Single - Pipelined design - 'Program guided execution'
For they follow instruction order.

Page No.
Date

VERY LONG INSTRUCTION WORD.

I

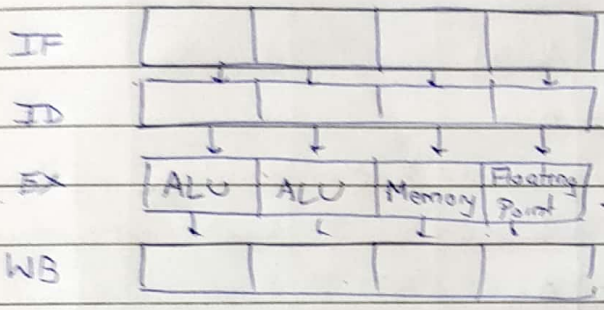
→ Beyond scalar architecture

1. Fetch multiple (but contiguous) instructions

eg - a = b + c
d = a + e
p = q + r

Can execute I1 and I3 simultaneously.

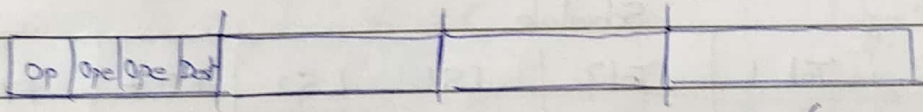
- Discover independent instructions → Software ⇒ Compiler
  → Hardware.



..... All pipelines are specialized based on proportions of types of instructions on average

Compiler will try to improve efficiency by executing independent instructions in that specific proportion of types.

- This system uses new ISA, consisting of '4' combinations of individual instructions.



'Very Long Instruction Word' (VLIW)

- If another pipeline is added, entire ISA changes ∴
- If specialized instruction is not found, NOP is executed in that pipeline

VLIW is tightly coupled with machine organization

- Immediate instruction dependency - No forwarding mechanism.
  - Dependent instruction is executed only when previous result is written back
  - Compiler is more complex ∴

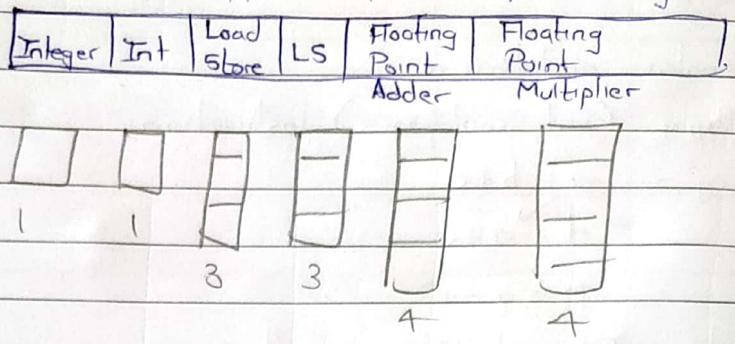
Superscalar architecture is more sensitive to cache miss than scalar

- More sensitive to I cache :- Stalling for more cycles
- Less D :- Even if miss, other instruction can execute

Page No. Date

→ Floating Point / Scientific application :- Loop Unrolling + Software Pipelining

eg



for (i=0; i<1000; i++) { A(i) = B(i) + C; }

All floats

r_1 contains base B(0)

r_2 A(0)

r_3 8000 + r_1

f_0 (floating point register) contains C

Meth 1

back:	load	f_1, (r_1) 0	1
	add	r_1, r_1, 8	2
	fadd	f_2, f_1, f_0	3
	store	f_2, (r_2) 0	4
	add	r_2, r_2, 8	5
	bneq	r_1, r_3, back	6

Schedule

Cycle	Int 1	Int 2	LS1	LS2	FP+	FP*
1	I2		I1			
2						
3						
4				I3		
5						
6						
7						
8	I5	I6	I4			

3 cycle delay before dependency

4 cycle delay

Blank spaces = NOPs ∴ ∴

Number of instructions to unroll:-

Minimum = Maximum latency of any pipeline = 4

Maximum depends on resources (registers, etc) available

Page No. | Date

∴ 8 cycles for one floating point operation (one iteration)

Terrible

Meth 2 Unroll the for loop:-

```
for (i = 0; i < 1000; i = i+4){
        A(i)   = B(i) + C;
        A(i+1) = B(i+1) + C;
          2        2      ;
          3        3      ;
}
```

• Assembly:

```
back:  1  load  f1, (r1)0
       2  load  f2, (r1)8
       3  load  f3, (r1)16
       4  load  f4, (r1)24
       5  add   r1, r1, 32
       6  fadd  f5, f1, f0
       7  fadd  f6, f2, f0
       8  fadd  f7, f3, f0
       9  fadd  f8, f4, f0
      10  store f5, (r2)0
      11  store f6, (r2)8
      12  store f7, (r2)16
      13  store f8, (r2)24
      14  add   r2, r2, 32
      15  bneq  r1, r3, back
```

Cycle	Int 1	Int 2	LS1	LS2	FP+	FP*
1			I1 load 1	I2 load 2		
2	I5 add r_1		I3 load 3	I4 load 4		
3					I6 fadd 5	
4					I7 fadd 6	
5					I8 fadd 7	
6					I9 fadd 8	
7			I10 store 5			
8			I11 store 6			
9			I12 store 7			
10						
11	I14 add r_2	I15 bneq	I13 store 8			

☺ Performance = $\frac{4}{11}$ floating point operations per cycle.

☹ Higher use of cache and no. of registers, the more you unroll

* VLIW requires ↑ cache BW than superscalar
: ISA has ~~the~~ NOPs included (↑ no. of bits) in VLIW

* VLIW does not need branch predictor

Page No.
Date

'Software Pipelining'

→ Better :- Pipeline each unrolled iteration
- Say, for eg, use LS1 for load only & LS2 for store only.

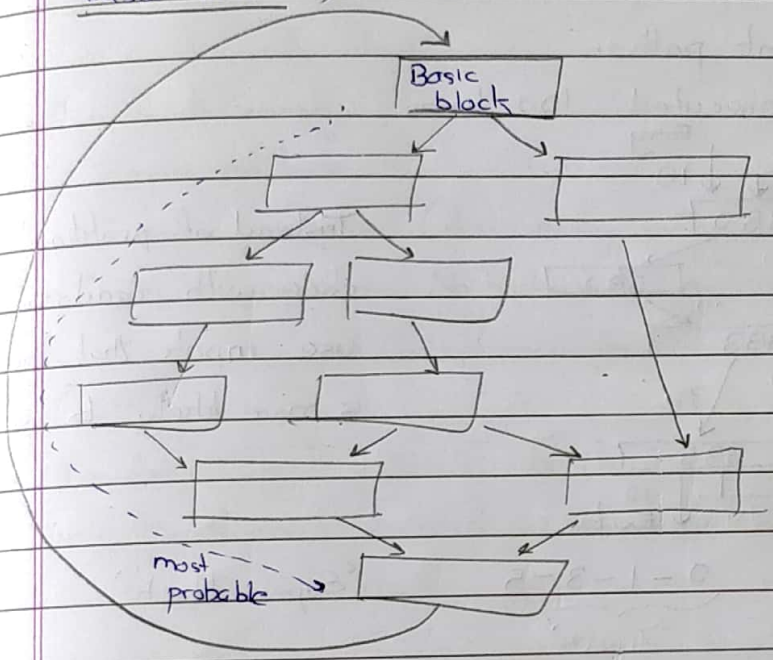
	Int 1	Int 2	LS1	LS2	FP+	FP*	
1			Lf_1				
2			Lf_2				
3			Lf_3				
4		addr,	Lf_4		fadd f_5		
5			Lf_1		fadd f_6		
6			Lf_2		fadd f_7		
7			Lf_3		fadd f_8		
8		addr,	Lf_4	Sf_5	fadd f_5		This block contains all instructions in each unrolled iteration.
9			Lf_1	Sf_6	fadd f_6		
10			Lf_2	Sf_7	fadd f_7		
11	addr$_2$	bneq	Lf_3	Sf_8	fadd f_8		
12		addr,	Lf_4	Sf_5	fadd f_5		
13			⋮	Sf_6	fadd f_6		
14				Sf_7	fadd f_7		
15	addr$_2$	bneq		Sf_8	fadd f_8		
16				Sf_5	⋮		
				⋮			

☺ Performance = One floating point operation per cycle

* "Lock-free advancement" - No checking anything ~~at~~ once instruction enters pipeline.
- No stalling (even in branches, because we can schedule branch only when ~~instruction~~ destination is computed)
- Stalling occurs only on cache miss.

→ Non-loop code :-

Usually, there are several branches. We can only correctly schedule instructions between consecutive branches (in one 'basic block') ∴



While compiling, run a few sample test cases. Choose a static branch prediction based on your findings. Then schedule instructions considering most probable path.

- If most probable path is taken, life is chill.

Most probable path = 'Super block'

- If most probable path is not taken → high penalty
  - Compensation must be done - Changes of wrong prediction ~~may~~ have to be undone.

14/1 * Basic block ≙ Must enter from the top, leave from the bottom

- If we ~~try~~ to predict just one branch ahead - no major benefit.

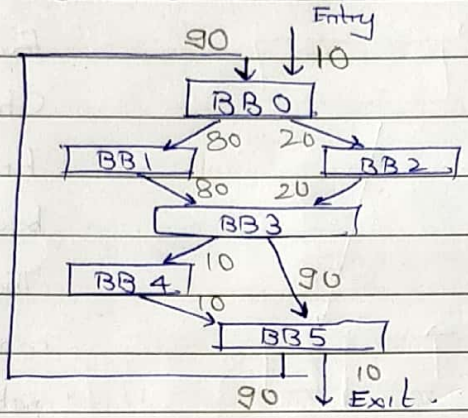
eg

a = b + c	①	①
d = a + e	②	②
p = q + r	①	not ①
s = p + t	②	③
i = j + k	③	②
m = i + l	④	③
	4 cycles	3 cycles

- To improve performance, increase no. of instructions and variety of type of instructions within a basic block (to keep the pipeline filled)

- Finding most frequent path :-

Following code is executed 100 times



Instead of 'profiling' the code with random inputs, use inputs that the code is more likely to see.

Most probable path :- 0 - 1 - 3 - 5 ('Trace')

'Super block'

Treat super block as a basic block and schedule all of its instructions together. (better spatiality of reference) ☺

- Cache utilization will also improve, because BB2, 4 need not be brought into cache.

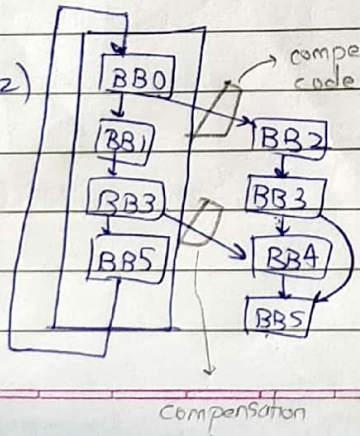
- Deviations from super block path.

eg - You want to go to BB2

- Two possible policies :-

1) ~~Entry into superblock not allowed~~ Exit from superblock not allowed in between (allowed only after BB5) (↑ compensation code)
~~Exit from superblock allowed~~ Exit allowed from between (BB0 or BB3)

2) ~~Vice versa~~ Entry back into super block is not allowed.



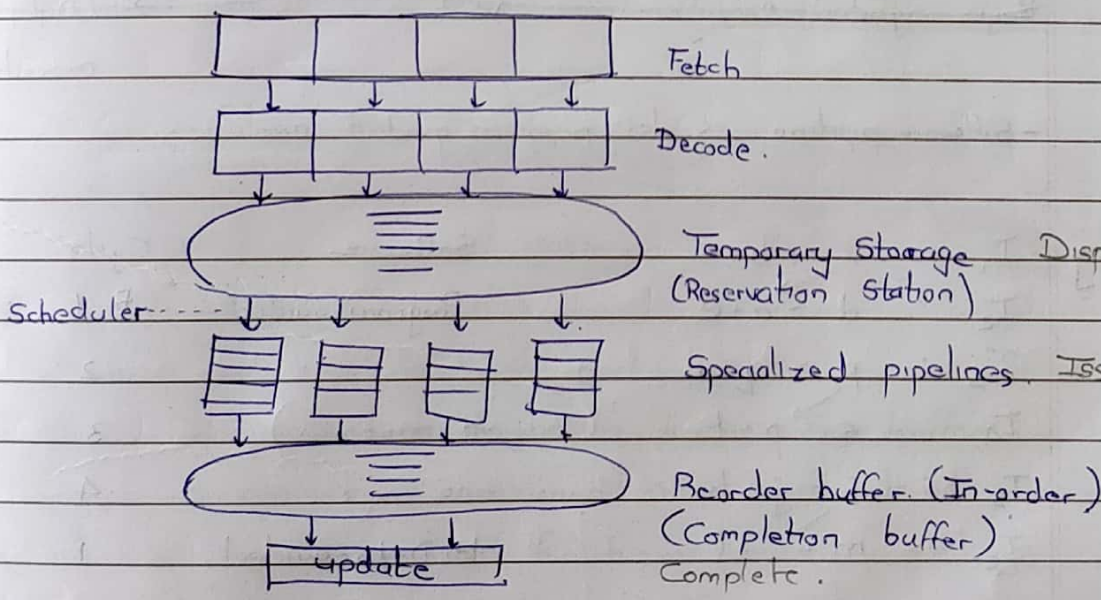
Compensation code is required because incorrect instructions from further blocks have to be executed.

Page No.
Date

- Super-block formation – No loops in the middle
  – Every path's probability ≥ Some predecided threshold (say 80%)
  (∵ frequency of deviation is low)

→ VLIW Summary
- Simple machine
- Complex compiler (discovering independent instructions)
- ISA changed – No backward compatibility ☹
  (Code from older machines cannot be run on this)

- We don't want ISA to change.
  ⇒ Hardware, instead of compiler, should be made to discover independent instructions



Page No.
Date

Temporary storage - Instructions are scheduled when all their operands are available

- It is important to maintain order of instructions while writing back (for debugging, interrupts, etc)

* When an interrupt occurs, all further instructions have to be fetched again.

∴ Use another temporary buffer at the end of pipelines.

17/1 *

	~~Software~~ Pipeline EE309	Hardware Parallelism Detection
Fetch single instruction	✓	×
Uniform pipelines	✓	×
Lock-step advancement	✓	×
~~Program-guided execution~~	✓	× (~~Data-guided~~ execution)

- Software pipeline is also program guided execution.

eg

I_1 : $a = b + c$
I_2 : $d = a + e$
I_3 : $p = q + r$
I_4 : $s = p + t$
I_5 : $i = j + k$
I_6 : $m = i + l$

~~Software~~ Program-guided :-

Cycle	Instruction
1	1
2	2,3
3	4,5
4	6

~~Data~~-guided

Cycle	Instruction
1	1,3,5
2	2,4,6

Problems :- ~~Interrupts~~, Debug. → Need to write back in program order and maintain a PC as well.

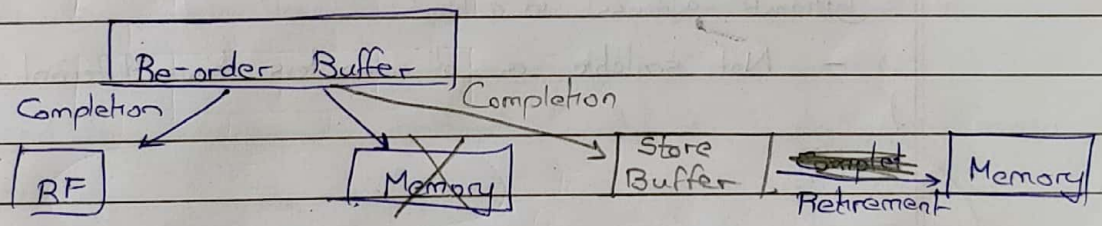
Programmer perceives program-guided execution.

Page No.
Date

- How to maintain order before updating?
  - Remember (pass through every pipeline) PC of every instruction
    - ✗, because of branches, loops in code.
  - When you ~~fetch~~ an instruction, move to reservation station make a corresponding entry in Re-order buffer immediately.
    - Instructions remain in order till decode stage.
  - ~~*~~ For stability:- No. of instructions that can be written back = No. of instructions fetched.

- Reorder Buffer ~ Circular Queue of instructions
  - When result of (consecutively next) instruction is available, write back and deallocate its place in reorder buffer.

- Writing back is ceremonial.
  - Any future dependent instruction can get its operand from the results stored in re-order buffer itself.



  - Memory can have only one port ☹
  - Many Load instructions might be in reorder buffer
    - Load execution is always in critical path of pipeline. & and is important for future instructions. If correct value is not loaded in time, it might cause a future cache miss ☹ ☹
    - ∴ Among Load/Store, Load must be prioritized.
    - ∴ Use a Store buffer to store results of store and write back, ~~or~~ from SB to memory only when memory is not busy.

Page No.
Date

→ Challenges

We want High Bandwidth Fetch

1 Cache Miss for ~~Causes~~ ~~each~~ instruction cache.

2 Branches - Many fetched instructions go waste.

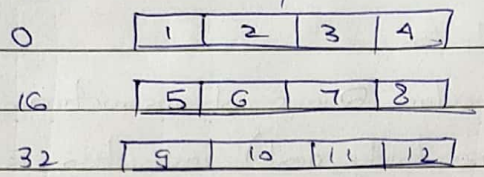
Improve prediction rate

- When there are many branches in the code, overall probability of correct branch prediction reduces due to cascade

3 Branch target misalignment

- Memory organization :- (4 bytes each)

If 4 instructions are fetched at a time, memory is organized in multiples of 16 bytes. Can only fetch such 4 instructions together.



If 4 is a branch whose destination is 11, then 9 and 10 would still have to be fetched along with 11 and 12.

- Solution 1 :- ~~xx~~ 0

Compiler allocates destination of branches so that they lie in the leftmost address in a block.

- Not scalable as all computers have different memory architecture.

21/1

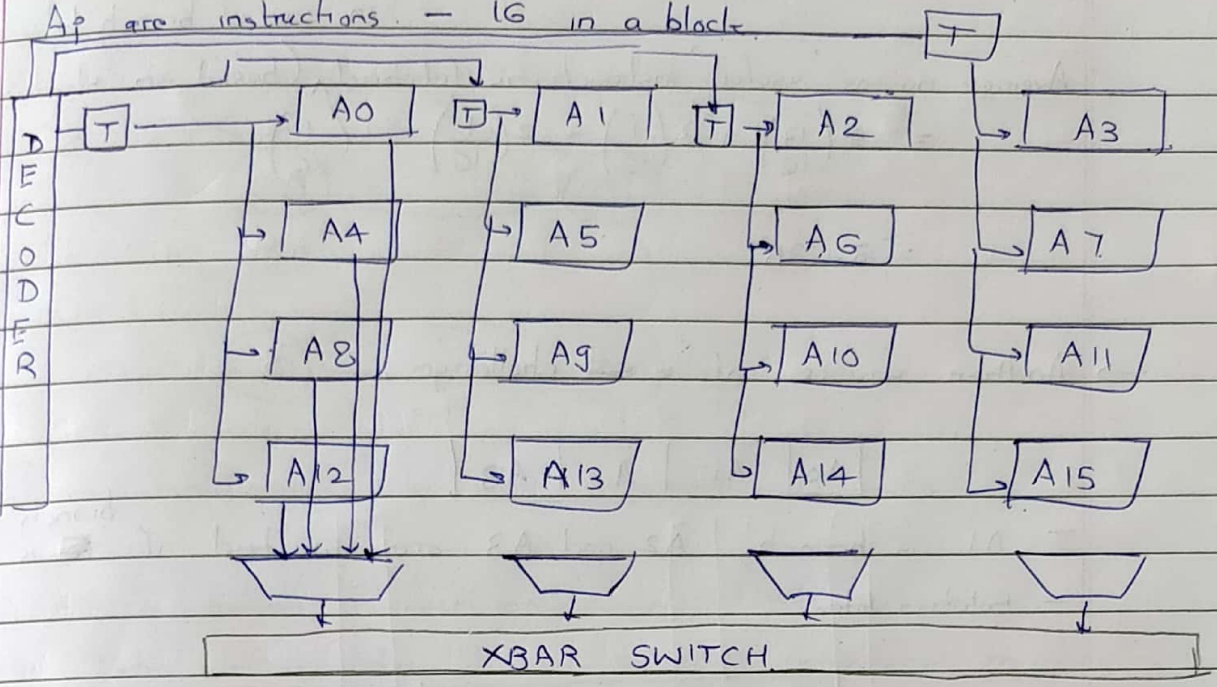
- To execute 4 instructions, we also need to write back four instructions per cycle. (Inflow = Outflow)
- RF can have more than one port.
- Memory cannot, as it will increase memory size and access time.

IBM's solution

One cache block has 64 bytes (16 instructions)

This block is arranged over 4 rows. (PTO)

Cache is 2-way set associative.



- In any cycle, you can read one instruction from each ~~row~~ column
- Every column has a T-logic
- If starting address to be fetched is A2, fetch A2 and A3 from columns 3 and 4, but A4 and A5 from columns 1 and 2.
  - No use if starting address is A13, A14, A15

- ∵ Need to reorder :- Instead of $A_2\ A_3\ A_4\ A_5$, you seem to be reading $A_1\ A_5\ A_2\ A_3$.
  - Done by 'XBAR Switch'
    - This is big design. May not have 1 cycle access time.
    - Possible solution - Pipeline this fetching of instruction
      - Cycle 1 : Match from starting address
      - 2 : Fetch 4 instructions
      - 3 : Reorder.
    - Bad in case of branch ~~prediction~~ taken or branch misprediction
    - Second solution :- Increase cycle time ☹

Page No.
Date

in case of branch taken, and no

Average no. of 'useful' instructions fetched (based on starting address)

$$= 4\left(\frac{13}{16}\right) + 3\left(\frac{1}{16}\right) + 2\left(\frac{1}{16}\right) + 1\left(\frac{1}{16}\right)$$

$$= 3.625$$

→ Another source of challenge.

A0	A1	A2	A3

If A1 is branch, A2 and A3 are wasted if branch is taken.

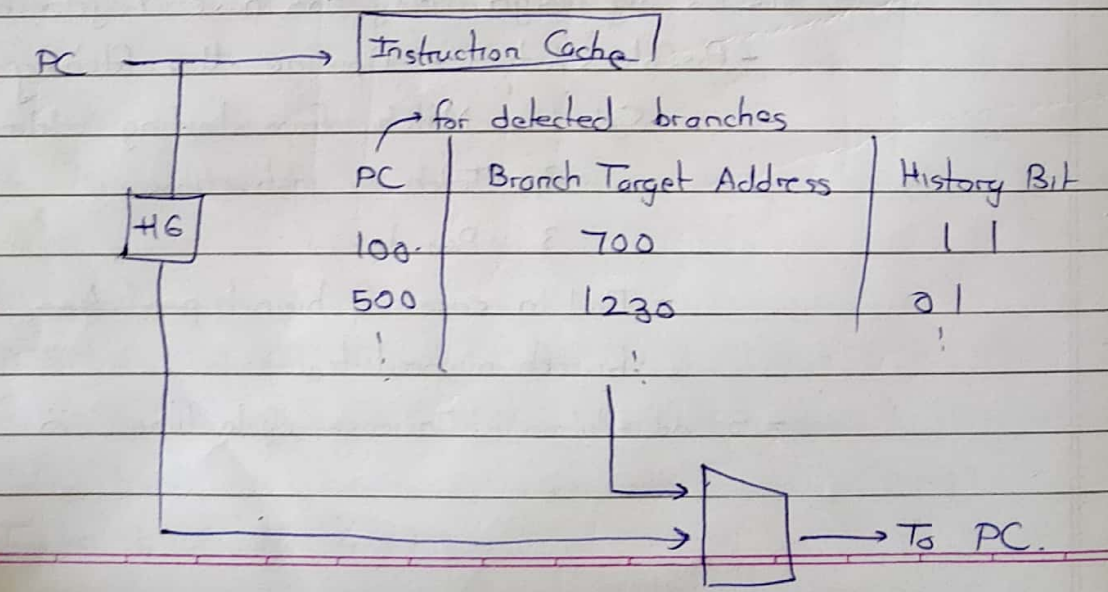
- Solution later

BRANCH PREDICTION

→ Branch → Conditional
→ Unconditional

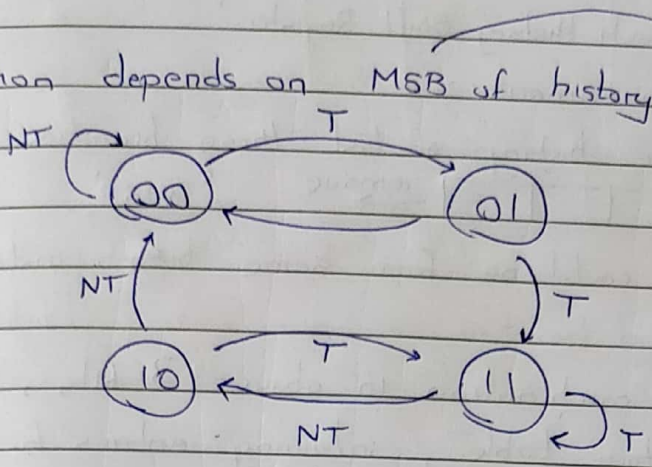
- Unconditional branch instructions can be detected and acted upon in decode stage.
- For conditional branch, we need to know:-
  1) Direction (Taken or not taken)
  2) Target Address if taken.

• For direction :- Branch History Table



cache miss

- Prediction depends on MSB of history bits. { 0 = NT, 1 = T



'Change prediction only when mispredicted twice'.

- 85-90 % ~~better~~ correct prediction ⇒ Not enough
  - Extra history bits do not increase accuracy much :

- Better :- Make prediction based on other nearby branches (Determine correlation between branches).

eg- if (i>100) { ..... }
    if (i==0) { ..... }

If first branch is taken, second will never be.

- Size of History table :- ('Pattern History Table')
  - Cannot be very large ⇒ Access time > 1 cycle.
  - Will have to remove some other instruction if overflow
    - Lose all learning for the removed branch.

i) Separate 'Target address' (smaller (64 entries)) and 'History' table.
  - Instead of fully associative, make history table direct mapped
    - If PC is 32 bits, only check entry corresponding to last 10 bits
      - Total 1024 entries in history table.
      - Problematic if two branches (behaving differently) have same last 10 bits of address.

'Aliasing'

Both will update history bits

Fully associated history table size cannot exceed 128

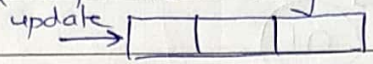
Page No.
Date

TWO LEVEL PREDICTORS

'Branch History Shift Register'

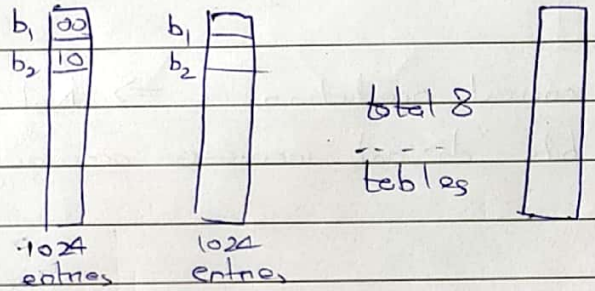
- Learning about environment :-

Maintain a separate history of last three branches



The three entries could be from same branch instructions.

For each possible combination in above 3 bits, maintain one pattern history table, containing entries for every branch instruction.



Every combination of above 3 bits is an indication of the 'path' via which the current branch was arrived at.

- Predict your prediction based on the PHT corresponding to above 3 bits.

~~- If you consider~~

- If you consider these 8 PHTs as one long PHT, it will be indexed by 13 bits (10 from PC + 3 from BHSR)

24/1 eg

```
for (j=0 ; j<100, j++) {
    for (i=0; i<3; i++) {
    } b2
} b1
```

In order ~~TTT~~ TTT NT TTTT NT TTTTNT

$b_2 b_2 b_2 \; b_2 \; b_1 b_2 b_2 b_2 b_2$

If your BHSR is 4 bits long, there will be 16 PHTs, each having one entry each for b_1 and b_2, each entry ~~PHT~~ containing (two) history bits

- BHSR size :- Higher :- Checks correlation over farther branches ☺
  Exponentially high number of PHTs ☹

- If BHSR size = 4 (say), you can either consider there to be 16 independent PHTs; ~~or one~~ indexed by 10 bits of PC, or one large monolithic PHT indexed by both BHSR value and last 10 bits of PC.
  - We can choose any BHSR size and no. of bits of PC such that
    → BHSR size + no. of bits of PC = constant
    = maximum possible that would allow for single cycle use of PHT.
  eg BHSR size = 4, no. of bits of PC = 10.

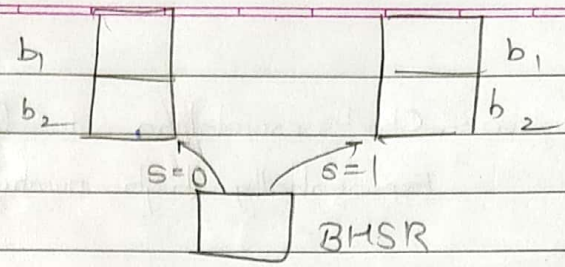
G-Skew Predictor

Better! Take BHSR size = 14, no. of bits of PC = 14.
- Index of PHT = BHSR ⊕ PC bits
  - XOR operation randomizes the indices
  - High BHSR :- Farther correlation ☺
  - Randomization + more no. of PC bits :- Lower aliasing ☺

eg
```
if (i % 2 == 0) then        // b1        (Taken if divisible)
{
}
if (i % 10 == 0) then       // b2
{
}
```

- BHSR size = 1 ⇒ 2 PHTs, with single history bit (all initially zero (not taken))
- Inputs = {10, 15, 21, 22, 27, 37, 40}

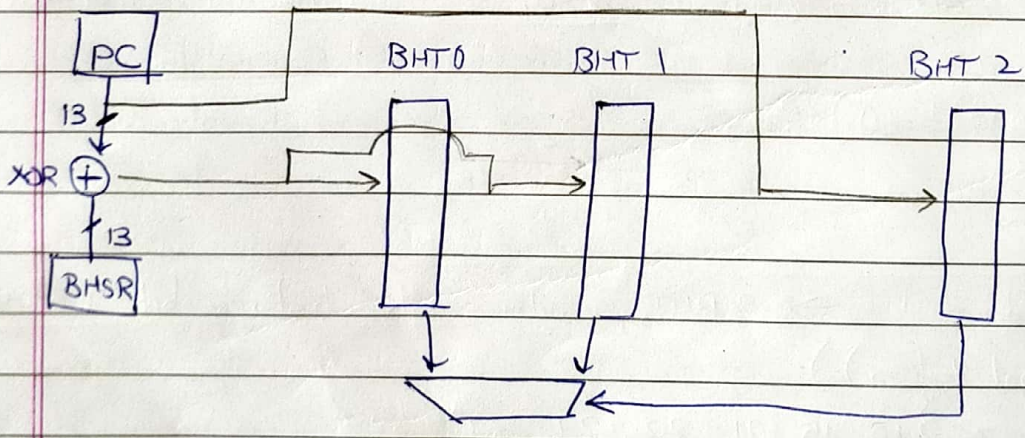
Find prediction accuracy.



	i	10	15	21	22	27	37	40
S=0	b_1 predicted	0		1	0	1	0	0
	b_1 actual	1		0	1	0	0	1
	b_2 predicted		0	0		0	0	
	b_2 actual		0	0		0	0	
S=1	b_1 pred		0					
	b_1 actual		0					
	b_2 pred	0			1			0
	b_2 actual	1			0			1

Observation :- For S=0, b_2 is always correctly predicted
(if not divisible by 2 ⇒ not divisible by 10)

BIMODE PREDICTOR



BHT0, BHT 1, BHT 2 are entire PHTs, with each entry having 2 ~~entry~~ history bits

BHT0, BHT 1 are indexed by BHSR ⊕ PC

BHT2 is indexed by PC alone.

Page No.
Date

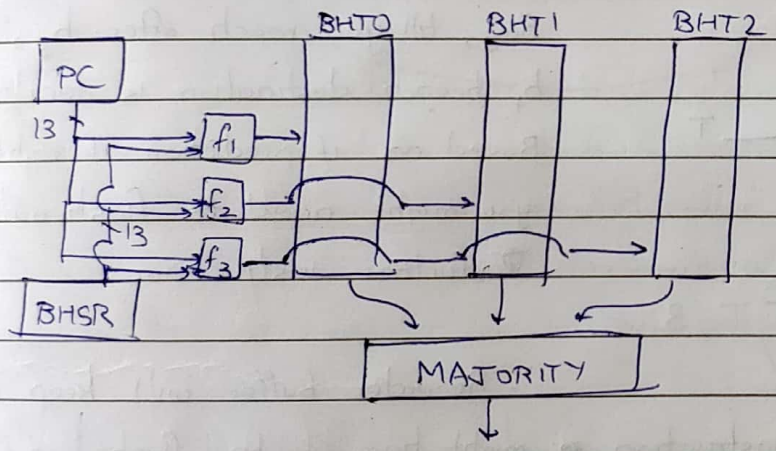
∴ BHT0/1 and BHT2 are aliasing in different ways.

- Based on MBSB in BHT2, pick BHT0 or BHT1.
  Predict based on entry in picked BHT.
  Update BHT2 and picked BHT
- ~~BHT0 will e~~
- Eventually, BHT0 gets biased towards T/NT
  BHT1 NT/T

} If both aliasing branches behave differently.

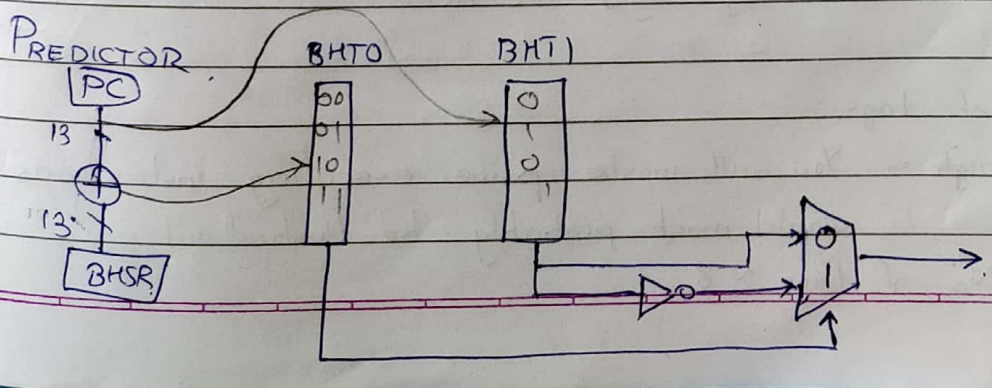
'Dealiasing'

MAJORITY PREDICTOR



BHTs 0, 1, 2 are trained independently and indexed by different hashing functions of PC and BHSR.
∴ Aliasing is different in all 3 tables.
Final prediction is majority of all 3 predictions.

AGREE PREDICTOR

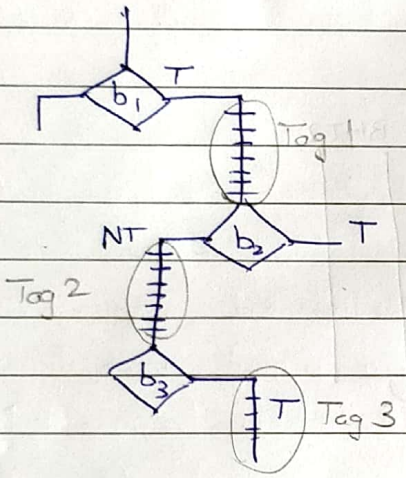


- BHT1 is filled up by compiler. This allows for better initial prediction, before hardware training is done.

- BHT0 is trained while running. Its entries tell us if prediction should agree with 0 or differ from compiler prediction in BHT1.

  11 → Strongly Agree

  10 → Weakly Agree.

→ 

Say, you predicted b_1 T, b_2 NT and b_3 T.

Say, till you reach after b_3, your b_1 branch destination is decided. Based on if prediction if right, you might need to flush out further instructions.

∴ Re-order buffer will keep track of whether an instruction might have to be flushed out ('speculative' bit) and which Tag a speculative instruction belongs.

- If b_1 prediction is right, Tag 1 instructions are made non-speculative. Tag 1 is freed.
- If b_2 prediction was wrong, Tag 2 and Tag 3 instructions are made invalid. Tag 2 & 3 are freed.

- No. of tags :-

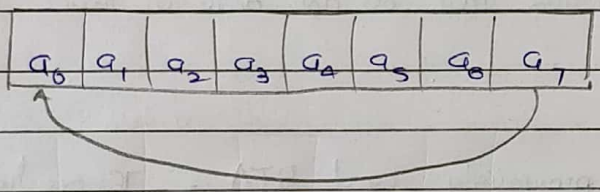
Too high :- You will waste power executing instructions that would most probably be flushed out.

∴ No. of tags ~ 8.

If all tags have been allocated, stop fetching new instructions till tags are freed (cause stalls)

→ "Return" Branch

- Unconditional jump, but you cannot predict destination, which depends on where that function was called from.
- store the return addresses for each ~~seq~~ function call in a circular queue on hardware.
- We only know the correct address to return to in EX stage of return instruction



- ~~e~~ If a return instruction prediction is resolved, erase its entry.
- Queue will be full if there are 8 nested ~~rd~~ function calls. If a ninth function is called, ~~its predicted address will overwrite~~ a_0 it has no more place on hardware queue.

Option 1: Stop predicting till a_7 is resolved (stall)

Option 2: Place a_8 in place of a_0. While you continue executing, your returns will be resolved in reverse order (8-7-6---) By the time you come to the 0^{th} return, hope that its return address has been received from execution stage. If not received, stall till received.

Predict NT
T

- Actual T = Fine, because address will be stored
  NT = Bad, don't have original address.
  ∴ Every branch instruction must store PC

28/1.

BRANCH TARGET ADDRESS

- We can make fully associative :- PC → BTA
  - Size cannot exceed 128.

Better
- Make directly mapped :- (based on last 10 bits of PC)
  - Also store a 22-bit TAG to correctly match entire PC

TAG (22)	BTA (32)

To solve aliasing, you could store more tags

→ Multitargeted branches

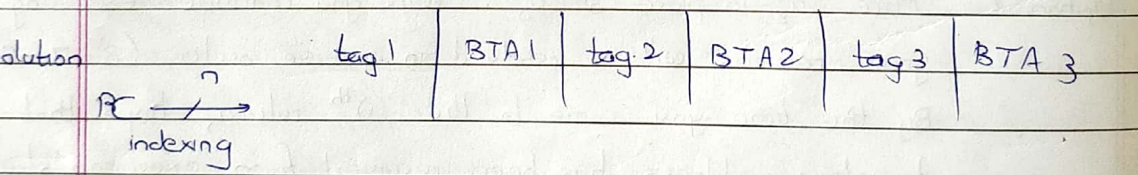
Problem :- Branches that do not have a fixed BTA.

eg Switch Case

- Always store previously used BTA. If prediction is wrong, penalty

* For example, branch could be multitargeted because its BTA could come from a register value.
* Multitargeted branches could come from object-oriented programming
  eg - Operator overload
  eg - Functions that behave differently based on no. of parameters

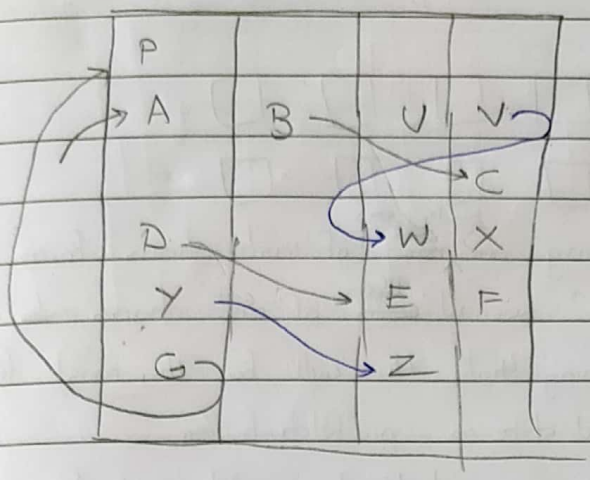
Solution



Multitargeted branches :- 2 of the tags will be same. BTA's will be different.

- 'Selection logic' - If more than one tag is same, choose one of the BTA's based on some pattern-based logic
  - Could be implemented by perceptron neural networks.

We get 40% accuracy in multitarget prediction



Cycle #	No. of useful instructions fetched	
1	2	(AB)
2	1	
3	1	
4	2	
5	1	
6	4	

Very few average useful instructions fetched per cycle

- Maintain a mapping in cache, based on PC of current (first) instruction 'A' in hardware → 'Trace cache'

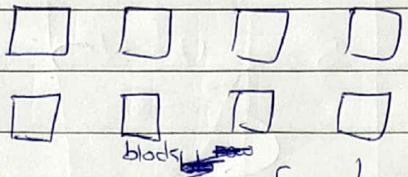
	PC	3 Predicted branches	Trace	Instructions in a trace are entirely fetched and stored in trace cache
PC of taken branch instruction	AB	1 1 1	ABCDEFGP	
	AB	0 1 1	ABUVWXYZ	

Only frequently taken paths are stored in cache, not all possible paths.

- Which traces are tabulated is decided by code profiling, based on which instructions have been written back.
  (Build trace cache at the time of retiring the instruction)

- Such a trace cache takes a while to be populated.

→ Pre-fetch buffer.

- L1 cache → 

Whenever you are fetching one block of instructions from L1 cache to CPU, 'pre-fetch' second block of instructions from L2 cache, (hoping that this will be the most likely next block to be executed), and store in pre-fetch buffer.
This is helpful because second block might not be in L1 cache

→ Victim buffer

→ Instead of replacing old block in cache by new one, the old one is temporarily moved to victim buffer. Hence, if you need the victim block again you can avoid a miss by simply swapping L1 cache block and victim buffer.

∴ CPU can fetch one of :- L1 cache block, Trace cache, pre fetch buffer, victim buffer, branch predictor.

- If the next required instruction is available in more than one source, preferrably fetch from trace cache (because it is most likely to give most likely trace of instructions)

→ If there is a loop running many times, we can store all its instructions in another buffer.

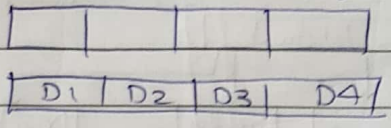
→ Intel bypasses Fetch and Decode for further times by storing decoded instructions in trace cache.

Page No.
Date

31/1

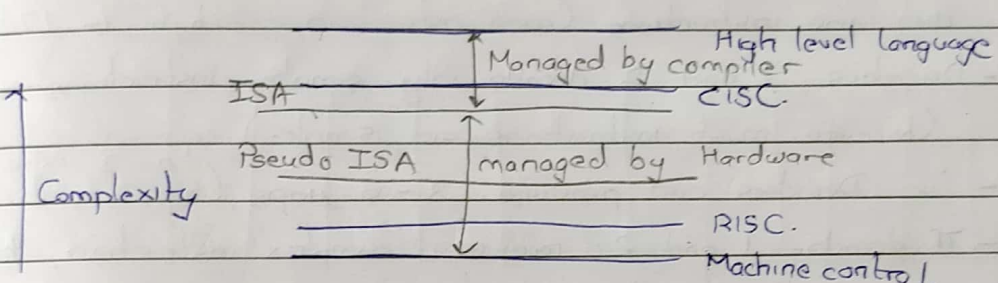
'Intra fetch group dependence'

→ Dependence of instructions fetched together on each other.



$2 \times {}^4C_2$ no. of dependencies possible ∀
↳ each operand
– This information is used during dispatch.

→



- Intel uses RISC-~~ISA~~ like ISA in background, but packaged for user as a CISC-like ISA.
ISA needs to be translated into pseudo-ISA by hardware.

- Every complex CISC-like instruction will be broken down into smaller RISC-like 'micro-operations'

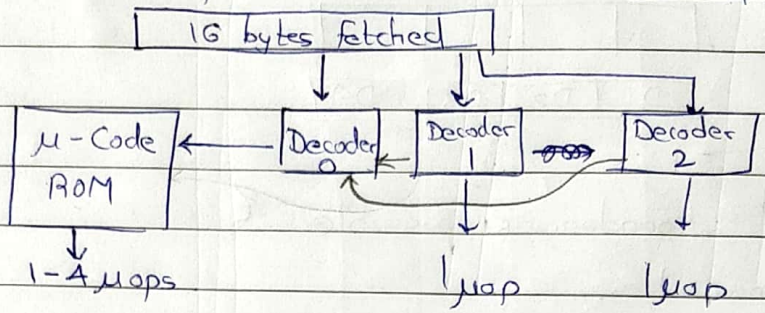
eg- add r_1, a (↓ address) →
load t_1, a.
add t_2, r_2, t_1
store t_2, a

– Simple instructions remain simple (eg- add r_1, r_2)

- ↑ Power consumption by hardware for translation into pseudo ISA.
↓ Efficiency because each instruction is translated one by one

- 16 bytes are fetched together : May contain variable number of CISC-like instructions.

- Translation into pseudo - ISA : 'Look-up Table'



- More than one decoder because fetched bytes can contain more than one instructions (each instruction :- 1 byte to 17 bytes)
- Decoders 1 and 2 decode only simple instructions. (because most instructions are simple)
- ∴ Decoders can produce 3 - 6 μops. (1 or 2 instructions would have to be stored in buffer)
- If decoder 1 and 2 receive a complex instruction, they forward it to decoder 0 in next cycle.

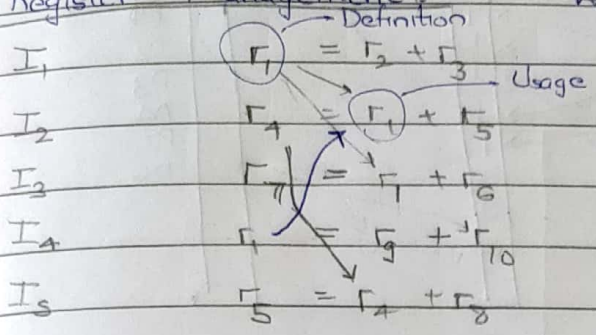
- We aim to produce 4 μops per cycle (like before)

- Interrupts :-

All μops of one instructions will have one PC.
Interrupt may occur in the midst of μops sequence.

Maintain 'Atomic' bit :- 1, if all μops are indivisible.
Must be written back together, if at all.
- 0, if μop is simple instruction

→ **Register Management** – ~~'Write After Write' (WAW)~~



Both I_1 and I_4 produce r_1 as result

If I_4 is executed before I_1 and I_2, I_2 might use I_4's value for r_1 ☹

$I_1 - I_2$: 'True' dependency : 'Read after Write' (RAW) dependency

$I_2 - I_4$: 'False' dependency – Because of absence of lock-step advancement

Output dependency (I_1 I_4)	Anti-dependency (I_2 I_4)
'Write After Write' (WAW) hazard	'Write After Read' (WAR) Hazard

- Bad Solution : ☹
  Stall I_4 till I_2 is completed.
- Better :
  Execute I_4, but store its result somewhere else.
- Best :- 'Register Renaming'

Architecture Register File

	Busy	Value (ARF)	Tag
R0		10	
R1	1	20	P1
R2		30	
R3		40	
4	1	50	P2
5	1	60	P5
6		70	
7	1	80	P3
8		90	
9		100	
10		110	
11		120	

Renamed Register File (RRF)

Busy (assigned)	Valid
1	P0
1	P2
1	P3
1	P4
1	P5
0	P6
0	

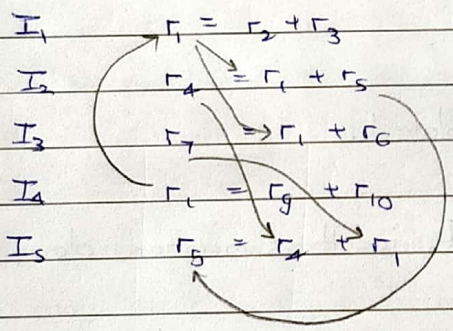
Reservation Station

Inst	OP	Operand 1		Operand 2		Ready	Tag
		Value	Valid				
1	ADD	30	1	40	1	1	P1
2	"	P1	0	60	1	0	P2
3	"	P1	0	70	1	0	P3
4	"	100	1	110	1	1	P4
5	1	P2	0	P4	0	0	P5

Ready = Valid 1 AND Valid 2

Re-order Buffer

Inst	PC	Dest	Tag
1	100	R1	P1
2	101	R4	P2
3	102	R7	P3
4	103	R1	P4
5	104	R5	P5



- Instructions are listed one by one in RS and RoB
- If both operands are available from registers, Valid 1 = Valid 2 = Ready = 1
  Else, we write tag (P1)

From where operand will come when P1 is valid (ie - if register is busy)

- A non-busy tag is allotted to destination register and Busy bits of both register and its tag are made 1.
- Begn by executing instructions whose Ready = 1 : I_1
  Store result in P1 (and make P1 Valid)
  ⇒ Instructions 2 & 3 became Ready
  Move on to I_4, then I_2, I_3

- When I1 retires (in lock-step advancement),
  Result transfers from P1 to R1
  P1 becomes not Busy

Maintain a list of free registers

P85	P42	P30

Allocate new RR's from this list.

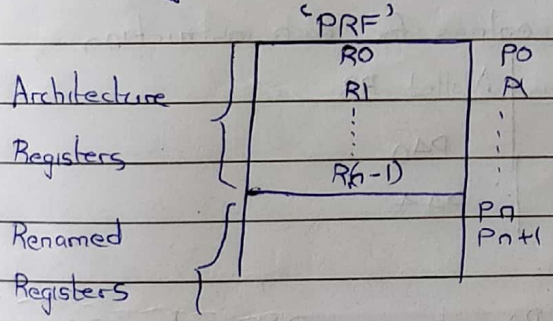
Page No.
Date

- It checks if tag of R1 is equal to the tag which is retiring. (If yes, make R1 not Busy)
  Because I_4 also has destination R1, R1 has tag P4 ≠ P1
  In this case, R1 must be kept Busy

4/2 • Value of operand 1 comes from a MUX with $2n + m$ inputs

- n = No. of instructions fetched per cycle
- $2n$ = No. of operands required per cycle = No. of ports that ARF needs.
- m = No. of specialized pipelines, each of which can broadcast its result after execution.

• Instead of making separate ARF and RRF, make one monolithic RF



- During retirement of RR's (writing them back to AR's), significant power is used.
- For allocation of unused RR's, looking for first non-busy RR would require heavy, time consuming logic of Priority encoder ∴

Look ↑

- During retirement of RR's, don't move value in RR to corresponding AR. Rename that RR as AR.
  eg - Make P40 'R0', instead of moving from P40 to R0.

- The instantaneous location in PRF of every AR is tabulated in Register ~~Architecture~~ Alias Table (RAT) / Register Map Table (RMT)

R0	P40
R1	P23
⋮	⋮
R31	P54

5-bits ~8 bits

RAT is updated only during retirement of instruction.
∴ Cannot be used for operand read.

- Speculative RMT
  - Just like RAT, except that it is updated whenever ~~execution of instructions is completed~~ instruction is decoded
  - Can be used to judge where to get operands from.
  - Since you are maintaining speculative RMT, you no longer need to store 'tag' for each AR.
-
- ROB contains following entries for each instruction

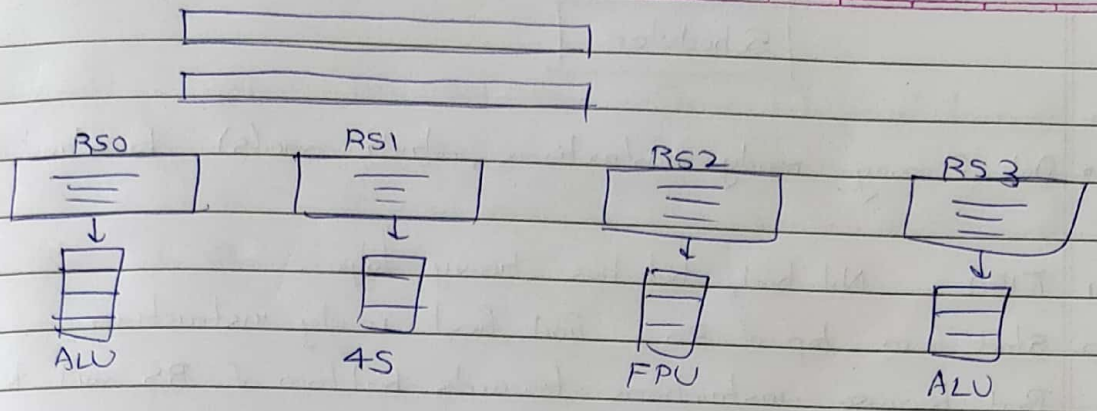
	Destination	Allotted RR
I1	R0	P40
I2	R0	P44

- ∴ Simply change R0 entry in RMT to P40 instead of moving P40 value to RMT

'Distributed Architecture'

→ Instead of one Reservation Station, every specialized pipeline has its own RS, meant only for instructions that use that particular pipeline.

P.T.O.



- If two pipelines are identical, we need to decide which RS to send those type of instructions.

- Centralized (Non-distributed) RS :-
  - Better utilization of space :)
  - Scheduler is complicated : Needs to scan through many more instructions to send to different pipelines. :(

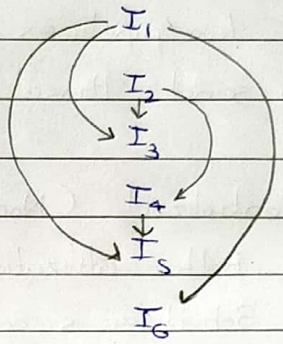
- Distributed RS :-
  - Scheduler is simple : Needs to schedule only one instruction (for its own pipeline)
  - Poorer utilization of space :
    For some particular code, one RS might remain empty while other RS is overflowing

- Hybrid RS :-
  - Pipelines which are identical/similar are managed by same RS and scheduler.
  - Middle path.

Scheduler

- Out of many ready instructions, which one(s) to schedule?

1. FIFO - Not bad, but has heavy logic.
2. Start from top of RS, find first ready instruction :- Bad, because instructions towards bottom of RS will be starved.
3. Maintain a dependency chain
   Out of ready instructions, schedule the instruction on which most number of instructions depend.
   - 'Fanout' - Doing this will make more and more instructions ready to be scheduled.

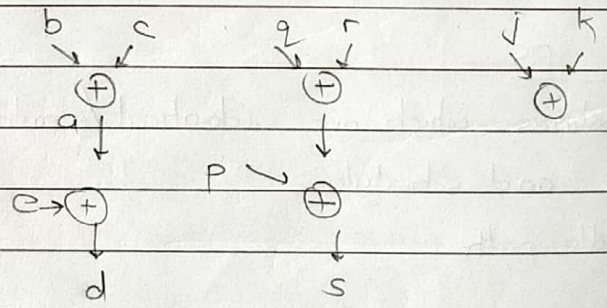


Better! Schedule that instruction on which most ~~#~~ number of instructions having only this one dependency depend.
eg - I_3 and I_5 will not become ready immediately after I_1, I_6 will.

- These heuristic methods come with heavy overload.

---

eg
$a = b + c$
$d = a + e$
$p = q + r$
$s = p + t$
$f = j + k$



Control Flow Limit : 2 cycles . IPC = 2.5

By doing ALL we have learnt, we can achieve 'Control Flow Limit' or 'Data Flow Limit'

Page No.
Date

- Beyond Control Flow limit → Operand prediction
  * 40% times, we use value of register used last time.
  * 60% times, we use one of the values used last 4 times
  - Need to verify prediction

VALUE PREDICTION

eg Functions could return same output for same input. ~~Keep~~ Memorize the output in hardware.

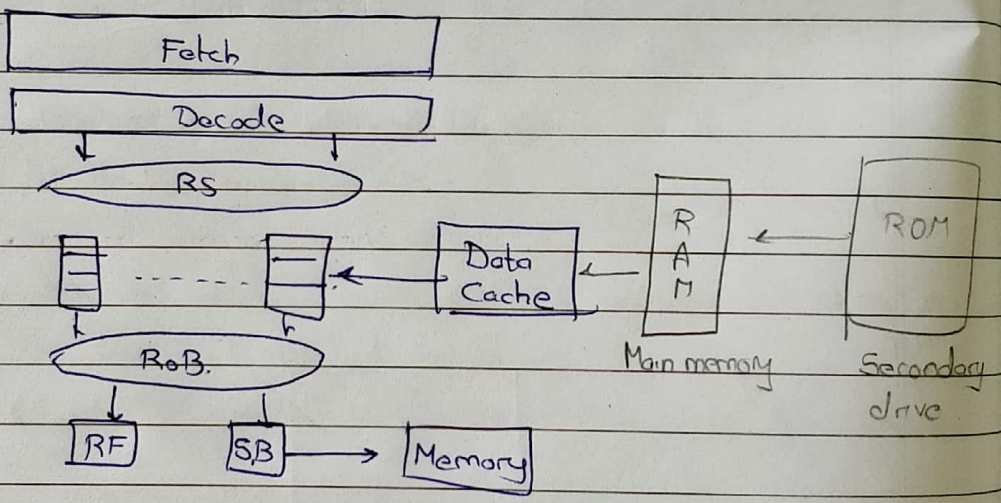
* In software, it is called memorization.

<break>

→ Memory Related Instructions.

load $r_1, (r_2)50$ // $r_1 \leftarrow M(r_2 + 50)$

store $r_5, (r_6)50$ // $M(r_6 + 5) \leftarrow r_5$



Stages of pipeline

1. Address computation :- $r_2 + 50$
   - 'Virtual Address' - Not actual physical address.
   - VA space size = $2^{\text{No. of bits in address}}$
   - VA space divided into 'pages'
   - Pages for different programs (processes) are moved to main memory from ~~virtual memory~~ ROM secondary drive before executing.
   - Page size :- High :- Wastage
     Low :- More memory (ROM) accesses required.
     ~ 4kB, to 16 kB.

ROM has virtual pages

Page No.
Date

eg 40 bit secondary (1TB), 1kB (10 bit) page size, 4GB (32 bit) main memory

∴ In ROM → | Virtual Page No. (30) | offset (10) |

In RAM → | Physical Page No. (22) | offset (10) |

Code cannot be accessed from virtual memory. It must first be brought to main memory.

We want PPN from VPN

- Lookup Table - 'Page Table'

  Virtual Page Number (ROM) → Physical Page Number (RAM)

  - Stored in RAM
  - 2^{30} entries, each containing 22 bits ☹☹ (3GB ××)
    ↳ those present in main memory.
    - Reduce size of lookup table. Use LRU policy for replacement of data.

Solution - 'Inverted Page Table': PPN (RAM) → VPN (ROM)

- 2^{22} entries, each containing 30 bits
- Much smaller (15 MB) ☺
- How to index? ☹
  We wanted to find physical page no. of virtual page no.
  - Cannot always scan all VPN entries.

Solution Better - ~~'Paged Page Table'~~ 'Translation Lookaside Buffer' (TLB)

- First time you want VPN of PPN, scan IPT (takes time)
- Enter this VPN → PPN mapping into PPT, so that you don't have to scan again.

'Cache for VPN → PPN mappings'

CPU works with 40 bit PC

3-stage pipeline

0. Steps
1. Generate address to fetch data from (I_2+50)
   ↳ virtual
2. Translate address (access TLB)
   Hence obtain ₱ main memory page number (and thus main memory address)
3. Read from main memory / data cache.

* Data and Instructions are stored in different TLBs, as they are in different pages in ROM.

If VPN entry is not present in TLB, scan IPT, ~~fetch~~ & find PPN and add entry to TLB.
- If entry is not present, TLB raises a page fault.

~~Hazard~~

11/2

→ Store instructions cannot be stored in RoB.
- They have 32 bits address + 32 bits data.
- Allotting so much space for every instruction is wasteful, because not all instructions are store.

∴ Store buffer is required

∴ When Store enters RS, entry is made in SB ~~instead of RoB~~. (ID of this entry is remembered) in its entry made in RoB

SB

Address	Data	Completed
200	504	0

RoB

Destination	RR	Valid
R0	P11	0
R5	P44	
–	ID of Store Buffer	

Store Instruction

- Valid bit of RoB is set after Store is executed.
- When store comes to top of RoB eventually, ~~execution is completed~~, make Completed = 1.
  'Completion'
- 'Retirement' - Perform 'lazy write' back to memory (as it is not in critical path of processor).
  - Only SB entries with Completed = 1 can be written back.

* SB is FIFO

- Hazard :- Load of same address following Store that hasn't been retired.
  - Cannot wait for Store to be retired (as performance ↓ a lot because of pending Load)
  - ∴ Cannot insist that all Loads and Stores are written back in program order.

- Entry of ~~Load~~ Store is made in RoB and SB when dispatched into RS.
  ∴ Before scheduling Load, match its address against all addresses in SB.
  - If no match is found, ⇒ no dependency.
    Allow Load to be scheduled
    'Load Bypassing'
  - If match is found, there is dependency. Do not schedule Load
    - Sometimes, addresses in SB are only stored as last 10 bits of actual address.
      If match is found, match might be fake due to aliasing.
      Still do not allow Load Bypassing.
    - If SB addresses are entire 32 bits, and match is found, and Store in RoB is valid, Load can obtain data directly from SB (saves one memory access)

- If Store is not entirely executed and its address is not known, we have to 'speculate' matching.
  - Easy speculation - No match.
    - Need to detect if speculation was right or wrong and correct if wrong.
    - Detection

First check address of Load against store instructions. Take no match speculation if no match is found.

'Completed'

- Take a 'no match' speculation. Execute Load.
  - 'Load Buffer'

    Execute Load as soon as address is available.

    When you executes, make a note of this address in LB.

LB
Address
300
⋮

- When Store is 'Completed' (reached top of RoB), the address in SB is matched against all addresses in LB. If match(es) is found, that Load was wrongly speculated. Correction is required.

- Correction:-

① Repeat just Load and all of its dependent instructions again. We will also require all other operands of all instructions again :(

② Treat as branch misprediction. Start fresh from that Load instruction.

- Need to store PC in LB. Or store pointer to RoB, which already has PC

- Dynamic match speculation
  - Requires PHT-like tables
  - Profile load-store behaviour.
  ① If matched before, it is likely to be matched again
    - Do not execute such load-store out of order.
  ② Make load-store pairs that are aliasing (matching) in LB.

?

6/2

* 'Total Store Order' type architecture - All stores are written back in programming order.

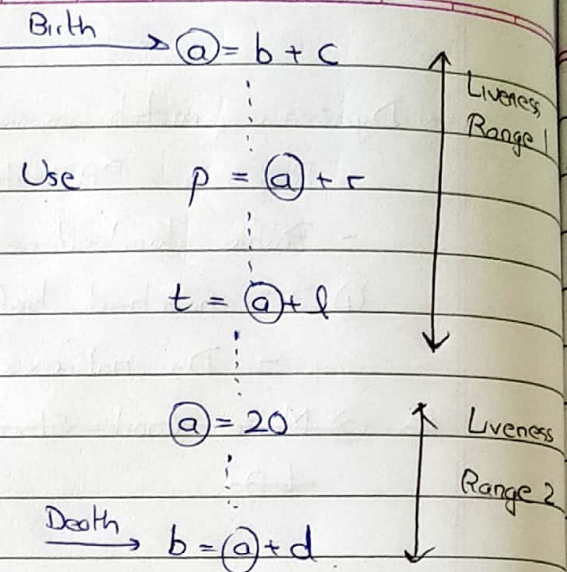
→ Load Store Aliasing

1) eg Spill Code :-
When all registers are occupied, value of some register must be replaced.

```
q = p + r        // r1 initially contains b. All registers are occupied, p is in memory
:
t = l + b
```

b must be STORED into memory/cache.
LOADed into RF
p
Later, for t = l + b, b must be LOADed back
This generates an aliasing LOAD-STORE pair for b.

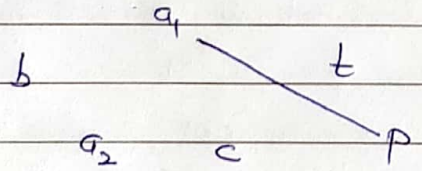
- Birth of variable



- Earlier,
  Liveness Range = No. of statements from birth till death.
- Now, 'Range Splitting'
  - Every variable can be 'assigned' only once.
    ★ 'SSA'

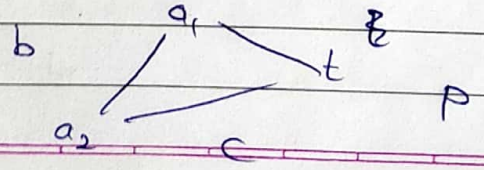
- Variables having overlapping liveness range should not use same register. Non-overlapping variables can. (eg - a_1, t, a_2)

- 'Conflict Graph' - Shows overlaps as ~~nodes~~ edges



'Colouring Problem' - No. of colours required to paint the nodes, so that adjacent nodes do not have same colour
= Minimum no. of registers required to not have to LOAD/STORE ~~to~~/from/to memory.
- If you have fewer registers than this, need to choose which register(s) to move to/from memory.

- Alternatively, 'Anticonflict graph' - No overlaps



Page No. | Date

Lower no. of registers $\propto$ 1/Spill Code

~~Fat~~ Clique $\triangleq$ Fully connected subgraph (a_1, t, a_2)
~~a~~ = Can share same register.

$\therefore$ Minimum no. of registers = No. of cliques.

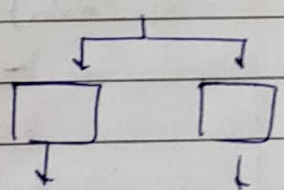
- Spill code causes aliasing of load-store. (store followed by load of same address)

2) Pointers
- Address is dynamically generated (only known at runtime)
- Can cause load-store aliasing.

eg store r_1, r_2 } if r_2 and r_6 have same value
load r_5, r_6 }

- Higher Fetch Width ($\gtrsim 4$)
- ~~Prefe~~ We will fetch say 8 instructions, which are likely to contain LOAD and STORE.
  In our attempt to bypass LOAD, this will cause accumulation of instructions.
  $\therefore$ We are forced to increase ports on memory. :(
  ↓
  ↑ size, cost, access time.

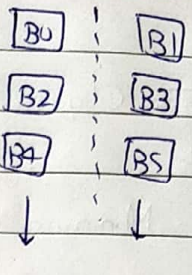
Better - Use two identical data caches
- Must be written to simultaneously
- Can read independently
  - Can perform LOADs twice as fast



'Replication'

P.T.O.

Page No.
Date



Alternate Single Cache
- Odd-numbered and even-numbered cache blocks can be read from separately.
- Even-odd division, because we usually read from consecutive addresses
- Cheaper, not as good as replication.

- 'Non-blocking Cache'

eg - Load X, Load Y executing simultaneously.
X is not in L1 cache, Y is.
Y should be read and not wait for X to come to L1 cache
'Non-blocking'

- While X is being fetched, the address 'X' is stored in 'Miss Status Holding Register'

MSHR

100

300

⋮

- Data is brought to L1 cache in 'block' granularity.
  - Multiple requests in MSHR can be serviced (possibly out of order) if they are present in same block.
    - Reduces average access time.

14/2 → Memory-related Instruction Handling

1. Non-blocking D-cache (L1) :- MSHR
2. Speculative execution of loads
3. Multiported cache
4. Use of victim buffer
5. Prefetching.

Page No.
Date

→ Pre-fetching
... of next likely block of data

- Can be done by software or hardware.
- Data can be prefetched into memory (L1 cache) or prefetch buffer.
  - Whenever there is a hit on block in prefetch buffer, it is ~~swapped~~ moved to L1 cache before being used.

- Many ISAs provide a 'Prefetch (Block #)' instruction.
  - Prefetches into L1 D-cache.

- We may not know which block # needs to be fetched.
  - If you ~~fe~~ prefetch a block and never use it, it is bad because it replaced another block that could have been used.
  - We cannot know which block # to be fetched in unstructured program.
    eg - Can be useful for dense matrix computation
    eg - ~~Some~~ Prefetched data is useless for sparse matrix multiplication (most elements zero.)
    eg - Mostly used in loops.
- 'Cache pollution' - Bringing in useless blocks into cache.
  - Can occur because of both hardware and software prefetching

- When to prefetch? (Total time to fetch ~ 150 cycles)
  - Too early - Can be replaced before use. (mostly useless)
  - Timely - ☺ Fetch only as early as is needed to make data available when needed.
  - Late - Data does not reach on time.
    Cache miss is registered.
    But still saves some cycles as opposed to actual cache miss.

Fetching unnecessary instructions into prefetch buffer is less bad

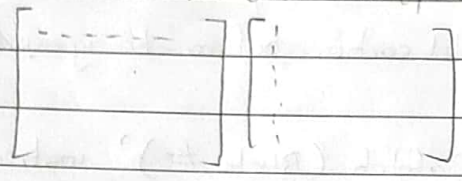
- Where to prefetch ?

1) Software can only prefetch to ~~softwa~~ L1 D-cache

2) Hardware can prefetch to cache or prefetch buffer.

Software

eg 100 x 100 matrix multiplication



One computation :- 100 multiplication followed by 99 addition

Say :- Operation of row 1 on all columns takes 100 cycles

Say :- Fetching one row from memory takes 200 cycles.

∴ Fetching next row : Too early

∴ Every row 2 prefetches next-to-next row : Timely.

(Non-prefetched rows will cause cache miss)

- Cache misses are served preferentially to prefetching stuff

Hardware

eg You are fetching from an array : A(1), A(5), A(9) ....

'Stride Prefetcher' – Detect pattern :- 4 elements apart.

– All these fetches are coming from same load.

load $r_1, (r_2)\, r_3$

$r_3 = r_3 + 16$

– Hardware maintains load PC V/s Address V/s Difference

PC	Address	'Stride' = Difference	Confidence
500	132	16	1
	116	16	0
	100	–	0

When stride remains same for some predecided no. of executions make Confidence = 1, and start speculating which address will be needed next.

- Per-hit fetch - (↑ energy, BW, performance)
  Whenever there is a hit in cache, prefetch the block that contains the earliest speculated data that is not already in the current block of cache.
  ∴ - Can create traffic, because you are checking if next required block is fetched at every hit (hit rate is high)

- Per-miss fetch :-
  Whenever there is a miss in cache, prefetch the "next" block
  - Must have one miss, unlike per-hit prefetch.

- For timely fetch, we should not always fetch the very next block.
  'Prefetch depth' :- How many blocks away are you fetching your block from?
  'Prefetch degree' :- How many blocks to prefetch at a time.

- Stride prefetcher is certainly useful for arrays
  - Also useful for linked list.

- Different patterns :- A(1), A(5), A(9), A(29), A(35), A(35)
  (4, 4, 4, 2)

18/2

DATA PREDICTION

Going beyond data flow limit.

- Can advance future instructions to current cycle.
- Thus, resource utilization can be increased to beyond 30%.

- Need to validate predicted value.
- 'No-validation'
  - We are sure about prediction. We have checked its history and the value will not change
    'Instruction Reuse'
  - Most values do not change for a long time
    eg - Array operation (B+D) :- B is constant.
    eg - Most of the comparison of values is against zero
  - Sometimes, there is a pattern in the values over iterations
    eg: i, i+4, i+8 ...

eg - a = b*16
If b does not change, neither will a.

eg - Factorial (n)
As you iteratively calculate, store in a table

PC	Argument	Result
100	3	6
100	5	120
⋮	⋮	⋮

★ This should not be done when operating on global variables
- Not advantageous as they can have too many values.

* Local variables are the ones stored in stack and accessed using SP.

- Such tabulation of function results need not be validated.
- Tabulation can be done for functions or even single (critical) instructions.

- Predictions must always be validated.

LAST VALUE PREDICTION

- Table for results produced

PC	Value	~~Last Value~~ Last Value	Confidence Counter
100	786		0001
⋮	⋮		00001

- Instructions dependent on this I_{100} can fetch the result from here.

- Alternatively, first column could be ~~value~~ PC of dependent instruction and it will map against its operand value ~~used~~ to be used.

- When same result is obtained ~~same predecided no. of times~~, ~~make~~ Confidence Counter += 1.
  - If CC ≥ x, that value can be used
    - Used value will be flagged as 'speculative'
  - If some other value is obtained, make CC = 0.
  - New result can be matched with table value as soon as execution is complete

STRIDE BASED PREDICTOR

- Sometimes, values are in AP. Modify table.

PC	Value	Stride	CC
100	786	16	01
⋮	⋮	⋮	⋮

- If same stride occurs again, CC += 1.

eg:- Instruction :- $r_1 = r_1 + 16$.

- If values remain constant, stride remains 0 ✓✓

eg 30, 40, 50, M M M M, 30, 40, 50, M M M

Stride predictor fails to recognize pattern.

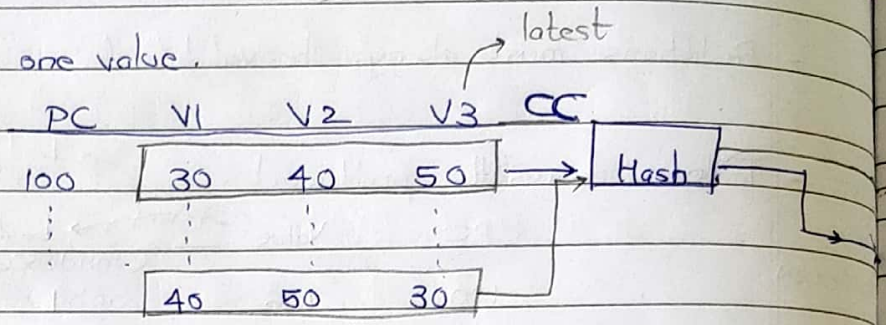
P.T.O.

Page No.
Date

Finite Context Method (FCM)

- Profile more than one value.

Value Table :-



- Output of hash function gives index within prediction table.
- If predicted value is correct, CC for that pattern += 1
  incorrect = 0

* eg - Fetching 8 instructions per cycle :- IPC < 3
  ⇒ Poor utilization of resources ☹
  (~ 70% resources idle at all times)
- Poor utilization because of :- 1) Dependence of instructions
  2) Constraint to write back in program order
  3) Cache misses

* 'Software Memorization'
- Even software maintains past values to predict future values for functions.

* To increase performance, we can increase independence of instructions.

Simultaneously fetch 8 instructions from more than one program, which will be inherently independent.

8 = 4 + 4 or 8 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

P.T.O

Page No.
Date

Single Pipeline.

Prediction Table.
40
30

Cycle	IF	ID	EX	Mem	WB
1	I_1^1				
2	I_1^2	I_1^1			
3	I_1^3	I_1^2	I_1^1		

- Each program perceives multicycle execution.
  - No need for branch predictors, forwarding mechanism, etc ☺
- Higher throughput ☺.
- But lower performance of single program ☹.

- Superscalar can have more such pipelines.
  eg - Fetching 8 instructions per cycle :- IPC ≈ 6. ☺.

☆ Register conflict ?

• How to improve performance of even a single program?

eg :- $\begin{bmatrix} - \end{bmatrix}_A \times \begin{bmatrix} \quad \end{bmatrix}_B = \begin{bmatrix} \otimes \otimes \end{bmatrix}_C$

- Computation of every element in C is independent ☺
- 'Threads' ≜ Such independent streams within a single program.
  - Can be executed in parallel.

7/3

- If we execute threads of one program parallelly, we can get ↓ execution time along with ↑ throughput

- Executing a process requires the correct PC, stack, architecture registers, code segment, data segment, file descriptors, state

'Memory Image'

- State = { Active, Ready, Wait }
  - Active → Ready transition :- (time expired)
    Store PC, registers, stacks, etc.
  - When there is a long-duration wait event,
    Active → Wait
    (eg - Page Fault)
  - When wait event is over
    Wait → Ready
    (eg - Page Fault is serviced)

- Forking (Making new)
  - ~~Copying~~ memory image takes time ∴
    ∴ Make threads

- Threads of same process share code segment, data segment.
  - Different PC, RF, stack.

- Switching from one thread to another
  eg - When long-duration wait occurs, like L2 or L3 cache miss.

- Store CPU context, retrieve new CPU context, execute.
  CPU context → PC, RF, Stack.

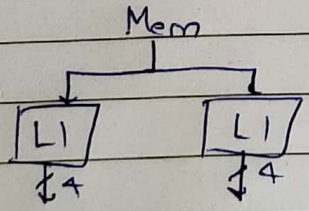
- When same thread runs for a long-time, ⇒ poor resource utilization ∴
  * Switching threads only for long-duration wait ≙ Coarse grain switching

- Fetch one instruction from each of 8 threads at a time
  - Better utilization of resources, ⇒ IPC close to 8 ☺
  - Better identification of independent instructions
- We need to have 8 different CPU contexts (8 PCs, 8 ARFs,

- We are fetching instructions from different locations
  ⇒ Memory must have 8 ports (although of 1/8th size)
  ∴ ☹
  * Increasing no. of ports also increases static power consumption

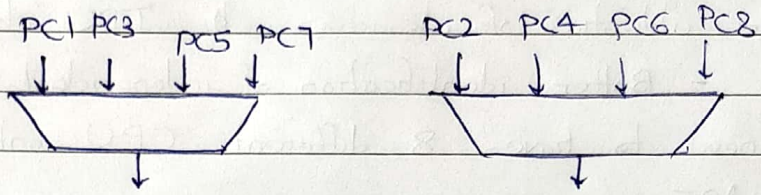
- 'Banking' :- ~~Say memory has 2~~
  Memory can have 'n' partitions and 'n' ports, each port can only give from a corresponding partition.
  - Threads need not be running at different partitions.
  ∴ ☺

- ∴ Choose to have only 2 ports, each of which can fetch 4 contiguous instructions from 2 different locations



- 'Round Robin'
  Fetch 4 instructions each of T1, T2, then T3, T4, ... per cycle.

- This has poorer identification of independent instructions, but is still fine.



∴ - Because each thread has PC at different locations that may map to same location in L1/L2/L3 cache, we have a huge problem of cache misses.
  * All threads are sharing the same cache hierarchy.
  ∴ L1 hit rate ↓.
  ∴ Need to increase associativity of L1 cache
    - Because increasing associativity can increase access time of cache, we need to 'pipeline' L1 cache
      - Stage 1 :- Tag Matching
        Stage 2 :- Reading