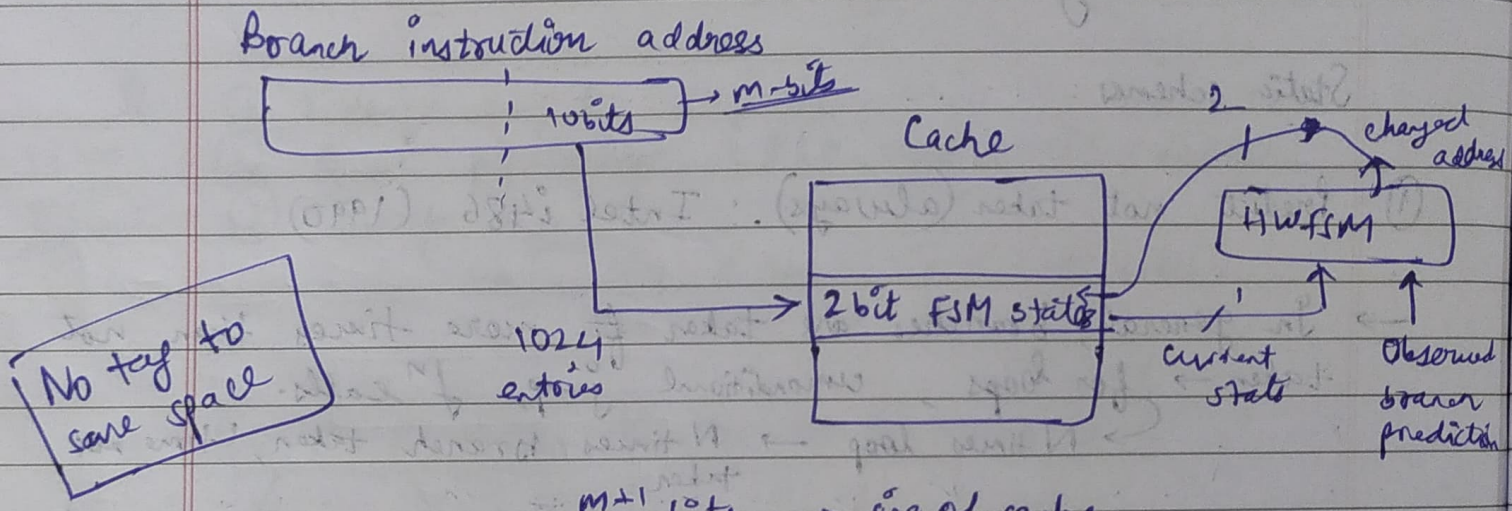
* Dynamic Branch Predictions:

- Observe branch outcomes, predict based on observed outcome
- learn from mistakes



2^{M+1} bits → size of cache

26 bit FSM state

Prediction = MSB of state

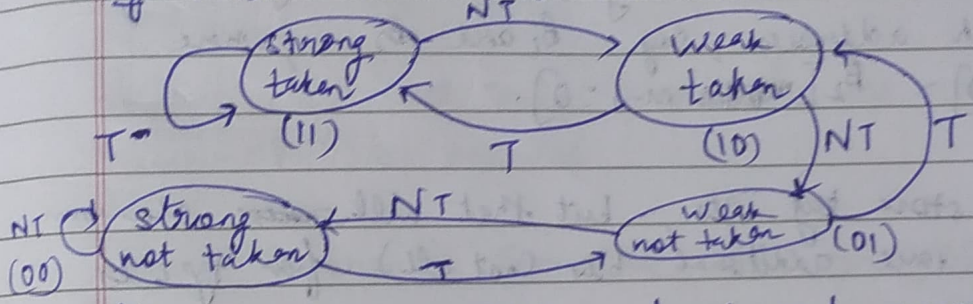
11 → strongly taken
10 → weakly taken
01 → weakly not taken
00 → strongly not taken

This cache has different names
1. Branch Prediction buffer
2. Pattern history table

① Local 1-level branch predictor:

Branch predictor FSM.

eg: 2-bit FSM has two states ← can also be 3-bit, 4-bit etc.



NT = Not taken
T = taken.

MSB → decides prediction of branch predictor

MSB=1 ⇔ taken predicted

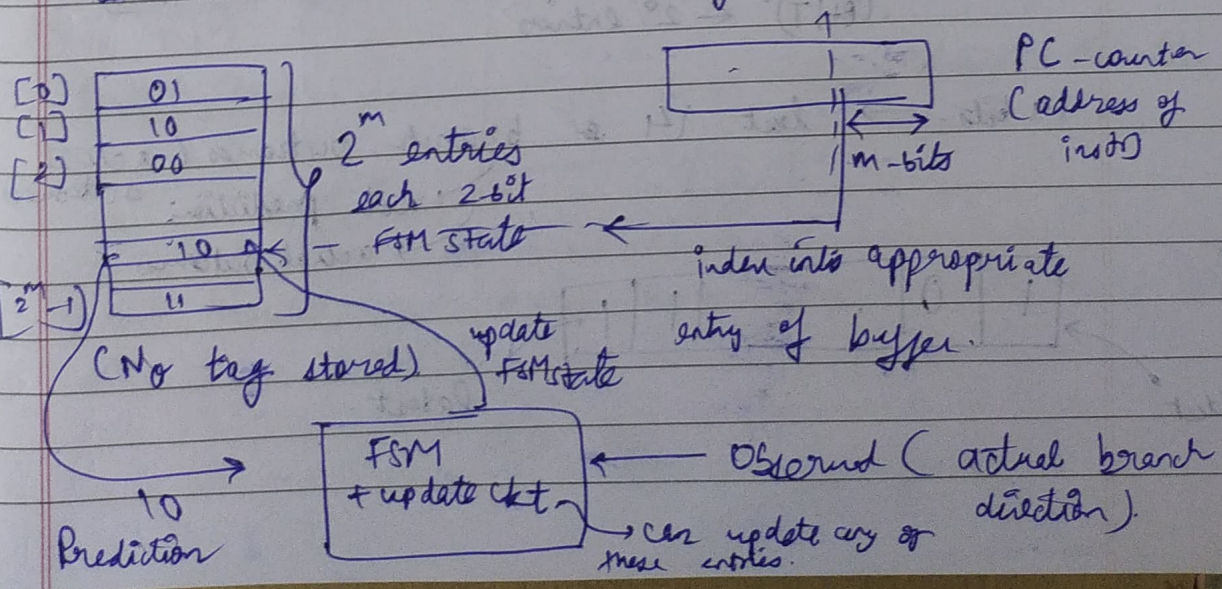
MSB=0 ⇔ not taken predicted

Moore machine → output is just a function of current state.

There are multiple branch instructions in a program. We maintain a buffer to store the state of FSM for each of these branch instructions.

Branch prediction buffer / Pattern history table (confusing names)
(BPB) (PHT).

The FSM can initialize to anything based on branch address.

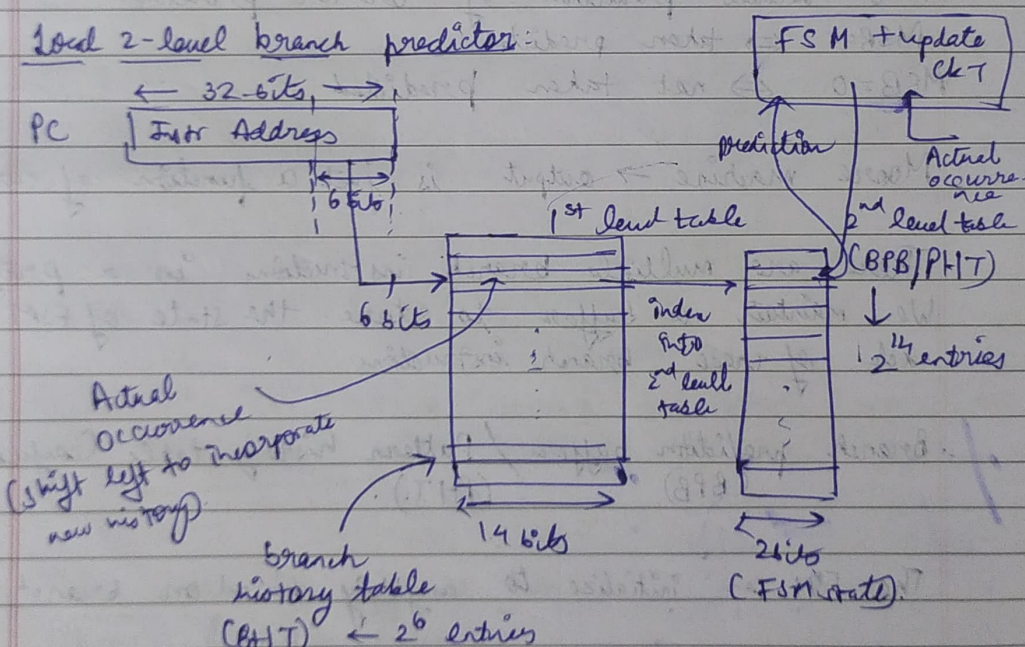


Branch prediction buffer supports only distinct 2^m branch instructions.

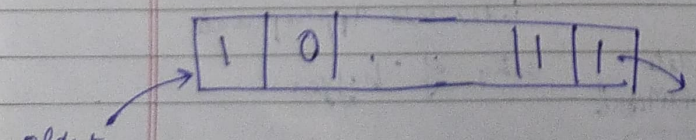
Issue: the index bits might cause aliasing among branch addresses, say B_1 and B_2, where $B_1[m-1:0] = B_2[m-1:0]$.

Sol^n: can store tag bits, but that will slow down. Can store some additional bits (not all) for tag bits.

② Local 2-level branch predictor:



14 bits → last 14 branch outcomes (actual outcomes)
↳ not predictions
↳ actual outcomes



Initial branch

Possible that B_1, B_2 (with

14 bits change instruction is

Also 2 branch same predict

Eg: B_1: if C

B_2: if C

B_3: if

Observation: if
⇒ Behaviour of

* Correlating bra

① Global (m,n):

Global for all branches

or

Initial branch history table → maybe all 0s or all 1s

Possible that 4-bit history may be of two branch instruction B_1, B_2 (with 6-bits same).

4 bits change every time the corresponding branch instruction is executed.

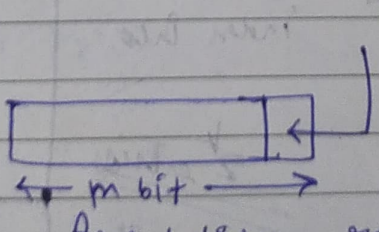
Also 2 branch instructions with same history will have same prediction.

Eg:
```
B1: if (aa==2)
        aa=0;

B2: if (bb==2);
        bb=0;

B3: if (aa!=bb) {  }
```

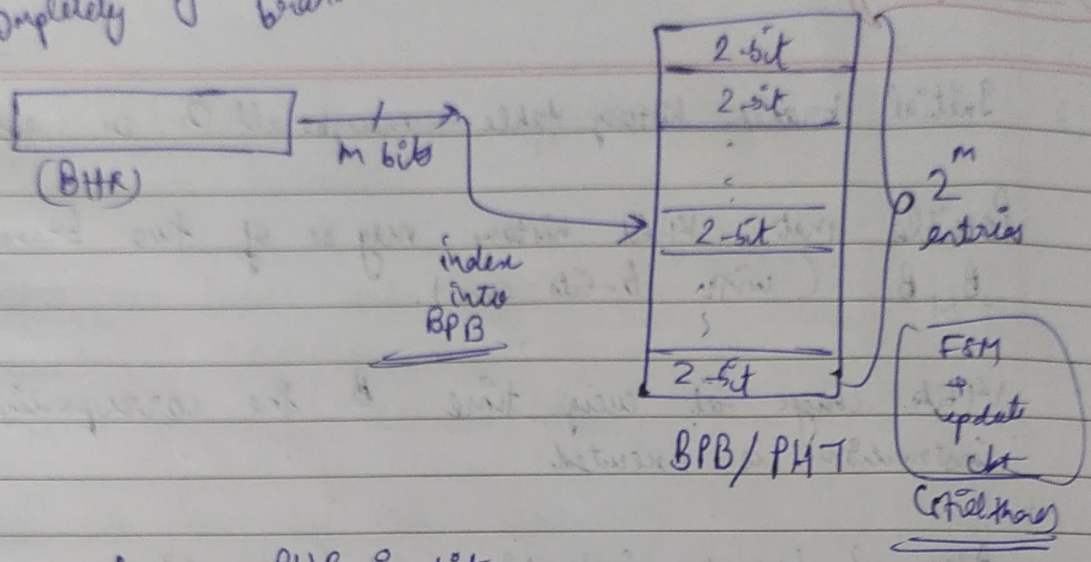
Observation: if B_1 is taken and B_2 is taken ⇒ B_3 is not taken.
⇒ Behaviour of B_3 is correlated to behaviour of B_1 and B_2.

* Correlating branch predictors:

① Global (m,n):

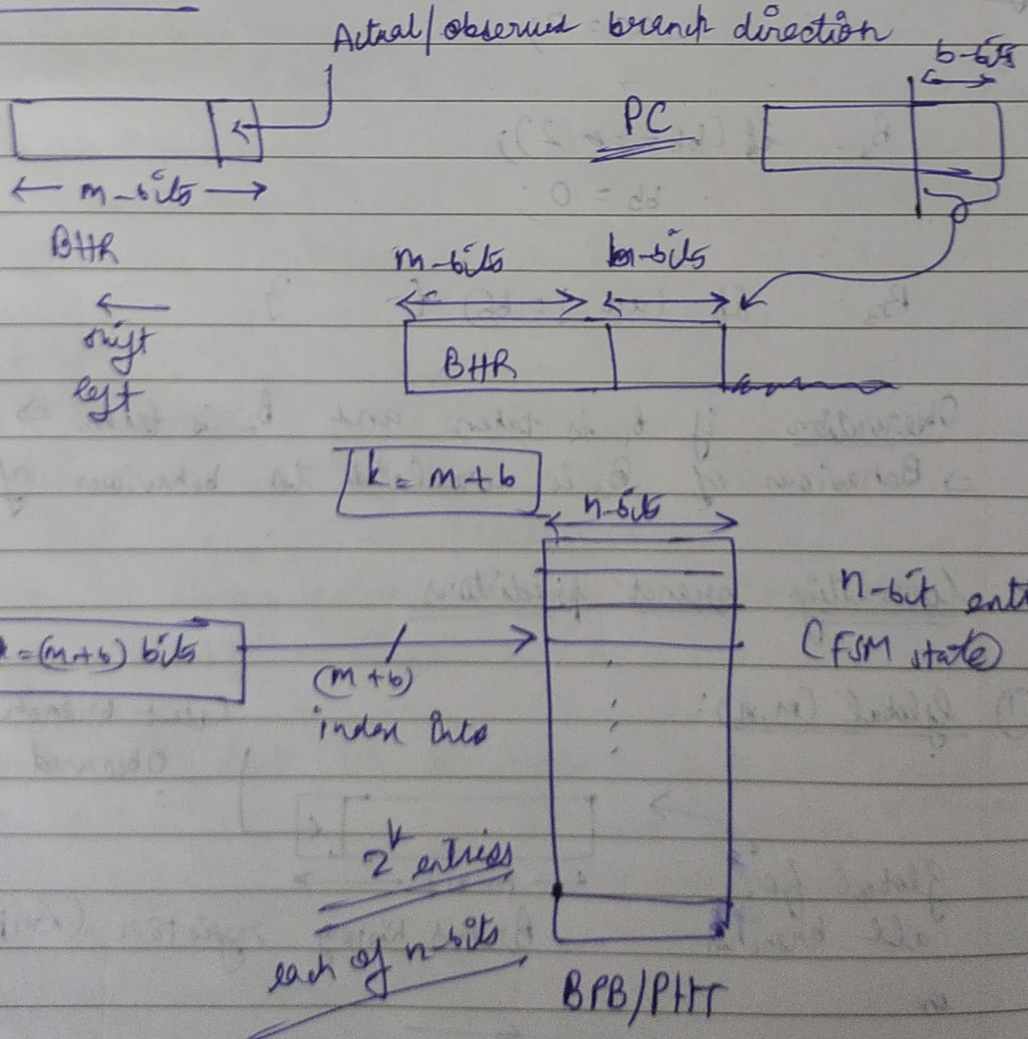


(Completely ignore current branch instruction address)



m = size of BHR in bits
n = no of bits in FSM state (here, n=2).

② Hybrid (k, n):-



Eg: $k=6$, $m=4$, $b=2$

0	1	1	0	1	0	= 26

↳ FSM state [26]

Say k is fixed.

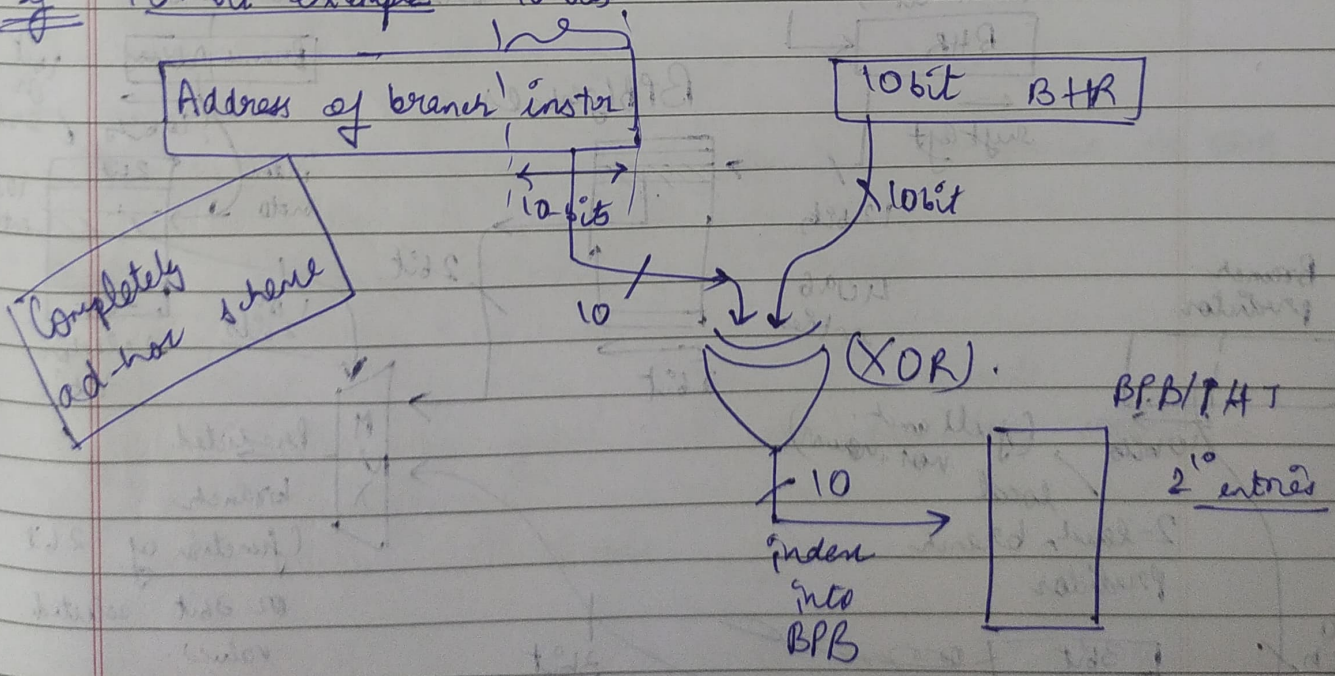
$b \uparrow (\Rightarrow m \downarrow)$ → aliasing probability decreases.
→ less global history can be preserved. (BHR is smaller)

Hybrid (k,n) ≡ Hybrid (k, m, n) ≡ Smith's algorithm (gshare)

③ Hybrid correlating: ~~gselect~~ [Macfarling 1993] :-
→ Insight: LSB of branch address & BHR bit strings are not correlated.

~~Eg: 10-bit example~~

Eg: 10-bit example:



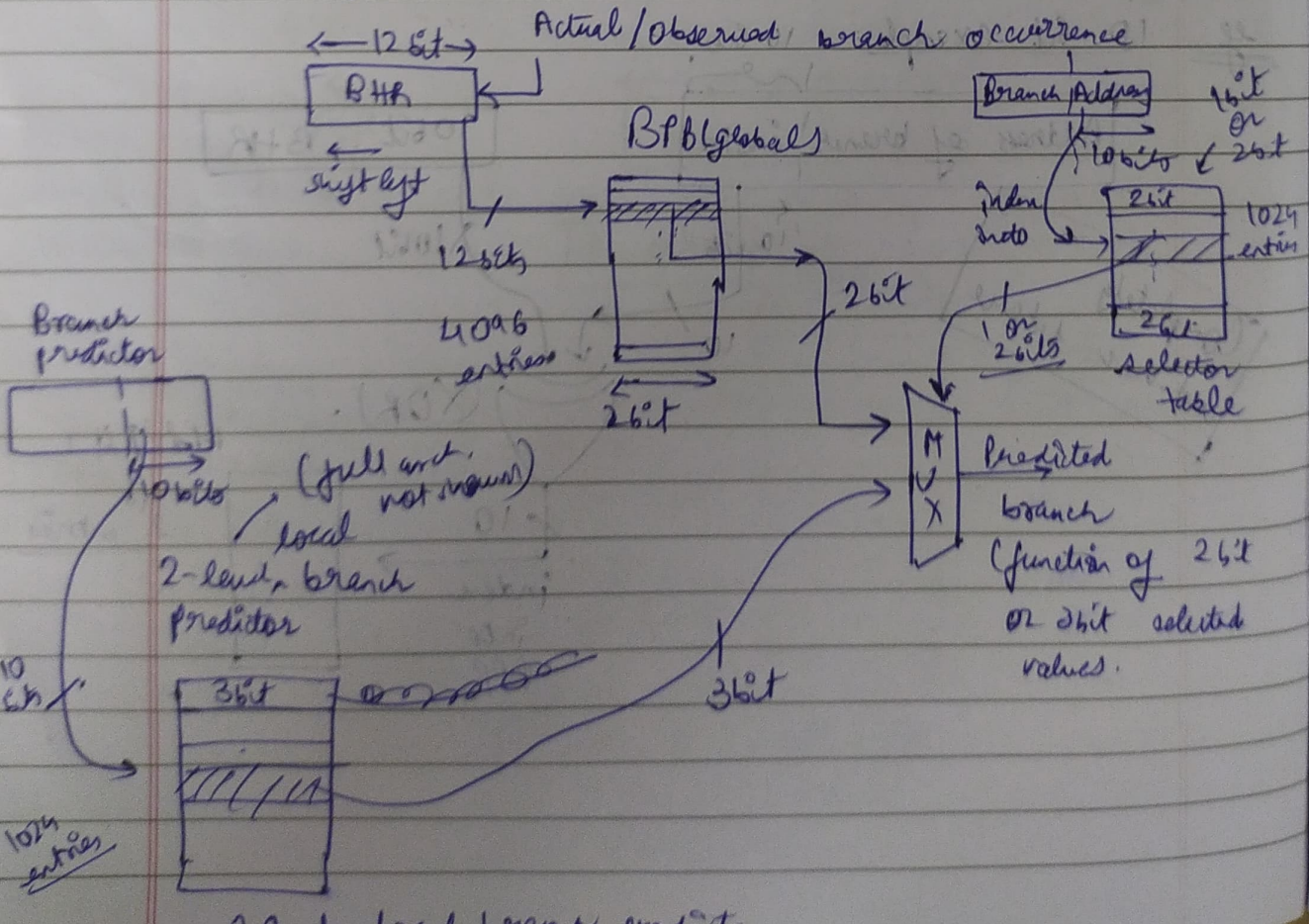
(4) Tournament Predictor :-

Idea :

(1) Have both local and global prediction
(2) Local prediction : 2-level local predictor
(3) Dynamically (during runtime), by observing the recent performance (accuracy of prediction), select either global / local predictor. (Change predictor based on ~~performance~~ performance).

→ Change predictor type after 2 consecutive mispredictions.

Example :- BHR → 12-bits, global predictor, BPB : 4096 entries, each 2 bit



2 level local branch predictor

F D X W
F D

We can allow both local and global predictors to update always. But we will use the selector table and choose prediction from appropriate predictor.

→ L(1) < L(2) < L(3) < L(4).
P(1) → h[0 : L(1)-1]
P(2) → h[0 : L(2)-1]
P(3) → h[0 : L(3)-1]
P(4) → h[0 : L(4)-1].

→ hash :→ only some bits of pc.

* Tomasulo's algorithm:

1 reservation station for each adder/multiplier.

① fadd f1, f2, f3 RAW hazard
③ fadd f4, f1, f2

① : say EX finishes in 4th cycle 5. Value pushed to CDB at end of 5th cycle.
① : WB stage completes in 6th cycle.

② : Cannot start EX before 7th cycle.
→ just read from CDB into reservation station in 6th cycle.
→ EX in 7th cycle.

① fadd f1, f2, f3
③ fadd f1, f4, f2
③ must write f1 after ①.

WAW hazard
① not allowed to execute or not allowed to update reg file.

Registers

hardware uses them internally → buffers (internal, non-architectural)

→ architectural registers (can be used by programmers).

Broadcast values on CDB. → Every unit is hearing CDB.

Network on chip → use LAN type protocols on-chips to communicate b/w units.

→ Read TB before speculation ←

1 Reservation station is present for 1 adder/multiplier unit.

WAW and WAR hazards can be removed by register re-naming. But RAW hazards cannot be removed by this technique.

Register re-naming can be done by the compiler as well. (software techniques).

Branch predictive speculative execution is used in deep pipelines and EX stage takes multiple cycles and even ID stage takes multiple cycles.
→ branch predictors are extremely accurate. Hence this allows 10-20 instructions to complete execution before knowing correct branch decision.

Address calculation is always in-order. (Simplify hazards through memory. The addresses are pushed to load buffer and store buffer only after they are resolved.

Reservation stations are intelligent to detect RAW hazards.
No direct inter-connection b/w units.
CDB does not hold values, it transmits values.

Reservation station reads value into CDB in clock cycle i and updates itself.
It is passed to adder/multiplier in clock cycle (i+1).

1 reservation units holds information for 1 instruction for 1 execution unit.
→ Not the case for modern processors like Intel i7.

~~Say values are available on CDB in 8th clock cycle. The reservation unit~~

Say execution unit writes to CDB at end of 8th clock cycle. The value is still available during 9th clock cycle. The reservation station reads and update value during 9th clock cycle. The corresponding adder/multiplier for this reservation station starts execution in 10th clock cycle.

→ Reservation station of a unit is not allowed to update till execution of this unit completes. It stays in busy state till then. This is needed because the internal data structures need to be maintained (for other units) till execution completes.
~~Once execution~~

~~Say we have 2 reservation stations for adder.~~

Load/store buffer has only 1 memory unit. Execute in-order.

DMA controller can take hold of memory bus and not allow CPU to access memory.

CDB has to be wide enough because multiple units can write to CDB within same clock cycles.

We can reorder loads. But we cannot reorder "stores and stores" or "stores and loads".

When issuing a load to memory, we first check if there is a store before this load with same effective address as that of load. If there is such instruction, we first allow store to complete.

~~Once functional unit completes, it update its reservation station and CDB in simultaneously.~~

Once functional unit completes, it updates all other reservation stations, not its own reservation station. However, the status of execution is stored in parent reservation stations. This status is used by other reservation ~~stations~~ stations.

Trap: Software interrupt
→ say a process calls a system call. This will raise software interrupt / trap.

Interrupt: May/May not be error.
→ Timer interrupt

Exceptions: Generated internally by hardware
→ Div by 0
→ Invalid op-code.

If exception is caused by an instruction that must not have been executed (due to speculation), we ignore this exception.

Lasso

8
17

Lasso

fdiv
store

⇒ This needs t

There is a
This buffer

BTB

Branch address
(of instr)
add

It is helpful
not in ins

Reorder buffer → s
this). Co

Full speculati

Precise except
the process
did specula
an exception
an excepti

Lasso

8
17

Lasso

fdiv → creates exception
store → say this completes before fdiv.

⇒ This needs to be handled by special control ckt.

There is a buffer b/w L3 cache and memory. This buffer brings in next cache block as well.

BTB

Branch address (of instr)	Branch Target address	Instruction at target address
addr$_1$	addr$_2$	add A_1, A_2, A_3

It is helpful if instruction at address addr$_2$ is not in instruction cache.

8 Reorder buffer → stores values in program order (logic ensures this). Commit happens in program order.

Full speculation → not implemented till now.

Precise exceptions → Exceptions are not raised if the processor executes an instruction that it did speculatively. (Do not raise an exception if there would not have been an exception during perfect in order execution)

Speculative execution wastes a lot of energy. You might be executing instructions assuming wrong branch prediction.