each memory and I/O access and for $\overline{INTA}$ cycles.
- ALE (address latch enable) is an 8086 output signal that can be used to demultiplex the multiplexed 8086 pins including AD_0–AD_{15} into A_0–A_{15} and D_0–D_{15} at the falling

edge of ALE.

- M/$\overline{IO}$ is an 8086 output signal. It is used to distinguish a memory access (M/$\overline{IO}$ = HIGH) from an I/O access (M/$\overline{IO}$ = LOW). When the 8086 executes an I/O instruction such as IN or OUT, it outputs a LOW on this pin. On the other hand, the 8086 outputs HIGH on this pin when it executes a memory reference instruction such as `MOV AX,[SI]`.
- $\overline{WR}$ is used by the 8086 for a write operation. The 8086 outputs a low on this pin to indicate that the processor is performing a write memory or write I/O operation, depending on the M/$\overline{IO}$ signal. Similarly, $\overline{RD}$ is low whenever the 8086 is reading data from memory or an I/O location.
- For interrupt acknowledge cycles (for the INTR pin), the 8086 outputs LOW on the $\overline{INTA}$ pin.
- HOLD (input) and HLDA (output) pins are used for DMA. A HIGH on the HOLD pin indicates that another master is requesting to take over the system bus. The processor receiving the HOLD request will output a HIGH on the HLDA as an acknowledgment. At the same time, the processor tristates the system bus. Upon receipt of LOW on the HOLD pin, the processor places LOW on the HLDA pin and takes over the system bus.
- CLK (input) provides the basic timing for the 8086 and bus controller.
- READY (input) pin is used for slow peripheral devices.

There are four versions of the 8086. They are 8086, 8086-1, 8086-2, and 8086-4. There is no difference between the four versions other than the maximum allowed clock speeds. The 8086 can be operated from a maximum clock frequency of 5 MHz. The maximum clock frequencies of the 8086-1, 8086-2 and 8086-4 are 10 MHz, 8 MHz and 4 MHz, respectively. Because the design of these processors incorporates dynamic cells, a minimum frequency of 2 MHz is required to retain the state of the machine. The 8086-4, 8086, and 8086-2 will be referred to as 8086 in the following discussion.



Pin Name	Description
X_1, X_2	Crystal connections
F/$\overline{C}$	Clock source select
CLK	MOS CLOCK for the 8086
$\overline{RES}$	Reset input to the 8284 from an RC circuit
RESET	Reset input to the processor
Vcc	+5 V
GND	0V
OSC	Oscillator output
TANK	Used with overtone crystal
EFI	External clock input
CSYNC	Clock synchronization input
RDY1, RDY2	Ready signals from two multibus systems
$\overline{AEN1}$, $\overline{AEN2}$	Address enables for ready signals
PCLK	TTL clock for peripherals
READY	Ready output

FIGURE 9.9 8284 pins and signals

The reset, clock, and the ready signals of the 8086 can be generated by the Intel 8284. Figure 9.9 shows the pins and signals of the 8284.

The 8284 is an 18-pin chip designed for providing three input signals for the 8086:

1. 8086 CLK input
2. 8086 Reset input
3. 8086 Ready input

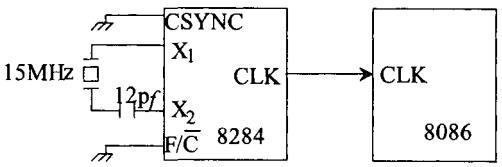
The 8284 pins and signals are described in the following.

Clock Generation Signals

Because the 8086 has no on-chip clock generator circuitry, the 8284 chip is required to provide the 8086 clock input. The 8284 F/$\overline{C}$ input pin is provided for clock source selection. When the F/$\overline{C}$ pin is connected to LOW, a crystal connected between 8284's X_1 and X_2 pins is used. On the other hand, when F/$\overline{C}$ is connected to HIGH, an external clock source is used; the external clock source is connected to the 8284 EFI (external frequency input) pin. The 8284 divides the clock inputs at the X_1X_2 pins or the EFI pin by 3. This means that if a 15-MHz crystal is connected at the X_1X_2 or EFI pins, the 8284 CLK output pin will be 5 MHz. The 8284 CLK pin will be connected to the 8086 CLK pin. This provides the clock input for the 8086. When selecting a crystal for use with the 8284, the crystal series resistance should be as low as possible. The oscillator delays in the 8284 appear as inductive elements to the crystal and cause the 8284 to run at a frequency below that of the pure series resonance: a capacitor C_L should be placed in series with the crystal and the 8284 X_2 pin. The capacitor cancels the inductive element. The impedance of the capacitor $X_c = 1/(2\pi fC_L)$ where f is the crystal frequency. Intel recommends that the crystal series resistance plus X_c should be kept less than 1 KΩ.

As the crystal frequency increases, C_L should be decreased. For example, a 12-MHz crystal may require C_L = 24 pf whereas a 22-MHz crystal may require C_L = 8pf. C_L values of 12 to 15 pf may be used with a 15-MHz crystal. Two crystal manufacturers recommended by Intel are Crystle Corp., Model CY 15A (15 MHz), and CTS Knight, Inc., Model CY 24A (24 MHz). Note that the 8284 CLK output pin is the MOS clock for the 8086.

There are two more clock outputs on the 8284, the PCLK (peripheral clock) pin and the OSC (oscillator) clock pin. These signals are provided to drive peripheral ICs. The 8284 divides the frequency of the crystal at the X_1X_2 pins or the external clock at the EFI pin by 6 to provide the PCLK. Therefore, the frequency of the PCLK is half the frequency of the 8284 CLK output pin. This means that for a 15-MHz crystal, the PCLK and CLK outputs are 2.5 MHz and 5 MHz respectively. Furthermore, PCLK is provided at the TTL-compatible level rather than at the MOS level. The OSC clock, on the other hand, is derived from the crystal oscillator inside the 8284 and has the same clock frequency as the crystal. Therefore, the OSC output is three times that of the CLK output. The OSC is also TTL compatible. Finally, the CSYNC (clock synchronization) input pin when connected to HIGH provides external synchronization in systems that employ multiple clocks. A typical 8284 interface to the 8086 for providing a 5-MHz clock to the 8086 is shown in the following figure:



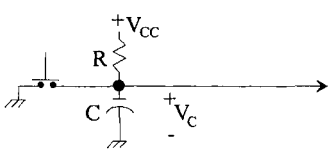
Reset Signals

When designing the microprocessor's reset circuit, two types of reset must be considered: power-up reset and manual reset. These reset circuits must be designed using the parameters specified by the manufacturer.

Therefore, a microprocessor must be reset when its Vcc pin is connected to power. This is called "power-up reset." After some time during normal operation the microprocessor can be reset upon activation of a manual switch such as a pushbutton. A reset circuit, therefore, needs to be designed following the timing parameters associated with the microprocessor's reset input pin specified by the manufacturer. The reset circuit, once designed, is connected to the microprocessor's reset pin.

As mentioned before, the 8086 reset input provides a hardware mechanism for initializing the 8086 microprocessor. This is typically done at power-up to provide an orderly start-up of the system. The 8284 $\overline{\text{RES}}$ (reset input) pin when driven active LOW generates a HIGH on the 8284 reset output pin. The 8284 reset pin is connected to the 8086 reset (input) pin. As mentioned before, Intel designed the 8086 in such a way that the 8086 requires its reset pin to be HIGH for at least four clock cycles in order to obtain the physical address (FFFF0H) of the first instruction to be executed, except after power-on, which requires a 50-μsec reset pulse.

According to Intel, in order to guarantee a reset from power-up, the 8086 reset input must remain below 1.05 V for 50 μsec after Vcc has reached the minimum supply voltage of 4.5 V. The 8284 $\overline{\text{RES}}$ input can be driven by an *RC* circuit as shown in the following figure:



To 8284 $\overline{\text{RES}}$ input pin

The voltage across the capacitor initially is zero upon connecting +Vcc to power. If the switch is not depressed, the capacitor charges to +Vcc through the resistor after a definite time determined by the time constant RC.

The charging voltage across the capacitor can be determined from the following equation. Capacitor voltage, $V_c(t) = V_{cc} \times [1 - \exp(-t/RC)]$, where t = 50 μsec and $V_c(t)$ = 1.05 V, and V_{cc} = 4.5 V. Substituting these values in the equation, RC = 188 μsec. For example, if C is chosen to be 0.1 μF, then R is 1.88 KΩ.

When the switch is depressed, the 8284 $\overline{\text{RES}}$ input pin is short-circuited to ground. This takes the 8284 $\overline{\text{RES}}$ pin to LOW and thus discharges the capacitor. As the switch is released, the direct short to ground is broken. However, the 8284 $\overline{\text{RES}}$ pin remains effectively short-circuited to ground through the discharged capacitor. The capacitor now starts to recharge with time toward the $+V_{cc}$ voltage level.

The 8284 generates a reset signal from an internal Schmitt trigger input. A Schmitt trigger is a special analog circuit that shifts the switching threshold based on whether the input changes from LOW to HIGH or from HIGH to LOW. To illustrate this, consider a

TTL Schmitt trigger inverter. Suppose that the input of this inverter is at 0 V (logic 0). The output will be approximately 3.4 V (logic 1). Now, because of the Schmitt trigger circuit, if the input voltage is increased, the output will not go to low until the value is about 1.7 V. Also, after reaching a low output, the inverter will not produce a HIGH output until the input is decreased to about 0.9 V. Thus, the switching threshold for positive-going input changes is about 1.7 V and for negative-going input changes is about 0.9 V.

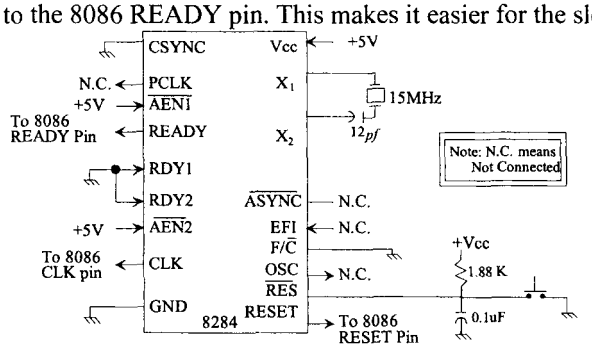
The difference between the two thresholds is called "hysteresis." The Schmitt trigger inverter provides 1.7 V - 0.9 V = 0.8 V of hysteresis. Schmitt trigger inputs provide high noise immunity and will normally not respond to the noise encountered in microprocessor systems if its hysteresis is greater than the noise amplitude.

As the voltage across the capacitor increases with time, it remains at logic 0 level as long as the logic 1 threshold of the Schmitt trigger. Thus, the 8284 $\overline{\text{RES}}$ input is maintained at logic 0 for at least four clock cycles so that the 8284 RESET output will apply a HIGH at the 8086 reset input for at least four clock cycles. Note that whenever the 8282 $\overline{\text{RES}}$ input is at logic 0, the reset output pin of the 8284 is switched to logic 1 according to the timing parameters.

Ready Signals

The 8284 Ready (output) pin is connected to the 8086 Ready (input) pin to insert wait states for slow peripheral devices connected to the 8086. There are two main ways to disable this function when not used. One way is to connect the 8086 Ready pin to HIGH, and keep the 8284 Ready output pin floating. The other way is to connect the 8284 RDY1 and RDY2 pins to LOW, and the $\overline{\text{AEN1}}$ and $\overline{\text{AEN2}}$ to HIGH, which will permanently disable this function. The 8284 Ready (output) pin can then be connected to the 8086 Ready input pin.

The RDY1, $\overline{\text{AEN1}}$ and RDY2, $\overline{\text{AEN2}}$ input signals provide logic for operation with multiprocessor systems and the 8284 ready output. In multiprocessor systems, these signals are used to control access over the system bus by several 8086's. The 8284 TANK pin is replaced by the $\overline{\text{ASYNC}}$ input pin on the newer version of 8284. The $\overline{\text{ASYNC}}$ pin can be driven to LOW by a slower device to generate the 8284 READY output pin which can be connected to the 8086 READY pin. This makes it easier for the slower devices to



interface to the 8086. Typical 8284 clock (using a 15-MHz crystal), reset, and ready signal (unused) connections to single 8086-appropriate pins are shown in the above figure.

In the maximum mode, some of the 8086 pins in the minimum mode are redefined. For example, pins HOLD, HLDA, $\overline{\text{WR}}$, M/$\overline{\text{IO}}$, DT/$\overline{\text{R}}$, $\overline{\text{DEN}}$, ALE, and $\overline{\text{INTA}}$ in the minimum mode are redefined as $\overline{\text{RQ}}$/$\overline{\text{GT0}}$, $\overline{\text{RQ}}$/$\overline{\text{GT1}}$, $\overline{\text{LOCK}}$, $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$, QS_0, and QS_1, respectively. In maximum mode, the 8288 bus controller decodes the status information from $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ to generate the bus timing and control signals that are required for a bus

cycle. $\overline{S_0}$, $\overline{S_1}$, and $\overline{S_2}$ are 8086 outputs and are decoded as follows:

S_2	S_1	S_0	Function
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Inactive

The $\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$ request/grant pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional, with $\overline{RQ}/\overline{GT0}$ having higher priority than $\overline{RQ}/\overline{GT1}$.These pins have internal pull-up resistors so that they may be left unconnected. The request/grant function of the 8086 works as follows:

- A pulse (one clock wide) from another local bus master ($\overline{RQ}/\overline{GT0}$ or $\overline{RQ}/\overline{GT1}$ pin) indicates a local bus request to the 8086.
- At the end of the current 8086 bus cycle, a pulse (one clock wide) from the 8086 to the requesting master indicates that the 8086 has relinquished the system bus and tristates the outputs. Then the new bus master subsequently relinquishes control of the system bus by sending a LOW on $\overline{RQ}/\overline{GT0}$ or $\overline{RQ}/\overline{GT1}$ pin. The 8086 then regains bus control.
- The 8086 outputs LOW on the $\overline{LOCK}$ pin to prevent other bus masters from gaining control of the system bus.

Note that since the 8086 RESET vector is located at the physical address FFFF0H, there may not be enough locations available to write programs. The following 8086 instruction sequence can be used with 8086 assembler (HP 64XXX) to jump to a different code segment upon hardware reset to write programs:

```
ORG 0FFFFH:0000H ; Reset Vector    ORG 1000H:0200H
JMP FAR PTR START        START      —} User
                                    —} Programs
```

The above instruction sequence will allow the 8086 to jump to the offset START (0200H) in code segment 1000H upon hardware reset where the user can write programs.

9.9.2 Basic 8086 System Concepts

This section describes basic concepts associated with the 8086 bus cycles, address and data bus, in minimum mode.

8086 Bus Cycle

To communicate with external devices via the system for transferring data or fetching instructions, the 8086 executes a bus cycle. The 8086 basic bus cycle timing diagram is shown in Figure 9.10. The minimum bus cycle contains four microprocessor clock periods or four *T states*. Note that each cycle is called a T state. The bus cycle timing diagram depicted in Figure 9.10 can be described as follows:

1. During the first T state (T_1), the 8086 outputs the 20-bit address computed from a segment register and an offset on the multiplexed address/data/status bus.
2. For the second T state (T_2), the 8086 removes the address from the bus and either

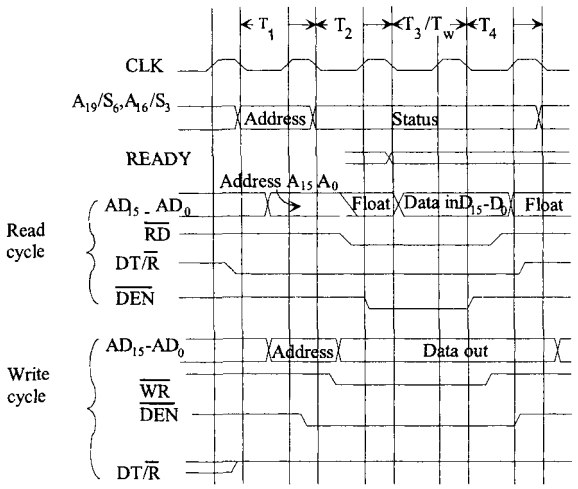


FIGURE 9.10 Basic 8086 bus cycle

tristates or activates the AD_{15}–AD_0 lines in preparation for reading data via the AD_{15}–AD_0 lines during the T_3 cycle. In the case of a write bus cycle, the 8086 outputs data on the AD_{15}–AD_0 lines during the T_3 cycle. Also, during T_2, the upper four multiplexed bus lines switch from address (A_{19}–A_{16}) to bus cycle status (S_6, S_5, S_4, S_3). The 8086 outputs LOW on $\overline{RD}$ (for the read cycle) or $\overline{WR}$ (for the write cycle) during portion of T_2, all of T_3, and portion of T_4.

3. During T_3, the 8086 continues to output status information on the four A_{19}–A_{16}/ S_6–S_3 lines and will continue to output write data or input read data to or from the AD_{15}–AD_0 lines.
4. If the selected memory or I/O device is not fast enough to transfer data to the 8086, the memory or I/O device activates the 8086's READY input line LOW by the start of T_3. This will force the 8086 to insert additional clock cycles (wait states T_w) after T_2. Bus activity during T_W is the same as that during T_3. When the selected device has had sufficient time to complete the transfer, it must activate the 8086 ready pin HIGH. As soon as the T_w clock period ends, the 8086 executes the last bus cycle (T_4). The 8086 will latch data on the AD_{15}–AD_0 lines during the last wait state or during T_3 if no wait states are requested.
5. During T_4, the 8086 disables the command lines and the selected memory and I/O devices from the bus. Thus, the bus cycle is terminated in T4. The bus cycle appears to devices in the system as an asynchronous event consisting of an address to select the device, a register or memory location within the device, a read strobe, or a write strobe along with data.
6. The $\overline{DEN}$ and DT/$\overline{R}$ pins are used by the 8286/8287 transceiver in a minimum system. During the read cycle, the 8086 outputs $\overline{DEN}$ LOW during part of the T_2 and all of the T_3 cycles. This signal can be used to enable the 8286/8287 transceiver. The 8086 outputs a LOW on the DT/$\overline{R}$ pin from the start of the T_1 through part of the T_4 cycles. The 8086 uses this signal to receive (read) data from the receiver during T_3–T_4. During a write cycle, the 8086 outputs $\overline{DEN}$ LOW during part of the T_1, all of the T_2, and T_3, and part of the T_4 cycles. The signal can be used to enable the transceiver. The 8086 outputs a HIGH on DT/$\overline{R}$ throughout the 4 bus cycles to transmit (write) data to the transceiver during T_3–T_4.

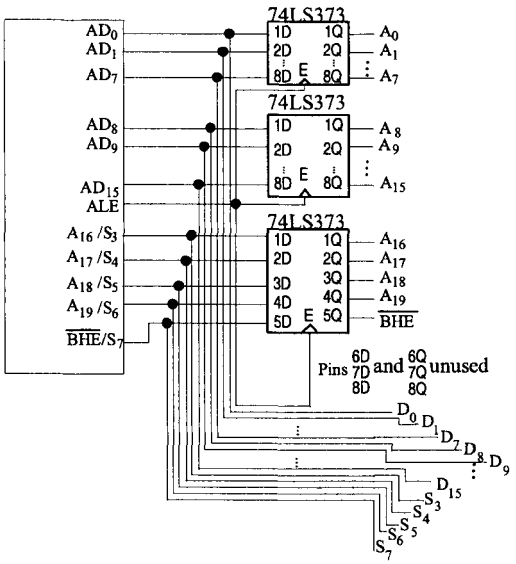


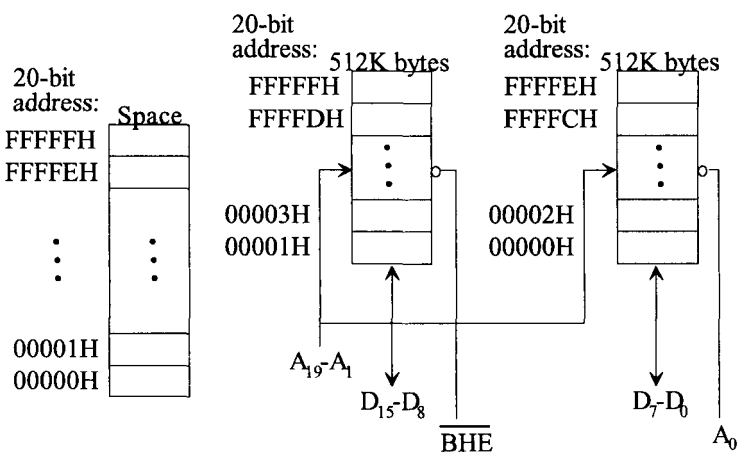
FIGURE 9.11 Demultiplexing address, data, and status lines of the 8086

Address and Data Bus Concepts

The majority of memory and I/O chips capable of interfacing to the 8086 require a stable address for the duration of the bus cycle. Therefore, the address on the 8086 multiplexed address/data bus during T_1 should be latched. The latched address is then used to select the desired I/O or memory location. To demultiplex the bus, the 8086 ALE pin can be used along with three 74LS373 latches.

The 74LS373 Output Control ($\overline{OC}$) pin can be connected to ground with the 74LS373 pin represented by G or C or LE (shown as E in Figure 9.11) in data book tied to 8086 ALE. This will latch the 8086 address and $\overline{BHE}$ pins at the falling edge of ALE. Figure 9.11 shows how this can be accomplished.

The programmer views the 8086 memory address space as a sequence of one



(a) One megabyte address space (b) Physical implementation of address space

FIGURE 9.12 8086 Memory

mega bytes in which any byte may contain an 8-bit data element and any two consecutive bytes may contain a 16-bit data element. There is no constraint on byte or word addresses (boundaries). The address space is physically implemented on a 16-bit data bus by dividing the address space into two banks of up to 512K bytes as shown in Figure 9.12. These banks can be selected by $\overline{BHE}$ and A_0 as follows:

$\overline{BHE}$	A_0	Byte transferred
0	0	Both bytes via demultiplexed D_0–D_{15} pins for even address.
0	1	Upper byte to/from odd address via demultiplexed D_8–D_{15} pins.
1	0	Lower byte to/from even address via demultiplexed D_0–D_7 pins.
1	1	None

One bank is connected to D_7–D_0 and contains all even-addressed bytes ($A_0 = 0$). The other bank is connected to D_{15}–D_8 and contains odd-addressed bytes ($A_0 = 1$). A particular byte in each bank is addressed by A_{19}–A_1. The even-addressed bank is enabled by a LOW on A_0, and data bytes are transferred over the D_7–D_0 lines. The 8086 outputs a HIGH on $\overline{BHE}$ (bus high enable) and thus disables the odd-addressed bank. The 8086 outputs a LOW on $\overline{BHE}$ to select the odd-addressed bank and a HIGH on A_0 to disable the even-addressed bank. This directs the data transfer to the appropriate half of the data bus.

Activation of A_0 and $\overline{BHE}$ is performed by the 8086 depending on odd or even addresses and is transparent to the programmer. As an example, consider execution of the instruction `MOV [BX],DH`. Suppose the 20-bit address computed by BX and DS is even. The 8086 outputs a LOW on A_0 and a HIGH on $\overline{BHE}$ .This will select the even-addressed bank. The content of DH is placed on the D_7–D_0 lines by a memory chip. The 8086 writes this data via D_7–D_0 and automatically places it in the selected memory location. Next, consider writing a 16-bit word by the 8086 with the low byte at an even address as shown in Figure 9.13. For example, suppose that the 8086 executes the instruction `MOV [BX],CX`. Assume [BX] = 0004H and [DS] = 2000H. The 20-bit physical address for the word is 20004H. The 8086 outputs a LOW on both A_0 and $\overline{BHE}$, enabling both banks simultaneously. The 8086 outputs [CL] to the D_7–D_0 lines and [CH] to the D_{15}–D_8 lines, with $\overline{WR}$ = LOW and M/$\overline{IO}$ = HIGH. The enabled memory banks obtain the 16-bit data and write [CL] to location 20004H and [CH] to location 20005H.

Next, consider writing an odd-addressed 16-bit word by the 8086 using `MOV [BX],CX`. For example, suppose the 20-bit physical address computed by the 8086 is 20005H. The 8086 accomplishes this transfer in two bus cycles. In the first bus cycle, the 8086 outputs a HIGH on A_0 and a LOW on $\overline{BHE}$, and thus enables the odd-addressed bank and disables the even-addressed bank. The 8086 also outputs a LOW on the $\overline{WR}$ and a HIGH on the M/$\overline{IO}$ pins. In this bus cycle, the 8086 writes data to odd memory bank via D_{15}–D_8 lines; the 8086 writes the contents of CL to address 20005H. In the second

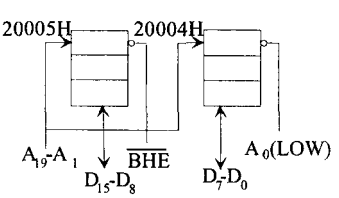


FIGURE 9.13 Even-addressed word transfer

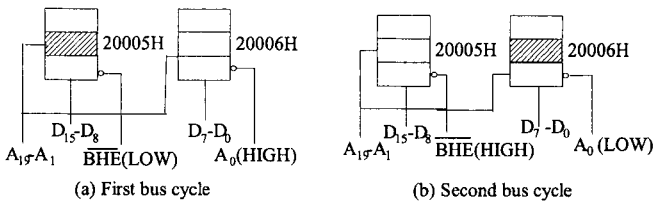


FIGURE 9.14 Odd-addressed word transfer

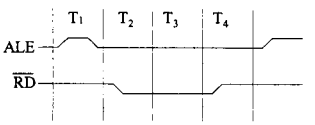


FIGURE 9.15 Relationship of ALE and read

bus cycle, the 8086 outputs a LOW on A_0 and a HIGH on $\overline{BHE}$ and thus enables the even-addressed bank and disables the odd-addressed bank. The 8086 also outputs a LOW on the $\overline{WR}$ and a HIGH on the M/$\overline{IO}$ pins. The 8086 writes data to even memory bank via D_7–D_0 lines; the 8086 writes the contents of CH to address 20006H. This odd-addressed word write is shown in Figure 9.14.

If memory or I/O devices are directly connected to the multiplexed bus, the designer must guarantee that the devices do not corrupt the address on the bus during T_1. To avoid this, the memory or I/O devices should have an output enable controlled by the 8086 read signal. The 8086 timing guarantees that the read is not valid until after the address is latched by ALE as shown in Figure 9.15.

All Intel peripherals, EPROMs, and RAMs for microprocessors provide output enable for read inputs to allow connection to the multiplexed bus. Several techniques are available for interfacing the devices without output enables to the 8086 multiplexed bus. However, these techniques will not be discussed here.

9.9.3 Interfacing with Memories

In Figure 9.16, the 16-bit word memory in the 8086 is partitioned into odd and even 8-bit banks on the upper and lower halves of the data bus selected by $\overline{BHE}$ and A_0. This is typically used for RAMs. Note that RAMs are needed when subroutines and interrupts requiring stack are desired in an application.

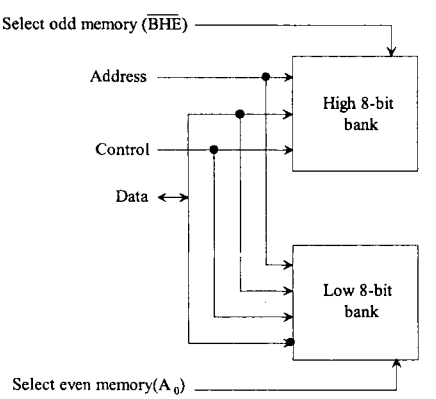
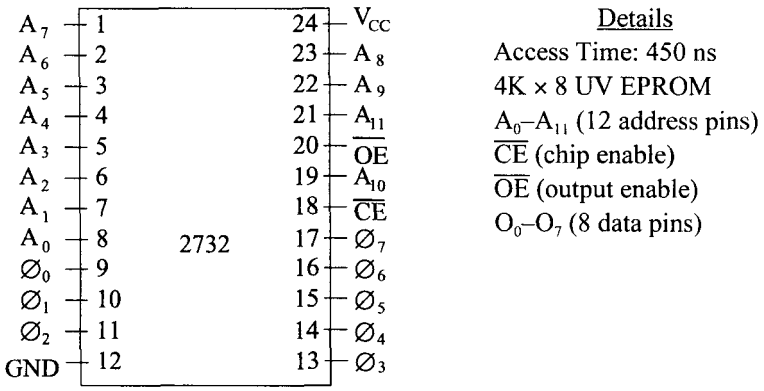
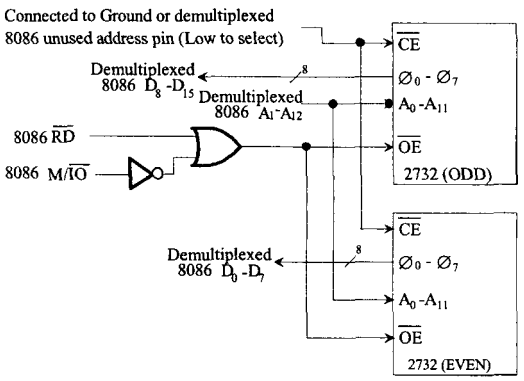


FIGURE 9.16 8086 memory array



(a) 2732 Pins and Signals



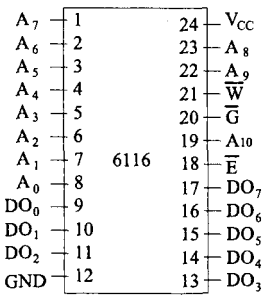
(b) 8086–2732 Connections

FIGURE 9.17 8086–2372 interface along with 2732 pins and signals

ROMs and EPROMs

ROMs and EPROMs are the simplest memory chips to interface to the 8086. Because ROMs and EPROMs are read-only devices and the 8086 always reads 16-bit data but discards unwanted bytes (if necessary), A_0 and $\overline{BHE}$ are not required to be part of the chip enable/select decoding (chip enable is similar to chip select decoding except that chip enable also provides whether the chip is in active or standby power mode). The 8086 address lines must be connected to the ROM/EPROM chips starting with A_1 and higher to all the address lines of the ROM/EPROM chips. The 8086 unused address lines can be used as chip enable/select decoding. To interface the ROMs/EPROMs directly to the 8086 multiplexed bus, they must have output enable signals. Figure 9.17 shows the 8086 interfaced to two 2732 chips along with the pin diagram of 2732.

The 8086's interface to 2732 EPROMs in Figure 9.17(b) does not use 8086 $\overline{BHE}$ and A_0 to distinguish between even and odd 2732s. The 8086 $\overline{RD}$ and inverted $\overline{M/IO}$ pins are ORed and connected to the 2732 $\overline{OE}$ pins. The 8086 $\overline{CE}$ can be connected to either ground or an unused 8086 address pin. Note that both 2732's are enabled for all data reads; the odd 2732 places data on the demultiplexed 8086 D_8–D_{15} pins while the even 2732 places data on the demultiplexed 8086 D_0–D_7 pins. The 8086 reads the desired data and discards unwanted data if necessary depending on byte, odd word address or even word address transfers.

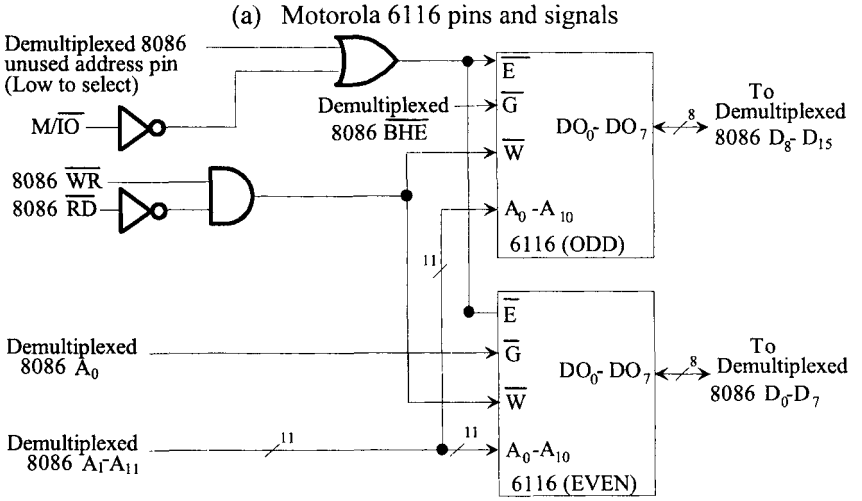


Details
Access Time: 120 ns
2K × 8 SRAM designed using HCMOS
A_0–A_{10} (11 addresses)
DO_0–DO_7 (8 data pins)
$\overline{W}$ (write enable)
$\overline{G}$ (output enable)
$\overline{E}$ (chip enable)
Vcc +5 V
Vss Ground

Mode Selection	$\overline{E}$	$\overline{G}$	$\overline{W}$
Read	L	L	H
Write	L	L	L
Stand by or Disable (Tristate)	H	X	X

X = don't care

(a) Motorola 6116 pins and signals



(b) 8086-6116 connections

FIGURE 9.18 8086–6116 interface along with 6116 pin diagram

Static RAMs (SRAMs)

Because static RAMs are read/write memories and data will be written to RAM(s) once selected by the 8086, both A_0 and $\overline{BHE}$ must be included in the chip select logic. For each static RAM, the data lines must be connected to either the upper half (AD_{15}–AD_8) or the lower half (AD_7–AD_0) of the 8086 data lines. Figure 9.18 shows the 8086 interface to two 6116 static RAMs along with the pin diagram of the 6116. Note that the 6116 signals, $\overline{W}$ (Write Enable), $\overline{G}$ (Output enable), and $\overline{E}$ (Chip enable) are decoded as follows: when $\overline{G}$ = 0 and $\overline{E}$ = 0, then $\overline{W}$ = 1 for read and $\overline{W}$ = 0 for write.

In Figure 9.18, the 8086 demultiplexed $\overline{BHE}$ signal is used to select odd 6116 SRAM chips; the data lines of this odd 6116 are connected to the demultiplexed 8086 D_8–D_{15} pins. The 8086 demultiplexed A_0 signal, on the other hand, is used to select even 6116 SRAM chip; the data lines of this even 6116 are connected to the demultiplexed 8086 D_0–D_7 pins. Note that the 6116 has two chip enables $\overline{E}$ and $\overline{G}$ along with a single read/write pin ($\overline{W}$) .When the 6116 is enabled, $\overline{W}$ = 1 for read and $\overline{G}$ = 0 for write.

Dynamic RAMs (DRAMs)

Dynamic RAMs store information as charges in capacitors. Because capacitors can hold charges for a few milliseconds, refresh circuitry is necessary in dynamic RAMs for retaining these charges. Therefore, dynamic RAMs are complex devices to use to design a system. To relieve the designer of most of these complicated interfacing tasks, Intel provides dynamic RAM controllers to interface with the 8086 to build a dynamic memory system. Dynamic RAMs are used for microcomputers requiring large memories. DRAMs are typically used when memory requirements are 16k words or larger. DRAM is addressed via row and column addressing. For example, one megabit DRAM requiring 20 address bits is addressed using 10 address lines and two control lines, $\overline{RAS}$ (Row Address Strobe) and $\overline{CAS}$ (Column Address Strobe). To provide a 20-bit address into the DRAM, a LOW is applied to $\overline{RAS}$ and 10 bits of the address are latched. The other 10 bits of the address are applied next and $\overline{CAS}$ is then held LOW.

The addressing capability of the DRAM can be increased by a factor of 4 by adding one more bit to the address line. This is because one additional address bit results into one additional row bit and one additional column bit. This is why DRAMs can be expanded to larger memory very rapidly with inclusion of additional address bits. External logic is required to generate the $\overline{RAS}$ and $\overline{CAS}$ signals, and to output the current address bits to the DRAM.

DRAM controller chips take care of refreshing and timing requirements needed by the DRAMs. DRAMs typically require 4 millisecond refresh time. The DRAM controller performs its task independent of the microprocessor. The DRAM controller sends a wait signal to the microprocessor if the microprocessor tries to access memory during a refresh cycle.

Because of large memory, the address lines should be buffered using 74LS244 or 74HC244 (Unidirectional buffer), and data lines should be buffered using 74LS245 or 74HC245 (Bidirectional buffer) to increase the drive capability. Also, typical multiplexers such as 74LS157 or 74HC157 can be used to multiplex the microprocessors address lines into separate row and column addresses.

9.9.4 8086 I/O Ports

Devices with 8-bit I/O ports can be connected to either the upper or the lower half of the data bus. If the I/O port chip is connected to the lower half of the 8086 data lines (AD_0–AD_7), the port addresses will be even ($A_0 = 0$). On the other hand, the port addresses will be odd ($A_0 = 1$) if the I/O port chip is connected to the upper half of the 8086 data lines (AD_8–AD_{15}). A_0 will always be 1 or 0 for the partitioned I/O chip. Therefore, A_0 cannot be used as an address input to select registers within a particular I/O chip. If two chips are connected to the lower and upper halves of the 8086 address bus that differ only in A_0 (consecutive odd and even addresses), A_0 and $\overline{BHE}$ must be used as conditions of chip select decoding to avoid a write to one I/O chip from erroneously performing a write to the other.

The 8086 uses either standard I/O or memory-mapped I/O. The standard I/O uses the instructions IN and OUT, and is able to provide up to 64K bytes of I/O locations. The standard I/O can transfer either 8-bit data or 16-bit data to or from a peripheral device. The 64-Kbyte I/0 locations can then be configured as 64K 8-bit ports or 32K 16-bit ports. All I/O transfers between the 8086 and peripheral devices take place via AL for 8-bit ports (AH is not involved) and AX for 16-bit ports.

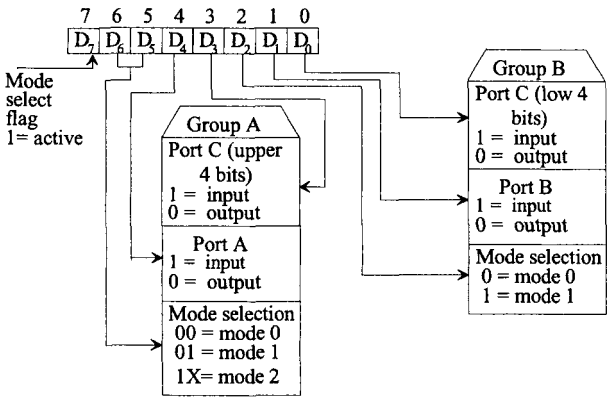


FIGURE 9.19 8255 control register

The I/O port addressing can be done either directly or indirectly as follows:

- ***Direct***

  `IN AX,PORTA` or `IN AL,PORTA` inputs 16-bit contents of port A into AX or 8-bit contents of port A into AL, respectively.

  `OUT PORTA,AX` or `OUT PORTA,AL` outputs 16-bit contents of AX into port A or 8-hit contents of AL into port A, respectively.

- ***Indirect***

  `IN AX,DX` or `IN AL,DX` inputs 16-bit data into a port addressed by DX into AX or 8-bit data into a port addressed by DX into AL, respectively.

  `OUT DX,AX` or `OUT DX,AL` outputs 16-bit contents of AX into a port addressed by DX or 8-bit contents of AL into a port addressed by DX, respectively.

Memory-mapped I/O is basically accomplished by using the memory instructions such as `MOV AX` or `AL,[BX]` and `MOV [BX],AX` or `AL` for inputting or outputting, 8- or 16-bit data to/from AL or AX addressed by the 20-bit address computed from DS and BX. Note that any 8- or 16-bit general purpose register and memory modes can be used in memory-mapped I/O.

The 8086 programmed I/O capability will be explained in the following paragraphs using the 8255 I/O chip. The 8255 chip is a general-purpose programmable I/O chip. The 8255 has three 8-bit I/O ports: ports A, B, and C. Ports A and B are latched 8-bit ports for both input and output. Port C is also an 8-bit port with latched output, but the inputs are not latched. Port C can be used in two ways: It can be used either as a simple I/0 port or as a control port for data transfer using handshaking via ports A and B.

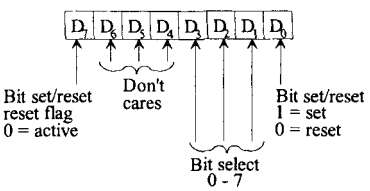
The 8086 configures the three ports by outputting appropriate data to the 8-bit control register. The ports can be decoded by two 8255 input pins A_0 and A_1, as follows:

A_1	A_0	Port Name
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

The definitions of the control register are shown in Figure 9.19.

Bit 7 (D_7) of the control register must be 1 to send the definitions for bits 0–6 (D_0–D_6) as shown in the diagram. In this format, bits D_0–D_6, are divided into two groups: groups A and B. Group A configures all 8 bits of port A and the upper 4 bits of port C; group B defines all 8 bits of port B and the lower 4 bits of port C. All bits in a port can be configured as a parallel input port by writing a 1 at the appropriate bit in the control register by the 8086 OUT instruction, and a 0 in a particular bit position will configure the appropriate port as a parallel output port. Group A has three modes of operation: modes 0, 1, and 2. Group B has two modes: modes 0 and 1. Mode 0 for both groups provides simple I/O operation for each of the three ports. No handshaking is required. Mode 1 for both groups is the strobed I/O mode used for transferring I/O data to or from a specified port in conjunction with strobes or handshaking signals. Ports A and B use the pins on port C to generate or accept these handshaking signals. Mode 2 of group A is the strobed bidirectional bus I/O and may be used for communicating with a peripheral device on a single 8-bit data bus for both transmitting and receiving data (bidirectional bus I/O). Handshaking signals are required. Interrupt generation and enable/disable functions are also available.

When $D_7 = 0$, the bit set/reset control word format is used for the control register as follows:



This format is used to set or reset the output on a pin of port C or when enabling of the interrupt output signals for handshake data transfer is desired. For example, the 8 bits (0XXX1100) will clear bit 6 of port C to zero. Note that the control word format can be output to the 8255 control register by using the 8086 OUT instruction. Now, let us define the control word format for mode 0 more precisely by means of a numerical example. Consider that the control word format is 10000010_2. With this data in the control register, all 8 bits of Port A are configured as outputs and the 8 bits of port C are also configured as outputs. All 8 bits of port B, however, are defined as inputs. On the other hand, outputting 10011011_2 into the control register will configure all three 8-bit ports (ports A, B, and C) as inputs.

9.9.5 Important Points To Be Considered for 8086 Interface to Memory and I/O

From the preceding discussions, the following points can be summarized:

1. For ROMs/EPROMs/E²PROMs, $\overline{BHE}$ and A_0 are not required as part of chip enable/select decoding.
2. For RAMs and I/O port chips, both $\overline{BHE}$ and A_0 must be used in chip select logic.
3. For ROMs/EPROMs/E²PROMs and RAMs, both even and odd chips are required. However, for I/O chips, an odd-addressed I/O chip, an even-addressed I/O chip, or both can be used, depending on the number of ports required in an application. The 8086 $\overline{BHE}$ and/or A_0 must be used in I/O chip select logic depending on the number and type (odd/even) of I/O chips used.
4. For interfacing ROMs/EPROMs/ E²PROMs to the 8086, the same chip select logic must be used for both the even and its corresponding odd memory chip. The same thing applies to RAM and I/O chips except that both $\overline{BHE}$ and A_0 must be

used for RAMs and I/O; however, this is applicable to I/O if both odd and even I/O chips are present in the system.

5. ROMs/EPROMs/E²PROMs must be connected in such a way that the 8086 reset vector address FFFF0H is contained in the memory map.

Example 9.19

An 8086-8255-2732-6116–based microcomputer is required to drive an LED connected to bit 2 of port B based on two switch inputs connected to bits 6 and 7 of port A. If both switches are either HIGH or LOW, turn the LED ON; otherwise, turn it OFF. Assume a HIGH will turn the LED ON and a LOW will turn it OFF. Write an 8086 assembly language program to accomplish this.

Solution

```
PORTA  EQU    0F8H
PORTB         EQU    0FAH
CNTRL  EQU    0FEH
PROG          SEGMENT
              ASSUME CS: PROG
              MOV    AL, 90H        ;  Configure port A
              OUT    CNTRL, AL      ;  as input and port B
                                    ;  as output
BEGIN:        IN     AL, PORTA      ;  Input port A

              AND    AL, 0C0H       ;  Retain bits 6 and 7
              JPE    LEDON          ;  If both switches are either
                                    ;  HIGH or LOW, turn the LED ON
              MOV    AL, 00H        ;  Otherwise turn the
              OUT    PORTB, AL      ;  LED OFF
              JMP    BEGIN          ;  Repeat
LEDON:        MOV    AL, 04H        ;  Turn LED
              OUT    PORTB, AL      ;  ON
              JMP    BEGIN
PROG          ENDS
              END
```

Example 9.20

Write an 8086 assembly language program to drive an LED connected to bit 7 of port A based on a switch input at bit 0 of port A. If the switch is HIGH, turn the LED ON; otherwise, turn the LED OFF. Assume an 8086/2732/6116/8255 microcomputer. Also, write a C++ program to accomplish the same task. Compare the 68000 assembly program with the compiled assembly code. Comment on the result.

Solution

The 8086 assembly language program and the C++ program along with the compiled assembly code are shown below. The 8086 assembly program contains 11 instructions whereas the 8086 C++ code generates 16 instructions. This example illustrates that although C++ programming can handle I/O, it generates more codes than assembly language programming. Although programs in C++ are easier to write compared to assembly, the machine code generated by the equivalent assembly language is shorter. Also note that C++ programs are not 100 % portable while the same I/O programs are written using C++ for microprocessors by two different manufactures. This is because of the different hardware configurations (I/O and memory maps) for different manufacturers.

Note that the assembly language program can also be written by rotating bit 0 (switch input) of port A to bit 7 (LED output) of port A only once by using ROR Al,1 rather than RCL AL,CL with [CL]=7. The equivalent C++ program will still generate more assembled codes than the assembly language program.

8086/8255 Microcomputer Assembly Code for Switch and LED (MASM) of Example 9.20

```
= 00F8          PORTA     EQU       0F8H
= 00FE          CTLREG    EQU       0FEH
0000            LAB       SEGMENT
                          ASSUME    CS:LAB
0000  B1 07               MOV       CL,7
0002  B0 90    REPEAT:     MOV       AL,90H
0004  E6 FE               OUT       CTLREG,AL      ; set PORTA as input
0006  E4 F8               IN            AL,PORTA   ; read switch
0008  8A D8               MOV       BL,AL          ; save switch status
000A  B0 80               MOV       AL,80H
000C  E6 FE               OUT       CTLREG,AL      ; set PORTA as output
000E  8A C3               MOV       AL,BL          ; get switch status
0010  D2 D0               RCL       AL,CL          ; rotate switch status
0012  E6 F8               OUT       PORTA,AL       ; output to LED
0014  EB EC               JMP       REPEAT         ; repeat
0016            LAB       ENDS
                          END
```

```
#include <dos.h>
#define PORTA 0x0F8
#define CNTLREG 0x0FE
int main (){
        int x;
        while(1){
                outportb(CNTLREG, 0x90);        // set PORTA as input
                x = inportb(PORTA);             // read switch
                outportb(CNTLREG, 0x80);        // set PORTA as output
                outportb(PORTA, x << 7);        // output to LED
        }
}
```

• Assembly code generated from C++ code above using Microsoft DEBUG unassembler:
• 8086/8255 Microcomputer C++ program for Switch and LED (C++ Compiler) of Example 9.20

```
-r
AX=0000  BX=0000  CX=022E  DX=0000  SP=FFEE  BP=0000  SI=0000
DI=0000
DS=159B  ES=159B  SS=159B  CS=159B  IP=0100   NV UP EI PL NZ NZ PO NC
159B:0100 800C00          OR      BYTE PTR [SI],00
    DS:0000=CD
-u 2aa 2c8
159B:02AA  BAFE00         MOV     DX,00FE
159B:02AD  B090           MOV     AL,90
159B:02AF  EE             OUT     DX,AL
159B:02B0  BAF800         MOV     DX,00F8
159B:02B3  EC             IN      AL,DX
159B:02B4  B400           MOV     AH,00
```

```
159B:02B6   8BD8            MOV     BX,AX
159B:02B8   BAFE00          MOV     DX,00FE
159B:02BB   B080            MOV     AL,80
159B:02BD   EE              OUT     DX,AL
159B:02BE   B107            MOV     CL,07
159B:02C0   8AC3            MOV     AL,BL
159B:02C2   D2E0            SHL     AL,CL
159B:02C4   BAF800          MOV     DX,00F8
159B:02C7   EE              OUT     DX,AL
159B:02C8   EBE0            JMP     02AA
```

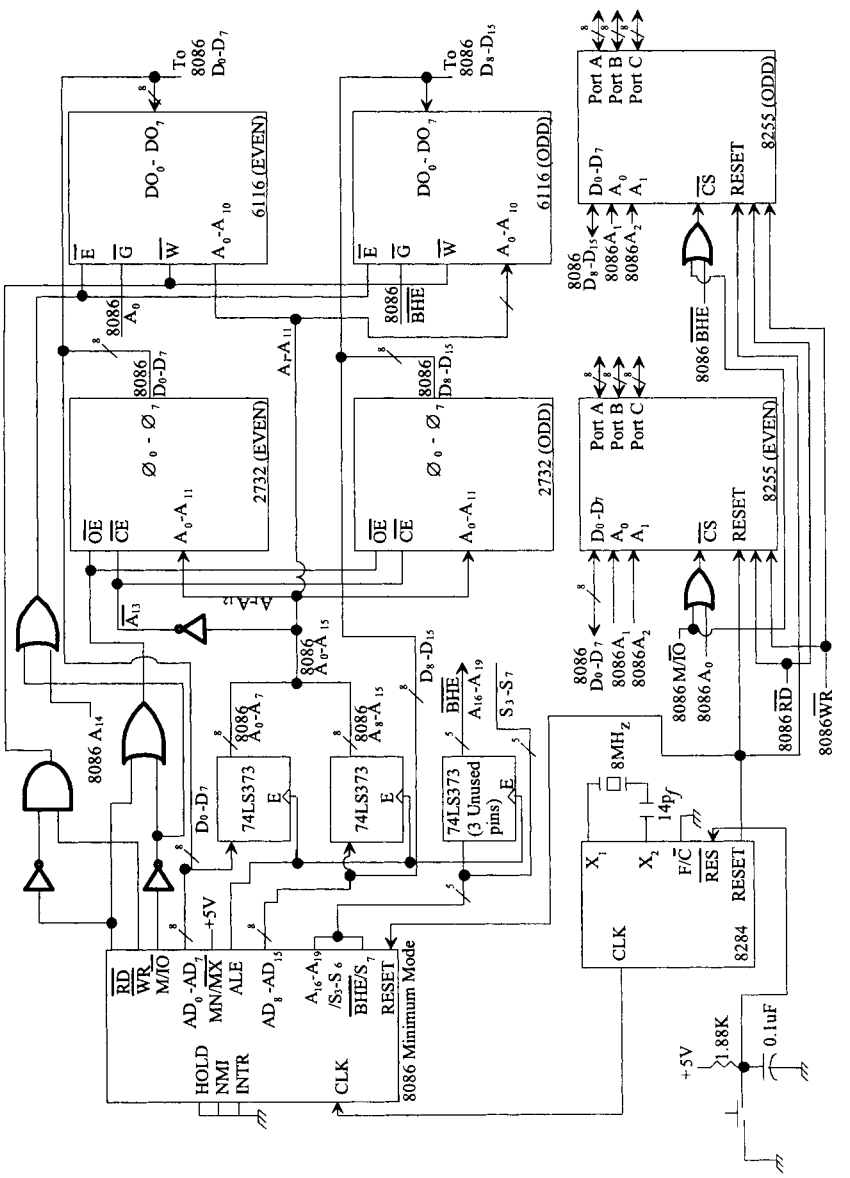


FIGURE 9.20 8086-based microcomputer

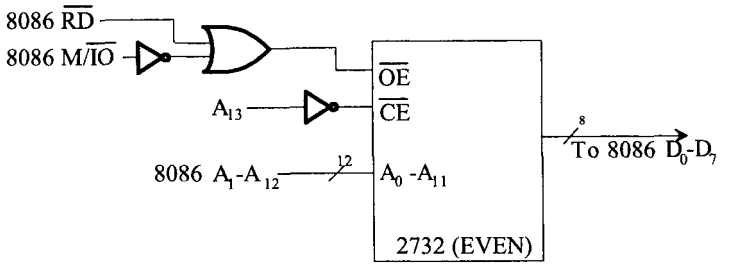


FIGURE 9.21 Even 2732 with pertinent connections

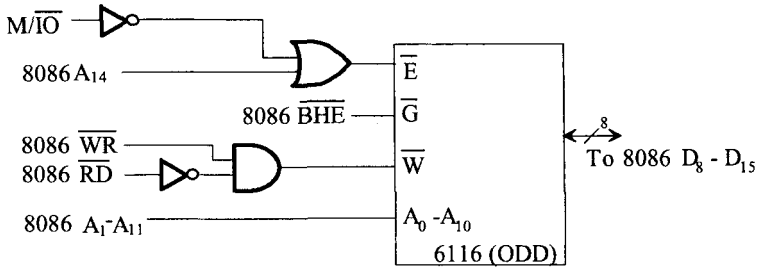


FIGURE 9.22 Odd 6116 with pertinent connections

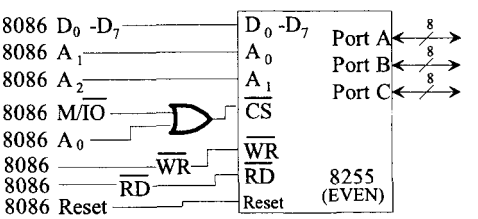


FIGURE 9.23 Even 8255 with pertinent connections

9.10 8086-Based Microcomputer

In this section, an 8086 will be interfaced in minimum mode to provide 4K × 16 EPROM, 2K × 16 static RAM, and six 8-bit I/O ports. The 2732 EPROM, 6116 static RAM, and 8255 I/O chips are used for this purpose. Memory and I/O maps are determined. Figure 9.20 shows a hardware schematic for accomplishing this.

The power and ground pins of all chips must be connected together to the power supply's power and ground pins. The 8086 MN/$\overline{MX}$ is connected to +5 V for minimum mode (single processor) operation. Linear decoding is used to select both EPROMs and SRAMs. 8086 demultiplexed A_{13} = 1 is used to select 2732s and 8086 demultiplexed A_{14} = 0 is used for 6116s. No unused address pin is used for selecting the 8255s because the 8086 M/$\overline{IO}$ pin distinguishes between memory and I/O.

Let us determine the 8086 memory and I/O maps. To determine the memory map for 2732 EPROMs, consider Figure 9.21 (obtained from Figure 9.20), which shows pertinent connections for the even 2732.

In Figure 9.20, M/$\overline{IO}$ = 1 when the 8086 executes a memory-oriented instruction such as `MOV [BX], DL` to access the memory. Also, in the figure, A_{13} = 1 is used to select the EPROMs and A_{14} = 1 is used to deselect the RAMs. This is done to include the 8086 reset vector $FFFF0_{16}$ in the EPROMs. Therefore, an inverter is used to invert A_{13}.

TABLE 9.12 Memory and I/O Maps for the Microcomputer of Figure 9.20

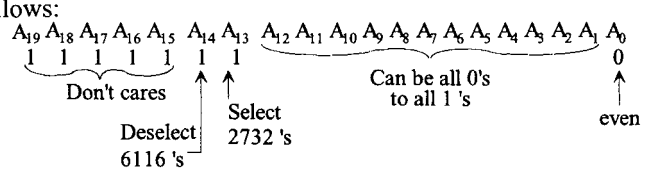
Memory Map

Chip Number		Physical Address	Logical Address	
			Segment Value	Offset
Even	2732 EPROM	FE000H, FE002H, ... , FFFFEH	FE00H	0000H, 0002H, ... , 1FFEH
Odd	2732 EPROM	FE001H, FE003H, ... , FFFFFH	FE00H	0001H, 0003H, ... , 1FFFH
Even	6116 SRAM	F9000H, F9002H, ... , F9FFEH	F900H	0000H, 0002H, ... , 0FFEH
Odd	6116 SRAM	F9001H, F9003H, ... , F9FFFH	F900H	0001H, 0003H, ... , 0FFFH

I/O Map

Chip Number		Port Address
Even	8255	Port A = F8H, Port B = FAH, Port C = FCH, Control Register = FEH
Odd	8255	Port A = F9H, Port B = FBH, Port C = FDH, Control Register = FFH

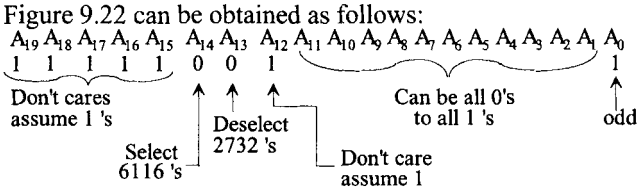
Note that 8086 address pins A_{15}–A_{19} are not used and are, therefore, don't cares. Assume the don't cares to be HIGH. The even memory map for the 2732 in Figure 9.21 can be obtained as follows:



Therefore, the memory map for the even 2732 contains the even addresses FE000H, FE002H, ..., FFFFEH. Similarly, the memory map for the odd 2732 can be determined as: FE001H, FE003H, ..., FFFFFH. Note that the reset vector FFFF0H is included in this map.

Let us now determine the memory map for the odd 6116. Consider Figure 9.22 (obtained from Figure 9.20), which shows pertinent connections for the odd 6116.

In Figure 9.20, $A_{13} = 0$ deselects 2732s and $A_{14} = 0$ selects 6116s. Also, the 8086 outputs HIGH on its M/$\overline{IO}$ pin (M/$\overline{IO}$ = 1) when it executes a memory-oriented instruction such as `MOV CX, [SI]`. Furthermore, the 8086 outputs a LOW on the $\overline{BHE}$ pin for odd addresses. With don't care addresses, pins A_{15}–A_{19} and A_{12} as ones, the odd memory map for the 6116 in Figure 9.22 can be obtained as follows:



Therefore, the memory for the odd 6116 contains the odd addresses F9001H, F9003H, ..., F9FFFH. Similarly, the memory map for the even 6116 can be obtained as F9000H, F9002H, ..., F9FFEH.

Finally, the I/O map for the 8255s is determined. Consider Figure 9.23 (obtained

from Figure 9.20), which shows pertinent connections for the even 8255. The 8086 outputs LOW on its M/$\overline{IO}$ pin (M/$\overline{IO}$ = 0) when it executes an IN or OUT instruction. The 8086 outputs LOW (A_0 = 0) for an even port address. This will produce a LOW on the $\overline{CS}$ pin of the even 8255. The even 8255 will thus be selected.

Using 8086 A_1 and A_2 pins for port addresses, the I/O map for the even 8255 chip can be determined as follows:

Port B	X X X X X 0 1 0 = FAH Don't cares assume 1 's — Port B — even
Port C	X X X X X 1 0 0 = FCH Don't cares assume 1 's — Port C — even
Control Register	X X X X X 1 1 0 = FEH Don't cares assume 1 's — Control register — even

Similarly, the I/O map for the odd 8255 chip is:

Port addresses for the odd 8255		
Port A	=	F9H
Port B	=	FBH
Port C	=	FDH
Control Register	=	FFH

Table 9.12 summarizes the memory and I/O maps.

9.11 8086 Interrupts

The 8086 assigns every interrupt a type code so that the 8086 can identify it. Interrupts can be initiated by external devices or internally by software instructions or by exceptional conditions such as attempting to divide by zero.

9.11.1 Predefined Interrupts

The first five interrupt types are reserved for specific functions.

Type 0:	INT0	Divide by zero
Type 1:	INT1	Single step
Type 2:	INT2	Nonmaskable interrupt (NMI pin)
Type 3:	INT3	Breakpoint
Type 4:	INT4	Interrupt on overflow

The interrupt vectors for these five interrupts are predefined by Intel. The user must provide the desired IP and CS values in the interrupt pointer table. The user may also initiate these interrupts through hardware or software. If a predefined interrupt is not used in a system, the user may assign some other function to the associated type.

The 8086 is automatically interrupted whenever a division by zero is attempted.

This interrupt is nonmaskable and is implemented by Intel as part of the execution of the divide instruction.

When the TF (trap flag) is set by an instruction, the 8086 goes into single-step mode. The TF can be cleared to zero as follows:

```
PUSHF                    ;     Save flags
MOV BP, SP               ;     Move [SP] to [BP]
AND 0[BP], 0FEFFH        ;     Clear TF
POPF                     ;     Pop flags
```

Note here that 0[BP] rather than [BP] is used because BP cannot normally be used without displacement in the 8086 assembler. Now, to set TF, the AND instruction just shown should be replaced by `OR 0[BP],0100H`. Once TF is set to 1, the 8086 automatically generates a type 1 interrupt after execution of each instruction. The user can write a service routine at the interrupt address vector to display memory locations and/or register to debug a program. Single-step mode is nonmaskable and cannot be enabled by the `STI` (enable interrupt) or disabled by the `CLI` (disable interrupt) instruction.

The nonmaskable interrupt is initiated via the 8086 NMI pin. It is edge triggered (LOW to HIGH) and must be active for two clock cycles to guarantee recognition. It is normally used for catastrophic failures such as a power failure. The 8086 obtains the interrupt vector address by automatically executing the `INT2` (type 2) instruction internally.

The type 3 interrupt is used for breakpoints and is nonmaskable. The user inserts the 1-byte instruction `INT3` into a program by replacing an instruction. Breakpoints are useful for program debugging.

The interrupt on overflow is a type 4 interrupt. This interrupt occurs if the overflow flag (OF) is set and the `INTO` instruction is executed. The overflow flag is affected, for example, after execution of a signed arithmetic (such as `IMUL`, signed multiplication) instruction. The user can execute an `INTO` instruction after the `IMUL`. If there is an overflow, an error service routine written by the user at the type 4 interrupt address vector is executed.

9.11.2 Internal Interrupts

The user can generate an interrupt by executing an interrupt instruction `INT`*nn*. The `INT`*nn* instruction is not maskable by the interrupt enable flag (IF). The `INT`*nn* instruction can be used to test an interrupt service routine for external interrupts. Type codes 32–255 can be used; type codes 5 through 31 are reserved by the Intel for future use. If a predefined interrupt is not used in a system, the associate type code can be utilized with the INT*nn* instruction to generate software (internal) interrupts.

9.11.3 External Maskable Interrupts

The 8086 maskable interrupts are initiated via the INTR pin. These interrupts can be enabled or disabled by STI (IF = 1) or CLI (IF = 0), respectively. If IF = 1 and INTR active (HIGH) without occurrence of any other interrupts, the 8086, after completing the current instruction, generates $\overline{\text{INTA}}$ LOW twice, each time for about one cycle.

$\overline{\text{INTA}}$ is only generated by the 8086 in response to INTR, as shown in Figure 9.24. The interrupt acknowledge sequence includes two $\overline{\text{INTA}}$ cycles separated by two clock cycles. ALE is also generated by the 8086 and will load the address latches with indeterminate information. The first $\overline{\text{INTA}}$ bus cycle indicates that an interrupt acknowledge cycle is in progress and allows the system to be ready to place the interrupt type code on the

next $\overline{\text{INTA}}$ bus cycle. The 8086 does not obtain the information from the bus during the first cycle. The external hardware must place the type code on the lower half of the 16-bit data bus (D_0–D_7) during the second cycle.

In the minimum mode, the M/$\overline{\text{IO}}$ is LOW, indicating I/O operation during the $\overline{\text{INTA}}$ bus cycles. The 8086 internal LOCK signal is also LOW from T_2 of the first bus cycle until T_2 of the second bus cycle to keep the BIU from accepting a hold request between the two $\overline{\text{INTA}}$ cycles. Figure 9.25 shows a simplified interconnection between the 8086 and 74LS244 for servicing the INTR. $\overline{\text{INTA}}$ enables the 74LS244 to place type code *nn* on the 8086 data bus. In the maximum mode, the status lines S_0–S_2 will generate the $\overline{\text{INTA}}$ output.

9.11.4 Interrupt Procedures

Once the 8086 has the interrupt type code (via the bus for hardware interrupts, from software interrupt instructions `INT`*nn*, or from the predefined interrupts), the type code is multiplied by 4 to obtain the corresponding interrupt vector in the interrupt vector table. The 4 bytes of the interrupt vector are the least significant byte of the instruction pointer, the most significant byte of the instruction pointer, the least significant byte of the code segment register, and the most significant byte of the code segment register. During the transfer of control, the 8086 pushes the flags and current code segment register and instruction pointer onto the stack. The new CS and IP values are loaded. Flags TF and IF are then cleared to zero. The CS and IP values are read by the 8086 from the interrupt vector table. No segment registers are used when accessing the interrupt pointer table. S_4S_3 has the value 10_2 to indicate no segment register selection.

9.11.5 Interrupt Priorities

As far as the 8086 interrupt priorities are concerned, the single-step interrupt has the highest priority, followed by NMI, followed by the software interrupts. This means that a

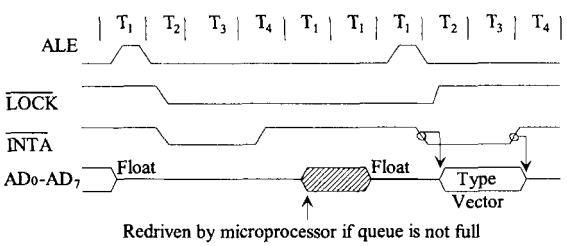


FIGURE 9.24 $\overline{\text{INTA}}$ Cycle

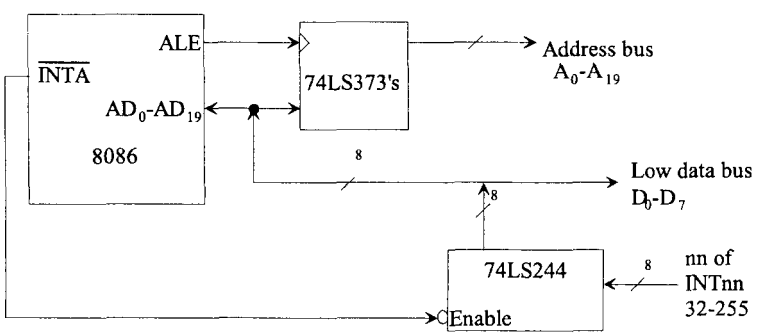


FIGURE 9.25 Servicing the INTR in the minimum mode

simultaneous NMI and single-step interrupt will cause the NMI service routine to follow the single step; a simultaneous software interrupt and single step interrupt will cause the software interrupt service routine to follow the single step; and a simultaneous NMI and software interrupt will cause the NMI service routine to be executed prior to the software interrupt service routine. The INTR is maskable and has the lowest priority. A priority interrupt controller such as the 8259A can be used with the 8086 INTR to provide eight levels of interrupts. The 8259A has built-in features for expansion of up to 64 levels with additional 8259s. The 8259A is programmable and can be readily used with the 8086 to obtain multiple interrupts from the single 8086 INTR pin.

9.11.6 Interrupt Pointer Table

The interrupt pointer table provides interrupt address vectors (IP and CS contents) for all the interrupts. There may be up to 256 entries for the 256 type codes. Each entry consists of two addresses, one for storing IP and the other for storing CS. Note that in the 8086 each interrupt address vector is a 20-bit address obtained from IP and CS.

To service an interrupt, the 8086 calculates the two addresses in the pointer table where IP and CS are stored for a particular interrupt type as follows:

For INT*nn*
(*nn* ← Type code)

The table address for IP = 4 × *nn* and the table address for CS = 4 × *nn* + 2. For example, consider INT2:

Address for IP = 4 × 2 = 00008H
Address for CS = 00008 + 2 = 0000AH

The values of IP and CS are loaded from location 00008H and 0000AH in the pointer table. Similarly, the IP and CS addresses for other INT*nn* are calculated, and their values are obtained from the contents of these addresses in the pointer table (Table 9.13). The 8086 interrupt vectors are defined as follows:

Vectors 0–4	For predefined interrupts
Vectors 5–31	For Intel's future use
Vectors 32–255	For user interrupts

Interrupt service routines should be terminated with an IRET (interrupt return) instruction, which pops the top three stack words into the IP, CS, and flags, thus returning control to the right place in the main program.

9.12 8086 DMA

When configured in minimum mode (MN/$\overline{\text{MX}}$ HIGH) the 8086 provides HOLD and HLDA (hold acknowledge) signals to control the system bus for DMA applications. In this type of DMA, the peripheral device can request the DMA transfer via the DMA request (DRQ) line connected to a DMA controller chip such as the 8257. In response to this request, the 8257 sends a HOLD signal to the 8086. The 8257 then waits for the HLDA signal from the 8086. On receipt of this HLDA, the 8257 sends a $\overline{\text{DMACK}}$ signal to the peripheral device. The 8257 then takes over the bus and controls data transfer between the RAM and peripheral device. On completion of data transfer, the 8257 returns control to the 8086 by disabling the HOLD and $\overline{\text{DMACK}}$ signals.

TABLE 9.13 8086 Interrupt Pointer Table

Interrupt Type Code		*20-Bit Memory Address*
0	IP	00000H
	CS	00002H
1	IP	00004H
	CS	00006H
2		00008H
		0000AH
.		.
.		.
.		.
255	IP	003FCH
	CS	003FEH

Example 9.21

In Figure 9.26, an 8086-based microcomputer is required to implement a voltmeter to measure voltage in the range 0 to 5 V and display the result in two decimal digits: one integer part and one fractional part. The microcomputer is required to start the A/D converter at the falling edge of a pulse via bit 0 of Port C. When the conversion is completed, the A/D's "conversion complete" signal will go HIGH. During the conversion, the A/D's "conversion complete" signal stays LOW. Use the 8255 control register = FEH, Port A = F8H, Port B = FAH, and Port C = FCH.

Using programmed I/O, the microcomputer is required to poll the A/D's "conversion complete" signal. When the conversion is completed, the microcomputer will send a LOW of the A/D converter's "output enable" line via bit 1 to port C and then input the 8-bit output from A/D via port B and display the voltage (0 to 5 V) in two decimal digits (one integer and one fractional) via port A on two TIL 311 displays. Note that the TIL 311 has an on-chip BCD to seven-segment decoder. The microcomputer will output each decimal digit on the common lines (bits 0–3 of port A) connected to the DCBA inputs of the displays. Each display will be enabled by outputting LOW on each $\overline{\text{LATCH}}$ line

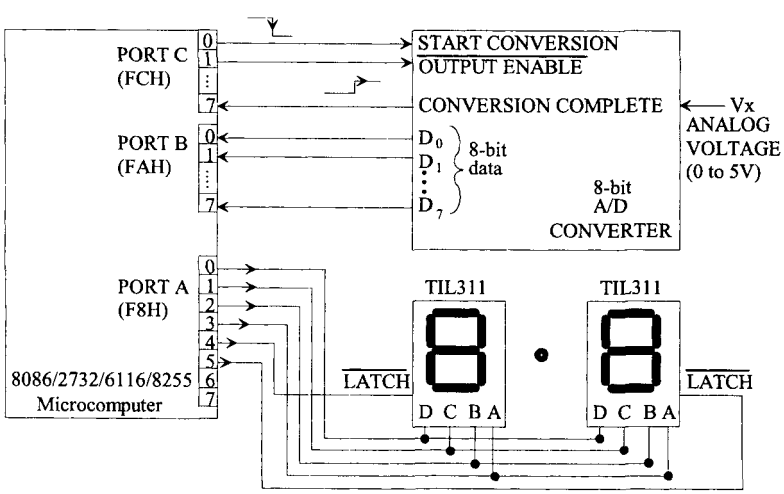


FIGURE 9.26 Figure for Example 9.21

in sequence (one after another) so that the input voltage V_x (0 to 5 V) will be displayed with one integer part and fractional part. Write an 8086 assembly language program to accomplish this.

Using interrupt I/O (both NMI and INTR), repeat the task. Write the main program to initialize the 8255 control register and start the A/D. The service routine will input the A/D data, display the result, and stop. Write an 8086 assembly language program for the main program and the service routine. Use the memory map of your choice. Write the service routines for both NMI and INTR starting at IP=2000H, CS=1000H. Use 8086 assembler directive such as ORG CS:IP for the HP (Hewlett-Packard) 64XXX microcomputer development system in the following programs.

Solution

Because the maximum decimal value that can be accommodated in 8 bits is 255_{10} (FF_{16}), the maximum voltage of 5 V will be equivalent to 255_{10}. This means the display in decimal is given by

$$D = 5 \times (\text{Input}/255)$$
$$= \underbrace{\text{Quotient}}_{\text{Integer part}} + \text{Remainder}$$

This gives the integer part. The fractional part in decimal is

$$F = (\text{Remainder}/51) \times 10$$
$$\cong (\text{Remainder})/5$$

For example, suppose that the decimal equivalent of the 8-bit output of A/D is 200.

$$D = 200/51 \Rightarrow \textit{Quotient} = 3, \textit{Remainder} = 47$$
$$\text{Integer part} = 3$$
$$\text{Fractional part, } F = 47/5 = 9$$

Therefore, the display will show 3.9 V.

(a) The 8086 assembly language program using programmed I/O can be written as follows:

```
ORG    0FE00H:0100H; CS=FE00H, IP= 0100H
CDSEG. SEGMENT
       ASSUME CS:CDSEG
PORTA  EQU    0F8H
PORTB  EQU    0FAH
PORTC  EQU    0FCH
CNTRL  EQU    0FEH
MOV    AL,8AH          ;      Configure PORTA, PORTB
       OUT    CNTRL,AL        ;      and PORTC
       MOV    AL,03H          ;      Send 1 to START pin of A/D
OUT    PORTC,AL       ;      and 1 to (OUTPUT ENABLE)
       MOV    AL,02H          ;      Send 0 to start pin
       OUT    PORTC,AL        ;      of A/D
BEGIN: IN     AL,PORTC        ;      Check conversion
       ROL    AL,1            ;      Complete bit for HIGH
       JNC    BEGIN
       MOV    AL,00H          ;      Send LOW to (OUTPUT ENABLE)
       OUT    PORTC,AL
       IN     AL,PORTB        ;      Input A/D data
       MOV    AH,0            ;      Convert input data to 16-bit
                              ;      unsigned number in AX
```

```
        MOV     DL,51           ;       Convert data to
        DIV     DL              ;       integer part
        MOV     CL,AL           ;       Save quotient (integer) in CL
        XCHG    AH,AL           ;       Move remainder to AL
        MOV     AH,0            ;       Convert remainder to unsigned
                                ;       16-bit number
        MOV     BL,5            ;       Convert data to
        DIV     BL              ;       fractional part
        MOV     DL,AL           ;       Save quotient (fraction) to DL
        MOV     AL,CL           ;       Move integer part
        OR      AL,20H          ;       Disable fractional display
        AND     AL,2FH          ;       Enable integer display
        OUT     PORTA,AL        ;       Display integer part
        MOV     AL,DL           ;       Move fractional part
        OR      AL,10H          ;       Disable integer display
        AND     AL,1FH          ;       Enable fractional display
        OUT     PORTA,AL        ;       Display fractional part
        HLT
CDSEG   ENDS
        END
```

(b) Using NMI

In Figure 9.26, connect the "conversion complete" to 8086 NMI; all other connections in Figure 9.26 will remain unchanged. Note that all addresses selectable by the user are arbitrarily chosen in the following. The main program in 8086 assembly language is

```
            ORG     3900H:0100H ; SS = 3900H, SP = 0100H
STSEG       SEGMENT
            DB      32 DUP (?)
STSEG       ENDS
            END
PORTA       EQU     0F8H
PORTB       EQU     0FAH
PORTC       EQU     0FCH
CNTRL       EQU     0FEH
            ORG     0FE00H:0100H ; CS = FE00H, IP = 0100H
CDSEG       SEGMENT
            ASSUME CS:CDSEG,SS:STSEG,DS:DATA
            MOV     AX,3900H        ;   Initialize
            MOV     SS,AX           ;   stack segment
            MOV     AX,0000H        ;   Initialize
            MOV     DS,AX           ;   data segment
            MOV     SP,0100H        ;   Initialize SP
            MOV     AL,8AH      ;   Configure PORTA, PORTB
            OUT         CNTRL,AL        ;   and PORTC
            MOV     AL,03H      ;   Send 1 to START pin of A/D
            OUT     PORTC,AL        ;   and 1 to (OUTPUT ENABLE)
            MOV     AL,02H      ;   Send 0 to start pin
            OUT     PORTC,AL        ;   of A/D
DELAY: JMP    DELAY             ;   Wait for interrupt
CDSEG       ENDS
            END
```

```
ORG     0000H:0008H          ;  DS = 0000H, Offset = 0008H
DATA    SEGMENT
        DW      2000H        ;  Initialize IP = 2000H,
        DW      1000H        ;  CS = 1000H
DATA    ENDS                 ;  for Pointer Table
        END
```

The NMI Service routine is:

```
        ORG     1000H:2000H  ;  CS = 1000H, IP = 2000H
CODE    SEGMENT              ;  Start Program at
        ASSUME CS:CODE       ;  CS = 1000H, IP = 2000H
        MOV     AL,00H       ;  Send LOW to (OUTPUT ENABLE)
        OUT     PORTC,AL
        IN      AL,PORTB     ;  Input A/D data
        MOV     AH,0         ;  Convert input to 16-bit unsig num.
        MOV     DL,51        ;  Convert data to
        DIV     DL           ;  integer part
        MOV     CL,AL        ;  Save quotient (integer) in CL
        XCHG    AH,AL        ;  Move remainder to AL
        MOV     AH,0         ;  Convert remainder to unsigned 16-bit
        MOV     BL,5         ;  Convert data to
        DIV     BL           ;  fractional part
```

```
        MOV     DL,AL        ;  Save quotient (fraction) to DL
        MOV     AL,CL        ;  Move integer part
        OR      AL,20H       ;  Disable fractional display
        AND     AL,2FH       ;  Enable integer display
        OUT     PORTA,AL     ;  Display integer part
        MOV     AL,DL        ;  Move fractional part
        OR      AL,10H       ;  Disable integer display
        AND     AL,1FH       ;  Enable fractional display
        OUT     PORTA,AL     ;  Display fractional part
        HLT                  ;  Stop
CODE    ENDS
        END
```

(c) Using INTR

All connections in Figure 9.26 will be same except A/D's "conversion complete" to 8086 INTR as shown in Figure 9.27. All other connections in Figure 9.26 will remain unchanged. INT FFH is used. In response to INTR, the 8086 pushes IP and SR onto the stack, and generates LOW on $\overline{\text{INTA}}$. An octal buffer such as 74LS244 can be enabled by this $\overline{\text{INTA}}$ to transfer FF_{16} in this case (can be entered via eight DIP switches connected to + 5 V through a 1 KΩ resistor) to the input of the octal buffer. The output of the octal buffer is connected to the demultiplexed D_0–D_7 lines of the 8086. The 8086 executes INT FFH and goes to the interrupt pointer table to load the contents of physical addresses 003FCH (logical address:

CS = 0000H, IP = 03FCH) and 003FEH (logical address: CS = 0000H, IP = 03FEH) to obtain IP and CS for the service routine respectively. Suppose that it is desired to write the service routine at IP = 2000H and CS = 1000H; these IP and CS values must be stored at addresses 003FCH and 003FEH respectively. All user selectable addresses are arbitrarily chosen. The main program in 8086 assembly language is

```
ORG    3900H:8500H   ;       SS = 3900H, SP = 8500H
STSEG  SEGMENT
       DB     32 DUP (?)
STSEG  ENDS
       END
PORTA  EQU    0F8H
PORTB  EQU    0FAH
PORTC  EQU    0FCH
CNTRL  EQU    0FEH
       ORG    0F300H:0100H  ;  CS = F300H, IP = 0100H
CDSEG  SEGMENT
       ASSUME CS:CDSEG, SS:STSEG,DS:DATA
       MOV    AX,3900H      ;  Initialize
       MOV    SS,AX         ;  stack segment
       MOV    AX,0000H      ;  Initialize
       MOV    DS,AX         ;  data segment


MOV    SP,8500H      ;       Initialize SP
       MOV    AL,8AH        ;  Configure port A, port B,
       OUT    CNTRL,AL      ;  and port C
       STI                  ;  Enable Interrupt
       MOV    AL,03H        ;  Send one to start pin of A/D
       OUT    PORTC,AL      ;  and one to (OUTPUT ENABLE)
       MOV    AL,02H        ;  Send zero to start pin of A/D
       OUT    PORTC,AL
DELAY: JMP    DELAY         ;  Wait for interrupt
CDSEG  ENDS
       END
       ORG    0000H:03FCH   ;  DS = 0000H, Offset = 03FCH
DATA   SEGMENT
       DW     2000H         ;  Initialize IP = 2000H,
       DW     1000H         ;  CS = 1000H
DATA   ENDS                 ;         for Pointer Table
       END
```

The INTR Service routine is:

```
ORG    1000H:2000H          ;  CS = 1000H, IP = 2000H
CODE   SEGMENT
       ASSUME CS:CODE
       MOV     AL,0         ;  Send LOW to
       OUT     PORTC,AL     ;  (OUTPUT ENABLE)
       IN      AL,PORTB     ;  Input A/D data
       MOV     AH,0         ;  Convert input data to
                            ;  16-bit unsigned number in AX
       MOV     DL,51        ;  Convert data
       DIV     DL           ;  to integer part
       MOV     CL,AL        ;  Save quotient (integer) in CL
       XCHG    AH,AL        ;  Move remainder to AL
       MOV     AH,0         ;  Convert remainder to unsigned 16-bit
       MOV     BL,5         ;  Convert data
       DIV     BL           ;  to fractional part
       MOV     DL,AL        ;  Save quotient (fraction) in DL
       MOV     AL,CL        ;  Move integer part
       OR      AL,20H       ;  Disable fractional display
       AND     AL,2FH       ;  Enable integer display
       OUT     PORTA,AL     ;  Display integer part
       MOV     AL,DL        ;  Move fractional part
       OR      AL,10H       ;  Disable integer display
       AND     AL,1FH       ;  Enable fraction display
       OUT     PORTA,AL     ;  Display fractional part
       HLT                  ;  Stop
CODE   ENDS
       END
```

9.13 Interfacing an 8086-Based Microcomputer to a Hexadecimal Keyboard and Seven-Segment Displays

This section describes the characteristics of the 8086-based microcomputer used with a hexadecimal keyboard and a seven-segment display.

9.13.1 Basics of Keyboard and Display Interface to a Microcomputer

A common method of entering programs into a microcomputer is via a keyboard. A popular way of displaying results by the microcomputer is by using seven-segment displays. The main functions to be performed for interfacing a keyboard are:

Sense a key actuation.

Debounce the key.

Decode the key.

Let us now elaborate on keyboard interfacing concepts. A keyboard is arranged in rows and columns. Figure 9.28 shows a 2 × 2 keyboard interfaced to a typical microcomputer. In Figure 9.28, the columns are normally at a HIGH level. A key actuation is sensed by sending a LOW (closing the diode switch) to each row one at a time via PA0 and PA1 of port A. The two columns can then be input via PB2 and PB3 of port B to see whether any of the normally HIGH columns are pulled LOW by a key actuation. If so, the rows can be

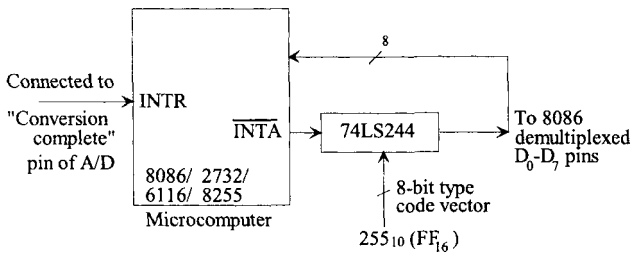


FIGURE 9.27 Hardware interface for 8086 INTR

checked individually to determine the row in which the key is down. The row and column code for the pressed key can thus be found.

The next step is to debounce the key. Key bounce occurs when a key is pressed or released—it bounces for a short time before making the contact. When this bounce occurs, it may appear to the microcomputer that the same key has been actuated several times instead of just once. This problem can be eliminated by reading the keyboard after about 20 ms and then verifying to see if it is still down. If it is, then the key actuation is valid. The next step is to translate the row and column code into a more popular code such as hexadecimal or ASCII. This can easily be accomplished by a program. Certain characteristics associated with keyboard actuations must be considered while interfacing to a microcomputer. Typically, these are two-key lockout and *N*-key rollover. The two-key lockout ensures that only one key is pressed. An additional key depressed and released does not generate any codes. The system is simple to implement and most often used. However, it might slow down the typing because each key must be fully released before the next one is pressed down. On the other hand, the *N*-key rollover will ignore all keys pressed until only one remains down.

Now let us elaborate on the interfacing characteristics of typical displays. The following functions are typically performed for displays:

1. Output the appropriate display code.
2. Output the code via right entry or left entry into the displays if there are more than one displays.

These functions can easily be realized by a microcomputer program. If there are more than one display, the displays are typically arranged in rows. A row of four displays is shown in Figure 9.29. In the figure, one has the option of outputting the display code via right entry or left entry. If the code is entered via right entry, the code for the least significant digit of the four-digit display should be output first, then the next digit code, and so on. The program outputs to the displays are so fast that visually all four digits will appear on the display simultaneously. If the displays are entered via left entry, then the most significant digit must be output first and the rest of the sequence is similar to the right entry.

Two techniques are typically used to interface a hexadecimal display to the microcomputer: nonmultiplexed and multiplexed. In nonmultiplexed methods, each hexadecimal display digit is interfaced to the microcomputer via an I/O port. Figure 9.30 illustrates this method. BCD to seven-segment conversion is done in software. The microcomputer can be programmed to output to the two display digits in sequence. However, the microcomputer executes the display instruction sequence so fast that the displays appear to the human eye at the same time. Figure 9.31 illustrates the multiplexing method of interfacing the two hexadecimal displays to the microcomputer. In the multiplexing scheme, appropriate seven-segment code is sent to the desired displays on

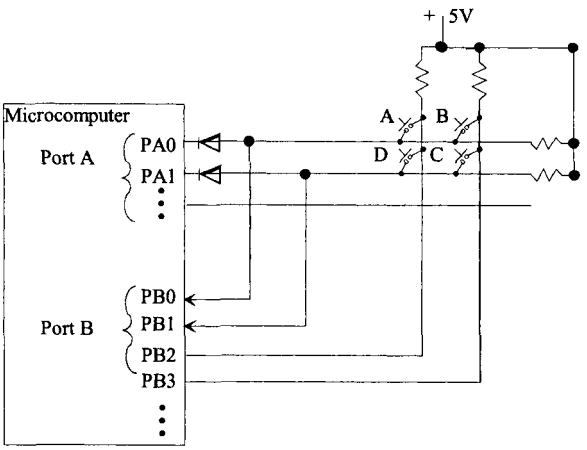


FIGURE 9.28 Typical microcomputer-keyboard interface

FIGURE 9.29 A row of four displays

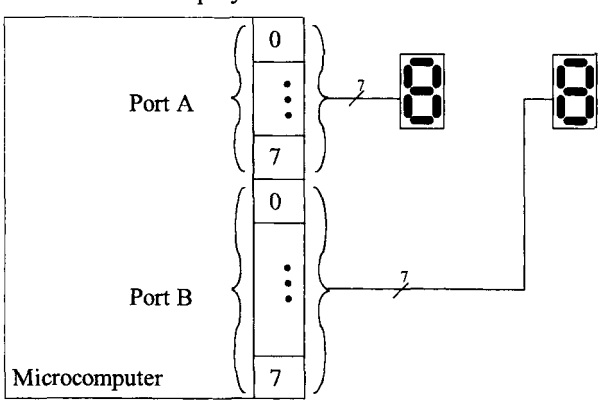


FIGURE 9.30 Nonmultiplexed hexadecimal displays

seven lines common to all displays. However, the display to be illuminated is grounded. Some displays such as Texas Instrument's TIL 311 have on-chip decoder. In this case, the microcomputer is required to output four bits (decimal) to a display.

The keyboard and display interfacing concepts described here can be realized by either software or hardware. To relieve the microprocessor of these functions, microprocessor manufacturers have developed a number of keyboard/display controller chips. These chips are typically initialized by the microprocessor. The keyboard/display functions are then performed by the chip independent of the microprocessor. The amount of keyboard/display functions performed by the controller chip varies from one manufacturer to another. However, these functions are usually shared between the controller chip and the microprocessor.

9.13.2 Hex Keyboard Interface to an 8086-Based Microcomputer

In this section, an 8086-based microcomputer is designed to display a hexadecimal digit

entered via a keypad (16 keys). Figure 9.32 shows the hardware schematic.

1. Port A is configured as an input port to receive the row–column code.
2. Port B is configured as an output port to display the key(s) pressed.
3. Port C is configured as an output port to output zeros to the rows to detect a key actuation.

The system is designed to run at 2 MHz. Debouncing is provided to avoid unwanted oscillation caused by the opening and closing of the key contacts. To ensure stability for the input signal, a delay of 20 ms is used for debouncing the input.

The program begins by performing all necessary initializations. Next, it makes sure that all the keys are opened (not pressed). A delay loop of 20 ms is included for debouncing, and the following instruction sequence is used (Section 9.8):

```
        MOV   CX,0930H
DELAY:  LOOP  DELAY
```

The next three lines detect a key closure. If a key closure is detected, it is debounced. It is necessary to determine exactly which key is pressed. To do this, a sequence of row-control codes (0FH, 0EH, 0DH, 0BH, 07H) are output via port C. The row–column code is input via port A to determine if the column code changes corresponding to each different row code. If the column code is not 0FH (changed), the input key is identified. The program then indexes through a look-up table to determine the row–column code saved in DL. If the code is found, the corresponding index value, which equals the input

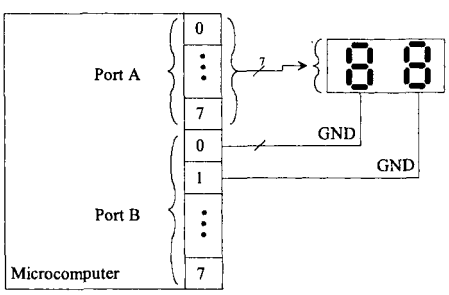


FIGURE 9.31 Multiplexed displays

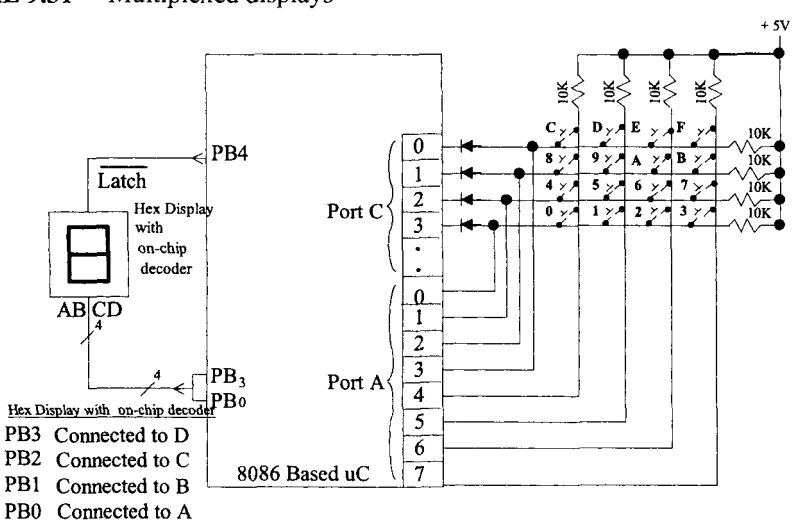


FIGURE 9.32 8086-based microcomputer interface to keyboard and display

key's value (a single hexadecimal digit) is displayed. The program is written such that it will continuously scan for input key and update the display for each new input. Note that lowercase letters are used to represent the 8086 registers in the program. For example, `al`, `ah`, and `ax` in the program represent the 8086 AL, AH, and AX registers, respectively.

The memory and I/O maps are arbitrarily chosen. A listing of the 8086 assembly language program is given in the following:

```
0000                    CDSEG   SEGMENT
                        ASSUME CS:CDSEG,DS:DTSEG
 = 00F8         PORTA   EQU     0F8h        ; Hex keyboard input
                                            ; (row/column)
 = 00FA         PORTB   EQU     0FAh        ; LED displays/controls
 = 00FC         PORTC   EQU     0FCh        ; Hex keyboard row controls
 = 00FE         CSR     EQU     0FEh        ; Control status register
 = 00F0         OPEN    EQU     0F0h        ; Row/column codes if all
                                            ;  keys are opened
 0000  BB 0100          mov     bx, 0100h
 0003  8E DB            mov     ds, bx
 0005  B0 90  start:    mov     al, 90h     ; Config ports A, B, C
                                            ; as i/o/o
 0007  E6 FE            out     CSR, al
 0009  2A C0            sub     al, al      ; Clear al
 000B  E6 FA            out     PORTB,al    ; Enable/initialize display
 000D  2A C0  scan_key:sub      al, al      ; Clear al
 000F  E6 FC            out     PORTC, al   ; Set row controls to zero
 0011  E4 F8  key_open:in       al, PORTA   ; Read PORTA
0013  3C F0             cmp     al, OPEN    ; Are all keys opened?
 0015  75 FA            jnz     key_open    ; Repeat if closed
 0017  B9 0930          mov     cx, 0930h   ; Delay of 20 ms
 001A  E2 FE  delay1: loop      delay1      ;  key opened
 001C  E4 F8  key_close:in      al, PORTA   ; read PORTA
 001E  3C F0            cmp     al, OPEN    ; Are all keys closed?
 0020  74 FA            jz      key_close   ; repeat if opened
 0022  B9 0930          mov     cx, 0930h   ; delay of 20 ms
 0025  E2 FE delay2: loop       delay2      ; Debounce key closed
 0027  B0 FF            mov     al, 0FFh    ; Set al to all 1's
 0029  F8               clc                 ; carry
 002A  D0 D0 next_row: rcl      al, 1       ; Set up row mask
 002C  8A C8            mov     cl, al      ; Save row mask in cl
 002E  E6 FC            out     PORTC, al   ; Set a row to zero
 0030  E4 F8            in      al, PORTA   ; Read PORTA
 0032  8A D0            mov     dl, al      ; Save row/coln codes in dl
 0034  24 F0            and     al, 0F0h    ; Mask row code
 0036  3C F0            cmp     al, 0F0h    ; Is coln code affected?
 0038  75 05            jnz     decode      ; If yes, decode coln code
 003A  8A C1            mov     al, cl      ; Restore row mask to al
 003C  F9               stc                 ; if no, set carry
 003D  EB EB            jmp     next_row    ; Check next row
003F  BE FFFF decode: mov      si, -1      ; Initialize index register
 0042  B9 000F          mov     cx, 000Fh   ; Set up counter
 0045  46      search: inc      si          ; Increment index
```

```
0046  3A 94 0000 R   cmp     dl,[TABLE+si]; Index thru table of
                                          ;      codes
004A  E0 F9           loopne  search     ; Loop if not found
004C  8A C1  done:    mov     al,cl      ; get character and enable
                                          ; display
004E  E6 FA           out     PORTB,al   ; display key
0050  EB BB           jmp     scan_key   ; Return to scan another
                                          ; key input
0052          CDSEG   ends
0000          DTSEG   segment
0000  77      TABLE   DB      77h        ; Code for F
0001  B7              DB      0B7h       ; Code for E
0002  D7              DB      0D7h       ; Code for D
0003  E7              DB      0E7h       ; Code for C
0004  7B              DB      7Bh        ; Code for B
0005  BB              DB      0BBh       ; Code for A
0006  DB              DB      0DBh       ; Code for 9
0007  EB              DB      0EBh       ; Code for 8
0008  7D              DB      7Dh        ; Code for 7
0009  BD              DB      0BDh       ; Code for 6
000A  DD              DB      0DDh       ; Code for 5
000B  ED              DB      0EDh       ; Code for 4
000C  7E              DB      7Eh        ; Code for 3
000D  BE              DB      0BEh       ; Code for 2
000E  DE              DB      0DEh       ; Code for 1
000F  EE              DB      0EEh       ; Code for 0
0010          DTSEG   ends
                      end
```

In the program, the "Key-open" loop ensures that no keys are closed. On the other hand, the "Key-close" waits in the loop for a key actuation. Note that in this program, the table for the codes for the hexadecimal numbers 0 through F are obtained by inspecting Figure 9.32.

For example, consider key F. When key F is pressed and if a LOW is output by the program to bit 0 of port C, the top row and the rightmost column of the keyboard will be LOW. This will make the content of port A as:

Bit number :	7	6	5	4	3	2	1	0	
Data :	0	1	1	1	0	1	1	1	$= 77_{16}$
			7				7		

Thus, a code of 77_{16} is obtained at Port A when the key F is pressed. Diodes are connected at the four bits (Bits 0-3) of Port C. This is done to make sure that when a 0 is output by the program to one of these bits (row of the keyboard), the diode switch will close and will generate a LOW on that row.

Now, if a key is pressed on a particular row which is LOW, the column connected to this key will also be LOW. This will enable the programmer to obtain the appropriate key code for each key.

QUESTIONS AND PROBLEMS

9.1 What is the basic difference between the 8086, 8086-1, 8086-2, and 8086-4?

9.2 Assume (DS)=1000H, (SS)=2000H, (CS)=3000H, (BP)=000FH, (BX)=000AH before execution of the following 8086 instructions:
(a) MOV CX,[BX] (b) MOV DX,[BP]
Which instruction will be executed faster by the 8086, and why ?

9.3 What is the purpose of the 8086 MN/$\overline{\text{MX}}$ pin?

9.4 If (DS) = 205FH and OFFSET = 0052H, what is the 8086 physical address? Does the EU or BIU compute this physical address?

9.5 In an 8086 system, SEGMENT 1 contains addresses 00100H–00200H and SEGMENT 2 also contains addresses 00100H–00200H. What are these segments called?

9.6 Determine the addressing modes for the following 8086 instructions:
(a)
```
CLC
```
(b)
```
CALL  WORDPTR [BX]
```
(c)
```
MOV   AX, DX
```
(d)
```
ADD   [SI], BX
```

9.7 Find the overflow, direction, interrupt, trap, sign, zero, parity, and carry flags after execution of the following 8086 instruction sequence:
```
MOV   AH, 0FH
SAHF
```

9.8 What is the content of AL after execution of the following 8086 instruction sequence?
```
MOV   BH, 33H
MOV   AL, 32H
ADD   AL, BH
AAA
```

9.9 What happens after execution of the following 8086 instruction sequence? Comment.
```
MOV   DX, 001FH
XCHG  DL, DH
MOV   AX, DX
IDIV  DL
```

9.10 What are the remainder, quotient, and registers containing them after execution of the following 8086 instruction sequence?
```
MOV   AH, 0
MOV   AL, 0FFH
MOV   CX, 2
IDIV  CL
```

9.11 Write an 8086 instruction sequence to set the trap flag for single stepping without affecting the other flags in the Status register.

9.12 Write an 8086 assembly language program to subtract two 64-bit numbers. Assume SI and DI point to the low words of the numbers.

9.13 Write an 8086 assembly program to add a 16-bit number stored in BX (bits 0 to 7 containing the high-order byte of the number and bits 8 to 15 containing the low-order byte) with another 16-bit number stored in CX (bits 0 to 7 containing the low-order 8 bits of the number and bits 8 thorough 15 containing the high-order 8 bits). Store the result in AX.

9.14 Write an 8086 assembly program to multiply the top two 16-bit unsigned words of the stack. Store the 32-bit result onto the stack.

9.15 Write an 8086 assembly language program to add three 16-bit numbers. Store the 16-bit result in AX.

9.16 Write an 8086 assembly language to find the area of a circle with radius 2 meters and save the result in AX.

9.17 Write an 8086 assembly language program to convert 255 degrees in Celsius in BL to Fahrenheit degrees and store the value in AX. Use the equation

$$F = (C/5) * 9 + 32$$

9.18 Assume AL, CX and DXBX contain a signed byte, a signed word, and a signed 32-bit number respectively. Write an 8086 assembly language program that will compute the signed 32-bit result: AL – CX + DXBX → DXBX.

9.19 Write an 8086 assembly program to divide an 8-bit signed number in CH by an 8-bit signed number in CL. Store the quotient in CH and the remainder in CL.

9.20 Write an 8086 assembly program to add 25 16-bit numbers stored in consecutive memory locations starting at displacement 0100H in DS = 0020H. Store the 16-bit result onto the stack.

9.21 Write an 8086 assembly program to find the minimum value of a string of 10 signed 8-bit numbers using indexed addressing. Assume Offset 5000H contains the first number.

9.22 Write an 8086 assembly program to move 100 words from a source with offset 0010H in ES to a destination with offset 0100H in the same extra segment.

9.23 Write an 8086 assembly program to divide a 28-bit unsigned number in the high 28 bits of DX AX by 8_{10}. Do not use any divide instruction. Store the quotient in the low 28 bits of DX AX. Discard remainder.

9.24 Write an 8086 assembly program to compare two strings of 15 ASCII characters. The first character (string 1) is stored starting at offset 5000H in DS followed

by the string. The first character of the second string (string 2) is stored starting at 6000H in ES. The ASCII character in the first location of string 1 will be compared with the first ASCII character of string 2, and so on. As soon as a match is found, store $00EE_{16}$ onto the stack; otherwise, store 0000_{16} onto the stack.

9.25 Write a subroutine in 8086 assembly language that can be called up by a main program in a different code segment. The subroutine will compute the 16-bit sum

$$\sum_{i=1}^{100} X_i$$

Assume the X_i's are signed 8-bit numbers and are stored in consecutive locations starting at displacement 0050H. Also, write the main program that will call this subroutine to compute

$$\sum_{i=1}^{100} \frac{X_i}{100}$$

and store the 16-bit result (8-bit remainder and 8-bit quotient) in two consecutive memory bytes starting at offset 0400H.

9.26 Write a subroutine in 8086 assembly language to convert a 2-digit unpacked BCD number to binary. The most significant digit is stored in a memory location starting at offset 4000H, and the least significant digit is stored at offset 4001H. Store the binary result in DL.Use the value of the 2-digit BCD number, $V = D_1 \times 10 + D_0$. Note that arithmetic operations will provide binary result.

9.27 Assume an 8086/2732/6116/8255 microcomputer. Suppose that four switches are connected at bits 0 through 3 of port A and an LED is connected at bit 4 of port B. If the number of LOW switches is even, turn the port B LED ON; otherwise, turn the port B LED OFF. Write an 8086 assembly language program to accomplish this. Do not use any instructions involving the Parity flag.

9.28 Interface two 2732 and one 8255 odd to an 8086 to obtain even and odd 2732 locations and odd addresses for the 8255's port A, port B, port C, and control registers. Show only the connections for the pins shown in Figure P9.28. Assume all unused address lines to be zeros.

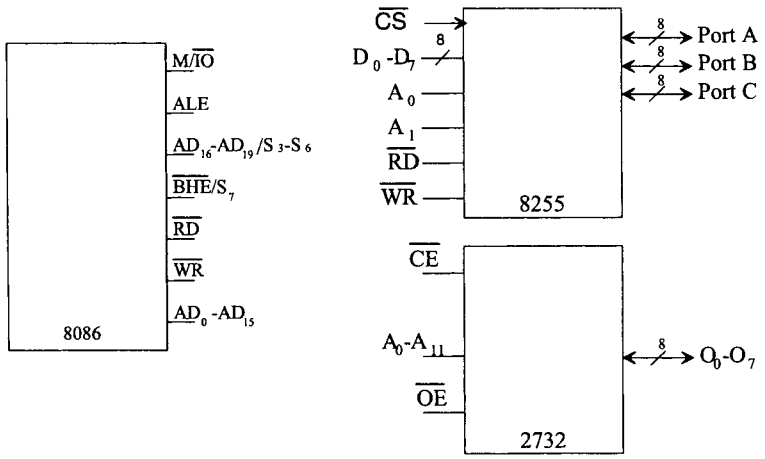


FIGURE P9.28

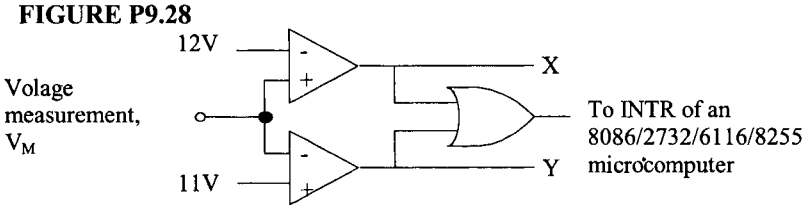


FIGURE P9.29

9.29 In Figure P9.29, if $V_M > 12$ V, turn the LED ON connected at bit 4 of port A. On the other hand, if $V_M < 11$ V, turn the LED OFF. Use ports, registers, and memory locations of your choice. Draw a hardware block diagram showing the microcomputer and the connections of the figure to its ports. Write a service routine in 8086 assembly language. Assume all segment registers are already initialized. The service routine should be written as CS=1000H, IP=2000H. The main program will initialize SP to 2050H, initialize ports, and wait for interrupts.

9.30 Repeat Problem 9.29 using the 8086 NMI interrupt.

9.31 An 8086/2732/6116/8255-based microcomputer is required to drive the LEDs connected to bit 0 of ports A and B based on the input conditions set by switches connected to bit 1 of ports A and B. The I/O conditions are as follows:

- If the input at bit 1 of port A is HIGH and the input at bit 1 of port B is low, then the LED at port A will be ON and the LED at port B will be OFF.
- If the input at bit 1 of port A is LOW and the input at bit 1 of port B is HIGH, then the LED at port A will be OFF and the LED at port B will be ON.
- If the inputs at both ports A and B are the same (either both HIGH or both LOW), then both LEDs at ports A and B will be ON.

Write an 8086 assembly language program to accomplish this. Do not use any instructions involving the parity flag.

9.32 An 8086/2732/6116/8255-based microcomputer is required to test a NAND gate. Figure P9.32 shows the I/O hardware needed to test the NAND gate. The microcomputer is to be programmed to generate the various logic conditions for the NAND inputs, input the NAND output, and turn the LED ON connected to bit 3 of port A if the NAND gate chip is found to be faulty. Otherwise, turn the LED ON connected to bit 4 of port A. Write an 8086 assembly language program to accomplish this.

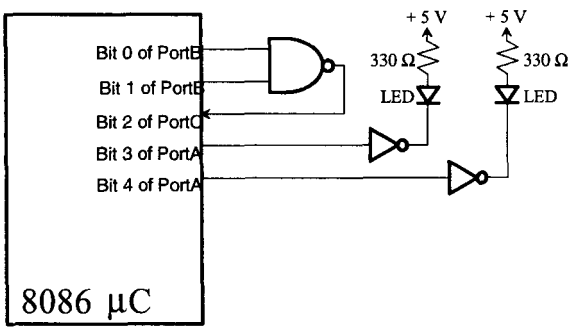


FIGURE P9.32 (Assume both LEDs are OFF initially)

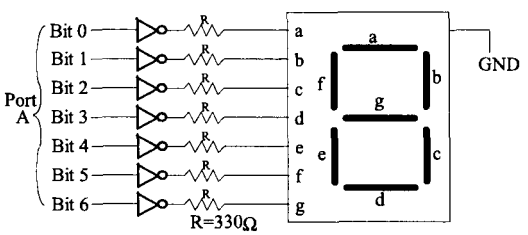


FIGURE P9.33

9.33 An 8086/2732/6116/8255 microcomputer is required to add two 3-bit numbers in AL and BL and output the sum (not to exceed 9) to a common cathode seven-segment display connected to port A as shown in Figure P9.33.Write an 8086 assembly language program to accomplish this by using a look-up table. Do not use XLAT instruction.

9.34 Write an 8086 assembly language program to turn an LED OFF connected to bit 2 of port A of an 8086/2732/6116/8255 microcomputer and then turn it on after delay of 15 s. Assume the LED is ON initially.

9.35 What are the factors to be considered for interfacing a hex keyboard to a microcomputer?

9.36 An 8086/2732/6116/8255 microcomputer is required to input a number from 0 to 9 from an ASCII keyboard interfaced to it and output to an EBCDIC printer. Assume that the keyboard is connected to port A and the printer is connected to port B. Write an 8086 assembly language to accomplish this. Use XLAT instruction.

9.37 Will the circuit shown in Figure P9.37 work? If so, determine the I/O map in hex. If not, justify briefly, modify the circuit and determine the I/O map in hex. Use only the pins and signals provided. Assume all don't cares to be zeros. Note that I/O map includes the addresses for port A, port B, port C, and the control register. Using the logical port addresses, write an instruction sequence to configure port A as input and port B as output.

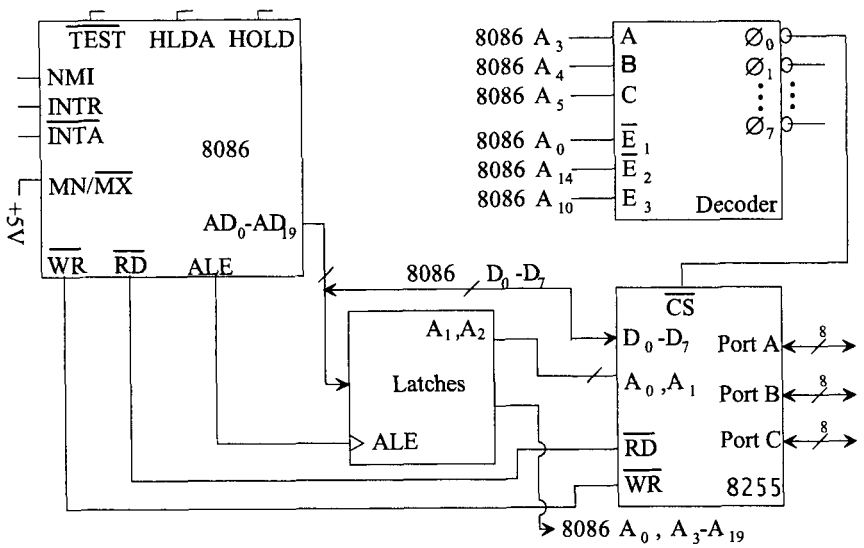


FIGURE P9.37