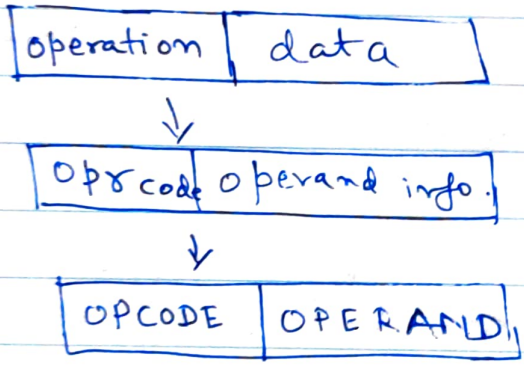
Digital Computer

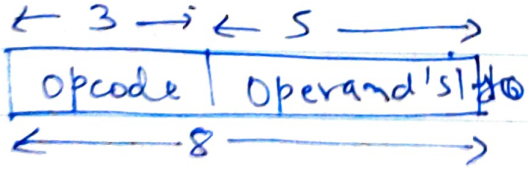
HLL PROG → Lang. Translation (compilation) → LLP (Machine code / 1010 / 111 / Binary code / Bytecode)

* A group of bits which instructs a computer to perform some operation

Instruction



Ex) A CPU has 8 bits instruction



3 bits opcode : 2^3 operations supported

$= 8 \rightarrow 000, 001, 010, \ldots 111$

→ one opcode will denote unique information in CPU

compiler with

Suppose 000 → ADD, 001 - COMP, 010 : AND

Max: 8 distinct instructions can be supported by CPU

Instruction Set Architecture (ISA)

* Collection of all type of instructions which CPU can support

Size of ISA in previous example = 8 (2^3)

Q) 16 distinct instructions supported by CPU so how many OPCODE bits

A) 16, 2^4, so 4 bits

Types of Instructions:

Data Transfer : MOV, LDI (Load immediate), LDA (Load Accumulator)

Arithematic & Logic : ADD, SUB, AND, OR

Machine Control : EI (Enable interrupt)
DI (Disable interrupt), PUSH, POP

Iterative : LOOP, LOOPE (Loop on Equal), LOOPZ (Loop on Zero)

Branch : JMP, CALL, RET, JZ (Jump on zero), JNZ (Jmp on zero)

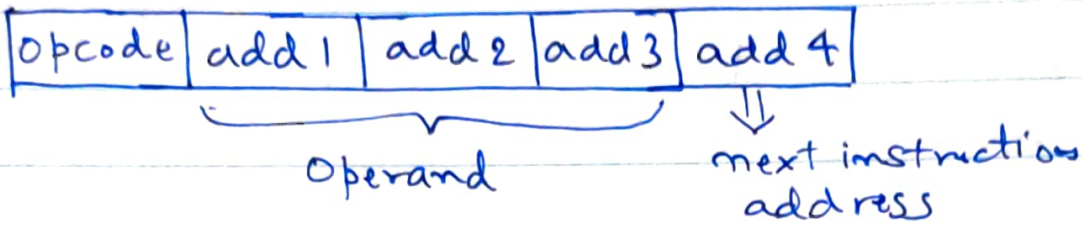
↳ Size of ISA = 19, min. opcode bits = 5 bits

Types of Instruction (based on Operand info)

→ 4-address
→ 3-address
→ 2- "
→ 1 - "
→ 0 - "

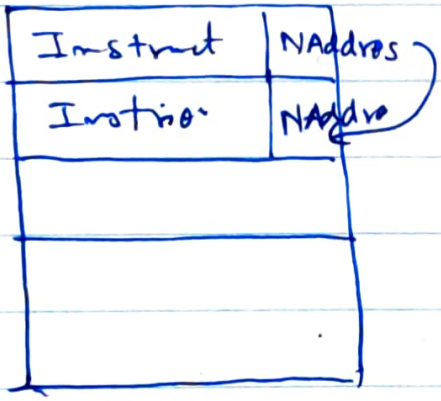
4 - Address Instruction

(*) within each instruction maximum 4-addresses can be specified



(*) But we keep next instruction address in PC, so reason is, these 4-Address instruction PCs don't have Program Counter ".

.. without PC



Disadvantages:

→ Larger instruction size
→ Larger program in memory
→ Instruction fetch takes more time
→ Relocation is not easy

↓

(*) If OS relocates any instruction then all address directions are hampered

3-Address Instruction

opcode	add1	add2	add3

in this PC keeps track of next instruction

- Maximum 3 addresses can be specified within an instruction

→ All disadvantages from previous 4-addres instruction are eliminated

2-Address Instruction

- Max 2 addresses can be specified within an instruction

opcode	add1	add2

Ex) for 3 address

opcod	add 1	add 2	add 3
10101 0	10	01	11
ADD	R2	R1	R3

$R_2 \leftarrow R_1 + R_3$

for 2 address

opcode	add1	add2
101010	10	01
ADD	R_2	R_1

$R_2 \leftarrow R2 + R1$

- One operand is used as source and destination both

2-Address vs 3-Address

Suppose we have to execute

$x = (a+b) * (c+d)$

3-add

$R1 \leftarrow a+b$

$R2 \leftarrow c+d$

$x \leftarrow R_1 * R_2$

2-add (max 2 operands)

Instructions increased

$R_1 \leftarrow a$

$R_1 \leftarrow R_1 + b$

$R_2 \leftarrow c$

$R_2 \leftarrow R_2 + d$

$R_1 \leftarrow R_1 * R_2$

$x \leftarrow R_1$

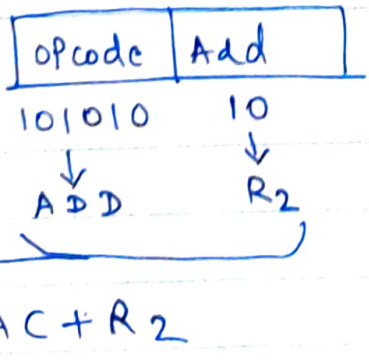
[we can't write $a \leftarrow a+b$ as original 'a' value is lost]

Disadvantages of 2-Add

1) More no. of instn. for a prog as compared to 3-add instr. format
2) More memory for program

So 3-add is preffered everywhere

1-Address Format [Accumulator]



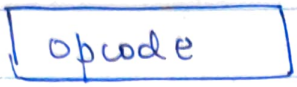
[AC- is used as second operand implicitally]
[AC-based architecture]

Disadvantage:

→ More no. of instruction for a program

0 - Address Instruction

No address mentioned in instruction



10|0|10

↓

ADD [operand ??]

↓ ↳ through Stacks

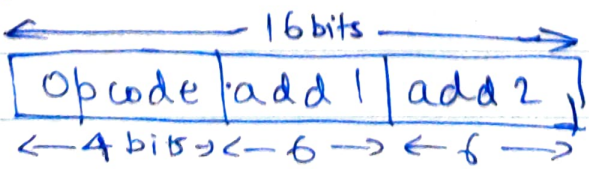
ADD [2 operands are taken from stack]

⊛ If a CPU supports 3-address instruction
=> can support 2-add, 1-add
0-add

⊛ ——— 2-address => 1 add, 0-add

So backward compatible

Q) Consider a digital computer which only supports 2 address instructions each with 16-bits. If address length is 6-bits then maximum and minimum how many instructions the system can support ?



Max) $2^4 = 16$ opcodes, ISA (size) = 16 | MIN: 1

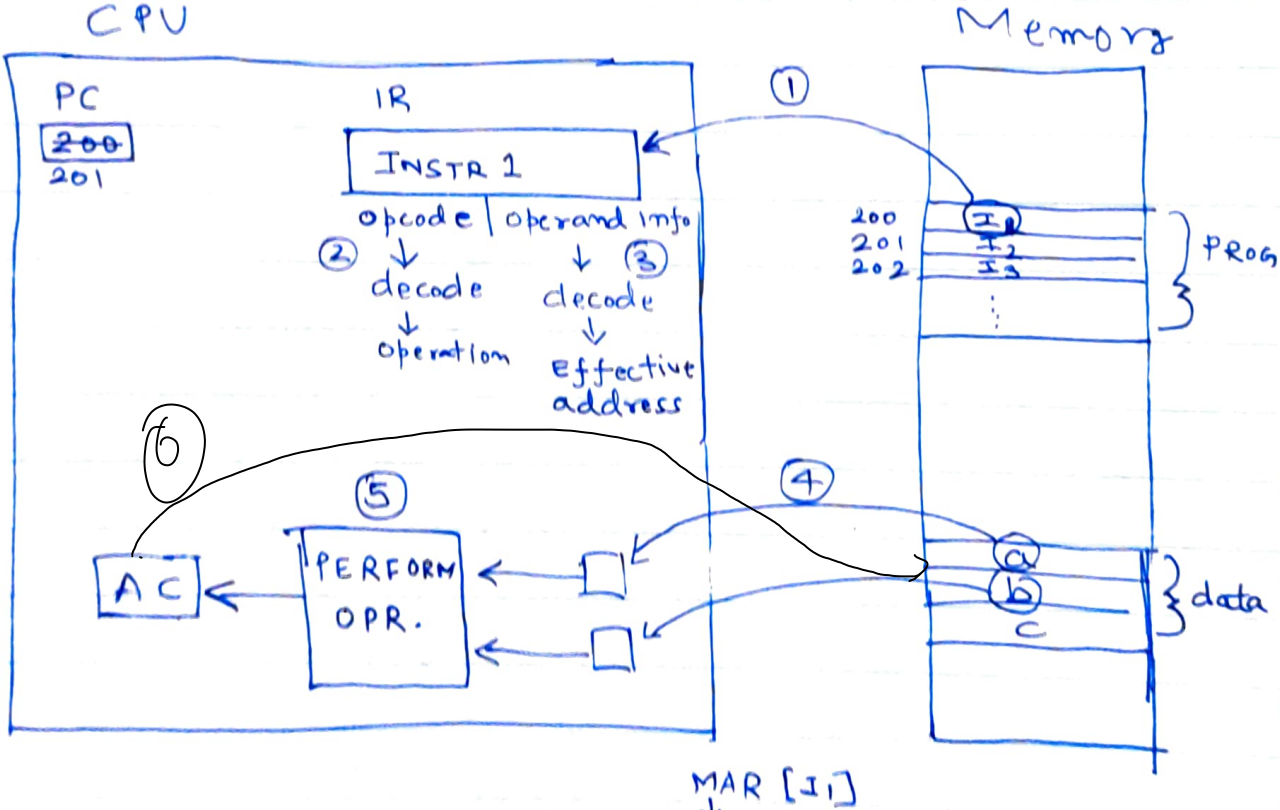
Effective Address

(ADD, SUB, MOV, AND)

* Address of operand in a computation type instruction or
* The target address in a branch type instruction
  - branch → JUMP

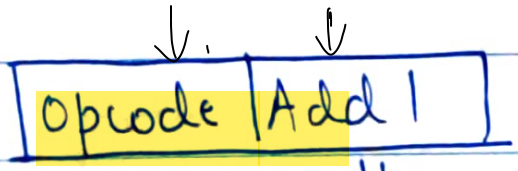
Instruction Cycle

* CPU performs 6 phases to execute an instruction.



1) Instruction fetch ($I_1 \rightarrow MDR[I] \rightarrow IR$)
2) OPcode decoded (Instruction decode)
3) Operand info decoded → Effective address calculated
4) Operand Fetch
5) Execution
6) Copy back result (write back)

Addressing Modes ✓



1101 → INCR

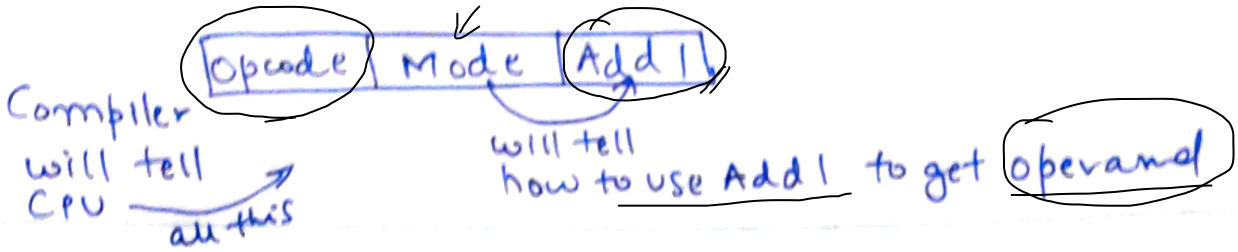
10 → Register = INCR R2
10 → MEM ADD = INCR M[2]
10 → .....

CPU will come to know abt OPERATION using OP decode

But it does not know what this is (confusion)

→ So we need some mechanism

↳ So to avoid confusion "Mode" bit is sent and which tells the CPU that what "10" is and how use it



Addressing modes: it specifies how and from where the operands are obtained for an instruction

Q) Why different addressing modes?

A) As we use / implement data in various ways in our HLL prog. (using LL, arrays, pointer, etc.) so the CPU should also have this flexibility to access operand in various ways. That's why we need diff. addressing modes

Types of Addressing Modes:

1) Implied Mode
2) Immediate Mode
3) Direct Mode
4) Indirect Mode
5) Register Mode
6) Register Indirect Mode

→ non computable mode (no computation (ADD, SUB, INCR) not required)

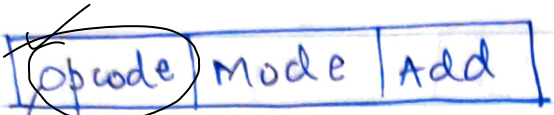
7) Autoincrement + Autodecrement Mode
8) Indexed / Indexed Register Mode
9) PC-Relative Mode
10) Base Register Mode

→ Computable Mode

IMPLIED MODE

opcode defination itself defines the operand

INCR STATUS REG



what if Opcode tells, OPR and OPERAND
↳ this is called as implied mode

ex) Real world example : Father instructing Son to go home

ex) INCA (Increment Accumulator)

* For special purpose register address is not explicity mentioned

will tell what kind of addressing mode is this

001 → Implied Mode

IMMEDIATE MODE

Operand value

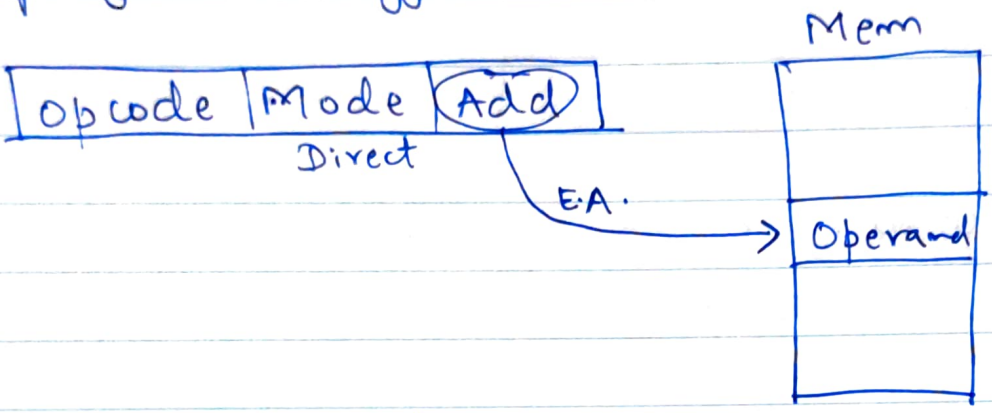
ADD 10 will increment the value stored in the accumulator by 10.
MOV R #20 initializes register R to a constant value 20.

* The address field of instruction specifies the operand value
  ↳ It is used to initialize registers with constant value

* in CPU design only Binary combination will tell what Mode is, ex: 000 → Implied mode
  001 → Immediate mode, etc.

Direct Mode (Absolute Mode)

The address field of instruction specifies the effective address

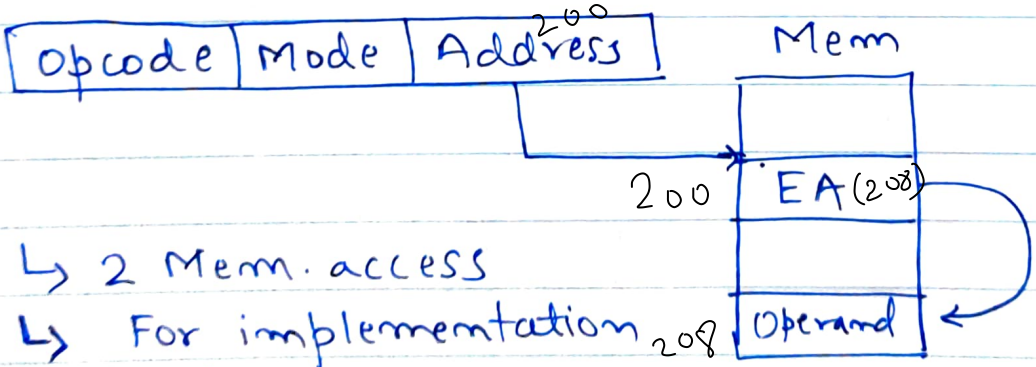


↳ Best mode to access memory operand

ADD X will increment the value stored in the accumulator by the value stored at memory location X.
AC ← AC + [X]

INDIRECT MODE

The address field of instruction specifies the address of effective address



↳ 2 Mem. access

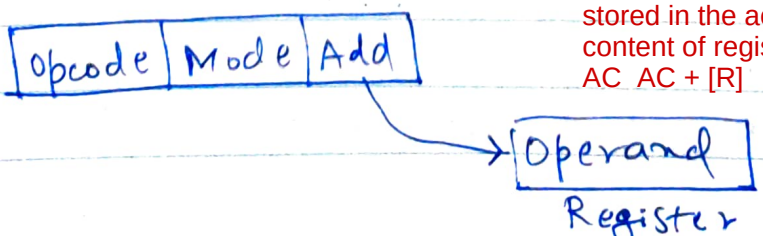
↳ For implementation of pointers

ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.
AC ← AC + [[X]]

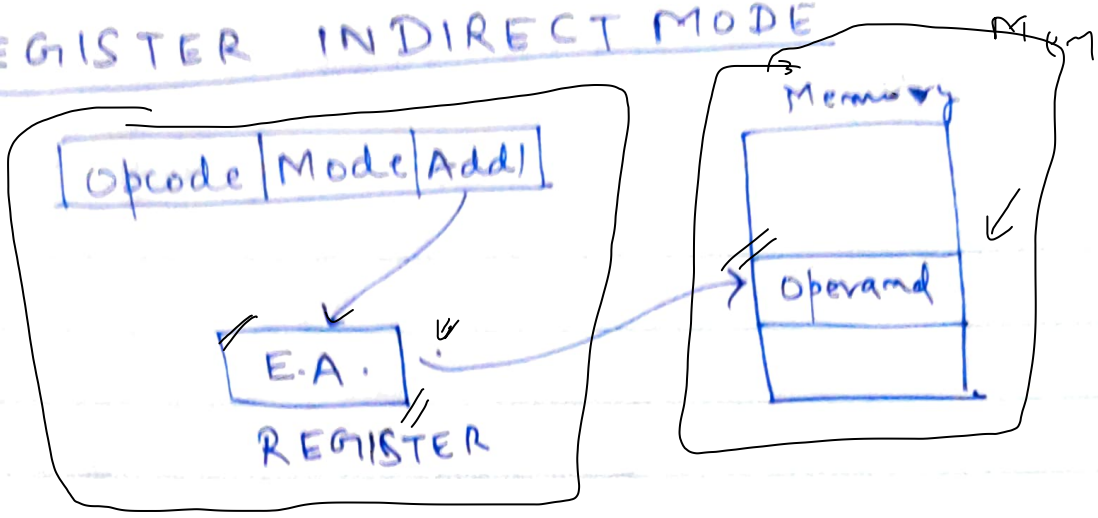
REGISTER MODE (Register direct Mode)

The address field of instruction specifies a register which holds operand

ADD R will increment the value stored in the accumulator by the content of register R.
AC ← AC + [R]



REGISTER INDIRECT MODE



The address field of instruction specifies ~~the address of Effective address of register.~~

- the register which holds the Effective address.

ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.
AC ← AC + [[R]]

Why this mode is used ?

↳ Memory is very large so it is not feasible to store add (Mem) in "Addl", so the EA is stored in specific Register

↳ to shorten the instruction length

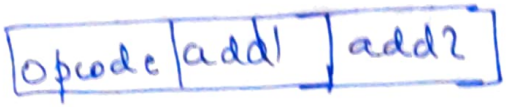
↳ One register + one memory access required to get the operand

↳ Reg access time <<<< Mem access time

↳ So total access time = Mem access time

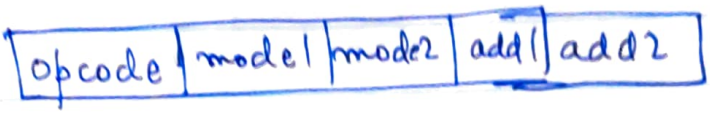
↳ Reg. indirect mode and Reg direct mode take almost same time

↳ but this is only done when CPU supports ~~variable~~ flexible complex instructions.

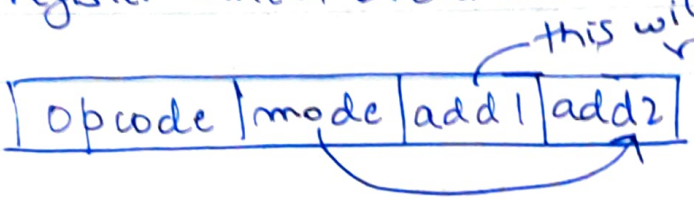
what if 

↳ will ask CPU req. info on operand 1 or add1 or on both?

↳ Yes → then



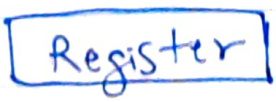
↳ for register - mem architecture

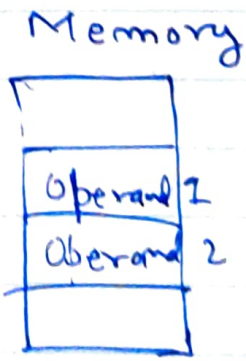


Computable Addressing Modes

Auto increment / Auto decrement Mode

→ variant of register indirect mode in which the contents of register is automatically incremented or decremented to access the sequential data





↳ For accessing array elements continously using a loop we need this addressing mode

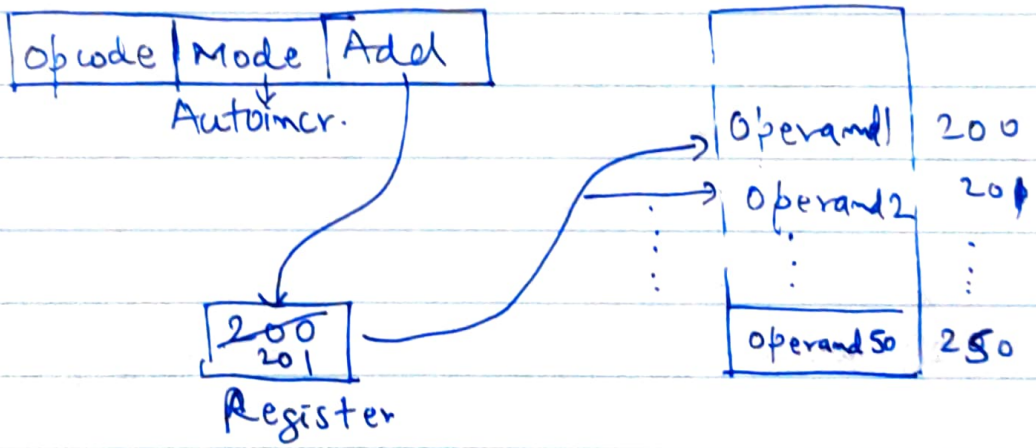
→ Same operation on all elements of the array

Ex:
```
for (i=0; i<50; i++)
{
    Operation on A[i];
}
```
↓

Opcode	add

CPU May have to run 50 instructions or better use auto increment mode

Execution



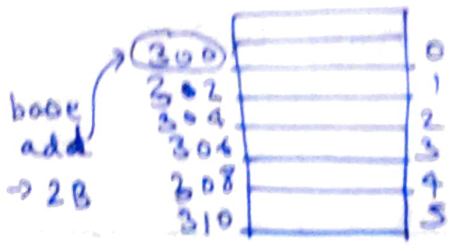
If this sequential data has to be accessed in reverse mode → then we use auto decrement mode.

Auto inc/dec ⇒ Post increment/decrement
Auto dec ⇒ Pre decrement

Indexed or Index Register Mode

↳ to access a special element of array

int A [5];



↳ Suppose, A [3] wants to be accessed

↳ So we need address of A [3]

address of A [3] : Effective Address

$$LOC(A[3]) = Base + Size\ of\ element * 3$$
$$= Base + 2 * index$$
$$= 300 + 2 * 3$$
$$= 300 + 6$$
$$LOC(A[3]) = 306$$

⇓

Compiler generates intr.

⇓

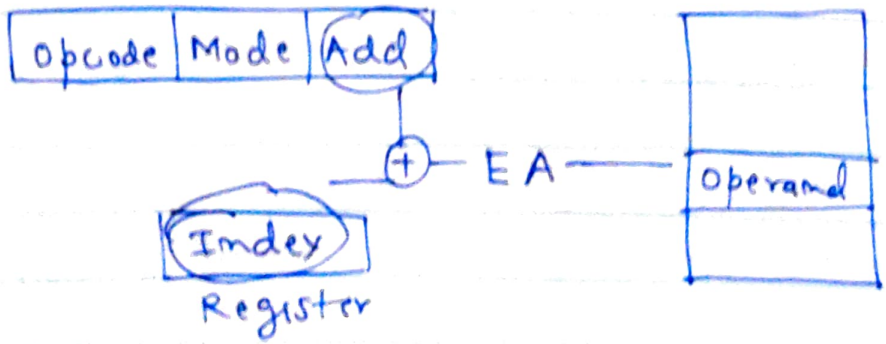
1) index reg. ⇐ 2 * i

2) loc(A[i]) ← M [Base + index reg]

So,

Effective address = Base add + index (2 * i)

Only this needs to be provided



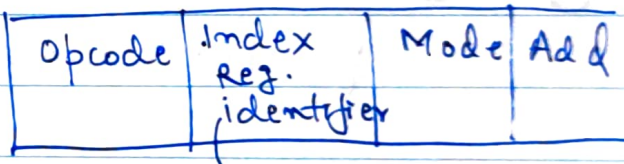
E.A. = Add part of intr. (base add) + index Reg value (index)

⊛ How many time memory accessed
= 1

↳ It does not support relocation because of relocation, then the base add shld be updated in instruction. which is a tedious and costly task.

↳ index Reg

- Special purpose
- General purpose (made as index reg.)



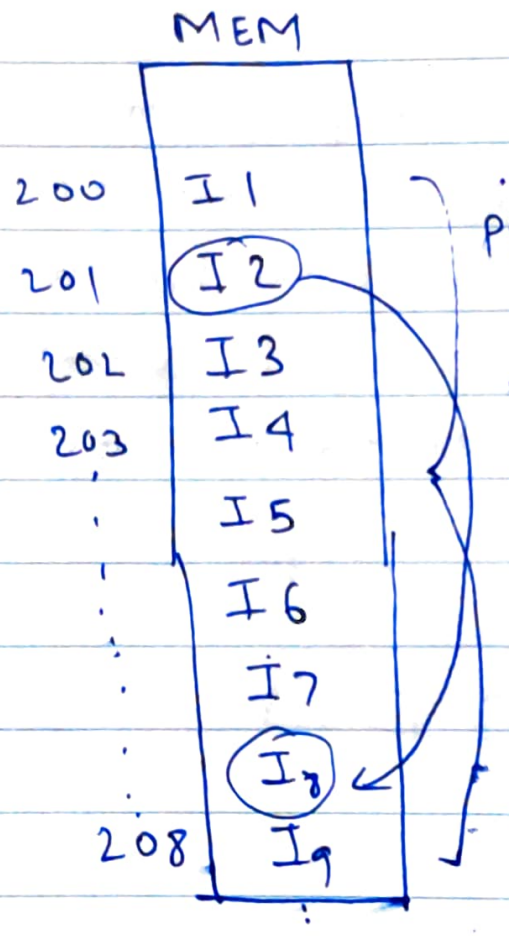
this will tell which GPR is made index reg.

PC - Relative Mode (used in branching)
(intra segment branching)

PC value added in address field value (offset) of instruction to get effective address.

(discussed before)

Branch Instruction



ASSUME

* CPU is executing instr I_2

PC = 202

* After decode I_2 is branch instr.
* Branch due to condition

I_2 branch

TRUE → BRANCH TAKEN

Assume $I_2 \rightarrow I_2$ (JUMP)

Target instr. = I_8 executed

FALSE → I_3

Next in the Sequence instr. I_3 executes

(Branch not taken)

→ No change in PC

Q) How will CPU know it has to jump

A) Change PC, PC → 208 (target address)

I_8 is target instruction.

* Unconditional branch PC → target address always

Q) How will CPU generate target address

Target address = PC + Relative no. of memory locations to skip (offset)

in previous ex.

Target address ⇒ PC = 202 + 5

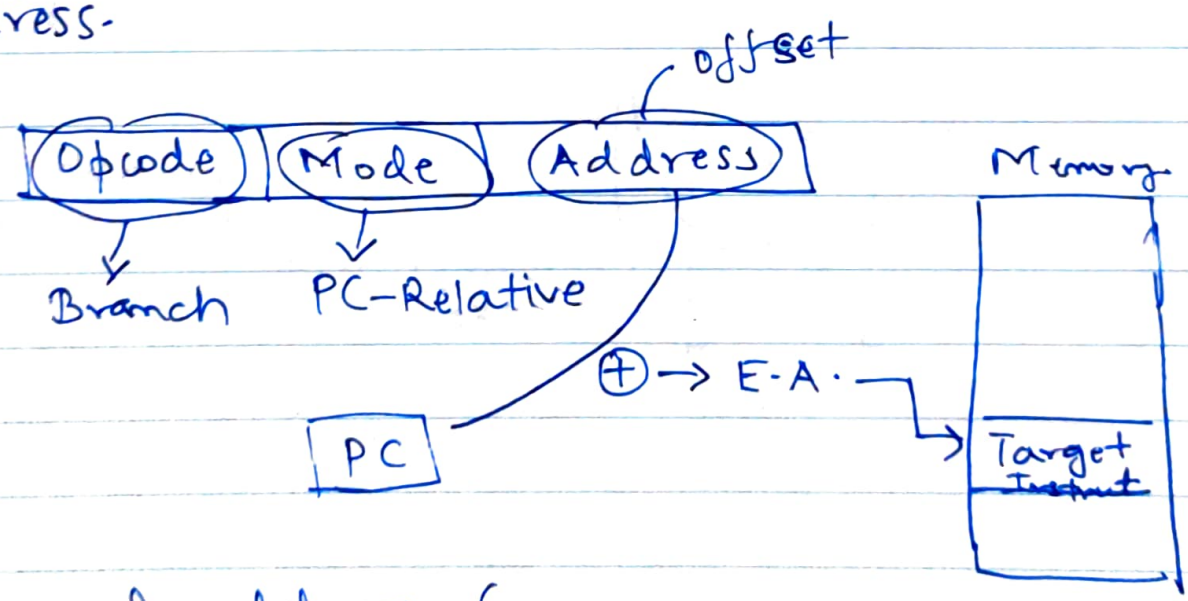
Q) Who generated addresses for each instruction ?

A) Compiler as per the HLL program

↳ So compiler knows offset value

PC - Relative Mode (used for branch type instr)

PC - value added in address field value (offset) of instruction to get effective address.



E.A. Operand address (.

E-A. Target address (branch type instr)

Effective address = PC value + address part value of instn.

IN Branch Instr.

	PC
Before fetch I_2	201
After fetch I_2	202

Decode ⇒ Its a branch instruction

E.A. calculation ⇒ 202 + 5 = 207

Execution ⇒ CPU checks the condition & updates PC if condn is true

(*) If I_8 is branch and going to I_4

So in this case -ve offset value is stored in address of PC relative mode

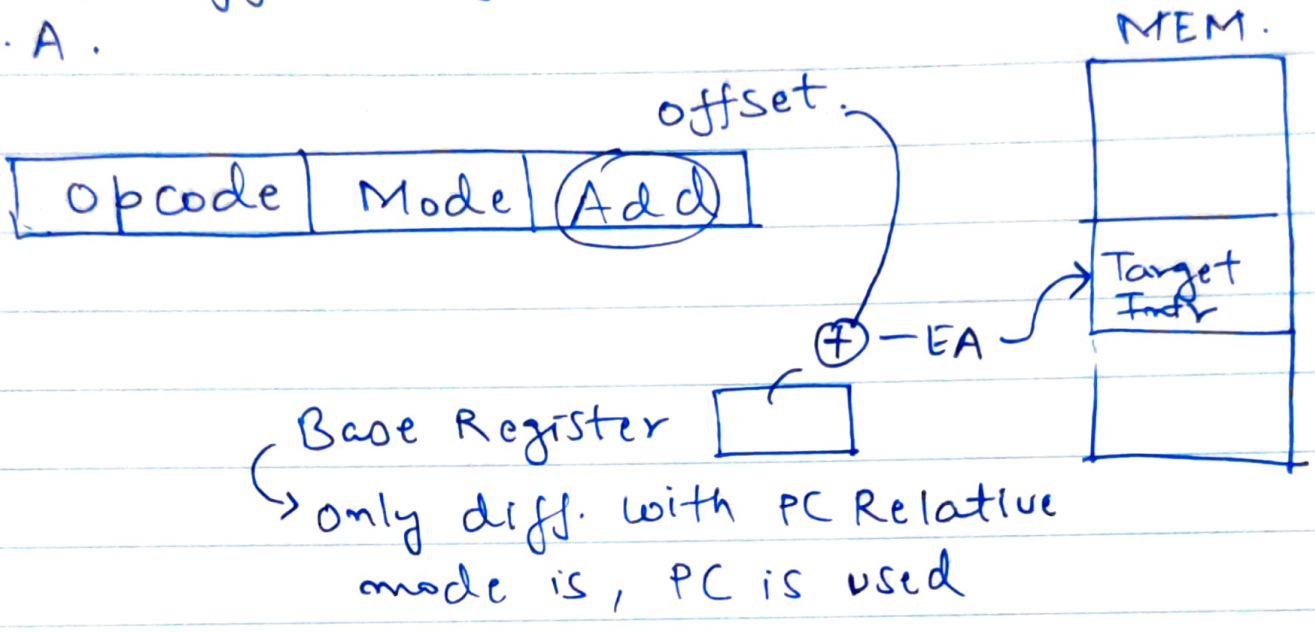
Target address formula will not change

For forward jumping = offset (+ve)

" backward " = offset (-ve)

Base Register Mode (Inter Segment branching)

Base register value added in address field value (offset) of instruction to get E.A.



→ only diff. with PC Relative mode is, PC is used

* PC - Relative Mode also known as position independent mode

E.A. (Base Register Mode)
= Base Reg. value + add of instr.

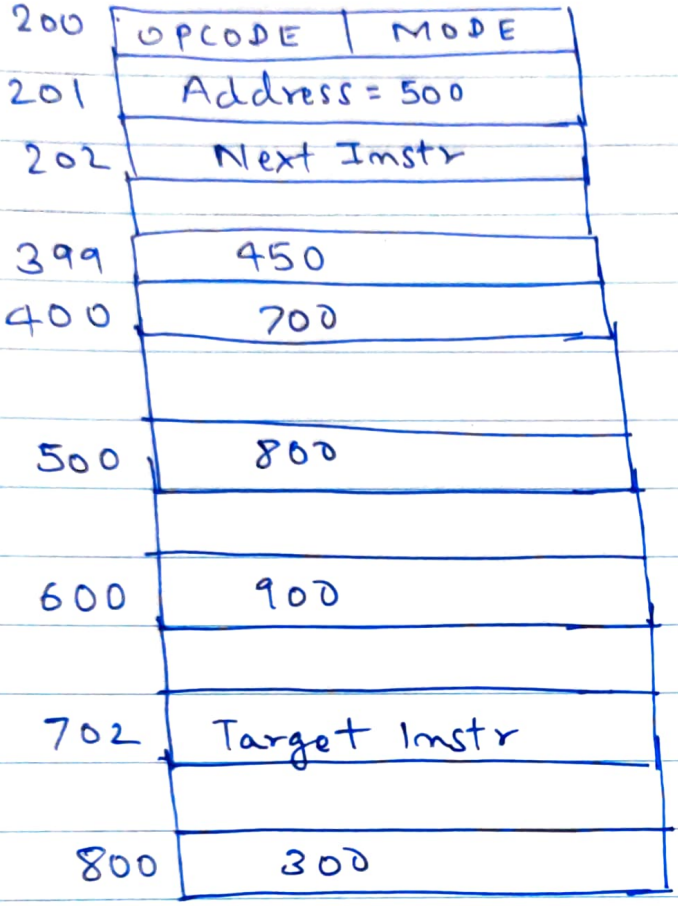
+ PC - Relative, Base register mode supports relocation, also supports branching

*

EXAMPLE

Opcode	Mode	addr

MEMORY



Register

PC - ~~200~~ 202

~~Reg~~ = 400

XR = 100
INDEX REG

AC

1) IF , PC → 200 → PC → 202 (next instruction)

2) Decode
↳ OPCODE

3) E.A. (calc.)

MODE	E A (memory)	OPERAND
1) Immediate Mode	201	500
2) Direct Mode	500	800
3) Indirect Mode	800	300
4) Register Mode	—	400 (assume 500 → Reg)
5) Register Indirect Mode	400	700
6) Auto decrement Mode	399	450
7) Indexed Mode	500+100 = 600	900
8) PC-Relative Mode	500+202 = 702	Target Instr.