Microprocessor Systems and Interfacing

EEE 342

Nesruminallah

nesruminallah@cuilahore.edu.pk

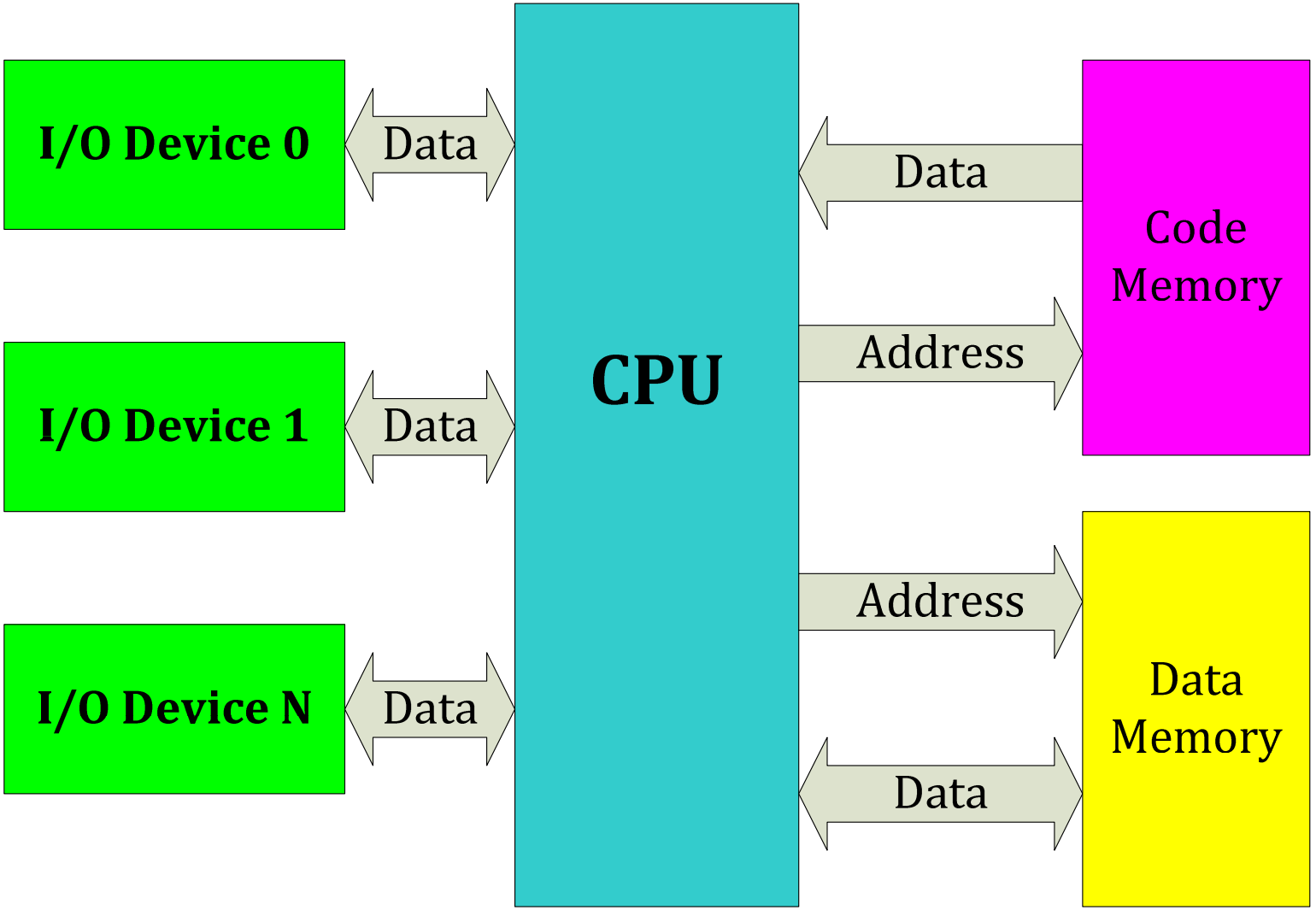
Microprocessor/Microcontroller Architecture

CLO	Bloom Taxonomy	Specific Outcome
CLO1	C2	Comprehend the theoretical knowledge of computer architecture using hardware architectures
CLO1	C2	Comprehend the theoretical knowledge of Intel 8086/88 programmers model using SISE architecture
CLO1	C2	Comprehend the theoretical knowledge of addressing modes using 8086/88 programmers model
CLO1	C2	Comprehend the theoretical knowledge of assembly language using 8086/88 programmers model and addressing modes
CLO1	C5	Write and manipulate the Intel-assembly code using the knowledge of programmer model, addressing mode and assembly language programming concepts

Microprocessor/Microcontroller Architecture

- Outline
  - Introduction to MPU System
  - Microprocessors and Microcontrollers
  - Microprocessor Architecture
  - Addressing Mode
  - Address calculation
  - Assembly Language Programming

Introduction



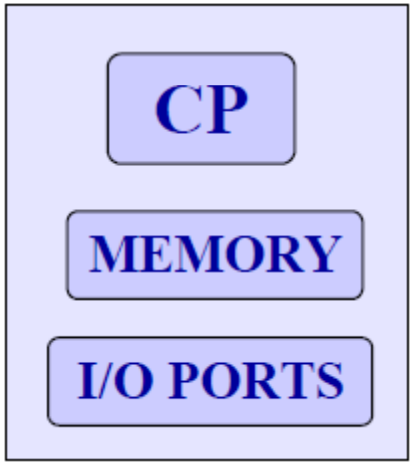
Microprocessor Architecture

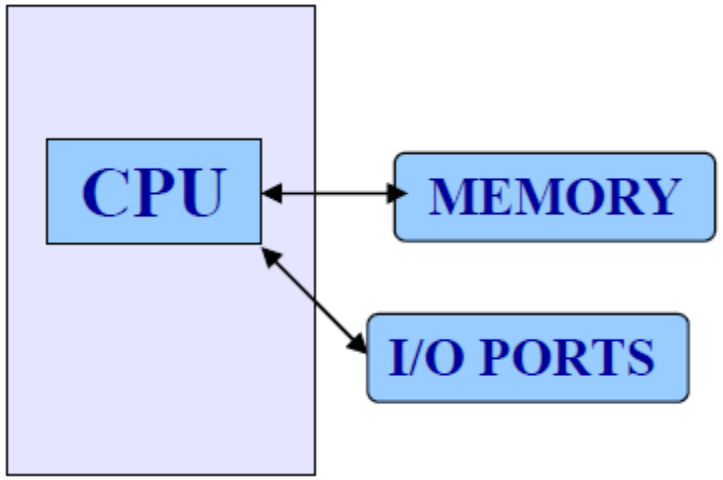
MICRO CONTROLLER

- It is a single chip
- Consists Memory, I/o ports

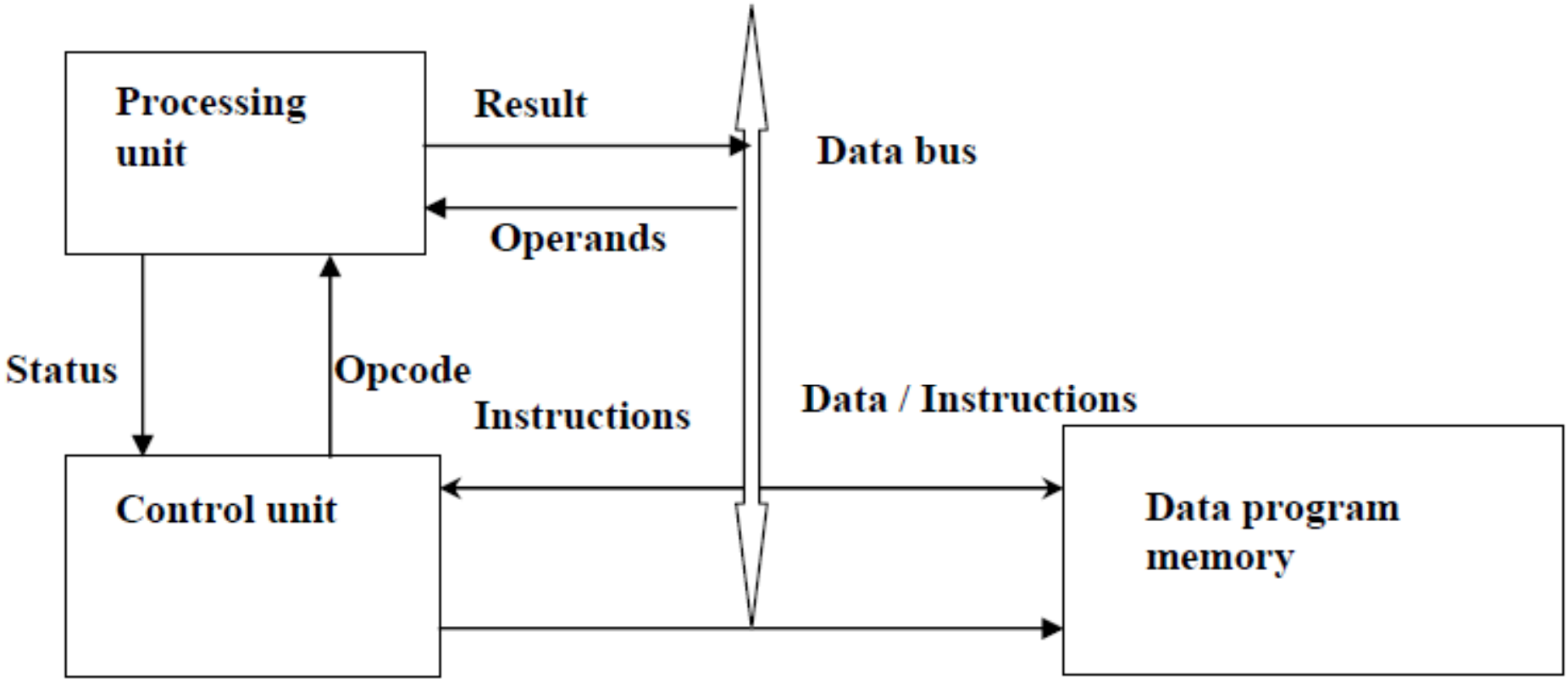
MICRO PROCESSER

- It is a CPU
- Memory, I/O Ports to be connected externally

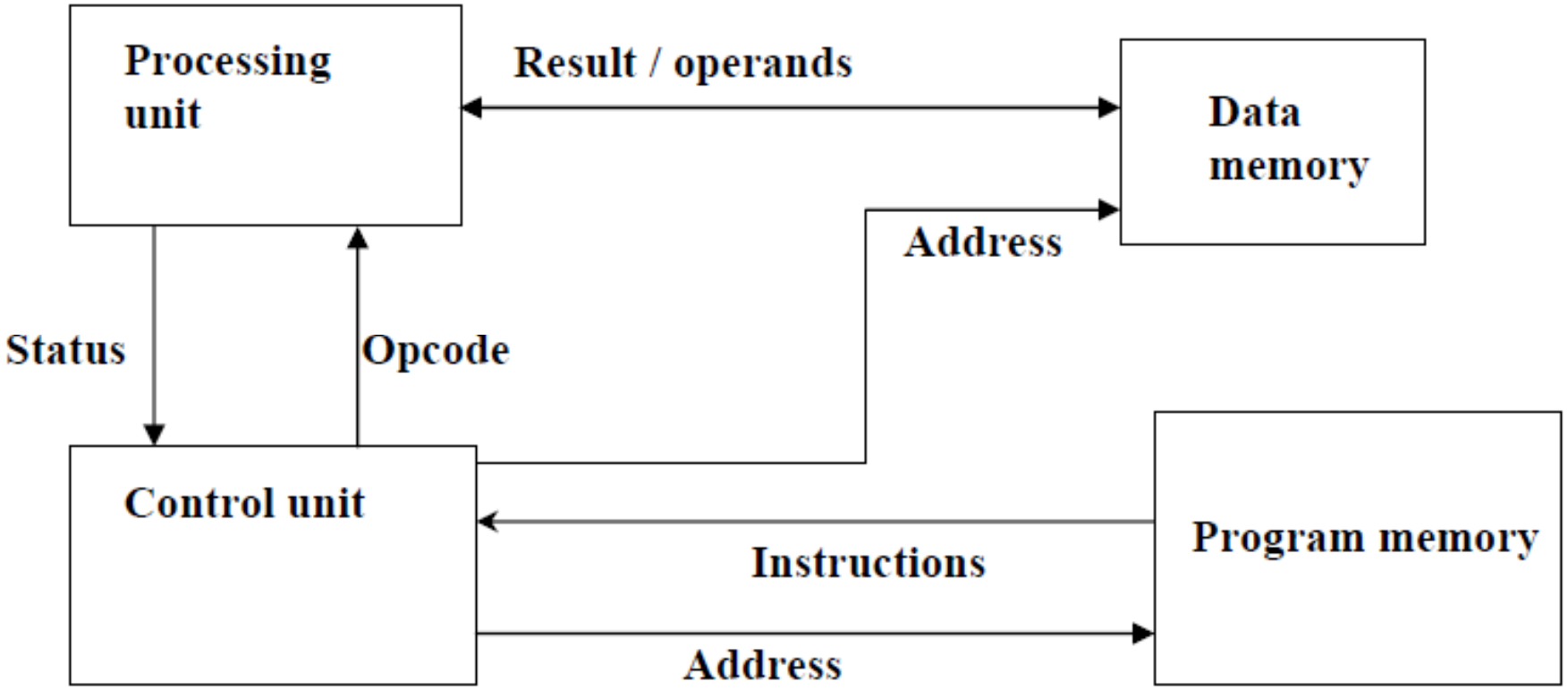




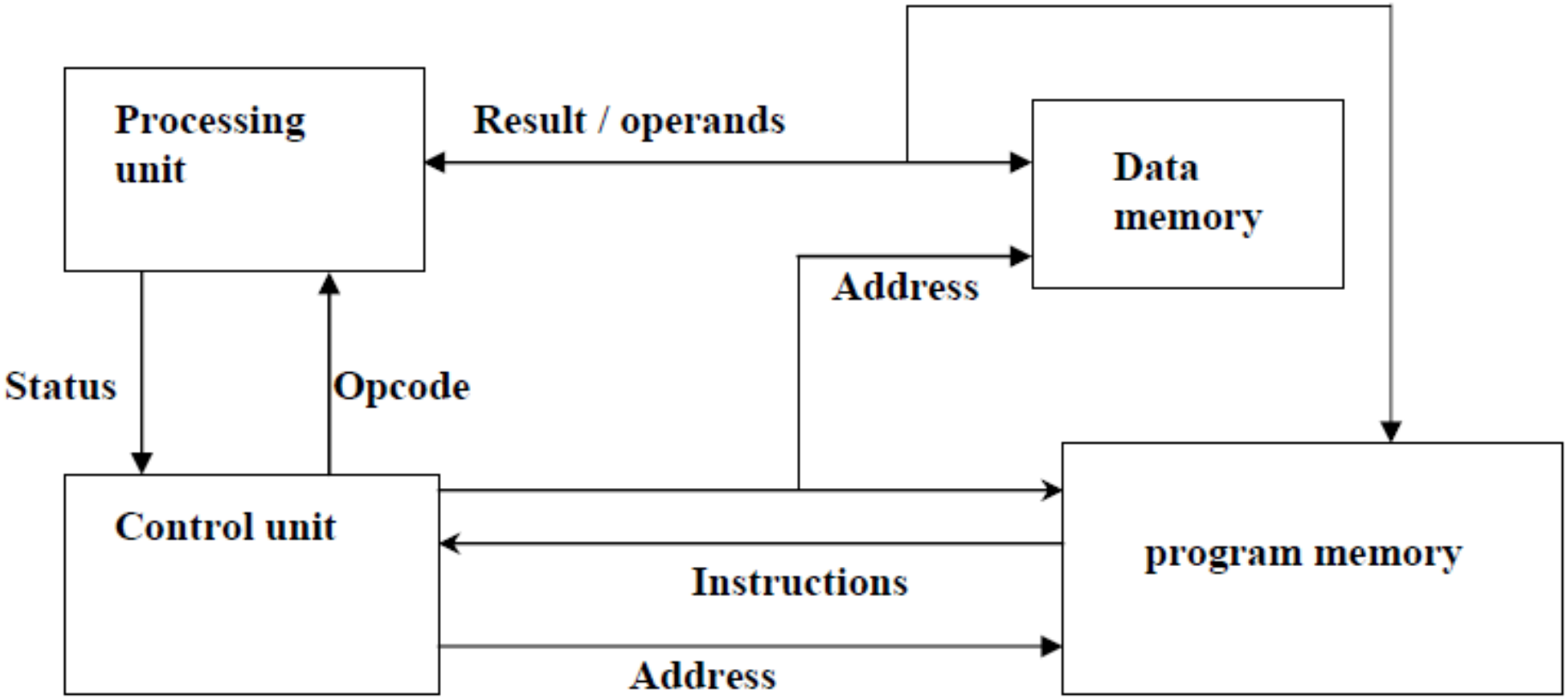
Von Neuman Architecture



Harvard Architecture



Modified Harvard Architecture



Addressing Mode

- Addressing Mode
  - The term addressing modes refers to the way in which the operand of an instruction is specified.
  - Addressing modes of 8086
    - Register
    - Immediate
    - Direct
    - Register indirect
    - Based relative
    - Indexed relative
    - Based indexed relative

Addressing Mode

- Register
  - Involves the use of registers
  - Memory is not accessed, so faster
  - Source and destination registers must match in size.
    - MOV BX,DX
    - MOV ES,AX
    - ADD AL,BH
    - MOV AL,CX ;not possible

Addressing Mode

- Immediate
  - Source operand is a constant
  - Possible in all registers except segment and flag registers.
    - MOV BX,1234H ; move 1234H into BX
    - MOV CX,223 ; load the hexadecimal value into CX
    - ADD AL,40H ;
    - MOV DS,1234H ;illegal

Addressing Mode

- Direct
  - Address of the data in memory comes immediately after the instruction operand is a constant
  - The address is the offset address. The offset address is put in a rectangular bracket
    - MOV DL,[2400] ; move contents of DS:2400H into DL

Addressing Mode

- Find the physical address of the memory location and its content after the execution of the following operation. Assume DS=1512H
  - MOV AL,99H
  - MOV [3518],AL
  - Physical address of DS:3518 => 15120+3518=18638H
  - The memory location 18638H will contain the value 99H

Addressing Mode

- Register indirect
  - The address of the memory location where the operand resides is held by a register.
  - SI, DI and BX registers are used as the pointers to hold the offset addresses.
  - They must be combined with DS to generate the 20-bit physical address
    - MOV AL,[BX] ; moves into AL the contents of the memory location pointed to by DS:BX
    - MOV CL,[SI] ; move contents of DS:SI into CL
    - MOV [DI],AH ; move the contents of AH into DS:DI

Addressing Mode

- Based relative
  - BX and BP are known as the base registers. In this mode base registers as well as a displacement value are used to calculate the effective address.
  - The default segments used for the calculation of Physical address (PA) are DS for BX, and SS for BP.
    - MOV CX,[BX]+10 ; move DS:BX+10 and DS:BX+11 into CX ; PA = DS (shifted left) +BX+10
    - Note that, the content of the low address will go into CL and the high address contents will go into CH.
    - There are alternative coding:
    - MOV CX,[BX+10], MOV CX,10[BX]
    - BX+10 is effective address

Addressing Mode

- Indexed relative
  - Indexed relative addressing mode works the same as the based relative addressing mode.
  - Except the registers DI and SI holds the offset address.
    - MOV DX,[SI]+5 ;PA=DS(shifted left)+SI+5
    - MOV CL,[DI]+20 ;PA=DS(shifted left)+DI+20

Addressing Mode

- Based indexed relative
  - The combination of the based and indexed addressing modes.
  - One base register and one index register are used.
    - MOV CL,[BX][DI]+8 ;PA=DS(shifted left)+BX+DI+8
    - MOV CH,[BX][SI]+20 ;PA=DS(shifted left)+BX+SI+20
    - MOV AH,[BP][DI]+12 ;PA=SS(shifted left)+BP+DI+12
    - MOV AL,[BP][SI]+29 ;PA=SS(shifted left)+BP+SI+29
    - Alternative coding
    - MOV CL,[BX+DI+8]
    - MOV CL,[DI+BX+8]

Address calculation

- Physical Address
  - The 20 bit address which we needs to be stored.
  - It ranges from 00000H to FFFFFH (Hexadecimal notation) .
- Base Address
  - The address at which a given memory segment starts and we use it for de-markation.
- Offset address
  - (Distance from the base address) is a location with 64 kb segment range. It ranges from 0000H to FFFFH

Address calculation

- Logical address
  - Something we denote on paper as a short hand representation of the above addresses. It consists of a segment value and offset address.
  - Logical address is specified as Segment base : Offset value.

Address calculation

- Physical address is obtained by shifting the segment address 4 bits to left adding the offset address.
  - Physical address = (Segment base*10H) + Offset Value.
    - Logical address:A4FBH:4872H
    - Segment's base address:A4FBH
    - Offset value:4872H
    - Shifting the segment address 4 bits to left A4FBH<<4 gives A4FB0H
    - Now adding offset address to A4FB0H
    - A4FB0H+4872H=A9822H

Assembly Language Programming

- Data Movement Instructions
- Arithmetic and Logic Instructions
- Program Control Instructions
- Special Instructions
- Assembly Language Version
  - operation destination, source
  - operation destination
  - operation source
  - operation

Assembly Language Programming

- Data Movement Instructions
  - ‘MOV’ transfers a constant, contents of memory or register to a register or memory.
    - MOV AX,DX Transfers contents of DX to AX, leaving DX unchanged
    - MOV AL,CH Transfers contents of CH to AL, leaving CH unchanged
    - MOV BL,CX Not allowed
    - MOV AH,56 56 decimal is transferred to AH
    - MOV BX,0ABCDH Transfers ABCD hexadecimal to BX
    - MOV 67H,BL Not allowed
    - MOV DS,3467H Not allowed
    - MOV DS,BX Loads DS with contents of BX, leaving BX unchanged
    - MOV [2345H],AH Transfers contents of AH to the memory location whose offset is 2345H
    - MOV [DI],56H Transfers 56H to the memory location whose offset is stored in DI
    - MOV [DI],[SI] Memory to memory move is not allowed

Assembly Language Programming

- MOV Instructions

MOV Source Operand	Destination Operand			
	General Register	Segment Register	Memory Location	Constant
General Register	Yes	yes	yes	no
Segment Register	Yes	no	yes	no
Memory Location	Yes	yes	no	no
Constant	Yes	no	yes	no

Assembly Language Programming

- Data Movement Instructions
  - ‘XCHG’ swaps the contents of its operands. Operand can both be registers or a register and a memory location.
    - XCHG AX,BX
    - XCHG AL,CL
    - XCHG 32H,BH Invalid operation
    - XCHG [2345H],[SI] Invalid operation

XCHG Source Operand	Destination Operand	
	General Register	Memory Location
General Register	yes	yes
Memory Location	yes	no

Assembly Language Programming

- Data Movement Instructions
  - PUSH
    - PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed to by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.
  - POP
    - POP transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

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)**

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**

Instruction Set

1. Data Transfer Instructions

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

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

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

Assembly Language Programming

- Arithmetic Instructions
  - ADD
    - ADD instruction adds values exiting in two registers, register and memory, immediate and memory, immediate and register. To add two values existing in AL and BH register, ADD is used as:
    - ADD AL, BH
    - This instruction adds the contents of AL and BH registers, leaving sum in AL and BH unchanged. ADD is applied, for example, on CX and DX in the same way. Consider CX is containing 18 and DX containing 25 before executing the following instruction.
    - ADD CX, DX
    - After this instruction is executed, CX is left with 43, DX with 25 and CF with 0.
    - Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with ADD. For 16-bit addition, ADD can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of ADD instruction for other 16-bit registers yourself.

Assembly Language Programming

- Arithmetic Instructions
  - ADC
    - ADC, pronounced as Add with Carry, instruction adds the content of operands along with value in carry flag that exists in it at the instant instruction ADC is executed. Consider SI contains 15678, DI contains 325 and CF is set. After executing following instruction,
    - ADC SI, DI
    - SI is left with 16004 (3E84H), DI with 325 (145H) and CF is cleared as the latest result does not produce overflow condition.

Assembly Language Programming

- Arithmetic Instructions
  - INC
    - Increment instruction INC adds 1 to the contents of its operand. It can be applied on any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL). For 16-bit increment, INC can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of INC instruction for other 16-bit registers yourself.
    - Consider AL contains 97H before execution of following intrusion.
    - INC AL
    - After execution, AL is left with 98H.

Assembly Language Programming

- Arithmetic Instructions
  - SUB
    - SUB instruction subtracts the contents of operand2 from operand1 and leaves the difference in operand1, with operand2 unchanged.
    - SUB operand1, operand2
    - Consider content of AH are to be subtracted from content of DL then SUB can be used as,
    - SUB DL, AH
    - For 16-bit subtraction,
    - SUB DX, DI
    - SUB BX, AX
    - Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with SUB. For 16-bit subtraction, SUB can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of SUB instruction for other 16-bit registers yourself.

Assembly Language Programming

- Arithmetic Instructions
  - SBB
    - SBB, pronounced as Subtract with Borrow, subtracts the contents of operand2 and CF from the content of operand1 and leaves the result in operand1. Operand2 is left unchanged and CF depends upon whether most significant bit in operand1 required a borrow bit or not. If borrow was required CF is set, otherwise cleared. Format of SBB is,
    - SBB operand1, operand2
    - Consider CF is set, BX = 67ABH and DX = 100H before executing following instruction.
    - SBB BX, DX
    - After execution, BX is left with 66AAH, DX with 100H and CF is cleared.

Assembly Language Programming

- Arithmetic Instructions
  - DEC
    - Decrement instruction DEC subtracts 1 from the contents of its operand. It can be applied on any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL). For 16-bit decrement, DEC can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of DEC instruction for other 16-bit registers yourself.
    - Consider DI contains 55968 before execution of following intrusion.
    - DEC DI ; after execution, DI is left with 55967.

Assembly Language Programming

- Arithmetic Instructions
  - NEG
    - NEG instruction is single operand instruction that leaves the operand with 2's complement of its original data. For example register AX contains 12656 (3170H). After executing NEG on AX, new contents of AX will be CE90H which is -12656. NEG can be applied on 8-bit registers, 16-bit registers or memory location. Consider,
    - NEG AX

Assembly Language Programming

- Arithmetic Instructions
  - MUL
    - MUL carry out multiplication on two operands in 8086-88 CPU. MUL is a single operand instruction whereas other operand is assumed to be in a specified register. MUL can be applied on two 8-bit values producing 16-bit result, and on 16-bit values producing 32-bit result.
    - For 8-bit multiplication, one operand is assumed in AL register whereas other is the part of instruction. For example, in order to multiply 230 with 165, one of these values must be in AL register. Other operand can be in any 8-bit register or in memory location. Consider following code.
    - MOV AL, 230
    - MOV BL, 165
    - MUL BL
    - After executing these three instructions, 16-bit result (37950 in this case) will be found in AX register, while content of BL are left unchanged.

Assembly Language Programming

- Arithmetic Instructions
  - MUL
    - For 16-bit multiplication, one operand is assumed in AX register whereas other is the part of instruction. For example, in order to multiply 22330 with 10365, one of these values must be in AX register. Other operand can be in any 16-bit register. Consider following code.
    - MOV AX, 22330
    - MOV BX, 10365
    - MUL BX
    - After executing these three instructions, 32-bit result (231450450 in this case) will be found in DX-AX registers. DX register contains most significant 16 bits (from bit 16 to bit 31) and AX contains least significant 16 bits (from bit 0 to bit 15). Content of BX are left unchanged.

Assembly Language Programming

- Arithmetic Instructions
  - DIV
    - To do division in 8086-88, DIV instruction is provided by the instruction set of 8086-88 CPU. Like MUL, DIV can be done on 8-bit data and on 16-bit data. In 8-bit division, dividend (numerator) is stored in AX register while 8-bit divisor (denominator) is stored in 8-bit register or in memory. After executing DIV, 8-bit quotient moves in AL while 8-bit remainder moves in AH.
    - Consider the following code that divides 2334 by 167.
    - MOV AX, 2334
    - MOV BH, 167
    - DIV BH
    - After executing above three instructions, AL contains DH (13 decimal, the quotient) and AH contains A3H (163 decimal, the remainder).

Assembly Language Programming

- Arithmetic Instructions
  - DIV
    - In 16-bit division, 32-bit dividend (numerator) is stored in DX-AX registers such that most significant 16 bits are in DX and least significant 16 bits are in AX. 16-bit divisor (denominator) is stored in a register. After executing DIV, 16-bit quotient moves in AX while 16-bit remainder moves in DX.
    - Consider the following code that divides 235634 (39872H) by 45667 (B263H).
    - MOV AX, 9872H
    - MOV DX, 3H
    - MOV BX, 0B263H
    - DIV BX
    - After executing above three instructions, AX contains 5H and DX contains 1C83H.

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**

Instruction Set

2. Arithmetic Instructions

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

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

Instruction Set

2. Arithmetic Instructions

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

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

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…**

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**

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**

Assembly Language Programming

- Logic Instructions
  - NOT
    - NOT instruction takes one complement of operand that can be 8-bit data or 16-bit data. For example in order to apply NOT on AH register, following syntax is used.
    - NOT AH
    - After executing this instruction, AH is left with complement of its contents prior to execute this instruction.
    - Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with NOT. For 16-bit NOT, it can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of SUB instruction for other 16-bit registers yourself.

Assembly Language Programming

- Logic Instructions
  - AND
    - AND instruction do logical AND on two 8-bit data or on two 16-bit data. For example, in order to logically AND contents of BH and DL, AND is used as:
    - AND BH, DL
    - This instruction logically ANDs the contents of BH and DL registers, leaving result in BH and DL remains unchanged. AND can be applied on 16-bit data stored in registers in the similar way. Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with AND. For 16-bit AND, instruction can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of AND instruction for other 16-bit registers yourself.

Assembly Language Programming

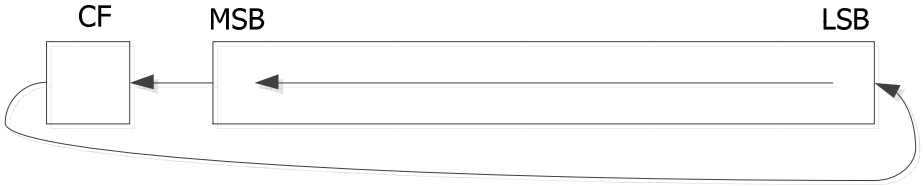
- Logic Instructions
  - OR
    - OR instruction executes logical OR on two 8-bit data or on two 16-bit data. For example, in order to logically OR contents of BH and DL, OR is used as:
    - OR BH, DL
    - This instruction logically ORs the contents of BH and DL registers, leaving result in BH and DL remains unchanged. OR can be applied on 16-bit data stored in registers in the similar way. Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with OR. For 16-bit OR, instruction can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of OR instruction for other 16-bit registers yourself.

Assembly Language Programming

- Logic Instructions
  - XOR
    - XOR instruction executes exclusive OR on two 8-bit data or on two 16-bit data. For example, in order to exclusive OR contents of BH and DL, XOR is used as:
    - XOR BH, DL
    - This instruction XORs the contents of BH and DL registers, leaving result in BH and DL remains unchanged. XOR can be applied on 16-bit data stored in registers in the similar way. Any 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL) can be used with XOR. For 16-bit XOR, instruction can be applied on AX, BX, CX, DX, SI and DI. You are encouraged to check the validity of XOR instruction for other 16-bit registers yourself.

Assembly Language Programming

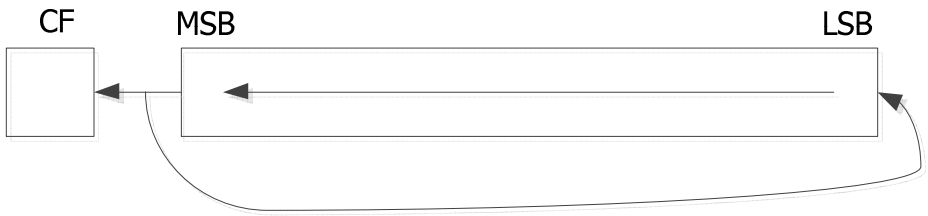
- Logic Instructions
  - RCL
    - There are four different types of rotate instructions. RCL instruction stands for “Rotate Left through Carry” positions the bits in a register or in memory location according to following scenario.



    - Previous value of CF goes to LSB of operand and MSB of operand goes in to the CF. Bits in the operand are shifted left by one bit at a time.
    - Applying RCL on AH, for example, is:
    - RCL BL, 6
    - Above instruction accomplish RCL operation on BL for six times. If number of shift operations is a variable value, then it is to be placed in CL. Result is stored back in BL.

Assembly Language Programming

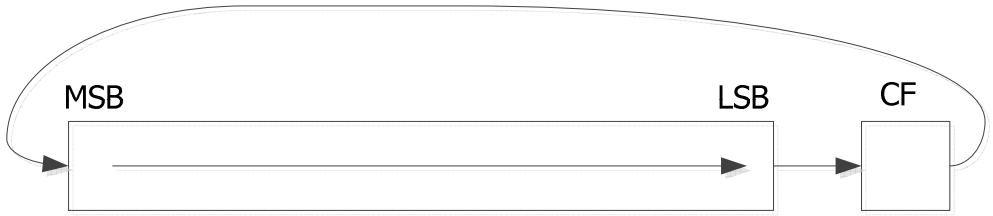
- Logic Instructions
  - ROL
    - The rotate instruction ROL stands for “Rotate Left”. ROL positions the bits in a register or in memory location according to following scenario.



    - Previous value of MSB of operand goes in to the CF and the same also enters in to the LSB position, with each bit in operand shifting left by one bit at a time.
    - Applying ROL on SI, for example, is:
    - RCL SI,14
    - Above instruction executes ROL operation on SI for fourteen times. If number of shift operations is a variable value, then it is to be placed in CL. Result is stored back in SI.

Assembly Language Programming

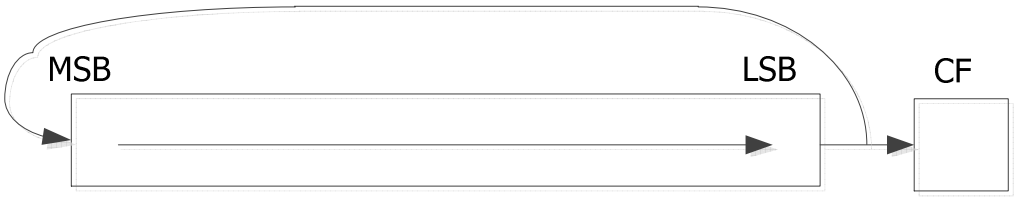
- Logic Instructions
  - RCR
    - RCR, “Rotate Right through Carry”, behaves in reverse of RCL. Function of RCR is represented in following diagram.



    - Previous value of CF goes to MSB of operand and LSB of operand goes in to the CF. Bits in the operand are shifted right by one bit at a time.
    - Consider following instruction.
    - RCR AH, CL
    - After execution, original content of AH register are rotated right times the value in CL.
    - Result is stored back in AH.

Assembly Language Programming

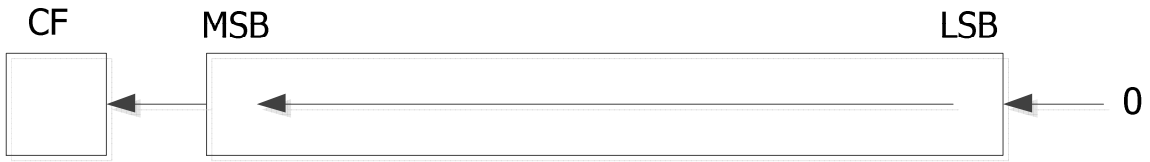
- Logic Instructions
  - ROR
    - The rotate instruction ROR stands for “Rotate Right”. ROR positions the bits in a register or in memory location according to following scenario.



    - Previous value of LSB of operand goes in to the CF and the same also enters in to the MSB position, with each bit in operand shifting right by one bit at a time.
    - Consider following instruction.
    - RCR [23ABH], CL
    - After execution, original content of memory location with offset address 23ABH are rotated right though carry times the value in CL. Result is stored back in memory location with offset address 23ABH.

Assembly Language Programming

- Logic Instructions
  - SHL
    - Like rotate instructions, shift instructions are four in number. Instruction SHL, abbreviated from “Shift Left”, shifts the bits in operand left one bit at a time. Instruction SAL, “Shift Arithmetic Left” behaves exactly identical to SHL as the MSB, that reflects the positivity or negativity of value is altered in shift left operation.
    - Bit shifted out from MSB of operand enters in to the CF. 0 is entered in the LSB position for a single SHL or SAL operation. This operation is diagrammatically represented below.

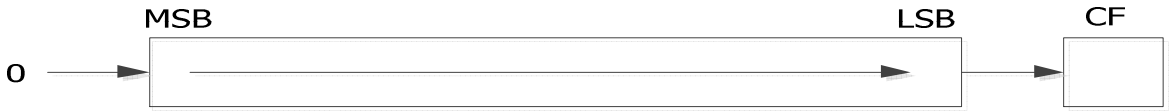


Assembly Language Programming

- Logic Instructions
  - SHL
    - Consider following instructions.
    - SHL BX, 4
    - SAL DI, 6
    - The instruction SHL BX, 4shifts the contents of BX four bits left, leaving result in BX. If number of shifts is a variable value then it is to be stored in CL. SHL can be done on 8-bit register, 16-bit register or on a memory location.
    - The instruction SAL DI, 6 behaves exactly similar to above discussed instruction with difference that content of DI are shifted six bits left now.

Assembly Language Programming

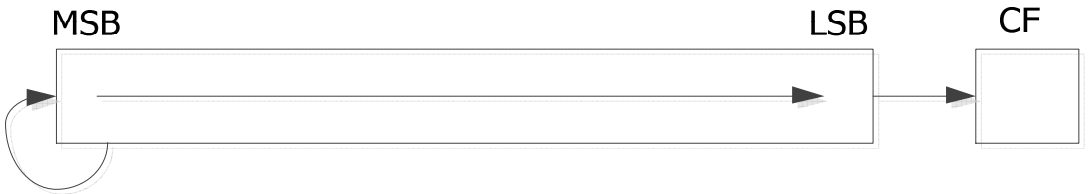
- Logic Instructions
  - SHR
    - Instruction SHR, abbreviated of “Shift Right”, shifts the bits in operand right one bit at a time. Bit shifted out from LSB of operand enters in to the CF. 0 is entered in the MSB position for a single SHR operation. This operation is diagrammatically represented below.



    - Consider following instruction.
    - SHR DX, CL
    - After execution, original content of DX are shifted right times the value in CL. Result is stored back in DX register.

Assembly Language Programming

- Logic Instructions
  - SAR
    - Instruction SAR, abbreviated of “Shift Arithmetic Right”, shifts the bits in operand right one bit at a time. Bit shifted out from LSB of operand enters in to the CF with MSB remains unchanged. This retains the positivity or negativity of value in operand. This operation is diagrammatically represented below.



    - Consider following instruction.
    - SAR DH, CL
    - After execution, original content of DH are shifted right times the value in CL with MSB preserved. Result is stored back in DH register.

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

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

Instruction Set

3. Logical Instructions

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

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

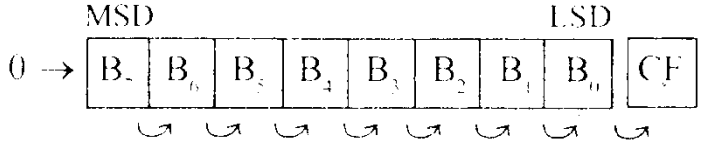
SHR mem

i) SHR mem, 1

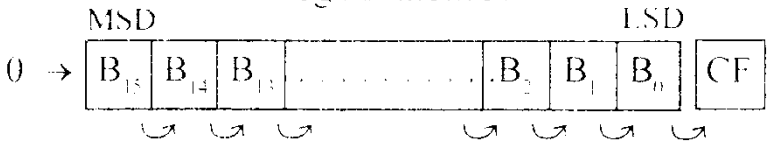
ii) SHR mem, CL

$CF \leftarrow B_{LSD} ; B_n \leftarrow B_{n+1} ; B_{MSD} \leftarrow 0$

reg 8 / mem 8



reg 16 / mem 16



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

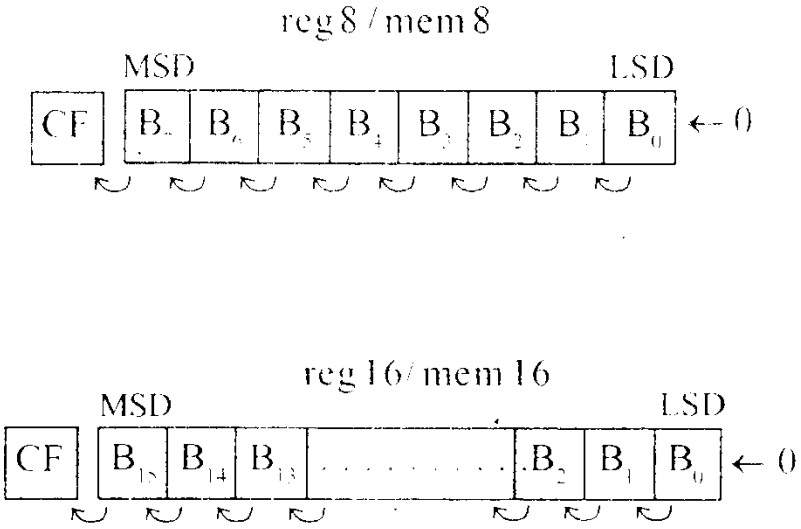
ii) SHL reg, CL or SAL reg, CL

SHL mem or SAL mem

i) SHL mem, 1 or SAL mem, 1

ii) SHL mem, CL or SAL mem, CL

$CF \leftarrow B_{MSD} ; B_{n-1} \leftarrow B_n ; B_{LSD} \leftarrow 0$



Instruction Set

3. Logical Instructions

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

RCR reg/mem

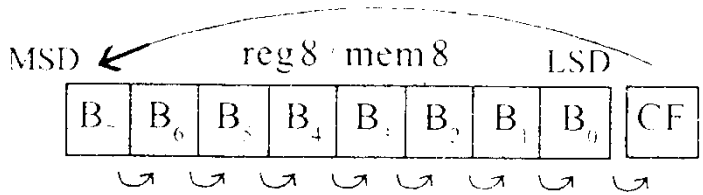
RCR reg

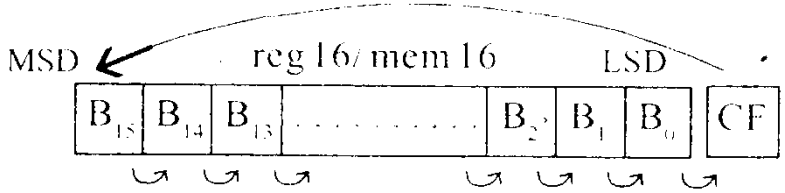
i) RCR reg, 1
ii) RCR reg, CL

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 …**

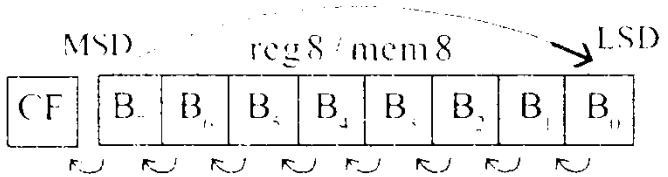
ROL reg/mem

ROL reg

i) ROL reg, 1

ii) ROL reg, CL

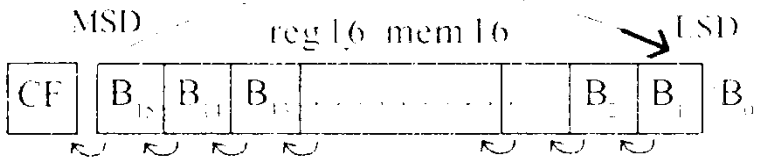
$B_{n+1} \leftarrow B_n ; CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$



ROL mem

i) ROL mem, 1

ii) ROL mem, CL



Assembly Language Programming

- Program Control Instructions
  - JMP
    - Abbreviation of jump, JMP instruction directs the program flow to the instruction associated with the label name that follows the JMP instruction. For example consider following use of JMP instruction.
    - ```
      JMP   CIIT
      ```
    - This instruction diverts the program flow to the instruction which is labeled as CIIT. Program starts executing from that instruction and flows in a normal sequential way onwards. Consider following piece of code.
    - ```
      MOV   CL, BH
      AND   BH, 32H
      JMP   SP14
      XOR   AX, AX
      DEC   BX
      ```

Assembly Language Programming

- Program Control Instructions
  - JMP
    - SP14:
    - MOV AH, 16
    - MOV CL, 3
    - SHR AH, CL
    - Program execution starts from line 1 and goes to line 3 according to normal flow. In line 3, instruction JMP is encountered that directs the program flow to instruction in line 7. Remember that a label name is not an instruction. Once program flow is directed to line 7 then it continues execution subsequently by executing instructions in line 8 and then in line 9 and so on. Redirecting the program from one instruction to another which is not in subsequence of previous instruction is called jumping. As in this example program jumps to SP14 label with no dependency on results of last instruction, it is called unconditional jumping.

Assembly Language Programming

- Program Control Instructions
  - CMP
    - CMP (compare) instruction compares its operands through subtraction. CMP does not modify the operands rather it updates the flag register only. Consider use of CMP given below.
    - CMP BL, AL
    - This instruction subtracts the contents of AL register from contents of BL register but result is not sored anywhere. Only flags are updated according to one of three possible outcomes which are “Is AL greater than BL”, “Is AL less than BL” and “Is AL is equal to BL”.

Assembly Language Programming

- Program Control Instructions
  - JZ
    - JZ (Jump if Zero) is conditional jump instruction. It is used as follows.
    - JZ Label name
    - IF result of last operation (operation completed just before using JZ instruction) is zero, zero flag (ZF) would have been set, otherwise cleared. This instruction checks the ZF and if it is found set, then program is directed to the instruction associated with label name that is used in JZ instruction. Otherwise, jumping is skipped and instruction following the JZ is executed. Consider following piece of code.

```
MOV     AX, 3456
MOV     BX, AX
SUB     AX, BX
JZ      ZERO
AND     BX, 2345H
MOV     SI, BX
```

Assembly Language Programming

- Program Control Instructions
  - JZ
    - ZERO:
    - MOV DI, SI
    - AND DI, AX
    - When instruction in line 4 is to be executed, it will be checked if ZF is set or not. As due to the instructions in line 1 to line 3 cause ZF = 1, jump to label ZF will be taken and instruction in line 8 shall be executed after instruction in line 4. If ZF would have not been set at the time of execution of instruction in line 4, then next instruction would be of line 5. Note that as label name is not an instruction, in above code, instruction in line 8 would be executed after instruction in line 6.

Assembly Language Programming

- Program Control Instructions
  - JNZ
    - JNZ (jump if Not Zero) behaves oppositely of JZ. Jump would be taken to specified label if ZF is cleared and will not be taken if ZF is set at the time of execution of JNZ.
  - JC
    - JC (Jump if Carry) directs the program flow to the specified label if CF is set at the time of execution of JC instruction. Jumping will be skipped otherwise.
  - JNC
    - JNC (Jump if No Carry) directs the program flow to the specified label if CF is cleared at the time of execution of JNC instruction. Jumping will be skipped otherwise.

Assembly Language Programming

- Program Control Instructions
  - JG
    - JG (Jump if Greater) instructions deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is greater than operand 2, then JG will direct the flow to the label associated with it.
    - CMP operand 1, operand 2
    - JG label
  - JGE
    - JGE (Jump if Greater or Equal) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is greater than or equal to operand 2, then JG will direct the flow to the label associated with it.
    - CMP operand 1, operand 2
    - JGE label

Assembly Language Programming

- Program Control Instructions
  - JL
    - JL (Jump if Less) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is less than operand 2, then JL will direct the flow to the label associated with it.
    - CMP operand 1, operand 2
    - JL label
  - JLE
    - JLE (Jump if Less or Equal) instructions also deals operands of CMP instruction as signed numbers. This instruction is generally used in conjunction with CMP instruction. Upon comparison, if operand 1 is less than or equal to operand 2, then JLE will direct the flow to the label associated with it.
    - CMP operand 1, operand 2
    - JLE label

Assembly Language Programming

- Program Control Instructions
  - CALL
    - CALL is used to direct program to a subroutine. Consider following piece of code.

```
        MOV     CL, BH
        AND     BH, 32H
        CALL    SP22
        XOR     AX, AX
        DEC     BX
SP22:
        MOV     AH, 16
        MOV     CL, 3
        RET
```

Assembly Language Programming

- Program Control Instructions
  - CALL
    - When program flow executes CALL instruction in line 3, it is directed to the instruction in line 7 from where it starts execution sequentially. When flow encounters the RET instruction in line 9, it directs program back to the instruction following immediately after the most recent executed CALL instruction. In this example, after executing RET instruction in subroutine SP14, instruction in line 4 is executed and program flows onwards normally. This is called “returning from call”. Another CALL instruction can be used without returning from a call. This is called “nested calling”
  - RET
    - RET (Return) instruction, normally placed at the end of a subroutine to terminate it, brings the control back to the instruction that follows immediate after the CALL instruction using which the current subroutine was called.

Assembly Language Programming

- Program Control Instructions
  - Loop
    - LOOP instruction moves to prescribed label iteratively. Value in the CX register determines the number of iterations. With each LOOP execution, CX decrements automatically. LOOP keeps on executing as long as CX reaches zero. Consider following code.

```
        MOV     CX, 100
        XOR     AX, AX
HERE:
        ADD     AX, 1
        LOOP    HERE
```

Assembly Language Programming

- Program Control Instructions
  - INT
    - To invoke DOS or BIOS routine, the INT (interrupt) instruction is used. It has the format
    - INT interrupt_number
    - Where interrupt_number is a number that specifies a routine. For example INT 16h invokes a BIOS routine that performs keyboard input. In this Lab you will use a particular DOS routine, INT 21h. INT 21h may be used to invoke a large number of DOS functions. A particular function is requested by placing function number in the AH register and invoking INT 21h.
  - HLT
    - To halt the running programme