B → high speed internal connection, to send control signals etc.

Page No. Date

MOD-5

* Architecture of 8086 (Block diagram). (16 bit ALU.)
  i) Bus interface unit (BIU)
  ii) Execution unit (EU).

* BIU performs all operations related to system bus like instruction fetching, reading & writing operands and calculation the address of memory operands.

* EU executes the instructions from the instruction system byte Queue.

BIU:

* Instruction queue has 6 instruction bytes.

* segment registers ⇒ CS = code segment
  DS = data segment
  SS = stack segment
  ES = extra segment
  IP = instruction pointer.

* E ⇒ address adder.

*

EU:

* control system, instruction decoder (ALU)

* Pointer registers ⇒ SP = stack pointer (16-bit)
  BP = base pointer

* Index registers ⇒ SI = Source index
  DI = Destination index.

* BIU & EU works asynchronously.
* Address bus / segment bus = 1 mb.

* General purpose registers : (8 bits) × 2.

  AX = accumulator registers
  BX = base registers
  CX = counters
  DX = hold I/O port address for I/O instruction.
     = Destination / data register.

* ALU ⇒ Operands & Flags.

-

* Both units work asynchronously to give 8086 an overlapping instruction fetch & execution mechanism called pipeline.

* fetching & execution is done at same time = pipelining.
* Fetch next instruction while executing the first is called pipelining.

BIU : 16 bit data bus, 20 bit address bus.

- responsible for all external bus operations.
  - instruction fetch
  - instruction queue.
  - ~~instruction~~ operand fetch & storeage.
  - address relocation
  - bus control.

- uses a mechanism → instruction queuing / instruction stream byte queue to implement pipe line.

* In the instruction queue, 6 instruction can be pre-fetched.
* The queue permits.
* These instructions are held in FIFO queue.
* with its 16 bit data bus, BIU fetches 2 instruction in single memory cyle.
* After the byte is loaded at the i/p end of queue, it automatically shifts up.
* EU access the queue from O/P end. It reads one instruction byte after the other from the output of the queue.
* intervals of no of bus activity, which may occur betn bus cycles are known as IDLE state. i.e. when queue is empty & has no task to perform.
* If BIU is fetching inst. & in that time EU requests to read/write operands from memory Then BIU first completes the task of fetching & then initiating the operand read/write cycle.
* BIU has address adder which is used to generate 20 bit physical address i.e. output on address bus. Lo 20 bit adress line.
* 20 bit physical address is generated by adding an appended 16 bit segment address & 16 bit offset add.
(es, DS, BS, ES) (BI, SP, DI, IP)
* CS → IP * BS — SI
* DS → SP * DS - DI

* BIU is also responsible for generating control signals

Execution Unit:

- responsible for decoding & executing instructions.
- extracts instruction from queue (1) & then decodes them & generates the operands, passes the operands to BIU, to perform read or write operations.

* EU tests the status & control flags & updates them according to result.
* control flag = controls operation of execution unit
  1. Trap flag = allow one inst at a time
  2. Interrupt flag = interrupt enable / disable flag.
  3. Direction flag = used in string operation

* If queue is empty, EU waits for next inst byte to be fetched & shifted to top of queue

* When EU executes branch or jump inst, it transfers control to a location corresponding to another set of instruction.

* BIU automatically resets the queue & then begin to fetch the ~~data~~ instruction from new location.

16 bit – data bus
20 bit – address bus.

* Pipelining fails when we have a branch

conditional flags
* carry flag ⇒ indicates overflow condition.
* auiliary flag ⇒ Binary to BCD conversion, to pass the carry generated

16 bits
* parity flag ⇒ even parity, odd parity.
* zero flag ⇒ flag = 1, when result = 0.
* sign flag ⇒ hold sign of result, flag = 1, ans = -ve.
* overflow flag ⇒ represents result when system capacity exceeds.

control flags
* Trap flag ⇒ allows only one instruction execution.
* Interrupt flag ⇒ enables & disable interrupts.
* Direction flag ⇒ used in string operations.

16 bit
* CS → code segment, addressing mem. loc in code seg. of mem.
* DS → data segment, —11— (full) & consists of data used by program
* SS → stack segment, handles memory to store data & addr. during execution.
* ES → extra seg., additional data segment
* IP → instruction ptr = 16 bit.

CS → IP
DS → SBI
SS → SP/BP
ES → DI
A → offset.

address location → (IP, SI, SP, DI) 16 bits

* physical address = seg x 10_H + offset.

* BIU
① Fetch inst.
② calc phy. add
③ manage queue

CS & IP will give address.
* operands → temp. registor.
* flags → status about result.

Protect out
* increase clock freq. to make processor faster
* increase ~~address~~ data bus size.

* Pipelining → makes processor fast

Hazards of pipelining ⇒ control hazards
⇒ data dependency
⇒ structural hazar.

Addressing Mode:

- Different ways in which source operand is denoted in an instruction is known as addressing mode

MOV Instruction Formats

→ MOV is important instruction, as it moves data in 8086.
→ has wide variety of parameters

Format

Mov destination, source.

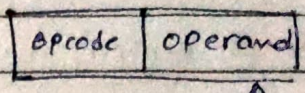
destination	source	example.
register	register	mov ax; bx
register	immediate	mov ax, 10H
register	memory	mov ax, es:[bx]
memory	immediate	mov aNumber, 10H
memory	register	mov aDigit, ax

* MOV copies the data in the source to the destination.
* Data is either a byte or word.

* for mov, we only use general purpose registers.
* can't access segment registers.

Page No.
Date

* limitations in MOV inst.



① an immediate value cannot be moved into segment register (cs, DS ..etc) directly.
i.e. (mov ds, 10)

② segment registers cannot be copied directly.
i.e. (mov es , ds) .

③ a memory locatn cannot be copied to another memory location
i.e. (mov aNumber, a Digit)

④ es cannot be copied to. i.e. (mov cs , ax)

* Addressing modes : different ways in which source operand is a part of instruction.

① Implicit addressing mode: INCA (increment accumulator)
- CRC = clear carry
- CLA = compliment accumulator.

* no parameter / data to be given.

* ② Immediate addressing mode:
- operand is provided as immediate / constant
- data is immediate / constant

⇒ Add R1 , # 3
↳ direct data / immediate.

* value should be betn $0 - 2^{12}$

* ③ Register mode

- operand / data is present in register.

opcode	operand

↑ Register.

eg: Add Ax, cx

↑ register.

- operand / data is the register no, we have 16 registers, so from 0000 to 1111 i.e. FFFF will be given to Instruction.

eg: Add Ax, 0001

↑ register number.

* ④ Register indirect mode:

- Register in operand contains address of the operand.

opcode	register with address

eg: MOV AX, [BX] ← 4 bit

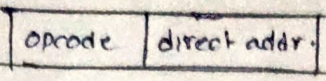
↑

register which contains location / address of the operand.

⑤ Direct Addressing mode:

0-4095 location

- Actual address is given in instruction.
- used to access variable.



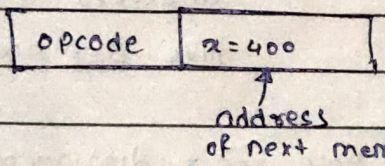
of 12 bits only

eg: Add ax, [1234H]

↑ memory location, (address of data)

⑥ Indirect addressing mode:

- used to implement pointers & passing parameters
- 2 memory access required.



effective addr. = m[x].

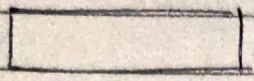
- operand will contain the address to the value of final operand.

⑦ Relative a.m:

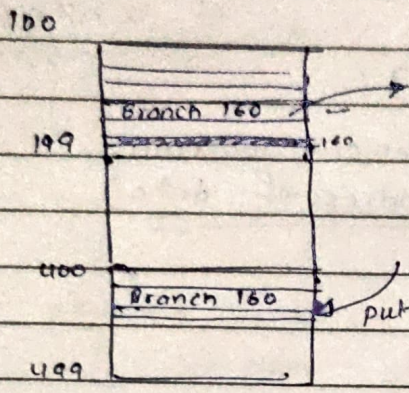
- effective address = PC + offset.
- give displacement (-1).

* (8) Base register am:

- used in program relocation.



Base registor



* this will create error as the location is changed,
* in such case, rather than giving specific address location, we give displacement.

∴ rather than giving Br = 160.

We give Br 160

∴ Effective address = base reg + displacement

= 400 + 60. OR 100 + 60.

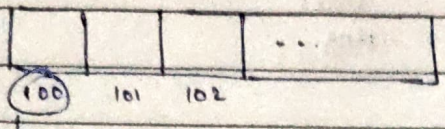
= 460 = 160.

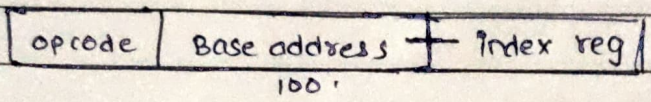
eg: MOV DX, [BX + 04].

operand. displacement

⑨ Indexed am :

- to access or implement array.
- multiple registers used.
- Any element can be accessed w/o changing instruction





to access other elements, use any registers as index registers

→ (SI, DI)

Effective address = Base addr. + Index reg. value

eg: MOV BX, [SI + 16]

Base address = 16, Index reg. value = SI.

⑩ Based index : offset = base reg + index register.

Add CX [AX + SI]

↓

offset.

displacement

(6) Based indexed with displacement:

offset = base registers + index registers + displacement

eg: MOV AX, [BX + DI + 02],

BX → base reg
DI → index Reg
02 → displacement

displacement = 8 or 16 bit

(11) Based index with displacement:

offset = base registers + index registers + displacement
8 or 16 bit

eg: MOV AX, [BX + DI + 08],

BX → base reg
DI → index Reg
08 → displacement