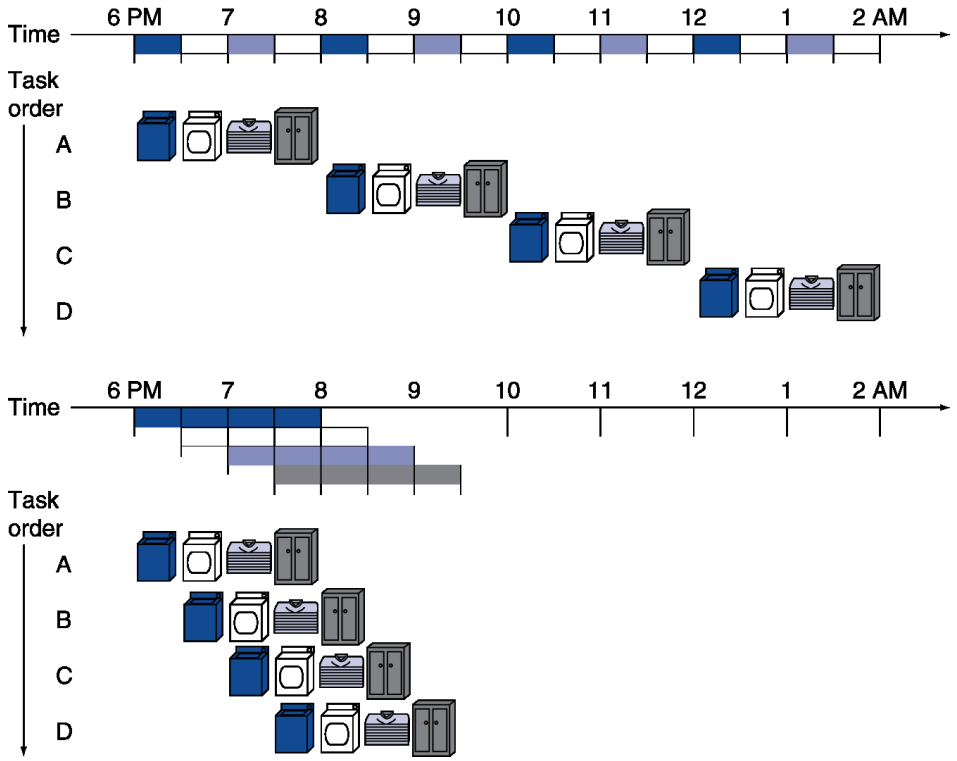
Pipelined MIPS Processor

Dmitri Strukov

ECE 154A

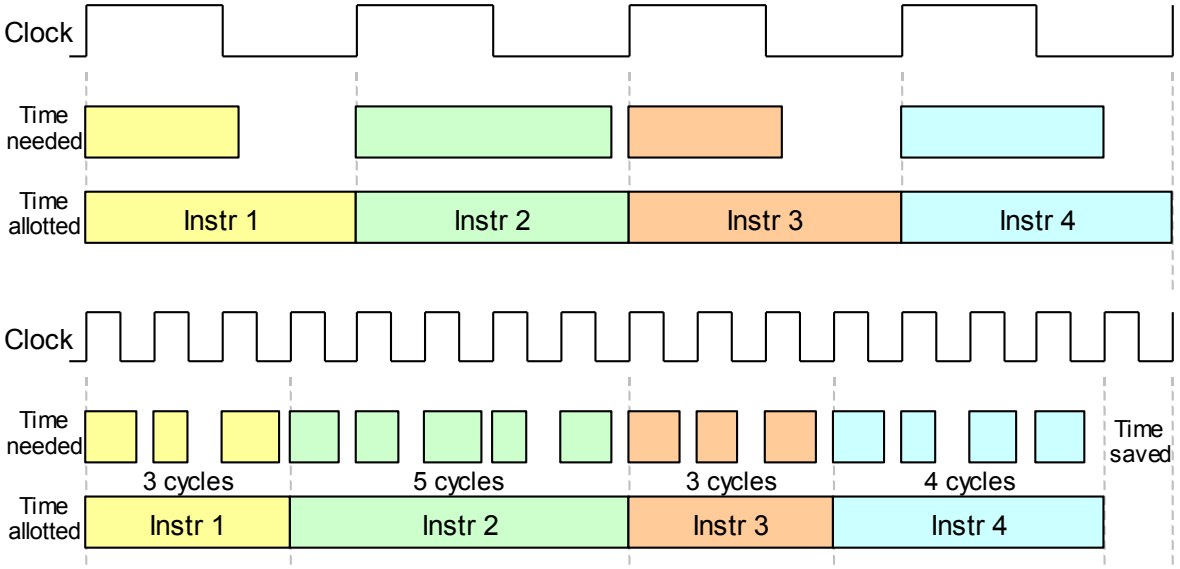
Pipelining Analogy

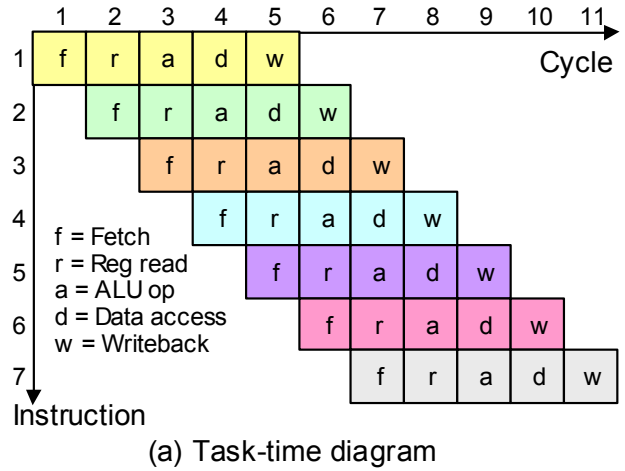
- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop:
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

Single-Cycle vs. Multicycle vs. Pipelined



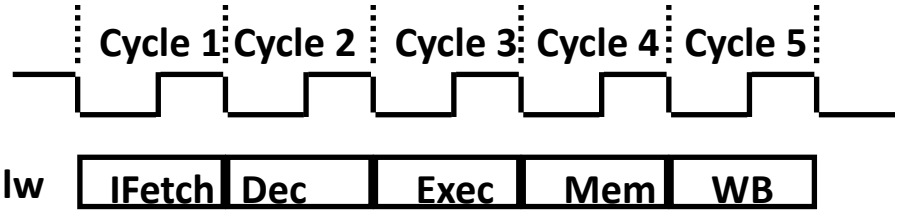


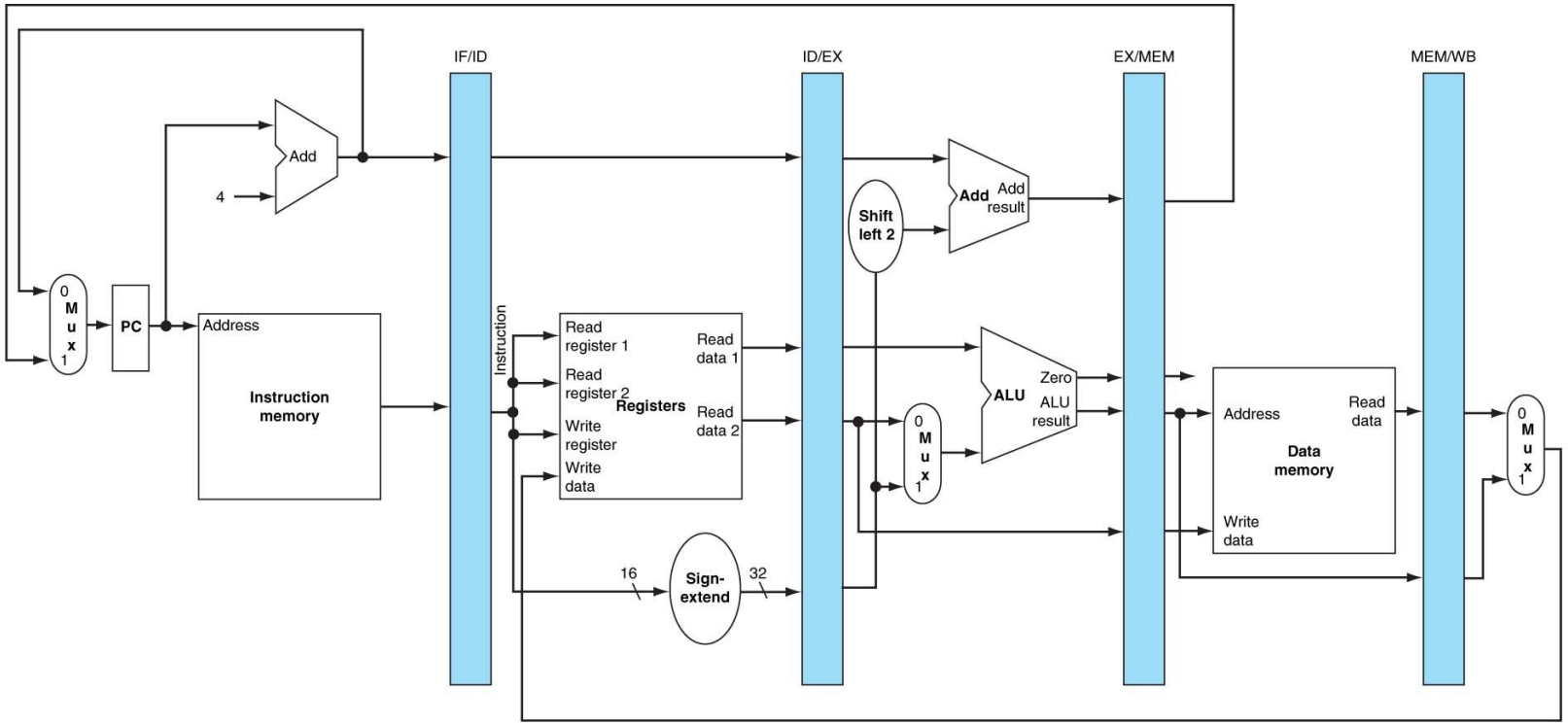
(a) Task-time diagram

MIPS Pipeline

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register



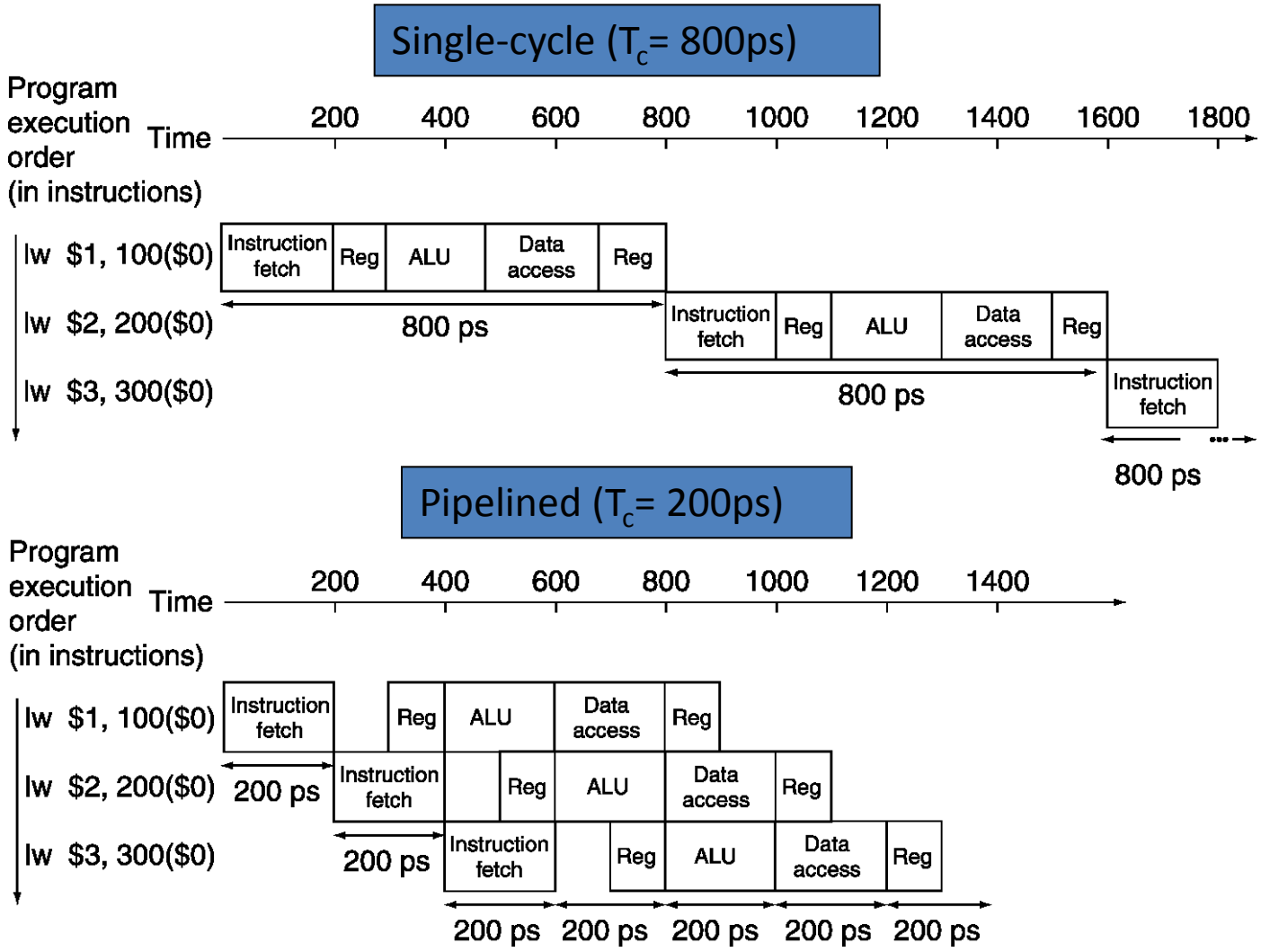


Pipeline Performance Example

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance Example



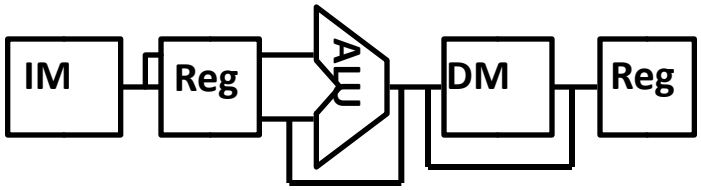
Pipeline Speedup Example

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Pipelining and ISA Design

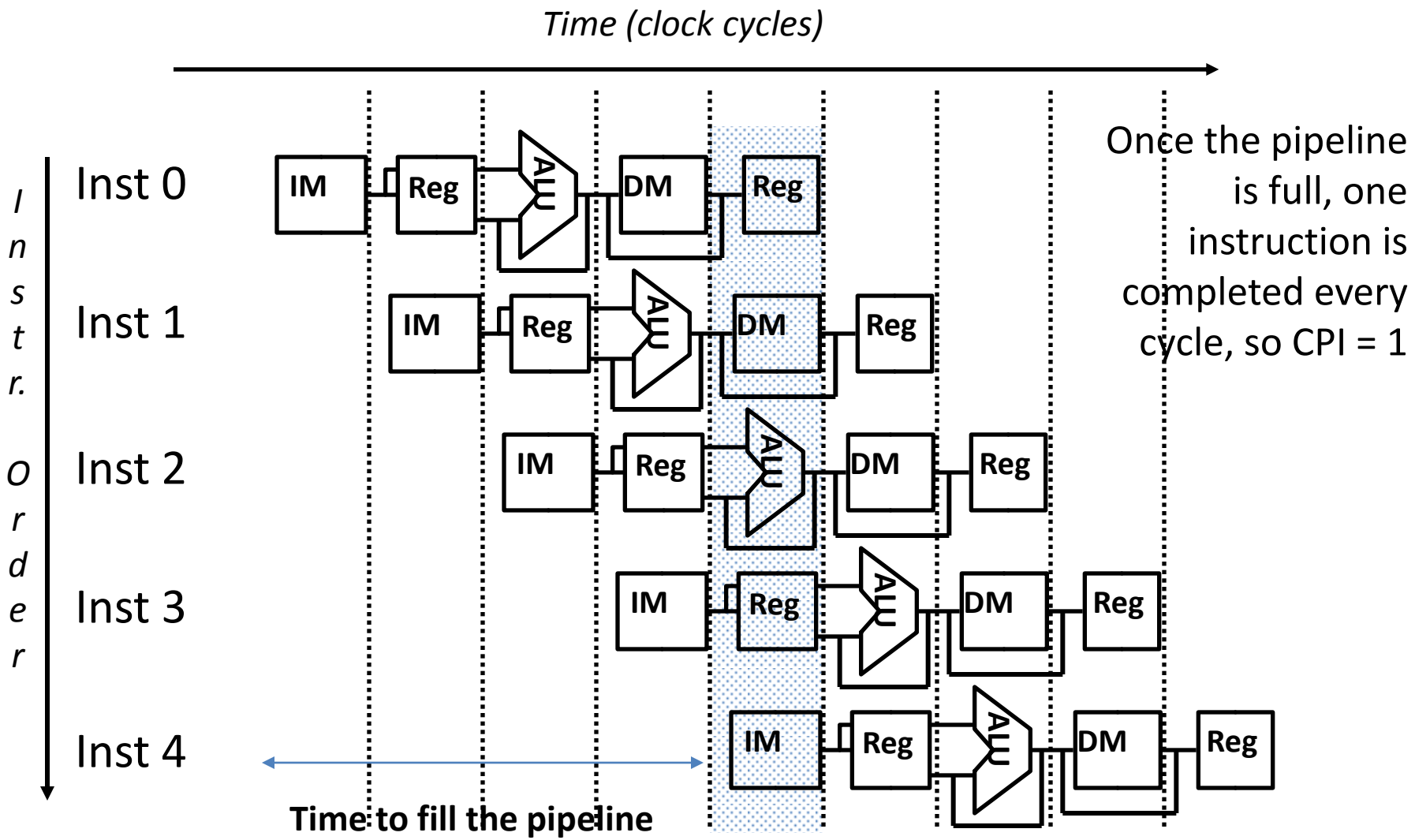
- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Graphically Representing MIPS Pipeline



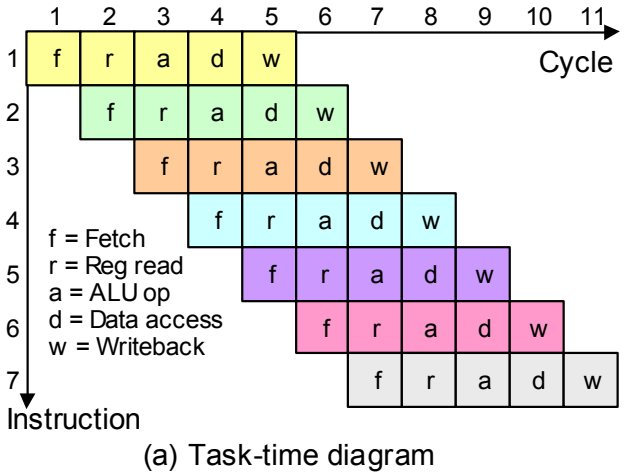
- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Why Pipeline? For Performance!

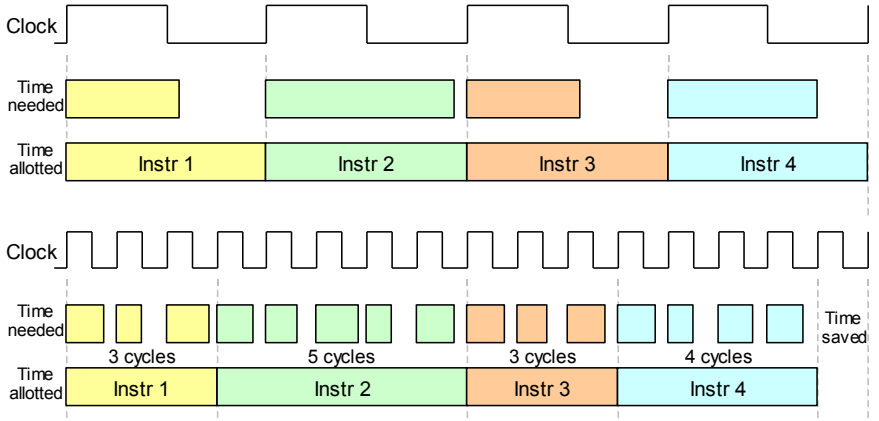


Review from Last Lecture

multi cycle pipelined



(a) Task-time diagram



Execution time = 1/ Performance = Inst count x CPI x CCT

N = # of stages for pipeline design or ~ maximum number of steps for MC

$CPI_{ideal\ MCP} = N$ /InstCount + 1 – 1/InstCount

→ large N and/or small InstCount result in worse CPI

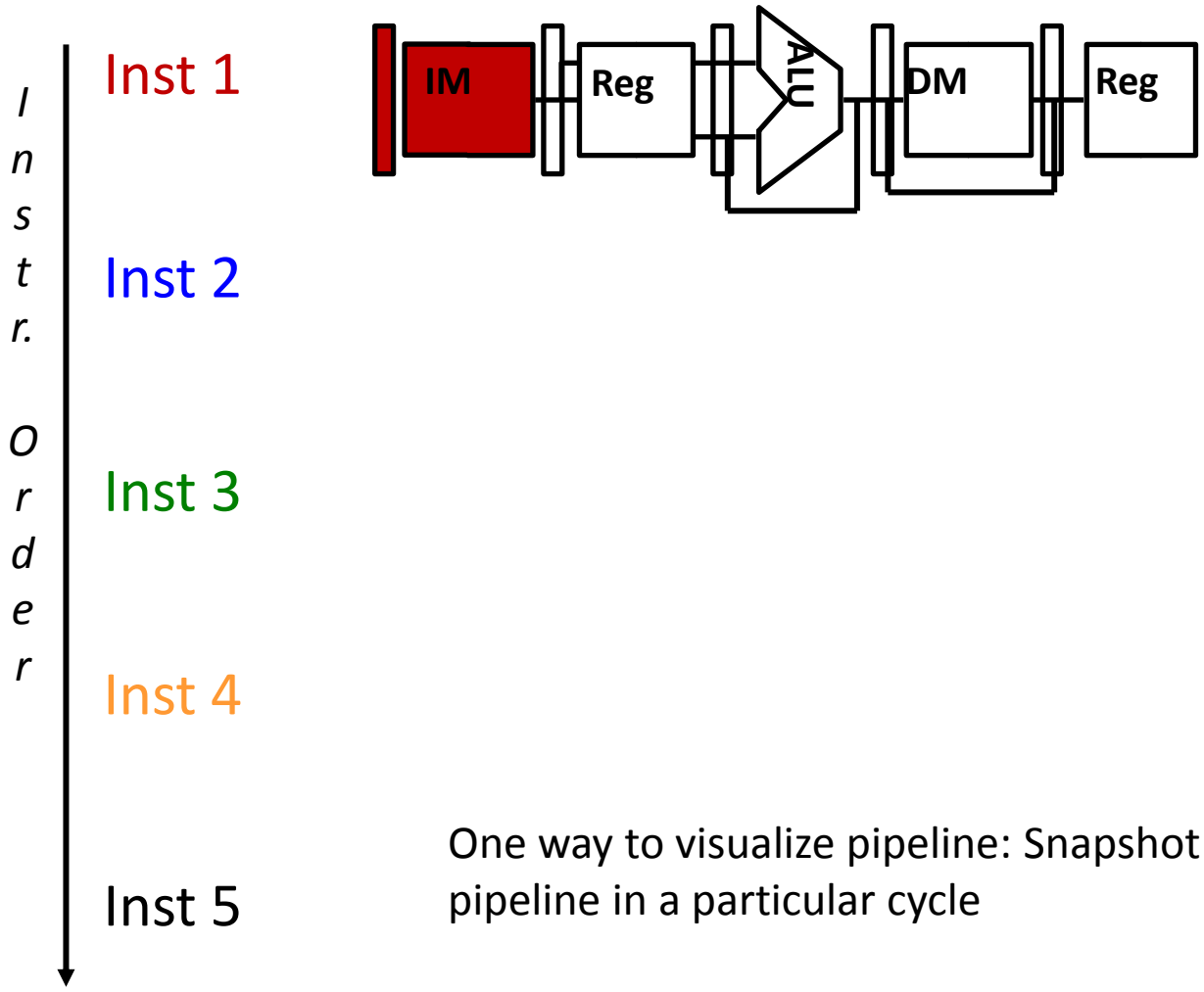
→ Performance to run one instruction is the same as of CP (i.e. latency for single instruction is not reduced)

Design	Inst count	CPI	CCT
Single Cycle (SC)	1	1	1
Multi cycle (MC)	1	$N \geq CPI > 1$ (closer to N than 1)	> 1/N
Multi cycle pipelined (MCP)	1	> 1	>1/N

What are the other issues affecting CCT and CPI for MC and MCP?

Visualizing pipeline - I

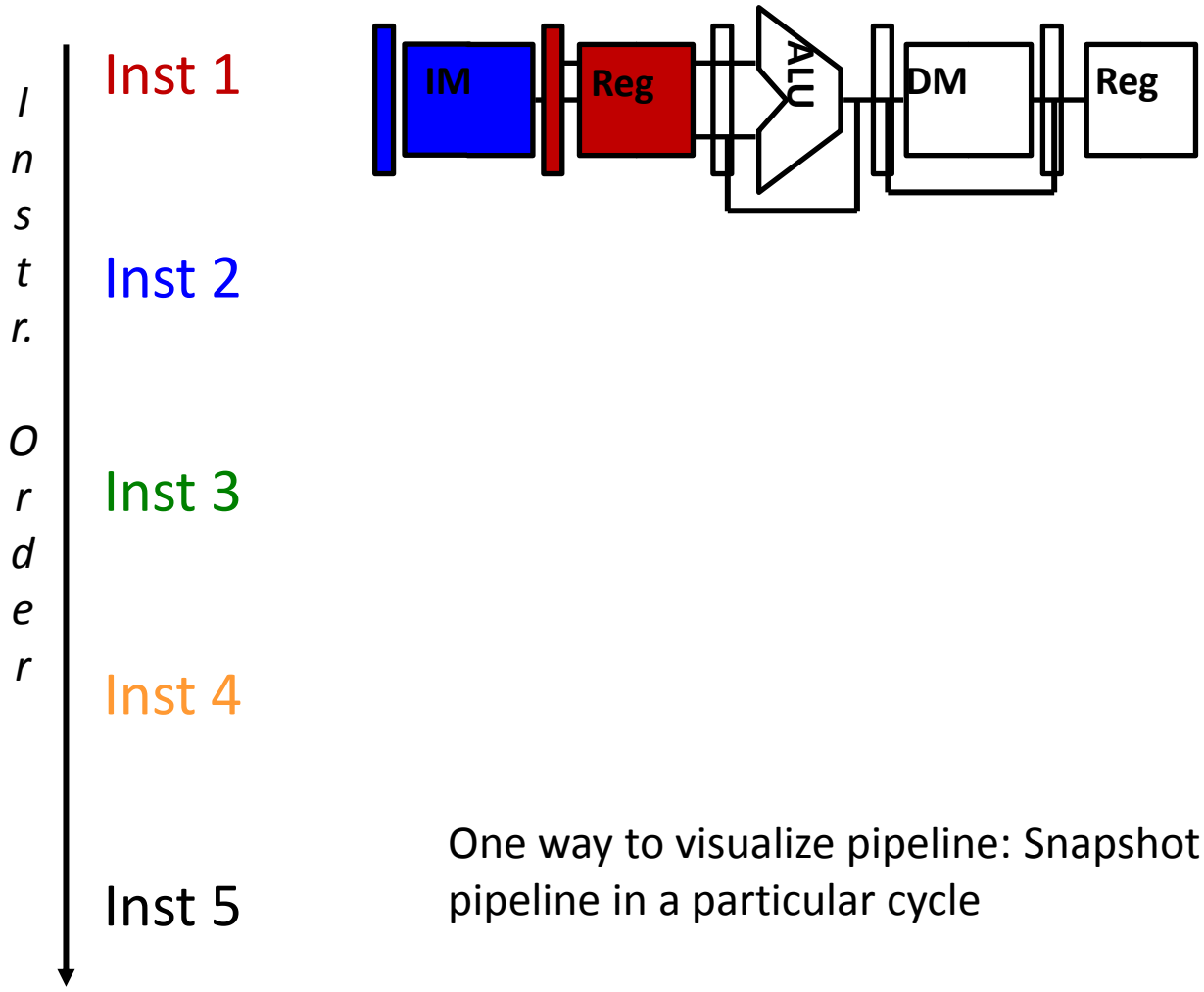
Cycle 1



One way to visualize pipeline: Snapshot of what it is in pipeline in a particular cycle

Visualizing pipeline - I

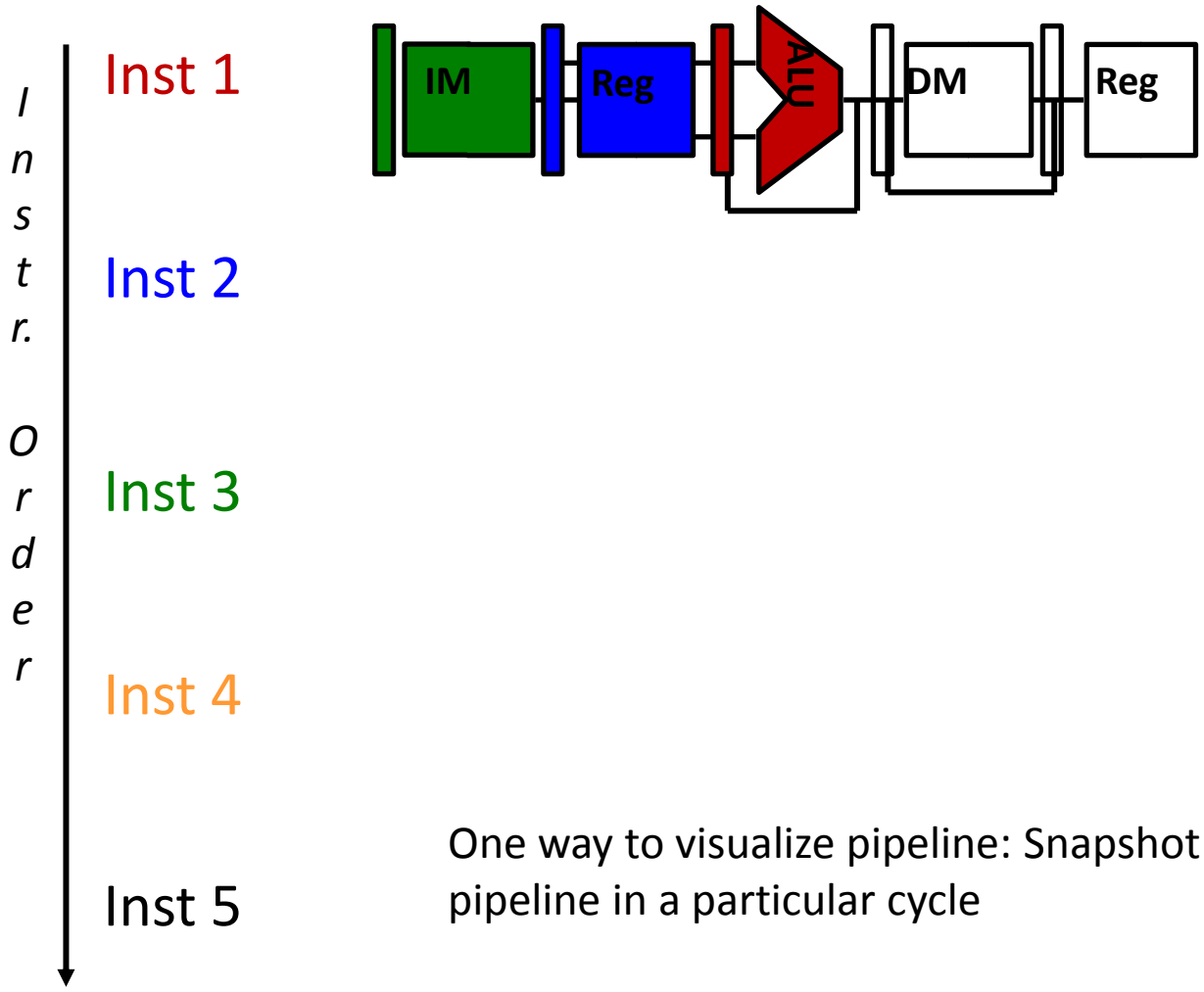
Cycle 2



One way to visualize pipeline: Snapshot of what it is in pipeline in a particular cycle

Visualizing pipeline - I

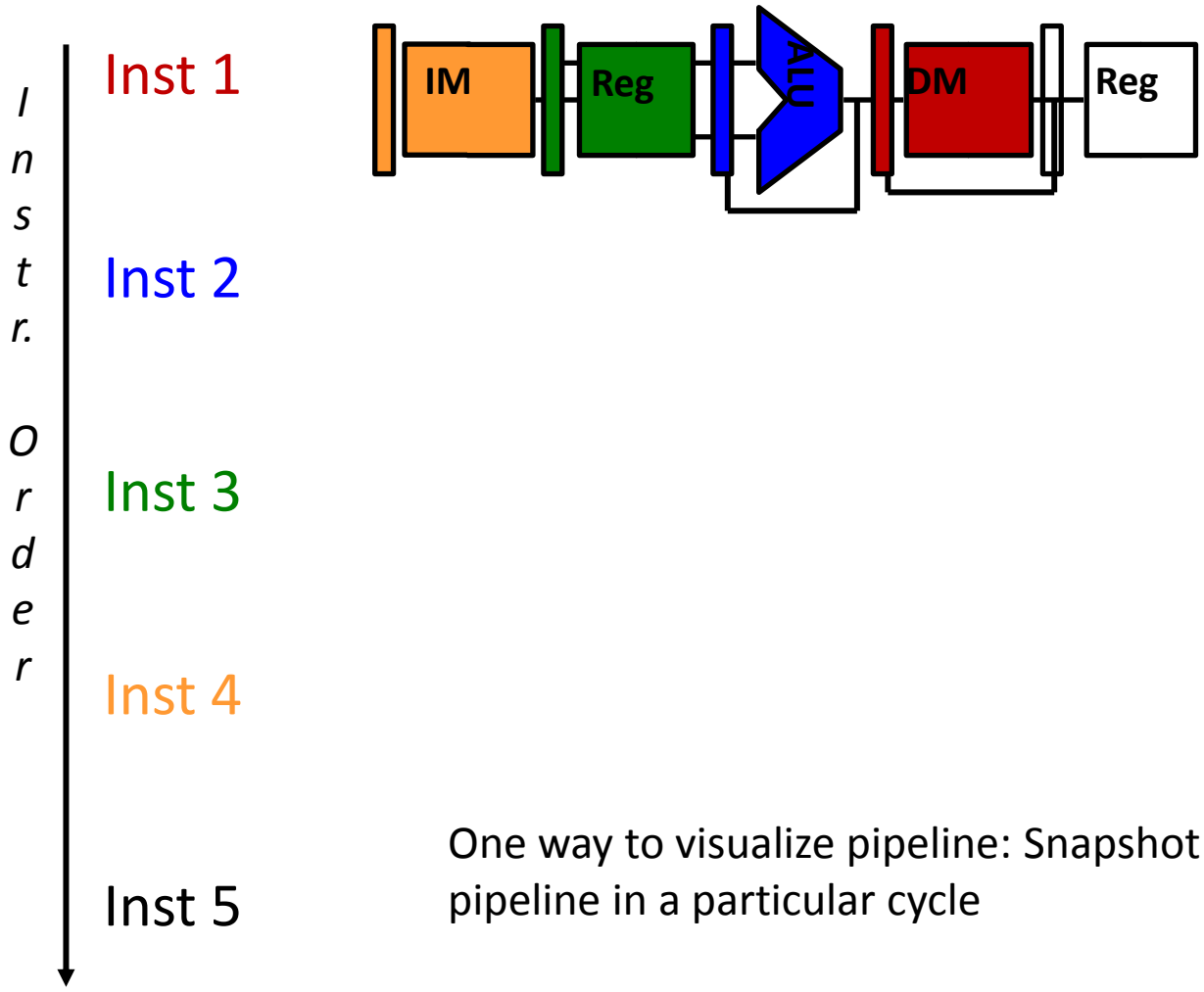
Cycle 3



One way to visualize pipeline: Snapshot of what it is in pipeline in a particular cycle

Visualizing pipeline - I

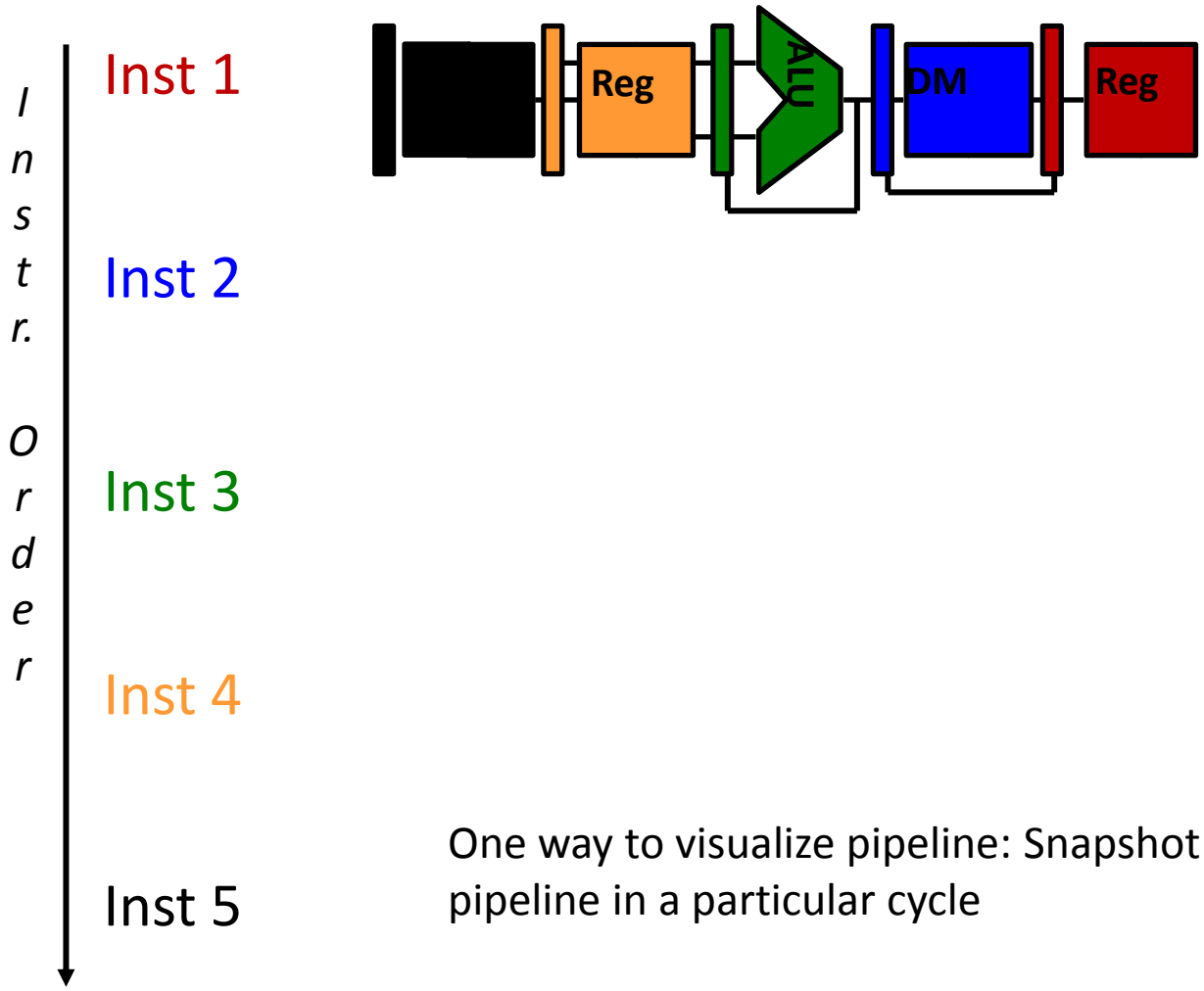
Cycle 4



One way to visualize pipeline: Snapshot of what it is in pipeline in a particular cycle

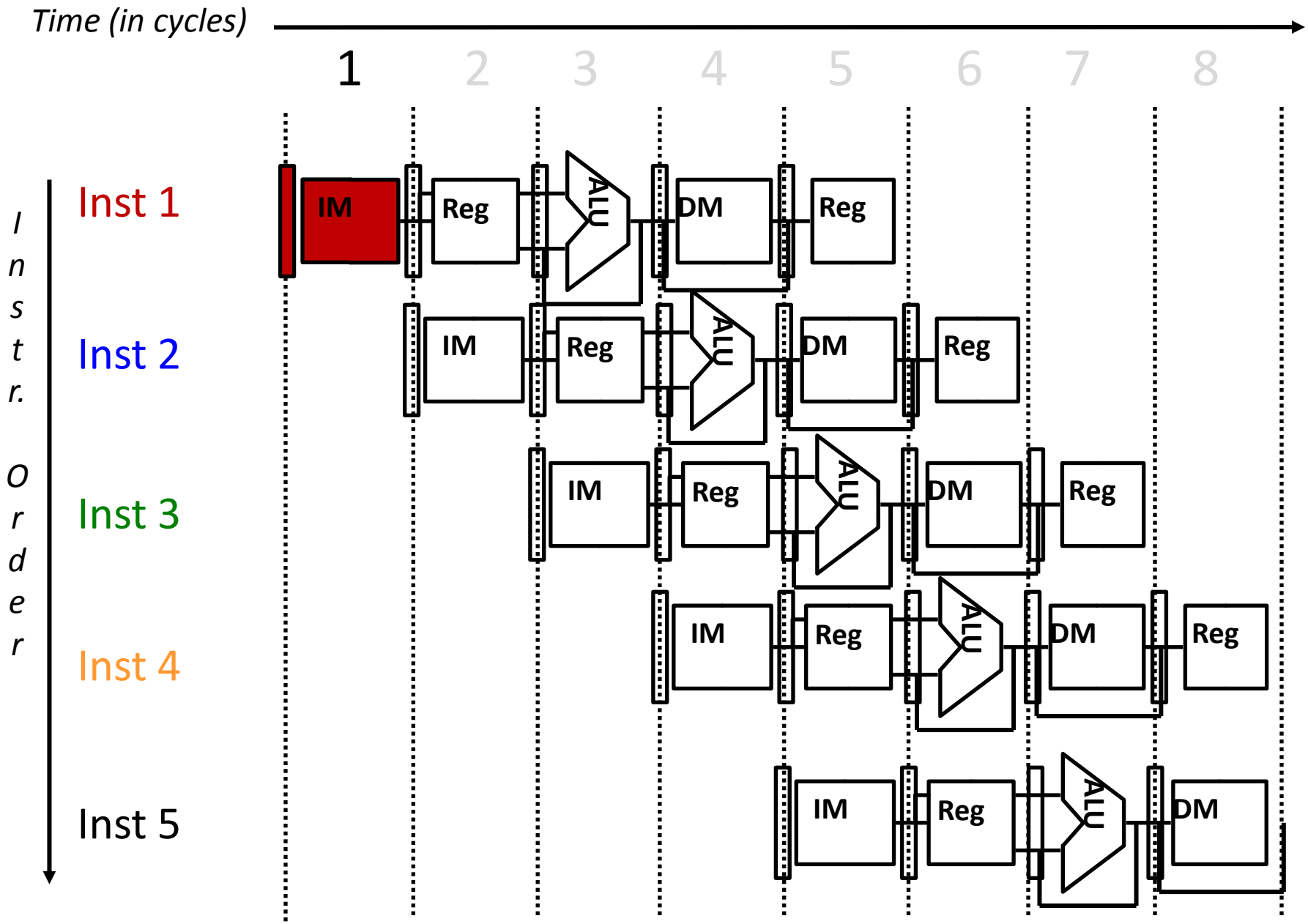
Visualizing pipeline - I

Cycle 5

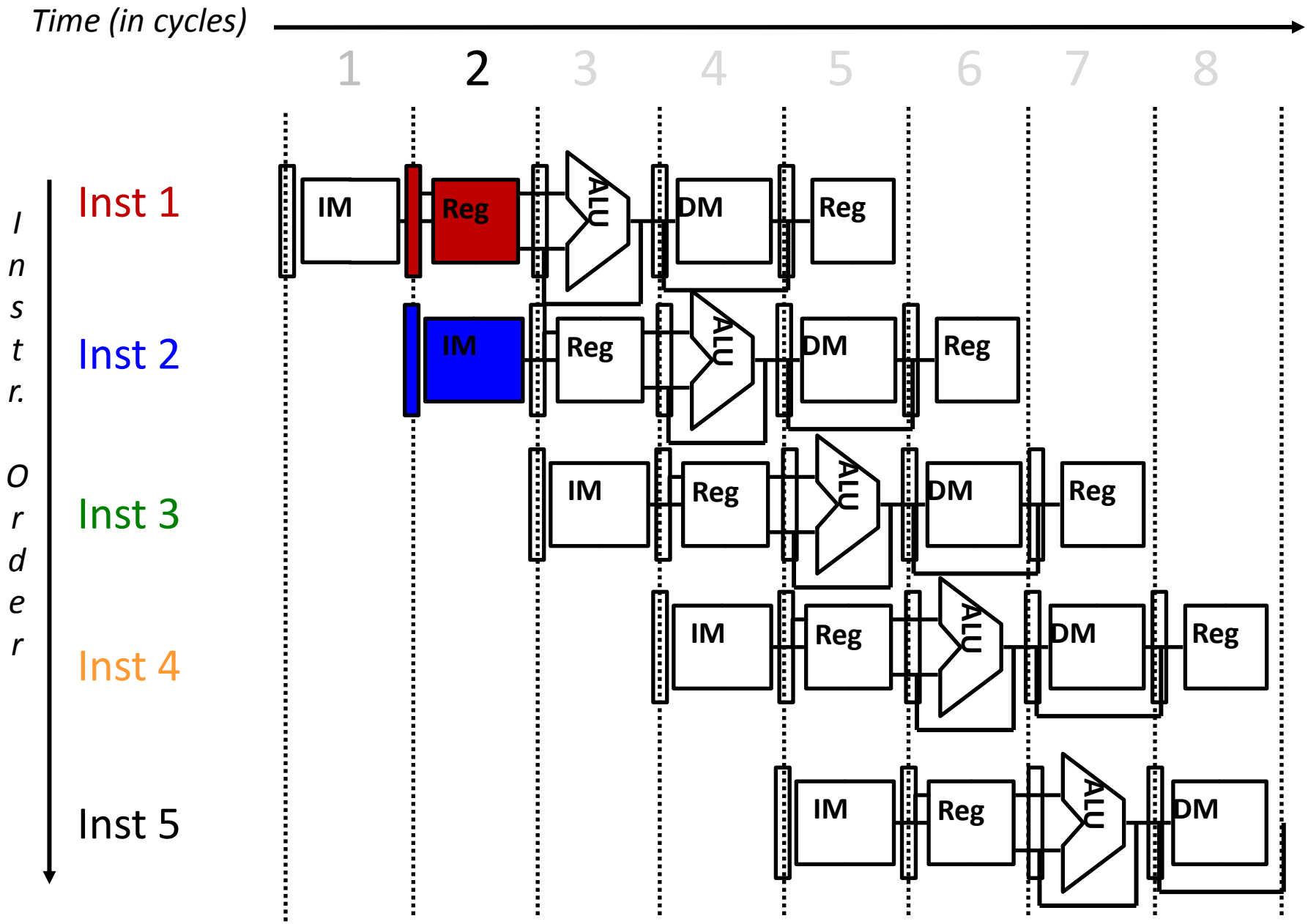


One way to visualize pipeline: Snapshot of what it is in pipeline in a particular cycle

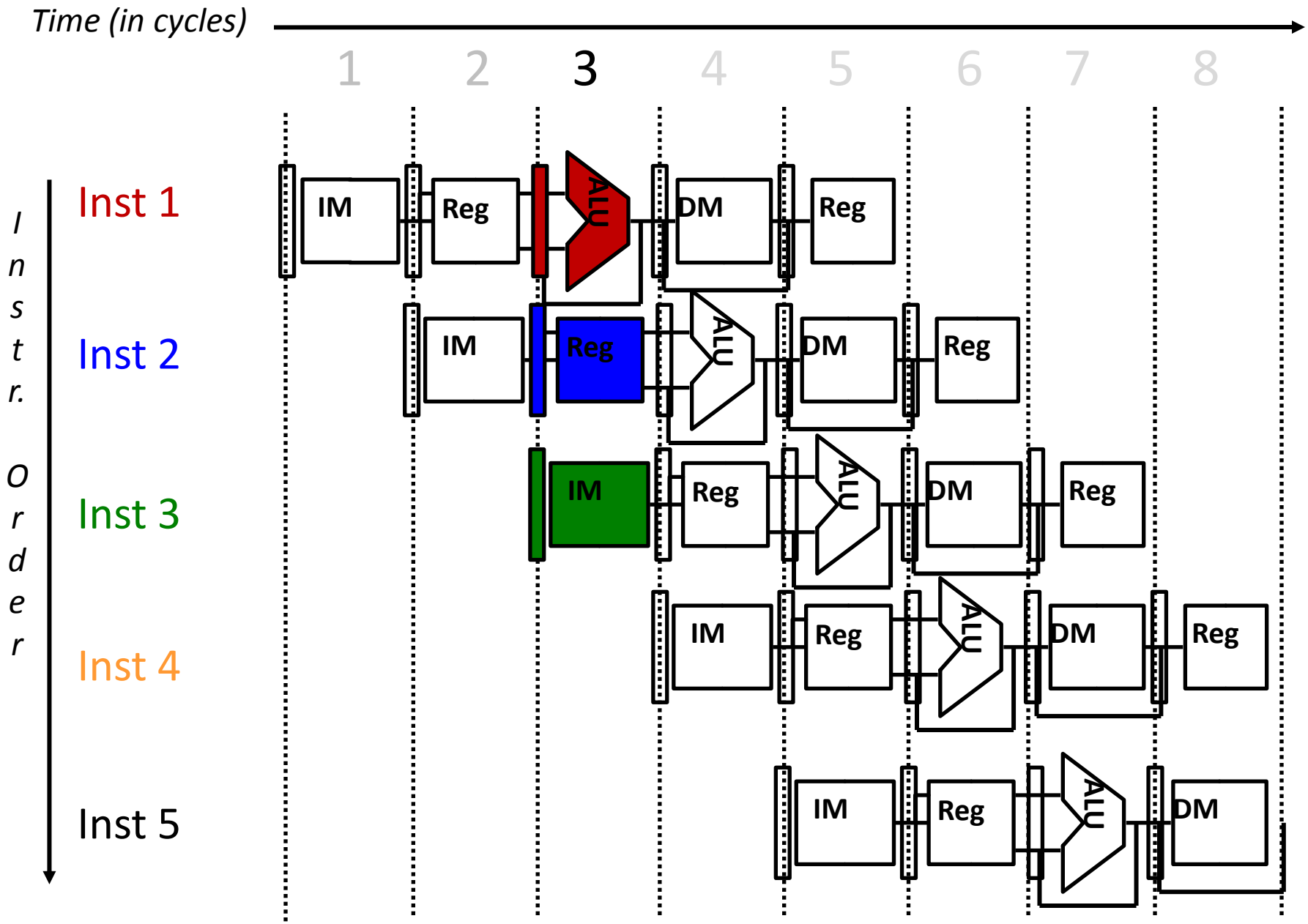
Visualizing pipeline - II



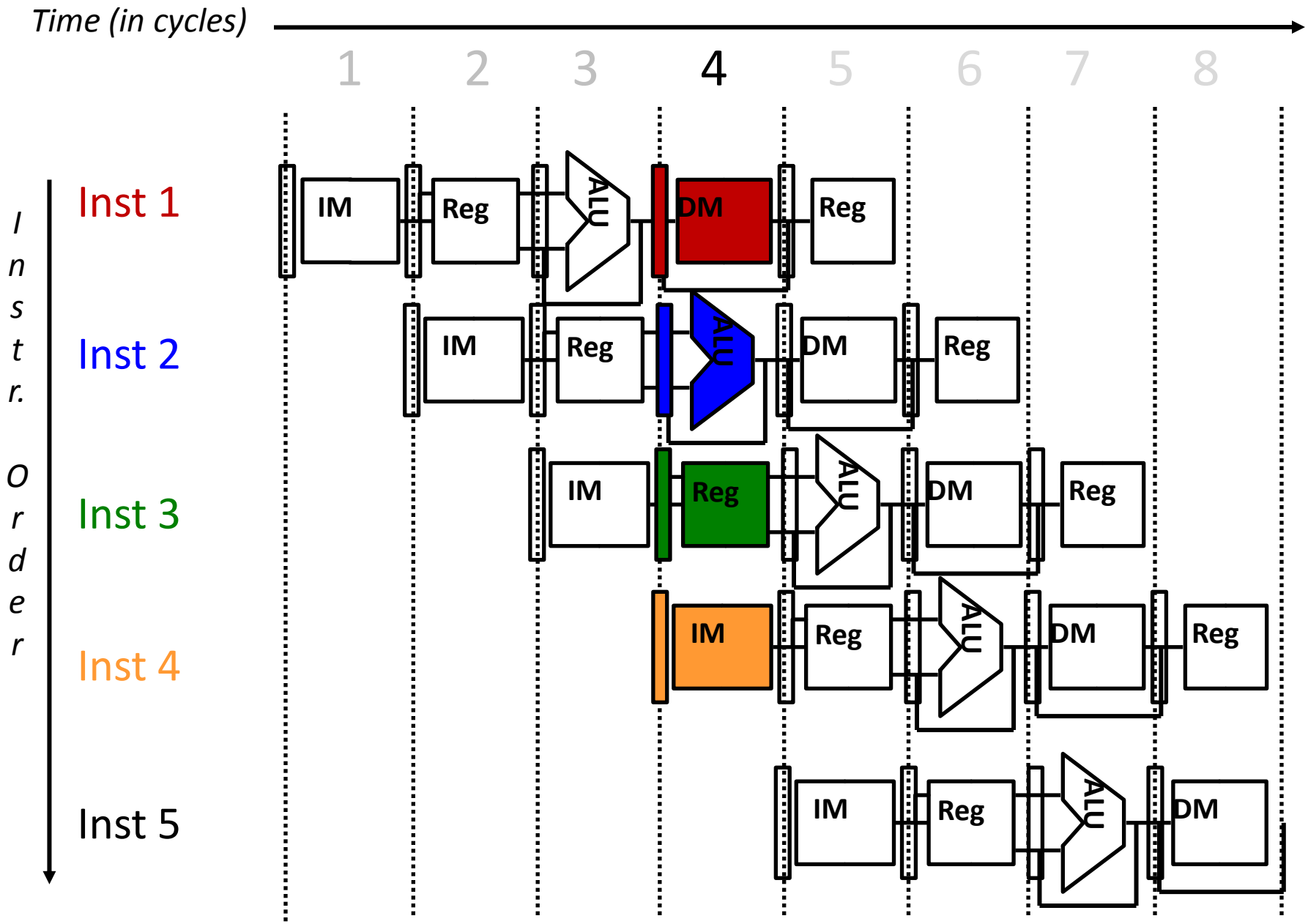
Visualizing pipeline - II



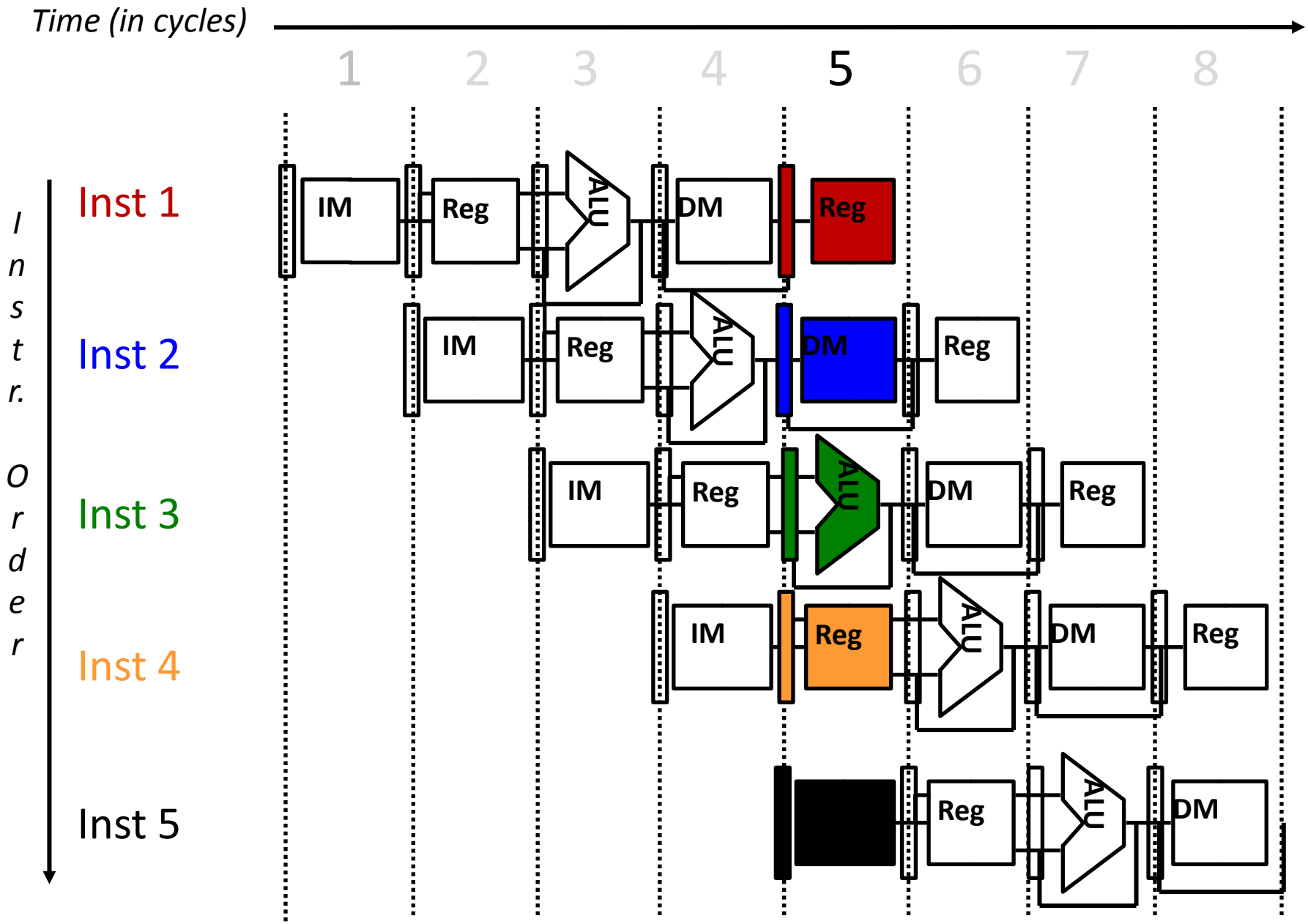
Visualizing pipeline - II



Visualizing pipeline - II



Visualizing pipeline - II



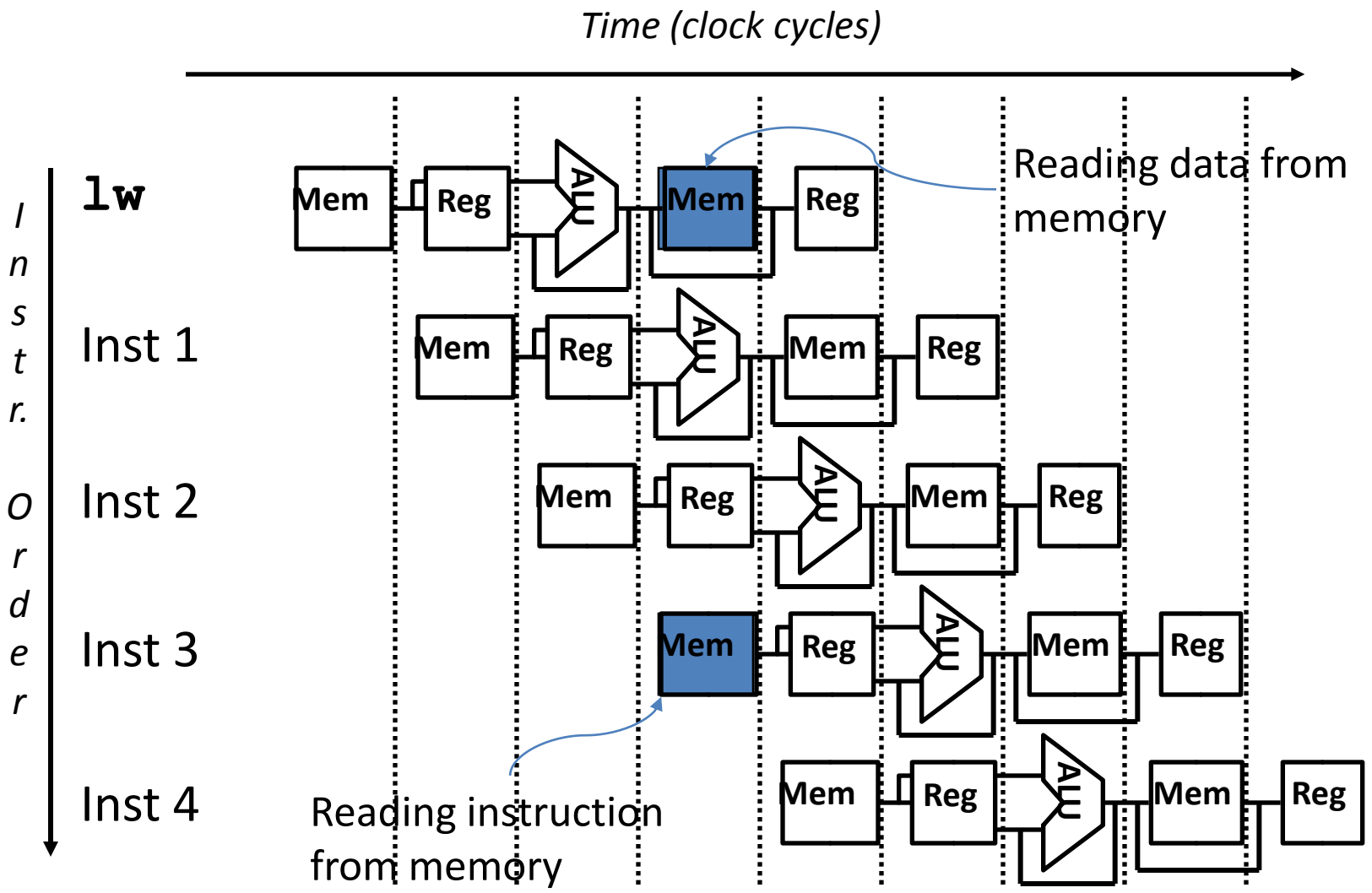
Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- 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”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

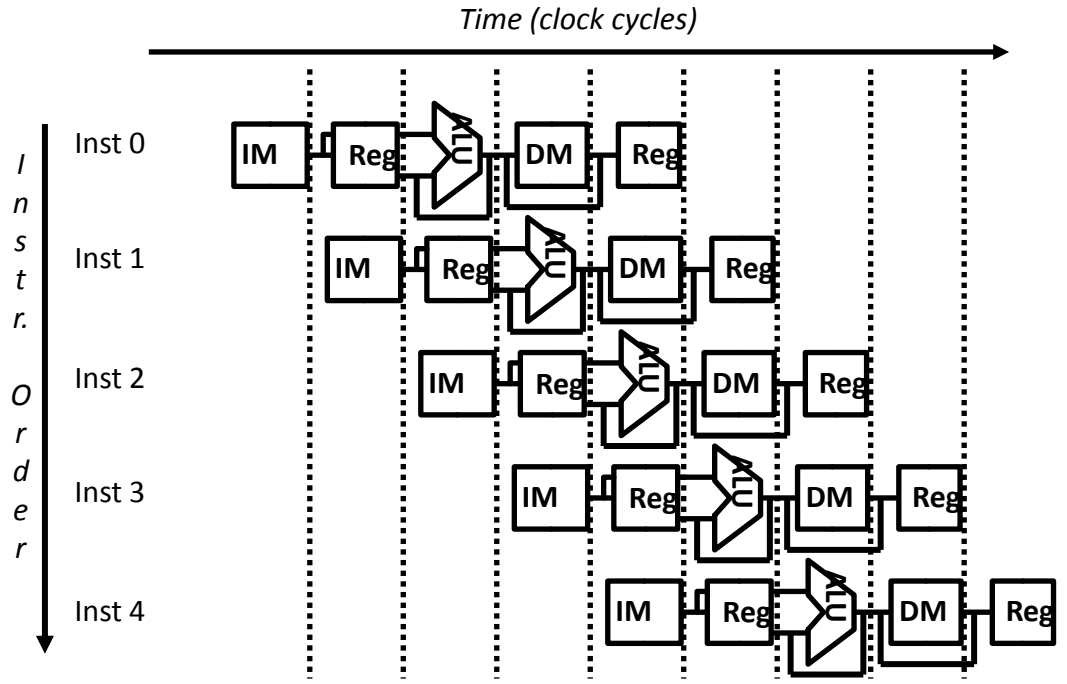
A Single Memory Would Be a Structural Hazard



- Fix with separate instr and data memories (I$ and D$)

Note that all instructions will take effectively 5 cycles even if some stages are not used for or instruction finishes early

Why?



Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add  $s0, $t0, $t1`
    `sub  $t2, $s0, $t3`

Data Dependencies

instruction *j is said data dependent on instruction i if either of the* following holds

1. Instruction *i produces a result that may be used by instruction j, or*
2. Instruction *j is data dependent on instruction k and instruction k is* data dependent on instruction *i*

Typically only type 1 data dependency is sufficient to satisfy for the correct execution of the program since type 2 dependency just implies that one instruction is dependent on another if there exist a chain of dependencies of the first type between the two instructions. A *dependency between two instructions will only result in a data hazard if the* instructions are close enough together for the considered simple datapath in class. In general, it may also become a hazard for advanced pipelined designs when the processor executes multiple and/or out-of-order instructions

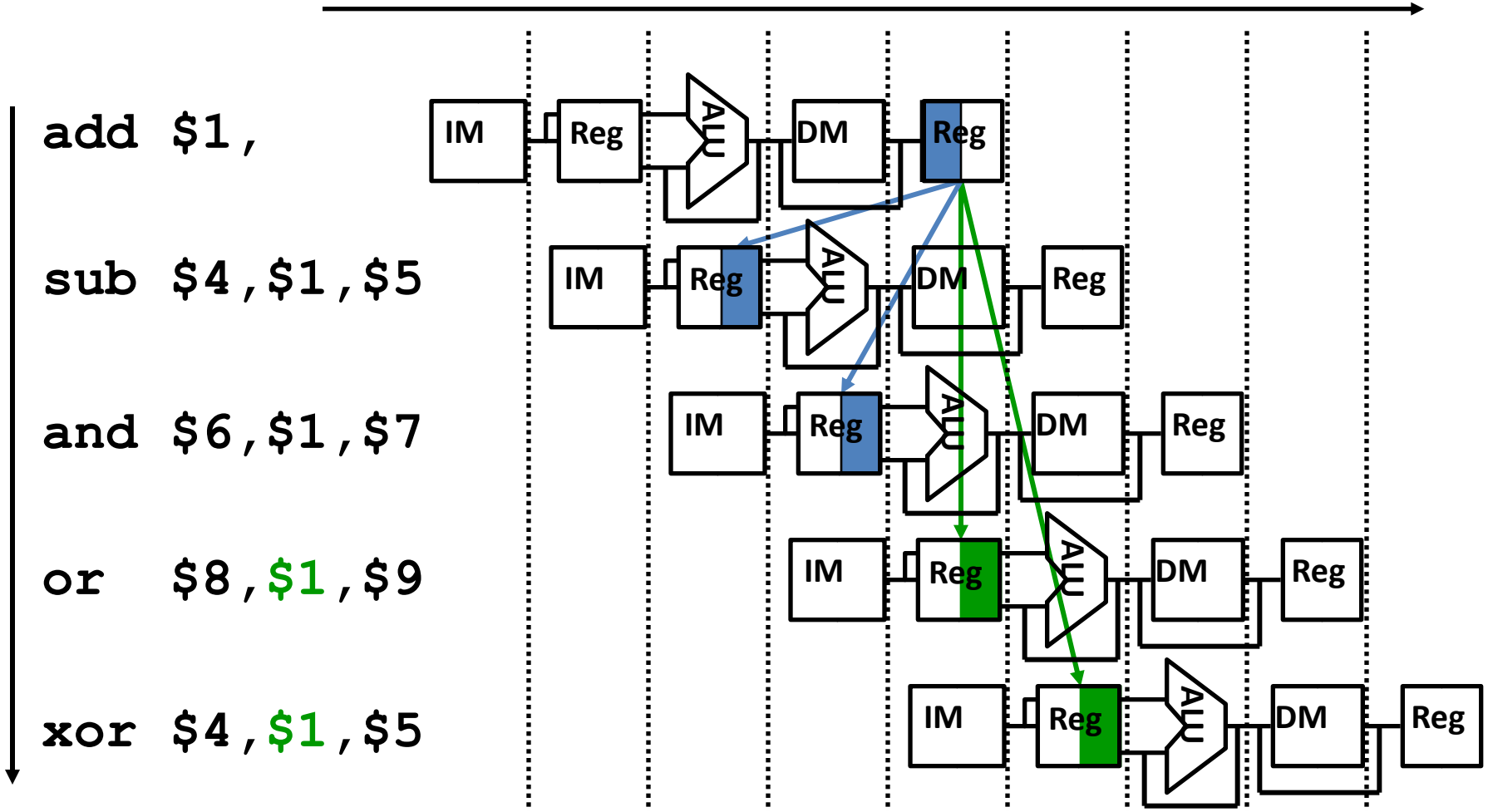
There are three particular data dependencies:

1. RAW (read after write) – *j reads a source after i writes it*
2. WAW (write after write) – *j writes an operand after it is written by I*
3. WAR (write after read) – *j writes a destination after it is read by i*

Note that RAW is what is called “true data dependency” because there is a flow of data between the instructions. WAW and WAR are called “name dependency”, since two instructions use the same register of memory location (but there is no flow of data between the instructions).

Register Usage Can Cause Data Hazards

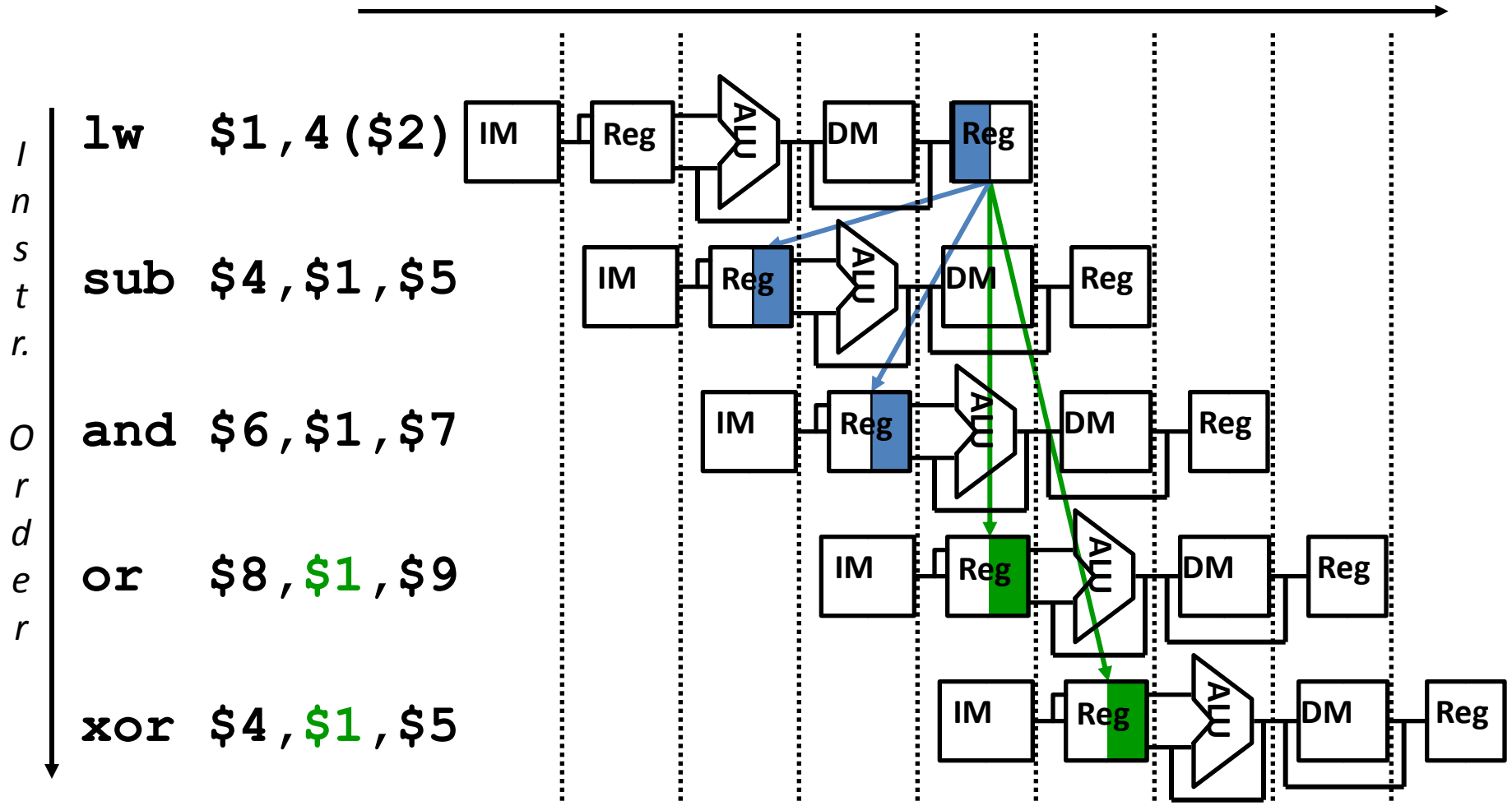
- Dependencies backward in time cause hazards



- Read before write data hazard

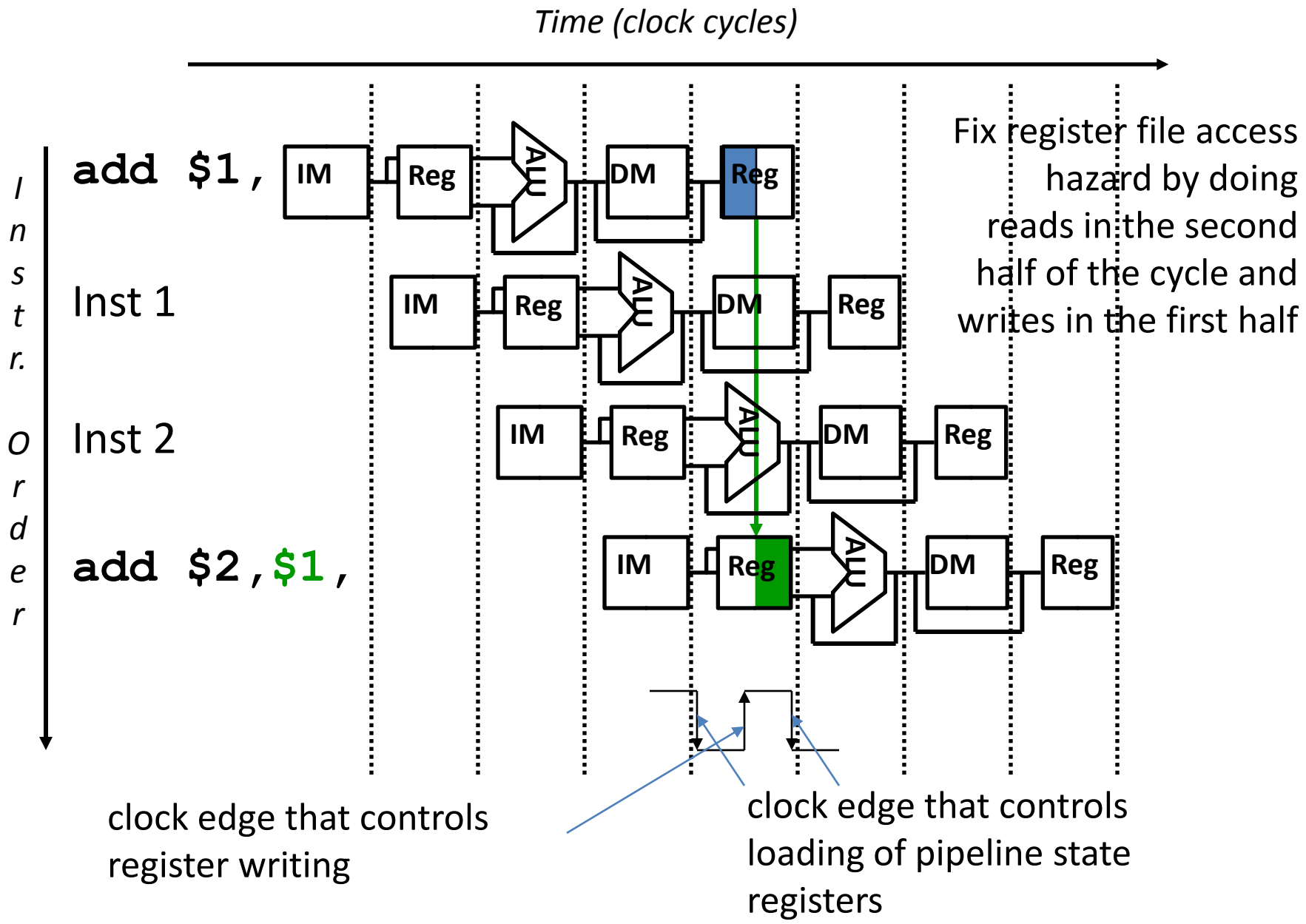
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

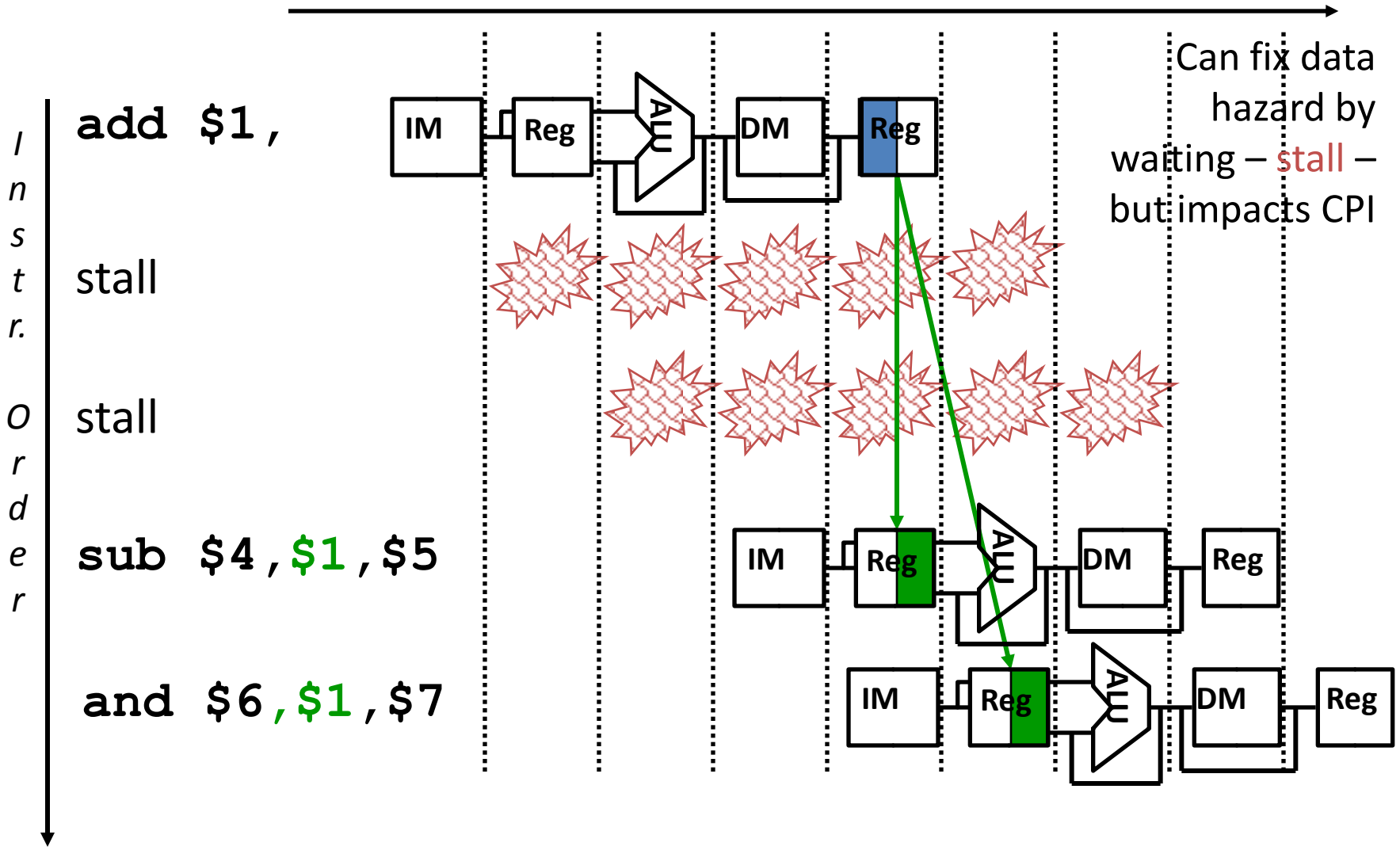


- Load-use data hazard

How About Register File Access?

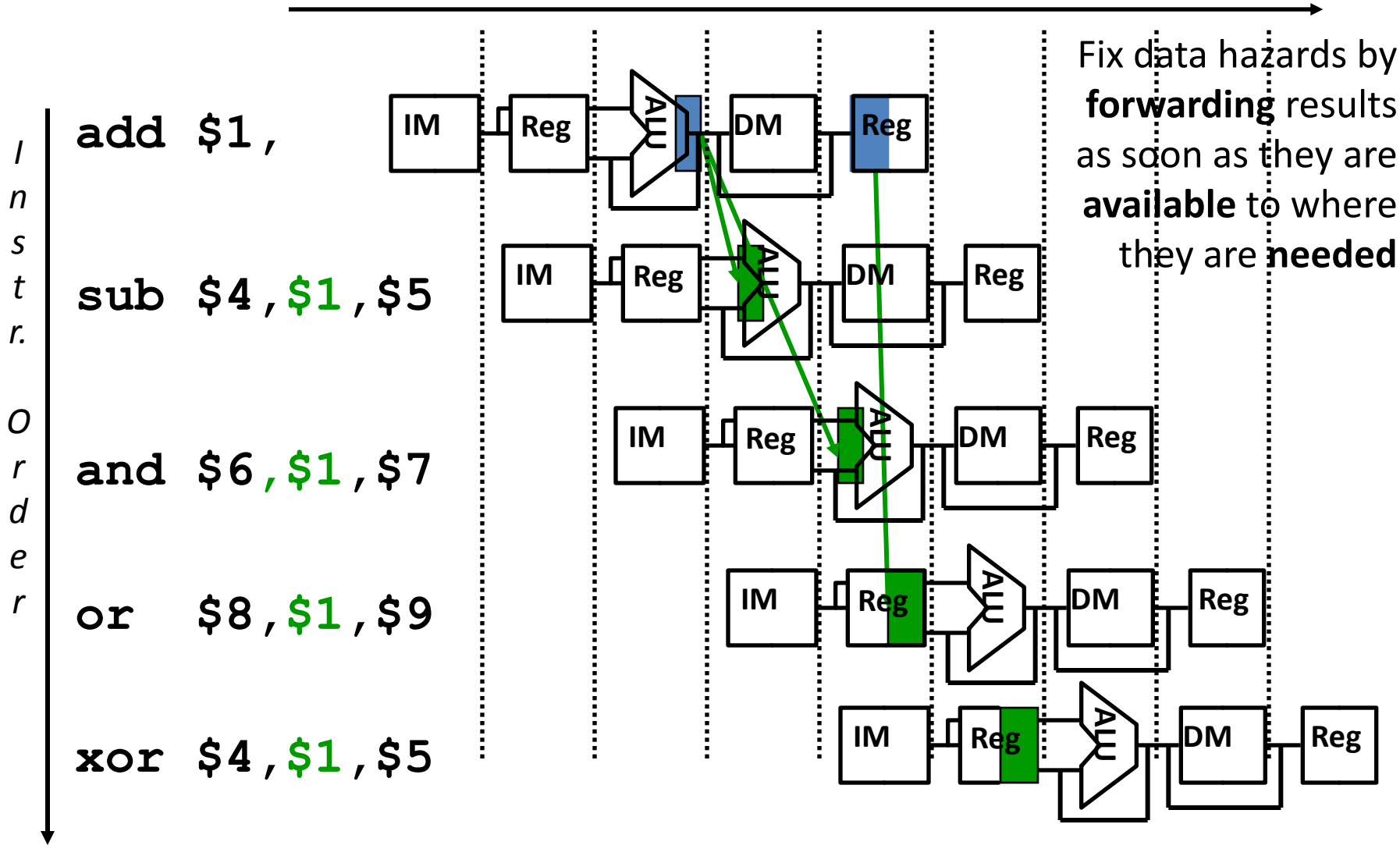


One Way to "Fix" a Data Hazard



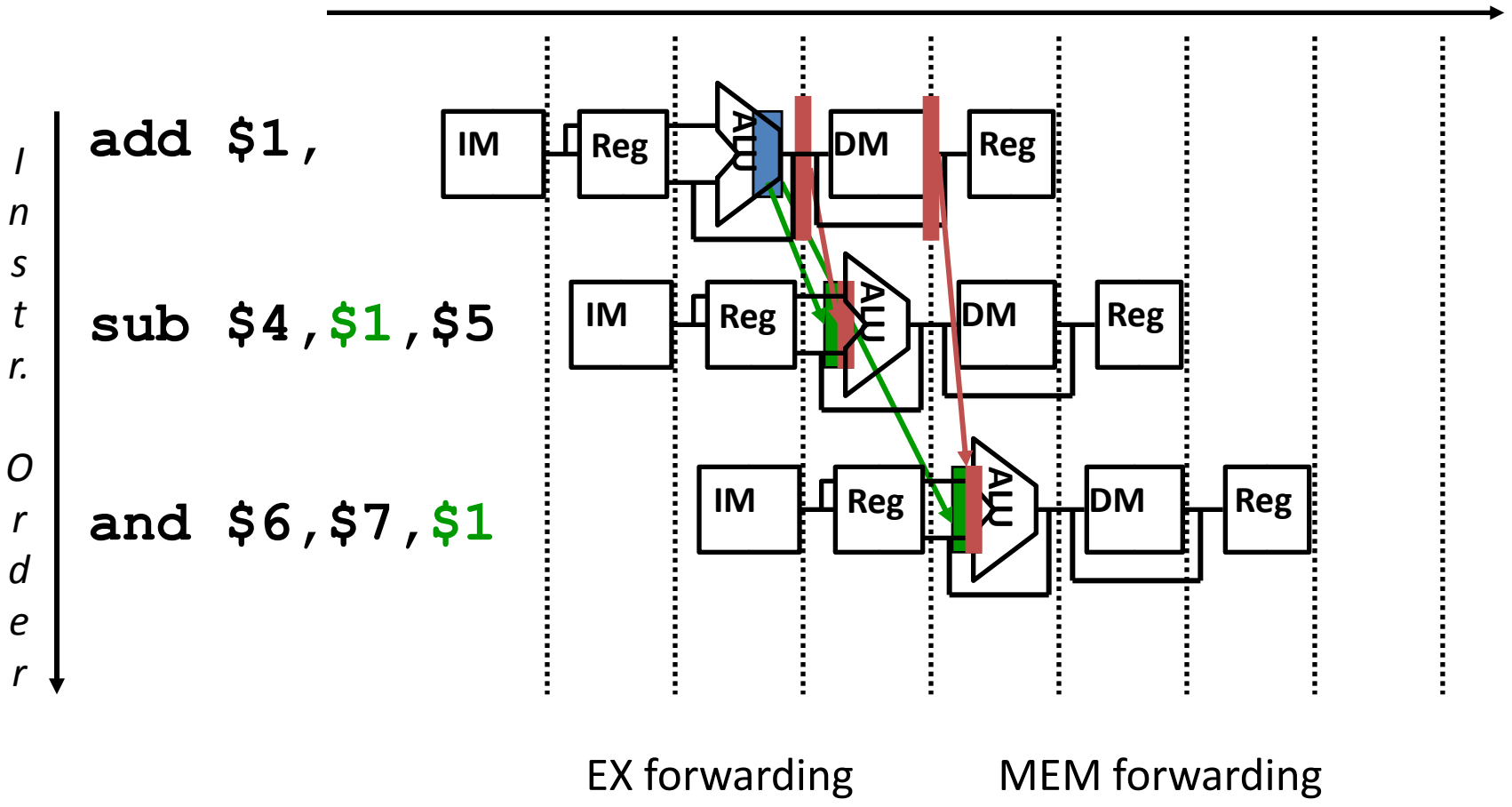
How to implement stall?

Forwarding: Another Way to "Fix" a Data Hazard



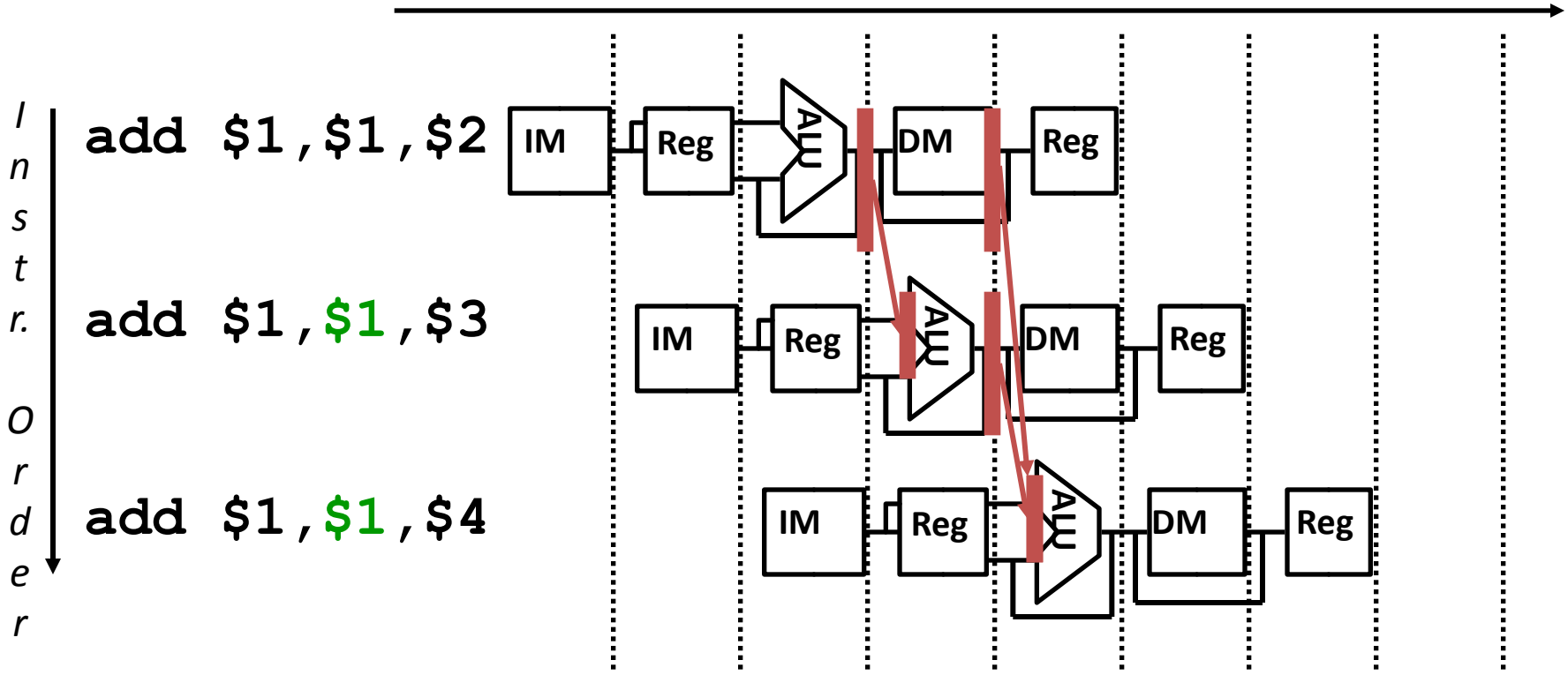
Requires extra connection in a datapath!

Forwarding Illustration



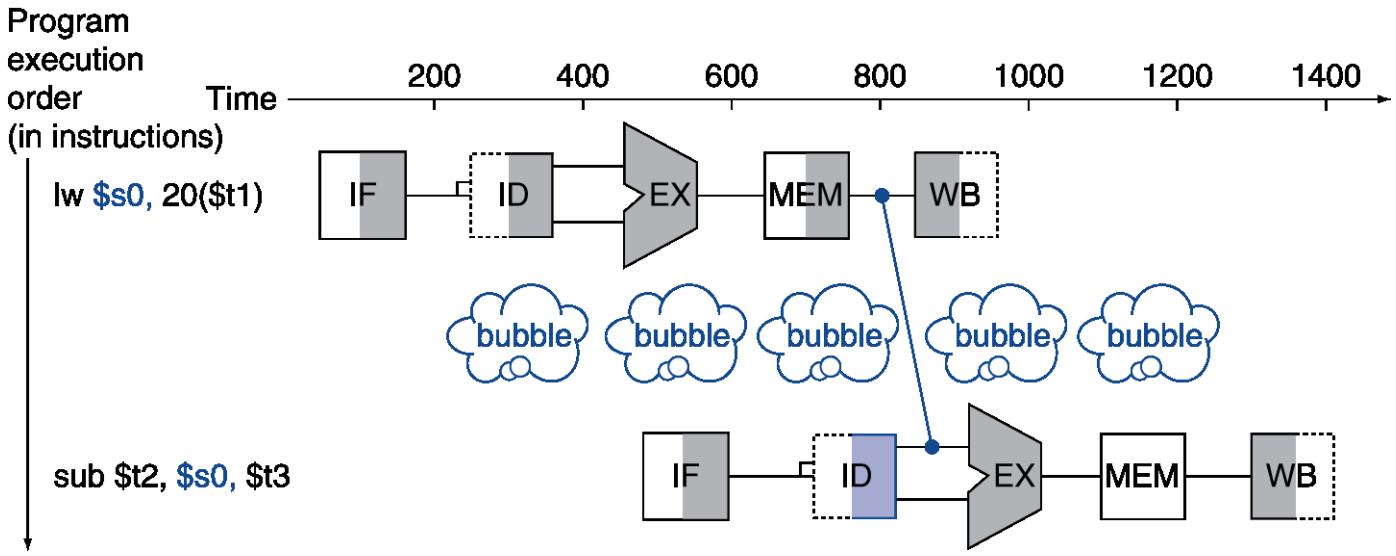
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



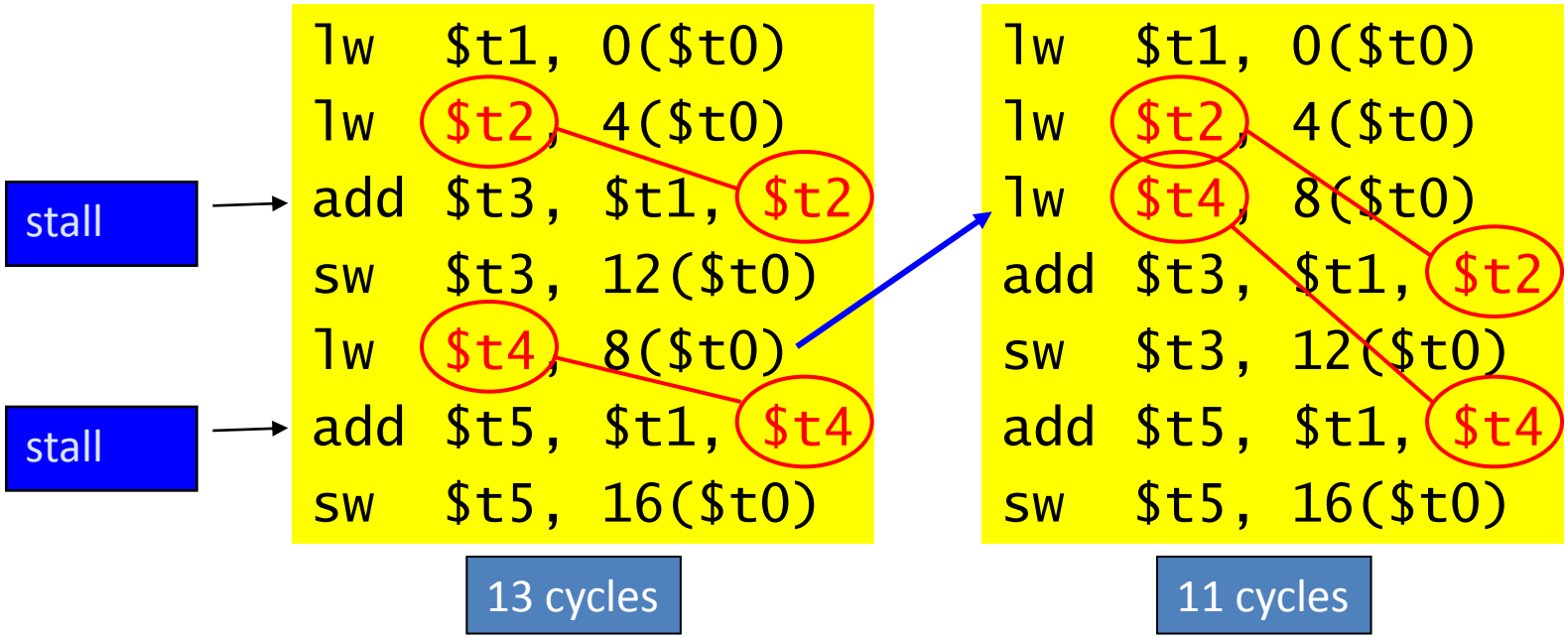
Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



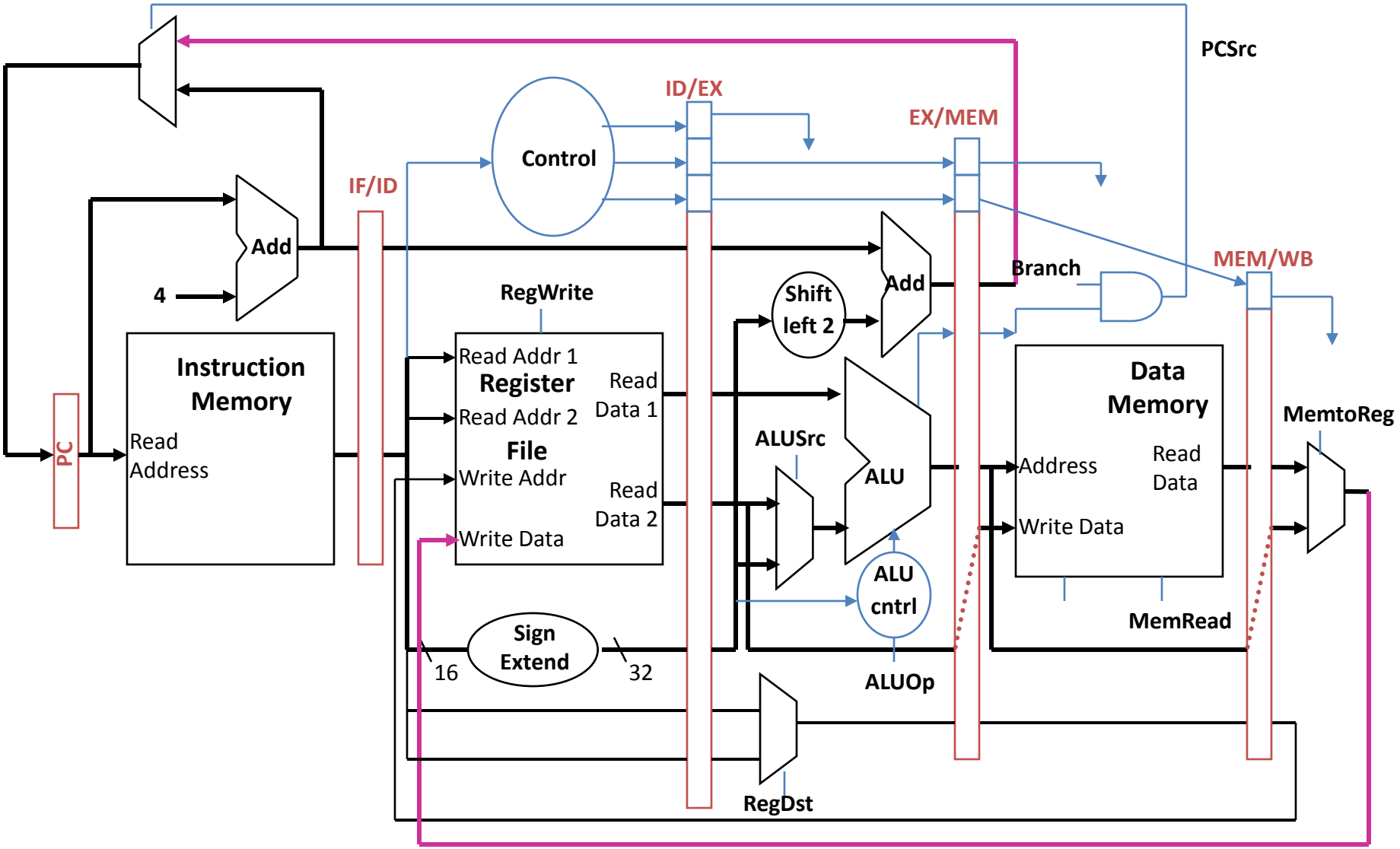
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages

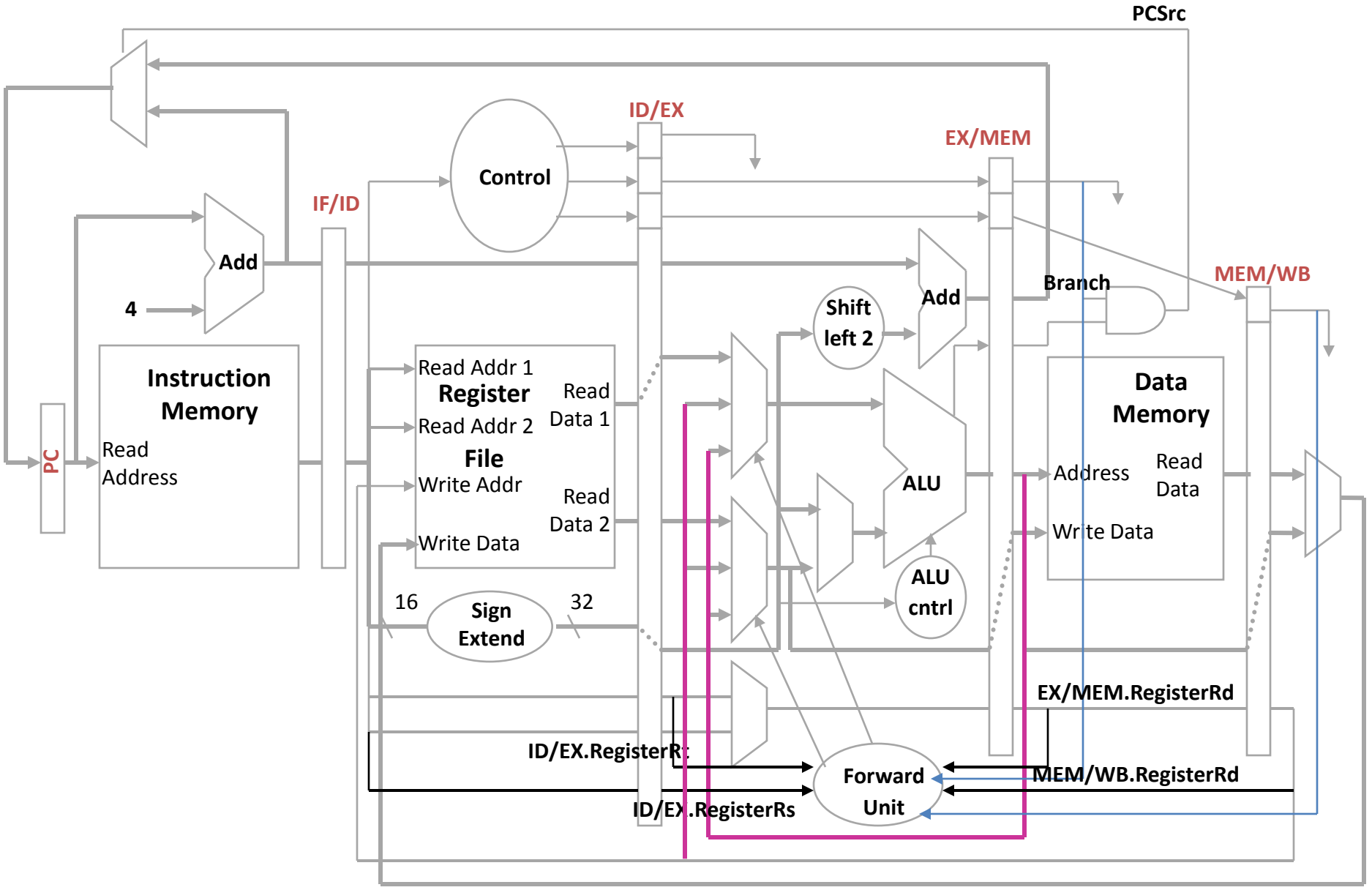


Pipeline Control

- IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)
- ID Stage: no optional control signals to set

	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

Datapath with Forwarding Hardware



Data Forwarding Control Conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

2. MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
      ForwardB = 01
```

Forwards the result from the previous or second previous instr. to either input of the ALU

Load-use Hazard Detection Unit

- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

1. ID Hazard detection Unit:

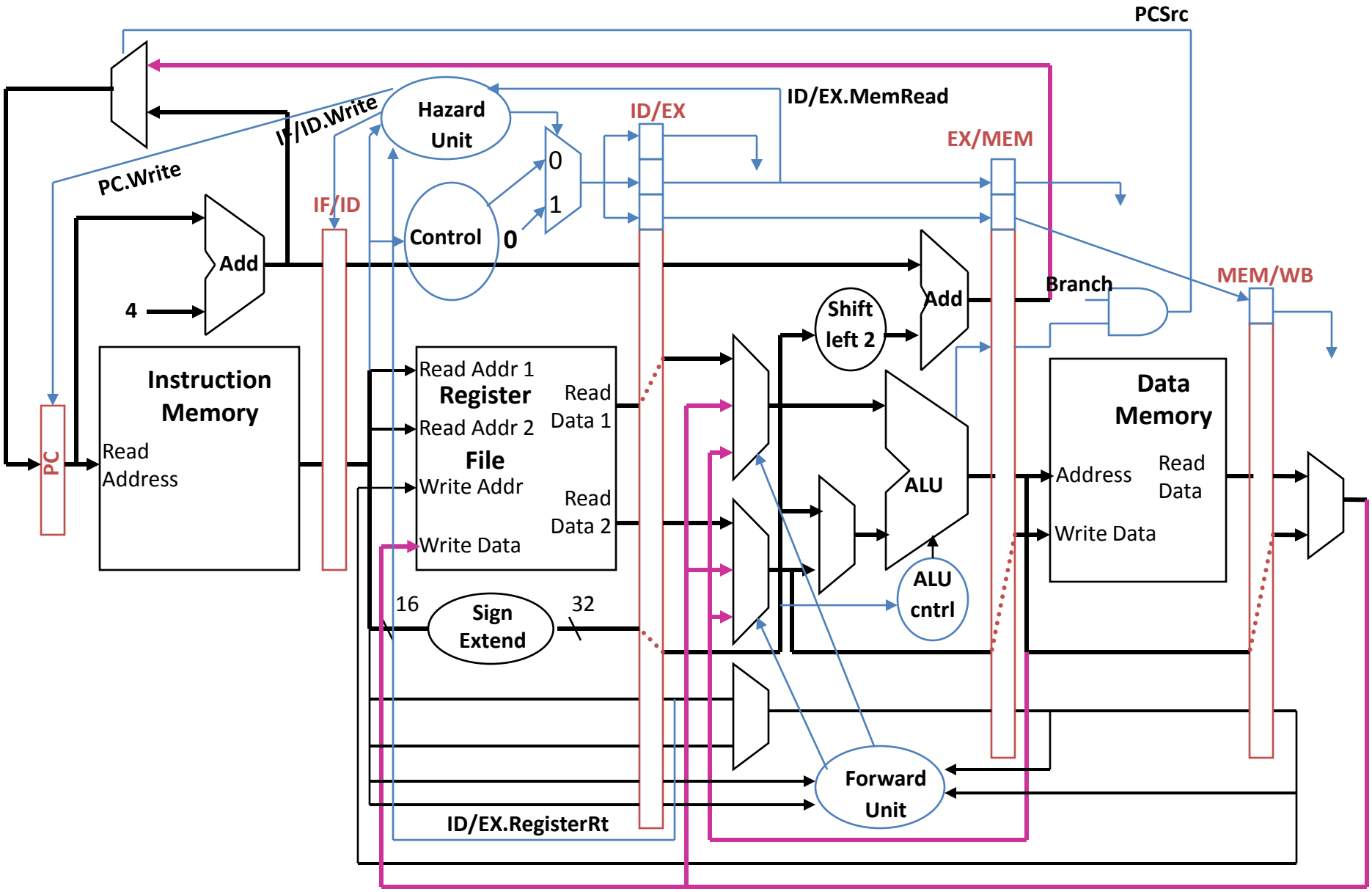
```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

- The first line tests to see if the instruction now in the EX stage is a `lw`; the next two lines check to see if the destination register of the `lw` matches either source register of the instruction in the ID stage (the load-use instruction)
- After this one cycle stall, the forwarding logic can handle the remaining data hazards

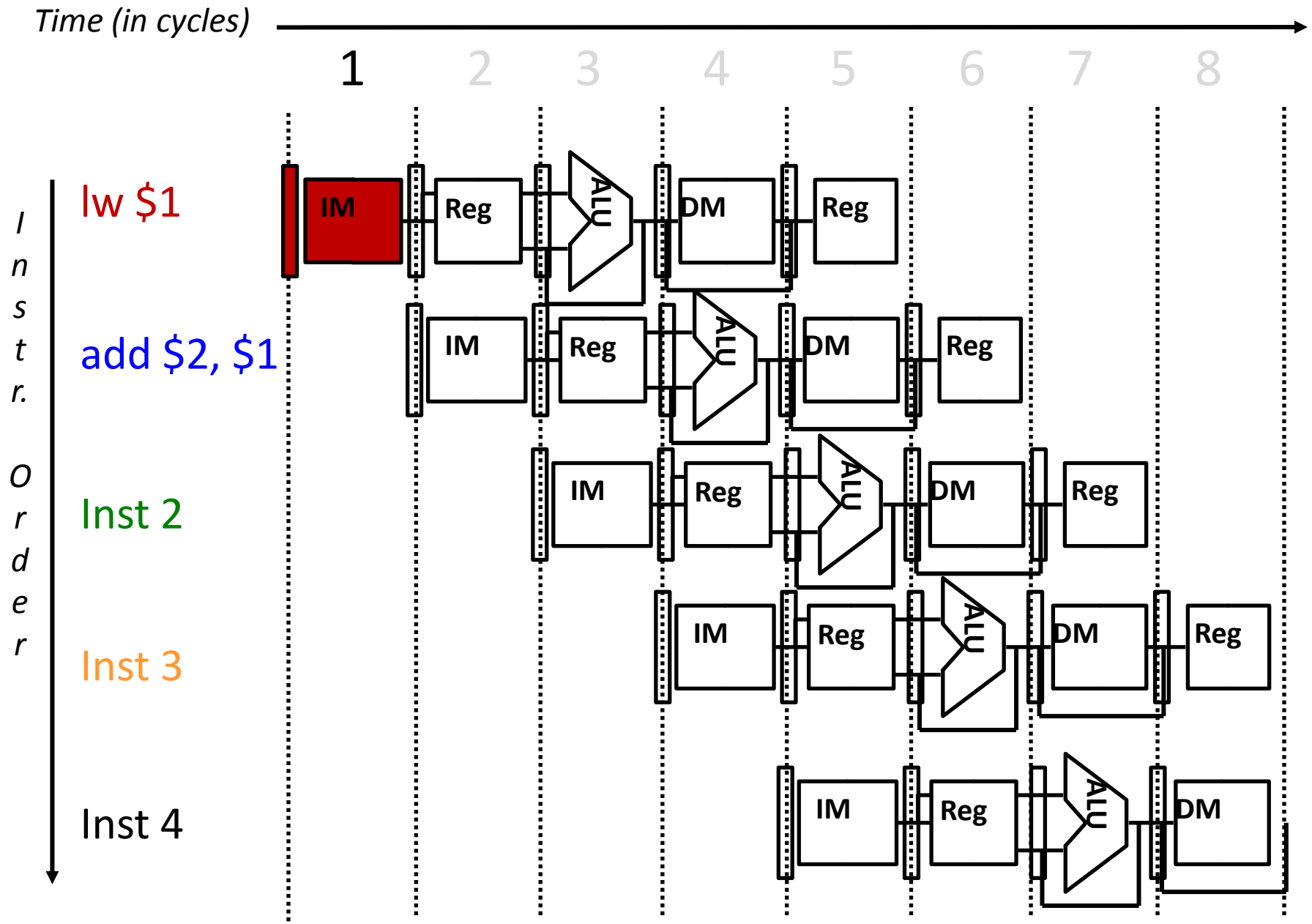
Hazard/Stall Hardware

- Along with the Hazard Unit, we have to implement the stall
- Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
  - Hazard detection Unit controls the writing of the PC (`PC.write`) and IF/ID (`IF/ID.write`) registers
- Insert a "bubble" between the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `nop` in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`nop`). The Hazard Unit controls the mux that chooses between the real control values and the 0's.
- Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

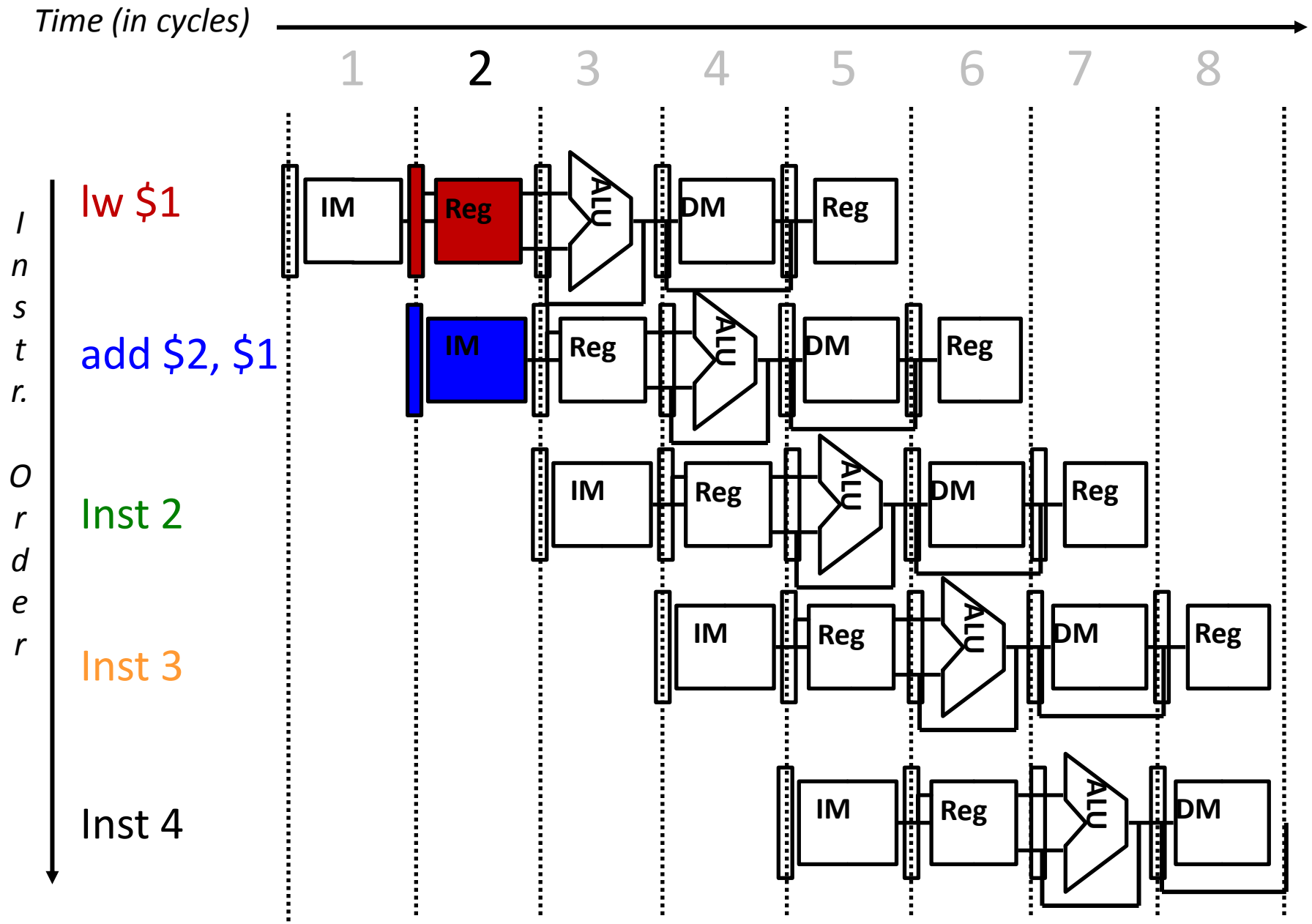
Adding the Hazard/Stall Hardware



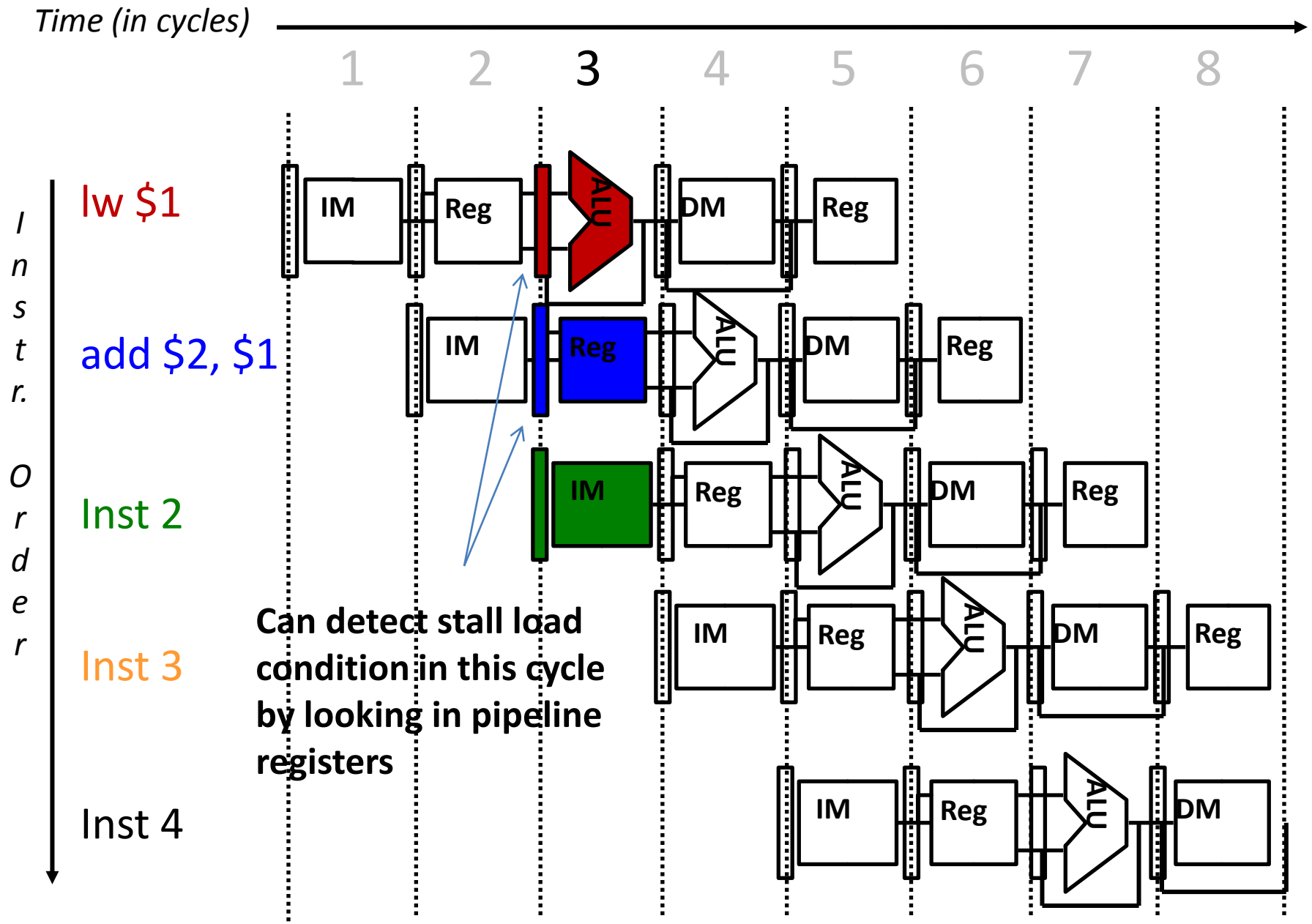
Visualizing Load-Use Stall



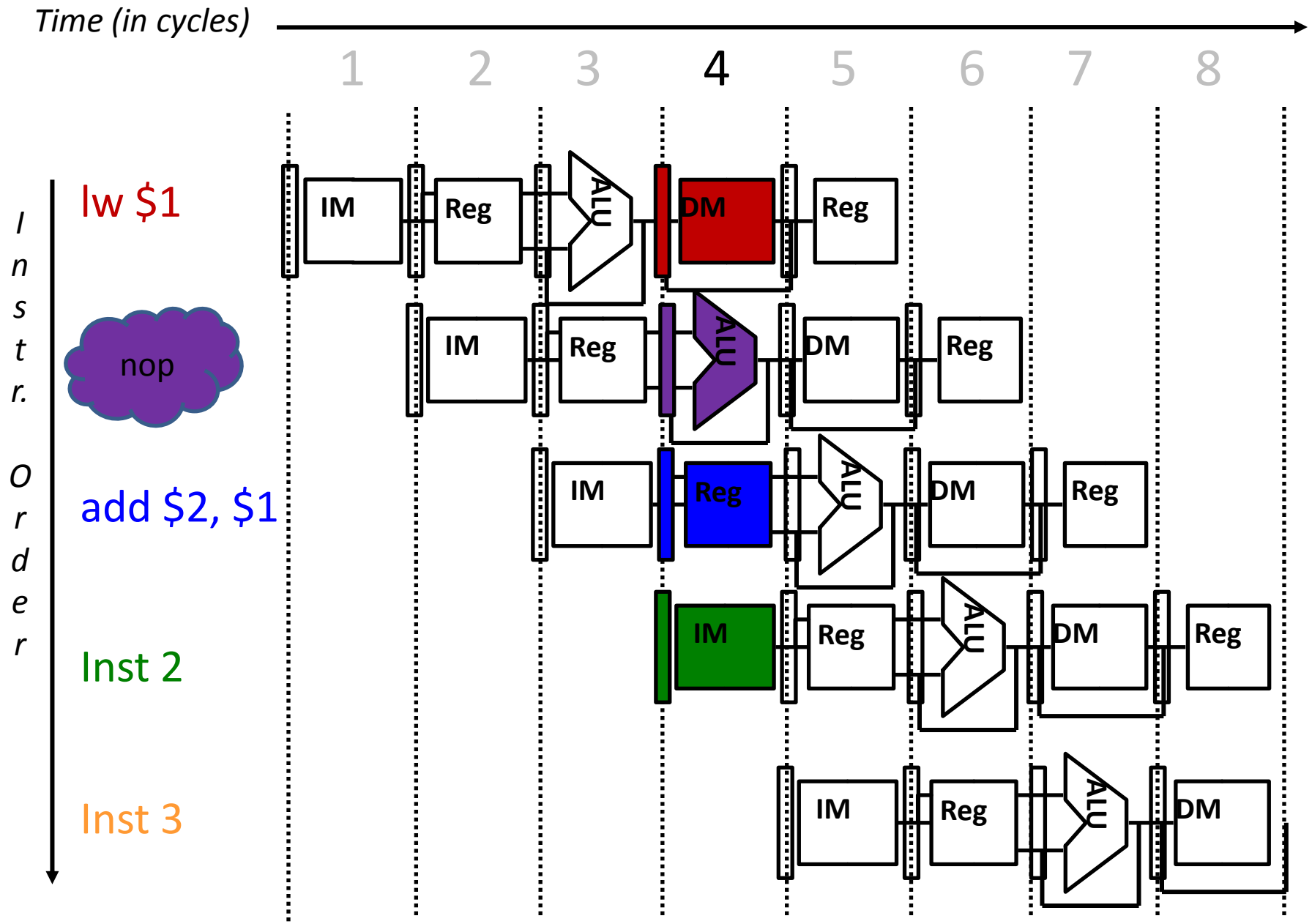
Visualizing Load-Use Stall



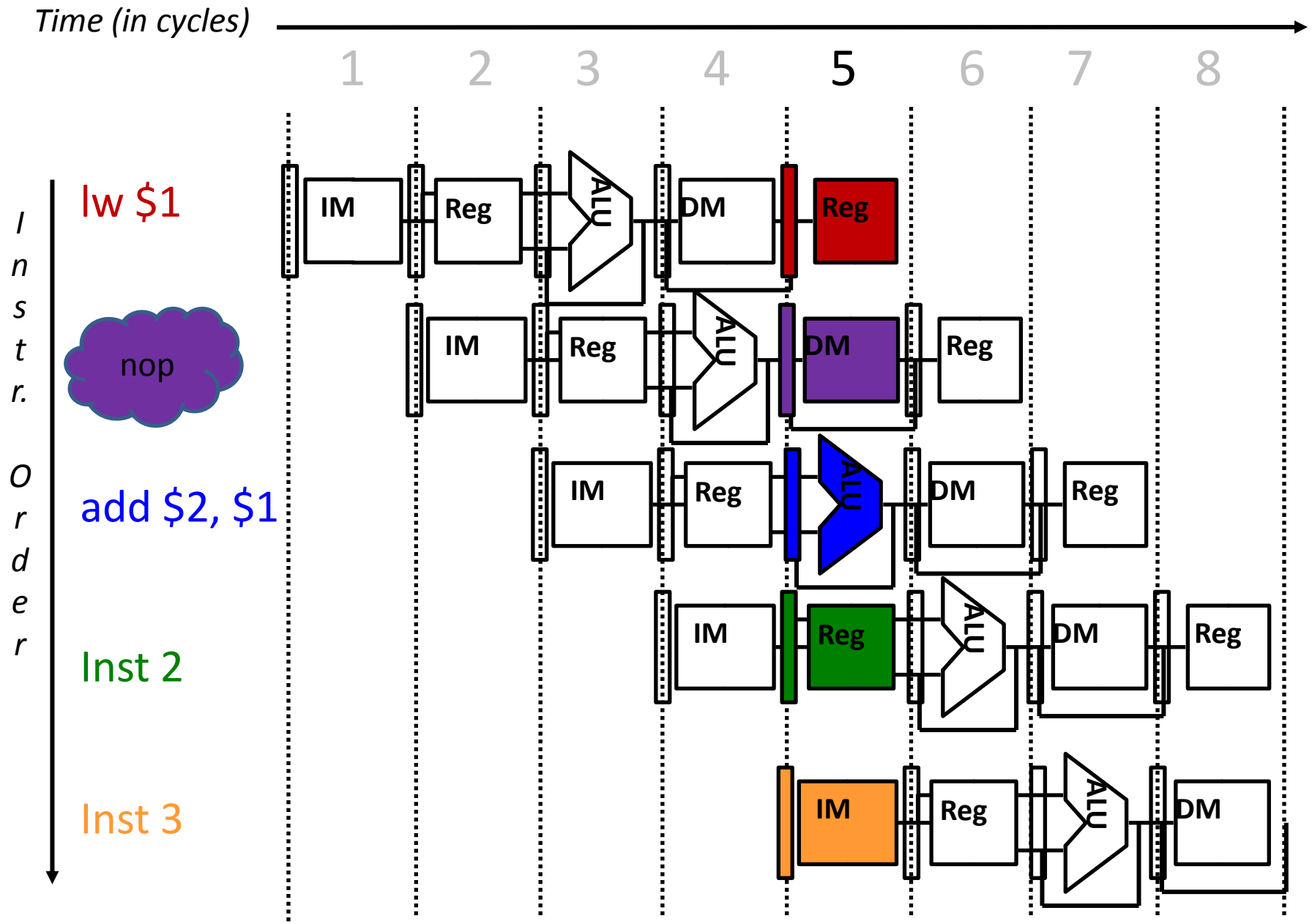
Visualizing Load-Use Stall



Visualizing Load-Use Stall



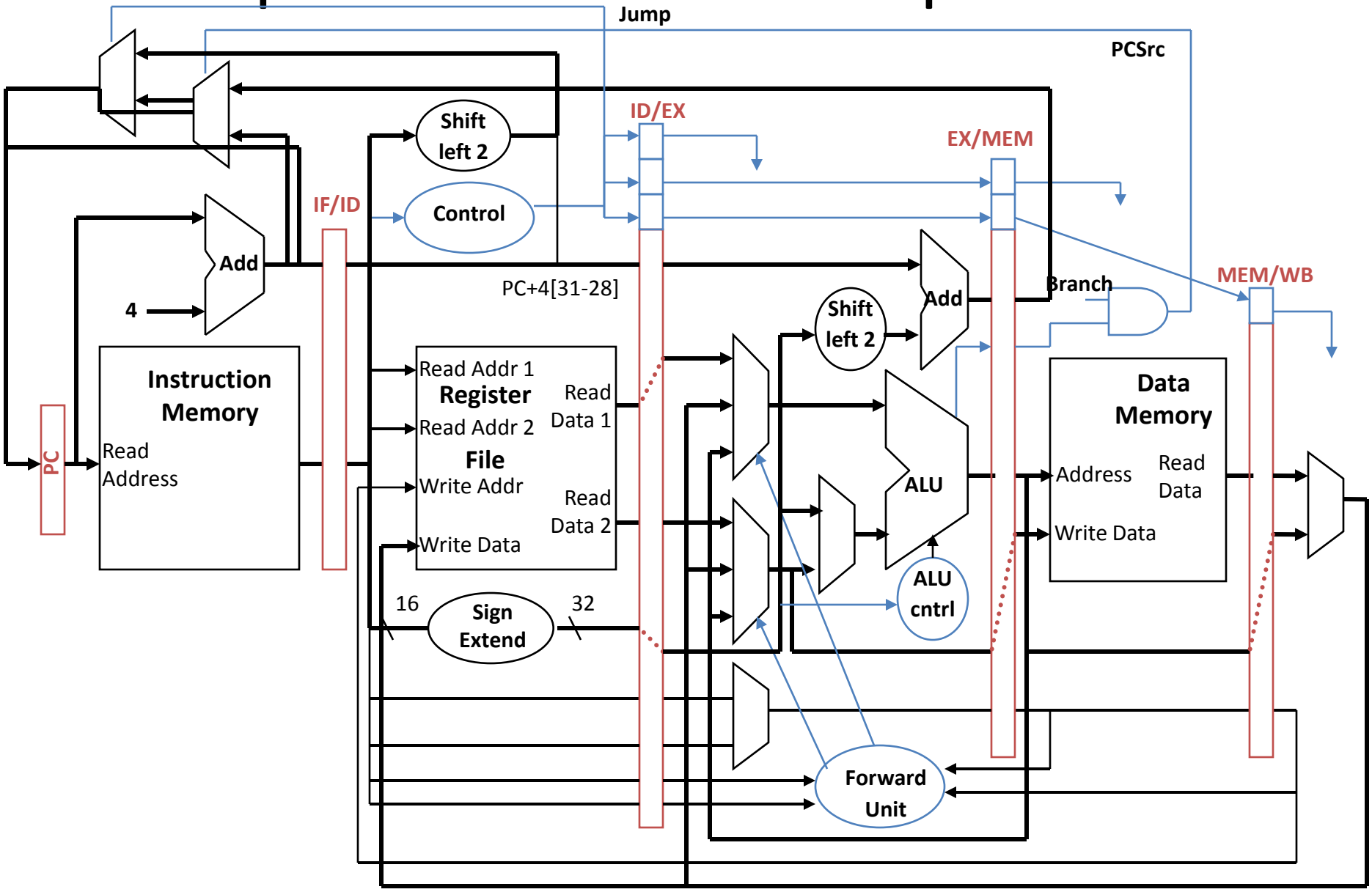
Visualizing Load-Use Stall



Control Hazards

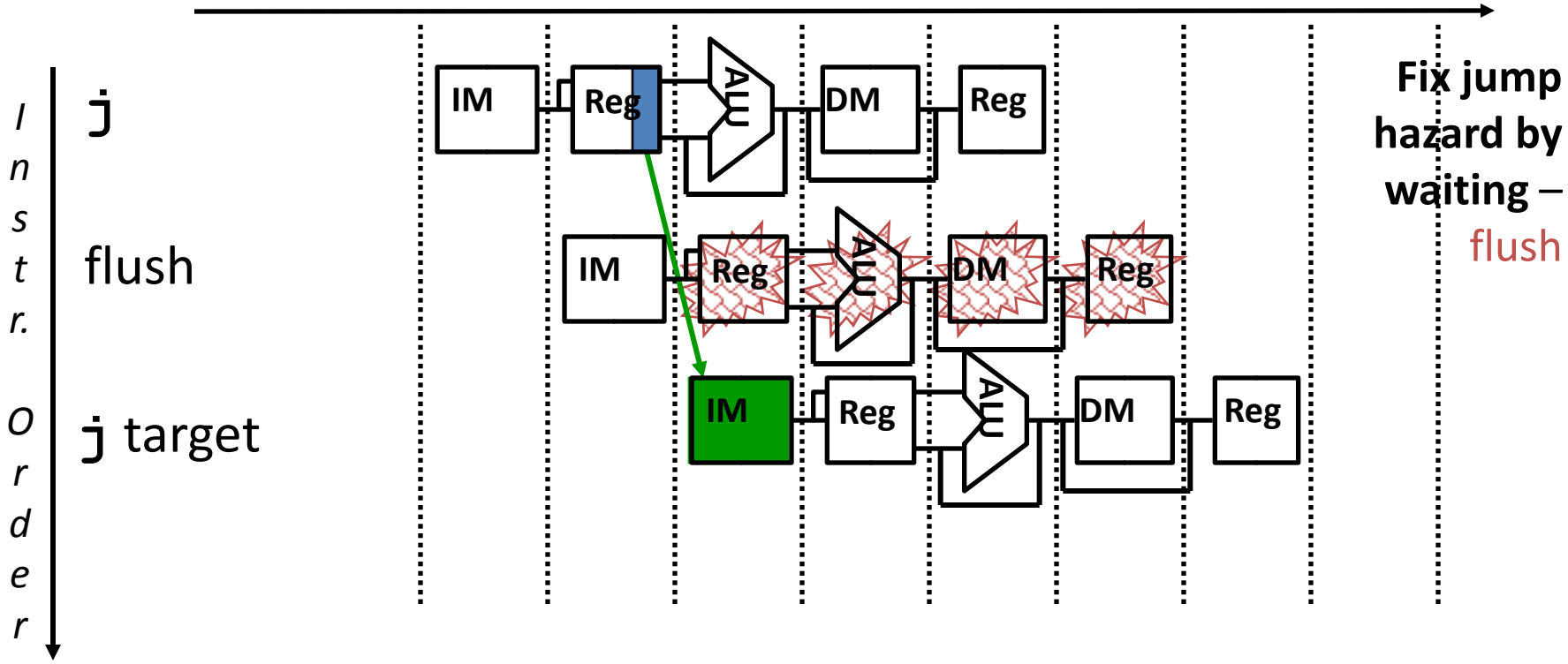
- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (`j, jal, jr`)
  - Conditional branches (`beq`, `bne`)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !
- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Datapath Branch and Jump Hardware



Jumps Incur One Stall

- Jumps not decoded until ID, so one flush is needed
  - To flush, set `IF.Flush` to zero the instruction field of the IF/ID pipeline register (turning it into a `noop`)

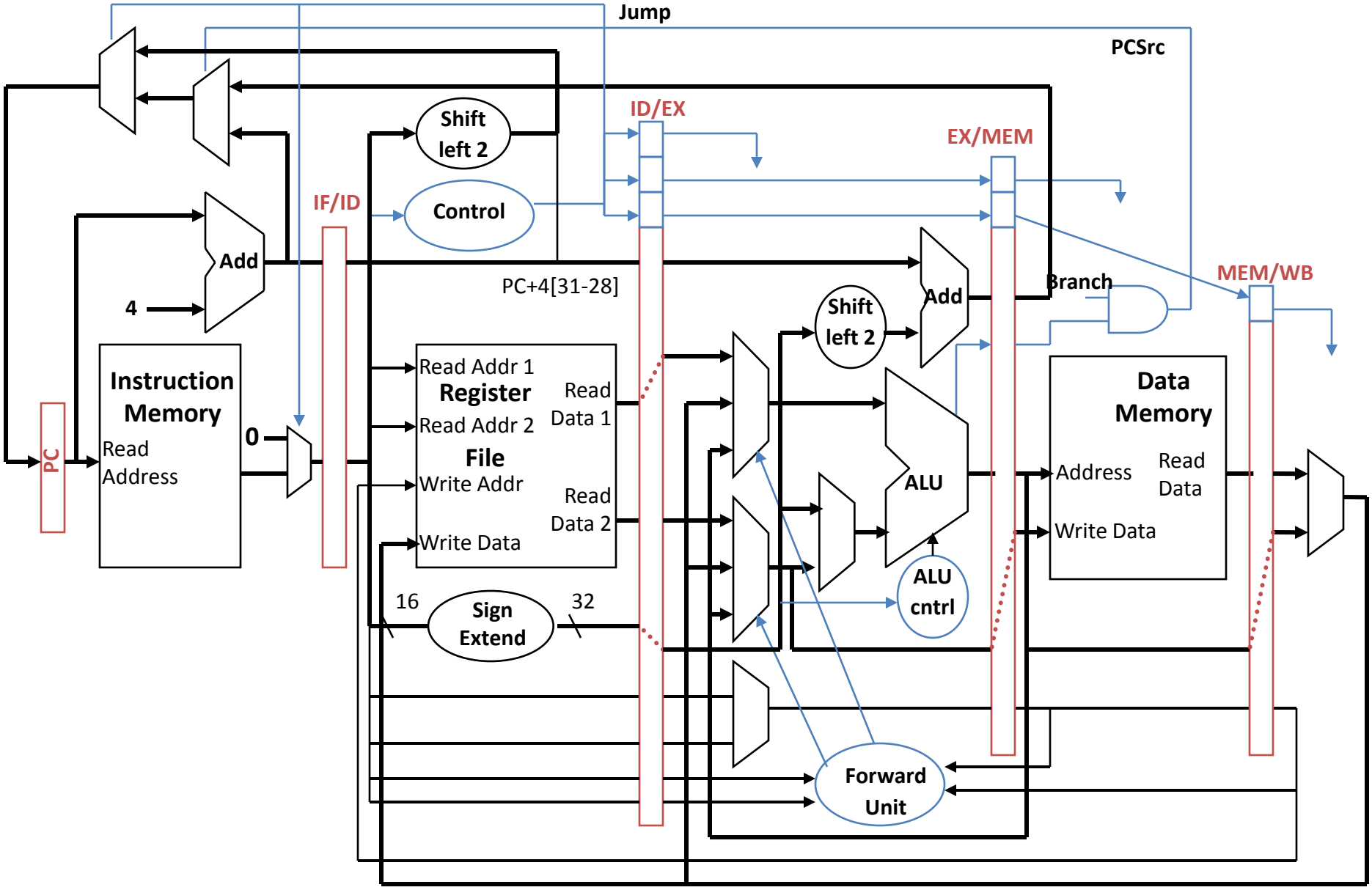


- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

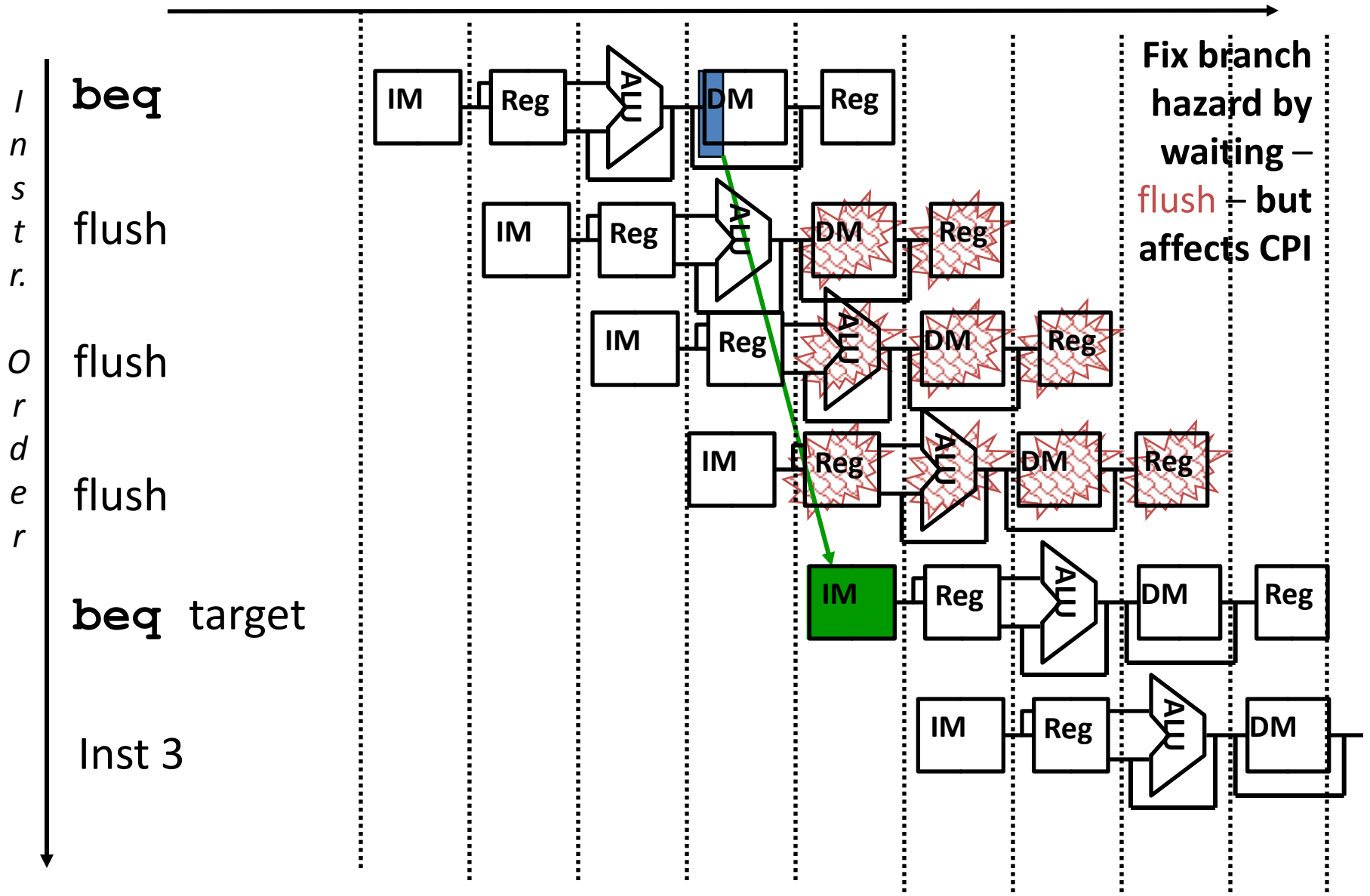
Two “Types” of Stalls

- `Nop` instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert `nop` by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- Flushes (or instruction squashing) were an instruction in the pipeline is replaced with a `nop` instruction (as done for instructions located sequentially after `j` instructions)
  - Zero the control bits for the instruction to be flushed

Supporting ID Stage Jumps



One Way to "Fix" a Branch Control Hazard

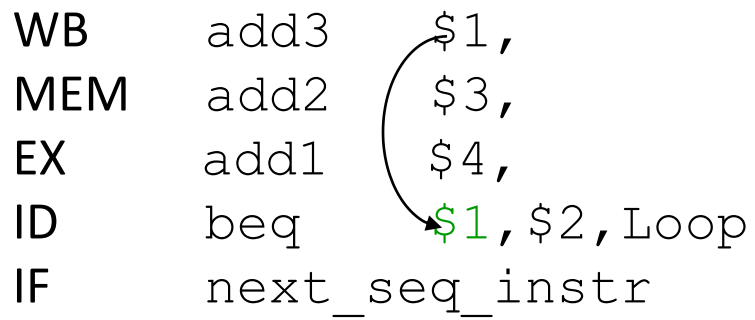


Reducing the Delay of Branches

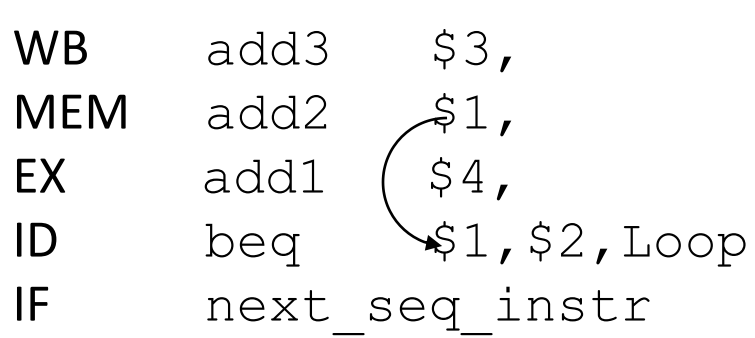
- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall (flush) cycles to two
  - Adds an `and` gate and a 2x1 `mux` to the EX timing path
- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall (flush) cycles to one (like with jumps)
    - But now need to add forwarding hardware in ID stage
  - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  - Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an `and` gate to the ID timing path
- For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

ID Branch Forwarding Issues

- MEM/WB "forwarding" is taken care of by the normal RegFile write before read operation



- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like

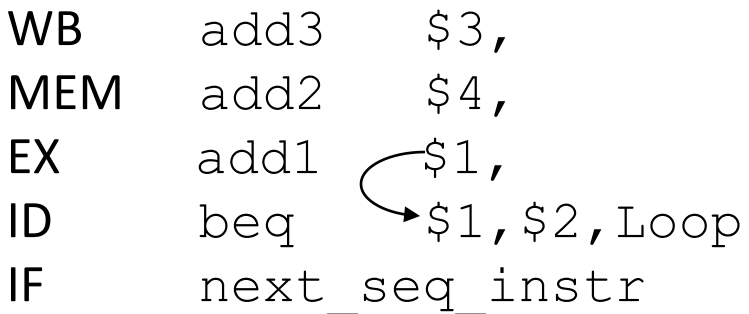



```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
       ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
       ForwardD = 1
```

Forwards the result from the second previous instr. to either input of the compare

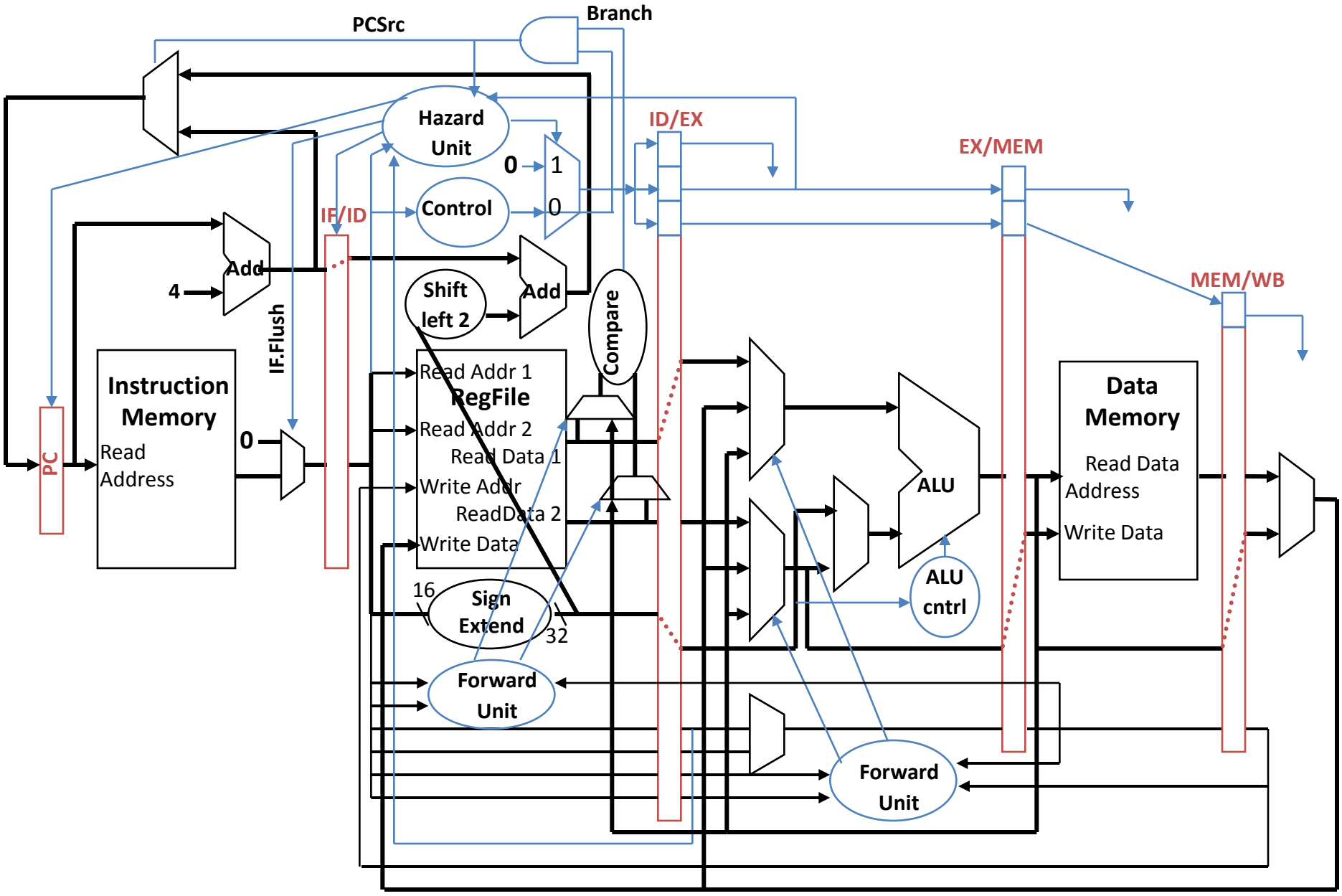
ID Branch Forwarding Issues, con't

- If the instruction immediately before the branch produces one of the branch source operands, then a stall needs to be inserted (between the `beq` and `add1`) since the EX stage ALU operation is occurring at the *same time* as the ID stage branch compare operation



  - "Bounce" the `beq` (in ID) and next_seq_instr (in IF) in place (ID Hazard Unit deasserts `PC.Write` and `IF/ID.Write`)
  - Insert a stall between the `add` in the EX stage and the `beq` in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)
- If the branch is found to be taken, then flush the instruction currently in IF (`IF.Flush`)

Supporting ID Stage Branches

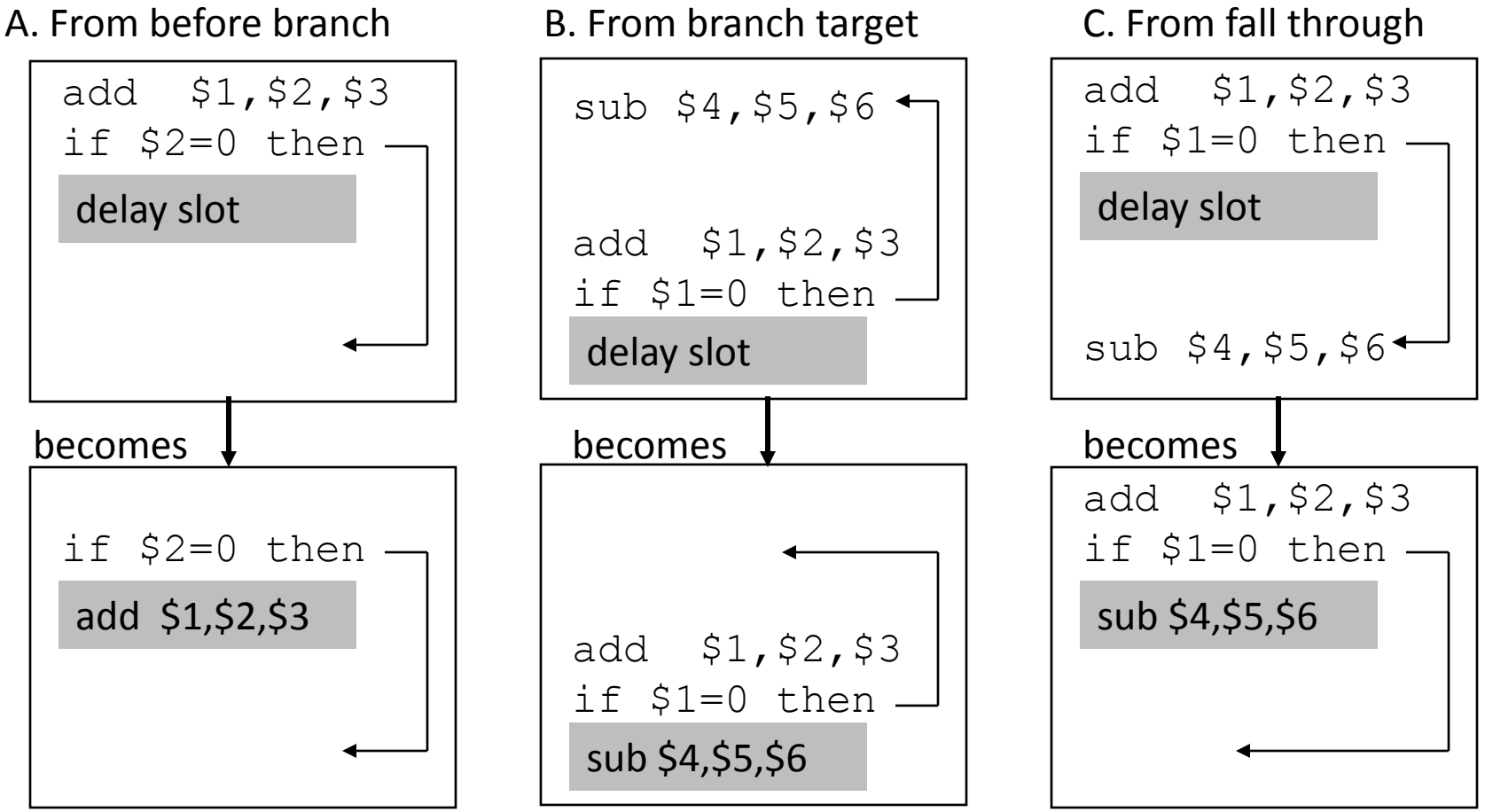


Delayed Branches

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
  - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay

- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots



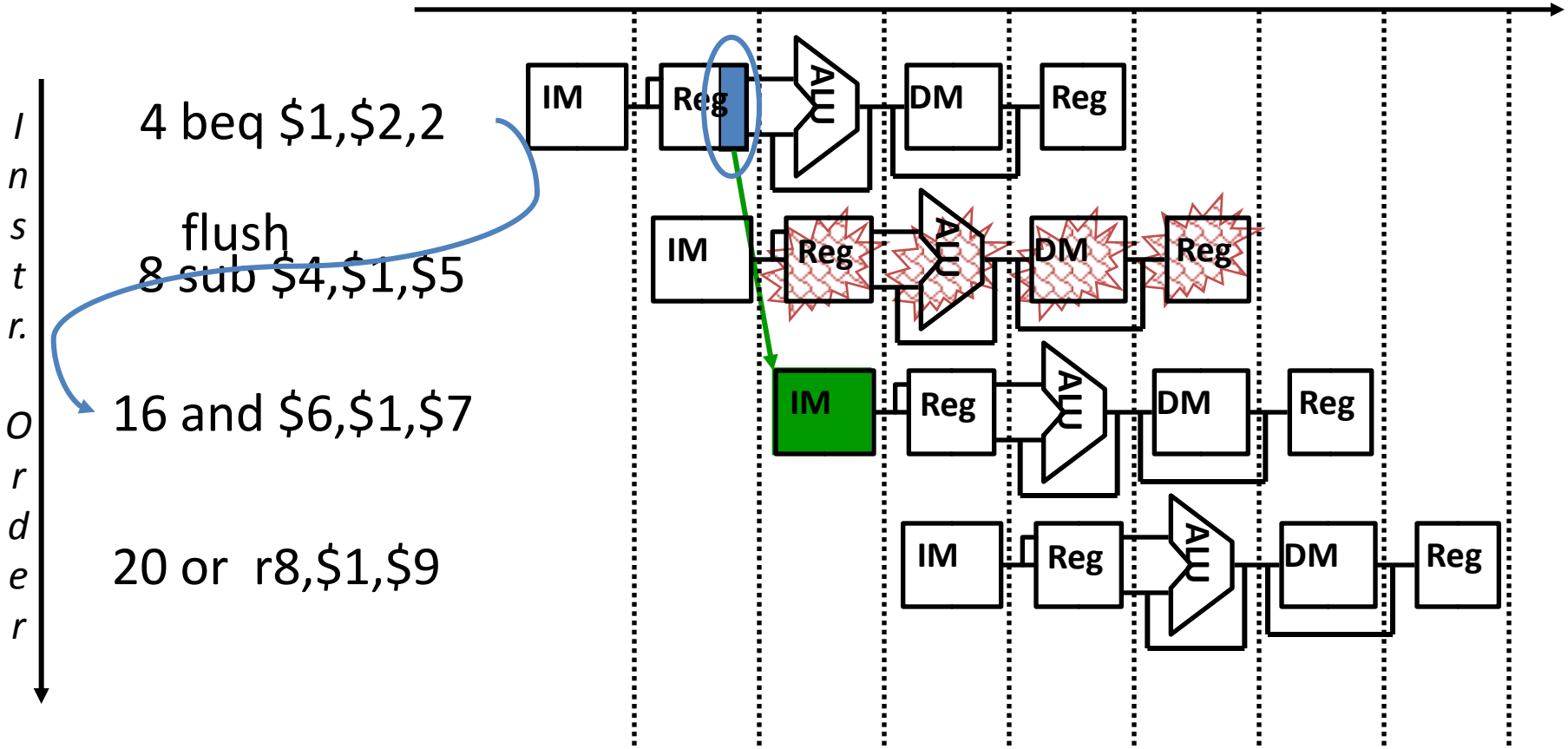
- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
   - If taken, flush instructions after the branch (earlier in the pipeline)
     - in IF, ID, and EX stages if branch logic in MEM – three stalls
     - In IF and ID stages if branch logic in EX – two stalls
     - in IF stage if branch logic in ID – one stall
   - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
   - restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, assert `IF.Flush` to zero the instruction field of the IF/ID pipeline register (transforming it into a `noop`)

Branching Structures

- Predict not taken works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
            .
            .
            .
      last loop instr
      j  Loop
Out:  fall out instr
```

- Predict not taken doesn't work well for "bottom of the loop" branching structures

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

Static Branch Prediction, con't

- Resolve branch hazards by assuming a given outcome and proceeding

2. Predict taken – predict branches will always be taken
   - Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
   - Is there a way to "cache" the address of the branch target instruction ??

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution

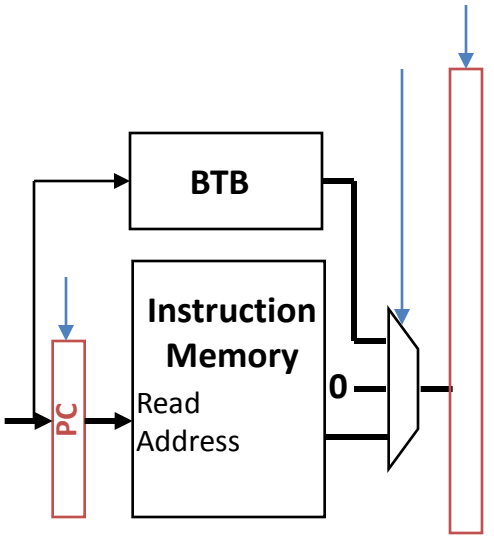
3. Dynamic branch prediction – predict branches at run-time using *run-time* information

Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
    - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

Branch Target Buffer

- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
    - Would need a two read port instruction memory
    - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



- If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

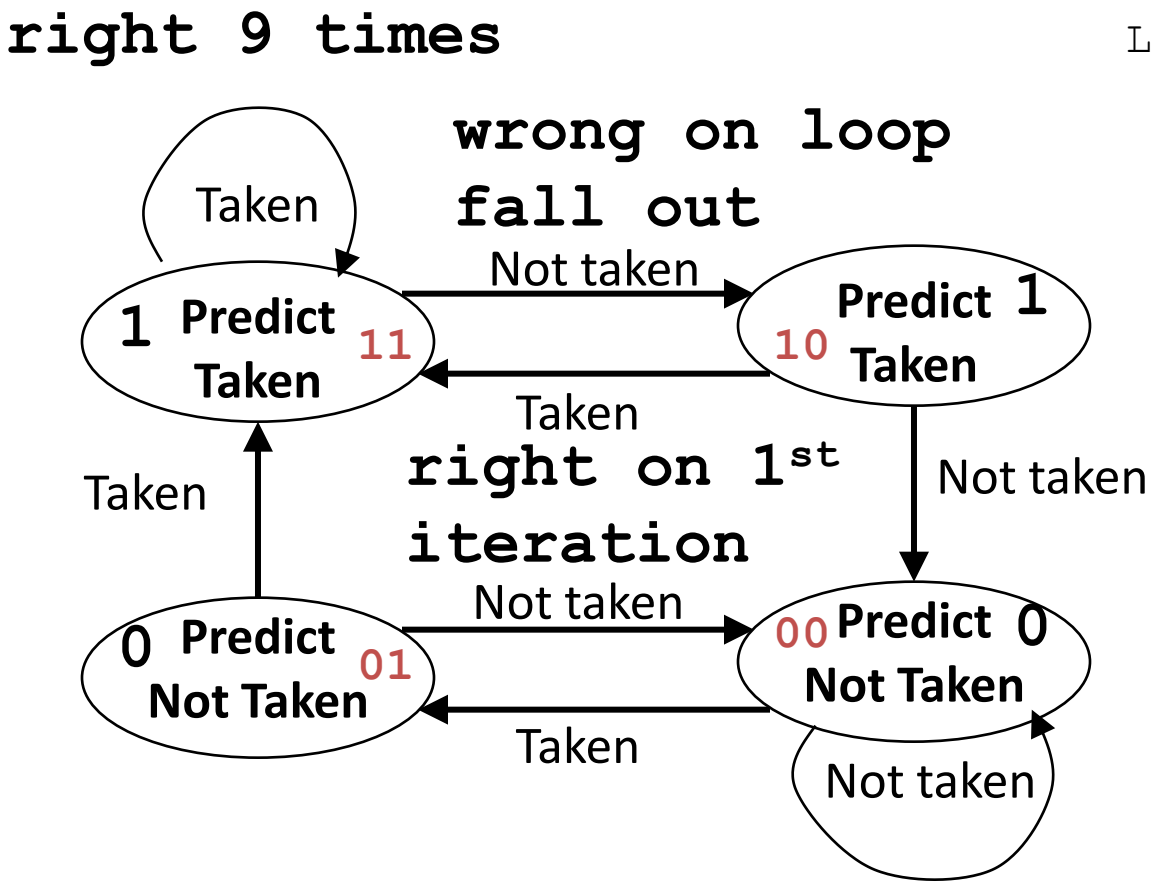
- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- BHT also stores the initial FSM state

Dealing with Exceptions

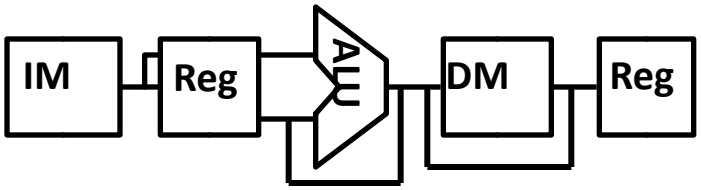
- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- The software (OS) looks at the cause of the exception and “deals” with it

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
  - simply suspend and resume user program

- Traps (Exception) – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so much stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

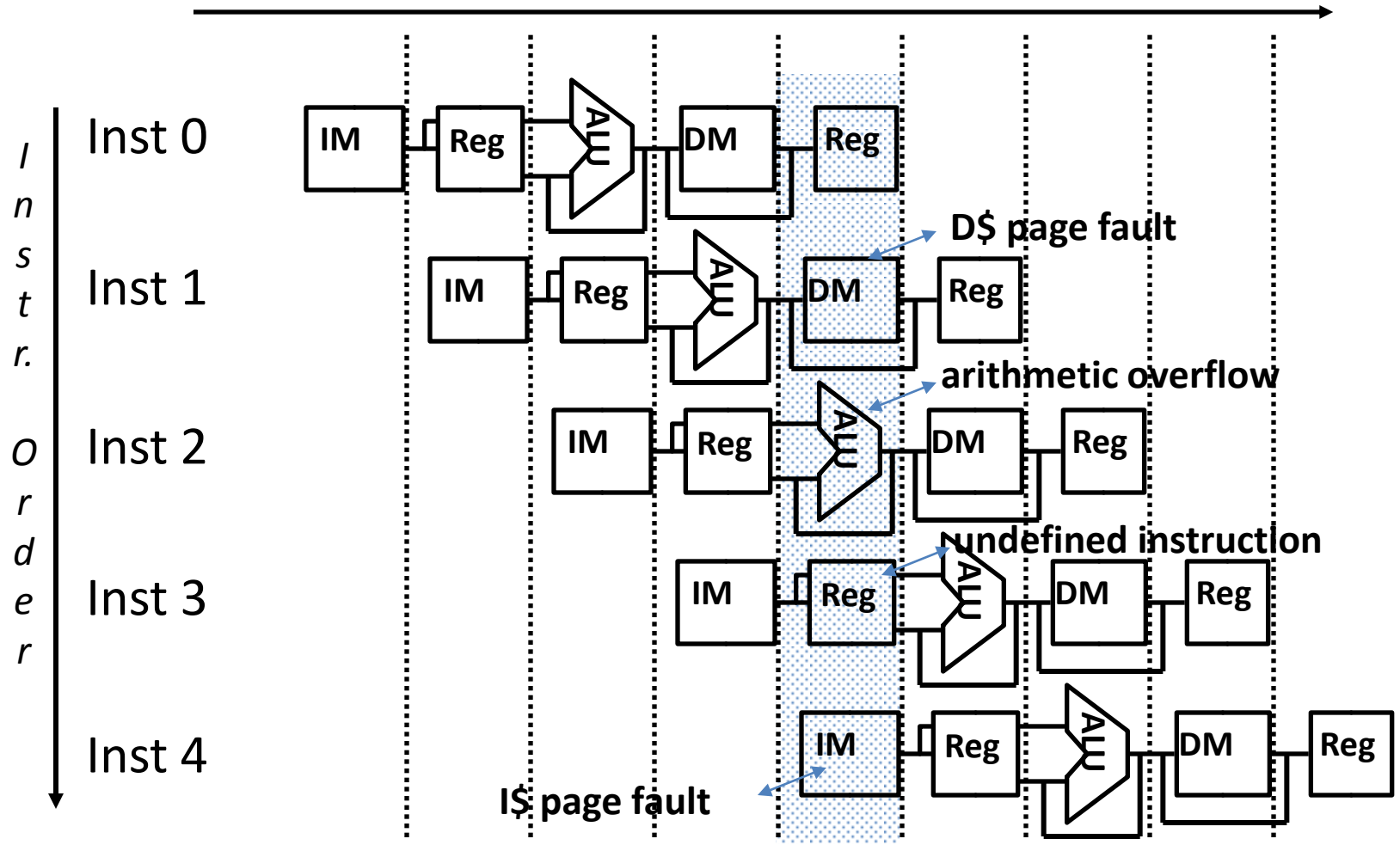
Where in the Pipeline Exceptions Occur



	Stage(s)?	Synchronous?
• Arithmetic overflow	EX	yes
• Undefined instruction	ID	yes
• TLB or page fault	IF, MEM	yes
• I/O service request	any	no
• Hardware malfunction	any	no

- Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Multiple Simultaneous Exceptions

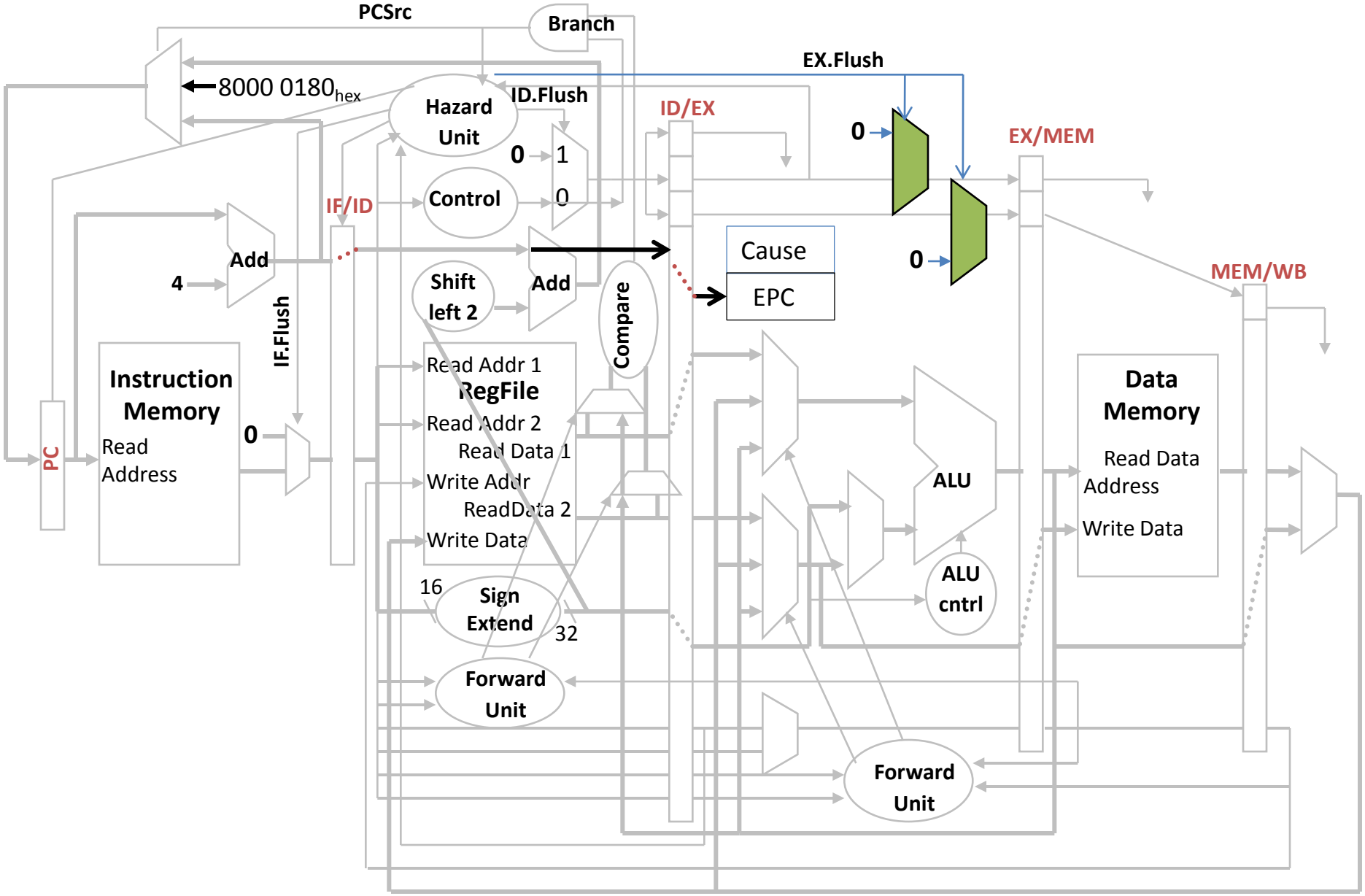


- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

Additions to MIPS to Handle Exceptions (Fig 6.42)

- Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (`CauseWrite`)
- EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (`EPCWrite`)
  - Exception software must match exception to instruction
- A way to load the PC with the address of the exception handler
  - Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., $8000\ 0180_{hex}$ for arithmetic overflow)
- A way to flush offending instruction and the ones that follow it

Datapath with Controls for Exceptions



Stalling vs. Flushing Example

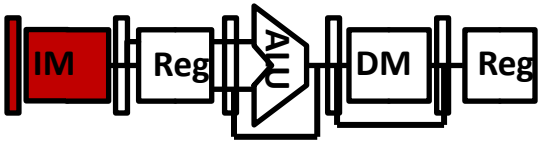
Stall here →

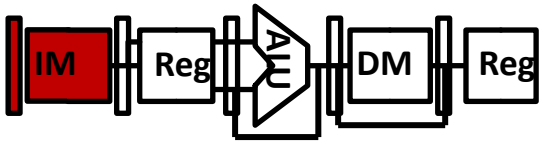
```
Inst1:  lw   $1, 0($2)
Inst2:  add  $2, $1, $1
Inst3:  add  $3, $2, $1
Inst4:  bne  $1,  $1, label
Inst5:  and  $1, $2, $3
Inst6:  or   $1, $1, $1
```

Flush here → (assuming no delay slot)

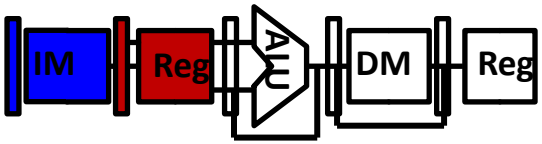
```
Inst1:  j    Inst4
Inst2:  add  $2, $1, $1
Inst3:  add  $3, $2, $1
Inst4:  bne  $1,  $1, label
Inst5:  and  $1, $2, $3
Inst6:  or   $1, $1, $1
```

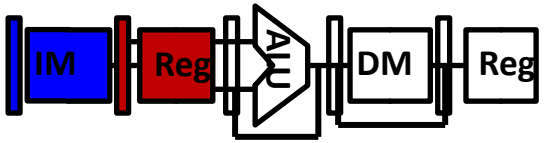
Cycle 1





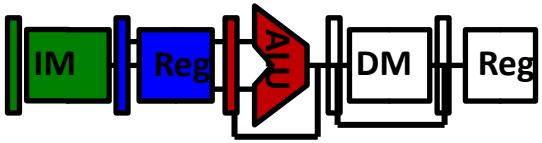
Cycle 2

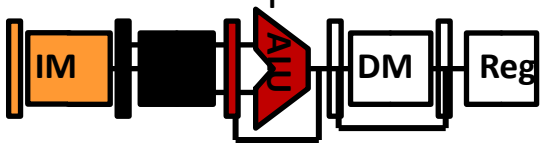




inst2→nop

Cycle 3

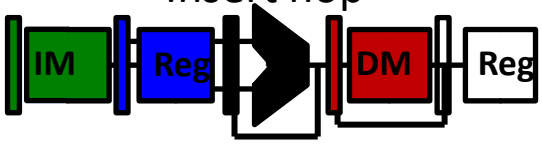


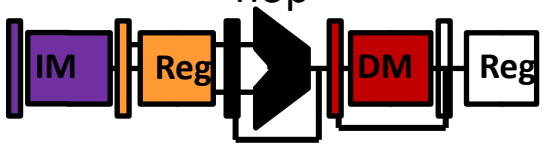


Insert nop

nop

Cycle 4

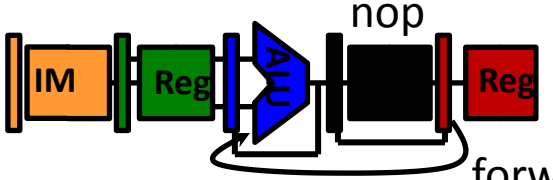


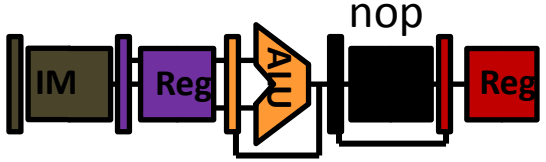


nop

nop

Cycle 5





forwarding

Stalling vs. Flushing Example

Stall here →

```
Inst1:  lw    $1, 0($2)
Inst2:  add   $2, $1, $1
Inst3:  add   $3, $2, $1
Inst4:  bne   $1,  $1, label
Inst5:  and   $1, $2, $3
Inst6:  or    $1, $1, $1
```

Flush here → (assuming no delay slot)

```
Inst1:  j     Inst4
Inst2:  add   $2, $1, $1
Inst3:  add   $3, $2, $1
Inst4:  bne   $1,  $1, label
Inst5:  and   $1, $2, $3
Inst6:  or    $1, $1, $1
```

cycle

	1	2	3	4	5	6	7	8	9	10	11
Instr1	IF	ID	EX	M	W						
Instr2		IF	ID	ID	EX	M	W				
Instr3			IF	IF	ID	EX	M	W			
Instr4					IF	ID	EX	M	W		
Instr5						IF	ID	EX	M	W	
Instr6							IF	ID	EX	M	W

cycle

	1	2	3	4	5	6	7	8	9	10	11
Instr1	IF	ID	EX	M	W						
Instr2		IF									
Instr3											
Instr4			IF	ID	EX	M	W				
Instr5				IF	ID	EX	M	W			
Instr6					IF	ID	EX	M	W		

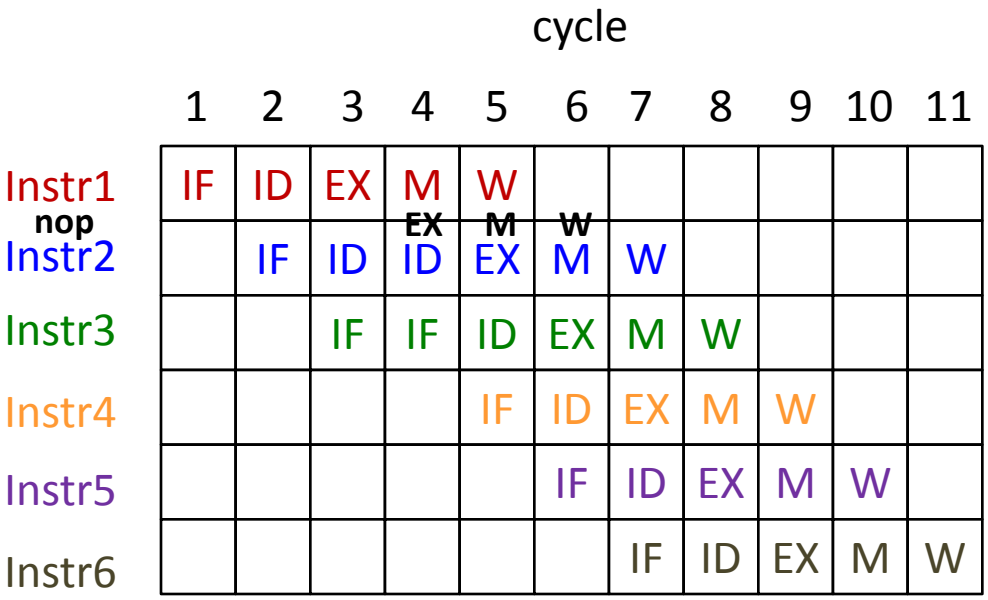
Stalling vs. Flushing Example

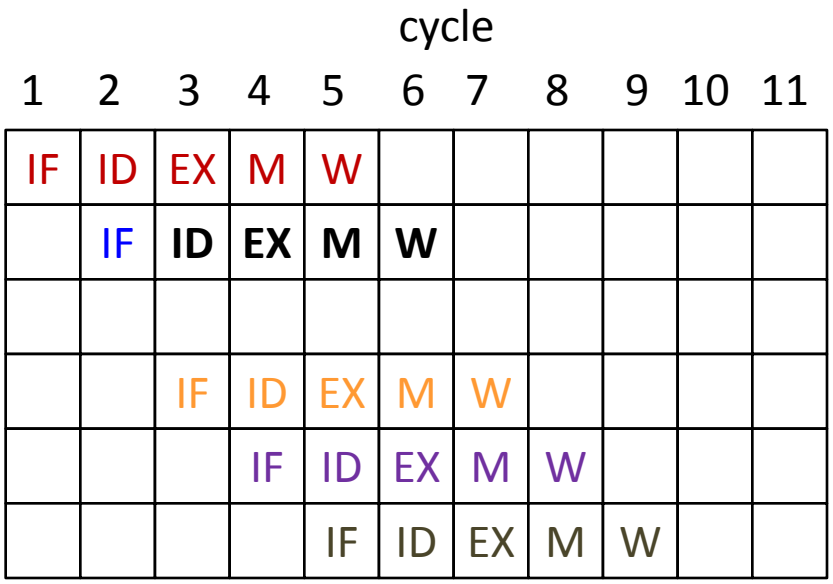
Stall here →

```
Inst1:  lw    $1, 0($2)
Inst2:  add   $2, $1, $1
Inst3:  add   $3, $2, $1
Inst4:  bne   $1,  $1, label
Inst5:  and   $1, $2, $3
Inst6:  or    $1, $1, $1
```

Flush here → (assuming no delay slot)

```
Inst1:  j     Inst4
Inst2:  add   $2, $1, $1
Inst3:  add   $3, $2, $1
Inst4:  bne   $1,  $1, label
Inst5:  and   $1, $2, $3
Inst6:  or    $1, $1, $1
```





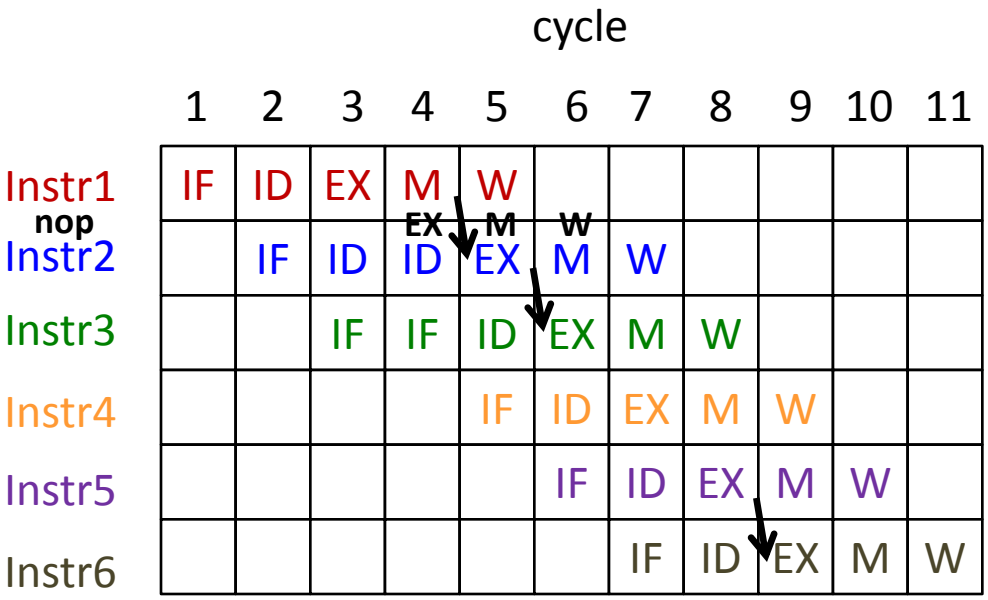
Stalling vs. Flushing Example

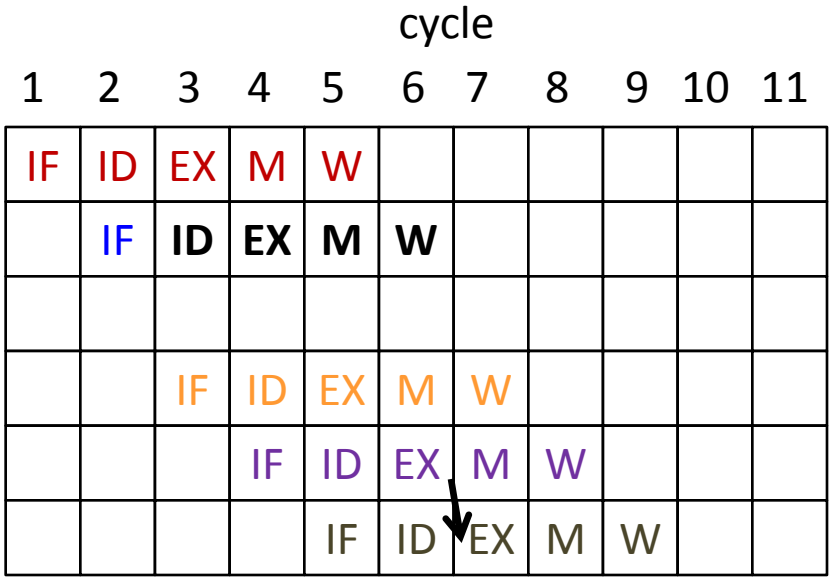
Stall here →

```
Inst1:  lw    $1, 0($2)
Inst2:  add   $2, $1, $1
Inst3:  add   $3, $2, $1
Inst4:  bne   $1,  $1, label
Inst5:  and   $1, $2, $3
Inst6:  or    $1, $1, $1
```

Flush here → (assuming no delay slot)

```
Inst1:  j     Inst4
Inst2:  add   $2, $1, $1
Inst3:  add   $3, $2, $1
Inst4:  bne   $1,  $1, label
Inst5:  and   $1, $2, $3
Inst6:  or    $1, $1, $1
```





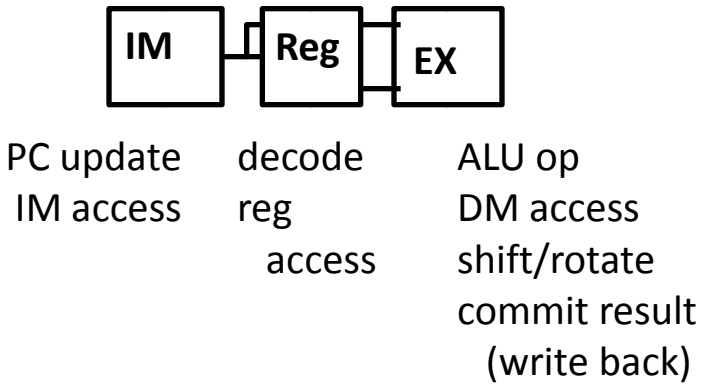
Pipeline Summary

The BIG Picture

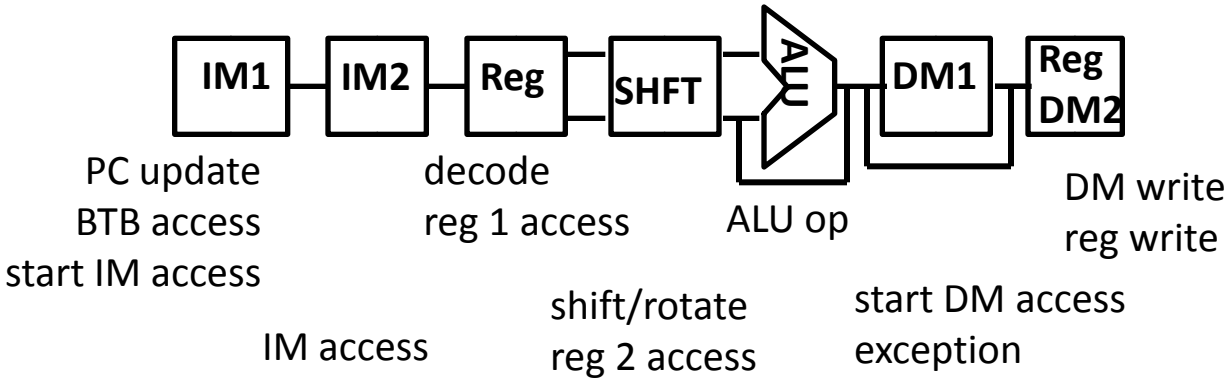
- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Other Sample Pipeline Alternatives

- ARM7



- XScale



Acknowledgments

Some of the slides contain material developed and copyrighted by M.J. Irwin (Penn state), B. Parhami (UCSB), and instructor material for the textbook