UNIT – III

COMPUTER FUNDAMENTALS

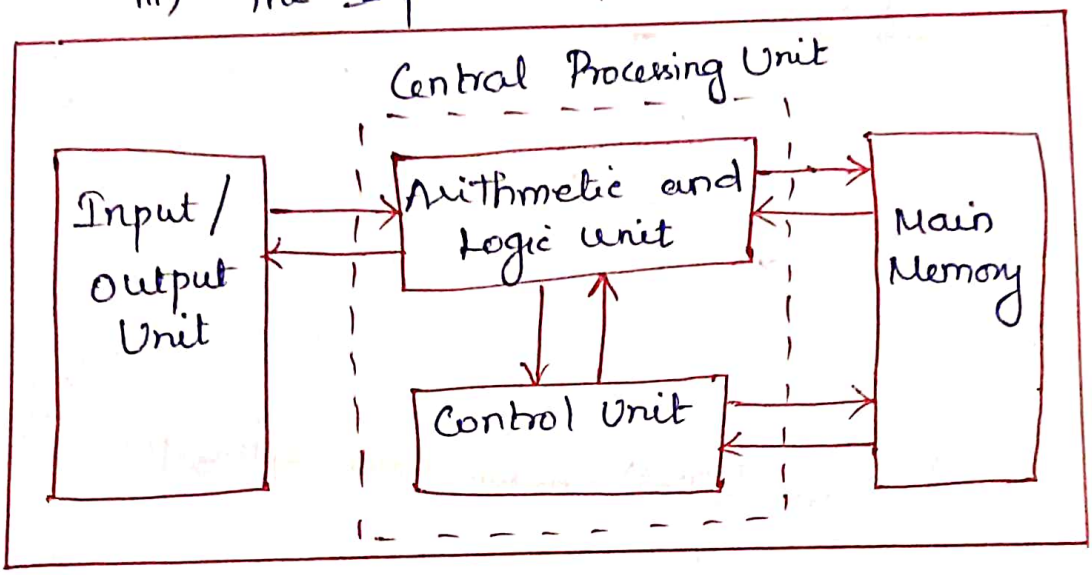
Functional units of a Digital Computer : Von Neumann Architecture – Operation and Operands of Computer Hardware Instruction – Instruction Set Architecture (ISA) : Memory Location, Address and operation – Instruction and Instruction Sequencing – Addressing Modes, Encoding of Machine Instruction – Instruction between Assembly and High level Language.

FUNCTIONAL UNITS OF A DIGITAL COMPUTER

VON NEUMANN ARCHITECTURE

→ Published by John Von Neumann in 1945
→ used in most computers.

* The computer architecture is composed of three main units:
  i) The central Processing Unit (CPU)
  ii) The Memory Unit
  iii) The Input / output Unit (I/O unit)



i) Central Processing Unit:

* CPU is the heart of the computing system.
* CPU takes data and Instructions from the memory unit and makes all sort of calculations based on the instructions given and the type of data provided.

→ 3 Main components:
  a) Control Unit (CU)
  b) Arithmetic and Logic Units (ALU)
  c) Various Registers.

a) Control Unit: - coordinates & controls all the activities.
* Determines the order in which instructions should be executed
- controls the retrival of the proper operands.

* It takes care of step by step processing & all operations inside the computer.

b) Arithmetic Logic Unit:

* ALU performs all arithmetic and logic operations
  - It performs arithmetic functions like addition, subtraction, multiplication, division and logic operations like greater than, less than and equal to etc.

c) Registers:

* Registers are temporary storage locations to quickly store and transfer the data and instructions being used.
* Registers have faster access time than memory.

ii) Memory Unit:

* The memory unit is used for storing data and instructions before and after processing.

2 Types:

a) Primary Memory
b) Secondary Memory.

a) Primary Memory:

classification

i) RAM (Random Access Memory)
ii) ROM (Read only Memory).

i) RAM:

* Read and write Memory.
* Both read & write can be performed.
* volatile memory
  - having limited storage capacity.
  - The contents of RAM are erased once the compulie is switched off.

ii) ROM:

* Read operation alone can be performed.
* Non-volatile Memory.
  ↓
  - Contents are not lost even when the computer is switched off.
* ROM contains manufacturer's instructions.
  - initial program called the Bootstrap Loader whose function is to start the operation of computer system once the power is turned on.

b) Secondary Memory:

* Used to store the content of programs and data permanently.
* Non-volatile, permanent, low cost, cheap memory.

Secondary storage devices – Two types

- Magnetic Devices → Hard Disks
- Optical Devices → CD, DVDs, Pendrive etc.

* Known as back-up memory.
* Computer can run without secondary memory.
  - Slower than primary memory.

iii) Input/output Unit:

* Peripheral devices are used to take inputs and supply outputs.

→ Input devices like Keyboard, mouse etc.
→ output devices are Monitor, Printer.

* Program or data is read into main memory from the input device or secondary storage
* output devices are used to output the information from a computer.

Buses:

* Data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory, by means of Buses.

Types:

a) Data Bus
b) Address Bus
c) Control Bus.

a) Data Bus:

- Carries data among the memory unit, the I/o devices and the processor.

b) Address Bus:

- carries the address of data between memory and processor.

c) Control Bus:

- carries control commands from the CPU in order to control and coordinate all the activities within the computer.

Von Neumann Bottleneck:

* Instructions can only be done one at a time.
- can only be carried out sequentially.

---

i/p unit → provides data to the computer s/m from the outside.
→ Takes data from the i/p devices, converts into machine language and then loads into computer.

o/p unit → Provides the results of computer process to the users.

OPERATIONS AND OPERANDS OF COMPUTER HARDWARE INSTRUCTION

* The ARM assembly language notation

```
ADD a, b, c
```

- instructs a computer to add two variables b and c and their sum in a.

* Sequence of instruction that adds 4 variables.

$a = b + c + d + e$

```
ADD a, b, c    ; Sum of b & c, placed in a
ADD a, a, d    ; Sum of b, c, d, & placed in a
ADD a, a, e    ; Sum of b, c, d & e and placed in a
```

- Three instructions to sum four variables.

; → after the sharp symbols are comments.

I) OPERATIONS OF COMPUTER HARDWARE

ARM Assembly Language Instructions:

classification:

1) Arithmetic Instructions
2) Data Transfer Instructions
3) Logical Instruction
4) Conditional Branch Instruction
5) Unconditional Branch Instruction.

1) Arithmetic Instructions:

i) addi:

```
ADD r1, r2, r3
```

$r_1 = r_2 + r_3$

ii) Subtract:

```
SUB r1, r2, r3
```

$r_1 = r_2 - r_3$

OPERATIONS AND OPERANDS OF COMPUTER HARDWARE INSTRUCTION

* The ARM assembly language notation

```
ADD a, b, c
```

- instructs a computer to add two variables b and c and their sum in a.

* Sequence of instruction that adds 4 variables:

a = b+c+d+e

```
ADD a, b, c    ; Sum of b+c, placed in a
ADD a, a, d    ; Sum of b, c, d, & placed in a
ADD a, a, e    ; Sum of b, c, d & e and placed in a
```

- Three instructions to sum four variables.

; → after the sharp symbols are comments.

II) OPERATIONS OF COMPUTER HARDWARE

ARM Assembly Language Instructions:

classification:

1) Arithmetic Instructions
2) Data Transfer Instructions
3) Logical Instruction.
4) Conditional Branch Instruction.
5) Unconditional Branch Instruction.

1) Arithmetic Instructions:

i) addi:

ADD r1, r2, r3

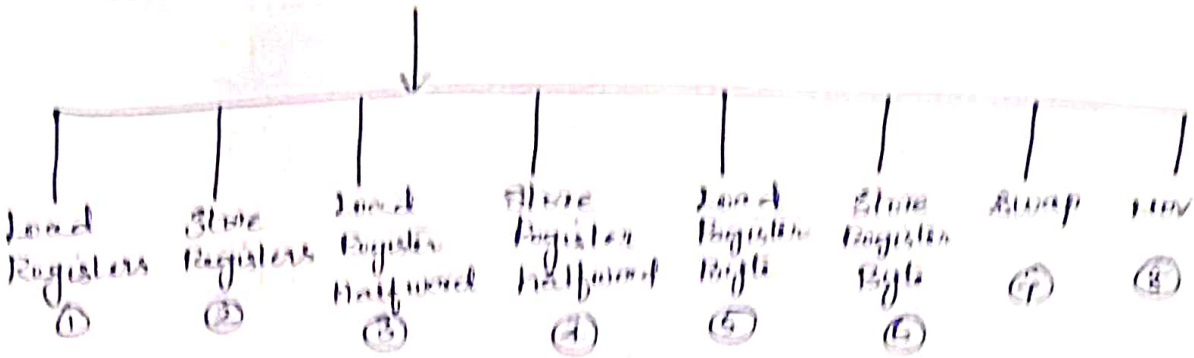
$r_1 = r_2 + r_3$

ii) Subtract:

SUB r1, r2, r3

$r_1 = r_2 - r_3$

2) Data Transfer Instruction



i) Load Registers:

LDR R1, [R2, #20]

R1 = Memory [R2+20]

- Load word from memory to register.

ii) Store register:

STR R1, [R2, #20]

Memory [R2+20] = R1

- Store word from register to memory.

iii) Load Register halfword:

LDRH R1, [R2, #20]

R1 = Memory [R2+20]

- Load halfword from memory to register.

iv) Store register halfword:

STRH R1, [R2, #20]

Memory [R2+20] = R1

- Store halfword from register to memory.

v) Load Register Byte:

LDRB R1, [R2, #20]

R1 = Memory [R2+20]

- Load Byte from memory to register.

vi) Store register Byte:

STRB r1, [r2, #=20]

memory [r2+20] = r1

- Store Byte from register to memory.

vii) Swap:

SWP r1, [r2, #=20]

r1 = memory [r2+20]

Memory [r2+20] = r1

- swap register and memory.

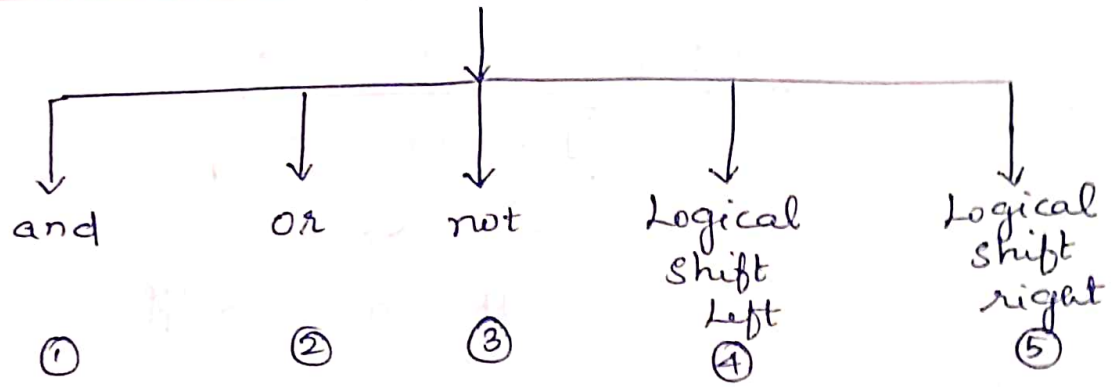
viii) MOV:

MOV r1, r2

r1 = r2

- Copy value of r2 into r1

3) Logical Instruction:



i) and:

AND r1, r2, r3

r1 = r2 & r3

- Three register operands.
- Perform bit-by-bit AND of r2 and r3 and save in r1

ii) OR:

```
ORR  R1, R2, R3
```

R1 = R2 | R3

- 3 register operands
- Perform bit-by-bit OR of R2 and R3 and save in R1

iii) not:

```
MVN  R1, R2
```

$r_1 = \sim r_2$

- 2 register operands
- Perform bit-by-bit NOT operation.

iv) Logical Shift Left:

```
LSL  R1, R2, #10
```

R1 = R2 << 10

- Shift left the content of R2 by 10.

v) Logical Shift Right:

```
LSR  r1, r2, #10
```

$r_1 = r2 >> 10$

- Shift right r2 by 10.

4) Conditional Branch Instruction:

i) compare:

```
CMP  r1, r2
```

- conditional flag = R1 - R2
- compare for conditional branch.

ii) Branch on EQ, NE, LT, LE, GT, GE

BEQ 25

- Branch if equal.

5) Unconditional Branch Instruction:

i) Branch (always)

B 2500

Branch

- goto pc + 8 + 10000

ii) Branch and Link

BL 2500

2500 × 4 = 10000

- for procedure call
- goto pc+8 + 10000

II) OPERANDS OF COMPUTER HARDWARE:

Operand:

- Data on which the operation is performed.

Three operands:

i) Register operands
ii) Memory operands
iii) Constant or Immediate operands

i) Register operands:

* The operands of arithmetic instructions are restricted
- stored in a limited number of special locations called register.
* The size of a register in ARM architecture is 32 bits
  * Word is a group of 32 bits.

Word = 32 bits
(or)
4 bytes
(or)
2 halfword

* The number of registers in ARM architecture is 16 to 32 registers.

→ The reason for limited number of registers:

i) smaller number of registers is faster.

ii) Large number of registers may increase clock cycle time.

- use r0, r1, ..., r15 to refer to registers 0, 1, 2, ..., 15

Ex:

C assignment statement

$$f = (g+h) - (i+j)$$

ARM instruction:

```
ADD   r5, r0, r1      register r5 contains g+h
ADD   r6, r2, r3      register r6 contains i+j
SUB   r4, r5, r6      r4 gets r5-r6
```

variables f, g, h, i, j are assigned the registers r4, r0, r1, r2, r3

ii) Memory operand

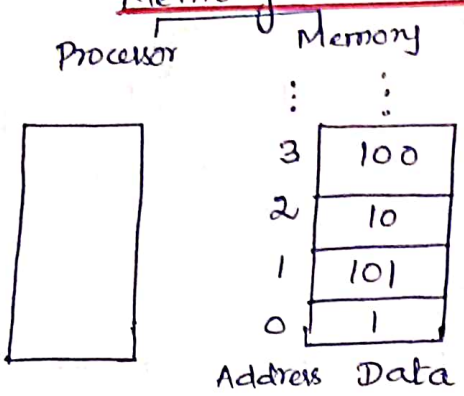
* Data elements that belong to complex data structures are kept in memory
  - They can't be placed in registers.
* The CPU can keep only a small amount of data in registers
  - but computer memory contains billions of data elements.

Memory:

* Memory is a large, single dimensional array, with the address acting as index to the array, starting at 0.

* To access a word (32 bit) in memory, the instruction must supply memory address

Memory addresses and contents of Memory.



- The address of the 3rd element is 2
- The value of Memory[2] is 10

* To access the data in memory, use data tranfer instructions:
  - Instructions that transfer data between memory and registers.

→ The data transfer Load instruction is used to copy data from memory to a register.

Format of Load Instruction

Name of operand	Register to be Loaded	Constant and register used to access memory

Ex: C assignment statement — Memory operand

$$g = h + A[8]$$

ARM instruction

```
LDR  r5, [r3, #8]
ADD  r1, r2, r5
```

A → an Array

Variables g, h → registers r1, r2

Temporary variable r5 is used

- Base Address of array is in r3.
- register r5 get A[8]

* The address of the array element is the sum of the base address of array [r3] and the number to select element 8. i.e) [r3+8]

$$r_5 \leftarrow r_3 + 8$$

iii) Constant or Immediate operands:

* Use a constant in an operation. Constant as an operand
* If one operand of arithmetic instruction is a constant, it is called immediate constant.

Ex:

To add 4 to register 3

i)

```
ADD r3, r3, #4
```

r3 = r3+4

* Hash symbol (#) means the following number is a constant

2)

```
LDR r5, [r1, #Addr constant 4]
ADD r3, r3, r5
```

r1 + addr constant 4 → memory address of the constant

r5 = constant 4

r3 = r3 + r5

r3 = r3 + 4

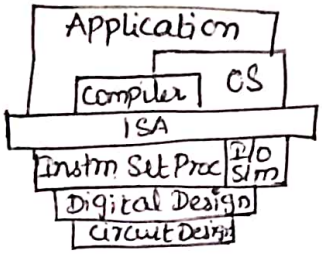
Index Register:

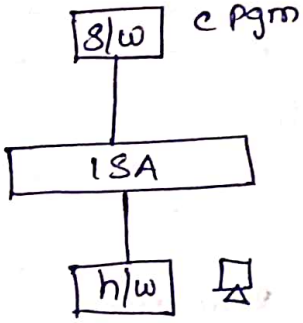
* The register in the data transfer instruction holds an index of an array, with the offset used for the starting address of an array.

x ——— x

INSTRUCTION SET ARCHITECTURE (ISA)

* ISA :
  - An interface between the hardware and software.
  * instruct the hardware about what to be done through the software instructions





ISA → instructions are written in machine readable form on ROM that processor execute

* ISA helps in defining the operations, modes of operation supported, addressing, storage related inputs, how to use, how to access the operands through instructions etc.
* A processor is detailed through the ISA.
* ISA is a collection of instructions and formats supported for the processor.

→ An instruction can be a word in processor's language

Format of Instruction

Opcode	Addr of op1	Addr of OP2	Dest Addr	Addr of next instn

* **Instruction Set:**
  - collection of instruction that execute by processor

classification of ISAs

- Determined by means used for storing data in CPU

i) Stack Architecture

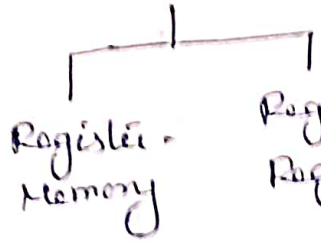
- Operands are implicitly on top of the stack

ii) Accumulator Architecture

- One operand is in the accumulator (reg).

iii) General-Purpose Registers Architecture

- Operands are in registers or specific memory locations



Ex: $C = A + B$

Comparison of Architectures

Stack	Accumulator	Register-Memory	Register-Register
PUSH A PUSH B Add POP C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B ADD R3, R1, R2 Store C, R3

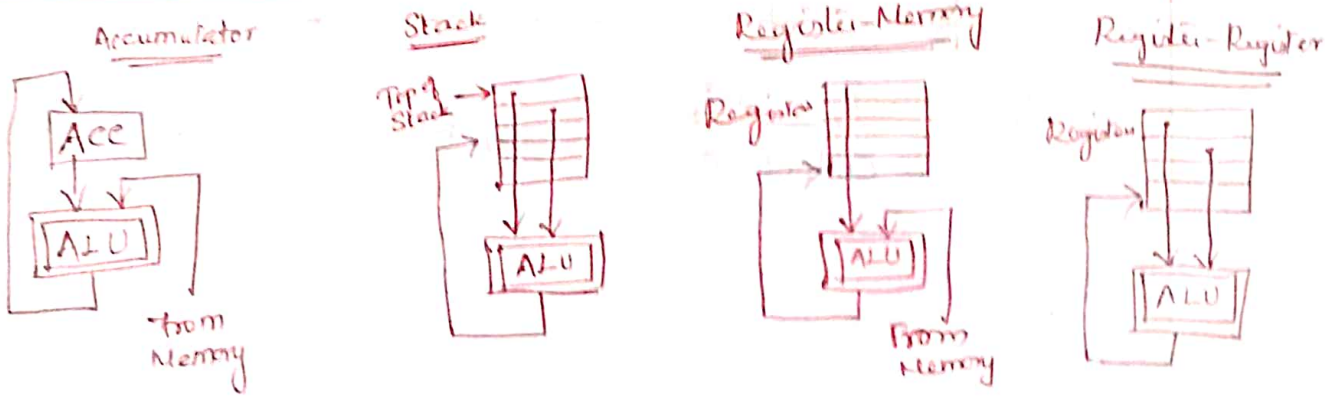
* Table
* Fig

classification of operations

i) Data Transfer - load, store, mov

ii) Data Manipulation - Arithmetic, Logical

iii) Control Transfer - conditional & unconditional Branch

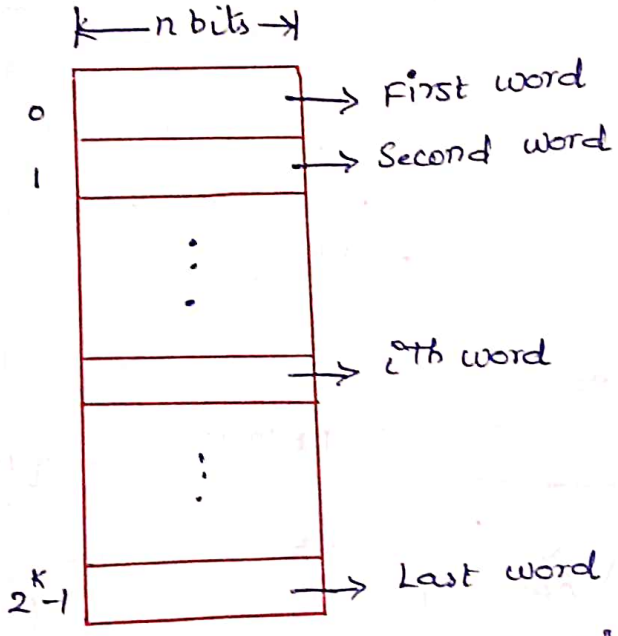


MEMORY LOCATION, ADDRESS AND OPERATION

Memory location:

Memory: The memory consists of many millions of storage cells
- each of which can store a bit of information having the value 0 or 1
- Group of fixed size

* Each group of n bits is referred to as a word of information.
  - n is called the word length.
* The memory of a computer can be represented as a collection of words.
* Modern computers have word lengths range from 16 to 64 bits.

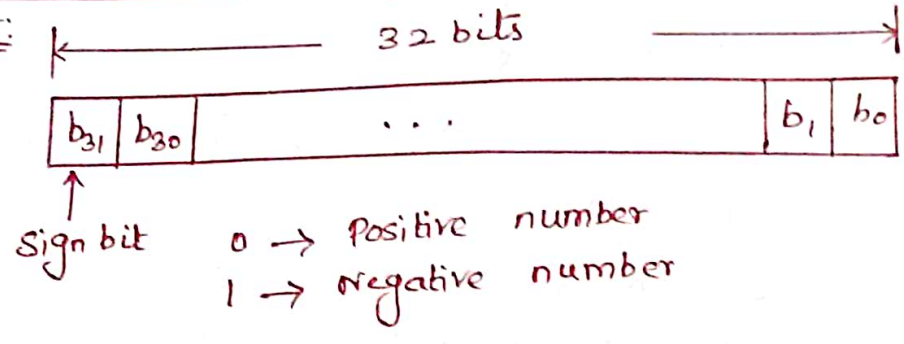


* A unit of 8 bits is called a "byte"
* Accessing the memory to store or retrieve, requires addresses for each item locations

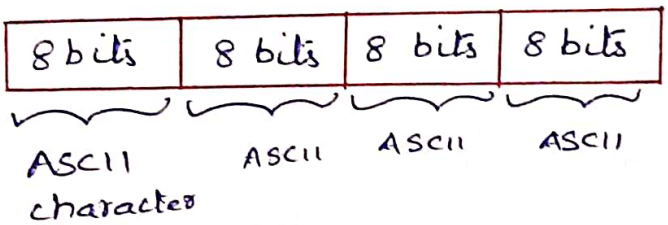
→ The memory locations are addressed from 0 to 2^K-1

Representation of integer and characters in Memory:

Integer:



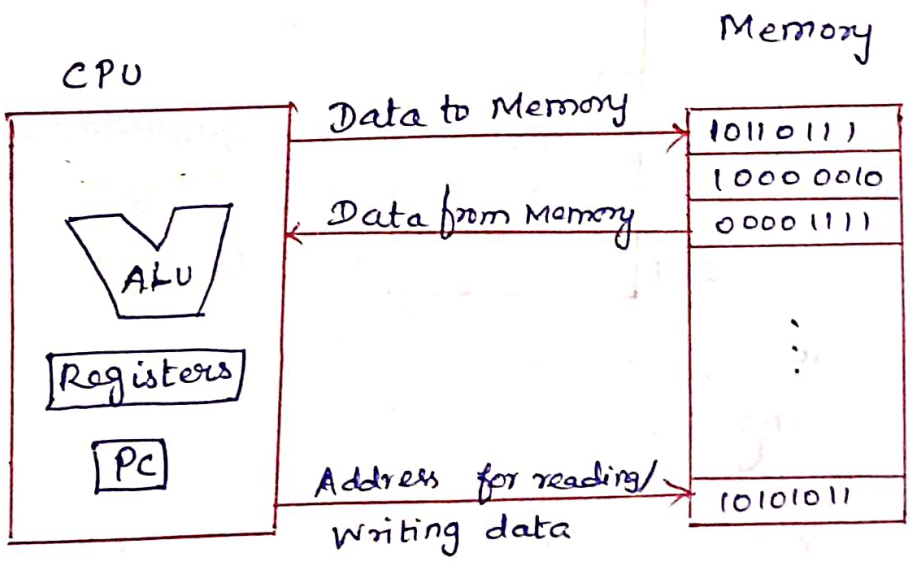
characters:



Ne → 1 word
cha → 4 parts
A → 65
B → 66

Memory Location:

Address	Value
0x00	01001010
0x01	10111010
0x02	10101011
⋮	⋮
0xFF	10111011



MEMORY ADDRESS

* A 'k' bit address generates an address space of 2^k locations (0 to $2^k - 1$)

k	Number of locations
10	$2^{10} = 1024 = 1K$
16	$2^{16} = 65,536 = 64K$
20	$2^{20} = 1,048,576 = 1M$
24	$2^{24} = 16,777,216 = 16M$

* To retrieve information from memory, either for one word or one byte (8-bit), addresses for each location are needed.

k-bit
address
2^k
$2^2 = 4$
0 0 — 0
0 1 — 1
1 0 — 2
1 1 — 3

24-bit memory:

$2^{24} = 16\,M \quad (1M = 2^{20})$

32-bit memory:

$2^{32} = 4G \quad (1G = 2^{30})$

$1K$ (Kilo) $= 2^{10}$

$1T$ (tera) $= 2^{40}$

	1 Byte	= 8 bits
	1 KB	$= 2^{10}$ Bytes
(Mega)	1 MB	$= 2^{20}$ Bytes
(Giga)	1 GB	$= 2^{30}$ Bytes
	1 TB	$= 2^{40}$ Bytes
(Peta)	1 PB	$= 2^{50}$ Bytes

$= 1024 \times 1024$ Bytes

Byte Addressability:

* A byte is always 8 bits
* Word length ranges from 16 to 64 bits
* Assign distinct addresses to individual bit locations in the memory.
  - Successive addresses refer to successive byte locations
* The assignment is called byte-addressable memory.

Ex:

i) A Byte Addressable Memory with 24-Bit Addresses. Find out the number of memory locations present.?

$$K = 24$$

$$2^K = 2^{24} = 16,777,216 \text{ locations.}$$

$$= 2^{20} \times 2^4$$

$$= 1MB \times 16$$

$$= \boxed{16\,MB} \text{ memory locations}$$

ii) A Byte Addressable Memory with 32-bit Addresses. Find out the number of memory locations present.

$$2^K = 2^{32}$$

$$= 2^{30} \times 2^2$$

$$= 1GB \times 4$$

$$= \boxed{4\,GB}$$

* Byte locations have addresses 0, 1, 2 ...
  - If word length is 32 bits, the successive words are located at addresses 0, 4, 8, ...

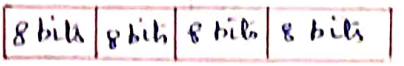
32 bits = 4 bytes.

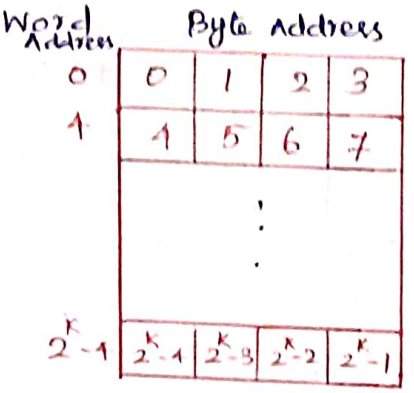
Big-Endian and Little-Endian Assignments.

* 2 ways of assigning Byte addresses

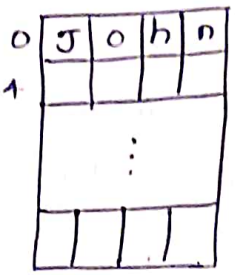
i) Big-Endian:

* Lower byte addresses are used for the most significant bytes of the word.

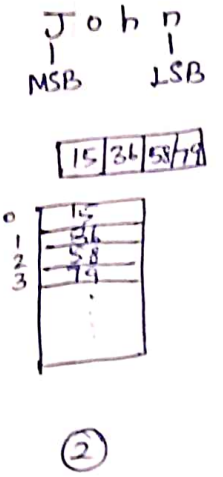




Ex: John



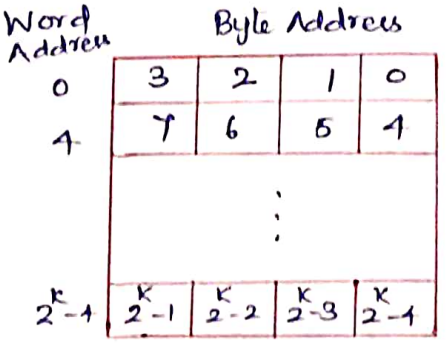
①



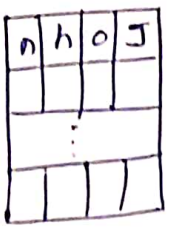
②

ii) Little Endian:

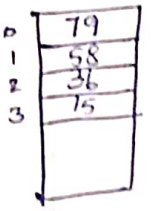
* opposite ordering
* Lower byte addresses are used for the least significant bytes of the word.



Ex: John



①



②

* used in commercial machines.

Word Alignment:

* Words are said to be aligned in memory if they begin at a byte address.
  - a multiple of the number of bytes in a word.

16-bit word:
Word addresses: 0, 2, 4, ...

32-bit word:
Word addresses: 0, 4, 8, ...

64-bit word:
Word addresses: 0, 8, 16, ...

* Access numbers using word address.
* individual characters, are accessed using byte address
  - character strings of variable length by indicating 'end of string'
    '\0' → null character.

MEMORY OPERATIONS

Two Basic operations:

i) Read (Load / Fetch)

ii) Write (Store)

Read operation: (Load / Fetch) – Memory to Register

* Transfers a copy of the contents of a specific memory location to the processor.
  - Memory contents remain unchanged.
* To start a Read operation,
  - the processor sends the address of the desired location to the memory and requests that its contents be read.
* The memory reads the data stored at that address and sends them to the processor.
  → Registers can be used.

→ Load instructions.

Write operation: (Store) – Register to Memory.

* Transfers an item of information from the processor to a specific memory location.
  - overwriting the former contents of that location.
* To initiate a write operation,
  - the processor sends the address of the desired location to the memory, together with the data to be written into that location.
* The memory then uses the address and data to perform the write.
  → Registers can be used.
* After completion of processing, the results should be stored in memory.

→ Store instructions.

INSTRUCTION AND INSTRUCTION SEQUENCING

- → Register Transfer Notation (RTN)
- → Assembly Language Notation (ANN)
- → Basic Instruction Type
- → Instruction Execution & Straight-line sequencing
- → Branching
- → Condition codes
- → Generating Memory Address

Instruction:

* Computer Program consists of sequence of small steps
  - Adding two numbers
  - Testing for a particular condition
  - reading a character from the keyboard
* To perform any tasks in the computer, instructions are needed. They are stored in memory

Four Types of operations:

1. Data transfers between the memory and the Processor registers.
2. Arithmetic and logical operations
3. Program sequencing and control
4. IIo Transfers.

Register Transfer Notation

* Transfer of information from one location in the computer to another.

Possible locations:

- memory locations (LOC, PLACE, A)
- Processor registers (R0, R1, R2, R3, R4, R5)
- registers in the I/O subsystems (DATAIN, OUTSTATUS)

contents → [ ]

* Identify a location by a symbolic name standing for its hardware binary address (LOC, R0, DATAIN, ...)

→ Contents of a location are denoted by placing square brackets around the name of the location

EX:

1) $R_1 \leftarrow [LOC]$
2) $R_3 \leftarrow [R1] + [R2]$

LOC → memory location in bytes
- content in mem. loc to reg. R1

* Add the contents of the registers R1 & R2 and then places their sum into Register R3

- Right side → value
- Left side → name of the location where the value is to be placed.

Assembly Language Notation:

"Not Portable"

* Assembly language (symbolic machine code) takes complete control over the system and its resources

→ Represent machine instructions and programs

EX:

1) Move LOC, R1 = R1 ← [LOC]
   (Source, Dest)

   Load R2, LOC (des, source)

   * Contents of LOC are moved to R1
     - unchanged in the LOC.
     - old contents of register R1 are overwritten

2) Add R1, R2, R3 = R3 ← [R1] + [R2]
   (Source, Dest)

   * Add the contents in R1 & R2 and place the sum to R3.

Basic Instruction Type

* Based on number of operands that are specified in the instruction

i) 3- Address Instructions:

Format:

Operation Source1, Source2, Destination

Ex: | ADD R2, R3, R1 |

R1 ← [R2] + [R3]

* To large to fit in one word for a reasonable word length

ii) 2- Address Instructions:

Format:

Operation Source, Destination

Ex:

1) | ADD R2, R1 |

R1 ← [R1] + [R2]

* Add the contents of R1 & R2 and place in R1. Contents in R1 will be overwritten.

2) | Move B, C |

C ← [B]

iii) 1- Address Instructions:

* only one operand

Ex: | ADD M |

AC ← [AC] + [M]

* use Accumulator Register (Processor Register)

Load A
Store A

→ A to Acc

Format:

Operation Destination

Ex: C = A+B

Load A
Add B
store C

iv) Zero - address Instruction:

* Source & destination are specified implicitly
* Contents are stored in push Down stack.

1) ADD
2) MUL

Ex: C = A+B

```
PUSH A
PUSH B
ADD
POP  C
```

TOS ← (A+B)

C ← TOS

TOS = Top of Stack

Evaluate C = A+B using processor registers.

```
Move  A, Ri
Move  B, Rj
Add   Ri, Rj
Move  Rj, C
```

Instruction Execution and Straight-Line Sequencing:

Instruction Execution:

- Executing an instruction is a two-phase Procedure

1) Instruction Fetch:

* Instruction is fetched from the main memory location whose address in the PC.
* This instruction is placed in the Instruction Register in the processor.

PC → contains the address of next instruction to be executed.

→ The processor contains a register called Program Counter (PC) which holds the address of the next instruction to be executed.

a) Instruction Execution

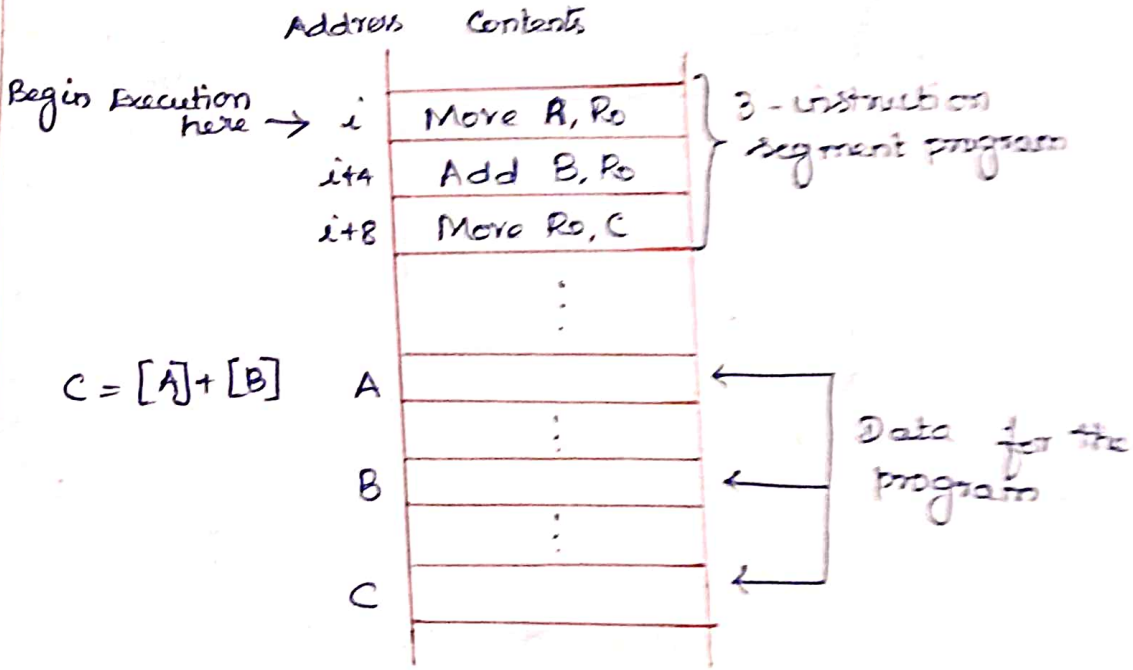
IR - instruction Register

* To begin executing a program, the address of the first instruction must be placed into the PC.
* The instruction in IR is examined to determine which operation is to be performed
  - The specified operation is then performed by the processor.

Straight-line Sequencing:

* Processor control circuits use PC to fetch and execute instructions, one at a time, in the order of increasing addresses called straight-line sequencing.

Program segment - Memory of a computer



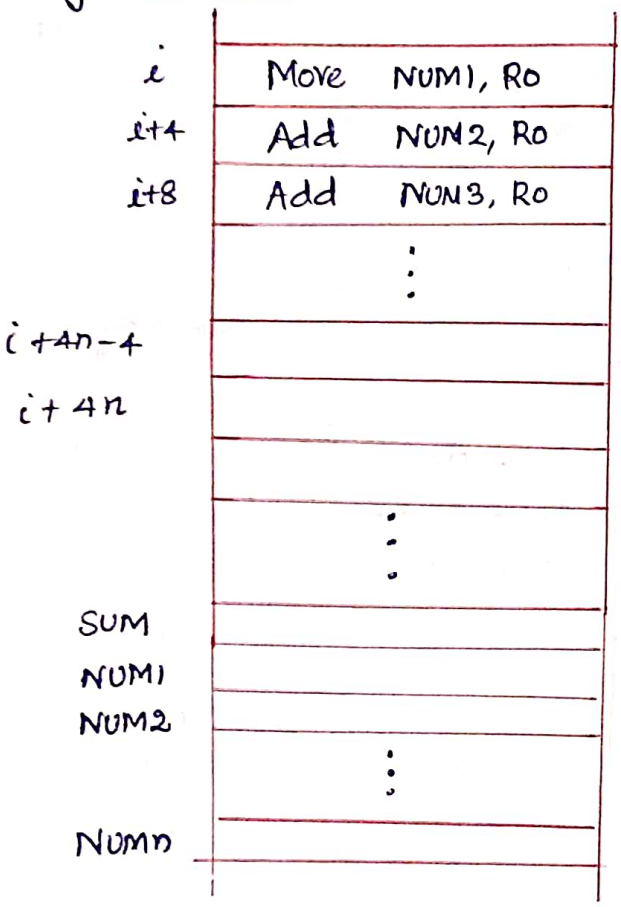
Assumptions:

* one memory operand per instruction
* 32-bit word length
* Memory is byte addressable
  - Full memory address can be directly specified in a single-word instruction.

* During each instruction execution, the PC is incremented by 4 to point to the next instruction.
  ie) i, i+4, i+8, ...
  - The three instructions of the program are in successive word locations.
* Starting at location i.
  - Since each instruction is 4 bytes long, the 2nd, 3rd instructions are at the addresses i+4, i+8.

Branching:

A straight-line program for adding n numbers



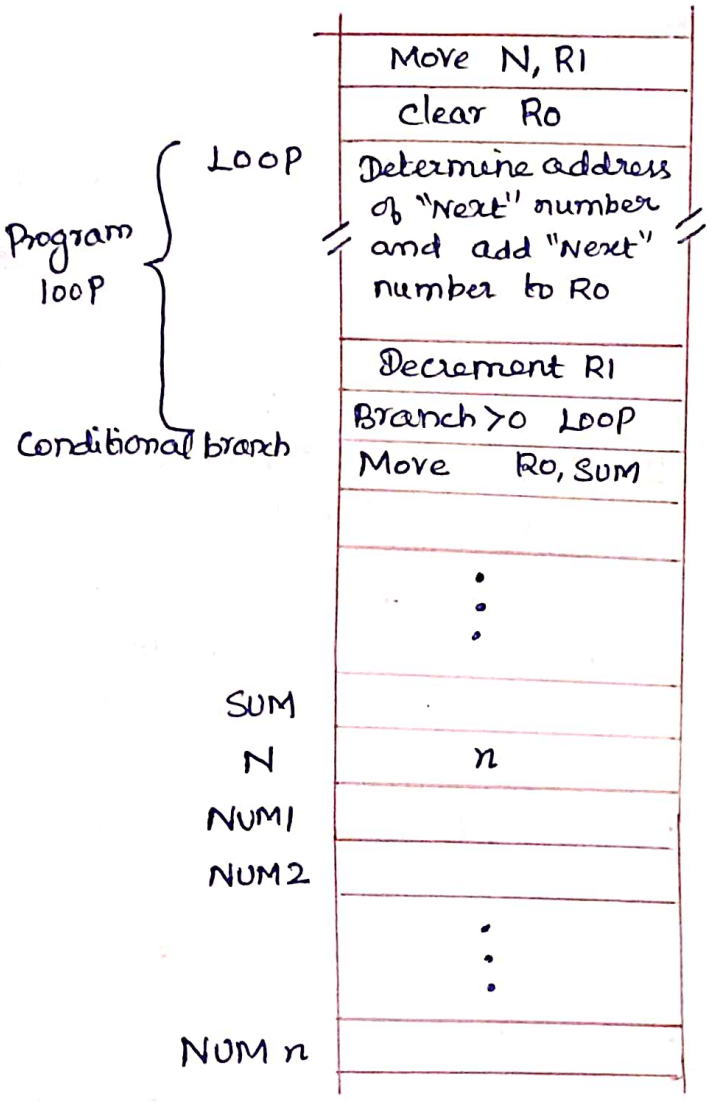
* Addresses of memory locations containing n numbers are NUM1, NUM2, ... NUM n
* Separate Add & Load Instructions are used to add each number to the contents of register.
* After all numbers have been added, the result is placed in memory location SUM.
* Implement a program loop in which the instructions read the next number & add it to the current sum
→ To add all numbers, the loop has to be executed as many times as there are nos

* Loads a new value into the PC
* Processor fetches and executes the instruction at the new address called branch target
  - follows the branch instruction in sequential address order

* A conditional branch instruction causes a branch if a specified condition is satisfied.
- If the condition is not satisfied, the PC is incremented and the next instruction in sequential order is fetched and executed.

Ex:

Branch > 0 LOOP



Using a loop to add n numbers

* A conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater than 0.

* The loop is repeated as long as there are entries in the list that are yet to be added to R0.
- At the end of the n^{th} pass through the loop, the Decrement instruction produces a value of zero and hence, branching does not occur.
- Instead, the Move instruction is fetched and executed.
- It moves the final result from R0 into memory location SUM.

Condition Codes:

* The Processor keeps track of information about the results of various operations

Condition code flags:

- Records the required information in individual bits

Condition code Register or Status register:

- Flags are grouped together in a special processor register.
* Flags are set to 1 or cleared to 0.

Four Flags:

N (negative)	Set to 1 if the result is negative, otherwise, cleared to 0.
Z (zero)	Set to 1 if the result is 0, otherwise, cleared to 0
V (overflow)	Set to 1 if arithmetic overflow occurs, otherwise, cleared to 0.
C (carry)	Set to 1 if a carry-out results from the operation; otherwise cleared to 0.

Ex: Compute A − B

A : 11110000

B : 00010100

Soln:

B's 2's complement

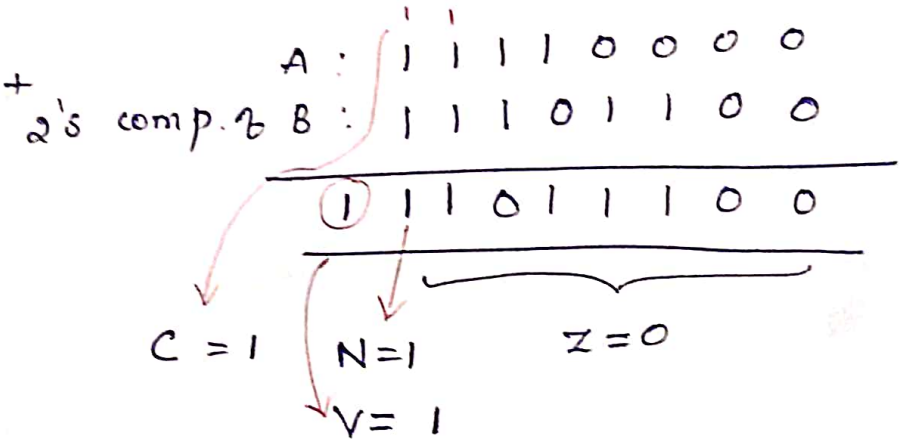
00010100

⇓

1's complement 11101011

1

2's complement 11101100



C - carry
N - Negative
Z - zero
V - overflow

Generating Memory Addresses:

* To specify the address of an operand.
* The instruction set of a computer provides a number of addressing modes.

Ex:

Using a loop to add n numbers.

→ The Add instruction in a block refers to a different address during each pass

R_i → used to hold the memory address of an operand.

* If it is initially loaded with the address NUM1 before the loop is entered and is then incremented by 4 on each pass through the loop.

ADDRESSING MODES

* Multiple forms of addressing are called addressing modes.
  - The way in which an operand is specified in the instruction

Addressing Modes:

1. Immediate addressing
2. Register addressing
3. Scaled Register addressing
4. PC-relative addressing
5. Immediate offset
6. Register offset
7. Scaled Register offset
8. Immediate offset Pre-indexed
9. Immediate offset Post-indexed
10. Register offset Pre-indexed
11. Scaled Register offset Pre-indexed
12. Register offset Post-indexed.

1. Immediate Addressing:

* The operand is a constant within the instruction itself.

Example:

```
ADD r2, r0, #5
```

Format:

4 bits	2bits	6 bits	4bits	4bits	12 bits
Cond	f	opcode	Rn	Rd	Immediate

* 32-bit Addresses - ARM instructions

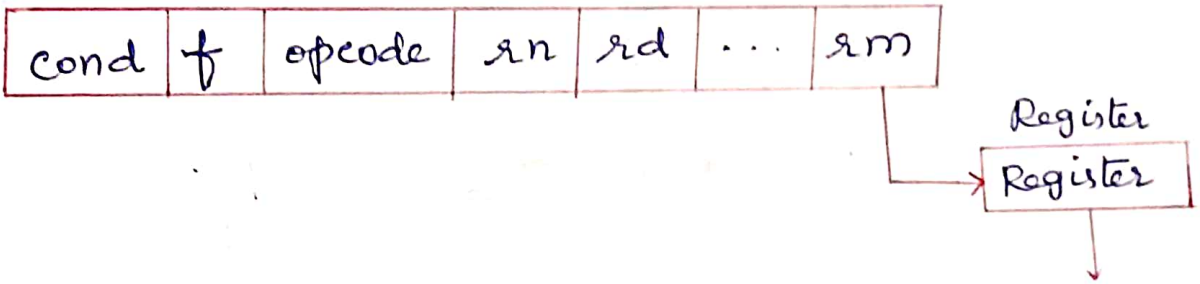
2. Register Addressing:

* operand is a register

EX:

```
ADD  R2, R0, R1
```

Format:



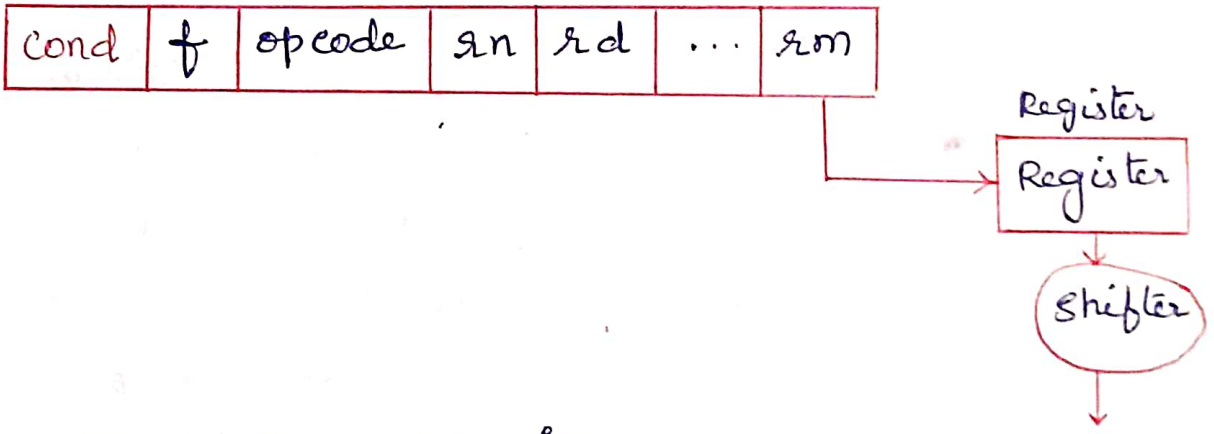
3. Scaled Register Addressing:

* The register operand is shifted first

EX:

```
ADD  R2, R0, R1, LSL #2
```

Format:



4. PC-relative addressing:

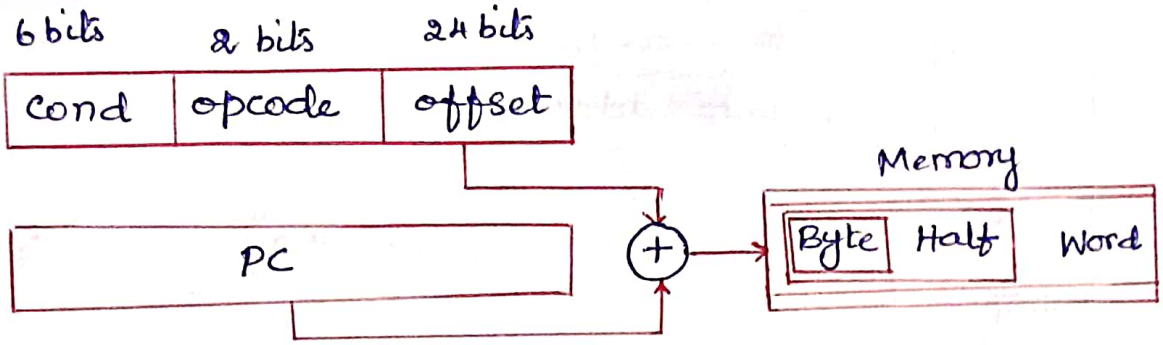
* One of the 16 ARM registers is the Program Counter
* PC relative addressing with loads – to fetch 32-bit constants that could be placed in memory along with the program.

→ Branch address is the sum of the PC and a constant in the instruction

EX:

```
BEQ 1000
```

Format:



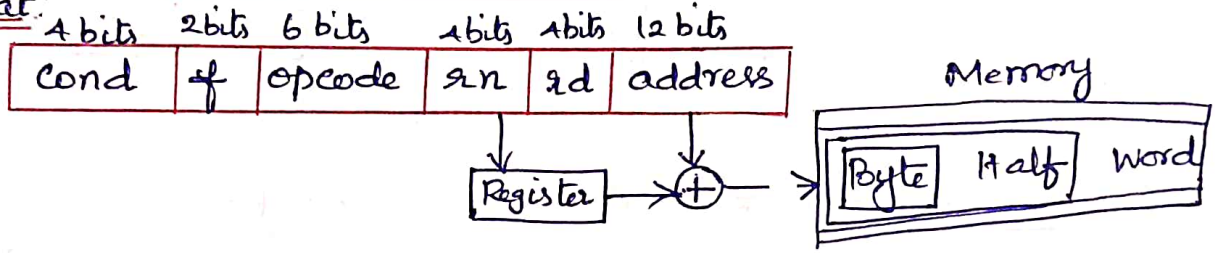
5. Immediate offset:

* Constant address is added to the base register.

Ex:

```
LDR r2, [r0,#8]
```

Format:



6. Register offset:

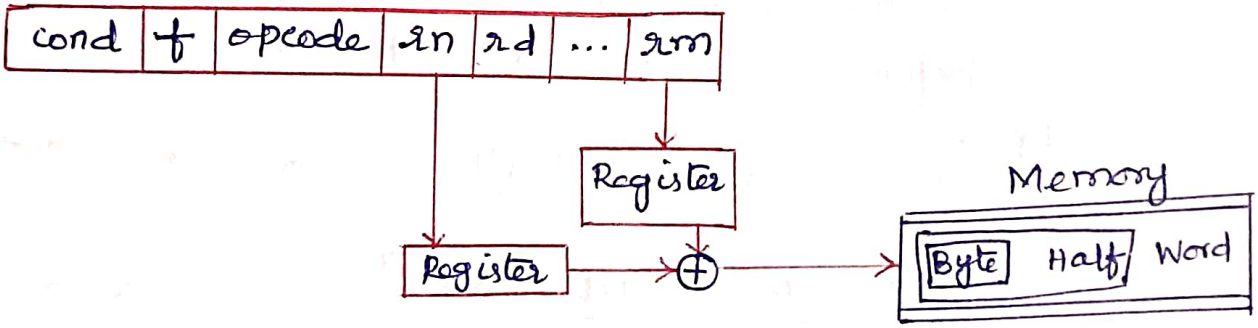
* Instead of adding a constant to the base register, another register is added to the base register.

→ This mode can help with an index into an array,
- array index is in one register
- the base of the array is in another

Ex:

```
LDR r2, [r0,r1]
```

Format:



7. Scaled Register offset:

* The second operand can be optionally shifted left or right in data processing instructions.

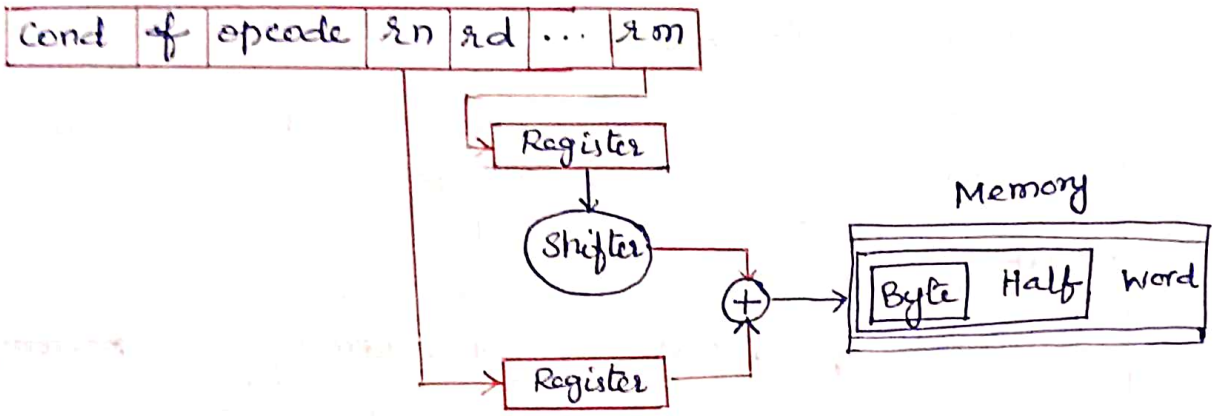
- This addressing mode allows the register to be

shifted before it is added to the base register
- This mode can be useful to turn an array index into a byte address by shifting it left by 2 bits.

Ex:

```
LDR r2, [r0, r1, LSL #2]
```

Format:



8. Immediate offset Pre-indexed:

* This mode updates the base register with the new address as part of the addressing mode.
- On load instruction,
  - the content of the destination register changes based on the value fetched from memory and the base register changes to the address that was used to access memory
* This mode can be useful when going sequentially through an array.
- In this mode, the addition or subtraction occurs before the address is sent to memory.

Ex:

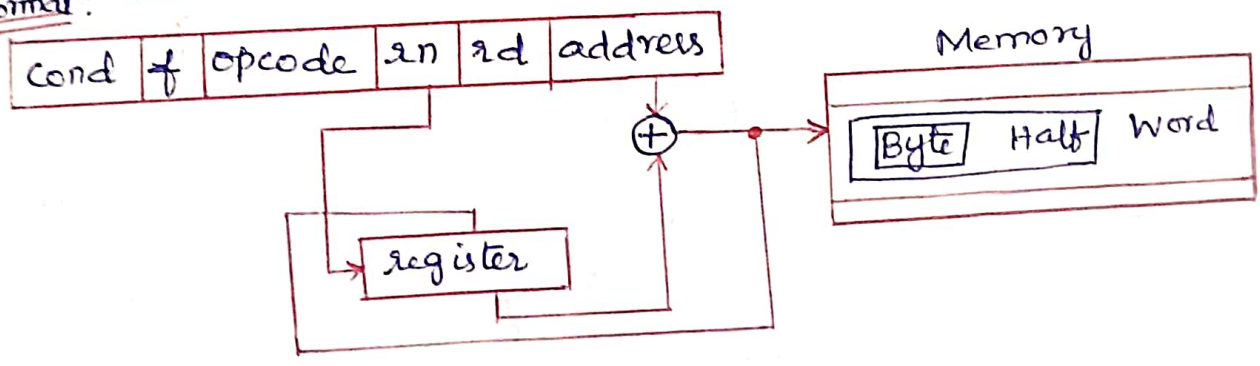
```
LDR r2, [r0, #4]!
```

with update

$r_2 \leftarrow$ mem. word [r0+4]

$r_0 \leftarrow r_0 + 4$

Format:



9. Immediate offset Post-indexed:

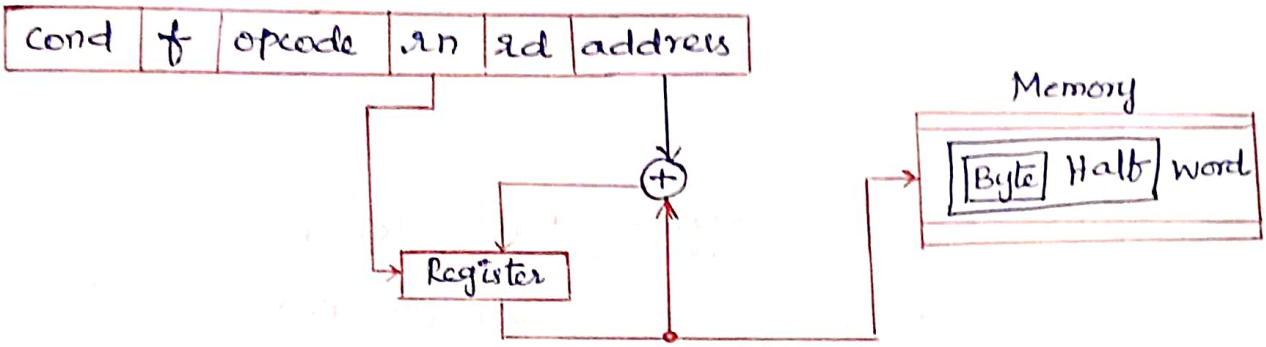
* Like immediate pre-indexed except that the address in the base register is used to access memory first and then the constant is added or subtracted later.

r2 ← mem.word[r0]
r0 ← r0+4

Ex:

Format:

```
LDR r2, [r0], #4
```



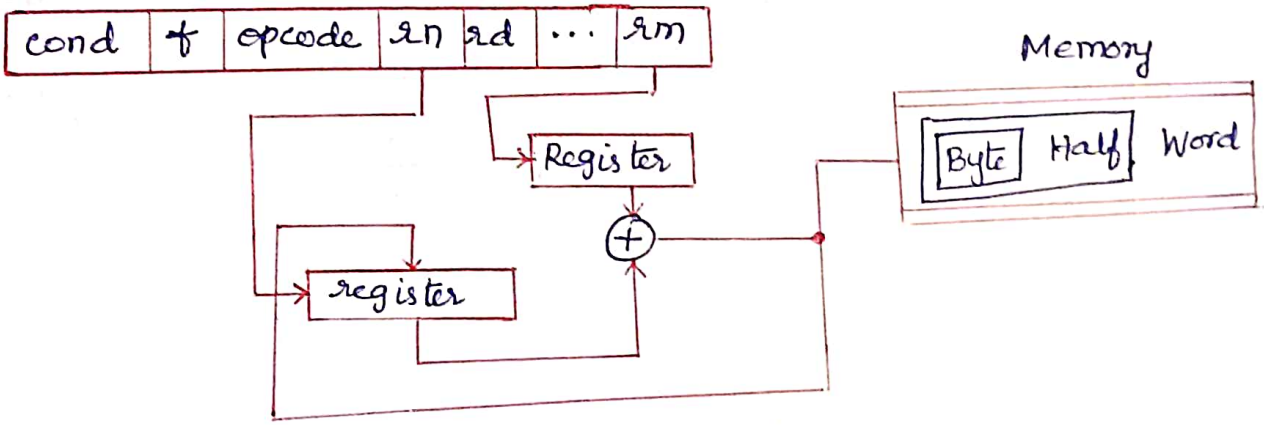
10. Register offset Pre-indexed:

* Same as immediate offset Pre-indexed, except add or subtract a register instead of a constant.

Ex:

```
LDR r2, [r0, r1]!
```

Format:



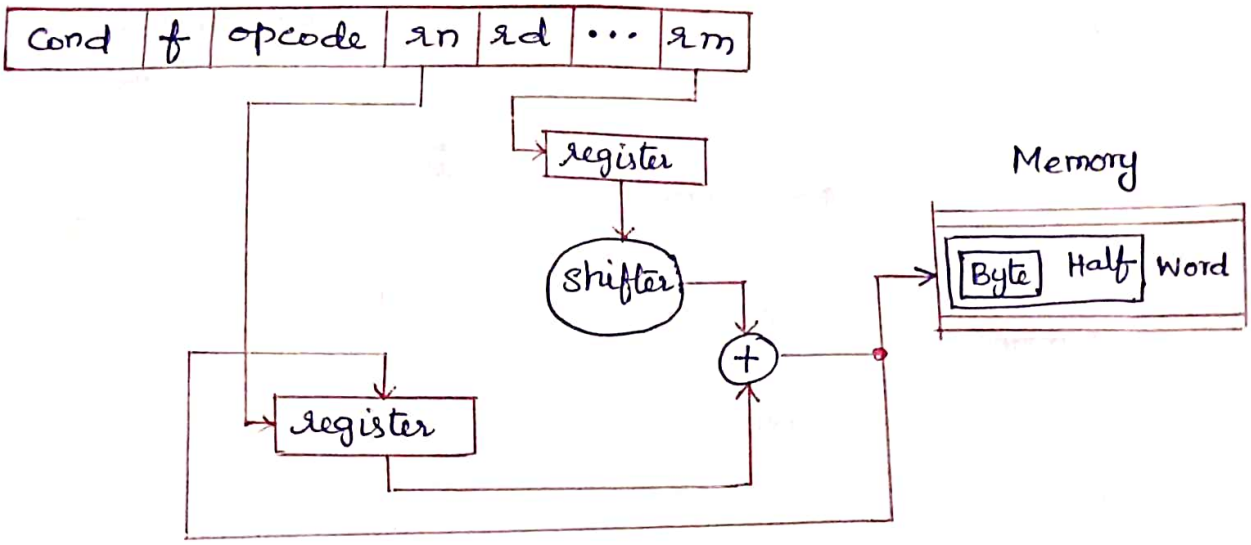
11. Scaled register offset pre-indexed:

* Same as Register Pre-indexed, except Shift the register before adding or subtracting it.

Ex:

```
LDR r2, [r0, r1, LSL #2]!
```

Format :



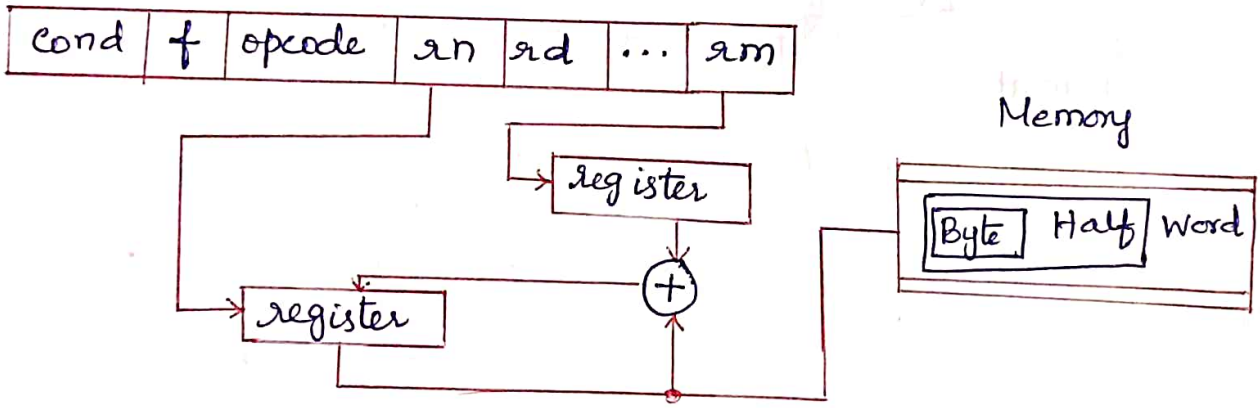
12. Register offset Post-indexed :

* Same as Immediate Post-indexed, except add or subtract a register instead of a constant.

Ex:

```
LDR r2, [r0], r1
```

Format:



ENCODING OF MACHINE INSTRUCTION

* The instructions specify the actions that must be performed by the processor circuitry to carry out the desired tasks.

Machine instructions:

→ To be executed in a processor, an instruction must be encoded in a compact binary Pattern

Assembly Language instructions:

→ The instructions that use symbolic names and acronyms
- which are converted into the machine instructions using the assembler program.

Assumption:

- all instructions are one word in length

→ 32-bit words

operations:

- add, subtract, move, shift, rotate, branch.

operands of different sizes

* 32-bit and 8-bit numbers or 8-bit ASCII-encoded character

OP code:

* The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the OP code for the given instruction.
  - 8 bits
  - 256 possibilities for specifying different instructions

* Unique number given to every operation
  ADD – 07H, MOV – 15H etc

Examples:

*) Add R1, R2

2) Move 24(R0), R5

3) LShiftR #2, R0

4) Move #$3A, R1

5) Branch>0 LOOP

1) Add R1, R2:

* If the processor has 16 registers.
→ Registers ⇒ 4 bits, So R1, R2 → 4+4 bits
→ Additional bits are needed to indicate that the Register Addressing mode for each operand.

2) Move 24(R0), R5

* opcode & 2 reg } → 16 bits
* Some bits to express the source operand uses the Index addressing mode and that the index value is 24

3) LShiftR #2, R0 & 4) Move #$3A, R1

* Immediate values → 2, $3A ⇒ 14 bits
18 bits → to specify the opcode, addressing modes and the register.

5) Branch>0 LOOP

8 bits → opcode
24 bits → branch offset
Absolute or Register Indirect → addressing mode.
* Jump Instructions.

* The instruction must include operation, addressing information and operands.
– Size of the instruction depends on how much information must be encoded. (Type of operation, Number of operands, Size of operands)

Possible Formats:

Encoding instructions into 32-bit words

i) One-word Instruction

8	7	7	10
Opcode	Source	Dest	Other info

* 8 bits are used for the op code
* two 7-bit fields for specifying the source and destination operands
  - identifies the addressing mode and the register involved
- "Other info" field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

ii) Two-word Instruction — CISC

Opcode	Source	Dest	other info
Memory address/Immediate operand			

Ex: The instruction a) Move R2, LOC

- Absolute Addressing mode

18 bits → opcode, the addressing modes and the register

14 bits → LOC

- include a second word as part of the instruction

b) Add #$FF000000, R2

- immediate operand

iii) Three - operand Instruction

RISC

OPcode	Ri	Rj	Rk	Other info

Ex:

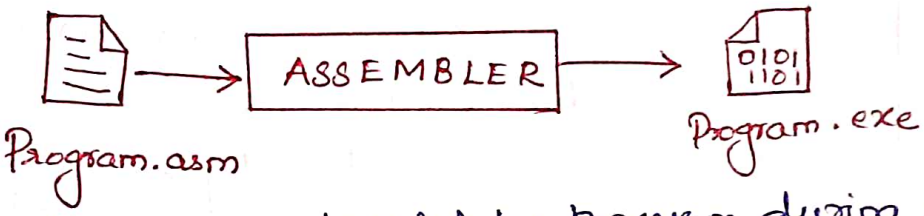
Add R1, R2, R3

- The instruction that uses three operands.

$R3 \leftarrow [R1] + [R2]$

Assembler:

* Instructions are encoded by assembler



* Instructions are decoded by processor during execution.

Two Types of Computers:

i) CISC
ii) RISC

i) CISC - Complex instruction set computer.

* The complex instructions are implemented using multiple words, closely resembling operations in high-level programming languages.

- Define a highly functional instruction set, which makes extensive use of the processor register.

Add LOC, R2 $\Rightarrow$ Add (R3), R2

→ Load the address LOC into R3 before execution. R3 is used as a pointer to the desired memory location.

How to load a 32-bit address into a register that serves as a pointer to memory locations?

Solution:

1. To direct the assembler to place the desired address in a word location in a data area close to the program.
   - Then Relative addressing mode is used to load the address.
2. To use logical and shift instructions to construct the desired 32-bit address. - use immediate addressing mode.

ii) RISC - Reduced Instruction Set computers

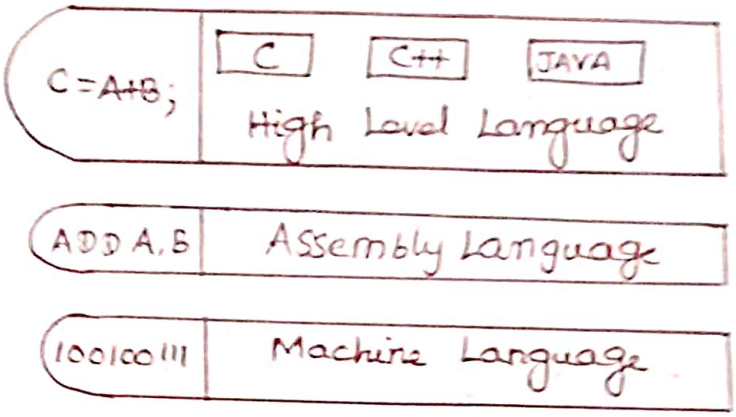
* An instruction must copy only one word has led to a style of computers.
* All manipulation of data must be done on operands that are already in processor registers.

EX: Add R1, R2, R3 $R3 \leftarrow [R1] + [R2]$

- In an instruction set where all arithmetic & logical operations use only register operands, the only memory references are made to load/store the operands into / from the processor registers.

* few instructions
* small instructions
* simple addressing modes.
* All instruction have same size
* Simple fixed encoding format.

INTERACTION BETWEEN ASSEMBLY AND HIGH-LEVEL LANGUAGE



Assembly Level Language :

* Allows users to write a program using alphanumeric mnemonic codes

High Level Language :

* Enables a user to write a program in a language that resembles English words and familiar mathematical symbols
  - COBOL → 1st
  - Ex: Python

	Assembly Level Language	High Level Language
1.	It needs an assembler for conversion	It needs a compiler/Interpreter for conversion
2.	Assembly level language to machine level language	High level language to Assembly Level language to machine level language
3.	Machine dependent	Machine Independent
4.	Mnemonic, codes are used	English statement is used
5.	Easy to access hardware component	Difficult to access hardware component

6.	Very difficult to debug and understand the code of an assembly language	Very easy to debug and understand
7.	More efficient	Less efficient
8.	Execution of code takes less time	More time for execution
9.	Better accuracy	Less accuracy.
10.	Provides support for low-level operations	Does not provide any support for low level languages
11.	More compact code	No compactness
	EX: add a,b,c add a,a,d	a = b + c + d;