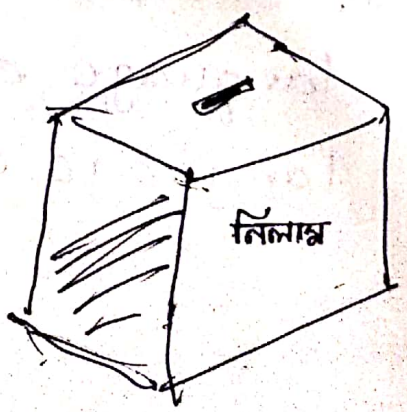
Difference between Assembly Language and High Level Language

Definition of Assembly Language

A low-level programming language which (1) uses symbols and (2) lack variables and functions and which (3) work directly with CPU. Assembly language is coded differently for every type of processor. X86 and X64 processors have a different code of assembly language for performing the same tasks. Assembly language has the same commands as machine language but instead of 0 and 1, it uses names.



Assembly language vs High Level Language

1. In assembly language, programs written for one processor will not run on another type of processor. In high level language, programs run independently of processor type.

2. Performance and accuracy of assembly language code are better than a high level.

3. High level languages have to give extra instruction to run the code on the computer.

4. Code of assembly language is difficult to understand and debug than a high level language.

5. One or two statements of high level language expand into many assembly language codes.

6. Assembly language can communicate better than a high level. Some types of hardware actions can only be performed by assembly language.

7. In assembly language, we can directly read pointers of a physical address which is not possible in high level lang.

8. Assemblers is used to translate codes in assembly language while the compiler is used to compile the

9. The executable code of high level language is larger than assembly language code so it takes longer time to execute

10. Due to long executable code, high

level programs are less efficient than assembly language programs.

11. High level language programmer does not need to know detail about hardware like registers in the processor as compared to assembly programmers.

Example of ~~Assembly~~ Lang:

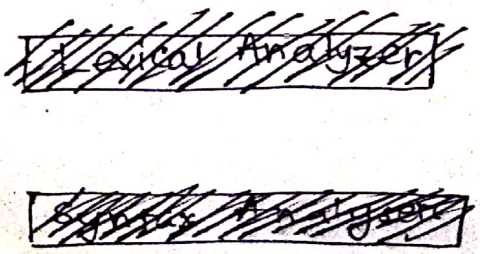
- ARM
- MIPS
- x86
- Z80
- 68000
- 6502
- 6510

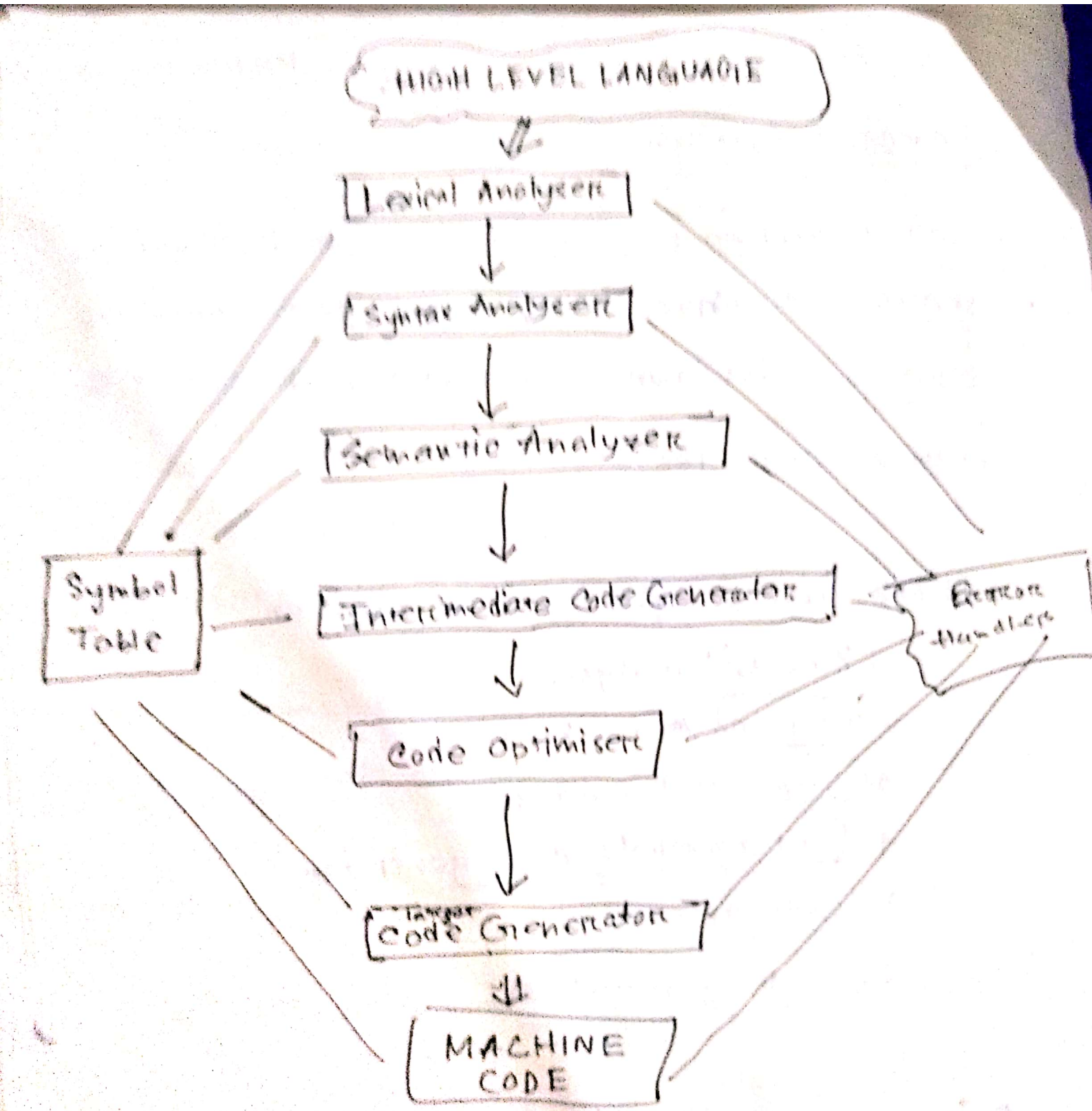
Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from previous stage, has its own representation of source program. and feeds +

There are six phases in a compiler.

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator





Symbol Table:

It is a data structure being used and maintained by the compilers, consists all the identifiers name along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The compilers has two modules namely front end and back end. Front end consists of Lexical Analyser, semantic analysers, syntax analyzer and intermediate code generator. And the rest are assembled from the back end.

1. Lexical Analyser:

It reads the program and converts it into tokens. It converts a stream of lexemes into a stream of tokens. Tokens are defined by regular expressions which are understood by the lexical analyser. It also removes white spaces and comments.

2. Syntax Analyser:

- It is sometimes called parser
- It constructs the parse tree
- It takes all the tokens one by one and uses Context Free Grammar and construct parse tree.

Why Grammar?

- The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.

- Syntax errors are detected in this level if the input is not in accordance with the grammar.

Parse Tree
↓
Semantic Analyser
↓
Semantically Verified Parse Tree

3. Semantic Analyser :

- Verifies the parse tree whether it is meaningful or not
- It furthermore produces verified parse tree
- It also does type checking, label checking and flow control checking.

4. Intermediate Code Generator

- Generates intermediate code, that is a for which can be readily executed by machine.
- Till intermediate code, it is same for every compiler out there. But after that it depends on platform.
- To build a compiler we don't need the compiler ~~for~~ from scratch. We can take intermediate code from the already existing compiler & build the last two parts

5. Code Optimiser —

- It transforms the code so that it consumes fewer resources and produces more speed

- removes unnecessary code lines and arranges the sequence of statements in order to speed up the program execution without wasting resources.

6. Target Code Generator

- write codes that machine can understand

- register allocation, instruction selection

Definition of Loader

As the program that has to be executed currently must reside in the main memory of the computer. It is the responsibility of the loader, a program in an operating system, to load the executable file module of a program generated by the linker, to the main memory for execution. It allocates the memory space to the executable module in the main memory.

There are three kinds of loading approaches —

1. Absolute loading
2. Relocatable loading
3. Dynamic Runtime loading

Absolute loading:

- This approach loads the the executable file of a program into a same main memory location each time.
- Disadvantage — A programmer must be aware of the assignment strategy for loading the modules to main memory.
- Disadvantage — The program is to be modified involving some insertion and deletion in the program, then all the addresses of the program have to be altered.

Relocatable Loading:

- In this approach, the compiler/assembler does not produce actual main memory address. It produces the relative addresses.

Definition of linker

An assembler generates object code of a source program and hands it over to the linker. The linker takes this object code and generates the executable code for the program, and hand it to the loader

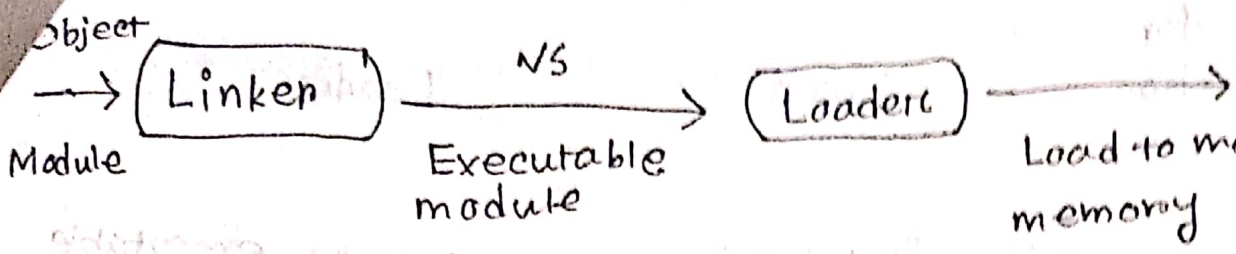
The high level language, programs have some built-in- libraries and header files. The source program may contain some library functions whose definition was stored in built-in libraries. In case the built-in libraries are not found it informs the compiler and the compiler then generates the error.

Sometimes the large programs are divided into the subprograms which are called modules. Now when these modules are compiled and assembled, the object moduks of the source program

are generated, The linkers has the responsibility of combining/linking all the object modules to generate a single executable file of the source program. We have two types of linkers -

i. Linkage Editors
   - generates relocatable, executable module

ii. Dynamic Linker
   - It defers/postpones the linkage of some external modules until the load module/executable module is generated. Here, linking is done during load time / run time.



Linker and loader are the utility program that plays a major role in the execution of a program. The source code of a program pass through compiler, assembler, linker, loader in the respective order, before execution. On the one hand, where the linker intakes the object code generated by the assembler and combine them to generate executable module, On the other hands, the loader loads this executable module to the main memory for execution.

Comparison between [redacted]
LINKER & LOADER

Basis for Comparison	[redacted] Linker	Loader
1. Basic	It generates the executable module of a source program	It generates the executable module to the main memory.

Basis for comparison	Linker	Loader
2. Input	It takes the object input; the object code generated by an assembler	It takes executable module generated by a linker
3. Function	It combines all the object modules of a source code to generate an executable module	It allocates the address to an executable module in main memory for execution
4. Type/Approach	Linkage Editors/ Dynamic Linkers	Absolute loading/ Relocatable loading and Dynamic - Run-time loading

Dynamic Run-Time Loading :-

- In this approach the absolute address for a program is generated when an instruction of an executable module is actually executed. It is very flexible, the loadable module/executable module can be loaded into any region of memory. The executing program can be interrupted in between and be swapped out of the disk and back to the main memory address.

The linker takes the object modules of a program from the assembler and links them together to generate an executable module of a program. The executable module is then loaded by the loader into the main memory for

Interpreter	Compiler
1. Translates the program one statement at a time.	1. Scans the whole programs and translates it as a whole into machine code.
2. Takes less amount of time to analyze the source code but overall execution time is slower.	2. Takes large amount of time to analyze the source code but overall execution time is comparatively faster.
3. No intermediate object code is generated. Memory Efficient.	3. Generates intermediate code which further requires linking hence requires more memory.
4. Continues translating the program until the first error is met. Easy debug.	4. Generates error msg after scanning the whole program.
5. Python, Ruby	5. C / C++

- MODEL SMALL Code এর আইজ কতটুকু হবে। ৬ রকমের অপশন আছে।

- STACK 100H

- DATA

    ; Variables are declared

- CODE

    ; CODE SEGMENT

    ; FUNCTIONS INCLUDING MAIN

    MAIN PROC

        ; STATEMENTS

        ; return 0;
        MOV AH, 4CH
        INT 21H

    MAIN ENDP

END MAIN ; EXIT (0)

- Instruction Destination, Source

```
MOV
ADD
SUB
MUL
DIV
```

- MOV NUM1, NUM2 ; দুইটা ভেরিয়েবল — সম্ভব না।

- ```
  MOV BH, 4
  MOV BL, 5
  ADD BH, BL  // BH += BL
  ```

- ADD NUM1, BH ; NUM1 = NUM1 + BH

- DB → 8 bit register BH, BL, ... (1 byte)
  DW → 16 bit register BX , ... (1 word)

- NUM1 DB 'd' ; d = 100 in ASCII, equivalent ASCII ; representation দিতে হবে।

```
NUM2 DW ?
CHAR DB '#'
STR DB "My Name is EMU8086"
```

- ```
  MOV AX, @DATA
  MOV DS, AX  ; DATA SEGMENT - DS
  ```

// কোডে ডিক্লেয়ার করা ভেরিয়েবলগুলো ইউজ করার জন্য।

```
NUM1  DB  48
CHAR  DB  'A'
MSG   DB  "My Name is Sayef $"

MOV AH, 2      ; Print Mode
MOV DL, NUM1
INT 21H
```

} 0 (48 ASCII Value '0' এর জন্য)

- New line প্রিন্ট করা

```
MOV AH, 2      ; Print Mode
MOV DL, 0AH    ; New line
INT 21H

MOV DL, 0DH    ; Carriage Return
INT 21H
```

- স্ট্রিং প্রিন্ট করা

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

- .MODEL SMALL
  MEDIUM
  COMPACT
  LARGE
  HUGE

বোর্ডের আইডিয়ার উপর ডিপেন্ড করে।

☐ সিংগেল ক্যারেক্টার ইনপুট

```
MOV AH, 1    ; ইনপুট মুড ।
INT 21H      ; ইনপুটটা AL এ থাকবে ।
```

প্রিন্ট করা –

```
MOV AH, 2  ; প্রিন্টিং মুড
MOV DL, AL
INT 21H
```

☐ ADD এর ফাংশানালিটি

A, B - ইনপুট নিয়ে A+B যোগ করবে ।

উত্তরঃ

```
.MODEL SMALL
.STACK 100H

.DATA
    A DB '?'
    B DB ?


.CODE
    MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        MOV AH, 1
        INT 21H
        MOV A, AL
        SUB A, 48

        INT 21H
        MOV B, AL
        SUB B, 48

        MOV BH, A // BH=A
        ADD BH, B // BH+=B
        ADD BH, 48 // BH+=48

        ; new line
        MOV AH, 2
        MOV DL, 0AH
        INT 21H
        MOV DL, 0DH
        INT 21H

        MOV AH, 2
        MOV DL, BH
        INT 21H

        MOV AH, 4CH
        INT 21H
    MAIN ENDP
END MAIN
```

Using for-like

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ;Variables

.CODE
    MAIN PROC
        MOV CX, 5
        MOV BX, 0

        START:
          CMP BX, CX   [BX < CX]
          JE LAST
          PRINTN "HERE"
          INC BX
          JNE START

        LAST:
          PRINTN "FINISHED"

        MOV AH, 4CH
        INT 21H
    MAIN ENDP
END MAIN
```

- INC BX BX++
- INC CX CX++

if-else using JMP

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ;Variables

.CODE
  MAIN PROC
    MOV BX, -10
    CMP BX, 0
    JL IF
    JGE ELSE

    ELSE:
    IF:
      PRINTN "NEGETIVE"
      JMP ENDER

    ELSE:
      PRINTN "POSITIVE"
      JMP ENDER

    ENDER:
      ;blank scope

    MOV AH, 4CH
    INT 21H
  MAIN ENDP
END MAIN
```

- XCHG // swapping

```
MOV BL, 53   ; 5
MOV BH, 55   ; 7
XCHG BL, BH
```

```
MOV AH, 2
MOV DL, BL
INT 21H
```
} 7.

```
MOV AH, 2
MOV DL, BH
INT 21H
```
} 5

- INCLUDE 'EMU8086.INC'

PRINT = Print a string without newline
PRINTN = Print a string with new line

- Unconditional Jump – JMP

- Conditional Jump

CMP COMPARED_TO, COMPARED_WITH // CMP BX, CX

JE/JE Jump equal / Jump Zero
JNE/JNZ Jump Not equal / Jump

JG

Loop keyword

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ;Variables

.CODE
    MAIN PROC
        MOV CX, 5  ; লুপ ৫ বার ঘুরবে

        TOP:
            PRINTN "LOOP IS RUNNING"
            LOOP TOP

        PRINTN "LOOP IS ENDED"
            ;

        MOV AH, 4CH
        INT 21H
    END MAIN
MAIN ENDP
END MAIN
```

Output:

```
LOOP IS RUNNING
LOOP IS RUNNING
LOOP IS RUNNING
LOOP IS RUNNING
LOOP IS RUNNING
LOOP IS ENDED
```

For loop / Loop using JMP

1+2+3+ .... +n = ?

Solution:

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    MSG1  DB  "ENTER N: "
    MSG2  DB  "THE SUM IS: "
    MSG3  DB
    MSG4  DB
    N     DB  ?    SUM DB 0
.CODE
    MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        MOV AH, 9
        LEA DX, MSG1
        INT 21H

        MOV AH, 1
        INT 21H    SUB AL, 48
        MOV N, AL

        ;newline
        MOV DL, 0AH
        INT 21H
        MOV DL, 0DH
        INT 21H

        MOV BL, 1

        TOP:
          CMP BL, N
          JG EXIT_LOOP
          ADD SUM, BL
          INC BL
          JMP TOP
```

```
EXIT_LOOP:
    MOV AH, 9
    LEA DX, MSG2
    INT 21H

    MOV AH, 2
    MOV DL, SUM
    INT 21H

MOV AH, 4CH
INT 21H

MAIN ENDP
END MAIN
```

NESTED LOOP

Print square like

```
***
***
***
```

n=3

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
  N DB ?

.CODE
 MAINPROC
    MOV AX, @DATA
    MOV DS, AX

    MOV AH, 1   ; INPUT MODE
    INT 21H
    SUB AL, 48 ; MAKING NUM
    MOV N, AL

~~MOV BH~~

~~MOV BH, 1 ; PRINT~~

~~PRINT~~
;newline
MOV AH, 2    ; OUTPUT
MOV DL, 0AH
INT 21H
MOV DL, 0DH
INT 21H

MOV BH, 1   ;(i=1)
MOV AH, 2   ; PRINT MODE
MOV DL, '*'

LOOP1:
    CMP BH, N
    JG LOOP1_EXIT

    MOV BL, 1    ;(j=1)
    LOOP2:
        CMP BL, N
        JG LOOP2_EXIT

        INT 21H ; print *
        INC BL ; j++
        JMP LOOP2

        LOOP2_EXIT:
        ~~MOV~~ ; newline
            MOV AH, 2
            MOV DL, 0AH
            INT 21H
            MOV DL, 0DH
            INT 21H

            INC BH (i++)
            MOV DL, '*'

            JMP LOOP1

    LOOP1_EXIT:

    MOV AH, 4CH
    INT 21H
MAIN ENDP
END MAIN
```

Logical Operators

AND

```
MOV BH, 1111B      ; 15D   (Binary)
AND BH, 0100B      ; BH = 0100B

;Printing
MOV AH, 2
MOV DL, BH
INT 21H
```

OR Operators

```
MOV BH, 1111B
MOV
OR  BH, 0100B      BH

MOV BH, 0101B
OR  BH, 0010B      // 0111
ADD BH, 48         // characterise
```

XOR Operator

```
XOR BH, 0101B

XOR BH, BH         // BH ← 0
```

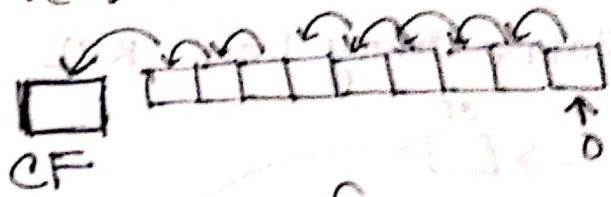
NOT Operator

```
MOV BH, 11110011B
NOT BH

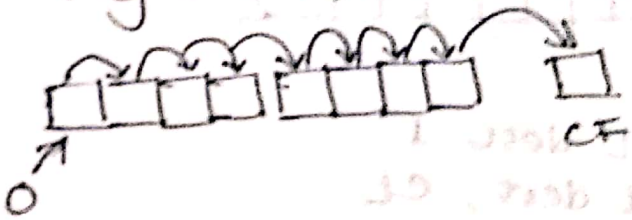
MOV AH, 2
MOV DL, BH
INT 21H
```
} 00001100

SHIFT INSTRUCTION

left shift

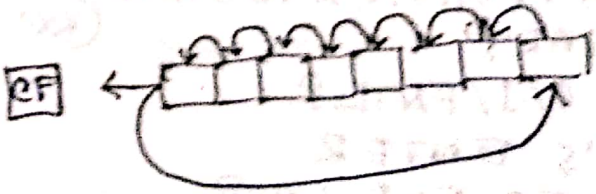


right shift



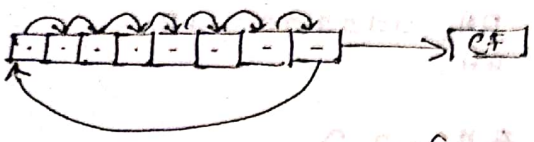
Rotate Left

just like left shift, but MSB is shifted to rightmost




```
ROL destination, 1
ROL destination, CL
```

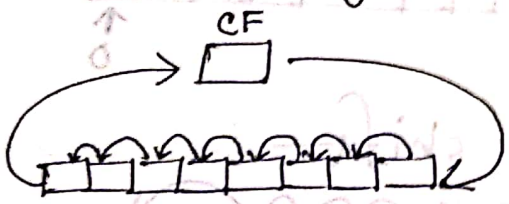
Rotate Right



the rightmost bit is shifted to MSB

```
ROR dest, 1
ROR dest, CL
```

Rotate Carry Left RCL




```
RCL dest, 1
RCL dest, CL
```

Binary input নিতে হবে।

CLEAR A REGISTER TO STORE DATA (BX)
LOOP START
INPUT 0/1/ENTER
IF IT'S ENTER
IF ENTER END LOOP
BX SHL
SAVE THE INPUT TO LSB

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ;VARS

.CODE
    MAIN PROC
        MOV BX, 0

        MOV AH, 1   ;input mod

        FOR1:
            INT 21H
            CMP AL, 0DH
            JE END-FOR1
            SUB AL, 48
            SHL BX, 1
            OR BL, AL
            JMP FOR1

        END-FOR1:

        MOV AH, 4CH
        INT 21H
    MAIN ENDP

END MAIN
```

g binarcy input and printing
output (Sayef Regash video -- 20)

```
INCLUDE 'EMU8086.INC'
.MODEL SMAL
.STACK 100H

.DATA
    COUNT DB 0

.CODE
  MAIN PROC
      MOV AX, @DATA
      MOV DS, AX

      MOV BX, 0

      MOV AH, 1

    FOR1:
      INT 21H
      CMP AL, 0DH
      JE END_FOR1

      SUB AL, 48
      SHL BX, 1
      OR BL, AL
      INC COUNT
      JMP FOR1

    END_FOR1:

    PRINTN { MOV AH, 2
             MOV DL, 0AH
             INT 21H
             MOV DL, 0DH
             INT 21H }

      MOV CL, 16
      SUB CL, COUNT
      RCL BX, CL

    { XOR CH, CH
      MOV CL, COUNT }

    MOV AH, 2

    FOR2:
      RCL BX, 1
      JC OUTPUT_ONE
      MOV DL, '0'
      INT 21H
      JMP LAST_OF_LOOP
      OUTPUT_ONE:
        MOV DL, '1'
        INT 21H
      LAST_OF_LOOP:
        LOOP FOR2

      MOV AH, 4CH
      INT 21H

  MAIN ENDP
  END MAIN
```

~~Hex II~~

Hex Input Output

(Sayef Rqadh – 2122)

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ; Vars

.CODE
MAIN PROC
    MOV BX, 0
    MOV CL, 4
    PRINT "ENTER A HEXNUM:"
    MOV AH, 1 ;INPUT MODE

    FOR1:
        INT 21H
        CMP AL, 0DH
        JE END-FOR

        CMP AL, 41H
        JGE LETTER

        ; OTHERWISE DIGIT
        SUB AL, 48
        JMP SHIFT

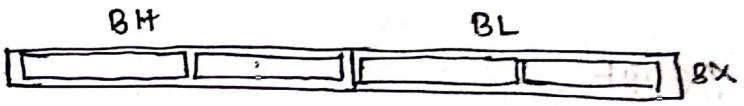
    LETTER:
        SUB AL,

    SHIFT:
        SHL BX, CL
        OR BL, AL

    JMP FOR1
    END FOR1:

    ~~MOV AH~~

    ;Outputting hex
```




```
    XOR CH, CH
    MOV CL, 4
    MOV AH, 2

    FOR2:
        MOV DL, BH
        SHR DL, 4
        SHL BX, 4
        ~~0000~~

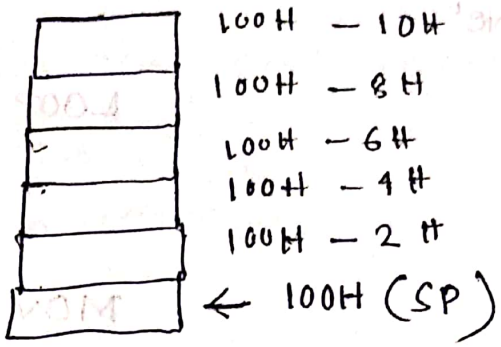
        CMP DL, 48
        JGE LETTER2

        ;OTHERWISE DIGIT
        ADD DL, 48
        INT 21H
        JMP END2

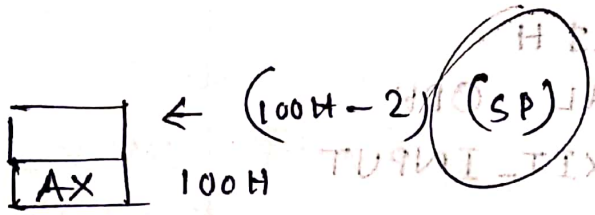
        LETTER2:
            ADD DL, 55
            INT 21H

        END2:
            LOOP FOR2
ender
```

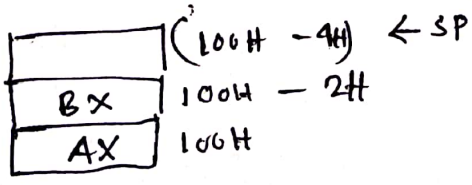
ODUCTION TO STACK



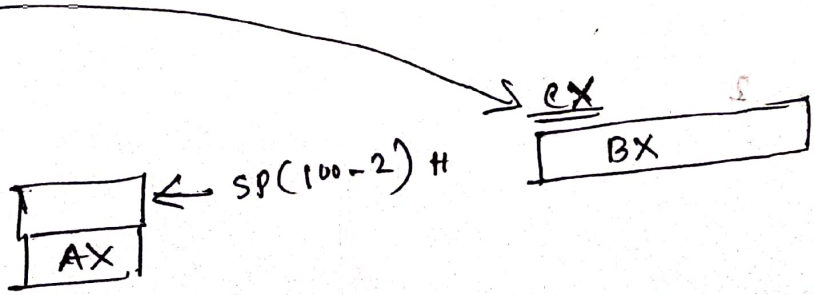
Push AX



Push BX



POP CX



String Reverse Using Stack

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ; Vars

MAIN PF

.CODE

MAIN PROC
    PRINT "ENTER: "
    MOV AH, 1   ; Input Mode
    XOR CX, CX

    INPUT:
        INT 21H
        CMP AL, 0DH
        JE EXIT_INPUT

        PUSH AX
        INC CX

        JMP INPUT

    EXIT_INPUT:

    PRINTN
    PRINT "OUTPUT: "


    MOV AH, 2
        OUTPUT:
            POP DX
            INT 21H
        LOOP OUTPUT


        MOV AH, 1CH
        INT 21H
    MAIN ENDP

    END MAIN
```

Introduction to Function

```
return_type function_name (parameters){
        // actions
        return return_type
}
```

→ in high level languages

→ ~~in assem~~

→ END_MAIN indicates the whole program is finished. So functions should be written before END_MAIN

```
FUNCTION_NAME  PROC
        // STATEMENTS
        RET
FUNCTION_NAME  ENDP

// calling
CALL FUNCTION_NAME
```

যেখান প্রসিজিউরের যেখান থেকে কল করা হয়েছে তার পরবর্তী লাইনে ব্যাক করবে

Sample Program of using function

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    ; Variables

.CODE
    MAIN PROC
        CALL FN
        PRINTN "I am the main fn"

        MOV AH, 4CH
        INT 21H
    MAIN ENDP

    FN PROC
        PRINTN "Hello World"
        PRINTN "I am the function"

        RET
    FN ENDP
END MAIN
```

Output:

```
Hello World
I am the function
I am the main fn
```

Multiplication

```
                1 1 1 1 1 1 1 1
                    1 0 0 1 1
---------------------------------
        . . . 1 1 1 1 1 1 1 1 0
      . . . 1 1 1 1 1 1 1 1 0 1
    . . 0 0 0 0 0 0 0 0 0 0 2
  . 0 0 0 0 0 0 0 0 0 0 0 3
1 1 1 1 1 1 1 1 0 0 0 0 4
```

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA

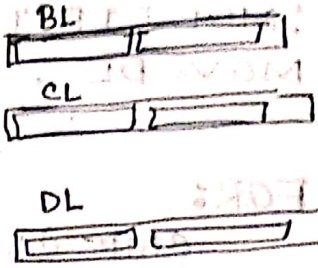
.CODE

MAIN PROC

    CALL MULTIPLICATION
    MOV AH,4CH
    INT 21H
MAIN ENDP

MULTIPLICATION PROC
    ;INPUT
    PRINT "FIRST NUMBER: "
    MOV AH, 2
    INT 21H
    MOV BL, AL      SUB BL, 48
    PRINTN
    PRINT "SECOND NUMBER: "
    INT 21H
    MOV CL, AL      SUB CL, 48
    PRINTN


    ;MULTIPLICATION
    MOV DL,
```



num1 8 bit

num2 8 bit

num3 8bit which is multiplication product of num1 & num2

```
;MULTIPLICATION
MOV DL , 0    ;TO STORE THE SUMMATION

FOR:
    CMP CL , 0
    JE END_FOR
    SHR CL, 1
    JNC CHECK
    ;OTHERWISE '1' IN CARRY FLAG
    ADD DL , BL
    @
    CHECK:
    SHL BL , 1
    JMP FOR

END_FOR:

;OUTPUT

MOV AH, 2
INT 21H


RET

MULTIPLICATION ENDP

END MAIN
```

Multiplication Function
(Sayef Reyadh - 27)

MUL and IMUL -function

MUL — for signed numbers
IMUL — for unsigned numbers } (?)

multiplier/multiplicant এর যে কোন একটাকে AX এ নিতেই হবে

```
MOV AX, 10
MOV BX, 9
```

10X9 = 90 = 5Ah

0000	005A
HCX	HCX
DX	AX

```
MUL BX          ; BX ~~~~ B
    ; DX : AX = BX * AX
```

```
MOV AL, 1
MOV DL, 2
MUL BL          ; AH : AL = BL * AL
```

28. DIV & IDIV

dividend = 5
divisor = 2

```
MOV AX, 5
MOV BX, 2
DIV BX      ; DX : AX = AX / BX
```

DX = Remainder
AX = Result / Quotent

IDIV

```
MOV AX, 5
MOV BX, -2
IDIV BX     ; DX : AX = AX / BX
```

Rem 2's complement

```
MOV AL, 5
MOV BL, 2
DIV BL      ; AH : AL = AL / BL
```

Decimal Input output

Decimal Input

```
INCLUDE 'EMU8086.INC"
.MODEL SMALL
.STACK 100H

.DATA
    N   DB  0

.CODE

MAIN PROC
    MOV AX, @DATA
    MOV DS, AX

    CALL DECIMAL_INPUT
    MOV AH, 2  INT 21H  → to see in ASCII chart
    MOV AH, 4CH
    INT 21H
MAIN ENDP


DECIMAL_INPUT PROC
    PRINT "INPUT: "
```

~~XOR CX, CX ; COUNTER~~
XOR DX, DX

```
FOR:
    MOV AH, 1
    INT 21H

    CMP AL, 0DH
    JE END_FOR
    SUB AL, 48
    MOV N, AL
    MOV AL, 10
    MUL DL
    ADD AL, N
    MOV DL, AL
    JMP FOR

END_FOR:

    RET
DECIMAL_INPUT ENDP

END MAIN
```

Outputting Decimal

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    N DB 0
    COUNT DB 0

.CODE
    MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        XOR CX, CX
        XOR DX, DX

        CALL DECIMAL_INPUT

        PRINTN
        MOV N, DL
        CALL DECIMAL_OUTPUT


        MOV AH, 4CH
        INT 21H

    MAIN ENDP
```

```
DECIMAL_INPUT PROC
    PRINT "INPUT: "

    FOR:
        MOV AH, 1
        INT 21H

        CMP AL, 0DH
        JE END_FOR

        SUB AL, 48

        MOV N, AL
        MOV AL, 10
        MUL DL
        ADD AL, N
        MOV DL, AL
        INC COUNT
        JMP FOR

    END_FOR!

    RET
DECIMAL_INPUT ENDP
```

```
DECIMAL_OUTPUT PROC
    PRINT "OUTPUT: "

    XOR CX, CX
    MOV CL, ~~10~~ COUNT

    MOV BL, 10
    XOR AH, AH  // remainder রাখবো
    MOV AL, N   // remaining থাকবে

    FOR2:
        DIV BL ; AL = AL / BL
        XOR DX, DX
        MOV DL, AH
        ADD DL, 48
        PUSH DL
    LOOP FOR2

    XOR CX, CX
    MOV CL, COUNT

    FOR3:
        MOV AH, 2
        POP DX
        INT 21H
    LOOP FOR3
```

31. Array Introduction

```
NUM   DB   0, 1, 2, 3, 4, 5, 6, 7, 8, 9

NUM2  DB   10 DUP (?)
NUM3  DB   10 DUP (0)   ← প্রতিটি element 0 দিয়ে initialized হবে।
```

```
TST   DB   5, 4, 3 DUP (2)
                ↙ sizeofdup
```

$\Rightarrow \{5, 4, 2, 2, 2\}$

```
TST2  DB   5, 4, 3 DUP (2, 3 DUP (4))
```

$\Rightarrow$ 5 4 2 4 4 4 2 4 4 4 2 4 4 4

32. Array Index

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    NUM DB 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    NUM2 DW 61, 62, 63, 64,
            65, 66, 67, 68, 69

.CODE
MAIN PROC
    MOV AX, @DATA
    MOV DS, AX

    LEA SI, NUM2

    MOV, CX, 10
    MOV AH, 2

FOR:
    MOV DL, [SI]
    ADD DL, 48
    INT 21H
    ADD SI, 1
LOOP FOR

    MOV AH, 4CH
    INT 21H
MAIN ENDP
END MAIN
```

Register Indirect Mode

// ADD SI, 2

Based & Index Addressing Mode

see 33 34

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA

    NUMBER1 DB  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
    NUMBER2 DW  65, 66, 67, 68, 69

.CODE
    MAIN PROC
        MOV AX, @DATA
        MOV DS, AX

        MOV CX, 10  ; FOR LOOPING
        XOR BX, BX  ; BX=0 (i=0)

        MOV AH, 2

        FOR:
            MOV DL, NUMBER1[BX]
            ADD DL, 48
            INT 21H
            ADD SI, 1

        LOOP FOR

        MOV AH, 4CH
        INT 21H
    MAIN ENDP

END MAIN
```

Array input and output

```
INCLUDE 'EMU8086.INC'
.MODEL SMALL
.STACK 100H

.DATA
    NUM  DB  10 DUP (?)

.CODE

MAIN PROC
    MOV AX, @DATA
    MOV DS, AX

    XOR BX, BX
    MOV CX, 16

    MOV AH, 1
    FOR:
      INT 21H
      MOV NUMBER[BX], AL
      INC BX
    LOOP FOR

    XOR BX, BX
    MOV CX, 16

    MOV AH, 2

    FORRR:
      MOV DL, NUMBER[BX]  INT 21H
      INC BX
    LOOP FORRR
    =
```

CT # 4 Trocks

Assembly & Micro

- One of the control flags is the direction flag (DF)
- DF is implemented by two index register -
  - SI
  - DI

- When DF = 0
  SI, DI goes from left to right
  When DF = 1
  SI, DI goes from right to left

- CLD // clear direction flag
  DF := 0
  STD // set direction flag
  DF := 1

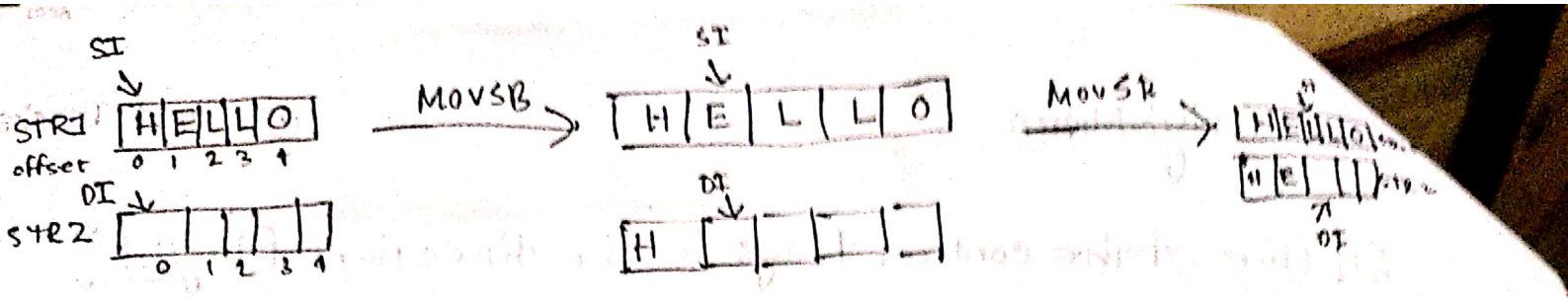
- ~~MOVSB~~
  MOVSB
  Copies the content of DS:SI to the byte addressed by ES:DI.

```
MOV AX, DATA
MOV DS, AX
MOV ES, AX

LEA SI, STR1
LEA DI, STR2

CLD    // left to right
```

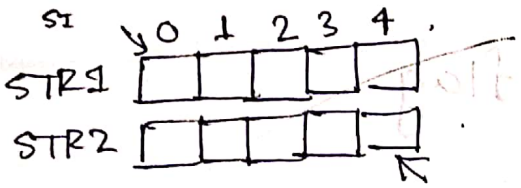
```
MOVSB   // moves the first byte
MOVSB   // moves the second byte
```



** MOVSB /MOVSW
— ~~an~~ instructions to permit memory - memory operation.

```
CLD   // left to right
LEA SI, STR1
LEA DI, STR2
MOV CX, 5
REP MOVSB  // 5 times MOVSB will be functioned.
```

Copy STR1 to STR2 in reverse order etc.




```
LEA SI, STR1
LEA DI, STR2 + 4
CLD   // left to right

MOV CX, 5  // loop will be executed for five times

LOOPMOV:
    MOVS B   // str2[DI] = str1[SI]; SI++; DI++;
    DEC DI
    DEC DI   // two times
LOOP LOOPMOV
```

- STOSB ; Store a string byte
  STOSW ; Store a string word

STOSB

— moves the content of AL to the byte pointed by DI

STOSW

— moves the content of AL to the word pointed by ES:DI

- ```
  MOV AX, @DATA
  MOV ES, AX

  CLD //left to right
  MOV AL, 'A'

  LEA DI, STR1+0
  STOSB
  STOSB
  ```

H	E	L	L	0
A	E	L	L	0
A	A	L	L	0

- **LODSB**

— loads a string byte from DS:SI into AL

— for word, LODSW

```
MOV AX, @DATA
MOV DS, AX    LEA SI, STR1
CLD //left to right
LODSB
LODSB
```

STR1

0	L	A
0	L	A
0	L	A

AL

?
0
L
```

For moving a word — MOVSW

ARR DW | 10 | 20 | 40 | 50 | 60 | ? |
0 2 4 6 8 10

Insert 30 between 20 and 40

Output will be like: ARR DW | 10 | 20 | 30 | 40 | 50 | 60 |
0 2 4 6 8 10

Solution:

```
.MODEL SMALL
.STACK 100H

.DATA
   ARR DW 10, 20, 40, 50, 60, ?
          0   2   4   6   8   10

.CODE
   MAIN PROC
        MOV AX, @DATA
        MOV DS, AX
        MOV ES, AX

        ~~LEA SI, ARR+4~~
        ~~LEA DI, ARR+6~~
        ~~CLD //left to right~~
        ~~MOV CX, 3~~
        ~~REP MOVSB~~

        LEA SI, ARR+8
        LEA DI, ARR+10
        STD ;right to left
        MOV CX, 3
        REP MOVSB

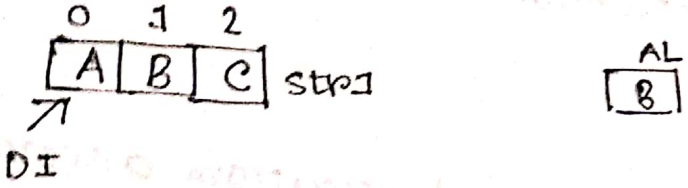
        ~~MOV ARR+4, 30~~ // LEA DI, ARR+4
                          MOV WORD PTR [DI], 30

   MAIN ENDP

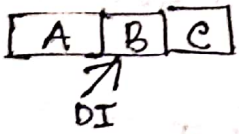
END MAIN
```

III SCASB

– Compares content of AL with content of ES:DI

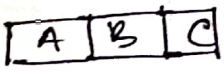


SCASB



AL: B — ZF = 0 Not found!

SCASB



AL: B — ZF = 1 Yo! Found.

~~IV Comparing Strings~~

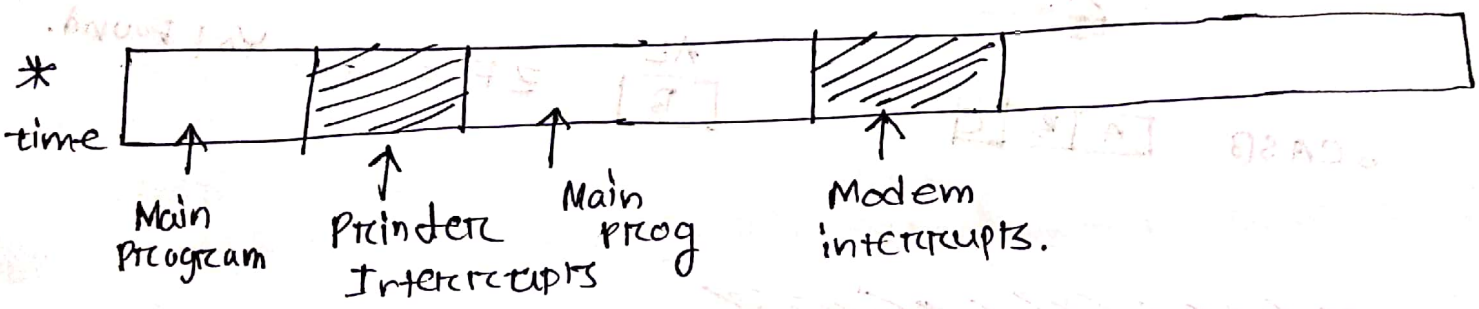
Interrupt in 8086

* An interrupt vector table is a data structure that associates a list of interrupt handlers with a list of interrupt requests in a table of interrupt vectors.

* Interrupt is the method of creating a temporary halt during program execution and allow peripheral/external devices to access microprocessors.

* Interrupt Service Routine (ISR) is a short program to instruct the microprocessor how to handle the interrupts.

* Processor jumps to the special program according to the interrupt service routines to serve the peripherals.



* Types of interrupts –
  - i. hardware interrupt
    - Maskable interrupt (can continue with main prog)
    - Non maskable interrupt
  - ii. software interrupt — 256 types of interrupt

~~Interrupts~~

* HW interrupt caused by any peripheral devices.

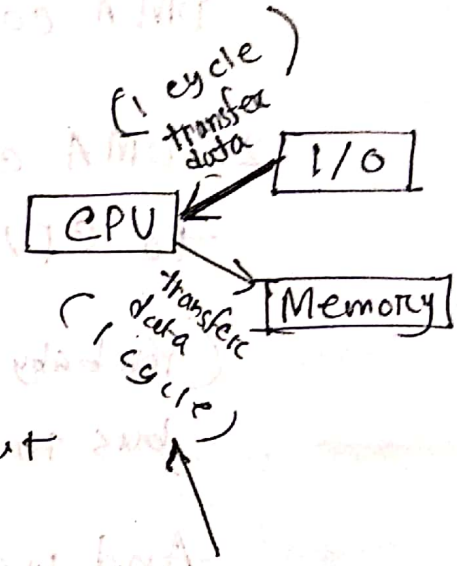
* HW interrupt pins
  - NMI (Non maskable interrupt) cannot continue with another program
  - INTR
  - INTA (Interrupt Acknowledgement)

☐ Software interrupts

INT (interrupt instruction with the type number)

eg. INT 3 — break point of interrupt
INT 0 — interrupt of over-flow

☐ DMA Controller 8257
(Direct Memory Access)

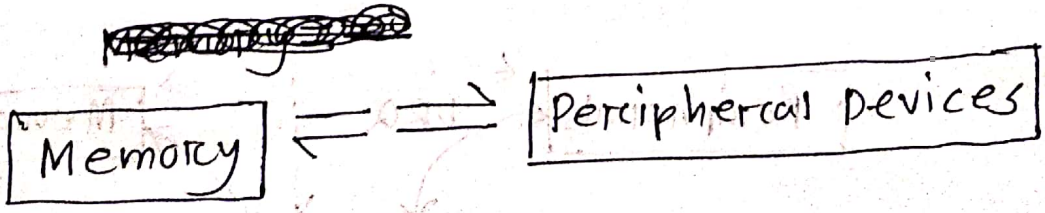


- The input/output devices want to access memory directly without the CPU.

A cycle contains
- fetching
- decoding
- executing

☐ To make the data transfer rate fast — DMA

☐ It is designed by INTEL to transfer data at the fastest rate



Ex: How is DMA operations performed?

1. Peripherical device wants to send data to memory
2. first peripherical device has to send DMA request to DMA controller.
3. DMA controller send HLQ (Hold request) to CPU.

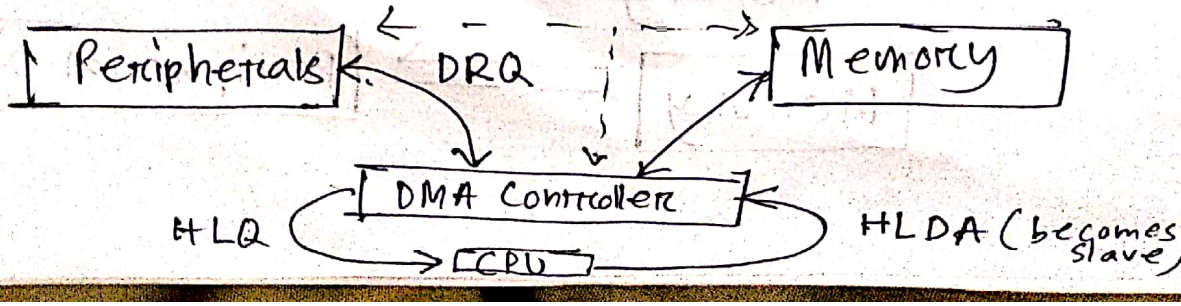
(You baby please remain hold and handover the system bus to meh)

And waits for the CPU to send hold acknowledgement (HLDA)

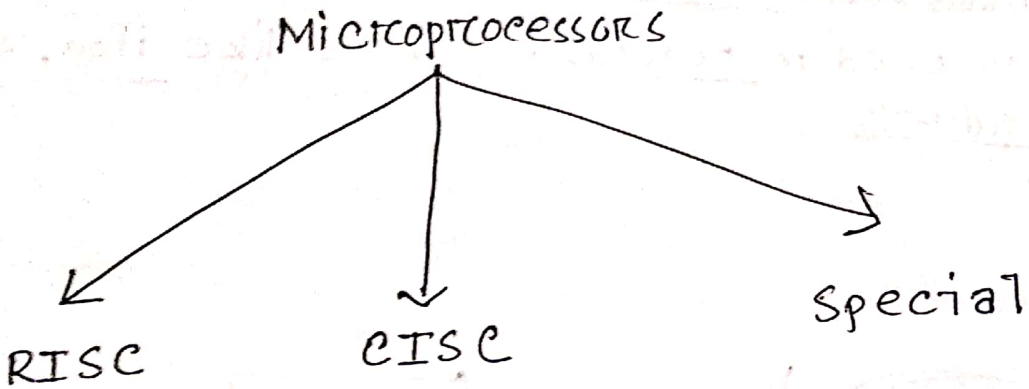
4. Then CPU becomes slave, DMA becomes the master.

DIRECT MEMORY ACCESS

5. DMA controller has to manage operations over buses between CPU, Memory & peripherals



RISC [Reduced Instruction set Computer]

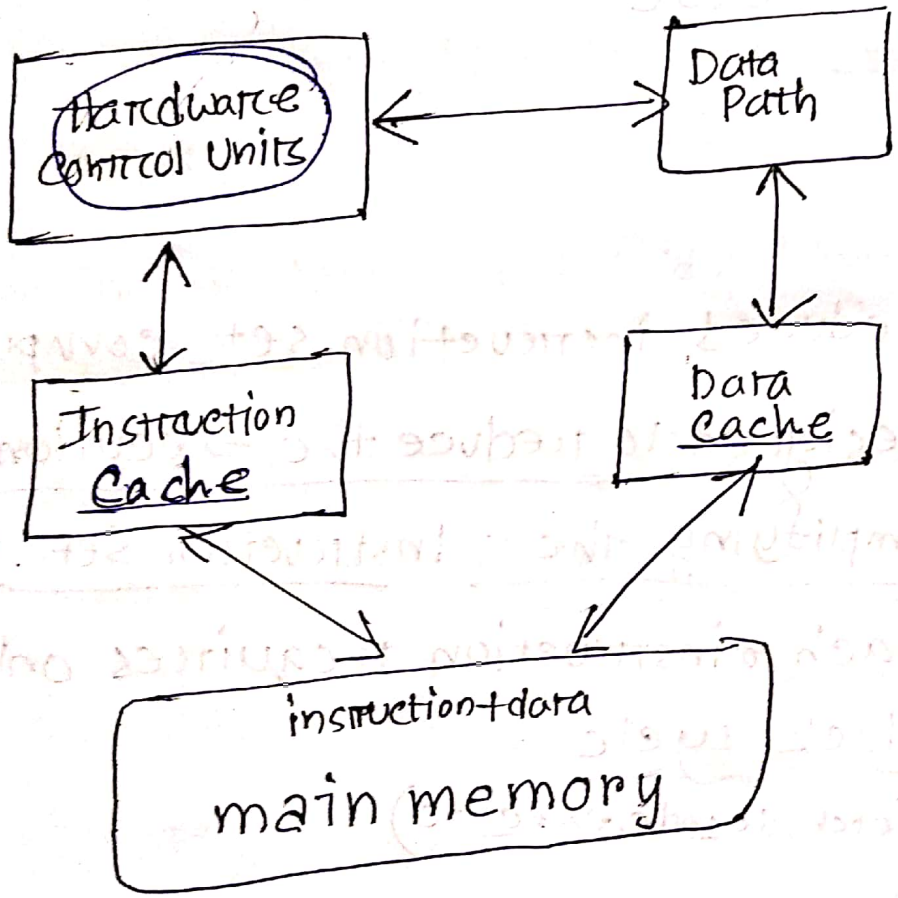


RISC:

- reduced instruction set computer
- designed to reduce the execution time by simplifying the Instruction Set Computer
- each instruction requires only one clock cycle (fetch, decode, execute)
-

Architecture

- it uses highly optimised set of instruction
- it is used in portable devices like iPod, smartphone and tablets
-



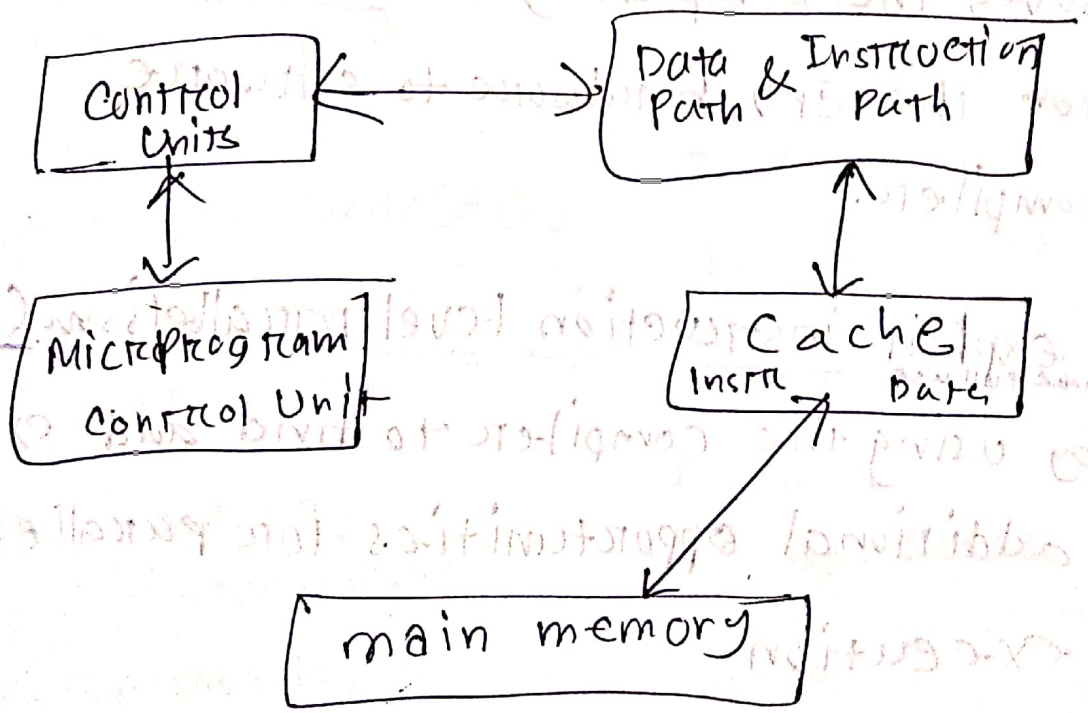
Characteristics of RISC:

- consists of simple instructions
- various data type formats supported
- utilizing simple addressing modes
- large number of registers + transistors
- only one cycle execution time.
- having the load and store instructions to access the memory location.

CISC [Complex Instruction Set Computer]

- designed to minimise the num of instruction per program
- ignoring the number of cycles per instruction
- used in desktop, laptops.
- compiler has to do little work to convert high level language to machine language as the length of the code gets shorter.
- very little RAM is required to store instruction.

Architecture —



Characteristics —

i. variety of addressing modes
ii. large num of instructions are minimised embedding them int small number
iii. several cycles maybe required to execute one instruction.
iv. instruction decoding logic is complex.

EPIC (Explicitly Parallel Instruction computing) is a 64-bit microprocessor instruction set jointly defined and designed by Hewlett Packard (HP) and Intel that provides up to 128 general and floating point unit registers and uses speculative loading, predication and explicit parallelism to acomplish its computing tasks.

Wait, what was that ?!

- moves the complexity of instruction scheduling from the CPU hardware to software compiler.
- exploit (make full use) instruction level parallelism (ILP) by using the compiler to find and exploit additional opportunities for parallel execution
- frees up space and power for other functions.

Difference between Min & Max mode in µP

Minimum Mode

1. Only 8086 is available

2. MN / $\overline{MX}$
   - to indicate minimum mode
   1 / $\overline{1}$ ⇒ 1/0

3. Address Latch Enable (ALE) is given by 8086 as it is only processor in the circuit.

4. $\overline{DEN}$ (Data enable)
   DT / $\overline{R}$ (Data transreceiver) given by 8086 itself.

5. Direct control signals given by 8086

Maximum Mode

1. Multiples processors with 8086 like 8087 & 8089 (coprocessors)

2. MN / $\overline{MX}$
   = 0 / $\overline{0}$
   = 0 / 1

3. ALE is given by 8288 bus controllers as there can be multiple processors in a circuit.

4. DT / $\overline{R}$
   by 8288

5. Instead of control signals, each processor generates status signals called $\overline{S_2}$ $\overline{S_1}$ $\overline{S_0}$

6. control signals decoded by 3:8 decoder.

6. status signals decoded by controller like 8288

8086 µP [40 Pins]

16 bit µP : ~~AD0 — AD~~
$AD_0 - AD_{15}$

data bus — 16 bit
address bus — 20 bit

22 - READY
21 - RESET
CS: FFFF
IP: 0000
DS, ES, SS
0000
All flags will be clear
stack will be empty

A_{16}/S_3
A_{17}/S_4
A_{18}/S_5
A_{19}/S_6
$\overline{BHE}/S_7$

Bus High Enable
- to distinguish between low byte and higher bit for 16 bit

CLK (19)
heartbit of CPU
- connected to 8284 clock genera

NMI (17)
- Non Maskable int (execute that first)

INTR (18)

$\overline{BHE}$	A_0	
0	0	16 bit ($D_0 - D_{15}$)
0	1	Upper 8 bit ($D_8 - D_{15}$)
1	0	lower 8 bit ($D_0 - D_7$)
1	1	idle bus

LAZY BOY

SIAM