Internal Operation of 8086

1

Inside The 8086/8088

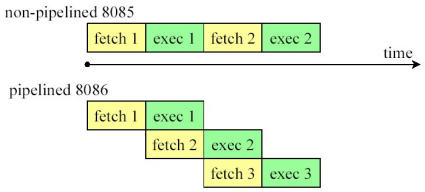
Concepts important to the internal operation of 8088/8086

- Pipelining
- Registers

2

Inside The 8086/8088…*pipelining*

- Pipelining
  - Two ways to make CPU process information faster:
    - Increase the working frequency – technology dependent
    - Change the internal architecture of the CPU
  - Pipelining is to allow Microprocessor to fetch and execute at the same time



3

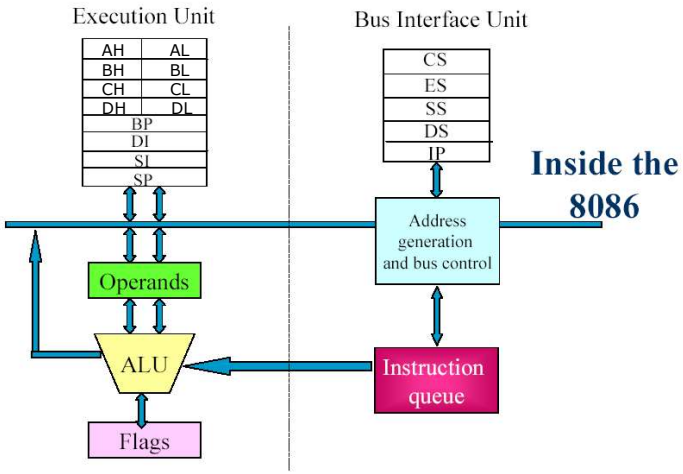
Inside The 8088/8086…*pipelining*

Intel implemented the concept of pipelining by splitting the internal structure of the 8086/8088 into two sections that works simultaneously:

- Execution Unit (EU) – executes instructions previously fetched
- Bus Interface Unit (BIU) – accesses memory and peripherals

4

Inside The 8088/8086



5

Overview

- Registers
  - General purpose registers (8)
    - Operands for logical and arithmetic operations
    - Operands for address calculations
    - Memory pointers
  - Segment registers (6)
  - FLAGS register
  - The instruction pointer register
- The stack

6

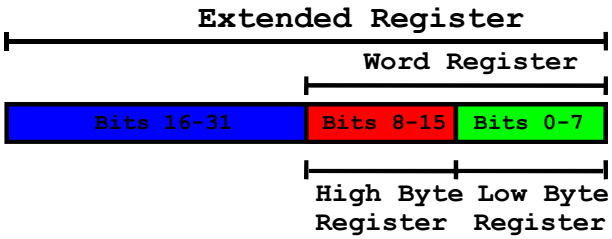
Inside The 8088/8086...*registers*

- Registers
  - To store information temporarily

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment)
		SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

7

Anatomy of a Register



8

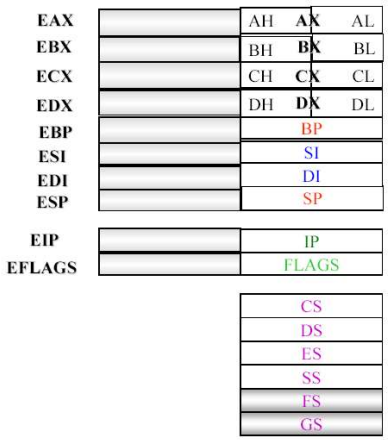
General Registers

32 bit Registers		16 bit Registers		8 bit Registers	
EAX	EBP	AX	BP	AH	AL
EBX	ESI	BX	SI	BH	BL
ECX	EDI	CX	DI	CH	CL
EDX	ESP	DX	SP	DH	DL
Bits 16-31		Bits 8-15		Bits 0-7	

9

Registers
6 Category

- General
- Pointer
- Index
- Segment
- Instruction
- Flag



10

Register names

- Accumulator
- Base index
- Count
- Data
- Stack Pointer
- Base Pointer
- Destination index
- Source index
- Instruction Pointer
- Flags

Segment registers

- Code
- Data
- Extra
- Stack

11

General Registers I

- **EAX/AX** – 'Accumulator'
  - accumulator for operands and results data
  - usually used to store the return value of a procedure
- **EBX/BX** – 'Base Register'
  - pointer to data in the DS segment
- **ECX/CX** – 'Counter'
  - counter for string and loop operations
- **EDX/DX** – 'Data Register'
  - I/O pointer

12

General Registers II

- **ESI/SI** – 'Source Index'
  - source pointer for string operations
  - typically a pointer to data in the segment pointed to by the DS register
- **EDI/DI** – 'Destination Index'
  - destination pointer for string operations
  - typically a pointer to data/destination in the segment pointed to by the ES register

13

General Registers III

- **EBP/BP** – 'Base Pointer'
  - pointer to data on the stack
  - points to the current stack frame of a procedure
- **ESP/SP**– 'Stack Pointer'
  - pointer to the top address of the stack
  - holds the stack pointer and as a general rule should not be used for any other purpose

14

Segment Registers

- **`CS`** – 'Code Segment'
  - contains the segment selector for the code segment where the instructions being executed are stored
- **`DS(ES,FS,GS)`** – 'Data Segment'
  - contains the segment selectors for the data segment where data is stored
- **`SS`** – 'Stack Segment'
  - contains the segment selector for the stack segment, where the procedure stack is stored

15

The EFLAGS/FLAG Register I

- Carry Flag – `CF` (bit 0)
  - **Set** if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; **cleared** otherwise.
- Parity Flag – `PF` (bit 2)
  - **Set** if the least-significant byte of the result contains an even number of 1 bits; **cleared** otherwise.
- Adjust Flag – `AF` (bit 4)
  - Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; **cleared** otherwise.

16

The EFLAGS Register II

- Zero Flag – `ZF` (bit 6)
  - **Set** if the result is zero; **cleared** otherwise
- Sign Flag – `SF` (bit 7)
  - **Set** equal to the most-significant bit of the result, which is the sign bit of a signed integer
- Overflow Flag – `OF` (bit 11)
  - **Set** if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; **cleared** otherwise

17

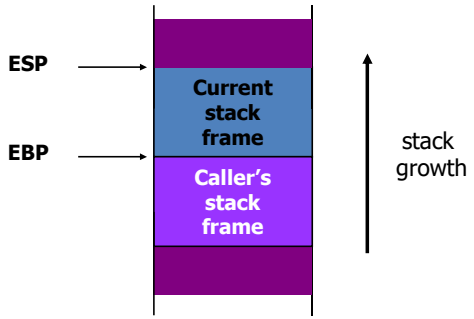
Instruction Pointer

- **`EIP/IP`**
  - 'Instruction Pointer'
  - Contains the offset within the code segment of the next instruction to be executed
  - Cannot be accessed directly by software

18

The Stack

The stack starts in high memory and grows toward low memory



19

Addressing Modes
Instruction set
Interfacing memory and i/o ports

20

Introduction

Program is a set of instructions written to solve a problem. Instructions are the directions which a microprocessor follows to execute a task or part of a task. Broadly, computer language can be divided into two parts as high-level language and low level language. Low level language are machine specific. Low level language can be further divided into machine language and assembly language.

Machine language is the only language which a machine can understand. Instructions in this language are written in binary bits as a specific bit pattern. The computer interprets this bit pattern as an instruction to perform a particular task. The entire program is a sequence of binary numbers. This is a machine-friendly language but not user friendly. Debugging is another problem associated with machine language.

To overcome these problems, programmers develop another way in which instructions are written in English alphabets. This new language is known as Assembly language. The instructions in this language are termed *mnemonics. As* microprocessor can only understand the machine language so mnemonics are translated into machine language either manually or by a program known as *assembler.*

Efficient software development for the microprocessor requires a complete familiarity with the instruction set, their format and addressing modes. Here in this chapter, we will focus on the addressing modes and instructions formats of microprocessor 8086.

Introduction

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)

DATA SEGMENT                ;Assembler directive

        ORG 1104H           ;Assembler directive
        SUM DW 0            ;Assembler directive
        CARRY DB 0          ;Assembler directive

DATA ENDS                   ;Assembler directive

CODE SEGMENT                ;Assembler directive

        ASSUME CS:CODE      ;Assembler directive
        ASSUME DS:DATA      ;Assembler directive
        ORG 1000H           ;Assembler directive

        MOV AX,205AH        ;Load the first data in AX register
        MOV BX,40EDH        ;Load the second data in BX register
        MOV CL,00H          ;Clear the CL register for carry
        ADD AX,BX           ;Add the two data, sum will be in AX
        MOV SUM,AX          ;Store the sum in memory location (1104H)
        JNC AHEAD           ;Check the status of carry flag
        INC CL              ;If carry flag is set,increment CL by one
AHEAD:  MOV CARRY,CL        ;Store the carry in memory location (1106H)
        HLT

CODE ENDS                   ;Assembler directive
END                         ;Assembler directive
```

Program
A set of instructions written to solve a problem.

Instruction
Directions which a microprocessor follows to execute a task or part of a task.

Computer language

High Level | **Low Level**

Machine Language

- **Binary bits**

Assembly Language

- **English Alphabets**
- **'Mnemonics'**
- **Assembler**
  Mnemonics → Machine Language

ADDRESSING MODES

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing 2. Immediate Addressing	Group I : Addressing modes for register and immediate data
3. Direct Addressing 4. Register Indirect Addressing 5. Based Addressing 6. Indexed Addressing 7. Based Index Addressing 8. String Addressing	Group II : Addressing modes for memory data
9. Direct I/O port Addressing 10. Indirect I/O port Addressing	Group III : Addressing modes for I/O ports
11. Relative Addressing	Group IV : Relative Addressing mode
12. Implied Addressing	Group V : Implied Addressing mode

8086 Microprocessor

Addressing Modes

Group I : Addressing modes for register and immediate data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

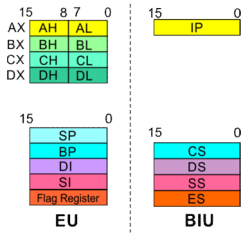
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

The content of 8-bit register DH is moved to another 8-bit register CL

(CL) ← (DH)

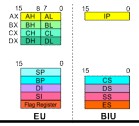


8086 Microprocessor

Addressing Modes

Group I : Addressing modes for register and immediate data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

The 8-bit data (08_H) given in the instruction is moved to DL

(DL) ← 08_H

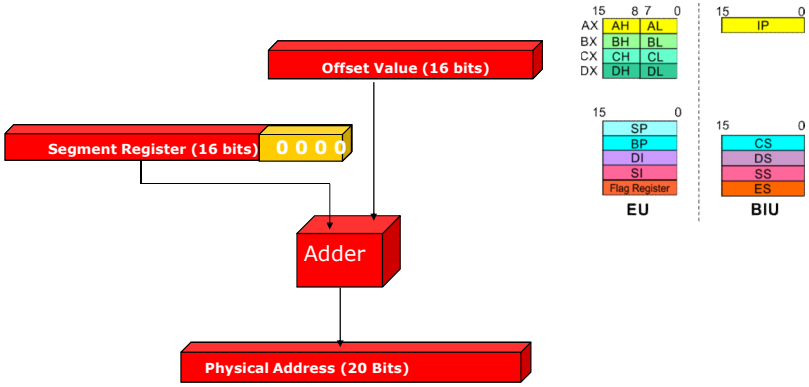
MOV AX, 0A9FH

The 16-bit data ($0A9F_H$) given in the instruction is moved to AX register

(AX) ← $0A9F_H$

8086 Microprocessor

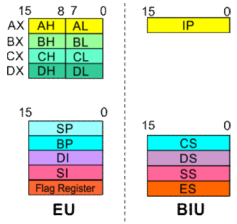
Addressing Modes : Memory Access



8086 Microprocessor

Addressing Modes : Memory Access

- **20 Address lines ⇒ 8086 can address up to 2^{20} = 1M bytes of memory**
- **However, the largest register is only 16 bits**
- **Physical Address will have to be calculated**
  Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- **Memory Address represented in the form –**
  Seg : Offset (Eg - 89AB:F012)
- **Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20- bit address**



16 bytes of contiguous memory

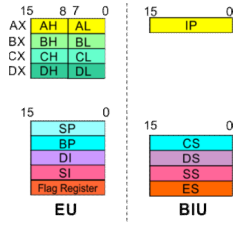
89AB : F012 → 89AB → 89AB0 (Paragraph to byte → 89AB x 10 = 89AB0)
F012 → 0F012 (Offset is already in byte unit)
+ -------
98AC2 (The absolute address)

8086 Microprocessor

Addressing Modes : Memory Access

- **To access memory we use these four registers: BX, SI, DI, BP**
- **Combining these registers inside [ ] symbols, we can get different memory locations (Effective Address, EA)**
- **Supported combinations:**

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]



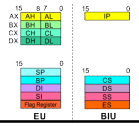
BX **BP**	**SI** **DI**	**+ disp**

8086 Microprocessor

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

The effective address is just a 16-bit number written directly in the instruction.

Example:

```
MOV  BX, [1354H]
MOV  BL, [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

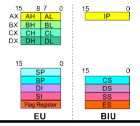
This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.

8086 Microprocessor

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

Operations:

EA = (BX)
BA = (DS) x 16_{10}
MA = BA + EA

(CX) ← (MA) or,

(CL) ← (MA)
(CH) ← (MA +1)

Note : Register/ memory enclosed in brackets refer to content of register/ memory

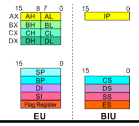
31

8086 Microprocessor

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, BX or BP is used to hold the base value for effective address and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

When BX holds the base value of EA, 20-bit physical address is calculated from BX and DS.

When BP holds the base value of EA, BP and SS is used.

Example:

MOV AX, [BX + 08H]

Operations:

0008_H ← 08_H (Sign extended)
EA = (BX) + 0008_H
BA = (DS) x 16_{10}
MA = BA + EA

(AX) ← (MA) or,

(AL) ← (MA)
(AH) ← (MA + 1)

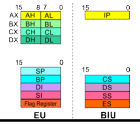
32

8086 Microprocessor

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

$EA = (SI) + FFA2_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(CX) \leftarrow (MA)$ or,

$(CL) \leftarrow (MA)$
$(CH) \leftarrow (MA + 1)$

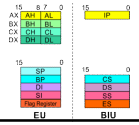
33

8086 Microprocessor

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$
$BA = (DS) \times 16_{10}$
$MA = BA + EA$

$(DX) \leftarrow (MA)$ or,

$(DL) \leftarrow (MA)$
$(DH) \leftarrow (MA + 1)$

34

8086 Microprocessor

Addressing Modes

Group II : Addressing modes for memory data

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in SI register and the EA of destination is stored in DI register.

Segment register for calculating base address of source data is DS and that of the destination data is ES

Example: MOVS BYTE

Operations:

Calculation of source memory location:
EA = (SI) BA = (DS) x 16_{10} MA = BA + EA

Calculation of destination memory location:
EA_E = (DI) BA_E = (ES) x 16_{10} MA_E = BA_E + EA_E

(MAE) ← (MA)

If DF = 1, then (SI) ← (SI) – 1 and (DI) = (DI) - 1
If DF = 0, then (SI) ← (SI) +1 and (DI) = (DI) + 1

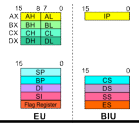
35

8086 Microprocessor

Addressing Modes

Group III : Addressing modes for I/O ports

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



These addressing modes are used to access data from standard I/O mapped devices or ports.

In direct port addressing mode, an 8-bit port address is directly specified in the instruction.

Example: IN AL, 09H

Operations: $PORT_{addr}$ = 09_H
(AL) ← (PORT)

Content of port with address 09_H is moved to AL register

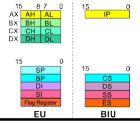
36

8086 Microprocessor

Addressing Modes

Group IV : Relative Addressing mode

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: **JZ 0AH**

Operations:

$000A_H \leftarrow 0A_H$ (sign extend)

If ZF = 1, then

$EA = (IP) + 000A_H$
$BA = (CS) \times 16_{10}$
$MA = BA + EA$

If ZF = 1, then the program control jumps to new address calculated above.

If ZF = 0, then next instruction of the program is executed.

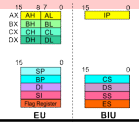
37

8086 Microprocessor

Addressing Modes

Group IV : Implied Addressing mode

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: **CLC**

This clears the carry flag to zero.

38

INSTRUCTION SET

Instruction Set

8086 supports 6 types of instructions.

1. **Data Transfer Instructions**
2. **Arithmetic Instructions**
3. **Logical Instructions**
4. **String manipulation Instructions**
5. **Process Control Instructions**
6. **Control Transfer Instructions**

Instruction Set

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

Instruction Set

1. Data Transfer Instructions

Mnemonics: MOV, XCHG, PUSH, POP, IN, OUT ...

MOV reg2/ mem, reg1/ mem	
MOV reg2, reg1	(reg2) ← (reg1)
MOV mem, reg1	(mem) ← (reg1)
MOV reg2, mem	(reg2) ← (mem)
MOV reg/ mem, data	
MOV reg, data	(reg) ← data
MOV mem, data	(mem) ← data

XCHG reg2/ mem, reg1	
XCHG reg2, reg1	(reg2) ↔ (reg1)
XCHG mem, reg1	(mem) ↔ (reg1)

8086 Microprocessor

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

PUSH reg16/ mem	
PUSH reg16	(SP) ← (SP) – 2 MA_S = (SS) x 16_{10} + SP (MA_S ; MA_S + 1) ← (reg16)
PUSH mem	(SP) ← (SP) – 2 MA_S = (SS) x 16_{10} + SP (MA_S ; MA_S + 1) ← (mem)
POP reg16/ mem	
POP reg16	MA_S = (SS) x 16_{10} + SP (reg16) ← (MA_S ; MA_S + 1) (SP) ← (SP) + 2
POP mem	MA_S = (SS) x 16_{10} + SP (mem) ← (MA_S ; MA_S + 1) (SP) ← (SP) + 2

8086 Microprocessor

Instruction Set

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

IN A, [DX]		**OUT [DX], A**	
IN AL, [DX]	$PORT_{addr}$ = (DX) (AL) ← (PORT)	OUT [DX], AL	$PORT_{addr}$ = (DX) (PORT) ← (AL)
IN AX, [DX]	$PORT_{addr}$ = (DX) (AX) ← (PORT)	OUT [DX], AX	$PORT_{addr}$ = (DX) (PORT) ← (AX)
IN A, addr8		**OUT addr8, A**	
IN AL, addr8	(AL) ← (addr8)	OUT addr8, AL	(addr8) ← (AL)
IN AX, addr8	(AX) ← (addr8)	OUT addr8, AX	(addr8) ← (AX)

44

8086 Microprocessor

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

ADD reg2/ mem, reg1/mem	
ADC reg2, reg1	(reg2) ← (reg1) + (reg2)
ADC reg2, mem	(reg2) ← (reg2) + (mem)
ADC mem, reg1	(mem) ← (mem)+(reg1)
ADD reg/mem, data	
ADD reg, data	(reg) ← (reg)+ data
ADD mem, data	(mem) ← (mem)+data
ADD A, data	
ADD AL, data8	(AL) ← (AL) + data8
ADD AX, data16	(AX) ← (AX) +data16

45

8086 Microprocessor

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

ADC reg2/ mem, reg1/mem	
ADC reg2, reg1	(reg2) ← (reg1) + (reg2)+CF
ADC reg2, mem	(reg2) ← (reg2) + (mem)+CF
ADC mem, reg1	(mem) ← (mem)+(reg1)+CF
ADC reg/mem, data	
ADC reg, data	(reg) ← (reg)+ data+CF
ADC mem, data	(mem) ← (mem)+data+CF
ADC A, data	
ADD AL, data8	(AL) ← (AL) + data8+CF
ADD AX, data16	(AX) ← (AX) +data16+CF

46

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SUB reg2/ mem, reg1/mem	
SUB reg2, reg1	(reg2) ← (reg1) - (reg2)
SUB reg2, mem	(reg2) ← (reg2) - (mem)
SUB mem, reg1	(mem) ← (mem) - (reg1)
SUB reg/mem, data	
SUB reg, data	(reg) ← (reg) - data
SUB mem, data	(mem) ← (mem) - data
SUB A, data	
SUB AL, data8	(AL) ← (AL) - data8
SUB AX, data16	(AX) ← (AX) - data16

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

SBB reg2/ mem, reg1/mem	
SBB reg2, reg1	(reg2) ← (reg1) - (reg2) - CF
SBB reg2, mem	(reg2) ← (reg2) - (mem)- CF
SBB mem, reg1	(mem) ← (mem) - (reg1) –CF
SBB reg/mem, data	
SBB reg, data	(reg) ← (reg) – data - CF
SBB mem, data	(mem) ← (mem) - data - CF
SBB A, data	
SBB AL, data8	(AL) ← (AL) - data8 - CF
SBB AX, data16	(AX) ← (AX) - data16 - CF

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

INC reg/ mem	
INC reg8	**(reg8) ← (reg8) + 1**
INC reg16	**(reg16) ← (reg16) + 1**
INC mem	**(mem) ← (mem) + 1**
DEC reg/ mem	
DEC reg8	**(reg8) ← (reg8) - 1**
DEC reg16	**(reg16) ← (reg16) - 1**
DEC mem	**(mem) ← (mem) - 1**

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…

MUL reg/ mem	
MUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
MUL mem	**For byte : (AX) ← (AL) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**
IMUL reg/ mem	
IMUL reg	**For byte : (AX) ← (AL) x (reg8)** **For word : (DX)(AX) ← (AX) x (reg16)**
IMUL mem	**For byte : (AX) ← (AX) x (mem8)** **For word : (DX)(AX) ← (AX) x (mem16)**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

DIV reg/ mem	
DIV reg	For 16-bit :- 8-bit : (AL) ← (AX) :- (reg8) Quotient (AH) ← (AX) MOD(reg8) Remainder For 32-bit :- 16-bit : (AX) ← (DX)(AX) :- (reg16) Quotient (DX) ← (DX)(AX) MOD(reg16) Remainder
DIV mem	For 16-bit :- 8-bit : (AL) ← (AX) :- (mem8) Quotient (AH) ← (AX) MOD(mem8) Remainder For 32-bit :- 16-bit : (AX) ← (DX)(AX) :- (mem16) Quotient (DX) ← (DX)(AX) MOD(mem16) Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP…**

IDIV reg/ mem	
IDIV reg	For 16-bit :- 8-bit : (AL) ← (AX) :- (reg8) Quotient (AH) ← (AX) MOD(reg8) Remainder For 32-bit :- 16-bit : (AX) ← (DX)(AX) :- (reg16) Quotient (DX) ← (DX)(AX) MOD(reg16) Remainder
IDIV mem	For 16-bit :- 8-bit : (AL) ← (AX) :- (mem8) Quotient (AH) ← (AX) MOD(mem8) Remainder For 32-bit :- 16-bit : (AX) ← (DX)(AX) :- (mem16) Quotient (DX) ← (DX)(AX) MOD(mem16) Remainder

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg2/mem, reg1/ mem	
CMP reg2, reg1	**Modify flags ← (reg2) – (reg1)** **If (reg2) > (reg1) then CF=0, ZF=0, SF=0** **If (reg2) < (reg1) then CF=1, ZF=0, SF=1** **If (reg2) = (reg1) then CF=0, ZF=1, SF=0**
CMP reg2, mem	**Modify flags ← (reg2) – (mem)** **If (reg2) > (mem) then CF=0, ZF=0, SF=0** **If (reg2) < (mem) then CF=1, ZF=0, SF=1** **If (reg2) = (mem) then CF=0, ZF=1, SF=0**
CMP mem, reg1	**Modify flags ← (mem) – (reg1)** **If (mem) > (reg1) then CF=0, ZF=0, SF=0** **If (mem) < (reg1) then CF=1, ZF=0, SF=1** **If (mem) = (reg1) then CF=0, ZF=1, SF=0**

Instruction Set

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg/mem, data	
CMP reg, data	**Modify flags ← (reg) – (data)** **If (reg) > data then CF=0, ZF=0, SF=0** **If (reg) < data then CF=1, ZF=0, SF=1** **If (reg) = data then CF=0, ZF=1, SF=0**
CMP mem, data	**Modify flags ← (mem) – (mem)** **If (mem) > data then CF=0, ZF=0, SF=0** **If (mem) < data then CF=1, ZF=0, SF=1** **If (mem) = data then CF=0, ZF=1, SF=0**

8086 Microprocessor

Instruction Set

2. Arithmetic Instructions

Mnemonics: ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...

CMP A, data	
CMP AL, data8	**Modify flags ← (AL) – data8** **If (AL) > data8 then CF=0, ZF=0, SF=0** **If (AL) < data8 then CF=1, ZF=0, SF=1** **If (AL) = data8 then CF=0, ZF=1, SF=0**
CMP AX, data16	**Modify flags ← (AX) – data16** **If (AX) > data16 then CF=0, ZF=0, SF=0** **If (mem) < data16 then CF=1, ZF=0, SF=1** **If (mem) = data16 then CF=0, ZF=1, SF=0**

8086 Microprocessor

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

AND A, data	
AND AL, data8	(AL) ← (AL) & data8
AND AX, data16	(AX) ← (AX) & data16

AND reg/mem, data	
AND reg, data	(reg) ← (reg) & data
AND mem, data	(mem) ← (mem) & data

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

OR reg2/mem, reg1/mem	
OR reg2, reg1	(reg2) ← (reg2) | (reg1)
OR reg2, mem	(reg2) ← (reg2) | (mem)
OR mem, reg1	(mem) ← (mem) | (reg1)

OR reg/mem, data	
OR reg, data	(reg) ← (reg) | data
OR mem, data	(mem) ← (mem) | data

OR A, data	
OR AL, data8	(AL) ← (AL) | data8
OR AX, data16	(AX) ← (AX) | data16

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	(reg2) ← (reg2) ^ (reg1)
XOR reg2, mem	(reg2) ← (reg2) ^ (mem)
XOR mem, reg1	(mem) ← (mem) ^ (reg1)

XOR reg/mem, data	
XOR reg, data	(reg) ← (reg) ^ data
XOR mem, data	(mem) ← (mem) ^ data

XOR A, data	
XOR AL, data8	(AL) ← (AL) ^ data8
XOR AX, data16	(AX) ← (AX) ^ data16

8086 Microprocessor

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

TEST reg2/mem, reg1/mem	
TEST reg2, reg1	Modify flags ← (reg2) & (reg1)
TEST reg2, mem	Modify flags ← (reg2) & (mem)
TEST mem, reg1	Modify flags ← (mem) & (reg1)
TEST reg/mem, data	
TEST reg, data	Modify flags ← (reg) & data
TEST mem, data	Modify flags ← (mem) & data
TEST A, data	
TEST AL, data8	Modify flags ← (AL) & data8
TEST AX, data16	Modify flags ← (AX) & data16

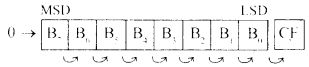
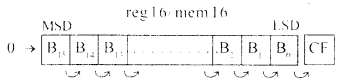
59

8086 Microprocessor

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

SHR reg/mem	$CF \leftarrow B_{LSD}$; $B_n \leftarrow B_{n+1}$; $B_{MSD} \leftarrow 0$
SHR reg	reg8 / mem8
i) SHR reg, 1	
ii) SHR reg, CL	
SHR mem	reg16 / mem16
i) SHR mem, 1	
ii) SHR mem, CL	

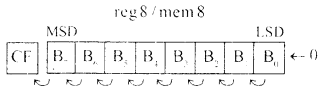
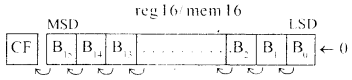
60

8086 Microprocessor

Instruction Set

3. Logical Instructions

Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

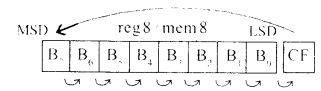
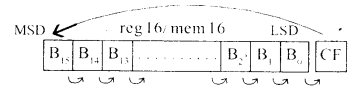
SHL reg/mem or SAL reg/mem SHL reg or SAL reg i) SHL reg, 1 or SAL reg, 1 ii) SHL reg, CL or SAL reg, CL SHL mem or SAL mem i) SHL mem, 1 or SAL mem, 1 ii) SHL mem, CL or SAL mem, CL	$CF \leftarrow B_{MSD}$; $B_{n+1} \leftarrow B_n$; $B_{LSD} \leftarrow 0$  

8086 Microprocessor

Instruction Set

3. Logical Instructions

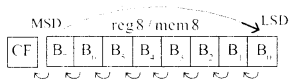
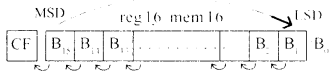
Mnemonics: AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...

RCR reg/mem RCR reg i) RCR reg, 1 ii) RCR reg, CL RCR mem i) RCR mem, 1 ii) RCR mem, CL	$B_n \leftarrow B_{n-1}$; $B_{MSD} \leftarrow CF$; $CF \leftarrow B_{LSD}$  

Instruction Set

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

ROL reg/mem	$B_{n+1} \leftarrow B_n : CF \leftarrow B_{MSD} : B_{LSD} \leftarrow B_{MSD}$
ROL reg	
i) ROL reg, 1	
ii) ROL reg, CL	
ROL mem	
i) ROL mem, 1	
ii) ROL mem, CL	

Instruction Set

4. String Manipulation Instructions

- **String : Sequence of bytes or words**
- **8086 instruction set includes instruction for string movement, comparison, scan, load and store.**
- **REP instruction prefix : used to repeat execution of string instructions**
- **String instructions end with S or SB or SW.**
  S represents string, SB string byte and SW string word.
- **Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.**
- **Depending on the status of DF, SI and DI registers are automatically updated.**
- **DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.**
- **DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

REP	
REPZ/ REPE **(Repeat CMPS or SCAS until ZF = 0)**	**While CX ≠ 0 and ZF = 1, repeat execution of string instruction and (CX) ← (CX) – 1**
REPNZ/ REPNE **(Repeat CMPS or SCAS until ZF = 1)**	**While CX ≠ 0 and ZF = 0, repeat execution of string instruction and (CX) ← (CX) - 1**

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

MOVS	
MOVSB	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E) ← (MA)** **If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1** **If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1**
MOVSW	**MA = (DS) x 16_{10} + (SI)** **MA_E = (ES) x 16_{10} + (DI)** **(MA_E ; MA_E + 1) ← (MA; MA + 1)** **If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2** **If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2**

8086 Microprocessor

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Compare two string byte or string word

CMPS	

CMPSB	MA = (DS) x 16_{10} + (SI)
	MA_E = (ES) x 16_{10} + (DI)
	
	Modify flags ← (MA) - (MA_E)
	
	If (MA) > (MA_E), then CF = 0; ZF = 0; SF = 0
	If (MA) < (MA_E), then CF = 1; ZF = 0; SF = 1
	If (MA) = (MA_E), then CF = 0; ZF = 1; SF = 0

CMPSW	<u>For byte operation</u> If DF = 0, then (DI) ← (DI) + 1; (SI) ← (SI) + 1 If DF = 1, then (DI) ← (DI) - 1; (SI) ← (SI) - 1 <u>For word operation</u> If DF = 0, then (DI) ← (DI) + 2; (SI) ← (SI) + 2 If DF = 1, then (DI) ← (DI) - 2; (SI) ← (SI) - 2

67

8086 Microprocessor

Instruction Set

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Scan (compare) a string byte or word with accumulator

SCAS	

SCASB	MA_E = (ES) x 16_{10} + (DI)
	Modify flags ← (AL) - (MA_E)
	
	If (AL) > (MA_E), then CF = 0; ZF = 0; SF = 0
	If (AL) < (MA_E), then CF = 1; ZF = 0; SF = 1
	If (AL) = (MA_E), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) ← (DI) + 1
	If DF = 1, then (DI) ← (DI) – 1

SCASW	MA_E = (ES) x 16_{10} + (DI)
	Modify flags ← (AL) - (MA_E)
	
	If (AX) > (MA_E ; MA_E + 1), then CF = 0; ZF = 0; SF = 0
	If (AX) < (MA_E ; MA_E + 1), then CF = 1; ZF = 0; SF = 1
	If (AX) = (MA_E ; MA_E + 1), then CF = 0; ZF = 1; SF = 0
	
	If DF = 0, then (DI) ← (DI) + 2
	If DF = 1, then (DI) ← (DI) – 2

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Load string byte in to AL or string word in to AX

LODS	
LODSB	MA = (DS) x 16_{10} + (SI) (AL) ← (MA) If DF = 0, then (SI) ← (SI) + 1 If DF = 1, then (SI) ← (SI) – 1
LODSW	MA = (DS) x 16_{10} + (SI) (AX) ← (MA ; MA + 1) If DF = 0, then (SI) ← (SI) + 2 If DF = 1, then (SI) ← (SI) – 2

Instruction Set

4. String Manipulation Instructions

Mnemonics: REP, MOVS, CMPS, SCAS, LODS, STOS

Store byte from AL or word from AX in to string

STOS	
STOSB	MA_E = (ES) x 16_{10} + (DI) (MA_E) ← (AL) If DF = 0, then (DI) ← (DI) + 1 If DF = 1, then (DI) ← (DI) – 1
STOSW	MA_E = (ES) x 16_{10} + (DI) (MA_E ; MA_E + 1) ← (AX) If DF = 0, then (DI) ← (DI) + 2 If DF = 1, then (DI) ← (DI) – 2

Instruction Set

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF ← 1
CLC	Clear CF ← 0
CMC	Complement carry CF ← CF/
STD	Set direction flag DF ← 1
CLD	Clear direction flag DF ← 0
STI	Set interrupt enable flag IF ← 1
CLI	Clear interrupt enable flag IF ← 0
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem/ reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

Instruction Set

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

- 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

Instruction Set

6. Control Transfer Instructions

- 8086 signed conditional branch instructions
- 8086 unsigned conditional branch instructions

- Checks flags
- If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP

8086 Microprocessor

Instruction Set

6. Control Transfer Instructions

- 8086 signed conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

- 8086 unsigned conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

Instruction Set

6. Control Transfer Instructions

- 8086 conditional branch instructions affecting individual flags

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

Assembler directives

Assembler Directives

- Instructions to the Assembler regarding the program being executed.
- Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.
- Also called 'pseudo instructions'
- Used to :
  - specify the start and end of a program
  - attach value to variables
  - allocate storage locations to input/ output data
  - define start and end of segments, procedures, macros etc..

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Define Byte
- Define a byte type (8-bit) variable
- Reserves specific amount of memory locations to each variable
- Range : 00_H – FF_H for unsigned value;
  00_H – $7F_H$ for positive value and
  80_H – FF_H for negative value
- General form : variable DB value/ values

Example:

LIST DB 7FH, 42H, 35H

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

Assembler Directives

DB
DW
SEGMENT
ENDS
ASSUME
ORG
END
EVEN
EQU
PROC
FAR
NEAR
ENDP
SHORT
MACRO
ENDM

- Define Word
- Define a word type (16-bit) variable
- Reserves two consecutive memory locations to each variable
- Range : 0000_H – $FFFF_H$ for unsigned value;
  0000_H – $7FFF_H$ for positive value and
  8000_H – $FFFF_H$ for negative value
- General form : variable DW value/ values

Example:

ALIST DW 6512H, 0F251H, 0CDE2H

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assembler Directives

DB
DW
SEGMENT
ENDS
ASSUME
ORG
END
EVEN
EQU
PROC
FAR
NEAR
ENDP
SHORT
MACRO
ENDM

- SEGMENT : Used to indicate the beginning of a code/ data/ stack segment
- ENDS : Used to indicate the end of a code/ data/ stack segment
- General form:

```
Segnam SEGMENT
    ...
    ...
    ...
    ...
    ...
    ...
Segnam ENDS
```

Program code
or
Data Defining Statements

User defined name of the segment

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Informs the assembler the name of the program/ data segment that should be used for a specific segment.
- General form:

ASSUME segreg : segnam, .. , segreg : segnam

Segment Register

User defined name of the segment

Example:

ASSUME CS: ACODE, DS:ADATA	Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- ORG (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- END is used to terminate a program; statements after END will be ignored
- EVEN : Informs the assembler to store program/ data segment starting from an even address
- EQU (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000_H
LOOP EQU 10FEH	Value of variable LOOP is $10FE_H$
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200_H and that of B will be 1202_H and 1203_H.

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- General form

```
procname PROC[NEAR/ FAR]
        ...
        ...
        ...
        RET

procname ENDP
```

Program statements of the procedure

Last statement of the procedure

User defined name of the procedure

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR ... ... RET ADD64 ENDP	The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return
CONVERT PROC FAR ... ... RET CONVERT ENDP	The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- Reserves one memory location for 8-bit signed displacement in jump instructions

Example:

JMP SHORT AHEAD	The directive will reserve one memory location for 8-bit displacement named AHEAD

Assembler Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **MACRO** Indicate the beginning of a macro
- **ENDM** End of a macro
- General form:

```
macroname MACRO[Arg1, Arg2 ...]
        ...
        ...
        ...
macroname ENDM
```

Program statements in the macro

User defined name of the macro

Interfacing memory and i/o ports

Memory

Memory

Store Programs and Data

Processor Memory

- Registers inside a microcomputer
- Store data and results temporarily
- No speed disparity
- Cost ↑

Primary or Main Memory

- Storage area which can be directly accessed by microprocessor
- Store programs and data prior to execution
- Should not have speed disparity with processor ⇒ Semi Conductor memories using CMOS technology
- ROM, EPROM, Static RAM, DRAM

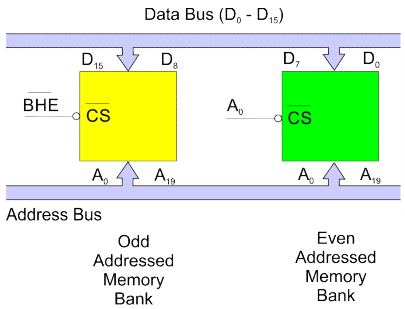
Secondary Memory

- Storage media comprising of slow devices such as magnetic tapes and disks
- Hold large data files and programs: Operating system, compilers, databases, permanent programs etc.

8086 Microprocessor

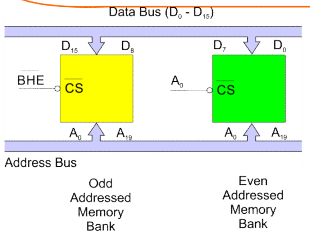
Memory organization in 8086

- **Memory IC's : Byte oriented**
- **8086 : 16-bit**
- **Word : Stored by two consecutive memory locations; for LSB and MSB**
- **Address of word : Address of LSB**
- **Bank 0 : $A_0 = 0$ ⇒ Even addressed memory bank**
- **Bank 1 : $\overline{BHE} = 0$ ⇒ Odd addressed memory bank**



8086 Microprocessor

Memory organization in 8086



	Operation	$\overline{BHE}$	A_0	Data Lines Used
1	Read/ Write byte at an even address	1	0	$D_7 - D_0$
2	Read/ Write byte at an odd address	0	1	$D_{15} - D_8$
3	Read/ Write word at an even address	0	0	$D_{15} - D_0$
4	Read/ Write word at an odd address	0	1	$D_{15} - D_0$ in first operation byte from odd bank is transferred
		1	0	$D_7 - D_0$ in first operation byte from odd bank is transferred

Memory organization in 8086

- Available memory space = EPROM + RAM
- Allot equal address space in odd and even bank for both EPROM and RAM
- Can be implemented in two IC's (one for even and other for odd) or in multiple IC's

Interfacing SRAM and EPROM

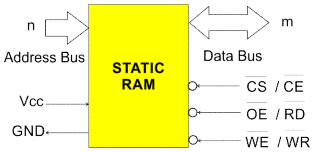
- Memory interface ⇒ Read from and write in to a set of semiconductor memory IC chip
- EPROM ⇒ Read operations
- RAM ⇒ Read and Write

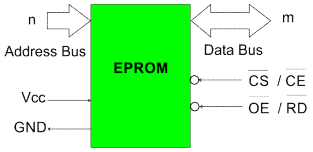
In order to perform read/ write operations,

- Memory access time < read / write time of the processor
- Chip Select (CS) signal has to be generated
- Control signals for read / write operations
- Allot address for each memory location

Interfacing SRAM and EPROM

- **Typical Semiconductor IC Chip**



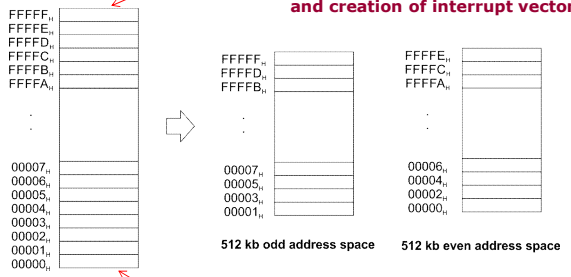


No of Address pins	Memory capacity			Range of address in hexa
	In Decimal	In kilo	In hexa	
20	2^{20}= 10,48,576	1024 k = 1M	100000	00000 to FFFFF

Interfacing SRAM and EPROM

- **Memory map of 8086**

EPROM's are mapped at $FFFFF_H$
⇒ Facilitate automatic execution of monitor programs and creation of interrupt vector table



RAM are mapped at the beginning; 00000H is allotted to RAM

Interfacing SRAM and EPROM

Monitor Programs

- ⇒ Programing 8279 for keyboard scanning and display refreshing
- ⇒ Programming peripheral IC's 8259, 8257, 8255, 8251, 8254 etc
- ⇒ Initialization of stack
- ⇒ Display a message on display (output)
- ⇒ Initializing interrupt vector table

Note :

8279	Programmable keyboard/ display controller
8257	DMA controller
8259	Programmable interrupt controller
8255	Programmable peripheral interface

Interfacing I/O and peripheral devices

I/O devices

- ⇒ For communication between microprocessor and outside world
- ⇒ Keyboards, CRT displays, Printers, Compact Discs etc.
- ⇒ Microprocessor ↔ Ports / Buffer IC's (interface circuitry) ↔ I/ O devices
- ⇒ Data transfer types
  - Programmed I/ O
    Data transfer is accomplished through an I/O port controlled by software
    - Memory mapped
    - I/O mapped
  - Interrupt driven I/ O
    I/O device interrupts the processor and initiate data transfer
  - Direct memory access
    Data transfer is achieved by bypassing the microprocessor

8086 and 8088 comparison

Memory mapping	I/O mapping
20 bit address are provided for I/O devices	8-bit or 16-bit addresses are provided for I/O devices
The I/O ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transmission between I/O device and processor	Only IN and OUT instructions can be used for data transfer between I/O device and processor
Data can be moved from any register to ports and vice versa	Data transfer takes place only between accumulator and ports
When memory mapping is used for I/O devices, full memory address space cannot be used for addressing memory. ⇒ Useful only for small systems where memory requirement is less	Full memory space can be used for addressing memory. ⇒ Suitable for systems which require large memory capacity
For accessing the memory mapped devices, the processor executes memory read or write cycle. ⇒ M / $\overline{IO}$ is asserted high	For accessing the I/O mapped devices, the processor executes I/O read or write cycle. ⇒ M / $\overline{IO}$ is asserted low

8086 and 8088 comparison

8086 and 8088 comparison

8086	8088
Similar EU and Instruction set ; dissimilar BIU	
16-bit Data bus lines obtained by demultiplexing $AD_0 - AD_{15}$	8-bit Data bus lines obtained by demultiplexing $AD_0 - AD_7$
20-bit address bus	8-bit address bus
Two banks of memory each of 512 kb	Single memory bank
6-bit instruction queue	4-bit instruction queue
Clock speeds: 5 / 8 / 10 MHz	5 / 8 MHz
In MIN mode, pin 28 is assigned the signal M / $\overline{IO}$	In MIN mode, pin 28 is assigned the signal IO / $\overline{M}$
To access higher byte, $\overline{BHE}$ signal is used	No such signal required, since the data width is only 1-byte

8087 Coprocessor

Co-processor – Intel 8087

Multiprocessor system

- A microprocessor system comprising of two or more processors
- Distributed processing: Entire task is divided in to subtasks

Advantages

- Better system throughput by having more than one processor
- Each processor have a local bus to access local memory or I/O devices so that a greater degree of parallel processing can be achieved
- System structure is more flexible.
  One can easily add or remove modules to change the system configuration without affecting the other modules in the system

Co-processor – Intel 8087

8087 coprocessor

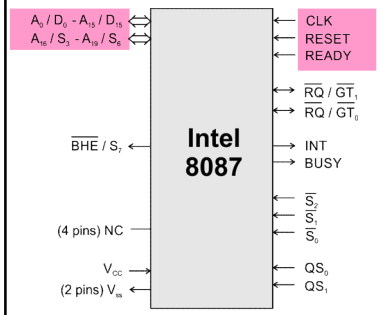
- Specially designed to take care of mathematical calculations involving integer and floating point data
- "Math coprocessor" or "Numeric Data Processor (NDP)"
- Works in parallel with a 8086 in the maximum mode

Features

1) Can operate on data of the integer, decimal and real types with lengths ranging from 2 to 10 bytes
2) Instruction set involves square root, exponential, tangent etc. in addition to addition, subtraction, multiplication and division.
3) High performance numeric data processor ⇒ it can multiply two 64-bit real numbers in about 27μs and calculate square root in about 36 μs
4) Follows IEEE floating point standard
5) It is multi bus compatible

8086 Microprocessor

Co-processor – Intel 8087

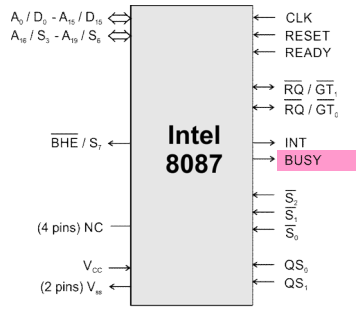


- **16 multiplexed address / data pins and 4 multiplexed address / status pins**
- **Hence it can have 16-bit external data bus and 20-bit external address bus like 8086**
- **Processor clock, ready and reset signals are applied as clock, ready and reset signals for coprocessor**

103

8086 Microprocessor

Co-processor – Intel 8087

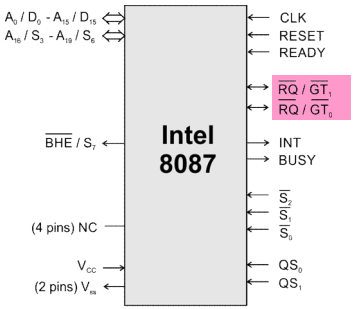


BUSY

- **BUSY signal from 8087 is connected to the $\overline{TEST}$ input of 8086**
- **If the 8086 needs the result of some computation that the 8087 is doing before it can execute the next instruction in the program, a user can tell 8086 with a WAIT instruction to keep looking at its $\overline{TEST}$ pin until it finds the pin low**
- **A low on the BUSY output indicates that the 8087 has completed the computation**

104

Co-processor – Intel 8087



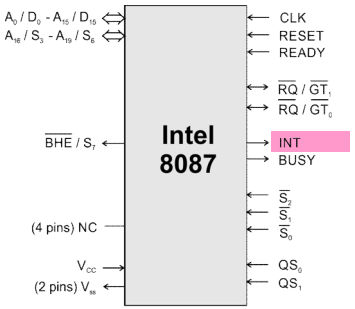
$\overline{RQ}$ / $\overline{GT_0}$

- **The request / grant signal from the 8087 is usually connected to the request / grant ($\overline{RQ}$ / $\overline{GT_0}$ or $\overline{RQ}$ / $\overline{GT_1}$) pin of the 8086**

$\overline{RQ}$ / $\overline{GT_1}$

- **The request / grant signal from the 8087 is usually connected to the request / grant pin of the independent processor such as 8089**

Co-processor – Intel 8087

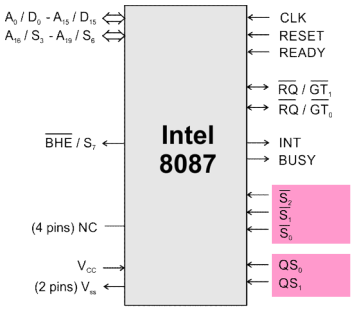


INT

- **The interrupt pin is connected to the interrupt management logic.**
- **The 8087 can interrupt the 8086 through this interrupt management logic at the time error condition exists**

8086 Microprocessor

Co-processor – Intel 8087



$\overline{S_0}$- $\overline{S_2}$

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Status
1	0	0	Unused
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive

QS_0 – QS_1

QS_0	QS_1	Status
0	0	No operation
0	1	First byte of opcode from queue
1	0	Queue empty
1	1	Subsequent byte of opcode from queue

107

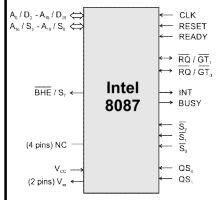
8086 Microprocessor

Co-processor – Intel 8087

- **8087 instructions are inserted in the 8086 program**

- **8086 and 8087 reads instruction bytes and puts them in the respective queues**
- **NOP**
- **8087 instructions have 11011 as the MSB of their first code byte**

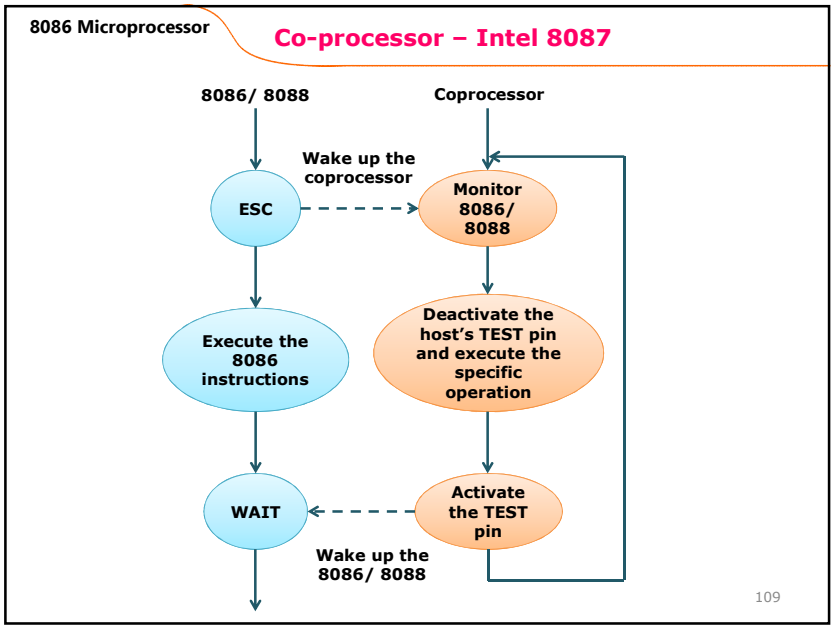
- **8087 keeps track for ESC instruction by monitoring $\overline{S_2}$ - $\overline{S_0}$ and AD_0 – AD_{15} of 8086.**
- **Also keeps track of QS_0 – QS_1.**
- **Q status 00; does nothing**
- **Q status 01; 8087 compares the five MSB bits with 11011**
- **If there is a match, then the ESC instruction is fetched and executed by 8087**
- **If there is error during decoding an ESC instruction, 8087 sends an interrupt request**



- **Memory read/ write**
- **Additional words : $\overline{RQ}$ - $\overline{GT_0}$**
- **8087 BUSY pin high**
- **$\overline{TEST}$**
- **WAIT**

108

Ref: Microprocessor, Atul P. Godse, Deepali A. Gode, Technical publications, Chap 11


8086 Microprocessor
Co-processor – Intel 8087
8086/ 8088
Coprocessor
Wake up the coprocessor
ESC
Monitor 8086/ 8088
Execute the 8086 instructions
Deactivate the host's TEST pin and execute the specific operation
WAIT
Activate the TEST pin
Wake up the 8086/ 8088
109