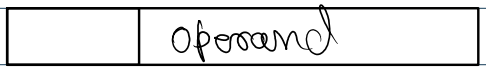
Addressing Mode

A : Content of an address field in instruction
R : Content of an address field in instruction that refers register

① Immediate Addressing

Instruction

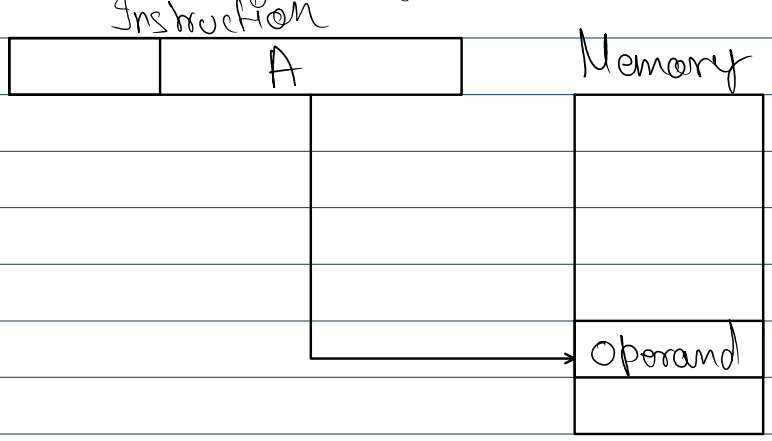


→ No memory Refrence

operand = A

↳ Limitation of operand magnitude

② Direct Addressing



EA = A → Simple address in memory

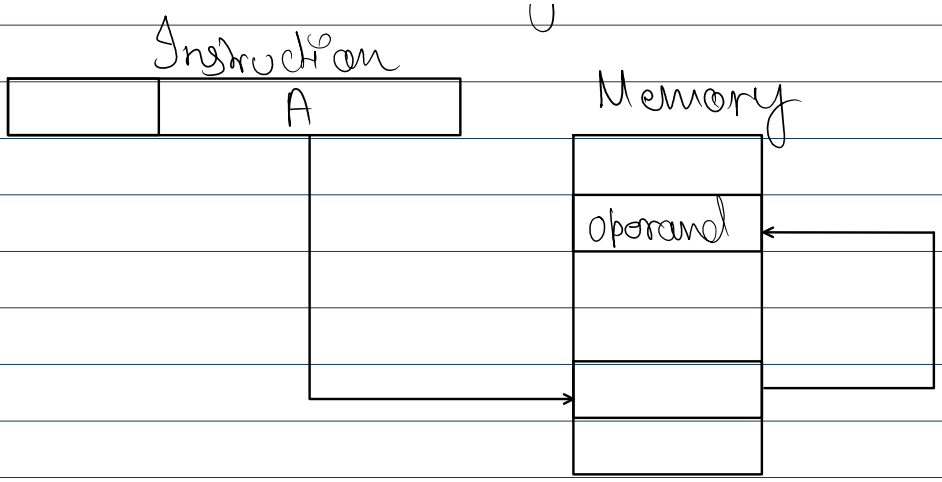
↓ effective address

↳ requires min. one memory refrence

↳ provides limited address space

③ Indirect Addressing

Instruction



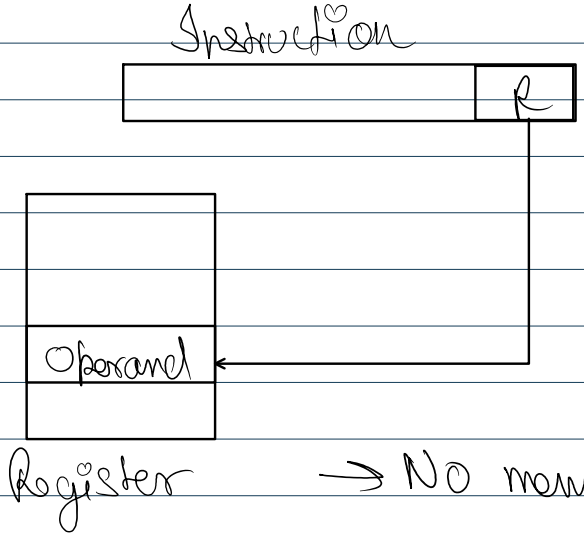
EA = (A)

In this address field refers to another address containing the operand

→ Gives advantage of large address space

↳ Multiple memory refrence required

④ Register Addressing



EA = R

(↳ Similar to direct addressing)

Address field refers to Register

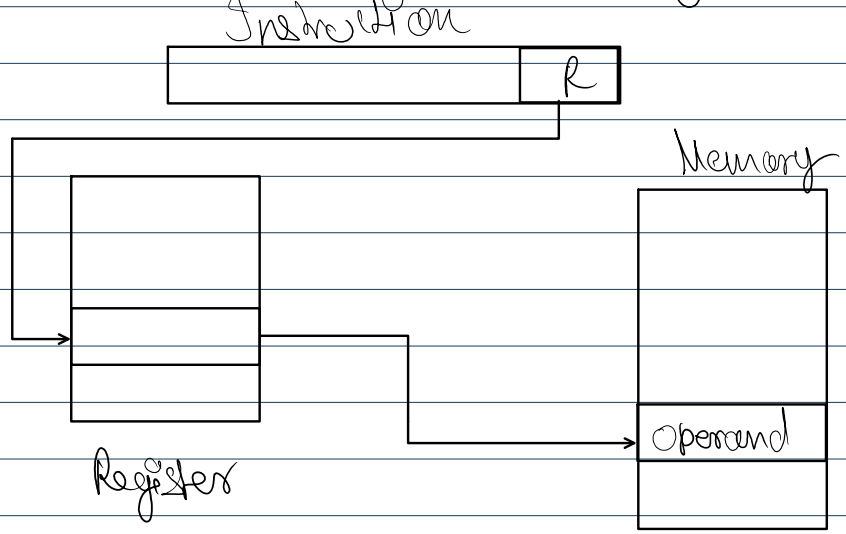
→ NO memory refrence required

↳ Limited address Space (very limited Space)

→ only small address field needed in instruction

→ no time-consuming memory refrences

(5) Register Indirect Addressing



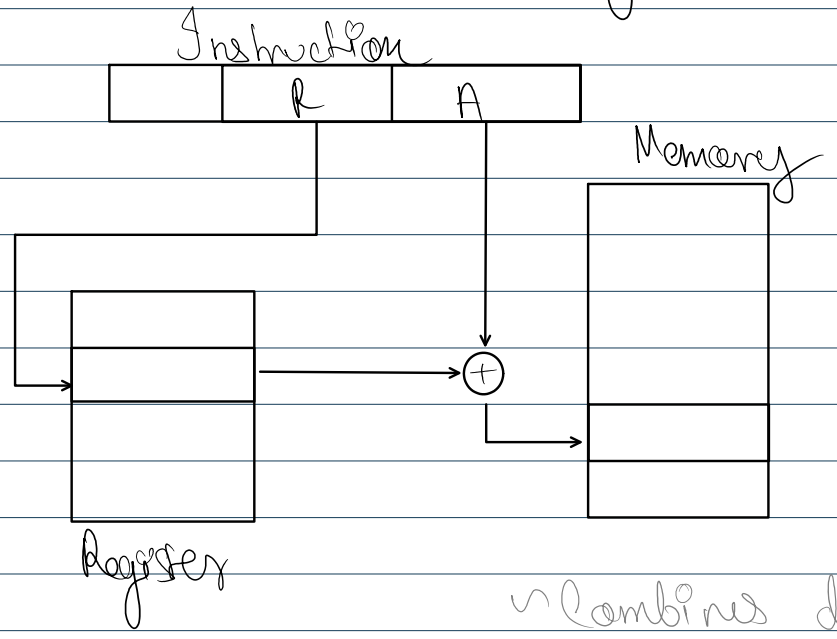
↳ analogous to indirect Addressing

EA = (R)

→ Large address Space

Register gives memory refrence overcoming limitation of Register addressing

(6) Displacement Addressing



Very powerful mode of addressing

EA = A + (R)

↳ combines direct addressing &

Register

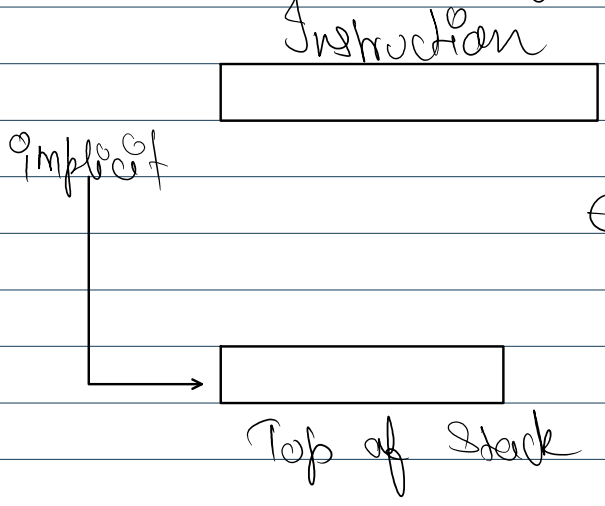
↳ Combines direct addressing & Register in-direct addressing

→ highly flexible mode of addressing

↳ high complexity

most commonly used in
- Relative addressing
- Base Register addressing
- Indexing

(7) Stack Addressing



Thus reference to Stack location is actually Register indirect addressing

→ No memory ref required

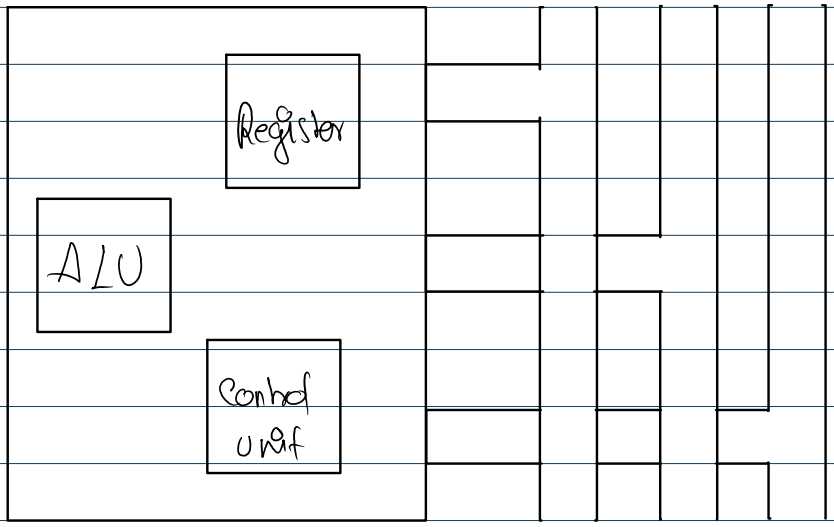
↳ Limited applicability

Processor Organization

Steps that processor must do ;—

1. **Fetch Instruction ;** The processor reads the instruction from the memory
2. **Interpret Instruction ;** Instruction is decoded to determine what action is required
3. **Fetch data ;** The execution of instruction may require reading data from memory or I/O module
4. **Process data ;** The execution of instruction may require performing some arithmetic or logical operation on data
5. **Write data ;** The results of a execution may require writing data into memory or I/O module

To do these functions the processor needs to store some data temporarily
in order to remember the location of last instruction ... etc

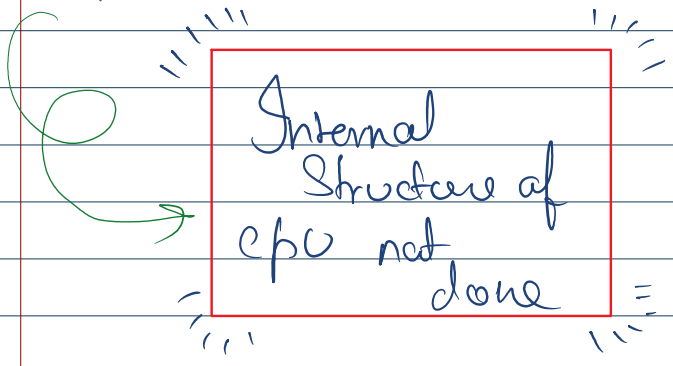


Simplified diag. of CPU

Control Bus | Data Bus | Address Bus

System Bus

imp



Register Organization

Register

General Purpose

- AX Accumulator
- BX Base
- CX Code
- DX data

↓

Used to Store temporary data while instructions are executed

(16 bit length)

divided into two subparts of 8-bit length (high & low)

Segment

- CS Code Seg.
- DS data seg.
- SS Stack seg.
- ES Extra seg.

↓

Code & Instruction are Stored inside different segments

(20 bit length)

↓

16 bit for logical address

Pointer & Index

- SP Stack pointer
- BP Base ptr
- SI Source Index
- DI destination Index
- IP (Instruction ptr)

↓

These Store some address or memory location

Flag

9 flags

(States & Control)

↓

(16 bit length)

tells the states of the process / processor

General purpose Registers

	D15 D8	D7 D0
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

General Purpose Registers

AX ; Accumulator ; 16 bit Register devided into two parts High & Low

AH (8-bit) , AL (8-bit)

generally used for arithematic & logical instruction

ADD AX, AX (AX = AX + AX)

BX ; Base Register ; 16-bit register , also devided into BH, BL

Used to store offset values MOV BL, 500H (BL = 500H)

CX ; Counter Register ;

Used in looping & Rotation

```
MOV CX, 0005
LOOP
```

DX ; Data Register ;

Used in multiplication , input/output port addressing

MUL BX (DX, AX = AX * BX)

Segment Register

It is a 20-bit wide physical address to access 1MB memory location.

As the microprocessor's register has only 16-bit registers, thus the memory is segmented

1. Data Segment (DS) ; Used to point to base address of data segment
2. Code Segment ; Used to point to base address of code segment
3. Stack Segment ; Stores the base address of stack memory segment
4. Extra Segment ; Stores the base address of Extra Memory segment

Pointer & Index Registers

These are 16-bit register, each of them is associated with with each segment register, in order to generate 20 bit physical address

Stack pointer	SP
Base pointer	BP
Source Index	SI
Destination Index	DI
Program Counter / Instruction Pointer	IP

(1) Stack pointer; holds the address of stack top (uppermost stack location)
There is no direct access to the register, modifications are done depending contents of stack

(2) Base pointer; primarly used in accessing parameters passed by the stack

(3) Source Index; Used as pointer addressing data as a source in some string related operation

(4) Destination Index; Used as pointer addressing data as a destination in some string related operation

(5) Instruction pointer; Gives the offset address of next instruction to be fetched from the code segment

When reset, it fetches the first instruction to

be performed.

8086 has 20bit memory which cannot be accessed via 16 bit segment Registers alone.

In order to access memory, 16-bit address is offset with respect to segment

So, physical address + offset

Suppose data segment holds the base address as 1000h & the data you need is present on 0020H memory location (offset) of data segment, the calculations are

- Left Shift 16 bit address present in segment by 4-bits

0001 0000 0000 0000 (0000)

- Add 16-bit offset address to shifted base address

```
  0001 0000 0000 0000 0000
+       0000 0000 0010 0000
-------------------------------
  0001 0000 0000 0010 0000
```

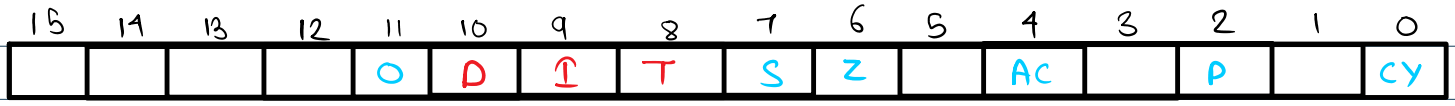
So the actual address turns out to be 10020h

Flags in 8086

Flags are special purpose Registers, which depending upon value of result after arithematic or logical operation the flag bit becomes either (0) or (1)

- Flags
  - Status flag
    1. Sign Flag (S)
    2. Zero flag (Z)
    3. Auxiliary Carry flag (AC)
    4. Parity flag (P)
    5. Carry flag (CY)
    6. Over flow flag (O)
  - Control flag
    1. Directional flag (D)
    2. Interrupt flag (I)
    3. Trap flag (T)

16-Bit flag Register



- Empty Spot : Undefined
- 6 Status flag
- 3 Control flag

Status flag

1. Sign flag : If MSB is 1 then number is -ve (1) [Set] as for +ve, MSB 0 (0) [Reset]

② Zero flag : After any arithmatical or logical operation of result ⇒ 0 (00)H

then, zero flag [Set] (1)

③ Auxiliary Carry flag : Used in BCD number system

after arith/logical operation D(3) generates any carry & passes onto D(4) then the flag is [Set]

Only flag not accessible by programmer

④ Parity flag : If the result is in even parity
ie.
even number of 1's, then the flag is [Set]

⑤ Carry flag : Carry is generated when performing n-bit operation, the result is more than n bits
If so then the flag is [Set]
↳ also called borrow flag

⑥ Overflow flag : flag is set when, result of signed operation is too large to fit in available no. of bits

Control flags

① Directional flags : used in string Instructions

- If Set = 1 ; access memory from higher memory location towards lower memory location

vice versa

② Interrupt flag : This flag is for Interrupt

- If Set = 1 ; processor will recognise Interrupt

request from peripherals

vice versa

(3) Trap flag : Used for On-chip debugging

- If Set = 1 ; puts processor into single step mode for debugging

  (or)

  generate automatic internal interrupt after each instruction

vice versa.

8086 Instruction Set

1. Data transfer Instruction
2. Arithmatic instructions
3. Bit manuplation instruction
4. String Instruction
5. Program execution transfer instruction
6. Processor Control Instruction
7. Iteration Control Instruction

Data transfer Instruction

used to transfer data from source operand to destination operand

- MOV, POP, PPUSH, PUSHA, POPA (for word)
- IN, OUT (Instruction for Input/output port transfer)
- LEA, LDS, LES (Ins. for address transfer)
- LAHF, SAHF, PUSHF, POPF (Ins. for flag transfer)

Arithematic Instructions

Used to perform arithmetic operations like addition, subtraction, multiplication, division

; ADD, ADC, INC, AAA, DAC (Addition operation)

; SUB, SBB, DEC, NPG, AAS (Subtraction operations)

; MUL, IMUL, AAM (Multiplication operations)

; DIV, IDIV, AAD (Division operations)

Bit Manuplation Instructions

Used to perform operation where data bits are involved

; NOT, AND, OR, XOR, TEST (for logical operations)

; SHL, SHR, SAL, SAR (for shift operations)

; ROL, ROR, RCR, RCL (for rotate operations)

String Instruction

Basically all operations involving strings

; REP, MOVS, COMS, INS, OUTS, SCAS, LODS

Program Execution & transfer Instruction

(or Branch & loop Instruction)

Used to transfer/Branch Instruction during execution

- transfer the instruction during execution without condition

  ; CALL, RET, JMP

- transfer the instruction during execution with condition

  ; JAE/JBE, JC/JNC, JE/JZ, JGE JGNE,

  etc.

Processor Control Instruction

Used to control processor by setting/resetting carry flag values

; STC, STD, STI, CLC, CLD, CLI

Set flag — Clear flag

Iteration Control Flag

Used to control the number of times the instruction is executed

; LOOP, LOOPE/LOOPZ, JCXZ

~NE ~NZ

Interrupt Instructions

These are used to call Interrupt during program Execution

; INT, INTO , IRET

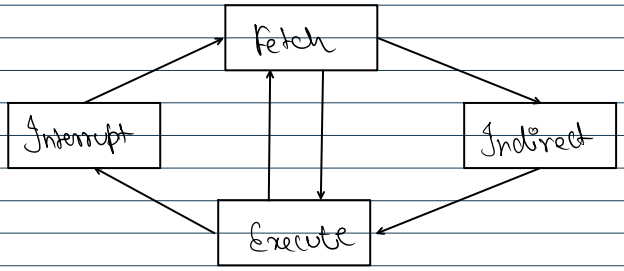
Instruction Cycle

Instruction cycle consists of

1. **Fetch** ; Reading the instructions from memory into processor
2. **Execute** ; Interpret the opcode & perform the indicated operation
3. **Interrupt** ; If Interrupt is enabled and an Interrupt occurs then save the current process state & service the interrupt

Execution of an instruction may involve one or more operands in memory
↓
each of which requires memory access

Thus Indirect cycle / Addressing is required



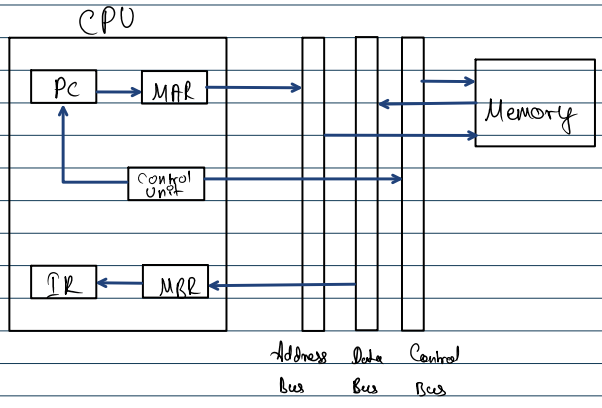
Indirect is just another step in which required operands are fetched through indirect addressing

Fetch cycle

→ Instruction is read from memory
→ The program counter contains the next instruction to be fetched
→ Then this address is moved to MAR and placed on address bus
→ The control unit requests a memory read, and the result is placed on data bus & copied into MBR (Memory Buffer Register) then moved to IR (Instruction register)
→ Meanwhile the PC (program counter) is incremented by 1

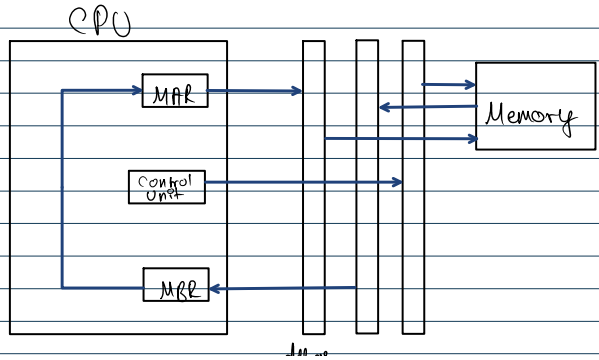
Then next fetch cycle is prepared (by control unit)

↳ Once the fetch cycle is over CU examines the contents of the IR to contains operand Specifier using indirect addressing, if so then an indirect cycle is performed

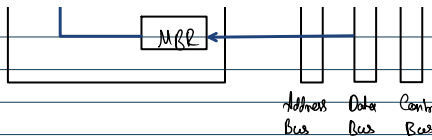


Execute cycle

→ The execute cycle takes many forms
↳ The forms depend on various machine instruction in the IR (Instruction Register)
→ The cycle involves transfering data among register read or write from memory or I/O, and/or the invocation of ALU.



→ The cycle involves transfering data among register read or write from memory or I/O, and/or the invocation of ALU.



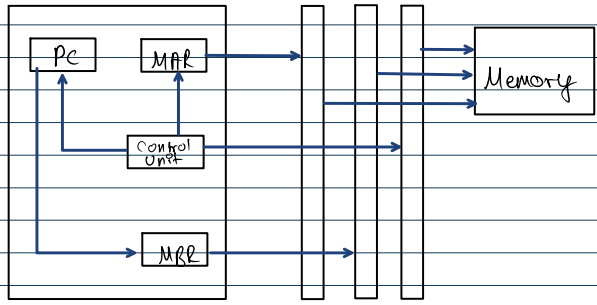
Interrupt Cycle

→ Similar to fetch cycle interrupt cycle is simple & predictable in nature

→ The current content of PC (program counter) must be saved so that the processor can resume normal activity after the interrupt

→ Thus the content of PC are transfered to the MBR to be written into memory

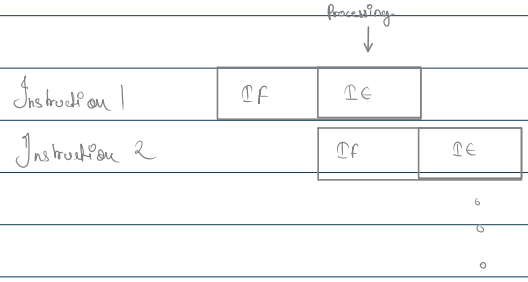
→ Special memory location reserved for this purpose is loaded into the MAR from the control unit



Instruction pipelining

Instruction processor can be divided into ① Instruction fetch
② Instruction Execute

On pipelining, suppose first half of instruction (fetch) is done while the second half is being done during that time can be saved if during this time first half of another instruction can be done



Due to Instruction prefetch, more buffer / registers are required.

as more data for storage is required

If Instruction fetch & Instruction decode required same time the execution time would be halved but that is not the case

As, Execution requires more time than fetch

If there are more stages the processes can be speed up

① 6 Stage Instruction pipelining

1) Instruction fetch (IF) : Read the next expected instruction into buffer
2) Decode Instruction (DE) : Determine opcode & Operand specifier

3) Calculate Operands (CO) : Calculate effective address of each source
4) Fetch Operands (FO) : Fetch each operand from memory
5) Execute Instruction (EI) : Perform indicated operation
6) Write Operation (WO) : Store the result in memory

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO
Instruction 10														

Time →

① 5 Stage Instruction pipelining

1) Instruction fetch (IF)
2) Instruction decode (ID)
3) Instruction Execute (IE)
4) Memory Access (MA)
5) Write back (WB)