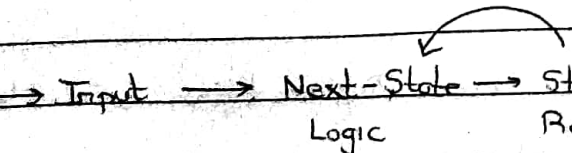
6/9
बुधवार

* A high number of instructions use immediate addressing. Pretty useful
* Base + Displacement :- Useful for iterating over arrays
* Auto increment/decrement :- Push/pop of stack, etc.
* Memory indirect :- Memory contains address of operand

- Begin designing system by defining ISA.
  - Specify no. of registers, etc.
  - Specify no. of instructions, addressing formats, addressing modes
- There is a systematic way of converting your ISA to a real circuit
  = 'Hardware Flow Chart'
  - Start by building logic gates.
  - Your operations should be performable via smaller functional units
    eg- Multiplication by repeated addition
    Subtraction by addition of 2's complement
  - Need more storage which stores partially computed values.
    - Registers, which are accessible to programmer
    - Temp storage, which is invisible to programmer
  - Steering Logic (Transport system)
    - Point-to-point data transfer and Broadcast (bus)
    - Transferring data from ~~register to~~ storage to storage, storage to functional units, functional units to storage (P2P)
    - Only one set of data should be transferred at a time on the bus
  - Controller
    - Controls which instruction is executed when
    - Maintains an FSM for state transitions.
    - No FSM would have been required if all instructions were of the single-cycle kind.



* Mealy machine :- O/p depends on current I/p and current state
Moore — current state only

* Implementing FSMs by hard-coding, as opposed to ≠ Mealy/Moore
- Simpler, no next-state logic required.
- Stores states of every instruction in ROM by computing beforehand.

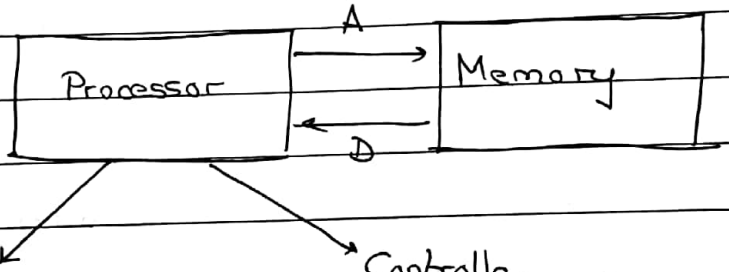
Add	T_1	100100
	T_2	101101
	T_3	100001
Sub	P_1	001000
	P_2	100101

T_i's, P_i's are states.

- Instruction will say which state to move to at given instant

17/9

CISC Architecture



Datapath
→ Functional units
→ Architecture registers
Temporary registers
→ Steering Logic
→ Point-to-point
→ Bus

Controller
↳ FSM
→ Random Logic
↳ ROM based.

• To do :- Convert ISA into Implementation (Gate-level 'netlist')
- Make hardware flow chart.

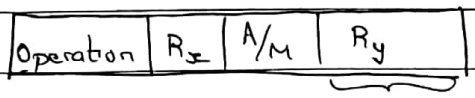
→ Say, you want to design a microprocessor with following specs:-

1. 3 Addressing modes :-

Register Direct, Register Indirect, Based Displacement

AR	AI	AB
00	01	10

2. Instruction format :-



---- 1 'word'

eg :-

```
ADD R1 AR R2        ; R2 = R1 + R2
ADD R1 AI R2        ; [R2] = R1 + [R2]
ADD R1 AB R2 Disp   ; [R2 + Disp] = R1 + [R2 + Disp]
```

- R_x is always directly addressible.
- If you are using B+D, the displacement will be given after the instruction in the next word.

- Steps to follow while execution :- (examples 1 & 2 :- AR, AI)

1. Fetch instruction
2. Understand instruction, find locations of control signals in ROM.
3. Add
4. Write result in memory.

- Steps 1, 2, 4 are generic. Step 3 is particular to the instruction.

- Steps for B+D instructions -

1. Fetch instruction
2. Understand instruction
3. Fetch second word (displacement)
4. Compute operand 2 :- R_y + disp
5. Fetch second operand
6. Add
7. Write result in memory

Only ADD, SUB, NAND, LOAD can modify zero flag

- The steps written above are states of the controller FSM
- Steps 1 and 2 are generic in all 3 examples.
  - Two ways to proceed
    1) Execute in the order written

       OR

    2) Execute the generic steps at the end of the loop
       - The first instruction is fetched and understood by Power ON sequence.
- Instructions we want to implement:- (ISA)

ADD, SUB, NAND, TEST, LOAD, STORE, PUSH, POP, BZ

3 addressing modes as discussed

LOAD: Load to R_x from memory indirectly addressed by R_y — $R_x \leftarrow [R_y]$

STORE: $[R_y] \leftarrow R_x$

POP: $R_x \leftarrow [R_y]$, $R_y = R_y + 1$

PUSH: Build stack downwards — $R_y = R_y - 1$, $[R_y] \leftarrow R_x$

TEST: Sets the zero flag if $R_x = 0$

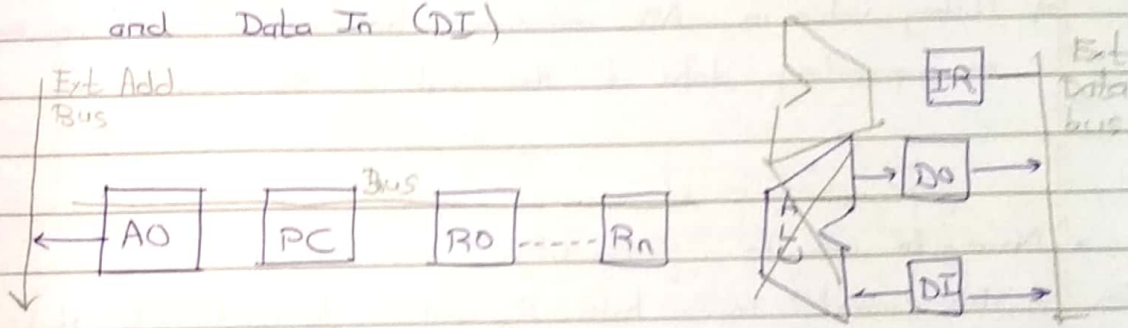
BZ: If the result of previous instruction is zero, jump to a particular location. Whether previous result is zero is specified as a flag

We will build a hardware flow chart to implement this.

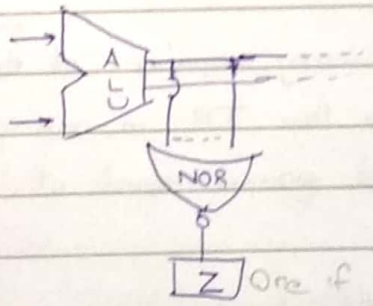
- Building the datapath:-
  - Functional units :- ALU (Add, subtract, ~~dc~~ NAND) is sufficient.
  - Registers :- R0 to Rn for architecture registers
    :- One register to act as instruction pointer (program counter)

- As an engineer, start with minimum no. of registers required Add later if desired.

- Steering Logic :- Broadcast Bus.
  - To communicate with external world, we need an external address bus and external data bus.
  - The external address bus is interfaced via a buffer register known as 'Address out (AO)'
    Similarly, external data bus is connected via Data out (DO) and Data In (DI)

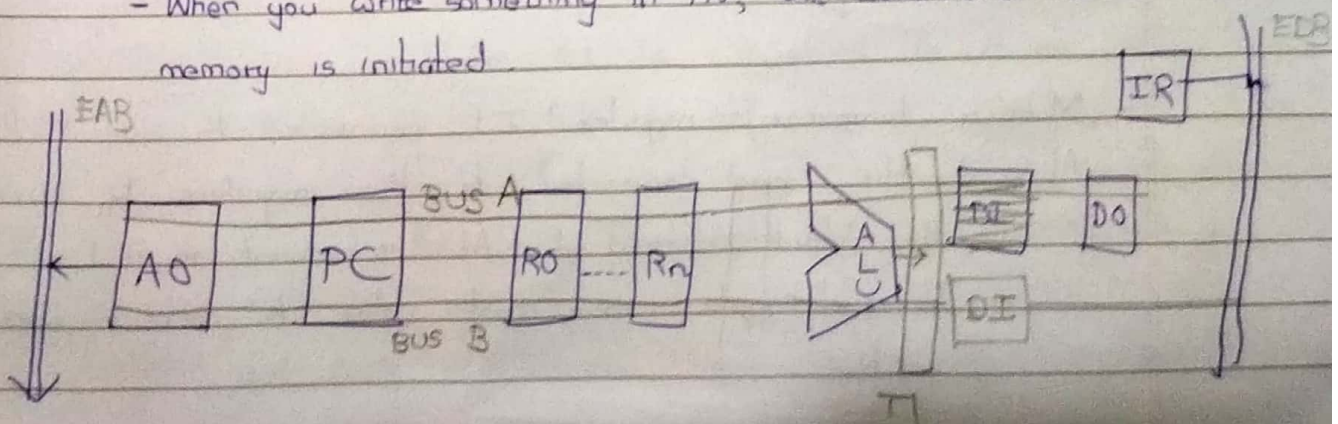


* Zero flag implemented at output of ALU



18/9

- When you write something in AO, the transmission to external memory is initiated.



- Instruction Register (IR) - Holds the instruction currently being executed.

- Manufacturing comes with restrictions.

- Rules of operation as a result of restrictions :-
  1) In one cycle, bus can be used once (for transport from one source to destination only - eg - R1 to ALU)
  2) Putting value in A0 initializes transmission, and we receive corresponding data at the end of that cycle.

- Now, to execute $R_x + R_y$

☹ - We cannot move both R_x and R_y to inputs of ALU within one cycle. We cannot move R_x & R_y one after another because ALU inputs are not memorized. ALU cannot hold values. It is combinational logic. Do what you want within that clock cycle.

Meth 1 - Make a temporary register at one input of ALU.
Move R_x to this TR in one cycle.
Move R_y to second input of ALU in second cycle.
Add.

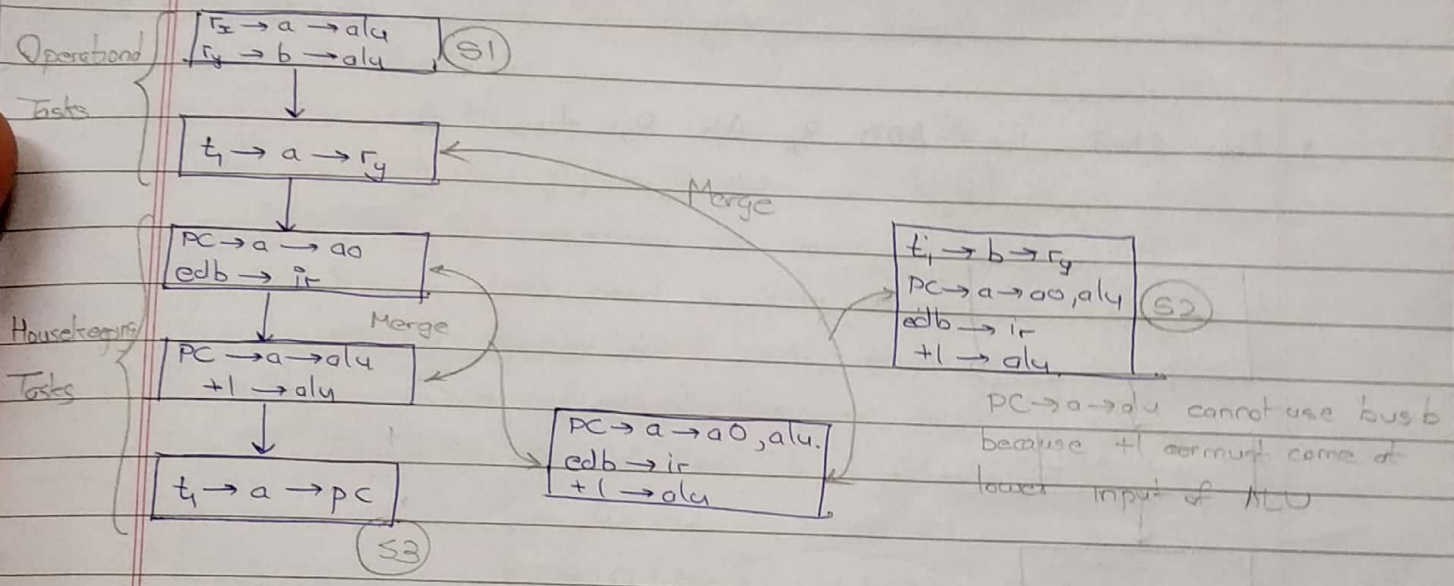
Meth 2 - Design another bus

☺ Move R_x and R_y simultaneously on different buses in one cycle

- Make a temporary register T1 connected to output of ALU directly, and connected to other registers by buses.
  - By default, output of ALU is stored in T1
  - In next cycle, move that value to correct destination via bus

- In order to process addition with constants (-1, 0, 1) for INC or DEC, we have a new register at input of ALU that can take one of the above three

Step 1 • Flow chart for addition instruction
One block = 1 state (execution takes 1 cycle)



Step 2 • Write operational and house-keeping tasks separately. Merge them as well as you can
• One block contains concurrent tasks, to be executed parallelly

Step 3 • Give extra information about operation (flag updates)
- ALU operation in state S1 is supposed to update zero flag. S2 is not

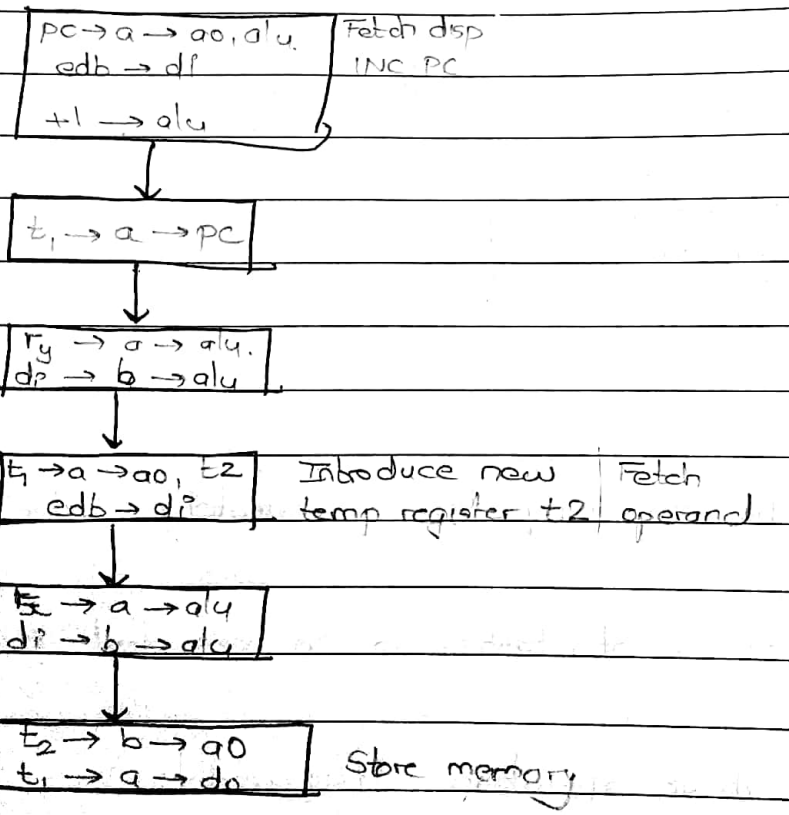
Step 4 • Now, if you find that your flow chart has identical states, combine them
eg - ADD and SUB instructions have same S2, S3. Only S1 is different

- Overall Steps :-

Level 1	→ Merged Level 1	→ Level 2	→ Merged Level 2
Flow Chart	FC	FC	FC
Write operations & house keeping separately	Merge as many states as possible	Encode flag updates	Combine identical blocks across instructions (like ADD, SUB, ANL)

- Flow Chart for 'ADD R_x AB R_y disp'

$[R_y + disp] \leftarrow R_x + [R_y + disp]$



- If your µP supports 'm' type of instructions (≈100), 'n' Addressing modes (≈10), 'p' states per instruction per addressing mode on average, (≈10)

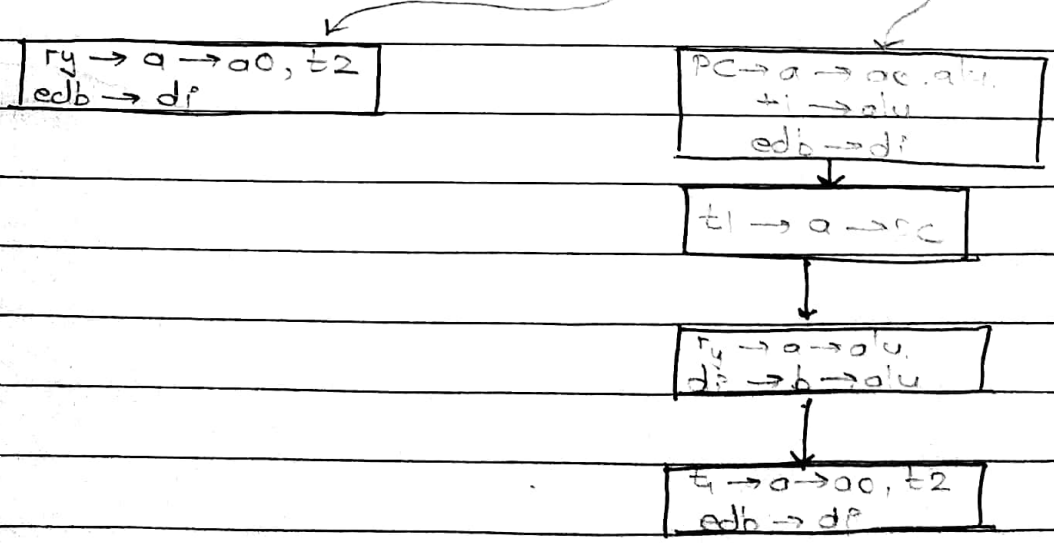
Total no. of states = $m n p$ --- Very high :-

- Assume instructions and addressing modes to be orthogonal
  - Any A/M can be used for any instruction.

- Operational Tasks
  - Addressing mode sequence (Fetching the operand(s))
  - Execution sequence (Execute)

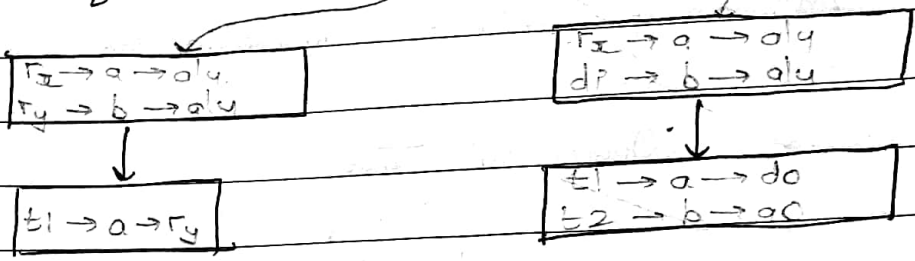
- We want to make both sequences distinct.
  - Standardize :-
    1) Always place operand in di
    2) Always place address of destination in t2.
  - ✡ Standardization will make O(m+n) instead of O(mn)
  - But we don't need to waste one cycle moving operand to di in Register direct mode, as registers are directly accessible
    - Do not standardize for register direct.
    - ∴ We will have to write separate execution sequences (one for register direct and one otherwise)
    - ✡ ~ O(2m+n)

eg :- Addressing mode sequence for AI and AB



Register direct does not have addressing mode test

eg Execution sequence for AR and AI/AB

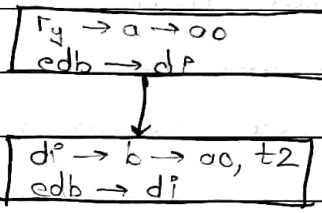


- Now that you have separate addressing mode sequence and execution sequence, which of those will you preferentially use to merge with housekeeping tasks?

Ans - Execution sequence, because ~~all~~ register direct addressing mode does not have addressing mode sequence.

eg Addressing mode sequence for AM:- Memory indirect (double pointer)

$$[[R_y]] \leftarrow R_x + [[R_y]]$$



HW Write for other instructions from slide

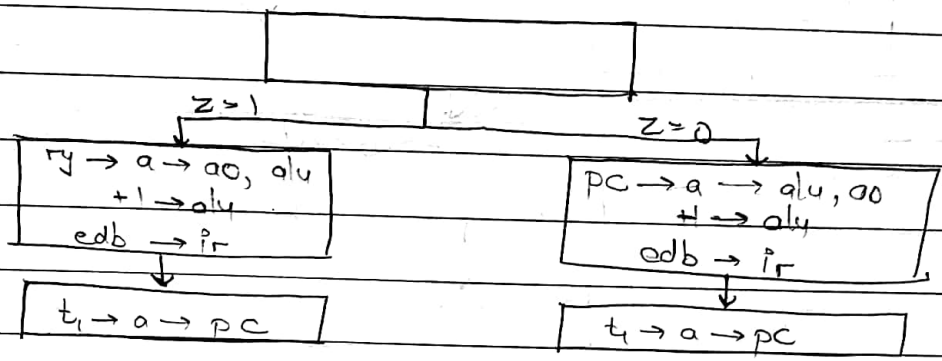
HW Study level 2 flow charts with zero flag updation augmented

All registers are n-bit

assuming R_y & PC have same no. of bits

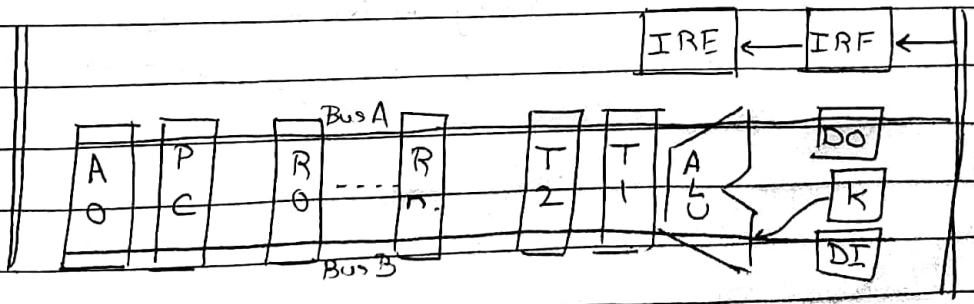
- BZ $[R_y]$:- IF Z (flag) = 1, then PC ← R_y

  else PC = PC + 1



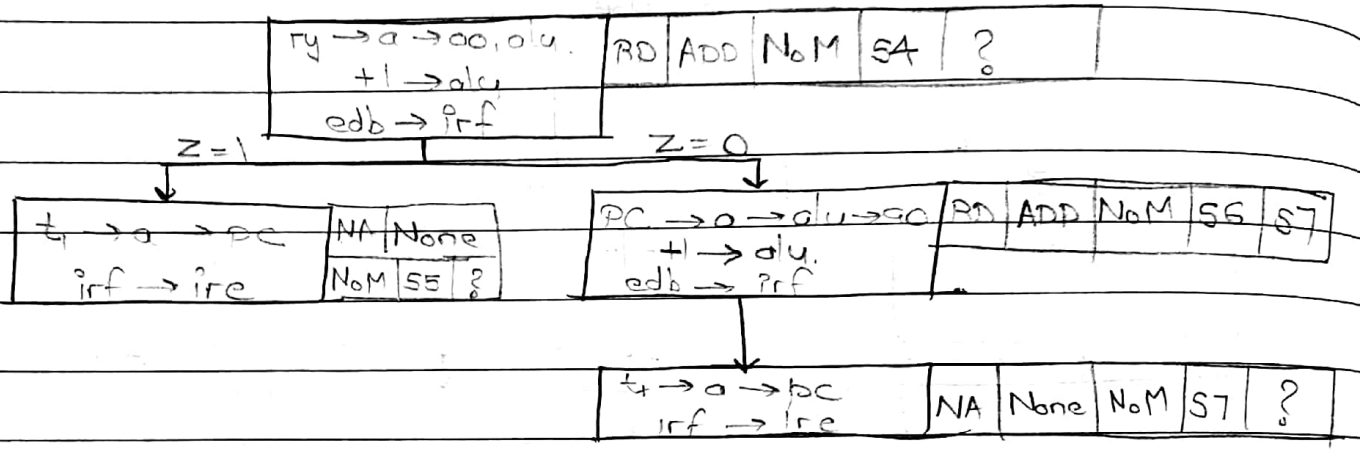
- This is execution sequence. There is no addressing mode sequence.
- BZ just chooses which instruction to execute next. No 'useful work' is done.

- Restriction on merging housekeeping tasks with execution tasks :-
  IR register should be updated only after previous instruction is completely executed.
  - Introduce IRF (Instruction register for fetching) and IRE (Instruction register for execution) in place of IR
    - Move IRF to IRE when execution of previous instruction is complete.



P.T.O.

• BZ (Modified) Level 2 flowchart.



– This is better because it has one state less.

– Observe :- ~70% of conditional jumps are actually taken. That is why, we moved Z = 1 state above and shortened time for Z = 1 branch preferentially.

⟶ Level 2 Flowchart.

Encode information about every state.

1 – Memory Access :- No access / Read / Write

2 – ALU operation :- OP / Add → addition; operation as defined by instruction

3 – Zero flag updation :- Modify / No modification

4 – Unique ID (of state)

5 – Next state ID

eg ADD R_x AR R_y

$r_x \rightarrow a \rightarrow alu.$ $ry \rightarrow b \rightarrow alu.$	NA, OP, Mod, S1, S2					
$t_1 \rightarrow b \rightarrow ry$ $pc \rightarrow a \rightarrow ao, alu.$ $t_1 \rightarrow alu$ $edb \rightarrow irf$	RD ↑ Read	ADD	No-mod	S2	S3	
$t_1 \rightarrow a \rightarrow pc$ $irf \rightarrow ire.$	NA	None	No-mod	S3	?	

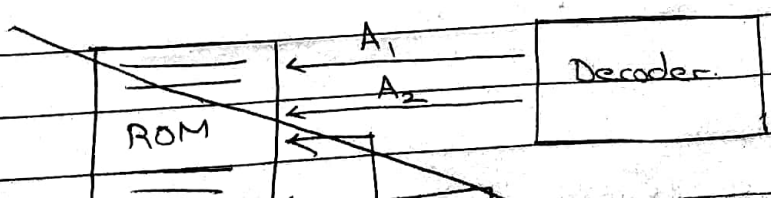
- See modified BZ for second example.

→ Decoder

Decoder of instruction must generate two addresses – one for first state of addressing mode sequence and one for first state of execution sequence.

- Assume, for every instruction, all states have consecutive unique IDs (S_i)

Better – Many states, like last instruction in both the above examples (S_3, S_5, S_7) are identical and redundant.
Merge all such 'equivalent states'
Now, for every instruction, states will not have consecutive IDs.



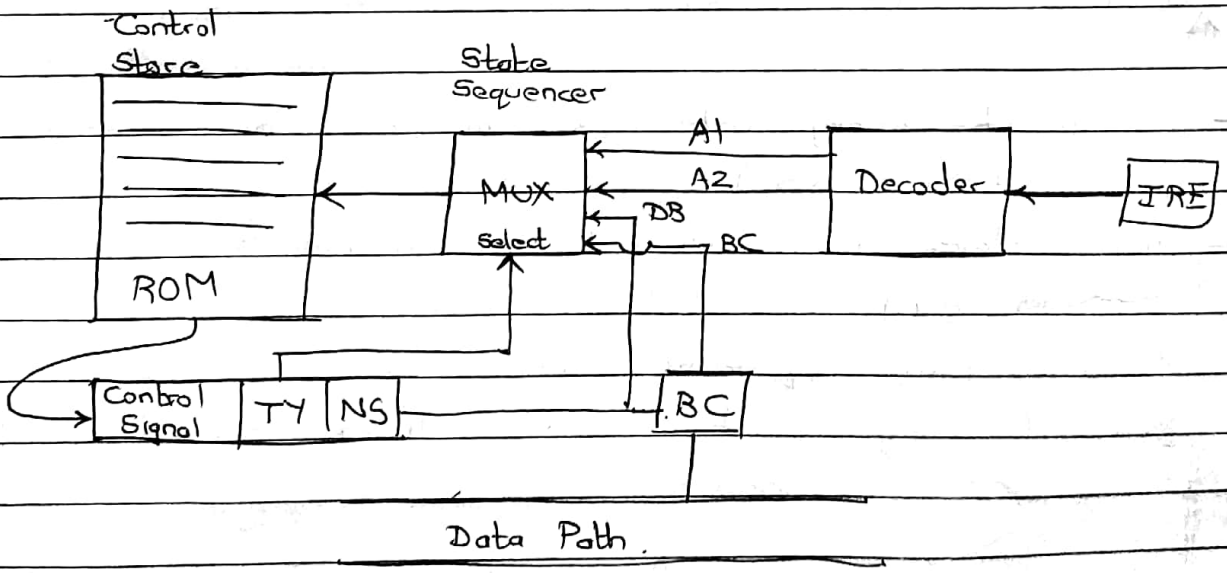
- Decoder gives :-
  - A1 :- ID of first state of addressing mode sequence
  - A2 :- execution

- Possible values 'Next state' of each state can take :-
  IB SB DB BC.

- IB = Instruction Branch
  at last state of execution sequence
  Current instruction is completed, Now fetch first state of next instruction (A1 of next instruction)

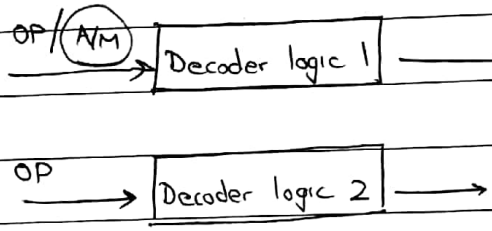
- SB = Sequence Branch
  'Next state' of last state of addressing mode sequence.
  Fetch the next state by A2 -

*



- Every state has its own (Control Signal, TY, NS) encoded.
  - TY :- 2-bits to choose from IB, SB, DB, BC.
  - NS :- ID of next state if we use DB or BC

• IF instruction does not have addressing mode sequence (register direct), A1 contains ID of execution sequence A2 is 'don't care'. IB will directly lead to execution sequence



These two logics can be combinationally

- DB = Direct Branch
Simply move to consecutively next * state. The address (ID) of next consecutive state ~~st~~ is stored in NS.

- BC

→ Control Signal :-

Parts for A0 PC Reg T2 T1 ALU D0 DT

• Consider part for PC
- Through "all" the states, what all jobs does PC do?
  - NONE, pc → a, a → pc, b → pc
- Since there are 4 jobs, part for PC is described in 2 bits.

Observe :- a → pc is used only once (addressing mode seq of B+D)

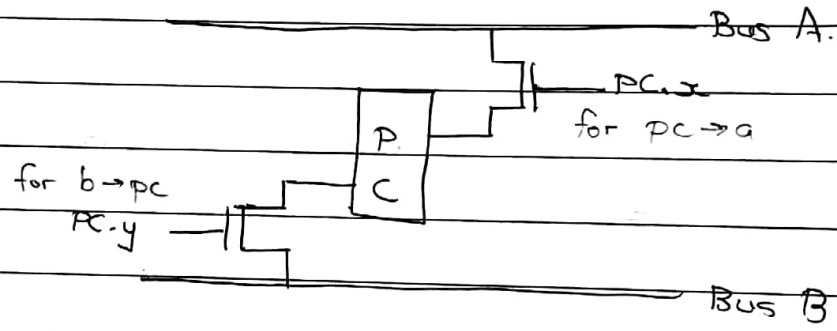
P.T.O.

Even in that state, bus b was lukkha.
Do not use $a \rightarrow pc$. Use $b \rightarrow pc$ only.

- If part for PC is xy

y \ x	0	1
0	NONE	$pc \rightarrow a$
1	$b \rightarrow pc$	X

- Since you have only 3 possible functions, use the (1,1) combination to simplify the decoder



- Consider T2

- All possible functions :-

  NONE, $t_2 \rightarrow a$, $t_2 \rightarrow b$, $a \rightarrow t_2$, $b \rightarrow t_2$

- $a \rightarrow t_2$ is used only once, when bus b is free.
  Use $b \rightarrow t_2$ instead
  - ∴ 2 bits of encoding are sufficient

-

y \ x	0	1
0	NONE	$t_2 \rightarrow a$ → $x\bar{y}$
1	$t_2 \rightarrow b$ → $\bar{x}y$	$b \rightarrow t_2$ → xy