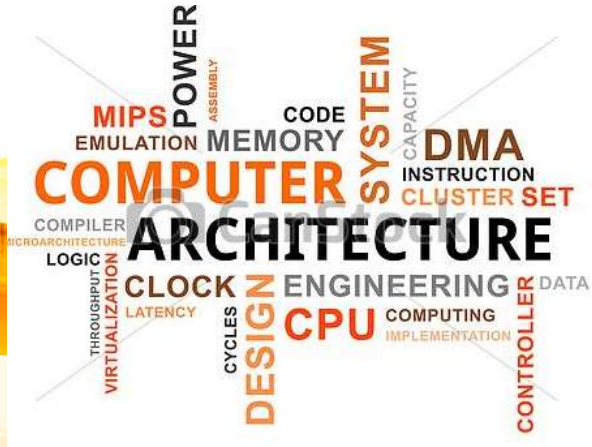
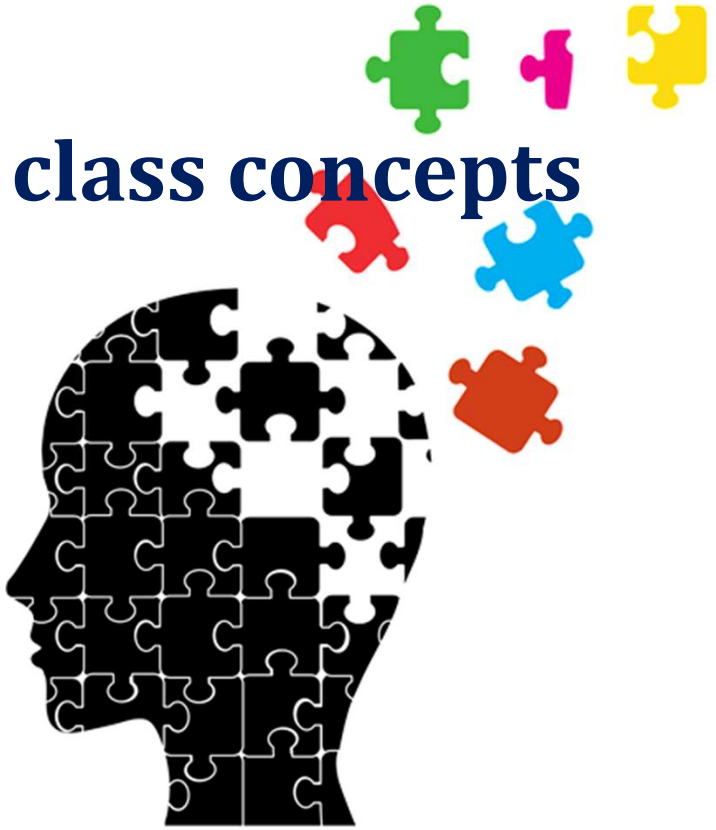
UNIT I
BASIC STRUCTURE OF COMPUTERS

Functional units – Basic operational concepts – Bus Structures – Performance – Memory locations and addresses – Memory operations – Instruction and Instruction sequencing – **Addressing modes – Assembly language** – Case study : RISC and CISC Architecture.



Recall the previous class concepts



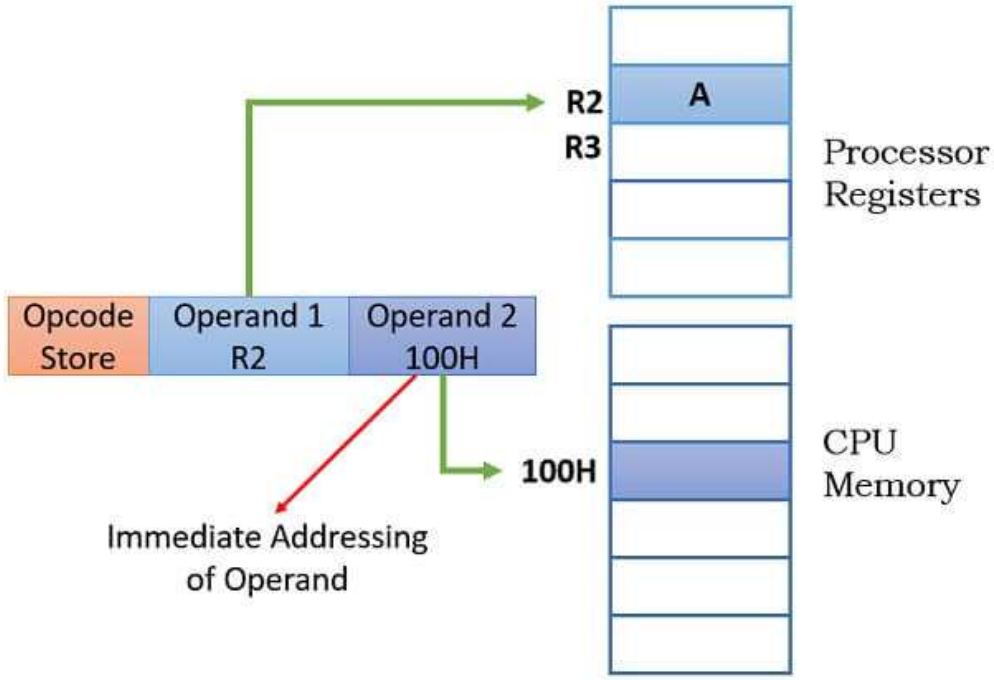
Addressing modes

- The different ways of specifying the location of an operand in an instruction are called as **addressing modes**.
- **Starting address** of memory segment.
- **Effective address or Offset**: An offset is determined by adding any combination of three address elements: **displacement, base and index.**
  - **Displacement:** It is an 8 bit or 16 bit immediate value given in the instruction.
  - **Base**: Contents of base register, BX or BP.
  - **Index**: Content of index register SI or DI.

Immediate addressing

- The operand is given explicitly in the instruction
- Example

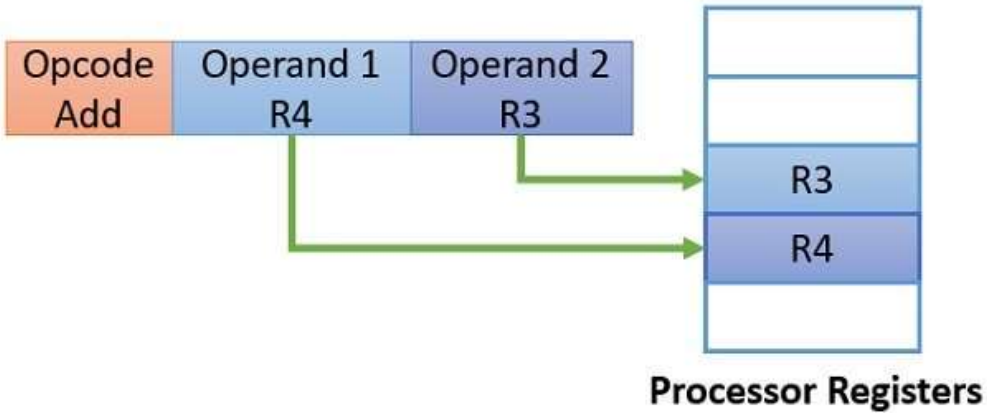
  Add #6,R1 R1 <-6+R1



Register addressing

- The operand is the contents of a processor register in the instruction
- Example

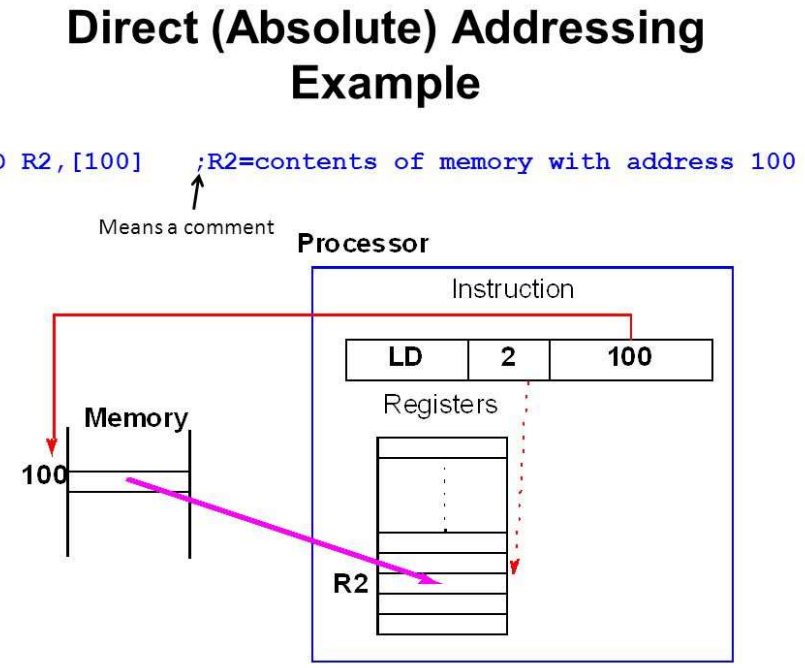
  Add R3,R4 R4 <- R4 + R3



Absolute(direct address) addressing

- Address of the operand is in the instruction
- Example

  Add B,R4 R4 <- R4 + B

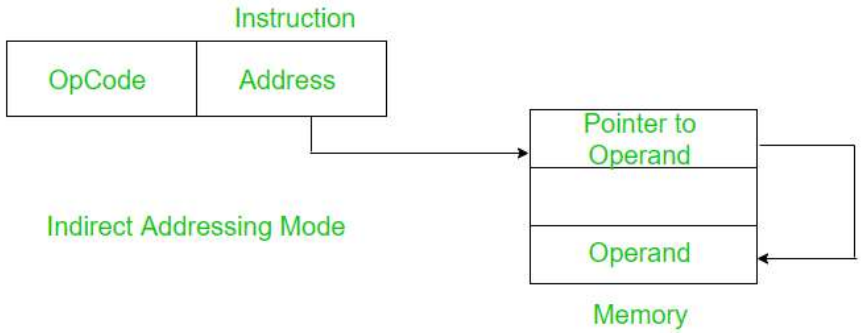


Indirect addressing

- The address of the operand is the contents of a register or memory location whose address appears in the instruction.
- Example

Add (R1),R0 R0 <-R0+M[M[R1]]

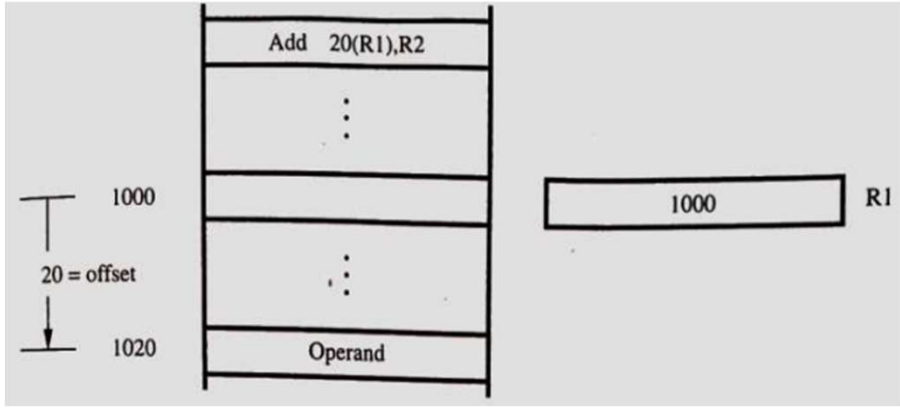
Add (A),R0 R0 <- R0+M[M[A]]



Index addressing

- The effective address of the operand is generated by adding a constant value to contents of a register.
- Example

Add 20(R1),R2 R2<- R2 +M[20+R1]



Base with Index addressing

- Add (R1 + R2), R4 R4 <- R4 + M[R1 + R2]]

  #R2 base register, R1 index register

Base with index and offset addressing

- Add 6(R1 + R2), R4 R4 <- R4 + M[6+R1 + R2]

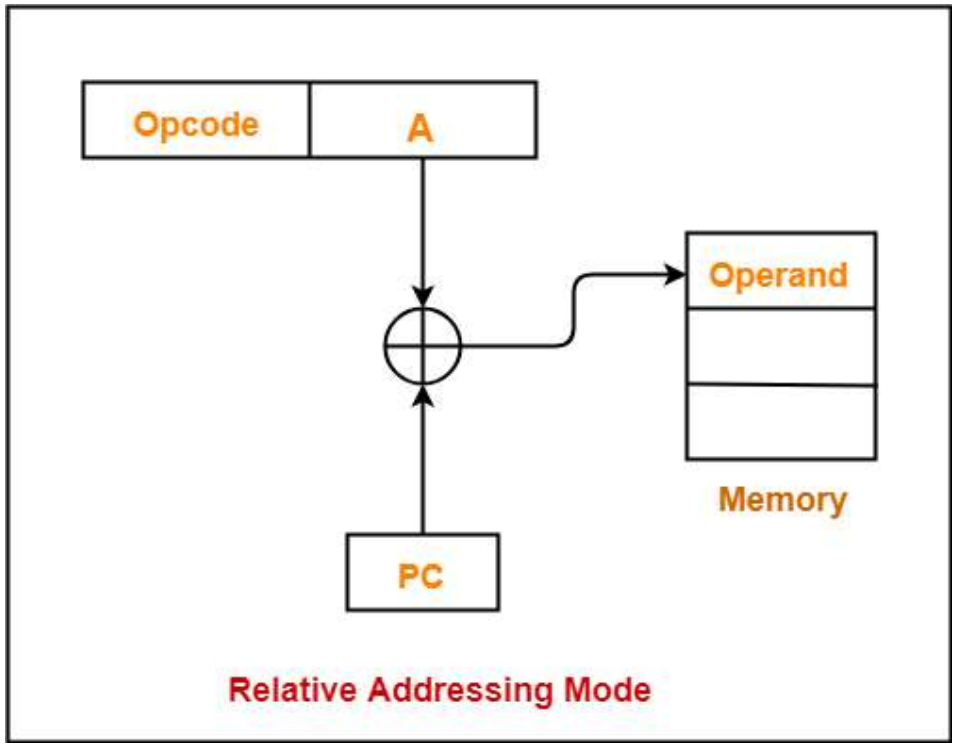
  #R2 base register, R1 index register

Base with index and offset addressing

- Add 6(R1 + R2), R4 R4 <- R4 + M[6+R1 + R2]

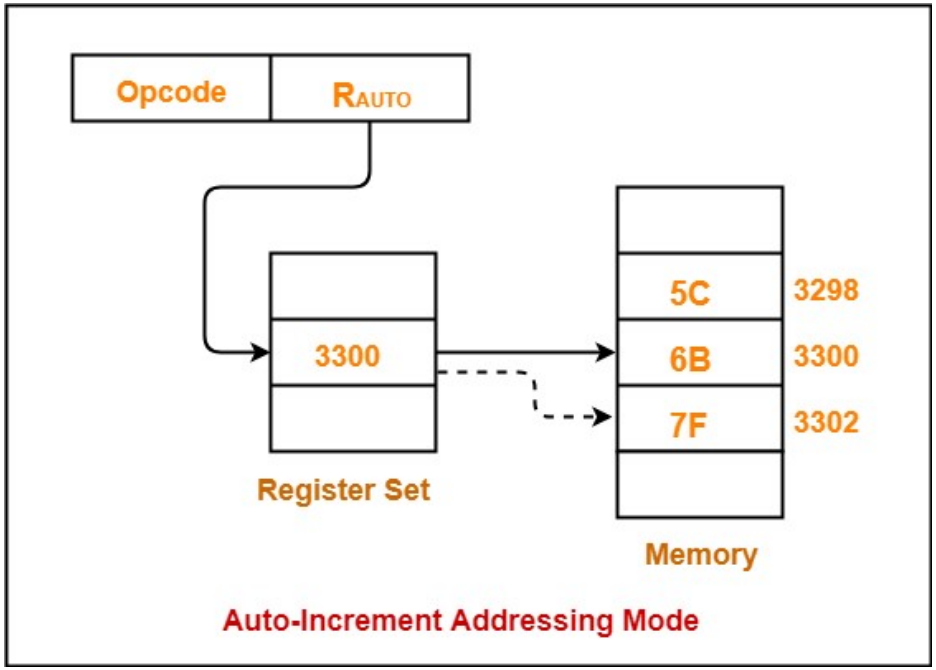
  #R2 base register, R1 index register

Relative Addressing Mode

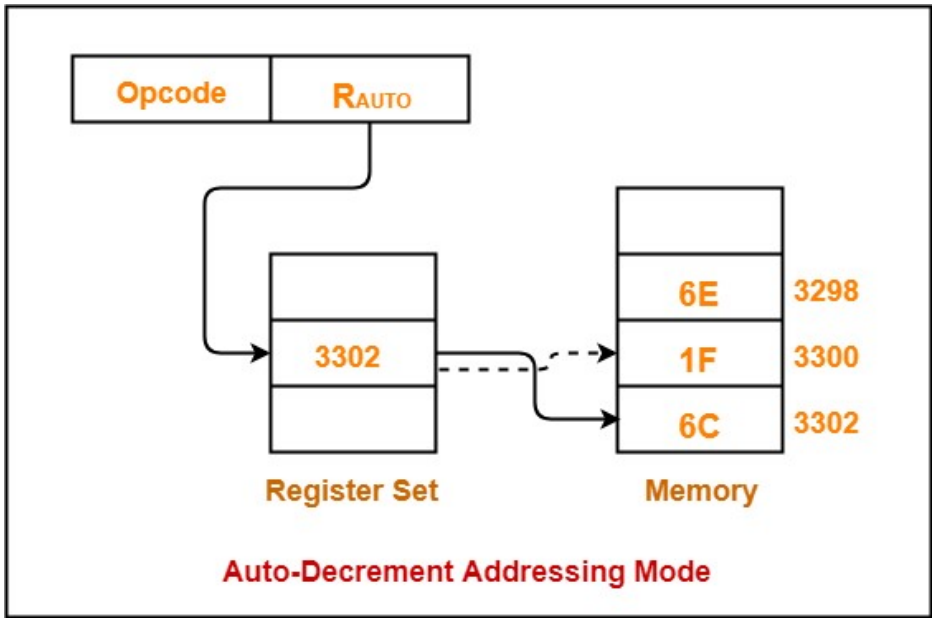


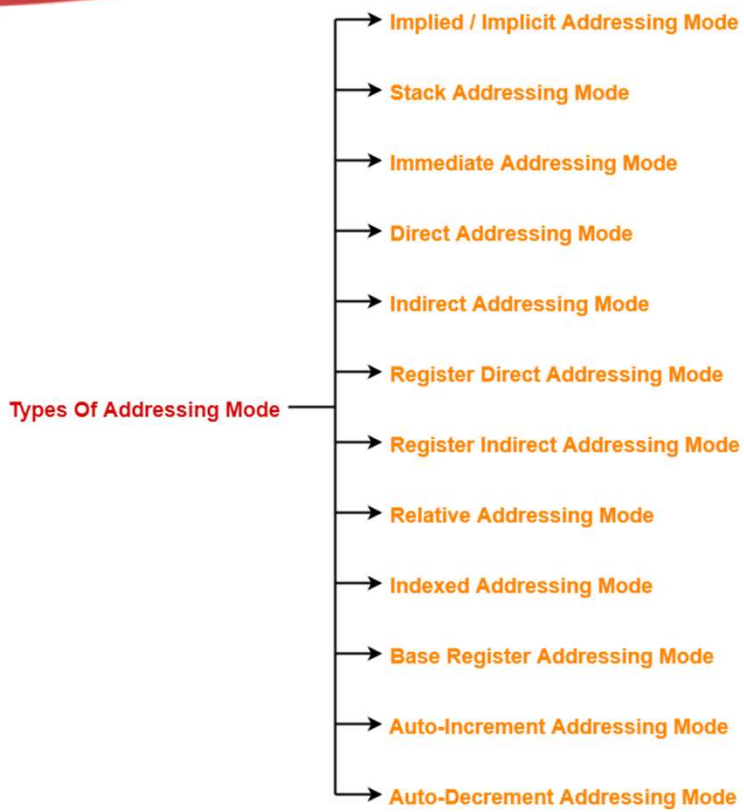
Effective Address
= Content of Program Counter +
Address part of the instruction

Auto-Increment Addressing Mode



Auto-Decrement Addressing Mode



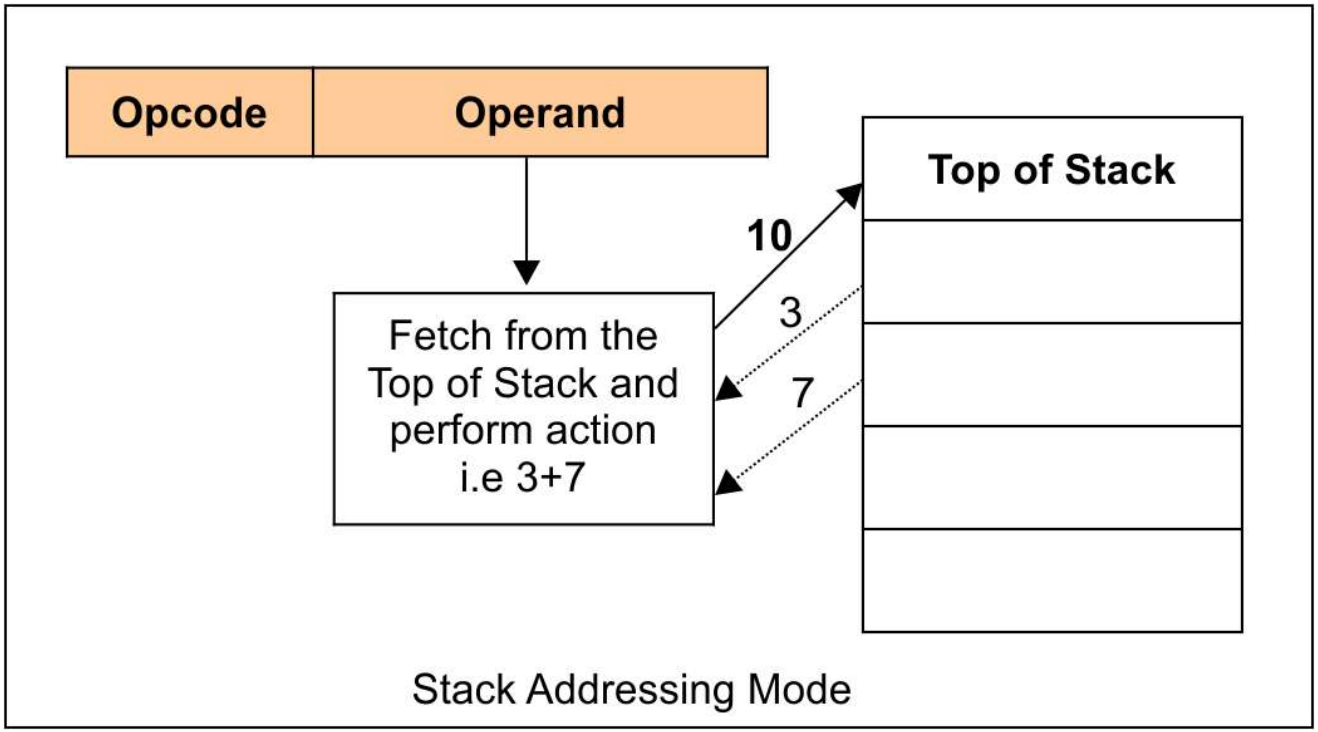


1. Implied / Implicit Addressing Mode
2. Stack Addressing Mode
3. Immediate Addressing Mode
4. Direct Addressing Mode
5. Indirect Addressing Mode
6. Register Direct Addressing Mode
7. Register Indirect Addressing Mode
8. Relative Addressing Mode
9. Indexed Addressing Mode
10. Base Register Addressing Mode
11. Auto-Increment Addressing Mode
12. Auto-Decrement Addressing Mode

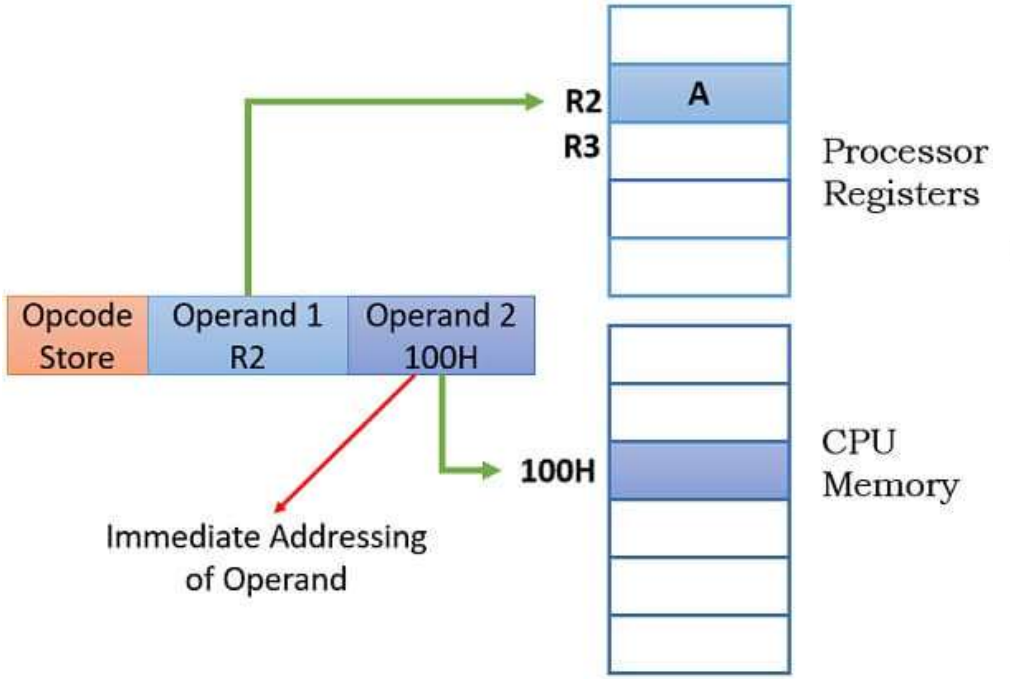
Implied / Implicit Addressing Mode

Example: MOV AL, 35H (move the data 35H into AL register)

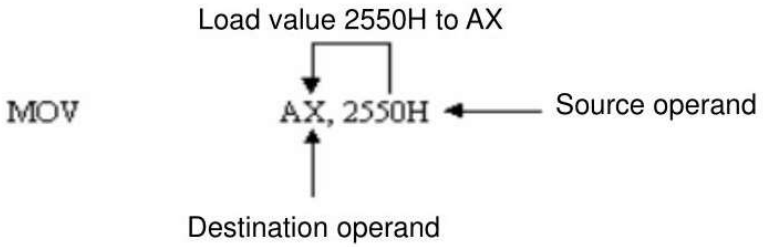
Stack Addressing Mode



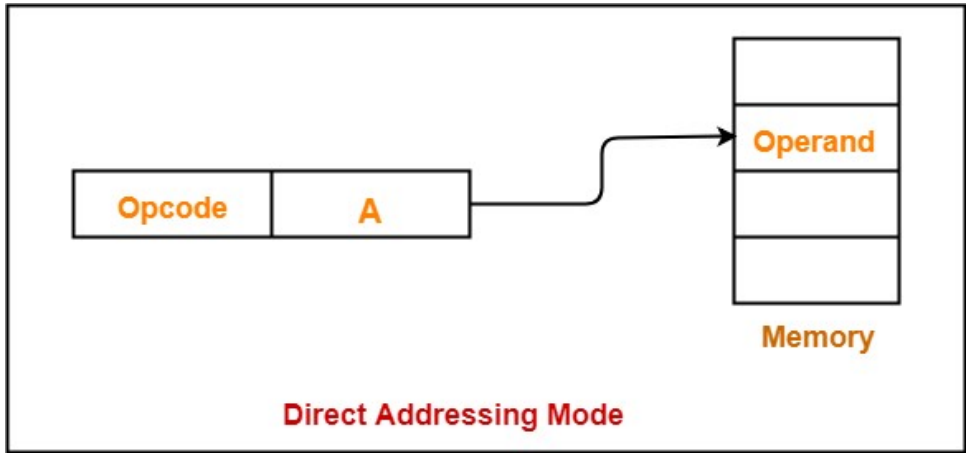
Immediate Addressing Mode



Example:



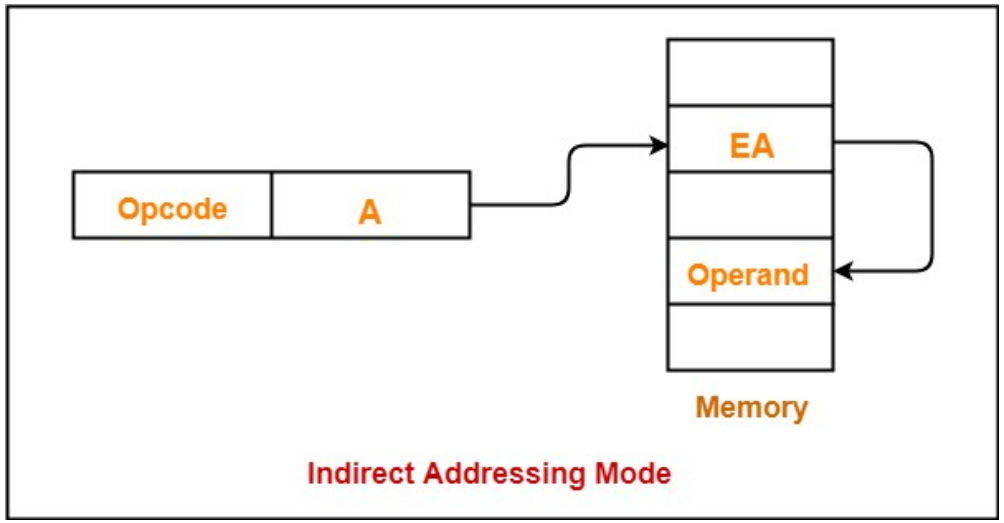
Direct Addressing Mode



ADD X will increment the value stored in the accumulator by the value stored at memory location X.

$AC \leftarrow AC + [X]$

Indirect Addressing Mode

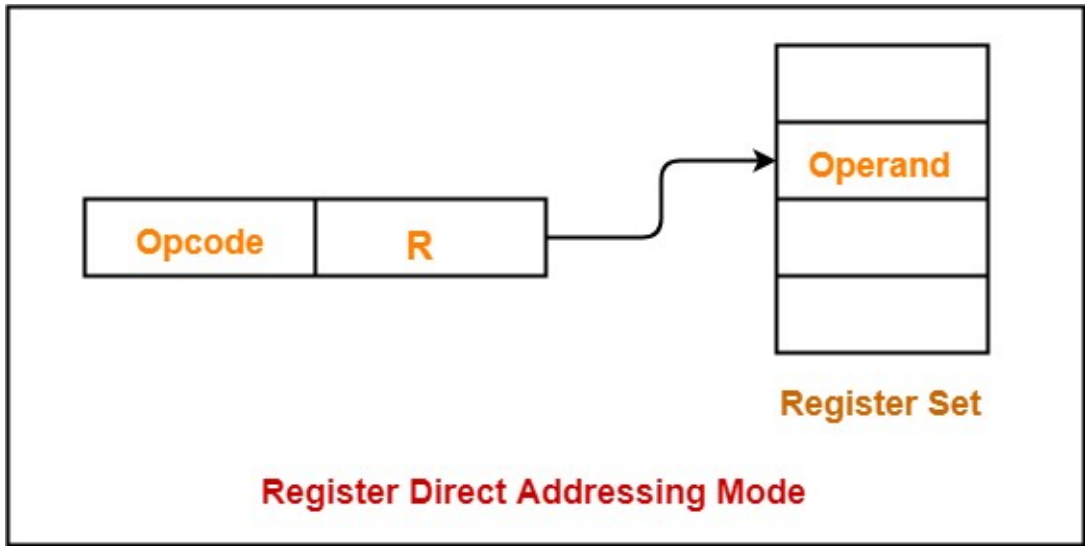


Indirect Addressing Mode

ADD X will increment the value stored in the accumulator by the value stored at memory location specified by X.

$AC \leftarrow AC + [[X]]$

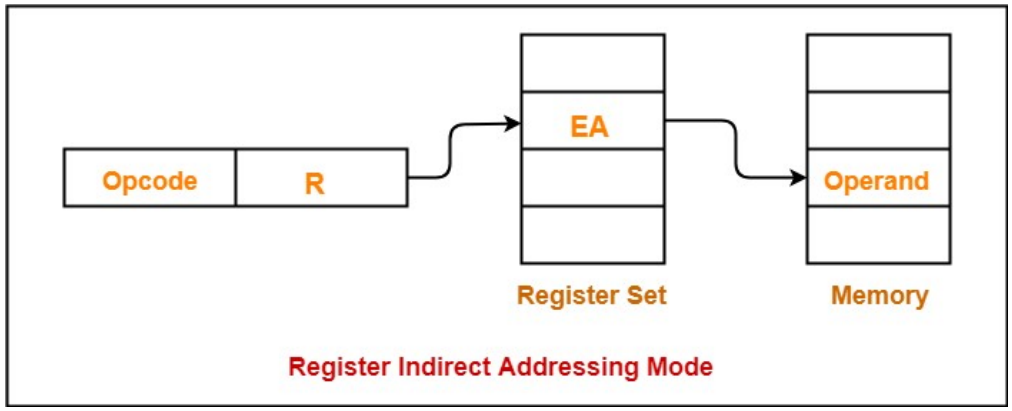
Register Direct Addressing Mode



ADD R will increment the value stored in the accumulator by the content of register R.

$AC \leftarrow AC + [R]$

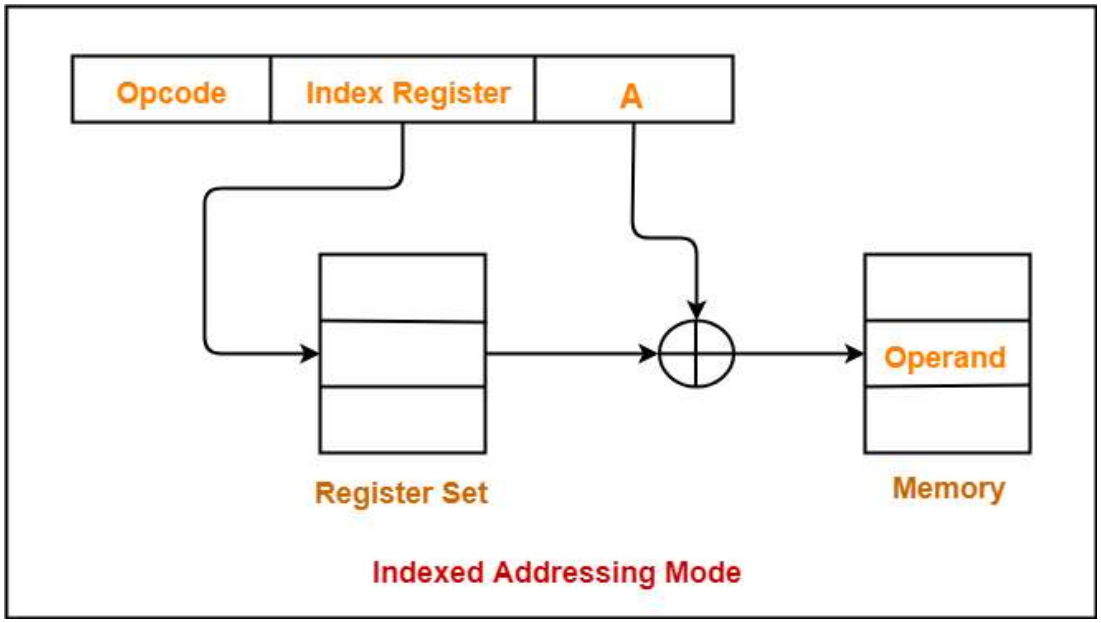
Register Indirect Addressing Mode



ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

$AC \leftarrow AC + [[R]]$

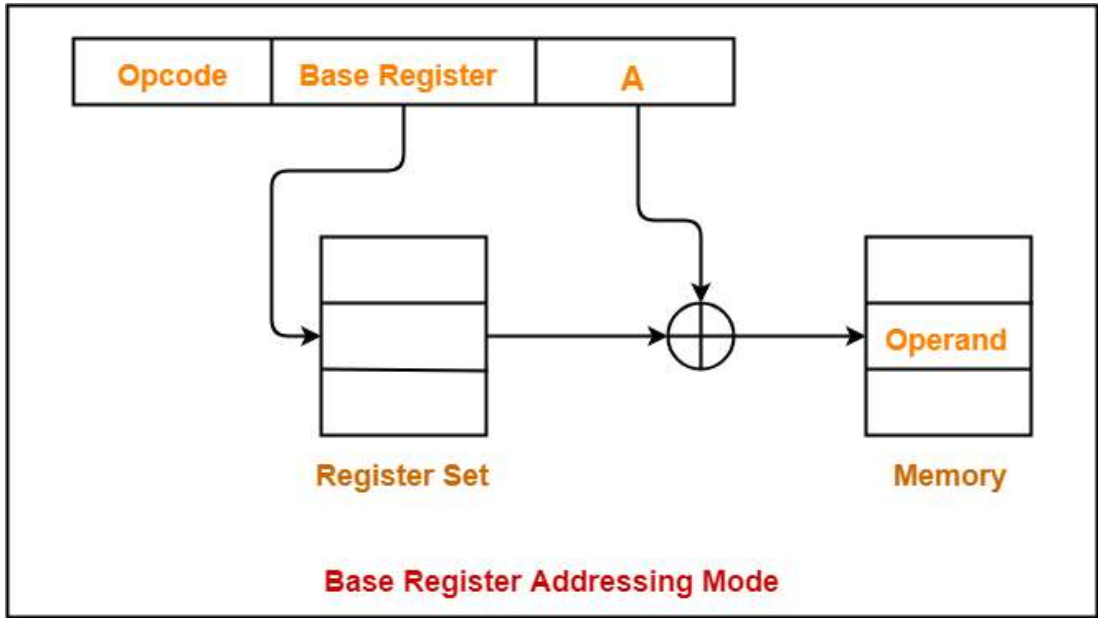
Indexed Addressing Mode



Indexed Addressing Mode

Effective Address
= Content of Index Register +
Address part of the instruction

Base Register Addressing Mode



Effective Address
= Content of Base Register +
Address part of the
instruction

Addressing Modes	Applications
Immediate Addressing Mode	To initialize **registers to a constant value**
Direct Addressing Mode and Register Direct Addressing Mode	To access **static data** To implement **variables**
Indirect Addressing Mode and Register Indirect Addressing Mode	To implement **pointers** because pointers are memory locations that store the address of another variable To pass **array as a parameter** because array name is the base address and pointer is needed to point the address
Relative Addressing Mode	For program relocation at run time i.e. for position independent code To change the normal sequence of execution of instructions For **branch type instructions** since it directly updates the program counter
Index Addressing Mode	For **array implementation or array addressing** For records implementation
Base Register Addressing Mode	For writing relocatable code i.e. for relocation of program in memory even at run time For **handling recursive procedures**
Auto-increment Addressing Mode & Auto-decrement Addressing Mode	For **implementing loops** For stepping through arrays in a loop For implementing a stack as push and pop

Assembly language

- Assembly language is a type of programming language that communicates with the hardware of a computer.
- Hardware from different manufacturers uses machine language, like binary or hexadecimal characters, to perform tasks.

Assembly language

- Assignment Statement is f = (g + h) - (i + j). What is the compiled MIPS code?

f,g,h,i,j is assigned to r0,r1,r2,r3,r4

Temp register r5,r6

```
add  r5, r1, r2
add  r6, r3, r4
sub  r0, r5, r6
```

Assembly language

Convert the following C Language into MIPS Assembly Language A[30] = h+ A[30]

```
lw  $t0, 32($s4)    # load word
add $t0, $s2, $t0
sw  $t0, 32($s4)    # store word
```

Index 8 requires offset of 32

Assignment statement is
g= h + A[8];
A[12]= h + A[8];

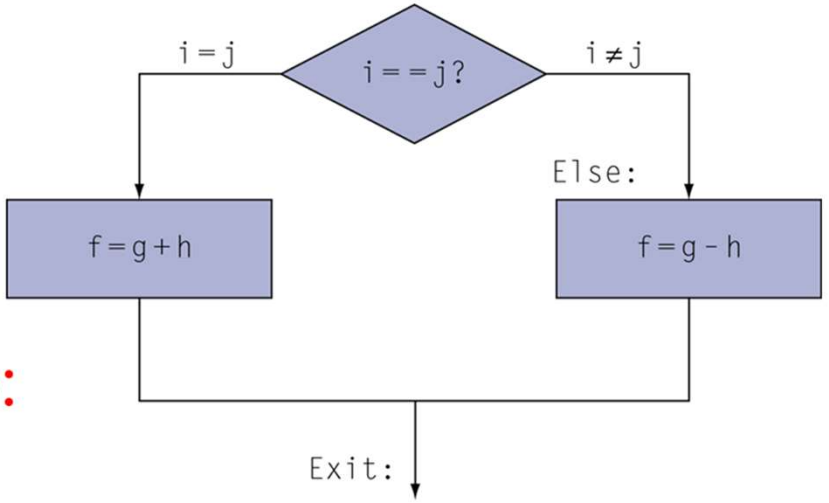
```
lw  $t0, 8($s3)    # load word
add $s1, $s2, $t0
```

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

C code:

```
if (i==j)
      f = g+h;
else f = g-h;
```

– f, g, … in $s0, $s1, …



Compiled MIPS code:

```
 beq $s3, $s4, Else
 add $s0, $s1, $s2
     j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```


Thank You