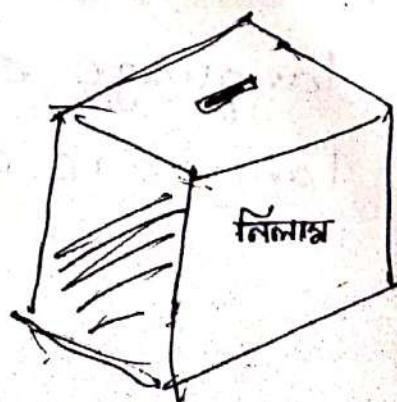


Difference between Assembly Language and High Level Language

Definition of Assembly Language

A low-level programming language which uses symbols and lack variables and functions and which ^{works} 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 ~~to be~~ 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 perform communication 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 at a physical address which is not possible in high level lang.
8. Assembler 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

High level programs are less efficient than assembly language programs.

11. High level language programmers 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

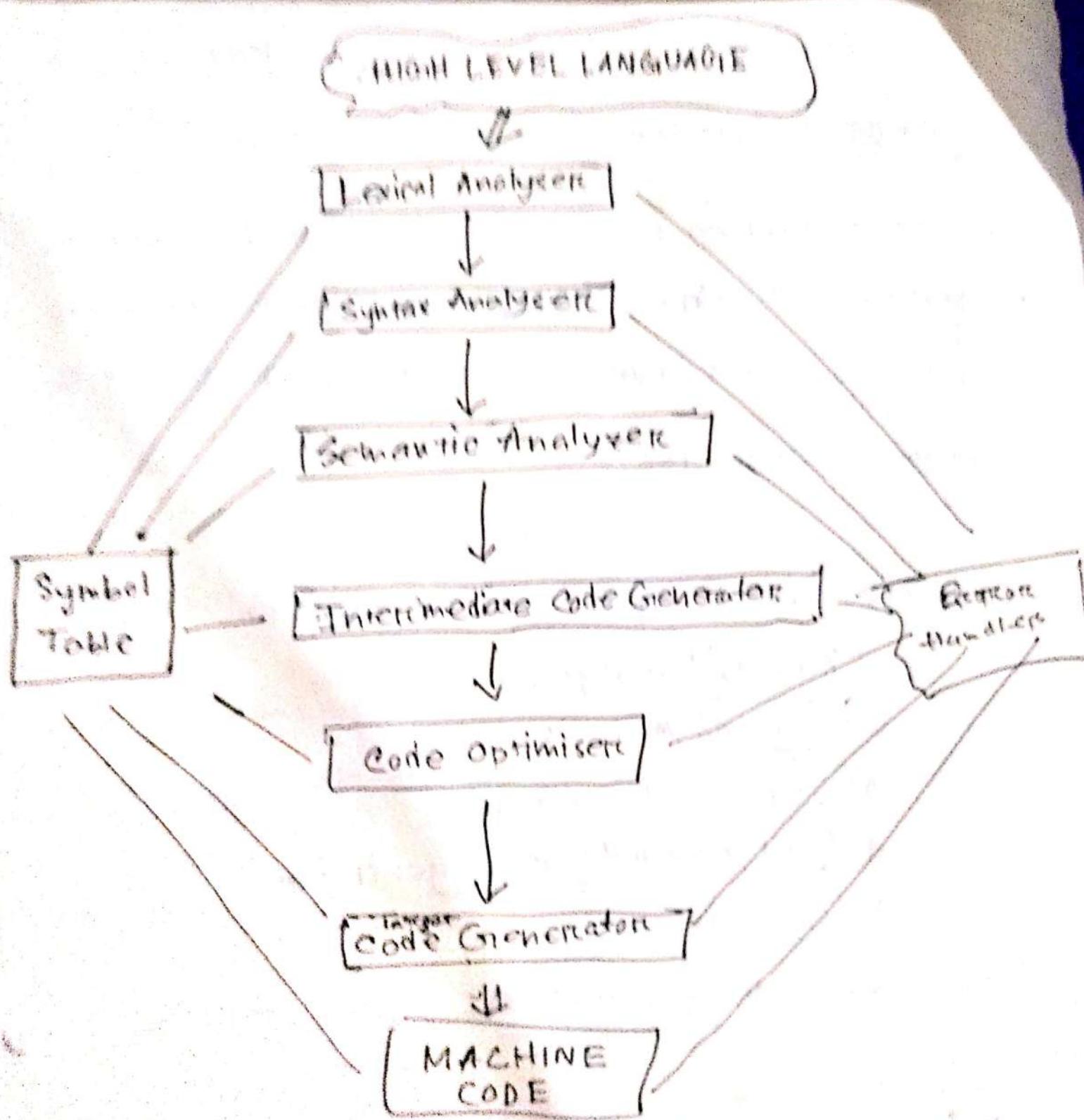
The compilation process is a sequence of various phases. Each phase takes input from previous stage, has its own representation of source program, and feeds +

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 ✓

~~Lexical Analyzer~~

~~Syntax Analyzer~~



Symbol Table:

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

The compiler has two modules namely front end and back end. Front end consists of Lexical Analyser, semantic analyser, syntax analyser 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 Generation

- Generates intermediate code, that is a form 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

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

allocate 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 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/assembly 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 errors.

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

are generated. The linker 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 -

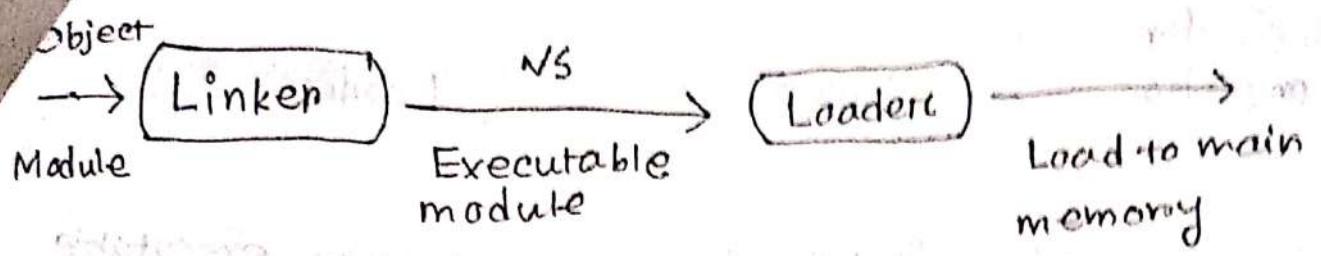
i. Linkage Editor

- 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 compilers, 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 combines them to generate executable module. On the other hand, the loader loads this executable module to the main memory for execution.

Comparison between LINKER & LOADER

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

Basics for compilation

Linker

Loader

2. Input

It takes input as
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~~ 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.

Interpreter

1. Translates the program one statement at a time.

2. Takes less amount of time to analyze the source code but overall execution time is slower.

3. No intermediate object code is generated.
Memory Efficient.

4. Continues translating the program until the first error is met. Easy Debug.

5. Python, Ruby

Compiler →

1. Scans the whole programs and translates it as a whole into machine code.

2. Takes large amount of time to analyze the source code but overall execution time is comparatively faster.

3. Generates intermediate code which further requires linking hence requires more memory.

4. Generates error msg after scanning the whole program.

5. C / C++



• MODEL SMALL code ଏହି ଆଇଜ୍ ଫର୍ମ୍ ହେଁ । ଏ ପକାଗୋଟିଏ

ମୁଣ୍ଡନ ଆଛେ ।

• STACK 100H

• DATA

; Variables are dedicated

• 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 ; দুটি ডেক্লারেশন - অন্তর্বর্তী

■ 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 ফিল হবে,

NUM2 DW ?

CHAR DB '#'

STR DB "My Name is EMU8086"

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

// কোটে ডিস্ট্রিব্যুশন এক প্রয়োগ করলে এই
ক্ষণাত্মক অস্তি।

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

MOV AH, 2 ; Print Mode
MOV BL, NUM1
INT 21H

0 (48 ASCII value
'0' এর মৌলিক মান)

□ New line প্রিণ্ট করা

| MOV AH, 2 ; Print Mode
| MOV DL, 0AH ; New line
| INT 21H
| MOV DL, 0DH ; Carriage Return
| INT 21H

□ ফিল্ড প্রিণ্ট করা

MOV AH, 9
LEA DX, MSG
INT 21H

□ .MODEL SMALL

MEDIUM

COMPACT

LARGE

HUGE

কোনো অ্যারেজের উপর জিপেড় রয়েছে।

प्रा-डिएटाल क्यारेक्टर अप्लॉट

MOV AH, 3 ; इनपुट डिस्ट्री !
INT 21H ; ऑपरेटोर AL पर वापस !

प्रिंट रूट -

MOV AH, 2 ; प्रिंटिंग डिस्ट्री
MOV DL, AH AL,
INT 21H

□ ADD प्रा- फाँकाना लिटि

A, B - ऑपरेटोर नियम A+B = C में लगाया।

डेक्स्ट्रू

- 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, 0

MOV BL, 0AH

INT 21H

MOV BL, 0DH

INT 21H

MOV AH, 2

MOV DL, BH

INT 21H

MOV AH, 4CH

INT 21H

MAIN ENDP

END MAIN

JJAMP JECOM

MURGEM

IPDAWNSO

IPRAS

IPRAS

IPRAS

IPRAS

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 "NEGATIVE"
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 } 7. AT&AD.  
MOV DL, BL }  
INT 21H }
```

```
MOV AH, 2 } 5. C0DE.  
MOV DL, BH }  
INT 21H }
```

□ INCLUDE 'EMU8086.INC'

PRINT = Print a string without newline

PRINTN = Print a string with new line

Some ENDER

□ Unconditional Jump - JMP

Relative

□ Conditional Jump

Relative & Absolute

CMP COMPARED_TO, COMPARED_WITH

4 CMP BX, C

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 ; अंतर का मूल्य  
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 ENDED
```

For loop / Loop using JMP

$$J + 2 + 3 + \dots + n = ?$$

Solution:

```
INCLUDE 'EMU8086.INC'
```

```
.MODEL SMALL  
.STACK 100H
```

```
.DATA
```

```
MSG11 DB "ENTER N: "  
MSG12 DB "? "  
MSG13 DB "THE SUM IS: "  
MSG14 DB "SUM DB 0"  
N DB ?
```

```
.CODE
```

```
MAIN PROC
```

```
MOV AX, @DATA  
MOV DS, AX
```

```
MOV AH, 9  
LEA DX, MSG11
```

```
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, 10
```

```
TOP:
```

```
CMP BL, N
```

```
JGE EXIT_LOOP
```

```
ADD SUM, BL
```

```
INC BL
```

```
JMP TOP
```

EXIT_LOOP:

```
MOV AH, 9  
LEA DX, MSG1  
INT 21H
```

```
MOV AH, 2  
MOV DL, SUM  
INT 21H
```

```
MOV AH, 4CH  
INT 21H
```

```
MAIN ENDP  
END MAIN
```

NESTED LOOP

Print square like

```
INCLUDE 'EMU8086. INC'  
.MODEL SMALL  
.STACK 100H  
.DATA  
N DB ?  
.CODE  
MAIN PROC  
MOV AX, @DATA  
MOV DS, AX  
  
MOV AH, 1 ; INPUT MODE  
INT 21H  
SUB AL, 48 ; MAKING NUM  
MOV AL, N  
  
MAIN ENDP
```

MOV BH

```
MOV BH, 0 ; PRINT MODE  
;newline  
MOV AH, 2 ; OUTPUT  
MOV DL, OAH  
INT 21H  
MOV DL, ODH  
INT 21H
```

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

LOOP1:

```
CMP BH, N  
JGE LOOP1_EXIT
```

```
MOV BL, 1 ; (j=j)  
LOOP2:
```

```
CMP BL, N  
JGE LOOP2_EXIT  
INT 21H ; print *  
INC BL ; j++  
JMP LOOP2
```

LOOP2_EXIT:

```
DEC ; newline  
MOV AH, 2  
MOV DL, OAH  
INT 21H  
MOV DL, ODH  
INT 21H
```

```
INC BH (i++)  
MOV DL, '*'  
JMP LOOP2
```

Loop1_EXIT:

```
MOV AH, 4CH  
INT 21H
```

```
MAIN ENDP  
END MAIN
```

Logical Operations

AND

Binary:

~~MOV BH, 1111B ; 15D~~
~~AND BH, 0100B ; BH = 0100B~~

; Printing

~~MOV AH, 2 } INT 21H~~
~~MOV DL, BH } INT 21H~~

OR Operator

~~MOV BH, 1111B~~

~~MOV BH, 0100B ; BH~~

~~OR BH, 0100B ; BH~~

~~MOV BH, 0101B~~

~~OR BH, 0010B ; BH~~

~~ADD BH, 48H // character~~

XOR Operator

~~XOR BH, 0101B~~

~~XOR BH, BH // BH < 0~~

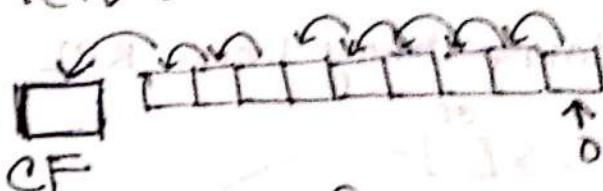
NOT Operation

~~MOV BH, 11110000B~~
~~NOT BH~~

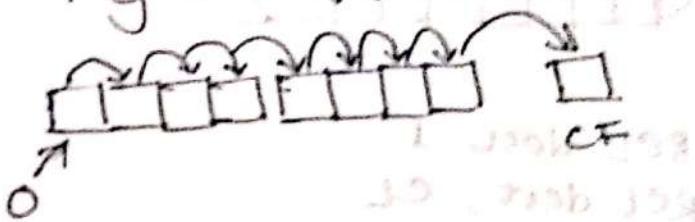
~~MOV BH, 2 } INT 21H~~
~~MOV DL, BH } 00001100~~

SHIFT INSTRUCTION

left shift

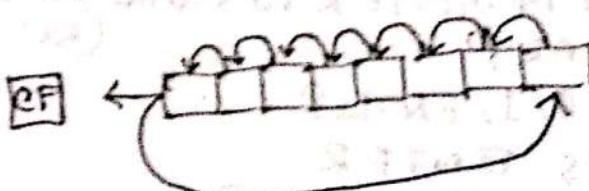


right shift



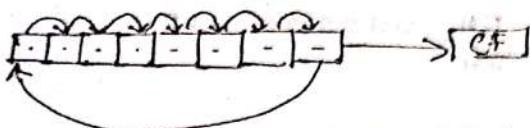
Rotate Left

just like left shift, but
MSB is shifted to rightmost



ROL destination, 1
 ROL destination, CL

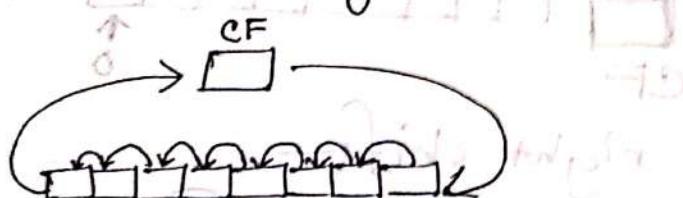
■ Rotate Right



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 FACTORY

CLEAR A REGISTER TO STORE DATA (BX)

LOOP START

INPUT 0/1/ENTER

IF IT'S ENTER

IF ENTER END LOOP

BX SHL INPUT TO LSB

SAVE THE INPUT TO LSB

INCLUDE 'EMU8086.ASM'

MODEL SMALL

STACK 100H

.DATA

; Variables

.CODE

MAIN PROC

MOV BX, 0

MOV AH, 1 ; Input mode

FOR J:

INT 21H

CMP AL, ODH

JZ END-FORJ

SUB AL, 18H

SHL BX, 1

OR BL, AL

JMP FORJ

END-FORJ:

MAIN ENDP

Mov AH, 4CH

INT 21H

MAIN ENDP

END MAIN

of binary input and printing
output (Sayef Regah video - 20)

```
INCLUDE INTRINSIC.INC  
.MODEL SMALL  
.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, ODH
```

```
JE END-FOR1
```

```
SUB AL, 48
```

```
SHL BX, 1
```

```
OR BL, AL
```

INC COUNT

```
JMP FOR1
```

END FOR1

PRINTN {
MOV AH, 9
INT 21H
MOV AH, 0AH
INT 21H }

```
MOV CL, 16  
SUB CL, COUNT  
REL BX, CL
```

{
XOR CH, CH
MOV CL, COUNT }

} MOV AH, 2

FOR2:

```
RELOD BX, AX  
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, 4CHD

```
INT 21H } JC
```

MAIN ENDP } END

END MAIN } END

MAIN ENDP ;

SP ? JA 0000

TJMP 0000

~~Hex~~

Hex Input Output (Sayef Readh - 212)

INCLUDE 'EMUS8086.INC'

MODEL SMALL
STACK 100H

DATA

; Vars

CODE

MAIN PROC

MOV AH, 00H
INT 21H

MOV BX, 0

MOV CL, 10H

INT 21H

PRINT "ENTER A HEXNUM:"

INT 21H

MOV AH, 01H ; INPUT MODE

INT 21H

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:

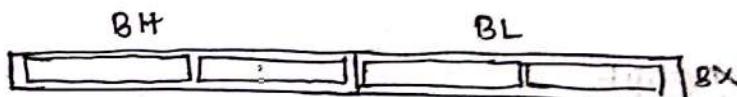
SHL BX, CL

OR BL, AL

JMP FOR1

END FOR1:

; Outputting hex



ALIAS @ 00401000

XOR CH, CH

MOV CL, 9

MOV AH, 2

00401000

FOR2:

MOV DL, BH

SHR DL, 4

SHL BX, 4

HEX TIME

00401000

CMP DL, 48

JGE LETTER2

; OTHERWISE DIGIT

ADD DL, 48

INT 21H

JMP END2

END2

LETTER2:

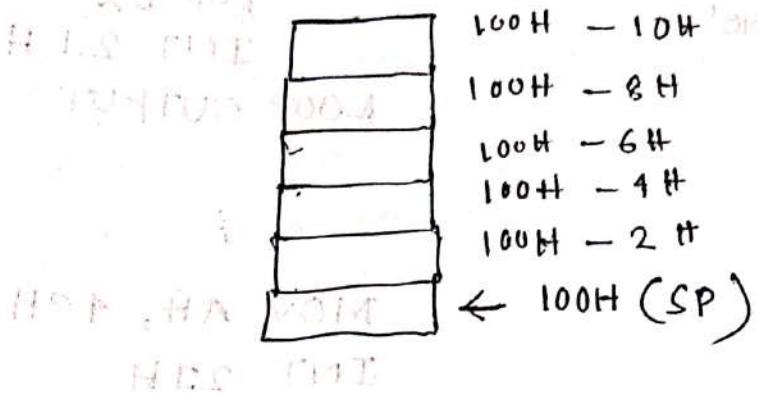
ADD DL, 55

INT 21H

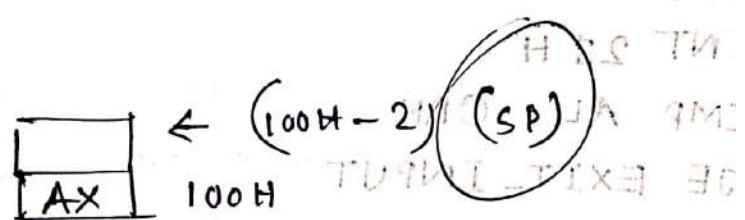
END2;

LOOP FOR2

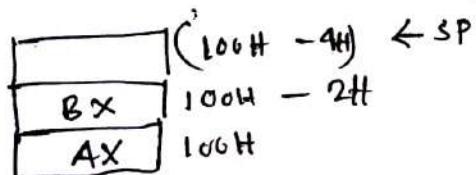
INTRODUCTION TO STACK



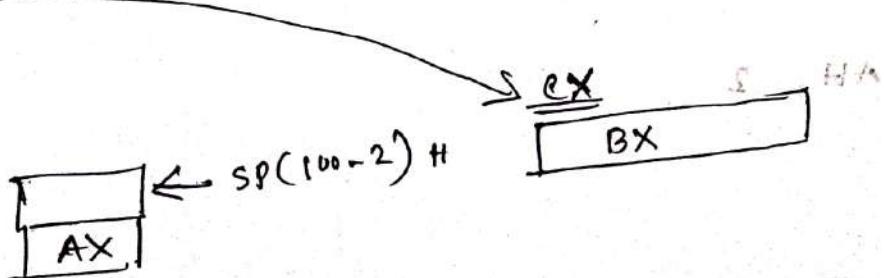
Push AX



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 21 H
```

```
CMP AL, 0DH
```

```
JE EXIT_INPUT
```

```
PUSH AX
```

```
INC CX
```

```
JMP INPUT
```

EXIT_INPUT:

PRINTN

PRINT "OUTPUT: "

```
MOV AH, 2
```

~~MAIN PF~~
OUTPUT:

```
POP DX
INT 21 H
LOOP OUTPUT
```

```
(92) Root →
MOV AH, 1CH
INT 21H
```

```
MAIN ENDP
```

END MAIN

Introduction to Function

return-type, function-name (parameters) {

// actions

return-type

function-name

→ in high level languages

→ in assem

→ END-MAIN indicates the whole program is finished. So functions should be written ~~in~~ before END-MAIN

FUNCTION-NAME PROC
// STATEMENTS
RET
FUNCTION-NAME ENDP

// calling
CALL FUNCTION-NAME

function definition

function call

function call

Sample Program of using function

Ch Multiplication

INCLUDE 'EMUG086.INC'

- MODEL SMALL
- STACK 100H

.DATA

Variables

CODE

MAIN PROC

CALL FN

INT 21 H

MAIN ENDP

FN · PROC

PRINTN "Hello World"

PRINTN "I am the function"

RET

EN ENDP

END MAIN

Output:

Hello World

I am the function

I am the main fn

	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1
.	1	1	1	1	1	1	1	1	1
.	1	1	1	1	1	1	1	1	0
.	0	0	0	0	0	0	0	0	0
.	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	0	0

```
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
```

```
PRINT N
```

```
PRINT "SECOND NUMBER: "
```

```
INT 21H
```

```
MOV CL, AL SUB CL, 48
```

```
PRINT N
```

~~MULTIPLICATION~~~~MOV DL,~~

num1 8 bit

num2 8 bit



num3 16 bit
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 END

END MAIN

Multiplication Function (Sayef Reyadh - 27)

MUL and IMUL - function

MUL - for signed numbers

IMUL - for unsigned numbers

? AX : AL

multiplier/multiplicand এবং গুণাত্মক একটিবারে AX এ
ফিল্ড হবে BX \ AX = AX : AX : AX : AL

MOV AX, 10
MOV BX, 9
MUL BX

$10 \times 9 = 90 = 5A_h$

[005A]
+1cx
DX

; DX : AX = BX * AX

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

AL : VOM

DL : VOM

BL : VOM

28. DIV & IDIV

dividend = 5

divisor = 2

resultant sum = sum of dividend + divisor

~~MOV AX, 5~~

standard operand width = 16bit

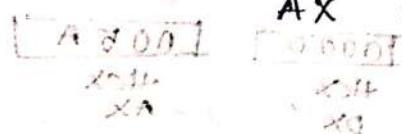
~~MOV BX, 2~~

standard operand width = 16bit

DIV BX ; DX : AX = AX / BX

DX = Remainder

AX = Result / Quotient



IDIV

XA * XB = XA : XB +

MOV AX, 5

MOV BX, -2

IDIV BX ; DX : AX = AX / BX

XA * XB = AH HA : BL BL
rem 2's complement

MOV AL, 5

MOV BL, 2

DIV BL ; BT = A

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 char
```

```
MOV AH, 4CH
```

```
INT 21H
```

```
MAIN ENDP
```

```
:S07
```

```
IN , HA VONI  
HEZ INT
```

```
HAD , JA 9MS
```

```
S07 - 9MS 4E
```

```
JA , JA 9MS
```

```
JA , H VONI
```

```
OL , JA VONI
```

```
JL , JL 9MS
```

```
IN , JA , DIA
```

```
JA , JA , VONI
```

```
S07 9MS
```

```
IN , JA , DIA
```

```
JA , JA , VONI
```

```
S07 9MS
```

```
IN , JA , DIA
```

```
JA , JA , VONI
```

XOR CX, CX ; COUNTER

XOR DX, DX

MAIN TU QTJ. JAM TO H

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
JMP FOR

END-FOR:

RET

DECIMAL-INPUT ENDP

END MAIN

Outputting Decimal

```
INCLUDE 'EMUG8086.INC'
```

```
.MODEL SMALL
```

```
.STACK 400H
```

```
.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

JBE 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 // set CX to 0 for loop
MOV CL, COUNT

MOV BL, 10

XOR AH, AH // remainder divisor
MOV AL, N // remaining number

FOR2:

DIV BL // AL = AL / BL

XOR DX, DX

MOV DL, AH

ADD DL, 48 // DL + 48 = STCR

PUSH DL

LOOP FOR2

XOR CX, CX

MOV CL, COUNT

FOR3:

MOV AH, 2

POP DX

INT 21 H

Loop FOR3

3.1. Array Introduction

NUM DB 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
THUS 10 VDM

NUM2 DB 10 DUP (?)

NUM3 DB 10 DUP (0) \leftarrow first element @
0 factor initialized

TST DB 5, 4, 3 DUP (2)
 \leftarrow size of dup

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

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

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

XO XO 40X
THUS 10 VDM

S HA VDM

XO 40X
THUS 10 VDM

32. Array Index

INCLUDE 'EMU8086.INC'

.MODEL SMALL
.STACK 100H

.DATA

NUM DB

1, 2, 3, 4, 5,

NUM2 DW

61, 62, 63, 64,

65, 66, 67, 68, 69

60, 61, 62, 63, 64, 65, 66, 67, 68, 69

60, 61, 62, 63, 64, 65, 66, 67, 68, 69

DATA SEGMENT

NUM DB 73, 00H,
NUM2 DW 0000H

AT&S.

.CODE

MAIN PROC

MOV AX, @DATA

MOV DS, AX

LEA SI, NUM

LEA SI, NUM2

MOV CX, 10

MOV AH, 2

FOR:

MOV DL, [SI] ; SI in DS

ADD DL, 10

INT 21H

ADD SI, 1

LOOP FOR

MOV AH, 4CH

INT 21H

MAIN ENDP

END MAIN

MAIN ENDP

Based & Index Addressing Mode

see 33 34

```
INCLUDE 'ENVIRON.INC'
MODEL SMALL
STACK 100H
DATA
    NUMBERS DB 1, 2, 3, 4, 5, 6, 7, 8, 9, 20
    NUMBER2 DW CS, 00, CT, 00, 00
CODE
```

MAIN PROC

```
MOV AX, @DATA
MOV DS, AX
```

```
MOV CX, 10    ; FOR Looping 0 to 9
```

```
XOR BX, BX    ; BX=0 (i=0)
```

```
MOV AH, 2
```

FOR:

```
MOV DL, NUMBER1[BX]
```

```
INT 21H
```

```
ADD DL, 10
```

```
INT 21H
```

```
ADD DL, 10
```

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:
```

```
JNT 291H
```

```
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
```

```
=
```

Assembly & Micro

One of the control flags is the direction flag (DF)

DF is implemented by two index registers -

SI
DI

When, DF = 0

SI, DI goes from left to right

When DF = 1

SI, DI goes from right to left

clear direction flag

CLD

// clear direction flag

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

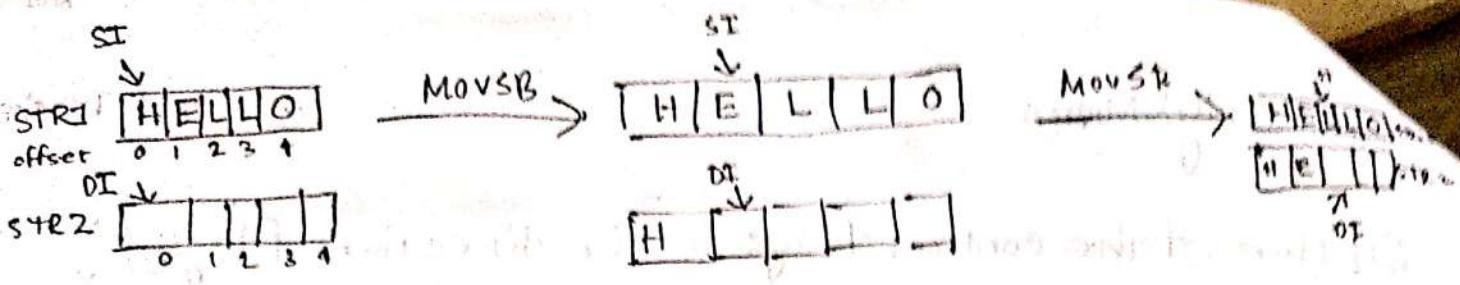
LEA SI, STR1

LEA DI, STR2

CLD // left to right

MOVSB // moves the first byte

MOVSB // moves the second byte



** MOVSB / MOVSW

— instructions to permit memory - memory operation.

□ CLD // left to right

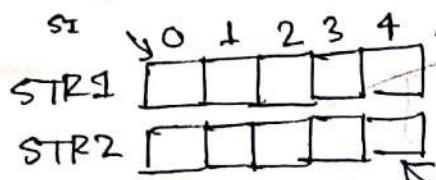
LEA SI, STR1

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 five times

LOOP MOV:

MOVSB // 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	O
A	E	L	L	O
A	A	L	L	O

- 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		O	L	A
------	--	---	---	---

AL

?

O	L	A
O	L	A

O

For moving a word — MOVSW

ARR DW $\boxed{10 \ 20 \ 40 \ 50 \ 60 \ ?}$
0 2 4 6 8 10

Insert 30 between 20 and 40

Output will be like: ARR DW $\boxed{10 \ 20 \ 30 \ 40 \ 50 \ 60}$
0 2 4 6 8 10

Solution:

.MODEL SMALL

.STACK 100H

.DATA

ARR DW $\boxed{10 \ 20 \ 40 \ 50 \ 60 \ ?}$
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~~

~~MOV ARR+4, 30~~

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

LEA SI, ARR+8

LEA DI, ARR+10

STD ;right to left

MOV CX, 3

REP MOVSB

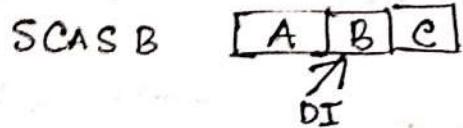
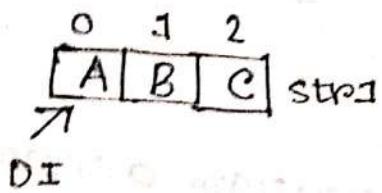
DI, ARR+4

MAIN ENDP

END MAIN

III SCASB

- Compares content of AL with content of ES:DI



AL
B ZF = 0 . Not found!



AL
B ZF = 1 . Yo! Found.

~~III Comparing strings~~

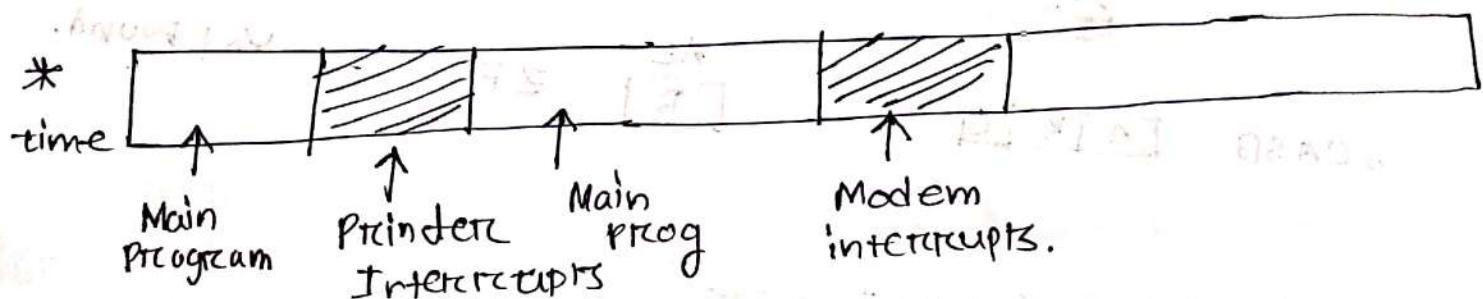
Interrupt in 8086

* 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 temporarily halt during program execution and allow peripheral/external devices to access microprocessors.

* Interrupt Service Routine 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
ii. software interrupt

(Can continue with main prog)
Maskable interrupt
Non maskable interrupt
256 types of interrupt

INTERRUPTS

HW interrupt caused by any peripheral devices.

* HW interrupt pins

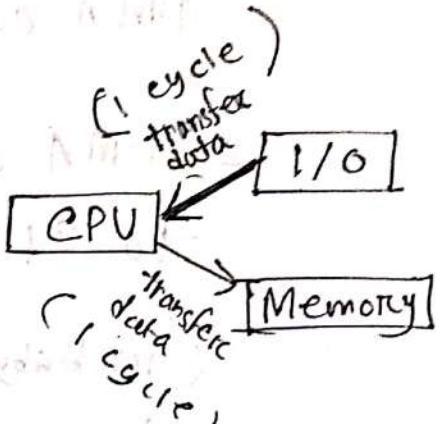
- NMI (Non maskable interrupt)
(17) cannot continue with another program
- INTR
- INTA (Interrupt Acknowledgement)

- ☐ Software interrupts
 - INT (interrupt instruction with the type number)

eg. INT3 — break point of interrupt
INT0 — interrupt of overflow

- ☐ DMA Controller (Direct Memory Access)

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

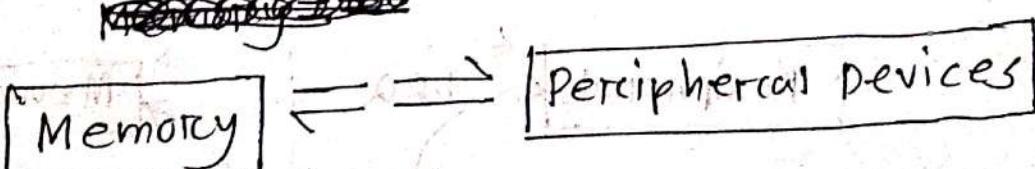


- ☐ To make the data transfers fast — DMA

A cycle contains

- fetching
- decoding
- executing

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



Q. How is DMA operations performed?

1. Peripheral device wants to send data to memory

2. first ^{peripheral} device has to send DMA request to DMA controller.

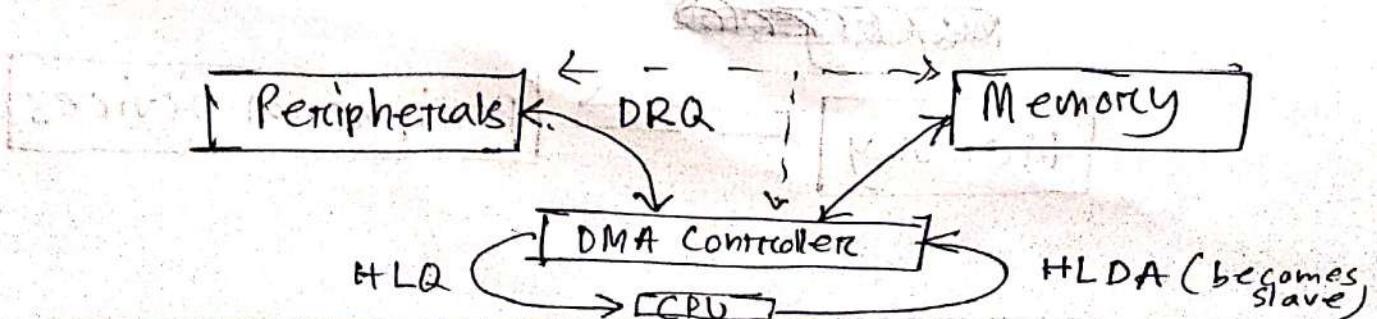
2. DMA controller send HLQ (Hold request) to CPU.

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

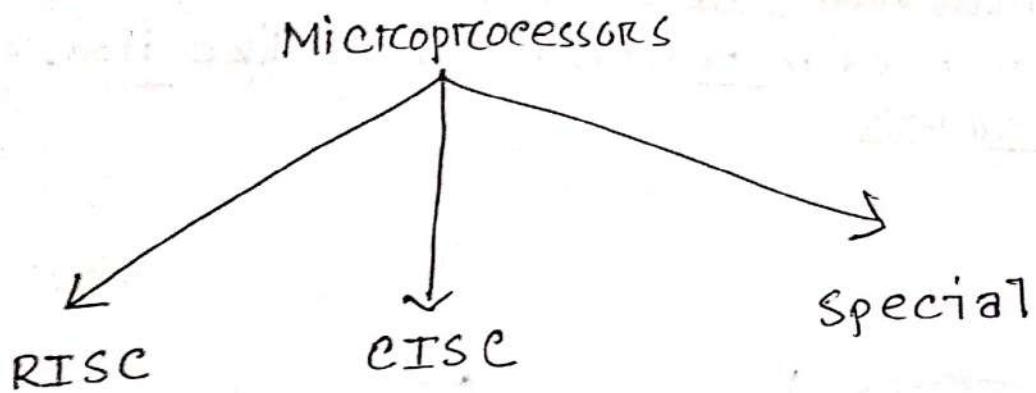
And waits for the CPU to send hold acknowledgement (HLDA)

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

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



RISC [Reduced Instruction set Computer]

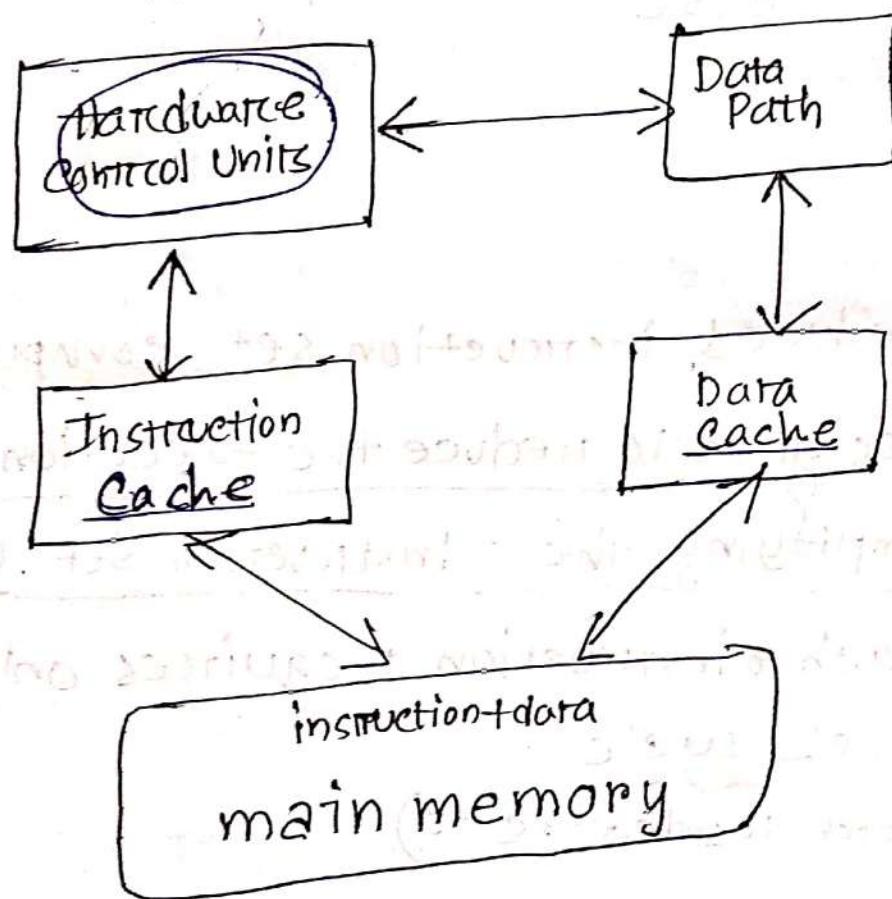


RISC :

- 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:

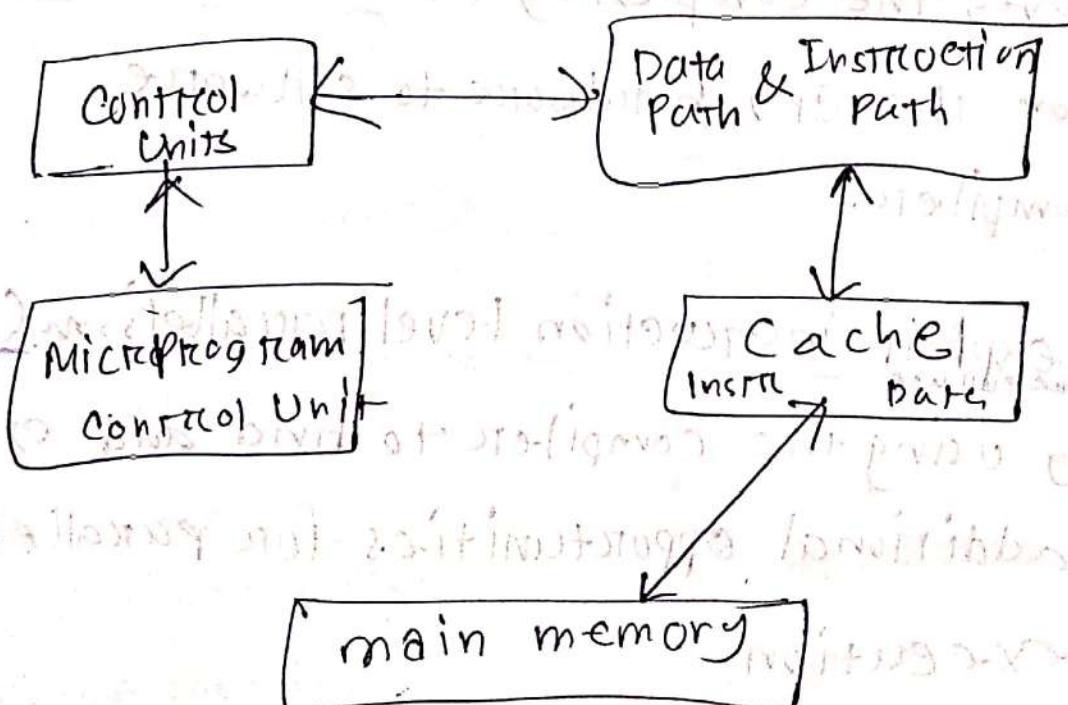
- consists of simple instructions
- various data type formats supported
- utilizing simple addressing modes

- large numbers of registers
- 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 number 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 number of instructions are minimised by embedding them in small number

iii. several cycles may be required to execute one instruction.

iv. instruction decoding logic is complex.

EPIIC (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 accomplish its computing tasks.

Wait, what was that?!

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

Difference between Min & Max mode in 8086

Minimum Mode

1. Only 8086 is available

2. MN / \overline{MX}

→ indicates minimum mode

$$1 / \overline{1} \Rightarrow I/O$$

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 transceiver) given by 8086 itself.

5. Direct control signals given by 8086

Maximum Mode

1. Multiples ~~one~~ processors with 8086 like ~~multiple~~ 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 ~~8086~~ 8288

5. Instead of control signals, each processor generates status signals called S_2, S_1, S_0

6. control signals decoded
by 3:8 decoders

6. status signals decoded
by controllers like 8288

8086 MP [40 Pins]

16 bit dup : ~~AD₀ - AD₁₅~~
~~AD₀ - AD₁₅~~

data bus — 16 bit

address bus — 20 bit

22 - READY

21 - RESET

CS: FFFF

IP: 00 00

DS, ES, SS
0000

A7 flags will
be cleared

stack will be
empty

A₁₆ / S₃

A₁₇ / S₄

A₁₈ / S₅

A₁₉ / S₆ SIAM . E

BHE / S₇

Bus High Enable

— to distinguish

between low byte and

high byte for 16 bit

CLK (19)

heartbit of CPU

— connected to

• furnishing output 8284 clock generator

NMI (17)

— Non Maskable int
(execute + halt firm)

INTR (18)

BHE A₀

0

16 bit (D₀ - D₁₅)

0

Upper 8 bit (D₈ - D₁₅)

1

Lower 8 bit (D₀ - D₇)

1

idle bus LAZY BOY

SIAM