Addressing Modes.

- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

— Variety of addressing modes
  ↳ to give programming flexibility to the user.
  ↳ to use the bits in the address field of the instruction efficiently.

1. Implied mode :— In this mode the operands are specified implicitly in the definition of the instruction.

e.g:- CLA, CME, INP ⇓ Input Character

CMA ↓ (current value in accumulator is replaced by its 1's comp.)

opcode	mode	Address

Instruction format with mode field.

EA = AC
EA = SP (Stack)

All register reference instructions that use an accumulator are implied-mode instructions.

By default inst^n knows that from where it has to access the operand.

zero-address instructions

2. Immediate Mode :— operand is specified in the instruction itself.

↳ operand field ↓

It contains the actual operand to be used in conjunction with the operation specified in the instruction.

- fast to acquire an operand.

Inst^n

opcode	operand

→ Inst^n itself contains operand.

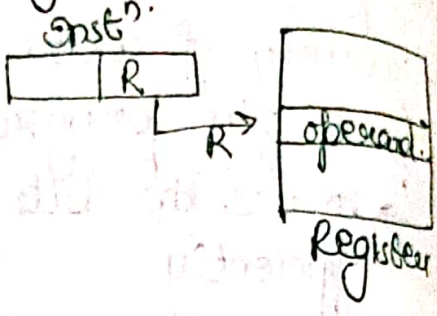
---

ow to calculate the effective memory address of an operand y using info. field in registers.

3. Register Mode :— operands are in registers that reside within the CPU.

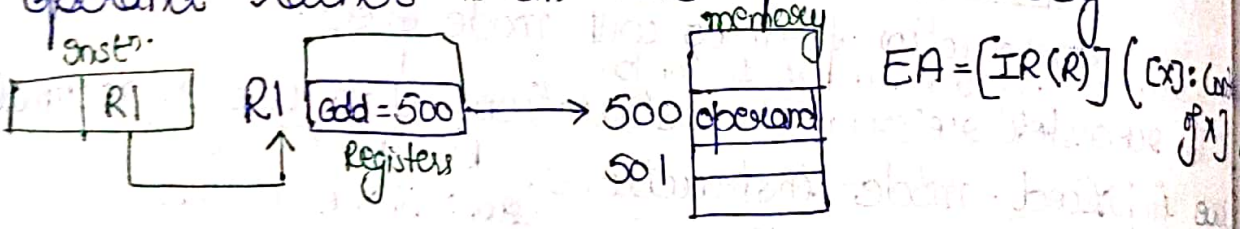
eg:- R1 | 500 ← address of operand.

- Shorter address than the memory address.
- EA = IR(R)
  register field of IR.



4. Register Indirect Mode :— Instruction specifies a register which contains the memory address of the operand.

- Selected register contains the address of the operand rather than the operand itself.



5. Auto increment (Auto indexed) mode :— (R1)+

6. Autodecrement mode :— −(R1)

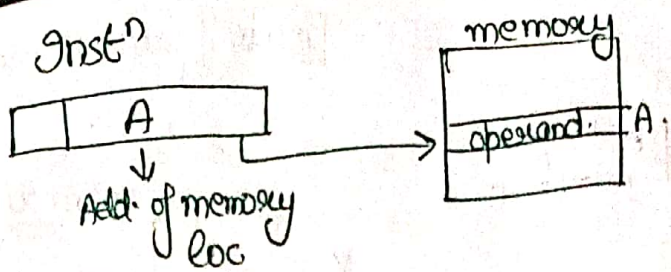
} when address in the register is used to access memory the value in the register is inc. or dec. by 1 automatically.

7. Direct mode :— Inst^n specifies the memory address which can be used directly to access the memory.

EA = IR(addr)

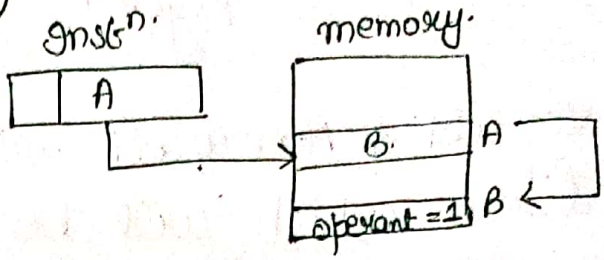
(IR(addr) : address field of IR)

- faster than the other memory addressing modes.
- operand resides in memory & its address is given directly by the address field of the instruction



8. Indirect Addressing modes :—

- Address field of the inst^n gives the address where the effective address is stored in memory.



- Control fetches the inst^n from memory & uses its address part to access memory again to read the effective address. (to get Phone no of X $\xrightarrow{\text{Call}}$ Phone no of Y ↓ He knows Phone no of X)

9. Relative Addressing modes :— In this mode

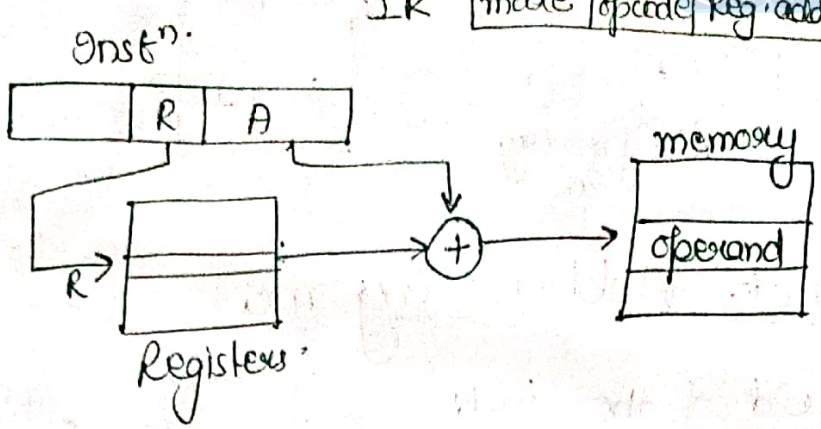
the Content of the program counter is added to the address part of the instruction in order to obtain the effective address.

- —EA = f (IR (address), R)

3 different Relative addressing modes depending on R.

PC Relative addressing mode (R = PC)

EA = PC + IR (address)



Reg. add & Memory add will be added to give physical address which will be mapped onto the memory to access operand. ⟶ known as Displacement addressing

Displacement addressing — Relative R = PC
— Base register R = Base add. 2000 + 20. 2020
— Index register R = IXR. starting base add of m

Relative ⟶ R = PC

EA = PC + IR (add.)

PC — Contains — 825
IR(add) — 24

Instⁿ at location 825 is read from memory during fetch phase & PC is then ↑inc. by one to 826.

EA for relative = 826 + 24 = 850.
add.

Base register addressing mode (R = BAR → Base reg. addr.)

EA = BAR + IR (addr.)

e.g.: BAR = 2000 ; IR (addr) = 20 ; E.A = 2000 + 20 = 2020 ; (memory addr.)

Indexed Address mode (R = IX → Index Reg.)

EA = IX + IR (address).

Addressing mode example

PC = 200

R1 = 400

XR = 100

AC

Address	
200	Load to AC mode
201	Address = 500
202	Next Instn
399	450
400	700
401	
500	800
600	900
702	325
800	300

Relative

IR (add) − 500

202

702

Index reg.

EA = 100 + 500

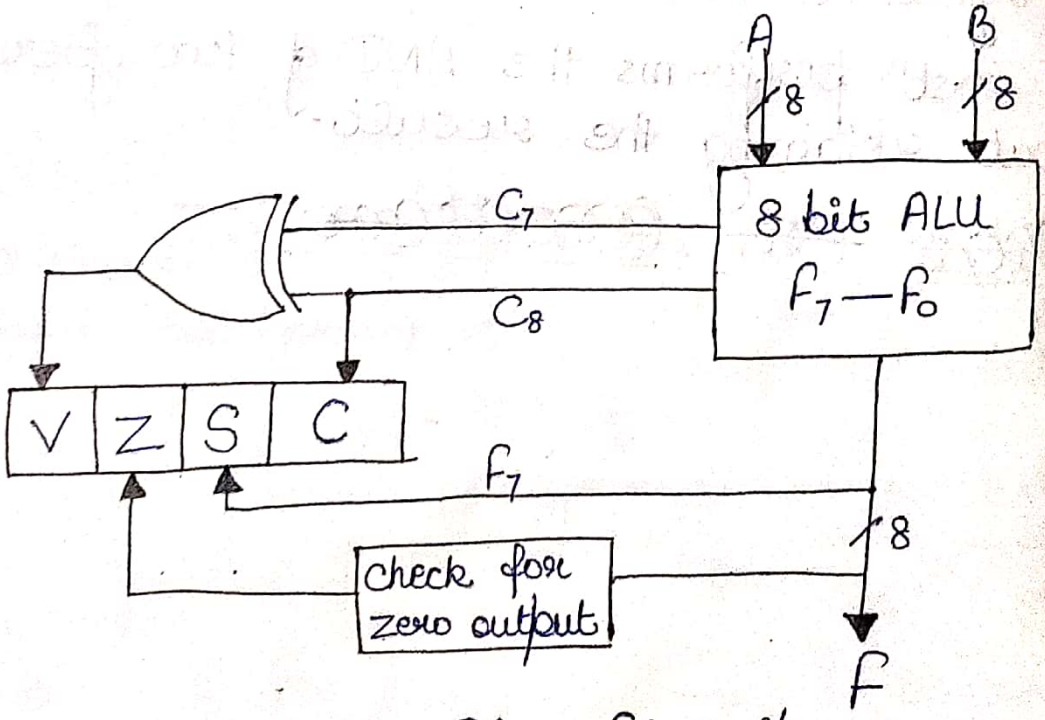
Addressing mode	effective address		Content of AC
Direct add.	500	AC ← 500	800
Immediate operand	—	AC ← 500	500
Indirect add.	800	AC ← 800	300
Relative "	702	AC ← PC + 500	325
Indexed "	600	AC ← XR + 500	900
Register	—	AC ← R1	400
" indirect	400	AC ← R1	700
Auto inc.	400	AC ← (R1)+	700
Auto dec.	399	AC ← −(R1)	450

Conditional Branch Instruction

- Each mnemonic is constructed with letter B (for branch)
- Letter N (for no) is inserted to define the 0 state.

Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0
	Unsigned compare conditions (A - B)	
BHI	Branch if higher	A > B
BHE	Branch if higher or equal	A ≥ B
BLO	Branch if lower	A < B
BLOE	Branch if lower or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B
	Signed compare conditions (A - B)	
BGT	Branch if greater than	A > B
BGE	Branch if greater or equal	A ≥ B
BLT	Branch if less than	A < B
BLE	Branch if less or equal	A ≤ B
BE	Branch if equal	A = B
BNE	Branch if not equal	A ≠ B

Processor Status Word (PSW)



Status Flag Circuit

• Status bit Conditions can be stored for further analysis

• Status bits are also called Condition-Code bits or flag bits.

• Diagram — 8 bit ALU with 4 bit status register
— 4 Status bits are C, S, Z & V.
— The bits are set or cleared as a result of an op" performed in ALU.

• In basic Computer, the processor had several (status) flags – 1 bit value that indicated various info. about the processor's state – E, FGI, FGO, I, IEN, R

• In some processors, flags like these are often combined into a register – the processor status register also called a processor status word (PSW). (PSR)

• Common flags in PSW are :—

→ C (carry) : set to 1 if end carry C_8 is 1.
Cleared to 0 if carry is 0.

→ S (sign) : Set to 1 if MSB F_7 is 1.
" " 0 " " " " 0

→ Z (zero) : Set to 1 if output of ALU contains All 0s
Cleared to 0 otherwise.

→ V (overflow) : is set to 1 if the XOR of the last two carries is equal to 1 & cleared to 0 otherwise.

For 8 bit ALU, V = 1, if output is greater than +127 or less than −128. (range of signed no's)

Unsigned no's → 0 – 255.

• The subtraction of two no's is the same whether they are unsigned or in signed — 2's Complement representation

• Let A = 11110000 and B = 00010100

To perform A−B, the ALU takes 2's Complement of B and adds it to A.

$A - B = A + \bar{B} + 1$

1 1 ← last two Carries

A : 11110000

$\bar{B}+1$: +11101100

A−B : 1 | 10111100

C_8 S F_7 F_0

B : 00010100

$\bar{B}$: 11101011

+ 1

11101100

∴ C = 1, S = 1, V = 0, Z = 0

V = 0 because last two carries are both equal to 1.

⇒ If we assume unsigned no's

A = 240 (decimal equivalent)

B = 20

A−B = 240 − 20 = 220.

Binary result 11011100 ≡ 220

Since 240 > 20 ∴ A > B and A ≠ B

The inst^n that will cause a branch after this Comparison are
- BHI (Branch if Higher)
- BHE (" " or equal)
- BNE (" not equal)

- If we assume signed number
decimal equivalent of A = -16
- The sign of A is -ve & 11110000 is the 2's Complement of 00010000, which is decimal equivalent of +16.
decimal equivalent of B = +20.
- Subtraction (A-B) = (-16) - (+20) = -36.
- Binary result 11011100 (2's Comp. of 00100100) is indeed the equivalent of decimal -36.
- Since (-16) < (+20) we have A<B & A≠B.
- The inst^n that will cause branch after this comparison are
  - BLT (Branch if less than)
  - BLE (" " " or equal)
  - BNE (" " not equal).

Subroutine Call and Return

- Subroutine → a self-Contained sequence of inst^{n}s that perform a given computational task.
During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program.
- Transfer program Control to a subroutine is known
  - ↳ call subroutine
  - ↳ branch to subroutine
  - ↳ branch & save address