COMPUTER ARCHITECTURE

✱ PERFORMANCE METRICS

* Performance $\propto \frac{1}{\text{Time}}$

$$\text{Time} = \underbrace{\overset{(IC)}{\#\text{instructions}}}_{\text{compiler tech.}} \times \underbrace{\overset{(CPI)}{\text{cycles per instruction}}}_{DLP} \times \underbrace{\text{time per cycle}}_{\text{clock speed}}$$

* ISA (Instruction set architecture) is the interface between software and hardware. It provides a behavioural abstraction to the software and a set of specifications to the hardware.
Eg:- ARM ISA can run on Apple M1 or Snapdragon or ARM native hardware
x86 ISA can run on Intel or AMD hardware.

* The hardware i.e. computer organisations is chosen based on its frequent job. i.e. floating point ops, integer ops, data i/o.
We generally rate machines based on:-
- No. of ~~cases~~ cores (For multithreaded programs)
- GPU (For vectorized operations)
- clock speed
- RAM

* SPEC (System Performance Evaluation Consortium): Created a set of programs representative of all programs. These are the SPEC CPU Benchmarks (2017).
Companies generally release the ratio of time taken by their machine to the time taken by a standard machine.

* Another metric used to evaluate is MIPS (Million instructions per second). $= \frac{\#\text{Instr.}}{\text{Time} \times 10^6}$

$$= \frac{IC}{IC \times CPI \times \frac{1}{f} \times 10^6} = \frac{f}{CPI \times 10^6}$$

Generally used to report the upper bound of a machine's performance.

$2^3 \times 2^{10} \times 2^{10} \times 2^{10} = 2^{33}$

Page No.:
Date:

Similarly, we also have MFLOPS (Million floating point ops per sec.), generally used to measure performance of super computers.

* To find average values, we use :-
  - for time : Arithematic mean
  - for time ratios : Geometric mean
  - for MIPS / MFLOPS : Harmonic mean.

ISA DESIGN :

The ISA needs to perform a minimal set of arithematic & logical operations taking convenience into account. It should also be able to ~~des~~ communicate with memory (RAM). It should also have instructions for changing control flow.
These provide enough generality to be Turing complete.
A typical instruction will look like :
Operation Operands Destination.
The size of the instructions will be (in bits).

$$= \lceil \log_2(\text{No. of possible operations}) \rceil + (\text{No. of operands} + \text{Dest}) \times \lceil \log_2(\text{size of memory in bytes}) \rceil$$

Note: Running a program with limited RAM: If the size of the program is larger than the RAM size, it is run sequentially, taking only parts out of the secondary storage to the RAM. This has a lot of time overhead.

Since memory is expensive, we try to make the ISA as memory efficient as possible. Another constraint to the ISA is slowness of memory (multiple clock cycles needed).
The ISA should also address ease of programming.

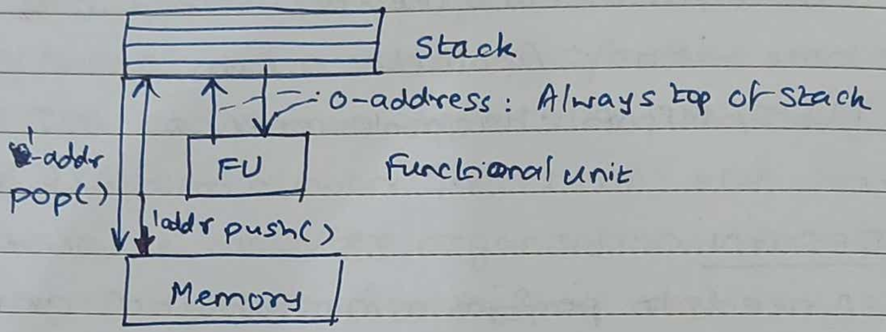
| Op | Operand1 | Operand 2 | Dest. |

: 3 addr format

* Targetting limited memory

In the instruction, we can reduce no. of bits needed for the addresses of the operands.

→ Can we construct a machine where all addresses are implicit i.e. only operator needs to be specified? Yes!



Eg: to calculate A = B+C*D ⇒ Convert to postfix

push C → 1 addr   A = BCD~~*~~*+

push D → 1 addr

mul → 0 addr

push B → 1 addr

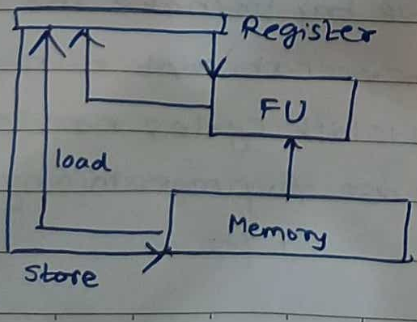
add → 0 addr

pop A → 1 addr.

If all operators are binary, then, in the parse forest, the 3 addr format takes 3x(no. of internal nodes) memory accesses. The 0 addr format takes (No. of leaves + No. of roots) memory ~~address~~ accesses.

In this stack based architecture, stack is the bottleneck

→ 1-address format:



A = B+C

Load B

Add C

Store A

$A = B + C * D$: load C ; mul D ; add B ; store A ;

This is called an <u>accumulator based architecture</u>.

$A = B*C + D*E$: load B ; mul C ; store T_1 ;
load D ; mul E ; add T_1 ; store A ;

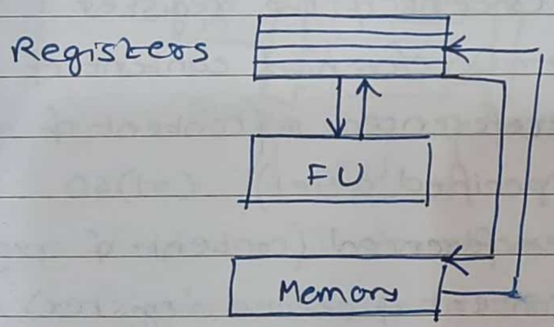
This illustrates that the accumulator is again a bottleneck

→ We can overcome the issue using a small number of registers to act as temporary storage (which also overcomes slowness of memory).

OP	AM	Addr.

↑ Addressing Mode: Register or Main memory.

With the introduction of registers, the accumulator was eliminated. Now, arithematic & logical operations were performed only on registers. Since no. of registers is small, we ~~could~~ can afford to have 3 address format for arithematic and logical opperations



Thus, a 3 address format was introduced on registers.

OP	Reg 1	Reg 2	Reg 3

To transfer data between reg. and memory,

Op	Reg	Mem. addr

$A = B + C$: Load r1, B ; load r2, C ; add r1, r2, r0 ; store r0, A ;

If the number of registers is too small, we may need too many memory references. We can overwrite registers after that variable has died. Based on empirical evidence, we use 32 or 64 registers. If the number of live variables is more, we may have to load & store from memory.

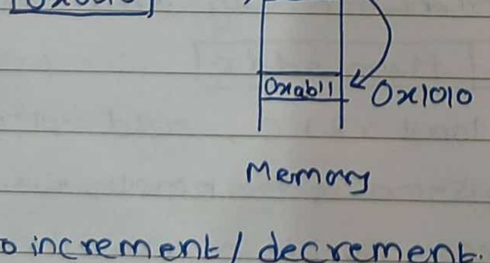
* We have dealt with accessing operands from memory. But, we still need to access instructions from memory. This now becomes the bottleneck. This demands more complex instructions, so that less no. of instruction reads are needed

* Ease Of Programming:

- Pointers: We can store addresses in registers just as we store data. This also needs a dereference operator. This requeires an addressing modes, to specify if we are refering to the content of the register or if it is the data at the address which is content of the register.

  Addressing modes:

  1. Direct Memory: Direct constant address : 0x0ab1
  2. Register Direct: Content of the register : r1
  3. Register Indirect: Dereferenced content of register: (r1)
  4. Base + offset : Dereferenced (content of register + specified offset): (r1)40
  5. Base + index : Dereferenced (content of register + content of second register): (r1)(r2)
  6. Memory indirect:



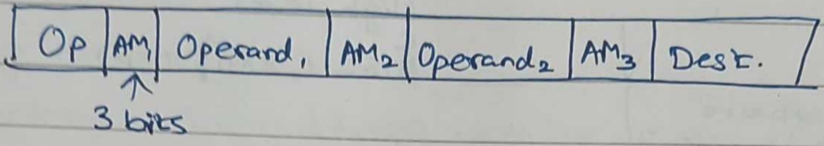
  7. Auto increment / decrement : Used to implement stack: Eg:- push %eax %rsp.

     Means put content of eax register into address contained in rsp, and increment the content of rsp suitably to point to next available language.

* 16 → boundaries

* Youtube : CPU on a breadboard.

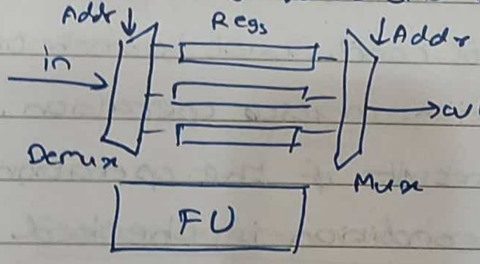
8. Constants: One of the operands is a constant. We can give the value of the constant in the instruction itself. Called Immediate Addressing. There is a limit (usually 32 bits) to the size of constant allowed; beyond that, it has to be treated as a variable.



* Memory is byte-addressable, so the instructions are made in multiples of 8 bits.

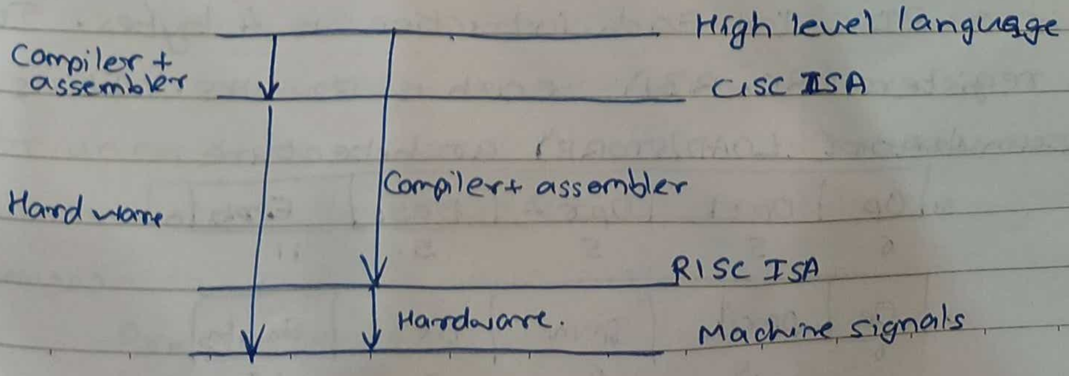
* RISC ARCHITECTURE:

The machine can only use register contents for ALU, and directly use the instructions ~~[illegible]~~ as machine signals



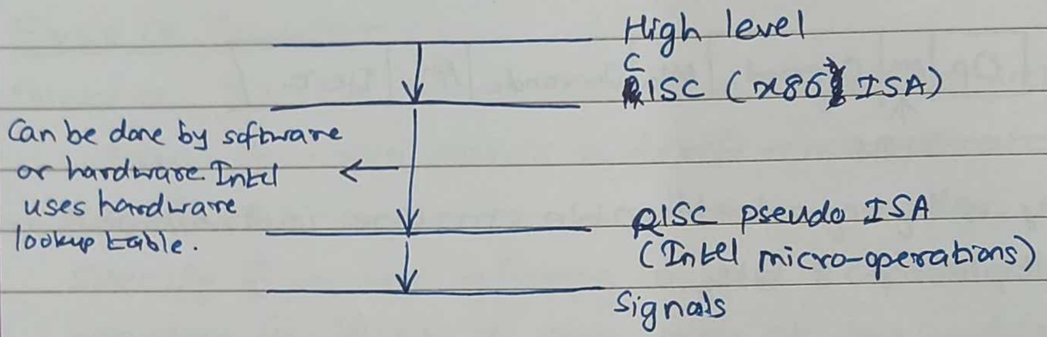
RISC = Reduced Instruction Set Computer. Eg:- ARM
CISC = Complex Instruction Set Computer. Eg:- x86

* Type of encoding: If fixed length (used in RISC) encoding is used, it leads to ~~left~~ lot of inefficiency in CISC. Thus, variable length encoding is used by CISC. This poses problems with decoding and parallel processing of instructions.



(Note: During development we use low compiler optimization as we want it to compile ~~qoo~~ quickly for us to debug. ~~sfns~~ In production, we use high level of optimization).

Intel provides a CISC interface to its software. But internally, it converts it to a RISC pseudo-ISA. This is called ISA indirection.


High level
~~R~~ CISC (x86 ISA)
Can be done by software or hardware. Intel uses hardware lookup table.
RISC pseudo ISA
(Intel micro-operations).
Signals

* Parts of ISA:

- Arithmetic & logical • Memory communication
- Control flow change: Unconditional branch like in functional calls. Also, conditional branch on the result of some arithematic operation. It can happen that the result of the operation gets erased before the condition is checked. Hence, we have CPU flags, which are set when some condition is met. Eg:- carry is 1, result is 0, overflow flags. These flags are updated after every ALU operation. When we perform a jump, the final address may be absolute or relative to current address.

* MIPS ISA:

It is a RISC ISA. Each instruction is 4 bytes. It has 32 registers (R0-R31), each is 32 bits. It is a ~~accumulator~~ LOAD/STORE architecture.

31	Op	Opr1	Opr 2	Dest	Empty	0
	6	5	5	5	11	

31	Op	Opr1	Imm.	Dest	Empty	0
	6	5	6	5	10	

- All ALU ops. are performed only on registers.
  It also allows immediate (constants) addressing mode. To distinguish these, we have different instructions for the immediate variants of the operation. Alternatively, the last 6 bits of the instruction are also used; for register operation, first 6 bits are 0 and last 6 are op-code; for immediate operation, first 6 bits are used for the op-code.
- Memory communication: It has only LOAD and STORE instruction. The address can be given as Base+Displacement or Base+Index

Op	Opr1(Reg.)	Dest	Imm ^{16}
	↑ Base (R1)	↑ R2	↑ Displacement (eg: 80).

This does $R2 = *(R1+80)$ if Op = Load.
and $*(R1+80) = R2$ if Op = Store.
Note: Everything here is byte addressed.
We have instructions: $\underbrace{\text{SB, LB}}_{\text{Store byte / Load}}$, $\underbrace{\text{SH, LH}}_{\text{Store Half / Load word}}$, $\underbrace{\text{SW, LW}}_{\text{Store Word / Load}}$

- For Base+index, we have:

Op	Opr1(Reg.)	Opr2(Reg.)	Dest(Reg.)
	Base	Index	

- Jump: Conditional
  BEQZ Reg, Loc (or BNEQZ).

Op	Opr1	Imm(16)

If Reg = 0, then shift Imm ~~bytes~~ words ahead. ← as each instruction is 1 word long.
(Note that for this, we need a program counter register, which is actually a pointer to the instruction address in RAM).
For unconditional: J Loc

Op	Imm(26)

Always,
Program counter (Bytes) += Imm * 4.

We also have JAL instruction, which loads the program counter to R31 before jumping. This allows function calls. To return, we use JR Reg.

* When we do a jump, the Imm field gives no. of instructions to go relative to the current instruction. But in BEQ / BEQZ, it is w.r.t. next instruction.
Eg :-

SEQ R1, R2, R3 ← Set R_3 to 0 on equality of R_1 and R_2
BEQ R3, 12 ← If $R_3 = 0$, Go to $(12+1) \times 4$ instructions ahead.

∴ A jump instruction takes a 26 bit Imm field ∴ It can move $2^{26} \times 4$ bytes away. For a bigger jump, multiple jumps must be taken.
Also, we have $\boxed{\text{JR Reg.}}$ instruction, which says "jump to "absolute location" given in register Reg. (in words).

* Function calls :- $\boxed{\text{JAL Loc}}$: Jump to Loc (Relative) and store (current PC + 4) in R31 to allow for return.
We also have $\boxed{\text{JALR Reg.}}$
To return, we can then use JR R31.
Note that a recursive function call would over-write R31. Hence, a stack needs to implemented in memory.
To load a constant value into a register, we use $\boxed{\text{LHI Reg. Imm(16)}}$. The Imm is moved to the most significant 16 bits of Reg. Thus to store a 32 bit address, we can load the ~~[illegible]~~ most significant 16 bits first, then add the least significant 16 bits. We also have read-only register R0 which always stores 0.
Thus, we can now have a particular register as stack pointer, so we can have :- (R5 is stack pointer here).

SWI R31, (R5)0
ADD R5, R5, 4

Q. Write code to sum the elements of A[100], where start location of the array is 0x00001000.

```
→ ADD R1, R0, 100              ← 0
ADD R2, R0, 0                  ← 4
ADD R3, R0, 0x00001000         ← 8
LOOP:                          ← 12
~~LO~~ LWI R4, (R3)0
ADD R2, R2, R4                 ← 16
ADD R3, R3, 4                  ← 20
SUB R1, R1, 1                  ← 24
BNEQ R1, LOOP                  ← 28
~~STORE~~ SWI R2, (R3)0        ← 32
HLT                            ← 36
```

So, Loop translates to $-\frac{(32-12)}{4} = -5$ at instruction at address 28.

* In a function call, when we pass variables, they are stored in the stack. Then, as needed, they can be stored in registers. This also necessitates preserving the registers on the stack.

* ARM ISA:
- Predicated execution: Eg:- ADDZ R1, R1, #1
  $\uparrow$ if zero flag is set.

  The flags can be set by instructions suffixed by S, like MOVS
- Programable ~~p~~ program counter: R15 is PC. You can carry out a jump like ADD R15, R15, R1.
- Instructions to push and pop all registers to/from memory.

✱ MACHINE ORGANISATION FOR RISC (~~[illegible]~~ MIPS):

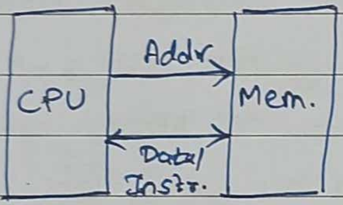
Simple instructions, few instruction formats, fixed encoding

Op	Opr1	Opr2	Dest		Func	R
6	5	5	5	5	6	

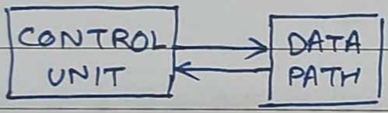
Op	Opr1	Dest	Imm16	I
6	5	5	16	

Op	Imm 26	J
6	16	

Let us assume that all ALU ops are R type, all memory addressing is base+displacement (I type), BEQ is I type and Jump is J type.



CPU looks like:



The data path ~~[illegible]~~ consists of:-

1) Functional units : Eg: ALU
2) Storage : Programmer's Registers, Program counter, temporary registers, cache
3) Steering logic (communication infrastructure):
   It can be broadcast based (IO Bus + Tristate logic), or point to point (Multiplexor based).

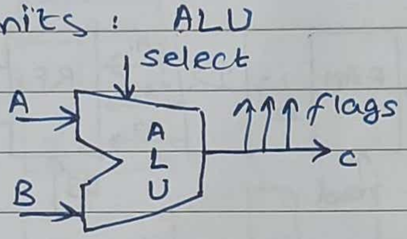
The control unit can either be based on combinational logic or finite state machine.

The state of the machine as visible to the programmer is given by memory+registers. State transitions can ~~also~~ occur in one clock cyle or multiple clock cycles.

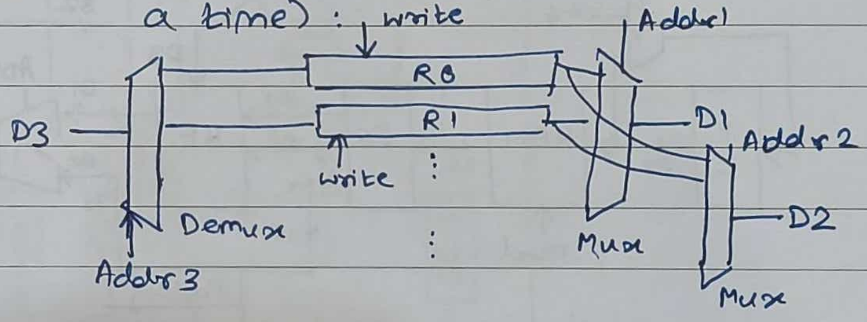
For the multiple cycle model, the control unit has to be designed as a finite state machine (~~wit~~ where the internal states aren't visible to the programmer).

* DESIGN :-

- Functional units : ALU

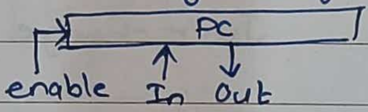


- Storage ~~Memory~~ : Programmer's registers (can read only 2 at a time) :

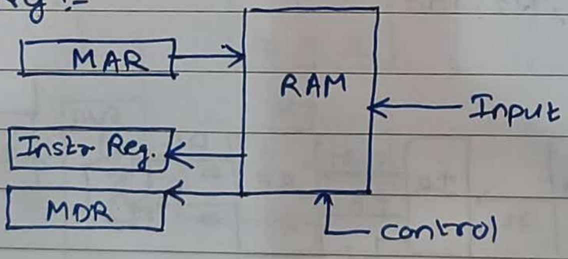


Register file.

Program counter : Single register :



- Memory :-

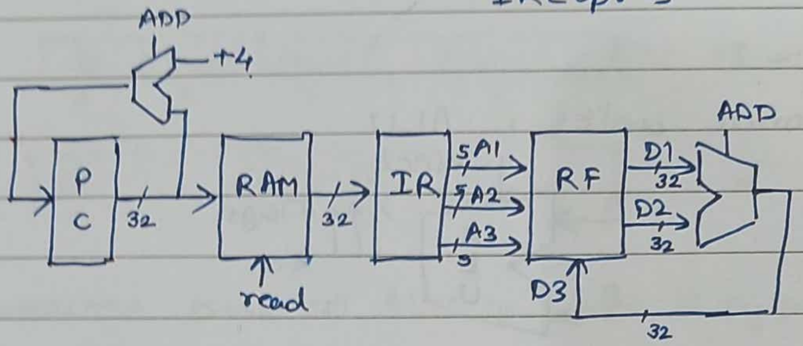


* A simple instruction in execution :-

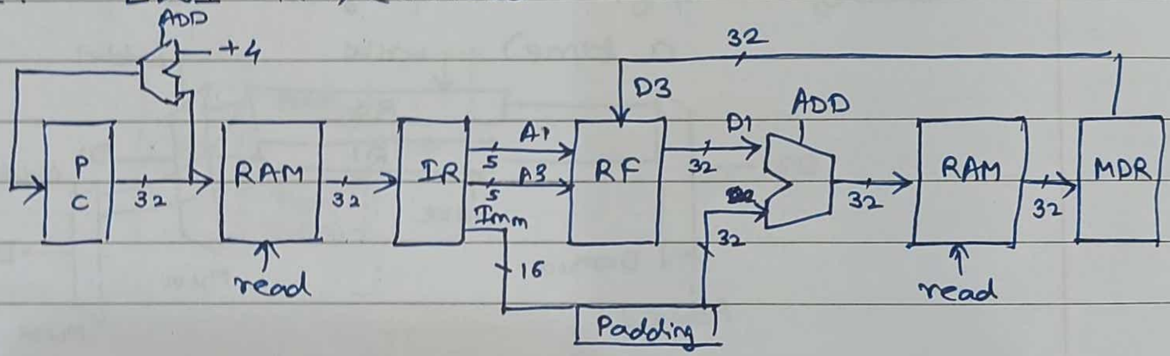
- R Type ; ADD R1, R2, R3 :

Micro~~prog~~code :-
- PC → Memory → IR
- Increment PC by 4 : PC → alu_A, 4 → alu_B, alu_C → PC
- IR [Operation] → Control unit

- IR[Opr1] → Registerfile A_1 → D1 → Alu_a
- IR[Opr2] → Registerfile A_2 → D2 → Alu_b
- Add in ALU → Alu_c → D3 of RF → Registerfile.
  IR[Opr3] →



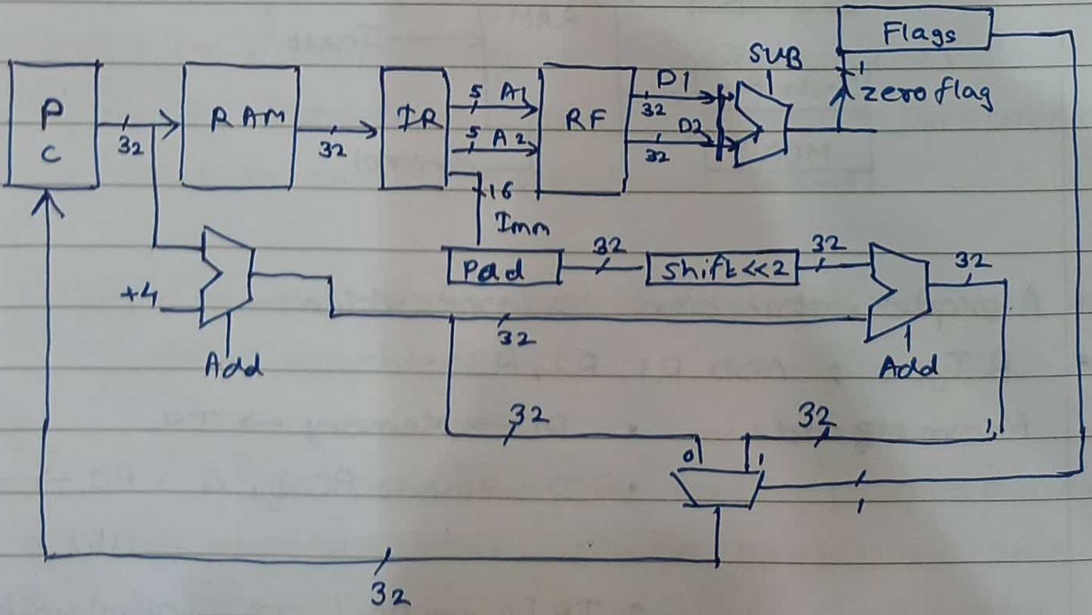
- I Type: LWI R2, (R1)1000



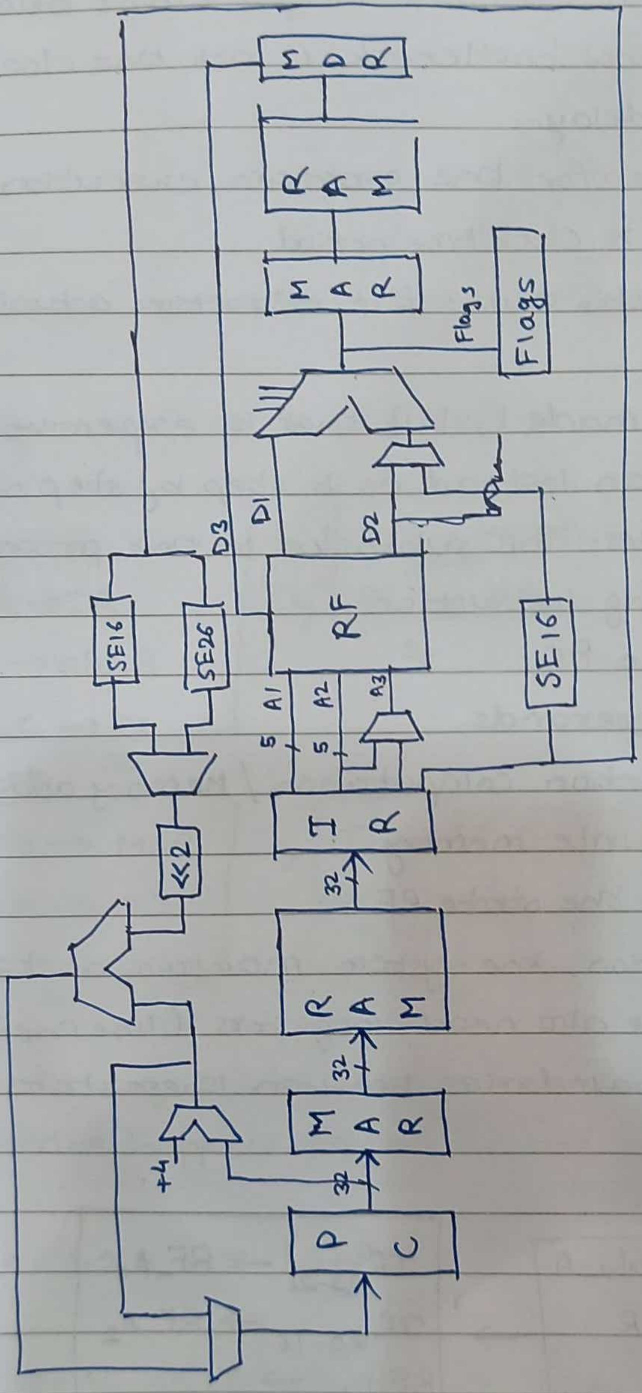
SWI R2, (R1)1000

Just change read to write and take D2 as input

BEQ, R1, R2, 100



* Full RISC Machine :



All these devices need control signals, which need to be supplied parallely/ sequentially. The circuit which does this is called the controller. This can be constructed using combinational logic.
The time taken depends on the longest (most delayed) path. The technological bottleneck is not the clock speed, it is the circuit delay.
The total time taken for the program execution

$= \underset{\sim}{IC} \times CPI \times$ Clock time period

Dynamic instruction count: # of instructions actually executed.

~~[illegible]~~ The CPI can be made 1, but that is expensive in terms of resources. We can instead do it step by step over multiple clock cycles. The subtasks in this process are generally :
1) Fetching instruction
2) Update PC
3) Read operands
4) Instruction computation / Memory address
5) Read/write memory
6) Update the ~~state~~ RF

Thus, for an instruction, the system maintains a state for each clock cycle. We also need registers (like the IR, MAR, MDR) at the boundaries between these states.

- R type: 1,2,3,4,6

Internal states:-

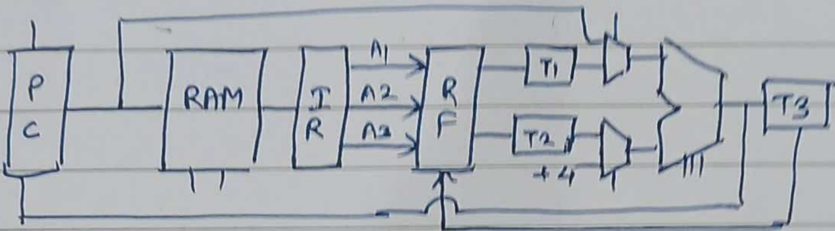
$$\begin{tikzcd} \boxed{\begin{array}{l} PC \rightarrow MAR,\ alu=A \\ RAM \rightarrow ~~MDR~~\ IR \\ +4 \rightarrow alu\_B \\ alu\_C \rightarrow PC \end{array}} \arrow[r] & \boxed{\begin{array}{l} IR_{25-21} \rightarrow RF\_A_1 \\ IR_{20-16} \rightarrow RF\_A_2 \\ RF\_D_1 \rightarrow T1 \\ RF\_D_2 \rightarrow T_2 \end{array}} \arrow[r] & {} \\ S_0 & S_1 & \end{tikzcd}$$

$T1 \rightarrow alu\_A$
$T2 \rightarrow alu\_B$
$alu\_C \rightarrow T3$

S_2 $\longrightarrow$

$T3 \rightarrow RF\_D_3$
$IR_{16-11} \rightarrow RF\_A_3$

S_3



• I type : 1, 2, 3, 4, 5, 6 (LW)

$PC \rightarrow MAR,\ alu\_A$
RAM ~~MDR~~ $\rightarrow IR$
$+4 \rightarrow alu\_B$
$alu\_C \rightarrow PC$

S_4 $\longrightarrow$

$IR_{25-21} \rightarrow RF\text{-}A_1$
$RF\text{-}D_1 \rightarrow T_1$

S_5 $\longrightarrow$

$T1 \rightarrow alu\_A$
$IR_{15-0} \rightarrow SE16 \rightarrow \ll 2 \rightarrow alu\_B$
$alu\_C \rightarrow T3$

S_6 $\longrightarrow$

~~S_8~~

$T3 \rightarrow MAR$
$MDR \rightarrow T3$

S_7 $\longrightarrow$

$T3 \rightarrow RF\_D_3$
$IR_{20-16} \rightarrow RF\_A_3$

S_8

* Note : In any state, read happens ~~[illegible]~~ in the beginning.
Write happens for all registers at the very end.

• I type : 1, 2, 3, 4, 5 (SW).

$PC \rightarrow MAR,\ alu\_A$
$RAM \rightarrow IR$
$+4 \rightarrow alu\_B$
$alu\_C \rightarrow PC$

S_9 $\longrightarrow$

$IR_{25-21} \rightarrow RF\_A_1$
$IR_{20-16} \rightarrow RF\_A_2$
$RF\_D_1 \rightarrow T_1$
$RF\_D_2 \rightarrow T_2$

S_{10} $\longrightarrow$

$T1 \rightarrow alu\_A$
$IR_{15-0} \rightarrow SE16 \rightarrow \ll 2 \rightarrow alu\_B$
$alu\_C \rightarrow T3$

S_{11}