MODULE – 1

THE x86 MICROPROCESSOR & ALP

THE x86 MICROPROCESSOR

BRIEF HISTORY OF THE x86 FAMILY:

A study of history is not essential to understand the microprocessor, but it provides a historical perspective of the fast-paced evolution of the computer.

Evolution from 8080/8085 to 8086:

In 1978, Intel Corporation introduced a 16-bit microprocessor called the 8086. This processor was a major improvement over the previous generation 8080/8085 series Intel microprocessors in several ways:

1. The 8080/8085 was an 8-bit system (meaning that, the microprocessor could work on only 8 bits of data at a time; data larger than 8 bits need to be broken into 8-bit pieces to be processed by the CPU). In contrast, the 8086 is a 16-bit microprocessor.
2. The 8086's capacity of 1 mega-byte of memory exceeded the 8080/8085's capability of handling a maximum of 64K bytes of memory.
3. The 8086 was a pipelined processor, as opposed to the non-pipelined 8080/8085 (In a system with pipelining, the data and address buses are busy transferring data, while the CPU is processing information; thereby increasing the effective processing power of the micro-processor).

Table: Evolution of Intel microprocessors up to the 8088

Product	8008	8080	8085	8086	8088
Year introduced	1972	1974	1976	1978	1979
Technology	PMOS	NMOS	NMOS	NMOS	NMOS
Number of pins	18	40	40	40	40
Number of transistors	3000	4500	6500	29,000	29,000
Number of instructions	66	111	113	133	133
Physical memory	16KB	64KB	64KB	1MB	1MB
Virtual memory	None	None	None	None	None
Internal data bus	8	8	8	16	16
External data bus	8	8	8	16	8
Address bus	8	16	16	20	20
Data types	8	8	8	8/16	8/16

Evolution from 8086 to 8088:

The 8086 is a microprocessor with a 16-bit data bus internally and externally, meaning that all registers are 16 bits wide and there is a 16-bit data bus to transfer data in and out of the CPU.

Although the introduction of the 8086 marked a great advancement over the previous generation of microprocessors, there was still some resistance in using the 16-bit external data bus:

- ✓ At that time, all peripherals were designed around an 8-bit microprocessor
- ✓ In addition, a printed circuit board with a 16-bit data bus was much more expensive.

Therefore, Intel came out with the 8088 version. It is identical to the 8086 as far as programming is concerned, but externally it has an 8-bit data bus instead of a 16-bit bus. It has the same memory capacity, 1MB.

Success of the 8086:

In 1981, Intel's fortunes changed forever when IBM picked up the 8088 as their microprocessor of choice in designing the IBM PC. The 8088-based IBM PC was an enormous success, because IBM and Microsoft made it an open system (meaning that, all documentation and specifications of the hardware and software of the PC were made public). This made it possible for many other vendors to clone the hardware successfully and thus generated a major growth in both hardware and software designs based on the IBM PC. This is in contrast with the Apple computer, which was a closed system (blocking any attempt at cloning by other manufacturers, both domestically and overseas).

Other Microprocessors: the 80286, 80386, and 80486:

Intel introduced the **80286** in 1982. Its features included –

- ✓ 16-bit internal and external data buses.
- ✓ 24 address lines, which give 16 mega-bytes of memory (2^{24} = 16M bytes).
- ✓ Virtual memory – a way or fooling the microprocessor into thinking that it has access to an almost unlimited amount of memory by swapping data between disk storage and RAM.
- ✓ The 80286 can operate in one of two modes: *real mode* and *protected mode*. Real mode is simply a faster 8088/8086 with the same maximum of 1M bytes of memory. Protected mode allows for 16M bytes of memory but is also capable of protecting the operating system and programs from accidental or deliberate destruction by a user, a feature that is absent in the single-user 8088/8086. IBM picked up the 80286 for the design of the IBM PC AT.

With users demanding even more powerful systems, in 1985 Intel introduced the **80386** (sometimes called 80386DX):

- ✓ Internally and externally a 32-bit microprocessor.
- ✓ 32-bit address bus; capable of handling physical memory of up to 4 giga-bytes (2^{32} = 4G bytes).
- ✓ Virtual memory was increased to 64 terabytes (2^{46} = 64T bytes).
- o All microprocessors discussed so far were general-purpose microprocessors and could not handle mathematical calculations rapidly. For this reason, Intel introduced numeric data processing chips, called math-coprocessors, such as the 8087, 80287, and 80387.

- Later Intel introduced the 386SX, which is internally identical to the 80386 but has a 16-bit external data bus and a 24-bit address bus, which gives a capacity of 16M bytes (2^{24} = 16M bytes) of memory. This makes the 386SX system much cheaper.
- With the introduction of the 80486 in 1989, Intel put a greatly enhanced version of the 80386 and the math-coprocessor on a single chip plus additional features such as cache memory. Cache memory is static RAM with a very fast access time. Note that, all programs written for the 8088/86 will run on 286, 386, and 486 computers.

In 1992, Intel released the newest x86 microprocessor – the **Intel Pentium**:

- ✓ By using submicron fabrication technology, Intel designers were able to utilize more than 3 million transistors on the Pentium chip.
- ✓ The Pentium had speeds of 60 and 66 MHz (twice that of 80486 and over 300 times faster than that of the original 8088).
- ✓ Separate 8K cache memory for code and data.
- ✓ 64-bit external data bus with 32-bit register and 32-bit address bus capable of addressing 4GB of memory.
- ✓ Improved floating-point processor.
- ✓ Pentium is packaged in a 273-pin PGA chip.
- ✓ It uses BICMOS technology, which combines the speed of bipolar transistors with the power efficiency of CMOS technology.

Table: Evolution of Intel's Microprocessors (from the 8086 to the Pentium Pro)

Product	**8086**	**80286**	**80386**	**80486**	**Pentium**	**Pentium Pro**
Year introduced	1978	1982	1985	1989	1993	1995
Technology	NMOS	NMOS	CMOS	CMOS	BICMOS	BICMOS
Clock rate (MHz)	3 – 10	10 – 16	16 – 33	25 – 33	60, 66	150
Number of pins	40	68	132	168	273	387
Number of transistors	29,000	134,000	275,000	1.2 million	3.1 million	5.5 million
Physical memory	1MB	16MB	4GB	4GB	4GB	64GB
Virtual memory	None	1GB	64TB	64TB	64TB	64TB
Internal data bus	16	16	32	32	32	32
External data bus	16	16	32	32	64	64
Address bus	20	24	32	32	32	36
Data types	8/16	8/16	8/16/32	8/16/32	8/16/32	8/16/32

In 1995, Intel introduced the **Pentium Pro**, the sixth generation of the x86 family.

- ✓ Pentium Pro is an enhanced version of Pentium that uses 5.5 million transistors.
- ✓ It was designed to be used for 32-bit servers and workstations.

- In 1997, Intel introduced its Pentium II processor. This 7.5-million-transistor processor 'featured MMX (Multi-Media extension) technology incorporated into the CPU. MMX allows for fast graphics and audio processing.
- In 1998 the Pentium II Xcon processor was released. Its primary market is for servers and workstations.
- In 1999 the Celeron was released. Its lower cost and good performance make it ideal for PCs used to meet educational and home business needs.
- In 1999, Intel released the Pentium III. This 9.5-million-transistor processor includes 70 new instructions called SIMD that enhance video and audio performance in such areas as 3-D imaging, and streaming audio that have become common features of on-line computing. In 1999, Intel also introduced the Pentium III Xeon processor, designed more for servers and business workstations with multiprocessor configurations.

Table: Evolution of Intel's Microprocessors (from the Pentium II to Itanium)

Product	Pentium II	Pentium III	Pentium 4	Itanium II
Year introduced	1997	1999	2000	2002
Technology	BICMOS	BICMOS	BICMOS	BICMOS
Number of transistors	7.5 million	9.5 million	42 million	220 million
Cache size	512K	512K	512K	3MB
Physical memory	64GB	64GB	64GB	64GB
Virtual memory	64TB	64TB	64TB	64TB
Internal data bus	32	32	32	64
External data bus	64	64	64	64
Address bus	36	36	36	64
Data types	8/16/32	8/16/32	8/16/32	8/16/32/64

- The **Pentium 4**, which debuted late in 1999, had the speeds of 1.4 to 1.5 GHz. The Pentium 4 represents the first completely new architecture since the development of the Pentium Pro. The new 32-bit architecture, called NetBurst, is designed for heavy multimedia processing such as video, music, and graphic file manipulation on the Internet. The system bus operates at 400 MHz. In addition, new cache and pipelining technology and an expansion of the multimedia instruction set are designed to make the P4 a high- end media processing microprocessor.
- Intel has selected Itanium as the new brand name for the first product in its **64-bit** family of processors, formerly called Merced. The evolution of microprocessors is increasingly influenced by the evolution of the Internet. The Itanium architecture is designed to meet

Internet-driven needs for powerful servers and high-performance work-stations. The Itanium will have the ability to execute many instructions simultaneously plus extremely large memory capabilities.

INSIDE THE 8088/86:

The following Fig shows the internal block diagram of the 8088/86 CPU.

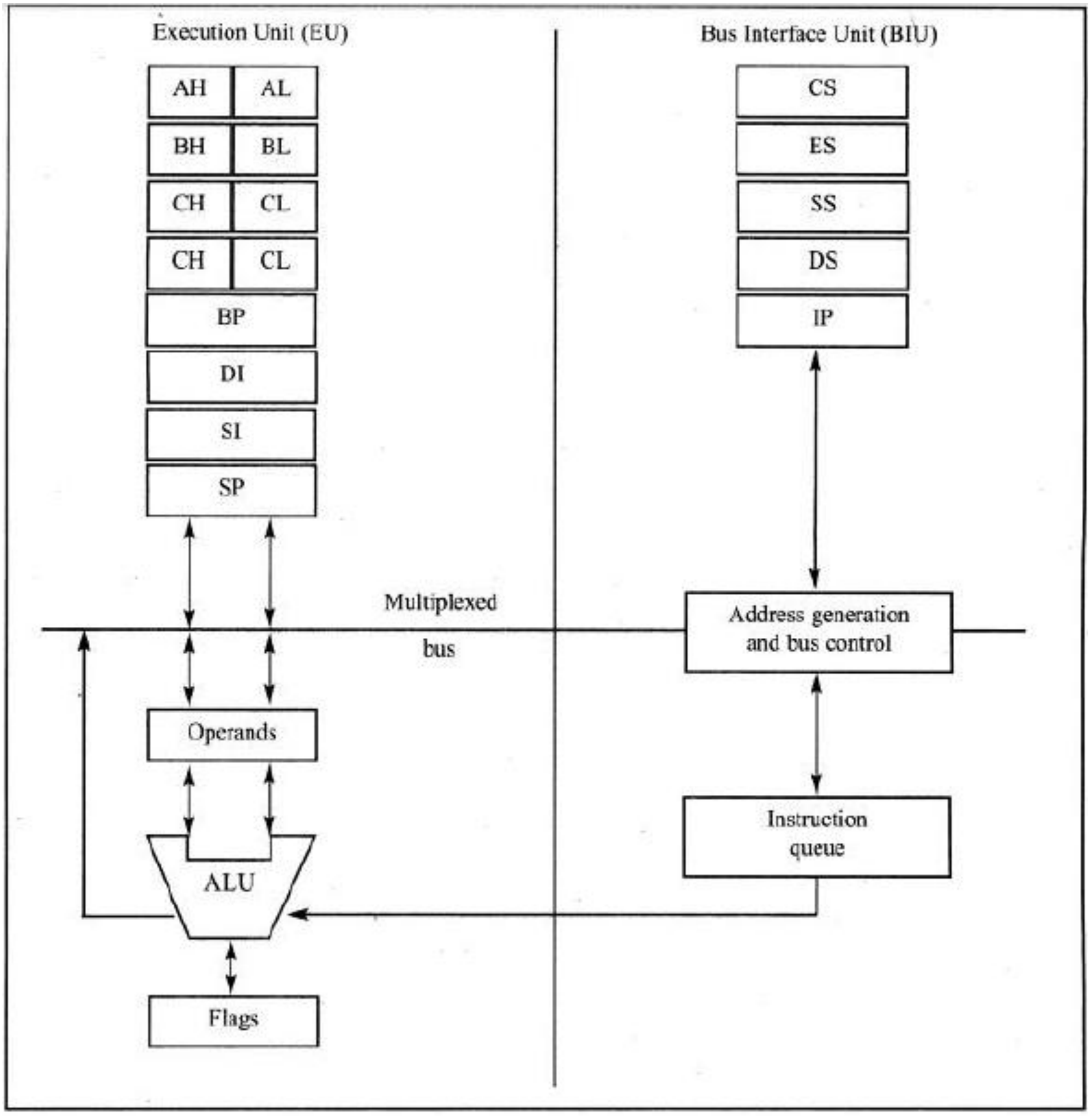


Fig: Internal Block Diagram of the 8088/86 CPU

Pipelining:

There are two ways to make the CPU process information faster:

1. Increase the working frequency – The designers can make the CPU work faster by increasing the frequency under which it runs. But, it is technology dependent, meaning that the designer must use whatever technology is available at the time, with consideration for cost. The technology and materials used in making ICs (integrated circuits) determine the working frequency, power consumption and the number of transistors packed into a single-chip microprocessor.
2. Change the internal architecture of the CPU – The processing power of the CPU can be altered by changing the internal working of the CPU. (In 8085, the CPU had to fetch an instruction from memory, then execute it and then fetch again, execute it, and so on; i.e., 8085 CPU could either fetch or execute at a given time).

The idea of pipelining is to allow the CPU to fetch and execute at the same time as shown in following Fig.

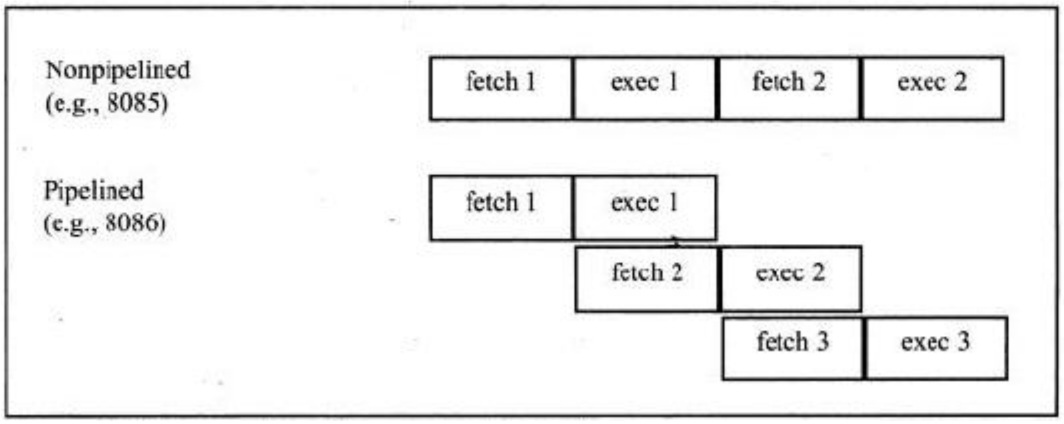


Fig: Pipelined vs. Non-pipelined Execution

Intel implemented the concept of pipelining in the 8088/86 by splitting the internal structure of the microprocessor into two sections:

- The execution unit (EU)
- The bus interface unit (BIU) – These two sections work simultaneously.
  - ✓ The BIU accesses memory and peripherals while the EU executes instructions previously fetched.
  - ✓ This works only if the BIU keeps ahead of the EU; thus the BIU of the 8088/86 has a buffer, or queue. The buffer is 4 bytes long in the 8088 and 6 bytes in the 8086. If any instruction takes too long to execute, the queue is filled to its maximum capacity and the buses will sit idle.
  - ✓ The BIU fetches a new instruction whenever the queue has room for 2 bytes in the 6-byte 8086 queue and for 1 byte in the 4-byte 8088 queue. In some circumstances, the microprocessor must flush out the queue.

For example, when a jump instruction is executed, the BIU starts to fetch information from the new location in memory and information in the queue that was fetched previously is discarded. In this situation the EU must wait until the BIU fetches the new instruction. This is referred to in computer science terminology as a *branch penalty*. In a pipelined CPU, this means that too much jumping around reduces the efficiency of a program.

- ✓ *Pipelining* in the 8088/86 has two stages, *fetch* and *execute*, but in more powerful computers, pipelining can have many stages. The concept of pipelining combined with an increased number of data bus pins has, in recent years, led to the design of very powerful microprocessors.

Registers:

In the CPU, registers are used to store information temporarily. Information could ne one or two bytes of data to be processed or the address of the data. The registers of 8088/86 fall into six categories; as given in the following Table.

Table: Register of 8088/86/286 by Category

Category	**Bits**	**Register Names**
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, VH, CL, DH, DL
Pointer	16	SP (Stack Pointer), BP (Base Pointer)
Index	16	SI (Source Index), DI (Destination Index)
Segment	16	CS (Code Segment), DS (Data Segment), SS (Stack Segment), ES (Extra Segment)
Instruction	16	IP (Instruction Pointer)
Flag	16	FR (Flag Register)

The general-purpose registers in 8088/86 can be accessed as either 16-bit or 8-bit registers. All other registers can be accessed only as the full 16 bits. In 8088/86, data types are either 8 or 16 bits. To access 12-bit data, a 16-bit register must be used with the highest 4 bits set to 0.

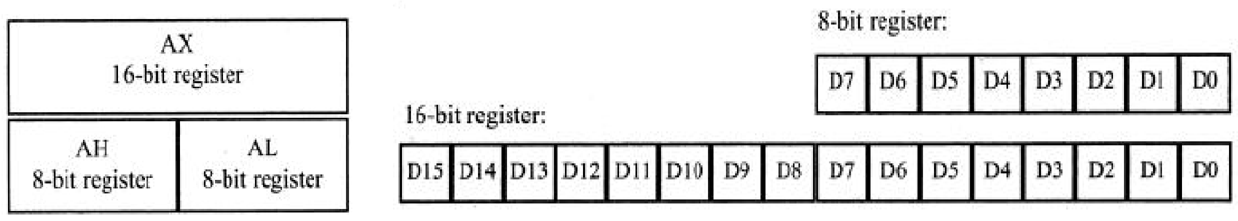


Fig: Structure of General-Purpose Register & Numbering Bits of a Register

Different registers in the 8088/86 are used for different functions. Some instructions use only specific registers to perform their tasks. The first letter of each general-purpose register indicates its use:

- ✓ AX is used for the accumulator

- ✓ BX as a base addressing register
- ✓ CX as a counter in loop operations
- ✓ DX to point to data in I/O operations.

INTRODUCTION TO ASSEMBLY PROGRAMMING:

- The CPU can work only in binary; it can do so at very high speeds. But, it is quite tedious and slow for humans to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called ***machine language***.
- Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, ***Assembly languages*** were developed, which provided mnemonics for the machine code instructions, plus other features that made programming faster and less prone to error.
- The term ***mnemonic*** is typically used in computer science and engineering literature to refer to codes and abbreviations that are relatively easy to remember.
- Assembly language programs (ALPs) must be translated into machine code by a program called an ***assembler***.
- Assembly language is referred to as a ***low-level language*** because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know the number of registers and their size, as well as other details of the CPU.

- Today, one can use many different programming languages, such as C/C++, BASIC, C#, and numerous others. These languages are called ***high-level languages***; because the programmer does not have to be concerned with the internal details of the CPU.
- An assembler is used to translate an Assembly language program into machine code (sometimes called *object code);* high-level languages are translated into machine code by a program called a ***compiler***. For instance, to write a program in C, one must use a C compiler to translate the program into machine language.
- There are numerous assemblers available for translating x86 Assembly language programs into machine code. Most commonly used assemblers, MASM / TASM.

Assembly Language Programming:

An *Assembly language program* consists of –

- ✓ A series of lines of Assembly language instructions –
  - An Assembly language instruction consists of a mnemonic
  - Optionally followed by one or two operands.

- ✓ The *operands* are the data items being manipulated, and the *mnemonics* are the commands to the CPU, telling it what to do with those items.

E.g.:

Opcode (Mnemonic)

Source operand (register Relative addressing)

AGAIN: ADD AX, COUNT [BX] ; ADD ELEMENT OF COUNT TO AX.

Label—provides a Means of branching To this instruction

Destination operand (register addressing)

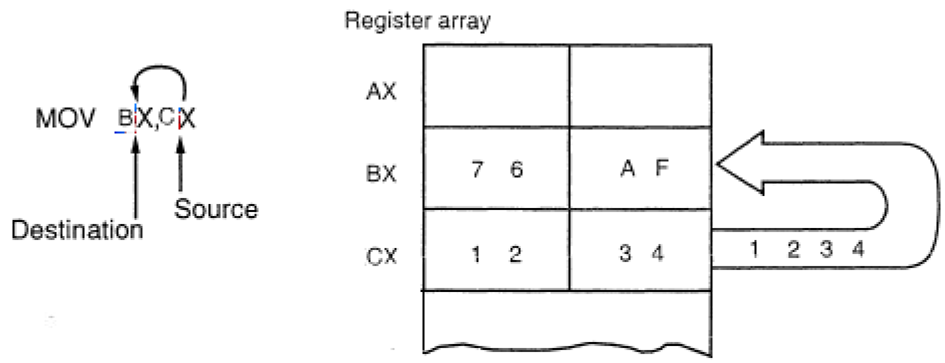
Comment

MOV Instruction:

The MOV instruction copies data from one location to another. The format is –

```
MOV  destination,source ;copy source operand to destination
```

The Following Figure shows the operation of the MOV BX, CX instruction.



The MOV instruction does not affect the source operand. The following program first loads CL with value 55H, then moves this value around to various registers inside the CPU.

```
MOV  CL,55H ;move 55H into register CL
MOV  DL,CL ;copy the contents of CL into DL (now DL=CL=55H)
MOV  AH,DL ;copy the contents of DL into AH (now AH=DL=55H)
MOV  AL,AH ;copy the contents of AH into AL (now AL=AH=55H)
MOV  BH,CL ;copy the contents of CL into BH (now BH=CL=55H)
MOV  CH,BH ;copy the contents of BH into CH (now CH=BH=55H)
```

The use of 16-bit registers is demonstrated below:

```
MOV   CX,468FH   ;move 468FH into CX (now CH=46,CL=8F)
MOV   AX,CX     ;copy contents of CX to AX (now AX=CX=468FH)
MOV   DX,AX     ;copy contents of AX to DX (now DX=AX=468FH)
MOV   BX,DX     ;copy contents of DX to BX (now BX=DX=468FH)
MOV   DI,BX     ;now DI=BX=468FH
MOV   SI,DI     ;now SI=DI=468FH
MOV   DS,SI     ;now DS=SI=468FH
MOV   BP,DI     ;now BP=DI=468FH
```

In 8086 CPU, data can be moved among all the registers (except the flag register) as long as the source and destination registers match in size.

```
MOV   AX,58FCH   ;move 58FCH into AX   (LEGAL)
MOV   DX,6678H   ;move 6678H into DX   (LEGAL)
MOV   SI,924BH   ;move 924B  into SI   (LEGAL)
MOV   BP,2459H   ;move 2459H into BP   (LEGAL)
MOV   DS,2341H   ;move 2341H into DS   (ILLEGAL)
MOV   CX,8876H   ;move 8876H into CX   (LEGAL)
MOV   CS,3F47H   ;move 3F47H into CS   (ILLEGAL)
MOV   BH,99H     ;move 99H into BH     (LEGAL)
```

Note the following three points with regarding MOV instruction:

1. Values cannot be loaded directly into any segment register (CS, DS, SS, and ES). To load a value into a segment register, first load it to a non-segment register and then move it to the segment register, as shown below.

```
MOV   AX,2345H   ;load 2345H into AX
MOV   DS,AX     ;then load the value of AX into DS

MOV   DI,1400H   ;load 1400H into DI
MOV   ES,DI     ;then move it into ES, now ES=DI=1400
```

2. If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros. E.g.: MOV BX, 5   ; result will be BX = 0005, i.e., BH = 00 and BL = 05.
3. Moving a value that is too large into a register will cause an error.

```
MOV   BL,7F2H      ;ILLEGAL: 7F2H is larger than 8 bits
MOV   AX,2FE456H   ;ILLEGAL: the value is larger than AX
```

ADD Instruction:

The ADD instruction has the following format –

```
ADD   destination,source   ;ADD the source operand to the destination
```

The ADD instruction tells the CPU to add the source and the destination operands and put the result in the destination.

```
MOV   AL,25H   ;move 25 into AL  | MOV   DH,25H   ;move 25 into DH
MOV   BL,34H   ;move 34 into BL  | MOV   CL,34H   ;move 34 into CL
ADD   AL,BL    ;AL = AL + BL     | ADD  DH,CL    ;add CL to DH: DH = DH + CL
```

Executing above program results in AL (or DH) = 59H (25H + 34H = 59H) and BL (or CL) = 34H. Notice that, the contents of the source operand do not change.

It is not necessary to move both data items into registers before adding them together.

```
MOV DH,25H   ;load one operand into DH
ADD DH,34H   ;add the second operand to DH
```

Hence, for MOV and ADD instructions, the source operand may be an immediate data – this is called an *immediate operand*. Please note, the destination operand has always been a register.

The largest number that an 8-bit register can hold is FFH. To use numbers larger than FFH (255 decimal), 16-bit registers (such as AX, BX, CX, or DX) must be used.

```
MOV AX,34EH   ;move 34EH into AX
MOV DX,6A5H   ;move 6A5H into DX
ADD  DX,AX    ;add AX to DX: DX = DX + AX
```

```
MOV  CX,34EH  ;load 34EH into CX
ADD CX,6A5H ;add 6A5H to CX (now CX=9F3H)
```

Running the above program(s) give DX (or CX) = 9F3H (34E + 6A5 = 9F3H) and AX = 34EH.

INTRODUCTION TO PROGRAM SEGMENTS:

A typical Assembly language program consists of at least three segments:

1. Code segment – contains the Assembly language instructions that perform the tasks that the program was designed to accomplish.
2. Data segment – is used to store information (data) that needs to be processed by the instructions in the code segment.
3. Stack segment – is used by the CPU to store i information temporarily.

Origin and Definition of the Segment:

A ***segment*** is an area of memory that includes up to 64K bytes and begins on an address evenly divisible by 16 (such an address ends in 0H). In 8085, there was only 64K byte (2^{16} = 16KB) of memory for all code, data, and stack information; in the 8088/86 there can be up to 64K bytes of memory assigned to each category. Within an Assembly language program, these categories are called the *code segment*, *data segment*, and *stack segment*. For this reason, the 8088/86 can only handle a maximum of 64K bytes of code, 64K bytes of data, and 64K bytes of stack at anygiven time, all though it has a range of 1M bytes (2^{20} = 1M bytes) of memory.

Logical Address and Physical Address:

There are three types of addresses mentioned with the 8086:

1. The physical address – is the 20-bit address that is actually put on the address pins of the 8086 microprocessor and decoded by memory interfacing circuitry. This is an actual physical location

in RAM or ROM within the 1M byte memory range. This address can have a range of 00000H – FFFFFH for the 8086, and real mode 286, 386, and 486 CPUs.

2. The offset address – is a location within a 64K byte segment range. Hence, an offset address can range from 0000H – FFFFH.
3. The logical address – consists of a segment value and an offset address.

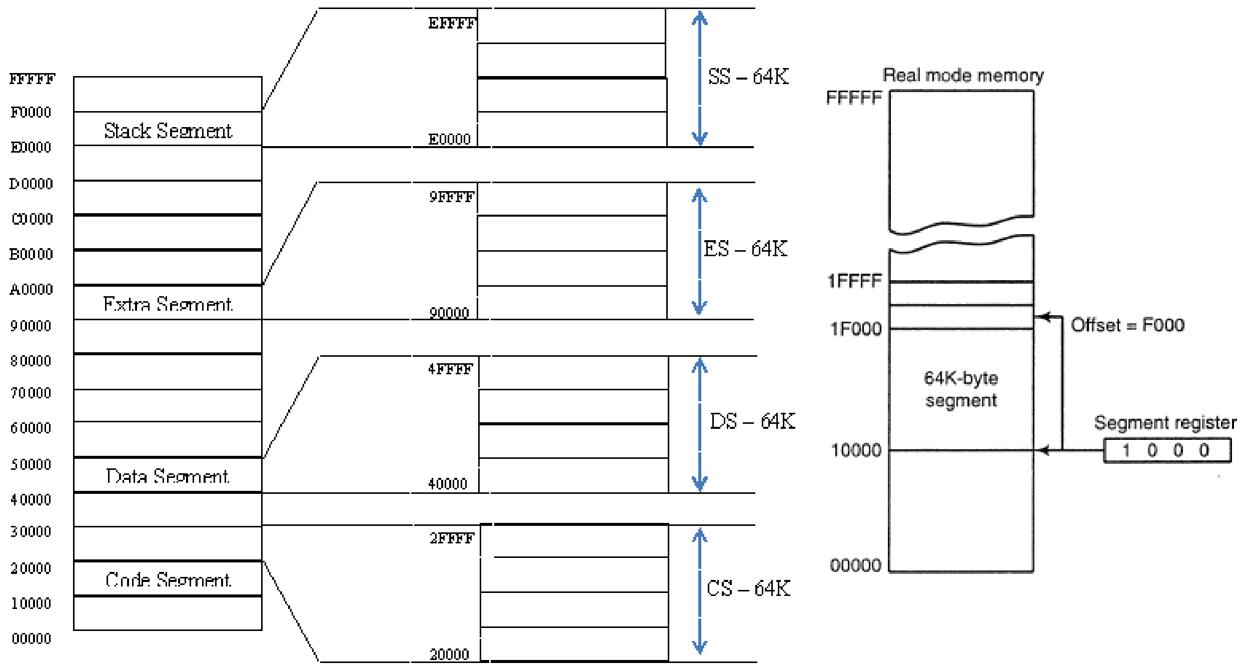
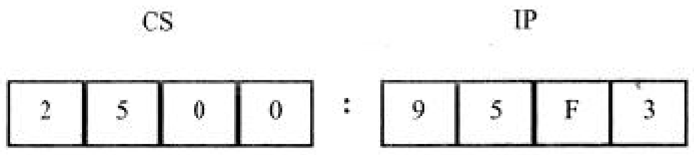


Fig: Illustration of Physical Address, Offset, and Logical Address

Code Segment:

To execute a program, the 8086 fetches the instruction (opcode and operands) from the code segment. The logical address of an instruction always consists of a CS (code segment) and an IP (instruction pointer), shown in the following Fig.



The physical address for the location of the instruction is generated by

- Shifting the CS left by one hex digit and then adding it to the IP. IP contains the offset address. The resulting 20-bit address is called the physical address.
- To clarify this concept; assume values in CS and IP as shown in the above diagram. The offset address is contained in IP; in this case it is 95F3H. The logical address is CS: IP, or 2500: 95F3H. Then the physical address will be 25000 +95F3 = 2E5F3H.

The physical address of an instruction can be calculated as follows:

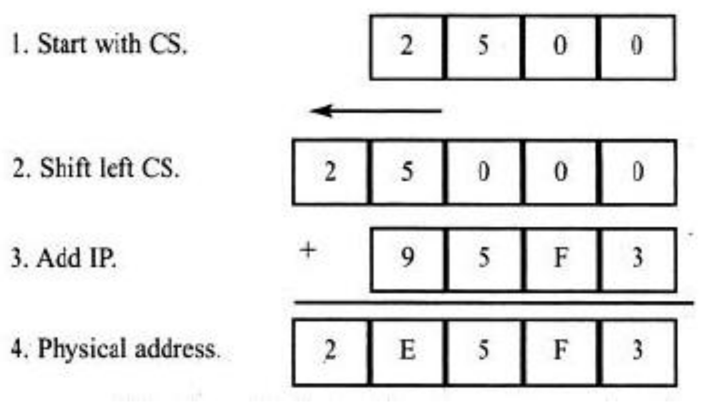
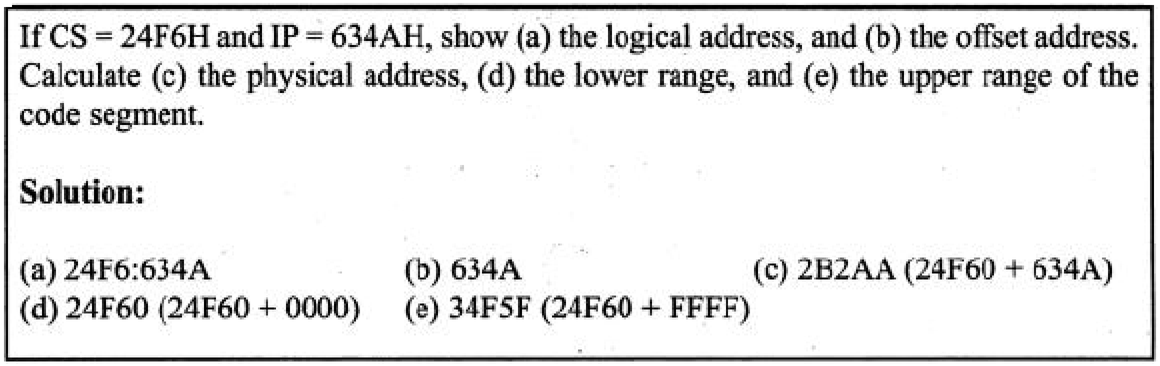


Fig: Calculation of Physical Address

The microprocessor will retrieve the instruction from memory locations starting at 2E5F3. Since IP can have a minimum value of 0000H and a maximum of FFFFH; the logical address range in this example is 2500:0000 to 2500: FFFF. This means that the lowest memory location of the code segment will be 25000H (25000+0000) and the highest memory location will be 34FFFH (25000+FFFF).

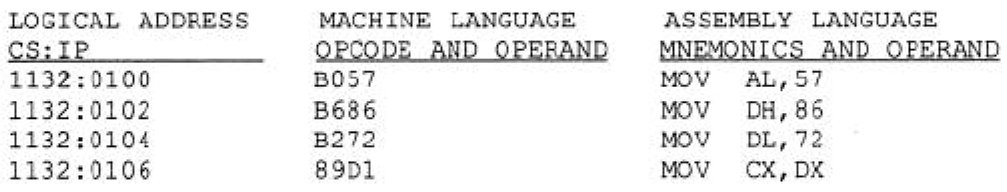

If CS = 24F6H and IP = 634AH, show (a) the logical address, and (b) the offset address. Calculate (c) the physical address, (d) the lower range, and (e) the upper range of the code segment.

Solution:

(a) 24F6:634A (b) 634A (c) 2B2AA (24F60 + 634A)
(d) 24F60 (24F60 + 0000) (e) 34F5F (24F60 + FFFF)

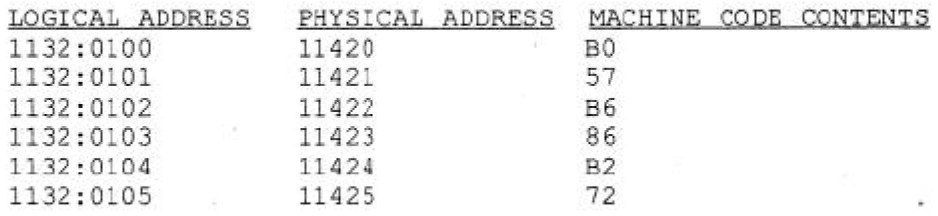
Logical Address vs. Physical Address in the Code Segment:

In the code segment, CS and IP hold the logical address of the instructions to be executed. The following Assembly language instructions have been assembled (translated into machine code) and stored in memory. The three columns show the logical address of CS: IP, the machine code stored at that address, and the corresponding Assembly language code.


```
LOGICAL ADDRESS     MACHINE LANGUAGE       ASSEMBLY LANGUAGE
CS:IP               OPCODE AND OPERAND     MNEMONICS AND OPERAND
1132:0100           B057                   MOV  AL,57
1132:0102           B686                   MOV  DH,86
1132:0104           B272                   MOV  DL,72
1132:0106           89D1                   MOV  CX,DX
```

The program above shows that the byte at address 1132:0100 contains B0, which is the opcode for moving a value into register AL, and address 1132:0I101I contains the operand (in this case 57) to be moved to AL. Therefore, the instruction "MOVAL, 57" has a machine code of B057, where B0 is the opcode and 57 is the operand.

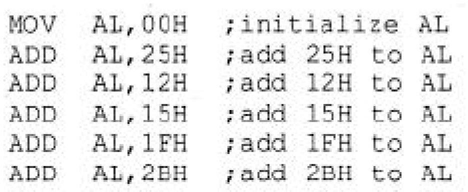
The following are the physical addresses and contents of each location for the above program.



LOGICAL ADDRESS	PHYSICAL ADDRESS	MACHINE CODE CONTENTS
1132:0100	11420	B0
1132:0101	11421	57
1132:0102	11422	B6
1132:0103	11423	86
1132:0104	11424	B2
1132:0105	11425	72

Data Segment:

Assume that a program is being written to add 5 bytes of data, such as 25H, 12H, 15H, IFH, and 2BH. One way to add them is as follows:

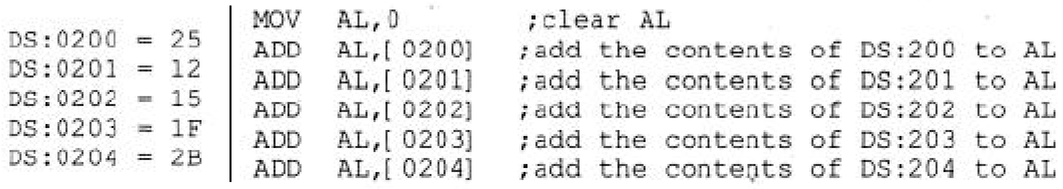


```
MOV   AL,00H   ;initialize AL
ADD   AL,25H   ;add 25H to AL
ADD   AL,12H   ;add 12H to AL
ADD   AL,15H   ;add 15H to AL
ADD   AL,1FH   ;add 1FH to AL
ADD   AL,2BH   ;add 2BH to AL
```

In the program above, the data and code are mixed together. The problem with writing the program this way is that, if the data changes, the code must be searched for every place the data is included, and the data retyped.

The idea to overcome the problem is to set aside an area of memory is strictly for data. In x86 microprocessors, the area of memory set aside for data is called the *data segment.* Just as the code segment is associated with CS and IP as its segment register and offset, the data segment uses register DS and an offset value.

The following demonstrates how data can be stored in the data segment and the program rewritten so that it can be used for any set of data. Assume that the offset for the data segment begins at 200H.



```
                 | MOV   AL,0         ;clear AL
DS:0200 = 25     | ADD   AL,[0200]    ;add the contents of DS:200 to AL
DS:0201 = 12     | ADD   AL,[0201]    ;add the contents of DS:201 to AL
DS:0202 = 15     | ADD   AL,[0202]    ;add the contents of DS:202 to AL
DS:0203 = 1F     | ADD   AL,[0203]    ;add the contents of DS:203 to AL
DS:0204 = 2B     | ADD   AL,[0204]    ;add the contents of DS:204 to AL
```

NOTE:

1. The offset address is enclosed in brackets. The brackets indicate that the operand represents the address of the data and not the data itself. If the brackets were not included, as in "MOV AL, 0200", the CPU would attempt to move 200 into AL instead of the contents of offset address 200.
2. DEBUG assumes that all numbers are in hex (no "H" suffix is required), whereas MASM/TASM assumes that they are in decimal and the "H" must be included for hex data.

This program will run with any set of data. Changing the data has no effect on the code. Although this program is an improvement over the preceding one, it can be improved even further.

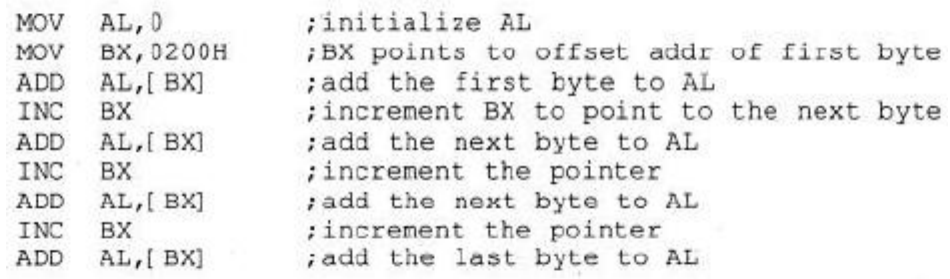
If the data had to be stored at a different offset address (say 450H), the program would have to be rewritten. One way to solve this problem would be to use a register to hold the offset address, and before each ADD, to increment the register to access the next byte.

The 8088/86 allows only the use of registers BX, SI, and DI as offset registers for the data segment In other words, while CS uses only the IP register as an offset, DS uses only BX, DI, and SI to hold the offset address of the data.

Table: Default Segments and Offset Register Pairs

Segment	**Offset**	**Special Purpose**
CS	IP	Instruction address
DS	SI, DI, BX, an 8- or 16-bit number	Data address
SS	SP or BP	Stack address
ES	SI, DI, BX for string instructions	String destination address

The term *pointer* is often used for a register holding an offset address. In the following example, BX is used as a pointer.

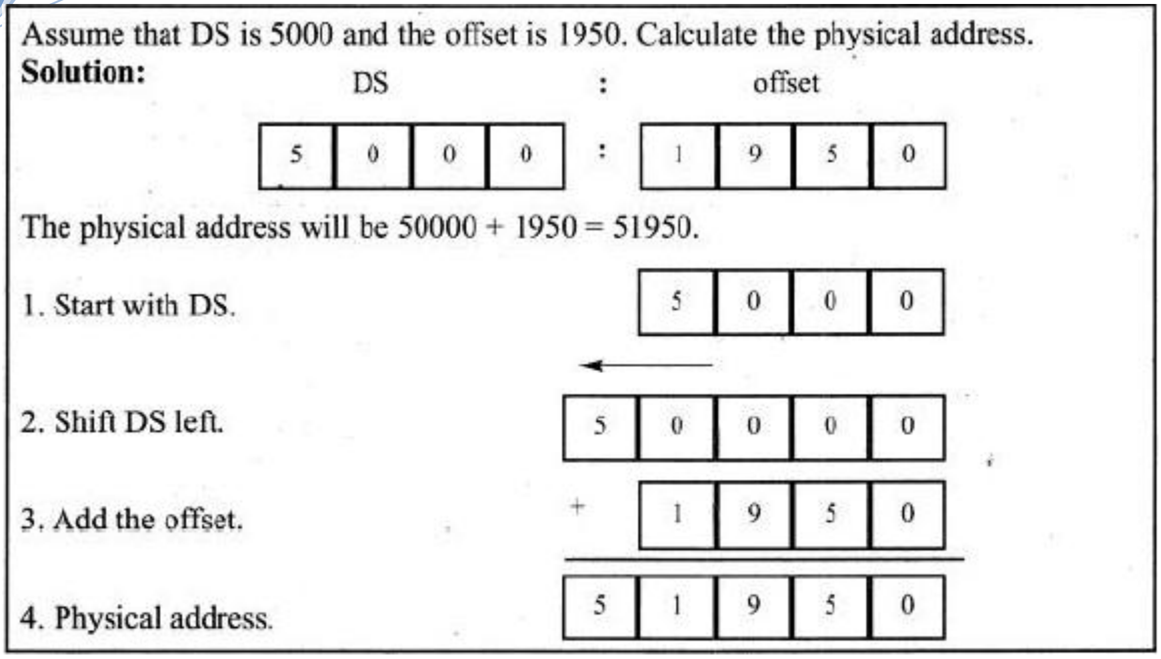


```
MOV   AL,0        ;initialize AL
MOV   BX,0200H    ;BX points to offset addr of first byte
ADD   AL,[ BX]    ;add the first byte to AL
INC   BX          ;increment BX to point to the next byte
ADD   AL,[ BX]    ;add the next byte to AL
INC   BX          ;increment the pointer
ADD   AL,[ BX]    ;add the next byte to AL
INC   BX          ;increment the pointer
ADD   AL,[ BX]    ;add the last byte to AL
```

The INC instruction adds 1 to (increments) its operand. "INC BX" achieves the same result as "ADD BX, 1".

Logical Address and Physical Address in the Data Segment:

The physical address for data is calculated using the same rules as for the code segment. That is, the physical address of data is calculated by shifting DS left one hex digit and adding the offset value, as shown in following Examples.

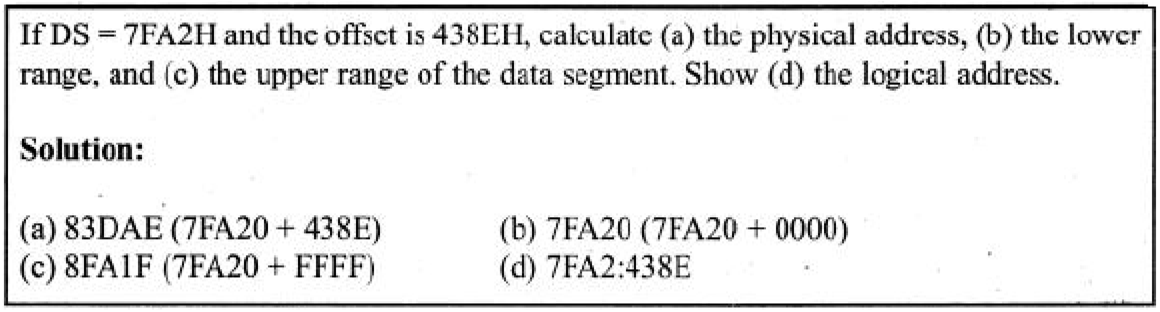


Assume that DS is 5000 and the offset is 1950. Calculate the physical address.

Solution:

The physical address will be 50000 + 1950 = 51950.

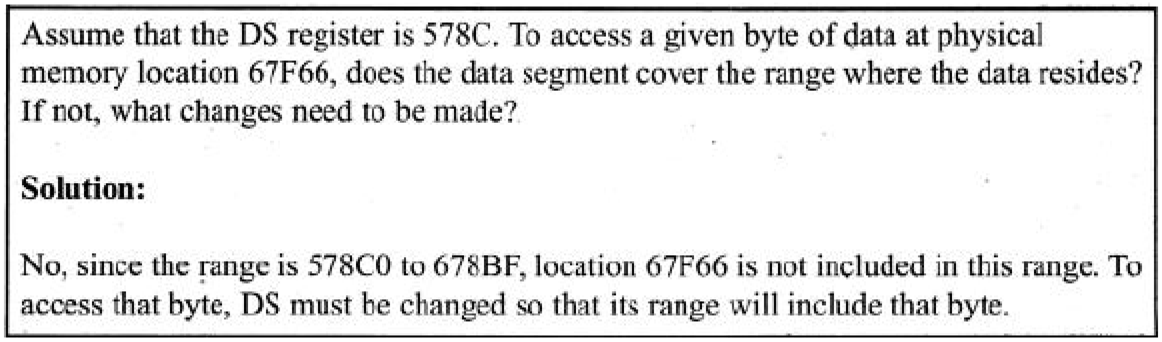
1. Start with DS.
2. Shift DS left.
3. Add the offset.
4. Physical address.



If DS = 7FA2H and the offset is 438EH, calculate (a) the physical address, (b) the lower range, and (c) the upper range of the data segment. Show (d) the logical address.

Solution:

(a) 83DAE (7FA20 + 438E) (b) 7FA20 (7FA20 + 0000)
(c) 8FA1F (7FA20 + FFFF) (d) 7FA2:438E



Assume that the DS register is 578C. To access a given byte of data at physical memory location 67F66, does the data segment cover the range where the data resides? If not, what changes need to be made?

Solution:

No, since the range is 578C0 to 678BF, location 67F66 is not included in this range. To access that byte, DS must be changed so that its range will include that byte.

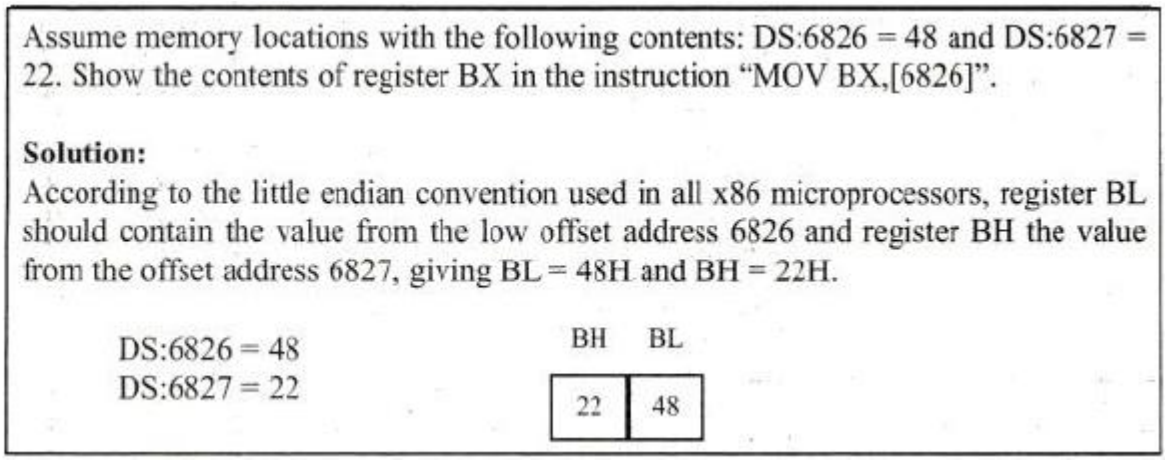
Little Endian Conversion:

Previous examples used 8-bit or 1-byte (16-bits) data. In this case the bytes are stored one after another in memory. The 16-bit data can be used as follows:

```
MOV  AX,35F3H ;load 35F3H into AX
MOV  [1500],AX  ;copy the contents of AX to offset 1500H
```

In this case, the low byte goes to the low memory location and the high byte goes to the high memory location. In the above example, memory location DS: 1500 contains F3H and memory location DS: 1501 contains 35H (DS: 1500 = F3 and DS: 1501 = 35). This is called little endian conversion.

NOTE: In the ***big endian method***, the high byte goes to the low address, where as in the ***little endian method***, the high byte goes to the high address and the low byte goes to the low address. All Intel microprocessors use the little endian conversion.



Assume memory locations with the following contents: DS:6826 = 48 and DS:6827 = 22. Show the contents of register BX in the instruction "MOV BX,[6826]".

Solution:
According to the little endian convention used in all x86 microprocessors, register BL should contain the value from the low offset address 6826 and register BH the value from the offset address 6827, giving BL = 48H and BH = 22H.

DS:6826 = 48
DS:6827 = 22

BH	BL
22	48

Extra Segment (ES):

ES is a segment register used as an extra data segment. Its use is essential for string operations.

Memory map of IBM PC:

The 20-bit address of 8088/86 allows a total of 1M bytes (1024K bytes) of memory space with the address range 00000H – FFFFFH. *Memory map* is the process of allocating the 1M bytes of memory space to various sections of the PC.

Out of 1MB –

- ✓ 640KB from the address 00000H – 9FFFFH were set aside for RAM;
- ✓ the 128KB from A0000H – BFFFFH were allocated for video memory;
- ✓ the remaining 256KB from C0000H – FFFFFH were set aside for ROM.

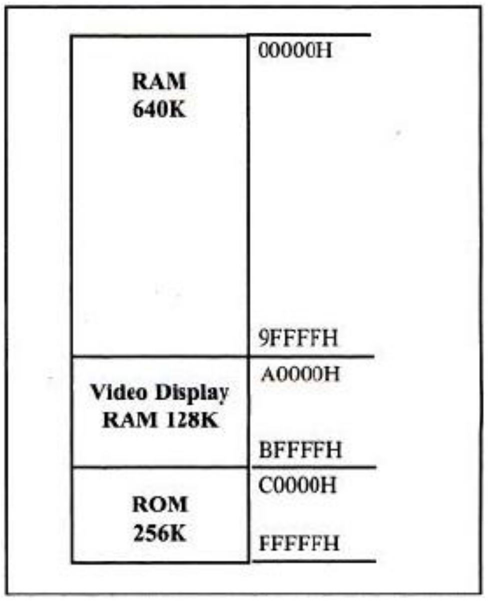


Fig: Memory Allocation in the PC

More about RAM:

In the early 1980s, most PCs came with only 64K to 256K bytes of RAM memory, which was considered more than adequate at the time. Users had to buy memory expansion boards to expand memory up to 640K, if they needed additional memory. The need for expansion depends on the Windows version being used and the memory needs of the application software being run.

The Windows operating system first allocates the available RAM on the PC for its own use and then lets the rest be used for applications such as word processors. The complicated task of managing RAM memory is left to Windows, since the amount of memory used by Windows varies among its various versions and the memory needs of the application packages vary. For this reason we do not assign any values for the CS, D S , and SS registers; since such an assignment means specifying an exact physical address in the range 00000-9FFFFH , and this is beyond the knowledge of the user.

Another reason is that assigning a physical address might work on a given PC but it might not work on a PC with a different OS version and RAM size. In other w ords, the program would not be portable to another PC.

Therefore, memory management is one of the most important functions of the operating system and should be left to Windows.

Video RAM:

From A0000H to BFFFFH is set aside for video. The amount used and the location vary depending on the video board installed on the PC.

More about ROM:

From C0000H to FFFFFH is set aside for ROM. Not all the memory space in this range is used by the PC's ROM. Of this, 256K bytes, only the 64K bytes from location F0000H – FFFFFH are used by BIOS (basic input/output system) ROM.

Some of the remaining space is used by various adapter cards (such as the network card), and the rest is free. In recent years, newer versions of Windows have gained some very powerful memory management capabilities and can put to good use all the unused memory space beyond 640.

The 640KB memory space from 00000 to 9FFFFH is referred to as ***conventional memory***, while the 384K bytes from A0000H to FFFFFH are called the UMB ***(upper memory block)*** in Microsoft literature.

Functions of BIOS ROM:

Since the CPU can only execute programs that are stored in memory, there must be some permanent (nonvolatile) memory to hold the programs, telling the CPU what to do when the power is turned on. This collection of programs held by ROM is referred to as BIOS in the PC literature.

BIOS, which stands for *basic input-output system*, contains programs to test RAM and other components connected to the CPU. It also contains programs that allow Windows to communicate with peripheral devices such as the keyboard, video, printer, and disk.

It is the function of BIOS to test all the devices connected to the PC when the computer is turned on and to report any errors. For example, if the keyboard is disconnected from the PC before the computer is turned on, BIOS will report an error on the screen, indicating that condition.

After testing and setting up the peripherals; BIOS will load Windows from disk into RAM and hand over control of the PC to Windows. Windows always controls the PC once it is loaded.

THE STACK:

What is Stack, and Why is it Needed?

- There must be some place for the CPU to store information safely and temporary. The *stack* is a section of read/write memory (RAM) used by the CPU to store information temporarily.
- The CPU needs this storage area since there are only a limited number of registers.
- The disadvantage of the stack is its access time – since the stack is in RAM, it takes much longer to access compared to the access time of registers. Note that, the registers are inside the CPU and RAM is outside.

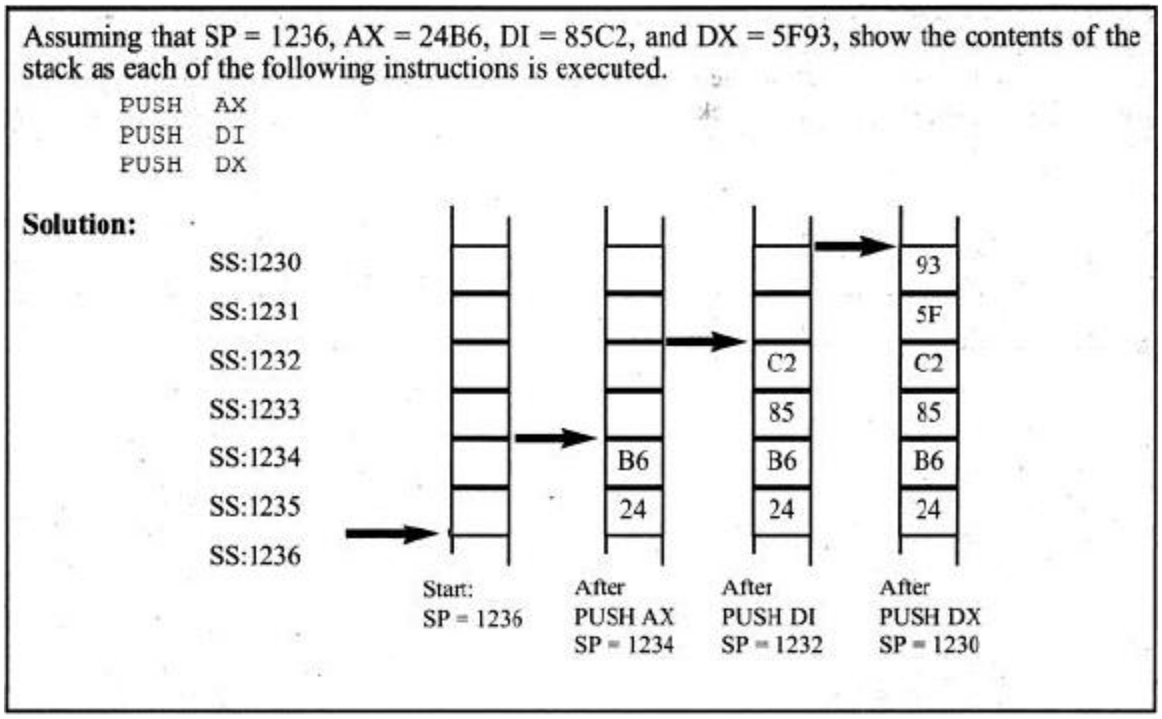
How the Stack are Accessed?

- If the stack is a section of RAM, there must be registers inside the CPU to point to it.
- The two main registers used to access the stack are the SS (stack segment) register and the SP (stack pointer) register.
- These registers must be loaded before any instructions accessing the stack are used.
- Every register inside the x86 (except segment registers and SP) can be stored in the stack and brought back into the CPU from the stack memory.
- The storing of a CPU register in the stack is called a ***push***, and loading the contents of the stack into the CPU register is called a ***pop***. In other words, a register is pushed onto the stack to store its contents and popped off the stack to retrieve it.
- The job of the SP is very critical when push and pop are performed. In the x86, the stack pointer register (SP) points at the current memory location used for the top of the stack and as data is pushed onto the stack it is decremented. It is incremented as data is popped off the stack into the CPU.
- When an instruction pushes or pops a general-purpose register, it must be the entire 16-bit register. In other words, one must code "PUSH AX"; there are no instructions such as "PUSH AL" or "PUSH AH".

- The reason that the SP is decremented after the push is to make sure that the stack is growing downward from upper addresses to lower addresses. This is the opposite of the IP (instruction pointer). As was seen in the preceding section, the IP points to the next instruction to be executed and is incremented as each instruction is executed.

Pushing onto the Stack:

As each PUSH is executed, the contents of the registers are saved on the stack and SP is decremented by 2. For every byte of data saved on stack, SP is decremented.



Assuming that SP = 1236, AX = 24B6, DI = 85C2, and DX = 5F93, show the contents of the stack as each of the following instructions is executed.

```
PUSH   AX
PUSH   DI
PUSH   DX
```

Solution:

Notice, how the data is stored on the stack. In the x86, the lower byte is always stored in the memory location with the lower address.

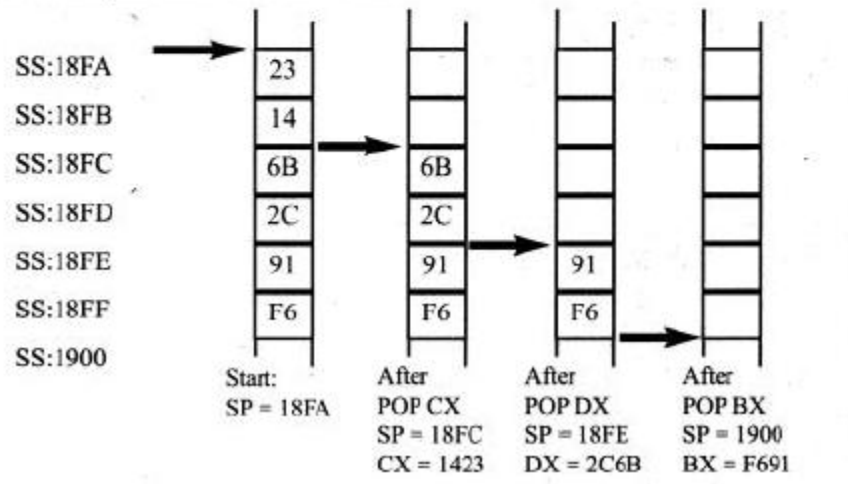
Popping the Stack:

With every POP, the top 2 bytes of the stack are copied to the register specified by the instruction and the stack pointer in incremented twice. Although the data actually remains in memory, it is not accessible since the stack pointer is beyond that point.

Assuming that the stack is as shown below, and SP = 18FA, show the contents of the stack and registers as each of the following instructions is executed:

```
POP   CX
POP   DX
POP   BX
```

Solution:

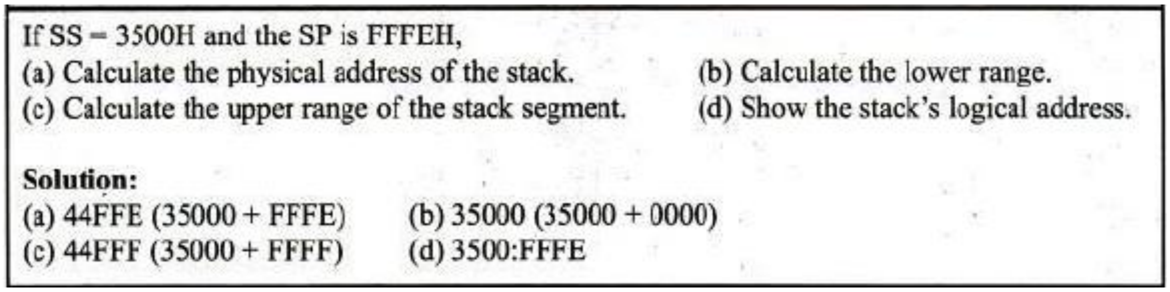


Logical Address vs. Physical Address for the Stack:

- The exact physical location of the stack depends on the value of the SS (stack segment) register and SP (stack pointer). To compute the physical address for stack, shift left SS and then add offset SP register.
- Memory management is the responsibility of the operating system. Hence, the Windows operating system will assign the values for the SP and SS.
- The top of the stack is the last stack location occupied. BP is another register that can be used as an offset into the stack.

Table: Default Segments and Offset Register Pairs

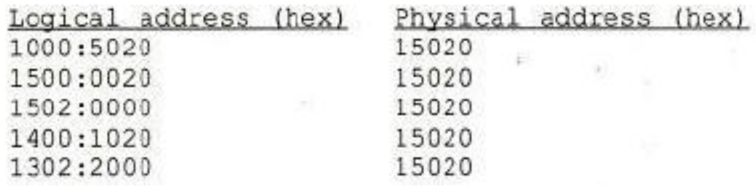
Segment	Offset	Special Purpose
CS	IP	Instruction address
DS	SI, DI, BX, an 8- or 16-bit number	Data address
SS	SP or BP	Stack address
ES	SI, DI, BX for string instructions	String destination address


If SS = 3500H and the SP is FFFEH,
(a) Calculate the physical address of the stack.
(b) Calculate the lower range.
(c) Calculate the upper range of the stack segment.
(d) Show the stack's logical address.

Solution:
(a) 44FFE (35000 + FFFE)
(b) 35000 (35000 + 0000)
(c) 44FFF (35000 + FFFF)
(d) 3500:FFFE

NOTE:

1. A single physical address may belong to many different logical addresses. This shows the dynamic behavior of the segment and offset concept in the 8086 CPU.



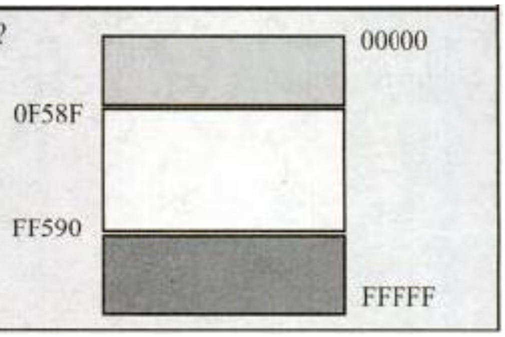
Logical address (hex)	Physical address (hex)
1000:5020	15020
1500:0020	15020
1502:0000	15020
1400:1020	15020
1302:2000	15020

2. When adding the offset to the shifted segment register; if an address beyond the maximum allowed range (FFFFFH) is resulted, then wrap-around will occur.

What is the range of physical addresses if CS = FF59?

Solution:

The low range is FF590 (FF590 + 0000).
The range goes to FFFFF and wraps around,
from 00000 to 0F58F (FF590 + FFFF = 0F58F),
as shown in the illustration.



3. In calculating the physical address, it is possible that two segments can overlap, as illustrated in the following Fig.

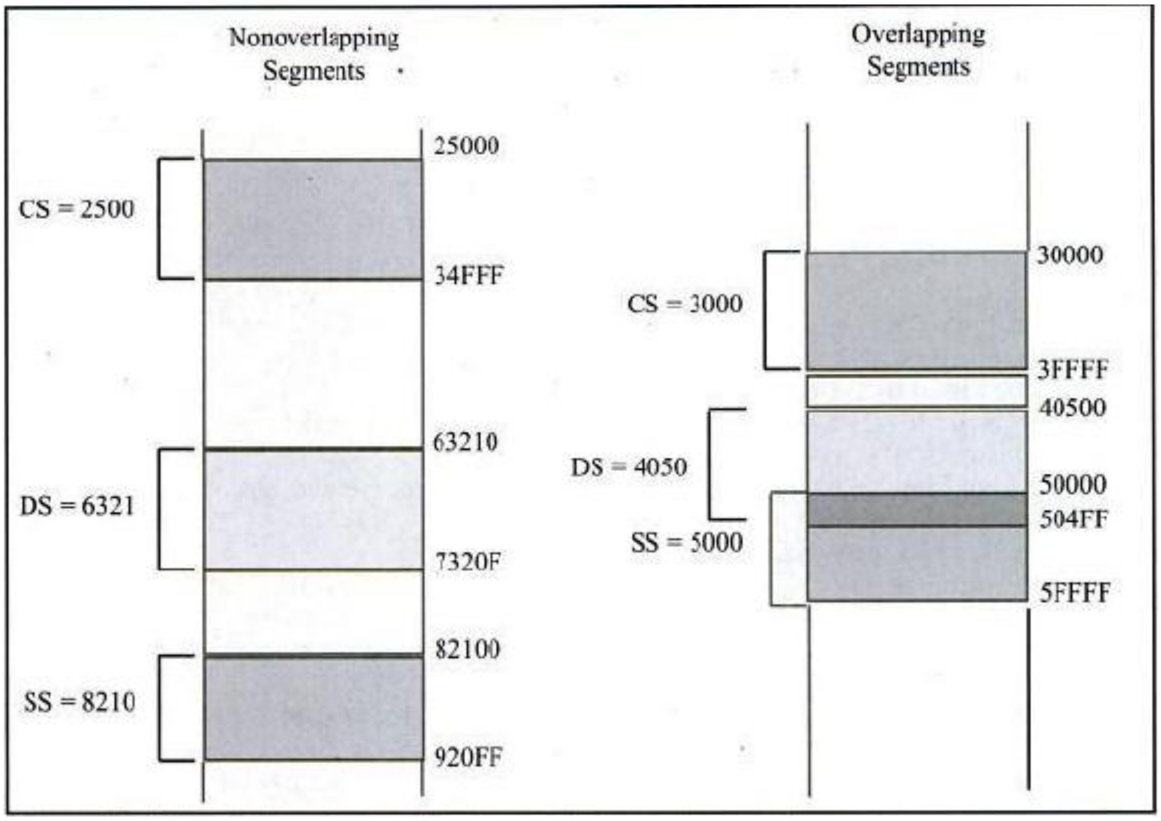
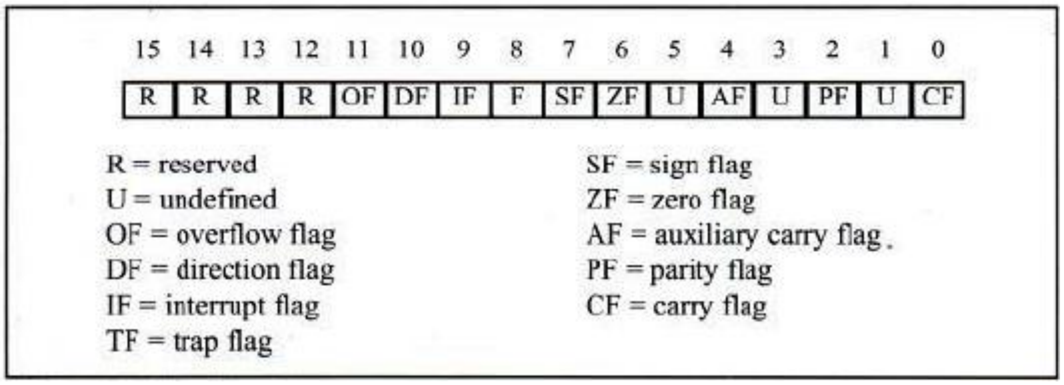


Fig: Non-overlapping vs. Overlapping Segments

<u>**THE FLAG REGISTER:**</u>

- The ***flag register*** is a 16-bit register sometimes referred to as the ***status register***. Although the register is 16 bits wide, only some of the bits are used. The rest are either undefined or reserved by Intel.
- Six of the flags are called ***conditional flags***, meaning that they indicate some condition that resulted after an instruction was executed. These six are CF, PF, AF, ZF, SF, and OF.
- The three remaining flags are sometimes called ***control flags***, since they are used to control the operation of instructions before they are executed.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	R	R	R	OF	DF	IF	F	SF	ZF	U	AF	U	PF	U	CF

R = reserved	SF = sign flag
U = undefined	ZF = zero flag
OF = overflow flag	AF = auxiliary carry flag.
DF = direction flag	PF = parity flag
IF = interrupt flag	CF = carry flag
TF = trap flag	

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					OF	DF	IF	TF	SF	ZF		AF		PF		CF

Fig: Flag Register

Key to remember: in **O**ne **D**ay **I**nternational **T**endulkar **S**cored **Z**ero, **A**ll **P**eople **C**ried.

Bits of the Flag Register:

CF, the Carry Flag – This flag is set whenever there is a carry out, either from d7 after an 8-bit operation or from d15 after a 16-bitdata operation.

PF, the Parity Flag – After certain operations, the parity of the result's low-order byte is checked. If the byte has an even number of 1s, the parity flag is set to 1; otherwise, it iscleared.

AF, Auxiliary Carry Flag – If there is a carry from d3 to d4 of an operation, this bit is set; otherwise, it is cleared (set equal to zero). This flag is used by the instructions that perform BCD (binary coded decimal) arithmetic.

ZF, the Zero Flag – The zero flag is set to 1 if the result of arithmetic or logical operation is zero; otherwise, it is cleared.

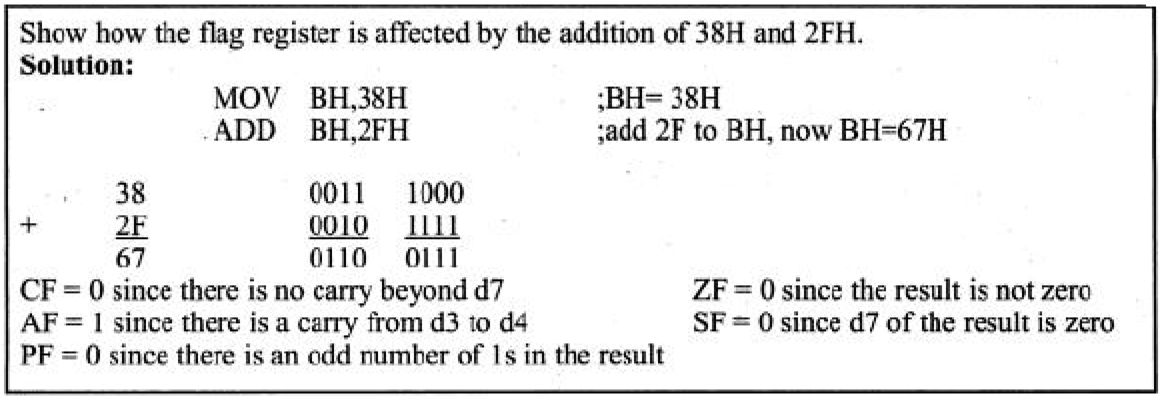
SF, the Sign Flag – Binary representation of signed numbers uses the most significant bit as the sign bit. After arithmetic or logic operations, the status of this sign bit is copied into the SF, thereby indicating the sign of the result.

TF, the Trap Flag – When this flag is set, it allows the program to single-step, meaning to execute one instruction at a time. Single-stepping is used for debugging purposes.

IF, Interrupt Enable Flag – This bit is set or cleared to enable or disable only the external maskable interrupt requests.

DF, the Direction Flag – This bit is used to control the direction of string operations. If D = 1, the registers are automatically decremented; if D = 0, the registers are automatically incremented. The state of the D flag bit is controlled by STD (set D flag) and CLD (clear D flag) instructions**.**

OF, the Overflow Flag – This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations. The overflow flag is only used to detect errors in signed arithmetic operations.



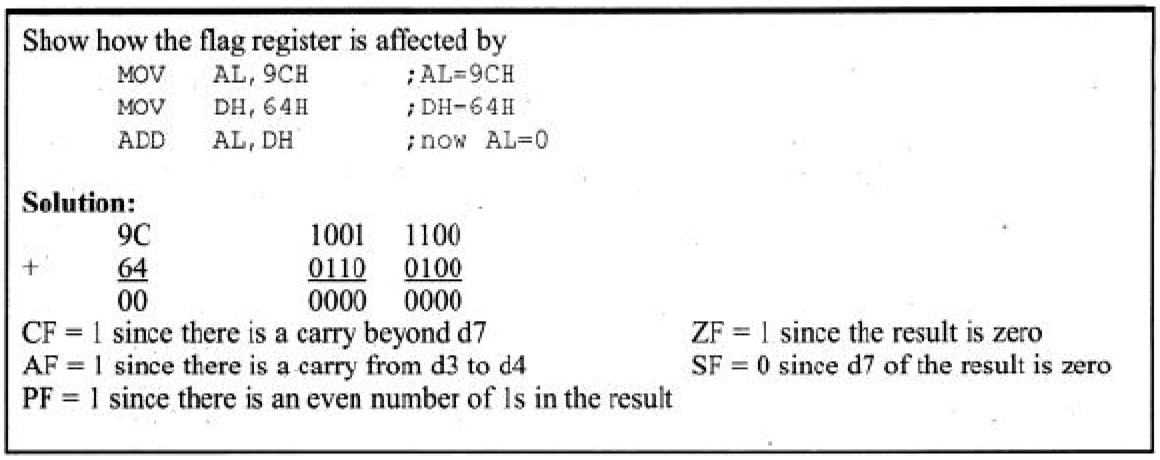
Show how the flag register is affected by the addition of 38H and 2FH.
Solution:

```
        MOV   BH,38H          ;BH= 38H
        ADD   BH,2FH          ;add 2F to BH, now BH=67H

    38        0011  1000
+   2F        0010  1111
    67        0110  0111
```

CF = 0 since there is no carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 0 since there is an odd number of 1s in the result
ZF = 0 since the result is not zero
SF = 0 since d7 of the result is zero

Flag Register and ADD Instruction:

The flag bits affected by the ADD instruction are CF, PF, AF, ZF, SF, and OF. The following examples are given to understand how each of these flag bits is affected. Please note that, MOV instructions have no effect on the flag.



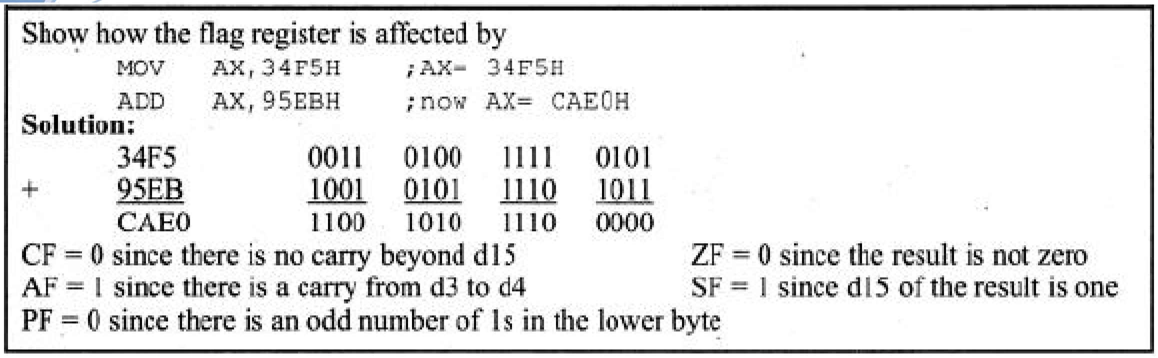
Show how the flag register is affected by

```
        MOV   AL,9CH       ;AL=9CH
        MOV   DH,64H       ;DH-64H
        ADD   AL,DH        ;now AL=0
```

Solution:

```
    9C        1001  1100
+   64        0110  0100
    00        0000  0000
```

CF = 1 since there is a carry beyond d7
AF = 1 since there is a carry from d3 to d4
PF = 1 since there is an even number of 1s in the result
ZF = 1 since the result is zero
SF = 0 since d7 of the result is zero

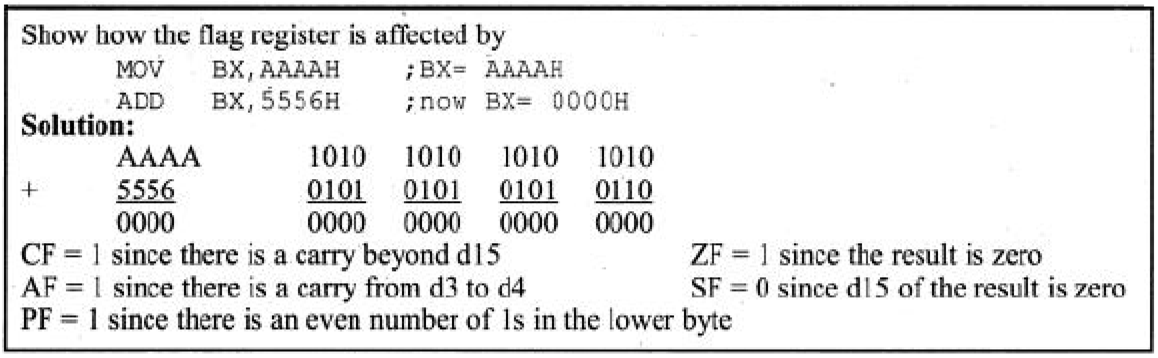

Show how the flag register is affected by

```
        MOV    AX,34F5H     ;AX= 34F5H
        ADD    AX,95EBH     ;now AX= CAE0H
```

Solution:

```
        34F5            0011   0100   1111   0101
+       95EB            1001   0101   1110   1011
        CAE0            1100   1010   1110   0000
```

CF = 0 since there is no carry beyond d15
ZF = 0 since the result is not zero
AF = 1 since there is a carry from d3 to d4
SF = 1 since d15 of the result is one
PF = 0 since there is an odd number of 1s in the lower byte

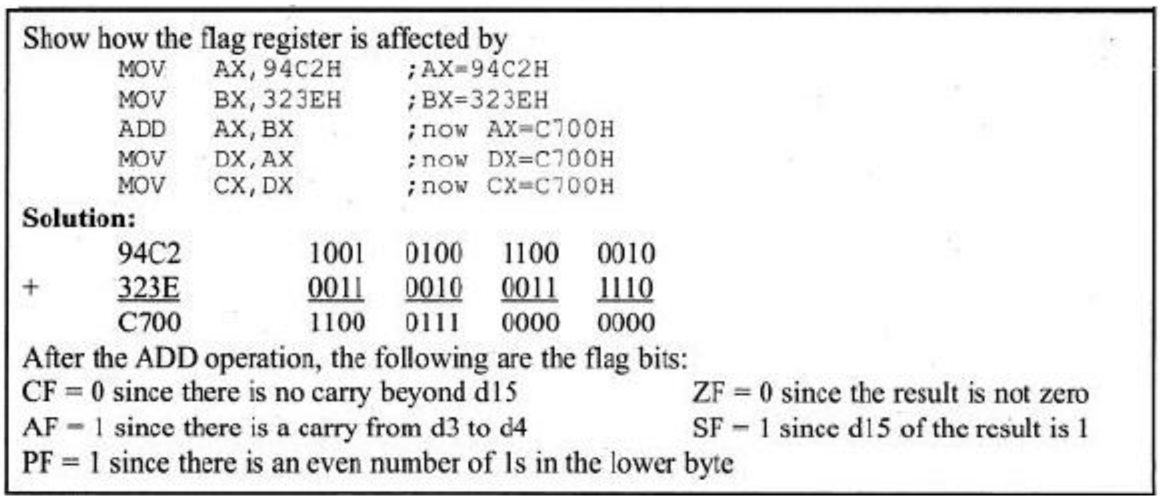

Show how the flag register is affected by

```
        MOV    BX,AAAAH     ;BX= AAAAH
        ADD    BX,5556H     ;now BX= 0000H
```

Solution:

```
        AAAA            1010   1010   1010   1010
+       5556            0101   0101   0101   0110
        0000            0000   0000   0000   0000
```

CF = 1 since there is a carry beyond d15
ZF = 1 since the result is zero
AF = 1 since there is a carry from d3 to d4
SF = 0 since d15 of the result is zero
PF = 1 since there is an even number of 1s in the lower byte


Show how the flag register is affected by

```
        MOV    AX,94C2H     ;AX=94C2H
        MOV    BX,323EH     ;BX=323EH
        ADD    AX,BX        ;now AX=C700H
        MOV    DX,AX        ;now DX=C700H
        MOV    CX,DX        ;now CX=C700H
```

Solution:

```
        94C2            1001   0100   1100   0010
+       323E            0011   0010   0011   1110
        C700            1100   0111   0000   0000
```

After the ADD operation, the following are the flag bits:
CF = 0 since there is no carry beyond d15
ZF = 0 since the result is not zero
AF = 1 since there is a carry from d3 to d4
SF = 1 since d15 of the result is 1
PF = 1 since there is an even number of 1s in the lower byte

Use of Zero Flag for Looping:

- ✓ One of the most widely used applications of the flag register is the use of the zero flag to implement program loops.
- ✓ The term *loop* refers to a set of instructions that is repeated a number of times. For example, to add 5 bytes of data, a counter can be used to keep track of how many times the loop needs to be repeated. Each time the addition is performed the counter is decremented and the zero flag is checked. When the counter becomes zero, the zero flag is set (ZF = 1) and the loop is stopped.

- ✓ The following example shows the implementation of the looping concept in the program, which adds 5 bytes of data. Register CX is used to hold the counter and BX is the offset pointer (SI or DI could have been used instead). AL is initialized before the start of the loop.
- ✓ In each iteration; ZF is checked by the JNZ instruction. JNZ stands for "Jump Not Zero" meaning that, if ZF = 0, jump to a new address. If ZF = 1, the jump is not performed and the instruction below the jump will be executed.
- ✓ Notice that the JNZ instruction must come immediately after the instruction that decrements CX since JNZ needs to check the effect of "DEC CX" on ZF. If any other instruction(s) were placed between them, that instruction(s) might affect the zero flag.

```
        MOV   CX,05     ;CX holds the loop count
        MOV   BX,0200H  ;BX holds the offset data address
        MOV   AL,00     ;initialize AL
ADD_LP: ADD   AL,[BX]   ;add the next byte to AL
        INC   BX        ;increment the data pointer
        DEC   CX        ;decrement the loop counter
        JNZ   ADD_LP    ;jump to next iteration if counter not zero
```

x86 ADDRESSING MODES:

The CPU can access operands (data) in various ways, called *addressing modes*. The number of addressing modes is determined when the microprocessor is designed and cannot be changed. The x86 provides a total of seven distinct addressing modes:

[1] Register [2] Immediate [3] Direct [4] Register Indirect
[5] Based Relative [6] Indexed Relative [7] Based Indexed Relative

Table: Summary of the x86 Addressing Modes

Addressing Mode	Operand	Default Segment
Register	reg	none
Immediate	data	none
Direct	[offset]	DS
Register indirect	[BX]	DS
	[SI]	DS
	[DI]	DS
Based relative	[BX]+disp	DS
	[BP]+disp	SS
Indexed relative	[DI]+disp	DS
	[SI]+disp	SS DS
Based indexed relative	[BX][SI]+disp	DS
	[BX][DI]+disp	DS
	[BP][SI]+disp	SS
	[BP][DI]+disp	SS

1. Register AddressingMode

The register addressing mode involves the use of registers to hold the data to be manipulated. Memory is not accessed when this addressing mode is executed;therefore, it is relatively fast.

```
MOV   BX,DX  ;copy the contents of DX into BX
MOV   ES,AX  ;copy the contents of AX into ES
ADD   AL,BH  ;add the contents of BH to contents of AL
```

2. Immediate Addressing Mode

In immediate addressing mode (as the name implies), when the instruction is assembled, the operand comes immediately after the opcode. For this reason, this addressing mode executes quickly. In this addressing mode, the source operand is a constant. Immediate addressing mode can be used to load information into any of the registers except the segment registers and flag registers.

```
MOV   AX,2550H      ;move 2550H into AX
MOV CX,625          ;load the decimal value 625 into CX
MOV BL,40H          ;load 40H into BL
```

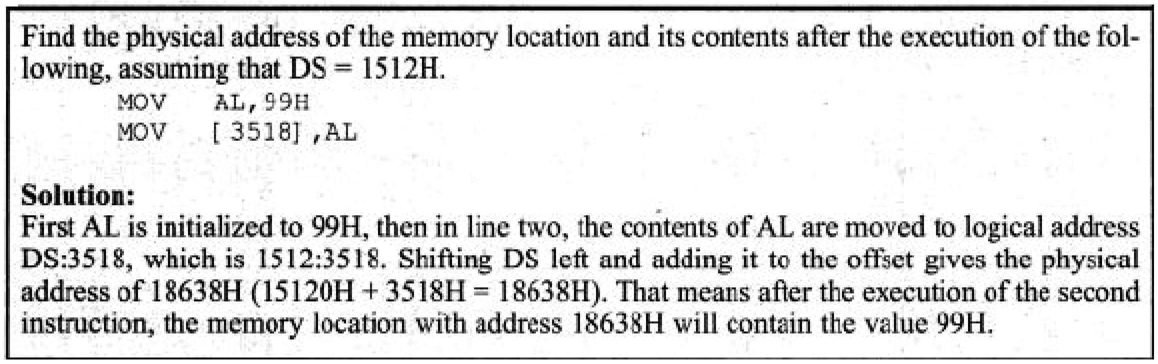
3. Direct Addressing Mode

In the direct addressing mode, the data is in some memory location(s) and the address of the data in memory comes immediately after the instruction. Note that, in immediate addressing mode, the operand itself is provided with the instruction; whereas in direct addressing mode, the address of the operand is provided with the instruction. This address is the offset address and one can calculate the physical address by shifting left the DS register and adding it to the offset as follows:

$$PA = \left\{ \quad DS \quad \right\} : \left\{ \text{Direct Address} \right\}$$

```
MOV DL,[2400]   ;move contents of DS:2400H into DL
```

Notice the bracket around the address. In the absence of this bracket, executing the command will give an error since it is interpreted to move the value 2400 (16-bit data) into register DL, an 8-bit register.



Find the physical address of the memory location and its contents after the execution of the following, assuming that DS = 1512H.

```
MOV    AL,99H
MOV    [3518],AL
```

Solution:
First AL is initialized to 99H, then in line two, the contents of AL are moved to logical address DS:3518, which is 1512:3518. Shifting DS left and adding it to the offset gives the physical address of 18638H (15120H + 3518H = 18638H). That means after the execution of the second instruction, the memory location with address 18638H will contain the value 99H.

Before *After*

Eg:	*MOV BX, [5634]*	*BX*	*~~ABCDH~~*	*8645H*
		DS:5634H	*45H*	*LS byte*
		DS:5635H	*86H*	*MS byte*
			Before	*After*

Eg:	*MOV CL, [5634]*	*CL*	*~~F2H~~*	*45H*
		DS:5634H	*45H*	
		DS:5635H	*86H*	

4. Register Indirect Addressing Mode

In the register indirect addressing mode, the address of the memory location where the operand resides is held by a register. The registers used for this purpose are SI, Dl, and BX. If these three registers are used as pointers, that is, if they hold the offset of the memory location, they must be combined with DS in order to generate the 20-bit physical address.

$$PA = \left\{ DS \right\} : \left\{ \begin{matrix} BX \\ SI \\ DI \end{matrix} \right\}$$

```
MOV AL,[BX]  ;moves into AL the contents of the memory
             ;location pointed to by DS:BX.
```

Notice that BX is in brackets. In the absence of brackets, the code is interpreted as an instruction moving the contents of register BX to AL (which gives an error because source and destination do not match); instead of the contents of the memory location whose offset address is in BX. The physical address is calculated by shifting DS left one hex position and adding BX to it. The same rules apply when using register SI or DI.

```
MOV   CL,[SI]       ;move contents of DS:SI into CL
MOV   [DI],AH       ;move contents of AH into DS:DI
```

Assume that DS = 1120, SI = 2498, and AX = 17FE. Show the contents of memory locations after the execution of "`MOV [SI],AX`".

Solution:

The contents of AX are moved into memory locations with logical address DS:SI and DS:SI + 1; therefore, the physical address starts at DS (shifted left) + SI = 13698. According to the little endian convention, low address 13698H contains FE, the low byte, and high address 13699H will contain 17, the high byte.

5. Based Relative Addressing Mode

In the based relative addressing mode, base registers BX and BP, as well as a displacement value are used to calculate (what is called) the *effective address*. The default segments used for the calculation of the physical address (PA) are DS for BX and SS for BP.

$$PA = \left\{\begin{matrix} DS \\ \text{or} \\ SS \end{matrix}\right\} : \left\{\begin{matrix} BX \\ \text{or} \\ BP \end{matrix}\right\} + \text{8 or 16 bit displacement}$$

```
MOV CX,[BX]+10     ;move DS:BX+10 and DS:BX+10+1 into CX
                   ;PA = DS (shifted left) + BX + 10
```

Alternative codings are *"MOV CX, [BX+10]"* or *"MOV CX, 10[BX]"*. In the case of BP register –

```
MOV  AL,[BP]+5    ;PA = SS (shifted left) + BP + 5
```

Alternative codings are *"MOV AL, [BP+5]"* or *"MOV AL, 5[BP]"*.

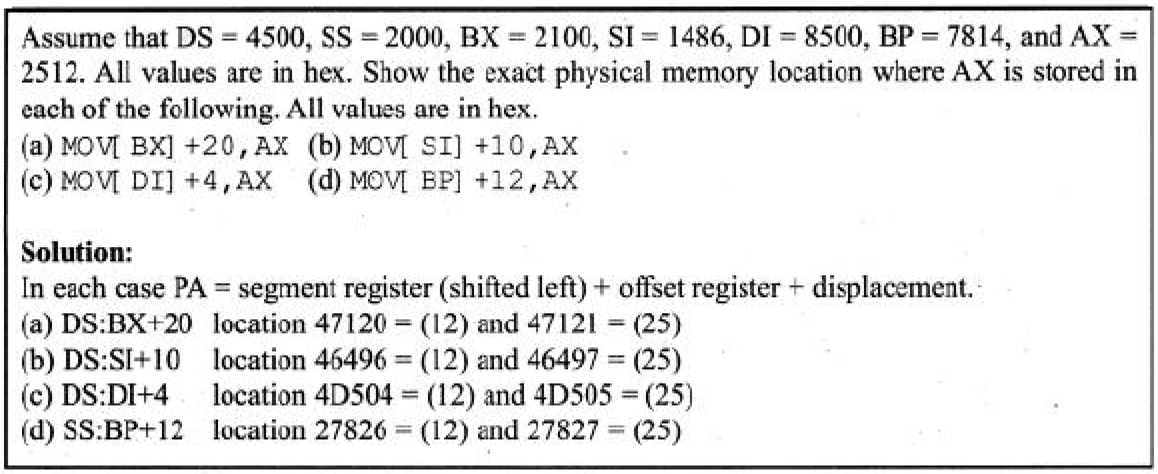
- In *"MOV AL, [BP+5]"*, BP+5 is called the effective address; since the 5th byte from the beginning of the offset BP is moved to register AL. Similarly, in *"MOV CX, [BX+10]"*, BX+10 is called the effective address.

6. Indexed Relative Addressing Mode

The indexed relative addressing mode works the same as the based relative addressing mode, except that registers DI and SI hold the offset address.

$$PA = \left\{\begin{matrix} DS \\ \text{or} \\ SS \end{matrix}\right\} : \left\{\begin{matrix} SI \\ \text{or} \\ DI \end{matrix}\right\} + \text{8 or 16 bit displacement}$$

```
MOV  DX,[SI]+5     ;PA = DS (shifted left) + SI + 5
MOV  CL,[DI]+20    ;PA = DS (shifted left) + DI + 20
```


Assume that DS = 4500, SS = 2000, BX = 2100, SI = 1486, DI = 8500, BP = 7814, and AX = 2512. All values are in hex. Show the exact physical memory location where AX is stored in each of the following. All values are in hex.

(a) `MOV[BX]+20,AX` (b) `MOV[SI]+10,AX`

(c) `MOV[DI]+4,AX` (d) `MOV[BP]+12,AX`

Solution:

In each case PA = segment register (shifted left) + offset register + displacement.

(a) DS:BX+20 location 47120 = (12) and 47121 = (25)

(b) DS:SI+10 location 46496 = (12) and 46497 = (25)

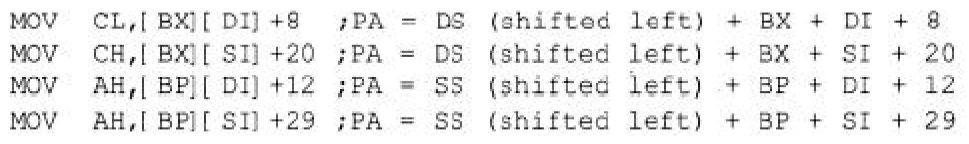
(c) DS:DI+4 location 4D504 = (12) and 4D505 = (25)

(d) SS:BP+12 location 27826 = (12) and 27827 = (25)

7. Based Indexed Addressing Mode

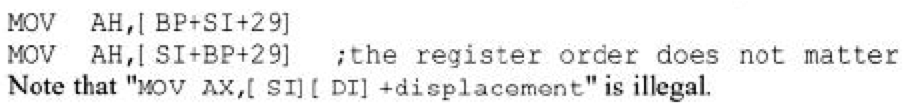
By combining based and indexed addressing modes, a new addressing mode is derived called the based indexed addressing mode. In this mode, one base register and one index register are used.

$$PA = \begin{Bmatrix} DS \\ \text{or} \\ SS \end{Bmatrix} : \begin{Bmatrix} BX \\ \text{or} \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ \text{or} \\ DI \end{Bmatrix} + 8 \text{ or } 16\text{bit displacement}$$



```
MOV  CL,[BX][DI]+8   ;PA = DS (shifted left) + BX + DI + 8
MOV  CH,[BX][SI]+20  ;PA = DS (shifted left) + BX + SI + 20
MOV  AH,[BP][DI]+12  ;PA = SS (shifted left) + BP + DI + 12
MOV  AH,[BP][SI]+29  ;PA = SS (shifted left) + BP + SI + 29
```

The coding of the instructions above can vary. The last example can also be written as –



```
MOV  AH,[BP+SI+29]
MOV  AH,[SI+BP+29]   ;the register order does not matter
```

Note that "MOV AX,[SI][DI]+displacement" is illegal.

Segment Overrides:

The following Table summarizes the offset registers that can be used with the four segment registers.

Table: Default Segments and Offset Register Pairs

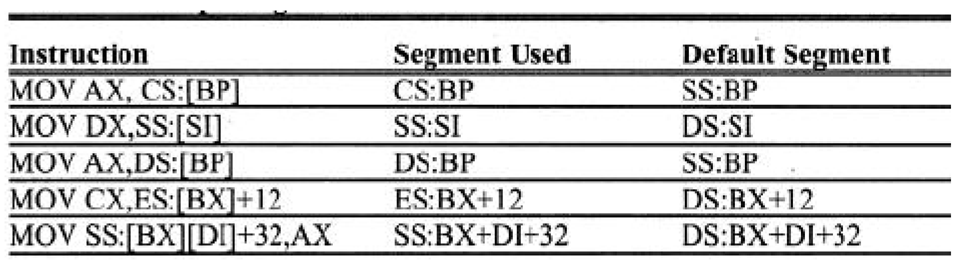
Segment	Offset	Special Purpose
CS	IP	Instruction address
DS	SI, DI, BX, an 8- or 16-bit number	Data address
SS	SP or BP	Stack address
ES	SI, DI, BX for string instructions	String destination address

The x86 CPU allows the program to override the default segment and use any segment register. To do that, one needs to specify the segment in the code.

For example, in "MOV AL, [BX]", the physical address of the operand to be moved into AL is DS: BX. To override that default, specify the desired segment in the instruction as "MOV AL, ES: [BX] ". Now the address of the operand being moved to AL is ES: BX instead of DS: BX.

The following Table shows more examples of segment overrides shown next to the default address in the absence of the override.

Table: Sample Segment Overrides



Instruction	Segment Used	Default Segment
MOV AX, CS:[BP]	CS:BP	SS:BP
MOV DX,SS:[SI]	SS:SI	DS:SI
MOV AX,DS:[BP]	DS:BP	SS:BP
MOV CX,ES:[BX]+12	ES:BX+12	DS:BX+12
MOV SS:[BX][DI]+32,AX	SS:BX+DI+32	DS:BX+DI+32

ASSEMBLY LANGUAGE PROGRAMMING

DIRECTIVES AND A SIMPLE PROGRAM:

A given Assembly language program (ALP) is a series of statements. There are two types of statements in x86 ALP:

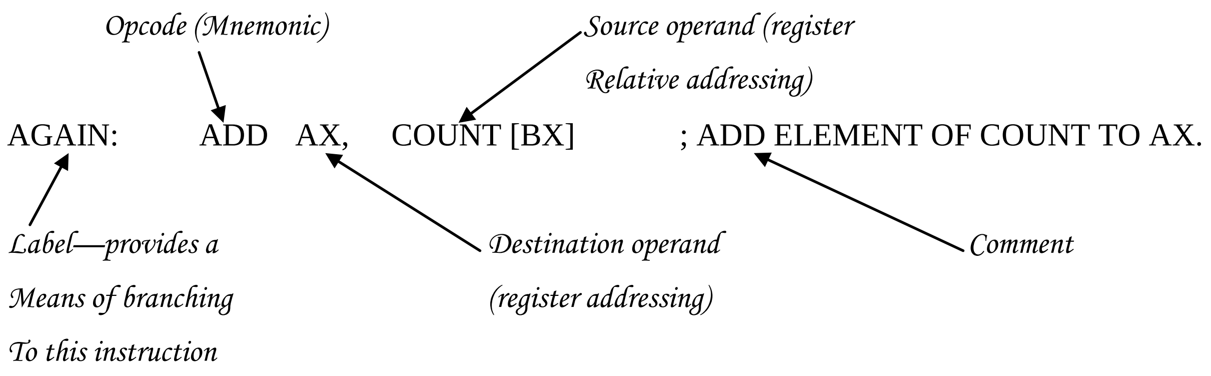
1. *Assembly language instructions* – instructions that are given to the microprocessor to do the specific task. The Assembly language instruction can be translated into object code or machine language. (*E.g.: MOV, ADD, etc.*)
2. *Pseudo instructions/Directives* – instructions that give directions to the assembler about how it should translate the Assembly language instructions into machine code. These instructions are not translated into machine code. They are used by the assembler to organize the program as well as other output files. (*E.g.: DB, DW, ASSUME, etc.*)

An Assembly language instruction consists of four fields:

```
[label:]   mnemonic  [operands]  [;comment]
```

Brackets indicate that the field is optional; do not type the brackets.

E.g.:



```
AGAIN:      ADD   AX,    COUNT [BX]        ; ADD ELEMENT OF COUNT TO AX.
```

Opcode (Mnemonic)
Source operand (register Relative addressing)
Label—provides a Means of branching To this instruction
Destination operand (register addressing)
Comment

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed 31 characters. Labels for directives do not need to end with a colon. A label must end with a colon when it refers to an opcode generating instruction; the colon indicates to the assembler that this refers to code within this code segment.

2, 3. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as *ADD AL, BL* or *MOV AX, 6764*; ADD and MOV are mnemonic opcode, and "*AL, BL*" and "*AX, 6764"* are the operands.

4. The comment filed begins with a ";". The assembler ignores comments. The comments are optional, but are highly recommended for someone to read and understand the program.

Model Definition:

The first statement in an Assembly language program is the MODEL directive. This directive selects the size of the memory model. Among the options for the memory model are SMALL, MEDIUM, COMPACT, and LARGE.

•MODEL SMALL ; this directive defines the model as small

SMALL is one of the most widely used memory models for Assembly language programs This model uses a maximum of 64K bytes of memory for code and another 64KB for data. The other models are defined as follows:

```
.MODEL MEDIUM     ;the data must fit into 64K bytes
                  ;but the code can exceed 64K bytes of memory
.MODEL COMPACT    ;the data can exceed 64K bytes
                  ;but the code cannot exceed 64K bytes
.MODEL LARGE      ;both data and code can exceed 64K
                  ;but no single set of data should exceed 64K
.MODEL HUGE       ;both code and data can exceed 64K
                  ;data items (such as arrays) can exceed 64K
.MODEL TINY       ;used with COM files in which data and code
                  ;must fit into 64K bytes
```

Segment Definition:

The x86 CPU has four segment registers: CS (code segment), DS (data segment), SS (stack segment), and ES (extra segment). Every line of an Assembly language program must correspond to one of these segments. The simplified segment definition format uses three simple directives: ".CODE", ".DATA", and ".STACK", which correspond to the CS, DS, and SS registers, respectively.

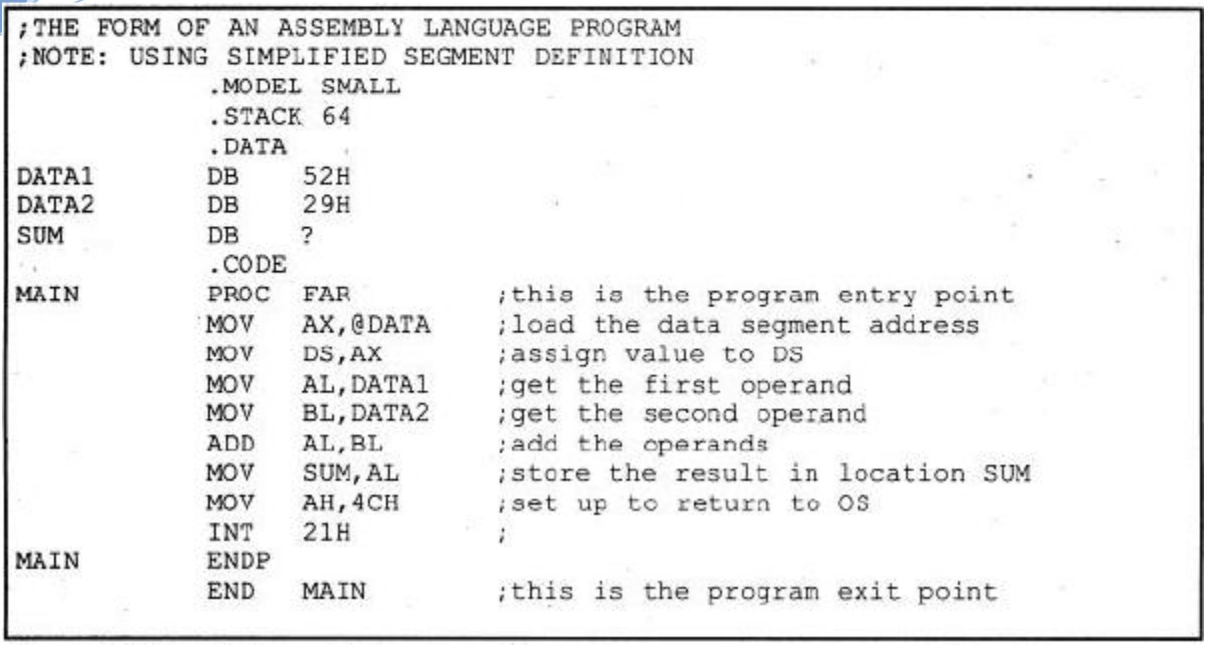
Segments of a Program:

Although one can write an Assembly language program that uses only one segment, normally a program consists of at least three segments: the stack segment, the data segment, and the code segment.

```
.STACK     ;marks the beginning of the stack segment
.DATA      ;marks the beginning of the data segment
.CODE      ;marks the beginning of the code segment
```

Assembly language statements are grouped into segments in order to be recognized by the assembler and consequently by the CPU.

- ✓ The stack segment defines storage for the stack
- ✓ The data segment defines the data that the program will use
- ✓ The code segment contains the Assembly language instructions.



```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
             .MODEL SMALL
             .STACK 64
             .DATA
DATA1        DB     52H
DATA2        DB     29H
SUM          DB     ?
             .CODE
MAIN         PROC   FAR          ;this is the program entry point
             MOV    AX,@DATA     ;load the data segment address
             MOV    DS,AX        ;assign value to DS
             MOV    AL,DATA1     ;get the first operand
             MOV    BL,DATA2     ;get the second operand
             ADD    AL,BL        ;add the operands
             MOV    SUM,AL       ;store the result in location SUM
             MOV    AH,4CH       ;set up to return to OS
             INT    21H          ;
MAIN         ENDP
             END    MAIN         ;this is the program exit point
```

Fig: Simple Assembly Language Program

•MODEL SMALL – directive defines a model that uses a maximum of 64KB of memory for code and another 64KB of memory for data.

•STACK 64 – directive reserves 64 bytes of memory for the stack.

•DATA – directive marks the beginning of the data segment.

- ✓ The data segment defines three data items: DATA1, DATA2, and SUM. Each is defined as DB (define byte). The DB directive is used by the assembler to allocate memory in byte-sized chunks. Memory can be allocated in different sizes; such a 2 bytes, which has the directive DW (define word).
- ✓ The data items defined in the data segment can be accessed in the code segment by their labels.
- ✓ DATA1 and DATA2 are given initial vales in the data section; and SUM in not given an initial value, but storage is set aside for it.

•CODE – directive marks the beginning of the code segment.

- ✓ MAIN – is the name (label) of procedure.
- ✓ PROC – directive defines a procedure. A ***procedure*** is a group of instructions designed to accomplish a specific function.
- ✓ A PROC directive may have the option FAR or NEAR, which are the program entry point(s).
- ✓ ENDP – directive defines the end of the procedure.
- ✓ PROC and ENDP statements must have the same label (here it is MAIN).

It is the job of the OS (operating system) to assign exact values for the segment registers. When program begins executing, the OS allocates some of RAM available to the segment registers. This is done as follows:

```
MOV AX,@DATA ;DATA refers to the start of the data segment
MOV DS,AX
```

No segment register can be loaded directly. Hence, two lines are required, as shown above.

END – directive ends the entire program by indicating to OS that the entry point MAIN has ended. The label for the entry point (MAIN, here) and the END must match.

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
; USING SIMPLIFIED SEGMENT DEFINITION
            .MODEL SMALL
            .STACK 64
            .DATA
            ;
            ;place data definitions here
            ;
            .CODE
MAIN        PROC   FAR         ;this is the program entry point
            MOV    AX,@DATA    ;load the data segment address
            MOV    DS,AX       ;assign value to DS
            ;
            ;place code here
            ;
            MOV    AH,4CH      ;set up to
            INT    21H         ;return to OS
MAIN        ENDP
            END    MAIN        ;this is the program exit point
```

ASSEMBLE, LINK AND RUN A PROGRAM:

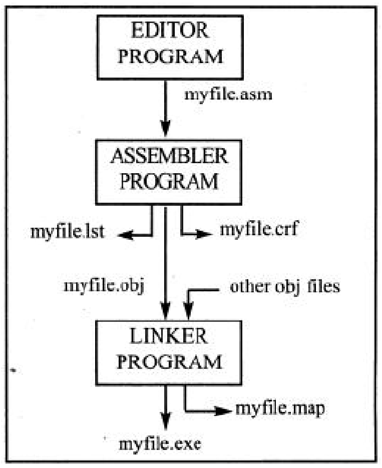
Once the Assembly language program has been written; there are three steps to create an executable Assembly language program:

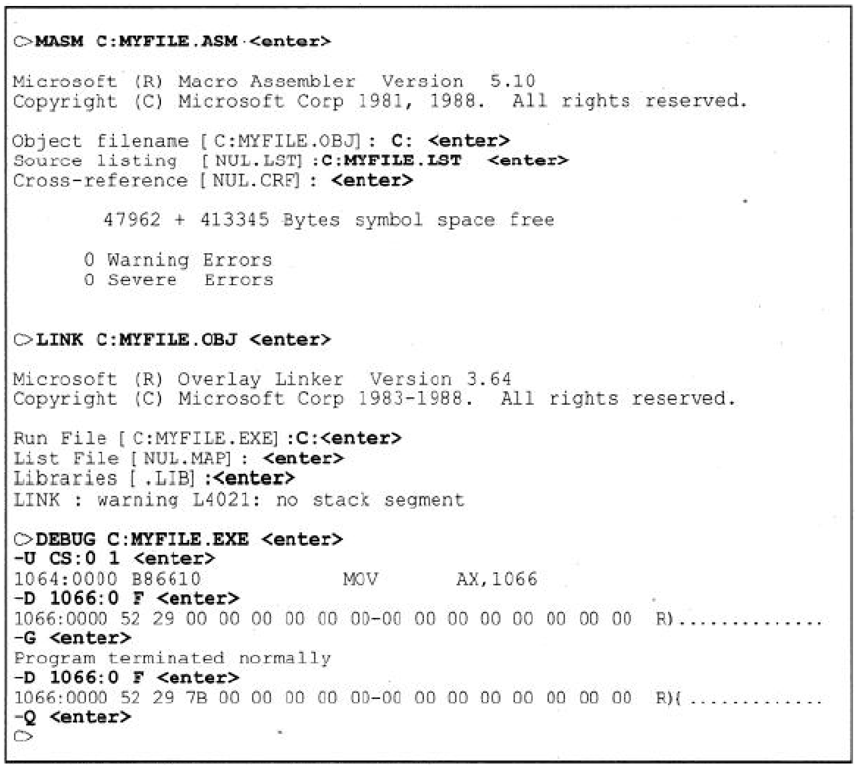
Step	Input	Program	Output
1. Edit the program	Keyboard	Editor	myfile.asm
2. Assemble the program	myfile.asm	MASM or TASM	myfile.obj
3. Link the program	myfile.obj	LINK or TLINK	myfile.exe

- Text editors are used to create and/or edit the program. These editors must be able to produce an ASCII file.
- The source file must end in ".asm" for these assemblers. This ".asm" file will be assembled by an assembler (such MASM/TASM).
    - The MASM and LINK programs are the assembler and linker programs for Microsoft's MASM assembler. In Borland's TASM assembler, TASM and TLINK programs are the assembler and linker programs.
- The assembler will produce an object file (.obj) and a list file (.lst), along with other files that may be useful to the programmer. All syntax errors produced by the assembler must be corrected in the object file.
    - The assembler creates the opcodes, operands, and offset addresses under the ".obj" file.

- The list file (.lst) lists all the opcodes and the offset addresses, as well as errors that the assembler detected. This file can be displayed on the monitor by the command: C>type myfile.lst | more.
- The cross-reference file (.crf) provides an alphabetical list of all symbols and tables used in the program as well as program line numbers in which they are referenced.

- The object file (.obj) is the input for the LINK program, which produces the executable program (.exe). The LINK program sets up the file, so that, it can be loaded by the OS and executed.
- We use DEBUG to execute the program and analyze the results.
  - When the program is working successfully, it can be run at the OS level by typing the command: C>myfile. When the program name is typed in at the OS level, the OS loads the program in memory. This is referred as *mapping*; which means that the program is mapped into the physical memory of the PC.
  - When there are many segments for code or data, there is a need to see where each is located and how many bytes are used by each. The ".map" file gives the name of each segment, where it starts, where it stops, and its size in bytes.





```
C>MASM C:MYFILE.ASM <enter>

Microsoft (R) Macro Assembler  Version  5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

Object filename [C:MYFILE.OBJ]: C: <enter>
Source listing  [NUL.LST]:C:MYFILE.LST  <enter>
Cross-reference [NUL.CRF]: <enter>

        47962 + 413345 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors


C>LINK C:MYFILE.OBJ <enter>

Microsoft (R) Overlay Linker  Version 3.64
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.

Run File [C:MYFILE.EXE]:C:<enter>
List File [NUL.MAP]: <enter>
Libraries [.LIB]:<enter>
LINK : warning L4021: no stack segment

C>DEBUG C:MYFILE.EXE <enter>
-U CS:0 1 <enter>
1064:0000 B86610            MOV      AX,1066
-D 1066:0 F <enter>
1066:0000 52 29 00 00 00 00 00 00-00 00 00 00 00 00 00 00   R)..............
-G <enter>
Program terminated normally
-D 1066:0 F <enter>
1066:0000 52 29 7B 00 00 00 00 00-00 00 00 00 00 00 00 00   R){.............
-Q <enter>
C>
```

Fig: Steps to Create a Program & Creating and Running the .exe File

PAGE and TITLE Directives:

The PAGE and the TITLE are two directives used make the ".lst" file more readable.

```
PAGE [lines],[columns]
```

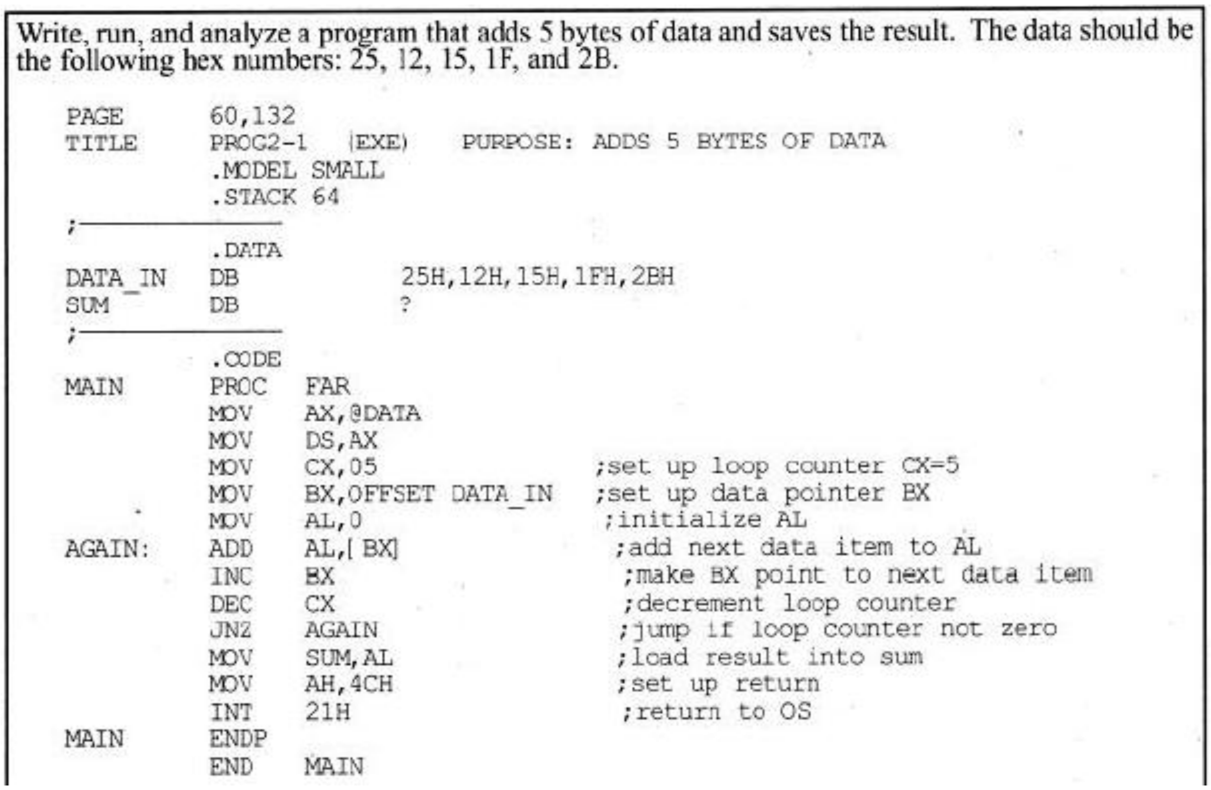
The PAGE directive tells the printer how the list should be printed. In the default mode, the output will have 66 lines per page and with a maximum of 80 characters per line. The default settings can be altered to 60 and 132 as follows:

```
PAGE 60,132
```

When the list is printed in more than one page, the assembler can be instructed to print the title of the program on the top of each page by using the TITLE directive. The text after the TITLE pseudo-instruction cannot be more than 60 ASCII characters.

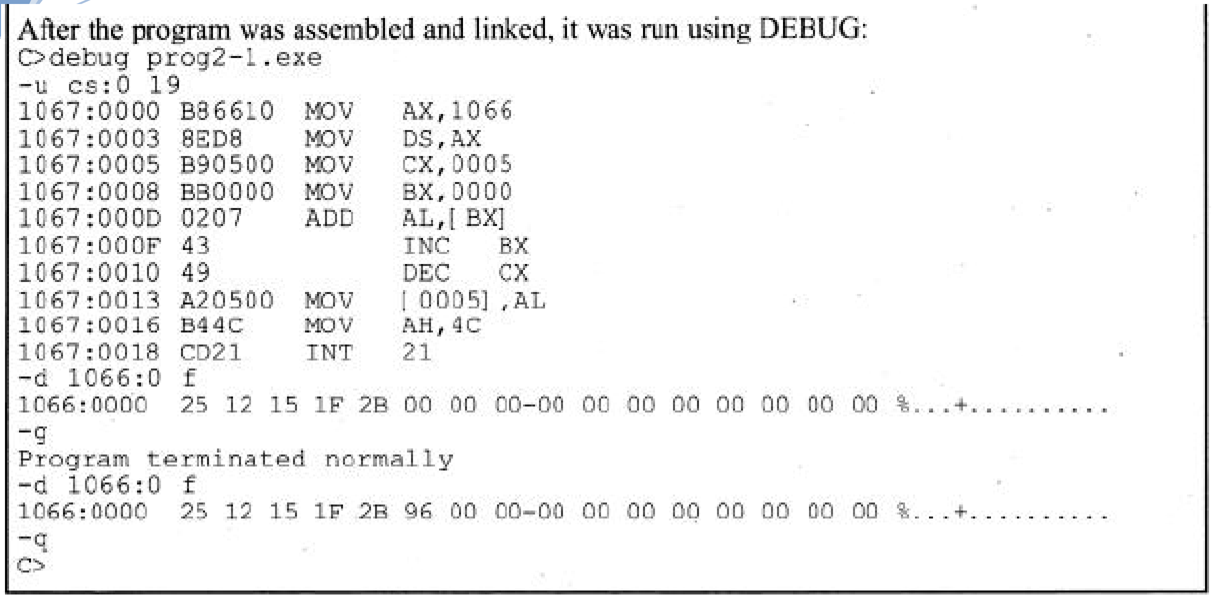
MORE SAMPLE PROGRAMS:

The following Fig shows the program and the list file generated when the program was assembled. After the program was assembled and linked, DEBUG was used to dump the code segment to see what value is assigned to the OS register. Remember that the value you get could be different for "MOV AX, xxxx" as well as for CS in the program examples.



Write, run, and analyze a program that adds 5 bytes of data and saves the result. The data should be the following hex numbers: 25, 12, 15, 1F, and 2B.

```
PAGE       60,132
TITLE      PROG2-1  (EXE)   PURPOSE: ADDS 5 BYTES OF DATA
           .MODEL SMALL
           .STACK 64
;---------------
           .DATA
DATA_IN    DB            25H,12H,15H,1FH,2BH
SUM        DB            ?
;---------------
           .CODE
MAIN       PROC    FAR
           MOV     AX,@DATA
           MOV     DS,AX
           MOV     CX,05               ;set up loop counter CX=5
           MOV     BX,OFFSET DATA_IN   ;set up data pointer BX
           MOV     AL,0                ;initialize AL
AGAIN:     ADD     AL,[BX]             ;add next data item to AL
           INC     BX                  ;make BX point to next data item
           DEC     CX                  ;decrement loop counter
           JNZ     AGAIN               ;jump if loop counter not zero
           MOV     SUM,AL              ;load result into sum
           MOV     AH,4CH              ;set up return
           INT     21H                 ;return to OS
MAIN       ENDP
           END     MAIN
```


After the program was assembled and linked, it was run using DEBUG:

```
C>debug prog2-1.exe
-u cs:0 19
1067:0000 B86610  MOV   AX,1066
1067:0003 8ED8    MOV   DS,AX
1067:0005 B90500  MOV   CX,0005
1067:0008 BB0000  MOV   BX,0000
1067:000D 0207    ADD   AL,[BX]
1067:000F 43            INC   BX
1067:0010 49            DEC   CX
1067:0013 A20500  MOV   [0005],AL
1067:0016 B44C    MOV   AH,4C
1067:0018 CD21    INT   21
-d 1066:0 f
1066:0000  25 12 15 1F 2B 00 00 00-00 00 00 00 00 00 00 00 %...+.........
-g
Program terminated normally
-d 1066:0 f
1066:0000  25 12 15 1F 2B 96 00 00-00 00 00 00 00 00 00 00 %...+.........
-q
C>
```

Fig: Program 2-1

⇨ **INC** destination – adds 1 to the specified destination. The destination may be a register or memory location.

Flags affected: AF, OF, PF, SF, and ZF. The CF is not affected.

Eg1: INC AL ; Add one to the contents of AL.

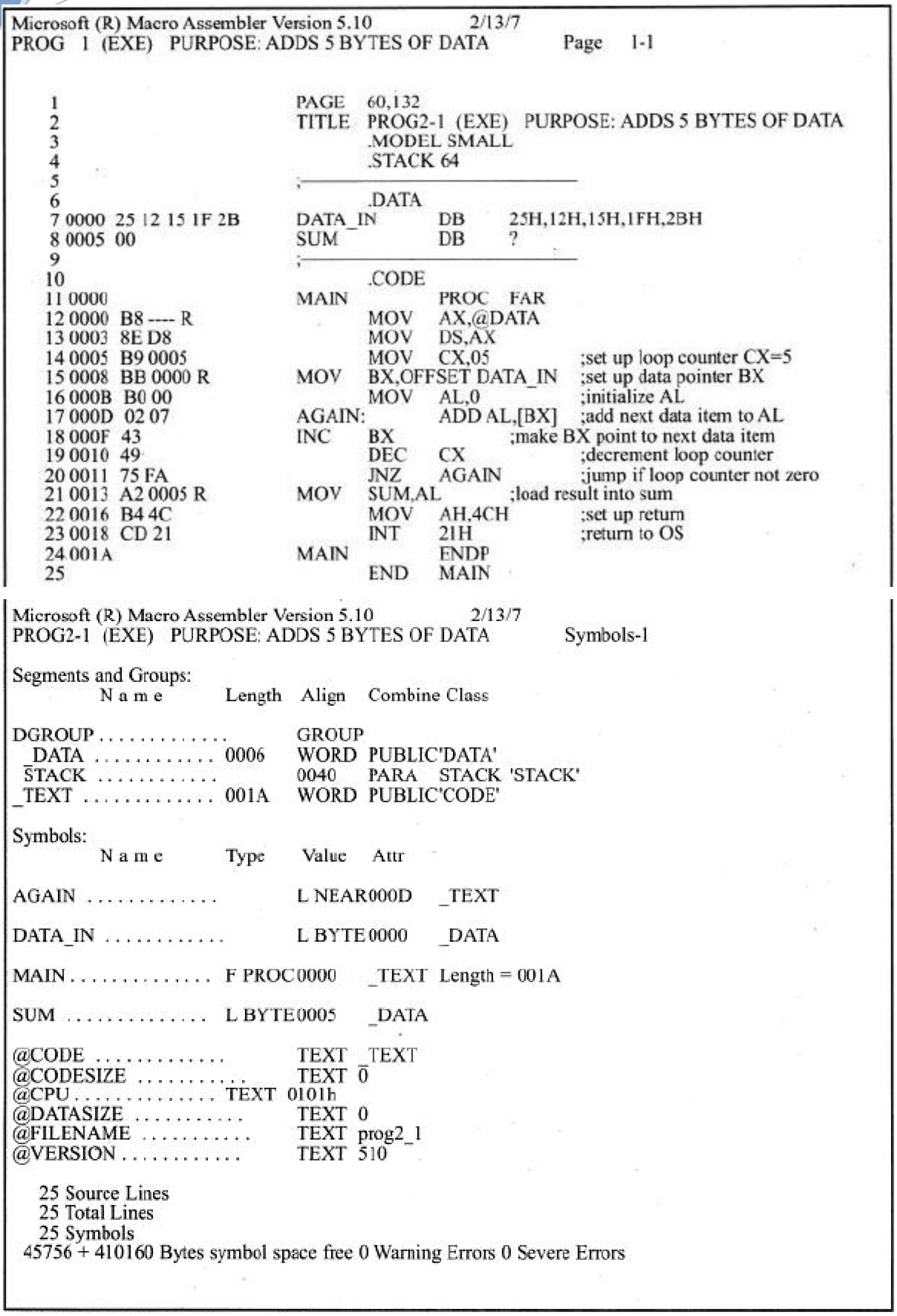
Eg2: INC BX ; Add one to the contents of BX.

⇨ **DEC** destination – subtract 1 from the specified destination. The destination may be a register or a memory location.

Flags affected: AF, OF, PF, SF, and ZF. The CF is not affected.

Eg: DEC AL ; Subtract 1 from the contents of AL.

⇨ **JNZ** label – jump if not zero; if ZF = 0, jumps to the label specified. Checks for zero flag.



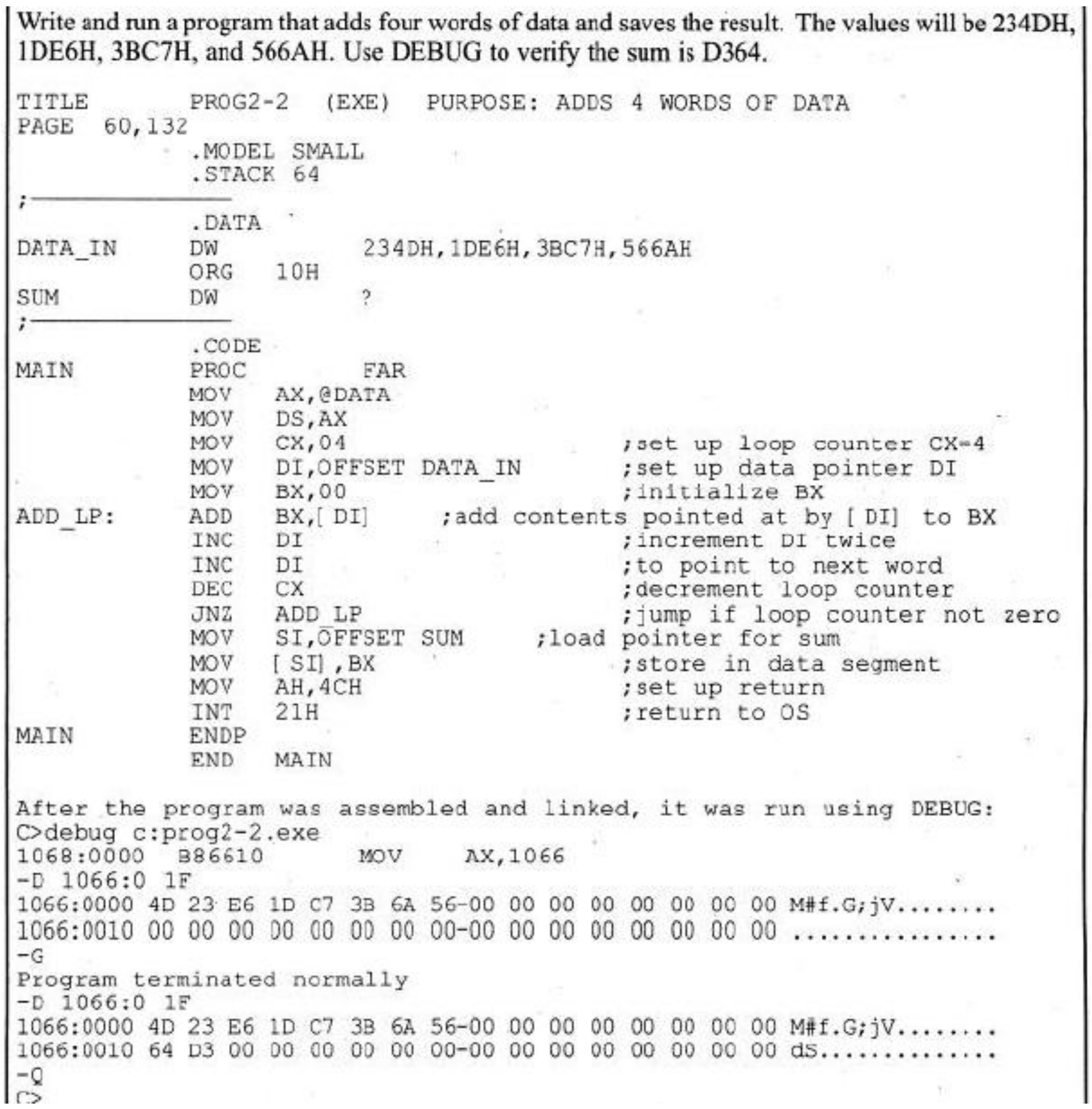
```
Microsoft (R) Macro Assembler Version 5.10          2/13/7
PROG   1  (EXE)   PURPOSE: ADDS 5 BYTES OF DATA            Page    1-1


     1                          PAGE   60,132
     2                          TITLE  PROG2-1 (EXE)  PURPOSE: ADDS 5 BYTES OF DATA
     3                                 .MODEL SMALL
     4                                 .STACK 64
     5                          ;-----------------------------
     6                                 .DATA
     7 0000  25 12 15 1F 2B     DATA_IN        DB      25H,12H,15H,1FH,2BH
     8 0005  00                 SUM            DB      ?
     9                          ;-----------------------------
    10                                 .CODE
    11 0000                     MAIN           PROC    FAR
    12 0000  B8 ---- R                 MOV     AX,@DATA
    13 0003  8E D8                     MOV     DS,AX
    14 0005  B9 0005                   MOV     CX,05          ;set up loop counter CX=5
    15 0008  BB 0000 R          MOV    BX,OFFSET DATA_IN      ;set up data pointer BX
    16 000B  B0 00                     MOV     AL,0           ;initialize AL
    17 000D  02 07              AGAIN:         ADD AL,[BX]    ;add next data item to AL
    18 000F  43                 INC    BX              ;make BX point to next data item
    19 0010  49                        DEC     CX             ;decrement loop counter
    20 0011  75 FA                     JNZ     AGAIN          ;jump if loop counter not zero
    21 0013  A2 0005 R          MOV    SUM,AL          ;load result into sum
    22 0016  B4 4C                     MOV     AH,4CH         ;set up return
    23 0018  CD 21                     INT     21H            ;return to OS
    24 001A                     MAIN           ENDP
    25                                 END     MAIN
```

```
Microsoft (R) Macro Assembler Version 5.10          2/13/7
PROG2-1  (EXE)   PURPOSE: ADDS 5 BYTES OF DATA            Symbols-1

Segments and Groups:
            N a m e         Length   Align   Combine Class

DGROUP . . . . . . . . . . . . .         GROUP
  _DATA . . . . . . . . . . . .  0006    WORD  PUBLIC'DATA'
  STACK . . . . . . . . . . . .          0040     PARA   STACK 'STACK'
_TEXT . . . . . . . . . . . . .  001A    WORD  PUBLIC'CODE'

Symbols:
            N a m e         Type     Value    Attr

AGAIN . . . . . . . . . . . .            L NEAR000D    _TEXT

DATA_IN . . . . . . . . . . .            L BYTE 0000    _DATA

MAIN . . . . . . . . . . . . . . F PROC 0000     _TEXT  Length = 001A

SUM . . . . . . . . . . . . . .  L BYTE 0005     _DATA

@CODE . . . . . . . . . . . .            TEXT  _TEXT
@CODESIZE . . . . . . . . . .            TEXT  0
@CPU . . . . . . . . . . . . . . TEXT  0101h
@DATASIZE . . . . . . . . . .            TEXT  0
@FILENAME . . . . . . . . . .            TEXT  prog2_1
@VERSION . . . . . . . . . . .           TEXT  510

    25 Source Lines
    25 Total Lines
    25 Symbols
  45756 + 410160 Bytes symbol space free  0 Warning Errors  0 Severe Errors
```

Fig: MASM List for Program 2-1

OFFSET: It is an operator which tells the assembler to determine the offset or displacement of a named data item (variable) from the start of the segment.

Eg: MOV AX, OFFSET MES1 ; Loads the offset of variable MES1 in AX register.

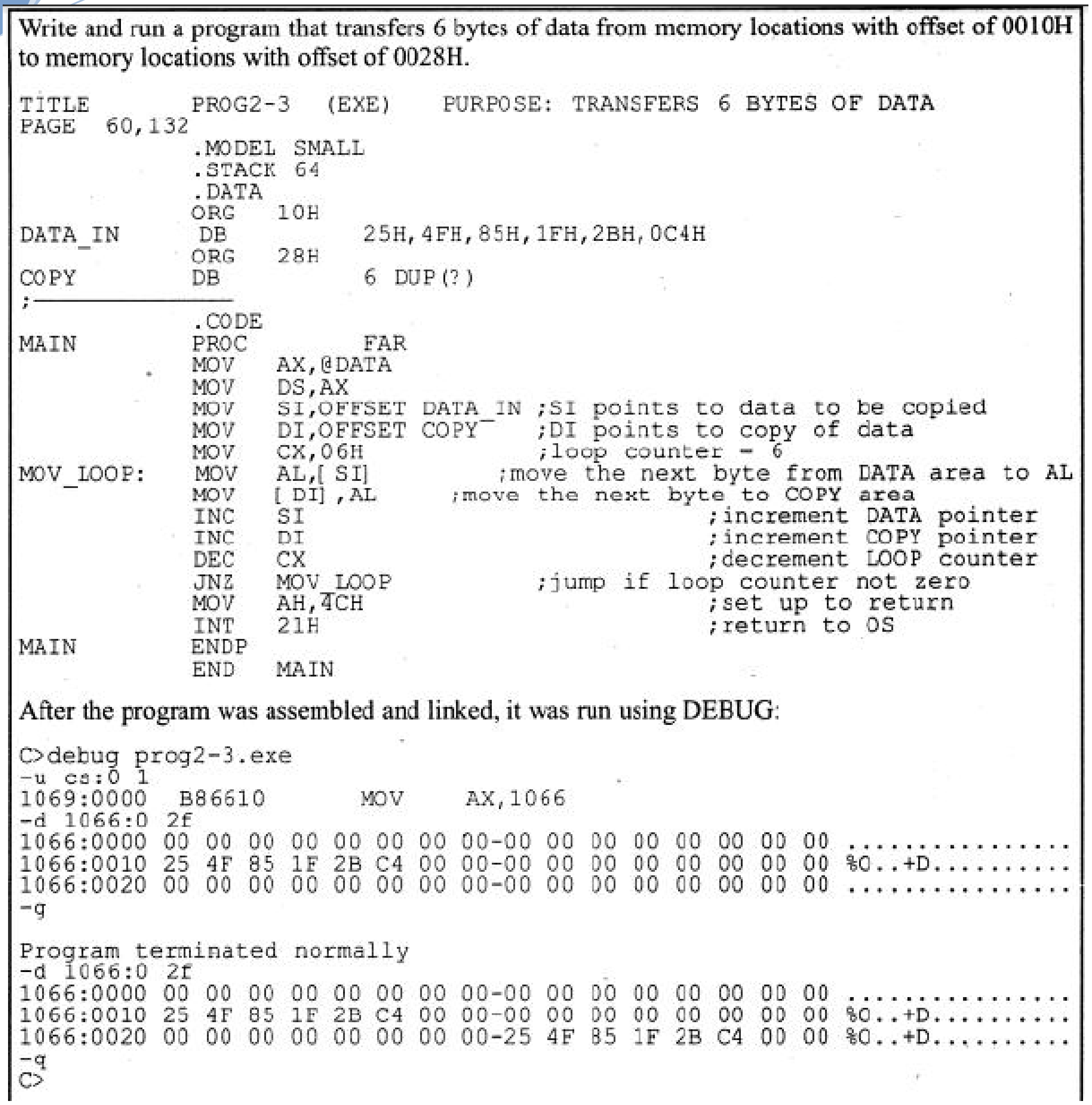


Write and run a program that adds four words of data and saves the result. The values will be 234DH, 1DE6H, 3BC7H, and 566AH. Use DEBUG to verify the sum is D364.

```
TITLE        PROG2-2   (EXE)   PURPOSE: ADDS 4 WORDS OF DATA
PAGE   60,132
             .MODEL SMALL
             .STACK 64
;-----------------
             .DATA
DATA_IN      DW           234DH,1DE6H,3BC7H,566AH
             ORG    10H
SUM          DW           ?
;-----------------
             .CODE
MAIN         PROC         FAR
             MOV    AX,@DATA
             MOV    DS,AX
             MOV    CX,04                    ;set up loop counter CX=4
             MOV    DI,OFFSET DATA_IN        ;set up data pointer DI
             MOV    BX,00                    ;initialize BX
ADD_LP:      ADD    BX,[DI]     ;add contents pointed at by [DI] to BX
             INC    DI                       ;increment DI twice
             INC    DI                       ;to point to next word
             DEC    CX                       ;decrement loop counter
             JNZ    ADD_LP                   ;jump if loop counter not zero
             MOV    SI,OFFSET SUM     ;load pointer for sum
             MOV    [SI],BX                  ;store in data segment
             MOV    AH,4CH                   ;set up return
             INT    21H                      ;return to OS
MAIN         ENDP
             END    MAIN

After the program was assembled and linked, it was run using DEBUG:
C>debug c:prog2-2.exe
1068:0000   B86610       MOV     AX,1066
-D 1066:0 1F
1066:0000 4D 23 E6 1D C7 3B 6A 56-00 00 00 00 00 00 00 00 M#f.G;jV........
1066:0010 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-G
Program terminated normally
-D 1066:0 1F
1066:0000 4D 23 E6 1D C7 3B 6A 56-00 00 00 00 00 00 00 00 M#f.G;jV........
1066:0010 64 D3 00 00 00 00 00 00-00 00 00 00 00 00 00 00 dS..............
-Q
C>
```

Fig: Program 2-2

The **ORG directive** can be used to set the offset addresses for data items. In the above program, the ORG directive causes SUM to be stored at DS: 0010.



Write and run a program that transfers 6 bytes of data from memory locations with offset of 0010H to memory locations with offset of 0028H.

```
TITLE        PROG2-3   (EXE)    PURPOSE: TRANSFERS 6 BYTES OF DATA
PAGE   60,132
             .MODEL SMALL
             .STACK 64
             .DATA
             ORG    10H
DATA_IN       DB          25H,4FH,85H,1FH,2BH,0C4H
             ORG    28H
COPY         DB          6 DUP(?)
;─────────────
             .CODE
MAIN         PROC        FAR
             MOV    AX,@DATA
             MOV    DS,AX
             MOV    SI,OFFSET DATA_IN ;SI points to data to be copied
             MOV    DI,OFFSET COPY    ;DI points to copy of data
             MOV    CX,06H            ;loop counter = 6
MOV_LOOP:     MOV    AL,[SI]        ;move the next byte from DATA area to AL
             MOV    [DI],AL      ;move the next byte to COPY area
             INC    SI                            ;increment DATA pointer
             INC    DI                            ;increment COPY pointer
             DEC    CX                            ;decrement LOOP counter
             JNZ    MOV_LOOP          ;jump if loop counter not zero
             MOV    AH,4CH                        ;set up to return
             INT    21H                           ;return to OS
MAIN         ENDP
             END    MAIN
```

After the program was assembled and linked, it was run using DEBUG:

```
C>debug prog2-3.exe
-u cs:0 1
1069:0000  B86610       MOV     AX,1066
-d 1066:0 2f
1066:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0010 25 4F 85 1F 2B C4 00 00-00 00 00 00 00 00 00 00 %O..+D..........
1066:0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-g

Program terminated normally
-d 1066:0 2f
1066:0000 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0010 25 4F 85 1F 2B C4 00 00-00 00 00 00 00 00 00 00 %O..+D..........
1066:0020 00 00 00 00 00 00 00 00-25 4F 85 1F 2B C4 00 00 %O..+D..........
-q
C>
```

Fig: Program 2-3

CONTROL TRANSFER INSTRUCTIONS:

In an ALP, instructions are executed sequentially. Sometimes, it is often necessary to transfer program control to a different location. Since the CS: IP registers always point to the address of the next instruction to be executed; they must be updated when a control transfer instruction is executed. There are many instructions in the x86 to achieve this.

FAR and NEAR:

- If control is transferred to a memory location within the current code segment, it is ***NEAR***. This is sometimes called ***intra-segment*** (within segment) jump.

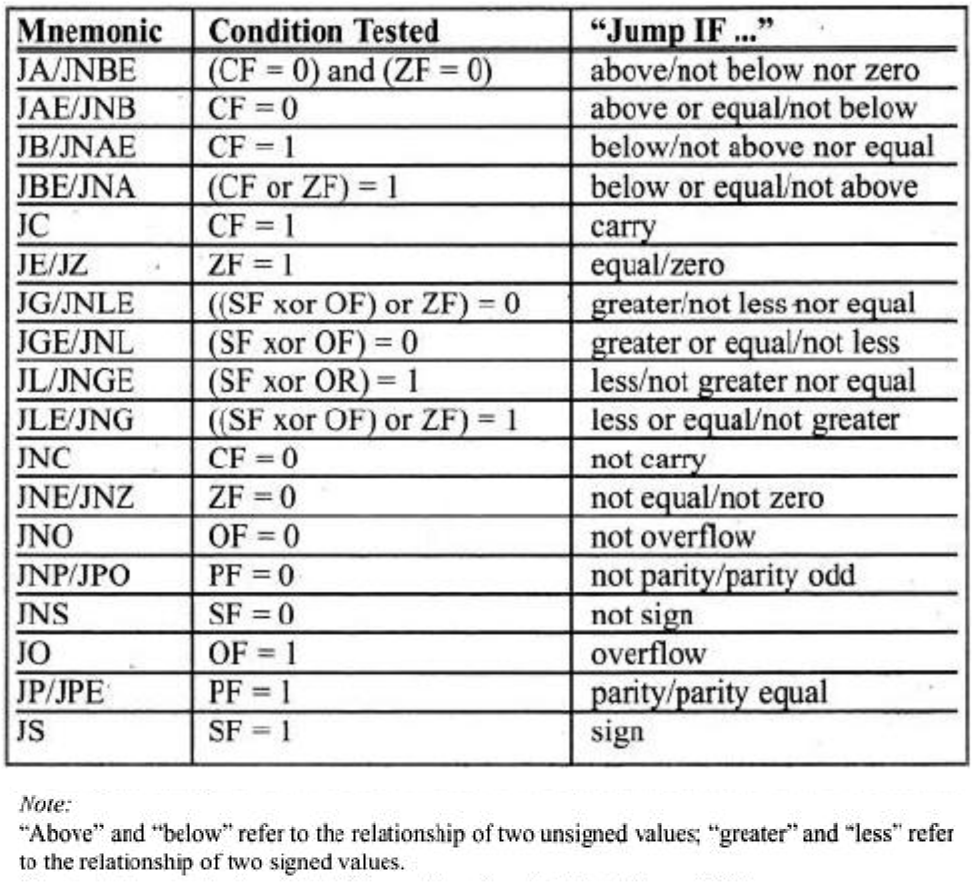
- In a NEAR jump, the IP is updated and CS remains the same, since control is still inside the current code segment.

- If control is transferred to a memory location outside the current code segment, it is a ***FAR*** or ***intersegment*** (between segments) jump.
  - In a FAR jump, because control is passing outside the current code segment, both CS and IP have to be updated to the new values.

Conditional Jumps:

In the conditional jump, control is transferred to a new location if a certain condition is met. The flag register is the one that indicates the current condition. For example, with "JNZ label", the processor looks at the zero flag to see if it is raised. If not, the CPU starts to fetch and execute instructions from the address of the label. If ZF = I, it will not jump but will execute the next instruction below the JNZ.

Table: 8086 Conditional Jump Instructions



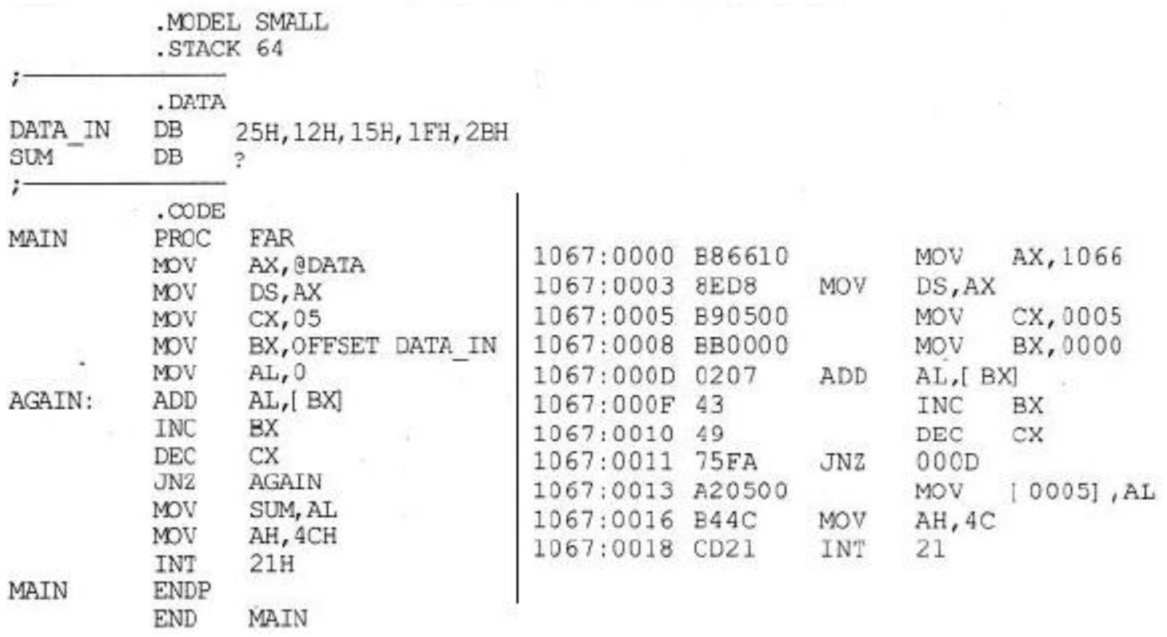
Mnemonic	Condition Tested	"Jump IF ..."
JA/JNBE	(CF = 0) and (ZF = 0)	above/not below nor zero
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OR) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNO	OF = 0	not overflow
JNP/JPO	PF = 0	not parity/parity odd
JNS	SF = 0	not sign
JO	OF = 1	overflow
JP/JPE	PF = 1	parity/parity equal
JS	SF = 1	sign

Note:
"Above" and "below" refer to the relationship of two unsigned values; "greater" and "less" refer to the relationship of two signed values.

Short Jumps:

- All conditional jumps are short jumps. In a short jump, the address of the target must be within –128 to +127 bytes of the IP.
- The conditional jump (short jump) is a two byte instruction: One byte is the opcode of the J condition and the second byte is a value between 00 and FF.

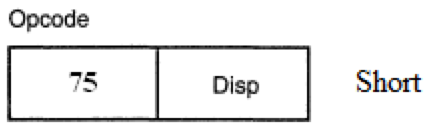
- An offset range of 00 to FF gives 256 possible addresses; these are split between backward jumps (to –128) and forward jumps (to +127).
- In a jump backward, the second byte is the 2's complement of the displacement value. To calculate the target address, the second byte is added to the IP of the instruction after the jump.



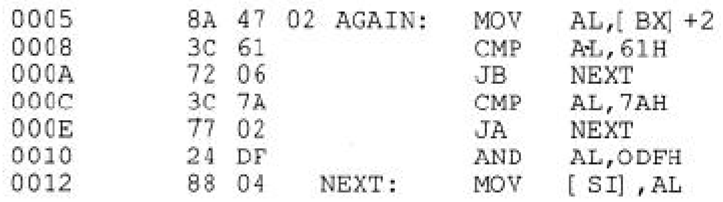
```
          .MODEL SMALL
          .STACK 64
;-----------------
          .DATA
DATA_IN   DB     25H,12H,15H,1FH,2BH
SUM       DB     ?
;-----------------
          .CODE
MAIN      PROC   FAR
          MOV    AX,@DATA
          MOV    DS,AX
          MOV    CX,05
          MOV    BX,OFFSET DATA_IN
          MOV    AL,0
AGAIN:    ADD    AL,[ BX]
          INC    BX
          DEC    CX
          JNZ    AGAIN
          MOV    SUM,AL
          MOV    AH,4CH
          INT    21H
MAIN      ENDP
          END    MAIN
```

```
1067:0000  B86610          MOV   AX,1066
1067:0003  8ED8      MOV   DS,AX
1067:0005  B90500          MOV   CX,0005
1067:0008  BB0000          MOV   BX,0000
1067:000D  0207      ADD   AL,[ BX]
1067:000F  43              INC   BX
1067:0010  49              DEC   CX
1067:0011  75FA      JNZ   000D
1067:0013  A20500          MOV   [ 0005] ,AL
1067:0016  B44C      MOV   AH,4C
1067:0018  CD21      INT   21
```

- The instruction "JNZ AGAIN" was assembled as "JNZ 000D", and 000D is the address of the instruction with the label AGAIN. The instruction "JNZ 000D" has the opcode 75 and the target address FA, which is located at offset addresses 0011 and 0012.



- This is followed by "MOV SUM, AL", which is located beginning at offset address 0013. The IP value of this MOV (0013), is added to FA to calculate the address of label AGAIN (0013+ FA= 000D) and the carry is dropped.
- In reality, FA is the 2's complement of -6, meaning that the address of the target is -6 bytes from the IP of the next instruction.
- Similarly, the target address for a forward jump is calculated by adding the IP of the following instruction to the operand. In that case the displacement value is positive, as shown next.



```
0005      8A 47 02 AGAIN:   MOV    AL,[ BX] +2
0008      3C 61             CMP    AL,61H
000A      72 06             JB     NEXT
000C      3C 7A             CMP    AL,7AH
000E      77 02             JA     NEXT
0010      24 DF             AND    AL,ODFH
0012      88 04    NEXT:    MOV    [ SI] ,AL
```

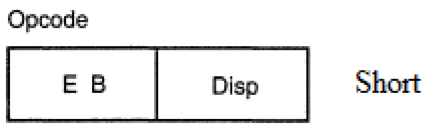
- In the program above, "JB NEXT" has the opcode 72 and the target address 06 and is located at IP = 000A and 000B.
- The jump will be 6 bytes from the next instruction, which is IP = 000C. Adding gives us 000CH + 0006H = 0012H, which is the exact address of the NEXT label.
- Look also at "JA NEXT", which has 77 and 02 for the opcode and displacement, respectively. The IP of the following instruction, 0010, is added to 02 to get 0012, the address of the target location.

Note that, regardless of whether the jump is forward or backward, for conditional jumps, the address of the target address can never be more than –128 to +127 bytes away from the IP associated with the instruction following the jump lf any attempt is made to violate this rule, the assembler will generate a "relative jump out of range" message. These conditional jumps are sometimes referred to as SHORT jumps.

Unconditional Jumps:

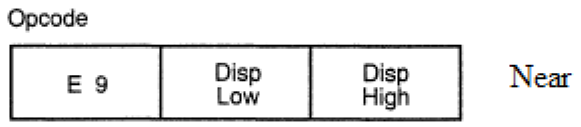
"JMP label" is an unconditional jump in which control is transferred unconditionally to the target location label. The unconditional jump can take the following forms:

1. SHORT JUMP – which is specified by the format "JMP SHORT label". This is a jump in which the address of the target location is within –128 to +127 bytes of memory relative to the address of the current IP.
   - ✓ In this case, the opcode is EB and the operand is 1 byte in the range 00 to FF. The operand byte is added to the current IP to calculate the target address. If the jump is backward, the operand is in 2's complement. This is exactly like the J condition case.
   - ✓ Coding the directive "short" makes the jump more efficient; i.e., it will be assembled into a 2-byte instruction instead of a 3-byte instruction.

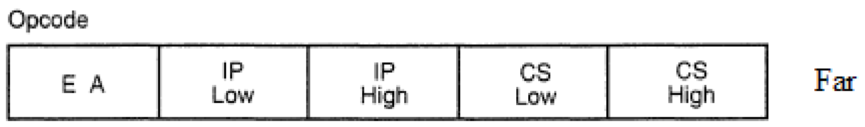


2. NEAR JUMP, which is the default, has the format "JNP label". This is a near jump (within the current code segment) and has the opcode E9. The target address can be any of the addressing modes of direct, register, register indirect, or memory indirect:
   - ✓ (a) Direct JUMP: is exactly like the short jump explained earlier, except that the target address can be anywhere in the segment within the range +32767 to –32768 of the current IP.
   - ✓ (b) Register indirect JUMP: the target address is in a register. For example, in "JMP BX", IP takes the value BX.

- ✓ (c) Memory indirect JMP: the target address is the contents of two memory locations pointed at by the register. Example: "JMP [DI]" will replace the IP with the contents of memory locations pointed at by DI and DI + 1.

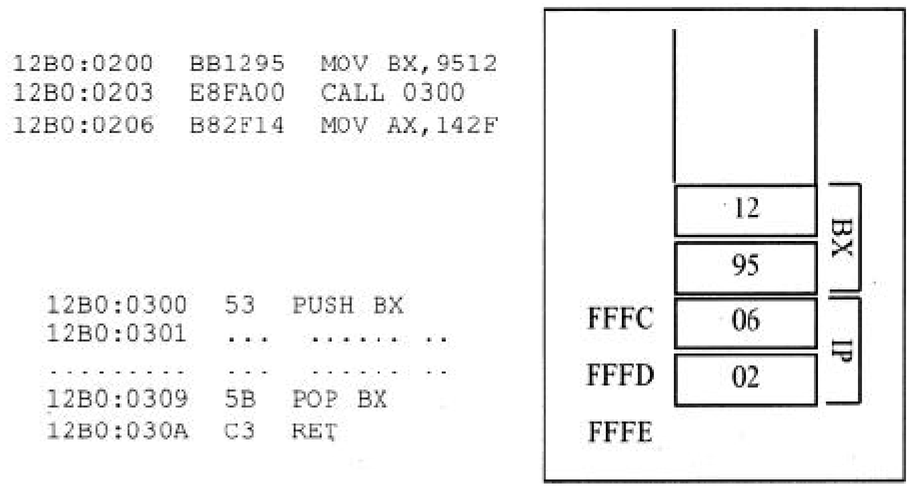


3. FAR JUMP, which has the format "JMP FAR PTR label". This is a jump out of the current code segment, meaning that not only the IP but also the CS is replaced with new values.



CALL Statement:

- Another control transfer instruction is the CALL instruction, which is used to call a procedure. CALLs to procedures are used to perform tasks that need to be performed frequently. This makes a program more structured.
- The target address could be in the current segment, in which case it will be a NEAR call or outside the current CS segment, which is a FAR call.
- To make sure that after execution of the called subroutine the microprocessor knows where to come back, the microprocessor automatically saves the address of the instruction following the call on the stack. It must be noted that in the NEAR call only the IP is saved on the stack, and in a FAR call both CS and IP are saved.
- When a subroutine is called, control is transferred to that subroutine and the processor saves the IP (and CS in the case of a FAR call) and begins to fetch instructions from the new location.
- After finishing execution of the subroutine, for control to be transferred back to the caller, the last instruction in the called subroutine must be RET (return). The RET instruction in the case of NEAR and FAR is different. For NEAR calls, the IP is restored; for FAR calls, both CS and IP are restored.
- This will ensure that control is given back to the caller. As an example, assume that SP = FFFEH and the following code is a portion of the program unassembled in DEBUG:



```
12B0:0200   BB1295   MOV BX,9512
12B0:0203   E8FA00   CALL 0300
12B0:0206   B82F14   MOV AX,142F


12B0:0300   53   PUSH BX
12B0:0301   ...   ...... ..
.........   ...   ...... ..
12B0:0309   5B   POP BX
12B0:030A   C3   RET
```

Fig: IP in the Stack

Since the CALL instruction is a NEAR call, (different IP, same CS), only IP is saved on the stack. In this case, the IP address of the instruction after the call is saved on the stack as shown in above Fig. This IP will be 0206, which belongs to the "MOV AX, 142F" instruction.

The last instruction of the called subroutine must be a RET instruction that directs the CPU to POP the top 2 bytes of the stack into the IP and resume executing at offset address 0206. For this reason, the number of PUSH and POP instructions (which alter the SP) must match. In other words, for every PUSH there must be a POP.

Assembly Language Subroutines:

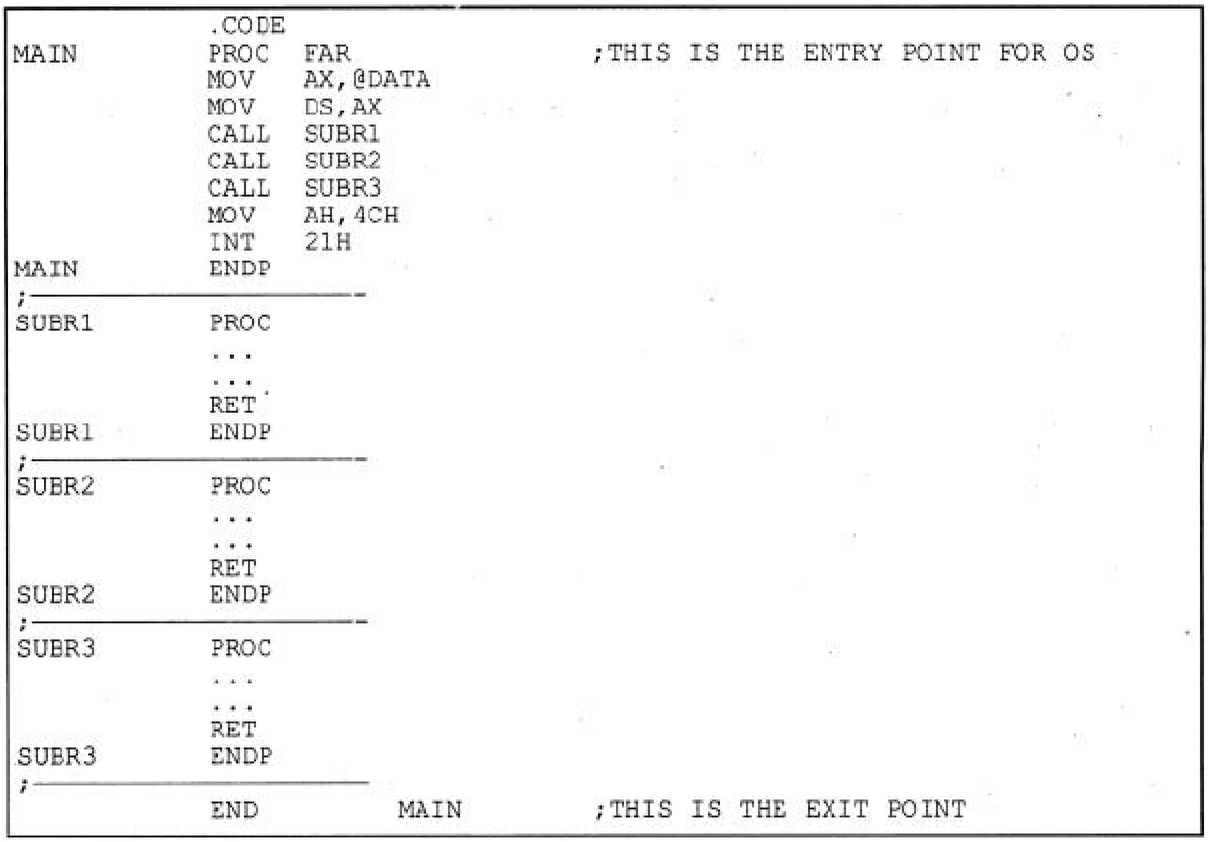
In Assembly language programming it is common to have one main program and many subroutines to be called from the main program. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together.

The main program is the entry point from the OS and is FAR, as explained earlier, but the subroutines called within the main program can be FAR or NEAR. Remember that NEAR routines are in the same code segment, while FAR routines are outside the current code segment. If there is no specific mention of FAR after the directive PROC, by default, it will be NEAR, as shown in the following Fig.

Rules for Names in Assembly Language:

- ✓ By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow.
- ✓ Each label name must be unique.
- ✓ The names used for labels in Assembly language programming consist of alphabetic letters in both upper- and lowercase, the digits 0 through 9, and the special characters question mark(?), period(.), at(@), under line(_), and dollar sign ($).

- ✓ The first character of the name must be an alphabetic character or special character. It cannot be a digit.
- ✓ Names may be up to 31 characters long.



```
                .CODE
MAIN            PROC   FAR                ;THIS IS THE ENTRY POINT FOR OS
                MOV    AX,@DATA
                MOV    DS,AX
                CALL   SUBR1
                CALL   SUBR2
                CALL   SUBR3
                MOV    AH,4CH
                INT    21H
MAIN            ENDP
;------------------------------------------
SUBR1           PROC
                ...
                ...
                RET
SUBR1           ENDP
;------------------------------------------
SUBR2           PROC
                ...
                ...
                RET
SUBR2           ENDP
;------------------------------------------
SUBR3           PROC
                ...
                ...
                RET
SUBR3           ENDP
;------------------------------------------
                END          MAIN         ;THIS IS THE EXIT POINT
```

Fig: Shell of Assembly Language Subroutines

DATA TYPES AND DATA DEFINITIONS:

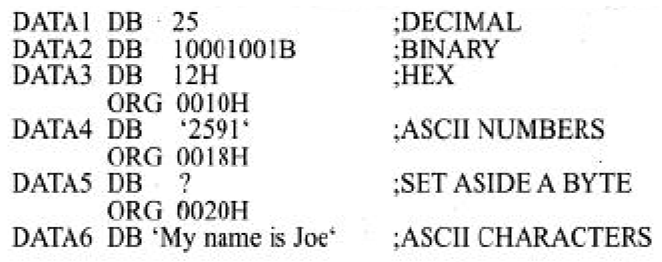
- The assembler supports all the various data types of the x86 microprocessor by providing data directives that define the data types and set aside memory for them.
- The 8088/86 microprocessor supports many data types, but none are longer than 16 bits wide since the size of the registers is 16 bits. It is the job of the programmer to break down data larger than 16 bits (0000 to FFFFH, or 0 to 65535 in decimal) to be processed by the CPU.
- The data types used by the 8088/86 can be 8-bit or 16-bit, positive or negative. If a number is less than 8 bits wide, it still must be coded as an 8-bit register with the higher digits as zero. Similarly, if the number is less than 16 bits wide it must use all 16 bits, with the rest being 0s.
- For example, the number 5 is only 3 bits wide (101) in binary, but the 8088/86 will accept it as 05 or "0000 0101" in binary. The number 514 is "10 0000 0010" in binary, but the 8088/86 will accept it as "0000 0010 0000 0010" in binary.

Assembler Data Directive:

The following are some of the data directives used by the x86 microprocessor and supported by all software vendors.

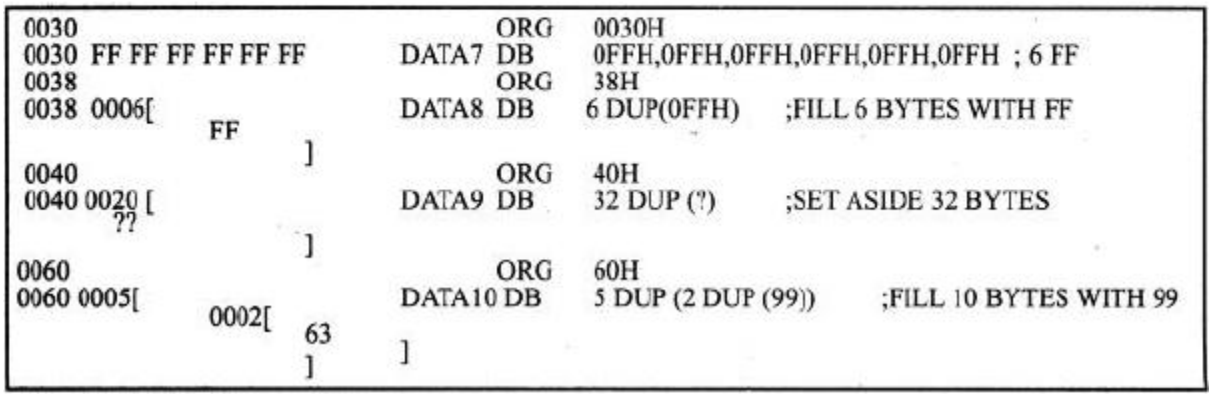
- **ORG** (origin) – is used to indicate the beginning of the offset address. The number that comes after ORG can be either in hex or in decimal. If the number is not followed by H, it is decimal and the assembler will convert it to hex.

- **DB** (define byte) – directive allows allocation of memory in byte-sized chunks. This is indeed the smallest allocation unit permitted. DB can be used to define numbers in decimal, binary, hex, and ASCII. For decimal, the D after the decimal number is optional, but using B (binary) and H (hexa- decimal) for the others is required. Regardless of which one is used, the assembler will convert numbers into hex. To indicate ASCII, simply place the string in single quotation marks ('like this'). Either single or double quotes can be used around ASCII strings.



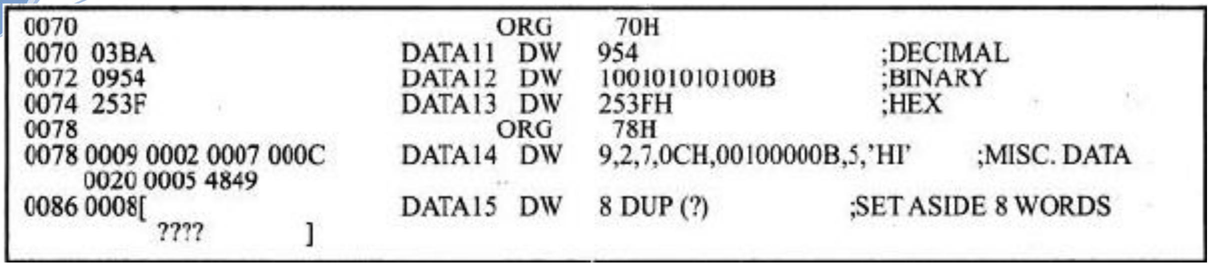
```
DATA1  DB   25                  ;DECIMAL
DATA2  DB   10001001B           ;BINARY
DATA3  DB   12H                 ;HEX
       ORG  0010H
DATA4  DB    '2591'             ;ASCII NUMBERS
       ORG  0018H
DATA5  DB    ?                  ;SET ASIDE A BYTE
       ORG  0020H
DATA6  DB 'My name is Joe'      ;ASCII CHARACTERS
```

- **DUP** (duplicate) – is used to duplicate a given number of characters. This can avoid a lot of typing. For example, contrast the following two methods of filling six memory locations with FFH:



```
0030                               ORG   0030H
0030  FF FF FF FF FF FF      DATA7 DB    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH  ; 6 FF
0038                               ORG   38H
0038  0006[                  DATA8 DB    6 DUP(0FFH)    ;FILL 6 BYTES WITH FF
            FF
                      ]
0040                               ORG   40H
0040 0020 [                  DATA9 DB    32 DUP (?)     ;SET ASIDE 32 BYTES
        ??
                      ]
0060                               ORG   60H
0060 0005[                   DATA10 DB   5 DUP (2 DUP (99))     ;FILL 10 BYTES WITH 99
            0002[
                      63
                      ]
                          ]
```

- **DW** (define word) – is used to allocate memory 2 bytes (one word) at a time. The following are some examples of DW:



```
0070                              ORG     70H
0070  03BA                 DATA11  DW     954                  ;DECIMAL
0072  0954                 DATA12  DW     100101010100B        ;BINARY
0074  253F                 DATA13  DW     253FH                ;HEX
0078                              ORG     78H
0078 0009 0002 0007 000C   DATA14  DW     9,2,7,0CH,00100000B,5,'HI'   ;MISC. DATA
     0020 0005 4849
0086 0008[                 DATA15  DW     8 DUP (?)          ;SET ASIDE 8 WORDS
          ????          ]
```

⇨ **EQU** (equate) – is used to define a constant without occupying a memory location. EQU does not set aside storage for a data item but associates a constant value with a data label so that when the label appears in the program; its constant value will be substituted for the label.

- EQU can also be used outside the data segment, even in the middle of a code segment.

Using EQU for the counter constant in the immediate addressing mode:

COUNT EQU 25	*COUNT DB 25*
When executing the instructions "MOV CX, COUNT", the register CX will be loaded with the value 25.	When executing the same instruction "MOV CX, COUNT" it will be in the direct addressing mode.

What is the real advantage of EQU? First, note that EQU can also be used in the data segment:

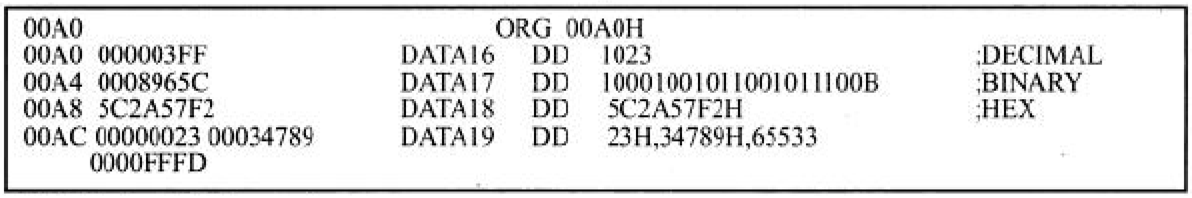
COUNT EQU 25

COUNTER1 DB COUNT

COUNTER2 DB COUNT

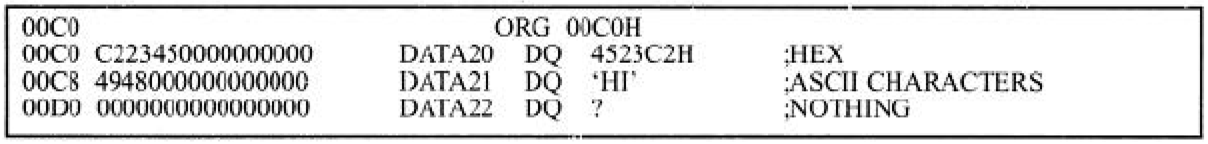
Assume that there is a constant (a fixed value) used in many different places in the data and code segments. By the use of EQU, one can change it once and the assembler will change all of them, rather than making the programmer tries to find every location and correct it.

⇨ **DD** (define double word) – directive is used to allocate memory locations that are 4 bytes (two words) in size. Again, the data can be in decimal, binary, or hex. In any case the data is converted to hex and placed in memory locations according to the rule of low byte to low address and high byte to high address. DD examples are:



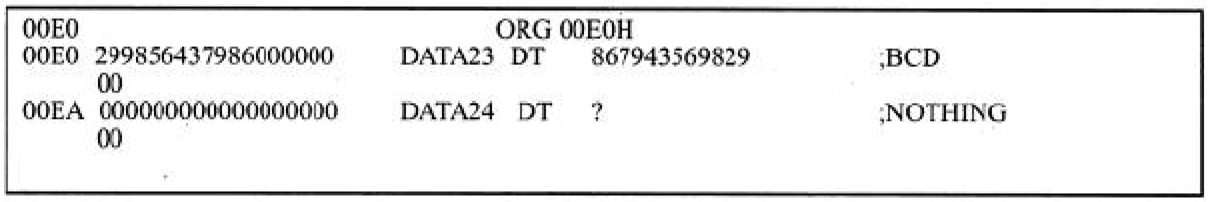
```
00A0                             ORG  00A0H
00A0  000003FF          DATA16    DD    1023                          ;DECIMAL
00A4  0008965C          DATA17    DD    10001001011001011100B         ;BINARY
00A8  5C2A57F2          DATA18    DD    5C2A57F2H                     ;HEX
00AC 00000023 00034789  DATA19    DD    23H,34789H,65533
      0000FFFD
```

⇨ **DQ** (define quad word) – is used to allocate memory 8 bytes (four words) in size. This can be used to represent any variable up to 64 bits wide:



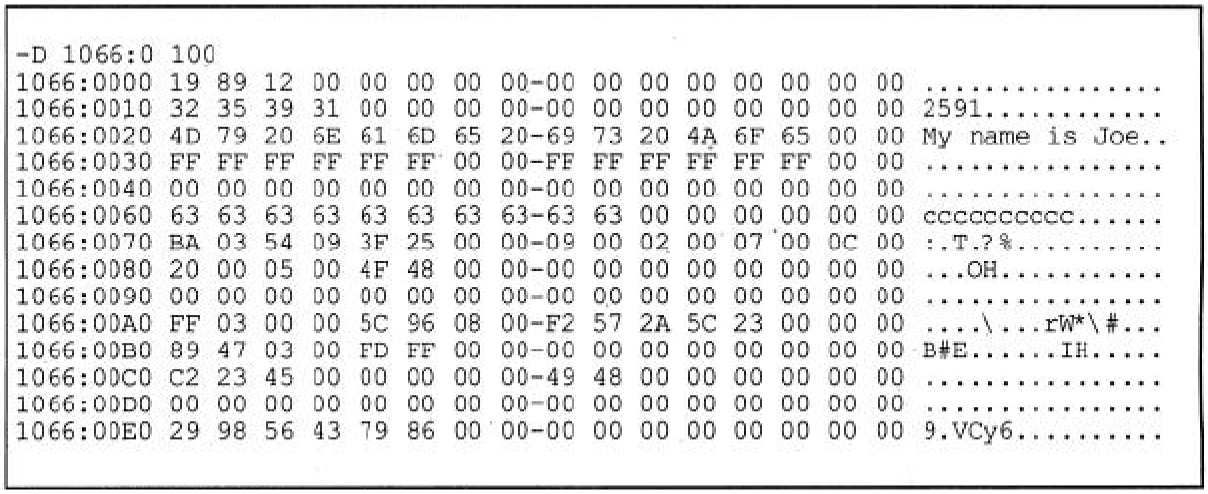
```
00C0                             ORG  00C0H
00C0  C223450000000000   DATA20   DQ   4523C2H       ;HEX
00C8  4948000000000000   DATA21   DQ   'HI'          ;ASCII CHARACTERS
00D0  0000000000000000   DATA22   DQ   ?             ;NOTHING
```

⇨ **DT** (define ten bytes) – is used for memory allocation of packed BCD numbers. The application of DT will be seen in the multibyte addition of BCD numbers. For now, observe how they are located in memory. Notice that the "H" after the data is not needed. This directive allocates 10 bytes, but a maximum of 18 digits can be entered.



```
00E0                                   ORG 00E0H
00E0  299856437986000000      DATA23  DT     867943569829          ;BCD
      00
00EA  000000000000000000      DATA24  DT     ?                     ;NOTHING
      00
```

It is essential to understand the way operands are stored in memory. The following Fig shows the memory dump of the data section, including all the examples discussed here.



```
-D 1066:0 100
1066:0000 19 89 12 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0010 32 35 39 31 00 00 00 00-00 00 00 00 00 00 00 00 2591............
1066:0020 4D 79 20 6E 61 6D 65 20-69 73 20 4A 6F 65 00 00 My name is Joe..
1066:0030 FF FF FF FF FF FF 00 00-FF FF FF FF FF FF 00 00 ................
1066:0040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:0060 63 63 63 63 63 63 63 63-63 63 00 00 00 00 00 00 cccccccccc......
1066:0070 BA 03 54 09 3F 25 00 00-09 00 02 00 07 00 0C 00 :.T.?%..........
1066:0080 20 00 05 00 4F 48 00 00-00 00 00 00 00 00 00 00 ...OH...........
1066:0090 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:00A0 FF 03 00 00 5C 96 08 00-F2 57 2A 5C 23 00 00 00 ....\...rW*\#...
1066:00B0 89 47 03 00 FD FF 00 00-00 00 00 00 00 00 00 00 B#E......IH.....
1066:00C0 C2 23 45 00 00 00 00 00-49 48 00 00 00 00 00 00 ................
1066:00D0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
1066:00E0 29 98 56 43 79 86 00 00-00 00 00 00 00 00 00 00 9.VCy6..........
```

Looking at the memory dump shows that, all of the data directives use the little endian format for storing data (the least significant byte is located in the memory location of the lower address and the most significant byte resides in the memory location of the higher address).

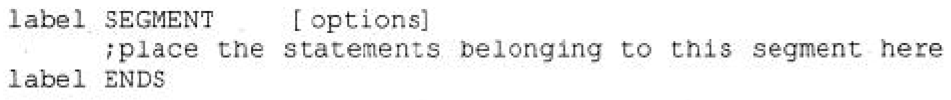
For example, look at the case of "*DATA20 DQ 4523C2*", residing in memory starting at offset 00C0H. C2, the least significant byte, is in location 00C0, with 23 in 00C1, and 45, the most significant byte, in 00C2. It must also be noted that for ASCII data, only the DB directive can be used to define data of any length, and the use of DO, DQ, or DT directive for ASCII strings of more than 2 bytes gives an assembly error. When DB is used for ASCII numbers, notice how it places them backwards in memory. For example, see *"DATA4 DB '2591' "* at origin 10H: 32, ASCII for 2, is in memory location 10H; 35, ASCII for 5, is in 11H; and so on.

FULL SEGMENT DEFINITION:

The way that segments have been defined in the programs above is a newer definition referred to as *simple segment definition*. It is supported by Microsoft's MASM 5.0 and higher and/or Borland's TASM version 1 and higher. The older, more traditional definition is called the *full segment definition*.

Segment Definition:

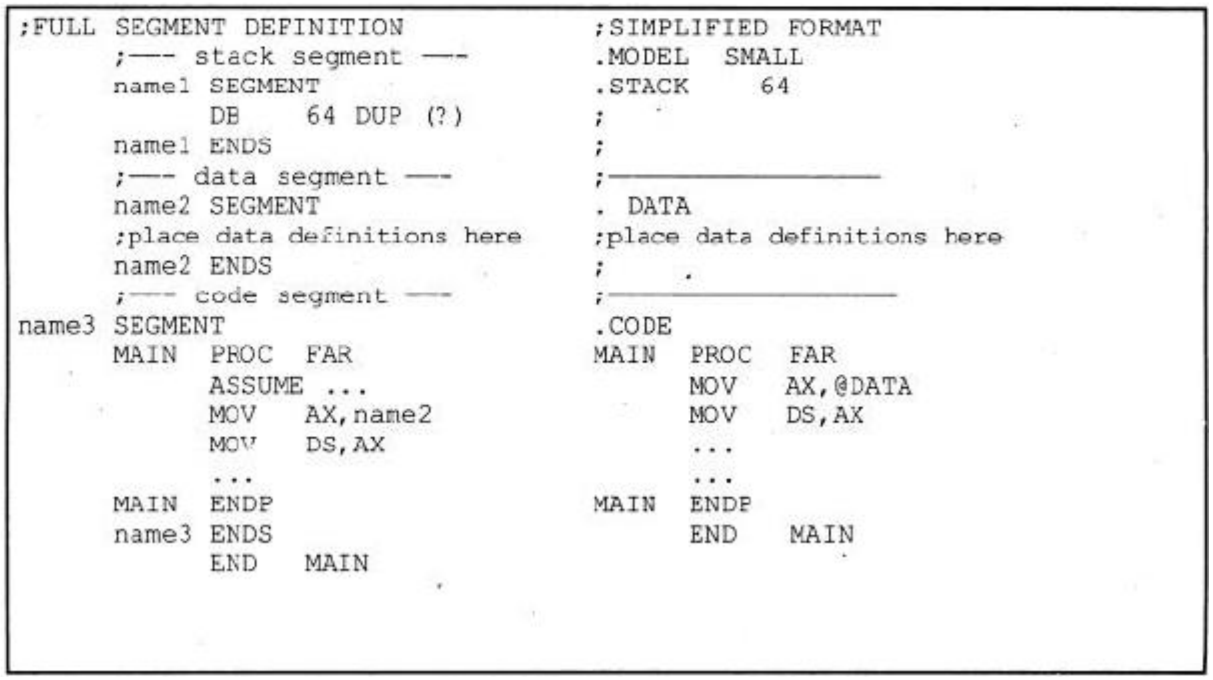
- ✓ In the full segment definition, the ".MODEL" directive is not used. Further, the directives " .STACK",".DATA", and" .CODE" are replaced by SEGMENT and ENDS directives that surround each segment.
- ✓ The SEGMENT and the ENDS directives indicate to the assembler the beginning and ending of a segment and have the following format:



```
label SEGMENT     [options]
      ;place the statements belonging to this segment here
label ENDS
```

- ✓ The label, or name, must follow naming conventions and must be unique.
- ✓ The [options] field gives important information to the assembler for organizing the segment, but is not required.
- ✓ The ENDS label must be the same label as in the SEGMENT directive.

The following Fig shows the full segment definition and simplified format, side by side.



```
;FULL SEGMENT DEFINITION                 ;SIMPLIFIED FORMAT
      ;---- stack segment ----           .MODEL  SMALL
      name1 SEGMENT                      .STACK    64
            DB     64 DUP  (?)           ;
      name1 ENDS                         ;
      ;---- data segment ----            ;------------------
      name2 SEGMENT                      . DATA
      ;place data definitions here       ;place data definitions here
      name2 ENDS                         ;
      ;---- code segment ----            ;-------------------
name3 SEGMENT                            .CODE
      MAIN  PROC   FAR                   MAIN  PROC   FAR
            ASSUME ...                         MOV    AX,@DATA
            MOV    AX,name2                    MOV    DS,AX
            MOV    DS,AX                       ...
            ...                                ...
      MAIN  ENDP                         MAIN  ENDP
      name3 ENDS                               END    MAIN
            END    MAIN
```

Stack Segment Definition:

The stack segment shown below contains the line: *"DB 64 DUP (?)"* to reserve 64 bytes of memory for the stack. The following three lines in full segment definition are comparable to "**.**STACK 64" in simple definition:

```
STSEG  SEGMENT                ;the "SEGMENT" directive begins the segment
       DB 64 DUP (?)          ;this segment contains only one line
STSEG  ENDS                   ;the "ENDS" segment ends the segment
```

Data Segment Definition:

In full segment definition, the SEGMENT directive names the data segment and must appear before the data. The ENDS segment marks the end of the data segment:

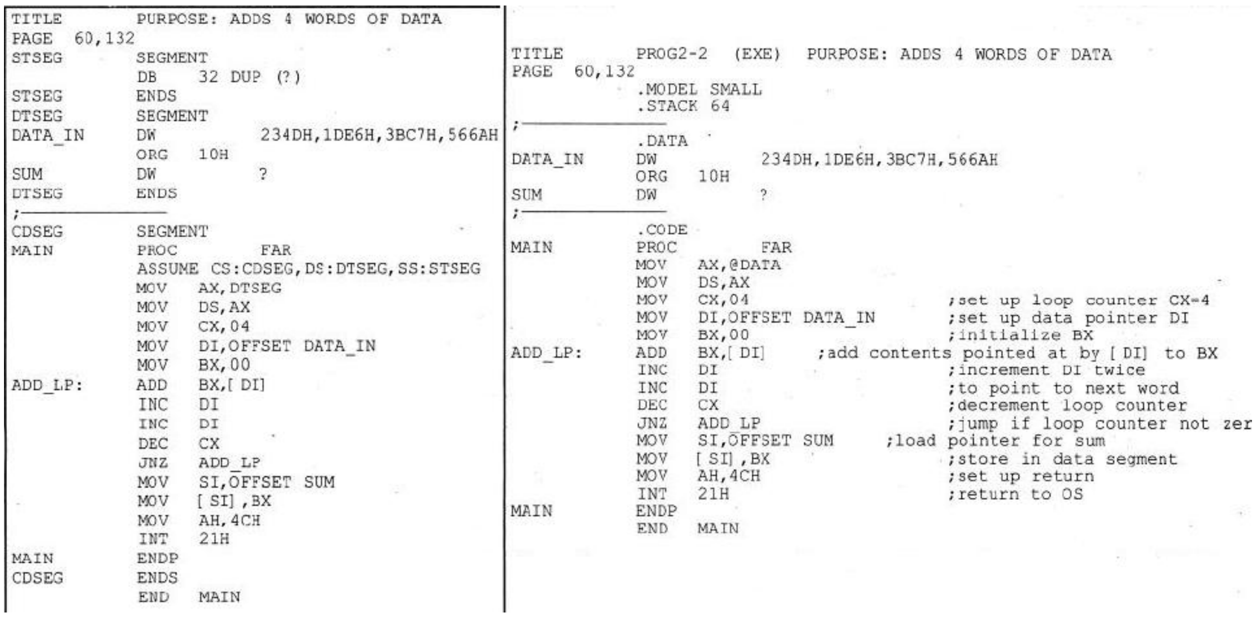
```
DTSEG      SEGMENT    ;the SEGMENT directive begins the segment
           ;define your data here
DTSEG      ENDS       ;the ENDS segment ends the segment
```

Code Segment Definition:

The code segment also begins and ends with SEGMENT and ENDS directives:

```
CDSSEG     SEGMENT    ;the SEGMENT directive begins the segment
           ;your code is here
CDSEG      ENDS       ;the ENDS segment ends the segment
```

Example:



```
TITLE       PURPOSE: ADDS 4 WORDS OF DATA
PAGE  60,132
STSEG       SEGMENT
            DB    32 DUP (?)
STSEG       ENDS
DTSEG       SEGMENT
DATA_IN     DW          234DH,1DE6H,3BC7H,566AH
            ORG   10H
SUM         DW          ?
DTSEG       ENDS
;-----------------
CDSEG       SEGMENT
MAIN        PROC        FAR
            ASSUME CS:CDSEG,DS:DTSEG,SS:STSEG
            MOV   AX,DTSEG
            MOV   DS,AX
            MOV   CX,04
            MOV   DI,OFFSET DATA_IN
            MOV   BX,00
ADD_LP:     ADD   BX,[DI]
            INC   DI
            INC   DI
            DEC   CX
            JNZ   ADD_LP
            MOV   SI,OFFSET SUM
            MOV   [SI],BX
            MOV   AH,4CH
            INT   21H
MAIN        ENDP
CDSEG       ENDS
            END   MAIN
```

```
TITLE       PROG2-2  (EXE)  PURPOSE: ADDS 4 WORDS OF DATA
PAGE  60,132
            .MODEL SMALL
            .STACK 64
;-----------------
            .DATA
DATA_IN     DW          234DH,1DE6H,3BC7H,566AH
            ORG   10H
SUM         DW          ?
;-----------------
            .CODE
MAIN        PROC        FAR
            MOV   AX,@DATA
            MOV   DS,AX
            MOV   CX,04                      ;set up loop counter CX=4
            MOV   DI,OFFSET DATA_IN          ;set up data pointer DI
            MOV   BX,00                      ;initialize BX
ADD_LP:     ADD   BX,[DI]      ;add contents pointed at by [DI] to BX
            INC   DI                         ;increment DI twice
            INC   DI                         ;to point to next word
            DEC   CX                         ;decrement loop counter
            JNZ   ADD_LP                     ;jump if loop counter not zer
            MOV   SI,OFFSET SUM        ;load pointer for sum
            MOV   [SI],BX                    ;store in data segment
            MOV   AH,4CH                     ;set up return
            INT   21H                        ;return to OS
MAIN        ENDP
            END   MAIN
```

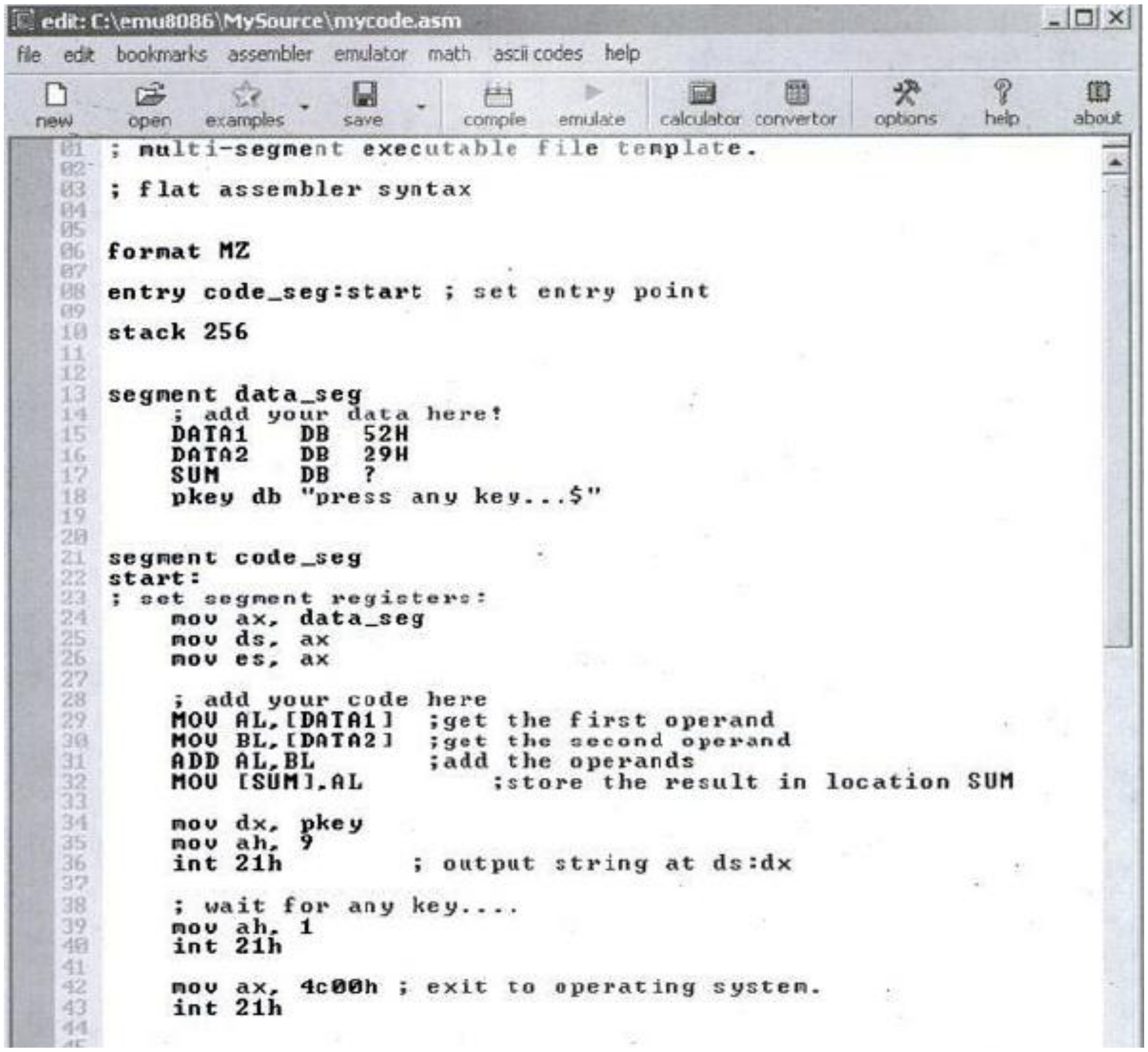
Fig: Program 2-2, Rewritten with Full Segment Definition

- ✓ In full segment definition, immediately after the PROC directive is the ASSUME directive, which associates segment registers with specific segments by assuming that the segment register is equal to the segment labels used in the program.

- ✓ If an extra segment had been used, ES would also be included in the ASSUME statement.
- ✓ The ASSUME statement is needed because a given Assembly language program can have several code segments; one or two or three or more data segments and more than one stack segment. But only one of each can be addressed by the CPU at a given time; since, only one of each of the segment registers available inside the CPU.
- ✓ ASSUME tells the assembler which of the segments defined by the SEGMENT directives should be used.

Using the emu8086 Assembler:

There is a simple and popular assembler called emu8086; that one can use for assembling the 8086 Assembly language programs. It is available from the www.emu8086.com website. Examine the following Fig for screenshots using emu8086.


edit: C:\emu8086\MySource\mycode.asm
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about

```
; multi-segment executable file template.

; flat assembler syntax


format MZ

entry code_seg:start ; set entry point

stack 256


segment data_seg
     ; add your data here!
     DATA1    DB   52H
     DATA2    DB   29H
     SUM      DB   ?
     pkey db "press any key...$"


segment code_seg
start:
; set segment registers:
     mov ax, data_seg
     mov ds, ax
     mov es, ax

     ; add your code here
     MOV AL,[DATA1]   ;get the first operand
     MOV BL,[DATA2]   ;get the second operand
     ADD AL,BL        ;add the operands
     MOV [SUM],AL         ;store the result in location SUM

     mov dx, pkey
     mov ah, 9
     int 21h         ; output string at ds:dx

     ; wait for any key....
     mov ah, 1
     int 21h

     mov ax, 4c00h ; exit to operating system.
     int 21h
```

Fig: emu8086

NOTE: emu8086 requires putting brackets around variables, unlike MASM/TASM.

EXE vs COM Files:

All program examples so far were designed to be assembled and linked into EXE files. The COM file, similar to the EXE file, contains the executable machine code and can be run at the OS level.

Why COM Files?

- ✓ The EXE file can be of any size. Due to limited amount of memory, one needs to have very compact code in the form of COM file.
- ✓ COM files are used because of their compactness, since they cannot be greater than 64K bytes. The reason for the 64K-byte limit is that the COM file must fit into a single segment, and since in the x86 the size of a segment is 64K bytes, the COM file cannot be larger than 64K.
- ✓ To limit the size of the file to 64K bytes requires defining the data inside the code segment and also using an area (the end area) of the code segment for the stack.

Table: EXE vs. COM File Format

EXE File	COM File
1. Unlimited size	1. Maximum size 64K bytes
2. Stack segment is defined	2. No stack segment definition
3. Data segment is defined	3. Data segment is defined in code segment
4. Larger file (takes more memory)	4. Smaller file (takes less memory)
5. Header block (contains information such as size, address location in memory, and stack address of the EXE module), which occupies 512 bytes of memory precedes every EXE file	5. Does not have a header file

FLOWCHARTS AND PSEUDOCODE:

Structured programming is a term used to denote programming techniques that can make a program easier to code, debug, and maintain over time. There are certain principles that every structured program should follow. Some of these are as follows:

1. The program should be designed before it is coded. By using techniques of flowcharting or pseudocode, the design of the program is clear to the person coding it, as well as to those who will maintain the program later.

2. Using comments within the program and documentation accompanying the program also will help someone else to know what the program does. It may even help the programmer who wrote the program remember how it worked years later!
3. The main routine should consist of calls to subroutines that perform the work of the program. This is sometimes called top-down programming. Use subroutines to accomplish tasks that are repeated. This saves time in coding and also makes the program easier to read.
4. Data control is very important. It can be very frustrating and time consuming to track through a long program to find where a variable was changed. First of all, the programmer should document the purpose of each variable, and which subroutines might alter its value. Further, each subroutine should document its input and output variables, and which input variables might be altered within it.

Flow Charts & Pseudocode:

Flowcharts use graphic symbols to represent different types of program operations. These symbols are connected together into a flowchart to show the flow of execution of the program.

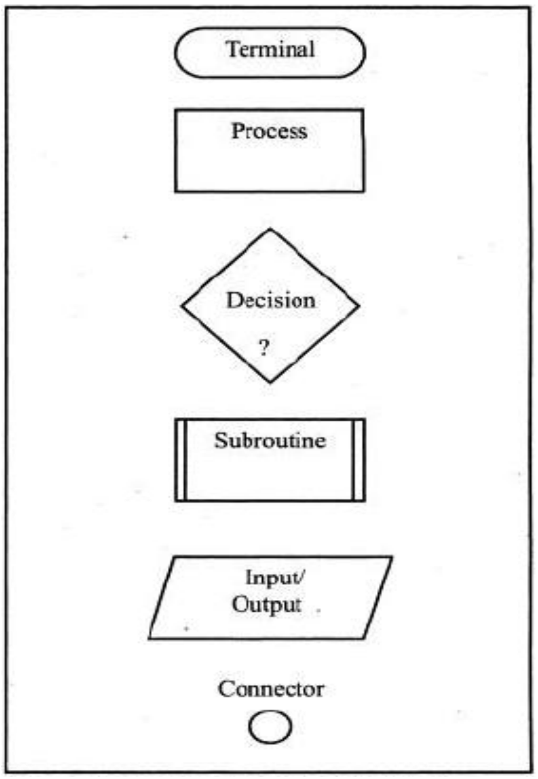


Fig: Commonly used Flowchart Symbols

The limitations of flowchart are –

- ✓ We can’t write much in the little boxes
- ✓ We can’t get the clear picture of the program without getting bogged down in the details.

An alternative to using flowchart is pseudocode, which involves writing brief descriptions of the flow of the code.

Control Structures:

Structured programming used three basic types of program control structures –

1. Sequence

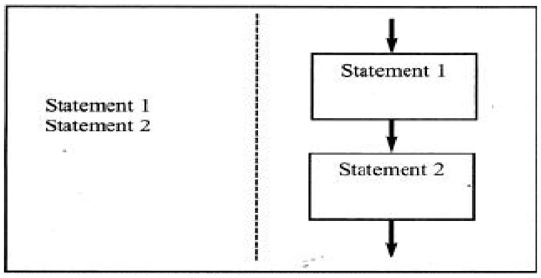


Fig: SEQUENCE Pseudocode vs Flowchart

2. Control

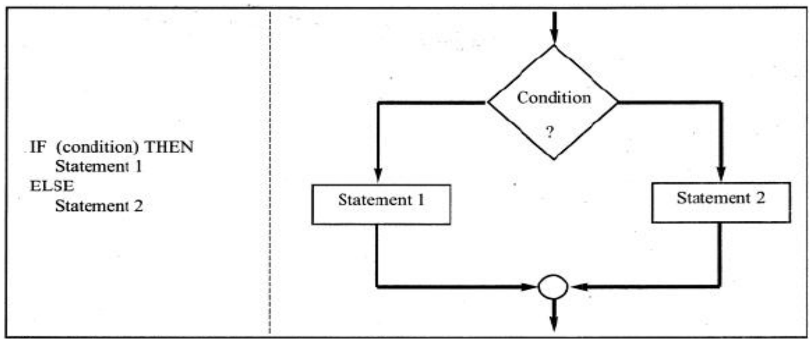


Fig: IF-THEN-ELSE Pseudocode vs Flowchart

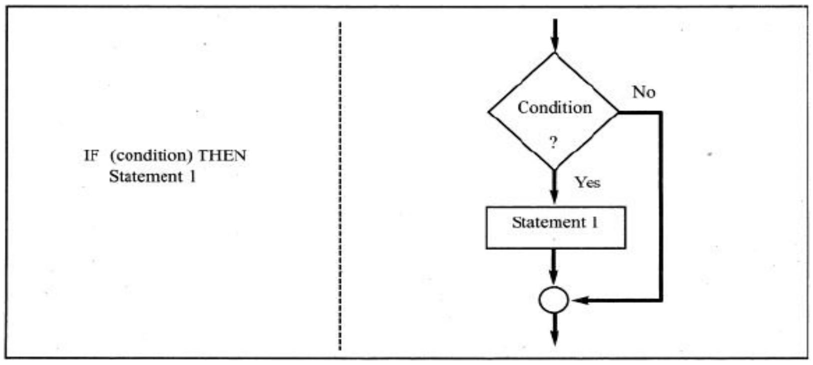


Fig: IF-THEN Pseudocode vs Flowchart

3. Iteration

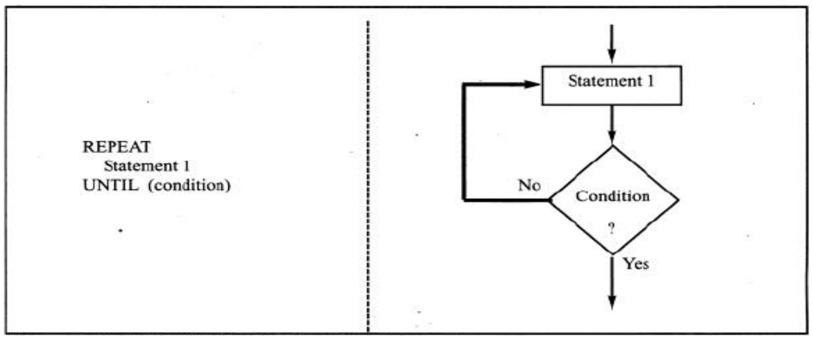


Fig: REPEAT-UNTIL Pseudocode vs Flowchart

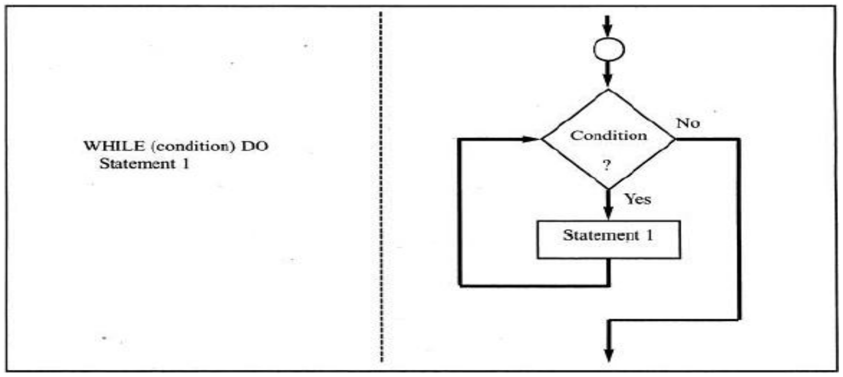


Fig: WHILE-DO Pseudocode vs Flowchart

The purpose of flowchart or pseudocode is to show the flow of the program and what the program does; not the specific Assembly language instructions.

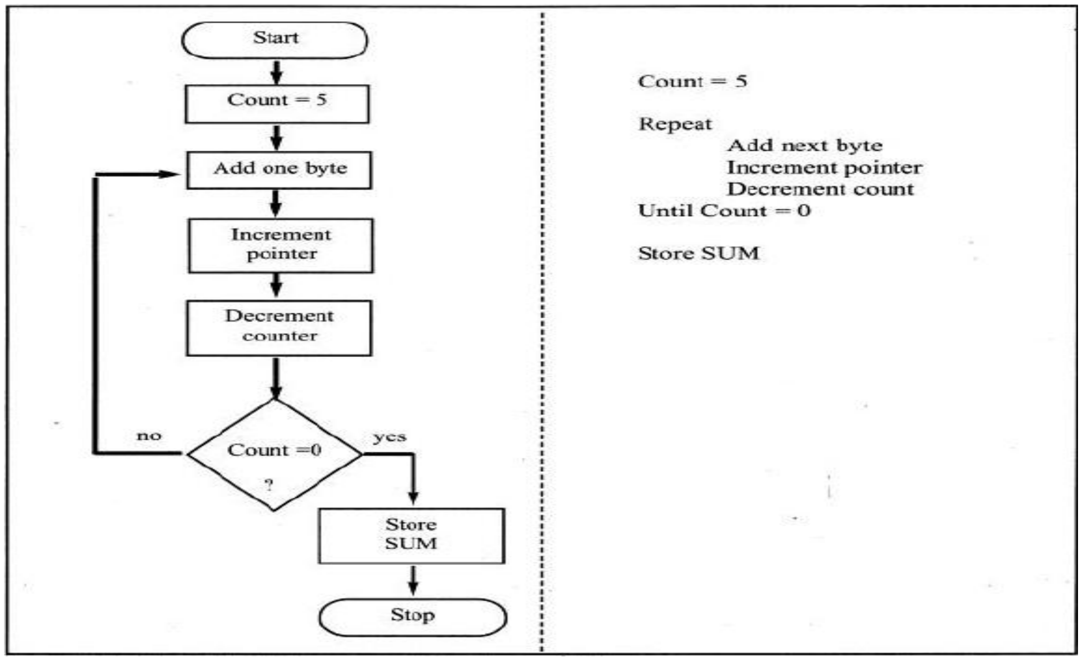


Fig: Flowchart vs Pseudocode for Program 2-1

By: MAHESH PRASANNA K.,

DEPT. OF CSE, VCET.

________________*********________________

****

MODULE – 2

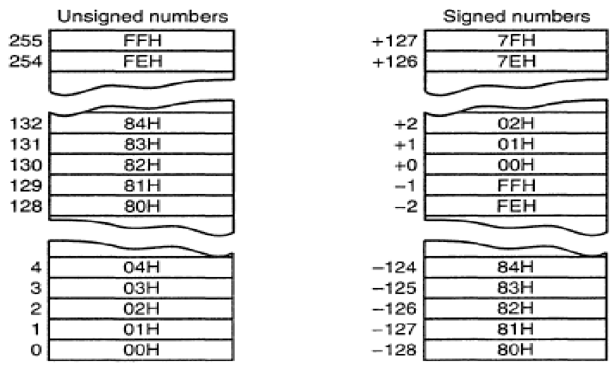
A AND L INSTRUCTIONS & INT 21H AND INT 10H PROGRAMMING

ARITHMETIC & LOGIC INSTRUCTIONS AND PROGRAMS

INTRUCTIONS SET DESCRIPTION:

UNSIGNED ADDITION AND SUBTRACTION:

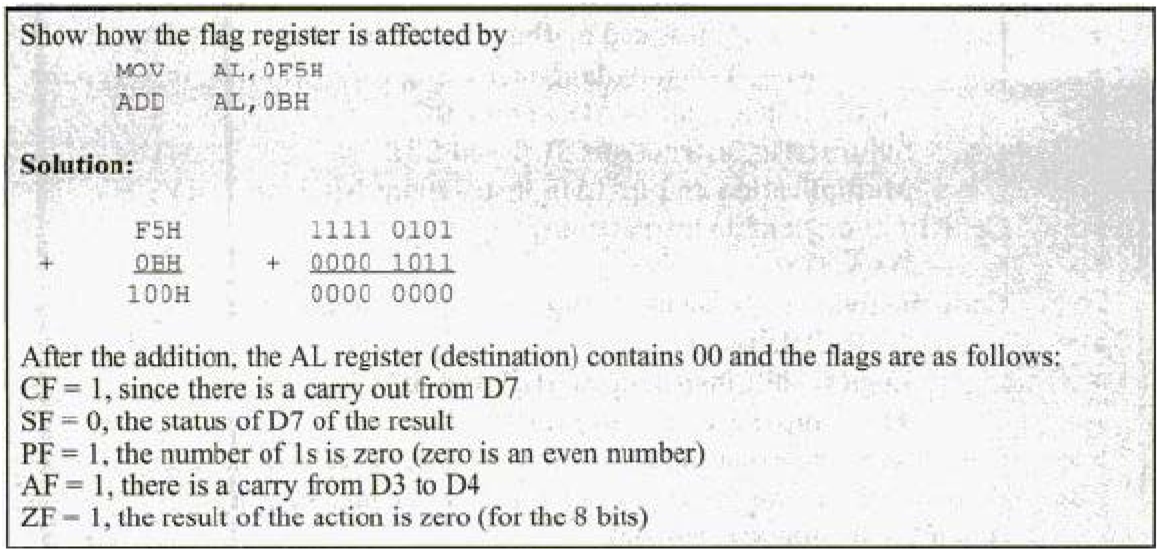
Unsigned numbers are defined as data in which all the bits are used to represent data and no bits are set aside for the positive or negative sign. This means that the operand can be between 00 and FFH (0 to 255 decimal) for 8-bit data, and between 0000 and FFFFH (0 to 65535 decimal) for 16-bit data.



Addition of Unsigned Numbers:

```
ADD destination,source ;destination = destination + source
```

- ✓ The instructions ADD and ADC are used to add two operands. The destination operand can be a register or in memory. The source operand can be a register, in memory, or immediate.
- ✓ Remember that memory-to-memory operations are never allowed in x86 Assembly language.
- ✓ The instruction could change any of the ZF, SF, AF, CF, or PF bits of the flag register, depending on the operands involved. The overflow flag is used only in signed number operations.



Show how the flag register is affected by

```
MOV   AL,0F5H
ADD   AL,0BH
```

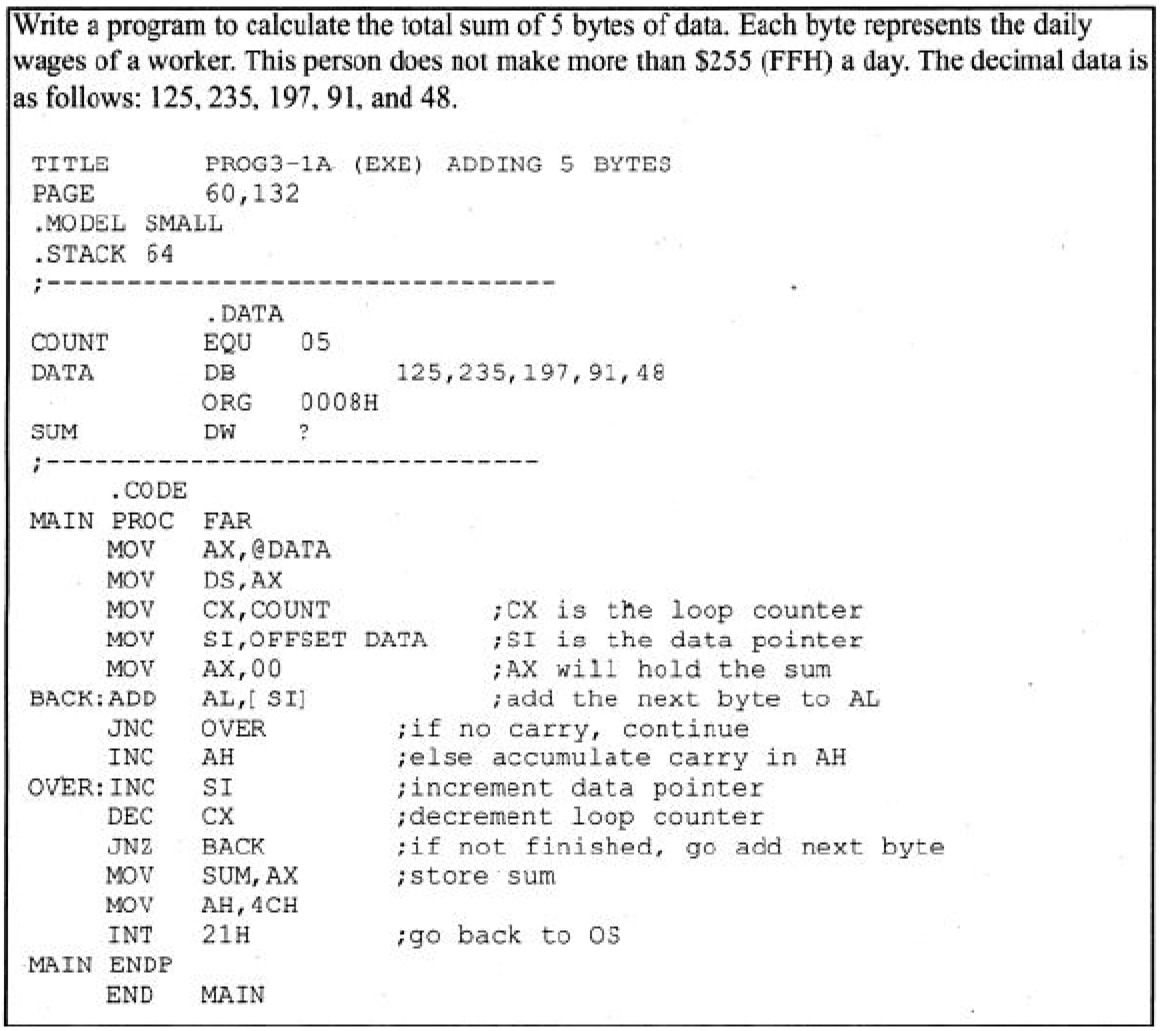
Solution:

```
    F5H        1111 0101
+   0BH    +   0000 1011
   100H        0000 0000
```

After the addition, the AL register (destination) contains 00 and the flags are as follows:
CF = 1, since there is a carry out from D7
SF = 0, the status of D7 of the result
PF = 1, the number of 1s is zero (zero is an even number)
AF = 1, there is a carry from D3 to D4
ZF = 1, the result of the action is zero (for the 8 bits)

With addition, two cases will be discussed:

CASE1: Addition of Individual Byte and Word Data:



Write a program to calculate the total sum of 5 bytes of data. Each byte represents the daily wages of a worker. This person does not make more than $255 (FFH) a day. The decimal data is as follows: 125, 235, 197, 91, and 48.

```
TITLE      PROG3-1A (EXE) ADDING 5 BYTES
PAGE       60,132
.MODEL SMALL
.STACK 64
;--------------------------------
           .DATA
COUNT      EQU   05
DATA       DB          125,235,197,91,48
           ORG   0008H
SUM        DW    ?
;-------------------------------
      .CODE
MAIN PROC  FAR
     MOV   AX,@DATA
     MOV   DS,AX
     MOV   CX,COUNT            ;CX is the loop counter
     MOV   SI,OFFSET DATA      ;SI is the data pointer
     MOV   AX,00               ;AX will hold the sum
BACK:ADD   AL,[SI]             ;add the next byte to AL
     JNC   OVER          ;if no carry, continue
     INC   AH            ;else accumulate carry in AH
OVER:INC   SI            ;increment data pointer
     DEC   CX            ;decrement loop counter
     JNZ   BACK          ;if not finished, go add next byte
     MOV   SUM,AX        ;store sum
     MOV   AH,4CH
     INT   21H           ;go back to OS
MAIN ENDP
     END   MAIN
```

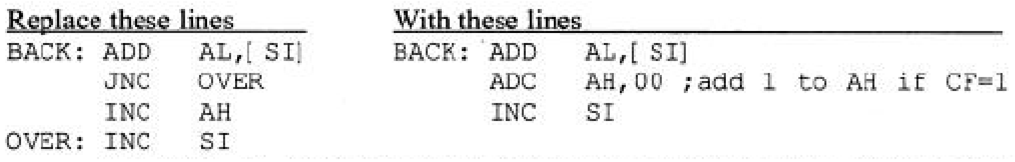
Program 3-1a

These numbers are converted to hex by the assembler as follows: 125 = 7DH, 235 = 0EBH, 197 = 0C5H, 91 = 5BH, 48 = 30H. This program uses AH to accumulate carries as the operands are added to AL register. Three iterations of the loop are shown below:

1. In the first iteration of the loop, 7DH is added to AL with CF = 0 and AH = 00. CX = 04 and ZF = 0.
2. In the second iteration of the loop, EBH is added to AL, which results in AL = 68H and CF = 1. Since a carry occurred, AH is incremented. CX = 03 and ZF = 0.
3. In the third iteration, C5H is added to AL, which makes AL = 2DH. Again a carry occurred, so AH is incremented again. CX = 02 and ZF = 0.

This process continues until CX = 00 and the zero flag becomes 1, which will cause JNZ to fall through. Then the result will be saved in the word-sized memory set aside in the data segment.

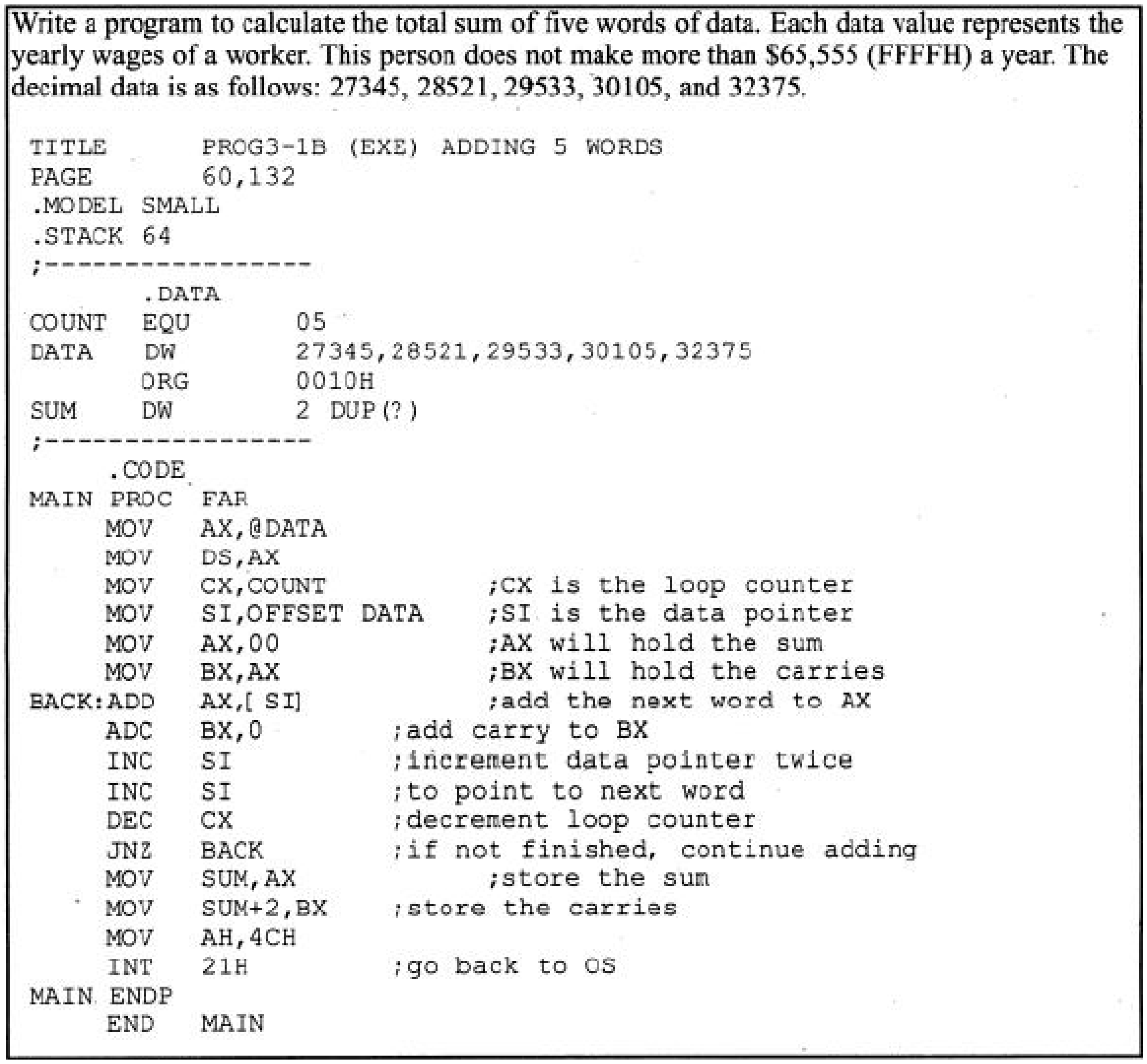
Although this program works correctly, due to pipelining it is strongly recommended that the following lines of the program be replaced:



```
Replace these lines            With these lines
BACK: ADD   AL,[SI]            BACK: ADD   AL,[SI]
      JNC   OVER                     ADC   AH,00 ;add 1 to AH if CF=1
      INC   AH                       INC   SI
OVER: INC   SI
```

The instruction "*JNC OVER*" has to empty the queue of pipelined instructions and fetch the instructions from the OVER target every time the carry is zero (CF = 0). Hence, the "*ADC AH, 00*" instruction is much more efficient.

The addition of many word operands works the same way. Register AX (or CX, DX, or BX) could be used as the accumulator and BX (or any general-purpose 16-bit register) for keeping the carries. Program 3-1b is the same as Program 3-1a, rewritten for word addition.

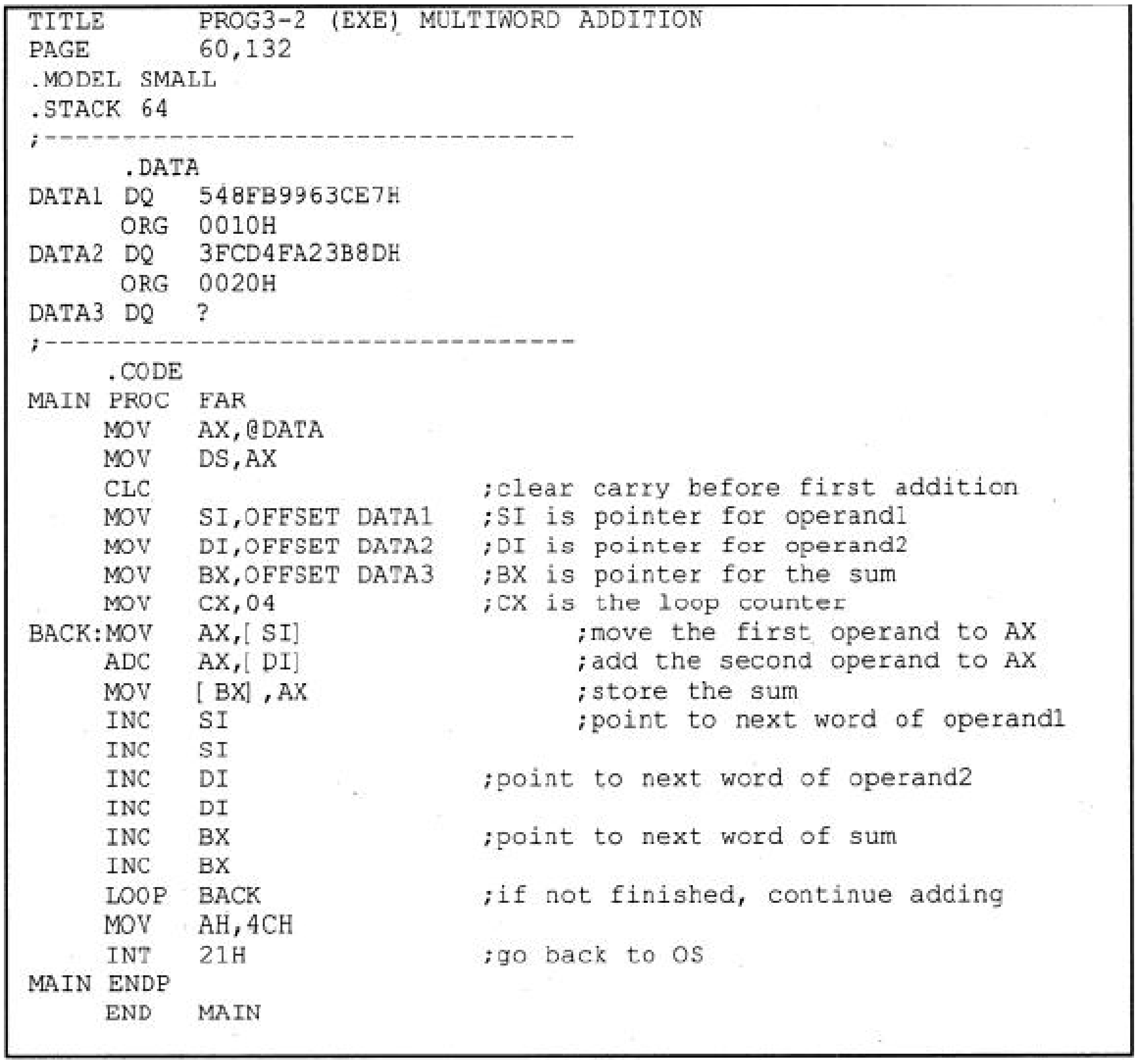


Write a program to calculate the total sum of five words of data. Each data value represents the yearly wages of a worker. This person does not make more than $65,555 (FFFFH) a year. The decimal data is as follows: 27345, 28521, 29533, 30105, and 32375.

```
TITLE       PROG3-1B (EXE) ADDING 5 WORDS
PAGE        60,132
.MODEL SMALL
.STACK 64
;------------------
       .DATA
COUNT  EQU        05
DATA   DW         27345,28521,29533,30105,32375
       ORG        0010H
SUM    DW         2 DUP(?)
;------------------
     .CODE
MAIN PROC  FAR
     MOV   AX,@DATA
     MOV   DS,AX
     MOV   CX,COUNT           ;CX is the loop counter
     MOV   SI,OFFSET DATA     ;SI is the data pointer
     MOV   AX,00              ;AX will hold the sum
     MOV   BX,AX              ;BX will hold the carries
BACK:ADD   AX,[SI]            ;add the next word to AX
     ADC   BX,0         ;add carry to BX
     INC   SI           ;increment data pointer twice
     INC   SI           ;to point to next word
     DEC   CX           ;decrement loop counter
     JNZ   BACK         ;if not finished, continue adding
     MOV   SUM,AX             ;store the sum
     MOV   SUM+2,BX     ;store the carries
     MOV   AH,4CH
     INT   21H          ;go back to OS
MAIN ENDP
     END   MAIN
```

Program 3-1b

CASE2: Addition of Multiword Numbers:



```
TITLE      PROG3-2  (EXE)  MULTIWORD ADDITION
PAGE       60,132
.MODEL SMALL
.STACK 64
;----------------------------------
      .DATA
DATA1 DQ    548FB9963CE7H
      ORG   0010H
DATA2 DQ    3FCD4FA23B8DH
      ORG   0020H
DATA3 DQ    ?
;----------------------------------
     .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     CLC                        ;clear carry before first addition
     MOV    SI,OFFSET DATA1     ;SI is pointer for operand1
     MOV    DI,OFFSET DATA2     ;DI is pointer for operand2
     MOV    BX,OFFSET DATA3     ;BX is pointer for the sum
     MOV    CX,04               ;CX is the loop counter
BACK:MOV    AX,[SI]                   ;move the first operand to AX
     ADC    AX,[DI]                   ;add the second operand to AX
     MOV    [BX],AX                   ;store the sum
     INC    SI                        ;point to next word of operand1
     INC    SI
     INC    DI                  ;point to next word of operand2
     INC    DI
     INC    BX                  ;point to next word of sum
     INC    BX
     LOOP   BACK                ;if not finished, continue adding
     MOV    AH,4CH
     INT    21H                 ;go back to OS
MAIN ENDP
     END    MAIN
```

Program 3-2

- Assume, a program is needed that will add the total Indian budget for the last 100 years or the mass of all the planets in the solar system.
- In cases like this, the numbers being added could be up to 8 bytes wide or even more. Since registers are only 16 bits wide (2 bytes), it is the job of the programmer to write the code to break down these large numbers into smaller chunks to be processed by the CPU.
- If a 16-bit register is used and the operand is 8 bytes wide, that would take a total of four iterations. However, if an 8-bit register is used, the same operands would require eight iterations.

✓ In writing this program, the first thing to be decided was the directive used for coding the data in the data segment. DQ was chosen since it can represent data as large as 8 bytes wide.

✓ In the addition of multibyte (or multiword) numbers, the ADC instruction is always used since the carry must be added to the next-higher byte (or word) in the next iteration. Before executing

ADC, the carry flag must be cleared (CF = 0) so that in the first iteration, the carry would not be added. Clearing the carry flag is achieved by the CLC (clear carry) instruction.

- ✓ Three pointers have been used: SI for DATA1, DI for DATA2, and BX for DATA3 where the result is saved.
- ✓ There is a new instruction in that program, "*LOOP xxxx*", which replaces the often used "*DEC CX*" and "*JNZ xxxx*".

```
LOOP   xxxx   ;is equivalent to   |   DEC    CX
                                  |   JNZ    xxxx
```

When "*LOOP xxxx*" is executed, CX is decremented automatically, and if CX is not 0, the microprocessor will jump to target address xxxx. If CX is 0, the next instruction (the one below "*LOOP xxxx*") is executed.

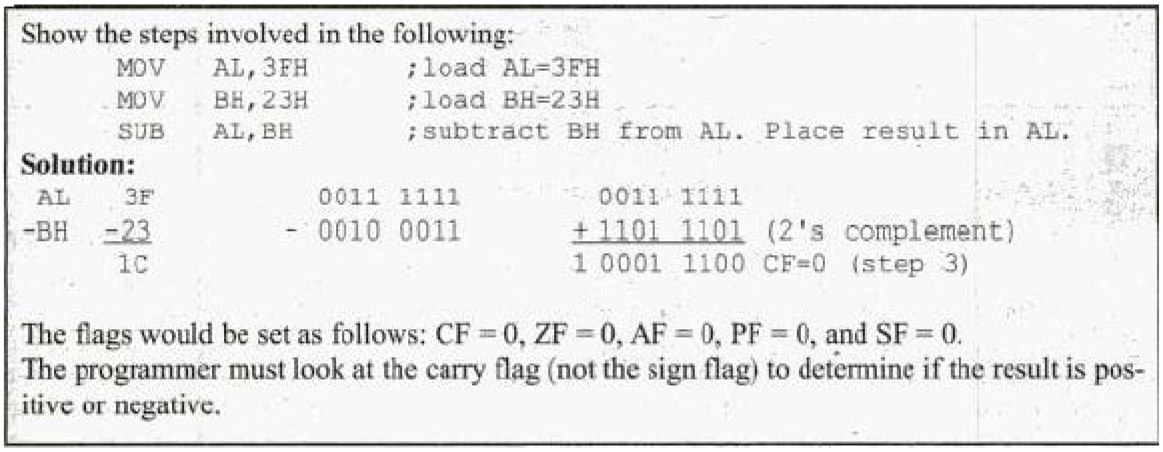
Subtraction of Unsigned Numbers:

```
SUB   dest,source;dest = dest - source
```

The x86 uses internal adder circuitry to perform the subtraction command. Hence, the 2's complement method is used by the microprocessor to perform the subtraction. The steps involved is –

1. Take the 2's complement of the subtrahend (source operand)
2. Add it to the minuend (destination operand)
3. Invert the carry.

These three steps are performed for every SUB instruction by the internal hardware of the x86 CPU. It is after these three steps that the result is obtained and the flags are set. The following example illustrates the three steps:



Show the steps involved in the following:

```
       MOV    AL,3FH         ;load AL=3FH
       MOV    BH,23H         ;load BH=23H
       SUB    AL,BH          ;subtract BH from AL. Place result in AL.
```

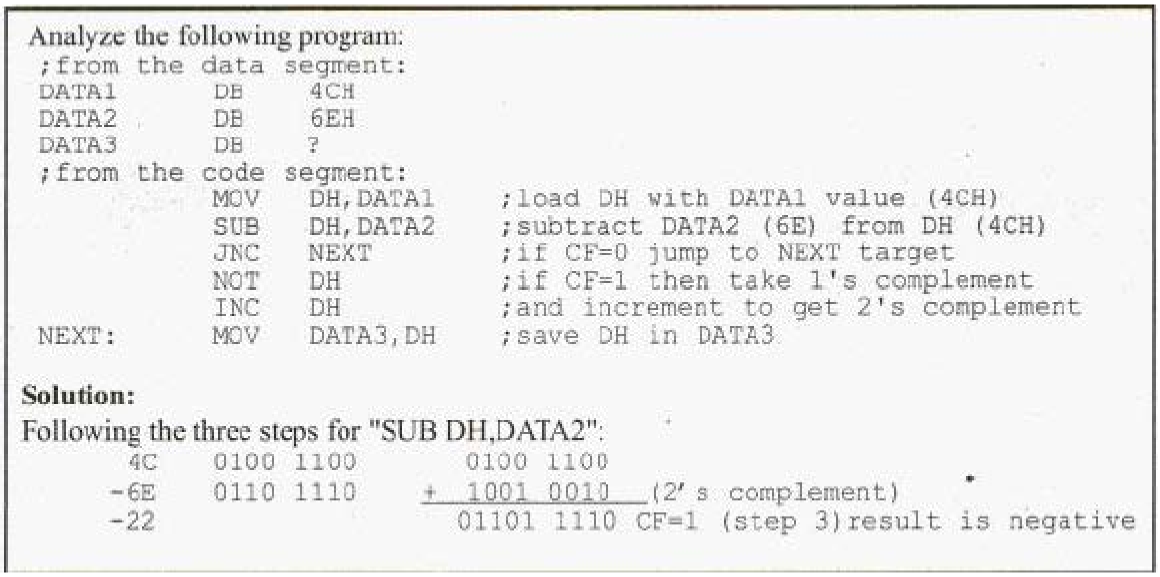
Solution:

```
 AL     3F           0011 1111          0011 1111
-BH    -23         - 0010 0011        + 1101 1101 (2's complement)
        1C                            1 0001 1100 CF=0  (step 3)
```

The flags would be set as follows: CF = 0, ZF = 0, AF = 0, PF = 0, and SF = 0.
The programmer must look at the carry flag (not the sign flag) to determine if the result is positive or negative.

- ✓ After the execution of SUB, if CF = 0, the result is positive; if CF = 1, the result is negative and the destination has the 2's complement of the result.

- Normally, the result is left in 2's complement, but the NOT and INC instructions can be used to change it. The NOT instruction performs the 1’s complement of the operand; then the operand is incremented to get the 2's complement; as shown in the following example:



Analyze the following program:

```
;from the data segment:
DATA1        DB      4CH
DATA2        DB      6EH
DATA3        DB      ?
;from the code segment:
             MOV     DH,DATA1      ;load DH with DATA1 value (4CH)
             SUB     DH,DATA2      ;subtract DATA2 (6E) from DH (4CH)
             JNC     NEXT          ;if CF=0 jump to NEXT target
             NOT     DH            ;if CF=1 then take 1's complement
             INC     DH            ;and increment to get 2's complement
NEXT:        MOV     DATA3,DH      ;save DH in DATA3
```

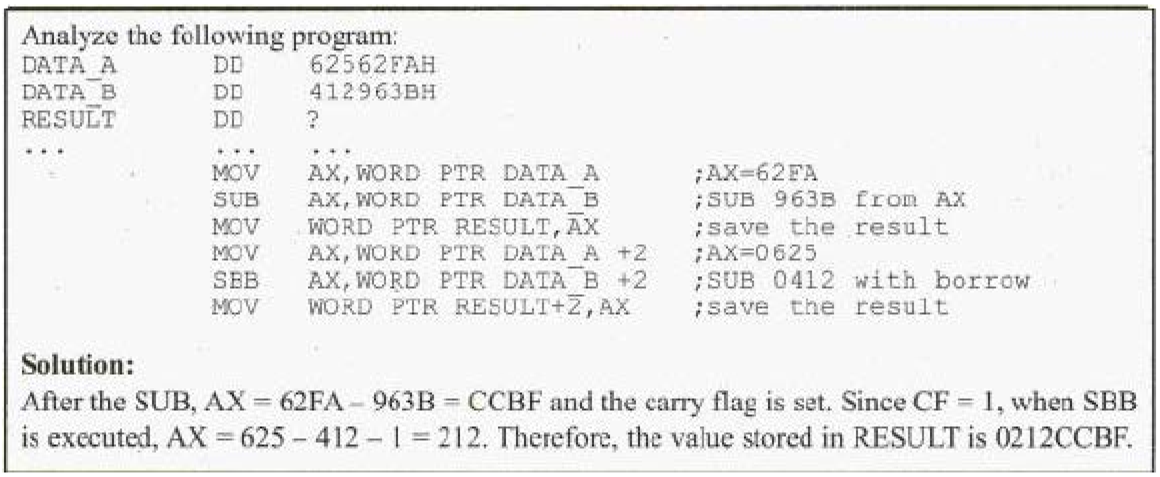
Solution:

Following the three steps for "SUB DH,DATA2":

```
      4C     0100 1100          0100 1100
    -6E      0110 1110      +  1001 0010   (2's complement)
    -22                        01101 1110 CF=1 (step 3)result is negative
```

SBB (Subtract with Borrow):

This instruction is used for multibyte (multiword) numbers and will take care of the borrow of the lower operand. If the carry flag is 0, SBB works like SUB. If the carry flag is 1, SBB subtracts 1 from the result. Notice the "*PTR*" operand in the following Example.



Analyze the following program:

```
DATA_A       DD      62562FAH
DATA_B       DD      412963BH
RESULT       DD      ?
...          ...     ...
             MOV     AX,WORD PTR DATA_A          ;AX=62FA
             SUB     AX,WORD PTR DATA_B          ;SUB 963B from AX
             MOV     WORD PTR RESULT,AX          ;save the result
             MOV     AX,WORD PTR DATA_A +2       ;AX=0625
             SBB     AX,WORD PTR DATA_B +2       ;SUB 0412 with borrow
             MOV     WORD PTR RESULT+2,AX        ;save the result
```

Solution:

After the SUB, AX = 62FA – 963B = CCBF and the carry flag is set. Since CF = 1, when SBB is executed, AX = 625 – 412 – 1 = 212. Therefore, the value stored in RESULT is 0212CCBF.

The PTR (pointer) data directive is used to specify the size of the operand when it differs from the defined size. In above Example; "*WORD PTR*" tells the assembler to use a word operand, even though the data is defined as a double word.

UNSIGNED MULTIPLICATION AND DIVISION:

One of the major changes from the 8080/85 microprocessor to the 8086 was inclusion of instructions for multiplication and division. The use of registers AX, AL, AH, and DX is necessary.

Multiplication of Unsigned Numbers:

In discussing multiplication, the following cases will be examined: (1) byte times byte, (2) word times word, and (3) byte times word.

8-bit * 8-bit	AL * BL	16-bit * 16-bit	AX * BX
16-bit	AX	32-bit	DX AX

byte x byte: In byte-by-byte multiplication, one of the operands must be in the AL register and the second operand can be either in a register or in memory. After the multiplication, the result is in AX.

```
RESULT   DW    ?           ;result is defined in the data segment
         ...
         MOV   AL,25H      ;a byte is moved to AL
         MOV   BL,65H      ;immediate data must be in a register
         MUL   BL          ;AL = 25 x 65H
         MOV   RESULT,AX   ;the result is saved
```

In the program above, 25H is multiplied by 65H and the result is saved in word-sized memory named RESULT. Here, the register addressing mode is used.

The next three examples show the register, direct, and register indirect addressing modes.

```
;from the data segment:
DATA1        DB      25H
DATA2        DB      65H
RESULT       DW      ?
;from the code segment:
       MOV   AL,DATA1
       MOV   BL,DATA2
       MUL   BL                 ;register addressing mode
       MOV   RESULT,AX
or
       MOV   AL,DATA1
       MUL   DATA2              ;direct addressing mode
       MOV   RESULT,AX
or
       MOV   AL,DATA1
       MOV   SI,OFFSET DATA2
       MUL   BYTE PTR [SI]      ;register indirect addressing mode
       MOV   RESULT,AX
```

- ✓ In the register addressing mode example, any 8-bit register could have been used in place BL.
- ✓ Similarly, in the register indirect example, BX or DI could have been used as pointers.
- ✓ If the register indirect addressing mode is used, the operand size must be specified with the help of the PTR pseudo-instruction. In the absence of the "*BYTE PTR*" directive in the example above,

the assembler could not figure out if it should use a byte or word operand pointed at by SI. This confusion may cause an error.

word x word: In word-by-word multiplication, one operand must be in AX and the second operand can be in a register or memory. After the multiplication, registers DX and AX will contain the result. Since word-by-word multiplication can produce a 32-bit result, DX will hold the higher word and AX the lower word.

```
DATA3       DW    2378H
DATA4       DW    2F79H
RESULT1     DW    2 DUP(?)
...         ....
            MOV  AX,DATA3     ;load first operand into AX
            MUL  DATA4        ;multiply it by the second operand
            MOV  RESULT1,AX   ;store the lower word result
            MOV  RESULT1+2,DX ;store the higher word result
```

word x byte: This is similar to word-by-word multiplication, except that AL-contains the byte operand and AH must be set to zero.

```
;from the data segment:
DATA5       DB    6BH
DATA6       DW    12C3H
RESULT3     DW    2 DUP(?)
;from the code segment:
            MOV  AL,DATA5     ;AL holds byte operand
            SUB  AH,AH        ;AH must be cleared
            MUL  DATA6        ;byte in AL mult. by word operand
            MOV  BX,OFFSET RESULT3 ;BX points to product
            MOV  [BX],AX           ;AX holds lower word
            MOV  [BX]+2,DX         ;DX holds higher word
```

Table: Unsigned Multiplication Summary

Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX
word × word	AX	register or memory	DX AX
word × byte	AL = byte, AH = 0	register or memory	DX AX

Division of Unsigned Numbers:

In the division of unsigned numbers, the following cases are discussed:

1. Byte over byte
2. Word over word
3. Word over byte
4. Double-word over word

8-bit	AL	Q: AL	16-bit	AX	Q: AX
8-bit	BL	R: AH	16-bit	BX	R: DX

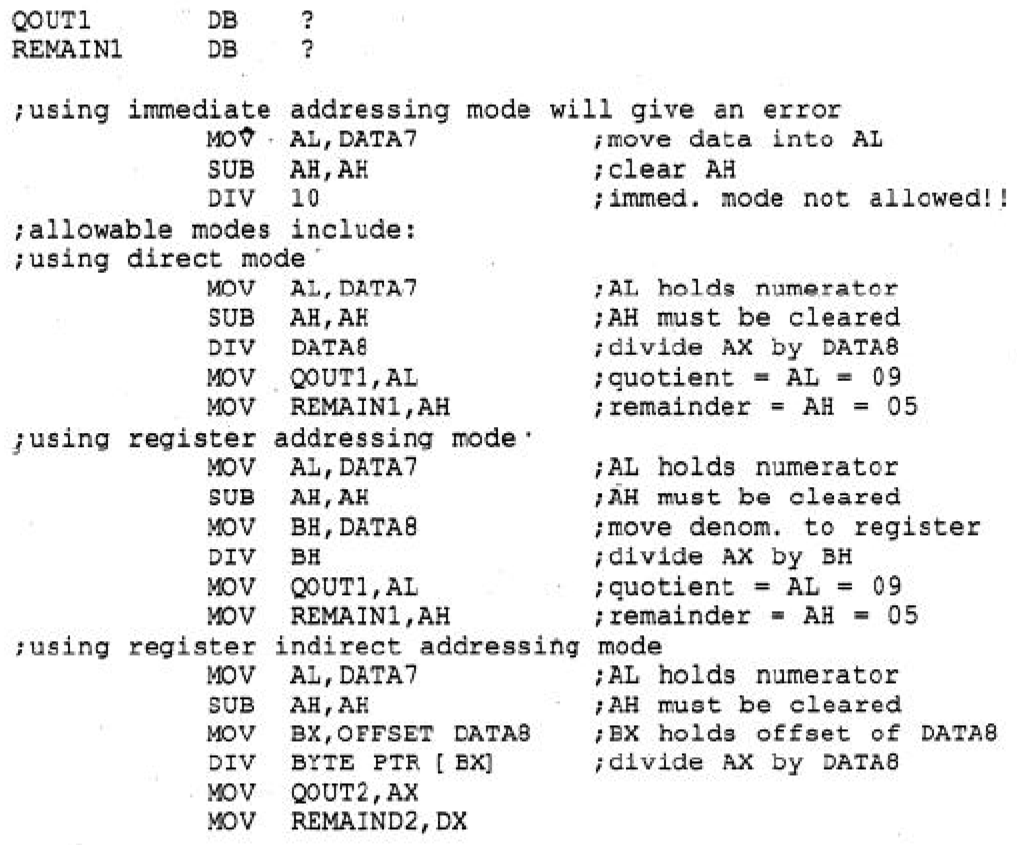
16-bit	AX	Q: AL	32-bit	DA AX	Q: AX
8-bit	BL	R: AH	16-bit	BX	R: DX

In divide, there could be cases where the CPU cannot perform the division. In these cases an ***interrupt*** is activated. This is referred to as an *exception*. In following situations, the microprocessor cannot handle the division and must call an interrupt:

1. If the denominator is zero (dividing any number by 00)
2. If the quotient is too large for the assigned register.

In the IBM PC and compatibles, if either of these cases happens, the PC will display the "divide error" message.

byte/byte: In dividing a byte by a byte, the numerator must be in the AL register and AH must be set to zero. The denominator cannot be immediate but can be in a register or memory. After the DIV instruction is performed, the quotient is in AL and the remainder is in AH.



```
QOUT1         DB    ?
REMAIN1       DB    ?

;using immediate addressing mode will give an error
            MOV   AL,DATA7           ;move data into AL
            SUB   AH,AH              ;clear AH
            DIV   10                 ;immed. mode not allowed!!
;allowable modes include:
;using direct mode
            MOV   AL,DATA7           ;AL holds numerator
            SUB   AH,AH              ;AH must be cleared
            DIV   DATA8              ;divide AX by DATA8
            MOV   QOUT1,AL           ;quotient = AL = 09
            MOV   REMAIN1,AH         ;remainder = AH = 05
;using register addressing mode
            MOV   AL,DATA7           ;AL holds numerator
            SUB   AH,AH              ;AH must be cleared
            MOV   BH,DATA8           ;move denom. to register
            DIV   BH                 ;divide AX by BH
            MOV   QOUT1,AL           ;quotient = AL = 09
            MOV   REMAIN1,AH         ;remainder = AH = 05
;using register indirect addressing mode
            MOV   AL,DATA7           ;AL holds numerator
            SUB   AH,AH              ;AH must be cleared
            MOV   BX,OFFSET DATA8    ;BX holds offset of DATA8
            DIV   BYTE PTR [BX]      ;divide AX by DATA8
            MOV   QOUT2,AX
            MOV   REMAIND2,DX
```

word/word: In this case, the numerator is in AX and DX must be cleared. The denominator can be in a register or memory. After the DIV; AX will have the quotient and the remainder will be in DX.

```
MOV   AX,10050     ;AX holds numerator
SUB   DX,DX        ;DX must be cleared
MOV   BX,100       ;BX used for denominator
DIV   BX
MOV   QOUT2,AX     ;quotient = AX = 64H = 100
MOV   REMAIND2,DX  ;remainder = DX = 32H = 50
```

word/byte: Here, the numerator is in AX and the denominator can be in a register or memory. After the DIV instruction, AL will contain the quotient, and AH will contain the remainder. The maximum quotient is FFH.

The following program divides AX = 2055 by CL = 100. Then AL = 14H (20 decimal) is the quotient and AH = 37H (55 decimal) is the remainder.

```
MOV   AX,2055      ;AX holds numerator
MOV   CL,100       ;CL used for denominator
DIV   CL
MOV   QUO,AL       ;AL holds quotient
MOV   REMI,AH      ;AH holds remainder
```

Double-word/word: The numerator is in DX and AX, with the most significant word in DX and the least significant word in AX. The denominator can be in a register or in memory. After the DIV instruction; the quotient will be in AX, and the remainder in DX. The maximum quotient is FFFFH.

```
;from the data segment:
DATA1        DD    105432
DATA2        DW    10000
QUOT         DW    ?
REMAIN       DW    ?
;from the code segment:
             MOV   AX,WORD PTR DATA1        ;AX holds lower word
             MOV   DX,WORD PTR DATA1+2;DX higher word of numerator
             DIV   DATA2
             MOV   QUOT,AX                  ;AX holds quotient
             MOV   REMAIN,DX                ;DX holds remainder
```

- ✓ In the program above, the contents of DX: AX are divided by a word-sized data value, 10000.
- ✓ The 8088/86 automatically uses DX: AX as the numerator anytime the denominator is a word in size.
- ✓ Notice in the example above that DATAl is defined as DD but fetched into a word-size register with the help of WORD PTR. In the absence of WORD PTR, the assembler will generate an error.

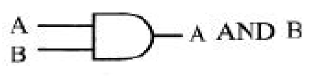
Table: Unsigned Division Summary

Division	Numerator	Denominator	Quotient	Rem.
byte/byte	AL = byte, AH = 0	register or memory	AL[1]	AH
word/word	AX = word, DX = 0	register or memory	AX[2]	DX
word/byte	AX = word	register or memory	AL[1]	AH
doubleword/word	DXAX = doubleword	register or memory	AX[2]	DX

LOGIC INSTRUCTIONS:

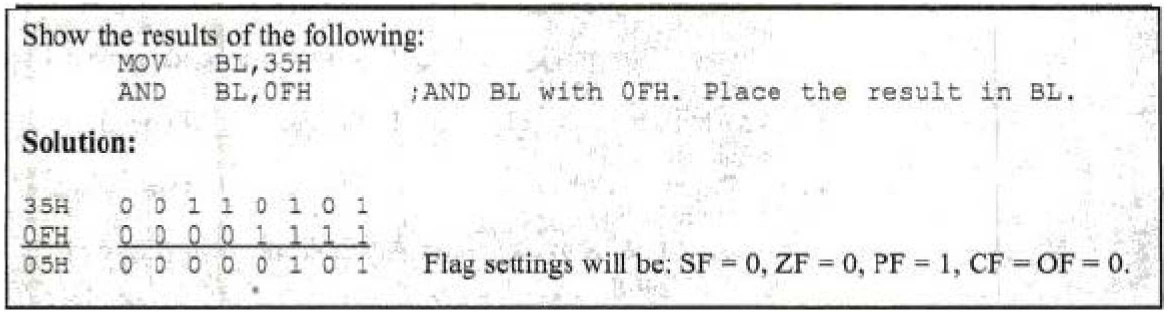
Here, the logic instructions AND, OR, XOR, SHIFT, and COMPARE are discussed with examples.

AND

Inputs		Output
A	**B**	**A AND B**
0	0	0
0	1	0
1	0	0
1	1	1
		

```
AND destination, source
```

- ✓ This instruction will perform a logical AND on the operands and place the result in the destination. The destination operand can be a register or memory. The source operand can be a register, memory, or immediate.
- ✓ AND will automatically change the CF and OF to zero, and PF, ZF, and SF are set according to the result. The rest of the flags are either undecided or unaffected.



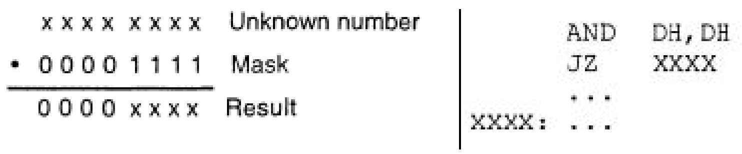
Show the results of the following:

```
        MOV   BL,35H
        AND   BL,0FH        ;AND BL with 0FH. Place the result in BL.
```

Solution:

```
35H    0 0 1 1 0 1 0 1
0FH    0 0 0 0 1 1 1 1
05H    0 0 0 0 0 1 0 1     Flag settings will be: SF = 0, ZF = 0, PF = 1, CF = OF = 0.
```

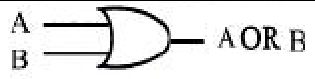
- ✓ AND can be used to mask certain bits of the operand. The task of clearing a bit in a binary number is called **masking**. It can also be used to test for a zero operand.



```
  x x x x  x x x x   Unknown number          AND   DH,DH
• 0 0 0 0 1 1 1 1    Mask                    JZ    XXXX
  0 0 0 0 x x x x    Result                  ...
                                      XXXX:  ...
```

- ✓ The above code will AND DH with itself, and set ZF =1, if the result is zero. This makes the CPU to fetch from the target address XXXX. Otherwise, the instruction below JZ is executed. AND can thus be used to test if a register contains zero.

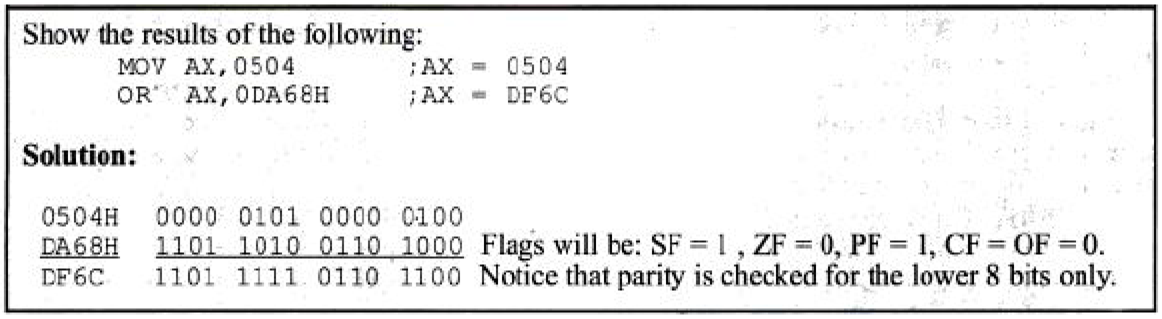
OR

Inputs		Output
A	**B**	**A OR B**
0	0	0
0	1	1
1	0	1
1	1	1
		

```
OR     destination,source
```

- ✓ The destination and source operands are ORed and the result is placed in the destination.
- ✓ The destination operand can be a register or in memory. The source operand can be a register, memory, or immediate.
- ✓ OR will automatically change the CF and OF to zero, and PF, ZF,

and SF are set according to the result. The rest of the flags are either undecided or unaffected.



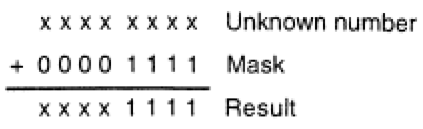
Show the results of the following:

```
        MOV  AX,0504        ;AX = 0504
        OR   AX,0DA68H      ;AX = DF6C
```

Solution:

```
 0504H   0000 0101 0000 0100
 DA68H   1101 1010 0110 1000  Flags will be: SF = 1 , ZF = 0, PF = 1, CF = OF = 0.
 DF6C    1101 1111 0110 1100  Notice that parity is checked for the lower 8 bits only.
```

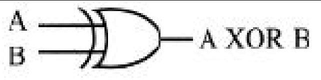
- ✓ The OR instruction can be used to test for a zero operand. For example, "*OR BL, 0*"will OR the register BL with 0 and make ZF = 1, if BL is zero. "*OR BL, BL*" will achieve the same result.
- ✓ OR can also be used to set certain bits of an operand to 1.



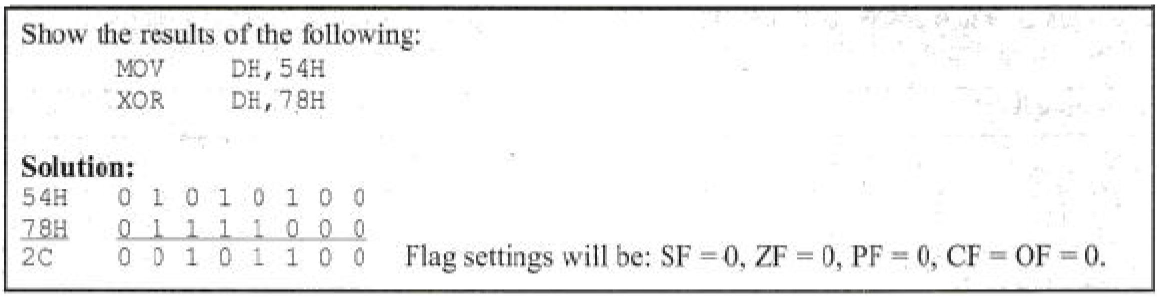
```
   x x x x  x x x x   Unknown number
 + 0 0 0 0  1 1 1 1   Mask
 ------------------
   x x x x  1 1 1 1   Result
```

XOR

```
XOR   dest,src
```

Inputs		Output
A	**B**	**A XOR B**
0	0	0
0	1	1
1	0	1
1	1	0
		

- ✓ The XOR instruction will eXclusive-OR the operands and place the result in the destination. XOR sets the result bits to 1 if they are not equal; otherwise, they are reset to 0.
- ✓ The destination operand can be a register or in memory. The source operand can be a register, memory, or immediate.
- ✓ OR will automatically change the CF and OF to zero, and PF, ZF, and SF are set according to the result. The rest of the flags are either undecided or unaffected.

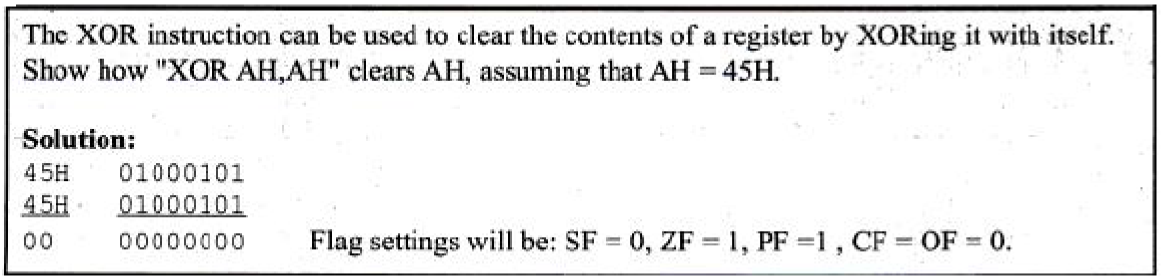


Show the results of the following:

```
        MOV     DH,54H
        XOR     DH,78H
```

Solution:

```
54H     0 1 0 1 0 1 0 0
78H     0 1 1 1 1 0 0 0
2C      0 0 1 0 1 1 0 0     Flag settings will be: SF = 0, ZF = 0, PF = 0, CF = OF = 0.
```



The XOR instruction can be used to clear the contents of a register by XORing it with itself. Show how "XOR AH,AH" clears AH, assuming that AH = 45H.

Solution:

```
45H     01000101
45H     01000101
00      00000000      Flag settings will be: SF = 0, ZF = 1, PF =1 , CF = OF = 0.
```

- ✓ XOR can be used to see if two registers have the same value. "*XOR BX, CX*" will make ZF = 1, if both registers have the same value, and if they do, the result (0000) is saved in BX, the destination.
- ✓ XOR can also be used to toggle (invert/compliment) bits of an operand. For example, to toggle bit 2 of register AL:

```
 xxxx xxxx   Unknown number
⊕0000 1111   Mask
 xxxx x̄x̄x̄x̄   Result
```

```
XOR  AL,04H        ;XOR  AL with 0000 0100
```

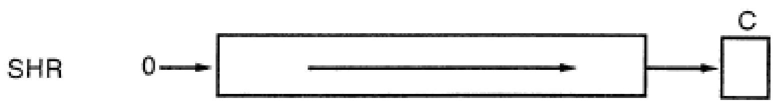
- ✓ This would cause bit 2 of AL to change to the opposite value; all other bits would remain unchanged.

SHIFT

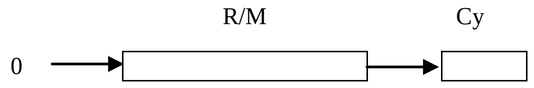
- o Shift instructions shift the contents of a register or memory location right or left.
- o The number of times (or bits) that the operand is shifted can be specified directly if it is once only, or through the CL register if it is more than once.
- o There are two kinds of shifts:
  - ✓ Logical – for unsigned operands
  - ✓ Arithmetic – signed operands.

SHR: This is the logical shift right. The operand is shifted right bit by bit, and for every shift the LSB (least significant bit) will go to the carry flag (CF) and the MSB (most significant bit) is filled with 0.

- ✓ SHR does affect the OF, SF, PF, and ZF flags.
- ✓ The operand to be shifted can be in a register or in memory, but immediate addressing mode is not allowed for shift instructions. For example, "*SHR 25, CL*" will cause the assembler to give an error.

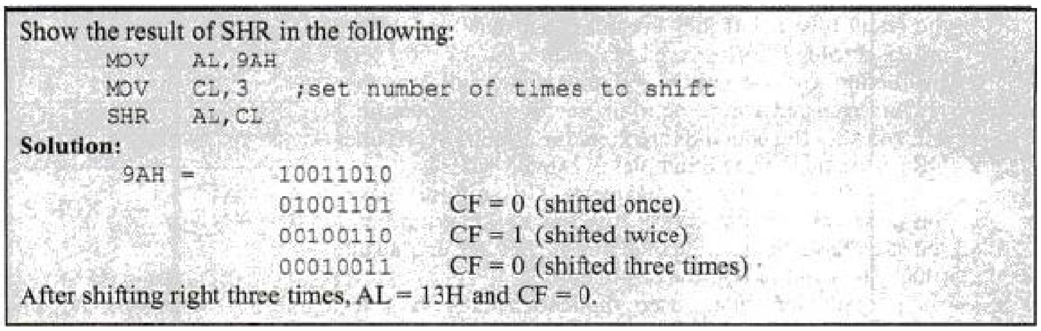


Eg:
SHR BH, CL



Shift right

	Before	*After*
BH	0100 0100	0001 0001
CL	02H	
Cy	1	0


Show the result of SHR in the following:

```
        MOV    AL,9AH
        MOV    CL,3    ;set number of times to shift
        SHR    AL,CL
```

Solution:

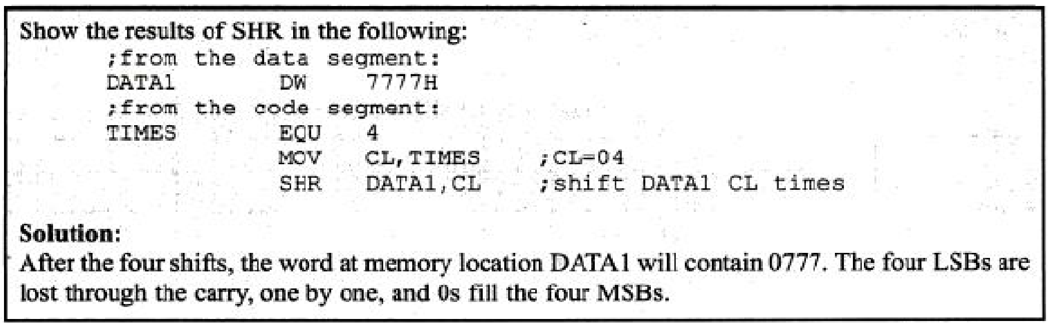
```
        9AH =       10011010
                    01001101      CF = 0 (shifted once)
                    00100110      CF = 1 (shifted twice)
                    00010011      CF = 0 (shifted three times)
```

After shifting right three times, AL = 13H and CF = 0.

- ✓ If the operand is to be shifted once only, this is specified in the SHR instruction itself rather than placing 1 in the CL. This saves coding of one instruction:

```
MOV    BX,0FFFFH    ;BX=FFFFH
SHR    BX,1         ;shift right BX once only
```

- ✓ After the above shift, BX = 7FFFH and CF = 1.

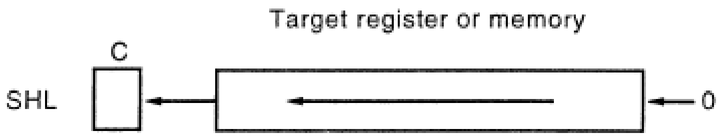

Show the results of SHR in the following:

```
        ;from the data segment:
        DATA1          DW     7777H
        ;from the code segment:
        TIMES          EQU    4
                       MOV    CL,TIMES     ;CL=04
                       SHR    DATA1,CL     ;shift DATA1 CL times
```

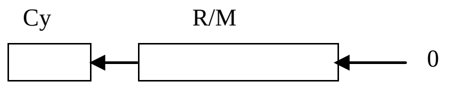
Solution:
After the four shifts, the word at memory location DATA1 will contain 0777. The four LSBs are lost through the carry, one by one, and 0s fill the four MSBs.

SHL: Shift left is also a logical shift. It is the reverse of SHR. After every shift the LSB is filled with 0 and the MSB goes to CF.

- ✓ SHL does affect the OF, SF, PF, and ZF flags.
- ✓ The operand to be shifted can be in a register or in memory, but immediate addressing mode is not allowed for shift instructions. For example, "*SHL 25, CL*" will cause the assembler to give an error.

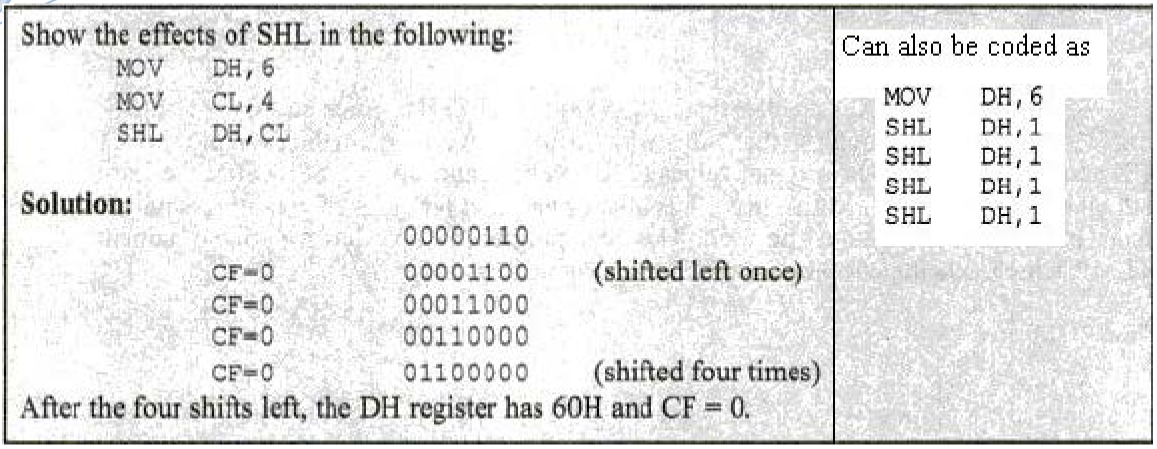


Eg:
SHL BH, CL



Shift left without Cy

	Before	*After*
BH	0010 0010	1000 1000
CL	02H	
Cy	1	0



Show the effects of SHL in the following:

```
        MOV     DH,6
        MOV     CL,4
        SHL     DH,CL
```

Solution:

```
                        00000110
        CF=0            00001100        (shifted left once)
        CF=0            00011000
        CF=0            00110000
        CF=0            01100000        (shifted four times)
```

After the four shifts left, the DH register has 60H and CF = 0.

Can also be coded as

```
MOV     DH,6
SHL     DH,1
SHL     DH,1
SHL     DH,1
SHL     DH,1
```

COMPARE of Unsigned Numbers:

```
CMP   destination,source   ;compare dest and src
```

- ✓ The CMP instruction compares two operands and changes the flags according to the result of the comparison. The operands themselves remain unchanged.
- ✓ The destination operand can be in a register or in memory and the source operand can be in a register, memory, or immediate.
- ✓ The compare instruction is really a SUBtraction, except that the values of the operands do not change.
- ✓ The flags are changed according to the execution of SUB. Although all the flags (CF, AF, SF, PF, ZF, and OF flags) are affected, the only ones of interest are ZF and CF.
- ✓ It must be emphasized that in CMP instructions, the operands are unaffected regardless of the result of the comparison. Only the flags are affected.

Table: Flag Settings for Compare Instruction

Compare Operands	**CF**	**ZF**	**Remark**
destination > source	0	0	destination – source; results CF = 0 & ZF = 0
destination = source	0	1	destination – source; results CF = 0 & ZF = 1
destination < source	1	0	destination – source; results CF = 1 & ZF = 0

```
DATA1  DW    235FH
       ...
       MOV   AX,0CCCCH
       CMP   AX,DATA1     ;compare CCCC with 235F
       JNC   OVER         ;jump if CF=0
       SUB   AX,AX
OVER:  INC   DATA1
```

- ✓ In the program above, AX is greater than the contents of memory location DATA1 (0CCCCH > 235FH); therefore, CF = 0 and JNC (jump no carry) will go to target OVER.

```
       MOV   BX,7888H
       MOV   CX,9FFFH
       CMP   BX,CX          ;compare 7888 with 9FFF
       JNC   NEXT
       ADD   BX,4000H
NEXT:  ADD   CX,250H
```

- ✓ In the above code, BX is smaller than CX (7888H < 9FFFH), which sets CF = 1, making "*JNC NEXT*" fall through so that "*ADD BX, 4000H*" is executed.
- ✓ In the example above, CX and BX still have their original values (CX = 9FFFH and BX =7888H) after the execution of "*CMP BX, CX*".
- ✓ Notice that CF is always checked for cases of greater or smaller than, but for equal, ZF must be used.

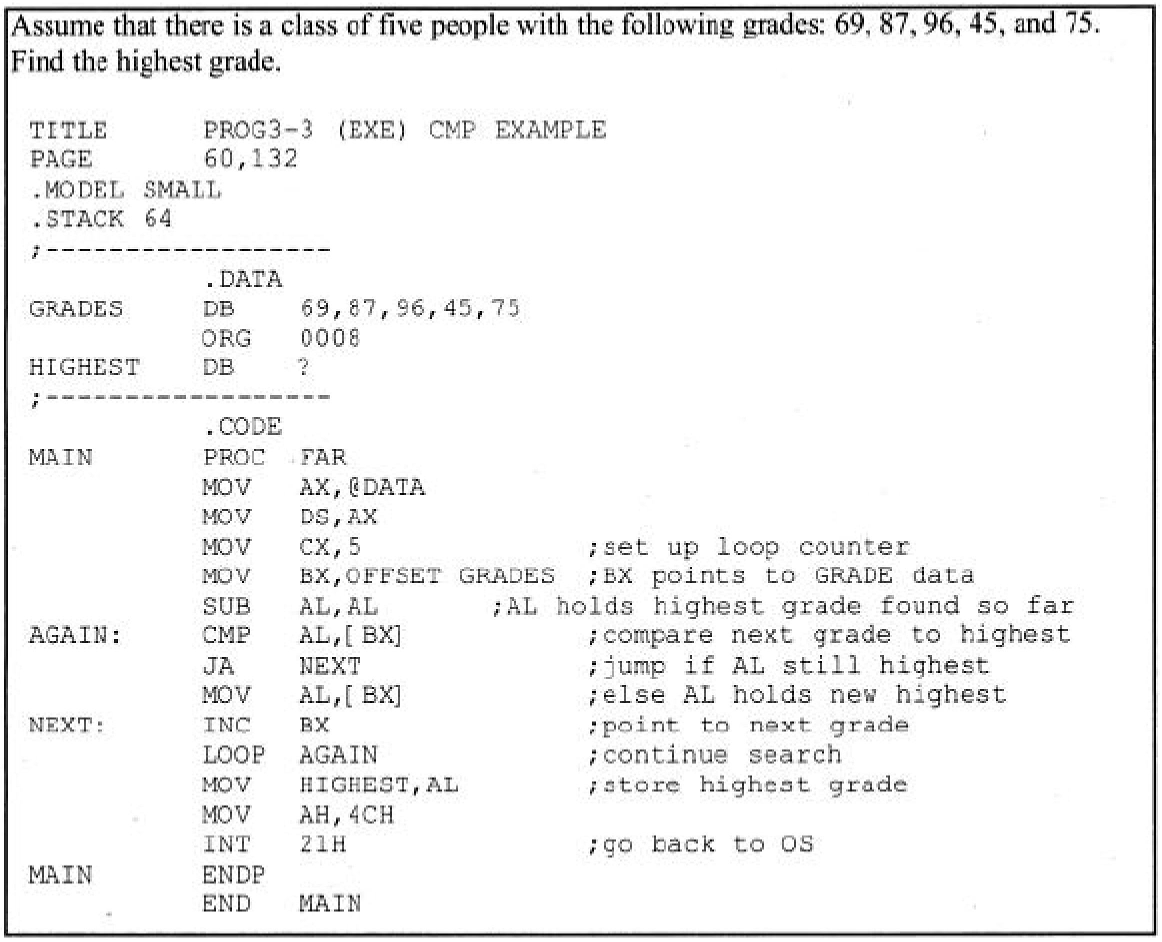
```
TEMP   DB    ?
...
       MOV   AL,TEMP        ;move the TEMP variable into AL
       CMP   AL,99          ;compare AL with 99
       JZ    HOT_HOT        ;if ZF=1 (TEMP = 99) jump to HOT_HOT
       INC   BX             ;otherwise (ZF=0) increment BX
...
HOT_HOT: HLT                ;halt the system
```

- ✓ The above program sample has a variable named TEMP, which is being checked to see if it has reached 99.

In the following Program the CMP instruction is used to search for the highest byte in a series of 5 bytes defined in the data segment.

- ✓ The instruction "*CMP AL, [BX]*" works as follows ([BX] is the contents of the memory location pointed at by register BX).
  - If AL < [BX], then CF = 1 and [BX] becomes the basis of the new comparison.
  - If AL > [BX], then CF = 0 and AL is the larger of the two values and remains the basis of comparison.
- ✓ Although JC (jump carry) and JNC (jump no carry) check the carry flag and can be used after a compare instruction, it is recommended that JA (jump above) and JB (jump below) be used because,
  - The assemblers will unassembled JC as JB, and JNC as JA.
- ✓ The below Program searches through five data items to find the highest grade.
- ✓ The program has a variable called "Highest" that holds the highest grade found so far. One by one, the grades are compared to Highest. If any of them is higher, that value is placed in Highest.
- ✓ This continues until all data items are checked. A REPEAT-UNTIL structure was chosen in the program design.
- ✓ The program uses register AL to hold the highest grade found so far. AL is given the initial value of 0. A loop is used to compare each of the 5 bytes with the value in AL.

- ✓ If AL contains a higher value, the loop continues to check the next byte. If AL is smaller than the byte being checked, the contents of AL are replaced by that byte and the loop continues.



Assume that there is a class of five people with the following grades: 69, 87, 96, 45, and 75. Find the highest grade.

```
TITLE       PROG3-3  (EXE)  CMP  EXAMPLE
PAGE        60,132
.MODEL SMALL
.STACK 64
;------------------
            .DATA
GRADES      DB     69,87,96,45,75
            ORG    0008
HIGHEST     DB     ?
;------------------
            .CODE
MAIN        PROC   FAR
            MOV    AX,@DATA
            MOV    DS,AX
            MOV    CX,5                ;set up loop counter
            MOV    BX,OFFSET GRADES    ;BX points to GRADE data
            SUB    AL,AL        ;AL holds highest grade found so far
AGAIN:      CMP    AL,[BX]             ;compare next grade to highest
            JA     NEXT                ;jump if AL still highest
            MOV    AL,[BX]             ;else AL holds new highest
NEXT:       INC    BX                  ;point to next grade
            LOOP   AGAIN               ;continue search
            MOV    HIGHEST,AL          ;store highest grade
            MOV    AH,4CH
            INT    21H                 ;go back to OS
MAIN        ENDP
            END    MAIN
```

Program 3-3

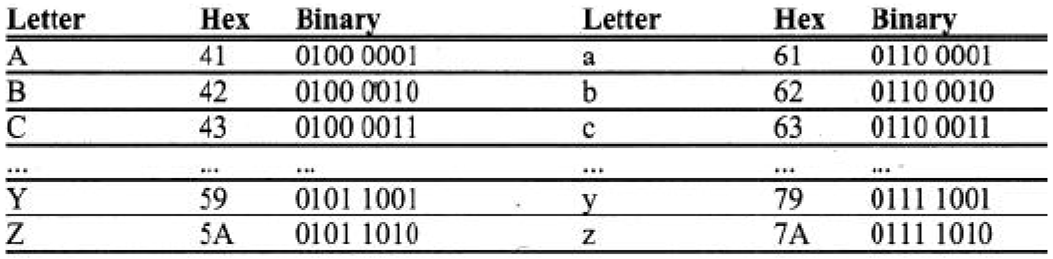
NOTE:

There is a relationship between the pattem of lowercase and uppercase letters, as shown below for *A* and *a*:

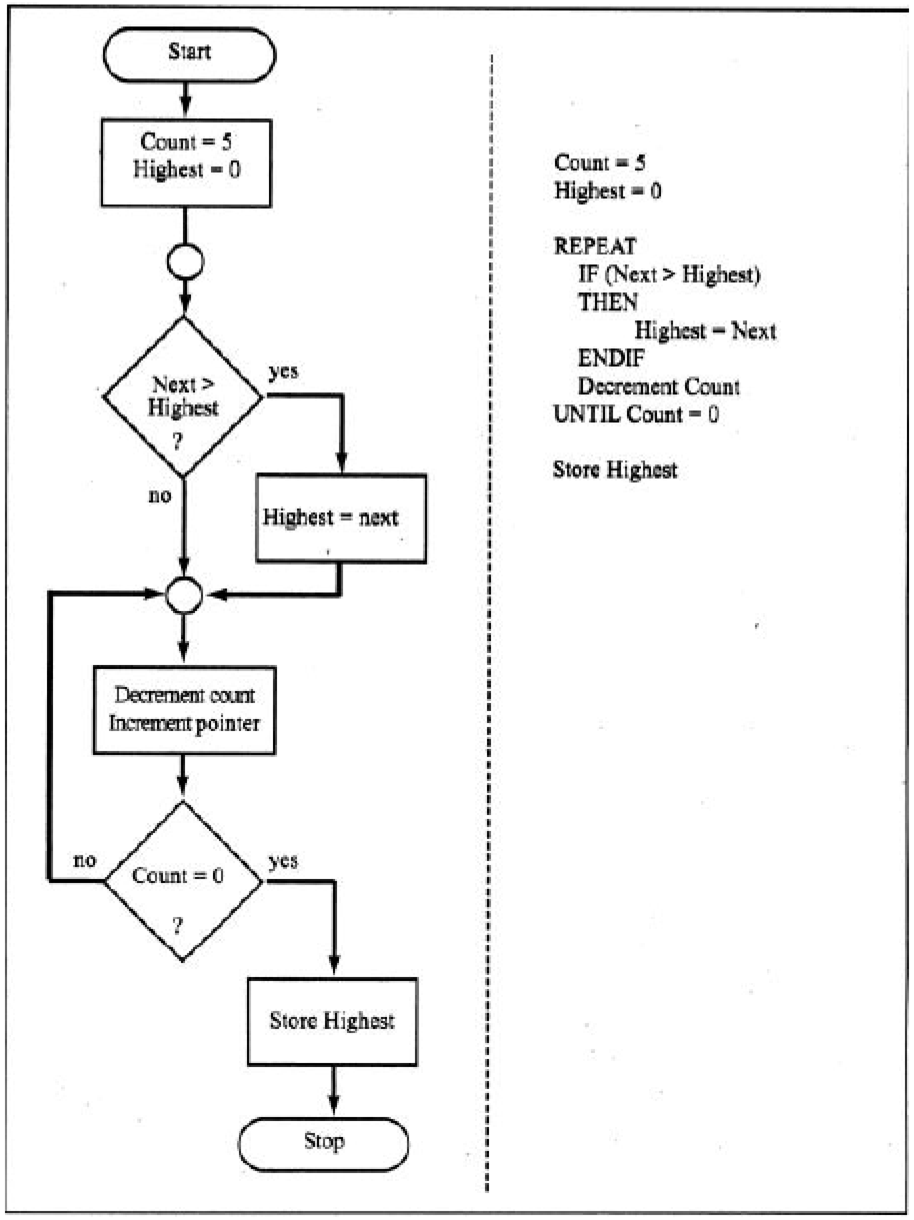
A	*0100 0001*	*41H*
a	*0110 0001*	*61H*

The only bit that changes is d5. To change from lowercase to uppercase , d5 must be masked.

Note that small and capital letters in ASCII have the following values:



Letter	Hex	Binary	Letter	Hex	Binary
A	41	0100 0001	a	61	0110 0001
B	42	0100 0010	b	62	0110 0010
C	43	0100 0011	c	63	0110 0011
...	...	...	...	...	...
Y	59	0101 1001	y	79	0111 1001
Z	5A	0101 1010	z	7A	0111 1010



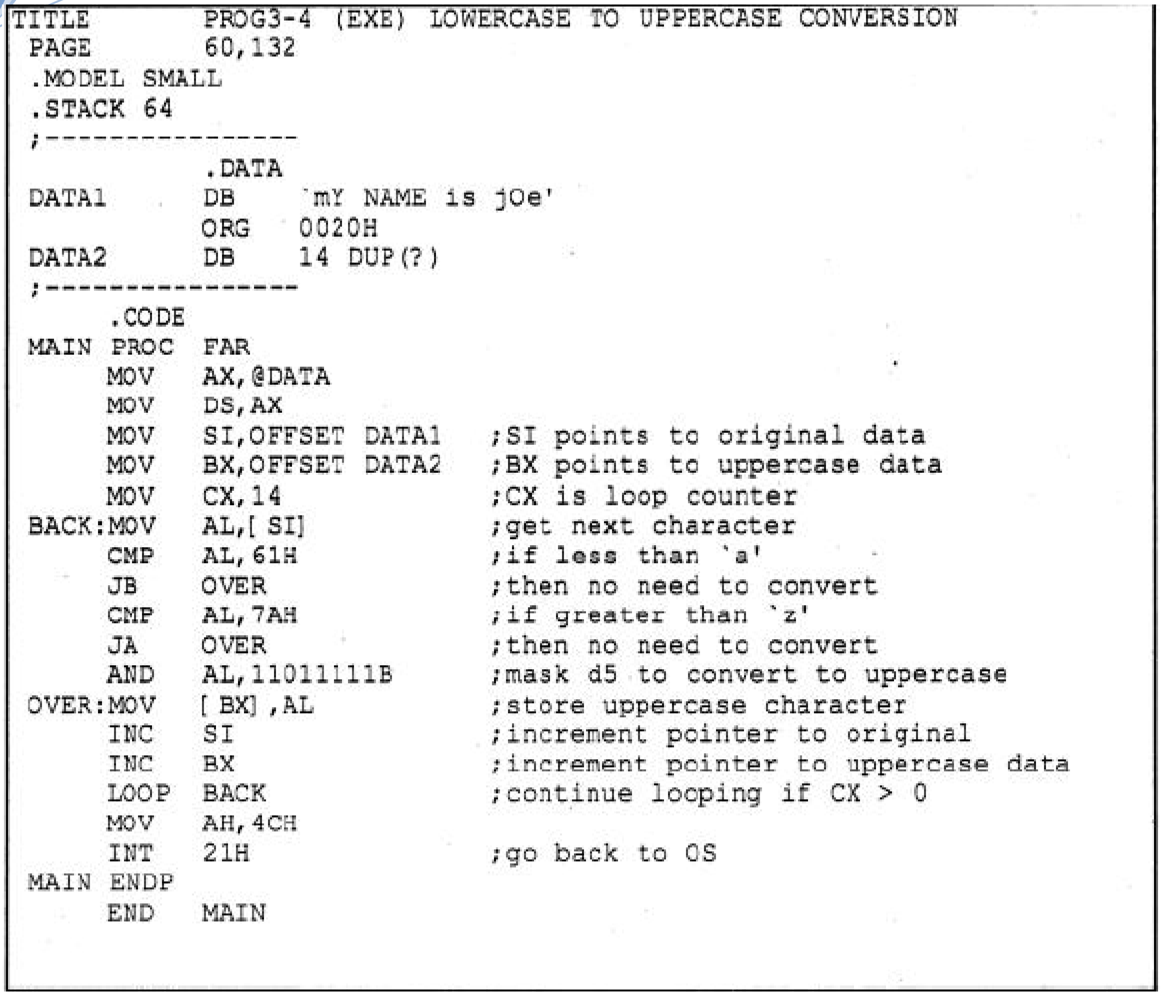
```
Count = 5
Highest = 0

REPEAT
  IF (Next > Highest)
  THEN
      Highest = Next
  ENDIF
  Decrement Count
UNTIL Count = 0

Store Highest
```

Fig: Flowchart and Pseudocode for Program 3-3

The following Program uses the CMP instruction to determine if an ASCII character is uppercase or lowercase.

- ✓ The following Program first detects if the letter is in lowercase, and if it is, it is ANDed wit h 1101 1111B = DFH. Otherwise, it is simply left alone.
- ✓ To determine if it is a lowercase letter, it is compared with 61H and 7AH to see if it is in the range a to z. Anything above or below this range should be left alone.

In the following Program, 20H could have been subtracted from the lowercase letters instead of ANDing with 1101 1111B.



```
TITLE       PROG3-4 (EXE) LOWERCASE TO UPPERCASE CONVERSION
 PAGE       60,132
 .MODEL SMALL
 .STACK 64
 ;----------------
            .DATA
 DATA1      DB     'mY NAME is jOe'
            ORG    0020H
 DATA2      DB     14 DUP(?)
 ;----------------
       .CODE
 MAIN PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      MOV   SI,OFFSET DATA1    ;SI points to original data
      MOV   BX,OFFSET DATA2    ;BX points to uppercase data
      MOV   CX,14              ;CX is loop counter
 BACK:MOV   AL,[SI]            ;get next character
      CMP   AL,61H             ;if less than 'a'
      JB    OVER               ;then no need to convert
      CMP   AL,7AH             ;if greater than 'z'
      JA    OVER               ;then no need to convert
      AND   AL,11011111B       ;mask d5 to convert to uppercase
 OVER:MOV   [BX],AL            ;store uppercase character
      INC   SI                 ;increment pointer to original
      INC   BX                 ;increment pointer to uppercase data
      LOOP  BACK               ;continue looping if CX > 0
      MOV   AH,4CH
      INT   21H                ;go back to OS
 MAIN ENDP
      END   MAIN
```

Program 3-4

BCD AND ASCII CONVERSION:

- BCD (*binary coded decimal*) is needed because we use the digits 0 to 9 for numbers in everyday life. Binary representation of 0 to 9 is called BCD.
- In computer literature, one encounters two terms for BCD numbers: (1) unpacked BCD, and (2) packed BCD.

Digit	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Unpacked BCD:

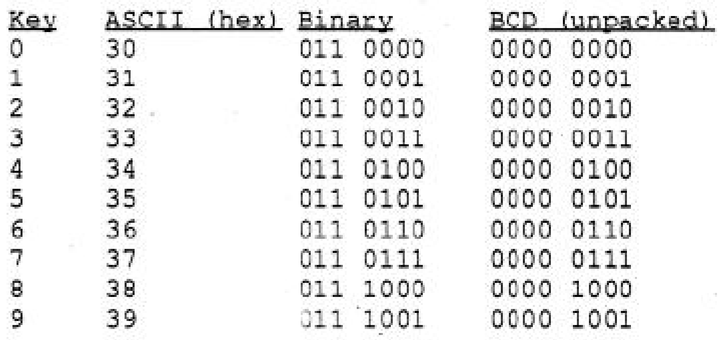
- In unpacked BCD, the lower 4 bits of the number represent the BCD number and the rest of the bits are 0.
    - Example: "0000 1001" and "0000 0101" are unpacked BCD for 9 and 5, respectively.
- In the case of unpacked BCD it takes 1 byte of memory location or a register of 8 bits to contain the number.

Packed BCD:

- In the case of packed BCD, a single byte has two BCD numbers in it, one in the lower 4 bits and one in the upper 4 bits.
  - For example, "0101 1001" is packed BCD for 59.
- It takes only 1 byte of memory to store the packed BCD operands. This is one reason to use packed BCD since it is twice as efficient in storing data.

ASCII Numbers:

- In ASCII keyboards, when key "0" is activated, for example, "011 0000" (30H) is provided to the computer. In the same way, 31H (011 0001) is provided for key "1", and so on, as shown in the following list:



Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

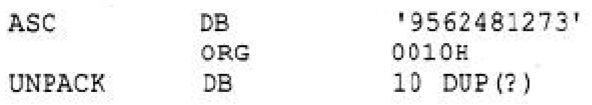
It must be noted that, although ASCII is standard in many countries, BCD numbers have universal application. So, the data conversion from ASCII to BCD and vice versa should be studied.

ASCII to BCD Conversion:

To process data in BCD, first the ASCII data provided by the keyboard must be converted to BCD. Whether it should be converted to packed or unpacked BCD depends on the instructions to be used.

ASCII to Unpacked BCD Conversion:

To convert ASCII data to BCD, the programmer must get rid of the tagged "011" in the higher 4 bits of the ASCII. To do that, each ASCII number is ANDed with "0000 1111" (0FH), as shown in the next example. These programs show three different methods for converting the 10 ASCII digits to unpacked BCD. All use the same data segment:



```
ASC         DB          '9562481273'
            ORG         0010H
UNPACK      DB          10 DUP(?)
```

The data is defined as DB.

- In the following Program 3-5a; the data is accessed in word-sized chunks.
- The Program 3-5b used the PTR directive to access the data.

- The Program 5-3c uses the based addressing mode (BX+ASC is used as a pointer.

```
        MOV   CX,5
        MOV   BX,OFFSET ASC       ;BX points to ASCII data
        MOV   DI,OFFSET UNPACK    ;DI points to unpacked BCD data
AGAIN:  MOV   AX,[BX]             ;move next 2 ASCII numbers to AX
        AND   AX,0F0FH            ;remove ASCII 3s
        MOV   [DI],AX             ;store unpacked BCD
        ADD   DI,2                ;point to next unpacked BCD data
        ADD   BX,2                ;point to next ASCII data
        LOOP  AGAIN
```

Program 3-5a

```
        MOV   CX,5                ;CX is loop counter
        MOV   BX,OFFSET ASC       ;BX points to ASCII data
        MOV   DI,OFFSET UNPACK    ;DI points to unpacked BCD data
AGAIN:  MOV   AX,WORD PTR [BX]    ;move next 2 ASCII numbers to AX
        AND   AX,0F0FH            ;remove ASCII 3s
        MOV   WORD PTR [DI],AX    ;store unpacked BCD
        ADD   DI,2                ;point to next unpacked BCD data
        ADD   BX,2                ;point to next ASCII data
        LOOP  AGAIN
```

Program 3-5b

```
        MOV   CX,10               ;load the counter
        SUB   BX,BX               ;clear BX
AGAIN:  MOV   AL,ASC[BX]          ;move to AL content of mem [BX+ASC]
        AND   AL,0FH              ;mask the upper nibble
        MOV   UNPACK[BX],AL       ;move to mem [BX+UNPACK] the AL
        INC   BX                  ;point to next byte
        LOOP  AGAIN               ;loop until it is finished
```

Program 3-5c

ASCII to Packed BCD Conversion:

To convert ASCII to packed BCD, it is first converted to unpacked BCD (to get rid of the 3) and then combined to make packed BCD.

For example, for 9 and 5 the keyboard gives 39 and 35, respectively. The goal is to produce 95H or"1001 0101", which is called packed BCD. This process is illustrated in detail below:

Key	ASCII	Unpacked BCD	Packed BCD
4	34	00000100	
7	37	00000111	01000111 or 47H

```
             ORG   0010H
VAL_ASC      DB    '47'
VAL_BCD      DB    ?
;reminder:   DB will put 34 in 0010H location and 37 in 0011H
             MOV   AX,WORD PTR VAL_ASC     ;AH=37,AL=34
             AND   AX,0F0FH                ;mask 3 to get unpacked BCD
             XCHG  AH,AL                   ;swap AH and AL.
             MOV   CL,4                    ;CL=04 to shift 4 times
             SHL   AH,CL                   ;shift left AH to get AH=40H
             OR    AL,AH                   ;OR them to get packed BCD
             MOV   VAL_BCD,AL              ;save the result
```

After this conversion, the packed BCD numbers are processed and the result will be in packed BCD format. There are special instructions, such as DAA and DAS, which require that the data be in packed BCD form and give the result in packed BCD.

- For the result to be displayed on the monitor or be printed by the printer, it must be in ASCII format. Conversion from packed BCD to ASCII is discussed next.

Packed BCD to ASCII Conversion:

To convert packed BCD to ASCII, it must first be converted to unpacked and then the unpacked BCD is tagged with 011 0000 (30H).

The following shows the process of converting from packed BCD to ASCII:

```
            Packed BCD    Unpacked BCD              ASCII
            29H           02H        & 09H          32H        & 39H
            0010 1001     0000 0010 & 0000 1001     011 0010 & 011 1001

      VAL1_BCD    DB      29H
      VAL3-ASC    DW      ?
                  ...
                  MOV     AL,VAL1_BCD
                  MOV     AH,AL        ;copy AL to AH. now AH=29,AL=29H
                  AND     AX,0F00FH    ;mask 9 from AH and 2 from AL
                  MOV     CL,4         ;CL=04 for shift
                  SHR     AH,CL        ;shift right AH to get unpacked BCD
                  OR      AX,3030H     ;combine with 30 to get ASCII
                  XCHG    AH,AL        ;swap for ASCII storage convention
                  MOV     VAL3_ASC,AX  ;store the ASCII
```

- After learning bow to convert ASCII to BCD, the application of BCD numbers is the next step.
- There are two instructions that deal specifically with BCD numbers: DAA and DAS.

BCD Addition and Correction:

In BCD addition, after adding packed BCD numbers, the result is no longer BCD. Look at this example:

```
MOV AL,17H
ADD AL,28H
```

Adding them gives 0011 1111B (3FH), which is not BCD! A BCD number can- only have digits from 0000 to 1001 (or 0 to 9). The result above should have been 17+ 28 = 45 (0100 0101).

✓ To correct this problem, the programmer must add 6 (0110) to the low digit: 3F + 06 = 45H.

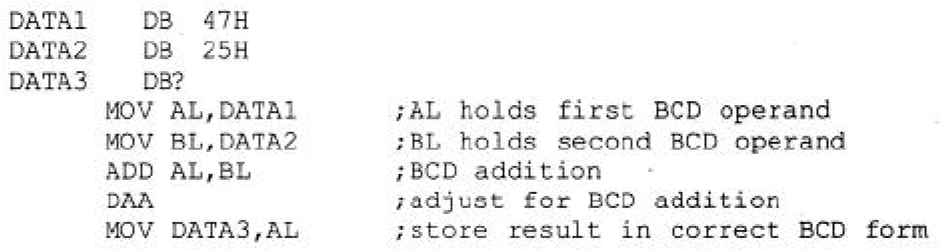
The same problem could have happened in the upper digit (for example, in 52H + 87H = D9H).

✓ Again to solve this problem, 6 must be added to the upper digit (D9H + 60H = 139H), to ensure that the result is BCD (52 + 87 = 139).

DAA

The DAA (*decimal adjust for addition*) instruction in x86 microprocessors is provided exactly for the purpose of correcting the problem associated with BCD addition. DAA will add 6 to the lower nibble or higher nibble if needed; otherwise, it will leave the result alone.

The following example will clarify these points:



```
DATA1    DB  47H
DATA2    DB  25H
DATA3    DB?
         MOV AL,DATA1        ;AL holds first BCD operand
         MOV BL,DATA2        ;BL holds second BCD operand
         ADD AL,BL           ;BCD addition
         DAA                 ;adjust for BCD addition
         MOV DATA3,AL        ;store result in correct BCD form
```

After the program is executed, the DATA3 field will contain 72H (47 + 25 =72).

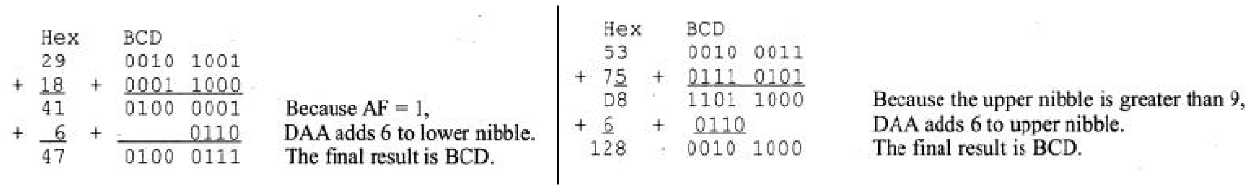
- ✓ Note that DAA works only on AL. In other words, while the source can be an operand of any addressing mode, the destination must be AL in order for DAA to work.
- ✓ It needs to be emphasized that DAA must be used after the addition of BCD operands and that BCD operands can never have any digit greater than 9. In other words, no A-F digit is allowed.
- ✓ It is also important to note that DAA works only after an ADD instruction; it will not work after the INC instruction.

Summary of DAA Action:

1. If after an ADD or ADC instruction the lower nibble (4 bits) is greater than 9, or if AF = 1, add 0110 to the lower 4 bits.
2. If the upper nibble is greater than 9, or if CF = 1, add 0110 to the upper nibble.

In reality there is no other use for the AF (auxiliary flag) except for BCD addition and correction. For example, adding 29H and 18H will result in 41H, which is incorrect as far as BCD is concerned.

See the following code:



```
  Hex     BCD
  29      0010 1001
+ 18    + 0001 1000
  41      0100 0001
+  6    +      0110
  47      0100 0111
```

Because AF = 1,
DAA adds 6 to lower nibble.
The final result is BCD.

```
  Hex     BCD
  53      0010 0011
+ 75    + 0111 0101
  D8      1101 1000
+ 6     + 0110
 128      0010 1000
```

Because the upper nibble is greater than 9,
DAA adds 6 to upper nibble.
The final result is BCD.

The above example shows that 6 is added to the upper nibble due to the fact it is greater than 9.

```
Eg1:                             ; AL = 0011 1001 = 39 BCD
                                 ; CL = 0001 0010 = 12 BCD
        ADD AL, CL               ; AL = 0100 1011 = 4BH
        DAA                      ; Since 1011 > 9; Add correction factor 06.
                                 ; AL = 0101 0001 = 51 BCD
```

```
Eg2:                                  ; AL = 1001 0110 = 96 BCD
                                      ; BL = 0000 0111 = 07 BCD
          ADD AL, BL                  ; AL = 1001 1101 = 9DH
          DAA                         ; Since 1101 > 9; Add correction factor 06
                                      ; AL = 1010 0011 = A3H
                                      ; Since 1010 > 9; Add correction factor 60
                                      ; AL = 0000 0011 = 03 BCD. The result is 103.
```

More Examples:

1: Add decimal numbers 22 and 18.

```
MOV AL, 22H              ; (AL)= 22H
ADD AL, 18H              ; (AL) = 3AH Illegal, incorrect answer!
DAA                      ; (AL) = 40H   Just treat it as decimalwith CF = 0

  3AH                    In this case, DAA same as ADD AL, 06H
+06H                     When LS hex digit in AL is >9, add 6 to it
=40H
```

2: Add decimal numbers 93 and 34.

```
MOV AL, 93H              ; (AL)= 93H
ADD AL, 34H              ; (AL) = C7H, CF = 0   Illegal & Incorrect!
DAA                      ; (AL) = 27H   Just treat it as decimal with CF = 1

  C7H                    In this case, DAA same as ADD AL, 60H
+60H                     When MS hex digit in AL is >9, add 6 to it
=27H
```

3: Add decimal numbers 93 and 84.

```
MOV AL, 93H             ; (AL)= 93H
ADD AL, 84H             ; (AL) = 17H, CF = 1   Incorrect answer!

DAA                     ; (AL) = 77H   Just treat it as decimal with CF = 1 (carry generated?)

  17H                   In this case, DAA same as ADD AL, 60H
+60H                    When CF = 1, add 6 to MS hex digit of AL and treat
=77H                    Carry as 1 even though not generated in this addition
```

4: Add decimal numbers 65 and 57.

```
MOV AL, 65H             ; (AL)= 65H
ADD AL, 57H             ; (AL) = BCH
DAA                     ; (AL) = 22H   Just treat it as decimal with CF = 1

  BCH                   In this case, DAA same as ADD AL, 66H
+66H
=22H  CF = 1
```

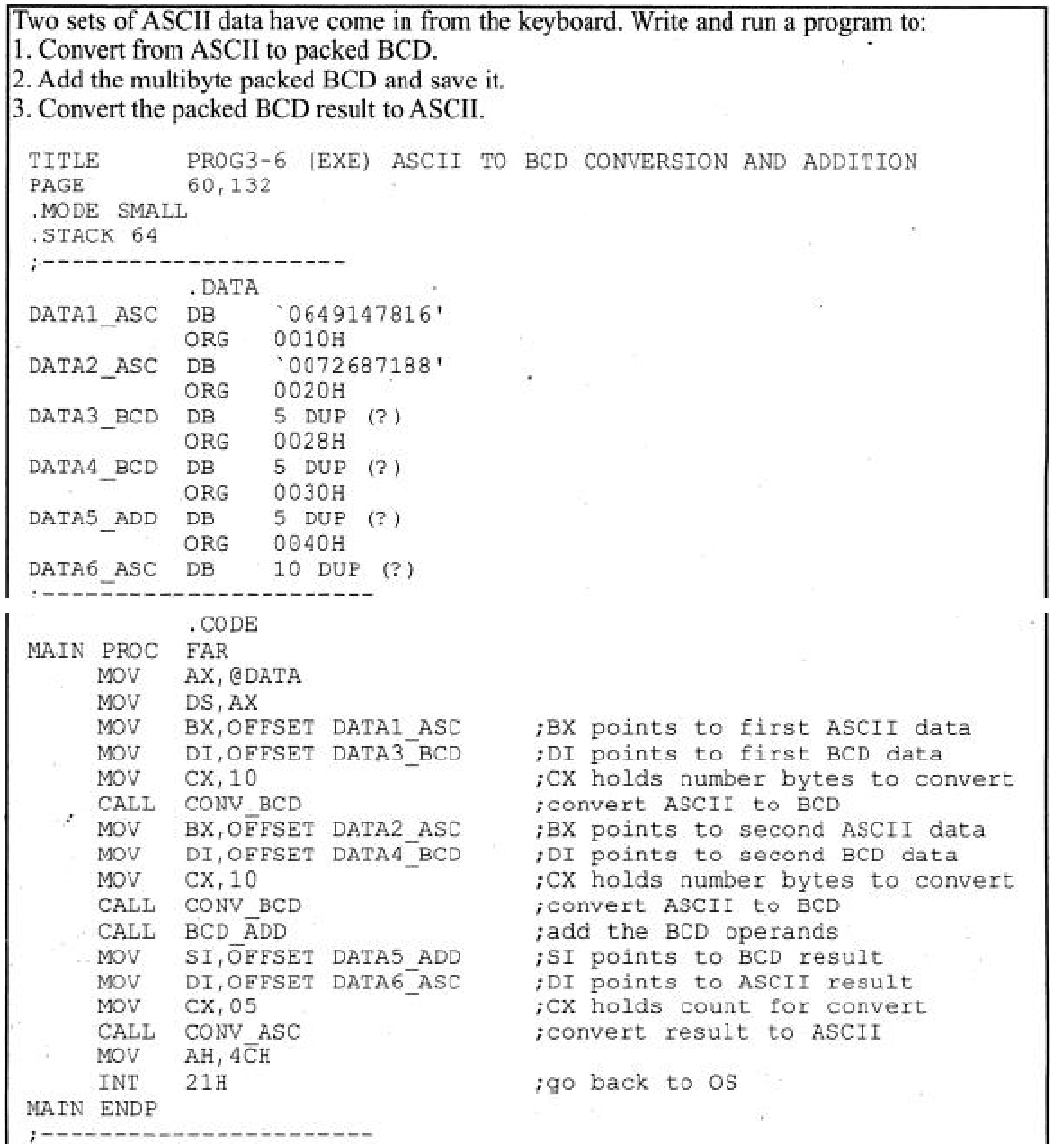
5: Add decimal numbers 99 and 28.

```
MOV AL, 99H             ; (AL)= 99H
ADD AL, 28H             ; (AL) = C1H, AF = 1
DAA                     ; (AL) = 27H   Just treat it as decimal with CF = 1

  C1H                   In this case, DAA same as ADD AL, 66H
+66H                    6 added to LS hex digit of AL, as AF = 1
=27H  CF = 1            6 added to MS hex digit of AL, as it is >9
```

6: Add decimal numbers 36 and 42.

```
MOV AL, 36H             ; (AL)= 36H
ADD AL, 42H             ; (AL) = 78H
DAA                     ; (AL) = 78H   Just treat it as decimal with CF = 0

  78H
+00H                    In this case, DAA same as ADD AL, 00H
=78H
```

The following Program demonstrates the use of DAA after addition of multibyte packed BCD numbers.



Two sets of ASCII data have come in from the keyboard. Write and run a program to:
1. Convert from ASCII to packed BCD.
2. Add the multibyte packed BCD and save it.
3. Convert the packed BCD result to ASCII.

```
TITLE      PROG3-6 (EXE) ASCII TO BCD CONVERSION AND ADDITION
PAGE       60,132
.MODE SMALL
.STACK 64
;----------------------
           .DATA
DATA1_ASC  DB     `0649147816'
           ORG    0010H
DATA2_ASC  DB     `0072687188'
           ORG    0020H
DATA3_BCD  DB     5 DUP (?)
           ORG    0028H
DATA4_BCD  DB     5 DUP (?)
           ORG    0030H
DATA5_ADD  DB     5 DUP (?)
           ORG    0040H
DATA6_ASC  DB     10 DUP (?)
;-----------------------
           .CODE
MAIN  PROC FAR
      MOV   AX,@DATA
      MOV   DS,AX
      MOV   BX,OFFSET DATA1_ASC    ;BX points to first ASCII data
      MOV   DI,OFFSET DATA3_BCD    ;DI points to first BCD data
      MOV   CX,10                  ;CX holds number bytes to convert
      CALL  CONV_BCD               ;convert ASCII to BCD
      MOV   BX,OFFSET DATA2_ASC    ;BX points to second ASCII data
      MOV   DI,OFFSET DATA4_BCD    ;DI points to second BCD data
      MOV   CX,10                  ;CX holds number bytes to convert
      CALL  CONV_BCD               ;convert ASCII to BCD
      CALL  BCD_ADD                ;add the BCD operands
      MOV   SI,OFFSET DATA5_ADD    ;SI points to BCD result
      MOV   DI,OFFSET DATA6_ASC    ;DI points to ASCII result
      MOV   CX,05                  ;CX holds count for convert
      CALL  CONV_ASC               ;convert result to ASCII
      MOV   AH,4CH
      INT   21H                    ;go back to OS
MAIN  ENDP
;-----------------------
```

```
;THIS SUBROUTINE CONVERTS ASCII TO PACKED BCD
CONV_BCD PROC
AGAIN:      MOV   AX,[BX]      ;BX=pointer for ASCII data
       XCHG  AH,AL
       AND   AX,0F0FH     ;mask ASCII 3s
       PUSH  CX           ;save the counter
       MOV   CL,4         ;shift AH left 4 bits
       SHL   AH,CL        ;to get ready for packing
       OR    AL,AH        ;combine to make packed BCD
       MOV   [DI],AL            ;DI=pointer for BCD data
       ADD   BX,2         ;point to next 2 ASCII bytes
       INC   DI           ;point to next BCD data
       POP   CX           ;restore loop counter
       LOOP  AGAIN
       RET
CONV_BCD ENDP
;----------------
;THIS SUBROUTINE ADDS TWO MULTIBYTE PACKED BCD OPERANDS
BCD_ADD PROC
      MOV  BX,OFFSET DATA3_BCD     ;BX=pointer for operand 1
      MOV  DI,OFFSET DATA4_BCD     ;DI=pointer for operand 2
      MOV  SI,OFFSET DATA5_ADD     ;SI=pointer for sum
      MOV  CX,05
      CLC
BACK: MOV  AL,[BX]+4    ;get next byte of operand 1
      ADC  AL,[DI]+4    ;add next byte of operand 2
      DAA               ;correct for BCD addition
      MOV  [SI]+4,AL    ;save sum
      DEC  BX           ;point to next byte of operand 1
      DEC  DI           ;point to next byte of operand 2
      DEC  SI           ;point to next byte of sum
      LOOP      BACK
      RET
BCD_ADD ENDP
;-----------------
;THIS SUBROUTINE CONVERTS FROM PACKED BCD TO ASCII
CONV_ASC PROC
AGAIN2: MOV AL,[SI]     ;SI=pointer for BCD data
     MOV   AH,AL        ;duplicate to unpack
     AND   AX,0F00FH    ;unpack
     PUSH  CX           ;save counter
     MOV   CL,04        ;shift right 4 bits to unpack
     SHR   AH,CL        ;the upper nibble
     OR    AX,3030H     ;make it ASCII
     XCHG  AH,AL        ;swap for ASCII storage convention
     MOV   [DI],AX            ;store ASCII data
     INC   SI           ;point to next BCD data
     ADD   DI,2         ;point to next ASCII data
     POP   CX           ;restore loop counter
     LOOP  AGAIN2
     RET
CONV_ASC ENDP
     END   MAIN
```

Program 3-6

BCD Subtraction and Correction:

The problem associated with the addition of packed BCD numbers also shows up in subtraction. Again, there is an instruction (DAS) specifically designed to solve the problem.

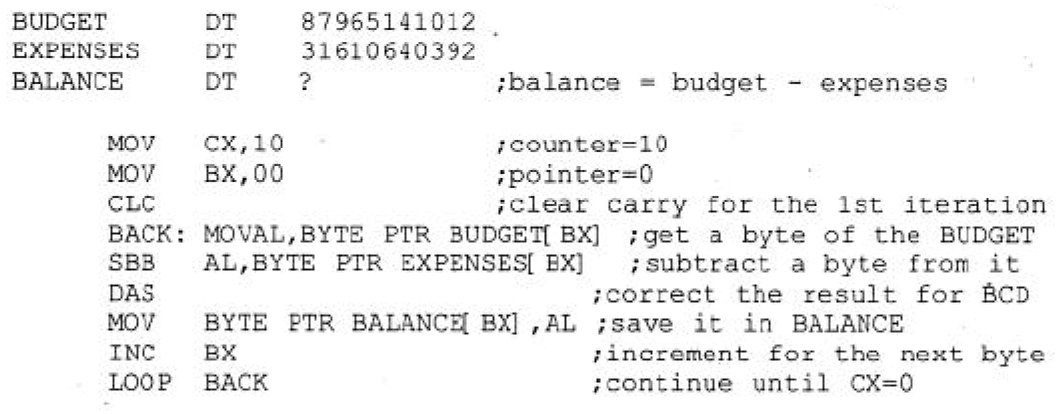
Therefore, when subtracting packed BCD (single-byte or multibyte) operands, the DAS instruction is put after the SUB or SBB instruction. AL must be used as the destination register to make DAS work.

Summary of DAS Action:

1. If after a SUB or SBB instruction the lower nibble is greater than 9, or if AF = 1 , subtract 0110 from the lower 4 bits.
2. If the upper nibble is greater than 9, or CF = 1, subtract 0110 from the upper nibble.

Due to the widespread use of BCD numbers, a specific data directive, DT, has been created. DT can be used to represent BCD numbers from 0 to $10^{20} – 1$ (that is, twenty 9s).

Assume that the following operands represent the budget, the expenses, and the balance, which is the budget minus the expenses.



```
BUDGET      DT    87965141012
EXPENSES    DT    31610640392
BALANCE     DT    ?             ;balance = budget - expenses

      MOV   CX,10               ;counter=10
      MOV   BX,00               ;pointer=0
      CLC                       ;clear carry for the 1st iteration
      BACK: MOVAL,BYTE PTR BUDGET[ BX]  ;get a byte of the BUDGET
      SBB   AL,BYTE PTR EXPENSES[ BX]   ;subtract a byte from it
      DAS                             ;correct the result for BCD
      MOV   BYTE PTR BALANCE[ BX] ,AL ;save it in BALANCE
      INC   BX                        ;increment for the next byte
      LOOP  BACK                      ;continue until CX=0
```

Notice in the code section above that,

- ✓ no H (hex) indicator is needed for BCD numbers when using the DT directive, and
- ✓ the use of the based relative addressing mode (BX + displacement) allows access to all three arrays with a single register BX.

```
Eg1:                                ; AL = 0011 0010 = 32 BCD
                                    ; CL = 0001 0111 = 17 BCD
          SUB AL, CL                ; AL = 0001 1011 = 1BH
          DAS                       ; Subtract 06, since 1011 > 9.
                                    ; AL = 0001 0101 = 15 BCD

Eg2:                                ; AL = 0010 0011 = 23 BCD
                                    ; CL = 0101 1000 =58 BCD
          SUB AL, CL                ; AL = 1100 1011 = CBH
          DAS                       ; Subtract 66, since 1100 >9 & 1011 > 9.
                                    ; AL = 0110 0101 = 65 BCD, CF = 1.
                      ; Since CF = 1, answer is – 65.
```

More Examples:

1: Subtract decimal numbers 45 and 38.

```
MOV AL, 45H          ; (AL)= 45H
SUB AL, 38H          ; (AL) = 0DH   Illegal, incorrect answer!
 DAS                 ; (AL) = 07H   Just treat it as decimal  with Cy = 0

  0DH                In this case, DAS same as SUB AL, 06H
-06H                 When LS hex digit in AL is >9, subtract 6
=07H
```

2: Subtract decimal numbers 63 and 88.

```
MOV AL, 63H          ; (AL)= 63H
SUB AL, 88H          ; (AL) = DBH, Cy=1   Illegal & Incorrect!
DAS                  ; (AL) = 75H   Just treat it as decimal with Cy = 1 (carry generated?)

  DBH                In this case, DAS same as SUB AL, 66H
-66H                 When Cy = 1, it means result is negative
=75H                 Result is 75, which is 10's complement of 25
                     Treat Cy as 1 as Cy was generated in the previous subtraction itself!
```

3: Subtract decimal numbers 45 and 52.

```
MOV AL, 45H          ; (AL)= 45H
SUB AL, 52H          ; (AL) = F3H, Cy = 1   Incorrect answer!
DAS                  ; (AL) = 93H   Just treat it as decimal with Cy = 1 (carry generated?)

  F3H                In this case, DAS same as SUB AL, 60H
-60H                 When Cy = 1, it means result is negative
=93H                 Result is 93, which is 10's complement of 07
```

4: Subtract decimal numbers 50 and 19.

```
MOV AL, 50H          ; (AL)= 50H
SUB AL, 19H          ; (AL) = 37H, Ac = 1
DAS                  ; (AL) = 31H   Just treat it as decimal with Cy =0

  37H                In this case, DAS same as SUB AL, 06H
-06H                 06H is subtracted from AL as Ac = 1
=31H
```

5: Subtract decimal numbers 99 and 88.

```
MOV AL, 99H          ; (AL)= 99H
SUB AL, 88H          ; (AL) = 11H
DAS                  ; (AL) = 11H   Just treat it as decimal with Cy = 0

  11H                In this case, DAS same as SUB AL, 00H
-00H
=11H
```

6: Subtract decimal numbers 14 and 92.

```
MOV AL, 14H          ; (AL)= 14H
SUB AL, 92H          ; (AL) = 82H, Cy = 1
DAS                  ; (AL) = 22H   Just treat it as decimal with Cy = 1

  82H                In this case, DAS same as SUB AL, 60H
-60H                 60H is subtracted from AL as Cy = 1
=22H                 22 is 10's complement of 78
```

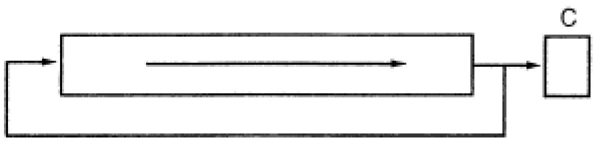
ROTATE INSTRUCTIONS:

In many applications there is a need to perform a bitwise rotation of an operand. The rotation instructions ROR, ROL and RCR, RCL are designed specifically for that purpose. They allow a program to rotate an operand right or left.

- In rotate instructions, the operand can be in a register or memory. If the number of times an operand is to be rotated is more than 1, this is indicated by CL. This is similar to the shift instructions.
- There are two types of rotations. One is a simple rotation of the bits of the operand, and the other is a rotation through the carry.

ROR (rotate right)

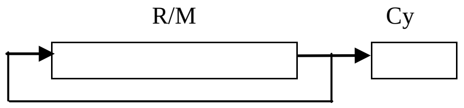
In rotate right, as bits are shifted from left to right they exit from the right end (LSB) and enter the left end (MSB). In addition, as each bit exits the LSB, a copy of it is given to the carry flag. In other words, in ROR, the LSB is moved to the MSB and is also copied to CF, as shown in the diagram.



If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, register CL is used to hold the number of times it is to be rotated.

Eg:

ROR BH, 1



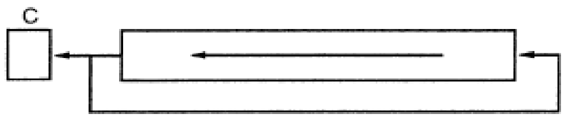
Rotate right without Cy

	Before	*After*
BH	0100 0010	0010 0001
Cy	1	0

```
          MOV   AL,36H      ;AL=0011 0110
          ROR   AL,1        ;AL=0001 1011  CF=0
          ROR   AL,1        ;AL=1000 1101  CF=1
          ROR   AL,1        ;AL=1100 0110  CF=1
;or:
          MOV   AL,36H      ;AL=0011 0110
          MOV   CL,3        ;CL=3 number of times to rotate
          ROR   AL,CL       ;AL=1100 0110 CF=1
;the operand can be a word:
          MOV   BX,0C7E5H   ;BX=1100 0111 1110 0101
          MOV   CL,6        ;CL=6 number of times to rotate
          ROR   BX,CL       ;BX=1001 0111 0001 1111 CF=1
```

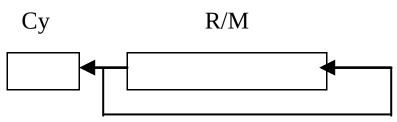
ROL (rotate left)

In rotate left, as bits are shifted from right to left they exit the left end (MSB) and enter the right end (LSB). In addition, every bit that leaves the MSB is copied to the carry flag. In other words, in ROL the MSB is moved to the LSB and is also copied to CF, as shown in the diagram.



If the operand is to be rotated once, the 1 is coded. Otherwise, the number of times it is to be rotated is in CL. **Eg:**

ROL BH, CL



Rotate left without Cy

	Before	*After*
BH	0010 0010	1000 1000
CL	02H	
Cy	1	0

```
        MOV    BH,72H        ;BH=0111 0010
        ROL    BH,1          ;BH=1110 0100  CF=0
        ROL    BH,1          ;BH=1100 1001  CF=1
        ROL    BH,1          ;BH=1001 0011  CF=1
        ROL    BH,1          ;BH=0010 0111  CF=1
;or:
        MOV    BH,72H        ;BH=0111 0010
        MOV    CL,4          ;CL=4 number of times to rotate
        ROL    BH,CL         ;BH=0010 0111  CF=1

;The operand can be a word:
        MOV    DX,672AH      ;DX=0110 0111 0010 1010
        MOV    CL,3          ;CL=3 number of times to rotate
        ROL    DX,CL ;DX=0011 1001 0101 0011 CF=1
```

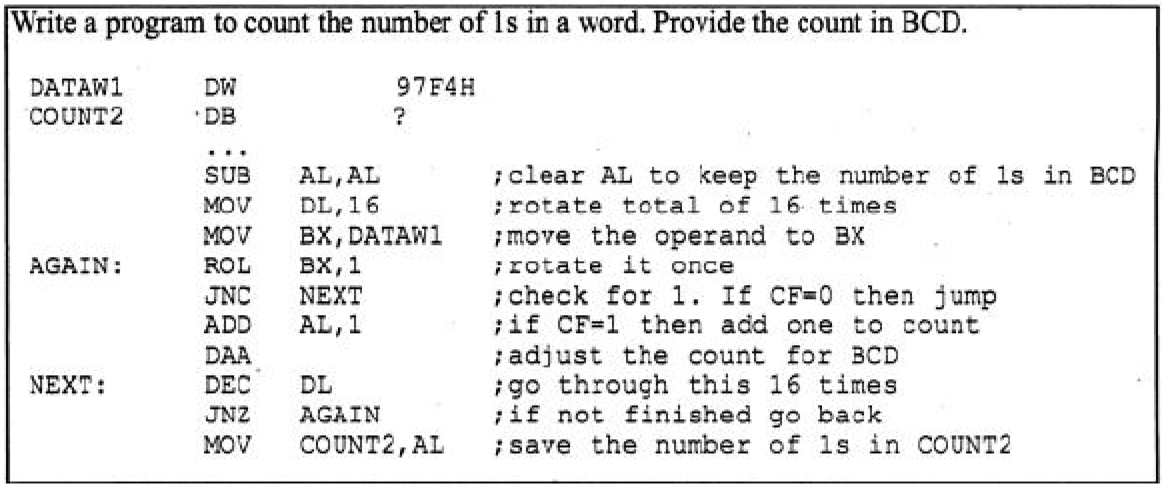
The following Program shows an application of the rotation instruction. The maximum count in Program will be 8 since the program is counting the number of 1s in a byte of data. If the operand is a 16-bit word, the number of 1s can go as high as 16.

Write a program that finds the number of 1s in a byte.

```
;From the data segment:
DATA1      DB          97H
COUNT      DB          ?
;From the code segment:
           SUB    BL,BL        ;clear BL to keep the number of 1s
           MOV    DL,8         ;rotate total of 8 times
           MOV    AL,DATA1
AGAIN:     ROL    AL,1         ;rotate it once
           JNC    NEXT         ;check for 1
           INC    BL           ;if CF=1 then add one to count
NEXT:      DEC    DL           ;go through this 8 times
           JNZ    AGAIN        ;if not finished go back
           MOV    COUNT,BL     ;save the number of 1s
```

Program 3-7

The Program is similar to the previous one, rewritten for a word-sized operand. It also provides the count in BCD format instead of hex. Reminder: AL is used to make a BCD counter because the because, the DAA instruction works only on AL.



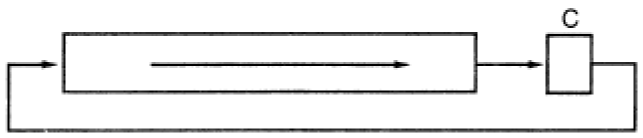
Write a program to count the number of 1s in a word. Provide the count in BCD.

```
DATAW1     DW          97F4H
COUNT2     DB          ?
           ...
           SUB   AL,AL       ;clear AL to keep the number of 1s in BCD
           MOV   DL,16       ;rotate total of 16 times
           MOV   BX,DATAW1   ;move the operand to BX
AGAIN:     ROL   BX,1        ;rotate it once
           JNC   NEXT        ;check for 1. If CF=0 then jump
           ADD   AL,1        ;if CF=1 then add one to count
           DAA               ;adjust the count for BCD
NEXT:      DEC   DL          ;go through this 16 times
           JNZ   AGAIN       ;if not finished go back
           MOV   COUNT2,AL   ;save the number of 1s in COUNT2
```

Program 3-8

RCR (rotate right through carry)

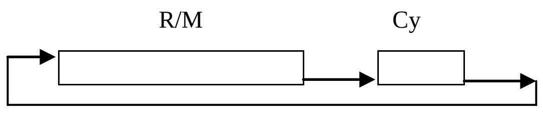
In RCR, as bits are shifted from left to right, they exit the right end (LSB) to the carry flag, and the carry flag enters the left end (MSB). In other words, in RCR the LSB is moved to CF and CF is moved to the MSB. In reality, CF acts as if it is part of the operand. This is shown in the diagram.



If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, the register CL holds the number of times.

Eg:

RCR BH, 1



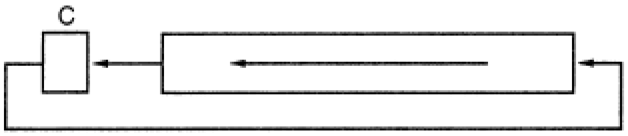
Rotate right with Cy

	Before	*After*
BH	0100 0010	1010 0001
Cy	1	0

```
        CLC                         ;make CF=0
        MOV    AL,26H               ;AL=0010 0110
        RCR    AL,1                 ;AL=0001 0011 CF=0
        RCR    AL,1                 ;AL=0000 1001 CF=1
        RCR    AL,1                 ;AL=1000 0100 CF=1
or:
        CLC                         ;make CF=0
        MOV    AL,26H               ;AL=0010 0110
        MOV    CL,3                 ;CL=3 number of times to rotate
        RCR    AL,CL                ;AL=1000 0100 CF=1

;the operand can be a word
        STC                         ;make CF=1
        MOV    BX,37F1H             ;BX=0011 0111 1111 0001
        MOV    CL,5                 ;CL=5 number of times to rotate
        RCR    BX,CL                ;BX=0001 1001 1011 1111 CF=0
```

RCL (rotate left through carry)

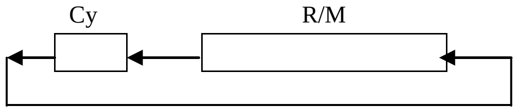
In RCL, as bits are shifted from right to left, they exit the left end (MSB) and enter the carry flag, and the carry flag enters the right end (LSB). In other words, in RCL the MSB is moved to CF and CF is moved to the LSB. In reality, CF acts as if it is part of the operand. This is shown in the following diagram.



If the operand is to be rotated once, the 1 is coded, but if it is to be rotated more than once, register CL holds the number of times.

Eg:

RCL BH, CL



Rotate left with Cy

	Before	*After*
BH	0010 0010	1000 1010
CL	02H	
Cy	1	0

```
        STC                         ;make CF=1
        MOV    BL,15H               ;BL=0001 0101
        RCL    BL,1                 ;0010 1011 CF=0
        RCL    BL,1                 ;0101 0110 CF=0
or:
        STC                         ;make CF=1
        MOV    BL,15H               ;BL=0001 0101
        MOV    CL,2                 ;CL=2 number of times for rotation
        RCL    BL,CL                ;BL=0101 0110 CF=0

;the operand can be a word:
        CLC                         ;make CF=0
        MOV    AX,191CH             ;AX=0001 1001 0001 1100
        MOV    CL,5                 ;CL=5 number of times to rotate
        RCL    AX,CL                ;AX=0010 0011 1000 0001 CF=1
```

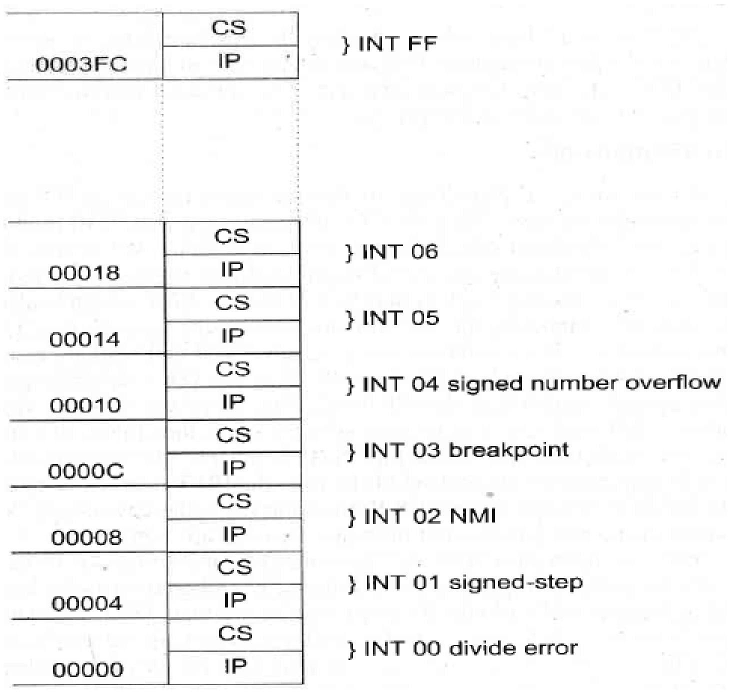
INTERRUPTS IN x86 PC

8088/86 INTERRUPTS

- An interrupt is an external event that informs the CPU that a device needs its service. In 8088/86, there are 256 interrupts: INT 00, INT 01, . . . , INT FF (sometimes called TYPEs).
- When an interrupt is executed, the microprocessor automatically saves the flag register (FR), the instruction pointer (IP), and the code segment register (CS) on the stack; and goes to a fixed memory location.
- In x86 PCs, the memory locations to which an interrupt goes is always four times the value of the interrupt number. For example, INT 03 will go to address 0000CH (4 * 3 = 12 = 0CH). The following Table is a partial list of the interrupt vector table.

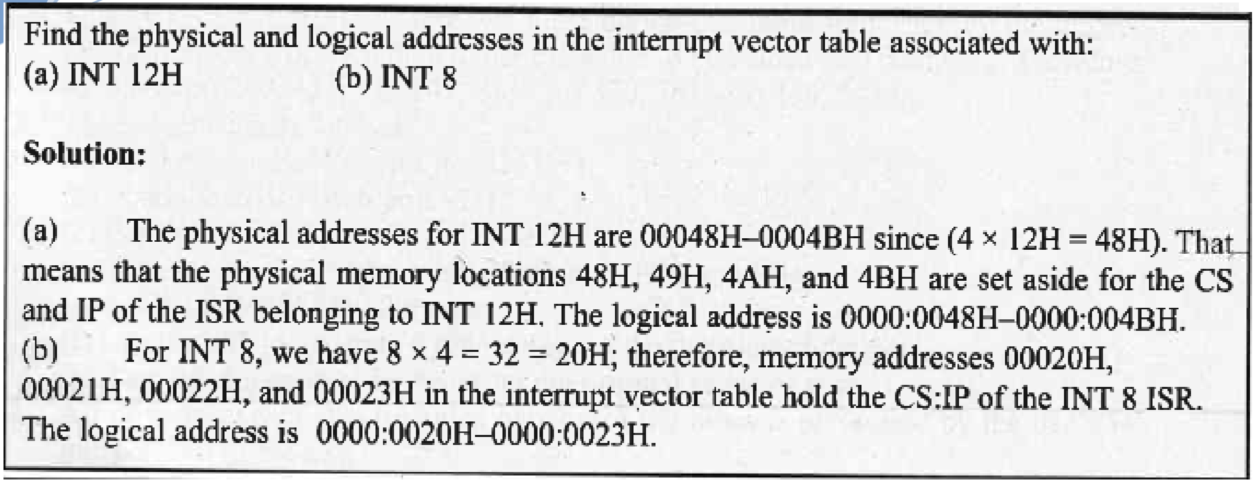
Table: Interrupt Vector

INT Number	Physical Address	Logical Address
INT 00	00000	0000 – 0000
INT 01	00004	0000 – 0004
INT 02	00008	0000 – 0008
INT 03	0000C	0000 – 000C
INT 04	00010	0000 – 0010
INT 05	00014	0000 – 0014
. . .	. . .	. . .
INT FF	003FC	0000 – 03FC



Interrupt Service Routine (ISR):

- ✓ For every interrupt there must be a program associated with it.
- ✓ When an interrupt is invoked, it is asked to run a program to perform a certain service. This program is commonly referred to as an *interrupt service routine* (*ISR*). The interrupt service routine is also called the *interrupt handler*.
- ✓ When an interrupt is invoked, the CPU runs the interrupt service routine. As shown in the above Table, for every interrupt there are allocated four bytes of memory in the interrupt vector table. Two bytes are for the IP and the other two are for the CS of the ISR.
- ✓ These four memory locations provide the addresses of the interrupt service routine for which the interrupt was invoked. Thus the lowest 1024 bytes (256 x 4 = 1024) of memory space are set aside for the interrupt vector table and must not be used for any other function.



Find the physical and logical addresses in the interrupt vector table associated with:
(a) INT 12H (b) INT 8

Solution:

(a) The physical addresses for INT 12H are 00048H–0004BH since (4 × 12H = 48H). That means that the physical memory locations 48H, 49H, 4AH, and 4BH are set aside for the CS and IP of the ISR belonging to INT 12H. The logical address is 0000:0048H–0000:004BH.
(b) For INT 8, we have 8 × 4 = 32 = 20H; therefore, memory addresses 00020H, 00021H, 00022H, and 00023H in the interrupt vector table hold the CS:IP of the INT 8 ISR. The logical address is 0000:0020H–0000:0023H.

Difference between INT and CALL Instructions:

The INT instruction saves the CS: IP of the following instruction and jumps indirectly to the subroutine associated with the interrupt. A CALL FAR instruction also saves the CS: IP and jumps to the desired subroutine (procedure).

The differences can be summarized as follows:

CALL Instruction	INT instruction
1. A CALL FAR instruction can jump to any location within the 1M byte address range of the 8088/86 CPU.	1. INT nn goes to a fixed memory location in the interrupt vector table to get the address of the interrupt service routine.
2. A CALL FAR instruction is used by the programmer in the sequence of instructions in the program.	2. An externally activated hardware interrupt can come-in at any time, requesting the attention of the CPU.
3. A CALL FAR instruction cannot be masked (disabled).	3. INT nn belonging to externally activated hardware interrupts can be masked.
4. A CALL FAR instruction automatically saves only CS: IP of the next instruction on the stack.	4. INT nn saves FR (flag register) in addition to CS: IP of the next instruction.
5. At the end of the subroutine that has been called by the CALL FAR instruction, the RETF (return FAR) is the last instruction. RETF pops CS and IP off the stack.	5. The last instruction in the interrupt service routine (ISR) for INT nn is the instruction IRET (interrupt return). IRET pops off the FR (flag register) in addition to CS and IP.

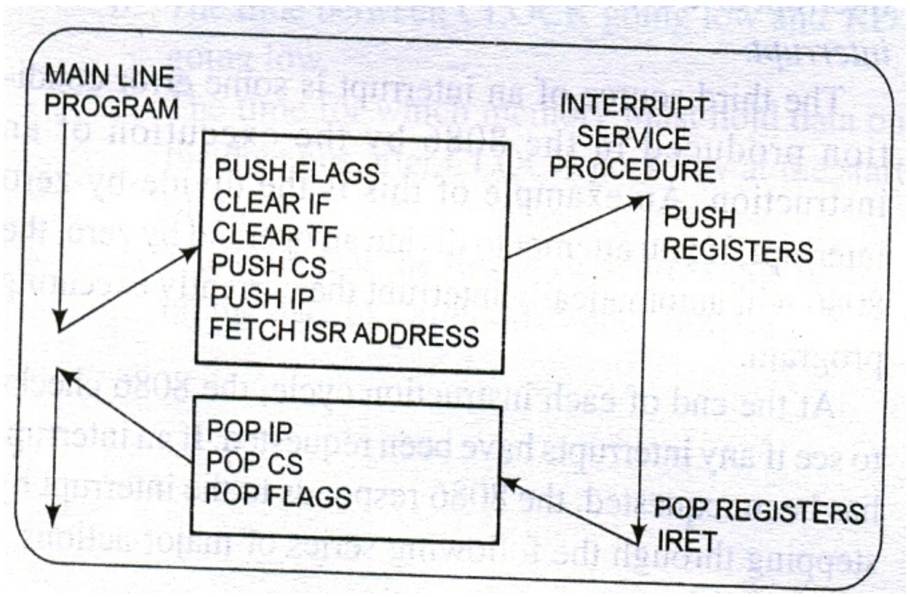
Processing Interrupts:

When the 8088/86 processes any interrupt (software or hardware), it goes through the following steps:

1. The flag register (FR) is pushed onto the stack and SP is decremented by 2, since FR is a 2-byte register.

2. IF (interrupt enable flag) and TF (trap flag) are both cleared (IF = 0 and TF = 0). This masks (causes the system to ignore) interrupt requests from the INTR pin and disables single stepping while the CPU is executing the interrupt service routine.
3. The current CS is pushed onto the stack and SP is decremented by 2.
4. The current IP is pushed onto the stack and SP is decremented by 2.
5. The INT number (type) is multiplied by 4 to get the physical address of the location within the vector table to fetch the CS and IP of the interrupt service routine.
6. From the new CS: IP, the CPU starts to fetch and execute instructions belonging to the ISR program.
7. The last instruction of the interrupt service routine must be IRET, to get IP, CS, and FR back from the stack and make the CPU run the code where it left off.

The following Figure summarizes these steps in diagram form.



Categories of Interrupts:

INT nn is a 2-byte instruction where the first byte is for the opcode and the second byte is the interrupt number. We can have a maximum of 256 (INT 00 INT FFH) interrupts. Of these 256 interrupts, some are used for software interrupts and some are for hardware interrupts.

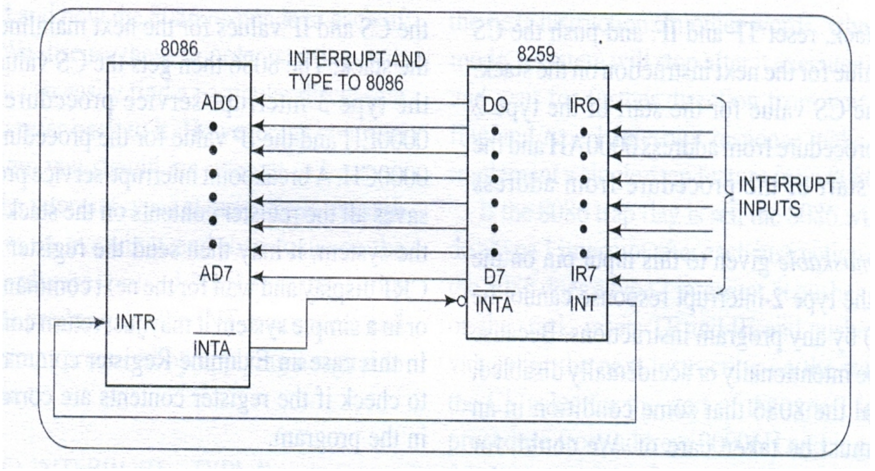
1. **Hardware Interrupts:**

- There are three pins in the x86 that are associated with hardware interrupts. They are INTR (interrupt request), NMI (non-maskable interrupt), and INTA (interrupt acknowledge).
- INTR is an input signal into the CPU, which can be masked (ignored) and unmasked through the use of instructions CLI (clear interrupt flag) and STI (set interrupt flag).

- If IF = 0 (in flag register), all hardware interrupt requests through INTR are ignored. This has no effect on interrupts coming from the NMI pin. The instruction CLI (clear interrupt flag) will make IF = 0.
- To allow interrupt request through the INTR pin, this flag must be set to one (IF = 1). The STI (set interrupt flag) instruction can be used to set IF to 1.
- NMI, which is also an input signal into the CPU, cannot be masked and unmasked using instructions CLI and STI; and for this reason it is called a *non-maskable interrupt*.

- INTR and NMI are activated externally by putting 5V on the pins of NMI and INTR of the x86 microprocessor.
- When either of these interrupts is activated, the x86 finishes the instruction that it is executing, pushes FR and the CS: IP of the next instruction onto the stack, then jumps to a fixed location in the interrupt vector table and fetches the CS: IP for the interrupt service routine (ISR) associated with that interrupt.
- At the end of the ISR, the IRET instruction causes the CPU to get (pop) back its original FR and CS: IP from the stack, thereby forcing the CPU to continue at the instruction where it left off when the interrupt came in.

- Intel has embedded "*INT 02*" into the x86 microprocessor to be used only for NMI.
- Whenever the NMI pin is activated, the CPU will go to memory location 00008 to get the address (CS: IP) of the interrupt service routine (ISR) associated with NMI.
- Memory locations 00008, 00009, 0000A, and 0000B contain the 4 bytes of CS: IP of the ISR belonging to NMI.
- The 8259 programmable interrupt controller (PIC) chip can be connected to INTR to expand the number of hardware interrupts to 64.



2. **Software Interrupts:**

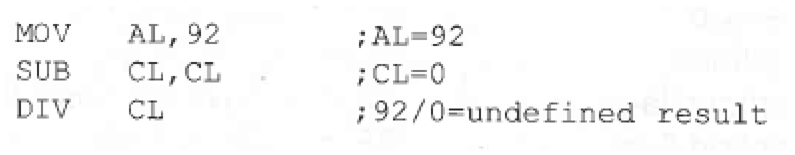
- If an ISR is called upon as a result of the execution of an x86 instruction such as "*INT nn*", it is referred to as software interrupt, since it was invoked from software, not from external hardware.
- Examples of such interrupts are DOS "*INT 21H*" function calls and video interrupts "*INT 10H*".
- These interrupts can be invoked in the sequence of code just like any other x86 instruction.
- Many of the interrupts in this category are used by the MS DOS operating system and IBM BIOS to perform essential tasks that every computer must provide to the system and the user.
- Within this group of interrupts there are also some *predefined functions* associated with some of the interrupts. They are "*INT 00*" (divide error), "*INT 01*" (single step), "*INT 03*" (breakpoint), and "*INT 04*" (signed number overflow). Each is described below.
- The rest of the interrupts from "*INT 05*" to "*INT FF*" can be used to implement either software or hardware interrupts.

Functions associated with INT 00 to INT 04:

Interrupts INT 00 to INT 04 have predefined tasks (functions) and cannot be used in any other way.

INT 00 (divide error)

- ✓ This interrupt belongs to the category of interrupts referred to as conditional or exception interrupts. Internally, they are invoked by the microprocessor whenever there are conditions (exceptions) that the CPU is unable to handle.
- ✓ One such situation is an attempt to divide a number by zero. Since the result of dividing a number by zero is undefined, and the CPU has no way of handling such a result, it automatically invokes the divide error exception interrupt.
- ✓ In the 8088/86 microprocessor, out of 256 interrupts, Intel has set aside only INT 0 for the exception interrupt.
- ✓ INT 00 is invoked by the microprocessor whenever there is an attempt to divide a number by zero.
- ✓ In the x86 PC, the service subroutine for this interrupt is responsible for displaying the message "DIVIDE ERROR" on the screen if a program such as the following is executed:



```
MOV    AL,92         ;AL=92
SUB    CL,CL         ;CL=0
DIV    CL            ;92/0=undefined result
```

- ✓ INT 0 is also invoked if the quotient is too large to fit into the assigned register when executing a DIV instruction. Look at the following case:

```
MOV   AX,0FFFFH    ;AX=FFFFH
MOV   BL,2         ;BL=2
DIV   BL           ;65535/2 = 32767 larger than 255
                   ;maximum capacity of AL
```

INT 01 (single step)

- ✓ In executing a sequence of instructions, there is a need to examine the contents of the CPU's registers and system memory. This is often done by executing the program one instruction at a time and then inspecting registers and memory. This is commonly referred to as single-stepping, or performing a trace.
- ✓ Intel has designated INT 01 specifically for implementation of single-stepping. To single-step, the trap flag (TF) (D8 of the flag register), must be set to 1. Then after execution of each instruction, the 8088/86 automatically jumps to physical location 00004 to fetch the 4 bytes for CS: IP of the interrupt service routine, which will dump the registers onto the screen.
- ✓ Intel has not provided any specific instruction for to set or reset (unlike IF, which uses STI and CLI instructions to set or reset), the TF; one can write a simple program to do that. The following shows how to make TF = 0:

```
PUSHF
POP    AX
AND    AX,1111111011111111B
PUSH   AX
POPF
```

- ✓ Recall that, TF is D8 of the flag register.
- ✓ To make TF = 1, one simply uses the OR instruction in place of the AND instruction above.

INT 02 (non-maskable interrupt)

- ✓ All Intel x86 microprocessors have a pin designated NMI. It is an active-high input. Intel has set aside INT 2 for the NMI interrupt. Whenever the NMI pin of the x86 is activated by a high (5 V) signal, the CPU jumps to physical memory location 00008 to fetch the CS: IP of the interrupt service routine associated with NMI.
- ✓ The NMI input is often used for major system faults, such as power failures. The NMI interrupt will be caused whenever AC power drops out. In response to this interrupt, the microprocessor stores all of the internal registers in a battery-backed-up memory or an EEPROM.

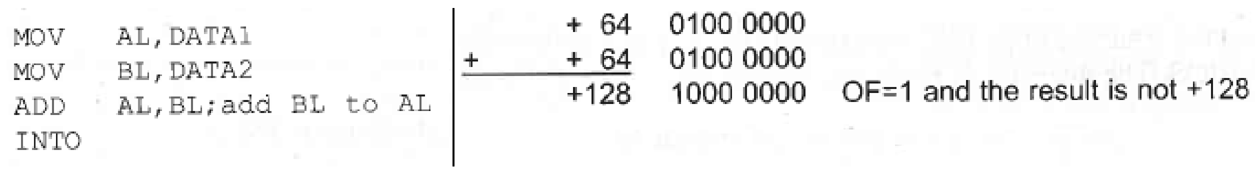
INT 03 (breakpoint)

- ✓ To allow implementation of breakpoints in software engineering, Intel has set aside INT 03.

- ✓ In single-step mode, one can inspect the CPU and system memory after the execution of each instruction, a breakpoint is used to examine the CPU and memory after the execution of a group of instructions.
- ✓ INT 3 is a 1-byte instruction; where as all other "*INT nn*" instructions are 2-byte instructions.

INT 04 (signed number overflow)

- ✓ This interrupt is invoked by a signed number overflow condition. There is an instruction associated with this, INTO (interrupt on overflow).
- ✓ The CPU will activate INT 04 if OF = 1. In cases, where OF = 0, the INTO instruction is not executed; but is bypassed and acts as a NOP (no operation) instruction.
- ✓ To understand this, look at the following example: Suppose in the following program; DATA1= +64 = 0100 0000 and DATA2 = +64 = 0100 0000. The INTO instruction will be executed and the 8088/86 will jump to physical location 00010H, the memory location associated with INT 04. The carry from D6 to D7 causes the overflow flag to become l.
- ✓ Now, the INTO causes the CPU to perform "*INT 4*" and jump to physical location 00010H of the vector table to get the CS: IP of the service routine.

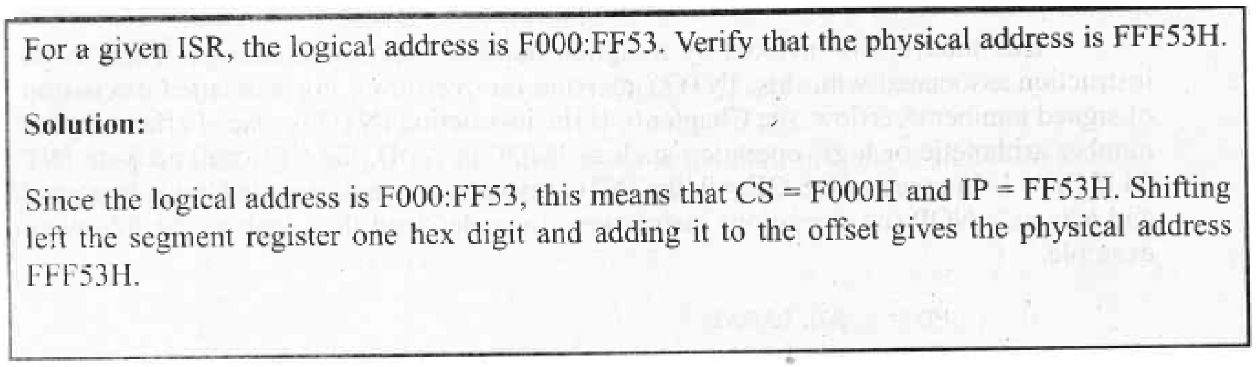


```
MOV   AL,DATA1            |      + 64   0100 0000
MOV   BL,DATA2            | +    + 64   0100 0000
ADD   AL,BL;add BL to AL  |      +128   1000 0000   OF=1 and the result is not +128
INTO                      |
```

- ✓ Suppose that the data in the above program was DATA1 = +64 and DATA2 = +17. In that case, OF would become 0; the INTO is not executed and acts simply as a NOP (no operation) instruction.

<u>x86 PC AND INTERRUPT ASSIGNMENT:</u>

- Of the 256 possible interrupts in the x86;
    - ✓ some are used by the PC peripheral hardware (BIOS)
    - ✓ some are used by the Microsoft operating system
    - ✓ the rest are available for programmers of software applications.



For a given ISR, the logical address is F000:FF53. Verify that the physical address is FFF53H.

Solution:

Since the logical address is F000:FF53, this means that CS = F000H and IP = FF53H. Shifting left the segment register one hex digit and adding it to the offset gives the physical address FFF53H.

INT 21H & INT 10H PROGRAMMING

The INT instruction has the following format:

```
INT  xx;the interrupt number xx can be 00 - FFH
```

Interrupts are numbered 00 to FF; this gives a total of 256 interrupts in x86 microprocessors. Of these 256 interrupts, two of them are the most widely used: INT 10H and INT 21H.

BIOS INT 10H PROGRAMMING:

- INT 10H subroutines are burned into the ROM BIOS of the x86-based IBM PC and compatibles and are used to communicate with the computer's screen video. The manipulation of screen text or graphics can be done through INT 10H.
- There are many functions associated with INT 10H. Among them are changing the color of characters or the background color, clearing the screen, and changing the location of the cursor.
- These options are chosen by putting a specific value in register AH.

Monitor Screen in Text Mode:

- ✓ The monitor screen in the x86 PC is divided into 80 columns and 25 rows in normal text mode (see the following Fig). In other words, the text screen is 80 characters wide by 25 characters long.

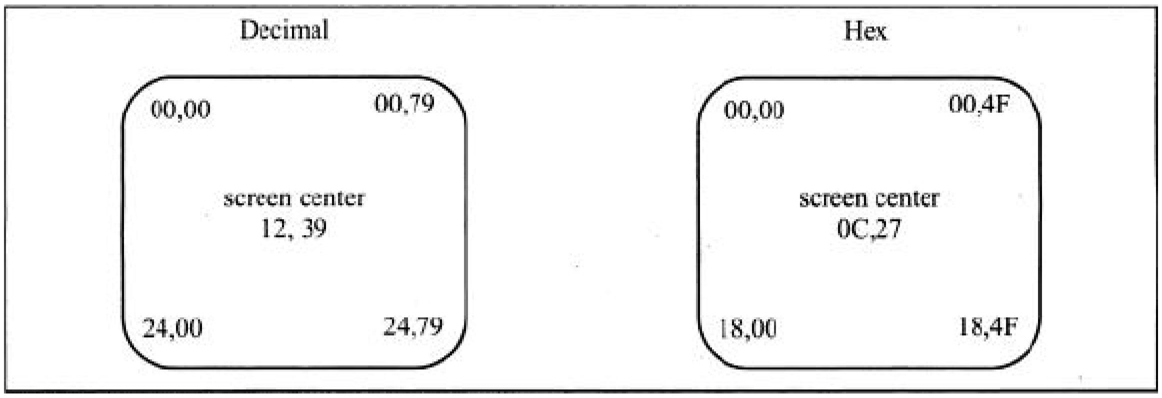
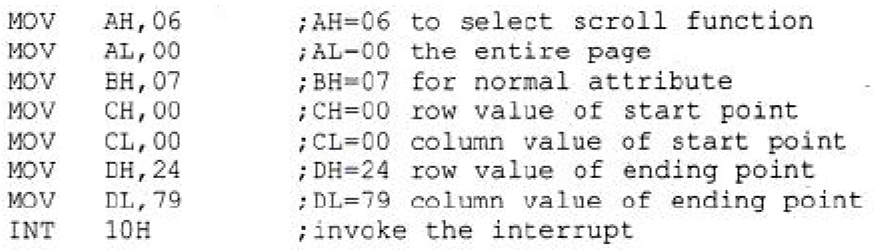


Fig: Cursor Locations (row, column)

- ✓ Since both a row and a column number are associated with each location on the screen, one can move the cursor to any location on the screen simply by changing the row and column values.
- ✓ The 80 columns are numbered from 0 to 79 and the 25 rows are numbered 0 to 24. The top left comer has been assigned 00, 00 (row = 00, column = 00). Therefore, the top right comer will be 00, 79 (row = 00, column = 79).
- ✓ Similarly, the bottom left comer is 24, 00 (row = 24, column = 00) and the bottom right corner of the monitor is 24, 79 (row = 24, column = 79).

INT 10H Function 06H: Clearing the Screen

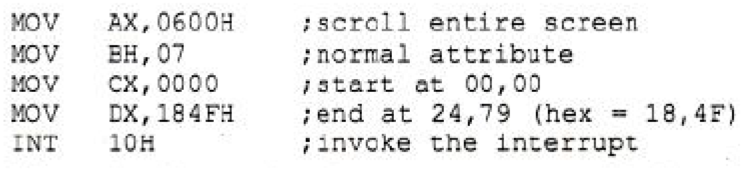
To clear the screen before displaying data; the following registers must contain certain values before INT 10H is called: AH = 06, AL = 00, BH = 07, CX = 0000, DH = 24, and DL= 79. The code will look like this:



```
MOV   AH,06        ;AH=06 to select scroll function
MOV   AL,00        ;AL=00 the entire page
MOV   BH,07        ;BH=07 for normal attribute
MOV   CH,00        ;CH=00 row value of start point
MOV   CL,00        ;CL=00 column value of start point
MOV   DH,24        ;DH=24 row value of ending point
MOV   DL,79        ;DL=79 column value of ending point
INT   10H          ;invoke the interrupt
```

- ✓ Remember that DEBUG assumes immediate operands to be in hex; therefore, DX would be entered as 184F. However, MASM assumes immediate operands to be in decimal. In that case DH = 24 and DL = 79.
- ✓ In the program above, one of many options of INT 10H was chosen by putting 06 into AH. Option AH = 06, called the scroll function, will cause the screen to scroll upward.
- ✓ The CH and CL registers hold the starting row and column, respectively, and DH and DL hold the ending row and column.
- ✓ To clear the entire screen, one must use the top left cursor position of 00, 00 for the start point and the bottom right position of 24, 79 for the end point.
- ✓ Option AH = 06 of INT 10H is in reality the "*scroll window up*" function; therefore, one could use that to make a window of any size by choosing appropriate values for the start and end rows and columns.

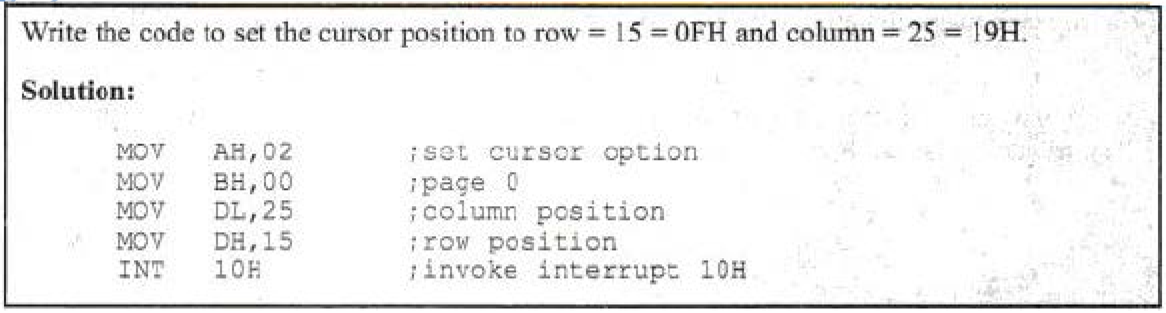
- ✓ To clear the screen, the top left and bottom right values are used for start and stop points in order to scroll up the entire screen. It is more efficient coding to clear the screen by combining some of the lines above as follows:



```
MOV   AX,0600H    ;scroll entire screen
MOV   BH,07       ;normal attribute
MOV   CX,0000     ;start at 00,00
MOV   DX,184FH    ;end at 24,79 (hex = 18,4F)
INT   10H         ;invoke the interrupt
```

INT 10H Function 02: Setting the Cursor to a Specific Location

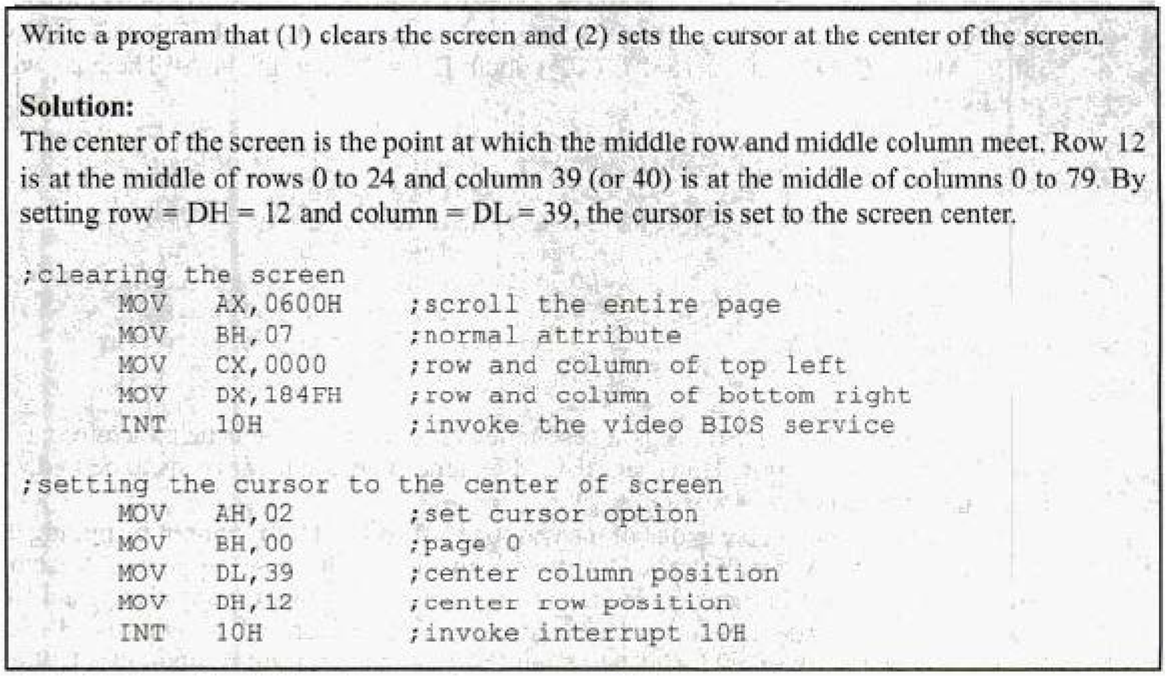
- ✓ INT 10H function AH = 02 will change the position of the cursor to any location.
- ✓ The desired position of the cursor is identified by the row and column values in DX, where DH = row and DL = column.
- ✓ Video RAM can have multiple pages of text, but only one of them can be viewed at a time. When AH = 02, to set the cursor position, page zero is chosen by making BH = 00.



Write the code to set the cursor position to row = 15 = 0FH and column = 25 = 19H.

Solution:

```
        MOV    AH,02          ;set cursor option
        MOV    BH,00          ;page 0
        MOV    DL,25          ;column position
        MOV    DH,15          ;row position
        INT    10H            ;invoke interrupt 10H
```



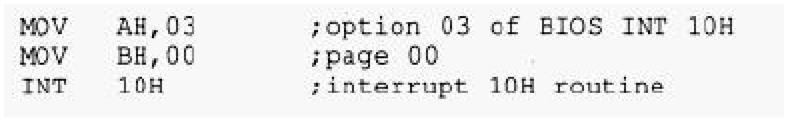
Write a program that (1) clears the screen and (2) sets the cursor at the center of the screen.

Solution:
The center of the screen is the point at which the middle row and middle column meet. Row 12 is at the middle of rows 0 to 24 and column 39 (or 40) is at the middle of columns 0 to 79. By setting row = DH = 12 and column = DL = 39, the cursor is set to the screen center.

```
;clearing the screen
        MOV    AX,0600H       ;scroll the entire page
        MOV    BH,07          ;normal attribute
        MOV    CX,0000        ;row and column of top left
        MOV    DX,184FH       ;row and column of bottom right
        INT    10H            ;invoke the video BIOS service

;setting the cursor to the center of screen
        MOV    AH,02          ;set cursor option
        MOV    BH,00          ;page 0
        MOV    DL,39          ;center column position
        MOV    DH,12          ;center row position
        INT    10H            ;invoke interrupt 10H
```

INT 10H Function 03: Get Current Cursor Position

In text mode, it is possible to determine where the cursor is located at any time by executing the following:



```
MOV    AH,03          ;option 03 of BIOS INT 10H
MOV    BH,00          ;page 00
INT    10H            ;interrupt 10H routine
```

- ✓ After execution of the program above, registers DH and DL will have the current row and column positions, and CX provides information about the shape of the cursor.
- ✓ The reason that page 00 was chosen is that the video memory could contain more than one page of data, depending on the video board installed on the PC.
- ✓ In text mode, page 00 is chosen for the currently viewed page.

Attribute Byte in Monochrome Monitors:

- ✓ There is an attribute associated with each character on the screen.

- ✓ The attribute provides information to the video circuitry, such as color and intensity of the character (foreground) and the background.
- ✓ The attribute byte for each character on the monochrome monitor is limited. The following Fig shows bit definitions of the monochrome attribute byte.

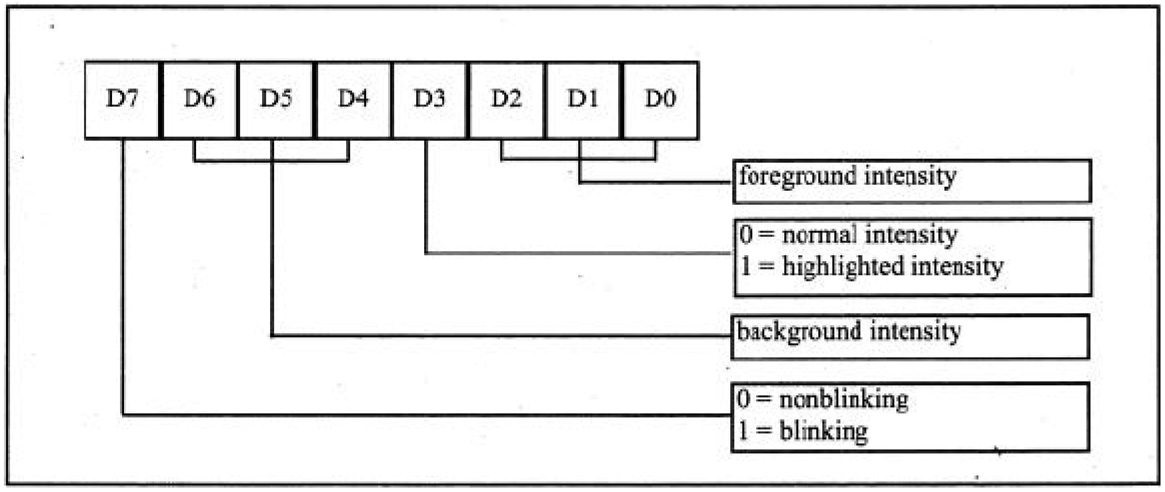
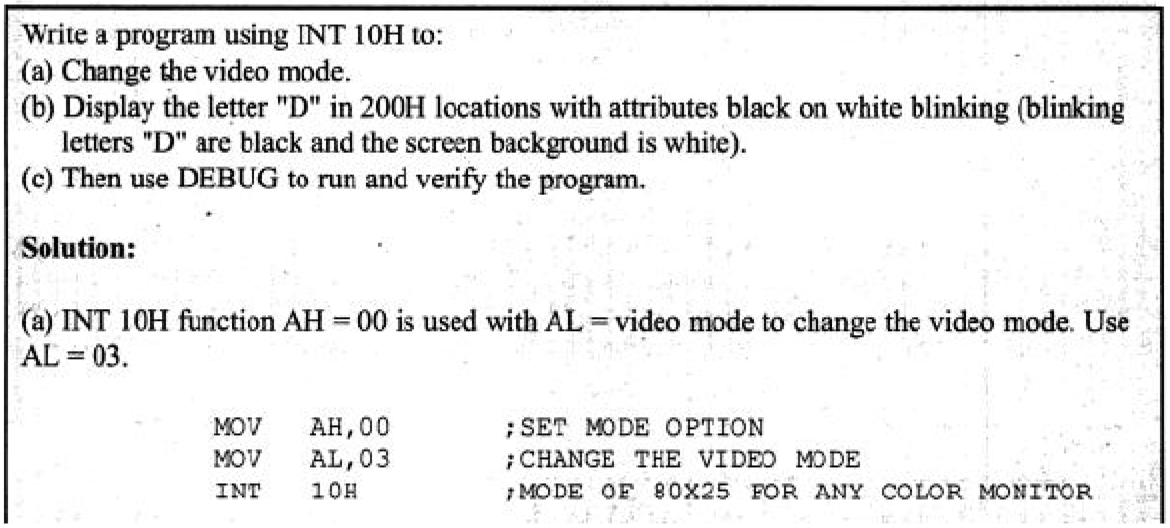


Fig: Attribute Byte for Monochrome Monitors

The following are some possible variations of the attributes shown in the above Fig.

Binary	Hex	Result
0000 0000	00	white on white (no display)
0000 0111	07	white on black normal
0000 1111	0F	white on black highlight
1000 0111	87	white on black blinking
0111 0111	77	black on black (no display)
0111 0000	70	black on white
1111 0000	F0	black on white blinking

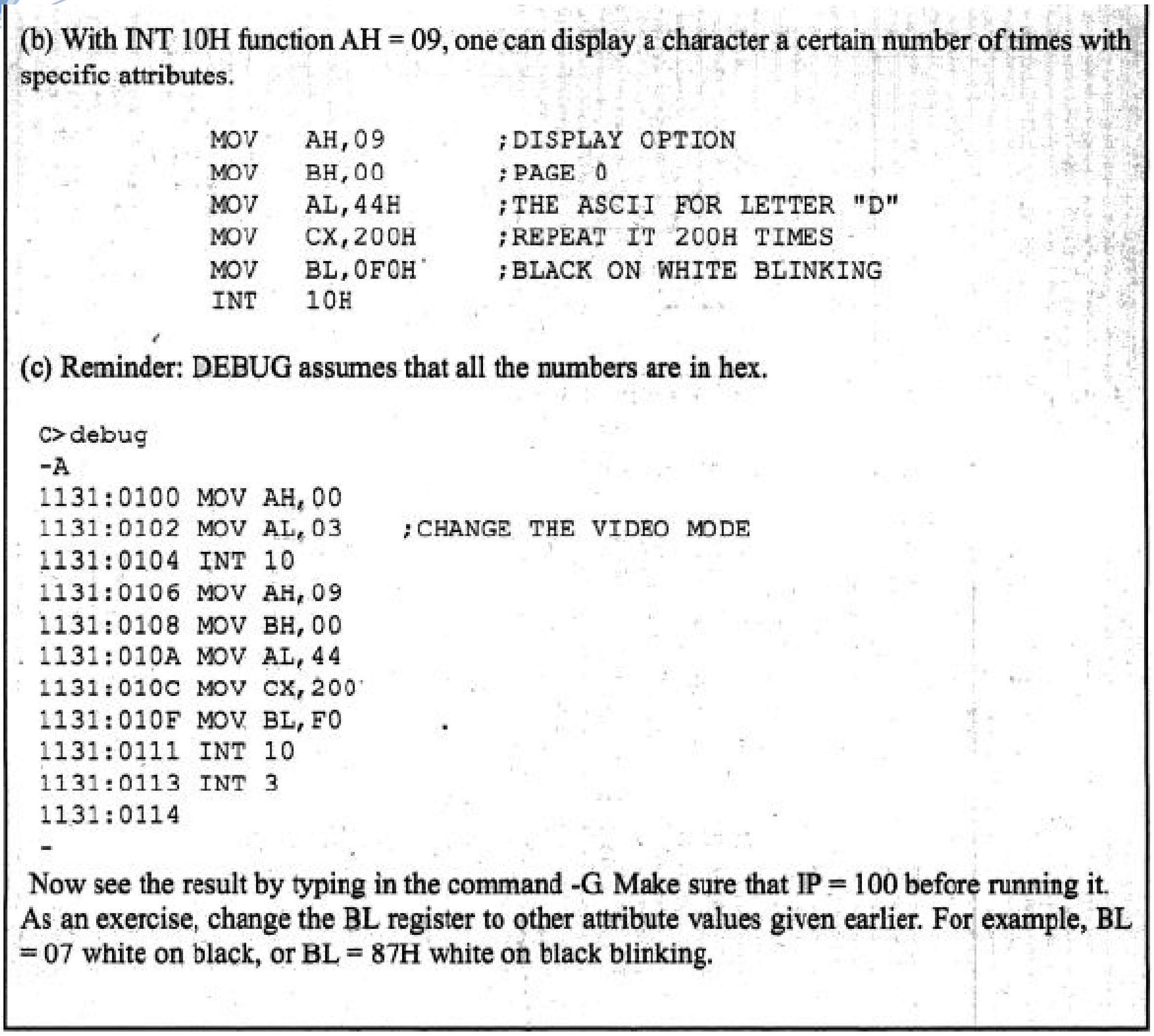


Write a program using INT 10H to:
(a) Change the video mode.
(b) Display the letter "D" in 200H locations with attributes black on white blinking (blinking letters "D" are black and the screen background is white).
(c) Then use DEBUG to run and verify the program.

Solution:

(a) INT 10H function AH = 00 is used with AL = video mode to change the video mode. Use AL = 03.

```
          MOV    AH,00          ;SET MODE OPTION
          MOV    AL,03          ;CHANGE THE VIDEO MODE
          INT    10H            ;MODE OF 80X25 FOR ANY COLOR MONITOR
```



(b) With INT 10H function AH = 09, one can display a character a certain number of times with specific attributes.

```
        MOV   AH,09       ;DISPLAY OPTION
        MOV   BH,00       ;PAGE 0
        MOV   AL,44H      ;THE ASCII FOR LETTER "D"
        MOV   CX,200H     ;REPEAT IT 200H TIMES
        MOV   BL,0F0H     ;BLACK ON WHITE BLINKING
        INT   10H
```

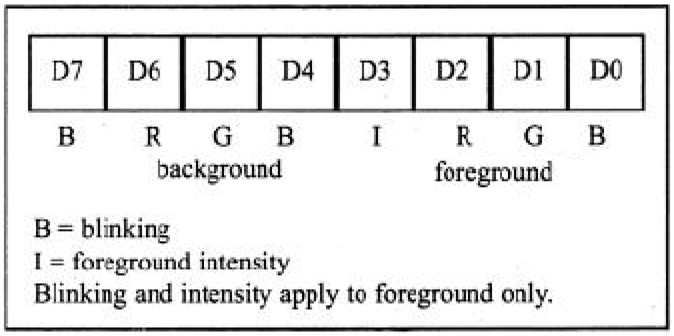
(c) Reminder: DEBUG assumes that all the numbers are in hex.

```
C>debug
-A
1131:0100 MOV AH,00
1131:0102 MOV AL,03    ;CHANGE THE VIDEO MODE
1131:0104 INT 10
1131:0106 MOV AH,09
1131:0108 MOV BH,00
1131:010A MOV AL,44
1131:010C MOV CX,200
1131:010F MOV BL,F0
1131:0111 INT 10
1131:0113 INT 3
1131:0114
-
```

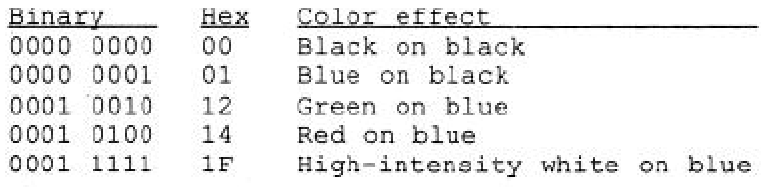
Now see the result by typing in the command -G Make sure that IP = 100 before running it. As an exercise, change the BL register to other attribute values given earlier. For example, BL = 07 white on black, or BL = 87H white on black blinking.

Attribute Byte in CGA Text Mode:

The bit definition of the attribute byte in CGA text mode is shown in the following Fig.

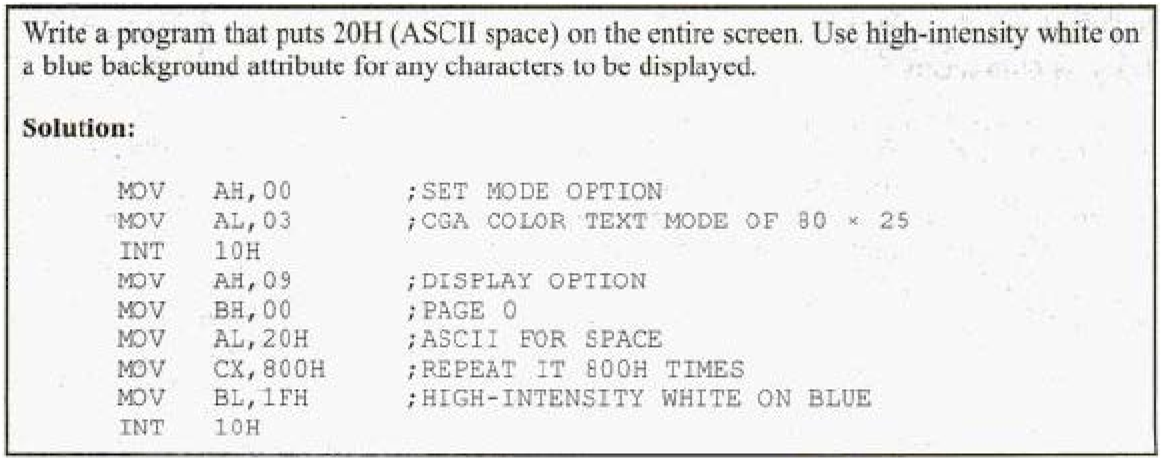


From the bit definition, it can be seen that, the background can take eight different colors by combining the prime colors red, blue, and green. The foreground can be any of 16 different colors by combining red, blue, green, and intensity.



Binary	Hex	Color effect
0000 0000	00	Black on black
0000 0001	01	Blue on black
0001 0010	12	Green on blue
0001 0100	14	Red on blue
0001 1111	1F	High-intensity white on blue

The following Program shows the use of the attribute byte in CGA mode.



Write a program that puts 20H (ASCII space) on the entire screen. Use high-intensity white on a blue background attribute for any characters to be displayed.

Solution:

```
MOV   AH,00        ;SET MODE OPTION
MOV   AL,03        ;CGA COLOR TEXT MODE OF 80 × 25
INT   10H
MOV   AH,09        ;DISPLAY OPTION
MOV   BH,00        ;PAGE 0
MOV   AL,20H       ;ASCII FOR SPACE
MOV   CX,800H      ;REPEAT IT 800H TIMES
MOV   BL,1FH       ;HIGH-INTENSITY WHITE ON BLUE
INT   10H
```

Graphics: Pixel Resolution and Color:

- In the text mode, the screen is viewed as a matrix of rows and columns of characters.
- In graphics mode, the screen is viewed as a matrix of horizontal and vertical pixels.
- The number of pixels varies among monitors and depends on monitor resolution and the video board.
- There are two facts associated with every pixel on the screen:
  - ✓ The location of the pixel
  - ✓ Its attributes, color, and intensity
- These two facts must be stored in the video RAM.
- Higher the number of pixels and colors, the larger the amount of memory is needed to store.
- The CGA mode can have a maximum of 16K bytes of video memory.
- This 16K bytes of memory can be used in three different ways:
  - ✓ Text mode of 80 x 25 characters: Use AL = 03 for mode selection in INT 10H option AH = 00. In this mode, 16 colors are supported.
  - ✓ Graphics mode of resolution 320 x 200 (medium resolution): Use AL = 04. In this mode, 4 colors are supported.
  - ✓ Graphics mode of resolution 640 x 200 (high resolution): Use AL = 06. In this mode, only 1 color (black and white) is supported.

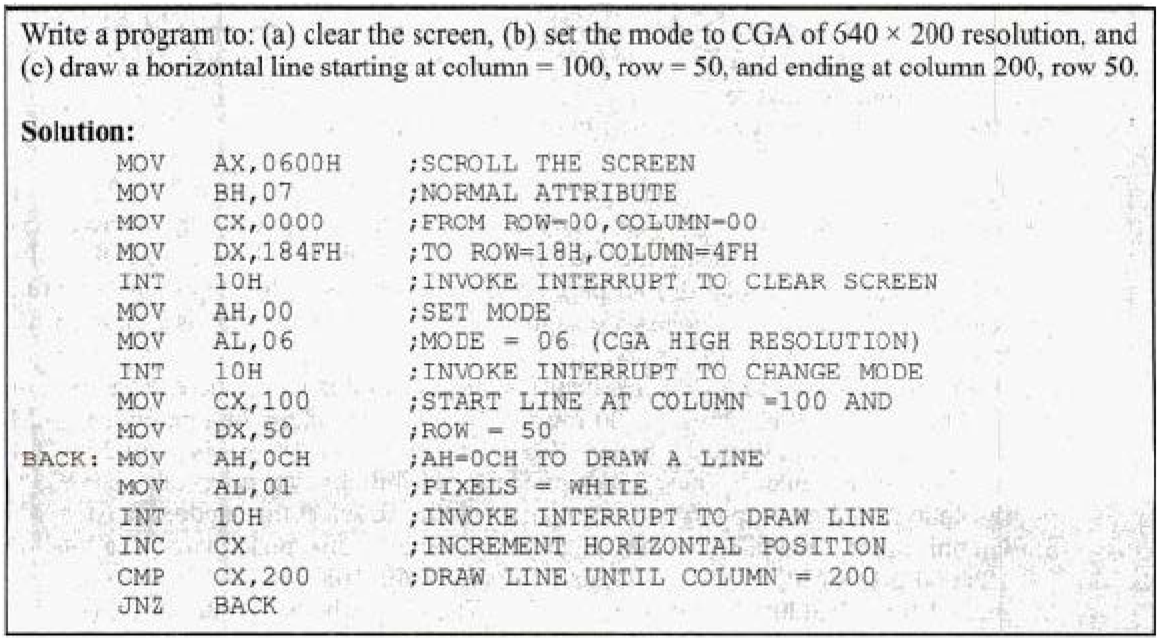
- Hence, with a fixed amount of video RAM, the number of supported colors decreases as the resolution increases.

Table: The 16 Possible Colors

I	R	G	B	Color		I	R	G	B	Color
0	0	0	0	Black		1	0	0	0	Gray
0	0	0	1	Blue		1	0	0	1	Light Blue
0	0	1	0	Green		1	0	1	0	Light Green
0	0	1	1	Cyan		1	0	1	1	Light Cyan
0	1	0	0	Red		1	1	0	0	Light Red
0	1	0	1	Magenta		1	1	0	1	Light Magenta
0	1	1	0	Brown		1	1	1	0	Yellow
0	1	1	1	White		1	1	1	1	High Intensity White

INT 10H and Pixel Programming:

To draw a horizontal line, choose values for the row and column to point to the beginning of the line and then continue to increment the column until it reaches the end of the line, as shown in Example below:



Write a program to: (a) clear the screen, (b) set the mode to CGA of 640 × 200 resolution, and (c) draw a horizontal line starting at column = 100, row = 50, and ending at column 200, row 50.

Solution:

```
        MOV   AX,0600H     ;SCROLL THE SCREEN
        MOV   BH,07        ;NORMAL ATTRIBUTE
        MOV   CX,0000      ;FROM ROW=00,COLUMN=00
        MOV   DX,184FH     ;TO ROW=18H,COLUMN=4FH
        INT   10H          ;INVOKE INTERRUPT TO CLEAR SCREEN
        MOV   AH,00        ;SET MODE
        MOV   AL,06        ;MODE = 06 (CGA HIGH RESOLUTION)
        INT   10H          ;INVOKE INTERRUPT TO CHANGE MODE
        MOV   CX,100       ;START LINE AT COLUMN =100 AND
        MOV   DX,50        ;ROW = 50
BACK:   MOV   AH,0CH       ;AH=0CH TO DRAW A LINE
        MOV   AL,01        ;PIXELS = WHITE
        INT   10H          ;INVOKE INTERRUPT TO DRAW LINE
        INC   CX           ;INCREMENT HORIZONTAL POSITION
        CMP   CX,200       ;DRAW LINE UNTIL COLUMN = 200
        JNZ   BACK
```

<u>**DOS INTERRUPT 21H:**</u>

- INT21H is provided by DOS, which is BIOS-ROM based.
- When the OS is loaded into the computer, INT 21H can be invoked to perform some extremely useful functions. These functions are commonly referred to as *DOS INT 21H* function calls.

INT 21H Option 09: Outputting a String of Data to the Monitor

- ✓ INT 21H can be used to send a set of ASCII data to the monitor. To do that, the following registers must be set: AH = 09 and DX = the offset address of the ASCII data to be displayed.
- ✓ The address in the DX register is an offset address and DS is assumed to be the data segment. INT 21H option 09 will display the ASCII data string pointed at by DX until it encounters the dollar sign "$".
- ✓ In the absence of encountering a dollar sign, DOS function call 09 will continue to display any garbage that it can find in subsequent memory locations until it finds "$".

```
DATA_ASC     DB     'The earth is but one country','$'

       MOV    AH,09          ;option 09 to display string of data
       MOV    DX,OFFSET DATA_ASC      ;DX= offset address of data
       INT    21H                     ;invoke the interrupt
```

INT 21H Option 02: Outputting a Single Character to the Monitor

- ✓ To output a single character to the monitor, 02 is put in AH, DL is loaded with the character to be displayed, and then INT 21H is invoked. The following displays the letter "J'.

```
MOV    AH,02        ;option 02 displays one character
MOV    DL,'J'       ;DL holds the character to be displayed
INT    21H          ;invoke the interrupt
```

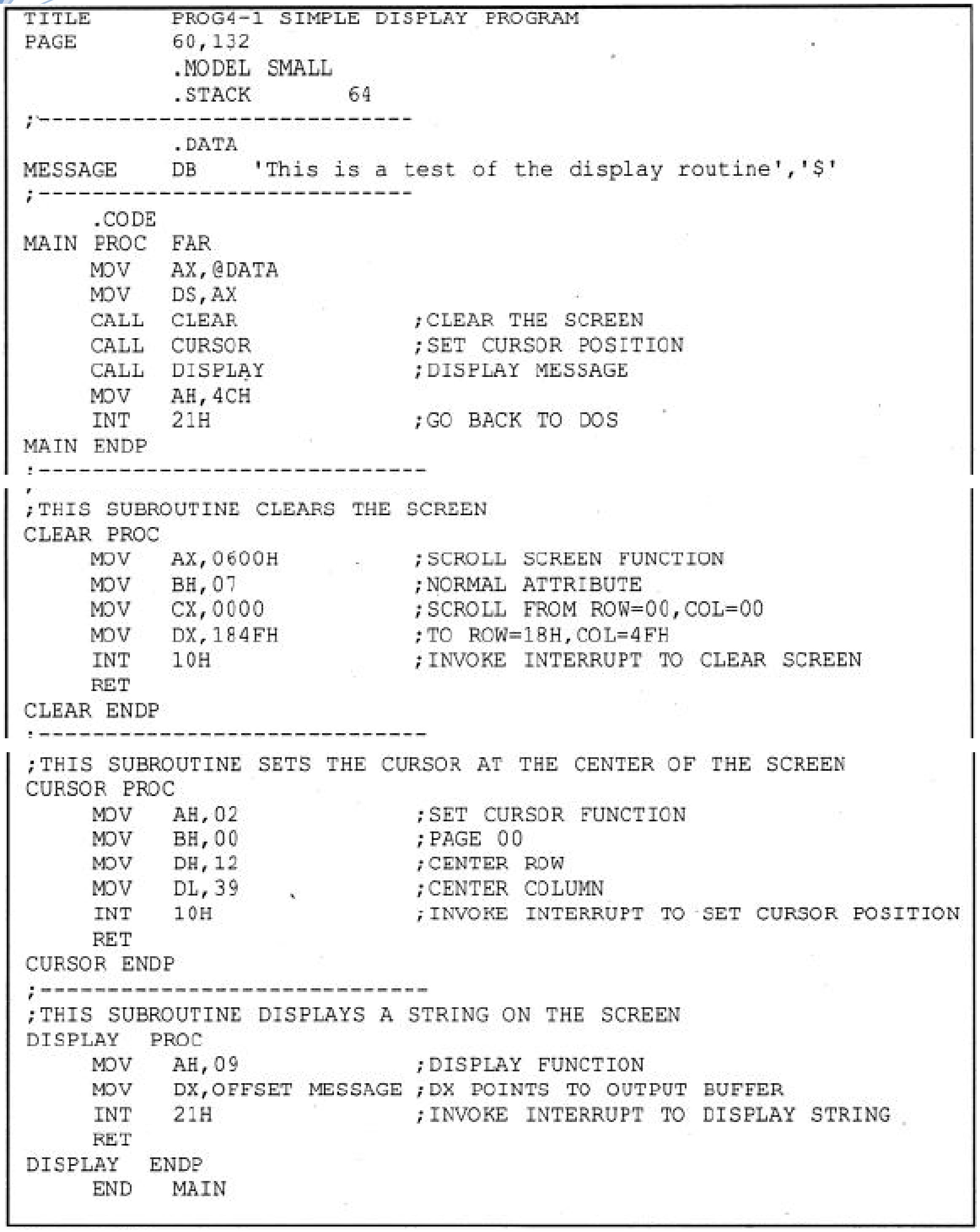
INT 21H Option 01: Inputting a Single Character, with Echo

This function waits until a character is input from the keyboard, and then echoes it to the monitor. After the interrupt, the input character (ASCII value) will be in AL.

```
MOV    AH,01 ;option 01 inputs one character
INT    21H   ;after the interrupt, AL = input character (ASCII)
```

The Program 4-1 does the following:

1. clears the screen
2. sets the cursor to the center of the screen, and
3. starting at that point of the screen, displays the message "This is a test of the display routine".



```
TITLE       PROG4-1 SIMPLE DISPLAY PROGRAM
PAGE        60,132
            .MODEL SMALL
            .STACK       64
;-----------------------------
            .DATA
MESSAGE     DB    'This is a test of the display routine','$'
;-----------------------------
      .CODE
MAIN  PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      CALL  CLEAR              ;CLEAR THE SCREEN
      CALL  CURSOR             ;SET CURSOR POSITION
      CALL  DISPLAY            ;DISPLAY MESSAGE
      MOV   AH,4CH
      INT   21H                ;GO BACK TO DOS
MAIN  ENDP
;------------------------------
;
;THIS SUBROUTINE CLEARS THE SCREEN
CLEAR PROC
      MOV   AX,0600H           ;SCROLL SCREEN FUNCTION
      MOV   BH,07              ;NORMAL ATTRIBUTE
      MOV   CX,0000            ;SCROLL FROM ROW=00,COL=00
      MOV   DX,184FH           ;TO ROW=18H,COL=4FH
      INT   10H                ;INVOKE INTERRUPT TO CLEAR SCREEN
      RET
CLEAR ENDP
;------------------------------
;THIS SUBROUTINE SETS THE CURSOR AT THE CENTER OF THE SCREEN
CURSOR PROC
      MOV   AH,02              ;SET CURSOR FUNCTION
      MOV   BH,00              ;PAGE 00
      MOV   DH,12              ;CENTER ROW
      MOV   DL,39              ;CENTER COLUMN
      INT   10H                ;INVOKE INTERRUPT TO SET CURSOR POSITION
      RET
CURSOR ENDP
;------------------------------
;THIS SUBROUTINE DISPLAYS A STRING ON THE SCREEN
DISPLAY  PROC
      MOV   AH,09              ;DISPLAY FUNCTION
      MOV   DX,OFFSET MESSAGE  ;DX POINTS TO OUTPUT BUFFER
      INT   21H                ;INVOKE INTERRUPT TO DISPLAY STRING
      RET
DISPLAY  ENDP
      END   MAIN
```

Program 4-1

INT 21H Option 0AH: Inputting a String of Data from the Keyboard

- ✓ Option 0AH of INT 21H provides a means by which one can get data from the keyboard and store it in a predefined area of memory in the data segment.
- ✓ To do this; the register options are: AH = 0AH and DX = offset address at which the string of data is stored.
- ✓ This is commonly referred to as a buffer area.

- ✓ DOS requires that a buffer area be defined in the data segment and the first byte specifies the size of the buffer. DOS will put the number of characters that came in through the keyboard in the second byte and the keyed-in data is placed in the buffer starting at the third byte.
- ✓ For example, the following program will accept up to six characters from the keyboard, including the return (carriage return) key. Six locations were reserved for the buffer and filled with FFH.
- ✓ The following shows portions of the data segment and code segment:

```
ORG    0010H
DATA1 DB    6,?,6 DUP (FF);0010H=06, 0012H to 0017H = FF

      MOV   AH,0AH           ;string input option of INT 21H
      MOV   DX,OFFSET DATA1  ;load the offset address of buffer
      INT   21H              ;invoke interrupt 21H
```

- ✓ The following shows the memory contents of offset 0010H:

```
0010  0011  0012  0013  0014  0015  0016  0017
06    00    FF    FF    FF    FF    FF    FF
```

- ✓ When this program is executed, the computer waits for the information to come in from the keyboard.
- ✓ When the data comes in, the IBM PC will not exit the INT 21H routine until it encounters the return key.
- ✓ Assuming the data that was entered through the keyboard was "USA" <RETURN>, the contents of memory locations starting at offset 0010H would look like this:

```
0010  0011  0012  0013  0014  0015  0016  0017
06    03    55    53    41    0D    FF    FF
USACR
```

- ✓ The step-by-step analysis is given below:

0010H = 06	DOS requires the size of the buffer in the first location.
0011H = 03	The keyboard was activated three times (excluding the RETURN key) to key in the letters U, S, and A.
0012H = 55H	This is the ASCII hex value for letter U.
0013H = 53H	This is the ASCII hex value for letter S.
0014H = 41H	This is the ASCII hex value for letter A.
0015H = 0DH	This is the ASCII hex value for CR (carriage return).

- ✓ The 0AH option of INT 21H accepts the string of data from the keyboard and echoes (displays) it on the screen as it is keyed in.

Use of Carriage Return and Line Feed:

- o In the Program 4-2, the EQU statement is used to equate CR (carriage return) with its ASCII value of 0DH, and LF (line feed) with its ASCII value of 0AH.
- o This makes the program much more readable. Since the result of the conversion was to be displayed in the next line, the string was preceded by CR and LF.

- In the absence of CR the string would be displayed wherever the cursor happened to be.
- In the case of CR and no LF, the string would be displayed on the same line after it had been returned to the beginning of the line.

```
;Program 4-2 performs the following: (1) clears the screen, (2) sets
;the cursor at the beginning of the third line from the top of the
;screen, (3) accepts the message "IBM perSonal COmputer" from the
;keyboard, (4) converts lowercase letters of the message to uppercase,
;(5) displays the converted results on the next line.

 TITLE      PROG4-2
 PAGE       60,132
            .MODEL SMALL
            .STACK      64

 ;----------------------------
            .DATA
 BUFFER     DB    22,?,22 DUP (?)          ;BUFFER FOR KEYED-IN DATA
            ORG   18H
 DATAREA    DB    CR,LF,22 DUP (?),'$'    ;DATA HERE AFTER CONVERSION
 ;DTSEG     ENDS
 CR    EQU 0DH
 LF    EQU 0AH

 ;----------------------------
      .CODE
 MAIN PROC  FAR
      MOV   AX,@DATA
      MOV   DS,AX
      CALL  CLEAR               ;CLEAR THE SCREEN
      CALL  CURSOR              ;SET CURSOR POSITION
      CALL  GETDATA             ;INPUT A STRING INTO BUFFER
      CALL  CONVERT             ;CONVERT STRING TO UPPERCASE
      CALL  DISPLAY             ;DISPLAY STRING DATAREA
      MOV   AH,4CH
      INT   21H           ;GO BACK TO DOS
 MAIN ENDP

 ;-----------------------------
 ;THIS SUBROUTINE CLEARS THE SCREEN
 CLEAR PROC
      MOV   AX,0600H            ;SCROLL SCREEN FUNCTION
      MOV   BH,07               ;NORMAL ATTRIBUTE
      MOV   CX,0000                   ;SCROLL FROM ROW=00,COL=00
      MOV   DX,184FH            ;TO ROW=18H,4FH
      INT   10H                 ;INVOKE INTERRUPT TO CLEAR SCREEN
      RET
 CLEARENDP
 ;THIS SUBROUTINE SETS THE CURSOR TO THE BEGINNING OF THE 3RD LINE
 CURSOR PROC
      MOV   AH,02               ;SET CURSOR FUNCTION
      MOV   BH,00               ;PAGE 0
      MOV   DL,01               ;COLUMN 1
      MOV   DH,03               ;ROW 3
      INT   10H                 ;INVOKE INTERRUPT TO SET CURSOR
      RET
 CURSOR ENDP

 ;-----------------------------
```

```
;THIS SUBROUTINE DISPLAYS A STRING ON THE SCREEN
DISPLAY PROC
      MOV    AH,09              ;DISPLAY STRING FUNCTION
      MOV    DX,OFFSET DATAREA ;DX POINTS TO BUFFER
      INT    21H                ;INVOKE INTERRUPT TO DISPLAY STRING
      RET
DISPLAY      ENDP

;------------------------------
;THIS SUBROUTINE PUTS DATA FROM THE KEYBOARD INTO A BUFFER
GETDATA PROC
      MOV    AH,0AH             ;INPUT STRING FUNCTION
      MOV    DX,OFFSET BUFFER   ;DX POINTS TO BUFFER
      INT    21H                ;INVOKE INTERRUPT TO INPUT STRING
      RET
GETDATA ENDP
;------------------------------
;THIS SUBROUTINE CONVERTS ANY SMALL LETTER TO ITS CAPITAL
CONVERT PROC
      MOV    BX,OFFSET BUFFER
      MOV    CL,[ BX] +1              ;GET THE CHAR COUNT
      SUB    CH,CH                    ;CX = TOTAL CHARACTER COUNT
      MOV    DI,CX                    ;INDEXING INTO BUFFER
      MOV    BYTE PTR[ BX+DI] +2,20H  ;REPLACE CR WITH SPACE
      MOV    SI,OFFSET DATAREA+2      ;STRING ADDRESS
AGAIN: MOV AL,[ BX] +2                ;GET THE KEYED-IN DATA
      CMP    AL,61H                   ;CHECK FOR 'a'
      JB     NEXT                     ;IF BELOW, GO TO NEXT
      CMP    AL,7AH                   ;CHECK FOR 'z'
      JA     NEXT                     ;IF ABOVE GO TO NEXT
      AND    AL,11011111B             ;CONVERT TO CAPITAL
NEXT: MOV   [ SI] ,AL                 ;PLACE IN DATA AREA
      INC    SI                       ;INCREMENT POINTERS
      INC    BX
      LOOP   AGAIN                    ;LOOP IF COUNTER NOT ZERO
      RET
CONVERT ENDP
      END    MAIN
```

Program 4-2

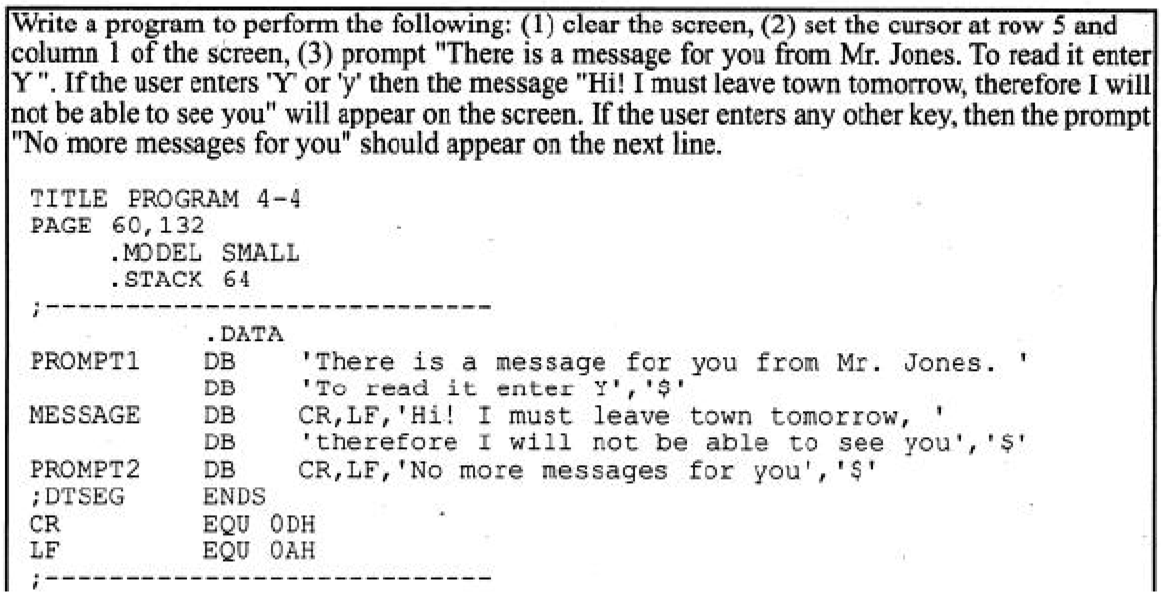
- The Program 4-3 prompts the user to type in a name. The name can have a maximum of eight letters.
- After the name is typed in, the program gets the length of the name and prints it to the screen.

```
TITLE      PROG4-3     READS IN LAST NAME AND DISPLAYS LENGTH
PAGE       60,132
           .MODEL SMALL
           .STACK 64 (?)
;------------------------------
```

```
            .DATA
MESSAGE1    DB     'What is your last name?','$'
            ORG    20H
BUFFER1     DB     9,?,9 DUP (0)
            ORG    30H
MESSAGE2    DB     CR,LF,'The number of letters in your name is: ','$'
ROW         EQU 08
COLUMN      EQU 05
CR          EQU 0DH      ;EQUATE CR WITH ASCII CODE FOR CARRIAGE RETURN
LF          EQU 0AH      ;EQUATE LF WITH ASCII CODE FOR LINE FEED
;----------------------------
            .CODE
MAIN        PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     CALL   CLEAR
     CALL   CURSOR
     MOV    AH,09                ;DISPLAY THE PROMPT
     MOV    DX,OFFSET MESSAGE1
     INT    21H
     MOV    AH,0AH               ;GET LAST NAME FROM KEYBOARD
     MOV    DX,OFFSET BUFFER1
     INT    21H
     MOV    BX,OFFSET BUFFER1 ;FIND OUT NUMBER OF LETTERS IN NAME
     MOV    CL,[ BX+1]           ;GET NUMBER OF LETTERS
     OR     CL,30H               ;MAKE IT ASCII
     MOV    MESSAGE2+40,CL       ;PLACE AT END OF STRING
     MOV    AH,09                ;DISPLAY SECOND MESSAGE
     MOV    DX,OFFSET MESSAGE2
     INT    21H
     MOV    AH,4CH
     INT    21H          ;GO BACK TO DOS
MAIN ENDP
;------------------------------
CLEAR PROC                       ;CLEAR THE SCREEN
     MOV    AX,0600H
     MOV    BH,07
     MOV    CX,0000
     MOV    DX,184FH
     INT    10H
     RET
CLEAR ENDP
;------------------------------
CURSOR PROC                            ;SET CURSOR POSITION
     MOV    AH,02
     MOV    BH,00
     MOV    DL,COLUMN
     MOV    DH,ROW
     INT    10H
     RET
CURSOR ENDP
     END    MAIN
```

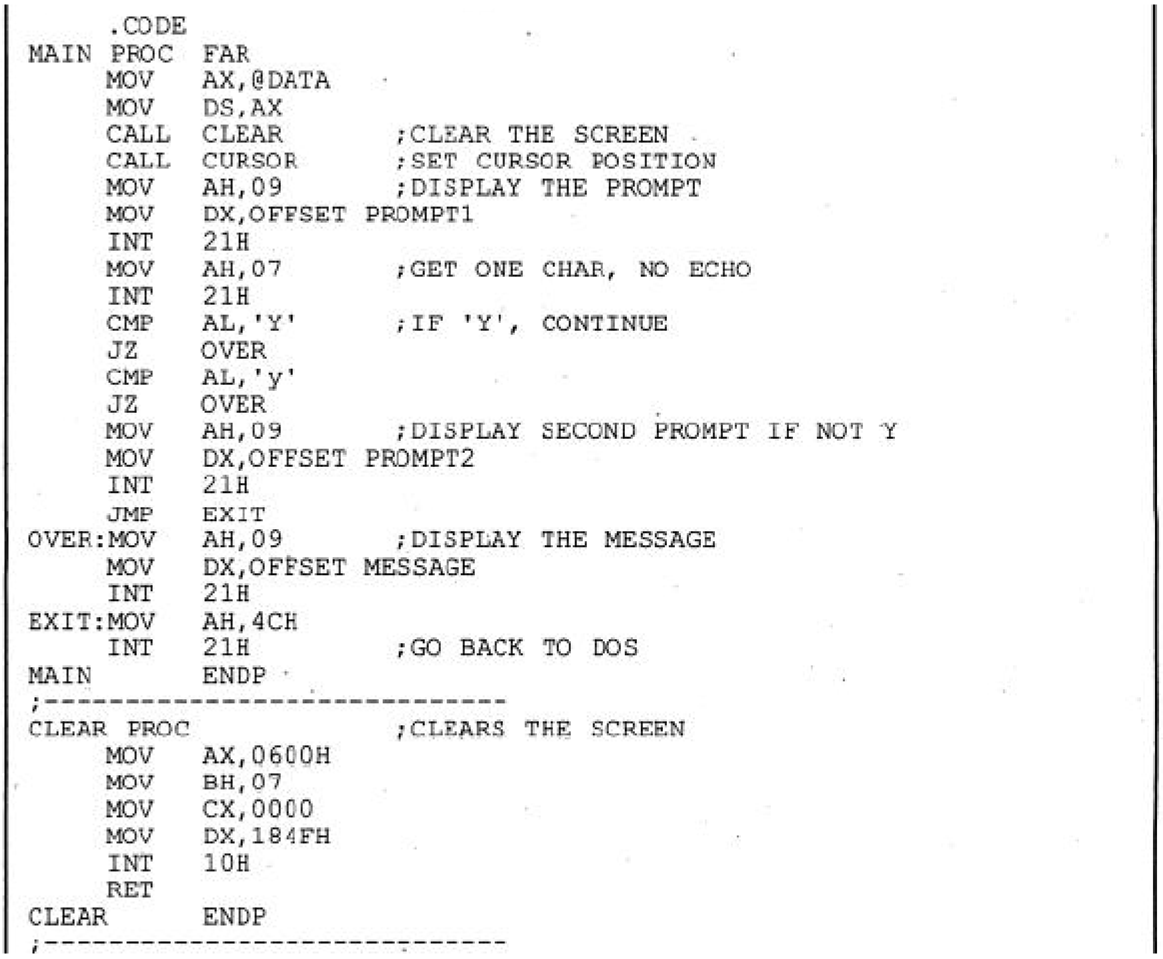
Program 4-3

- Program 4-4 demonstrates many of the functions described:



Write a program to perform the following: (1) clear the screen, (2) set the cursor at row 5 and column 1 of the screen, (3) prompt "There is a message for you from Mr. Jones. To read it enter Y ". If the user enters 'Y' or 'y' then the message "Hi! I must leave town tomorrow, therefore I will not be able to see you" will appear on the screen. If the user enters any other key, then the prompt "No more messages for you" should appear on the next line.

```
TITLE PROGRAM 4-4
PAGE 60,132
     .MODEL SMALL
     .STACK 64
;-----------------------------
           .DATA
PROMPT1    DB    'There is a message for you from Mr. Jones. '
           DB    'To read it enter Y','$'
MESSAGE    DB    CR,LF,'Hi! I must leave town tomorrow, '
           DB    'therefore I will not be able to see you','$'
PROMPT2    DB    CR,LF,'No more messages for you','$'
;DTSEG     ENDS
CR         EQU  0DH
LF         EQU  0AH
;-----------------------------
```



```
     .CODE
MAIN PROC  FAR
     MOV   AX,@DATA
     MOV   DS,AX
     CALL  CLEAR        ;CLEAR THE SCREEN
     CALL  CURSOR       ;SET CURSOR POSITION
     MOV   AH,09        ;DISPLAY THE PROMPT
     MOV   DX,OFFSET PROMPT1
     INT   21H
     MOV   AH,07        ;GET ONE CHAR, NO ECHO
     INT   21H
     CMP   AL,'Y'       ;IF 'Y', CONTINUE
     JZ    OVER
     CMP   AL,'y'
     JZ    OVER
     MOV   AH,09        ;DISPLAY SECOND PROMPT IF NOT Y
     MOV   DX,OFFSET PROMPT2
     INT   21H
     JMP   EXIT
OVER:MOV   AH,09        ;DISPLAY THE MESSAGE
     MOV   DX,OFFSET MESSAGE
     INT   21H
EXIT:MOV   AH,4CH
     INT   21H          ;GO BACK TO DOS
MAIN       ENDP
;------------------------------
CLEAR PROC              ;CLEARS THE SCREEN
     MOV   AX,0600H
     MOV   BH,07
     MOV   CX,0000
     MOV   DX,184FH
     INT   10H
     RET
CLEAR      ENDP
;------------------------------
```

```
CURSOR PROC                          ;SET CURSOR POSITION
       MOV    AH,02
       MOV    BH,00
       MOV    DL,05         ;COLUMN 5
       MOV    DH,08         ;ROW 8
       INT    10H
       RET
CURSOR ENDP
       END    MAIN
```

Program 4-4

INT 21H Option 07: Keyboard Input without Echo

- ✓ Option 07 of INT 21H requires the user to enter a single character but that character is not displayed (or echoed) on the screen.
- ✓ After execution of the interrupt, the PC waits until a single character is entered and provides the character in AL.

```
MOV    AH,07 ;keyboard input without echo
INT    21H
```

Using the LABEL Directive to Define a String Buffer:

- A more systematic way of defining the buffer area for the string input is to use the LABEL directive.
- The LABEL directive can be used in the data segment to assign multiple names to data. When used in the data segment it looks like this:

```
name  LABEL attribute
```

- The attribute can be BYTE, WORD, DWORD, FWORD, QWORD, or TBYTE.

```
JOE    LABEL BYTE
TOM    DB    20 DUP(0)
```

By: **MAHESH PRASANNA K.,**

DEPT. OF CSE, VCET.

****

****

MODULE – 3

SIGNED NUMBERS AND STRINGS & MEMORY INTERFACING & 8255

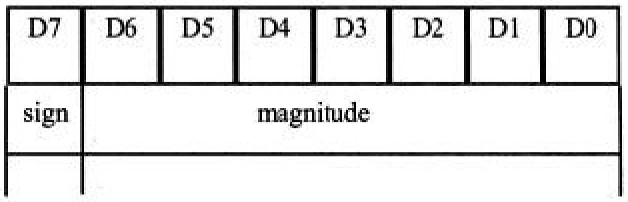
SIGNED NUMBERS & STRINGS

SIGNED NUMBER ARITHMETIC OPERATIONS:

- In everyday life, numbers are used that could be *positive or negative*. For example, a temperature of 5 degrees below zero can be represented as –5, and 20 degrees above zero as +20.
- Computers must be able to accommodate such numbers. To do that, an arrangement for the representation of signed positive and negative numbers is made:
  - ✓ The most significant bit (MSB) is set aside for the sign (+ or –)
  - ✓ The rest of the bits are used for the magnitude.
- The sign is represented by 0 for positive (+) numbers and 1 for negative (–) numbers.
- Note that, entire 8-bit or 16-bit operand will be treated as magnitude in the case of unsigned number representation.

Byte-sized Signed Numbers:

- In signed byte operands, D7 (MSB) is the sign and D6 to D0 are set aside for the magnitude of the number.
  - ✓ If D7 = 0, the operand is positive
  - ✓ If 07 = 1, the operand is negative.



- The range of *positive numbers* that can be represented by the format above is 0 to + 127.

```
0       0000  0000
+1      0000  0001
+5      0000  0101
...     ...   ....
+127    0111  1111
```

- If a *positive number* is larger than +127, a word sized operand must be used.
- For *negative numbers* D7 is 1, but the magnitude is represented in 2's complement.
- Although the assembler does the conversion, it is still important to understand how the conversion works. To convert to negative number representation (2's complement), follow these steps:
  - ✓ Write the magnitude of the number in 8-bit binary (no sign).
  - ✓ Invert each bit
  - ✓ Add 1 to it.

Decimal	Binary	Hex
-128	1000 0000	80
-127	1000 0001	81
-126	1000 0010	82
...	....	..
-2	1111 1110	FE
-1	1111 1111	FF
0	0000 0000	00
+1	0000 0001	01
+2	0000 0010	02
...	... ...	...
+127	0111 1111	7F

Show how the computer would represent –5.

Solution:

```
1. 0000 0101    5 in 8-bit binary
2. 1111 1010    invert each bit
3. 1111 1011    add 1 (hex = FBH)
```

This is the signed number representation in 2's complement for –5.

Show –34H as it is represented internally.

Solution:

```
1. 0011 0100
2. 1100 1011
3. 1100 1100     (which is CCH)
```

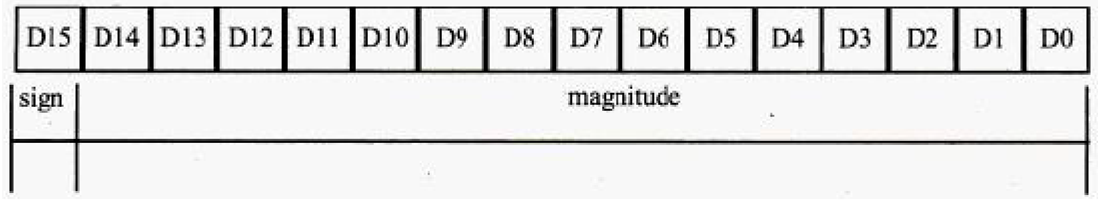
Show the representation for -128_{10}.

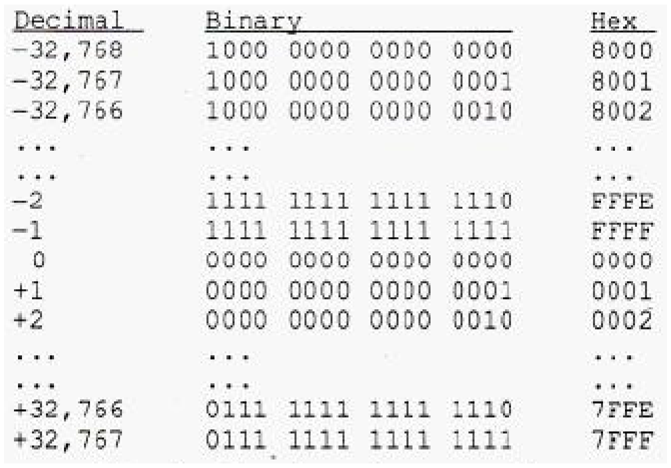
Solution:

```
1.   1000 0000
2.   0111 1111
3.   1000 0000
```
Notice that this is not negative zero (–0).

Word-sized Signed Numbers:

- In x86 computers a word is 16-bits in length. Setting aside the MSB (D15) for the sign leaves a total of 15 bits (D14 – D0) for the magnitude. This gives a range of –32,768 to +32,767.
- If a number is larger than this, it must be treated as a multiword operand and be processed chunk by chunk the same way as unsigned numbers.

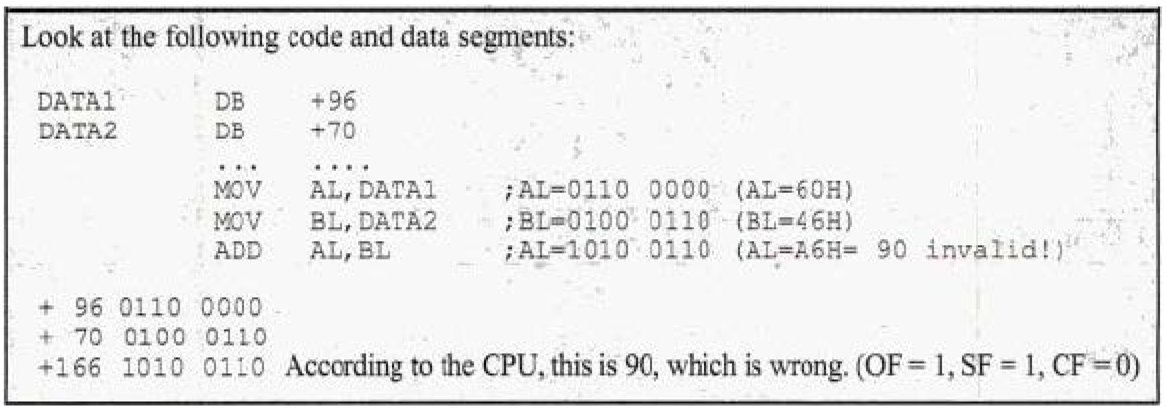




Decimal	Binary	Hex
−32,768	1000 0000 0000 0000	8000
−32,767	1000 0000 0000 0001	8001
−32,766	1000 0000 0000 0010	8002
...	...	...
...	...	...
−2	1111 1111 1111 1110	FFFE
−1	1111 1111 1111 1111	FFFF
0	0000 0000 0000 0000	0000
+1	0000 0000 0000 0001	0001
+2	0000 0000 0000 0010	0002
...	...	...
...	...	...
+32,766	0111 1111 1111 1110	7FFE
+32,767	0111 1111 1111 1111	7FFF

Overflow Problem in Signed Number Operations:

What is an overflow? If the result of an operation on signed numbers is too large for the register, an overflow occurs and the programmer must be notified. Look at following Example:



Look at the following code and data segments:

```
DATA1     DB     +96
DATA2     DB     +70
          ...    ....
          MOV    AL,DATA1     ;AL=0110 0000 (AL=60H)
          MOV    BL,DATA2     ;BL=0100 0110 (BL=46H)
          ADD    AL,BL        ;AL=1010 0110 (AL=A6H= 90 invalid!)

+ 96 0110 0000
+ 70 0100 0110
+166 1010 0110
```

According to the CPU, this is 90, which is wrong. (OF = 1, SF = 1, CF = 0)

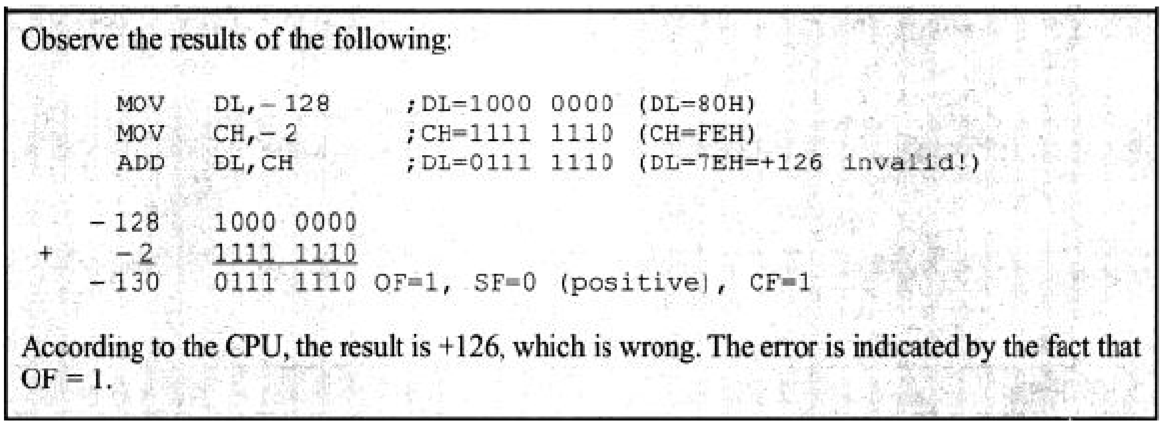
- In the example above; +96 is added to +70 and the result according to the CPU is –90 (5AH). Why?
- The reason is that, the result was more than what AL could handle. Like all other 8-bit registers, AL could only contain up to +127. The *designers of the CPU created the overflow flag specifically for the purpose of informing the programmer that the result of the signed number operation is erroneous.*

Hence, when using signed numbers, a serious problem with regarding overflow arises that must be dealt with. The CPU indicates the existence of the problem by raising the OF (overflow) flag, but it is up to the programmer to take care of it. The CPU understands only 0s and 1s and ignores the human convention of positive and negative numbers.

When Overflow Flag is Set in 8-bit Operations?

In 8-bit signed number operations, OF is set to 1, if either of the following two conditions occurs:

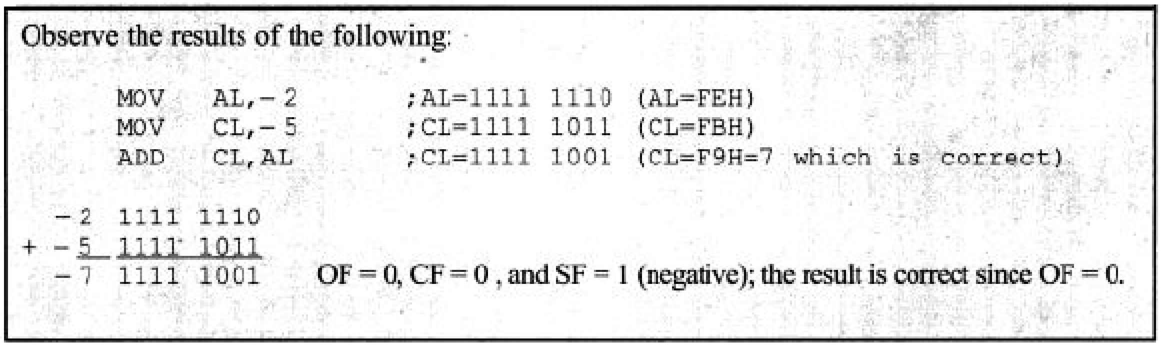
1. There is a carry from D6 to D7, but no carry out of D7 (CF = 0)
2. There is a carry from D7 out (CF = 1), but no carry from D6 to D7.



Observe the results of the following:

```
        MOV     DL,-128       ;DL=1000 0000  (DL=80H)
        MOV     CH,-2         ;CH=1111 1110  (CH=FEH)
        ADD     DL,CH         ;DL=0111 1110  (DL=7EH=+126 invalid!)

     -128       1000 0000
 +     -2       1111 1110
     -130       0111 1110 OF=1,  SF=0  (positive),  CF=1
```

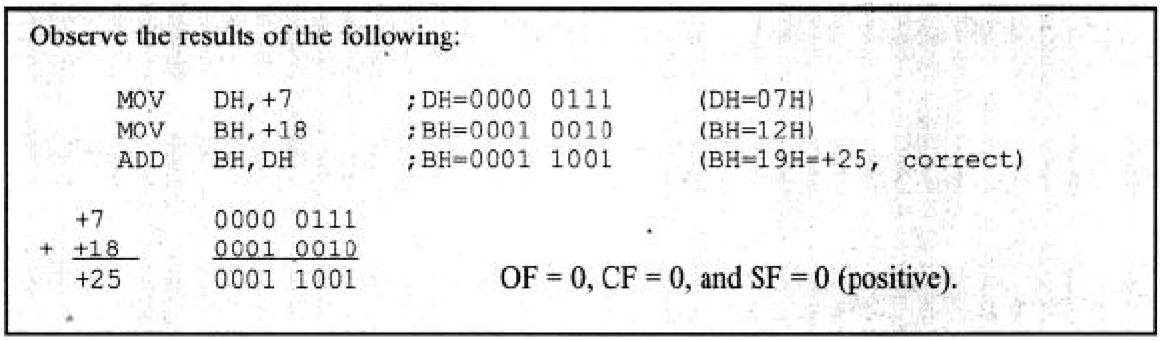
According to the CPU, the result is +126, which is wrong. The error is indicated by the fact that OF = 1.



Observe the results of the following:

```
        MOV     AL,-2         ;AL=1111 1110  (AL=FEH)
        MOV     CL,-5         ;CL=1111 1011  (CL=FBH)
        ADD     CL,AL         ;CL=1111 1001  (CL=F9H=7 which is correct)

    -2  1111 1110
 +  -5  1111 1011
    -7  1111 1001
```

OF = 0, CF = 0 , and SF = 1 (negative); the result is correct since OF = 0.



Observe the results of the following:

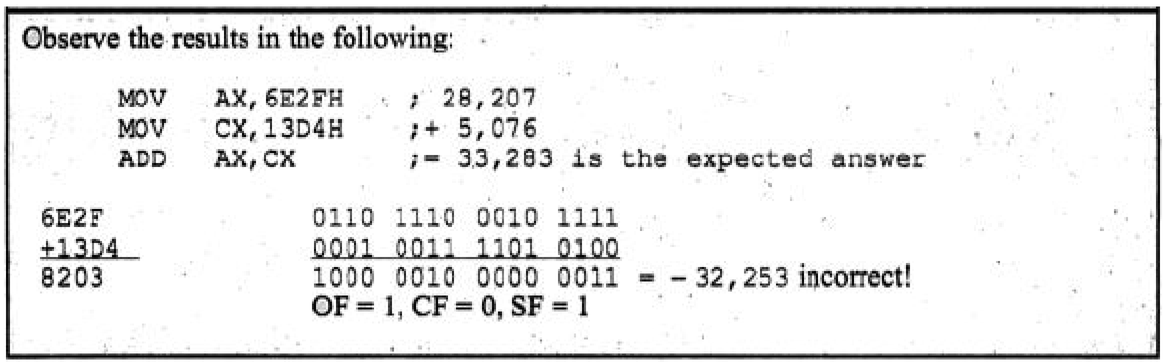
```
        MOV     DH,+7         ;DH=0000 0111      (DH=07H)
        MOV     BH,+18        ;BH=0001 0010      (BH=12H)
        ADD     BH,DH         ;BH=0001 1001      (BH=19H=+25,  correct)

     +7         0000 0111
 +  +18         0001 0010
    +25         0001 1001
```

OF = 0, CF = 0, and SF = 0 (positive).

When Overflow Flag is Set in 16-bit Operations?

In 16-bit signed number operations, OF is set to 1, if either of the following two conditions occurs:

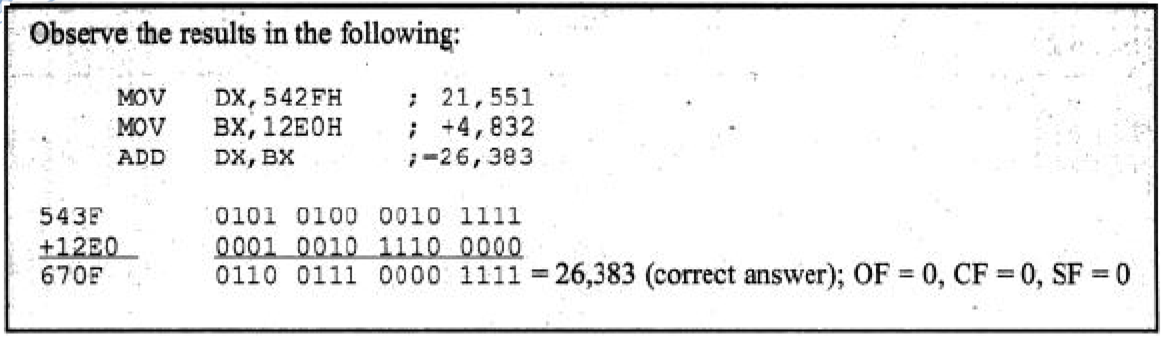
1. There is a carry from D14 to D15, but no carry out of D15 (CF = 0)
2. There is a carry from D15 out (CF = 1), but no carry from D14 to D15.



Observe the results in the following:

```
        MOV     AX,6E2FH      ;  28,207
        MOV     CX,13D4H      ;+ 5,076
        ADD     AX,CX         ;= 33,283 is the expected answer

  6E2F              0110 1110 0010 1111
 +13D4              0001 0011 1101 0100
  8203              1000 0010 0000 0011 = -32,253 incorrect!
```

OF = 1, CF = 0, SF = 1

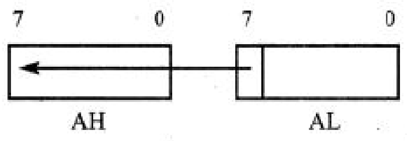


Observe the results in the following:

```
      MOV    DX,542FH     ; 21,551
      MOV    BX,12E0H     ; +4,832
      ADD    DX,BX        ;-26,383

543F         0101 0100 0010 1111
+12E0        0001 0010 1110 0000
670F         0110 0111 0000 1111 = 26,383 (correct answer); OF = 0, CF = 0, SF = 0
```

Avoiding Erroneous Results in Signed Number Operations:

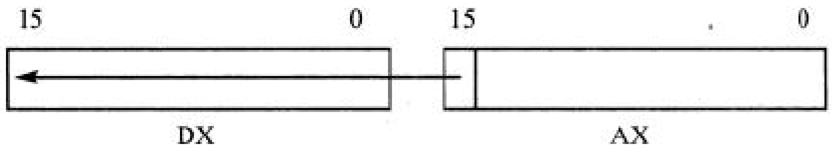
- To avoid the problems associated with signed number operations, one can sign extend the operand.
- Sign extension copies;
  - ✓ the sign bit (D7) of the lower byte of a register into the upper bits of the register, or
  - ✓ the sign bit of a 16-bit register into another register.
- The instructions used to perform the sign extension are;
- CBW (*convert signed byte to signed word*) – will copy D7 (the sign flag) of AL to all bit positions of AH register.



or:

```
MOV    AL,+96          ;AL=0110 0000
CBW                    ;now AH=0000 0000 and AL=0110 0000

MOV    AL,-2           ;AL=1111 1110
CBW                    ;AH=1111 1111 and AL=1111 1110
```

- CWD (*convert signed word to signed double word*): will copy D15 of AX to all bot positions of DX register.



example:

```
MOV    AX,+260       ;AX=0000 0001 0000 0100 or AX=0104H
CWD                  ;DX=0000H and AX=0104H
```

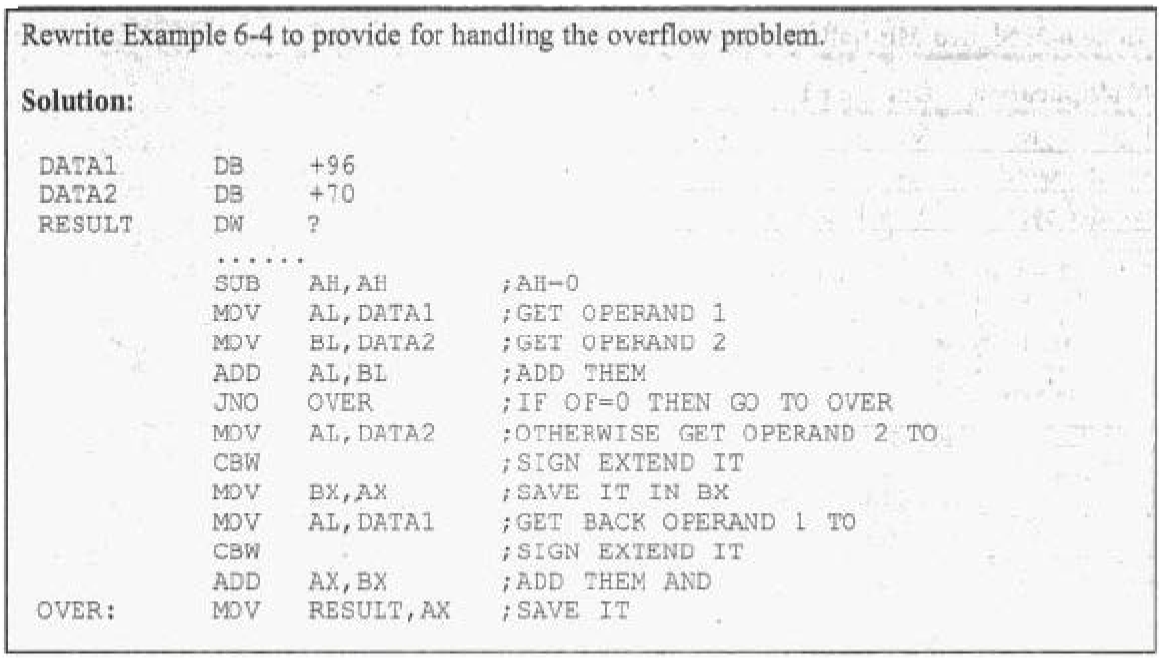
example:

```
MOV    AX,-32766     ;AX=1000 0000 0000 0010B or AX=8002H
CWD                  ;DX=FFFF and AX=8002
```

In the following Example (program for addition of any two signed bytes);

- ✓ If the overflow flag is not raised (OF = 0), the result of the signed number is correct and JNO (jump if no overflow) will jump to OVER.

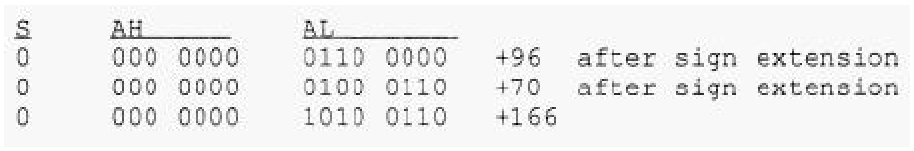
- ✓ If OF = 1, (which means that the result is erroneous), each operand must be sign extended and then added. That is the function of the code below the JNO instruction.



Rewrite Example 6-4 to provide for handling the overflow problem.

Solution:

```
DATA1     DB    +96
DATA2     DB    +70
RESULT    DW    ?
          ......
          SUB   AH,AH        ;AH=0
          MOV   AL,DATA1     ;GET OPERAND 1
          MOV   BL,DATA2     ;GET OPERAND 2
          ADD   AL,BL        ;ADD THEM
          JNO   OVER         ;IF OF=0 THEN GO TO OVER
          MOV   AL,DATA2     ;OTHERWISE GET OPERAND 2 TO
          CBW                ;SIGN EXTEND IT
          MOV   BX,AX        ;SAVE IT IN BX
          MOV   AL,DATA1     ;GET BACK OPERAND 1 TO
          CBW                ;SIGN EXTEND IT
          ADD   AX,BX        ;ADD THEM AND
OVER:     MOV   RESULT,AX    ;SAVE IT
```

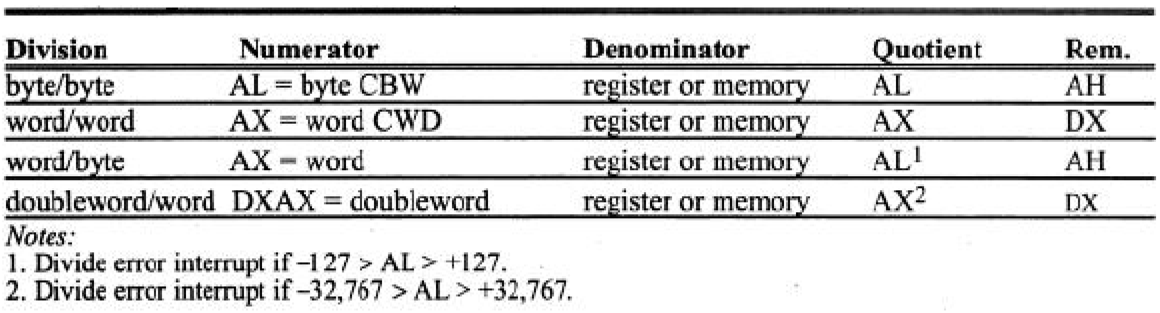


```
S    AH          AL
0    000 0000    0110 0000    +96   after sign extension
0    000 0000    0100 0110    +70   after sign extension
0    000 0000    1010 0110    +166
```

IDIV (signed number division):

The Intel manual says that IDIV means "integer division"; it is used for signed number division. In actuality, all arithmetic instructions of 8088/86 are for integer numbers regardless of whether the operands are signed or unsigned. To perform operations on real numbers, the 8087 coprocessor is used. Remember that real numbers are the ones with decimal points such as "3.56".

Division of signed numbers is very similar to the division of unsigned numbers (already discussed).



Division	Numerator	Denominator	Quotient	Rem.
byte/byte	AL = byte CBW	register or memory	AL	AH
word/word	AX = word CWD	register or memory	AX	DX
word/byte	AX = word	register or memory	AL[1]	AH
doubleword/word	DXAX = doubleword	register or memory	AX[2]	DX

Notes:
1. Divide error interrupt if –127 > AL > +127.
2. Divide error interrupt if –32,767 > AL > +32,767.

Eg1:

IDIV CH

F0H = -10H

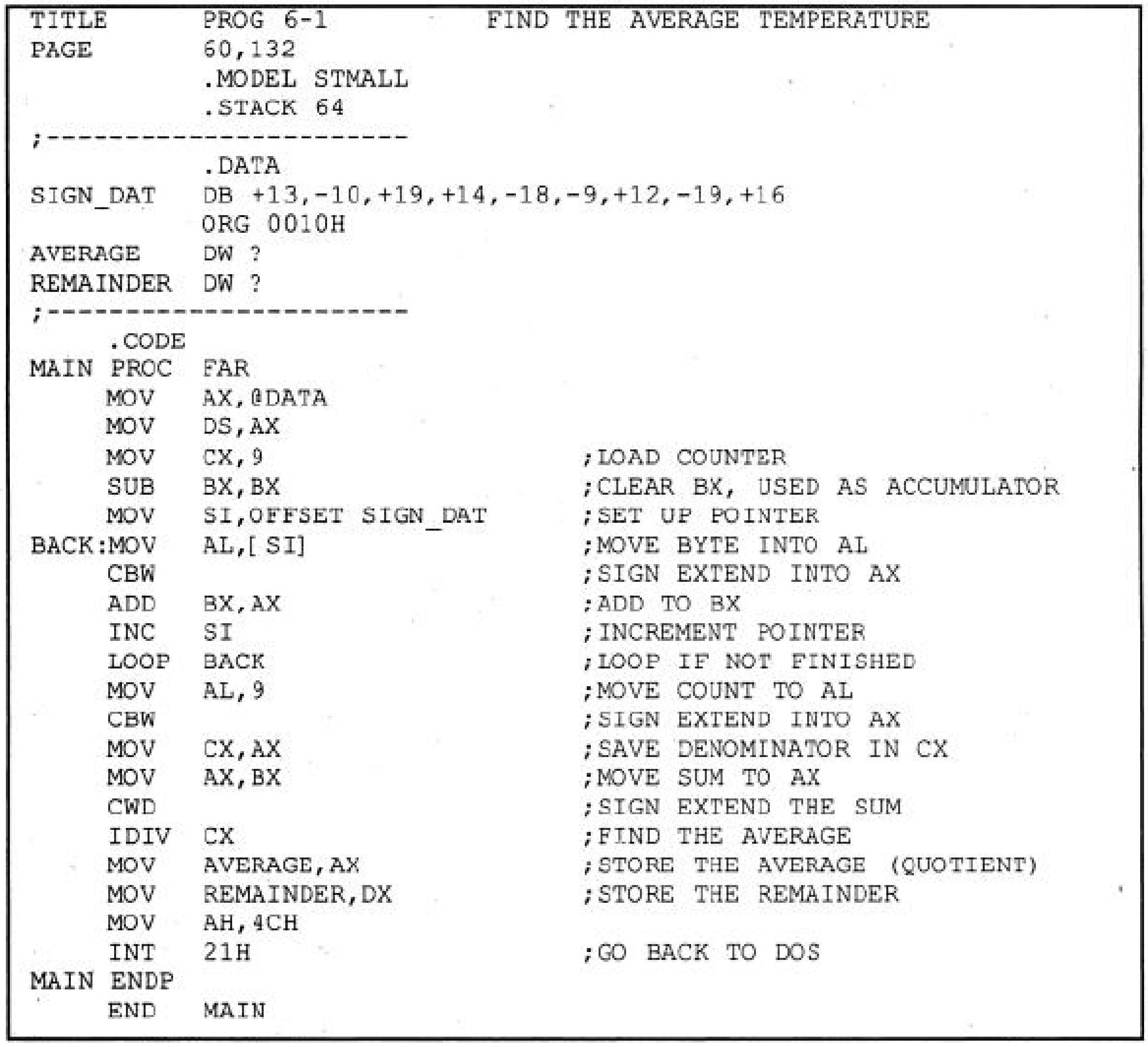
	Before	After	
CH	F0H		EE = -12H
AL	25H	EEH	Quotient
AH	01H	05H	Remainder

Eg2:

IDIV BL

		Before	After	
F0H = -3H	BL	FDH		FB = -5H
	AL	10H	EBH	Quotient
	AH	00H	01H	Remainder

An application of signed number arithmetic is given in the following Program. It computes the average of the Celsius temperatures: +13, -10, + 19, +14, -18, -9, +12, -19, and + 16.

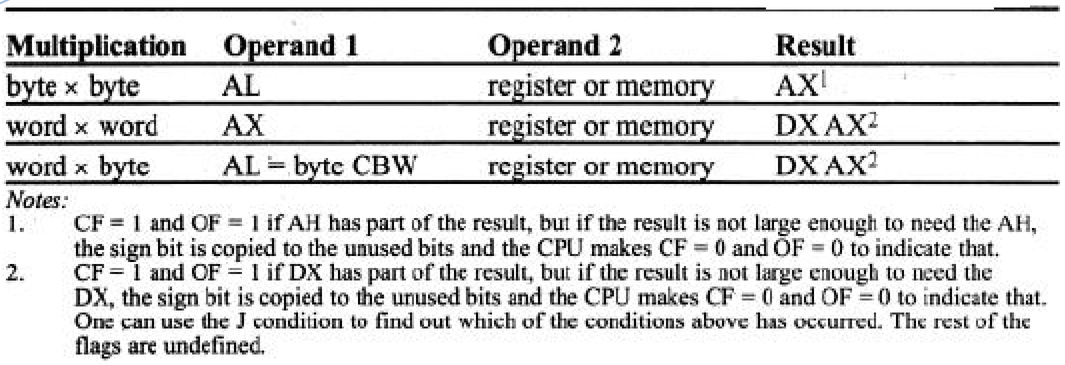


```
TITLE       PROG 6-1          FIND THE AVERAGE TEMPERATURE
PAGE        60,132
            .MODEL STMALL
            .STACK 64
;-----------------------
            .DATA
SIGN_DAT    DB +13,-10,+19,+14,-18,-9,+12,-19,+16
            ORG 0010H
AVERAGE     DW ?
REMAINDER   DW ?
;-----------------------
      .CODE
MAIN PROC   FAR
     MOV    AX,@DATA
     MOV    DS,AX
     MOV    CX,9                    ;LOAD COUNTER
     SUB    BX,BX                   ;CLEAR BX, USED AS ACCUMULATOR
     MOV    SI,OFFSET SIGN_DAT      ;SET UP POINTER
BACK:MOV    AL,[SI]                 ;MOVE BYTE INTO AL
     CBW                            ;SIGN EXTEND INTO AX
     ADD    BX,AX                   ;ADD TO BX
     INC    SI                      ;INCREMENT POINTER
     LOOP   BACK                    ;LOOP IF NOT FINISHED
     MOV    AL,9                    ;MOVE COUNT TO AL
     CBW                            ;SIGN EXTEND INTO AX
     MOV    CX,AX                   ;SAVE DENOMINATOR IN CX
     MOV    AX,BX                   ;MOVE SUM TO AX
     CWD                            ;SIGN EXTEND THE SUM
     IDIV   CX                      ;FIND THE AVERAGE
     MOV    AVERAGE,AX              ;STORE THE AVERAGE (QUOTIENT)
     MOV    REMAINDER,DX            ;STORE THE REMAINDER
     MOV    AH,4CH
     INT    21H                     ;GO BACK TO DOS
MAIN ENDP
     END    MAIN
```

Program 6-1

IMUL (signed number multiplication)

Signed number multiplication is similar in its operation to the unsigned multiplication. The only difference between them is that the operands in signed number operations can be positive or negative; therefore, the result must indicate the sign.



Multiplication	Operand 1	Operand 2	Result
byte × byte	AL	register or memory	AX[1]
word × word	AX	register or memory	DX AX[2]
word × byte	AL = byte CBW	register or memory	DX AX[2]

Notes:
1. CF = 1 and OF = 1 if AH has part of the result, but if the result is not large enough to need the AH, the sign bit is copied to the unused bits and the CPU makes CF = 0 and OF = 0 to indicate that.
2. CF = 1 and OF = 1 if DX has part of the result, but if the result is not large enough to need the DX, the sign bit is copied to the unused bits and the CPU makes CF = 0 and OF = 0 to indicate that. One can use the J condition to find out which of the conditions above has occurred. The rest of the flags are undefined.

Eg1:

IMUL CH

FEH = -02

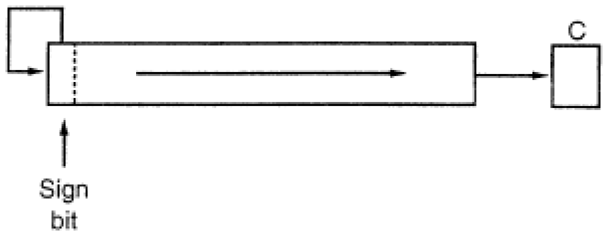
	Before	After	
CH	FEH		
AL	02H	FCH	FFFCH = -04
AH	34H	FFH	

Arithmetic Shift:

The arithmetic shift is used for signed numbers. It is basically the same as the logical shift, except that the sign bit is copied to the shifted bits. SAR (shift arithmetic right) and SAL (shift arithmetic left) are two instructions for the arithmetic shift.

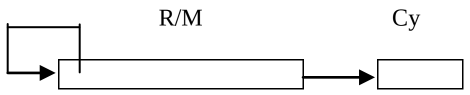
SAR (shift arithmetic right)

```
SAR    destination,count
```



Eg:

SAR BH, CL



Shift right

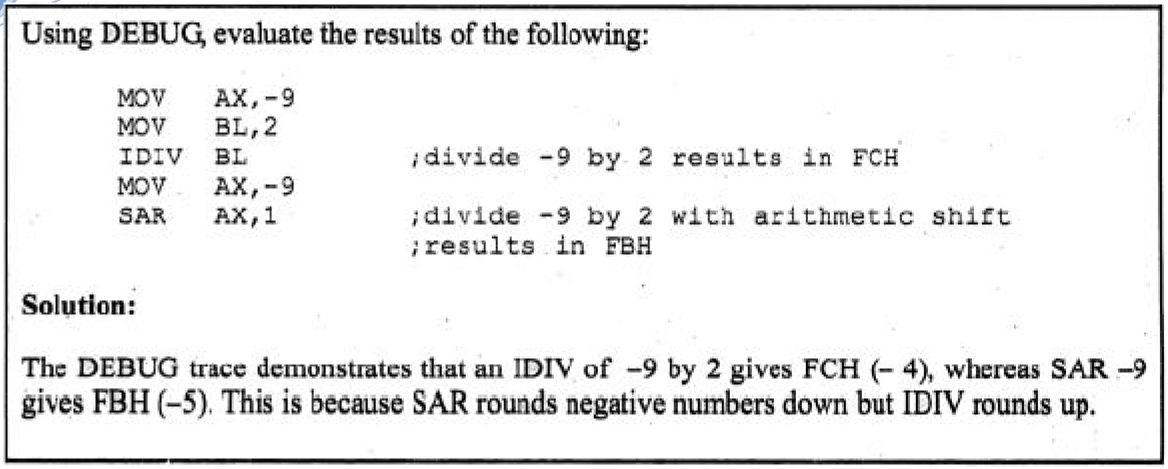
1100 0000 = -40H

1111 0000 = -10H

	Before	*After*
BH	1100 0000	1111 0000
CL	02H	
Cy	1	0

As the bits of the destination are shifted to the right into CF, the empty bits are filled with the sign bit. One can use the SAR instruction to divide a signed number by 2, as shown next:

```
MOV   AL,-10        ;AL=-10=F6H=1111 0110
SAR   AL,1          ;AL is arithmetic shifted right once
                    ;AL=1111 1011=FDH=-5
```



Using DEBUG, evaluate the results of the following:

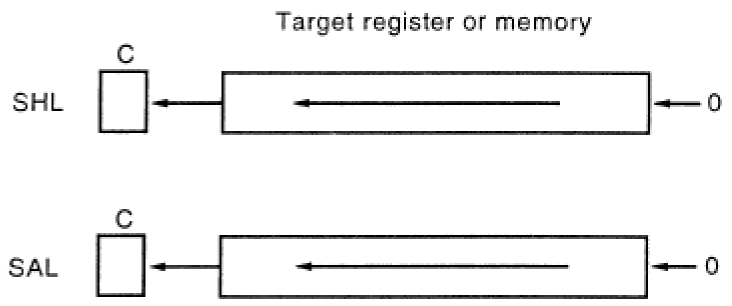
```
MOV   AX,-9
MOV   BL,2
IDIV  BL          ;divide -9 by 2 results in FCH
MOV   AX,-9
SAR   AX,1        ;divide -9 by 2 with arithmetic shift
                  ;results in FBH
```

Solution:

The DEBUG trace demonstrates that an IDIV of −9 by 2 gives FCH (− 4), whereas SAR −9 gives FBH (−5). This is because SAR rounds negative numbers down but IDIV rounds up.

SAL (shift arithmetic left)

SAL & SHL (shift left) do exactly the same thing.



Signed Number Comparison

```
CMP   dest,source
```

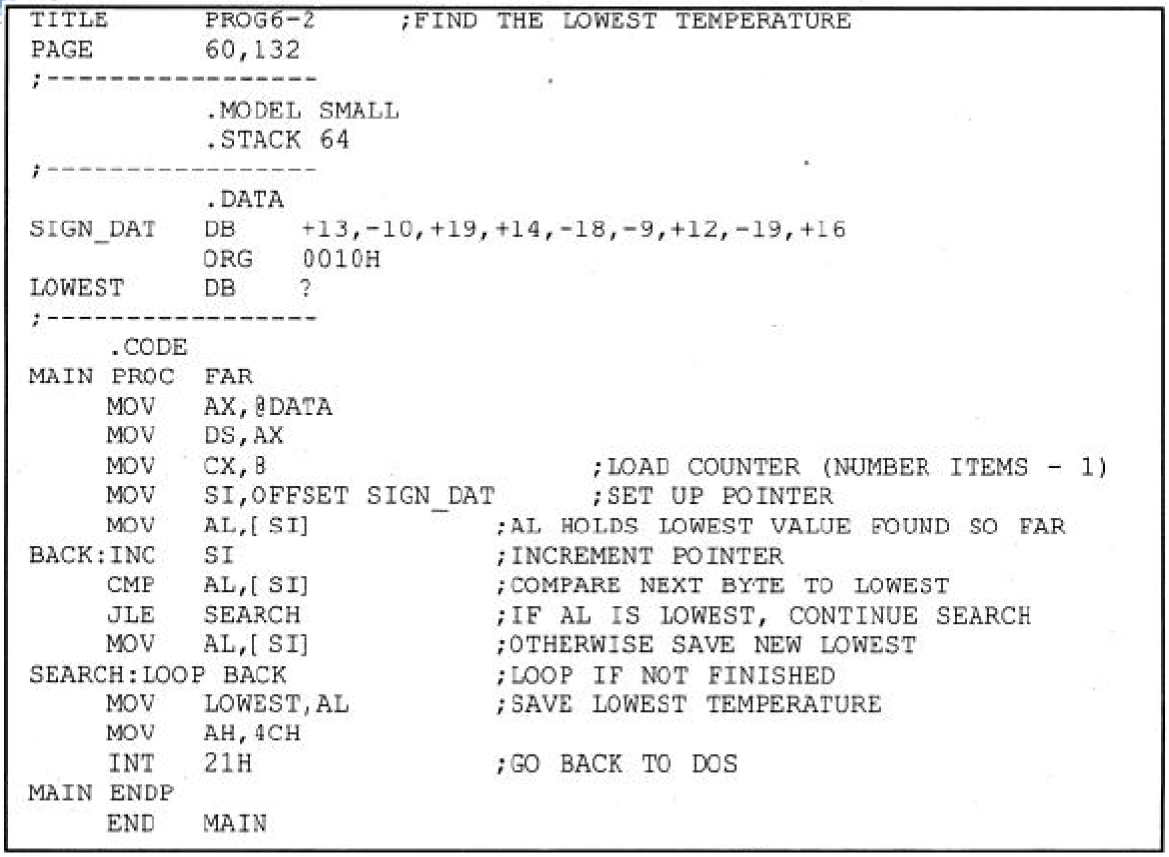
Although the CMP (compare) instruction is the same for both signed and unsigned numbers, the J condition instruction used to make a decision for the signed numbers is different from that used for the unsigned numbers.

- In unsigned number comparisons, CF and ZF are checked for conditions of larger, equal, and smaller.
- In signed number comparison, OF, ZF, and SF are checked.

destination > source	OF=SF or ZF=0
destination = source	ZF=1
destination < source	OF=negation of SF

- The memories used to detect the conditions above are as follows:

JG	Jump Greater	jump if OF=SF or ZF=0
JGE	Jump Greater or Equal	jump if OF=SF
JL	Jump Less	jump if OF=inverse of SF
JLE	Jump Less or Equal	jump if OF=inverse of SF or ZF=1
JE	Jump if Equal	jump of ZF = 1



```
TITLE       PROG6-2      ;FIND THE LOWEST TEMPERATURE
PAGE        60,132
;------------------
            .MODEL SMALL
            .STACK 64
;------------------
            .DATA
SIGN_DAT    DB     +13,-10,+19,+14,-18,-9,+12,-19,+16
            ORG    0010H
LOWEST      DB     ?
;------------------
      .CODE
MAIN PROC   FAR
      MOV   AX,@DATA
      MOV   DS,AX
      MOV   CX,8                      ;LOAD COUNTER (NUMBER ITEMS - 1)
      MOV   SI,OFFSET SIGN_DAT        ;SET UP POINTER
      MOV   AL,[SI]             ;AL HOLDS LOWEST VALUE FOUND SO FAR
BACK:INC    SI                  ;INCREMENT POINTER
      CMP   AL,[SI]             ;COMPARE NEXT BYTE TO LOWEST
      JLE   SEARCH              ;IF AL IS LOWEST, CONTINUE SEARCH
      MOV   AL,[SI]             ;OTHERWISE SAVE NEW LOWEST
SEARCH:LOOP BACK                ;LOOP IF NOT FINISHED
      MOV   LOWEST,AL           ;SAVE LOWEST TEMPERATURE
      MOV   AH,4CH
      INT   21H                 ;GO BACK TO DOS
MAIN ENDP
      END   MAIN
```

Program 6-2

STRING & TABLE OPERATIONS:

- There is a group of instructions referred to as string instructions in the x86 family of microprocessors.
- They are capable of performing operations on a series of operands located in consecutive memory locations.
- For example, while the CMP instruction can compare only 2 bytes (or words) of data, the CMPS (compare string) instruction is capable of comparing two arrays of data located in memory locations pointed at by the SI and DI registers. These instructions are very powerful and can be used in many applications,

Use of SI and DI, DS and ES in String Instructions:

- For string operations to work, designers of CPUs must set aside certain registers for specific functions. These registers must permanently provide the source and destination operands.
- In 088/86 microprocessor, the SI and DI registers always point to the source and destination operands, respectively.
- To generate the physical address, the 8088/86 always uses SI as the offset of the DS (data segment) register and DI as the offset of ES (extra segment).
- The ES register must be initialized for the string operation(s) to work.

Byte and Word Operands in String Instructions:

- In each of the string instructions, the operand can be a byte or a word.
- Operands are distinguished by the letters B (byte) and W (word) in the instruction mnemonic.

DF, the Direction Flag:

- To process operands located in consecutive memory locations; it requires that, the pointer be incremented or decremented.
- In string operations this is achieved by the direction flag. Of the 16 bits of the flag register (D0 – D15), bit 11 (D10) is set aside for the direction flag (DF).
- It is the job of the string instruction to increment or decrement the SI and DI pointers; but it is the job of the programmer to specify the choice of increment or decrement by setting the direction flag to high or low.
- The instructions CLD (*clear direction flag*) and STD (*set direction flag*) are specifically designed for the purpose.
- CLD (clear direction flag) will reset (put to zero) the DF, indicating that the string instruction should increment the pointers automatically. This is referred to as *auto-increment*.
- STD (set the direction flag) sets DF to 1, indicating to the string instruction that the pointers SI and DI should be decremented automatically. This is referred to as *auto-decrement*.

Table: Summary of String Operations

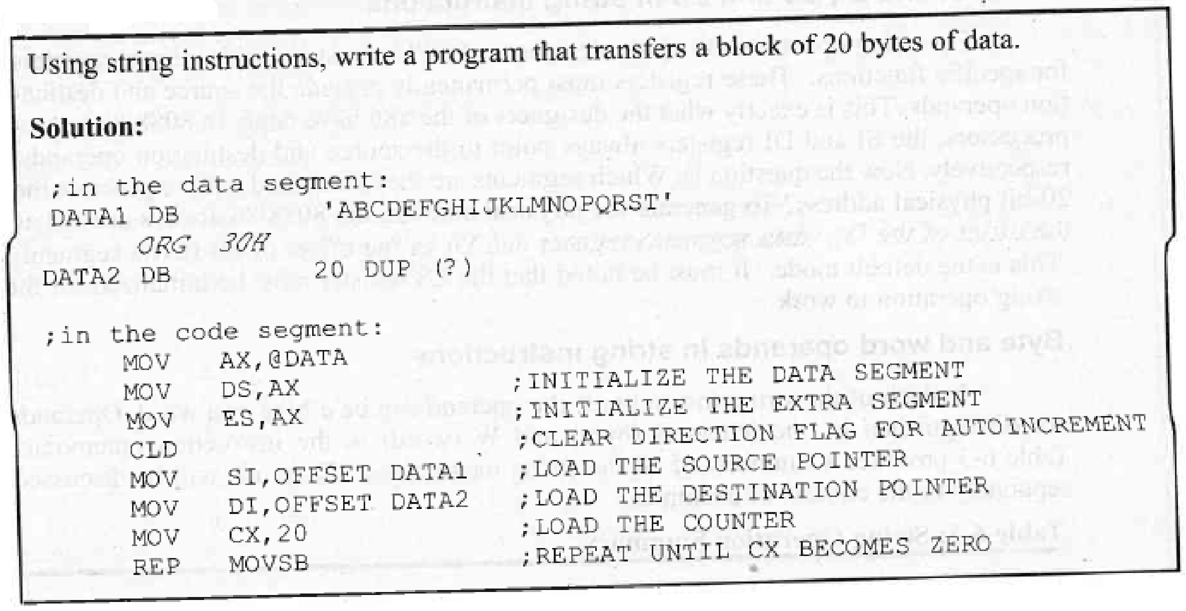
Instruction	Mnemonic	Destination	Source	Prefix
Move string byte	MOVSB	ES: DI	DS: SI	REP
Move string word	MOVSW	ES: DI	DS: SI	REP
Store string byte	STOSB	ES: DI	AL	REP
Store string word	STOSW	ES: DI	AX	REP
Load string byte	LODSB	AL	DS: SI	None
Load string word	LODSW	AX	DS: SI	None
Compare string byte	CMPSB	ES: DI	DS: SI	REPE/REPNE
Compare string word	CMPSW	ES: DI	DS: SI	REPE/REPNE
Scan string byte	SCASB	ES: DI	AL	REPE/REPNE
Scan string word	SCASW	ES: DI	AX	REPE/REPNE

REP/REPZ/REPNZ Prefix:

- **REP (repeat)** prefix allows a string instruction to perform the operation repeatedly.
- REP assumes that CX holds the number of times that the instruction should be repeated.
- In other words, the REP prefix tells the CPU to perform the string operation and then decrements the CX register automatically. This process is repeated until CX becomes zero.

- o **REPZ (repeat zero)/REPE (repeat equal)** repeat the string operation as long as source and destination operands are equal (ZF = 1) or until CX becomes zero.
- o **REPNZ (repeat not zero)/REPNE (repeat not equal)** repeat the string operation as long as source and destination operands are not equal (ZF = 0) or until CX becomes zero.

Instruction Code	Condition for Exit
REP	CX = 0
REPE/REPZ	CX = 0 or ZF = 0
REPNE/REPNZ	CX = 0 or ZF = 1



Using string instructions, write a program that transfers a block of 20 bytes of data.

Solution:

```
;in the data segment:
DATA1 DB        'ABCDEFGHIJKLMNOPQRST'
        ORG   30H
DATA2 DB          20  DUP  (?)

;in the code segment:
      MOV    AX,@DATA
      MOV    DS,AX                  ;INITIALIZE THE DATA SEGMENT
      MOV    ES,AX                  ;INITIALIZE THE EXTRA SEGMENT
      CLD                           ;CLEAR DIRECTION FLAG FOR AUTOINCREMENT
      MOV    SI,OFFSET DATA1        ;LOAD THE SOURCE POINTER
      MOV    DI,OFFSET DATA2        ;LOAD THE DESTINATION POINTER
      MOV    CX,20                  ;LOAD THE COUNTER
      REP    MOVSB                  ;REPEAT UNTIL CX BECOMES ZERO
```

- ✓ After the transfer of every byte by the MOVSB instruction, both the SI and DI registers are incremented automatically once only (notice CLD).
- ✓ The REP prefix causes the CX counter to be decremented and MOVSB is repeated until CX becomes zero.
- ✓ An alternative solution for above Example would change only two lines of code:

MOV CX, 10

REP MOVSB

- ✓ In this case the MOVSW will transfer a word (2 bytes) at a time and increment the SI and DI registers each twice. REP will repeat that process until CX becomes zero. Notice that, the CX has the value of 10 in it; since 10 words is equal to 20 bytes.

STOS and LODS Instructions:

STOSB – stores the byte in the AL register into memory location pointed at by ES: DI and then increment DI once (if DF = 0) or decrement DI once (if DF = 1).

STOSW – stores the content of AX in memory locations ES: DI and ES: DI+1 (AL into ES: DI and AH into ES: Dl+1) then increments DI twice (if DF = 0) or decrements DI twice (if DF = 1).

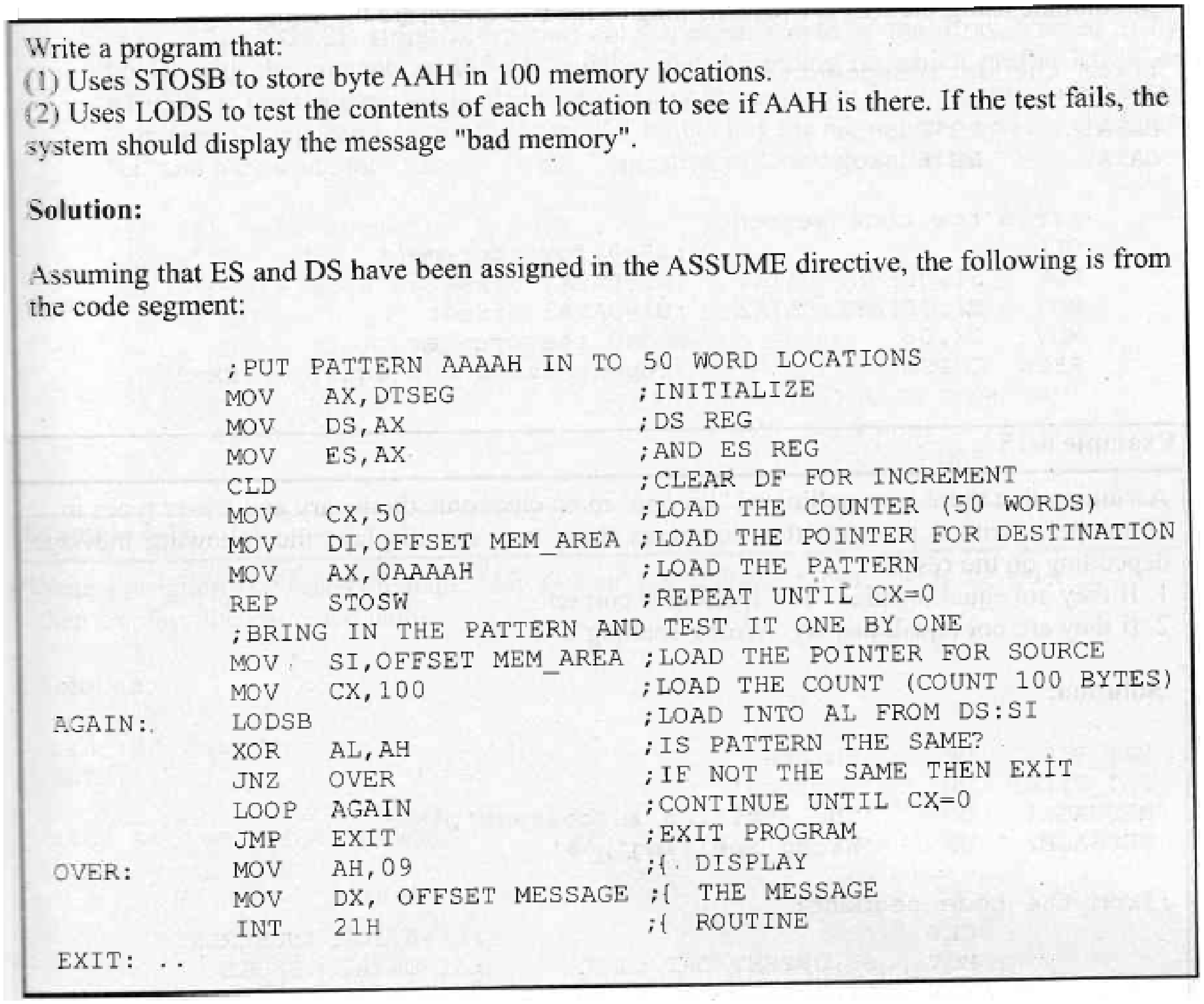
LODSB – loads the contents of memory location pointed at by DS: SI into AL and increments SI once (if DF = 0) or decrements SI once (if DF = l).

LODSW – loads the content of memory locations pointed at by DS: SI into AL and DS: SI+l into AH. The SI is incremented twice if DF = 0 or SI is decremented twice if DF = 1.

- LODS is never used with a REP prefix.

Testing Memory using STOSB and LODSB:

- ✓ The following Example uses string instructions STOSB and LODSB to test an area of RAM memory.
- ✓ First AAH is written into 100 locations by using word-sized operand AAAAH and a count of 50.
- ✓ In the test part, LODSB brings in the contents of memory locations into AL one by one, and each time it is eXclusive-ORed with AAH (the AH register has the hex value of AA).
    - o If they are the same, ZF = l and the process is continued.
    - o Otherwise, the pattern written there by the previous routine is not there and the program will exit.



Write a program that:
(1) Uses STOSB to store byte AAH in 100 memory locations.
(2) Uses LODS to test the contents of each location to see if AAH is there. If the test fails, the system should display the message "bad memory".

Solution:

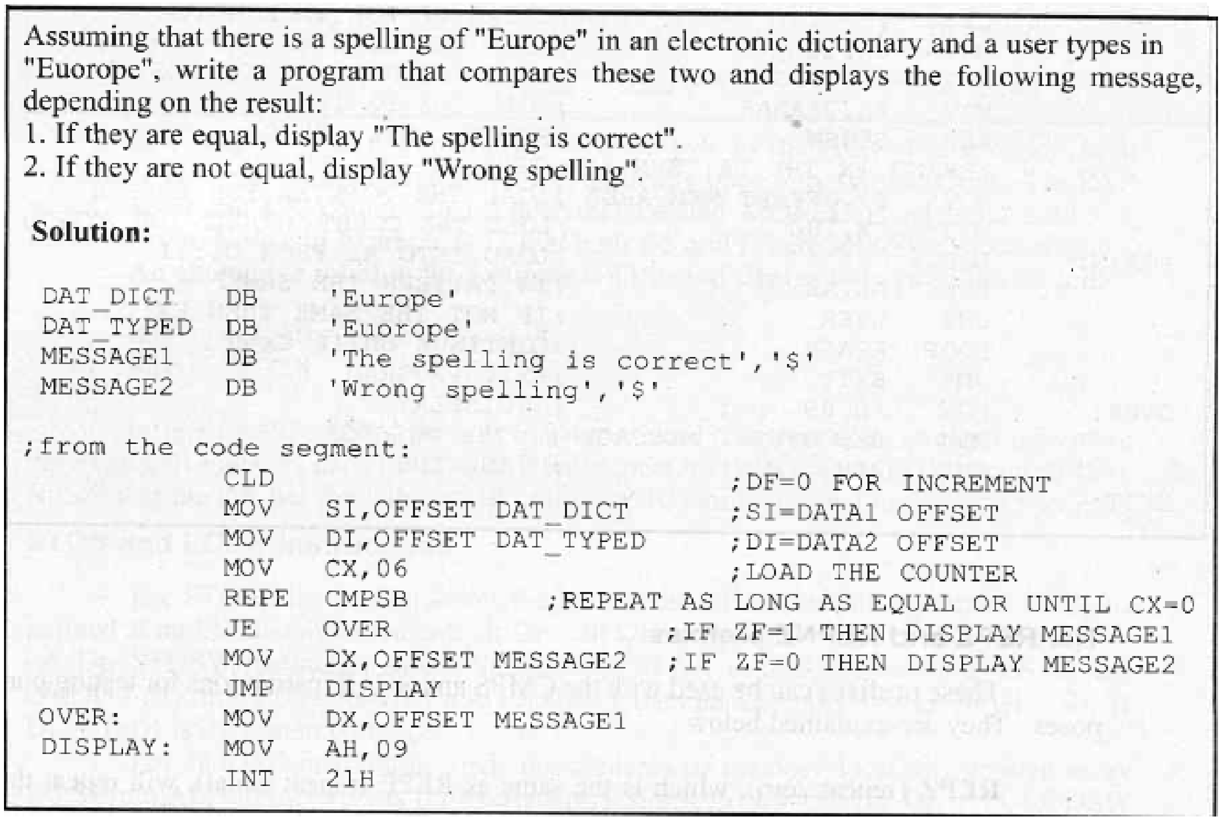
Assuming that ES and DS have been assigned in the ASSUME directive, the following is from the code segment:

```
            ;PUT PATTERN AAAAH IN TO 50 WORD LOCATIONS
            MOV    AX,DTSEG              ;INITIALIZE
            MOV    DS,AX                 ;DS REG
            MOV    ES,AX                 ;AND ES REG
            CLD                          ;CLEAR DF FOR INCREMENT
            MOV    CX,50                 ;LOAD THE COUNTER (50 WORDS)
            MOV    DI,OFFSET MEM_AREA    ;LOAD THE POINTER FOR DESTINATION
            MOV    AX,0AAAAH             ;LOAD THE PATTERN
            REP    STOSW                 ;REPEAT UNTIL CX=0
            ;BRING IN THE PATTERN AND TEST IT ONE BY ONE
            MOV    SI,OFFSET MEM_AREA    ;LOAD THE POINTER FOR SOURCE
            MOV    CX,100                ;LOAD THE COUNT (COUNT 100 BYTES)
AGAIN:      LODSB                        ;LOAD INTO AL FROM DS:SI
            XOR    AL,AH                 ;IS PATTERN THE SAME?
            JNZ    OVER                  ;IF NOT THE SAME THEN EXIT
            LOOP   AGAIN                 ;CONTINUE UNTIL CX=0
            JMP    EXIT                  ;EXIT PROGRAM
OVER:       MOV    AH,09                 ;{ DISPLAY
            MOV    DX, OFFSET MESSAGE    ;{  THE MESSAGE
            INT    21H                   ;{  ROUTINE
EXIT: ..
```

CMPS (Compare String):

- CMPS allows the comparison of two arrays of data pointed at by the SI and DI registers.
- One can test for the equality or inequality of data by the use of REPE or REPNE prefixes, respectively.
- The comparison can be performed a byte at a time or a word at time by using CMPSB or CMPSW forms of the instruction.

For example, if comparing "Euorop" and "Europe" for equality, the comparison will continue using the REPE CMPS as long as the two arrays are the same.



Assuming that there is a spelling of "Europe" in an electronic dictionary and a user types in "Euorope", write a program that compares these two and displays the following message, depending on the result:
1. If they are equal, display "The spelling is correct".
2. If they are not equal, display "Wrong spelling".

Solution:

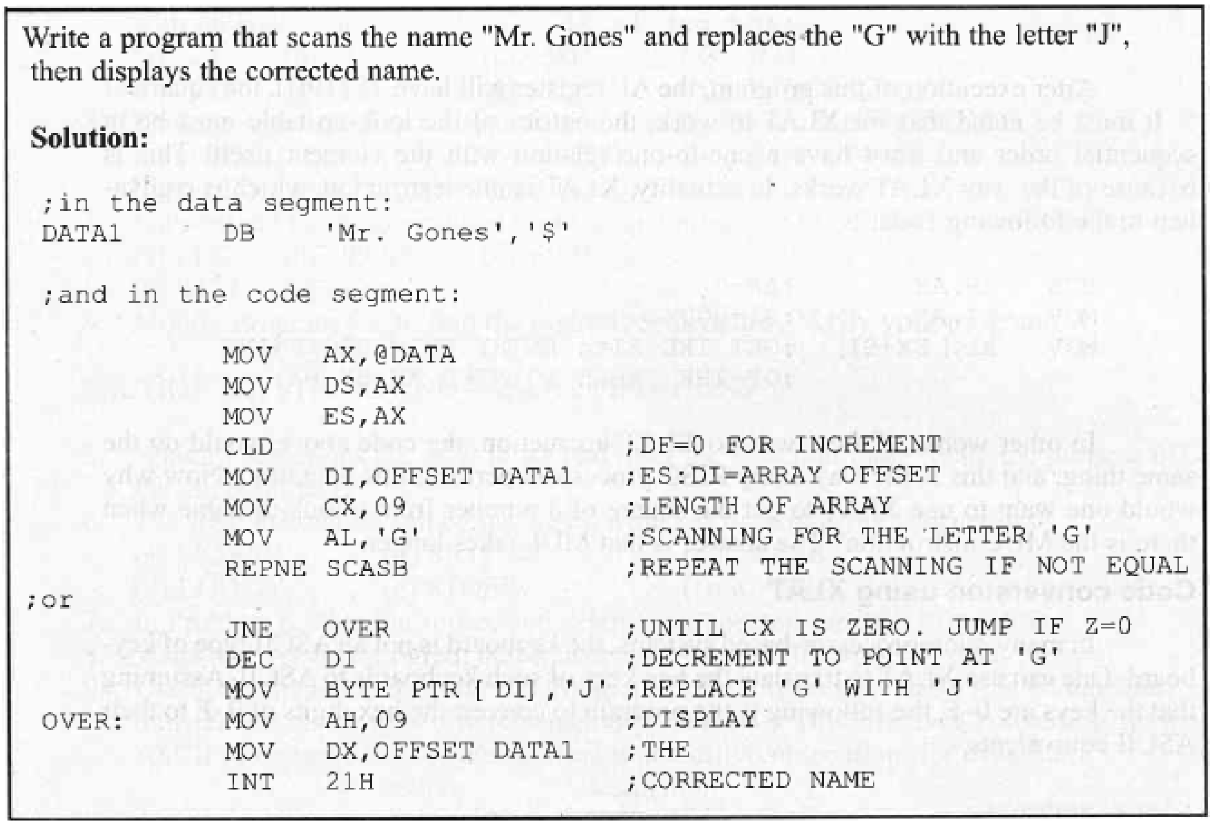
```
 DAT_DICT    DB    'Europe'
 DAT_TYPED   DB    'Euorope'
 MESSAGE1    DB    'The spelling is correct','$'
 MESSAGE2    DB    'Wrong spelling','$'

;from the code segment:
             CLD                                ;DF=0 FOR INCREMENT
             MOV   SI,OFFSET DAT_DICT           ;SI=DATA1 OFFSET
             MOV   DI,OFFSET DAT_TYPED          ;DI=DATA2 OFFSET
             MOV   CX,06                        ;LOAD THE COUNTER
             REPE  CMPSB           ;REPEAT AS LONG AS EQUAL OR UNTIL CX=0
             JE    OVER                   ;IF ZF=1 THEN DISPLAY MESSAGE1
             MOV   DX,OFFSET MESSAGE2     ;IF ZF=0 THEN DISPLAY MESSAGE2
             JMP   DISPLAY
 OVER:       MOV   DX,OFFSET MESSAGE1
 DISPLAY:    MOV   AH,09
             INT   21H
```

- ✓ Here, the two arrays are to be compared letter by letter.
- ✓ The first characters pointed at by SI and DI are compared. In this case they are the same ("E"), so the zero flag is set to 1 and both SI and DI are incremented.
- ✓ Since ZF = 1, the REPE prefix repeats the comparison.
- ✓ This process is repeated until the third letter is reached. The third letters “o” and "r" are not the same; therefore, ZF = 0, and the comparison will stop.

SCAS (Scan String):

- SCASB – compares each byte of the array pointed at by ES: DI with the contents of the AL register, and depending on which prefix, REPE or REPNE, is used, a decision is made for equality or inequality.

- For example, in the array "Mr. Gones", one can scan for the letter "G" by loading the AL register with the character "G" and then using the "REPNE SCASB" operation to look for that letter.



Write a program that scans the name "Mr. Gones" and replaces the "G" with the letter "J", then displays the corrected name.

Solution:

```
;in the data segment:
DATA1      DB     'Mr. Gones','$'

;and in the code segment:
           MOV    AX,@DATA
           MOV    DS,AX
           MOV    ES,AX
           CLD                        ;DF=0 FOR INCREMENT
           MOV    DI,OFFSET DATA1     ;ES:DI=ARRAY OFFSET
           MOV    CX,09               ;LENGTH OF ARRAY
           MOV    AL,'G'              ;SCANNING FOR THE LETTER 'G'
           REPNE SCASB                ;REPEAT THE SCANNING IF NOT EQUAL
;or
           JNE    OVER                ;UNTIL CX IS ZERO. JUMP IF Z=0
           DEC    DI                  ;DECREMENT TO POINT AT 'G'
           MOV    BYTE PTR [DI],'J'   ;REPLACE 'G' WITH 'J'
OVER:      MOV    AH,09               ;DISPLAY
           MOV    DX,OFFSET DATA1     ;THE
           INT    21H                 ;CORRECTED NAME
```

- ✓ Here, the letter "G" is compared with "M".
- ✓ Since they are not equal, DI is incremented and CX is decremented, and the scanning is repeated until the letter "G" is found or the CX register is zero. In this example, since "G" is found, ZF = 1, indicating that there is a letter "G" in the array.

Replacing the Scanned Character:

- SCASB can be used to search for a character in an array, and if it is found, it will be replaced with the desired character. (See Example given above).
- In string operations the pointer is incremented after each execution (if DF = 0). Therefore, in the example above, DI must be decremented, causing the pointer to point to the scanned character and then replace it.

XLAT Instruction and Look-Up Tables:

- There is often a need in computer applications for a table that holds some important information. To access the elements of the table, 8088/86 microprocessors provide the XLAT (translate) instruction.

- To understand the XLAT instruction, one must first understand tables. The table is commonly referred to as a look-up table.
- Assume that one needs a table for the values of x^2, where x is between 0 and 9. First the table is generated and stored in memory:

```
SQUR_TABLE   DB    0,1,4,9,16,25,36,49,64,81
```

- It is possible to access the square of any number from 0 to 9 by the use of XLAT instruction.
  - ✓ To do that, the register BX must have the offset address of the look-up table, and the number whose square is sought must be in the AL register.
  - ✓ Then after the execution of XLAT, the AL register will have the square of the number.
- The following shows how to get the square of 5 from the table:

```
MOV    BX,OFFSET SQUR_TABLE ;load the offset address of table
MOV    AL,05            ;AL=05 will retrieve 6th element
XLAT                    ;pull the element out of table
                        ;and put in AL
```

- After execution of this program, the AL register will have 25 (19H), the square of 5.
- It must be noted that, for XLAT to work the entries of the look-up table must be in sequential order and must have a one-to-one relation with the element itself. This is because of the way XLAT work.
- In actuality, XLAT is one instruction, which is equivalent to the following code:

```
SUB    AH,AH          ;AH=0
MOV    SI,AX          ;SI=000X
MOV    AL,[BX+SI]     ;GET THE SIth ENTRY FROM BEGINNING
                      ;OF THE TABLE POINTED AT BY BX
```

Code Conversion using XLAT:

- In many microprocessor-based systems, the keyboard is not an ASCII type of keyboard.
- One can use XLAT to translate the hex keys of such keyboards to ASCII.
- Assuming that the keys are 0-F, the following is the program to convert the hex digits of 0-F to their ASCII equivalents.

```
;data segment:
ASC_TABL     DB    '0','1','2','3','4','5','6','7','8'
             DB    '9','A','B','C','D','E','F'
HEX_VALU     DB    ?
ASC_VALU     DB    ?
;code segment:
             MOV   BX,OFFSET ASC_TABL       ;BX= TABLE OFFSET
             MOV   AL,HEX_VALU              ;AL=THE HEX DATA
             XLAT                      ;GET THE ASCII EQUIVALENT
             MOV   ASC_VALU,AL              ;MOVE IT TO MEMORY
```

MEMORY & MEMORY INTERFACING

SEMICONDUCTOR MEMORIES

» In the design of computers, semiconductor memories are used as primary storage for code and data. Semiconductor memories are connected directly to the CPU. For this reason, semiconductor memories are referred to as *primary memory*. Most widely used semiconductor memories are ROM and RAM.

» ***Read-only memory (ROM)*** *contains system software and permanent system data.*

» ***Random access memory (RAM)*** or ***read/write memory*** contains temporary data and application software.

Memory Organization:

» The number of bits that a semiconductor memory chip can store is called its *capacity*. It can be in the units of K bits (kilobits)/M bits (megabits).

» Memory chips are organized into a number of locations within the IC. Each location can hold 1 bit, 4-bits, 8-bits, or even 16-bits.

» Each memory chip contains 2^x locations, where x is the number of *address pins* on the chip.

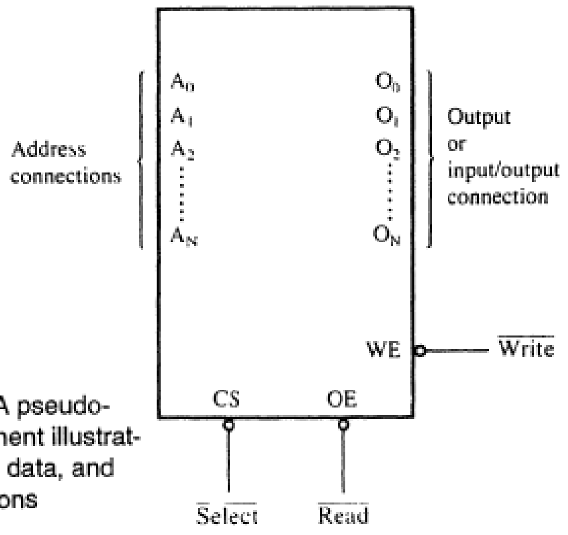
» Each location contains y bits, where y is the number of *data pins* on the chip.

» The entire chip will contain 2^x x y bits – the *capacity* of the chip.

The pin connections common to all memory devices are –

» ***Address Connections.*** All memory devices have address inputs that select a memory location within the memory device. Address inputs are always labeled from A_0 to A_n (Note, 'n' is one less than the total number of address pins). The number of address pins found on a memory device is determined by the number of memory locations found within it.

» ***Data Connections.*** All memory devices have a set of data outputs or input/outputs. The device illustrated in the following Figure has a common set of I/O (input/output) connections.



A pseudo-memory component illustrating the address, data, and control connections

- » As shown in the Fig. above; the memory chips have CS (chip select) pin that must be activated for memory contents to be accessed. That means, no data can be written into or read form the memory chip unless CS is activated.
- » Sometimes, OE (output enable)/RD (read)/WR (write) pins may also be present along with CS pin.

Examples: 1] A given memory chip has 12 address pins and 8 data pins. Find the memory organization and the capacity.

Solution:

- ⇨ Memory chip has 12 address lines ↔ 2^{12} = 4,096 locations.
- ⇨ Memory chip has 8 data lines ↔ Each location hold 8 bits of data.
- ⇨ Thus, the memory organization is 4,096 x 8 = 4K x 8 = 32K bits capacity.

Examples: 2] A 512K memory chip has 8 data pins. Find the organization.

Solution:

- ⇨ The memory chip has 8 data lines ↔ Each location within the chip can hold 8 bits of data.
- ⇨ Given, the capacity of the memory chip = 512K.
- ⇨ Hence, the locations within the memory chip = 512K / 8 = 64K.
- ⇨ Since, 2^{16} = 64K; the memory chip has 16 address lines.
- ⇨ Hence, the memory organization is: 64K x 8 = 512K bits capacity.

MEMORY ADDRESS DECODING:

- o Consider a 32K x 8 capacity memory chip. This chip has 15 (2^{15} = 32K) address lines and 8 data lines.
- o Suppose, this memory chip is to be interfaced to x86 microprocessor, which is having 20 address lines and 16 data lines.
- o This means that, the microprocessor sends out a 20-bit memory address whenever it reads or writes data. Hence there is a mismatch that must be corrected.
- o The *decoder* corrects the mismatch by decoding the address pins that do not connect to the memory component.

Simple Logic Gates as Address Decoder:

- ✓ The CS (chip select) input pin (in any memory chip) is usually active low and can be activated using some simple logic gates; such as NAND gate and Inverters.
- ✓ The following Fig. shows some simple NAND gate decoding for memory chips, along with the address range calculations.

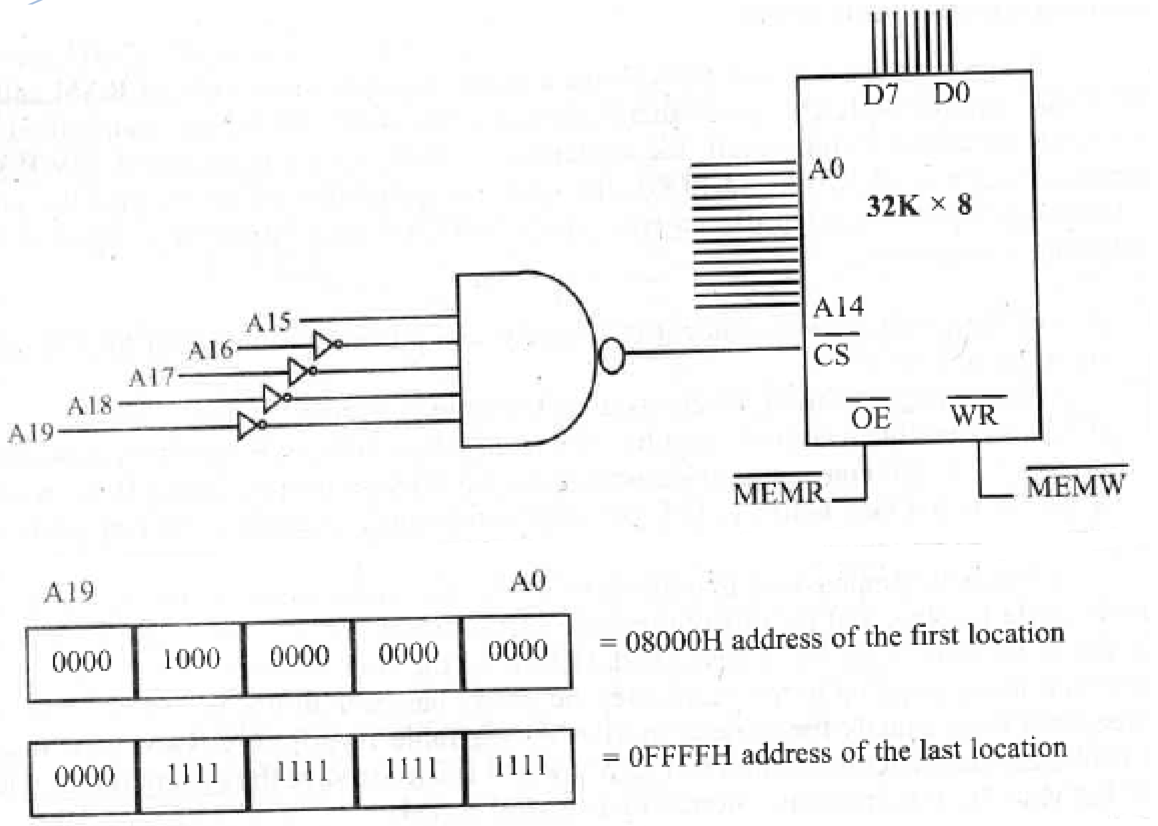


Fig: Simple Logic Gates as Decoder (1)

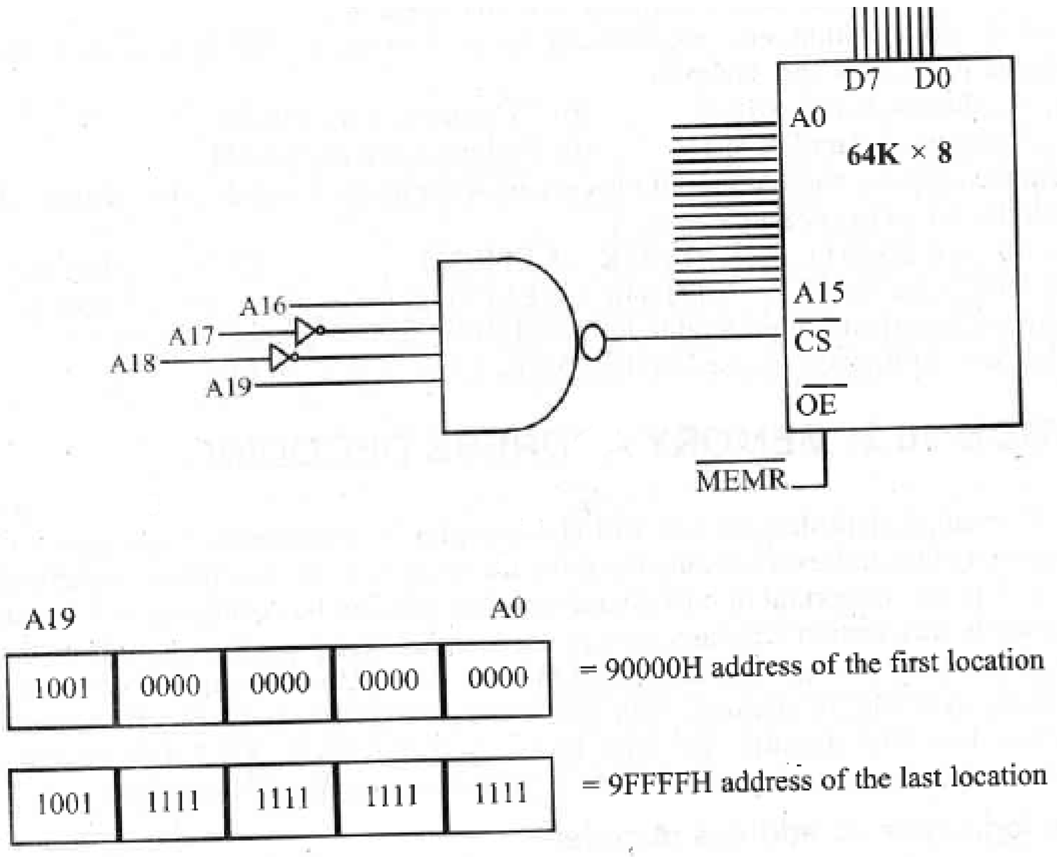


Fig: Simple Logic Gates as Decoder (2)

- Notice that, the output of the NAND gate is active low and that the CS pin is also active low. That makes them a perfect match.
- Also notice that Al9-A16 must equal 1001 in order for CS to be activated. This results in the assignment of addresses 9000H to 9FFFFH to this memory block.

Referring to above Fig., we see that the memory chip has 64K bytes of space. Show the calculation that verifies that address range 90000 to 9FFFFH is comprised of 64K bytes.

Solution:

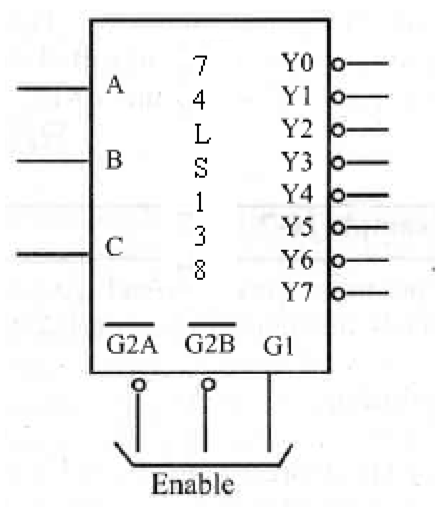
To calculate the total number of bytes for a given memory address range, subtract the two addresses and add 1 to get the total bytes in hex. Then the hex number is converted to decimal and divided by 1024 to get K bytes.

```
 9FFFF          FFFF
-90000        +    1
 0FFFF        10000 hex = 65,536 decimal = 64K
```

Using the 74LS138 as Decoder:

- The 74LS138 has 8 NAND gates in it; therefore, a single chip can control 8 blocks of memory.
- In 74LS138 decoder; the three inputs A, B, C generates eight active low outputs Y0 to Y7.

Block Diagram



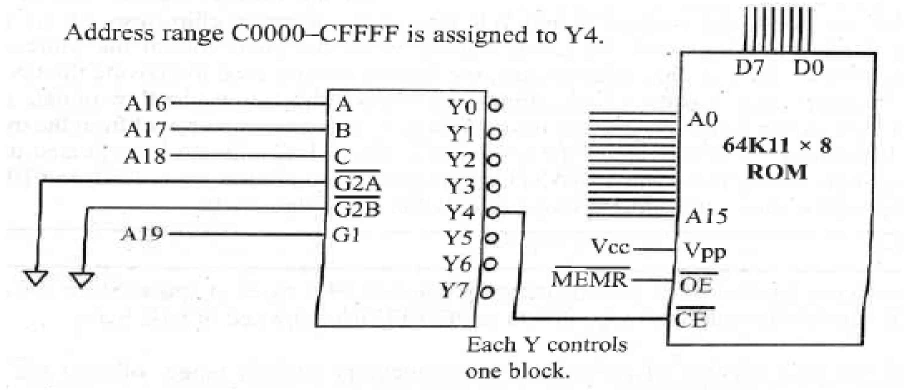
Function Table

Inputs					Outputs							
Enable		Select			Outputs							
G1	G2	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

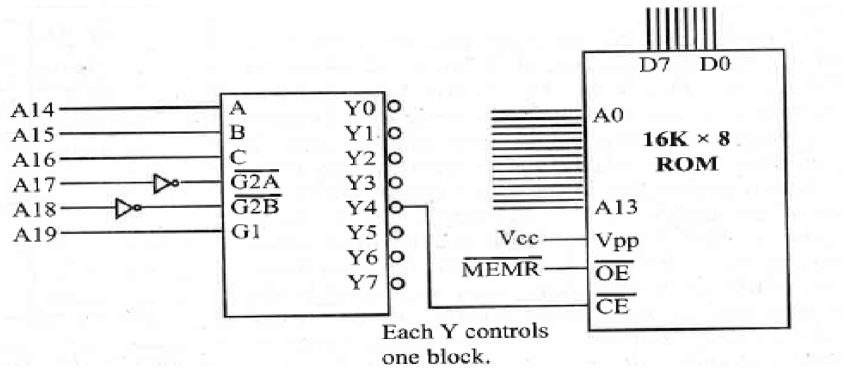
- Each Y output can be connected to the CS of memory chip, allowing control of 8 memory blocks by a single 74LS138.

✓ Consider the following memory decoding diagram. We have, A0-A15 from the CPU, directly connected to A0-A15 of the memory chip.

✓ A16-A18 are used for the A, B, and C inputs of 74LS138; A19 is controlling G1 pin. G2A and G2B are grounded.



- ✓ To enable 74LS138; G2A = 0, G2B = 0; and G1 = 1.
- ✓ To select Y4; CBA = 100.
- ✓ This gives the address range (for the memory chip controlled by Y4): C0000H to CFFFFH.



Looking at the design in above Fig. , find the address range for (a) Y4, (b) Y2, and (c) Y7, and verify the block size controlled by each Y.

Solution:

(a) The address range for Y4 is calculated as follows.

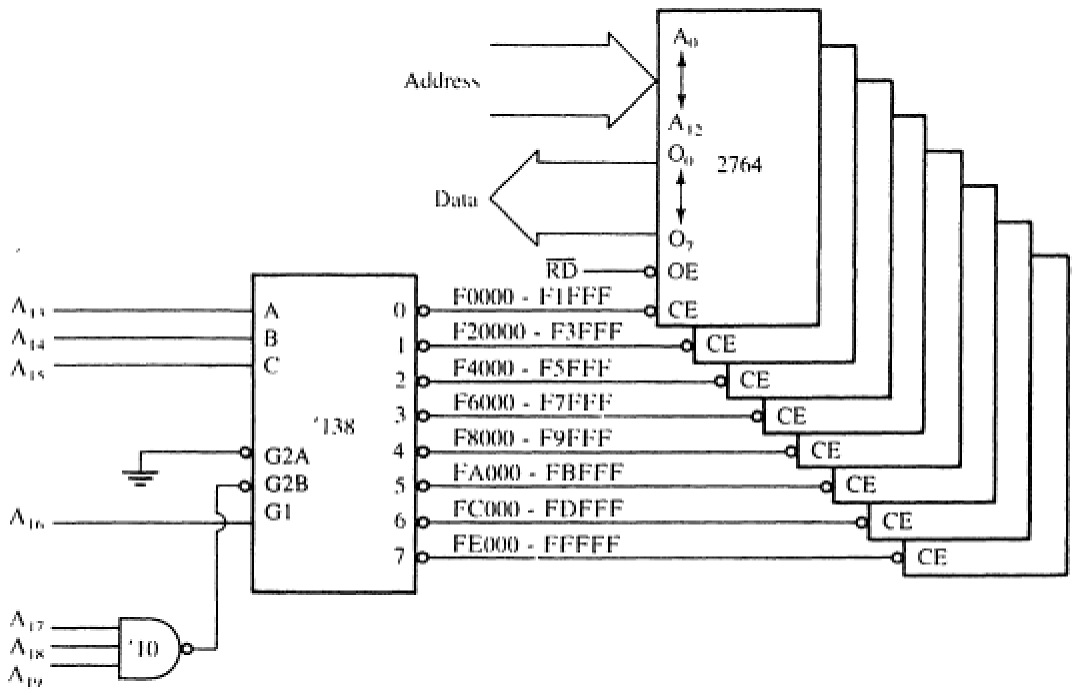
A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

The above shows that the range for Y4 is F0000H to F3FFFH. In Figure 10-13, notice that A19, A18, and A17 must be 1 for the decoder to be activated. Y4 will be selected when A16 A15 A14 = 100 (4 in binary). The remaining A13–A0 will be 0 for the lowest address and 1 for the highest address.

(b) The address range for Y2 is E8000H to EBFFFH.

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(c) The address range for Y7 is FC000H to FFFFFH. Notice that FFFFF – FC000H = 3FFFH, which is equal to 16,383 in decimal. Adding 1 to it because of the 0 location, we have 16,384. 16,384/1024 = 16K, the block (chip) size.



A circuit that uses eight 2764 EPROMs for a 64K × 8 section of memory in an 8088 microprocessor -based system. The addresses selected in this circuit are F0000H–FFFFFH.

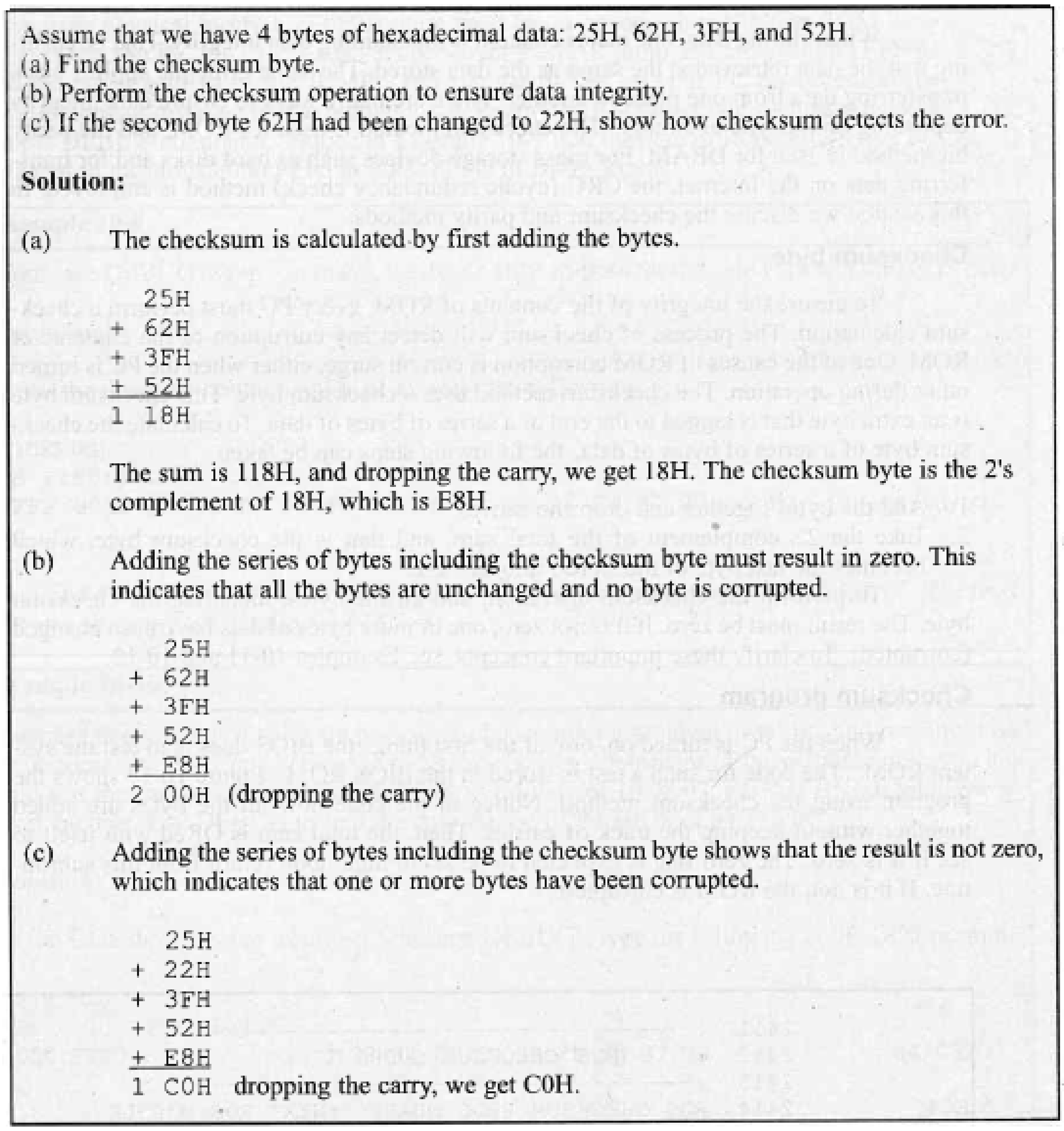
DATA INTEGRITY IN RAM & ROM:

- When storing data, one major concern is maintaining *data integrity – ensuring that, the data retrieved is the same as the data stored.*
- The same principle applies when transferring data from one place to another – *ensuring that, the data received is the same as the data transmitted.*
- There are many way to ensure data integrity depending on the type of storage.
- The *checksum* method is used for ROM and the *parity bit* method is used for DRAM.
- For mass storage devices such as hard disks and for transferring data on the Internet, the *CRC* (*cyclic redundancy check*) method is employed.

Checksum Byte:

- During the current surge, or when the PC is turned on, or during operation, the contents of the ROM may be corrupted.
- To ensure the integrity of the contents of ROM, every PC must perform a checksum calculation. The process of checksum will detect any corruption of the contents of ROM.
- The checksum method uses a checksum byte. This checksum byte is an extra byte that is tagged to the end of a series of bytes of data.
- To calculate the checksum byte of a series of bytes of data, the following steps can be taken .
  1. Add the bytes together and drop the carries.
  2. Take the 2's complement of the total sum, and that is the checksum byte, which becomes the last byte of the stored information.

- To perform the checksum operation, add all the bytes, including the checksum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted).



Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH, and 52H.
(a) Find the checksum byte.
(b) Perform the checksum operation to ensure data integrity.
(c) If the second byte 62H had been changed to 22H, show how checksum detects the error.

Solution:

(a) The checksum is calculated by first adding the bytes.

```
  25H
+ 62H
+ 3FH
+ 52H
1 18H
```

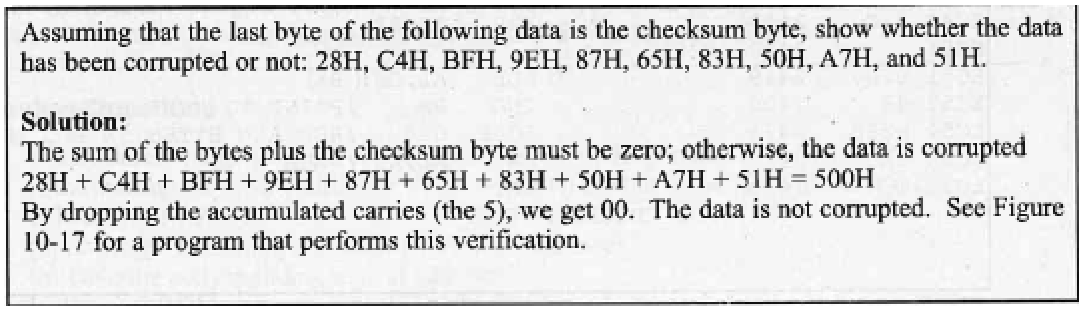
The sum is 118H, and dropping the carry, we get 18H. The checksum byte is the 2's complement of 18H, which is E8H.

(b) Adding the series of bytes including the checksum byte must result in zero. This indicates that all the bytes are unchanged and no byte is corrupted.

```
  25H
+ 62H
+ 3FH
+ 52H
+ E8H
2 00H  (dropping the carry)
```

(c) Adding the series of bytes including the checksum byte shows that the result is not zero, which indicates that one or more bytes have been corrupted.

```
  25H
+ 22H
+ 3FH
+ 52H
+ E8H
1 C0H   dropping the carry, we get C0H.
```



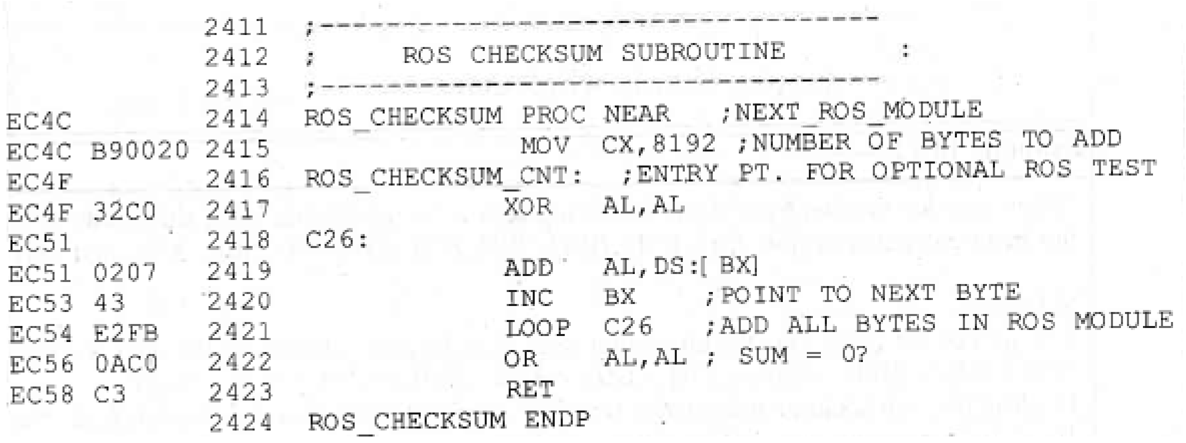
Assuming that the last byte of the following data is the checksum byte, show whether the data has been corrupted or not: 28H, C4H, BFH, 9EH, 87H, 65H, 83H, 50H, A7H, and 51H.

Solution:
The sum of the bytes plus the checksum byte must be zero; otherwise, the data is corrupted
28H + C4H + BFH + 9EH + 87H + 65H + 83H + 50H + A7H + 51H = 500H
By dropping the accumulated carries (the 5), we get 00. The data is not corrupted. See Figure 10-17 for a program that performs this verification.

Checksum Program:

- ✓ When the PC is turned on, one of the first things the BIOS does is to test the system ROM. The code for such a test is stored in the BIOS ROM.
- ✓ The following Figure shows the program using the checksum method.

- ✓ Notice in the code how all the bytes are added together without keeping the track of carries. Then, the total sum is ORed with itself to see if it is zero. The zero flag is expected to be set to high upon return from this subroutine. If it is not, the ROM is corrupted.



```
              2411  ;-----------------------------------
              2412  ;      ROS CHECKSUM SUBROUTINE       :
              2413  ;-----------------------------------
EC4C          2414  ROS_CHECKSUM PROC NEAR   ;NEXT_ROS_MODULE
EC4C B90020   2415              MOV  CX,8192 ;NUMBER OF BYTES TO ADD
EC4F          2416  ROS_CHECKSUM_CNT:   ;ENTRY PT. FOR OPTIONAL ROS TEST
EC4F 32C0     2417              XOR  AL,AL
EC51          2418  C26:
EC51 0207     2419              ADD  AL,DS:[ BX]
EC53 43       2420              INC  BX     ;POINT TO NEXT BYTE
EC54 E2FB     2421              LOOP C26    ;ADD ALL BYTES IN ROS MODULE
EC56 0AC0     2422              OR   AL,AL ; SUM = 0?
EC58 C3       2423              RET
              2424  ROS_CHECKSUM ENDP
```

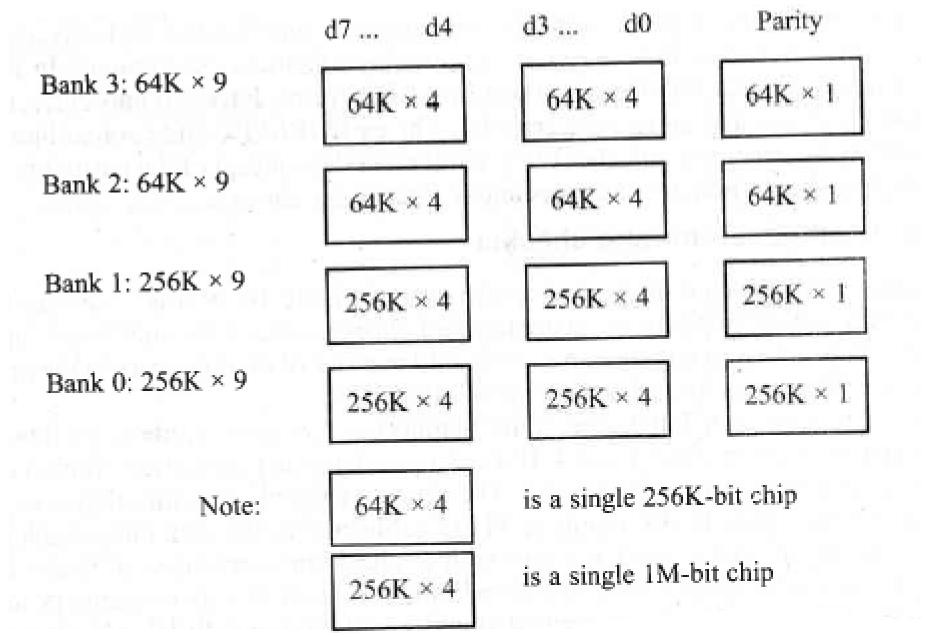
Fig: PC BIOS Checksum Routine

Use of Parity Bit in DRAM Error Detection:

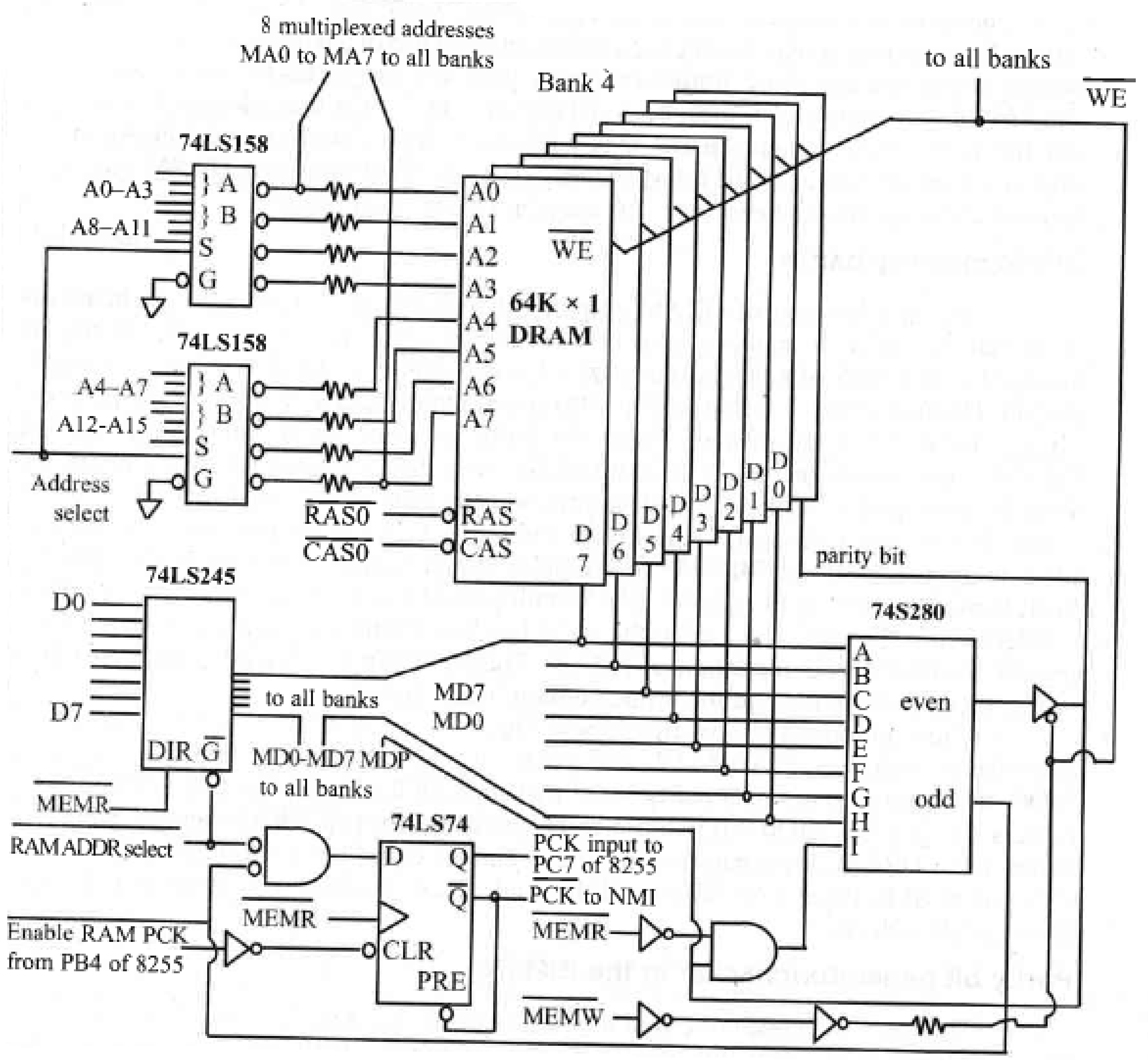
- o System boards or memory modules are populated with DRAM chips of various organizations, depending on the time they were designed and the availability of a given chip at a reasonable cost.
- o The memory technology is changing so fast that DRAM chips on the boards have a different look every year or two. While early PCs used 64K DRAMs, current PCs commonly use 1G chips.
- o To understand the use of a parity bit in detecting data storage errors, we use some simple examples from the early PCs to clarify some very important design concepts.

DRAM Memory Banks:

- ✓ The arrangement of DRAM chips on the system or memory module board is often referred to as a *memory bank*. For example, the 64K bytes of DRAM can be arranged as one bank of 8 IC chips of 64K x 1 organization, or 4 bank of 16K x 1 organization.
- ✓ The first IBM PC introduced in 1981, used memory chip of l6K x l organization.
- ✓ The following Figure shows the memory banks for 640K bytes of RAM using 256K and 1M DRAM chips.
- ✓ Notice the use of an extra bit for every byte of data to store the parity bit.
- ✓ With the extra parity bit every bank requires an extra chip of x 1 organization for parity check.



✓ The following Figure shows DRAM design and parity bit circuitry for a bank of DRAM.



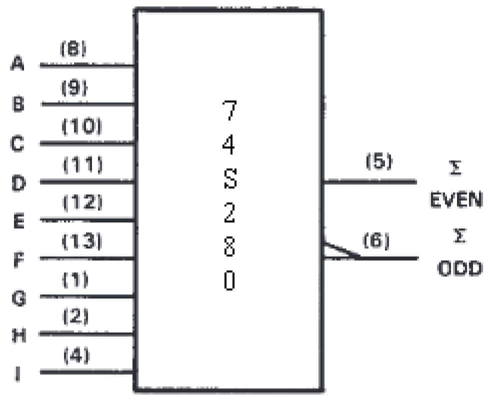
- ✓ First, note the use of the 74LS158 to multiplex the 16 address lines A0-A15, changing them to the 8 address lines of MA0-MA7 (multiplexed address) as required by the 64K x l DRAM chip.
- ✓ The resistors are for the serial bus line termination to prevent undershooting and overshooting at the inputs of DRAM. They range from 20 to 50 ohms, depending on the speed of the CPU and the printed circuit board layout.
- ✓ A few additional observations above Figure should be made. The output of multiplexer addresses MA0-MA7 will go to all the banks. Likewise, memory data MD0-MD7 and memory data parity MDP will go to all the banks.
- ✓ The 74LS245 not only buffers the data bus MD0-MD7 but also boosts it to drive all DRAM inputs. Since the banks of the DRAMs are connected in parallel and the capacitance loading is additive, the data line must be capable of driving all the loads.

Parity Bit Generator/Checker in IBM PC:

- o There are two types of errors that can occur in DRAM chips:
- o *Hard error* – some bits or an entire row of memory cell inside the memory chip get stuck to high or low permanently, thereafter always producing l or 0 regardless of what you write into the cell(s).
- o *Soft error* – a single bit is changed from 1 to 0 or from 0 to 1 due to current surge or certain kinds of particle radiation in the air. Parity is used to detect soft errors.
- o Including a parity bit to ensure data integrity in RAM is the most widely used method; since, it is the simplest and cheapest.
- o This method can only indicate if there is a difference between the data that was written to memory and the data that was read.
- o It cannot correct the error as is the case with some high-performance computers. In those computers and some of the x86-based servers, the EDC (error detection and correction) method is used to detect and correct the error bit.
- o The early IBM PC and compatibles use the 74S280 parity bit generator and checker to implement the concept of the parity bit.

74S280 Parity Bit Generator & Checker:

- ✓ The 74S280 chip has 9 inputs and 2 outputs. Depending on whether an even or odd number of ones appear in the input, the even or odd output is activated (according to following Table).
- ✓ As can be seen from Table, if all 9 inputs have an even number of 1 bits, the even output goes high (as in cases 1 and 4). If the 9 inputs have an odd number of high bits, the odd output goes high (as in cases 2 and 3).



Case	Inputs		Outputs	
	A – H	I	Even	ODD
1	Even	0	1	0
2	Even	1	0	1
3	Odd	0	0	1
4	Odd	1	1	0

The way the IBM PC uses this chip is as follows:

- ✓ Notice that in above Figure (DRAM design and parity bit circuitry for a bank of DRAM), inputs A – H are connected to the data bus, which is 8 bits, or one byte. The I input is used as a parity bit to check the correctness of the byte of data read from memory. When a byte of information is written to a given memory location in DRAM, the even-parity bit is generated and saved on the ninth DRAM chip as a parity bit with use of control signal $\overline{MEMW}$. This is done by activating the tri-state buffer using $\overline{MEMW}$. At this point, I of the 74S280 is equal to zero, since $\overline{MEMR}$ high.
- ✓ When a byte of data is read from the same location, the parity bit is gated into the I input of the 74S280 through $\overline{MEMR}$. This time the odd output is taken out and fed into a 74LS74. If there is a difference between the data written and the data read, the Q output (called PCK, parity bit check) of the 74LS74 is activated and Q activates NMI, indicating that there is a parity bit error, meaning that the data read is not the same as the data written. Consequently, it will display a parity bit error message.
- ✓ For example, if the byte of data written to a location has an even number of ls, A to H has an even number of ls, and I is zero, then the even-parity output of 74S280 becomes 1 and is saved on parity bit DRAM. This is case 1 shown in the above Table. If the same byte of data is read and there is an even number of ls (the byte is unchanged), I from the ninth bit DRAM, which is 1, is input to the 74S280, even becomes low, and odd becomes high, which is case 2 in the above Table. This high from the odd output will be inverted and fed to the 74LS74, making Q low. This means that $\bar{Q}$ is high thereby indicating that the written byte is the same as the byte read and there is no errors occurred.
- ✓ If the number of 1s in the byte has changed from even to odd and the 1 from the saved parity DRAM makes the number of inputs even (case 4 above), the odd output becomes low, which is inverted and passed to the 74LS74 D flip-flop. This makes Q = 1 and $\bar{Q}$ = 0, which signals the NMI to display a parity bit error message on the screen.

16-BIT MEMORY INTERFACING:

In this section, memory interfacing for 16-bit CPUs will be discussed. 80286 is taken as an example, but the concepts can apply to any 16-bit microprocessor.

ODD & EVEN Banks:

In a 16-bit CPU such as the 80286, memory locations 00000-FFFFF are designated as odd and even bytes as shown in the following Fig. This Figure shows only 1M byte of memory; the concept of odd and even banks applies to the entire memory space of a given processor with a 16-bit data bus.

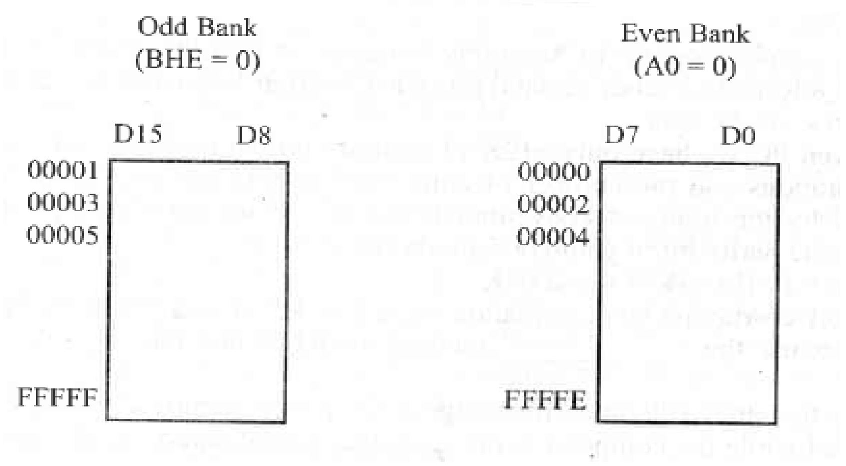


Fig: ODD & EVEN Banks of Memory

To distinguish between odd and even bytes, the CPU provides a signal called BHE (bus high enable). BHE in association with A0 is used to select the odd or even byte according to following Table.

BHE	A0	Memory Selection	
0	0	Even Word	D0 – D15
0	1	Odd Byte	D8 – D15
1	0	Even Byte	D0 – D7
1	1	None	-

The following Figure shows 640KB of DRAM for 16-bit buses.

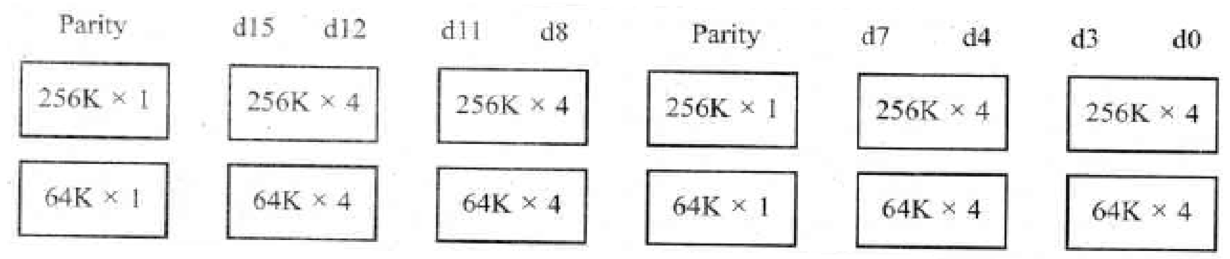


Fig: 640K Bytes of DRAM with ODD & EVEN Banks Designation

The following Figure shows the use of A0 and BHE as bank selectors. Here, the 74LS245 chip is used as a data bus buffer.

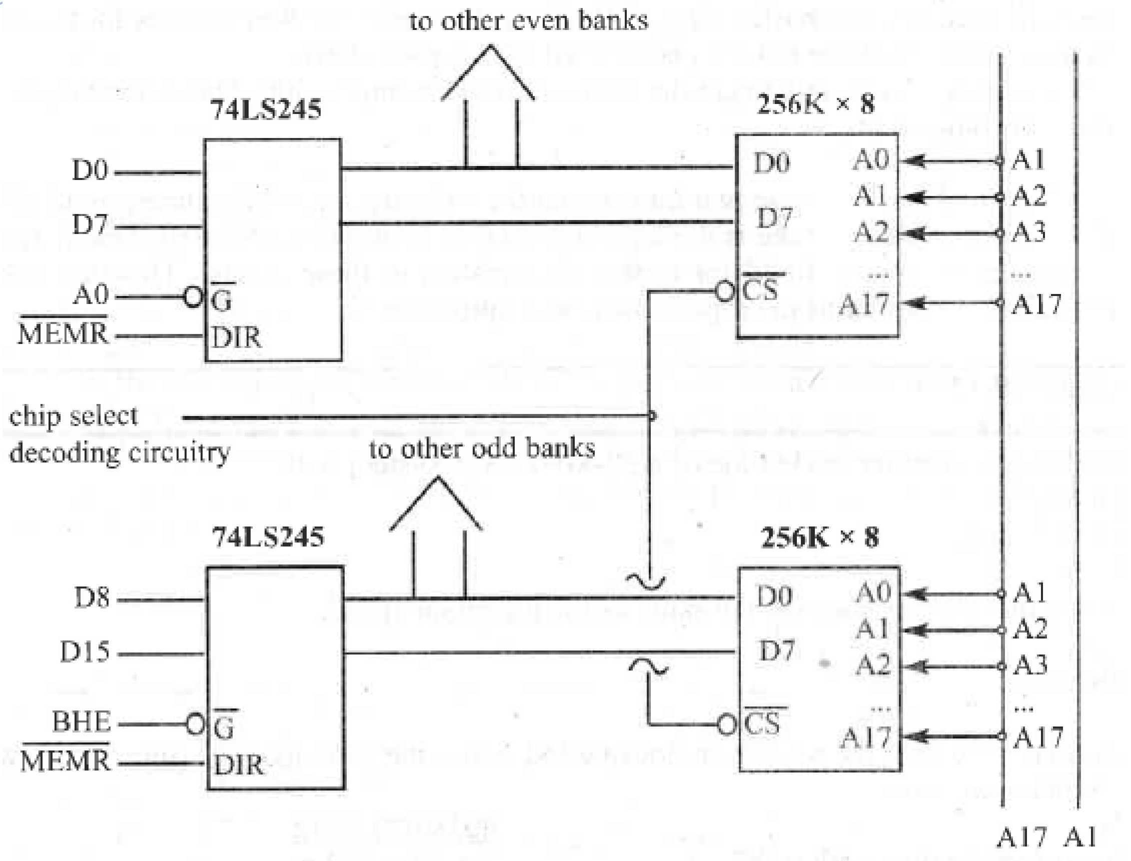
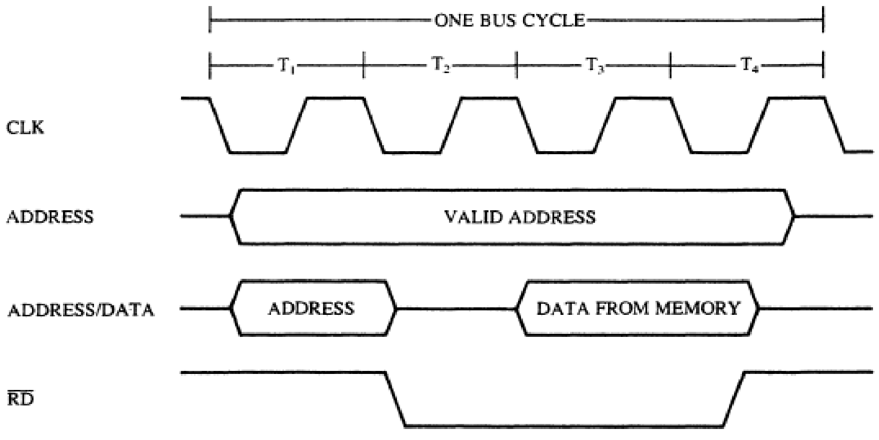


Fig: 16-bit Data Connection in the Systems with 16-bit Data Bus

Memory Cycle Time and Inserting Wait States:

- To access an external device such as memory or I/O, the CPU provides a fixed amount of time called a bus cycle time. During this bus cycle time, the read and write operation of memory or I/O must be completed.
- The bus cycle time used for accessing memory is often referred to as MC (memory cycle) time. The time from when the CPU provides the addresses at its address pins to when the data is expected at its data pins is called memory read cycle time.
- The processors such as the 8088/86, the memory cycle time takes 4 clocks, and from 286 to Pentium, the memory cycle time is only 2 clocks.
- If memory is slow and its access time does not match the MC time of the CPU, extra time can be requested from the CPU to extend the read cycle time. This extra time is called a wait state (WS).



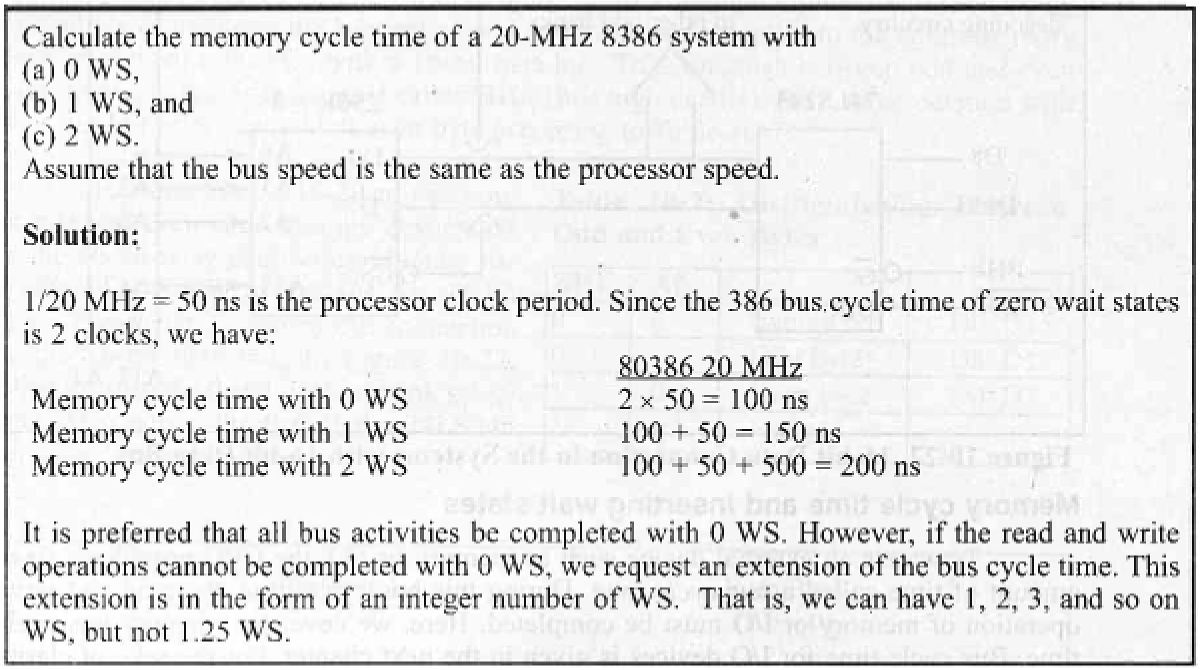
Simplified 8086/8088 read bus cycle

» It must be noted that, memory access time is not the only factor in slowing down the CPU. The other factor is the delay associated with signals going through the data and address path.

» Delay associated with reading data stored in memory has the following two components:

1. The time taken for address signals to go from CPU pins to memory pins, (going through decoders and buffers (e.g., 74LS245)); plus the time taken for the data to travel from memory to CPU, is referred to as a *path delay*.
2. The *memory access time* to get the data out of the memory chip. This is the larger (80% of the read cycle time) of the two components.

» The total sum of these two (path delay + memory access time) must equal the memory read cycle time provided by the CPU.



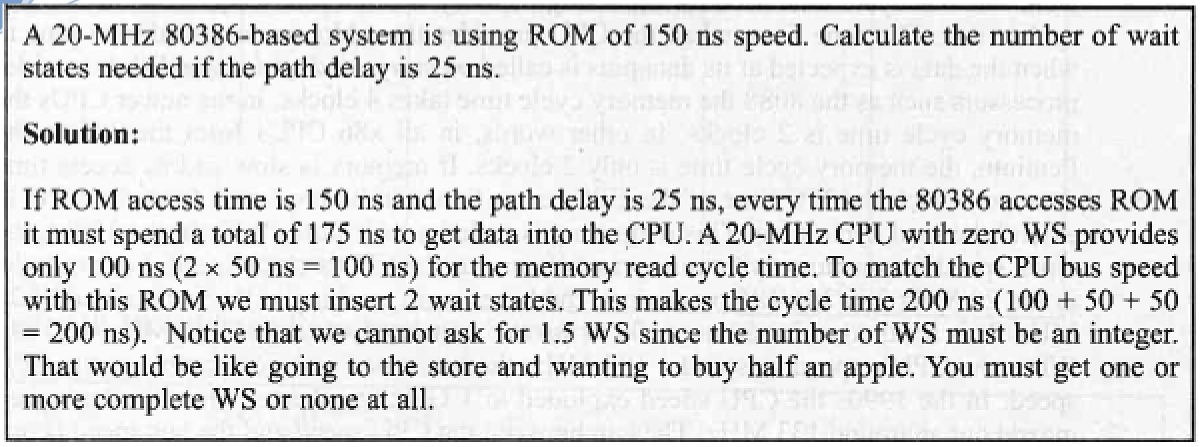
Calculate the memory cycle time of a 20-MHz 8386 system with
(a) 0 WS,
(b) 1 WS, and
(c) 2 WS.
Assume that the bus speed is the same as the processor speed.

Solution:

1/20 MHz = 50 ns is the processor clock period. Since the 386 bus cycle time of zero wait states is 2 clocks, we have:

	<u>80386 20 MHz</u>
Memory cycle time with 0 WS	2 × 50 = 100 ns
Memory cycle time with 1 WS	100 + 50 = 150 ns
Memory cycle time with 2 WS	100 + 50 + 500 = 200 ns

It is preferred that all bus activities be completed with 0 WS. However, if the read and write operations cannot be completed with 0 WS, we request an extension of the bus cycle time. This extension is in the form of an integer number of WS. That is, we can have 1, 2, 3, and so on WS, but not 1.25 WS.



A 20-MHz 80386-based system is using ROM of 150 ns speed. Calculate the number of wait states needed if the path delay is 25 ns.

Solution:

If ROM access time is 150 ns and the path delay is 25 ns, every time the 80386 accesses ROM it must spend a total of 175 ns to get data into the CPU. A 20-MHz CPU with zero WS provides only 100 ns (2 × 50 ns = 100 ns) for the memory read cycle time. To match the CPU bus speed with this ROM we must insert 2 wait states. This makes the cycle time 200 ns (100 + 50 + 50 = 200 ns). Notice that we cannot ask for 1.5 WS since the number of WS must be an integer. That would be like going to the store and wanting to buy half an apple. You must get one or more complete WS or none at all.

Accessing EVEN & ODD Words:

- Intel defines 16-bit data as a word. The address of a word can start at an even or an odd number.
- For example, in the instruction "*MOV AX, [2000]*" the address of the word being fetched into AX starts at an even address. In the case of "*MOV AX, [2007]*" the address starts at an odd address.
- In systems with a 16-bit data bus, accessing a word from an odd addressed location can be slower.
- As shown in the following Fig, in the 8-bit system, accessing a word is treated like accessing two bytes regardless of whether the address is odd or even. Since accessing a byte takes one memory cycle, accessing any word will take 2 memory cycles.

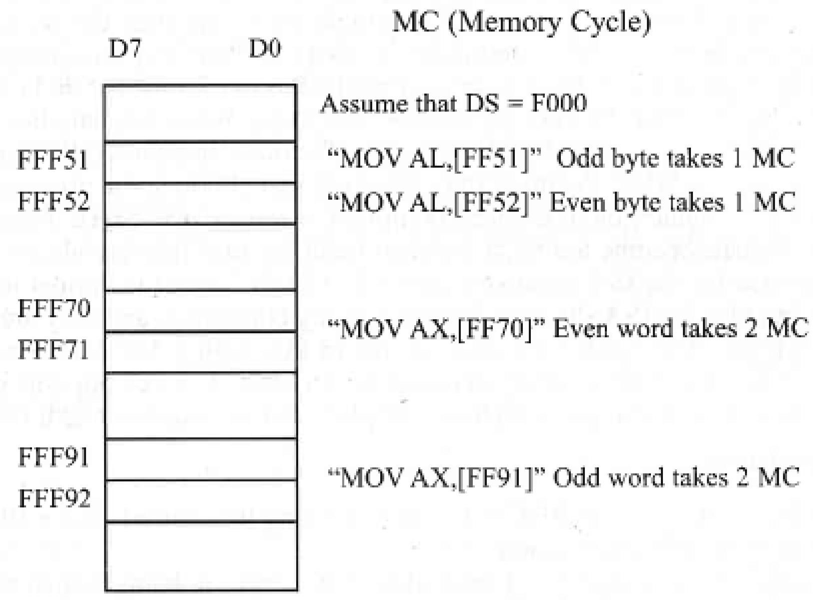

MC (Memory Cycle)
D7
D0
Assume that DS = F000
FFF51
FFF52
"MOV AL,[FF51]" Odd byte takes 1 MC
"MOV AL,[FF52]" Even byte takes 1 MC
FFF70
FFF71
"MOV AX,[FF70]" Even word takes 2 MC
FFF91
FFF92
"MOV AX,[FF91]" Odd word takes 2 MC

Fig: Accessing EVEN & ODD Words in 8-bit CPU

- In the 16- bit system, accessing a word with an even address takes one memory cycle. That is because; one byte is carried on D0-D7 and the other on D8-Dl5 in the same memory cycle.

- But, accessing a word with an odd address requires two memory cycles. For example, see how accessing the word in the instruction "*MOV AX, [F617]*" works as shown in following Fig.

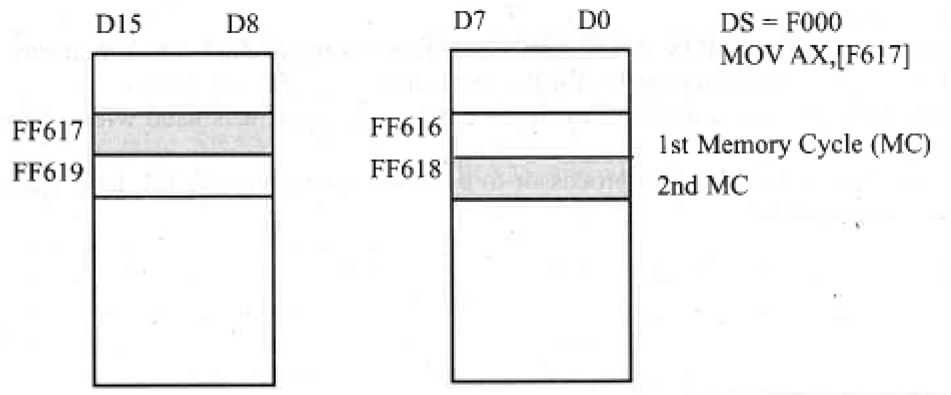


Fig: Accessing an Odd-Addressed Word in 16-bit Processor

- Assuming that DS = F000H in this instruction, the contents of physical memory locations FF6 l7H and FF6l8H are being moved into AX.
- In the first cycle, the 286 CPU accesses location FF617H and puts it in AL.
- In the second cycle, the contents of memory location FF618H are accessed and put into AH.
- Hence, it will be wise to put any words on an even address if the program is going to be run on a 16-bit system.
- A pseudo-instruction is specifically designed for this purpose. It is the EVEN directive and is used as follows:

```
                EVEN
VALUE1          DW      ?
```

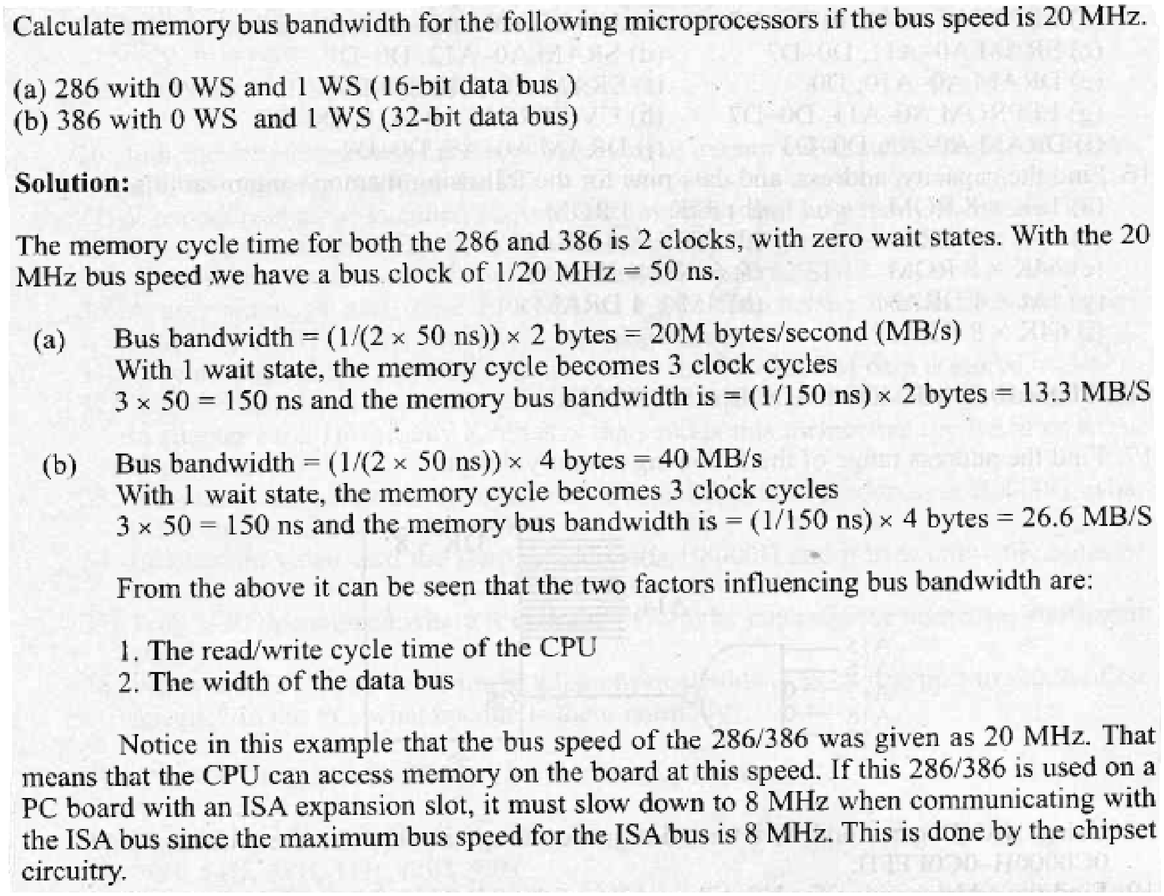
- This directive ensures that, the VALUE1, a word-sized operand, is located in an even address location. Hence, an instruction such as "*MOV AX, VALUE1*" will take only a single memory cycle.

Bus Bandwidth:

- The main advantage of the 16-bit data bus is; doubling of the rate of transfer of information between the CPU and the outside world. The rate of data transfer is generally called *bus bandwidth*. In other words, *bus bandwidth* is a measure of how fast buses transfer information between the CPU and memory or peripherals. The wider the data bus, the higher the bus bandwidth.
- But, the advantage of the wider external data bus comes at the cost of increasing the size of the printed circuit board. Bus bandwidth is measured in MB (megabytes) per second and is calculated as follows:

bus bandwidth = (1/bus cycle time) x bus width in bytes

- In the above formula, bus cycle time can be either memory or I/O cycle time.



Calculate memory bus bandwidth for the following microprocessors if the bus speed is 20 MHz.

(a) 286 with 0 WS and 1 WS (16-bit data bus)
(b) 386 with 0 WS and 1 WS (32-bit data bus)

Solution:

The memory cycle time for both the 286 and 386 is 2 clocks, with zero wait states. With the 20 MHz bus speed we have a bus clock of 1/20 MHz = 50 ns.

(a) Bus bandwidth = (1/(2 × 50 ns)) × 2 bytes = 20M bytes/second (MB/s)
With 1 wait state, the memory cycle becomes 3 clock cycles
3 × 50 = 150 ns and the memory bus bandwidth is = (1/150 ns) × 2 bytes = 13.3 MB/S

(b) Bus bandwidth = (1/(2 × 50 ns)) × 4 bytes = 40 MB/s
With 1 wait state, the memory cycle becomes 3 clock cycles
3 × 50 = 150 ns and the memory bus bandwidth is = (1/150 ns) × 4 bytes = 26.6 MB/S

From the above it can be seen that the two factors influencing bus bandwidth are:

1. The read/write cycle time of the CPU
2. The width of the data bus

Notice in this example that the bus speed of the 286/386 was given as 20 MHz. That means that the CPU can access memory on the board at this speed. If this 286/386 is used on a PC board with an ISA expansion slot, it must slow down to 8 MHz when communicating with the ISA bus since the maximum bus speed for the ISA bus is 8 MHz. This is done by the chipset circuitry.

- There are two ways to increase the bus bandwidth:
  - ✓ Use a wider data bus.
  - ✓ Shorten the bus cycle time.
- While the data bus width has increased from 16-bit in the 80286 to 64-bit in the Pentium, the bus cycle time is reaching a maximum of 133 MHz.

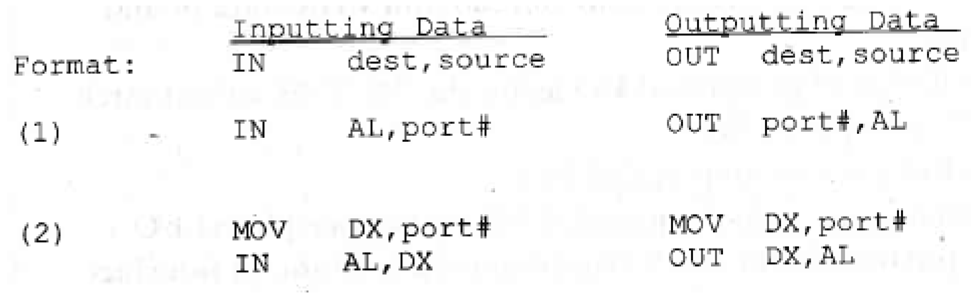
8255 I/O PROGRAMMING

8088 INPUT/OUTPUT INSTRUCTIONS:

- All x86 microprocessors, from the 8088 to the Pentium, can access external devices called ports. This is done using I/O instructions.
- The x86 CPU has I/O space in addition to memory space. While memory can contain Opcode and data, I/O ports contain data only.
- There are two instructions for this purpose: OUT and IN. These instructions can send data from the accumulator (AL or AX) to ports or bring data from ports into the accumulator.
- In accessing ports, we can use an 8-bit or 16-bit data port.

8-bit Data Ports:

- The 8-bit I/O operation of the 8088 is applicable to all x86 CPUs from the 8088 to the Pentium.
- The 8-bit port uses the D0-D7 data bus to communicate with I/O devices.
- In 8-bit port programming, register AL is used as the source of data, when using the OUT instruction; and as the destination, for the IN instruction. This means that to input or output data from any other registers, the data must first be moved to the AL register.
- Instructions OUT and IN have the following formats:



	Inputting Data		Outputting Data	
Format:	IN	dest,source	OUT	dest,source
(1)	IN	AL,port#	OUT	port#,AL
(2)	MOV IN	DX,port# AL,DX	MOV OUT	DX,port# DX,AL

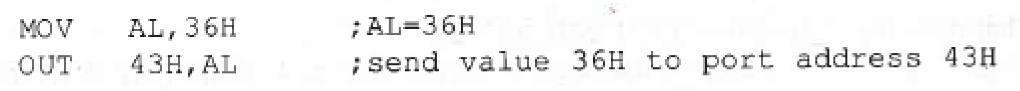
In format (l) –

- ✓ port# is the address of the port and can be from 00 to FFH, allowing up to 256 input and 256 output ports.
- ✓ In this format, the 8-bit port address is carried on address bus A0-A7.
- ✓ No segment register is involved in computing the address.

In format (2) –

- ✓ port# is the address of the port and can be from 0000 to FFFFH, allowing up to 65,536 input and 65,536 output ports.
- ✓ In this format, the 16- bit port address is carried on the address bus A0-A15.
- ✓ The use of a register as a pointer for the port address has an advantage in that the port address can be changed very easily, especially in. cases of dynamic compilations where the port address can be passed to DX.

- » I/O instructions are widely used in programming peripheral devices such as printers, hard disks, and keyboards.
- » The port address can be either 8-bit or 16-bit. For an 8-bit port address, we can use the immediate addressing mode.
- » The following program sends a byte of data to a fixed port address of 43H:



```
MOV    AL,36H         ;AL=36H
OUT    43H,AL         ;send value 36H to port address 43H
```

- » The 8-bit address used in immediate addressing mode limits the number of ports to 256 for input plus 256 for output. To have a larger number of ports we must use the 16-bit port address instruction.
- » To use the 16-bit port address, *register indirect addressing mode* must be used. The register used for this purpose is DX.
- » The following program sends values 55H and AAH to I/O port address 300H (a 16-bit port address).

```
BACK: MOV    DX,300H        ;DX = port address 300H
      MOV    AL,55H
      OUT    DX,AL          ;toggle the bits
      MOV    AL,0AAH
      OUT    DX,AL          ;toggle the bits
      JMP    BACK
```

- » We can only use register DX for 16-bit I/O addresses; no other register can be used for this purpose. Also, notice the use of register AL for 8-bit data:

```
MOV DX,378H             ;DX=378 the port address
MOV AL,BL               ;load data into accumulator
OUT DX,AL               ;write contents of AL to port
                        ;whose address is in DX
```

- » Just like the OUT instruction, the IN instruction uses the DX register to hold the address and AL to hold the arrived 8-bit data. In other words, DX holds the 16-bit port address while AL receives the 8-bit data brought in from an external port.
- » The following program gets data from port address 300H and sends it to port address 302H.

```
MOV    DX,300H      ;load port address
IN     AL,DX        ;bring in data
MOV    DX,302H
OUT    DX,AL        ;send it out
```

In a given 8088-based system, port address 22H is an input port for monitoring the temperature. Write Assembly language instructions to monitor that port continuously for the temperature of 100 degrees. If it reaches 100, then BH should contain 'Y'.

Solution:

```
BACK:       IN     AL,22H   ;get the temperature from port # 22H
            CMP    AL,100       ;is temp = 100?
            JNZ    BACK         ;if not, keep monitoring
            MOV    BH,'Y        ;temp = 100, load 'Y' into BH
```

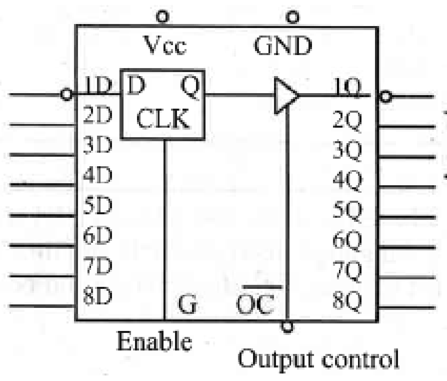
I/O ADDRESS DECODING & DESIGN:

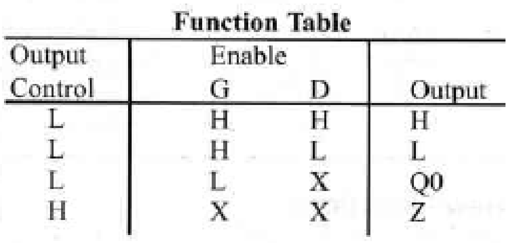
The decoding of I/O ports is done by using TTL logic gates 74LS373 and 74LS244. The following are the steps:

1. The control signals IOR and IOW are used along with the decoders.
2. For an 8-bit port address, A0-A7 is decoded.
3. If the port address is 16-bit (using DX), A0-A15 is decoded.

Using 74LA373 in an Output Port Design:

- In every computer, whenever data is sent out by the CPU via the data bus, the data must be latched by the receiving device. While memories have an internal latch to grab the data, a latching system must be designed for simple I/O ports.
- The 74LS373 can be used for this purpose. Notice in the following Fig. that in order to make the 74LS373 work as a latch, the OC pin must be grounded.





Function Table

Output Control	Enable G	D	Output
L	H	H	H
L	H	L	L
L	L	X	Q0
H	X	X	Z

Fig: 74LS373 D Latch

- For an output latch, it is common to AND the output of the address decoder with the control signal IOW to provide the latching action as shown in Figure.

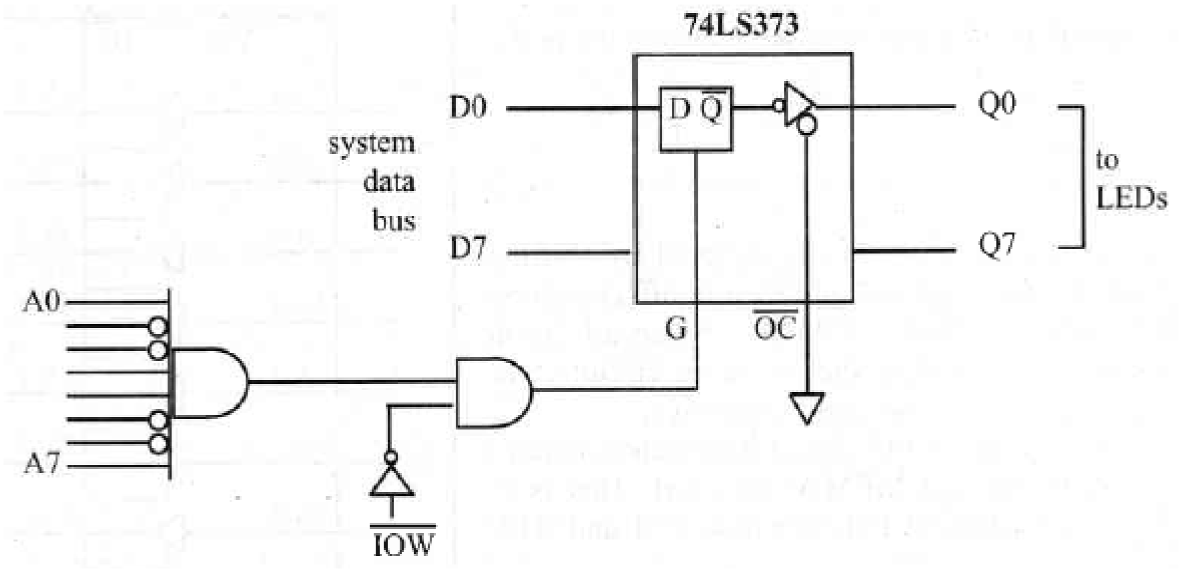


Fig: Design for *"OUT 99H, AL"*

Show the design of an output port with an I/O address of 31FH using the 74LS373.

Solution:

31F9H is decoded, then ANDed with IOW to activate the G pin of the 74LS373 latch. This is shown in Figure below.

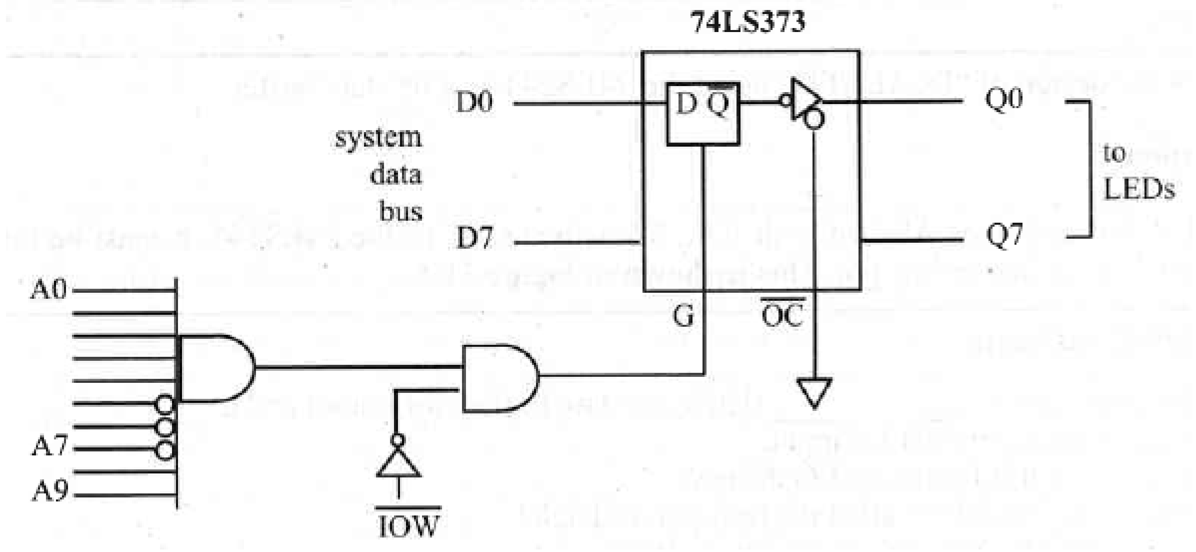


Fig: Design for Output Port Address of 31FH

IN Port Design Using the 74LA244:

- When the data is coming in by way of a data bus, it must come in through a three-state buffer. This is referred to as *tri-stated*. See the following Fig for the internal circuitry of 74LS244.

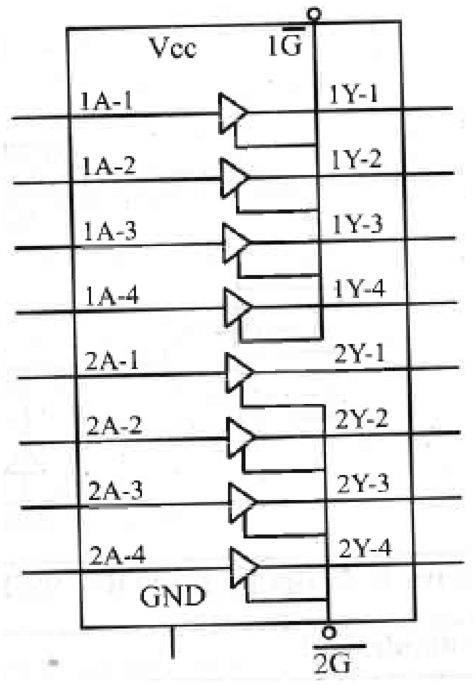


Fig: 74LS244 Octal Buffer

- Here, since 1G and 2G each control only 4 bits of 74LS244, both must be activated for 8 bits input. The following Fig shows the use of 74LS244 as an entry port to the system data bus. In the following Figures, the address decoder and IOR control signal together activate the tri-state input.

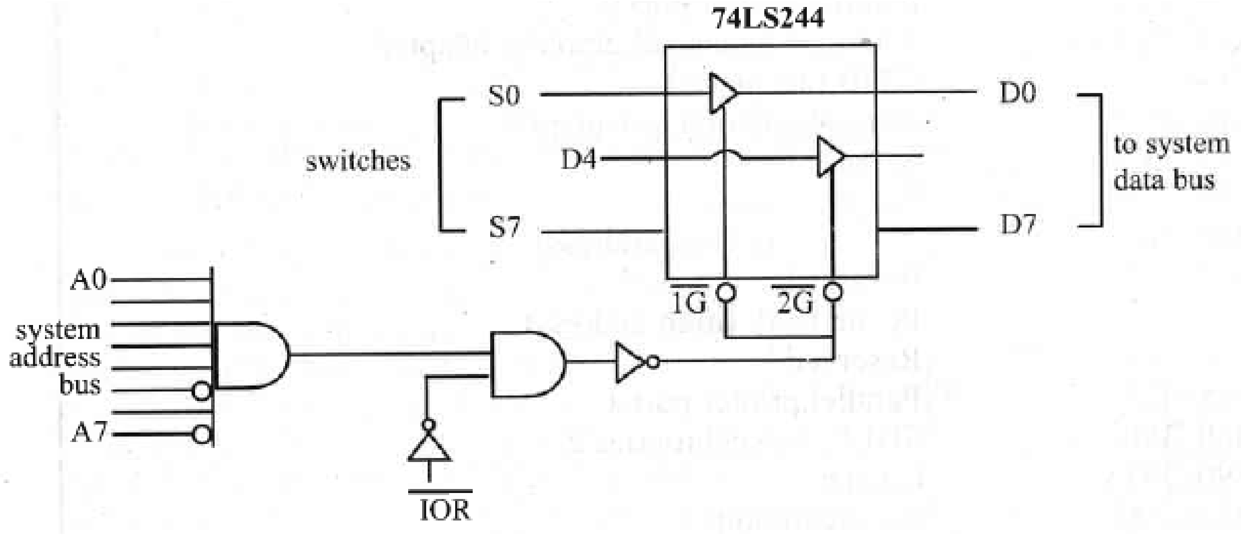


Fig: Input Port Design for *"IN AL, 5FH"*

Show the design of "IN AL,9FH" using the 74LS244 as a tri-state buffer.

Solution:

9FH is decoded, then ANDed with IOR. To activate OC of the 74LS244, it must be inverted since OC is an active-low pin. This is shown in Figure below.

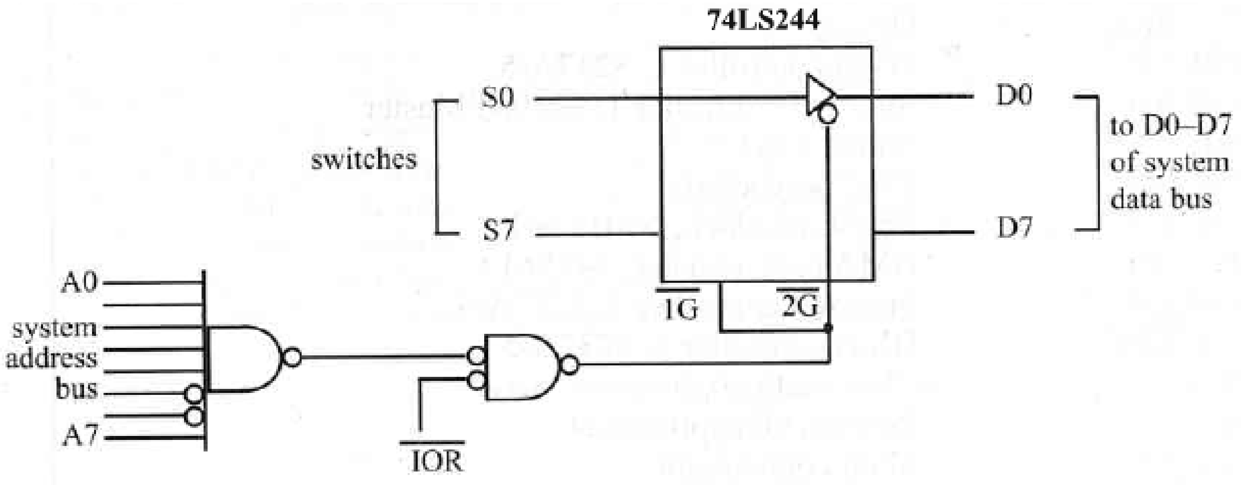


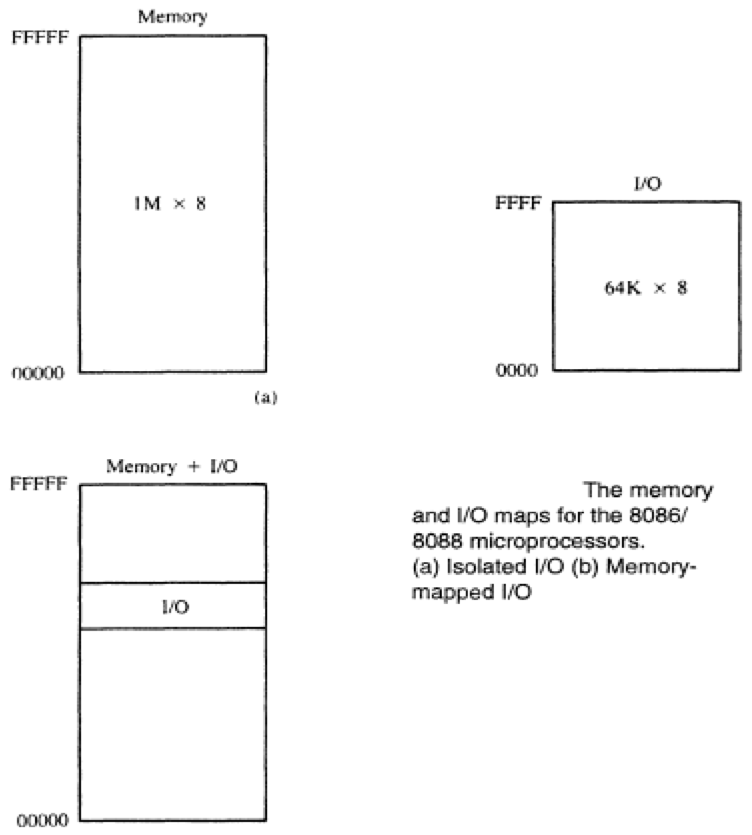
Fig: Design for *"IN AL, 9FH"*

Memory-Mapped I/O:

- Communicating with the I/O devices using IN and OUT instructions is referred to as *peripheral I/O*. Some designers also refer to it as *isolated I/O*.

» Some new RISC processors do not have IN and OUT instructions; they use *memory-mapped I/O*.

» In memory-mapped I/O, a memory location is assigned to be an input and output port.



The memory and I/O maps for the 8086/8088 microprocessors. (a) Isolated I/O (b) Memory-mapped I/O

» The following are the differences between peripheral I/O and memory-mapped I/O in x86 PC:

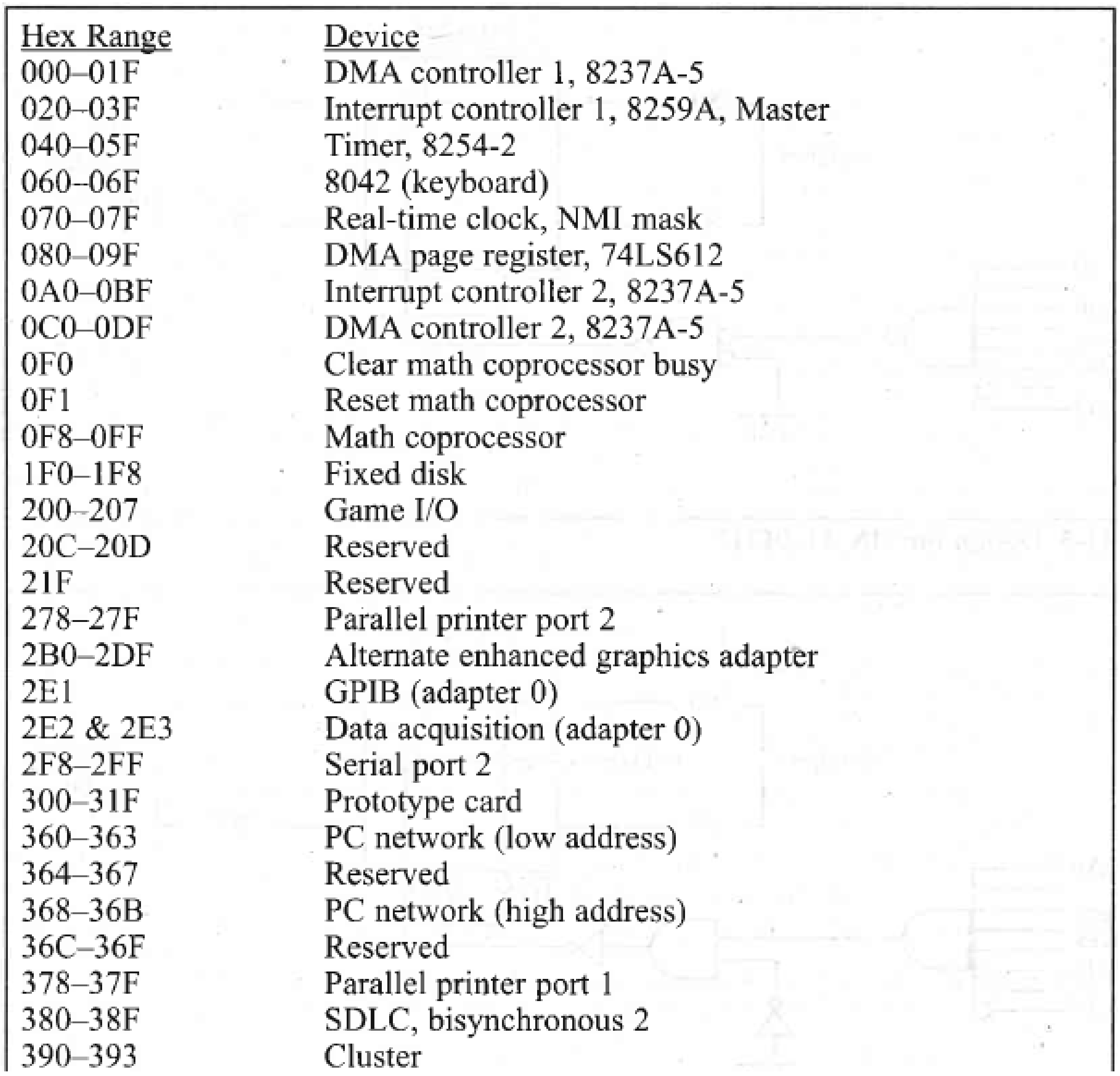
Isolated (Peripheral) I/O	**Memory-Mapped I/O**
1. The IN and OUT instructions transfer data between the microprocessors accumulator or memory and the I/O device.	1. Instructions that access memory locations are used instead of IN and OUT instructions: *MOV AL, [2000]* will access the input port & *MOV [2000], AL* will access the output port.
2. Only A0-A15 are decoded; Hence, DS initialization is not required; decoding circuitry may be less expensive.	2. Entire 20-bit address, A0-A19, must be decoded (decoding circuitry is expensive); Hence DS must be loaded before accessing memory-mapped I/O: `MOV AX,3000H ;load the segment value` `MOV DS,AX` `MOV AL,[5000] ;get a byte from loc. 35000H`
3. IOR and IOW control signals are used.	3. MEMR and MEMW control signals are used.
4. Limited only to 65,536 input ports	4. The number of ports can be as high as 2^{20} (1,048,576).

and 65,536 output ports.	
5. Data should be moved to accumulator for any kind of operations.	5. Arithmetic and logic operations can be performed directly, without moving data to accumulator.
6. The user can expand the memory to its full size without using any memory space for I/O devices.	6. Uses memory address space, which could lead to memory space fragmentation.

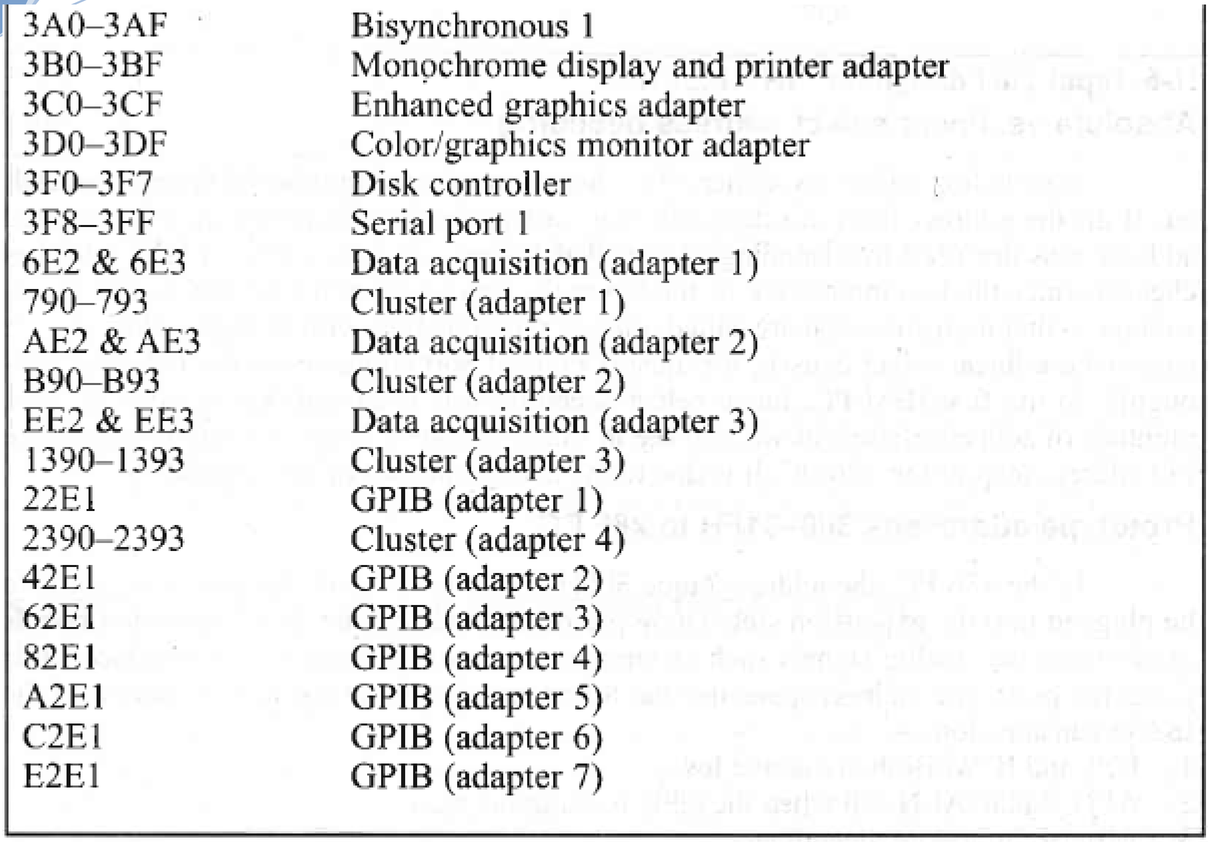
I/O ADDRESS MAP OF x86 PCs:

Any system that needs to be compatible with the x86 IBM PC must follow the I/O map of the following Table:

Table: I/O Map for x86 PC



Hex Range	Device
000–01F	DMA controller 1, 8237A-5
020–03F	Interrupt controller 1, 8259A, Master
040–05F	Timer, 8254-2
060–06F	8042 (keyboard)
070–07F	Real-time clock, NMI mask
080–09F	DMA page register, 74LS612
0A0–0BF	Interrupt controller 2, 8237A-5
0C0–0DF	DMA controller 2, 8237A-5
0F0	Clear math coprocessor busy
0F1	Reset math coprocessor
0F8–0FF	Math coprocessor
1F0–1F8	Fixed disk
200–207	Game I/O
20C–20D	Reserved
21F	Reserved
278–27F	Parallel printer port 2
2B0–2DF	Alternate enhanced graphics adapter
2E1	GPIB (adapter 0)
2E2 & 2E3	Data acquisition (adapter 0)
2F8–2FF	Serial port 2
300–31F	Prototype card
360–363	PC network (low address)
364–367	Reserved
368–36B	PC network (high address)
36C–36F	Reserved
378–37F	Parallel printer port 1
380–38F	SDLC, bisynchronous 2
390–393	Cluster



3A0–3AF	Bisynchronous 1
3B0–3BF	Monochrome display and printer adapter
3C0–3CF	Enhanced graphics adapter
3D0–3DF	Color/graphics monitor adapter
3F0–3F7	Disk controller
3F8–3FF	Serial port 1
6E2 & 6E3	Data acquisition (adapter 1)
790–793	Cluster (adapter 1)
AE2 & AE3	Data acquisition (adapter 2)
B90–B93	Cluster (adapter 2)
EE2 & EE3	Data acquisition (adapter 3)
1390–1393	Cluster (adapter 3)
22E1	GPIB (adapter 1)
2390–2393	Cluster (adapter 4)
42E1	GPIB (adapter 2)
62E1	GPIB (adapter 3)
82E1	GPIB (adapter 4)
A2E1	GPIB (adapter 5)
C2E1	GPIB (adapter 6)
E2E1	GPIB (adapter 7)

Absolute vs. Linear Select Address Decoding:

- In decoding addresses, either all the address lines or a selected number of them are decoded.
    - If all the address lines are decoded, it is called *absolute decoding*.
    - If only selected address pins are used for decoding, it is called *linear select decoding* – This is cheaper due to the less number of input and the fewer the gates needed for decoding. The disadvantage is that it creates what are called *aliases*, the same port with multiple addresses. Hence, port address documentation is necessary.

Portable Addresses 300 – 31FH in x86 PC:

In the x86 PC, the address range 300H – 31FH is set aside for prototype cards to be plugged into the expansion slot. These prototype cards can be data acquisition boards used to monitor analog signals such as temperature, pressure, and so on. Interface cards using the prototype address space use the following signals on the 62-pin section of the ISA expansion slot:

1. IOR and IOW. Both are active low.
2. AEN signal: AEN = 0 when the CPU is using the bus.
3. A0-A9 for address decoding.

Use of Simple Logic Gates as Address Decoders:

The following Fig shows the circuit design for a 74LS373 latch connected to port address 300H of an x86 PC via an ISA expansion slot. Notice the use of signals A0-A9 and AEN. AEN is low when the x86 microprocessor is in control of the buses. Here, we are using simple logic gates such as NAND and inverter gates for the I/O address decoder. These can be replaced with the 74LS138 chip because the 74LS138 is a group of NAND gates in a single chip.

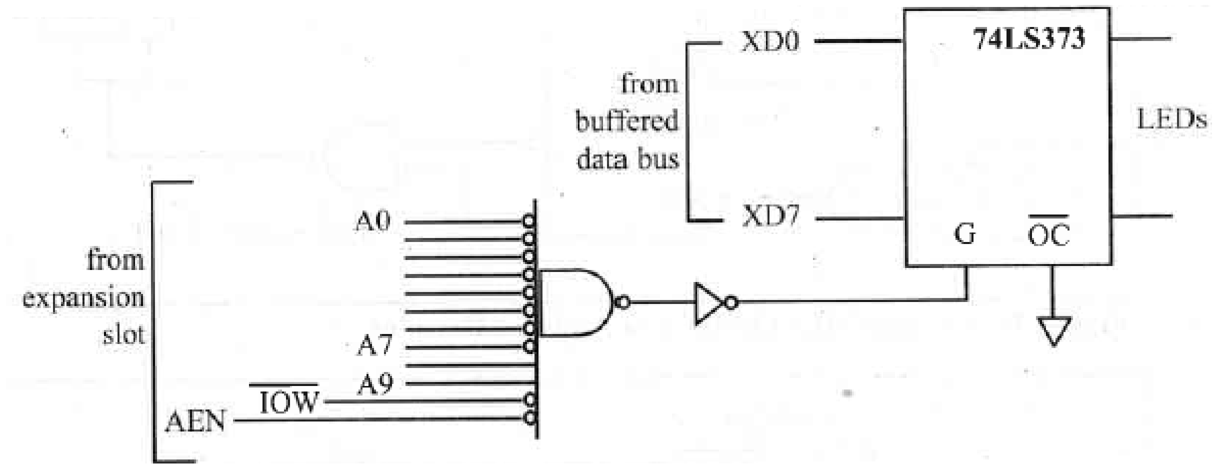
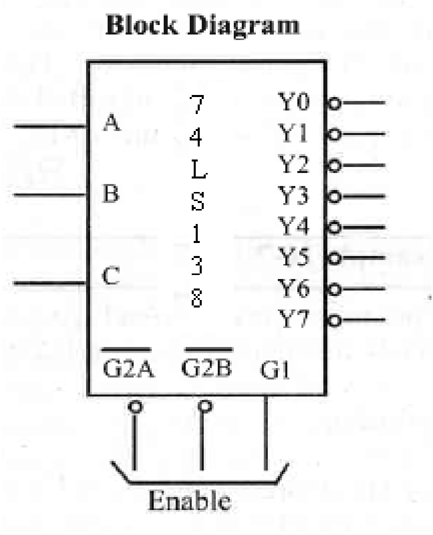
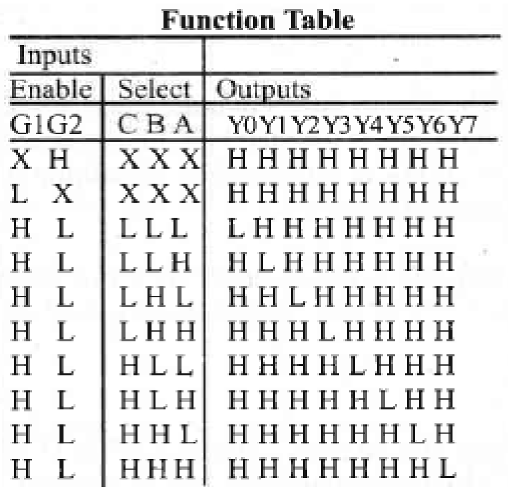


Fig: Using Simple Logic Gates for I/O Address Decoder (I/O Address 300H)

Use of 74LS138 as Decoder:

The following Fig shows the 74LS138.





Function Table

Inputs		
Enable	Select	Outputs
G1G2	C B A	Y0Y1Y2Y3Y4Y5Y6Y7
X H	X X X	H H H H H H H H
L X	X X X	H H H H H H H H
H L	L L L	L H H H H H H H
H L	L L H	H L H H H H H H
H L	L H L	H H L H H H H H
H L	L H H	H H H L H H H H
H L	H L L	H H H H L H H H
H L	H L H	H H H H H L H H
H L	H H L	H H H H H H L H
H L	H H H	H H H H H H H L

The following Fig is an example of the use of a 74LA138 for an I/O address decoder.

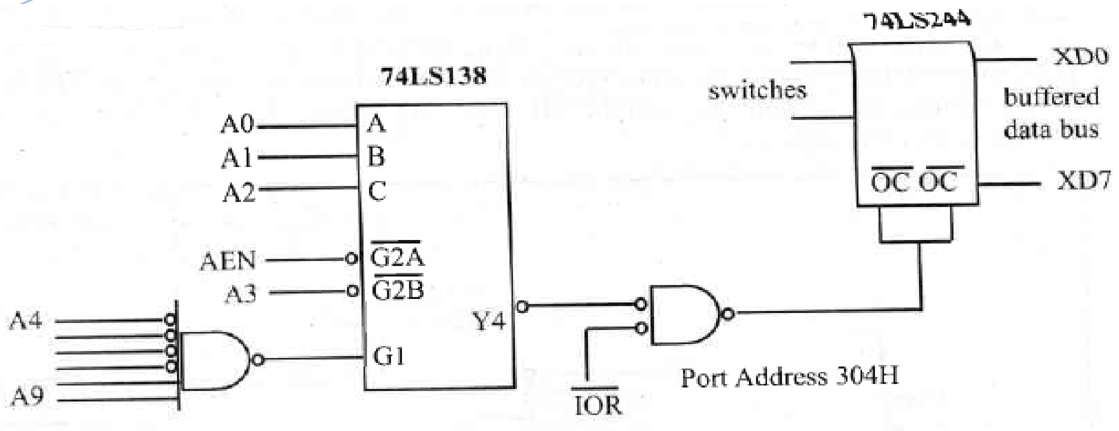


Fig: Using 74LS138 for I/O Address Decoding

- ✓ This is an address decoding for an input port located at address 304H.
- ✓ The Y4 output, together with the IOR signal, controls the 74LS244 input buffer.
- ✓ Note that, each Y output can control a single I/O device.

IBM PC I/O Address Decoder:

The following Fig shows a 74LS138 chip used as an I/O address decoder in the original IBM PC.

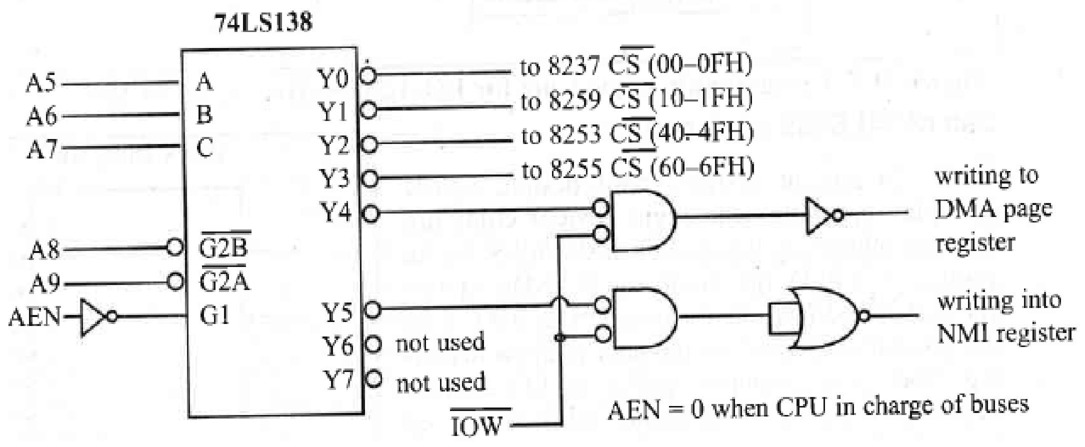


Fig: Port Address Decoding in the Original IBM PC

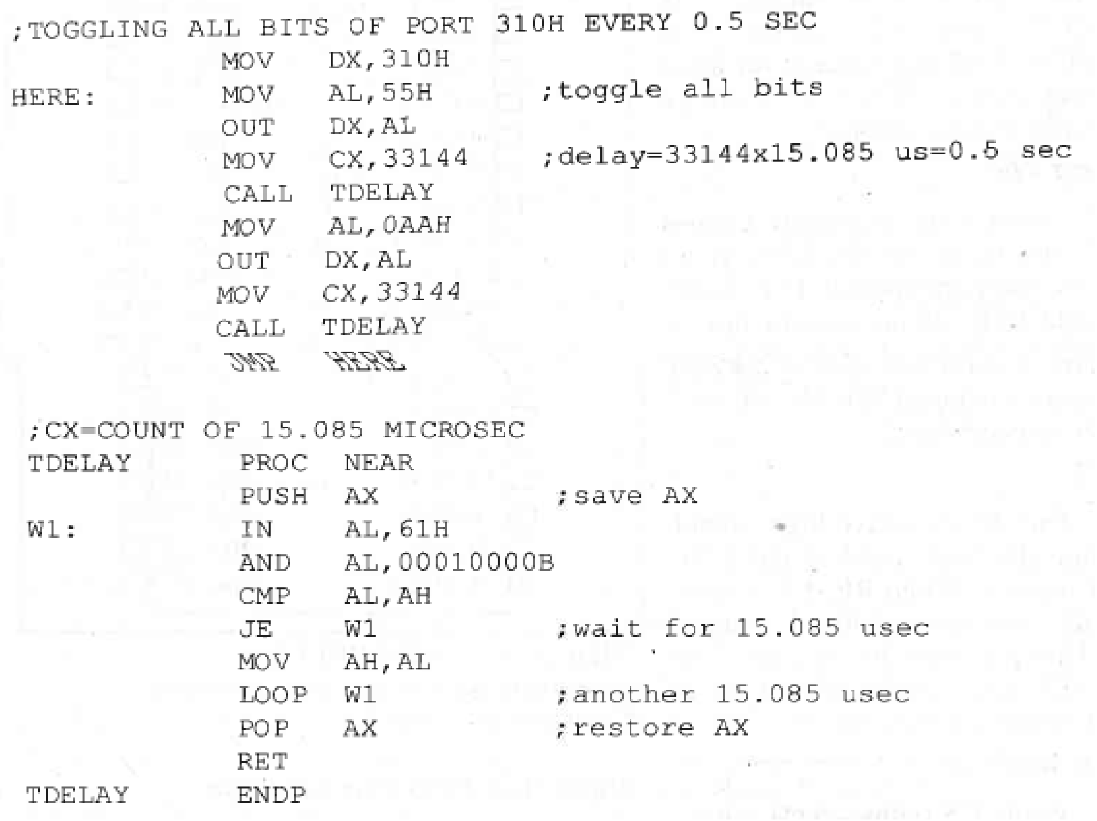
- ✓ Notice that, while A0 to A4 go to individual peripheral input addresses, A5, A6, and A7 are responsible for the selection of outputs Y0 to Y7.
- ✓ In order to enable the 74LS138, pins A8, A9, and AEN all must be low. While A8 and A9 will directly affect the port address calculations, AEN is low only when the x86 is in control of the system bus (see the following Table).

Table: Port Address Decoding Table on the Original PC

G1	G2A	G2B	C B A	
AEN	A9	A8	A7 A6 A5 A4 A3 A2 A1 A0	
0	0	0	0 0 0 0 0 0 0 0	00 Lowest port address
0	0	0	1 1 1 1 1 1 1 1	FF Highest port address

Port 61H and Time Delay Generation:

- In order to maintain compatibility with the IBM PC and run operating systems such as MS-DOS and Windows, the assignment of I/O port addresses must follow the standard.
- Port 61H is a widely used port. We can use this port to generate a time delay which will work in any PC with any type of processor from the 286 to the Pentium.
- I/O port 61H has eight bits (D0-D7). Bit D4 is of particular interest to us. In all 286 and higher PCs bit D4 of port 61H changes its state every 15.085 microseconds (µs) (stays low for 15.085 µs and then changes to high and stay high for the same amount of time before it goes low again).
- This toggling of bit D4 goes on indefinitely as long as the PC is on.
- The following program shows how to use port 61H to generate a delay of 1/2 second. In this program all the bits of port 310H are toggled with a 1/2 second delay in between.



```
;TOGGLING ALL BITS OF PORT 310H EVERY 0.5 SEC
             MOV    DX,310H
HERE:        MOV    AL,55H      ;toggle all bits
             OUT    DX,AL
             MOV    CX,33144    ;delay=33144x15.085 us=0.5 sec
             CALL   TDELAY
             MOV    AL,0AAH
             OUT    DX,AL
             MOV    CX,33144
             CALL   TDELAY
             JMP    HERE

 ;CX=COUNT OF 15.085 MICROSEC
 TDELAY       PROC   NEAR
              PUSH   AX                ;save AX
 W1:          IN     AL,61H
              AND    AL,00010000B
              CMP    AL,AH
              JE     W1                ;wait for 15.085 usec
              MOV    AH,AL
              LOOP   W1                ;another 15.085 usec
              POP    AX                ;restore AX
              RET
 TDELAY       ENDP
```

Notice that, when port 61H is read, all the bits are masked except D4. The program waits for D4 to change every 15.085 µs before it loops again.

PROGRAMMING & INTERFACING THE 8255:

The 8255 is –

» a widely used 40-pin DIP I/O chip.

» Having three separately accessible ports, A, B, and C, which can be programmed to be input or output port, hence the name PPI (*programmable peripheral interface*).

» They can also be changed dynamically, in contrast to the 74LS244 and 74LS373, which are hard-wired.

Port A (PA0-PA7):

» This 8-bit port A can be programmed all as input or all as output.

Port B (PB0-PB7):

» This 8-bit port B can be programmed all as input or all as output.

Port C (PC0-PC7):

» This 8-bit port C can be programmed all as input or all as output.

» It can also be split into two parts; CU (upper bits PC4-PC7) and CL (lower bits PC0-PC3). Each can be used as input or output.

» Any bit of Port C can be programmed individually.

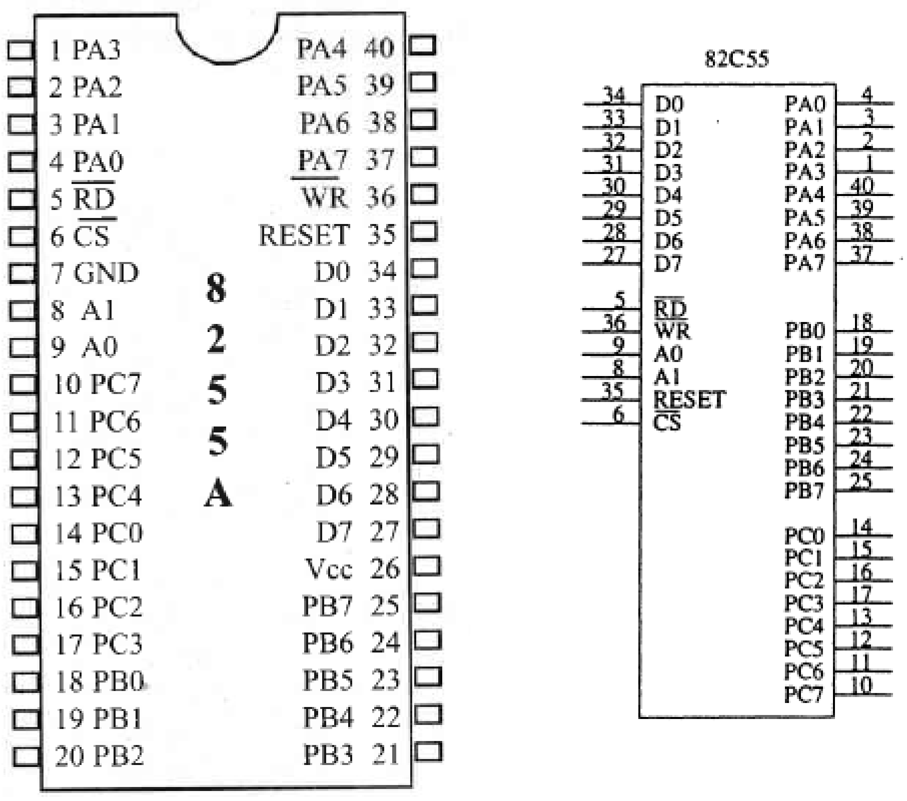


Fig: 8255 PPI Chip

RD and WR:

» Active low input signals to 8255.

» If 8255 is using peripheral I/O design, IOR and IOW of the system bus are connected to these two pins.

» If 8255 is using memory-mapped I/O, MEMR and MEMW of the system bus will activate these two pins.

RESET:

» Active high signal input to 8255.

» Used to clear the control register.

» When RESET is activated, all the ports are initialized as input ports.

» This pin must be connected to the RESET output of the system bus, or grounded, making it inactive.

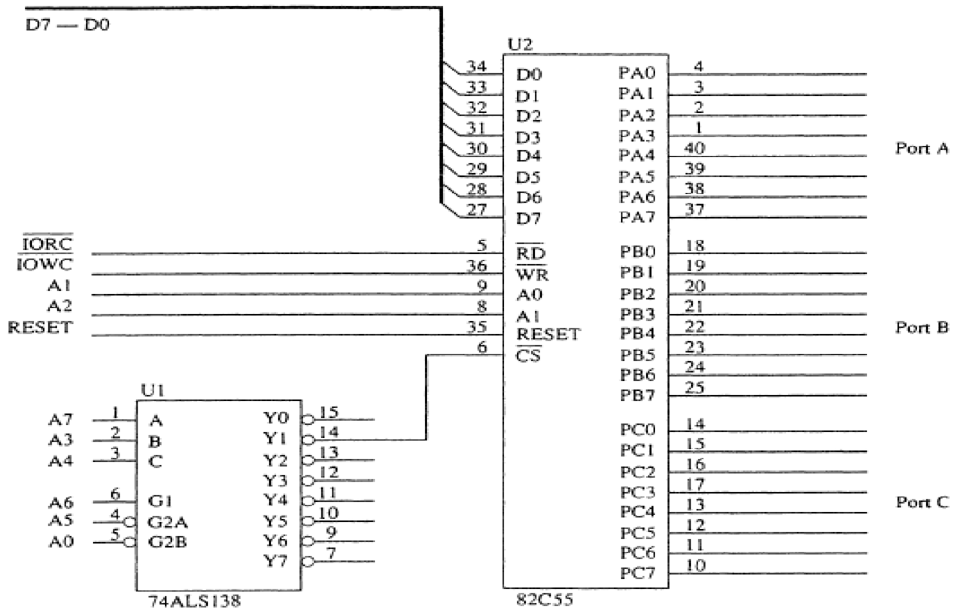
CS	A1	A0	Selects
0	0	0	Port A
0	0	1	Port B
0	1	0	Port C
0	1	1	Control Register
1	x	x	8255 is not selected

A0, A1, and CS:

» CS (chip select) selects the entire chip.

» Address pins A0 and A1 selects specific port within the 8255.

» These three pins are used to access ports A, B, C, or the control register; as shown in the table:



Mode Selection of the 8255A:

The ports (A, B, and C) of the 8255 can be programmed in various modes, as shown in the following Fig.

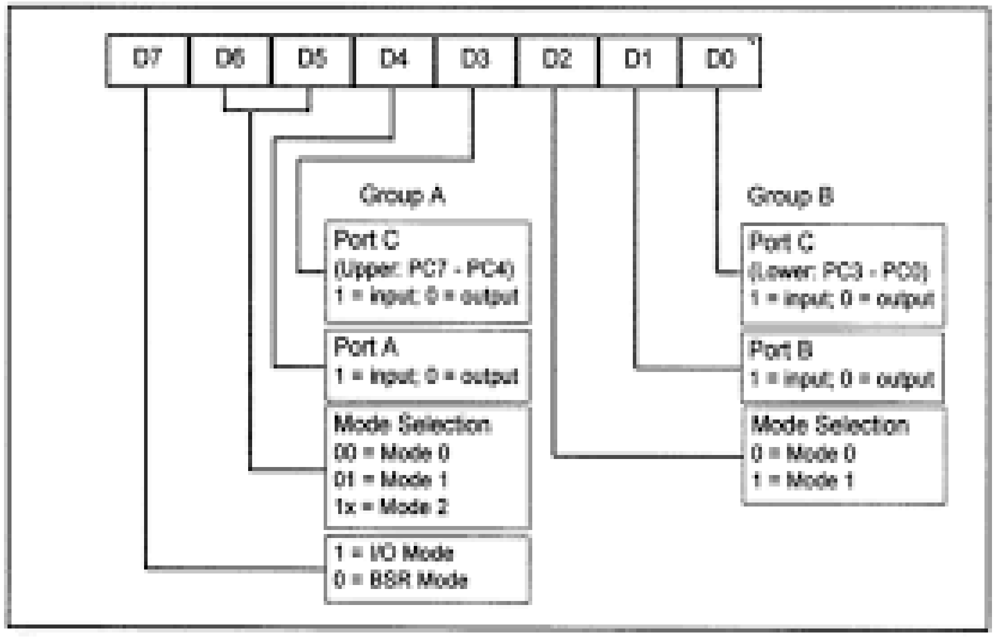
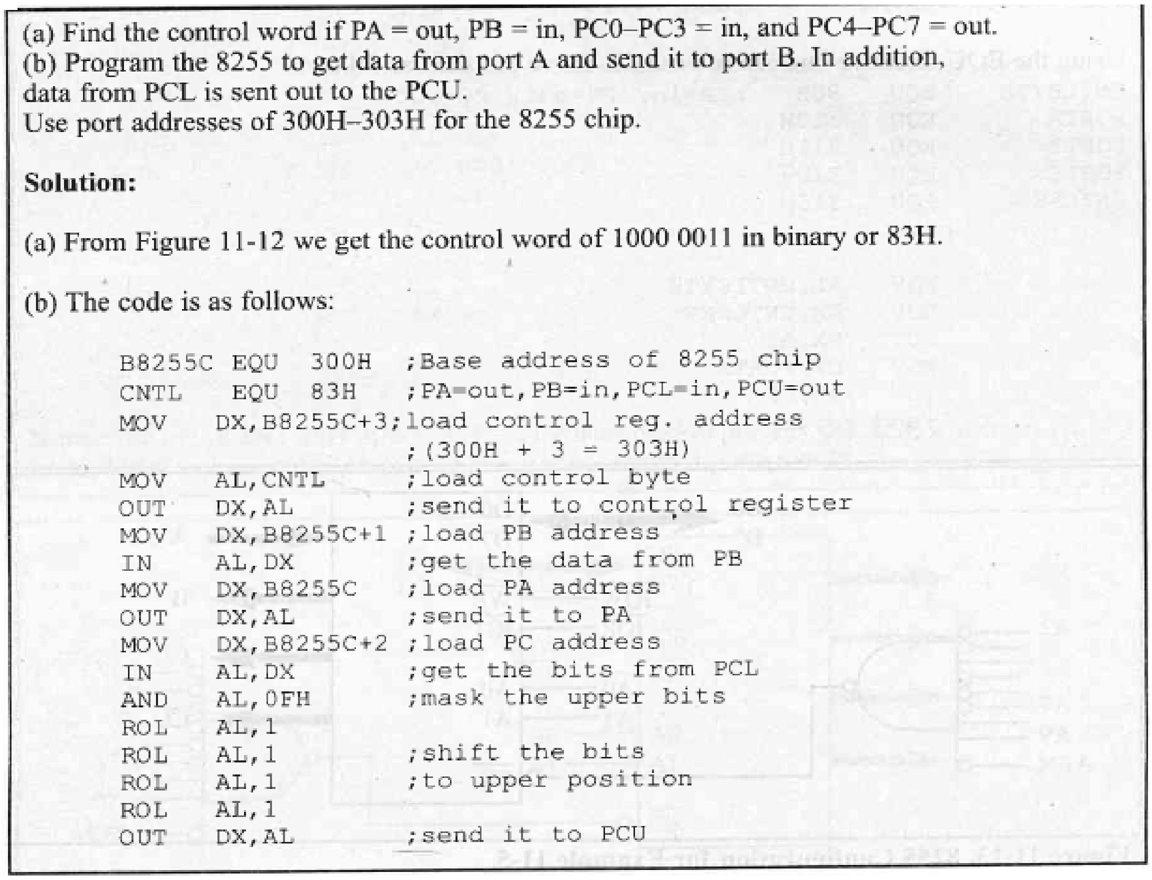


Fig: Control Word Format

Mode 0, the simple I/O mode, is the most widely used mode. In this mode, any of the ports A, B, CU, and CL can be programmed as input or output. In this mode, all bits are out or all are in. In other words, there is no control of individual bits.

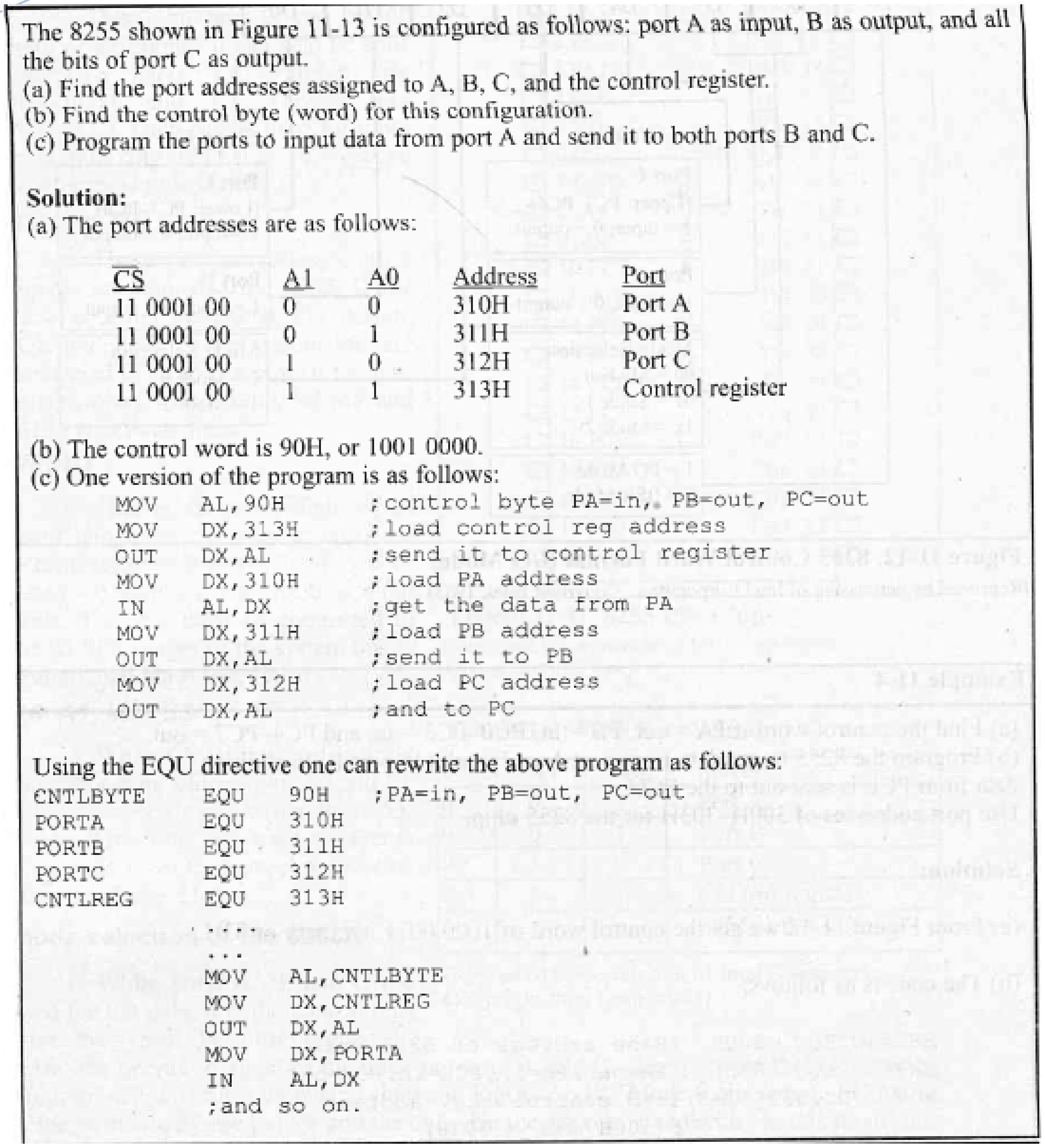

(a) Find the control word if PA = out, PB = in, PC0–PC3 = in, and PC4–PC7 = out.
(b) Program the 8255 to get data from port A and send it to port B. In addition, data from PCL is sent out to the PCU.
Use port addresses of 300H–303H for the 8255 chip.

Solution:

(a) From Figure 11-12 we get the control word of 1000 0011 in binary or 83H.

(b) The code is as follows:

```
B8255C EQU   300H   ;Base address of 8255 chip
CNTL    EQU   83H   ;PA=out,PB=in,PCL=in,PCU=out
MOV    DX,B8255C+3;load control reg. address
                  ;(300H + 3 = 303H)
MOV    AL,CNTL      ;load control byte
OUT    DX,AL        ;send it to control register
MOV    DX,B8255C+1 ;load PB address
IN     AL,DX        ;get the data from PB
MOV    DX,B8255C   ;load PA address
OUT    DX,AL        ;send it to PA
MOV    DX,B8255C+2 ;load PC address
IN     AL,DX        ;get the bits from PCL
AND    AL,0FH       ;mask the upper bits
ROL    AL,1
ROL    AL,1         ;shift the bits
ROL    AL,1         ;to upper position
ROL    AL,1
OUT    DX,AL        ;send it to PCU
```



The 8255 shown in Figure 11-13 is configured as follows: port A as input, B as output, and all the bits of port C as output.
(a) Find the port addresses assigned to A, B, C, and the control register.
(b) Find the control byte (word) for this configuration.
(c) Program the ports to input data from port A and send it to both ports B and C.

Solution:
(a) The port addresses are as follows:

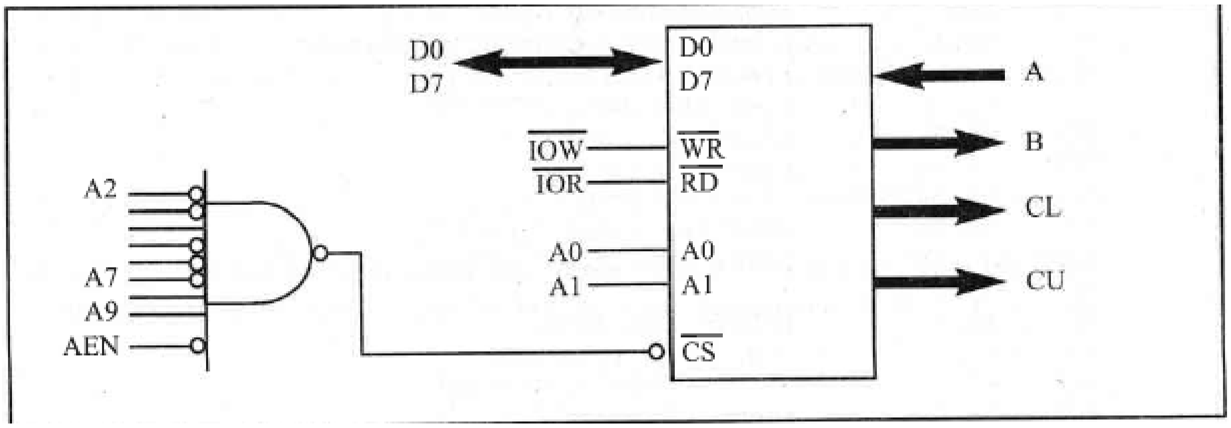
CS	A1	A0	Address	Port
11 0001 00	0	0	310H	Port A
11 0001 00	0	1	311H	Port B
11 0001 00	1	0	312H	Port C
11 0001 00	1	1	313H	Control register

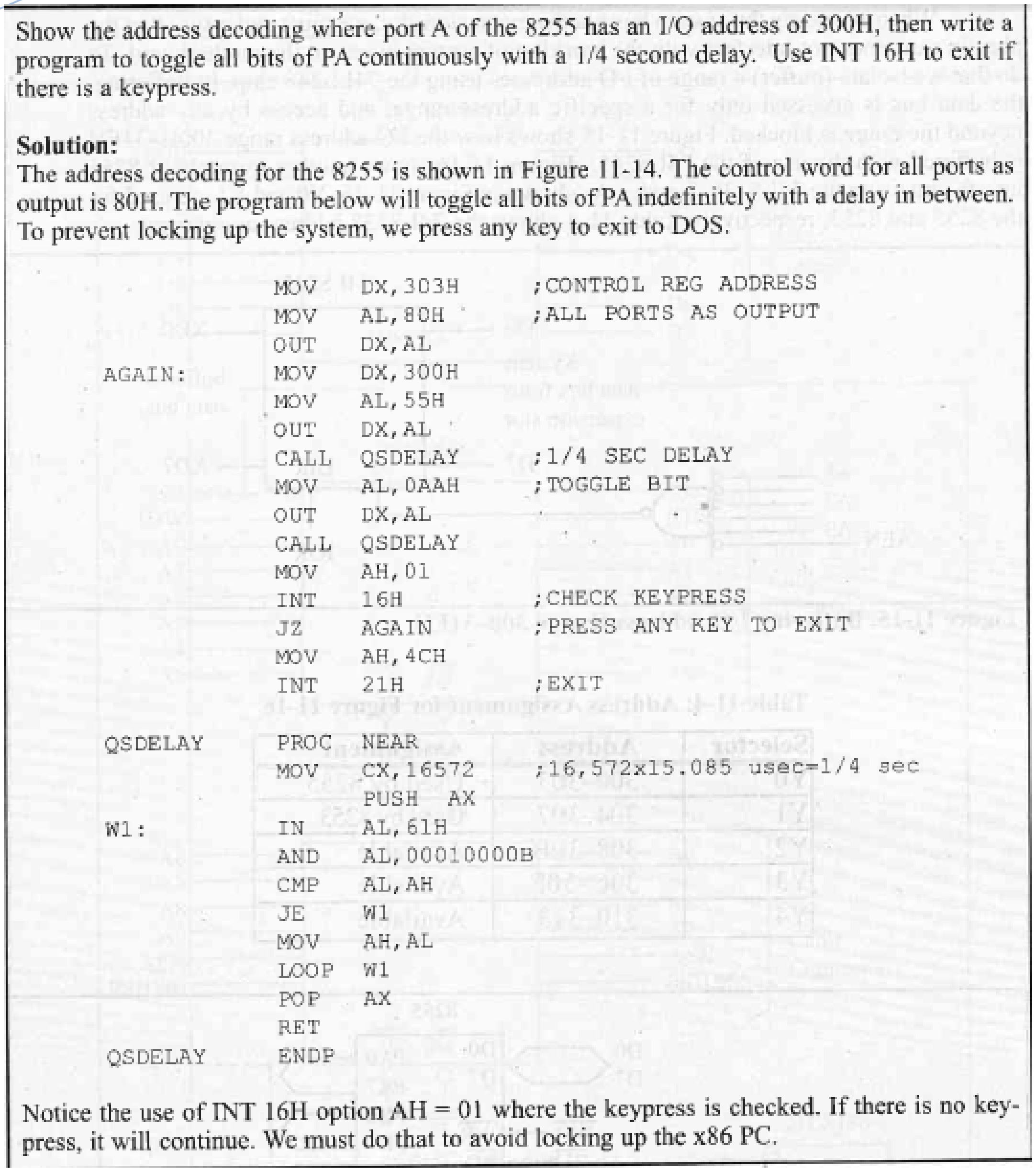
(b) The control word is 90H, or 1001 0000.
(c) One version of the program is as follows:

```
MOV     AL,90H          ;control byte PA=in, PB=out, PC=out
MOV     DX,313H         ;load control reg address
OUT     DX,AL           ;send it to control register
MOV     DX,310H         ;load PA address
IN      AL,DX           ;get the data from PA
MOV     DX,311H         ;load PB address
OUT     DX,AL           ;send it to PB
MOV     DX,312H         ;load PC address
OUT     DX,AL           ;and to PC
```

Using the EQU directive one can rewrite the above program as follows:

```
CNTLBYTE    EQU     90H     ;PA=in, PB=out, PC=out
PORTA       EQU     310H
PORTB       EQU     311H
PORTC       EQU     312H
CNTLREG     EQU     313H
            ...
            MOV     AL,CNTLBYTE
            MOV     DX,CNTLREG
            OUT     DX,AL
            MOV     DX,PORTA
            IN      AL,DX
            ;and so on.
```



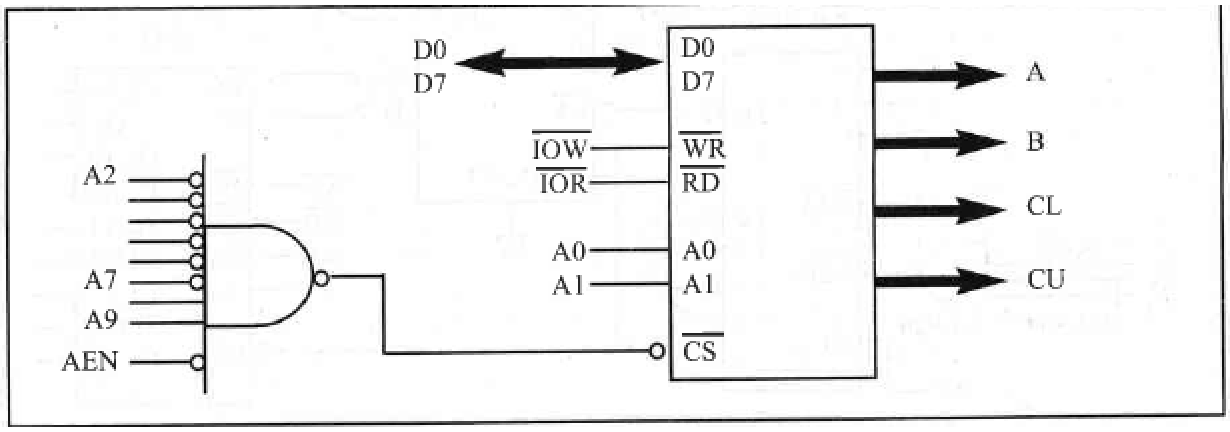


Show the address decoding where port A of the 8255 has an I/O address of 300H, then write a program to toggle all bits of PA continuously with a 1/4 second delay. Use INT 16H to exit if there is a keypress.

Solution:
The address decoding for the 8255 is shown in Figure 11-14. The control word for all ports as output is 80H. The program below will toggle all bits of PA indefinitely with a delay in between. To prevent locking up the system, we press any key to exit to DOS.

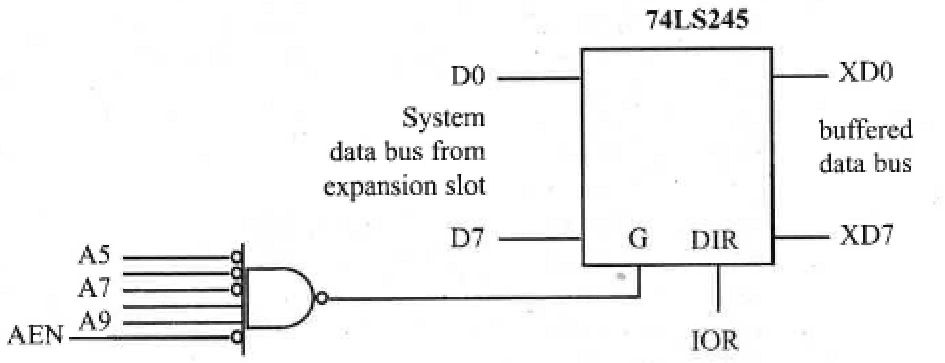
```
            MOV    DX,303H        ;CONTROL REG ADDRESS
            MOV    AL,80H         ;ALL PORTS AS OUTPUT
            OUT    DX,AL
AGAIN:      MOV    DX,300H
            MOV    AL,55H
            OUT    DX,AL
            CALL   QSDELAY        ;1/4 SEC DELAY
            MOV    AL,0AAH        ;TOGGLE BIT
            OUT    DX,AL
            CALL   QSDELAY
            MOV    AH,01
            INT    16H            ;CHECK KEYPRESS
            JZ     AGAIN          ;PRESS ANY KEY TO EXIT
            MOV    AH,4CH
            INT    21H            ;EXIT

QSDELAY     PROC   NEAR
            MOV    CX,16572       ;16,572x15.085 usec=1/4 sec
                   PUSH   AX
W1:         IN     AL,61H
            AND    AL,00010000B
            CMP    AL,AH
            JE     W1
            MOV    AH,AL
            LOOP   W1
            POP    AX
            RET
QSDELAY     ENDP
```

Notice the use of INT 16H option AH = 01 where the keypress is checked. If there is no keypress, it will continue. We must do that to avoid locking up the x86 PC.



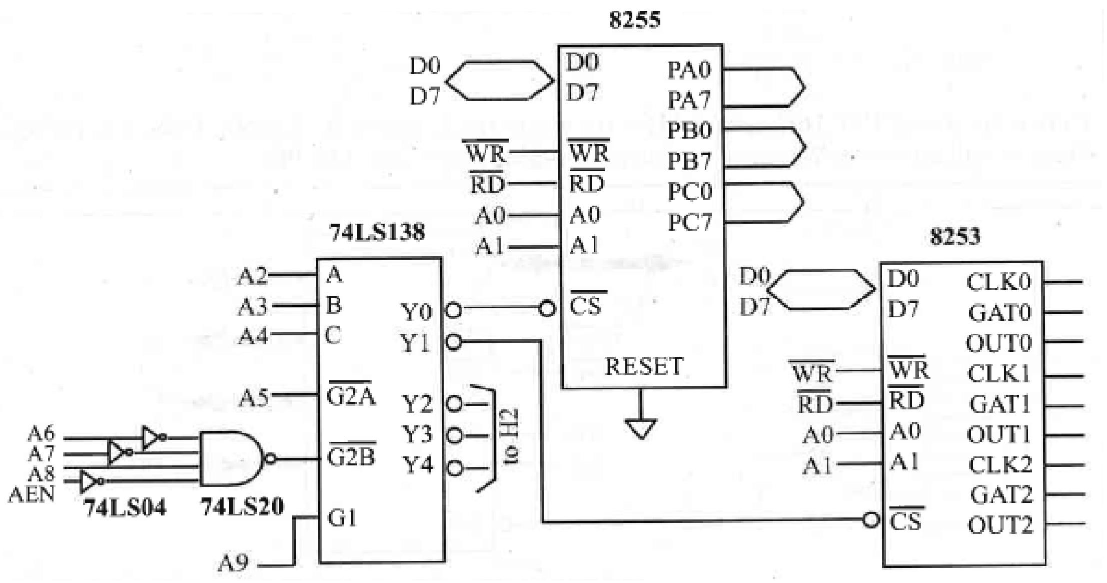
Buffering 300 – 31FH Address Range:

- When accessing the system bus via the expansion slot; we must make sure that the plug-in card does not interfere with the working of system buses on the motherboard.
- To do that we isolate (buffer) a range of I/O addresses using the 74LS245 chip.
- In buffering, the data bus is accessed only for a specific address range, and access by any address beyond the range is blocked.
- The following Fig shows how the I/O address range 300H-31FH is buffered with the use of the 74LS245.



- The following Fig shows another example of 8255 interfacing using the 74LS138 decoder. As shown in the Fig., Y0 and Y1 are used for the 8255 and 8253, respectively. The Table shows the 74LS 138 address assignment.

Selector	Address	Assignment
Y0	300–303	Used by 8255
Y1	304–307	Used by 8253
Y2	308–30B	Available
Y3	30C–30F	Available
Y4	310–313	Available



- The following Fig shows the circuit for buffering all the buses. The 74LS244 is used to boost the address and control signals.

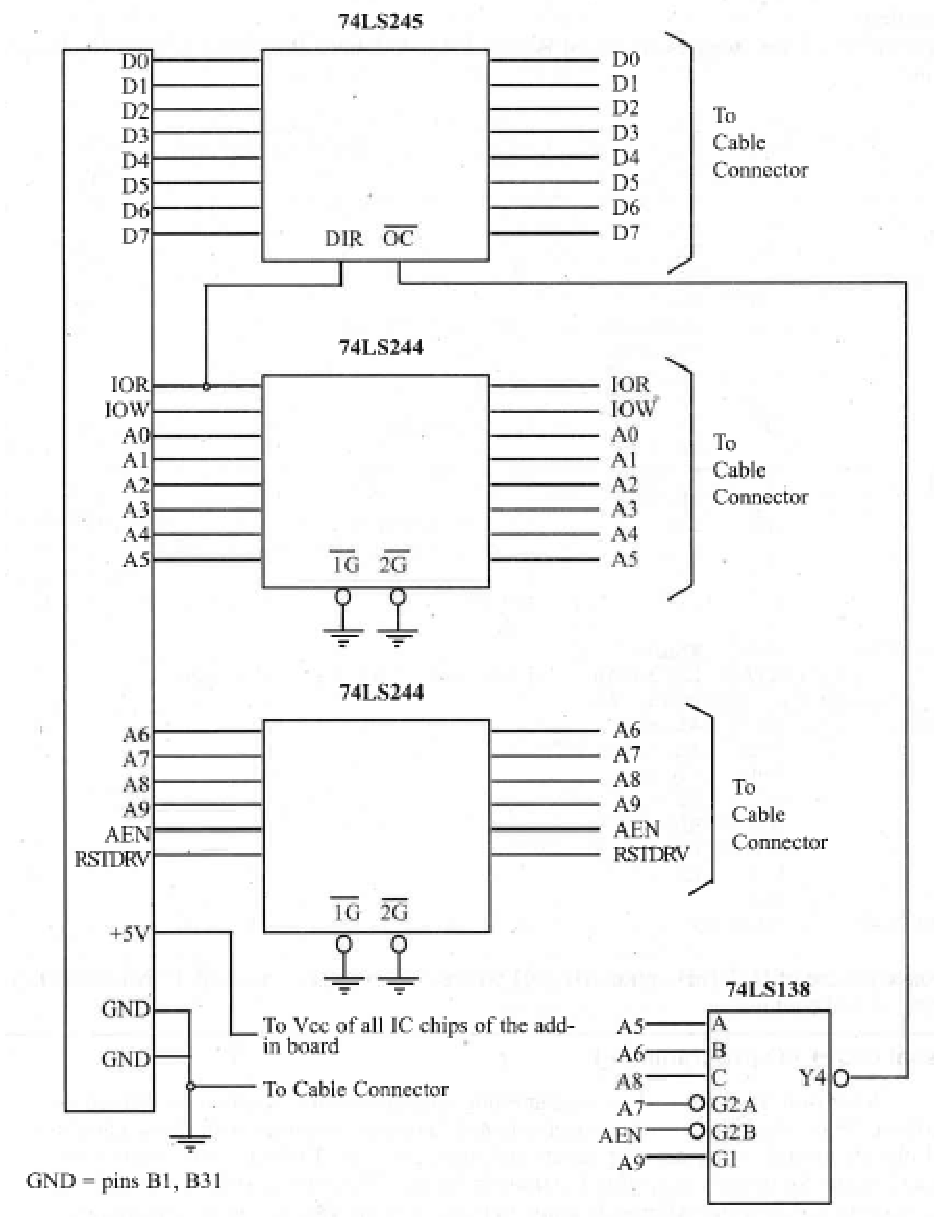
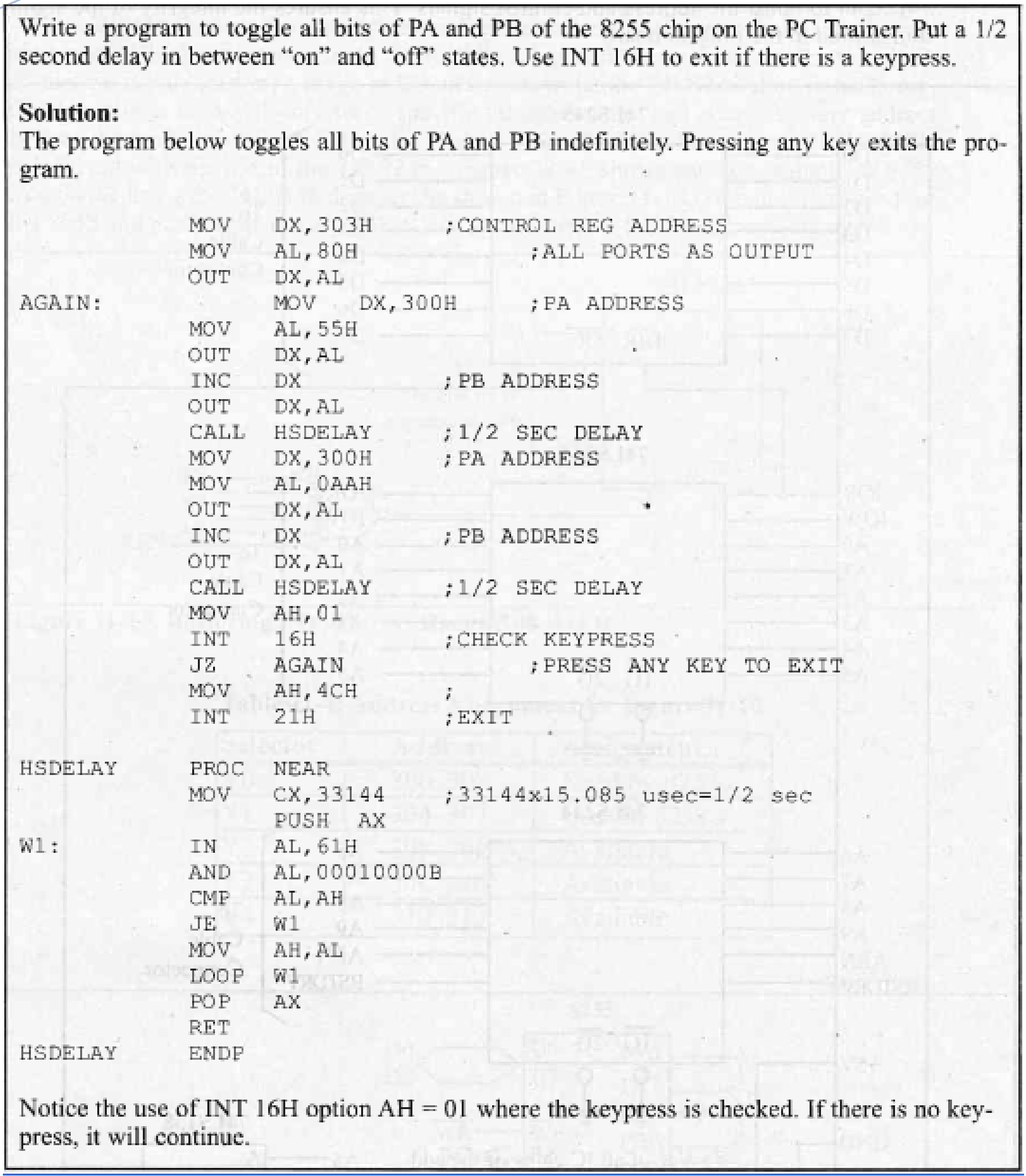


Fig: Design of 8-bit ISA PC Bus Extender

» The following shows a test program to toggle the PA and PB bits. Notice that in order to avoid locking up the system, INT 16H is used to exit upon pressing any key.



Write a program to toggle all bits of PA and PB of the 8255 chip on the PC Trainer. Put a 1/2 second delay in between "on" and "off" states. Use INT 16H to exit if there is a keypress.

Solution:
The program below toggles all bits of PA and PB indefinitely. Pressing any key exits the program.

```
            MOV     DX,303H         ;CONTROL REG ADDRESS
            MOV     AL,80H                ;ALL PORTS AS OUTPUT
            OUT     DX,AL
AGAIN:              MOV    DX,300H        ;PA ADDRESS
            MOV     AL,55H
            OUT     DX,AL
            INC     DX              ;PB ADDRESS
            OUT     DX,AL
            CALL    HSDELAY         ;1/2 SEC DELAY
            MOV     DX,300H         ;PA ADDRESS
            MOV     AL,0AAH
            OUT     DX,AL
            INC     DX              ;PB ADDRESS
            OUT     DX,AL
            CALL    HSDELAY         ;1/2 SEC DELAY
            MOV     AH,01
            INT     16H             ;CHECK KEYPRESS
            JZ      AGAIN                 ;PRESS ANY KEY TO EXIT
            MOV     AH,4CH          ;
            INT     21H             ;EXIT

HSDELAY     PROC    NEAR
            MOV     CX,33144        ;33144x15.085 usec=1/2 sec
                    PUSH   AX
W1:         IN      AL,61H
            AND     AL,00010000B
            CMP     AL,AH
            JE      W1
            MOV     AH,AL
            LOOP    W1
            POP     AX
            RET
HSDELAY     ENDP
```

Notice the use of INT 16H option AH = 01 where the keypress is checked. If there is no keypress, it will continue.

Visual C/C++ I/O Programming:

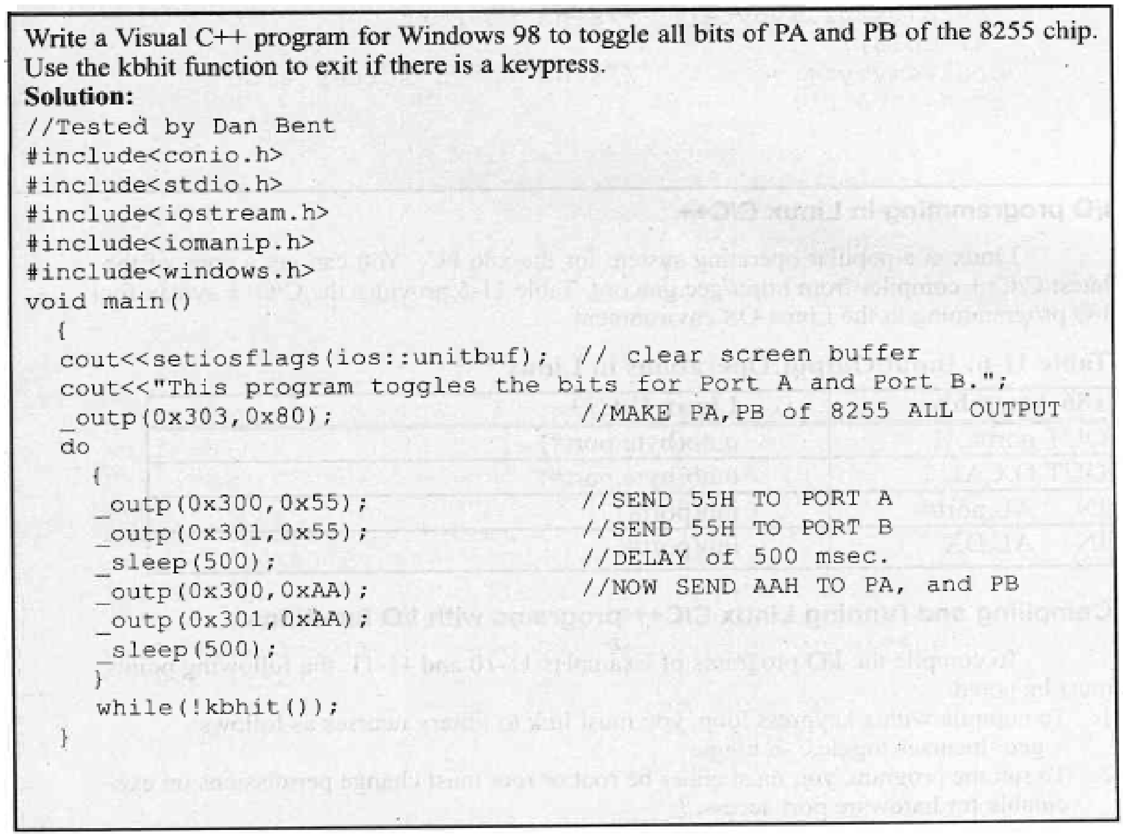
- Microsoft Visual C++ is a programming language widely used on the Windows platform.
- Since Visual C++ is an object-oriented language, it comes with many classes and objects to make programming easier and more efficient.
- But, there is no object or class for directly accessing I/O ports in the full Windows version of Visual C++.
- The reason for that is that Microsoft wants to make sure the x86 system programming is under full control of the operating system. This prevents any hacking into the system hardware.
- This applies to Windows NT, 2000, XP, and higher.

- o Hence, none of the system INT instructions such as INT 21H and I/O operations are applicable in Windows XP and its subsequent versions.
- o To access the I/O and other hardware features of the x86 PC in the XP environment you must use the Windows Platform SDK provided by Microsoft.

- The situation is different in the Windows 9x (95 and 98) environment.
- While INT 21H and other system interrupt instructions are blocked in Windows 9x, direct I/O addressing is available.
- To access I/O directly in Windows 9x, you must program Visual C++ in console mode.
- The instruction syntax for I/O operations is shown in the following Table.

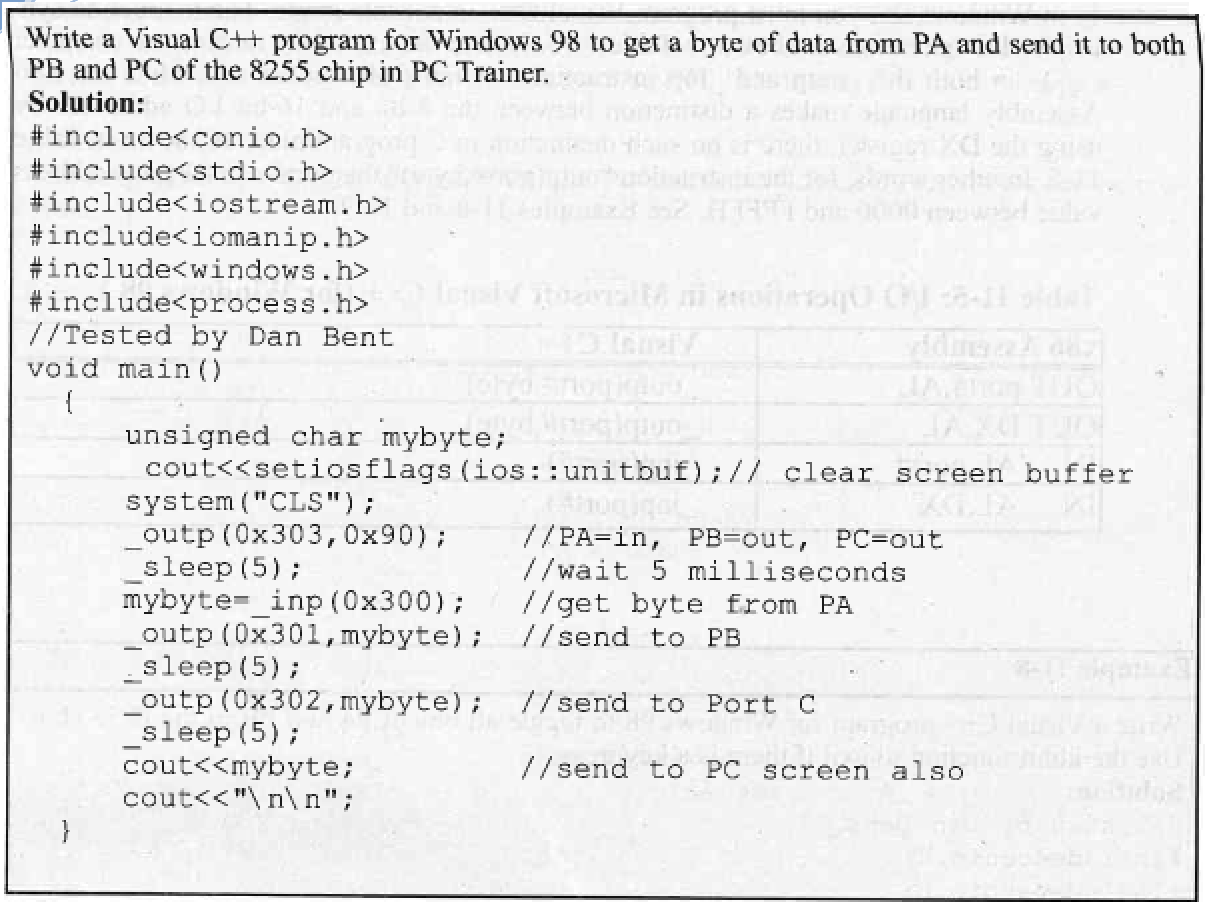
x86 Assembly	Visual C++
OUT port#, AL	_outp (port#, byte)
OUT DX, AL	_outp (port#, byte)
IN AL, port#	_inp (port#)
IN AL, DX	_inp (port#)

- Notice the use of the underscore character (_) in both the _outp and _inp instructions.
- Also note that, while the x86 Assembly language makes a distinction between the 8-bit and 16-bit I/O addresses by using the DX register, there is no such distinction in C programming. In other words, for the instruction "*outp (port#, byte)*" the port# can take any address value between 0000 and FFFFH.


Write a Visual C++ program for Windows 98 to toggle all bits of PA and PB of the 8255 chip. Use the kbhit function to exit if there is a keypress.

Solution:

```
//Tested by Dan Bent
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
#include<iomanip.h>
#include<windows.h>
void main()
  {
  cout<<setiosflags(ios::unitbuf);  // clear screen buffer
  cout<<"This program toggles the bits for Port A and Port B.";
  _outp(0x303,0x80);                //MAKE PA,PB of 8255 ALL OUTPUT
  do
    {
    _outp(0x300,0x55);              //SEND 55H TO PORT A
    _outp(0x301,0x55);              //SEND 55H TO PORT B
    _sleep(500);                    //DELAY of 500 msec.
    _outp(0x300,0xAA);              //NOW SEND AAH TO PA, and PB
    _outp(0x301,0xAA);
    _sleep(500);
    }
    while(!kbhit());
  }
```



Write a Visual C++ program for Windows 98 to get a byte of data from PA and send it to both PB and PC of the 8255 chip in PC Trainer.

Solution:

```
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
#include<iomanip.h>
#include<windows.h>
#include<process.h>
//Tested by Dan Bent
void main()
  {
     unsigned char mybyte;
      cout<<setiosflags(ios::unitbuf);// clear screen buffer
     system("CLS");
     _outp(0x303,0x90);    //PA=in, PB=out, PC=out
     _sleep(5);            //wait 5 milliseconds
     mybyte=_inp(0x300);   //get byte from PA
     _outp(0x301,mybyte);  //send to PB
     _sleep(5);
     _outp(0x302,mybyte);  //send to Port C
     _sleep(5);
     cout<<mybyte;         //send to PC screen also
     cout<<"\n\n";
  }
```

I/O Programming in Linux C/C++:

- Linux is a popular operating system for the x86 PC.
- The following Table provides the C/C++ syntax for I/O programming in the Linux OS environment.

x86 Assembly	Linux C/C++
OUT port#, AL	outb (byte, port#)
OUT DX, AL	outb (byte, port#)
IN AL, port#	inb (port#)
IN AL, DX	inb (port#)

Compiling & Running Linux C/C++ Programs with I/O Functions:

- To compile the I/O programs, the following points must be noted:
  - To compile with a keypress loop, you must link to library ncurses as follows:

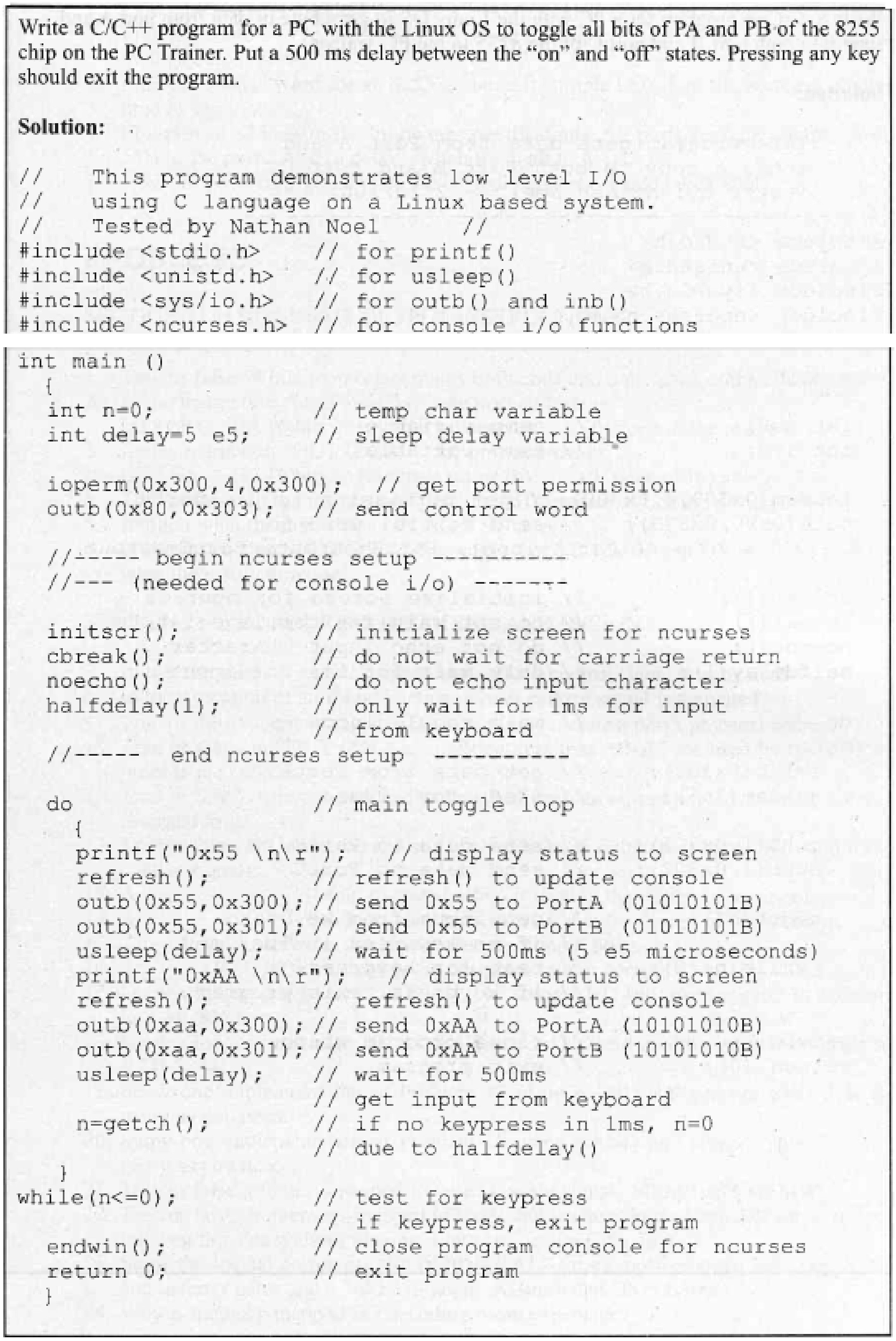
    > gcc -lncurses toggle.c -o toggle
- To run the program, you must either be root or root must change permissions on executable for hardware port access.

  Example: (as root or superuser)

> chown root toggle

> chmod 4750 toggle

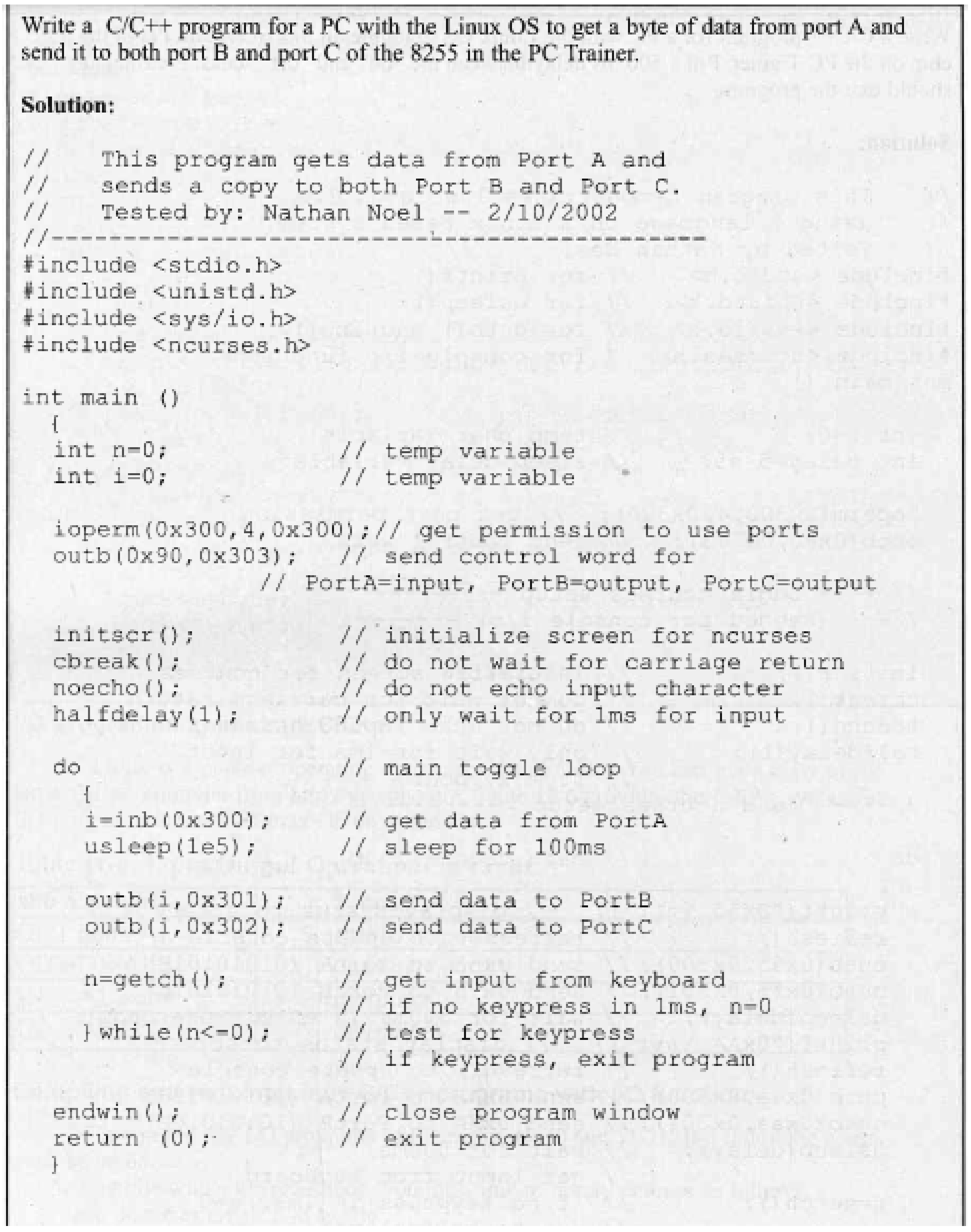
- Now toggle can be executed by users other than root.



Write a C/C++ program for a PC with the Linux OS to toggle all bits of PA and PB of the 8255 chip on the PC Trainer. Put a 500 ms delay between the "on" and "off" states. Pressing any key should exit the program.

Solution:

```
//    This program demonstrates low level I/O
//    using C language on a Linux based system.
//    Tested by Nathan Noel        //
#include <stdio.h>      // for printf()
#include <unistd.h>     // for usleep()
#include <sys/io.h>     // for outb() and inb()
#include <ncurses.h>    // for console i/o functions
int main ()
  {
  int n=0;              // temp char variable
  int delay=5 e5;       // sleep delay variable

  ioperm(0x300,4,0x300);  // get port permission
  outb(0x80,0x303);     // send control word

  //----- begin ncurses setup -----------
  //--- (needed for console i/o) -------

  initscr();            // initialize screen for ncurses
  cbreak();             // do not wait for carriage return
  noecho();             // do not echo input character
  halfdelay(1);         // only wait for 1ms for input
                        // from keyboard
  //-----  end ncurses setup  ----------

  do                    // main toggle loop
    {
    printf("0x55 \n\r");    // display status to screen
    refresh();          // refresh() to update console
    outb(0x55,0x300); // send 0x55 to PortA (01010101B)
    outb(0x55,0x301); // send 0x55 to PortB (01010101B)
    usleep(delay);      // wait for 500ms (5 e5 microseconds)
    printf("0xAA \n\r");    // display status to screen
    refresh();          // refresh() to update console
    outb(0xaa,0x300); // send 0xAA to PortA (10101010B)
    outb(0xaa,0x301); // send 0xAA to PortB (10101010B)
    usleep(delay);      // wait for 500ms
                        // get input from keyboard
    n=getch();          // if no keypress in 1ms, n=0
                        // due to halfdelay()
   }
while(n<=0);            // test for keypress
                        // if keypress, exit program
  endwin();             // close program console for ncurses
  return 0;             // exit program
  }
```



Write a C/C++ program for a PC with the Linux OS to get a byte of data from port A and send it to both port B and port C of the 8255 in the PC Trainer.

Solution:

```
//    This program gets data from Port A and
//    sends a copy to both Port B and Port C.
//    Tested by: Nathan Noel -- 2/10/2002
//-----------------------------------------------
#include <stdio.h>
#include <unistd.h>
#include <sys/io.h>
#include <ncurses.h>

int main  ()
  {
  int n=0;                // temp variable
  int i=0;                // temp variable

  ioperm(0x300,4,0x300);// get permission to use ports
  outb(0x90,0x303);     // send control word for
                     // PortA=input, PortB=output, PortC=output

  initscr();              // initialize screen for ncurses
  cbreak();               // do not wait for carriage return
  noecho();               // do not echo input character
  halfdelay(1);           // only wait for 1ms for input

  do                      // main toggle loop
    {
    i=inb(0x300);         // get data from PortA
    usleep(1e5);          // sleep for 100ms

    outb(i,0x301);        // send data to PortB
    outb(i,0x302);        // send data to PortC

    n=getch();            // get input from keyboard
                          // if no keypress in 1ms, n=0
    }while(n<=0);         // test for keypress
                          // if keypress, exit program

  endwin();               // close program window
  return  (0);            // exit program
  }
```

By: MAHESH PRASANNA K.,

DEPT. OF CSE, VCET.

****

****

MODULE – 4

ARM EMBEDDED SYSTEMS & ARM PROCESSOR FUNDAMENTALS

ARM EMBEDDED SYSTEMS

The ARM processor core is a key component of many successful 32-bit embedded systems. ARM cores are widely used in mobile phones, handheld organizers, and a multitude of other everyday portable consumer devices.

The first ARM1 prototype was designed in 1985. Over one billion ARM processors had been shipped worldwide by the end of 2001. The ARM Company bases their success on a simple and powerful original design, which continues to improve today through constant technical innovation.

For example, one of ARM's most successful cores is the ARM7TDMI. It provides up to 120 Dhrystone MIPS and is known for its high code density and low power consumption, making it ideal for mobile embedded devices.

THE RISC DESIGN PHYLOSOPHY:

- ✓ The ARM core uses *reduced instruction set computer (RISC)* architecture. RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed.
- ✓ The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware because it is easier to provide greater flexibility and intelligence in software rather than hardware. As a result, a RISC design places greater demands on the compiler.
- ✓ In contrast, the traditional *complex instruction set computer (CISC)* relies more on the hardware for instruction functionality, and consequently the CISC instructions are more complicated. The following Figure illustrates these major differences.

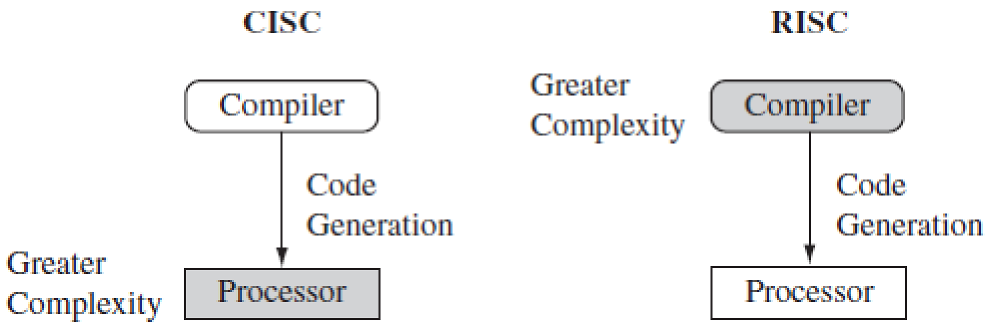


Fig: CISC vs. RISC

CISC	RISC
1. Complex instructions, taking multiple clock	1. Simple instructions, taking single clock
2. Emphasis on hardware, complexity is in the	2. Emphasis on software, complexity is in the

micro-program/processor	complier
3. Complex instructions, instructions executed by micro-program/processor	3. Reduced instructions, instructions executed by hardware
4. Variable format instructions, single register set and many instructions	4. Fixed format instructions, multiple register sets and few instructions
5. Many instructions and many addressing modes	5. Fixed instructions and few addressing modes
6. Conditional jump is usually based on status register bit	6. Conditional jump can be based on a bit anywhere in memory
7. Memory reference is embedded in many instructions	7. Memory reference is embedded in LOAD/STORE instructions

The RISC philosophy is implemented with four major **design rules**:

1. *Instructions*—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is having fixed length to allow the pipeline to fetch future instructions before decoding the current instruction.
    - In contrast, in CISC processors the instructions are often of variable size and take many cycles to execute.
2. *Pipelines*—The processing of instructions is broken down into smaller units that can be executed in parallel by pipelines. Ideally the pipeline advances by one step on each cycle for maximum throughput. Instructions can be decoded in one pipeline stage.
    - There is no need for an instruction to be executed by a mini-program called microcode as on CISC processors.
3. *Registers*—RISC machines have a large general-purpose register set. Any register can contain either data or an address. Registers act as the fast local memory store for all data processing operations.
    - In contrast, CISC processors have dedicated registers for specific purposes.
4. *Load-store architecture*—The processor operates on data held in registers. Separate load and store instructions transfer data between the register bank and external memory. Memory accesses are costly, so separating memory accesses from data processing provides an advantage because you can use data items held in the register bank multiple times without needing multiple memory accesses.
    - In contrast, with a CISC design the data processing operations can act on memory directly.

- These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies.
    - In contrast, traditional CISC processors are more complex and operate at lower clock frequencies.

THE ARM DESIGN PHYLOSOPHY:

There are a number of physical features that have driven the ARM processor design.

- ✓ Portable embedded systems require ***battery power***. The ARM processor has been specially designed to be small to reduce power consumption and extend battery operation—essential for applications such as mobile phones and personal digital assistants (PDAs).
- ✓ ***High code density*** is another major requirement since embedded systems have limited memory due to cost and/or physical size restrictions—useful for applications that have limited on-board memory, such as mobile phones and mass storage devices.
- ✓ Embedded systems are ***price sensitive***
    - Hence, *use slow and low-cost memory devices* to get substantial savings—essential for high-volume applications like digital cameras.
    - Also, *reduce the area of the die* taken up by the embedded processor; smaller the area used by the embedded processor, reduced cost of the design and manufacturing for the end product.
- ✓ ARM has incorporated ***hardware debug technology*** within the processor so that software engineers can view what is happening while the processor is executing code. With greater visibility, software engineers can resolve issues faster.
- ✓ The ARM core is ***not a pure RISC architecture*** because of the constraints of its primary application—the embedded system. In some sense, the strength of the ARM core is that it does not take the RISC concept too far.

Instruction Set for Embedded Systems:

The ARM instruction set differs from the pure RISC definition in several ways that make the ARM instruction set suitable for embedded applications:

- ✓ *Variable cycle execution for certain instructions*—Not every ARM instruction executes in a single cycle. For example, load-store-multiple instructions vary in the number of execution cycles depending upon the number of registers being transferred. The transfer can occur on sequential memory addresses. Code density is also improved since multiple register transfers are common operations at the start and end of functions.

- ✓ *Inline barrel shifter leading to more complex instructions*—The inline barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density.
- ✓ *Thumb 16-bit instruction set*—ARM enhanced the processor core by adding a second 16-bit instruction set called Thumb that permits the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30% over 32-bit fixed-length instructions.
- ✓ *Conditional execution*—An instruction is only executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.
- ✓ *Enhanced instructions*—The enhanced digital signal processor (DSP) instructions were added to the standard ARM instruction set to support fast 16×16-bit multiplier operations. These instructions allow a faster-performing ARM processor.

These ***additional features*** have made the ARM processor one of the most commonly used 32-bit embedded processor cores.

EMBEDDED SYSTEM HARDWARE:

Embedded systems can control many different devices, from small sensors found on a production line, to the real-time control systems used on a NASA space probe. All these devices use a combination of software and hardware components.

The following Figure shows a typical embedded device based on an ARM core. Each box represents a feature or function. The lines connecting the boxes are the buses carrying data.

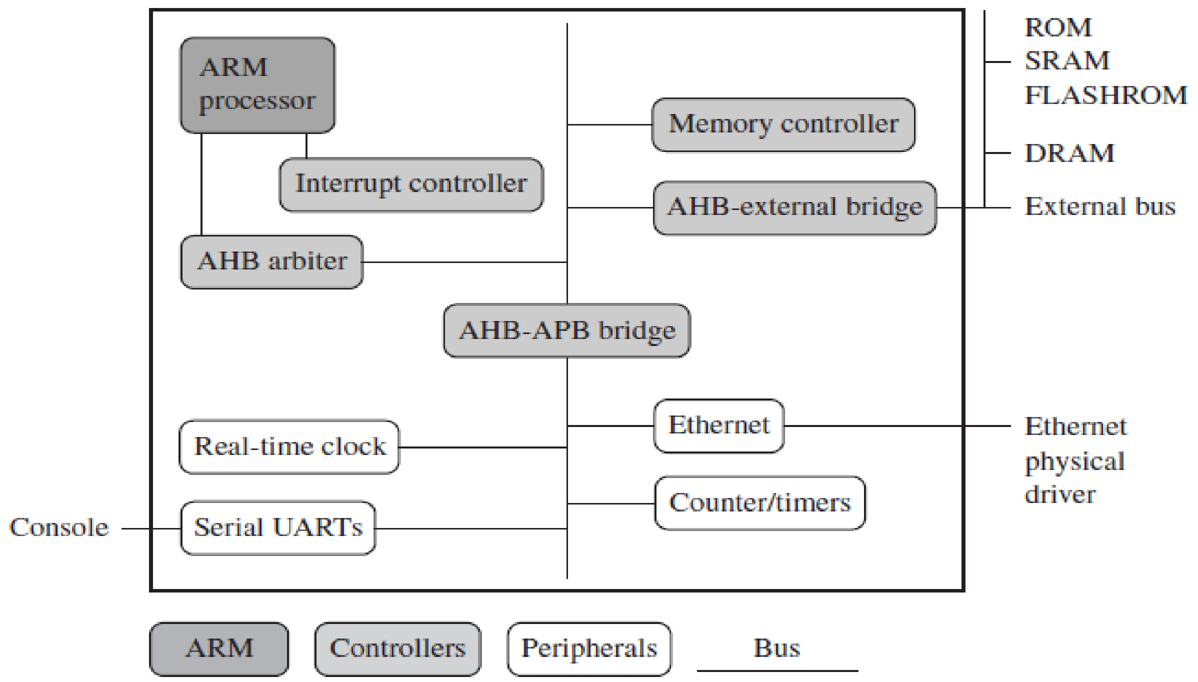


Figure: An ARM-based Embedded Device, a Microcontroller

We can separate the device into ***four main hardware components***:

1. The *ARM processor* controls the embedded device. Different versions of the ARM processor are available to suit the desired operating characteristics. An ARM processor comprises a core (the execution engine that processes instructions and manipulates data) plus the surrounding components (memory and cache) that interface it with a bus.
2. *Controllers* coordinate important functional blocks of the system. Two commonly found controllers are interrupt and memory controllers.
3. The *peripherals* provide all the input-output capability external to the chip and are responsible for the uniqueness of the embedded device.
4. A *bus* is used to communicate between different parts of the device.

ARM Bus Technology:

Embedded devices use an on-chip bus that is internal to the chip and that allows different peripheral devices to be interconnected with an ARM core.

There are ***two different classes of devices*** attached to the bus:

1. The *ARM processor core* is a bus master—a logical device capable of initiating a data transfer with another device across the same bus.
2. *Peripherals* tend to be bus slaves—logical devices capable only of responding to a transfer request from a bus master device.

A bus has ***two architecture*** levels:

A *physical level*—covers the electrical characteristics and bus width (16, 32, or 64 bits).

The *protocol*—the logical rules that govern the communication between the processor and a peripheral.

AMBA Bus Protocol:

- ✓ The *Advanced Microcontroller Bus Architecture (AMBA)* was introduced in 1996 and has been widely adopted as the on-chip bus architecture used for ARM processors.
- ✓ The first AMBA buses introduced were the *ARM System Bus (ASB)* and the *ARM Peripheral Bus (APB)*. Later ARM introduced another bus design, called the *ARM High Performance Bus* (AHB).
- ✓ Using AMBA, peripheral designers can reuse the same design on multiple projects. A peripheral can simply be bolted onto the on-chip bus without having to redesign an interface for each different processor architecture. This plug-and-play interface for hardware developers improves availability and time to market.
- ✓ AHB provides higher data throughput than ASB because it is based on a centralized multiplexed bus scheme rather than the ASB bidirectional bus design. This change allows the AHB bus to run at higher clock speeds.

- ✓ ARM has introduced ***two variations*** on the AHB bus: *Multi-layer AHB* and *AHB-Lite*.
    - o The Multi-layer AHB bus allows multiple active bus masters.
    - o AHB-Lite is a subset of the AHB bus and it is limited to a single bus master.
- ✓ The example device shown in the above Figure has three buses:
    - o an *AHB bus* for the high- performance peripherals
    - o an *APB bus* for the slower peripherals
    - o a third *bus for external peripherals*, proprietary to this device.

Memory:

An embedded system has to have some form of memory to store and execute code. You have to compare price, performance, and power consumption when deciding upon specific memory characteristics, such as hierarchy, width, and type.

Hierarchy: All computer systems have memory arranged in some form of hierarchy. The following Figure shows the memory trade-offs: the fastest memory cache is physically located nearer the ARM processor core and the slowest secondary memory is set further away. Generally the closer memory is to the processor core, the more it costs and the smaller its capacity.

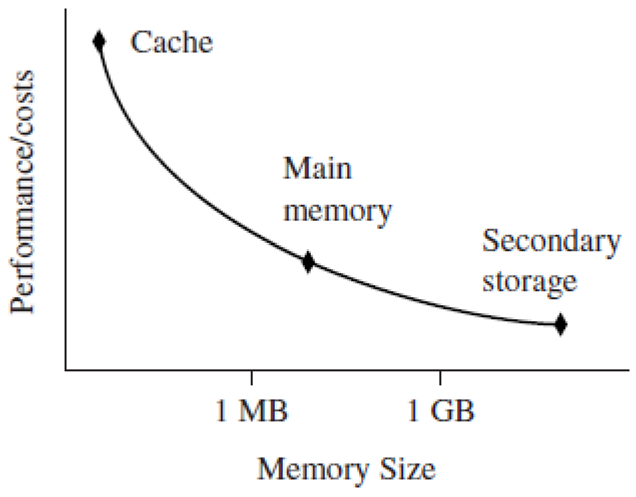


Figure: Memory Storage Trade-offs

- ✓ The *cache* is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. A cache provides an overall increase in performance but with a loss of predictable execution time. Although the cache increases the general performance of the system, it does not help real-time system response.
- ✓ The *main memory* is large—around 256 KB to 256 MB (or even greater), depending on the application—and is generally stored in separate chips. Load and store instructions access the main memory unless the values have been stored in the cache for fast access.

- ✓ *Secondary storage* is the largest and slowest form of memory. Hard disk drives and CD-ROM drives are examples of secondary storage.

Width: The memory width is the number of bits the memory returns on each access—typically 8, 16, 32, or 64 bits.

- ✓ The memory width has a direct effect on the overall performance and cost ratio. Lower bit memories are less expensive, but reduce the system performance.

The following Table summarizes theoretical cycle times on an ARM processor using different memory width devices.

Table: Fetching Instruction from Memory

Instruction Size	8-bit Memory	16-bit Memory	32-bit Memory
ARM 32-bit	4 cycles	2 cycles	1 cycles
Thumb 16-bit	2 cycles	1 cycles	1 cycles

Types: There are many *different types of memory*:

- ✓ *Read-only memory (ROM)* is the least flexible of all memory types because it contains an image that is permanently set at production time and cannot be reprogrammed.
  - o ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.
- ✓ *Flash ROM* can be written to as well as read, but it is slow to write so you shouldn't use it for holding dynamic data.
  - o Its main use is for holding the device firmware or storing long-term data that needs to be preserved after power is off. The erasing and writing of flash ROM are completely software controlled with no additional hardware circuitry required, which reduces the manufacturing costs.
- ✓ *Dynamic random access memory (DRAM)* is the most commonly used RAM for devices. It has the lowest cost per megabyte compared with other types of RAM. DRAM is dynamic—it needs to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controller before using the memory.
- ✓ *Static random access memory (SRAM)* is faster than the more traditional DRAM, but requires more silicon area. SRAM is static—the RAM does not require refreshing. The access time for SRAM is considerably shorter than the equivalent DRAM because SRAM does not require a pause between data accesses. But cost of SRAM is high.
- ✓ *Synchronous dynamic random access memory (SDRAM)* is one of many subcategories of DRAM. It can run at much higher clock speeds than conventional memory. SDRAM synchronizes itself with the processor bus, because it is clocked. Internally the data is fetched from memory cells, pipelined, and finally brought out on the bus in a burst.

Peripherals:

Embedded systems that interact with the outside world need some form of peripheral device. A ***peripheral device*** performs input and output functions for the chip by connecting to other devices or sensors that are off-chip.

- Each peripheral device usually performs a single function and may reside on-chip.
- Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.

✓ All ARM peripherals are *memory mapped*—the programming interface is a set of memory-addressed registers. The address of these registers is an offset from a specific peripheral base address.

✓ *Controllers* are specialized peripherals that implement higher levels of functionality within an embedded system.

- Two important types of controllers are memory controllers and interrupt controllers.

Memory Controllers: Memory controllers connect different types of memory to the processor bus.

- On power-up a memory controller is configured in hardware to allow certain memory devices to be active. These memory devices allow the initialization code to be executed.

Some memory devices must be set up by software; for example, when using DRAM, you first have to set up the memory timings and refresh rate before it can be accessed.

Interrupt Controllers: When a peripheral or device requires attention, it raises an *interrupt* to the processor. An *interrupt controller* provides a programmable governing policy that allows software to determine which peripheral or device can interrupt the processor at any specific time by setting the appropriate bits in the interrupt controller registers.

There are *two types of interrupt controller* available for the ARM processor: the standard interrupt controller and the vector interrupt controller.

1. The *standard interrupt controller* sends an interrupt signal to the processor core when an external device requests servicing. It can be programmed to ignore or mask an individual device or set of devices.
   - The *interrupt handler* determines which device requires servicing by reading a device bitmap register in the interrupt controller.
2. The *vector interrupt controller (VIC)* is more powerful than the standard interrupt controller, because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.
   - Depending on the type, the VIC will either call the standard interrupt exception handler, which can load the address of the handler.

EMBEDDED SYSTEM SOFTWARE:

An embedded system needs software to drive it. The following Figure shows four typical software components required to control an embedded device.

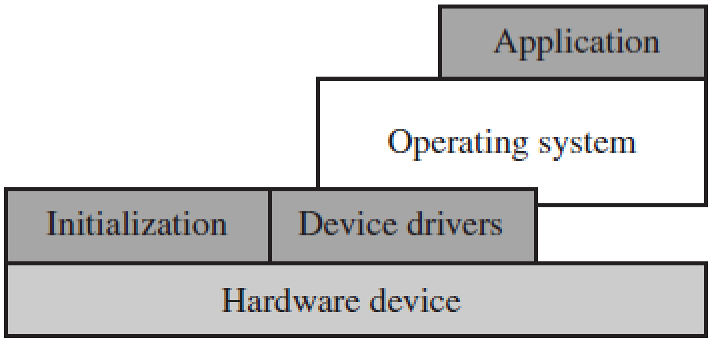


Figure: Software Abstraction Layers Executing on Hardware

- ✓ The *initialization code* is the first code executed on the board and is specific to a particular target or group of targets. It sets up the minimum parts of the board before handing control over to the operating system.
- ✓ The *operating system* provides an infrastructure to control applications and manage hardware system resources.
- ✓ The *device drivers* provide a consistent software interface to the peripherals on the hardware device.
- ✓ An *application* performs one of the tasks required for a device.
  - o For example, a mobile phone might have a diary application.

There may be multiple applications running on the same device, controlled by the operating system.

Initialization (Boot) Code:

- ✓ Initialization code (or boot code) takes the processor from the reset state to a state where the operating system can run. It usually configures the memory controller and processor caches and initializes some devices.
- ✓ The initialization code handles a number of administrative tasks prior to handing control over to an operating system image.
  - o We can group these different tasks into *three phases*: initial hardware configuration, diagnostics, and booting.

1. ***Initial hardware configuration*** involves setting up the target platform, so that it can boot an image. The target platform comes up in a standard configuration; but, this configuration normally requires modification to satisfy the requirements of the booted image.

- For example, the memory system normally requires reorganization of the memory map, as shown in the following Example.

Example: Initializing or organizing memory is an important part of the initialization code, because many operating systems expect a known memory layout before they can start.

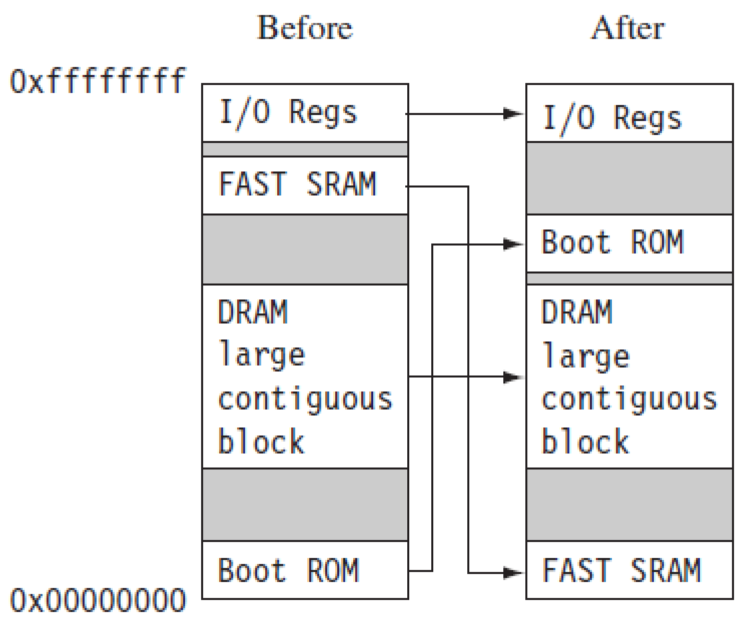


Figure: Memory Remapping

The above Figure shows memory before and after reorganization. It is common for ARM-based embedded systems to provide for memory remapping because it allows the system to start the initialization code from ROM at power-up. The initialization code then redefines or remaps the memory map to place RAM at address 0x00000000—an important step because then the exception vector table can be in RAM and thus can be reprogrammed.

2. ***Diagnostic***s are often embedded in the initialization code. Diagnostic code tests the system by exercising the hardware target to check if the target is in working order. It also tracks down standard system-related issues. The primary purpose of diagnostic code is fault identification and isolation.
3. ***Booting*** involves loading an image and handing control over to that image. The boot process itself can be complicated if the system must boot different operating systems or different versions of the same operating system.
    - Booting an image is the final phase, but first you must load the image. Loading an image involves anything from copying an entire program including code and data into RAM, to just copying a data area containing volatile variables into RAM. Once booted, the system hands over control by modifying the program counter to point into the start of the image.

Operating System:

- ✓ The initialization process prepares the hardware for an operating system to take control. An operating system organizes the system resources: the peripherals, memory, and processing time.
- ✓ ARM processors support over 50 operating systems. We can divide operating systems into *two main categories*: real-time operating systems (RTOSs) and platform operating systems.

1. ***RTOSs*** provide guaranteed response times to events. Different operating systems have different amounts of control over the system response time.
   - A *hard real-time* application requires a guaranteed response to work at all.
   - In contrast, a *soft real-time* application requires a good response time, but the performance degrades more gracefully if the response time overruns.
2. ***Platform operating systems*** require a memory management unit to manage large, non-real-time applications and tend to have secondary storage.
   - The Linux operating system is a typical example of a platform operating system.

Applications:

- ✓ The operating system schedules *applications*—code dedicated to handle a particular task. An application implements a processing task; the operating system controls the environment.
  - An embedded system can have one active application or several applications running simultaneously.
- ✓ ARM processors are found in numerous market segments, including networking, auto-motive, mobile and consumer devices, mass storage, and imaging.
- ✓ ARM processor is found in networking applications like home gateways, DSL modems for high-speed Internet communication, and 802.11 wireless communications.
- ✓ The mobile device segment is the largest application area for ARM processors, because of mobile phones.
- ✓ ARM processors are also found in mass storage devices such as hard drives and imaging products such as inkjet printers—applications that are cost sensitive and high volume.

- In contrast, ARM processors are not found in applications that require leading-edge high performance. Because these applications tend to be low volume and high cost, ARM has decided not to focus designs on these types of applications.

ARM PROCESSOR FUNDAMENTALS

A programmer can think of an ARM core as functional units connected by data buses, as shown in the following Figure.

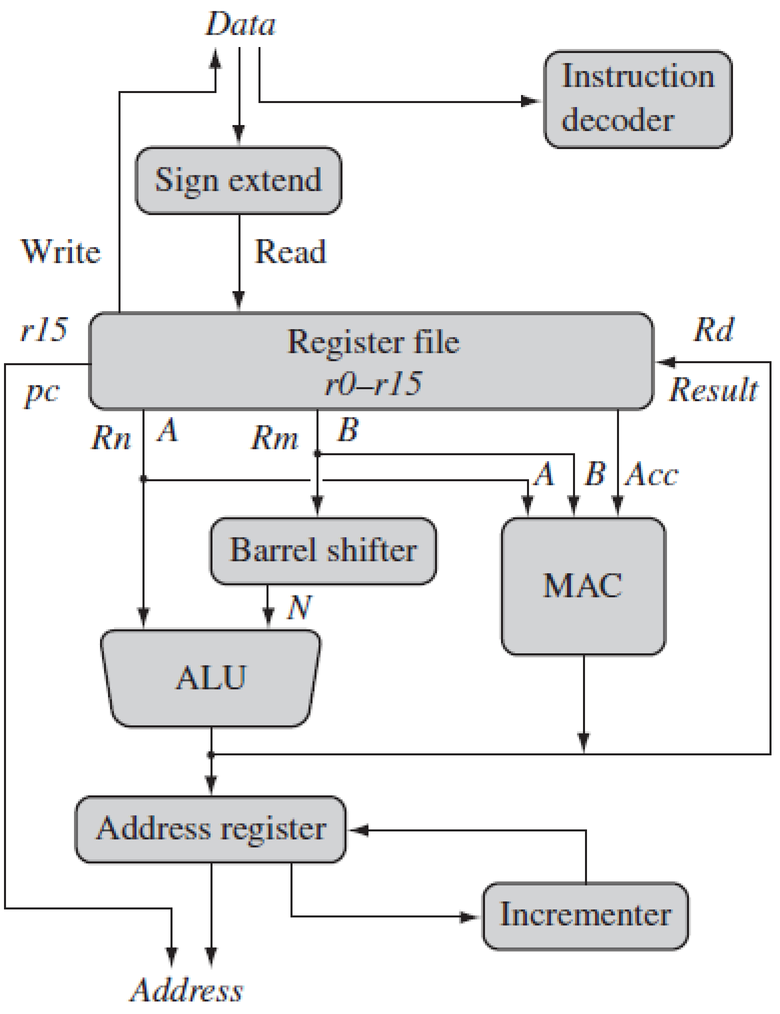


Figure: ARM Core dataflow Model

The arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area.

- ✓ Data enters the processor core through the Data bus. The data may be an instruction to execute or a data item.
  - o Figure shows a Von Neumann implementation of the ARM—data items and instructions share the same bus. (In contrast, Harvard implementations of the ARM use two different buses).
- ✓ The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.
- ✓ The ARM processor, like all RISC processors, uses ***load-store architecture***—means it has two instruction types for transferring data in and out of the processor:

- load instructions copy data from memory to registers in the core
- store instructions copy data from registers to memory

✓ There are no data processing instructions that directly manipulate data in memory. Thus, data processing is carried out in registers.

✓ Data items are placed in the register file—a storage bank made up of 32-bit registers.

- Since the ARM core is a 32-bit processor, most instructions treat the registers as holding signed or unsigned 32-bit values. The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.

✓ ARM instructions typically have two *source registers*, *Rn* and *Rm*, and a single result or *destination register*, *Rd*. Source operands are read from the register file using the internal buses A and B, respectively.

✓ The *ALU (arithmetic logic unit)* or *MAC (multiply-accumulate unit)* takes the register values *Rn* and *Rm* from the A and B buses and computes a result. Data processing instructions write the result in *Rd* directly to the register file.

✓ Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the Address bus.

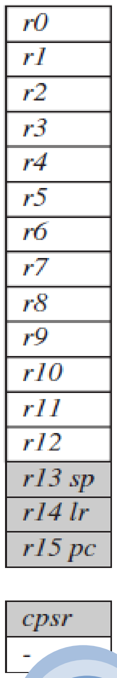
- One important feature of the ARM is that register *Rm* alternatively can be preprocessed in the barrel shifter before it enters the ALU. Together the barrel shifter and ALU can calculate a wide range of expressions and addresses.

✓ After passing through the functional units, the result in *Rd* is written back to the register file using the *Result bus*.

✓ For load and store instructions the *Incrementer* updates the address register before the core reads or writes the next register value from or to the next sequential memory location.

✓ The processor continues executing instructions until an exception or interrupt changes the normal execution flow.

REGISTERS:



General-purpose registers hold either data or an address. They are identified with the letter *r* prefixed to the register number. For example, register 4 is given the label *r4*. The Figure shows the active registers available in user mode. (A protected mode is normally used when executing applications).

✓ The processor can operate in seven different modes.

✓ All the registers shown are 32 bits in size.

✓ There are up to 18 active registers:

- 16 data registers and 2 processor status registers.
- The data registers visible to the programmer are *r0* to *r15*.

- ✓ The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*. They are given with different labels to differentiate them from the other registers.
    - o *Register r13* is traditionally used as the ***stack pointer (sp)*** and stores the head of the stack in the current processor mode.
    - o *Register r14* is called the ***link register (lr)*** and is where the core puts the return address whenever it calls a subroutine.
    - o *Register r15* is the ***program counter (pc)*** and contains the address of the next instruction to be fetched by the processor.
- ✓ In ARM state the registers *r0* to *r13* are orthogonal—any instruction that you can apply to *r0* you can equally well apply to any of the other registers.
- ✓ In addition to the 16 data registers, there are two program status registers: ***cpsr (current program status register)*** and ***spsr (saved program status register)***.

CURRENT PROGRAM STATUS REGISTER:

The ARM core uses the *cpsr* to monitor and control internal operations. The *cpsr* is a dedicated 32-bit register and resides in the register file. The following Figure shows the basic layout of a generic program status register. Note that the shaded parts are reserved for future expansion.

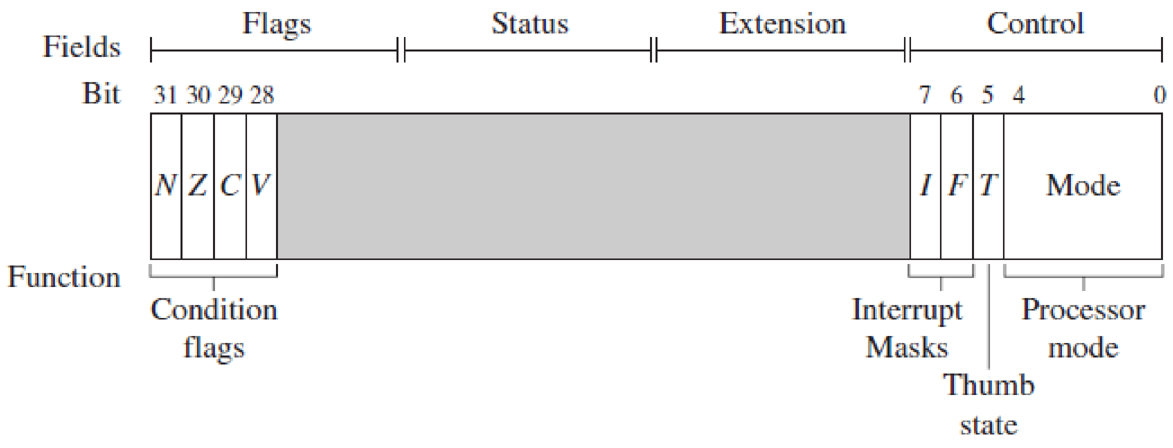


Figure: A Generic Program Status Register (psr)

The *cpsr* is divided into four fields, each 8 bits wide: *flags*, *status*, *extension*, and *control*. In current designs the extension and status fields are reserved for future use.

- ✓ The ***control field*** contains the *processor mode*, *state*, and *interrupt mask* bits.
- ✓ The ***flags field*** contains the *condition flags*.

Some ARM processor cores have extra bits allocated. For example, the *J bit*, which can be found in the flags field, is only available on *Jazelle-enabled processors*, which execute 8-bit instructions.

It is highly probable that future designs will assign extra bits for the monitoring and control of new features.

Processor Modes:

- ✓ The processor mode determines which registers are active and the access rights to the *cpsr* register itself. Each processor mode is either privileged or non-privileged:
  - o A *privileged mode* allows full read-write access to the *cpsr*.
  - o A *non-privileged mode* only allows read access to the control field in the *cpsr*, but still allows read-write access to the condition flags.
- ✓ There are *seven processor modes* in total:
  - o *six privileged modes* (abort, fast interrupt request, interrupt request, supervisor, system, and undefined)
    - The processor enters ***abort mode*** when there is a failed attempt to access memory.
    - ***Fast interrupt request*** and ***interrupt request modes*** correspond to the two interrupt levels available on the ARM processor.
    - ***Supervisor mode*** is the mode that the processor is in after reset and is generally the mode that an operating system kernel operates in.
    - ***System mode*** is a special version of user mode that allows full read-write access to the *cpsr*.
    - ***Undefined mode*** is used when the processor encounters an instruction that is undefined or not supported by the implementation.
  - o *one non-privileged mode* (user).
    - ***User mode*** is used for programs and applications.

Banked Registers:

The following Figure shows all 37 registers in the register file.

- ✓ Of these, 20 registers are hidden from a program at different times.
- ✓ These registers are called *banked registers* and are identified by the shading in the diagram.
- ✓ They are available only when the processor is in a particular mode; for example, abort mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*.
- ✓ Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or _mode.
- ✓ Every processor mode except user mode can change mode by writing directly to the mode bits of the *cpsr*.
- ✓ All processor modes except system mode have a set of associated banked registers that are a subset of the main 16 registers.
- ✓ A banked register maps one-to-one onto a user mode register.

- ✓ If you change processor mode, a banked register from the new mode will replace an existing register.
  - o For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named *r13* and *r14*. However, these registers are the banked registers *r13_irq* and *r14_irq*. The user mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers. A program still has normal access to the other registers *r0* to *r12*.

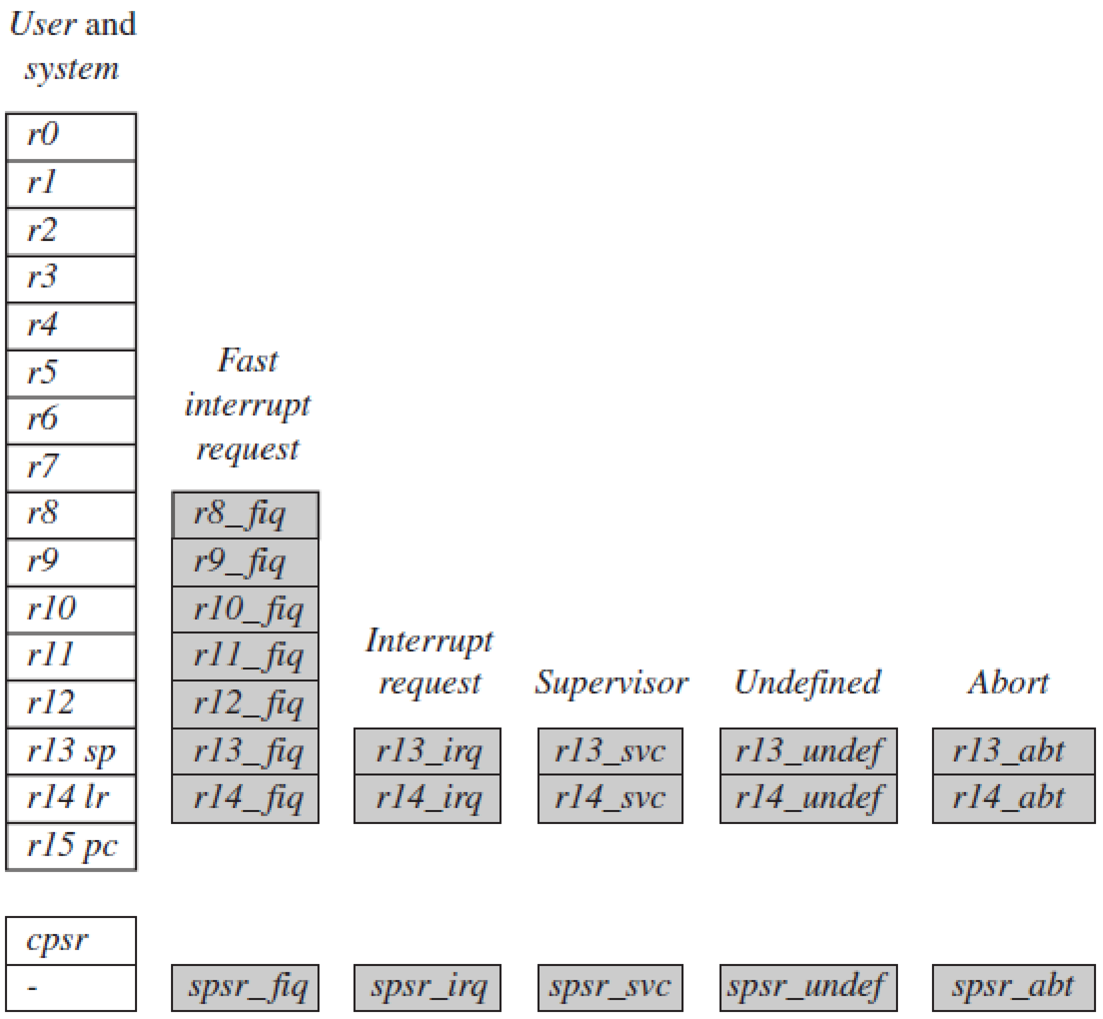


Figure: Complete ARM Register Set

- ✓ The processor mode can be changed by a program that writes directly to the *cpsr* (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt.
- ✓ The following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*.
- ✓ Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location.

✓ The following Figure illustrates what happens when an interrupt forces a mode change.

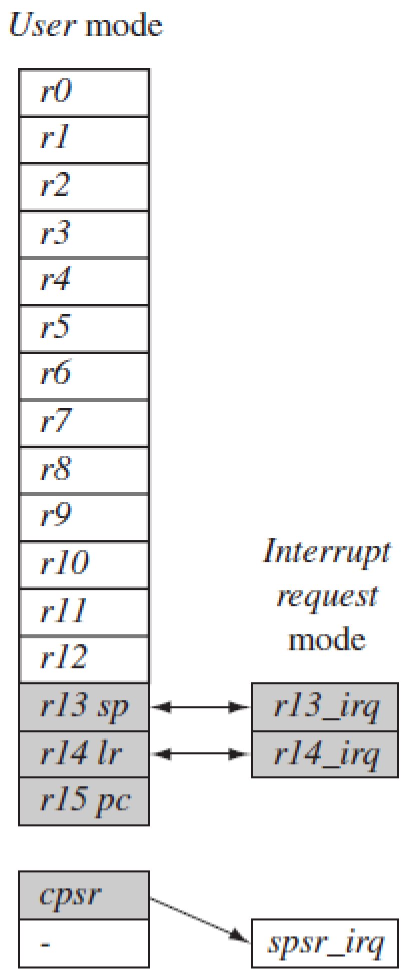


Figure: Changing Mode on an Exception

✓ The Figure shows the core changing from user mode to interrupt request mode, which happens when an interrupt request occurs due to an external device raising an interrupt to the processor core.

✓ This change causes user registers *r13* and *r14* to be banked. The user registers are replaced with registers *r13_irq* and *r14_irq*, respectively.

- o Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for interrupt request mode.

✓ The above Figure also shows a new register appearing in interrupt request mode: the *saved program status register (spsr)*, which stores the previous mode's *cpsr*. The *cpsr* being copied into *spsr_irq*.

✓ To return back to user mode, a special return instruction is used that instructs the core to restore the original *cpsr* from the *spsr_irq* and bank in the user registers *r13* and *r14*.

- ✓ Note that, the *spsr* can only be modified and read in a privileged mode. There is no *spsr* available in user mode.

- ✓ Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*. The saving of the *cpsr* only occurs when an exception or interrupt is raised.
- ✓ When power is applied to the core, it starts in supervisor mode, which is privileged. Starting in a privileged mode is useful since initialization code can use full access to the *cpsr* to set up the stacks for each of the other modes.
- ✓ The following Table lists the various modes and the associated binary patterns. The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

Table: Processor Mode

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast Interrupt Request	fiq	yes	10001
Interrupt Request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

State and Instruction Sets:

- ✓ The state of the core determines which instruction set is being executed. There are three instruction sets:
  - ARM
  - Thumb
  - Jazelle.
- ✓ The ***ARM instruction set*** is only active when the processor is in ARM state.
- ✓ The ***Thumb instruction set*** is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions.
- ✓ You cannot inter-mingle sequential ARM, Thumb, and Jazelle instructions.
- ✓ The Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor.
  - o When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions. This is the case when power is applied to the processor.
  - o When the *T* bit is 1, then the processor is in Thumb state.

- ✓ To change states the core executes a specialized branch instruction.

The following Table compares the ARM and Thumb instruction set features.

Table: ARM and Thumb Instruction Set Features

-	ARM (*cspr T* = 0)	Thumb (*cspr T* = 1)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +pc	8 general-purpose registers +7 high registers +pc

- ✓ The ARM designers introduced a third instruction set called Jazelle. **Jazelle** executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java byte-codes.
- ✓ To execute Java byte-codes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.

The following Table gives the Jazelle instruction set features.

Table: Jazelle instruction set features

-	Jezelle (*cspr T* = 0, *J* – 1)
Instruction size	8-bit
Core Instructions	Over 60% of the Java byte-codes are implemented in hardware; the rest of the codes are implemented in software

Interrupt Masks:

- ✓ *Interrupt masks* are used to stop specific interrupt requests from interrupting the processor.
- ✓ There are two interrupt request levels available on the ARM processor core—
  - o interrupt request (IRQ)
  - o fast interrupt request (FIQ).
- ✓ The *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ, respectively.
- ✓ The *I* bit masks IRQ when set to binary 1; and similarly, the *F* bit masks FIQ when set to binary 1.

Condition Flags:

- ✓ Condition flags are updated by comparisons and the result of ALU operations that specify the **S** instruction suffix.
  - o For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* Flag in the *cpsr* is set. This particular subtract instruction specifically updates the *cpsr*.
- ✓ With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation has occurred in an enhanced DSP instruction. The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly.
- ✓ In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state. The *J* bit is not generally usable and is only available on some processor cores. To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.
- ✓ Most ARM instructions can be executed conditionally on the value of the condition flags.

The following Table lists the condition flags and a short description on what causes them to be set.

Table: Condition Flags

Flag	Flag Name	Set When
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero
N	Negative	bit 31 of the result is a binary 1

These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution. The following Figure shows a typical value for the *cpsr* with both DSP extensions and Jazelle.

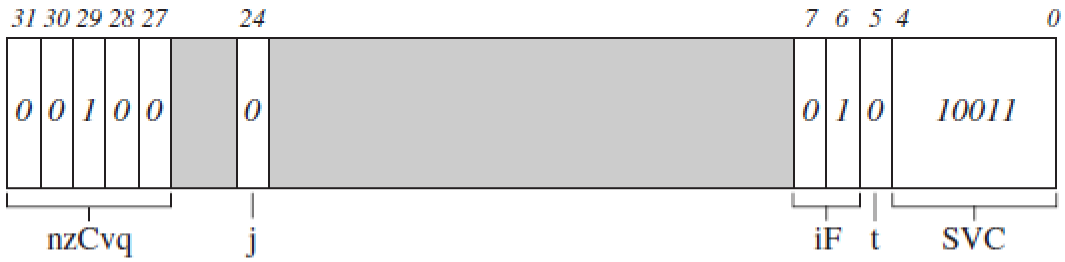


Figure: Example: *cspr* = nzCvqjiFt_SVC

- ✓ For the condition flags a capital letter shows that the flag has been set. For interrupts a capital letter shows that an interrupt is disabled.
- ✓ In the *cpsr* example shown in above Figure, the *C* flag is the only condition flag set. The rest nzvq flags are all clear.

- ✓ The processor is in ARM state because neither the Jazelle *j* nor Thumb *t* bits are set. The IRQ interrupts are enabled, and FIQ interrupts are disabled.
- ✓ Finally, you can see from the Figure, the processor is in *supervisor (SVC) mode*, since the mode[4:0] is equal to binary 10011.

Conditional Execution:

- ✓ Conditional execution controls whether or not the core will execute an instruction.
- ✓ Prior to execution, the processor compares the condition attribute with the condition flags in the *cpsr*. If they match, then the instruction is executed; otherwise the instruction is ignored.
- ✓ The condition attribute is post-fixed to the instruction mnemonic, which is encoded into the instruction.
- ✓ The following Table lists the conditional execution code mnemonics. When a condition mnemonic is not present, the default behavior is to set it to always (AL) execute.

Table: Condition Mnemonics

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

PIPELINE:

- ✓ A *pipeline* is the mechanism in a RISC processor, which is used to execute instructions.
- ✓ Pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.

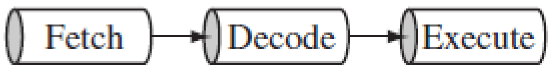


Figure: ARM7 Three-stage Pipeline

The above Figure shows a three-stage pipeline:

- *Fetch* loads an instruction from memory.
- *Decode* identifies the instruction to be executed.
- *Execute* processes the instruction and writes the result back to a register.

The following Figure illustrates pipeline using a simple example.

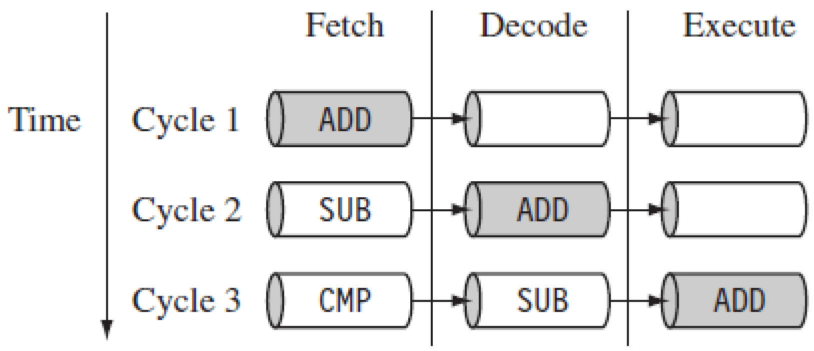


Figure: Pipelined Instruction Sequence

- ✓ The Figure shows a sequence of three instructions being fetched, decoded, and executed by the processor.
  - The three instructions are placed into the pipeline sequentially.
  - In the first cycle, the core fetches the ADD instruction from memory.
  - In the second cycle, the core fetches the SUB instruction and decodes the ADD instruction.
  - In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched.
- ✓ This procedure is called ***filling the pipeline***.
- ✓ The pipeline allows the core to execute an instruction every cycle.
- As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn *increases the performance*.
- The increased pipeline length also means increased *system latency* and there can be *data dependency* between certain stages.

- The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages, as shown in Figure.

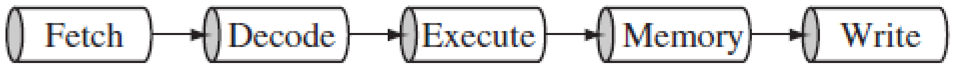


Figure: ARM9 Five-stage Pipeline

- The ARM9 adds a memory and writeback stage, which allows the ARM9 to –
  - process on average 1.1 Dhrystone MIPS per MHz
  - increase the instruction throughput in ARM9 by around 13% compared with an ARM7.
- The ARM10 increases the pipeline length still further by adding a sixth stage, as shown in the following Figure.

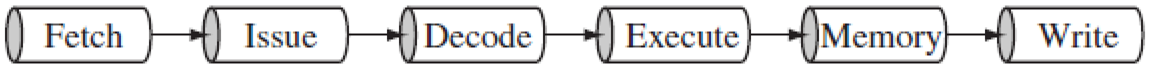


Figure: ARM10 Six-stage Pipeline

- The ARM10 –
  - can process on average 1.3 Dhrystone MIPS per MHz
  - have about 34% more throughput than an ARM7 processor core
  - but again at a higher latency cost.

NOTE: Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7. Hence, code written for the ARM7 will execute on an ARM9 or ARM10.

Pipeline Executing Characteristics:

- ✓ The ARM pipeline will not process an instruction, until it passes completely through the execute stage.
  - For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.

The following Figure shows an instruction sequence on an ARM7 pipeline.

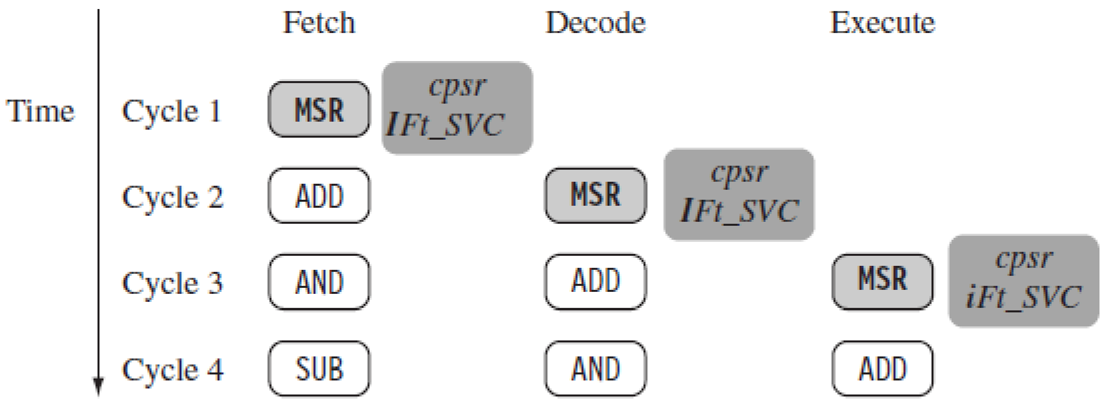


Figure: ARM Instruction Sequence

- ✓ The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline. It clears the *I* bit in the *cpsr* to enable the IRQ interrupts.

- ✓ Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

The following Figure illustrates the use of the pipeline and the program counter *pc*.

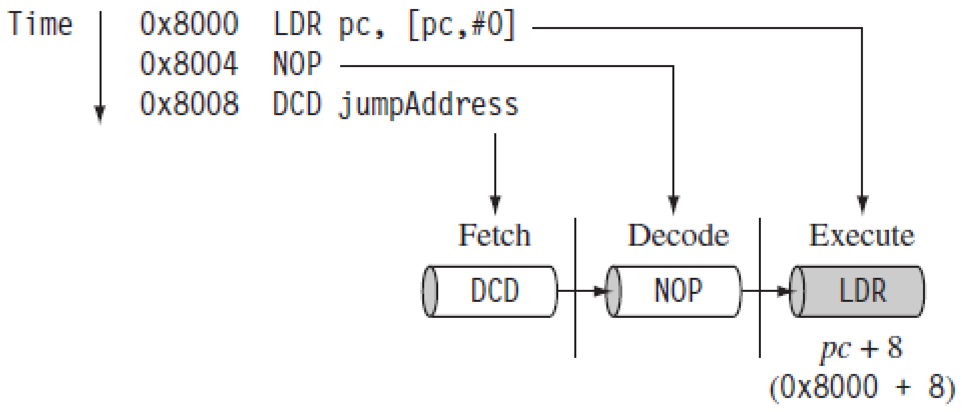


Figure: Example: *pc* = address + 8

- ✓ In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes. In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead.
- ✓ Note when the processor is in Thumb state the *pc* is the instruction address plus 4.

- ✓ There are three other characteristics of the pipeline.
  - o First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.
  - o Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.
  - o Third, an instruction in the execute stage will complete even though an interrupt has been raised. Other instructions in the pipeline will be abandoned, and the processor will start filling the pipeline.

EXCEPTIONS, INTERRUPTS AND THE VECTOR TABLE:

- ✓ When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address. The address is within a special address range called the *vector table*.
  - o The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
  - o The memory map address 0x00000000 (or in some processors starting at the offset 0xffff0000) is reserved for the vector table, a set of 32-bit words.
- ✓ When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see the following Table).

Table: The Vector Table

Exception/Interrupt	Shorthand	Address	High Address
Reset	RESET	0x00000000	0x00000000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	SABT	0x00000010	0xffff0010
Reserved	–	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- ✓ Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:
  - o ***Reset*** vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
  - o ***Undefined*** instruction vector is used when the processor cannot decode an instruction.
  - o ***Software interrupt*** vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.
  - o ***Prefetch abort*** vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
  - o ***Data abort*** vector is similar to a prefetch abort, but is raised when an instruction attempts to access data memory without the correct access permissions.
  - o ***Interrupt request*** vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.
  - o ***Fast interrupt request*** vector is similar to the interrupt request, but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the *cpsr*.

CORE EXTENSIONS:

- ✓ *Core extensions* are the standard hardware components placed next to the ARM core.
- ✓ They improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications.

Each ARM family has different extensions available. There are *three hardware extensions*: cache and tightly coupled memory, memory management, and the coprocessor interface.

Cache and Tightly Coupled Memory:

- ✓ The cache is a block of fast memory placed between main memory and the core. It allows for more efficient fetches from some memory types. With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.
- ✓ Most ARM-based embedded systems use a single-level cache internal to the processor.
- ✓ ARM has *two forms of cache*. The first is found attached to the Von Neumann–style cores. It combines both data and instruction into a single unified cache, as shown in the following Figure.

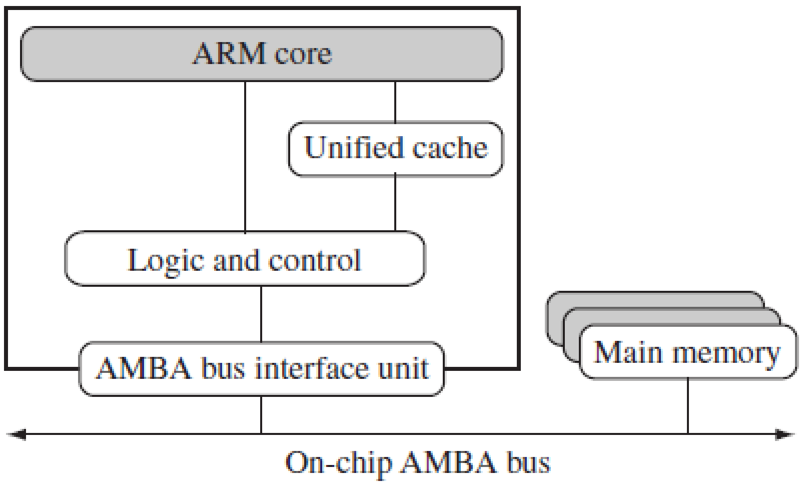


Figure: Von Neumann Architecture with Cache

- ✓ The second form, attached to the Harvard-style cores, has separate caches for data and instruction, as shown in the following Figure.

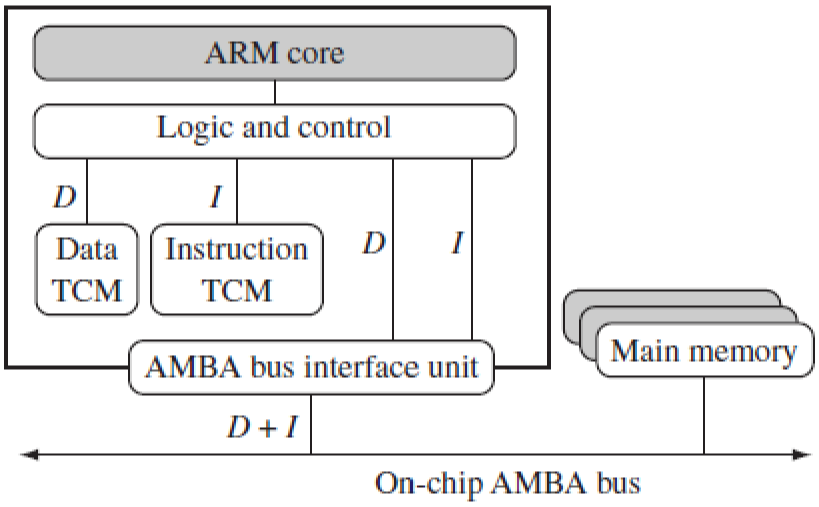


Figure: Harvard Architecture with TCMs

- ✓ A cache provides an overall increase in performance, but at the expense of predictable execution. But the real-time systems require the code execution to be deterministic— the time taken for loading and storing instructions or data must be predictable.

- ✓ This is achieved using a form of memory called *tightly coupled memory (TCM)*. TCM is fast SRAM located close to the core and guarantees the clock cycles required to fetch instructions or data.
- ✓ TCMs appear as memory in the address map and can be accessed as fast memory.

By combining both technologies, ARM processors can have both improved performance and predictable real-time response. The following Figure shows an example core with a combination of caches and TCMs.

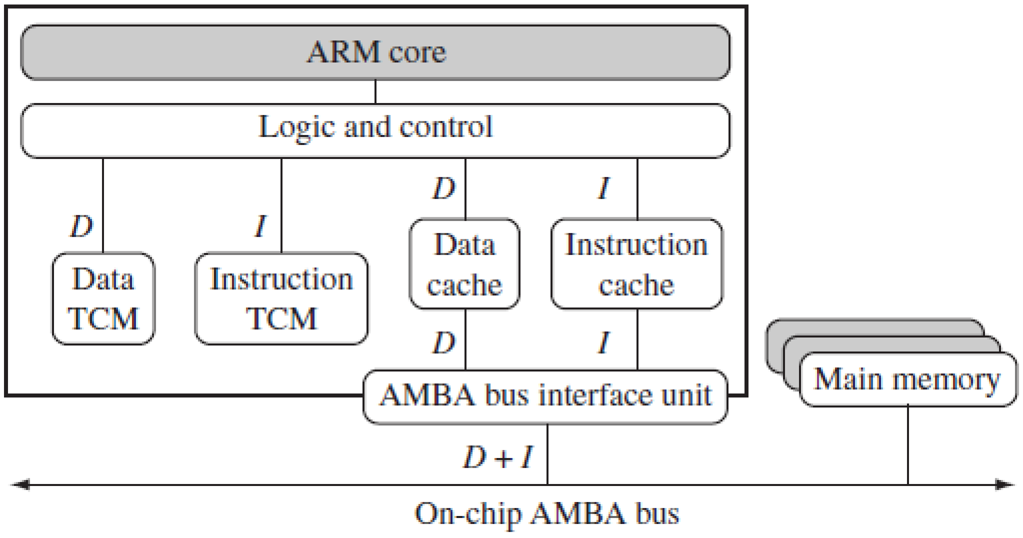


Figure: Harvard Architecture with Caches and TCMs

Memory Management:

- ✓ Embedded systems often use multiple memory devices. It is usually necessary to have a method to organize these devices and protect the system from applications trying to make inappropriate accesses to hardware. This is achieved with the assistance of memory management hardware.
- ✓ ARM cores have *three different types of memory management hardware*—
    - o no extensions providing no protection
    - o a memory protection unit (MPU) providing limited protection
    - o a memory management unit (MMU) providing full protection
- ✓ ***Non protected memory*** is fixed and provides very little flexibility. It is normally used for small, simple embedded systems that require no protection from rogue applications.
- ✓ ***MPU***s employ a simple system that uses a limited number of memory regions. These regions are controlled with a set of special coprocessor registers, and each region is defined with specific access permissions. This type of memory management is used for systems that require memory protection but don't have a complex memory map.

- ✓ ***MMU***s are the most comprehensive memory management hardware available on the ARM. The MMU uses a set of translation tables to provide fine-grained control over memory. These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions. MMUs are designed for more sophisticated platform operating systems that support multitasking.

Coprocessors:

- ✓ Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface.
- ✓ The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface.
  - For example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.
- ✓ The coprocessor can also extend the instruction set by providing a specialized group of new instructions.
  - For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations.
- ✓ These new instructions are processed in the decode stage of the ARM pipeline.
  - If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor.
  - If the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

By: MAHESH PRASANNA K.,

DEPT. OF CSE, VCET.

____________*********____________

****

MODULE – 5

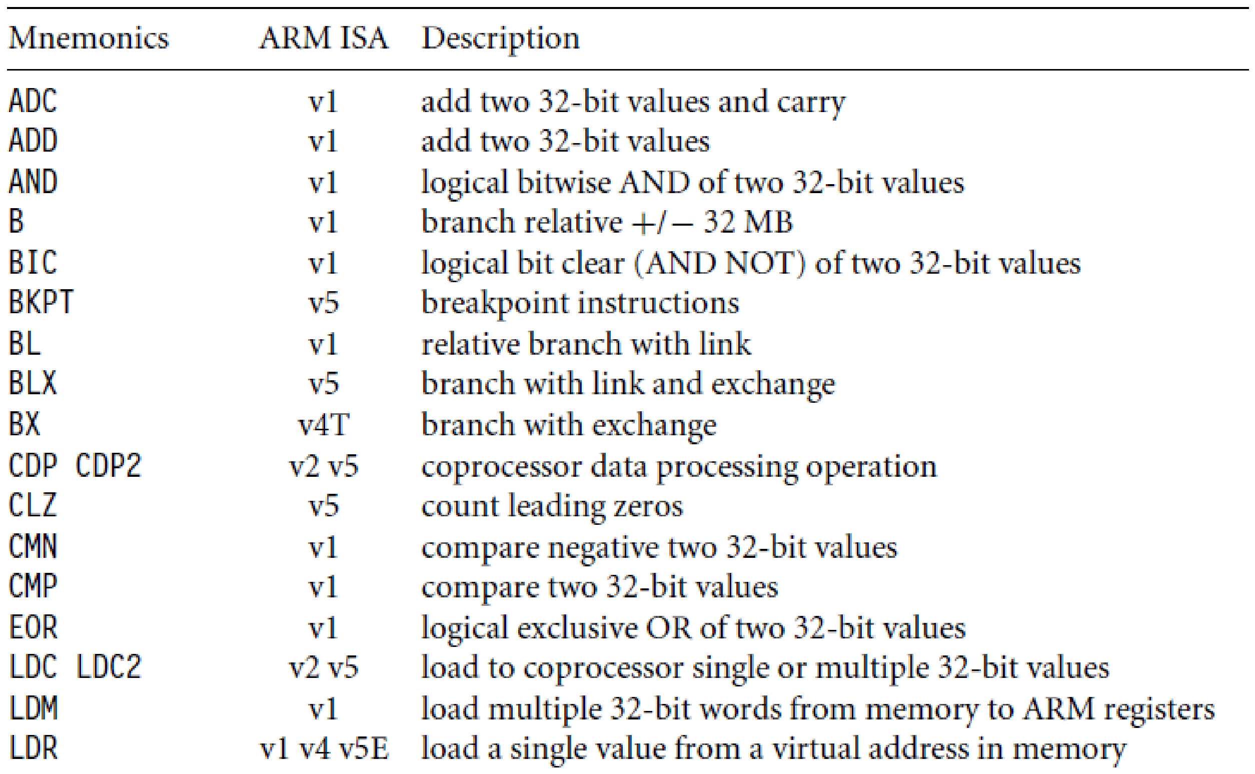
INTRODUCTION TO THE ARM INSTRUCTION SET

INTRODUCTION TO THE ARM INSTRUCTION SET

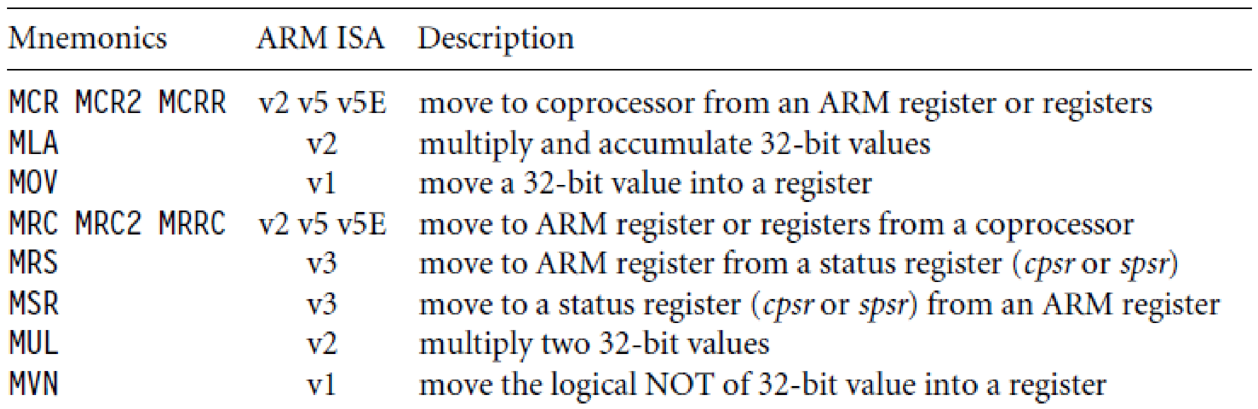
Different ARM architecture revisions support different instructions. However, new revisions usually add instructions and remain backwardly compatible. Code you write for architecture ARMv4T should execute on an ARMv5TE processor.

The following Table provides a complete list of ARM instructions available in the *ARMv5E instruction set architecture* (*ISA*). This ISA includes all the core ARM instructions as well as some of the newer features in the ARM instruction set.

Table: ARM Instruction Set



Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory



Mnemonics	ARM ISA	Description
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register

Mnemonics	ARM ISA	Description
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions $((16 \times 16) + 32 = 32\text{-bit})$
SMLAL	v3M	signed multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
SMLAL*xy*	v5E	signed multiply accumulate long $((16 \times 16) + 64 = 64\text{-bit})$
SMLAW*y*	v5E	signed multiply accumulate instruction $(((32 \times 16) \gg 16) + 32 = 32\text{-bit})$
SMULL	v3M	signed multiply long $(32 \times 32 = 64\text{-bit})$

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions $(16 \times 16 = 32\text{-bit})$
SMULW*y*	v5E	signed multiply instruction $((32 \times 16) \gg 16 = 32\text{-bit})$
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long $((32 \times 32) + 64 = 64\text{-bit})$
UMULL	v3M	unsigned multiply long $(32 \times 32 = 64\text{-bit})$

In the following sections, the hexadecimal numbers are represented with the prefix *0x* and binary numbers with the prefix *0b*. The examples follow this format:

PRE *<pre-conditions>*

<instruction/s>

POST *<post-conditions>*

In the pre- and post-conditions, memory is denoted as

mem<data_size>[address]

This refers to *data_size* bits of memory starting at the given byte address. For example, *mem32[1024]* is the 32-bit value starting at address 1 KB.

ARM instructions process data held in registers and memory is accessed only with load and store instructions.

ARM instructions commonly take two or three operands. For instance, the ADD instruction below adds the two values stored in registers *r1* and *r2* (the source registers). It writes the result to register *r3* (the destination register).

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
`ADD r3, r1, r2`	*r3*	*r1*	*r2*

ARM instructions classified as—data processing instructions, branch instructions, load-store instructions, software interrupt instruction, and program status register instructions.

DATA PROCESSING INSTRUCTIONS:

The data processing instructions manipulate data within registers. They are—

- ✓ move instructions, arithmetic instructions, logical instructions, comparison instructions, and multiply instructions.

Most data processing instructions can process one of their operands using the barrel shifter.

If you use the S suffix on a data processing instruction, then it updates the flags in the *cpsr*.

Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.

- The *C* flag is set from the result of the barrel shift as the last bit shifted out.
- The *N* flag is set to bit 31 of the result.
- The *Z* flag is set if the result is zero.

MOVe Instructions:

Move instruction copies *N* into a destination register *Rd*, where *N* is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

```
Syntax: <instruction>{<cond>}{S} Rd, N
```

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

Example: This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value *5*, and overwriting the value *8* in register *r7*.

PRE *r5 = 5*

r7 = 8

MOV r7, r5 ; let r7 = r5

POST *r5 = 5*

r7 = 5

Barrel Shifter:

In above Example, we showed a MOV instruction where *N* is a simple register. But *N* can be more than just a register or immediate value; it can also be a register *Rm* that has been preprocessed by the barrel shifter prior to being used by a data processing instruction.

- ✓ Data processing instructions are processed within the arithmetic logic unit (ALU).
- ✓ A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- ✓ Pre-processing or shift occurs within the cycle time of the instruction.
    - o This shift increases the power and flexibility of many data processing operations.
    - o This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- ✓ There are data processing instructions that do not use the barrel shift, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.

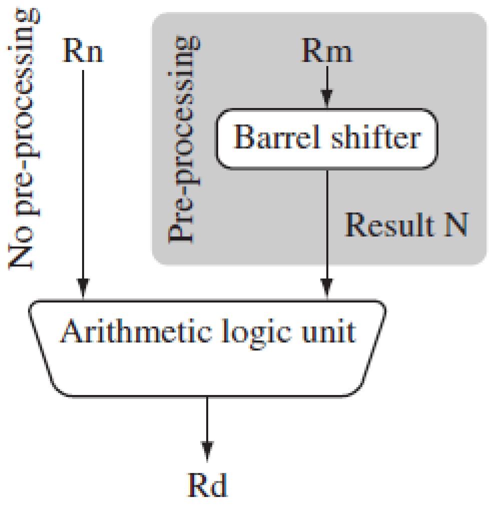


Figure: Barrel Shifter and ALU

- ✓ Figure shows the data flow between the ALU and the barrel shifter.
- ✓ Register *Rn* enters the ALU without any pre- processing of registers.
- ✓ We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register. This is the same as applying the standard *C* language shift operator « to the register.

- ✓ The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in the following Table.

Table: Barrel Shifter Operations

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	$(\text{unsigned})x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	$(\text{signed})x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	$((\text{unsigned})x \gg y) \mid (x \ll (32 - y))$	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

- ✓ The five different shift operations that you can use within the barrel shifter are summarized in the above Table.

```
PRE     r5 = 5
        r7 = 8
MOV     r7, r5, LSL #2    ; let r7 = r5*4 = (r5 << 2)
POST    r5 = 5
        r7 = 20
```

- ✓ The above example multiplies register *r5* by four and then places the result into register *r7*.
- ✓ The following Figure illustrates a logical shift left by one.

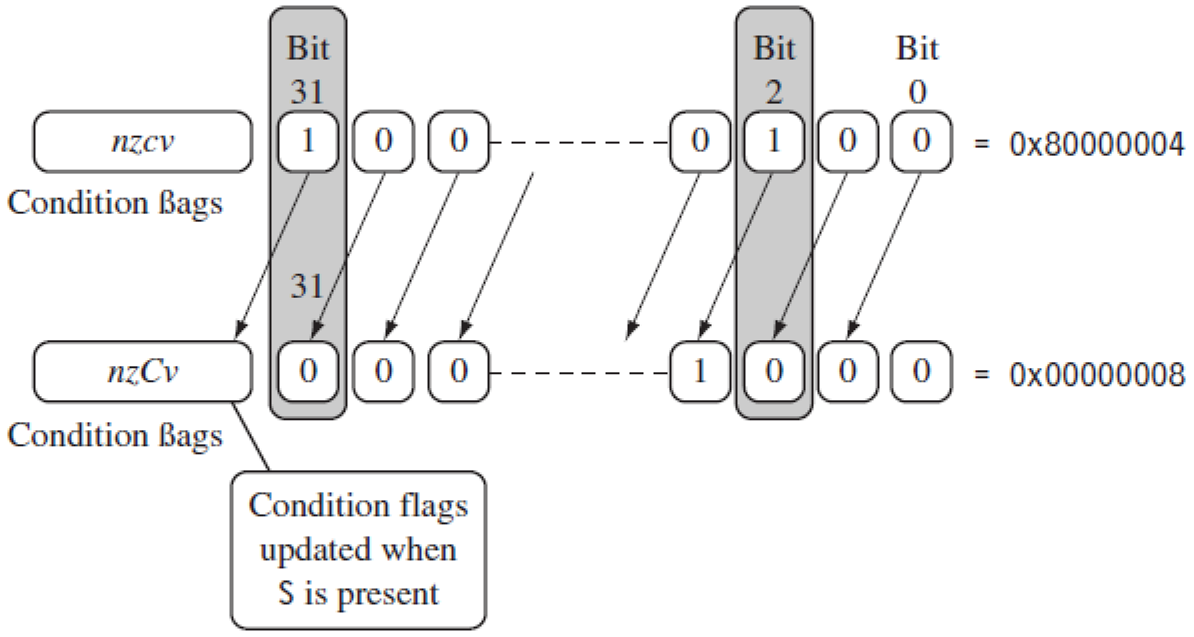


Figure: Logical Shift Left by One

- ✓ For example, the contents of bit *0* are shifted to bit *1*. Bit 0 is cleared. The C flag is updated with the last bit shifted out of the register. This is bit (32 - y) of the original value, where *y* is the shift amount. When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

Example: This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the C flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

```
PRE   cpsr = nzcvqiFt_USER
      r0 = 0x00000000
      r1 = 0x80000004
MOVS  r0, r1, LSL #1
POST  cpsr = nzCvqiFt_USER
      r0 = 0x00000008
      r1 = 0x80000004
```

The following Table lists the syntax for the different barrel shift operations available on data processing instructions. The second operand *N* can be an immediate constant preceded by #, a register value *Rm*, or the value of *Rm* processed by a shift.

Table: Barrel Shifter Operation Syntax for data Processing Instructions

N shift operations	Syntax
Immediate	#immediate
Register	Rm
Logical shift left by immediate	Rm, LSL #shift_imm
Logical shift left by register	Rm, LSL Rs
Logical shift right by immediate	Rm, LSR #shift_imm
Logical shift right with register	Rm, LSR Rs
Arithmetic shift right by immediate	Rm, ASR #shift_imm
Arithmetic shift right by register	Rm, ASR Rs
Rotate right by immediate	Rm, ROR #shift_imm
Rotate right by register	Rm, ROR Rs
Rotate right with extend	Rm, RRX

Arithmetic Instructions:

The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn -$!(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N -$!(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation.

Example: The following simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

```
PRE     r0  =  0x00000000
        r1 = 0x00000002
        r2 = 0x00000001
SUB r0, r1, r2
POST    r0 =  0x00000001
```

Example: The following reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

```
PRE     r0 = 0x00000000
        r1 = 0x00000077
RSB r0, r1, #0   ; Rd  =  0x0  - r1
POST    r0 = -r1 = 0xffffff89
```

Example: The SUBS instruction is useful for decrementing loop counters. In this example, we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the ZC flags being set.

```
PRE     cpsr = nzcvqiFt_USER
        r1 = 0x00000001
SUBS r1, r1, #1
POST    cpsr  =  nZCvqiFt_USER
        r1 = 0x00000000
```

Using the Barrel Shifter with Arithmetic Instructions:

The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set. The following Example illustrates the use of the inline barrel shifter with an arithmetic instruction. The instruction multiplies the value stored in register *r1* by three.

Example: Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000005
ADD     r0, r1, r1, LSL #1
POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Logical Instructions:

Logical instructions perform bitwise logical operations on the two source registers.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

Example: This example shows a logical OR operation between registers *r1* and *r2*. Register *r0* holds the result.

PRE *r0 = 0x00000000*

r1 = 0x02040608

r2 = 0x10305070

ORR r0, r1, r2

POST *r0 = 0x12345678*

Example: This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

PRE *r1 = 0b1111*

r2 = 0b0101

BIC r0, r1, r2

POST *r0 = 0b1010*

This is equivalent to – *Rd = Rn* ***AND NOT*** *(N)*

In this example, register *r2* contains a binary pattern where every binary *1* in *r2* clears a corresponding bit location in register *r1*.

This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.

NOTE: The logical instructions update the *cpsr* flags only if the S suffix is present. These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

Comparison Instructions:

- ✓ The comparison instructions are used to compare or test a register with a 32-bit value.
- ✓ They update the *cpsr* flag bits according to the result, but do not affect other registers.

- ✓ After the bits have been set, the information can then be used to change program flow by using conditional execution.
- ✓ It is not required to apply the S suffix for comparison instructions to update the flags.

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of Rn ^ N
TST	test bits of a 32-bit value	flags set as a result of Rn & N

N is the result of the shifter operation.

Example: This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *Z* flag prior to execution is *0* and is represented by a lowercase *z*. After execution the *Z* flag changes to *1* or an uppercase *Z*. This change indicates equality.

PRE *cpsr = nzcvqiFt_USER*

r0 = 4

r9 = 4

CMP *r0, r9*

POST *cpsr = nZcvqiFt_USER*

- ✓ The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- ✓ For each, the results are discarded but the condition bits are updated in the *cpsr*.
- ✓ It is important to understand that comparison instructions only modify the condition flags of the *cpsr* and do not affect the registers being compared.

Multiply Instructions:

The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register.

The long multiplies accumulate onto a pair of registers representing a 64-bit value. The final result is placed in a destination register or a pair of registers.

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm^* Rs) + Rn$
MUL	multiply	$Rd = Rm^* Rs$

Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*.

Example: This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value *2*, and *r2* is equal to *2*. The result, *4*, is then placed into register *r0*.

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000002
MUL r0, r1, r2  ; r0 = r1*r2
POST    r0 = 0x00000004
        r1 = 0x00000002
        r2 = 0x00000002
```

The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result. The result is too large to fit a single 32-bit register so the result is placed in two registers labeled *RdLo* and *RdHi*. *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result. The following shows an example of a long unsigned multiply instruction.

Example: The instruction multiplies registers *r2* and *r3* and places the result into register *r0* and *r1*. Register *r0* contains the lower 32 bits, and register *r1* contains the higher 32 bits of the 64-bit result.

```
PRE     r0 = 0x00000000
        r1 = 0x00000000
        r2 = 0xf0000002
        r3 = 0x00000002
UMULL r0, r1, r2, r3        ; [r1,r0] = r2*r3
POST    r0 = 0xe0000004     ; = RdLo
        r1 = 0x00000001     ; = RdHi
```

BRANCH INSTRUCTIONS:

A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, if-then-else structures, and loops.

The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

- ✓ The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction.
- ✓ *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

Example: This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B       forward
        ADD r1, r2, #4
        ADD r0, r6, #2
        ADD r3, r7, #4
forward
        SUB r1, r2, #4
 --------------------------------------------------
backward
        ADD r1, r2, #4
        SUB r1, r2, #4
        ADD    r4, r6, r7
        B       backward
```

In this example, *forward* and *backward* are the labels. The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

- ✓ The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call.

Example: This example shows a simple fragment of code that, branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

```
        BL      subroutine      ; branch to subroutine
        CMP r1, #5              ; compare r1 with 5
        MOVEQ r1, #0            ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV pc, lr              ; return by moving  pc = lr
```

- ✓ The branch exchange (BX) and branch exchange with link (BLX) are the third type of branch instruction.
- ✓ The BX instruction uses an absolute address stored in register *Rm*. It is primarily used to branch to and from Thumb code. The *T* bit in the *cpsr* is updated by the least significant bit of the branch register.
- ✓ Similarly the BLX instruction updates the *T* bit of the *cpsr* with the least significant bit and additionally sets the link register with the return address.

LOAD-STORE INSTRUCTIONS:

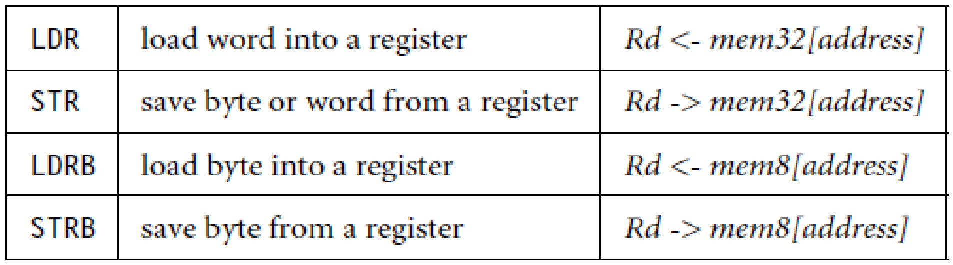
Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions: single-register transfer, multiple-register transfer, and swap.

Single-Register Transfer:

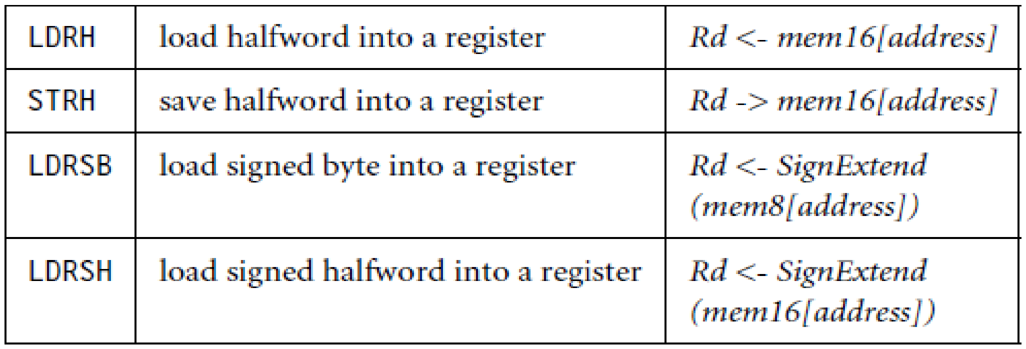
- ✓ These instructions are used for moving a single data item in and out of a register.
- ✓ The data types supported are signed and unsigned words (32-bit), half-words (16-bit), and bytes.

Here are the various load-store single-register transfer instructions.

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```



LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*



LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

- ✓ LDR and STR instructions can load and store data on a boundary alignment that is the same as the data type size being loaded or stored.
  - o For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on.

Example: This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR r0, [r1]            ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address  pointed  to by
; register r1.
;
        STR r0, [r1]            ; = STR r0, [r1, #0]
```

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes:

The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods: preindex with writeback, preindex, and postindex.

Table: Index Methods

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	`LDR r0,[r1,#4]!`
Preindex	*mem[base + offset]*	*not updated*	`LDR r0,[r1,#4]`
Postindex	*mem[base]*	*base + offset*	`LDR r0,[r1],#4`

Note: ! indicates that the instruction writes the calculated address back to the base address register.

- ✓ *Preindex with writeback* calculates an address from a base register plus address offset and then updates that address base register with the new address.
- ✓ *Preindex* offset is the same as the preindex with writeback but does not update the address base register.
    - o The preindex mode is useful for accessing an element in a data structure.
- ✓ *Postindex* only updates the address base register after the address is used.
    - o The postindex and preindex with writeback modes are useful for traversing an array.

Example:

```
PRE         r0 = 0x00000000
            r1 = 0x00090000
            mem32[0x00009000] = 0x01010101
            mem32[0x00009004] = 0x02020202
LDR r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1)     r0 = 0x02020202
            r1 = 0x00009004
LDR r0, [r1, #4]
```

Preindexing:

```
POST(2)     r0 = 0x02020202
            r1 = 0x00009000
LDR r0, [r1], #4
```

Postindexing:

```
POST(3)     r0 = 0x01010101
            r1 = 0x00009004
```

- ✓ The above Example used a preindex method. This example shows how each indexing method affects the address held in register *r1*, as well as the data loaded into register *r0*.

The addressing modes available with a particular load or store instruction depend on the instruction class. The following Table shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.

Table: Single-Register Load-Store Addressing, Word or Unsigned Byte

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	`[Rn, #+/-offset_12]`
Preindex with register offset	`[Rn, +/-Rm]`
Preindex with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]`
Preindex writeback with immediate offset	`[Rn, #+/-offset_12]!`
Preindex writeback with register offset	`[Rn, +/-Rm]!`
Preindex writeback with scaled register offset	`[Rn, +/-Rm, shift #shift_imm]!`
Immediate postindexed	`[Rn], #+/-offset_12`
Register postindex	`[Rn], +/-Rm`
Scaled register postindex	`[Rn], +/-Rm, shift #shift_imm`

- ✓ A signed offset or register is denoted by "+/-", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- ✓ Immediate means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- ✓ Register means the address is calculated using the base address register and a specific register's contents.
- ✓ Scaled means the address is calculated using the base address register and a barrel shift operation.

The following Table provides an example of the different variations of the LDR instruction.

Table: Examples of LDR Instructions using Different Addressing Modes

	Instruction	*r0* =	*r1* + =
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

The following Table shows the addressing modes available on load and store instructions using 16-bit halfword or signed byte data.

Table: Single-Register Load-Store Addressing, Halfword, Signed Halfword, Signed Byte and Doubleword

Addressing^2 mode and index method	Addressing^2 syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes.

The following Table shows the variations for STRH instructions.

Table: Variations of STRH Instructions

	Instruction	Result	*r1* + =
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Multiple-Register Transfer:

- ✓ Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction.
- ✓ The transfer occurs from a base address register *Rn* pointing into memory.
  - o Multiple-register transfer instructions are more efficient from single-register transfers for
    - ▪ moving blocks of data around memory and
    - ▪ saving and restoring context and stacks.
- ✓ Load-store multiple instructions can increase interrupt latency.
- ✓ ARM implementations do not usually interrupt instructions while they are executing.
  - o For example, on an ARM7 a load multiple instruction takes *2 + Nt* cycles, where *N* is the number of registers to load and *t* is the number of cycles required for each sequential access to memory.
- ✓ If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete.

- ✓ Compilers, such as *armcc*, provide a switch to control the maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency.

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

LDM	load multiple registers	$\{Rd\}^{*N}$ <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	$\{Rd\}^{*N}$ -> mem32[start address + 4*N] optional Rn updated

The following Table shows the different addressing modes for the load-store multiple instructions. Here *N* is the number of registers in the list of registers.

Table: Addressing Mode for Load-Store Multiple Instructions

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

- ✓ Any subset of the current bank of registers can be transferred to memory or fetched from memory.
- ✓ The base register *Rn* determines the source or destination address for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register *Rn* is followed by the ! character, similar to the single-register load-store using preindex with writeback.

Example: In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

```
PRE   mem32[0x80018] = 0x03
      mem32[0x80014] = 0x02
      mem32[0x80010] = 0x01
      r0 = 0x00080010
      r1 = 0x00000000
      r2 = 0x00000000
      r3 = 0x00000000
LDMIA r0!, {r1–r3}
POST  r0  =  0x0008001c
```

r1 = 0x00000001

r2 = 0x00000002

r3 = 0x00000003

The following Figure shows a graphical representation.

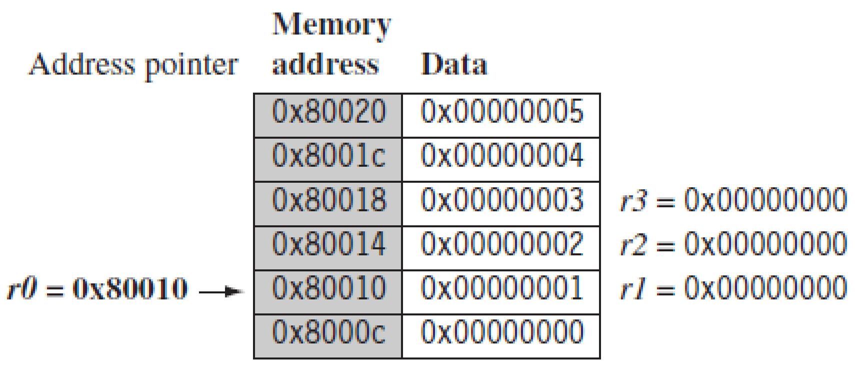


Figure: Pre-condition for LDMIA Instruction

- ✓ The base register *r0* points to memory address 0x80010 in the PRE condition.
- ✓ Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1, 2, and 3 respectively.
- ✓ After the load multiple instruction executes, registers *r1*, *r2*, and *r3* contain these values as shown in the following Figure.

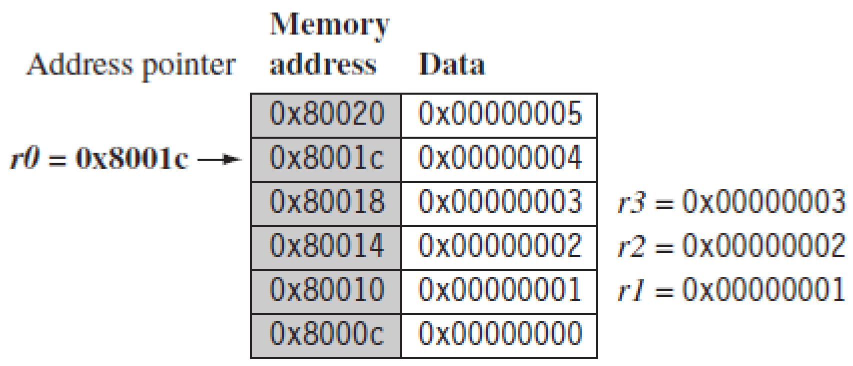


Figure: Post Condition for LDMIA Instruction

- ✓ The base register *r0* now points to memory address 0x8001c after the last loaded word.
- ✓ Now replace the LDMIA instruction with a load multiple and increment before LDMIB instruction and use the same PRE conditions.
- ✓ The first word pointed to by register *r0* is ignored and register *r1* is loaded from the next memory location as shown in the following Figure.

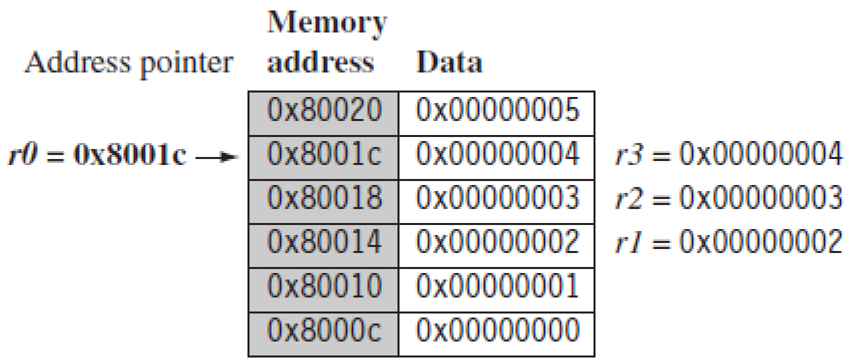


Figure: Post Condition for LDMIB Instruction

- ✓ After execution, register *r0* now points to the last loaded memory location. This is in contrast with the LDMIA example, which pointed to the next memory location.

- The decrement versions DA and DB of the load-store multiple instructions decrement the start address and then store to ascending memory locations.
- This is equivalent to descending memory but accessing the register list in reverse order.
- With the increment and decrement load multiples; you can access arrays forwards or backwards.
- They also allow for stack push and pull operations.

The following Table shows a list of load-store multiple instruction pairs.

Table: Load-Store Multiple Pairs when Base Update used

Store Multiple	**Load Multiple**
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

- If you use a store with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer.
- This is useful when you need to temporarily save a group of registers and restore them later.

Example: This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

```
PRE   r0 = 0x00009000
      r1 = 0x00000009
      r2 = 0x00000008
      r3 = 0x00000007
STMIB  r0!, {r1–r3}
MOV r1, #1
MOV r2, #2
MOV r3, #3
PRE(2) r0  =  0x0000900c
      r1 = 0x00000001
      r2 = 0x00000002
      r3 = 0x00000003
LDMDA r0!, {r1–r3}
POST   r0 = 0x00009000
      r1 = 0x00000009
      r2 = 0x00000008
      r3 = 0x00000007
```

The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

Example: We illustrate the use of the load-store multiple instructions with a block memory copy example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.

The example has two load-store multiple instructions, which use the same increment after addressing mode.

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source
       loop
; load 32 bytes from source and update r9 pointer
       LDMIA r9!, {r0–r7}
; store 32 bytes to destination and update r10 pointer
       STMIA r10!, {r0–r7}     ; and store them
; have we reached the end
       CMP    r9, r11
       BNE loop
```

- ✓ This routine relies on registers *r9*, *r10*, and *r11* being set up before the code is executed.
- ✓ Registers *r9* and *r11* determine the data to be copied, and register *r10* points to the destination in memory for the data.
- ✓ LDMIA loads the data pointed to by register *r9* into registers *r0* to *r7*. It also updates *r9* to point to the next block of data to be copied.
- ✓ STMIA copies the contents of registers *r0* to *r7* to the destination memory address pointed to by register *r10*. It also updates *r10* to point to the next destination location.

- ✓ CMP and BNE compare pointers *r9* and *r11* to check whether the end of the block copy has been reached.
- ✓ If the block copy is complete, then the routine finishes; otherwise the loop repeats with the updated values of register *r9* and *r10*.
- The BNE is the branch instruction B with a condition mnemonic NE (not equal). If the previous compare instruction sets the condition flags to not equal, the branch instruction is executed.

The following Figure shows the memory map of the block memory copy and how the routine moves through memory.

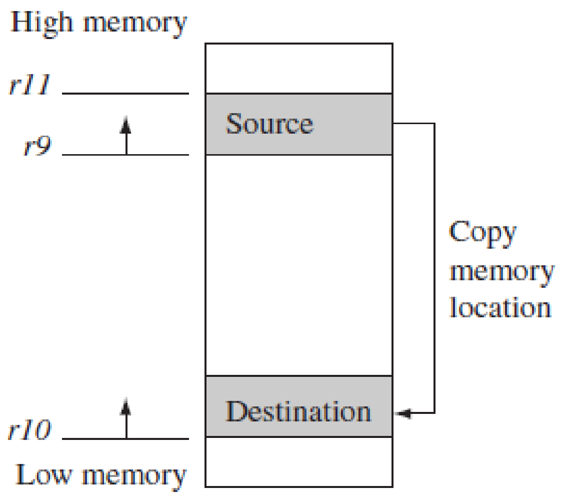


Figure: Block Memory Copy in the Memory map

Theoretically this loop can transfer 32 bytes (8 words) in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 MHz. These numbers assume a perfect memory system with fast memory.

Stock Operation: The ARM architecture uses the load-store multiple instructions to carry out stack operations.

- The *pop operation* (removing data from a stack) uses a load multiple instruction.
- The *push operation* (placing data onto the stack) uses a store multiple instruction.

- ✓ When using a stack you have to decide whether the stack will grow up or down in memory.
  - o A stack is either –
    - ▪ *ascending (A)* – stacks grow towards higher memory addresses or
    - ▪ *descending (D)* – stacks grow towards lower memory addresses.
- ✓ When you use a *full stack (F)*, the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack).

- ✓ If you use an *empty stack (E)* the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

- There are number of load-store multiple addressing mode aliases available to support stack operations (see the following Table).

Table: Addressing Methods for Stack Operations

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

- Next to the *pop* column is the actual load multiple instruction equivalent.
  - For example, a full ascending stack would have the notation FA appended to the load multiple instruction—LDMFA. This would be translated into an LDMDA instruction.
- ARM has specified an *ARM-Thumb Procedure Call Standard (ATPCS)* that defines how routines are called and how registers are allocated. In the ATPCS, stacks are defined as being full descending stacks. Thus, the LDMFD and STMFD instructions provide the *pop* and *push* functions, respectively.

Example: The STMFD instruction pushes registers onto the stack, updating the *sp*. The following Figure shows a *push* onto a full descending stack.

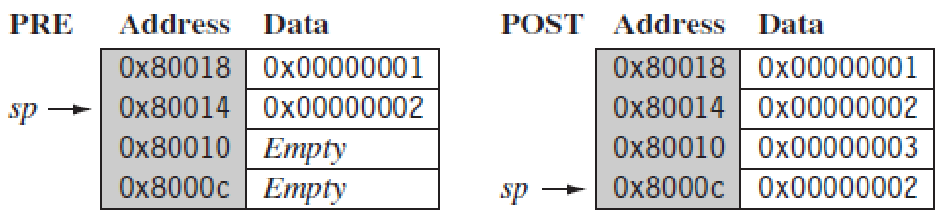


Figure: STMFD Instruction – Full Stack *push* Operation

You can see that when the stack grows the stack pointer points to the last full entry in the stack.

PRE *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080014

STMFD sp!, {r1, r4}

POST *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x0008000c

Example: The following Figure shows a *push* operation on an empty stack using the STMED instruction.

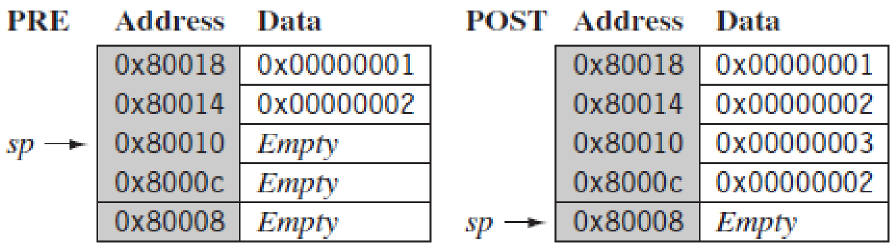


Figure: STMED Instruction – Empty Stack *push* Operation

The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

PRE *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080010

STMED sp!, {r1, r4}

POST *r1 = 0x00000002*

r4 = 0x00000003

sp = 0x00080008

- ✓ When handling a checked stack there are three attributes that need to be preserved: the stack base, the stack pointer, and the stack limit.
- ✓ The stack base is the starting address of the stack in memory.
- ✓ The stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously points to the top of stack. If the stack pointer passes the stack limit, then a stack overflow error has occurred.
- ✓ Here is a small piece of code that checks for stack overflow errors for a descending stack:

```
; check for stack overflow
        SUB sp, sp, #size
        CMP sp, r10
        BLLO _stack_overflow        ; condition
```

- ATPCS defines register *r10* as the stack limit or *sl*. This is optional since it is only used when stack checking is enabled.
- The BLLO instruction is a branch with link instruction plus the condition mnemonic LO.
    - If *sp* is less than register *r10* after the new items are pushed onto the stack, then *stack overflow* error has occurred.
    - If the stack pointer goes back past the stack base, then a *stack underflow* error has occurred.

Swap Instruction:

The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register.

This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Swap cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete. Also, swap instruction allows for both a word and a byte swap.

Example: The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

```
PRE     mem32[0x9000] = 0x12345678
        r0  =  0x00000000
        r1 = 0x11112222
        r2 = 0x00009000
SWP r0, r1, [r2]
POST    mem32[0x9000] = 0x11112222
        r0 = 0x12345678
        r1 = 0x11112222
        r2 = 0x00009000
```

Example: This example shows a simple data guard that can be used to protect data from being written by another task. The SWP instruction "holds the bus" until the transaction is complete.

```
 spin
        MOV r1, =semaphore
        MOV r2, #1
        SWP r3, r2, [r1]        ; hold the bus until complete
        CMP r3, #1
        BEQ     spin
```

The address pointed to by the semaphore either contains the value *0* or *1*. When the semaphore equals *1*, then the service in question is being used by another process. The routine will continue to loop around until the service is released by the other process—in other words, when the semaphore address location contains the value *0*.

SOFTWARE INTERRUPT INSTRUCTION:

A *software interrupt instruction (SWI)* causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

```
Syntax: SWI{<cond>} SWI_number
```

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

When the processor executes an SWI instruction, it sets the program counter *pc* to the offset *0x8* in the vector table. The instruction also forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature.

Example: Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

```
PRE     cpsr = nzcVqift_USER
        pc = 0x00008000
        lr = 0x003fffff            ;lr = r14
        r0 = 0x12
0x00008000    SWI    0x123456
POST    cpsr = nzcVqIft_SVC
        spsr = nzcVqift_USER
        pc = 0x00000008
        lr = 0x00008004
        r0 = 0x12
```

Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers. In this example, register *r0* is used to pass the parameter *0x12*. The return values are also passed back via registers.

Code called the ***SWI handler*** is required to process the SWI call. The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.

The SWI number is determined by

SWI_Number = <SWI instruction> ***AND NOT*** *(0xff000000)*

Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

Example: This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*.

You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
; Store registers r0-r12 and the link register
        STMFD sp!, {r0–r12, lr}
; Read the SWI instruction
        LDR r10, [lr, #–4]
; Mask off top 8 bits
        BIC r10, r10, #0xff000000
; r10 - contains the SWI number
        BL service_routine
; return from SWI handler
        LDMFD sp!, {r0–r12, pc}ˆ
```

The number in register *r10* is then used by the SWI handler to call the appropriate SWI service routine.

PROGRAM STATUS REGISTER INSTRUCTIONS:

The ARM instruction set provides two instructions to directly control a *program status register (psr)*.

- ✓ The *MRS instruction* transfers the contents of either the *cpsr* or *spsr* into a register.
- ✓ The *MSR instruction* transfers the contents of a register into the *cpsr* or *spsr*.

Together these instructions are used to read and write the *cpsr* and *spsr*.

In the syntax we can see a *label* called fields. This can be any combination of *control (c)*, *extension (x)*, *status (s)*, and *flags (f)*.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

MRS	copy program status register to a general-purpose register	*Rd* = *psr*
MSR	move a general-purpose register to a program status register	*psr[field]* = *Rm*
MSR	move an immediate value to a program status register	*psr[field]* = *immediate*

These fields relate to particular byte regions in a *psr*, as shown in the following Figure.

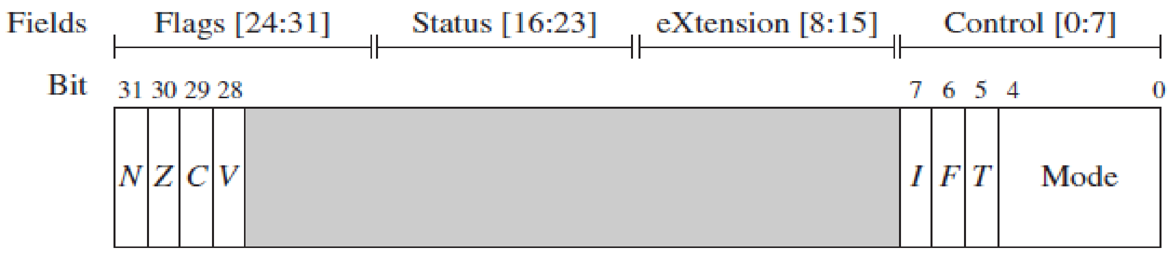


Figure: *psr* Byte Fields

The *c* field controls the interrupt masks, Thumb state, and processor mode.

The following Example shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

Example: The MSR first copies the *cpsr* into register *r1*. The BIC instruction clears bit *7* of *r1*. Register *r1* is then copied back into the *cpsr*, which enables IRQ interrupts. You can see from this example that this code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

```
PRE     cpsr = nzcvqIFt_SVC
MRS     r1,   cpsr
BIC r1, r1, #0x80        ; 0b01000000
MSR cpsr_c, r1
POST    cpsr = nzcvqiFt_SVC
```

This example is in SVC mode. In user mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

Coprocessor Instructions:

Coprocessor instructions are used to extend the instruction set.

- ✓ A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management.
- ✓ The coprocessor instructions include data processing, register transfer, and memory transfer instructions.
- ✓ Note that these instructions are only used by cores with a coprocessor.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

- ✓ In the syntax of the coprocessor instructions,
    - o The *cp* field represents the coprocessor number between *p0* and *p15*
    - o The *opcode* fields describe the operation to take place on the coprocessor.
    - o The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor.
- ✓ The coprocessor operations and registers depend on the specific coprocessor you are using.
- ✓ *Coprocessor 15 (CP15)* is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

Example: This example shows a *CP15* register being copied into a general-purpose register.

; transferring the contents of CP15 register c0 to register r10

MRC p15, 0, r10, c0, c0, 0

Here *CP15 register-0* contains the processor identification number. This register is copied into the general-purpose register *r10*.

LOADING CONSTANTS:

You might have noticed that there is no ARM instruction to move a 32-bit constant into a register. Since ARM instructions are 32 bits in size, they obviously cannot specify a general 32-bit constant.

To aid programming there are two pseudo-instructions to move a 32-bit value into a register.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

- The first pseudo-instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.
- The second pseudo-instruction writes a relative address into a register, which will be encoded using a pc-relative expression.

Example: This example shows an LDR instruction loading a 32-bit constant *0xff00ffff* into register *r0*.

LDR r0, [pc, #constant_number-8-{PC}]

:

constant_number

DCD 0xff00ffff

This example involves a memory access to load the constant, which can be expensive for time-critical routines.

The following Example shows an alternative method to load the same constant into register *r0* by using an MVN instruction.

Example: Loading the constant *0xff00ffff* using an MVN.

PRE *none...*

MVN r0, #0x00ff0000

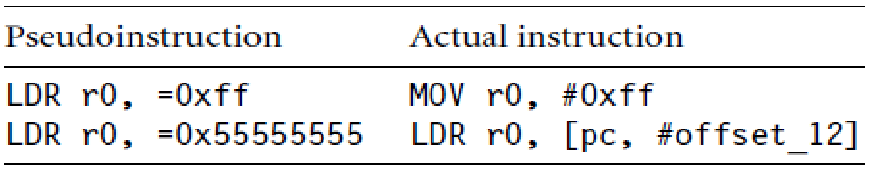
POST *r0 = 0xff00ffff*

As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load.

The LDR pseudo-instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a literal pool—a data area embedded within the code.

The following Table shows two pseudo-code conversions.

Table: LDR pseudo-instruction Conversion



Pseudoinstruction	Actual instruction
`LDR r0, =0xff`	`MOV r0, #0xff`
`LDR r0, =0x55555555`	`LDR r0, [pc, #offset_12]`

The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load.

Another useful pseudo-instruction is the ADR instruction, or address relative. This instruction places the address of the given label into register *Rd*, using a *pc*-relative add or subtract.

By: **MAHESH PRASANNA K.,**

DEPT. OF CSE, VCET.

____________*********____________

****