UNIT 4
Instruction Sets

What is Microprocessor

- 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.

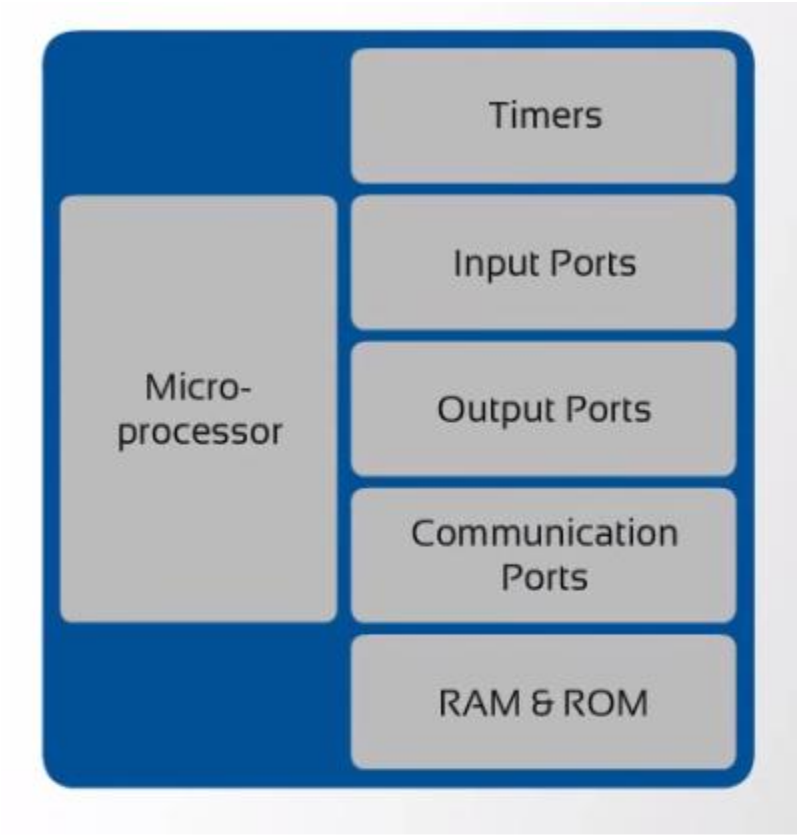
Three basic characteristics differentiate Microprocessors:

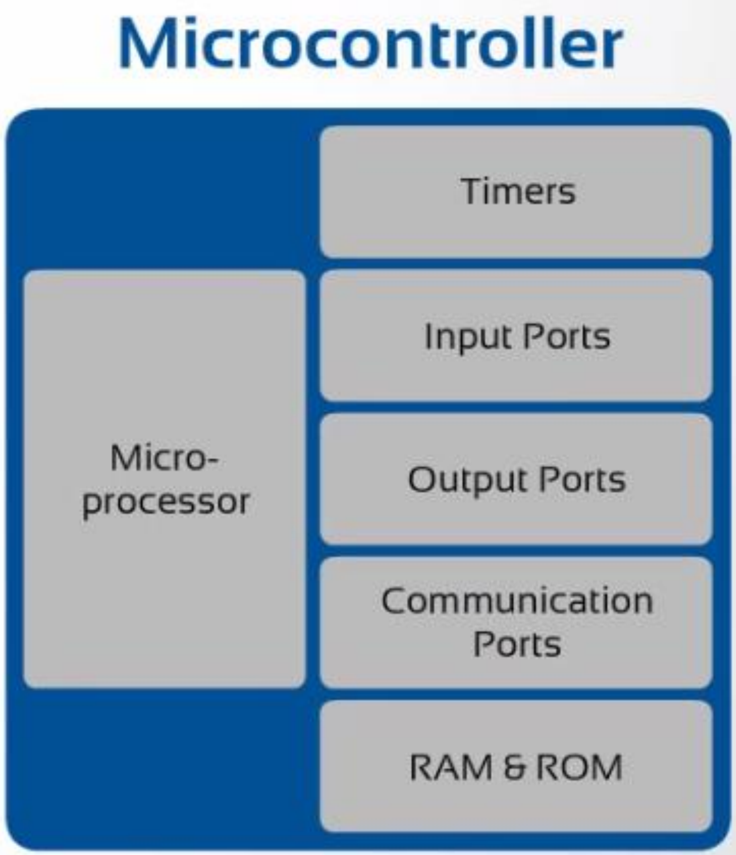
- **Instruction set**: The set of instructions that the microprocessor can execute.
- **Bandwidth :** The number of bits processed in a single instruction.
- **Clock speed :** Given in megahertz (MHz), the clock speed determines how many instructions per second the processor can execute.

In both cases, the higher the value, the more powerful the CPU.

For example, a 32-bit microprocessor that runs at 50MHz is more powerful than a 16-bit microprocessor that runs at 25MHz.

Microprocessor Vs. Microcontroller

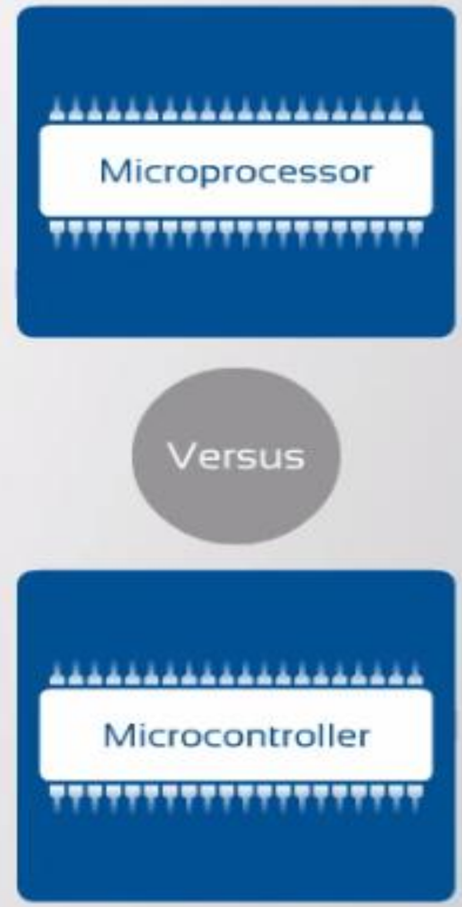




Microcontroller, as an Integrated Circuit (IC), is complex than a General Purpose Processor.

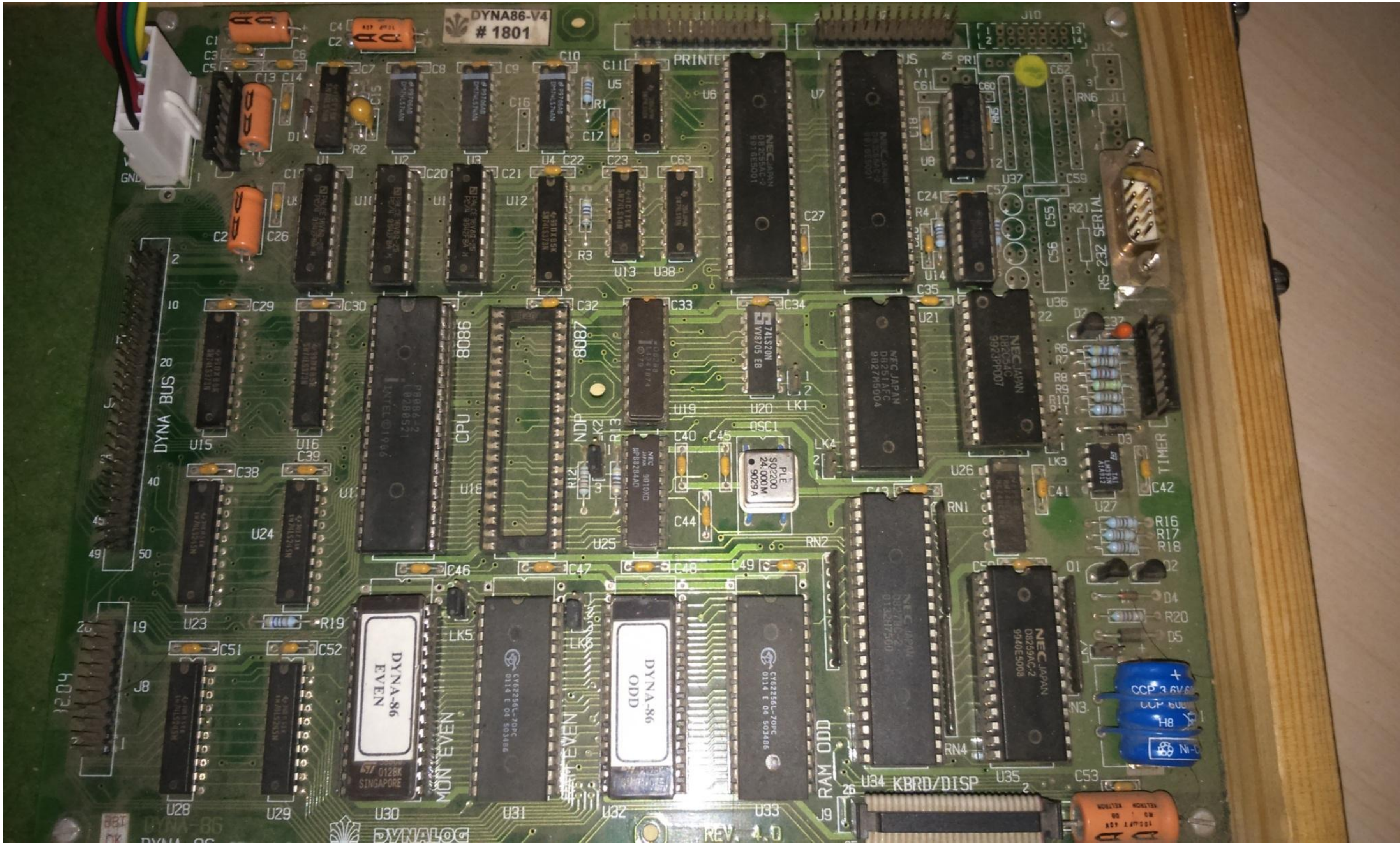
Differences between a Microprocessor and a Microcontroller:

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

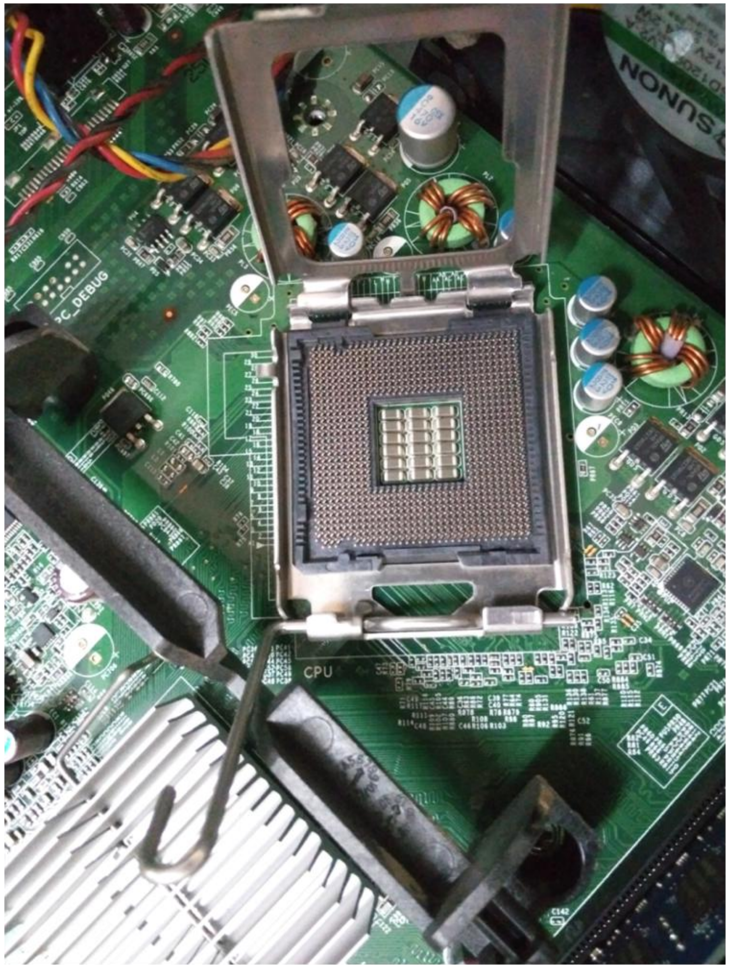


- Specific usages
- Contains the CPU and many peripheral devices
- System costs are lower
- Cannot operate at higher Clock speed
- Requires programming only once for a particular application

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

Microprocessor Functions

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

About 8086

- **It is 16 bit processor.** So that it has 16 bit ALU, 16 bit registers and internal data bus and 16 bit external data bus.

- 8086 has 20 bit address lines to access memory. Hence it can access.

2^20 = 1 MB memory location

- **Pipelining:-**8086 uses two stage of pipelining. First is Fetch Stage and the second is Execute Stage.
  - **Fetch stage** that prefetch upto 6 bytes of instructions stores them in the queue.
  - **Execute stage** that executes these instructions.

- Pipelining improves the performance of the processor so that operation is faster.

- **Segmentation** divides memory into 4 parts.

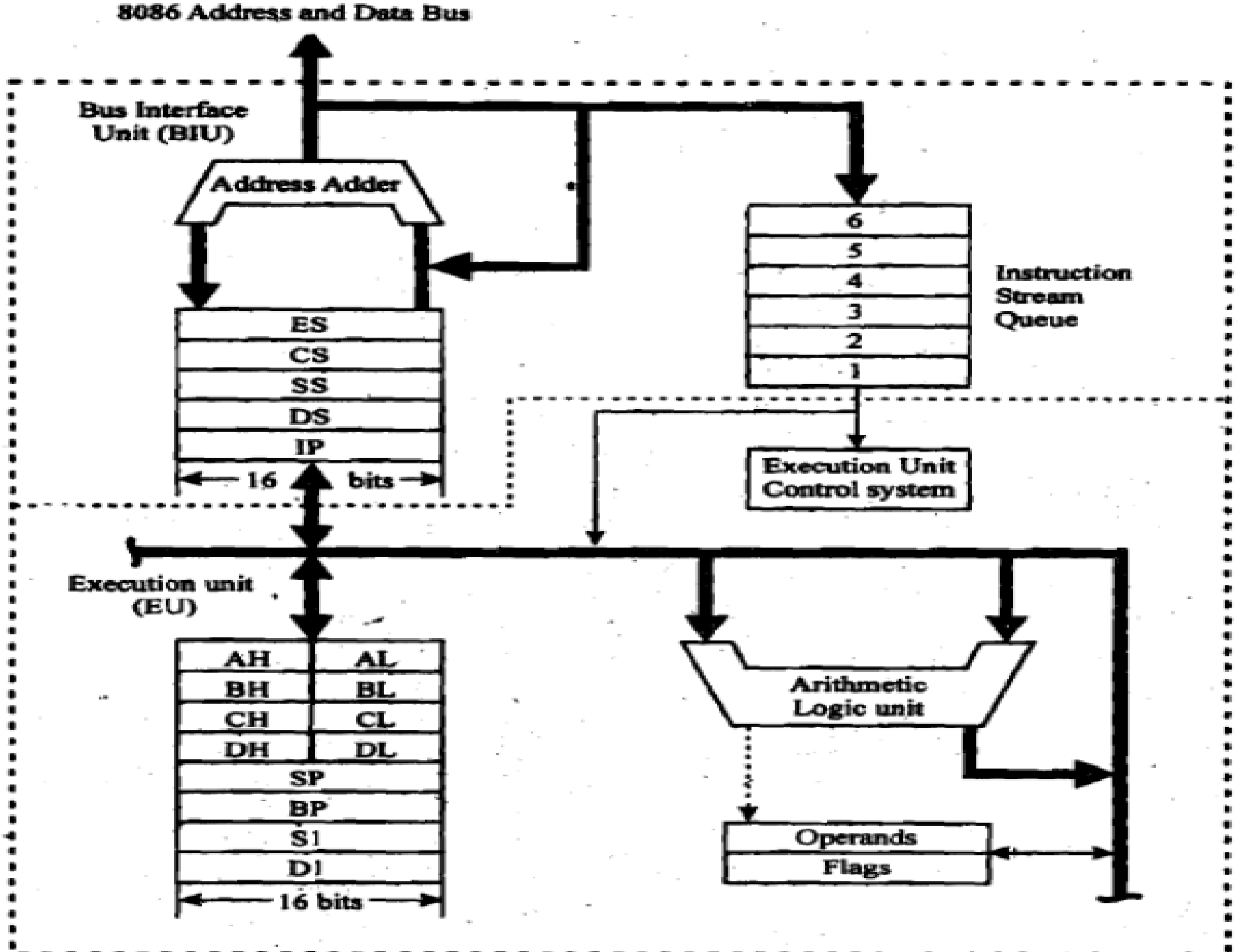
- **Operates in two modes:-**8086 operates in two modes:
  - **Minimum Mode:** A system with only one microprocessor.
  - **Maximum Mode:** A system with multiprocessor.

- **8086 uses memory banks:-**The 8086 uses a memory banking system. It means entire data is not stored sequentially in a single memory of 1 MB but memory is divided into two banks of 512KB.

- **Interrupts:-**8086 has 256 vectored interrupts.

- **Multiplication And Division:-**8086 has a powerful instruction set. So that it supports Multiply and Divide operation.

Architecture of 8086



Architecture of 8086

- The architecture of 8086 includes
  - Arithmetic Logic Unit (ALU)
  - Flag Register
  - General Registers
  - Instruction Stream Byte Queue
  - Segment Registers

EU & BIU

- The 8086 CPU logic has been partitioned into two functional units namely **Bus Interface Unit (BIU) and Execution Unit (EU).**

- The major **reason for this separation is to increase the processing speed** of the processor.

- The **BIU** has to interact with memory and input and output devices in **fetching** the instructions and data required by the EU.

- **EU** is responsible for **executing** the instructions of the programs and to carry out the required processing.

BUS INTERFACE UNIT (BU)

The BIU performs all bus operations for EU.

- **Fetching instructions**
- **Responsible for executing all external bus cycles.**
- **Read operands and write result.**

EXECUTION UNIT (EU)

Execution unit contains the complete infrastructure required to execute an instruction.

Bus Interface Unit

- The BIU has
  - Instruction Byte Queue
  - Segment Registers
  - Instruction Pointer

BIU – Instruction Byte Queue

- 8086 instructions vary from 1 to 6 bytes.

- Therefore **fetch and execution are taking place concurrently in order to improve the performance of the microprocessor.**

- The BIU feeds the instruction stream to the execution unit through a 6 byte prefetch queue.

BIU – Instruction Byte Queue

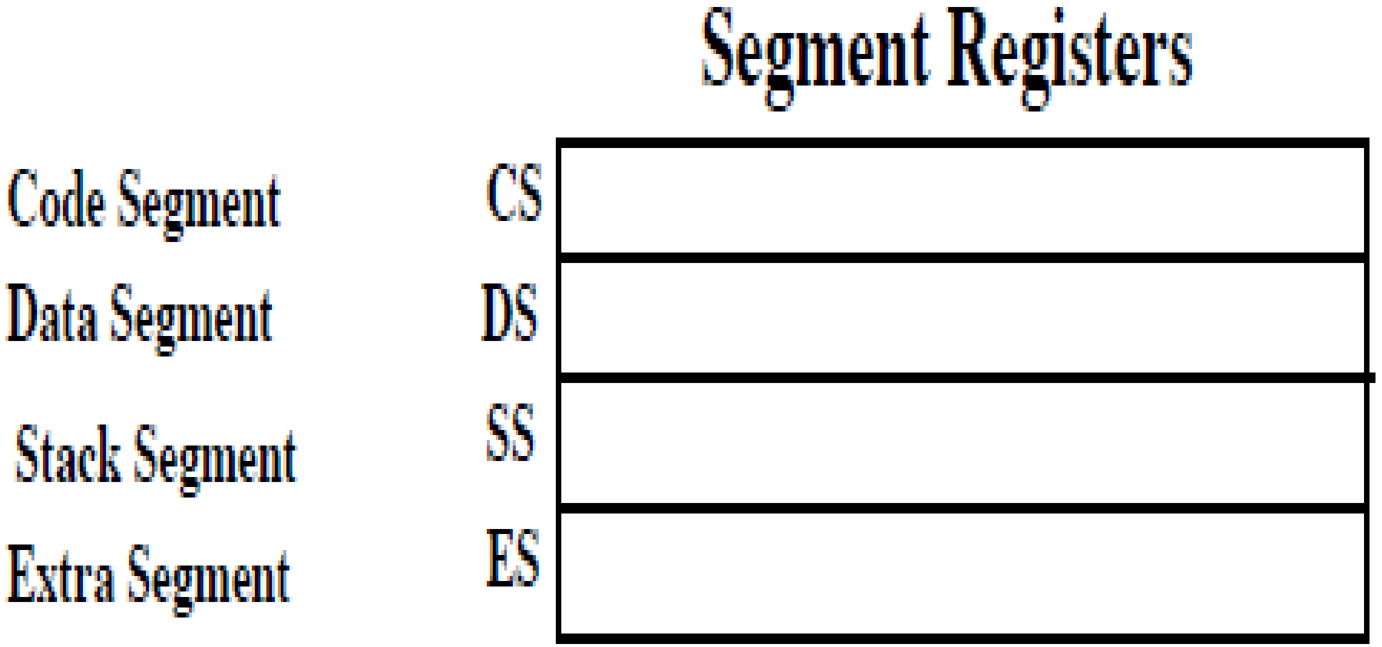
- Execution and decoding of certain instructions **do not require the use of buses.**

- While such instructions are executed, the **BIU fetches up to six instruction bytes for the following instructions (subsequent instructions).**

- The BIU store these prefetched bytes in a **first-in-first out** register by name **instruction byte queue**.

- When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in BIU.

- **Segment Registers**



Different Areas in Memory

- **Program memory** – The executable programs from the memory.
- **Data memory** – The processor can access the secondary data in any one out of four available segments.
- **Stack memory** – A stack is a section of the memory set aside to store addresses and data while a subprogram executes.
- **Extra segment** – This segment is also similar to data memory where additional secondary data may be stored and maintained.

Segment Registers

- **Code Segment (CS)** register is a 16-bit register containing address of 64 KB segment with Processor Instructions.
- The processor uses CS segment for all accesses to instructions referenced by Instruction Pointer (IP) register.

- **Stack Segment (SS)** register is a 16-bit register containing address of 64KB segment with Program Stack.
- By default, the processor assumes that all data referenced by the Stack Pointer (SP) and Base Pointer (BP) registers is located in the stack segment.

- **Data Segment (DS)** register is a 16-bit register containing address of 64KB segment with Program Data.
- By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and Index Register (SI, DI) is located in the data segment.

- **Extra Segment (ES)** register is a 16-bit register containing address of 64KB segment, usually with Program Data.
- By default, the processor assumes that the DI register references the ES segment in string manipulation instructions.

Execution Unit

- The Execution Unit (EU) has
  - Control Unit
  - Arithmetic And Logical Unit (ALU)
  - General Registers
  - Flag Register
  - Pointers
  - Index Registers

Execution Unit

- Control unit is **responsible for the co-ordination of all other units of the processor.**

- ALU performs various arithmetic and logical operations over the data.

- The Control Unit **translates the instructions fetched from the memory into a series of actions that are carried out by the EU.**

Programmer's Model of 8086

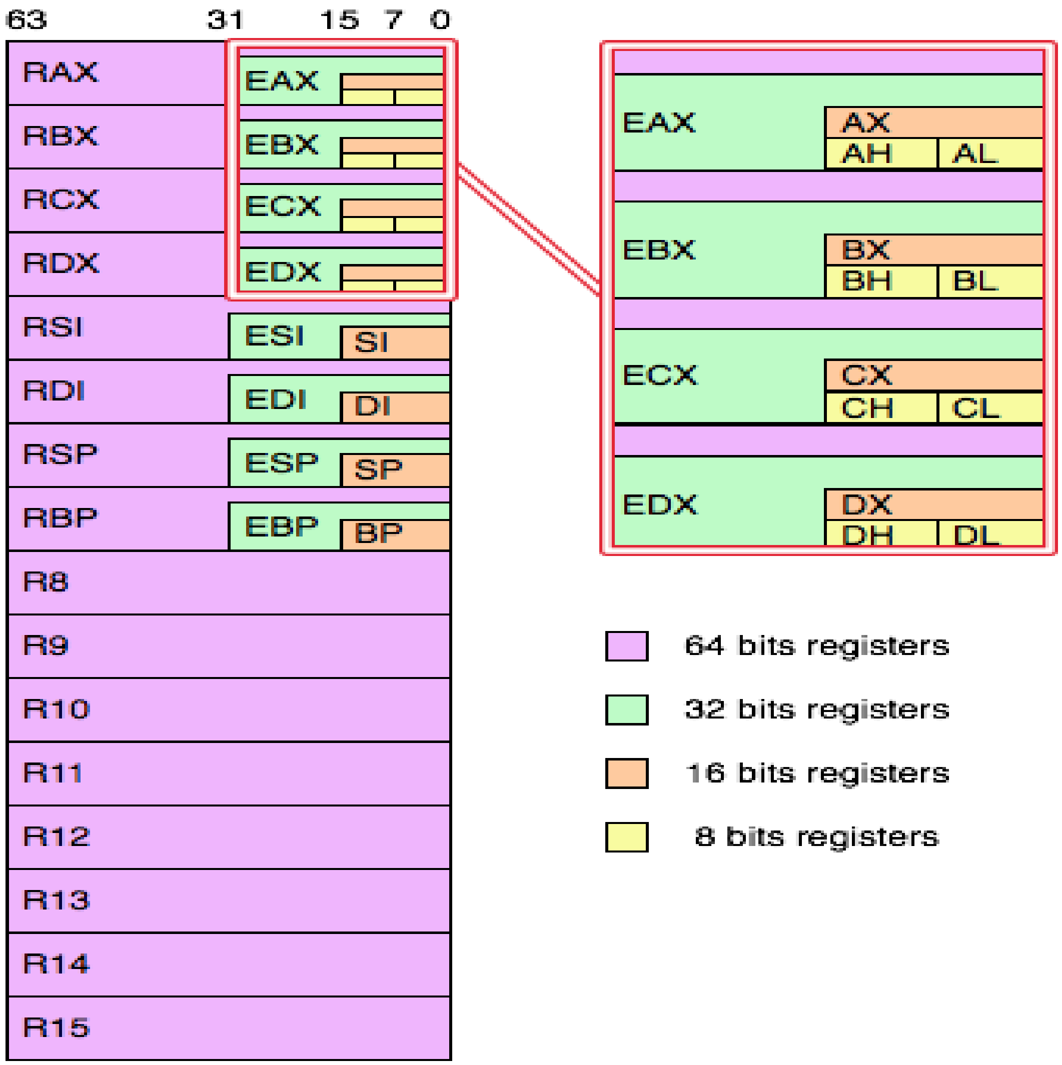
- **General Purpose Register**

		7 0	7 0
Accumulator	AX	AH	AL
Base	BX	BH	BL
Count	CX	CH	CL
Data	DX	DH	DL

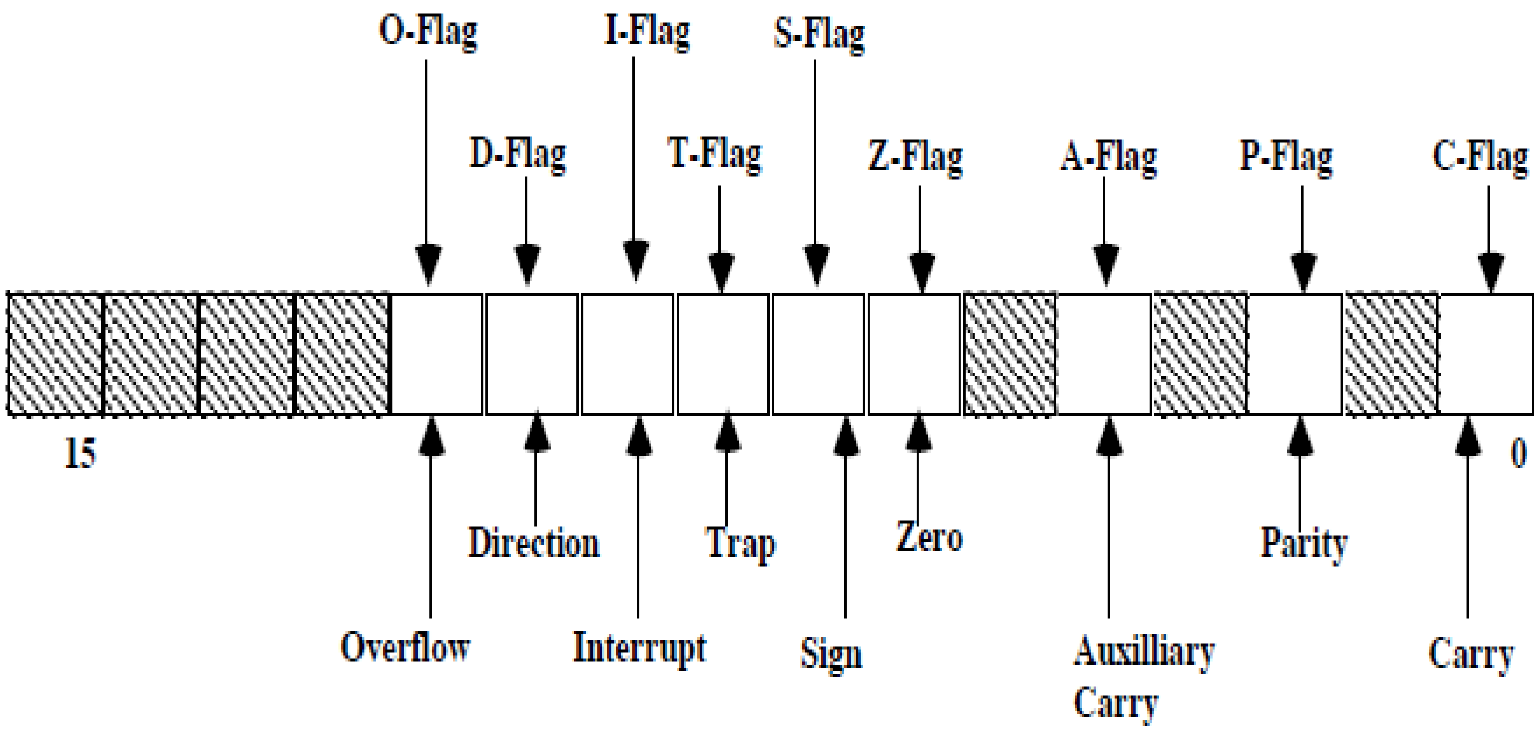
H: High Order Byte

L: Low Order Byte

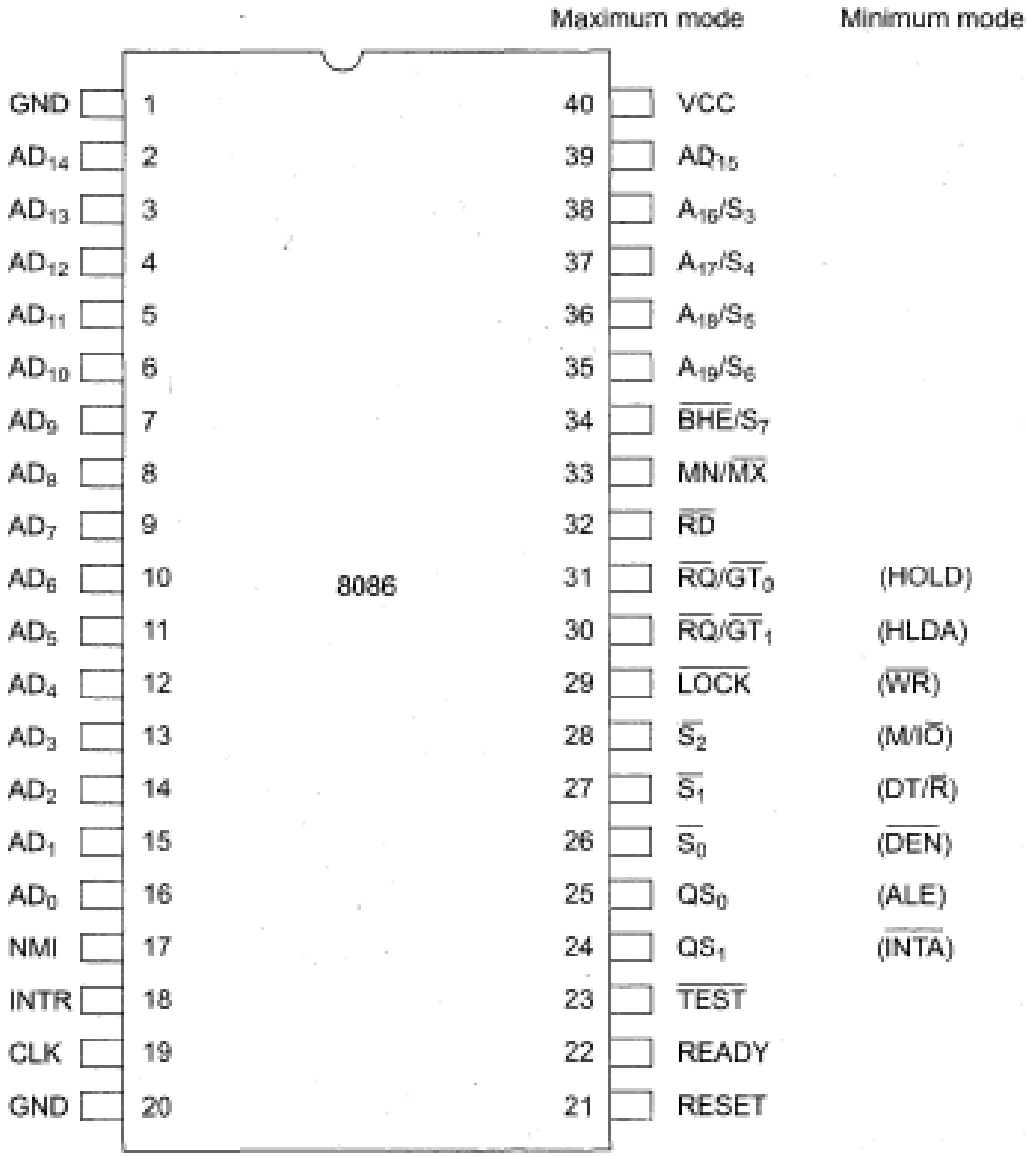
General purpose registers

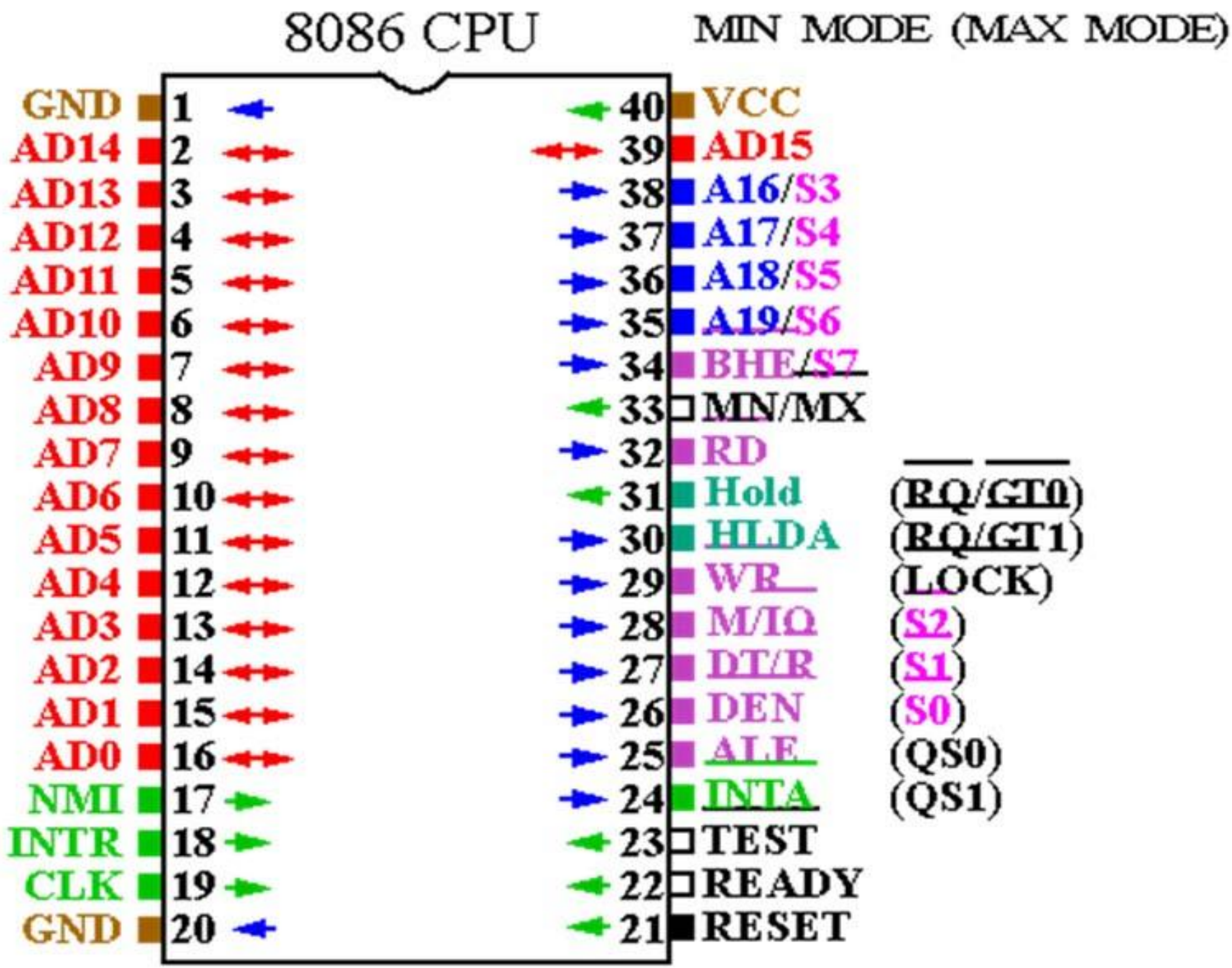


- **Flag Register**



Pin Diagram 8086




8086 CPU
MIN MODE (MAX MODE)
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE/S7
33 MN/MX
32 RD
31 Hold (RQ/GT0)
30 HLDA (RQ/GT1)
29 WR (LOCK)
28 M/IO (S2)
27 DT/R (S1)
26 DEN (S0)
25 ALE (QS0)
24 INTA (QS1)
23 TEST
22 READY
21 RESET

8086 can be work in two modes

- Minimum Mode: For single processor systems.
- Maximum Mode: For system with two or more processors.

Depending upon modes signals can be divided into

- Signals having common functions in both modes
- Signals for Minimum Mode
- Signals for Maximum Mode

Signals having common functions in both modes

- **AD15 – AD0: Address /Data Bus (BI)**

  T1 state: Address Bus

  T2,T3,Tw and T4: Data Bus

- **A19/S6 - A16/S3: Address/Status (OP)**

  T1 state : used to address upper 4 bits of address.

  T2, T3, Tw and T4 : Used to output status.

S3 and S4 indicates segment registers being used

S5: Status of Interrupt enable flag updated every clock cycle.

S6: When 8086 Shared system bus, then goes low or goes high.

S4	S3	Register
0	0	ES
0	1	SS
1	0	CS or none
1	1	DS

- **BHE(#)/S7: BHE (Bus High Enable) (OP)**

Low: If transfer is using higher order bytes (AD15-AD8)

High: If transfer is using lower order bytes (AD7-AD0)

S7 is used with HOLD Pin.

BHE	A0	Characteristics
0	0	Whole word
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

- **NMI (NON-MASKABLE INTERRUPT) (IP)**

  Interrupts can not be avoided.

- **RESET: (IP)**

  Causes the processor to immediately terminate its present activity.

- **CLK: (IP)**

  Provides the basic timing for the processor and bus controller. Power supply given to the system is converted to CLK signals by the Clock Generator.

- **READY: (IP)**

  It is the acknowledgement from the addressed <u>memory or I/O device that it is ready for the data transfer.</u> Otherwise 8086 will move to wait state.

- **TEST (#): (IP)**

This signal is used by WAIT instruction. Execution will continue, until TEST is low else it will be in idle state. TEST is synchronized internally during each clock cycle.

- **RD (#): (OP)**

This signal remains low when 8086 is reading data from memory or I/O devices.

- **MN/MX (#): (IP)**

Indicates what mode the processor is to operate in.

- **GND: Ground (OP)**

To prevent 8086 from thermal heating ,two ground signals are used.

- **VCC: (IP)**

+5V power supply pin.

Signals functions in Maximum modes

- **QS1,QS0 : (OP)**

Reflects status of Instruction Queue.

QS1	QS0	Status
0	0	No Operation
0	1	First Byte of Op Code from Queue
1	0	Queue is empty
1	1	Subsequent Byte from Queue

- **S2,S1,S0 (#): (OP)**

Indicates type of transfer takes place during current bus cycle.

S2	S1	S0	Machine Cycle
0	0	0	Interrupt ACK
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt

S2	S1	S0	Machine Cycle
1	0	0	Instruction Fetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Inactive-Passive (In T3)

- **LOCK: (OP)**

Bus is not used by another processor, current system have locked the rights. Used for notifying to another processors.

- **RQ(#)/GT1(#)&RQ(#)/GT0(#): (Bus request/Bus Grant) (BI)**

Using bus request signal another master can request a system bus and processor sends a grant signal as a acceptance.

RQ/GT0 is having greater priority than RQ/GT1.

Signals functions in Minimum modes

- **INTA (Interrupt Ack): (OP)**

It indicates recognition of an interrupt request**. It then sends two negative going pulse in next to clk cycles,** first informs interface that its interrupt request in accepted, in **next pulse interface replies with interrupt type.**

- **ALE (Address Latch Enable): (OP)**

It is provided to demultiplex AD0-AD15 to A0-A15 and D0-D15.

- **DEN (#) (Data Enable): (OP)**

This signal informs transceivers that 8086 is ready to send or receive data.

- **DT/R (#) (Data Transmit/Receive): (OP)**

It is used to control the direction of data flow through the transceiver.

High: 8086 is transmitting data

Low: 8086 is receiving data

- **M/IO (#) : (OP)**

It is used to distinguish a memory access from an I/O access.

High: Memory Access

Low: I/O Access

- **WR (#) (WRITE): (OP)**

Indicates that the processor is performing a writing data to memory or I/O.

- **HOLD (IP) / HLDA (OP) :**

HOLD: indicates that DMA master is requesting a local bus, on request processor replies High HLDA signal as a Ack.

Minimum-Mode and Maximum-Mode System (cont.)

Common signals		
Name	Function	Type
AD7-AD0	Address/data bus	Bidirectional, 3-state
A15-A8	Address bus	Output, 3-state
A19/S6-A16/S3	Address/status	Output, 3-state
MN/$\overline{MX}$	Minimum/maximum Mode control	Input
$\overline{RD}$	Read control	Output, 3-state
$\overline{TEST}$	Wait on test control	Input
READY	Wait state control	Input
RESET	System reset	Input
NMI	Nonmaskable Interrupt request	Input
INTR	Interrupt request	Input
CLK	System clock	Input
V_{CC}	+5 V	Input
GND	Ground	

Signals common to both minimum and maximum mode

Minimum-Mode and Maximum-Mode System (cont.)

Minimum mode signals ($MN/\overline{MX} = V_{CC}$)		
Name	**Function**	**Type**
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
$\overline{WR}$	Write control	Output, 3-state
$IO/\overline{M}$	IO/memory control	Output, 3-state
$DT/\overline{R}$	Data transmit/receive	Output, 3-state
$\overline{DEN}$	Data enable	Output, 3-state
$\overline{SSO}$	Status line	Output, 3-state
ALE	Address latch enable	Output
$\overline{INTA}$	Interrupt acknowledge	Output

Unique minimum-mode signals

Minimum-Mode and Maximum-Mode System (cont.)

Maximum mode signals ($MN/\overline{MX}$ = GND)		
Name	**Function**	**Type**
$\overline{RQ}/\overline{GT1}, 0$	Request/grant bus access control	Bidirectional
$\overline{LOCK}$	Bus priority lock control	Output, 3-state
$\overline{S2}$ - $\overline{S0}$	Bus cycle status	Output, 3-state
QS1, QS0	Instruction queue status	Output

Unique maximum-mode signals

Segmentation in 8086

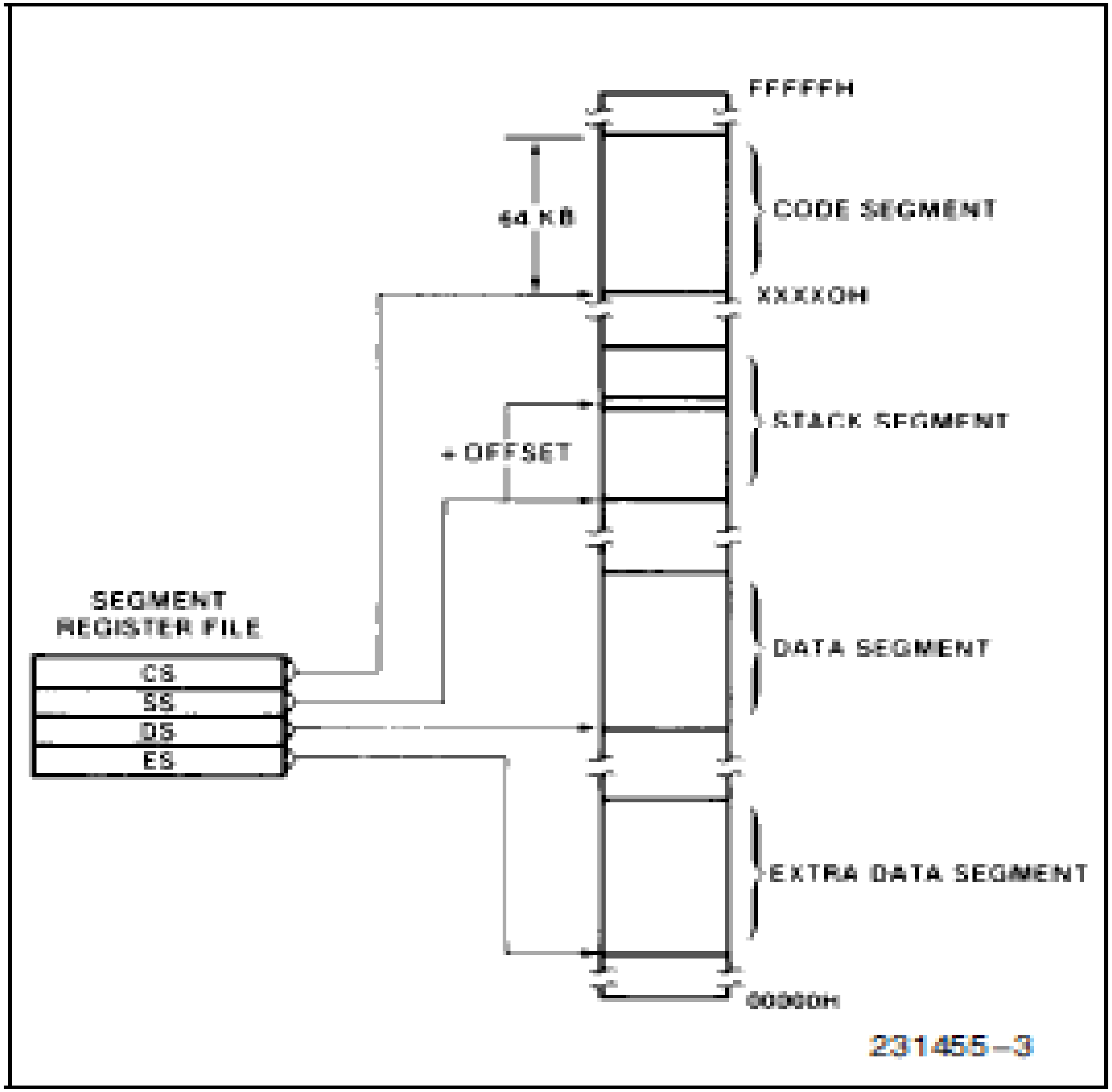
- The process of dividing memory is called Segmentation.

- Intel 8086 has 20 lines address bus.

- With 20 address lines, the memory that can be addressed is 2^20 bytes

  2^20 = 1,048,576 bytes (1 MB).

- 8086 can access memory with address ranging from 00000 H to FFFFF H.



FFFFFH
64 KB
CODE SEGMENT
XXXX0H
STACK SEGMENT
+ OFFSET
DATA SEGMENT
EXTRA DATA SEGMENT
SEGMENT REGISTER FILE
CS
SS
DS
ES
00000H
231455-3

- In 8086, memory has four different types of segments.

These are:

  - **Code Segment**
  - **Data Segment**
  - **Stack Segment**
  - **Extra Segment**

- These registers are 16-bit in size.
- Each register stores the base address (starting address) of the corresponding segment.
- Because the segment registers cannot store 20 bits, they only store the upper 16 bits.

Logical to physical address Translation in 8086

- The 20-bit address of a byte is called its **Physical Address.**

- High level languages have a **Logical Address.**

- Logical address is in the form of:

  Base Address : Offset

- Offset is the displacement of the memory location from the starting location of the segment.

Example

- The value of Data Segment Register (DS) is 2222 H.

- To convert this 16-bit address into 20-bit, the BIU appends 0H to the LSBs of the address.

- After appending, the starting address of the Data Segment becomes 22220H.

- If the data at any location has a logical address specified as:

  2222 H : 0016 H

- Then, the number 0016 H is the offset. 2222 H is the value of DS.

- To calculate the effective address of the memory, BIU uses the following formula:

  Effective Address =

  Starting Address of Segment + Offset

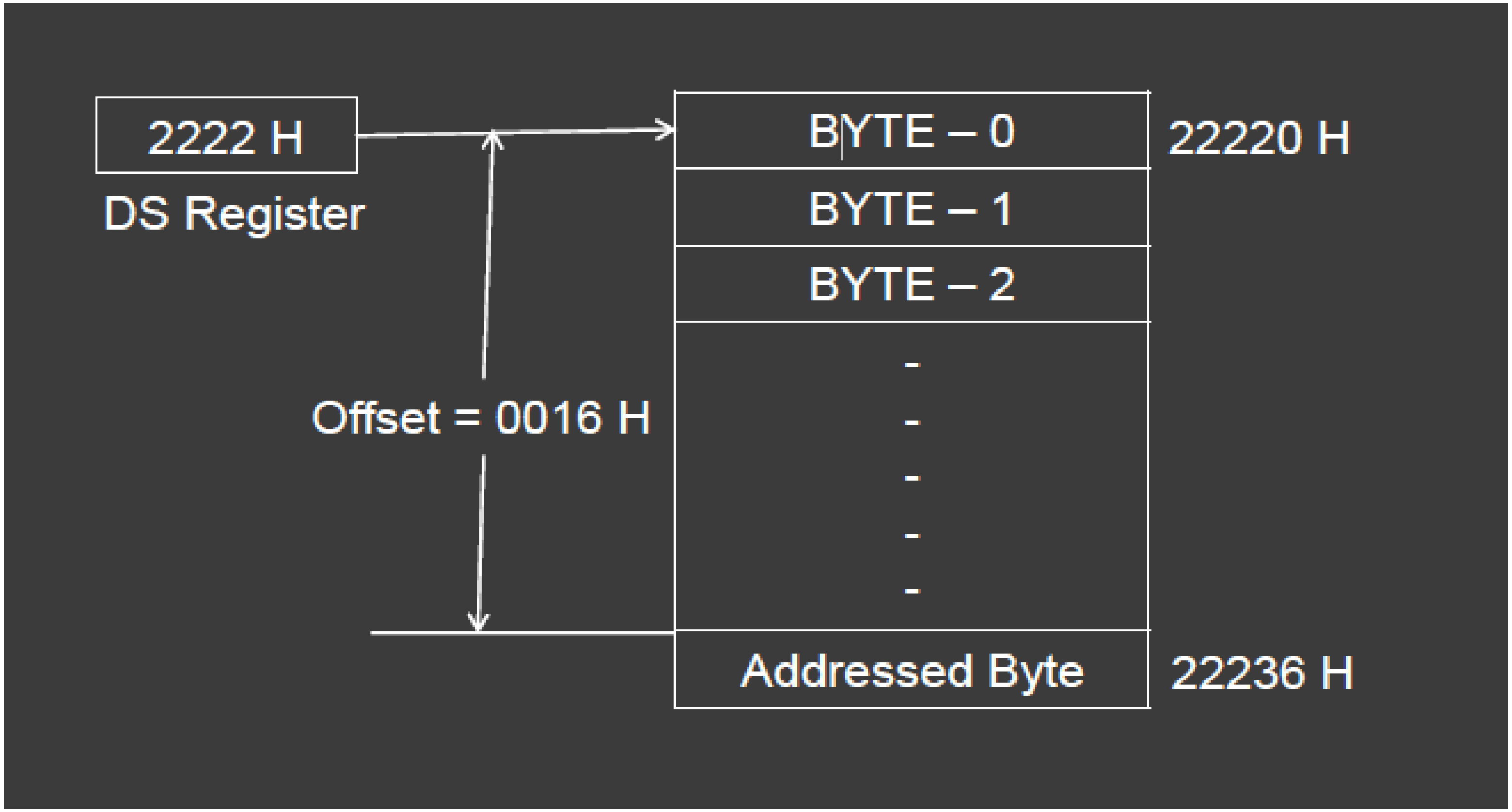
- To find the starting address of the segment, BIU appends the contents of Segment Register with 0H.
- Then, it adds offset to it.

  Therefore:

  EA = 22220 H

  \+ 0016 H

  22236 H



Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

Question 1

The contents of the following registers are:

- CS = 1111 H
- DS = 3333 H
- SS = 2526 H
- IP = 1232 H
- SP = 1100 H
- DI = 0020 H

Calculate the corresponding physical addresses for the address bytes in CS, DS and SS.

1. CS = 1111 H

- The base address of the code segment is 11110 H.
- Effective address of memory is given by 11110H + 1232H = **12342H.**

2. DS = 3333 H

- The base address of the data segment is 33330 H.
- Effective address of memory is given by 33330H + 0020H = **33350H.**

3. SS = 2526 H

- The base address of the stack segment is 25260 H.
- Effective address of memory is given by 25260H + 1100H = **26360H.**

Question 2

The contents of the following registers are:

- CS = 1234 H
- ES = 0014 H
- SS = 9526 H
- IP = 0042 H
- SP = 1800 H
- DI = 2020 H

Calculate the corresponding physical addresses for the address bytes in CS, ES and SS.

1. CS = 1234 H

- The base address of the code segment is 12340 H.
- Effective address of memory is given by 12340H + 0042H = **12382H.**

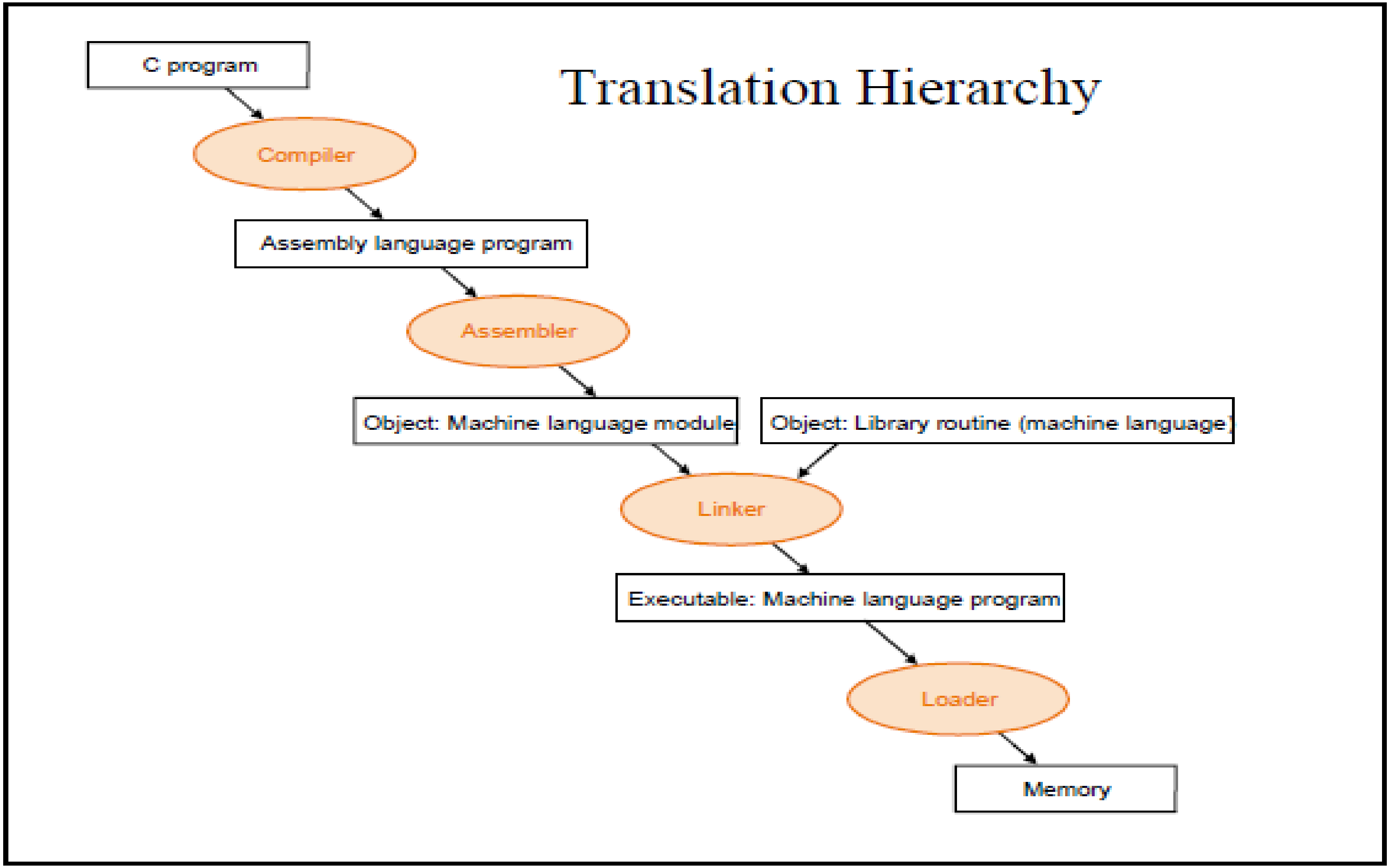
2. ES = 0014 H

- The base address of the data segment is 00140 H.
- Effective address of memory is given by 00140H + 2020H = **02160H.**

3. SS = 9526 H

- The base address of the stack segment is 95260 H.
- Effective address of memory is given by 95260H + 1800H = **96A60H.**

Assemblers, Linkers & Loaders



Assemblers

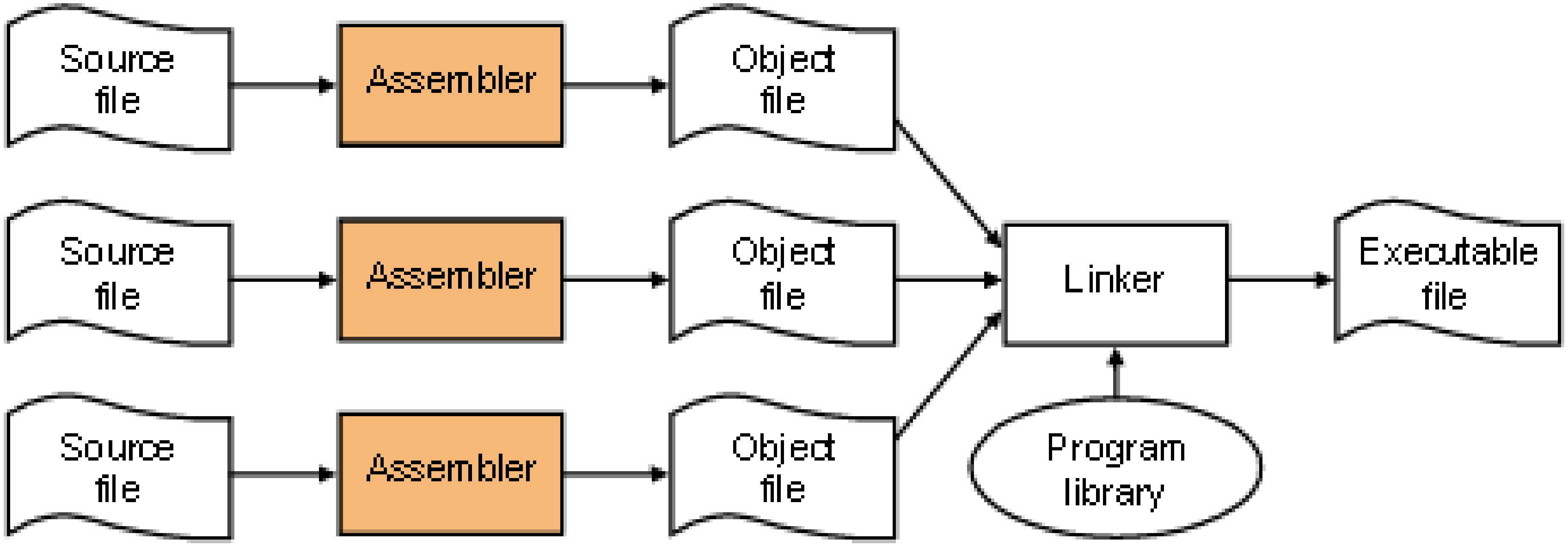
- **Assemblers need to**
  - translate assembly instructions and pseudo-instructions into machine instructions
  - Convert decimal numbers, etc. specified by programmer into binary
- **Typically, assemblers make two passes over the assembly file**
  - First pass: reads each line and records *labels* in a *symbol table*
  - Second pass: use info in symbol table to produce actual machine code for each line

Object file format

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

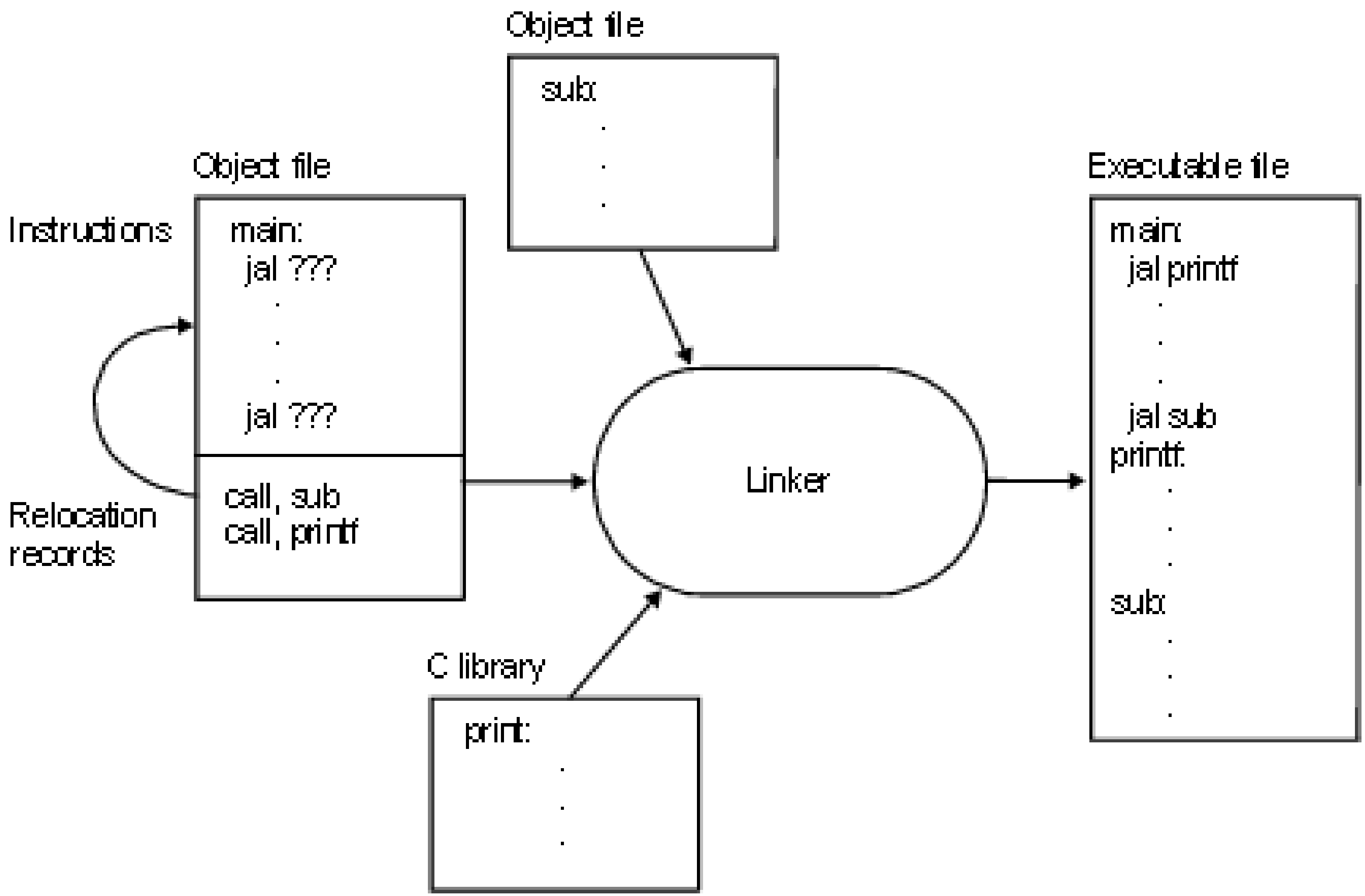
- Object file header describes the size and position of the other pieces of the file
- Text segment contains the machine instructions
- Data segment contains binary representation of data in assembly file
- Relocation info identifies instructions and data that depend on absolute addresses
- Symbol table associates addresses with external labels and lists unresolved references
- Debugging info

Process for producing an executable file



Linker

- **Tool that merges the object files produced by *separate compilation* or assembly and creates an executable file**
- **Three tasks**
  - Searches the program to find library routines used by program, e.g. printf(), math routines,...
  - Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
  - Resolves references among files

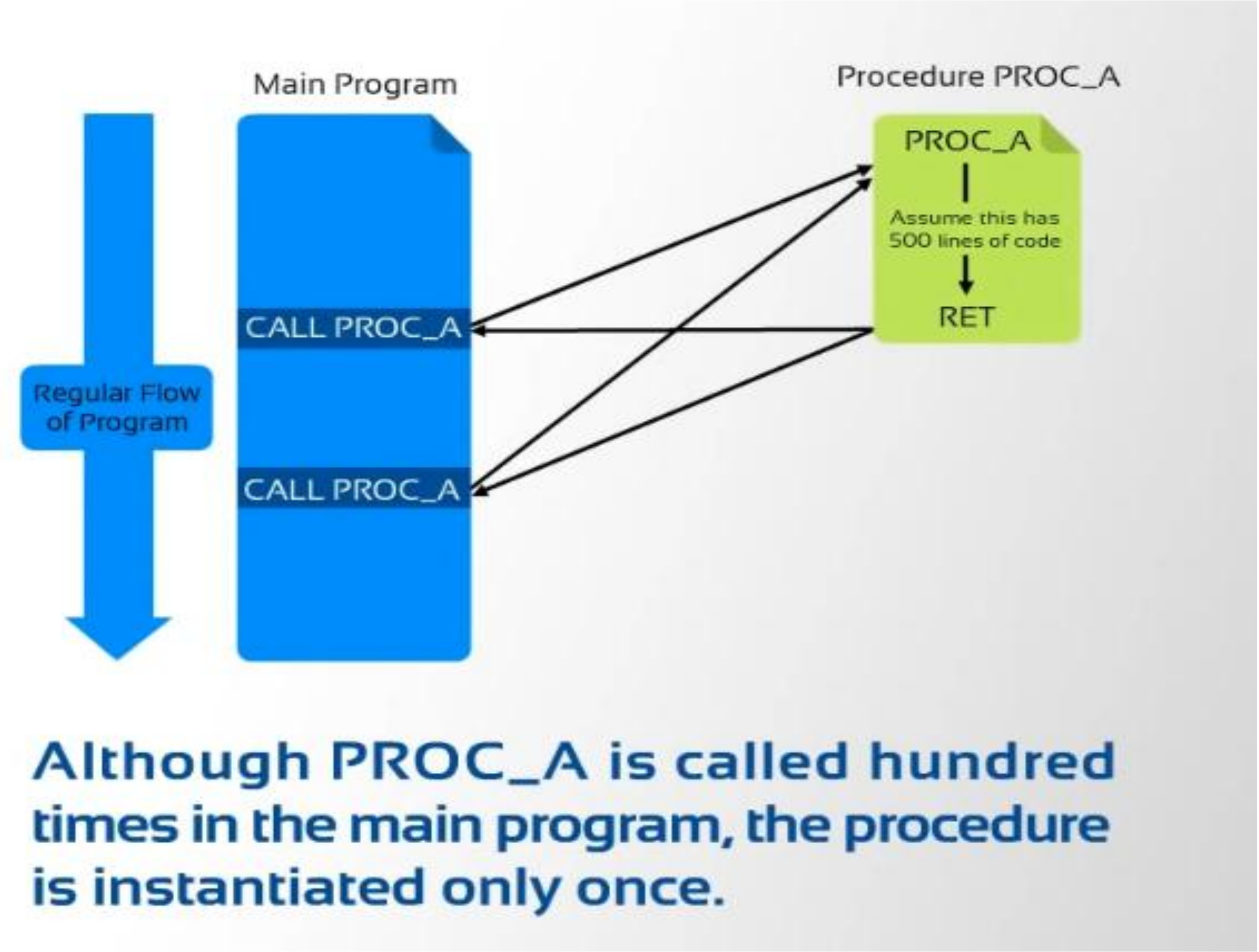

Object file
sub:
Object file
Instructions
main:
jal ???
jal ???
Relocation
records
call, sub
call, printf
Linker
Executable file
main:
jal printf
jal sub
printf:
sub:
C library
print:

Loader

- **Part of the OS that brings an executable file residing on disk into memory and starts it running**
- Steps
  - Read executable file's header to determine the size of text and data segments
  - Create a new address space for the program
  - Copies instructions and data into address space
  - Copies arguments passed to the program on the stack
  - Initializes the machine registers including the stack ptr
  - Jumps to a startup routine that copies the program's arguments from the stack to registers and calls the program's main routine

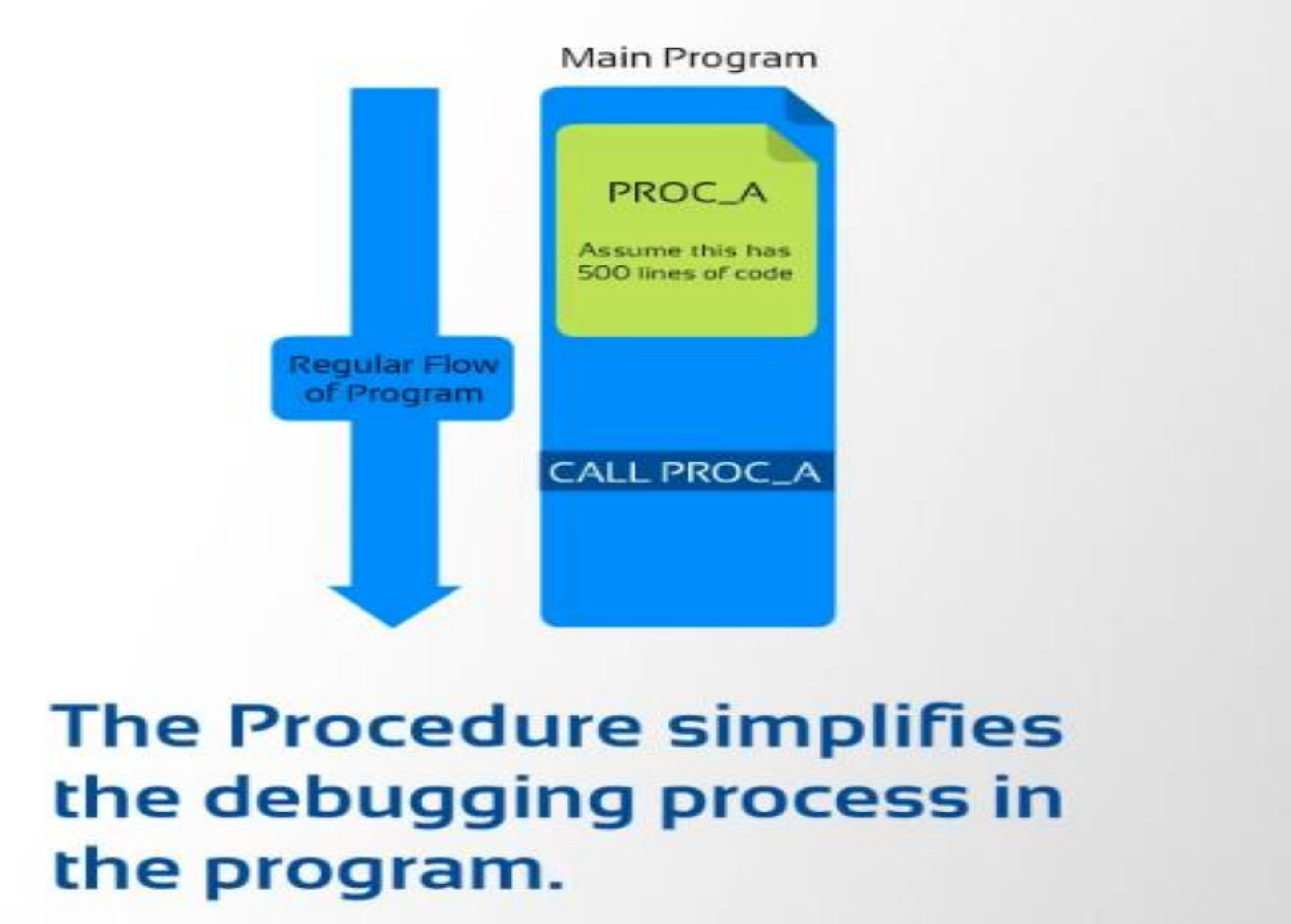
- Assembly language program
  - Assembly language program (.asm) file—known as source code
  - Converted to machine code by a process called assembling
  - Assembling performed by a software program—an 80x86 assembler
  - Machine (object) code that can be run is output in the executable (.exe) file
  - 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

Procedures

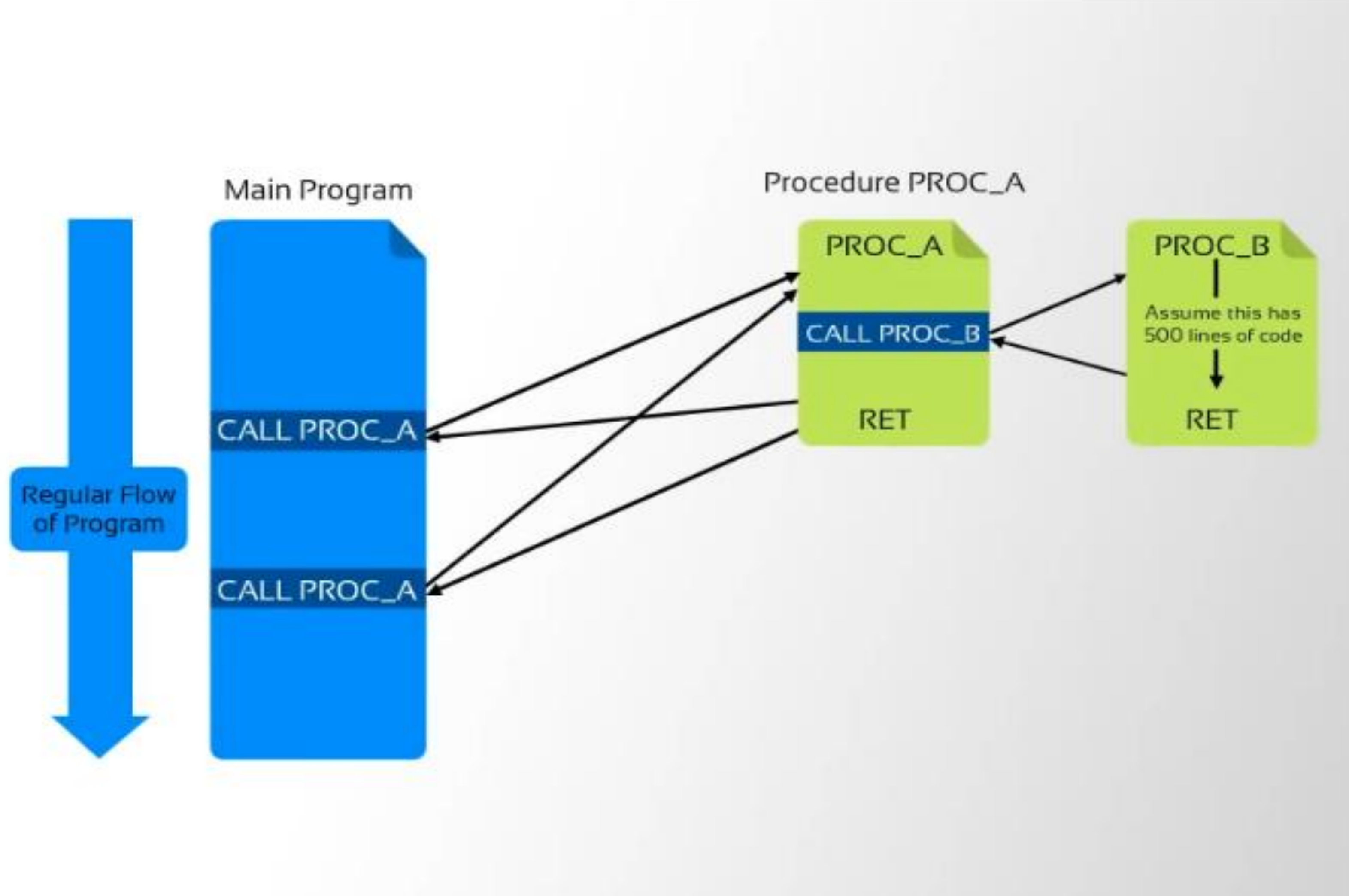


Main Program
Procedure PROC_A
PROC_A
Assume this has
500 lines of code
RET
CALL PROC_A
Regular Flow
of Program
CALL PROC_A
Although PROC_A is called hundred
times in the main program, the procedure
is instantiated only once.

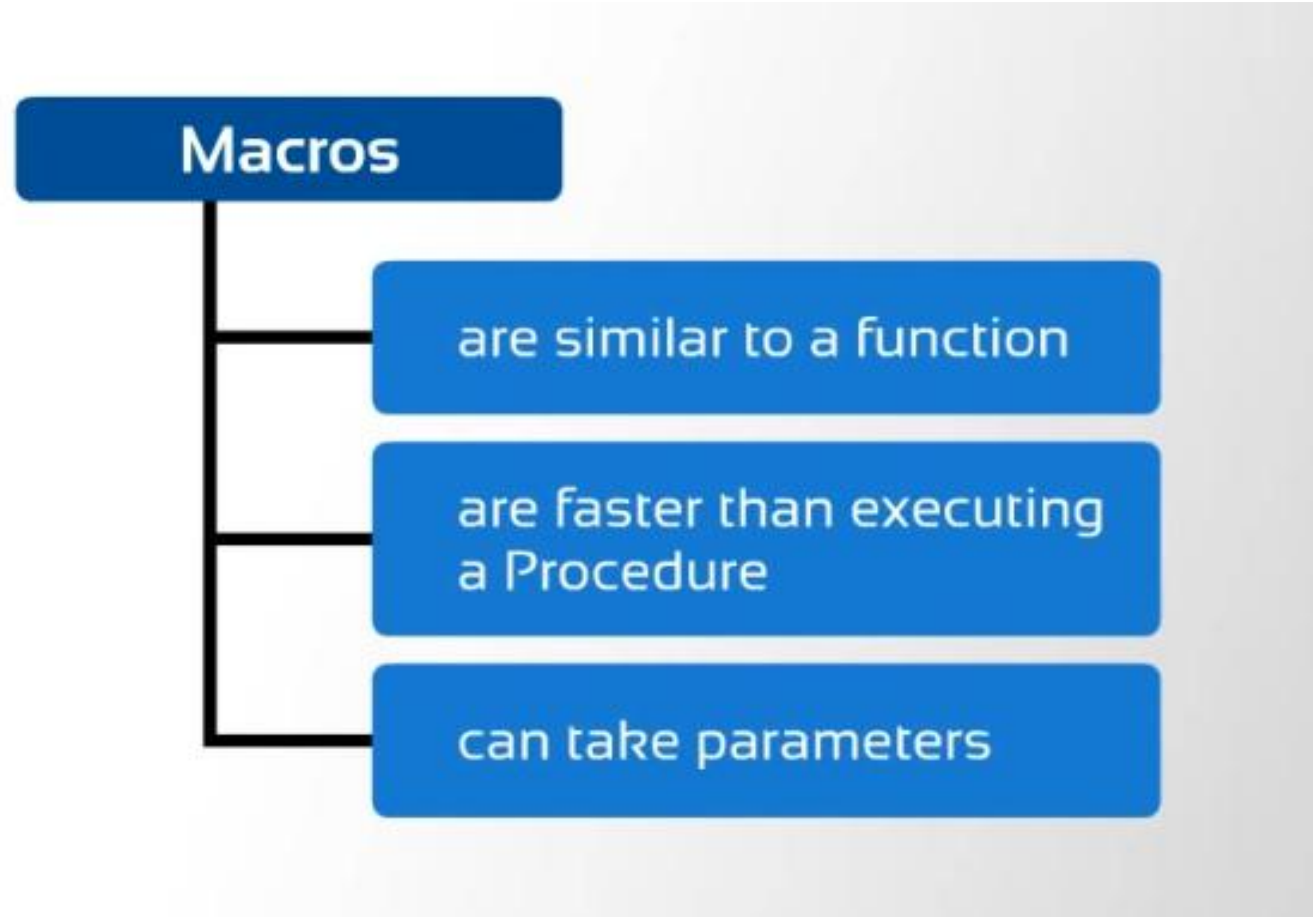
Why Procedures?



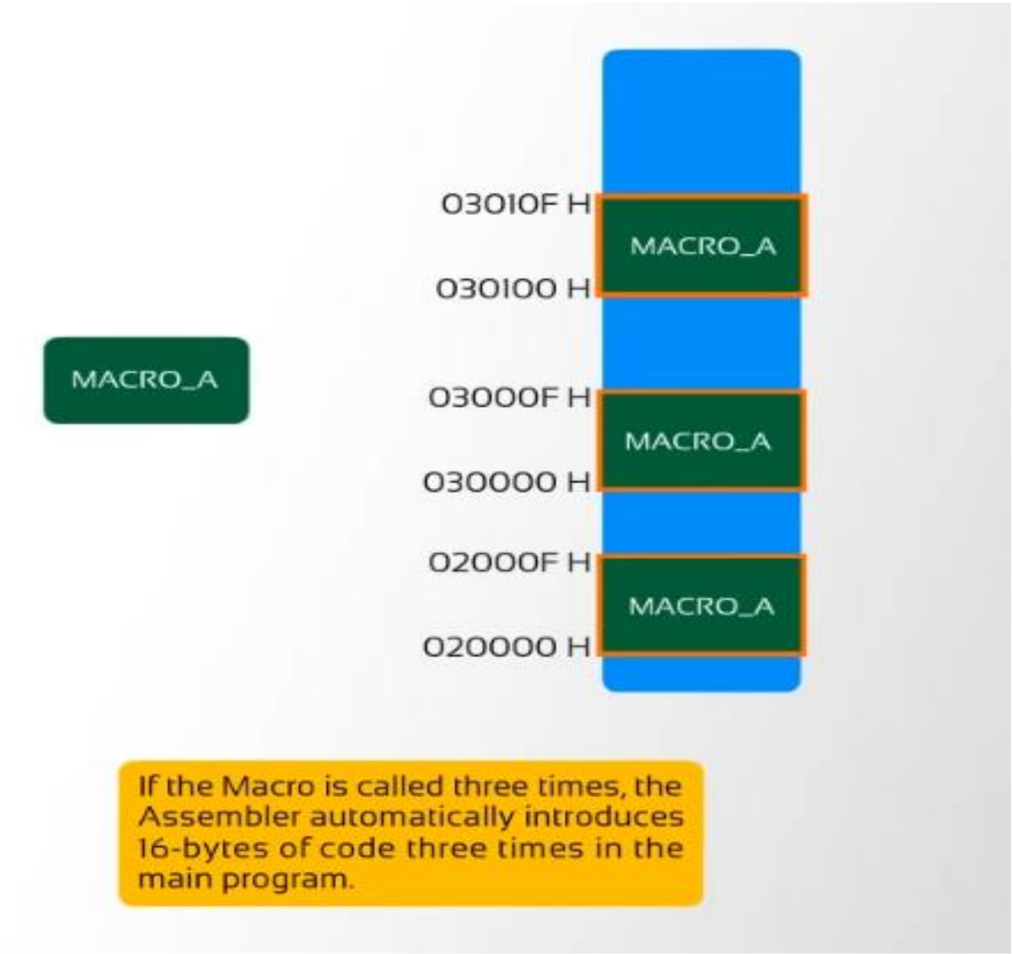
Nested Procedures



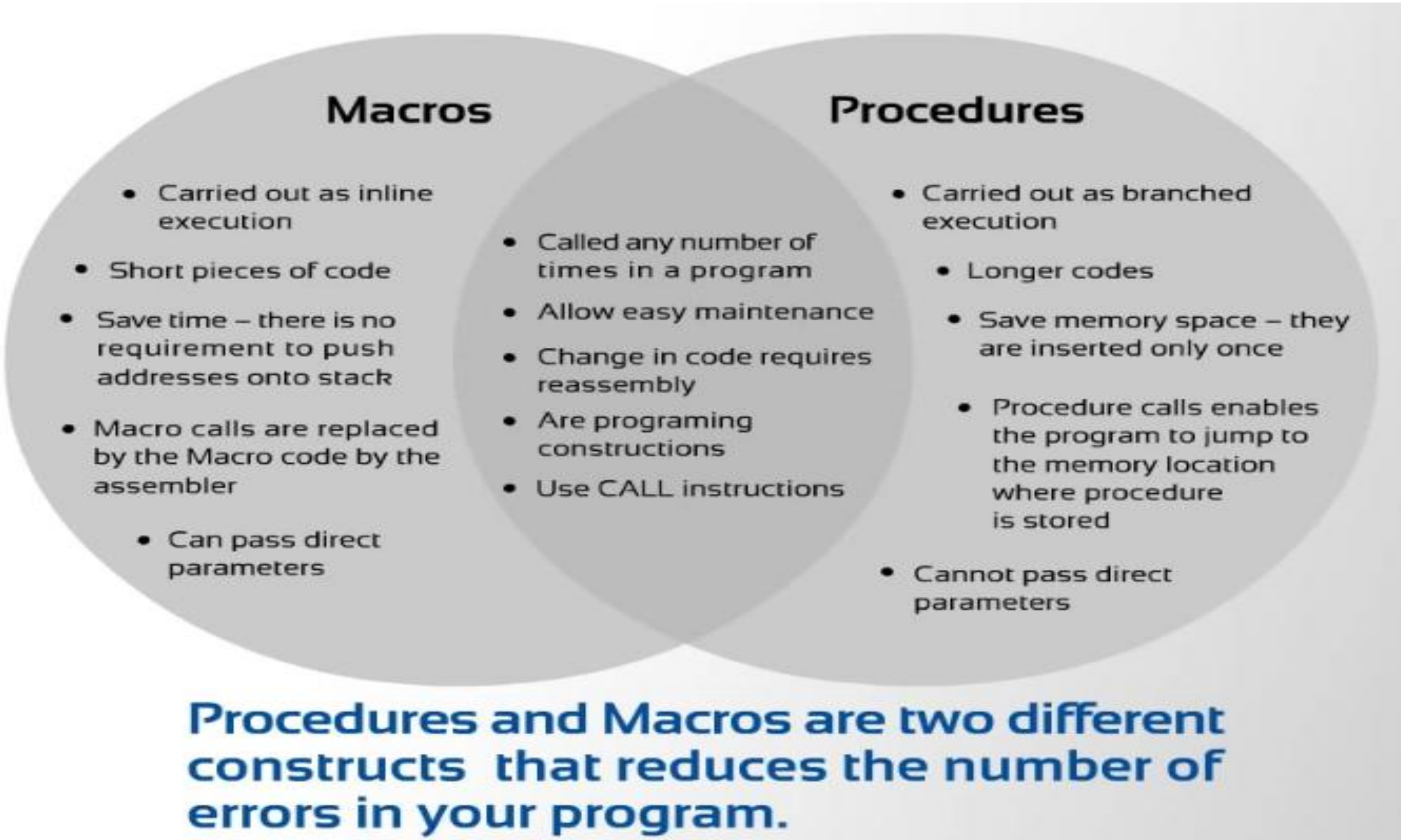
Macros


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

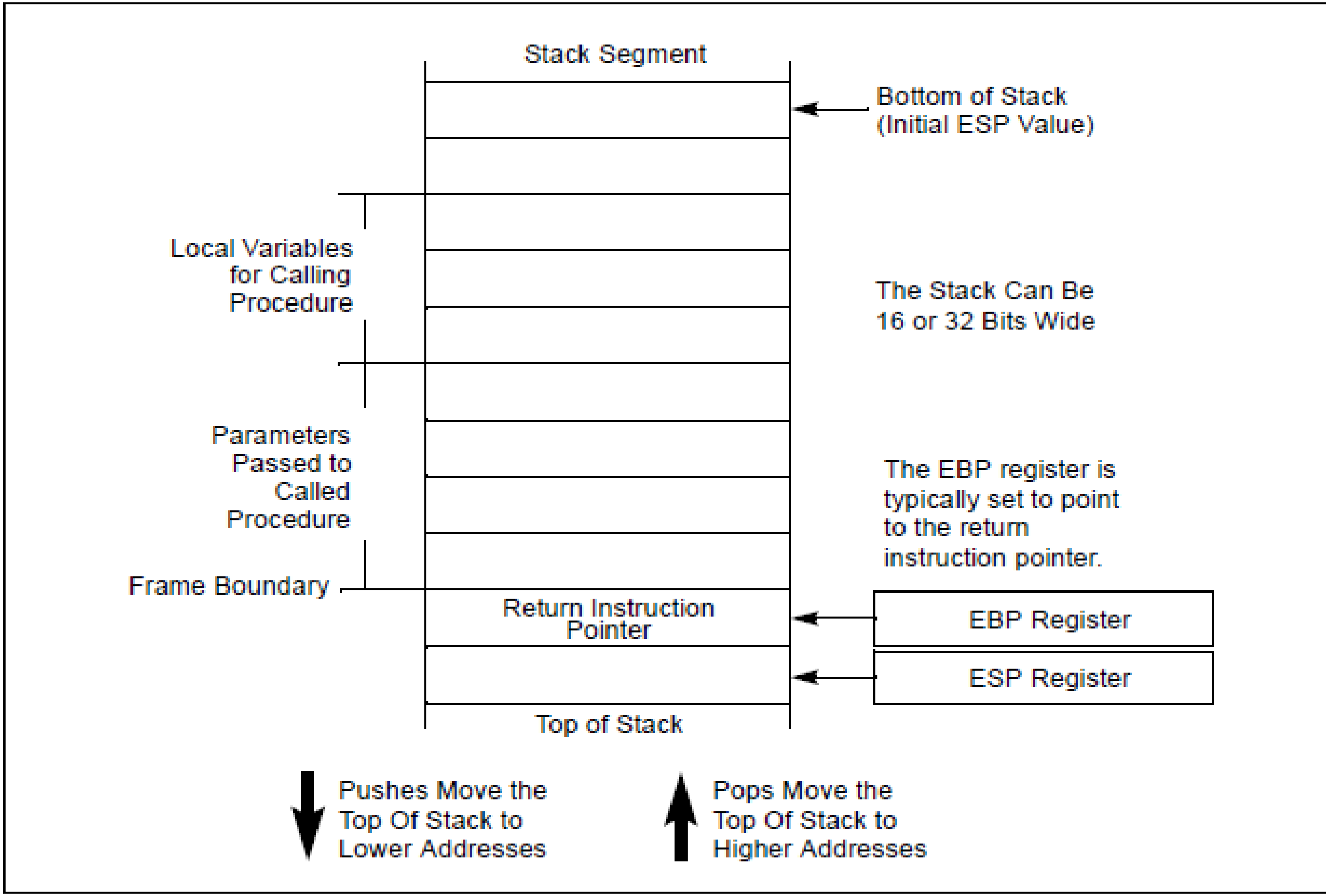
```
    count: resb 2
```

```
%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
```

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

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

- 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

	Kind of Instructions
1	Data Transfer Instructions
2	Arithmetic Instructions
3	Logical Instructions
4	Shift and Rotate Instructions
5	Branch Instructions
6	Loop Instructions
7	Processor Control Instructions
8	Flag Manipulation Instructions
9	String Manipulation Instructions

- **Flag Manipulation instructions**

  The Flag manipulation instructions directly modify some of the Flags of 8086.
  i. CLC – Clear Carry Flag.
  ii. CMC – Complement Carry Flag.
  iii. STC – Set Carry Flag.
  iv. CLD – Clear Direction Flag.
  v. STD – Set Direction Flag.
  vi. CLI – Clear Interrupt Flag.
  vii. STI – Set Interrupt Flag.

- **Machine Control instructions**

  The Machine control instructions control the bus usage and execution
  i. WAIT – Wait for Test input pin to go low.
  ii. HLT – Halt the process.
  iii. NOP – No operation.
  iv. ESC – Escape to external device like NDP
  v. LOCK – Bus lock instruction prefix.

Shift Instruction

Logical Shift

Fills the newly created bit with zero.

Arithmetic Shift

Fills the newly created bit with a copy of the number's sign bit.

How it works?

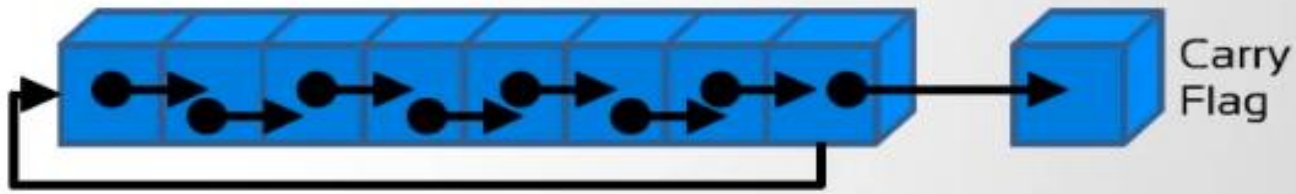
Instruction	Operand on which action will be performed	Number of times the operation will be performed
SAL or SHL or SAR or SHR	• A value in a register (Value can be shifted either right or left; can be either arithmetic right or logic right.) • A byte (8-bit) in memory • A word (16-bit) in memory • A double-word (32-bit) in memory **A shift can be left or right, arithmetic or logical, in memory or in register.**	• Blank (once) • The value in the CL register (E.g., If a CL register is loaded with value 4, it will shift either right or left, arithmetic or logical by 4.) • A defined number

Rotate Instructions

Rotate without Carry

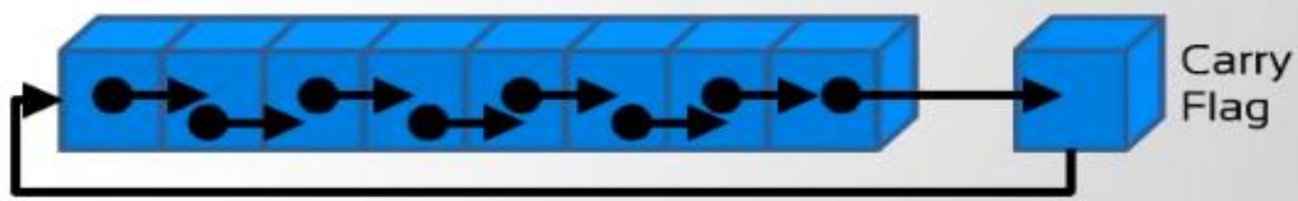
The bits are rotated right or left depending upon whether ROR or ROL instruction is used.

The bit rotated out gets copied into carry as well as into the extreme bit.

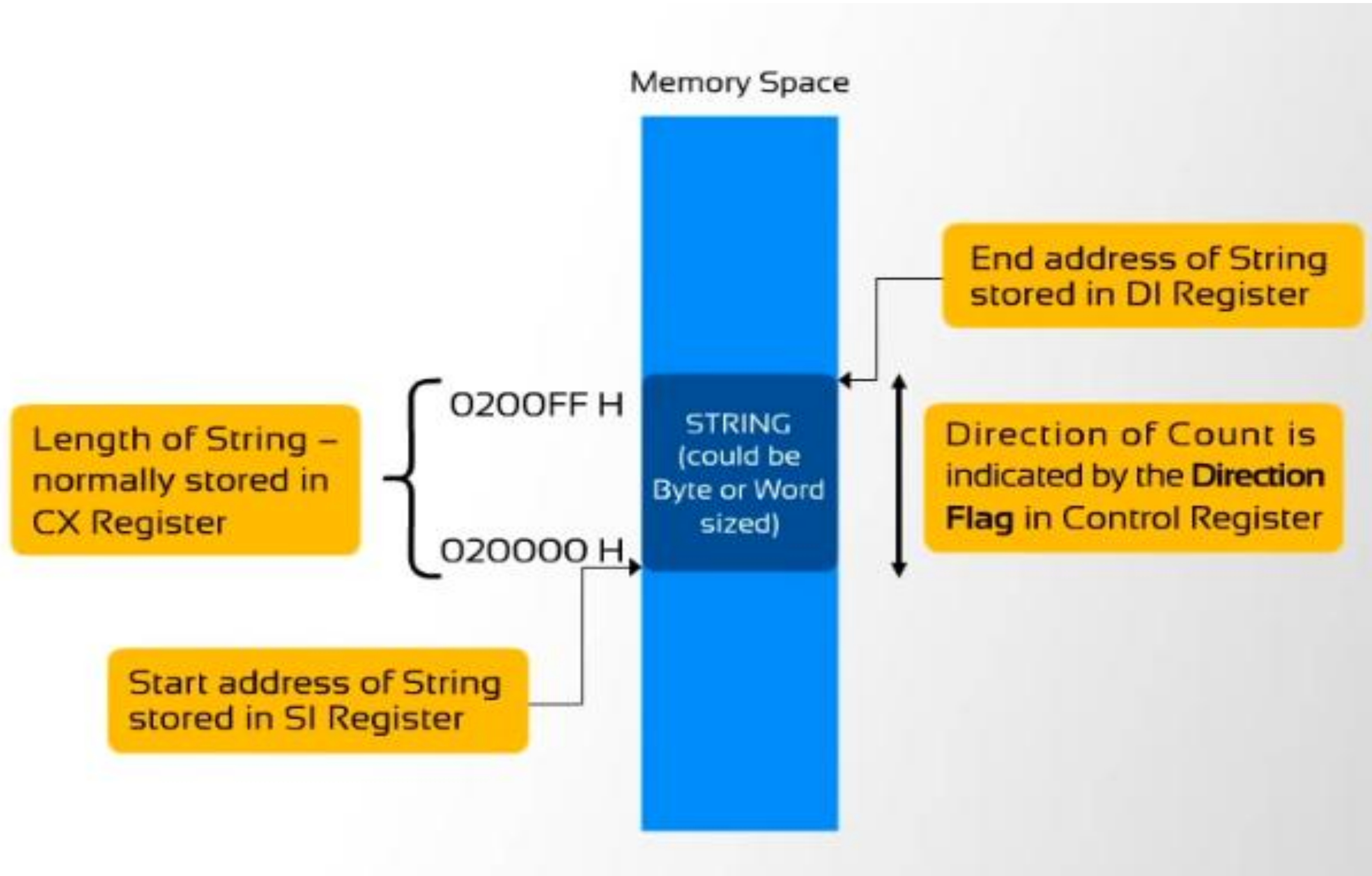


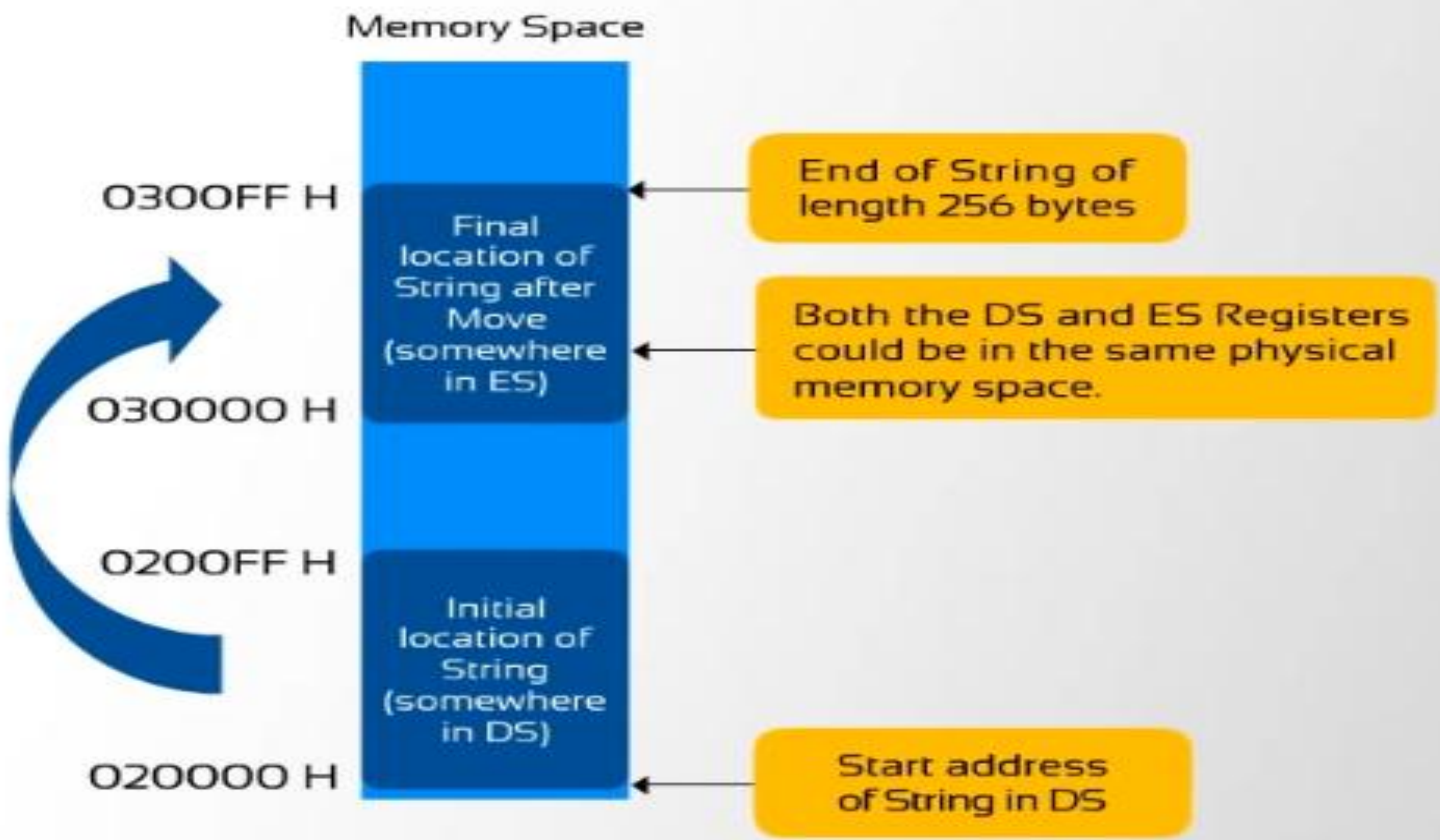
Rotate through Carry

The bits are rotated right or left through carry depending upon whether RCR or RCL instruction is used. The diagram below shows RCR.



String Instructions





MOVSB DST, SRC

MOVSW DST, SRC

If sting instructions are not used..

```
        MOV SI, OFFSET STRING1;      USE SI  AS SOURCE INDEX
        MOV DI, OFFSET STRING2;      USE DI  AS DESTINATION INDEX
        MOV CX, LENGTH STRING1;      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.

REP Instruction

• **These instructions are used along with string instructions only.**

	Condition 1	Condition 2	Instructions
REPE/REPZ	CX != ZERO	ZF = ONE	CMPSB, CMPSW, SCASB, SCASW
REPNE/REPNZ	CX != ZERO	ZF = ZERO	
REP	CX != ZERO	ZF = don't care	MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW

Addressing Modes

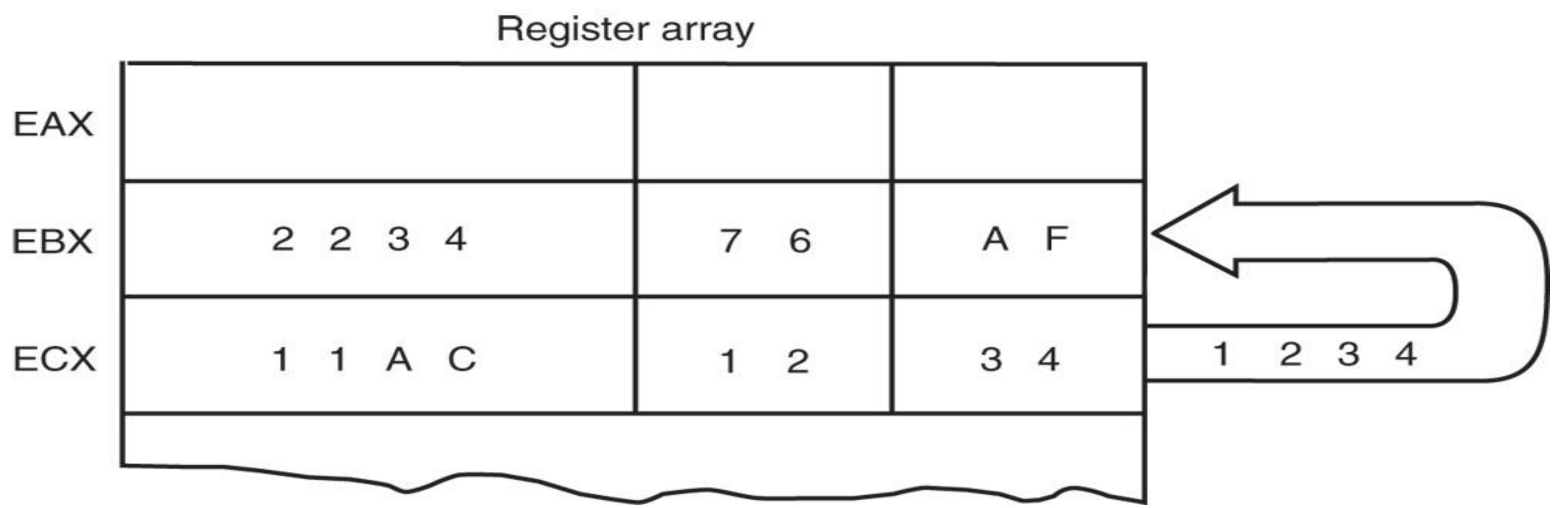
- What is Addressing Mode?
  - **The way operand is specified within an instruction,** i.e., either as an immediate operand or indirect operand or direct operand.
  - **The way to access Variables, Arrays, Records, Pointer and other complex data types.**

- **Types of addressing modes**
  - Register Addressing Modes
  - Immediate Operand Addressing
  - Memory Operand Addressing

- Each operand can use a different addressing Mode.

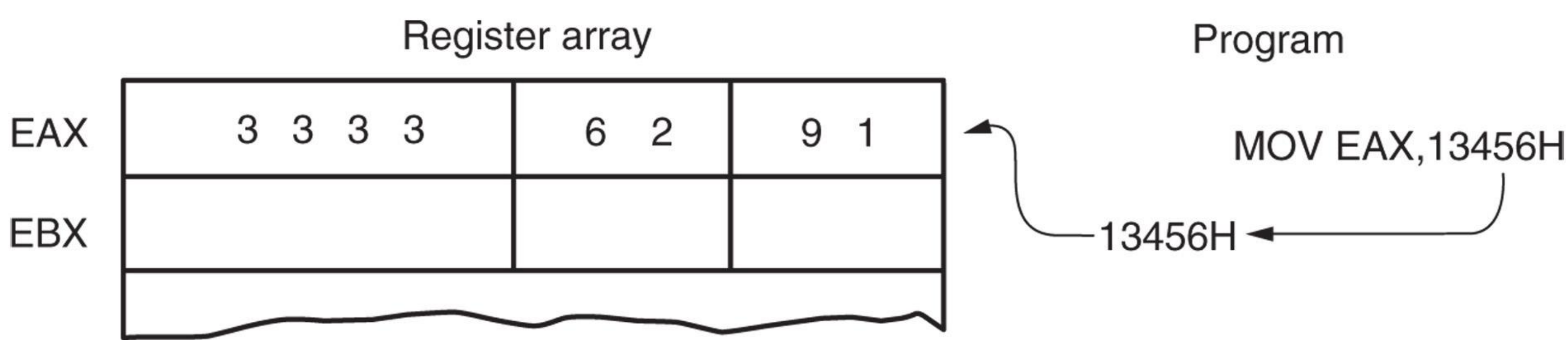
Register Addressing Mode

- The effect of executing the **{MOV BX,CX}** instruction at the point just before the BX register changes. Note that only the rightmost 16 bits of register EBX change.



Immediate Addressing Mode

- The operation of the **{MOV EAX,13456H}** instruction. This instruction copies the immediate data (13456H) into EAX.



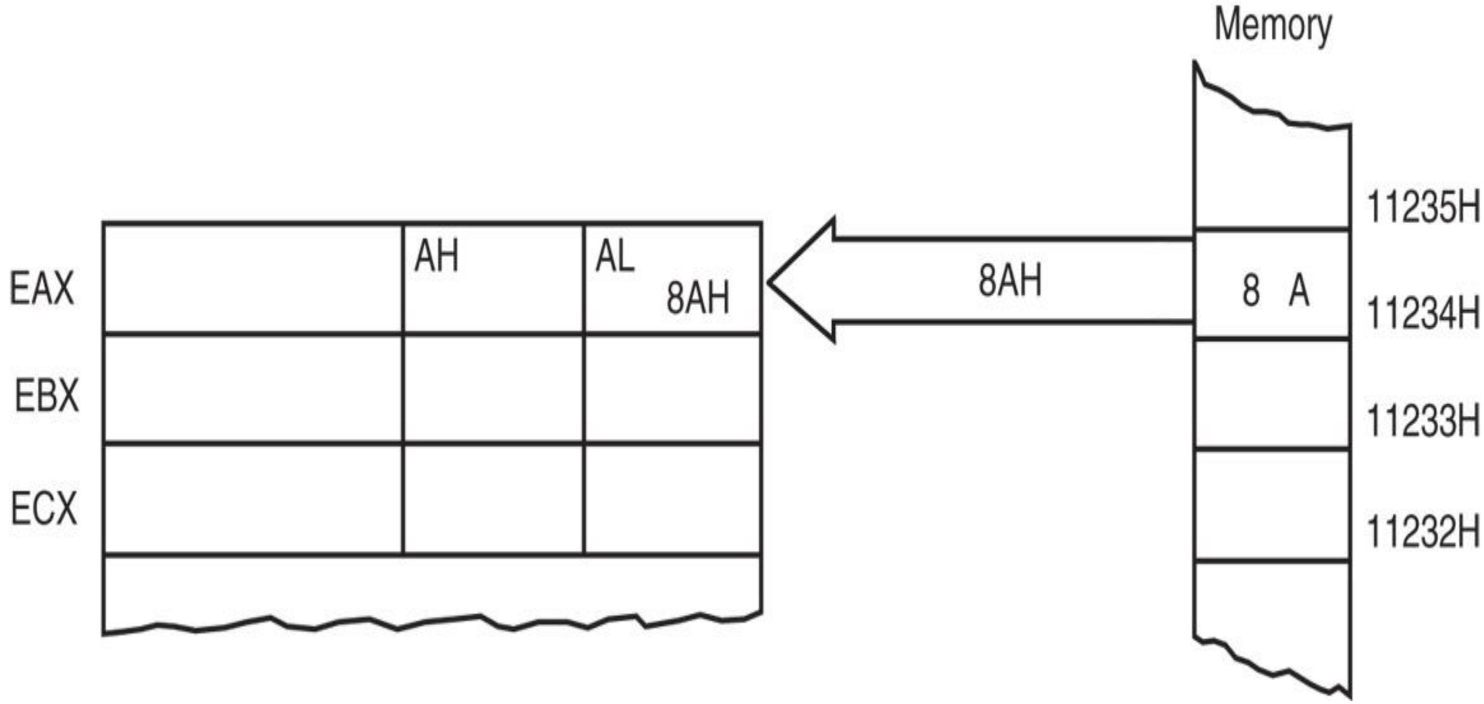
Memory Addressing Modes

- The 8086 processor generalized the memory addressing modes.

- In **8086** you are allowed to **use BX or BP as Base Registers** apart from Segment Registers and **SI or DI as Index Registers.**

1. Direct Data Addressing

- The operation of the **{MOV AL, byte[1234H]}** instruction when DS=1000H .

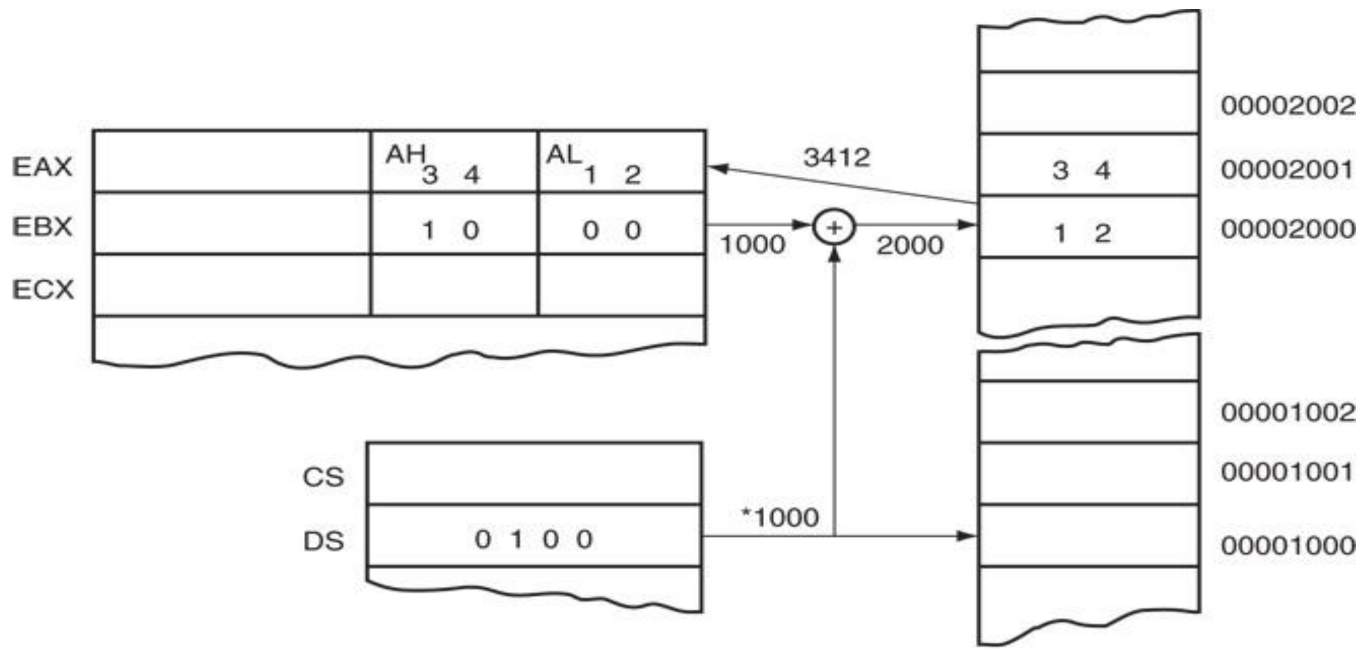

Memory
11235H
11234H
11233H
11232H
8 A
8AH
EAX
EBX
ECX
AH
AL
8AH

2. Register Indirect Addressing

- 8086 Allows data to be addressed at any memory location through an **offset address held in** any of the following registers: **BP, BX, DI, and SI.**

- **Base Address is given by Segment Registers.**

- The operation of the **{MOV AX, word[BX]}** instruction when BX = 1000H and DS = 0100H. Note that this instruction is shown after the contents of memory are transferred to AX.

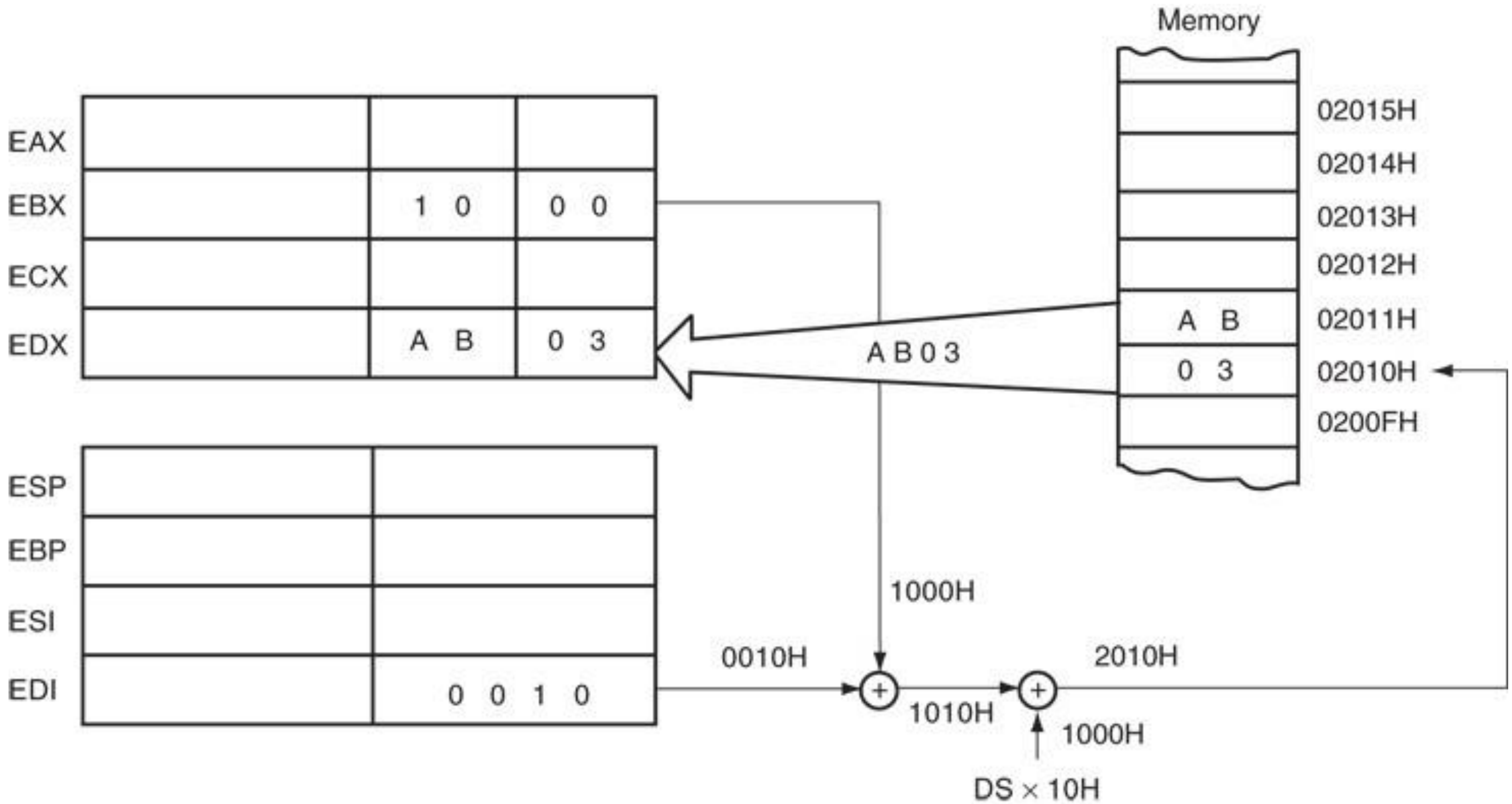


*After DS is appended with a 0.

3. Base+ Index Addressing

- An example showing how the base-plus-index addressing mode functions for the **{MOV DX, word[BX + DI]}** instruction.

  Note: DS=0100H, BX=1000H and DI=0010H.


Memory
EAX
EBX
ECX
EDX
ESP
EBP
ESI
EDI
1 0
0 0
A B
0 3
0 0 1 0
02015H
02014H
02013H
02012H
02011H
02010H
0200FH
A B
0 3
A B 0 3
1000H
0010H
1010H
2010H
1000H
DS × 10H

4. Base+ Index+ Displacement Addressing

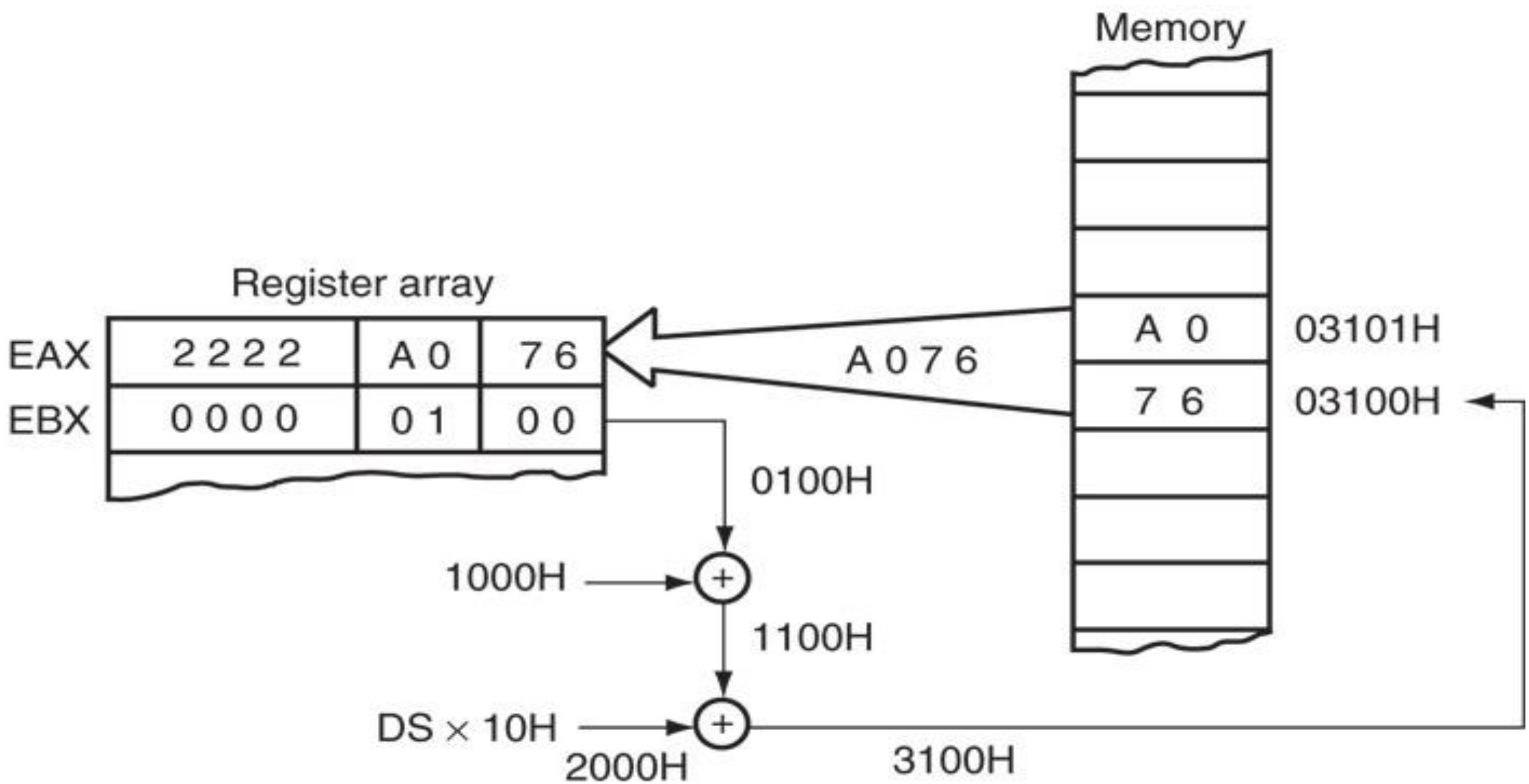
- Similar to base-plus-index addressing and displacement addressing.
  - Data in a segment of memory are addressed by **adding the displacement to the contents of a base or an index register (BP, BX, DI, or SI)**

- Figure shows the operation of the

{MOV AX, word[BX+1000H]} instruction.

when BX=0100H and DS=0200H

- The operation of the **{MOV AX, word[BX+1000H]}** instruction.



Implied/ Implicit Addressing Mode

- Instructions with no oprand belongs to this addressing mode.

Introduction to Nasm

<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
```

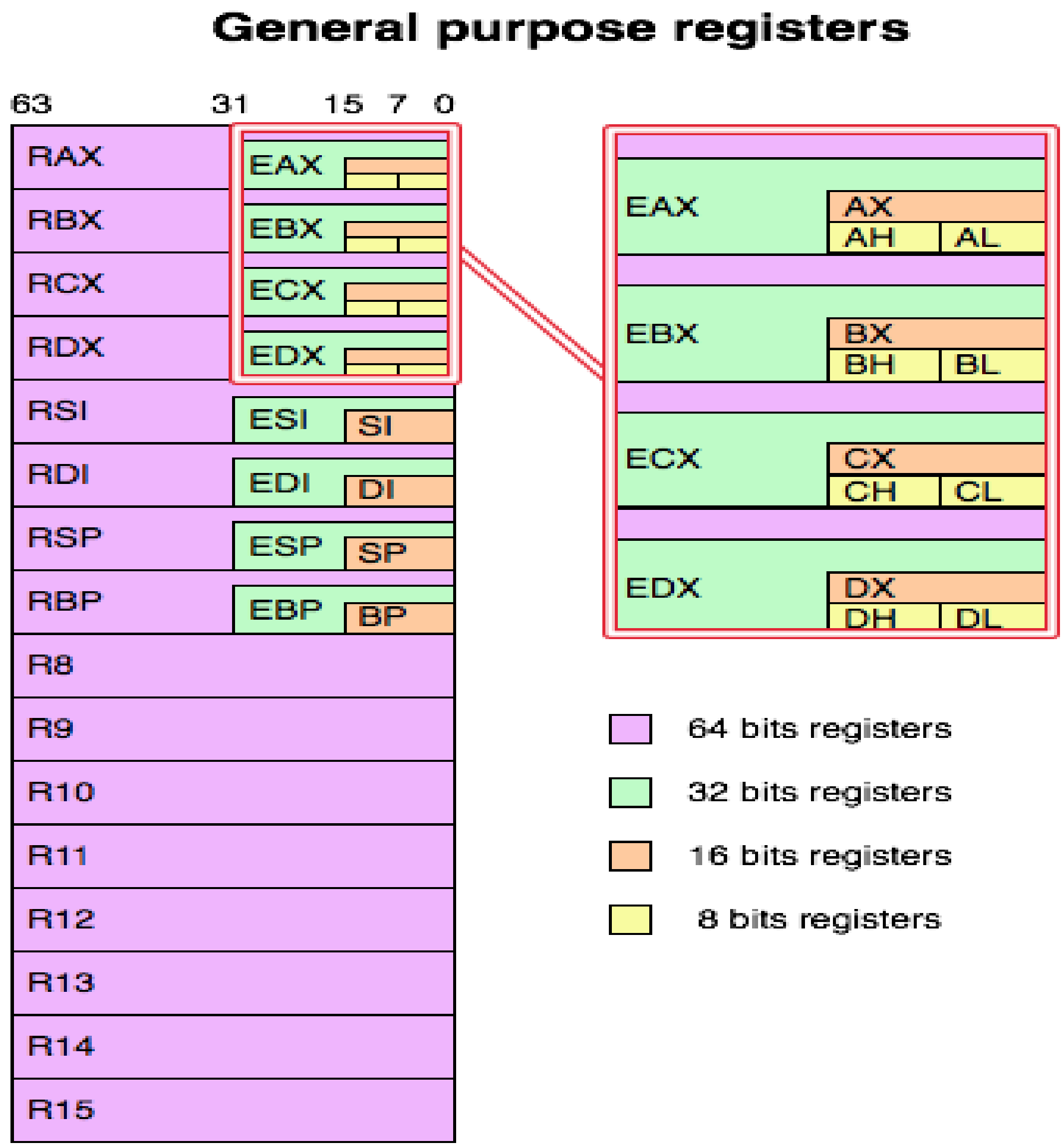
<u>Hello World: 32 Bit</u>

```
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
```

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

Commands:

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

Example:

- **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

<u>Commands:</u>

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

<u>Example:</u>

- **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 12345678A95CCDFE h

5. Memory addressing

section .data

num dq 9828919849096878h

section .bss

name resb 8 ; assembly

Memory addressing:

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

Syntax:

mov eax, *syscall number* *;03-read, 04-write, 01-exit*

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

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

mov edx, *length in bytes* *;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
```

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

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

ASCII TABLE

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