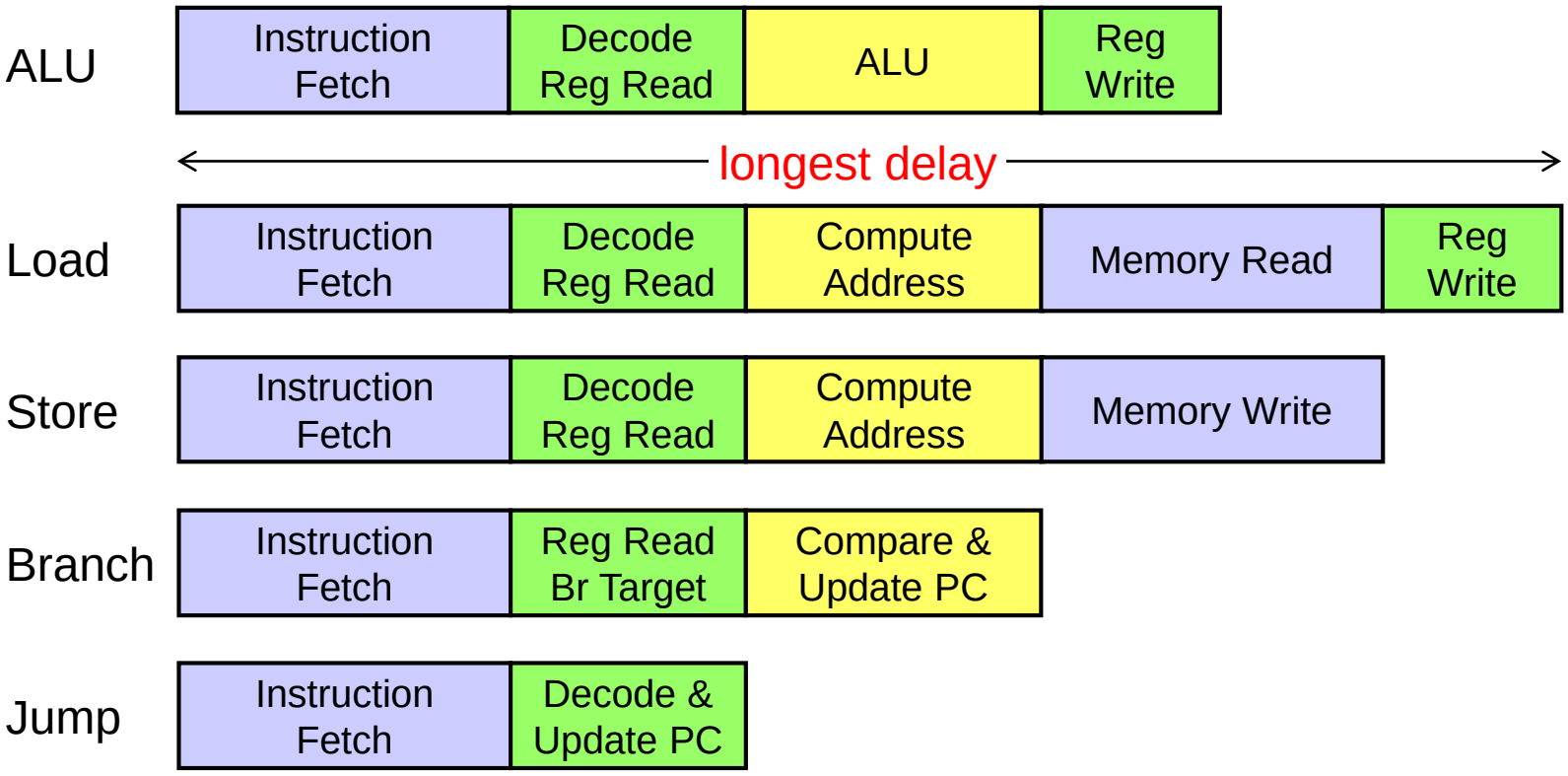
Pipelined Processor and Hazards

Agenda

- 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

Drawback of Single Cycle Processor

- Drawback is the Long cycle time
  - All instructions take as much time as the slowest instruction



Alternative: Multicycle Implementation

- Break instruction execution into five steps
  - Instruction fetch
  - Instruction decode, register read, target address for jump/branch
  - Execution, memory address calculation, or branch outcome
  - Memory access or ALU instruction completion
  - Load instruction completion
- One clock cycle per step (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

Performance Example

- Assume the following operation times for components:
  - Access time for Instruction and data memories: 200 ps
  - Delay in ALU and adders: 180 ps
  - Delay in Decode and Register file access (read or write): 150 ps
  - Ignore the other 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

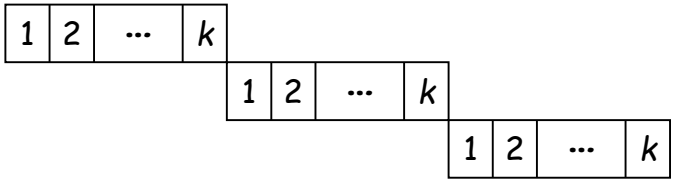
Solution

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 ← Compare and update PC			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 ps / (3.8 × 200 ps) = 880 / 760 = 1.16

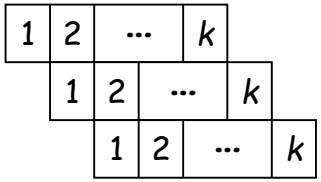
Serial Execution versus Pipelining

- 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

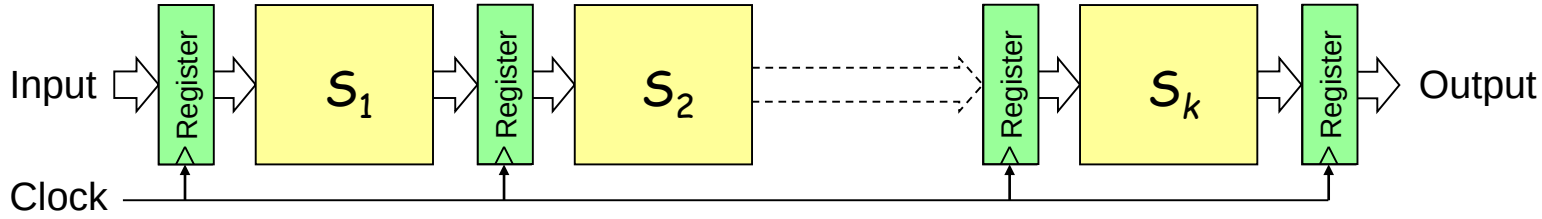


With Pipelining

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 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

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

MIPS Processor Pipeline

- Five stages, one cycle per stage

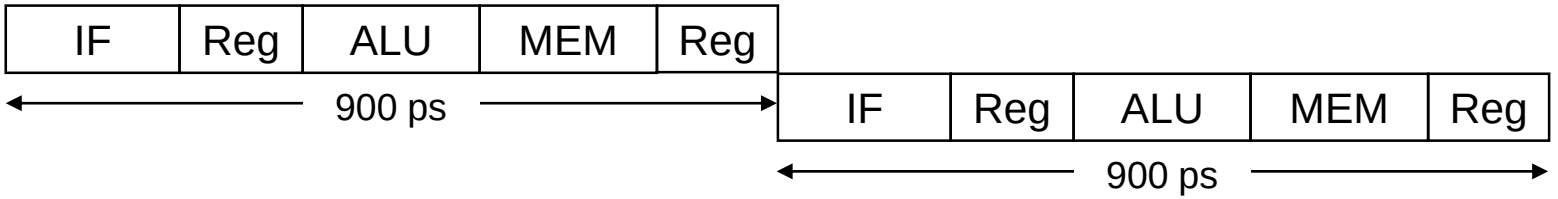
1. IF: Instruction Fetch from instruction memory
2. ID: Instruction Decode, register read, and J/Br address
3. EX: Execute operation or calculate load/store address
4. MEM: Memory access for load and store
5. WB: Write Back result to register

Single-Cycle vs Pipelined Performance

- Consider a 5-stage instruction execution in which …
  - Instruction fetch = ALU operation = Data memory access = 200 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?

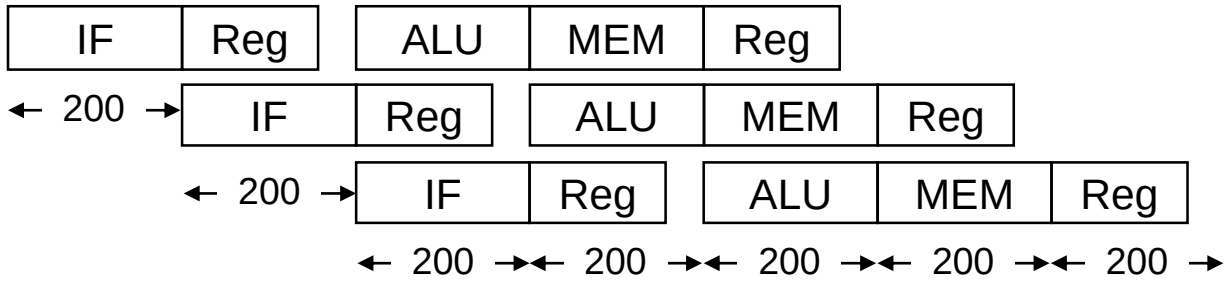
- Solution

  Single-Cycle Clock = 200+150+200+200+150 = 900 ps



Single-Cycle versus Pipelined – cont'd

- Pipelined clock cycle = max(200, 150) = 200 ps



- CPI for pipelined execution = 1
  - One instruction completes each cycle (ignoring pipeline fill)
- Speedup of pipelined execution = 900 ps / 200 ps = 4.5
  - 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

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

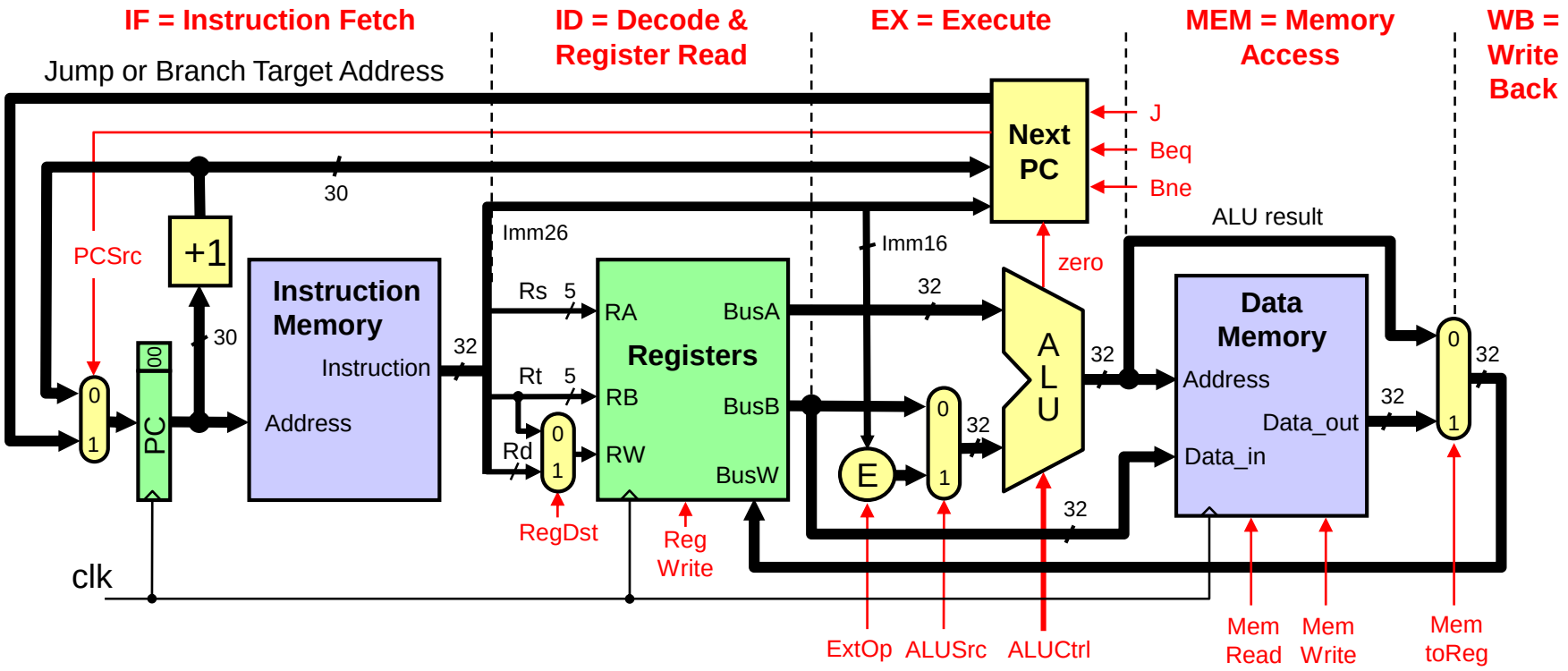
Next . . .

- 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

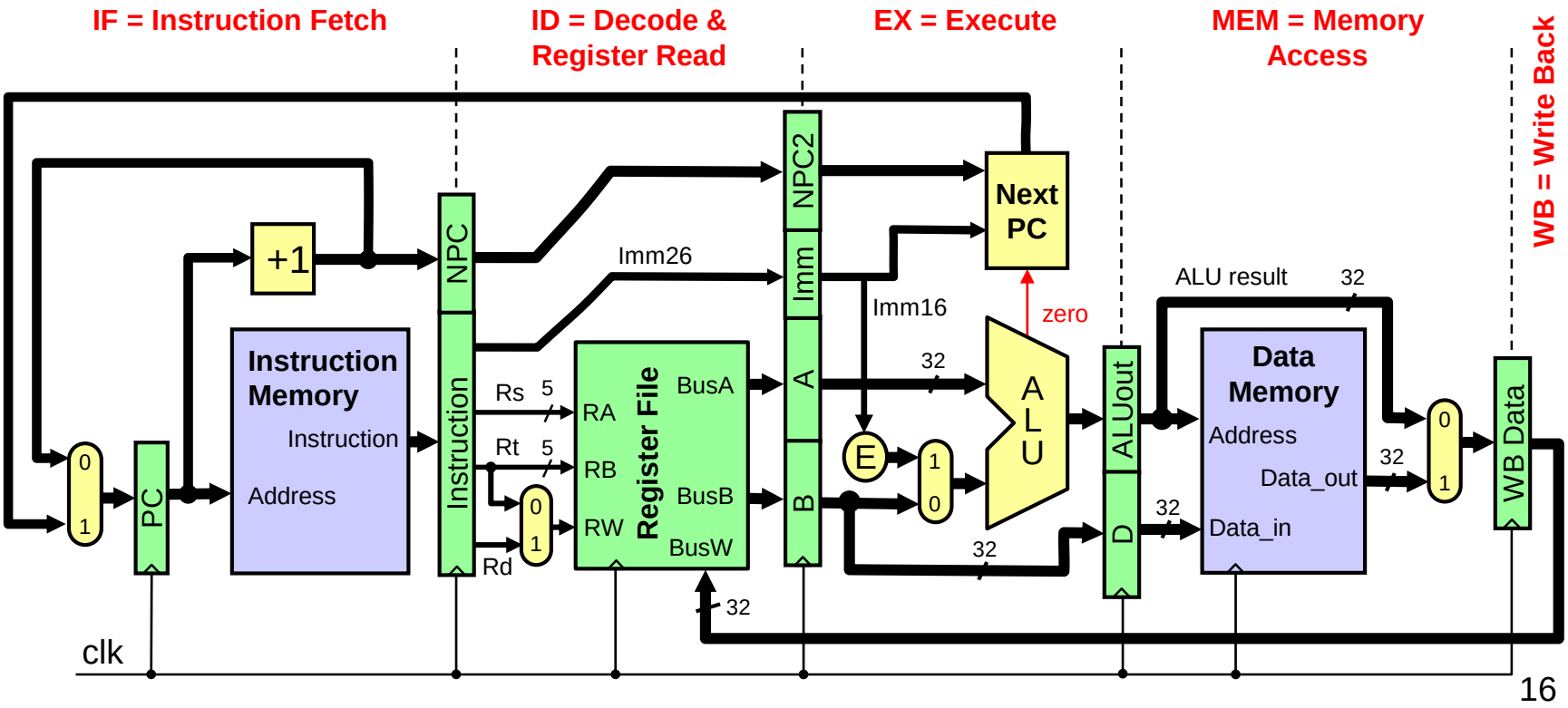
- How to pipeline this single-cycle datapath?

  Answer: Introduce pipeline register at end of each stage



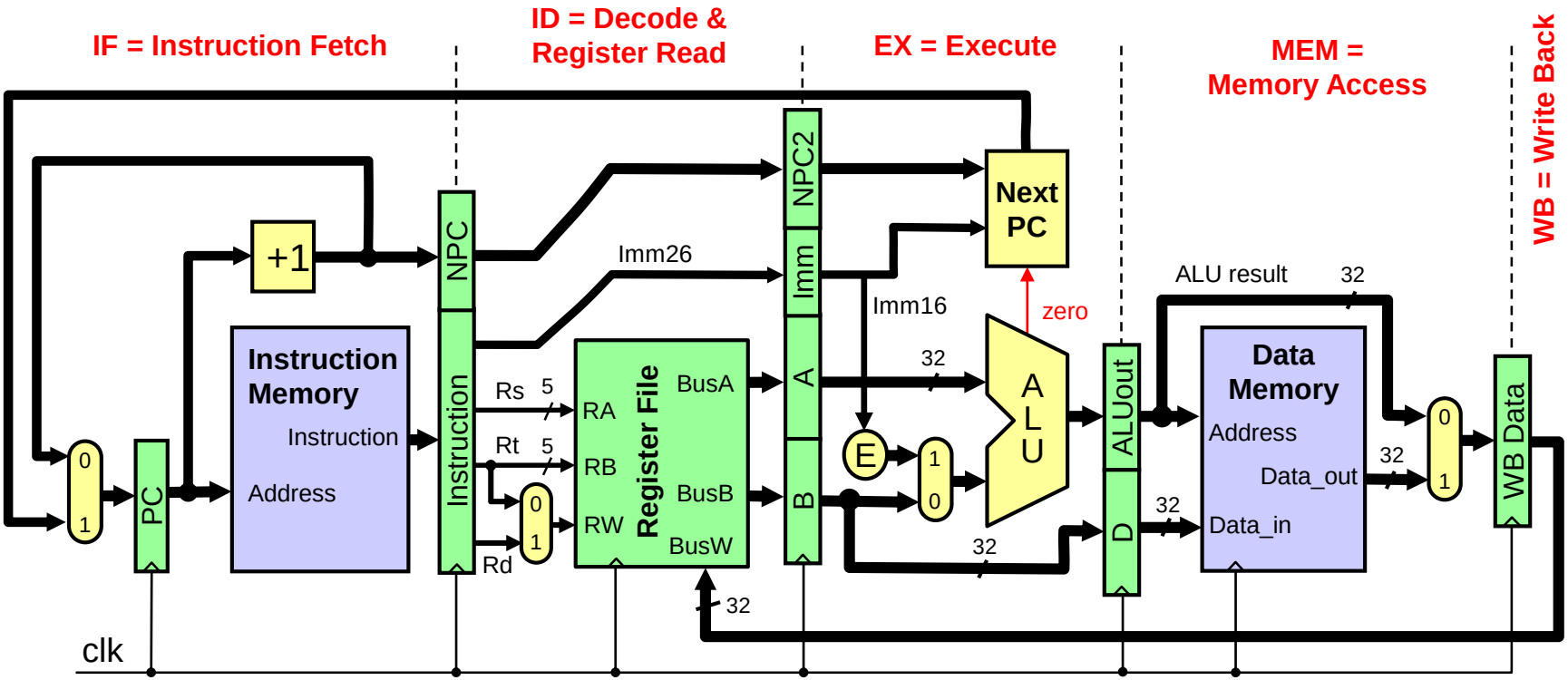
Pipelined Datapath

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



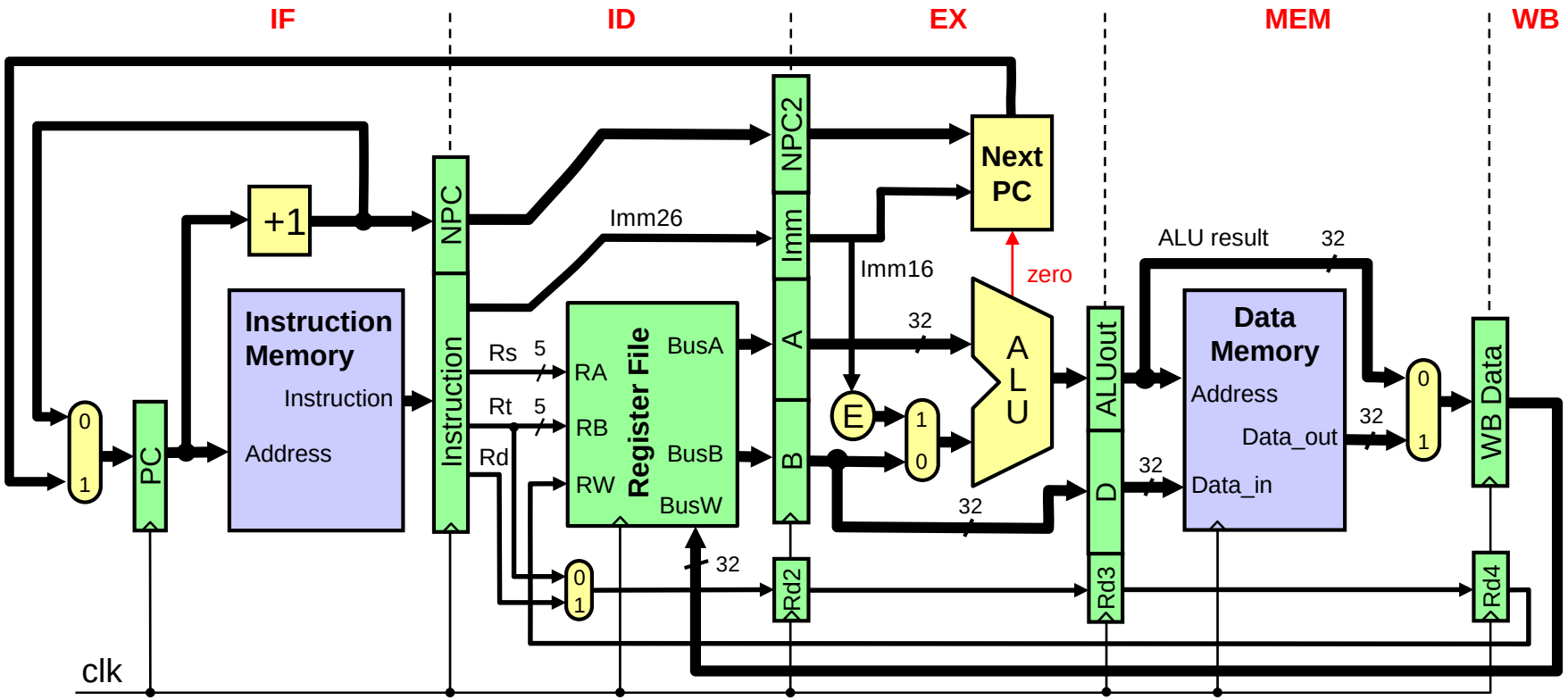
Problem with Register Destination

- 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



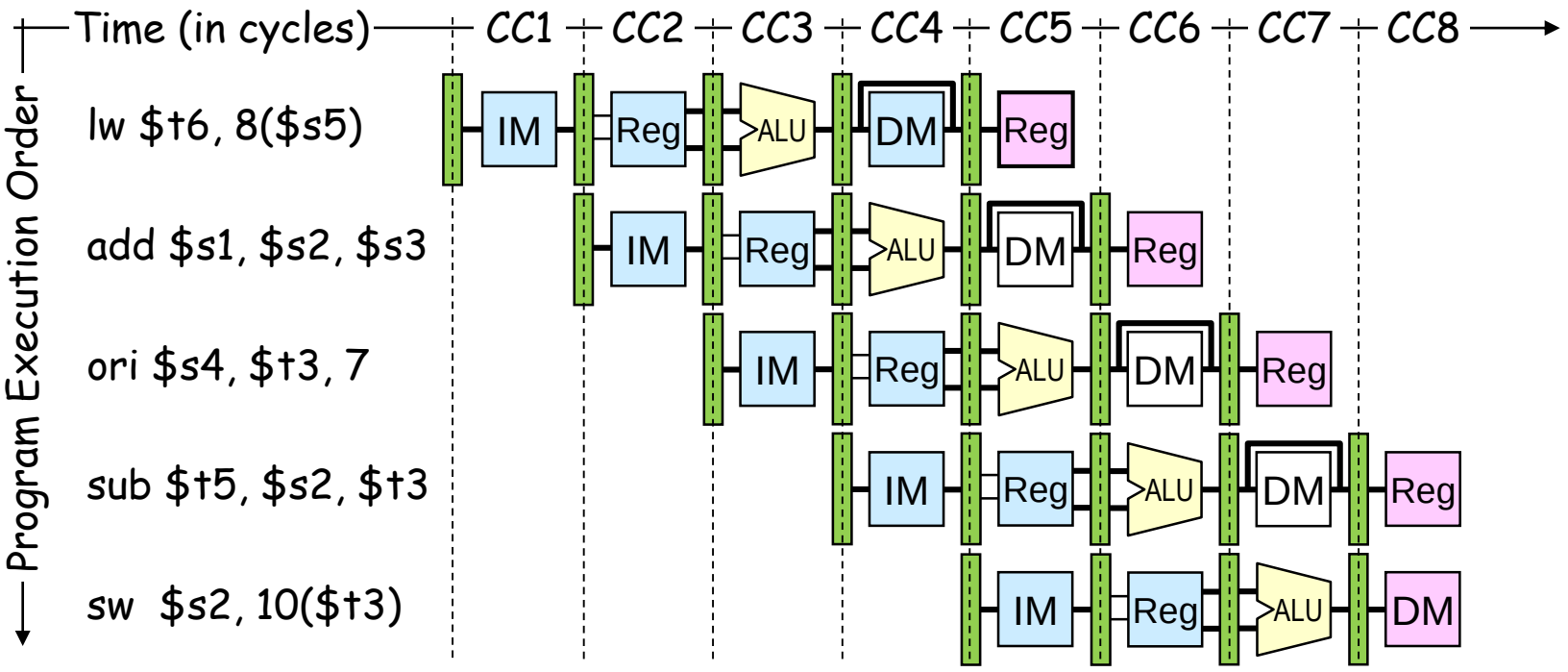
Pipelining the Destination Register

- 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



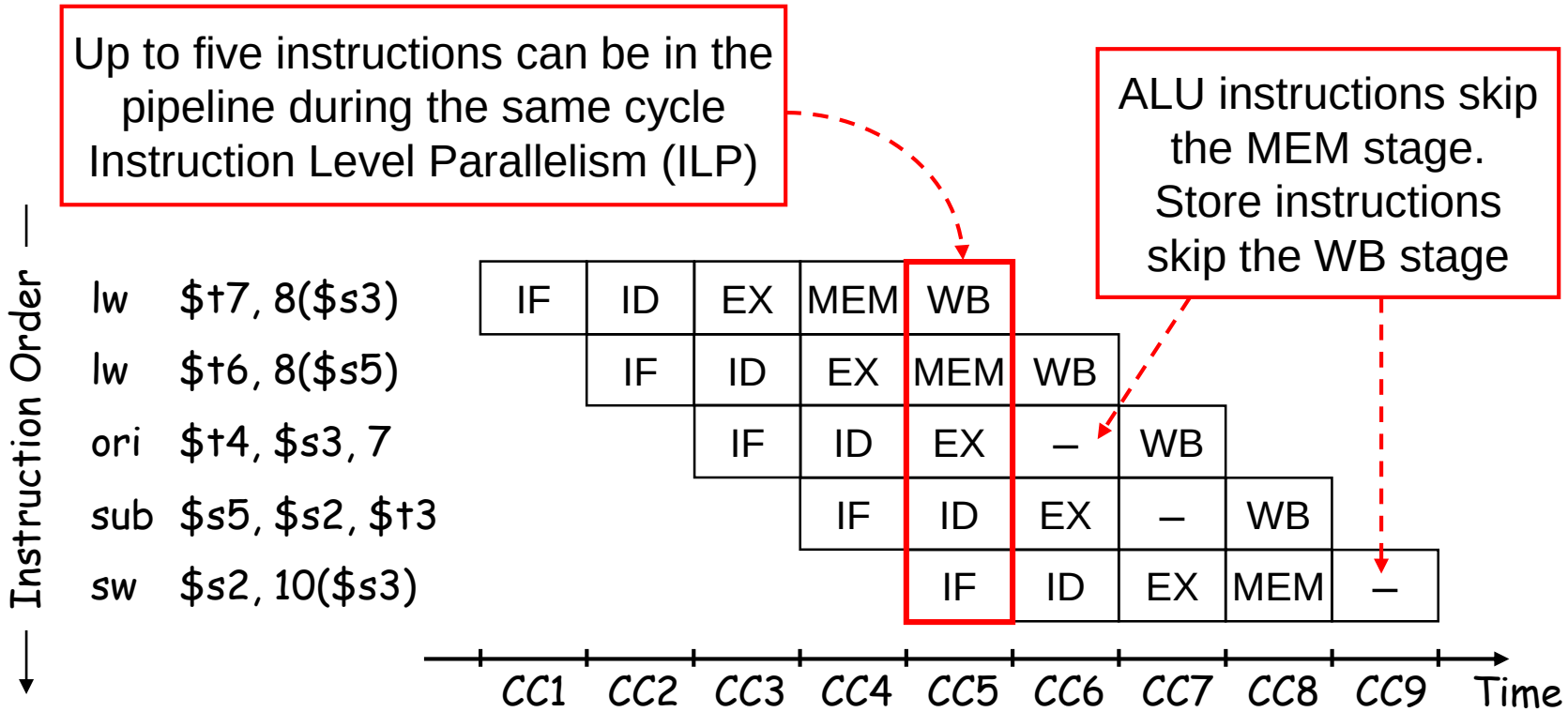
Graphically Representing Pipelines

- 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

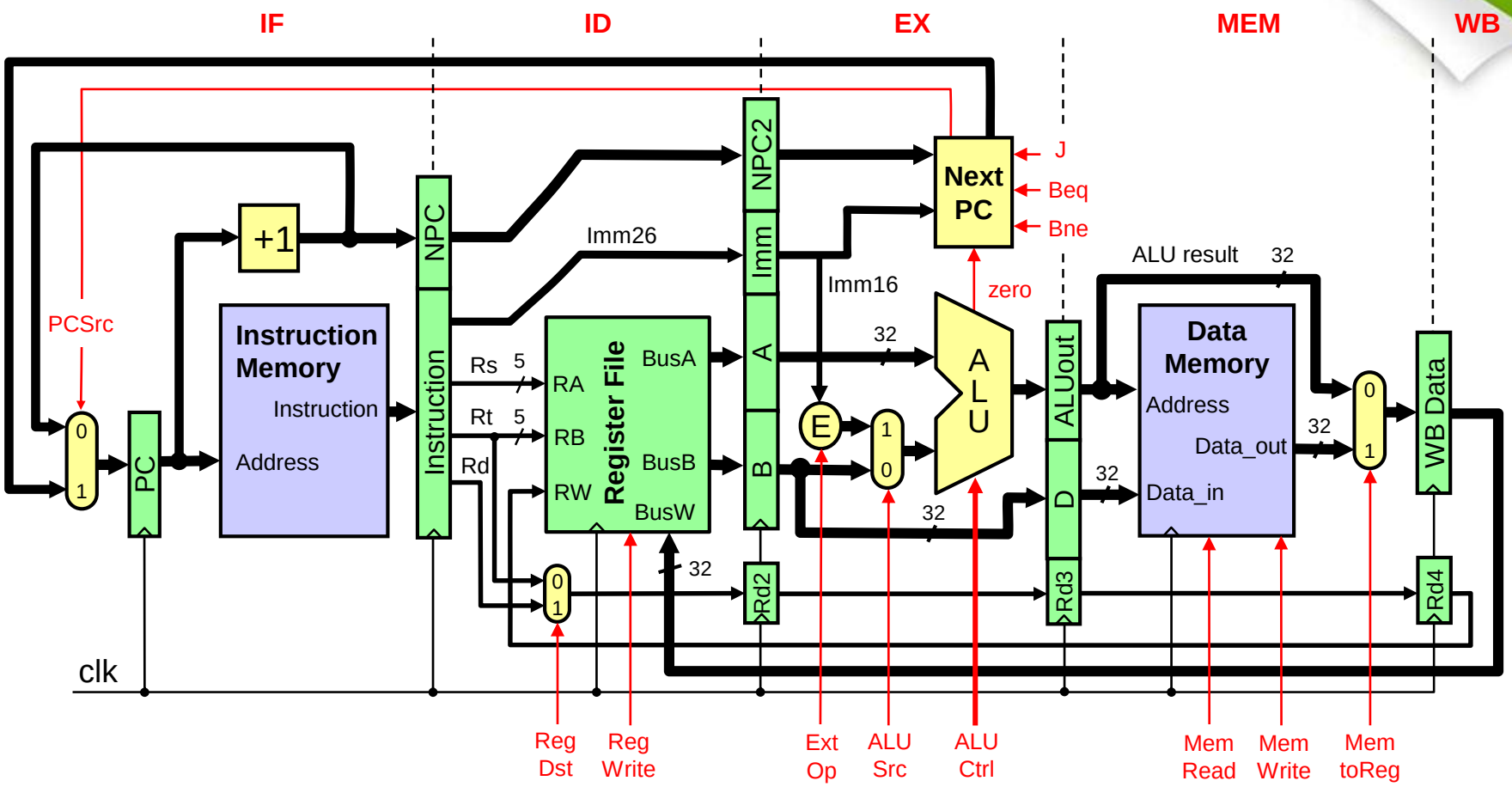


Instruction-Time Diagram

- Instruction-Time Diagram shows:
  - Which instruction occupying what stage at each clock cycle
- Instruction flow is pipelined over the 5 stages

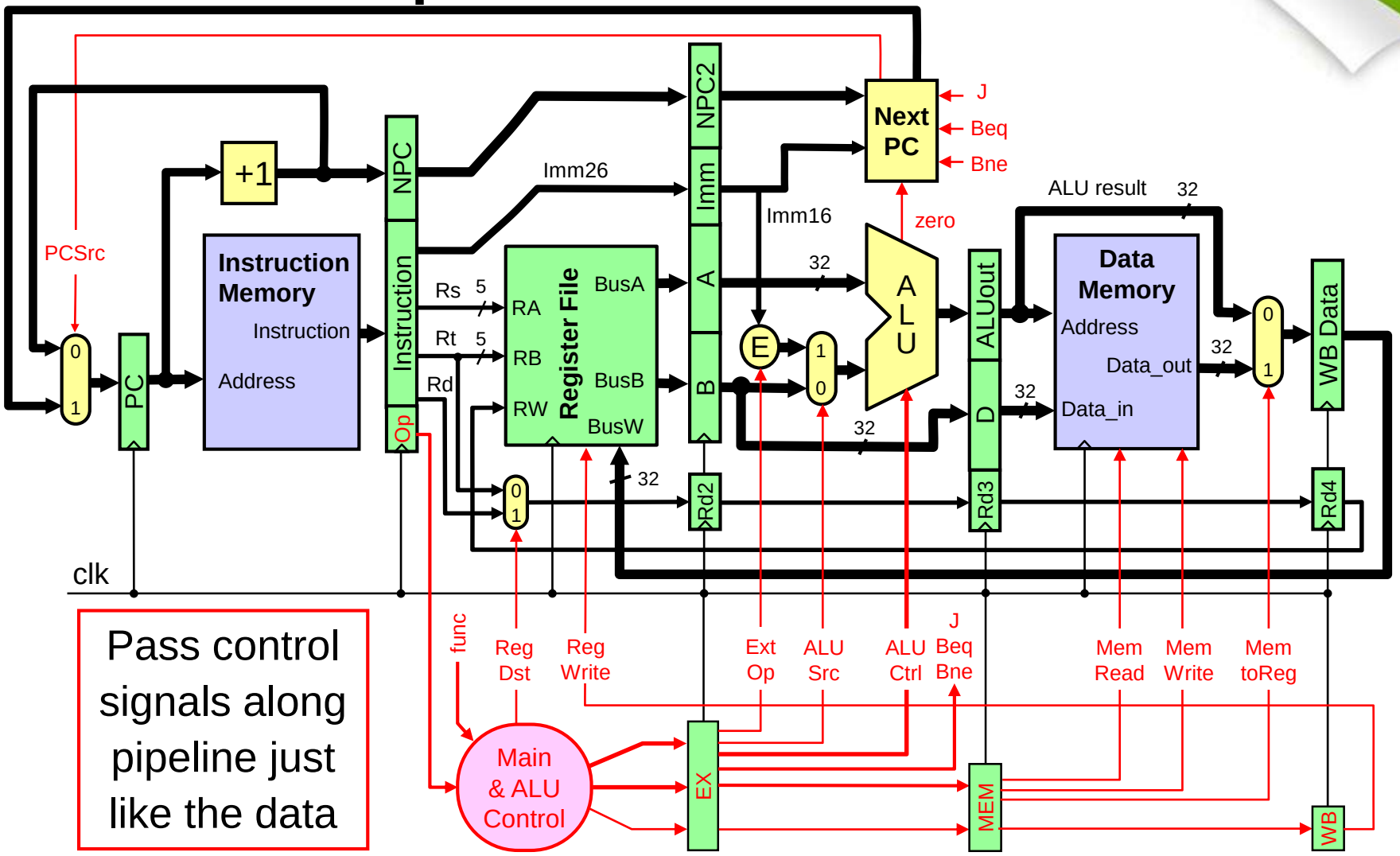


Control Signals



Same control signals used in the single-cycle datapath

Pipelined Control



Pass control signals along pipeline just like the data

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		Memory Stage			Write Back	PC Control
	RegDst	ExtOp	ALUSrc	ALUOp	MemRd	MemWr	WBdata	RegWr	PCSrc
R-Type	1=Rd	X	0=Reg	**func**	0	0	0	1	0 = next PC
ADDI	0=Rt	1=sign	1=Imm	ADD	0	0	0	1	0 = next PC
SLTI	0=Rt	1=sign	1=Imm	SLT	0	0	0	1	0 = next PC
ANDI	0=Rt	0=zero	1=Imm	AND	0	0	0	1	0 = next PC
ORI	0=Rt	0=zero	1=Imm	OR	0	0	0	1	0 = next PC
LW	0=Rt	1=sign	1=Imm	ADD	1	0	1	1	0 = next PC
SW	X	1=sign	1=Imm	ADD	0	1	X	0	0 = next PC
BEQ	X	X	0=Reg	SUB	0	0	X	0	0 or 2 = BTA
BNE	X	X	0=Reg	SUB	0	0	X	0	0 or 2 = BTA
J	X	X	X	X	0	0	X	0	1 = jump target

PCSrc = 0 or 2 (BTA) for BEQ and BNE, depending on the zero flag

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

Structure Hazards

- Problem
  - Attempt to use the same hardware resource by two different instructions during the same clock cycle
- Example
  - In MIPS pipeline with a single memory
    - Load/store requires data access
    - Instruction fetch would have to *stall* for that cycle
      - Would cause a pipeline “bubble”

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

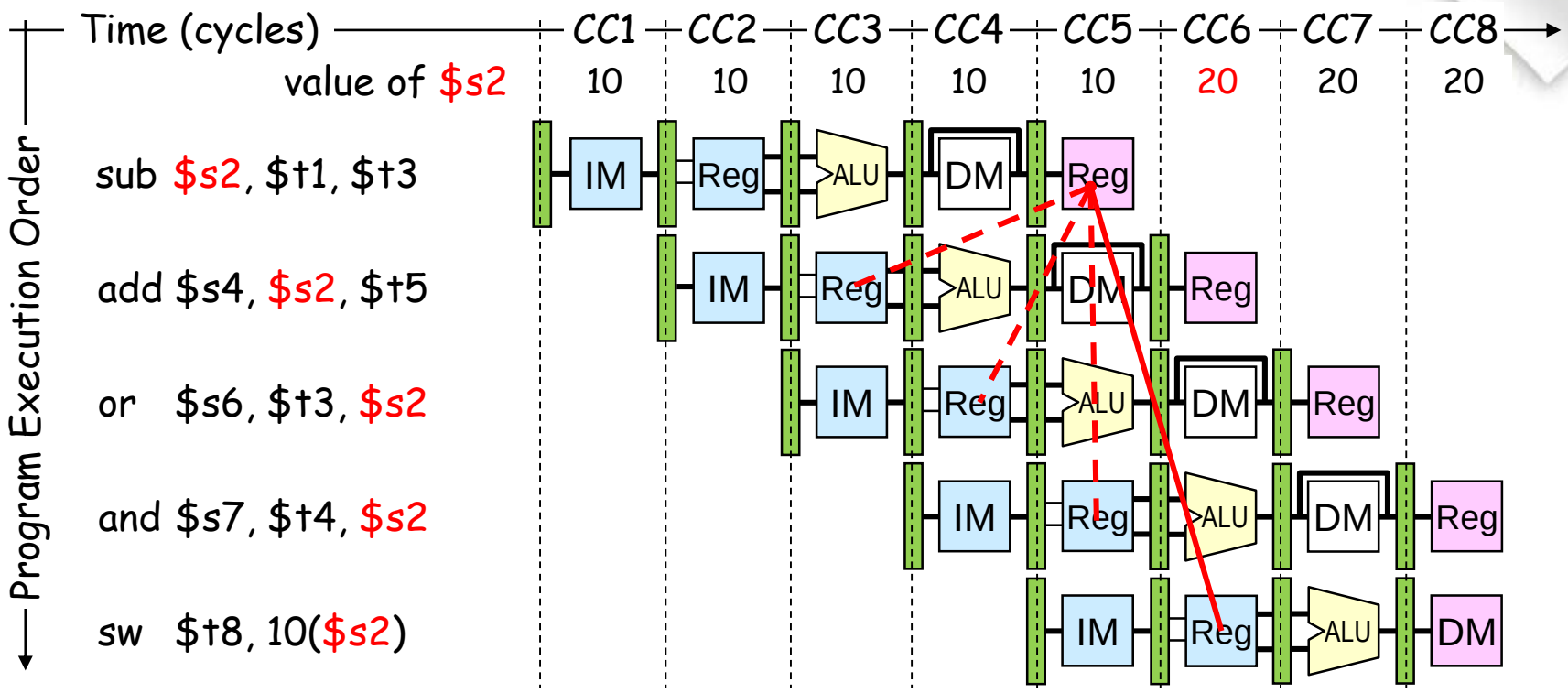
Data Hazards

- 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 $s1, $s2, $s3      # $s1 is written
J: sub $s4, $s1, $s3      # $s1 is read
```

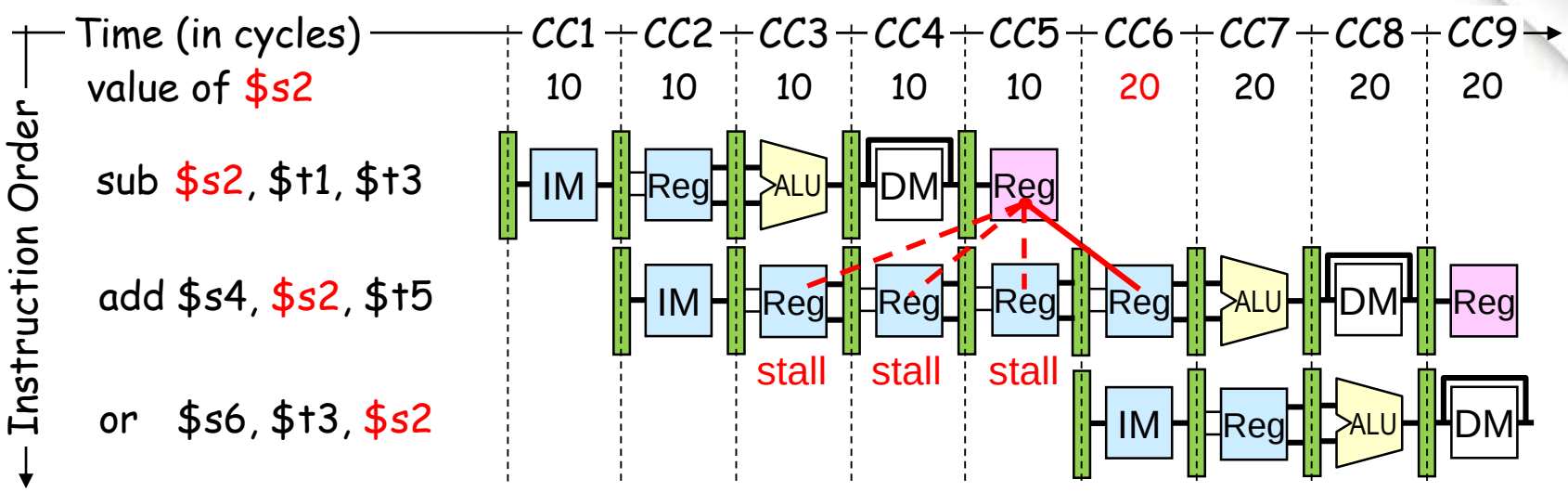
  - Hazard occurs when *J* reads the operand before *I* writes it

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 $s2 from reg file
- During CC5, $s2 is written at end of cycle, old value is read

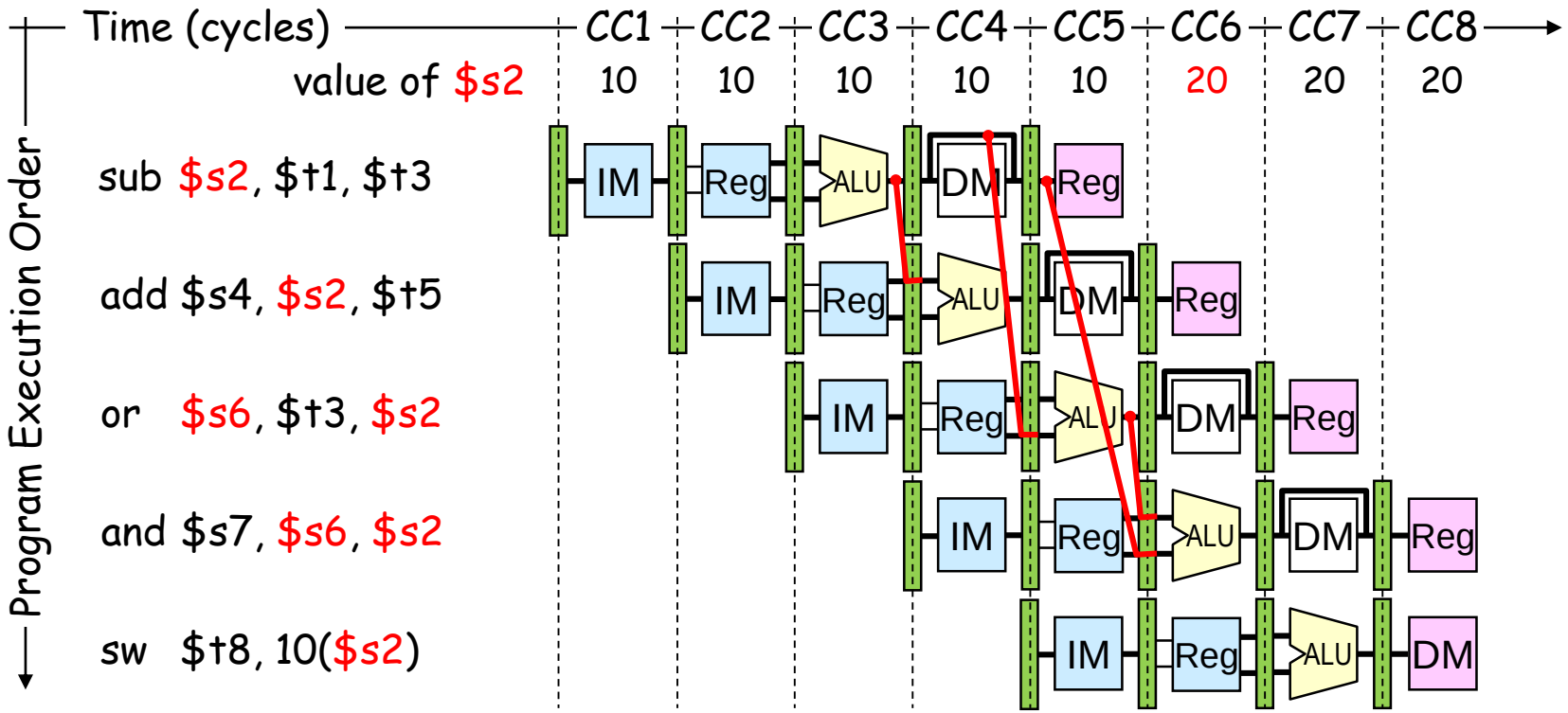
Solution 1: Stalling the Pipeline



- 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 $s2 until beginning of CC6
  - The add instruction remains in the Instruction register until CC6
  - The PC register is not modified until beginning of CC6

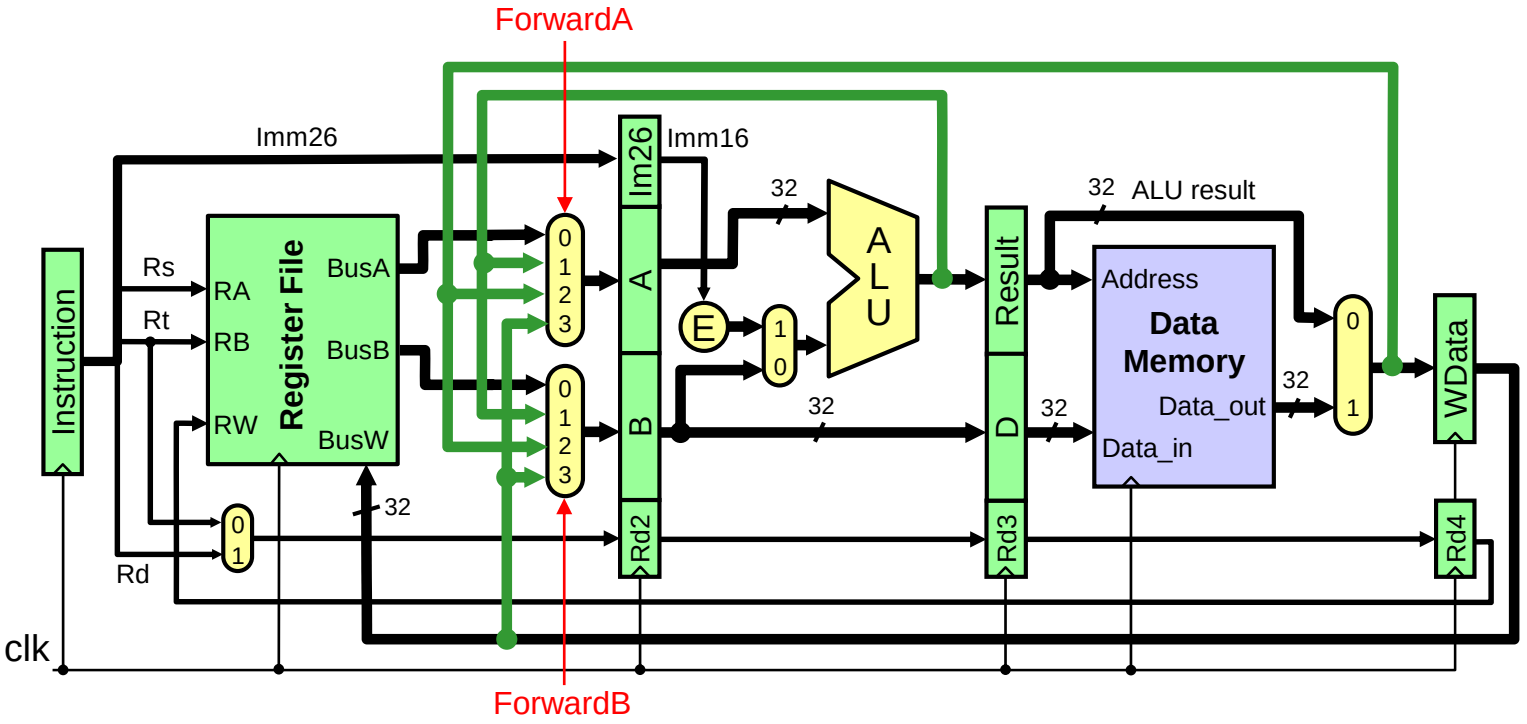
Solution 2: Forwarding ALU Result

- 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 2nd previous instruction to A (from MEM stage)
ForwardA = 3	Forward result of 3rd 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 2nd previous instruction to B (from MEM stage)
ForwardB = 3	Forward result of 3rd previous instruction to B (from WB stage)

Forwarding Example

Instruction sequence:

```
lw   $t4, 4($t0)
ori  $t7, $t1, 2
sub  $t3, $t4, $t7
```

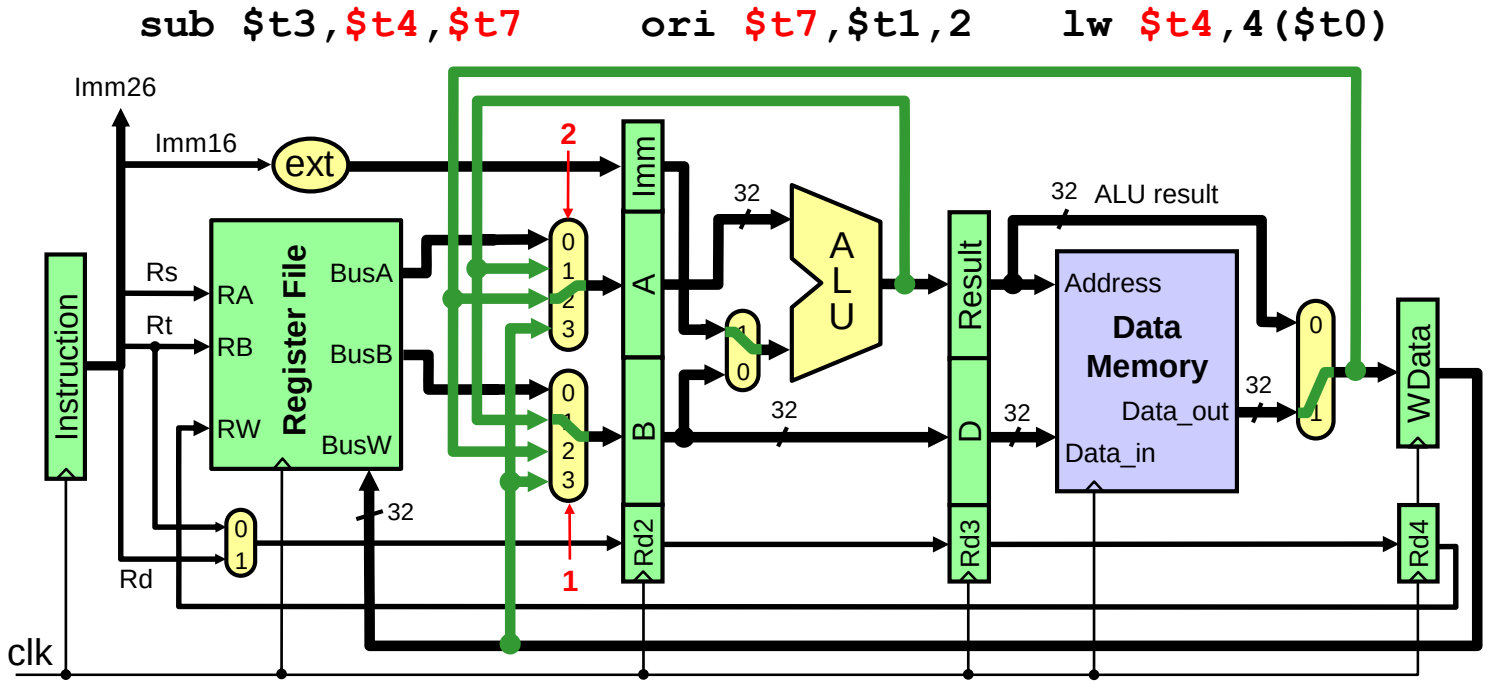
When **sub** instruction is fetched

ori will be in the ALU stage

lw will be in the MEM stage

ForwardA = 2 from MEM stage

ForwardB = 1 from ALU stage



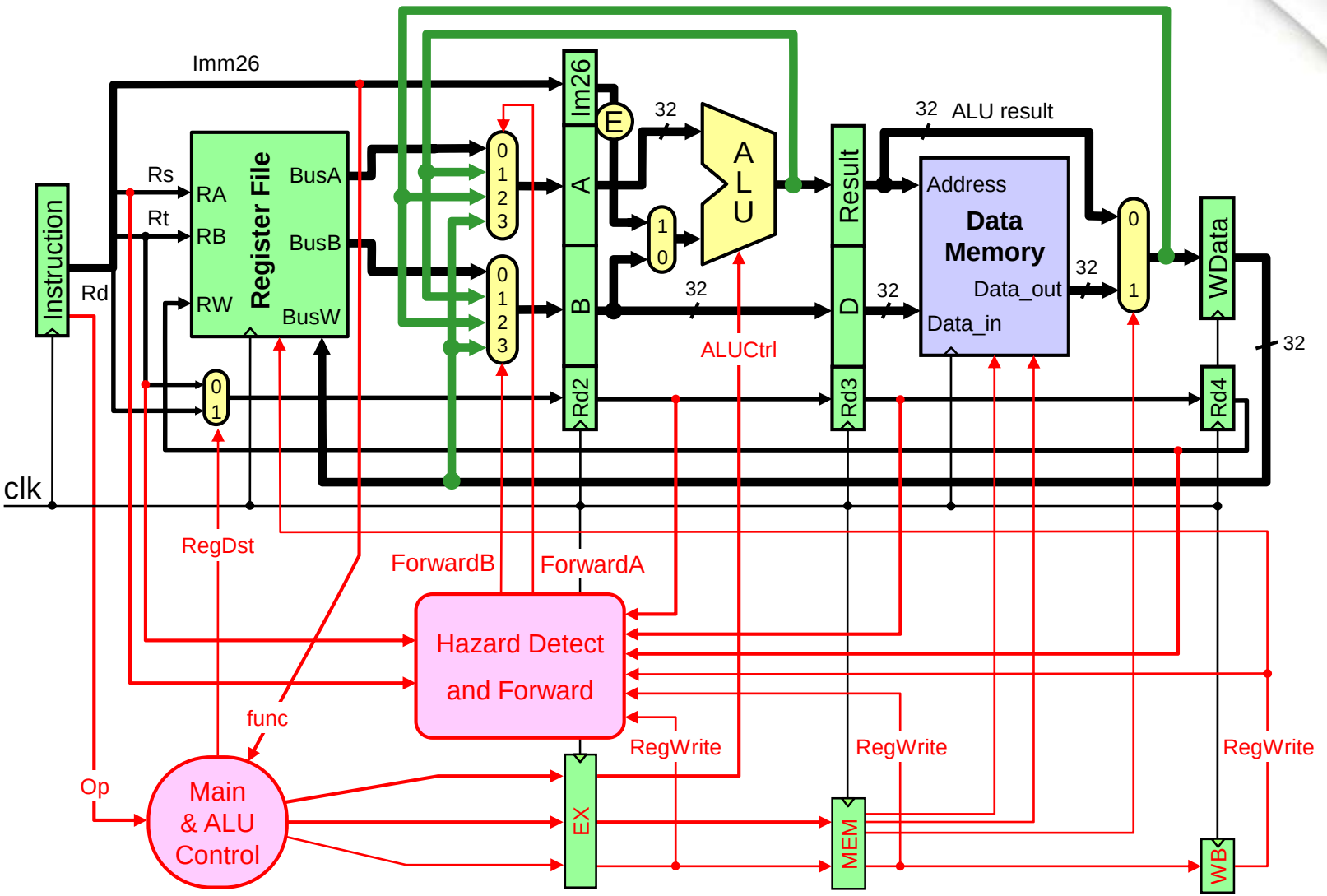
RAW Hazard Detection

- 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

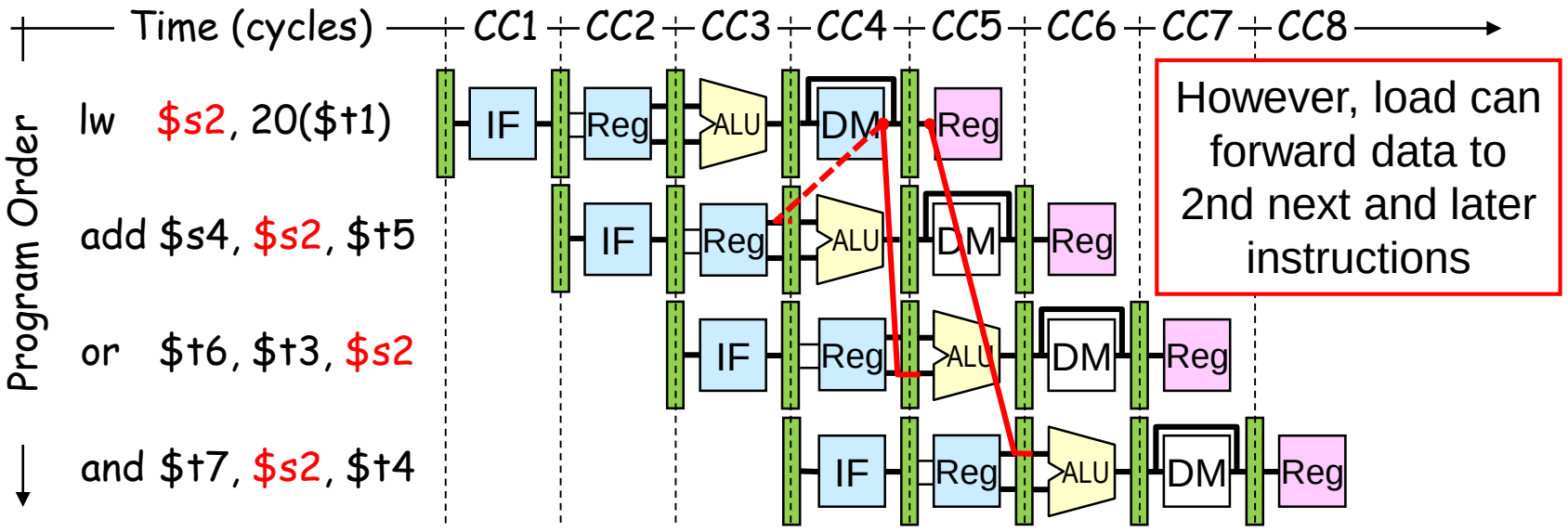


Next . . .

- 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

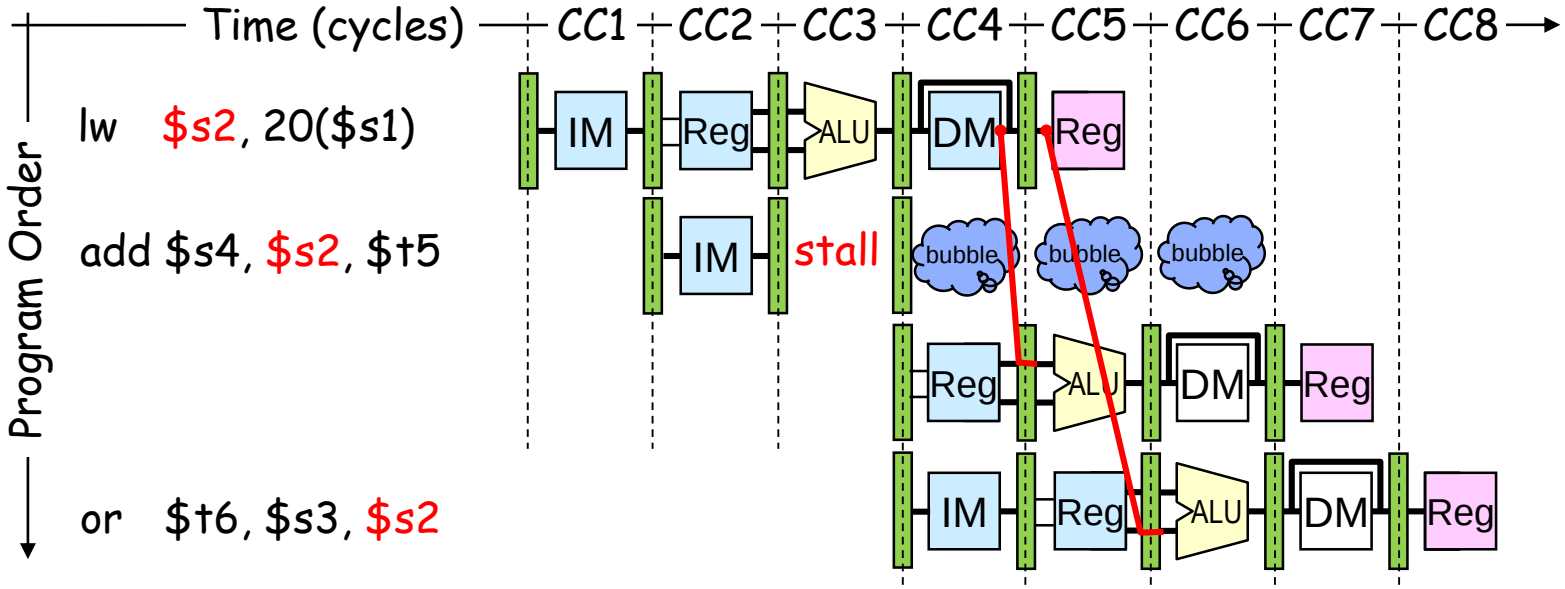
- Detecting a RAW hazard after a Load instruction:
  - The load instruction will be in the EX stage
  - Instruction that depends on the load data is in the decode stage
- Condition for stalling the pipeline

```
if((EX.MemRd == 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

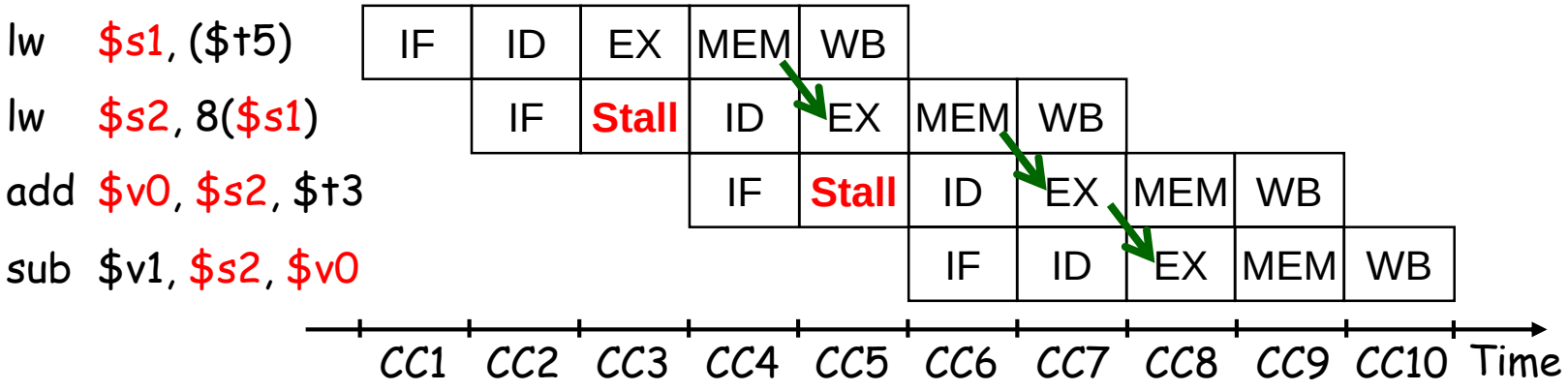
- ADD instruction depends on LW ➔ stall at CC3
  - Allow Load instruction in ALU stage to proceed
  - Freeze PC and Instruction registers (NO instruction is fetched)
  - Introduce a bubble into the ALU stage (bubble is a NO-OP)
- Load can forward data to next instruction after delaying it



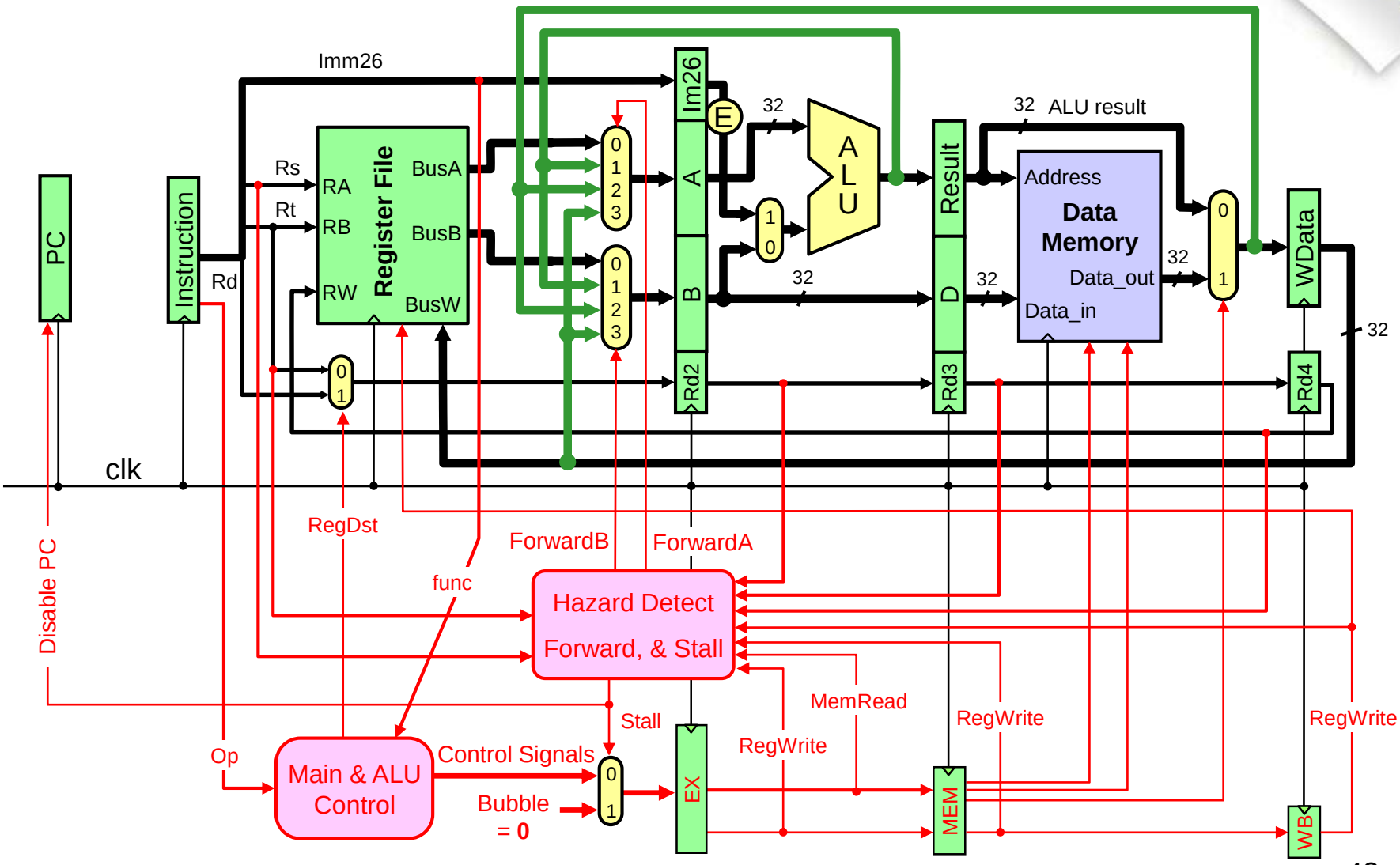
Showing Stall Cycles

- 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 **green 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   $t0,4($s0)     # &B = 4($s0)
lw   $t1,8($s0)     # &C = 8($s0)
add  $t2,$t0,$t1    # stall cycle
sw   $t2,0($s0)     # &A = 0($s0)
lw   $t3,16($s0)    # &E = 16($s0)
lw   $t4,20($s0)    # &F = 20($s0)
sub  $t5,$t3,$t4    # stall cycle
sw   $t5,12($0)     # &D = 12($0)
```

❖ Fast code: No Stalls

```
lw   $t0, 4($s0)
lw   $t1, 8($s0)
lw   $t3, 16($s0)
lw   $t4, 20($s0)
add  $t2, $t0, $t1
sw   $t2, 0($s0)
sub  $t5, $t3, $t4
sw   $t5, 12($s0)
```

Name Dependence: Write After Read

- Instruction **J** should write its result after it is read by **I**
- Called anti-dependence by compiler writers

```
I: sub $t4, $t1, $t3 # $t1 is read
J: add $t1, $t2, $t3 # $t1 is written
```

- Results from reuse of the name **$t1**
- NOT a data hazard in the 5-stage pipeline because:
  - Reads are always in stage 2
  - Writes are always in stage 5, and
  - Instructions are processed in order
- Anti-dependence can be eliminated by **renaming**
  - Use a different destination register for **add** (eg, **$t5**)

Name Dependence: Write After Write

- Same destination register is written by two instructions
- Called output-dependence in compiler terminology

```
I: sub $t1, $t4, $t3     # $t1 is written
J: add $t1, $t2, $t3     # $t1 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 **$t1** before instruction **I**
- Output dependence can be eliminated by **renaming $t1**
- Read After Read is NOT a name dependence

Next . . .

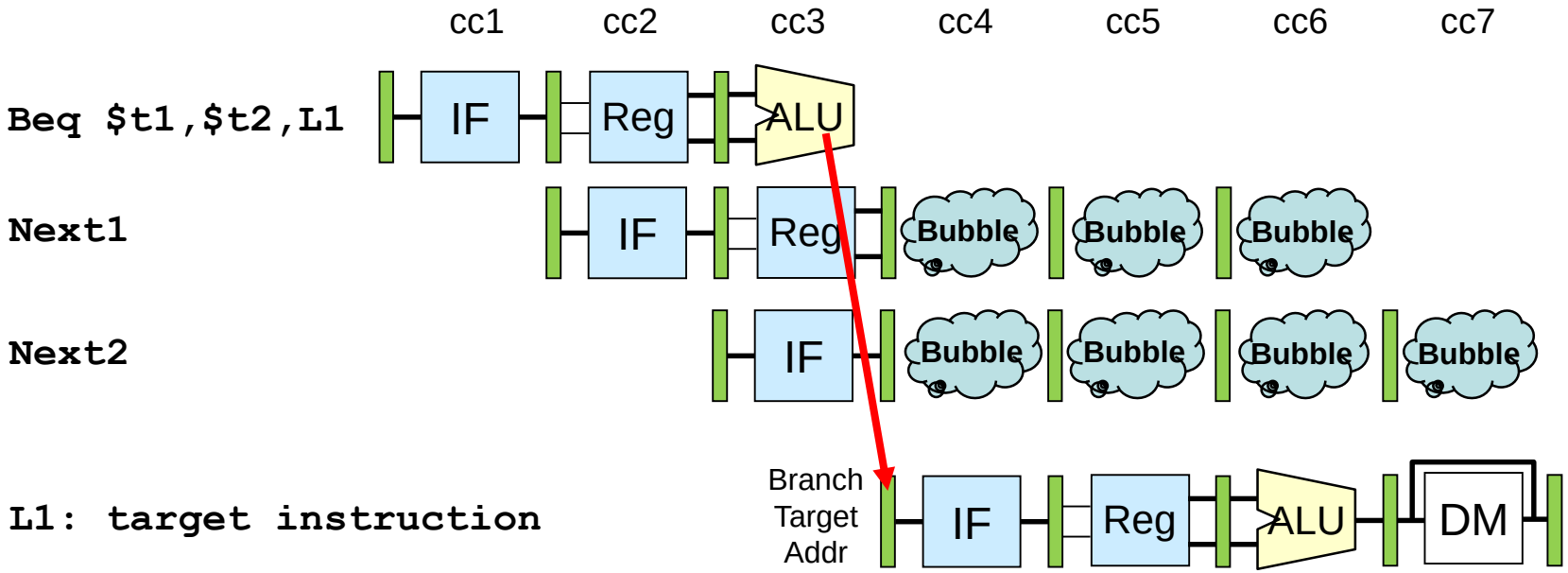
- Pipelining versus Serial Execution
- Pipelined Datapath and Control
- Pipeline Hazards
- Data Hazards and Forwarding
- Load Delay, Hazard Detection, and Stall
- Control Hazards
- Delayed Branch and Dynamic Branch Prediction

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 + 4 × 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)

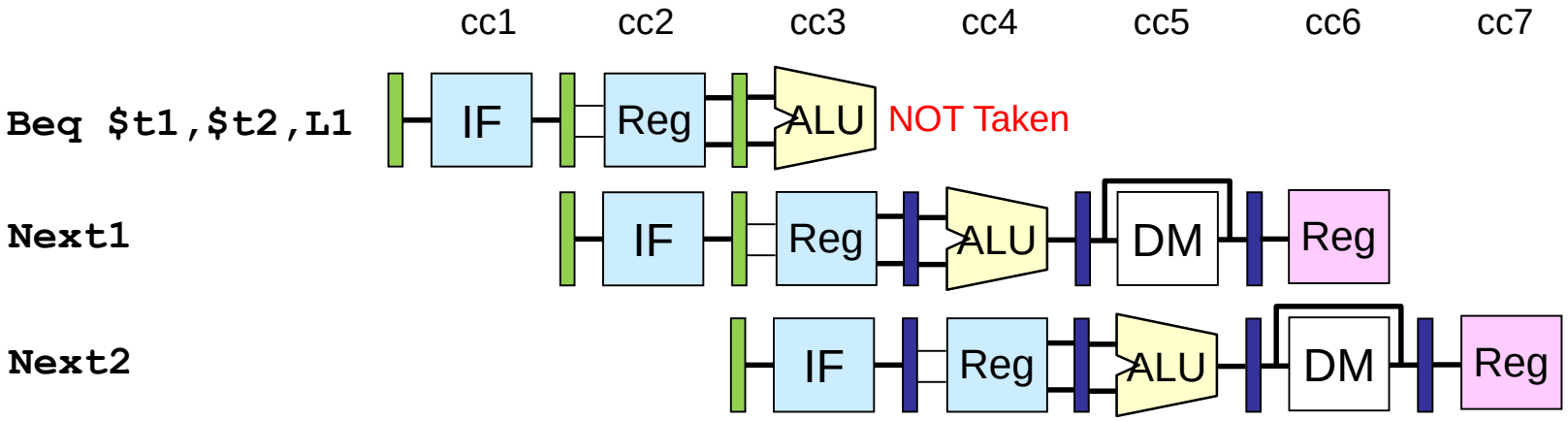
2-Cycle Branch Delay

- 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

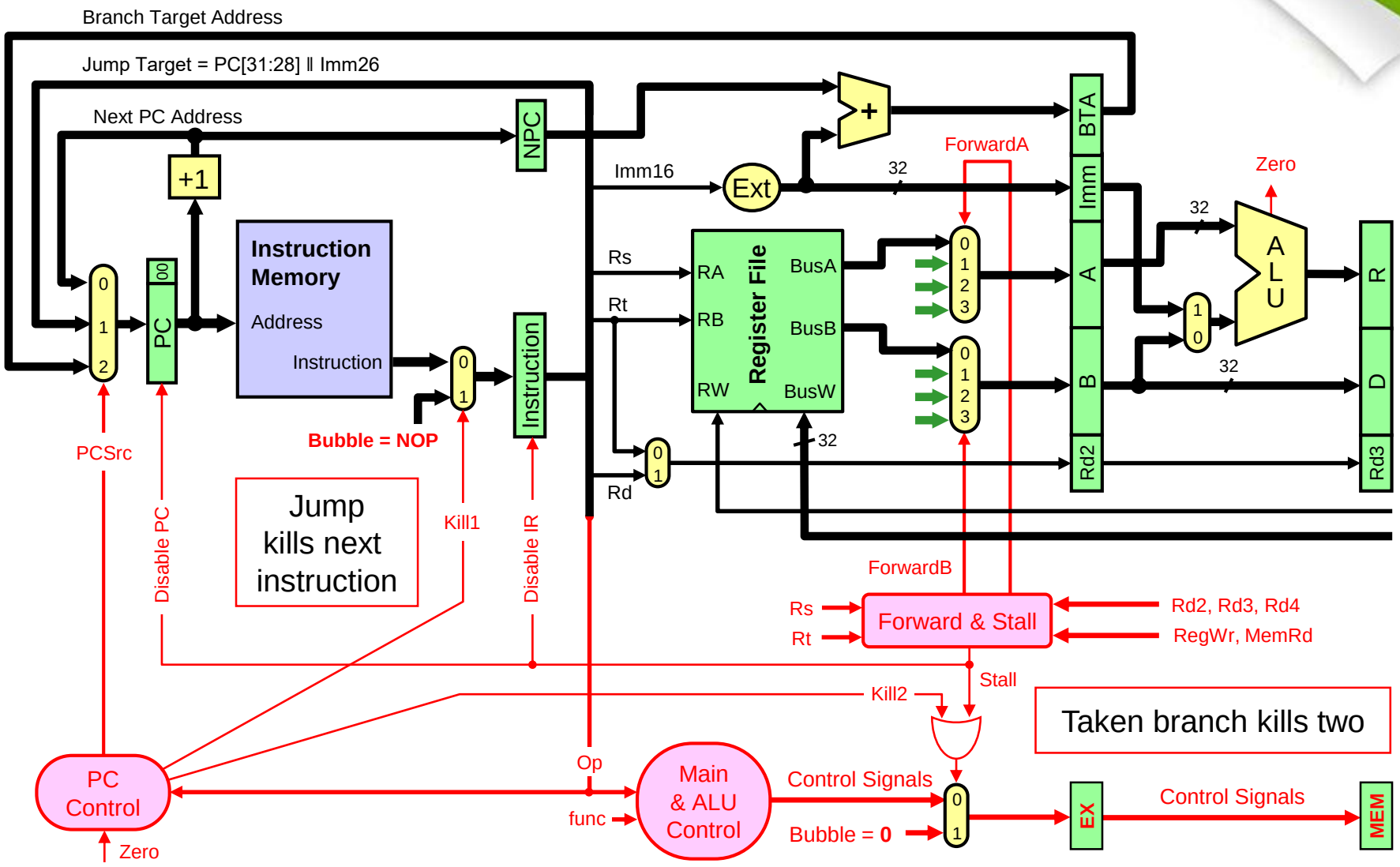


Predict Branch NOT Taken

- 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



Pipelined Jump and Branch



Jump and Branch Impact on CPI

- Base CPI = 1 without counting jump and branch
- Unconditional Jump = 5%, Conditional branch = 20%
- 90% of conditional branches are taken
- Jump kills next instruction, Taken Branch kills next two
- What is the effect of jump and branch on the CPI?

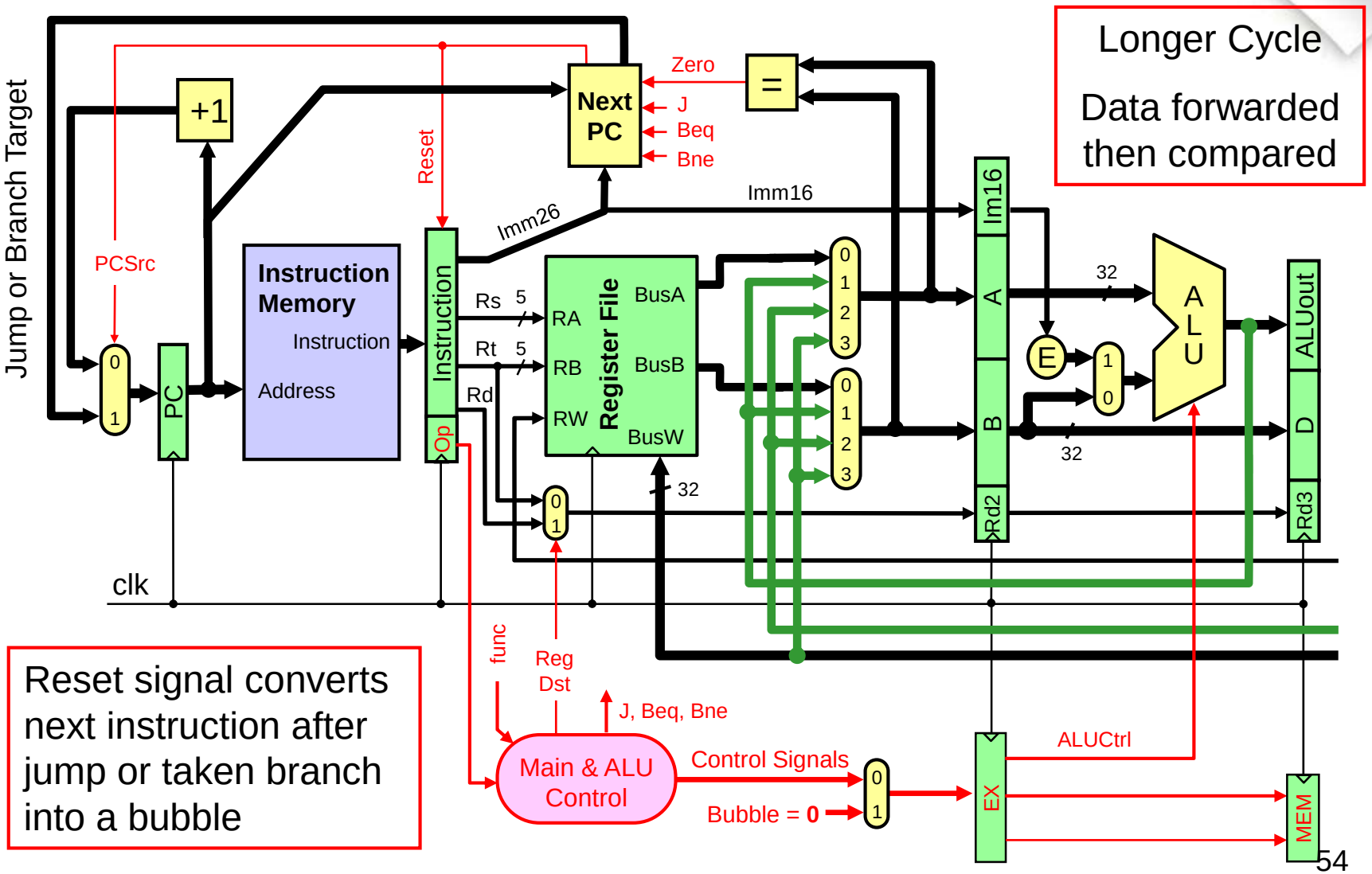
Solution:

- Jump adds 1 wasted cycle for 5% of instructions = 1 × 0.05
- Branch adds 2 wasted cycles for 20% × 90% of instructions
  = 2 × 0.2 × 0.9 = 0.36
- New CPI = 1 + 0.05 + 0.36 = 1.41 (due to 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



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

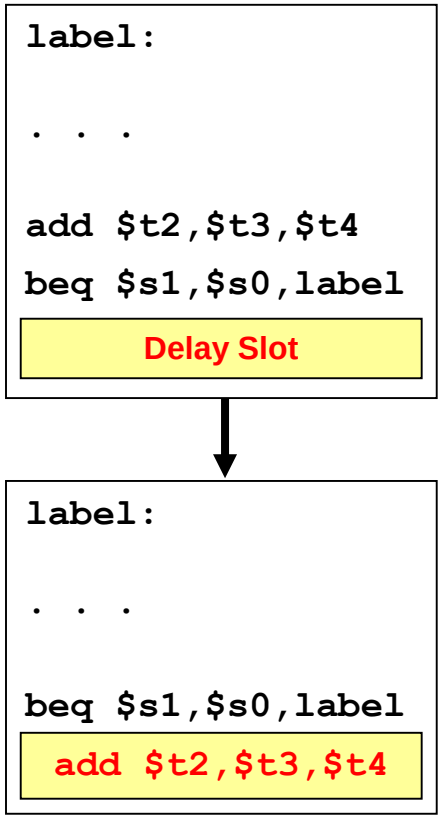
- Define branch to take place after the next instruction
- For a 1-cycle branch delay, we have one delay slot

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



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 and 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

Zero-Delayed Branching

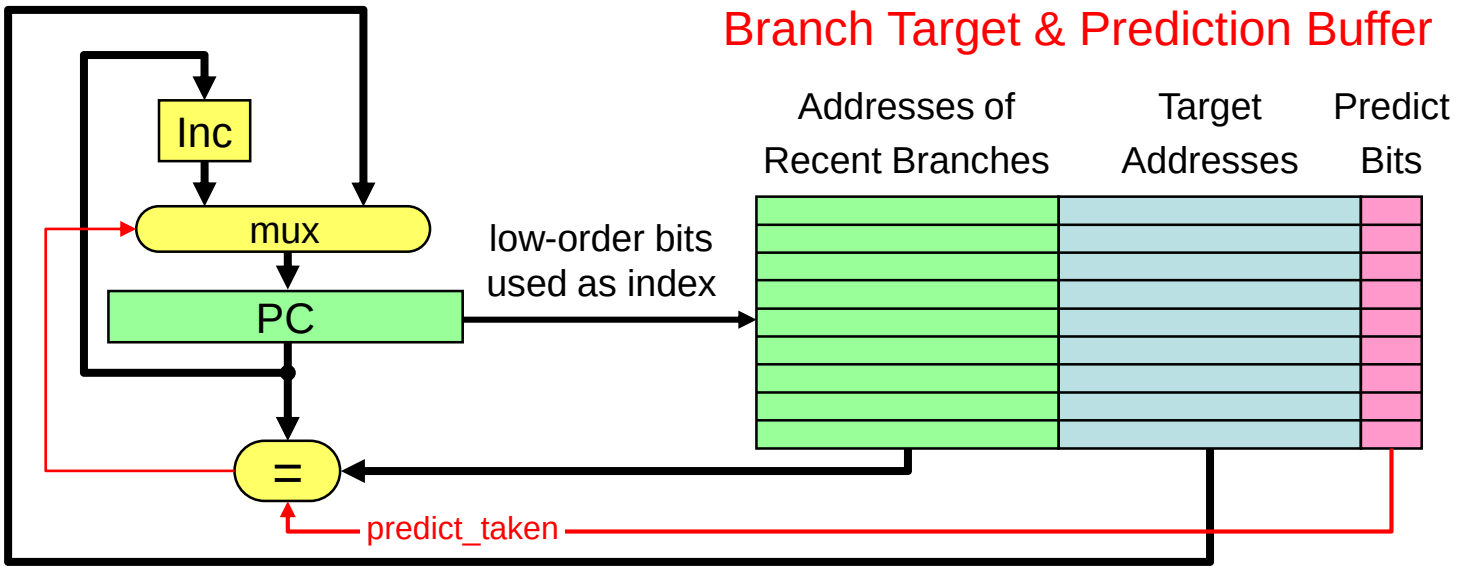
- How to achieve zero delay for a jump or a taken branch?
  - Jump or branch target address is computed in the ID stage
  - Next instruction has already been fetched in the IF stage

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 (IF Stage)

- 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



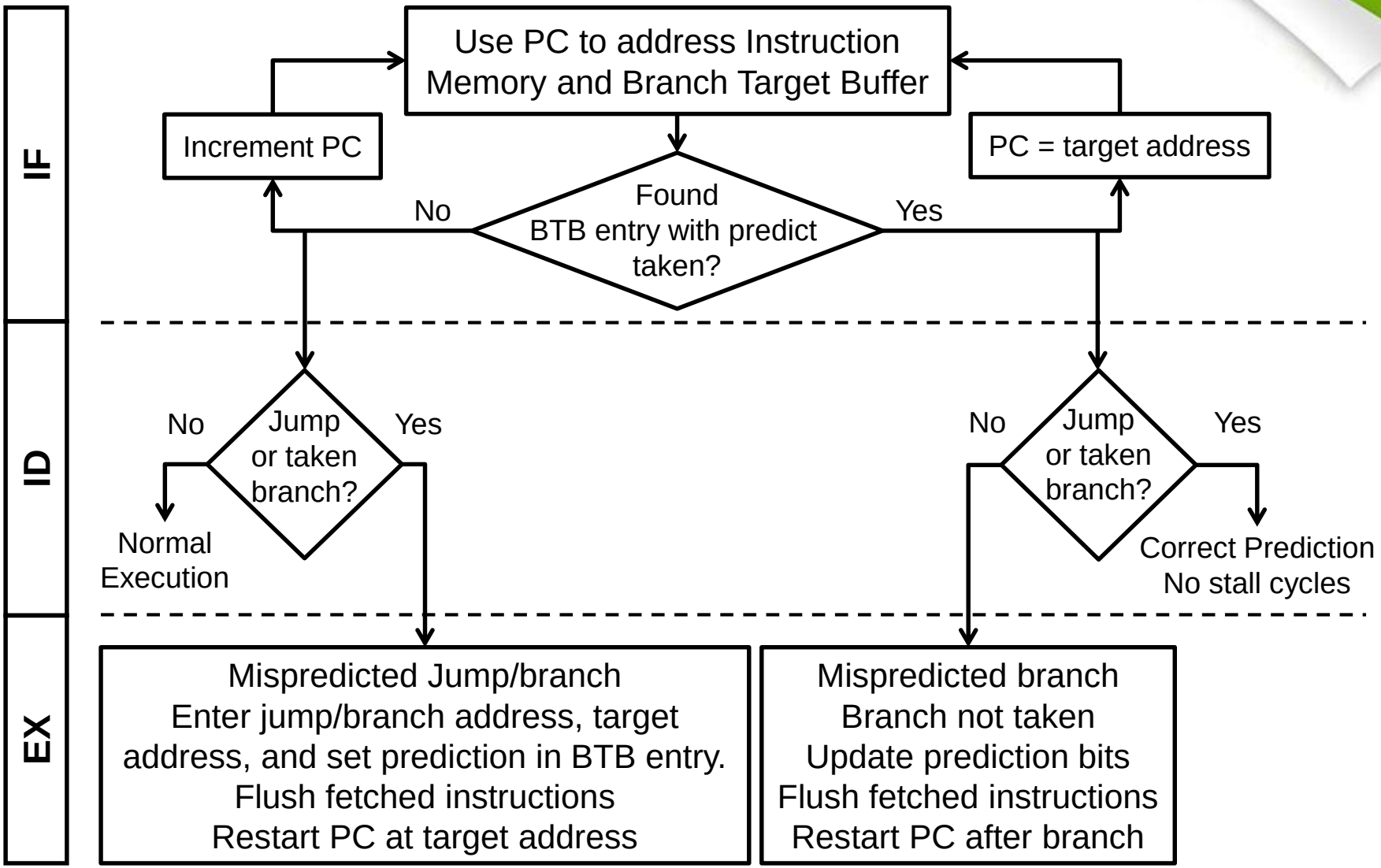
Branch Target Buffer – cont'd

- Each Branch Target Buffer (BTB) entry stores:
  - Address of a recent jump or branch instruction
  - Target address of jump or branch
  - Prediction bits for a conditional branch (Taken or Not Taken)
  - To predict jump/branch target address and branch outcome before instruction is decoded and branch outcome is computed
- Use the lower bits of the PC to index the BTB
  - Check if the PC matches an entry in the BTB (jump or branch)
  - If there is a match and the branch is predicted to be Taken then Update the PC using the target address stored in the BTB
- The BTB entries are updated by the hardware at runtime

Dynamic Branch Prediction

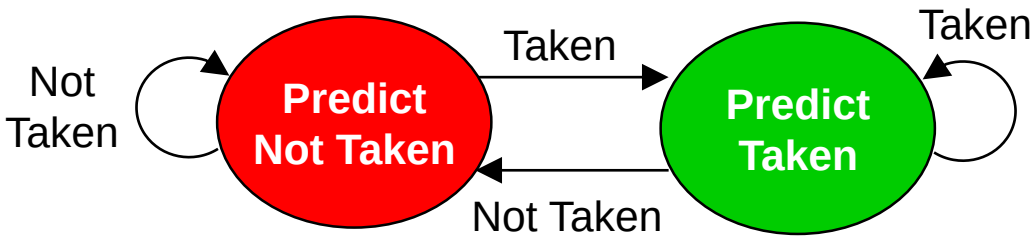
- Prediction of branches at runtime using prediction bits
- Prediction bits are associated with each entry in the BTB
  - Prediction bits reflect the recent history of a branch instruction
- Typically few prediction bits (1 or 2) are used per entry
- We don't know if the prediction is correct or not
- 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



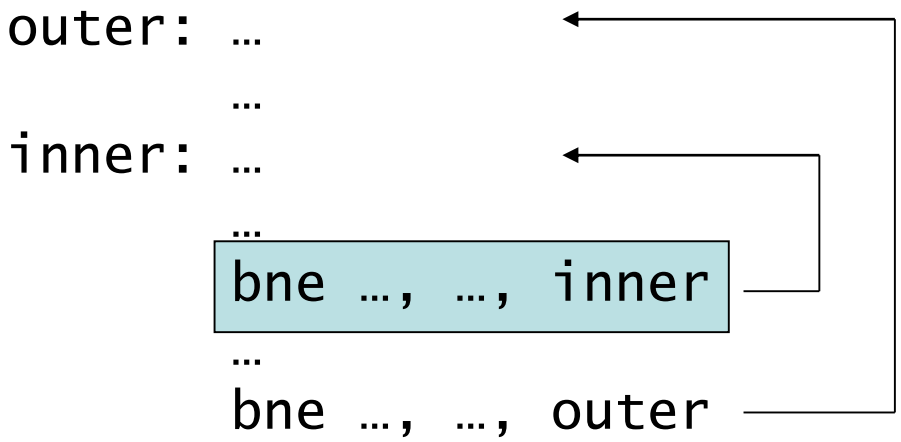
1-bit Prediction Scheme

- 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



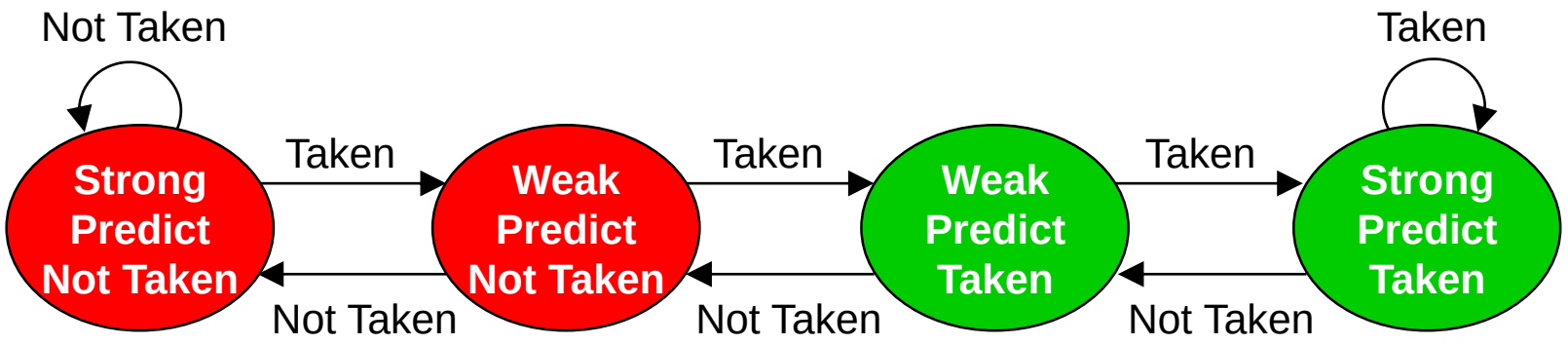
1-Bit Predictor: Shortcoming

- 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



Evaluating Branch Alternatives

Branch Scheme	Jump	Branch Not Taken	Branch Taken
Predict not taken	Penalty = 1 cycle	Penalty = 0 cycles	Penalty = 2 cycles
Delayed	Penalty = 0 cycles	Penalty = 0 cycles	Penalty = 1 cycle
BTB Prediction	Penalty = 1 cycle	Penalty = 2 cycles	Penalty = 2 cycles

- Assume: Jump = 2%, Branch-Not-Taken = 5%, Branch-Taken = 15%
- Assume a branch target buffer with hit rate = 90% for jump & branch
- Prediction accuracy for jump = 100%, for conditional branch = 95%
- What is the impact on the CPI? (Ideal CPI = 1 if no control hazards)

Branch Scheme	Jump = 2%	Branch NT = 5%	Branch Taken = 15%	CPI
Predict not taken	0.02 × 1	0	0.15 × 2 = 0.30	1+0.32
Delayed	0	0	0.15 × 1 = 0.15	1+0.15
BTB Prediction	0.02×0.1×1	0.05×0.9×0.05×2	0.15×(0.1+0.9×0.05)×2	1+0.05

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 forwarding
  - However, load delay cannot be eliminated and stalls the pipeline
  - Delayed branching can be a solution when branch delay = 1 cycle
  - BTB with branch prediction can reduce branch delay to zero
  - Branch misprediction should flush the wrongly fetched instructions