Computer Organizations and Architecture

Introduction to Assembly Language Coding

Spring 24-25, CS 3205, Section D

Dr. Nazib Abdun Nasir

Assistant Professor, CS, AIUB

nazib.nasir@aiub.edu

April 28, 2025

Outline

- Programming Languages
- Number Systems
- Conversion Between Number Systems
- Intel 8086 Microprocessor
  - → Architecture, Registers
- ASCII Table
- Introduction to Assembly Language
  - → Semantics and Syntax
  - → Assembly Instructions
  - → Code Example and Explanation
- FLAGS Registers
  - → Status FLAGS
  - → Carry FLAGS

Review of Previous Knowledge

Programming Languages

- Low-Level Languages
  - → Machine Language: Binary Bits (0 & 1)
  - → Assembly Language: Symbolic Syntax (MOV AX, A)
- High-Level Languages
  - → Closer to Human Language

Number Systems

- Number systems typically have a base (*b*) and a positional value (in terms of power of the base) for each digit.
  - → Binary
  - → Decimal
  - → Hexadecimal
  - → Octal

Conversion Between Number Systems

- Binary to Decimal Conversion
  - → To convert a binary number to decimal, each digit (bit) is multiplied by 2 raised to the power of its position (starting from 0 on the right). The results are then summed.

Example:

Convert 1011_2 to decimal.

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}$$

- Decimal to Binary Conversion
  - → To convert a decimal number to binary, repeatedly divide the number by 2 and record the remainders. The binary equivalent is formed by reading the remainders from bottom to top.

Example:

Convert 13_{10} to binary.

1. $13 \div 2 = 6$ remainder 1
2. $6 \div 2 = 3$ remainder 0
3. $3 \div 2 = 1$ remainder 1
4. $1 \div 2 = 0$ remainder 1

Reading from bottom to top, we get 1101_2.

Conversion Between Number Systems

- Hexadecimal to Decimal Conversion
  - → In hexadecimal (base 16), each digit represents a power of 16. Convert each digit to decimal and sum them up.

Example:

Convert $1A3_{16}$ to decimal.

$$1 \times 16^2 + A(10) \times 16^1 + 3 \times 16^0 = 256 + 160 + 3 = 419_{10}$$

- Decimal to Hexadecimal Conversion
  - → To convert a decimal number to hexadecimal, divide the number by 16 and record the remainders. Read the remainders from bottom to top.

Example:

Convert 255_{10} to hexadecimal.

1. $255 \div 16 = 15$ remainder $15(F)$
2. $15 \div 16 = 0$ remainder $15(F)$

Reading from bottom to top gives us FF_{16}.

Conversion Between Number Systems

› Hexadecimal to Binary Conversion

→ Each hexadecimal digit can be directly converted into a four-bit binary equivalent.

Example:

Convert $B4_{16}$ to binary.

- B = 1011_2
- 4 = 0100_2

Thus, $B4_{16} = 10110100_2$.

› Binary to Hexadecimal Conversion

→ Group the binary digits into sets of four (from right to left), and convert each group into its hexadecimal equivalent.

Example:

Convert 11010111_2 to hexadecimal.

Grouping:

- $1101|0111$

Converting:

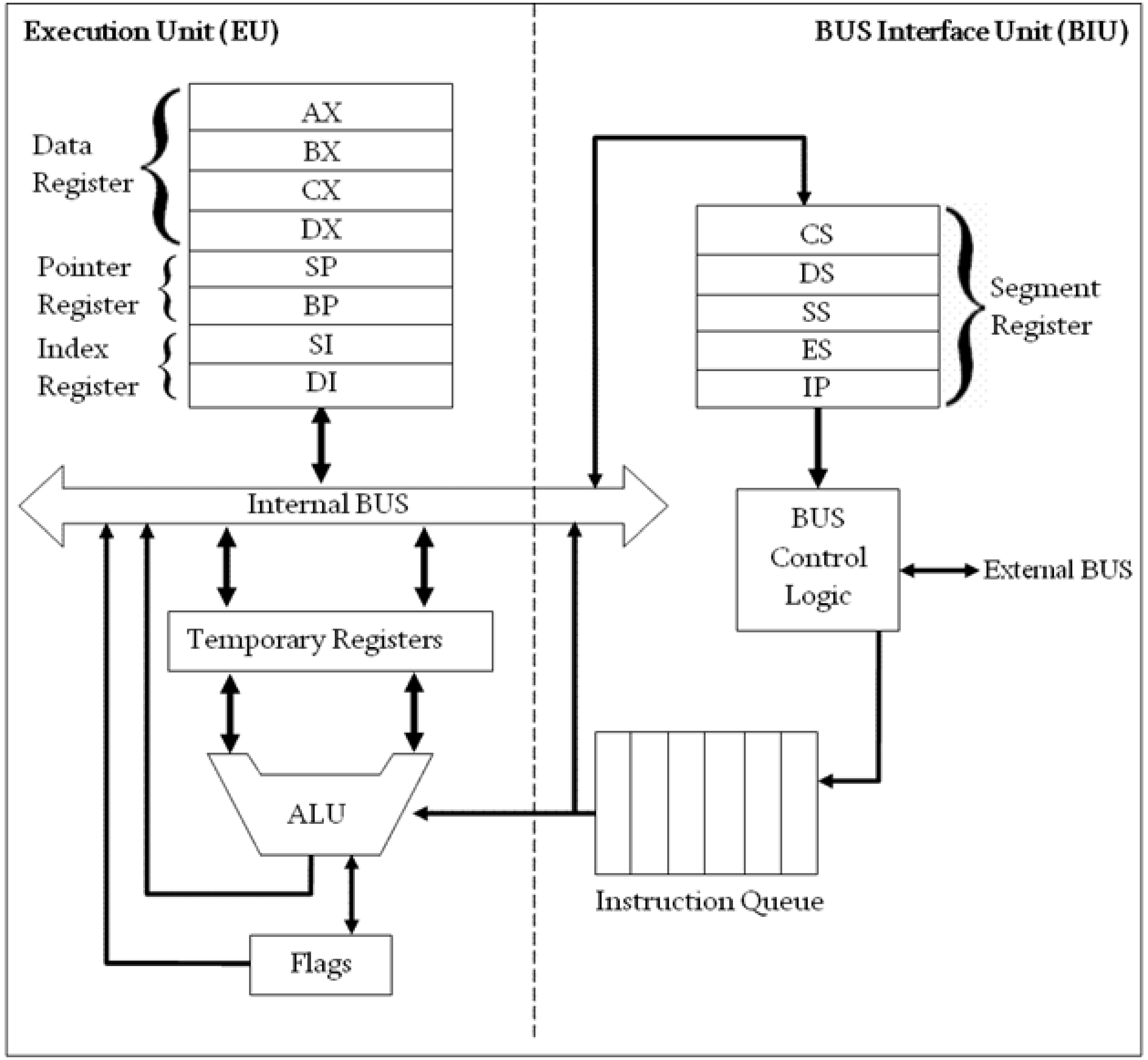
- $1101 = D_{16}$
- $0111 = 7_{16}$

Thus, $11010111_2 = D7_{16}$.

Conversion Between Number Systems

- Convert the following numbers
  - → 589 Hexadecimal to Decimal → 1417
  - → 415 Decimal to Hexadecimal → 19F
  - → 39A2 Hexadecimal to Binary → 0011100110100010
  - → 11010011 Binary to Decimal → 211
  - → 47 Decimal to Binary → 101111
  - → 1000011001000011 Binary to Hexadecimal → 8643

Intel 8086 Microprocessor Organization/Architecture



Intel 8086 Microprocessor Registers

General Register	AX	Accumulator AH	AL
	BX	Base BH	BL
	CX	Counter CH	CL
	DX	Data DH	DL
Segment Register		CS (code segment)	
		DS (data segment)	
		SS (stack segment)	
		ES (extra segment)	
Pointer and Index Register		IP (instruction pointer)	
		SP (stack pointer)	
		BP (base pointer)	
		SI (source index)	
		DI (destination index)	
FLAGS Register		Flag Register	

Intel 8086 Microprocessor Registers

- General Purpose Registers
  - → Accumulator Register (AX): Used primarily for arithmetic operations; it holds operands and results. It can be accessed as two separate 8-bit registers: AL (lower byte) and AH (higher byte).
  - → Base Register (BX): Holds the base address for memory access, facilitating data reading and writing.
  - → Counter Register (CX): Used as a counter in loop operations and for shift/rotate instructions. It can also be accessed as CL (lower byte) and CH (higher byte).
  - → Data Register (DX): Primarily used in multiplication and division operations where it can hold the high-order bits of results. It can also be accessed as DL (lower byte) and DH (higher byte).
- Segment Registers
  - → Code Segment Register (CS): Points to the segment containing executable code.
  - → Data Segment Register (DS): Points to the segment where variables are stored.
  - → Stack Segment Register (SS): Points to the segment used for stack operations, such as function calls and local variables.
  - → Extra Segment Register (ES): Used for additional data storage, particularly during string operations or when the data segment is insufficient.

Intel 8086 Microprocessor Registers

- Special Purpose Registers
- Pointer Registers
  - Stack Pointer (SP): This 16-bit register points to the current top of the stack in memory. It is automatically updated during push and pop operations, enabling efficient management of function calls and local variables.
  - Base Pointer (BP): Also a 16-bit register, the BP is used to point to the base address of the stack segment. It is particularly useful for accessing parameters passed to functions and local variables within a stack frame.
  - Instruction Pointer (IP): The IP register holds the address of the next instruction to be executed. It is automatically incremented after each instruction fetch, ensuring that the processor executes instructions sequentially unless directed otherwise by control flow instructions.
- Index Registers
  - Source Index (SI): This 16-bit register is used to hold the offset address of the source operand in string manipulation operations. It can be automatically incremented or decremented based on the direction flag, allowing for efficient processing of strings.
  - Destination Index (DI): Similar to SI, this 16-bit register holds the offset address of the destination operand during string operations. It works in conjunction with SI to facilitate copying or moving data between memory locations.
- Flag Register: The flag register is a 16-bit register that contains status flags indicating the outcome of operations performed by the processor. It includes:
  - Status Flags: Such as Carry Flag, Parity Flag, Auxiliary Carry Flag, Zero Flag, Sign Flag, and Overflow Flag.
  - Control Flags: Include Interrupt Flag, Direction Flag, and Trap Flag.

ASCII Table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Introduction to Assembly Language

Semantics and Syntax

- EMU 8086 executes one line/statement each time.
- Case Sensitive?
  - No. Upper case for coding.
- Statements (Instructions and Directives) can have 4 fields.

Name	Operation	Operand(s)	Comment
START	MOV	CX, 5	; initialize counter

- Name field cannot have blanks or begin with a number.

Semantics and Syntax

- Operation field takes in any of the built-in commands.
- Operand field indicates what registers/data to be operated on.
  - → Can have 0, 1, or most commonly 2 operands.
  - → First operand in **Destination** and second operand is **Source**.
- Comment field is used to add comments to the program.
  - → Starts with a semicolon (;).

Assembly Instructions

- MOV
  - MOV is used to transfer data between registers, register and memory-location or move number directly into register or memory location.
  - MOV AX, 2
- XCHG
  - XCHG is used to exchange the contents between two registers or register and memory-location.
  - XCHG AH, BL

Assembly Instructions

- ADD
  - → ADD is used to add content of two registers, register and memory-location or add a number to register or memory location.
  - → ADD BX, NUM
- SUB
  - → SUB is used to subtract content of two registers, register and memory-location or subtract a number from register or memory location.
  - → SUB AX, DX

Assembly Instructions

- INC
  - → INC is used to add 1 to the contents of a register or memory-location. EX: INC BL
- DEC
  - → DEC is used to subtract 1 from the contents of a register or memory-location.
  - → DEC WORD
- NEG
  - → NEG is used to negate the contents of the destination.
  - → NEG BX

Practice Coding

High-Level Statement	Assembly Language Code
B = A	
A = 5 - A	
A = B – 2 * A	

Practice Coding

High-Level Statement	Assembly Language Code	

B = A	MOV AX, A
	MOV BX, B
	MOV B, AX	

A = 5 - A	MOV AX, 5
	SUB AX, A
	MOV A, AX
		MOV AX, A
		NEG AX
		ADD AX, 5

A = B – 2 * A	MOV BX, B
	MOV AX, A
	ADD AX, A
	SUB BX, AX
	MOV A, BX	

Code Example and Explanation

```
.MODEL SMALL
.STACK 100H

.DATA

    MSG DB 0AH, 0DH, "Hello World$" ; comment

.CODE

MAIN:

    MOV AX, @DATA
    MOV DS, AX

    LEA DX, MSG
    MOV AH, 09H
    INT 21H

    MOV AH, 4CH
    INT 21H

END MAIN
```

Code Example and Explanation

- **.MODEL SMALL:** This directive specifies that the program will use the "small" memory model, which allows for one code segment and one data segment. This is suitable for small programs that fit within these constraints.
- **.STACK 100H:** This allocates a stack size of 256 bytes (100 hexadecimal) for storing temporary data such as function parameters and local variables.
- **.DATA:** This marks the beginning of the data segment where variables and constants are declared.
- **MSG DB 0AH, 0DH, "Hello World$":**
- - ‘MSG’ is a label for the string data.
- - ‘DB’ (Define Byte) is used to declare a byte or series of bytes in memory.
- - ‘0AH’ and ‘0DH’ are ASCII codes for line feed (LF) and carriage return (CR), respectively.
- - They are used to move the cursor to a new line before printing "Hello World".
- - "Hello World$" is the string to be printed. The ‘$’ character indicates the end of the string for DOS function 09h.
- **.CODE:** This indicates the start of the code segment where executable instructions are written.
- **MAIN:** This label marks the entry point of the program.

Code Example and Explanation

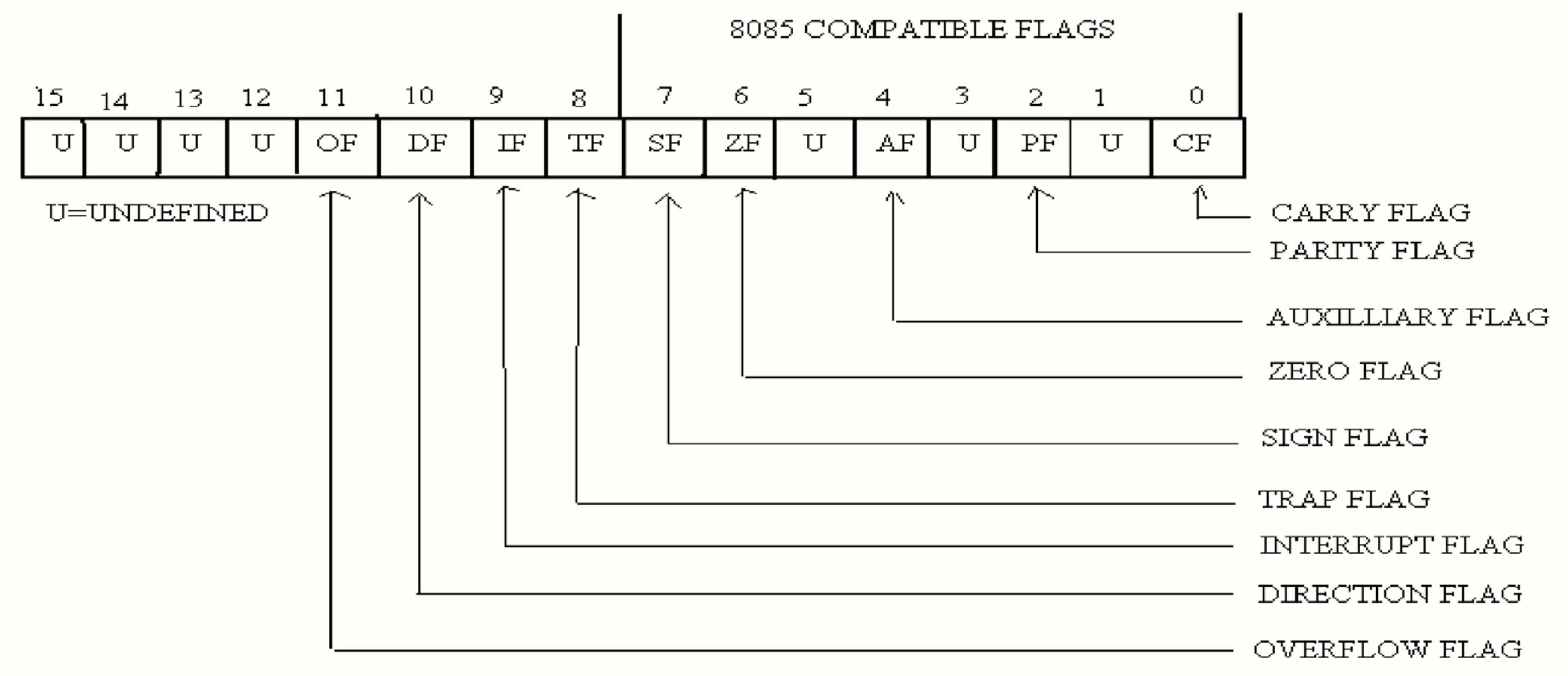
- **MOV AX, @DATA:** Loads the address of the data segment into register 'AX'. '@DATA' is a special symbol that refers to the beginning of the data segment.
- **MOV DS, AX:** Moves the value in 'AX' into 'DS', effectively setting up the data segment register to point to our data segment. This allows access to variables defined in '.DATA'.
- **LEA DX, MSG:** The 'LEA' (Load Effective Address) instruction loads the address of 'MSG' into register 'DX'. This address will be used by DOS to know where to find the string to display.
- **MOV AH, 09H:** Sets up for displaying a string by loading 'AH' with '09h', which is the function number for displaying a string using DOS interrupt 21h.
- **INT 21H:** Calls DOS interrupt '21h', executing function '09h', which prints the string located at the address in 'DX'. The string will be printed until it encounters a '$'.
- **MOV AH, 4CH:** Prepares to exit the program by setting 'AH' to '4Ch', which is the function number for terminating a program in DOS.
- **INT 21H:** Calls DOS interrupt '21h', executing function '4Ch', which terminates the program and returns control back to DOS.
- **END MAIN:** This directive indicates that this is where the program ends and specifies that execution should start at 'MAIN'.

FLAGS Registers

FLAGS Registers

- The 8086 processor’s state is represented with nine individual bits or flags.
- The flags are placed in the FLAGS register.
- Status FLAGS: Reflects the result of computation.
- Control FLAGS: used to enable or disable certain operations of processor.

FLAGS Registers



Status Flags: bit **0, 2, 4, 6, 7** and **11**

Control Flags: bit **8, 9** and **10**

***** bit 1,3,5,12,13,14,15** has **no significance**

nazib.nasir@aiub.edu

References

- All Previous Lectures