

Pipelined Processor Design

M S Bhat

Dept. of E&C,

NITK Suratkal

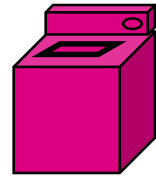
Presentation Outline

- ❖ **Pipelining versus Serial Execution**
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

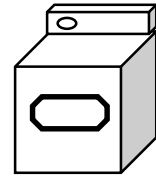
Pipelining Example

❖ Laundry Example: Three Stages

1. Wash dirty load of clothes



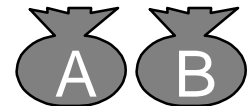
2. Dry wet clothes



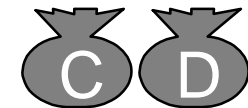
3. Fold and put clothes into drawers



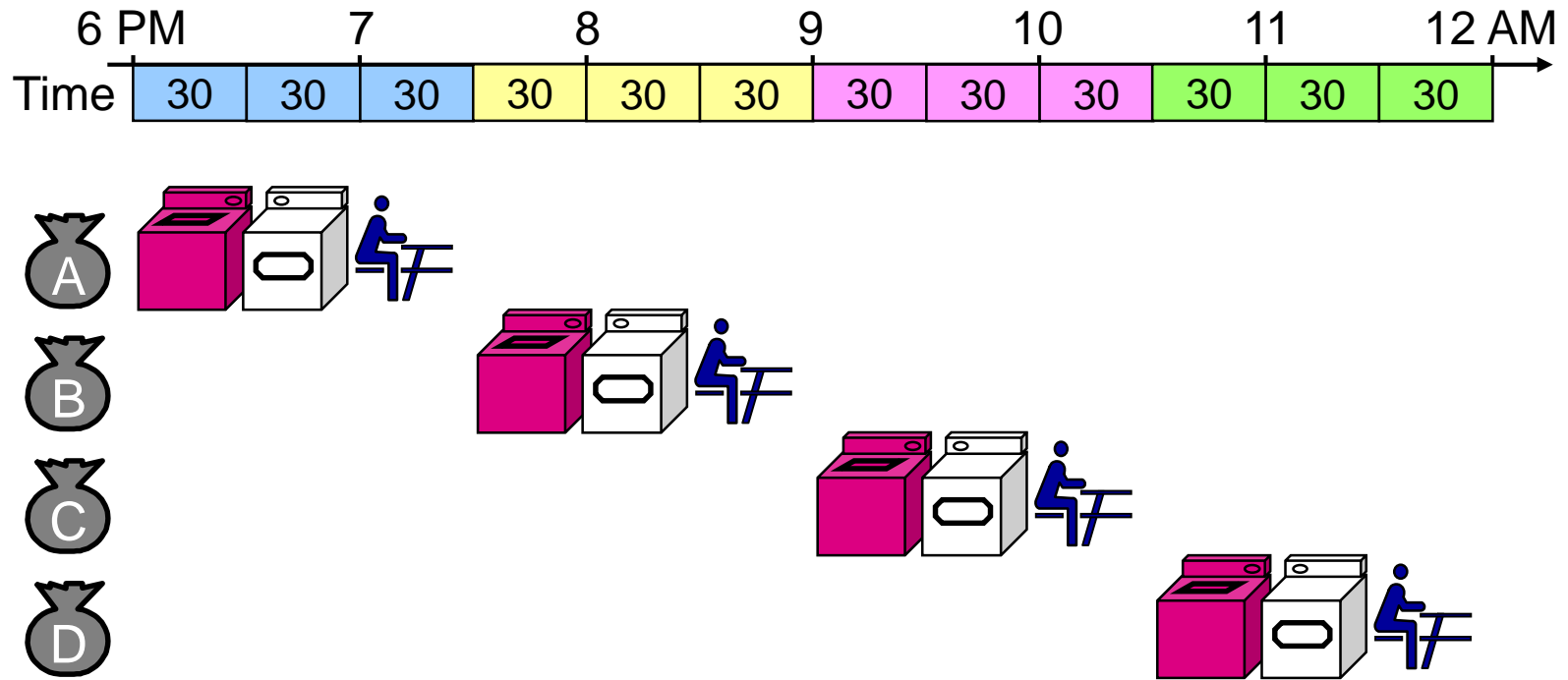
❖ Each stage takes 30 minutes to complete



❖ Four loads of clothes to wash, dry, and fold

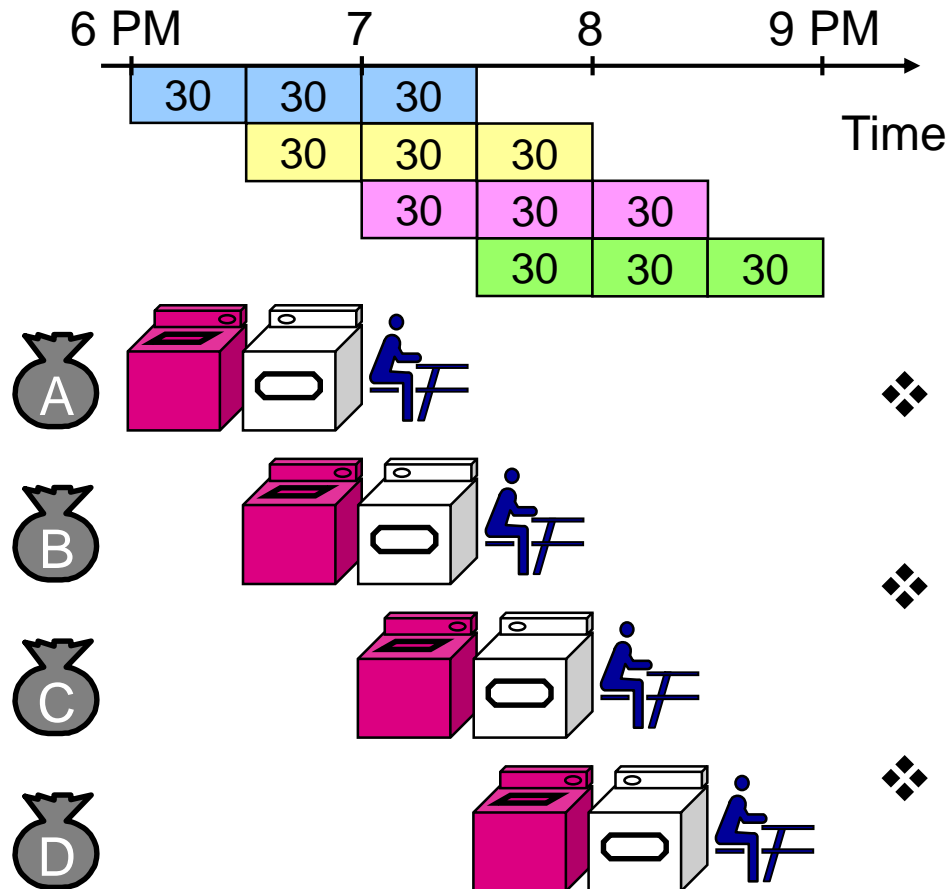


Sequential Laundry



- ❖ Sequential laundry takes **6 hours** for **4 loads**
- ❖ Intuitively, we can use **pipelining** to speed up laundry

Pipelined Laundry: Start Load ASAP



- ❖ Pipelined laundry takes **3 hours** for **4 loads**
- ❖ Speedup factor is **2** for **4 loads**
- ❖ Time to wash, dry, and fold one load is still the same (90 minutes)

Serial Execution versus Pipelining

❖ Consider a task that can be divided into k subtasks

✧ The k subtasks are executed on k different stages

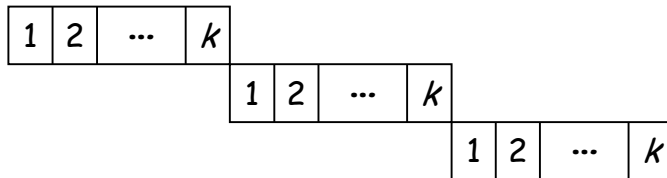
✧ Each subtask requires one time unit

✧ The total execution time of the task is k time units

❖ Pipelining is to overlap the execution

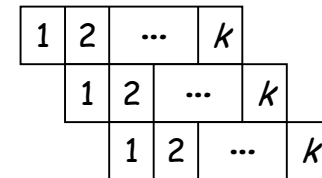
✧ The k stages work in parallel on k different tasks

✧ Tasks enter/leave pipeline at the rate of one task per time unit



Without Pipelining

One completion every k time units

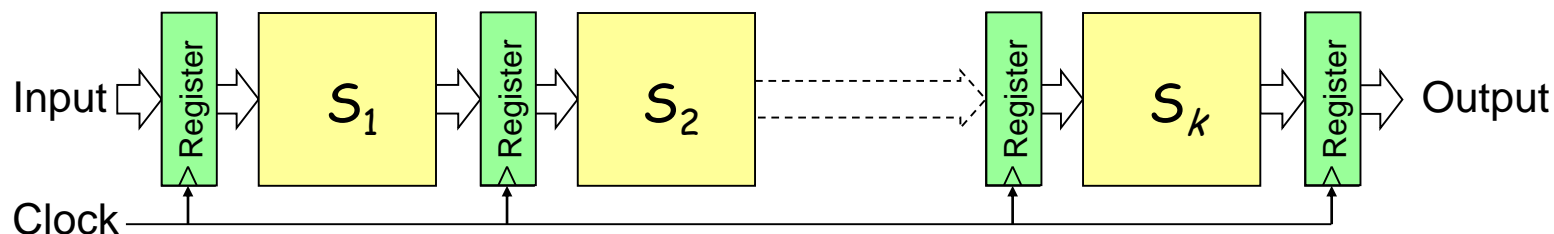


With Pipelining

One completion every 1 time unit

Synchronous Pipeline

- ❖ Uses **clocked registers** between stages
- ❖ Upon arrival of a clock edge ...
 - ✧ All registers hold the results of previous stages simultaneously
- ❖ The pipeline stages are **combinational logic** circuits
- ❖ It is desirable to have **balanced** stages
 - ✧ Approximately equal delay in all stages
- ❖ Clock period is determined by the **maximum stage delay**



Pipeline Performance

- ❖ Let τ_i = time delay in stage S_i
- ❖ Clock cycle $\tau = \max(\tau_i)$ is the **maximum stage delay**
- ❖ Clock frequency $f = 1/\tau = 1/\max(\tau_i)$
- ❖ A k -stage pipeline can process n tasks in $k + n - 1$ cycles
 - ✧ k cycles are needed to complete the first task
 - ✧ $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks
- ❖ Ideal **speedup** of a k -stage pipeline over serial execution

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k + n - 1} \quad S_k \rightarrow k \text{ for large } n$$

MIPS Processor Pipeline

❖ Five stages, one cycle per stage

1. IF: **Instruction Fetch** from instruction memory
2. ID: **Instruction Decode**, register read, and J/Br address
3. EX: **Execute** operation or calculate load/store address
4. MEM: **Memory access** for load and store
5. WB: **Write Back** result to register

Performance Example

- ❖ Assume the following operation times for components:
 - ✧ Instruction and data memories: 200 ps
 - ✧ ALU and adders: 180 ps
 - ✧ Decode and Register file access (read or write): 150 ps
 - ✧ Ignore the delays in PC, mux, extender, and wires
- ❖ Which of the following would be faster and by how much?
 - ✧ Single-cycle implementation for all instructions
 - ✧ Multicycle implementation optimized for every class of instructions
- ❖ Assume the following instruction mix:
 - ✧ 40% ALU, 20% Loads, 10% stores, 20% branches, & 10% jumps

Single-Cycle vs Multicycle Implementation

❖ Break instruction execution into **five steps**

- ✧ Instruction fetch
- ✧ Instruction decode and register read
- ✧ Execution, memory address calculation, or branch completion
- ✧ Memory access or ALU instruction completion
- ✧ Load instruction completion

❖ **One step = One clock cycle** (clock cycle is reduced)

- ✧ First 2 steps are the same for all instructions

Instruction	# cycles	Instruction	# cycles
ALU & Store	4	Branch	3
Load	5	Jump	2

Solution

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	200	150	180		150	680 ps
Load	200	150	180	200	150	880 ps
Store	200	150	180	200		730 ps
Branch	200	150	180			530 ps
Jump	200	150 ←	decode and update PC			350 ps

❖ For fixed single-cycle implementation:

✧ Clock cycle = 880 ps determined by longest delay (load instruction)

❖ For multi-cycle implementation:

✧ Clock cycle = $\max(200, 150, 180) = 200$ ps (maximum delay at any step)

✧ Average CPI = $0.4 \times 4 + 0.2 \times 5 + 0.1 \times 4 + 0.2 \times 3 + 0.1 \times 2 = 3.8$

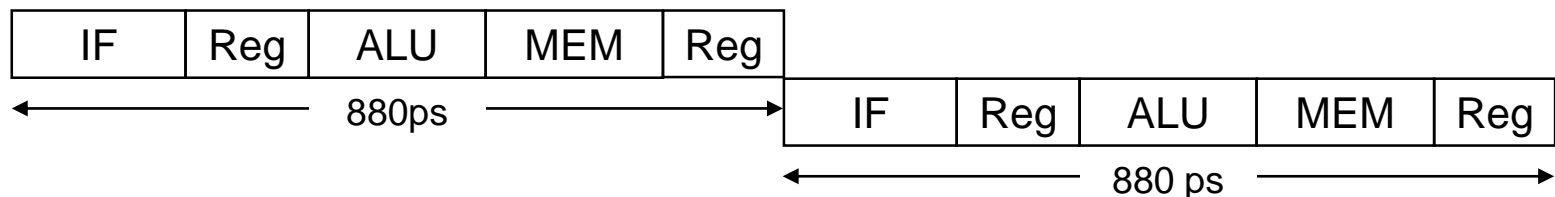
❖ Speedup = $880 \text{ ps} / (3.8 \times 200 \text{ ps}) = 880 / 760 = 1.16$

Single-Cycle vs Pipelined Performance

- ❖ Consider a 5-stage instruction execution in which ...
 - ✧ Instruction fetch = Data memory access = 200 ps
 - ✧ ALU operation = 180 ps
 - ✧ Register read = register write = 150 ps
- ❖ What is the clock cycle of the single-cycle processor?
- ❖ What is the clock cycle of the pipelined processor?
- ❖ What is the speedup factor of pipelined execution?

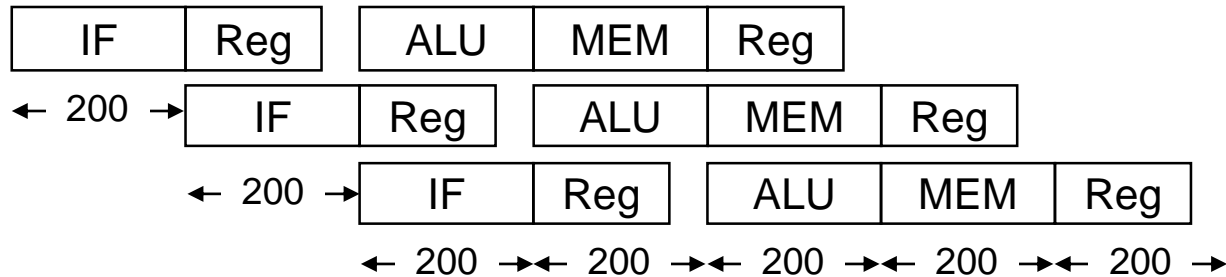
❖ Solution

Single-Cycle Clock = $200+150+180+200+150 = 880 \text{ ps}$



Single-Cycle versus Pipelined - cont'd

❖ Pipelined clock cycle = $\max(200, 180, 150) = 200 \text{ ps}$



❖ CPI for pipelined execution = 1

✧ One instruction is completed in each cycle (ignoring pipeline fill)

❖ Speedup of pipelined execution = $880 \text{ ps} / 200 \text{ ps} = 4.4$

✧ Instruction count and CPI are equal in both cases

❖ Speedup factor is less than 5 (number of pipeline stage)

✧ Because the pipeline stages are not balanced

❖ Throughput = $1/\text{Max}(\text{delay}) = 1/200 \text{ ps} = 5 \times 10^9 \text{ instructions/sec}$

Pipeline Performance Summary

- ❖ Pipelining doesn't improve **latency** of a single instruction
- ❖ However, it improves **throughput** of entire workload
 - ✧ Instructions are initiated and completed at a higher rate
- ❖ In a **k-stage** pipeline, **k** instructions operate **in parallel**
 - ✧ Overlapped execution using multiple hardware resources
 - ✧ Potential speedup = **number of pipeline stages k**
 - ✧ Unbalanced lengths of pipeline stages reduces speedup
- ❖ Pipeline rate is limited by **slowest** pipeline stage
- ❖ Unbalanced lengths of pipeline stages reduces speedup
- ❖ Also, time to **fill** and **drain** pipeline reduces speedup

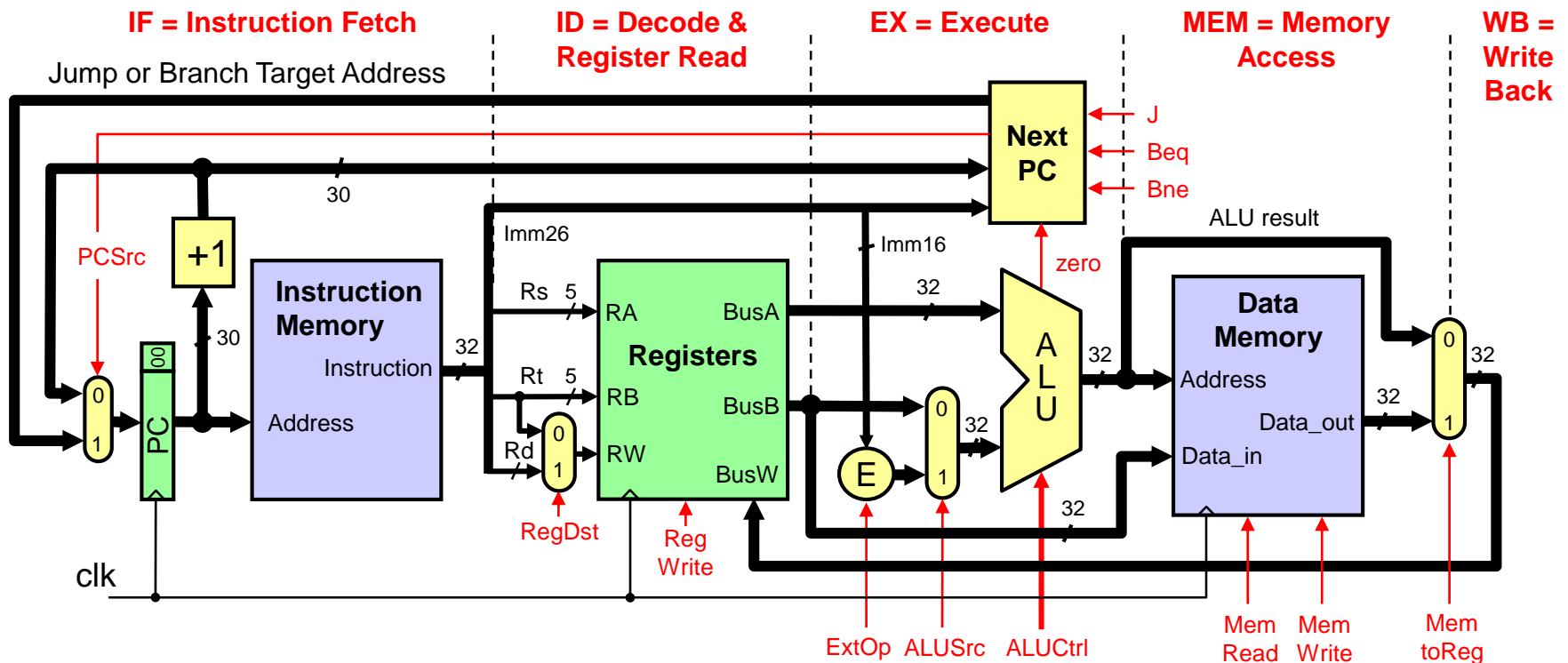
Next . . .

- ❖ Pipelining versus Serial Execution
- ❖ **Pipelined Datapath and Control**
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Single-Cycle Datapath

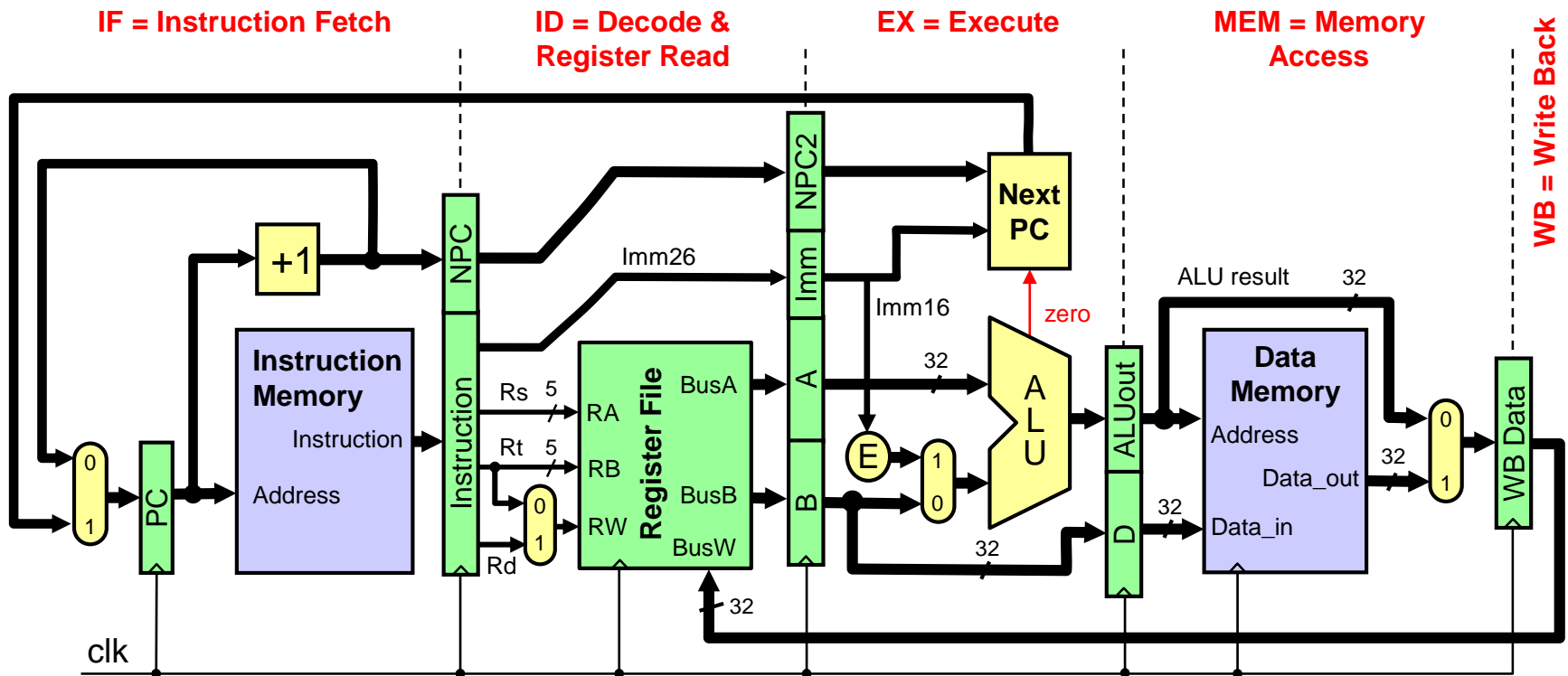
- ❖ Shown below is the single-cycle datapath
- ❖ How to pipeline this single-cycle datapath?

Answer: Introduce pipeline register at end of each stage



Pipelined Datapath

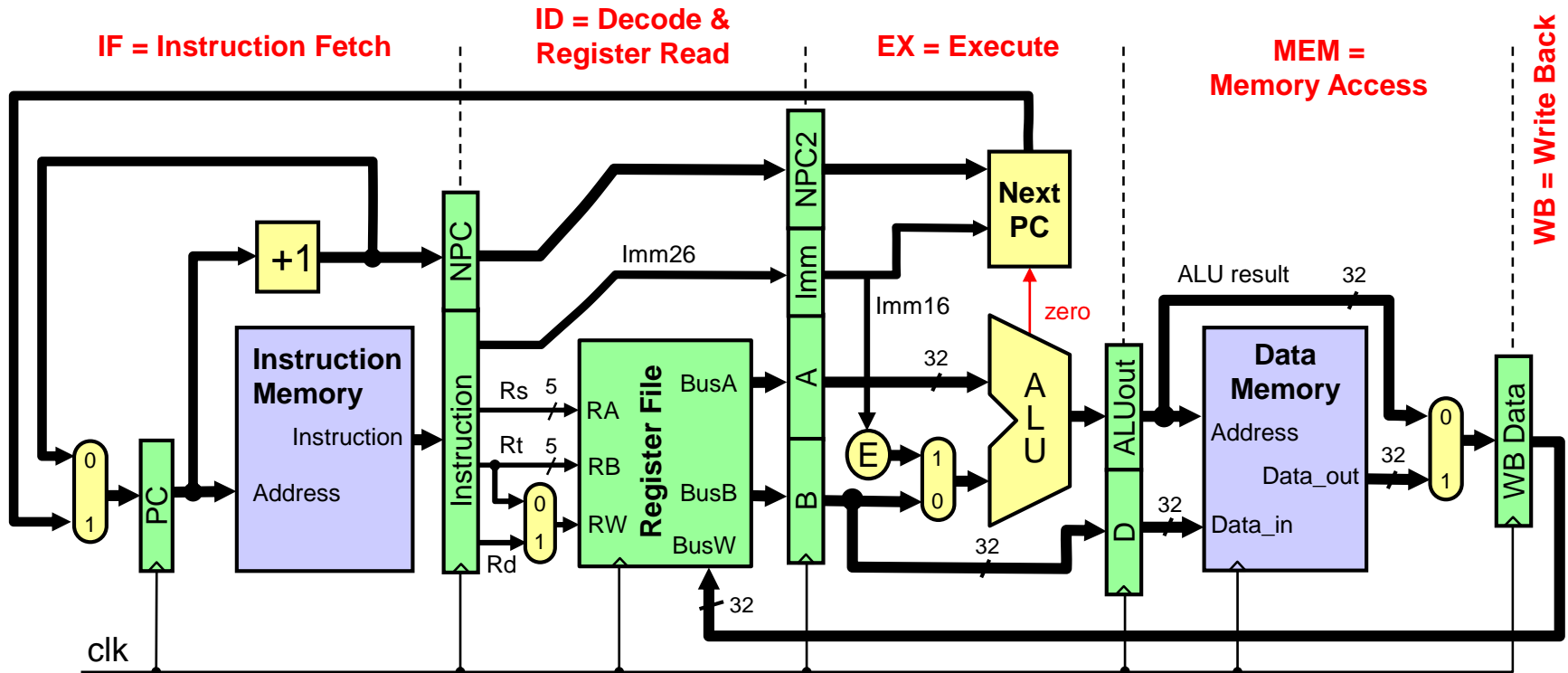
- ❖ Pipeline registers are shown in green, including the PC
- ❖ Same clock edge updates all pipeline registers, register file, and data memory (for store instruction)



Problem with Register Destination

❖ Is there a problem with the register destination address?

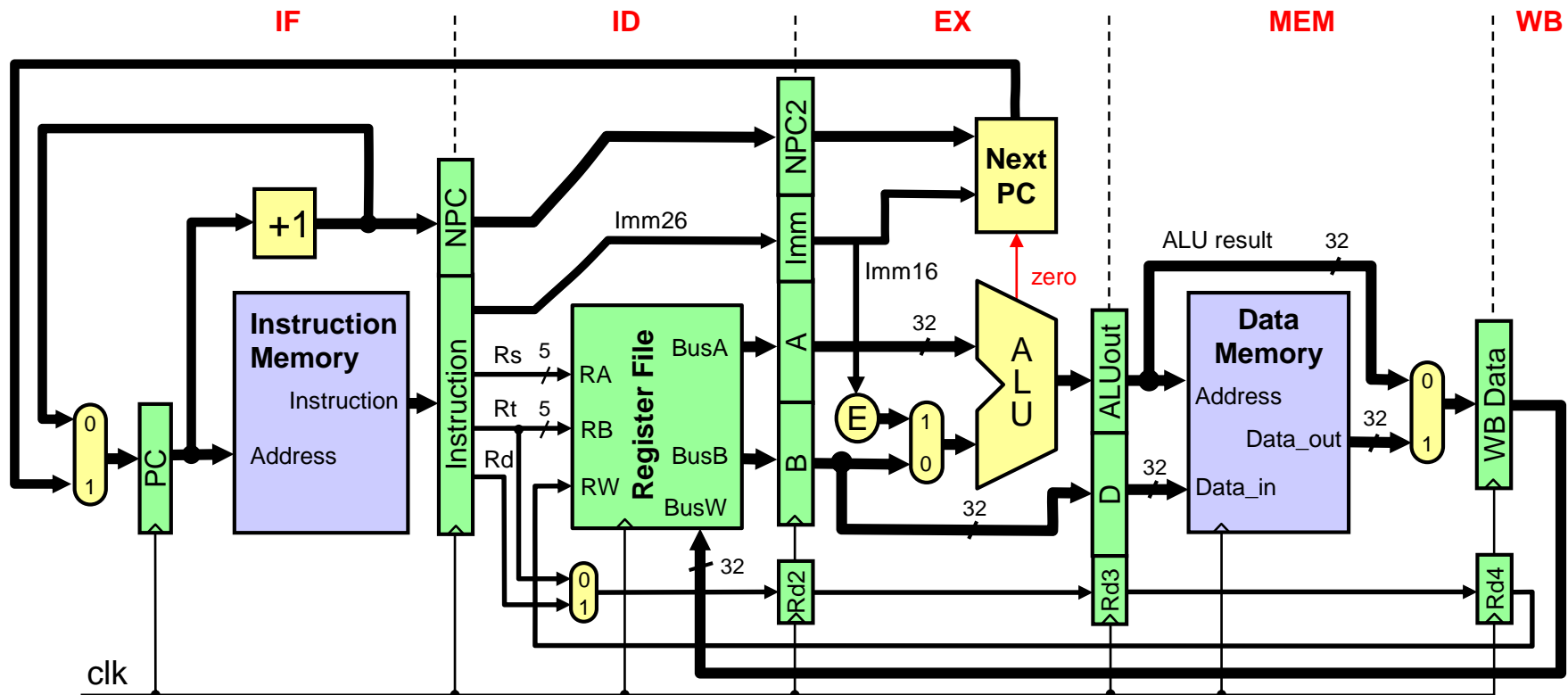
- ✧ Instruction in the ID stage different from the one in the WB stage
- ✧ Instruction in the WB stage is not writing to its destination register but to the destination of a different instruction in the ID stage



Pipelining the Destination Register

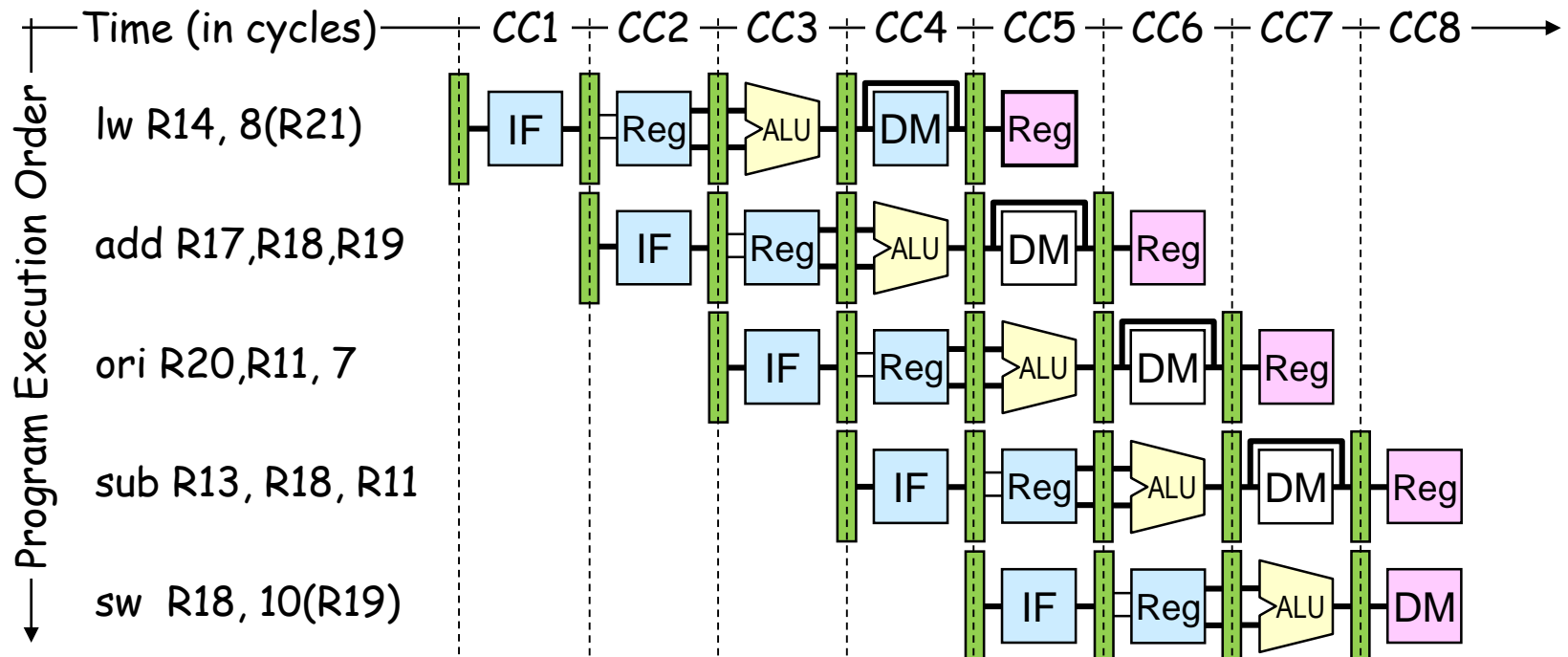
❖ Destination Register number should be pipelined

- ✧ Destination register number is passed from ID to WB stage
- ✧ The WB stage writes back data knowing the destination register



Graphically Representing Pipelines

- ❖ Multiple instruction execution over multiple clock cycles
 - ✧ Instructions are listed in execution order from top to bottom
 - ✧ Clock cycles move from left to right
 - ✧ Figure shows the use of resources at each stage and each cycle



Instruction-Time Diagram

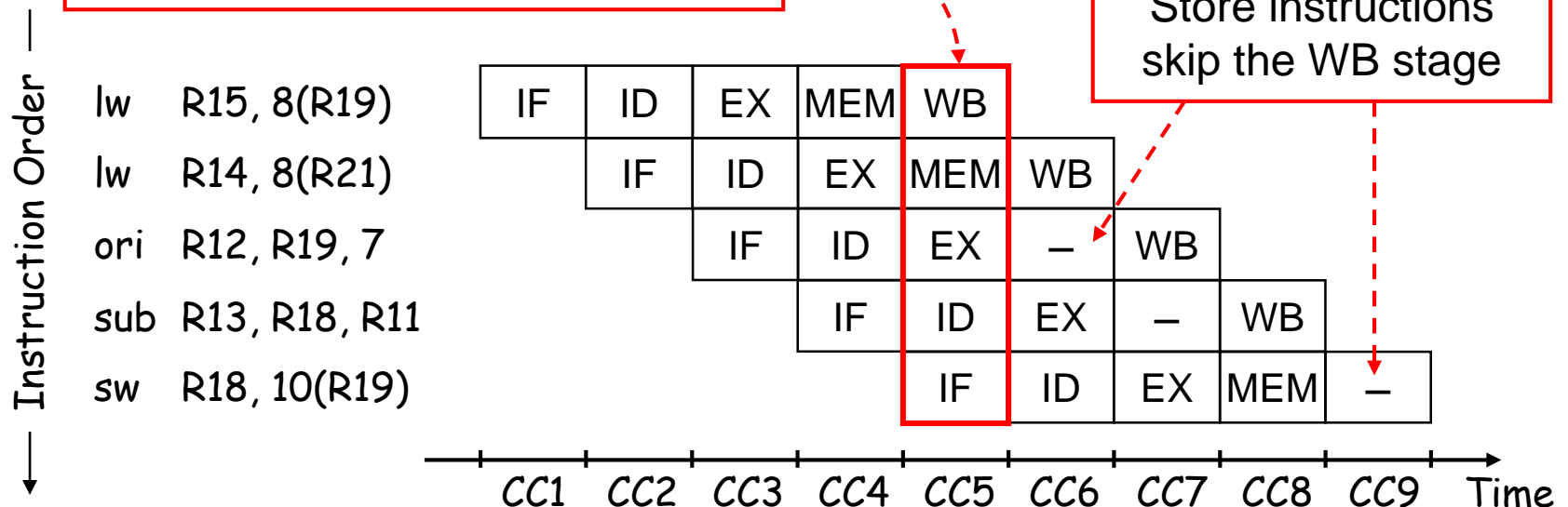
❖ Instruction-Time Diagram shows:

✧ Which instruction occupying what stage at each clock cycle

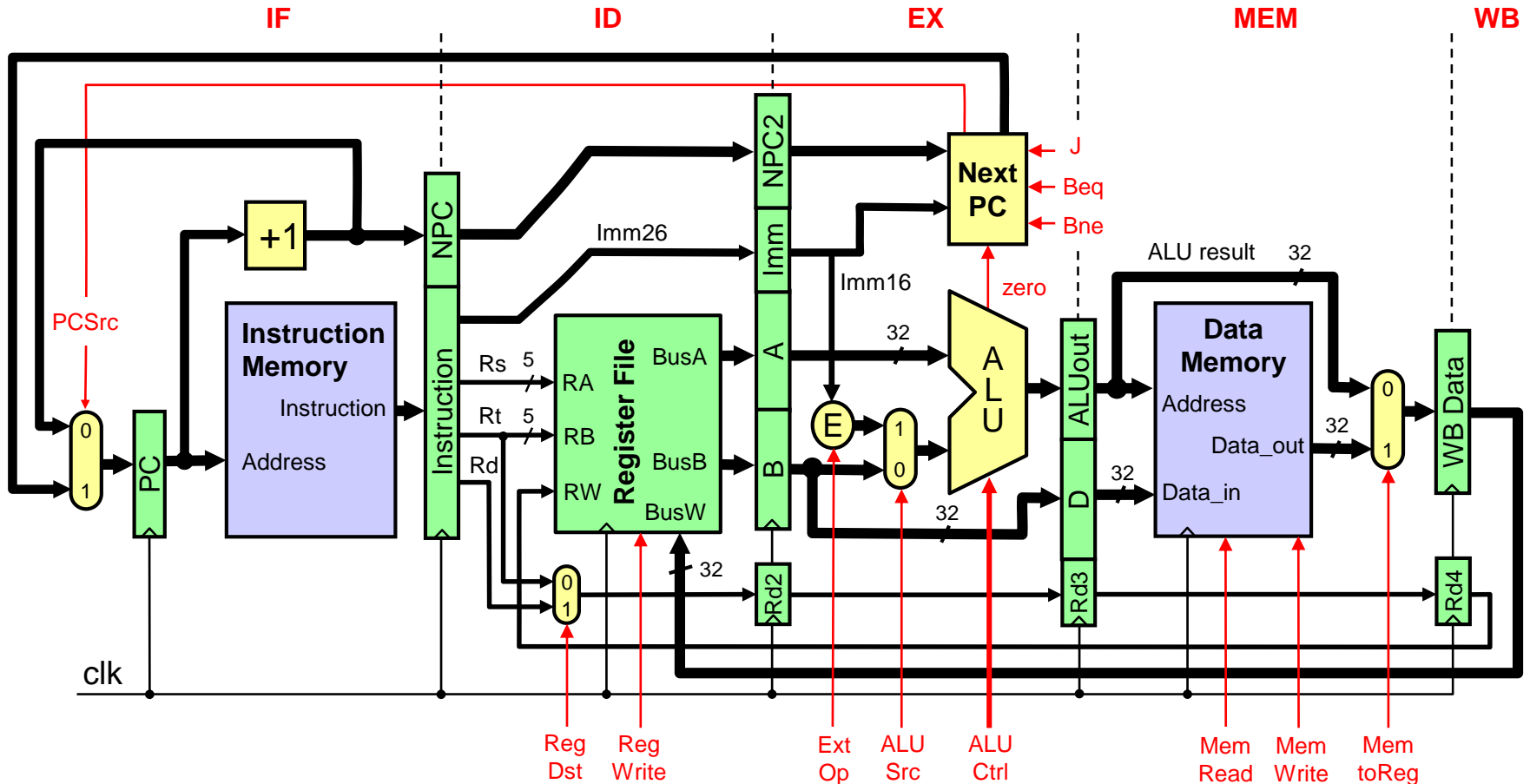
❖ Instruction flow is pipelined over the 5 stages

Up to five instructions can be in the pipeline during the same cycle. Instruction Level Parallelism (ILP)

ALU instructions skip the MEM stage. Store instructions skip the WB stage

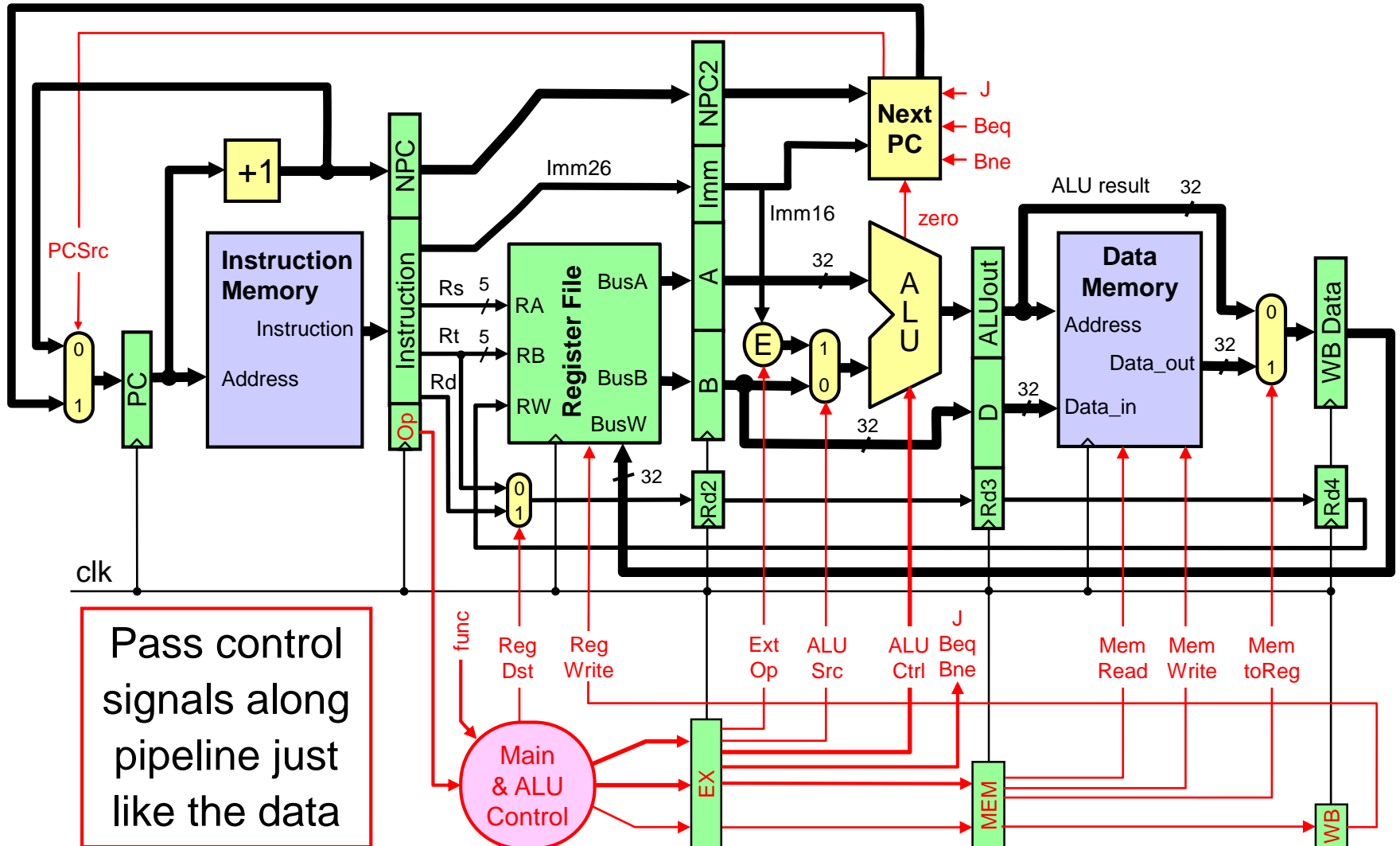


Control Signals



Same control signals used in the single-cycle datapath

Pipelined Control



Pipelined Control - Cont'd

- ❖ ID stage generates all the control signals
- ❖ Pipeline the control signals as the instruction moves
 - ✧ Extend the pipeline registers to include the control signals
- ❖ Each stage uses some of the control signals
 - ✧ Instruction Decode and Register Read
 - Control signals are generated
 - **RegDst** is used in this stage
 - ✧ Execution Stage => **ExtOp**, **ALUSrc**, and **ALUCtrl**
 - Next PC uses **J**, **Beq**, **Bne**, and **zero** signals for branch control
 - ✧ Memory Stage => **MemRead**, **MemWrite**, and **MemtoReg**
 - ✧ Write Back Stage => **RegWrite** is used in this stage

Control Signals Summary

Op	Decode Stage	Execute Stage Control Signals						Memory Stage Control Signals			Write Back
	RegDst	ALUSrc	ExtOp	J	Beq	Bne	ALUCtrl	MemRd	MemWr	MemReg	RegWrite
R-Type	1=Rd	0=Reg	x	0	0	0	func	0	0	0	1
addi	0=Rt	1=Imm	1=sign	0	0	0	ADD	0	0	0	1
slti	0=Rt	1=Imm	1=sign	0	0	0	SLT	0	0	0	1
andi	0=Rt	1=Imm	0=zero	0	0	0	AND	0	0	0	1
ori	0=Rt	1=Imm	0=zero	0	0	0	OR	0	0	0	1
lw	0=Rt	1=Imm	1=sign	0	0	0	ADD	1	0	1	1
sw	x	1=Imm	1=sign	0	0	0	ADD	0	1	x	0
beq	x	0=Reg	x	0	1	0	SUB	0	0	x	0
bne	x	0=Reg	x	0	0	1	SUB	0	0	x	0
j	x	x	x	1	0	0	x	0	0	x	0

Next . . .

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Pipeline Hazards

- ❖ **Hazards:** situations that would cause incorrect execution
 - ✧ If next instruction were launched during its designated clock cycle

1. Structural hazards

- ✧ Caused by resource contention
- ✧ Using same resource by two instructions during the same cycle

2. Data hazards

- ✧ An instruction may compute a result needed by next instruction
- ✧ Hardware can detect dependencies between instructions

3. Control hazards

- ✧ Caused by instructions that change control flow (branches/jumps)
- ✧ Delays in changing the flow of control

- ❖ Hazards complicate pipeline control and limit performance

Structural Hazards

❖ Problem

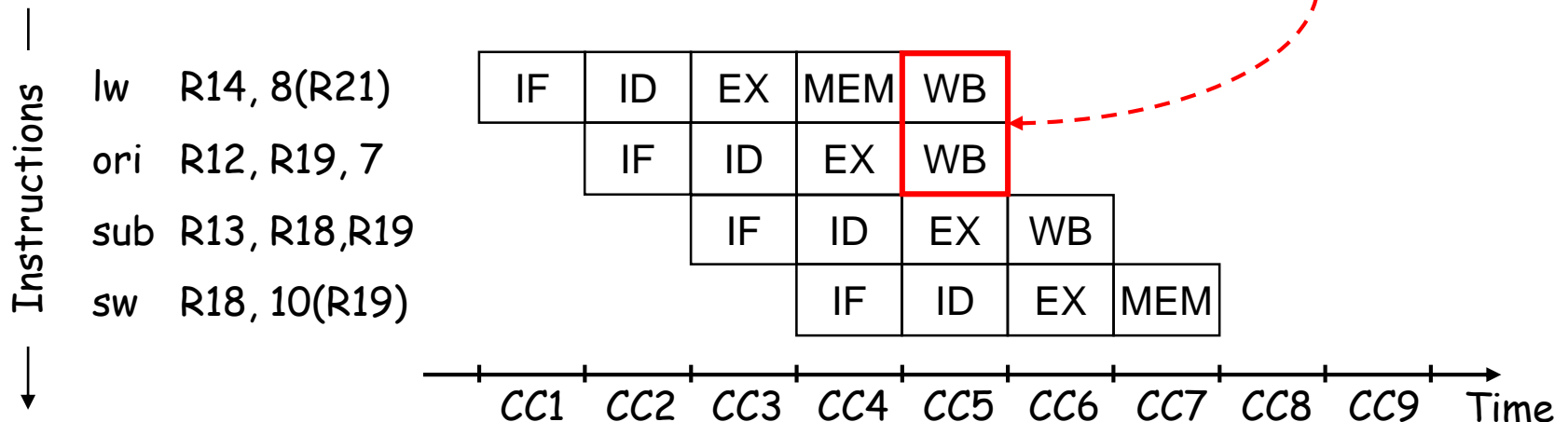
- ✧ Attempt to use the same hardware resource by two different instructions during the same cycle

❖ Example

- ✧ Writing back ALU result in stage 4
- ✧ Conflict with writing load data in stage 5

Structural Hazard

Two instructions are attempting to write the register file during same cycle



Resolving Structural Hazards

❖ Serious Hazard:

- ✧ Hazard cannot be ignored

❖ Solution 1: Delay Access to Resource

- ✧ Must have mechanism to delay instruction access to resource
- ✧ Delay all write backs to the register file to stage 5
 - ALU instructions bypass stage 4 (memory) without doing anything

❖ Solution 2: Add more hardware resources (more costly)

- ✧ Add more hardware to eliminate the structural hazard
- ✧ Redesign the register file to have two write ports
 - First write port can be used to write back ALU results in stage 4
 - Second write port can be used to write back load data in stage 5

Next . . .

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Data Hazards

- ❖ Dependency between instructions causes a data hazard
- ❖ The dependent instructions are close to each other
 - ✧ Pipelined execution might change the order of operand access

❖ Read After Write – RAW Hazard

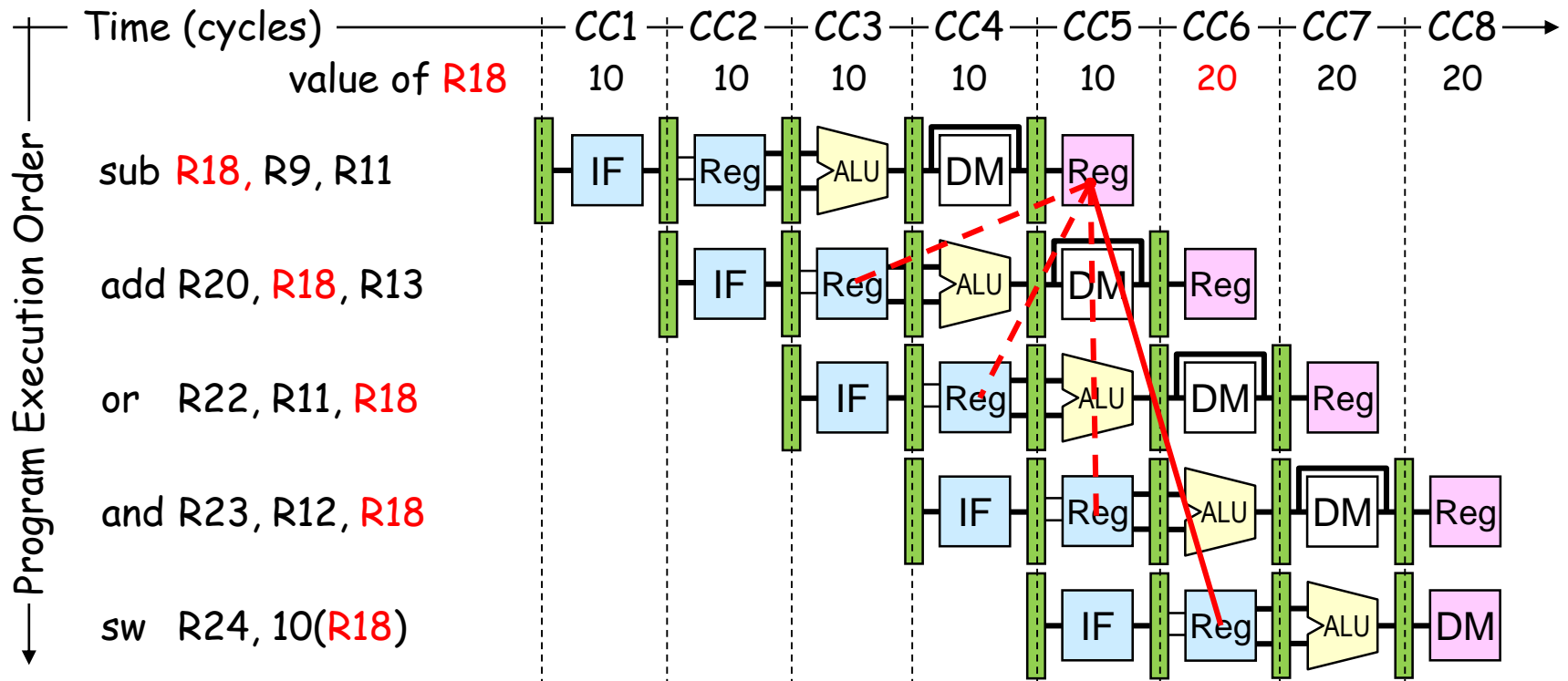
- ✧ Given two instructions *I* and *J*, where *I* comes before *J*
- ✧ Instruction *J* should read an operand after it is written by *I*
- ✧ Called a **data dependence** in compiler terminology

I: `add R17, R18, R19` **# R17 is written**

J: `sub R20, R17, R19` **# R17 is read**

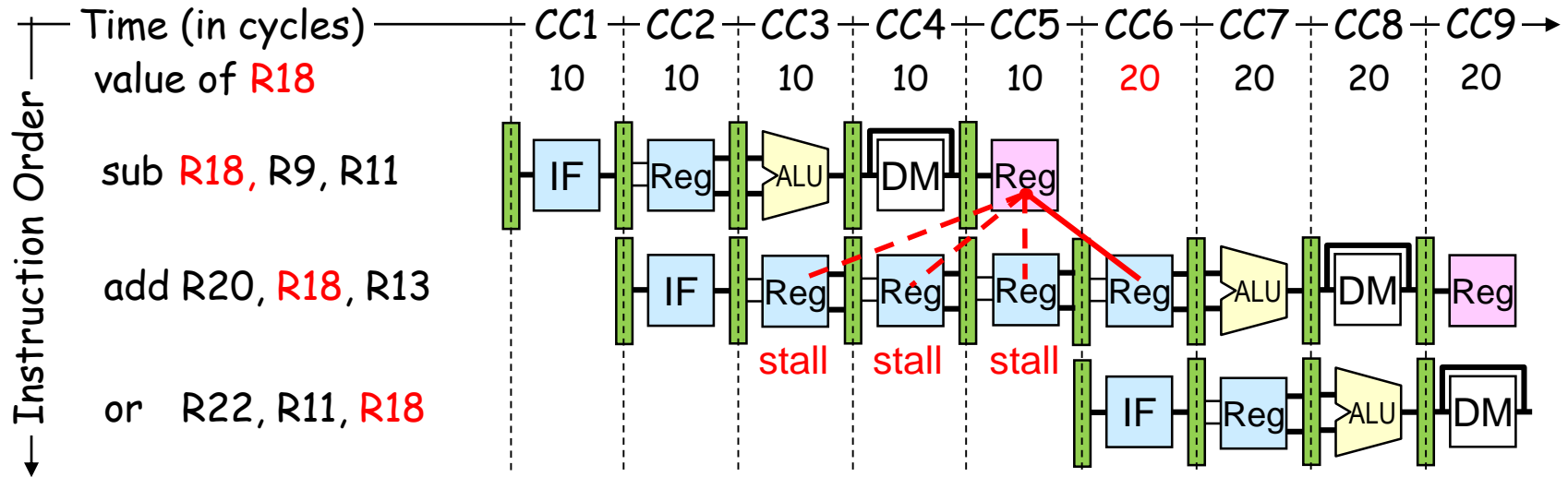
- ✧ Hazard occurs when *J* reads the operand before *I* writes it

Example of a RAW Data Hazard



- ❖ Result of **sub** is needed by **add**, **or**, **and**, & **sw** instructions
- ❖ Instructions **add** & **or** will read **old value** of **R18** from reg file
- ❖ During CC5, **R18** is written at end of cycle

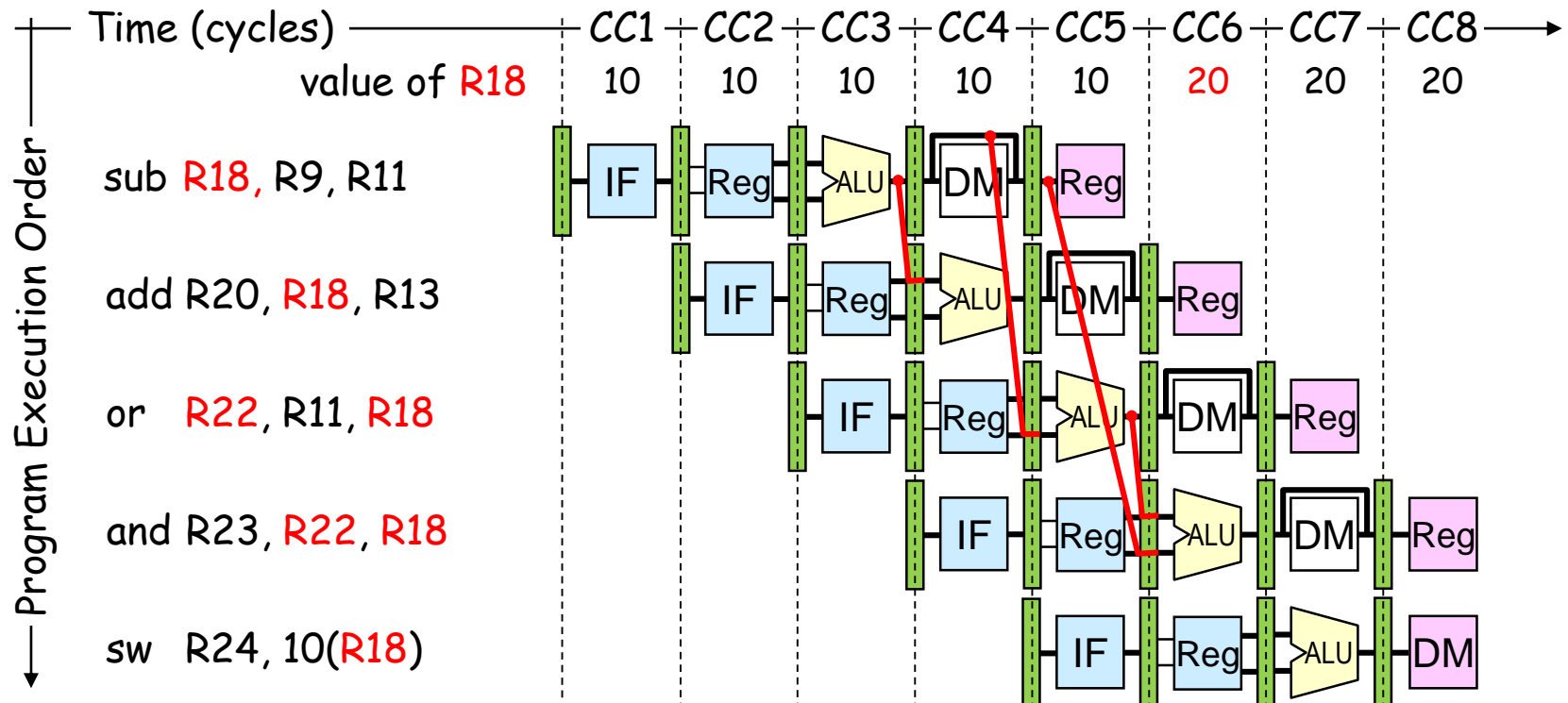
Solution 1: Stalling the Pipeline



- ❖ Three stall cycles during **CC3** thru **CC5** (wasting 3 cycles)
 - ✧ Stall cycles delay execution of **add** & fetching of **or** instruction
- ❖ The **add** instruction cannot read **R18** until beginning of **CC6**
 - ✧ The **add** instruction remains in the **Instruction register** until **CC6**
 - ✧ The **PC register** is not modified **until beginning of CC6**

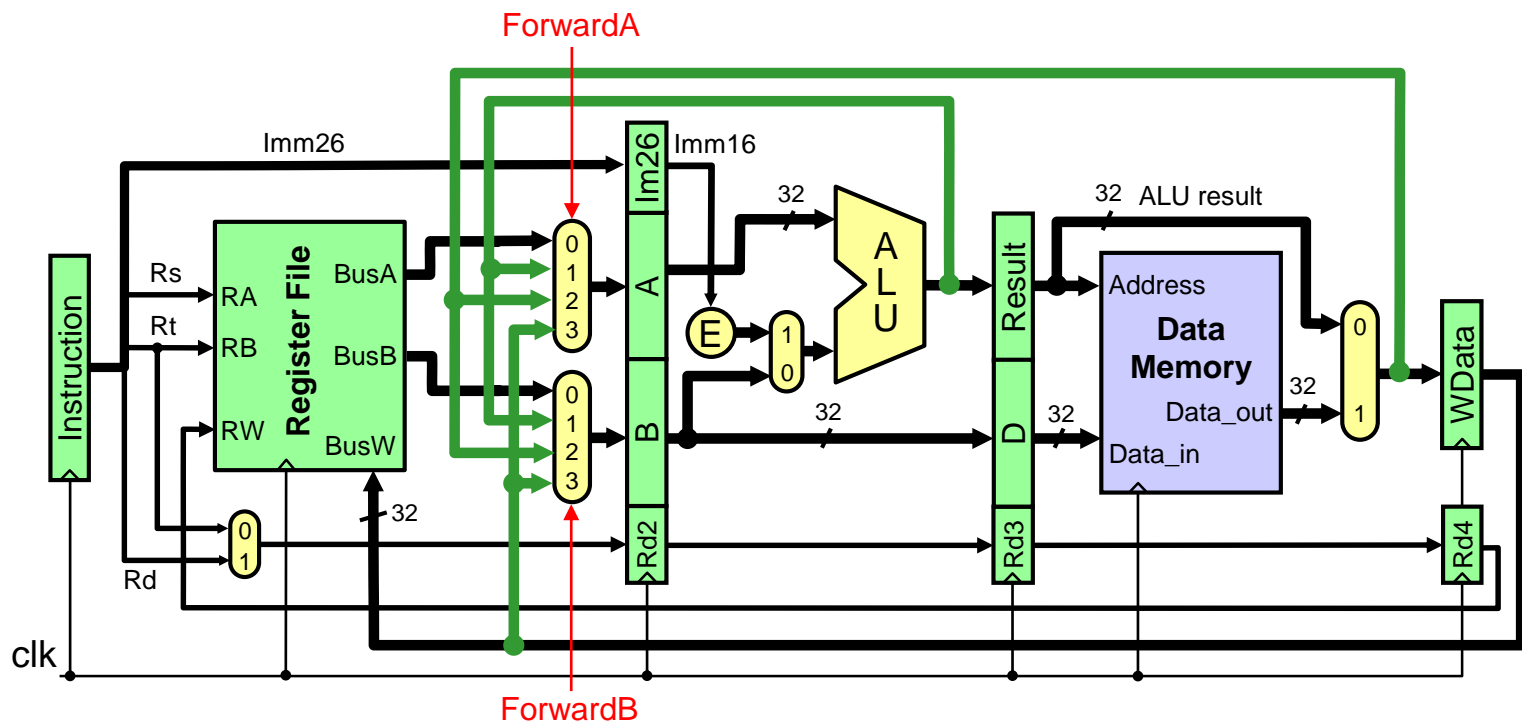
Solution 2: Forwarding ALU Result

- ❖ The **ALU result** is **forwarded** (fed back) to the **ALU input**
 - ✧ No bubbles are inserted into the pipeline and **no cycles are wasted**
- ❖ ALU result is forwarded from **ALU**, **MEM**, and **WB** stages



Implementing Forwarding

- ❖ Two multiplexers added at the inputs of A & B registers
 - ✧ Data from **ALU stage**, **MEM stage**, and **WB stage** is fed back
- ❖ Two signals: **ForwardA** and **ForwardB** control forwarding



Forwarding Control Signals

Signal	Explanation
ForwardA = 0	First ALU operand comes from register file = Value of (Rs)
ForwardA = 1	Forward result of previous instruction to A (from ALU stage)
ForwardA = 2	Forward result of 2 nd previous instruction to A (from MEM stage)
ForwardA = 3	Forward result of 3 rd previous instruction to A (from WB stage)
ForwardB = 0	Second ALU operand comes from register file = Value of (Rt)
ForwardB = 1	Forward result of previous instruction to B (from ALU stage)
ForwardB = 2	Forward result of 2 nd previous instruction to B (from MEM stage)
ForwardB = 3	Forward result of 3 rd previous instruction to B (from WB stage)

Forwarding Example

Instruction sequence:

lw **R12**, 4(R8)

ori **R15**, R9, 2

sub R11, **R12**, **R15**

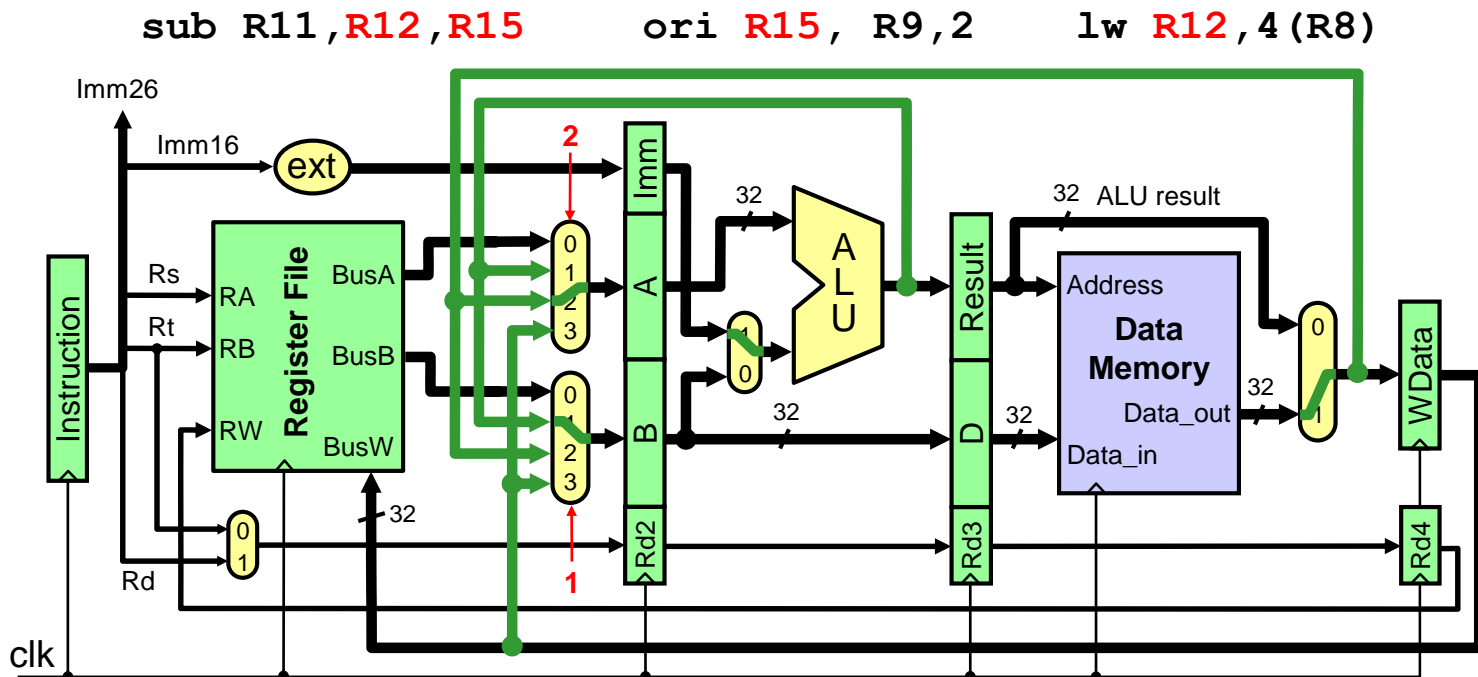
When **sub** instruction is in decode stage

ori will be in the ALU stage

lw will be in the MEM stage

ForwardA = 2 from MEM stage

ForwardB = 1 from ALU stage



RAW Hazard Detection

❖ **Current** instruction being decoded is in **Decode** stage

✧ **Previous** instruction is in the **Execute** stage

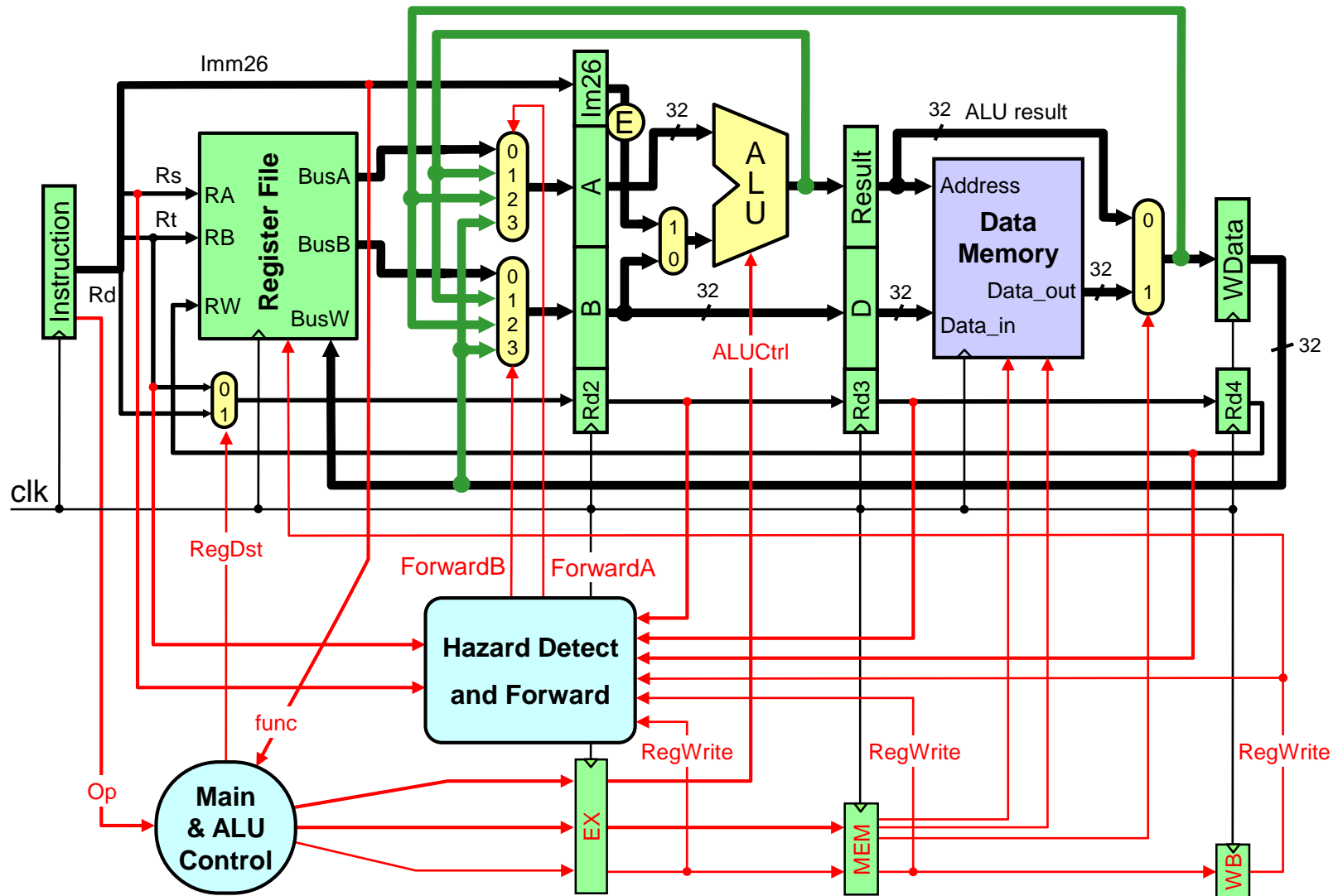
✧ **Second previous** instruction is in the **Memory** stage

✧ **Third previous** instruction in the **Write Back** stage

```
If ((Rs != 0) and (Rs == Rd2) and (EX.RegWrite))      ForwardA ← 1
Else if          ((Rs != 0) and (Rs == Rd3) and (MEM.RegWrite)) ForwardA ← 2
Else if          ((Rs != 0) and (Rs == Rd4) and (WB.RegWrite)) ForwardA ← 3
Else                                                     ForwardA ← 0
```

```
If ((Rt != 0) and (Rt == Rd2) and (EX.RegWrite))      ForwardB ← 1
Else if          ((Rt != 0) and (Rt == Rd3) and (MEM.RegWrite)) ForwardB ← 2
Else if          ((Rt != 0) and (Rt == Rd4) and (WB.RegWrite)) ForwardB ← 3
Else                                                     ForwardB ← 0
```

Hazard Detect and Forward Logic



Next . . .

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Pipeline Stall
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Load Delay

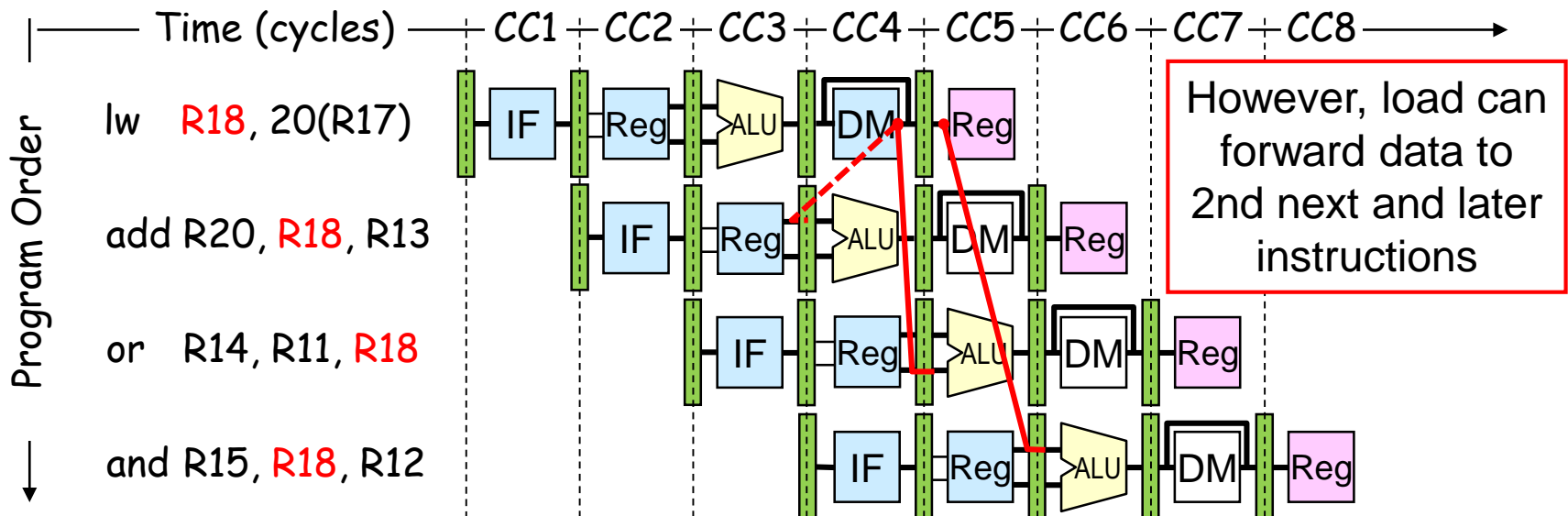
❖ Unfortunately, not all data hazards can be forwarded

✧ **Load** has a delay that cannot be eliminated by forwarding

❖ In the example shown below ...

✧ The **LW** instruction does not read data until end of CC4

✧ Cannot forward data to **ADD** at end of CC3 - **NOT possible**

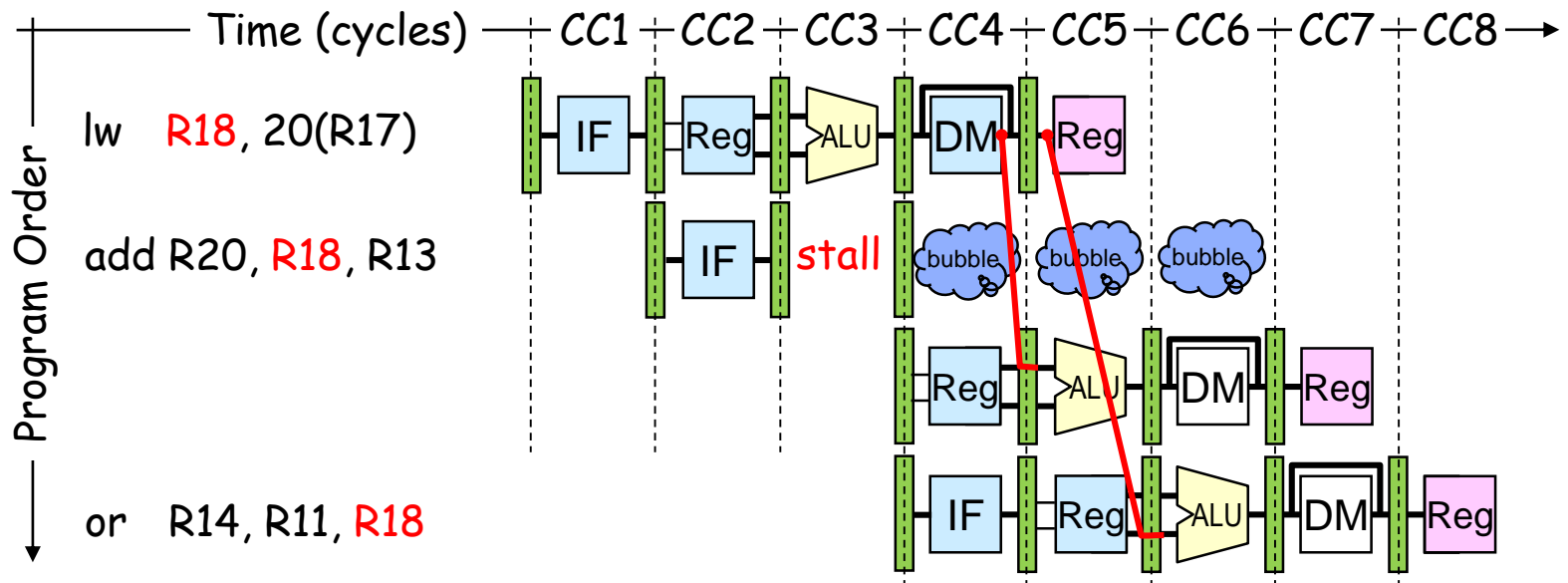


Detecting RAW Hazard after Load

- ❖ Detecting a RAW hazard after a Load instruction:
 - ✧ The **load** instruction will be in the **EX** stage
 - ✧ Instruction that depends on the load data is in the decode stage
- ❖ Condition for stalling the pipeline
 - if ((EX.MemRead == 1) // Detect Load in EX stage
 - and (ForwardA==1 or ForwardB==1)) Stall // RAW Hazard
- ❖ Insert a **bubble** into the EX stage after a load instruction
 - ✧ Bubble is a **no-op** that wastes one clock cycle
 - ✧ Delays the dependent instruction after load by once cycle
 - Because of RAW hazard

Stall the Pipeline for one Cycle

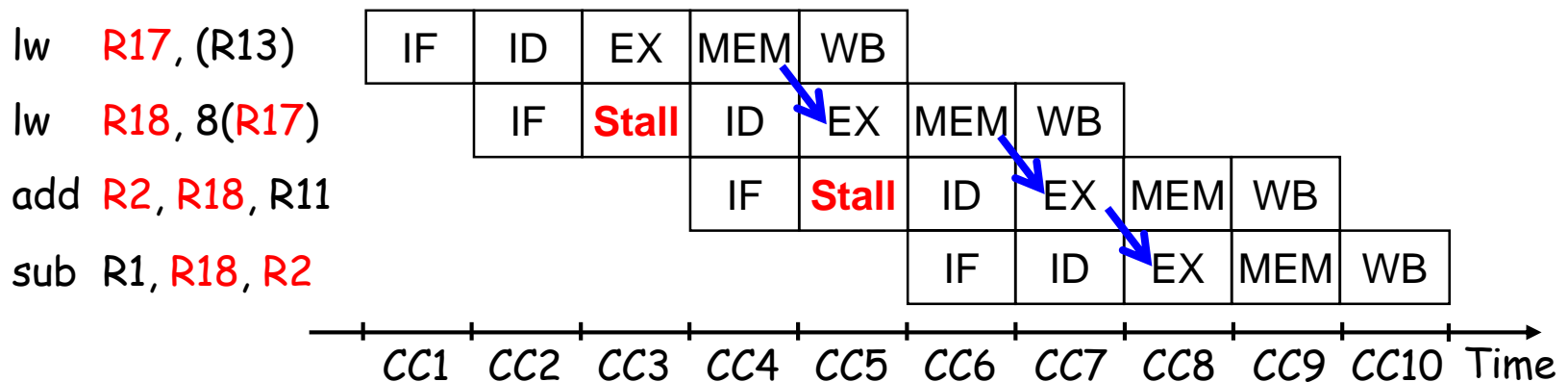
- ❖ **ADD** instruction depends on **LW** → stall at CC3
 - ✧ Allow **Load** instruction in **ALU** stage to proceed
 - ✧ Freeze **PC** and **Instruction** registers (NO instruction is fetched)
 - ✧ Introduce a **bubble** into the **ALU** stage (bubble is a NO-OP)
- ❖ **Load** can forward data to next instruction after delaying it



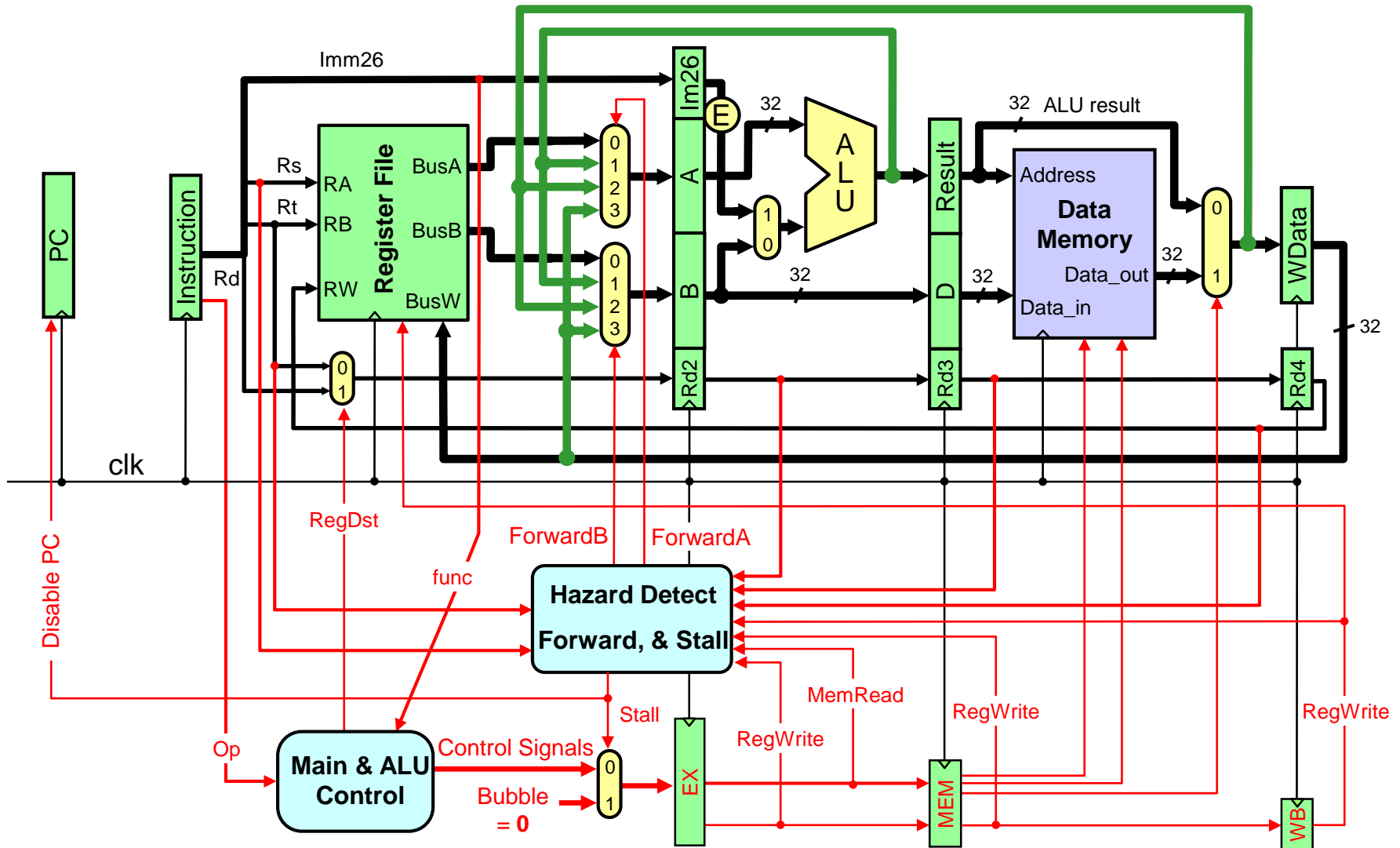
Showing Stall Cycles

- ❖ Stall cycles can be shown on instruction-time diagram
- ❖ Hazard is detected in the Decode stage
- ❖ Stall indicates that instruction is delayed
- ❖ Instruction fetching is also delayed after a stall
- ❖ Example:

Data forwarding is shown using **blue arrows**



Hazard Detect, Forward, and Stall



Code Scheduling to Avoid Stalls

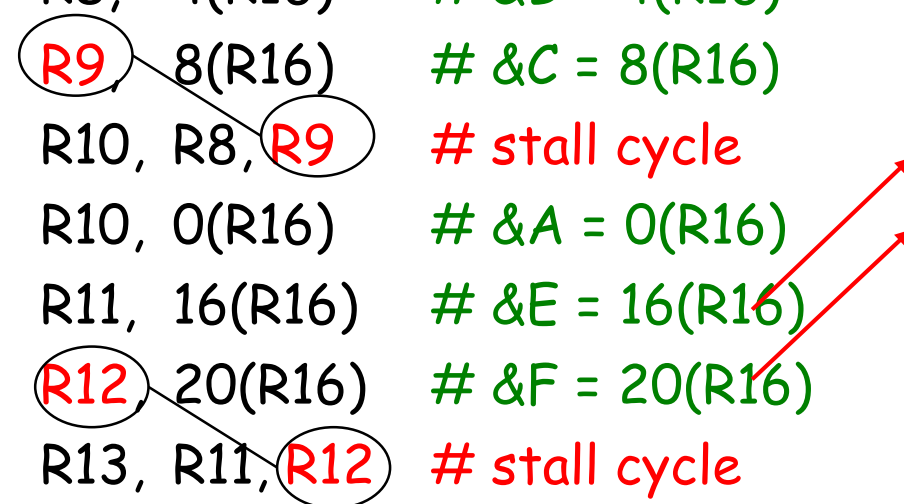
❖ Compilers reorder code in a way to avoid load stalls

❖ Consider the translation of the following statements:

`A = B + C; D = E - F; // A thru F are in Memory`

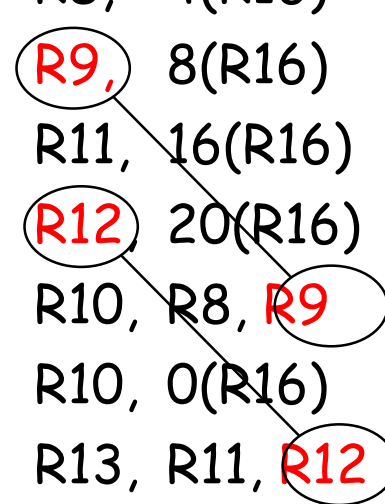
❖ Slow code:

lw	R8,	4(R16)	# &B = 4(R16)
lw	R9 ,	8(R16)	# &C = 8(R16)
add	R10,	R8, R9	# stall cycle
sw	R10,	0(R16)	# &A = 0(R16)
lw	R11,	16(R16)	# &E = 16(R16)
lw	R12 ,	20(R16)	# &F = 20(R16)
sub	R13,	R11, R12	# stall cycle
sw	R13,	12(R0)	# &D = 12(R0)



❖ Fast code: No Stalls

lw	R8,	4(R16)	
lw	R9 ,	8(R16)	
lw	R11,	16(R16)	
lw	R12 ,	20(R16)	
add	R10,	R8, R9	
sw	R10,	0(R16)	
sub	R13,	R11, R12	
sw	R13,	12(R0)	



Name Dependence: Write After Read

- ❖ Instruction J writes its result after it is read by I
- ❖ Called **anti-dependence** by compiler writers
 - I: `sub R12, R9, R11` # R9 is read
 - J: `add R9, R10, R11` # R9 is written
- ❖ Results from reuse of the name **R9**
- ❖ NOT a data hazard in the 5-stage pipeline because:
 - ✧ Reads are always in stage 2
 - ✧ Writes are always in stage 5, and
 - ✧ Instructions are processed in order
- ❖ Anti-dependence can be eliminated by **renaming**
 - ✧ Use a different destination register for `add` (eg, **R13**)

Name Dependence: Write After Write

- ❖ Same destination register is written by two instructions
- ❖ Called **output-dependence** in compiler terminology
 - I: `sub R9, R12, R11 # R9 is written`
 - J: `add R9, R10, R11 # R9 is written again`
- ❖ Not a data hazard in the 5-stage pipeline because:
 - ✧ All writes are ordered and always take place in stage 5
- ❖ However, can be a hazard in more complex pipelines
 - ✧ If instructions are allowed to complete out of order, and
 - ✧ Instruction J completes and writes **R9** before instruction I
- ❖ Output dependence can be eliminated by **renaming R9**
- ❖ **Read After Read is NOT a name dependence**

Hazards due to Loads and Stores

❖ Consider the following statements:

```
sw      R10, 0(R1)
lw      R11, 6(R5)
```

Is there any Data Hazard possible in the above code sequence?

If $0(R1) == 6(R5)$, then it means the same memory location!!

- ✧ Data dependency through Data Memory
- ✧ But no Data Hazard since the memory is not accessed by the two instructions simultaneously – writing and reading happens in two consecutive clock cycles
- ✧ But, in an **out-of-order execution** processor, this can lead to a Hazard!

Next . . .

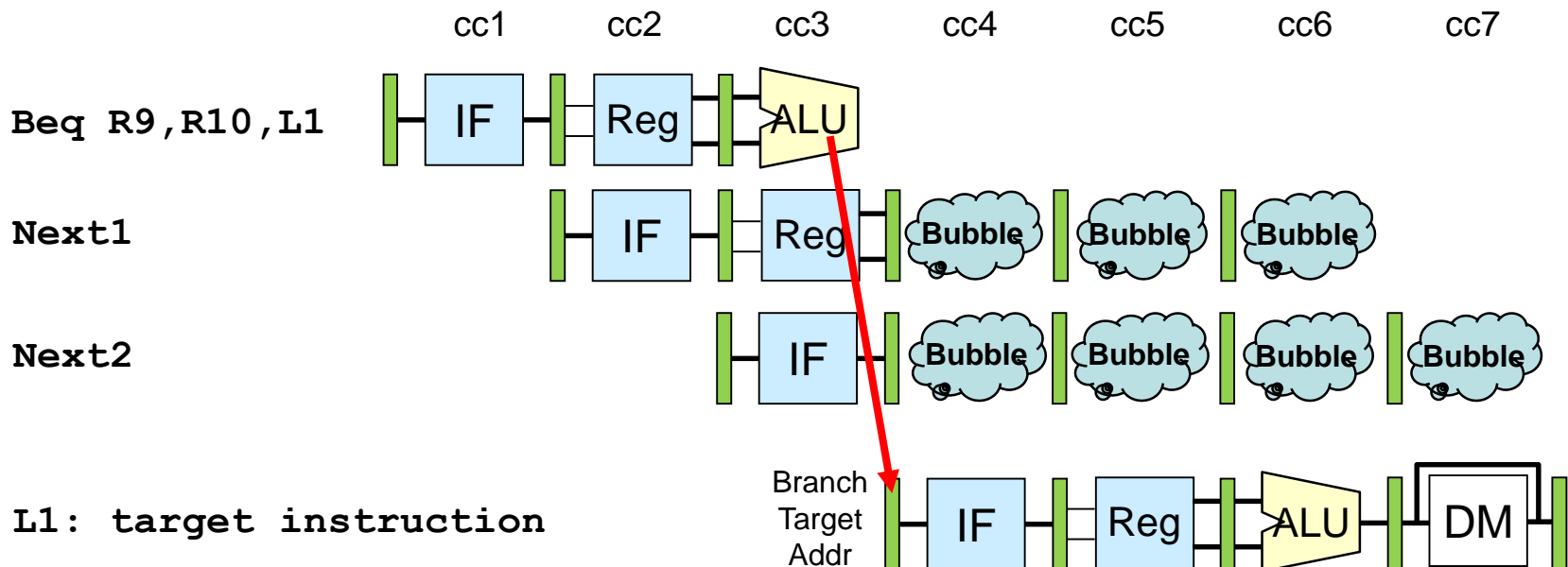
- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ **Control Hazards**
- ❖ Delayed Branch and Dynamic Branch Prediction

Control Hazards

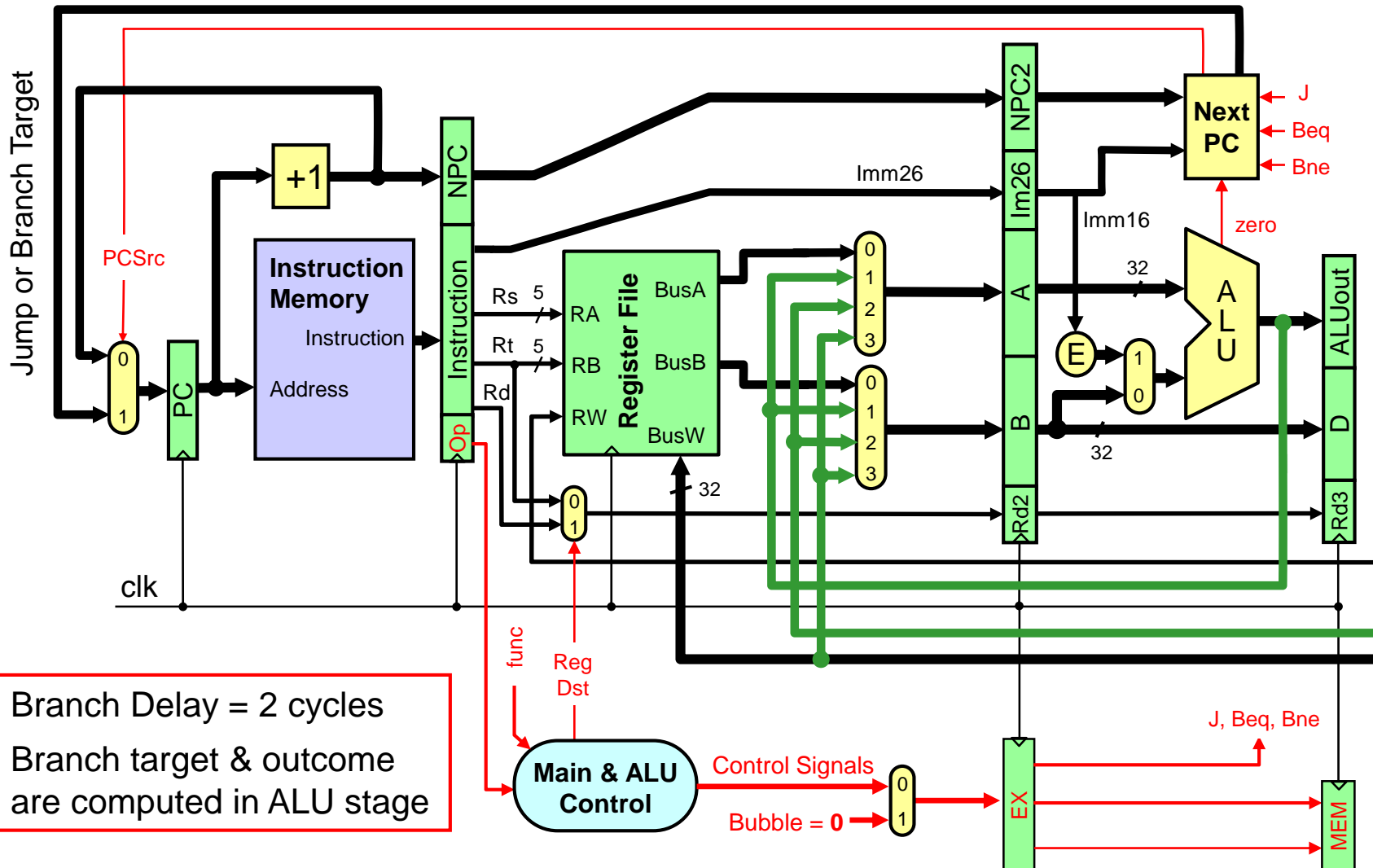
- ❖ Jump and Branch can cause great performance loss
- ❖ Jump instruction needs only the **jump target address**
- ❖ Branch instruction needs two things:
 - ✧ **Branch Result** Taken or Not Taken
 - ✧ **Branch Target Address**
 - $PC + 4$ If Branch is NOT taken
 - $PC + 4 \times \text{immediate}$ If Branch is Taken
- ❖ Jump and Branch targets are computed in the ID stage
 - ✧ At which point a new instruction is already being fetched
 - ✧ Jump Instruction: 1-cycle delay
 - ✧ Branch: 2-cycle delay for branch result (taken or not taken)

2-Cycle Branch Delay

- ❖ Control logic detects a **Branch** instruction in the 2nd Stage
- ❖ ALU computes the **Branch outcome** in the 3rd Stage
- ❖ **Next1** and **Next2** instructions will be fetched anyway
- ❖ Convert **Next1** and **Next2** into bubbles **if branch is taken**

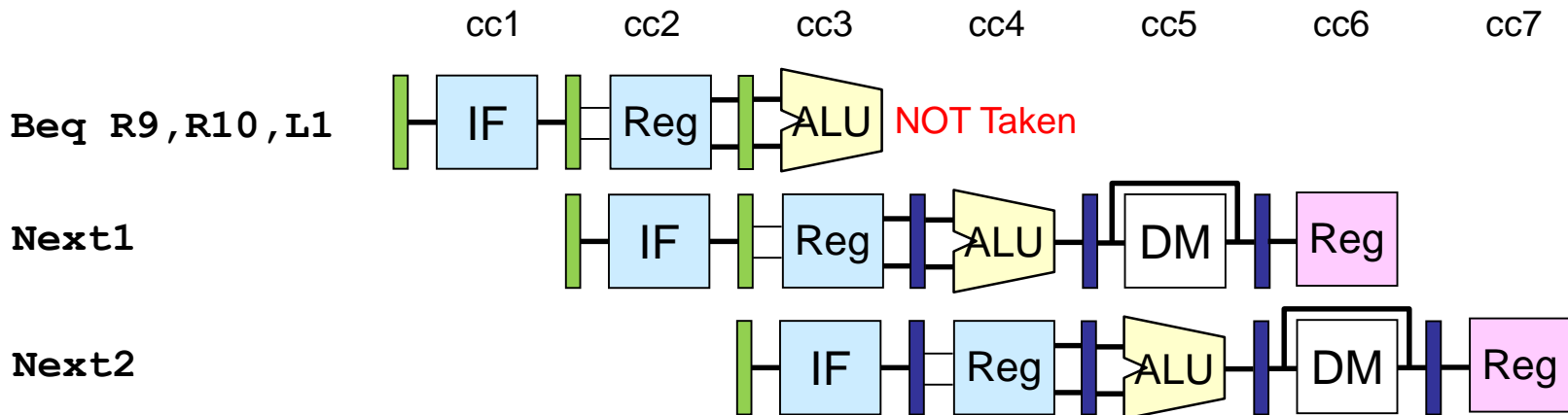


Implementing Jump and Branch



Predict Branch NOT Taken

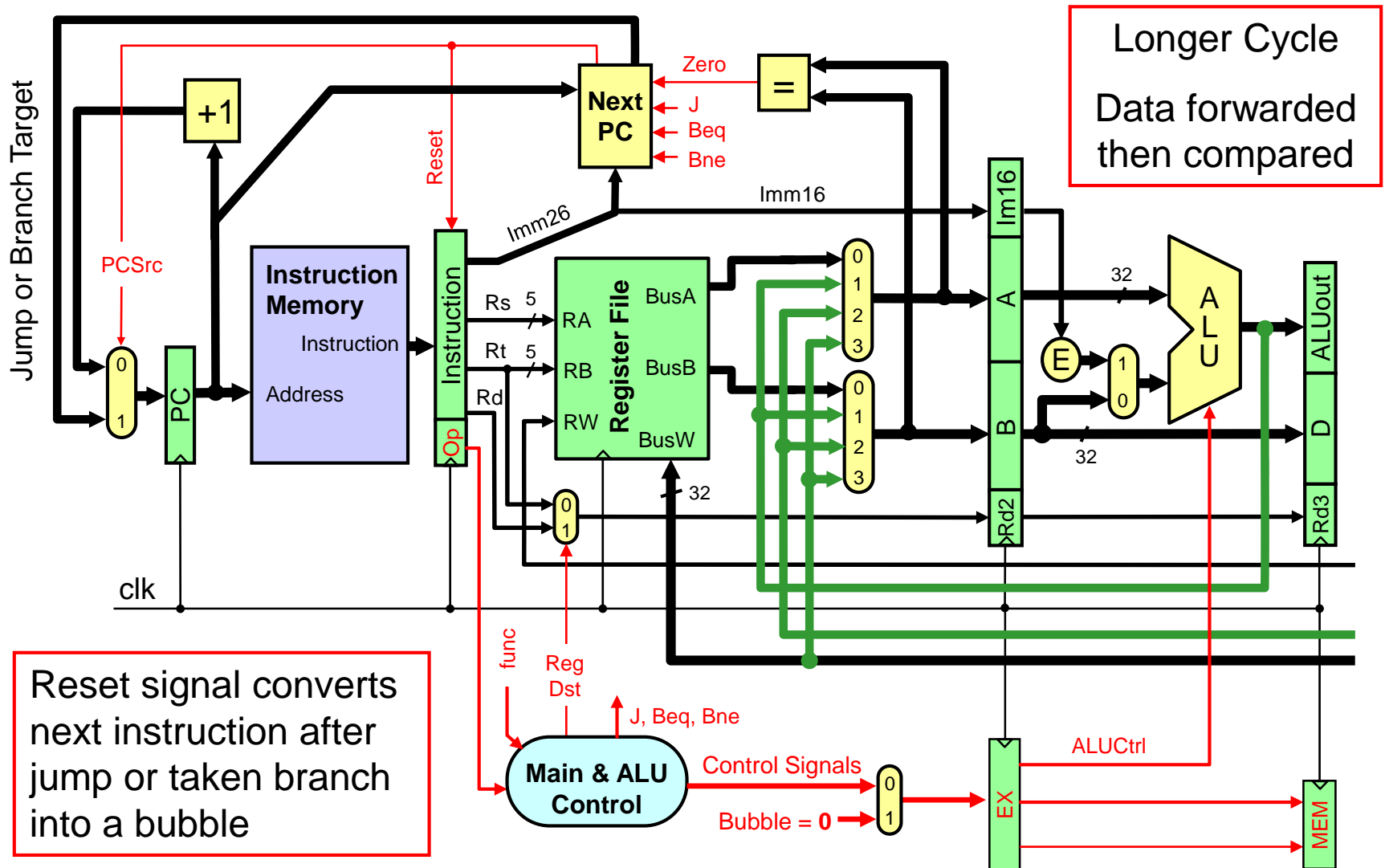
- ❖ Branches can be predicted to be NOT taken
- ❖ If **branch outcome** is **NOT taken** then
 - ✧ **Next1** and **Next2** instructions can be executed
 - ✧ Do not convert **Next1** & **Next2** into bubbles
 - ✧ **No wasted cycles**
- ❖ Else, convert **Next1** and **Next2** into bubbles – 2 wasted cycles



Reducing the Delay of Branches

- ❖ Branch delay can be reduced from 2 cycles to **just 1 cycle**
- ❖ Branches can be determined earlier in the Decode stage
 - ✧ A comparator is used in the decode stage to determine branch decision, whether the branch is taken or not
 - ✧ Because of forwarding the delay in the second stage will be increased and this will also increase the clock cycle
- ❖ Only **one instruction** that follows the branch is fetched
- ❖ If the branch is taken then only one instruction is flushed
- ❖ We should insert a bubble after jump or taken branch
 - ✧ This will convert the next instruction into a NOP

Reducing Branch Delay to 1 Cycle



Branch Behavior in Programs

- ❖ Based on SPEC benchmarks on DLX
 - ✧ Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
 - ✧ About 75% of the branches are forward branches
 - ✧ 60% of forward branches are taken
 - ✧ 85% of backward branches are taken
 - ✧ 67% of all branches are taken
- ❖ Why are branches (especially backward branches) more likely to be taken than not taken?

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- ✧ Execute successor instructions in sequence
- ✧ “Flush” instructions in pipeline if branch actually taken
- ✧ Advantage of late pipeline state update
- ✧ 33% DLX branches not taken on average
- ✧ PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- ✧ 67% DLX branches taken on average

#4: Define branch to take place **AFTER** **n** following instruction
(Delayed branch)

Next . . .

- ❖ Pipelining versus Serial Execution
- ❖ Pipelined Datapath and Control
- ❖ Pipeline Hazards
- ❖ Data Hazards and Forwarding
- ❖ Load Delay, Hazard Detection, and Stall
- ❖ Control Hazards
- ❖ Delayed Branch and Dynamic Branch Prediction

Branch Hazard Alternatives

- ❖ **Predict Branch Not Taken** (previously discussed)
 - ✧ Successor instruction is already fetched
 - ✧ Do NOT Flush instruction after branch if branch is NOT taken
 - ✧ Flush only instructions appearing after Jump or taken branch
- ❖ **Delayed Branch**
 - ✧ Define branch to take place **AFTER** the next instruction
 - ✧ Compiler/assembler **fills the branch delay slot (for 1 delay cycle)**
- ❖ **Dynamic Branch Prediction**
 - ✧ Loop branches are taken most of time
 - ✧ Must reduce branch delay to 0, but how?
 - ✧ How to predict branch behavior at runtime?

Delayed Branch

- ❖ Define branch to take place **after** the next instruction
- ❖ Instruction in branch delay slot is always executed
- ❖ Compiler (tries to) move a useful instruction into delay slot.

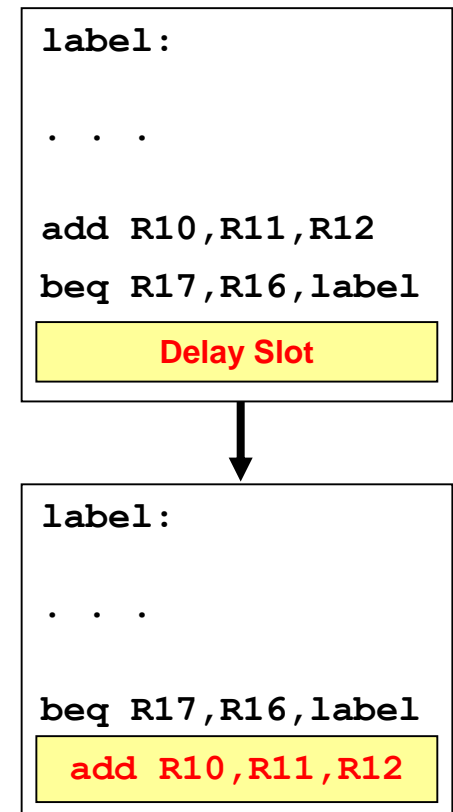
From before the Branch: Always helpful when possible

- ❖ For a 1-cycle branch delay, we have **one delay slot** branch instruction

branch delay slot (next instruction)

branch target (if branch taken)

- ❖ Compiler **fills the branch delay slot**
 - ✧ By selecting an **independent instruction**
 - ✧ From before the branch
- ❖ If no independent instruction is found
 - ✧ Compiler fills delay slot with a NO-OP



Delayed Branch

From the Target: Helps when branch is taken. May duplicate instructions

```
ADD R2, R1, R3
```

```
BEQZ R2, L1
```

```
DELAY SLOT
```

```
-
```

```
L1: SUB R4, R5, R6
```

```
L2:
```

```
ADD    R2, R1, R3
```

```
BEQZ   R2, L2
```

```
SUB R4, R5, R6
```

```
-
```

```
L1:    SUB R4, R5, R6
```

```
L2:
```

Instructions between **BEQ** and **SUB** (in fall through) **must not use R4**.

Why is instruction at **L1 duplicated**? What if **R5** or **R6** changed?

(Because, **L1 can be reached by another path !!**)

Delayed Branch

From Fall Through: Helps when branch is not taken (default case)

```
ADD R2, R1, R3
BEQZ R2, L1
DELAY SLOT
SUB R4, R5, R6
-
```

L1:

```
ADD    R2, R1, R3
BEQZ   R2, L1
SUB R4, R5, R6
-
```

L1:

Instructions at target (L1 and after) **must not** use R4 till set (written) again.

Filling delay slots

- ❖ Compiler effectiveness for single branch delay slot:
 - ✧ Fills about 60% of branch delay slots
 - ✧ About 80% of instructions executed in branch delay slots are useful in computation
 - ✧ About 50% (i.e., $60\% \times 80\%$) of slots usefully filled
- ❖ Canceling branches or nullifying branches
 - ✧ Include a prediction of **the branch is taken** or **not taken**
 - ✧ If the prediction is correct, the instruction in the delay slot is executed
 - ✧ If the prediction is incorrect, the instruction in the delay slot is quashed.
 - ✧ Allow more slots to be filled from the target address or fall through

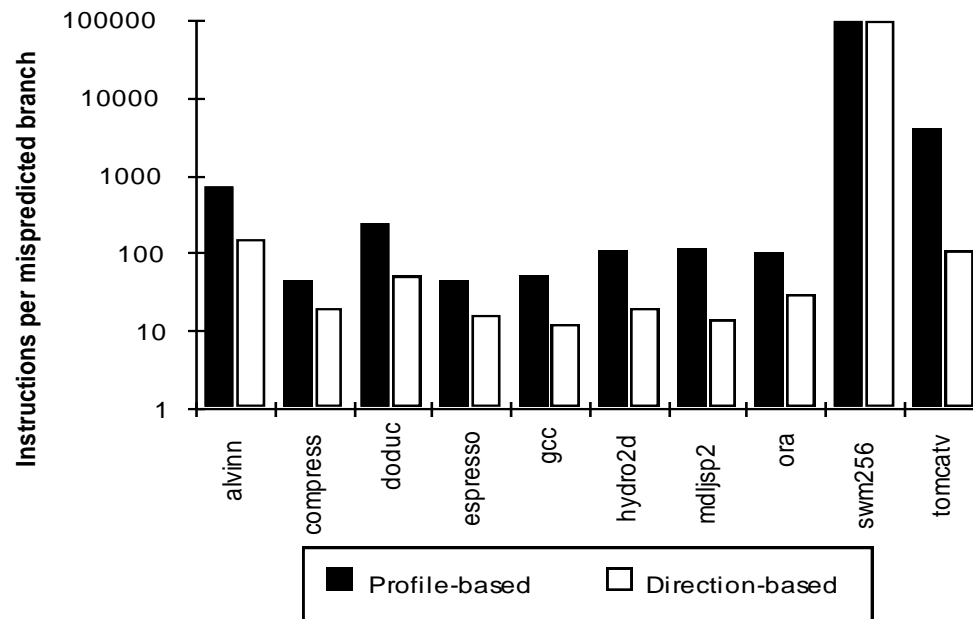
Drawback of Delayed Branching

- ❖ New meaning for branch instruction
 - ✧ Branching takes place after next instruction (Not immediately!)
- ❖ Impacts software and compiler
 - ✧ Compiler is responsible to fill the branch delay slot
 - ✧ For a 1-cycle branch delay → **One branch delay slot**
- ❖ However, modern processors are deeply pipelined
 - ✧ Branch penalty is multiple cycles in deeper pipelines
 - ✧ Multiple delay slots are difficult to fill with useful instructions
- ❖ MIPS used delayed branching in earlier pipelines
 - ✧ However, delayed branching is not useful in recent processors

Compiler "Static" Prediction of Taken/Untaken Branches

❖ Two strategies examined

- ✧ Backward branch predict taken, forward branch not taken
- ✧ Profile-based prediction: record branch behavior, predict branch based on prior run



Zero-Delayed Branching

❖ How to achieve **zero delay for a jump or a taken branch?**

- ✧ Jump or branch target address is computed in the ID stage
- ✧ Next instruction has already been fetched in the IF stage

Solution

❖ Introduce a **Branch Target Buffer (BTB)** in the **IF stage**

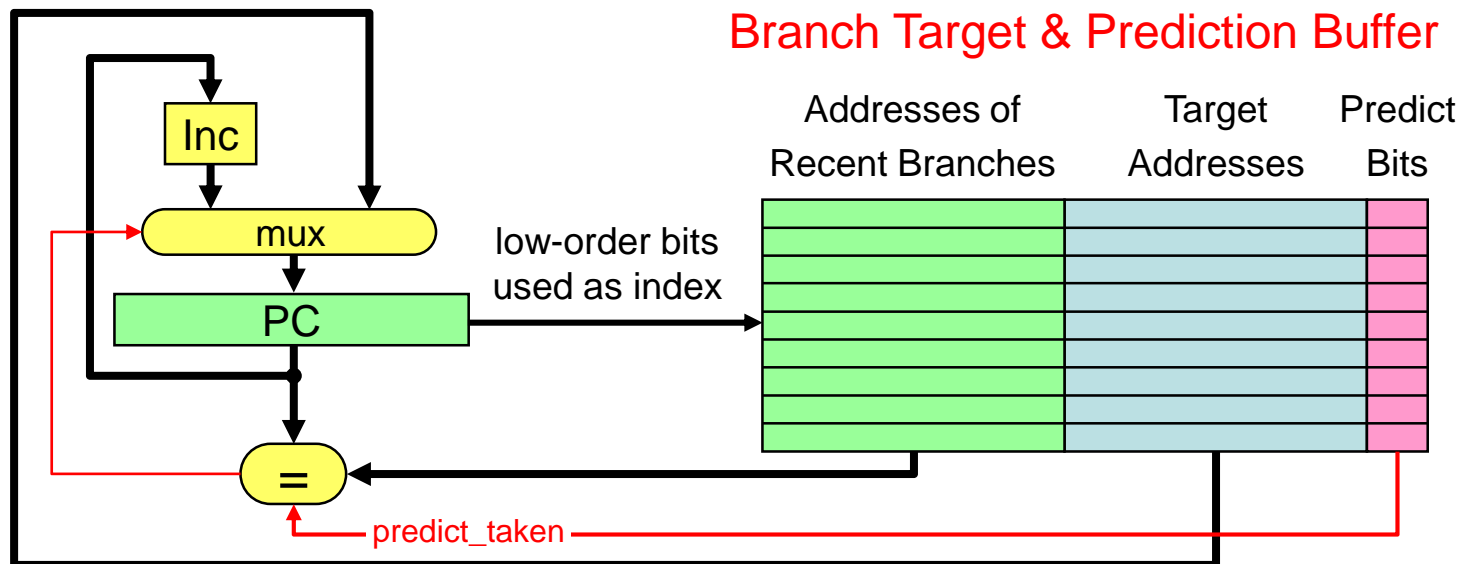
- ✧ Store the target address of recent branch and jump instructions

❖ Use the lower bits of the PC to index the BTB

- ✧ Each BTB entry stores Branch/Jump address & Target Address
- ✧ Check the PC to see if the instruction being fetched is a branch
- ✧ Update the PC using the target address stored in the BTB

Branch Target Buffer

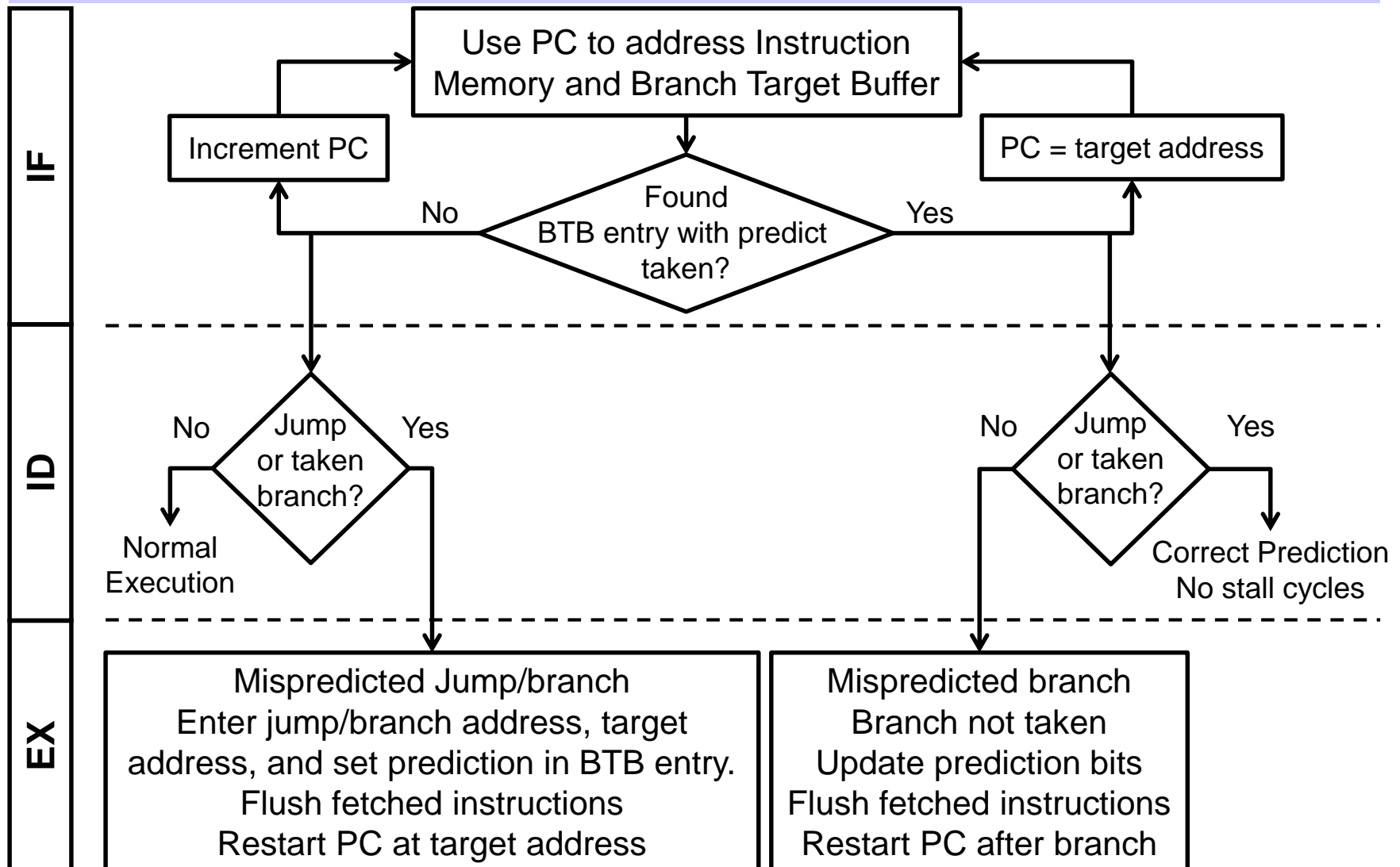
- ❖ The **branch target buffer** is implemented as a small cache
 - ✧ Stores the target address of recent branches and jumps
- ❖ We must also have **prediction bits**
 - ✧ To **predict** whether branches are taken or not taken
 - ✧ The prediction bits are dynamically determined by the hardware



Dynamic Branch Prediction

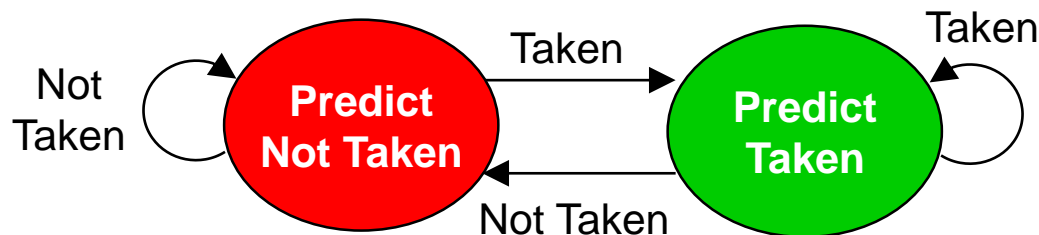
- ❖ Prediction of branches at runtime using **prediction bits**
- ❖ Prediction bits are associated with each entry in the BTB
 - ✧ Prediction bits reflect the recent history of a branch instruction
- ❖ Typically few prediction bits (1 or 2) are used per entry
- ❖ We don't know if the prediction is correct or not
- ❖ If correct prediction ...
 - ✧ Continue normal execution – no wasted cycles
- ❖ If incorrect prediction (misprediction) ...
 - ✧ Flush the instructions that were incorrectly fetched – wasted cycles
 - ✧ Update prediction bits and target address for future use

Dynamic Branch Prediction - Cont'd



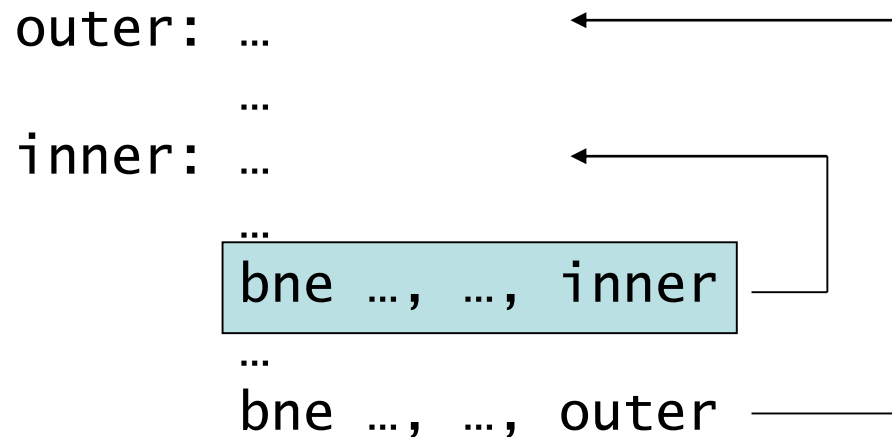
1-bit Prediction Scheme

- ❖ Prediction is just a hint that is assumed to be correct
- ❖ If incorrect then fetched instructions are flushed
- ❖ 1-bit prediction scheme is simplest to implement
 - ✧ 1 bit per branch instruction (associated with BTB entry)
 - ✧ Record last outcome of a branch instruction (Taken/Not taken)
 - ✧ Use last outcome to predict future behavior of a branch



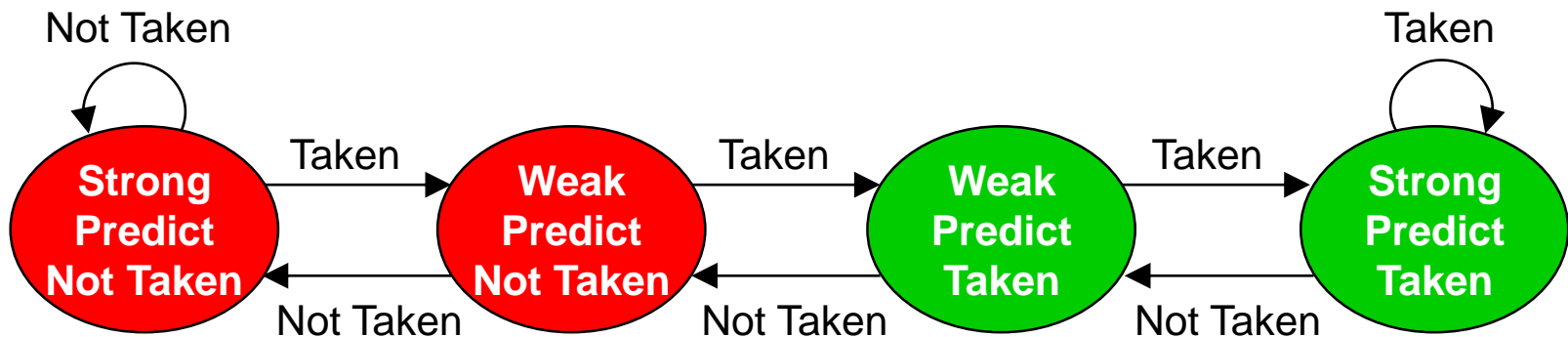
1-Bit Predictor: Shortcoming

- ❖ Inner loop branch mispredicted twice!
 - ✧ Mispredict as taken on last iteration of inner loop
 - ✧ Then mispredict as not taken on first iteration of inner loop next time around



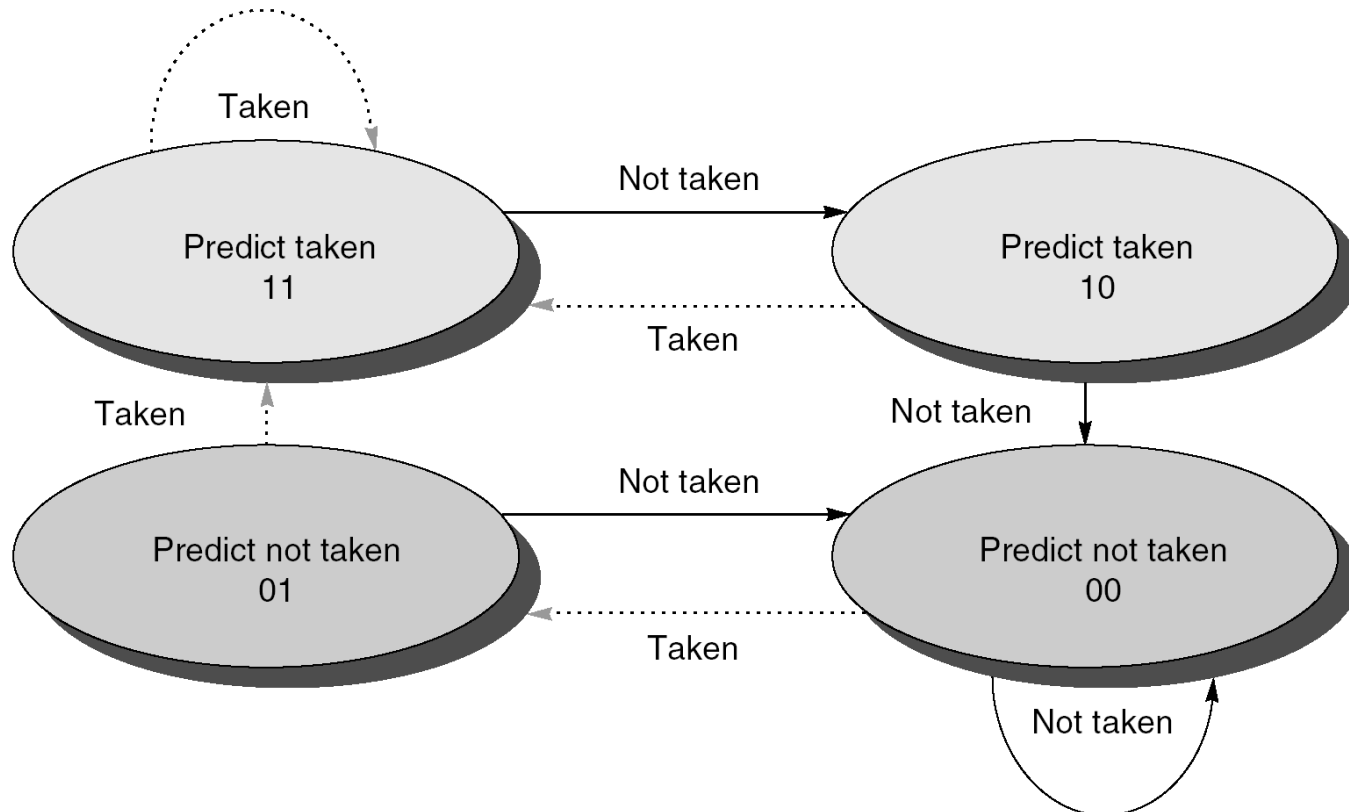
2-bit Prediction Scheme

- ❖ 1-bit prediction scheme has a performance shortcoming
- ❖ 2-bit prediction scheme works better and is often used
 - ✧ 4 states: strong and weak predict taken / predict not taken
- ❖ Implemented as a **saturating counter**
 - ✧ Counter is incremented to max=3 when branch outcome is taken
 - ✧ Counter is decremented to min=0 when branch is not taken



2-bit Prediction Scheme

❖ Alternative state machine



1-Bit Branch History Table

Example

```
main()
{
  int i, j;
  while (1)
    for (i=1; i<=4; i++) {
      j = i++;
    }
}
```

'while' branch : always TAKEN

1111111111111111....

'for' branch : TAKEN x 3, NOT TAKEN x 1

1110111011101110....

branch outcome	1	1	1	0	1	1	1	0	1	1	1	0	1	...
last outcome	1	1	1	1	0	1	1	1	0	1	1	1	0	...
prediction	T	T	T	T	N	T	T	T	N	T	T	T	N	...
new last outcome	1	1	1	0	1	1	1	0	1	1	1	0	1	...
correctness	O	O	O	X	X	O	O	X	X	O	O	X	X	...

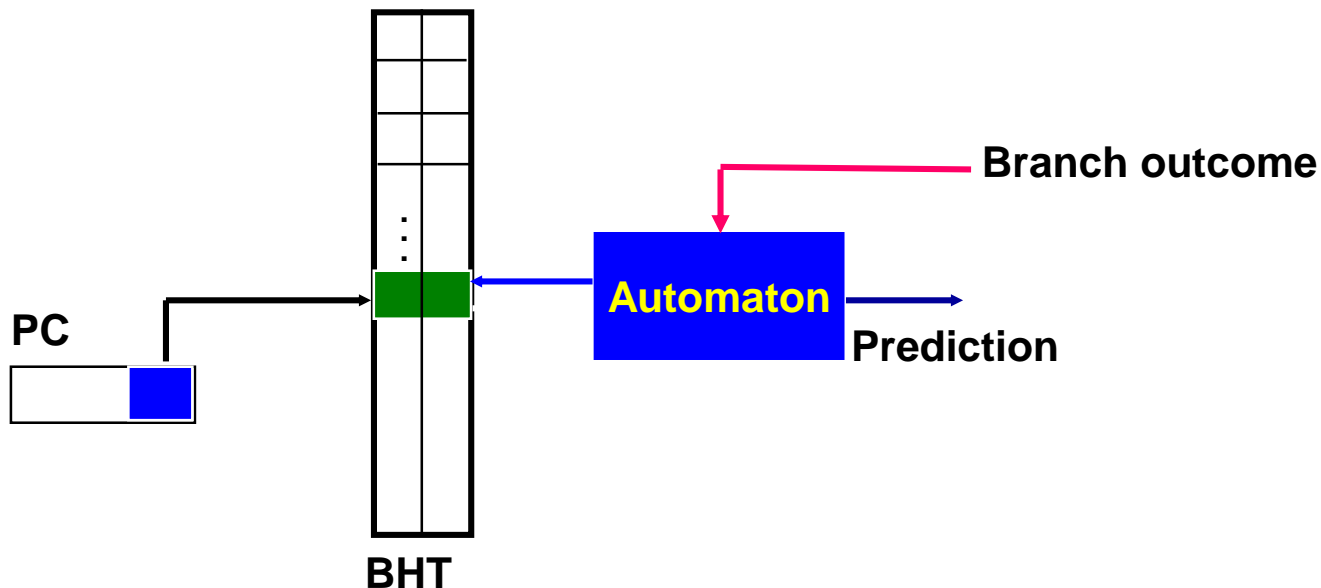
Assume initial last outcome = 1

O : correct, X : mispredict

Prediction accuracy of 'for' branch: 50%

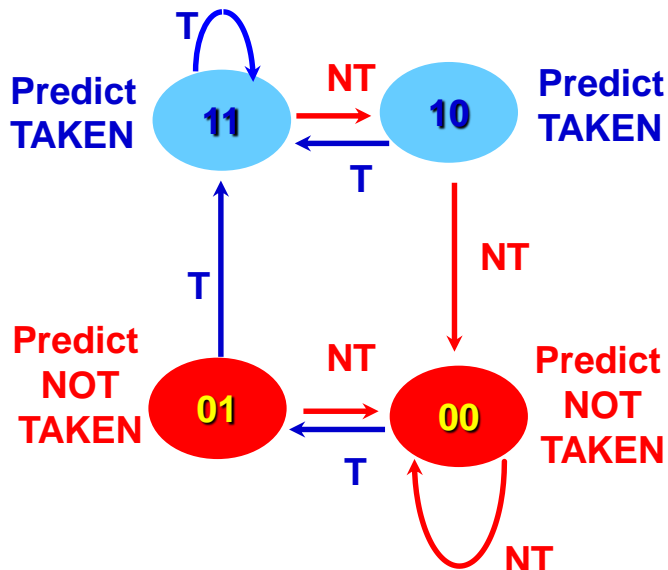
2-Bit Counter Scheme

- ❖ Prediction is made based on the *last two branch outcomes*
- ❖ Each of BHT entries consists of 2 bits, usually a 2-bit counter, which is associated with the state of the automaton (many different automata are possible)



2-Bit BHT

- ❖ 2-bit scheme where *change prediction only if get misprediction twice*
- ❖ MSB of the state symbol represents the prediction;
 - ✧ 1: TAKEN, 0: NOT TAKEN



branch outcome	1	1	1	0	1	1	1	0	1	1	1	0	...
counter value	11	11	11	11	10	11	11	11	10	11	11	11	...
prediction	T	T	T	T	T	T	T	T	T	T	T	T	...
new counter value	11	11	11	10	11	11	11	10	11	11	11	10	...
correctness	O	O	O	X	O	O	O	X	O	O	O	X	...

assume initial counter value : 11 O : correct, X : mispredict

Prediction accuracy of 'for' branch: 75 %

Case for Correlating Predictors

❖ Basic two-bit predictor schemes

- ✧ use recent behavior of a branch to predict its future behavior

❖ Improve the prediction accuracy

- ✧ look also at recent behavior of other branches

```
subi R3, R1, #2
bnez R3, L1      ; b1
add R1, R0, R0
L1: subi R3, R1, #2
    bnez R3, L2      ; b2
    add R2, R0, R0
L2: sub R3, R1, R2
    beqz R3, L3      ; b3
```

if (aa == 2) aa = 0;

if (bb == 2) bb = 0;

if (aa != bb) { }

b3 is correlated with b1 and b2;

If b1 and b2 are both untaken in the above code,

then b3 will be taken.

=>

Use correlating predictors or
two-level predictors.

Correlating Branches

Example:

if (d==0)

d=1; →

if (d==1)

b1 :

.....

b2 :

BNEZ R1,b1 ; (b1)(d!=0)
 ADDI R1,R1,#1; since d==0, make d=1
 SUBI R3,R1,#1
 BNEZ R3,b2; (b2)(d!=1)

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	Y	NT	1	Y	NT
1	N	T	1	Y	NT
2	N	T	2	N	T

If b1 is NT, then b2 is NT

1-bit self history predictor							
	d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
Sequence of 2,0,2,0,2,0,...	2	NT	T	T	NT	T	T
	0	T	NT	NT	T	NT	NT
	2	NT	T	T	NT	T	T
	0	T	NT	NT	T	NT	NT

All branches are mispredicted

Correlating Branches

Example:

if (d==0)

if (d==1)

d=1; →

```

b1 :    BNEZ    R1,L1 ;   branch b1 (d!=0)
        ADDI    R1,R1,#1; since d==0, make d=1
        SUBI    R3,R1,#1
        BNEZ    R3,L2;   branch b2(d!=1)
.....
b2 :
    
```

Self
Prediction
bits(XX)

NT/NT
NT/T
T/NT
T/T

Prediction, if last
branch action was NT

NT
NT
T
T

Global

Prediction, if last
branch action was T

NT
T
NT
T

d=?	b1 prediction	b1 action	new b1 prediction	b2 prediction	b2 action	new b2 prediction
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
1	T/NT	T	T/NT	NT/T	NT	NT/T

Initial self prediction bits NT/NT and
Initial last branch was NT.

Prediction used is shown in Red

Misprediction only in the first prediction !!

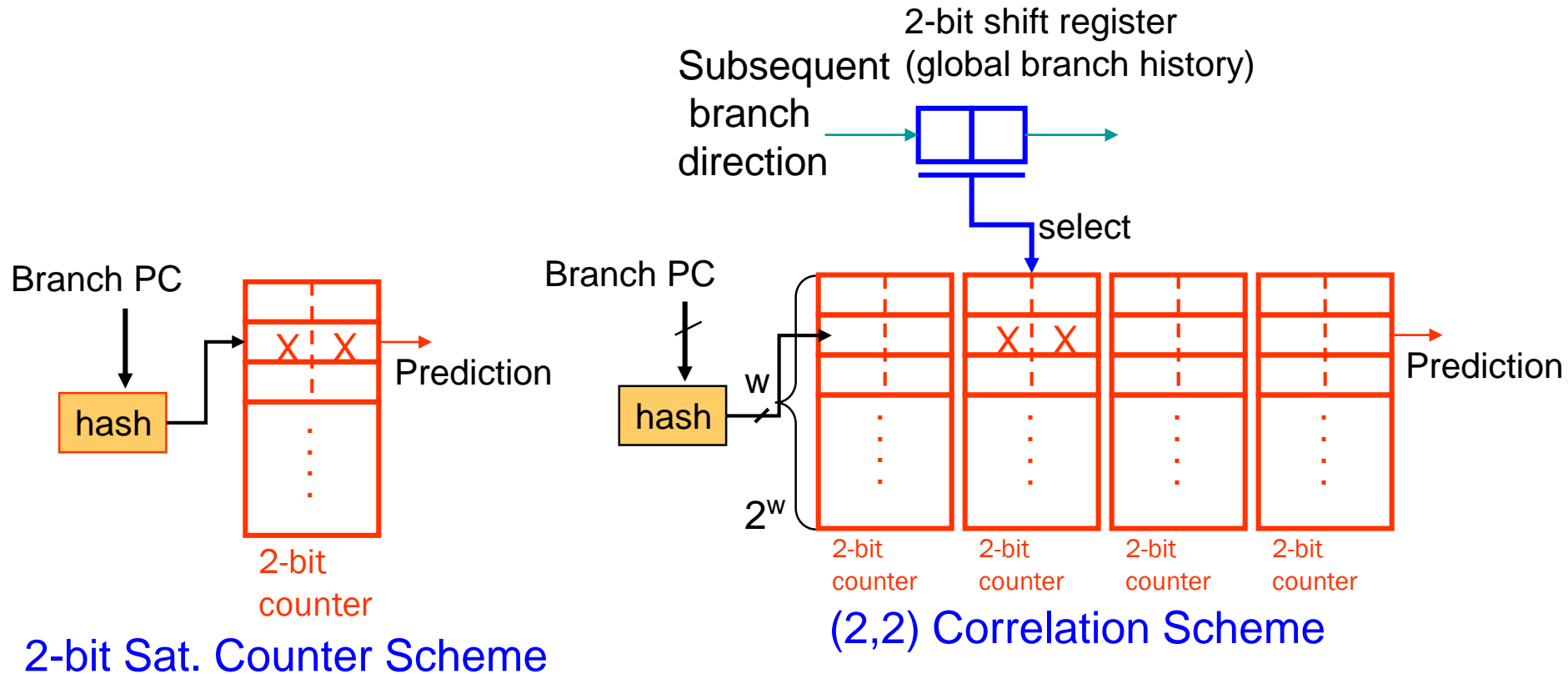
Local/Global Predictors

- ❖ Instead of maintaining a counter for each branch to capture the common case,
 - ✧ Maintain a counter for each branch and surrounding pattern
 - ✧ If the surrounding pattern belongs to the branch being predicted, the predictor is referred to as a **local predictor**
 - ✧ If the surrounding pattern includes neighboring branches, the predictor is referred to as a **global predictor**

Correlated Branch Prediction

- ❖ **Idea**: record m most recently executed branches as taken or not taken, and use that pattern to select the proper n -bit branch history table
- ❖ In general, **(m,n) predictor** means record last m branches to select between 2^m history tables, each with n -bit counters
 - ✧ Thus, old 2-bit BHT is a $(0,2)$ predictor
- ❖ **Global Branch History**: m -bit shift register keeping T/NT status of last m branches.
- ❖ Each entry in table has m n -bit predictors.

Correlated Branch Predictor



❖ (M,N) correlation scheme

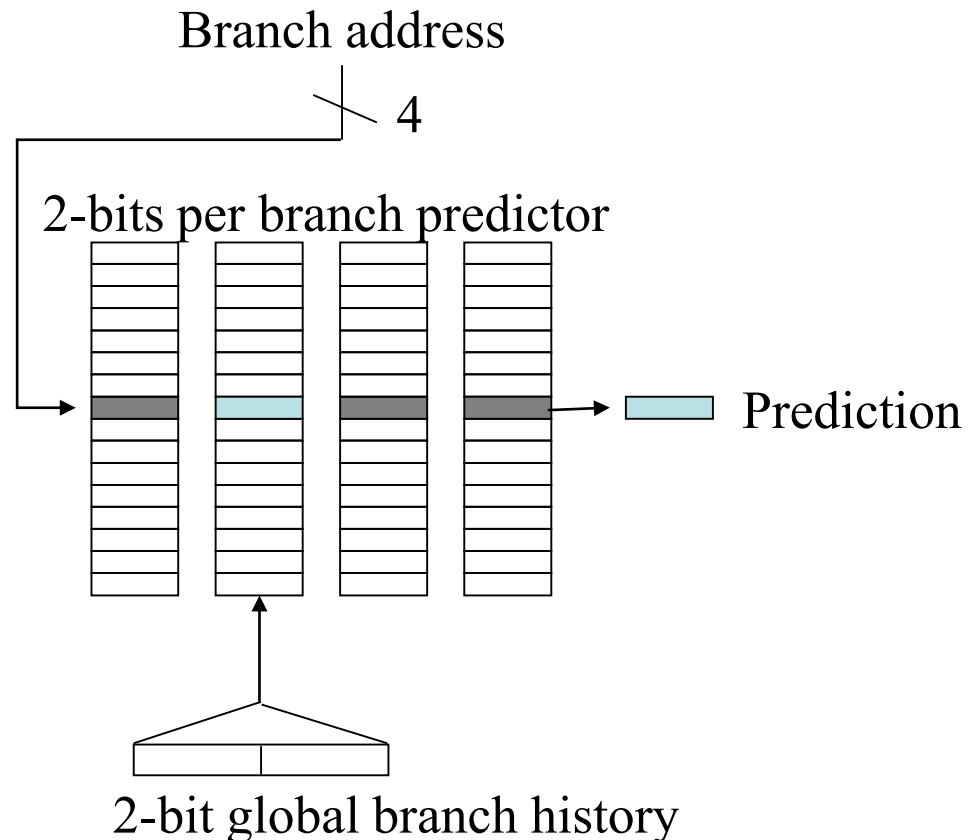
✧ M: shift register size (# bits)

✧ N: N-bit counter

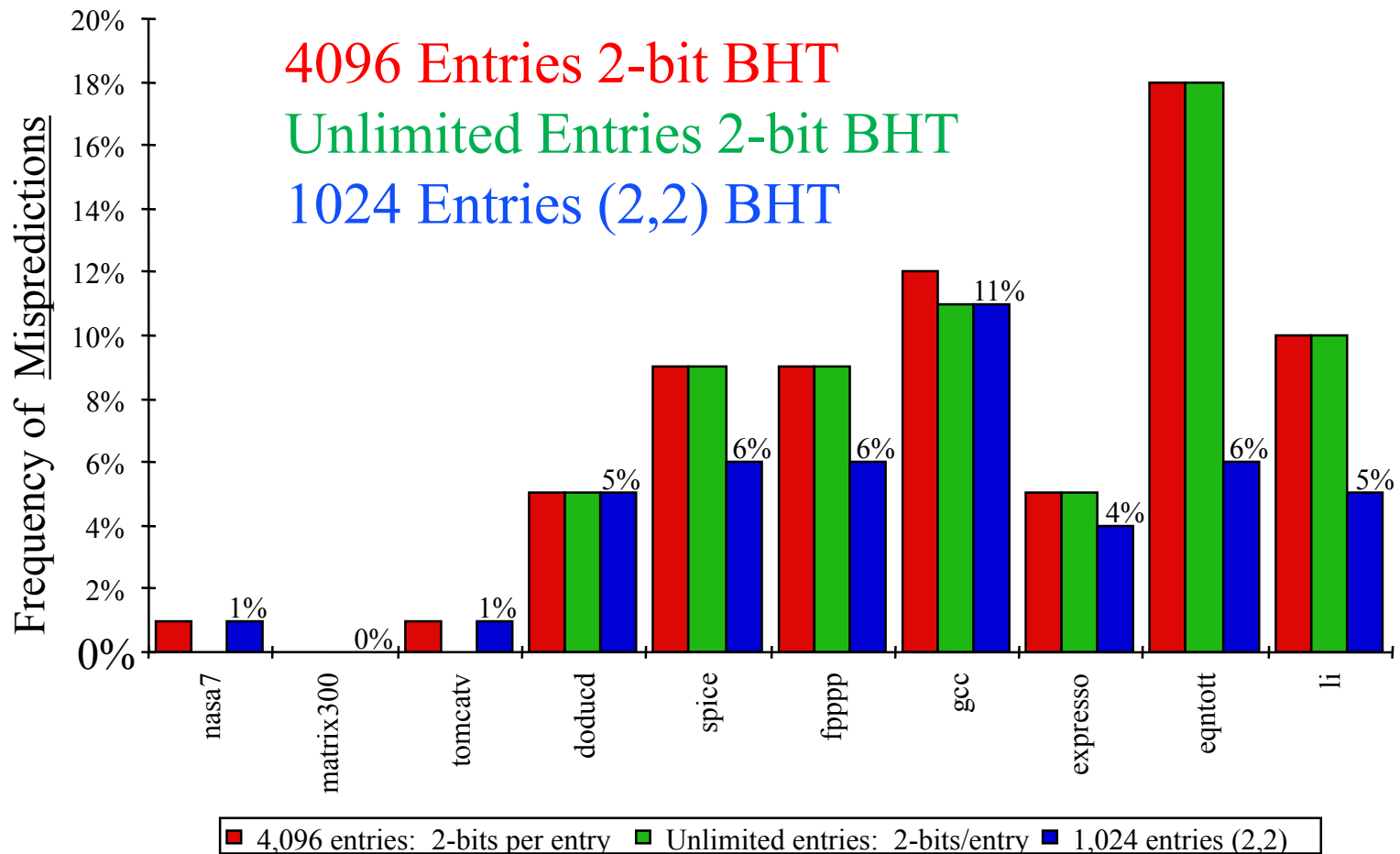
Correlating Branches

(2,2) predictor

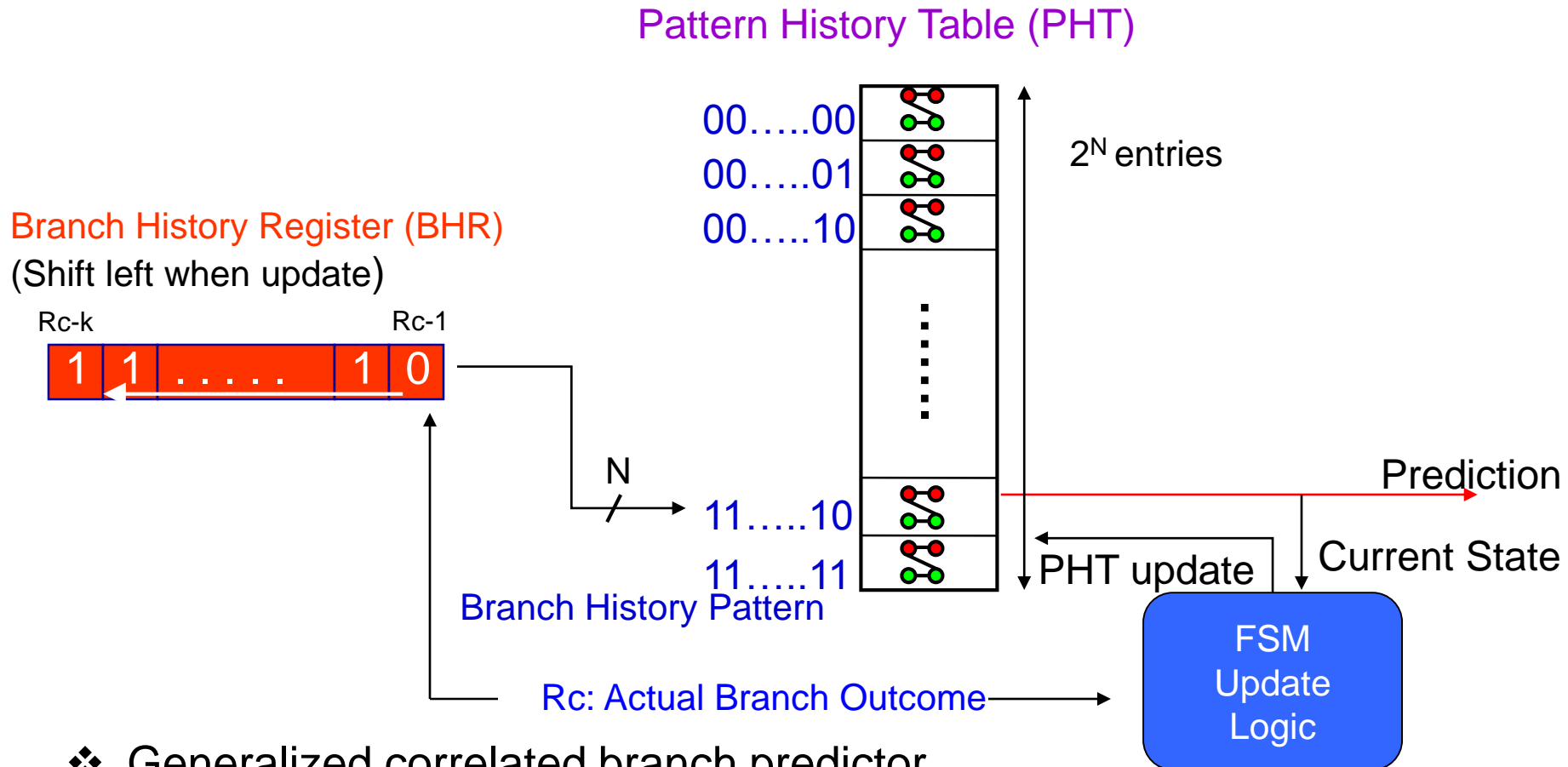
- Behavior of recent branches selects between four predictions of next branch, updating just that prediction



Accuracy of Different Schemes



Two-Level Branch Predictor



- ❖ Generalized correlated branch predictor
- ❖ 1st level keeps branch history in Branch History Register (BHR)
- ❖ 2nd level segregates pattern history in Pattern History Table (PHT)

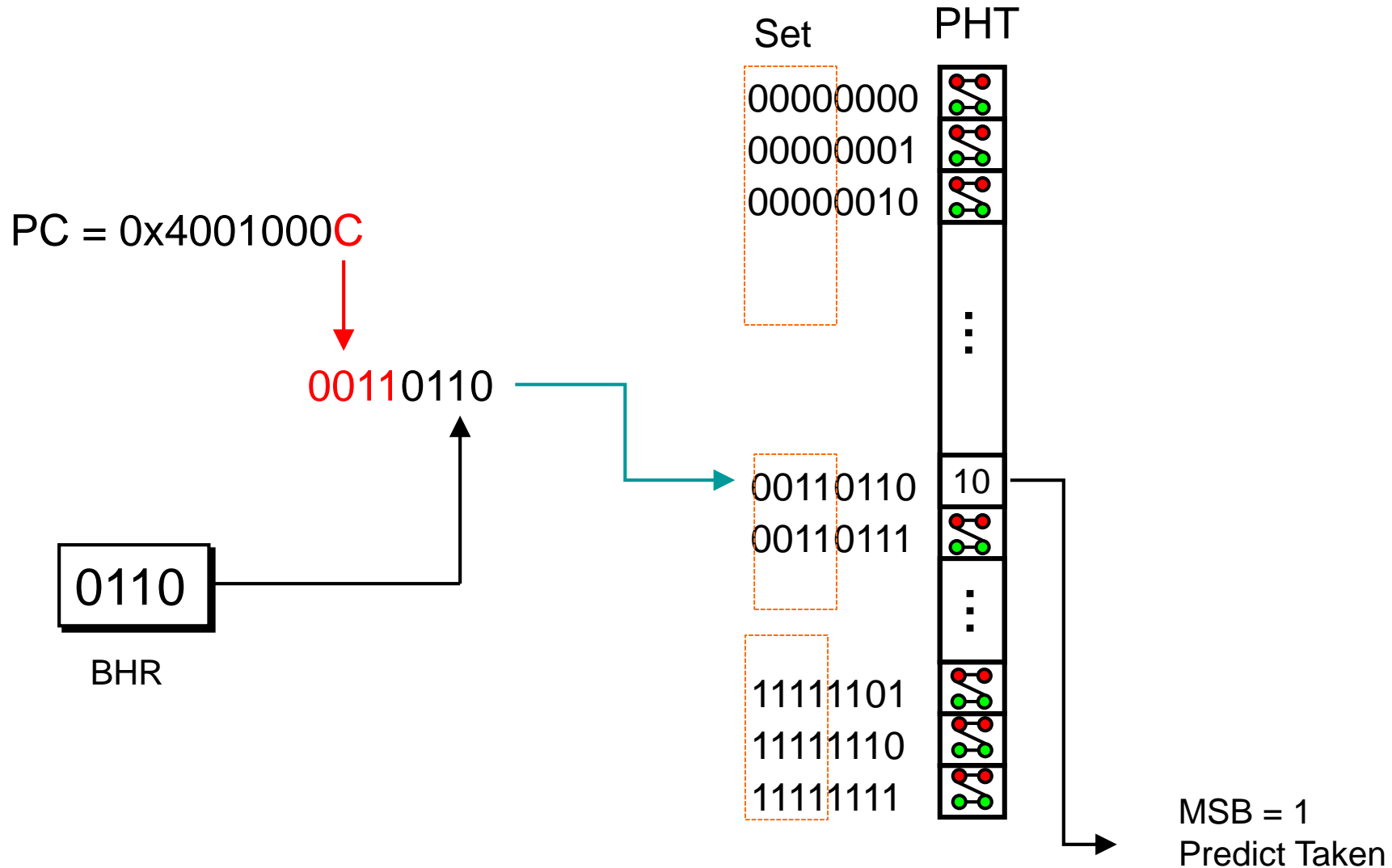
Branch History Register

- ❖ An N-bit Shift Register = 2^N patterns in PHT
- ❖ Shift-in branch outcomes
 - ✧ 1 \Rightarrow taken
 - ✧ 0 \Rightarrow not taken
- ❖ First-in First-Out
- ❖ BHR can be
 - ✧ Global
 - ✧ Per-set
 - ✧ Local (Per-address)

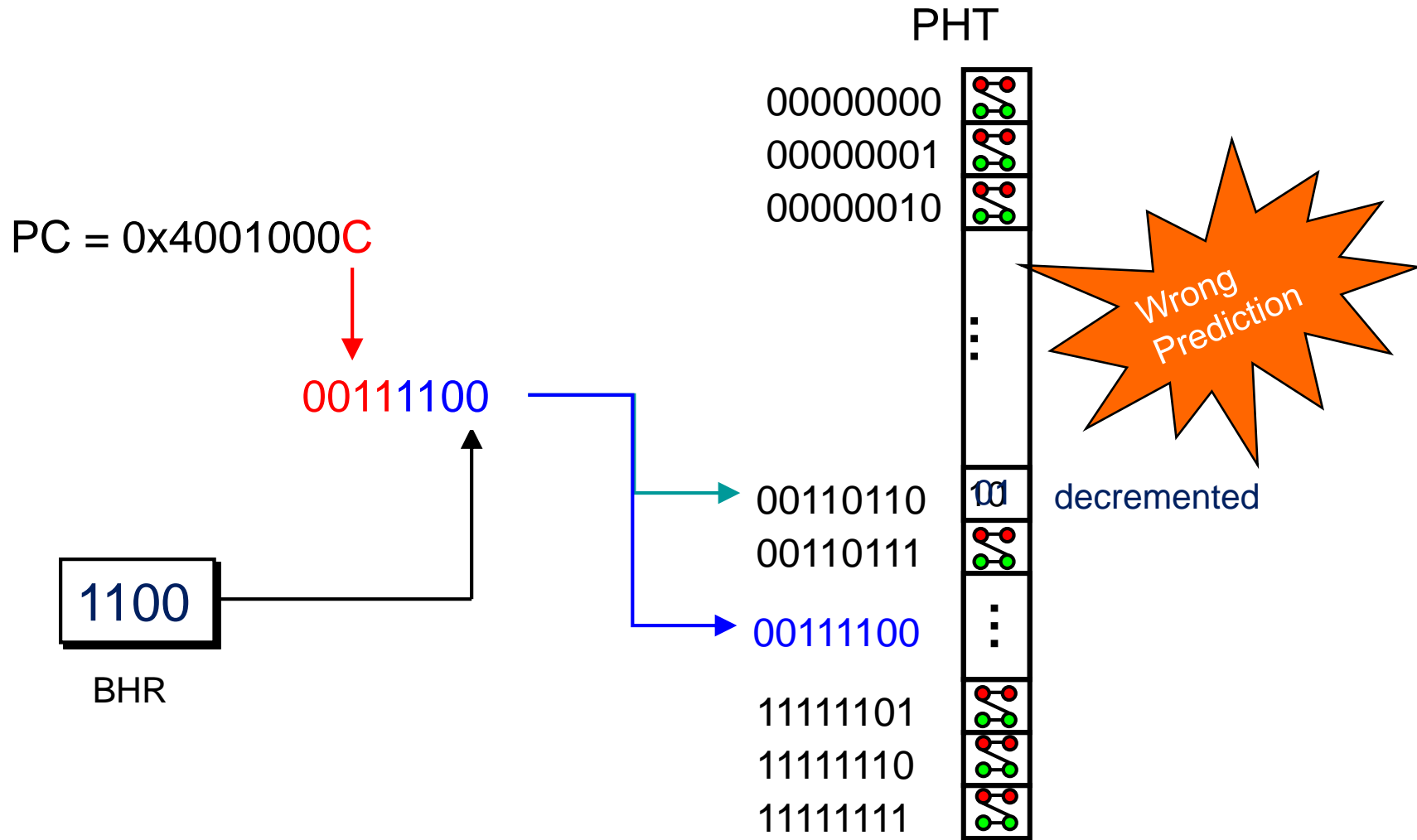
Pattern History Table

- ❖ 2^N entries addressed by N-bit BHR
- ❖ Each entry keeps a counter (2-bit or more) for prediction
 - ✧ Counter update: the same as 2-bit counter
 - ✧ Can be initialized in alternate patterns (01, 10, 01, 10, ..)
- ❖ Alias (or interference) problem

Two-Level Branch Prediction



Predictor Update (Actually, Not Taken)



- Update Predictor after branch is resolved

Tournament Predictors

- A local predictor might work well for some branches or programs, while a global predictor might work well for others
- Provide one of each and maintain another predictor to identify which predictor is best for each branch

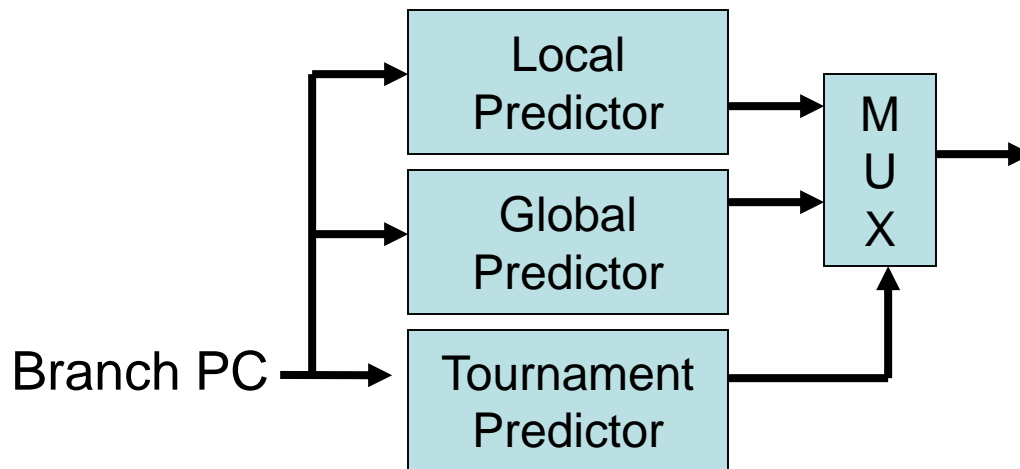
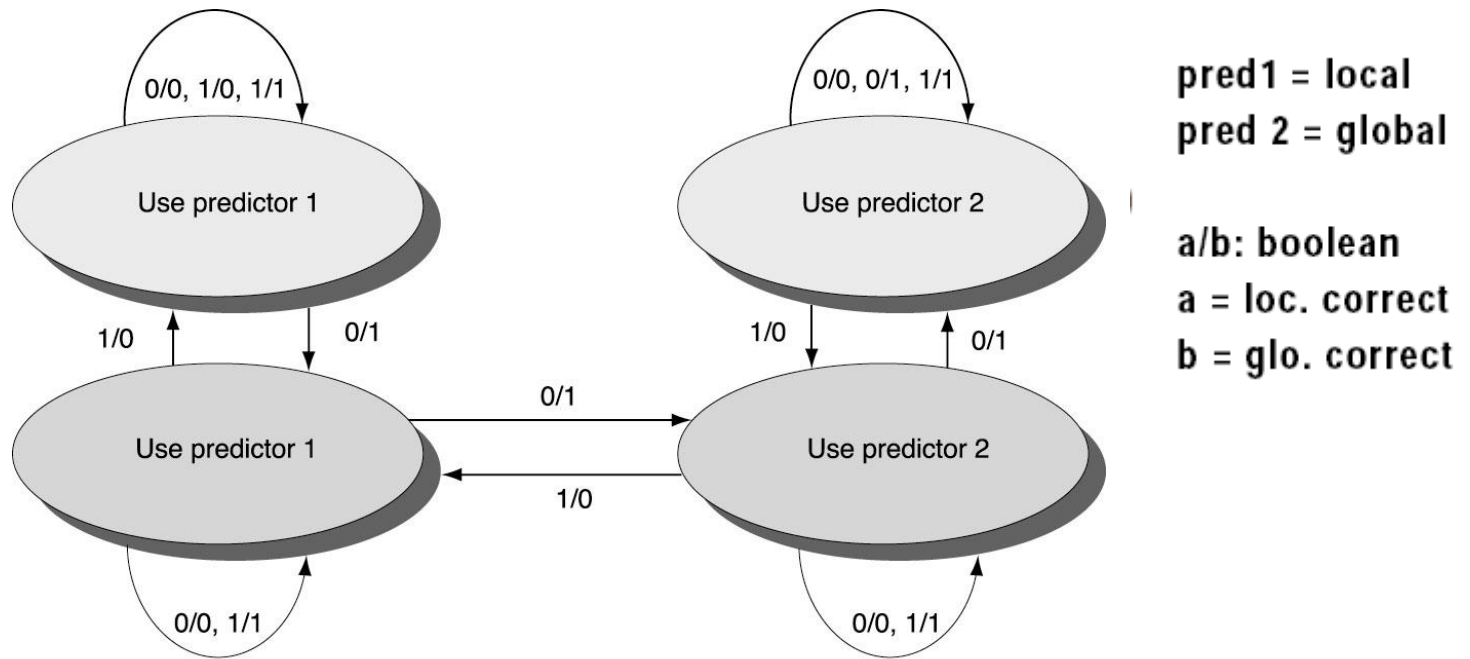


Table of 2-bit
saturating counters

Tournament Predictors

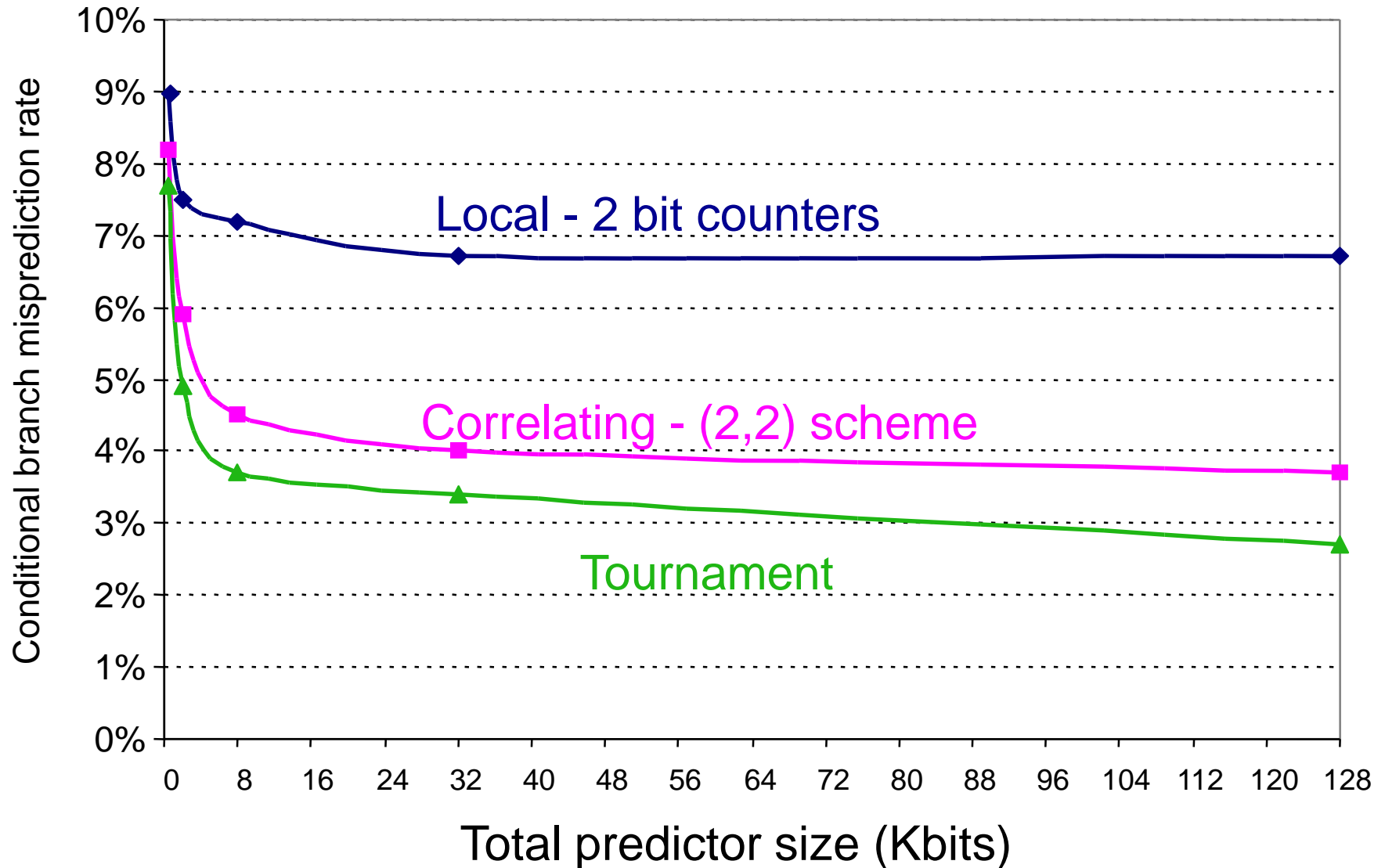
- ❖ Multilevel branch predictor
 - ✧ Selector for the Global and Local predictors of correlating branch prediction
- ❖ Use n -bit saturating counter to choose between predictors
- ❖ Usual choice between global and local predictors



Tournament Predictors

- Advantage of tournament predictor is the ability to select the right predictor for a particular branch
- A typical tournament predictor selects global predictor 40% of the time for SPEC integer benchmarks
- AMD Opteron and Phenom use tournament style

Accuracy v. Size (SPEC89)



Tournament Predictors (Intel Core i7)

- Based on predictors used in Core 2 Duo chip
- Combines three different predictors
 - Two-bit
 - Global history
 - Loop exit predictor
 - Uses a counter to predict the exact number of taken branches (number of loop iterations) for a branch that is detected as a loop branch
- Tournament: Tracks accuracy of each predictor
- Main problem of speculation:
 - A mispredicted branch may lead to another branch being mispredicted !

Branch Prediction is More Important Today

Conditional branches still comprise about 20% of instructions

Correct predictions are more important today – why?

- **pipelines deeper**
branch not resolved until more cycles from fetching - therefore the **misprediction penalty** greater
 - cycle times smaller - more emphasis on throughput (performance)
 - more functionality between fetch & execute
- **multiple instruction issue** (superscalars & VLIW)
branch occurs almost every cycle
 - flushing & refetching more instructions
- **object-oriented programming**
more indirect branches - which are harder to predict
- **dual of Amdahl's Law**
other forms of pipeline stalling are being addressed - so the portion of CPI due to branch delays is relatively larger

All this means that the potential stalling due to branches is greater

Branch Prediction is More Important Today

On the other hand,

- Chips are denser so we can consider sophisticated HW solutions
- Hardware cost is small compared to the performance gain

Directions in Branch Prediction

1: Improve the prediction

- correlated (2-level) predictor (Pentium III - 512 entries, 2-bit, Pentium Pro - 4 history bits)
- hybrid local/global predictor (Alpha 21264)
- confidence predictors

2: Determine the target earlier

- branch target buffer (Pentium Pro, IA-64 Itanium)
- next address in I-cache (Alpha 21264, UltraSPARC)
- return address stack (Alpha 21264, IA-64 Itanium, MIPS R10000, Pentium Pro, UltraSPARC-3)

3: Reduce misprediction penalty

- fetch both instruction streams (IBM mainframes, SuperSPARC)

4: Eliminate the branch

- predicated execution (IA-64 Itanium, Alpha 21264)

Predicated Execution

- ❖ Predicated instructions execute conditionally
 - ✧ Some other (previous) instruction sets a condition
 - ✧ Predicated instruction tests the condition & executes if the condition is true
 - ✧ If the condition is false, predicated instruction isn't executed
 - i.e., instruction execution is ***predicated*** on the condition
 - ✧ Eliminates conditional branch (expensive if mispredicted)
 - changes **a control hazard** to a **data hazard**
 - ✧ Fetch both true & false paths

Predicated Execution

❖ Example:

✧ Without predicated execution

add R10, R4, R5

beqz R10, Label

sub R2, R1, R6

Label: or R3, R2, R7

✧ With predicated execution

add R10, R4, R5

sub R2, R1, R6, R10

or R3, R2, R7

❖ Advantages

- No branch hazard,
- Creates straight line code
- More independent instructions

❖ Disadvantages

- Instructions on both paths are executed
- Hard to add predicated instructions to an existing instruction set;
- Additional register pressure
- Complex conditions if nested loops

Pipelining Complications

❖ **Exceptions: Events other than branches or jumps that change the normal flow of instruction execution. Some types of exceptions:**

- ❖ **I/O Device request**
- ❖ **Invoking an OS service from user program**
- ❖ **Tracing Instruction execution**
- ❖ **Breakpoint (programmer requested interrupt)**
- ❖ **Integer arithmetic overflow**
- ❖ **FP arithmetic anomaly**
- ❖ **Page fault (page not in main memory)**
- ❖ **Misaligned memory accesses**
- ❖ **Memory protection violation**
- ❖ **Use of undefined instruction**
- ❖ **Hardware malfunction**
- ❖ **Power failure**

Pipelining Complications

- ❖ Exceptions: Events other than branches or jumps that change the normal flow of instruction execution.
- ❖ 5 instructions executing in 5 stage pipeline
- ❖ How to stop the pipeline?
 - ✧ Who caused the interrupt?
 - ✧ How to restart the pipeline?

<u>Stage</u>	<u>Problems causing the interrupts</u>
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

Pipelining Complications

- ❖ Simultaneous exceptions in more than one pipeline stage, e.g.,
 - ✧ LOAD with data page fault in MEM stage
 - ✧ ADD with instruction page fault in IF stage
- ❖ Solution #1
 - ✧ Interrupt status vector per instruction
 - ✧ Defer check until last stage, kill state update if exception
- ❖ Solution #2
 - ✧ Interrupt ASAP
 - ✧ Restart everything that is incomplete

Pipelining Complications

- ❖ Our DLX pipeline only writes results at the end of the instruction's execution. Not all processors do this.
- ❖ Address modes: Auto-increment causes register change during instruction execution
 - ✧ Interrupts Need to restore register state
 - ✧ Adds WAR and WAW hazards since writes happen not only in last stage
- ❖ Memory-Memory Move Instructions
 - ✧ Must be able to handle multiple page faults
 - ✧ VAX and x86 store values temporarily in registers
- ❖ Condition Codes
 - ✧ Need to detect the last instruction to change condition codes

Fallacies and Pitfalls

❖ Pipelining is easy!

- ✧ The basic idea is easy
- ✧ The devil is in the details
 - Detecting data hazards and stalling pipeline

❖ Poor ISA design can make pipelining harder

- ✧ Complex instruction sets (Intel IA-32)
 - Significant overhead to make pipelining work
 - IA-32 micro-op approach
- ✧ Complex addressing modes
 - Register update side effects, memory indirection

Pipeline Hazards Summary

❖ Three types of pipeline hazards

- ✧ Structural hazards: conflicts using a resource during same cycle
- ✧ Data hazards: due to data dependencies between instructions
- ✧ Control hazards: due to branch and jump instructions

❖ Hazards limit the performance and complicate the design

- ✧ Structural hazards: eliminated by careful design or more hardware
- ✧ Data hazards are eliminated by data forwarding
- ✧ However, load delay cannot be completely eliminated
- ✧ Delayed branching can be a solution for control hazards
- ✧ BTB with branch prediction can reduce branch delay to zero
- ✧ Branch misprediction should flush the wrongly fetched instructions