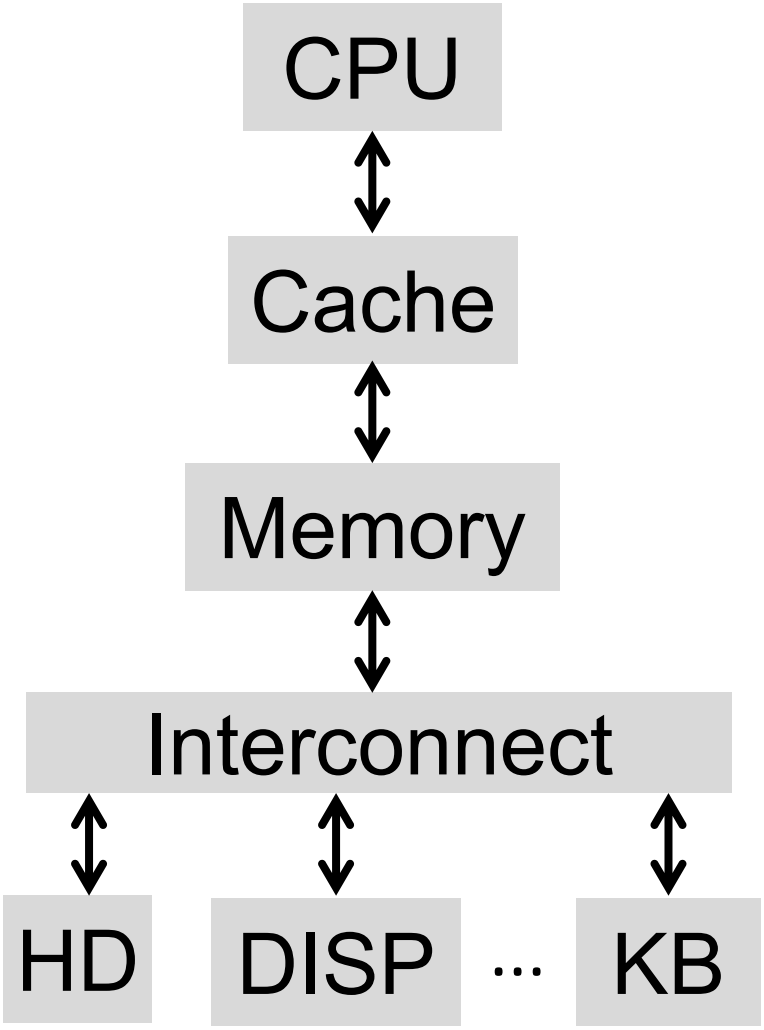
EEL 6764 Principles of Computer Architecture
Instruction-Level Parallelism

Dr Hao Zheng
Dept. of Comp Sci & Eng
U of South Florida

Reading

- Appendix C
- Chapter 3

A Simplified View of Computers

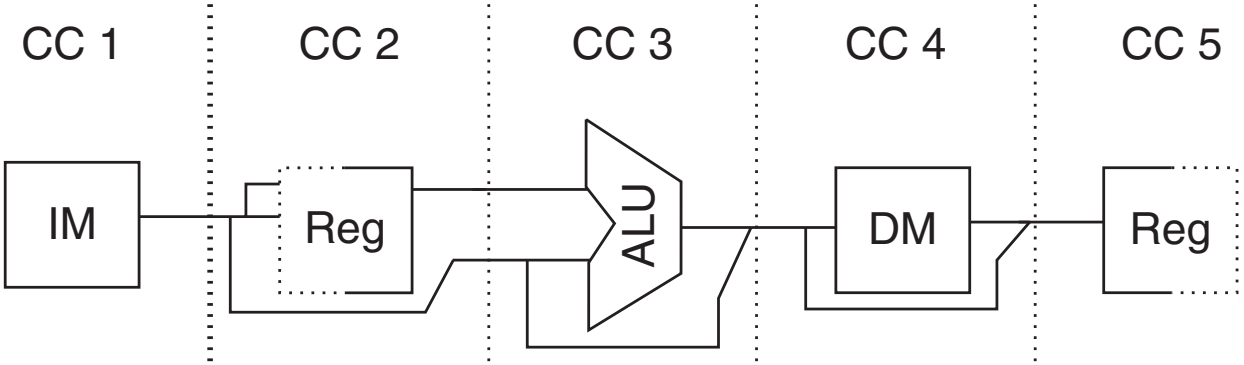


Introduction

- Design Principle – exploit parallelism
- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
- There are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMD processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

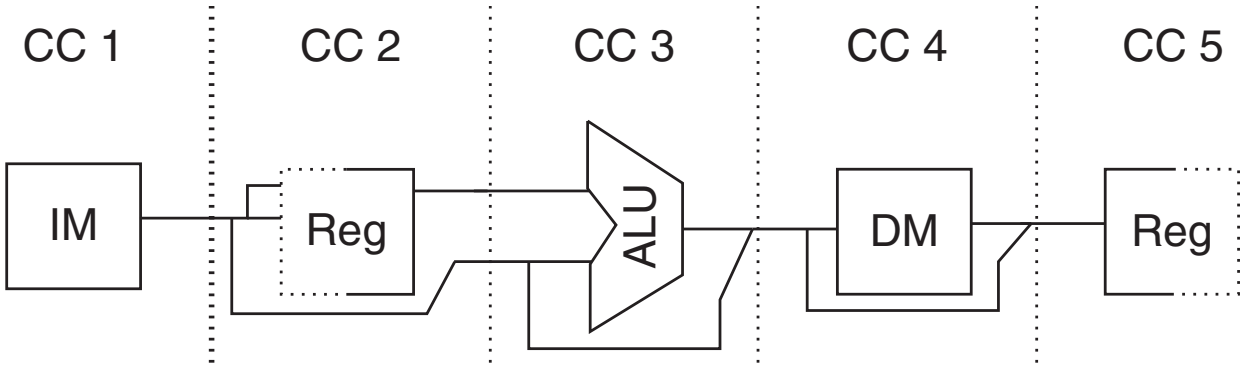
Instruction Execution of RISC

- Initial State: PC is set to point to the first instruction
- For each instruction, perform the following 5 steps:
  - Instruction Fetch (**IF**)
  - Instruction Decode/Register Read (**ID**)
  - Execution/Effective Address Calculation (**EX**)
  - Memory Access (**MEM**)
  - Write Back (**WB**)



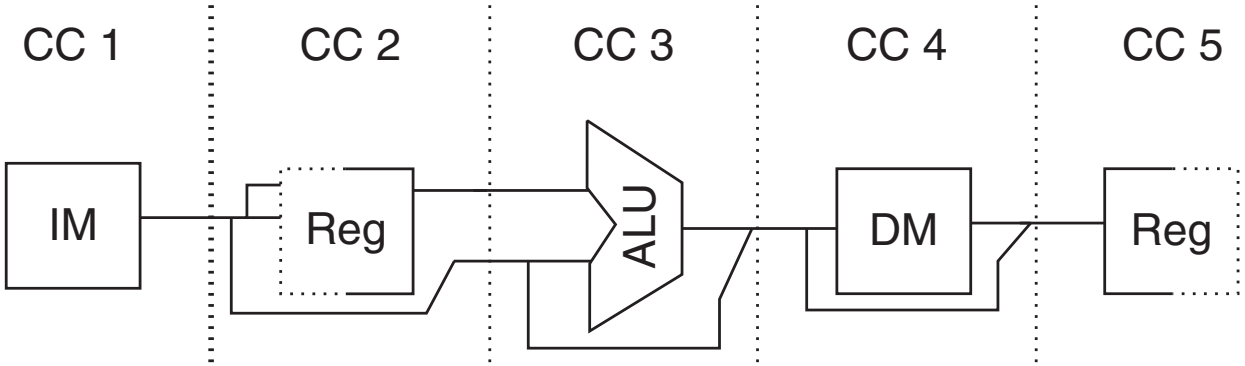
Instruction Execution

- **Instruction Fetch**:
  - Send PC to memory, assert MemRead signal
  - Instruction read out from memory
  - Place instruction in IR: IR ← [PC]
  - Update PC to next instruction: PC ← [PC] + 4



Instruction Execution

- **Instruction Decode**:
  - Instruction in IR decoded by control logic, instruction type and operands determined
  - Source operand registers read from general purpose register file

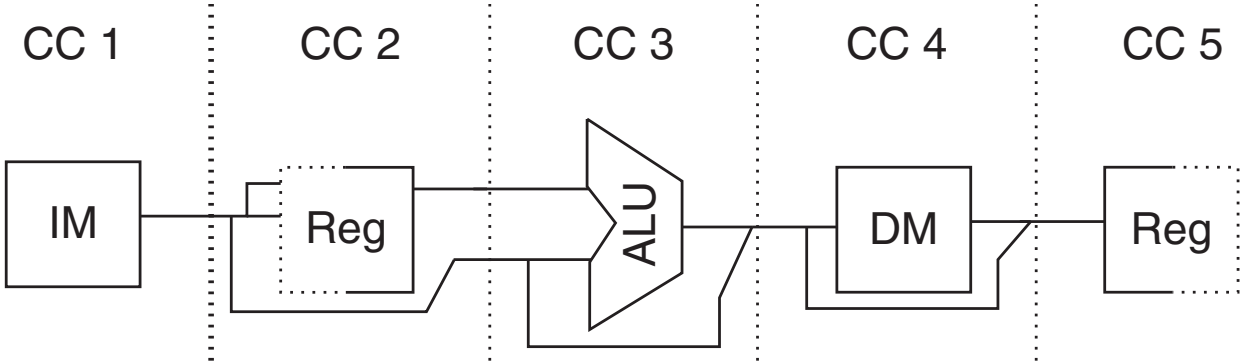


Instruction Execution

- **Execute**:
  - ALU operates on operands prepared in previous cycle
  - One of four functions depending upon opcode
  - Memory Reference
    - Form effective address from base register and immediate offset
    - ALU Output ← [A] + Imm
  - Register-Register ALU Instruction
    - ALU Output ← [A] **op** [B]
  - Register-Immediate ALU Instruction
    - ALU Output ← [A] **op** Imm
  - Branch
    - Compute branch target by adding Imm to PC
      - ALU Output ← [PC] + (Imm << 2)
    - Evaluate the branch condition

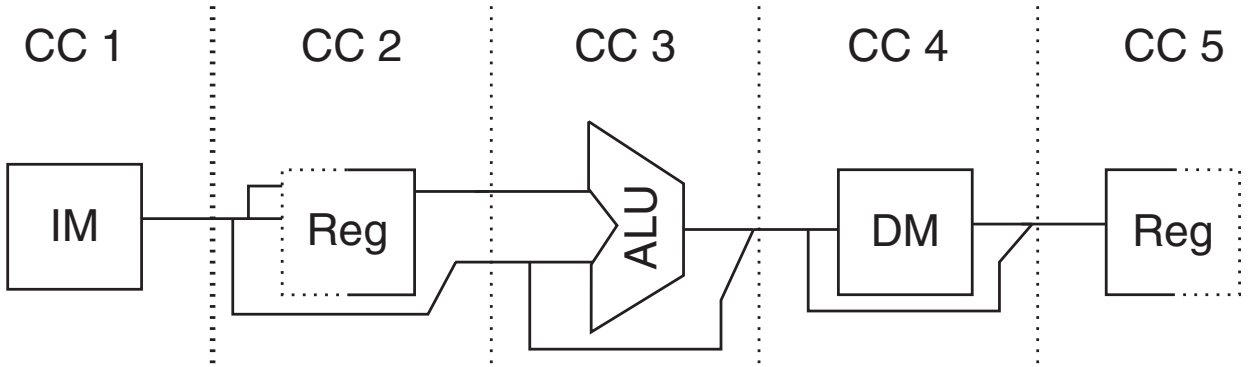
Instruction Execution

- **Memory Access**:
  - For load instructions, read data from memory
  - For store instructions, write data to memory



Instruction Execution

- **Write-back:**
  - Results written to destination register
  - Results from mem read or ALU



Instruction Execution – Example

- Add R3, R4, R5 ; R3←[R4]+[R5]
- Source registers: R4, R5 Destination register: R3
- Instruction steps:
  - **Fetch**: Fetch the instruction and increment the program counter
  - **Decode**: Decode the instruction in IR to determine the operation to be performed (add). Read the contents of registers R4 and R5
  - **Execute**: Compute the sum [R4] + [R5]
  - **Memory Access**: No action, since there are no memory operands
  - **Write-back**: Write the result into register R3

Instruction Execution – Example

- Load R5, X(R7) ; R5←[[R7]+X]
- Source register: R7 Destination register: R5
- Immediate value X is given in the instruction word
- Instruction steps:

  Fetch: Fetch the instruction and increment the program counter

  Decode: Decode the instruction in IR to determine the operation to be performed (load). Read the contents of register R7

  Execute: Add the immediate value X to the contents of R7

  Memory Access: Use the sum X+[R7] as the effective address of the source operand, read the contents of that location from memory

  Write-back: Write the data received from memory into register R5

Instruction Execution – Example

- Store R6, X(R8) ; Mem[X+[R8]]←[R6]
- Source registers: R6, R8 Destination register: None
- The immediate value X is given in the instruction word
- Instruction steps:
  - **Fetch**: Fetch the instruction and increment the program counter
  - **Decode**: Decode the instruction in IR to determine the operation to be performed (store). Read the contents of registers R6 and R8.
  - **Execute**: Compute the effective address X + [R8]
  - **Memory Access**: Store the contents of register R6 into memory location X + [R8]
  - **Writeback**: No action

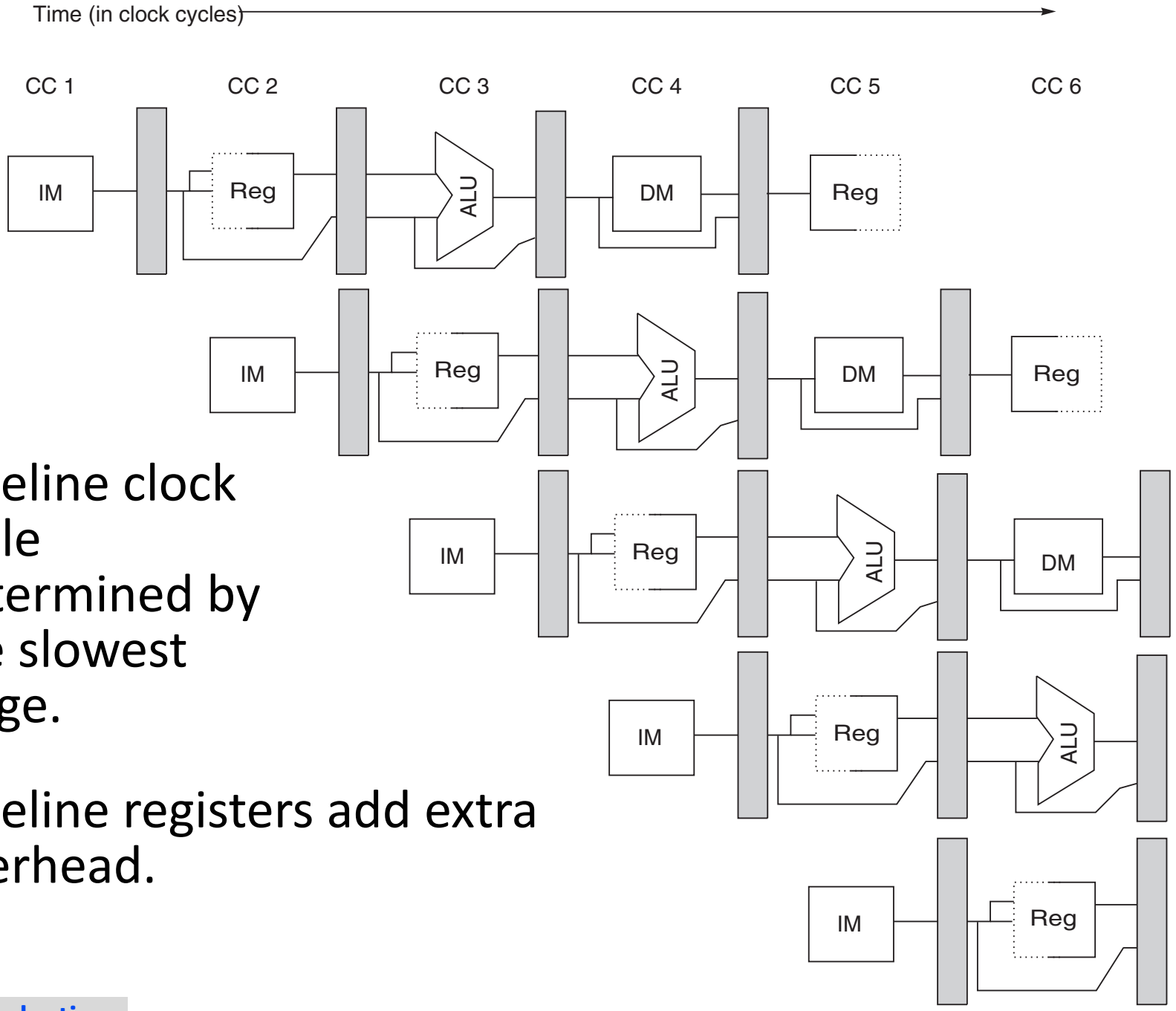
Basic Pipeline

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 =

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



- Pipeline clock cycle determined by the slowest stage.
- Pipeline registers add extra overhead.

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

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

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

$$= Number\ of\ pipeline\ stages$$

Pipeline Performance

- Example: A program consisting of 500 instructions is executed on a 5-stage processor. How many cycles would be required to complete the program. Assume ideal overlap in case of pipelining.
- Without pipelining:
  - Each instruction will require 5 cycles. There will be no overlap amongst successive instructions.
  - Number of cycles = 500 * 5 = 2500
- With pipelining:
  - Each pipeline stage will process a different instruction every cycle. First instruction will complete in 5 cycles, then one instruction will complete in every cycle, due to ideal overlap.
  - Number of cycles = 5 + ((500-1)*1) = 504
- Speedup for ideal pipelining = 2500/504 = 4.96

Pipeline Performance

- Problem: Consider a non-pipelined processor using the 5-stage datapath with 1 ns clock cycle. Assume that due to clock skew and pipeline registers, pipelining the processor adds 0.2 ns of overhead to the clock speed. How much speedup can we expect to gain from pipelining? Assume a balanced pipeline and ignore the pipeline fill and drain overheads. (A similar ex. in the book)
- Solution:
  - Without pipelining: Clock period = 1 ns, CPI = 5
  - With pipelining: Clock period = 1 + 0.2 = 1.2 ns, CPI = 1

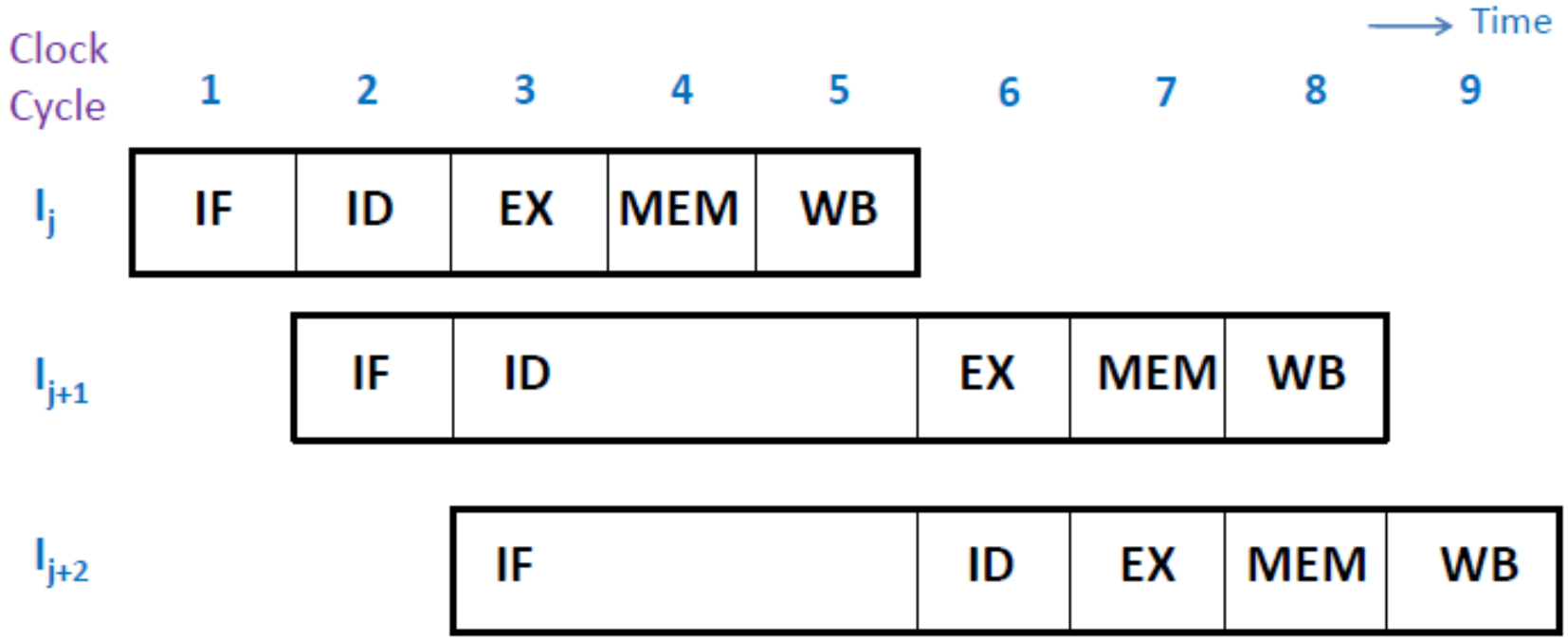
$$\text{Speedup from pipelining} = \frac{(Average\ time\ per\ instruction)_{non\_pipelined}}{(Average\ time\ per\ instruction)_{pipelined}}$$

$$= (1\ \text{ns} * 5) / (1.2\ \text{ns} * 1) = 5 / 1.2 = 4.17$$

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 Performance



- Assume that Instruction I_{j+1} is stalled in the decode stage for two extra cycles
- This will cause I_{j+2} to be stalled in the fetch stage, until I_{j+1} proceeds
- New instructions cannot enter the pipeline until I_{j+2} proceeds past the fetch stage after cycle 5 => execution time increases by two cycles

Pipeline Hazards

Dependences

- Independent instructions can be executed in parallel
- An instruction can depend on another in three ways
  - **Data dependences**
  - **Name dependences**
  - **Control dependences**

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer 8 bytes
        BNE     R1,R2,LOOP  ;branch R1!=R2
        DSUB    R1, R5, R6
```

Dependences limit how much parallelism can exploited

Pipeline Hazards

- Hazards prevent the next instruction in the instruction stream from executing during its designated clock cycle
- Three types of pipeline hazards
  - Structural hazard – a situation where two (or more) instructions require the use of a given hardware resource at the same time
  - Data hazard – any condition in which either the source or the destination operands of an instruction are not available, when needed in the pipeline
    - Instruction processing will be delayed until operands become available
  - Control hazard – a delay in the availability of an instruction or the memory address needed to fetch the instruction

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

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

Structural Hazards

- A situation where two (or more) instructions require the use of a given hardware resource at the same time
- Example: In the MIPS 5-stage pipeline, both the “IF” and “MEM” stages require memory access
- In the same cycle
  - A new instruction is fetched from memory in the IF stage
  - A “Load” instruction reads data from memory in the MEM stage
- What happens if the memory has only one read port?

Structural Hazards – Stalls

➜ Instructions ahead of the stall proceed to completion

➜ Stall delays instruction and those following it

	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				**Stall**	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Structural Hazards – Cost of Stalls

- Consider two pipelined processors. Suppose data references represent 40% of the instructions executed and that the ideal CPI of the pipelined processor (no structural hazard) is 1. Assume that the processor with the hazard has a clock rate that is 1.05 times higher than the hazard free processor and incurs a one cycle stall on structural hazards as in our example. Is the pipeline with or without the structural hazard faster, and by how much?
- Without structural hazard: Clock period = C, CPI = 1
- With structural hazard: Clock period = C/1.05, CPI = 1 + (0.4 * 1) = 1.4
- $(Execution\ Time)_{without\ structural\ hazard}/(Execution\ time)_{with\ structural\ hazard} = C * 1/((C/1.05)*1.4) = 1.05/1.4 = 0.75$
  - Processor without the structural hazard is faster

How to Reduce Structural Hazards

- **Key Idea: Add more hardware resources**
- How to avoid structural hazard on memory access during IF and MEM stages?
  - Separate instruction and data caches
- How to avoid structural hazard on register access during ID and WB stages?
  - Dual ported register file
  - Write early in WB stage + Read late in ID stage
- Why no structural hazard on ALU during IF (PC ← PC + 4) and MEM stages?
  - PC + 4 in IF stage uses a dedicated adder, not the ALU

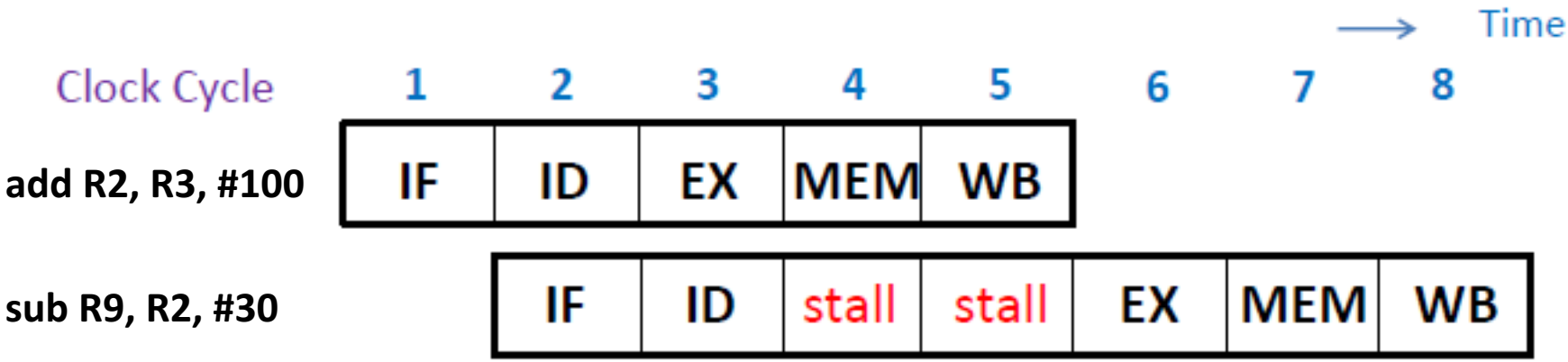
Data Hazards

- 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
sub 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



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

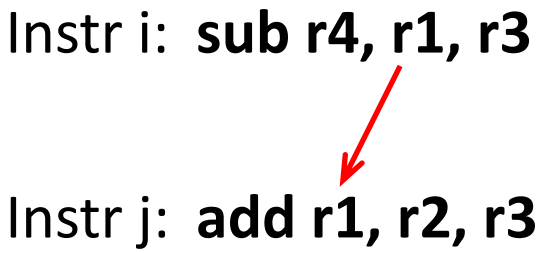
Instr i: **add r1, r2, r3**

Instr j: **sub r4, r1, r3**

➜ This hazard results from a true dependence: data flow from instruction *i* to instruction *j*

Types of Data Hazards - WAR

- $Inst_j$ tries to write operand before $Inst_i$ reads it (j > i)



- This is an anti-dependence (not a true dependence)
  - Arises from the reuse of register "r1"
  - If Instr i writes to r1 before Instr j reads r1, then Instr i will use the value written by Instr j => Incorrect behavior
- WAR hazards cannot happen in MIPS 5-stage pipeline:
  - Instructions are executed in order
  - Register reads happen earlier (stage-2) than register writes (stage-5)

Types of Data Hazards - WAW

- $Instr_j$ writes to operand before $Instr_i$ writes to it (j > i)

  Instr i: **sub r1, r4, r3**

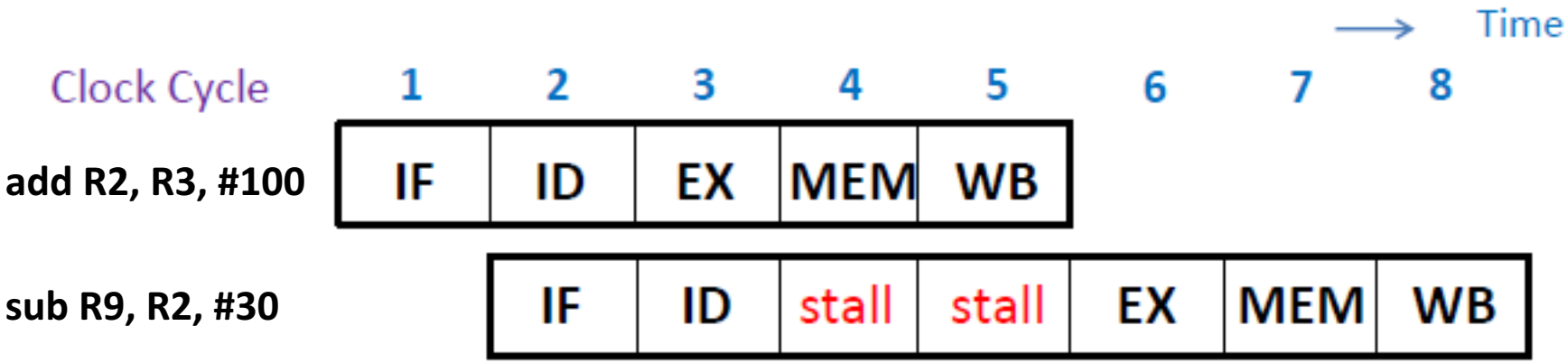
  

  Instr j: **add r1, r2, r3**

  Instr k: **mul r6, r1, r7**

- This is an output dependence (not a true dependence)
  - Arises from the reuse of register r1
  - If $Instr_j$ writes to r1 before $Instr_i$ then $Instr_k$ will use the value written by $Instr_i$ => violation of program order
- WAW hazards cannot happen in MIPS 5-stage pipeline because:
  - Instructions are executed in order
  - Register writes are always in stage-5

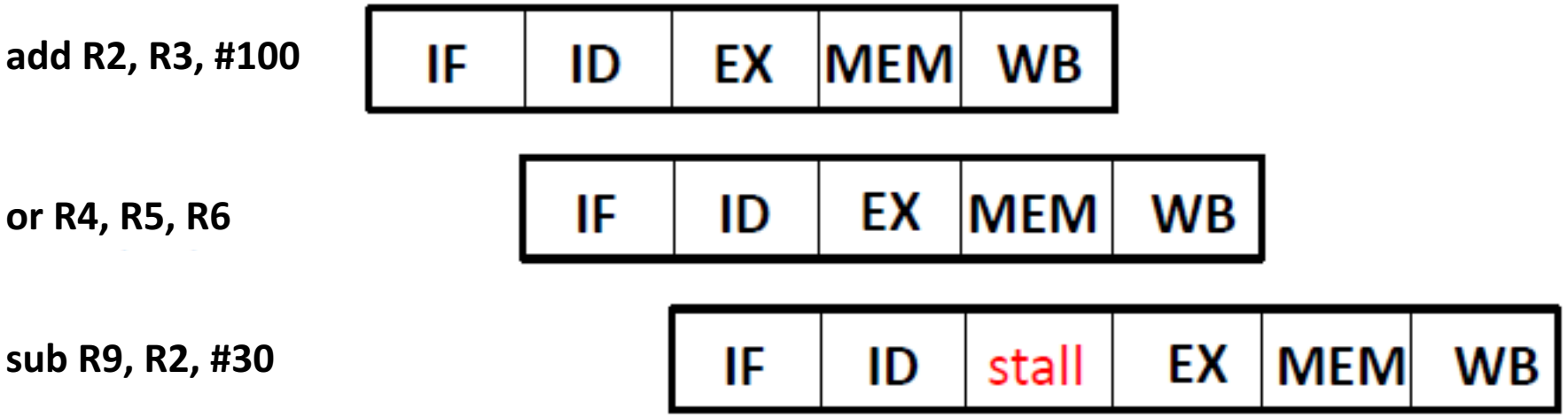
Data Hazards – Stall Penalty



➜ 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



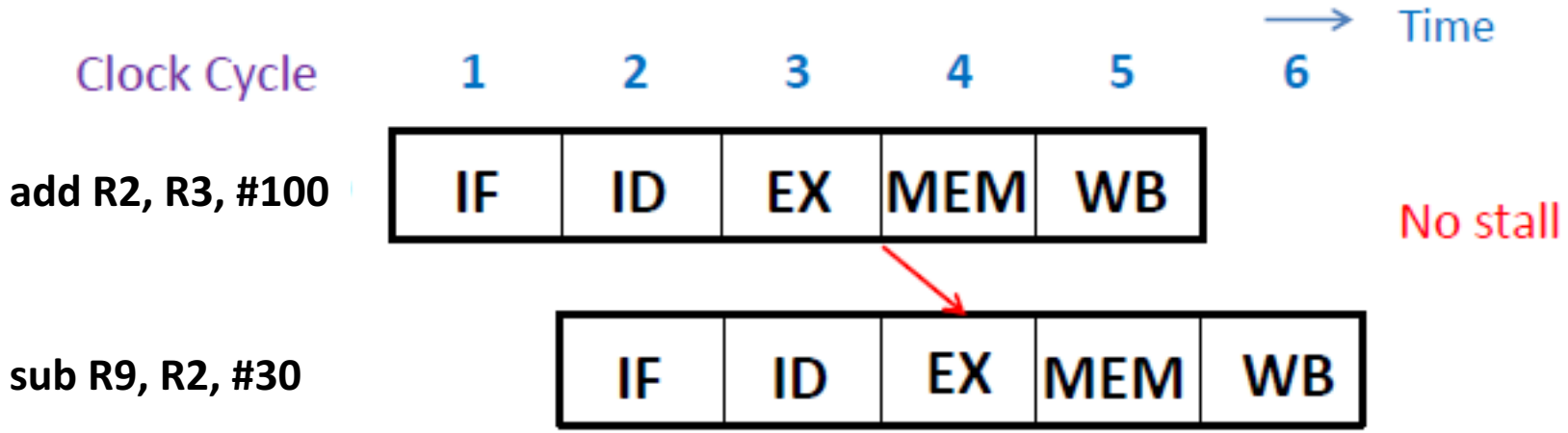
- What happens if the producer and dependent instructions are separated by an intermediate instruction?
- In this case, stall penalty = 1
- Stall penalty depends upon the **distance** between the producer and dependent instruction

Data Hazards – Stall Penalty

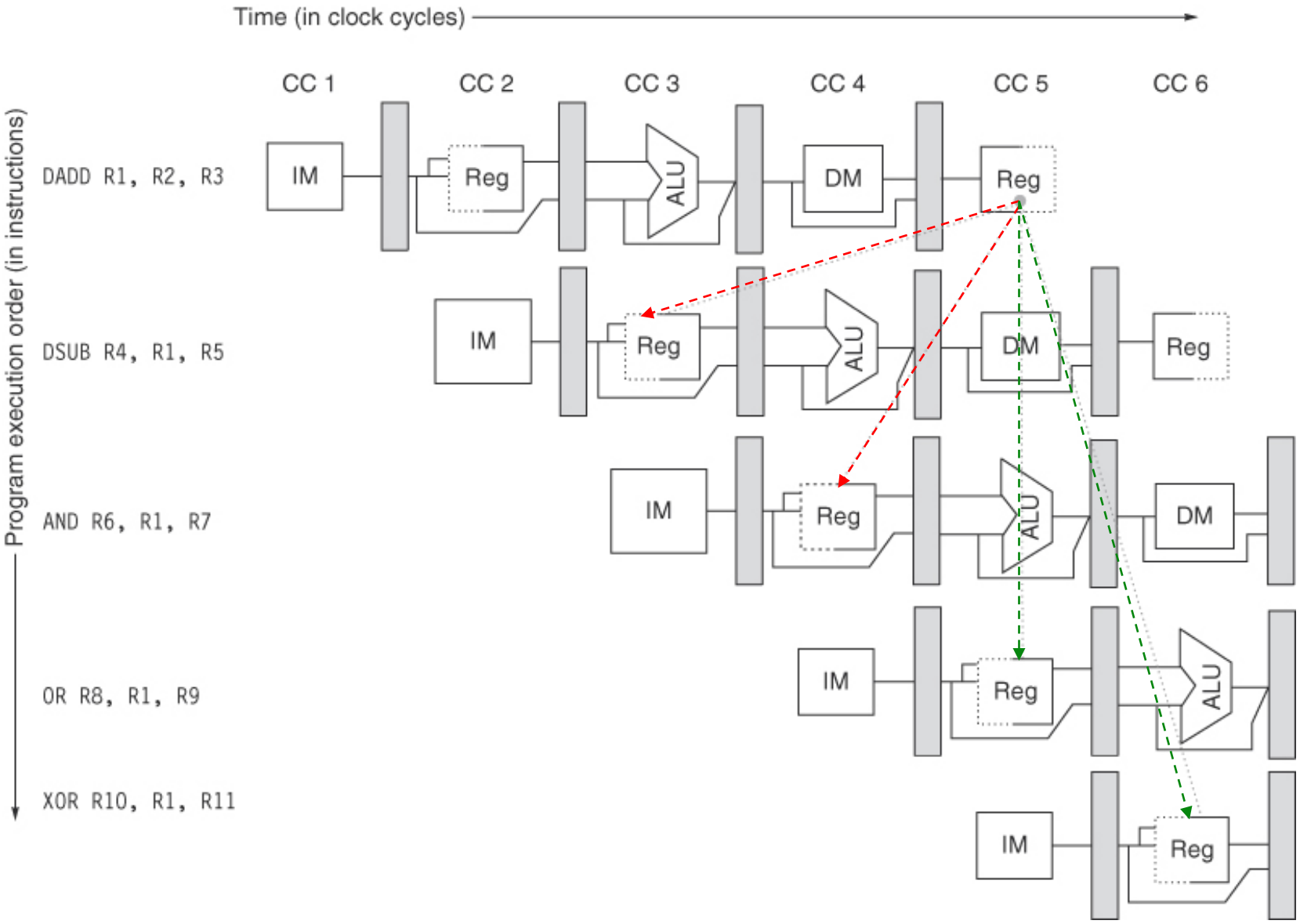
- Frequent stalls caused by data hazards can impact the performance significantly.
- **Example**: If every 5 instructions have 2 stall cycles due to a data hazard, then the CPI can increase by (1/5)*2 = 0.4 stall cycles
- Would like to avoid stalls caused by RAW hazards
  - Technique: **forwarding**
- Also called “bypassing” or “short-circuiting”
  - **Key Idea: Forward results from later stages in the pipeline to earlier stages**
  - *i.e.* Move data to where it is needed when it becomes available

Reduce Data Hazard Stalls – Forwarding

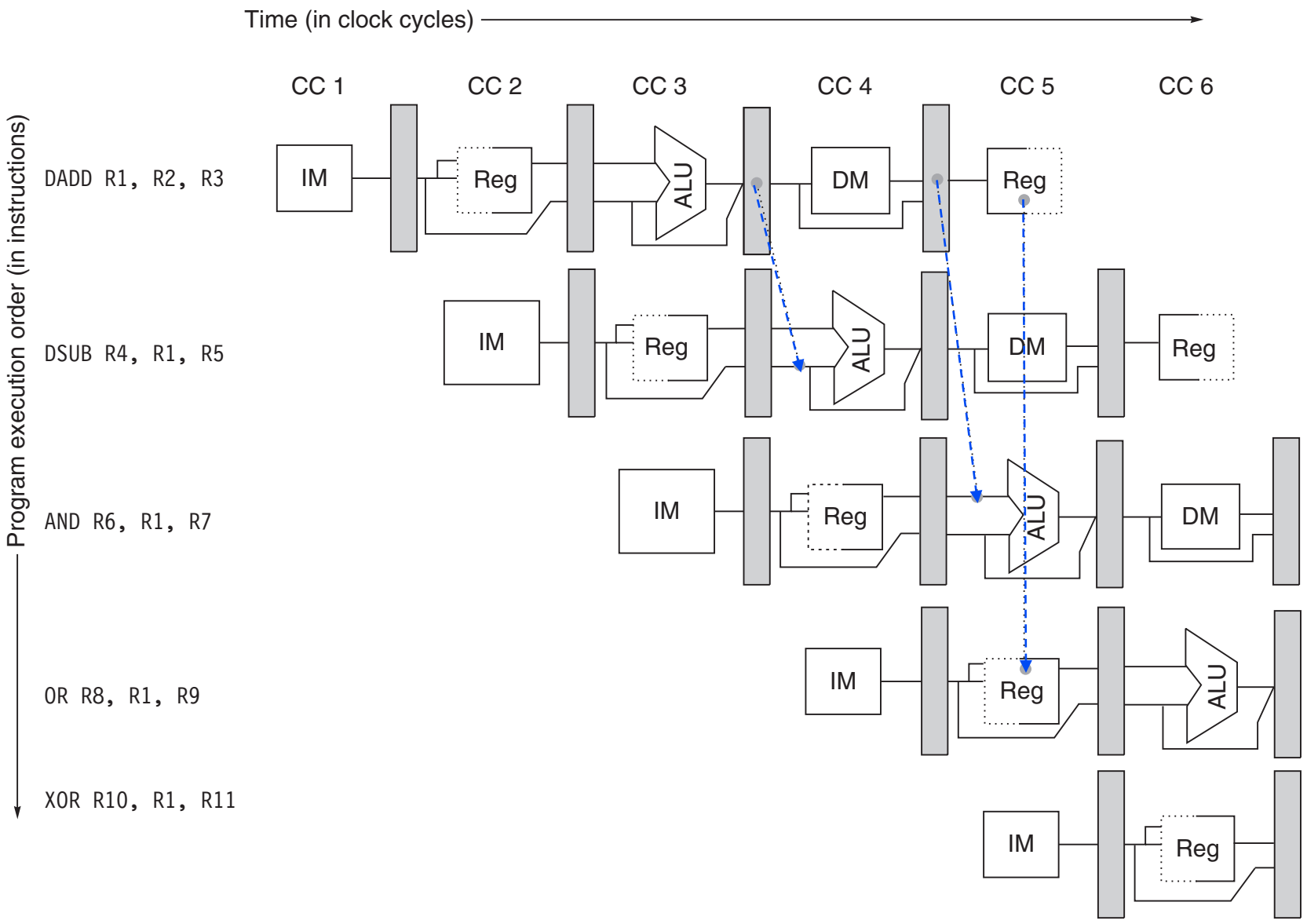
- Consider the two instructions discussed in the previous example
  - The result of producer instruction is actually available after the completion of EX stage (cycle 3), when the ALU completes its computation
  - Instead of stalling the dependent instruction, the hardware can forward the result from 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



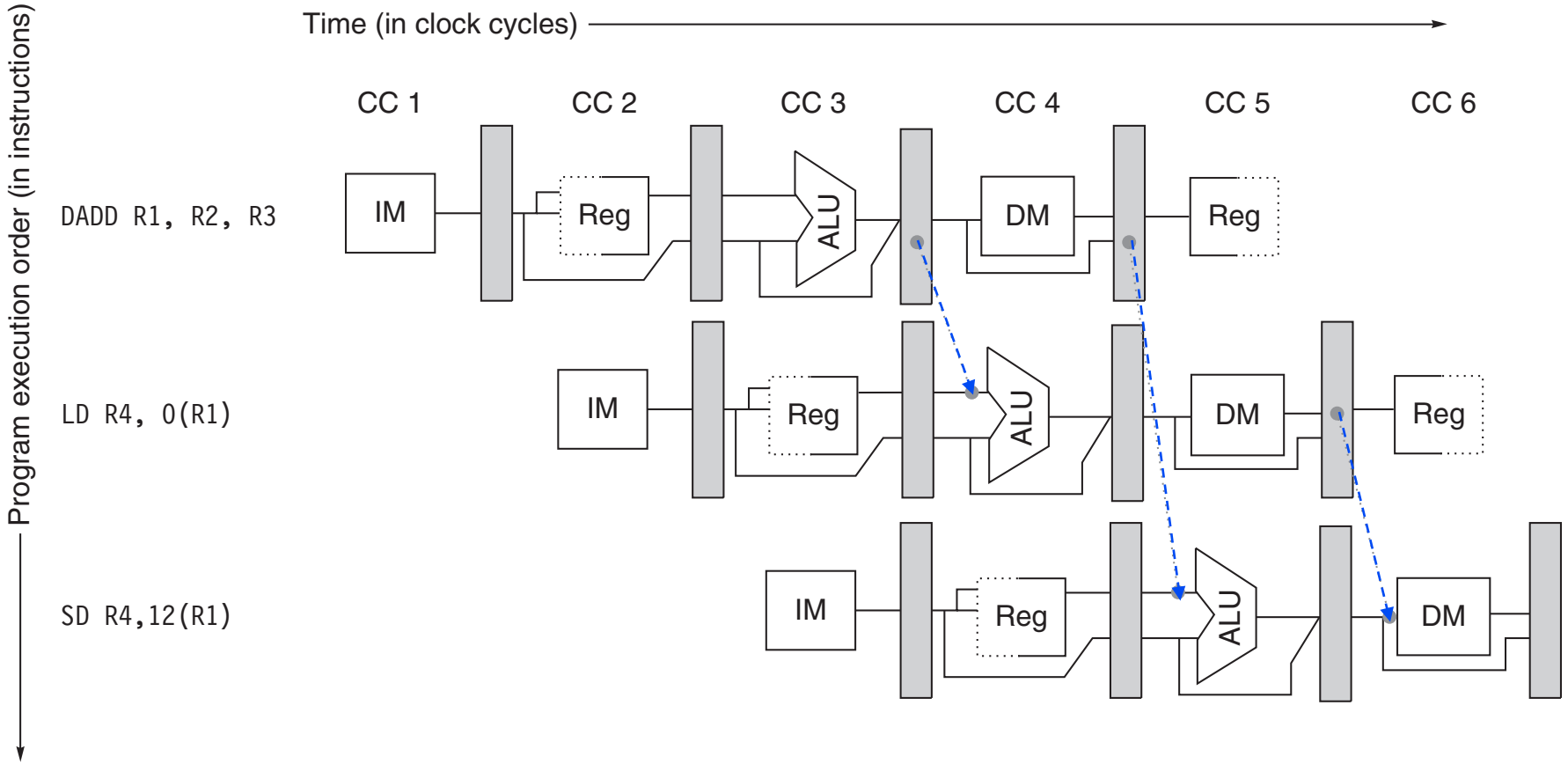
Pipeline Hazards – An Example



Reduce Data Hazard Stalls – Forwarding

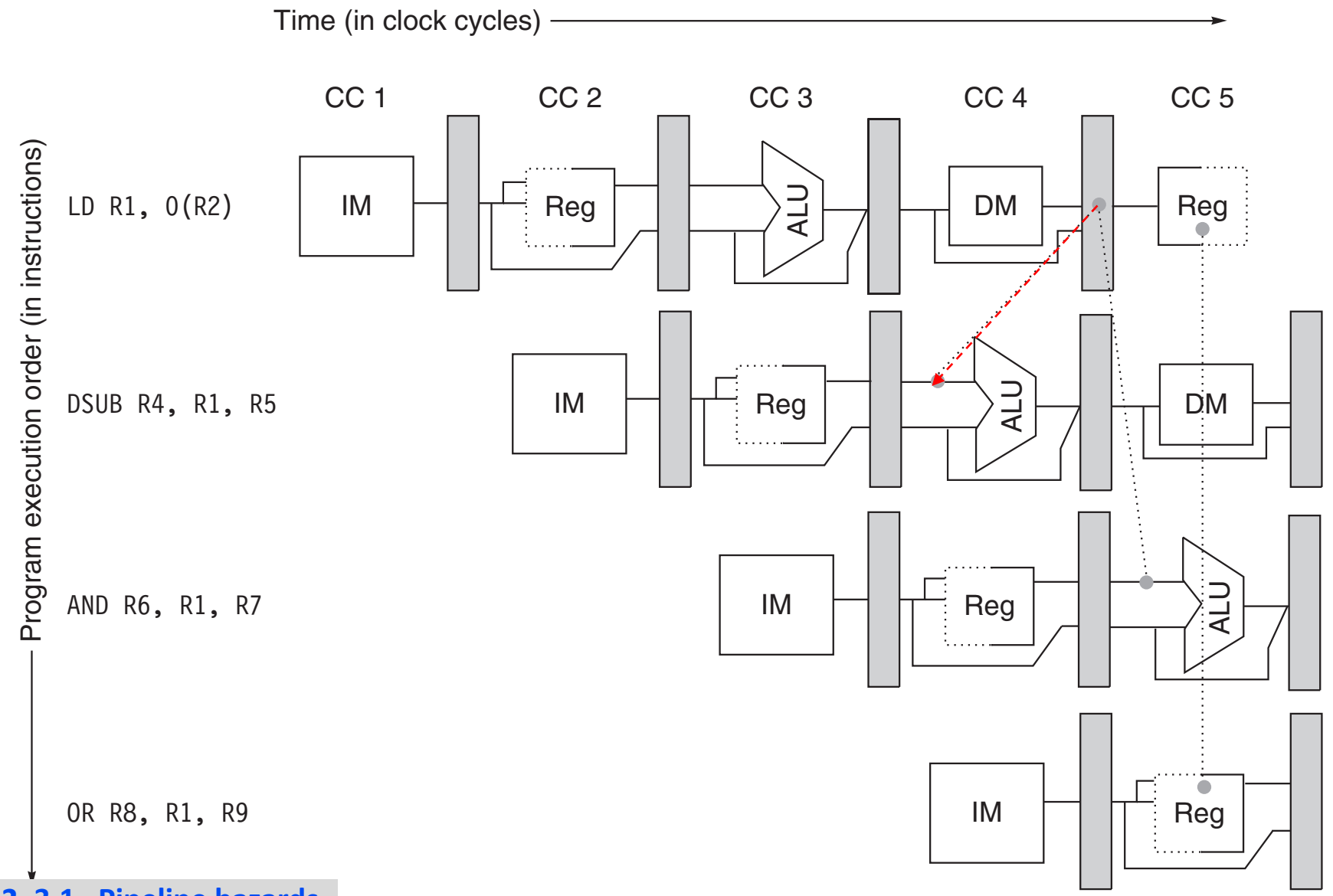


Reduce Data Hazard Stalls – Forwarding



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 – 1

➜ Consider a conditional branch instruction such as

```
loop:  ld R1, 0(R6)
       ...
       BEQ R5, R6, loop
       add R2, R4, R5
```

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

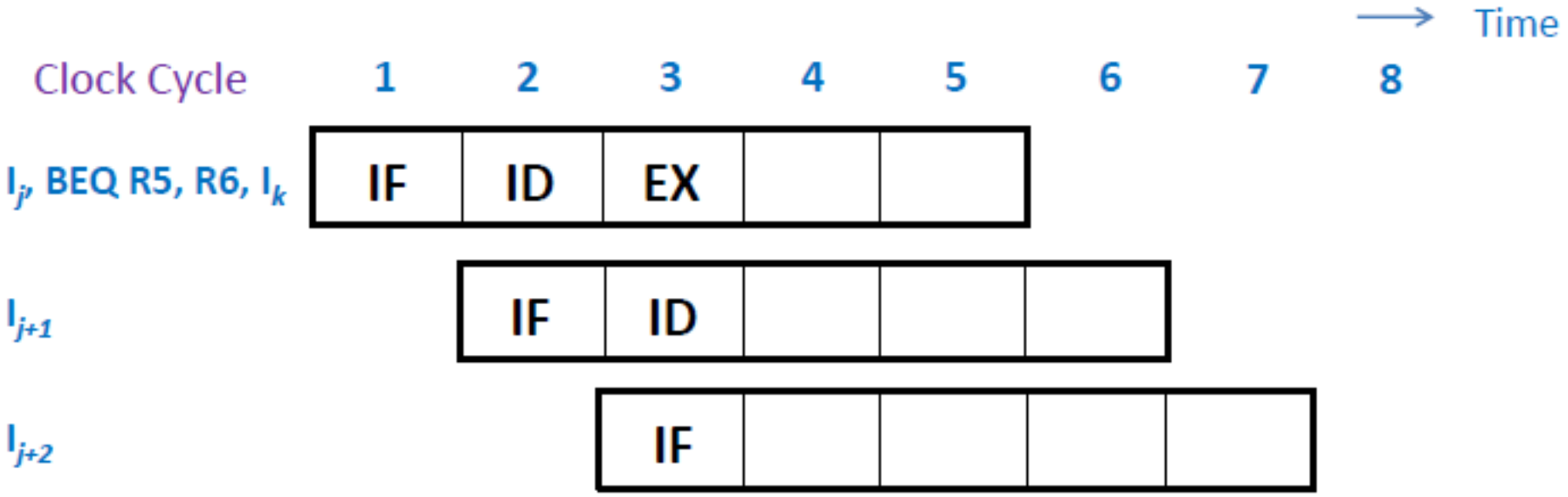
  ➜ also, compute branch target address PC + loop

Conditional Branches – 2

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

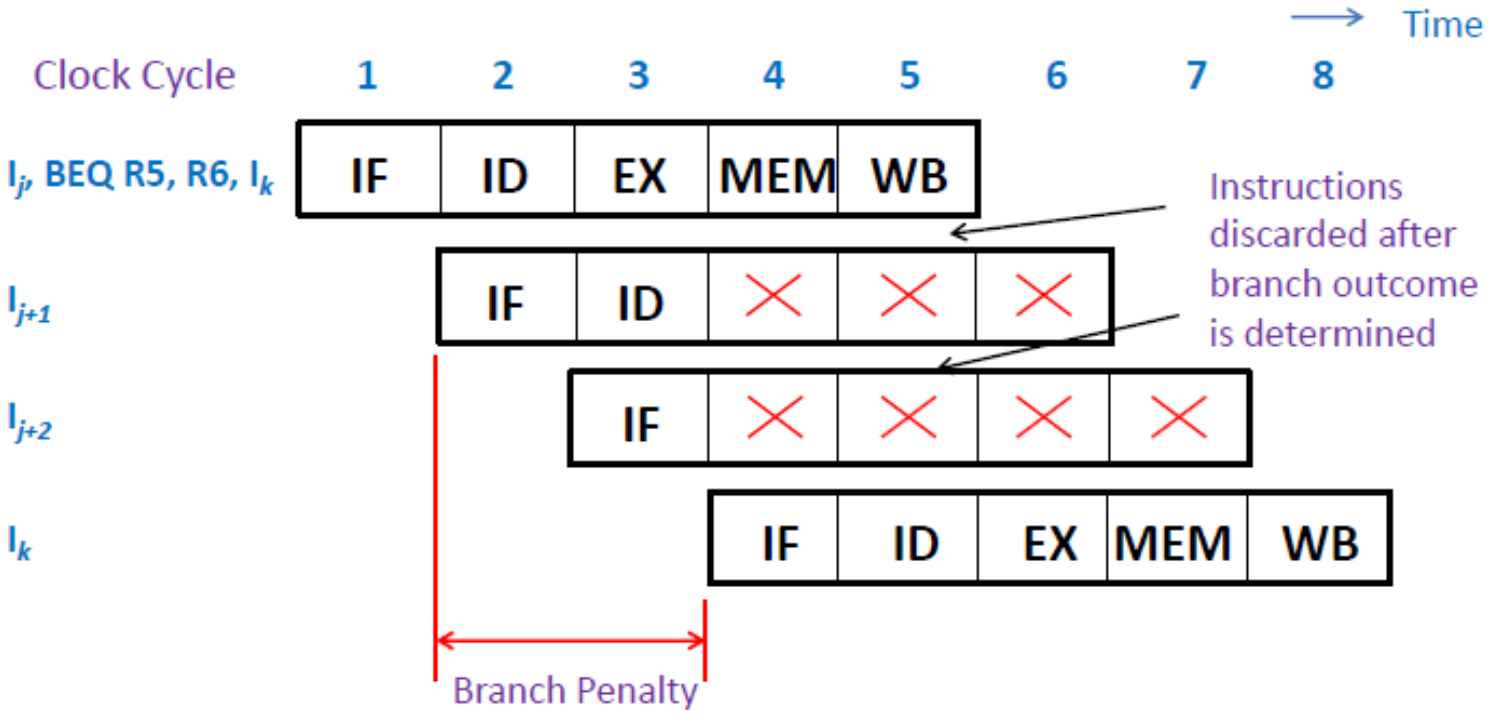
- If the branch is **taken**
  - Cannot fetch the branch target before the start of 4th cycle
  - No useful instruction fetched in cycles 2 and 3→ Control Hazard
  - Stall penalty = 2 cycles

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

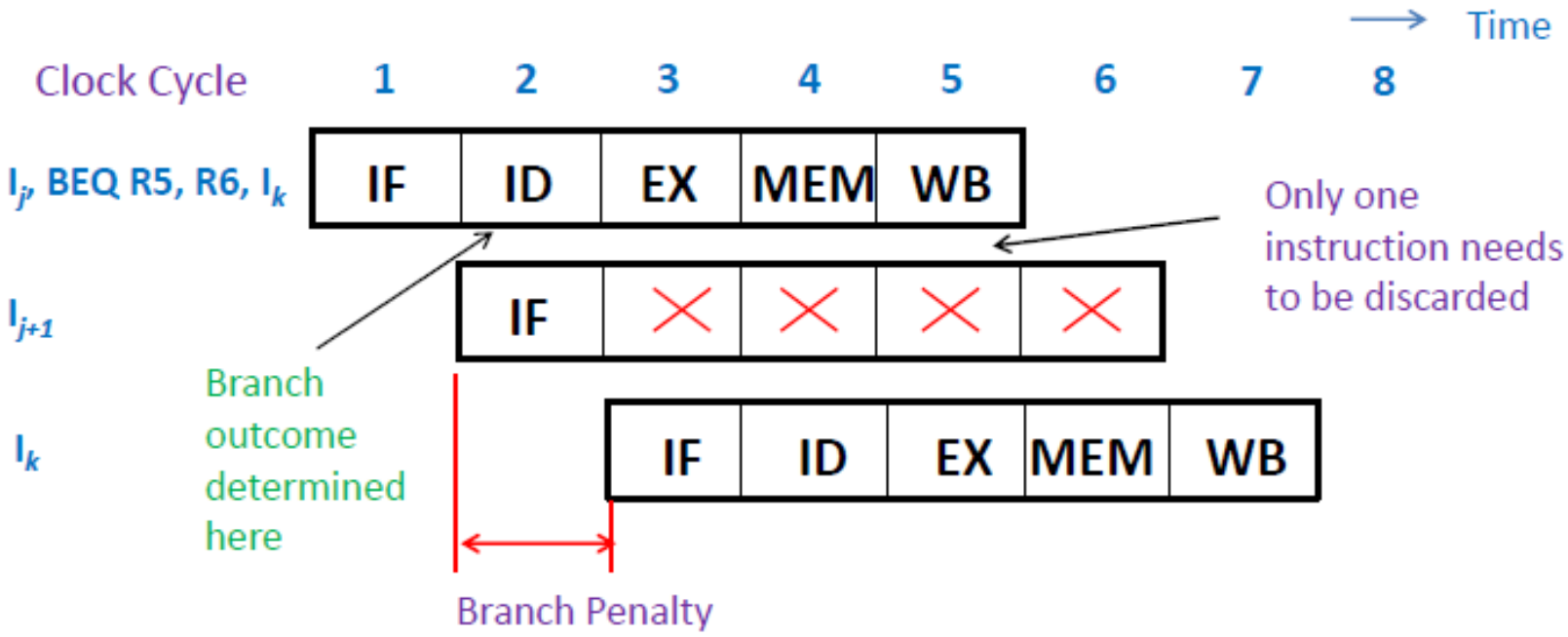


- Control transfers to branch target (I_k)
- Instructions I_{j+1} and I_{j+2} flushed from the pipeline
- Branch Penalty in this case is 2 cycles
- What would be the Branch Penalty if the branch was not taken?

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 – 1

- **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?

Reducing Pipeline Branch Penalties – 2

➜ Predicted-not-taken

Untaken branch instruction	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

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

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

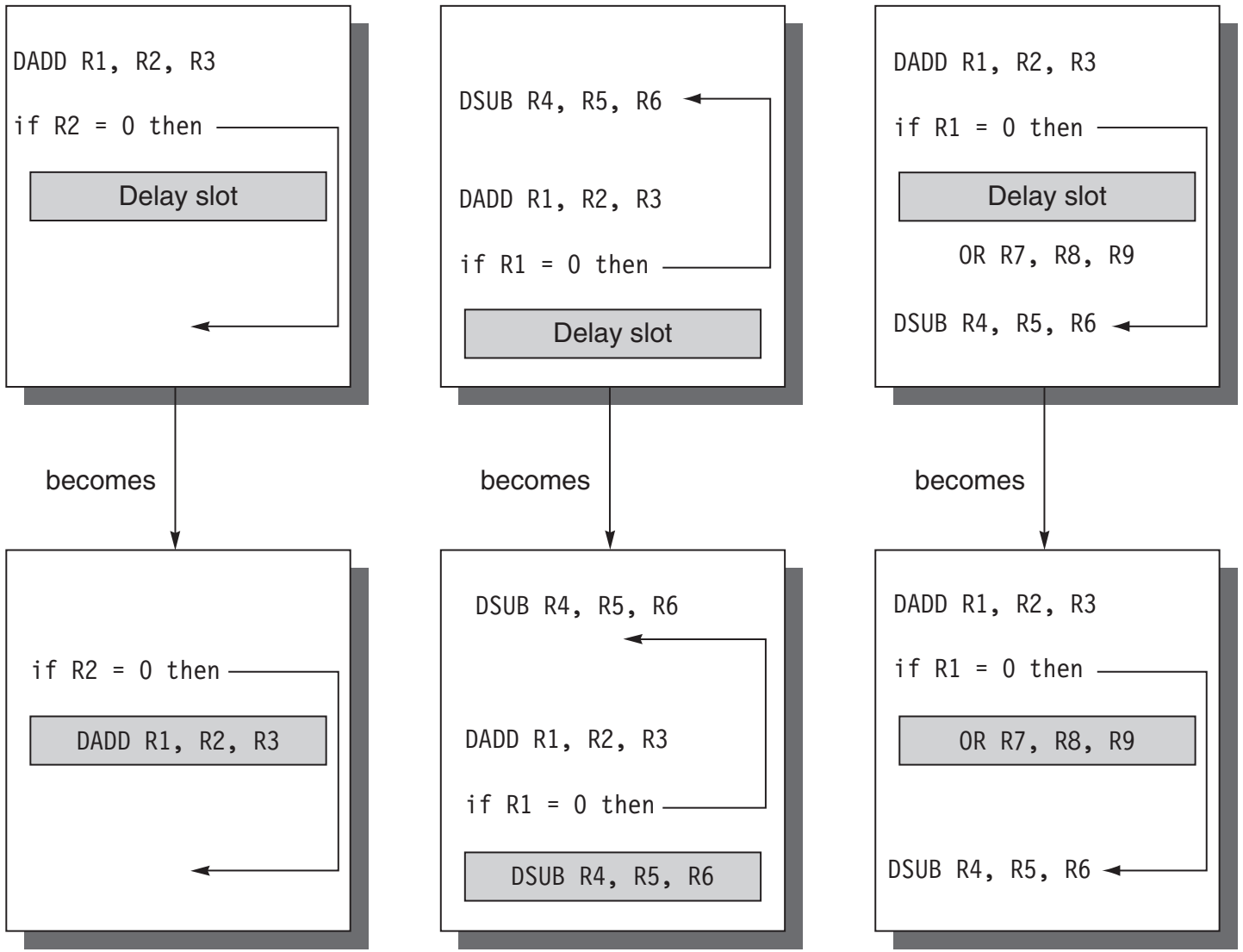
Sequential successor - in *branch delay slot*

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

Example

Suppose that branch penalty is 1 cycle.

```
        SD      R7, 0(R8)
        BEQ     R5, R6, IF
        LD      R2, 16(R1)
        DADD    R3, R4, R2
IF:     LD      R2, 64(R1)
        DSUB    R3, R4, R2
```

Branch hazards?

Data hazards?

Performance of Branch Schemes

$$\text{speedup} = \frac{\text{pipeline depth}}{1 + \text{stall cycles from branches}}$$

$$\text{stall cycles from branches} = \text{branch freq} \times \text{branch penalty}$$

$$\text{speedup} = \frac{\text{pipeline depth}}{1 + \text{branch freq} \times \text{branchpenalty}}$$

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

Another Example

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

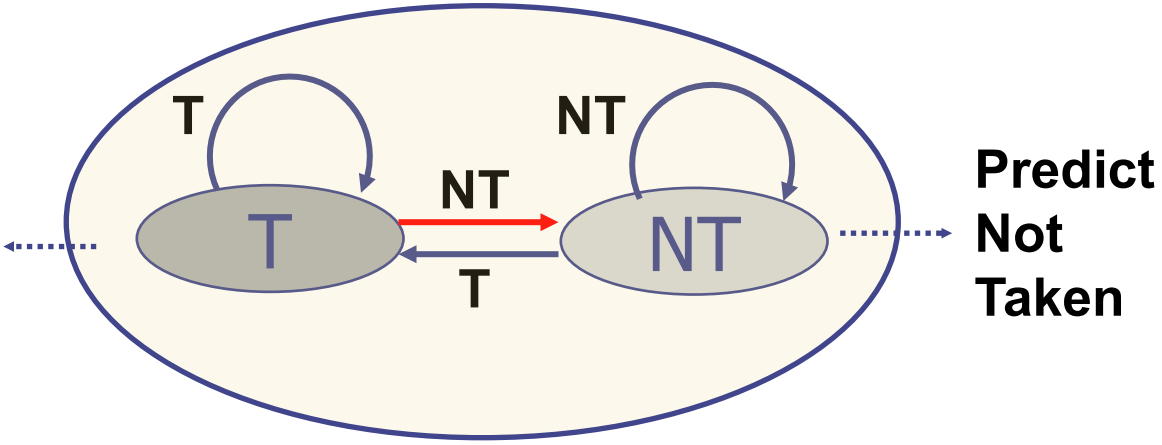
	Additions to the CPI from branch costs			
Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Speedup of “predicted-not-taken” over “stall pipeline” = 1.56 / 1.38 = 1.13

Branch Prediction

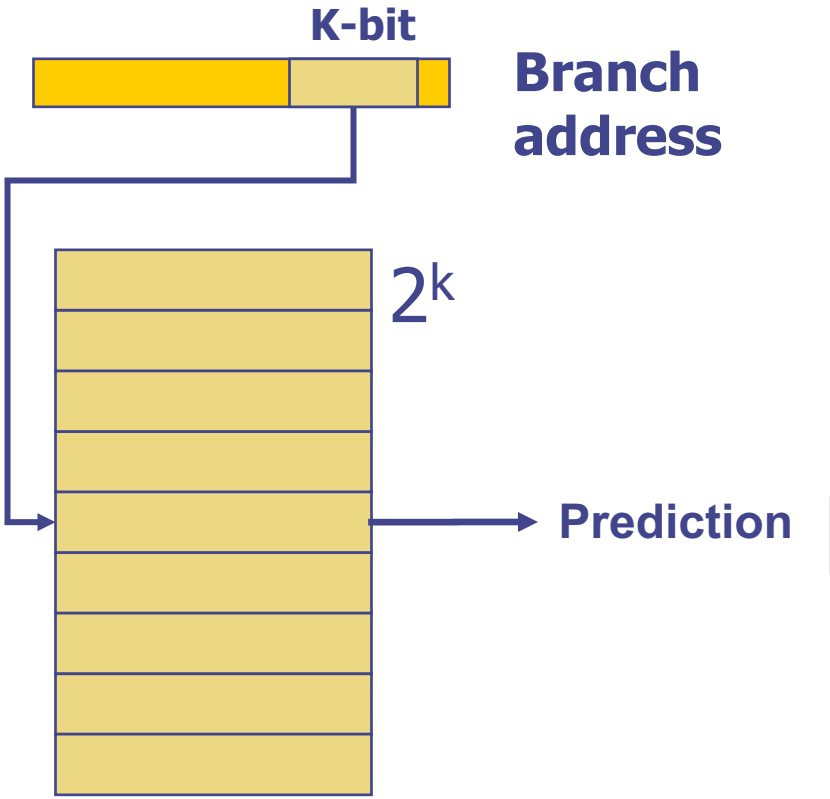
- **Static Prediction**
  - individual branches biased toward taken or not-taken
  - Opcode based
  - Displacement based (forward not taken, backward taken)
  - Compiler directed (branch likely, branch not likely)
- **Dynamic Prediction**
  - 1 bit predictor - remember last taken/not taken per branch
  - 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 –Why?
    - Multiple branches may share the same bit
  - 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



- BHT: Branch Prediction Buffer in textbook
- Can use only one 1-bit predictor
  - accuracy is low
- BHT: use a table of simple predictors, indexed by bits from PC
  - Similar to direct mapped cache
  - More entries, more cost, but less conflicts, higher accuracy
  - BHT can contain complex predictors

What about PCs of two branches having the same lower bits?



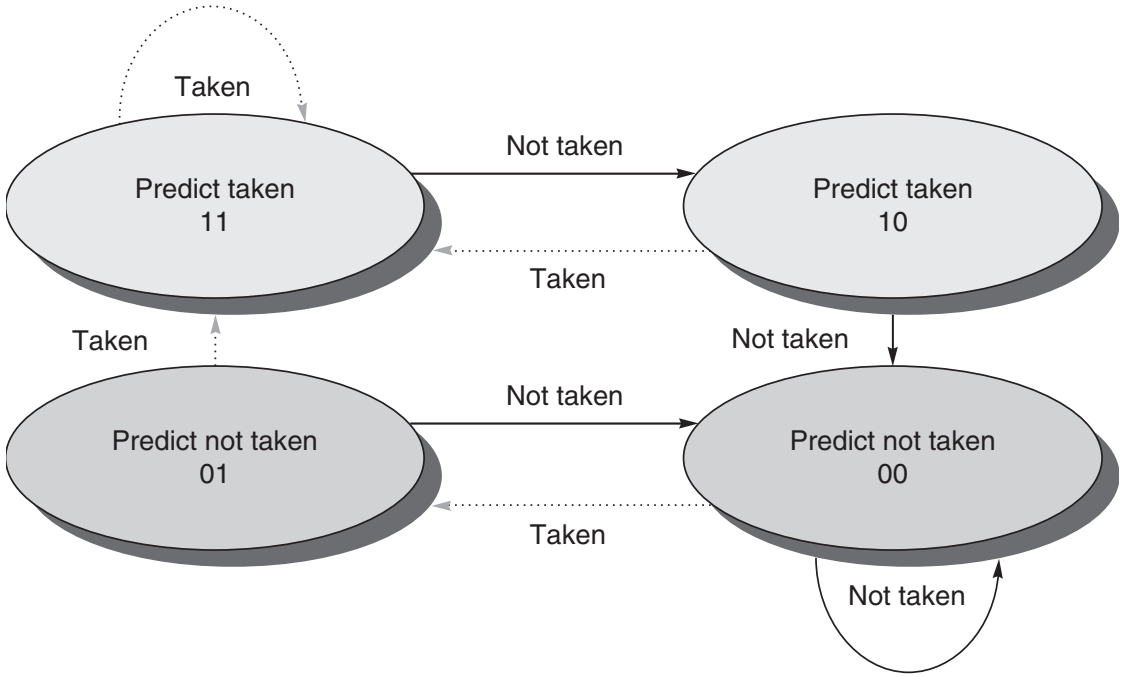
1-Bit Predictor

- 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;
}
```

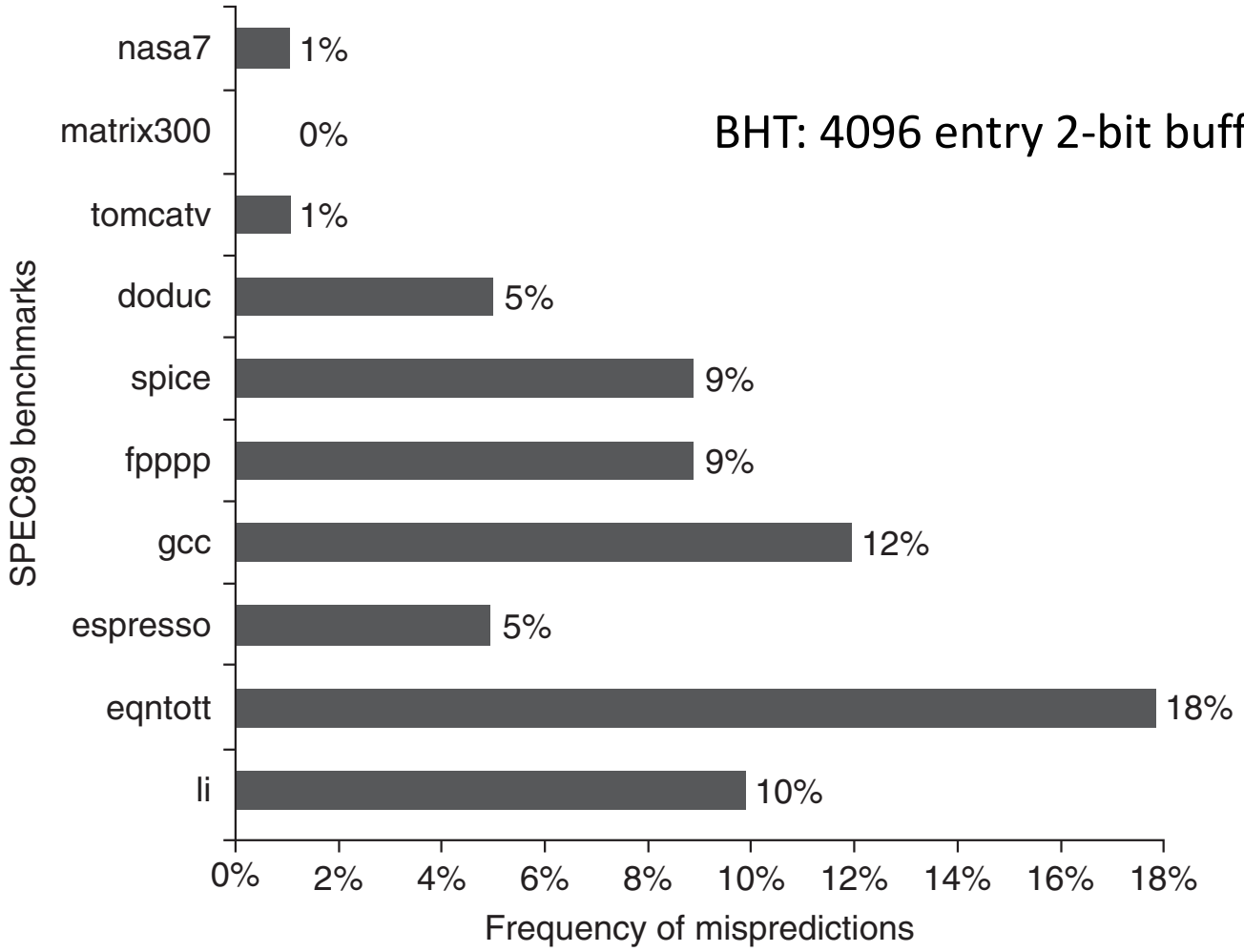
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Mis-predict *twice* every time the inner loop is executed

2-Bit Predictor

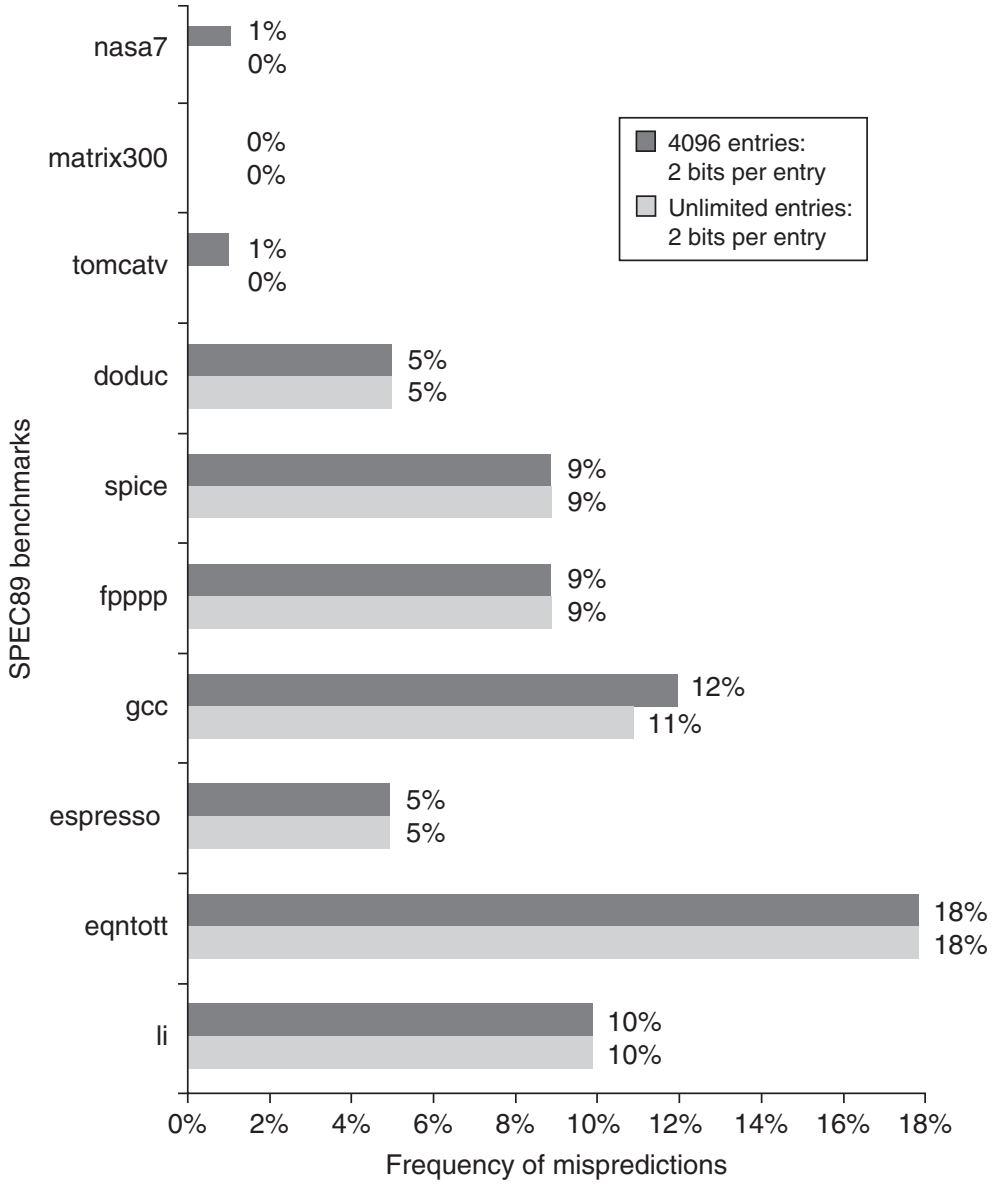


- Solution: 2-bit scheme where change prediction only if get mis-prediction *twice consecutively*
- Other solutions: correlating predictor, tournament predictors

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) {…}
```

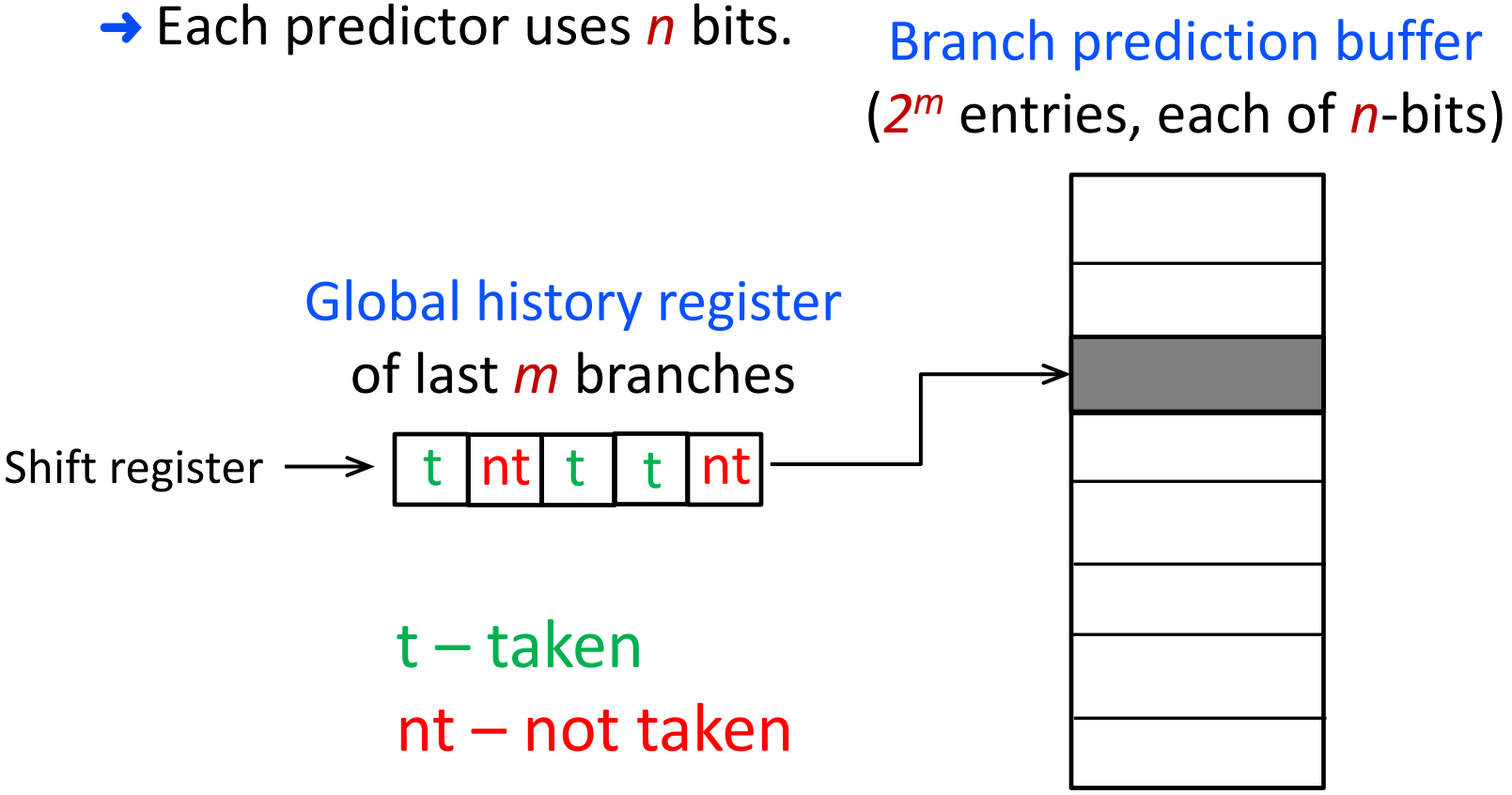
```
        DADDIU   R3,R1,#–2
        BNEZ     R3,L1          ;branch b1   (aa!=2)
        DADD     R1,R0,R0       ;aa=0
L1:     DADDIU   R3,R2,#–2
        BNEZ     R3,L2          ;branch b2   (bb!=2)
        DADD     R2,R0,R0       ;bb=0
L2:     DSUBU    R3,R1,R2       ;R3=aa-bb
        BEQZ     R3,L3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

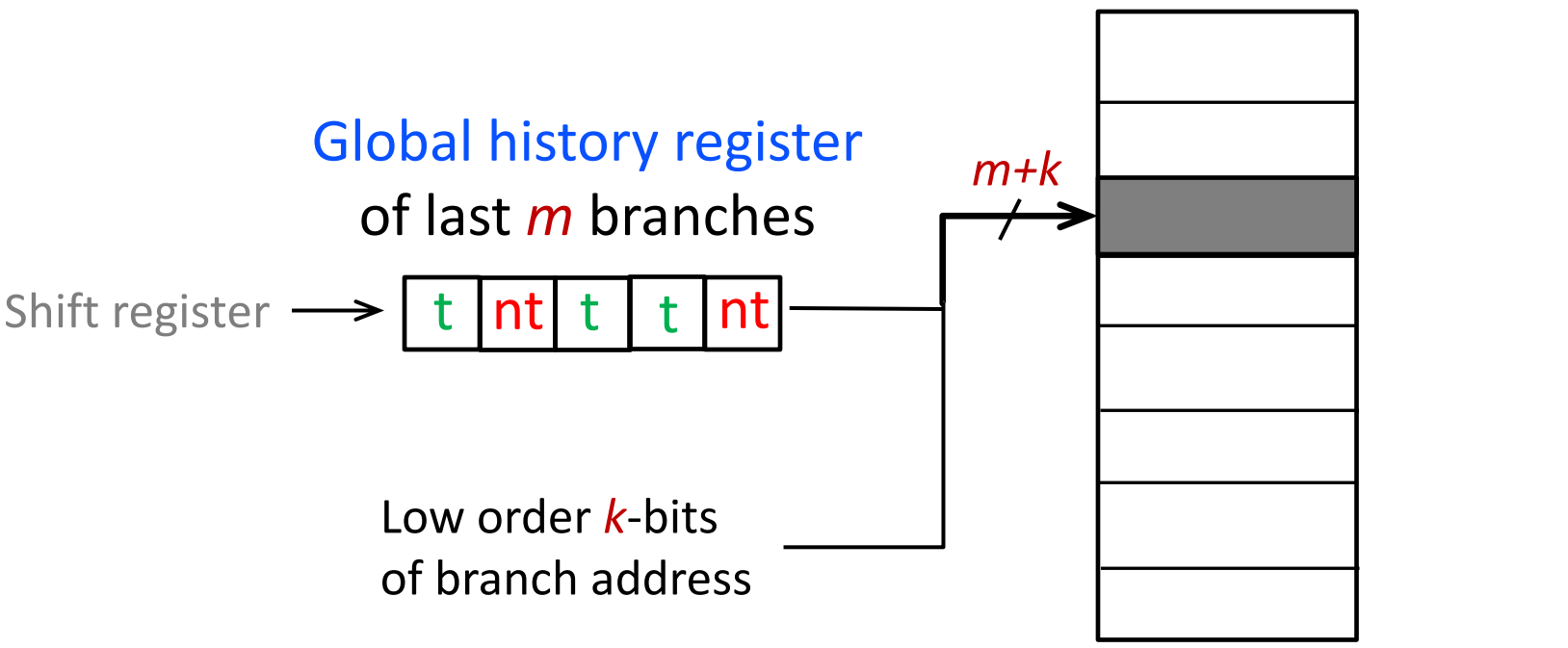
Correlating Branch Predictors

- Use history of last *m* branches to predict the current branch.
  - Each predictor uses *n* bits.

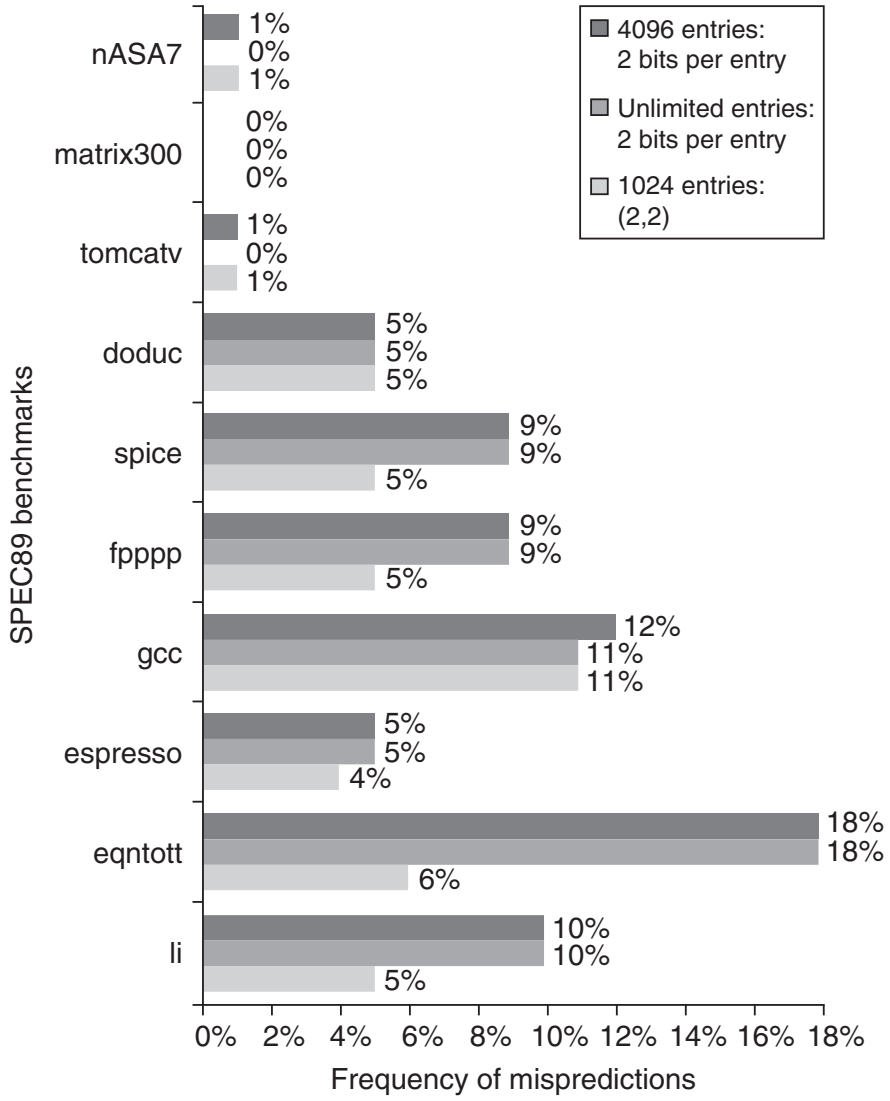


Correlating Branch Predictors

- Use history of last *m* branches to predict the current branch.
  - Each predictor uses *n* bits.



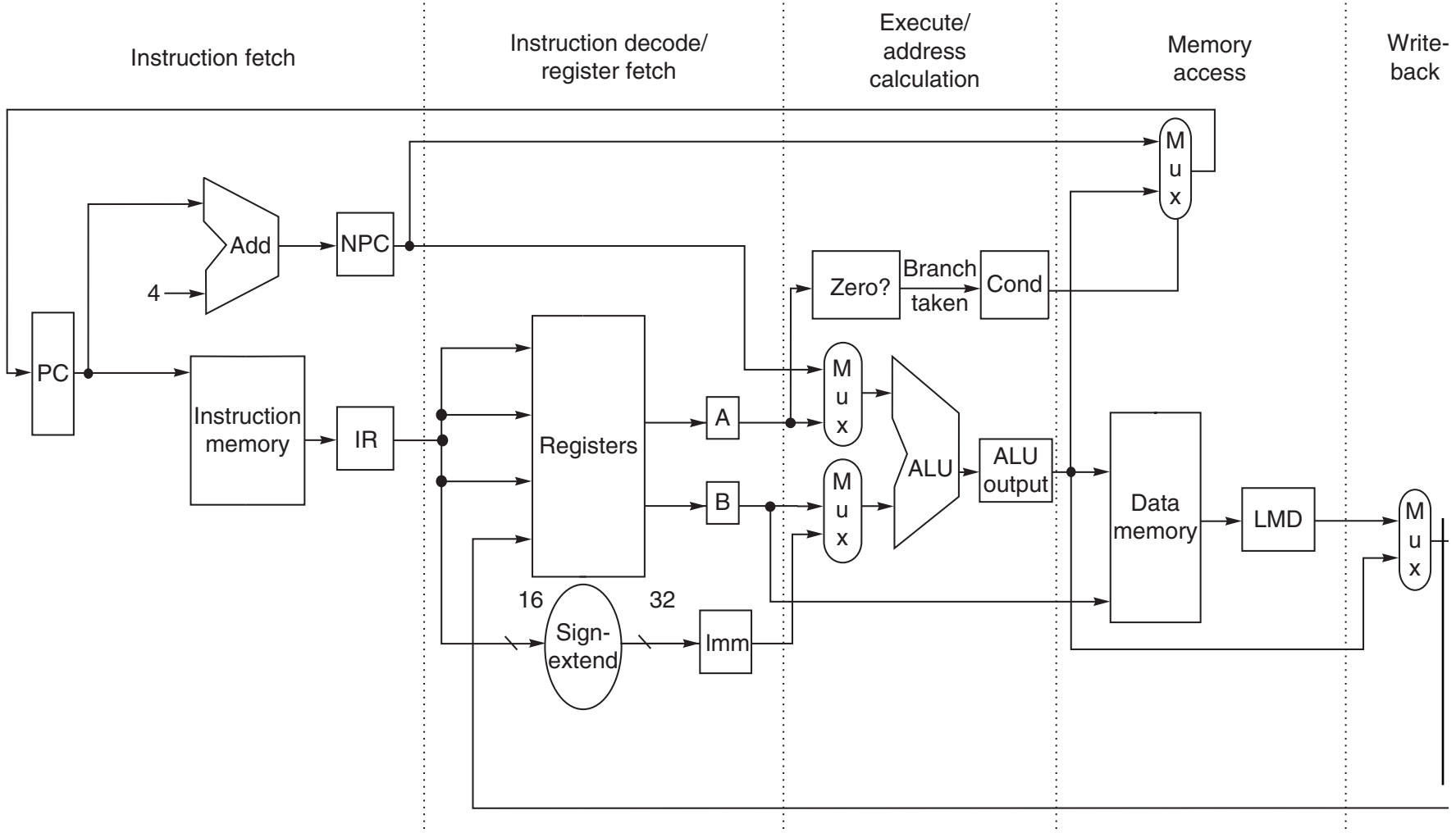
Correlating Branch Predictors



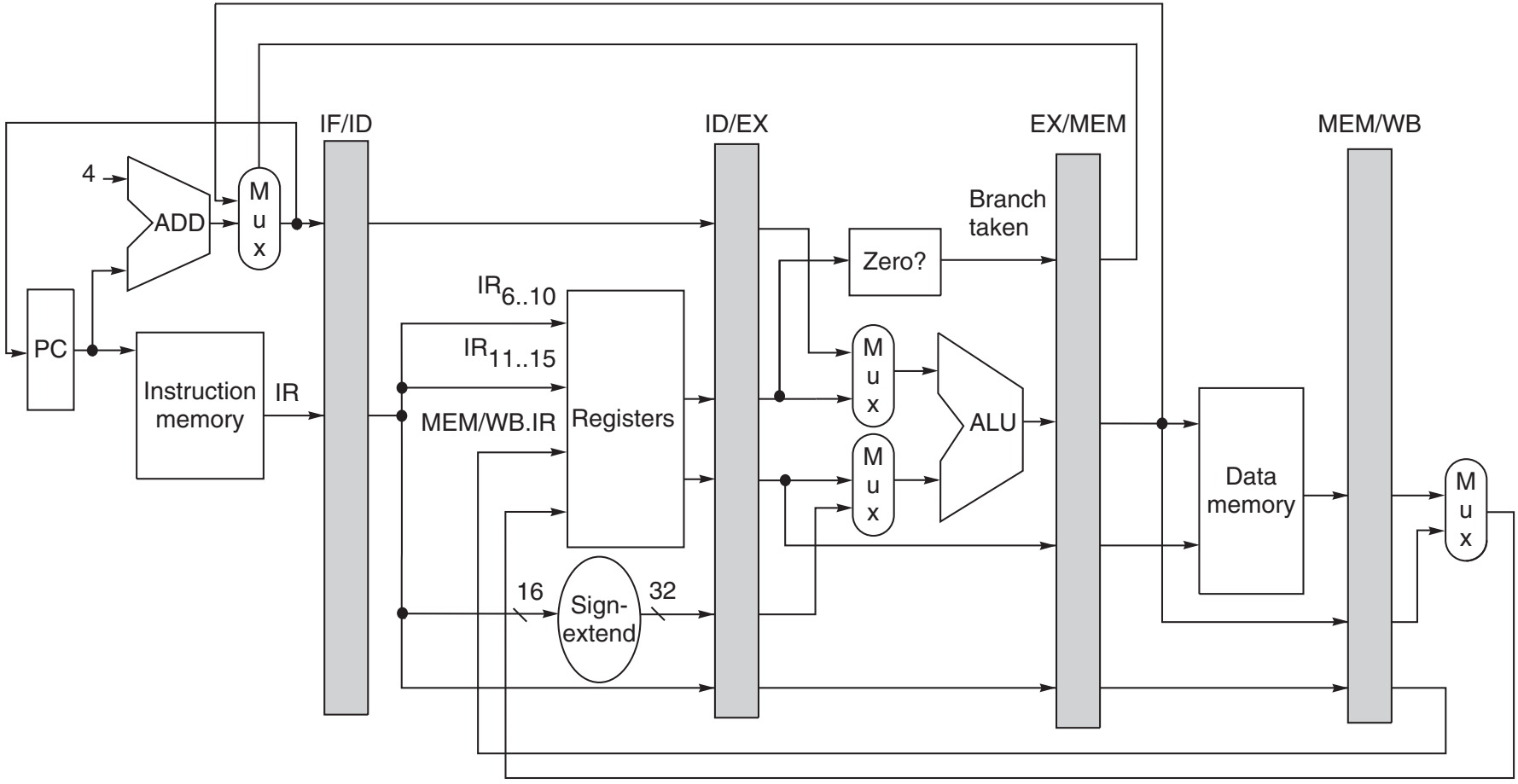
Tournament predictor selects the best of local and global predictors, and leads to better prediction.

Pipeline Implementation

Non-Pipelined Version

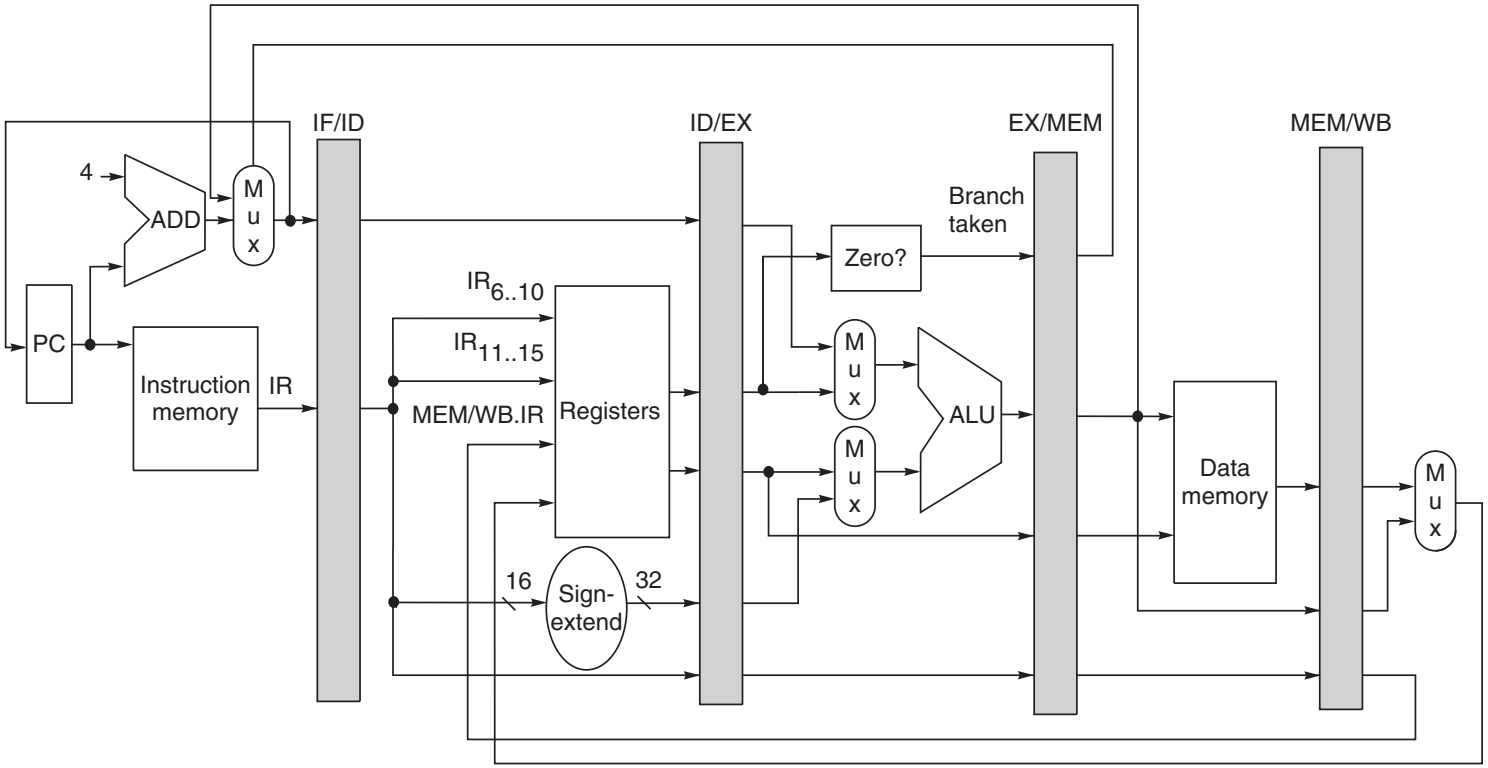


A Basic MIPS Pipeline



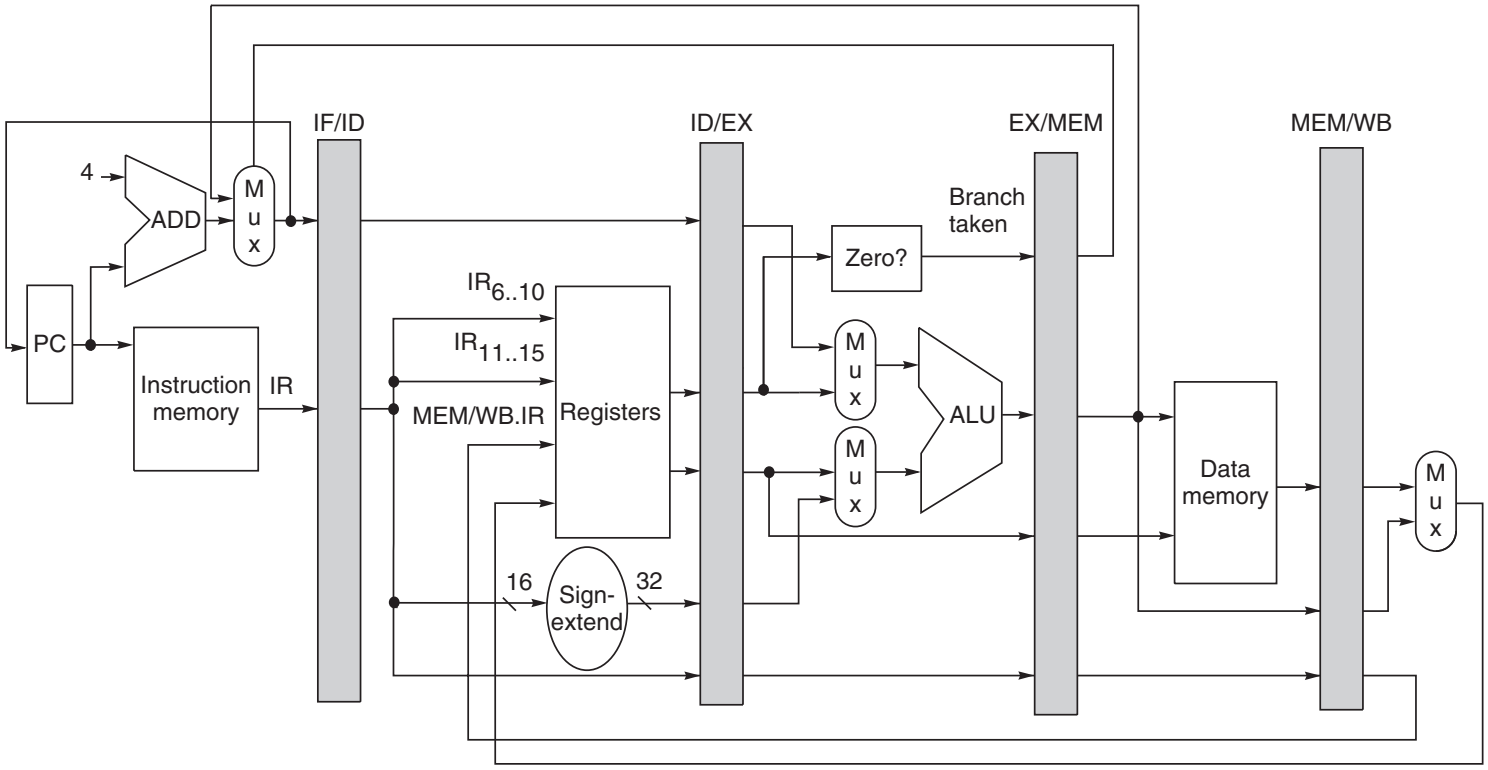
Temporary registers become part of PL registers.
PL registers carry partial results from one stage to the next.

A Basic MIPS Pipeline



Stage	Any instruction
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);

A Basic MIPS Pipeline

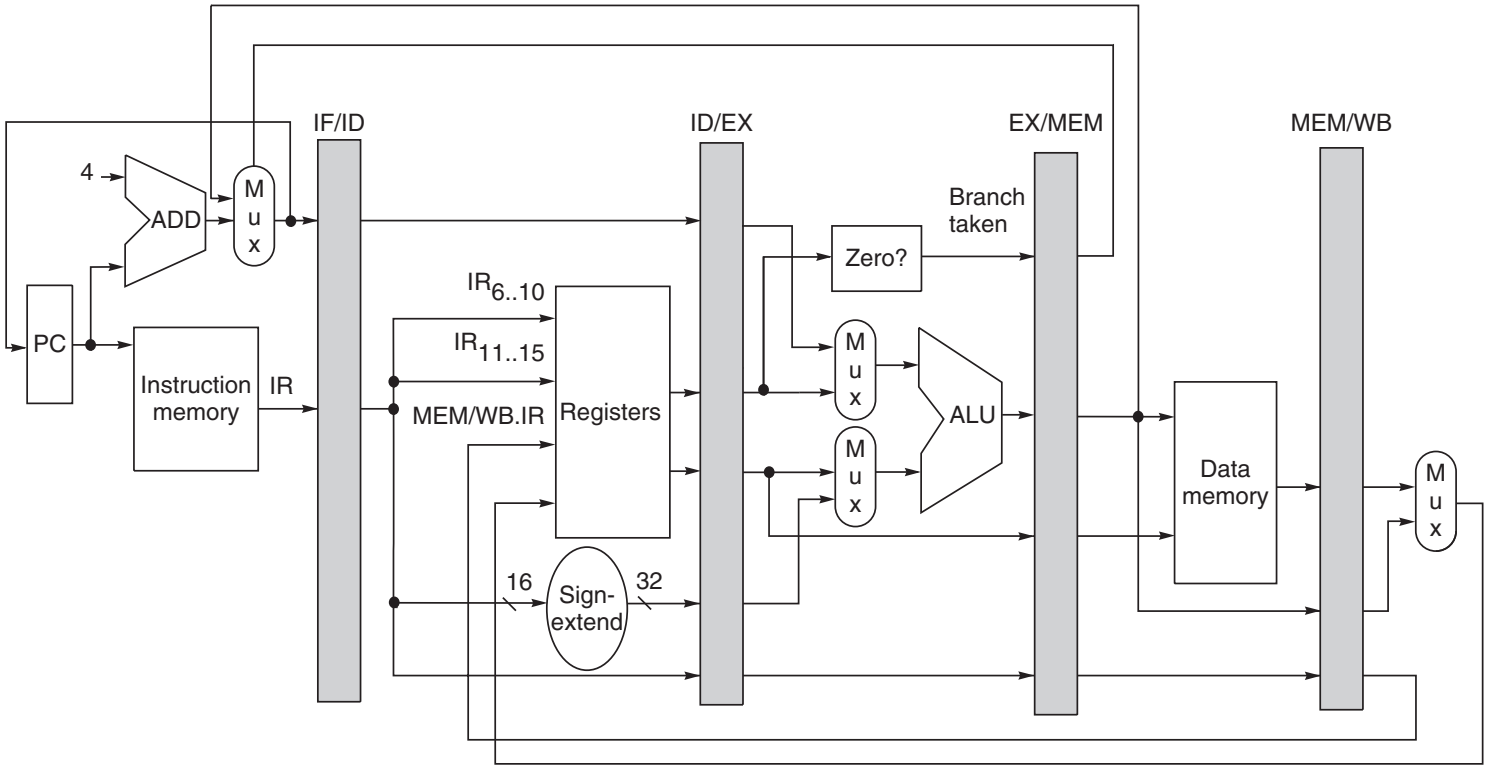


	ALU instruction	Load or store instruction	Branch instruction

EX	`EX/MEM.IR ← ID/EX.IR;`
	`EX/MEM.ALUOutput ←`
	`ID/EX.A func ID/EX.B;`
	or
	`EX/MEM.ALUOutput ←`
	`ID/EX.A` *op* `ID/EX.Imm;`
		`EX/MEM.IR to ID/EX.IR`
		`EX/MEM.ALUOutput ←`
		`ID/EX.A + ID/EX.Imm;`
			`EX/MEM.ALUOutput ←`
			`ID/EX.NPC +`
			`(ID/EX.Imm << 2);`

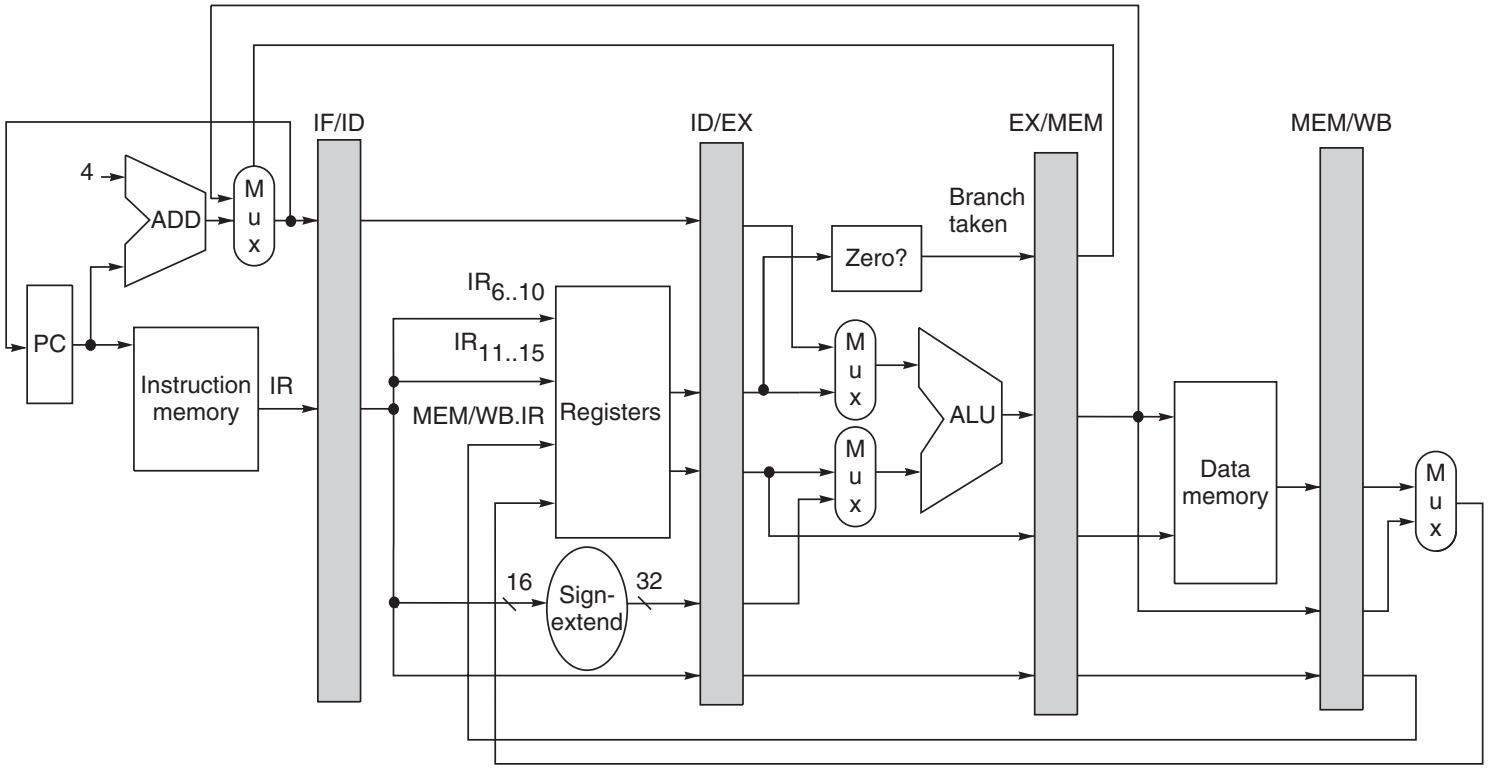
		`EX/MEM.B ← ID/EX.B;`	`EX/MEM.cond ←` `(ID/EX.A == 0);`

A Basic MIPS Pipeline



	ALU instruction	Load or store instruction	Branch instruction
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	

A Basic MIPS Pipeline



	ALU instruction	Load or store instruction	Branch instruction
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

Hazard Detection

- **Instruction issue**: move from ID to EX.
- Detect data hazards during ID
  - Stall the instruction before it is issued
  - Insert pipeline bubbles (no-ops) by changing control fields to 0s
  - (DADD R0, R0, R0)
- Early detection of interlocks (e.g., due to a load) reduces complexity
- Detect load interlock by comparing:
  - Source registers in IF/ID (consumers) with
  - Destination register in ID/EX or EX/MEM (producer)

Hazard Detection – Example

Situation	Example code sequence	Action

No dependence	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

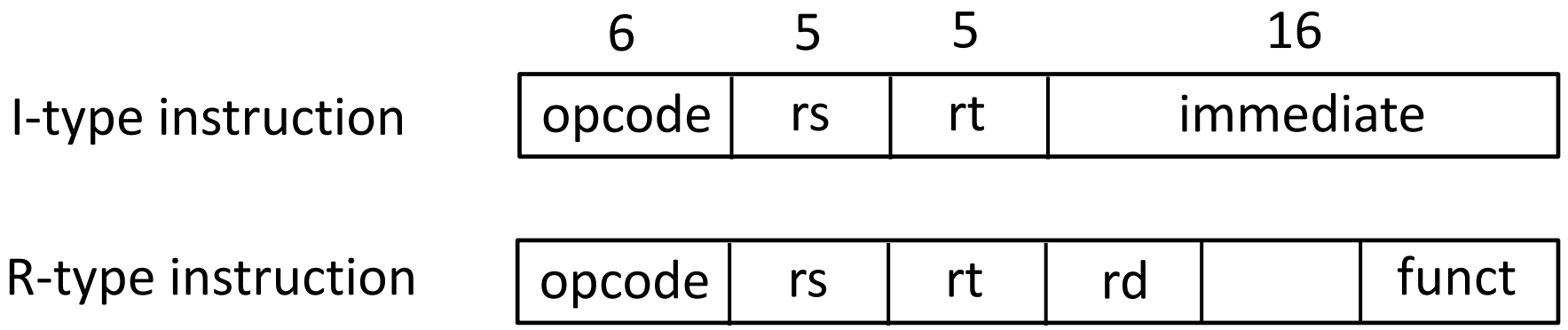
Dependence requiring stall	`LD R1,45(R2)`
	`DADD R5,R1,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.

Dependence overcome by forwarding	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R1,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.

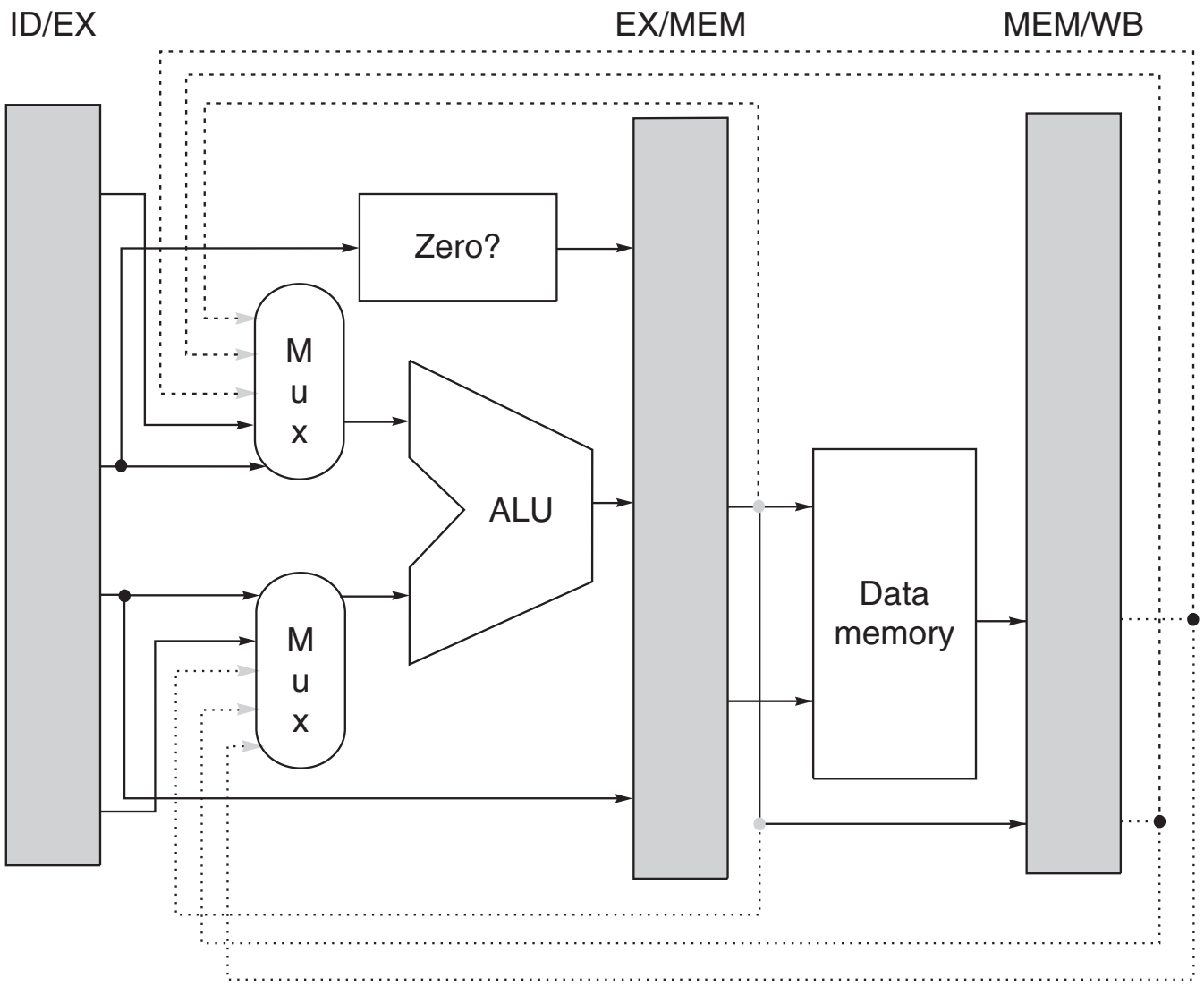
Dependence with accesses in order	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R1,R7`
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Hazard Detection – Example

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	`ID/EX.IR[rt] == IF/ID.IR[rs]`
Load	Register-register ALU	`ID/EX.IR[rt] == IF/ID.IR[rt]`
Load	Load, store, ALU immediate, or branch	`ID/EX.IR[rt] == IF/ID.IR[rs]`



Forwarding



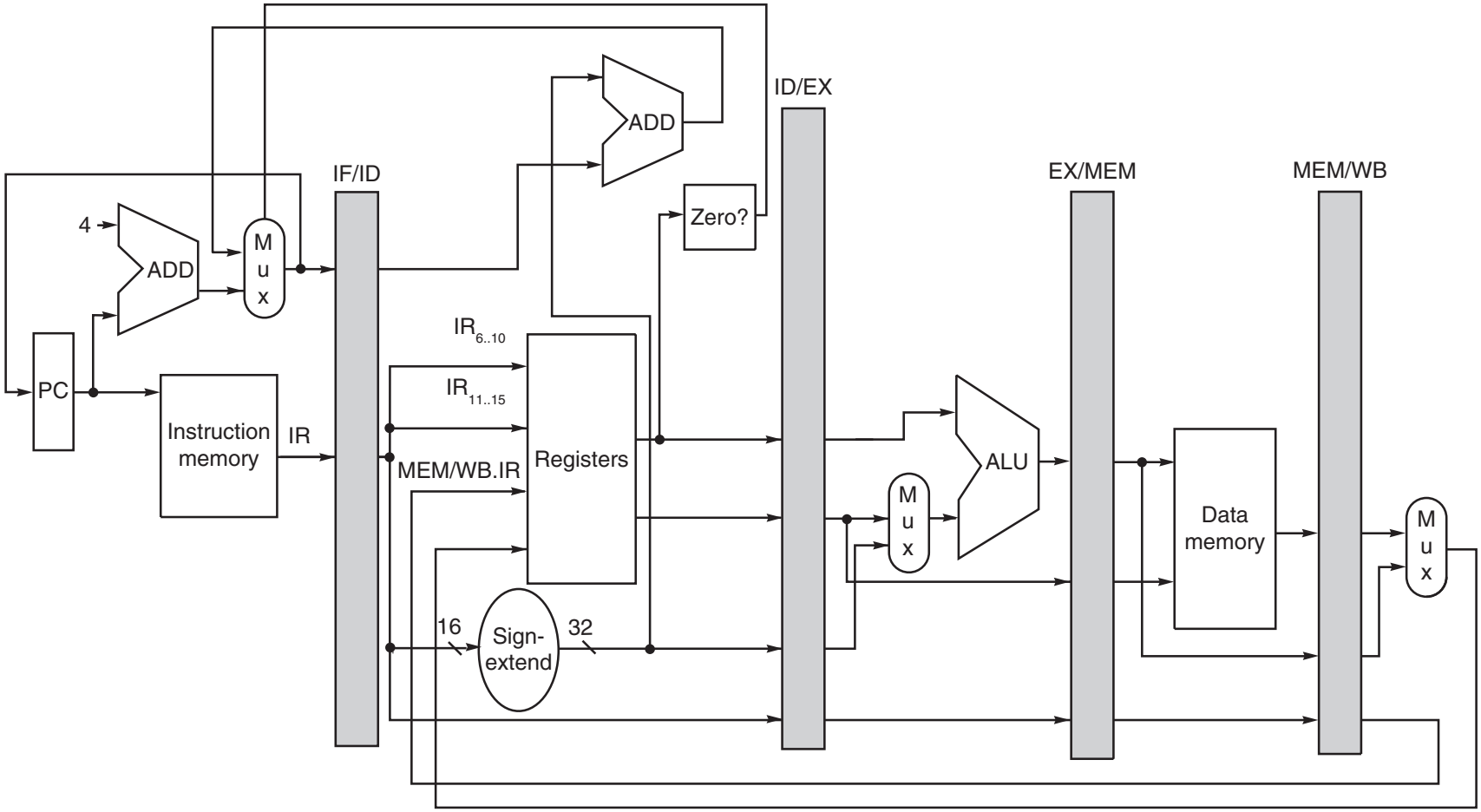
Forwarding

- Sources: outputs of
  - ALU or Memory
- Destinations: inputs of
  - ALU, Memory, Branch decision
- Forwarding done by controlling MUXs of the inputs at destinations
- Compare registers in EX/MEM or MEM/WB with registers in ID/EX and EX/MEM.
- Ex:

```
LD R1, 45(R2)
DADD ...
DSUB R8, R1, R7
```

MEM/WB.IR[rt] == ID/EX.IR[rt]
controls an input MUX to ALU.

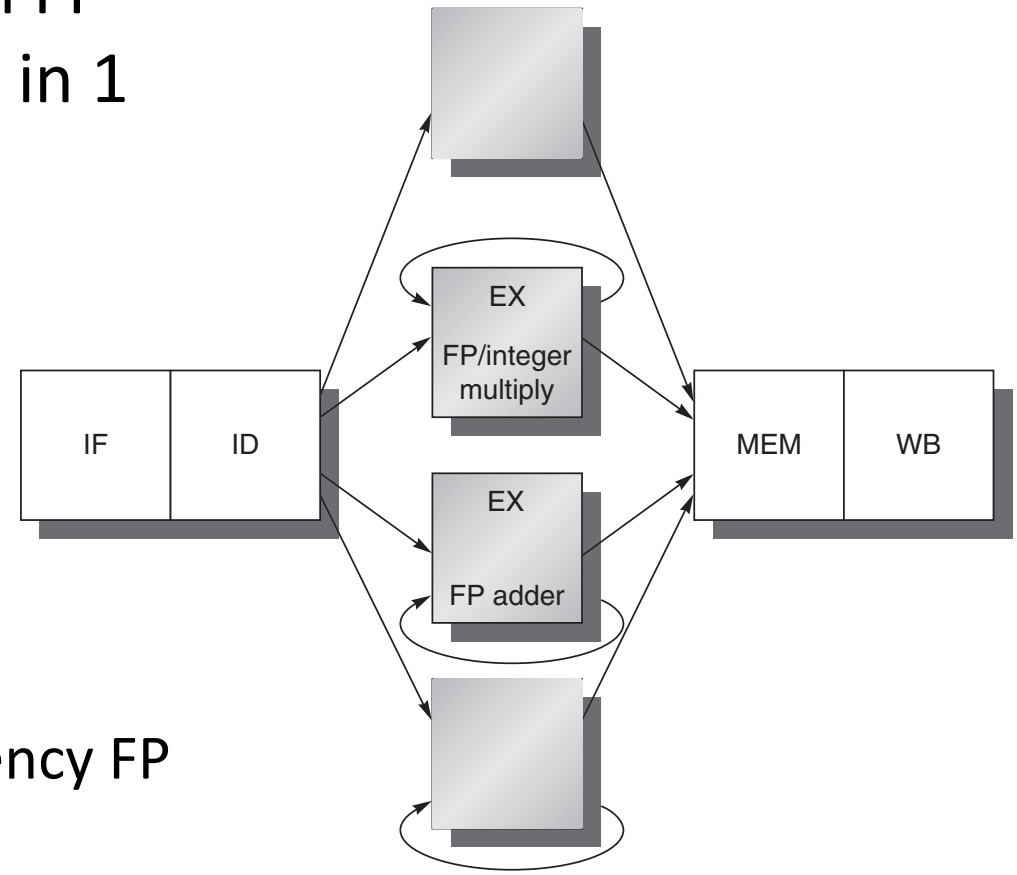
Branches in Pipeline



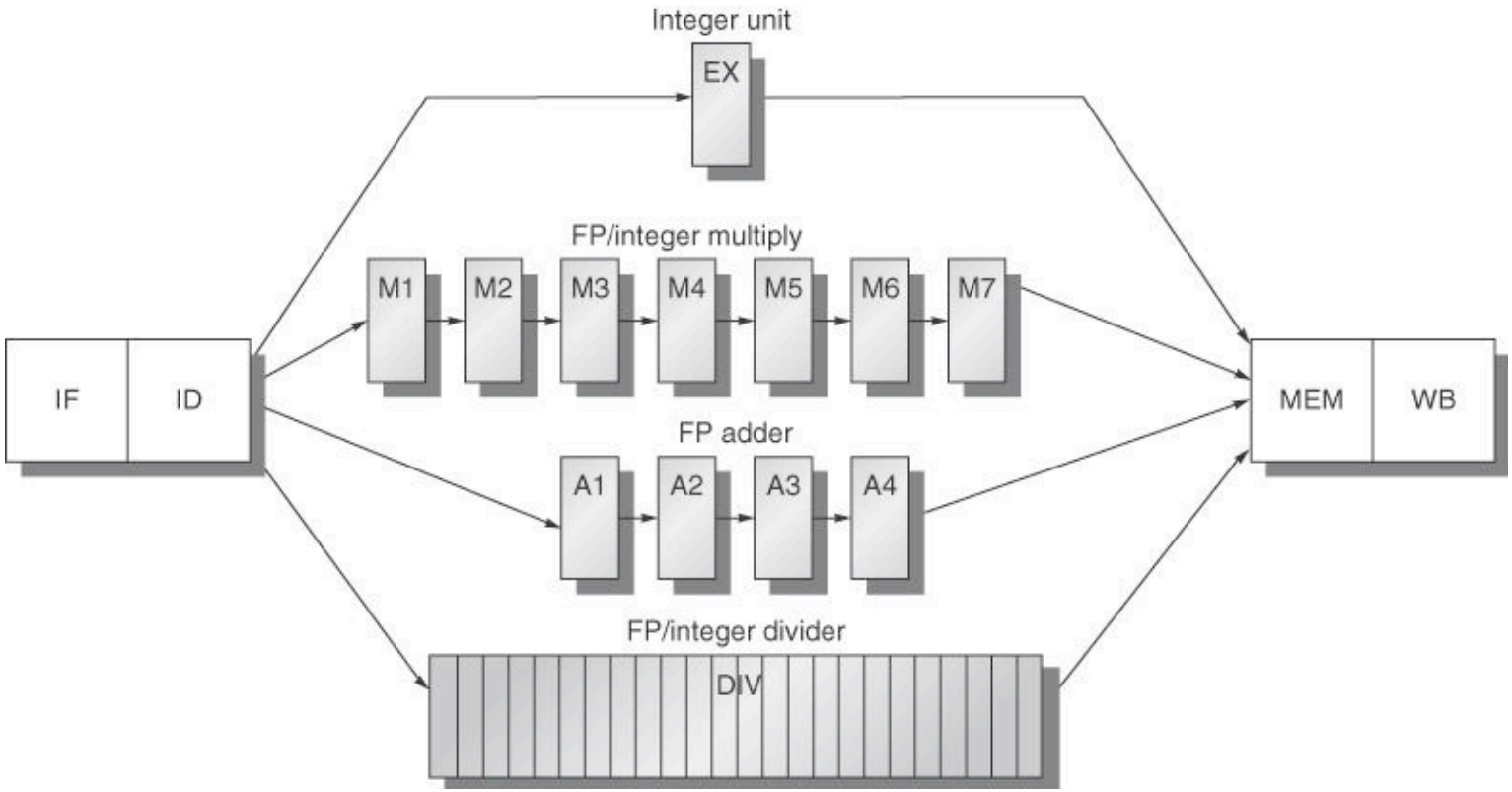
BEQZ Rx, *name* ; if (Regs[Rx]==0) PC <- *name*

Extend Pipeline to Handle Long Latency

- Impractical to require all FP operations to complete in 1 cycle
  - Longer cycle time
  - Large area overhead
- Long FP operations
  - take multiple cycles
- Non-pipelined
  - Stalls caused by long latency FP operations

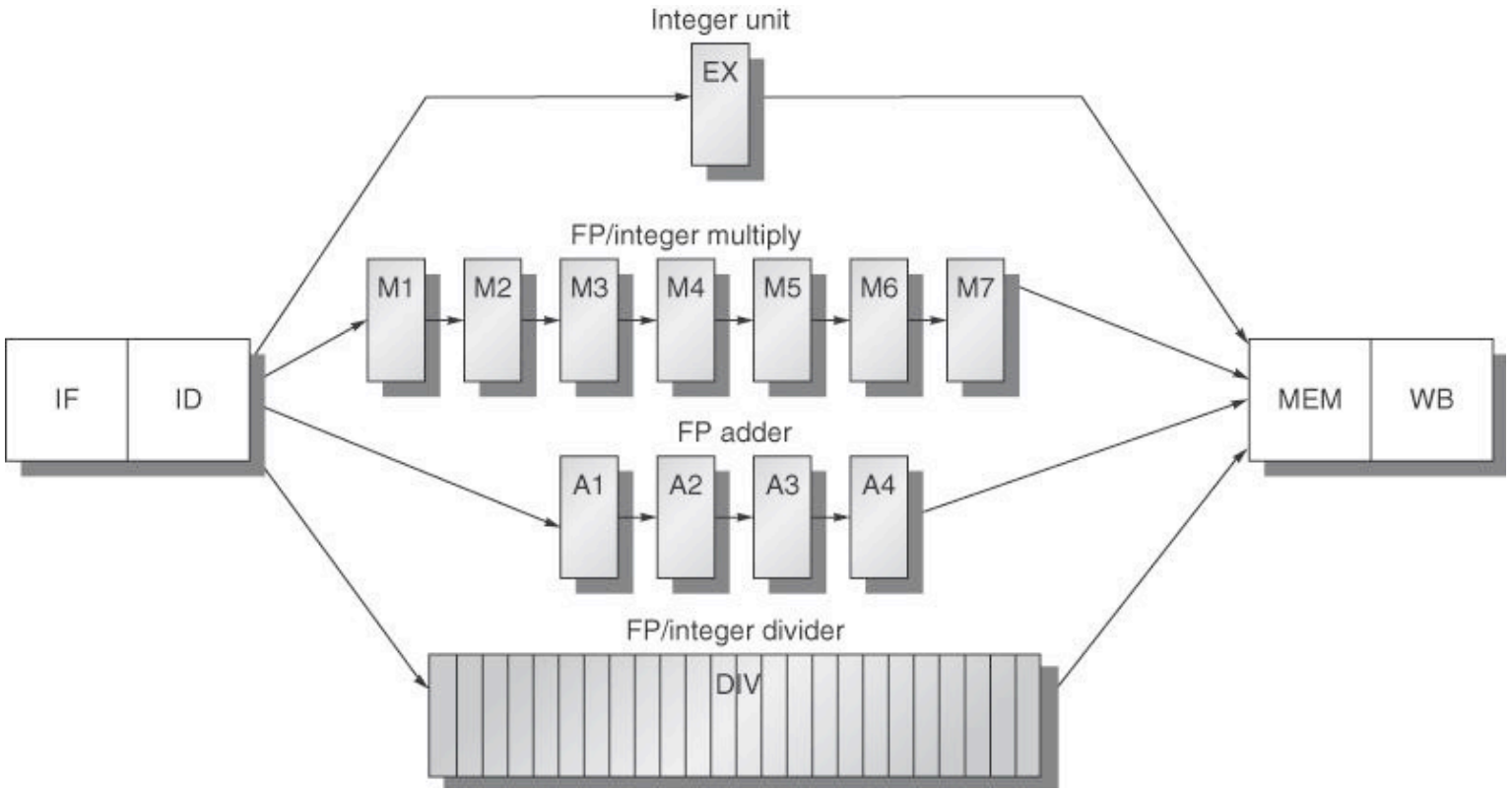


Extend Pipeline to Handle Long Latency



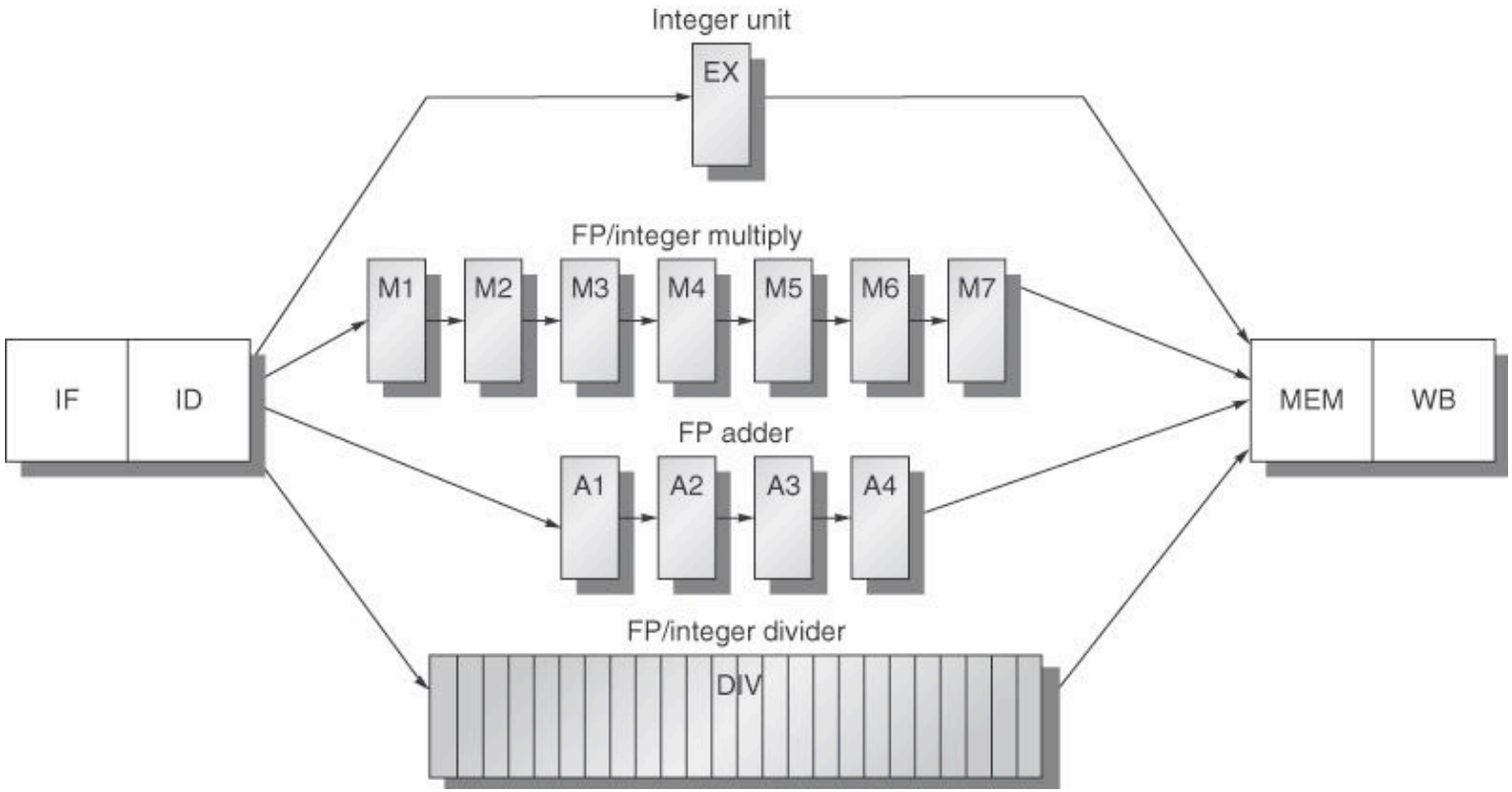
➜ Pipelining functional units allow multiple instructions to be executed.

Extend Pipeline to Handle Long Latency



Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Extend Pipeline to Handle Long Latency



Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Extend Pipeline to Handle Long Latency

- Structural hazards can occur due to long latency
- Simultaneous register writes are possible
  - due to various execution latencies
- WAW hazards are now possible
- Instructions can complete in different orders

	Clock cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
`MUL.D F0,F4,F6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`ADD.D F2,F4,F6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`L.D F2,0(R2)`							IF	ID	EX	MEM	WB

Extend Pipeline to Handle Long Latency

➜ Long latency causes more stalls for RAW hazards

		Clock cycle number																
Instruction		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
L.D	F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D	F0,F4,F6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D	F2,F0,F8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
S.D	F2,0(R2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM