Fourth Semester B.E. Degree Examination, CBCS - June / July 2017

Microprocessors and Microcontrollers

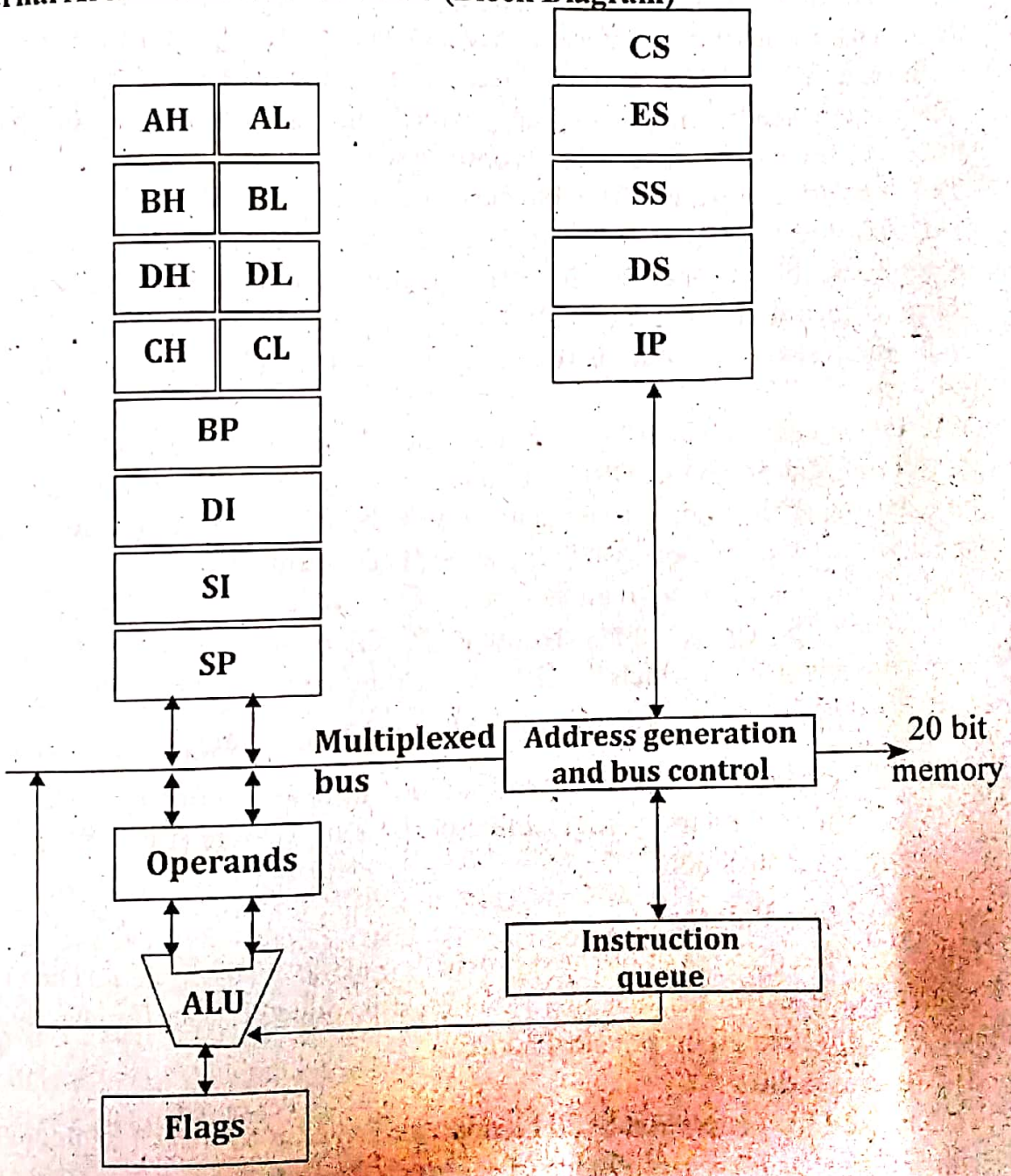
Time: 3 hrs. Max. Marks: 80

Note : *Answer any FIVE full questions, selecting ONE full question from each module.*

Module-1

1. a. Explain execution unit (EU) and Bus interface unit (BIU) of 8086 up with a neat diagram. (08 Marks)

Ans. Internal Architecture of 8086/8088 (Block Diagram)



It can be divided into two functional blocks 1. BIU and 2. EU

1. BIU (Bus Interface Unit)

It is interface to the outside world of 8086 (Between memory, I/O device and 8086).

Sunstar Exam Scanner

Scanned by CamScanner

It has 16 bit bi-directional databus and 20 bit unidirectional address bus. Its functions of BIU

a) It sends address of memory or I/O device.
b) It fetches instruction from Memory.
c) It reads data from Port/Memory.
d) It writes data into Port/Memory.
e) It Supports instructions queuing.
f) It Provides the address relocation facility.

To perform above function the BIU contains instruction queue, segment registers, instruction pointers, 20 bit address generation and bus control..

a) Instruction Queue : To speed up program execution the BIU fetches six instruction data/bytes ahead of time from memory and pass them to EU for further decoding and execution. When EU is ready for execution of the next instruction it fetches from the queue instead of memory (Memory fetching is slower than register fetching) and improves the overall speed of microprocessor. This feature is called pipelining.

b) Segment Register : The Physical address of 8086 is 20-bit wide to access 1M byte location. However 1M byte memory is divided (segmented) into 64-K byte blocks (16 segments) and only 4 segments are active at any time. To access these 16 segments we have (2^4) 4 bit address bus ($A_{16} - A_{19}$) and to access the individual segment location (64 K byte) we have (2^{16}) 16 bit wide address bus ($A_0 - A_{15}$).

The 4 segments which are active at any time are (segment registers) (i) Code Segment (CS) (ii) Data Segment (DS) (iii) Stock Segment (SS) and (iv) extra segment (ES) the starting address of the segments are stored in these segment registers. Starting means lowest addressed byte in the active code segment.

The function of segment registers are :

1. The CS register holds the upper 16-bits of the starting address of the code segment from which the BIU is fetching the instruction.
2. Similarly SS register is used to hold the starting address for the stack segment.
3. ES register and DS register are used to hold the 16-bits of the starting address of the two memory segments used for data storage (ES, DS). ES is used for string instruction.

c) Instruction Pointer :

It holds 16 bit address of the next instruction in the code segment. The value in IP is referred to as offset because this value must be added (offset from) to base segment in CS to produce 20 bit physical address.

d) Address Adder :

This is used for generation of 20 bit address. Physical address = Segment Register. 10h + Offset or PA = Base Value of register x 16 D + offset.

The BIU generates the 20 bit address as follows. Consider the IP contain address 123Ch. This is called offset value (or EA). Let the contents of CS be 2345h, this is called base value. This mean that the code segment starts from 23450h (Multiply

Sunstar Exam Scanner

Scanned by CamScanner

2345h by 16 ie 10h) and we want to fetch an instruction from a memory which is 123Ch bytes away from the base value of code segment (2345h), the BIU adds up 23450h the base address and 123Ch offset, using the address adder in the BIU and sends out the sum, 2468Ch as the physical address (PA) on the address bus.
16 bit EA or offset is IP = 123Ch
\20 bit PA is 2468Ch. (2345*10+123C = 23450+123C = 2468Ch)
This is represented as 2345 : 123C
Segment base : Offset

2. Execution Unit

The EU tells the BIU, which memory location to access along with required operation. It consists of the following units. 1) control circuitory. 2) Instrcution decoder. 3) ALU 4) Registers (GPSR) 5) Pointer and index register 6) Flag register.
The instructions are fetched from the instruction queue, and stored in the decoder and translates in the sequence of actions which the EU carries out.
The arithmetic and logical operation are performed in ALU. All the actions are controlled by control circuitry, which generates appropriate signals at fixed intervals of time.

a) Flag Register (Only Nine FLAGS of 16-bit microprocessor)

A flag is a Flip Flop which indicates some condition produced by the execution of an instruction or controls certain operations of EU. A 16-bit Flag register in the EU contains 9 active flags. These 9 Flags can be divided into 2 types a) condition flags b) control flags.
Condition flags are also called status flags since they shows the status of any Arithmetic operation. The description is shown in clause 1.8 page 37.

								←8085 compatible flags→							
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

X-Means not used

Condition Flags - CF, PF, AF, ZF, SF, OF

There are 6 condition flags that are set due to some condition generated, as a result of last instruction executed. 3 control Flags - TF, IF, DF to control CPU operation.

b) General Purpose Registers (GPRs) : 8086 has Four, 16 bit GPRs or Eight, 8 bit registers i.e. AX, BX, CX, DX or AH, AL, BH, BL, CH, CL, DH, DL.

AX : It is also called accumulator. Instructions like divide, rotate, shift, multiply etc. assume one of the operand to be in the accumulator. Here AX can be used as a complete word or a byte register (AL). Instructions like 16 bit multiply one of the operands must be in AX and the other can be a word in memory or a register. The resulting 32 bit product is stored in AX and DX, with DX storing MSB and AX storing LSB.

i.e. (DX) + (AX) ← AX x (XXh)
MSB LSB

Here AX with 16 bit capacity can be viewed as group of two 8 bit AH+AL. In some

Scanned by CamScanner

instruction like 8 bit multiply one of the operands must be in AL, the other can be byte in memory location or in another 8 bit register. The resulting 16 bit product is stored in AX with AH storing the MS byte and AL storing the LS byte.

AX ←(AL) x (XX)

Similarly for division operation it is used to store quotient, reminder. If a byte is divided AL stores quotient AH stores reminder. In case of division of a word AX stores quotient DX stores reminder. Also instructions like OUT, IN one of the operand is in AX or AL. AH also used to store function number whenever we call interrupts (DOS) like mov ah, 4ch, int 21h.

BX : It is used as a base register this means by default it is assumed to contain the offset of memory region within data segment.

Eg.: To read a word from the memory location 40 in BX to AX, instruction is mov BX, 40

mov AX, [BX] ; move the contents at location pointed table BX to AX.

It is also used to store base address of look up table.

Eg. : mov bx, offset table ; or lea bx, table

mov AL, 05h

XLAT ; move the data at location 05h in table to AL.

CX : In some loop instruction it is assumed to work as a counter which contains number to repeat instructions many times.

```
Eg.:    mov CX, 100
  REPTG :
  some instructions
  loop REPTG ; CX ←CX - 1.
```

DX : In i/p, o/p instruction it is assumed to contain the port address. The 8086 receives into or sends out data from AX or AL depending on a word or a byte operation. In these instructions if the address is greater than FFh it has to be given as contents of DX register. It acts as a data register.

```
Eg.:    mov dx, 0FFF3h
  in al, dx
```

Here the value at FFF3h is moved to al.DX is also used in division and multiplication of 16 bit data as a default register to store remainder and MSB of result.

Functions of GPR's

AX : To multiply word, divide word, I/O operation to act as source or destination.

AL : Byte multiply, divide (stores quotient)

Ah : Byte multiply, divide (to store reminder)

BX : Translate (XLat), Base Register, **CL-** Shift, Rotate for count in loop instruction.

DX - Word multiply, divide, Indirect I/O to store source or destination.

Pointers and index register (BP, SI, DI) : 8086 contains three 16 bit pointer and index registers. These 3 registers can also be used as GPRs. Their main purpose is to contain indices (offset) in stack segment, data segment and Extra segment respectively.

BP : We can use BP register instead of SP (stack pointer) for accessing the stack. In

Scanned by CamScanner

this case the 20 bit physical address is calculated from BP and SS.

SI : It is used to hold offset of data word in the data segment. In this case the 20 bit physical address is calculated from SI and DS.

DI : This is pointing to the ES where strings are stored. String instructions always use DS and DI to determine 20 bit physical address.

Stack Pointer (SP) : In 8086, 64 K memory is used as a stack. SP contains the 16 bit offset from the start of the segment to the top of the stack. The physical address is produced by adding the contents of SP to SS base address.

Segmented Memory (Without overlapping) in 8086/8088

The total memory capacity of 8086 is 1 MB and is further divided into 16 segments each of 64K, to enable programmer to access 1 MB memory using any general purpose registers whose capacity is 16 bit.

Advantages of segmentation

1. It allows to access the memory (of) upto 1 MB even though the address associated with each instruction is only 16 bit

b. Explain the different addressing modes used in 8086 up with suitable example. (08 Marks)

Ans. **x86 Addressing Mode**

Any instruction involves zero, one or two operands. It may be register, implied register, immediate data or memory. Method of specifying the operand(s) in an instruction or effective address of operands in memory is referred to as addressing mode.

There are 3 basic types:

(a) Data addressing mode - 5 types
- 1) Register addressing
- 2) Immediate addressing
- 3) Memory addressing
  - (i) Memory direct
  - (ii) Memory indirect
- 4) Implied addressing
- 5) Input, output port addressing
- 6) Relative addressing mode.

(b) Program memory addressing mode

(c) Stack memory addressing mode.

Scanned by CamScanner

Type	Instruction	Source	Address Generation	Destination
Register	MOV AX, BX	Register BX		Register AX
Immediate	MOV CH, 3AH	Data 3AH		Register CH
Direct	MOV [1234H], AX	Register AX	DS x 10H + DISP 10000H + 1234H	Memory address 11234H
Register indirect	MOV [BX], CL	Register CL	DS x 10H + BX 10000H + 0300H	Memory address 10300H
Base plus index	MOV [BX + SI], BP	Register SP	DS x 10H + BX + SI 10000H + 0300H + 0200H	Memory address 10500H
Register relative	MOV CL [BX + 4]	Memory address 10304H	DS x 10H + BX + 4 10000H + 0300H + 4	Register CL
Base relative plus index	MOV ARRAY [BX + SI], DX	Register DX	DS x 10H + ARRAY + BX + SI 10000H + 1000H + 0300H + 0200H	Memory address 11500H
Scaled Index	MOV [EBX +2 X ESI], AX	Register AX	DS x 10H + EBX + 2 X ESI 10000H + 00000300H +00000400H	Memory address 10700H

Note: EBX = 00000300H, ESI = 00000200H, ARRAY = 1000H, and DS = 1000H

Figure 10. 8086-Core 2 data addressing modes.

Sunstar Exam Scanner

Scanned by CamScanner

OR

2. a. **Explain all bits of flag register of 8086up with a neat diagram. Show the setting and resetting of flag bits with a suitable example.** **(06 Marks)**

Ans. **Flag Register and ADD Instrumentation**

Many Assembly language instructions alter bits of the flag register and some instructions will function differently based on the information in the flag register. After describing the bits of the flag register and their function, programming examples are given to demonstrate the use of the flag register.

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. A diagram of the flag register is shown below.

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
U = undefined
OF = overflow flag
DF = direction flag
IF = interrupt flag
TF = trap flag
SF = Sign Reg
ZF = Zero Flag
AF = Auxiliary carry Flag
PF = Parity Flag
CF = Carry Flag

Bits of the flag register

Below are listed the bits of the flag register that are used in x86 Assembly lan guage programming. A brief explanation of each bit is given. How these flag bits are used will be seen in programming examples throughout the textbook.

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 1's, the parity flag is set to 1; otherwise, it is cleared for odd parity (PF = 1 for even, PF = 0 for odd parity).

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 an arithmetic or logical operation is zero; otherwise, it is cleared. It is used for looping like JZ / JNZ

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.

Scanned by CamScanner

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, which are described..

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.

Now for setting and resetting the flags only CF, PF following instruction in each flag.

1) Carry flag (CF) CLC, CTC, CMC

CLC : CF ← 0

STC : CF ← 1

These two instructions do not take any operands but only sets or rests carry flag. During arithmetic operations if a carry is generated CF will be set (1) and to clear the carry flag can use CLC or to set the carry can use STC, to compliment CF use CMC

CMC ← ~ CF

2) Parity flag (PF)

PF = 1 for even number of one's in AL, PF = 0 for odd number of one's in AL. The instruction to check odd/even parity are JPO, TPE.

Similarly for direction flag we have

STD ; DF ← 1

CLD ; DF ← 0

and for interrupt flag (IF)

STI ; IF ← 1

CLI ; IF ← 0

However for other flag there is no specific instruction to set or reset, can use AND/OR instruction to clear/set Trap flag.

b. Write an assembly level program (ALP) to add two bytes of data stored at data 1 and data 2 and save the result in sum with comments. Indentify all the directives found in the program. **(06 Marks)**

Ans.

```
. model small
. Data                    ; Start of data segment
Data1 db 52h              ; declare Data 1 as byte
Data2 db 29h              ; declare Data 2 as byte
Sum   db  ?               ; declare sum
. code                    ; start of code
Main PROC  FAR            ; declare main as procedure
Start: MOV  AX, @ data    ; Initialise data segment
```


Sunstar Exam Scann

Scanned by CamScanner

```
        MOV DS, AX        ; to start of data
        MOV AL, Data 1    ; move Data 1 to AL Register
        MOV BL, Data 2    ; move Data 2 to BL Register
        Add AL, BL        ; Addition of AL + BL → AL
        MOV SUM, AL       ; move result to sum
        MOV AH, 4ch       ; terminate to DOS prompt int 21 h
Main ENDP                 ; Main procedure end
END start                 ; Physical end of program
```

The assembler directives used here are .model small means both code, data are placed in only one segment of 64 kB

```
.Data    ; Start of data segment
.code    ; Start of code segment
DB       ; define byte (8 bits)
PROC     ; Start of procedure say main
END P    ; End of procedure say main
END      ; Physical end of programme
```

c. Show the memory dump for the following data section or data segment. **(04 Marks)**

```
.DATA
ORG 0010H
DATA1  DB    25
DATA2  DB    10001001B
DATA3  DB    12H
ORG 0020H
DATA4  DB    '2591'
ORG 0030H
DATA5  DW    9, 2, 7, 0CH, 00100000B, 5
ORG 0040H
DATA6  DW    4 DUP(00H)
```

Ans. Showing memory dump (show data type and value stored at that location).

```
0010                                          ; ORG 0010h
0010  19
0011  89
0012  12
0020  32, 35, 39, 31                          ; ORG 0020h
0030  0009, 0002, 0007, 000C, 0020, 0005      ; ORG 0030h
0040  0004 [Fill 4 words with 00]        OR
0040  0008 [Fill 8 bytes with 00]             ; ORG 0040h
```

In data segment it will be stored as shown

```
DS : 0010  19  89  12  00  00  00  00  00
DS : 0020  32  35  39  31  00  00  00  00
DS : 0030  09  00  02  00  07  00  0C  00
```


Scanned by CamScanner

DS : 0038 20 00 05 00 00 00 00 00
DS : 0040 00 00 00 00 00 00 00 00

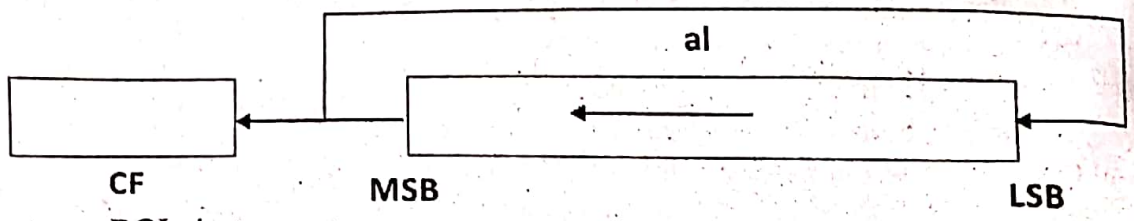
Module-2

3. a. Explain Rotate instructions with suitable example. **(06 Marks)**

Ans. ROTATE INSTRUCTION :

(i) ROL (Rotate Left) :

Rotate bits of word or byte left without CF. i.e. MSB to LSB and also to CF (MSB to CF).



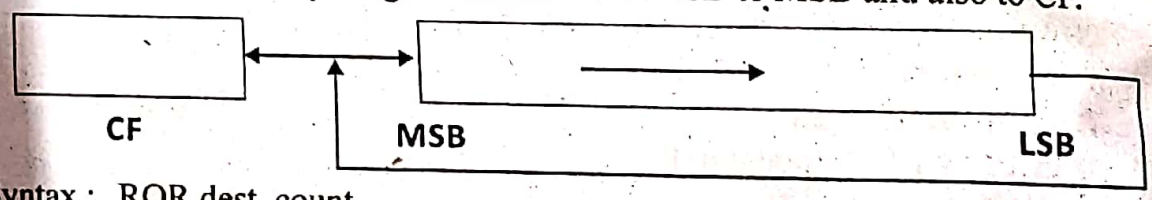
Syntax : ROL dest, count

eg. : ROL Al, 0lh; Al = 0101 1100 CF = 1

After the instruction Al = 10111000 CF = 0.

(ii) ROR (Rotate Right) :

Rotate bits of word or byte right without CF i.e. LSB to MSB and also to CF.



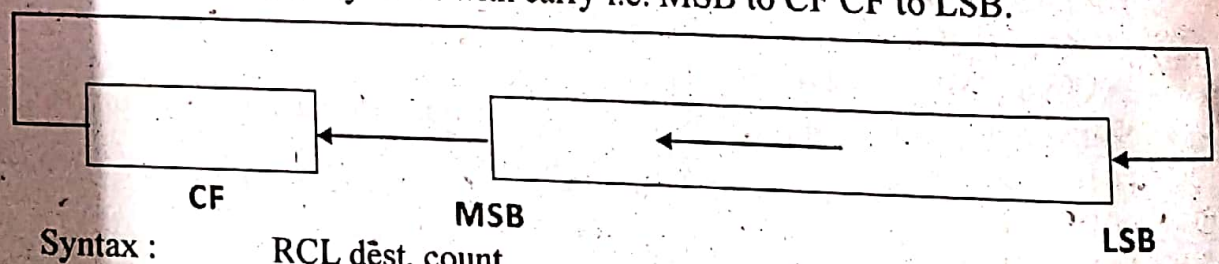
Syntax : ROR dest, count

eg. : ROR BX, 0lh if CF = 0, BX = 0011 1011 0111 0101

After the instruction CF = 1 BX = 1001 1101 1011 1010

(iii) RCL (Rotate Left with Carry) :

Rotate bits of word or byte left with carry i.e. MSB to CF CF to LSB.



Syntax : RCL dest, count

eg. : RCL al, 3 CF = 0 AL = 11010101 ; Initial State

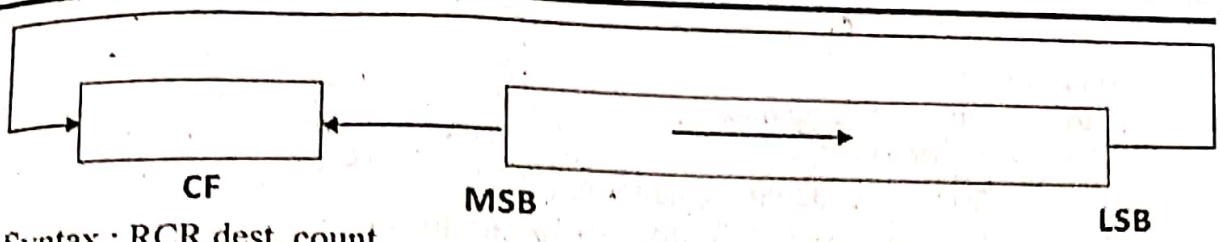
CF			
1	10101010	;	After first rotate
1	01010101	;	After 2nd rotation
0	10101011	;	After 3rd rotation

(iv) RCR (Rotate Right with Carry) :

Rotate bits of word or byte right with CF i.e. LSB to CF and CF to MSB.

Sunstar Exam Scanner

Scanned by CamScanner



Syntax : RCR dest, count

eg. : RCR Al, 01 CF = 0 Al = 11010101

After instruction CF = 1 Al = 01101010

LSB → CF, CF → MSB

b. With a suitable program show how a packed BCD value is converted to ASCII value. **(04 Marks)**

Ans.
```
. model small
. stack
. Data
Val_BCD  DB  29h        ; Packed BCD two digits per byte
Val_ASC1 DW  ?          ; ASC II value for 29 (0000 0010 0000 1001)
                        ; is 32h and 39h (0011 0010 and 0011 1001)
                        ; In memory it is stored like LSB | MSB
. code                  ; 39 | 32
Main proc FAR
Start; MOV AX, @data    ; Initialise Data segment
       MOV DS, AX
       MOV AL, Val_BCD  ; MOV Val_BCD to AL ← 29h
       MOV AH, AL       ; Save in (AH) ← (AL)
       AND AH, 0F0h     ; mask LN of AH AH ← 20h
       AND AL, 0Fh      ; mask HN of AL AL ← 09h
       MOV CL, 04h      ; move counter 04 to CL
       SHR AH, CL       ; SHift AW Right 4 times AH = 02h
       OR AX, 3030h     ; 0 Ring AX = 0209 with 30 30h
                        ; AX = 32 39h
       MOV Val_ASCI, AX ; Val_ASCI = 39, 32 in memory
       MOV AH, 4ch      ; Terminate to DOS
       int 21h
       Main ENDP        ; END of main procedure
END Start               ; Physical END of programme
```

c. Assume that there is a class of five people. With following grades: 69, 87, 96, 45, 75. Write an ALP to find the highest grade. **(06 Marks)**

Ans. Title program to find highest grade of 5 people

```
. model small
. data
Grades db 69, 97, 96, 45, 75
Highest db ?
```


Scanned by CamScanner

```
.code
Main PROC FAR
Start :     MOV AX, @ data
            MOV DS, AX
            MOV CX, 0005h ; counter in CX
            Lea BX, Grades ; Address of Grades to BX
            Sub AL, AL ; Initialise AL to 00.
AGAIN:      Cmp AL, [BX]  ; Compare 00 to 69
            JA NEXT       ; if AL > [BX] go to NEXT
            MOV AL, [SI]  ; else move bigger to AL
NEXT :      INC BX        ; take next byte
            loop AGAIN    ; Repeat 5 times
            MOV Highest, Al ; move highest in AL to memory highest
            MOV AH, 4ch   ; AL to Highest
            Int 21h       ; Terminate to DOS
  main ENDP     ; End of procedure main
  END Start
```

OR

4. a. Write on ALP that adds the following two multiword numbers and saves the result:
Data 1 = 548FB9963CE7H and
Data 2 = 3FCD4FA23B8DH **(08 Marks)**

Ans. Title Programme to add two multiword (More than a word say quad word - 4 words or 8 bytes or Less than or equal to 16 digits) Declare Data 1, Data 2 as DQ Define quad word that is 4 words. Since 8086 can add only maximum one word at a time, take CX counter as 4 words

```
. model small
. data
Data1 DQ 548FB9963CE7h; Data 1 EQU 548FB9963CE7h
Data2 DQ 3FCD4FA23B8DH; Since there are twelve digits in hexa
Data3 DQ 2 dup(0)
. code
Main Proc FAR
Start :     MOV AX, @ data
            MOV DS, AX   ; Initialise Data segment
            CLC          ; clear carry
            lea Si, Data1    ; move effective address of
            lea di, Data2    ; Data 1, Data 2, Data 3 to Si, di
            lea Bx, Data3    ; BX
            MOV CX, 04       ; counter to add 4 words
BACK:       MOV AX, [SI]  ; Add one word of Si to di by taking content of SI to AX
            Adc AX, [di]  ; along with carry
```


Sunstar Exam Scanner

Scanned by CamScanner

```
        MOV [BX], AX; move result to BX, BX+1.
                      ; BX points to Data 3
        Add Si, 02h   ; Take next word pointed to by SI
        Add di, 02h   ; and DI
        Add Bx, 02h   ; and BX
        Loop BACK     ; Repear 4 times
Main ENDP
```

NOTE : Program should have comments for all lines.

b. **Write an ALP to perform the following :**

i) Clear the screen

ii) Set the cursor at row 8 and column 5 of the screen.

iii) Prompt "There is a message for you from VTU : to read it enter Y. If the user enters 'Y' or 'y' then the message "Hello! All the best for your exams" 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. (08 Marks)

Ans. This program display message "There is a message for you from VTU" and display "Hello! All the best for your exams" if user enter 'Y' or y else message no more messages for you" if user enters other keys.

To move to next line use CR, LF i.e., 13, 10 or 0dh, 0ah?

```
. model small
. data
prompt 1 db   "There is a message for you from VTU to read it enter 'Y' or 'y'," '$'
msg      db   13, 10, "Hello! All the best for your exams"
prompt 2 db   13, 10, "No more messages for you", "$"
. code
Start :   MOV   AX, @data
          MOV   DS, AX            ; Initialise, data segment
          Call  clearscreen       ; To clear screen call procedure clearscreen
          Call  positioncursor    ; ve and to position the cursor call position cursor
          MOV   AH, 09h           ; To display prompt 1
          lea dX, prompt 1
          int 21h
          MOV AH, 07h             ; To accept Y or y
          int 21h
          cmp AL, 'y'             ; if your option is 'Y'
          JZ OVER
          cmp AL 'Y'
          JZ OVER
          lea dx, prompt 2        ; Else display
          MOV AH, 09h             ; Prompt 2
          int 21h
          JMP stop
```


Scanned by CamScanner

```
OVER :  MOV AH, 09h
        Lea dx, msg
        int 21h
STOP :  MOV AH, 4ch
        int 21h
        Clearscreen PROC
        MOV CX, 0000h
        MOV DX, 184Fh        ; 24 Rows, 79 Columns on screen
        MOV BH, 07h          ; Graphic mode
        MOV AL, 00h
        MOV AH, 06h          ; To clear screen
        int 10h              ; Interrupt for monitor
        Ret
        Clearscreen ENDP
        Positioncursor PROC
        MOV AH, 02h
        MOV BH, 00h
        MOV DL, 05h          ; 5th column
        MOV DH, 08h          ; 8th column, any Row/column can be separated
        Int 10h
        Ret
        Positioncursor ENDP
End start
```

Module-3

5. a. Explain handling of overflow problem arised in addition of signed numbers with a suitable example. **(06 Marks)**

When using signed numbers, problem arises This is the overflow problem. The CPU indicates the existence of the problem by raising the OF (overflow at position 11) flag, but it is up to the programmer to take care of it. The CPU under stands only 0's and 1's and ignores the human convention of positive and negative numbers. Now 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.

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
```


Sunstar Exam Scanner

Scanned by CamScanner

According to the CPU, this is –90 in decimal 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. 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 creat ed the overflow flag specifically for the purpose of informing the programmer that the result of the signed number operation is erroneous.

When the 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.

In other words, the overflow flag is set to If carry to sign bit and carry from sign bit are not same. If carry to sign bit is same as carry from sign bit no overflow. Hence OF = 0. In Example 4, since there is only a carry from D6 to D7 and no carry from D7 out, OF = 1. Examples 5, 6, and 7 give further illustrations of the overflow flag in signed arithmetic

Example :

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  ①0111 1110          OF=1, SF=O (positive), CF=1
```

According to the CPU, the result is +126, which is wrong. The error is indicated by the fact that OF= 1.

Here carry to sign bit = 0 carry from sign bit = 1. Hence over flow cannot represent -130 in 8 bit binary.

Example :

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  ①1111   1001
```

Carry to sign bit = 1, carry from sign bit = 1 (Equal) No overflow. Hence OF = 0, CF = 0 , and SF= I (negative); the result is correct since OF= 0.

Since sign bit = 1 take again 2'Sc to get the magnitude

11111001 →1'sc 00000110 + 1 = 00000111 = -7

Example :

Observe the results of the following:

Scanned by CamScanner

```
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).

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 8 and 9.

Avoiding erroneous results in signed number operations

To avoid the problems associated with signed number operations, one can sign extend the sign 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 :

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
                       ; greater than +32767
6E2F    0110 1110 0010 1111
+13D4   0001 0011 1101 0100
8203    1000 0010 0000 0011    = -32,253 incorrect!
        OF= 1, CF = 0, SF= 1
```

Since carry to sign bit = 1, carry from SB = 0. Hence overflow or under flow to check this take complement of sign bit. If it is 0 indicates over flow and if it is 1 indicates under flow.

Here sign bit = 1, complement of 1 is 0 hence overflow.

Example :

Observe the results in the following:

```
MOV     DX,542FH       ; 21,551-
MOV     BX,12EOH       ; +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 = o; SF= 0
```

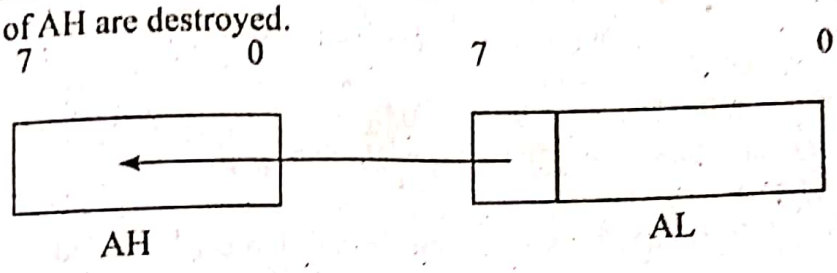

Scanned by CamScanner

Since carry to sign bit = 0. Carry fro sign bit = 0 Hence no overflow or under flow instruction for conversion.

INSTRUCTION FOR CONVERSION : 8 TO 16 OR 16 TO 32 bit

Convert Byte to word (CBW)

Convert Byte to word (CBW) will copy D7 (the sign bit) 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
```

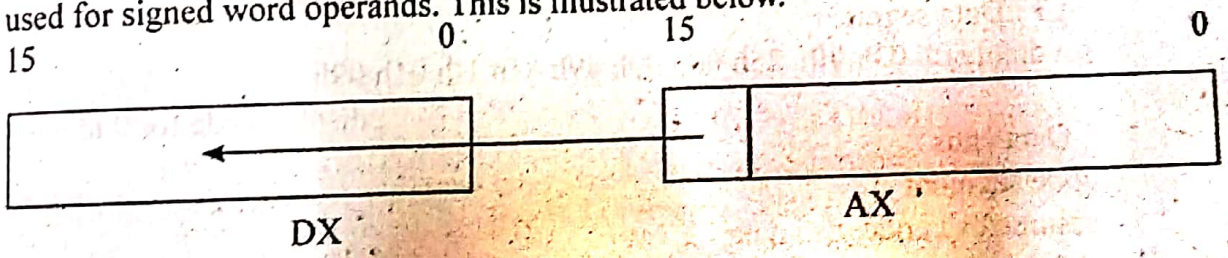
Similarly

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

Copy sign bit in AL to AH.

Convert word to double word

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
```

Here copy sign bit of AX to DX.

Similary (For Negative Numbers)

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

As can be seen in the examples above, CWD does not alter AX. The sign of AX is copied to the DX register. How can these instructions help correct the overflow error?. To answer that question, Example 10 shows Example 4 rewritten to correct the over flow problem.

In Example 10, 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. However, 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.

Scanned by CamScanner

The program in Example 10 works for addition of any two signed bytes.

b. Explain XLAT instruction with example. **(04 Marks)**

Ans. Xlat

Format : Xlat – Translate byte in al

It replaces a byte in AL register with a byte from a lookup table in memory. Register BX stores the offset of the starting address of the lookup table and al register stores the byte (index) from the lookup table. Here the instruction copies byte from address pointed to by [BX+AL] back to AL.

eg.1: If DS = 4000h, BX = 0050h, al = 05h

It copies data at (4000×10h+0050+05) = 40055h to al.

eg.2 :

```
Data Segment
table db 1, 4, 9, 16, 25, 36, 49    ; This declaration is also called
                                    ; lookup table.

Data Ends.
code    segment
        mov bx, offset table
        mov al, 05h ; 05 is moved to al
        xlat ; now al is having data 36
```

It is used to translate code from lookup table say 7 segment display code. Before translate al has a value say 00 after this instruction al will be having 7 segment display code for 0 that is 03h if 7 segment code is declared as under.

eg.3 : Data segment

```
sevdtable db 03h 9fh, 25h 0dh 99h 49h 41h 1fh 01h 09h    ;        7 segment
                                                         ; display code for 0 to 9 (BCD)

Data ends
code segment
assume cs : code, ds : data
again     : mov ax, data
            mov ds, ax
            lea bx, sevdtable
            mov al, 00              ; al contains 00h
            mov cl, 0ah
            mov dx, 7 segment dispaddress
L1          : mov ah, al            ; store al in ah
            XLAT                    ; now al as 03h, after XLAT
            out dx, al              ; dx contains port address (7 segment
            mov al, ah              display)
            inc al                  ; Take next code
            loop L1                 ; Repeat all instruction from Label L1
            mov ah, 4ch             ; till Cl is zero. DOS interrupt to
            int 21h                 ; Terminate the program.
```


Sunstar Exam Scanner

Scanned by CamScanner

code ends

c. **Explain 74138 decoder configuration to enable the memory address F0000H to F7FFFH to connect four 8k RAMS.** **(06 Marks)**

Ans. For 8kB RAMs it requires 2^{13} = 8K, 13 address lines

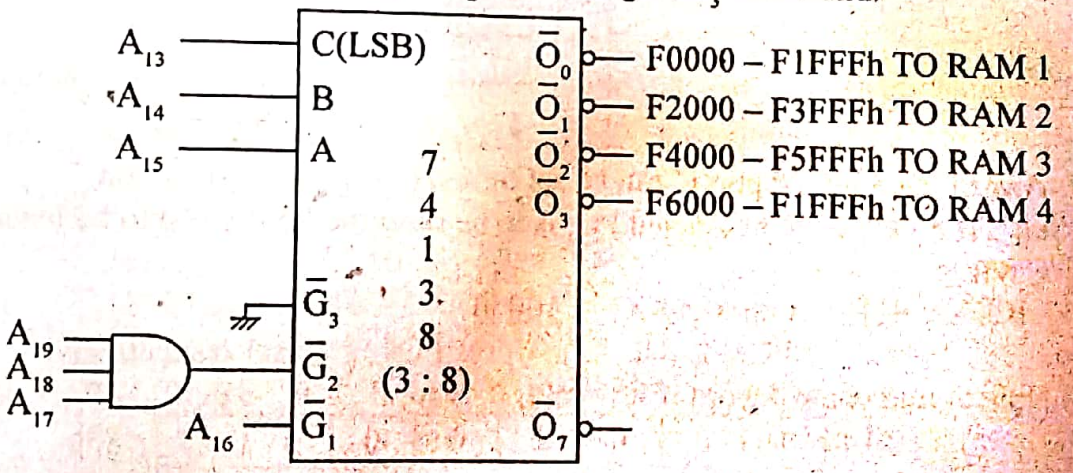
If starting address (SA) is F0000h Ending address (EA) for one (Forst) IC of 8K is F1FFFh

Similarly for 2nd IC of 8kB Starting address F2000h, EA = F3FFFh and for 3rd IC of 8K SA = F4000h, EA = F5FFFh and for 4th IC of 8K SA = F6000h, EA = F7FFFh

The addresses are shown below

		$A_{17}-A_{16}$	$A_{15}-A_{12}$	$A_{11}-A_8$	A_7-A_4	A_3-A_0
RAM 1, SA = F0000h	=	1111	0000	0000	0000	0000
RAM 1, EA = F1FFFh	=	1111	0001	1111	1111	1111
RAM 2, SA = F2000h	=	1111	0010	0000	0000	0000
RAM 2, EA = F3FFFh	=	1111	0011	1111	1111	1111
RAM 3, SA = F4000h	=	1111	0100	0000	0000	0000
RAM 3, EA = F5FFFh	=	1111	0101	1111	1111	1111
RAM 4, SA = F6000h	=	1111	0110	0000	0000	0000
RAM 4, EA = F7FFFh	=	1111	0111	1111	1111	1111

Now connect $A_0 - A_{12}$ to Address bus of IC $A_{13} - A_{15}$ to select lines of 74138 (A, B, C) and A_{16} to G_1(high) $A_{17} - A_{19}$ to $\overline{G}_2$ via AND gate $\overline{G}_3$ to Ground.



NOTE : We have considered 8K has one bank.

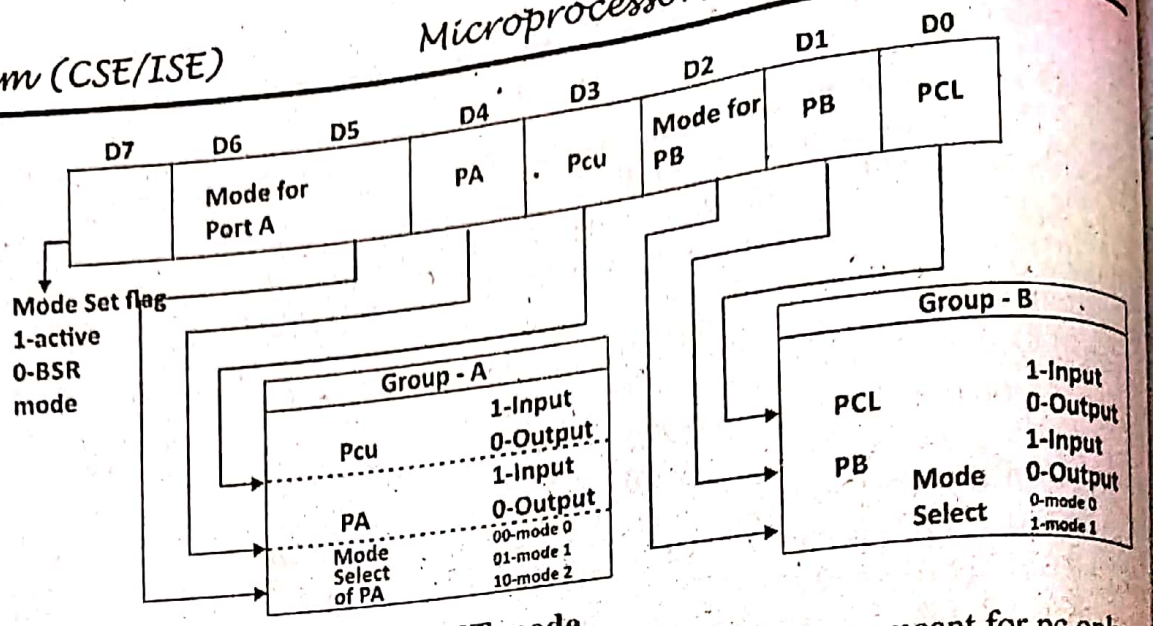
If Question says 8086 divide the 8K bank as (4K + 4K) one odd, one even bank and use only 12AL ($A_1 - A_{12}$) and A_0 to select even bank or odd bank A_0 = 0 for even bank A_0 = 1 for odd bank.

OR

6. a. **Briefly explain the control word format of 8255 in I/O mode and BSR mode. Find the control word if PA = out, PB = in, PCO - PC3 = in and PC4 - PC7 = out. Use port addresses of 300H - 303H for the 8255 chip. Then get data from port B and send it to port A.** **(08 Marks)**

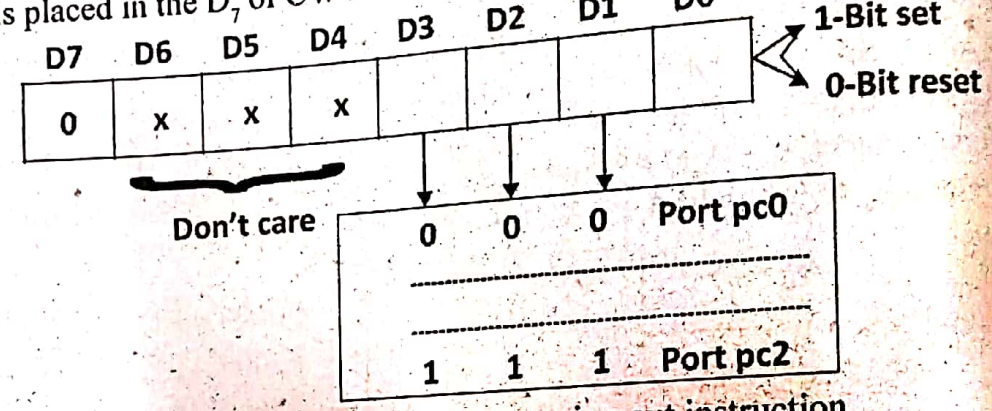
Ans. **Control Word Format of 8255 (mode 0)**

Scanned by CamScanner



BSR mode : BIT SET and RESET mode

If a 0 is placed in the D_7 of CW it is single bit set/reset features meant for pc only.



Any of the 8 bits of port C can be set or reset using out instruction.

eg. : If bit PC_3 is to be set and PC_4 is to be reset the BSR word to be loaded in to the CWR is

0 XXX <u>011</u> 1 and 0 XXX <u>100</u> 0

PC3 ↑ Set PC4 ↑ Reset respectively.

Now to find control word if PA = out, P_B = in, $P_{co} - P_{c3}$ = in, $P_{c4} - P_{c7}$ = out

1	0	0	0	0	0	1	1	=83h

Program to get data from P_B and send to P_A

```
8255C   EQU   300h
CW      EQU   83h
MOV     dx, 8255C+3     ; Address of CWR to dx
MOV     AL, CW          ; Initialise CWR by moving 83h to AL
OUT     dx, AL          ; to make PA = out, PB = IN
MOV dx, 8255C+1
IN AL, dx               ; Read PB
MOV     dx, 8255C       ; PA address
OUT     dx, AL          ; Send PB to PA
```


Sunstar Exam Scanner

Scanned by CamScanner

b. Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH and 52H.
i) Find the checksum byte
ii) Perform the checksum operation to ensure data integrity.
iii) If the second byte 62H had been changed to 22H. Show how checksum detects the error. (08 Marks)

Ans. (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.
```

Module-4

7. a. Differentiate between RISC and CISC processors. (06 Marks)

Ans.

	RISC (Reduced instruction set computing)	CISC (Complex instruction set computing)
(1) Instruction	Limited and simple instruction set about 58	more and complex instruction set
(2) Size of instruction	All instruction are of fixed length 32 bit (4 bytes)	Instruction are 1 to 18 bytes(Variable length)
(3) Registers	Large number of General purpose registers eg: ARM7TDMI had 37 registers	limited number of registers say 14 in 8086

Scanned by CamScanner

(4) Load-store Architecture	Separate load and store instruction to transfer data between register and memory	Can use move instructor for transfer data between register and memory
(5) Operation	Between Register - Register	Between Register to memory or memory to Register along with Register to Register
(6) Key unit	Emphasis is on compiler (SW)	Emphasis is on Processor (HW)
	Compiler → Code generator → Processor Eg : RISC - ARM 7TDMI, SPARC, MIPS R40000	Compiler → Code generator → Processor Eg : X86 with some RISC features

b. Explain ARM core data flow model with a neat diagram. **(06 Marks)**

Ans. ARM Core data flow model

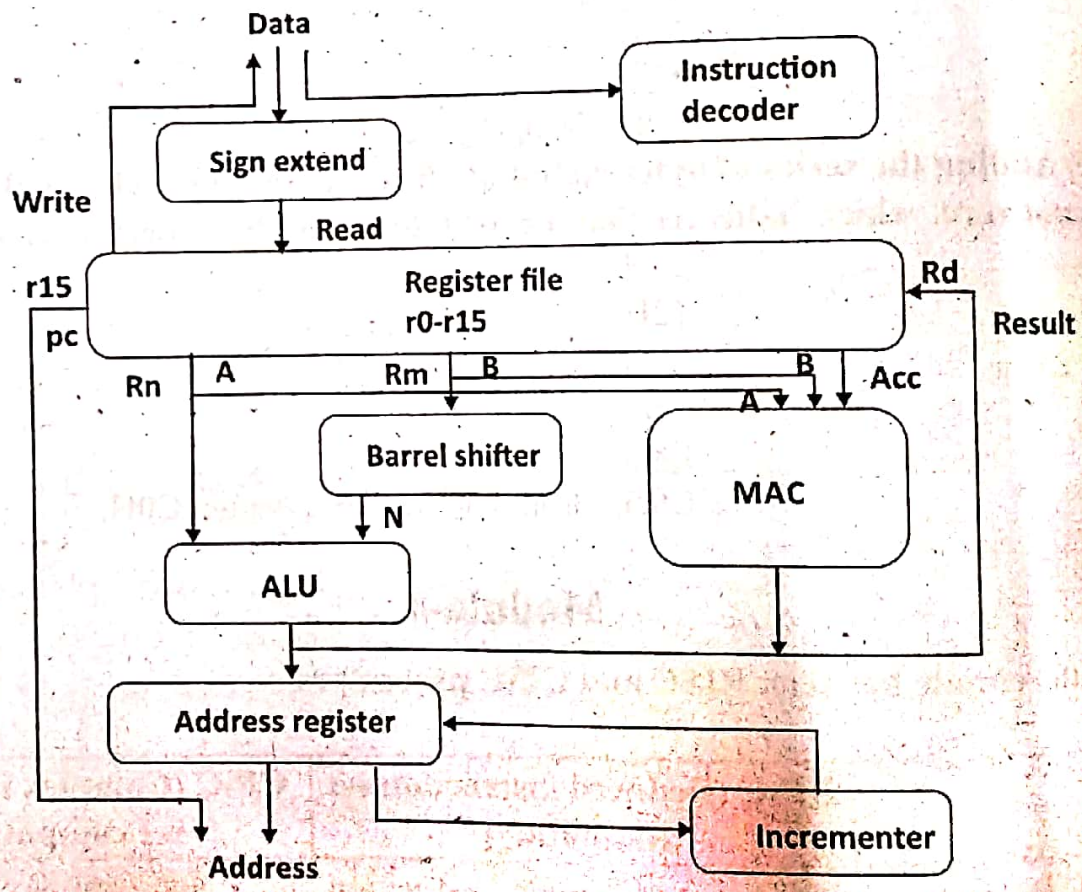


Figure : ARM core dataflow model.

The figure shows the components that makeup on ARM core. A programmer can think of an ARM core as functional units connected by data buses. The boxes represent either an operation unit or storage area.

Data Instruction or data item enters the processor core through the data bus. Instruction decoder translates instructions before they are executed.

Sunstar Exam Scanner

Scanned by CamScanner

The ARM Processor uses load-store architecture load instructions copy data from memory to registers in the core and store instructions copy data from registers to memory. The data processing is carried but only in register and no data processing instructions that directly manipulate data in memory.

The operational units are (8 types)

1. **Register File:** A storage bank made up of 32 bit register (r0 r15) where data items are placed. Since ARM core is 32 bit all registers are holding signed or unsigned 32 bit values.
2. **Sign Extended Hardware** converts 8 bit or 16 bit to 32 bit values as they are read form memory and placed in a register.
3. **ARM Instructions** can take three operands two source registers R_n and R_m and a single destination register R_d Source operands (Rn and Rm) are taken from register file using buses A and B respectively. The result in Rd is written back to register file using result bus.
4. **The ALU (Arithmetic Logic Unit) or MAC (Multiply-Accumulate Unit)** takes register 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 uses ALU to generate an address to be held in address register and broad cast on the address bus.
5. **Barrel Shifter (Shift operation along with ALU):** The register Rm 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.
6. **Incrementer:** For load and store instructions the incremeter updates the address register before the core reads or writes the next register value form or to the next sequential memory location

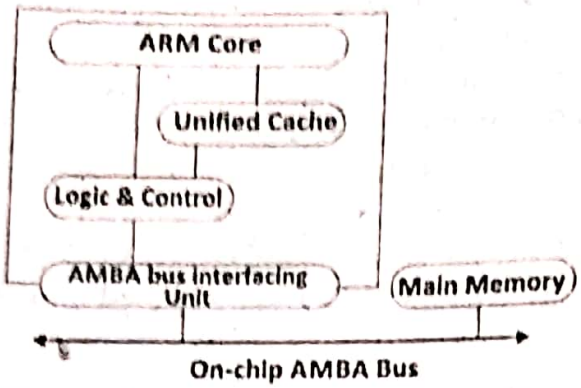
Note: ARM Processors was developed at ACRON computer limited between 1983 to 1985 later it is named as advanced RISC Machines Limited or ARM Limited in 1990. ARM Stood for Acron RISC Machine.

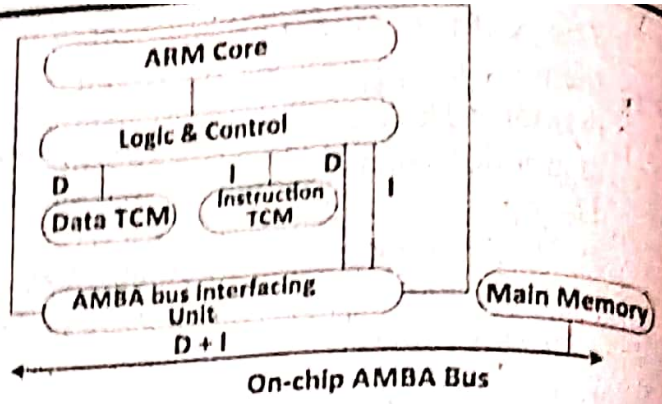
c. **Discuss briefly how coprocessors can be attached to ARM processor. (04 Marks)**

Ans. These are the standards components placed next to the ARM core to improve performance manage resources and provide extra functionality and are designed to provide flexibility in handling particular applications. The three components wraps around the core are **cache** and **tighty** coupled memory, memory management unit and coprocessor interface.

Cache and Tightly Coupled Memory:

Scanned by CamScanner



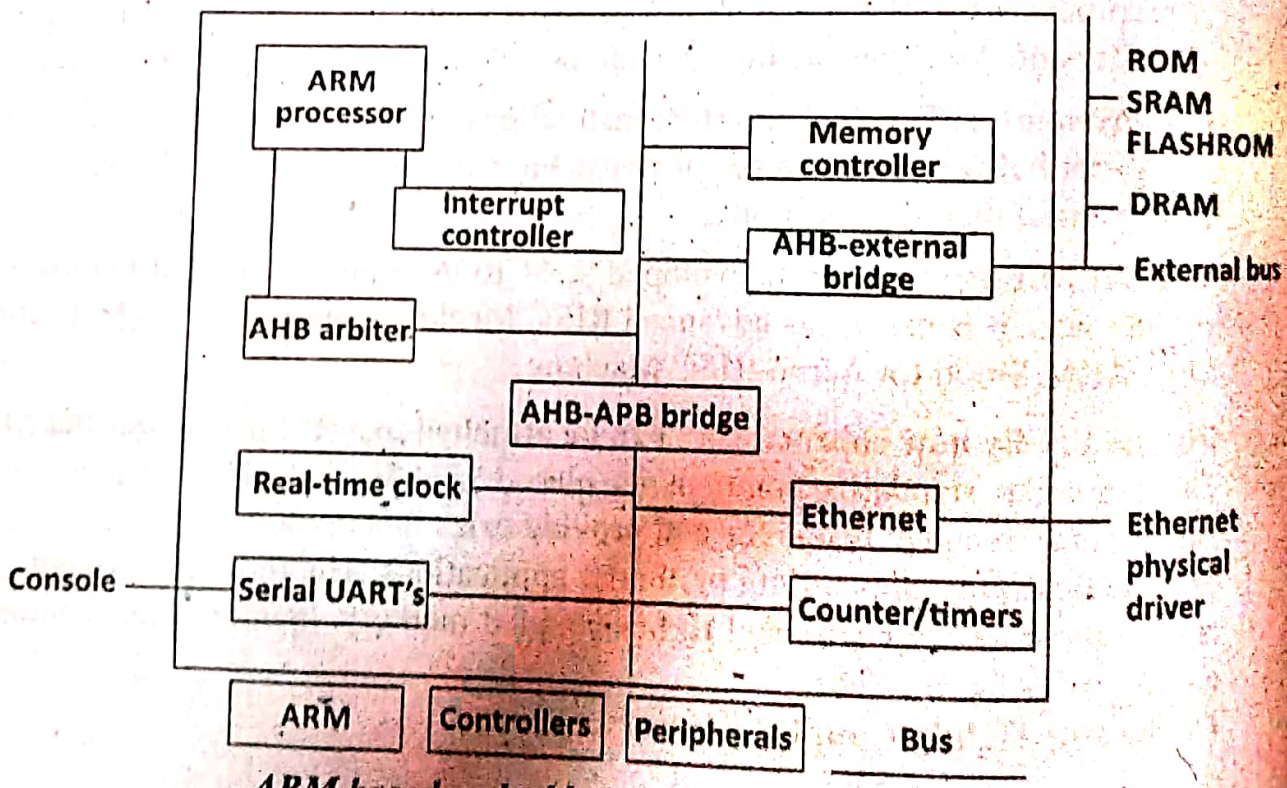


The cache is a fast memory placed between main memory and core, so that processor core can run without having to wait for data from a slow external memory. For real time systems the code execution is deterministic (Time taken for loading and storing instructions or data must be predictable). This is achieved by using tightly coupled memory (TCM). TCM is a SRAM and guarantees the clock cycles required to fetch instruction or data is critical for real time algorithms requiring deterministic behaviour.

OR

8. a. Explain the architecture of a typical embedded device based on ARM core with a neat diagram. (08 Marks)

Ans. Figure shows ARM_based embedded device with Four main components ARM controllers, peripherals, BUS, form a microcontroller



ARM based embedded device, a microcontroler

a. ARM Processor: This controls the embedded device. It has a core plus other components, like memory management and caches to interface it with a bus.

b. Controllers: These are for control and coordinate important functional blocks in

Sunstar Exam Scanner

Scanned by CamScanner

the system. Here it uses interrupt and memory controllers.

c. Peripherals: To provide input-output functions external to chip.

d. Bus: This is to communicate between various parts of the system.

ARM Bus Technology:

Embedded systems use an on-chip bus that is internal to the chip which allows different peripheral devices to be interconnected with ARM core.

There are two different classes of devices attached to the bus. The ARM processor core is a bus master which is a logical device capable of initiating a data transfer with another device across the same bus. Peripherals tend to be bus slave which is logical device capable only of responding to a transfer request from a bus master device.

The bus has two architecture levels. The first is a physical level that covers the electrical characteristics and width (16, 32 or 64 bits) and second deals with protocol. (The rules governs the communication between the processor and peripherals.)

The ON-Chip bus used for ARM Processors is the Advanced Microcontroller Bus Architecture (AMBA). The first AMBA buses introduced were ARM system bus and ARM Peripheral Bus (APB) later it introduced ARM High Performances Bus (AHB). 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 ply 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.

AHB bus runs at higher clock speeds and supports 64 and 128 bits.

Peripherals:

To interact with outside world the embedded systems need some form of peripheral device to perform input and output functions for the chip by connecting to other devices that are off-chip. Each peripheral device performs a single function and may reside on-chip. All ARM Peripherals are memory mapped the address of these registers is an offlet from a specific peripheral base address. The controllers are specialized peripherals. Two important types of controllers are memory controller and interrupt controllers.

Memory Controllers:

This Controllers Connect different types of memory to the processor bus On power-up memory controller is configured in hardware to allow contain memory devices to be active and intialization code to be executed. Some memory devices must be set up by software for example in DRAM it has to be set up the memory timings and refresh rate before it can be accessed.

Interrupt Controllers:

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 register.

There are two types of interrupt controllers the standard interrupt for external device controller and vector interrupt controller for prioritize the interrupts and simplifies the determination of device caused the interrupt.

Scanned by CamScanner

b. Explain the concept of pipeline and interrupts used in ARM processor. **(08 Marks)**

Ans. Pipeline:

Pipeline is a mechanism of RISC processor used to execute an instruction by Fetching the next instruction while other instructions are being decoded and executed in a 3 stage pipe line (Fetch, decode, Execute). When one instruction (Say instruction 1) is under execution, the next two instruction are under decoding (instruction 2) or Fetched (Instruction 3).

Fetch	Decode	Execute

Fetch loads an instruction from memory to instruction register.

Decode is to identify the operation and Fetch operands (identifies the operation, operands) . **Execute** is to processes the instruction and writes the result back to a register. Each takes one clock cycle to complete.

Eg : Add Ld, Ln, Lm
Sub Ld, Ln, Lm
Cmp Rn, Rm

In the clock cycle 1, Add is fetched by core from memory. In clock cycle 2 the core fetches sub and decodes Add. In the clock cycle 3 the Add is executed, Sub is decoded and cmp is Fetched. This procedure is called filling the pipeline.

Instruction	CC1	CC2	CC3
Add	F	D	E
Sub	-	F	D
Cmp	-	-	F

CC = Clock Cycle F = Fetch D = Decode E = Execute

The 3 stage pipeline is implemented in ARM7 Processor (7 = 3 + 4).

The 5 stage pipeline is implemented in ARM 9 processor (9 = 5 + 4)

The Five stages are F, D, E, M, W (M = Memory, W = Write back).

F	D	E	M	W

The 6 stage Pipeline is implemented in ARM 10 (10 = 6 + 4).

The six stages are (Same as 5 stage plus I (Issue)).

F, I (Issue), D, E, M, W.

F	I	D	E	M	W

ARM9 is 13% more through put compared to ARM7 and ARM 10 is 34% more through put compared to ARM 7 but with higher latency cost.

As the pipeline length increases the amount of work done at each stage reduces which allows the processor to attain higher operating frequency. This increases the performance, the latency of system also increases because it takes more cycle to fill, the pipeline before the core can execute an instruction. This increased pipeline length means there can be data dependency between stages and can be reduced by instruction scheduling.

Sunstar Exam Scanner

Scanned by CamScanner

Exception/ Interrupt and The Vector Table:

When an exception or interrupt occurs, the processor sets the PC to a specific memory address available in Vector table. The vector table stores the addresses of instructions that branch to specific routines to handle an exception or interrupt. The address 0x00000000 is reserved for vector table. Some processor the vector table is at 0 x ffff0000.

Also when exception or interrupt occurs the processor suspends normal execution and starts loading instructions from the vector table. Vector table entry contains a form of branch instruction pointing to the start of a specific routine.

Vector Table:

Exception / Interrupt	Shorthand	Address	High Address
Reset	RESET	0 × 00000000	0 × ffff0000
Undefined instruction	UNDEF	0 × 00000004	0 × ffff0004
S W Interrupt	SWI	0 × 00000008	0 × ffff0008
Prefetch abort	PABT	0 × 0000000C	0 × ffff000C
Data abort	DABT	0 × 00000010	0 × ffff0010
Reserved	-	0 × 00000014	0 × ffff0014
Interrupt Request	IRQ	0 × 00000018	0 × ffff0018
Fast Interrupt Request	FIQ	0 × 0000001C	0 × ffff001C

Reset Vector: is the location of first instruction is executed by processor when power is applied. This instruction branches to the initialization code.

Undefined Instruction Vector: is the location used when the processor cannot decode an instruction.

Software Interrupt Vector: is called when execute SWI instruction and is used to invoke an operating system routine.

Prefetch Instruction Abort Vector is called when the processor attempts to fetch an instruction from an address without the correct access permissions.

Data Abort Vector: Similar to a prefetch abort and is raised when an instruction attempts to access data memory without the correct access permission.

Interrupt Request Vector: is used by external hardware to interrupt the normal execution flow of the processor. It is used only if IRQ's are not masked in the CPSR.

Fast Interrupt request vector: is similar to interrupt request but is reserved for hardware requiring faster response time. It can only be raised if FIQ's are not masked in the CPSR.

Module-5

9. a. **Explain the following instructions of ARM processor with suitable example.**
i)MLA ii)QADD iii) SMULL iv) LSL. **(08 Marks)**

Scanned by CamScanner

Ans. i) MLA :

MLA - Multiply and add to another register, place the result in destination register

MLA {<cond>}{S} Rd, Rm, Rs, Rn

Rd = (Rm * Rs) + Rn

MLA r4, r3, r2, r1, r4 = (r3 × r2 + r1)

ii) QADD :

Signed saturated addition of 32 bit

QADD,<cond> Rd, Rm, Rn; Rm + Rn → Rd

QADD for Saturated addition. Normal ARM arithmatic instructions wrap around when over flow

Eg 0 × 7FFFFFFF + 1 = – 0 × 80000000 over flow

Hence cannot use ADD instruction use QADD

eg if cpsr = nzcvqiFt – SVC initially and

r0 = 0 × 00000000

r1 = 0 × 70000000

r2 = 0 × 7fffffff

QADD r0, r1, r2, Add r_1 with r_2 and place in r_0

After execution

cpsr = nzcvQiFt – SVC

r0 = 0 × 7fffffff

Q bit (bit 27 of + cpsr) has been set indicates saturation (indicated by Q)

Note : This instruction is available in ARMU5E extensions.

iii) SMULL :

Signed multiply long (R_{dHi}, R_{dLo}) = R_m * R_s

multiply R_m with R_s (64 bit) it takes 4 operands

Syntax : SMULL R_{dLo}, R_{dHi}, R_m, R_s.

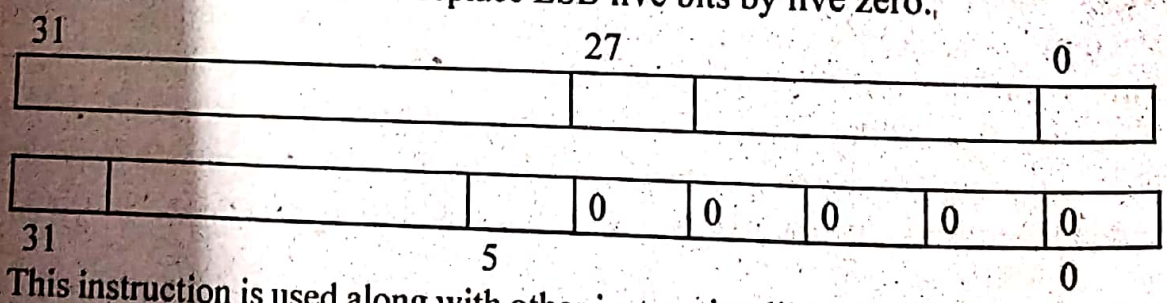
v) LSL :

LSL - Logical Shift Left - XLSLY ; X << Y

; x is shifted left logically Y times.

LSL shift logical left by a count set

LSL #5, shift left 5 bits and replace LSB five bits by five zero.



This instruction is used along with other instruction like MOVS r0, r1, lsL#1, r1 shift left once and move to r0.

b. Write an ALP to copy a block of data (Block 1) to another block (Block 2) using ARM instructions. **(08 Marks)**

Ans.

```
Area Blk copy, code, Readonly
SWI_Write c    EQU    &0        ; output char in r0
```

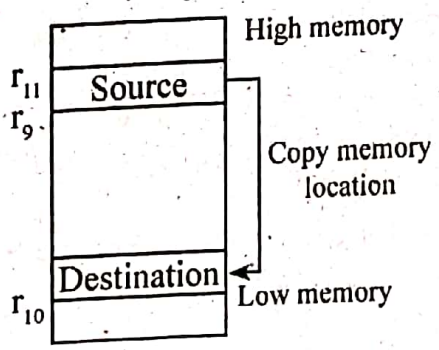

Sunstar Exam Scanner
Scanned by CamScanner

```
        SWI_Exit   EQU   &11      ; finish program
        ENTRY
START   ADR   r1,  A block        ; r1 → A block
        ADR   r2,  B block        ; r2 → B block
        ADR   r3,  T1 END         ; r3 → END
LOOP 1  LDR   r0,  [r1], #4       ; get Ablock Ist word in r1 to r0
        STR   r0,  [r2], #4       ; copy to B block in r2 from r0
        cmp   r1, r3              ; is finished if r1 = r2
        BLT   loop1               ; if not do more
LOOP2   ADR   r1,  Bblock         ; r1 → Bblock
        LDRB  r0,  [r1], #1       ; get next byte
        cmp   r0,  #0
        SWINE      SWI_Write C    ; if not end
        BNE        loop 2         ; loop back
        SWI        SWI_Exit       ; Finish
Ablock 1 "This is right msg" & 0a, &0d, 0
T1 END Align
Bblock 2 "This is the wrong", 0
END
```

NOTE The above problem can be solved using Load-store multiple instructions with block memory copy.

Assume r9, r10, r11 are being set up before

```
; r9     points to start of source data
; r10    points to start of destination data
; r11    points to end of source
ADR     r9, source start
ADR     r10, dest
ADR     r11, source end
loop    LDM1A   r9!,  {r0–r7}; load 32 bytes from source pointed to by r9 to r0 to
                      r7 and up-date r9 pointer
        STM1A   r10!  [r0–r7]; Store 32 bytes to destination pointed to by r10
                      from r0 to r7 and update r10 pointer to point to next
                      destination location and store them
        cmp     r9, r11          ; have we reached end
        BNE     loop             ; Repeat with updated values of r9 and r10.
```



Scanned by CamScanner

OR

10. a. Write an ALP using ARM instructions that calls subroutine fact to find factorial of a given number. (08 Marks)

Ans. Calling subroutine is out off syllabus
However sample code is given below

```
AREA fact-seg, CODE, READONLY
EXPORT    fact
; int fact (int n)
; for (i = 1; i <= n; i++)
; fact = fact * i
fact      LDR r3, #1
          LDR r0, #1
          LDR r1, #5      ; say to find factorial of 5
Loop      muls r2, r0, r1
          sub  r1, r1, r3
          cmp  r1, #0 x 0
          BNE LOOP
END
```

This program loads r3 with 1, r0 with 1, r1 with value to find factorial muls - multiply r0 * r1 → r2 Subtract r1 by r3 r1 – 1 → r1, cmp r1 with 0, BNE branch to loop if r1 ! = 0

b. Write short notes on memory access and branch instructions of ARM controller. (08 Marks)

Ans. **(i) Branch instructions (B, BL, BX, BLX):**

These instruction neither processes data nor moves data around. It changes the flow of execution or call a subroutine.

There are two categories (Four Types) conditional (B) and unconditional branch (BL, BX, BLX). The program counter PC points new address B-Branch, BL - Branch with Link, BX - Branch Exchange, BLX - Branch Exchange with Link.

Syntax:

```
B Label                 ; pc ← address of label
BL {<cond>} Label       ; PC ← Address of label
                        ; lr ← Address of next instruction
                          after the BL.
BX {<cond>} Rm          ; Branch and exchange
```

Here PC = Rm & 0xfffffffe, T = Rm & 1.

BLX - Branch exchange with link

PC = Label, T = 1 (T= Thumb bit in CPSR)

PC = Rm & 0xfffffffe, T = Rm & 1

lr = address of next instruction after the BLX.

Syntax:

```
BLX <address>
```


Sunstar Exam Scanner
Scanned by CamScanner

BLX <cond> Rm
BLX Rm

Branch instruction can put conditions by using Mnemonic used for conditional execution instruction like BEQ, BNE, BLD, BLE as given in clause 9.3.5.

Example 1:

```
B Label   ;Branch unconditonally toa label
or
B Forward        ;Label forward label after B instruction.
B Backward       ;Label backward is before B instruction.
BGT Loop         ;Conditionally continue a loop if greater than
```

Example 2:

```
BL Subroutines   ;Call Subroutine return with MOV Pc, lr.
BLVS Overflow          ;Call Subroutine on an overflow.
```

Example 3:

```
BX Lr            ;return from ARM to Thumb Subroutine.
BX r0            ;Branch to ARM or Thumb function pointer r0.
```

Example 4:

```
BLX Thumb_code         ;Call a Thumb subroutine from ARM state.
BL r0                  ;Call the subroutine pointed to by r0.
                 ; if r0 is even, ARM Code.
                 ; if r0 is odd, Thumb Code.
```

(ii) Memory access instructions

These instructions are used to move a single data item in and out of a register it supports signed and unsigned (32 bit) half word (H - 16 bit) and bytes (B - 8 bits)

```
Syntax :  LDR    Rd, Address      ; Rd ← mem32 [Address]
          (d)    (s)              ; load a word/byte to Rd
          STR    Rd, Address      ; Rd → mem32 [Address]
          (s)    (d)              ; store word/byte from
                                  ; Rd to memory
          LDRB   Rd, Address      ; Rd ← mem8 [Address]
          LDRH   Rd, Address      ; Rd ← mem16 [Address]
          STRH   Rd, Address
          STRB   Rd, Address
          LDRSB  Rd, Address      ; Rd ← sign extended
                                    mem8 [Address]
          LDRSH  Rd, Address      ; Rd ← sign extended
                                    mem16 [Address]
          LDR    r0, [r1]         ; r0 ← [r1 + 0]
          STR    r0, [r1]         ; r0 → [r1 + 0]
```


Scanned by CamScanner