**CS321 Mid Semester**

Name: P. V. Sriram

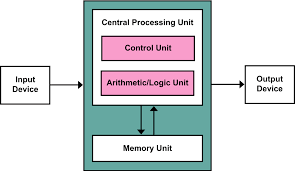
Roll No.: 1801CS37

**A1)**

Computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. Computer architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation and is concerned with balancing the performance, efficiency, cost, and reliability of a computer system

# Von Neuman Architecture

The von Neumann architecture also known as the Stored program computer or Princeton architecture is a computer architecture based on a 1945 description by John von Neumann.



The design architecture here consists of:

* Central Processing Unit (CPU)
* Immediate Access Store (IAS)
* Input / Output (I/O)

**Central Processing Unit**

The CPU governs the computer and handles the data.  It has four main parts as follows:

* **Arithmetic Logic Unit (ALU)**
  + This part handles all the arithmetic and logic operations such as calculations and comparisons.
* **Control Unit**
  + This part handles the movement of instructions to and from the memory and the execution of instructions one at a time.
* **Registers**
  + There are different types of registers depending on the information being stored.
    - Program Counter – stores location of next instruction
    - Current Instruction Register – stores current instruction being implemented
    - Accumulators – store results of calculations and comparisons
    - Status Register – stores data about the last operation
    - Interrupt Register – stores information regarding an interruption that occurred
* **Clock**
  + Instructions are executed to the beat of the clock. The faster the clock, the faster the computer is.

**Immediate Access Store**

This is commonly referred to as the Random-Access Memory (RAM) or Main Memory.  It stores both instructions and data. Buses which allow the movement of instructions and data between different parts of the computer is called a data bus.  Buses that detect locations in memory is called an address bus.

**Input / Output**

I/O refers to the accessories for inputting and outputting of data.  A computer needs to read in data and send out data through I/O ports.  An I/O controller is an interface that allows a user to attach any I/O device to the computer and send data in or out of the computer.

# Dataflow Architecture

In data flow architecture, the whole software system is seen as a series of transformations on consecutive pieces or set of input data, where data and operations are independent of each other. In this approach, the data enters into the system and then flows through the modules one at a time until they are assigned to some final destination (output or a data store).

There are three types of execution sequences between modules−

* Batch sequential
* Pipe and filter or non-sequential pipeline mode
* Process control

**Batch Sequential**

In Batch sequential, separate programs are executed in order and the data is passed as an aggregate from one program to the next.

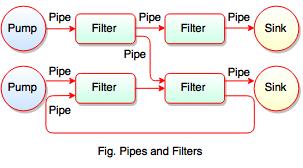
It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.



**Pipe and Filter**

Pipe is a connector which passes the data from one filter to the next. They are a directional stream of data implemented by a data buffer to store all data, until the next filter has time to process it.

Filter reads the data from its input pipes and performs its function on this data and places the result on all output pipes. If there is insufficient data in the input pipes, the filter simply waits. All filters are the processes that run at the same time, it means that they can run as different threads, coroutines or be located on different machines entirely.

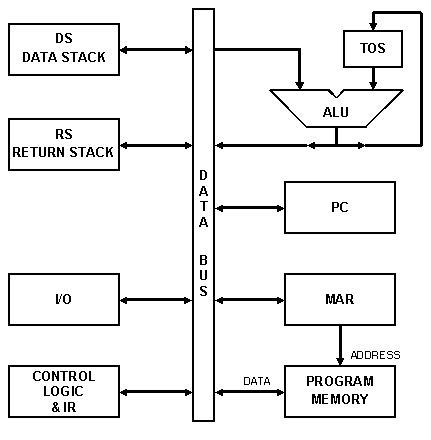


**Process Control**

Process Control Architecture is a type of Data Flow Architecture, where data is neither batch sequential nor pipe stream. In process control architecture, the flow of data comes from a set of variables which controls the execution of process.

# Stack Machine

Stack machine is a mode of computation where executive control is maintained wholly through append (push), readoff and truncation (pop), of a first-in-last-out (FILO, also last-in-first-out or LIFO) memory buffer, known as a stack, requiring very few processor registers.



In the stack machine, data is available at the top of the stack by default. The stack acts as a source and destination, push and pop instructions are used to access instructions and data from the stack.

In a stack machine, the operands used in the instructions are always at a known offset (set in the stack pointer), from a fixed location (the bottom of the stack, which in a hardware design might always be at memory location zero), saving precious in-cache or in-CPU storage from being used to store quite so many memory addresses or index numbers.

--------------------------------------------------------------------------------------------------------------------------

**A2)**

# Leap Year or not

I have formulated an 8085-assembly language program to identify is an input year is a leap year or not. The rule is the year should either be divisible with 400 or 4.

The inputs are given into 2050(lower byte) and 2051(higher byte). Firstly, we check if lower byte is 0. If yes, we should check if the higher byte is divisible by 4. Together, these two rules verify the divisibility with 400. If the lower byte is not 0 we should check if it is divisible with 4 which verifier divisibility with 4.

# Code

# ORG 2000

LXI H,2050

MOV A,M // B<-M

CPI 00

JNZ LB

MVI C,00 // C<-00H

INX H

MOV A,M // A<-M

ITER: CPI 04

JC STORE // check for carry

SUI 04 // A<-A-B

INR C // C<-C+1

JMP ITER

STORE: STA 3050 // 3050<-A

MOV A,C // A<-C

STA 3051 // 3051<-A

HLT // terminate the program

LB: CPI 04

JC STORE // check for carry

SUI 04 // A<-A-B

INR C // C<-C+1

JMP LB

# ORG 2050

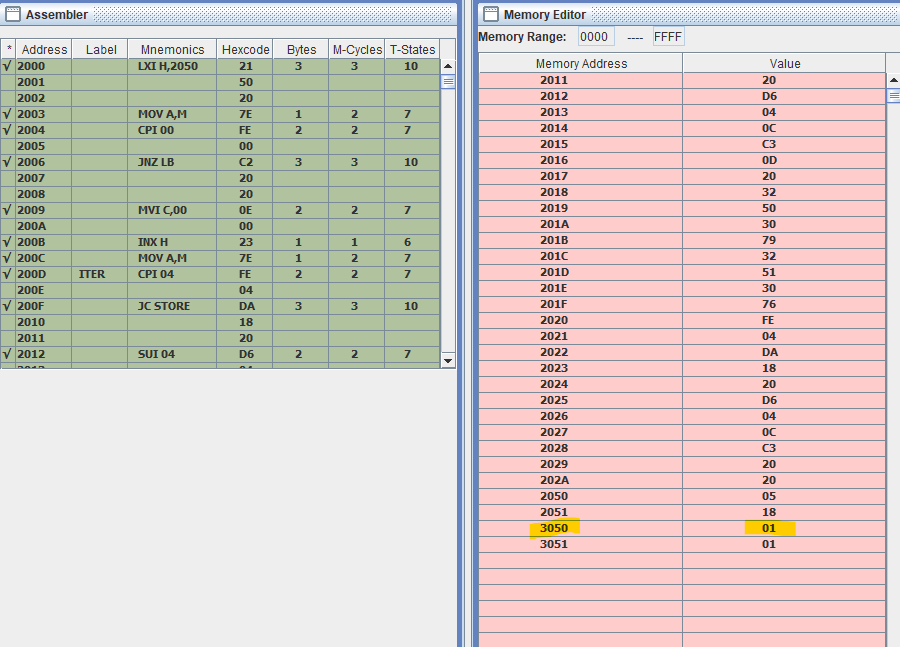
# DB 05, 24D

**Input**

**2405 D (Address pair - 2051H, 2050H)**

**Output**

**1 (3050H) (0 at 3050H indicates leap year, A non-zero number at 3050H indicates a non-leap year)**



--------------------------------------------------------------------------------------------------------------------------

**A3)**

# Date Difference function

I have devised an 8086-assembly language to find the number of days between two dates from the same year. The algorithm is to first calculate the number of days between the two dates and a reference date (1st Jan) and subtract the resultant numbers.

# Code

.model small

.stack 64

.data

date1 dw 31D, 01D

date2 dw 28D, 04D

.code

date:

MOV AX, @data ;Load Data in temp register

MOV DS, AX ;Load address into data segment

LEA SI, date1 ;Load first date pointer

LEA DI, date2 ;Load second date pointer

MOV AX, [SI] ;Load first date

MOV BX, [DI] ;Load second date

SUB BX, AX ;BX <- BX - AX

MOV DX, BX ;DX <- BX

month:

INC SI ;Increment pointer

INC SI ;Increment pointer

MOV AX, [SI] ;Load first month

MOV CX, AX ;CX <- AX

DEC CX ;Decrement CX

JZ next\_month ;If zero, Jump to next\_month

CMP AX, 03D ;Compare first month to 3

JC bef\_feb\_1 ;If carry, Jump to bef\_feb\_1

SUB CX, 02 ;CX <- CX - 2

SUB DX, 59D ;DX <- DX - 59

bef\_feb\_1:

SUB DX, 31D ;DX <- DX - 31

MOV AX, CX ;AX <- CX

MOV BL, 02D ;BL <- 02D

DIV BL ;AL <- AH / BL remainder

CMP AH, 00 ;Compare AH with 00

JNZ next\_month ;If not zero jump to next month

INC DX ;Increment DX by 1

LOOP bef\_feb\_1 ;Loop until CX = 0

next\_month:

INC DI ;Increment pointer

INC DI ;Increment pointer

MOV AX, [DI] ;Load second month

MOV CX, AX ;CX <- AX

DEC CX ;Decrement pointer

JZ exit ;If zero, jump to exit

CMP AX, 03D ;Compare second month to 3

JC bef\_feb\_2 ;Jump if carry to bef\_feb\_2

SUB CX, 02 ;CX <- CX - 2

ADD DX, 59D ;DX <- DX + 59

bef\_feb\_2:

ADD DX, 31D ;DX <- DX + 31

MOV AX, CX ;AX <- CX

MOV BL, 02D ;BL <- 02

DIV BL ;AL <- AH / BL remainder

CMP AH, 00 ;Compare AH with 00

JNZ exit ;If not zero jump to next month

DEC DX ;Increment DX by 1

LOOP bef\_feb\_2 ;Loop until CX = 0

exit:

HLT ;Terminate

end date

.end

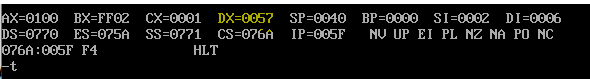
**Input**

**Date 1: 31D, 01D – 31st Jan**

**Date 2: 28D, 04D – 28th April**

**Output**

**57H – 87 Days (DX Register)**



--------------------------------------------------------------------------------------------------------------------------

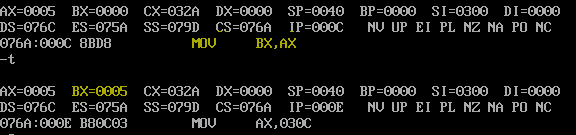
**A4)**

# Addressing Modes

**Register Mode**

Register mode is where the source of operands are registers themselves. We specify the name of the register which holds the data to be operated in the instruction

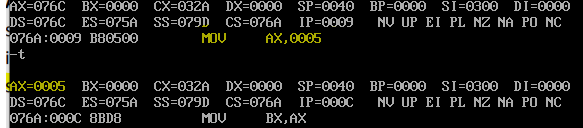
E.g. MOV BX, AX To move content of AX register into BX



**Immediate Mode**

Immediate mode is where the source operand is an 8 / 16-bit number and the destination operand are registers.

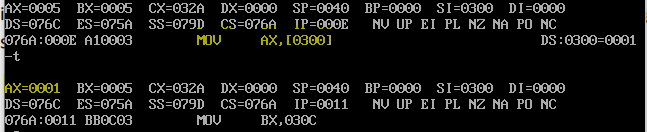
E.g. MOV AX, 05H To move value 05H into AX register



**Direct Mode**

Direct addressing mode is where the effective address of the memory location (16-bit number) at which data operand is stored is directly given in the instruction.

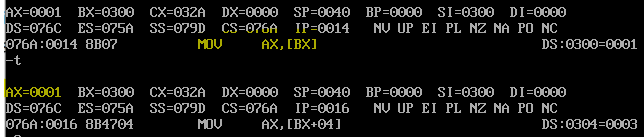
E.g. MOV AX, ds:[0300H] To move the content from effective address 0300H to AX



**Indirect Mode**

Indirect addressing mode allows data to be addressed at any memory location through an offset address held in any of the following registers: BP, BX, DI and SI. We use these registers to hold the effective address to be specified in the instruction

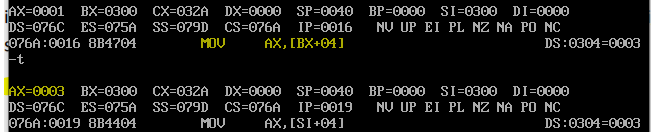
E.g. MOV AX, ds:[BX] To move the contents from Address (EA) stored in BX to AX



**Based Mode**

Based index mode is where the offset address of the operand is given by the sum of contents of the BX/BP registers and 8-bit/16-bit displacement. When BX holds the base value of EA, 20-bit physical address is calculated from BX, DS and if BP holds the base value, BP and SS are used.

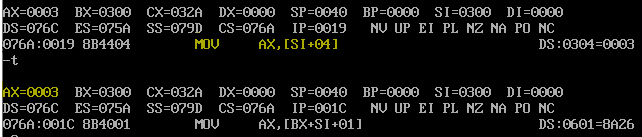
E.g. MOV AX, ds:[BX + 04H] Here, Effective address is BX + 04H



**Indexed Mode**

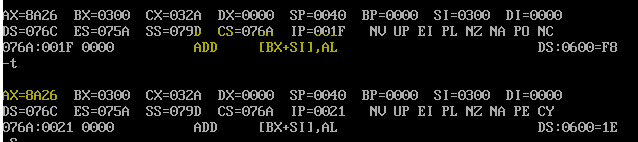
Indexed mode is where the offset address of the operand is given by the sum of contents of SI / DI registers and b-bit/16-bit displacements. SI / DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

E.g. MOV AX, ds:[SI + 04H] Here EA is SI + 04H



**Based-Index Mode**

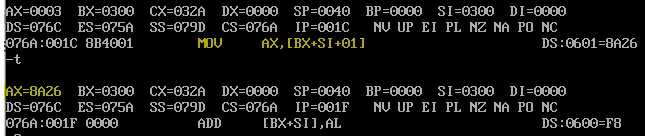
Based Index mode is where the offset address of the operand is given by the sum of contents of BX / BP (base register) and SI / DI (index register). SI / DI register is used to hold an index value for memory data and BX / BP store the base value of EA.



**Based Index Displacement Mode**

Based Index Displacement mode is where the offset address of the operand is given by the sum of contents of BX / BP (base register) and SI / DI (index register) and an 8-bit / 16-bit displacement. SI / DI register is used to hold an index value for memory data and BX / BP store the base value of EA.

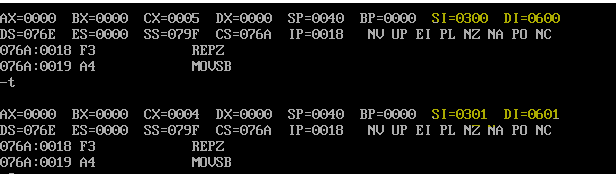
E.g. MOV AX, ds:[BX + SI + 01H]



**String Mode**

String mode is used to operate on string data. In this the value of SI and DI are auto incremented and decremented depending upon the value of directional flag. The effective address of the source data is stored in SI register and the EA of destination is stored in DI register; Base address is the DS.

E.g. REP MOVSB



**Input/Output Mode**

Input/Output Mode is used to access data from the standard I/O mapped devices or ports. In Direct I/O addressing, 8-bit address is used to specify the periphery location. In Indirect I/O addressing, we use the DX register to store the I/O address.

E.g. IN AL, 80H



**Relative Mode**

Relative addressing mode is where the EA of a program instruction is specified relative to Instruction pointer (IP) by an 8-bit signed displacement.

**Implied Mode**

Implied mode is where data operand is a part of the instruction itself. There are no operands here as the instructions themselves specify the data to be operated.

E.g. CMP AL, BL



# Code

.model small

.stack 64

.data

org 300H

arr dw 0001, 0002, 0003, 0004, 0005

.code

start:

MOV AX, @DATA ;Load Data in temp register

MOV DS, AX ;Load data into data into Data Segment

LEA SI, arr ;Load data into SI

MOV AX, 05H ; Immediate Addressing, moving 5 into AX

MOV BX, AX ; Register Addressing, moving AX content into BX

MOV AX, ds:[0300H];Direct Addressing, moving [BX] content into AX

MOV BX, 0300H; Immediate Addressing, moving 300 into BX

MOV AX, ds:[BX];Indirect Addressing, moving [BX] content into AX

MOV AX, ds:[BX + 04H];Base Addressing, moving [BX + 04] content into AX

MOV AX, ds:[SI + 04H];Index Addressing, moving[SI + 04] content into AX

MOV AX, ds:[BX + SI + 01H];Base Index addressing

end start

.end

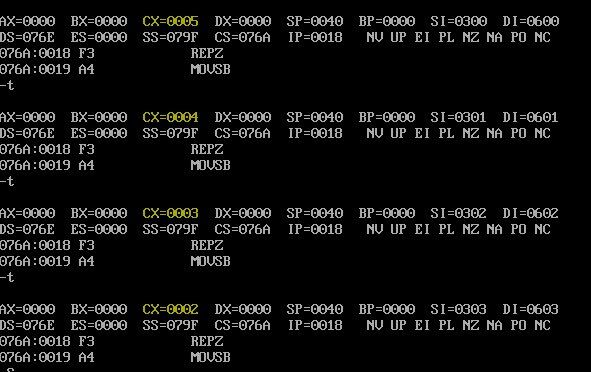
--------------------------------------------------------------------------------------------------------------------------

**A5)**

# String operations

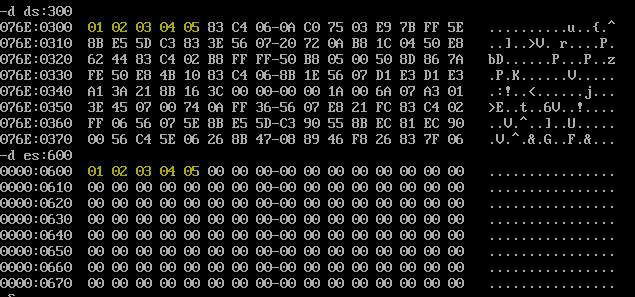
**REP**

Repeat a given instruction till CX=0



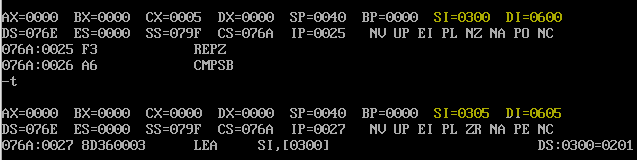
**MOVS**

Moves the contents of byte given by DS:SI into ES:DI



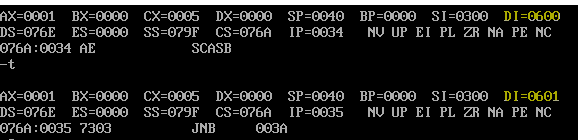
**CMPS**

Compares byte at ES:DI with word at DS:SI and sets flags



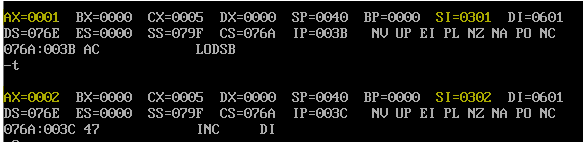
**SCAS**

Compares byte at ES:DI with AL and sets flags according to result



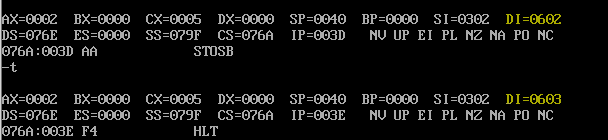
**LODS**

Moves the byte at address DS:SI into AL; SI is incr/decr by 1



**STOS**

Moves contents of AL to byte address given by ES:DI; DI is incr/dec by 1



# Code

;REP, MOVS,CMPS, SCAS, LODS, and STOS

.model small

.stack 64

.data

org 300H

arr db 01H, 02H, 03H, 04H, 05H

.code

same:

MOV DX, 01H

start:

MOV AX, @DATA ;Load Data in temp register

MOV DS, AX ;Load data into data into Data Segment

movs\_:

LEA SI, arr ;Load data into SI

MOV DI, 600H

MOV AX, 00H

MOV ES, AX

MOV CX, 05H

CLD

REP MOVSB

cmps\_:

LEA SI, arr ;Load data into SI

MOV DI, 600H

MOV CX, 05H

CLD

REPE CMPSB

scas\_:

LEA SI, arr ;Load data into SI

MOV DI, 600H

MOV CX, 05H

MOV AL, 01H

CLD

SCASB

JNC lods\_

MOV DX, 01H

lods\_:

INC SI

LODSB

stod\_:

INC DI

STOSB

HLT

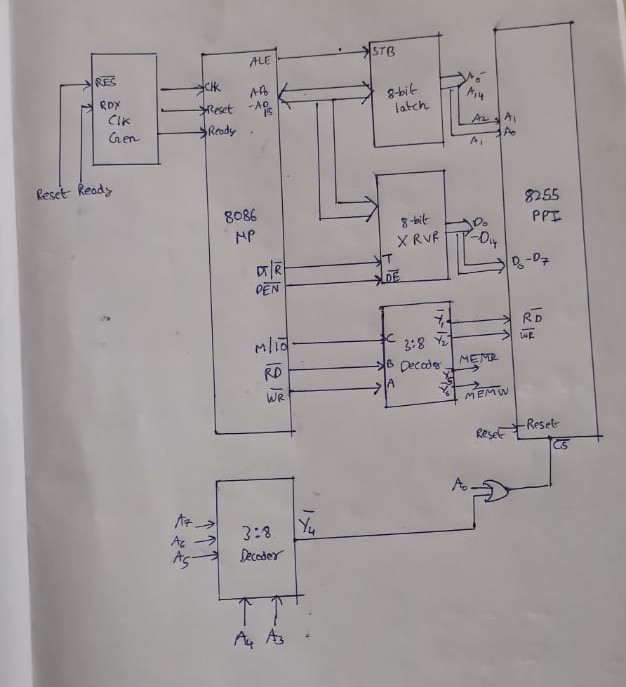
end start

.end

--------------------------------------------------------------------------------------------------------------------------

**A6)**

**Interfacing diagram of 8086, 8255 PPI**



# Code

.model small

.stack 64

.data

ORG 3000H

tem db 64

.code

start:

MOV DX, 3000H

IN AL, DX

CMP AL, 64H

JNZ START

INC DX

OUT DX, AL

skip:

HLT

end start

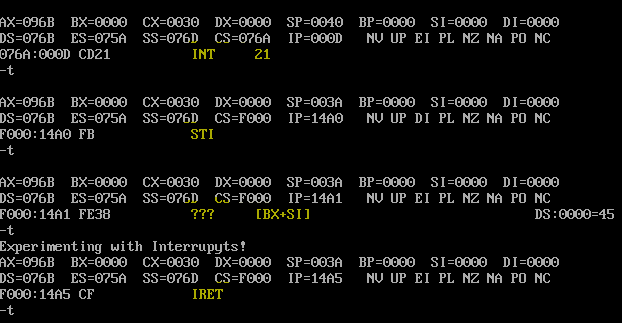
.end

--------------------------------------------------------------------------------------------------------------------------

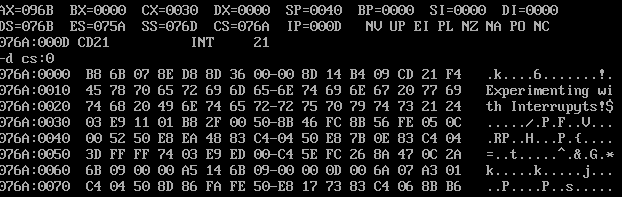
**A7)**

# Interrupt for printing Data (INT 21H)

We experiment here with the interrupt which helps in printing the data onto the console



**Pre interrupt Phase:**

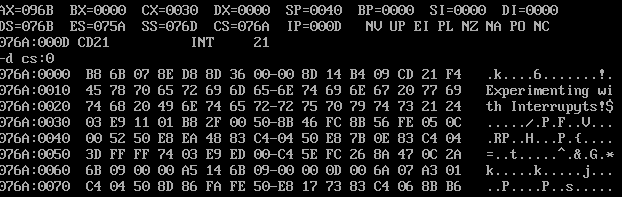


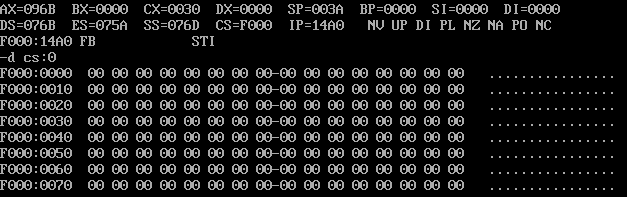
As we can see there are initially few instructions present in the code segment before the interrupt call and the SP points to 0040

As we can see, interrupts are executed in 3 parts, namely

1. **STI (Set interrupt flag)**

Here, the Interrupt Flag (IF) in the EFLAGS is set. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed.

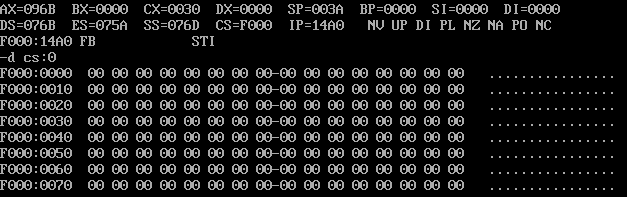


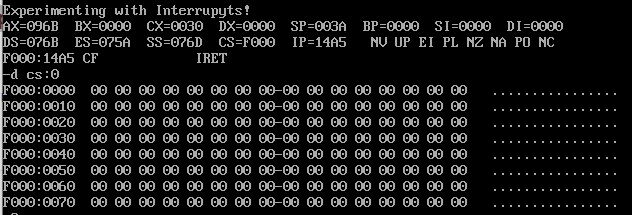


We can see that the Code segment has changed and SP now points to 003A. Due to this step, the processor registers and device code is saved.

1. **Execute Device code**

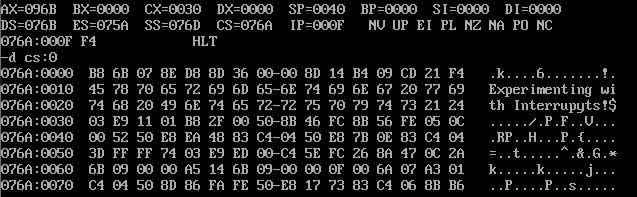
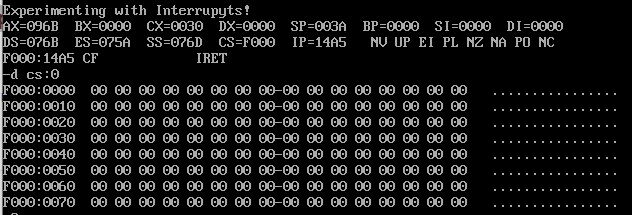
Here, the necessary peripheral device necessities are fulfilled and we can see the change in IP (Instruction pointer) to prove that. We can also see the line “Experimenting with interrupts” to say that the functions are executed





1. **IRET (Return to original program)**

Here, we come back to the initial program and continue the usual execution. We can show that by the change in CS content, Change in IP and SP.



# Code

.model small

.stack 64

.data

start db "Experimenting with Interrupyts!" , "$"

.code

MOV AX, @data ;Load Data in temp register

MOV DS, AX ;Load address into data segment

LEA SI, start ;Load first date pointer

lea dx , [SI]

mov ah , 09h

int 21h

HLT

end

.end