EEL 5764 Computer Architecture

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

Lecture 19-20: Pipelining (Contd.)

Announcements

- You should get project feedback and graded HW1 by October 9
- **If you have been asked to talk to me about project, please do so asap.**
- Mid-term 1 at **NEB 0202** October 22 8:30pm
  - → **No Class on Tuesday October 22**
  - → EDGE students **MUST** schedule appointment with ProctorU. **Please do this quickly. No make-up if you forget to (or otherwise don't) schedule appointment with ProctorU.**
- All of you are allowed one two-sided crib sheet
- EDGE students are also allowed 2 two-sided sheets of paper which can be used as scratch paper to work out the problems
  - → **Scratch papers must be scanned (a picture with your mobile phone will work) and submitted to the professor via email (sandip@ece.ufl.edu) no later than 15 minutes of the completion of the test**
- **Your HW2 deadline has been extended by two days**

Basic Pipeline

To improve performance, we can make circuit faster, or use …

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Ideal time/instruction =

$$\frac{\text{time/instruction unpipelined}}{\text{\# pipeline stage}}$$

Ideal Pipeline and Performance

- Balanced pipeline (each stage has the same delay)
- Zero overhead due to clock skew and pipeline registers
- Ignore pipeline fill and drain overheads

$$Average\ time/instruction = \frac{Average\ time/instruction_{non-pipelined}}{Number\ of\ pipeline\ stages}$$

$$Speedup = \frac{Average\ time/instruction_{non-pipeline}}{Average\ time/instruction_{pipeline}}$$

$$= Number\ of\ pipelinestates$$

Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages
- However, this increase would be achieved only if
  - → all pipeline stages require the same time to complete, and
  - → there is no interruption throughout program execution
- Unfortunately, this is not true
  - → there are times when an instruction cannot proceed from one stage to the next in every clock cycle

Pipeline Hazards

- → Hazards may require "stalling" the pipeline allowing some instructions to proceed while others are delayed
  - → With no additional instructions fetched
  - → until the conditions that caused the hazard do not exist anymore
- → This is called "clearing the hazard"
- → Stalling increases the CPI,
  - → reduces the speedup from pipelining

$$Speedup = \frac{Number\ of\ pipeline\ stages}{1 + Stall\ cycles/instruction}$$

Data Hazards

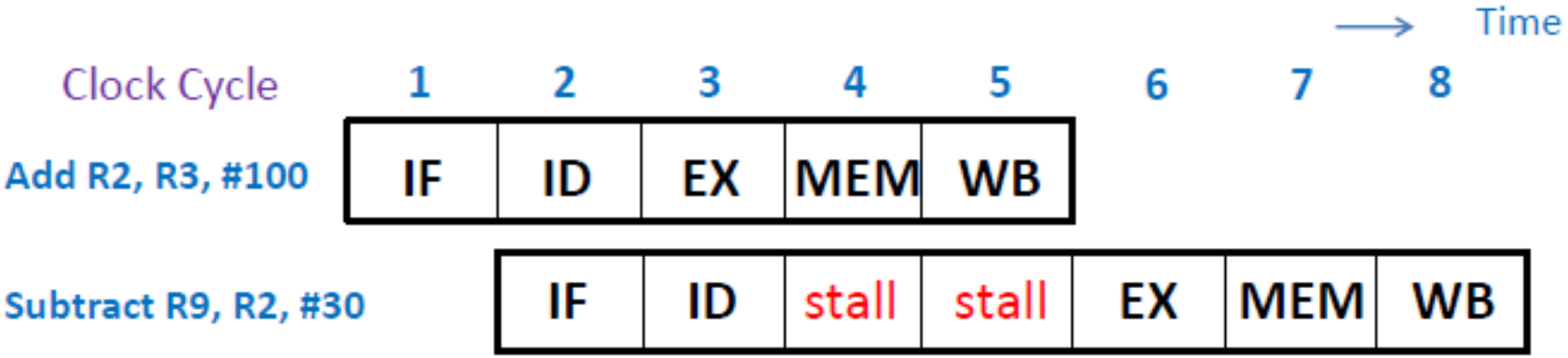
- Any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
- Example

  Add R2, R3, #100 ; R2 <- [R3] + 100

  Subtract R9, R2, #30 ; R9 <- [R2] + 30

  - First instruction writes to register R2 in the WB stage
  - Second instruction reads register R2 in the ID stage
  - If second instruction reads R2 before the first instruction writes R2, the result of second instruction would be incorrect, as it would be based on R2's old value (read-after-write dependence)
  - To obtain the correct result, second instruction needs to wait until the first instruction has written to R2

Data Hazards



- Subtract is stalled in decode stage for two additional cycles to delay reading R2 until the new value of R2 has been written by Add
  - How is this pipeline stall implemented in hardware?
- Control circuitry recognizes the data dependency when it decodes Subtract (comparing source register ids with dest. register ids of prior instructions)
- During cycles 3 to 5:
  - Add proceeds through the pipe
  - Subtract is held in the ID stage
- In cycle 5
  - **Add** writes R2 in the first half cycle, **Subtract** reads R2 in the second half cycle

Types of Data Hazards

- Read After Write (RAW) True dependence
  - Caused by "dependence". Subsequent instruction has actual need for data produced by earlier instruction
- Write after Read (WAR) False dependence
  - Called "anti-dependence". Can't occur in in-order pipelines (e.g., our simple MIPS pipeline). We'll see it later in advanced pipelines
- Write after Write (WAW) False dependence
  - Called "output dependence". Can't occur in in-order pipelines (e.g., our simple MIPS pipeline). We'll see it later in advanced pipelines

Types of Data Hazards - RAW

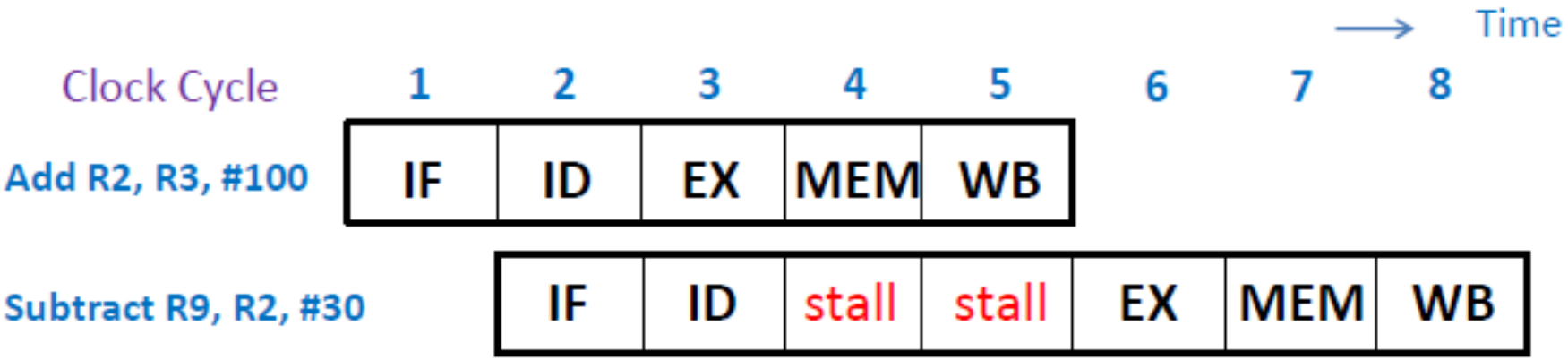
→ $Inst_j$ tries to read operand before $Inst_i$ write it (j > i)

$Instr_i$: **add r1, r2, r3**

$Instr_j$: **sub r4, r1, r3**

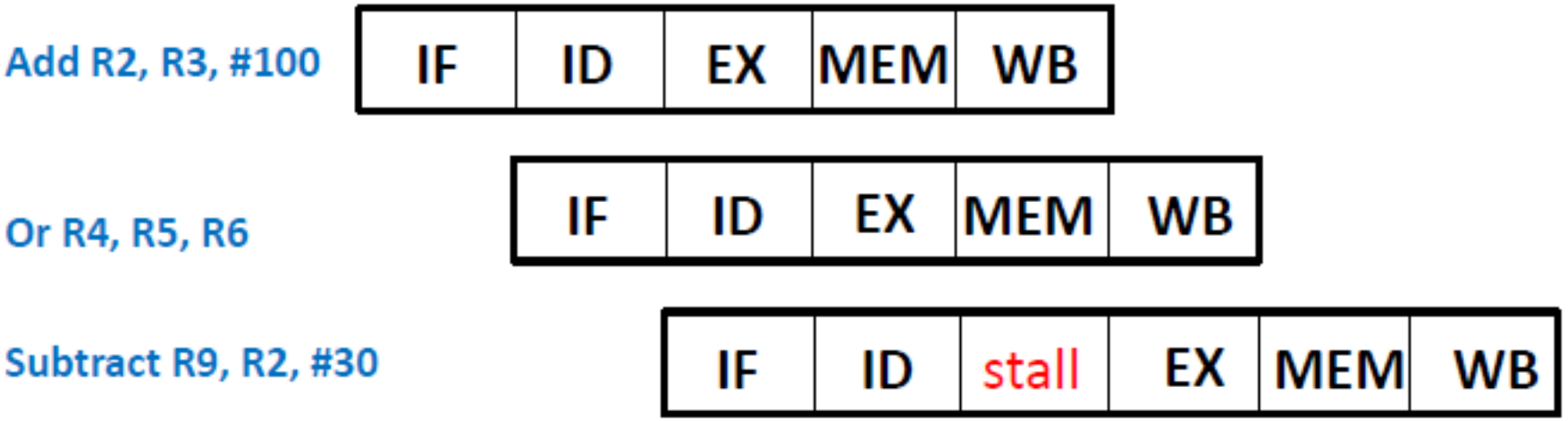
→ This hazard results from a true dependence: an actual need for communication from the first instruction to the second

Data Hazards – Stall Penalty



→ If the dependent instruction follows right after the producer instruction, we need to stall the pipe for 2 cycles (Stall penalty = 2)

Data Hazards – Stall Penalty



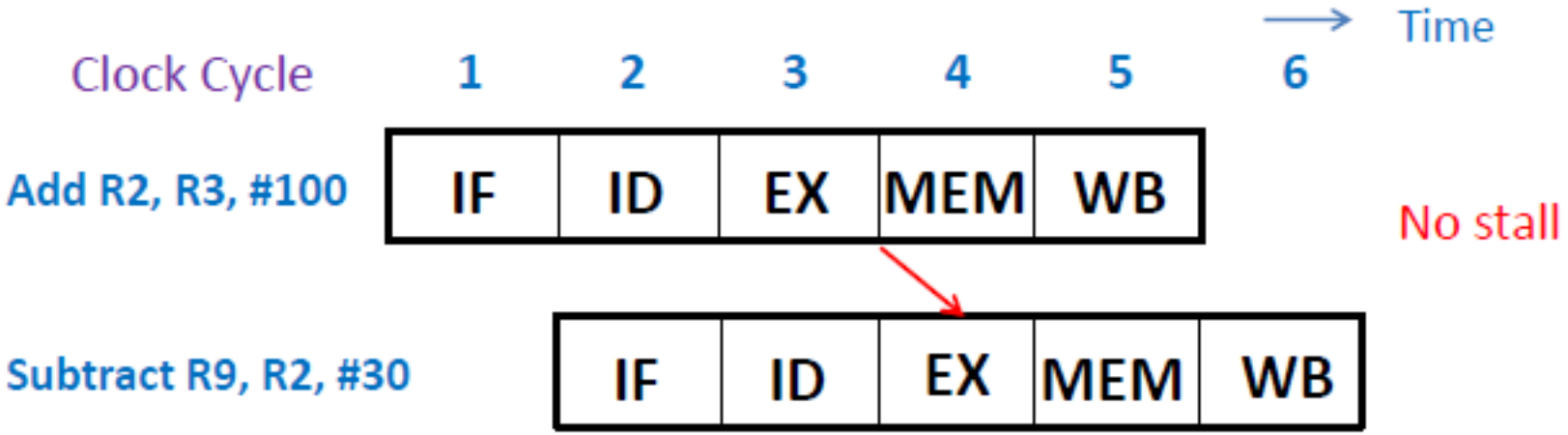
- → What happens if the producer and dependent instructions are separated by an intermediate instruction?
- → In this case, stall penalty = 1
- → Stall penalty depends upon the **distance** between the producer and dependent instruction

Data Hazards – Stall Penalty

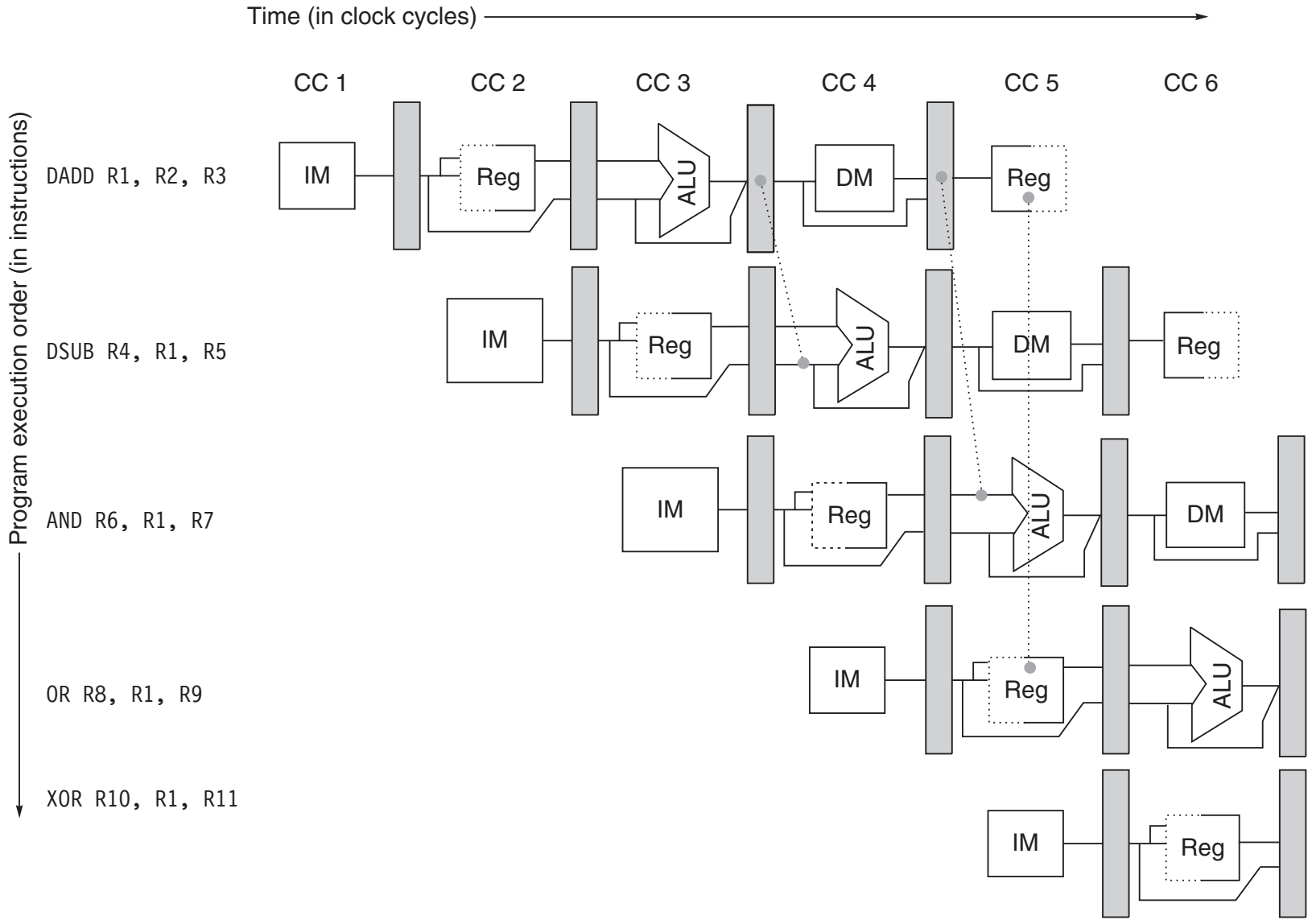
- → Frequent stalls caused by data hazards can impact the performance significantly.
- → **Example**: If every 5th instruction causes a 2-cycle stall due to a data hazard, then the CPI can increase by (1/5)*2 = 0.4 stall cycles
- → Would like to avoid stalls caused by RAW hazards
  - → Technique: forwarding
- → Also called “bypassing” or “short-circuiting”
  - → **Key Idea: Forward results from later stages in the pipeline to earlier stages**

Reduce Data Hazard Stalls – Forwarding

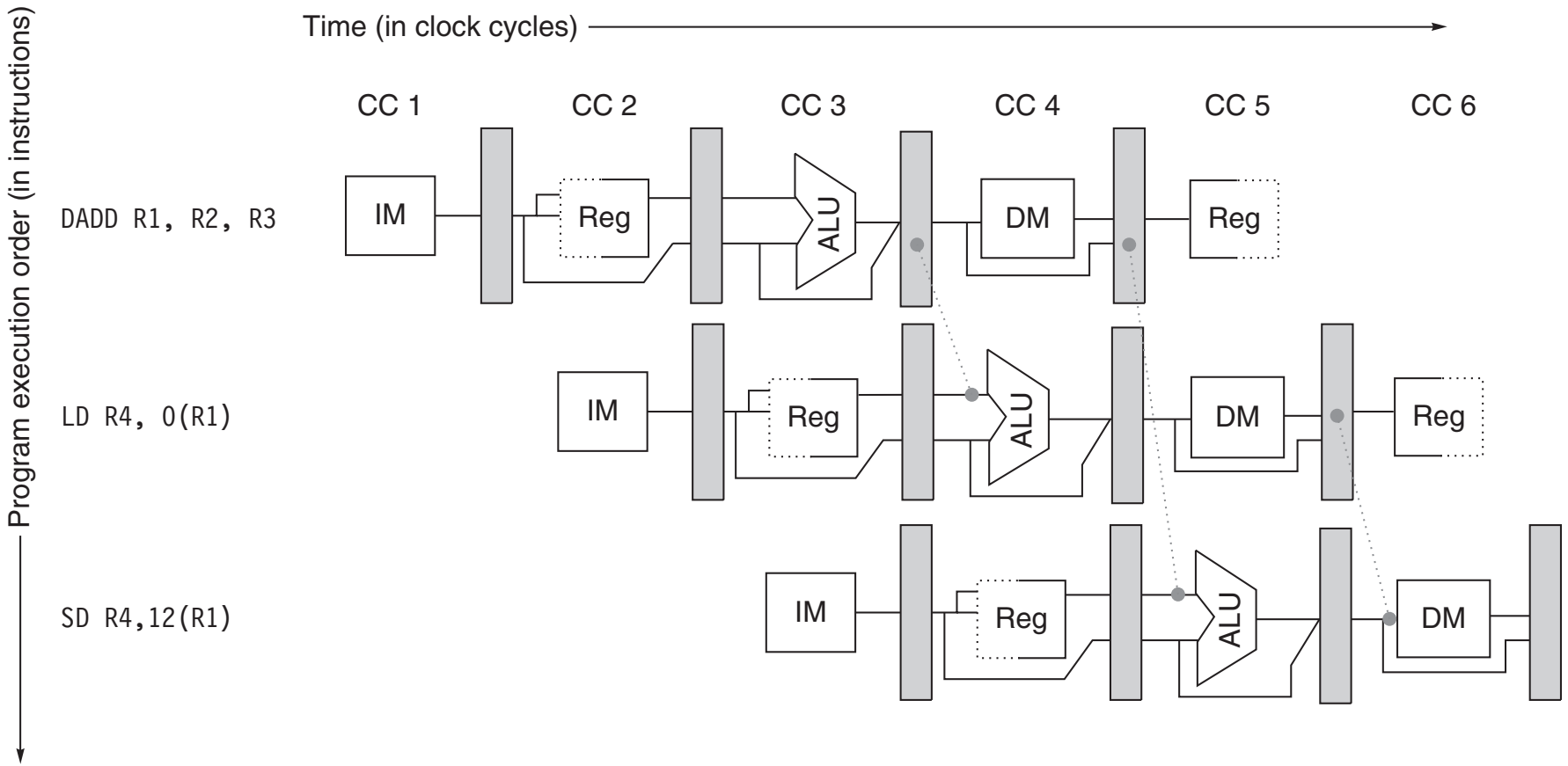
- Consider the two instructions discussed in the previous example
  - The result of producer instruction is actually available after the completion of EX stage (cycle 3), when the ALU completes its computation
  - Instead of stalling the dependent instruction, the hardware can forward the result from the output of the EX stage to the ALU, where it can be used by the dependent instruction
  - The arrow in below figure shows data being forwarded from EX stage of first instruction to EX stage of the second instruction



Reduce Data Hazard Stalls – Forwarding

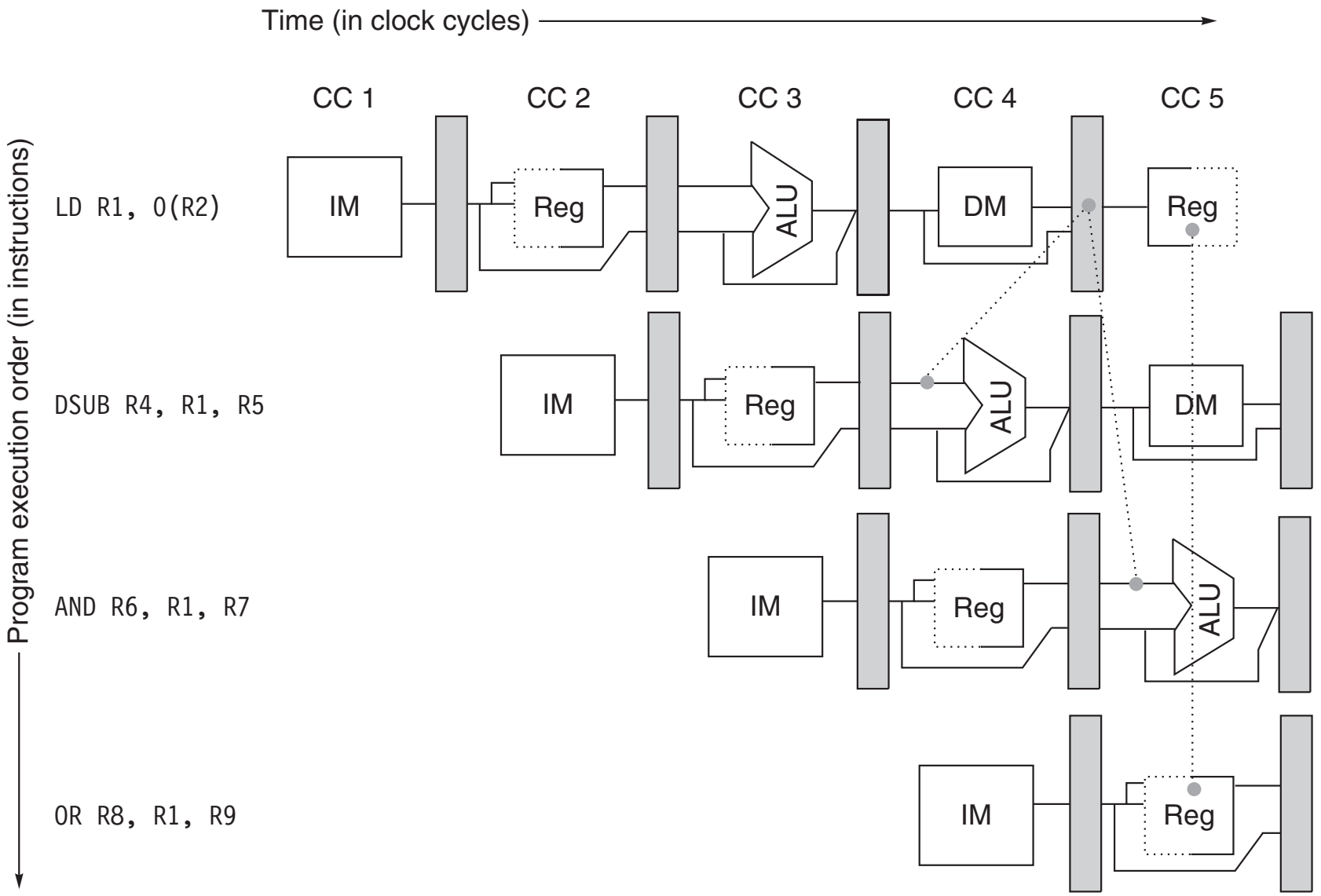


Reduce Data Hazard Stalls – Forwarding



Passing results from one stage to inputs of another stage

Data Hazard Requiring Stalls



Branch Hazards

- → Control hazards are caused by a delay in the availability of an instruction or the memory address needed to fetch the instruction
- → In ideal pipelined execution, a new instruction is fetched every cycle, while the previous instruction is being decoded
- → This will work fine as long as the instruction addresses follow a pre-determined sequence (e.g., PC ←[PC]+4)
- → However, branch instructions can alter this sequence
- → Branch instructions first need to be executed to determine whether and where to branch
- → Pipeline needs to be stalled before the branch outcome is decided

Conditional Branches

→ Consider a conditional branch instruction such as

BEQ R5, R6, LOOP

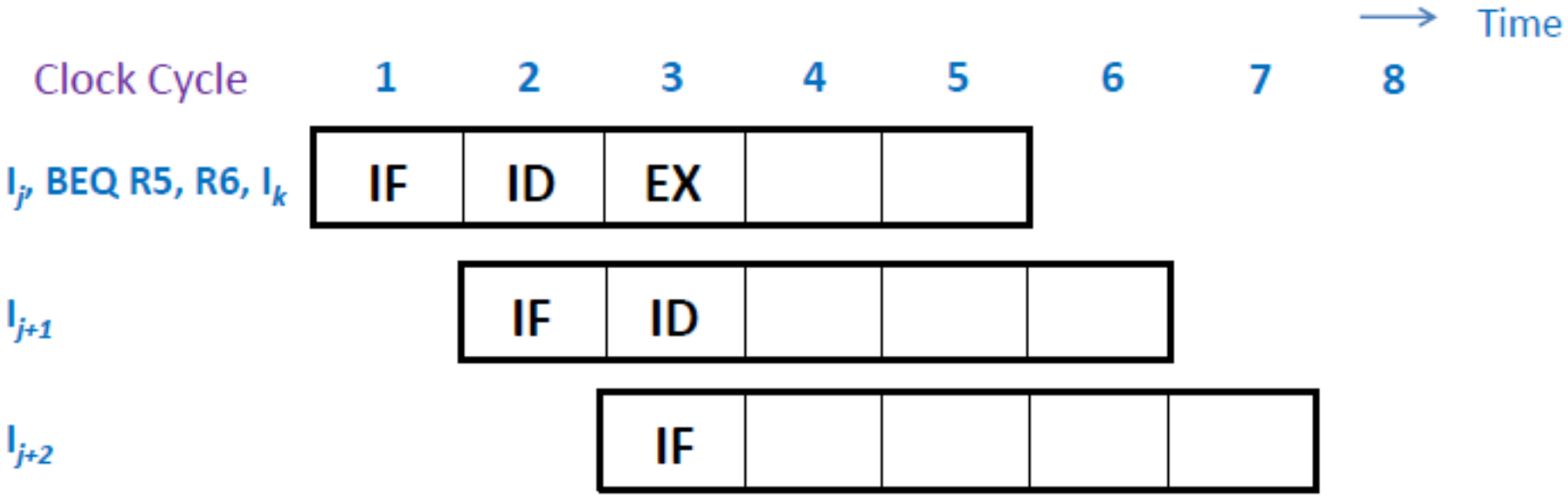
→ Testing the branch condition (comparison between [R5] and [R6]) determines whether the branch is taken or not taken

→ Both the comparison and target address calculation happens in the EX stage (3rd cycle of instruction processing)

→ If the branch is **taken**

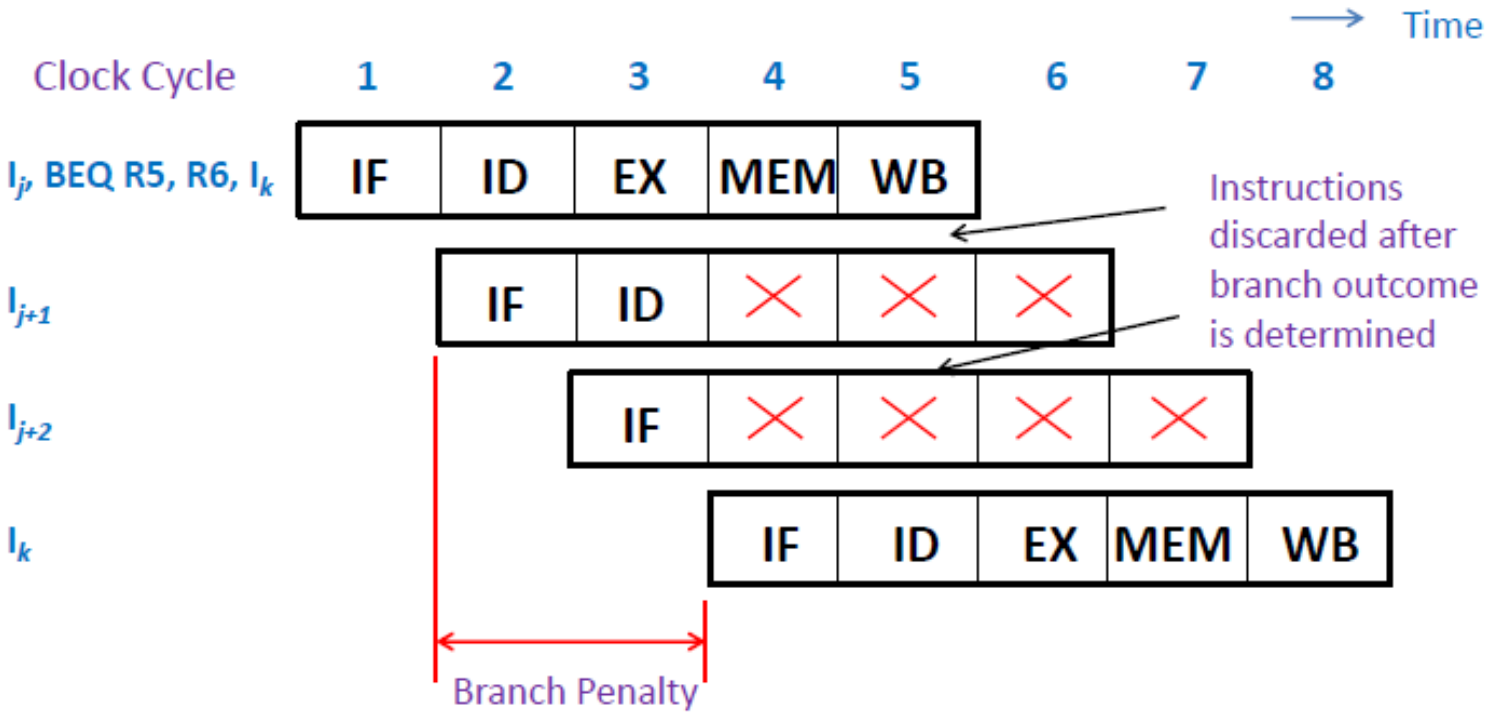
- → Cannot fetch the branch target before the start of 4th cycle
- → No useful instruction fetched in cycles 2 and 3→ Control Hazard

Handling Control Hazards



- → Let us assume R5 = R6
- → In cycle-3, EX stage determines that the branch is taken and computes the target address
- → PC is updated with the branch target address
- → Two instructions (I_{j+1} and I_{j+2}) on the wrong path have already been fetched

Handling Control Hazards



→ Control transfers to branch target (I_k)

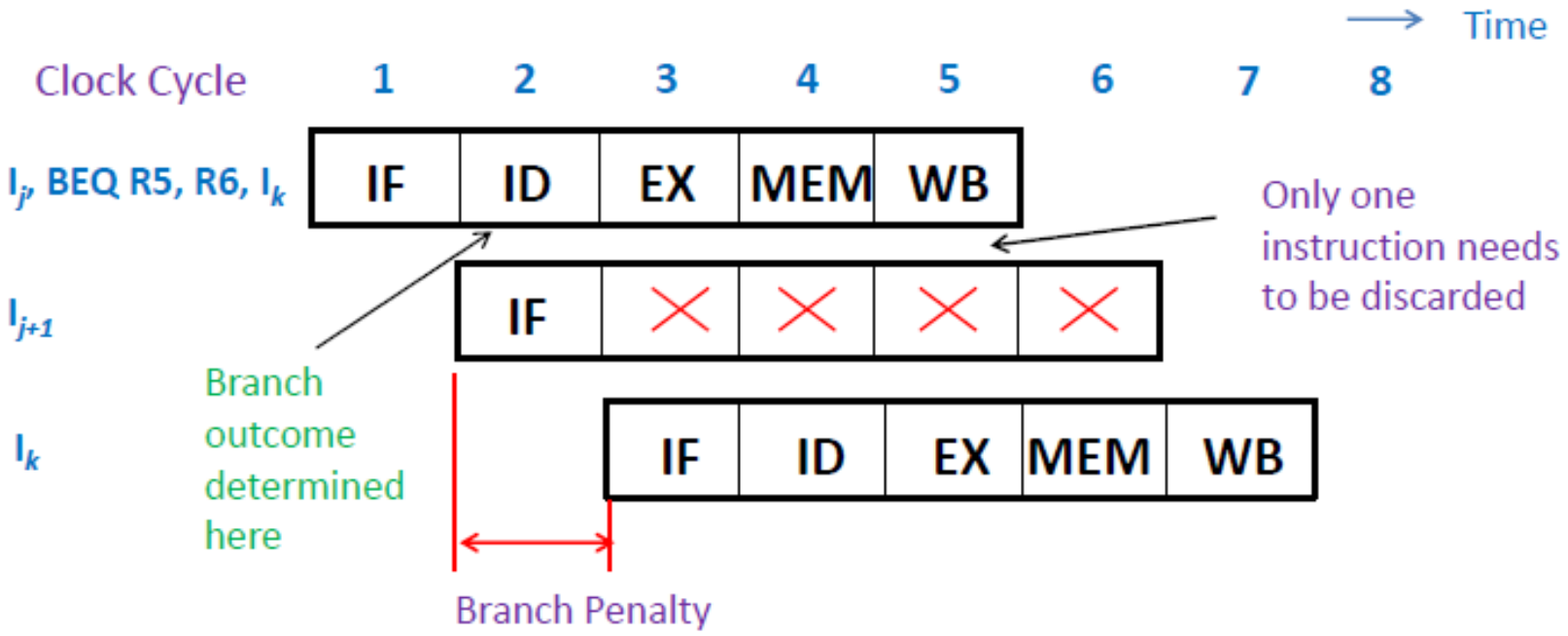
→ Instructions I_{j+1} and I_{j+2} flushed from the pipeline

→ Branch Penalty in this case is 2 cycles

Reduce Branch Penalty

- How to reduce the branch penalty for conditional branches?
  - Test the branch condition in the ID stage
    - Move the comparator from EX stage to ID stage
    - Branch condition tested in parallel with target address computation
    - Branch penalty reduced to one cycle
  - Also compute the branch target address in the ID stage
    - Need an adder in the ID stage to add the branch offset to the PC
    - When the decoder determines that the instruction is a conditional branch, the computed target address is available before the end of the ID stage
    - Update the PC before the end of ID stage
  - Negative Effect: ID stage lengthened; may have to increase the clock period

Reduce Branch Penalty



→ Branch condition and target address computed in cycle # 2

→ Branch penalty reduced to one cycle

→ Only one instruction (I_{j+1}) is fetched incorrectly and needs to be discarded

Reducing Pipeline Branch Penalties

- **Pipeline flush** – simple, but w. fixed branch penalties
  - Stall until branch target is known
- **Predicted-not-taken**
  - Increment PC by 4 every cycle and keep fetching sequential instructions after the branch instruction
  - After the branch outcome is computed
    - If the branch condition is false (Not-taken branch):
      - We have already been fetching instructions from the correct path => no problem
    - But if the branch condition is true (Taken branch):
      - Need to start fetching from the branch target
      - What happens to the instructions that have already been fetched on the wrong path?

Handling Control Hazards

- **Predict-taken**: Treat every "branch" as taken
  - In our 5-stage pipeline, we don't know target any earlier than we know branch outcome
  - No advantage, unless we could predict the target address earlier in the pipeline
- **Delayed Branch**
  - Compiler optimization
  - Use knowledge of the program semantics to rearrange instructions before/after the branch

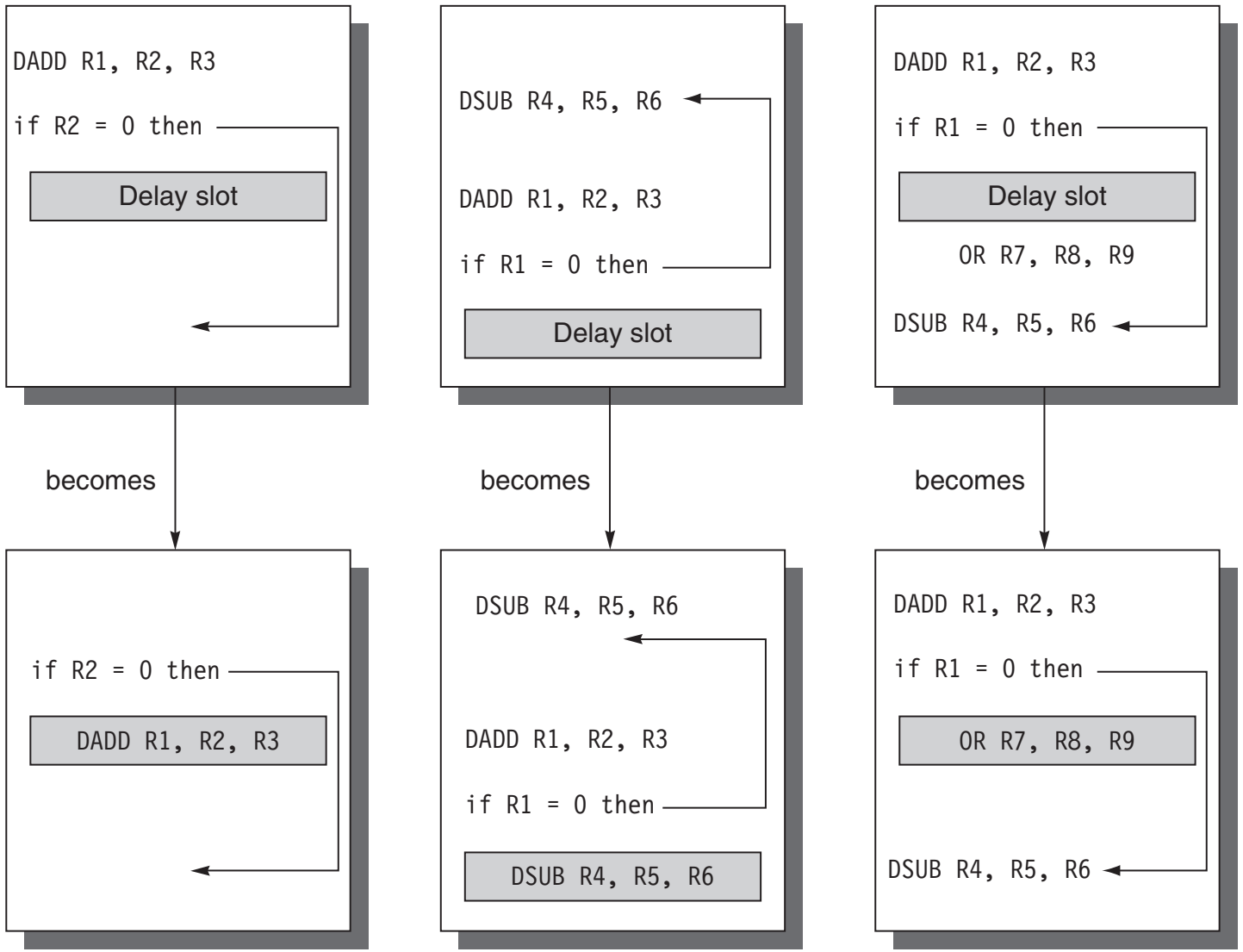
Delayed Branch

Branch instruction
Sequential successor - in branch delay slot
Branch target if taken

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Scheduling Branch Delay Slot



(a) From before (b) From target (c) From fall-through

Performance of Branch Schemes

$$Speedup = \frac{Pipeline\ depth}{1 + Stall\ cycles\ from\ branches}$$

$$Stall\ cycles\ from\ branches = branch\ freq \times branch\ penalty$$

$$Speedup = \frac{Pipeline\ depth}{1 + branch\ freq \times branch\ penalty}$$

Performance of Branch Schemes - Example

→ Assume that branches comprise 20% of all instructions. Also assume that the branch prediction is 80% accurate and incurs a 2 cycle stall on each misprediction. What is the impact of control hazards on the CPI of the pipelined processor? Ignore all other sources of pipeline hazards.

→ ----------

→ CPI without control hazards = 1

→ Added CPI due to control hazards = Branch frequency * (1 – Branch prediction accuracy) * Stall penalty = 20% * (1 - 80%) * 2 = 0.08

→ CPI with control hazards = 1 + 0.08 = 1.08

Branch Prediction

- **Static Prediction**
  - individual branches biased toward taken or not-taken
  - Opcode based
  - Displacement based (forward not taken, backward taken)
  - Compiler directed (branch likely, branch not likely)
- **Dynamic Prediction**
  - 1 bit predictor - remember last taken/not taken per branch
  - Use a branch-prediction buffer or branch-history table with 1 bit entry
  - Use part of the PC (low-order bits) to index buffer/table
  - Invert the bit if the prediction is wrong
  - Backward branches for loops will be mis-predicted twice
    - once upon loop exit and then again on loop entry

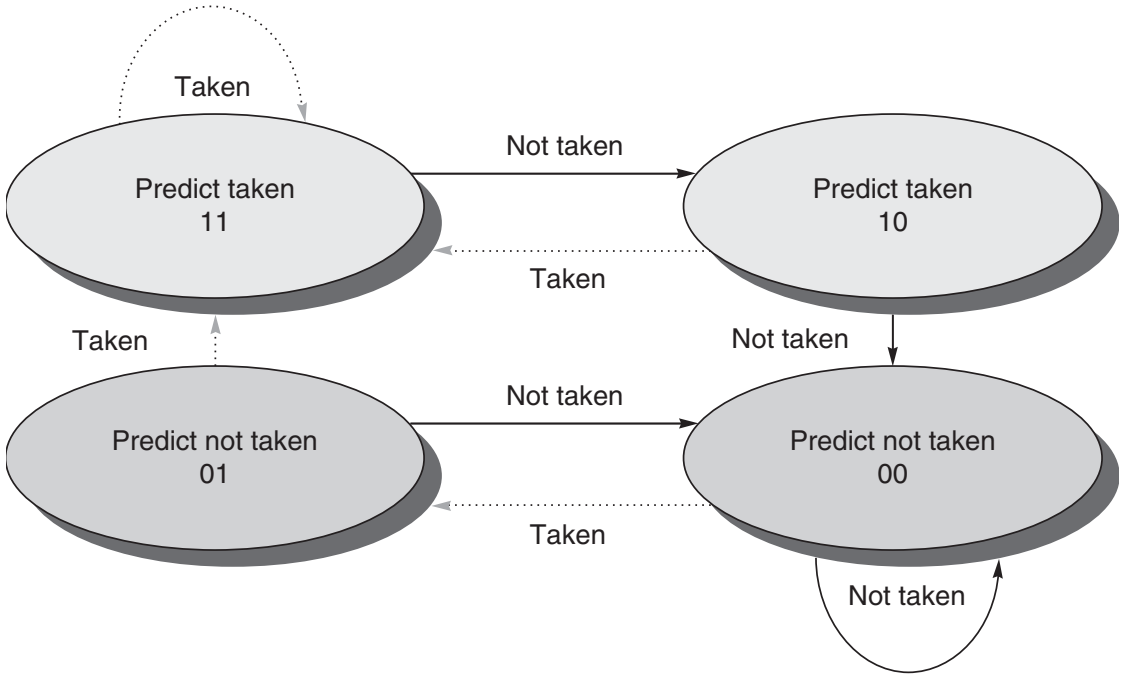
1-Bit Predictor

- Example: in a loop, 1-bit BHT will cause 2 mis-predictions
- Consider a loop of 9 iterations before exit:

```
for (…){
        for (i=0; i<9; i++)
                a[i] = a[i] * 2.0;
}
```

  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Mis-predict *twice* every time the inner loop is executed

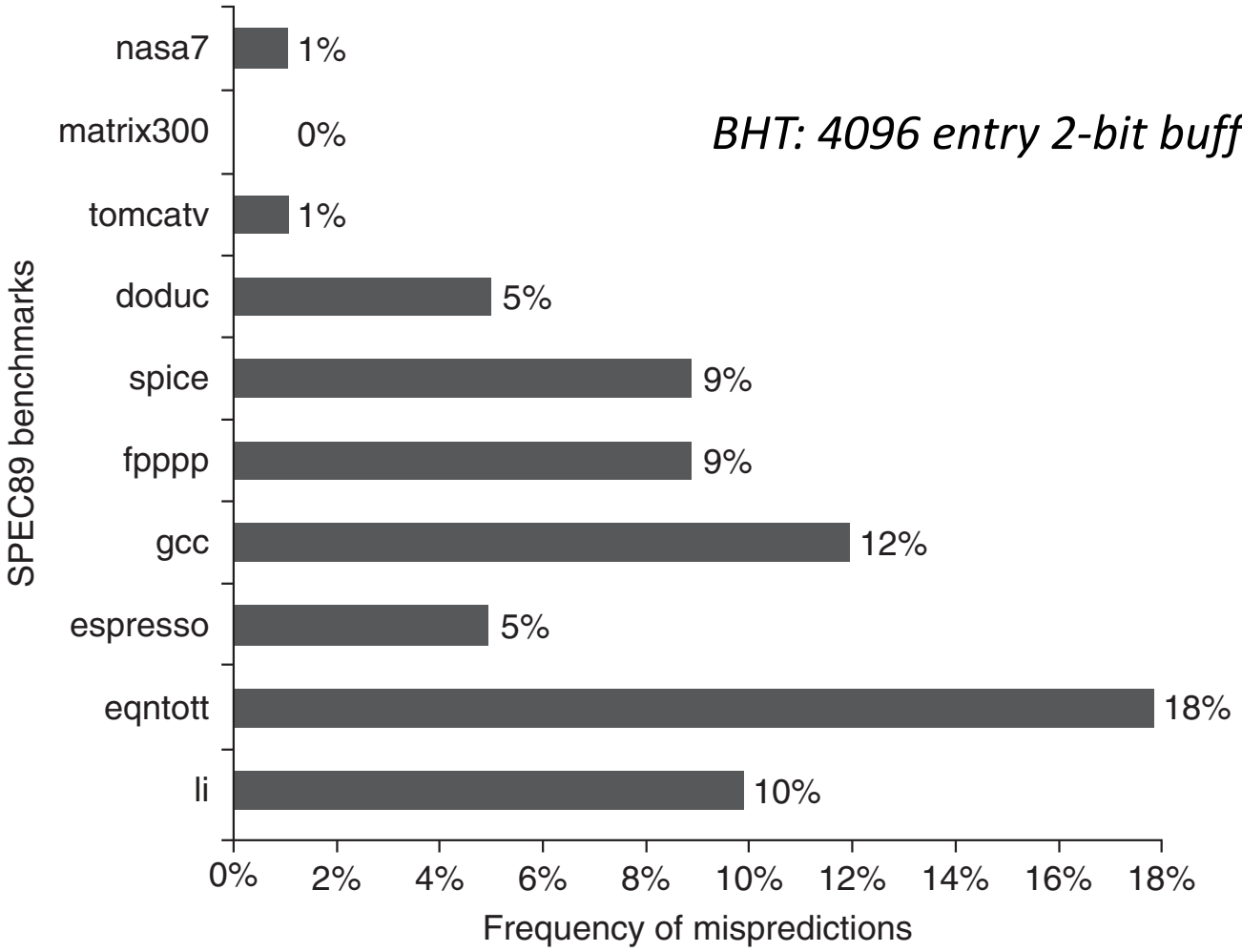
2-Bit Predictor



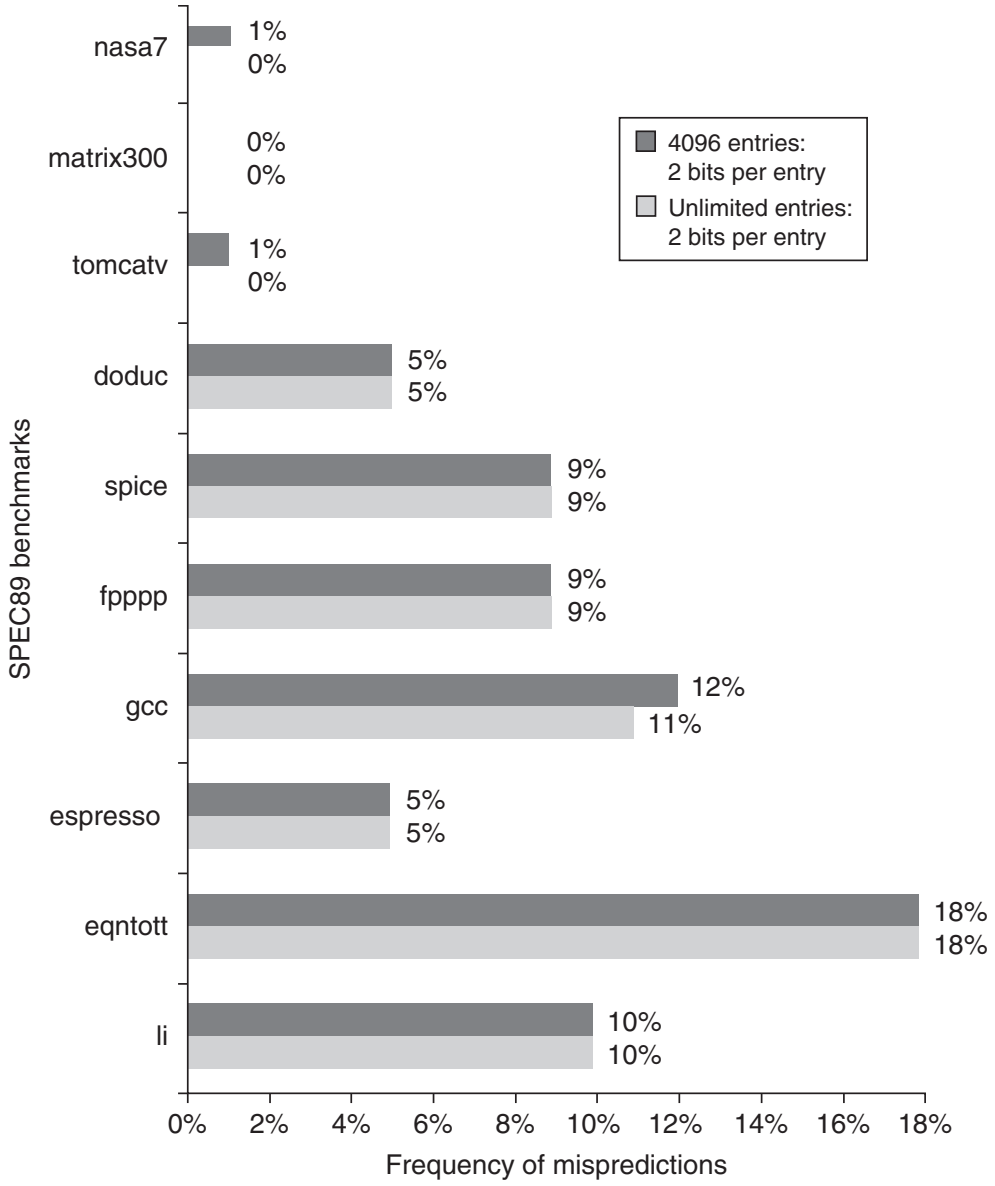
→ Solution: 2-bit scheme where change prediction only if get mis-prediction *twice consecutively*

→ Other solutions: correlating predictor, tournament predictors

2-Bit Predictor Accuracy



4K vs Unlimited Buffer for Prediction



Correlating Branch Predictors

→ Rationale: branch outcomes may be correlated

If (aa == 2)

aa = 0;

If (bb == 2)

bb = 0;

If (aa != bb) {…}

```
        DADDIU   R3,R1,#–2
        BNEZ     R3,L1          ;branch b1   (aa!=2)
        DADD     R1,R0,R0       ;aa=0
L1:     DADDIU   R3,R2,#–2
        BNEZ     R3,L2          ;branch b2   (bb!=2)
        DADD     R2,R0,R0       ;bb=0
L2:     DSUBU    R3,R1,R2       ;R3=aa-bb
        BEQZ     R3,L3          ;branch b3   (aa==bb)
```

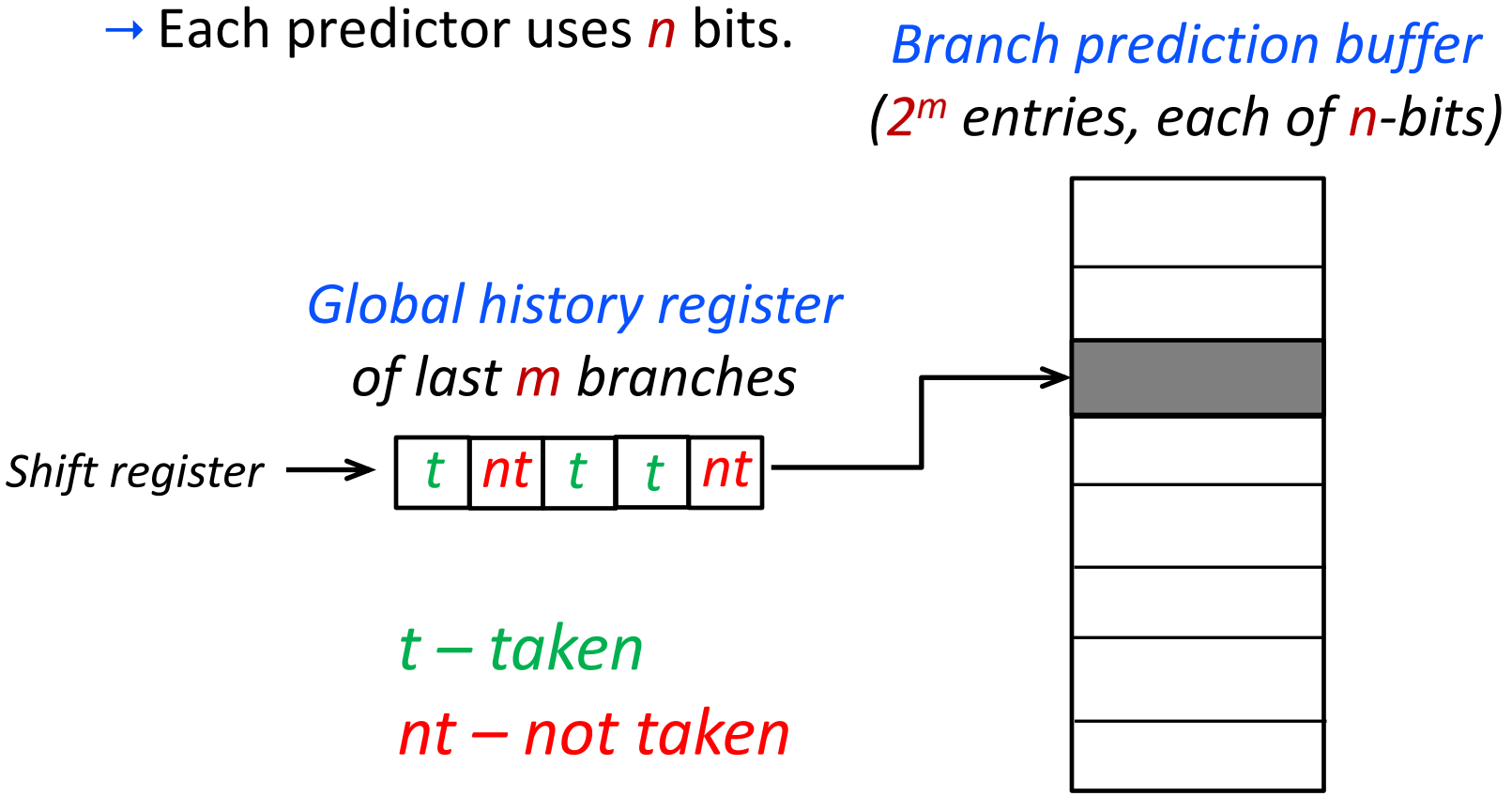
If both b1 and b2 are not taken, then b3 is taken.

Use history of last m branches to predict the current one.

Correlating Branch Predictors

→ Use history of last *m* branches to predict the current branch.

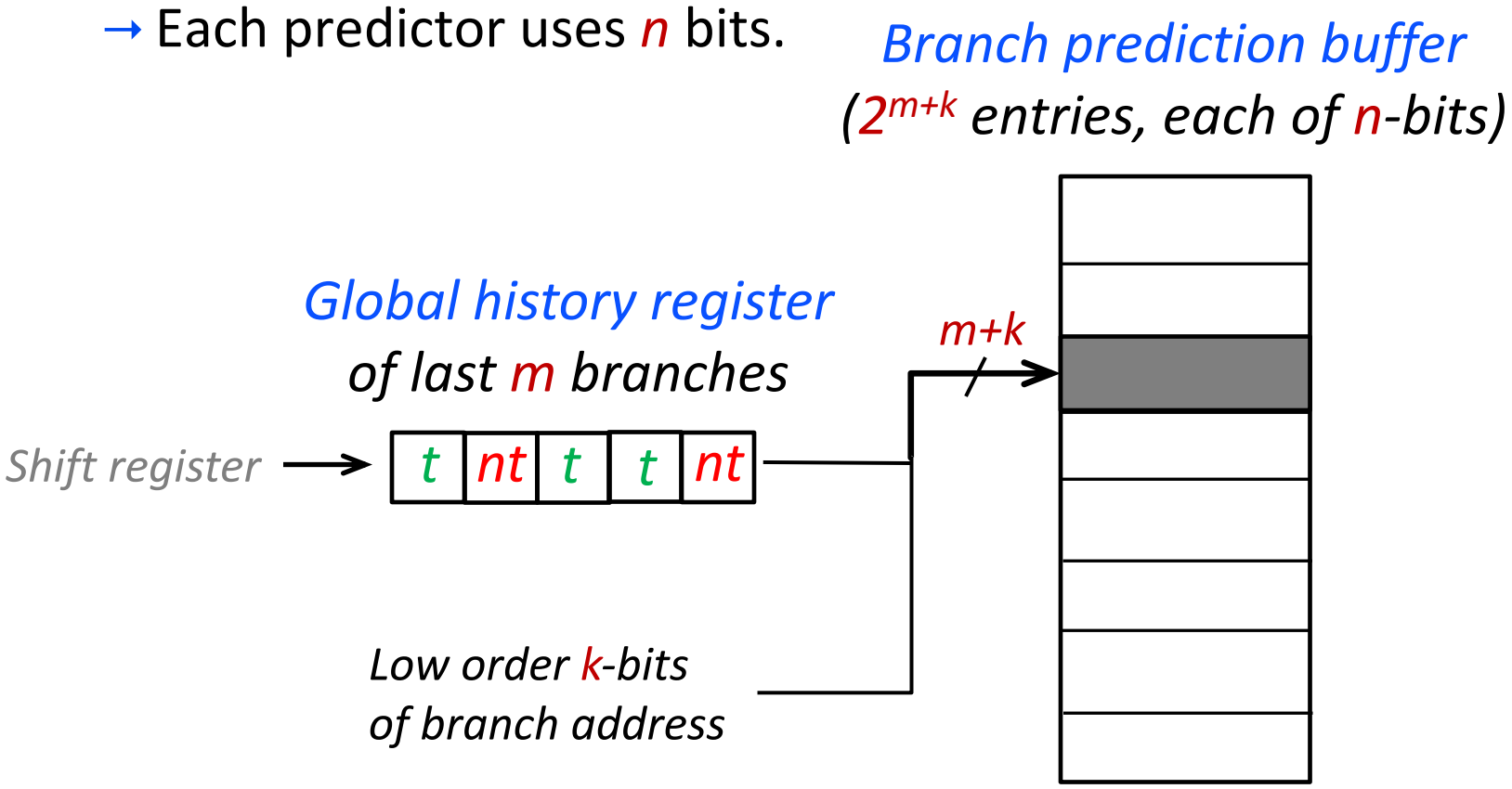
→ Each predictor uses *n* bits.



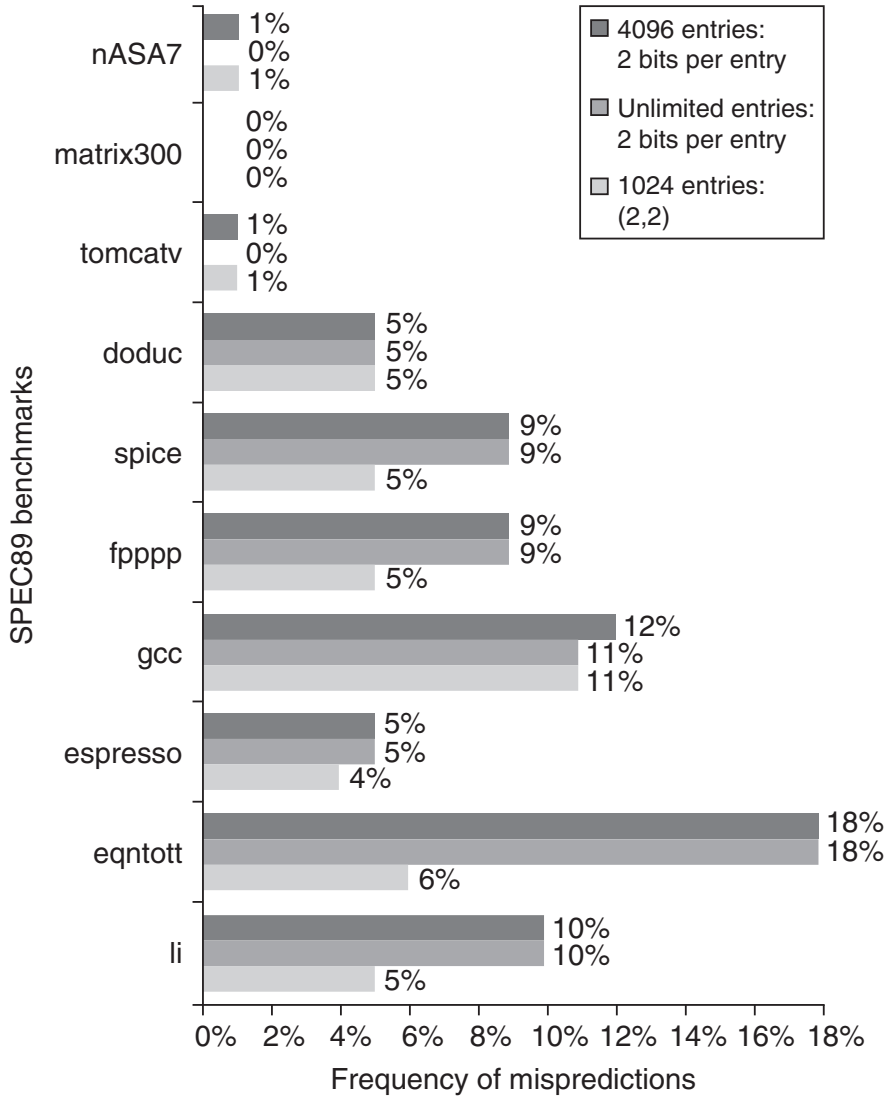
Correlating Branch Predictors

→ Use history of last *m* branches to predict the current branch.

→ Each predictor uses *n* bits.



Correlating Branch Predictors



Tournament predictor selects the best of local and global predictors, and leads to better prediction.