Certificate Course on

“Microprocessor and Assembly Language Programming”

PICT, Pune.

Prof. R. V. Bidwe PICT, Pune.
rvbidwe@pict.edu
www.ranjeetbidwe.in

Agenda

- Introduction to Computer
- Different Components in Computer
- Microprocessors
- Functions of Microprocessors
- Assembly Language Programming

Introduction to Computer

- Computer have 3 basic **Properties**:

1. For everything, we have program.

2. Everything is stored somewhere in memory. [May be RAM, ROM or other kind of memory]

3. Everything stored in memory will have a unique address. [This address is called as **Physical Address**]

- Computer have 2 basic **Rules**:

1. Processor never perform operation on actual data directly.

   [It makes copy of the data and performs operation.]

2. Users will never be given Physical Address directly by system.

   [Address will always be given in format of Base Address and Offset i.e.. **Logical Address**]

Different Components in Computer

- Following are the different components present in system.
  1. **Microprocessors**
  2. **Microcontrollers**
  3. **Memories**
     I. RAM
     II. ROM
     III. Cache
     IV. Registers
  4. **Timers**
  5. **Input/ Output Ports**
  6. **Communication Ports**
  7. **Interconnect Buses**

Microprocessor & Microcontrollers

- A microprocessor, sometimes called a ***Logic Chip***, is a computer processor on a microchip.
- It is also called as **"Heart of Computer."**
- The microprocessor contains all, or most of, the **Central Processing Unit (CPU)** functions.
- A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called ***Registers***.

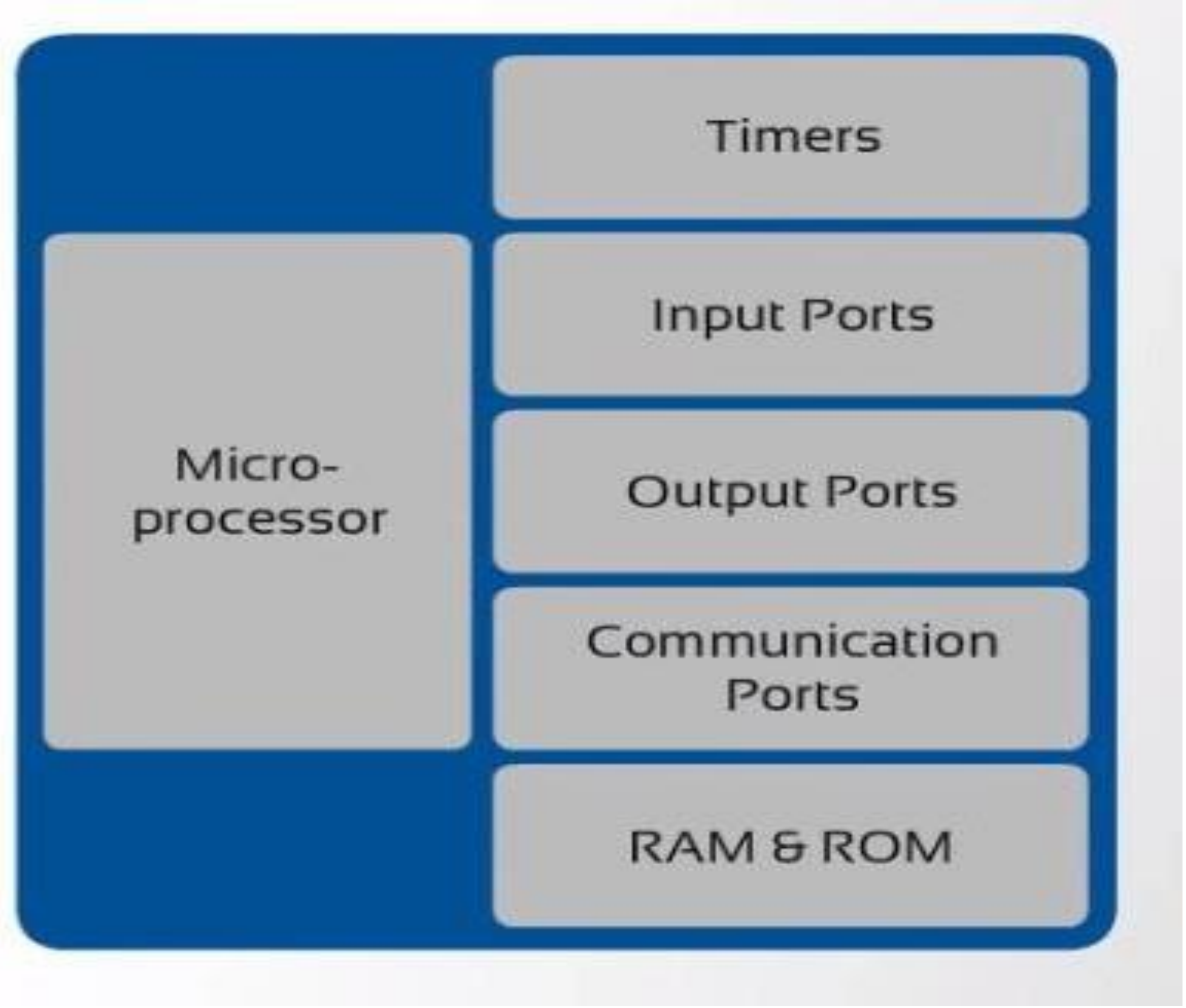
- Typical microprocessor operations include **adding, subtracting, comparing two numbers, and fetching numbers** from one area to another.

- These operations are the result of a set of instructions that are part of the microprocessor design.

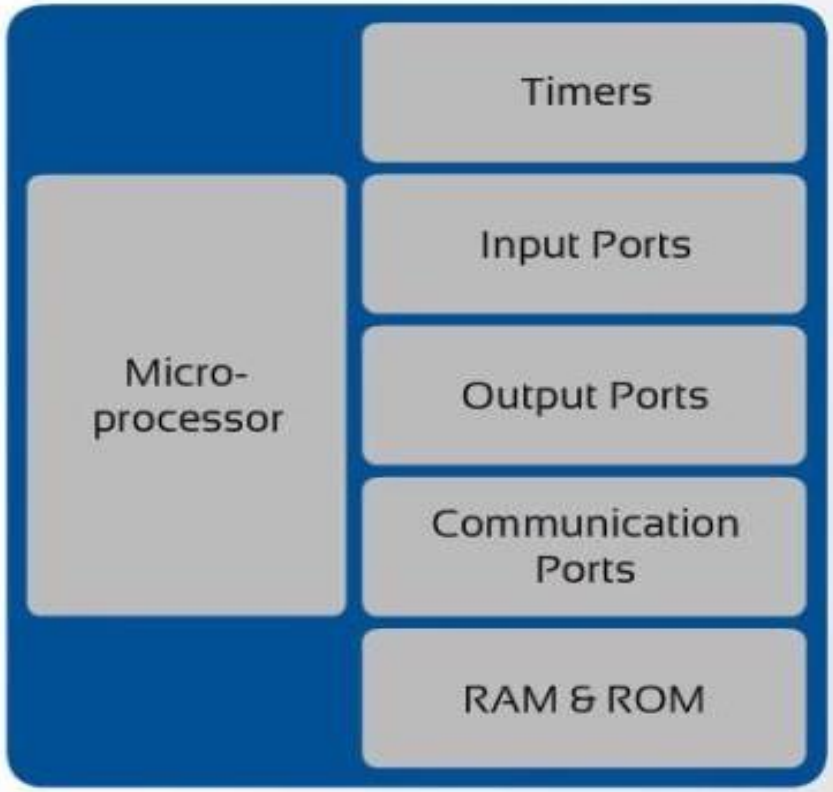
Three basic characteristics to differentiate Microprocessors:

- **Instruction set**: The set of instructions that the microprocessor can execute.
- **Bandwidth:** The number of bits processed in a single instruction.
- **Clock speed:** Given in megahertz (MHz), the clock speed determines how many floating point instructions per unit time the processor can execute.
  - Performance of processors are calculated by its clock speed. i.e. how fast processor can complete the execution.

Microprocessor Vs. Microcontroller



Microcontroller



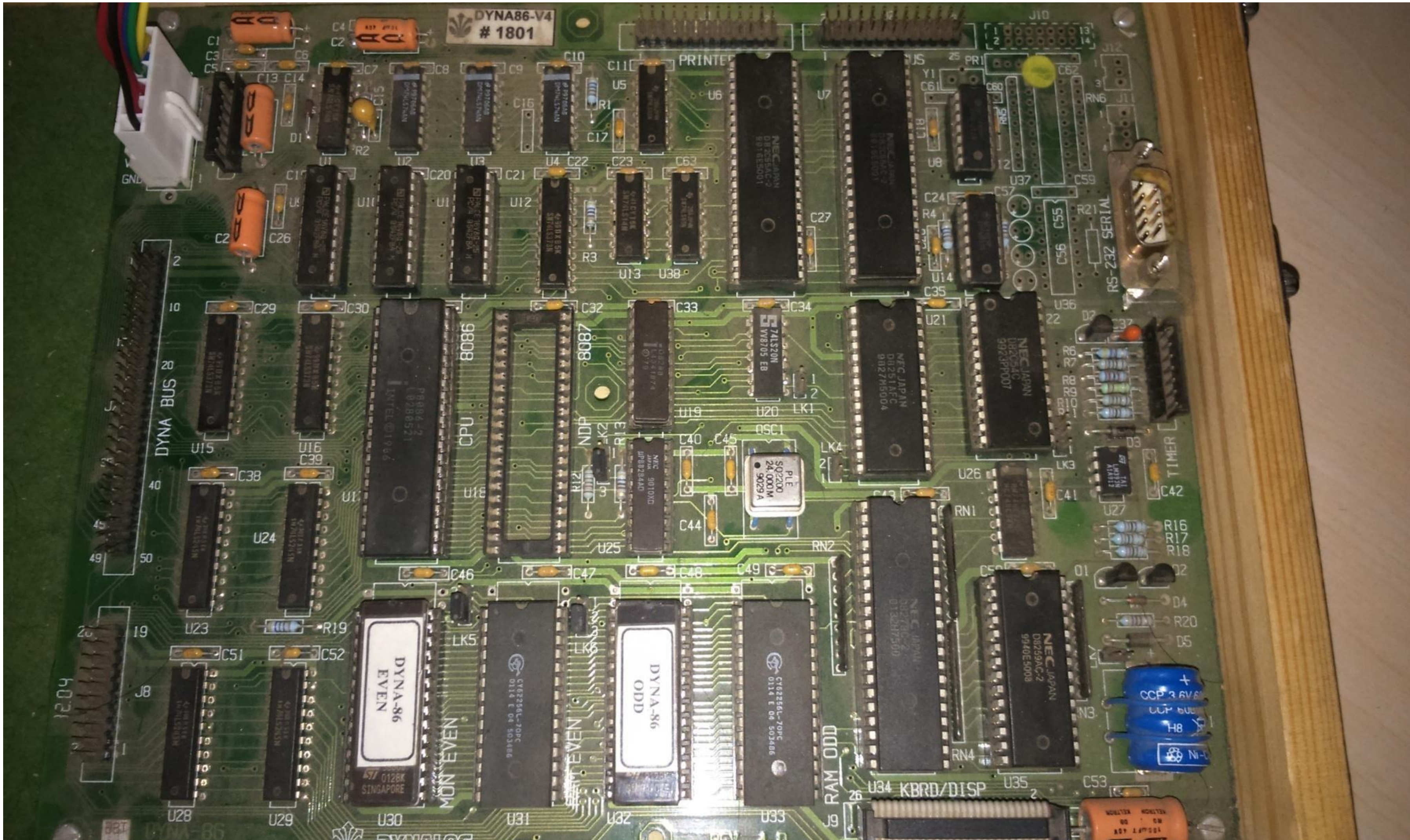
Microcontroller, as an Integrated Circuit (IC), is complex than a General Purpose Processor.

Differences between a Microprocessor and a Microcontroller:

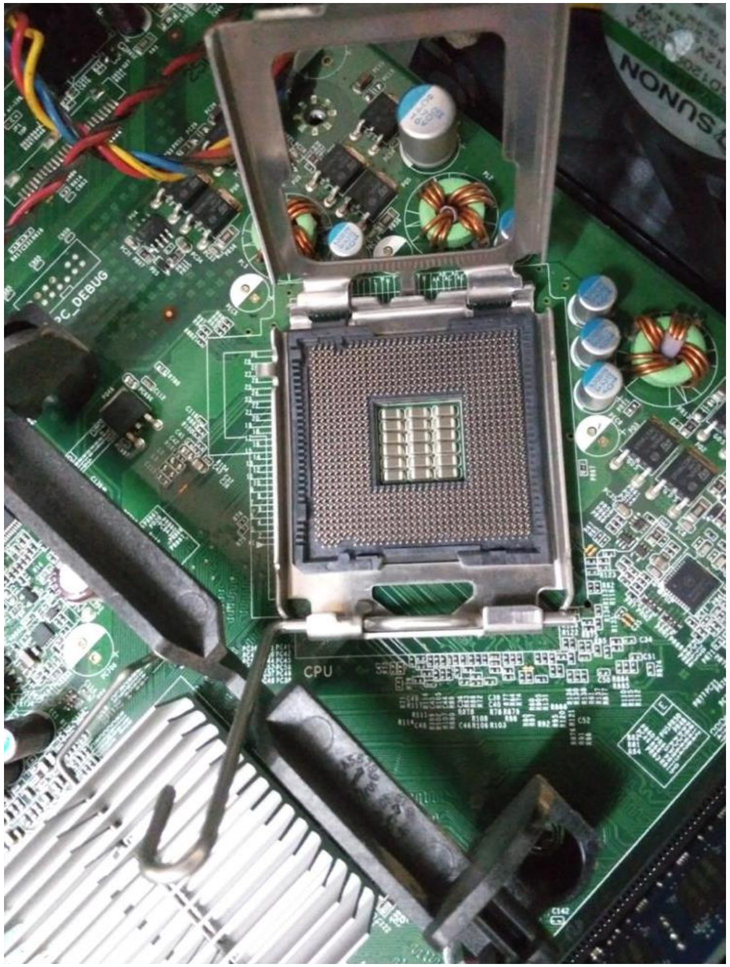
- Multipurpose
- Contains primarily the CPU
- System costs are higher
- Higher Clock speed
- Can be constantly reprogrammed as required

- Specific usages
- Contains the CPU and many peripheral devices
- System costs are lower
- Cannot operate at higher Clock speed
- Requires programming only once for a particular application

How It Looks: 8086 Processor Kit



Pinless Microprocessor




intel
DELL

History of Microprocessor

MP	Introduction	Data Bus (In Bits)	Address Bus (In Bits)
4004	1971	4	8
8008	1972	8	8
8080	1974	8	16
8085	1977	8	16
8086	1978	16	20
80186	1982	16	20
80286	1983	16	24
80386	1986	32	32
80486	1989	32	32
Pentium	1993 onwards	32	
Core solo	2006	32	
Dual Core	2006	32	
Core 2 Duo	2006	32	
Core to Quad	2008	32	
I3,i5,i7	2010	64	40

- Importance features possessed by Microprocessors:

1. **8086**
   I. **Segmentation:** Reduces Access Time
   II. **Pipelining:** Reduces Execution Time

2. **80386**
   I. **Data Security**

Microprocessor Functions

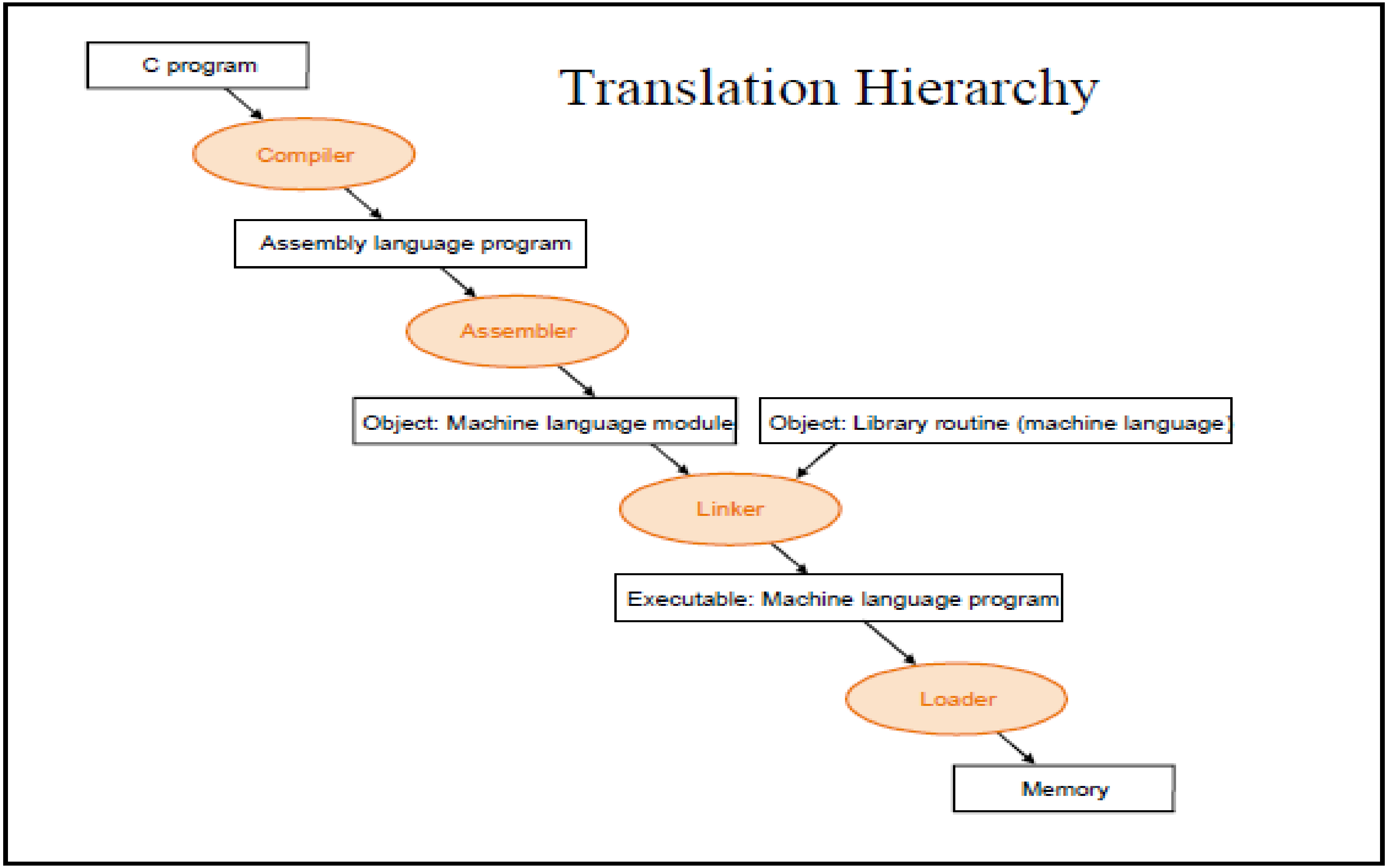
- Microprocessor functions mainly involve
  - **Instruction Fetch and Execute**
  - **Interrupts**
  - **I/O Function**

Assembly Language Programming

Agenda

- Assemblers, Linkers & Loaders
- Introduction to Nasm
- ALP constructs
  - Data Types
  - Byte Addressing in Memory
  - Memory Addressing
  - System Calls: 32 bit & 64 bit
  - Procedure and Macro
  - Stack
  - Directives
- ASCII Table and Conversions
- Programming case Studies

Assemblers, Linkers & Loaders



- Assembly language program
  - Assembly language program (.asm) file—known as source code
  - Converted to machine code by a process called assembling
  - Assembling performed by a software program—an 80x86 assembler
  - Machine (object) code that can be run is output in the executable (.exe) file
  - Source listing output in (.lst) file—printed and used during execution and debugging of program
- DEBUG—part of disk operating system (DOS) of the PC
  - Permits programs to be assembled and disassembled
  - Line-by-line assembler
  - Also permits program to be run and tested

- General structure of an assembly language statement

```
LABEL:      INSTRUCTION         ;COMMENT
```

  - Label—address identifier for the statement
  - Instruction—the operation to be performed
  - Comment—documents the purpose of the statement
  - Example:

```
START:   MOV   AX, BX    ; Copy BX into AX
```

- Other examples:

```
INC SI      ;Update pointer
ADD   AX, BX
```

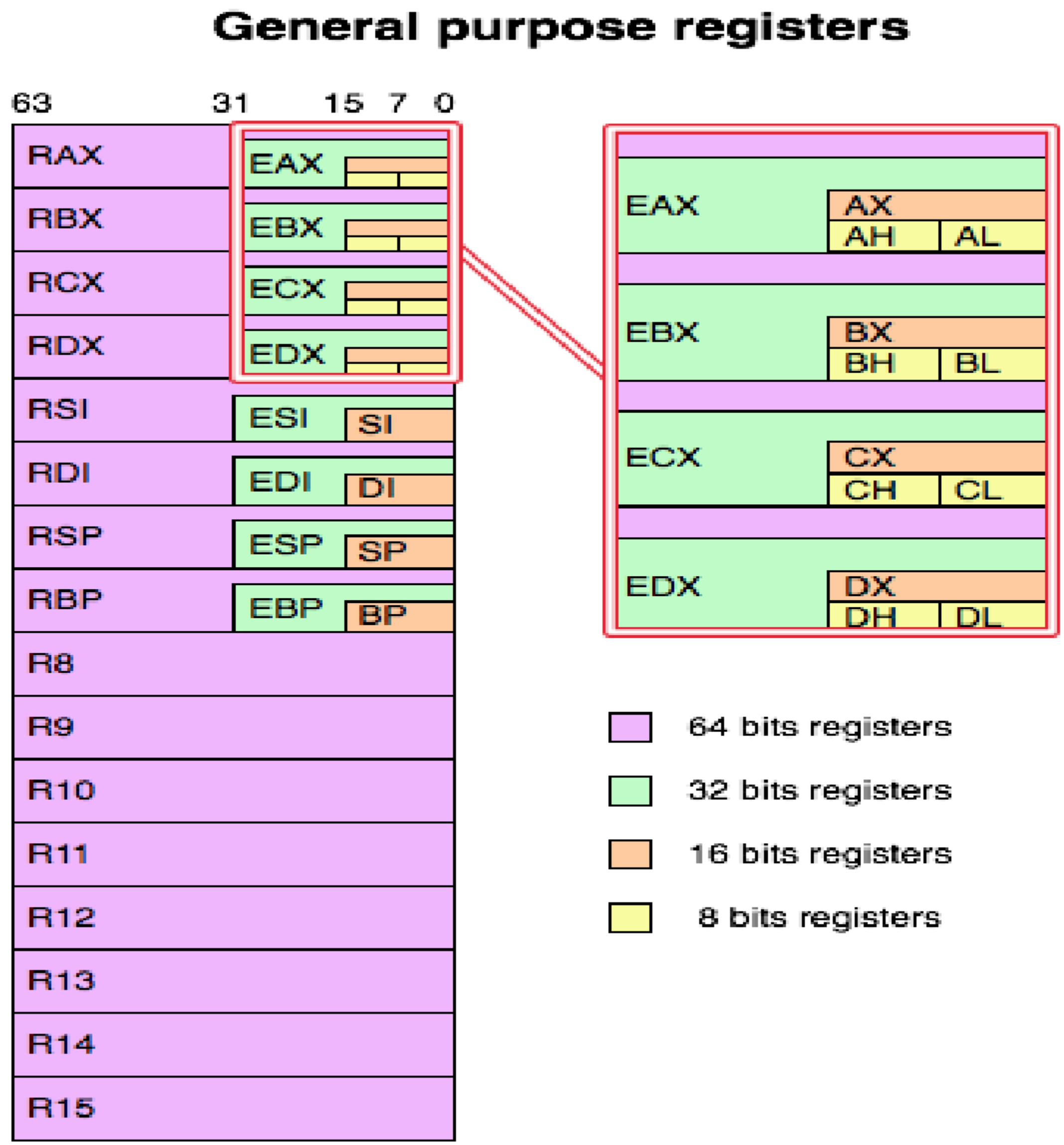
  - Few instructions have a label—usually marks a jump to point
  - Not all instructions need a comment

- Each instruction is represented by a mnemonic that describes its operation—called its operation code (opcode)
  - MOV = move → data transfer
  - ADD = add → arithmetic
  - AND = logical AND → logic
  - JMP = unconditional jump → control transfer
- Operands are the other parts of an assembly language Instructions
  - Identify whether the elements of data to be processed are in registers or memory
    - Source operand– location of one operand to be process
    - Destination operand—location of the other operand to be processed and the location of the result

1. **Assembler**
   - TASM (Windows) ::: 16 bit
   - MASM (Windows) ::: 64 bit
   - NASM (Linux) ::: 64 bit

2. **Different buses used of processor**
   - Data bus : Defines Bandwidth.
   - Address bus : Defines size of Physical Memory.

3. Register File



4. NASM program template

```
section .data

                    ; Predefine Data [Variable definition]

section .bss

                    ; Undefined Data [Variable declaration]

section .text
global main
main:

                    ; Source code


Mov  rax,60         ; Exit system call
Mov rdi, 00
syscall
```

5. Execution steps for NASM program

64-bit program execution on 64 bit machine

Commands:

- **Assemble:** *nasm -f elf64 filename.asm*
- **Linking:** *ld -o outputfilename filename.o*
- **Execute:** *./outputfilename*

Example:

- **Assemble:** *nasm -f elf64 addition.asm*
- **Linking:** *ld -o addition addition.o*
- **Execute:** *./addition*

5. Execution steps for NASM program

32-bit program execution on 32 bit machine

<u>Commands:</u>

- **Assemble:** *nasm -f elf filename.asm*
- **Linking:** *ld -o outputfilename filename.o*
- **Execute:** *./outputfilename*

<u>Example:</u>

- **Assemble:** *nasm -f elf addition.asm*
- **Linking:** *ld -o addition addition.o*
- **Execute:** *./addition*

5. Execution steps for NASM program

32-bit program execution on 64 bit machine

Commands:

- **Assemble:** *nasm -f elf filename.asm*
- **Linking:** *ld –m elf_i386 –o outputfilename filename.o*
- **Execute:** *./outputfilename*

Example:

- **Assemble:** *nasm -f elf64 addition.asm*
- **Linking:** *ld -m elf_i386 -o addition addition.o*
- **Execute:** *./addition*

ALP Constructs

1. Basic Data Types

Data Types:

- Byte (8-bit)
- Word (16-bit)
- Double word (32-bit)
- Quadword (64-bit)
- Ten bytes (80-bit)

2. Data Types

1. Definition directives
   - **db** (define byte)
   - **dw** (define word)
   - **dd** (define double word)
   - **dq** (define quad word)
   - **dt** (define ten bytes)

2. Declaration directives
   - **resb** (reserve byte)
   - **resw** (reserve word)
   - **resd** (reserve double word)
   - **resq** (reserve quad word)

3. Memory addressing directives

- byte
- word
- dword
- qword

3. Byte Ordering in Computer Memory (Data definition)

1. Little endian machine
   - Stores data **little-end first**
   - **Least significant byte** at smallest address
   - Example: Intel processors (all x86 processors)

2. Big endian machine
   - Stores data **big-end first**
   - **Most significant byte** at smallest address
   - Example: IBM processors (Power PC)

4. Byte Ordering in Computer Memory – Data Definition (Continued…)

Little endian

Memory location	**Data**
1000000A h	
10000009 h	12
10000008 h	34
10000007 h	56
10000006 h	78
10000005 h	A9
10000004 h	5C
10000003 h	CD
10000002 h	FE
10000001 h	
10000000 h	

Big endian

Memory location	**Data**
1000000A h	
10000009 h	FE
10000008 h	CD
10000007 h	5C
10000006 h	A9
10000005 h	78
10000004 h	56
10000003 h	34
10000002 h	12
10000001 h	
10000000 h	

Qnumber dq 12345678A95CDFE h

5. Memory addressing

section .data

num dq 9828919849096878h

section .bss

name resb 8 ; assembly

<u>Memory addressing:</u>

```
mov al, byte[num]          ; al = 78
mov ax , word [num]        ; ax = 6878
mov eax , dword [num]      ; eax = 49096878
mov rax , qword [num]
                           ; rax = 9828919849096878
```

Memory location	Data
1000000A h	
10000009 h	98
10000008 h	28
10000007 h	91
10000006 h	98
10000005 h	49
10000004 h	09
10000003 h	68
10000002 h	78
10000001 h	
10000000 h	

6. System calls 32-bit

<u>Syntax:</u>

mov eax, <u>*syscall number*</u> *;03-read, 04-write, 01-exit*

mov ebx, <u>*file descriptor*</u> *;01 – standard input/output e.g. console*

mov ecx, <u>*buffer*</u> *;buffer to be read / written*

mov edx, <u>*length in bytes*</u> *;number of bytes to read / write*

int 0x80

6. System calls 64-bit

Syntax:

mov rax, *syscall number* *;00-read, 01-write, 60-exit*

mov rdi, *file descriptor* *;01 – standard input/output e.g. console*

mov rsi, *buffer* *;buffer to be read / written*

mov rdx, *length in bytes* *;number of bytes to read / write*

syscall

File Descriptor	32 bit	64 bit	File descriptor	Description
Write	4	1	1	To print on screen
Read	3	0	0 / 1	To accept input
Exit	1	60	0	To end the program

6. System calls (Continued..)

<u>Example:</u> ***Write system call***

64 bit	*32bit*
mov rax, 01	mov eax, 04
mov rdi, 01	mov ebx, 01
mov rsi, name	mov ecx, name
mov rdx, 8	mov edx, 8
syscall	int 0x80

6. System calls (Continued..)

Example: ***Read system call***

64 bit	*32bit*
mov rax, 00	mov eax, 03
mov rdi, 01	mov ebx, 01
mov rsi, name	mov ecx, name
mov rdx, 8	mov edx, 8
syscall	int 0x80

6. System calls (Continued..)

Example: ***Exit system call***

64 bit	*32bit*
mov rax, 60	mov eax, 01
mov rdi, 00	mov ebx, 00
syscall	int 0x80

```
Section .data

Msg: db "Hello World",0x0A
Len: equ $-Msg

Section .text
Global Main
Main:

Mov rax, 1
Mov rdi, 1
Mov rsi, Msg
Mov rdx, Len
syscall

Mov rax, 60
Mov rdi,0
syscall
```

msg

100 : H

1 : e

2 : l

3 : l

4 : o

5 : _

6 : W

7 : o

8 : r

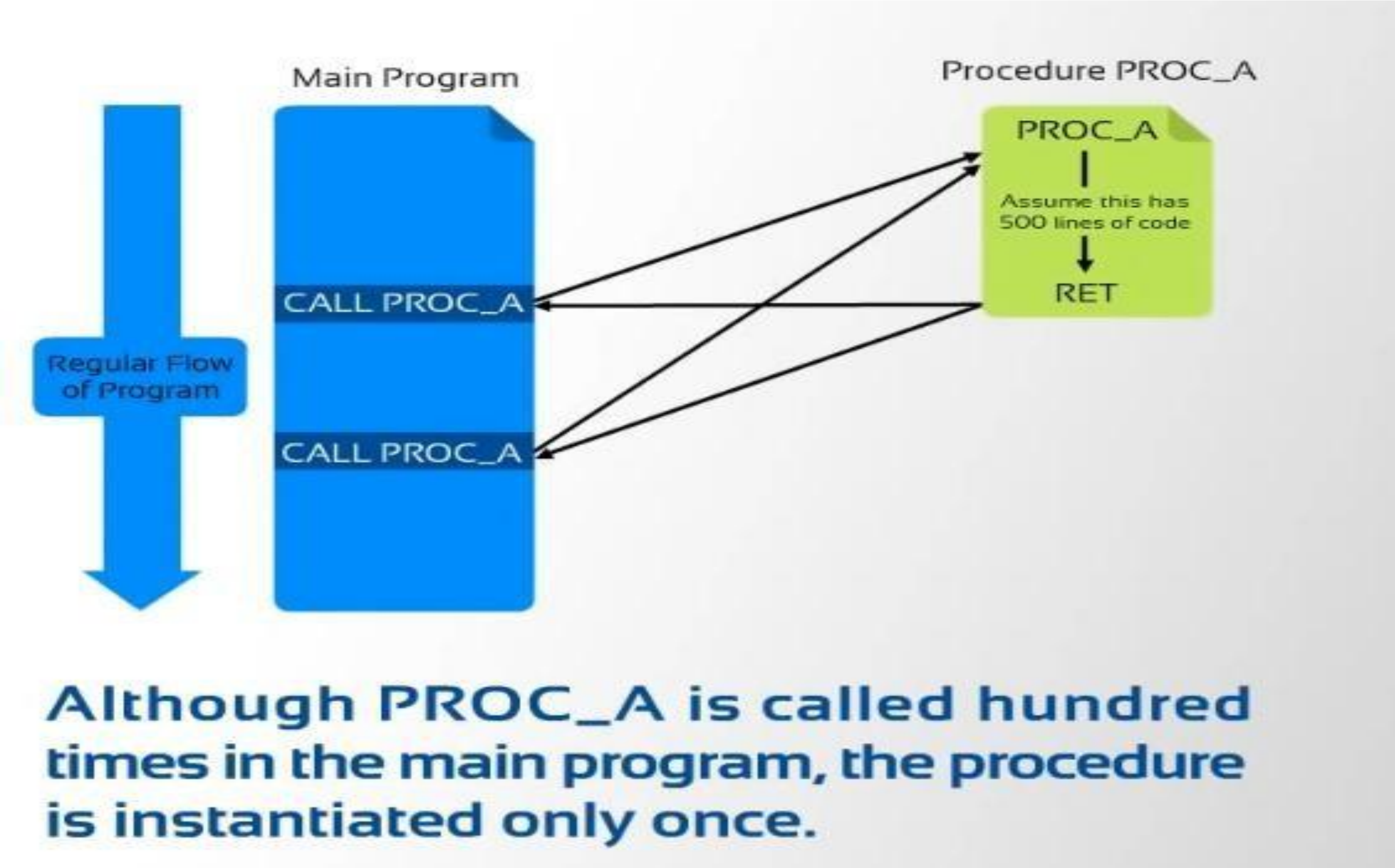
9 : l

A : d

10B : 10

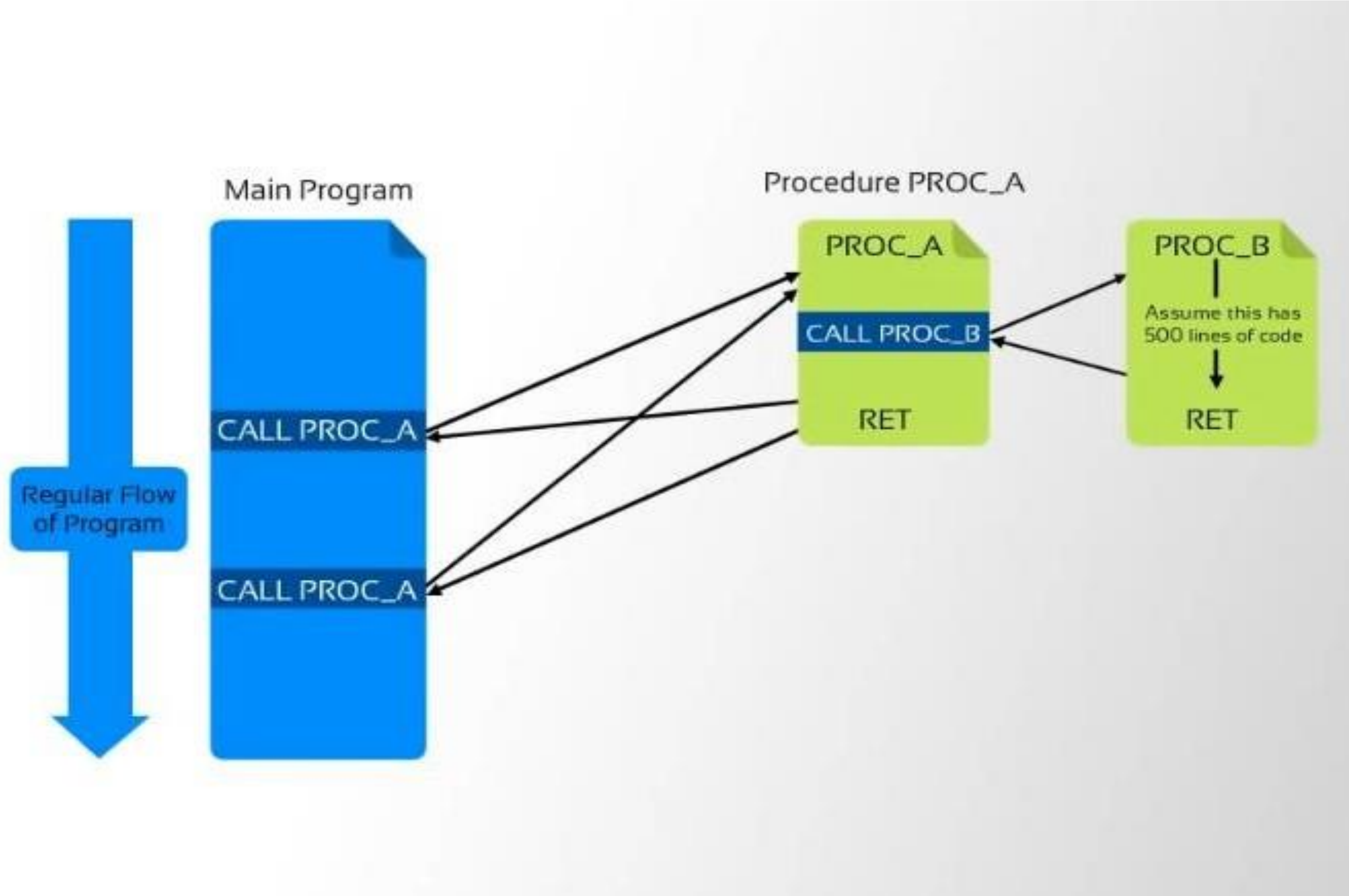
200 : C

7. Procedures



Why Procedures?

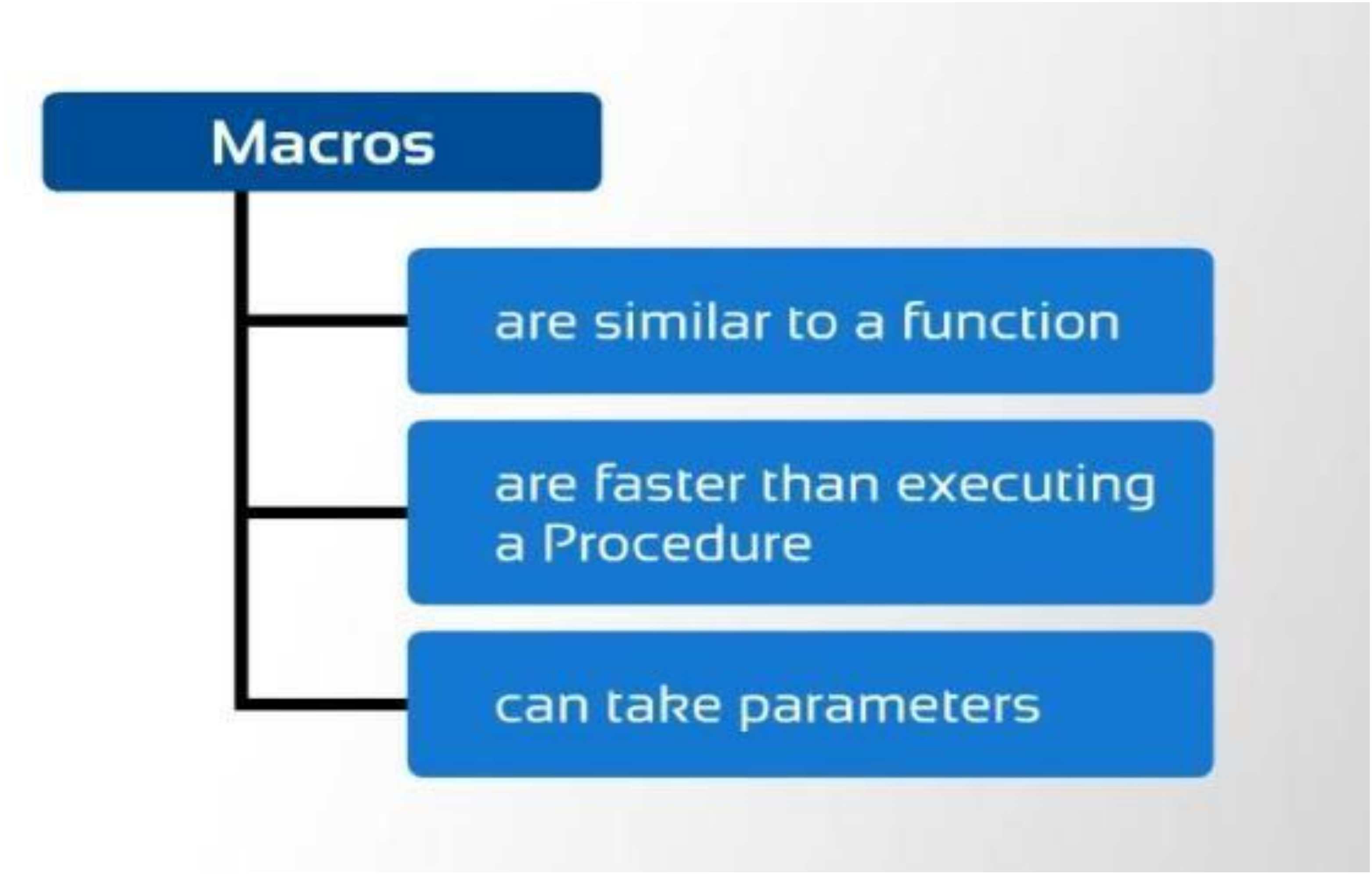
Nested Procedures



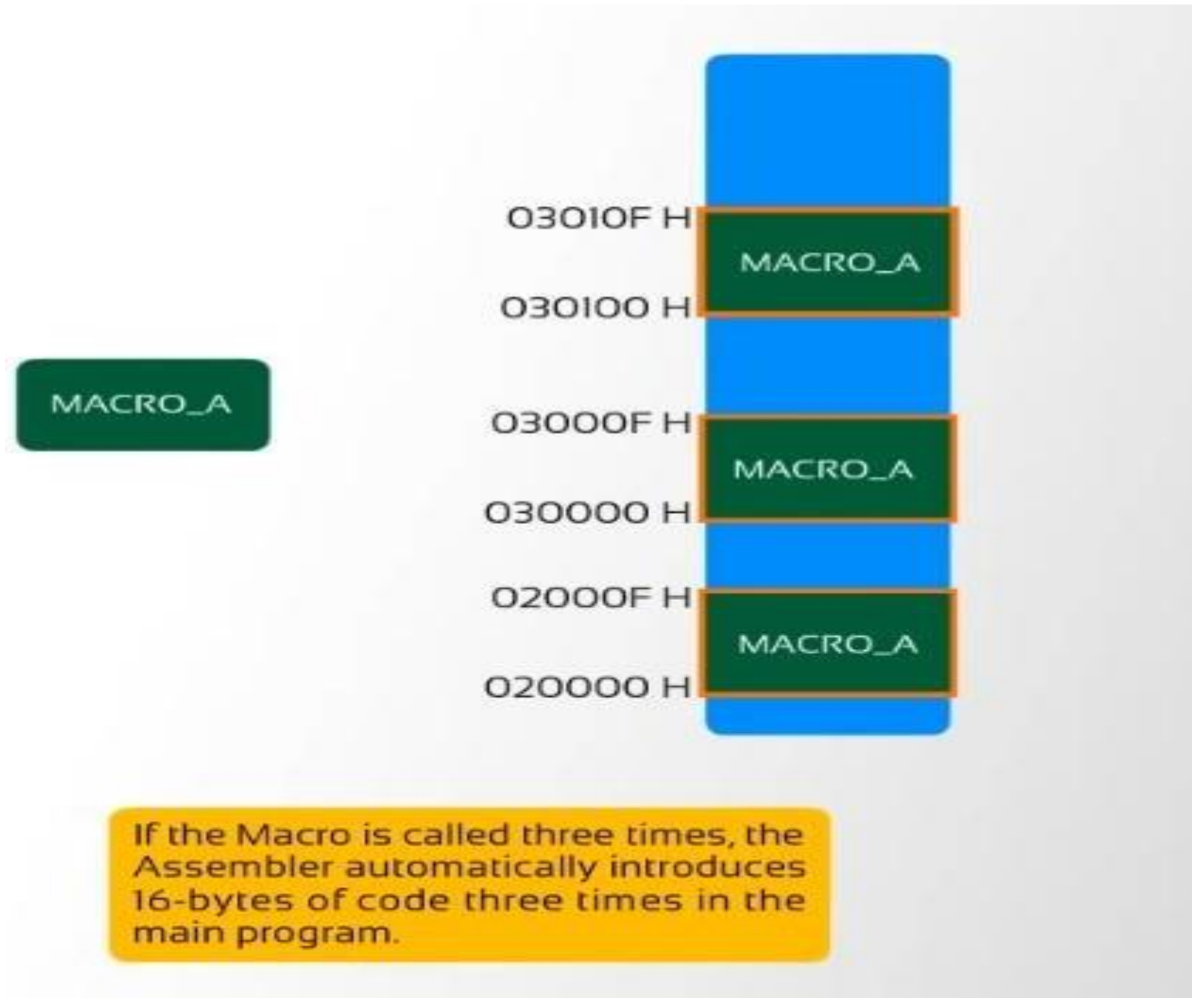
- Two kinds of Procedures:

1. **NEAR** Procedure
2. **FAR** Procedure

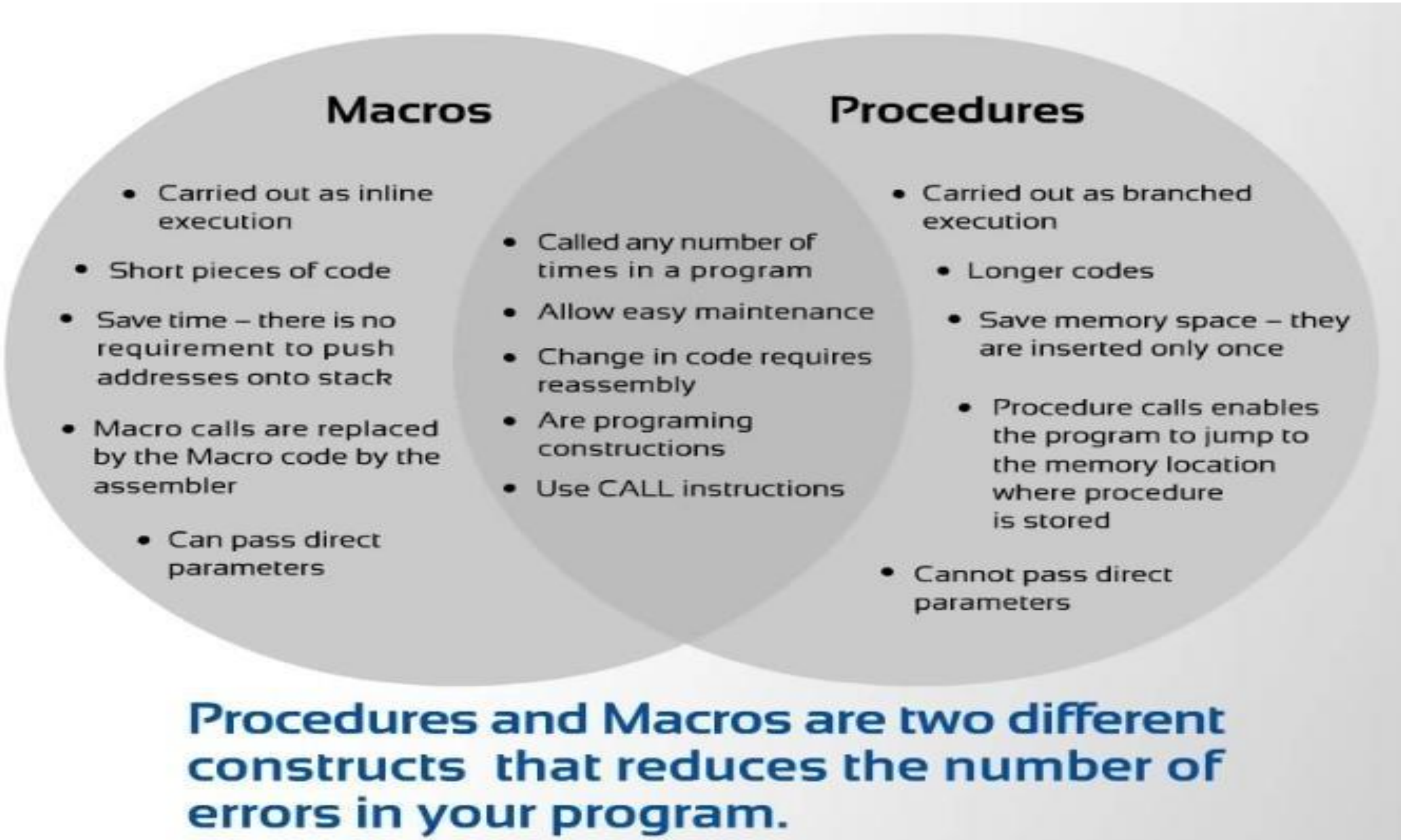
8. Macros


Macros
are similar to a function
are faster than executing a Procedure
can take parameters

Macros as Inline codes



Difference between Macro and Procedure



How to define macro

section .data

msg: db “hello”,10
len: equ $-msg

Section .bss

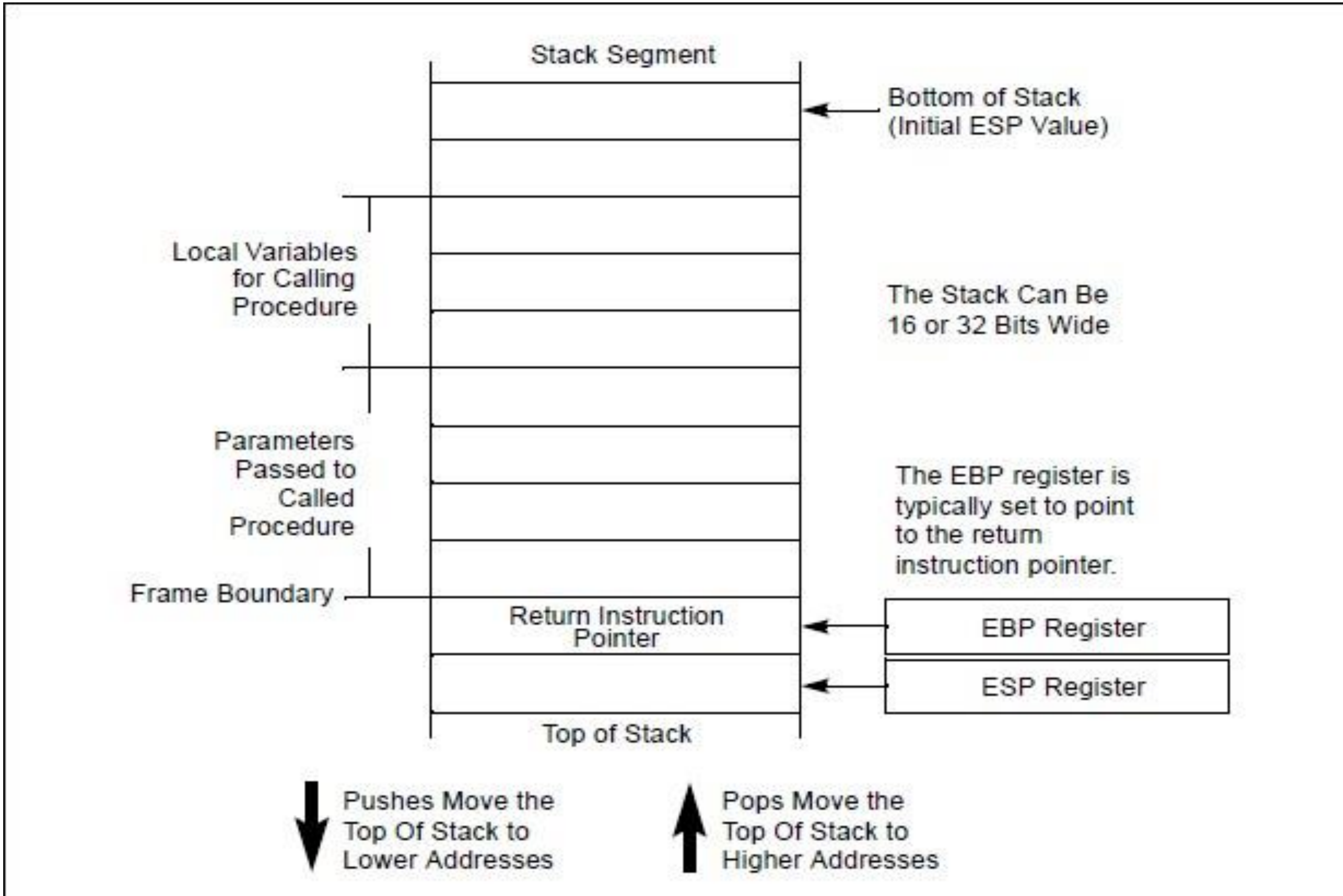
count: resb 2

```
%macro print 2
Mov rax,1
Mov rdi,1
Mov rsi, %1
Mov rdx, %2
Syscall
%endmacro
```

Section .text

```
Global  main
Main:
-
print msg,len
-
-
print msg,len
-
-
-
; code of addition and result stored in COUNT variable
print count,2
-
-
Mov rax,60
Mov rdi,0
syscall
```

9. Stack



Stack Segment
Bottom of Stack
(Initial ESP Value)
Local Variables
for Calling
Procedure
The Stack Can Be
16 or 32 Bits Wide
Parameters
Passed to
Called
Procedure
The EBP register is
typically set to point
to the return
instruction pointer.
Frame Boundary
Return Instruction
Pointer
EBP Register
ESP Register
Top of Stack
Pushes Move the
Top Of Stack to
Lower Addresses
Pops Move the
Top Of Stack to
Higher Addresses

10. Directives

- There are some instructions in the assembly language program **which are not a part of Processor Instruction Set**.

- These instructions are instructions to the **Assembler, Linker and Loader**. These are referred to as pseudo-operations or as assembler directives.

- **DB –** Define Byte
- **DD –** Define Doubleword
- **DQ –** Define Quadword
- **DT –** Define Ten Bytes
- **DW –** Define Word
- **ENDS**
- This directive is used with name of the segment to indicate the end of that logic segment.

CODE SEGMENT ; this statement starts the segment

CODE ENDS ; this statement ends the segment

- EQU

11. ‘Hex to ASCII’ & ‘ASCII to Hex’ conversion

ASCII TABLE

2
4
6

2 4
32 34

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

ASCII to Hex conversion

Consider the following sequence of instructions:

1. Accept 4 digit number from user using read system call.

2. Assume accepted number is stored in variable "num".

3. After conversion to hex, number will be stored to register BX.

ASCII to Hex conversion

```
mov bx,0
mov esi,num
mov byte[cnt],4

up:
rol bx,4
mov cl,byte[esi]
cmp cl,39h
jbe next
sub  cl,7
next:
sub cl,30h
add bl,cl
inc esi
dec byte[cnt]
jnz up
```

30	0
31	1
32	2
33	3
34	4
35	5
36	6
37	7
38	8
39	9

41	A
42	B
43	C
44	D
45	E
46	F

Hex to ASCII conversion

12AB
34BD
56CD

Consider the following sequence of instructions:

1. Assume 4 digit number which we want to print is stored in CX register.

   56CD

2. Converted ASCII value will be stored in variable "result".

3. Print value in result using write system call.

Hex to ASCII conversion

```
mov edi,result
mov byte[cnt],4

up2:
rol cx,4
mov bl,cl
and bl,0Fh
cmp bl,9
jbe next2
add bl,7
next2:
add bl,30h
mov byte[edi],bl
inc edi
dec byte[cnt]
jnz up2
```

4
3
2
1
0

CD
0F
0D
37
44

bl: D5, 0F, 05, 30, 35, 56, 0F, 06, 30, 36, 6C, 0F, 0C, 37, 43

cl: ~~56CD~~, ~~6CD5~~, CD56, D56C, 56CD

edi: result → 500: 35, 1: 36, 2: 43, 3: 44, 4

56CD

ALP Case Studies

Case Study 1: Hello world…!!!

<u>Hello World: 64 Bit</u>

```
Section .data
msg: db "HELLO!",0x0A
len: equ $-msg

Section .text
global main
main:

mov rax, 1
mov rdi, 1
mov rsi, msg
mov rdx, len
Syscall

mov rax, 60
mov rdi, 0
syscall
```

<u>Hello World: 32 Bit</u>

```
Section .data
msg: db "HELLO WORLD",10
len: equ $-msg

Section .text
global main
main:

mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, len
int 0x80

mov eax, 1
mov ebx, 0
int 0x80
```

Case Study 2

- How to check whether number is positive or negative?
- Twowaysto represent negativenumbers:
  - SignMagnitude
  - Two'sComplement

Algorithm

1. Start
2. Define array of 16/32/64 bit hexadecimal numbers
3. Initialize positive counter and negative counter to zero
4. Set the pointer
5. Set counter (number of elements in array)
6. Take number pointed by pointer check its MSB (use BT instruction)
7. If MSB is 1 then increment negative counter else increment positive counter.
8. Increment pointer and decrement counter
9. Goto 6 until counter become zero
10. Convert positive and negative counter (HEX to ASCII) & print
11. Stop

BT Instruction

Instruction	Op/En	Description
BT *r/m16, r16*	MR	Store selected bit in CF flag.
BT *r/m32, r32*	MR	Store selected bit in CF flag.
BT *r/m64, r64*	MR	Store selected bit in CF flag.
BT *r/m16, imm8*	MI	Store selected bit in CF flag.
BT *r/m32, imm8*	MI	Store selected bit in CF flag.
BT *r/m64, imm8*	MI	Store selected bit in CF flag.

Description:
Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand) and stores the value of the bit in the CF flag.

Flags Affected: Only CF is affected. The OF, SF, ZF, AF, and PF flags are undefined.

Case Study 3

- **Perform Arithmetic and Logical operations**

- Following operations can be done
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - AND
  - OR
  - XOR

[Status of the answer can be verified by checking contents of flag register.]

Algorithm

- Accept first number [May be Multiplicand or Dividend]
- Convert ASCII to Hex
- Accept second number [May be Multiplier or Divisor]
- Convert ASCII to Hex
- Perform operation
- Convert result to ASCII
- Print result

Addition

```
section .data

msg: db "answer is",10
len: equ $-msg

msg1: db "Enter first
number",10
len1: equ $-msg1

msg2: db "Enter second
number",10
len2: equ $-msg2

section .bss

num: resb 5
num1: resb 5
num2: resb 5
result: resb 8
cnt: resb 2
```

```
%macro scall 4
   mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
    int 0x80
%endmacro

section .text
global main
main:

scall 4,1,msg1,len1
scall 3,1,num,5
call a_to_h
[Generates 4 digit
answer in ASCII]
mov word[num1],bx
```

```
scall 4,1,msg2,len2
scall 3,1,num,5
call a_to_h
mov word[num2],bx

mov cx, word[num1]
mov dx, word[num2]

add cx,dx

call h_to_a

scall 4,1,msg,len
scall 4,1,result,4

mov eax,1
mov ebx,0
int 0x80
```

Subtraction

```
section .data

msg: db "answer is",10
len: equ $-msg

msg1: db "Enter first
number",10
len1: equ $-msg1

msg2: db "Enter second
number",10
len2: equ $-msg2

section .bss

num: resb 5
num1: resb 5
num2: resb 5
result: resb 8
cnt: resb 2
```

```
%macro scall 4
   mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
   int 0x80
%endmacro

section .text
global main
main:

scall 4,1,msg1,len1
scall 3,1,num,5
call a_to_h
[Generates 4 digit
answer in ASCII]
mov word[num1],bx
```

```
scall 4,1,msg2,len2
scall 3,1,num,5
call a_to_h
mov word[num2],bx

mov cx, word[num1]
mov dx, word[num2]

sub cx, dx

call h_to_a

scall 4,1,msg,len
scall 4,1,result,4

mov eax,1
mov ebx,0
int 0x80
```

Multiplication

section .data

```
msg: db "answer is",10
len: equ $-msg

msg1: db "Enter first
number",10
len1: equ $-msg1

msg2: db "Enter second
number",10
len2: equ $-msg2
```

section .bss

```
num: resb 5
num1: resb 5
num2: resb 5
result: resb 8
cnt: resb 2
```

```
%macro scall 4
    mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
    int 0x80
%endmacro
```

section .text

```
global main
main:

scall 4,1,msg1,len1
scall 3,1,num,5
call a_to_h
mov word[num1],bx

scall 4,1,msg2,len2
scall 3,1,num,5
```

```
call a_to_h
mov word[num2],bx

Mov eax, 0
mov ax, word[num1]
mov bx, word[num2]
```

mul bx

```
call h_to_a
[Generates 8 digit
answer in ASCII]

scall 4,1,msg,len
scall 4,1,result,8

mov eax,1
mov ebx,0
int 0x80
```

Division

section .data

```
msg: db "answer is",10
len: equ $-msg

msg1: db "Enter first number",10
len1: equ $-msg1

msg2: db "Enter second number",10
len2: equ $-msg2
```

section .bss

```
num: resb 5
num1: resb 5
num2: resb 3
result: resb 8
cnt: resb 2
```

```
%macro scall 4
    mov eax, %1
    mov ebx, %2
    mov ecx, %3
    mov edx, %4
    int 0x80
%endmacro
```

section .text

```
global main
main:

scall 4,1,msg1,len1
scall 3,1,num,5
call a_to_h
mov word[num1],bx

scall 4,1,msg2,len2
scall 3,1,num,3
```

```
call a_to_h2
[Generates 2 digit answer in Hex]
mov byte[num2],bl

Mov eax, 0
Mov bl, 0
mov ax, word[num1]
mov bl, byte[num2]
```

Div bl

```
call h_to_a

scall 4,1,msg,len
scall 4,1,result,4

mov eax,1
mov ebx,0
int 0x80
```

Case Study 4

- Block transfer:
  - Without String Instruction
  - With String Instruction

Non-Overlapped Block transfer

- Before

Address(A)	Value
101	10
102	20
103	30
104	40
105	50

- After

Address(B)	Value
201	10
202	20
203	30
204	40
205	50

Overlapped Block transfer

- Before

Address(A)	Value
101	10
102	20
103	30
104	40
105	50

- After

Address(B)	Value
101	10
102	20
103	30
104	10
105	20
106	30
107	40
108	50

Address(B)	Value
98	10
99	20
100	30
101	40
102	50
103	30
104	40
105	50

Without String instruction

1. Start
2. Print addresses and values of first array
3. Set pointer (Use pointer registers) at both the arrays
4. Set counter
5. Copy the data from first array to second array
6. Increment both pointers
7. Decrement counter
8. Goto step 5 until counter become zero
9. Print addresses and values of second array
10. Stop

Algorithm- add : value

1. Start
2. Setpointer (e.g rsi)andcounter
3. Takeaddress(i.e rsi) & call procedure(hex toascii)
4. Print the address
5. Print colon
6. TakeValue at Pointer(i.e.[rsi]) and call procedure(hex to ascii)
7. Print the value
8. Increment pointer
9. Decrement counter
10. Goto step 3 until counter is notzero
11. Stop

```
mov rsi, array
mov byte[count], 05

up:
 mov rbx, rsi
push rsi
 mov rdi, addr                    ;Converts Address (16 digit) to ASCII and print.
call HtoA1
pop rsi

mov dl, byte[rsi]
push rsi
 mov rdi, num1                    ;Converts Data (2 digit) to ASCII and print.
call HtoA2
pop rsi

inc rsi

dec byte[count]
jnz up
```

Data Transfer

```
mov rsi, array
mov rdi, array+5h
mov byte[count3], 05h

loop10:
mov dl, 00h
mov dl, byte[rsi]
mov byte[rdi], dl
inc rsi
inc rdi
dec byte[count3]
jnz loop10
```

String Data transfer

- Forward
- Backward

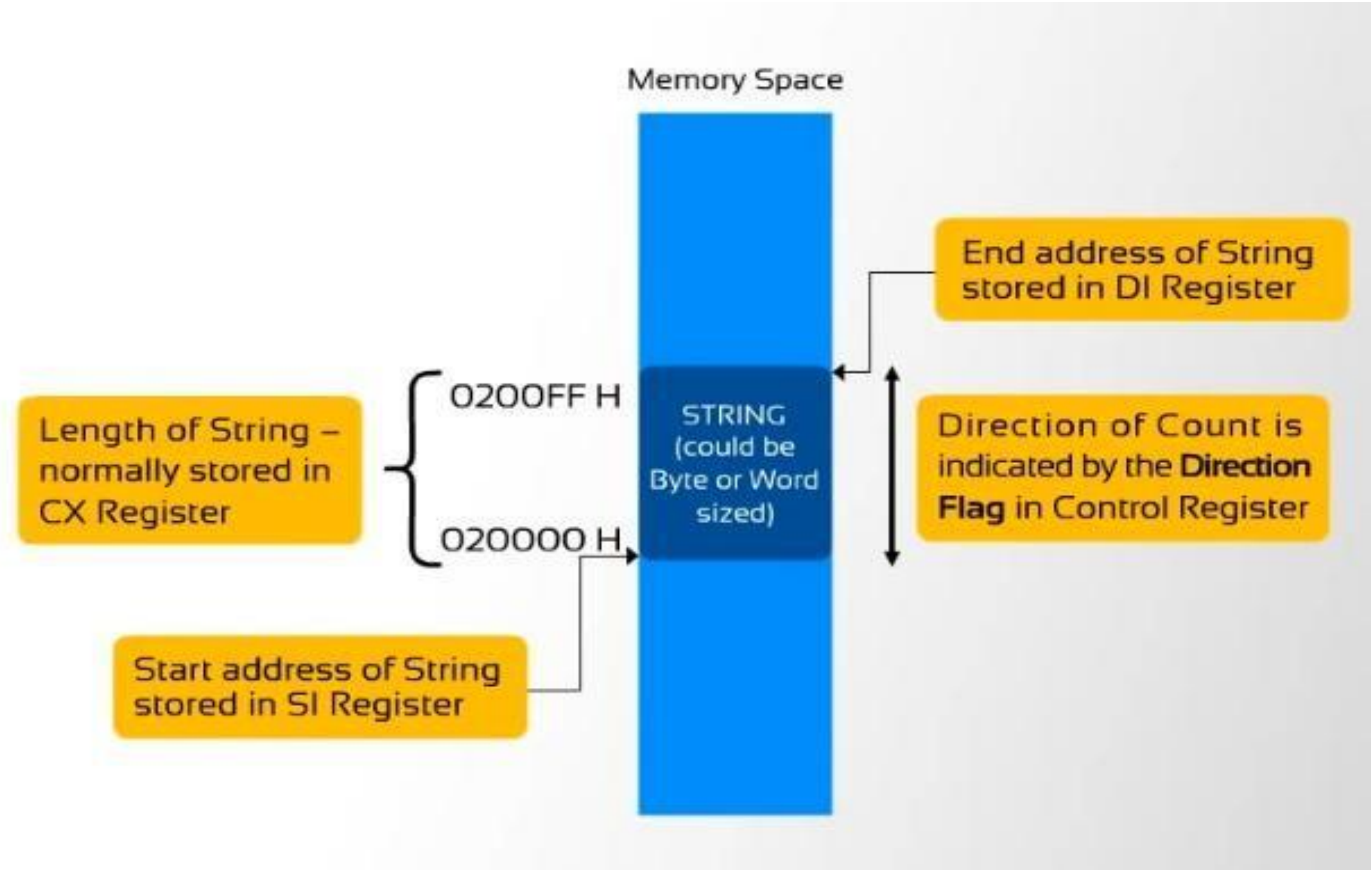
Forward

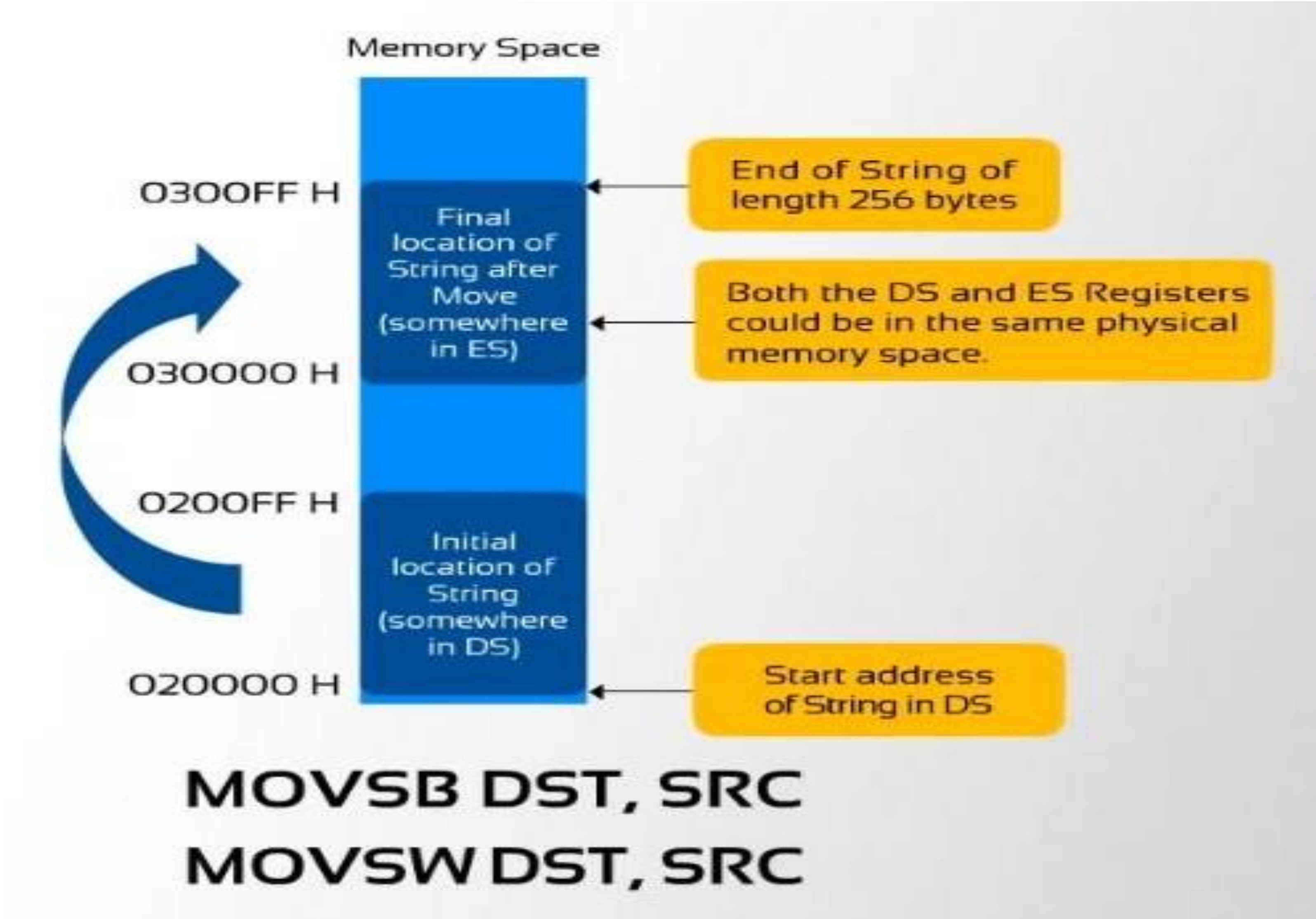
1. Start
2. Set offset of source string at rsi
3. Set offset of destination string at rdi
4. Set counter at rcx
5. Clear Direction flag(DF=0)
6. Transfer data(MOVS)
7. Stop

Backward

1. Start
2. Set offset of source string at rsi
3. Set offset of destination string at rdi
4. Set counter at rcx
5. Set Direction flag (DF=1)
6. Transfer data (MOVSQ)
7. Stop

String Instructions




Memory Space
0300FF H
030000 H
0200FF H
020000 H
Final location of String after Move (somewhere in ES)
Initial location of String (somewhere in DS)
End of String of length 256 bytes
Both the DS and ES Registers could be in the same physical memory space.
Start address of String in DS
MOVSB DST, SRC
MOVSW DST, SRC

If sting instructions are not used..

```
        MOV SI, OFFSET STRINGI;      USE SI  AS SOURCE INDEX
        MOV DI, OFFSET STRING2;      USE DI  AS DESTINATION INDEX
        MOV CX, LENGTH STRINGI;      PUT LENGTH OF STRING IN CX
MOVE:   MOV AL, (SI);                MOVE BYTE FROM SOURCE
        MOV (DI), AL;                TO DESTINATION
        INC SI;                      INCREMENT SOURCE INDEX
        INC DI;                      INCREMENT DESTINATION INDEX
        LOOP MOVE
```

LOD SB Loads a byte from a String in memory into AL. Automatically increments/decrements SI by 1. **LOD SW** Loads a word from a String in memory into AX. Automatically increments/decrements SI by 2.	Moving from String to Accumulator
STO SB Stores a byte from AL into a String location in memory. Automatically increments/decrements DI by 1. **STO SW** Stores a word from AX into a String location in memory. Automatically increments/decrements DI by 2.	Moving from Accumulator to String

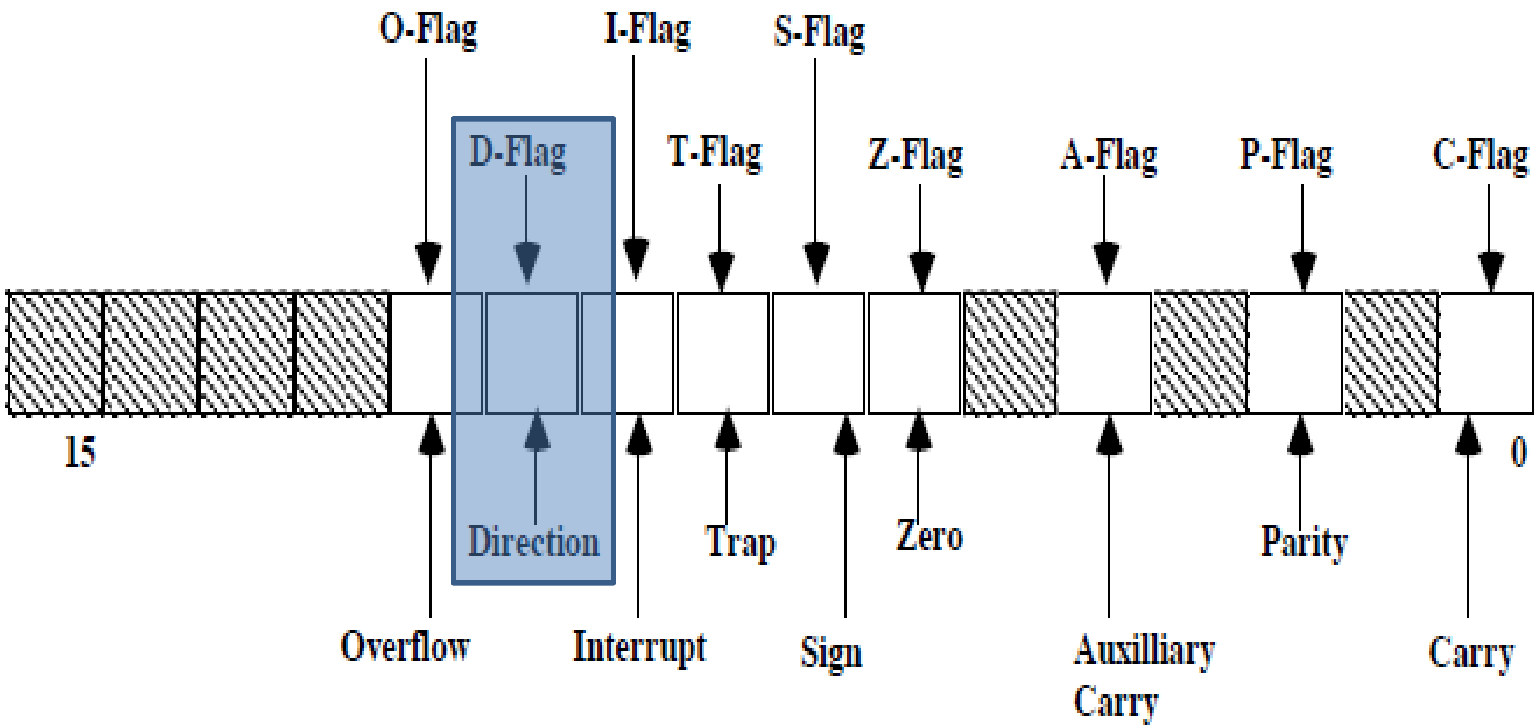
CMPSB or CMPSW – compares either Byte or Word Strings

- The CX Register holds length of STRINGs to be compared.
- STRING1 is pointed to by [DS:SI], STRING2 by [ES:DI].
- If STRING1 = STRING2; then Zero Flag is Set.

SCASB or SCASW – scans either Byte or Word Strings

- The CX Register holds length of STRING to be scanned.
- STRING is pointed to by [DS:SI].
- If the STRING contains value, Zero Flag is Set.

- **Flag Register 0f 8086**



Instruction of Direction Flag::

CLD: Clear Direction Flag
STD: Set Direction Flag