182.092 Computer Architecture

Chapter 4: The Processor

Adapted from

Computer Organization and Design, 4th Edition,

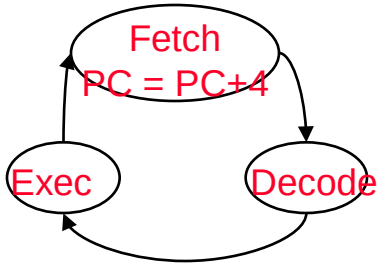
Patterson & Hennessy, © 2008, Morgan Kaufmann Publishers

and

Mary Jane Irwin (www.cse.psu.edu/research/mdl/mji)

The Processor: Datapath & Control

- Our implementation of the MIPS is simplified
  - memory-reference instructions: **`lw, sw`**
  - arithmetic-logical instructions: **`add, sub, and, or, slt`**
  - control flow instructions: **`beq, j`**
- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



- All instructions (except **`j`**) use the ALU after reading the registers

How? memory-reference? arithmetic? control flow?

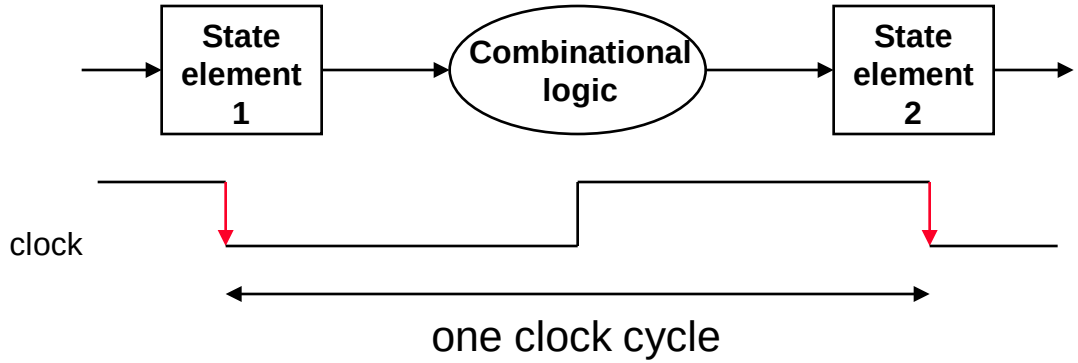
Aside: Clocking Methodologies

- The clocking methodology defines when data in a state element is valid and stable relative to the clock
  - State elements - a memory element such as a register
  - Edge-triggered – all state changes occur on a clock edge

Latch

Flip-Flop

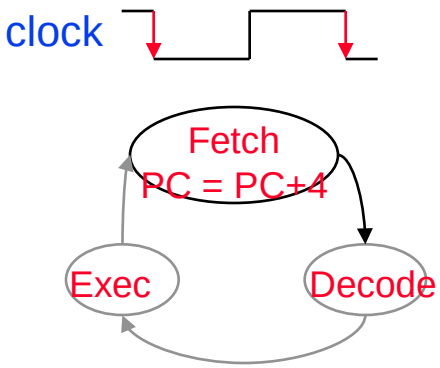
- Typical execution
  - read contents of state elements ➔ send values through combinational logic ➔ write results to one or more state elements

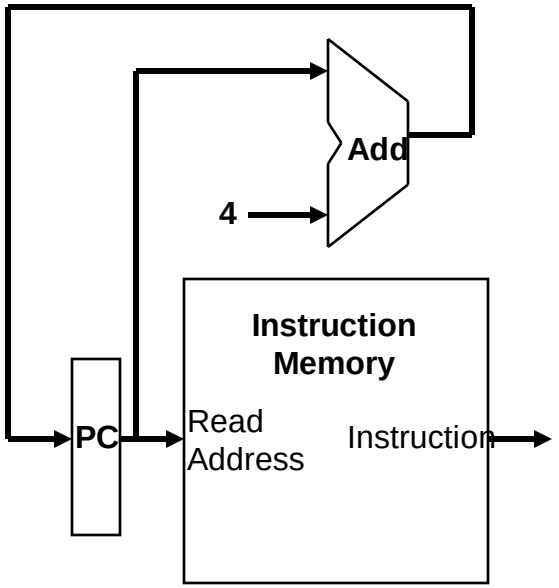


- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - write occurs only when both the write control is asserted and the clock edge occurs

Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction

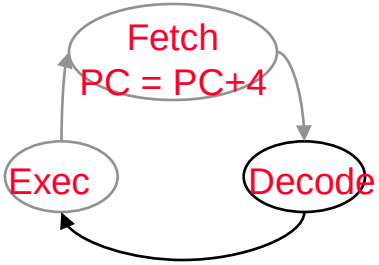




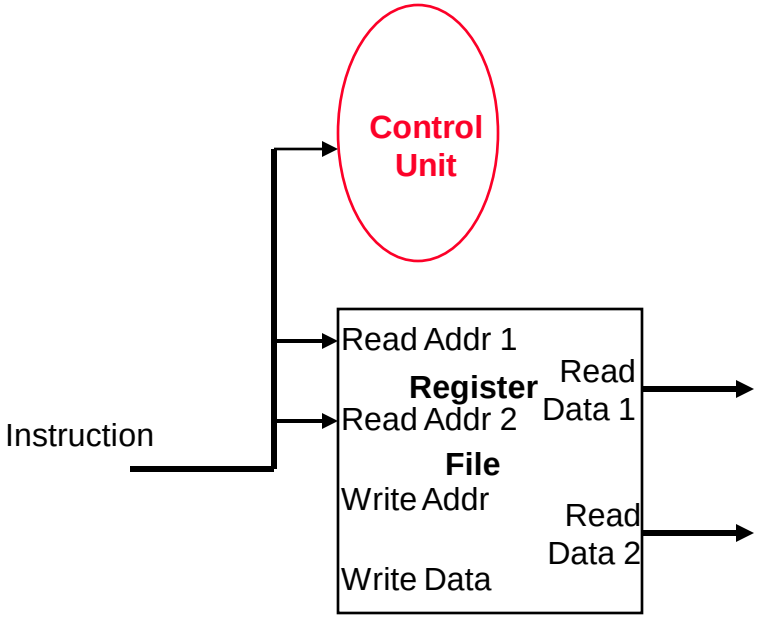
- PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
- Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit



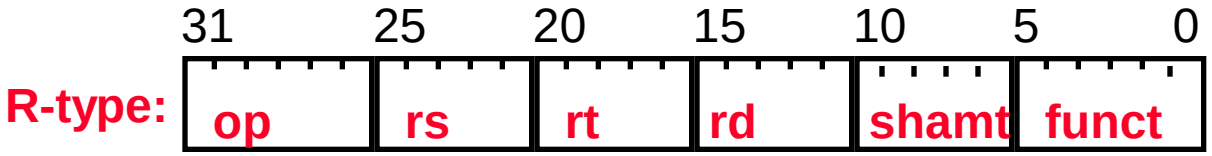
and



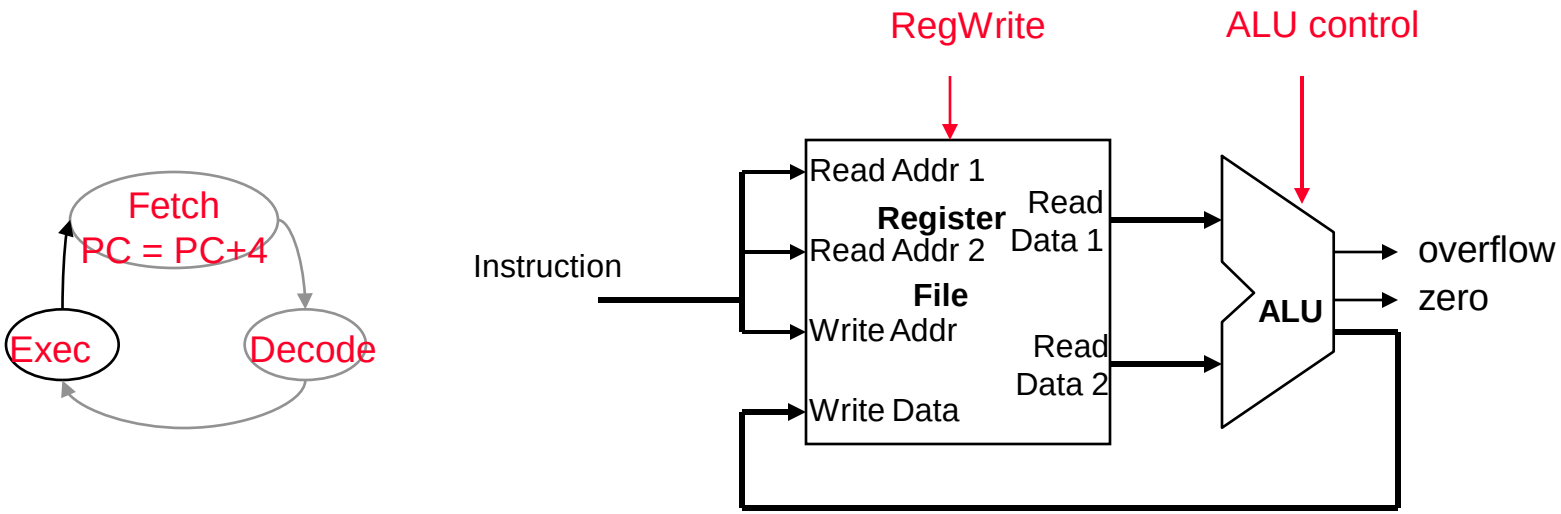
  - reading two values from the Register File
    - Register File addresses are contained in the instruction

Executing R Format Operations

- R format operations (**`add, sub, slt, and, or`**)



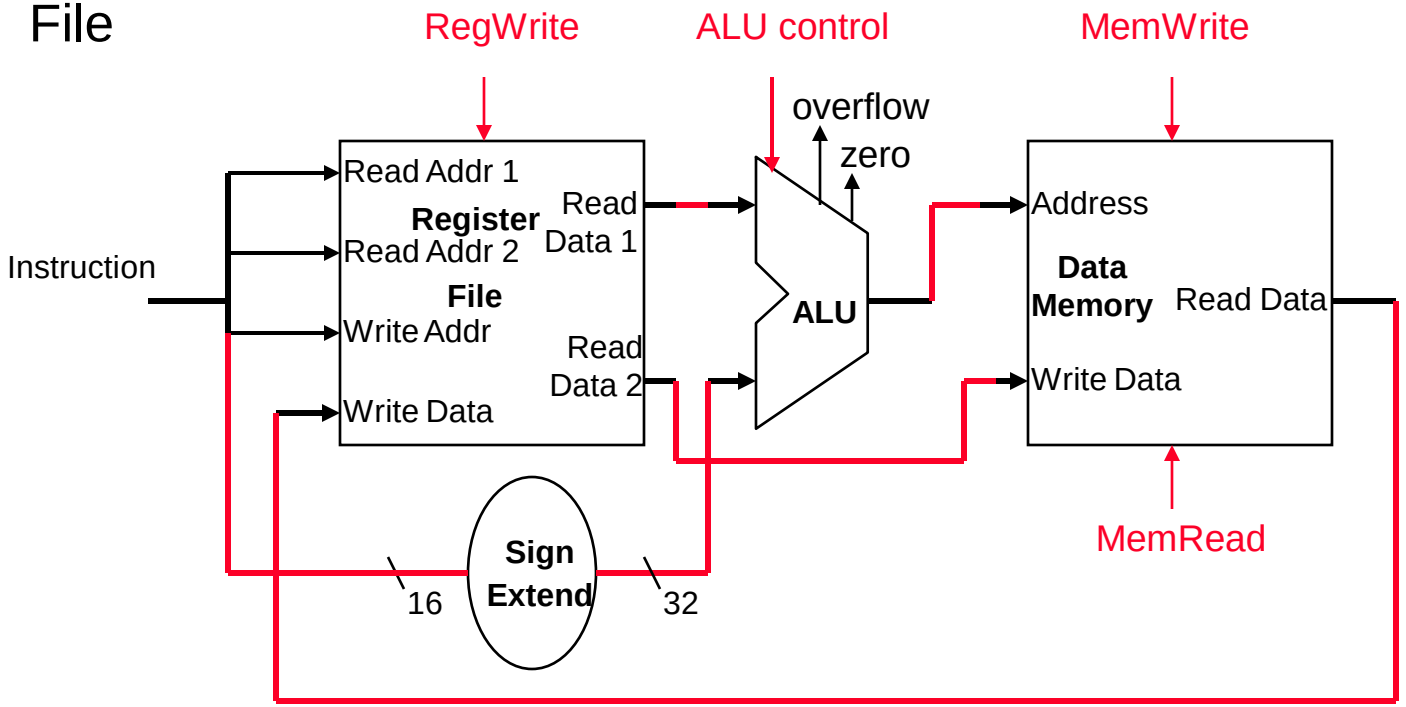
- perform operation (op and funct) on values in rs and rt
- store the result back into the Register File (into location rd)



- Note that Register File is not written every cycle (e.g. **`sw`**), so we need an explicit write control signal for the Register File

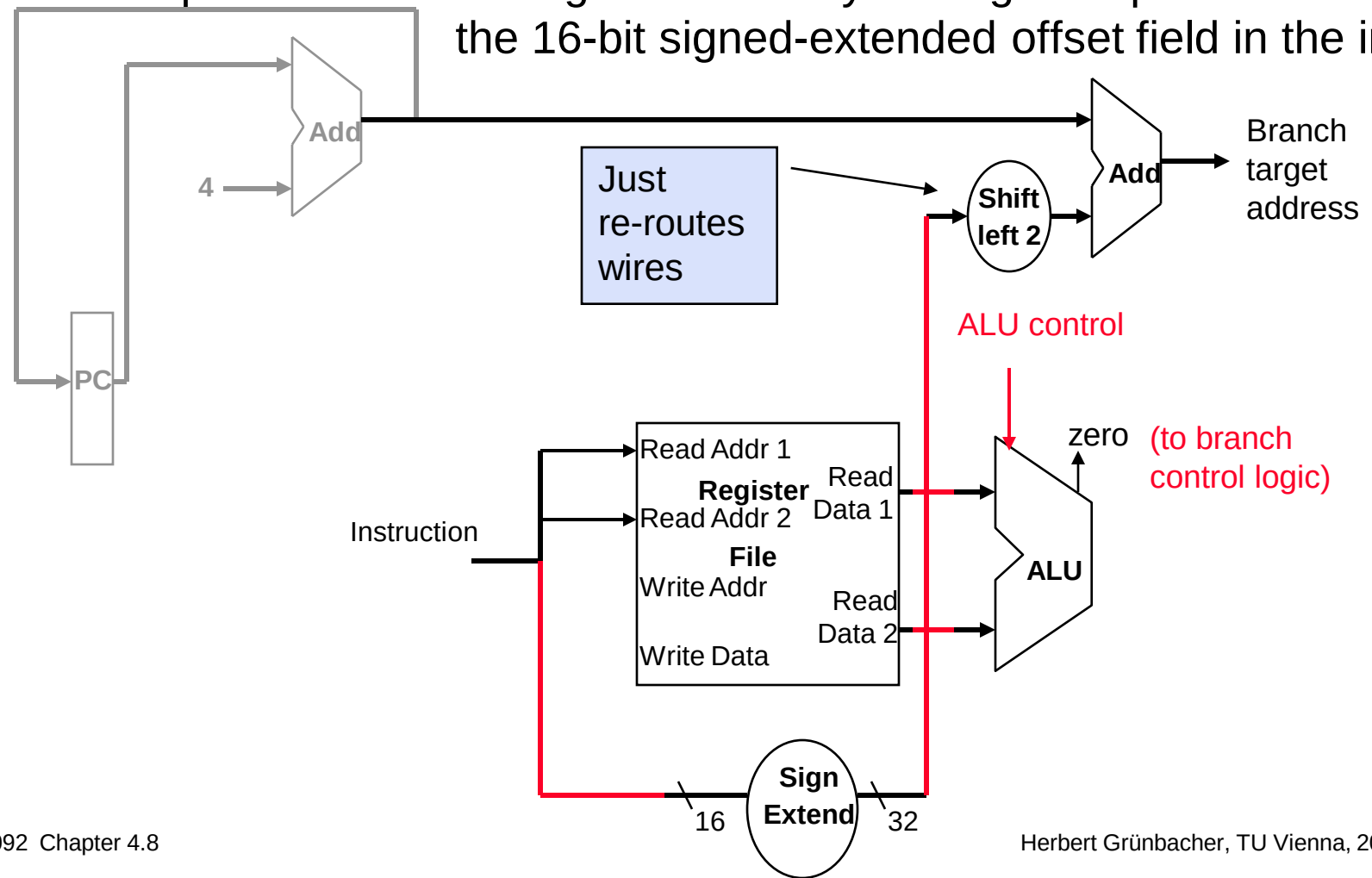
Executing Load and Store Operations

- Load and store operations involves
  - compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
  - store value (read from the Register File during decode) written to the Data Memory
  - load value, read from the Data Memory, written to the Register File



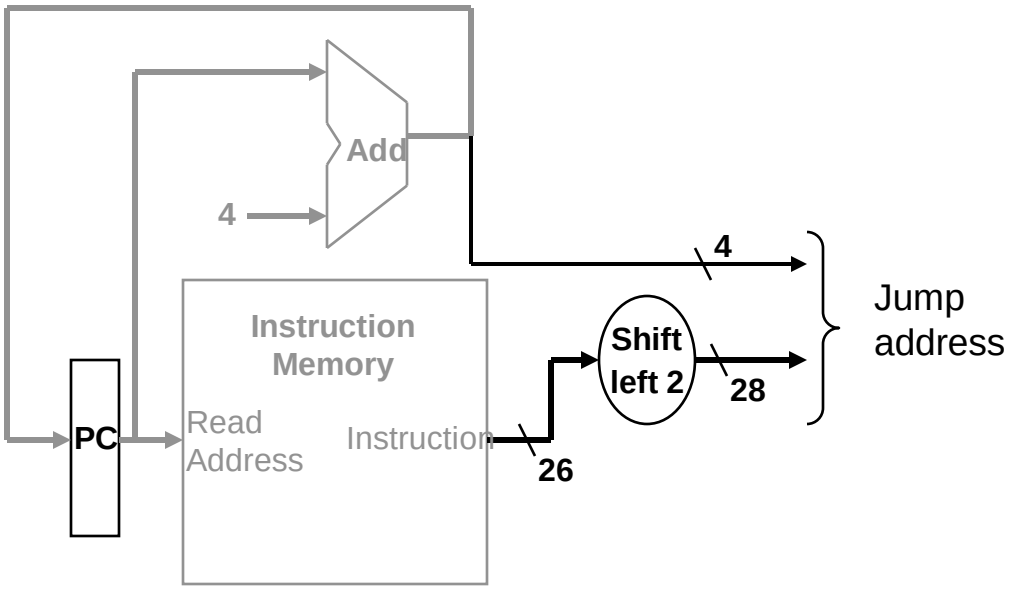
Executing Branch Operations

- Branch operations involves
  - compare the operands read from the Register File during decode for equality (**`zero`** ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



Executing Jump Operations

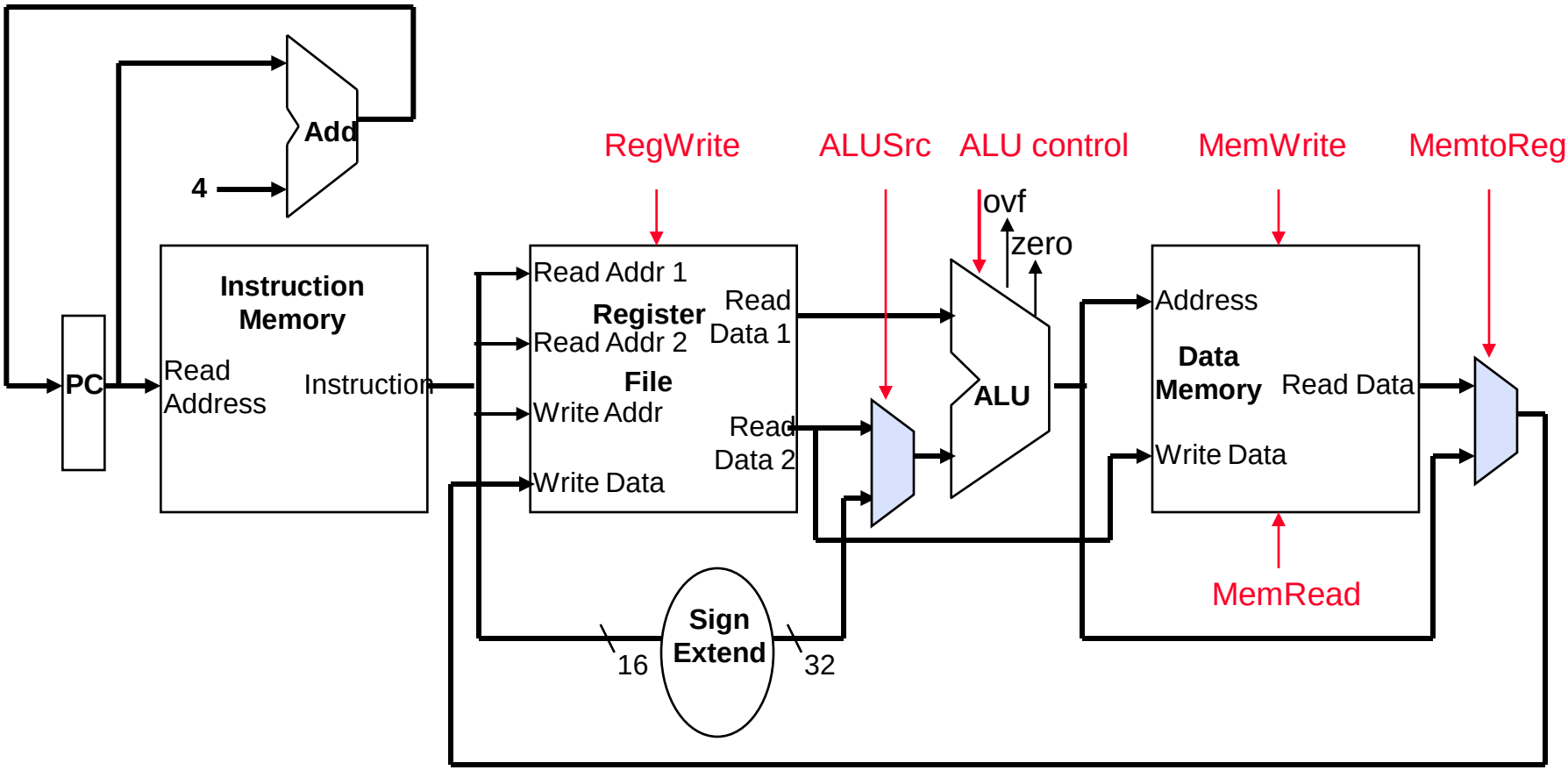
- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



Creating a Single Datapath from the Parts

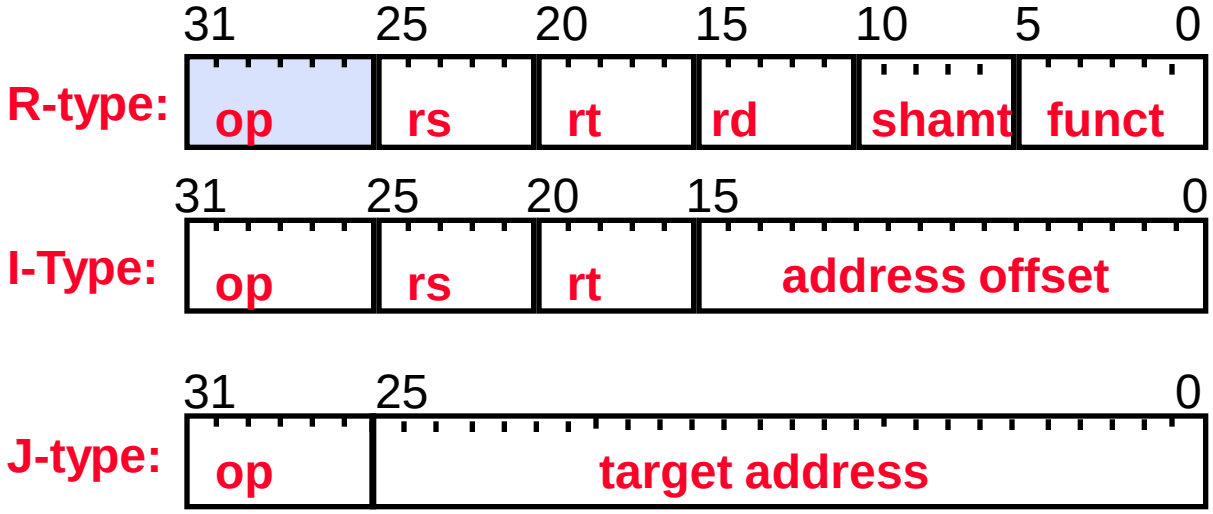
- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

Fetch, R, and Memory Access Portions



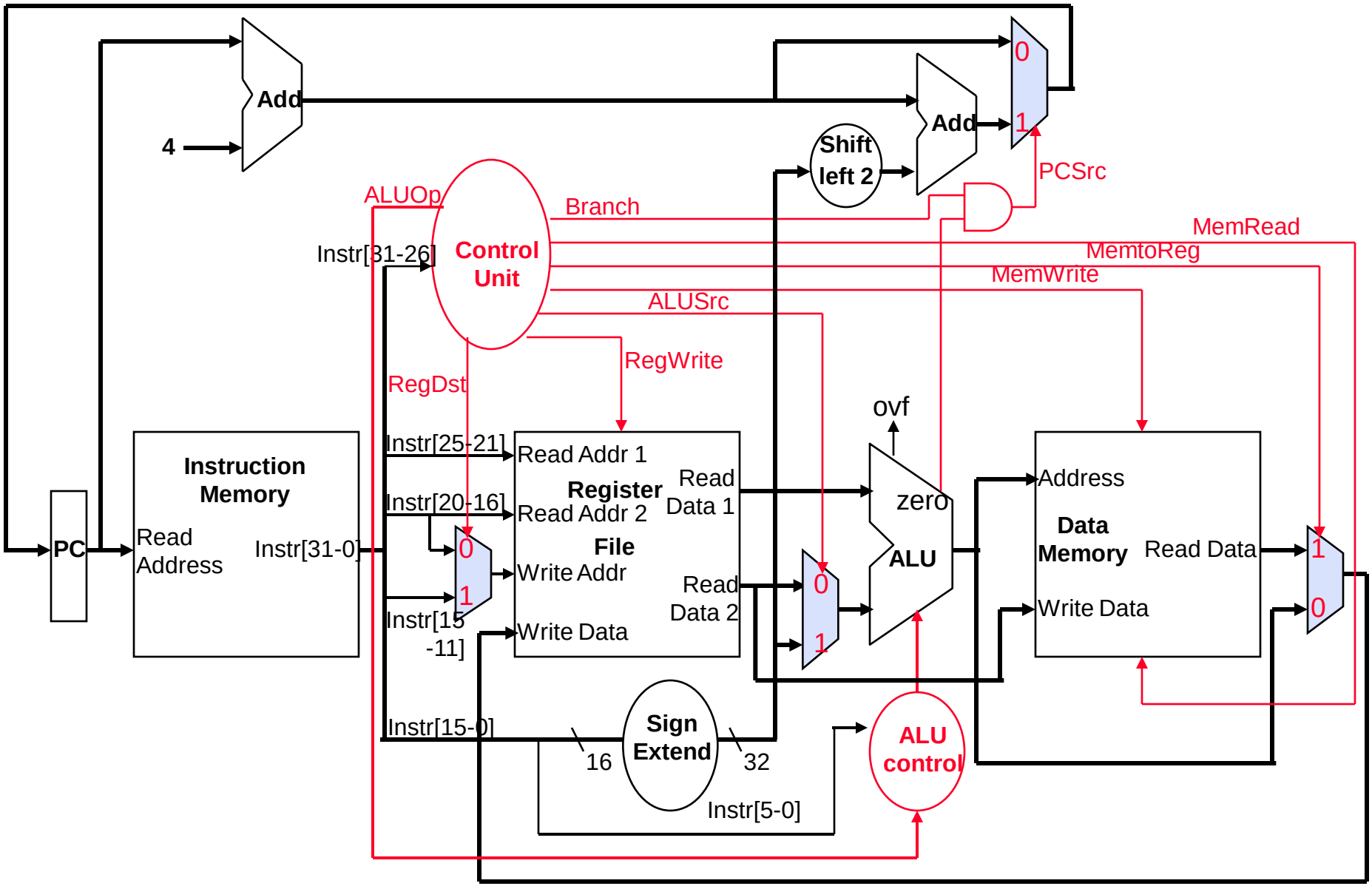
Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)

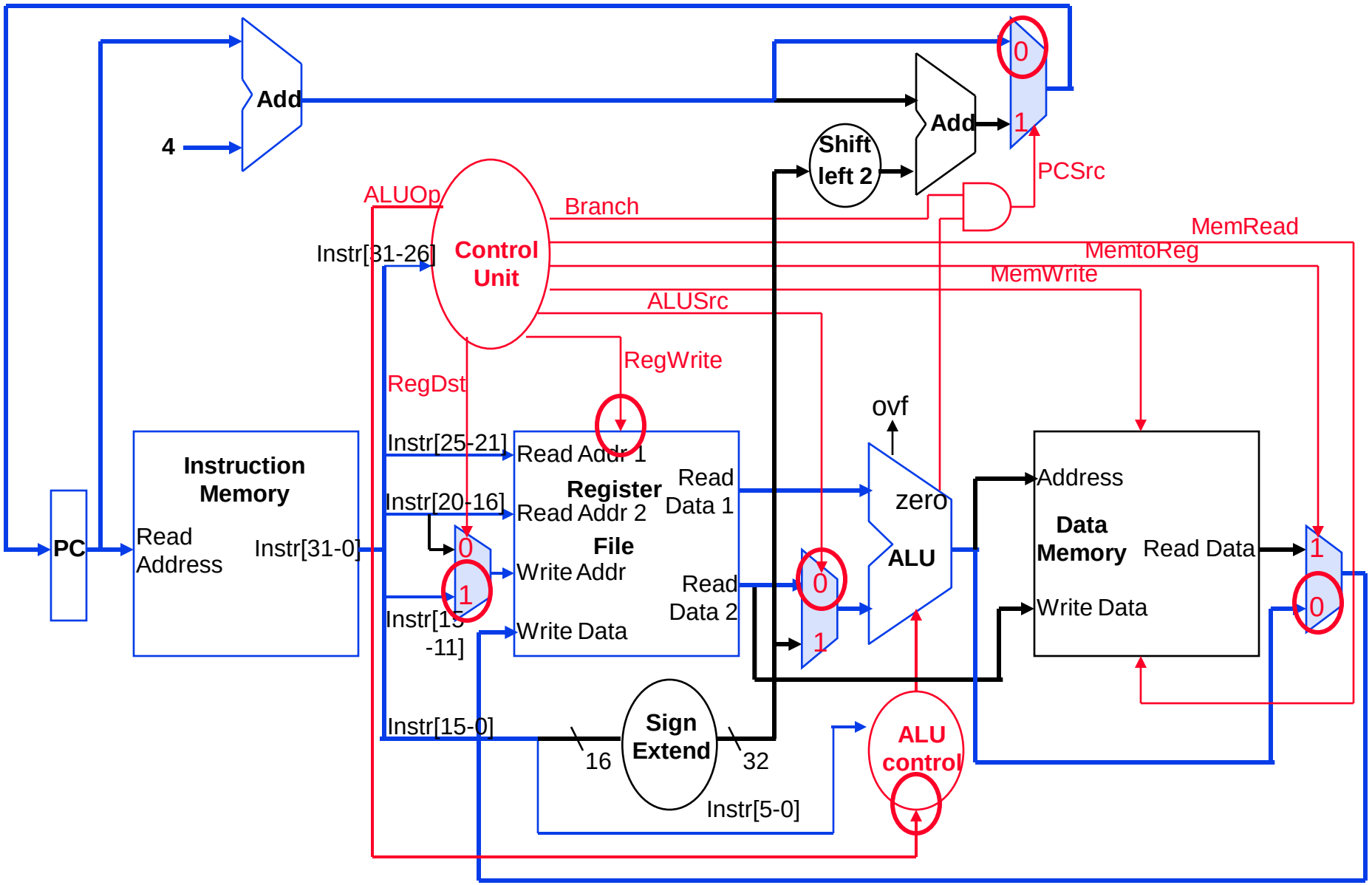


- Observations
  - op field always in bits 31-26
  - addr of registers to be read are always specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
  - addr. of register to be written is in one of two places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
  - offset for beq, lw, and sw always in bits 15-0

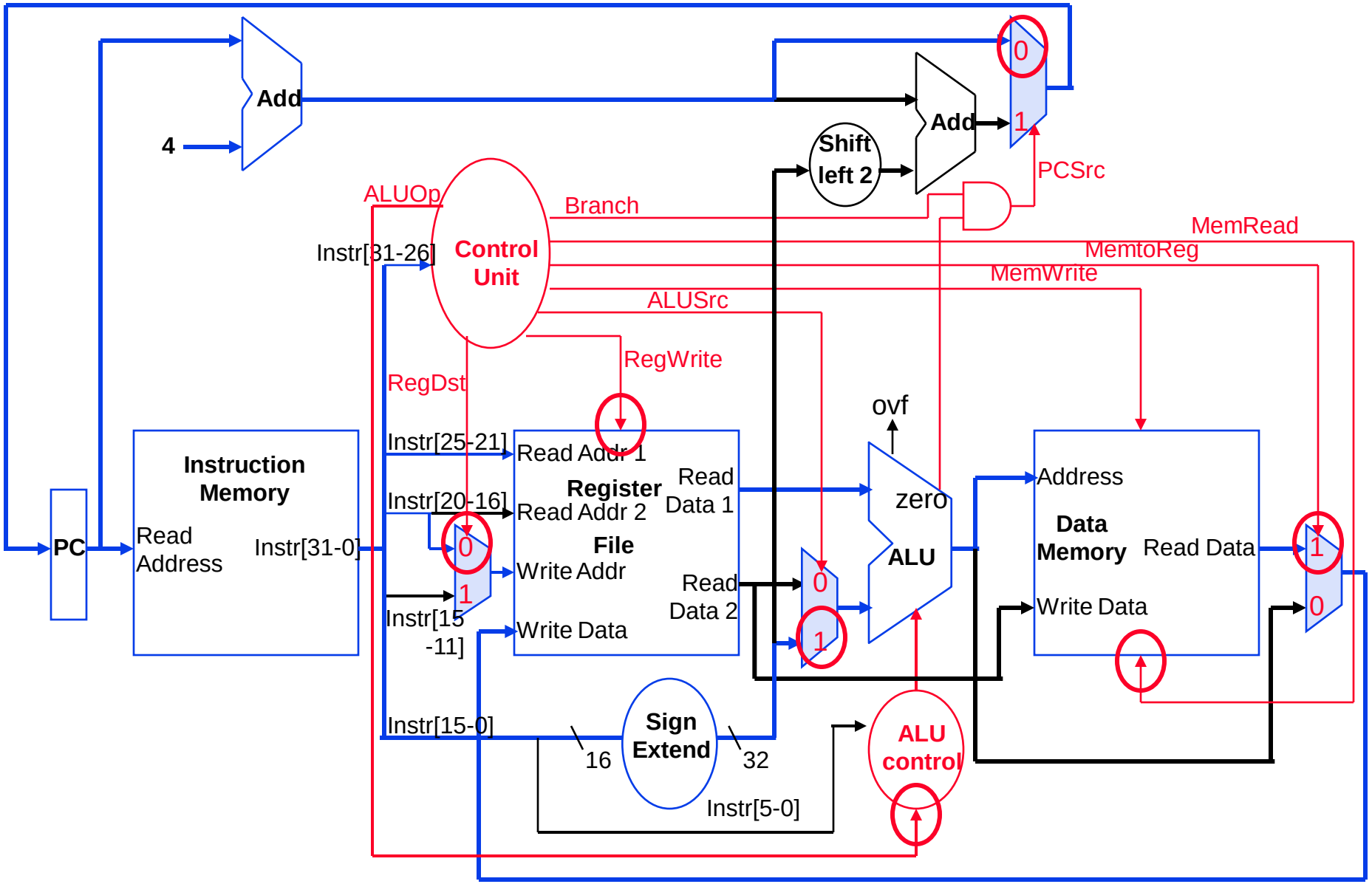
Single Cycle Datapath with Control Unit



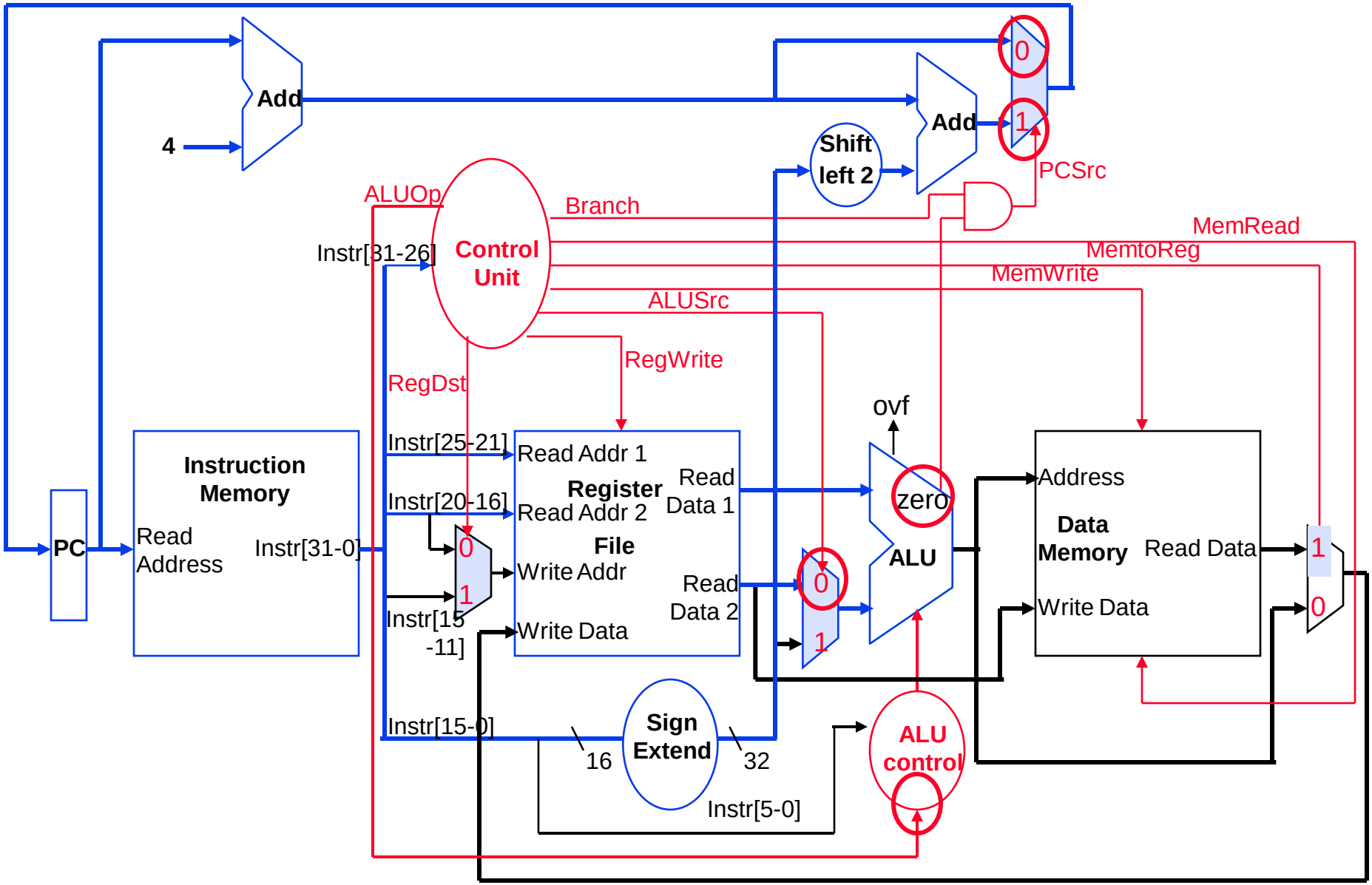
R-type Instruction Data/Control Flow



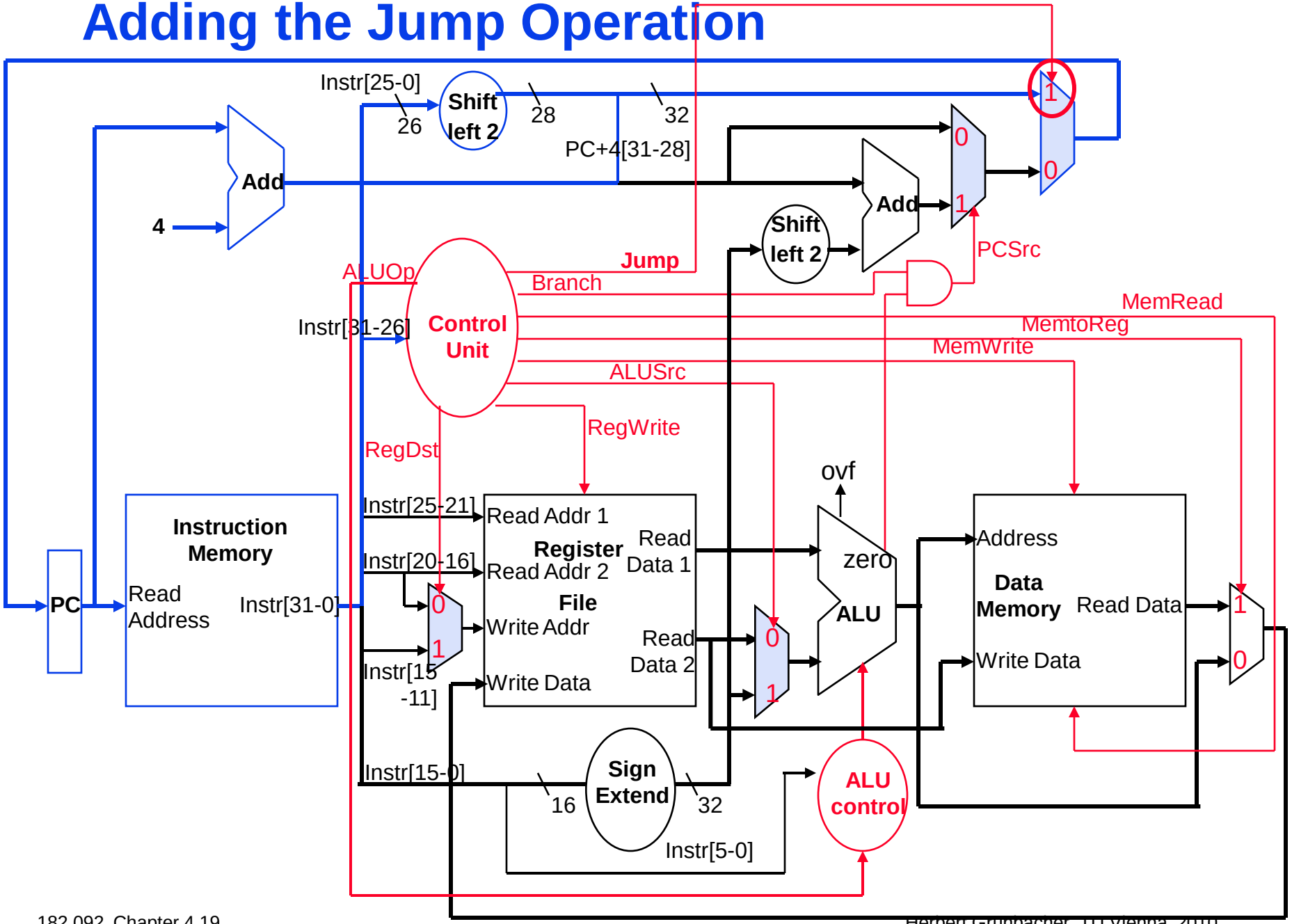
Load Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Adding the Jump Operation



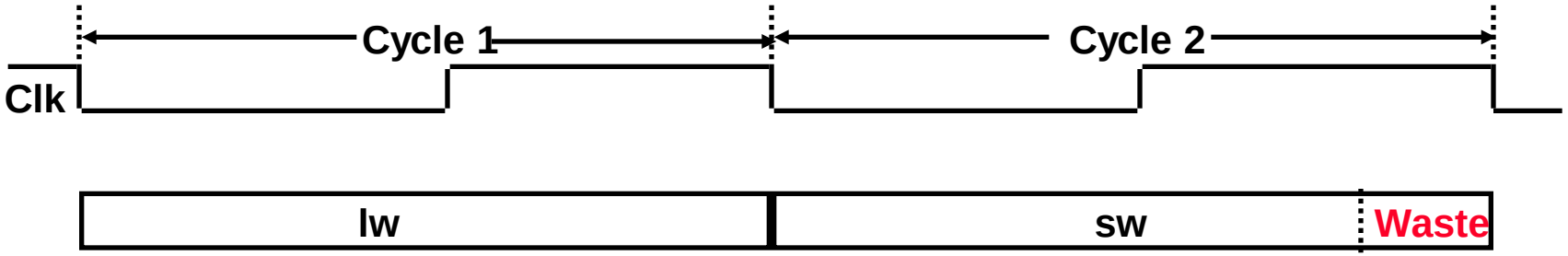
Instruction Critical Paths

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - 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

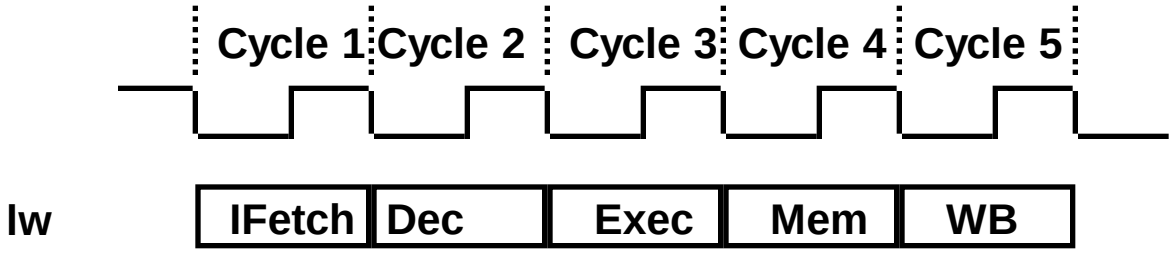
- Is simple and easy to understand

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

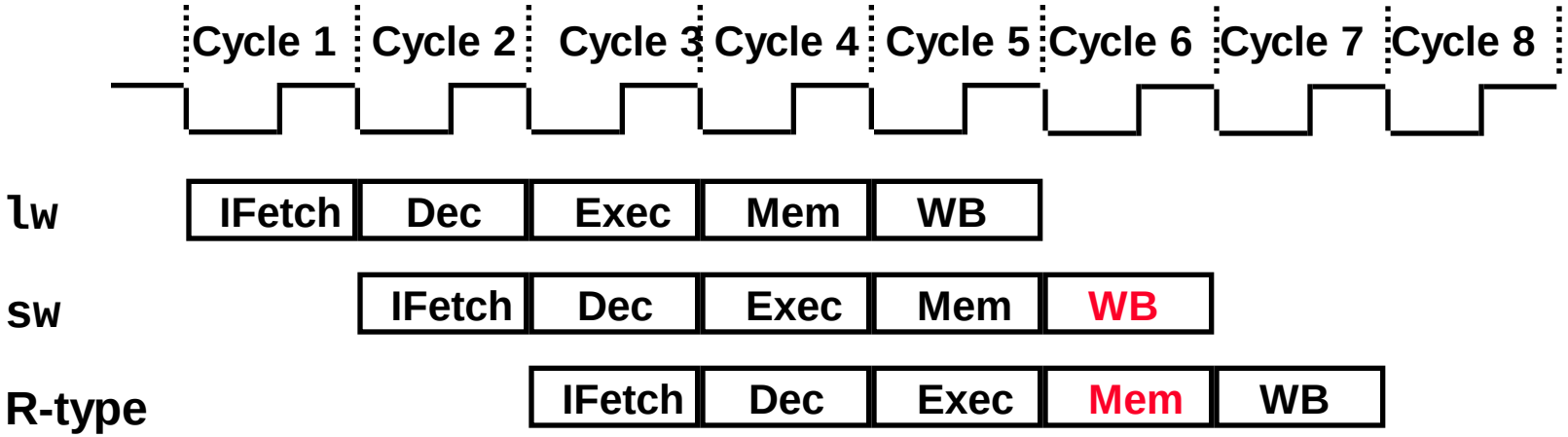
The Five Stages of Load Instruction



- 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

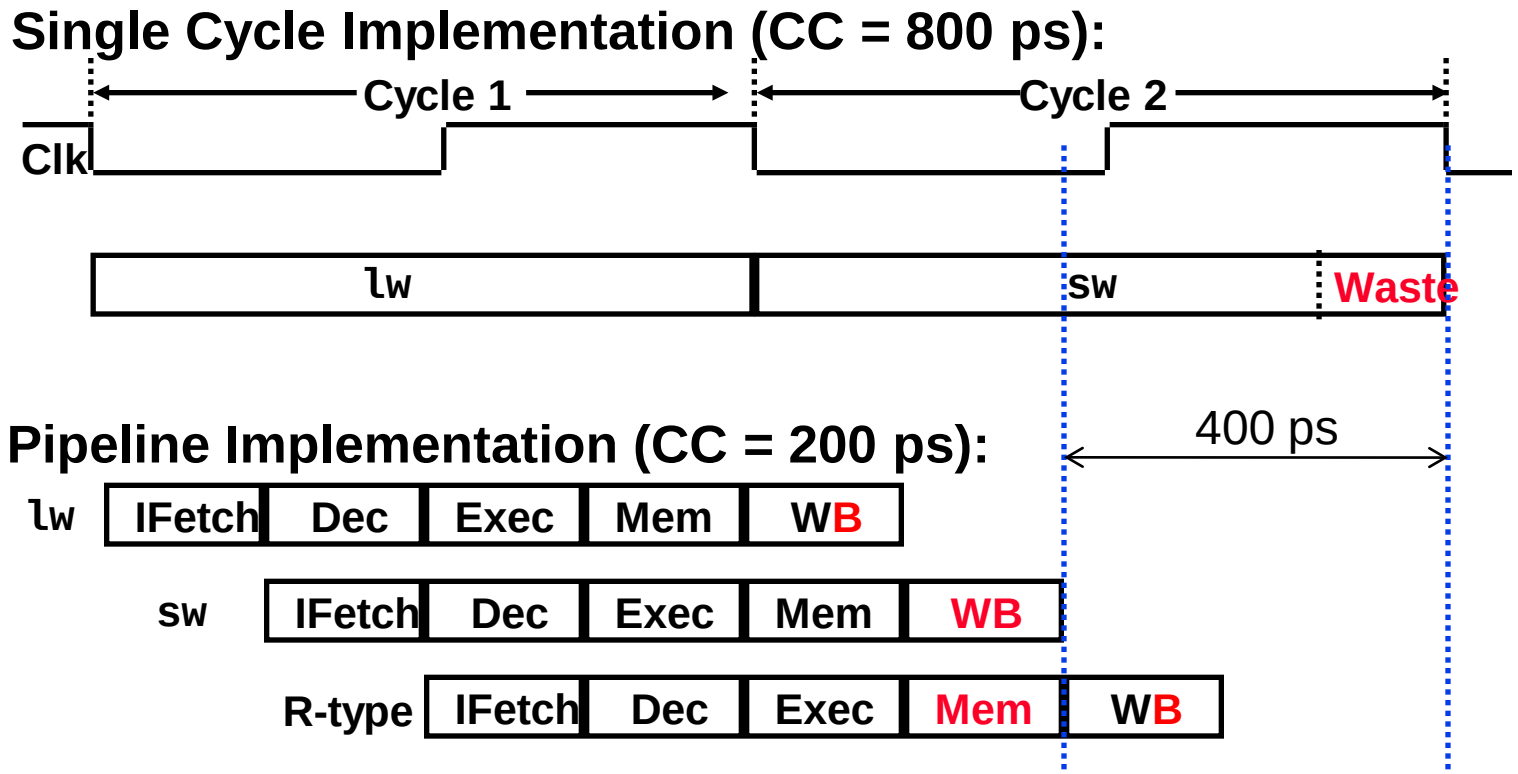
A Pipelined MIPS Processor

- 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



  - 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



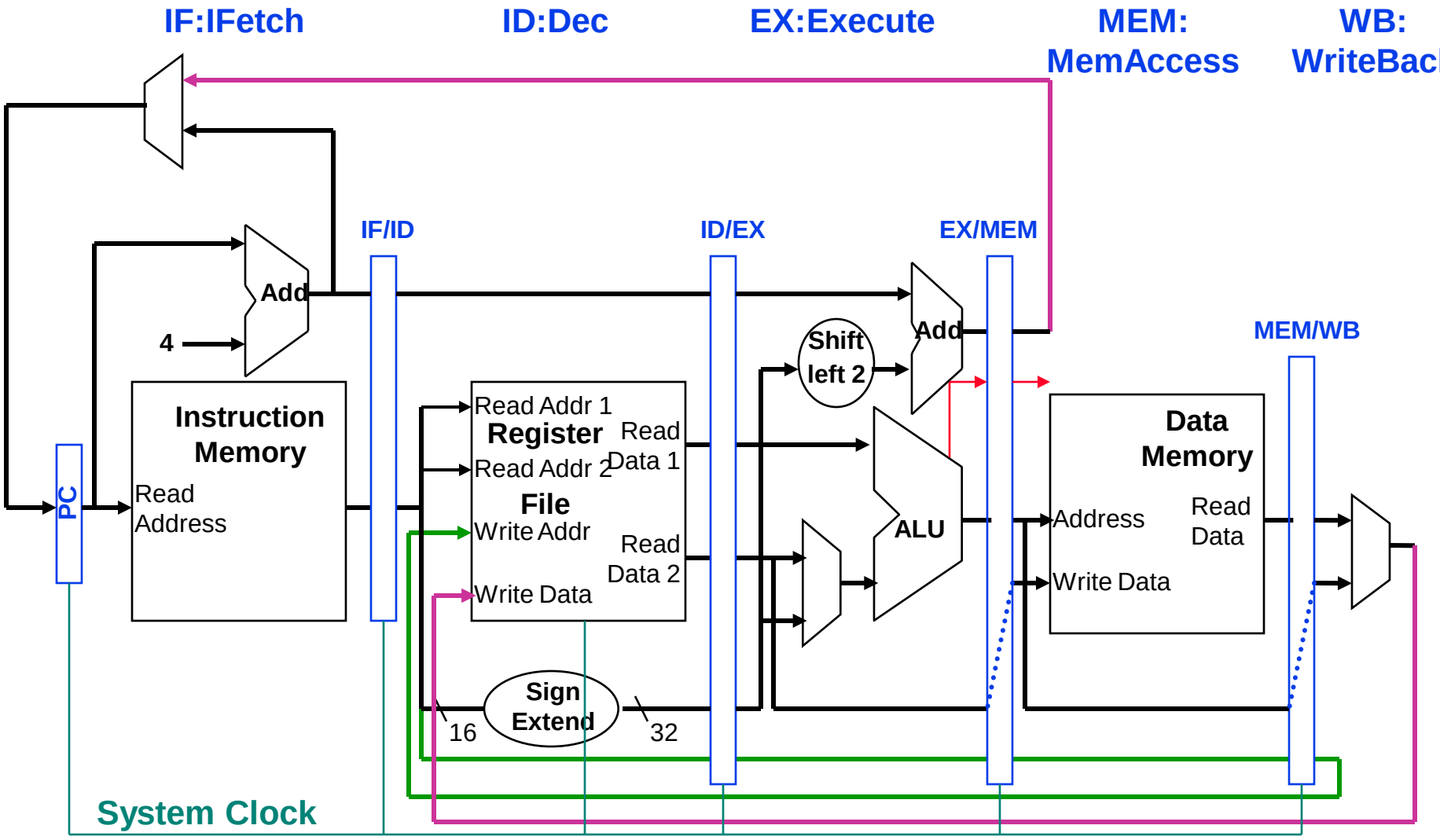
- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?

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

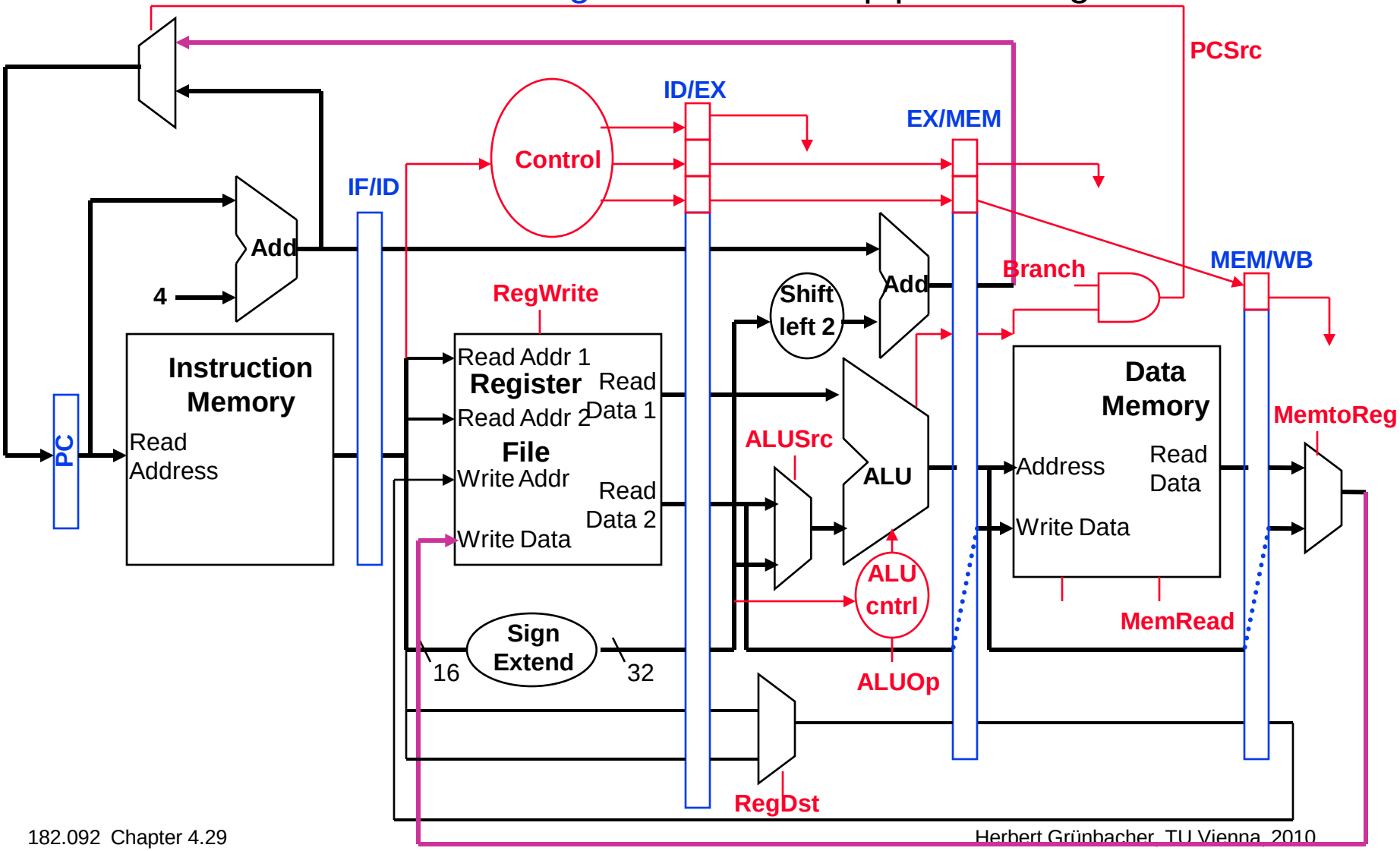
MIPS Pipeline Datapath Additions/Mods

- 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

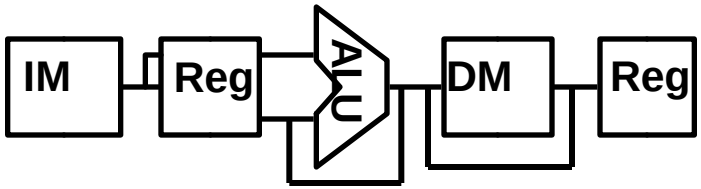


Pipeline Control

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

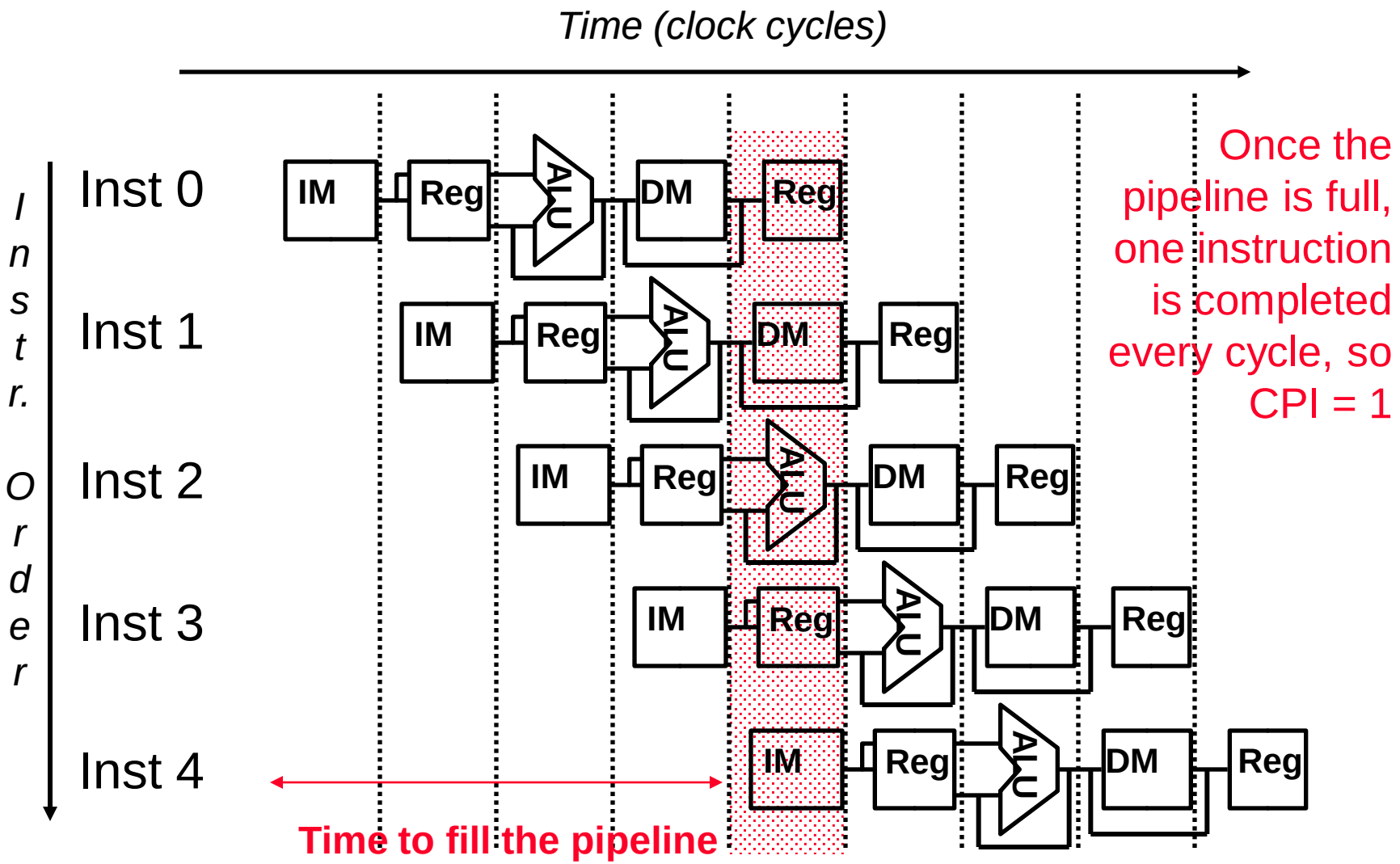
	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

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?

Why Pipeline? For Performance!

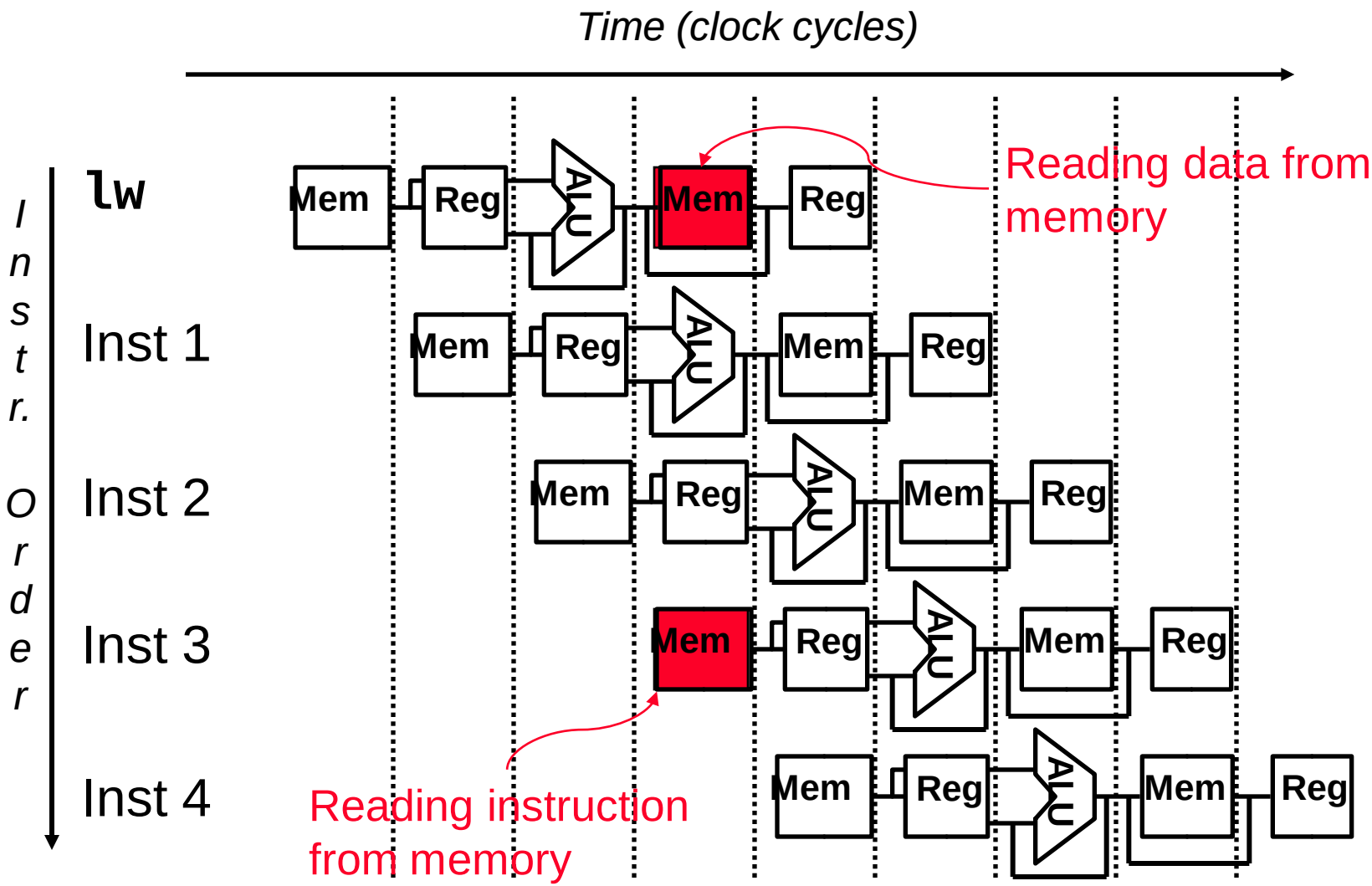


Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions

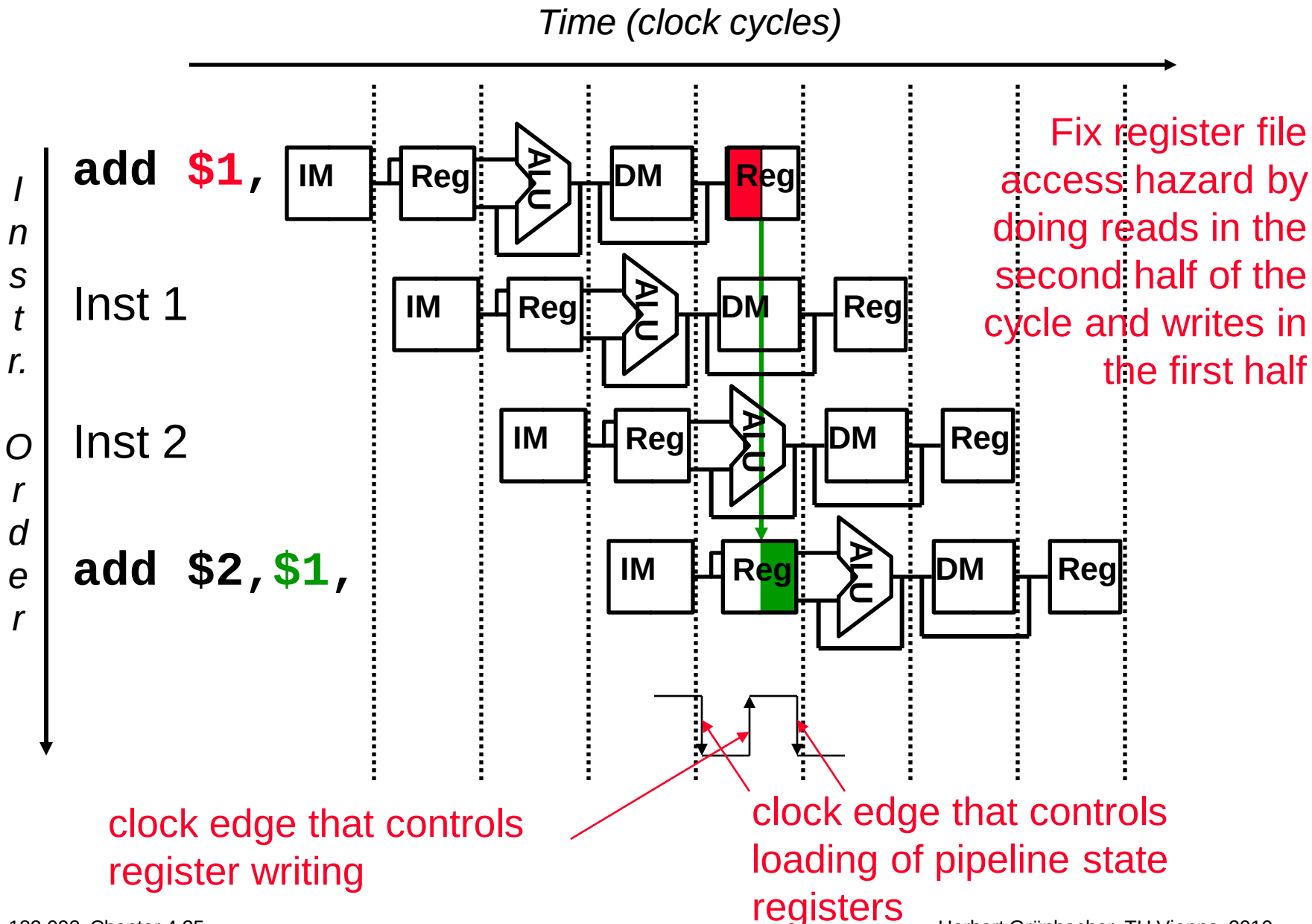
- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

A Single Memory Would Be a Structural Hazard



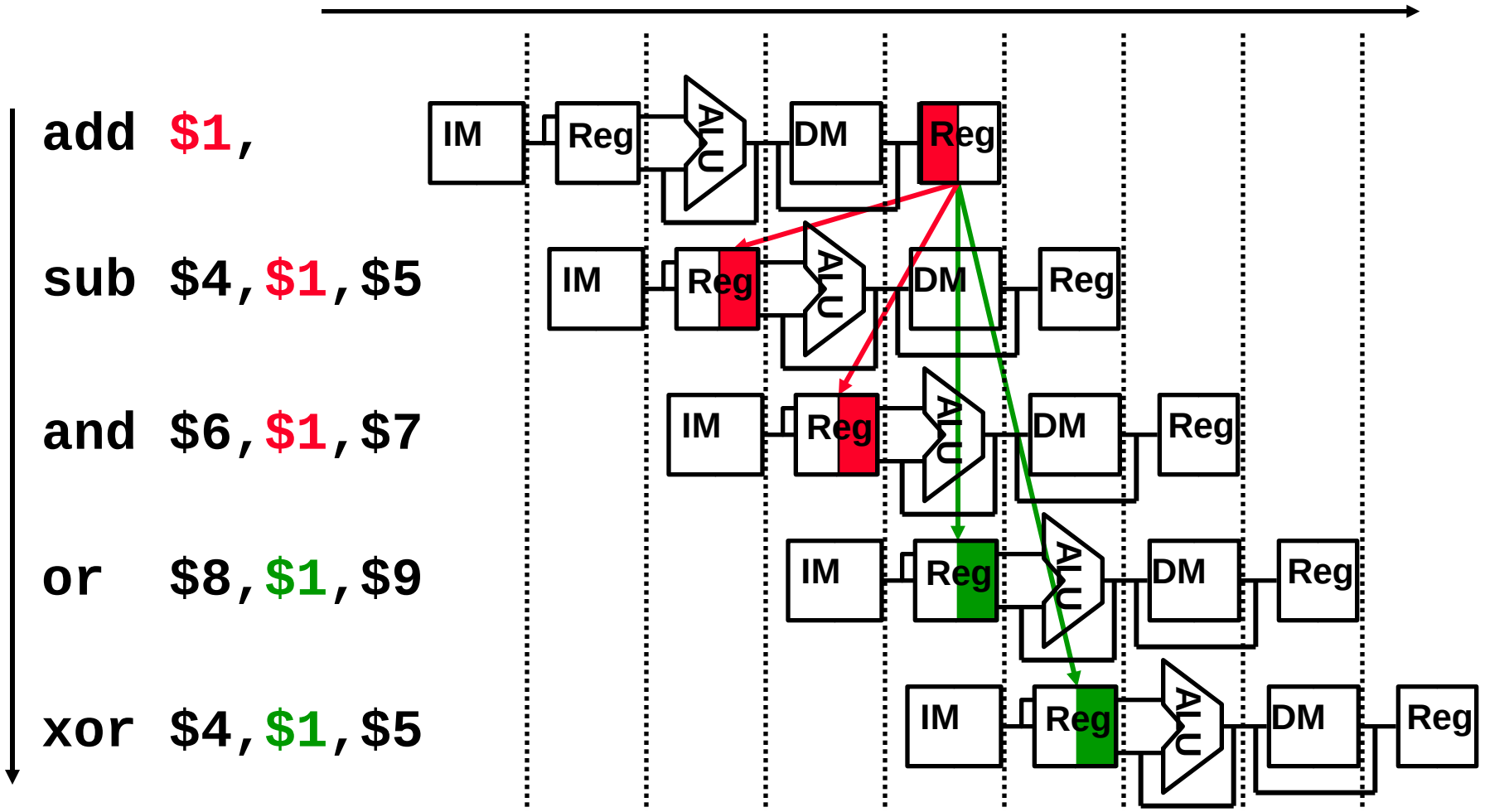
- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



Register Usage Can Cause Data Hazards

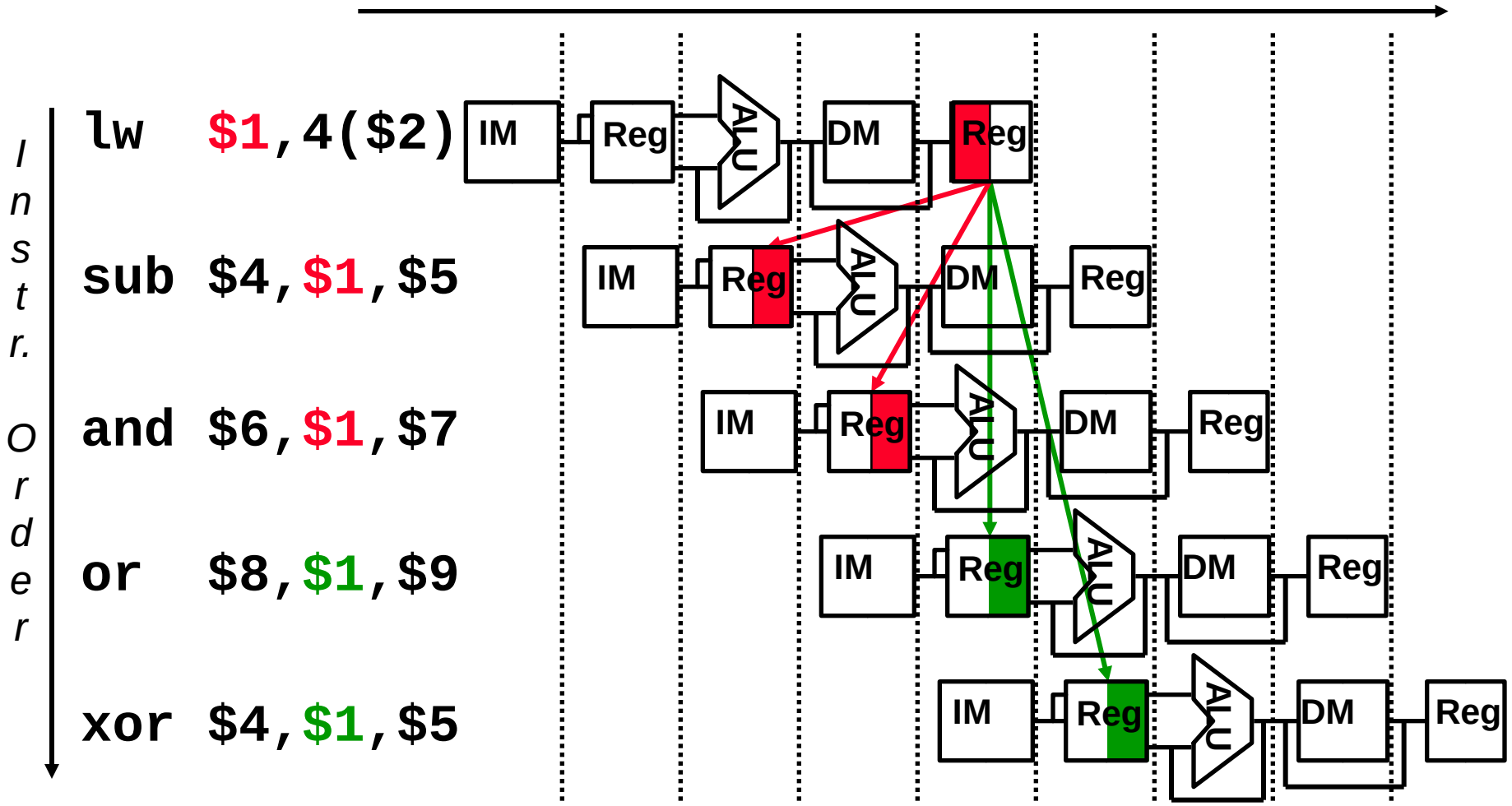
- Dependencies backward in time cause hazards



- Read before write data hazard

Loads Can Cause Data Hazards

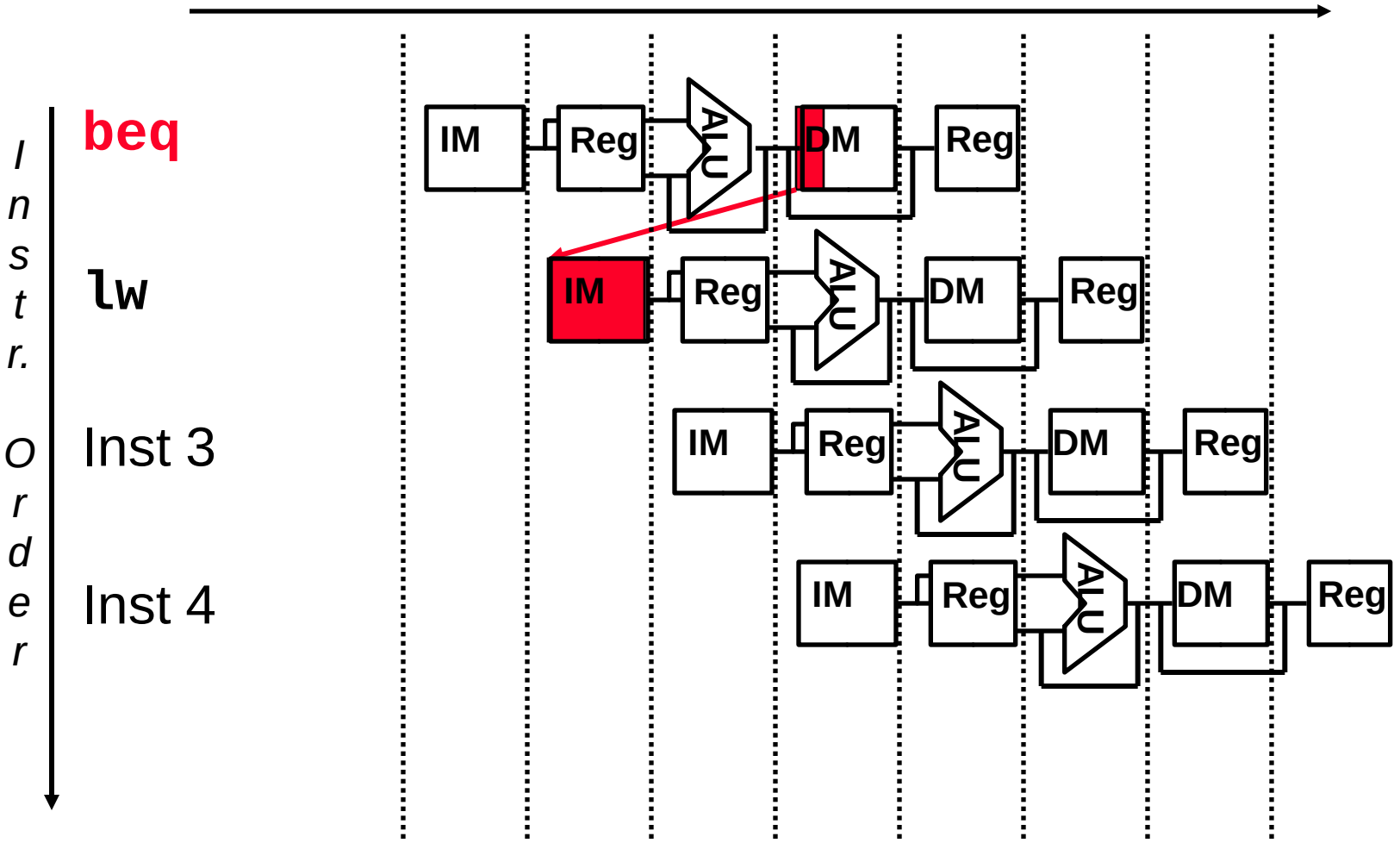
- Dependencies backward in time cause hazards



- Load-use data hazard

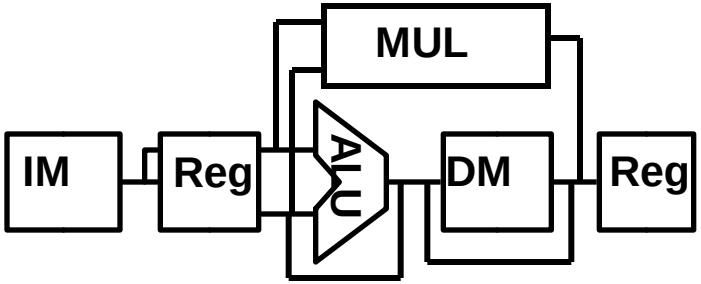
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards

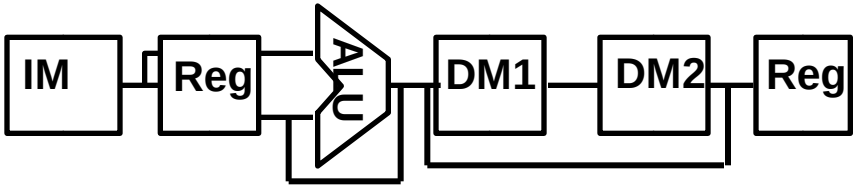


Other Pipeline Structures Are Possible

- 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)



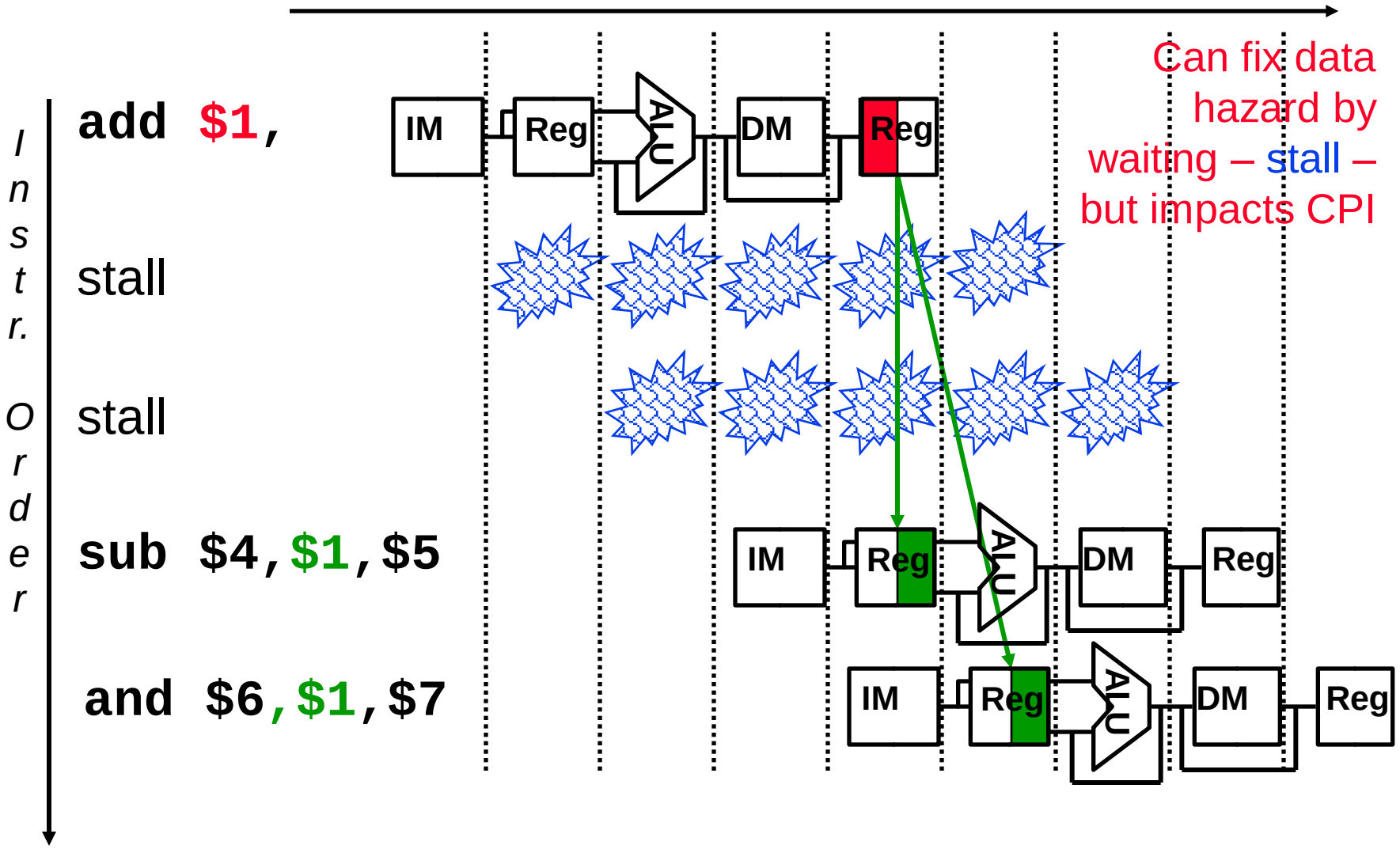
- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)



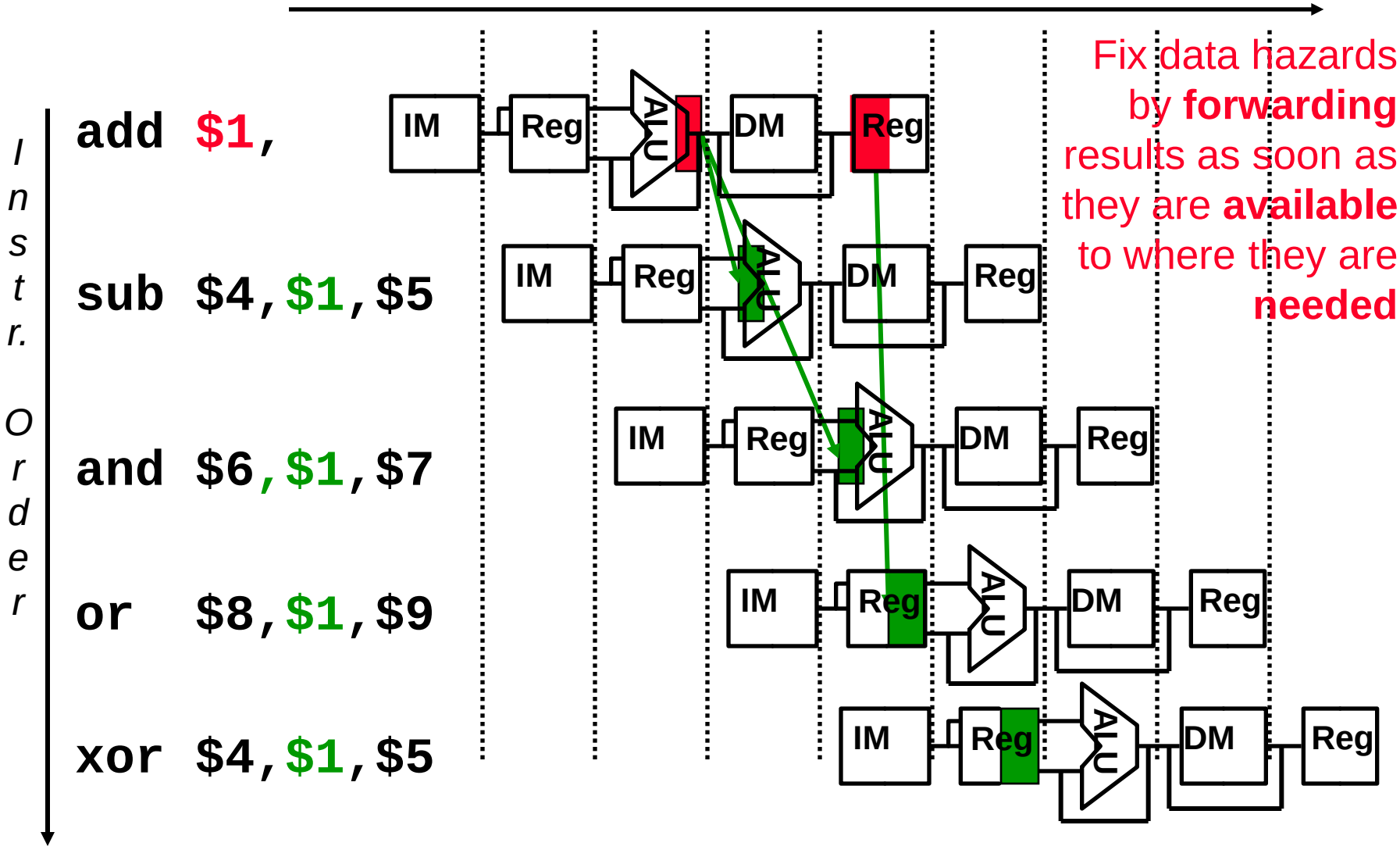
Summary

- All modern day processors use pipelining
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and fast a CC
- Pipeline rate limited by slowest pipeline stage
  - Unbalanced pipe stages makes for inefficiencies
  - The time to "fill" pipeline and time to "drain" it can impact speedup for deep pipelines and short code runs
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)

One Way to "Fix" a Data Hazard



Another Way to "Fix" a Data Hazard



Data Forwarding (aka Bypassing)

- Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- For ALU functional unit: the inputs can come from any pipeline register rather than just from ID/EX by
  - adding multiplexors to the inputs of the ALU
  - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs
  - adding the proper control hardware to control the new muxes
- Other functional units may need similar forwarding logic (e.g., the DM)
- With forwarding can achieve a CPI of 1 even in the presence of data dependencies

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
```

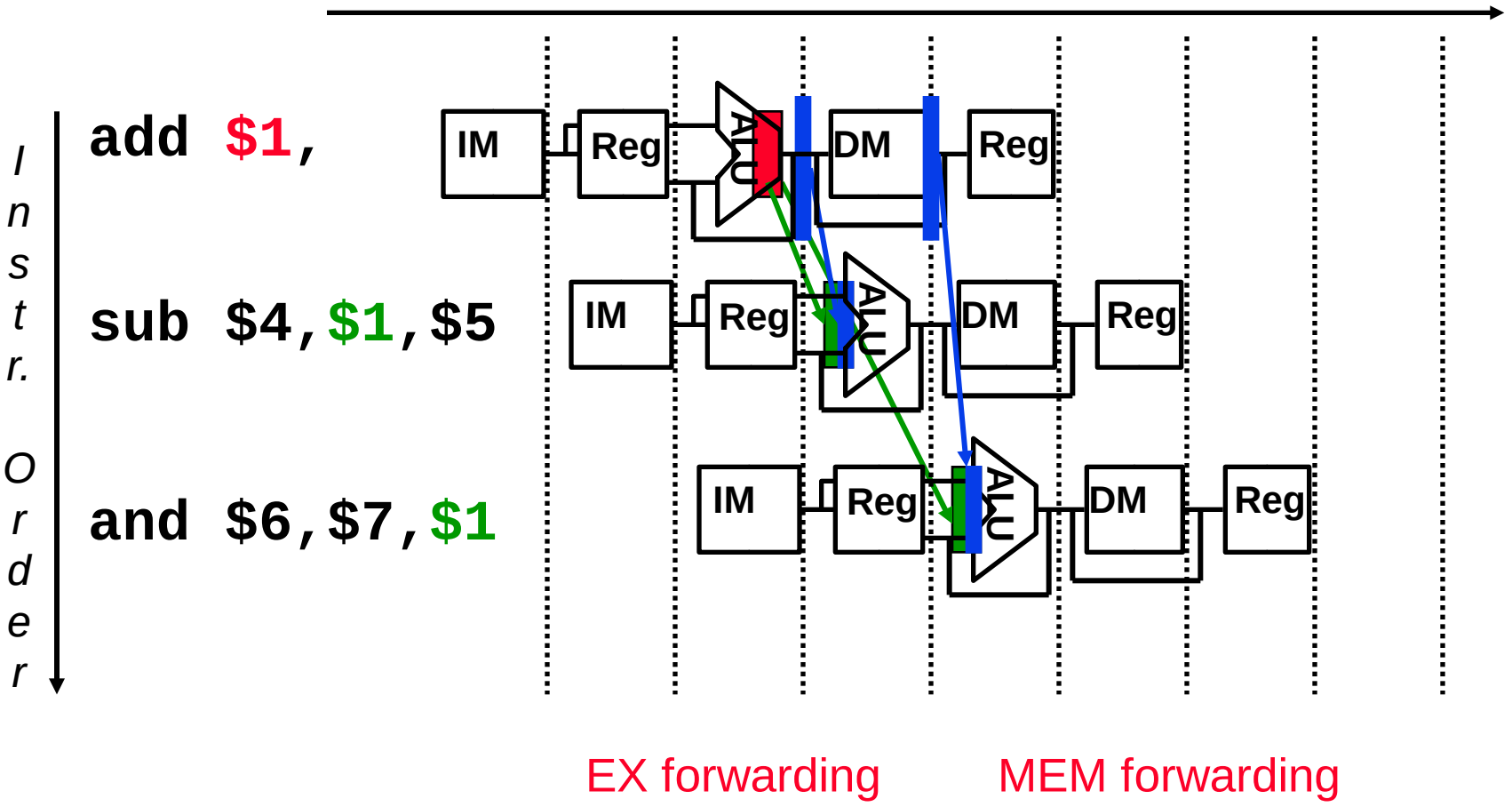
Forwards the result from the previous instr. to either input of the ALU

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
```

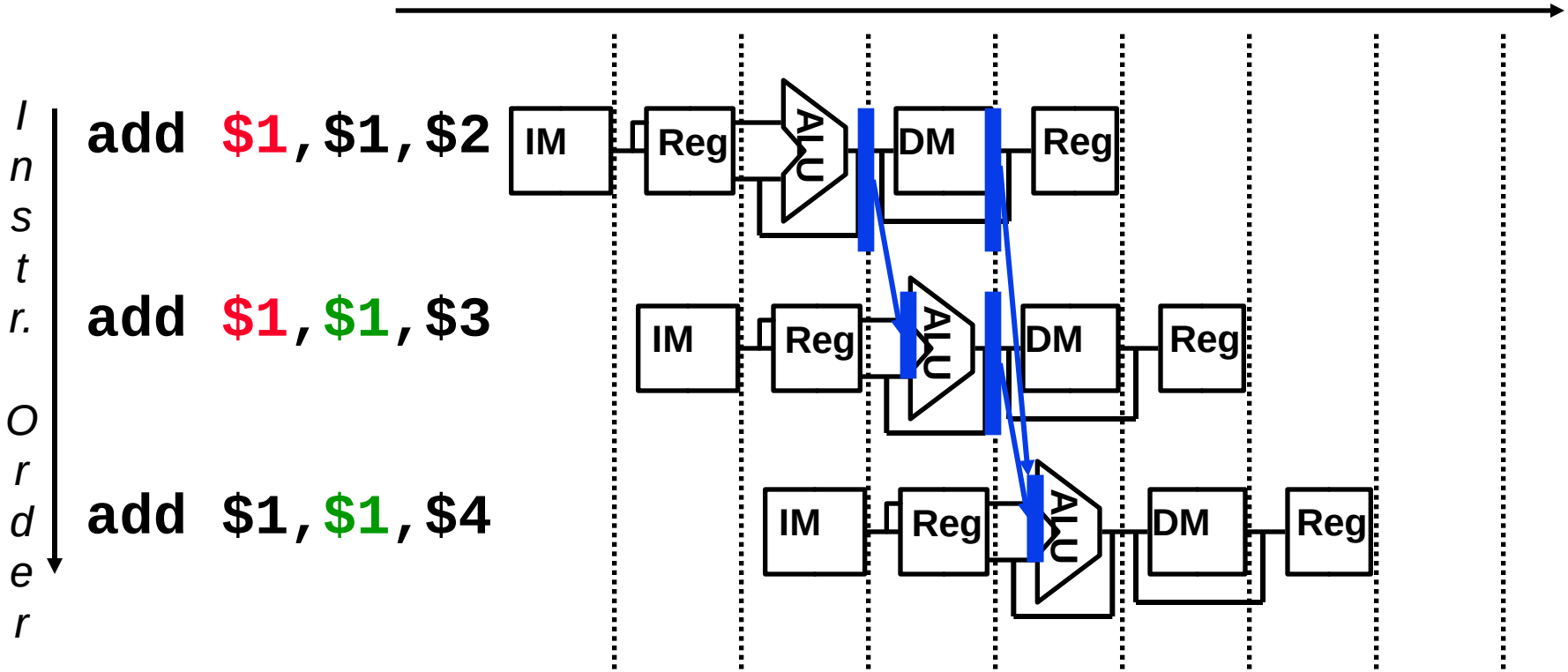
Forwards the result from the second previous instr. to either input of the ALU

Forwarding Illustration



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?



Corrected 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

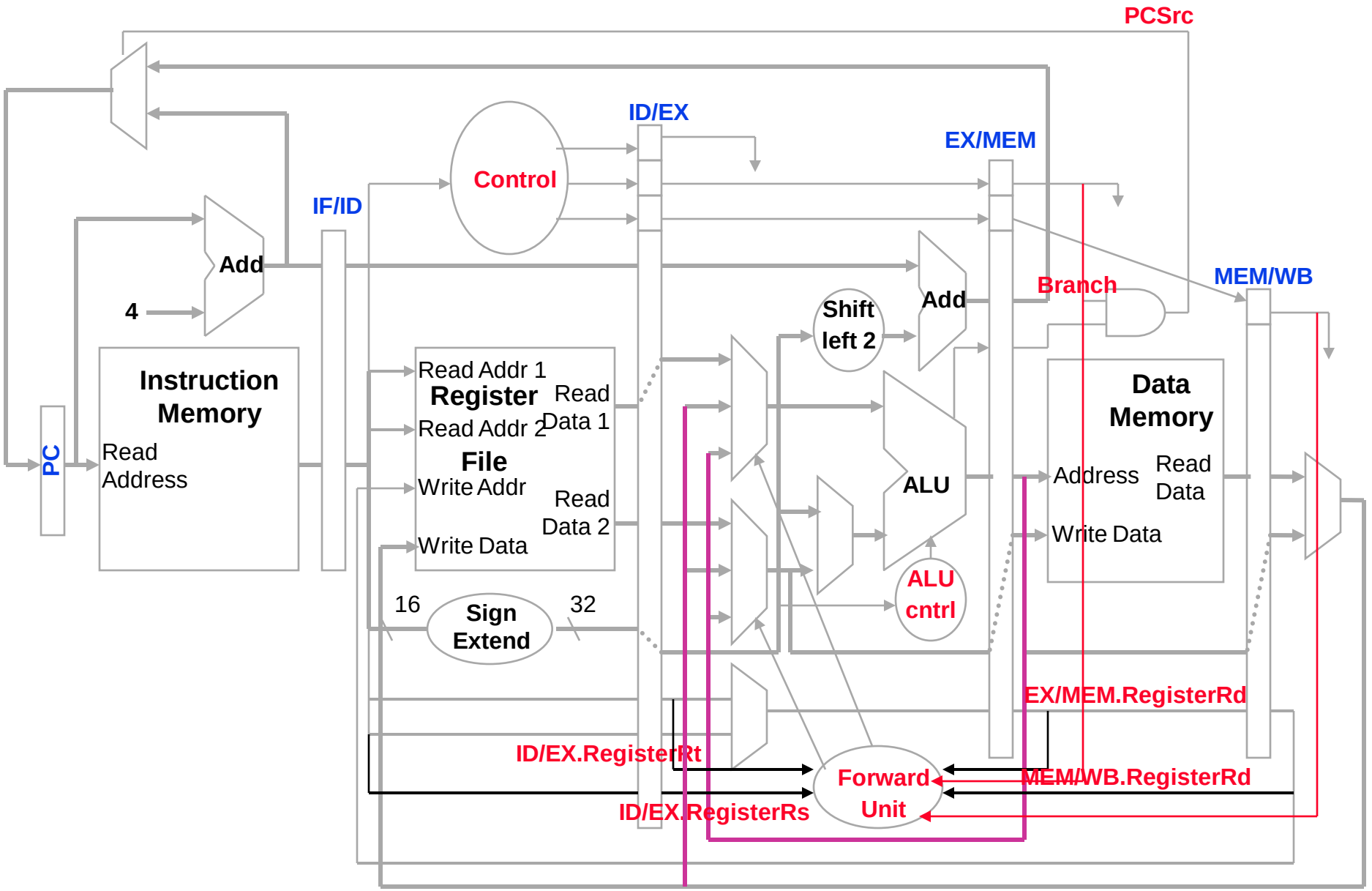
2. MEM Forward Unit:

```
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
```

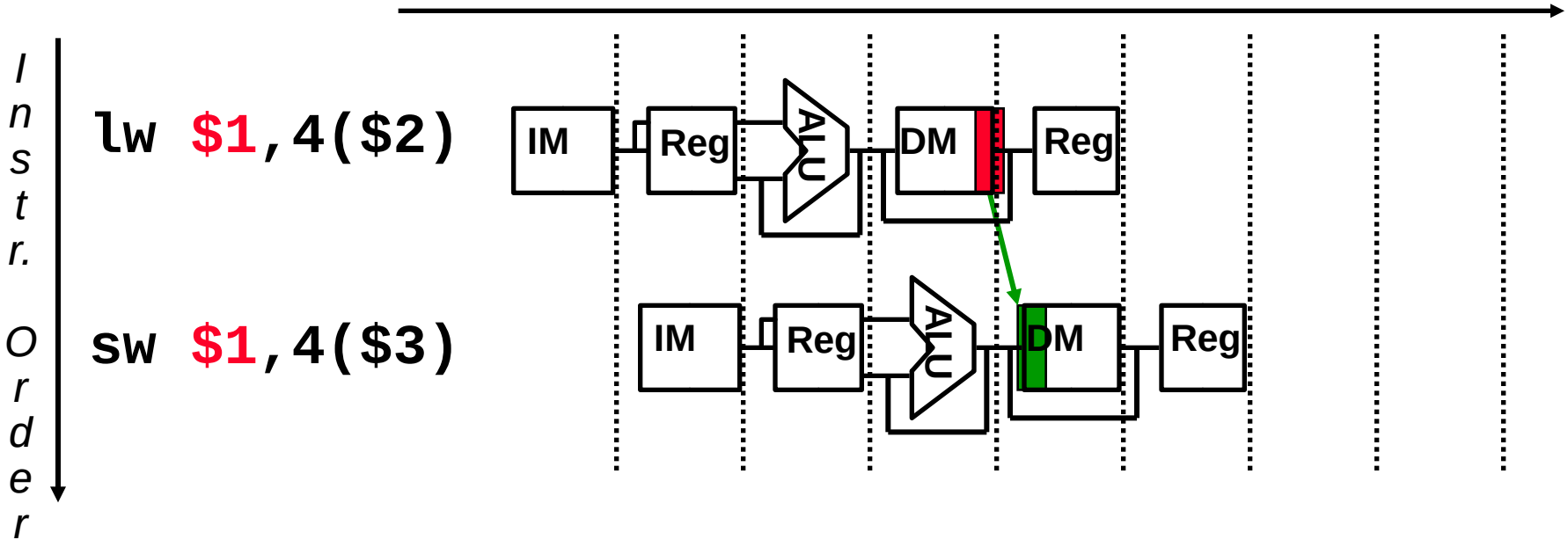
Forwards the result from the previous or second previous instr. to either input of the ALU

Datapath with Forwarding Hardware

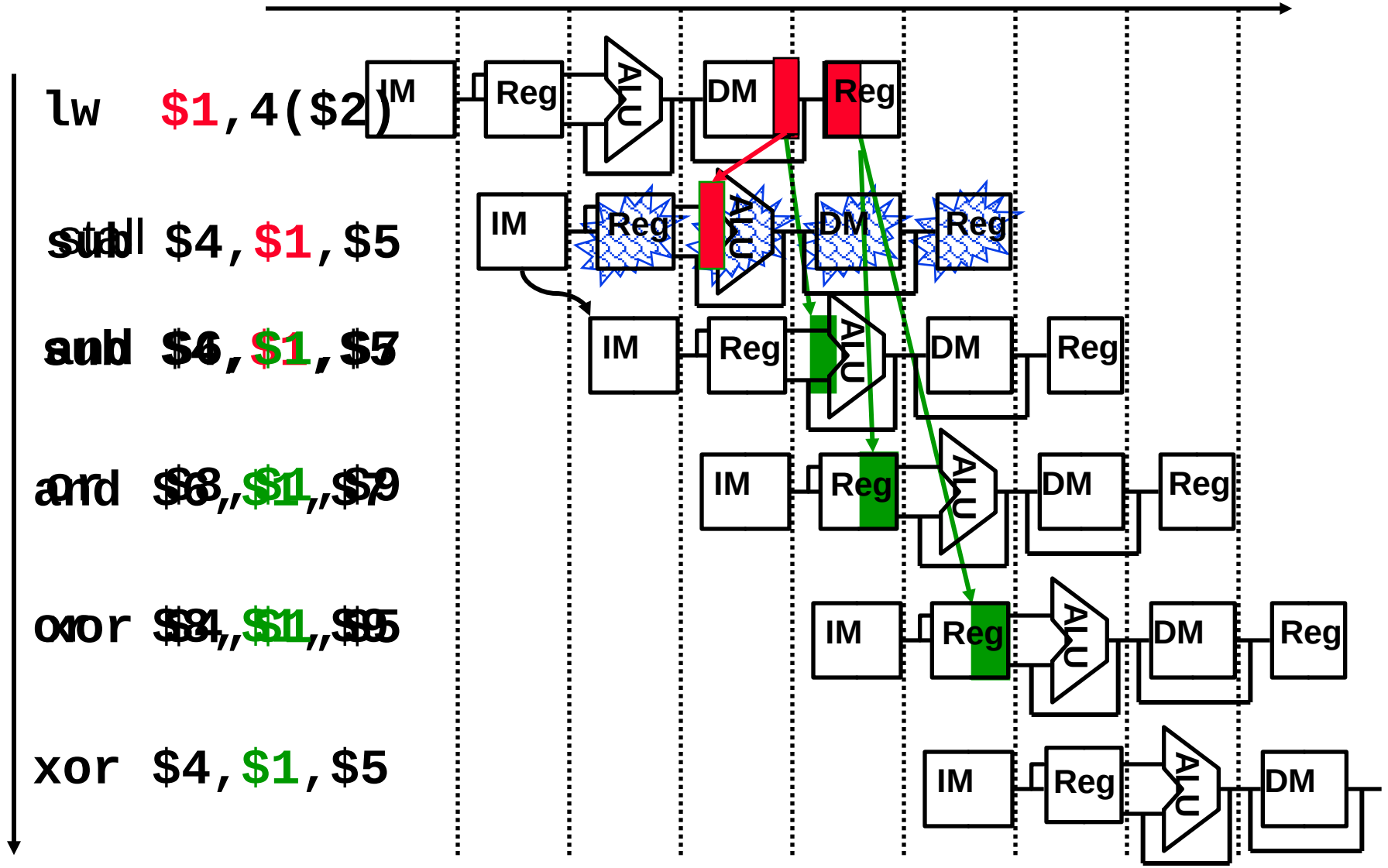


Memory-to-Memory Copies

- 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



Forwarding with Load-use Data Hazards



- Will still need one stall cycle even with forwarding

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:

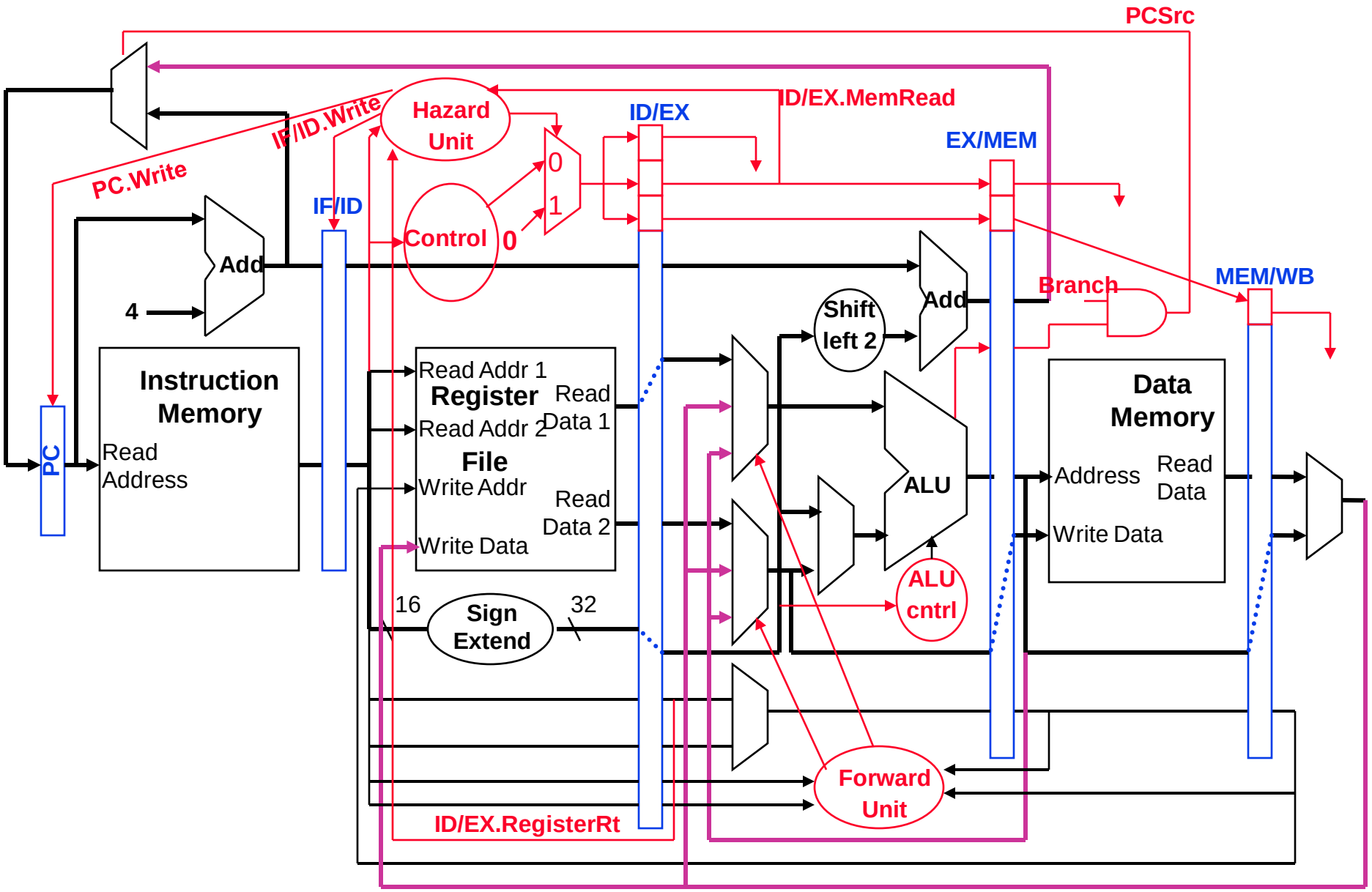
```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

- 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 `noop` in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). 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

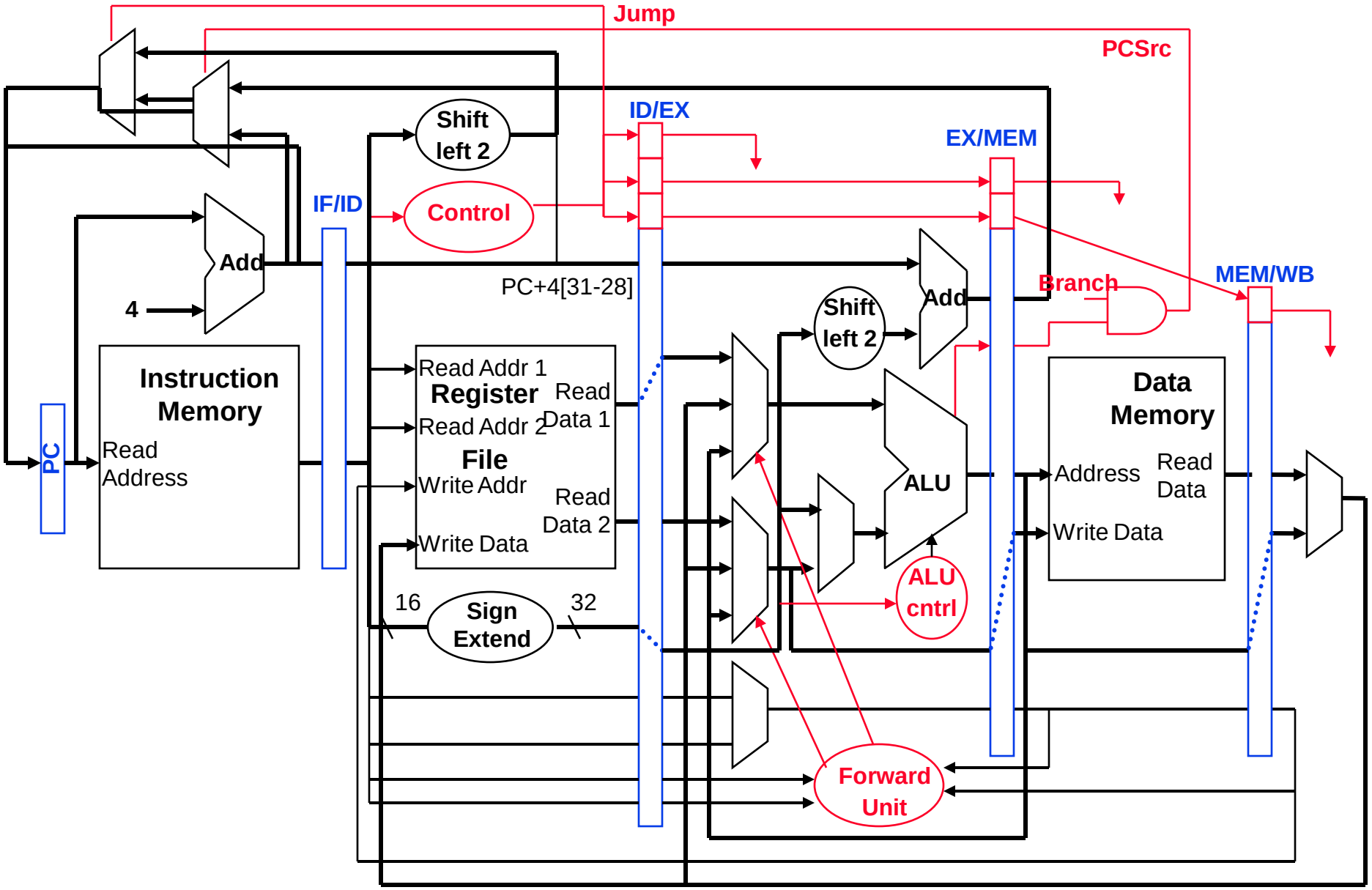
Adding the Hazard/Stall Hardware



Control Hazards

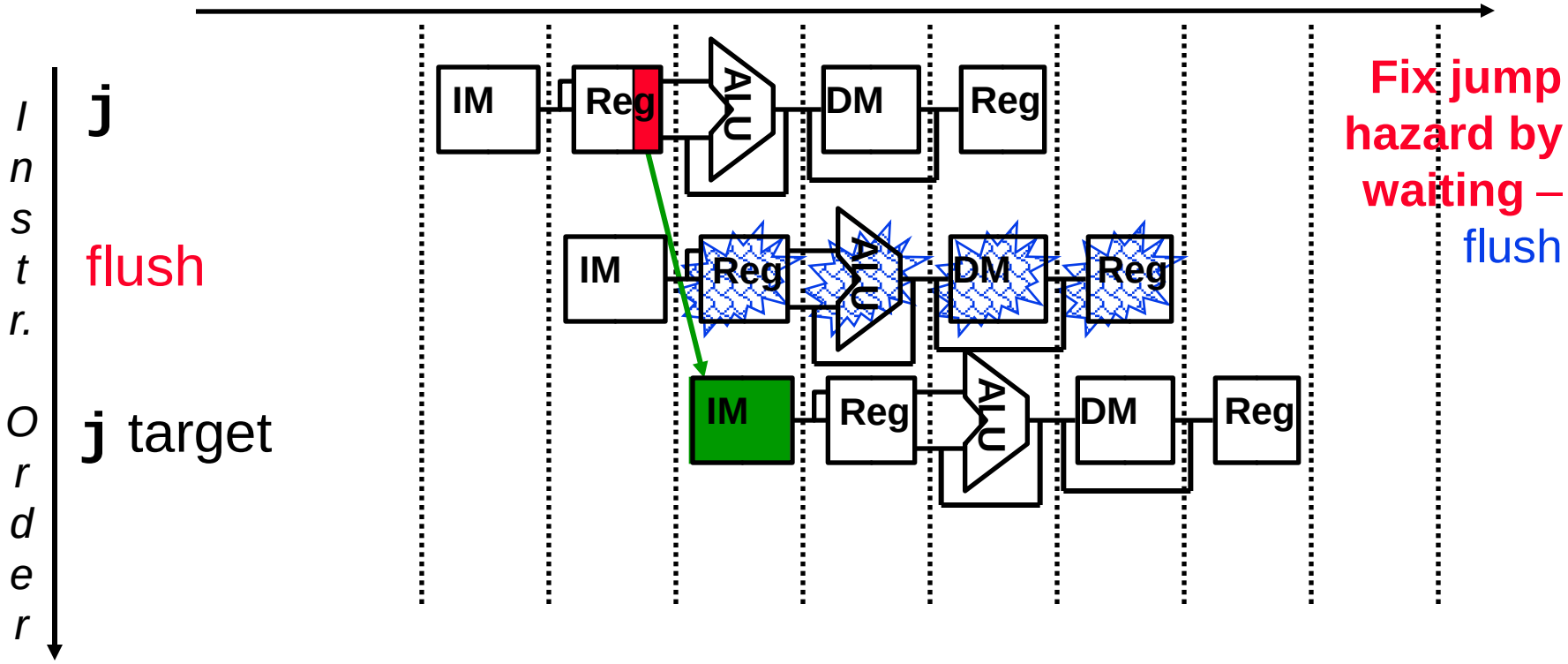
- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (`j, jal, jr`)
  - Conditional branches (`beq`, `bne`)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the 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)

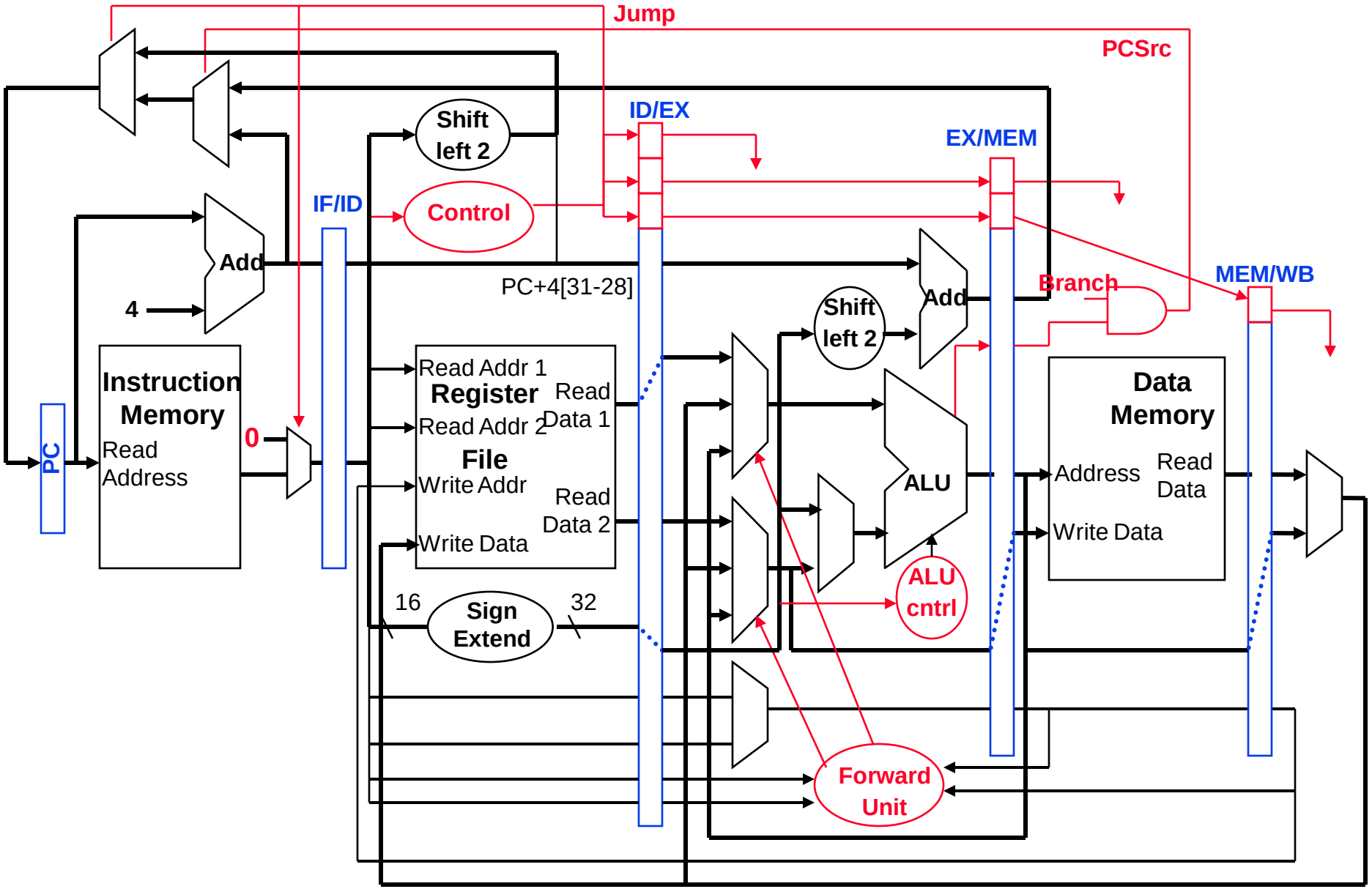


- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Two “Types” of Stalls

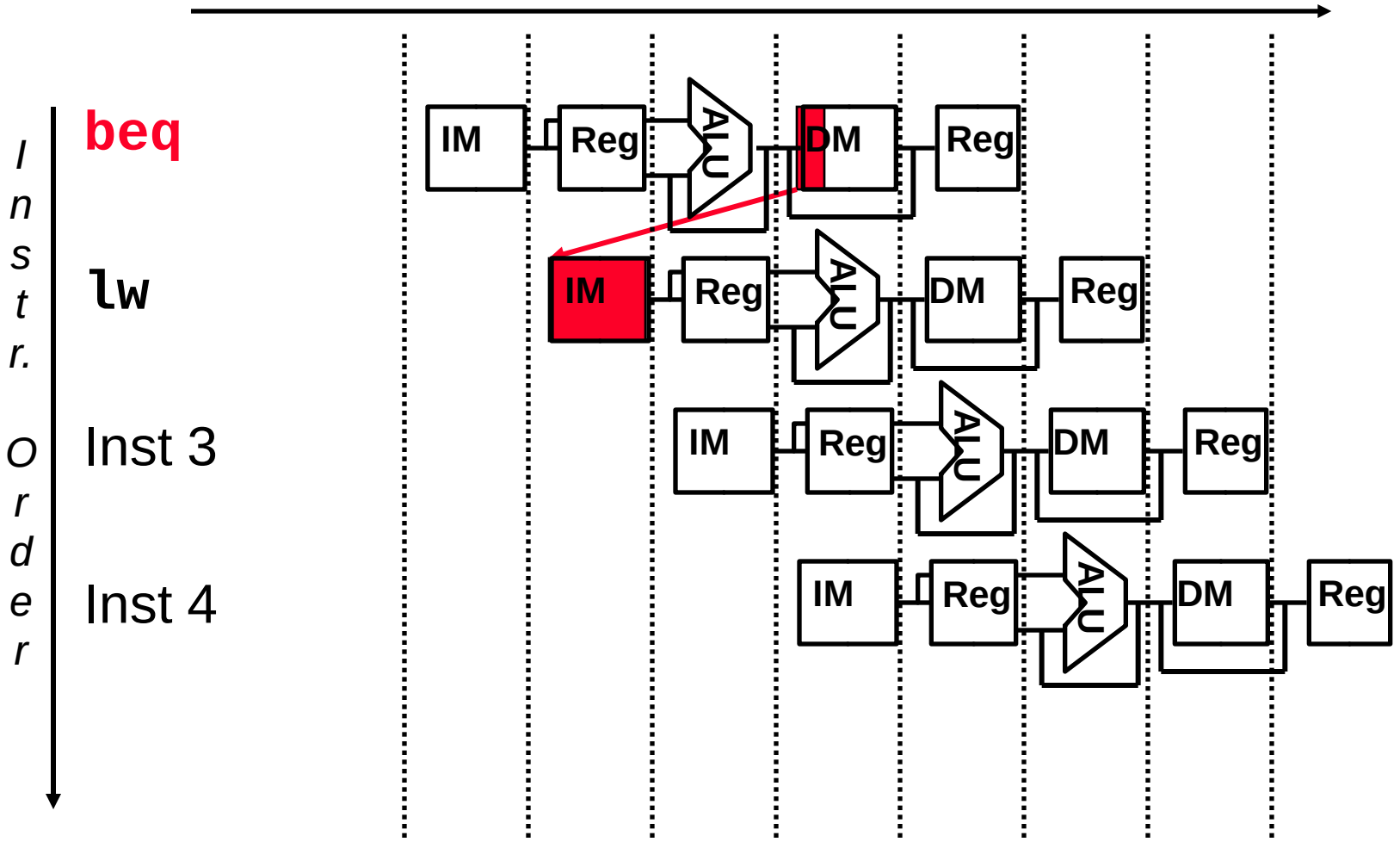
- Noop instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert `noop` 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 `noop` instruction (as done for instructions located sequentially after `j` instructions)
  - Zero the control bits for the instruction to be flushed

Supporting ID Stage Jumps

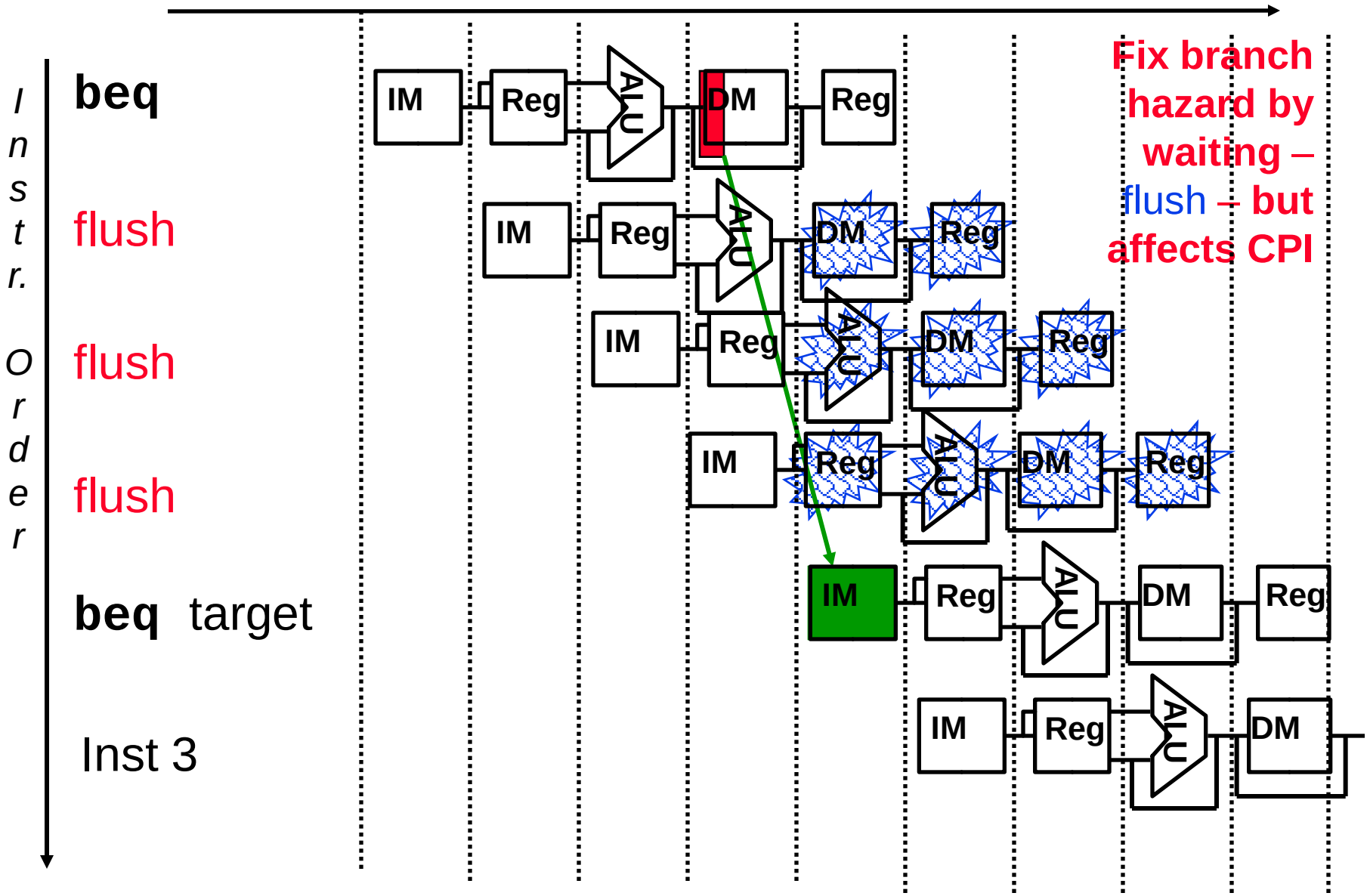


Review: Branch Instr's Cause Control Hazards

- Dependencies backward in time cause hazards

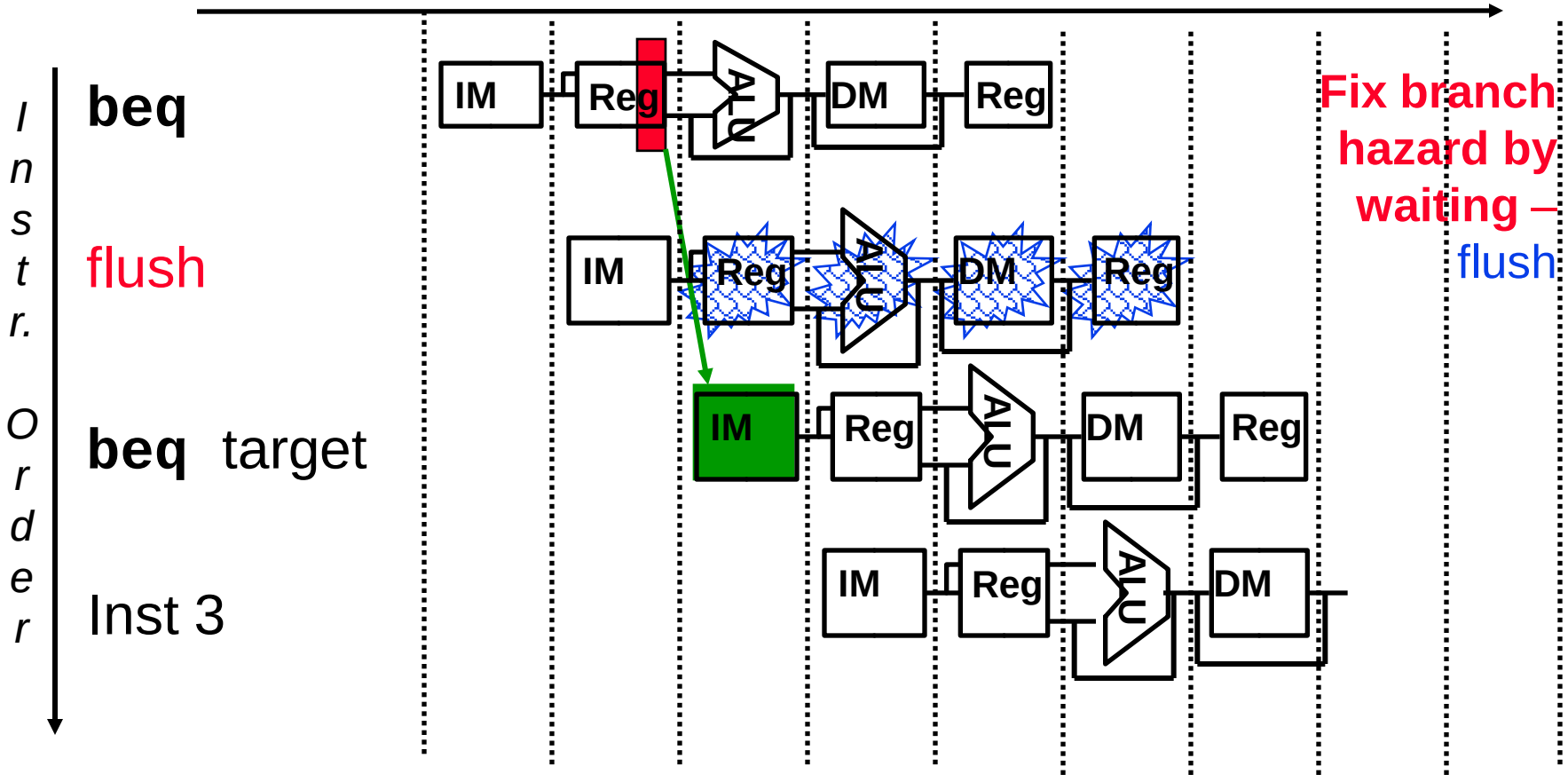


One Way to "Fix" a Branch Control Hazard



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

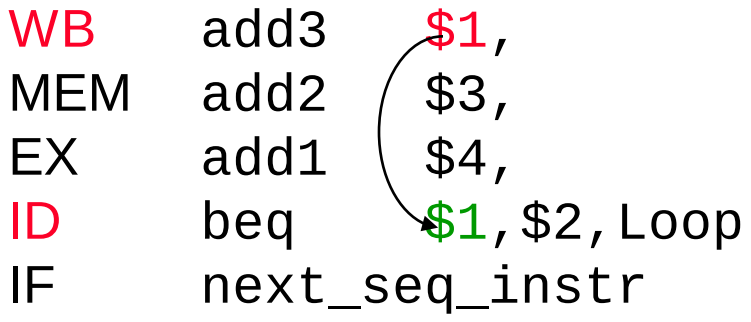


Reducing the Delay of Branches

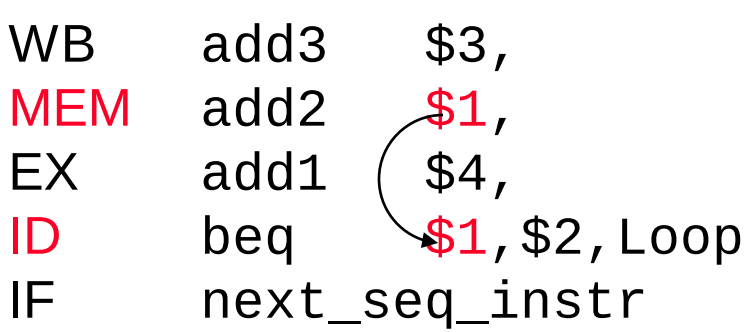
- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall (flush) cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path
- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall (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

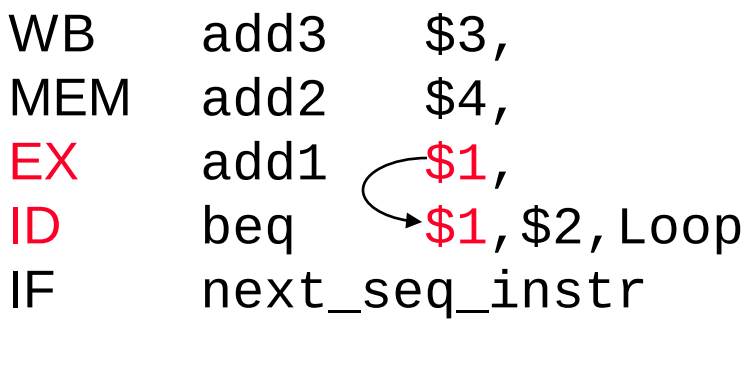



```
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
       ForwardC = 1
if (IDcontrol.Branch
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
       ForwardD = 1
```

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

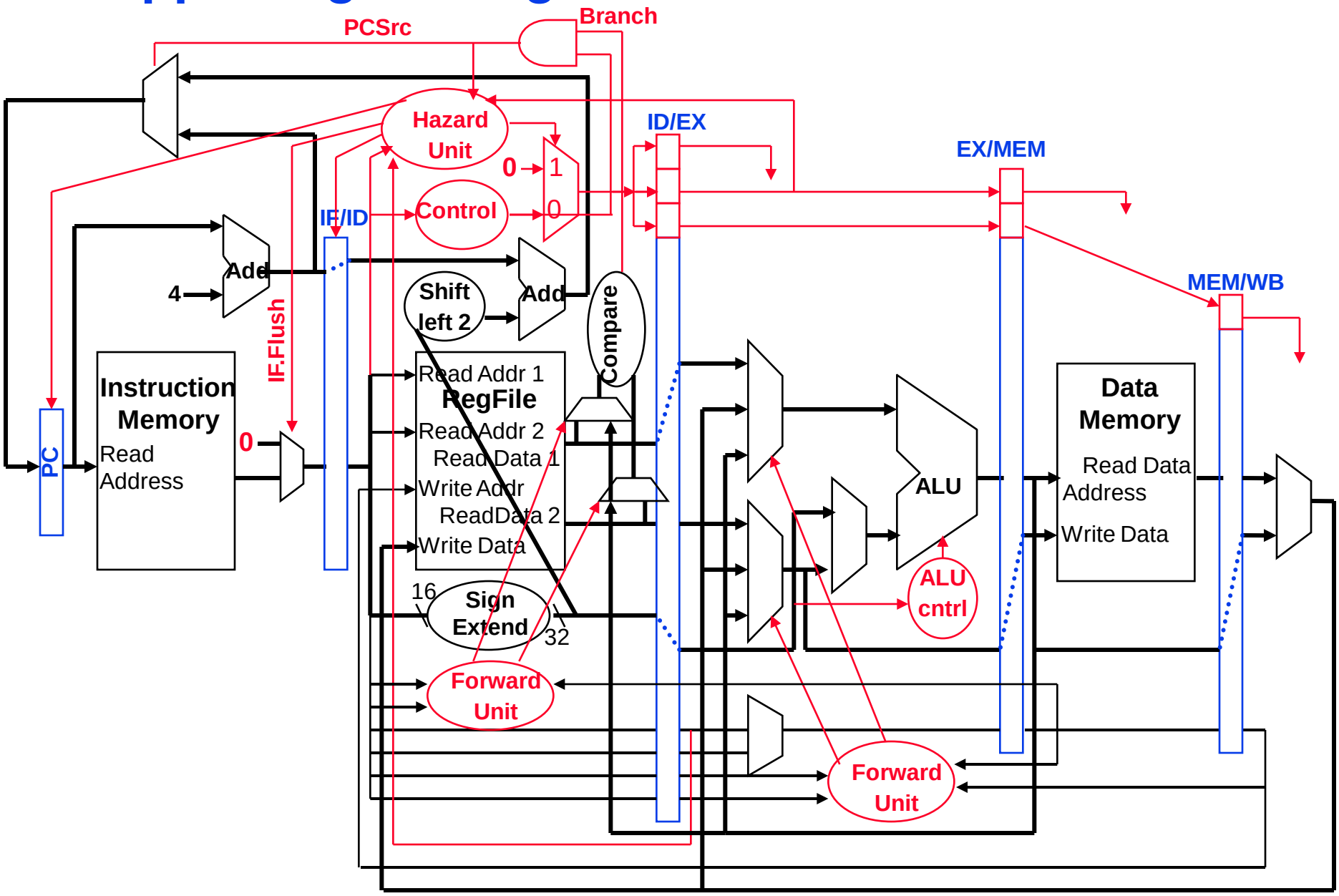
ID Branch Forwarding Issues, con't

- 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`)

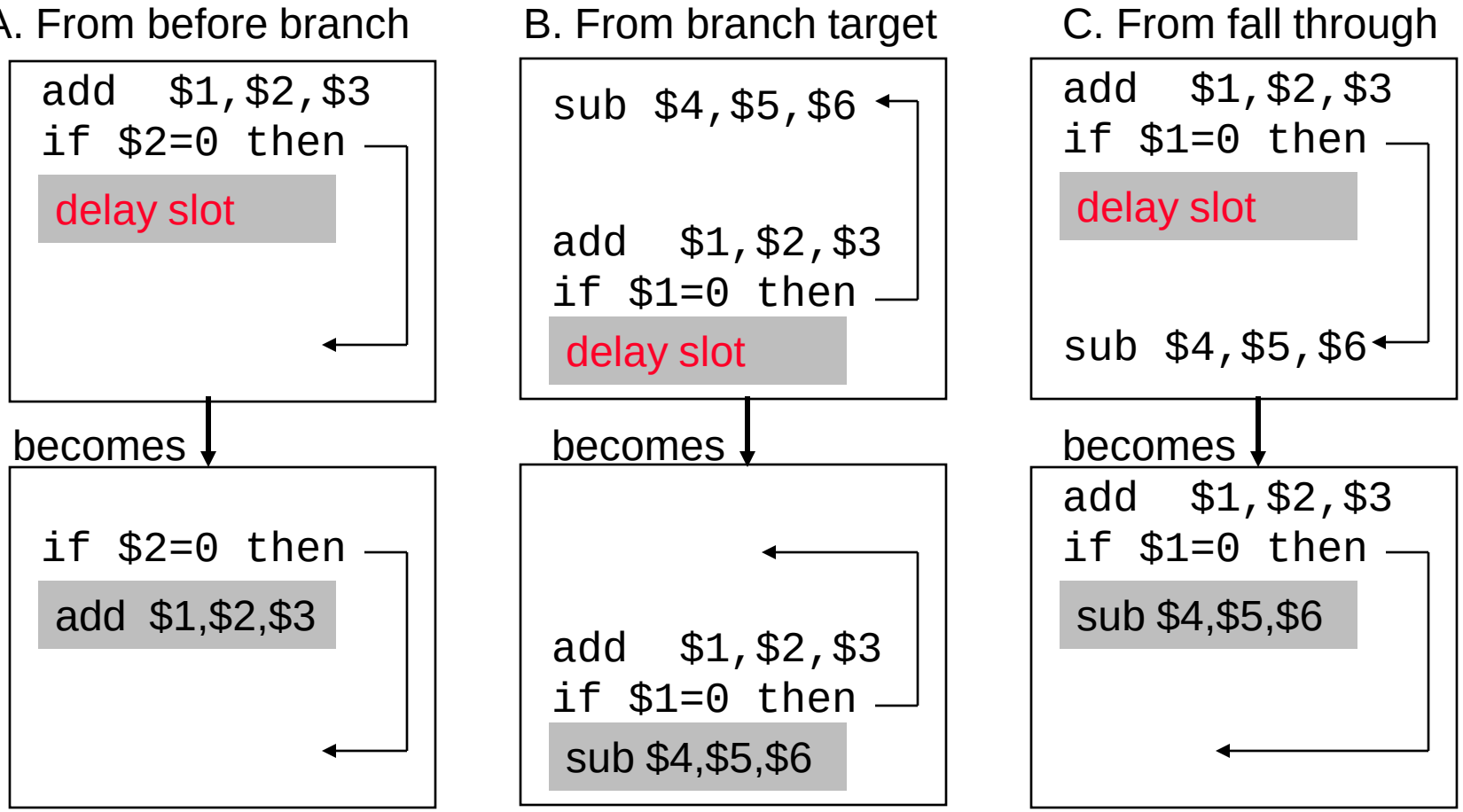
Supporting ID Stage Branches



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



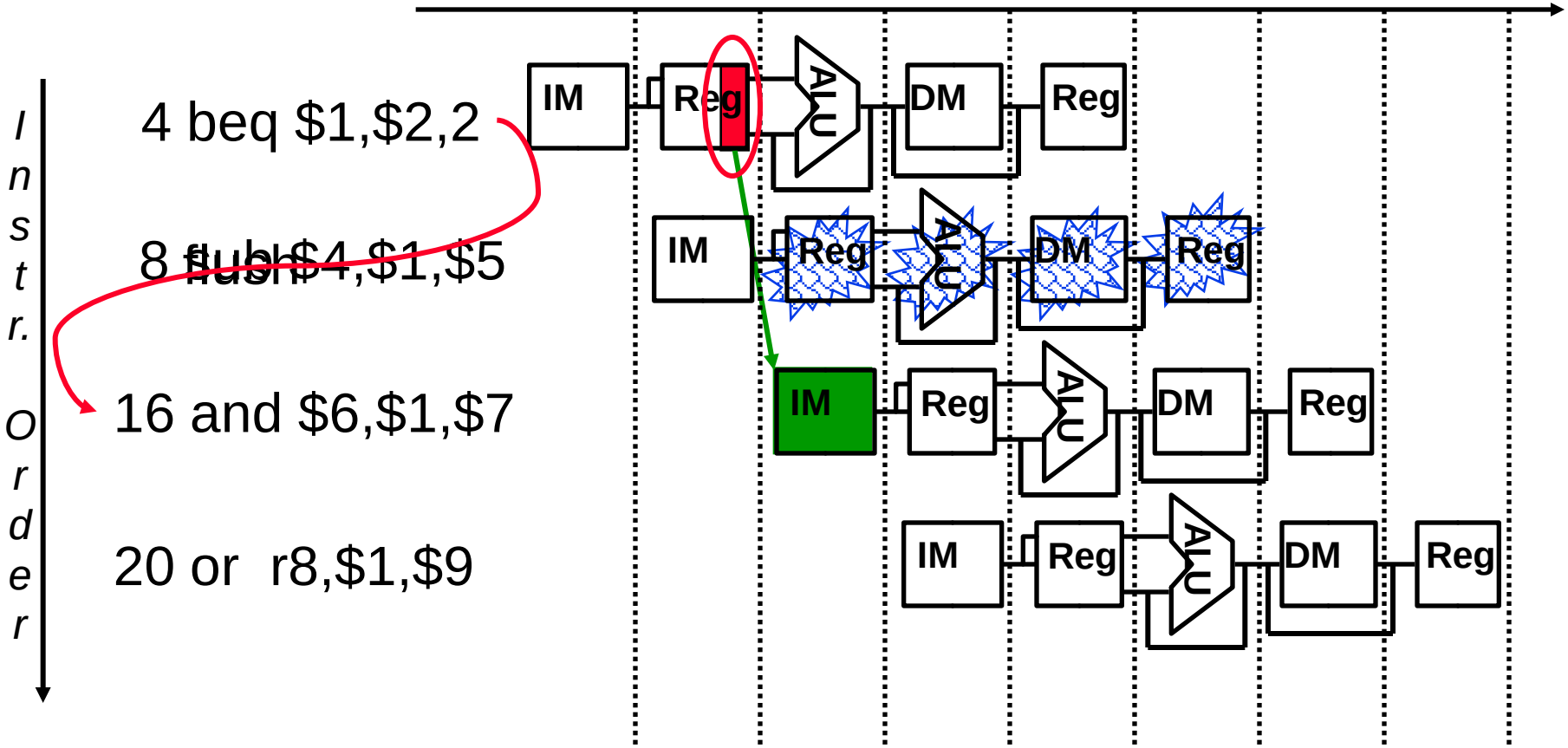
- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

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

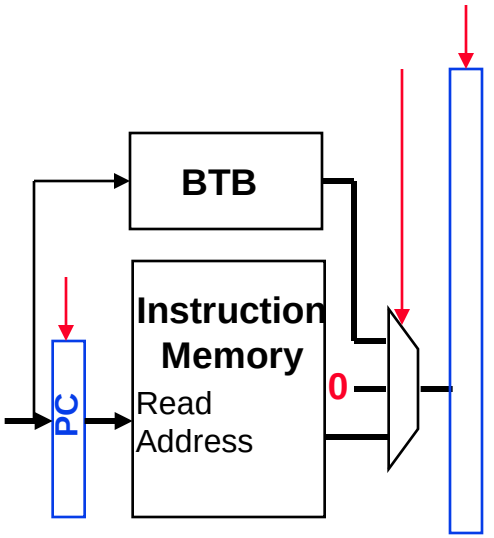
3. Dynamic branch prediction – predict branches at run-time using *run-time* information

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

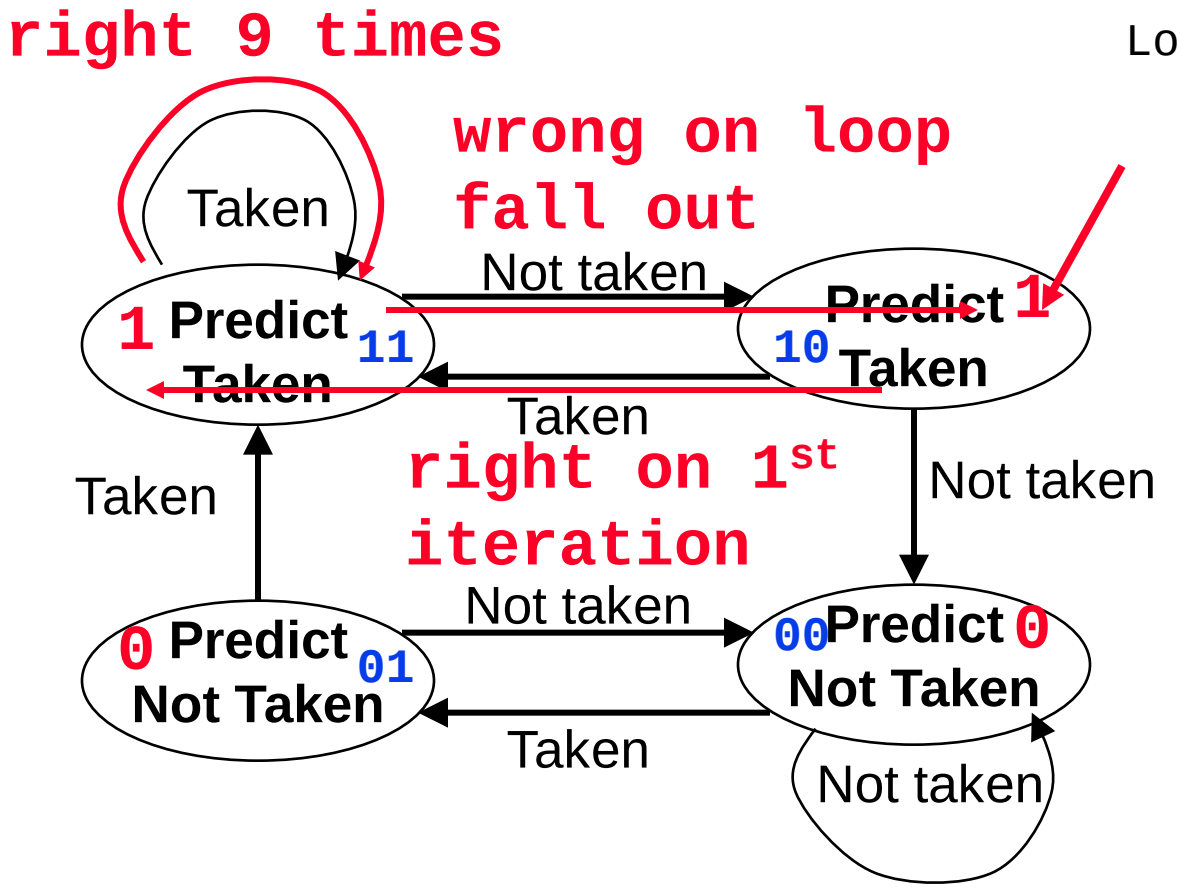
- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; 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

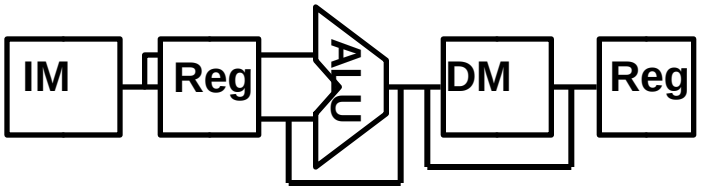
- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline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

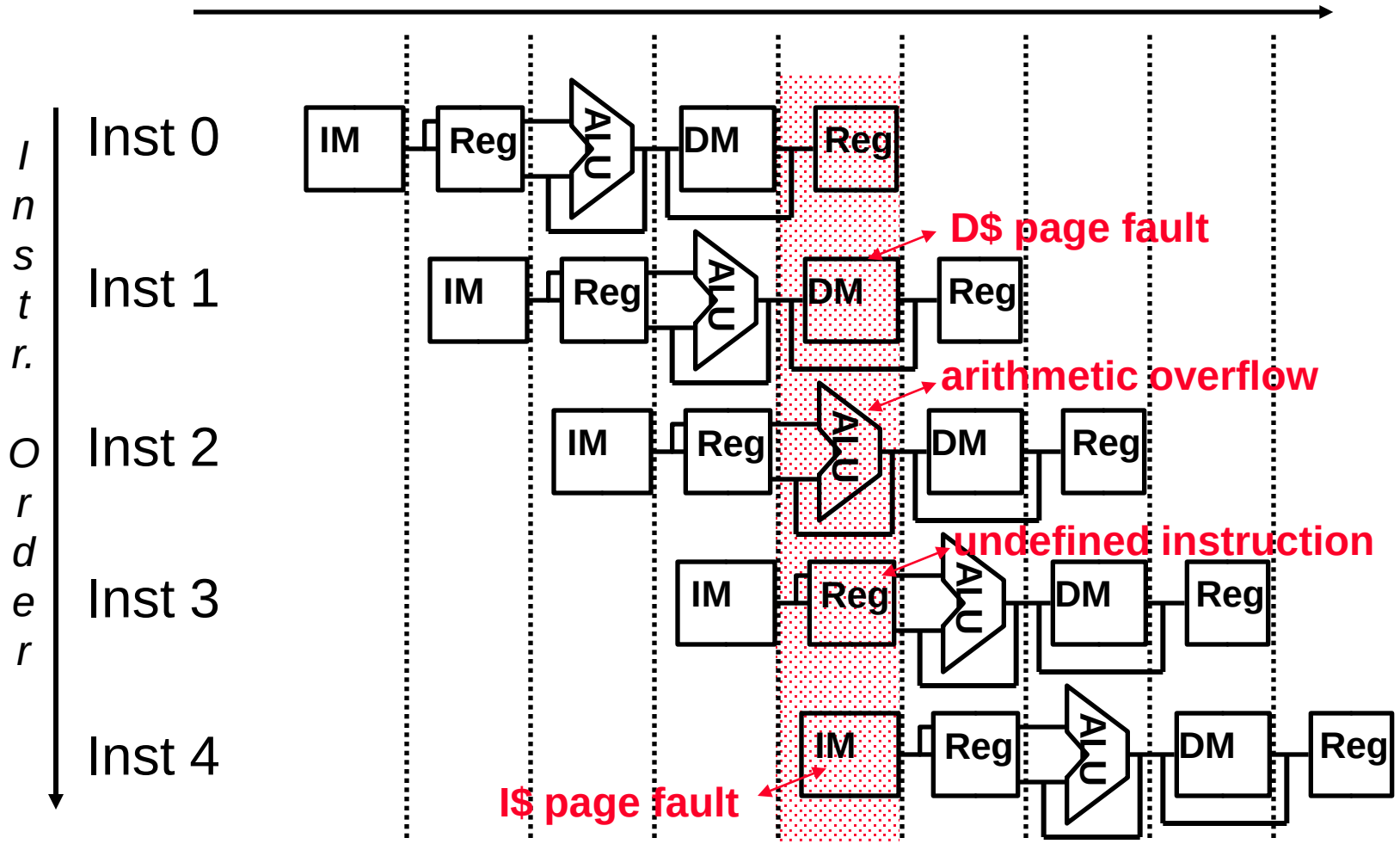
Where in the Pipeline Exceptions Occur



	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

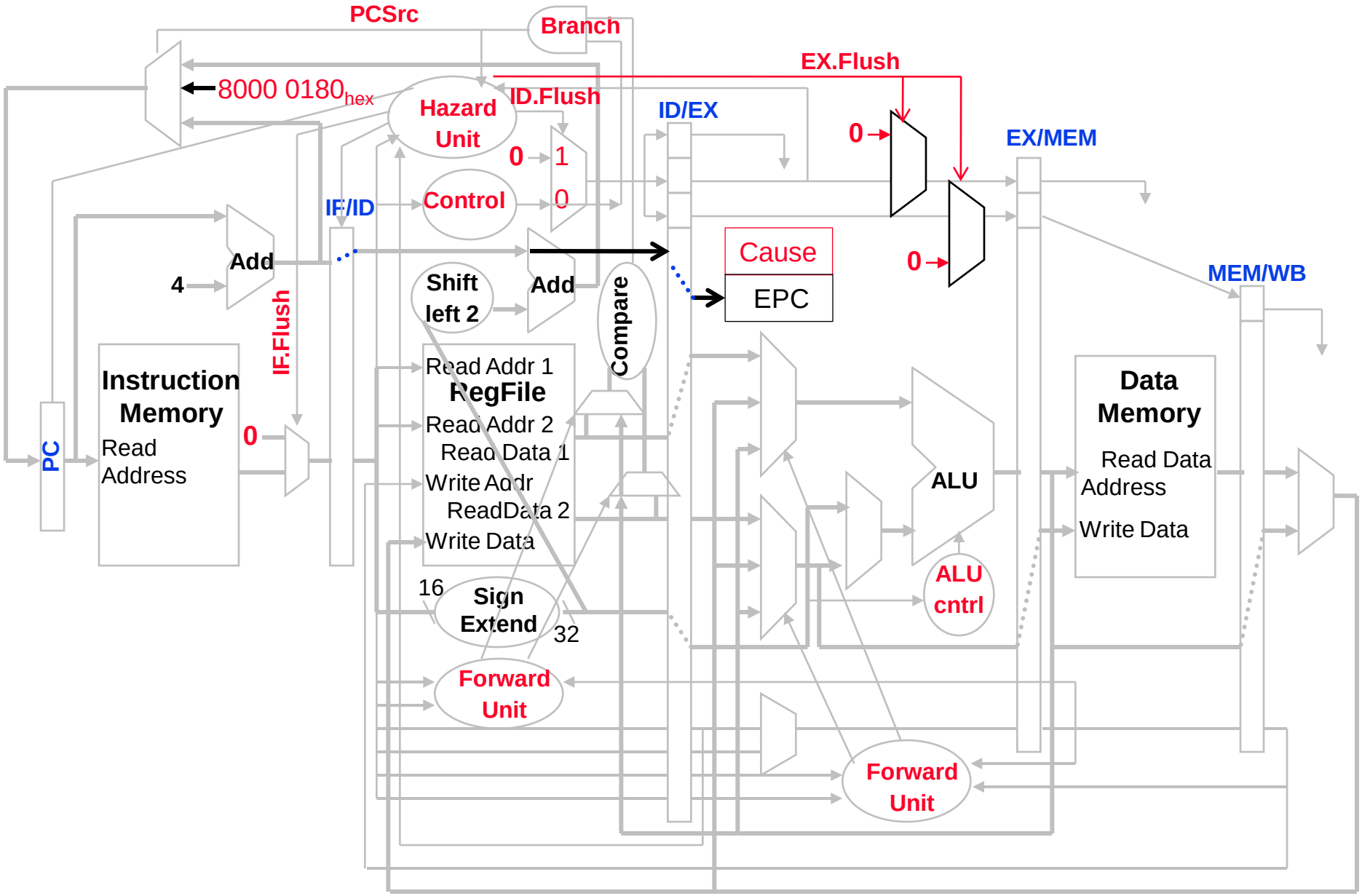


- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

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



Summary

- All modern day processors use pipelining for performance (a CPI of 1 and fast a 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

Review: Pipeline Hazards

- Structural hazards
  - Design pipeline to eliminate structural hazards
- Data hazards – read before write
  - Use data forwarding inside the pipeline
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream
- Control hazards – `beq`, `bne`,`j`,`jr`,`jal`
  - Stall – hurts performance
  - Move decision point as early in the pipeline as possible – reduces number of stalls at the cost of additional hardware
  - Delay decision (requires compiler support) – not feasible for deeper pipes requiring more than one delay slot to be filled
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (BHT) is correct and if the branched-to instruction is cached (BTB)

Extracting Yet *More* Performance

- Increase the depth of the pipeline to increase the clock rate – superpipelining
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time)
- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
  - The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

Types of Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- Data-level parallelism (DLP)

```
DO  I = 1  TO  100
   A[I] = A[I] + 1
CONTINUE
```

- Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time
- To achieve high performance, need *both* ILP and machine parallelism

Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128-bit “bundles” containing three instructions, each 41-bits plus a 5-bit template field (which specifies which FU each instruction needs)
    - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
    - Extensive support for speculation and predication
- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
  - E.g., IBM Power series, Pentium 4, MIPS R10K, AMD Barcelona

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle – issue slots
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Speculation

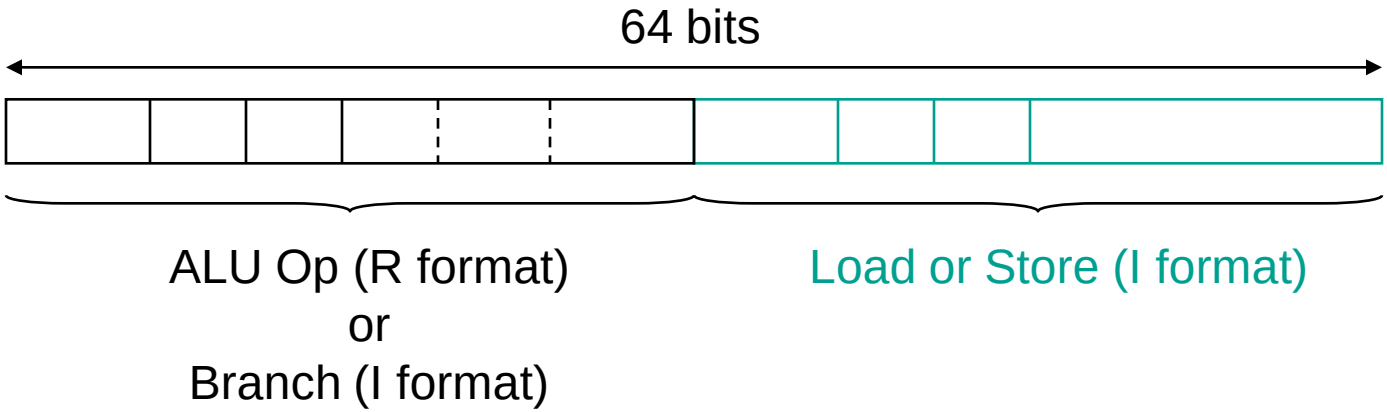
- Speculation is used to allow execution of future instr's that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously
  - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
  - The mix of instructions in the packet (bundle) is usually restricted – a single “instruction” with several predefined fields
  - The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards
- VLIW’s have
  - Multiple functional units
  - Multi-ported register files
  - Wide program bus

An Example: A VLIW MIPS

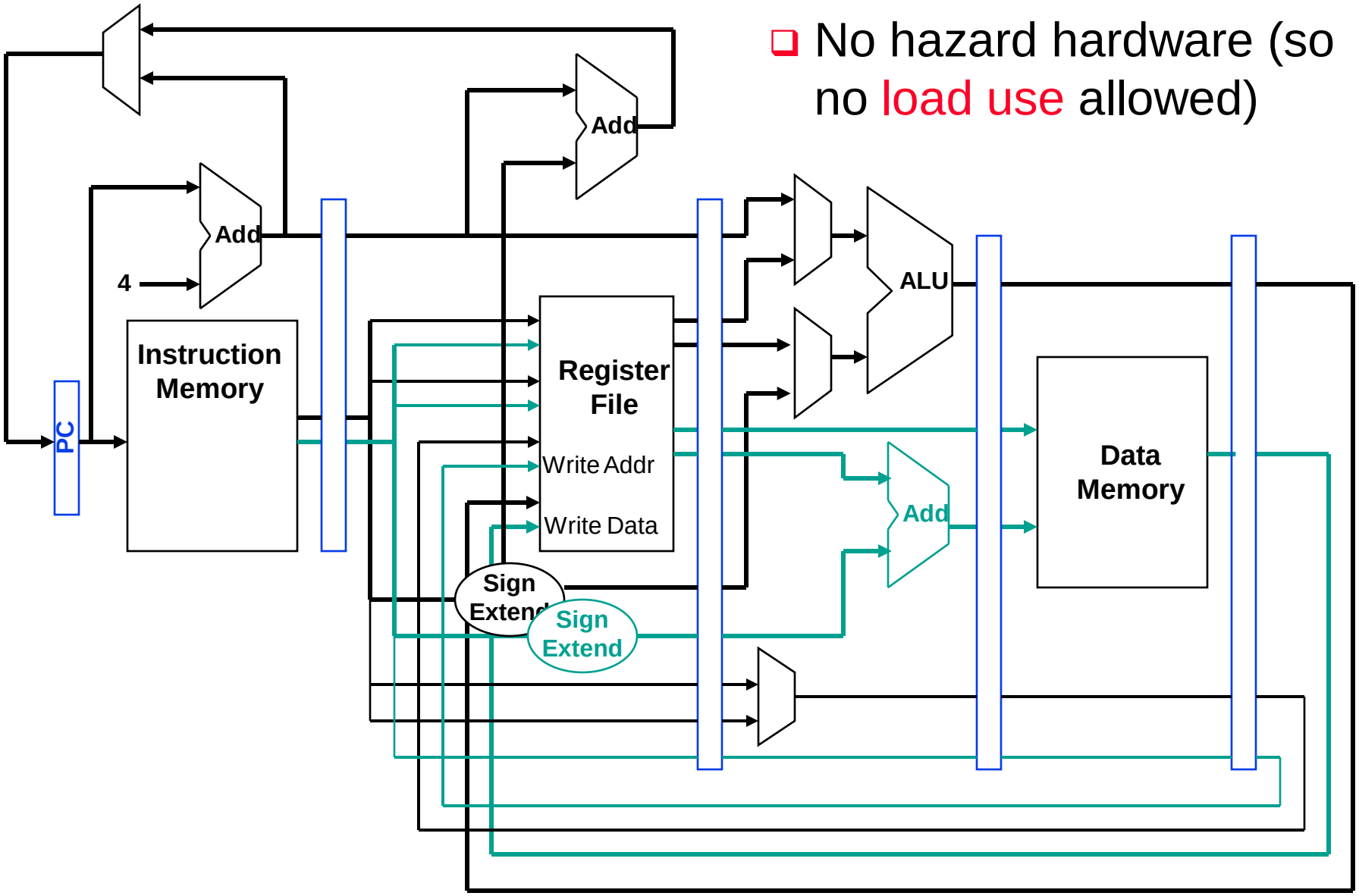
- Consider a 2-issue MIPS with a 2 instr bundle



- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports and 2 write ports and a separate memory address adder

A MIPS VLIW (2-issue) Datapath

- No hazard hardware (so no load use allowed)



Code Scheduling Example

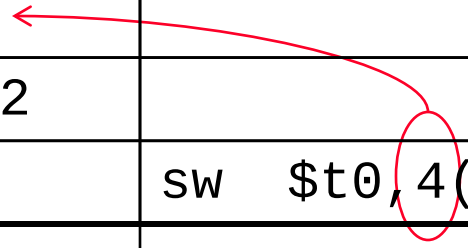
- Consider the following loop code

```
lp:   lw   $t0,0($s1)   # $t0=array element
      addu $t0,$t0,$s2  # add scalar in $s2
      sw   $t0,0($s1)   # store result
      addi $s1,$s1,-4   # decrement pointer
      bne  $s1,$0,lp    # branch if $s1 != 0
```

- Must "schedule" the instructions to avoid pipeline stalls
  - Instructions in one bundle *must* be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:		lw $t0,0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$0,lp	sw $t0,4($s1)	4
			5



- Four clock cycles to execute 5 instructions for a
  - CPI of 0.8 (versus the best case of 0.5)
  - IPC of 1.25 (versus the best case of 2.0)
  - noops don't count towards performance !!

Loop Unrolling

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP

- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards

- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true data dependencies

Unrolled Code Example

```
lp:   lw   $t0,0($s1)    # $t0=array element
      lw   $t1,-4($s1)   # $t1=array element
      lw   $t2,-8($s1)   # $t2=array element
      lw   $t3,-12($s1)  # $t3=array element
      addu $t0,$t0,$s2   # add scalar in $s2
      addu $t1,$t1,$s2   # add scalar in $s2
      addu $t2,$t2,$s2   # add scalar in $s2
      addu $t3,$t3,$s2   # add scalar in $s2
      sw   $t0,0($s1)    # store result
      sw   $t1,-4($s1)   # store result
      sw   $t2,-8($s1)   # store result
      sw   $t3,-12($s1)  # store result
      addi $s1,$s1,-16   # decrement pointer
      bne  $s1,$0,lp     # branch if $s1 != 0
```

The Scheduled Code (Unrolled)

	ALU or branch	Data transfer	CC
lp:	addi $s1,$s1,-16	lw $t0,0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2,8($s1)	3
	addu $t1,$t1,$s2	lw $t3,4($s1)	4
	addu $t2,$t2,$s2	sw $t0,16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2,8($s1)	7
	bne $s1,$0,lp	sw $t3,4($s1)	8

- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0)

Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - Done by code re-ordering (trace scheduling)
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
- The compiler predicts memory bank references to help minimize memory bank conflicts

VLIW Advantages & Disadvantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr's per VLIW bundle and add more FUs
- Disadvantages
  - Programmer/compiler complexity and longer compilation times
    - Deep pipelines and long latencies can be confusing (making peak performance elusive)
  - Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
  - Object (binary) code incompatibility
  - Needs lots of program memory bandwidth
  - Code bloat
    - Noops are a waste of program memory space
    - Loop unrolling to expose more ILP uses more program memory space

Dynamic Multiple Issue Machines (SS)

- Dynamic multiple-issue processors (aka SuperScalar) use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously
- Instruction-fetch and issue – fetch instructions, decode them, and *issue* them to a FU to await execution
  - Defines the Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- Instruction-execution – as soon as the source operands and the FU are ready, the result can be calculated
  - Defines the processor lookahead capability – complete execution of issued instructions beyond the current instruction
- Instruction-commit – when it is safe to, write back results to the RegFile or D$ (i.e., change the machine state)

In-Order vs Out-of-Order

- Instruction fetch and decode units are required to issue instructions in-order so that dependencies can be tracked
- The commit unit is required to write results to registers and memory in program fetch order so that
  - if exceptions occur the only registers updated will be those written by instructions before the one causing the exception
  - if branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation)
- Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order, the FUs are free to initiate execution whenever the data they need is available – out-of-(program) order execution
  - Allowing out-of-order execution increases the amount of ILP

Out-of-Order Execution

- With out-of-order execution, a later instruction may execute before a previous instruction so the hardware needs to resolve both read before write *and* write before write data hazards

```
lw   $t0,0($s1)
addu $t0,$t1,$s2
. . .
sub  $t2, $t0, $s2
```

  - If the `lw` write to `$t0` occurs after the `addu` write, then the `sub` gets an incorrect value for `$t0`
  - The `addu` has an output dependency on the `lw` – write before write
    - The issuing of the `addu` might have to be stalled if its result could later be overwritten by an previous instruction that takes longer to complete

Antidependencies

- Also have to deal with antidependencies – when a later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that executes later)

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

Antidependency
True data dependency
Output dependency

- The constraint is similar to that of true data dependencies, except *reversed*
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

Dependencies Review

- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  } storage conflicts

  manifests itself through the use of registers (or other storage locations)

- True dependencies represent the flow of data and information through a program

- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations leading to storage conflicts

Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors

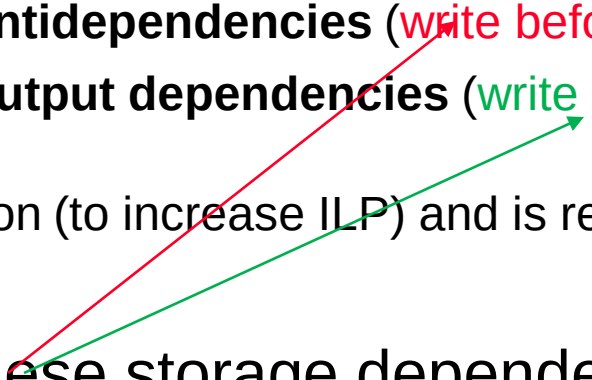
- Register renaming – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * R5          R3b := R3a * R5a
R4 := R3 + 1     ⇨     R4a := R3b + 1
R3 := R5 + 1           R3c := R5a + 1
```

  - The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

Summary: Extracting More Performance

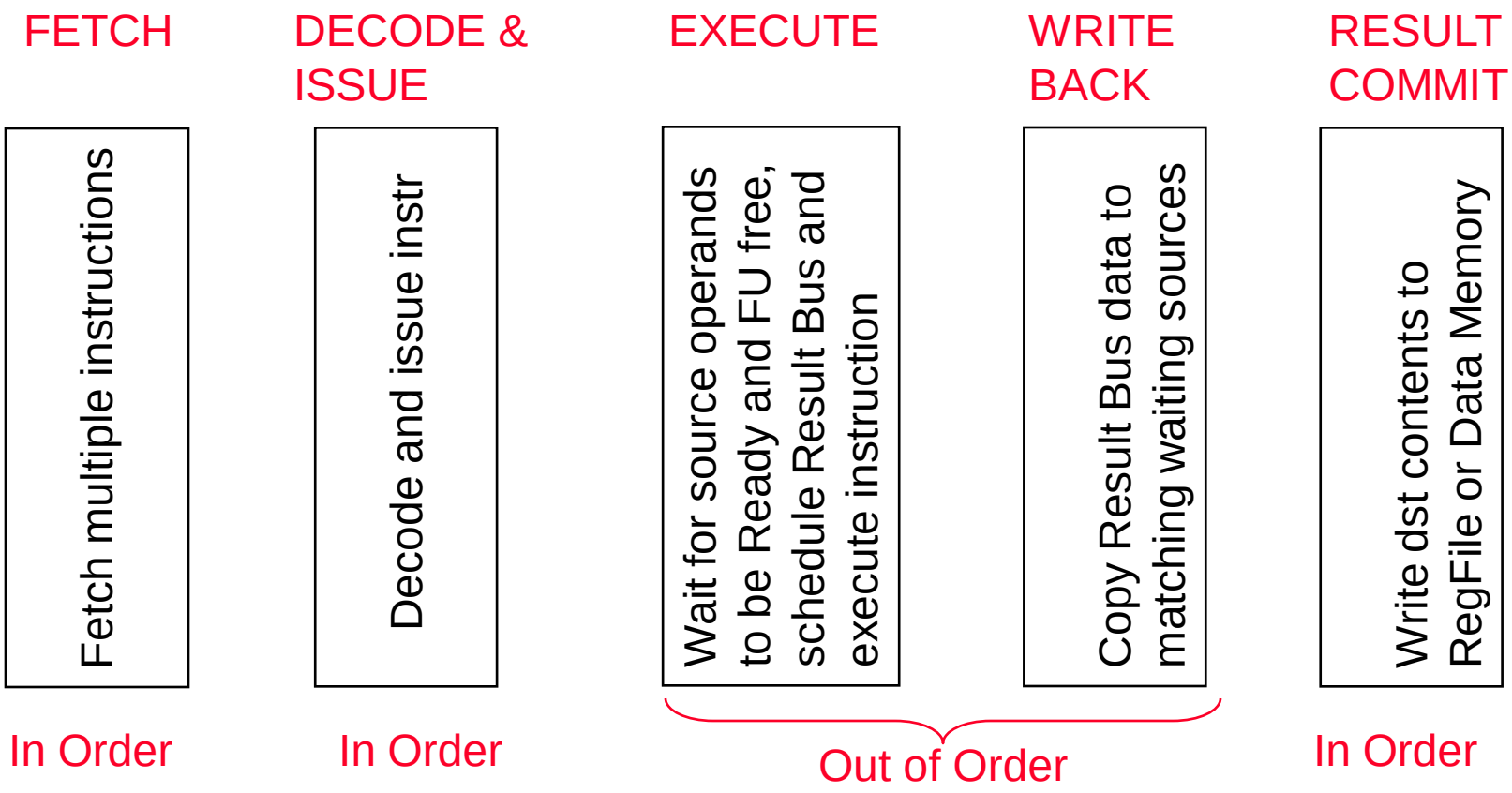
- To achieve high performance, need both machine parallelism and instruction level parallelism (ILP) by
  - Superpipelining
  - Static multiple-issue (VLIW)
  - Dynamic multiple-issue (superscalar)
- A processor's instruction issue and execution policies impact the available ILP
  - In-order fetch, issue, and commit and out-of-order execution
    - Pipelining creates **true** dependencies (read before write)
    - Out-of-order execution creates **antidependencies** (write before read)
    - Out-of-order execution creates **output dependencies** (write before write)
    - In-order commit allows speculation (to increase ILP) and is required to implement precise interrupts
- Register renaming can solve these storage dependencies



SimpleScalar Structure

- `sim-outorder`: supports out-of-order execution (with in-order commit) with a Register Update Unit (RUU)
  - Uses a RUU for register renaming and to hold the results of pending instructions. The RUU (aka reorder buffer (ROB)) retires (i.e., commits) completed instructions in program order to the RegFile
  - Uses a LSQ for store instructions not ready to commit and load instructions waiting for access to the D$
  - Loads are satisfied by either the memory or by an earlier store value residing in the LSQ if their addresses match
    - Loads are issued to the memory system only when addresses of *all* previous loads and stores are known

SS Pipeline Stage Functions




```
ruu_fetch()
        ruu_dispatch()
ruu_issue()
lsq_refresh()
                ruu_writeback()
                                ruu_commit()
```

Simulated SimpleScalar Pipeline

- `ruu_fetch()`: fetches instr's from one I$ line, puts them in the fetch queue, probes the cache line predictor to determine the next I$ line to access in the next cycle
  - fetch:ifqsize<size>: fetch width (default is 4)
  - fetch:speed<ratio>: ratio of the front end speed to the execution core (<ratio> times as many instructions fetched as decoded per cycle)
  - fetch:mplat<cycles>: branch misprediction latency (default is 3)
- `ruu_dispatch()`: decodes instr's in the fetch queue, puts them in the dispatch (scheduler) queue, enters and links instr's into the RUU and the LSQ, splits memory access instructions into two separate instr's (one to compute the effective addr and one to access the memory), notes branch mispredictions
  - decode:width<insts>: decode width (default is 4)

SimpleScalar Pipeline, con't

- `ruu_issue()` and `lsq_refresh()`: locates and marks the instr's ready to be executed by tracking register and memory dependencies, ready loads are issued to D$ unless there are earlier stores in LSQ with unresolved addr's, forwards store values with matching addr to ready loads
  - issue:width<insts>: maximum issue width (default is 4)
  - ruu:size<insts>: RUU capacity in instr's (default is 16, min is 2)
  - lsq:size<insts>: LSQ capacity in instr's (default is 8, min is 2)

  and handles instr's execution – collects all the ready instr's from the scheduler queue (up to the issue width), check on FU availability, checks on access port availability, schedules writeback events based on FU latency (hardcoded in `fu_config[]`)
  - res:ialu | imult | memport | fpalu | fpmult<num>: number of FU's (default is 4 | 1 | 2 | 4 | 1)

SimpleScalar Pipeline, con't

- `ruu_writeback()`: determines completed instr's, does data forwarding to dependent waiting instr's, detects branch misprediction and on misprediction rolls the machine state back to the checkpoint and discards erroneously issued instructions

- `ruu_commit()`: in-order commits results for instr's (values copied from RUU to RegFile or LSQ to D$), RUU/LSQ entries for committed instr's freed; keeps retiring instructions at the head of RUU that are ready to commit until the head instr is one that is not ready

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special Limited # of ports	Many GP Limited # of ports	GP and rename (RUU) Many ports	many, many GP Many ports
Memory reference	embedded in many instr's	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware dependency resolution	(compiler) code scheduling

Evolution of Pipelined, SS Processors

	Year	Clock Rate	# Pipe Stages	Issue Width	OOO?	Cores /Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
Sun USPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W