8086.

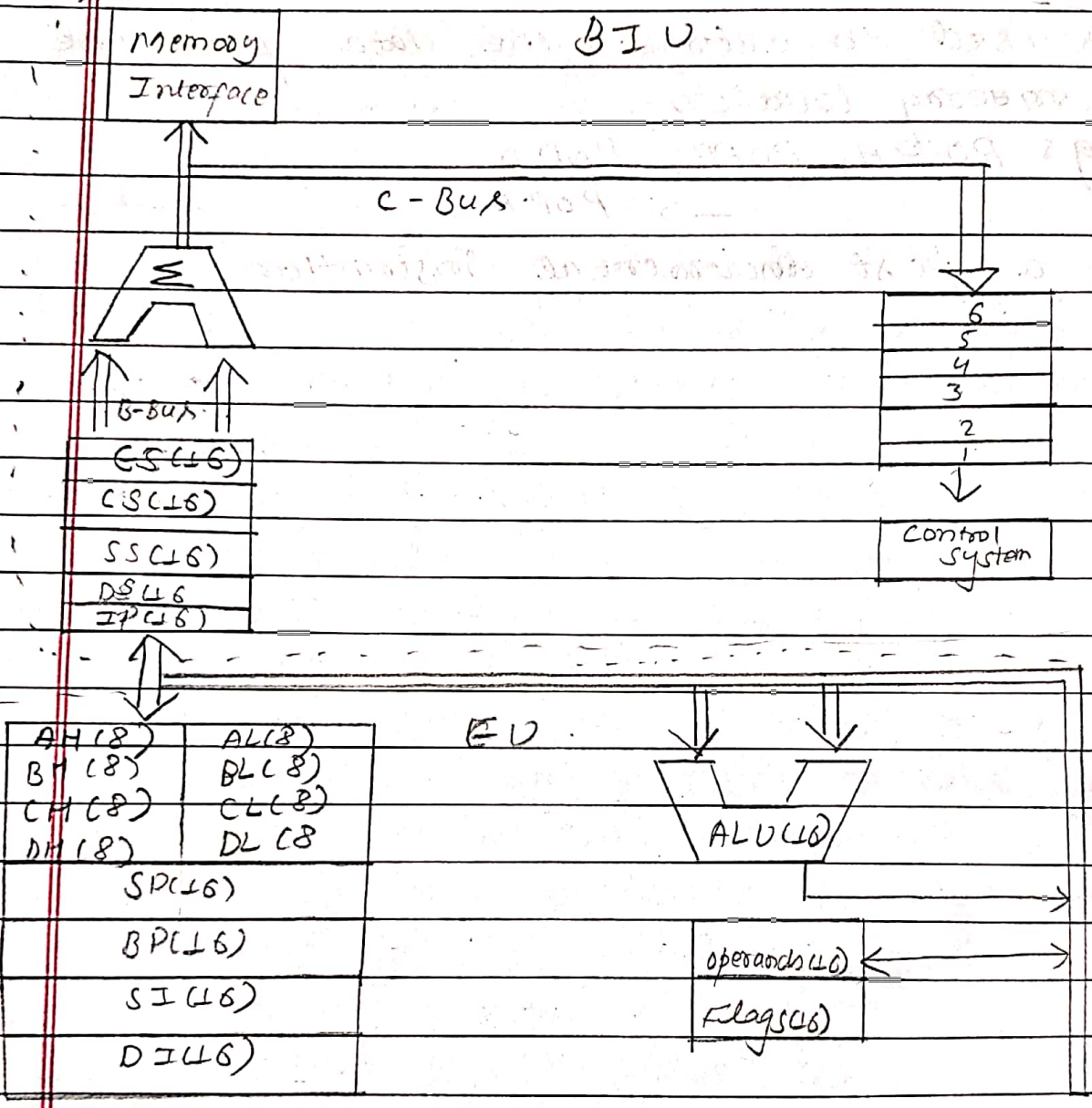


fig: Functional Block diagram / Internal Architecture of 8086

Bus Interface Unit

-> The BIU performs all the external bus operation upto six bytes of instructions can be pre-fetched from memory location in the order FIFO (First In First out). It in order to speed up the execution.

-> The control signals like read, write & I/O device are generated by control system of BIU.

(2) Execution Unit; it performs all the decoding and the execution functions which directs internal operations. The decoder in EU control system and translates the fetched instructions from memory. It has 16-bit ALU that performs arithmetic & logic operations.

Features:-

-> 8086 has 16-bit data bus & 20-bit address bus capable of providing $(2^{20}) = 1$ MB of memory. The address ranges from 00000 ~ FFFFFH.

-> Supports 'pipelining' i.e. fetching the next instruction while executing the current instruction. The 8086 prefetches upto 6 bytes of instructions from the memory at one time and queues them.

-> 8086 supports memory segmentation which means dividing the memory into logical components. The memory is divided into 4 components each having the capacity of 64KB - They are: code segment, Data segment, stack segment & Extra segment.

Init + offset address

Eg:- CS : 2000H; PA = 2000X10H + 1234
Offset : 1234H
= 20000 + 1234
= 21234H.

physical address ; segment address x 10H + Offset address.

BIU

* Segmented Memory.

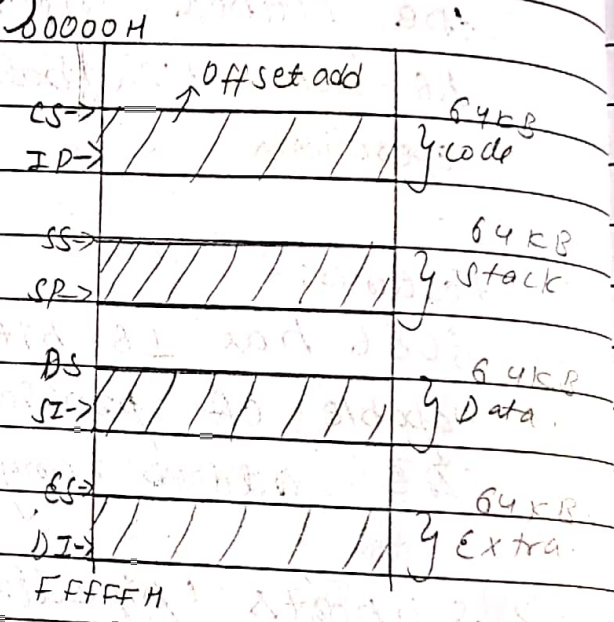
-> The complete physical memory may be divided into a no. of logical segments with each segment of 64K byte.

(1) Code segment (CS) and (IP)

-> Used to hold the program to be executed.

-> Holds the upper bit of starting address.

-> IP holds the 16-bit address or offset of the next codebyte.



(2) Stack segmend and SP

-> Upper 16-bit of starting address for stack memory kept here.

-> SP define the current stack segment.

(3) DS register.

-> Data and operand are stored.

-> Points to the current data segment.

(4) ES (Extra segment)

-> The ES register points to the extra segment which is used to store data when they exceed the size of 64K. It is used by string instruction.

(5) Base Pointer (BP) and index register.

→ In addition to the stack pointer register (SP) the EU contains 16 bit base pointer (BP) register. It also contains 16 bit destination index (DI) and 16 bit source index (SI).

(6) General Purpose Register

→ EU has 8 general purpose register labelled AH, AL, BH, BL, CH, CL and DH, DL. They can be used immediately as 8 bit or in pair to store 16 bit data AX, BX, DX & CX

Addressing Modes of 8086.

(1) Register Direct Mode: For this mode the source and destination are both registers.

i.e. MOV CX, BX

MOV AL, BL

CH | 12 | CL | 34 BH | 12 | BL | 34

(2) Immediate Add Mode: In this mode, the first operand is a register or memory location & second operand is a data.

i.e. MOV AX, 1024H

MOV AH, 24H.

(3) Direct Add Mode: In this mode, one of the operands references a memory location & the other operand is a register.

MOV AX, [5000H]

MOV [7000H], CX

4) Register Indirect Add mode.
-> The address of the memory location which contains data or operand is determined in a indirect way using offset register.
eg:- MOV AX, [BX].

5) Indexed Add, mode:
-> In this mode, offset of the operand is stored in one of the index registers.
eg: MOV AX, [SI]

6) Register - Relative Add mode:
-> In this mode the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the registers.
eg: MOV AX, 50H[BX].

(7) Base plus Index add mode:
-> In this mode, the effective address is formed by adding the content of a base register to the content of index register.
eg: MOV AX, [BX][SI].

(8) Base relative plus index Add Mode:
-> This add. mode, a variation of base-index combines a base register, index register and a displacement to form effective address.
eg: MOV AX, 50H[SI][BX].

Flags of 8086:

control Flags.

X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

-> Flag register consists of different flags whose status shows the properties of the result of ALU operation. The 8086 has 9 such flags out of which 6 are status flags and 3 are control Flags. The control bits in the Flags are set or Reset by the programmer.

1) CF (Carry Flag); it is set if the result of last operation generates a carry in the most significant bit otherwise Reset.

2) OF (overflow Flag); it is set if the result of a signed number operation is of large capacity.

3) Sign Flag (SF); it is set if the MSB of the operation is 1.

4) Auxillary Carry (AC); it is set if there is carry from lower half of the byte to the upper next half.

5) Parity Flag (PF); It is set if the result of the last operation has even number of 1's otherwise Reset.

6. Zero Flag (ZF); it is set if the result of the last operation is zero.

7. Direction Flag (DF); when this flag is set, it causes the string instruction to decrement and clearing DF will cause the instruction to auto-increment.

8. Trap Flag (TF); when this flag is set, 8086 goes into single step mode.

9. Interrupt Flag (IF); it is used to allow or prohibit the interruption of the program.

Imp

Assembler Directives:

→ Their main table is to inform the assembler about start/end of the segment, procedure or program, to reserve the appropriate space for the data storage etc.

→ They are called pseudo-instructions because they seem like instruction in program but are not instruction in real and are converted into object code during assembling.

→ Some of the common directives are:-

PAGE Directive.

→ It helps to control the format of a listing of an assembled program.

→ It is an optional directive.

→ At the start of the program, PAGE Directive designates

the maximum no of lines to list on the page and maximum no of characters on a line.

-> It's format is PAGE[LENGTH], [WIDTH]

(2) TITLE DIRECTIVE.

This directive defines the title of the program so the text typed to the right of this directive is printed at the top of the each page.

-> It's format is TITLE[TEXT]

eg: TITLE "program to add two numbers".

(3) DOSSEG

There is a standard order for placing the code, data and stack. This directive tells the assembler to place the segment in standard order.

(4) MODEL

-> This directives determine the size of the each segments

	Model	Description.
•	Tiny	-> CODE and data segment together can't be greater than 64 KB.
•	SMALL	-> Neither CODE nor DATA may be > 64KB.
•	Medium	-> only code > 64K
•	COMPACT	-> only data > 64K
•	LARGE	-> Both > 64K
•	HUGE	-> All available memory can be used for code & data.

(5) Stack: It sets the size of the program stack which may be upto 64K. A size of 100h is more than enough for most programs

(6) Code; It defines the part of the program that contains instructions.

(7) PROC: This directive creates the name and an address for the beginning of the procedure which is given a common name.

(8) ENDP; This directive marks the end of the procedure, it's name must match name of the PROC directive.

(9) DATA: All variables are declared in the area following DATA called data segment.

(10) END: The END directive terminates the assembly of the program and any statements after it are ignored.

Assembly Language Programming.

→ Assembly Language Programming represents the program level in between the machine level & high level languages. This language are represented by the instructions that are use english language type of statements which makes the programmer more convenient to write and understand instructions.

We use assembler to translate the source program into machine code known as "object code". Finally a linker program is used to complete the

machine addressing object code to generate executable format. The process of converting source code into object code is called assembling which is carried out by assembler.

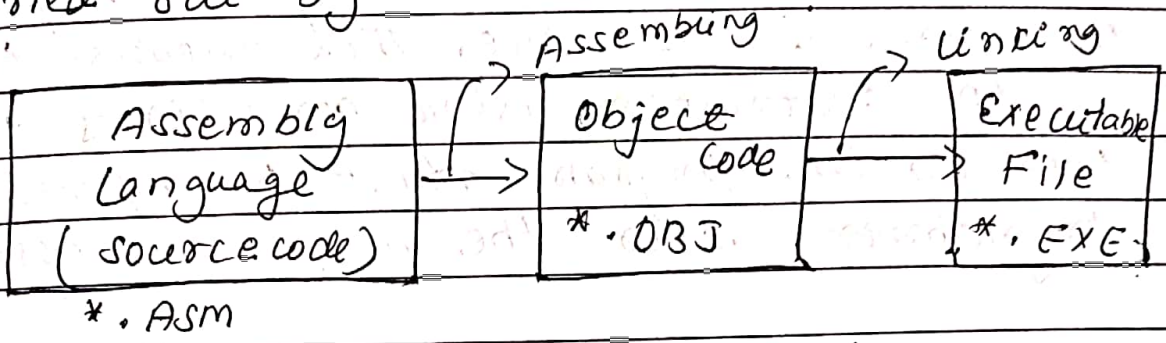


fig: process of Assembling.

(1) Assembler: The assembler converts the source code (*.ASM) into machine/object code (*.OBJ) & displays any error message on the screen. The process is called assembing. Types of assembler are:- 1) MASM (Macro Assembler)
2) TASM (Turbo Assembler).

(2) Co Linker; After assembling, if the program faces an error free message, the next step is to link the object code, that was created by the object code. This is called linking process.

(3) Executing: This is the last step to load the program for execution. The program having assembled and linked will be now executed. The program is first loaded into the memory before execution which will generate the (*.EXE) file.

One pass Assembler:

-> This assembler reads the entire source program (*.ASM) once and translates the code. This assembler has the problem of defining the forward reference. This means that a JUMP instruction using an address that appears later on the program must be defined by the programmer after the program is assembled.

Two Pass Assembler; It reads/scans the assembly ~~language~~ program twice.

-> The first pass generates the tables of the symbols which consists the labels and address assigned to them.

-> The second pass uses the symbol table generated in the first pass to complete the object code.

MASM (Macro Assembler).

-> It translates the program written in macro language to machine language.

-> It is just a block of code that can be used with some input parameters anywhere in the program just like a function.

-> They execute faster & reduces complexity (space & time).

eg: ADD MACRO P1, P2...

Macroname ↑

END M (End of Macro).

Macros Assembling.

→ A macro is a specific name assigned in an instruction sequence that appears repeatedly each time we call the macro in our program. The assembler will insert the defined group of instruction in place of that call. An important point here is that the macro assembler (MASM) generates machine code for the group of instruction each time the macro is called:-

```
ADDITION MACRO
    IN AX, PORT
    ADD AX, BX
    OUT PORT, AX.
    END M:
```

Here, "ADDITION" name is assigned for macro operation. To execute above repeatedly, macro assembler allow the macro name only to be type instead of all the instruction. The advantage of using macros are that the source program becomes shorter of program documentation becomes better.

It is a two pass-assembler because it reads the source program two times. In the first pass the source program symbol tables are constructed with the name of location of all labels and variables.

In second pass, all remaining references to symbol location will be resolved.

Imp Q) Difference between MACROS & PROCEDURE (Subroutine)

MACROS	PROCEDURE (Subroutine)
1. Macros are used for short sequence of instructions.	1. Procedure / subroutine are used for long seq. of instr
2. Macros don't require RACE & RETURN instruction.	2. It requires CALL & RETURN instruction.
3. Macros execute faster than PROCEDURE.	3. They are slower compared to MACROS.
4. Parameters are assigned as a part of statements.	4. Parameters are assigned in registers, memory locatio
5. Control of execution remains in the main program.	5. Control of execution is transferred to subroutine.
6. Machine code is generated everytime a macros is called.	6. Machine code is generated only once.
7. It eliminates the overhead time required.	7. It requires overhead time for CALL & RETURN.

8086 Interrupts & Interrupt vector Table.

=> The interrupt of entire Intel family of MPU include two hardware pins that require & request interrupts. (INTR and NMI)

=> An 8086 interrupt can come from any sources. One of them is external signal applied to the NMI (Non-maskable Interrupt) / Hardware generated interrupt.

=> A second source of interrupt is execution of the interrupt instructions (software generated).

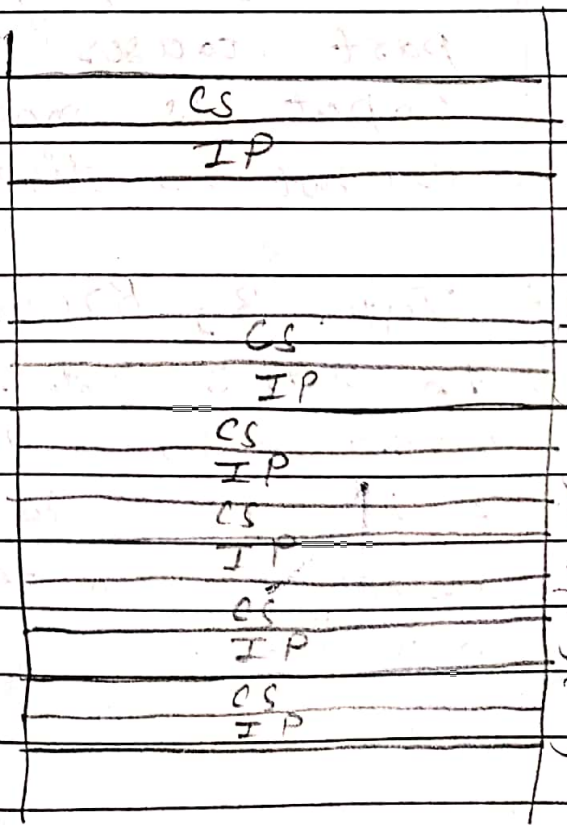
=> A third source of an interrupt is some error condition produced by 8086.
eg:- Divide by zero, unused opcode.

Interrupt Vectors.

=> The interrupt vector Table is located in the first 1024 bytes of memory at address 00000H - 000FFH.

=> This contains 256 interrupts.

=> Each vector is 4 byte long which contains the starting address of the ISR (Interrupt Service Routine)



1. Type 0, Divide by zero

→ Divide error occurs when the result of division overflows or whenever an attempt is made to divide by zero. Type 0 in 8086 is already set and cannot be disabled.

2. Type 1, Single Step Interrupt

→ If the 8086 trap flag is set, 8086 will automatically do a type 1 interrupt after each instruction executes.

→ In other words in single step mode, the system will stop after it executes each instruction and waits for further direction.

3. Type 2. Non-maskable Interrupt.

→ A result of placing logic 1 on the NMI part causes type 2 interrupt this input is non-maskable which means it cannot be disabled using EI/DI instruction.

4. Type 3, Breakpoint Interrupt.

→ The INT 3 instruction is often used to store a breakpoint in a program for debugging.

→ A breakpoint is used to examine the CPU and memory after the execution of a group of instructions.

5) Type 4; Overflow interrupt.

→ The INTO instruction interrupts the program if an overflow condition exists as shown by overflow flag.

→ There is an instruction associated with this INTO (Instruction on overflow).

Interrupt Priority of 8086.

⇒ To handle the situation of occurence of two or more interrupts at same time, priority has been fixed for the interrupts 8086 here has the highest to lowes in the order of

Interrupts	Priority
* Divide error, INTO	Highest
* NMI	⋮
* INTR	⋮
* single step	Lowest

Program to ADD two datas in 8086.

```
title add two datas
dosseg.
.model small
.stack 100h
.code
add  PROC                    -> Initialize data segment
{    mov ax, @data
     mov ds, ax.
```

```
mov al, val1 ; al ← 06
Add al, val2 ; al ← 06+12
mov result, al ; [result] ← 18
{mov ax, 4c00h } ; .exit  it terminates the program.
{int 21 h.
add endP
.data
    val1 db 06h
    val2 db 12h.
    result db ?
.end add
```

* WAP in 8086 to display the string "computer Engineering" at console.

=> title Display the string :-

```
dosseg
.model small
.stack 100h
.code
main proc
    mov ax, @data ; To initialize the data segment
    mov ds, ax
    mov ah, 09h
    mov dx, offset string ; For string o/p of console
    int 21h

    mov ax, 4c00H ; .exit
    int 21 h
main endp.
```

• data
string db 'computer Engineering', '$'
end main.

Q) WAP in 8086 to display the string "HELLO WORLD" without using any instruction

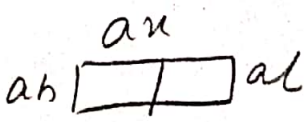
=> Title display the string.

dosseg
• model small
• stack 100h
• code
main proc
mov ax, @data.
mov du, ax
mov cx, len ; [cx] = LL
mov si, offset string
mov dl, [SI] ; [dl] = H, E, L.

[SI] = source index
DI = Destn Index

Character output { mov ah, 02h
int 21 h;
inc si
loop Again -> main endp

• data
string db "Hello world", '$'
Len dw $ - string.
end main



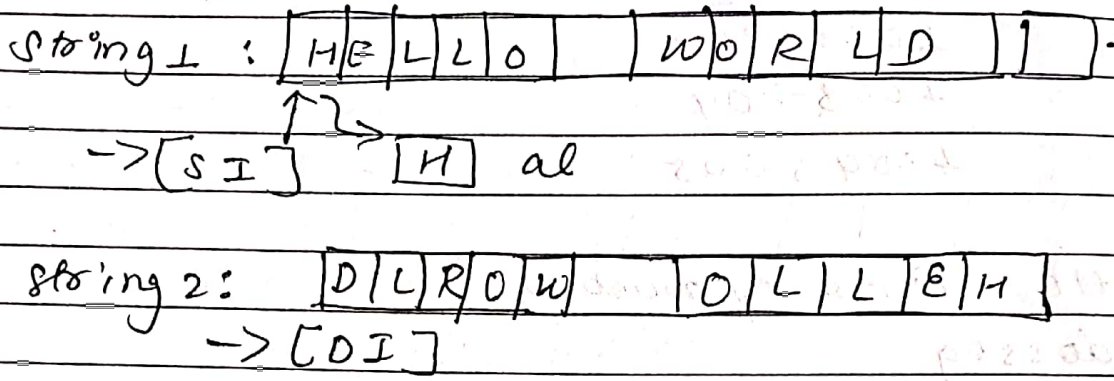
Add array of 5 numbers given by user.

```
title add numbers.
  dosseg
• model small
• stack 100h
• code
    main proc
initialize { mov ax, @data
data segment { mov dx, ax ← counter
          mov cx, 05h
          mov au, 00h
          mov bu, offset Array
          add al, [bx]
          inc bx
          dec cu ; [cu] = 04
          jnz loop
          mov sum, al ; sum = 15 of
          mov au, 4c00h } exit
          int 21h
          main endp
• data
  Array db 0, 01, 02, 03, 04, 05
  sum db ?
  end main.
```

Sum of Five numbers stored in memory location from 1000 -> 01
1001 -> 02
1002 -> 03
1003 -> 04
1004 ; -> 05

```
Title  Sum of numbers:
  dosseg
.model small
. stack 100h
. code
Add proc
  mov ax, @data
  mov dx, ax
MOV SI, Address
mov CL, 05H  //MOV CL, COUNTER
mov al, [SI] ; AL = 01
ADD BL, al ; BL ← 00+01
  INC SI ; SI = 10001
Dec CL ; 04
JNZ loop
  MOV ~~AL,~~ BL SUM, BL
  MOV ax, 4c00h
    int 21h
    Add endp.
  . data
    Address DW 1000H
    sum db ?  ; counter db 05.
end add
```

Q) Write an ALP in 8086 to reverse the string "HELLO_WORLD" i.e. DLROW OLLEH




```
dosseg
.model small
.stack 100h
.code
main proc
mov ax, @data
mov dx, ax
mov es, ax
mov cx, counter
mov si, 0
mov di, 0
add di, counter ; di = 11+1 bit
dec di ; di = 11th bit.
loop: mov al, string1 [si] ; [al] <= H.
mov string2 [di], al ; [di] <= H
inc si
dec di
JNZ loop
```

Initial data segment

extra segment

since we are using both SI & DI

cx = 11

```
mov ax, #c00h
int 21h
main endp
.data
string 1 db 'Hello world', '$'
string 2 db ?
counter db 11
end main
```

Data Definition Directive.

→ In assembly language, we define storage for variables using data definition directives. Data defn directive create storage at assembly time.. Different data directives are:-

DB/db; define byte (8-bit data)
DW/dw; define word (2-byte)
DD: define double word (4-byte)
DQ: define quad word (8-byte)
DT: define ten bytes.

Serial I/O lines, SOD & SID.

→ The 8085 µp has two pins specially designed for software controlled serial I/O. One is called SOD (serial output Data) and the other is called SID (Serial Input Data)

→ Data transfer is controlled through two instruction SIM & RIM

- SOD, The instruction SIM (set Interrupt Mask) is necessary to send the data through output serially from SOD line. It can be interpreted for serial output as;

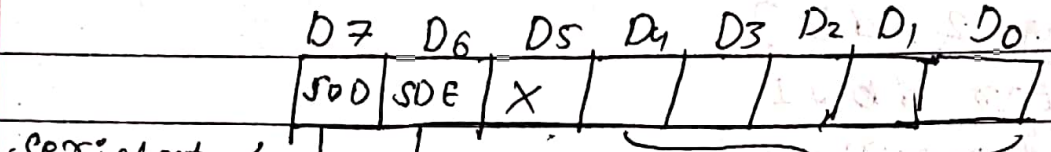


fig: control word/Register for SOD.

- SID: The RIM (Reset Interrupt mask) is used to serial data through the SID line. It can be interpreted as,

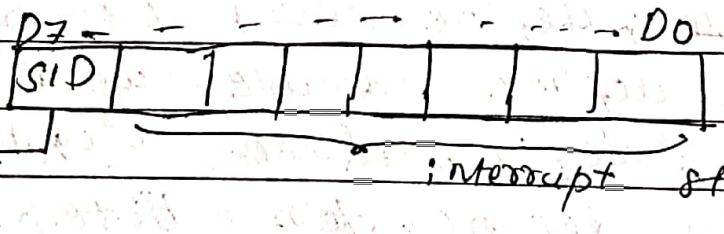


fig: Control word/Register for SID/ RIM instn for SID.

→ SID & SOD lines eliminates the use of input & output port in the software ~~enable~~ enabled serial I/O. The SID is 1-bit input port and SOD is 1 bit output port.