CBCS SCHEME

USN 1CR16CS103

15CS44

Fourth Semester B.E. Degree Examination, June/July 2018
Microprocessors and Microcontrollers

Time: 3 hrs.

Max. Marks: 80

Note: Answer any FIVE full questions, choosing one full question from each module.

Module-1

1 a. What is a microprocessor? With a neat diagram explain the internal block diagram of 8086 microprocessor along with functions of each block and registers. (10 Marks)

b. What is an addressing mode? List the addressing modes of 8086 μp with one example each (any six modes). (06 Marks)

OR

2 a. What are the assembler directives? Explain the following assembler directives:
(i) DB (ii) Assume (iii) OFFSET (iv) PTR (04 Marks)

b. What is a flag and flag register? Explain the format of flag register with a suitable example. (06 Marks)

c. Write an assembly level program (ALP) to sort a given set of 'n' 16-bit numbers in descending order. Using Bubble sort algorithm to sort given elements. (06 Marks)

Module-2

3 a. Explain the following instructions with a suitable example:
(i) MOV (ii) PUSH (iii) LEA (iv) SHR
(v) ROL (vi) CMP (vii) DAA (viii) TEST (08 Marks)

b. What is an interrupt? Explain various types with an interrupt vector table. (08 Marks)

OR

4 a. Explain the following instructions with a suitable example:
(i) XLAT (ii) RCR (iii) AAA (iv) MUL
(v) DIV (vi) LOOP (vii) ROL (viii) OR (08 Marks)

b. Explain rotate instructions with an example. (08 Marks)

Module-3

5 a. With example, explain how to identify overflow and underflow using flags in a flag register for performing an arithmetic operation on 16-bit numbers. (08 Marks)

b. Explain 74138 decoder configuration to enable the memory address 08000H to 0FFFFH to connect four 8K RAMS. (08 Marks)

OR

6 a. Briefly explain the control word format of 8255 IC in I/O mode and BSR mode. Find the control word if P_A = out, P_B = in, $P_{C0} - P_{C3}$ = in and $P_{C4} - P_{C7}$ = out. Use port address of 300H – 303H for the 8255 chip. Then get data from port A and send it to port B. (08 Marks)

b. Write an assembly level program (ALP) to read P_B and check number of one's in a 8-bit data as P_A and display FFh on P_A if it is even parity else 00h on Port A (P_A) if it is an odd parity. (08 Marks)

Module-4

7 a. Compare CISC with RISC. (05 Marks)
b. Explain registers used under various modes. (05 Marks)
c. Explain ARM core data flow model with a neat diagram. (06 Marks)

OR

8 a. Explain the architecture of a typical embedded device based in ARM core with a neat diagram. (08 Marks)
b. Explain the various fields in the current program status register. (08 Marks)

Module-5

9 a. Explain the following instructions of ARM processor with suitable example:
(i) MVN (ii) RSB (iii) ORR (iv) MLA
(v) SMULL (vi) LDR (vii) SWP (viii) SWPB (08 Marks)
b. Explain various formats of ADD instructions based on operands of ARM7 processor. (04 Marks)
c. If $r_5 = 5$, $r_7 = 8$ and using the following instruction, write values of r_5, r_7 after execution
MOV r_7, r_5, LSL $\neq 2$ (04 Marks)

OR

10 a. Explain software interrupt instruction of ARM processor. (06 Marks)
b. Explain various types of SWAP instructions with syntax and example. (06 Marks)
c. What are the silent features of ARM instruction set? (04 Marks)

* * * * *

Solutions of Microprocessor and Microcontrollers (15CS44) - July2018

Module 1

1a. What is microprocessor? With a neat diagram explain the internal block diagram of 8086 microprocessor along with functions of each block and registers.

Ans: Microprocessor is an integrated circuit used in computer for computations.

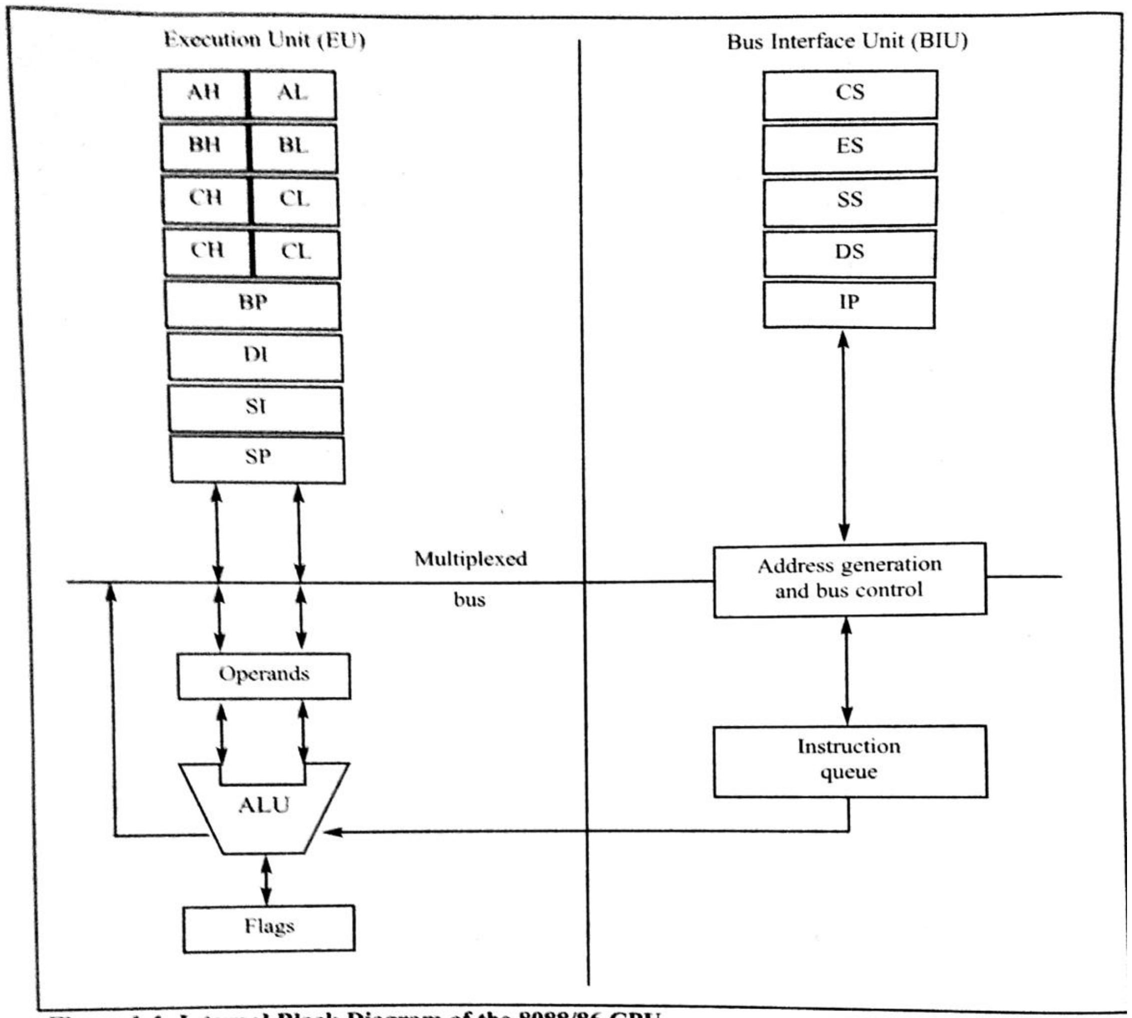


Figure 1-1. Internal Block Diagram of the 8088/86 CPU
(Reprinted by permission of Intel Corporation, Copyright Intel Corp. 1989)

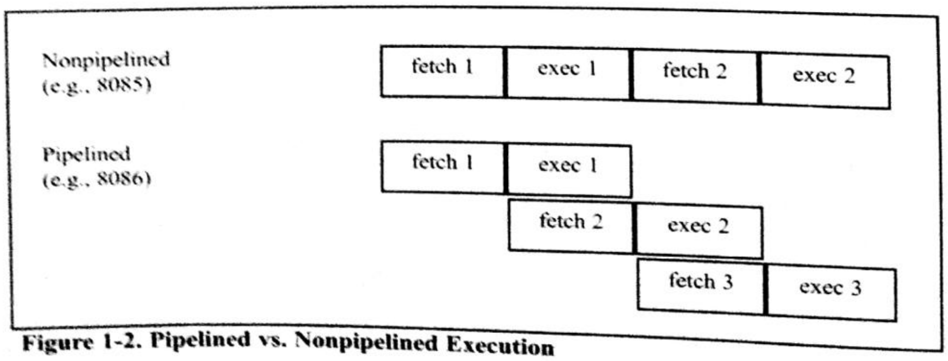
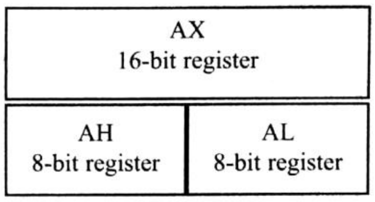


Figure 1-2. Pipelined vs. Nonpipelined Execution

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) and 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 (see Figure 1-1). 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. That information could be one or two bytes of data to be processed or the address of data. The registers of the 8088/86 fall into the six categories outlined in Table 1-4. The general-purpose registers in 8088/86 microprocessors can be accessed as either 16-bit or 8-bit registers. All other registers can be accessed only as the full 16 bits. In the 8088/86, data types are either 8 or 16 bits. To access 12-bit data, for example, a 16-bit register must be used with the highest 4 bits set to 0. The bits of a register are numbered in descending order, as shown below.

8-bit register:

D7	D6	D5	D4	D3	D2	D1	D0

16-bit register:

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Different registers in the 8088/86 are used for different functions, and since some instructions use only specific registers to perform their tasks, the use of registers will be described in the context of instructions and their application in a given program. The first letter of each general register indicates its use. AX is used for the accumulator, BX as a base addressing register, CX as a counter in loop operations, and DX to point to data in I/O operations. Table 1-4 lists the registers of the 8088/86/286.

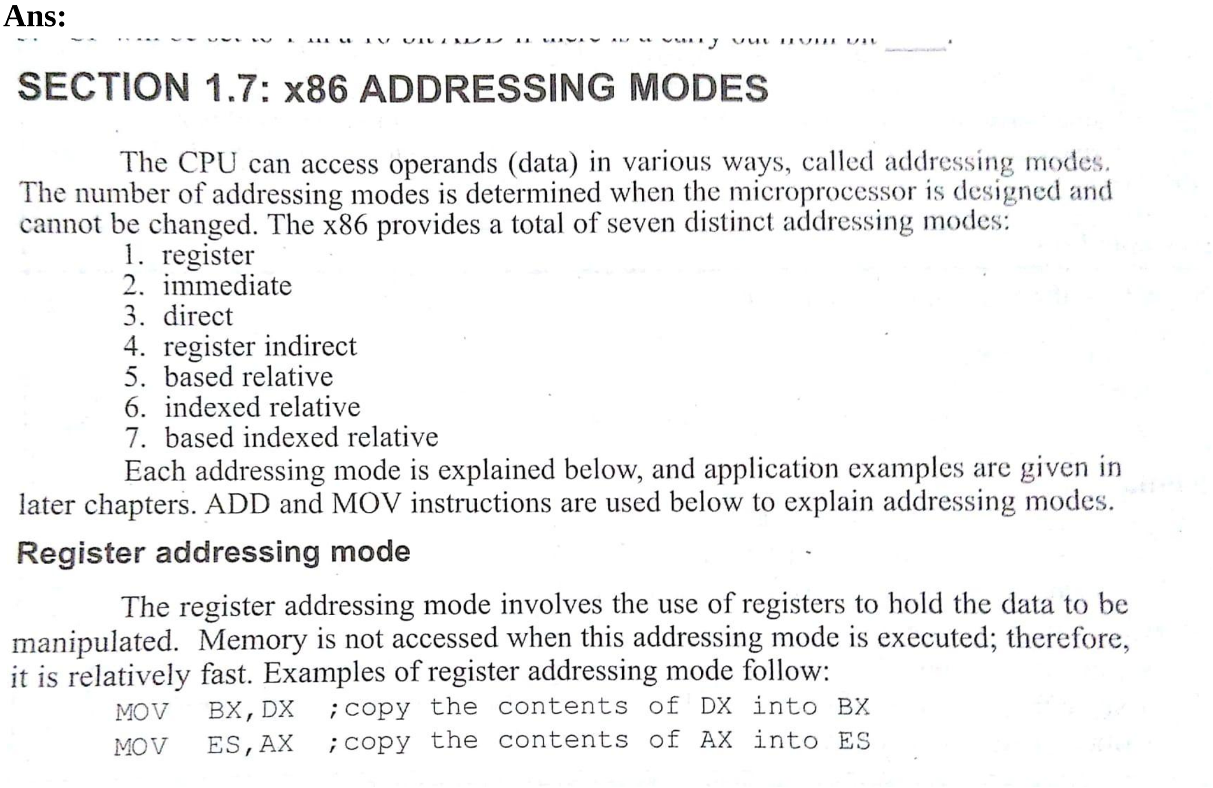
Table 1-4: Registers of the 8088/86/286 by Category

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, 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)

Note: The general registers can be accessed as the full 16 bits (such as AX), or as the high byte only (AH) or low byte only (AL).

1b. What is an addressing mode ? List the addressing modes of 8086 microprocessor with one example each(any six modes).

Ans:



SECTION 1.7: 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

Each addressing mode is explained below, and application examples are given in later chapters. ADD and MOV instructions are used below to explain addressing modes.

Register addressing mode

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. Examples of register addressing mode follow:

```
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
```

It should be noted that the source and destination registers must match in size. In other words, coding "MOV CL,AX" will give an error, since the source is a 16-bit register and the destination is an 8-bit register.

Immediate addressing mode

In the immediate addressing mode, the source operand is a constant. 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. However, in programming it has limited use. Immediate addressing mode can be used to load information into any of the registers except the segment registers and flag registers. Examples:

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

To move information to the segment registers, the data must first be moved to a general-purpose register and then to the segment register. Example:

```
MOV   AX,2550H
MOV   DS,AX
MOV   DS,0123H ;illegal! cannot move data into segment reg.
```

In the first two addressing modes, the operands are either inside the microprocessor or tagged along with the instruction. In most programs, the data to be processed is often in some memory location outside the CPU. There are many ways of accessing the data in the data segment. The following describes those different methods.

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, 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:

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

In this case the physical address is calculated by combining the contents of offset location 2400 with DS, the data segment register. 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. See Example 1-15.

Example 1-15

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.

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, DI, 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. For example:

```
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
```

The examples above moved byte-sized data. Example 1-16 shows 16-bit data.

Example 1-16

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.

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. For example:

```
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]". Again the low address contents will go into CL and the high address contents into CH. In the case of the BP register,

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

Again, alternative codings are "MOV AL,[BP+5]" or "MOV AL,5[BP]". A brief mention should be made of the terminology *effective address* used in Intel literature. In "MOV AL,[BP]+5", BP+5 is called the effective address since the fifth 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.

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. Examples:

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

Example 1-17 gives further examples of indexed relative addressing mode.

Example 1-17

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)

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. Examples:

```
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; for example, the last example could have been written in either of the following two ways:

```
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.

Table 1-5: Offset Registers for Various Segments

Segment register:	CS	DS	ES	SS
Offset register(s):	IP	SI, DI, BX	SI, DI, BX	SP, BP

In many of the examples above, the MOV instruction was used for the sake of clarity, even though one can use any instruction as long as that instruction supports the addressing mode. For example, the instruction "`ADD DL,[BX]`" would add the contents of the memory location pointed at by DS:BX to the contents of register DL.

Segment overrides

Table 1-5 summarizes the offset registers that can be used with the four segment registers. The x86 CPU allows the program to override the default segment and use any segment register. To do that, 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, as was shown earlier since DS is the default segment for pointer 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. Extensive use of all these

2a. What are the assembler directives? Explain the following directives.

Keywords which gives directions to assembler are called assembler directives. These are not converted to machine code.

i.DB (DEFINE BYTE)

The DB directive is used to declare a byte type variable, or a set aside one or more storage locations of type byte in memory.

PRICES DB 49H, 98H, 29H Declare array of 3 bytes named PRICE and initialize them with specified values.

ii.ASSUME

The ASSUME directive is used tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE. The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

iii. OFFSET

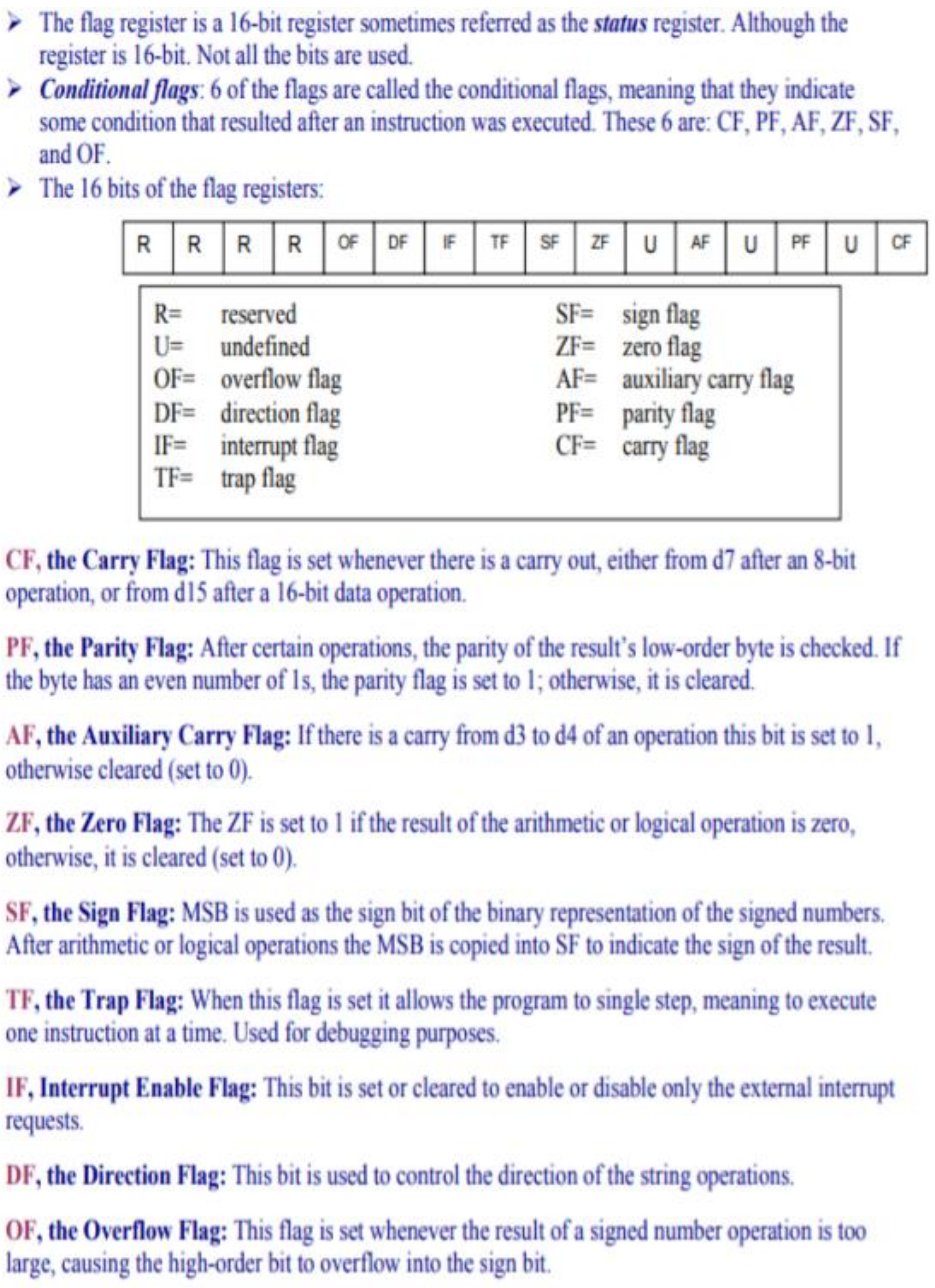
OFFSET is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it. When the assembler reads the statement MOV BX, OFFSET PRICES, for example, it will determine the offset of the variable PRICES from the start of the segment in which PRICES is defined and will load this value into BX.

iv.PTR (POINTER)

The PTR operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction INC [BX], for example, it will not know whether to increment the byte pointed to by BX. We use the PTR operator to clarify how we want the assembler to code the instruction. The statement INC BYTE PTR [BX] tells the assembler that we want to increment the byte pointed to by BX. The statement INC WORD PTR [BX] tells the assembler that we want to increment the word pointed to by BX. The PTR operator assigns the type specified before PTR to the variable specified after PTR. We can also use the PTR operator to clarify our intentions when we use indirect Jump instructions. The statement JMP [BX], for example, does not tell the assembler whether to code the instruction for a near jump. If we want to do a near jump, we write the instruction as JMP WORD PTR [BX]. If we want to do a far jump, we write the instruction as JMP DWORD PTR [BX].

2 b. What is the flag and flag register? Explain the format of flag register with a suitable example.

The 8086 PSW is 16 bits, but only 9 of its bits are used. Each bit of 8086 PSW is called a flag.



- The flag register is a 16-bit register sometimes referred as the ***status*** register. Although the register is 16-bit. Not all the bits are used.
- ***Conditional flags***: 6 of the flags are called the conditional flags, meaning that they indicate some condition that resulted after an instruction was executed. These 6 are: CF, PF, AF, ZF, SF, and OF.
- The 16 bits of the flag registers:

R	R	R	R	OF	DF	IF	TF	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		

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-bit data 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 is cleared.

AF, the Auxiliary Carry Flag: If there is a carry from d3 to d4 of an operation this bit is set to 1, otherwise cleared (set to 0).

ZF, the Zero Flag: The ZF is set to 1 if the result of the arithmetic or logical operation is zero, otherwise, it is cleared (set to 0).

SF, the Sign Flag: MSB is used as the sign bit of the binary representation of the signed numbers. After arithmetic or logical operations the MSB is copied into SF to indicate 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. Used for debugging purposes.

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

DF, the Direction Flag: This bit is used to control the direction of the string operations.

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.

2c. Write an assembly level program to sort of 'n' 16 bit –numbers in descending order. Use bubble sort algorithm to sort given elements.

PROGRAM TO SORT THE NUMBERS IN DESCENDING ORDER

```
DATA SEGMENT
x DW 42H,34H,26H,17H,09H
LEN EQU 05
ASCD DB 10 DUP(0)
```

```
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE,DS:DATA

START: MOV AX,DATA
MOV DS,AX
MOV BX,LEN-1
MOV CX,BX
UP1: MOV BX,CX
LEA SI,X

UP: MOV AX,[SI]
MOV DX,[SI+2]
CMP AX,DX
JA DOWN
MOV [SI],DX
MOV [SI+2],AX

DOWN: INC SI
INC SI
DEC BX
JNZ UP
DEC CX
JNZ UP1
MOV AH,4CH
INT 21H
CODE ENDS

END START
```

Module 2

3a. Explain the following instructions with a suitable example.

1.MOV – MOV Destination, Source

The MOV instruction copies a word or byte of data from a specified source to a specified destination. The destination can be a register or a memory location. The source can be a register, a memory location or an immediate number. The source and destination cannot both be memory locations. They must both be of the same type (bytes or words). MOV instruction does not affect any flag.

MOV CX, 037AH	Put immediate number 037AH to CX
MOV BL, [437AH]	Copy byte in DS at offset 437AH to BL
MOV AX, BX	Copy content of register BX to AX
MOV DL, [BX]	Copy byte from memory at [BX] to DL
MOV DS, BX	Copy word from BX to DS register

MOV RESULT [BP], AX Copy AX to two memory locations; AL to the first location, AH to the second; EA of the first memory location is sum of the displacement represented by RESULTS and content of BP. Physical address = EA + SS.

MOV ES: RESULTS [BP], AX Same as the above instruction, but physical address = EA + ES, because of the segment override prefix ES

2.

PUSH – PUSH Source

The PUSH instruction decrements the stack pointer by 2 and copies a word from a specified source to the location in the stack segment to which the stack pointer points. The source of the word can be general-purpose register, segment register, or memory. The stack segment register and the stack pointer must be initialized before this instruction can be used. PUSH can be used to save data on the stack so that it will not destroyed by a procedure. This instruction does not affect any flag.

➢ PUSH BX	Decrement SP by 2, copy BX to stack.
➢ PUSH DS	Decrement SP by 2, copy DS to stack.
➢ PUSH BL	Illegal; must push a word
➢ PUSH TABLE [BX]	Decrement SP by 2, and copy word from memory in DS at EA = TABLE + [BX] to stack

3

LEA – LEA Register, Source

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16-bit register. LEA does not affect any flag.

➢ LEA BX, PRICES	Load BX with offset of PRICE in DS
➢ LEA BP, SS: STACK_TOP	Load BP with offset of STACK_TOP in SS
➢ LEA CX, [BX][DI]	Load CX with EA = [BX] + [DI]

4.

SHR – SHR Destination, Count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is put in its place. The bit shifted out of the LSB position goes to CF. In the case of multi-bit shifts, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.

$$\begin{tikzcd}
0 \arrow[r] & \text{MSB} \arrow[r, dotted] & \text{LSB} \arrow[r] & \text{CF}
\end{tikzcd}$$

The destination operand can be a byte or a word in a register or in a memory location. If you want to shift the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. For shifts of more than 1 bit position, load the desired number of shifts into the CL register, and put “CL” in the count position of the instruction.

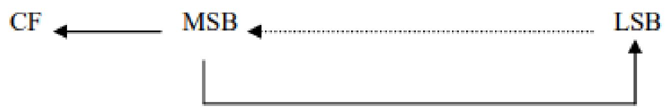
The flags are affected by SHR as follow: CF contains the bit most recently shifted out from LSB. For a count of one, OF will be 1 if the two MSBs are not both 0’s. For multiple-bit shifts, OF will be meaningless. SF and ZF will be updated to show the condition of the destination. PF will have meaning only for an 8-bit destination. AF is undefined.

➢ SHR BP, 1	Shift word in BP one bit position right, 0 in MSB
➢ MOV CL, 03H	Load desired number of shifts into CL
SHR BYTE PTR [BX]	Shift byte in DS at offset [BX] 3 bits right; 0’s in 3 MSBs

5.

ROL – ROL Destination, Count

This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF. In the case of multiple-bit rotate, CF will contain a copy of the bit most recently moved out of the MSB.



The destination can be a register or a memory location. If you to want rotate the operand by one bit position, you can specify this by putting 1 in the count position in the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

ROL affects only CF and OF. OF will be a 1 after a single bit ROL if the MSB was changed by the rotate.

➢ ROL AX, 1	Rotate the word in AX 1 bit position left, MSB to LSB and CF
➢ MOV CL, 04H	Load number of bits to rotate in CL
ROL BL, CL	Rotate BL 4 bit positions
➢ ROL FACTOR [BX], 1	Rotate the word or byte in DS at EA = FACTOR [BX] by 1 bit position left into CF

6.CMP – CMP Destination, Source

This instruction compares a byte / word in the specified source with a byte / word in the specified destination. The source can be an immediate number, a register, or a memory location. The destination can be a register or a memory location. However, the source and the destination cannot both be memory locations. The comparison is actually done by subtracting the source byte or word from the destination byte or word. The source and the destination are not changed, but the flags are set to indicate the results of the comparison. AF, OF, SF, ZF, PF, and CF are updated by the CMP instruction. For the instruction CMP CX, BX, the values of CF, ZF, and SF will be as follows:

Ex: CMP AL, 01H Compare immediate number 01H with byte in AL

7. DAA (DECIMAL ADJUST AFTER BCD ADDITION)

This instruction is used to make sure the result of adding two packed BCD numbers is adjusted to be a legal BCD number. The result of the addition must be in AL for DAA to work correctly. If the lower nibble in AL after an addition is greater than 9 or AF was set by the addition, then the DAA instruction will add 6 to the lower nibble in AL. If the result in the upper nibble of AL in now greater than 9 or if the carry flag was set by the addition or correction, then the DAA instruction will add 60H to AL.

Let AL = 59 BCD, and BL = 35 BCD

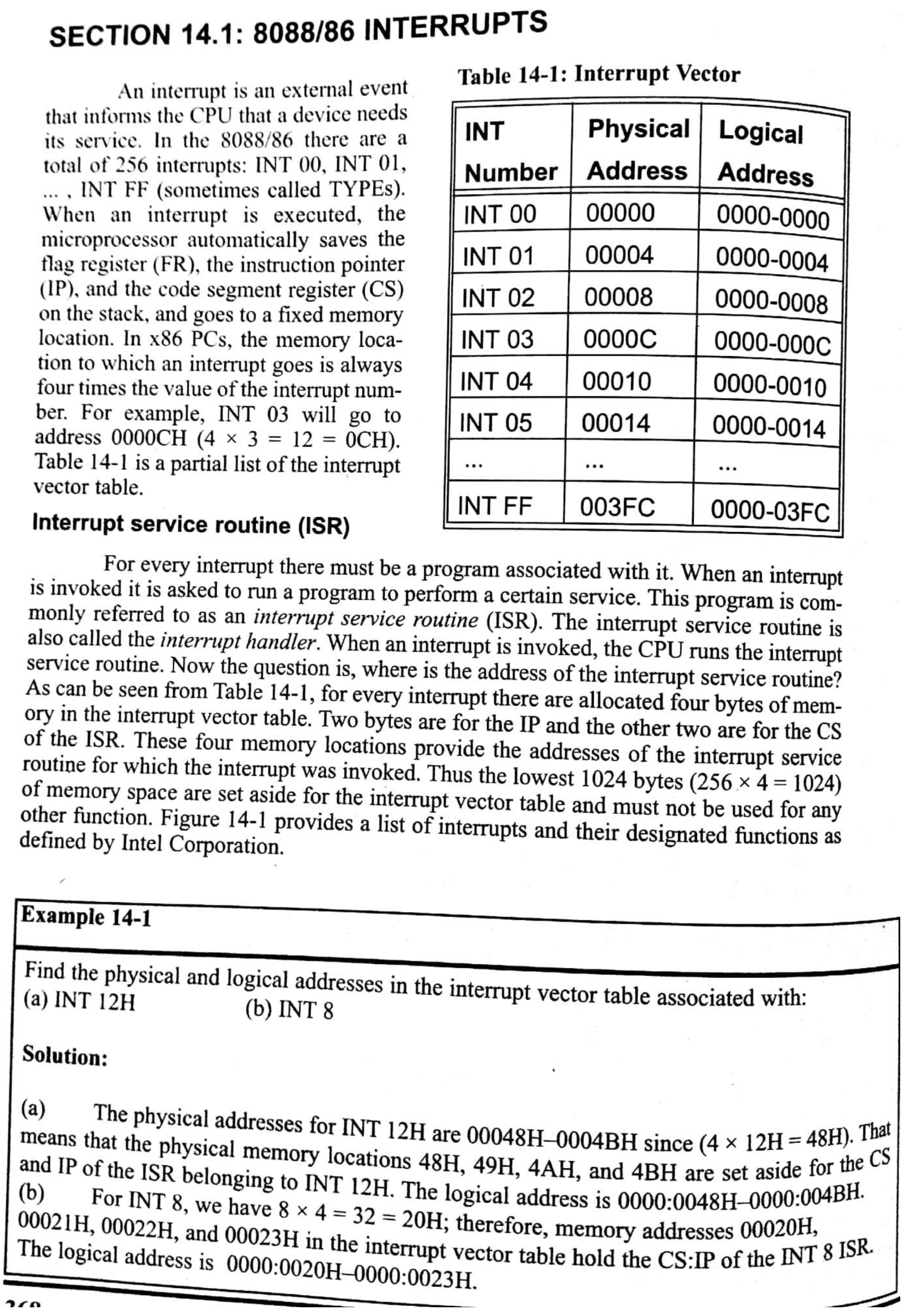
ADD AL, BL	AL = 8EH; lower nibble > 9, add 06H to AL
DAA	AL = 94 BCD, CF = 0

8. TEST – TEST Destination, Source

This instruction ANDs the byte / word in the specified source with the byte / word in the specified destination. Flags are updated, but neither operand is changed. The test instruction is often used to set flags before a Conditional jump instruction. The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and the destination cannot both be memory locations. CF and OF are both 0's after TEST. PF, SF and ZF will be updated to show the results of the destination. AF is be undefined.

TEST AL, BH	AND BH with AL. No result stored; Update PF, SF, ZF.
TEST CX, 0001H	AND CX with immediate number 0001H; No result stored; Update PF, SF, ZF

3b.What is an interrupt. ?Explain various types with an interrupt vector table.
Ans:



SECTION 14.1: 8088/86 INTERRUPTS

An interrupt is an external event that informs the CPU that a device needs its service. In the 8088/86 there are a total of 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 location 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). Table 14-1 is a partial list of the interrupt vector table.

Table 14-1: 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. Now the question is, where is the address of the interrupt service routine? As can be seen from Table 14-1, 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 × 4 = 1024) of memory space are set aside for the interrupt vector table and must not be used for any other function. Figure 14-1 provides a list of interrupts and their designated functions as defined by Intel Corporation.

Example 14-1

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.

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. This means that 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.

Hardware interrupts

As we saw in Chapters 9 and 10, there are three pins in the x86 that are associated with hardware interrupts. They are INTR (interrupt request), NMI (nonmaskable interrupt), and INTA (interrupt acknowledge). The use of INTA will be discussed in Section 14.3. INTR is an input signal into the CPU, which can be masked (ignored) and unmasked through the use of instructions CLI and STI. However, 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 *nonmaskable* interrupt. INTR and NMI are activated externally by putting 5 V 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. In contrast, this is not the case for the other hardware pin, INTR. There is no specific location in the vector table assigned to INTR. The reason is that INTR is used to expand the number of hardware interrupts and should be allowed to use any "INT nn" that has not been previously assigned. The 8259 programmable interrupt controller (PIC) chip can be connected to INTR to expand the number of hardware interrupts to 64. In the case of the IBM PC, one Intel 8259 PIC chip is used to add a total of 8 hardware interrupts to the microprocessor. IBM PC AT, PS/2 80286, 80386, 80486, and Intel Pentium computers use two 8259 chips to allow up to 16 hardware interrupts. The design of hardware interrupts and the use of the 8259 in the IBM PC are covered in Sections 14.3 and 14.4, while ISA bus interrupts are covered in Section 14.5.

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 a 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", which were covered in Chapter 4. These interrupts can be invoked in the sequence of code just like a CALL or 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. These interrupts are shown in Figure 14-1. Aside from "INT 00" to "INT 04", which have predefined functions, the rest of the interrupts from "INT 05" to "INT FF" can be used to implement either software or hardware interrupts.

Interrupts and the flag register

Among bits D0 to D15 of the flag register, there are two bits that are associated with the interrupt: D9, or IF (interrupt enable flag), and D8, or TF (trap or single step flag). In addition, OF (overflow flag) can be used by the interrupt. See Figure 14-2.

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. There are many more exception handling interrupts in x86 CPUs, which are discussed in Section 14.5. 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
```

Put INT 3 at the end of the above two programs in DEBUG and see the reaction of the PC. For further discussion of divide error interrupts due to an oversized quotient, see Chapter 3.

INT 01 (single step)

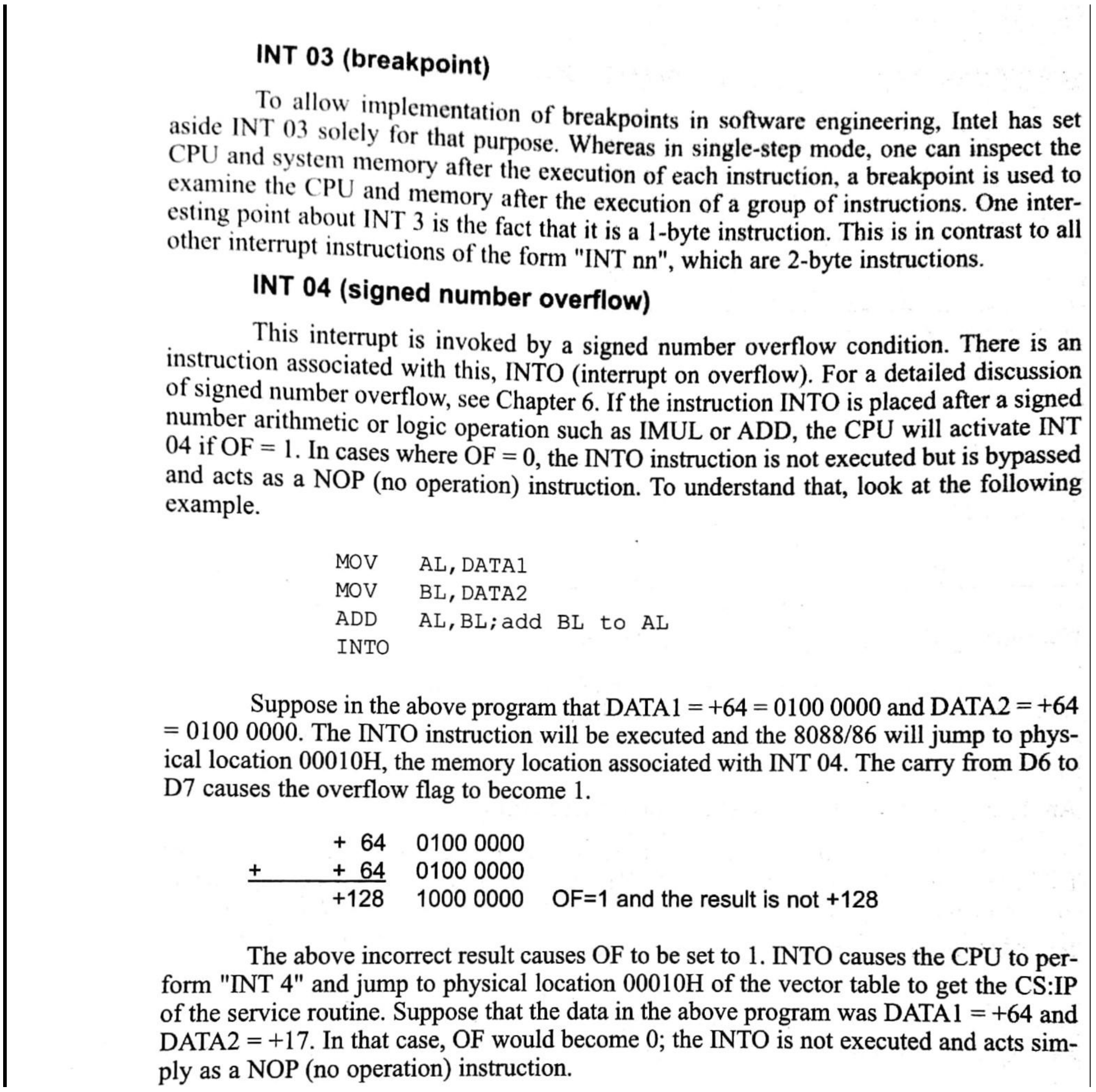
In executing a sequence of instructions, there is a need to examine the contents of the CPU's registers and system memory. This is often done by executing the program one instruction at a time and then inspecting registers and memory. This is commonly referred to as *single-stepping*, or performing a 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, whose job is, among other things, to dump the registers onto the screen. Now the question is, how is the trap flag set or reset? Although Intel has not provided any specific instruction for this purpose (unlike IF, which uses STI and CLI instructions to set or reset), 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. The analysis of the above two programs is left to the reader. To make TF = 1, one simply uses the OR instruction in place of the AND instruction above.

INT 02 (nonmaskable 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. Section 14.4 contains a detailed discussion of its purpose and application.



INT 03 (breakpoint)

To allow implementation of breakpoints in software engineering, Intel has set aside INT 03 solely for that purpose. Whereas 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. One interesting point about INT 3 is the fact that it is a 1-byte instruction. This is in contrast to all other interrupt instructions of the form "INT nn", which 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). For a detailed discussion of signed number overflow, see Chapter 6. If the instruction INTO is placed after a signed number arithmetic or logic operation such as IMUL or ADD, 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 that, look at the following example.

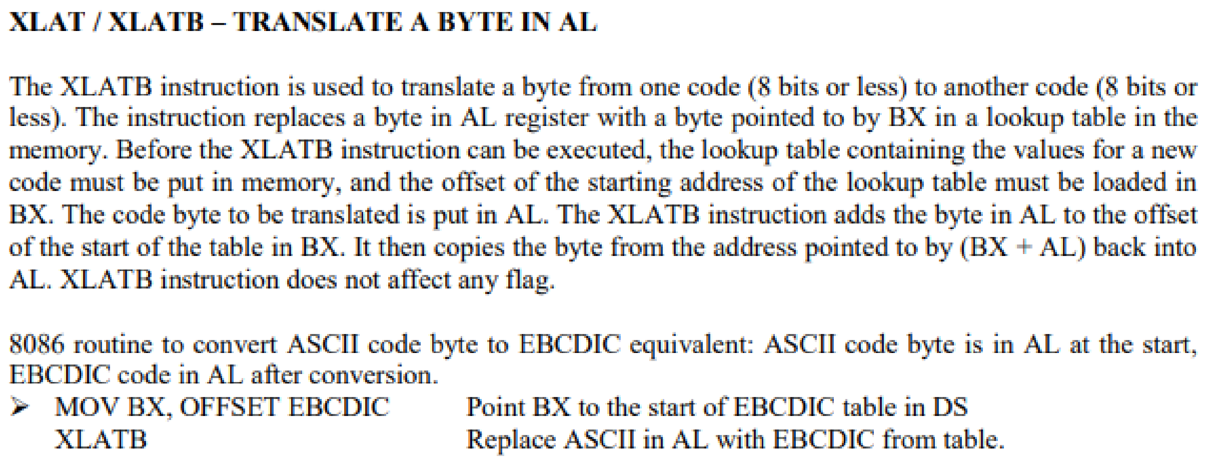
```
MOV   AL,DATA1
MOV   BL,DATA2
ADD   AL,BL;add BL to AL
INTO
```

Suppose in the above program that 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 1.

```
        + 64    0100 0000
+       + 64    0100 0000
        +128    1000 0000   OF=1 and the result is not +128
```

The above incorrect result causes OF to be set to 1. 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. 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.

4a.Explain the following instructions with a suitable example:.



XLAT / XLATB – TRANSLATE A BYTE IN AL

The XLATB instruction is used to translate a byte from one code (8 bits or less) to another code (8 bits or less). The instruction replaces a byte in AL register with a byte pointed to by BX in a lookup table in the memory. Before the XLATB instruction can be executed, the lookup table containing the values for a new code must be put in memory, and the offset of the starting address of the lookup table must be loaded in BX. The code byte to be translated is put in AL. The XLATB instruction adds the byte in AL to the offset of the start of the table in BX. It then copies the byte from the address pointed to by (BX + AL) back into AL. XLATB instruction does not affect any flag.

8086 routine to convert ASCII code byte to EBCDIC equivalent: ASCII code byte is in AL at the start, EBCDIC code in AL after conversion.

```
➢ MOV BX, OFFSET EBCDIC       Point BX to the start of EBCDIC table in DS
  XLATB                       Replace ASCII in AL with EBCDIC from table.
```

ii.

RCR – RCR Destination, Count

This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotate around into MSB of the operand.

$$\begin{tikzcd}
CF \arrow[r] & MSB \arrow[r, dotted] & LSB \arrow[ll, bend left]
\end{tikzcd}$$

For multi-bit rotate, CF will contain the bit most recently rotated out of the LSB.

The destination can be a register or a memory location. If you want to rotate the operand by one bit position, you can specify this by putting a 1 in the count position of the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

RCR affects only CF and OF. OF will be a 1 after a single bit RCR if the MSB was changed by the rotate. OF is undefined after the multi-bit rotate.

➢ RCR BX, 1	Word in BX right 1 bit, CF to MSB, LSB to CF
➢ MOV CL, 4	Load CL for rotating 4 bit position
RCR BYTE PTR [BX], 4	Rotate the byte at offset [BX] in DS 4 bit positions right
	CF = original bit 3, Bit 4 – original CF.

iii.

AAA (ASCII ADJUST FOR ADDITION)

Numerical data coming into a computer from a terminal is usually in ASCII code. In this code, the numbers 0 to 9 are represented by the ASCII codes 30H to 39H. The 8086 allows you to add the ASCII codes for two decimal digits without masking off the "3" in the upper nibble of each. After the addition, the AAA instruction is used to make sure the result is the correct unpacked BCD.

➢ Let AL = 0011 0101 (ASCII 5), and BL = 0011 1001 (ASCII 9)

ADD AL, BL	AL = 0110 1110 (6EH, which is incorrect BCD)
AAA	AL = 0000 0100 (unpacked BCD 4)
	CF = 1 indicates answer is 14 decimal.

The AAA instruction works only on the AL register. The AAA instruction updates AF and CF; but OF, PF, SF and ZF are left undefined.

iv.

MUL – MUL Source

This instruction multiplies an *unsigned* byte in some *source* with an *unsigned* byte in AL register or an unsigned word in some *source* with an unsigned word in AX register. The source can be a register or a memory location. When a byte is multiplied by the content of AL, the result (product) is put in AX. When a word is multiplied by the content of AX, the result is put in DX and AX registers. If the most significant byte of a 16-bit result or the most significant word of a 32-bit result is 0, CF and OF will both be 0's. AF, PF, SF and ZF are undefined after a MUL instruction.

If you want to multiply a byte with a word, you must first move the byte to a word location such as an extended register and fill the upper byte of the word with all 0's. You cannot use the CBW instruction for this, because the CBW instruction fills the upper byte with copies of the most significant bit of the lower byte.

➢ MUL BH	Multiply AL with BH; result in AX
➢ MUL CX	Multiply AX with CX; result high word in DX, low word in AX
➢ MUL BYTE PTR [BX]	Multiply AL with byte in DS pointed to by [BX]

v.

DIV – DIV Source

This instruction is used to divide an *unsigned* word by a byte or to divide an *unsigned* double word (32 bits) by a word. When a word is divided by a byte, the word must be in the AX register. The divisor can be in a register or a memory location. After the division, AL will contain the 8-bit quotient, and AH will contain the 8-bit remainder. When a double word is divided by a word, the most significant word of the double word must be in DX, and the least significant word of the double word must be in AX. After the division, AX will contain the 16-bit quotient and DX will contain the 16-bit remainder. If an attempt is made to divide by 0 or if the quotient is too large to fit in the destination (greater than FFH / FFFFH), the 8086 will generate a type 0 interrupt. All flags are undefined after a DIV instruction.

If you want to divide a byte by a byte, you must first put the dividend byte in AL and fill AH with all 0's. Likewise, if you want to divide a word by another word, then put the dividend word in AX and fill DX with all 0's.

➢ DIV BL	Divide word in AX by byte in BL; Quotient in AL, remainder in AH
➢ DIV CX	Divide down word in DX and AX by word in CX; Quotient in AX, and remainder in DX.

vi.

LOOP (JUMP TO SPECIFIED LABEL IF CX ≠ 0 AFTER AUTO DECREMENT)

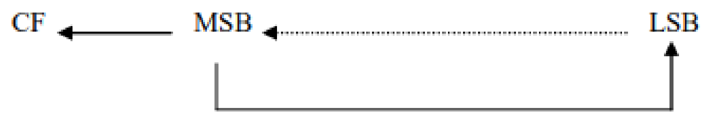
This instruction is used to repeat a series of instructions some number of times. The number of times the instruction sequence is to be repeated is loaded into CX. Each time the LOOP instruction executes, CX is automatically decremented by 1. If CX is not 0, execution will jump to a destination specified by a label in the instruction. If CX = 0 after the auto decrement, execution will simply go on to the next instruction after LOOP. The destination address for the jump must be in the range of –128 bytes to +127 bytes from the address of the instruction after the LOOP instruction. This instruction does not affect any flag.

```
➢ MOV BX, OFFSET PRICES      Point BX at first element in array
  MOV CX, 40                 Load CX with number of elements in array
  NEXT: MOV AL, [BX]         Get element from array
  INC AL                     Increment the content of AL
  MOV [BX], AL               Put result back in array
  INC BX                     Increment BX to point to next location
  LOOP NEXT                  Repeat until all elements adjusted
```

vii.

ROL – ROL Destination, Count

This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF. In the case of multiple-bit rotate, CF will contain a copy of the bit most recently moved out of the MSB.



The destination can be a register or a memory location. If you to want rotate the operand by one bit position, you can specify this by putting 1 in the count position in the instruction. To rotate more than one bit position, load the desired number into the CL register and put "CL" in the count position of the instruction.

ROL affects only CF and OF. OF will be a 1 after a single bit ROL if the MSB was changed by the rotate.

➢ ROL AX, 1	Rotate the word in AX 1 bit position left, MSB to LSB and CF
➢ MOV CL, 04H	Load number of bits to rotate in CL
ROL BL, CL	Rotate BL 4 bit positions
➢ ROL FACTOR [BX], 1	Rotate the word or byte in DS at EA = FACTOR [BX] by 1 bit position left into CF

viii.

OR – OR Destination, Source

This instruction ORs each bit in a source byte or word with the same numbered bit in a destination byte or word. The result is put in the specified destination. The content of the specified source is not changed.

The source can be an immediate number, the content of a register, or the content of a memory location. The destination can be a register or a memory location. The source and destination cannot both be memory locations. CF and OF are both 0 after OR. PF, SF, and ZF are updated by the OR instruction. AF is undefined. PF has meaning only for an 8-bit operand.

➢ OR AH, CL	CL ORed with AH, result in AH, CL not changed

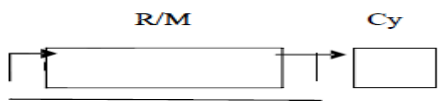
4 b. Explain rotate instructions with an example.

Ans:

ROR R/M, 1/CL	Used for division by power of 2

CL has no. of times rotation is to be done

ROR BH, 1

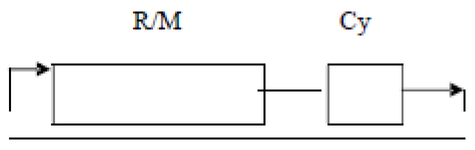


Rotate right without cy		Before	After
	BH	0100 0010	0010 0001
Cy		1	0

RCR R/M, 1/CL

CL has no. of times rotation is to be done

RCR BH, 1

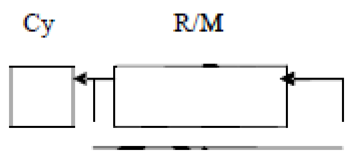


Rotate right with cy

	Before	After
BH	0100 0010	1010 0001
Cy	1	0

ROL R/M, 1/CL Used for multiplication by 2^n

ROL BH, CL

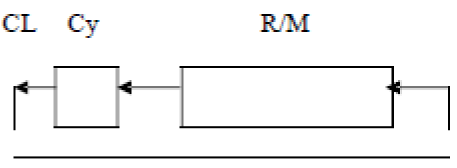


Rotate left without cy

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

RCL R/M, 1/CL

RCL BH, CL



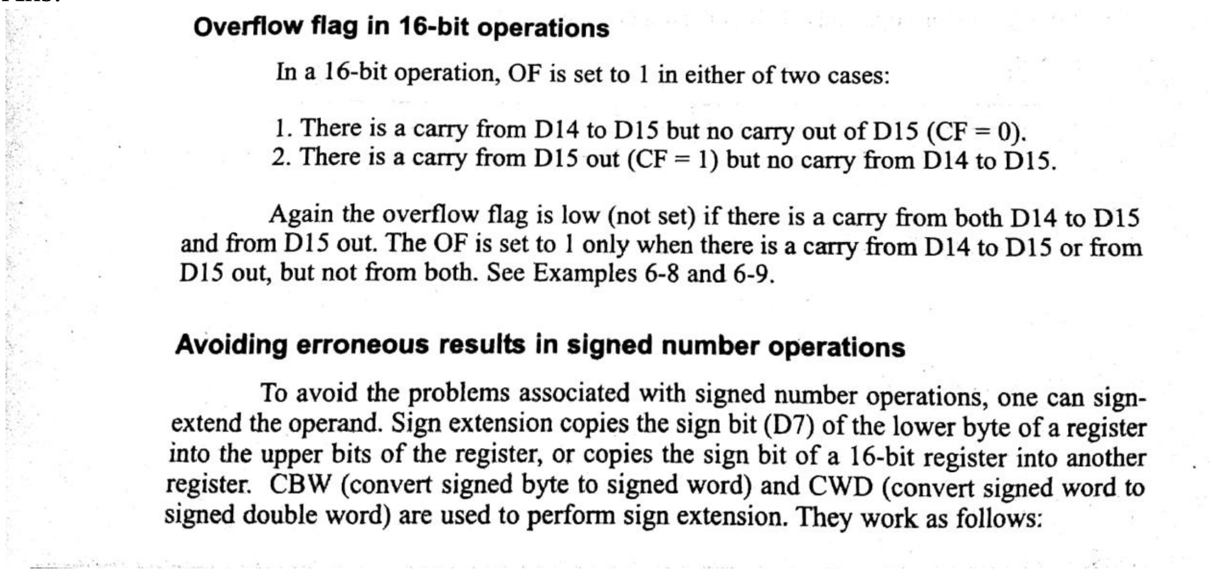
Rotate left with cy

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

Module 3

5a. With example , explain how to identify overflow using flags in a flag register for performing an arithmetic operation on 16 bit numbers.

Ans:



Overflow flag in 16-bit operations

In a 16-bit operation, OF is set to 1 in either of two cases:

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.

Again the overflow flag is low (not set) if there is a carry from both D14 to D15 and from D15 out. The OF is set to 1 only when there is a carry from D14 to D15 or from D15 out, but not from both. See Examples 6-8 and 6-9.

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 copies the sign bit of a 16-bit register into another register. CBW (convert signed byte to signed word) and CWD (convert signed word to signed double word) are used to perform sign extension. They work as follows:

Example 6-8

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
```

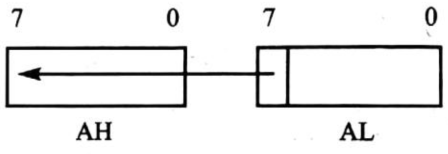
Example 6-9

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
```

CBW will copy D7 (the sign flag) to all bits of AH. This is demonstrated below. Notice that the operand is assumed to be AL and the previous contents of AH are destroyed.

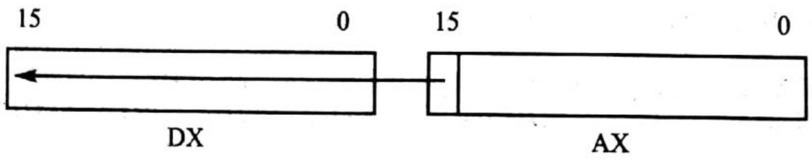


```
        MOV     AL,+96          ;AL=0110 0000
        CBW                     ;now AH=0000 0000 and AL=0110 0000
```

or:

```
        MOV     AL,-2           ;AL=1111 1110
        CBW                     ;AH=1111 1111 and AL=1111 1110
```

CWD sign-extends AX. It copies D15 of AX to all bits of the DX register. This is used for signed word operands. This is illustrated below.



Look at the following example:

```
MOV     AX,+260         ;AX=0000 0001 0000 0100 or AX=0104H
CWD                     ;DX=0000H and AX=0104H
```

Another example:

```
MOV     AX,-32766       ;AX=1000 0000 0000 0010B or AX=8002H
CWD                     ;DX=FFFF and AX=8002
```

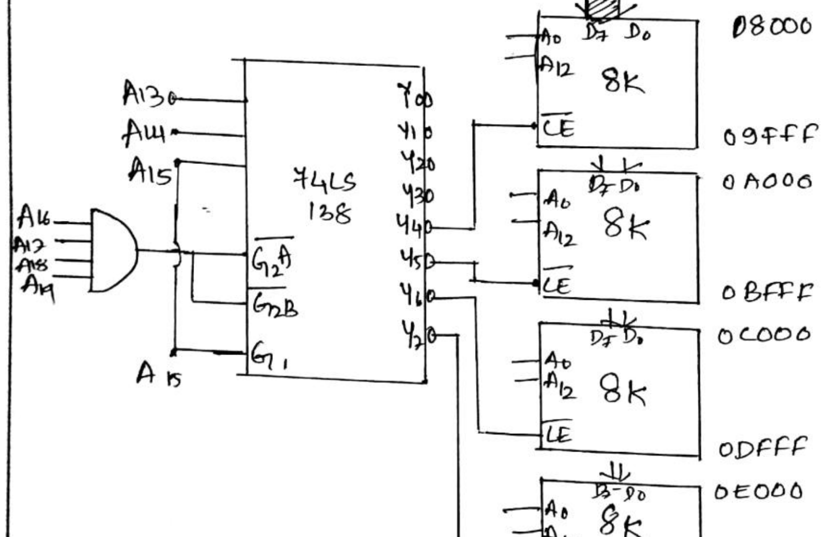
5b. Explain 74138 decoder configuration to enable the memory address 08000 H to 0FFFF H to connect four 8k RAMS.

Ans.

- Address lines req^d to connect 8K RAM ⇒ $2^{13} = 8K$ is 13 lines $(A_0 - A_{12})$
- Address Range

	$A_{19} A_{18} A_{17} A_{16}$	$A_{15} A_{14} A_{13}$	A_{12}	A_{11} — A_8	A_7 — A_4	A_3 — D_0
08000	0 0 0 0	1 0 0	0	0000	0000	0000
0FFF F	0 0 0 0	1 1 1	1	1111	1111	1111

$\underbrace{A_{15} A_{14} A_{13}}_{\text{I/p to 74LS138 decoder}}$ $\underbrace{A_{12} \ldots A_0}_{\text{Address lines}}$



	A_{15} A_{14} A_{13}	Chip Selected
Y_4	1 0 0	1
Y_5	1 0 1	2
Y_6	1 1 0	3
Y_7	1 1 1	4

6.

a. **Briefly explain the control word format of 8255 in I/O and BSR mode. Find the control word if P_A=o/p, P_B =i/p, P_{CL}=i/p, P_{CU}=o/p. Use port address of 300h-303H and write a program to read from port A and send to Port B.**

8255

The 8255 is a 40-pin DIP chip. It has three separately accessible ports. The ports are each 8-bit, and are named A, B, and C. The individual ports of the 8255 can be programmed to be input or output, and can be changed dynamically. In addition, 8255 ports, have handshaking capability, thereby allowing interface with devices needs handshaking signals, such as printers.

Mode selection of the 8255

While ports A, B, and C are used to input or output data, it is the control register that must be programmed to select the operation mode of the three ports. The ports of the 8255 can be programmed in any of the following modes.

1. Mode 0, simple I/O mode. In this mode, any of the ports A, B, CL, and CU can be programmed as input or output. In this mode, all bits are out or all are in. In other words, there is no such thing as single-bit control as we have seen in PO – P3 of the 8051. Since the vast majority of applications involving the 8255 use this simple I/O mode, we will concentrate on this mode in this chapter.
2. Mode 1. In this mode, ports A and B can be used as input or output ports with handshaking capabilities. Handshaking signals are provided by the bits of port C.
3. Mode 2. In this mode, port A can be used as a bidirectional I/O port with hand shaking capabilities whose signals are provided by port C. Port B can be used either in simple I/O mode or handshaking mode 1.
4. BSR (bit set/reset) mode. In this mode, only the individual bits of port C can be programmed.

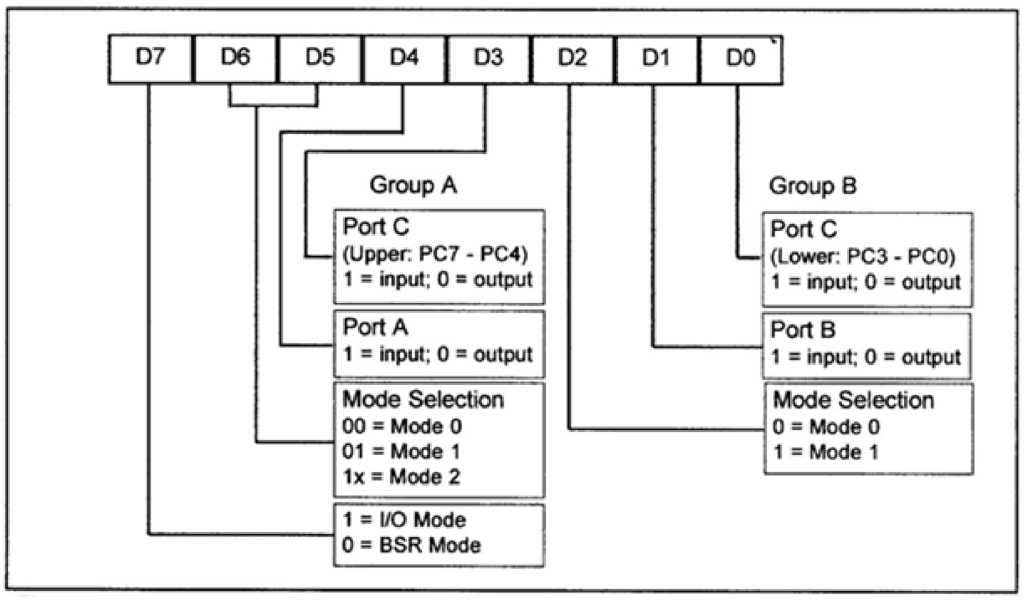
The 8255 chip is programmed in any of the 4 modes mentioned by sending a byte to the control register of the 8255. We must first find the port addresses assigned to each of ports A, B, C, and the control register. This is called *mapping* the I/O port.

Instructions for input and output port transfer

- **IN** − Used to read a byte from the provided port to the accumulator.
- **OUT** − Used to send out a byte from the accumulator to the provided port.

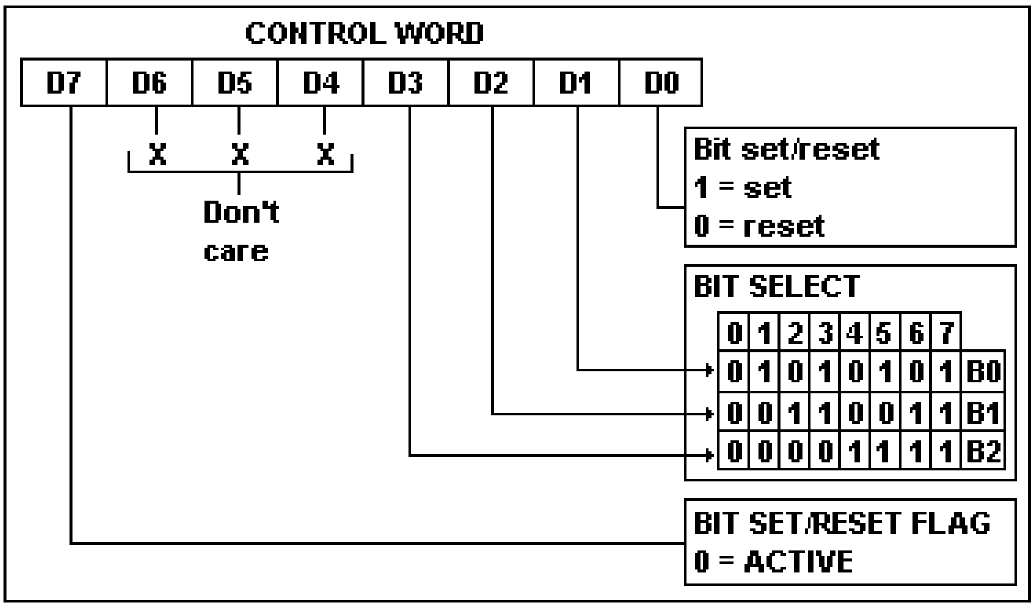
Control Word register format:

I/O mode



Control Word Format 8255A

<u>BSR Mode</u>



0	1	2	3	4	5	6	7	
0	1	0	1	0	1	0	1	B0
0	0	1	1	0	0	1	1	B1
0	0	0	0	1	1	1	1	B2

<u>Control word if P_A=o/p, P_B =i/p, P_{CL}=i/p, P_{CU}=o/p</u>

CONTROL WORD: 10000011 = 83H

<u>Control word if P_A=i/p, P_B =0/p, P_C=o/p</u>

CONTROL WORD: 10010000 = 90H

```
.MODEL SMALL
.STACK 100
.DATA
PA EQU 300H
PB EQU 301H
CT EQU 303H
.CODE
MOV AX, @DATA
MOV DS, AX
MOV DX, CT
MOV AL, 90H
OUT DX, AL
MOV DX, PA
IN AL, DX
MOV DX, PB
OUT DX, AL
MOV AH, 4CH
INT 21H
END
```

b. Write an ALP to read PB and check the number of ones in an 8 bit data. Put FFH on Port A if it is even parity else display 00 on port A.

Control word if PA=o/p, PB =i/p, PC=o/p

CONTROL WORD: 10000010 = 82H

```
.MODEL SMALL
.STACK 100
.DATA
PA EQU 300H
PB EQU 301H
CT EQU 303H
.CODE
MOV AX, @DATA
MOV DS, AX
MOV DX, CT
MOV AL, 82H
OUT DX, AL
MOV DX, PB
IN AL, DX
; check the number of ones
MOV CX,8
MOV BL,00
BACK:SHR AL,1
JNC ZERO
INC BL; Number of ones
ZERO:LOOP BACK
SHR BL,1 ;check number of ones even number or not
JNC DISP
MOV AL,00H
```

```
JMP LAST

DISP:MOV AL,0FFH

LAST:MOV DX, PA

OUT DX, AL

MOV AH, 4CH

INT 21H

END
```

MODULE 4

7. ***Module 4***
    a. ***Compare CISC with RISC***

The RISC (Reduced Instruction Set Computer) 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

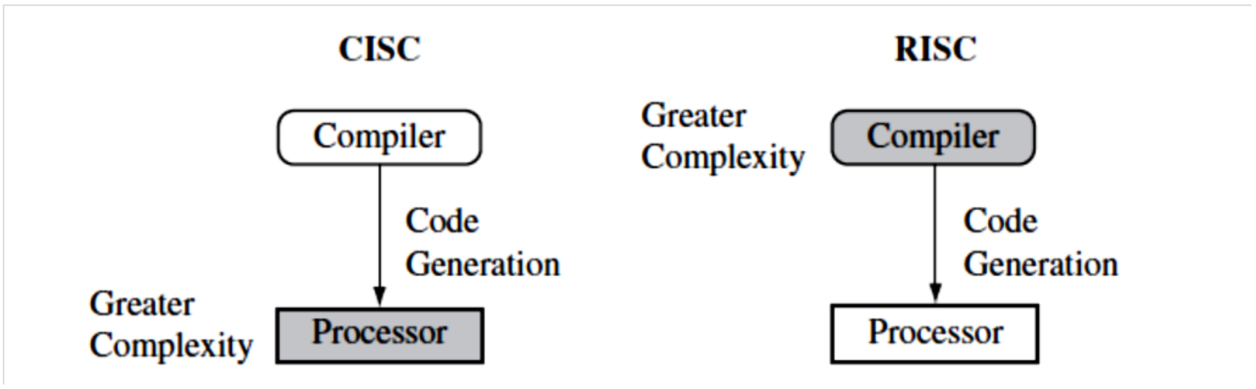
Instructions—RISC processors have a reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (for example, a divide operation) by combining several simple instructions. Each instruction is a 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.

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.

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.

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.

Hardware complexity- RISC emphasizes on software complexity while CISC emphasizes on hardware complexity



b. Explain registers used under various modes.

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. The least significant 5 bits in the CPSR determines the mode. Each processor mode is either privileged or non-privileged.

- A privileged mode allows full read-write access to the CPSR. Conversely, 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: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one non-privileged mode (user).
- The processor can change mode by either writing directly in to the control field (b0-b4) when it is in a privileged mode or when exceptions or interrupts happens.
- 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 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.
- *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 when the processor encounters an instruction that is undefined or not supported by the implementation.
- *User* mode is used for programs and applications.

Banked Registers

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 the processor mode changes change processor mode, a banked register from the new mode will replace an existing register.

Banked registers of a particular mode are denoted by an underline character post-fixed to the mode
Mnemonic or *_mode*.

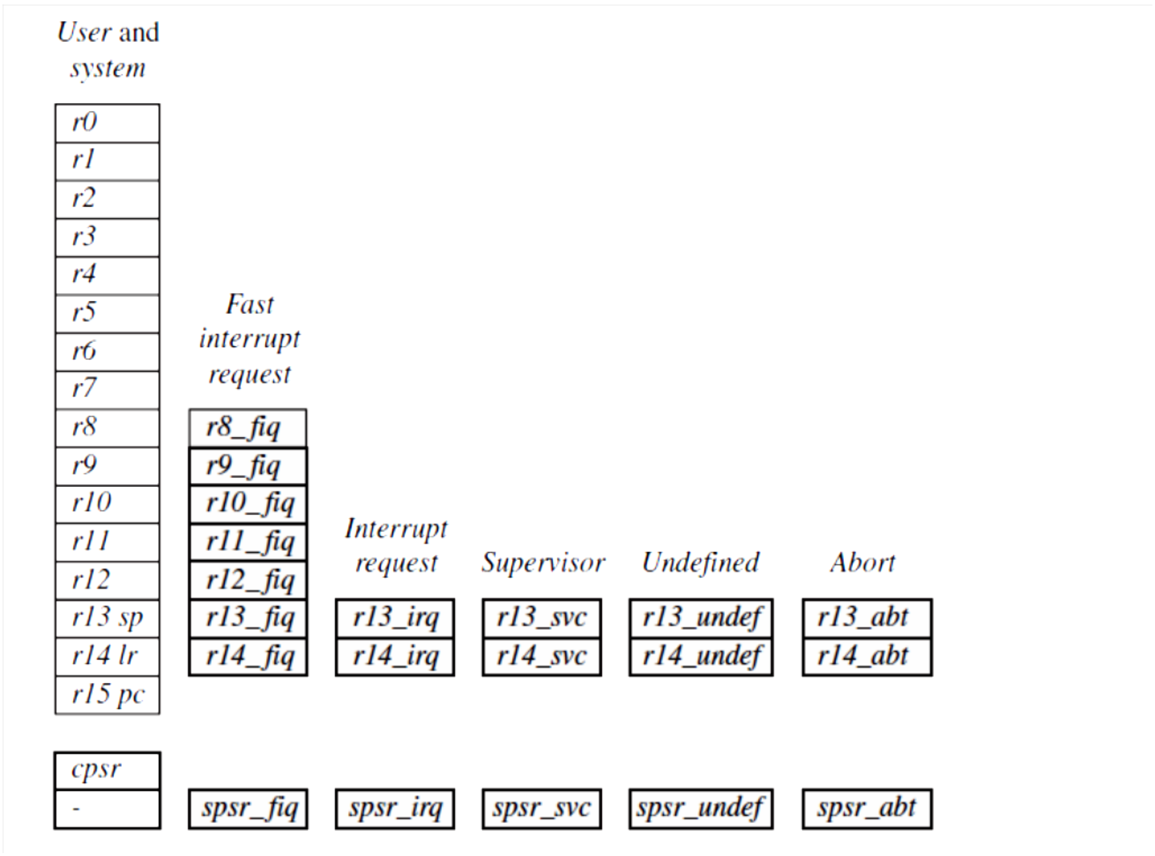


Fig 7.b.1 banked registers

Figure 7.b.1 shows all 37 registers in the register file. Of those, 20 registers are hidden from a program at different times. These registers are called *banked registers*. They are available only when the processor is in a particular mode.
For example, when the processor is in the interrupt request mode, the instructions user 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.

- The r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode, the *cpsr_usr* will be 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.

- Another important feature 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.

c. *Explain ARM core dataflow model*

A programmer can think of an ARM core as functional units connected by data buses, as shown in Figure 7.c.1, where, the arrows represent the flow of data, the lines represent the buses, and the boxes represent either an operation unit or a storage area. This model is called data flow model or programmers view of architecture.

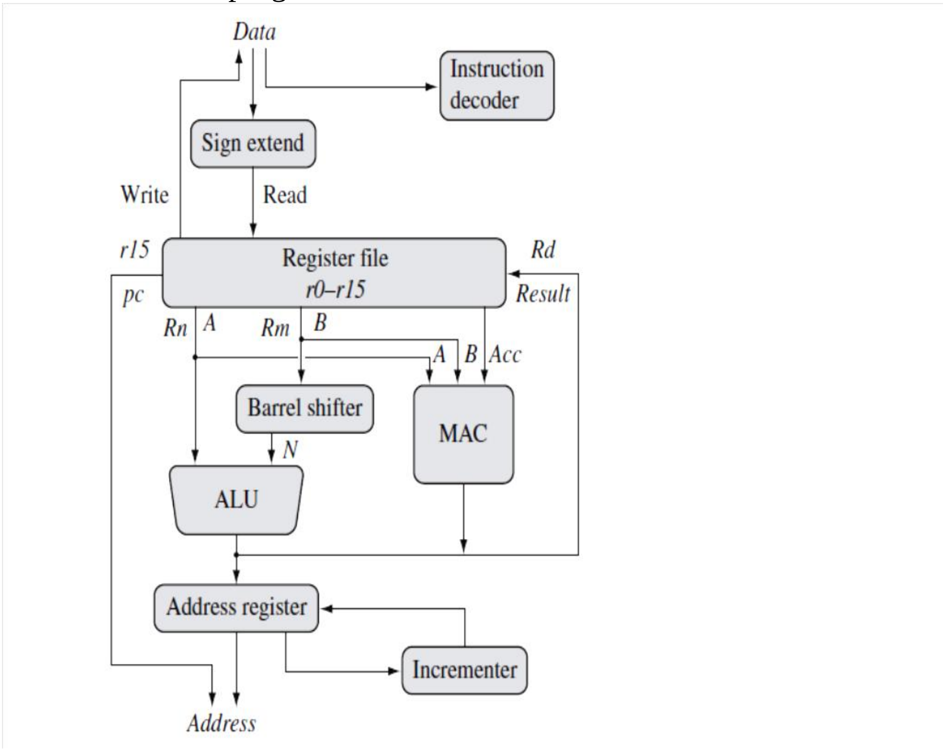


Fig 7.c.1 ARM core data flow model (Von Neumann Model)

- Data enters the processor core through the *Data* bus. The data may be an instruction to execute or a data item. The instruction decoder translates instructions before they are executed. Each instruction executed belongs to a particular instruction set.
- The ARM processor uses a *load-store architecture*. This 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, and conversely the 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 solely in registers.
- Data items are placed in the register file—a storage bank made up of 32-bit registers.

- Since the ARM7 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 pre-processed 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.

Registers
General-purpose registers are identified with the letter r prefixed to the register number. For example, register 1is given the label r1.There are up to 18 active registers: 16 data registers and 2 processor status registers. The data registers are visible to the programmer as r0 to r15.

Three registers are assigned with special function:

- Register r13 is traditionally used as the stack pointer (sp) and stores the head of the stack in the current processor mode.
- Register r14 is called the link register (lr) and is where the core puts the return address whenever it calls a subroutine.
- Register r15 is the program counter (pc) and contains the address of the next instruction to be fetched by the processor.
- In addition to the 16 data registers, there are two program status registers: cpsr (current PSR) and spsr (Saved PSR).
- The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

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. However, there are instructions that treat r14 and r15 in a special way.

8.

a. Explain the architecture of typical embedded device based in ARM core with a neat diagram.

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. Each component is chosen for efficiency and, if applicable, is designed for future extension and expansion.

Embedded System Hardware:

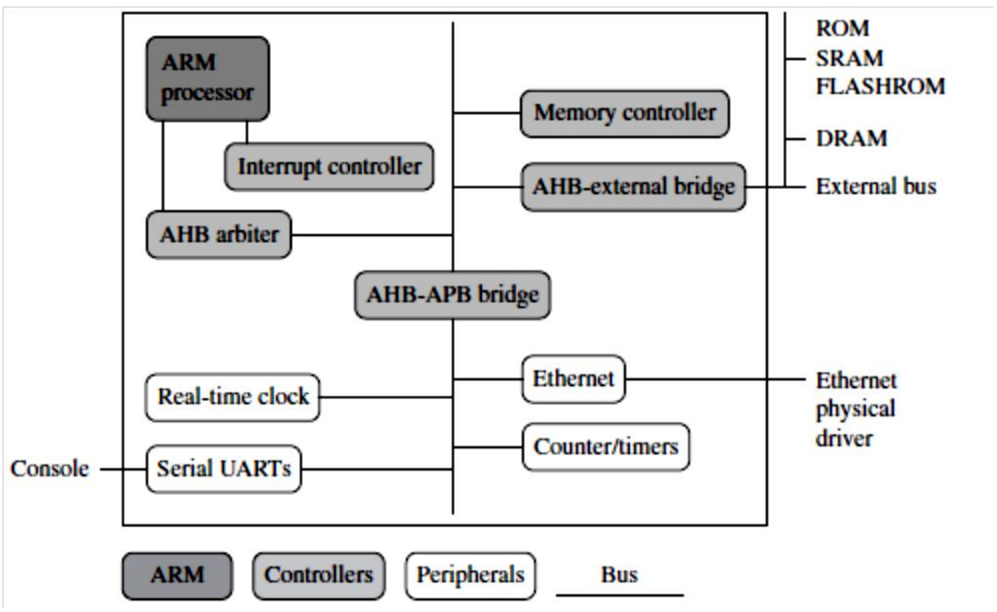


Figure 8.a.1 ARM based embedded device

Figure 8.a.1 shows a typical embedded device based on an ARM core. 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 that interface it with a bus. These components can include memory management and caches.
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. The ARM processor core is a bus master—a logical device capable of initiating a data transfer with another device across the same bus. Peripherals tend to be bus slaves—logical devices capable only of responding to a transfer request from a bus master device.

- AMBA Bus Protocol: The Advanced Microcontroller Bus Architecture (AMBA) 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 on to the on-chip bus without having to redesign an interface for each different processor architecture.
- ASB is a bidirectional bus design.
- APB is used with slower peripherals.
- AHB is based on a centralized multiplexed bus scheme, thus runs at higher clock speeds and provides higher data through put. AHB bus is used for the high performance peripherals.

Memory:

An embedded system has to have some form of memory to store and execute code. Cost, performance, and power consumption are the parameters considered while deciding upon specific memory characteristics, such as hierarchy, width, and type. Like if memory has to run twice as fast to maintain a desired bandwidth, then the memory power requirement may be higher.

- Hierarchy: Memory can be Cache, Main memory or Secondary memory.

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. The cache is placed between main memory and the core. It is used to speed up data transfer between the processor and main memory. The main memory is large 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. Many small embedded systems do not require the performance benefits of a cache.

- 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. If you have an un-cached system using 32-bit ARM instructions and 16-bit-wide memory chips, then the processor will have to make two memory fetches per instruction. Each fetch requires two 16-bit loads. This obviously has the effect of reducing system performance, but the benefit is that 16-bit memory is less expensive. In contrast, if the core executes 16-bit Thumb instructions, it will achieve better performance with a 16-bit memory. The higher performance is a result of the core making only a single fetch to memory to load an instruction. Hence, using Thumb instructions with 16-bit-wide memory devices provides both improved performance and reduced cost.

- Types: RAM or ROM
    - 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. ROMs are used in high-volume devices that require no updates or corrections. Many devices also use a ROM to hold boot code.
    - Random Access memory (RAM)- SRAM,DRAM or SDRAM

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 that are off-chip.

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 connect different types of memory to the processor bus. On power-on 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.

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 (VIC). 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 VIC is more powerful than the standard interrupt controller because it prioritizes interrupts and simplifies the determination of which device caused the interrupt.

b. What is CPSR? Explain relevant bits

- The CPSR is a dedicated 32-bit register which resides in the register file.
- The ARM core uses the CPSR to monitor and control internal operations.

The Figure 8.b.1 shows the CPSR layout.

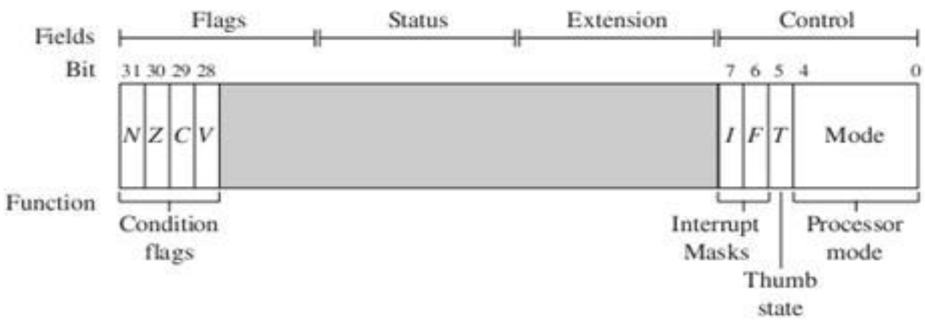


Fig 8.b.1 CPSR Layout

The CPSR is divided into four fields, each 8bits wide: **flags, status, extension and control**. In current designs the extension and status fields are reserved for future use.

Control Field: The control field contains the processor mode, state, and interrupt mask bits.

Processor Modes: The processor mode determines which registers are active and the access rights to the CPSR register itself. The least significant 5 bits in the CPSR determines the mode.

Each processor mode is either privileged or non-privileged.

- A privileged mode allows full read-write access to the CPSR. Conversely, 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: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one non-privileged mode (user).
- The processor can change mode by either writing directly in to the control field (b0-b4) when it is in a privileged mode or when exceptions or interrupts happens.

State and Instruction Sets: The state of the core determines which instruction set is being executed. There are three instruction sets: ARM, Thumb, and Jazelle.

- The ARM instruction set is only active when the processor is in ARM state. Similarly 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. Jazelle executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes.
- The Jazelle J (bit 24, which falls in flag field) and Thumb T bits in the CPSR reflect the state of the processor.

- When both J and T bits are 0, the processor is in ARM state and executes ARM instructions. If J=1 then the core is in Jazelle state and T=1 then the core is in Thumb state.

Interrupt masks: ARM7 entertains two kinds of hardware interrupts interrupt request (IRQ) and fast interrupt request (FIQ). Bit 6 and Bit 7 of CPSR is used to mask these interrupt requests.

- If I=1 then IRQ is disabled and if F=1 FRQ is disabled.
- When processor mode changes the exception or interrupt handler makes IRQ bit 1 to disable further interrupt requests.

Flags: The flags field contains the condition flags.

Some ARM processor cores have extra bits allocated. For example, the J bit (24), which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8 bit java code.

The bits are described as given below along with condition to set the bits.

V-oVerflow: the result causes a signed overflow

C-Carry: the result causes an unsigned carry

Z- Zero: the result is zero, frequently used to indicate equality

N- Negative: bit 31 of the result is a binary 1

Q (bit 27)-Saturation: the result causes an overflow and/or saturation when extended instructions are used. eg: QADD

Module 5

9. ***Module 5***
   a. ***Explain the following instruction of ARM with example:***

Sl No	Instruction	Desciption	Syntax	Operation
1	MVN	Move negate	MVN{S}{cond} Rd, N	Rd= !N(one's complement of N) N : unchanged
2	RSB	Reverse Substract	RSB{S}{cond} Rd, Rn,N	Rd=Rn+!N , Rn,N : unchanged
3	ORR	Logical OR	ORR{S}{cond} Rd, Rn,N	Rd=Rn+N Rn,N : unchanged
4	MLA	Multiply and accumulate	MLA{cond}{S} Rd, Rm, Rs, Rn	Rd =(Rm∗Rs)+Rn
5	SMULL	signed multiply long	SMULL{cond}{S} RdLo, RdHi, Rm, Rs	[RdHi,RdLo]=Rm∗Rs

6	LDR	load signed/unsigned Byte/Halfword/Word into a register	LDR{cond}{B} Rd,addressing	Rd <- mem[address]
7	SWP	SWP{cond} Rd,Rm,[Rn]	swap a word between memory and a registe	tmp=mem32[Rn] mem32[Rn]=Rm Rd=tmp
8	SWPB	SWPB{cond} Rd,Rm,[Rn]	swap a byte between memory and a registe	tmp=mem8[Rn] mem8[Rn]=Rm Rd=tmp

b. Explain various formats of ADD instruction based on various operands of ARM7

ADD{S}{cond} Rd, Rn, Operand2; Rd= Rn+operand2

S-is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation.

Cond-is an optional condition code (see Conditional execution).

Rd-destination register
Rn- operand1 register
Operand2- Can be a register, immediate value or barrel shifted register

Based on operand 2 various syntax are possible.

- ADD Rd, Rn, Rm ; Rd= Rn+Rm
- ADD Rd, Rn, #imm ; Rd= Rn+imm
- ADD Rd,Rn,Rm, LSL #2 ; Rd= Rn+(Logically shift left twice(Rm))

<u>*Example:*</u>

Pre

r2=0x5,r3=0x2

ADD r1,r2,r3

ADD r4,r2,#4

ADD r5,r2,r3,LSL #2

Post:

r2=0x5, r3=0x2, r1=0x7, r4=0x9, r5=0xD

c. ***If r5=5 and r7=8 and using the following instruction, write values of r5 and r7 after execution of***
MOV r7,r5,LSL#2

Post
r5=0x5
r7=0x14 (r7=20)

10.

a. ***Explain SWI instruction of ARM***

A software interrupt instruction (SWI) causes a software interrupt exception. It provides a mechanism for applications to call operating system routines, like

- Read or write operation on hard disc
- Parallel port printing
- Invoke Serial or parallel communication.

SWIs allow the Operating System to have a modular structure, which means that the code required to create a complete operating system can be split up into a number of small parts (modules) and a module handler.

When the SWI handler gets a request for a particular routine number it finds the position of the routine and executes it, passing any data. No IRQ request is entertained while executing SWI instruction.

When the processor executes an SWI instruction, it sets the program counter pc to the offset 0x8 in the vector table. The instruction forces the processor mode to SVC, which allows an operating system routine to be called in a privileged mode.

Syntax: SWI {<cond>} SWI_number

SWI number is used to represent a particular function call or feature.

The SWI number is determined by SWI_Number = <SWI instruction>AND NOT (0xff000000); In the SWI instruction opcode the MSB two nibbles correspond to SWI and the rest to the SWI_ Number.

A code called the SWI handler is required to process the SWI call. The handler fetches SWI opcode using the address of the executed SWI instruction, which is calculated from the link register content, to obtain the SWI number. On execution of SWI, the following updates take place:

- LR_SVC=address of instruction following the SWI
- PC= IVT_Base address+0x8
- SPSR_SVC=CPSR

  Putting the processor into Supervisor mode switches out 2 registers r13 and r14 and replaces these with r13_svc and r14_svc.
- CPSR mode=SVC CPSR I=1 (mask IRQ interrupts)

b. ***Explain the syntax of SWAP instruction of ARM7***

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are “locked” together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

Syntax {cond}{B} Rd,Rm,[Rn]
{cond} two-character condition mnemonic.
{B} if B is present then byte transfer, otherwise word transfer
Rd,Rm,Rn are expressions evaluating to valid register numbers

SWP R0,R1,[R2] ; Load R0 with the word addressed by R2, and ; store R1 at R2.
SWPB R2,R3,[R4] ; Load R2 with the byte addressed by R4, and ; store bits 0 to 7 of R3 at R4.
SWPEQ R0,R0,[R1] ; Conditionally swap the contents of the ; word addressed by R1 with R0.

Example:
Pre
r0=0x2
r1=0x7
r3=0x9000
mem32[0x9000] = 0x10

SWP r0,r1,[r2]

Post
r0= mem32[0x9000] = 0x10
r1=0x7
mem32[0x9000] = 0x7

c. ***What are the salient features of ARM instruction set?***
- All instructions are 32 bits long.
- Most instructions execute in a single cycle.
- Most instructions can be conditionally executed.
- Three operand format
- Combined ALU and shifter for high speed bit manipulation
- 32 bit, 16 bit and 8 bit data types.

- Flexible multiple register load and store instructions
- Instruction set extension via coprocessors
- A load/store architecture –
    - Data processing instructions act only on registers
    - Specific memory access instructions with powerful auto -indexing addressing modes.