Microprocessors and Interfacing

Unit I

The 8086 Microprocessor

Introduction to 8086 - Microprocessor Architecture - Addressing modes - Instruction set and assembler directives - Assembly language programming - Stacks - procedures - macros - Interrupts and interrupt service routines - byte and string manipulation

* Introduction to 8086

→ The Intel 8086 is a 16-bit microprocessor that is intended to be used in the CPU in a microprocessor.

→ The ALU, internal registers & most of its instructions are designed to work with 16-bit binary words.

→ The 8086 has a 16-bit data bus, so it can read data / write data to memory and ports either 16 or 8 bits at a time

* Architecture

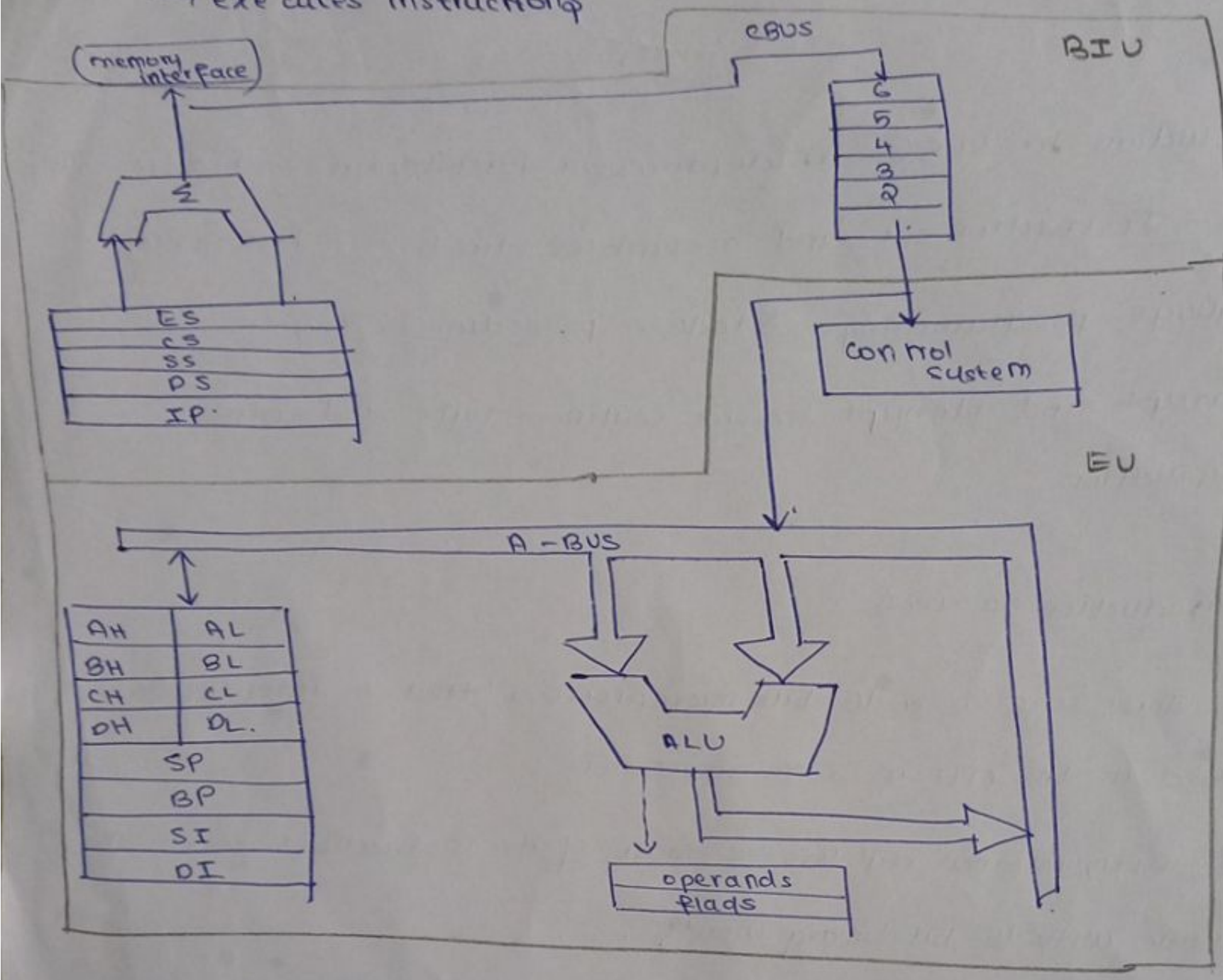
→ The 8086 CPU is divided into 2 independent functional parts, the bus interface unit or BIU, and the execution unit or the EU.

→ BIU : sends out addresses
fetches instructions from memory
reads, data from ports & ports
writes

EIU → tells BIU where to fetch instructions from

→ decodes instructions

→ executes instructions



Bus Interface Unit

Segment Registers

→ has 5 16-bit registers, namely ES, CS, SS, DS and IP

→ obtain starting address from ES, CS, SS, DS

offset is given by IP

→ The memory is of 1MB, the registers are of 16 bits (20 bits)

→ To calculate the physical address, left-shift starting address by 1, and add the offset

for eg.

CS = 1A1A offset = 1B1B

left shift ⇒ 1A1A0

⇒ 1A1A0
  1B1B

1BC BB

Instruction pointer – holds the 16 bits address of the next code byte in the segment (offset) value. This must be added to the segment base address in CS to produce the required 20-bit physical address

Instruction Queue

→ To speed up program execution, the BIU fetches 6 instruction bytes ahead of time from the memory.

→ These are held in a group of registers called queue.

Execution Unit (EU)

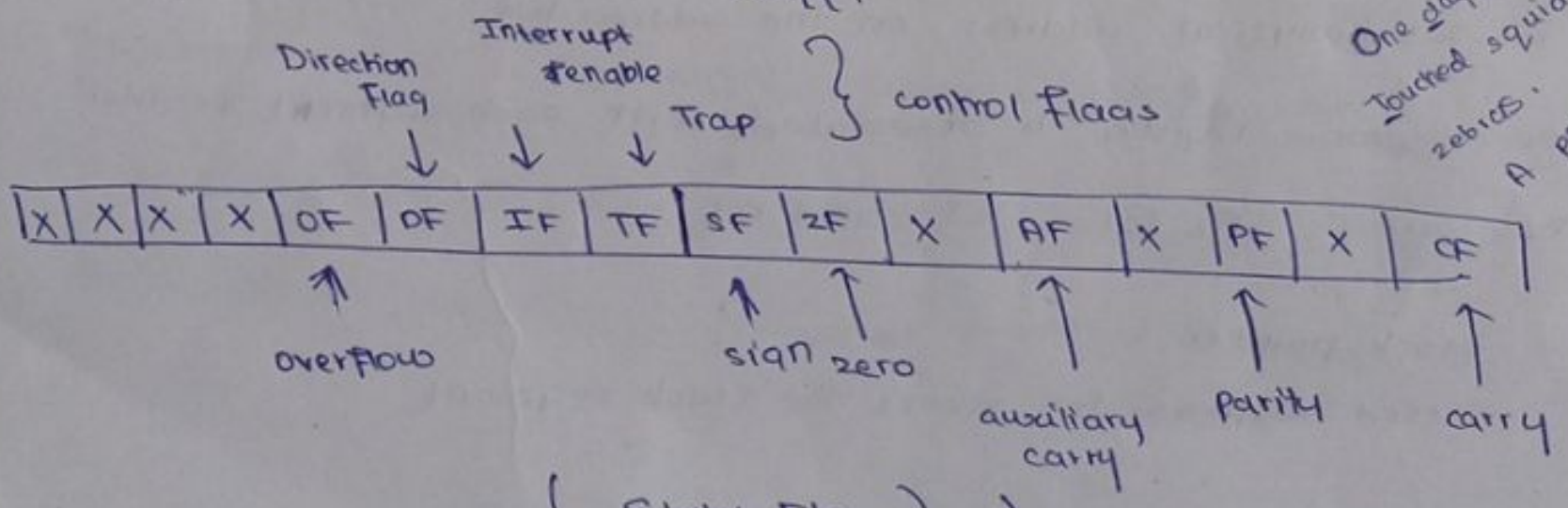
ALU

→ 16 bits, responsible for addition, subtraction, AND, OR, XOR, incrementation, decrementation, complement and shifting.

Flag Register

→ a flip which indicates some condition produced by the execution of an instruction.

→ Flag register has 9 active flags.



General Purpose Registers

(i) AX

→ accumulator register

→ used in arithmetic, logic and data transfer instructions as it generates the shortest machine language code

→ must be used in multiplication and division operations

→ Must be used in I/O operations

(ii) BX

→ base register

→ serves as an address register

(iii) CX

→ count register

→ used as a loop counter

→ used in shift and rotate operations

(iv) DX

→ data register

→ used in multiplication & division

→ also used in I/O operations

Pointer and Index Registers

→ All of the segment registers are 16-bits. But it is necessary to use a 20-bit physical address on the address bus.

→ One or more register is associated with each segment register. These are SP, BP, SI and DI.

SP = stack pointer

- used with SS to access the stack segment

BP - base pointer

→ used to access data on the stack

→ can be used to access data in other segments

SI - source index register

→ required for some string operations

→ points to memory locations in the data segment addressed by the DS register

→

DI : destination index register

→ required for some string operations

→ points to memory locations in data segment addressed by the ES register

Note = SI and DI may be used to access data stored in arrays

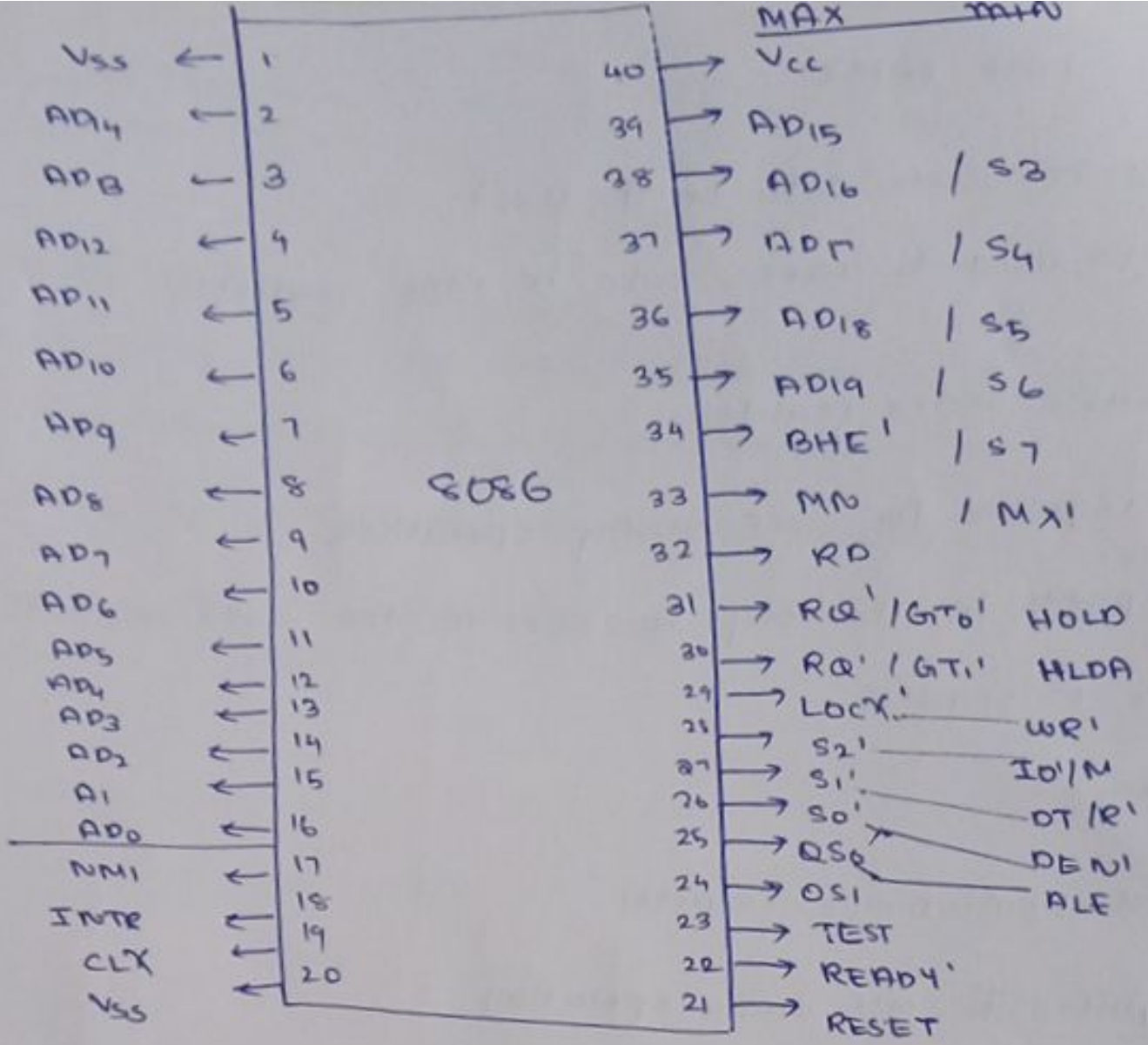
* Intel 8086 - Pin Diagram

→ The 8086 microprocessor contains 40 pins

→ It operates in 2 modes - minimum & maximum mode.

→ The minimum mode is a single processor configuration while the maximum mode is a multiple processor configuration.

→ Due to this, 8 pins - 24 to 32 are assigned different configurations separately according to the 2 modes



V_{CC} = Pin 40 - External power supply of +5V

V_{SS} = Pin 1 & 20 - ground - direct extra current of the microprocessor to the ground

$AD_0 - AD_{15}$ - Pin 2-16 and 39 - multiplexed address and data bus.

(The 8086 microprocessor has a 20-bit address bus and a 16-bit data bus. So the 16 lines of the address and data bus are multiplexed together, to reduce the number of lines inside the IC.)

→ At a particular instant, either data or address will be transmitted by the bus.

A16/S3 , A17/S4, A18/S5 and A19/S6 – Pins 35 - 38

→ Out of 20 address bits, 4 are present in the multiplexed form with the status signals.

→ In case of memory operations, these pins act as an address bus and contain the memory address of any particular data or instruction

→ However, during I/O operations – those pins are low and shows the status of the processor

→ The signal at S3 and S4 show the segment that is currently accessed by the microprocessor.

S3	S4	STATUS
0	0	ES
0	1	SS
1	0	CS or idle
1	1	DS

S5 → when enabled, serves as an interrupt flag

S6 → shows the status of the bus master - i.e if 8086 is the bus master or if any other ~~pr~~ device is acting as the bus master.

0 ⇒ 8086

1 ⇒ other processor.

BHE' / S7 – Pin 34 = Bus High Enable

→ combination of BHE & S7 status informs about the existence of data on the bus.

BHE	S7	STATUS
0	0	All 16 bits will be accessed
0	1	upper byte accessed
1	0	lower byte accessed
1	1	none or idle status

MN/MX' – Pin 33

→ shows whether the processor is operating in the min or max mode

min → 0

max ⇒ 1

RD' – Pin 32 → An active low signal that shows that the microprocessor is performing read operation with either the memory or I/O devices.

CLK – Pin 19 → provides timing to the internal operations being executed inside the processor.

NMI – Pin 17 – Non-maskable interrupt, uncontrollable interrupt generated inside the processor. When NMI occurs, there is an ISR generated by the interrupt vector table.

TEST – Pin 23 – shows the wait instruction. Whenever a low signal at this pin occurs, then the processing inside the processor continues, else remains idle.

INTR - Pin 18 - INTR = interrupt request. After each clock cycle, the processor samples the INTR, if high, then the processor controls that interrupt internally.

READY - Pin 22 - used by peripherals & memory devices to show the readiness for the next operation.

RESET - Pin 21 - resets & terminates recent task.

* Pins in Minimum Mode.

INTA' - Pin 24 = interrupt acknowledgement

ALE - Pin 25 = address latch enable

DEN - Pin 26 = data enable - separates data from the multiplexed address and data bus.

DT/R' - Pin 27 - whether data is getting transmitted (↓ high) or received (↓ low)

M/IO' - Pin 28 - whether a processor is performing an operation with memory (↓ high) or I/O devices (↓ low)

HOLD - Pin 31 - when an external device enables this pin

HLDA - Pin 30 - response for hold request

* Pins in Maximum Mode

S0', S1', S2' - Pins 26-28 - handling of interrupts

S_0	S_1	S_2	Status
0	0	0	INTA
0	0	1	read I/O
0	1	0	write I/O
0	1	1	halt
1	0	0	code access
1	0	1	read memory
1	1	0	write memory
1	1	1	none - passive

$\overline{QS_0}$ and $\overline{QS_1}$ - Pin 24 & 25 → indicate the status of the 6 byte pre-fetch queue

QS_0	QS_1	Status
0	0	no operation
0	1	first byte from queue
1	0	empty queue
1	1	subsequent byte from queue

$\overline{LOCK}$ - Pin 29 - when a single processor is accessing the buses and peripherals, then it locks the resources used by it - no other entity can access it

$\overline{RQ}/\overline{GT_0}$ and $\overline{RQ}/\overline{GT_1}$ - Pin 30 & 31 - request and grant permission for accessing the buses, memory & peripherals

* Addressing Modes

Group I - Addressing modes for register and immediate data.

1. Register Addressing

The instruction will specify the name of the register which holds the data to be operated by the instruction

Example: MOV CL, DH

The content of 8-bit register DH is moved to another 8-bit register CL

2. Immediate Addressing

An 8-bit or 16-bit data is specified as part of the instruction

Example: $(DL) \leftarrow 05_H$

or $(AX) \leftarrow 0A9F_H$

Group II - Addressing modes for memory data.

3. Direct Addressing : The effective address of the memory location at which the data operand is stored is given in the instruction

Example: MOV BX, [1354H]

→ The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction copies the contents of the memory location into the BX register.

→ The addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction

Physical Address Calculation

Seg: offset = 89AB: F012

⇒ 89AB0
+ F012
98AC2

4. Register Indirect Addressing

→ The name of the register which holds the effective address (EA) is specified in the instruction.

→ Registers used in EA to hold EA are BX, BP, DI and SI.

→ The content of the DS register is used for base address calculation.

Example mov cx, [BX]

operations EA = (BX)

BA = DS × 16_{10}

MA = BA + EA

CX ← (MA)

or CL ← (MA)

and CH ← (MA+1)

5. Based Addressing

→ In based addressing, BX or BP is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

→ In case of 8-bit displacement, it is sign-extended to 16-bit before adding to the base value.

→ When BX holds the base value, the 20-bit address is calculated from BX and DS.

→ When BP holds the base value, BP and SS are used.

Example: mov AX, [BX + 05H]

Operations: Sign extend 05_H to 0005_H

EA ← (BX) + 0005_H

BA ← (DS) × 16_{10}

MA ← BA + EA

AX ← (MA) or (AL) ← (MA) or
(AL) ← (MA+1)

6. Indexed Addressing

→ SI or DI registers are used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction

→ Displacement is added to the index value in SI or DI register to obtain the EA.

→ In case of 8-bit displacement, it is sign-extended to 16 bit before adding to the base value

mov CX, [SI + 0A3H]

Operations.

sign extend $A2_H$ to FFA_2H

EA = (SI) + $FFA2_H$

BA = (DS) × 16_{10}

MA = BA + EA

(CX) ← (MA) or
(CL) ← (MA)
(CH) ← (MA+1)

7. Based Index Addressing

→ The effective address is computed from the sum of a base register (BX or BP), and an index register (SI or DI) and a displacement

mov DX, [BX + SI + 0AH]

Operations:

$0A_H \rightarrow 000AH$ (sign extended)

EA = (BX) + (SI) + 000AH

BA = (DS) × 16_{10}

MA = BA + EA

(DX) ← MA or

(DL) ← MA

(DH) ← (MA+1)

8. String Addressing

→ Employed in string operations to operate on string data.

→ The effective address (EA) of the source data is stored in the SI register and the EA of the destination is stored in the DI register.

→ Segment register for calculating the base address of source data is DS and that of the destination data is ES

Example movs BYTE

source addr: EA = (SI)

BA = (DS) × 1610

MA = BA + EA

destination addr:

$EA_E = (DI)$

$BA_E = (ES) \times 16_{10}$

$MA_E = BA_E + EA_E$

$(MA_E) = (MA)$

If DF = 1 (SI) ← (SI) − 1 and (DI) ← (DI) − 1

If DF = 0 (SI) ← (SI) + 1 and (DI) ← (DI) + 1

Group III - Addressing modes for I/O ports

9. Direct Port Addressing

→ An 8-bit port address is directly specified in the instruction

Example: IN AL, [09H]

ops $PORT_{addr} = 09_H$

(AL) ← (PORT)

Content of port with address 09H is moved to the AL register

10. Indirect Port Addressing

→ The instruction specifies the name of the register which holds the port address

→ 16-bit port address stored in DX register

Example: OUT [DX], AX

ops: $PORT_{addr} = (DX)$

(PORT) ← (AX)

Content of AX is moved to port whose address is specified by the DX register.

Group IV : Relative Addressing mode

11. Relative Addressing mode

→ In this addressing mode, the effective address of a program instruction is specified relative to the IP by an 8-bit signed displacement.

Example : JZ 0AH

ops 0A$_H$ ⇒ 000A$_H$ (sign-extend)

If ZF=1 then

EA = (IP) + 000A$_H$

BA = (CS) x 16$_{10}$

MA = BA + EA

ZF = 1 ⇒ jump to new address

ZF = 0 ⇒ next inst is executed

12. Implied Addressing

→ Instructions using this mode have no operands.

→ The instruction itself will specify the data to be operated by the instruction

Example : CLC

This clears the zero flag to zero

* Assembler Directives

→ Instructions to the assembler regarding the program being executed.

→ controls generation of machine code & organization, but no machine code is generated for the assembler directives

→ called pseudo instructions

→ used to:
- (i) specify start & end of program
- (ii) attach value to variables
- (iii) allocate storage to I/O data
- (iv) define start & end of segments, procedures & macros

(1) DB - define byte - reserves 8 bits

```
<varname> DB <value>
myvar DB 42H
```

(2) DW - define word - reserves 16 bits - reserves to consec memory locations to each word

```
<varname> DW <value>
```

(3) segment, ends - defines beginning & end of segment

```
mysegment SEGMENT
    :
    :
mysegment ENDS
```

(4) ASSUME - tells what segment is what

eg. ASSUME DS : myseqment

→ shows that myseqment is supposed to be DS

(5) ORG - origin

whatever is written below it will start from the offset address

eg. ORG 2000H (offset by 2000H)

(6) END - terminate a program. - statements after END are ignored.

(7) EVEN - informs assembler to store program starting from an even address

(8) EQU - associate a value to a var

eg. A equ 25H

creates a constant

equivalent to #define in C

when equ & when db/dw ?

If something changes, make it a variable, assembler gets the address.

equ will fetch value (saves time)

(9) proc - stands for procedure - used to define a function

eg. proc myfunction

:

endp myfunction

call proc with

'call <procname>'

(10) macro - used to define a macro

eg. macro mymacro

:

endm mymacro

call macro simply by name

(11) NEAR / FAR -

↓ ↳ inter segment calls

intra segment calls

eg. ADD_NUMS PROC NEAR

:

ADD_NUMS ENDP

(12) SHORT - reserves one memory location 8 bit signed displacement in jump instructions

eg. JMP SHORT myalp

(reserves one memory location for 8-bit displacement named my disp)

* Differences between a procedure & macro

PROC	MACRO
1. Program goes to proc to execute and then returns	1. macro code pasted into program
2. advantage of proc - overall code length decreases	2. code length increases
3. requires a push & pop (CS & IP) - causes overhead	3. no push & pop => no overhead
4. requires a stack	4. doesn't need stack
5. proc uses branching, pipelining fails, slows down	5. pipelining doesn't fail
6. use a macro if there's a loop	6. use proc if code is being used too many times
7. invoked w/ call 'procname'	7. just use the name of macro

* 8086 Memory Banks

CAT-1Q → 8086 has a 16-bit data bus, so it should be able to access 16-bit data in one cycle

→ To do so, it needs to read from 2 memory locations, as one memory location carries only 1 byte (8 bits) of data

→ However, if both of these memory locations are in the same memory chip, then they cannot be accessed simultaneously.

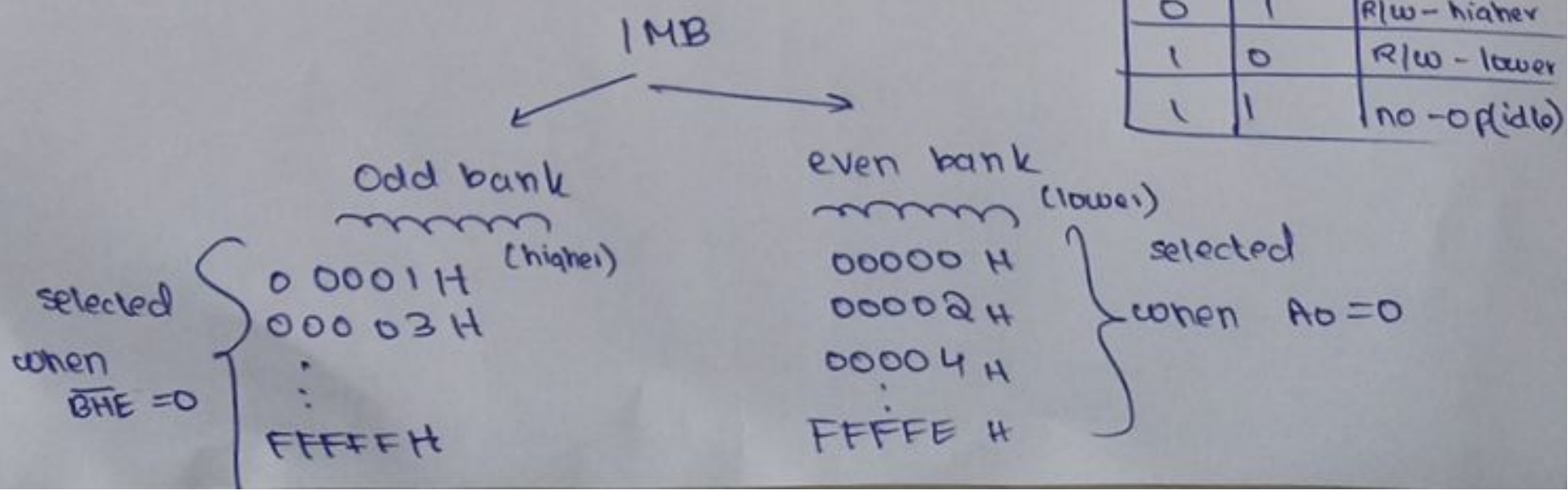
→ Thus 8086's memory is divided into 2 banks - where each bank provides 8 bits.

→ Each chip contains alternating locations

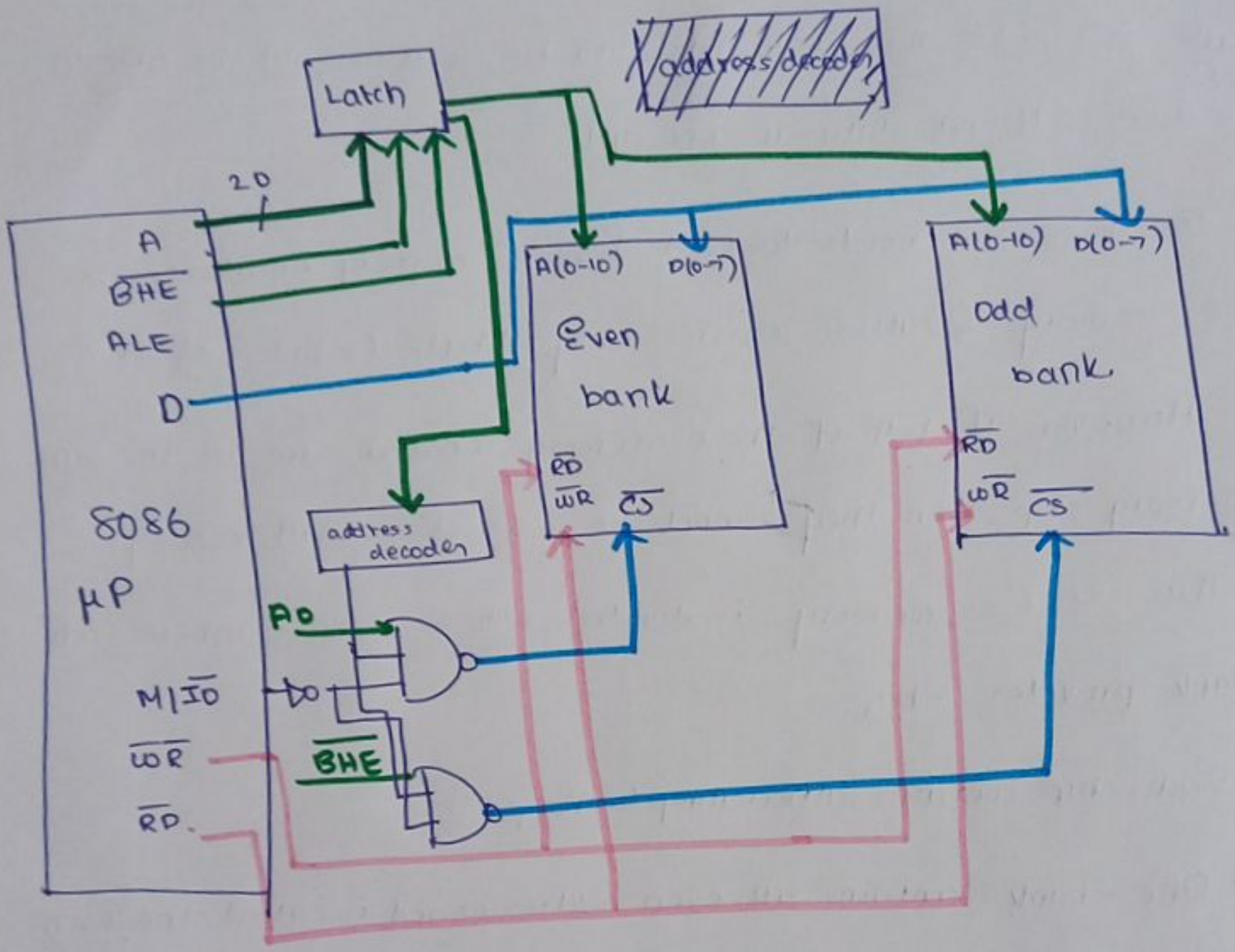
→ One bank contains all even addresses and is called the even bank, and the other is called the odd bank, with all the odd addresses.

→ For a 16-bit operation, the even bank provides the lower byte and the odd bank provides the higher byte.

$\overline{BHE}$	A	op
0	0	R/W 16 bit from both banks
0	1	R/W - higher
1	0	R/W - lower
1	1	no-op (idle)



Memory bank diagram

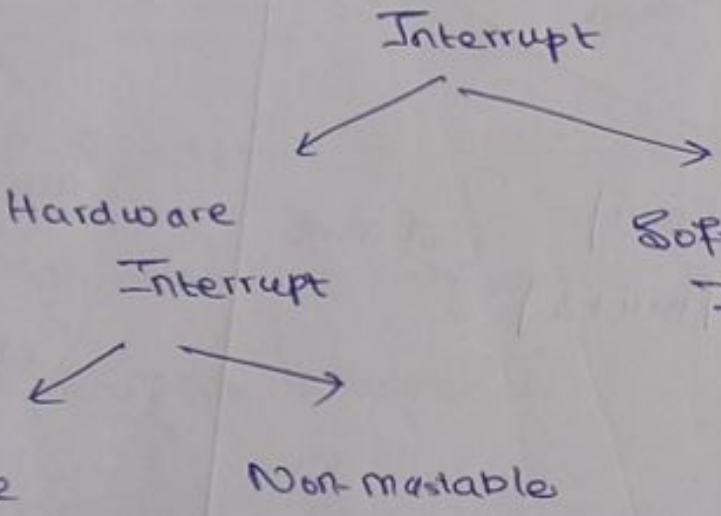


* Interrupts and Interrupt Service Routines

Interrupt - a method of creating a temporary halt during program execution & allows the peripheral devices to access the microprocessor.

ISR - a short program that instructs the microprocessor on how to handle the interrupt.

Classification



Hardware Interrupts.

→ occur as external pins of the μP

→ has 2 pins - NMI & INTR (INTA = interrupt acknowledge)
↓ non maskable
↓ maskable has lower priority

A. **NMI** - ~~on μP~~ executes INT
Follow the generic procedure taking n=2

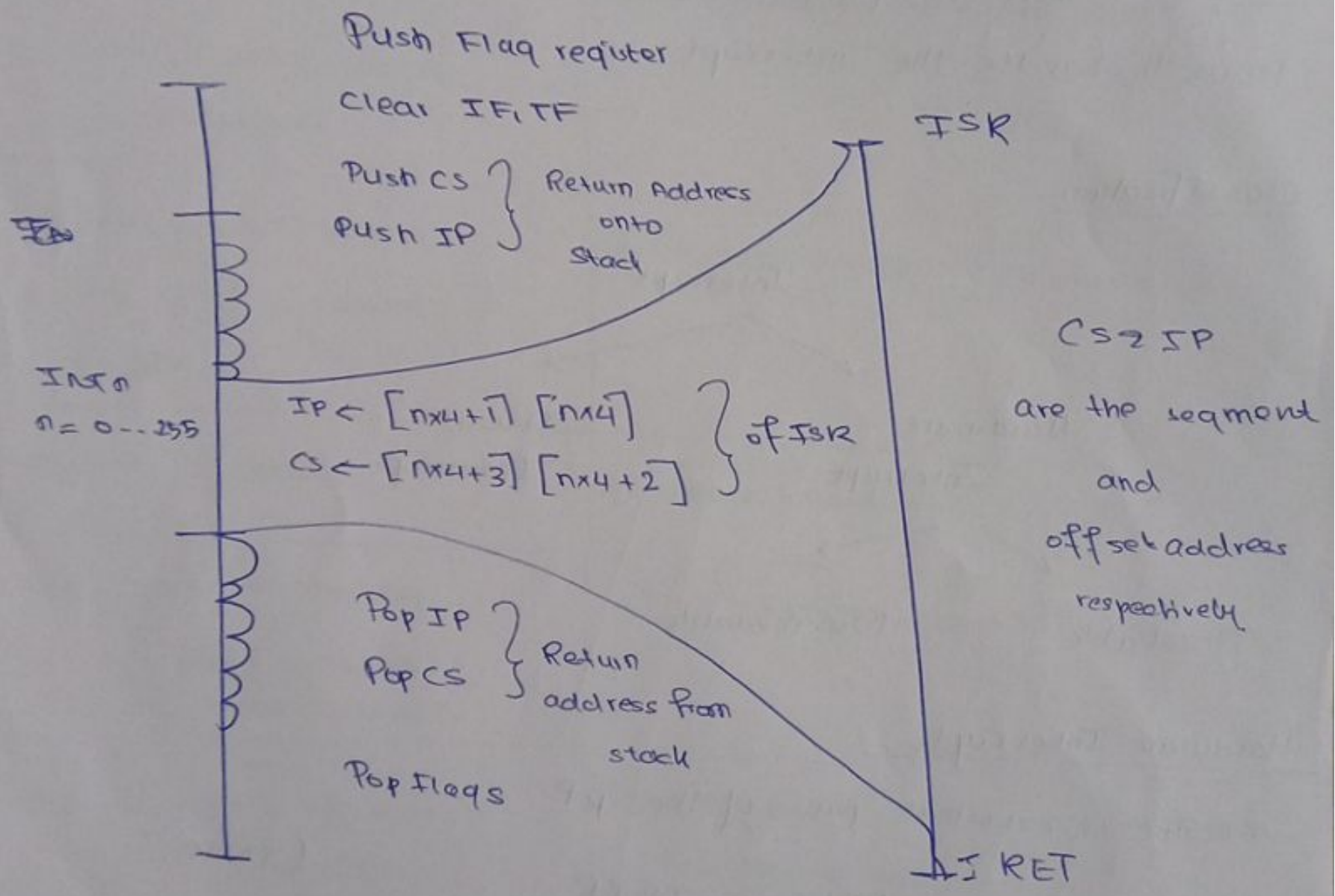
B. **INTR** - will interrupt only if interrupts are enabled using the set interrupt flag instruction
- activated by an IO port

→ execute 2 $\overline{INTR}$ pulses

(i) 1st pulse - calculates (prepares to send) vector number

(ii) 2nd pulse - sends vector number N to the μP

* Generic Procedure for all interrupts



→ An interrupt (INT n) occurs. Program halts executes ISR & comes back

→ n can be from 0 - 255

→ The branching is intersegment, meaning both CS & IP have to be changed.

→ Push the flag register (8 values)

→ Clear IF & TF. This disables INTR and ss

→ Push current CS & IP into stack

→ Obtain value of CS & IP of the ISR from the IVT

$$IP \leftarrow [n\times4+1][n\times4]$$
$$CS \leftarrow [n\times4+3][n\times4+2]$$

→ Pop IP, CS & Flags when IRET is called.

→ Reset flags to 0

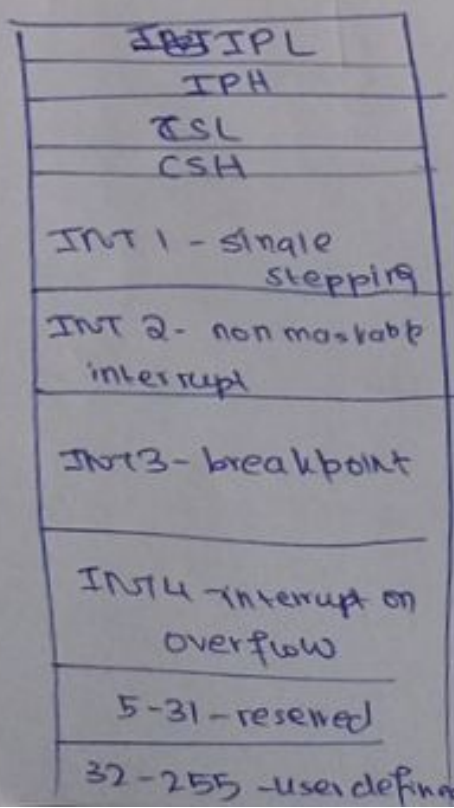
* Interrupt Vector Table (IVT)

→ has ISR address for 256 interrupts

→ Each ISR address is stored as CS & IP - each ISR addr requires 4 locations to be stored.

→ Total IVT size = 256 × 4 = 1KB

→ IVT starting address is 00000H



INT 0 - divide error

→ happens when overflow flag is set & INTO instruction is executed

* Software Interrupts

→ inserted at the desired program position in the program

→ can test the working of various interrupt handlers

* Priority

INT 0, INT n, INTO
NMI
INTR
Single stepping

} can interrupt an ISR

} cannot interrupt an ISR since IF & TF are set to 0 before entering ISR.

* Stacks, Procedures & Macros

A. Stack

→ A section of memory set aside for storing return addresses

→ can be used to store the contents of registers for the calling program

→ stack segment holds the segment base address of the stack segment

→ stack pointer holds offset of last word written on the stack

B. Procedure

→ To avoid writing the sequence of instructions in the program each time, write a subprogram that is called a procedure

→ direct / indirect within segment call = NEAR

→ inter segment call = FAR

→ Passing parameters through & from procedure can be done by:

(i) registers

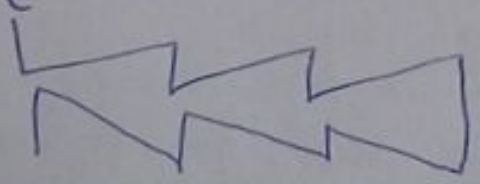
(ii) dedicated memory locations accessed by name

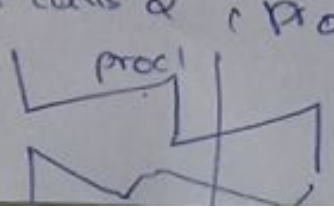
(iii) pointers passed in registers

(iv) w/ the stack

→ Reentrant - can be called by multiple processes simultaneously

→ Recursive





* Instruction Set

(1) PUSH

(16-bit) } has to be a 16 register

eg: PUSH BX

⇒ push value of Reg. BX into stack

SP-2	BL
SP-1	BH
SP	XX

SS: [SP-1] ← BH

SS: [SP-2] ← BL.

SP ← SP-2

(2) POP

eg. POP DX

SP	34
SP+1	12
SP+2	XX

BL ← SS: [SP]

BH ← SS: [SP+1]

SP ← SP+2

(3) IN → accepting into an I/O device

IN dest, source

(Reg) direct address or DX

(4) OUT → sending out of an I/O device

OUT dest, source

DX AX
AL
AH

Examples

(i) send data 25H to port 80H (8 bit)

mov AL, 25H

OUT 80H, AL

can directly be written

(ii) send data 25H to port 8000H (16 bit)

mov AL, 25H

mov DX, 8000H

OUT DX, AL

must use DX

(5) mov :

mov req2, req1 ⇒ req2 ← req1

(6) XCHG

XCHG req2, req1 ⇒ req2 ⇔ req1

Arithmetic Instructions

(1) ADD

ADD BL, CL ⇒ BL ← BL + CL

However,

FF + FF would result in

1FE

BL ← FE

CF ← 1

(2) ADC → add with carry

AD BL, CL ⇒ BL ← BL + CL + CF

carry before the operation, i.e carry of the previous operation.

→ used when there are large numbers

eg.

12 FFH	
00 01H	
13 00	

ADC is used when 2 separate nums are added

For eg. to do 32-bit addition do 2 separate BPs

lower bits with ADD and the higher order with ADC:

(3) SUB BL, CL ⇒ SUB BL - CL

(4) SBB : subtract with borrow

SBB BL, CL ⇒ SBB BL, CL ← BL - CL - CF

(5) INC BL ⇒ BL ← BL+1

(6) DEC BL ⇒ BL ← BL−1

(7) MUL operand

↳ the other operand is in an accumulator

↓

8-bit ← AL

16-bit ← AX

32-bit – DX, AX
(H) (L)

8bit
eg. MUL BL

AX ← ⇒ AL × BL

MUL BX

DX, AX ← AX × BX

(8) DIV divisor

DIV BL

⇒ AL / BL

eg. 75/12

MOV AL, 75

MOV BL, 12

IMUL 2

IDIV for signed numbers

16bit / 8bit ← (AH)(AL) ← AX ÷ BL
↓ ↓
R Q

32bit by 16bit ← (DX)(AX) ← (DX)(AX) ÷ BX
↓ ↓
R Q

(9) CWD → convert word to double word

extends 16 bit to 32 bit

This is done when you need to divide 16bits by 16 bit

Take msb & extend

eg. Divide

(10) CMP BL, CL.

BL - CL. - does not store result, just check the flags

	CF	ZF
BL > CL	0	0
BL = CL	0	1
BL < CL	1	0

(11) DAA - decimal adjust after addition

used only after addition of decimal numbers.

AL

If HN > 9
or CF = 1

add 60

If LN > 9
or
AC = 1

add 06

Example: 25H
25H
4A.
06
50

or 26H
26H
4C
06
52.

or 28H
28H
50
06
56

or

1101 0001
0110
1101 0111
0110 0000
1) 0011 0111
37

```
  9 9
  9 9  +
-------
  132
-------
CF    AL.

   +06
-------
  138
=======
```

(12) DAS.

* Logical Instructions

(i) AND

(ii) OR

(iii) XOR

(iv) NOT

(v) TEST BL, CL.

BL ∧ CL same op as AND
does not store
just affects flags.

(vi) ROL → rotate left

ROL reg, count

↳ if count = 1 → ROL, BL 1

but ROL BL, 4 X

then it has to be written as

mov CL, 04

ROL, BL, CL.

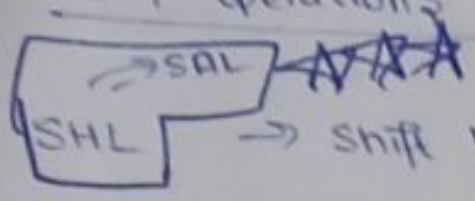
note that the MSB goes into the CF and to the LSB

(vii) ROR → rotate right

(viii) RCL → rotate left with carry

(ix) RCR → rotate right with carry

Shift operations



(i) SHL → shift left , SHL . BL, 1

carry does not move to LSB,

rather a 0 enters at the end.

(ii) SHR → shift right SHR BL, 1

(iii) SAR ⇒ shift right, but consider the sign

retain MSB & shift

swapping nibbles

Rotate a number 4 times

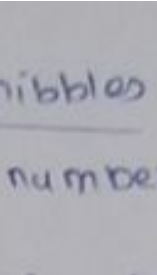
```
MOV CL, 04
ROL BL, CL
```

Processor Contro / Machine Control Instr

directly operate on flags

(i) PUSHF → push flag reg value into stack

(ii) POPF → pop " from "



(iii) LAHF → load AH from flag

AH ← F

take the 8 lower bits of flag reg & put into AH

(iv) SAHF → store AH into flag

put it back into place.

where to use these? if one wants to use a specific value of the flag, after say, 100 instructions, then PUSHF into stack, after 100 instructions POPF.

Instructions for specific flags

(1) For CF

(a) STC = set CF ← 1

(b) CLC = clear CF ← 0

(c) CMC = complement CF ← $\overline{CF}$

(2) For DF

(a) STD = set DF ← 1

(b) CLD = ~~set~~ clear, DF ← 0

(3) For IF

(a) STI = set interrupt flag

IF ← 1

(b) CLI = clear interrupt flag

IF ← 0

(X) Procedure to change the TF

↳ is a control flag but no direct operation

(i) PUSHF

(ii) POP BX

To make TF 1

(iii) OR with 01 H

(OR BH, 01H)

(iv) Push BX

(v) POPF

To make TF 0

AND with FEH

(AND BH, 0FEH)

XLAT

AL ← DS: [BX + AL]

helps look up values

add AL to BX

replaces w/ new value

* Branching Instructions

Jump vs. call

↓ ↳ call subroutine & return to next inst. from call

jump z

remain at pos

JMP — JMP label.

call — procedure same as JR.

* Conditional Jumps

JC → jump if carry CF = 1

JNC → no carry CF = 0

~~JE~~ → jump if zero ZF

JNZ ⇒ jump if not zero

JPE, JPO — jump if parity even/odd. PF

JO / JNO - jump if overflow / no overflow

JS / JNS - jump if sign / no sign SF

** JCXZ — jump if CX register is 0.

* String Instructions

Source string - DS SI

dest string - ES DI

(1) movs - move string

(i) movsB : ES : [DI] ← DS : [SI]

extra segment offset address DI gets data from data segment offset SI.

256.

SI ← SI + 1

DI ← DI + 1

→ To transfer multiple bytes use REP (instruction prefix)

CX holds count

MOV CX , 000A H

REP MOVSB ES:[DI] CX ← CX − 1

→ direction to go by is specified by direction flag

use CLD - clear dir flag or STD

↓ ↓

inc dec.

(ii) movsw

movsw: ES : [DI], ES : [DI + 1] ← DS: [SI] DS : [SI + 1]

(2) LODS → load string

LODSB : AL ← DS : [SI]

SI ← SI ± 1

LODSW : AX ← DS : [SI] , DS : [SI+1]

SI ← SI ± 2

(3) STOS → store string

STOSB : ES : [DI] ← AL

DI ← DI ± 1

STOSW : ES : [DI], ES : [DI + 1] ← AX

(4) CMPS - compare string

source & destination

CMPSB : DS:[SI] with ES:[DI] check zero flag

CMBSW : SI ← SI ± 1

DI ← DI ± 1

(5) SCAS - compare AL no present in accumulator w.r.t every no. in the string

scanning string

compare

SCASB : AL with ES[DI]

DI ← DI + 1

* Repeat Types

REPE | REPZ --- ZF = 1

repeat on equal, repeat on ~~equal~~ zero

REPNE | REPNZ

repeat on not zero, repeat on not zero

Microprocessor Lab

Programming in 8086

SET 1 - Arithmetic operations

(1) 8-bit addition

```
mov  AH, [2000]
mov  BH, [2001]
mov  CH, 00
ADD  AH, BH
JNC  Label
INC  CH
Label: mov [2002], AH
       mov [2003], CH
       HLT
```

(2) 8-bit subtraction

```
mov  AH, [2000]
mov  BH, [2001]
mov  CH, 00
SUB  AH, BH
JNC  Label
INC  CH
NEG  AH
Label: mov [2002], AH
       mov [2003], CH
       HLT
```

(3) 8-bit multiplication

```
mov  AL, [2000]
mov  BL, [2001]
MUL  BL
mov  [2002], AX      → low high
HLT
```

(4) 8-bit division

```
mov  AH, 00
mov  AL, [2000]
mov  BL, [2001]
DIV  BL
mov  [2002], AX
HLT
```

AL → quotient
AH → remainder

(5) 16-bit addition.

```
mov  AX, [2000]
mov  BX, [2002]
mov  CX, 00
ADD  AX, BX
JNC  Label
INC  CH
Label: mov [2004], AX
       mov [2006], CX
       HLT
```

(6) 16-bit Subtraction

```
mov AX,[2000]
mov BX, [2002]
mov CX, 00
SUB AX,BX
JNC Label
INC CX
NEG AX
Label: mov [2004], AX
       mov [2006], CX
HLT
```

(7) 16-bit Multiplication

```
mov AX, [2000]
mov BX, [2002]
MUL BX          (DX AX)
mov [2004], AX
mov [2006], DX
HLT
```

(8) 16-bit division

```
mov AX, [2000]
mov BX, [2002]
mov DX, 0000H
DIV BX      (DX)(AX) → AX ÷ BX
            AX = Q
            DX = R
mov [2004], AX
mov [2006], DX
HLT.
```

(9) Addition of 2-32 bit numbers

```
mov AX, [2000]   → 16 bits num1 (l)
mov BX, [2002]   → 16bit num1(h)
mov CX, [2004]   → 16 bits num2 (h)
mov DX, [2006]   → 16 bits num2 (l)
ADD BX, DX
ADC AX, CX
mov [2004], AX
mov [20+07], BX
HLT.
```

(10) Multiplication of 2 32 bit numbers

(1) Put the First n even numbers into an array.

```
MOV CL, [2000]
MOV AX, [2002]
MOV DS, AX

MOV DI, 0000H

MOV AX, 0000H   ← first even number
Label:
MOV [DI], AX    ← move into array

INC DI
ADD AX, 02H

LOOP Label
HLT
```

when putting in array, don't put brackets

```
MOV AX, 3000H   ← base
MOV DX, AX

MOV DI, 0000H   ← like the i value
DX:DI = 3000:0000
        first location
```

keep incrementing DI

(2) Find the sum of n elements in an array.

making the array

~~MOV CL, [2000].~~ ← get count of elements, 2000 is where the data begins, first element is the count

~~MOV DX, 2000 INC~~

~~MOV DS, AX~~

```
MOV SI, 1100H
MOV DI, 1200H   ⟶ will hold output value
MOV CX, [SI]    ⟶ store count in CL
INC SI          ⟶ move SI to next location
```

```
mov AX, 0000H   → inital set the accumulator to ha.
                                                  0

myloop: mov BX, [SI]
        ADD AX, BX

        INC SI      → move index

        DEC CX      → decrement counter.

        JNZ myloop


mov [DI], AX

HLT
```

(3) Programto find the largest number in an array

```
mov SI, 1200H  ← start index

mov DI, 1400H  ← destination index
```

written neatly again on next page

```
mov CX, [SI]    ← move count value into registel
  INC SI
  mov AX,[SI]   → assume that first value is the highest
DEC CX          ← loop should go to 1 less than count

                ← move to next pos
Loop1: DEC CX
       MOV AX, [SI]
       INC SI
       CMP AX, [SI]

       JNC Loop2

       mov AX, [SI]

Loop2: Loop1.   → compare the next elemoul
       JNZ
       mov [DI], AX
       HLT
```

if AX < [SI] ⇒ there will be a carry
if there is no carry, it means it is largos
i.e you should retain it

05 23 0366 01 05

```
MOV  SI, 1200H
mov  DI, 1400H

MOV  CX, [SI]
INC  SI
mov  AL, [SI]

DEC  CX

checkloop:  INC  SI
            CMP  AL, [SI]
            JNC  loop2
            mov  AL, [SI]
            DEC  CX

loop2 :     DEC  CX
            JNZ  checkloop
```

* automatically goes to loop 2 in sequence - if the jnc is valid. it just skips line *

(4) Program to find the second largest number in an array

```
mov   SI, 1200H
mov   DI, 1200H

mov   CX, [SI]
INC  SI
INC  SI

  mov  AL, [SI]
   DEC  CX

checkloop:  INC  CX
            CMP  AL, [SI]
            INC  loop2
```

```
        mov DL, AL
        mov AL, [SI]

loop2:  dec CX
        JNZ checkloop
        mov [DI], DL
        HLT
```

(5) Find the number of occurrences of a specific number in an array

```
mov   SI, 1200H
mov   DI, 1400H
mov   BL, 00H
mov   CX, [SI]
INC   SI
INC   SI
mov   DL, [SI]

checkloop:  INC SI
mov   AL, [SI]
CMP   AL, BL
JNE   LOOP1

LOOP1 : DEC CX
JNZ   CHECKLOOP

mov   [DI], BL
HLT
```

(6) Find the number of even elements in ~~the~~ an array

```
mov   SI, 1200 H
mov   DI, 1400H
mov   CX, [SI]
INC   SI
INC   SI
mov   DX, 0000H
Evenloop:  mov  AX, [SI]
           SHR  AX, 1
           JC   LOOP1
           INC  DX
LOOP1:  INC SI
        INC SI
        DEC CX
        JNZ evenloop.
```

06 0A 22 0B 22 55 06

```
mov [DI], AX   ← result in DI
HLT
```

(7) Generate the first n terms of the Fibonacci Series

```
mov   SI, 1200H
mov   CX, [SI]   → n numbers
mov   AX, 1202 H
mov   DS , AX        } base + offset
mov   DI, 0000H
mov   AX, 0000H  → first number
mov   BX, 0001 H
DEC   CX
fibloop:  ADD AX, BX
          mov DX, AX
          mov [DI], DX
```

```
mov AX, BX      => a = b
mov BX, DX      => b = c
INC DS
DEC CX
JNZ fibloop
HLT
```

(3) Sorting an array in ascending order

```
mov DI, 1300H     → holds the count
mov CH, [DI]      → count of outer loop (i)
outer: mov CL, [DI]  → initialize inner loop (j)
       mov SI, 1200  → go to beginning of array
inner: mov AL, [SI]
       CMP AL, [SI+1]
       JC noswap
       XCHG [SI+1], AL
       XCHG [SI], AL
noswap: INC SI
        DEC CL    ← decrement inner loop
        JNZ inner ← stay in inner loop until done
        DEC CH    ← decrement outer loop
        JNZ outer ← jump to outer loop
        HLT
```

for descending, use JNC

1) Matrix Addition.

```
mov  AL, [2000]
mov  BL, [2001]
mov  CL, [2002]
mov  DL, [2003]
```
} matrix dimensions

```
CMP  AL, BL
JNZ  Label 2
CMP  CL, DL
JNZ  label 2
MUL  DL
```
} check dimension

← find total no. of iterations

```
mov  CX, AX
```
← store count

```
mov  SI, 3000
```
← matrix1

```
mov  DI, 3500
```
← matrix2

```
mov  BX, 3600
```
← result matrix

```
Label 1:  mov  AL, [SI]
```
← move mat1 value into AL

```
          ADD, AL, [DI]
```
← add (use sub otherwise)

```
          mov [BX], AL
```
← store result

```
          INC SI
          INC BX
          INC DI
          loop label 1
```
} inc. index of matrix1, matrix2 and dest matrix

```
Label 2:  HLT
```

```
MOV AX, BX      => a = b
MOV BX, DX      => b = c
INC DI
DEC CX
JNZ fibloop
HLT
```

(8) Sorting an array in ascending order

```
MOV DI, 1300H     → holds the count
MOV CH, [DI]      → count of outer loop (i)
outer: MOV CL, [DI]   → initialize inner loop (j)
       MOV SI, 1200   → go to beginning of array
inner: MOV AL, [SI]
       CMP AL, [SI+1]
       JC noswap
       XCHG [SI+1], AL
       XCHG [SI], AL
noswap: INC SI
       DEC CL     ← decrement inner loop
       JNZ inner  ← stay in inner loop until done
       DEC CH     ← decrement outer loop
       JNZ outer  ← jump to outer loop
       HLT
```

for descending, use JNC

1) Matrix Addition.

```
mov  AL, [2000]
mov  BL, [2001]
mov  CL, [2002]
mov  DL, [2003]
```
} matrix dimensions

```
CMP  AL, BL
JNZ  Label 2
CMP  CL, DL
JNZ  label 2
MUL  DL
```
} check dimensions

← find total no. of iterations

```
mov  CX, AX
```
← store count

```
mov  SI, 3000
```
← matrix 1

```
mov  DI, 3500
```
← matrix 2

```
mov  BX, 3600
```
← result matrix

```
Label 1:  mov  AL, [SI]
          ADD, AL, [DI]
          mov  [BX], AL
          INC  SI
          INC  BX
          INC  DI
          loop Label 1
```
← move mat1 value into AL

← add (use sub otherwise)

← store result

} inc. index of matrix1, matrix2 and dest matrix

```
Label 2:  HLT
```

Set 3- Conversion Questions

(1) Convert HEX to BCD

```
MOV  BL, [1300]
MOV  AL, 00
L1:  ADD  AL, 01
     DAA
     DEC  BL
     JNZ  L1
     MOV  [1301], AL
     HLT
```

(2) Convert BCD to Hex

```
MOV  AL, [1200]
MOV  AH, AL    => store a copy
AND  AH, 0F    => lower byte
MOV  BL, AH
AND  AL, 0F0   => higher byte
MOV  CL, 04
ROR  AL, CL
MOV  BH, 0A
MUL  BH
ADD  AL, BL
MOV  [1201], AL
HLT
```

Steps to follow:

1. store a copy
2. AND with 0F ⇒ lower bits
3. store copy
3. AND with 0F0 ⇒ higher bits with 0 at the end
4. store no. of rotations
5. ROR that many rotations
6. move 0A into a register.
7. multiply with higher byte
8. add to lower byte.

(3) Unpack a BCD number ⇒ If a BCD number is 45 it means' to make it 04 05

```
MOV  AL, [1200]   ← packed BCD
MOV  AH, AL       ← make a copy
AND  AL, 0FH      ← lower bits
MOV  BL, AL       ← store in lower half of BX register
AND  AL, 0F0      ← higher order bits
MOV  CL, 04
ROR  AL, CL       ← rotate to make LSB
MOV  BH, AL       ← move to higher bits of BX register
MOV  [1202], BX
```

(4) Pack a BCD number ⇒ make 0405 to 45

```
MOV  BX, [1200]
MOV  CL, 04
ROR  BH, CL       → rotate the higher bits to LSB
ADD  AH, AL       → 40
                    05
MOV  [1204], AH   → packed value in AH
```

(5) Convert packed BCD to ASCII

```
MOV  AL, [1200]
MOV  AH, AL
AND  AL, 0F
AND  AH, 0F0
```

```
MOV  CL, 04
ROR  AH, CL

OR   AX, 3030

mov  [1204], AX
```

(6) Convert ASCII to packed BCD

```
mov  BX, [1200]
AND  BX, 0F0F

mov  CL, 04
ROL  BH, CL

ADD  BH, BL

mov  [1204], BH
```

3435
⇓
0405

* More Questions

(1) Find the factorial of a number.

```
    mov ax, [2000h]
    mov CX, AX
    mov AX, 0001 H
factloop: MUL CX
          DEC CX
          CMP CX, 0001 H
          JNE factloop
mov [2004], AX
HLT
```

(2) Find the 2's complement of a 32 bit number

```
MOV   AX, 1A2BH
MOV   BX, 3C4DH

NOT AX
NOT BX
ADD AX, 1
ADC BX, 0
```

(3) Swap nibbles of a byte of data.

```
MOV. AL, [2000]
MOV AH, AL
AND AH, 0FH
MOV CL, 04H
ROL AH, CL
AND AL, 0F0H
MOV CL, 04H
ROR AL, CL
MOV [2002], AH
HLT
```

(3) ~~Swap alternating bytes in a bit of data~~

(4) Given an array of numbers, swap the lower and higher nibbles of elements at even positions. (15)

eg. [1A, 2B, 3C, 4D, 5E]

⇓

[1A, B2, 3C, D4, 5E]

```
MOV   SI, 2000
MOV   CX,[SI]
INC   SI
INC   SI
```

SWAPLOOP:

```
MOV. AL,[SI]
MOV  AH,AL
AND  AL,0FH
MOV  CL,04H
ROL  AL,CL
AND  AH, 0F0H
MOV  CL,04
ROR  AH,04H
OR   AH,04H
OR   AH,AL
MOV  [SI], AH
INC  SI
INC  SI
DEC  CX
DEC  CX
JNZ  SWAPLOOP , HLT
```

and 0s

(5) Count the number of 1s, in a byte and store it in a reg

```
mov AX, [2000]
mov BX, 0000H
mov CX, 0000H
mov DX, 0010H   set to 17, otherwise coould be one iteration short
checkloop:
    ROR AX, 1
    JC one

    INC BX
    JMP nextitr

ONE: INC CX

nextitr: DEC CX
        jnz checkloop

mov [2004], BX
mov [2006], CX
HLT
```

(6) Write a program to convert the BCD numbers stored in an array to hexadecimal if the first two digits are 10.

```
mov CX, [2000]
mov SI, 2002
checkloop: mov BL, [SI]
           mov AL, BL
```

0111 1111
1000 0000
1111 1111

1011 1111
1000 0000
1011 1111

```
OR  BL, 80H
CMP  BL, [SI]
JNZ  no change
MOV  AH, AL
AND  AH, 0FH
MOV  BL, AH
AND  AL, 0F0H
MOV  CL, 04
ROR  AL, CL
MOV  BH, 0AH
MUL  BH
ADD  AL, BL
MOV  [SI], AL
nochange:  INC  SI
           DEC CX
           JNZ  checkloop
```

SET 4 - String Programs

① Move a string byte

```
MOV  CX, [1200]     ← length
MOV  SI, 2000       ← source
MOV  DI, 2500       ← dest
CLD                 ←
REP  MOVSB          ← moves contents until CX = 0
HLT
```

(2) Compare a string (Check if 2 strings are equal)

```
mov cx, [2000]   ← length
mov SI, 2500     ← str 1
mov DI, 2700     ← str 2
CLD
REPE CMPSB       ← compare until the string bytes are equal
mov [2800], cx
HLT
```

(3) Scan a string byte. (Find if a character is present in a string)

```
mov cx, [2000]
mov DI, 2000     ← of string
mov AL, [2500]   ← character to look for
INC cx
CLD
REPNE SCASB      ← keep looking until found
mov [2800], cx
HLT
```

(4) Move data from one location to another without using string instructions

```
MOV CX, [2400]
MOV SI, 2000
```

```
        mov   DI, 2500
loop :  mov   AL, [SI]
        mov   [DI], AL
        INC   SI
        INC   DI
        DEC   CX
        JNZ   loop
        HLT
```

(5) Length of a string

Assume that the string terminates with FFH

```
mov   SI, 1200   ← start of string
mov   DX, 0FFH   ← initialize to the max value
mov   AH, OFF H  ← terminating character.
        INC DX
loop:   mov AL, [SI]
        INC SI
        CMP AL, AH
        JNZ loop
```

(6) Reverse a string (equivalent to reversing an array)

```
mov   SI, 2000
mov   DI, 2500
mov   CX, [1000]   ← length
DEC CX
ADD SI, CX  ← go to last index
```

```
loop: mov AL, [SI]
      mov [DI], AL
      DEC SI
      INC DI
      DEC CX
      JNZ loop
      HLT
```

⑦ Palindrome check

```
mov  CX, [2000]
mov  SI, 1200
mov  DI, 1300
DEC  CX
ADD  AX, SI
ADD  AX, CX      } calculate last index
mov  DI, AX
INC  CX
mov  AH, 00H  ← flag (changes if not palindrome)
checkloop:
      mov  BL, [SI]
      mov  DL, [DI]
      CMP  BL, DL
      JZ   nextitr
      mov  AH, 01
```

```
nextitr :   INC  SI
            INC  DI
            DEC  CX
            JNZ  checkloop
mov [2100], AH
HLT
```