- Performance :- Arithmetic / Logical :- 50%
  Load :- 20%
  Store :- 10%
  ~~Branch~~ BEq :- 15
  Jump :- 5%

  For multicycle, Performance = N × ~~12.45~~ 4.15 × 3ns
  = 12.45
  single cycle = 12.5

- Depending on fractions of instruction types, either of single cycle or multicycle implementation.
- In spite of fewer registers, single-cycle is more expensive overall in terms of hardware.
- Single cycle :- Lowest possible CPI (1)
  Multi cycle — cycle time
  - Possible to strike a balance :- eg- 2 cycle implementation.

→ Pipelining.

- Begin next instruction before previous is over, using the idle resources
- In RISC, subtasks in every instruction are very similar - fetch instruction, fetch operands, alu operation, etc.

- It is possible to get least possible cycle time (like multicycle)
  - If we assume all instructions to be independent, then we can do pipelining and then we will get cycle time equal to max{sub-task time}

Mealy :- Output logic & Next-state logic depend on both input and current state

Moore :- NSL is same as before
OL depends only on current state

→ Subtasks

1. R-type :- Instruction fetch (IF), Instruction Decode/Operand Read (ID/OR), Execute (EX), Write Back (WB)
2. LOAD :- IF, ID/OR, EX, Memory Read (Mem), WB
3. Store :- IF, ID/OR, EX, Mem
4. BEQ :- IF, ID/OR, EX
5. JUMP :- IF, ID, EX

exhaustive set of subtasks

Cycle 1 2 3 4 5

- First 4 cycles, no instruction will be completed
After that, one instruction will be completed every cycle.
(for independent instructions and if instructions are consecutive from top to bottom (no control flow changes in between))

→ Performance = N × 1 × 3ns
CPI (CPI ≈ 1) Cycle time. Cycle time = max(subtask time)

- Every instruction will go through all 5 stages
IF → ID/OR → EX → Mem → WB
eg - JUMP will waste time in Mem & WB stages

- Wasting time :- Do the job in each state, but don't write to any register (using control signals on registers)

multi cyple is example of mealy and pipelining is example of moore as in pipelining no matter what the instruction is (ie input) it will go through all the stages

- Making 5 stages common for all instructions converts each stage from a Mealy machine (which depended on instruction) to a Moore machine (which does not)
Moore :- Control signals generated depended only on current state
Mealy

- To use pipelining, all instructions must use similar type of stages, and in same order too.

	IF	ID/OR	EX	Mem	WB
Cycle 1	I_1				
2	I_2	I_1			
3	I_3	I_2	I_1		
4	I_4	I_3	I_2	I_1	
5	I_5	I_4	I_3	I_2	I_1

Latency = '5' not 4

→ How do you design a pipeline (determine the required subtasks)?

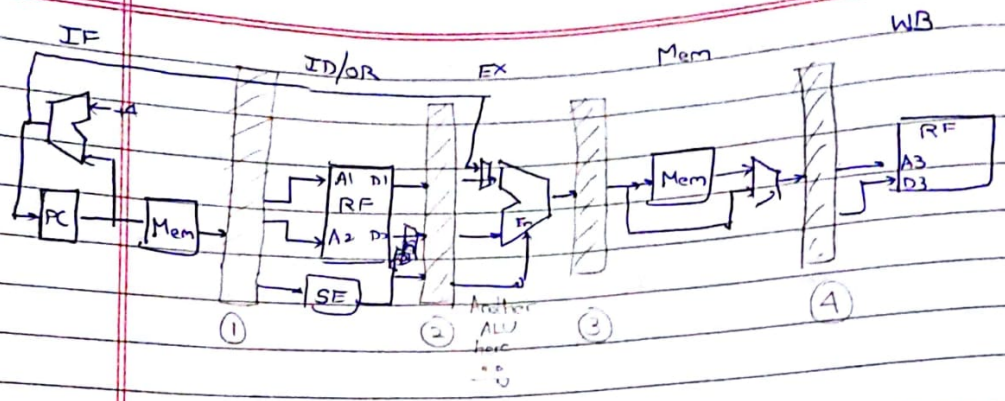
∴ - Start from multicycle :- We will have minimal resources used.
Could be an issue if resource is shared by subtasks

eg- In multicycle, same memory is used by IF & Mem

- ∴ Start with single cycle to get maximal resources.

→ - We generate control signals through combinational decoder of single cycle implementation (like always), and "remember" them in a register
The control signals for each instruction must be propagated consecutively to each task along with the instruction itself.
The respective subtask uses whichever control signals it needs.
- So every subtask will have a register (to store control signals) of the size equal to number of control signals flowing from previous state to that state
~~Register in subtask 1 will have size equal to total number of control signals.~~
- ~~PC is also propagated through the stages, as it will be required for returning control flow changes~~

Instructions move in lock step manner (= FIFO, no jumping)



- Signals crossing interface after IF :-
  Instruction (32 bit), PC+4 (32 bit), etc

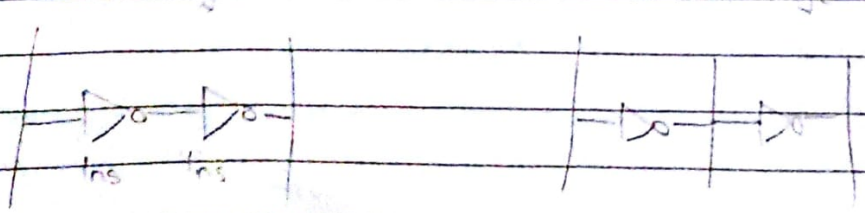
- Interface 2 :-
  PC+4 (32-bit), Operand 1 (32-bit), Operand 2 (32), Sign-extended Imm (32), ALU control signals, destination register ~~reg~~ address etc

- Interface 3 :-
  ALU output (32), A3 address etc

- Interface 4 :-
  Destination register address, data to be written, etc

→ How many stages to divide the instruction into?

- More no. of (smaller) stages will give lower cycle time ☺
  - Becomes costlier, as you need more registers for interfaces ☹
  - Performance ↑, CPI still ≈ 1



Cycle time = 2ns

Cycle time = 1ns

- Choose the value of no. of stages (n) that maximize $\frac{\text{Performance}}{\text{Cost}}$

Cost $= G + (n-1)k$ (k → avg cost of interface registers)

Performance $= \frac{1}{\frac{T}{n} + q}$ (T → time without pipelining; q ← some small delay inherent)

G → cost without pipelining

* $\frac{\partial}{\partial n} \frac{1}{\left(\frac{T}{n} + q\right)(G + (n-1)k)} = 0$

22/10

→ Our assumptions :- (Hazards)

eg Dependent instructions :- I1 : $r_1 = r_2 + r_3$

I2 : $r_4 = r_1 + r_5$

In ID/OR stage, I2 will read incorrect r_1

- Detecting violation of independence of consecutive instructions :-

Source registers in ID/OR stage instruction = Destination register of Ex stage instruction.

i.e → $(RS1^{ID/OR} == RD^{Ex}) \;||\; (RS2^{ID/OR} == RD^{Ex})$

- If this violation occurs, 'freeze' I2 in the ID/OR stage for 3 cycles (until I1 completes). Then resume.
  - On every violation, we are losing 3 cycles
  - eg - If 10% instructions have this 'immediate' dependency.

New CPI = $1 + 3 \times 10\% = 1.3$

I1	—	—	—	—
I2	I1	—	—	—
I3	I2	I1	—	— freeze
I3	I2	—	I1	—
I3	I2	—	—	I1

- The result of I_1 is getting written back in stage 5 (WB). But it is being produced in stage 3 (EX) itself. This result will be stored in the pipelining interface register at the boundary of EX and Mem. Let I2 fetch the result from there.

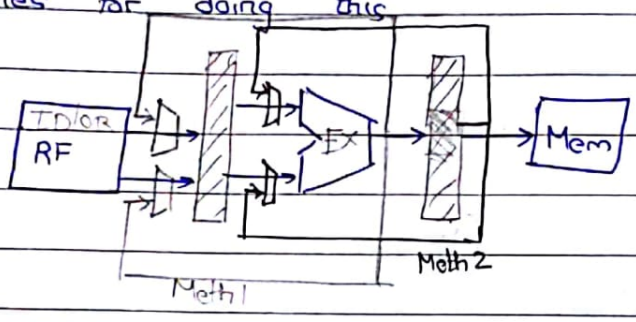
I_1 — — — —

I_2 I_1 — — —

I_3 I_2 I_1 — —

I_4 I_3 I_2 I_1 —

Two possibilities for doing this



→ Non-consecutive dependency.

I_1 :- $r_1 = r_2 + r_3$

I_2 :- $r_4 = r_5 + r_6$

I_3 :- $r_7 = r_1 + r_8$

1. (RS1 or RS2)^{ID} == RD^{Mem} :- Where to fetch operand from?

2. (RS1 or RS2)^{ID} == RD^{WB} :- Obtain result from rf.d3 of WB stage

→ Multiple write conflicts.

$r_1 = r_2 + r_3$

$r_1 = r_4 + r_5$

$r_6 = r_1 + r_7$

• Priority order :- For conflicting dependencies, fetching result :-

Ex > Mem > WB

(Ex: newest instruction; WB: oldest instruction)

→ Load instruction produces result in Mem stage.

I1 :- $r_1 \leftarrow (R2 + 50)$

I2 :- $r_3 = r_1 + r_2$

When I2 is in ID/OR, the required result has not been produced yet. Forwarding will not work. We have to wait (freeze) for one cycle at least in such cases.

1	–	–	–	–	
2	1	–	–	–	
3	2	1	–	–	~~freeze~~
4	3	2	1	–	freeze ---- 'Bubble' or 'Stall'
4	3	2	–	1	resume
5	4		3	2	–

23/10 – We are losing one cycle for every instruction of this type.

eg – α% AL instructions, β% Load (of which λ% have such immediate hazard), γ% Store, δ% Branch.

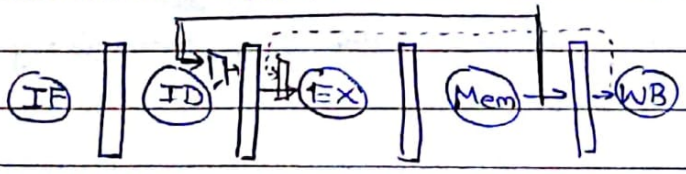
New CPI = $1 + \beta\lambda(1)$

• Bubble or Stall :-

All interface registers should have an enable signal, which is made zero to stop updation.

Make enable of PC zero, too.

• For hazardous Load.



→ Minimizing the damage of Load hazard :-

eg $A = B + C + D$ → load r_1, B

$T1 = B + C$

$A = T1 + D$

load r_2, C

add r_3, r_2, r_1

store r_3, t_1

load r_4, D

add r_5, r_4, r_3

store r_5, A

A, B, C, D, T1 are in memory

Smart compiler →

load r_1, B

load r_2, C

load r_4, D

add r_3, r_2, r_1

store r_3, t_1

add r_4, D

store r_5, A

'Compiler optimization'

→ Change in Control Flow

eg I_1 contains JUMP.

∴ I_2 is not supposed to be executed.

IF ID/OR

a) 1 — — — —

b) 2 1 — — —

- Every interface register should have an extra bit, which will decide if an instruction is to be disabled.

If I_1 is JUMP, then the disabled I_2 will go through all stages, but will not write to any registers, flags

Meth 2 Delayed Branch Execution

compiler

X After every branching/jumping instruction, automatically adds a NOP instruction.

P.T.O

If I2 is JUMP,

- If I2 does not depend on I1 (unconditional jump), move I1 to just after I2.
- If I2 depends on I1 (conditional jump), add a NOP instruction to just after I2.

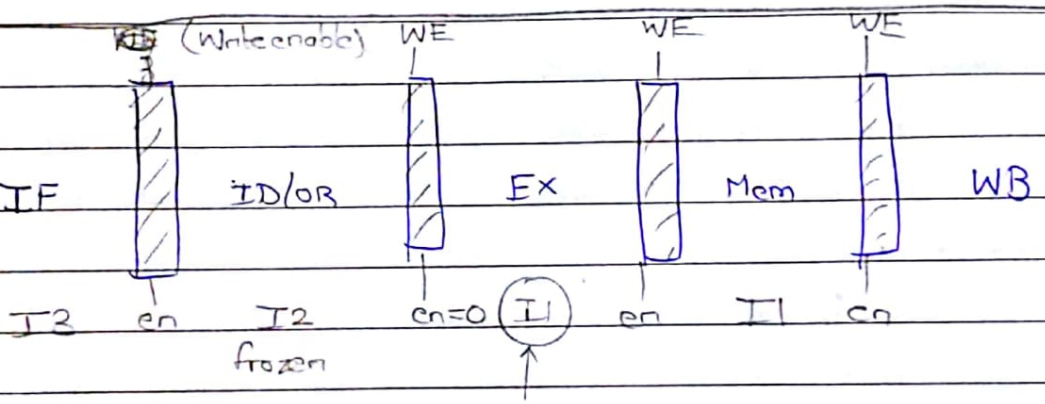
→ Conditional branching

eg I1 is BEQ

1 _ _ _ _

2 1 _ _ _

3 2 1 _ _



Comes here because interface register 2 was not updated in previous cycle.
This extra instruction needs to be killed.

- Instead of using $(R_{S1}^{ID} == R_D^{Ex}) \,||\, (R_{S1}^{ID} == R_D^{Mem}) \,||\, (R_{S1}^{ID} == R_D^{WB})$, we now need more checks because I1 is duplicated.

Use: $\left(R_{S1}^{ID} == R_D^{Ex} \;\&\&\; R_{WE}^{Ex} == 1\right) \,||\, \left(R_{S1}^{ID} == R_D^{Mem} \;\&\&\; R_{WE}^{Mem}\right) \,||\, (\ldots\ldots)$

&& it should be LOAD instruction

&& MEM_AD == 0

Only LOAD instruction reads from data memory.

Stall Requirement detection

If this condition is met, $IF_{EN} = ID/OR_{EN} = PC_{EN} = RF_{WR} = 0 = Flag_{WR}$

$Mem\_WR = 1$

→ Unconditional JUMP.

Let IF interface b register have a bit 'valid'
If I1 is found to be jump, make 'valid' of I2 from IF (in IF) 0'.
If valid = 0, don't let that instruction write to flags, registers, memory. (make WE of subsequent interface registers 0)

★ Assume that for JUMP instruction, we can update PC as soon as ID stage (using separate ALU)

→ Conditional JUMP (BEQ) (I_1)

- Assume branch is not taken. Bring in I2, I3 serially. When I_1 reaches EX stage, it decides whether branch is taken or not.
- If branch is to be taken, don't let I2, I3 update flags/registers/memory (assumption is invalidated)
  - Penalty of 2 cycles ∴
  * Most b of the times, branches are taken

- Assume that if branch is to be taken, PC is updated at the end of EX. stage.

• Our initial assumption cannot be to take the branch, as we don't know yet where to branch to.

• $\alpha \rightarrow$ AL, $\beta \rightarrow$ Load, $\gamma \rightarrow$ Store, $\delta \rightarrow$ Branch (Conditional), $\epsilon \rightarrow$ Jump
  ↓ λ% independency ↓ 70% branches are taken ↓

$CPI = 1 + \beta\lambda(1) + \epsilon(1) + \delta(0.7)(2)$
★ Why

If branch is taken,

I1	—	—	—	—
I2	I1	—	—	—
3	2	1	—	—
20	—	—	1	—
21	20	—	—	1

- Optimization for when branch is taken:-

For first time of execution, ~~compute~~ proceed as before.

Compute destination of branch and store it in Look-up table.

PC	Branch target Address
100	700
150	755
⋮	⋮

Such instructions are mostly executed many times (in a loop)

Second time onwards, change assumption to branch taken

1	—	—	—	—
2	(1) ✓ instruction decoded	—	—	—
branch taken → 20	2	1	—	—
21	20	2	1	—

One cycle penalty still

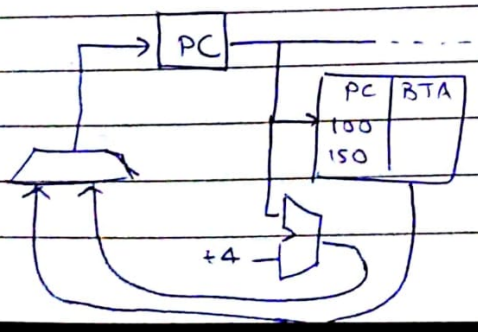
Now assumption is correct 70% of the times.

☆ Verify — New CPI = $1 + BA(1) + E(1) + S(0.3)(\frac{1}{2})$

- Better still (for no penalty if branch is taken) detection in IF stage of I1

Match the lookup table with new value of PC.

If current PC is found to be present in table, fetch I20 in next consecutive cycle.



1	—	—	—	—
20	1	—	—	—
⋮				

☆ Verify — CPI = $1 + BA(1) + E(1)$

→ If branch is not taken (2 cycle penalty)

- We have lost original PC. Where to go ~~back~~?
  - We need to carry forward original PC+4 (~~of I1~~) forward
    Anyways we were carrying it forward till EX to compute PC + 4 + Imm

→ Further optimization :-

- Branch instructions tend to show similar behaviour as the last time they were executed.
- In the back-up table, add a bit to say if last time branch was taken or not.
  Change your assumption accordingly.

PC	BTA	History (0 → not taken)

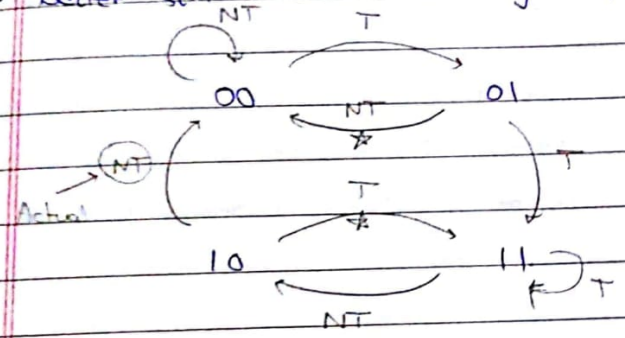
In execution stage of I1, update history bit every time.

Prediction	Actual	
0	1	..... Entry into loop
1	1	8 times
1	1	
⋮	⋮	
1	0	..... Exit from loop
0	1	

We were right 80% of time. (no penalty)

classmate
Date
Page

- Better still :- 2 bit history (past 2 decisions)



$H_{t-1}H_{t-2}$

Prediction

00 → Strongly NT
01 → Weakly NT
10 → Weakly T
11 → Strongly T

Now you will not make mistake for entry into loop (90% correct)

29/10

MEMORY SYSTEM

- For modern memory (DRAM), one memory access takes 150 - 200 cycles (not 1 as before)
  - Not counting multiple memory accesses (just one, to fetch ir), each instruction will take ~200 cycles
  - Pipelining makes no sense, as a fetched instruction will take only #stages cycles more to complete

- One possible solution:- 'Cache'
Between processor and memory, add a very small, but fast block of memory. This has, say, only 8 bytes (worth 2 instructions) and is made from more expensive SRAM. SRAM memory is accessible in single cycle. (must fetch consecutively written instructions)
  - Fetch 8 bytes (2 instructions) from memory into SRAM (~200 cycles)

Processor fetches I_1 (200st cycle)
I_2 (201nd cycle) ... pipelining
I_3 (400rd cycle)
I_4 (401th cycle) ... pipelining.

  - Performance almost doubles by introducing SRAM, as at least 2 instructions are being pipelined

30/10 →

eg Eg Program

```
for i in range(n):
    I1
    I2
    I3
    I4
```

Assume instruction do not access memory

If SRAM is 16 bytes, 4 instructions will be fetched in 201 cycles and stored in SRAM
For every subsequent iteration, we don't need to fetch again and instructions can be pipelined as before

CPI → 1 ☺

eg for i in range(100):
    if i%2==0: I_1 I_2 I_3 I_4
    else I_5 I_6 I_7 I_2

- With 16 bytes of SRAM, we will have to fetch and replace the four instructions again and again.
Total no. of cycles = $(200+1+1+1) \times 100$ ☹

- With 32 bytes of SRAM, we can get CPI → 1, but only if all 8 instructions are consecutively written. Otherwise same as before.
$(200+1+1+1$ ~~$+1+1+1$~~$) + 50 + (1+1+1+1$ ~~$+1+1+1+1$~~$) \times 49$

- With 2 SRAM chunks of 16 bytes each, we can get CPI → 1
The instructions in each individual chunk must be consecutive.
$(200+1+1+1 + 200+1+1+1) + (1+1+1+1+$ ~~$1+1+1+1$~~$) \times 48$

• The SRAM chunks are hidden from programmer and known as 'Cache'.

10/10

→ Bandwidth of Memory:-

$$\frac{\text{Data}}{\text{sec}} = \frac{\text{Data}}{\text{Instruction}} \times \frac{\text{Instruction}}{\text{Cycle}} \times \frac{\text{Cycle}}{\text{Sec}}$$

$$= \left(\frac{\#\text{Bytes}}{\text{Instruction}} + \frac{\#\text{Bytes of data}}{\text{Instruction}}\right) \times \text{IPC} \times \text{freq}$$

$$= (4B + 4B) \times 1 \times 3 \times 10^9$$

(1 → max IPC = 1)

$\approx$ 24 GBps

This is for processor demand from memory (each instruction requires 4B data to be fetched too)

→ Final CPI depends on no. of 'stalls' required by the program
- Instructions may stall, for reading from or writing to memory, for 200 cycles
- Stalling also happens to fetch instructions not present in cache

• Instructions can be 'hit' or 'miss'
  - present in cache: No penalty
  - not present: 200 cycle penalty

• No. of stalls = No. of misses × penalty

$$= \#\text{Instruction} \times \frac{\#\text{miss}}{\text{Instruction}} \times \text{penalty}$$

$$= \#\text{Inst} \times \frac{\#\text{Memory references}}{\text{Inst}} \times \frac{\#\text{miss}}{\#\text{Mem ref}} \times \text{penalty}$$

$$= \text{Inst Count (IC)} \times \frac{\#\text{Mem ref}}{\text{Inst}} \times \underbrace{\text{Miss rate} \times \text{penalty}}_{\text{In the hands of memory designer}}$$

→ Miss Rate reduction

1. Increase no. of blocks in SRAM.

2. Replacing contents of cache :-
Preferentially, do not remove instructions that are consecutively likely to follow current instruction (spatial likeliness) or that are likely to be executed again at some later point in time (temporal likeliness)

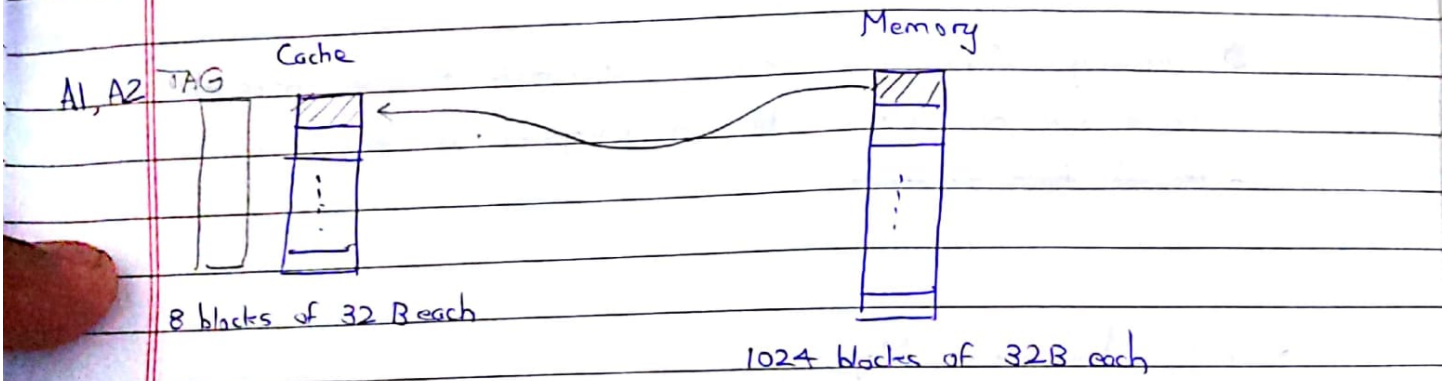
'Locality of reference'

A1, A2

- Increasing the number of blocks in SRAM beyond a certain point will increase access time from 1 cycle to 2 cycles ∴

→ Questions to be answered

1. In which block of cache to place?
2. How to identify where required instruction is placed in cache?
3. Which cache instruction to replace when overwriting?
4. When to write ~~to cache~~ & data (obtained from processor) from cache to memory?



In memory, every byte can be addressed as

Block No.	Offset

10 bits (mem) 5 bits
3 bits (cache)

cache

- We will only be transferring blocks as a whole from memory to cache. After transferring, offset part of each byte remains same. Block no. will be changed accordingly
  - We maintain a mapping of which block no. of memory is ~~map~~ transferred to which block no. of cache

- Simplest Mapping = 'Direct Mapping'

(Mem Block No.) % 8 = (Cache Block No.)

eg- Mem Block 45 is transferred to Cache block 5.

P.T.O.

i.e - ~~In~~ Mem blocks

(Cache block no.) = Last 3 bits of (Mem blocks no.)

∴ (Cache address) = Last 8 bits of (Mem address)

- The first seven bits of Mem address (which were last in transferring to cache) will be stored in an adjacent array to cache. (Tag)
  - TAG is an 8-element array of 7-bit elements each (one element for each block)

1/11 • Checking if required memory data/instruction is present in cache can be done with a comparator in TAG.
~~- We will have to check~~

→ Fully Associative Mapping

- Any memory block number can be placed in any cache block number
- Better utilization. ☺
- Now, TAG[element] needs to have all 10 bits of memory block number
- Now, comparators have to check TAGs of every cache block
  - May not happen in 1 cycle ∴

→ Take a middle path : 'Set Associative'

- Divide cache into multiple sets

eg - 4 sets of 2 blocks each

'2-way set associative'

- Mapping of memory to each 'set' in cache is fixed. But, within that set, any block can be used.

(Cache set no.) = Last 2 bits of (Mem block no.)

∴ Tag element size = 10 - 2 = 8 bits.

A1 Write-~~Back~~ Through Policy

Write back into memory whenever you are writing from processor to cache.

2 Write-~~through~~ back Policy

Write to memory only when that block needs to be rewritten.

A3 Replacement

- In direct mapping, you don't have a choice. Need to rewrite in same block no.

- In fully associative mapping or set associative mapping,
  - Add in empty block if present,
  - If not block is empty, we want to replace the block that is least useful,
  - Possibility - FIFO
    But first-in block need not be least useful.
  - 'Least recently used'
    eg → In 2-way set associative
    Use an LRU bit per set of 2 blocks

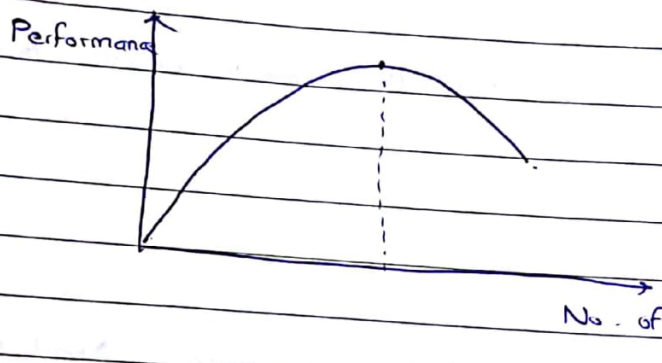
LRU = 0 :- Replace 0 . Make LRU = 1
1 :- 1 . 0

} Keep toggling LRU bit

D_1 D_2

If you are using Write Back policy, maintain a 'dirty' bit for each block.
Make dirty = 1 if block was updated.
If dirty = 1, write back to memory before replacement.

* IBM's experiment for pipelined design.
- Given the typical programs we run, • least achievable CPI = 1.15



- What we have studied :- 'Scalar pipeline'
1) Fetch one instruction at a time.
2) All instructions pass through the same 'Unified' pipeline
3) Instructions advance in lock step.

- Lift the restriction of fetching one instruction at a time.
Then ideal CPI < 1
- Fetch multiple instructions at a time

- If all instructions are independent,

```
a = b + c
d = e + f
g = h + i
j = k + l
```

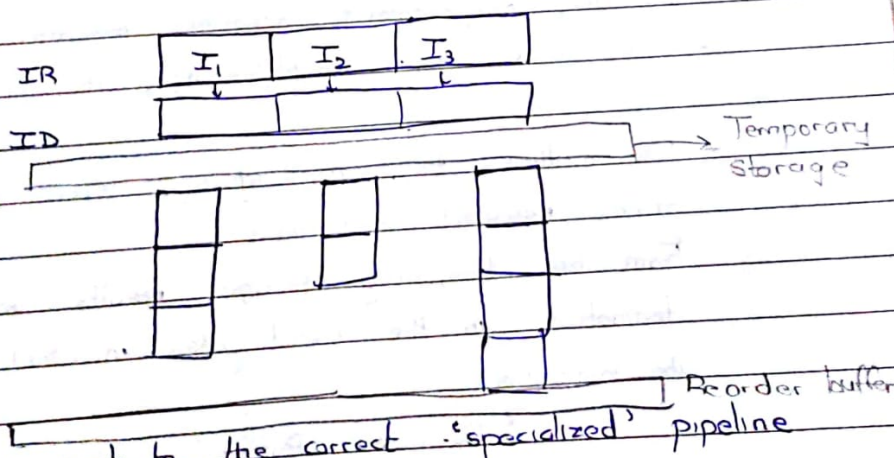
Then, fetch, say, 2 instructions per cycle. (Relax 1)
Duplicate the entire datapath to make two pipelines. (Relax 2)
(RF, memory needs to have more I/O ports)
∴ IPC = 2, CPI = 0.5

- If there is dependency between instructions,

I_1 : $a = b + c$
I_2 : $d = a + e$
I_3 : $p = q + r$
I_4 : $s = p + t$
I_5 : $i = j + k$

- Relax condition 3 :- Remove lock step advancement
  - By-pass I_2 and I_4 for the time being.
  - Execute I_1 and I_3 simultaneously first, I_2 and I_4 simultaneously next.

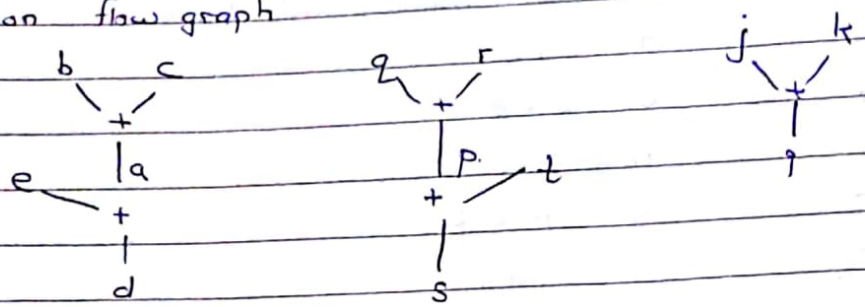
- Relax condition 2 :- Different pipelines have different number of stages.



Each instruction is sent to the correct 'specialized' pipeline

- Between decoder and pipelines, make a temporary storage to store, say, 100 decoded instructions
  - In this storage, which instructions are independent is determined
    - Independent instructions are sent for execution simultaneously first.

- Instruction flow graph



Can be done in minimum 2 cycles $\Rightarrow IPC_{max} = 2.5, CPI = 0.5$

- Create the following mechanism:-
  - Each instruction in temporary storage will check if its operands have been produced and are available.
    Once they are available, it is ready to be sent to pipeline.

- But now, programmer's intended program order is violated.
  - Will cause issues in debugging, interrupt execution, etc.

  - Use another temporary storage after the pipelines, which stores executed instructions.
    From this temporary storage, results are written back to destination in the exact order in which programmer wrote the instructions.

'Re-order Buffer'

- When any instruction is added to temporary storage 1, a note is made in re-order buffer, so that instruction order is remembered