Microcontrollers(23EC501)

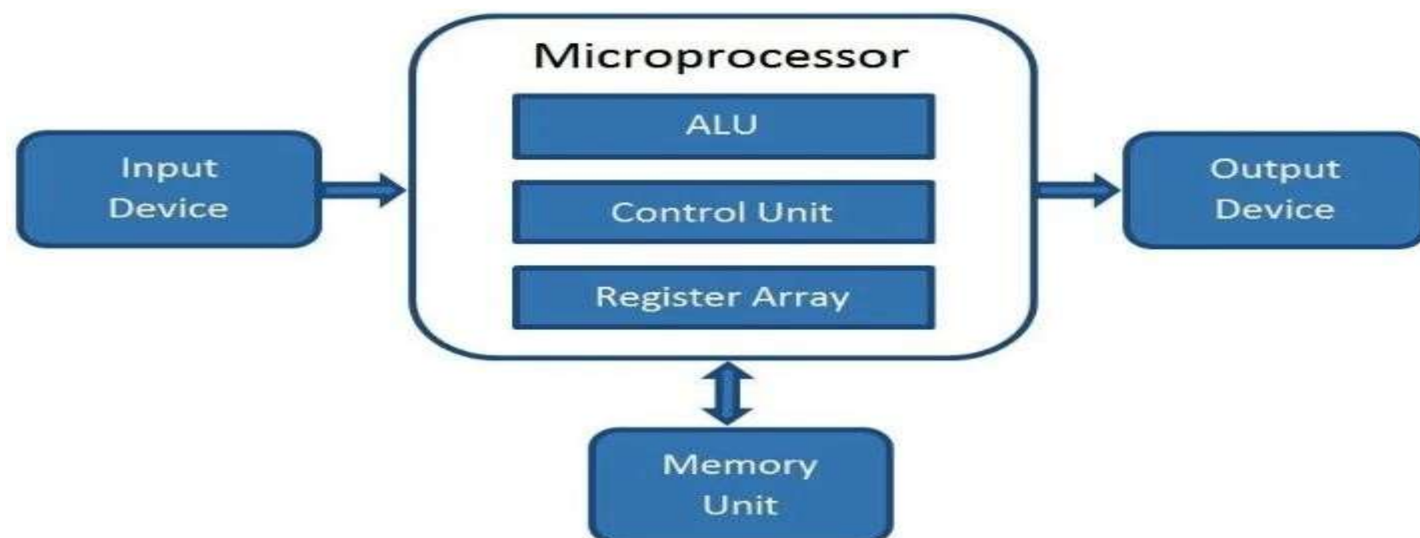
Prepared by

K V SIVA Nagalakshmi

UNIT -1

THE 8086 MICROPROCESSORS

- A **microprocessor** is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.



Evolution of Microprocessor

Processor	Introduced in	Data Bus	Memory address capability	Clock signal
4004	1971	4 bit	1KB	
8008	1972	8 bit, 40 pin	16KB	
8080	1973	8 bit	64KB	
8085	1976	8 bit, 40 pin	64KB	8-6 MHz
8086	1988	16 bit μp, 40 pin	1MB	5-10MHz
80286	1982	16 bit μp, 68 pin	16MB	6-12.5MHz
80386	1985	32 bit, 132 pin	4GB	22-33MHz
80486	1989	32 bit, 168 pin	4GB	26-100MHz
Pentium	1993	32 bit , 168 pin	4GB	100-150MHz

Microprocessor	Micro Controller
Read-Only Memory (ROM), Read-Write Memory, Microprocessor, System Bus, Serial Interface, Timer, I/O Port	Microcontroller, Read-Only Memory, Read-Write Memory, Timer, I/O Port, Serial Interface
Microprocessor is heart of Computer system.	Micro Controller is a heart of embedded system.
It is just a processor. Memory and I/O components have to be connected externally	Micro controller has external processor along with internal memory and i/O components
Since memory and I/O has to be connected externally, the circuit becomes large.	Since memory and I/O are present internally, the circuit is small.
Cannot be used in compact systems and hence inefficient	Can be used in compact systems and hence it is an efficient technique
Cost of the entire system increases	Cost of the entire system is low
Due to external components, the entire power consumption is high. Hence it is not suitable to used with devices running on stored power like batteries.	Since external components are low, total power consumption is less and can be used with devices running on stored power like batteries.
Most of the microprocessors do not have power saving features.	Most of the micro controllers have power saving modes like idle mode and power saving mode. This helps to reduce power consumption even further.
Since memory and I/O components are all external, each instruction will need external operation, hence it is relatively slower.	Since components are internal, most of the operations are internal instruction, hence speed is fast.
Microprocessor have less number of registers, hence more operations are memory based.	Micro controller have more number of registers, hence the programs are easier to write.
Microprocessors are based on von Neumann model/architecture where program and data are stored in same memory module	Micro controllers are based on Harvard architecture where program memory and Data memory are separate
Mainly used in personal computers	Used mainly in washing machine, MP3 players

UNIT 1
THE 8086 MICROPROCESSOR

Introduction to 8086 – Microprocessor architecture – Addressing modes - Instruction set and assembler directives – Assembly language programming – Modular Programming - Linking and Relocation - Stacks - Procedures – Macros – Interrupts and interrupt service routines – Byte and String Manipulation.

UNIT 1

THE 8086 MICROPROCESSOR

FEATURES OF 8086

- The 8086 is a 16 bit processor.
- The 8086 has a 16 bit Data bus.
- The 8086 has a 20 bit Address bus.
- Direct addressing capability 1 M Byte of Memory (2^{20})
- It provides fourteen 16-bit register.
- 24 Operand addressing modes.
- Four general-purpose 16-bit registers: AX, BX, CX, DX
- Available in 40pin Plastic Package and Lead Chip.

8086 MICROPROCESSOR ARCHITECTURE

the 8086 processor are partitioned logically into two processing units

- **Bus Interface Unit (BIU)**

  The BIU fetches instructions, reads data from memory and ports, and writes data to memory and I/O ports.

- **Execution Unit (EU)**

  EU receives program instruction codes and data from the BIU, executes these instructions and stores the results either in the general registers or output them through the BIU. EU has no connections to the system buses.

The BIU contains

- Segment registers
- Instruction pointer
- Instruction queue

The EU contains

- ALU
- General purpose registers
- Index registers
- Pointers
- Flag register

General Purpose Registers

All general registers of the 8086 microprocessor can be used for arithmetic and logic operations.

- **Accumulator register (AX)**

  Accumulator can be used for I/O operations and string manipulation.

- **Base register (BX)**

  BX register usually contains a data pointer used for based, based indexed or register indirect addressing.

- **Count register (CX)**

  Count register can be used as a counter in string manipulation and shift/rotate instructions.

- **Data register (DX)**

  Data register can be used as a port number in I/O operations.

Segment Registers:

Most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data.

- Code segment (CS)

  The CS register is automatically updated during FAR JUMP, FAR CALL and FAR RET instructions.

- Stack segment (SS)

  SS register can be changed directly using POP instruction.

- Data segment (DS)

  DS register can be changed directly using POP and LDS instructions.

- Extra segment (ES)

  ES register can be changed directly using POP and LES instructions.

Pointer Registers

Stack Pointer (SP)

It is a 16-bit register pointing to program stack.

Base Pointer (BP)

It is a 16-bit register pointing to data in the stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Index Registers

Source Index (SI)

It is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.

Destination Index (DI)

It is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

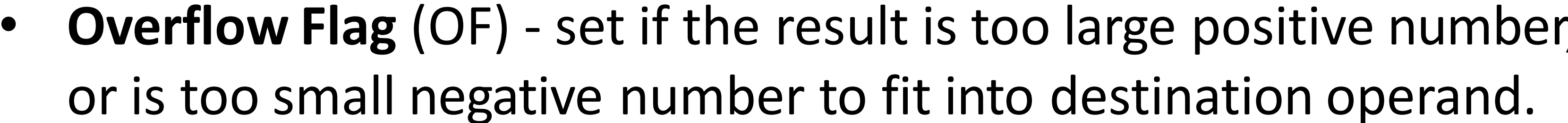
Instruction Pointer (IP)

It is a 16-bit register. The operation is same as the program counter. The IP register is updated by the BIU to point to the address of the next instruction. Programs do not have direct access to the IP, but during execution of a program the IP can be modified or saved and restored from the stack.

Flag register

It is a 16-bit register containing nine 1-bit flags:

- Six status or condition flags (OF, SF, ZF, AF, PF, CF)
- Three control flags (TF, DF, IF)



- **Overflow Flag** (OF) - set if the result is too large positive number, or is too small negative number to fit into destination operand.
- **Sign Flag** (SF) - set if the most significant bit of the result is set.
- **Zero Flag** (ZF) - set if the result is zero.
- **Auxiliary carry Flag** (AF) - set if there was a carry from or borrow to bits 0-3 in the AL register.
- **Parity Flag** (PF) - set if parity (the number of “1” bits) in the low-order byte of the result is even.
- **Carry Flag** (CF) - set if there was a carry from or borrow to the most significant bit during last result calculation.
- **Trap or Single-step Flag** (TF) - if set then single-step interrupt will occur after the next instruction.
- **Direction Flag** (DF) - if set then string manipulation instructions will auto-decrement index registers. If cleared then the index registers will be auto-incremented.**Interrupt-enable Flag** (IF) - setting this bit enables maskable
interrupts.

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

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment

IP	Instruction Pointer

OF	DF	IF	TF	SF	ZF	AF	PF	CF	Flags

Instruction Queue

The instruction queue is a First-In-First-out (FIFO) group of registers where 6 bytes of instruction code is pre-fetched from memory ahead of time. It is being done to speed-up program execution by overlapping instruction fetch and execution. This mechanism is known as **PIPELINING.**

ALU

It is a 16 bit register. It can add, subtract, increment, decrement, complement, shift numbers and performs AND, OR, XOR operations.

Control unit

The control unit in the EU directs the internal operations like RD , WR , M/IO

Instruction Set

- Data moving instructions.
- Arithmetic instructions - add, subtract, increment, decrement, convert byte/word and compare.
- Logic instructions - AND, OR, exclusive OR, shift/rotate and test.
- String manipulation instructions - load, store, move, compare and scan for byte/ word.
- Control transfer instructions - conditional, unconditional, call subroutine and return from subroutine.
- Input/Output instructions.
- Other instructions - setting/clearing flag bits, stack operations, software interrupts, etc.

Addressing modes

- **Implied** - the data value/data address is implicitly associated with the instruction.
- **Register** - references the data in a register or in a register pair.
- **Immediate** - the data is provided in the instruction.
- **Direct** - the instruction operand specifies the memory address where data is located.
- **Register indirect** - instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.
- **Based** - 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to location where data resides.
- **Indexed** - 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.
- **Based Indexed** - the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.
- **Based Indexed with displacement** - 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

Interrupts

Hardware interrupts

Maskable and non-maskable interrupts

Software interrupts

256 software interrupts

ADDRESSING MODES

- An addressing mode is the way the 8086 identifies the operands for the instruction. All instructions that access the data use one or more of the addressing modes.
- The memory address of an operand consists of two components

  1.Starting address of the memory segment

  2.Offset
- When an operand is stored in a memory location, how for the operand's memory location is within a memory segment from the starting address of the segment, is called **Offset** or **Effective Address** (EA).
- The 8086 uses 20 bit memory address. The segment register gives 16 MSBs of the starting address of the memory segment. The BIU generates 20 bit starting address of the memory segment by shifting the content of the segment register left by 4 bits. In other words it puts 4 zeros in 4 LSB positions.
- **Memory Address = Starting address of the memory segment + Offset**

The 8086 has the following addressing modes:

- Register Addressing Mode
- Immediate Addressing Mode
- Direct Addressing Mode
- Register Indirect Addressing Mode
- Base Addressing Mode
- Indexed Addressing Mode
- Based Indexed Addressing Mode
- String Addressing Mode
- I/O Port Addressing Mode
- Relative Addressing Mode
- Implied Addressing Mode

Register Addressing Mode

- Both source and destination operands are registers. The operand sizes must match. MOV *destination, source*
- ***Examples:***
- MOV AL, AH
- MOV AX, BX

Immediate Addressing Mode

- The data operand is supplied as part of the instruction. The immediate operand can only be a source.
- ***Examples:***
- MOV CH, 3A H
- MOV DX, 0C1A5 H

Direct Addressing Mode

- One of the operands is a memory location, given by a constant offset.
- In this mode the 16 bit effective address (EA) is taken directly from the displacement field of the instruction.
- ***Examples:***
- MOV AX,[1234 H]
- MOV DL, [3BD2 H],

Register Indirect Addressing Mode

- One of the operands is a memory location, with the offset given by one of the BP, BX, SI, or DI registers.
- ***Example:***
- MOV [BX], CL
- MOV DL, [BX]

Base Addressing Mode

- In this mode EA is obtained by adding a displacement (signed 8 bit or unsigned 16 bit) value to the contents of BX or BP. The segment registers used are DS and SS.
- ***Example:***
- MOV AX, [BP + 200]

Indexed Addressing Mode

- The operand's offset is the sum of the content of an index register SI or DI and an 8-bit or 16-bit displacement.
- ***Example:***
- MOV AH, [DI]

Based Indexed Addressing Mode

- In this mode, the EA is computed by adding a base register (BX or BP), an index register (SI or DI) and a displacement (unsigned 16 bit or sign extended 8 bit)
- ***Example:***
- MOV AX, [BX + SI + 1234 H]
- MOV CX, [BP][SI] + 4

String Addressing Mode

- The instruction is a string instruction, which uses index registers implicitly to access memory.
- ***Example:***
- MOVS B
- MOVS W

Implied Addressing Mode

- Instructions using this mode have no operands.
- ***Examples:***
- CLC, STC, CMC

INSTRUCTION SET

- Intel 8086 has approximately 117 instructions. These instructions are used to transfer data between registers, register to memory, memory to register or register to I/O ports and other instructions are used for data manipulation.
- But in Intel 8086 operations between memory to memory is not permitted. These instructions are classified in to six-groups as follows.

  1. Data Transfer Instructions
  2. Arithmetic Instructions
  3. Bit Manipulation Instructions
  4. String Instructions
  5. Program Execution Transfer Instructions
  6. Processor Control Instructions

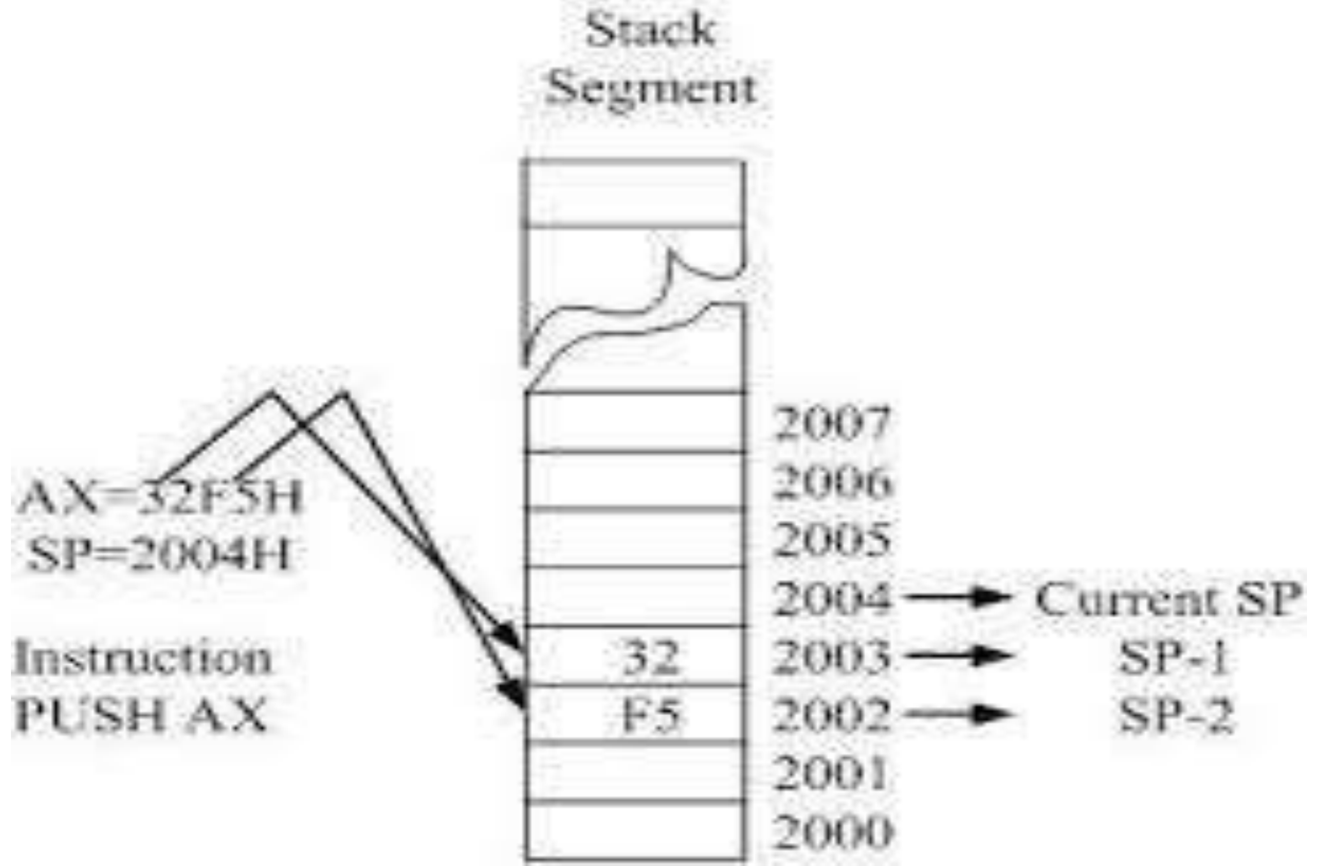
Data Transfer Instructions

1.MOV

- *MOV destination, source*
- This **(Move)** instruction transfers a byte or a word from the source operand to the destination operand.
- (DEST) ← (SRC)
- DEST = Destination
- SRC = Source
- ***Example :***
- MOV AX, BX
- MOV AX, 2150H
- MOV AL, [1135]

2.PUSH

- *PUSH Source*
- This instruction decrements SP (stack pointer) by 2 and then transfers a word from the source operand to the top of the stack now pointed to by stack pointer.
- (SP) ← (SP) – 2
- ((SP)+1 : (SP)) ← (SRC)
- ***Example :***
- PUSH SI
- PUSH BX



Stack
Segment
AX=32F5H
SP=2004H
Instruction
PUSH AX
32
F5
2007
2006
2005
2004
2003
2002
2001
2000
Current SP
SP-1
SP-2

3.POP

- *POP destination*
- This instruction transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by 2, pointing to the new top of the stack.
- (DEST) ← ((SP)+1:(SP))
- (SP) ← (SP) + 2
- ***Example :***
- POP DX
- POP DS

LAHF

- *Load Register AH from Flags*
- This instruction copies Sign flag(S), Zero flag (Z), Auxiliary flag (AC), Parity flag (P) and Carry flag (C) of 8086 into bits 7, 6, 4, 2 and 0 respectively, of register AH

(AH) ←

S	Z	X	AC	X	P	X	C

SAHF

- *Store Register AH into Flags*
- This instruction transfers bits 7, 6, 4, 2 and 0 from register AH into S, Z, AC, P and C flags respectively, thereby replacing the previous values.

S	Z	X	AC	X	P	X	C

← (AH)

XCHG

- *XCHG destination, source*
- This **(Exchange)** instruction switches the contents of the source and destination operands.

(Temp) ← (DEST)

(DEST) ← (SRC)

(SRC) ← (Temp)

Example :

XCHG AX, BX

XCHG BL, AL

XLAT

- *XLAT table*
- This **(Translate)** instruction replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. XLAT is useful for translating characters from one code to another.
- AL ← ((BX) + (AL))
- ***Example :***
- XLAT ASCII_TAB
- XLAT Table_3

LEA

- *LEA destination, source*
- This **(Load Effective Address)** instruction transfers the offset of the source operand (memory) to the destination operand (16-bit general register).
- (REG)← EA
- **Example :**
- LEA BX, [BP] [DI]
- LEA SI, [BX + 02AF H]

LDS

- *LDS destination, source*
- This **(Load pointer using DS)** instruction transfers a 32-bit pointer variable from the source operand (memory operand) to the destination operand and register DS.
- (REG)← (EA)
- (DS)← (EA+2)
- ***Example :***
- LDS SI, [6AC1H]

LES

- *LES destination, source*
- This **(Load pointer using ES)** instruction transfers a 32-bit pointer variable from the source operand (memory operand) to the destination operand and register ES.
- (REG) ← (EA)
- (ES) ← (EA+2)
- ***Example :***
- LES DI, [BX]

IN

- *IN accumulator, port*
- This **(Input)** instruction transfers a byte or a word from an input port to the accumulator (AL or AX).
- (DEST) ← (SRC)
- ***Example :***
- IN AX, DX
- IN AL, 062H

OUT

- *OUT port, accumulator*
- This **(Output)** instruction transfers a byte or a word from the accumulator (AL or AX) to an output port.
- (DEST) ← (SRC)
- ***Example :***
- OUT DX, AL
- OUT 31, AX

Arithmetic Instructions

ADD

- ADD destination, source
- This **(Add)** instruction adds the two operands (byte or word) and stores the result in destination operand.
- (DEST) ← (DEST) + (SRC)
- ***Example :***
- ADD CX, DX
- ADD AX, 1257 H
- ADD BX, [CX]

ADC

- *ADC destination, source*
- This **(Add with carry)** instruction adds the two operands and adds one if carry flag (CF) is set and stores the result in destination operand.
- (DEST) ← (DEST) + (SRC) + 1
- ***Example :***
- ADC AX, BX
- ADC AL, 8
- ADC CX, [BX]

SUB

- *SUB destination, source*
- This **(Subtract)** instruction subtracts the source operand from the destination operand and the result is stored in destination operand.
- (DEST)← (DEST) – (SRC)
- ***Example :***
- SUB AX, 6541 H
- SUB BX, AX
- SUB SI, 5780 H

SBB

- ***SBB destination, source***
- This **(Subtract with Borrow)** instruction subtracts the source from the destination and subtracts 1 if carry flag (CF) is set. The result is stored in destination operand.
- (DEST)← (DEST) – (SRC) –1
- ***Example :***
- SBB BX, CX
- SBB AX, 2

CMP

- ***CMP destination, source***
- This **(Compare)** instruction subtracts the source from the destination, but does not store the result.
- (DEST) – (SRC)
- ***Example :***
- CMP AX, 18
- CMP BX, CX

INC

- *INC destination*
- This **(Increment)** instruction adds 1 to the destination operand (byte or word).
- (DEST) ← (DEST) + 1
- ***Example :***
- INC BL
- INC CX

DEC

- *DEC destination*
- This **(Decrement)** instruction subtracts 1 from the destination operand. (DEST) ← (DEST) –1
- ***Example :***
- DEC BL
- DEC AX

NEG

- *NEG destination*
- This **(Negate)** instruction subtracts the destination operand from 0 and stores the result in destination. This forms the 2's complement of the number.
- (DEST) ← 0 – (DEST)
- ***Example :***
- NEG AX
- NEG CL

DAA

- This **(Decimal Adjust for Addition)** instruction converts the binary result of an ADD or ADC instruction in AL to packed BCD format.

DAS

- This **(Decimal Adjust for Subtraction)** instruction converts the binary result of a SUB or SBB instruction in AL to packed BCD format.

AAA

- This **(ASCII Adjust for Addition)** instruction adjusts the binary result of ADD or ADC instruction.
- If bits 0-3 of AL contain a value greater than 9, or if the auxiliary carry flag (AF) is set, the CPU adds 06 to AL and adds 1 to AH. The bits 4-7 of AL are set to zero.
- $(AL) \leftarrow (AL) + 6$
- $(AH) \leftarrow (AH) + 1$
- $(AF) \leftarrow 1$

AAS

- This **(ASCII Adjust for Subtraction)** instruction adjusts the binary result of a SUB or SBB instruction.
- If D_3–D_0 of AL > 9,
- (AL) ← (AL) – 6
- (AH) ← (AH) – 1
- (AF) ← 1

MUL

- *MUL source*
- This **(Multiply)** instruction multiply AL or AX register by register or memory location contents. Both operands are unsigned numbers. If the source is a byte (8 bit), then it is multiplied by register AL and the result is stored in AH and AL.
- If the source operand is a word (16 bit), then it is multiplied by register AX and the result is stored in AX and DX registers.
- If 8 bit data, (AX) ← (AL) x (SRC)
  If 16 bit data, (AX), (DX) ← (AX) x (SRC)
  Example :
- MUL 25
- MUL CX

- **IMUL**
- *IMUL Source*
- This **(Integer Multiply)** instruction performs a signed multiplication of the source operand and the accumulator.
- If 8 bit data, (AX)← (AL) x (SRC)
- If 16 bit data, (AX), (DX) ←(AX) x (SRC)
- ***Example :***
- IMUL 250
- IMUL BL

AAM

- This **(ASCII Adjust for Multiplication)** instruction adjusts the binary result of a MUL instruction. AL is divided by 10(0AH) and quotient is stored in AH. The remainder is stored in AL.
- (AH)← (AL/0AH)
- (AL) ←Remainder

DIV

- *DIV Source*
- This **(Division)** instruction performs an unsigned division of the accumulator by the source operand. It allows a 16 bit unsigned number to be divided by an 8 bit unsigned number, or a 32 bit unsigned number to be divided by a 16 bit unsigned number.
- For 8 bit data, AX / source
  (AL) ← Quotient
  (AH) ← Remainder
- For 16 bit data, AX, DX / Source
  (AX) ← Quotient
  (DX) ← Remainder
- ***Example :***
- DIV CX
- DIV 321

IDIV

- *IDIV source*
- This **(Integer Division)** instruction performs a signed division of the accumulator by the source operand.
- For 8 bit data, AX / Source

  (AL)← Quotient

  (AH)← Remainder
- For 16 bit data, AX, DX / Source

  (AX)← Quotient

  (DX)← Remainder
- ***Example :***
- IDIV CL
- IDIV AX

AAD

- This **(ASCII Adjust for Division)** instruction adjusts the unpacked BCD dividend in AX before a division operation. AH is multiplied by 10(0AH) and added to AL. AH is set to zero.
- (AL) ← (AH x 0AH) + (AL)
- (AH)← 0

CBW

- This **(Convert Byte to Word)** instruction converts a byte to a word. It extends the sign of the byte in register AL through register AH. This instruction can be used for 16 bit IMUL or IDIV instruction.
- IF AL < 80 H, then AH = 00 H
- IF AL > 80 H, then AH = FFH

CWD :

- This **(Convert Word to Double word)** instruction converts a word to a double word.
- It extends the sign of the word in register AX through register DX.
- If AX < 8000 H, then DX = 0000 H
- If AX > 8000 H, then DX = FFFFH

Bit Manipulation Instructions

(i)Logical Instructions: AND, OR, XOR, NOT, TEST

(ii)Shift Instructions: SHL, SAL, SHR, SAR

(iii)Rotate Instructions: ROL, ROR, RCL, RCR

AND

- *AND destination, source*
- This **(AND)** instruction performs the logical "AND" of the source operand with the destination operand and the result is stored in destination.
- (DEST) ¬ (DEST) "AND" (SRC)
- ***Example :***
- AND BL, CL
- AND AL, 0011 1100 B

OR

- *OR destination, source*
- This **(OR)** instruction performs the logical "OR" of the source operand with the destination operand and the result is stored in destination.
- (DEST) ¬ (DEST) "OR" (SRC)
- ***Example :***
- OR AX, BX
- OR AL, 0000 1111B

2 = OR : LOGIC OR

OR instruction source operand immediate , register or memory location to the destination operand

OR AX , 0098H content of AX if 3F0FH

OR AX , BX

0011	1111	0000	1111	= 3F0F H [AX]
				OR
0000	0000	1001	1000	= 0098 H
0011	1111	1001	1111	= 3F9F H [AX]

XOR

- *XOR destination, source*
- This **(Exclusive OR)** instruction performs the logical “XOR” of the two operands and the result is stored in destination operand.
- (DEST) ¬ (DEST) “XOR” (SRC)
- ***Example :***
- XOR BX, AX
- XOR AL, 1111 1111B

NOT

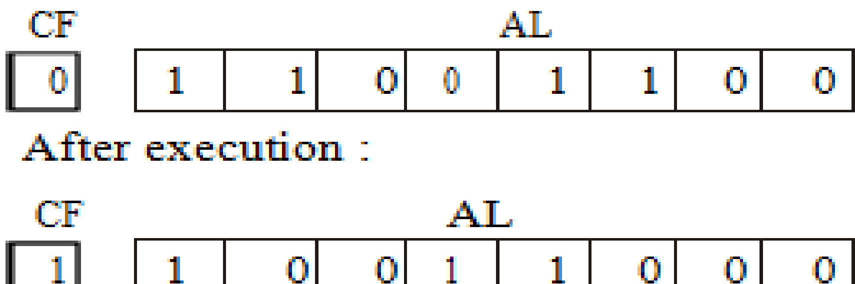
- *NOT destination*
- This **(NOT)** instruction inverts the bits (forms the 1’s complement) of the byte or word.
- (DEST) ¬ 1’s complement of (DEST)
- ***Example :***
- NOT AX

TEST

- *TEST destination, source*
- This **(TEST)** instruction performs the logical “AND” of the two operands and updates the flags but does not store the result.
- (DEST) “AND” (SRC)
- ***Example :***
- TEST AL, 15 H
- TEST SI, DI

SHL

- *SHL destination, count*
- This **(Shift Logical Left)** instruction performs the shift operation. The number of bits to be shifted is represented by a variable count, either 1 or the number contained in the CL register.
- ***Example***
- SHL AL, 1
- Before execution :

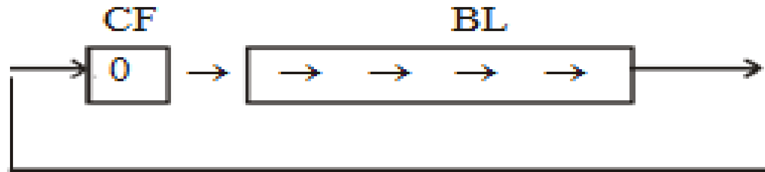


SAL

- *SAL destination, count*
- SAL **(Shift Arithmetric Left)** and SHL (Shift Logical Left) instructions perform the same operation and are physically the same instruction.
- ***Example***
- SAL AL, CL
- SAL AL, 1

SHR

- *SHR destination, count*
- This **(Shift Logical Right)** instruction shifts the bits in the destination operand to the right by the number of bits specified by the count operend, either 1 or the number contained in the CL register.
- ***Example***
- SHR BL, 1
- SHR BL, CL

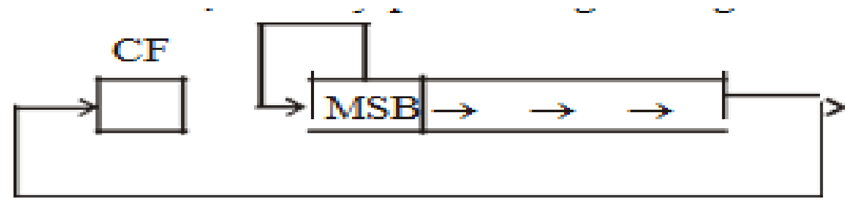


The SHR instruction may be used to divide a number by 2. For example, we can divide 32 by 2,

MOV BL, 32	;	0010	0000	(32)
SHR BL, 1	;	0001	0000	(16)
SHR BL, 1	;	0000	1000	(8)
SHR BL, 1	;	0000	0100	(4)
SHR BL, 1	;	0000	0010	(2)

SAR

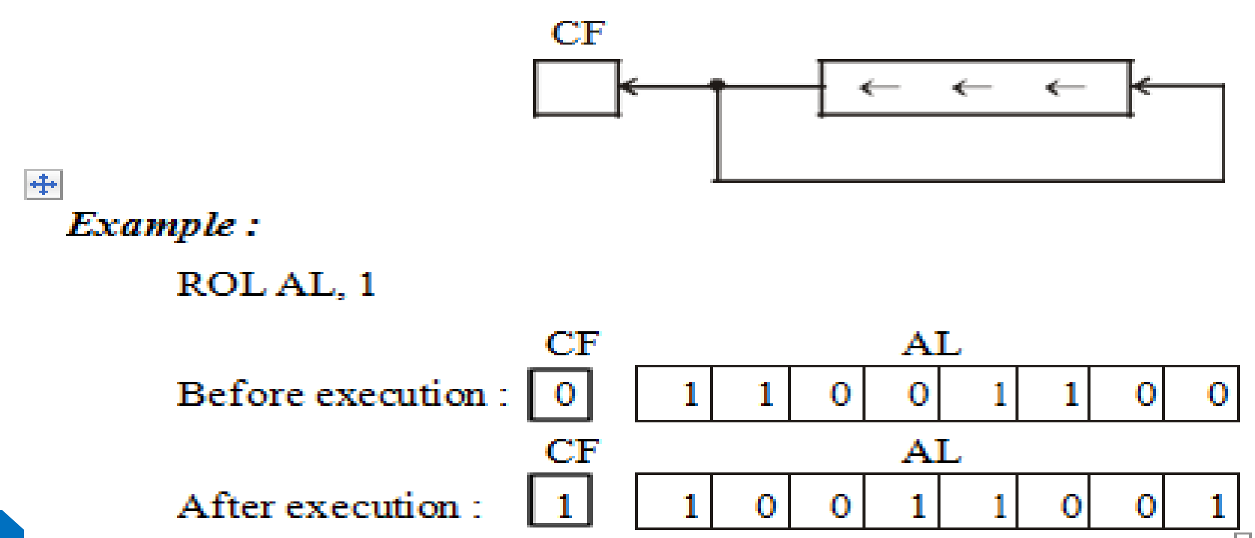
- *SAR destination, count*
- This **(Shift Arithmetic Right)** instruction shifts the bits in the destination operand to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bits are shifted in on the left, thereby preserving the sign of the original value.



ROL

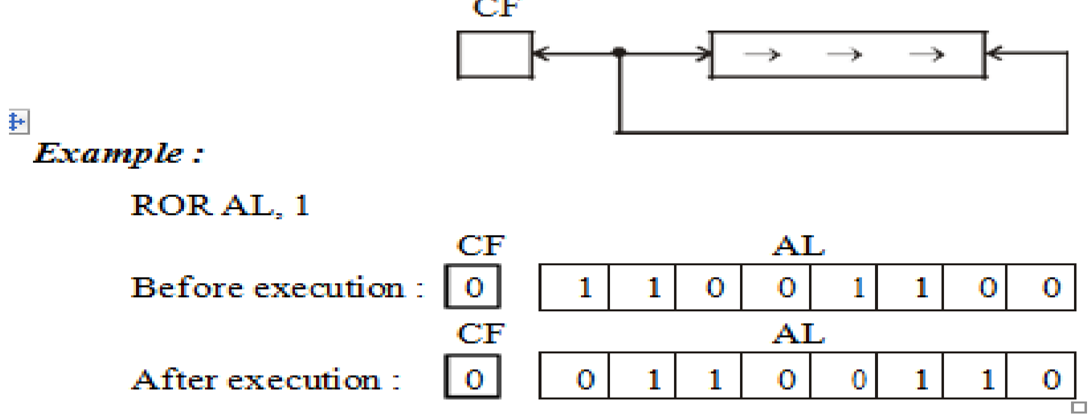
ROL destination, count

This **(Rotate Left)** instruction rotates the bits in the byte/word destination operand to the left by the number of bits specified in the count operand.



ROR

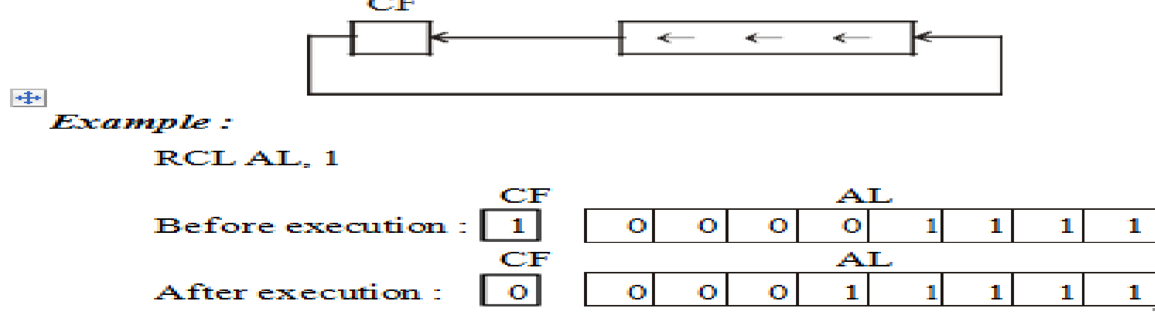
- *ROR destination, count*
- This **(Rotate Right)** instruction rotates the bits in the byte/word destination operand to the right by the number of bits specified in the count operand.



RCL

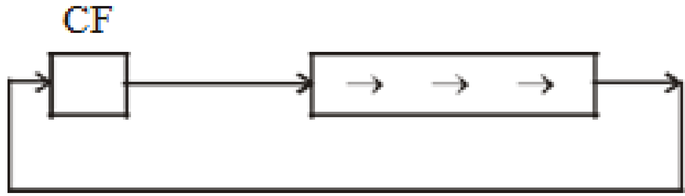
RCL destination, count

This **(Rotate through Carry Left)** instruction rotates the contents left through carry by the specified number of bits in count operand.



RCR

- *RCR destination, count*
- This **(Rotate through Carry Right)** instruction rotates the contents right through carry by the specified number of bits in the count operand.



Example :

RCR AL, 1

	CF	AL							
Before execution :	1	1	1	0	0	0	0	1	0
After execution :	0	1	1	1	0	0	0	0	1

STRING INSTRUCTIONS

REP

- *REP MOVS destination, Source*
- This **(Repeat)** instruction converts any string primitive instruction into a re-executing loop. It specifies a termination condition which causes the string primitive instruction to continue executing until the termination condition is met.
- ***Example :***
- REP MOVS CL, AL
- The other Repeat instructions are :
- REPE - Repeat while Equal
- REPZ - Repeat while zero
- REPNE - Repeat while Not Equal
- REPNZ - Repeat while Not Zero
- The above instructions are used with the CMPS and SCAS instructions.

MOVS

- *MOVS destination - string, source-string*
- This **(Move String)** instruction transfers a byte/word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element.
- (DEST) ← (SRC)
- ***Example :***
- MOVS Buffer 1, Buffer 2

CMPS

- *CMPS destination-string, source-string*
- This **(Compare String)** instruction subtracts the destination byte/word (addressed by DI) from the source byte/word (addressed by SI). It affects the flags but does not affect the operands.
- ***Example :***
- CMPS Buffer 1, Buffer 2

SCAS

- *SCAS destination-string*
- This **(Scan String)** instruction subtracts the destination string element (addressed by DI) from the contents of AL or AX and updates the flags.
- **Example :**
- SCAS Buffer

LODS

- *LODS source-string*
- This **(Load String)** instruction transfers the byte/word string element addressed by SI to register AL or AX and updates SI to point to the next element in the string.
- (DEST) ← (SRC)
- ***Example :***
- LODSB name
- LODSW name

STOS

- *STOS destination - string*
- This **(Store String)** instruction transfers a byte/word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string.
- (DEST) ← (SRC)
- ***Example :***
- STOS display

Program Transfer Instructions

- (i)Unconditional instructions: CALL, RET, JMP
- (ii)Conditional instructions: JC, JZ, JA.....
- (iii)Iteration control instructions :LOOP, JCXZ
- (iv)Interrupt instructions: INT, INTO, IRET

CALL

- *CALL procedure - name*
- This **(CALL)** instruction is used to transfer execution to a subprogram or procedure. RET (return) instruction is used to go back to the main program. There are two basic types of CALL : NEAR and FAR
- ***Example :***
- CALL NEAR
- CALL AX

NRCM
your roots to success...

RET

- This **(Return)** instruction will return execution from a procedure to the next instruction after the CALL instruction in the main program.
- ***Example :***
- RET
- RET 6

JMP

- *JMP target*
- This **(Jump)** instruction unconditionally transfers control to the target location. The target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).
- ***Example :***
- JMP BX

Conditional JMP

Instruction	Operation
JC	Jump if carry
JNC	Jump if no carry
JZ	Jump if Zero
JNZ	Jump if not zero
JS	Jump if sign or negative
JNS	Jump if positive
JP/JPE	Jump if parity/parity even
JNP/JPO	Jump if not parity/odd parity
JO	Jump if overflow
JNO	Jump if no overflow
JA/JNBE	Jump if above/not below or equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above or equal
JBE/JNA	Jump if below or equal / not above
JG/JNLE	Jump if greater/not less than nor equal
JGE/JNL	Jump if greater or equal/not less than
JL/JNGE	Jump if less/neither greater nor equal
JLE/JNG	Jump if less than or equal / not greater

LOOP

- *LOOP label*
- This **(Loop if CX not zero)** instruction decrements CX by 1 and transfers control to the target operand if CX is not zero. Otherwise the instruction following LOOP is executed.
- If CX≠0, CX = CX–1
- IP = IP+displacement
- If CX=0, then the next sequential instruction is executed.
- ***Example :***
- LOOP again

Processor Control Instructions

HLT

- This **(Halt)** instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state.

WAIT

- This **(Wait)** instruction causes the 8086 to enter the wait state while its test line is not active.

ESC

- This **(Escape)** instruction provides a mechanism by which other coprocessors may receive their instructions from the 8086 instruction stream and make use of the 8086 addressing modes. The 8086 does a no operation (NOP) for the ESC instruction other than to access a memory operand and place it on the bus.

NOP

- This **(No operation)** instruction causes the CPU to do nothing. NOP does not affect any flags.

Flag operations

Instruction	Operation
CLC	Clear the carry flag (CF)
CMC	Complement the carry flag (CF)
STC	Set the carry flag (CF)
CLD	Clear the direction flag (DF)
STD	Set the direction flag (DF)
CLI	Clear the interrutp flag (IF)
STI	Set the interrupt flag (IF)

ASSEMBLER DIRECTIVES

- An assembler is a program which translates an assembly language program into machine language program.
- An assembler directive is a statement to give direction to the assembler to perform the task of assembly process.
- The assembler directives control organization of the program and provide necessary information to the assembler to understand assembly language programs to generate machine codes.
- An assembler supports directives to define data, to organize segments, to control procedures, to define macros etc.
- An assembly language program consists of two types of statements: Instructions and Directives.

Some assembler directives are,

- Borland Turbo Assembler (TASM)
- IBM Macro Assembler (MASM)
- Intel 8086 Macro Assembler (ASM)
- Microsoft Macro Assembler

The general assembler directives are

ASSUME	EXTRN
DB	GROUP
DW	INCLUDE
DD	LABEL
DQ	MACRO
DT	ORG
END	PTR
ENDP	PROC
ENDM	PUBLIC
ENDS	RECORD
EQU	SEGMENT
EVEN	STRUC

ASSUME

- The ASSUME directive enables error-checking for register values.
- It is used to inform the assembler the names of the logical segments, which are to be assigned to the different segments used in an assembly language program
- Format:
- **ASSUME** *segregister*:*name* [[, *segregister*:*name*]]...
- **ASSUME** *dataregister*:*type* [[, *dataregister*:*type*]]...
- **ASSUME** *register*:**ERROR** [[, *register*:**ERROR**]]...
- **ASSUME** [[*register*:]] **NOTHING** [[, *register*:**NOTHING**]]...

DB (Define Byte)

- It can be used to define data like **BYTE**.
- Format:
- *Name of the Variable DB Initial values*
- Example:
- WEIGHTS DB 18, 68, 45

DW (Define Word)

- It can be used to define data like **WORD** (2 bytes).
- Format:
- *Name of the Variable DW Initial values*
- Example:
- SUM DW 4589

DD (Define Double Word)

- It can be used to define data like **DWORD** (4 bytes).
- <u>Format</u>:
- *Name of the Variable DD Initial values*
- <u>Example</u>:
- NUMBER DD 12345678

DQ (Define Quad Word)

- It can be used to define data like **QWORD** (8 bytes).
- <u>Format</u>:
- *Name of the Variable DQ Initial values*
- <u>Example</u>:
- TABLE DQ 1234567812345678

DT (Define Ten Bytes)

- It can be used to define data like **TBYTE** (10 bytes).
- Format:
- *Name of the Variable DT Initial values*
- Example:
- AMOUNT DT 12345678123456781234

END (End of program)

- It marks the end of a program module and, optionally, sets the program entry point to *address*.
- Format:
- **END** *[[address]]*
- Example:
- END label

ENDP (End Procedure)

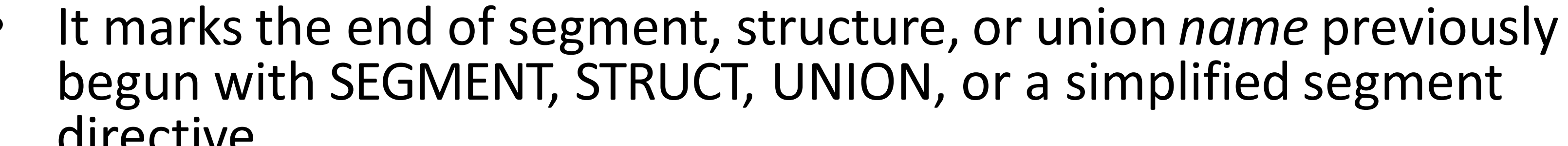
- It marks the end of procedure.
- *name* previously begun with PROC.
- Format:
- *name***ENDP**
- Example:

```
CONTROL PROC FAR
  .
  .
  .
 CONTROL ENDP
```

- **ENDM (End Macro)**
- It terminates a macro or repeat block.
- Format:
- **ENDM**
- Example:

```
 CODE          MACRO
  .
  .
  .
  ENDM
```

- **ENDS (End of Segment)**



- It marks the end of segment, structure, or union *name* previously begun with SEGMENT, STRUCT, UNION, or a simplified segment directive.
- Format:
- *name* **ENDS**
- Example:

```
CODE SEGMENT
 .
 .
 .
CODEENDS
```

EQU (Equate)

- It assigns numeric value of *expression or text* to *name*. The *name* cannot be redefined later.
- Format:
- *name* **EQU** *expression*
- *name* **EQU** **<***text***>**
- Example:
- CLEAR_CARRY EQU CLC

- **EVEN (Align on Even memory Address)**
- Format:
- EVEN
- Example:
- SALES DB 9
- EVEN
- DATA_ARRAY DW 100 DUP (?)

INCLUDE

- This directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.
- Format:
- **INCLUDE** *filename*
- Example:
- INCLUDE C: \ MICRO \ ASSEM.LEV
- The above directive informs assembler to include all statements mentioned in the file, ASSEM.LEV from the directory C: \ MICRO.

MACRO

- A sequence of instructions to which a name is assigned is called a macro. The name of a macro is used in assembly language programming. Macros and subroutines are similar. Macros are used for short sequences of instructions, where as subroutines for longer ones. Macros execute faster than subroutines. A subroutine requires CALL and RET instructions whereas macros do not.
- Format:
- *name* **MACRO** [optional arguments]
- *statements* **ENDM**

ASSEMBLY LANGUAGE PROGRAMMING

Program

A computer can only do what the programmer asks to do. To perform a particular task the programmer prepares a sequence of instructions, called a program.

Programming languages

- Microcomputer programming languages can typically be divided into three main types:
    1. Machine language
    2. Assembly language
    3. High-level language

Machine language

- A program written in the form of 0s and 1s is called a machine language program. In the machine language program there is a specific binary code for each instruction.
- A microprocessor has a unique set of machine language instructions defined by its manufacturer.
- For example, the Intel 8085 uses the code $1000\ 1110_2$ for its addition instruction while the Motorola 6800 uses the code $1011\ 1001_2$.

The machine language program has the following demerits:

- It is very difficult to understand or debug a program.
- Program writing is difficult.
- Programs are long.
- More errors occur in writing the program.
- Since each bit has to be entered individually the entry of a program is very slow.

Assembly language

- Assembly language programming is writing machine instructions in mnemonic form, using an assembler to convert these mnemonics into actual processor instructions and associated data.

The advantages of assembly language programming

1.The computation time is less. 2.It is faster to produce result.

The disadvantages of assembly language programming

- many instructions are required to achieve small tasks
- source programs tend to be large and difficult to follow

High-level language

- High level language programs composed of English-language-type statements rectify all deficiencies of machine and assembly language programming. The high level languages are FORTRON, COBAL, BASIC, C, C++, Pascal, Visual Basic etc.

The high level language program has the following demerits:

- One has to learn the special rules for writing programs in a particular high level language.
- Low speed.
- A **compiler** has to be provided to convert a high level language program into a machine language program. The compiler is costly.

Assembly language program

- Assembly language statements are written one per line.
- A machine code program thus consists of a sequence of assembly language statements, where each statement contains a mnemonic.
- Each line of an assembly language program is split into four fields, as below:

  1. Label field
  2. Mnemonic or Opcode field
  3. Operand field
  4. Comment field

As an example, a typical program for block transfer of data written in 8086 assembly language is given here.

LABEL	OPCODE	OPERAND	COMMENTS
	CLD		Clear direction flag DF
	MOV	SI, 0200	Source address in SI
	MOV	DI, 0302	Destination address in DI
	MOV	CX, [SI]	Count in CX
	INC	SI	Increment SI
	INC	SI	Increment SI
BACK:	MOV	SB	Move byte
	LOOP	BACK	Jump to BACK until CX = 0
	INT		Interrupt program

LABEL

- The label field is optional. A label is an identifier.
- A label can be used to refer to a memory location the value of a piece of data the address of a program, sub-routine, code portion etc.

```
START: LDAA #24H
JMP START
```

- Here, the label START is equal to the address of the instruction LDAA #24H. The label is used in the program as a reference. This would result in the processor jumping to the location (address) associated with the label START, thus executing the instruction LDAA #24H immediately after the JMP instruction.

OPCODE

- Each instruction consists of an opcode (Mnemonic) and possible one or more operands. In the above instruction

  JMP START

- The opcode is JMP and the operand is the address of the label START.

Mnemonics are used because they

- are more meaningful than hex or binary values
- reduce the chances of making an error
- are easier to remember than bit values

OPERAND

- The operand field consists of additional information or data that the opcode requires. In certain types of addressing modes, the operand is used to specify
- constants or labels
- immediate data
- data contained in another accumulator or register
- an address

Examples of operands are

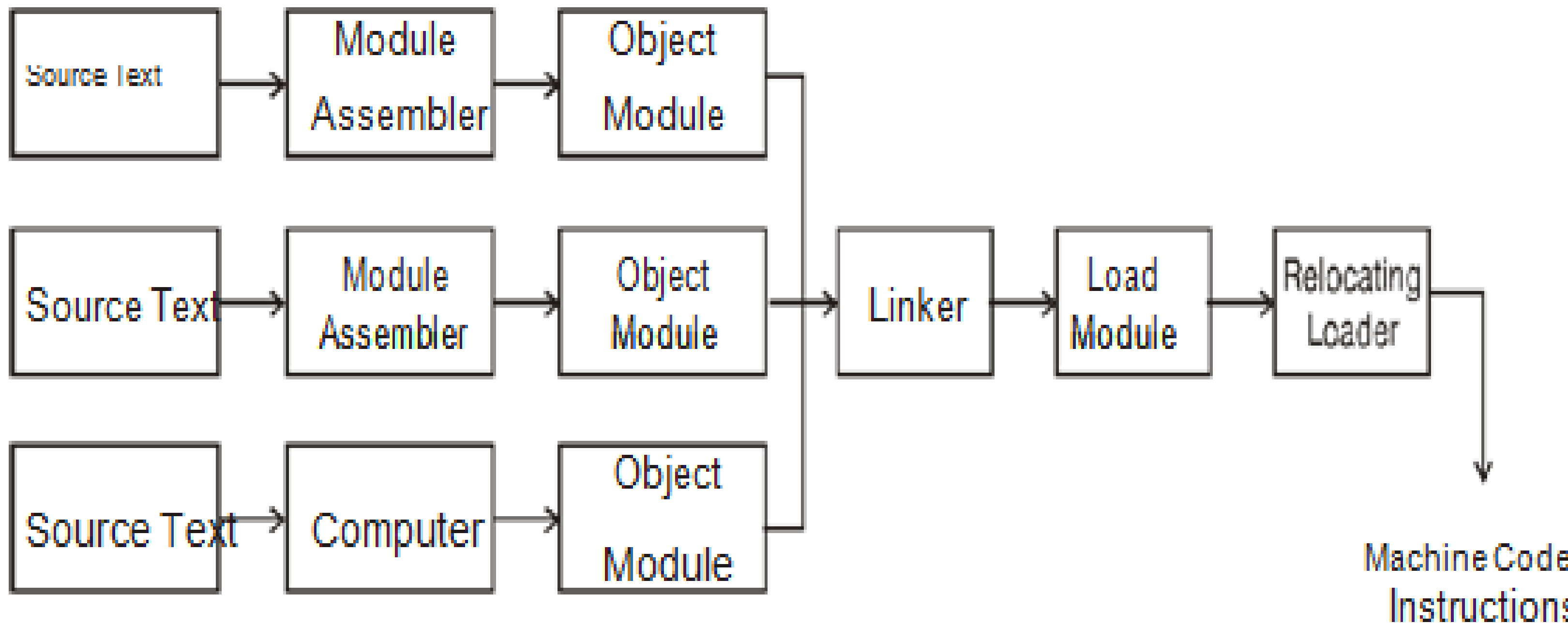
- JNZ STEP1
- MOV AX, 5000 H
- MOV AX, BX
- MOV AX, [3000 H]

COMMENTS

- The comment field is optional, and is used by the programmer to explain how the coded program works. Comments are preceded by a semi-colon. The assembler, when generating instructions from the source file, ignores all comments.

Assembly Language Program - Development Tools

- Editor
- Assembler
- Linker
- Locator
- Loader
- Debugger
- Emulator



Editor:

- An editor is a program which allows creating a file containing the assembly language statements for the program.

Assembler:

- An assembler is a program which translates an assembly language program into machine language program.

Linker:

- A linker is a program which links smaller programs together to form a large program. It is used to join several object files into one large object file. It also links the subroutines with the main program.

Locator:

- A locator is a program which assigns specific memory addresses for the machine codes of the program, which is to be loaded into the memory.

Loader:

- A loader is a program which loads object code into system memory. It can accept programs in absolute or relocatable format.

Debugger:

- A debugger is a program which allows user to test and debug programs.

Emulator:

- An emulator is a mixture of software and hardware. It is usually used to test and debug the software and hardware of an external system.

1. Addition of two 16-Bit Data

Label	Mnemonics	Comments
	MOV AX, DATA1	Load the first data in AX register
	MOV CL, 00H	Clear the CL register for carry
	ADD AX, DATA2	Add 2nd data to AX, sum will be in AX
	MOV 2000H, AX	Store sum in memory location 1
	JNC STEP	Check the status of carry flag
	INC CL	If carry is set; increment CL by one
STEP :	MOV 2002H, CL	Store carry in memory location 2
	HLT	Halt

3. Multiplication of Two 16-Bit Data

Label	Mnemonics	Comments
	MOV AX, [2000]	Move the first data to AX register from memory location 2000 H
	MUL [2002]	Multiply the data in AX with the data in memory location 2002 H
	MOV [2100], DX	Save the MSW (high order) of the result in DX register
	MOV [2102], AX	Save the LSW (Lower Order) or the result in AX register
	HLT	Halt

MODULAR PROGRAMMING

- Modular programming is subdividing the complex program into separate subprograms such as functions and subroutines.
- Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code so that the code can be reused by other applications.
- For example, if a program needs initial and boundary conditions, use subroutines to set them.
- Then if someone else wants to compute a different solution using the program, only these subroutines need to be changed. This is very easier than having to read through a program line by line, trying to figure out what each line is supposed to do and whether it needs to be changed.

- Subprograms make the actual program shorter, hence easier to read and understand. Further, the arguments show exactly what information a subprogram is using. That makes it easier to figure out whether it needs to be changed when modifying the program.

ALPs are developed by essentially the same procedure as high-level language programs by,

- Exactly stating what the program is to do.
- Splitting the overall problem into tasks.
- Defining exactly what each task must do and how it is to communicate with the other tasks.
- Putting the tasks into assembler language modules and connecting the modules together to form the program.
- Debugging and testing the program.
- Documenting the program.

The benefits of using modular programming are,

- Modular programming allows many programmers to collaborate on the same application.
- Same code can be used in many applications.
- Code is short, simple and easy to understand.
- Code is stored across multiple files.
- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Errors can easily be identified, as they are localized to a subroutine or function.

LINKING AND RELLOCATION

Operating system
Commands
Resident monitor
I/O drivers
Source module
Assembler
Object module
Linker
Load module
Loader
Executing program
Listing
Memory map
Other object modules
Library
Make corrections

The process combines the following.

- Find the object modules to be linked.
- Construct the load module by assigning the positions of all of all the segments in all of the object modules being linked.
- Fill in all offset that could not be determined by the assembler.
- Fill in all segment address.
- Load the program for execution.

Segment combination

- In addition to the linker commands, the assembler provides a means of regulating the way segments in different object modules are organized by the linker. Segments with same name are joined together by using the modifiers attached to the SEGMENT directives. SEGMENT directive may have the form:
- **Segment name SEGMENT Combination-type**

PROCEDURES & MACROS

- A single instruction that expands automatically into a set of instructions to perform a particular task.
- A **macro** (which stands for "macroinstruction") is a programmable pattern which translates a certain sequence of input into a preset sequence of output. **Macros** can be used to make tasks less repetitive by representing a complicated sequence of keystrokes, mouse movements, commands, or other types of input.

Macro definition:

name MACRO [parameters,...]

statements >

ENDM

Example:

```
MyMacro    MACRO P1, P2, P3
           MOV AX, P1
           MOV BX, P2
           MOV CX, P3
ENDM
```

Advantages of macros

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages of macros

- In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

Procedure (PROC)

- This directive marks the start and end of a procedure block called *label*. The statements in the block can be called with the **CALL** instruction.

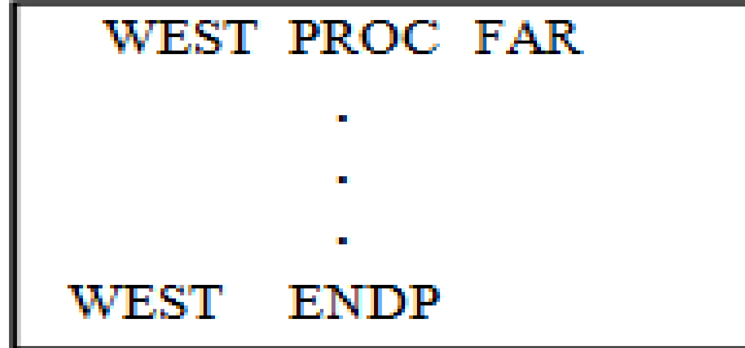
PROC definition:

label **PROC** [[near / far]]

<Procedure instructions>

label **ENDP**

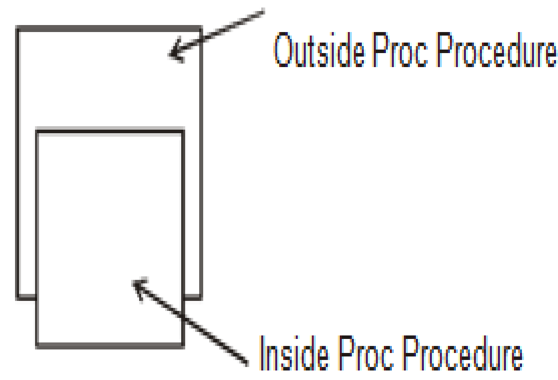
Example:

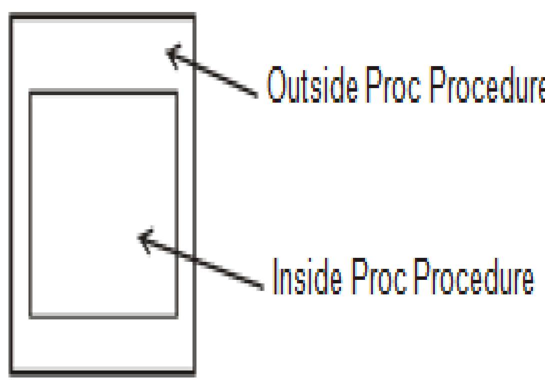


```
  WEST  PROC  FAR
          .
          .
          .
WEST   ENDP
```

Nested Proc

Overlapping Proc





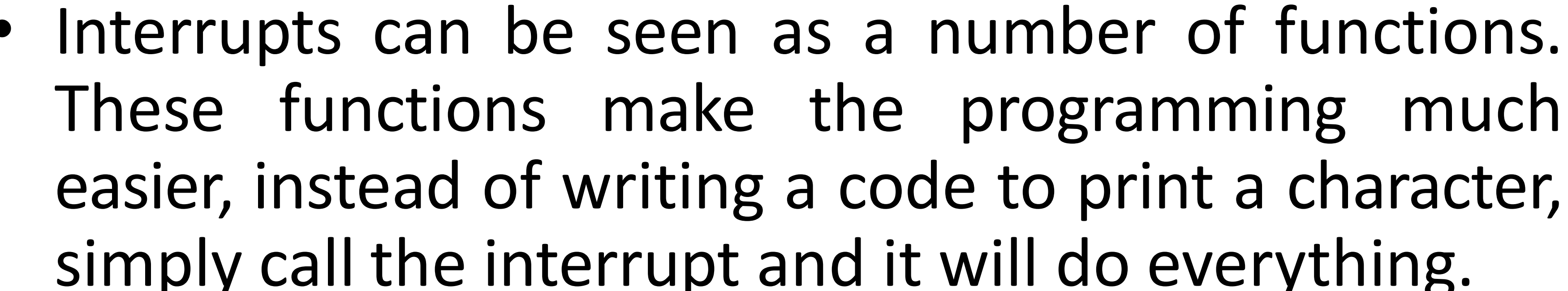
Differences between Macros and Procedures

S.No.	PROCEDURES	MACROS
1.	To use a procedure **CALL** and **RET** instructions are needed	To use a macro, just type its name.
2.	It occupies less memory.	It occupies more memory.
3.	Stack is used.	Stack is not used.
4.	To mark the end of the procedure, type the name of the procedure before the **ENDP** directive.	To mark the end of the macro **ENDM** directive is enough.
5.	Overhead time is required to call the procedure and return to the calling program.	No overhead time during the execution

INTERRUPTS AND INTERRUPT SERVICE ROUTINES

Interrupts

- A signal to the processor to halt its current operation and immediately transfer control to an interrupt service routine is called as interrupt. Interrupts are triggered either by hardware, as when the keyboard detects a key press, or by software, as when a program executes the INT instruction.



- Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character, simply call the interrupt and it will do everything.
- There are also interrupt functions that work with disk drive and other hardware. They are called as **software interrupts**.
- Interrupts are also triggered by different hardware, these are called **hardware interrupts**.
- To make a software interrupt there is an INT instruction, it has very simple syntax: INT *value*.
- Where value can be a number between 0 to 255 (or 00 to FF H).

Interrupt Service Routines (ISRs)

- ISR is a routine that receives processor control when a specific interrupt occurs.
- The 8086 will directly call the service routine for 256 vectored interrupts without any software processing. This is in contrast to non vectored interrupts that transfer control directly to a single interrupt service routine, regardless of the interrupt source.

Interrupt vector table:

AVAILABLE INTERRUPT	3FF H 3FC H	TYPE 255 POINTER: (AVAILABLE)
		. .
	084 H	TYPE 33 POINTER: (AVAILABLE)
	080 H	TYPE 32 POINTER: (AVAILABLE)
RESERVED INTERRUPT	07F H	TYPE 31 POINTER: (AVAILABLE)
		. .
	014 H	TYPE 5 POINTER: (RESERVED)
DEDICATED INTERRUPT	010 H	TYPE 4 POINTER: OVERFLOW
	00C H	TYPE 3 POINTER: 1-BYTE INT INSTRUCTION
	008 H	TYPE 2 POINTER: NON MASKABLE
	004 H	TYPE 1 POINTER: SINGLE STEP
	000 H	TYPE 0 POINTER: DIVIDE ERROR

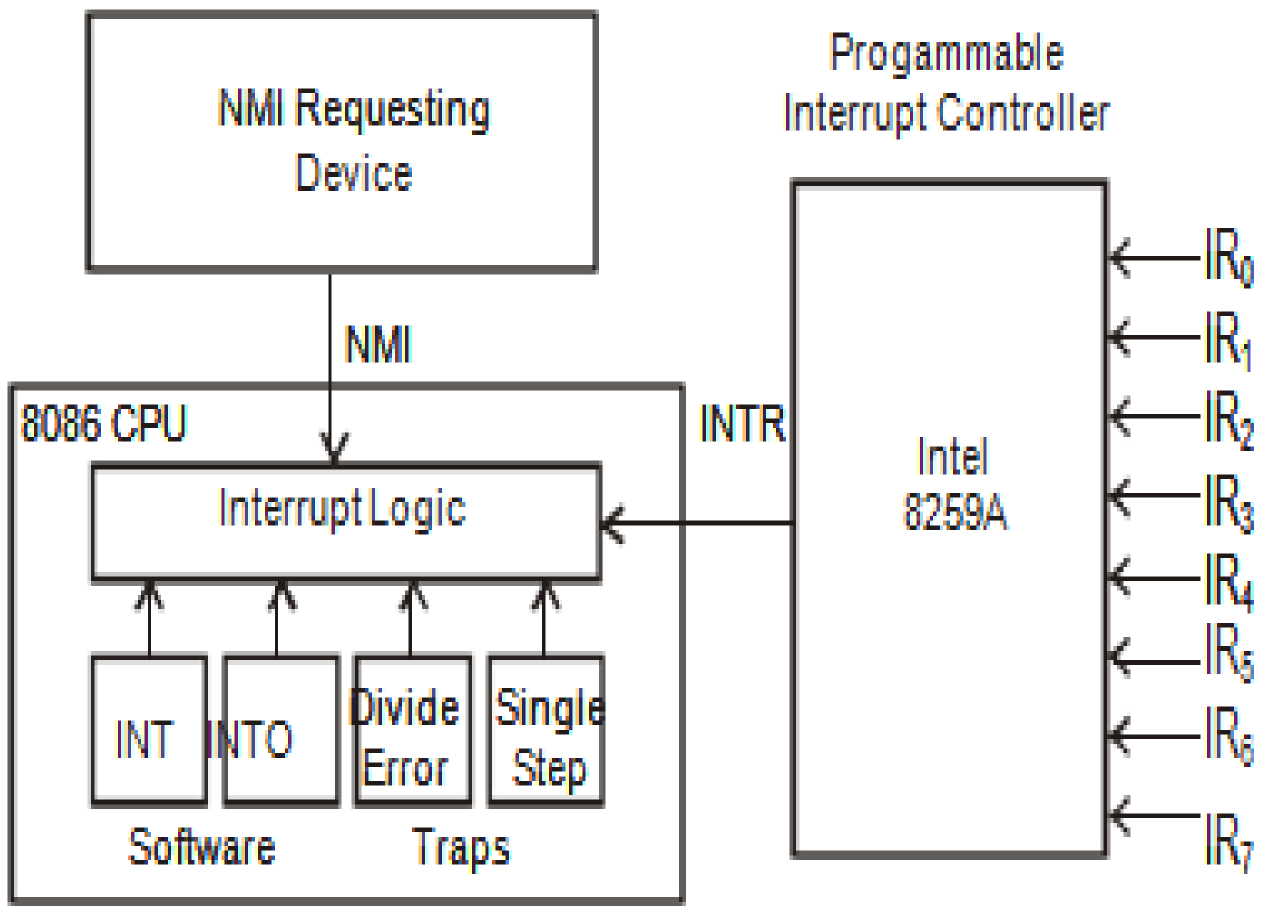
When an interrupt occurs, regardless of source, the 8086 does the following:

- The CPU pushes the flags register onto the stack.
- The CPU pushes a far return address (segment:offset) onto the stack, segment value first.
- The CPU determines the cause of the interrupt (i.e., the interrupt number) and fetches the four byte interrupt vector from address 0 **:** vector x 4 (0:0, 0:4, 0:8 etc)
- The CPU transfers control to the routine specified by the interrupt vector table entry.

After the completion of these steps, the interrupt service routine takes control. When the interrupt service routine wants to return control, it must execute an IRET (interrupt return) instruction. The interrupt return pops the far return address and the flags off the stack

Types of Interrupts

- Hardware Interrupt - External uses INTR and NMI
- Software Interrupt - Internal - from INT or INTO
- Processor Interrupt - Traps and 10 Software Interrupts
- *External* - generated outside the CPU by other hardware (INTR, NMI)
- *Internal* - generated within CPU as a result of an instruction or operation (INT, INTO, Divide Error and Single Step)



NMI Requesting Device
NMI
Progammable Interrupt Controller
8086 CPU
Interrupt Logic
INTR
Intel 8259A
INT
INTO
Divide Error
Single Step
Software
Traps
IR0
IR1
IR2
IR3
IR4
IR5
IR6
IR7

Dedicated Interrupts

- **Divide Error Interrupt (Type 0)**

  This interrupt occurs automatically following the execution of DIV or IDIV instructions when the quotient exceeds the maximum value that the division instructions allow.

- **Single Step Interrupt (Type 1)**

  This interrupt occurs automatically after execution of each instruction when the Trap Flag (TF) is set to 1.It is used to execute programs one instruction at a time, after which an interrupt is requested. Following the ISR, the next instruction is executed and another single stepping interrupt request occurs.

- **Non Maskable Interrupt (Type 2)**

  It is the highest priority hardware interrupt that triggers on the positive edge.

  This interrupt occurs automatically when it receives a low-to-high transition on its NMI input pin.

  This interrupt cannot be disabled or masked. It is used to save program data or processor status in case of system power failure.

- **Breakpoint Interrupt (Type 3)**

  This interrupt is used to set break points in software debugging programs.

- **Overflow Interrupt (Type 4)**

Software Interrupts (INT n)

- The software interrupts are non maskable interrupts. They are higher priority than hardware interrupts.

Hardware Interrupts

- INTR and NMI are called hardware interrupts. INTR is maskable and NMI is non-maskable interrupts.

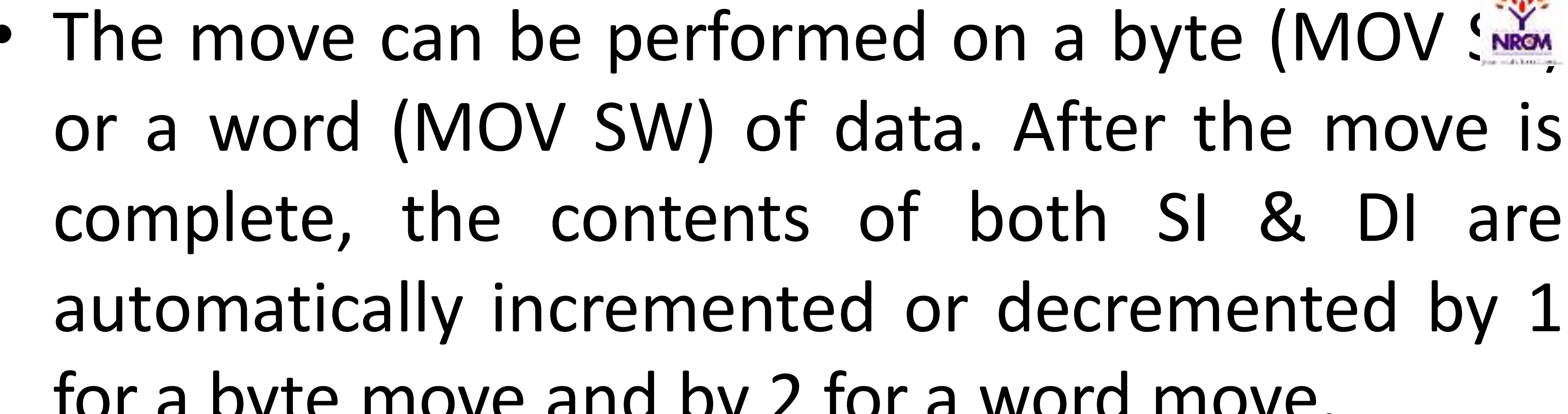
Interrupt Priority

Interrupt	Priority
INT n, INTO, Divide Error	Highest
NMI	↓
INTR	↓
Single Step	Lowest

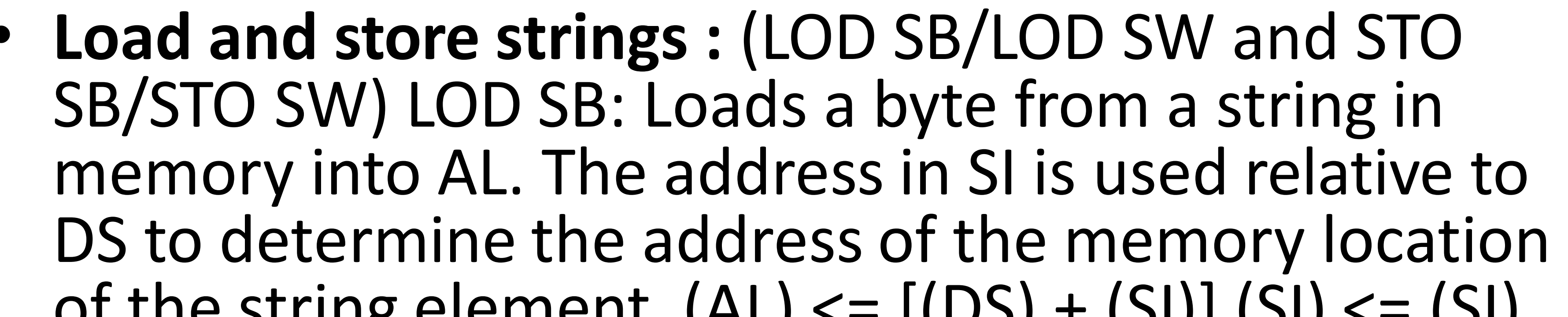
Byte And String Manipulation

- The 8086 microprocessor is equipped with special instructions to handle string operations.
- By string we mean a series of data words or bytes that reside in consecutive memory locations.
- The string instructions of the 8086 permit a programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory.

- A second type of operation that is easily performed is to scan a string and data elements stored in memory looking for a specific value.
- Other examples are to compare the elements and two strings together in order to determine whether they are the same or different.
- **Move String** : MOV SB, MOV SW: An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register.



- The move can be performed on a byte (MOV SB) or a word (MOV SW) of data. After the move is complete, the contents of both SI & DI are automatically incremented or decremented by 1 for a byte move and by 2 for a word move.
- Address pointers SI and DI increment or decrement depends on how the direction flag DF is set.



- **Load and store strings :** (LOD SB/LOD SW and STO SB/STO SW) LOD SB: Loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element. (AL) <= [(DS) + (SI)] (SI) <= (SI) + 1
- LOD SW : The word string element at the physical address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2. (AX) <= [(DS) + (SI)] (SI) <= (SI) + 2
- STO SB : Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the address of the storage location in memory [(ES) + (DI)] <= (AL) (DI) <=(DI) + 1
- STO SW : [(ES) + (DI)] <= (AX) (DI) <= (DI) + 2

1.A. 16 BIT ADDITION USING 8086

ADDRESS	LABEL	MNEMONICS	OPCODE	COMMENTS
1000		MOV AX,[1200]H	A1	Clear C register
			00	
			12	
1003		ADD AX,[1202]H	03	Move the immediate data 1 to accumulator
			06	
			02	
			12	
1007		MOV[1204]H,AX	A3	Move the immediate data 2 to B register
			04	
			12	
100A		HLT	F4	End the program

INPUT		OUTPUT	
1200	04	1204	05
1201	02	1205	07
1202	01		
1203	05		

1.B. 16BIT SUBTRACTION USING 8086

ADDRESS	LABEL	MNEMONICS	OPCODE	COMMENTS
1000		MOV AX,[1200]H	A1	Clear C register
			00	
			12	
1003		SUB AX,[1202]H	2B	Move the immediate data 1 to accumulator
			06	
			02	
			12	
1007		MOV[1204]H, AX	A3	Move the immediate data 2 to B register
			04	
			12	
100A		HLT	F4	End the program

INPUT		OUTPUT	
1200	08	1204	06
1201	04	1205	01
1202	02		
1203	03		

INPUT		OUTPUT	
2000	02	2100	06
2001	03	2101	00
2002	03	2102	09
2003	03	2103	00

INPUT		OUTPUT	
2000	00	2100	00
2001	90	2101	00
2002	00	2102	03
2003	30	2103	00

8086 program to determine largest number in an array of n numbers

Algorithm –

- Load data from offset 500 to register CL and set register CH to 00 (for count).
- Load first number(value) from next offset (i.e 501) to register AL and decrease count by 1.
- Now compare value of register AL from data(value) at next offset, if that data is greater than value of register AL then update value of register AL to that data else no change, and increase offset value for next comparison and decrease count by 1 and continue this till count (value of register CX) becomes 0.
- Store the result (value of register AL) to memory address 2000 : 600.

MEMORY ADDRESS	MNEMONICS	COMMENT
400	MOV SI, 500	SI<-500
403	MOV CL, [SI]	CL<-[SI]
405	MOV CH, 00	CH<-00
407	INC SI	SI<-SI+1
408	MOV AL, [SI]	AL<-[SI]
40A	DEC CL	CL<-CL-1
40C	INC SI	SI<-SI+1
40D	CMP AL, [SI]	AL-[SI]
40F	JNC 413	JUMP TO 413 IF CY=0
411	MOV AL, [SI]	AL<-[SI]
413	INC SI	SI<-SI+1
414	LOOP 40D	CX<-CX-1 & JUMP TO 40D IF CX NOT 0
416	MOV [600], AL	AL->[600]
41A	HLT	END

Output Data
40
Memory Address(offset)
600

Explanation –

- **MOV SI, 500** : set the value of SI to 500
- **MOV CL, [SI]** : load data from offset SI to register CL
- **MOV CH, 00** : set value of register CH to 00
- **INC SI** : increase value of SI by 1.
- **MOV AL, [SI]** : load value from offset SI to register AL
- **DEC CL** : decrease value of register CL by 1
- **INC SI** : increase value of SI by 1
- **CMP AL, [SI]** : compares value of register AL and [SI] (AL-[SI])
- **JNC 413** : jump to address 413 if carry not generated
- **MOV AL, [SI]** : transfer data at offset SI to register AL
- **INC SI** : increase value of SI by 1
- **LOOP 40C** : decrease value of register CX by 1 and jump to address 40D if value of register CX is not zero
- **MOV [600], AL** : store the value of register AL to offset 600
- **HLT** : stop

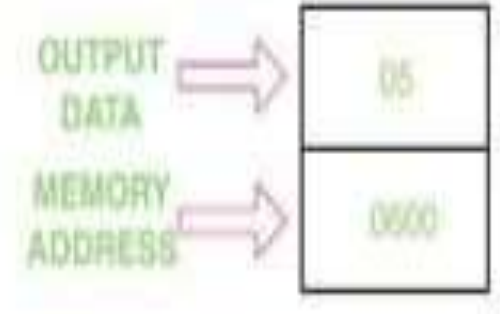
8086 program to find the min value in a given array

Algorithm –

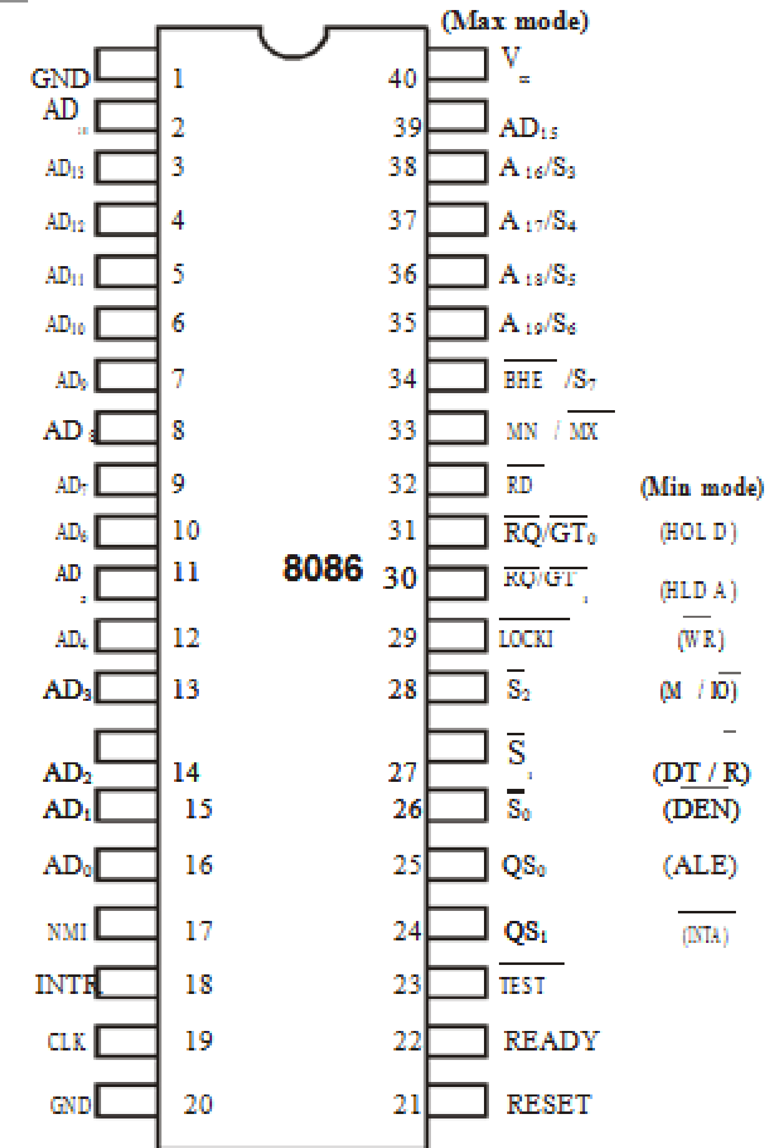
- Assign value 500 in SI and 600 in DI
- Move the contents of [SI] in CL and increment SI by 1
- Assign the value 00 H to CH
- Move the content of [SI] in AL
- Decrease the value of CX by 1
- Increase the value of SI by 1
- Move the contents of [SI] in BL
- Compare the value of BL with AL
- Jump to step 11 if carry flag is set
- Move the contents of BL in AL
- Jump to step 6 until the value of CX becomes 0, and decrease CX by 1
- Move the contents of AL in [DI]
- Halt the program

MEMORY ADDRESS	MNEMONICS	COMMENTS
0400	MOV SI, 500	SI <- 500
0403	MOV DI, 600	DI <- 600
0406	MOV CL, [SI]	CL <- [SI]
0408	MOV CH, 00	CH <- 00
040A	INC SI	SI <- SI+1
040B	MOV AL, [SI]	AL <- [SI]
040D	DEC CX	CX <- CX-1
040E	INC SI	SI <- SI+1
040F	MOV BL, [SI]	BL <- [SI]
0411	CMP AL, BL	AL-BL
0413	JC 0417	Jump if carry is 1
0415	MOV AL, BL	AL <- BL
0417	LOOP 040E	Jump if CX not equal to 0
0419	MOV [DI], AL	[DI] <- AL
041B	HLT	End of the program

INPUT DATA	04	18	AD	05	2B
MEMORY ADDRESS	0500	0501	0502	0503	0504



PIN DIAGRAM



MINIMUM MODE SIGNALS

Address/data/status		
AD_{15}-AD_0 A_{19}/S_6-A_{16}/S_3	Address/data bus Address/status bus	Bidirectional, 3-state output,3-state
$\overline{RD}$ READY M/ $\overline{IO}$ $\overline{WR}$ ALE DT/ $\overline{R}$ $\overline{DEN}$ $\overline{BHE}$ /S_7	Read from memory/IO Ready signal Select memory or IO Write to memory/IO Address latch enable Data transmit/receive Data bus enable Bus high enable	output,3-state Input output,3-state output,3-state output output output output
INTR NMI RESET $\overline{INTA}$	Interrupt request Non-maskable interrupt Reset Interrupt acknowledge	Input Input Input output

HOLD	Hold request	Input
HLDA	Hold acknowledge	output
$\overline{TEST}$	Test pin tested by WAIT instruction	Input
MN/ $\overline{MX}$	Minimum/maximum mode, 5V	Input
CLK	Clock pin for basic timing signal	Input
V_{cc}	Power supply, +5 V	
GND	Ground connection, 0V	

MAXIMUM MODE SIGNALS

Address/data/status		
AD_{15}-AD_0 A_{19}/S_6-A_{16}/S_3	Address/data bus Address/status bus	Bidirectional, 3-state output,3-state
$\overline{RD}$ READY $\overline{BHE}/S_7$ $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$	Read from memory/IO Ready signal Bus high enable Status/handshake bits indicating the function of the current bus cycle	output,3-state input output output
INTR NMI RESET	Interrupt request Non-maskable interrupt Reset	input input input

$\overline{RQ}/\overline{GT}_1$, $\overline{RQ}/\overline{GT}_0$	Request/grant pins for bus access	bidirectional
$\overline{LOCK}$	Used to lock the bus, activated by LOCK prefix on any instruction	output
QS_1, QS_0	Queue status	output
$\overline{TEST}$	Test pin tested by WAIT instruction	input
$MN/\overline{MX}$	Minimum/maximum mode, 0V	input
CLK	Clock pin for basic timing signal	input
V_{cc}	Power supply, +5 V	
GND	Ground connection, 0V	

Address / Data Bus (AD_{15}–AD_0)

- The multiplexed Address/ Data bus acts as address bus during the first part of machine cycle (T1) and data bus for the remaining part of the machine cycle.

Address/Status (A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3)

- During T1 these are the four most significant address lines for memory operations.
- During I/O operations these lines are LOW.

S_4	S_3	Function
0	0	ES, Extra segment
0	1	SS, Stack Segment
1	0	CS, Code segment
1	1	DS, Data segment

$\overline{BHE}$	A_0	Characteristics
0	0	Whole word
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

Read(RD)

- This signal is used to read data from memory or I/O device which reside on the 8086 local bus.

Ready

- If this signal is low the 8086 enters into WAIT state.
- The READY signal from memory/ IO is synchronized by the 8284A clock generator to form READY.
- This signal is active HIGH.

Interrupt Request (INTR)

- It is a level triggered maskable interrupt request.
- A subroutine is vectored via an interrupt vector lookup table located in system memory.

TEST

- This input is examined by the “Wait” instruction.
- If the TEST input is LOW execution continues,
- otherwise the processor waits in an ``Idle” state.

Non-Maskable Interrupt (NMI)

- It is an edge triggered input which causes a type 2 interrupt.
- NMI is not maskable internally by software.

Reset

- This signal is used to reset the 8086.
- It causes the processor to immediately terminate its present activity.
- The signal must be active HIGH for at least four clock cycles.
- It restarts execution when RESET returns LOW.

Clock (CLK)

- This signal provides the basic timing for the processor and bus controller.
- The clock frequency may be 5 MHz or 8 MHz or 10 MHz depending on the version of 8086.

V_{CC}

- It is a +5V power supply pin.

Ground (GND)

- Two pins (1 and 20) are connected to ground ie, 0 V power supply.

Minimum/Maximum (MN/ MX)

- This pin indicates what mode the processor is to operate in.

MEMORY / IO (M/ IO)

- It is used to distinguish a memory access from an I/O access. M = HIGH, I/O = LOW.

WRITE(WR)

- It indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the M/ IO signal.
- **Interrupt Acknowledge (INTA)** This signal indicates recognition of an interrupt request. It is used as a read strobe for interrupt acknowledge cycles.

Address Latch Enable (ALE)

- This signal is used to demultiplex the AD_0-AD_{15} into A_0-A_{15} and D_0-D_{15}. It is a HIGH pulse active during T1 of any bus cycle.

Data Enable(DEN)

This signal informs the transceivers (8286/8287) that the 8086 is ready to send or receive data.

Hold

- This signal indicates that another master (DMA or processor) is requesting the host 8086 to handover the system bus.

Hold Acknowledge (HLDA)

- On receiving HOLD signal 8086 outputs HLDA signal HIGH as an acknowledgement.

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Machine cycle
0	0	0	Interrupt acknowledge
0	0	1	I/O read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

Request/Grant (RQ / GT_0 , RQ / GT_1)

- These pins are used by other local bus masters to force RQ / GT_1 the processor to release the local bus at the end of the processor's current bus cycle

LOCK

- This signal indicates that other system bus masters are not to gain control of the system bus while LOCK is active LOW.
- The LOCK signal is activated by the "LOCK" prefix instruction and remains active until the completion of the next instruction.

QUEUEue Status (QS_1, QS_0)

- The queue status is valid during the CLK cycle after which the queue operation is performed.

QS_1	QS_0	Characteristics
0	0	No operation
0	1	First byte of opcode from Queue
1	0	Empty the Queue
1	1	Subsequent byte from Queue

SYSTEM BUS STRUCTURE

- System bus is a single computer bus that connects the major components of a computer system.
- It consists of data bus, address bus and control bus.
- To communicate with external world, microprocessor make use of buses.

DATA BUS

- It is used for the exchange of d
  between the processor, memory a
  peripherals.
- It is bi-directional so that it all
  data flow in both directions.
- The width of the data bus can differ
  every microprocessor.
- When the microprocessor issues
  address of the instruction, it gets b
  the instruction through the data bus.

ADDRESS BUS

- The address bus contains the connections between the microprocessor and memory or output devices
- It is unidirectional.
- The width of the address bus corresponds to the maximum addressing capacity

CONTROL BUS

- The control bus carries the signals relating to the control and coordination of the various activities across the computer, which can be sent from the control unit within the CPU.
- Microprocessor uses control bus to process data, that is what to do with the selected memory location.

CPU
ALU
Registers
and Controls

Memory

Input and
Output

System Bus

Data Bus

Address Bus

Control Bus

MIN-MAX MODE OF OPERATION

Intel 8086 has two modes of operation. They are:

- Minimum mode
- Maximum mode

- When only 8086 microprocessor is to be used in a microcomputer system, the 8086 is used in the **minimum mode** of operation.
- In this mode, the microprocessor issues the control signals required by memory or I/O devices.
- In a multiprocessor system it operates in the **maximum mode**. In this mode, the control signals are issued by Intel 8288 bus controller.

- The pin MN/ MX (33) decides the operating mode of 8086.
- When MN/ MX = 0, maximum mode of operation.

  = 1, minimum mode of operation.
- Pins 24 to 31 have different functions for minimum mode and maximum mode.

Minimum Mode

- For minimum mode of operation MN/ MX is connected to $\mathbf{V}_{CC}$ (+5 volts).

- All control signals for controlling memory and I/O devices are generated inside the 8086 microprocessor.
- In this mode , peripheral devices can be used with the microprocessor without any special consideration

Crystal Oscillator

Vcc

8284 A Clock Generator

CLK

RESET

MN/MX

RD

WR

M/IO

RAM

EPROM

I/O Devices

Interrupt Controller

From Interrupt Controller

INTR

To Interrupt Controller

INTA

From DMA Controller

HOLD

To DMA Controller

HLDA

INTEL 8086 CPU

ALE

STB

GND

OE

BHE

Addr/Data

8282 Latch 2 or 3

Address

8286 Transceiver (2)

Data

DT/R

T

DEN

OE

$M/\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation
0	0	1	I/O Read
0	1	0	I/O Write
1	0	1	Memory Read
1	1	0	Memory Write

READ CYCLE

One Bus Cycle

T_1 T_2 T_3 T_4

CLK

Address, BHE OUT

$A_{19}/S_6 - A_{16}/S_3$ and $\overline{BHE}/S_7$

Status OUT

AD_{15}–AD_0

Address OUT

Data IN

ALE

$M/\overline{IO}$

LOW = I/O Read, HIGH = Memory Read

$\overline{RD}$

$DT/\overline{R}$

$\overline{DEN}$

WRITE CYCLE

One Bus Cycle

CLK

T_1 T_2 T_3 T_4

Address, BHE OUT

$A_{19}/S_6 - A_{16}/S_3$ and $\overline{BHE}/S_7$

Status OUT

$AD_{15}-AD_0$

Address OUT

Data IN

ALE

$M/\overline{IO}$

LOW = I/O Write, HIGH = Memory Write

$\overline{WR}$

$DT/\overline{R}$

$\overline{DEN}$

Maximum mode operation’

Crystal
Oscillator
8284 A
Clock Generator
GND
CLK
MN/MX
S0
S1
S2
LOCK
RQ / GT1
RQ / GT2
INTR
INTEL
8086
CPU
BHE
AD0–AD15
A16–A19
GND
Addr/Data
S0
S1
S2
8288
BUS
Controller
NC
AMWC
DEN
DT/R
ALE
MRDC
MWTC
IORC
IOWC
INTA
AIOWC
NC
ST
OE
B
8282
Latch
2 or 3
Address Bus
T
OE
8286
Transceiver
(2)
Data Bus
RAM
EPROM
I/O Devices
Interrupt
Controller
DMA
Controller

- In maximum mode 8086 based system, an external Bus Controller (Intel 8288) has to be employed to generate the bus control signals.
- The important signals are :
- MRDC - Memory Read Command
  MWTC - Memory Write Command
  IORC - I/O Read Command
  IOWC - I/O Write Command
  AMWC - Advanced Memory Write Command
  AIOWC - Advanced I/O Write Command

- Three numbers of 8 bit latches (Intel 8282) are employed to demultiplex the address lines.
- The latches are enabled by using the ALE signal generated by the bus controller.
- Two numbers of octal bus transceivers (Intel 8286) are used as data transceivers.
- The signals DEN and DT/ R are generated by the bus controller are used as enable and direction control respectively.
- The clock generator (Intel 8284) is used to generate clock, reset and ready signals for 8086.
- A quartz crystal of frequency 15 MHz is connected to 8284.

Read cycle

Write cycle

One Bus Cycle

T_1 T_2 T_3 T_4

CLK

$\overline{S_2}$–$\overline{S_0}$

$\overline{S_2}$–$\overline{S_0}$

$\overline{S_2}$–$\overline{S_0}$ Inactive

$\overline{BHE}$, A_{19}–A_{16}

Address/Status and $\overline{BHE}$/S

$\overline{S_7}$–$\overline{S_3}$

Float

Address/data AD_{15}–AD_0

A_{15}–A_0

Data IN D_{15}–D_0

*ALE

*$\overline{AMWC}$ or $\overline{AIOWC}$

*$\overline{MWTC}$ or $\overline{IOWC}$

*DEN

*8288 Bus Controller Outputs