Week-01

Few Instructions For Both Theory and Sessional

When we are in the classroom:

- ➔ No Side Talking
- ➔ No Group Talking
- ➔ No Yelling
- ➔ No Pairing
- ➔ No Browsing or Messaging
- ➔ After 10 mins late, attendance may not consider

These will impact your observation as well as final marks.

Topics

- About CSE-305
- Why Assembly and 8086
- What is 8086
- System Architecture 8086
  - Registers
  - Segments
- Basic of Assembly Language
  - Program Structure
  - Input Output
  - Basic Instruction
- Emulator

Book: Assembly Language Programming and Organization of the IBM PC

About CSE-305

Course Outline: [CSE-305]

Topics: [CSE-305_Week_wise]

Classroom Code: os55nic

- Course Teacher:
  - Assoc Prof Abdus Sattar
  - Lec Mustaqim Abrar (Myself)
- My Topics
  - Assembly Language
  - Microcontroller

Instruction about Sessional

- Create a group of 4 members for the sessional and project.
- Regarding this a link of google spreadsheet will be given in google classroom.
- Put your group members name & ID in that spreadsheet before your sessional class(**Before Lab**).

User vs Programmer vs Computer Scientist

Using a **developed** software (set of instruction).

Building a software for a **particular architecture** or in a **particular language**.

Building and Design the **system architecture** or the **compiler** or the **interpreter**.

Only by studying assembly language it is possible to gain a feeling for the way the computer "thinks and why certain things happen the way they do inside the computer. High-level languages tend to obscure the details of the compiled machine language program that the computer actually executes.”

Why Assembly & 8086

- To understand the **system architecture**, we will see 80X86 as examples
- **To understand how the Higher Level Instruction is Executed by the Hardware**

High Level Instruction	Low Level Language	Machine Code
a = a + b	MOV AX, B ADD A, AX	01000111 11010001

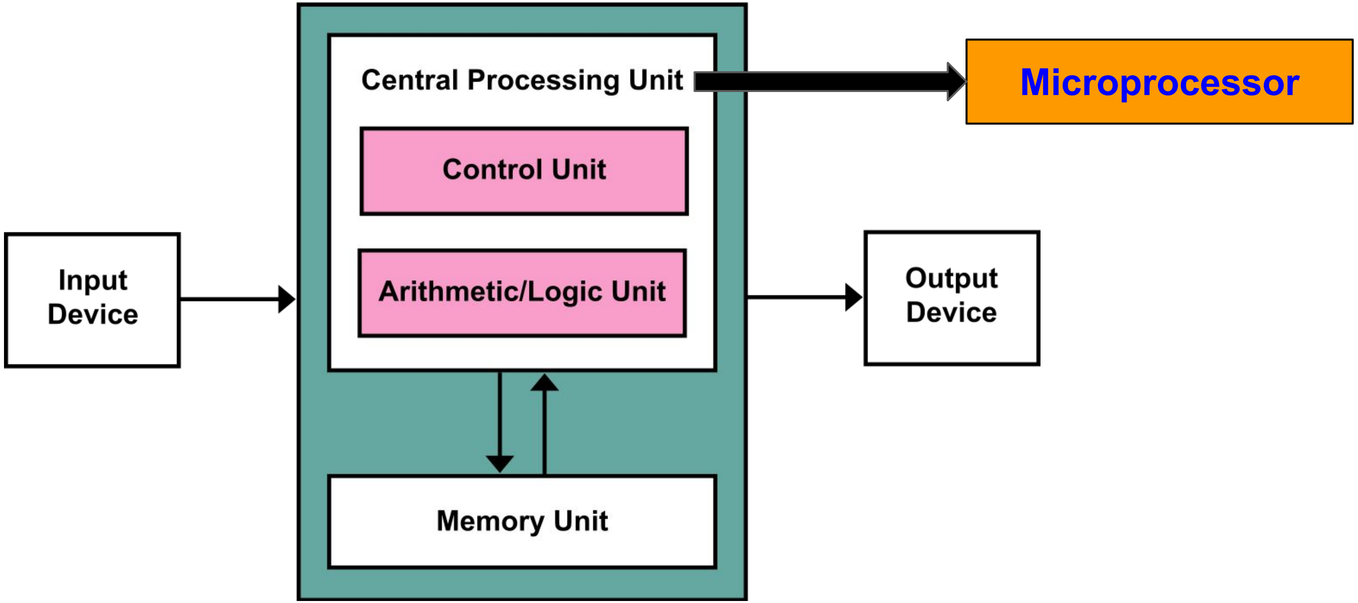
- It will help you to design **Compiler** and to understand few concepts of **OS** such as **Atomic Instruction, Mutex etc.**
- Finally, To understand the insight of the computer

Quiz

Which one is faster, Hardware or Software?

What is 8086?

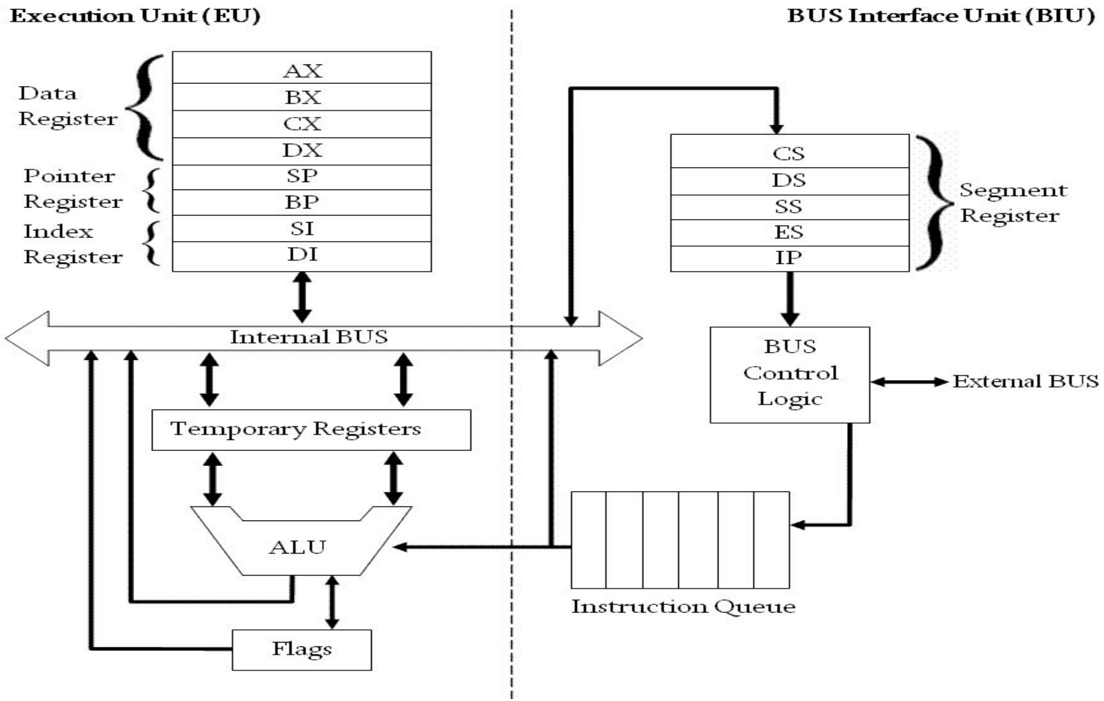
All CPUs are microprocessors.
And 80X86 are **microprocessor chips** designed by Intel.



Von Neumann Diagram

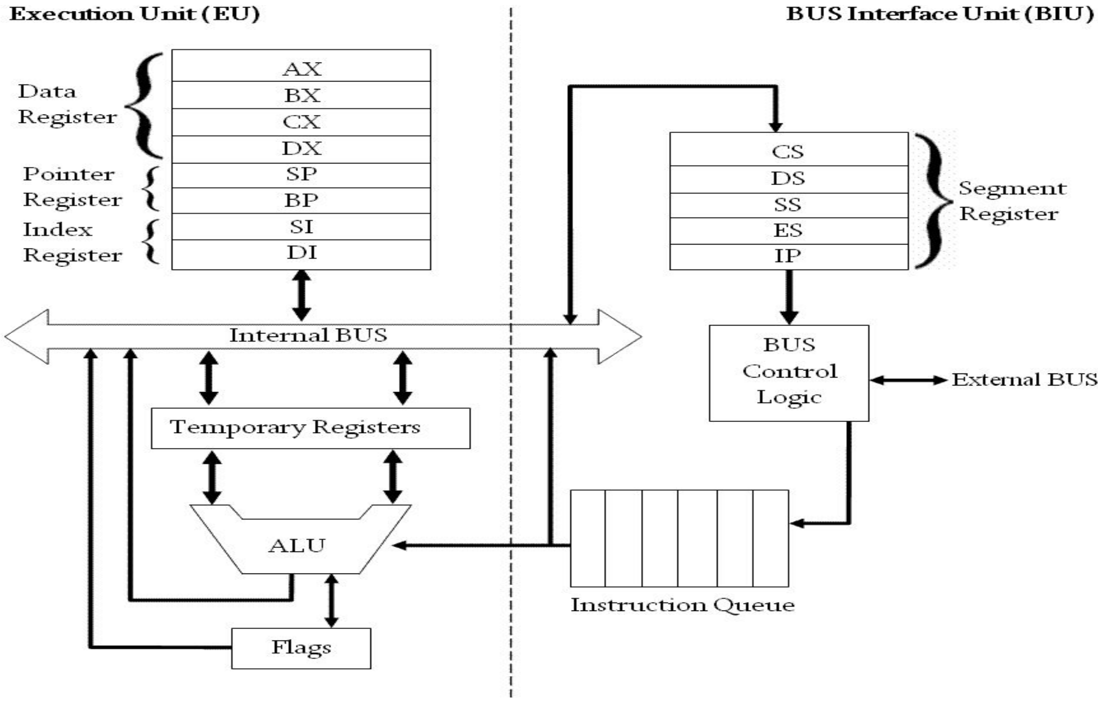
System Architecture: 8086 and Other Components

- 8086 CPU
  - **EU**
  - **BIU**
- Memory
  - RAM **(Instructions and Data)**
  - ROM (**Firmware**)
- Peripherals (I/O Devices)
  - Keyboard, Display, Disk drivers



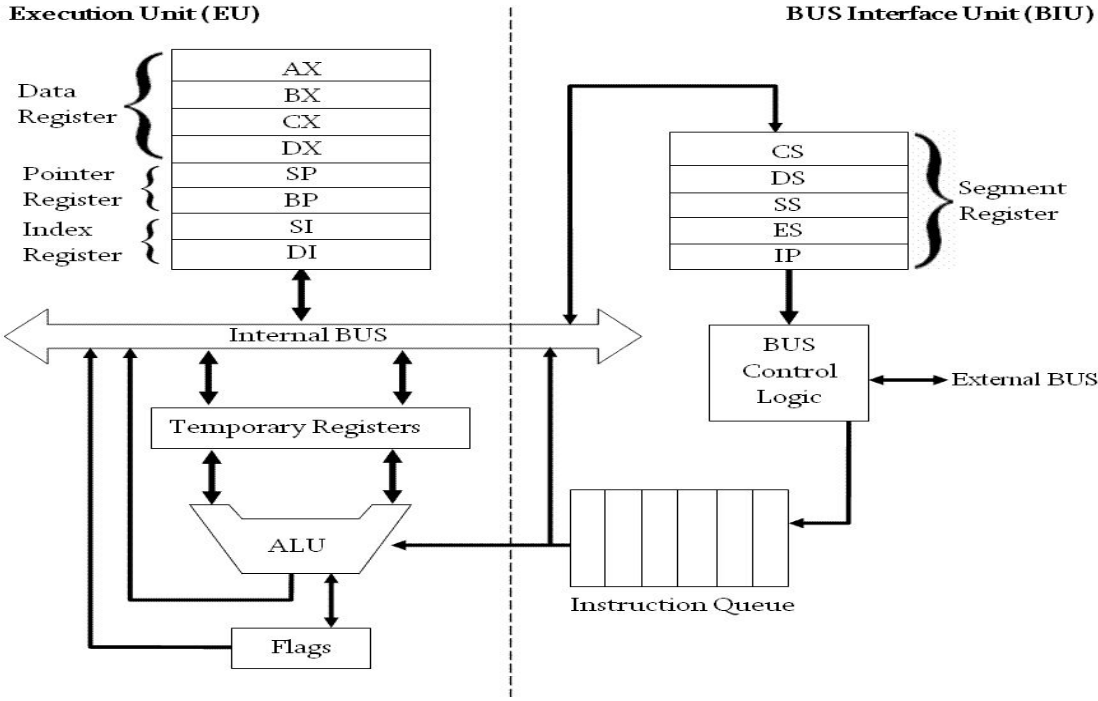
System Architecture: **8086**

- 8086 CPU
  - **EU**
    - **ALU circuit** to perform Arithmetic & Logical Instructions on general regs
    - **General regs** are used by ALU ckt
    - **Flags** is used for control flow or condition
  - **BIU**
    - Use to communicate between **EU** and **RAM** or **I/O**
    - **Have regs** to hold the address of memory location
- 8086 Properties
  - 16-bit **Microprocessor**
    - Can work on 16-bit at a time
  - 20-bit Address **Bus**
    - Can support 2^{20} memory bytes



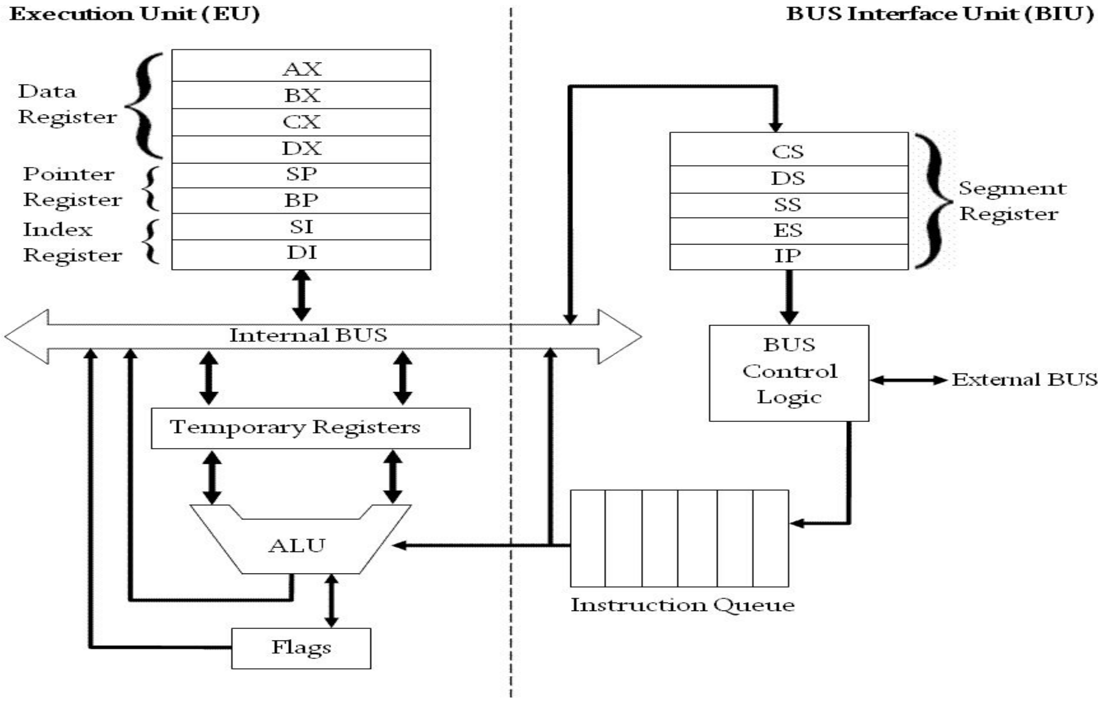
System Architecture: Instruction Execution

- Fetch
  - Bring instructions (Opcode + Operand) from Memory to CPU
- Decode
  - Determine the operation based on Opcode.
- Execute
  - Perform the operation & store the results



Registers

- Data Reg (4)
- Address Reg (9)
- Flag Reg (1)
- Each of them are **16 bit**
- Total 14 register



Program Structure

- Statement: Instruction to execute or Directives for the assembler
- Memory Models
    - .MODEL memory_model (SMALL)
- Data Segment
    - Declare variables and constants
- Stack Segment
    - .STACK size (100H)
- Code Segment
    - Define procedures
        - name PROC
        - name ENDP
    - MAIN refers to the main procedure

```
.MODEL SMALL
.STACK 100H
.DATA
.CODE

MAIN PROC

    MAIN ENDP
END MAIN
```

Emulator

- We will use emulator to see how the 8086 execute instructions line by line.

Link: Emu 8086

Any Question?

Assembly Language

Topics

- Recap
- Commonly Used Opcode
- High Level to Low Level Conversion
- Single Character Input and Output
- String Output
- **Array in Assembly**
- Flow Control Basic (JMP, JC, JNC, LOOP)
- **Array Traverse**

Book: [Assembly Language Programming and Organization of the IBM PC]

Topics

- Recap
- Basic of Assembly Language
- Commonly Used Opcode
- Emu 8086
- High Level to Low Level Conversion

Book: Assembly Language Programming and Organization of the IBM PC

Recap

Google Classroom

Link:

Code:

CSE 305 (4 credit)

- There will be 4 class per week.
- Three classes will be conducted with the normal CSE 305 (3 credit).
- Your **4th period** will be on
- And you need to remind me about your attendance (specially **4th period**).

Registers

- Type of fast memory
- Used to store and transfer data and instructions

Types of Registers

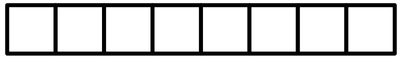
- 3 types
  - Data Register
  - Address Register
  - Status Register

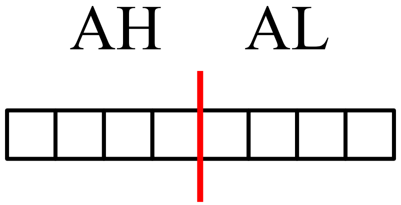
We will mainly focus on ***Data Register***

Data Register

- Holds data
- 16 bit
- 4 types:
    - AX: Accumulator Register
    - BX: Base Register
    - CX: Count Register
    - DX: Data Register

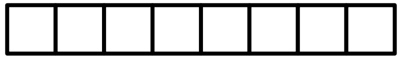
AX: Accumulator Register

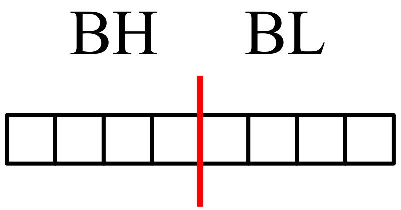
- 16 bit AX: 
- Upper and lower bytes can be used separately



- Special Purpose: Multiplication, Division, Input, Output

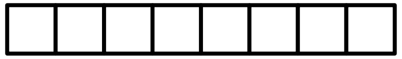
BX: Base Register

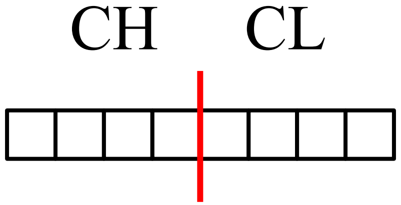
- 16 bit BX: 
- Upper and lower bytes can be used separately



- Special Purpose: Address register

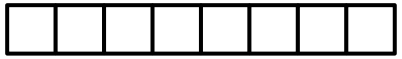
CX: Count Register

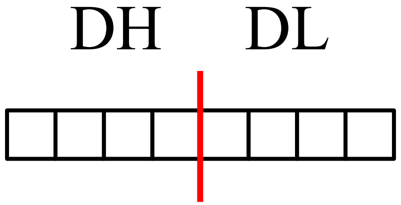
- 16 bit CX: 
- Upper and lower bytes can be used separately



- Special Purpose: Controlling loop

DX: Data Register

- 16 bit DX: 
- Upper and lower bytes can be used separately



- Special Purpose: Multiplication, Division, Output

Basic of Assembly Language

Statement of Assembly Language

Syntax

- Instruction
- Assembler Directive
-

Name Field	Operation Field	Operand Field	Comment Field

Program Data

- Numbers
- Characters

Variables and Constants

DB	DW	DD	DQ	DT	EQU

Format of Instruction

Name	Opcode	Operands	Comments
START:	Mov	CX, 4	; We don’t code in C, we code in Assembly

Represent the location of the Instruction.

Assembly Code

```
.MODEL SMALL; Small program
.STACK 100H    ; Declare Stack Size
.DATA          ; Data Segment Start
A DW 2
B DW 5
SUM DW ?
.CODE          ; Code Segment Start
MAIN PROC      ; Main Procedure Start
START: MOV CX, 4
MAIN ENDP      ; Main Procedure Start
```

Representing Numbers

- Binary Numbers
  - Must end with ***b***
  - Example: 10110111b
- Hexa Numbers
  - Must end with ***h***
  - Must start with ***digit***
  - Legal Number: 2Ah
  - Illegal Number: DAh
  - Legal Number: 0DAh
- Decimal Numbers
  - Can end with ***d***
  - Legal Number: 128d
  - Legal Number: 128

- Find the base of the following numbers
  - 10110111b **Binary**
  - 1Bh **Hexa**
  - 10110111 **Decimal**
  - 3B **Wrong Format**
  - 38 **Decimal**
  - EAh **Wrong Format** **Correct: 0EAh**

Registers and Data Types

Register	*16 bit*	*8 bit*
Accumulator	AX	AH, AL
Base	BX	BH, BL
Count	CX	CH, CL
Data	DX	DH, DL

- List of Registers

Data type	*Full form*	*Number of bits*
DB	Define Byte	8
DW	Define Word	16
DD	Define Double Word	32
DQ	Quad Word	64

- List of Data Types

Declaration of Variables

Variable Name	Data type	Initial Value
var1	db	5

- Must be initialized with a legal value
- Verify the following initializations:
  - Var1 db 320 **Illegal**
  - Var1 db 1001100110101b **Illegal**
  - Var1 db 0000000110101b **Legal**

Naming of Variables

- Can be combination of
  - Letter
  - Digit
  - Special Characters like: @ $? _
- Can not be started with digit and can not be consist of only digits
- Case insensitive
- Verify the following variable names:
  - 1Var db 127 **Illegal**
  - 123 db 127 **Illegal**
  - ? db 127 **Legal**
  - AbC$? db 127 **Legal**
  - @1 db 127 **Legal**

Emu 8086

Commonly Used Opcode

MOV

- MOV DEST, SRC
- Equivalent Task: DEST = SRC
- DEST = Registers, Variables
- SRC = Registers, Variables, Values
- Example: MOV CX, 8
- Equivalent Task: CX = 8
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time
- Verify the following instructions:
  - MOV AH, DL **;Legal** ; AH = DL
  - MOV var1, AL **;Legal if Var1 is DB** ; var1 = AL
  - MOV AX,Var1 **;Legal if Var1 is DW** ; AX = var1
  - MOV var1, var2 **;Illegal**
  - MOV AL,000000000000011b **;Legal**
  - MOV CL, 100000000000b **;Illegal**
  - MOV BH, 287 **;Illegal**
  - MOV var1, 125 **;Legal**

XCHG

- XCHG   ARG1, ARG2
- Equivalent Task: SWAP(ARG1, ARG2)
- ARG1 = Registers, Variables
- ARG2 = Registers, Variables
- Example: XCHG   AL, BH
- Equivalent Task: SWAP(AL, BH)
- Number of bits (ARG1) = Number of bits (ARG2)
- ARG1 and ARG2 can not be variables at a time
- Verify the following instructions:
  - XCHG AH, DL   **;Legal** ; SWAP(AH, DL)
  - XCHG var1, AL   **;Legal if Var1 is DB**   ; SWAP(var1, AL)
  - XCHG AX,30   **;Illegal**
  - XCHG var1, var2   **;Illegal**

ADD

- ADD   DEST, SRC
- Equivalent Task: DEST= DEST + SRC
- DEST = Registers, Variables
- SRC = Registers, Variables, Values
- Example: ADD   AL, BH
- Equivalent Task: AL = AL + BH
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time
- Verify the following instructions:
  - ADD AH, DL   **;Legal** ; AH=AH+DL
  - ADD var1, AL   **;Legal if Var1 is DB**   ; var1=var1+AL
  - ADD AX,Var1   **;Legal if Var1 is DW** ; AX=AX+var1
  - ADD var1, var2   **;Illegal**
  - ADD var1, 125 **;Legal** ; var1=var1+125
  - ADD ch, 125   **;Legal** ; ch=ch+125
  - ADD 320, al   **;Illegal**

SUB

- SUB DEST, SRC
- Equivalent Task: DEST= DEST - SRC
- DEST = Registers, Variables
- SRC = Registers, Variables, Values
- Example: SUB AL, BH
- Equivalent Task: AL = AL - BH
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time

NEG

- NEG   DEST
- DEST = Registers, Variables
- Example: NEG   AL
- Equivalent Task: DEST= - DEST
- Equivalent Task: AL = - AL

INC

- INC DEST
- DEST = Registers, Variables
- Example: INC AL

- Equivalent Task: DEST= DEST + 1
- Equivalent Task: AL = AL + 1

DEC

- DEC DEST
- DEST = Registers, Variables
- Example: DEC AL

- Equivalent Task: DEST= DEST - 1
- Equivalent Task: AL = AL - 1

Home Task: High Level to Low Level

- Write the instructions in Assembly Language and implement in Emu8086 for the following task:

  - A = B – 2*A [Consider A and B as db type variables]

Task: High Level to Low Level

- Write instructions in Assembly Language for the following task:
  - A = B – 2*A [Consider A and B as db type variables]

```
MOV  AL, A        ; AL = A
ADD  AL, AL       ; AL = 2*A
SUB  AL, B        ; AL = 2*A - B
MOV  A, AL        ; A = 2*A - B
NEG  A            ; A = B – 2*A
```

Single Character Input

- Required Instruction: INT 21H
- Set AH = 1 before calling INT 21H
- Input character will be stored at ***AL*** after calling INT 21H
- Value of AH works as Mode Selector here

```
MOV  AH,1
INT  21H
; Now use the input from AL
```

Single Character Output

- ❑ Required Instruction: INT 21H
- ❑ Set AH = 2 before calling INT 21H
- ❑ Store the output character at ***DL*** before calling INT 21H
- ❑ Value of AH works as Mode Selector here

```
; Set your required character at DL
MOV  AH,2
INT  21H
```

String Output

- Required Instruction: INT 21H
- Set AH = 9 before calling INT 21H
- Store the output string with a $ at the end as a DB type variable
- Move the variable at DX by LEA before calling INT 21H

```
STRING1  DB  ‘HELLO WORLD$’

MOV  AH,9
LEA  DX, STRING1
INT  21H
```

Point To Remember

- Value of AL gets changed after calling INT 21H
- So, if you have a desired value at AL, store it at other place before calling INT 21H

Input/ Output

Function	AH Register	Interrupt	Execution
Single Key Input	1	INT 21H	Input ASCII code in AL
Single Character Output	2		Display ASCII character from DL (AL get changed)
Character String Output	9		To display string

Arrays In Assembly

Same as variable: `NAME TYPE VALUES ;comma separated`

For character strings use `""` or `''`.

Can declare Numbers and Characters in one definition.

In string `"$"` means the end mark of the string (Like '\0' character)

LEA Instruction or **OFFSET** to load the address of the array into register.

Which registers can be used as address in 8086?

Use **SI, DI** or **BX** to traverse the array.

Note: To use Data Segment we need to initialize DS. Mov @DATA to DS, where @DATA refers to the segment number.

Arrays and Memory

```
B_ARRAY DB 10H,20H,30H
;If the assembler assigns the offset address 0200h to B_ARRAY
```

Symbol	Address	Contents
B_ARRAY	200h	10h
B_ARRAY+1	201h	20h
B_ARRAY+2	202h	30h

Arrays and Memory

```
W_ARRAY DW 1000, 40, 29887, 329
;If the assembler assigns the offset address 0200h to W_ARRAY
```

Symbol	Address	Contents
W_ARRAY	0300h	1000
W_ARRAY+2	0302h	40
W_ARRAY+4	0304h	29887
W_ARRAY+6	0306h	329

```
WORD1 DW 1234H
```

- Lower Part: WORD1 → 34H
- Higher Part: WORD1+1 → 12H

Arrays Traverse

```
.DATA           ; Data Segment Start
ARR DB 48, 49, 50, 51, 52
.CODE           ; Code Segment Start
MAIN PROC       ; Main Procedure Start
;Initialize DS
MOV AX, @DATA
MOV DS, AX
; Load BX with the Address of ARR
LEA BX, ARR     ; Or, MOV BX, OFFSET ARR
; For Traverse Need to Know the Flow Control
MAIN ENDP       ; Main Procedure Start
```

Arrays Traverse Using LOOP

```
;For Traverse Need to Know the Flow Control
MOV CX, 5    ; MOV CX, Number of element in array
PRINT:
   MOV DL, [BX]; Loading the value in BX location
   INC BX
   ;PRINT
   LOOP PRINT
```

Arrays Traverse Using Flow Control

```
;For Traverse Need to Know the Flow Control
MOV CX, 5    ; MOV CX, Number of element in array
PRINT:
   MOV DL, [BX]; Loading the value in BX location
   INC BX
   DEC CX
   ;PRINT
   JNZ PRINT
```

Flag Register of 8086 (16-bit)															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	**DF**	**IF**	**TF**	**SF**	**ZF**		**AF**		**PF**		**CF**

Flag Register

- When an Arithmetic or Logical Operation is performed the Flag Register is Updated.
- Example:

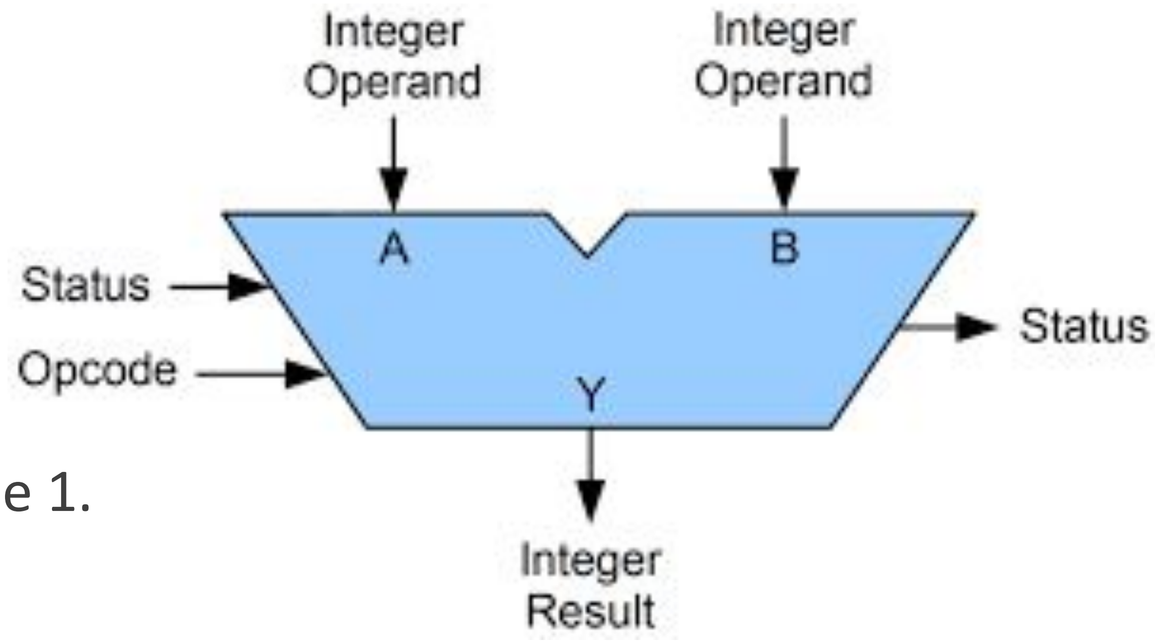
  Assume. AL = 05h and BL = 05h

  SUB AL, BL ; AL = AL - BL → AL = 0

  As, the **ALU circuit** results **0** then the **ZF** will be 1.

  If BL = 10h, then?

  SF will be 1.



Few Logical Opcode (We will Use in the Lab)

SHIFT LEFT

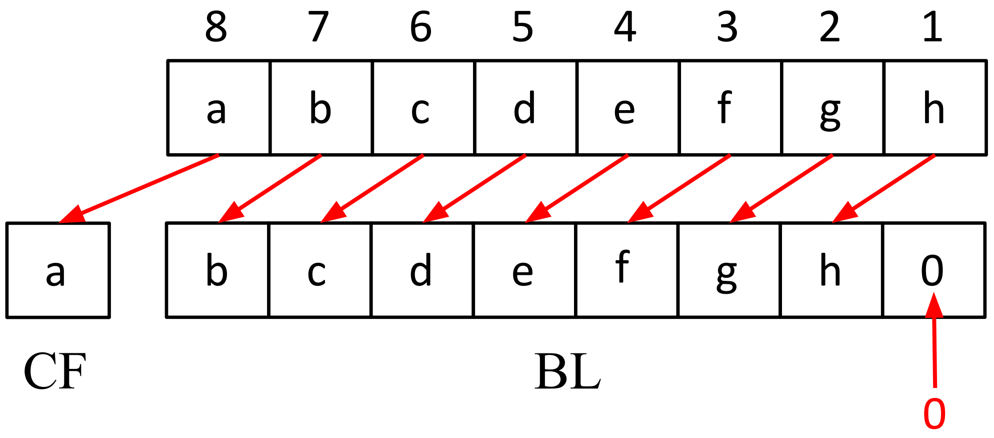
- Performs bitwise Left Shift operation
- Instruction format: SHL A, B
- Shifts A for B number of times at left
- Stores the MSB at Carry Flag (CF) after one left shift

- *Restrictions*:
- ***B must be value or CL***

- A **SHL** on a binary number will multiplies it by 2.

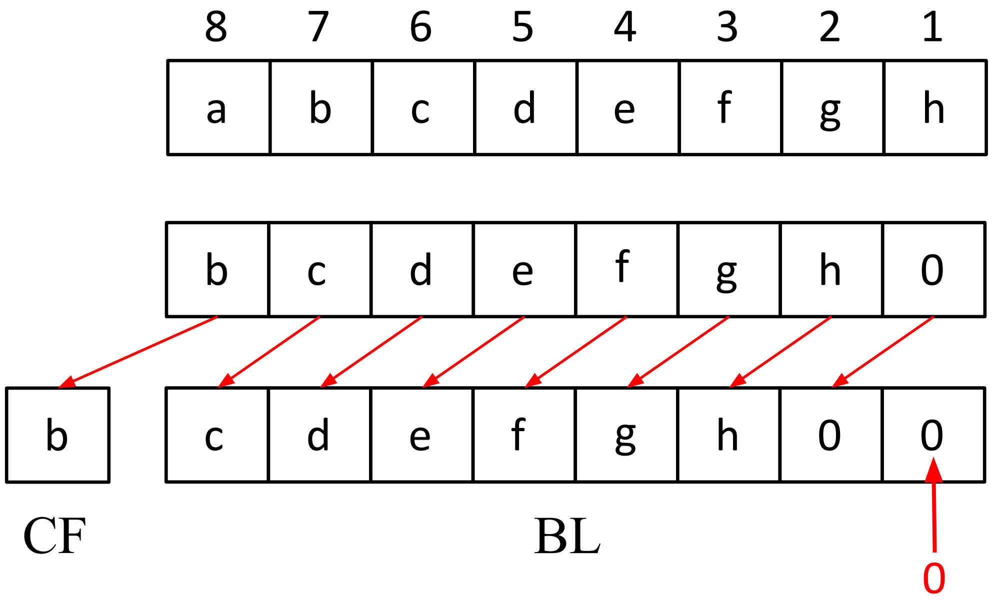
SHIFT LEFT

- Consider BL:
- SHL BL,1:



SHIFT LEFT

- Consider BL:
- SHL BL,1:
- Again SHL BL,1:



SHIFT LEFT

- Maximum how many Left Shift operation needed for setting a n-bit register to 0?
  - n
- Write a code that sets BL to 0 by Left Shift operation

First Way:

```
SHL BL,8
```

Second Way:

```
MOV CL,8
SHL BL,CL
```

Wrong Way:

```
MOV AL, 8
SHL BL, AL
```

- Write a code for printing the binary value of a 8 bit register/variable

SHIFT RIGHT

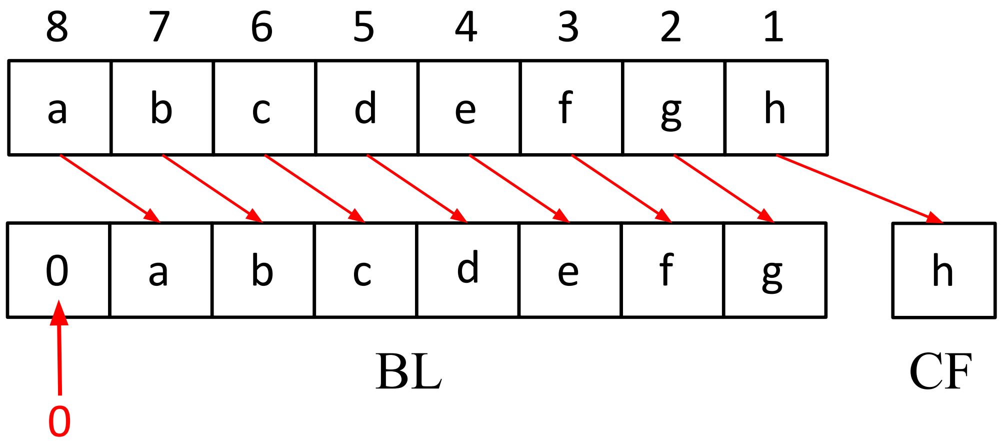
- Performs bitwise Right Shift operation
- Instruction format: SHR A, B
- Shifts A for B number of times at right
- Stores the LSB at Carry Flag (CF) after one right shift

- <u>*Restrictions*</u>:
- <u>***B must be value or CL***</u>

- A **SHR** on a positive binary number will divides it by 2.

SHIFT RIGHT

- Consider BL:
- SHR BL,1:



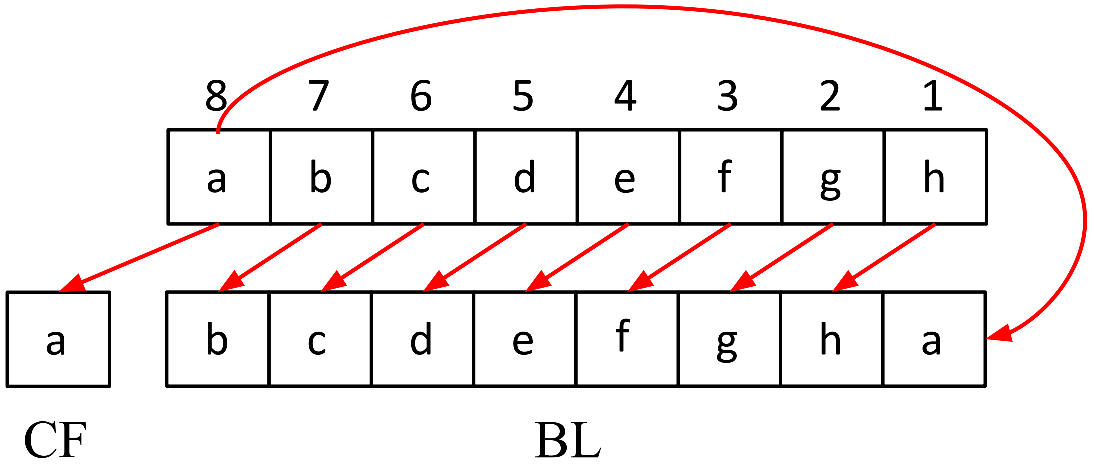
ROTATE LEFT

- Performs bitwise Left Rotate operation
- Instruction format: ROL A, B
- Rotate A for B number of times at left
- Stores the MSB at Carry Flag (CF) and at LSB after one left rotate

- *Restrictions*:
- ***B must be value or CL***

ROTATE LEFT

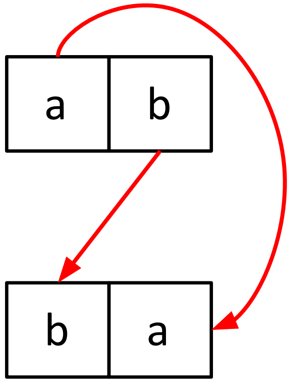
- Consider BL:
- ROL BL,1:



ROTATE LEFT

- Let X is a 2 bit register, Rotate it for 2 times (left/right), And then write down the observation

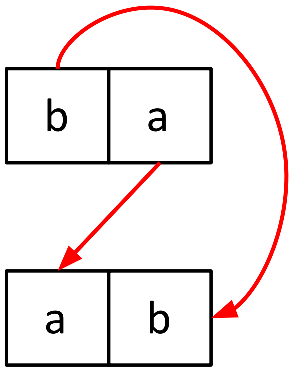
- A:
- ROL AL,1:



ROTATE LEFT

- Let X is a 2 bit register, Rotate it for 2 times (left/right), And then write down the observation

- A:
- ROL AL,1:
- ROL AL,1:



- The initial value gets restored if a n-bit register/memory is rotated(left/right) for n times

- Write down the advantages of printing the binary value of a register/variable using rotate operation rather than using shift operation.

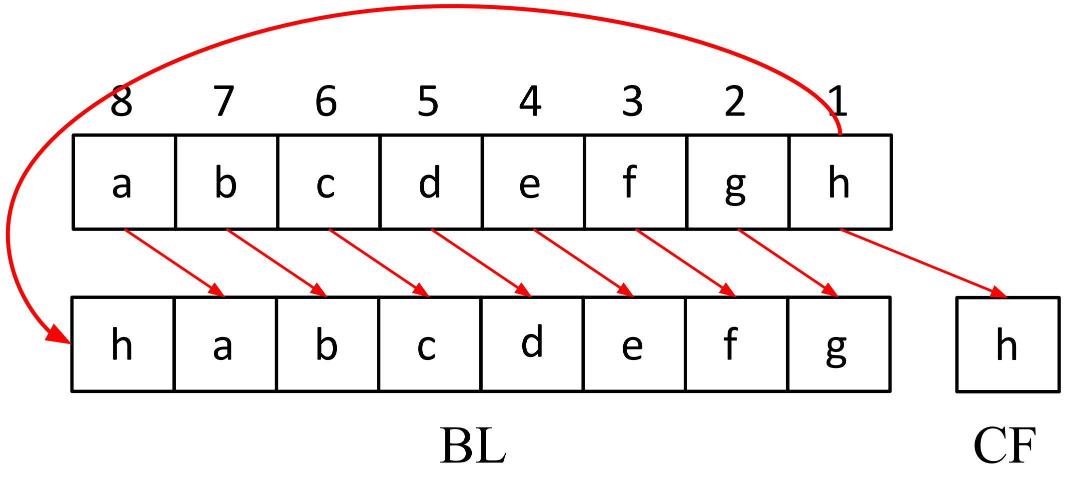
ROTATE RIGHT

- Performs bitwise Right Rotate operation
- Instruction format: ROR A, B
- Rotate A for B number of times at right
- Stores the LSB at Carry Flag (CF) and at MSB after one left rotate

- *Restrictions*:
- ***B must be value or CL***

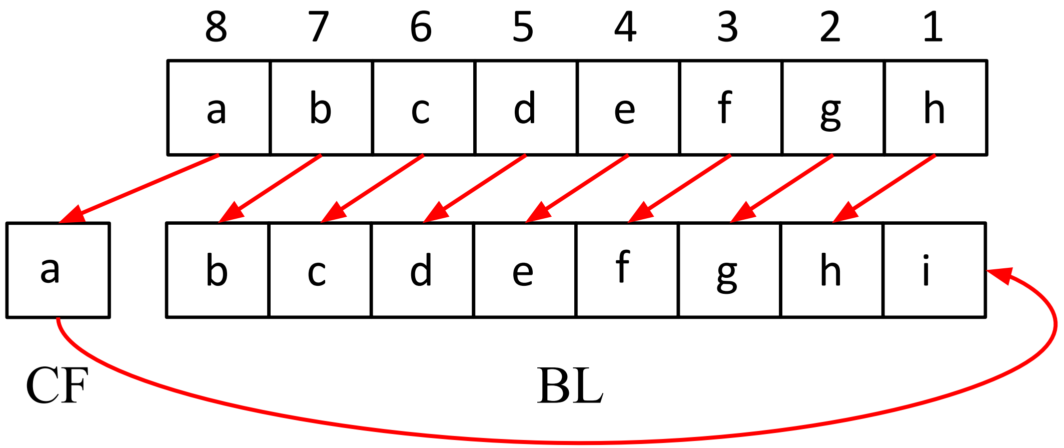
ROTATE RIGHT

- Consider BL:
- SHR BL,1:



ROTATE CARRY LEFT

- Consider BL:
- RCL BL,1:

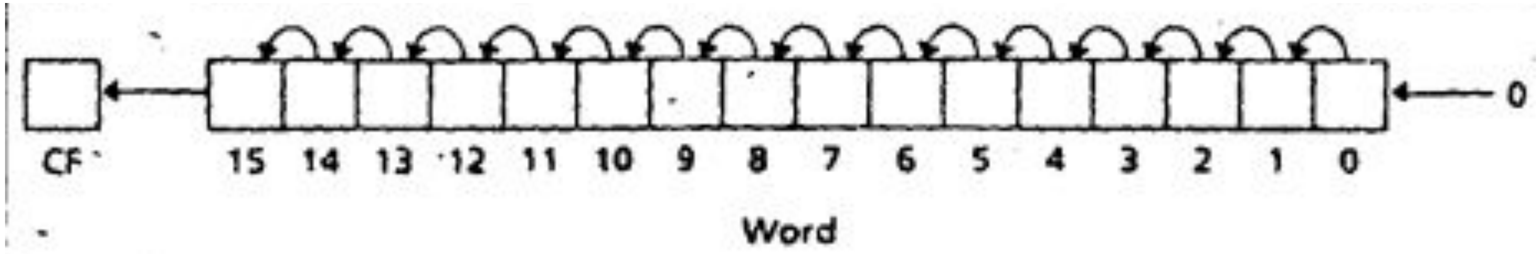


- ROTATE CARRY RIGHT will be in same way

Shift Left

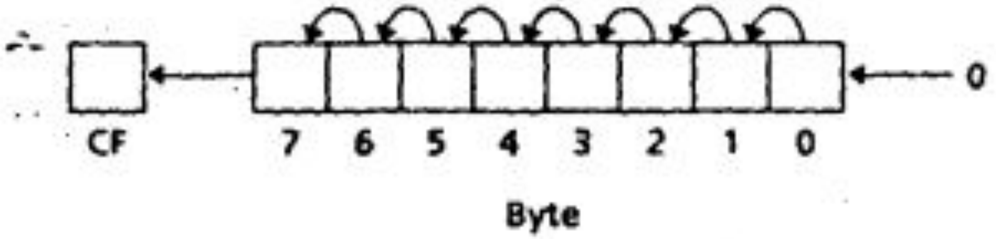
SHL destination,1

SHL destination, CL



SAL destination,1

SAL destination, CL



Shift Right

```
SHR destination,1

SHR destination, CL
```

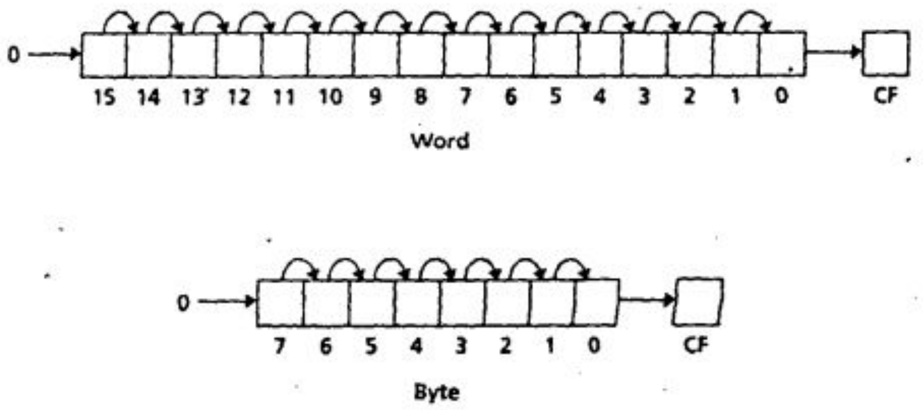


Fig: SHR

SAR `destination,1`

SAR `destination, CL`

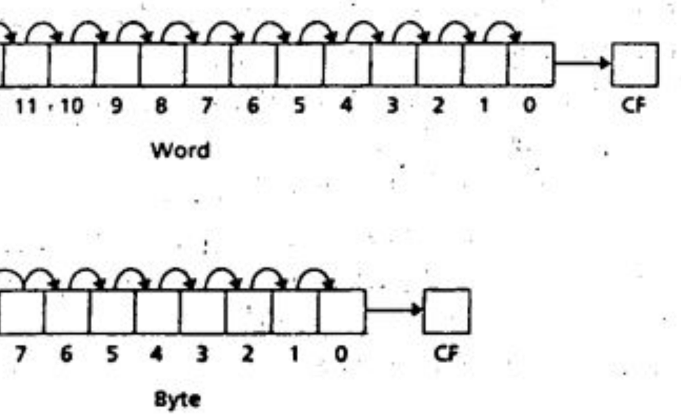


Fig: SAR

Use **Shift Arithmetic** in numeric Multiplication and Division.

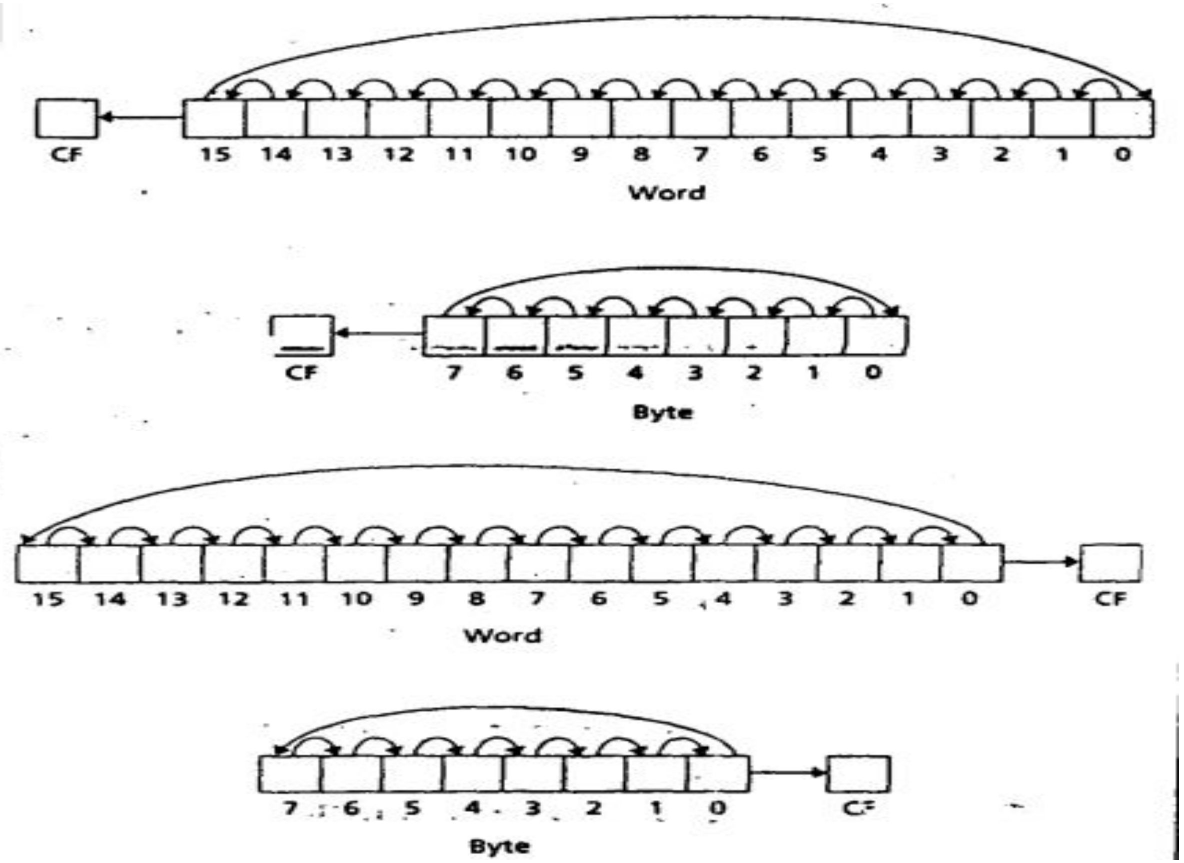
Rotate

ROL destination,1

ROL destination, CL

ROR destination,1

ROR destination, CL



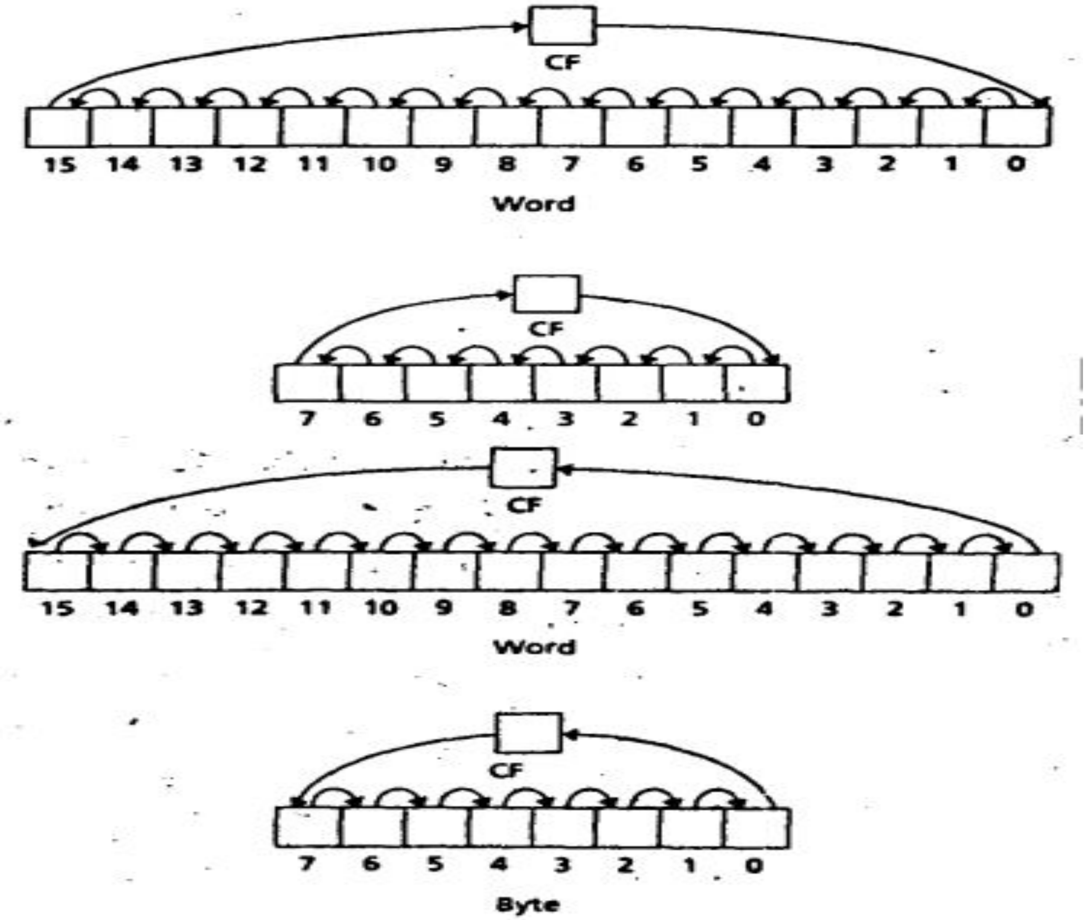
Rotate

RCL destination,1

RCL destination, CL

RCR destination,1

RCR destination, CL



Logical Operations

- AND
- OR
- TEST
- XOR
- SHL
- SHR
- ROL
- ROR
- RCL
- RCR

AND

- Performs bitwise AND operation
- Instruction format: AND DEST, SRC
- Equivalent operation: DEST = DEST AND SRC
- Example: AND AL, 3
- Equivalent operation: AL = AL AND 3

- *Restrictions*:
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time

AND

❑ *Working Principle*:

- Let AL = 189d

128	64	32	16	8	4	2	1
1	0	1	1	1	1	0	1

- Let BL = 0D3h

8	4	2	1	8	4	2	1
1	1	0	1	0	0	1	1

- AL AND BL

1	0	0	1	0	0	0	1

- AL in decimal 145d
- AL in Hexa 91h

OR

- Performs bitwise OR operation
- Instruction format: OR DEST, SRC
- Equivalent operation: DEST = DEST OR SRC
- Example: OR AL, 3
- Equivalent operation: AL = AL OR 3

- ***Restrictions***:
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time

OR

❑ ***Working Principle*:**

- Let AL = 213d

128	64	32	16	8	4	2	1
1	1	0	1	0	1	0	1

- Let BL = 9h

8	4	2	1	8	4	2	1
0	0	0	0	1	0	0	1

- AL OR BL

1	1	0	1	1	1	0	1

- AL in decimal   221d
- AL in Hexa   0DDh

XOR

- Performs bitwise XOR operation
- Instruction format: XOR DEST, SRC
- Equivalent operation: DEST = DEST XOR SRC
- Example: XOR AL, 3
- Equivalent operation: AL = AL XOR 3

- *Restrictions*:
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time

X	Y	F
0	0	0
1	0	1
0	1	1
1	1	0

- *Special Properties of XOR*:
- X XOR 0 = X
- X XOR 1 = $\overline{X}$

XOR

- ***Working Principle:***

- Let AL = 213d

128	64	32	16	8	4	2	1
1	1	0	1	0	1	0	1

- Let BL = 9h

8	4	2	1	8	4	2	1
0	0	0	0	1	0	0	1

- AL XOR BL

1	1	0	1	1	1	0	0

- AL in decimal 220d
- AL in Hexa 0DCh

SET A BIT

	8	7	6	5	4	3	2	1
▪ Consider AL:	a	b	c	d	e	f	g	h
▪ Bitwise OR with:	0	0	0	0	1	0	0	0
	a	b	c	d	1	f	g	h

- Set the 4^{th} bit of AL to 1 and keep other bits unchanged

RESET A BIT

- Set the 5th bit of AL to 0 and keep other bits unchanged

	8	7	6	5	4	3	2	1
Consider AL:	a	b	c	d	e	f	g	h
Bitwise AND with:	1	1	1	0	1	1	1	1
	a	b	c	0	e	f	g	h

COMPLEMENT/CHANGE A BIT

	8	7	6	5	4	3	2	1
▪ Consider AL:	a	b	c	d	e	f	g	h
▪ Bitwise XOR with:	0	0	0	1	0	0	0	0
	a	b	c	$\overline{d}$	e	f	g	h

- Complement the 5th bit of AL and keep other bits unchanged

COMPLEMENT A REGISTER

- Complement all the bits of AL

	8	7	6	5	4	3	2	1
▪ Consider AL:	a	b	c	d	e	f	g	h
▪ Bitwise XOR with:	1	1	1	1	1	1	1	1
	$\overline{a}$	$\overline{b}$	$\overline{c}$	$\overline{d}$	$\overline{e}$	$\overline{f}$	$\overline{g}$	$\overline{h}$

CMP (Like **SUB** Opcode)

- CMP DEST, SRC
- Equivalent Task: DEST - SRC while DEST unchanged
- DEST = Registers, Variables
- SRC = Registers, Variables, Values
- Example: CMP AL, BH
- Equivalent Task: AL - BH while AL unchanged
- Number of bits (SRC) = Number of bits (DEST)
- SRC and DEST can not be variables at a time
- As the ALU perform the subtraction, the **flags will be changed** according to the results.

CHECK ODD/EVEN

- Check whether the value stored in AL is even or odd

	8	7	6	5	4	3	2	1
▪ Consider AL:	a	b	c	d	e	f	g	h
▪ Bitwise AND with:	0	0	0	0	0	0	0	1
▪ AL:	0	0	0	0	0	0	0	h

- h is 0 if AL is even, h is 1 if AL is odd

- If h=1, then AL=1, means AL is odd
- If h=0, then AL=0, means AL is even

CMP AL,1
JZ ODD
JNZ EVEN

TEST (Like **AND** Opcode)

- Performs bitwise AND operation
- Instruction format: TEST A, B
- Difference with AND is it doesn't store the resultant value
- But it changes the value of the Flags
- If the resultant value is 0, it sets the Zero Flag to 1, else 0
- Equivalent operation: TEST A,B means if A AND B is 0 then Z=1, else Z=0

- *Restrictions*:
- Number of bits (A) = Number of bits (B)
- A and B can not be variables at a time

TEST EXAMPLE

	8	7	6	5	4	3	2	1
▪ Consider A:	0	1	1	0	1	1	0	1
▪ Consider B:	1	0	0	0	0	0	1	0
▪ A AND B:	0	0	0	0	0	0	0	0

- Result is 0
- So, Z will be set to 1

TEST EXAMPLE

	8	7	6	5	4	3	2	1
Consider A:	0	1	1	0	1	1	0	1
Consider B:	1	1	0	0	0	0	1	1
A AND B:	0	1	0	0	0	0	0	1

- Result is 65 (Not Zero)
- So, Z will be set to 0

CHECK A BIT

	8	7	6	5	4	3	2	1
▪ Consider AL:	a	b	c	d	e	f	g	h
▪ TEST AL with:	0	0	0	0	1	0	0	0
▪ TEST result:	0	0	0	0	e	0	0	0

- Check whether the 4^{th} bit of AL is 0 or 1
- If 4^{th} bit (e) is 1, then Test result is not Zero, means Z=0
- If 4^{th} bit (e) is 0, then Test result is Zero, means Z=1

```
TEST AL,00001000b
JZ Fourth_Bit_Is_Zero
JNZ Fourth_Bit_Is_One
```

CHECK ODD/EVEN

- Check whether the value stored in AL is even or odd

- Consider AL:

8	7	6	5	4	3	2	1
a	b	c	d	e	f	g	h

- TEST AL with:

0	0	0	0	0	0	0	1

- TEST Result:

0	0	0	0	0	0	0	h

```
TEST AL,1
JZ EVEN
JNZ ODD
```

Thank You

QUESTION?

Number Input Output

Topics

- Binary I/O
- Hex I/O

Book: Assembly Language Programming and Organization of the IBM PC

Binary Input-Output

Algorithm for Binary Input

- For binary Input, we assume a program reads in a binary number from the keyboard, followed by a **carriage return**.
- The number actually is a character string of 0's and 1's.
- As each character is entered, we need to convert it to a bit value, and insert the bit in a register.

Algorithm for Binary Input

Suppose Input: 1010

					Input
Before Itr 1	0	0	0	0	1
Itr 1	0	0	0	1	0
Itr 2	0	0	1	0	1
Itr 3	0	1	0	1	0
Itr 4	1	0	1	0	CR

Algorithm for Binary Input

```
Clear BX /* BX will hold binary value */
Input a character /* '0' or '1' */
WHILE character <> CR DO
     Convert character to binary value
     Left shift BX
     Insert value into LSB of BX
     Input a character
END_WHILE
```

Algorithm for Binary Output

Suppose Output: 1010

					CF
Before Itr	1	0	1	0	X
Itr 1	0	1	0	1	1
Itr 2	1	0	1	0	0
Itr 3	0	1	0	1	1
Itr 4	1	0	1	0	0

Algorithm for Binary Output

```
FOR 16 times DO
     Rotate left BX /* BX holds output value, put msb into CF
     */
     IF CF = 1
     THEN
          Output '1'
     ELSE
          Output ‘0’
     END_IF
END_FOR
```

Hex Input-Output

Algorithm for Hex Input

- Hex input consists of digits ("0" to "9") and letters ("A" to "F'”) followed by a carriage return. For simplicity, we assume that:
  - only uppercase letters are used, and
  - the user inputs no more than four hex characters.
- Unlike Binary input, need to shift the register four times.

Algorithm for Hex Input

```
Clear BX /* BX will hold input value */
Input a hex character /* '0' tO '9' or 'A' tO 'F'*/
WHILE character <> CR DO
     Convert character to binary value
     Left shift BX 4 times
     Insert value into lower 4 bits of BX
     Input a character
END_WHlLE
```

Algorithm for Hex Output

```
FOR 4 times DO
     Mov BH to DL /* "BX holds output value */
     Shift DL 4 times to the right
     IF DL < 10
     THEN
          Convert to character in '0' to '9
     ELSE
          Convert to-character in 'A' to 'F'
     END_IF
     Output character
     Rotate BX left 4 times
END FOR
```

Thank You

QUESTION?

Stack & Procedure

Objectives

- Stack Operation
- Terminology of Procedures
- CALL and RET

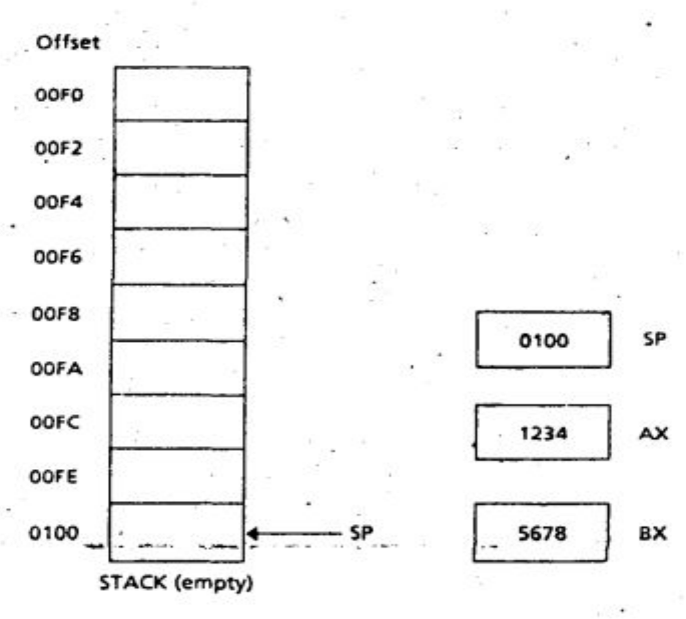
Book

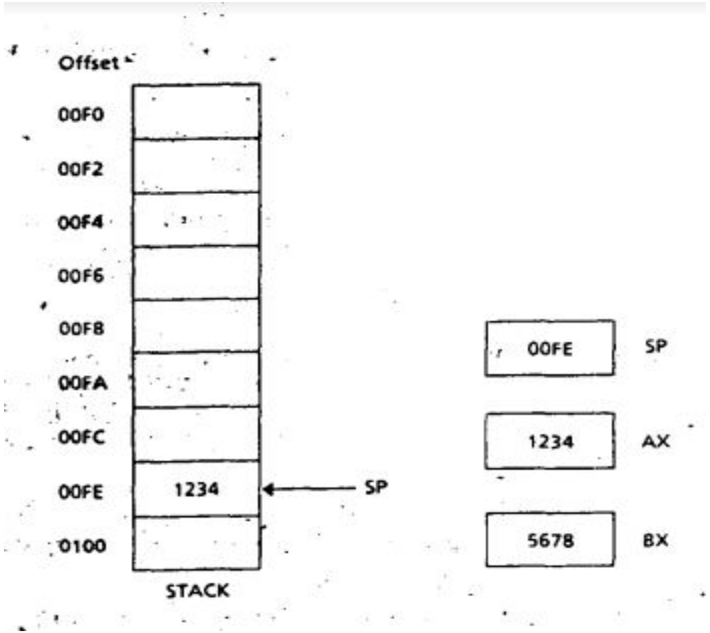
Stack Operation: PUSH

- **Syntax:** `PUSH SOURCE` ; **16-bit** register or memory word.
- SP is decreased by **2**.
- A copy of the source content is moved to the address specified by SS:SP. The source is unchanged.

Stack Operation: PUSH

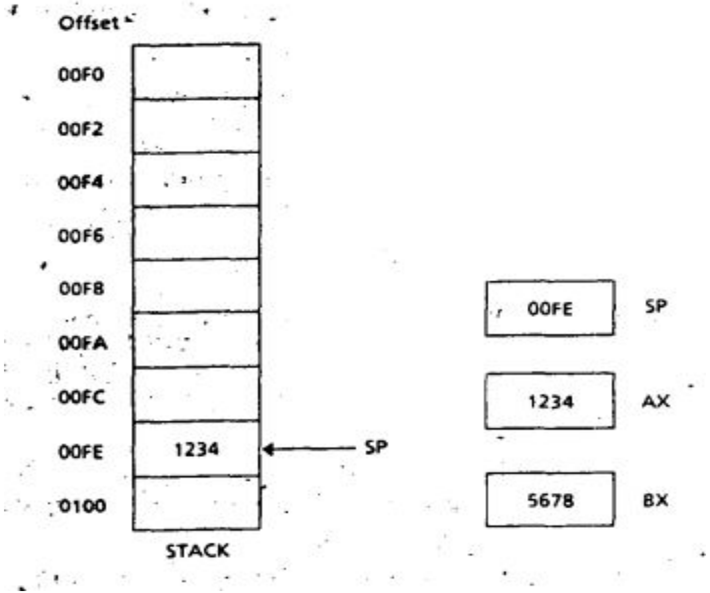
Instruction: **PUSH AX**

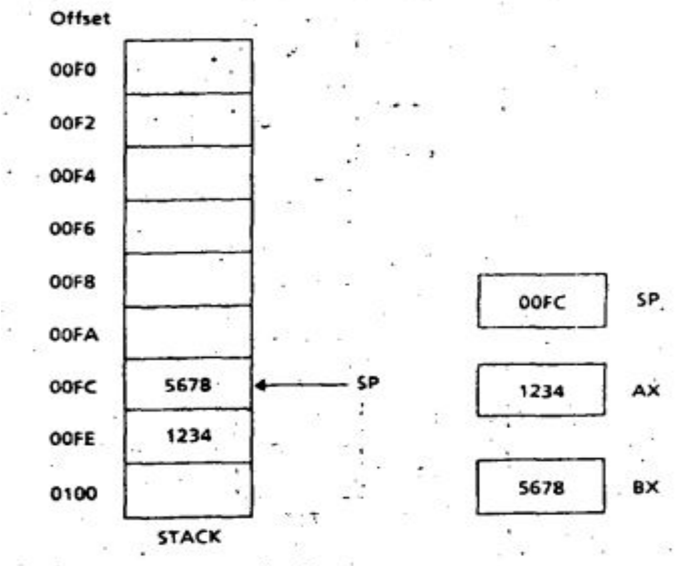




Stack Operation: PUSH

Instruction: **PUSH BX**



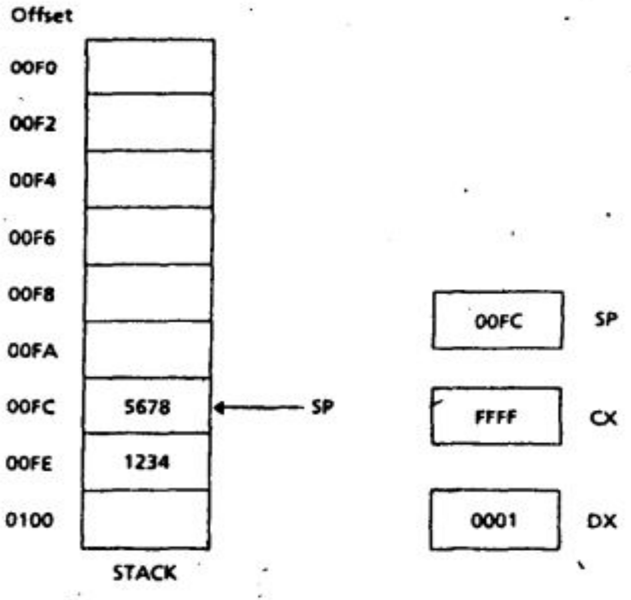


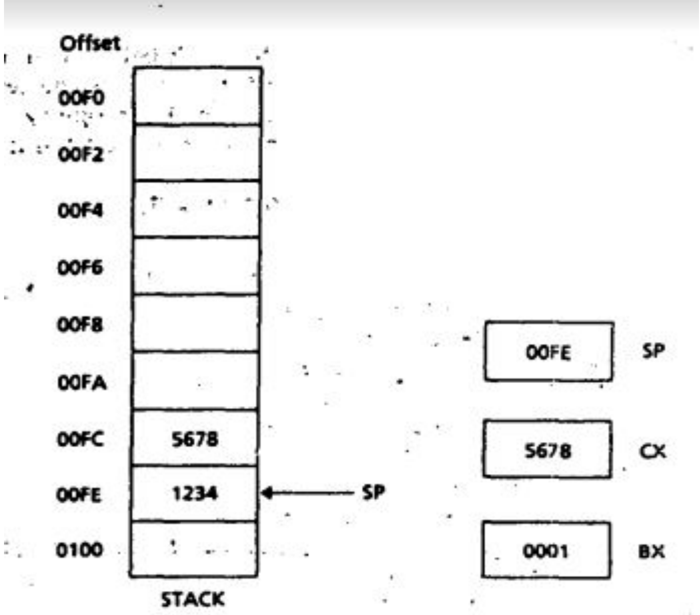
Stack Operation: POP

- **Syntax:** `POP DESTINATION` ; **16**-bit register (except IP) or memory word.
- The content of SS:SP (the top of the stack) is moved to the destination.
- SP is increased by **2**.

Stack Operation: POP

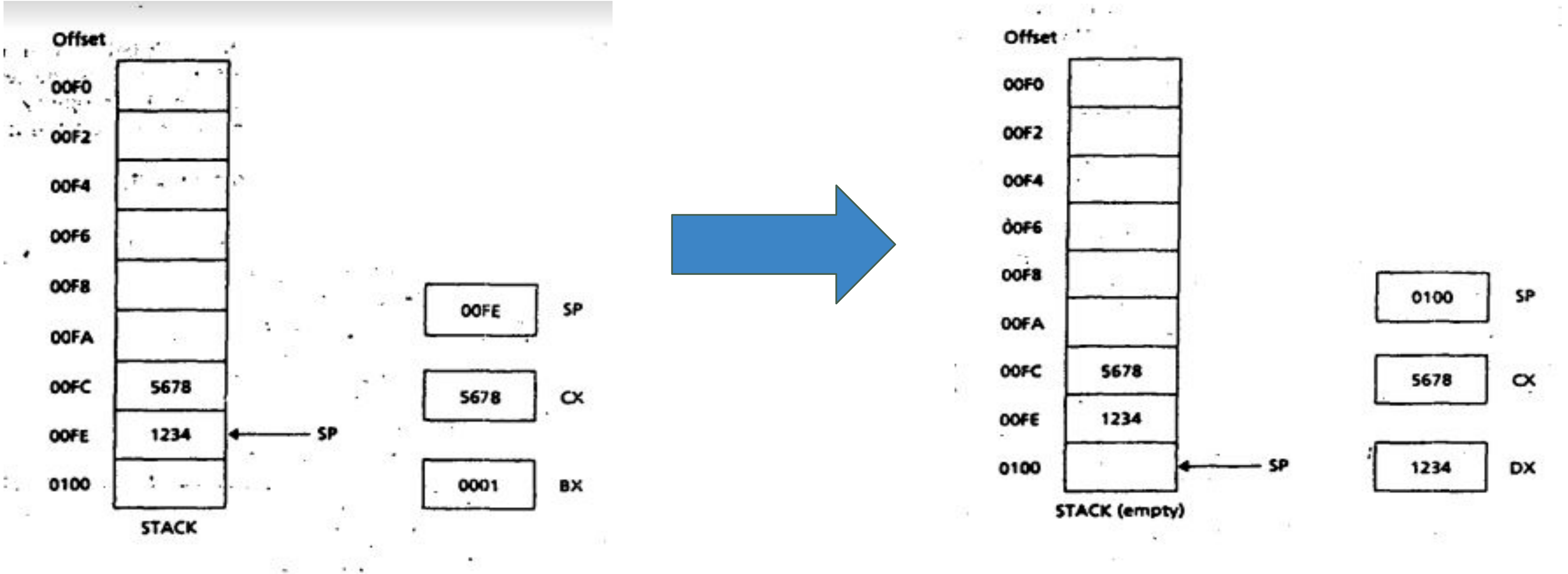
Instruction: `POP CX`





Stack Operation: POP

Instruction: `POP DX`



Stack Operation: Application

Algorithm to Reverse Input.

```
Display a '?'
Initialize count to 0
Read a character
WHILE character is not a carriage return DO
    push character onto the stack
    increment count
    read a character
END WHILE;
Go to a new line
FOR count times DO
    pop a character from the stack;
    display it;
END FOR
```

Stack Operation: More

- `PUSHF`
- `POPF`
- Applications??

Procedure Declaration

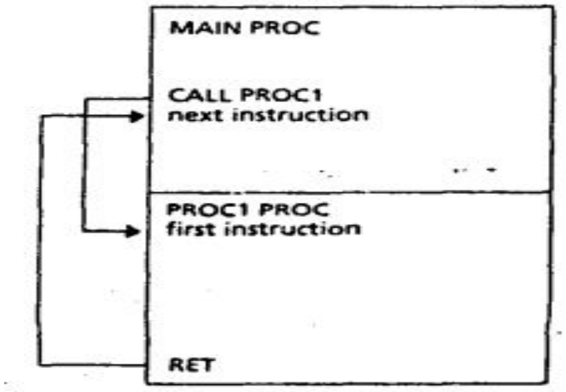
- `name` is the user-defined name of the procedure.
- The optional operand ***type*** is *NEAR* or *FAR* (if type is omitted, NEAR is assumed).
- *NEAR* means that the statement that calls the procedure is in the same segment as the procedure itself.
- *FAR* means that the calling statement is in a different segment.
- Assembly language procedures does not have parameter lists, so it's up to the programmer to devise a way for procedures to communicate. For example, if there are only a few input and output values, they can be placed in registers.

```
name PROC type
;body of the procedure
     RET
name ENDP
```

CALL and RET

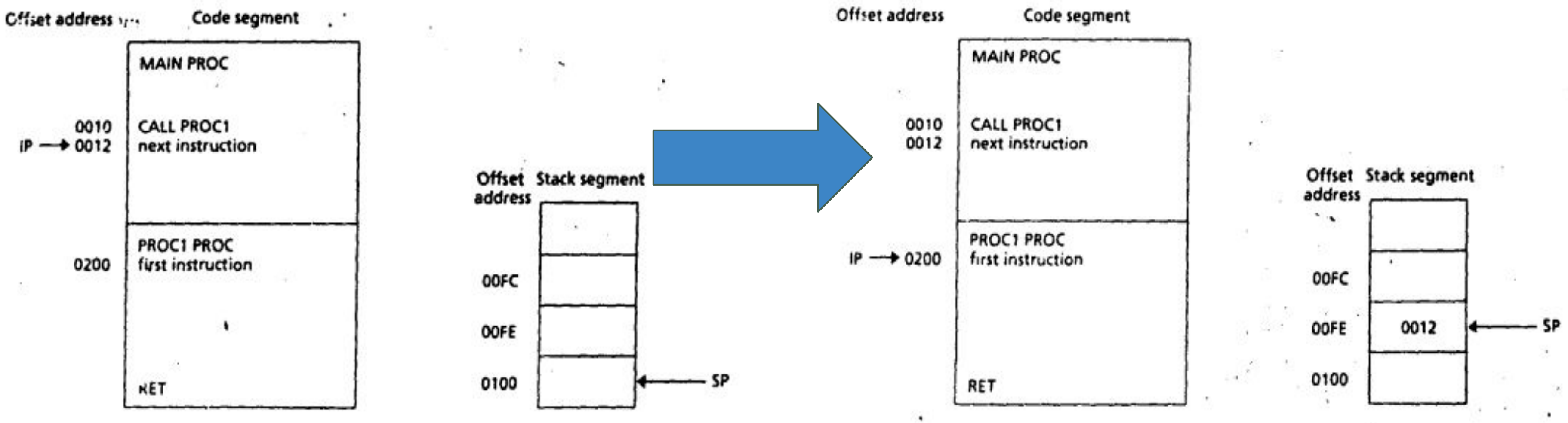
- To invoke a procedure, the CALL instruction is used.
  - `CALL name`
  - `CALL address_exprcssion`

- The **RET** (return) instruction causes control to transfer back to the calling procedure. Every procedure (except the main procedure) should have a RET someplace; usually it's the last statement in the procedure.



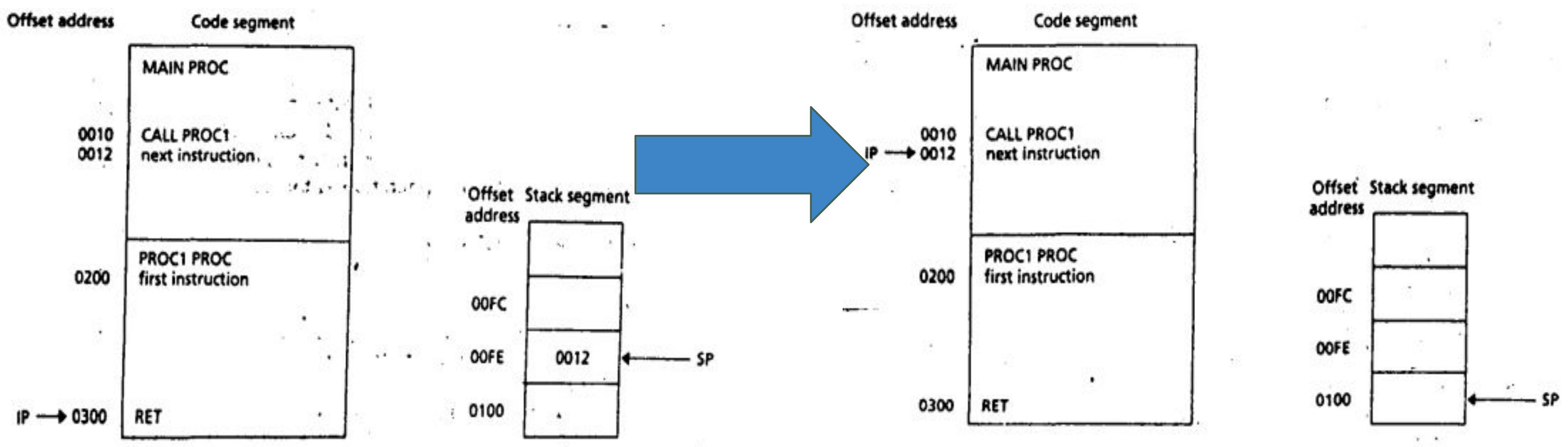
CALL and RET

- `CALL name`
  - Store the return address in stack.
  - IP changes to the address of the called procedure.



CALL and RET

➢ **RET**
- POP out from the stack to IP·



Thank You

QUESTION?

Decimal Input Output

Objectives

- Multiplication Instructions
- Division Instructions
- Decimal I/O

Book

Multiplication Instructions

- Syntax: `Opcode Source`
- `Opcode`
  - "MUL" for unsigned multiplication
  - "IMUL" for signed multiplication
- `Source`
  - Byte (8 bit) => `AX = AL * Source`
  - Word (16 bit) => `DX:AX = AX * Source`

Effect of MUL/ IMUL on the status flags	
SF, ZF, AF, PF	Undefined
CF/ OF:	
After MUL	= 0 if the upper half of the result is zero.
	= 1 otherwise.
After IMUL	= 0 if the upper half of the result is the sign extension of the lower half (this means that the bits of the upper half are the same as the sign bit of the lower half).
	= 1 otherwise.

- If the lower destination(AL or AX) can not store the result then CF/OF is 1.

Multiplication Instructions: Examples

Book

1. Suppose AX contains 1 and BX contains FFFFh. → MUL BX, IMUL BX
2. Suppose AX contains FFFFh and BX contains FFFFh. → MUL BX, IMUL BX
3. Suppose AX contains 0FFFh. → MUL AX, IMUL AX
4. Suppose AX contains 0100h and CX contains FFFFh. → MUL CX, IMUL CX
5. Suppose AL contains 80h and BL contains FFh. → MUL BL, IMUL BL

What will be the value of the registers, CF and OF Flags after the **MUL** and **IMUL** instructions.

Class Assessment

Convert the following High Level Instruction to Assembly Language using IMUL (suppose there is no overflow):

A = 5 x A - 12 x B

```
MOV AX, 5        ;AX m 5
IMUL A           ;AX = 5 x A
MOV A, AX        ;A = 5 x A
MOV AX, 12       ;AX = 12
IMUL B           ;AX = 12 x B
SUB A, AX        ;A = 5 x A - 12 x B
```

MUL Operation: Application

Algorithm for Factorial.

```
FACTORIAL PROC
;computes N!  ·
;input: CX = N
;output: AX = N!
    MOV AX,l
TOP:·
    MUL CX
    LOOP TOP
    RET
FACTORIAL ENDP
```

Division Instructions

- Syntax: **`Opcode Divisor`**
- **`Opcode`**
  - “DIV” for unsigned division
  - “IDIV” for signed division
- **`Divisor`**
  - Byte (8 bit) => `AL = AX / Divisor` **and** `AH = AX % Divisor`
  - Word (16 bit) => `AX = DX:AX / Divisor` **and** `DX = DX:AX % Divisor`

Division Instructions: Examples

Book

1. Suppose DX contains 0000h, AX contains 0005h, and BX contains 0002h. → DIV BX, IDIV BX
2. Suppose DX contains 0000h, AX contains 0005h, and BX contains FFFEh. → DIV BX, IDIV BX
3. Suppose DX contains FFFFh, AX contains FFFBh, and BX contains 0002h. → DIV BX, IDIV BX
4. Suppose AX contains 00FBh and BL contains FFh. → DIV BL, IDIV BL

What will be the value of the registers after the **DIV** and **IDIV** instructions.

Sign Extension of the Dividend

Word Division	**Byte Division**
For DIV, DX should be cleared.	For DIV, AH should be cleared.
For IDIV, DX should be made the sign extension of AX. The instruction **CWD** (convert word to doubleword) will do the extension.	For IDIV, AH should be made the sign extension of AL. The instruction **CBW** (convert byte to word) will do the extension.

Class Assessment

Convert the following High Level Instruction to Assembly Language using IMUL (suppose there is no overflow):

A = -1250 / B

```
MOV AX,-1250      ;AX gets dividend
CWD               ;Extend sign to DX
MOV BX,7          ;BX has divisor
IDIV BX           ;AX gets quotient, DX has remainder
```

Decimal Input: Algorithm

```
total = 0
read an ASCII digit
REPEAT
    convert character to a binary value
    total = 10 x total + value
    read a character
UNTIL character is a carriage return
```

Decimal Input

Input: 123

Initialize, total = 0			
Iter=0	read ‘1’	convert ‘1’ to 1	total = 10 x 0 + 1 = 1
Iter=1	read ‘2’	convert ‘2’ to 2	total = 10 x 1 + 2 = 12
Iter=2	read ‘3’	convert ‘3’ to 3	total = 10 x 12 + 3 = 123

Decimal Output: Algorithm

```
IF AX < 0 /• AX hclds output value •/
THEN
     print a minus sign
     replace AX by its two's complement
END IF
Get the digits in AX's decimal representation
Convert these digits to characters and print Them
```

Decimal Output: Algorithm

```
Get the digits in AX' s decimal representation
count = 0 /• will count decimal digits •/
REPEAT
     divide quotient by 10
     push remainder on the stack
     count = count + 1
UNTIL quotient = 0
```

```
Convert these digits to characters and print Them
FOR count times DO
     pop a digit from the stack
     convert it to a character
     output the character
END_FOR
```

Decimal Output

Output: 24618

Initially, AX = 24618			
Iter=0	Divide 24618 by 10	Quotient= 2461	remainder= 8
Iter=1	Divide 2461 by 10	Quotient= 246	remainder= 1
Iter=2	Divide 246 by 10	Quotient= 24	remainder= 6
Iter=3	Divide 24 by 10	Quotient= 2	remainder= 4
Iter=4	Divide 2 by 10	Quotient= 0	remainder= 2

Thank You

QUESTION?

Week: 09-12

Micro-controllers

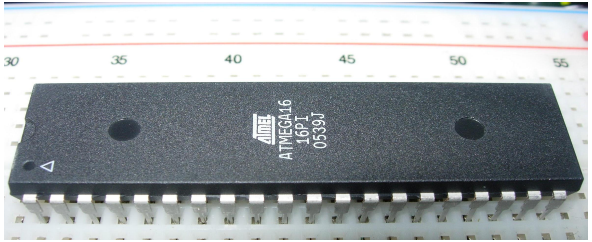
Prepared By:
Lec Abrar,
Dept of CSE, MIST
Email: abrar@cse.mist.ac.bd

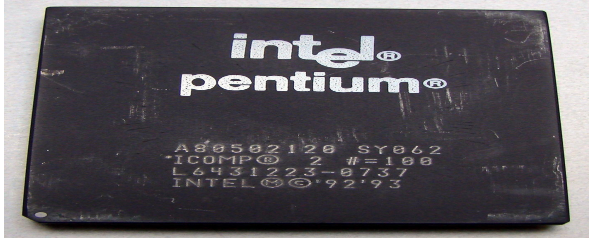
Topics-10th Week

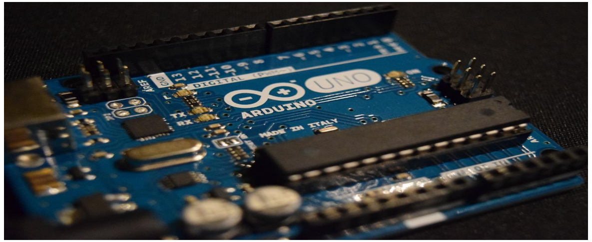
- Microprocessors, Micro-controllers
- Arduino
- Pin Diagram

Book: https://drive.google.com/file/d/1Z_1xdJuqq9HiUi07VAzyjcTnhyGeqib7/view

Microprocessors, Micro-controllers



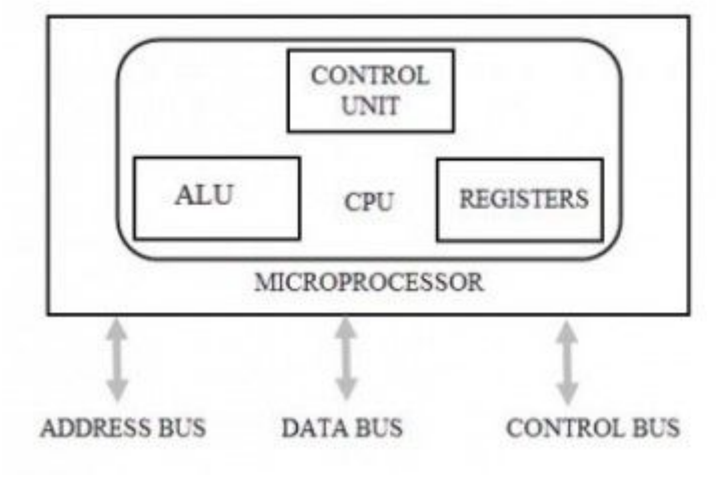




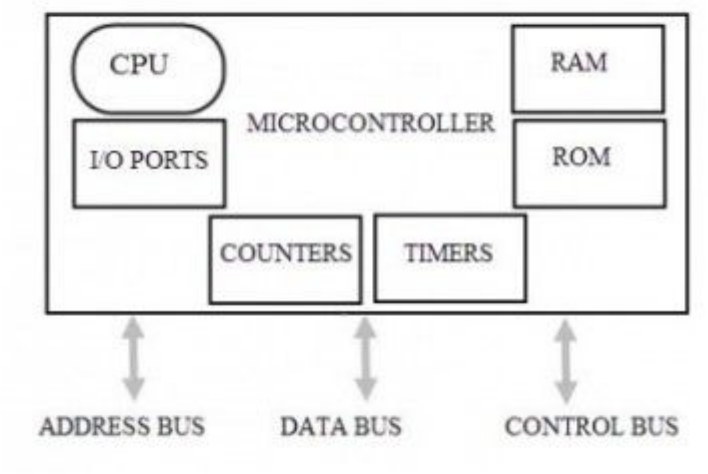
Microprocessors

Micro-controllers

Microprocessors, Micro-controllers



Microprocessors



Micro-controllers

Microprocessors, Micro-controllers

- Microprocessor consists of only a CPU and Microcontroller contains a CPU, Memory, I/O all integrated into one chip.
- Microprocessor is used in PCs and Microcontroller is used in an embedded system.
- Microprocessor is complicated and expensive but Microcontroller is inexpensive and straightforward.

Proteus

- Proteus is a electrical suite for circuit simulation purposes.
- Proteus Download Link: https://drive.google.com/file/d/1aQ-QefxwPn7Coc0X_xKwAKFt6r3PQLiD/view
- Wokwi: https://wokwi.com/
- Arduino IDE: https://downloads.arduino.cc/arduino-nightly-windows.zip
- Arduino Uno datasheet: https://www.farnell.com/datasheets/1682209.pdf
- Arduino Uno figure: https://drive.google.com/file/d/1Vt29W6dWGlfW3rGKwDYeUG2LrjclE05N/view?usp=sharing
- Functions: https://www.arduino.cc/reference/en/

Arduino Uno

- Arduino is an open-source hardware and software company that designs and manufactures single-board microcontrollers and microcontroller kits for building digital devices.
- The Arduino Uno is a microcontroller board based on the ATmega328.
- A ceramic resonator generates the ATmega328P's clock signal, though this oscillator features less precision than its crystal-based cousin.

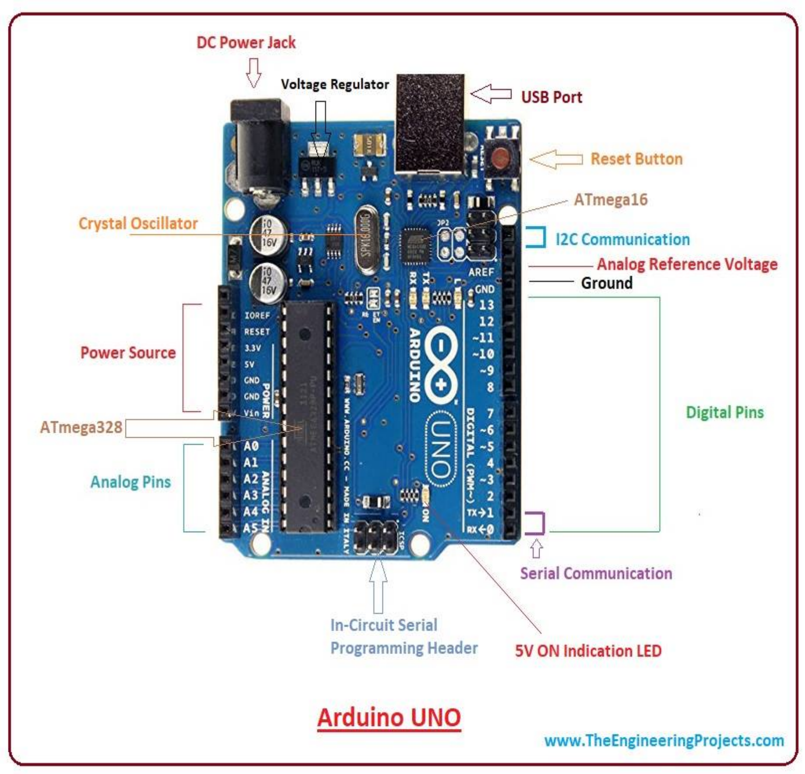
Few Sensors

- Analog Sensors
  - Resistance-based sensors
    - LDR (Light-dependent resistor)
    - RTD (Resistance-based Temperature Detectors)
      - P/NTC (Positive/ Negative Temperature Calibration)
    - Position Sensors → Joystick
  - Voltage-based sensors
    - LM35 temperature sensor (The voltage difference is proportional to the temperature.)
  - Current-based sensors
    - Phototransistors/ Photodiode

Arduino Uno

Pin Diagram

- 14 digital input/output pins (of which 6 can be used as PWM outputs)
- 6 analog inputs (also can be used as digital I/O)
- One UART interface found on pin 0 (RX0) and pin 1 (TX0)
- One I2C/TWI module
- And a SPI bus



Arduino Uno Input-Output

- Digital I/O
  - Pins: 0 to 13 + Analog Pins
  - Req: pinMode(pin, mode)
    - mode: INPUT/ OUTPUT
  - Input: digitalRead(pin)
  - Output: digitalWrite(pin, value)
  - Examples:
  - https://wokwi.com/projects/335733385937814098
  - https://wokwi.com/projects/335736198446187091

Arduino Uno Input-Output

- Analog Input
  - Pins: A0 to A5
  - Technique: Use 10 bits ADC
  - Input: analogRead(pin)
    - Return value: 0-1023 (10 bits)
  - analogReference(type);
    - DEFAULT: On Board (5 or 3.3V)
    - EXTERNAL: Applied on AREF pin (0 to 5V only)
  - Discrete value to Analog Value
    - analogVoltage = (digitalValue * V_{ref}) / 1023.00;
    - analogVoltage = map(digitalValue, fromLow, fromHigh, toLow, toHigh);
      - fromLow = 0, fromHigh = 1023
      - toLow = 0, toHigh = V_{ref}
  - Example: https://wokwi.com/projects/335731558488998483

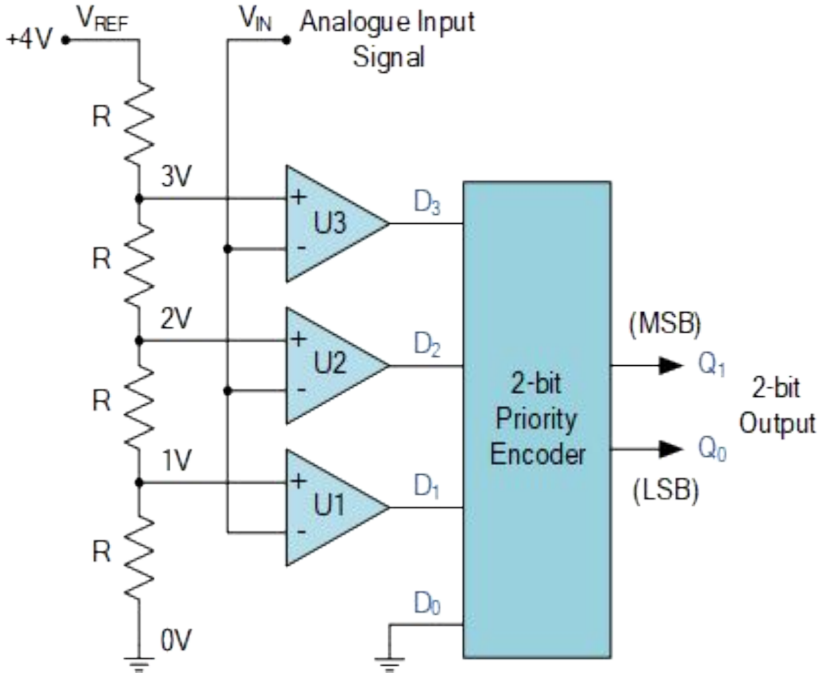
FLASH ADC

Analog Inputs: 2-bit Analog to Digital Converter Circuit

- Analog signals can be continuous and provide an infinite number different voltage values.
- Digital circuits on the other hand work with binary signal which have only two discrete states, a logic “1” (HIGH) or a logic “0” (LOW).
- Analog signals come from various sources and sensors which can measure sound, light, temperature or movement.
- Many actuators may need analog signals for controlling purposes like motor speed.

Analog Inputs: 2-bit Analogue to Digital Converter Circuit

Comparator Outputs				Digital Outputs	
D_3	D_2	D_1	D_0	Q_1	Q_0
0	0	0	0	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1



2-bit ADC will give digital value from 0 to 2^2 - 1 = 3, total 4 values. Hence for 10-bit it is 1024(0 to 1023).

Analog Inputs: Designing n-bit Flash ADC

- Requirements for n-bit Flash ADC:
  - Number of Comparators: $\mathbf{2^n - 1}$
  - Number of Resistance: $\mathbf{2^n}$
  - $\mathbf{2^n}$ to **n** Priority Encoder
  - $\mathbf{V_{ref}}$ and $\mathbf{V_{in}}$
  - Compare $\mathbf{V_{in}}$ and $\mathbf{V_{xR}}$
    - $\mathbf{V_{in}}$ is Analog Input Signal
    - $\mathbf{V_{xR}}$ is the voltage after xth resistance

Analog Inputs: Digital Output value Calculation

- **ADC Resolution (1 LSB) = V_{ref} / (2^n)**
  - The smallest incremental voltage that can be recognized.
- **Digital value = (V_{in} / V_{ref}) * (2^n – 1)**
- **Aout = (Digital value * V_{ref}) / (2^n – 1)**

Where,

Vref - The reference voltage is the maximum value that the ADC can convert.[Uno has a **Vref pin**]

To keep things simple, let us consider that Vref is 5V,

For 0 Vin, digital o/p value = 0

For 5 Vin, digital o/p value = 1023 (10-bit)

Analog Inputs: Problems

Design a 3-bit flash ADC using the followings:

- Resistors
- Comparators
- Priority Encoder

For both 3-bit and 10-bit ADC calculates the followings (Suppose Vref is 5v):

- ADC Resolution (1 LSB)
- Digital Value when Vin = 3.3
- Aout from the digital value

Analog Inputs: Temperature Sensor (LM35)

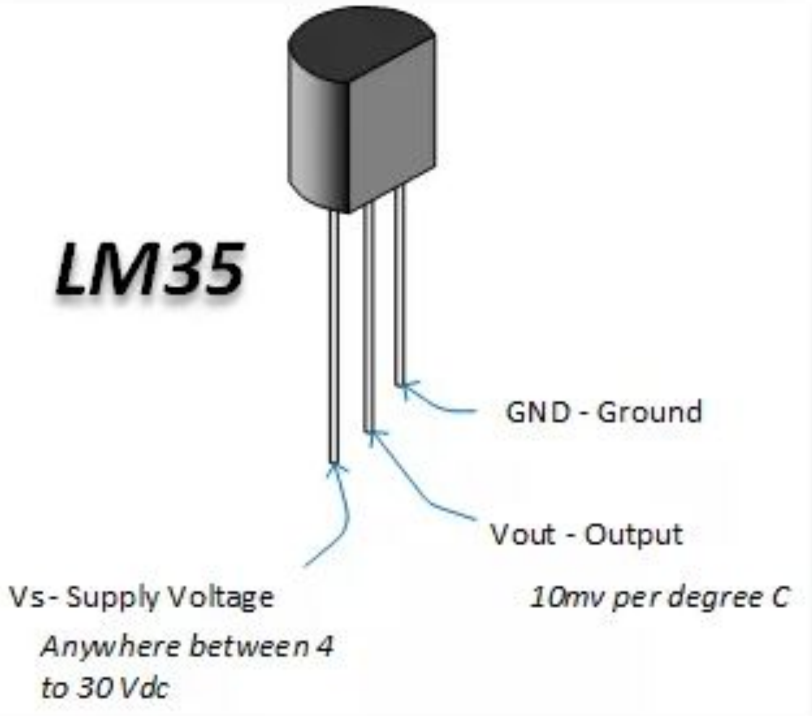
It's a temperature device with an output voltage linearly proportional to the Centigrade temperature. Temperature range: −55°C to 150° C. The sensitivity of LM35 is 10 mV/degree Celsius.

Pin Connection:

- First pin(Vs) on the left to 4-30V power,
- Second pin(Vout) to any analog input pins,
- Third pin(GND) to the ground

Formula,

- temp = (1/ 2.048) * analogRead(Vout)

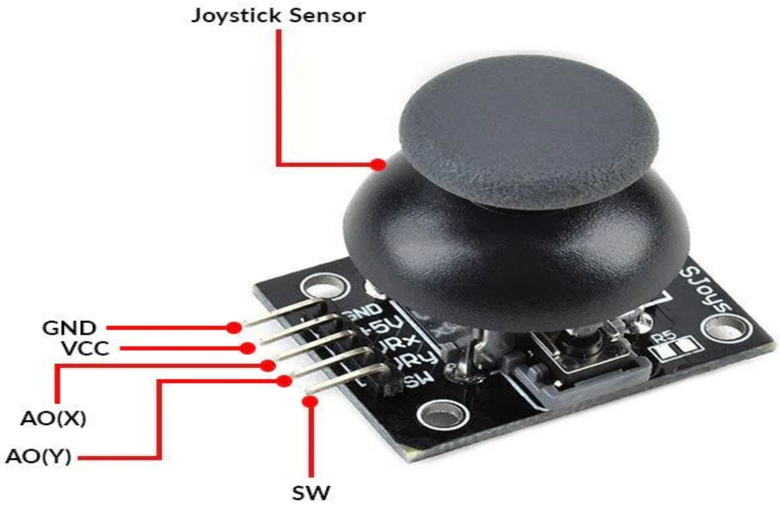


Analog Inputs: Analog Joystick

- Joystick is an input device. It is used to control the pointer movement in 2-dimension axis.
- The joystick has two potentiometers to read user's input.
- One potentiometer is used to get the analog output voltage for X-Direction movement and the other potentiometer is used to get the analog output voltage for Y-Direction movement.
- At Idle position output voltage, will be VCC/2.

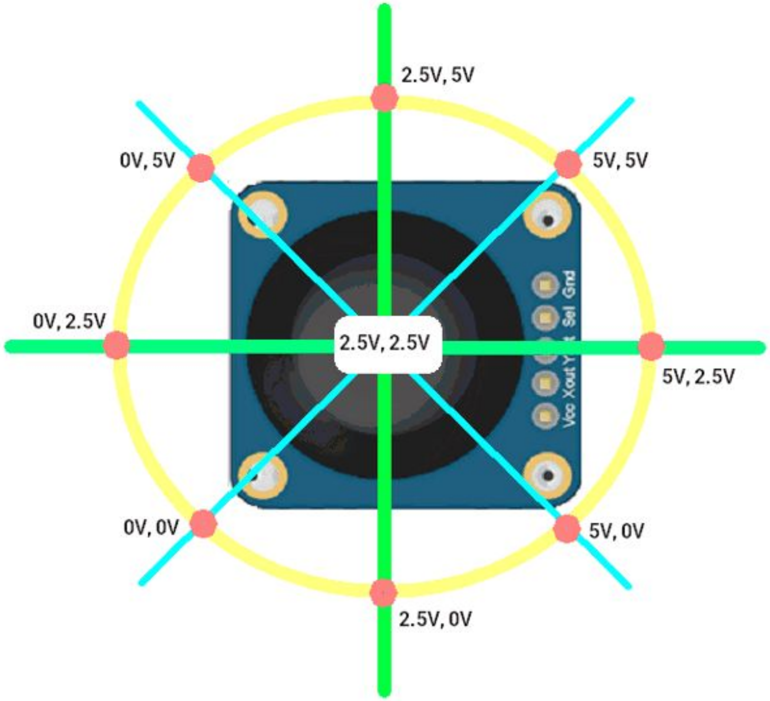
Pin Connection:

- VCC: First pin(Vs) on the left to 4-30V power,
- GND: Ground
- AO(X): A0 to A5
- AO(Y): A0 to A5



Analog Inputs: Analog Joystick

- Formula:
  - Left:
  - Right:
  - Up:
  - Down:
- Example: https://wokwi.com/projects/335776852784185938



Arduino Uno Input-Output

- Analog Output
  - Pins: 3, 5, 6, 9, 10, 11
  - Technique: PWM
  - Output: analogWrite(pin, value)
    - value: the duty cycle: between 0 (always off) and 255 (always on). Allowed data types: int.
    - PWM voltage=(Duty cycle ÷ 255) x 5 V.
  - Used to control the intensity of the transducer like led, motor.
  - Example: https://wokwi.com/projects/335737308028338770

Pulse Width Modulation

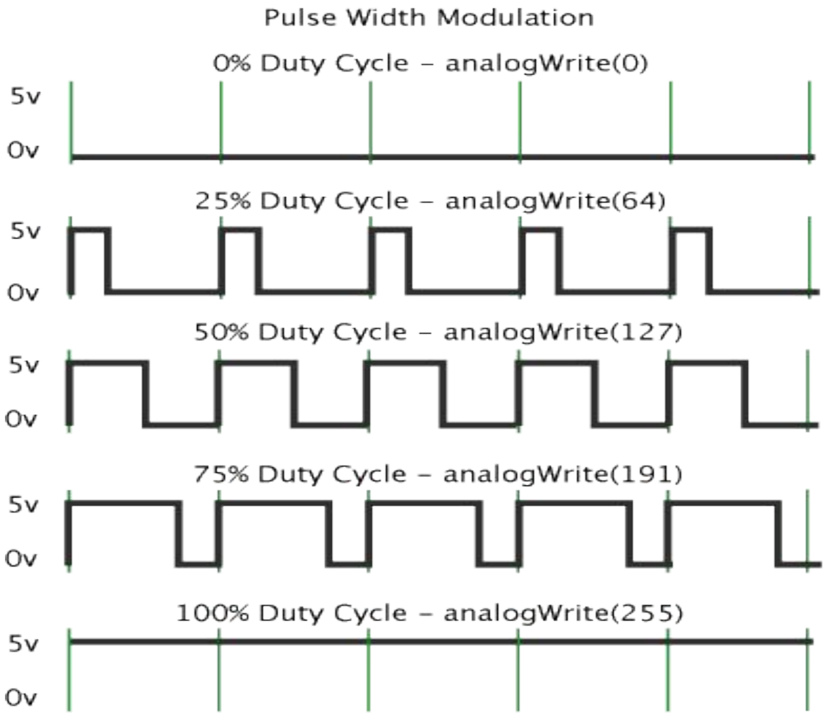
Analog Outputs: PWM (Pulse Width Modulation) ‘~’

Arduino does not have in-board DAC. Instead, it uses PWM(‘~’).

Can be used in:

- **Controlling the brightness of LED**
- Speed control of DC motor
- Controlling a servo motor or
- Where you have to get analog output with digital means

PWM voltage=(Duty cycle ÷ 255) x 5 V.

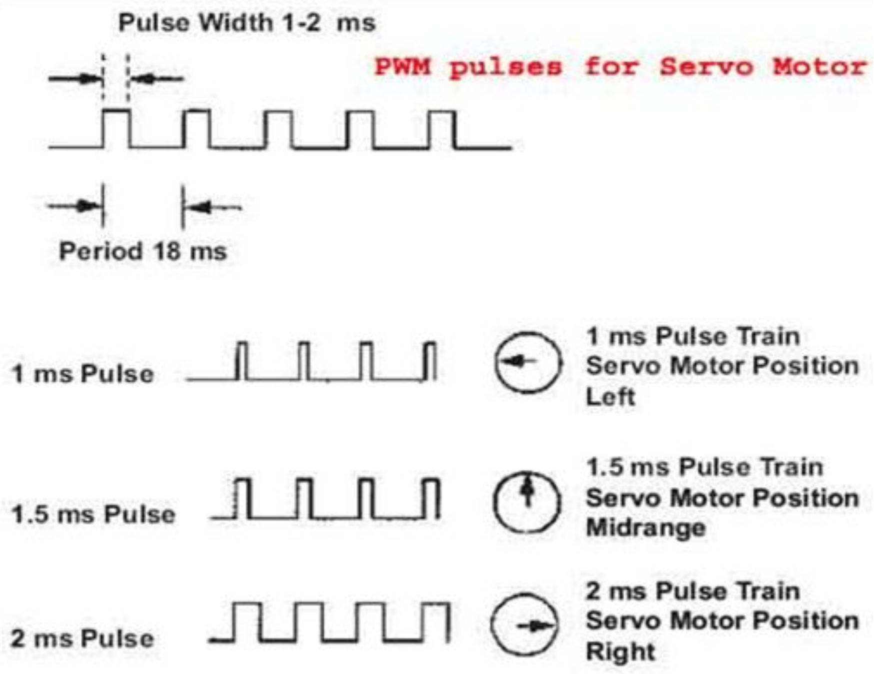


Analog Outputs: Servo Motor

- **Pins:**
  - VCC
  - GND
  - PWM

Analog Outputs: Servo Motor

- **Servo Motor:**
  - Servo checks the pulse in every 20 milliseconds.
  - The pulse of 1 ms (1 millisecond) width can rotate the servo to 0 degrees, 1.5ms can rotate to 90 degrees (neutral position) and 2 ms pulse can rotate it to 180 degree.



Examples: https://wokwi.com/projects/335916736176980563

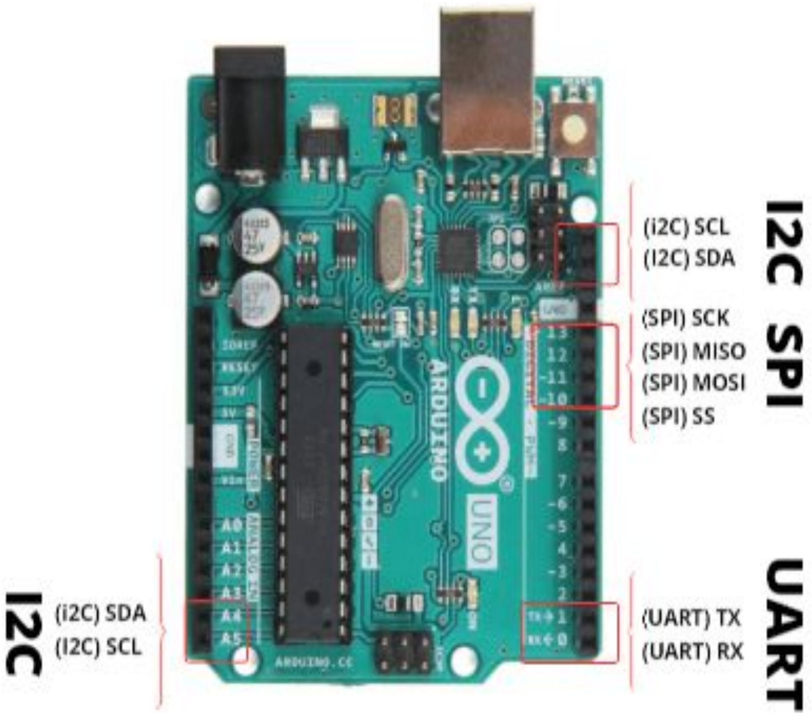
Serial Communication

Arduino Communication Peripherals: Serial Communication

- UART
- I2C
- SPI

Arduino Communication Peripherals: Serial Communication

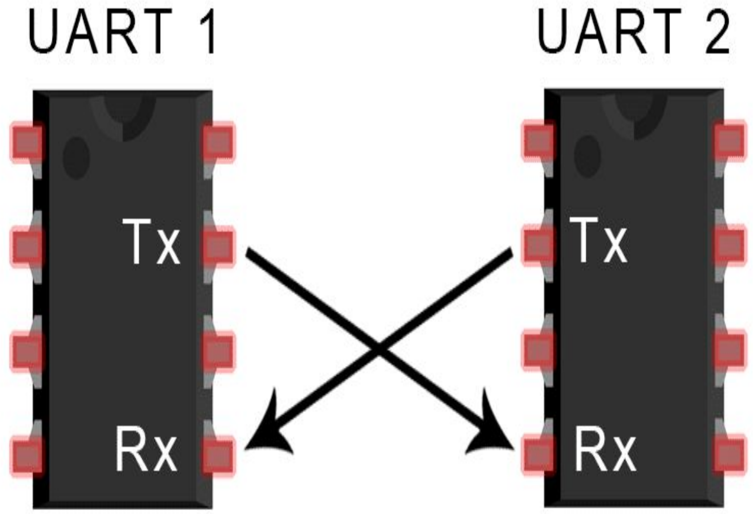
- UART
- I2C
- SPI



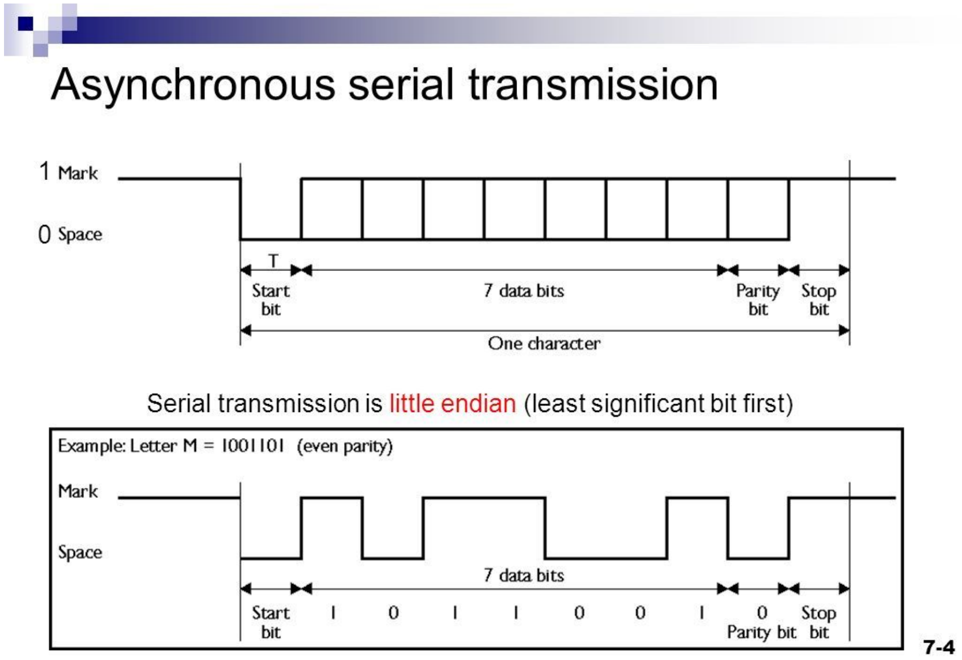
UART(Universal Asynchronous Reception & Transmission)

- Simple serial communication protocol that allows the host (Arduino) to communicate with serial devices [Pin 0 & Pin 1] by using Serial.print(“---”) instructions.
- Supports **Bidirectional**, **Asynchronous** and Serial Data Transmission
- **No clock** needed, uses **Synchronization Bits**, **Data Bits**, **Parity Bits**, **Baud Rate**
- **SoftwareSerial** library has been developed to allow serial communication on other digital pins of the Arduino

UART(Universal Asynchronous Reception & Transmission)



Connection



Communication

Serial Communication (UART): Hardware Mechanism

- Two ways to communicate using UART :
  - Hardware Mechanism
  - Software Mechanism
- Hardware Mechanism uses 2 pins
  - RX - Pin 0 - Used for receiving
  - TX - Pin 1 - Used for transmitting
  - We can also use more than one UART using software mechanism
  - Computer via the USB port (Serial Monitor/Virtual Terminal)
    - Onboard USB-to-Serial converter (Atmega16U2)

Serial Communication (UART): Hardware Mechanism

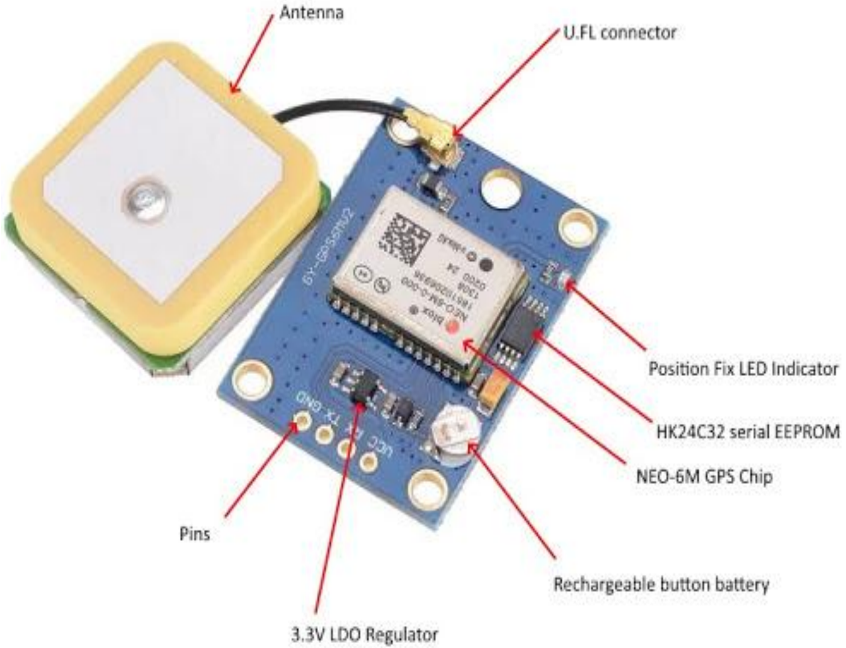
- By **Serial** command we can use UART pins.
- Some Basic UART Functions
  - Serial.begin(Baud rate)
  - Serial.end()
  - Serial.read()
  - Serial.write(parameters)
    - val //1 byte
    - string //series of bytes
    - buf, len//an array of len
- Example: https://www.tinkercad.com/things/0w8iOvi3soz-uart/editel

Serial Communication (UART): Software Mechanism

- Arduino IDE has a built-in software serial library which allows use to perform serial communication using other digital input-output pins.
  - #include <SoftwareSerial.h>
- Create an instant or object (often in global section).
  - SoftwareSerial <obj name> (RX, TX);
  - SoftwareSerial UART0(2, 3);
- To use the UART functions we have to use the instance.
- We can use the UART functions as the member function of the instance.
  - UART0.begin(9600);

Serial Communication (UART): GPS (Global Positioning System) Module

- Connect through serial communication
- Communicate with the Arduino using the UART pins
- Transmit information in NMEA format
- Need TinyGPS library to decode the information



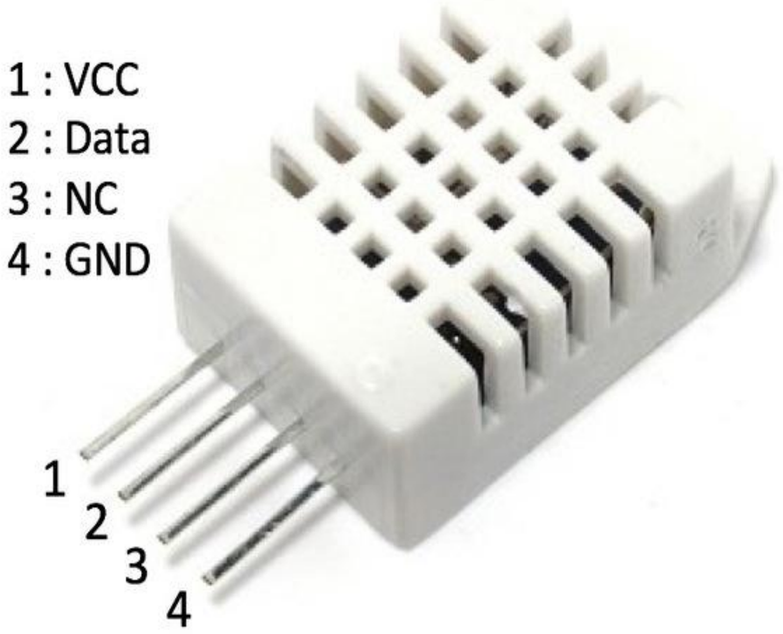
NEO-6M GPS module

Humidity Sensor (DHT22)

It's a digital-output, relative humidity, and temperature sensor.

Pin Connection:

- First pin(VCC) on the left to 3-5V power,
- Second pin(Data) to any digital input pins,
- Third pin(NC) is no-connect pin,
- The right-most pin(GND) to the ground



Link:
https://wokwi.com/projects/335781020271903315

I^2C(Inter-Integrated-Circuit)

- Bidirectional two-wire **Synchronous Half-duplex** serial bus and only 2 wires (SDA and SCL) to transmit information between devices connected to the bus
- Can communicate between up to almost 128 (112) slave devices using 7 bits addressing
- Supports multiple master devices and Two signals − SCL and SDA
    - SCL(A5) is the clock signal, and
    - SDA(A4) is the data signal
- Master can choose any slave device by its unique address
- Need to include “**Wire.h**” library
- Example: https://wokwi.com/projects/336327908183245396

SPI(Serial Peripheral Interface)

- Bidirectional, **Synchronous, Full-duplex** serial bus and 4 wires to transmit information between devices connected to the bus:
  - SCK[13] − This is the serial clock driven by the master.
  - MOSI[11] − This is the master output / slave input driven by the master.
  - MISO[12] − This is the master input / slave output driven by the master.
  - SS[10] − This is the slave-selection wire.
- Compared to UART and I2C, it is the fastest communication peripheral but doesn't support multiple master
- Master can choose any slave device by slave select pin
- Need to include “**SPI.h**” library

Ref

- https://www.nxp.com/docs/en/application-note/AN5250.pdf
- https://www.pjrc.com/teensy/td_libs_TinyGPS.html
- https://lastminuteengineers.com/neo6m-gps-arduino-tutorial/
- https://labjack.com/support/software/examples/lua-scripting/i2c#:~:text=LJStreamUD%20Update%20Package-,I2C%20Sensor%20Examples,Adafruit%2010%2DDOF%20IMU%20Breakout
- https://www.deviceplus.com/arduino/arduino-communication-protocols-tutorial/
- https://circuitdigest.com/microcontroller-projects/arduino-timer-tutorial#:~:text=A%20timer%20uses%20counter%20which,for%20every%2062%20nano%20second.

Thank You