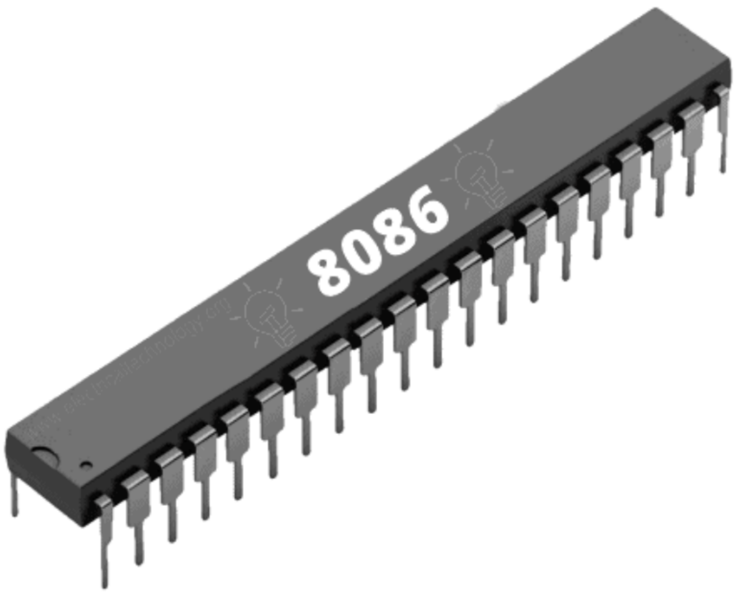
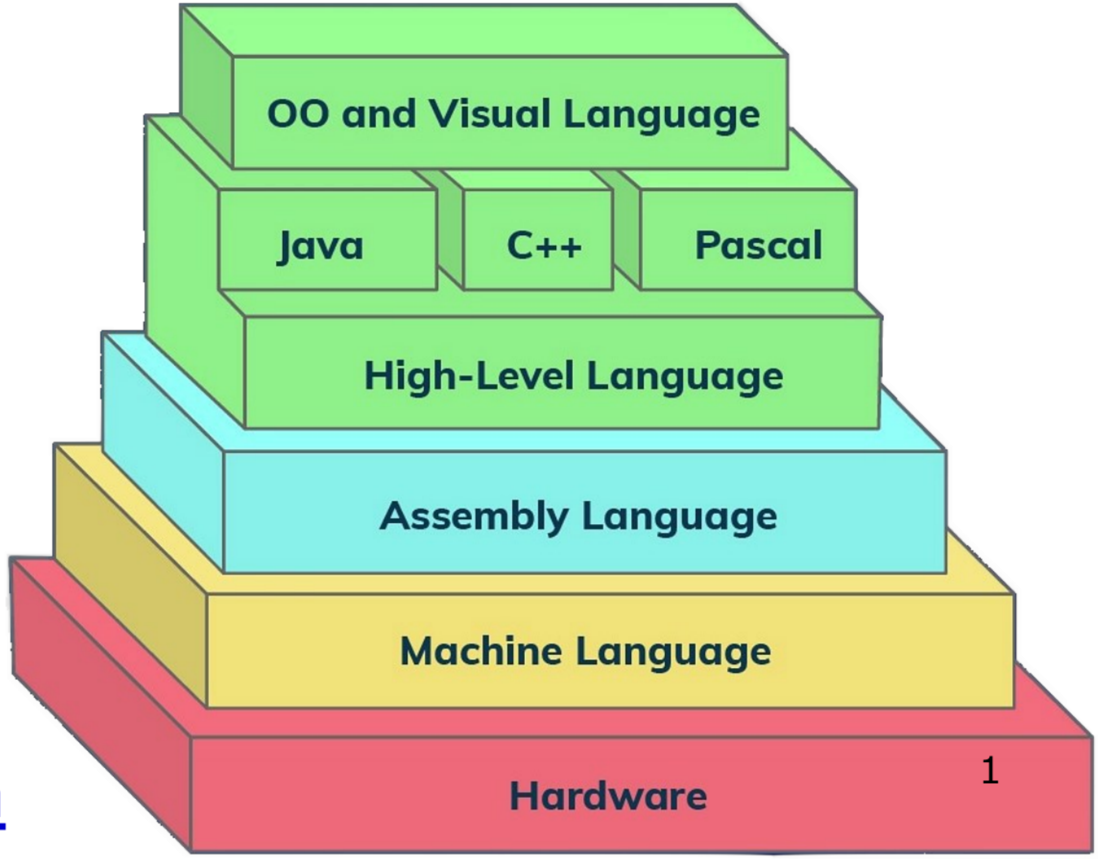
CSec15233
Malicious Software Analysis

Review of 16-bit Assembly programming





Qasem Abu Al-Haija

Why Assemble for Cybersecurity Experts?

Understanding assembly code is so important in Code interpretation.

Irrespective of the type of high-level language being used, it must first be translated into assembly language before the code gets translated to machine code. This makes assembly language still important despite the evolution of high-level languages.

Understanding assembly code is so important in Control System Resources.

It helps in taking complete control over the system and its resources. By learning assembly language, the programmer can write the code to access registers and retrieve the memory address of pointers and values.

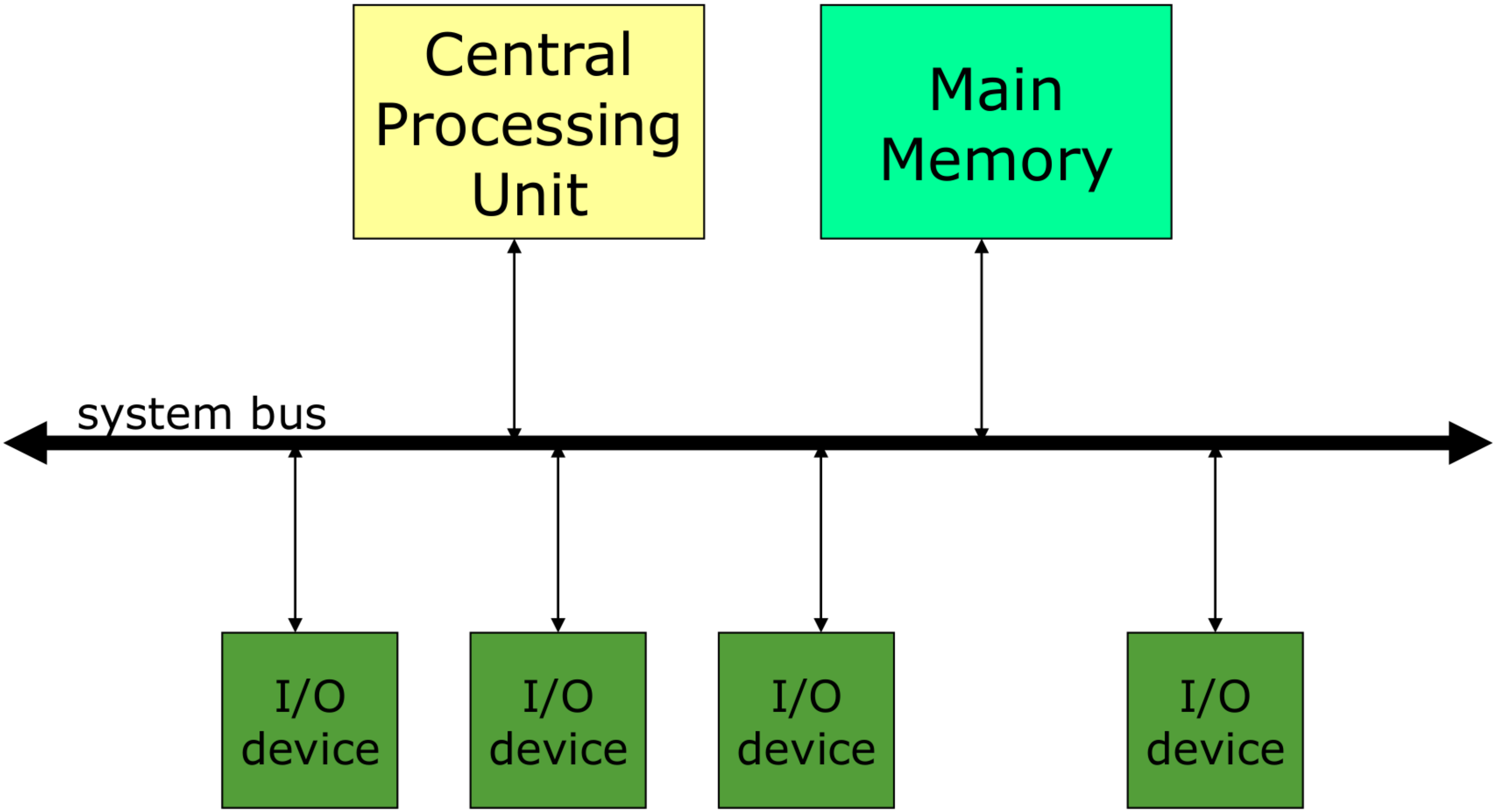
Understanding assembly code is so important in Malware analysis.

Assembly is an essential programming language as cybersecurity experts might use it to interpret malware and understand their modes of attack. Cybersecurity professionals defend against traditional and contemporary malware continuously, so it's essential to understand how malware functions.

Understanding assembly code is so important in malware reverse engineering.

Knowledge of assembly language programming is a must in malware reverse engineering because malware authors do not normally publish their source code, and for that reason, reverse engineering is done

Review of System Diagram



History Intel Processors

Early Intel Processors

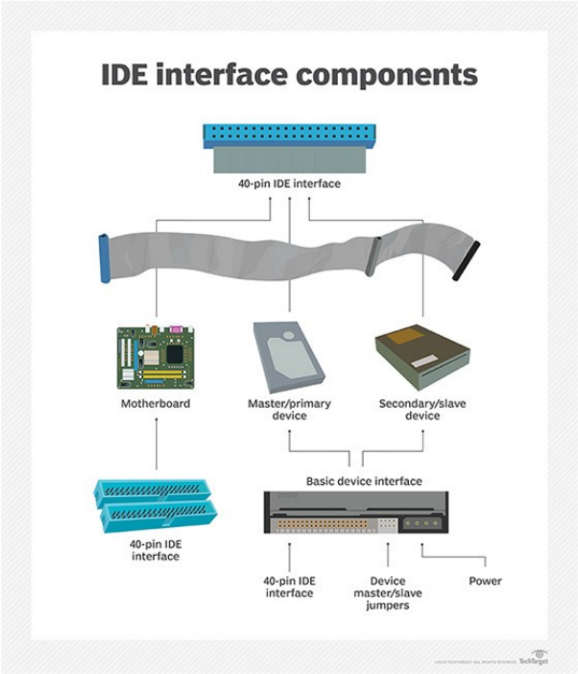
- 1971: 4004 (first 4-bit processor)
- 1972: 8008 (first 8-bit processor)
- 1974: 8080 (widely used by CP/M)
- 1978: 8086/8088 (first 16-bit processor)
- 1982: 80286: (introduced protected mode)
- 1985: 80386: (first 32-bit processor)
- 1989: 80486: (integrated floating-point)
- …..

Early Intel microprocessors

- Intel 8080 (1972)
  - 64K addressable RAM
  - 8-bit registers
  - CP/M operating system
  - 5,6,8,10 MHz
  - 29K transistors
- Intel 8086/8088 (1978) ← first real computer
  - IBM-PC used 8088
  - 1 MB addressable RAM
  - 16-bit registers
  - 16-bit data bus (8-bit for 8088)
  - separate floating-point unit (8087)
  - used in low-cost microcontrollers now

The IBM-AT

- Intel 80286 (1982)
  - 16 MB addressable RAM
  - Protected memory
  - several times faster than 8086
  - introduced IDE bus architecture
  - 80287 floating point unit
  - Up to 20MHz
  - 134K transistors



Intel IA-32 Family

- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

Intel P6 Family

- Pentium Pro (1995)
  - advanced optimization techniques in microcode
  - More pipeline stages
  - On-board L2 cache
- Pentium II (1997)
  - MMX (multimedia) instruction set
  - Up to 450MHz
- Pentium III (1999)
  - SIMD (streaming extensions) instructions (SSE)
  - Up to 1+GHz
- Pentium 4 (2000)
  - NetBurst micro-architecture, tuned for multimedia
  - 3.8+GHz
- Pentium D (2005, Dual core)

In this review class

We will focus on reviewing the Intel8086

Microprocessor

For efficient use of any µP

- Understand the main feature.
- Understand the internal HW architecture.
- Understand the instruction set architecture (ISA).

In this review, we are interested in Intel 8086 µP

Introduction to I8086 μp

- In 1972, Intel launched the **8008**, the first 8-bit microprocessor.
- It needed several additional ICs to produce a functional computer.

- In 1972, Intel launched **8080,** employing the new 40-pin DIP.
- Originally developed for calculator ICs to enable a separate address bus

- In 1977, Intel launched **8085** with a single +5 V power supply chip.
- Other well-known 8-bit μp: Motorola 6800, Zilog Z80, and others.

- In 1978, Intel launched **8086** (iAPX 86) as the first 16-bit μp chip.
- It gave rise to x86 architecture family: Intel's most successful line of μp.

Intel 8086 μp

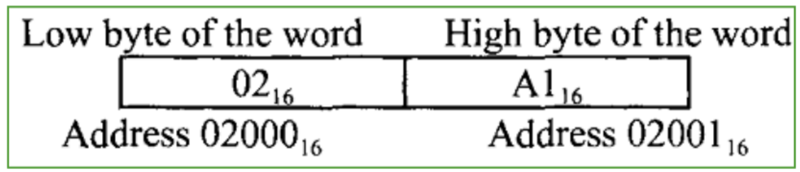
I8086 µp Features

- 8086 is the first 16-bit µp released by Intel (1978).
  - 40-pin DIPs, 16-bit data bus (D0-D15), and 20-bit address bus (A0-A19).
  - Higher execution speed - larger memory size (of previous µps).
  - Run at 2.5 MIPS ➔ T_{exe} of one instruction = 400 ns (=1/MIPS=1/(2.5x10^6)).
  - Contains a small pre-fetch 6-byte instruction queue➔ **Pipelining**.
  - 8086 µp is an example of a complex instruction set computer (CISC).
  - 8086 µp is an example of a von Neumann Architecture (VNA) computer.
  - 8086 clock input signal is generated by the 8284-clock generator chip.
  - Instruction execution times vary between 2 and 30 clock cycles.
  - Four versions: 8086 (5 MHz),8086- 1 (10 MHz),8086-2(8 MHz) & 8086-4 (4 MHz).
  - 8086 has two modes of operation (Min mode and Max mode).

I8086 μp Features

• 8086 Memory Addressing.

- 8086 has 20 address pins ➔ 2^{20} bytes=1 MB of memory uniquely addressable.
- 8086 memory is Byte addressable: 00000_{16}; 00001_{16}; .... $FFFFF_{16}$.
- 8086 has 16 data pins➔ can read 8-bit or 16-bit word (2- con. byte) from memory

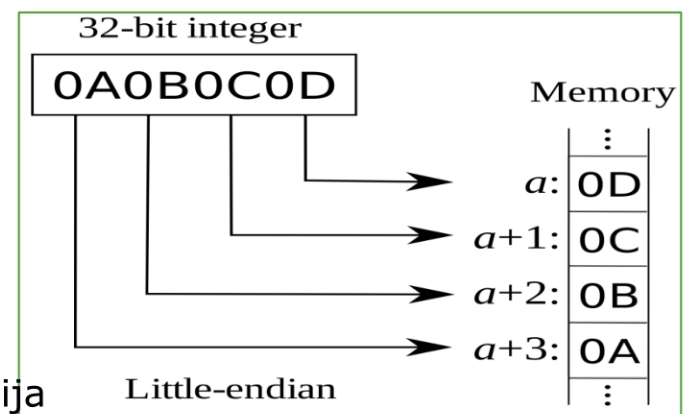


• 8086 Registers Naming.

- 8086 register names followed by the letters X, H, or L (to specify 16 or 8-bit).
- Examples: MOV AX, [START] MOV AL, [START].

• 8086 Endianness

– 8086 uses Little-endian byte order to compute the physical address.

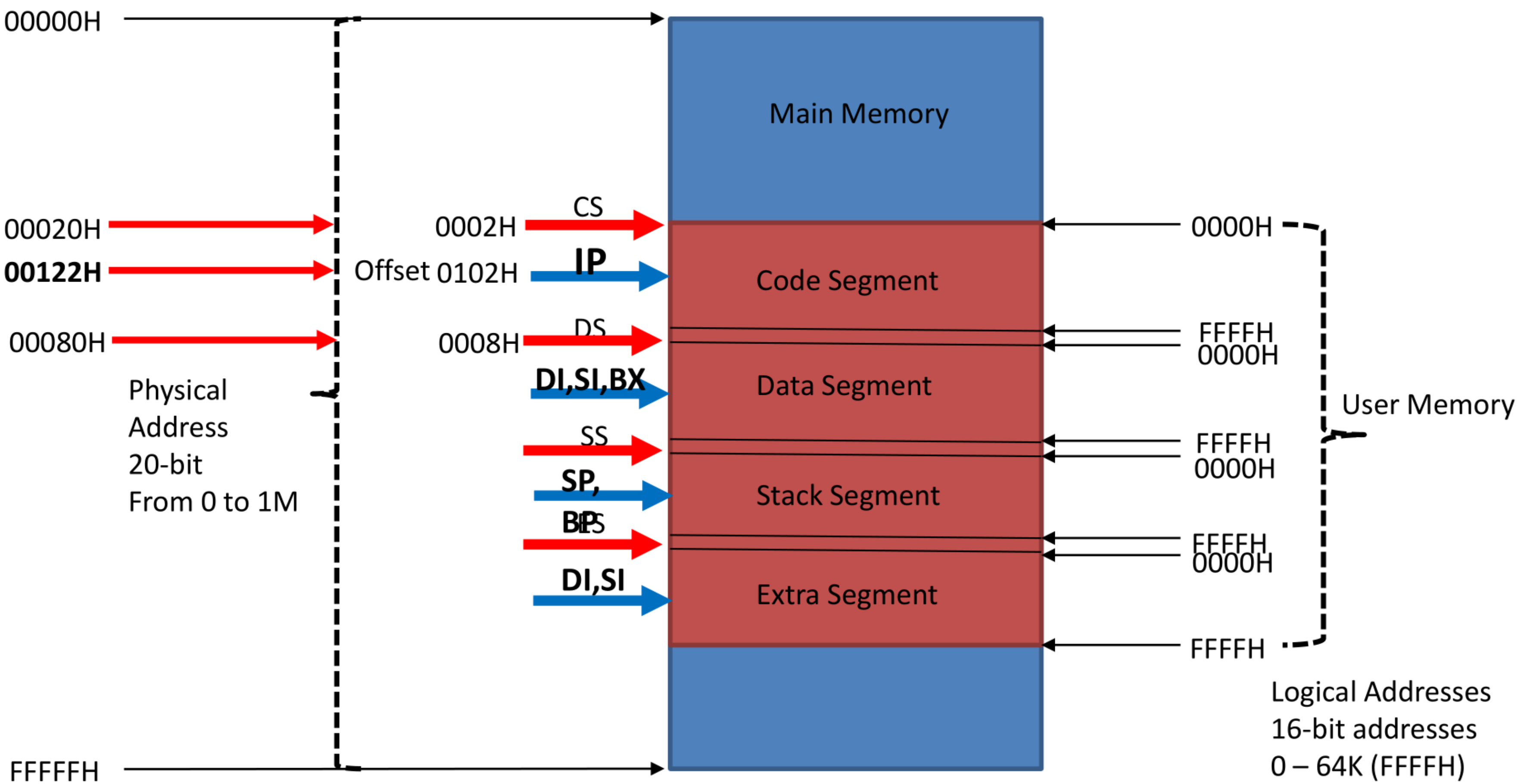


8086 Main Memory

- 8086 uses a segmented memory.
  - **+Ve:** Manipulates 16-bit components only and effectively used in time-shared systems.
  - **Thus:** 8086 Memory can be divided into 16 segments (**1 MB = 16 x 64 KB**).
  - **8086** segments may contain: codes or data or stack or extra.
  - Segments can be: Contiguous, Partially overlapped, Fully overlapped, or disjointed
  - **Therefore,** 8086 employs 16-bit registers to address segments such as: DS, CS.
  - **By this,** we will have two kind of addresses: Physical address and Logical address.

- Physical address of μP (20 bit) ➔ Not used to access Memory.
  - Instead: Logical Address with two 16-bit components [Segment: Offset] is used.
  - 8086 includes on-chip HW to translate between physical & logical addresses.
  - Shifting segment register 4 times to left then adding it to offset register.

8086 Main Memory



Example: Physical Address Calculation

- Given: CS = 0020H and IP = 0121H
- What is the Physical Address?

➔Add a zero to the right of the segment register, then Add it to IP

CS = 00200H

IP = 0121 H

= 00321 H

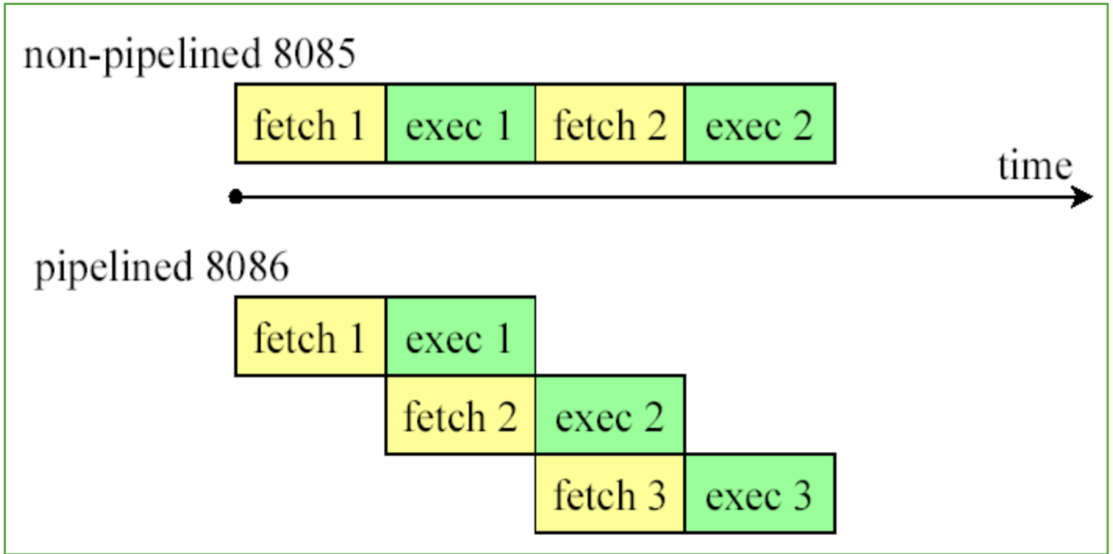
8086 Hardware Architecture

- Enhanced internal architecture via Pipelining.
  - Pipelining is to allow the CPU to fetch and execute at the same time.
  - This can be accomplished by having several units works simultaneously

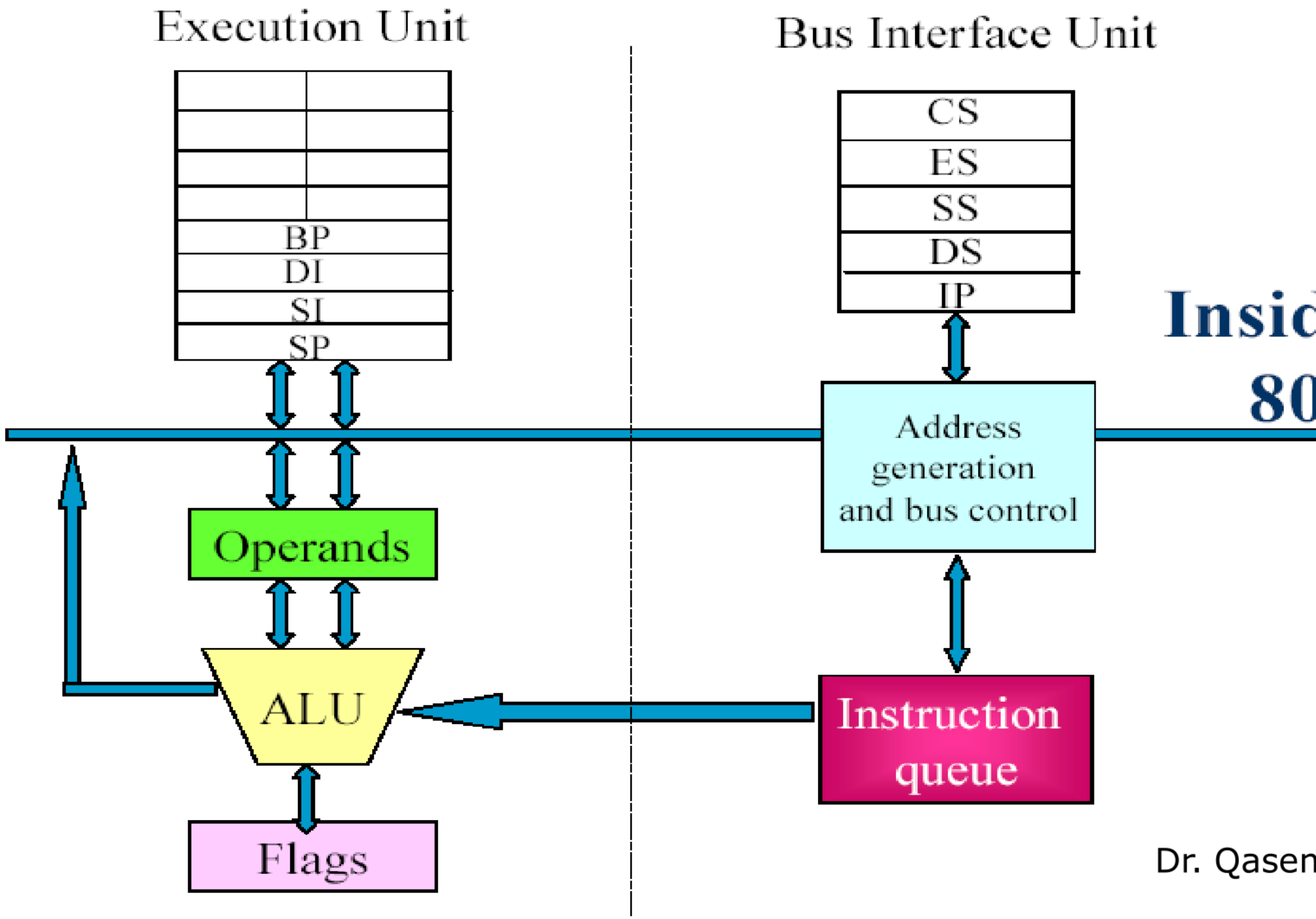
- Thus, internal structure of 8086 is split into:
  - Execution Unit (EU)

    Executes instructions already fetched
  - Bus Interface Unit (BIU)

    Accesses memory and peripherals



8086 Hardware Architecture



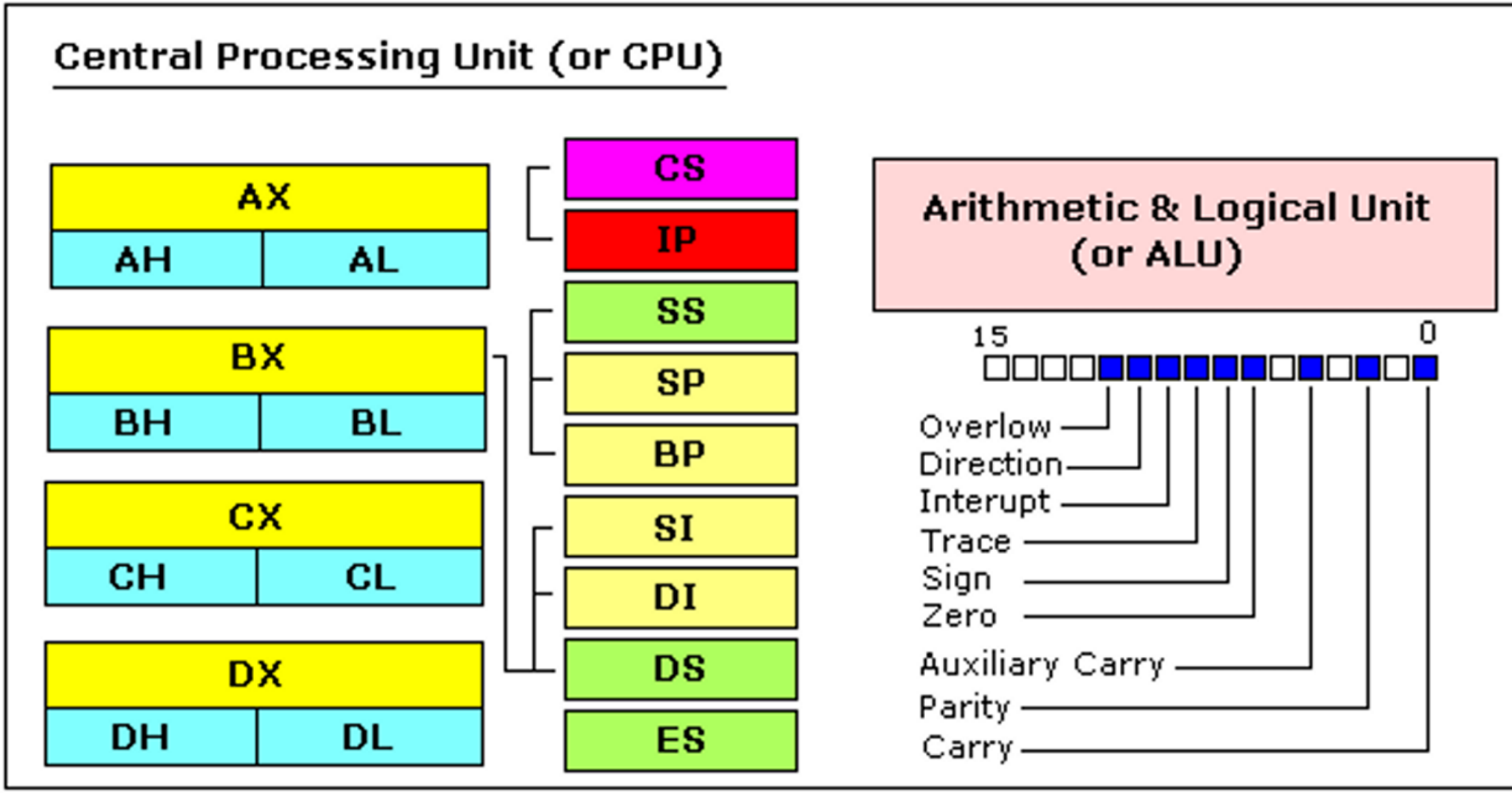
Inside the 8086

Dr. Qasem Abu Al-Haija

Execution Unit (EU)
EU executes instructions that are already fetched by BIU.
BIU and EU function separately.

Bus Interface Unit (BIU)
Reads (fetch) instructions, reads operands and writes results.

8086 Registers



8086 Registers

- BIU Registers
  - IP: 16-bit Instruction Pointer (offset points to the current instruction).
  - CS: 16-bit Code Segment Register (points to current code segment).
  - DS: 16-bit Data Segment Register (points to current data segment).
  - SS: 16-bit Stack Segment Register (points to current stack segment).
  - ES: 16-bit Extra Segment Register (points to current extra segment).
- General Purpose Registers (GPRs)
  - AX (Accumulator): used IN/OUT instructions, MUL, and DIV instructions.
  - BX (Base): used for memory addressing and operands.
  - CX (Counter): used mainly by SHIFT, ROTATE, and LOOP instructions.
  - DX (Data): used mainly to hold a High 16-bit result after 16x16-bit MUL or High 16-bit dividend before a 32÷16 DIV (LOW 16-bit in AX)
- Pointer Registers (SP/BP).
  - Stack Pointer & Base Pointer are used to access data in the stack segment.
  - SP is to be used as an offset access STACK memory with SS as segment register.
  - SP is auto-incremented or decremented due to execution stack instructions.
  - BP is used by the user in the based addressing mode (later).

8086 Registers

- Index Registers (SI and DI).
  - Source index and destination index are used with string Instructions along with DS & ES, respectively.
- Flags Register (FL)
  - FL bits are set or reset by EU to reflect the results of ALU.
  - DF: Controlling string operations.
  - IF: Controlling Maskable interrupts.
  - TF : Provides Single-Step debugging.

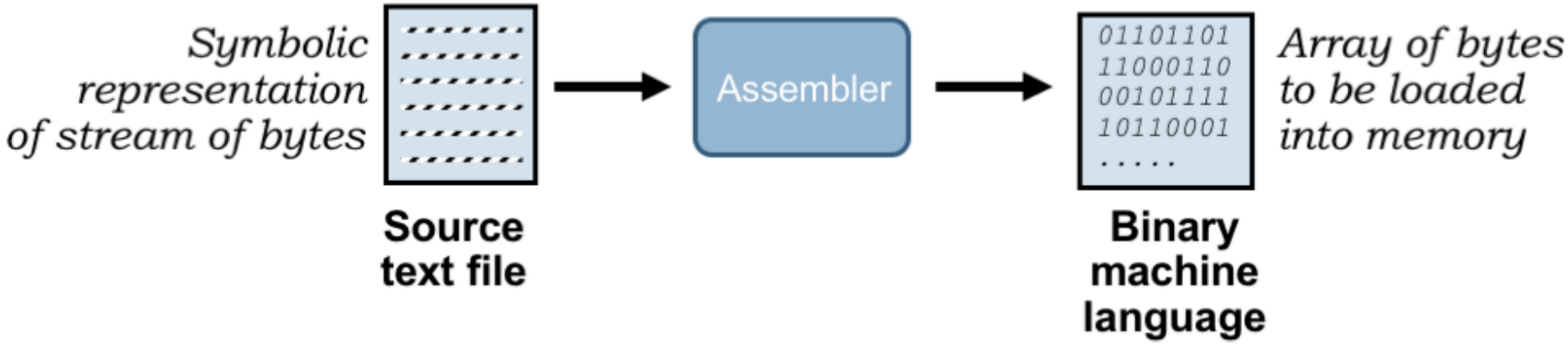
U	U	U	U	OF	DF	IF	TF	SF	ZF	U	AF	U	PF	U	CF

1.	CF	CARRY FLAG	Conditional Flags (Compatible with 8085, except OF)
2.	PF	PARITY FLAG	
3.	AF	AUXILIARY CARRY	
4.	ZF	ZERO FLAG	
5.	SF	SIGN FLAG	
6.	OF	OVERFLOW FLAG	
7.	TF	TRAP FLAG	Control Flags
8.	IF	INTERRUPT FLAG	Those can be set or cleared By Programmer
9.	DF	DIRECTION FLAG	

How can we practice using and programming 8086?

- We need to use a virtual machine (emulator system for the 8086).
- EMU8086.
- 8086 is programmed using Assembly language using its own predefined ISA

Assembly Language



- Abstracts bit-level representation of instructions and addresses
- Main elements:
  - Values
  - Symbols
  - Labels (symbols for addresses)
  - Macros

Program template in EMU8086

```
DATA SEGMENT
; DEFINE YOUR DATA HERE
 ENDS

STACK SEGMENT
   DW   128  DUP(0)
ENDS
```

; keep it as is...stack contains 128 words of memory

```
CODE SEGMENT
START:
   MOV AX, DATA
   MOV DS, AX
   MOV ES, AX
```

; always include these three lines... get the address of data segment at runtime

```
; WRITE YOUR CODE HERE

   MOV AX, 4C00H
   INT 21H
```

;Two lines: exit to the operating system and terminate the program

```
ENDS
END START
```

Directives and Instructions

- Assembly language statements are either directives or instructions

- Instructions are executable statements. They are translated by the assembler into machine instructions. Ex:
  - CALL MySub ;transfer of control
  - MOV AX,5 ;data transfer

- Directives tell the assembler how to generate machine code and allocate storage. Ex:

  COUNT DB 50 ;creates 1 byte of storage initialized to 50

8086 Assembly Directives

8086 Assembler Directives- Variable/Constant Definition

- **DB, DW, DD, DQ, DT, directives.**

  Reserve Byte, Word, Double Word, Quad Word, Ten Bytes in memory for storing variables.

- **EQU or =**

  The assembler does not allocate storage to a constant.

- **DUP directive**

  Initialize Several Locations to an Initial Value.

• BYTE	8-BIT
• WORD	16-BIT
• DWORD	32-BIT
• FWORD	48-BIT
• QWORD	64-BIT

Example

- NUMS DB 20
- LIST DB 1, 2, 8, 9, 5

8086 Assembler Directives- Variable/Constant Definition

Example	Comments
DATA1 DB 20H	Reserve one byte to store DATA1 initialized to 20H.
ARRAY1 DB 10H,20H,30H	Reserve 3 bytes to store ARRAY1 initialized with 10H, 20H, 30H
CITY DB “DAMMAM”	Reserve a list named CITYT initialized with Chars’ ASCII codes.
DATA2 DW 1020H	Reserve one word to store DATA2 initialized to 1020H.
NUMBER EQU 50H	Assign the value 50H to NUMBER
NAME EQU “QASEM”	Assign the string “QASEM” to NAME
START DW 4 DUP (0)	Reserves 4 words starting at offset START in DS initialized to 0.
BEGIN DB 100 DUP (?)	Reserves 100 bytes of uninitialized data to offset BEGIN in DS.
X DW 2A05H Y DW 052AH PRODUCT EQU X * Y	Using Expressions
SUNDAY EQU 1 MONDAY EQU SUNDAY + 1	Using Expressions

Example (1): Variable Definition

- VAL1 DB 10
- VAL2 DB 0AH
- ARRAY1 DB 3, 5, 1, 0
- CHAR1 DB "A" ; SINGLE QOUTEA ARE OK TOO
- VAL3 DB ?
- STR1 DB "Hello World"
- Msg1 DB "welcome", 0Ah, 0DH
- VAL4 DW 90A1H, 0FH
- BIG DD 11223344H
- LIST DB 2, 0, 1
  DB 10
  DB 1

See the representation of data in memory – next slide

Representing the data in memory

ADDRESS	CONTENTS	
0	10	VAL1
1	0A	VAL2
2	3	ARRAY1
3	5	
4	1	
5	0	
6	"A"	CHAR1
7	-	VAL3
8	"H"	STR1
9	"E"	
A	"L"	
B	"L"	
C	"O"	
D	SP	
E	"W"	
F	"O"	
10	"R"	
11	"L"	
12	"D"	
13	"W"	MSG1
14	"E"	
15	"L"	
16	"C"	
17	"O"	
18	"M"	
19	"E"	
1A	0A	
1B	0D	
1C	A1	VAL4
1D	90	
1E	0F	
1F	00	
20	44	BIG
21	33	
22	22	
23	11	
24	2	LIST
25	0	
26	1	
27	10	
28	1	
29		
2A		
2B		
2C		
2D		
2E		
2F		

Example (2): Using DUP Operator (For Arrays)

- ARR1 BYTE 20 DUP(0) ; 20 bytes, all equal to zero
- ARR2 DB 20 DUP (0); SAME AS ABOVE
- LIST1 DB 20 DUP(?) ; 20 bytes, uninitialized

Example (3): Working with constants

```
COUNT = 5
mov al, COUNT ; AL = 5
COUNT = 10
mov al, COUNT ; AL = 10
COUNT = 100
mov al, COUNT ; AL = 100
```

8086 Assembler Directives-Related to Code Location.

- **ORG (ORIGIN) Directive.**

  Tells the assembler where to load instructions and data into memory.

  Initialize CS and IP with initial address (logical) as a starting address.

  If its not mentioned at the start of segment➔ Offset is initialized to 0000H.

- **Example: ORG 0100H**

  The first instruction is stored from at offset 0100H within the code segment.

- **OFFSET and SEG Directives.**

  Used to determine the Offset and Segment addresses of a given data item.

- **Example: MOV BX, OFFSET TABLE / MOV AX, SEG ARRAY1**

- **EVEN Directive.**

  Used to declare a data item to start at even memory address.

- **Example: EVEN / ARRAY2 DW 20 DUP (0)**

Using pointers to access memory

- You can use any of the pointers in the data segment to access your data and arrays such as BX, SI, DI.

- Assume we have the following array:

  Nums db 2, 1, 5, 0, 1 ; array contains 5 elements

  1. Use a pointer BX to point at the first address in the array:

     Mov BX, offset nums or LEA BX, nums

  2. Start a loop and access the contents of the array using [BX]:

     Mov AL, [BX]

  3. move the pointer to the next location using:

     INC BX

  4. repeat the loop until you finish all the 5 elements

Example: Accessing the contents of an Array

```
.DATA
Nums db 2, 1, 5, 0, 1                ; array contains 5 elements
.CODE
            MOV CX, 5                ; counter for the loop
            MOV BX, OFFSET NUMS      ; let BX points to first location in Nums
LOOP1 :     MOV AL, [BX]             ; access location in Nums pointed at by BX
            INC BX                   ; let BX point to the next location in Nums
            DEC CX                   ;  subtract 1 from the counter
            JNZ LOOP1                ; repeat the loop until CX=0
```

Another Example: Access arrays without using offset

In this example, we will access the memory using another method, Just like Higher-level Languages, and using any pointer (BX, DI, SI)

```
.DATA
Nums db 2, 1, 5, 0, 1                ; array contains 5 elements
.CODE
            MOV CX, 5                ; counter for the loop
            MOV BX, 0                ; initialize BX to Zero... It will be the index for the array
LOOP1 :     MOV AL, Nums[BX]         ; access location in nums pointed at by BX
            INC BX                   ; let BX point to the next location in Nums
            DEC CX                   ;  subtract 1 from the counter
            JNZ LOOP1                ; repeat the loop until CX=0
```

8086 Assembler Directives- For Segment Declaration.

- **SEGMENT and ENDS directives.**

  Indicate the Start & End of a logical segment (Segment name ≤ 31 characters).

```
Segnam SEGMENT
    ...
    ...
    ...
    ...
    ...
    ...
Segnam ENDS
```

- Example:

```
START  SEGMENT
    X1  DB     F1H
    X2  DB     50H
    X3  DB     25H
START  ENDS
```

Programmer must then use 8086 instructions to load START into DS, such as:

```
MOV    BX, START
MOV    DS, BX
```

- **ASSUME directive.**

  Links the logical segments with the declared segment names.

- Example 1:

```
CODE      SEGMENT
ASSUME    CS:CODE, DS:CODE, ES:CODE, SS:CODE
⁝
CODE      ENDS
```

- Example 2: ASSUME CS : PROGRAM_1, DS : DATA_1, SS : STACK_1

8086 Assembler Directive- Procedures Declaration.

- **PROC and ENDP directives.**

  Indicates the start and the end of a named procedure (NEAR or FAR).

- **Example1:**

```
SQUARE_ROOT PROC        NEAR
     ⋮                    ⋮
SQUARE_ROOT ENDP
```

  Define a procedure “SQUARE_ROOT”, which is to be called by a program located in the same segment (Near).

- **Example2:**

```
SQUARE_ROOT PROC        FAR
     ⋮                    ⋮
SQUARE_ROOT ENDP
```

  Define a procedure “SQUARE_ROOT”, which is to be called by a program located in another segment (Far).

8086 Assembler Directive- Macros Declaration.

- **MACRO and ENDM directives.**

  Indicates the start and the end of a named MACRO (Can take parameters).

- Example 1 :

```
CALCULATE    MACRO
             MOV AX, [BX]
             ADD AX, [BX+2]
             MOV [SI], AX
             ENDM
```

Can be used any time in the main program, just use its name

Example 2 :

```
CALCULATE    MACRO  OPERAND, RESULT
             MOV BX, OFFSET OPERAND
             MOV AX, [BX]
             ADD AX, [BX+2]
             MOV SI, OFFSET RESULT
             MOV [SI], AX
             ENDM
```

Parameters OPERAND and RESULT can be replaced by OPERAND1, RESULT1, and OPERAND2, RESULT2 while calling the above macro as shown below:

```
..............................
CALCULATE            OPERAND1, RESULT1
.............................
............................
CALCULATE            OPERAND2, RESULT2
.............................
```

8086 Assembler Directives-Other Directives.

- **PTR (Pointer) directive.**

  Used to declare the type of memory operand (prefixed by BYTE or WORD).
- **Examples: INC BYTE PTR [SI] / INC WORD PTR [BX].**
- **NAME directive.**

  Used to assign a name to an assembly language program module.
- **Examples:** NAME “Hi-World”
- **TYPE directive.**

  Return the data type used to define a specific data (Word 2, Double 4, Byte 1).
- **Example: MOV BX, TYPE DATA1.**
- **LENGTH Directive (or $ operator).**

  Used to determine the length of an array in bytes .
- **Example: MOV CX, LENGTH ARRAY**

- **See other directives such as:**

  SHORT, LABEL, GROUP, EXTRN & PUBLIC, GLOBAL & LOCAL

Example: Using $ operator to calculate the size of arrays/lists

- **Example (1)**
  - List db 1, 5, 2, 8, 9, 10, 3, 1
  - List_size = ($ - list) ;;;; list equals 8

- **Example (2)**
  - myString "This is a long string, containing"
  - myString_len = ($ - myString) ;;;; list equals 33

Intel 8086 Assembly Instructions

Simple Instructions

• MOV	(Assignment)
• INC	(Add 1)
• DEC	(Subtract 1)
• ADD	(Add two numbers)
• SUB	(Subtract two numbers)
• CMP	(Compare two numbers)
• JMP	(Go to)
• JNZ	(Go to if results is not zero)
• JZ	(Go to if results is zero)

Instructions Format

- Two Operand Instructions

General Form:

<Instruction> <Target Operand>, <Source Operand>

Examples

```
MOV AX, 5          ; ASSIGN AX THE VALUE 5
MOV DX, AX         ; ASSIGN DX WHATEVER VALUE IN AX
MOV NUMS, 2        ; STORE 2 IN VARIABLE NUMS
ADD CX, 2          ; ADD 2 TO THE VALUE OF CX
CMP AX, 5          ; COMPARE THE VALUE OF AX WITH 5
SUB AX, BX         ; AX = AX - BX
```

One operand instructions

- **General form:**

  <Instruction> <Destination>

  Destination: reg., variable

Examples

```
INC AX          ; AX = AX +1
INC NUMS        ; NUMS = NUMS + 1
DEC BX          ; BX = BX -1
JMP   LABEL1    ; GO TO LABEL1
JNZ   LABEL1    ; DON'T JUMP IF RESULTS IS ZERO
JZ    LABEL1    ; JUMP IF REULTS IS ZERO
```

General Assembly Language Rules

1. Operands must be equal size at all times

   – Mov Op1 (8-bit), Op2 (8-bit) ...... Ok
   – Mov Op1 (16-bit), Op2 (16-bit) ...... OK
   – Mov Op1 (16-bit), Op2 (8-bit) Wrong.... Wrong

General Assembly Language Rules

- An instruction can not refer or use two memory locations. The two operands can not be memory locations. Its ok to use a memory location with a register or a constant
  - Mov num1, num2 .... Wrong
- MOV AX, NUM1 ...... OK
- MOV NUM2, AX .... OK
  - Mov num1, [BX] ..... Wrong
  - Mov num1, 10 .... OK

General Assembly Language Rules

- **The destination of any instruction should not be a constant**
  - **Mov 10, num …… Wrong**
  - **Inc 10 ……….. Wrong**

General Assembly Language Rules

- One of the operands of any instruction should specify the size (8 or 16 bit) of the instruction
  - Mov [BX], 10 …… Wrong
  - Inc [BX]……….. Wrong
  - Inc Byte PTR [BX]…..OK
  - Add Word PTR [BX], 10 …..OK
  - Mov [BX], AL…… OK
- Note: [BX] may refer to an 8-bit or 16-bit location. It does not really specify the size

Data Transfer Instructions

- **Examples of MOV instruction.**
  - **MOV CX, DX** ; Copies 16-bit contents of DX into CX
  - **MOV AX, 2025H** ; Moves immediate data 2025 to AX register
  - **MOV CH, [BX]** ; BX = 0050H, DS = 2000H, Mem Loc (20050) = 08
    ; 8-bit contents of memory location DS+BX will be transferred to CH register, memory location is 20000 + 00050 = (20050)H ➔ CH will contain 08H
  - **MOV START [BP], CX** ; CX = 5009H, BP = 0030H, SS = 3000H, START = 06H
    ;16-bit contents of register CX will be stored in memory location SS+START+BP = 30000 + 00030 + 06 =(30036)H = 09H(CL) and memory location (30037) = 50H (CH).

- **XCHG instruction.**
  - Exchanges the register contents with the contents of memory location.
  - It cannot exchange directly the contents of two memory locations.
  - The source and destination must both be words or must both be bytes.
  - The segment registers cannot be used in this instruction.
- **Examples : XCHG AL, BL / XCHG CX,BX / XCHG AL, [BX].**

Data Transfer Instructions

- **LEA instruction (**Load Effective Address**)**
  - Determines the offset address of a variable or memory location named as the source and puts this offset address in the indicated 16-bit register.
  - The general format of LEA instruction is: LEA register, source.
  - **Examples :**
  - **LEA BX, COST** ; BX= Offset address of COST in data segment where COST is ; the name assigned to a memory location in data segment.
  - **LEA CX, [BX][SI]** ; CX= (BX)+(SI) (content of BX and SI respectively).

- **LDS instruction (**Load register and DS with words from memory**)**
  - Copies a word from memory location specified in the instruction into register and then copies a word from the next memory location into the DS register.
  - LDS is useful for initializing SI and DS registers at the start of a string before using one of the String instructions.
  - **Examples :**
  - **LDS SI,[2000H]** ; Copy the contents of memory word at offset address 2000H in ; data segment to SI register and the contents of memory word ; at offset address 2002H in data segment to DS register.

- **LES, LSS instructions**
  - Similar to LDS instruction except that instead of DS register, ES and SS registers are loaded respectively along with the register specified in the instruction.

Data Transfer Instructions

- **PUSH instruction.**
  - Used to store a word from a register or a memory location into stack.
  - SP is decremented by 2 after execution of PUSH.
  - **Example: PUSH CX, PUSH DS**

- **POP instruction.**
  - Copies the top word from stack into a destination specified in the instruction.
  - The destination can be a GPR, a segment register or a memory location.
  - SP is incremented by 2 after execution of POP to point to the next word in stack.
- **Examples : POP CX / POP DS / POP [SI].**

Example: if BX, DX, and SI are PUSHed:

```
PUSH    BX
PUSH    DX
PUSH    SI
```

Then: they must be POPped using:

```
POP     SI
POP     DX
POP     BX
```

Data Transfer Instructions

Example of PUSH instruction: PUSH [BX] , Assume that:

DS = 2000H, BX = 0200H, SP = 3000H, SS = 4000H, (20200) = 0120H

BEFORE					AFTER				
SP	**3000**	**Memory locations**	**20200**	**20**	**SP**	**2FFE**	**Memory locations**	**2020 0**	**20**
DS	**2000**		**20201**	**01**	**DS**	**2000**		**20201**	**01**
SS	**4000**				**SS**	**4000**			
BX	**0200**	**Memory locations**	**42FFE**	**xx**	**BX**	**0200**	**Memory locations**	**42FFE**	**20**
			42FFF	**xx**				**42FFF**	**01**

Arithmetic Instructions

	Addition
ADD a, b	Add byte or word
ADC a, b	Add byte or word with carry
INC reg/mem	Increment byte or word by one

Destination	Source
Register	Register
Register	Memory
Memory	Register
Register	Immediate
Memory	Immediate
Accumulator	Immediate

(a)

Destination
Reg 16
Reg 8
Memory

(b)

(a) Allowed operands for ADD and ADC
(b) Allowed operands for INC

The AF, CF, OF, PF, SF and ZF flags are affected by the execution of ADD/SUB instruction

	Subtraction
SUB a, b	Subtract byte or word
SBB a, b	Subtract byte or word with borrow
DEC reg/mem	Decrement byte or word by one
NEG reg/mem	Negate byte or word
CMP a, b	Compare byte or word

Destination	Source
Register	Register
Register	Memory
Memory	Register
Accumulator	Immediate
Register	Immediate
Memory	Immediate

(b)

Destination
Reg16
Reg 8
Memory

(c)

Destination
Register
Memory

(d)

(b) Allowed operands for SUB and SBB instructions
(c) Allowed operands for DEC instruction
(d) Allowed operands for NEG instruction

a = “reg” or “mem,” b = “reg” or “mem” or “data.”

Arithmetic Instructions

Multiplication		
MUL reg/mem	Multiply byte or word unsigned	for byte
IMUL reg/mem	Integer multiply byte or word (signed)	[AX] ← [AL] · [mem/reg]
		for word
		[DX][AX] ← [AX] · [mem/reg]

Source = Mem8/Mem16/Reg8/Reg16

Division		
DIV reg/mem	Divide byte or word unsigned	16 ÷ 8 bit; $[AX] \leftarrow \frac{[AX]}{[mem/reg]}$
IDIV reg/mem	Integer divide byte or word (signed)	[AH] ← remainder
		[AL] ← quotient
		32÷16 bit; $[DX{:}AX] \leftarrow \frac{[DX{:}AX]}{[mem/reg]}$
		[DX] ← remainder
		[AX] ← quotient

Source = Mem8/Mem16/Reg8/Reg16

—*NOTE: if you are accessing memory with a single operand operation such as MUL, DIV, INC…, then you will have to specify the type of data (byte or word) ==>Two assembler directives are used for this purpose:*

BYTE PTR** & **WORD PTR

Arithmetic Instructions

- **Examples :**
  - **ADD BL, 80H** ; Add immediate data 80H to BL
  - **ADD CX, 12B0H** ; Add immediate data 12B0H to CX
  - **ADD AX, CX** ; Add content of AX and CX and store result in AX
  - **ADD AL, [BX]** ; Add AL to the byte from memory at [BX] and store result in AL.
  - **ADD CX, [SI]** ; Add CX and the word from memory at [SI] and store result in CX.
  - **ADD [BX], DL** ; Add DL with the byte from Mem at [BX] & store result in Mem at [BX].
  - **SUB AL, BL** ; Subtract BL from AL and store result in AL
  - **SUB CX, BX** ; Subtract BX from CX and store result in CX
  - **SUB BX, [DI]** ; Subtract the word in memory at [DI] from BX and store result in BX
  - **SUB [BP], DL** ; Subtract DL from the byte in Mem at [BP] & store result in Mem at [BP].
  - **INC CL** ; Increment content of CL by 1
  - **INC AX** ; Increment content of AX by 1
  - **INC BYTE PTR [BX]** ; Increment byte in memory at [BX] by 1
  - **INC WORD PTR [SI]** ; Increment word in memory at [SI] by 1

Arithmetic Instructions

- **Examples :**

– MUL CH	; Multiply AL and CH and store result in AX
– MUL BX	; Multiply AX and BX and store result in DX-AX
– MUL BYTE PTR [BX]	; Multiply AL with byte in memory at [BX] & store result in AX
– IMUL BL	; Multiply AL with BL and store result in AX
– IMUL AX	; Multiply AX and AX and store result in DX-AX
– IMUL BYTE PTR [BX]	; Multiply AL with byte from memory at [BX] & store result in AX
– IMUL WORD PTR [SI]	; Multiply AX with word from memory at [SI] & store result in DX-AX
– DIV DL	; Divide word in AX by byte in DL.
	; Quotient is stored in AL and remainder is stored in AH
– DIV CX	; Divide double word (32 bits) in DX-AX by word in CX.
	; Quotient is stored in AX and remainder is stored in DX
– DIV BYTE PTR [BX]	; Divide word in AX by byte from memory at [BX].
	; Quotient is stored in AL and remainder is stored in AH.

Arithmetic Instructions

EX: if (AX) = 0005_{16} & (CL) = 02_{16} → DIV CL → (AH) = 01_{16} (Rem) & (AL) = 02_{16} (Quot).

EX: If (CX) = 2 and (DX AX) = -5_{10} = $FFFFFFFB_{16}$ → IDIV, after this IDIV, DX and AX will contain:

DX	AX
FFFF	FFFE
16-bit remainder = -1_{10}	16-bit quotient = -2_{10}

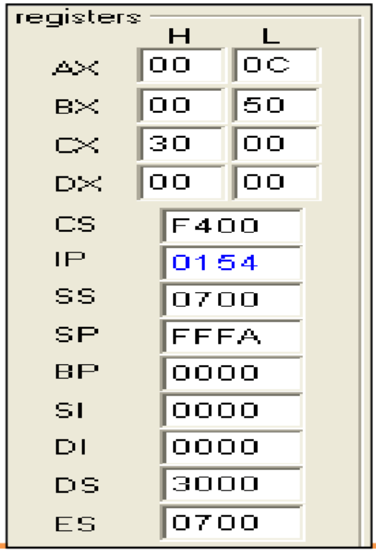
EX: If (AL)=20_{16} & (BL) = 02_{16} → MUL BL. → AX will contain 0040_{16}

EX: If (CL) = FDH = -3_{10} & (AL) = FEH = -2_{10} → IMUL CL → AX contains 0006H.

EX : If (AL) = FF_{16}= -1_{10} and (DH) = 02_{10} → IMUL DH → AX = $FFFE_{16}$ (-2_{10}).

Example:

```
MOV    BX, 0050H
MOV     CX, 3000H
MOV    DS, CX
MOV    [BX],0006H
MOV    AX, 0002H
MUL    WORD PTR [BX]
```



Remember, signed numbers:

if 8 bit (−128 to 127)

if 16 bit (−32768 to 32767)

Arithmetic Instructions

EX: ADC AX, [BX] ; 0020+0100+1 = 0121

Before				After			
AX	0020	Memory locations		AX	0121	Memory locations	
DS	2020	20500	00	DS	2020	20500	00
BX	0300	20501	01	BX	0200	20501	01
CF	1			CF	0	PF = 0, AF = 0, ZF = 0, SF = 0, OF = 0	

EX: SBBCH,DL ; 03 – 02 - 1 = 0

Before		After		
CH	03	CH	0	
DL	02	DL	02	
CF	1	CF	1	PF = 1, AF = 1, ZF = 1, SF = 0, OF = 0

EX: CMP DH, BL.

Before Execution:

Assume:
(DH) = 40H
(BL) = 30H

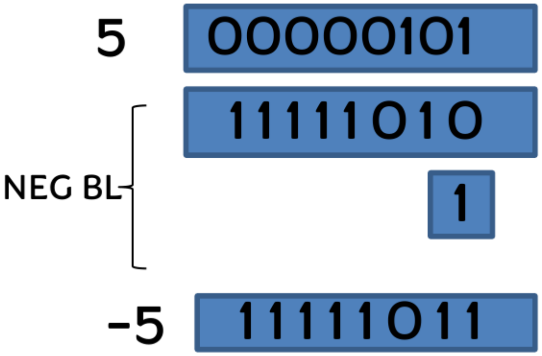
After Execution:

Result 10H is not provided
Flags are: CF= 0, PF=0,
AF=0, ZF=0, SF=0, & OF= 0

Arithmetic Instructions

NEG (2'S COMPLEMENT)

- NEG DESTINATION
- DESTINATION REG., MEMORY (8-BIT OR 16-BIT)
- EXAMPLE:
- MOV BL, 5
- NEG BL



Flags affected: ZF, OF, SF, CF

Arithmetic Instructions

- **CBW: Convert byte to word (No Operand)**

  if high bit of AL = 1 then: AH = 255 (0FFh)
  Else, AH = 0

  Example:

```
MOV AX, 0   ; AH = 0, AL = 0
MOV AL, -5  ; AX = 000FBh (251)
CBW         ; AX = 0FFFBh (-5)
RET
```

C	Z	S	O	P	A
unchanged					

- **CWD: Convert word to double word (No Operand)**

  if high bit of AX = 1 then: DX = 65535 (0FFFFh)
  Else, DX = 0

  Example:

```
MOV DX, 0   ; DX = 0
MOV AX, 0   ; AX = 0
MOV AX, -5  ; DX AX = 00000h:0FFFBh
CWD         ; DX AX = 0FFFFh:0FFFBh
RET
```

C	Z	S	O	P	A
unchanged					

Summary of Arithmetic Instructions

	Flag affected				
Instruction	*Z-flag*	*C-flag*	*S-flag*	*O-flag*	*A-flag*
ADD	Yes	Yes	Yes	Yes	Yes
ADC	Yes	Yes	Yes	Yes	Yes
SUB	Yes	Yes	Yes	Yes	Yes
SBB	Yes	Yes	Yes	Yes	Yes
INC	Yes	No	Yes	Yes	Yes
DEC	Yes	No	Yes	Yes	Yes
NEG	Yes	Yes	Yes	Yes	Yes
CMP	Yes	Yes	Yes	Yes	Yes
MUL	No	Yes	No	Yes	No
IMUL	No	Yes	No	Yes	No
DIV	No	No	No	No	No
IDIV	No	No	No	No	No
CBW	No	No	No	No	No
CWD	No	No	No	No	No

Example: 8086 Assembly Programming Using MASM

Write a program to add two 8-bit data (F0H and 50H) in 8086 and store results in memory.

```
DATA     SEGMENT                ; Beginning of data segment
OPER1    DB      F0H            ; First operand
OPER2    DB      50H            ; Second operand
RESULT   DB      01 DUP (?)     ; A byte of memory is reserved for result
CARRY    DB      01 DUP (?)     ; A byte is reserved for storing carry
DATA     ENDS                   ; End of data segment

CODE    SEGMENT                 ; Beginning of code dement
START:   MOV AX, DATA           ; Initialize AX with the segment address of DS
         MOV DS, AX             ; Move AX content to DS
         MOV BX, OFFSET OPER1   ; Move the offset address of OPER1 to BX
         MOV AL, [BX]           ; Move first operand to AL
         ADD AL, [BX+1]         ; Add second operand to AL
         MOV SI, OFFSET RESULT  ; Store offset address of RESULT in SI
```

Example: 8086 Assembly Programming Using MASM

```
        MOV [SI], AL        ; Store content of AL in the location RESULT
        INC SI              ; Increment SI to point location of carry
        JC CAR              ; If carry =1, go to the place CAR
        MOV [SI], 00H       ; Store 00H in the location CARRY
        JMP LOC1            ; go to the place LOC1
CAR:    MOV [SI], 01H       ; Store 01H in the location CARRY

LOC1:   MOV AH, 4CH
        INT 21H             ; Return to DOS prompt
CODE    ENDS                ; End of code segment

END START                   ; Program ends
```

logical Instructions

- **AND : Turns off specific bits (used with masking) .**
- **TEST : Same as AND, does not change destination .**
- **OR : Turns on specific bits .**
- **NOT : Complement all the bits .**
- **XOR : Complement specific bits .**
- **SHR : Shift Right**
- **SAR : Shift Arithmetic Right**
- **SHL : Shift Left**
- **SAL : Shift Arithmetic Left**
- **ROL : Rotate Left**
- **ROR : Rotate Right**
- **RCL : Rotate Carry Left**
- **RCR : Rotate Carry Right**

logical Instructions

Logicals	
NOT mem/reg	NOT byte or word → *One's complement*
AND a, b	AND byte or word
OR a, b	OR byte or word
XOR a, b	Exclusive OR byte or word
TEST a, b	Test byte or word

Shifts	
SHL/SAL mem/reg, CNT	Shift logical/arithmetic left byte or word
SHR/SAR mem/reg, CNT	Shift logical/arithmetic right byte or word

Rotates	
ROL mem/reg, CNT	Rotate left byte or word
ROR mem/reg, CNT	Rotate right byte or word

a = "reg" or "mem," b = "reg" or "mem" or "data," CNT = number of times to be shifted.
If CNT > 1, then CNT is contained in CL. Zero or negative shifts and rotates are illegal.
If CNT = 1 then CNT is immediate data. Up to 255 shifts are allowed.

logical Instructions

EX: TEST CL, 05H

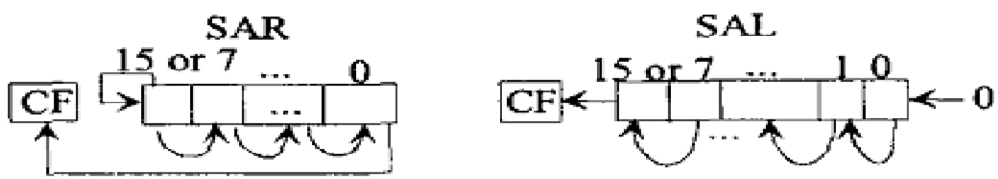
Logically ANDs (CL) with 00000101. Does not store the result in CL, All flags are affected.

EX: Let AL=0111 1111 =7FH,

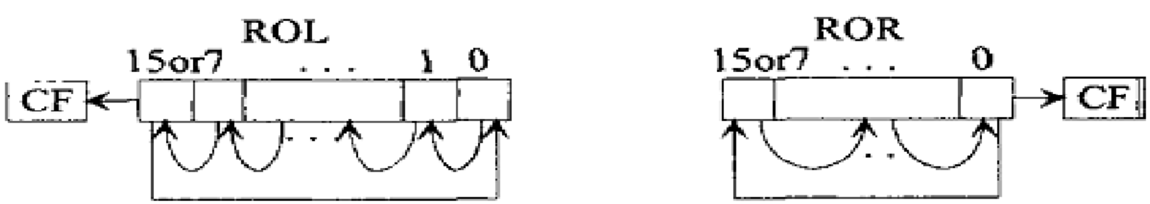
TEST AL, 80H; AL=7FH (unchanged), ZF=1 since (AL) AND (80H)=00H; SF=0; PF=1

EX: MOV CL,2 ; Shift count 2 is moved into CL

SHR DX,CL; Logically shifts (DX) twice to right



8086 SAR and SAL instructions



8086 ROR and ROL instructions

Unconditional Transfer Instructions

- **Used to transfer control to: Intra-segment or Inter-segment.**

CALL reg/mem/disp 16	Call subroutine
RET or RET disp 16	Return from subroutine
JMP disp8/disp 16 /reg16/mem16	Unconditional jump

- **CALL Instruction.**

—Intra-segment CALL: IP changes, CS is fixed, EX: **CALL NEAR PROC**.
—Inter-segment CALL: Both IP & CS are changed, **EX: CALL FAR PROC**.

- **RET instruction.**

—Placed at the end of the subroutine to transfer control back to the main program.

- **JMP Instruction.**

—Intra-segment JMP: IP changes, CS is fixed, EX: **JMP START.**
—Inter-segment JMP: Both IP & CS are changed, **EX: JMP FAR BEGIN.**

Unconditional Transfer Instructions

```
CODE        SEGMENT
            ASSUME      CS:CODE, DS:DATA, SS:STACK
             ------------
             ------------
             CALL   MULTI
             ------------
             ------------
             HLT
MULTI        PROC        NEAR
             ------
             ------
             RET
MULTI        ENDP
CODE         ENDS
```

Example of: Intra-segment CALL

```
ORG 100h
.CODE
MOV AX, 2
MOV BX, 2
JMP LABEL_SUB
ADD AX, BX    ;this instruction will never execute
LABEL_SUB:
              SUB AX, BX
RET
```

Example of: JMP destination_label

Unconditional Transfer Instructions

Example of: Inter-segment CALL

```
CODE        SEGMENT
            ASSUME      CS:CODE, DS:DATA, SS:STACK
            ------------
            ------------
            CALL   MULTI
            ------------
            ------------
            HLT
CODE        ENDS
SUBR        SEGMENT
MULTI       PROC        FAR
            ASSUME      CS:SUBR
            ------
            ------
            RET
MULTI       ENDP
SUBR        ENDS
```

Conditional Branch Instructions

Jumps for Unsigned Numbers

Instructions	Meaning	Condition
JA JNA	Jump if above >	CF = 0 and ZF = 0
JAE JNAE	Jump if above or equal ≥	CF = 0
JB JNB	Jump if below <	CF = 1
JBE JNBE	Jump if below or equal ≤	CF = 1 or ZF = 1
JC JNC	Jump in carry	CF = 1

Conditional Branch Instructions

Example:

Instructions	UNSIGNED FLAGS	SATISFIED JUMP
MOV AL, 10	-	
MOV BL, 5	-	
CMP AL, BL	ZF=0. CF=0	JA, JAE, JNB, JNC, JNZ
CMP BL, AL	ZF = 0. CF = 1	JB, JC, JNA

Conditional Branch Instructions

Jumps for Signed Numbers

Instructions	Meaning	Condition
JG	Jump if greater	ZF = 0 and SF = OF
JGE	Jump if greater or equal	SF=OF
JL	Jump if less	SF<>OF
JLE	Jump if less or equal	ZF=1 OR SF<>OF
JS	Jump on signe flag	SF=1
JNS	Jump on not signe flag	SF=0
JO	Jump on over flow	OF=1
JNO	Jump on not over flow	OF=0

8086 Instruction Set

Group 5: Conditional Branch Instructions

Example:

```
        MOV   AX, 1000H
        MOV   DS, AX                 ;Initialize DS
        MOV   BX, 2000H
        MOV   CX, 3000H
AGAIN:  MOV   WORD PTR[BX], 0000H
        INC   BX
        INC   BX
        CMP   CX, BX
        JGE   AGAIN
```

JGE treats CMP operands as twos complement numbers.

Example:

```
ax = 2;
if ( ax != bx )
{
ax = ax + 1 ;
}
bx = bx + 1 ;
```

→

```
mov ax, 2 ; ax = 2
sub ax, bx ; ax = 2 - bx
jz nextl ; jump if (ax-bx) == 0
      inc ax ; ax = ax + 1
nextl:
      inc bx
```

Iteration Control Instructions

All these instructions have relative addressing modes.

8086 Iteration Control Instructions

LOOP disp8	Decrement CX by 1 without affecting flags and branch to label if CX ≠ 0; otherwise, go to the next instruction.
LOOPE/LOOPZ disp8	Decrement CX by 1 without affecting flags and branch to label if CX ≠ 0 and ZF = 1; otherwise (CX=0 or ZF=0), go to the next instruction.
LOOPNE/LOOPNZ disp8	Decrement CX by 1 without affecting flags and branch to label if CX ≠ 0 and ZF = 0; otherwise (CX=0 or ZF=1), go to the next instruction.
JCXZ disp8	JMP if register CX =0.

Example:

```
        DEC   SI
        MOV   CX,50                 ; Initialize CX with array count
BACK:   INC   SI                    ; Update pointer
        CMP   BYTE PTR[SI],00H      ; Compare array element with 00H
        LOOPE BACK
```

String Instructions

String is an array of data bytes/words, stored in a consecutive memory Locations.

MNEMONICS	FUNCTION
MOVSB	*Move string byte from DS:[SI] to ES:[DI]*
MOVSW	*Move string word from DS:[SI] to ES:[DI]*
CMPSB	*Compare string byte (Done by subtracting byte at ES:[DI] from the byte at DS:[SI]). Only flags are affected and the content of bytes compared is unaffected.*
CMPSW	*Compare string word (Done by subtracting word at ES:[DI] from the word at DS:[SI]). Only flags are affected and the content of words compared is unaffected.*
LODSB	*Load string byte at DS:[SI] into AL*
LODSW	*Load string word at DS:[SI] into AX*
STOSB	*Store string byte in AL at ES:[DI]*
STOSW	*Store string word in AX at ES:[DI]*
SCASB	*Compare string byte (Done by subtracting byte at ES:[DI] from the byte at AL). Only flags are affected and the content of bytes compared is unaffected.*
SCASW	*Compare string word (Done by subtracting word at ES:[DI] from the byte at AX). Only flags are affected and the content of words compared is unaffected.*
REP	*Decrement CX and Repeat the following string operation if CX ≠ 0.*
REPE or REPZ	*Decrement CX and Repeat the following string operation if CX ≠ 0 and ZF=1.*
REPNE or REPNZ	*Decrement CX and Repeat the following string operation if CX ≠ 0 and ZF=0.*

8086 Instruction Set

Group 7 : String Instructions

EX: MOVS WORD.

Assume that:

(DF)=0 *(DS)=1000_{16}* *(SI)=0002_{16}* *(ES)=3000_{16}*

(DI)= 0004_{16} *(10002)=1234_{16}* *(30021)= 05I6*

Then, after this MOVS:

(30004)=1234_{16}, *(SI)=0004_{16},* *(DI)=0006_{16}*

Assuming (10002_{16}) = 1234_{16} → 8086 Insts to accomplish this???

8086 Instruction Set

Group 7 : String Instructions

```
CLD                     ;DF = 0
MOV     AX, 1000H       ;DS = 1000_16
MOV     DS, AX
MOV     BX, 3000H       ;ES = 3000_16
MOV     ES, BX
MOV     SI, 0002H       ;Initialize SI to 0002_16
MOV     DI, 0004H       ;Initialize DI to 0004_16
MOVSW
```

8086 Instruction Set

Group 7 : String Instructions

EX: if (DF) = 0, (DS) = 1000_{16}, (ES) = 3000_{16}, (SI) = 0002_{16},

(DI) = 0004_{16}, (10002) = 1234_{16}, (30004) = 1234_{16}

Then, after ***CMPS WORD:***

(10002) = 1234_{16}, (30004) = 1234_{16}, (SI) = 0004_{16}, (DI) = 0006_{16}.

Flags: CF = 0, PF= 1, AF= 1, ZF= 1, SF=0, OF =0

EX: if : (DI) = 0000_{16} , (ES) = 2000_{16} , (DF) = 0, (20000) = 05_{16}, (AL) = 03_{16},

Then, after SCASB:

DI will contain 0001_{16} because (DF) = 0.

All flags are affected based on the operation (AL) - (20000).

Emulator Example

```
data segment

   STR1 db 1, 2, 3, 4, 5

   STR2 db 6, 0, 7, 8, 9

   pkey db "press any key...$"

ends

stack segment
   dw   128  dup(0)
ends
```

```
code segment
start: ; set segment registers:
    mov ax, data
    mov ds, ax
    mov si, offset STR2
    cld

    mov cx, 5
    rep    movsb

    lea dx, pkey
    mov ah, 9
    int 21h        ; output string at ds:dx

    ; wait for any key....
    mov ah, 1
    int 21h

    mov ax, 4c00h ; exit to operating system.
    int 21h
ends

end start ; set entry point and stop the assembler.
```

Flags manipulation Instructions

- These instructions are ZERO operand instructions.*(Implied addressing Modes)*
- Can be executed any where in the code.

Mnemonics	Function
LAHF	Load low byte of flag register into AH
SAHF	Store AH into the low byte of flag register
PUSHF	Push flag register's content into stack
POPF	Pop top word of stack into flag register
CMC	Complement carry flag (CF = complement of CF)
CLC	Clear carry flag (CF= 0)
STC	Set carry flag (CF= 1)
CLD	Clear direction flag (DF= 0)
STD	Set direction flag (DF= 1)
CLI	Clear interrupt flag (IF= 0)
STI	Set interrupt flag (IF=1)

Input / Output Using DOS Interrupt 21H

SERVICE	DESCRIPTION	EXAMPLE
1	Read one character from the keyboard	MOV AH, 1 INT 21H AL= ASCI CHARCTER

2	Display a character	MOV AH, 2
		MOV DL, 35h
		Int 21H

9	Display a String	STR DB “Hello$” MOV AH, 9 MOV DX,OFFSET STR INT 21H
OAH	Reading String	inputarea db 10,O,10 dup(' ') mov dx,offset inputarea mov ah,OA int 21h

The ASCII table

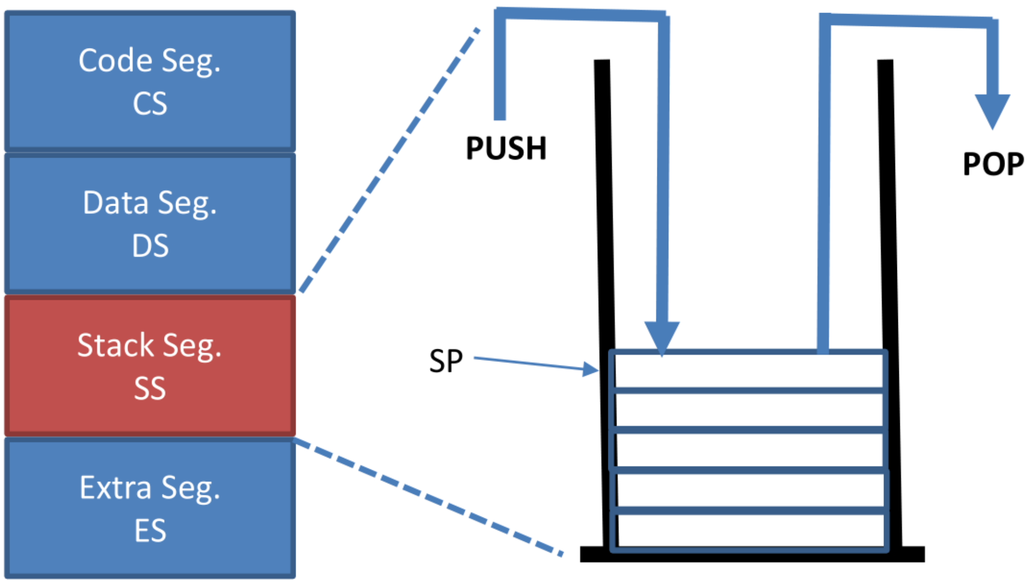
Character	Decimal	Hex	Binary	Octal
0	48	30	110000	60
1	49	31	110001	61
2	50	32	110010	62
3	51	33	110011	63
4	52	34	110100	64
5	53	35	110101	65
6	54	36	110110	66
7	55	37	110111	67
8	56	38	111000	70
9	57	39	111001	71

Character	Decimal	Hex	Binary	Octal
A	65	41	1000001	101
B	66	42	1000010	102
C	67	43	1000011	103
D	68	44	1000100	104
E	69	45	1000101	105
F	70	46	1000110	106
G	71	47	1000111	107
H	72	48	1001000	110
I	73	49	1001001	111
J	74	4A	1001010	112
K	75	4B	1001011	113
L	76	4C	1001100	114
M	77	4D	1001101	115
N	78	4E	1001110	116
O	79	4F	1001111	117
P	80	50	1010000	120
Q	81	51	1010001	121
R	82	52	1010010	122
S	83	53	1010011	123
T	84	54	1010100	124
U	85	55	1010101	125
V	86	56	1010110	126
W	87	57	1010111	127
X	88	58	1011000	130
Y	89	59	1011001	131
Z	90	5A	1011010	132

Character	Decimal	Hex	Binary	Octal
a	97	61	1100001	141
b	98	62	1100010	142
c	99	63	1100011	143
d	100	64	1100100	144
e	101	65	1100101	145
f	102	66	1100110	146
g	103	67	1100111	147
h	104	68	1101000	150
i	105	69	1101001	151
j	106	6A	1101010	152
k	107	6B	1101011	153
l	108	6C	1101100	154
m	109	6D	1101101	155
n	110	6E	1101110	156
o	111	6F	1101111	157
p	112	70	1110000	160
q	113	71	1110001	161
r	114	72	1110010	162
s	115	73	1110011	163
t	116	74	1110100	164
u	117	75	1110101	165
v	118	76	1110110	166
w	119	77	1110111	167
x	120	78	1111000	170
y	121	79	1111001	171
z	122	7A	1111010	172

8086 STACK

- Consists of 16-bit locations
- SP points at the top of the stack
- PUSH instruction is used add data to the top of the stack
- POP instruction is used to remove items from the top of the stack
- STACK is a LIFO structure
- SP is decremented with every PUSH
- SP is incremented with every POP
- Procedures use the stack
- The CALL instruction adds the return address on the top of the stack
- The RET instructions removes the address from the top of the stack
- Every PUSH Instruction must be associated with a matching POP
- Every CALL must also be associated with a matching RET
- Must be careful when using PUSH, POP, CALL, and RET



Using the STACK

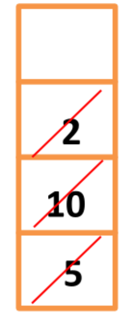
- Useful for storing and retrieving data (16-bit)
- Useful for implementing the concept of local variables
- Can be used to reverse the order of stored data
- Useful for problems requiring backtracking

```
MOV   AX,5
MOV   BX,10
MOV   CX,2
PUSH  AX
PUSH  BX
PUSH  CX
DEC   AX
ADD   BX,2
MOV   CX, BX
POP   CX
POP   BX
POP   AX
```

Note: the POPs are in reverse order of the PUSHs

AX	BX	CX
5	10	2
4	12	12
5	10	2

STACK

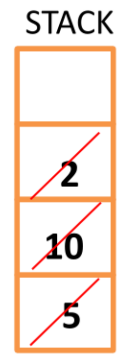


```
MOV   AX,5
MOV   BX,10
MOV   CX,2
PUSH  AX
PUSH  BX
PUSH  CX
DEC   AX
ADD   BX,2
MOV   CX, BX
POP   AX
POP   BX
POP   CX
```

Note: the POPs are in same order of the PUSHs

AX	BX	CX
5	10	2
4	12	12
2	10	5

Note: register values are reversed



EXAMPLE: USING CALL, RET, PUSH AND POP

Call pushes the address of the next instruction on the stack. RET: returns back the address from the top of the stack

REMEMBER

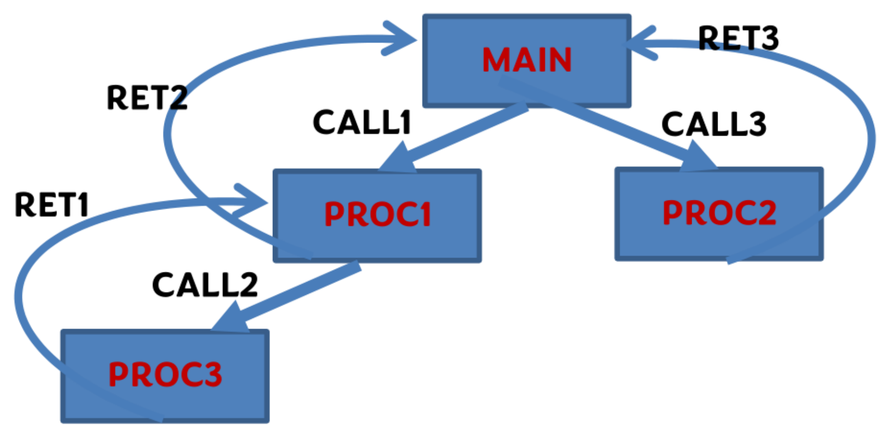
```
.DATA
MPROC1 DB "IN PROCEDURE 1",0DH,0AH,"$"
MPROC2 DB "IN PROCEDURE 2",0DH,0AH,"$"
MPROC3 DB "IN PROCEDURE 3",0DH,0AH,"$"

.CODE
  MOV    AX,5
  PUSH   AX
  CALL   PROC1
  CALL   PROC2
  MOV    AH,4CH
  INT    21H

  PROC1 :
        MOV AH,9
        LEA DX, MPPOC1
        INT 21H
        MOV   AX,2
        PUSH  AX
        POP   AX
        CALL  PROC3
  RET

PROC2 :
        MOV AH,9
        LEA DX, MPROC2
        INT 21H
  RET

PROC3 :
        MOV AH,9
        LEA DX, MPROC3
        INT 21H
  RET
END
```



Please Practice more examples using EMU 8086