# EC-310 Microprocessor & Microcontroller based Design

**Dr. Taimoor Zahid**

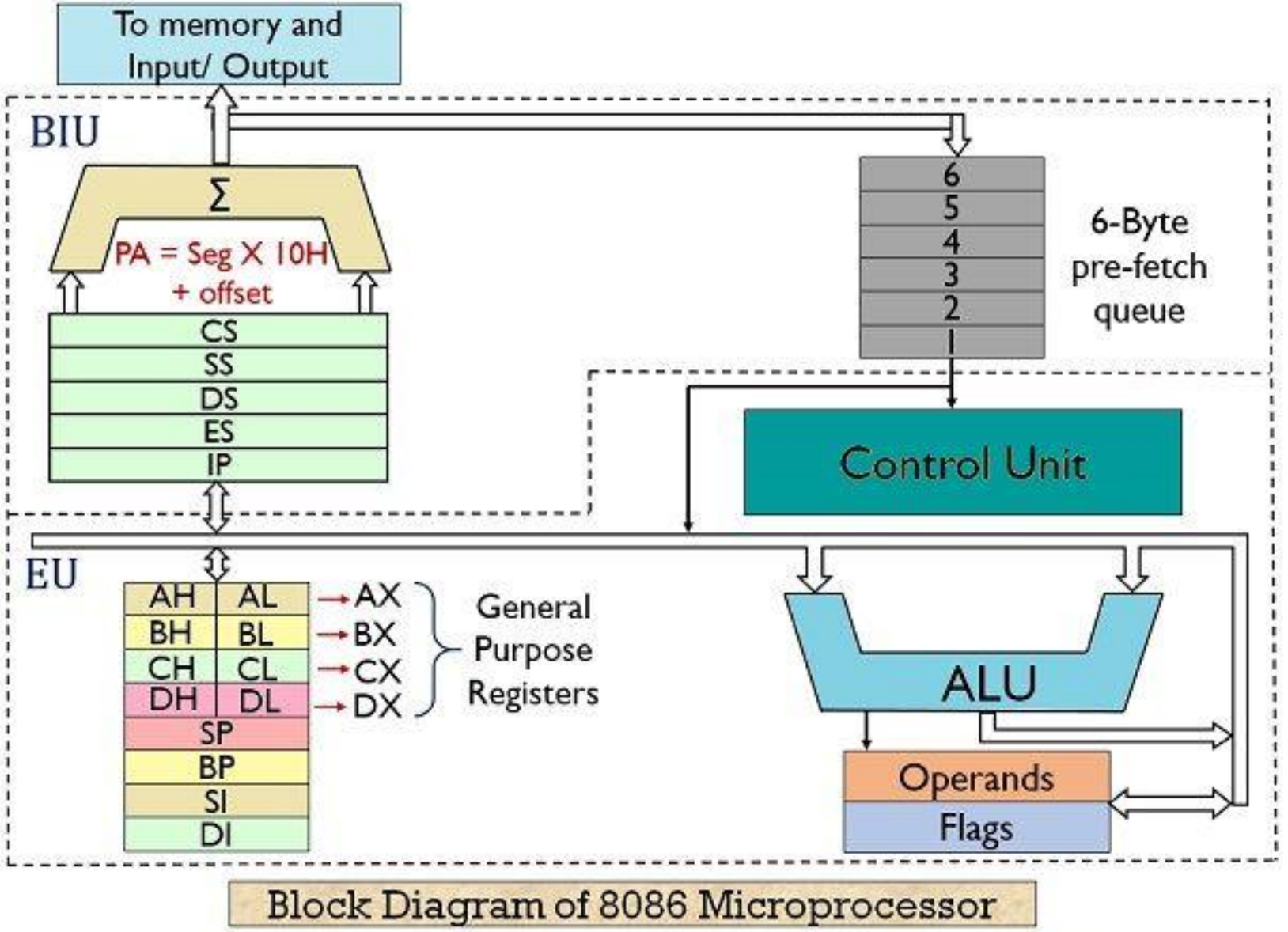
**Department of Computer and Software Engineering (DC&SE)**

**College of Electrical and Mechanical Engineering (CEME)**

**National University of Sciences and Technology (NUST)**

Week # 2

# Internal Architecture of Intel 8086 Microprocessor




Block Diagram of 8086 Microprocessor

# Assembly Language Syntax

| Name | operation | operand(s) | comment |
|---|---|---|---|

Assembly program consists of statements that are either,

- **Instruction** (assembler translates to machine code)
- **Assembler Directives** (instructions for assembler e.g. allocating memory)

## Name Field

- Used for instruction labels, procedure names and variable names
- Up to 31 characters long, consists of letters, digits and special characters (? . @ _ $ %)
- Examples:

**@sum $1000 .test done? X_y total value 7even X. Y&m**

# Assembly Language Syntax

| Name | operation | operand(s) | comment |
|---|---|---|---|

## Operation Field

- **Instructions** -> symbolic operation code (Opcode) -> translated to machine opcode
  - Examples:  MOV, ADD, SUB, NEG etc.
- **Assembler Directives** -> pseudo operation code (pseudo-op) -> not translated to machine code
  - Examples: PROC to create a procedure

# Assembly Language Syntax

| Name | operation | operand(s) | ;comment |
|---|---|---|---|

## Operand Field

- **Instructions** -> specifies the data for the operation
  - Examples: NOP (no operand), INC AX (one operand), ADD AX , DX (two- Destination, Src)

- **Assembler Directives** -> contains more information about the directive

# Variables or Defining Data

## Data Defining Pseudo-ops

- Each variable has a data type and a memory address assigned to it
- Pseudo-ops can be used to generate one or more data items

| Pseudo-op | Stands for | Size |
|---|---|---|
| DB | Define byte | 8 bits |
| DW | Define word | 2 bytes |
| DD | Define double word | 4 bytes |
| DQ | Define quad word | 8 bytes |
| DT | Define ten bytes | 10 bytes |

# Variables or Defining Data

## Byte Variables

- Format: [variable name] **DB** **[**Initial Value] Range: -128~127 **OR** 0~255
- Examples: Alpha DB 4 Beta DB ?

## Word Variables

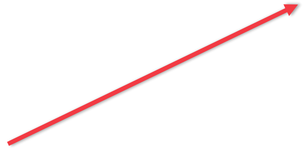
- Format: : [variable name] **DB** **[**Initial Value] Range: -32768~32767 **OR** 0~65535
- Examples: Alpha DW 1234H Beta DW ?

| | Offset | Contents |
|---|---|---|
| Alpha | 0000h | 34 |
| Alpha + 1 | 0001h | 12 |

# Variables or Defining Data

## Arrays/Lists

- Sequence of memory bytes or words
- Examples:
  - B_Array DB 10H,20H,30H
  - W_Array DW 1000,50, 4568, 30
  - Letters DB ‘ABC’
  - LETTERS DB 41h, 42h, 43h
  - MSG DB ‘HELLO’, 0AH, 0DH, ‘$’



Byte Offset

| Offset | Contents |
|---|---|
| 0000h | 1000d |
| 0002h | 50d |
| 0004h | 4568d |
| 0006h | 30d |

# Named Constants

## EQU (Equates)

- To assign name to a constant
- No memory is allocated for EQU names
- Examples:
  - LF EQU 0AH
  - CR EQU 0DH
  - PROMT EQU ‘TYPE YOUR NAME’

# Data Transfer Instructions

## MOV, XCHG

- MOV Destination, Source
- E.g. MOV AX, BX

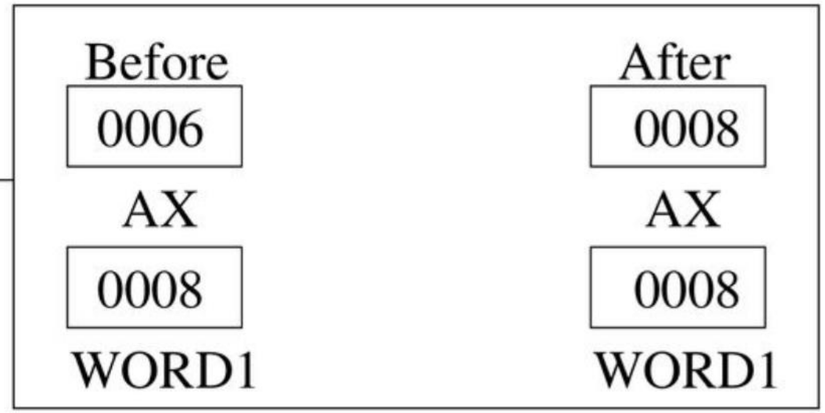
**Legal Combinations of operands for MOV**

| Source operand | Destination Operand: General register | Segment register | Memory location | Constant |
|---|---|---|---|---|
| General register | yes | yes | yes | no |
| Segment register | yes | no | yes | no |
| Memory location | yes | yes | no | no |
| Constant | yes | no | yes | no |

- Illegal: MOV WORD1, WORD2
- Legal: MOV AX, WORD2
  MOV WORD1, AX
- Illegal: MOV DS, CS
- Legal: MOV AX, CS
  MOV DS, AX

- XCHG AX, Word1

Example:
MOV AX, WORD1
MOV AX, BX
MOV AX, 'A'




➢**Both operands can't be memory locations or segment registers**

# Arithmetic Instructions

## ADD, SUB

- ADD destination, Source — ADD AX,DX ; AX = AX+DX
- SUB destination, Source — SUB AX,DX ; AX = AX-DX

➢**Both operands can't be memory locations**

## INC, DEC

- Increment or decrement by 1 in the contents of memory location or register
- INC AX DEC AX

## NEG

- Negate the contents of destination by taking 2's complement. E.g. NEG BX

# I/O Instructions

## IN, OUT Instructions

- Access the I/O ports directly and provides fast communication.
- **Cons:** Port addresses vary among computers
- I/O service routines provided by manufacturer are easier to use

- Two Categories of I/O service routines:
    1) Basic Input / Output System Routines (BIOS)
    2) The DOS routines

# I/O Instructions

## INT Instruction

- To invoke DOS or BIOS routine, the INT (interrupt) instruction is used.
- A particular function is requested by placing a function number in the AH register and invoking **INT 21h**
- Format: **INT interrupt_number e.g. INT 16H OR INT 21Hh**

## INT 21H

- Functions expect input values to be in certain registers and return output values in other registers

**Function 1**: Single Key Input

**Input:** AH = 1

**Output:** AL = ASCII code if character key is pressed
= 0 if non-character key is pressed

# I/O Instructions - INT 21H

**Function 2:** Display Character or Execute control function

**Input:** AH = 2

DL = ASCII code of the character

**Output:** AL = ASCII code of the character

| ASCII code (Hex) | Symbol | Function |
|---|---|---|
| 07H | BEL | beep (sounds a tone) |
| 08H | BS | backspace |
| 09H | HT | tab |
| 0AH | LF | line feed (new line) |
| 0DH | CR | carriage return (start of current line) |

Function 9: **Display a String**

**Input:** DX = Offset address of string

The string must end with a '$' character.

**Output:** On the screen

# LEA Instruction

## LEA (Load Effective Address)

- LEA destination, Source
  - ➢ Destination -> General Register  Source-> Memory Location
- Copies the offset address of the memory location to register

- To print HELLO! On the screen:
  - MSG DB ‘HELLO!$’
  - LEA DX, MSG  OR  MOV DX, OFFSET MSG
  - MOV AH, 9
  - INT 21h

# Program Structure

## Program Skeleton

```
.MODEL SMALL
.STACK 100H
.DATA
        ; data definitions go here
.CODE
MAIN PROC
        ; instructions go here
MAIN ENDP
        ;other procedures go here
END MAIN
```

# Program Structure - Memory Models

- Size of code and data of a program is determined by specifying memory models
- .MODEL directive is used to specify memory model
  - **Tiny:** code + data <= 64K
  - **Small:** One segment for each
  - **Medium:** data = 1 segment, Code in more than 1 segment
  - **Compact:** code = 1 segment, Data in more than 1 segment
  - **Large:** multiple code & data segments, no array > 64KB
  - **Huge:** multiple code & data segments, arrays can exceed 64KB

# Program Structure - Data Segment

- Contains all variable definitions
- Constant definitions are often made here as well but can be place elsewhere in the code

```
.Data

Word1 DW 2

Word2 DW 5

MASK EQU  10001111B
```

# Program Structure - Stack Segment

- **Format:** .Stack Size
- To set aside a block of memory to be used for stack
- If size is omitted, **default value is 1KB**
- ~~Stack data access principle?~~

# Program Structure - Code Segment

- It contains program's instructions
- **Format:** .CODE NAME
- NAME is *optional parameter* and not needed for .MODEL SMALL (only 1 code segment)
- Inside code segment, Instructions can be organized as procedures

```
Name PROC
    ; body of the procedure
Name ENDP
```

# Sample Assembly Program

## Case Conversion Program

```
. MODEL SMALL
.STACK 100H
.DATA
        CR   EQU  0DH
        LF   EQU  0AH
        MSG1 DB  'Enter a lower case letter:   $'
        MSG2 DB  CR, LF, 'In upper case it is:  '
        CHAR DB  ?,'$'
.CODE
MAIN PROC
```

# Sample Assembly Program

## Case Conversion Program

```
        MOV AX, @DATA   ; initialize DS
        MOV DS, AX
; print user prompt
        LEA DX, MSG1
        MOV AH, 9
        INT 21H
;input a character and convert to upper case
?
?
```

Or

**.Startup**

**Program Segment Prefix (PSP)**

- DOS prefaces a 256bytes of program related information, when it is loaded into memory
- DOS places it's segment number in both DS and ES
- DS doesn't contain Data segment number at the start

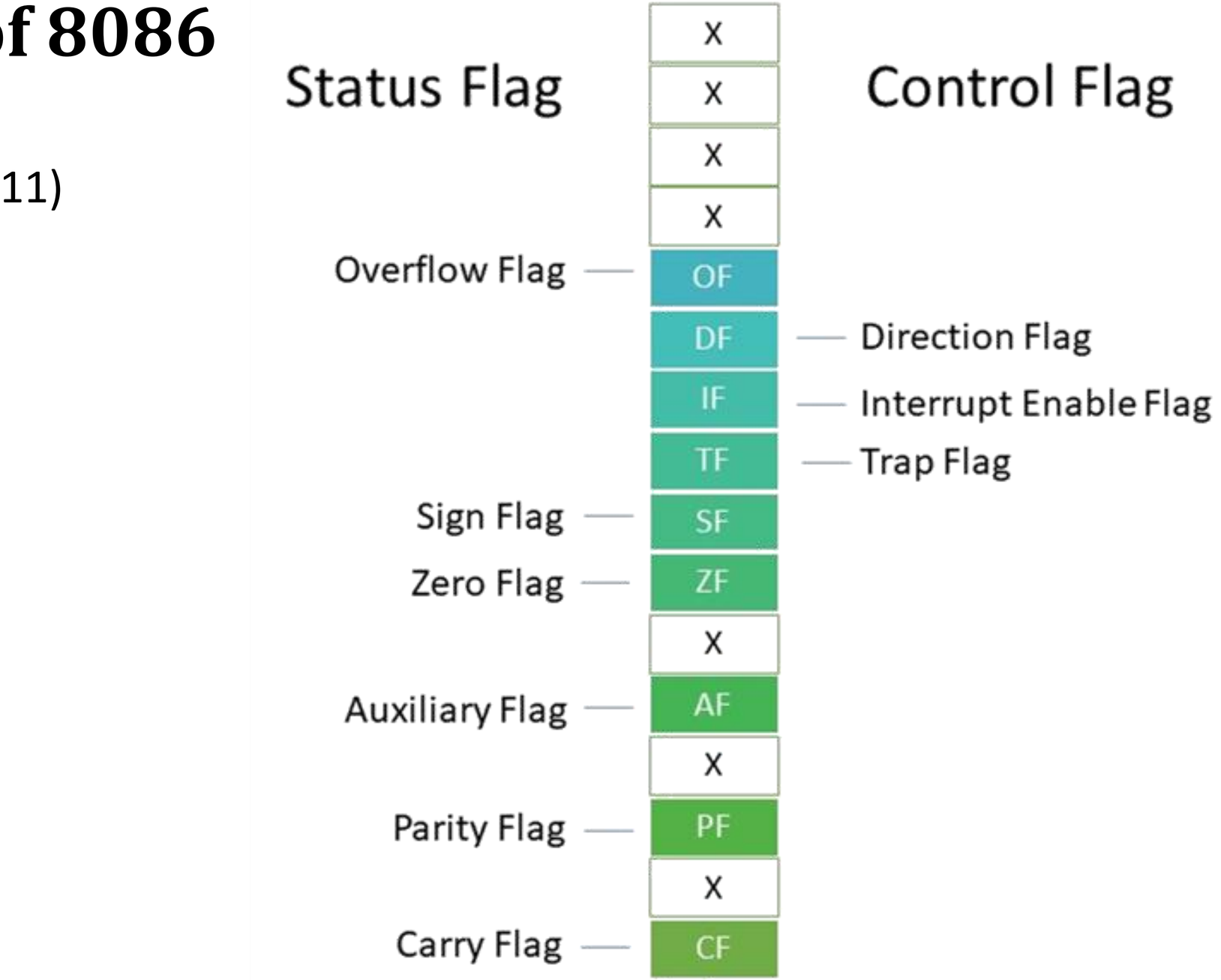
# Sample Assembly Program

## Case Conversion Program

```
; display on the next line
        LEA DX, MSG2
        MOV AH, 9
        INT 21H
;DOS exit
        MOV AH, 4CH
        INT 21H
MAIN ENDP
        END MAIN
```

# FLAGS Register of 8086

- **6** Status Flags (0,2,4,6,7,11)
- **3** Control Flags (8,9,10)



# Status Flags

## Carry Flag (CF)

- **CF =1,** when carry out or borrow into the MSB happens (e.g. ADD, SUB,  instr.)
- **PF = 1,** if Low byte of result has even number of one bits (even parity)
- **AF = 1,** when carry out or borrow into the bit 3 (used in BCD operations)
- **ZF = 1,** for a zero result
- **SF = 1,** if MSB  of a result is 1 (negative no.)
- **OF =1,** if signed overflow occurs, otherwise it is 0

| | | | | 11 | 10 | 9 | 8 | 7 | 6 | | 4 | | 2 | | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | OF | DF | IF | TF | SF | ZF | | AF | | PF | | CF |

*The Flags Register*

# Overflow

- Overflow occurs when the **result falls outside the range** supported by given memory storage
- Signed & unsigned overflows are independent to each other
- Four Cases: 1) No overflow 2) Signed only 3) Unsigned only 4) Both
- **Examples:**

AL = 0FFh, BL =01h

**ADD AL, BL**

1111 1111
+00000001
1 0000 0000

*Unsigned overflow*

AL = 7Fh, BL =7Fh

**ADD AL, BL**

0111 1111
+0111 1111
1111 1110

*Signed overflow*

AL = 0FEh, BL =81h

**ADD AL, BL**

1111 1110
+1000 0001
1 0111 1111

*Both*

# Overflow

- How the Processor indicates overflow ?
  - Processors sets OF =1 for signed overflow and CF = 1 for unsigned overflow
- How the Processor determines that overflow has occurred ?
- Unsigned Overflow
  - When there is a carry out OR a borrow into the MSB (CF = 1)
- Signed Overflow
  - When the carry into and out of the MSB don’t match, then signed overflow occurs
  - OF = 1

# Instructions Affecting Flags

- Not all instructions affect all the flags
- Example:

| Instructions | Affect Flags |
|---|---|
| MOV, XCHG | None |
| ADD, SUB | All |
| INC, DEC | All except CF |
| NEG | All (CF = 1 unless result is 0, OF = 1 if byte operand is 80h or word operand 8000h) |

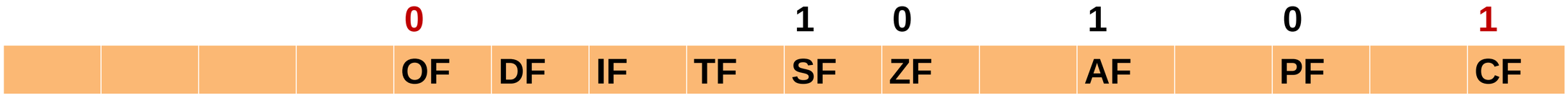
AL = 0FFh, BL =0FFh

**ADD AL, BL**

1111 1111

+1111 1111

1 1111 1110



# Instructions Affecting Flags

- Not all instructions affect all the flags

| Instructions | Affect Flags |
|---|---|
| MOV, XCHG | None |
| ADD, SUB | All |
| INC, DEC | All except CF |
| NEG | All (CF = 1 unless result is 0, OF = 1 if byte operand is 80h or word operand 8000h) |

- Example:

AX = 8000h

**NEG AX**

**8000h =** 1000 0000 0000 0000

1's compl.= 0111 1111 1111 1111

+1

1000 0000 0000 0000

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 1 | | | | 1 | 0 | | 1 | | 1 | | 1 |
| | | | | OF | DF | IF | TF | SF | ZF | | AF | | PF | | CF |