Lecture 7-9 August 23 and 25, 2021
Computer Architecture and Organization-II
Biplab K Sikdar

Pipeline Control Hazard

0.1 Instruction Pipeline

In this discussion, we consider three structures of instruction pipeline.

1. 4-stage instruction pipeline:

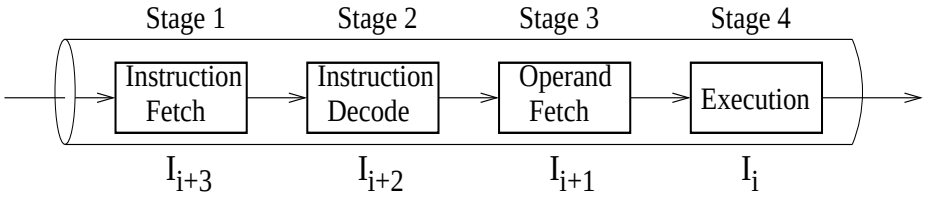


Figure 1: 4-stage instruction Pipeline

2. 5-stage instruction pipeline:

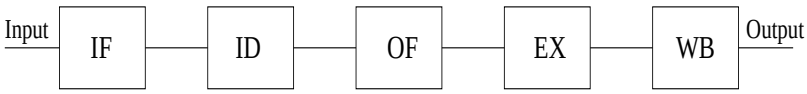


Figure 2: Instruction pipeline

3. 5-stage DLX-like instruction pipeline:

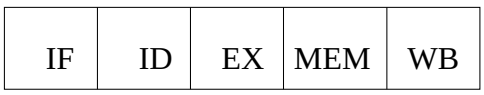


Figure 3: 5-stage DLX-like instruction pipeline

0.2 Control Hazards

Delay between instruction fetch and decision on control flow (branch/jump) prevents next instruction from execution. (control hazards, Figure 4)

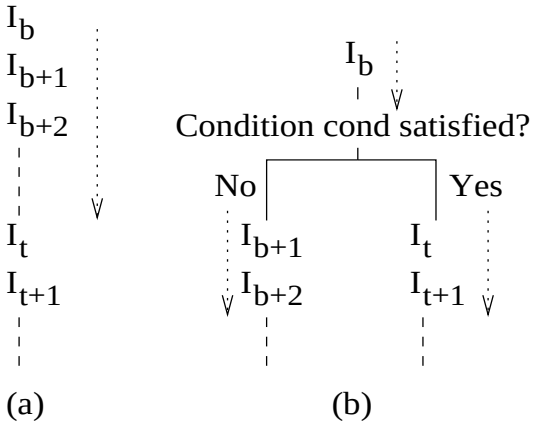


Figure 4: Branch control

Let consider execution of the following sequence of instructions.

I_1
I_2
I_3
I_4
I_5
I_6
I_7
I_8
$\vdots$

I_3 is an unconditional branch (say, Jump to I_8).

Execution of this sets PC to point to I_8.

Snapshot of execution is shown in Figure 5.

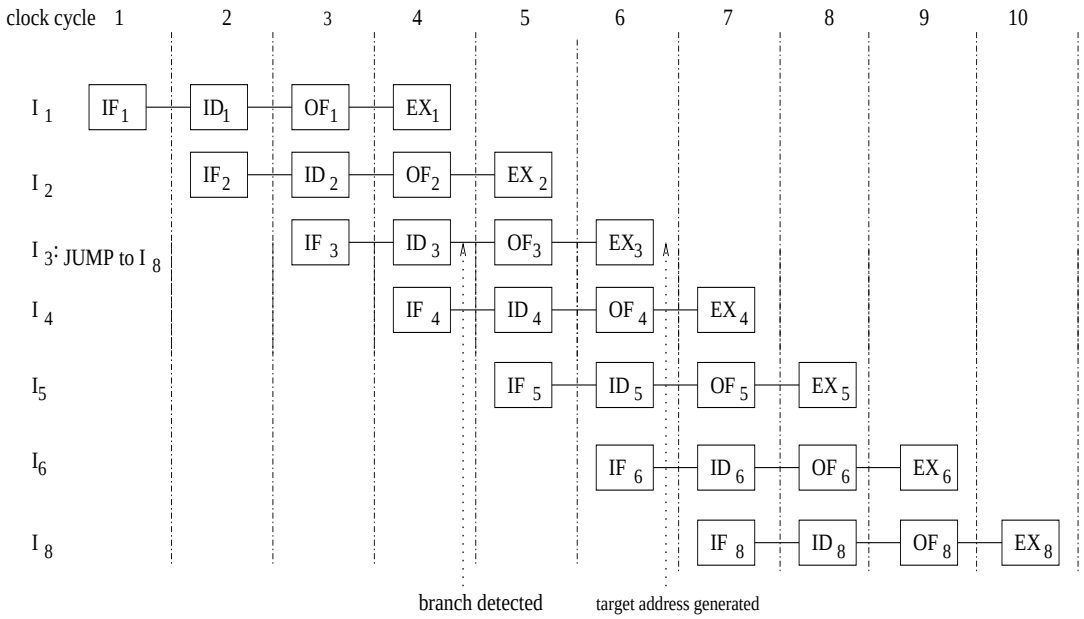


Figure 5: Control hazard

In clock cycle 4, PC content points to address of I_5.

At this stage I_3 is identified as branch and next instruction to be fetched is I_8.

That is, PC content is to be modified to point to address of instruction I_8.

At the end of cycle 6 (EX phase of I_3), branch target (address of I_8) can be known.

In the mean time, instructions I_5 and I_6 enter in pipeline.

I_4, I_5 and I_6 in the pipeline may affect desired output of program in execution.

This results in a *control hazard* in the pipeline.

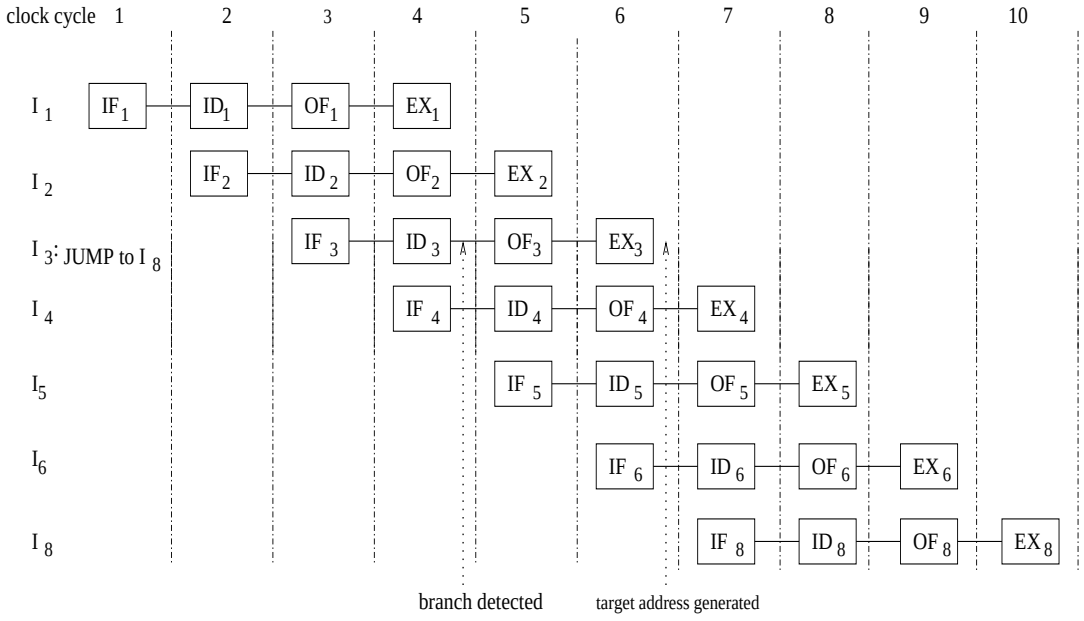
There are two aspects of the control hazard

1. Undesirable instructions in pipe affect desired output of program in execution

2. Entry of undesirable instructions in pipe wastes some clock cycles (here, 3)

Branch can yield 10%-30% performance loss in pipeline. Solutions are

1. Introduction of stalls
2. Reordering of instructions etc.



0.3 Stalls To Avoid Control Hazards

Introduction of two stalls are shown in Figure 6.

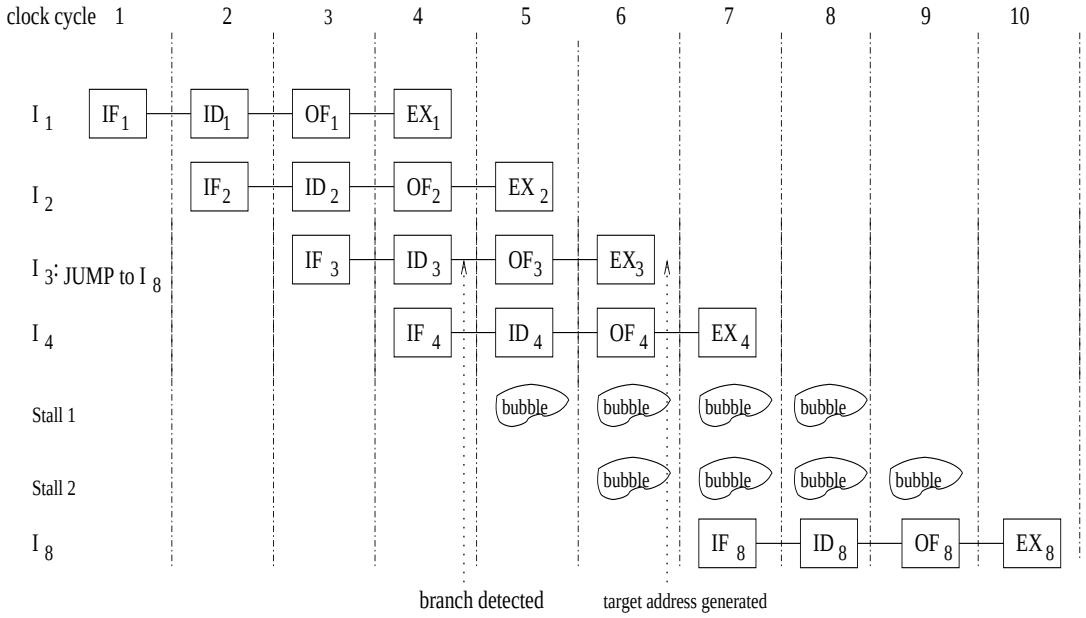


Figure 6: Stalls to avoid control hazards

These are to block instructions I_5 and I_6 from entering into pipe.

However, I_4 is still in the pipeline.

-Prior to identification of I_3 as branch (end of clock cycle 3), I_4 is fetched.

We need to take special measure to nullify effect of I_4.

-That is, flush out the effect of I_4.

Here, stalls to avoid control hazard effectively wastes 3 clock cycles.

0.3.1 Reduce stalls

Hardware solution: Early computation of target address.

Target address is computed as early as ID phase (Figure 7).

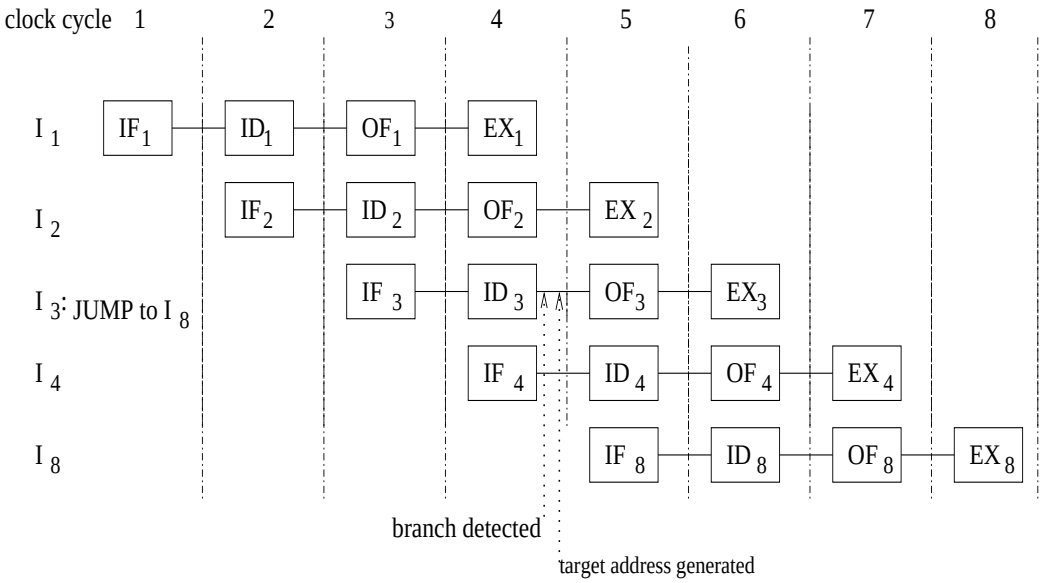


Figure 7: Early computation of target address to avoid stalls

Number of stalls required is 2 in Figure 5.

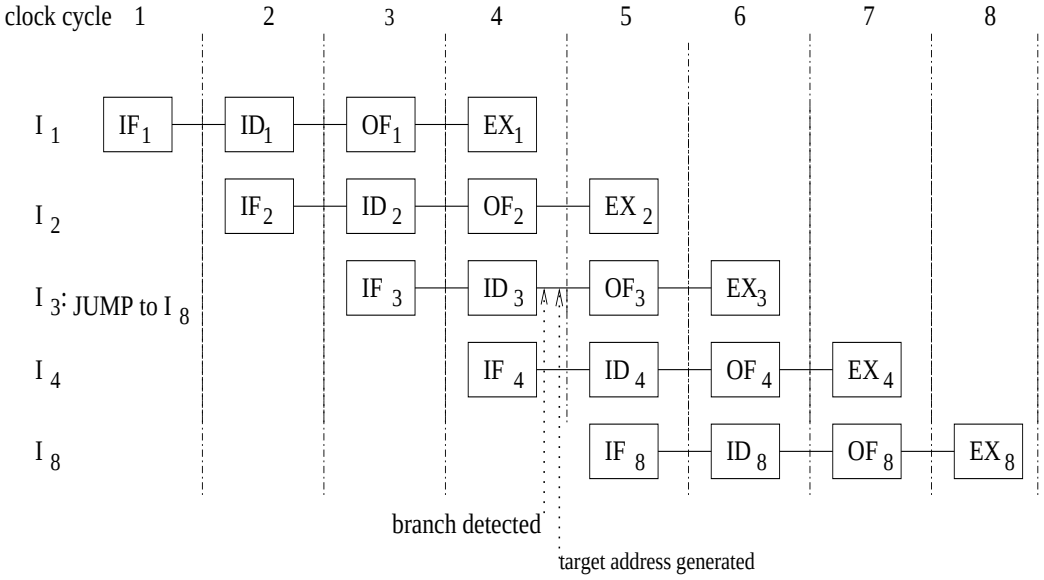
Early computation of absolute address of target needs additional logic (ALU).

As ALU may be busy to perform execution of other instruction.

Such a logic is costly and adds delay in pipeline path.

0.3.2 Flush out

Flush out instruction I_{b+1} (I_4).



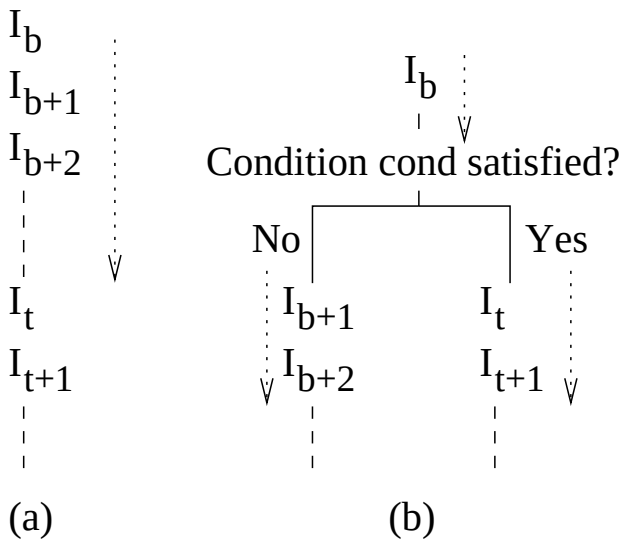
Flush out of I_4 : Realized with with NOP.

While decoding I_4 (clock cycle 5 of Figure 7) opcode of I_4 is replaced by NOP.

Anyway, one clock cycle waste due to I_4 cannot be avoided.

0.4 Conditional Branch Hazards

Control flow



Control hazards for conditional branch can cause a higher performance loss.

In 5-stage pipeline of Figure 8, I_b is a conditional branch and I_t is target instruction.

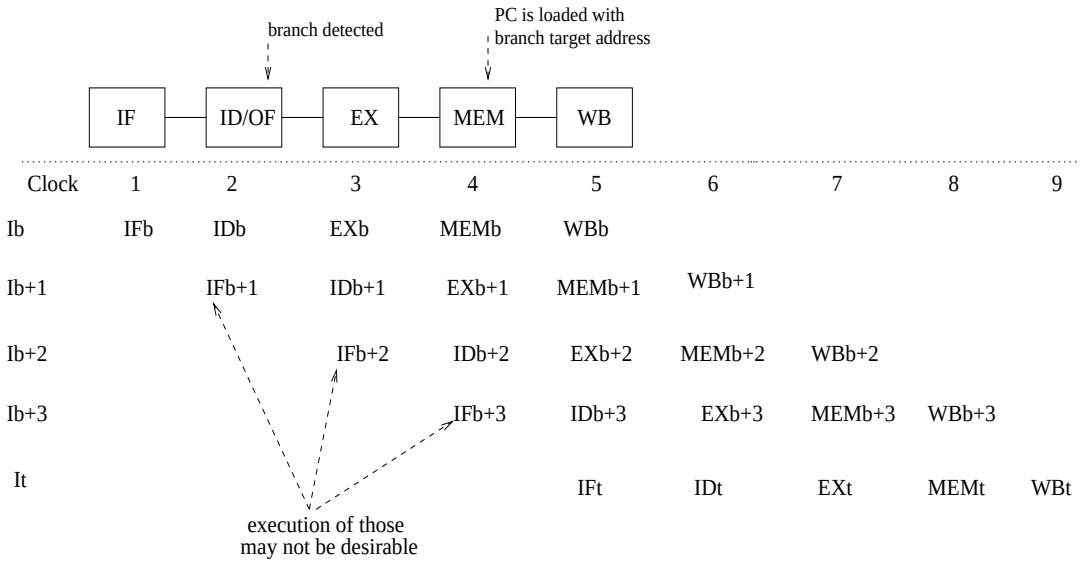


Figure 8: Control hazard in 5-stage pipeline

I_{b+1}, I_{b+2}, and I_{b+3} are in pipe prior to decision taken on branch target I_t.

If branch condition is satisfied, then execution of I_{b+1}, I_{b+2}, and I_{b+3} is not desirable.

Target address is set in MEM stage (say).

Pipeline control can block I_{b+2}, and I_{b+3} and enter bubble/stall once I_b is decoded.

But still I_{b+1} is in the pipe. It is essentially a stall called *delay slot*.

Blocking of I_{b+2}, and I_{b+3} is a correct move if branch is taken.

For untaken branch it affects efficiency as after I_b/I_{b+1}, I_{b+2}/I_{b+3} are to be executed.

That is, introduction of stalls for I_{b+2} and I_{b+3} wastes pipeline cycles.

It is different than that of unconditional branch.

Performance of a pipeline with stalls

$$\text{speedup} = \frac{Pipeline\ depth\ (number\ of\ stages)}{(1+average\ pipeline\ stall\ cycles\ for\ branch\ instructions)}$$

$$= \frac{Pipeline\ depth\ (number\ of\ stages)}{1+branch\ frequency \times branch\ penalty\ (average\ number\ of\ stall\ cycles\ per\ branch)}.$$

Even with frequent conditional branch, efficiency of pipeline can be improved through

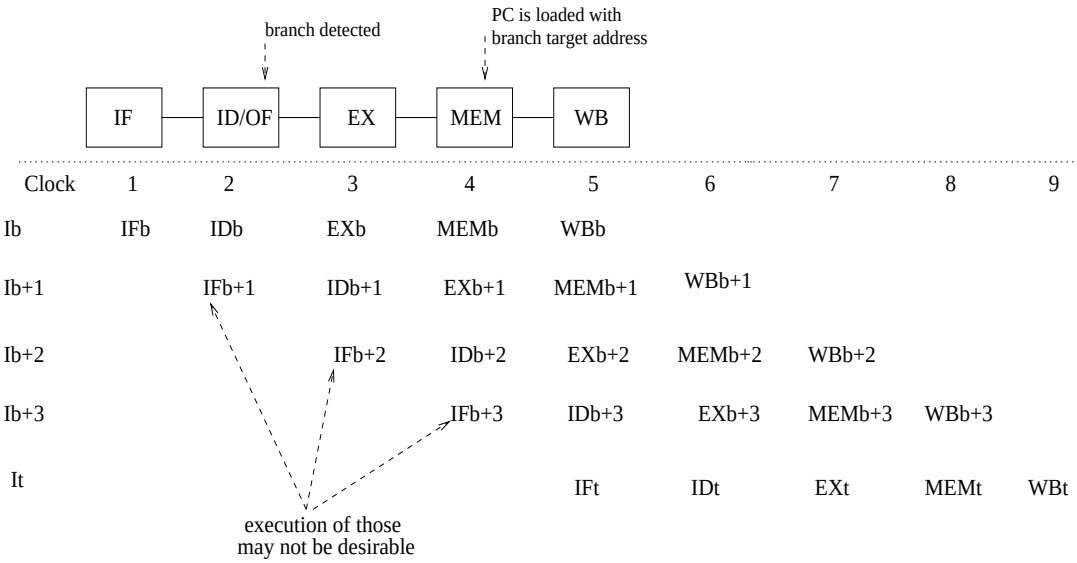
1. Reduction of number of stall cycles, and
2. Scheduling of instructions in branch delay slots

0.4.1 Reduction of number of stall cycles

It specifies that branch target I_t is to be identified at an early stage, possibly at ID.

For an unconditional branch this may be comparatively easier.

For conditional branch, even if address of I_t is computed early, early decision on branch condition (satisfied/not-satisfied) is difficult to realize.



Best choice: Settle in ID-phase whether branch will be taken or not-taken.

However, early computation of status of condition may take the stale (old) value.

As early computation of branch condition may not always be possible, then predict.

0.4.2 Prediction

Branch prediction: Predict taken/not-taken (switch), predict never taken, predict always taken, predict by opcode, and scheme based on branch history table.

Predict never taken, predict always taken and predict by opcode - are static.

Predict taken/not-taken and scheme based on branch history table are dynamic.

These depend on execution history.

A. Predict-not-taken: Here, it is considered that branch I_b is an untaken branch.

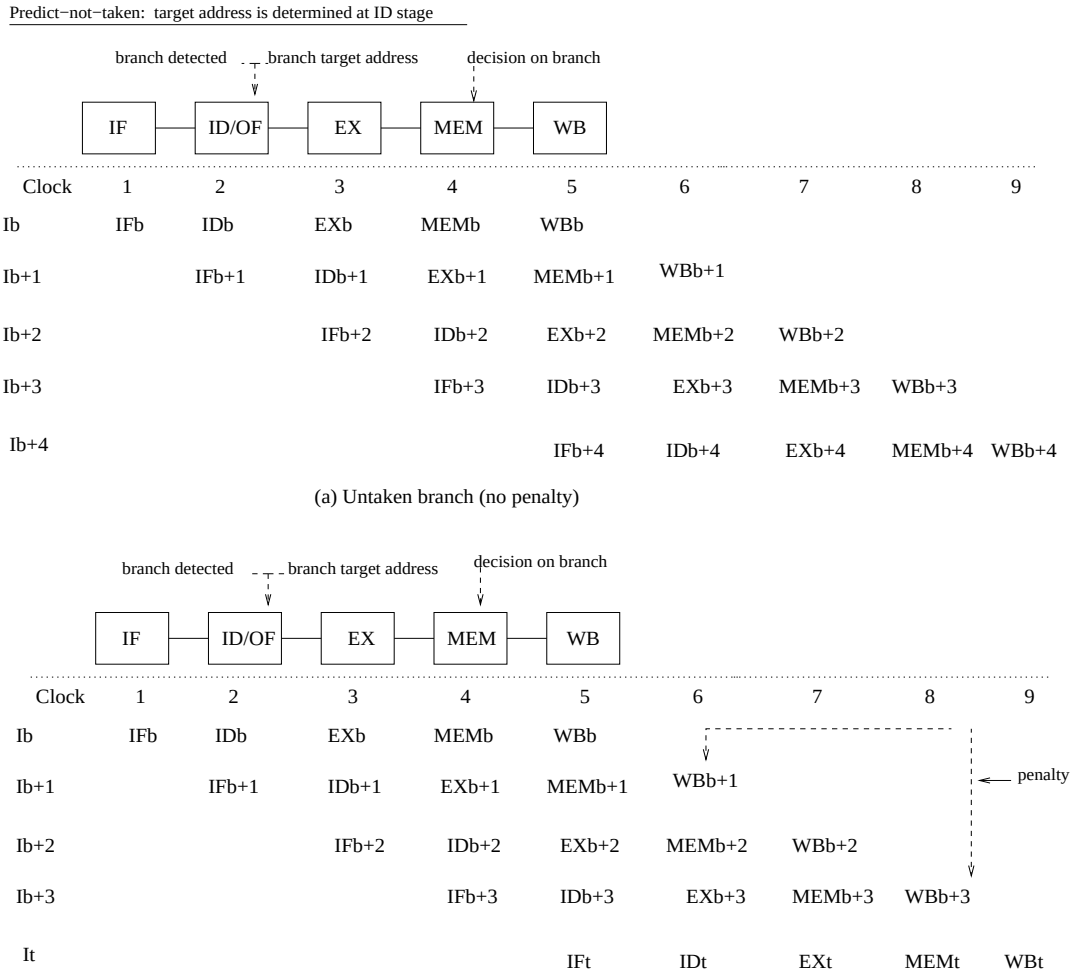


Figure 9: Predict-not-taken scheme to reduce branch stalls

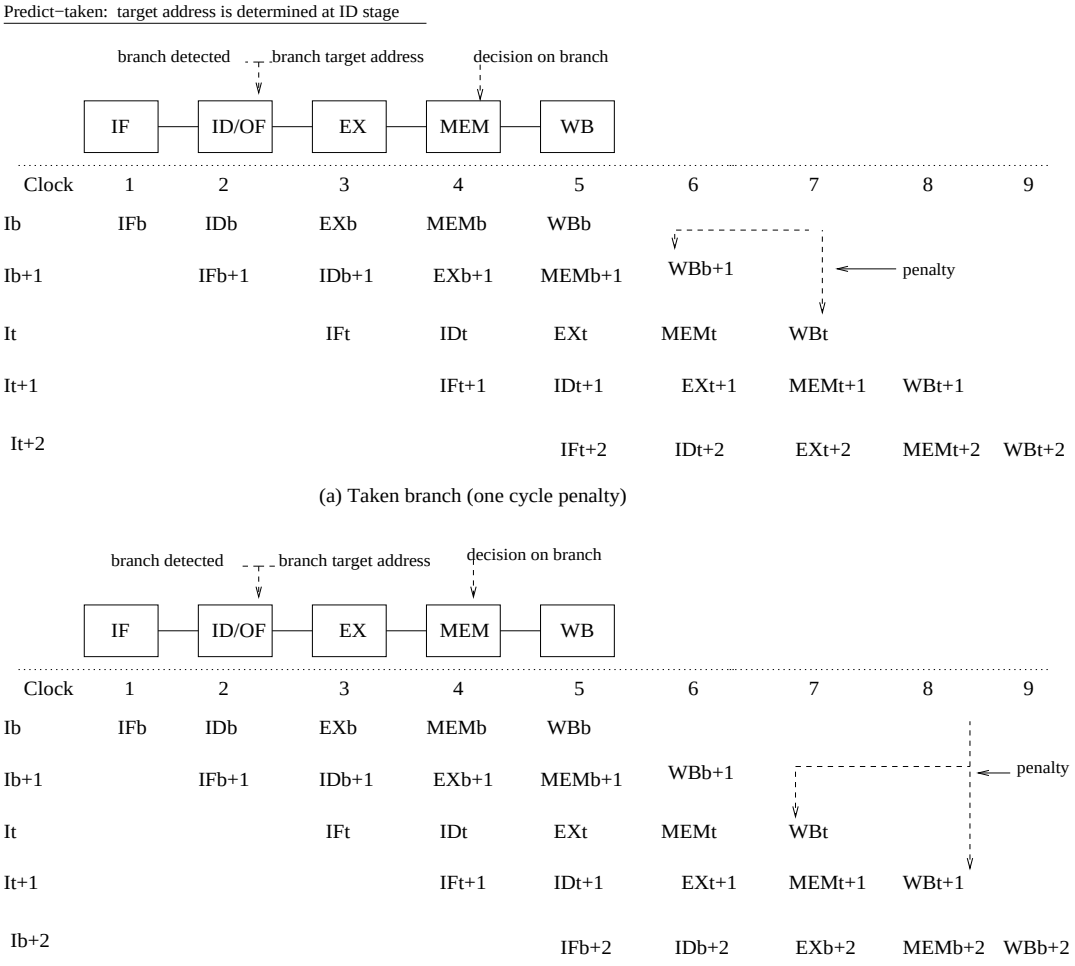
i) If untaken branch, I_{b+1}/I_{b+2}/I_{b+3} are to be executed (Figure 9(a)) - no penalty.

ii) If taken branch, I_{b+1}/I_{b+2}/I_{b+3} (fetched) are to be flushed out.

New instructions from I_t are to be fetched (Figure 9(b)) - penalty: 3 clock cycles.

B. Predict-taken: Here, while decoding I_b target address I_t is computed.

Pipeline control assumes branch is taken - instructions are fetched from I_t.



(a) Taken branch (one cycle penalty)

(b) Untaken branch (2−cycles penalty; declares It, It+1 as NOP)

Figure 10: Predict-taken scheme to reduce branch stalls

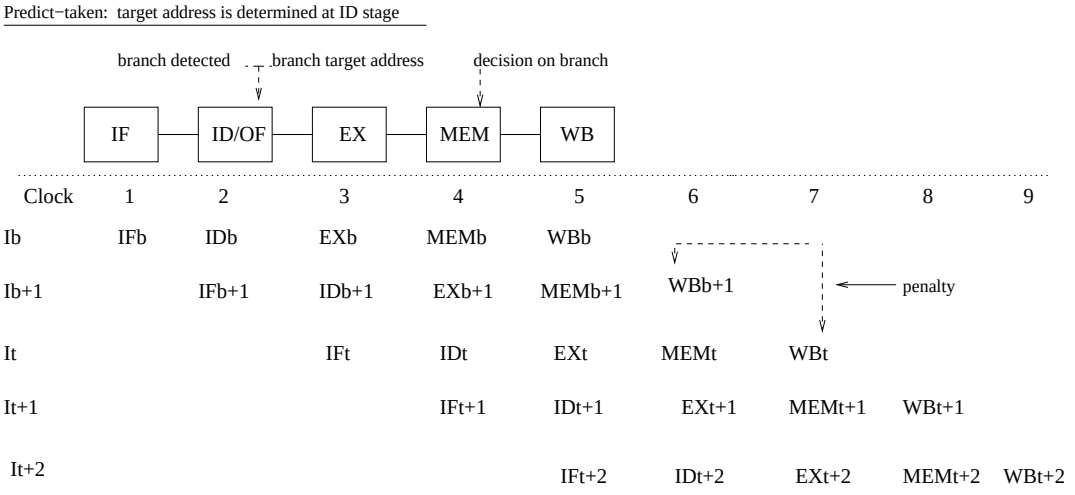
i) If branch is taken branch, penalty is one clock cycle - as I_{b+1} is in delay slot.

ii) If branch is untaken, penalty is of two clock cycles - as I_t and I_{t+1} are in pipe.

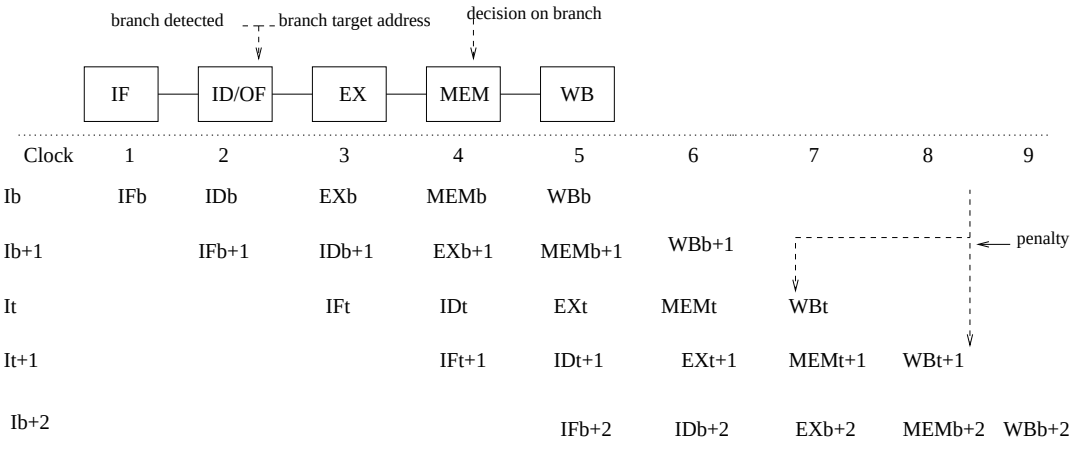
Performance summary,

Prediction	Actual	Penalty
Predict-not-taken	untaken branch	nil
Predict-not-taken	taken branch	3 cycles
Predict-taken	taken branch	1 cycle
Predict-taken	untaken branch	2 cycles.

A correct prediction, penalty is - nil or 1 cycle. 1 cycle penalty is due to delay slot.



(a) Taken branch (one cycle penalty)



(b) Untaken branch (2−cycles penalty; declares It, It+1 as NOP)

0.4.3 Managing delay slot

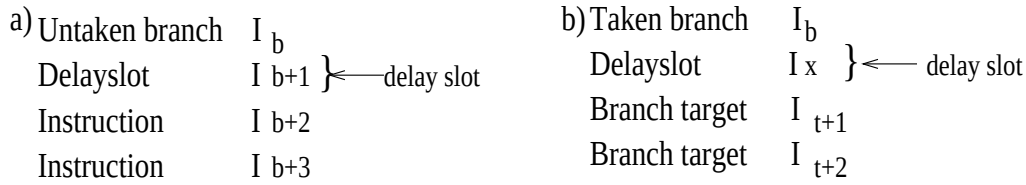


Figure 11: Delay slot

Waste of clock cycle for delay slot is nullified by reordering of instructions.

An appropriate instructions are placed in the delay slot.

It can be taken care of at compilation time.

Those instructions are executed while branch (target) is being resolved.

I_b is branch and I_{b+1} in delay slot.

I_{b+1} must be such that it is always executed irrespective of branch condition status.

That is, reorder instructions without a data hazard.

Scheduling delay slot Figure 12 shows the three processes.

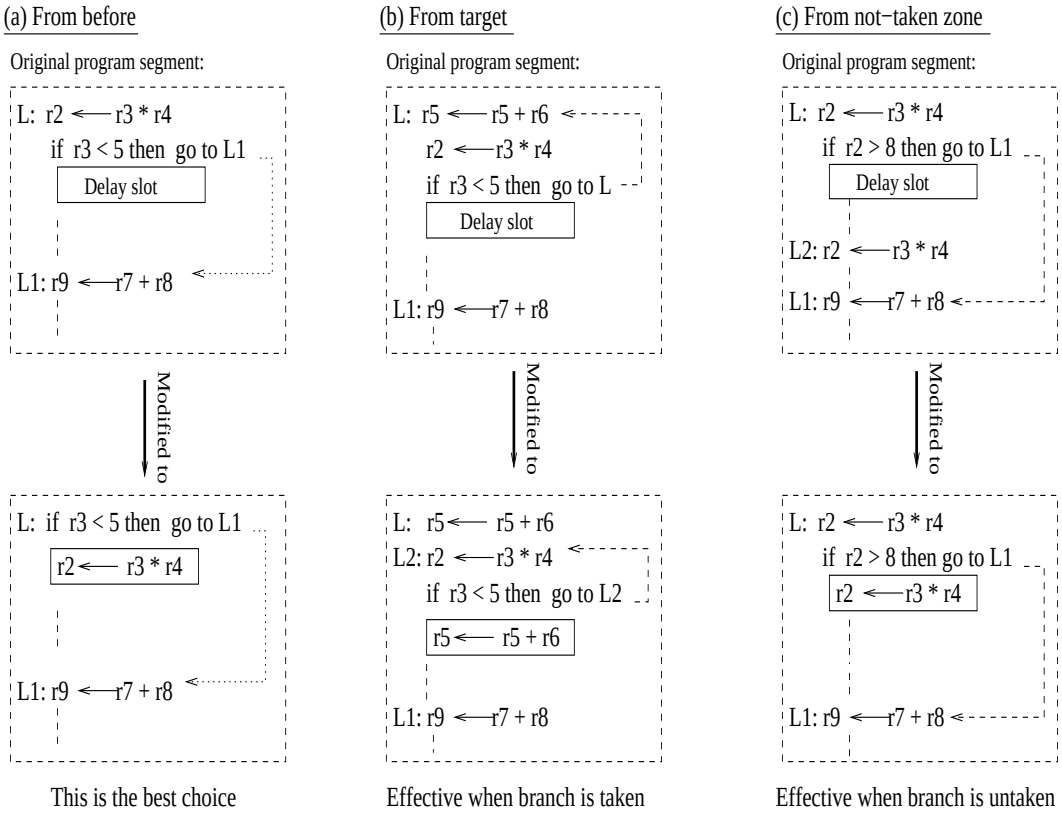


Figure 12: Delay slot scheduling

Figure 12(b): Selection of candidate (I_d) for delay slot from target.

It is preferred when there is high probability that branch will be taken.

That is, loop branch, unconditional branch.

Figure 12(c): Candidate selected from not-taken zone of branch.

Preferred when probability of untaken branch is very high.

In (b) and (c), I_d is to be turned into NOP if branch goes in unexpected direction.

Best choice - select I_d from instructions appear before branch (Figure 12(a)).

0.5 Realization of Branch Prediction

Simplest prediction: 'predict never taken' (permanently, predict-not-taken scheme).

Also 'predict always taken' -that is, set permanently the predict-taken scheme.

In 'predict always taken', system always fetches instruction from branch target.

Studies have shown that conditional branches are taken more than 50% of cases.

Therefore, 'predict always taken' should give better performance.

0.5.1 Static prediction

Prediction direction -that is, taken or not-taken is usually set into processor.

That is, either always assumes 'predict-taken' or always assumes 'predict-not-taken'.

In such static prediction, decision on prediction cannot be changed once is set.

In *semi-static*, prediction direction for each branch can be set by programmer/compiler.

0.5.2 Dynamic hardware prediction

It follows recent branch history for branch prediction (taken/or not-taken).

Requires additional hardware.

Branch history can be realized with n bits (n=1/2-bit prediction).

If count is $\geq (2^{n-1})$, branch prediction is 'predict-taken'.

1-bit prediction Decision on prediction depends on status of a bit called *pred*.

If *pred* = 1, then 'predict-taken'.

If *pred* = 0, 'predict-not-taken' scheme is considered.

pred is modified (flipped 1→0 or 0→1) iff prediction is found incorrect.

That is,

prediction for I_b was 'predict-taken' but I_b is untaken branch (*pred* flipped 1→0 Figure 13(a)),

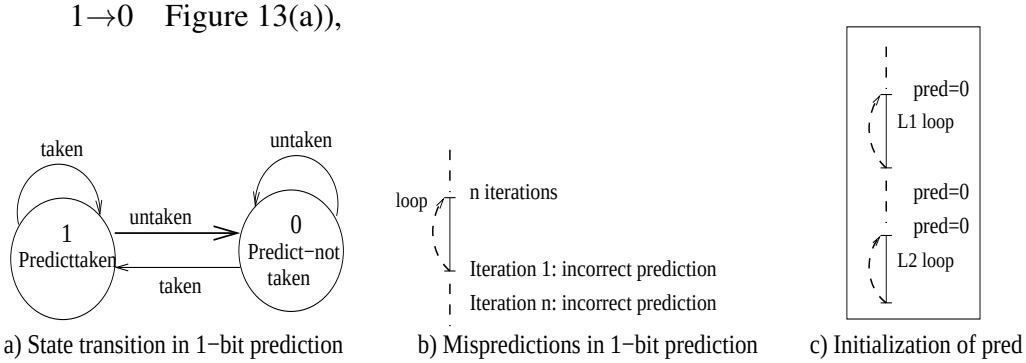


Figure 13: 1-bit prediction

or,

prediction for I_b was 'predict-not-taken' but I_b is taken branch (*pred* flipped 0→1).

In loop branch, number of mispredictions is only one for 'predict-taken' scheme.

In loop of i iterations, branch is taken i times, but untaken once (exits from loop).

Normally, in 1-bit prediction scheme, *pred* is initialized as 0 (Figure 13)(c)).

In 1-bit prediction, there can be two incorrect predictions for loop (Figure 13)(b)).

For a loop with 10 iterations, 8 correct predictions.

Its performance then is 80% (in 'predict-taken' scheme it is 90%).

Similarly, for a loop with 100 iterations, it gives 98 correct predictions.

Therefore, performance is 98% (in 'predict-taken' scheme it is 99%).