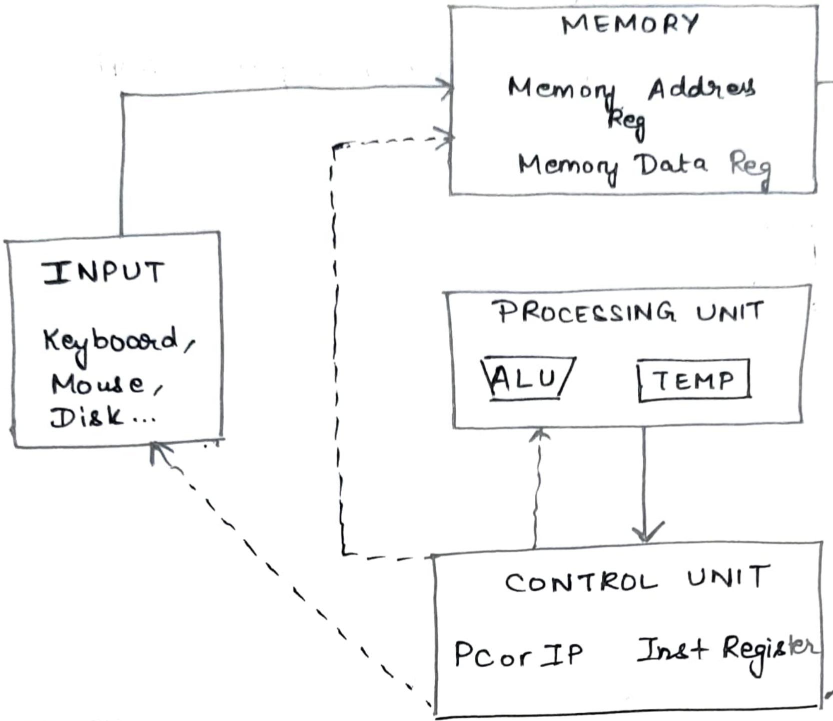
The Von Neumann Model :



MEMORY

→ In the memory, Data (Data Memory) } is stored .
Program (Code Memory) }

→ Contains bits grouped into → bytes (8 bits = 1 byte)
+ words (1 byte, 2 byte, 3 byte so on)
(defined by ISA).

→ Addressability is determined by how the bits are accessed in the memory

Types: 1) Word addressable : each word (32 bit) has an address
2) Byte addressable : (8 bits) each byte has address.

→ The total number of addresses = Address space

- MIPS — 2^{32} (32 bit address) = 4 GB memory
- X86-64 — 2^{48} (48 bit address) = =

Word Addressable Memory:

- Every data word → Unique address.
- For each 32 bit word → A unique address (as per ISA).

Word Address	Data					Memory
0000 0003	D 1	6 1	7 A	1 C	→	W1
0000 0002	1 3	C 8	1 7	5 5	→	W2
0000 0001	F 2	F 1	F 0	F 7	→	W3
~~0000 0008~~ 0000 0000	8 9	A B	C D	E F	→	W4

(Byte address)

= 0000 000C

0000 0008

0000 0004

0000 0000

Big Endian : In this order scheme, high order byte is stored on the starting address A and low order byte is stored on the next address (A +1)

Little Endian : Here the low order byte is stored on the starting address A and high order byte is stored on the next address (A +1).

E.g

Big Endian				Little Endian			
C	D	E	F	F	E	D	C
8	9	A	B	B	A	9	8
4	5	6	7	7	6	5	4
0	1	2	3	3	2	1	0

MSB → LSB
Ascending

MSB ← LSB
Descending

Ways of storing memory:

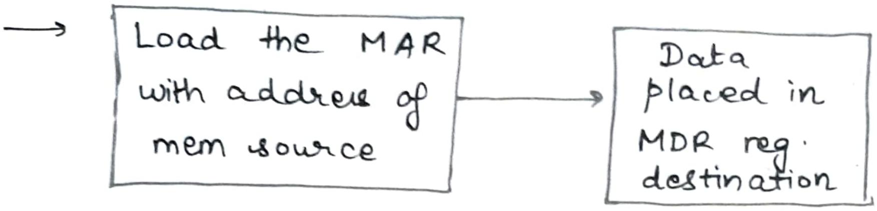
There are 2 ways of accessing memory:

1) MAR – Memory Address Register (Holds the address of the current instruction to be fetched from mem)
2) MDR – Memory Data Register. (Holds the contents found at address held in MAR)

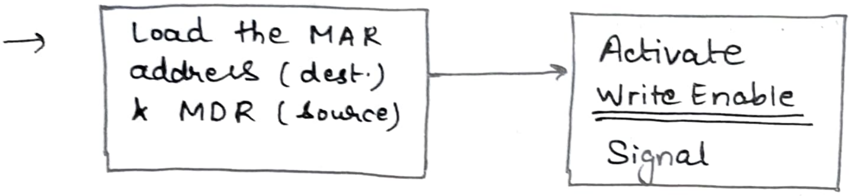
Two Operations:

1) Reading data (From memory)
2) Writing data. (to memory)

To read data:


Load the MAR with address of mem source
Data placed in MDR reg. destination

To write data:


Load the MAR address (dest.) & MDR (source)
Activate Write Enable Signal

PROCESSING UNIT

- Consisting of many functional units.

Basic start = ALU (Arithmetic & Logic Unit) deals with the computations.

E.g. add, sub, mul, div, or, and, nor etc.

→ Takes up a data as a word (32 bit)

Temp storage = Registers (Much faster than memory)

For. e.g if 3 + (2 * 5) - 3 is to be calculated, first 2 + 5 is calculated ~~in~~ and stored in a temp register.

Size is one word.

CONTROL UNIT:

→ Does step by step execution of instructions in a program.

→ Tracking is done with the help of an instruction register (IR)

→ The PC (Program counter) / IP (Instruction pointer) stores the address of the next instruction to be fetched.

Properties:

→ Also called stored program computer (i.e all instructions in memory)

↳ Stored program : • Every instruction stored in a linear memory array.
- Unification of memory (Instructions & data)
- A stored value is interpreted depends on the control signals.

↳ Sequential instruction processing:
- 1 instruction processed at a time.
- PC identifies the current inst.
- Advances sequentially except for control transfer.

E.g. lw $t2 , 32($0)   or   0x8CA00020

add ($s0) , ($s1), ($s2))   0x02328020.

Working: First instruction $\xrightarrow[PC + 4]{Execute}$ Next instruction ⟶ Goes on . . .

add: 0000 0000   0000 0004

For (word addressable memory) → PC + 1.

TYPES OF INSTRUCTIONS:

3 types →
- Operate (Execute inst. in ALU)
- Data Movement (Read or write)
- Control Flow (Change sequence of execution)

Eg.
Addition

c = a + b   or   add a, b, c. ⟶ [a ← b+c]

Dest. operand.

Mnemonic for type of operation

Source operands

In MIPS

a = $s1

b = $s2

c = $s0

Conversion of Assembly Code to Machine Code

Instruction in MIPS

add $s0, $s1, $s2;   (32 bytes)

Field Values

0	17	18	16	0	32
op	rs	rt	rd	shamt	funct

↓

Machine Code

op	rs	rt	rd	shamt	funct
00000	10001	10010	10000	00000	100000

Whole to hexadecimal = 0x02328020.

MIPS Instruction Set:

All instructions are 32 bits with 3 formats:

3 types → R type

op	rd	rs	rt	shamt	funct
6 bit	5	5	5	5	6 bit

31 26 21 16 10 6 0

I - type

op	rs	rt	immediate
6 bit	5 bit	5 bit	16 bits

31 25 21 16 0

J - type

op	target address
6 bit	32 bits

31 26 0

Different field values.

op : Type of operation

rd : Destination register

rs, rt : Source register

shamt : Shift amount

funct : tells what to do (select variant of op field only in R type)

immediate : ~~immediate~~ Data or address

target address : Address to be jumped (Jumping destination) r type → add , sub, or , nor, xor, slt etc.

Load / store → lw , sw etc.

Immediate → addi , subi etc.

Branch → beq , brne etc.

Jump → jmp

- - - - - - - - - - - - -

R-Type :

3 register operands.

Fields — op - 6 bits

rd - 5 bits

rs, rt - 5 bit

shamt = 5 bit

funct = 6 bits

Load Word in MIPS :

E.g word = A[2] // High level language

lw $s3 , 2($s0) // MIPS assembly

$$\$s3 \leftarrow \text{Memory}(\underbrace{\$s0}_{\text{base address}} + \underbrace{2}_{\text{offset.}})$$

Immediate instruction:

I-type: lw $\overset{rt}{\$s3}$, $\overset{imm}{8}(\overset{rs}{\$s0})$ // LOAD == Mem → Reg.

op	rs	rt	imm
35	16	19	8

sw $s3 , 8($s0) // STORE == Reg → Memory

op	rs	rt	imm
35	16	19	8

Shift in MIPS :

- Allows splitting / combining bytes in MIPS (32 bit)
- Do left SHIFT or right SHIFT.

Ex : sll $s0 , $s1 , 8;

⇒ $s0 ← $s1 << 8. (Left shift)

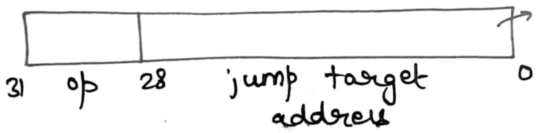
op	rs	rt	rd	shamt	funct
0	0	9	8	4	0

sll $8 , $9 , 4

- Largest shift value = $2^5 - 1 = 31$.
- If different bit operands are given, convert them into 32 bits. by appending 0's in MSB. (zero extension)

Jump in MIPS : (J type)

- Unconditional branch or jump.
- MIPS



target address is in the instruction.

Pc gets updated to.

$PC \leftarrow PC + [31:28] \mid \text{sign-extend}(\text{target}) * 4$

Here 4 MSB bits of incremented PC concatenates with current target (26 b) then left shift 2.

- Variants → jal → Jump & link.
  jr → Jump Register

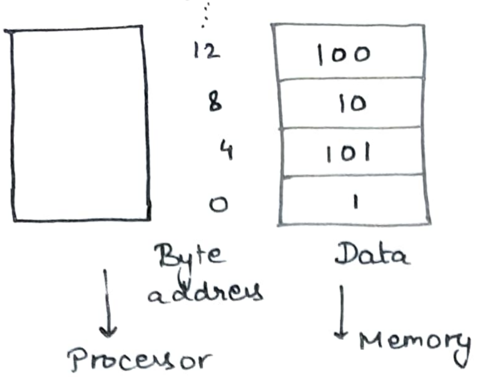
- General jumps can be anywhere in memory.
- Best case is to specify a 32-bit address space but it is not possible. (bcz target address can be of maximum 26 bits)

Optimization:

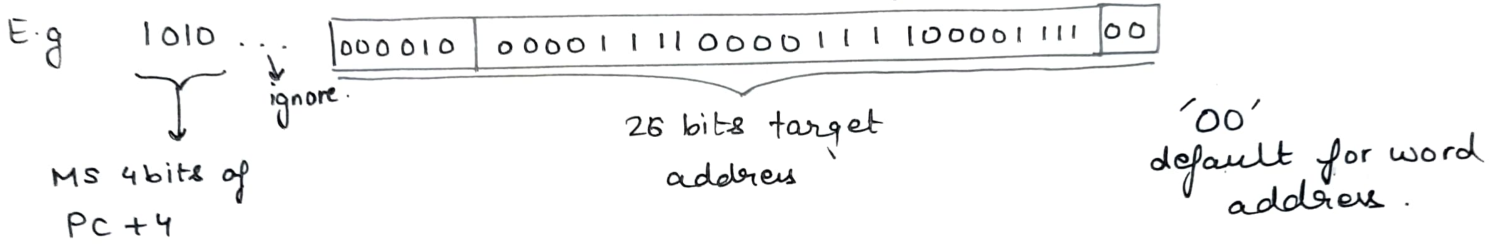
- Jump will jump to word aligned addresses so last 2 bits = 00 So assume the address end with 00 and leave them out.

Now instead of 26, 28 bits of target address.

- In many architectures, word must start at address that are multiples of 4. (MIPS) therefore faster data transfer.



- MIPS gets the 4 other bits by taking the 4 most significant bits from PC + 4 (the next instruction ← jump instruction), due to which we cannot jump to anywhere in memory, but sufficient.



Instruction Execution:

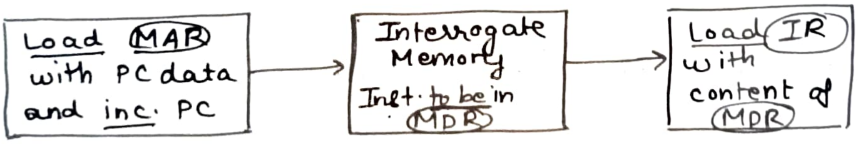
→ Instruction cycle: Process of executing one instruction.

A sequence of steps / phases go through.

- FETCH :

Memory ⟶ Instruction $\xrightarrow{\text{loads}}$ IR. Common to every instruction type.

Process :


Load MAR with PC data and inc. PC
Interrogate Memory Inst. to be in MDR
Load IR with content of MDR

DECODE: Identifies the instruction (by 4 → 16 decoder by identifying the one of 16 opcodes going to be processed)

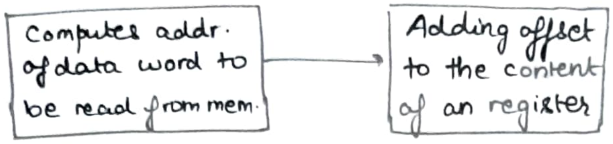
Input = IR [15 : 12]

Remaining 12 bits for what else to be processed in the specific instruction.

EVALUATE ADDRESS: Computes address of the memory location where inst. need to be processed.

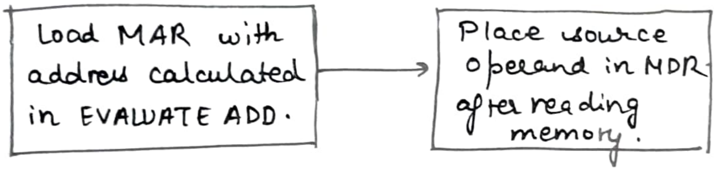
→ Neccessary in LDR. but not ADD

Process:

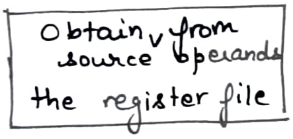

Computes addr. of data word to be read from mem. → Adding offset to the content of an register

FETCH OPERANDS: Obtains the source of the operands needed to process the instruction.

Case 1: LDR


Load MAR with address calculated in EVALUATE ADD. → Place source operand in MDR after reading memory.

Case 2: ADD


Obtain source operands from the register file

EXECUTE: Executes the instruction.

STORE RESULT: Writes the output to the designated solution.

CHANGING THE SEQUENCE EXECUTION:

- A normal program runs in the sequential order (usually top to bottom). But there is a possible chance for changing the sequence execution.

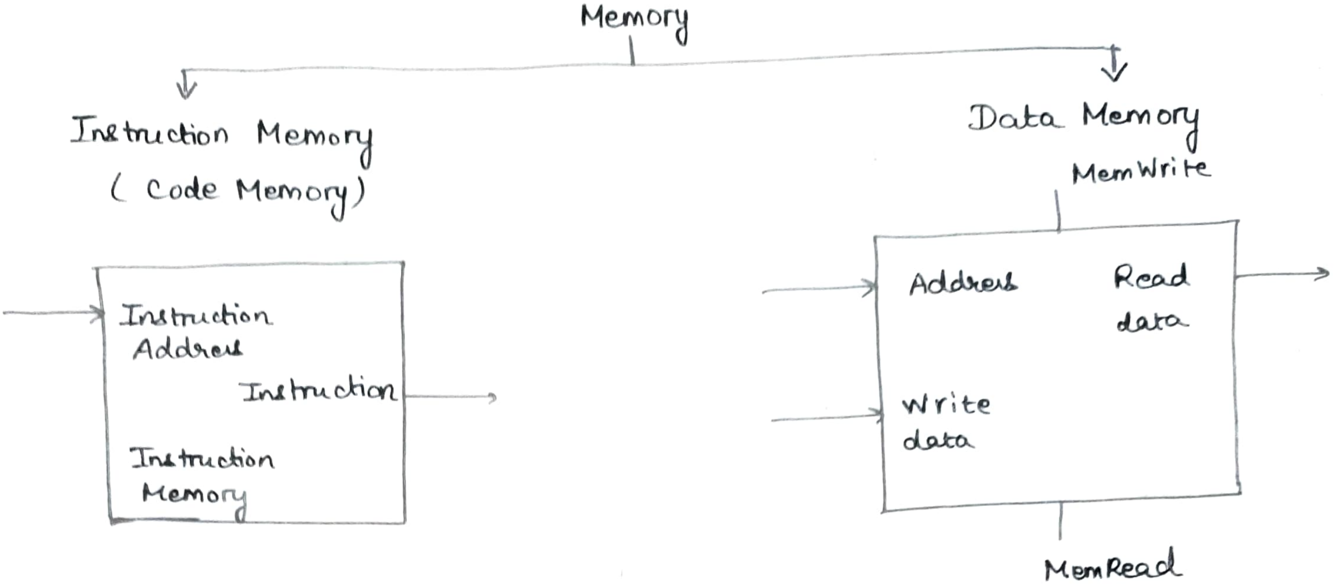
This can be done with the help of control statements.

Two steps:

Step 1: PC change $\xrightarrow{\text{loading}}$ done in ~~FETCH~~. EXECUTE phase

Erasing inc. PC $\xrightarrow{\text{loading}}$ done in FETCH phase.

So basically after each instruction, the program counter increments(i.e new inst).



Basic terminologies:

- Program counter - 32-bit register containing the address of the current fetched instruction.
- Code Memory: Input : 32 bit address
  Read : 32 bit data.
  Output: Destination register.
- Register file: 32 - element, 32 bit
  ~~Two~~ 3 components : 2 read and 1 write port
- Data Memory: 1 read and 1 write port.

```
if WE == 1 {
    write data WD in address A }
else if WE == 0
    reads data from address A onto RD
```

DATAPATH (INSTRUCTION PROCESSING)

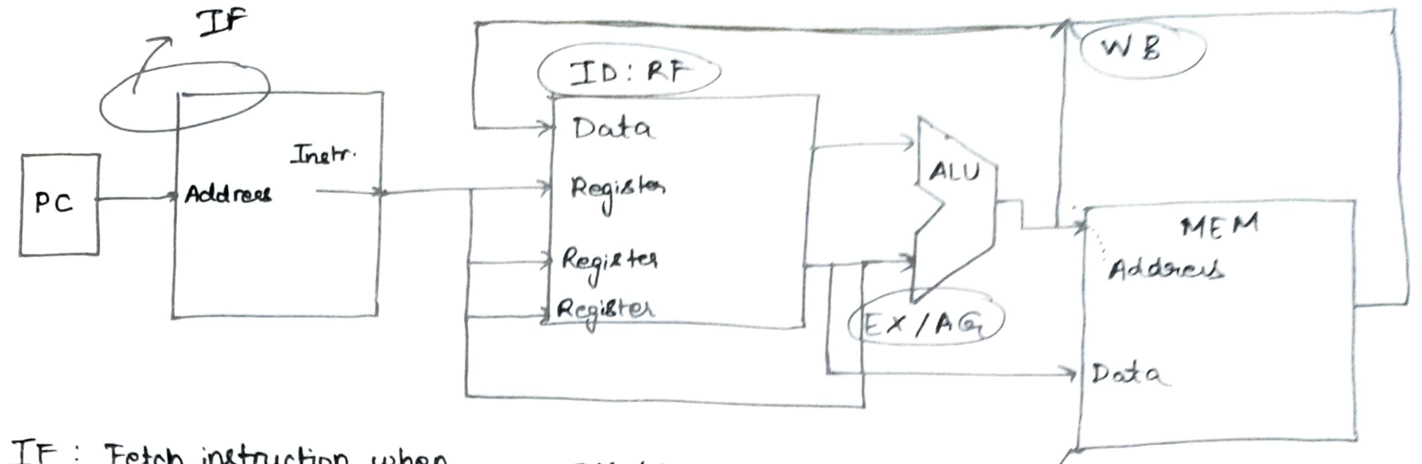
5 steps: → Instruction fetch (IF)

→ Instruction decode and register operand fetch (ID / RF)

→ Execute memory address (EX)

→ Memory operand fetch (MEM)

→ Store / Write result (WB)



IF : Fetch instruction when PC is incremented.

ID / RF : Decode and get operands from reg. file

EX / AG : Execute at ALU / get address from it

WB : Store here / Write to reg lw

MEM : Use the address to get data.

~~SINGLE/CYCLE~~

SINGLE CYCLE

1. Every instruction takes a single clock cycle.

2. State updates are made at the end of the instruction's exe.

3. Disadvantage:
The slowest instruction determines the cycle time.

MULTI-CYCLES

1. Every instruction is broken into multiple clock cycles.

2. State updates can be made during instruction's execution
Architectural state updates made at end of instruction's execution.

3. Advantages
The slowest stage of instruction determines the cycle time.