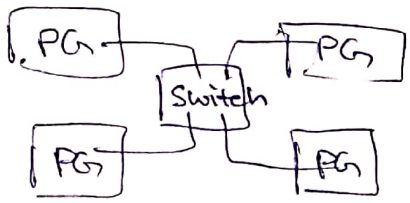
Programmable System Design:-

Flexible System:-

Implement using flexible fabric.

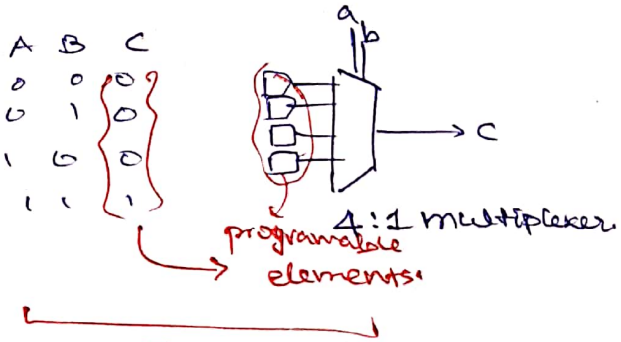
FPGA: field programmable gated Arrays
while in use.



PG: implement using LUT.
Lookuptable.

Eg: AND gate:-

A ∧ B = C



* 5 input LUT 32 storage 32:1
6 input LUT 64 storage 64:1

we get flexibility.

Redesign circuit →

Map on FPGA

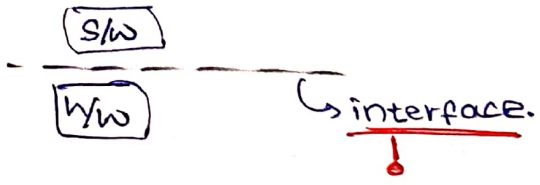
↓

1) Map logic on LUT 2) interconnect.

Programmable Fabric:-

* Generic hardware; which can be programmed as per requirement.

S/w code



Program
translates State of the systems
from one state to another.
output depends on inputs & State.
mealy machine.

State S ⟶ State S'
program

Ex. in this state transition,
all required computations
are done.

Eg: program is

$a \leftarrow b + c$

$d \leftarrow d + e$

S:	prog.	S':
a=0	⟶	a=30
b=10		b=10
c=20		c=20
d=30		d=70
e=40		e=40

Ta-da

* we need to have interface!

instruction set ← language the generic hardware can understand & act on.

ISA: instruction set Architecture.

Instruction Set:- ⟶ structure of computer, a machine language programmer must understand, to write a correct (timing ~~independent~~) program for the machine.

↓

machine description,

a hardware designer must understand; to design a correct implementation of computer.] generic hardware.

Eg: Statement in C++:-

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

Assembly instructions:-

add to, g, h

add t_1, i, j

sub f, to, t_1

{ op-code/mnemonic , operand , source/destination }

* So; what instructions should an instruction set have?
can it have (a+b)-(c+d) as a single instruction with four operands? I mean; is this worth it?
isn't this too specific.
unnecessary!

church's thesis:-

A very primitive computer can compute anything, that a fancy computer can compute.

- you only need
  1) logical functions
  2) read and write to memory
  3) data dependent decisions. } if-else conditions??

∴ General ISA selection is for practical reasons of cost & performance. Computability is not an issue at all.

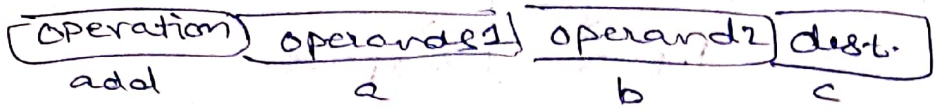
like; ISA might have direct instructions for additions.

→ nvidia have raytracing cores
→ Apple M1 has ML cores.
→ normal processors have floating point cores.

* If our processor doesn't have "+" instruction; then the S/W needs to send the (a+b) instruction in terms of
→ $b_1 \wedge b_2$
→ $\neg b_1$
etc...
"pain in the ass".
But possible.

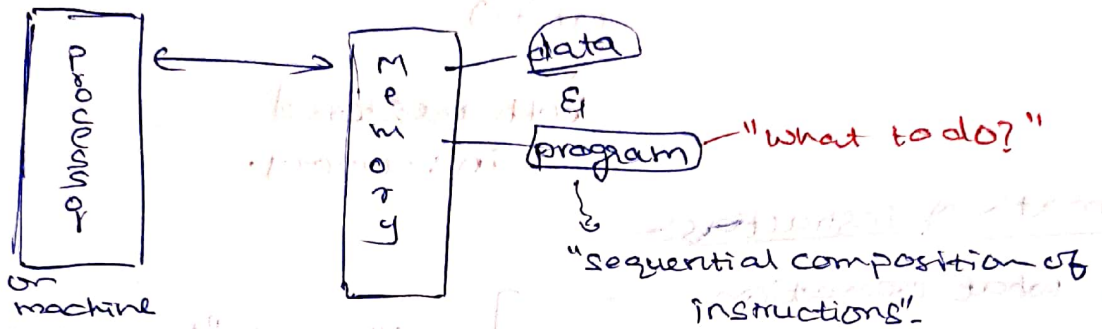
Usually '+' instruction is available.
one of basics.

* Every instruction be like:-



"need some format for EVERY instruction".

→ computing system:- (oh lord! I'm tired!)



* Whenever we write on a high-level language, it needs to be compiled; and this compiled binary contains machine languages & depends on ISA we are using.

All 3 have very different languages.

- x86 - intel, AMD
- ARM - mobile, tablet, M1
- POWER - IBM.

arm technologies.

* Now the question is, how do we develope such a language which is "sufficient". lets work on it.

we need

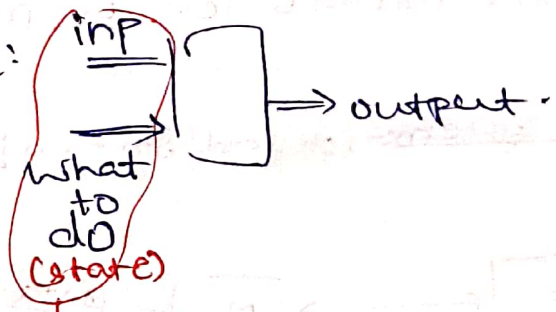
- Logical operations (OR arthimatic too!)
- memory communication
- control flow change. (branches)
  - data dependent actions.
  - like if (i == j) then this line; else that line.

a primitive computer got to have these. Mine has many more. Hehe.

* First, we need to decide upon interface....

if machine was fixed;

but since machine is generic:



both are stored in memory.

* Formats of instructions:-

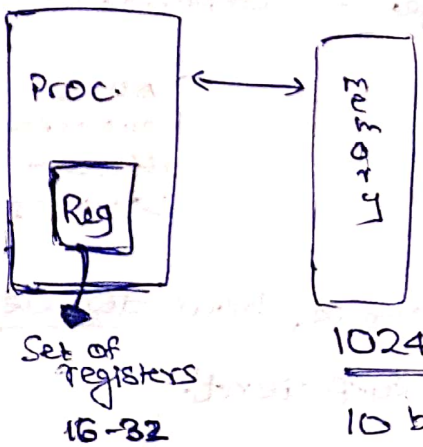
1) what operation
2) who are operands
3) where to place result.

] we'll need "temporary" space.

Registers!

State of system:-

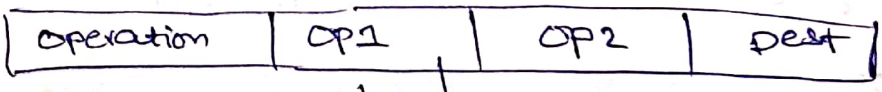
- content of memory
- content of registers.



Set of registers
16-32

1024 bytes. (1KB)

10 bits; enough to positioning the memory.

* 

memory
Eg; 200

register
Eg; R5

- addressing modes.
  - Base + offset
  - constant
  - exact loc.

→ Kind of instructions

1) Add/ Sub/ AND/ ¬ (logical)

2) LW Load. read from memory.

Eg: load R1 a

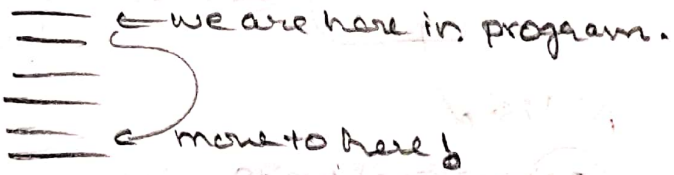
R1 → register (R1)

a → memory location.... (500)

3) SW Store write to memory.

Eg: Store R1 a

3) Jump <to location>.
(unconditional branch).
maybe we have function.



Like "go to".

(or)
conditional branch:-

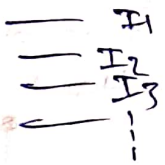
BZ R_1 R_2 loc. BEQ: branch on equality.

if $R_1 == R_2$; then Jump to 'loc'.

Otherwise go to seq. next instructions.

* we'll have one special register; to keep track of our location in list of instructions.

Program Counter (PC) / Instruction pointer (IP)



tracks where we are.

(Keep pointer to the NEXT instruction).

* okay! we have instructions set now!

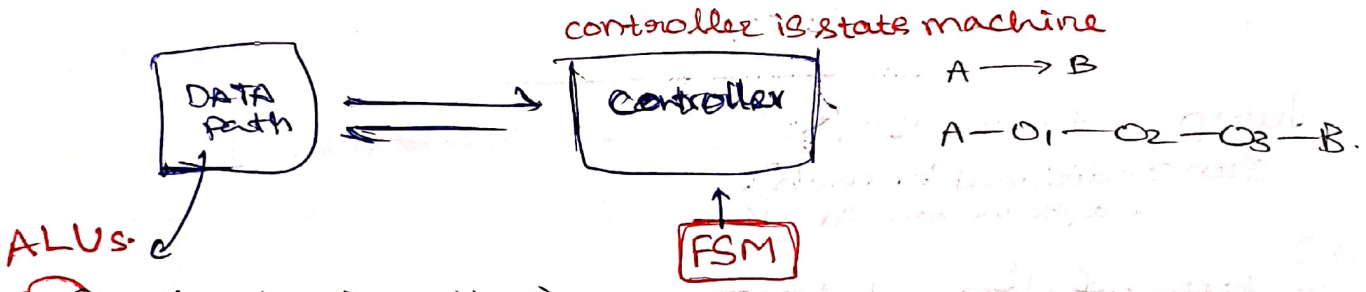
How to design the generic hardware! (a.k.a. processor)

what be the components?

1) Resources to perform the job } DATAPATH

2) mechanism to instruct } Controller.

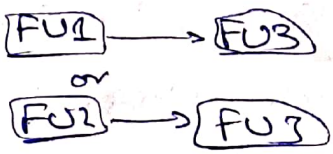
* inside processor:-



- Functional units (Add, sub)
- Storage to store temp. data } registers — some for instruction set
  - some t1, t2 .. for temp. storage.
- steering logic. for data movement.
  - just logic! MUXES are in place.
  - Broadcast
  - point to point.

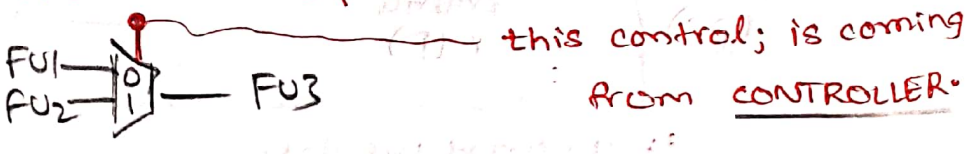
→ Controlled by controller.

Eg:



gotta do in controlled way.

Have a multiplexer.



* DATAPATH is a slave.

CONTROLLER is intelligent.

* Lets dive more into instructions....

ADD/AND/SUB ; for simplicity; say that all operands are to be in registers.

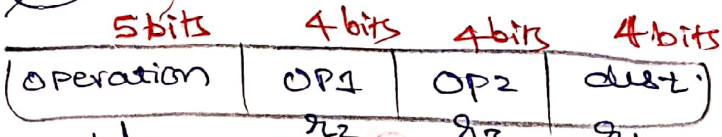
can't read from memory on-the-go.

Eg:

say; I have 20 operations overall.

∴ use (LW) for memory to register.

(LW)



add

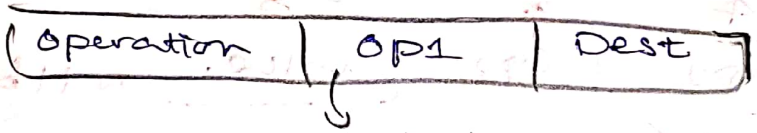
say OP1, OP2 are registers. $r_1 = r_2 + r_3$.

we have 16 registers (ARM has 16)

∴ 4 bits enough to address

R0 - R15

- **LW instruction** (memory read)



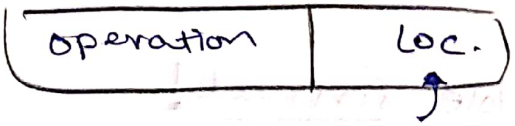
location in memory.

Say Base + offset kind of mode.

Stored in a register. hmmm.. 4 bits.

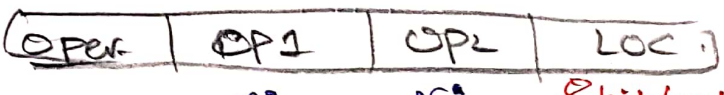
give direct value (not stored in address...) 8 bits.

- **Jump location:-**



maybe, relative to current value.

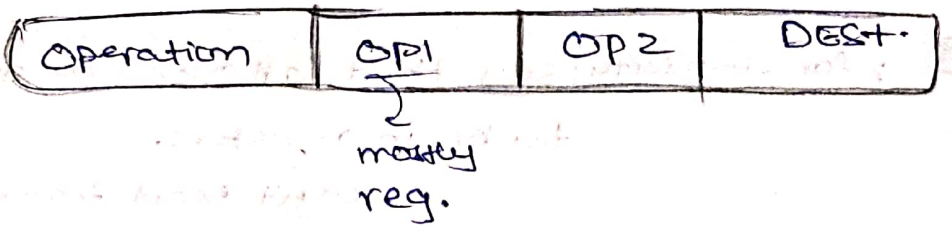
BEQ



if $r_i == r_j$; go to loc.

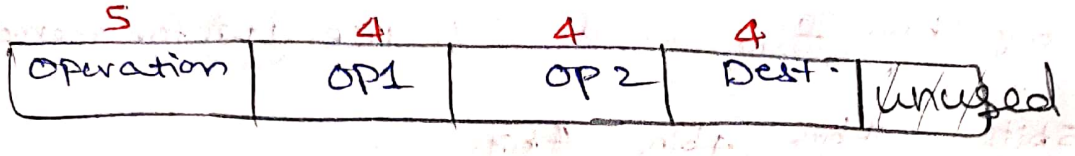
Otherwise next instruction.

- uniform format of instructions:-



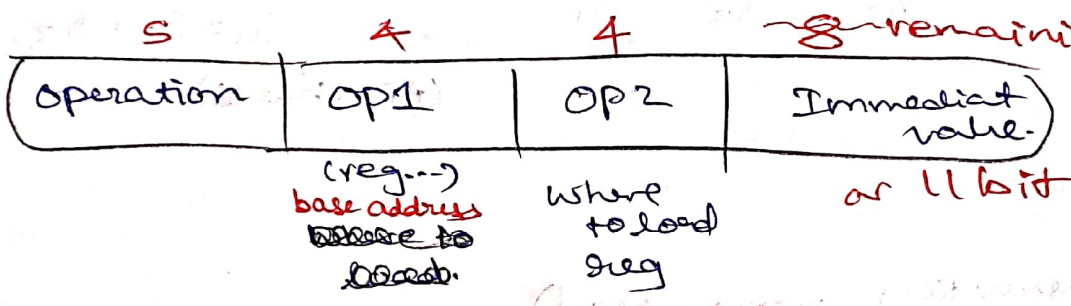
Arthimetic;

17 bits



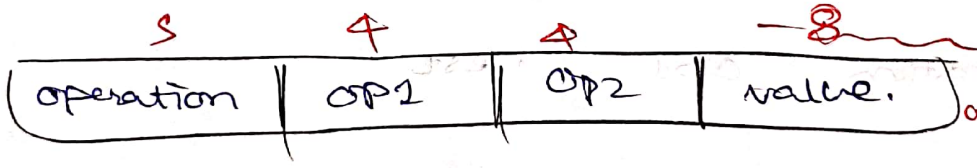
LW; SW:

21 bits

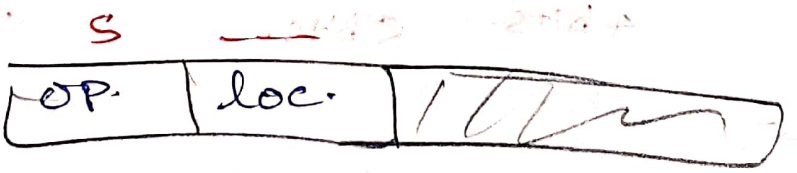


BEQ;

21 bits.



Jump;



* we need in byte addressable format!.

24 bits VV.

ALU: (or functions in DATAPATH)

< L29; Pg 17–21 >

See how ALU are made & control is provided by MUX.

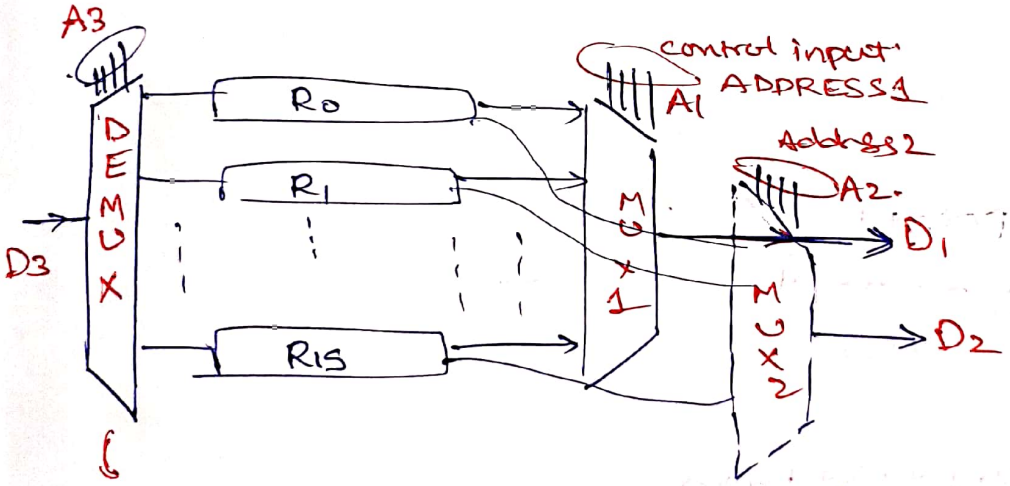
Add / Sub ✓ Shift ✓ AND OR MUX ✓

Storage: - Registers → Programmers registers + S/W. (S/W) + temp. registers + PC. H/W

R_0 — R_{15}. Acting simultaneously.

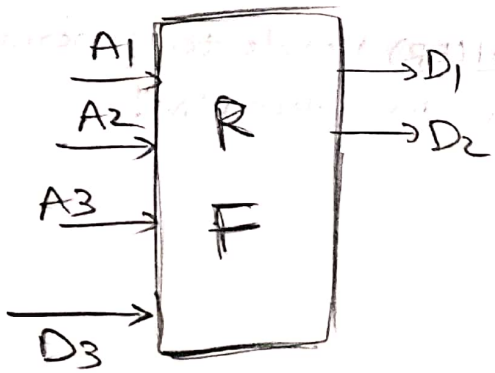
register file.

- each register ↔ multiple flipflops.
- write in parallel; read in parallel.



We'll write atmost 1 register at per operation.

We need 2 independent MUX; Since we'll need; parallely two values in operation.



* we'll need some more temporary registers.

* we'll need a program counter.

*

PC

↳ atleast able to address each MEMORY

∴ 4GB memory; 32 bit PC.

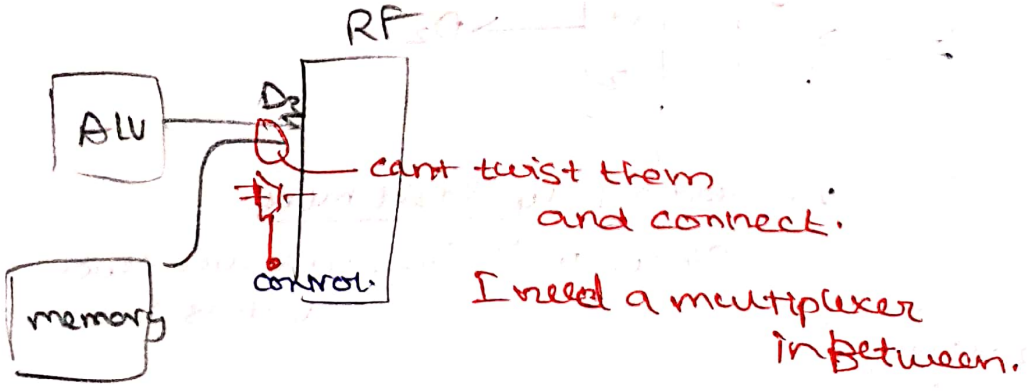
say; 24 bit PC.

∴ Storage:- RF + PC + temp. registers

needed for intermediate data.

Steering logic:-

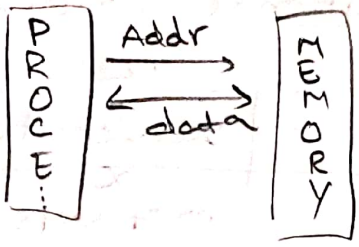
↳ point-to-point.



"State machine (aka CONTROLLER) needs to generate these control signals based on situation"

x——x

* one more time....

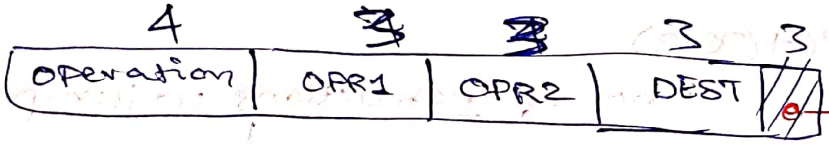


say processor has 8 registers.
↑
3 bits to address.

& say total 10 operations defined
4 bits to symbolize

1) Arithematic & logic:-

16bit:



→ memory (Enhance instructions) is byte addresable.

ADD r_1 r_2 r_3

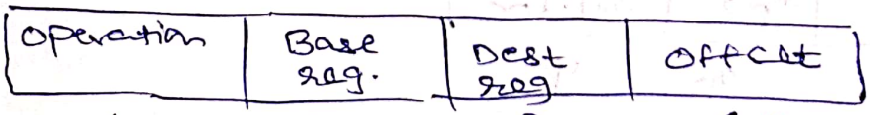
$r_3 = r_1 + r_2$

2) load and store instructions:-

operation, reg, mem_location

as base + offset (offset: supplied directly in instruction.)

base value; Stored in register



16bit: 4 | 3 | 3 | 6

$6 // 2^6 = 64$ bytes in memory.

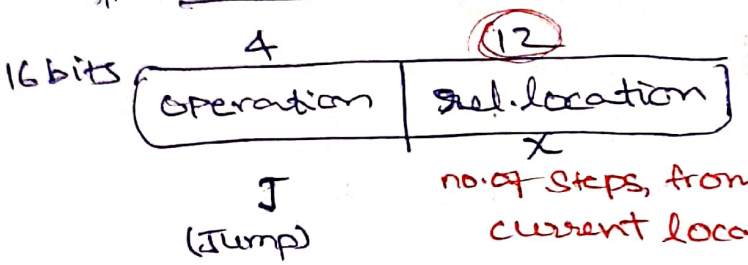
r_2 r_1

Store the data in memory into r_1.

Same for SW.

3) Branch:-

* unconditional GOTO; function calls...



normally $PC = PC+2$.

but now,

$PC = PC+(X-2)$

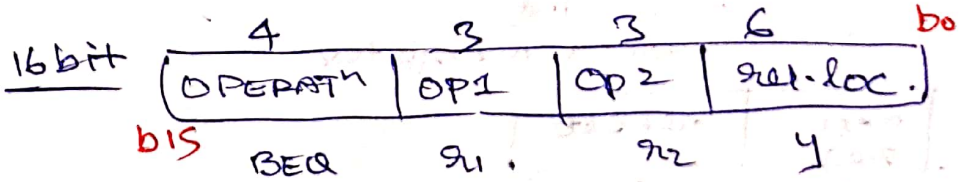
↳ no. of instructions to jump.

each is 2 bytes!

pointer in memory; indicating the next instruction

* conditional: (data-dependent)

BEQ r_1 r_2 rel. loc.



if $r_1 = r_2$ (their stored data) then

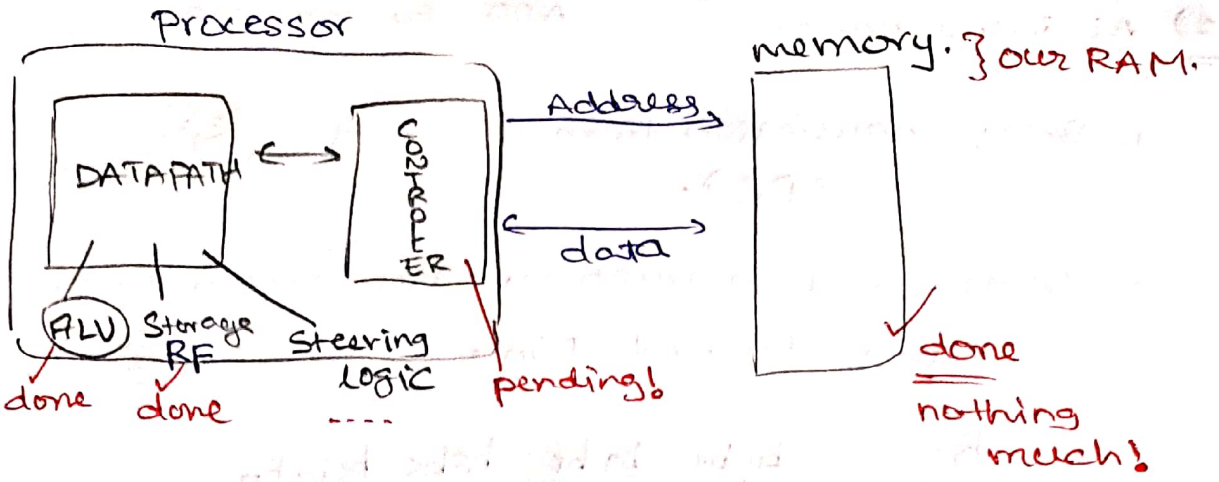
$PC = PC+2Y$

else

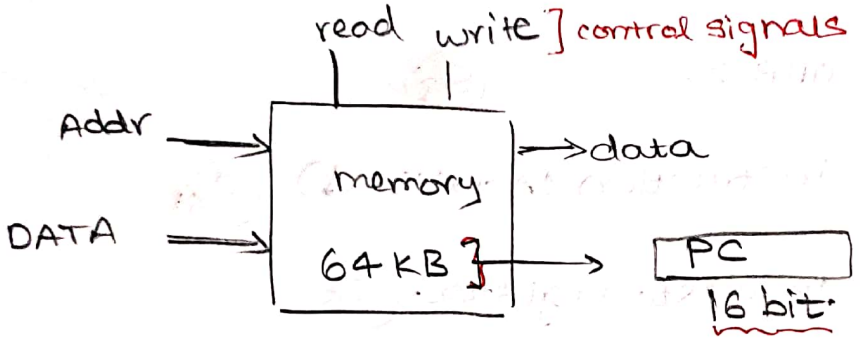
$PC = PC+2$

* This is a minimal set of instructions.

< L30 pg7 > } just write everything together.



* memory API :-



2^{16} = 64 KBytes addresable.

memory is Byte Addresable only

* Now; lets see our controller.

FSM!!

lets see what all our controller has to do,

what are its States &
State transitions

What would be our
Steering logic....

* 16 bit adder;

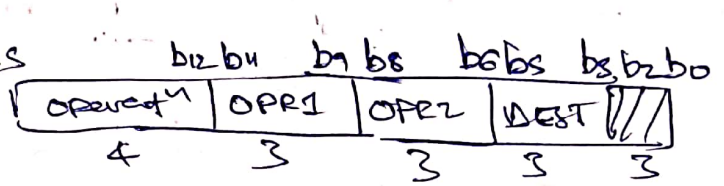
each register → 16 flipflops....

1) AL instruction:- (Arithematic, Logic)

ADD R1 R2 R3

1. Bring instruction from memory (PC). (S1)

2. Understand instruction. Easy; since we have a clearly defined ISA. (S2)



3. Read OPR1 & OPR 2. (S3)

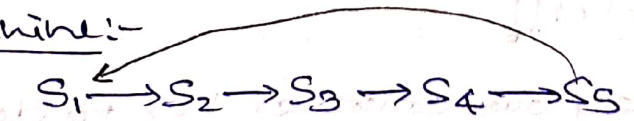
4. Compute the instruction (Addition!) (S4)

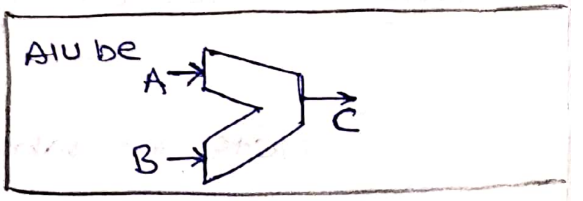
5. Write result in dest. register. (S5)

6. update PC. PC = PC+2.] this job can be done in first step only.

" these are the 5 states I need to pass through; to complete the instruction"!

State machine:-



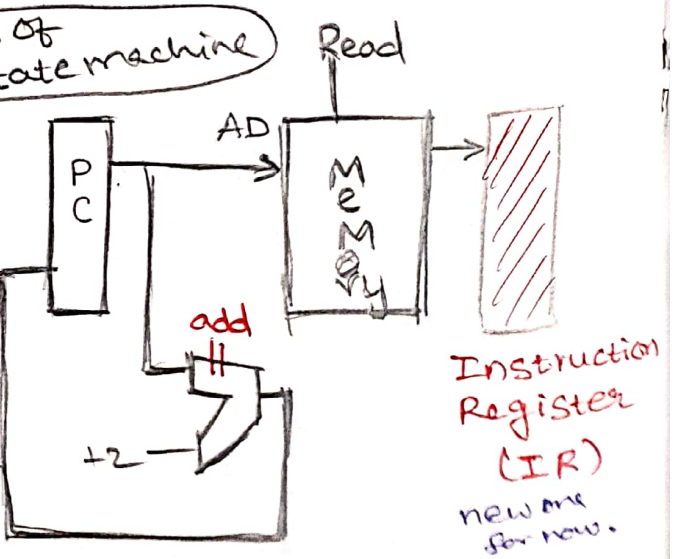


* now, we need to define the State transition logic & the outputs for every state.

* S1:

Tasks	control signals
PC → Mem. addr	• MR → memory read
mem.data → IR	• PC_W → PC write
PC → ALU_A	• IR_W → instruction register write
+2 → ALU_B	• alu_add
ALU_C → PC	

outputs of state machine



merge S_2 & S_3 → S_2'

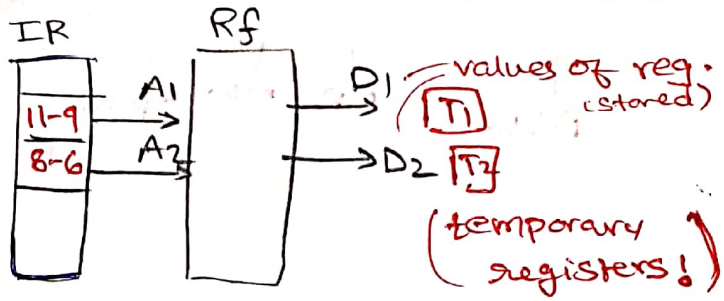
nothing much to do.

S_2':

now; IR has instruction....

we'll give A_1, A_2 to muxes; we'll get D_1, D_2.

tasks	control signals
$ir_{11-9} \rightarrow rf_{A1}$	• t_1_w
$ir_{8-6} \rightarrow rf_{A2}$	• t_2_w
$rf\_D_1 \rightarrow t_1$	
$rf\_D_2 \rightarrow t_2$	



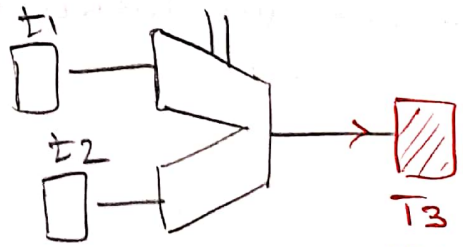
D_1, D_2; will initially be written into RF using LW instruct^n.

– By SOFTWARE.

S_4: (execute instruction)

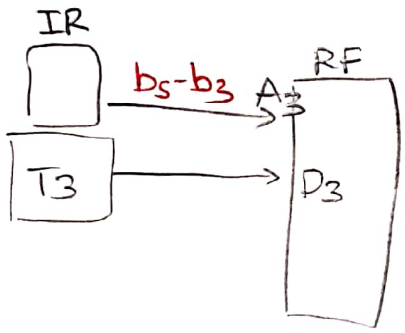
control given from instructions.

tasks	control signals
$t_1 \rightarrow alu\_A$	• alu-opcode
$t_2 \rightarrow alu\_B$	• t_3_w
$alu\_C \rightarrow t_3$	



S_5: (update in RF) maybe memory will be updated later, by SW instruction.

tasks	control signals
$t_3 \rightarrow rf\_D_3$	• Rf_w
$IR_{5-3} \rightarrow rf\_A_3$	

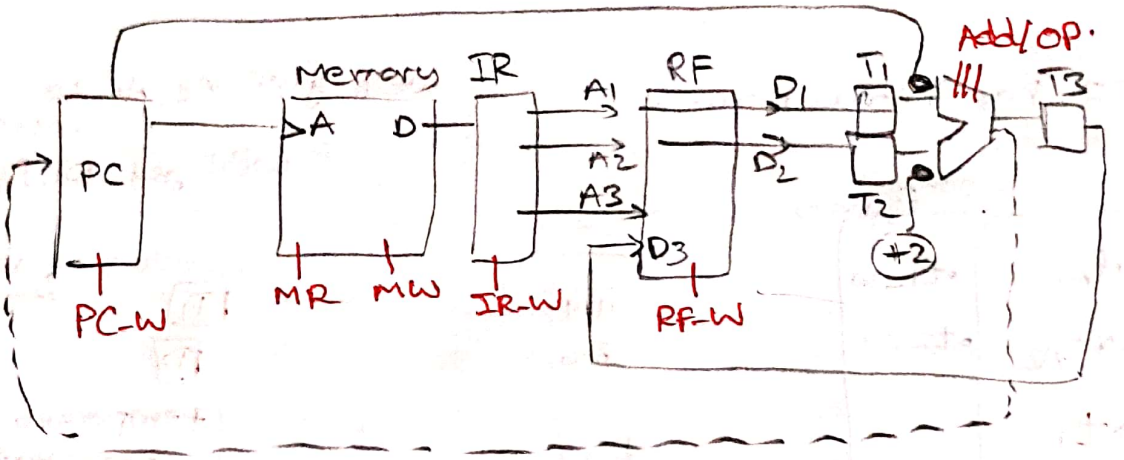


∴ register address = A_3

value written = D_3

∴ we've shown $S_1 \rightarrow S_2' \rightarrow S_4 \rightarrow S_5$

* we get an idea of complete datapath:-



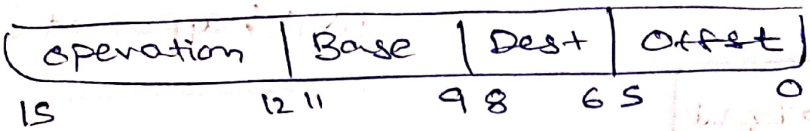
Registers : IR, t1, t2, t3.

Controls: for t1W
t2W
t3W
for ALU

∴ we have a state machine, which can perform AL instruction.

Lets see other instructions.

2) Load Instruction:-



State operations:-

S6 1) read instruction from memory. & update PC.

S7 2. Understand and read the base address of memory.

S8 3. compute the address of memory (Baseaddr + Offset)

S9 4. Read memory

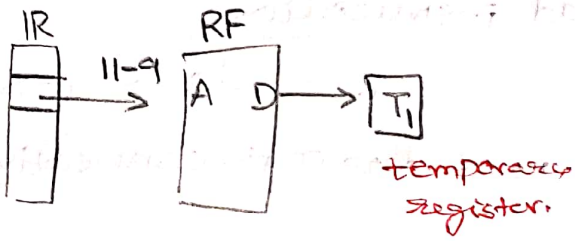
S10 5. Update the register.

S6:-

Same as S1.

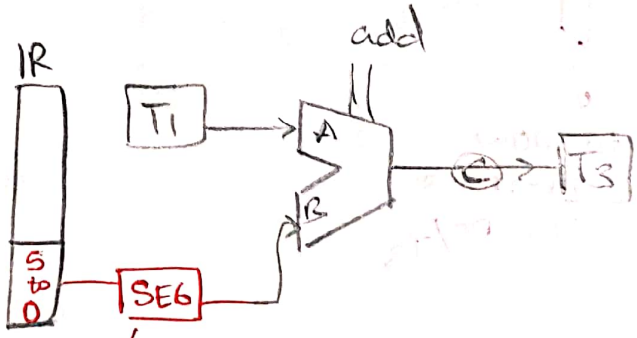
S7:- (read base mem. address)

$ir_{11-9} \rightarrow rf\text{-}A_1$ $rf\text{-}D_1 \rightarrow T_1$	$\cdot T_1\text{-}W$



S8:- (compute the mem. loc.)

$t_1 \rightarrow alu\text{-}A$ $IR_{5-0} \rightarrow SE6 \rightarrow ALUB$ $alu\text{-}C \rightarrow t_3$	$\cdot alu\text{-}add$ $\cdot T_3\text{-}W$

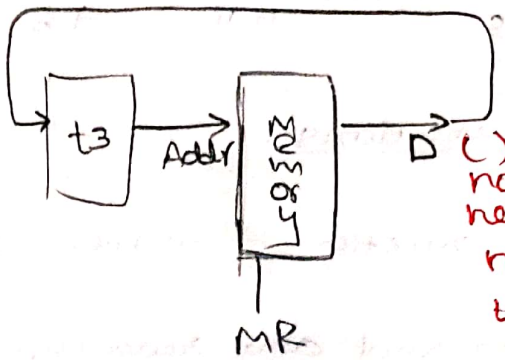


Sign extender from 6 bits to 16 bits.

S9: (memory read)

"read will happen during cycle; write will happen at end of clock!"

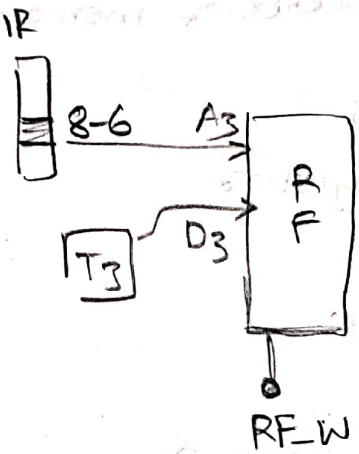
$t_3 \rightarrow$ mem-Addr	• t_3-W
mem-D $\rightarrow t_3$	• MR



() no need of new temp. registr.

S10: (update reg).

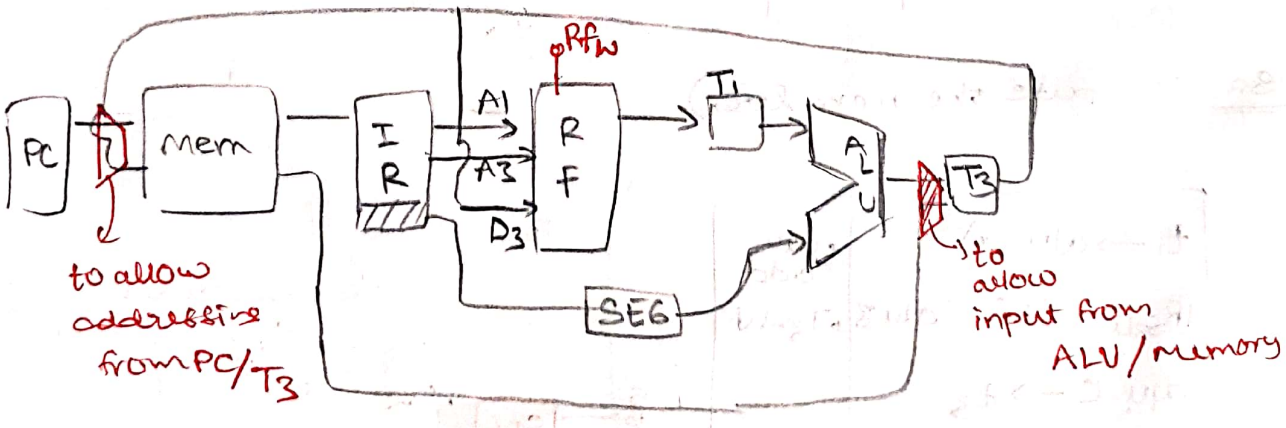
$t_3 \rightarrow$ Rf_D3	• Rf_W
$ir_{8-6} \rightarrow$ Rf_A3	



∴ Load instruction

$S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 - S_{10}$.

We'll need datapath connectivity as :-



control signals → output of state machine

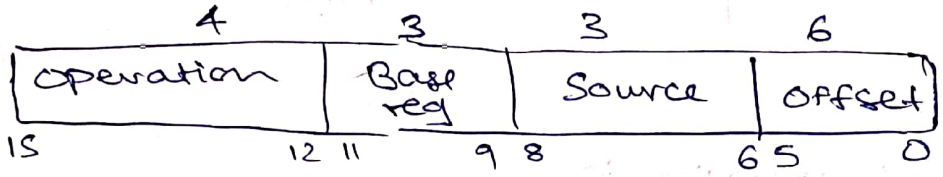
3) Store instruction:

S1 1) read instruction from memory; & update PC.

S12 2) Understand & read operands ⇐

S13 3) Compute the address

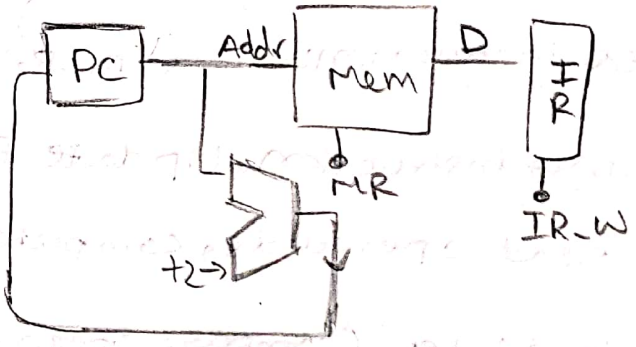
S14 4) write to memory.



S11:

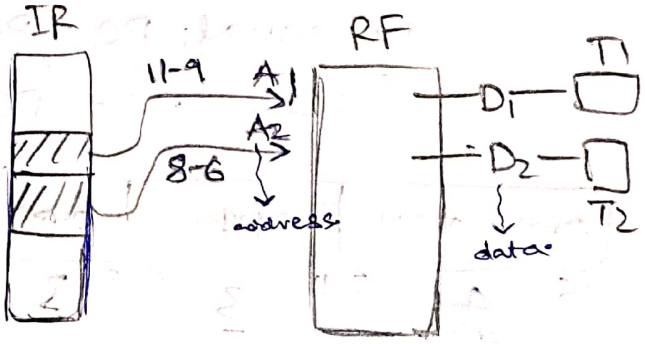
same S_1

PC → mem-A	• PC-W
memD → IR	• IR-W
PC → alu-A	• ALU-add
+2 → ALU-B	
ALUC → PC	



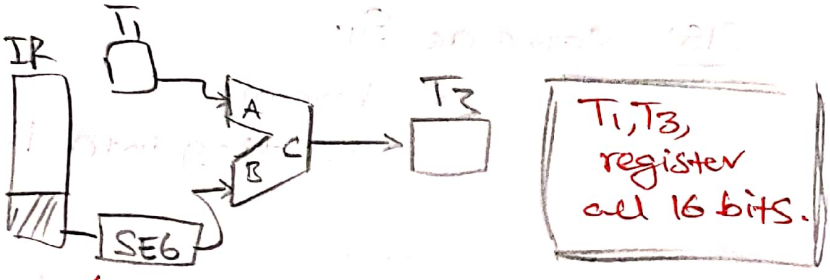
S12: (read both) ✓

$ir_{11-9} \rightarrow rf\_A1$	~~no control~~
$ir_{8-6} \rightarrow rf\_A2$	since we are reading · no control.
$rf\_D_1 \rightarrow t_1$	• t_1-W
$rf\_D_2 \rightarrow t_2$	• t_2-W



S13: (compute mem_LOC).

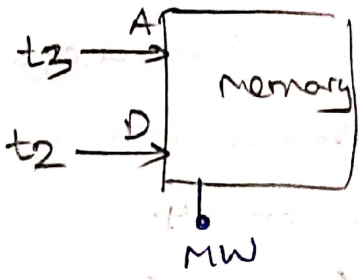
$t_1 \rightarrow$ alu-A	• alu-add
$ir_{0-5} \rightarrow$ SE6 → alus	• t_3-W
alu-c → t_3	



Sign extender 6 to 16 bits.

S14: (write to memory)

$t_3 \rightarrow mem\_A$ $t_2 \rightarrow mem\_D$	MW



∴ Store instruction is

$$S_{11} \rightarrow S_{12} \rightarrow S_{13} \rightarrow S_{14}$$

---

4) BEQ instruction:- Beq R_1 R_2 rel.loc.

S15 1) read instruction. Update PC by 2.

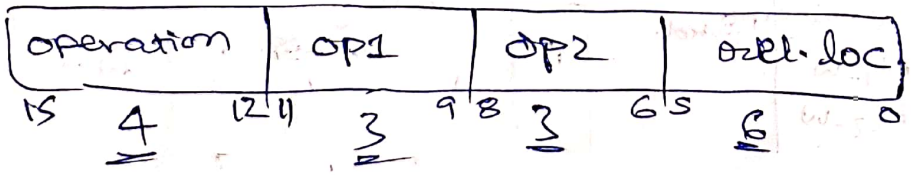
S16 2) read operands, compute address for taken branch.

S17 3) execute! (compare operands & make decision).

if equal; PC = PC + 2·offset + 2

else PC = PC + 2

→ see how offset is defined! additional increase to the regular increase of 2 bytes.

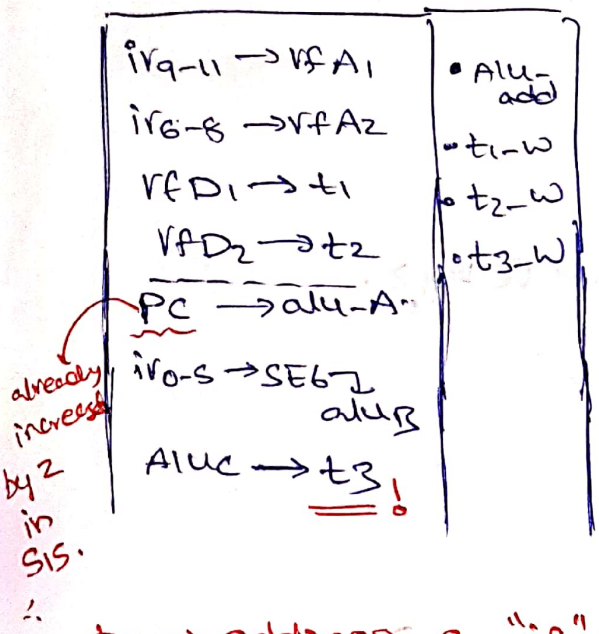


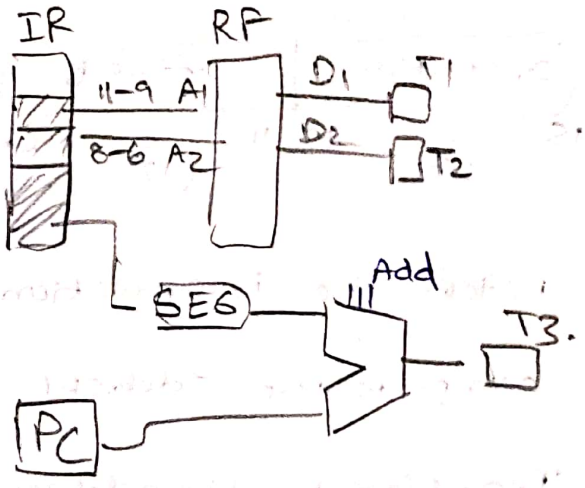
S15: Same as S_1.

✓ hmmm...

reading into instruction register.

S16: (read operands; pre-calculate the taken value for PC)

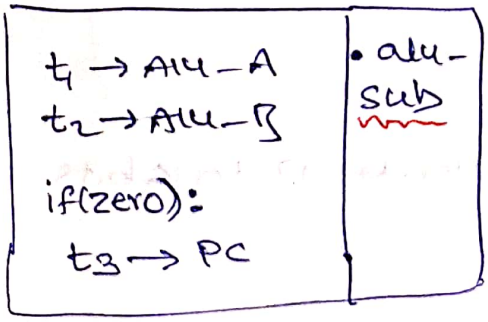


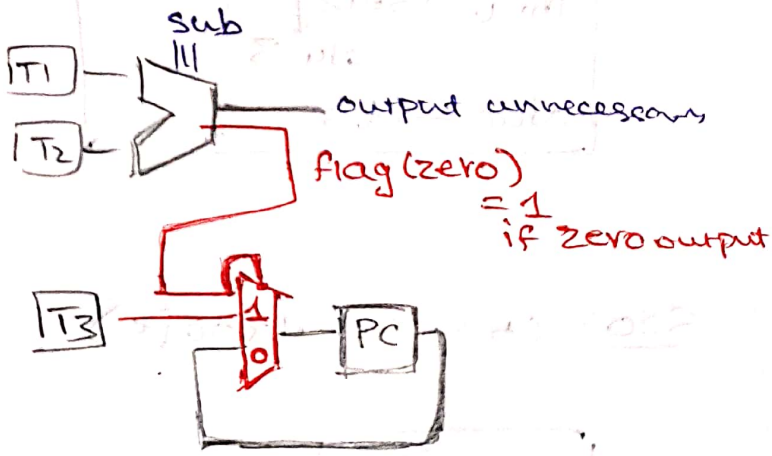


t_3 → address for "if" case

PC → address for "else" case

S17: (compare; & decide for PC)

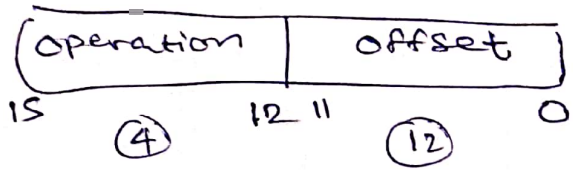




• we'll need a zero detector.

5) Jump:-

do PC = PC+2+2(Offset)



S18

1) Fetch the instruction. update PC by 2.

S19

2) compute the address

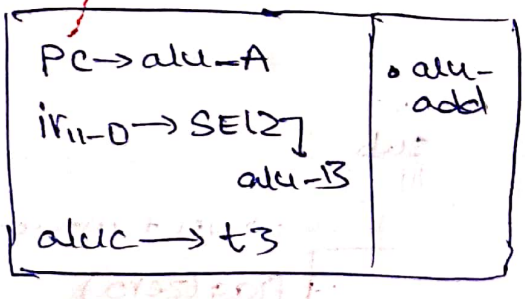
S20

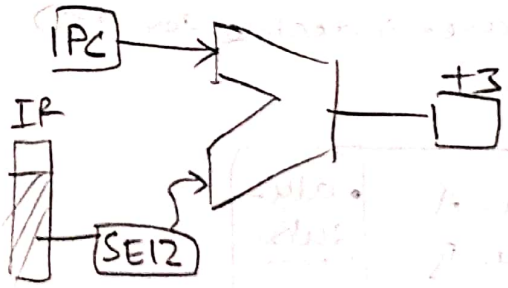
3) Transfer to the address.

S18: Same as Step-1 (PC=PC+2 done here!)

S19:

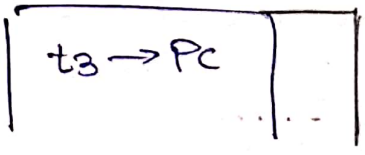
already increased by 2 in S18.





Sign extender 12 to 16 bits.

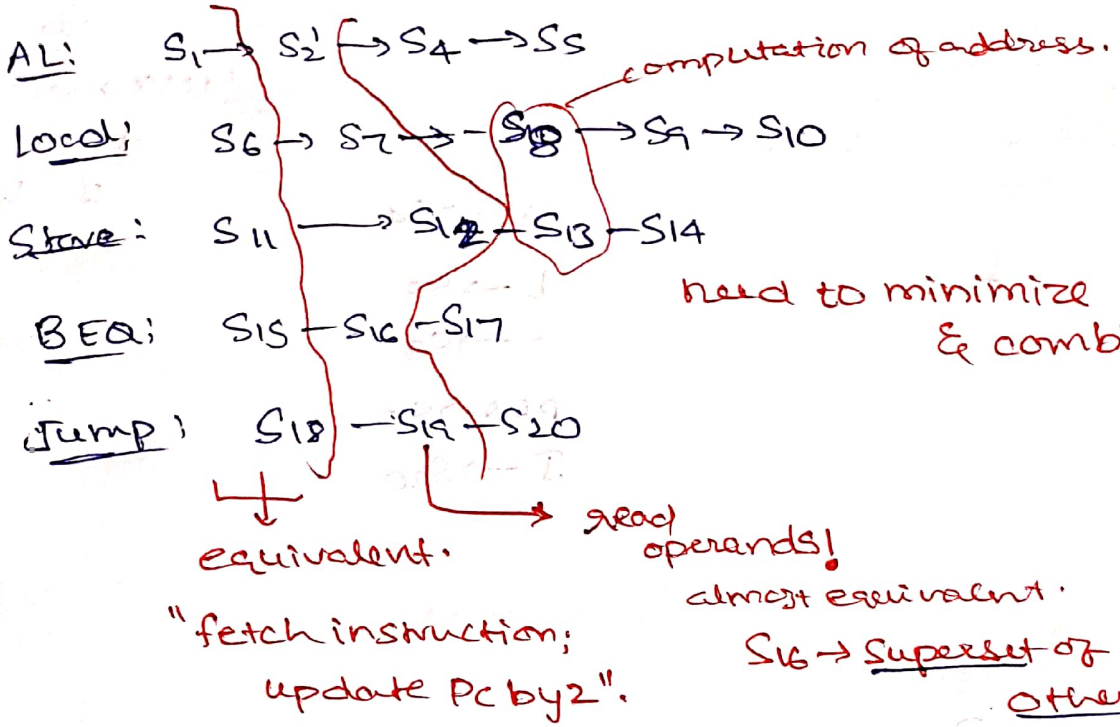
S20: (transfer t3 to PC)



lol.

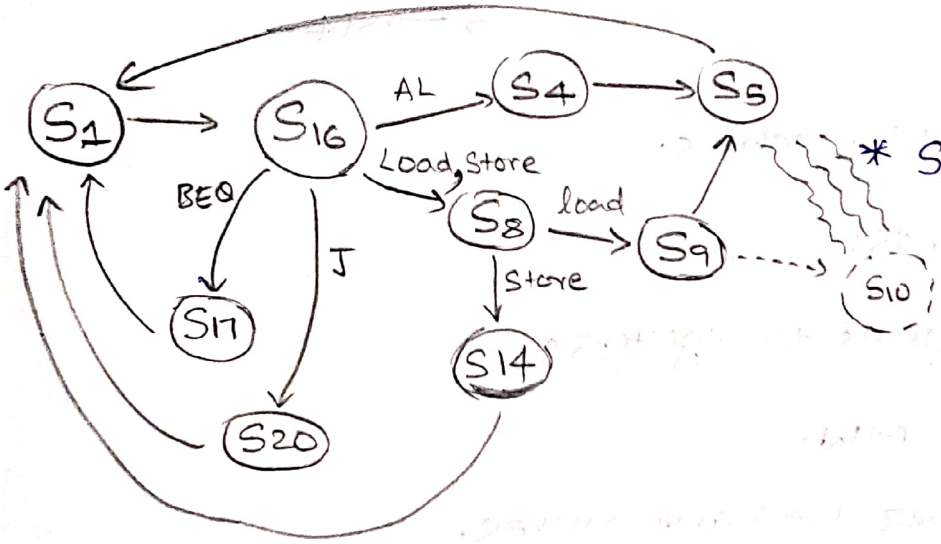
S18 - S19 → S20.

* Now I want to combine all states:-



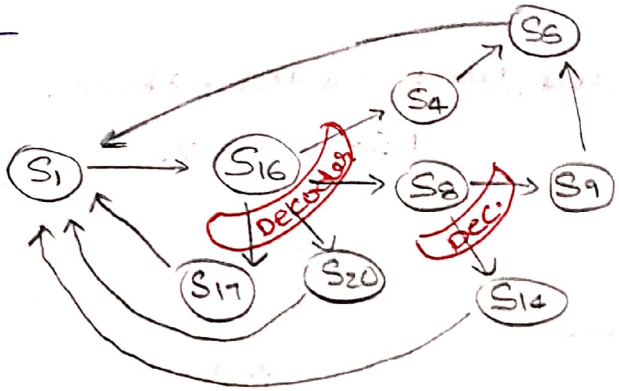
* STG:-

∴ S2, S7 might have redundant operations.



* S5, S10 both write to register equivalence

further:-

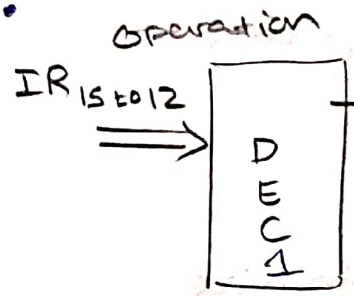


9 states.
4 bits addressing!

PTO

→ decoders:- using decoders, when we don't know where to go.

1.



(like more than 1 child).

why not MUX?

AL → S4

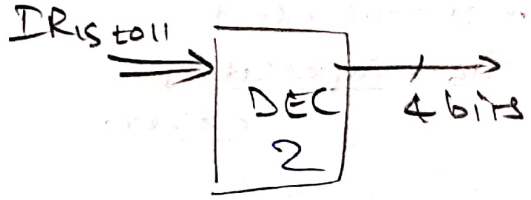
L → S8

S → S8

BEQ → S17

J → S20

2.



L → S9

S → S14.

∴ we made state machine.

→ control points:-

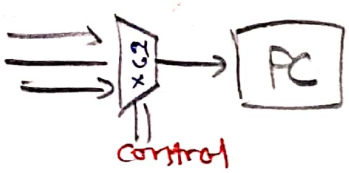
1. wr. signals to registers.
2. MR & MW.
3. wherever we have MUXES,

Eg:

3 inputs from ALU; t3 or else....

can't just twist these three.

use a MUX.



dunno if 3 is correct check!

Eg: for ALU-A; how many different input wires?

PC → ALU-A
t1 → ALU-A
} 2:1 MUX. ✓

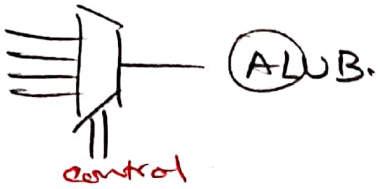
alu-B:

t2 → alu-B

+2 → alu-B 4:1 multiplexer

SE6 → aluB

SE12 → aluB



* like this point out, all control points.

* Now we can fully define state machine.

State machine:-

(1) Next state logic

(2) output → control points