CENG 3420
Lecture 06: Pipeline

Bei Yu

byu@cse.cuhk.edu.hk

Outline

- ❑ Pipeline Motivations
- ❑ Pipeline Hazards
- ❑ Exceptions
- ❑ Background: Flip-Flop Control Signals

Outline

- ❑ Pipeline Motivations
- ❑ Pipeline Hazards
- ❑ Exceptions
- ❑ Background: Flip-Flop Control Signals

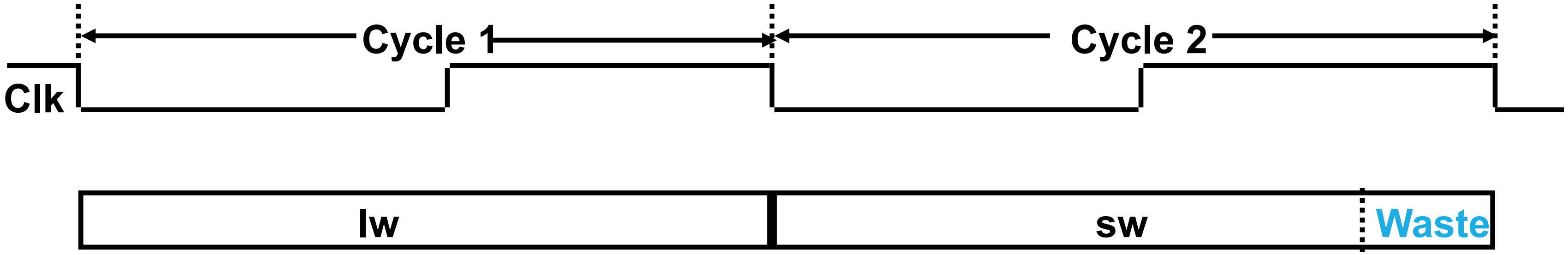
Review: Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires) except:
  - Instruction and Data Memory (4 ns)
  - ALU and adders (2 ns)
  - Register File access (reads or writes) (1 ns)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	4	1	2		1	8
load	4	1	2	4	1	12
store	4	1	2	4		11
beq	4	1	2			7
jump	4					4

Review: Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instr
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

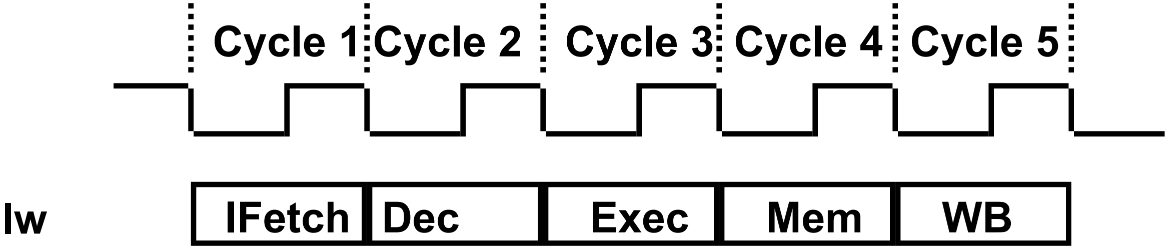
- It is simple and easy to understand

How Can We Make It Faster?

- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Remember *the* performance equation:

    CPU time = CPI * CC * IC
- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
  - A five stage pipeline is nearly five times faster because the CC is “nearly” five times faster
- Fetch (and execute) more than one instruction at a time
  - Superscalar processing – stay tuned

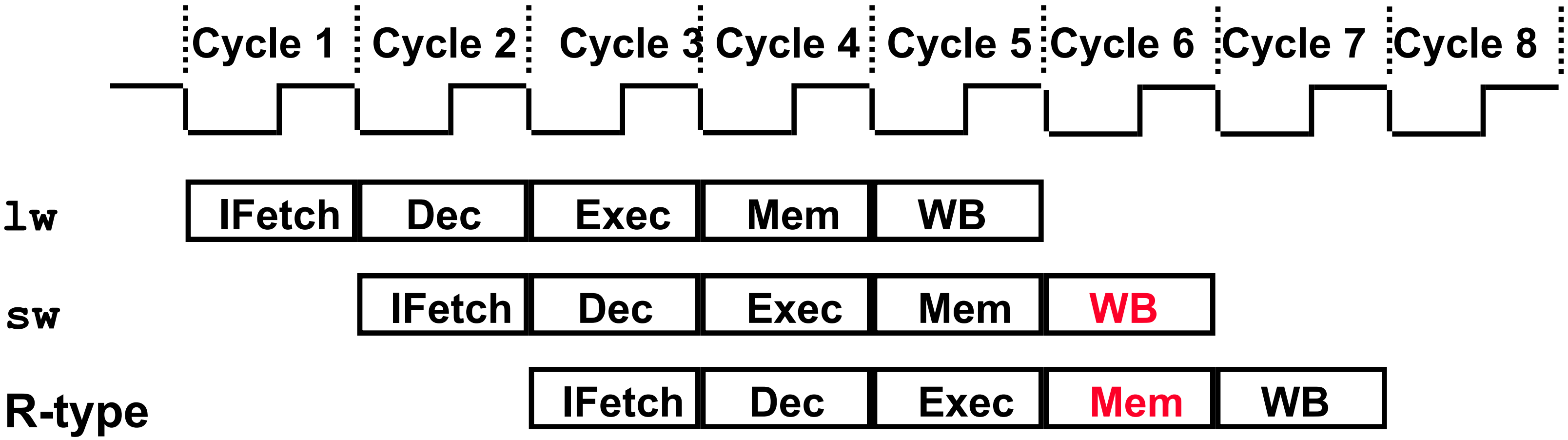
The Five Stages of Load Instruction



- IFetch: Instruction Fetch and Update PC
- Dec: Registers Fetch and Instruction Decode
- Exec: Execute R-type; calculate memory address
- Mem: Read/write the data from/to the Data Memory
- WB: Write the result data into the register file

A Pipelined MIPS Processor

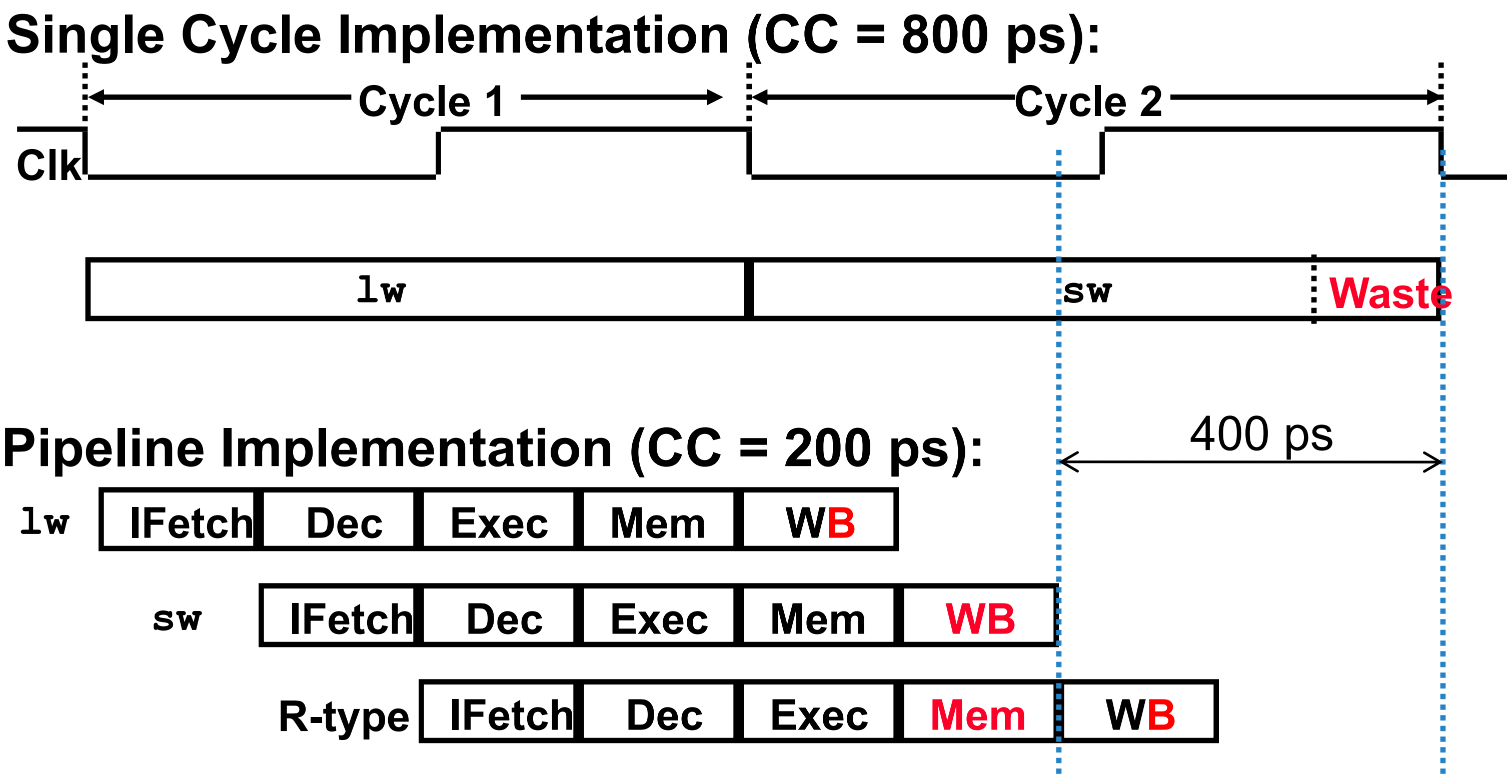
- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
`lw`	IFetch	Dec	Exec	Mem	WB			
`sw`		IFetch	Dec	Exec	Mem	WB		
R-type			IFetch	Dec	Exec	Mem	WB	

- clock cycle (pipeline stage time) is limited by the **slowest** stage
  - for some stages don't need the whole clock cycle (e.g., WB)
  - for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)

Single Cycle versus Pipeline



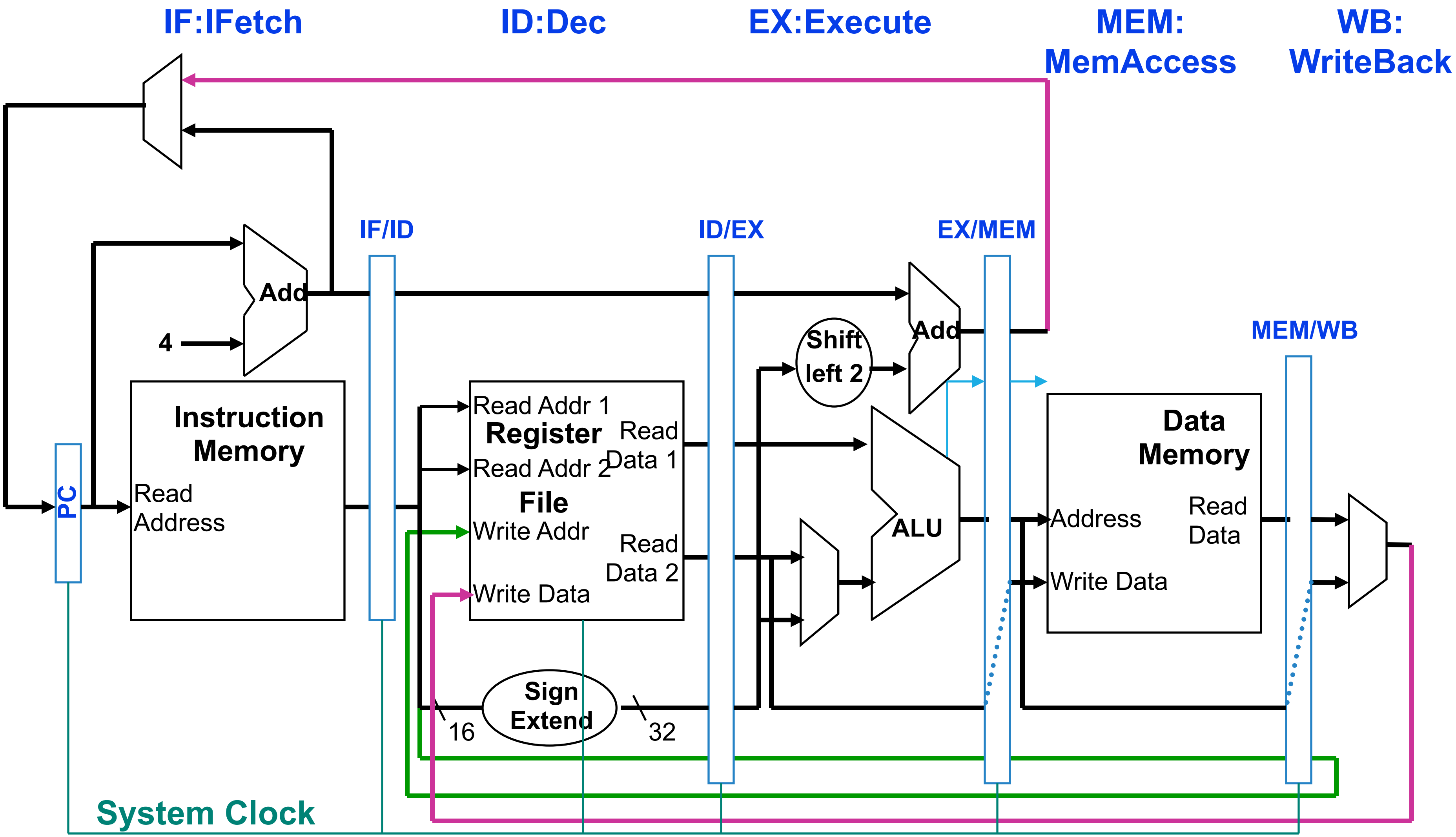
- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
  - operands must be aligned in memory so a single data transfer takes only one data memory access

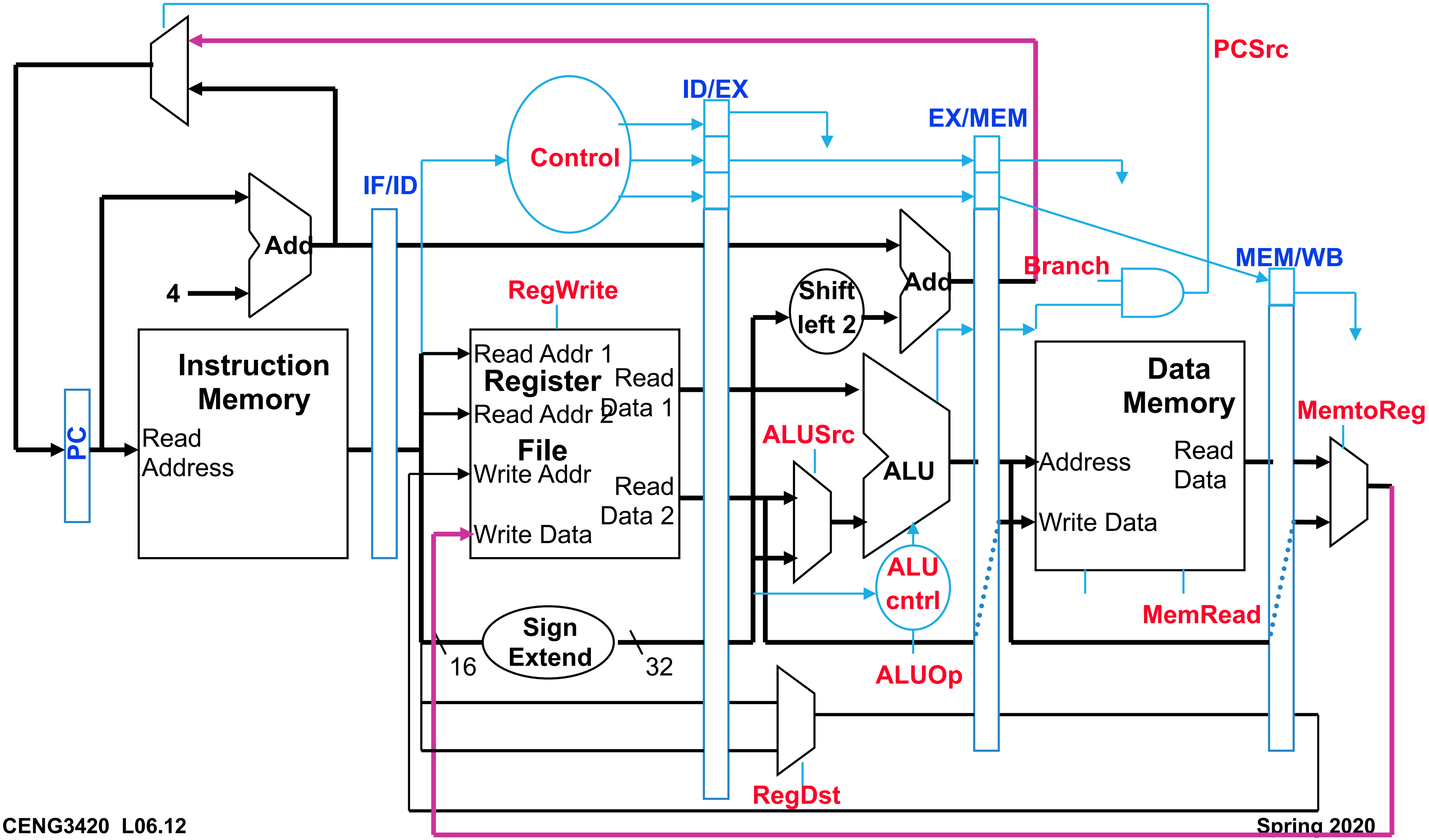
MIPS Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them



MIPS Pipeline Control Path Modifications

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

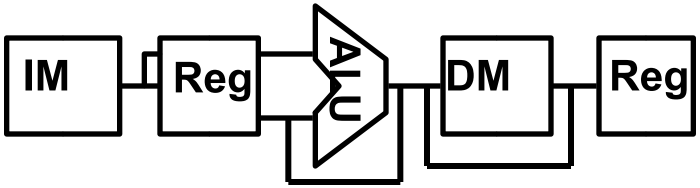


Pipeline Control

- 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

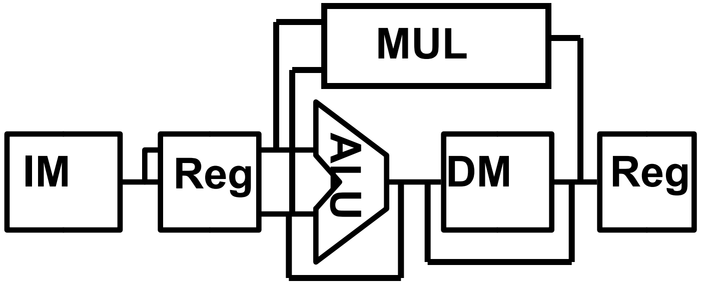
Graphically Representing MIPS Pipeline



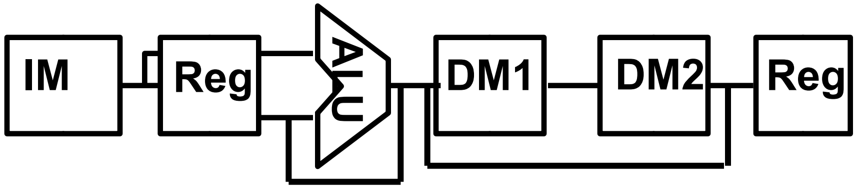
- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)

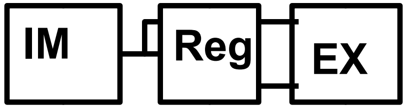


- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



Other Sample Pipeline Alternatives

- ARM7

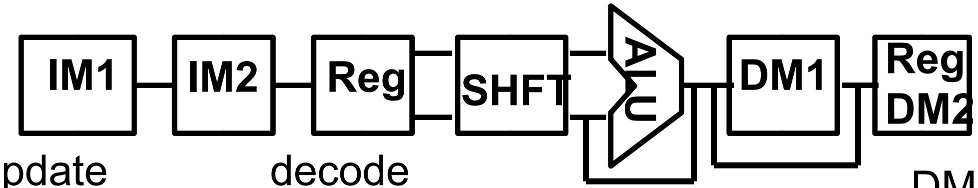


PC update
IM access

decode
reg
access

ALU op
DM access
shift/rotate
commit result
(write back)

- XScale



PC update
BTB access
start IM access

IM access

decode
reg 1 access

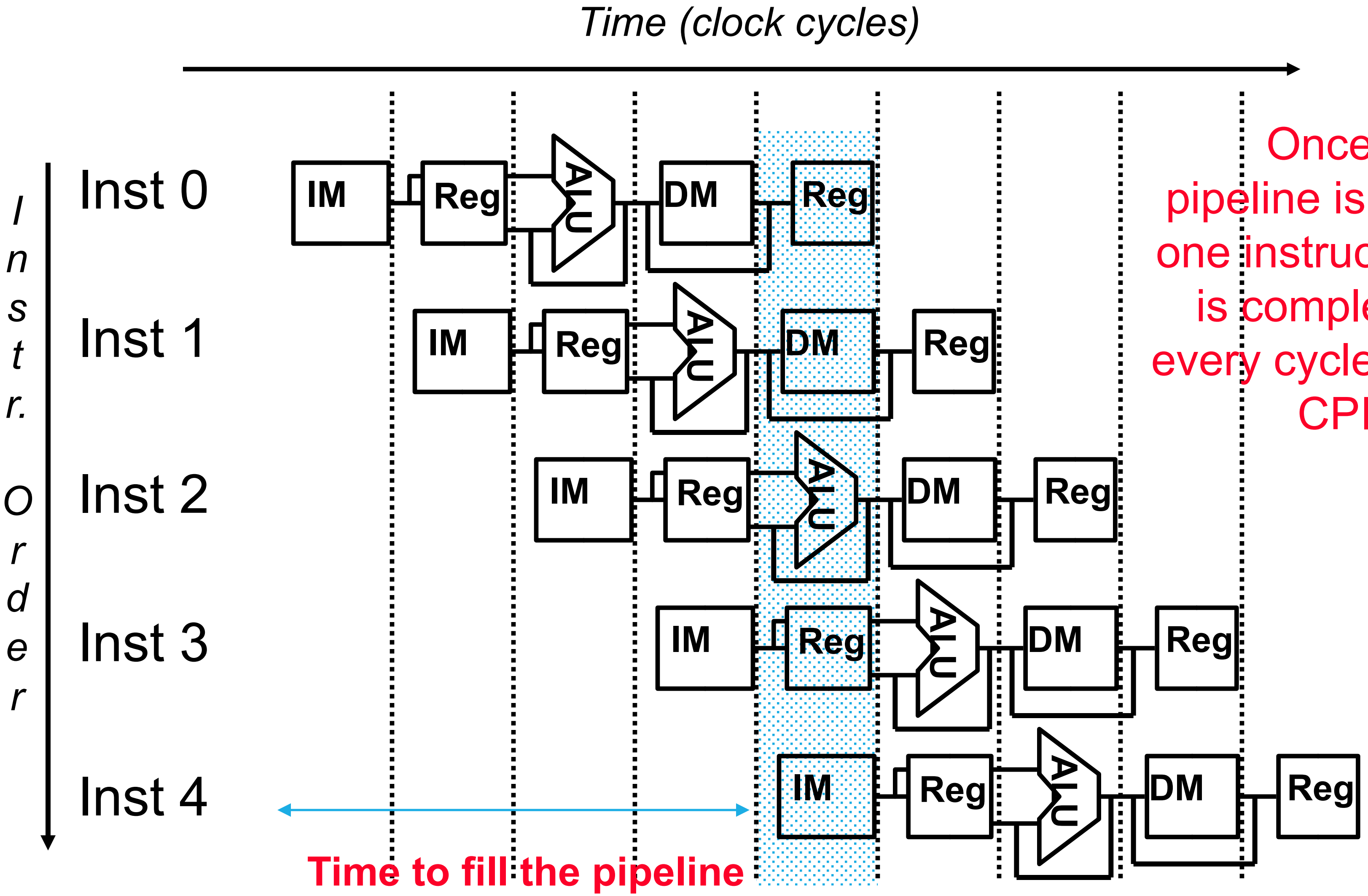
shift/rotate
reg 2 access

ALU op

start DM access
exception

DM write
reg write

Why Pipeline? For Performance!



Once the pipeline is full, one instruction is completed every cycle, so CPI = 1

Time to fill the pipeline

Outline

- ❑ Pipeline Motivations
- ❑ Pipeline Hazards
- ❑ Exceptions
- ❑ Background: Flip-Flop Control Signals

Can Pipelining Get Us Into Trouble?

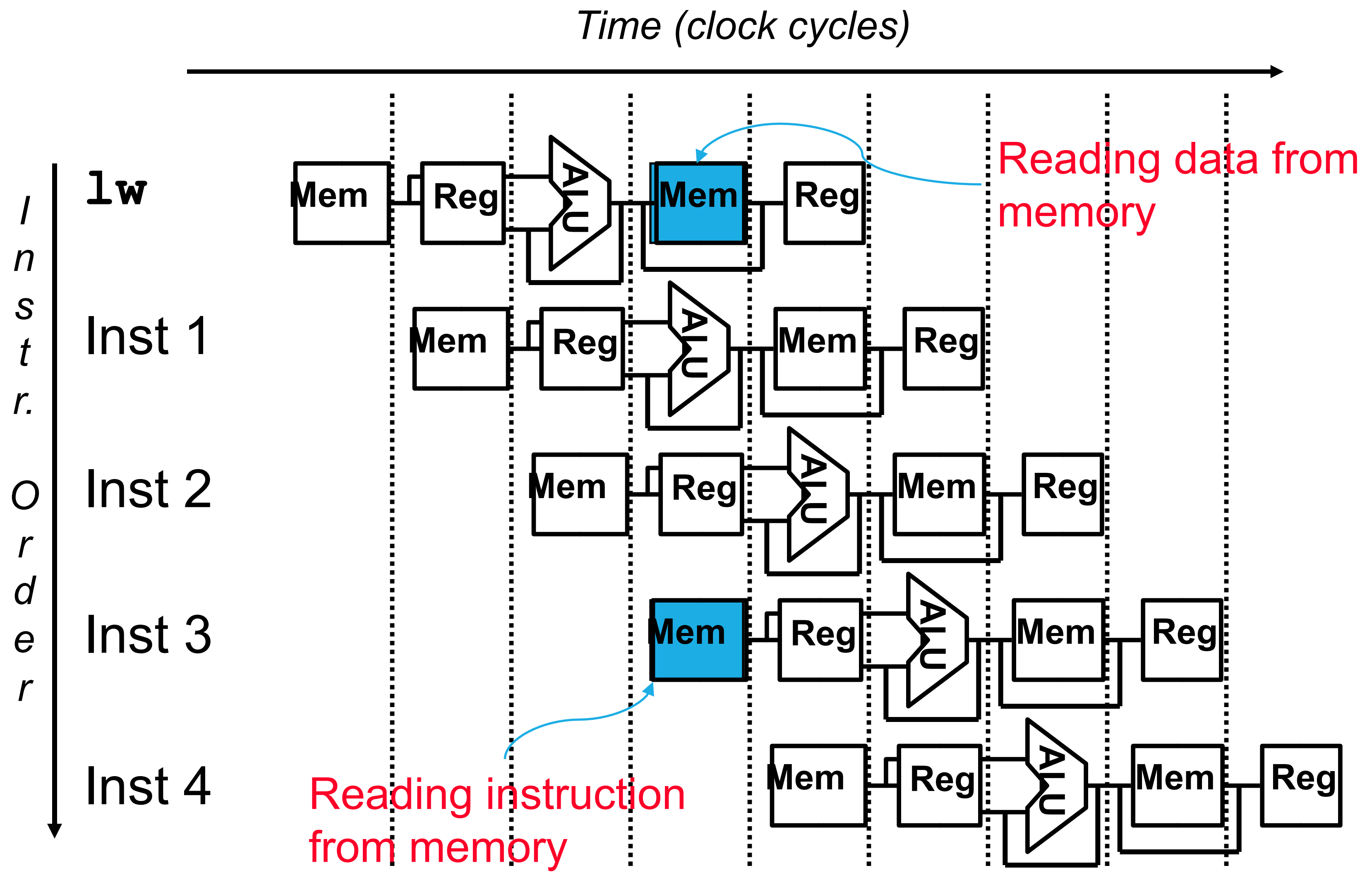
- Yes: Pipeline Hazards
  - structural hazards:
    - a required resource is busy
  - data hazards:
    - attempt to use data before it is ready
  - control hazards:
    - deciding on control action depends on previous instruction

- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

Structure Hazards

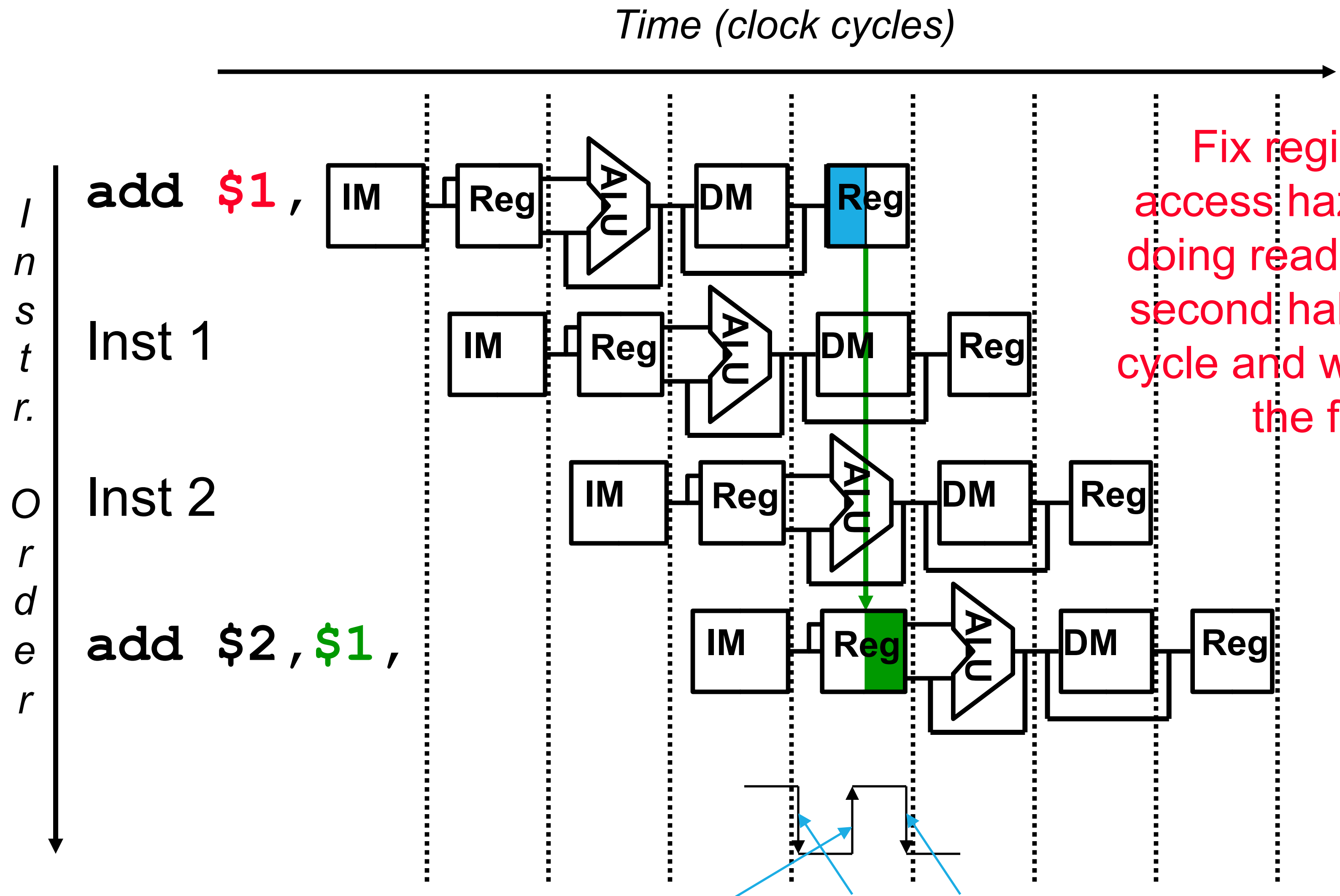
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch requires instruction access
- Hence, pipeline datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- Since Register File

Resolve Structural Hazard 1



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

Resolve Structural Hazard 2



Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half

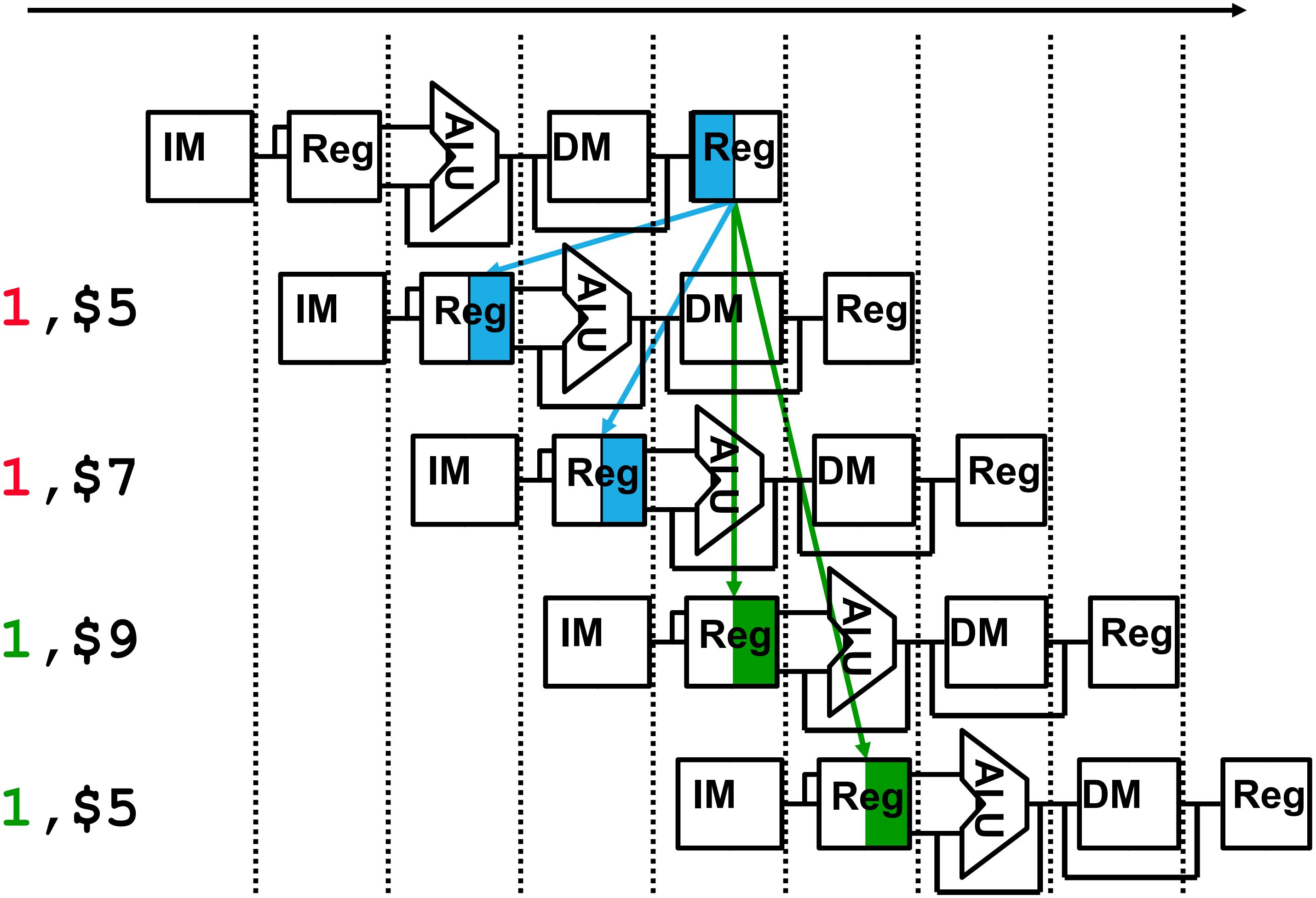
clock edge that controls register writing

clock edge that controls loading of pipeline state registers

Data Hazards: Register Usage

- Dependencies backward in time cause hazards

```
add $1,
sub $4,$1,$5
and $6,$1,$7
or  $8,$1,$9
xor $4,$1,$5
```



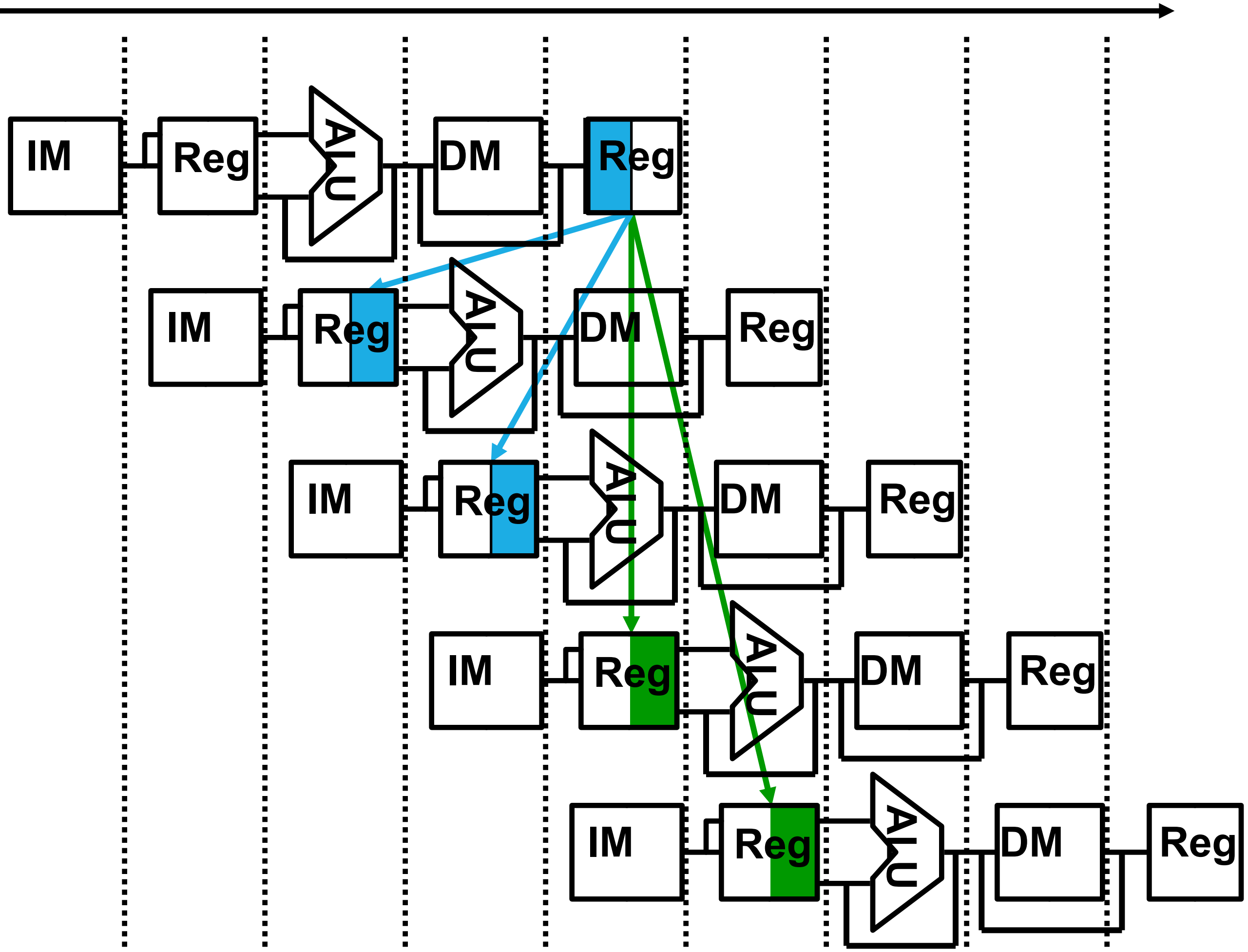
- Read before write data hazard

Data Hazards: Load Memory

- Dependencies backward in time cause hazards

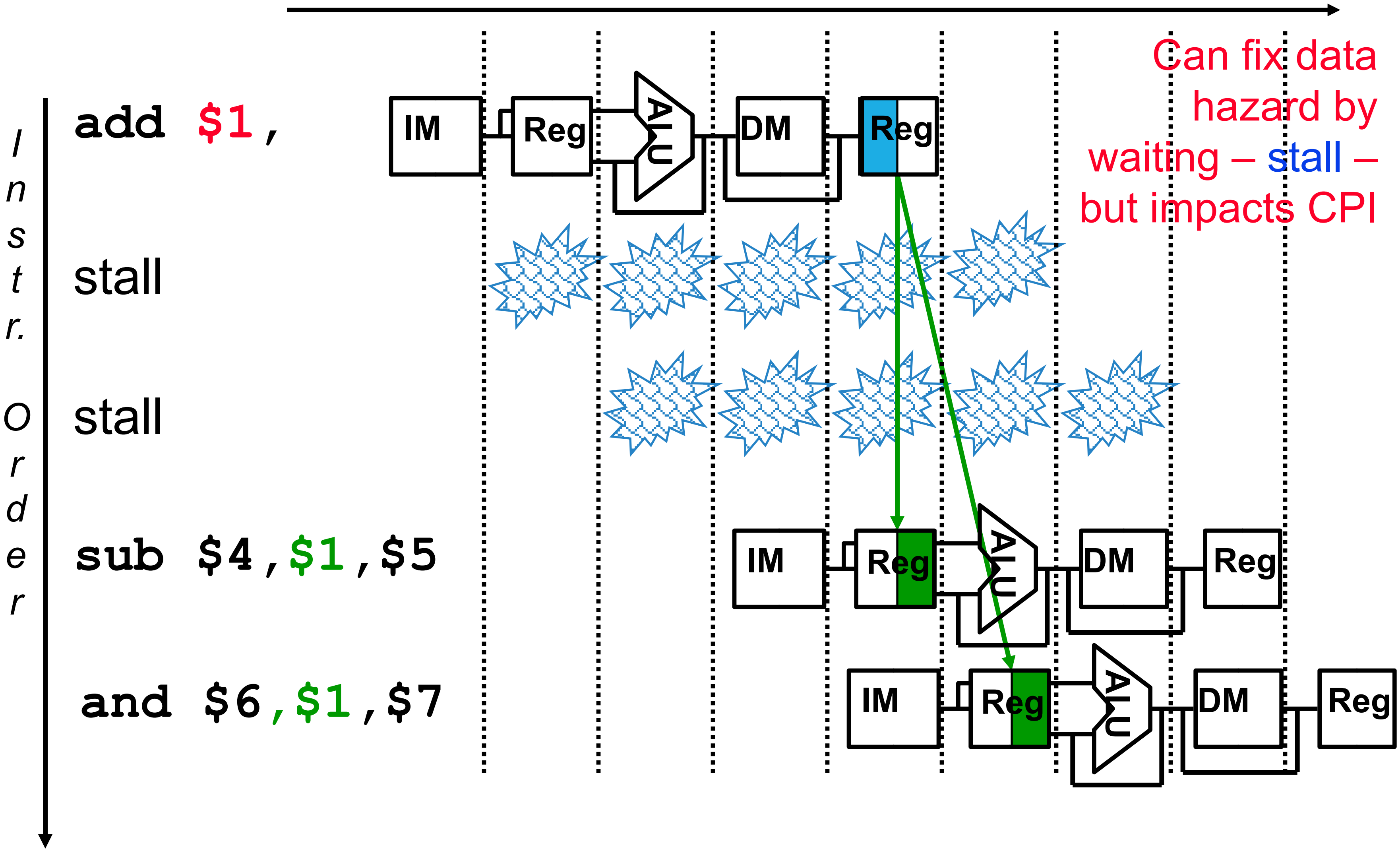
Instr. Order

```
lw   $1,4($2)
sub  $4,$1,$5
and  $6,$1,$7
or   $8,$1,$9
xor  $4,$1,$5
```



- Load-use data hazard

Resolve Data Hazards 1: Insert Stall



add $1,

stall

stall

sub $4,$1,$5

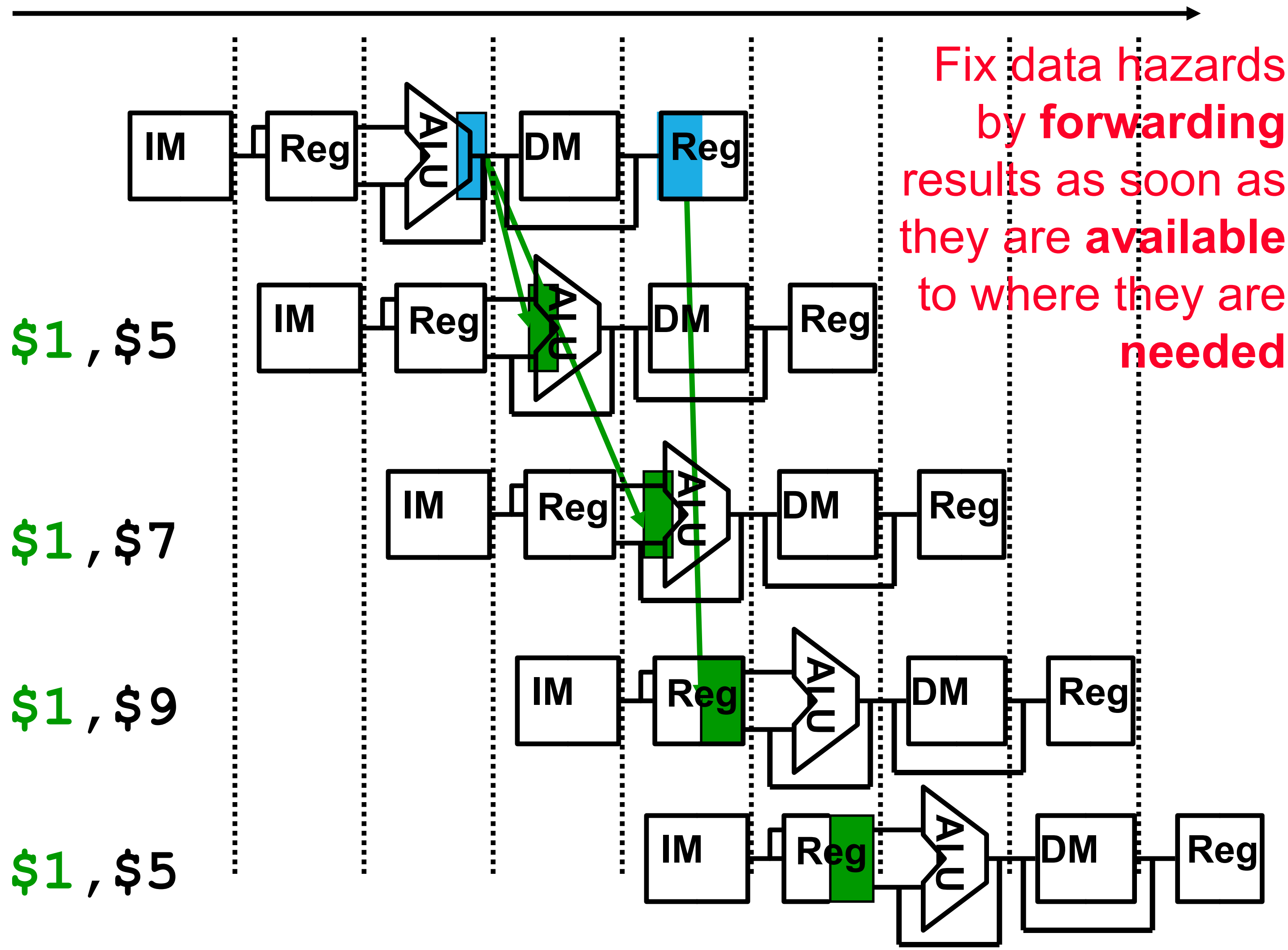
and $6,$1,$7

Can fix data hazard by waiting – stall – but impacts CPI

Resolve Data Hazards 2: Forwarding

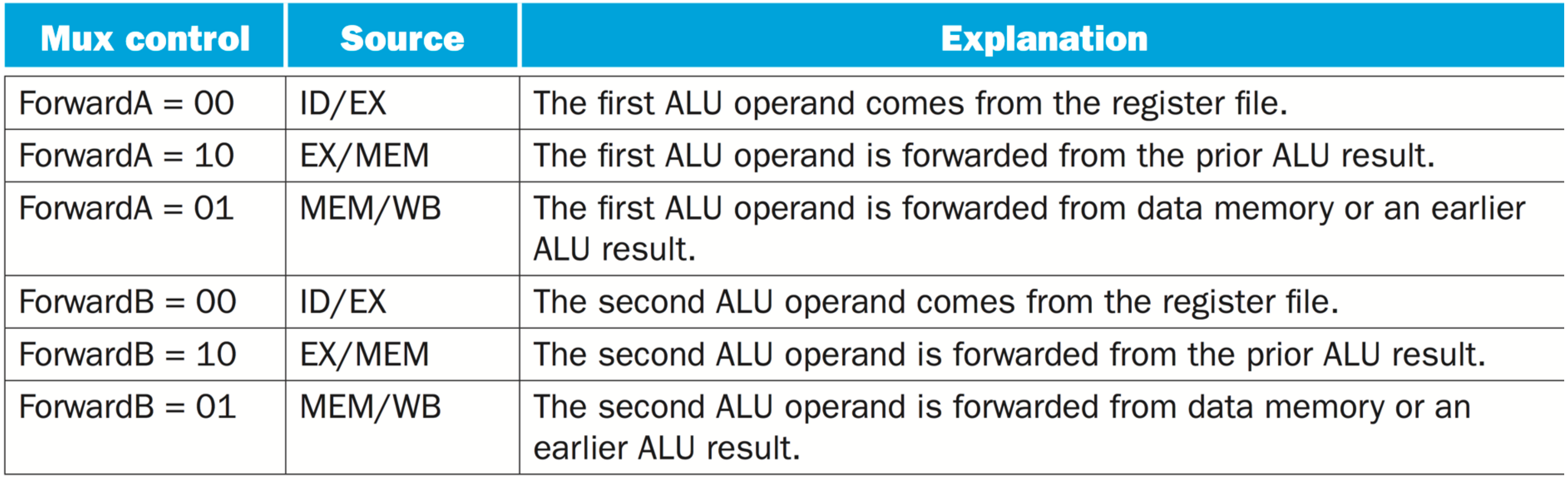
Instr. Order

```
add $1,
sub $4,$1,$5
and $6,$1,$7
or  $8,$1,$9
xor $4,$1,$5
```



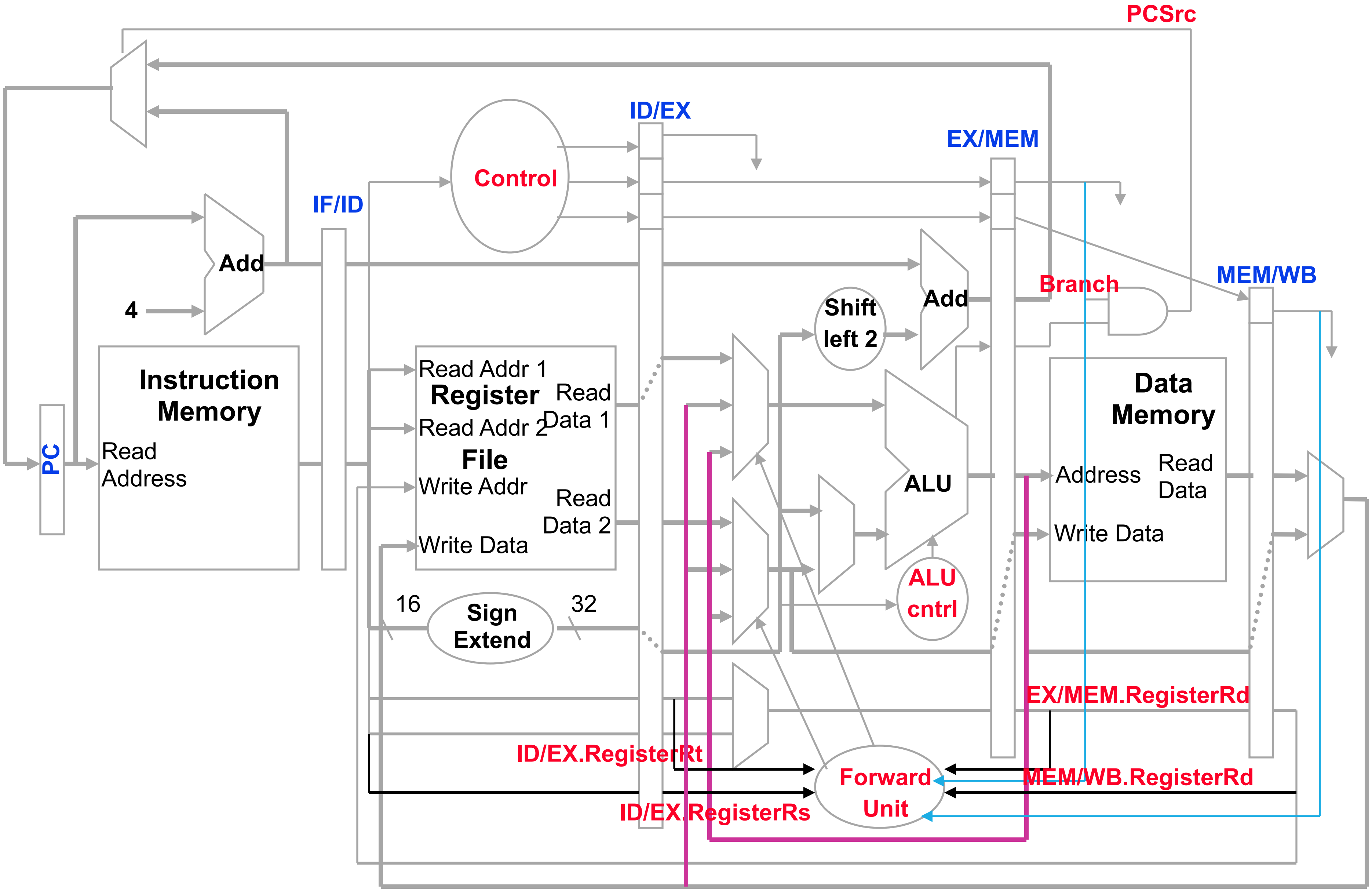
Fix data hazards by **forwarding** results as soon as they are **available** to where they are **needed**

Forward Unit Output Signals



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

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

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

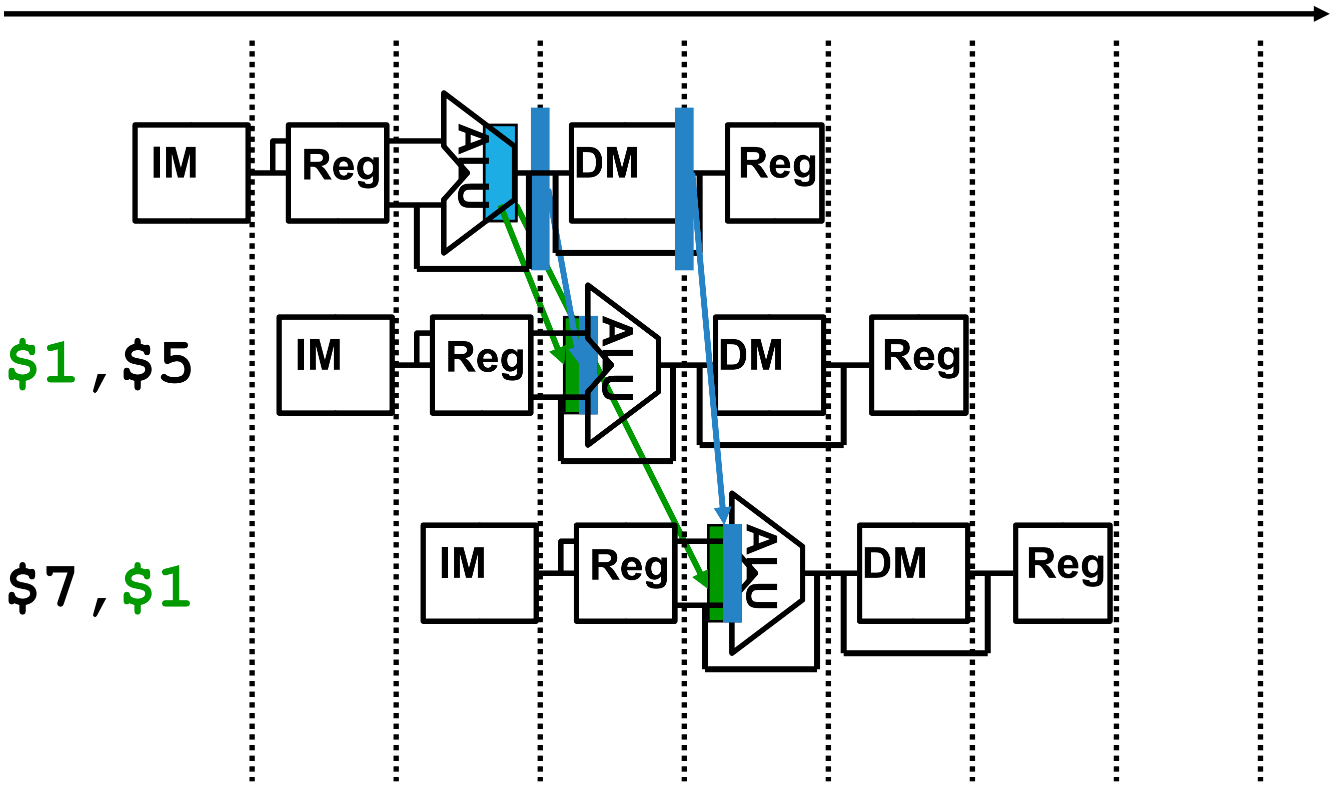
Forwarding Illustration

Instr. Order

`add $1,`

`sub $4,$1,$5`

`and $6,$7,$1`

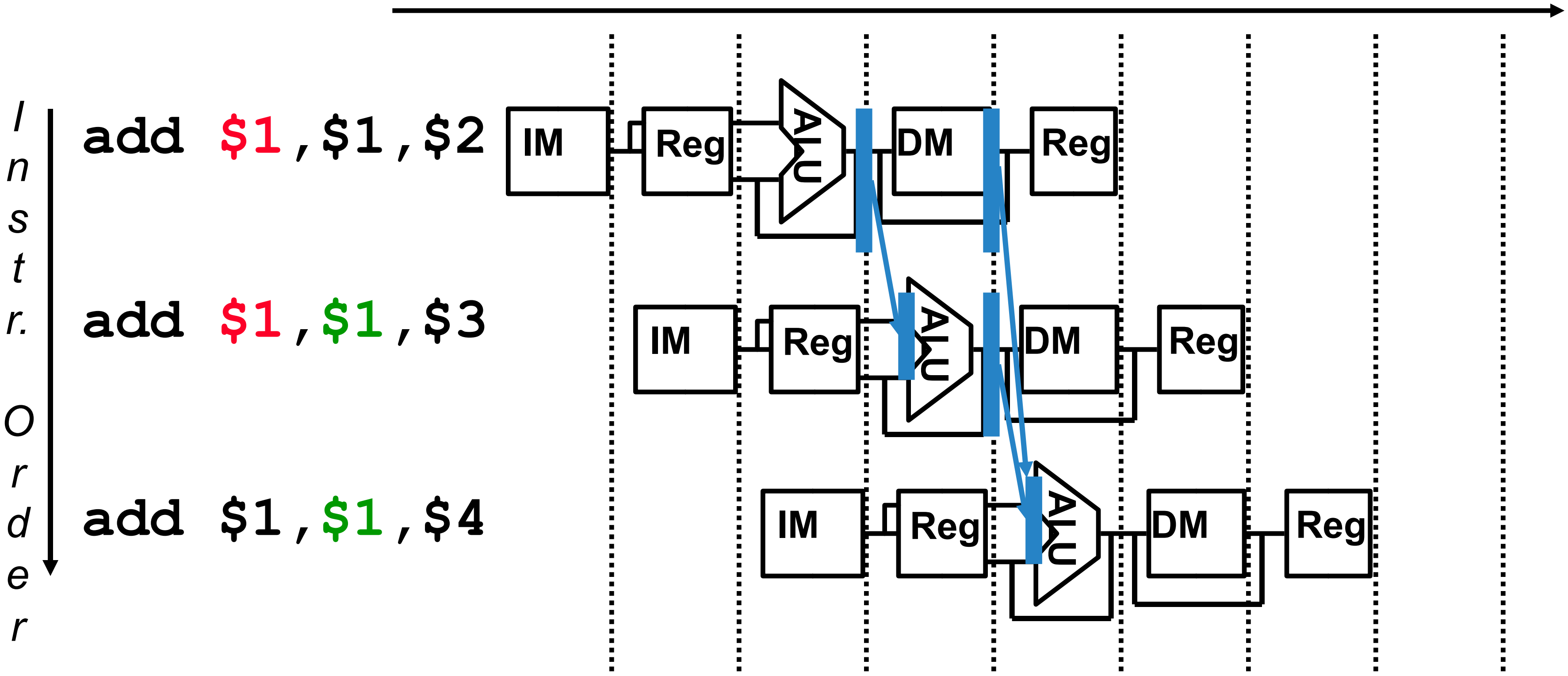


EX forwarding

MEM forwarding

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?



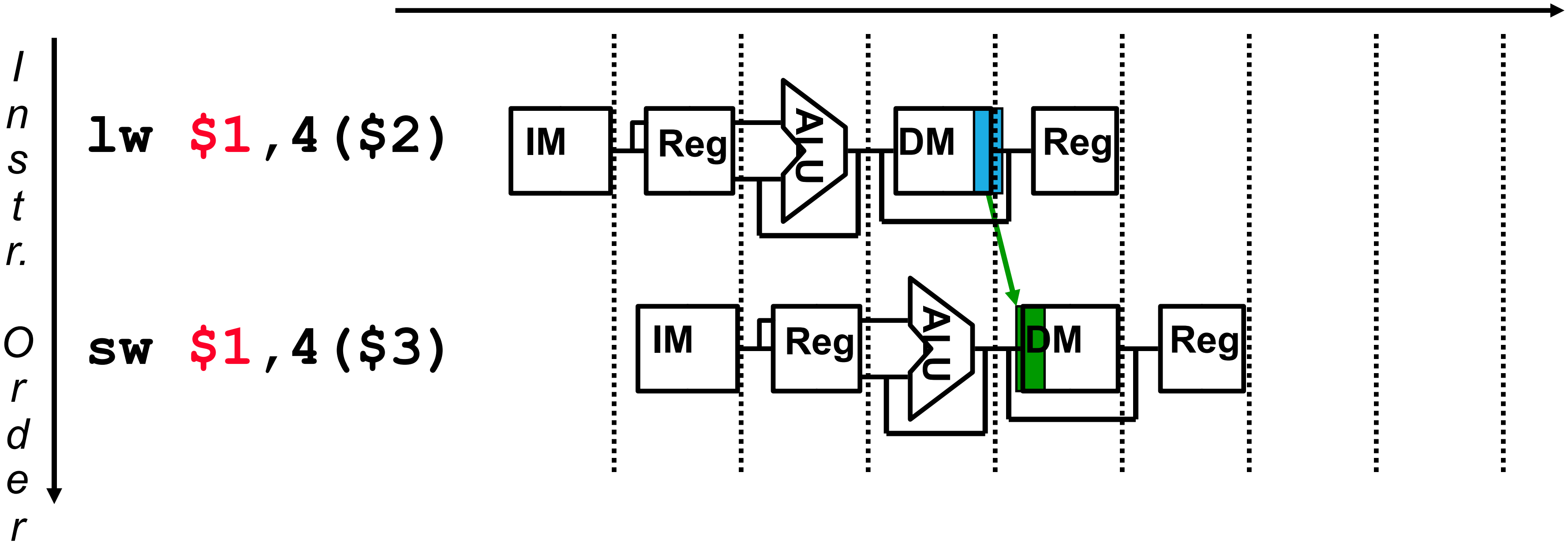
EX: Corrected MEM Forward Unit

- MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)

and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
      ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)

and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
      ForwardB = 01
```

Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
  - Would need to add a Forward Unit and a mux to the MEM stage



Instr. Order

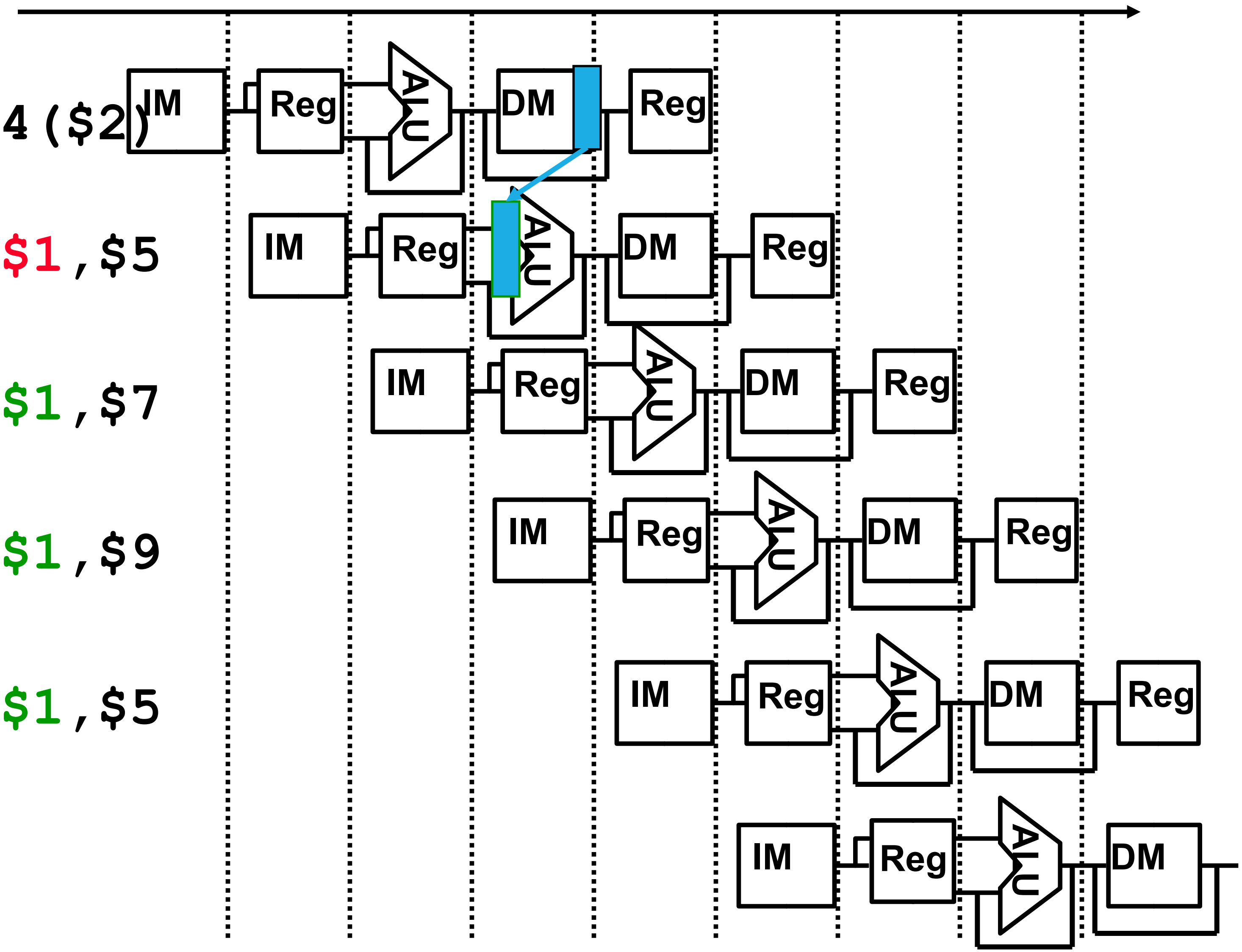
`lw $1,4($2)`

`sw $1,4($3)`

Forwarding with Load-use Data Hazards

Instr. Order

```
lw   $1,4($2)
sub  $4,$1,$5
and  $6,$1,$7
or   $8,$1,$9
xor  $4,$1,$5
```

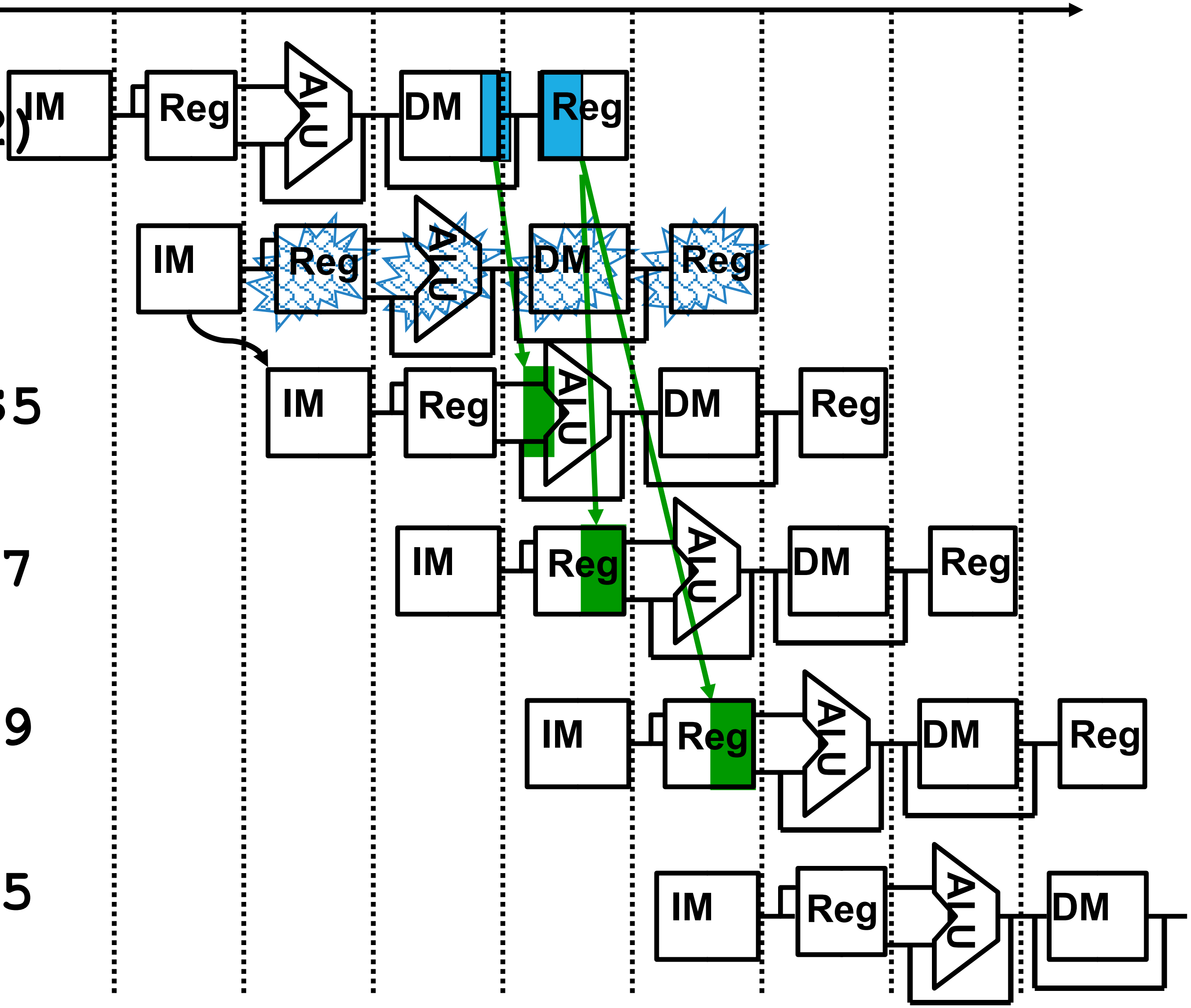


- Will still need one stall cycle even with forwarding

Forwarding with Load-use Data Hazards

Instr. Order

```
lw   $1,4($2)
stall
sub  $4,$1,$5
and  $6,$1,$7
or   $8,$1,$9
xor  $4,$1,$5
```



- Will still need one stall cycle even with forwarding

Load-use Hazard Detection Unit (optional)

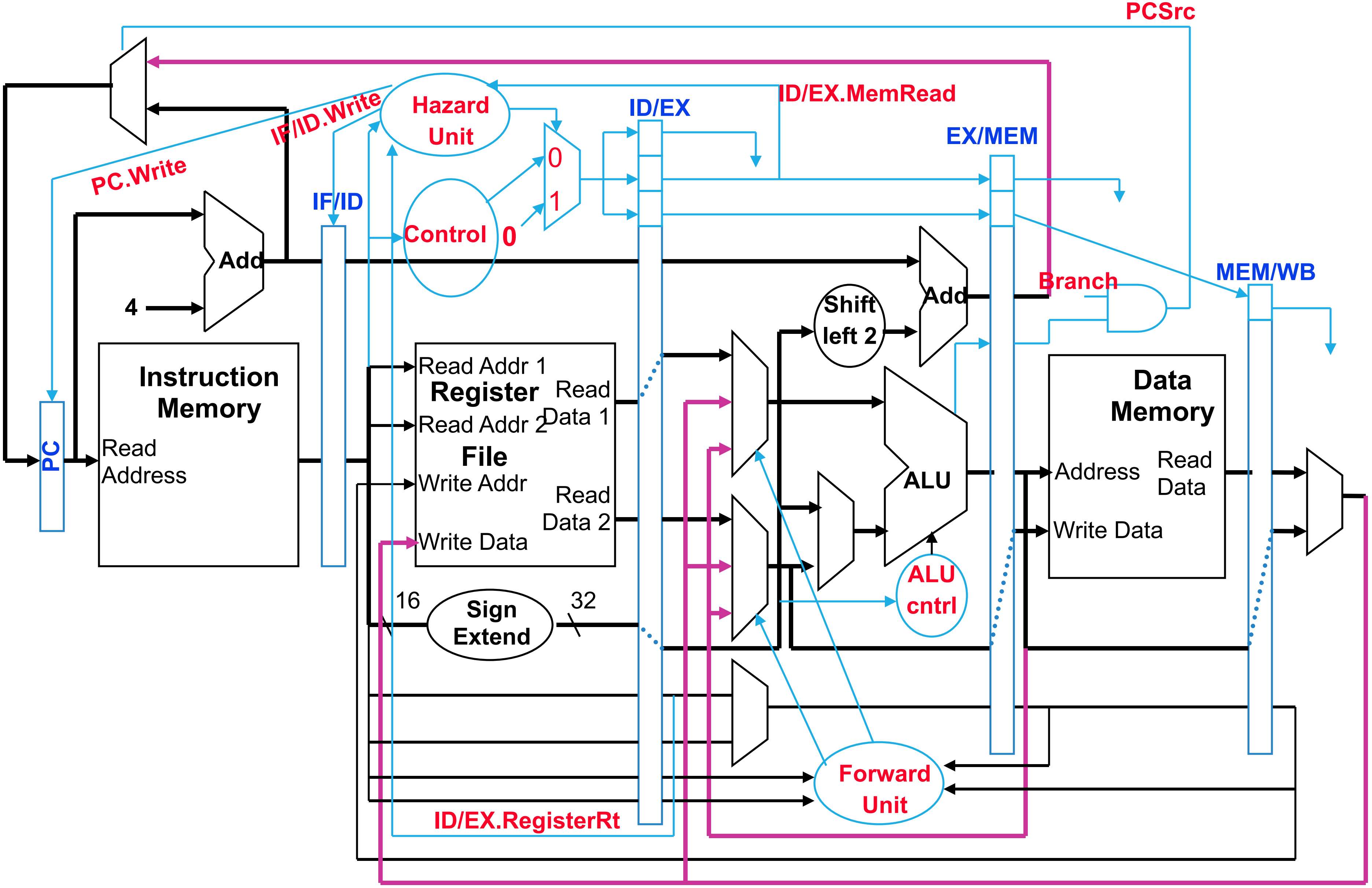
- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

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

Adding the Hazard/Stall Hardware (optional)

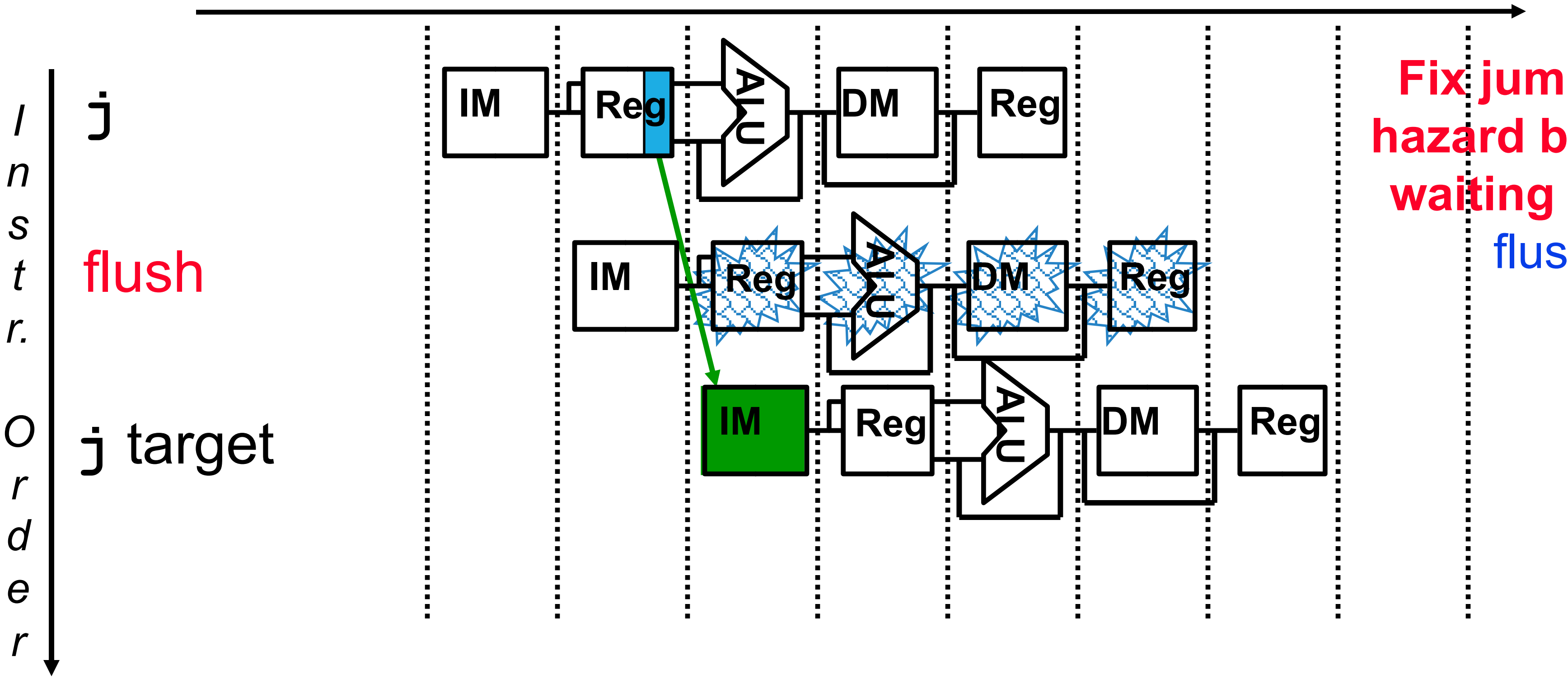


Control Hazards

- 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

Control Hazards 1: Jumps Incur One Stall

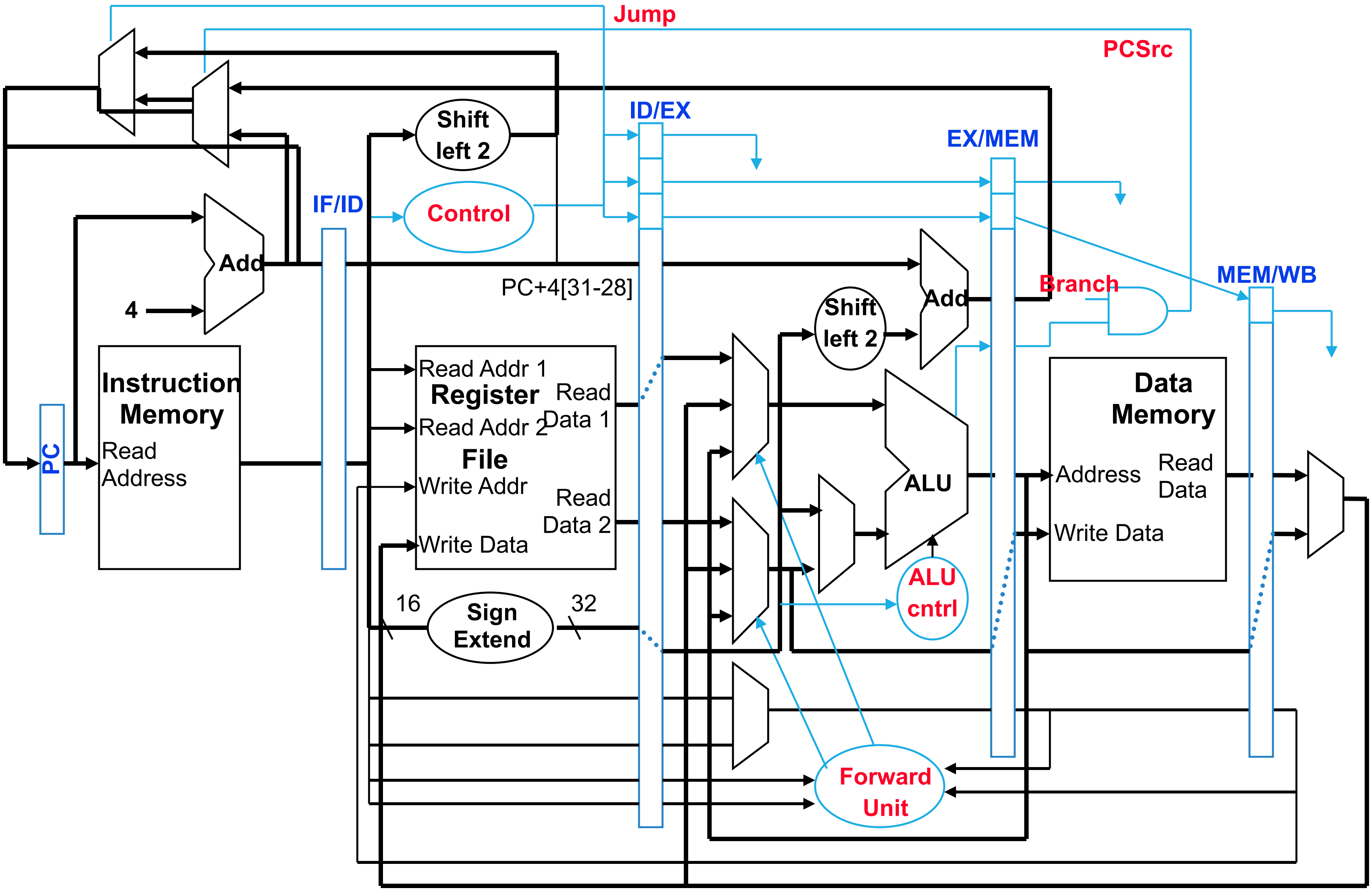
- Jumps not decoded until ID, so one flush is needed
  - To flush, set `IF.Flush` to zero the instruction field of the IF/ID pipeline register (turning it into a `nop`)



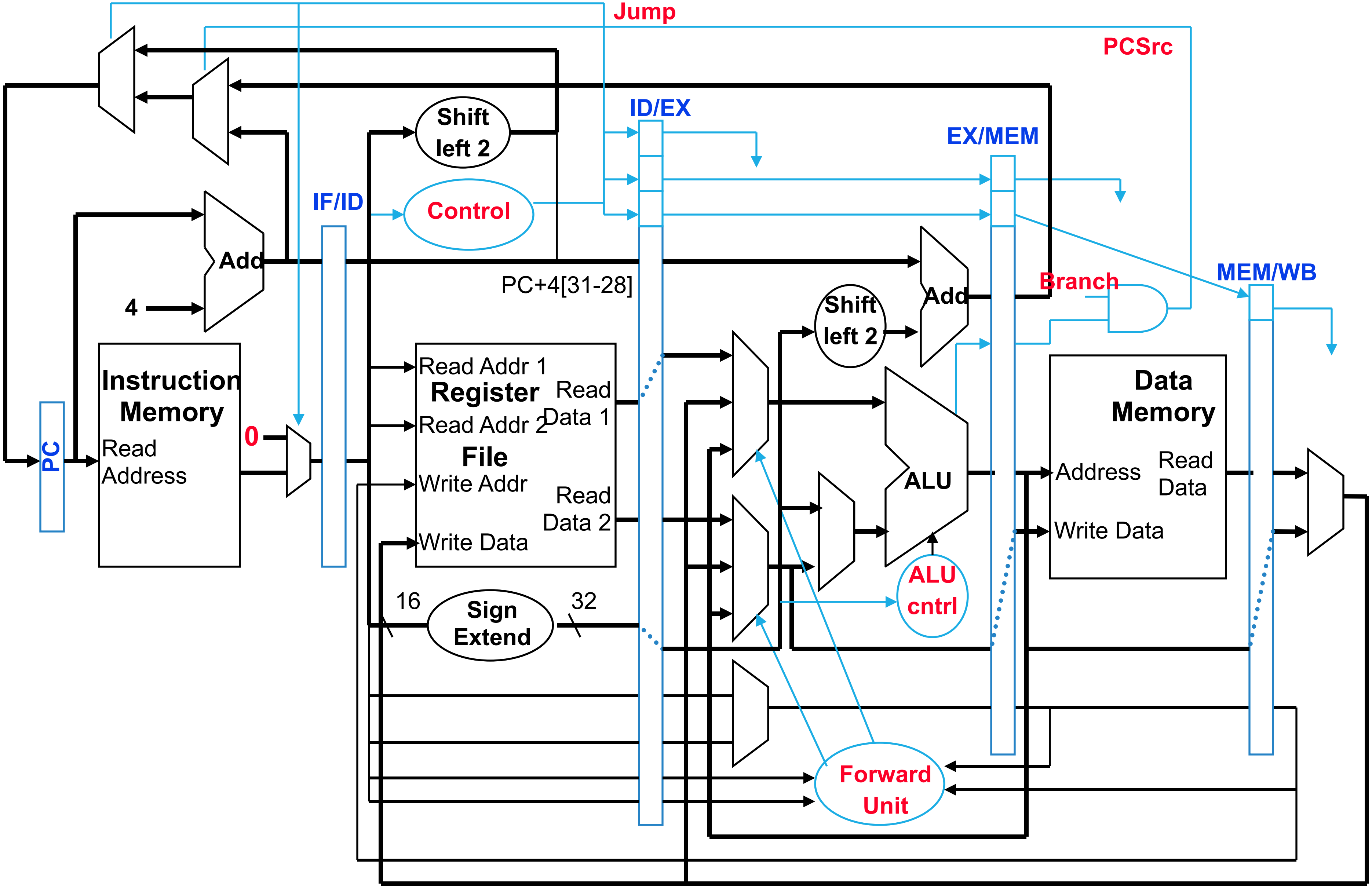
Fix jump hazard by waiting – flush

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

Datapath Branch and Jump Hardware

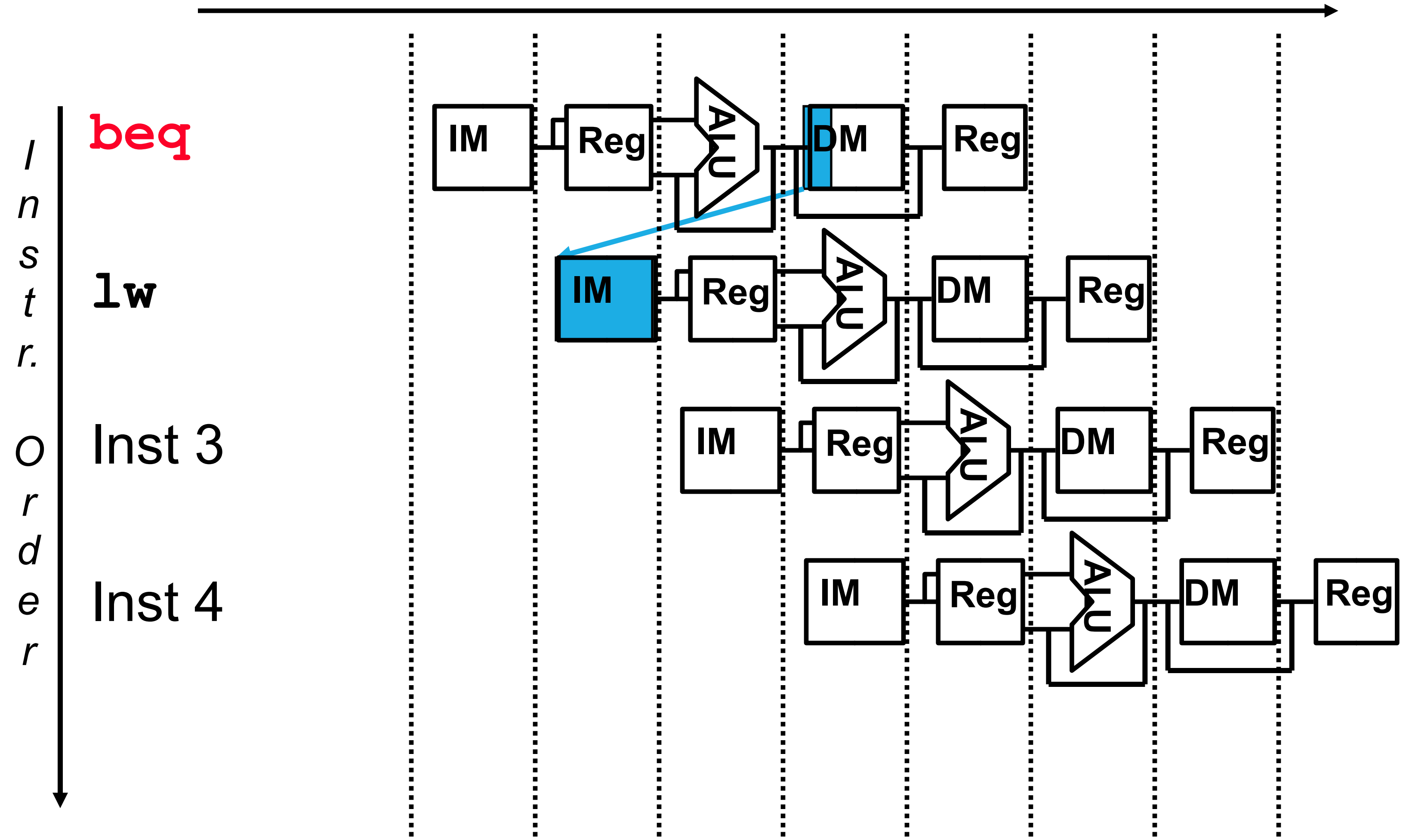


Supporting ID Stage Jumps

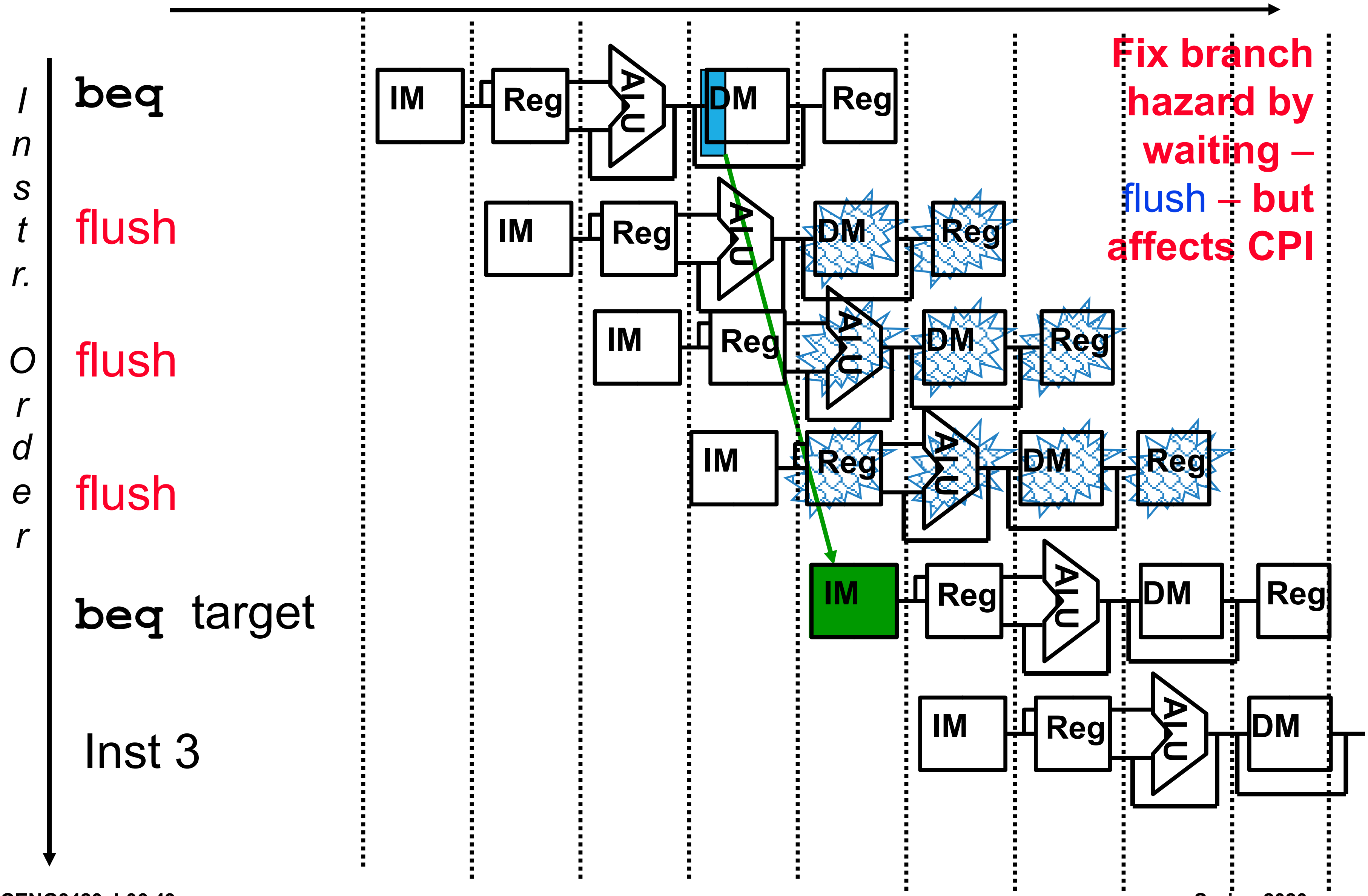


Control Hazards 2: Branch Instr

- Dependencies backward in time cause hazards



One Way to “Fix” a Branch Control Hazard



Instr. Order

beq

flush

flush

flush

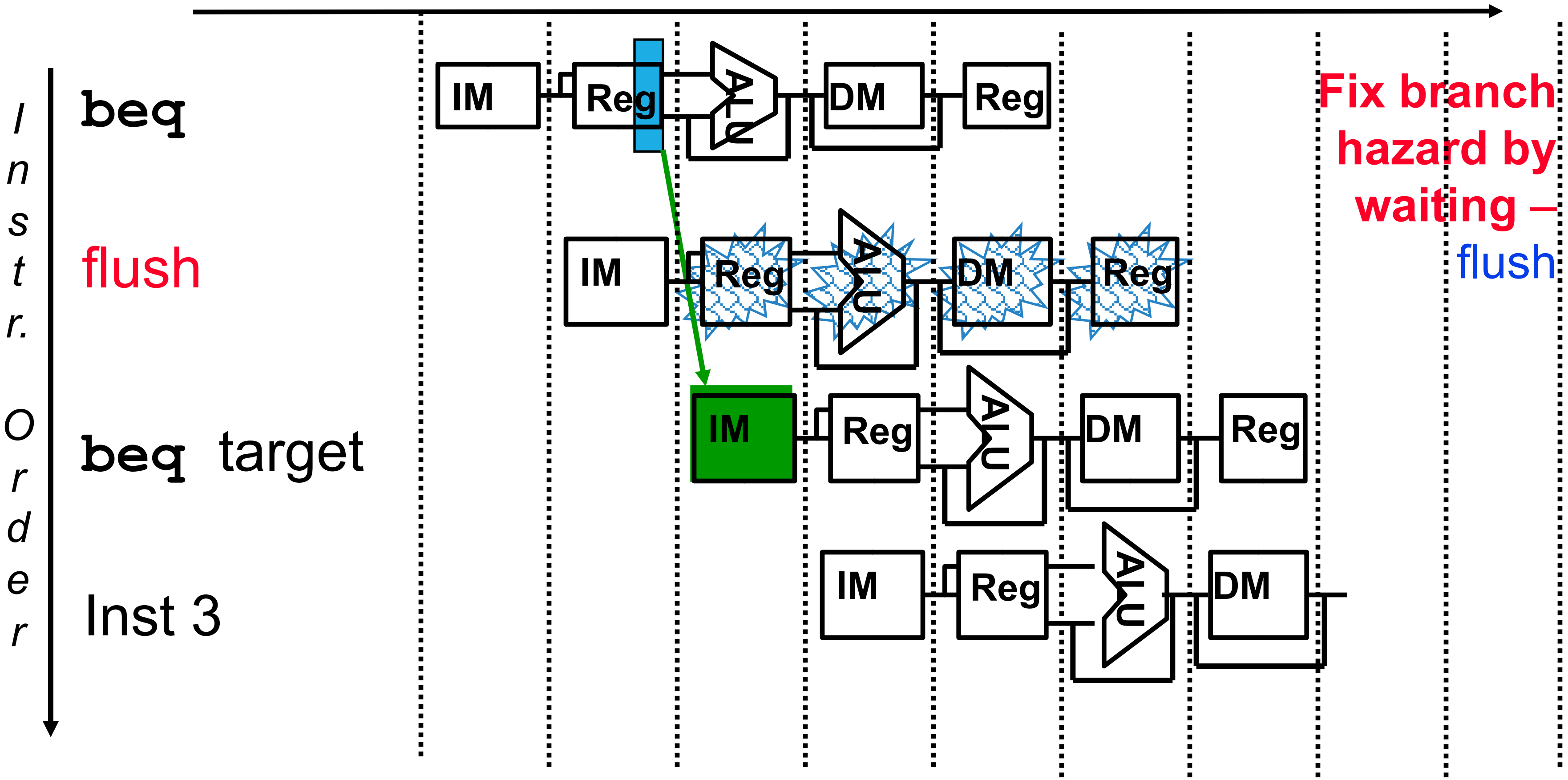
beq target

Inst 3

Fix branch hazard by waiting – flush – but affects CPI

Another Way to "Fix" a Branch Control Hazard

- Move branch decision hardware back to as early in the pipeline as possible – i.e., during the decode cycle



Fix branch hazard by waiting – flush

Two "Types" of Stalls

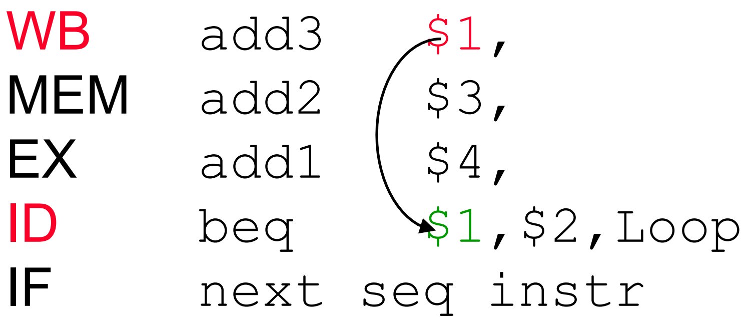
- `Nop` instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle ("bounce" them in place with write control signals)
  - Insert `nop` 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

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

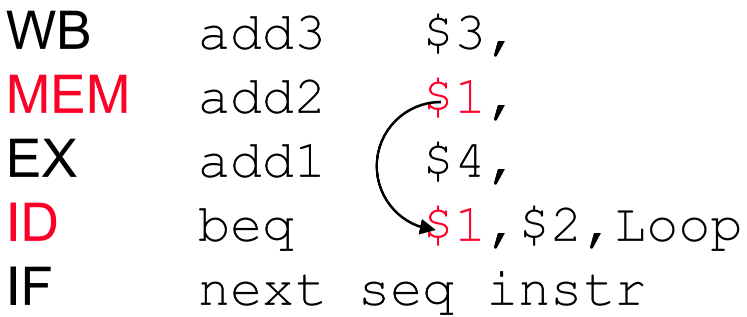
ID Branch Forwarding Issues

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



```
WB     add3   $1,
MEM    add2   $3,
EX     add1   $4,
ID     beq    $1,$2,Loop
IF     next_seq_instr
```

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

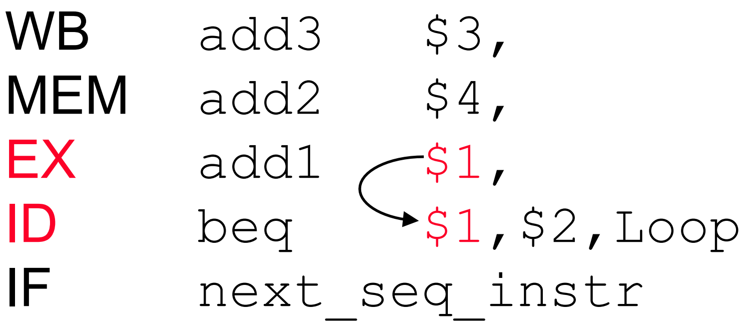


```
WB     add3   $3,
MEM    add2   $1,
EX     add1   $4,
ID     beq    $1,$2,Loop
IF     next_seq_instr
```

```
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

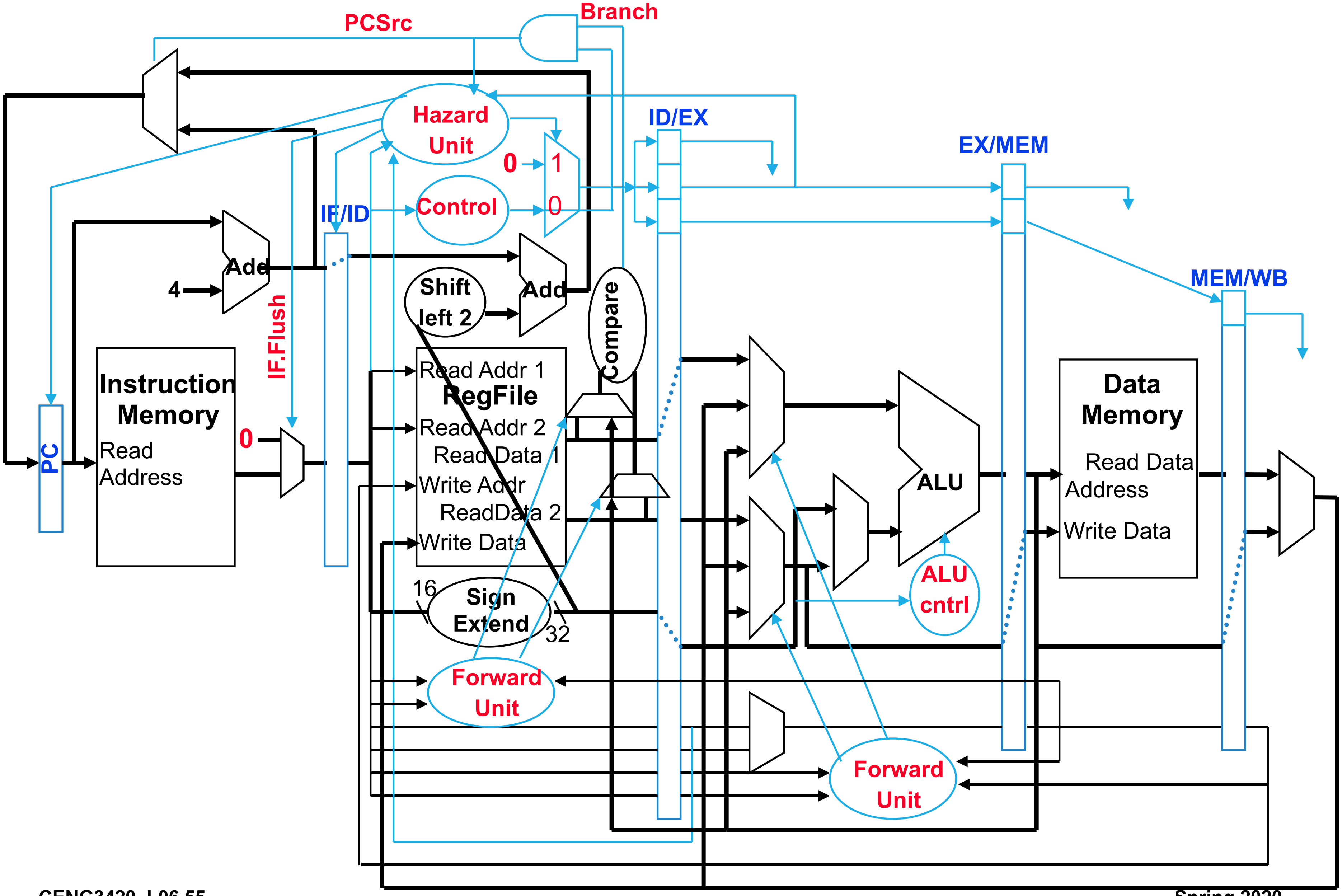
ID Branch Forwarding Issues, con't



```
WB    add3   $3,
MEM   add2   $4,
EX    add1   $1,
ID    beq    $1,$2,Loop
IF    next_seq_instr
```

- 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`)

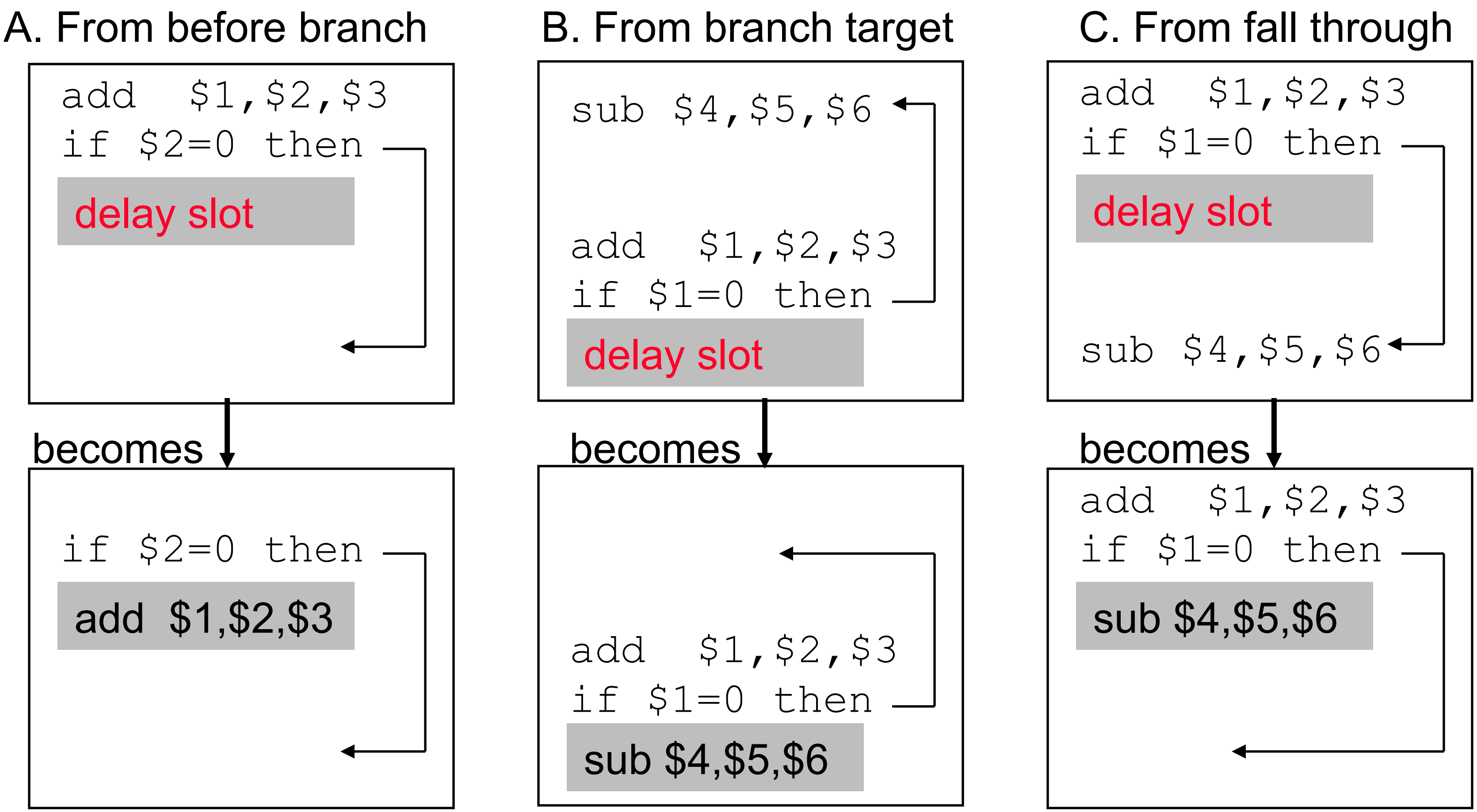
Supporting ID Stage Branches (optional)



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. From before branch

```
add  $1,$2,$3
if $2=0 then
delay slot
```

becomes

```
if $2=0 then
add $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
delay slot
```

becomes

```
add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

C. From fall through

```
add  $1,$2,$3
if $1=0 then
delay slot

sub $4,$5,$6
```

becomes

```
add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

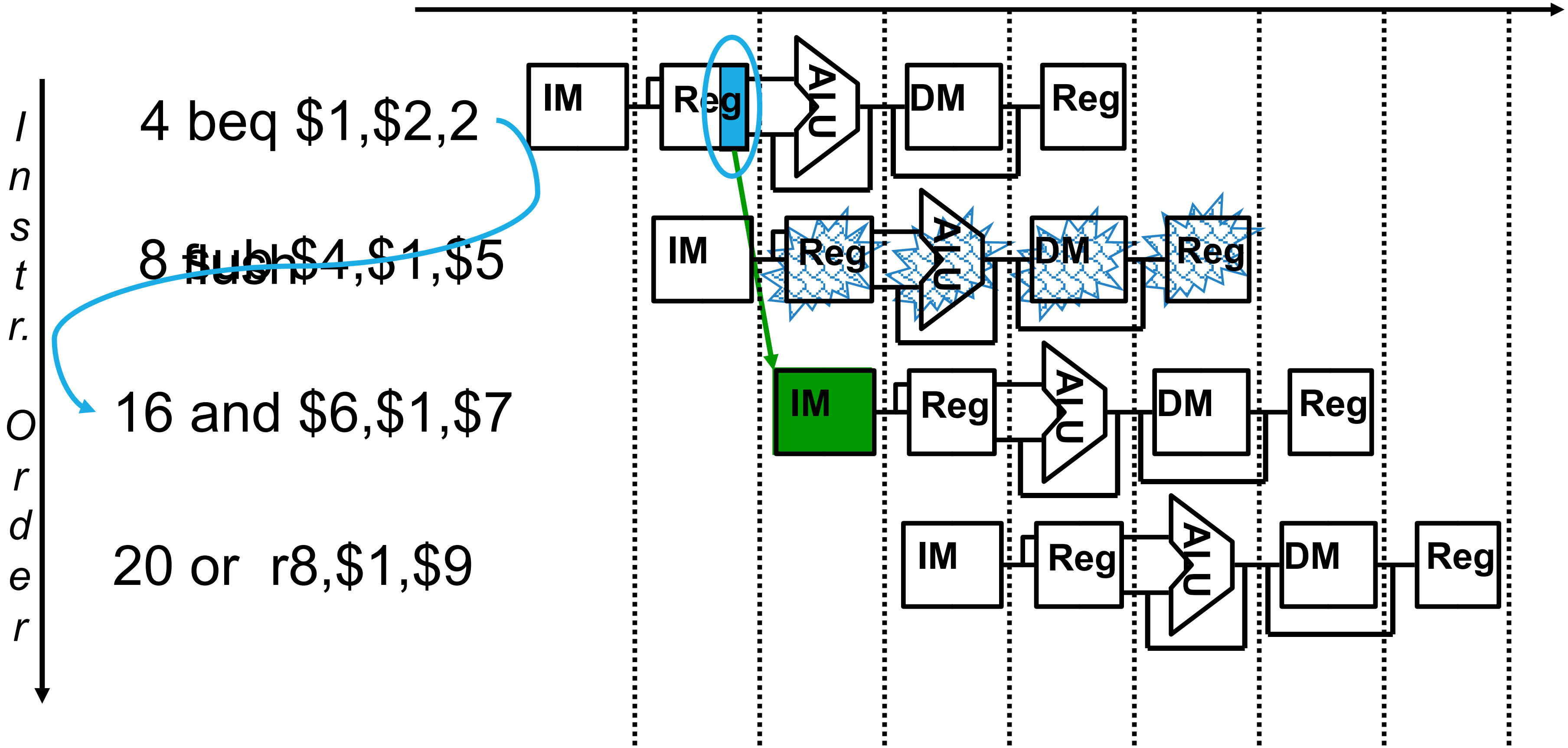
- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

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 `nop`)

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

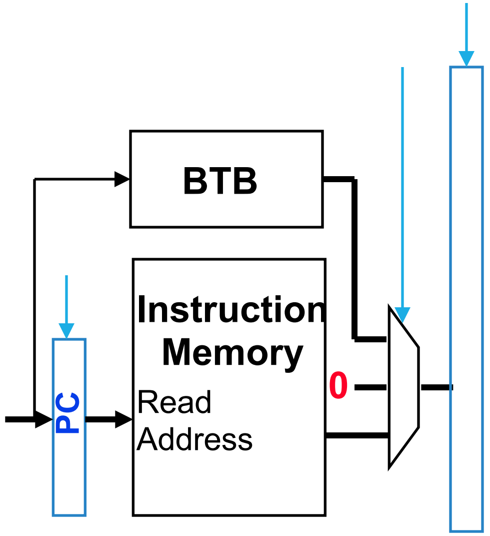
3. Dynamic branch prediction – predict branches at run-time using *run-time* information

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

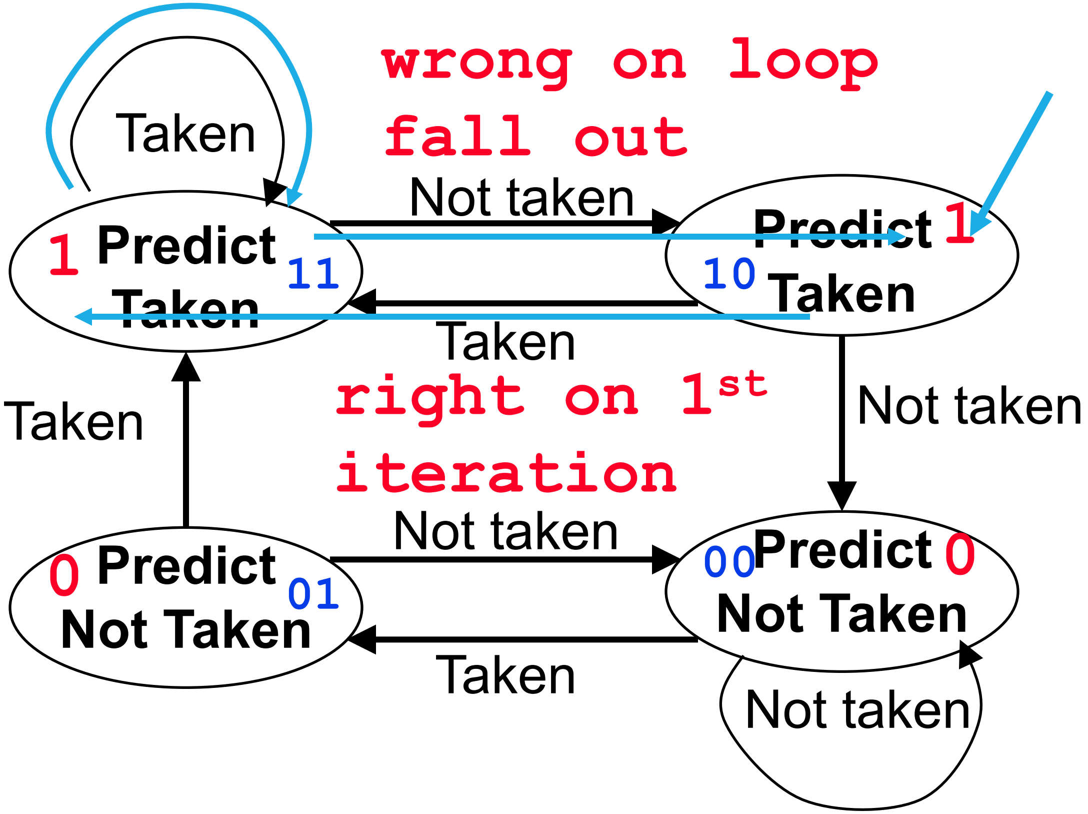
2-bit Predictors

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

right 9 times

wrong on loop fall out

right on 1st iteration



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

- BHT also stores the initial FSM state

CHOOSE YOUR WEAPON
PYTHON
JAVA
C++
ASSEMBLY

Outline

- ❑ Pipeline Motivations
- ❑ Pipeline Hazards
- ❑ Exceptions
- ❑ Background: Flip-Flop Control Signals

Dealing with Exceptions

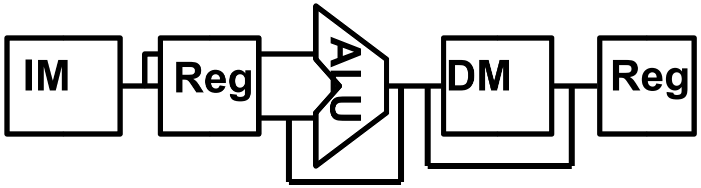
- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline 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

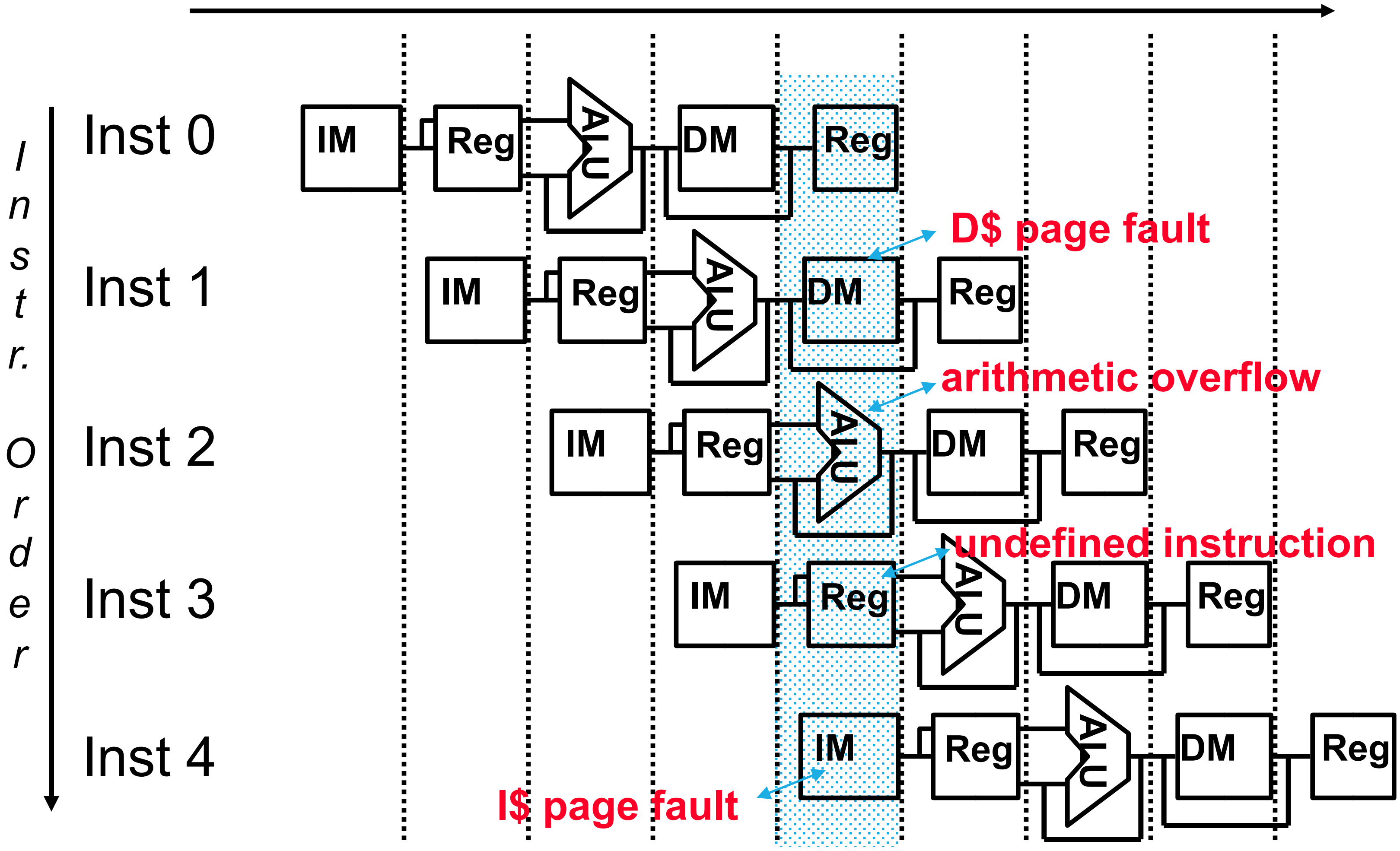
Where in the Pipeline Exceptions Occur



	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

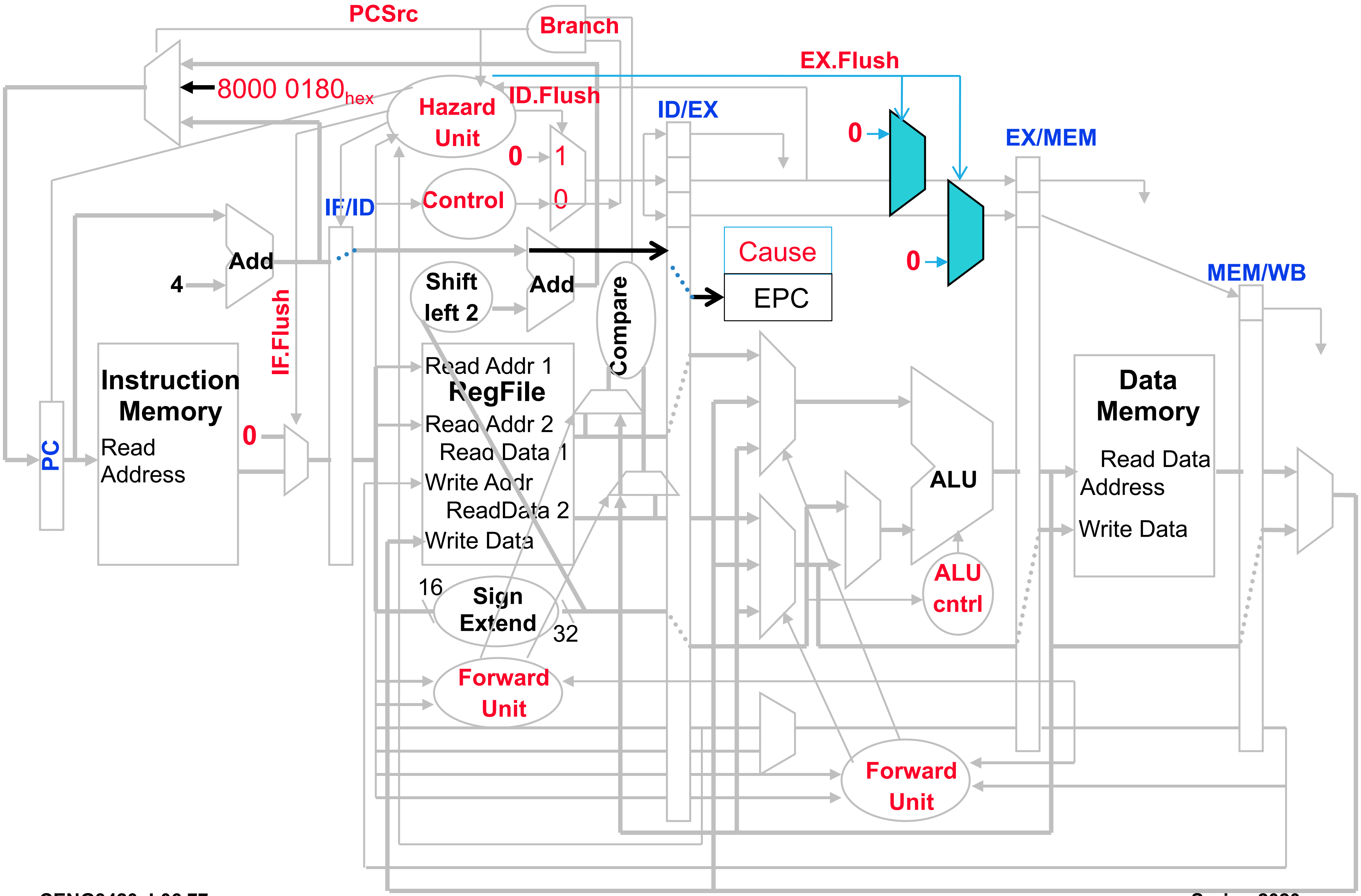


- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

Additions to MIPS to Handle Exceptions (optional)

- 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 (optional)



Summary

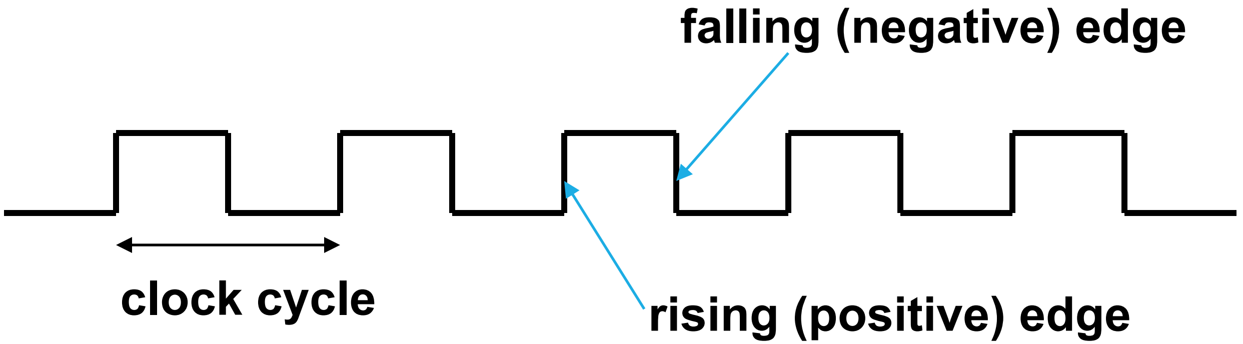
- All modern day processors use pipelining for performance (a CPI of 1 and a fast CC)
- Pipeline clock rate limited by slowest pipeline stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
    - Stall (impacts CPI)
    - Delay decision (requires compiler support)
    - Static and dynamic prediction (requires hardware support)
- Pipelining complicates exception handling

Outline

- □ Pipeline Motivations
- □ Pipeline Hazards
- □ Exceptions
- □ Background: Flip-Flop Control Signals

Clocking Methodologies

- ❑ Clocking methodology defines when signals can be read and when they can be written



clock rate = 1/(clock cycle)

e.g., 10 nsec clock cycle = 100 MHz clock rate

1 nsec clock cycle = 1 GHz clock rate

- ❑ State element design choices
  - level sensitive latch
  - master-slave and edge-triggered flipflops

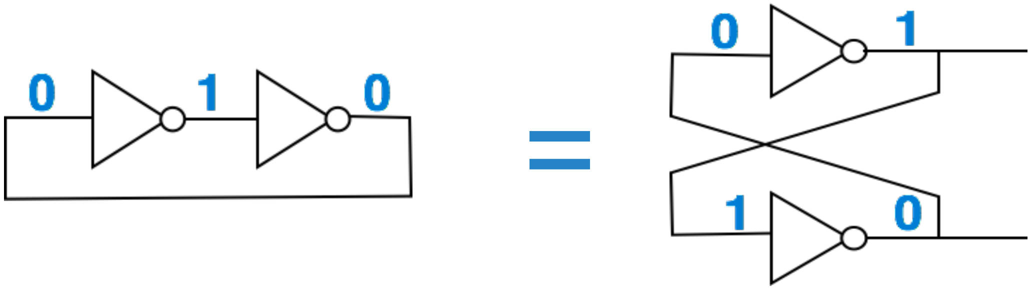
Review:Latches vs Flipflops

- Output is equal to the stored value inside the element
- Change of state (value) is based on the clock
  - Latches: output changes whenever the inputs change and the clock is asserted (level sensitive methodology)
    - Two-sided timing constraint
  - Flip-flop: output changes only on a clock edge (edge-triggered methodology)
    - One-sided timing constraint

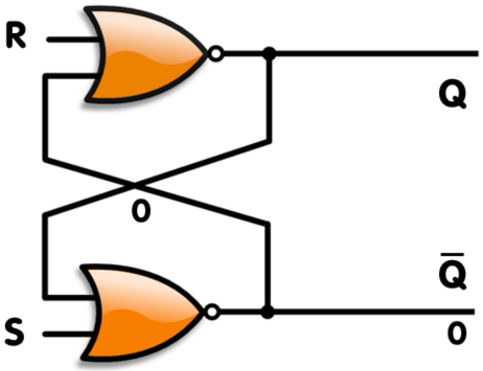
A clocking methodology defines when signals can be read and written – would NOT want to read a signal at the same time it was being written

Review: Design A Latch

- Store one bit of information: cross-coupled invertor

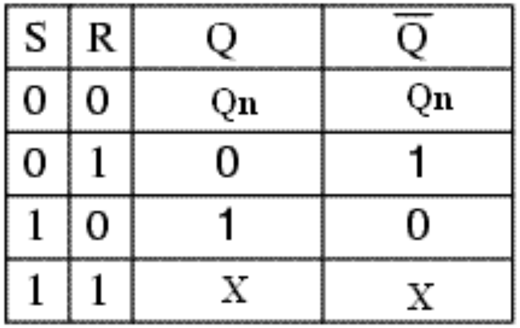


- How to change the value stored?



SR-Latch

R: reset signal
S: set signal

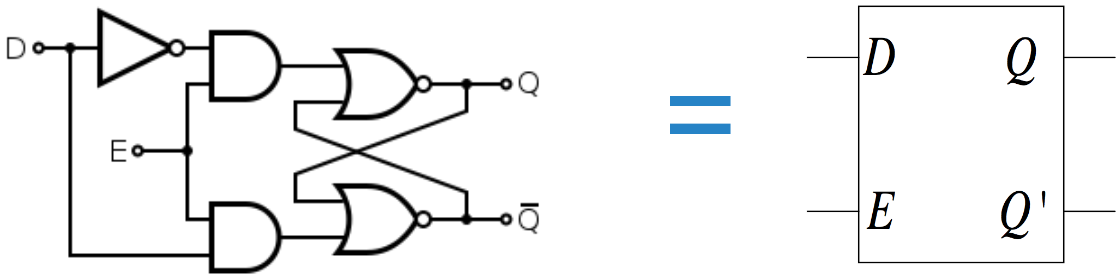


S	R	Q	$\overline{Q}$
0	0	Qn	Qn
0	1	0	1
1	0	1	0
1	1	X	X

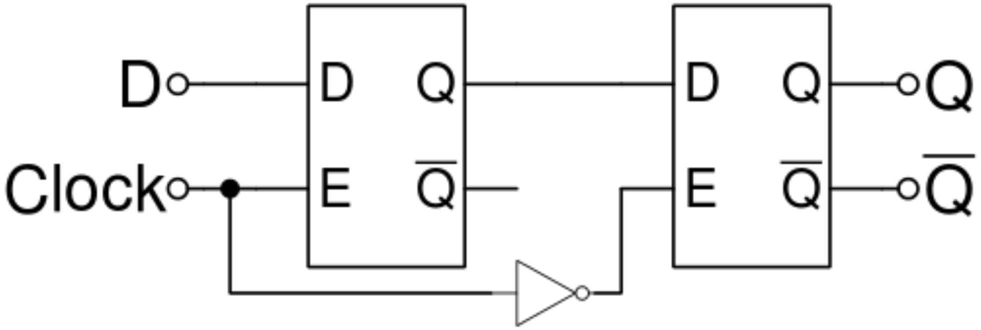
other Latch structures

Review: Design A Flip-Flop

- Based on Gated Latch

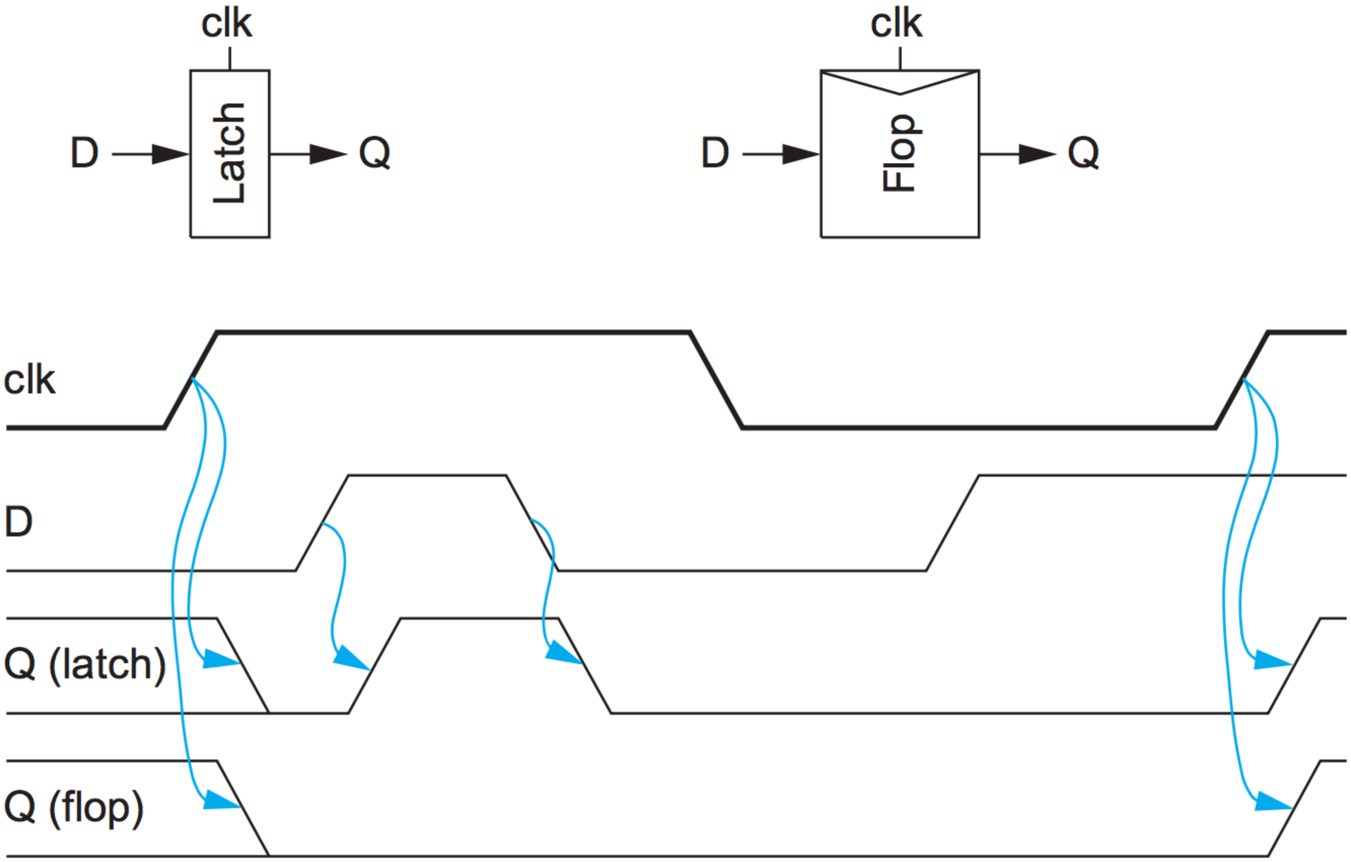


- Master-slave positive-edge-triggered D flip-flop



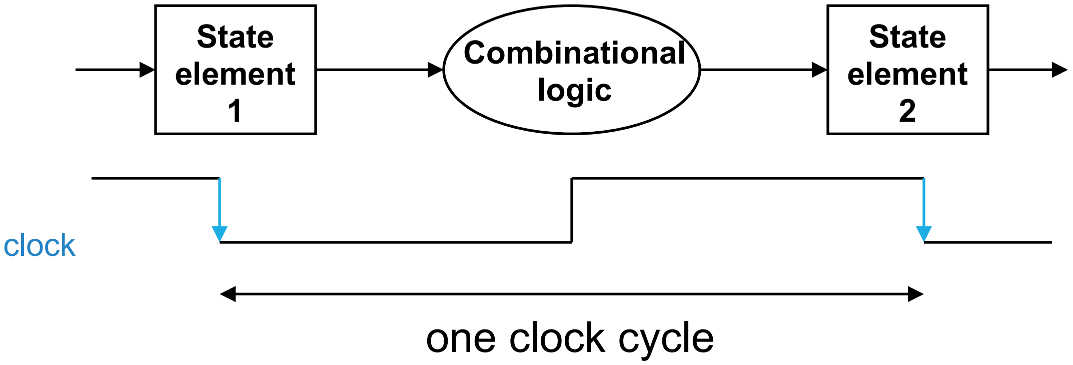
Review: Latch and Flip-Flop

- ❑ Latch is level-sensitive
- ❑ Flip-flop is edge triggered



Our Implementation

- An edge-triggered methodology
- Typical execution
  - read contents of some state elements
  - send values through some combinational logic
  - write results to one or more state elements



- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs