16.482 / 16.561 Computer Architecture and Design

Instructor: Dr. Michael Geiger

Fall 2013

Lecture 3:

Datapath and control

Pipelining

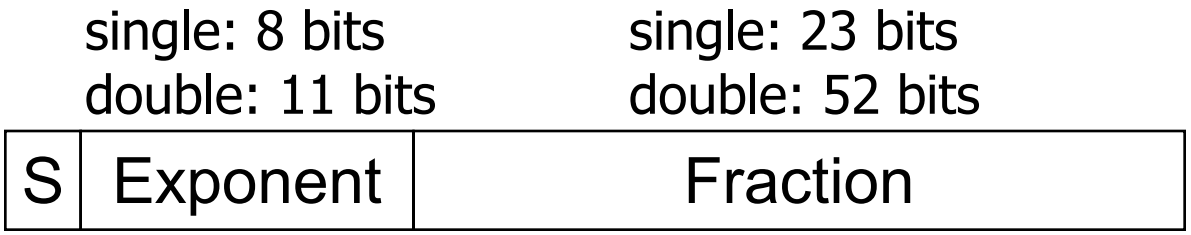
Lecture outline

- Announcements/reminders
  - HW 1 due 2/7
  - HW 2 to be posted; due 2/13
    - Will test material from today's lecture + FP arithmetic (covered last week)
- Review: Arithmetic for computers
- Today's lecture
  - Basic datapath design
  - Single-cycle datapath
  - Pipelining

Review: Binary multiplication

- Generate shifted partial products and add them
- Hardware can be condensed to two registers
  - N-bit multiplicand
  - 2N-bit running product / multiplier
  - At each step
    - Check LSB of multiplier
    - Add multiplicand/0 to left half of product/multiplier
    - Shift product/multiplier right
- Signed multiplication: Booth's algorithm
  - Add extra bit to left of all regs, right of prod./multiplier
  - At each step
    - Check two rightmost bits of prod./multiplier
    - Add multiplicand, -multiplicand, or 0 to left half of prod./multiplier
    - Shift product multiplier right
  - Discard extra bits to get final product

Review: IEEE Floating-Point Format



$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

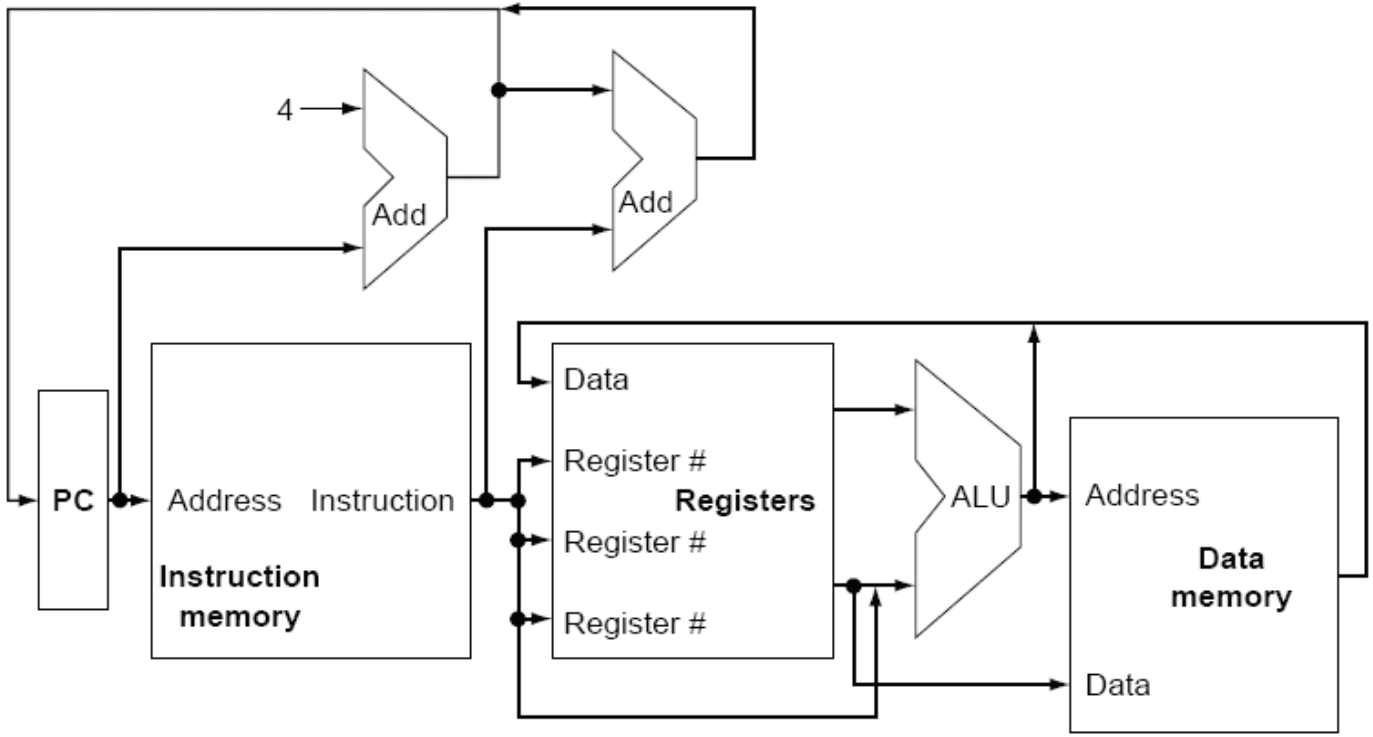
- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Significand is Fraction with the "1." restored
- Actual exponent = (encoded value) - bias
  - Single: Bias = 127; Double: Bias = 1023
- Encoded exponents 0 and 111 ... 111 reserved
- FP addition: match exponents, add, then normalize result
- FP multiplication: add exponents, multiply significands, normalize results

Datapath & control intro

- Recall: How does a processor execute an instruction?
  1. Fetch the instruction from memory
  2. Decode the instruction
  3. Determine addresses for operands
  4. Fetch operands
  5. Execute instruction
  6. Store result (and go back to step 1 …)
- First two steps are the same for all instructions
- Next steps are instruction-dependent

Basic processor implementation

- Shows *datapath elements*: logic blocks that operate on or store data
- Need *control signals* to specify multiplexer selection, functional unit operations



Implementation issues: datapath / control

- Basic implementation shows *datapath elements:* logic blocks that either
  - Operate on data (ALU)
  - Store data (registers, memory)
- Need some way to determine
  - What data to use
  - What operations to perform
- *Control signals:* signals used for
  - Multiplexer selection
  - Specifying functional unit operation

Design choices

- Single-cycle implementation
  - Every instruction takes one clock cycle
  - Instruction fetch, decoding, execution, memory access, and result writeback in same cycle
  - Cycle time determined by longest instruction
  - Will discuss first as a means for simply introducing the different hardware required to execute each instruction
- Pipelined implementation
  - Simultaneously execute multiple instructions in different stages of execution

MIPS subset

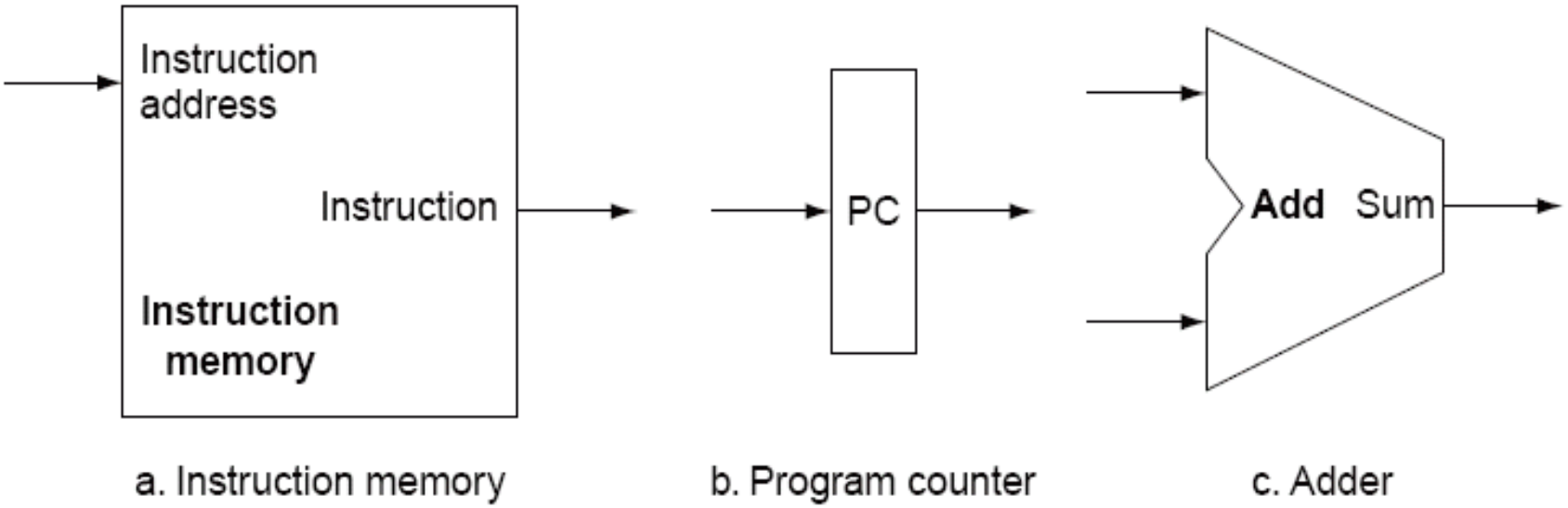
- We'll go through simple processor design for subset of MIPS ISA
  - `add, sub, and, or` (also immediate versions)
  - `lw, sw`
  - `slt`
  - `beq, j`
- Datapath design
  - What elements are necessary (and which ones can be re-used for multiple instructions)?
- Control design
  - How do we get the datapath elements to perform the desired operations?
- Will then discuss other operations

Datapath design

- What steps in the execution sequence do all instructions have in common?
  1. Fetch the instruction from memory
  2. Decode the instruction
- Instruction fetch brings data in from memory
- Decoding determines control over datapath

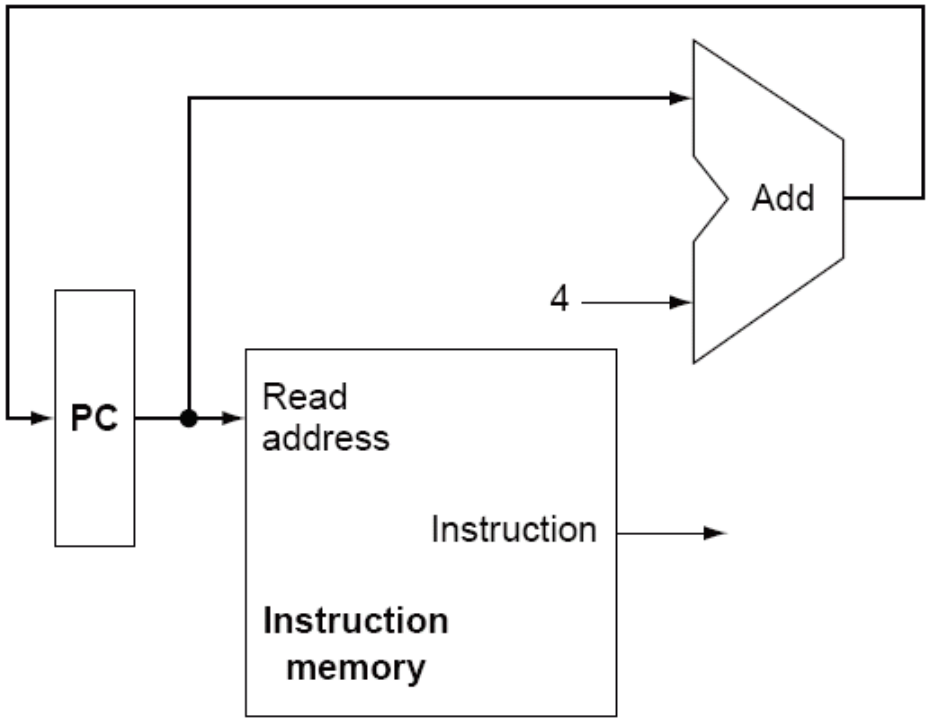
Instruction fetch

- What logic is necessary for instruction fetch?
  - Instruction storage → Memory
  - Register to hold instruction address → *program counter*
  - Logic to generate next instruction address
    - Sequential code is easy: PC + 4
    - Jumps/branches more complex (we'll discuss later)



Instruction fetch (cont.)

- *Loader* places starting address of main program in PC
- PC + 4 points to next sequential address in main memory

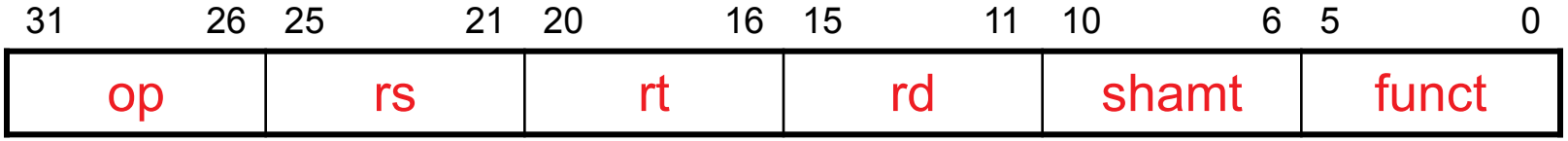


Executing MIPS instructions

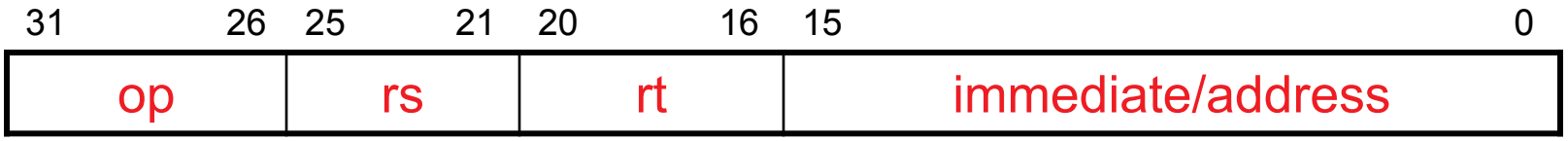
- Look at different classes within our subset (ignore immediates for now)
  - `add, sub, and, or`
  - `lw, sw`
  - `slt`
  - `beq, j`
- 2 things processor does for all instructions except `j`
  - Reads operands from register file
  - Performs an ALU operation
- 1 additional thing for everything but `sw`, `beq`, and `j`
  - Processor writes a result into register file

Instruction decoding

- Generate control signals from instruction bits
- Recall: MIPS instruction formats
  - Register instructions: R-type



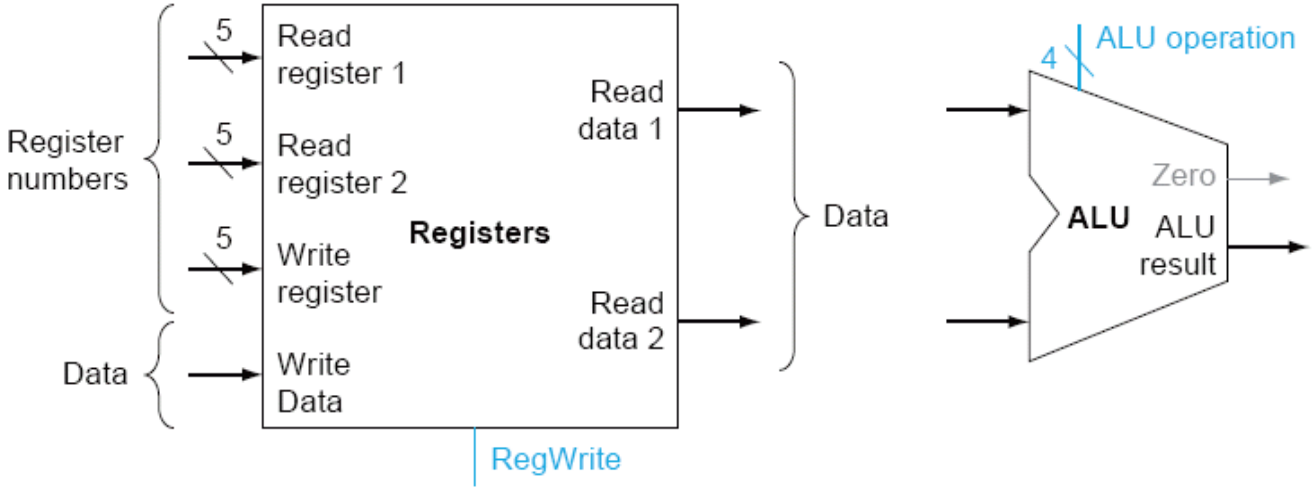
  - Immediate instructions: I-type



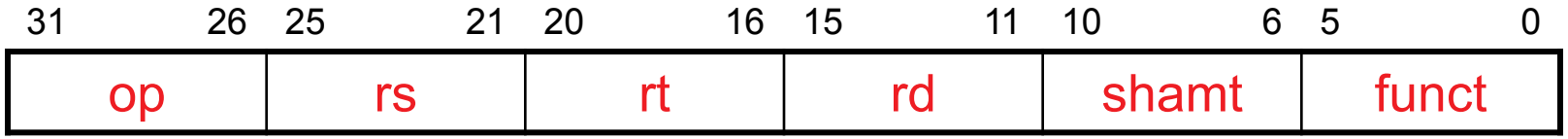
  - Jump instructions: J-type

R-type execution datapath

- R-type instructions (in our subset)
  - `add, sub, and, or, slt`
- What logic is necessary?
  - Register file to provide operands
  - ALU to operate on data

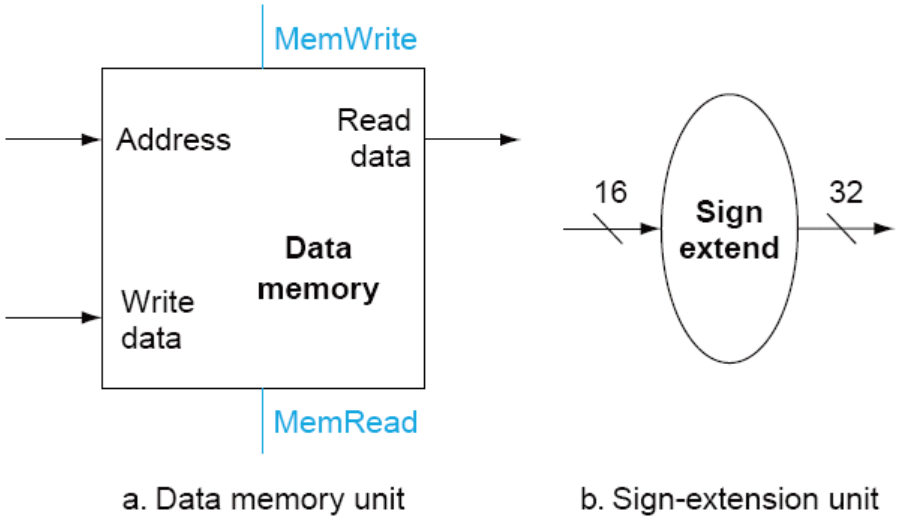


a. Registers b. ALU



Memory instructions

- `lw, sw` → I-type instructions
- Use register file
  - Base address
  - Source data for store
  - Destination for load
- Compute address using ALU
- What additional logic is necessary?
  - Data memory
  - Sign extension logic (why?)



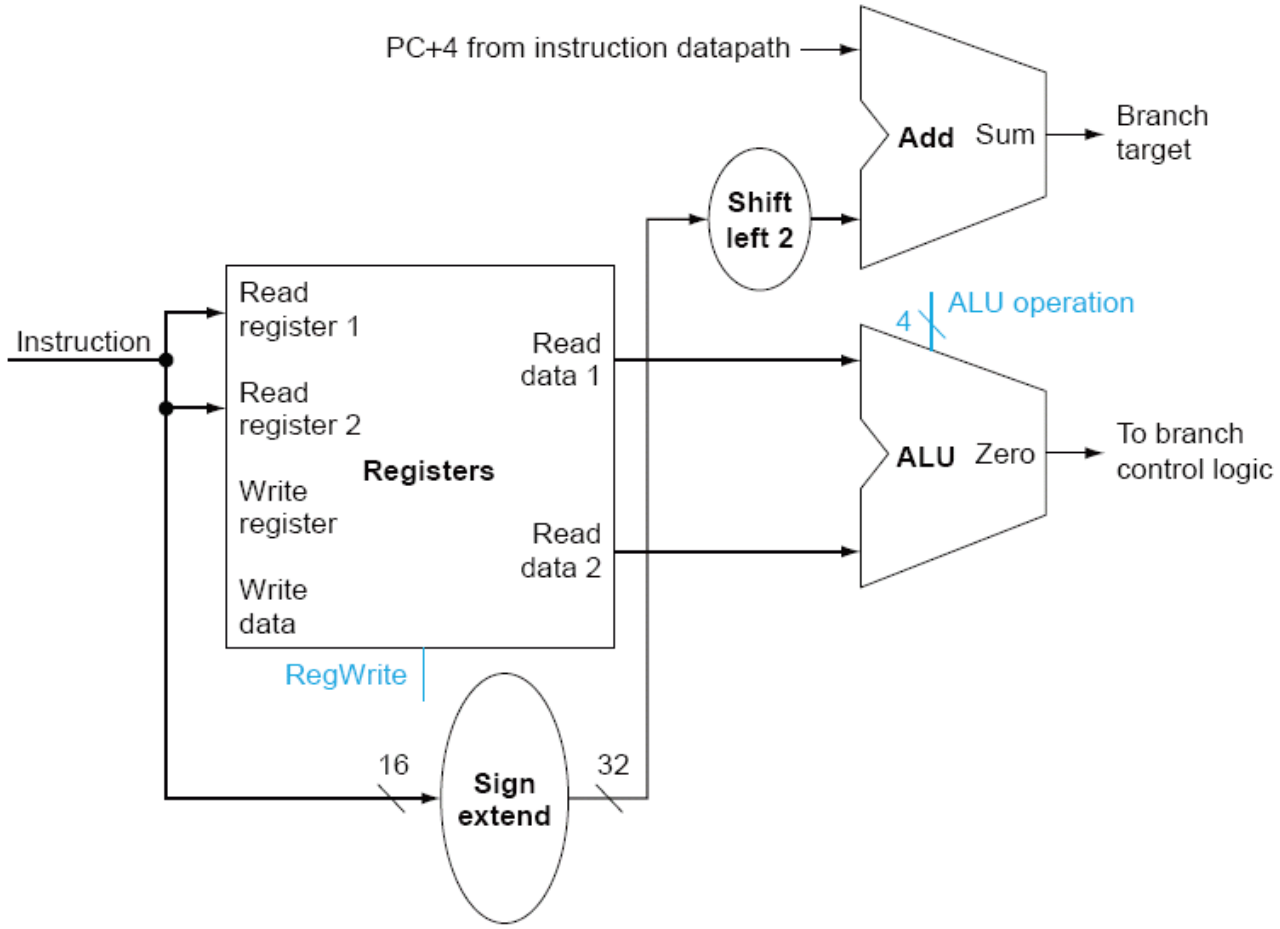
a. Data memory unit

b. Sign-extension unit

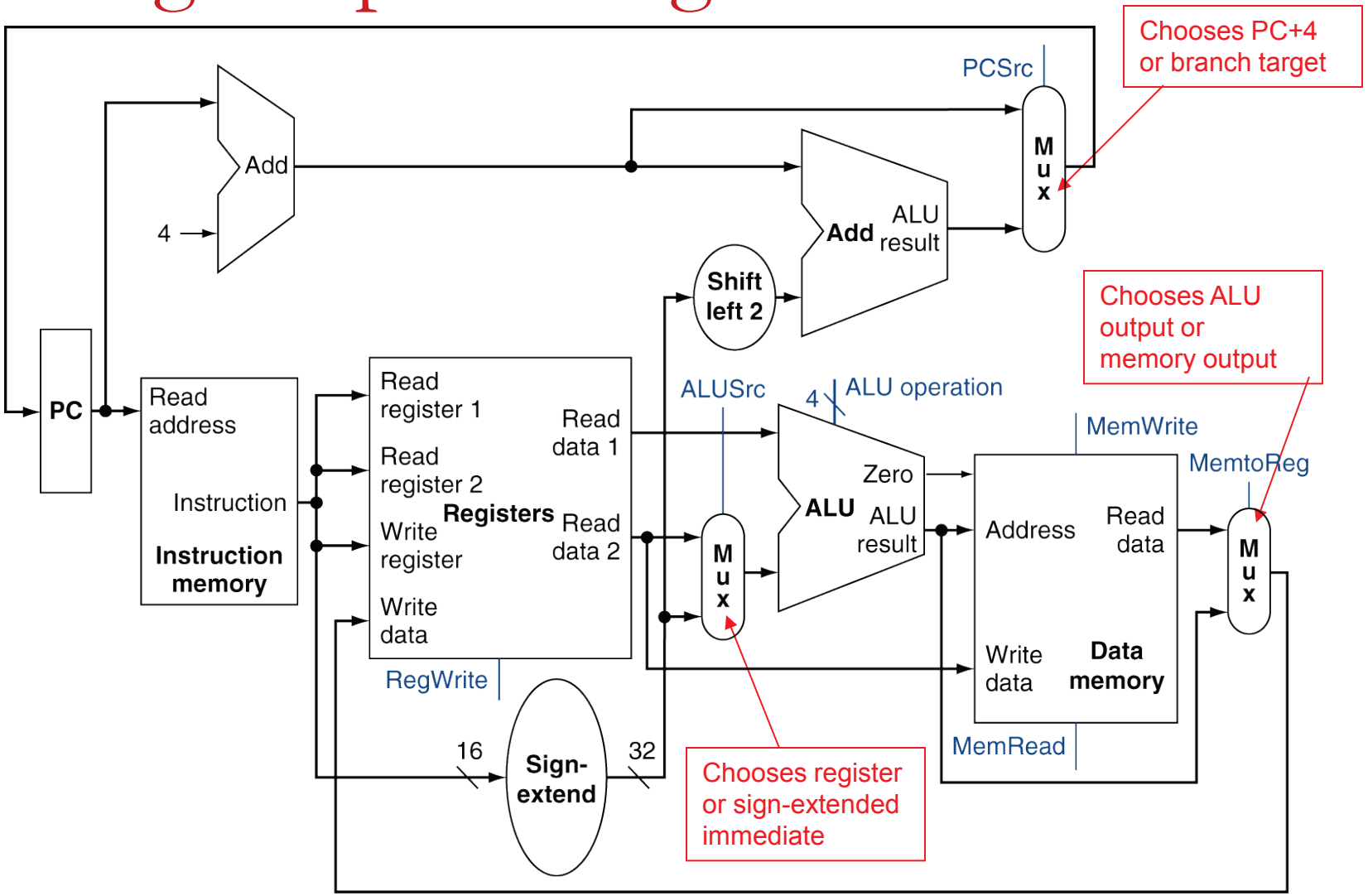
Branch instructions

- Simple `beq` implementation
  - Branch if (a – b) == 0; use ALU to evaluate condition
- Branch address = (PC + 4) + (offset << 2)
  - Where's offset encoded?
    - Immediate field
  - Why shift by 2?
    - Instruction addresses word aligned—2 least significant bits are 0

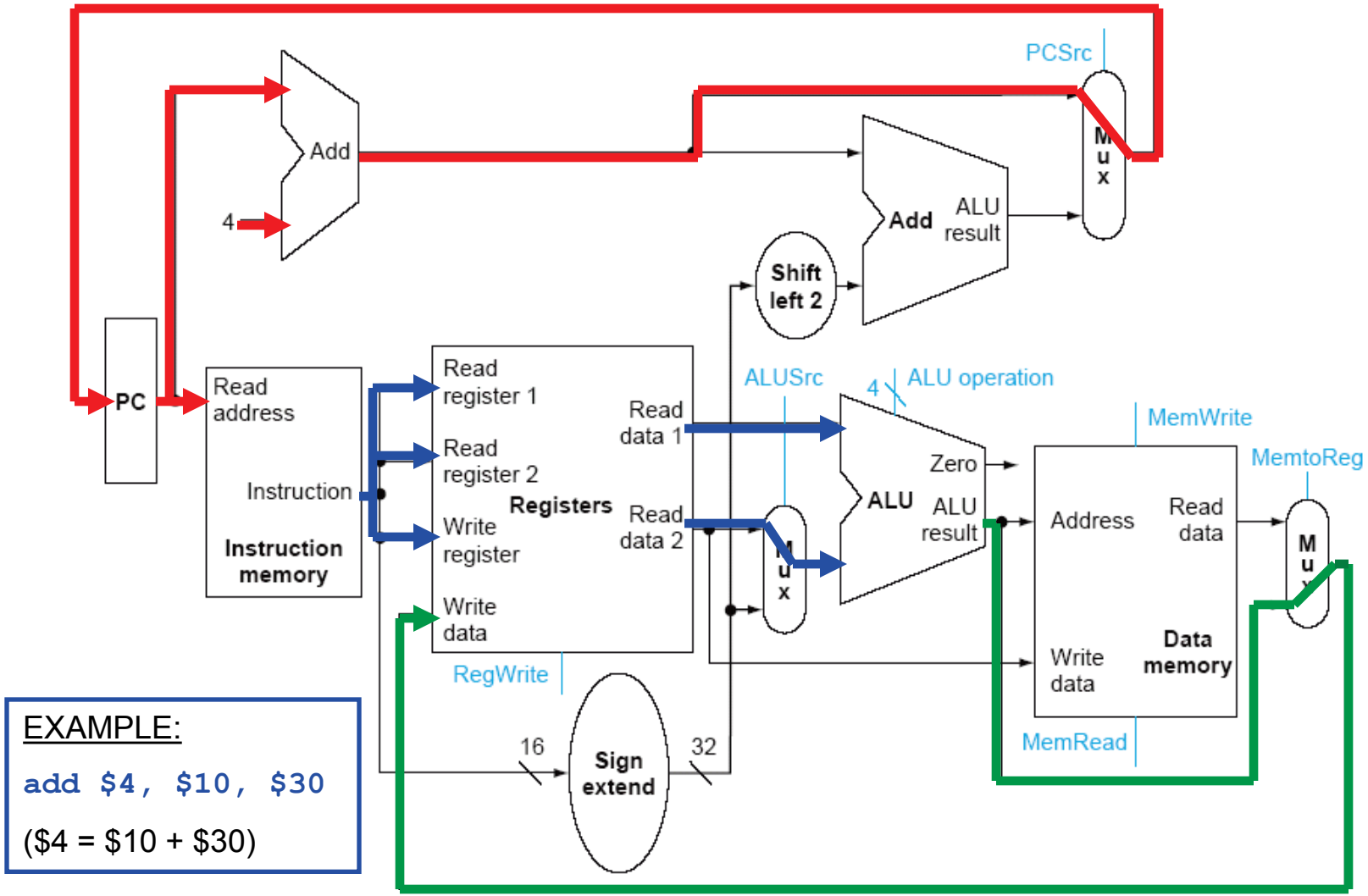
Branch instruction datapath



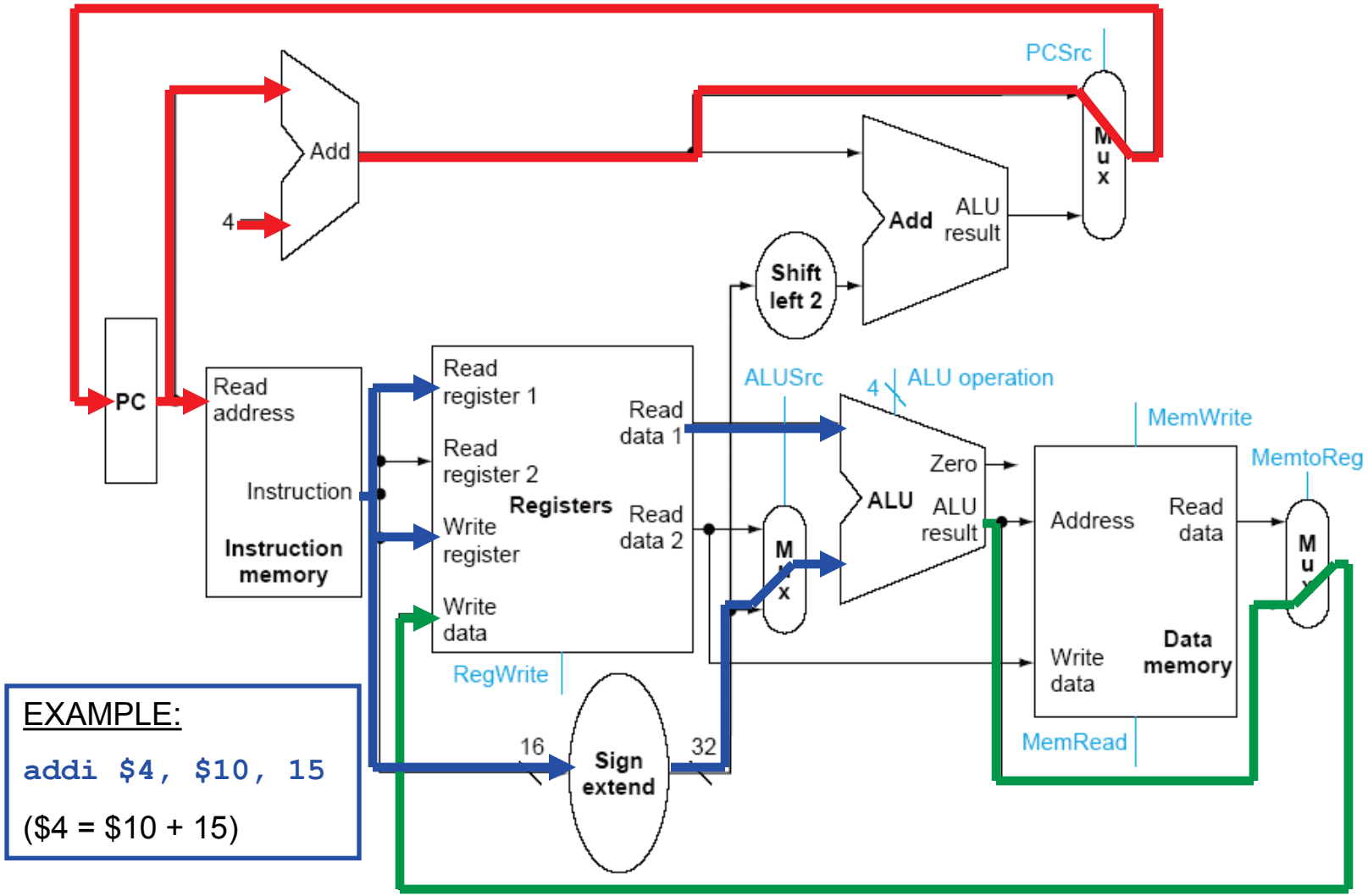
Putting the pieces together



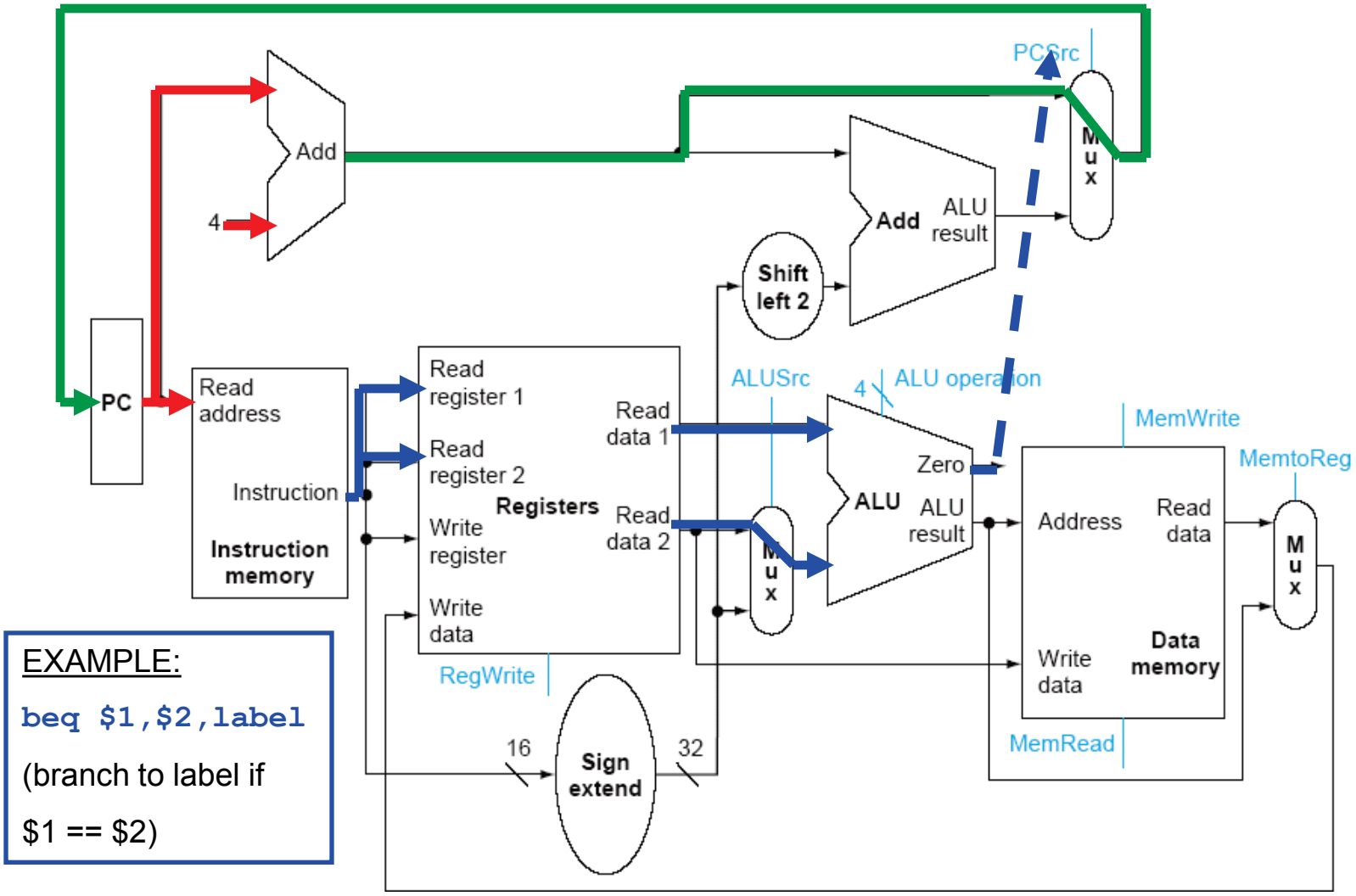
Datapath for R-type instructions



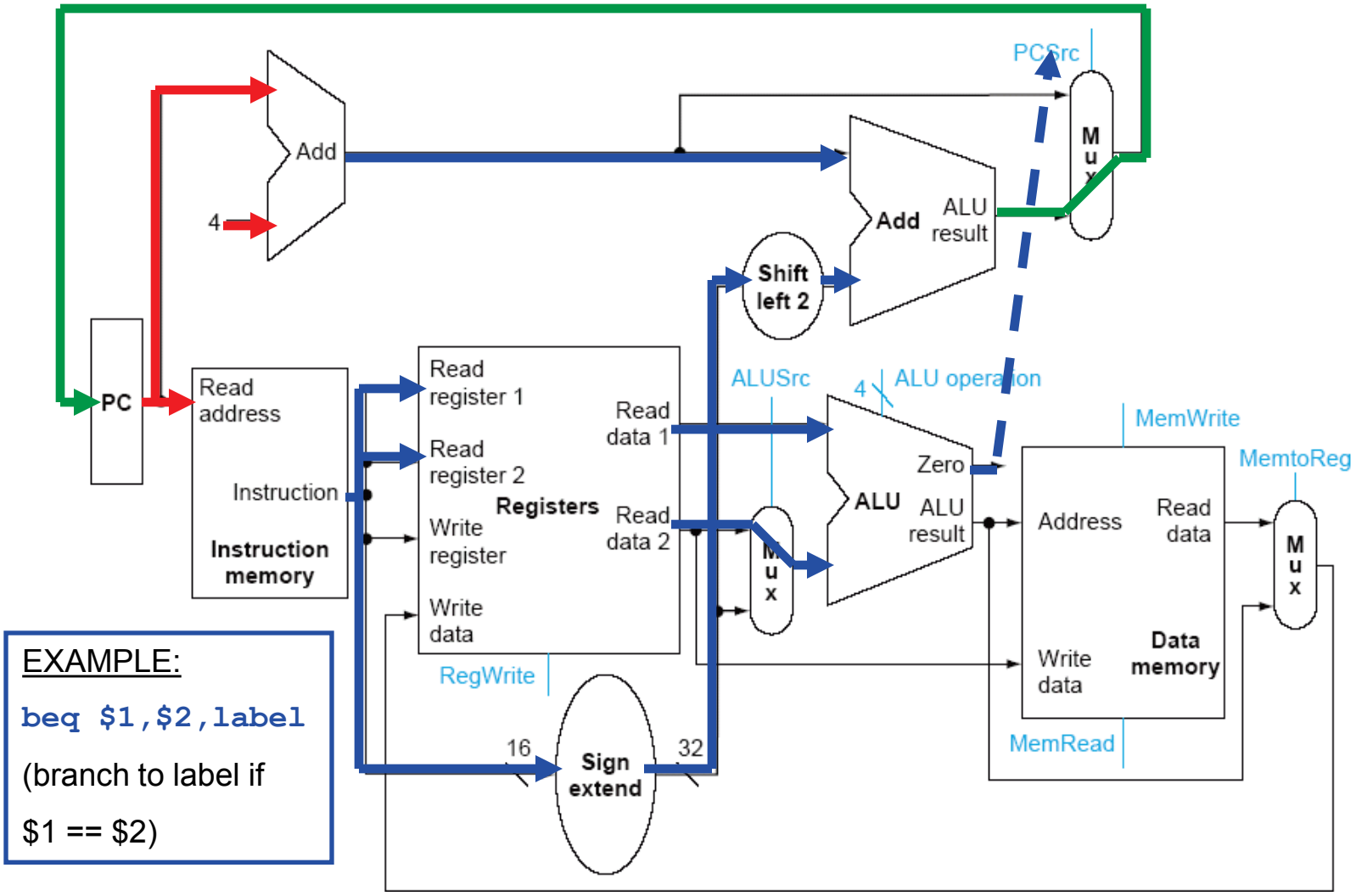
Datapath for I-type ALU instructions



Datapath for beq (not taken)



Datapath for beq (taken)



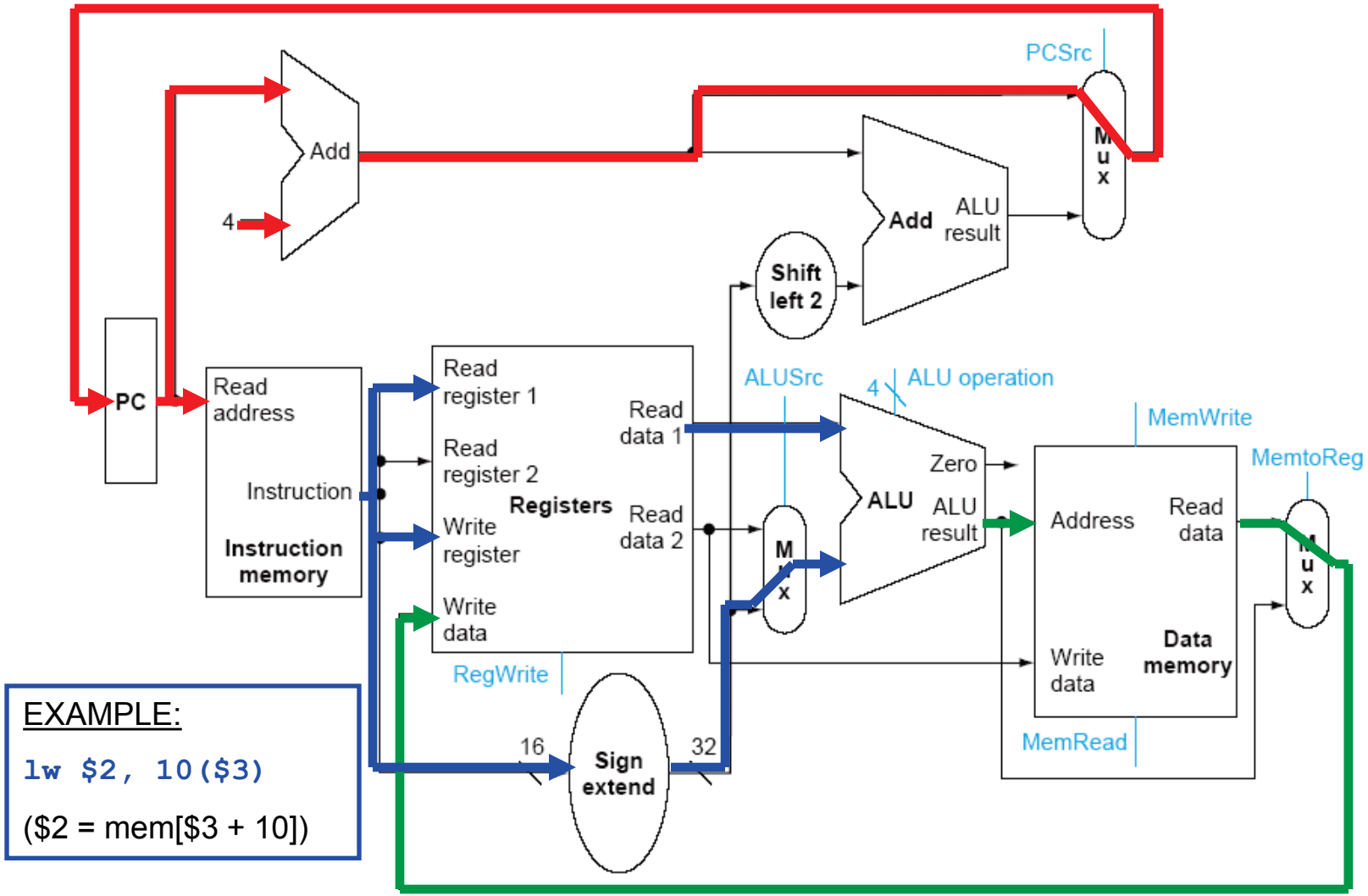
EXAMPLE:

`beq $1,$2,label`

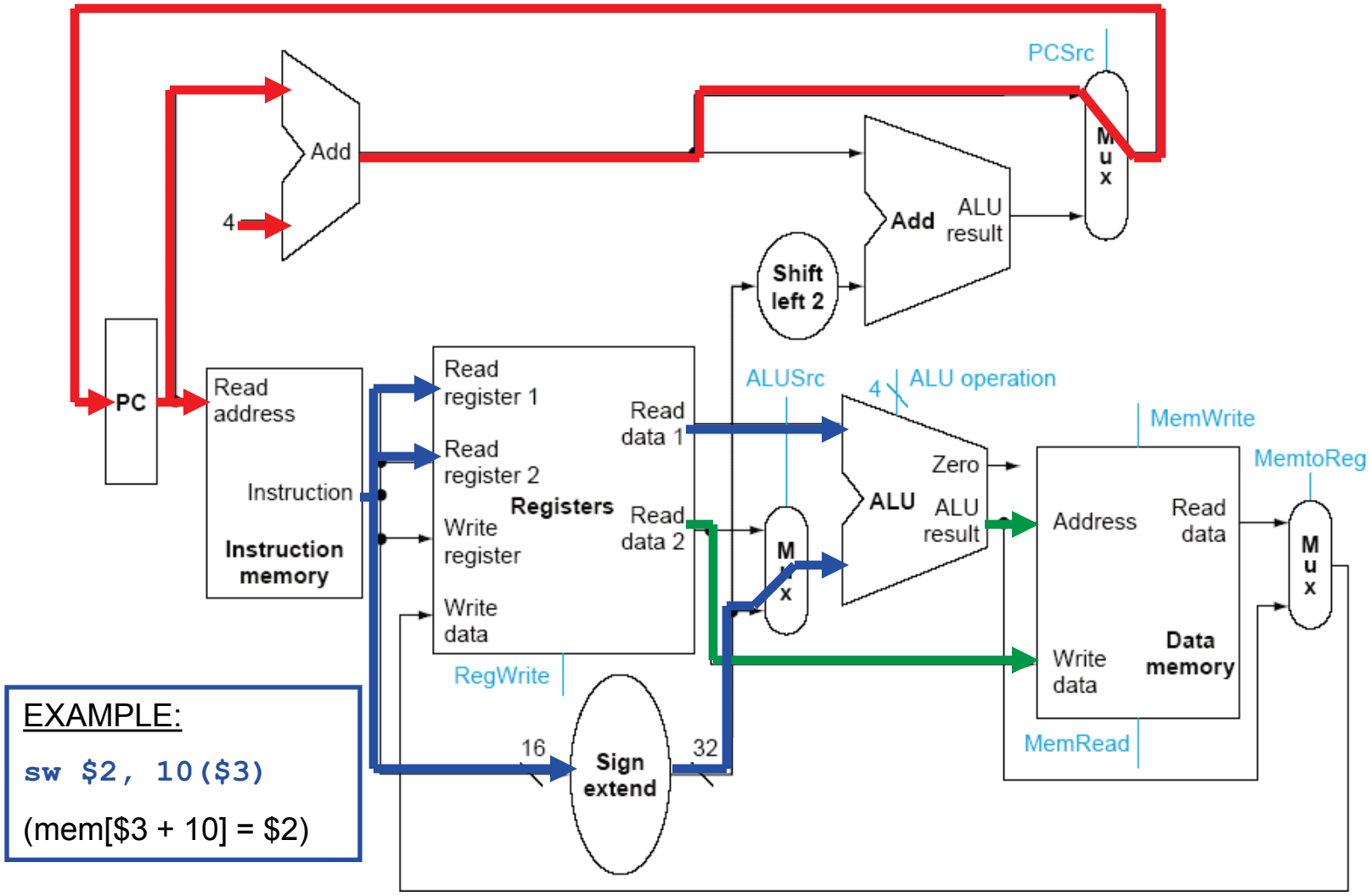
(branch to label if

$1 == $2)

Datapath for `lw` instruction

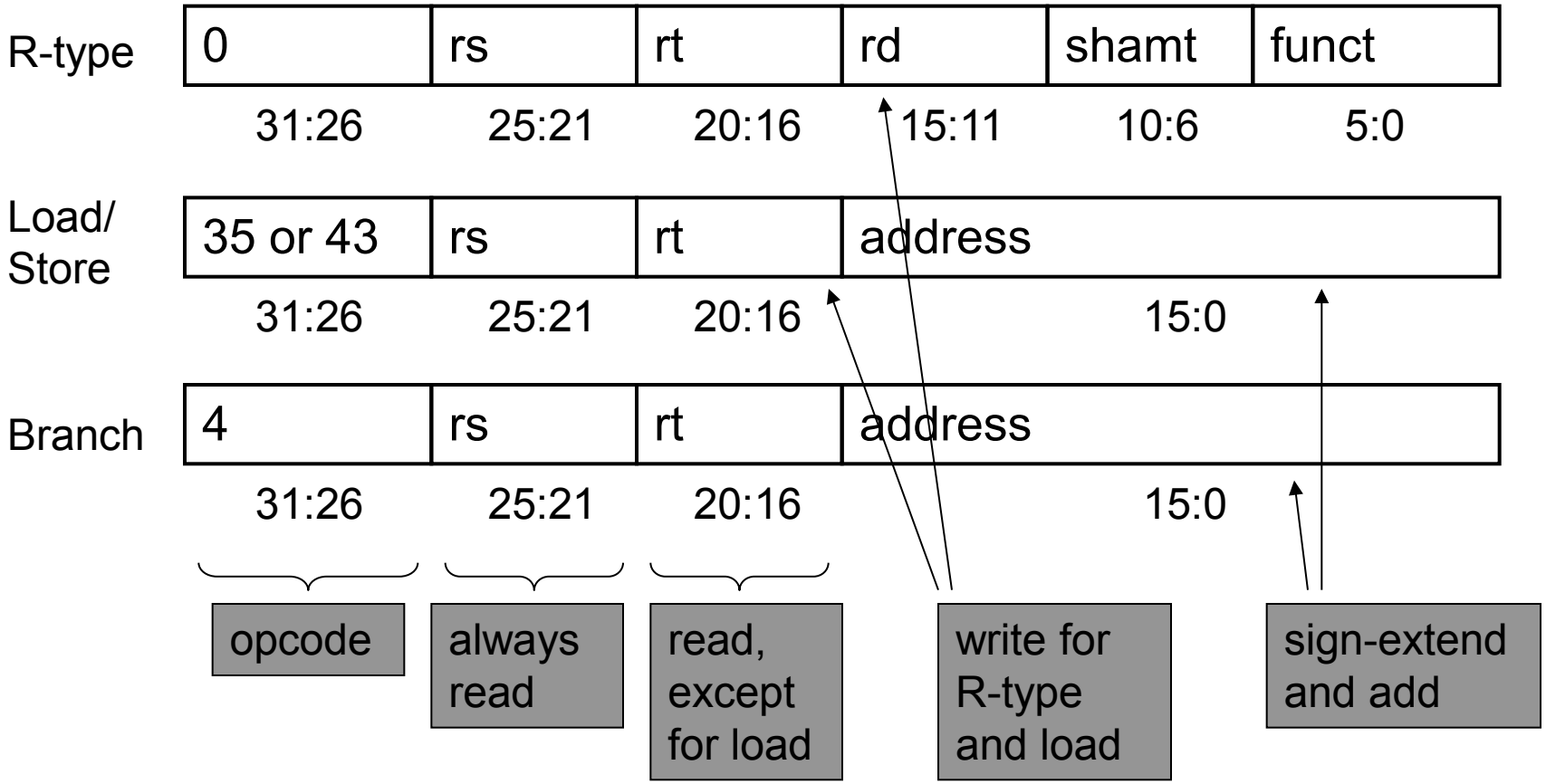


Datapath for sw instruction

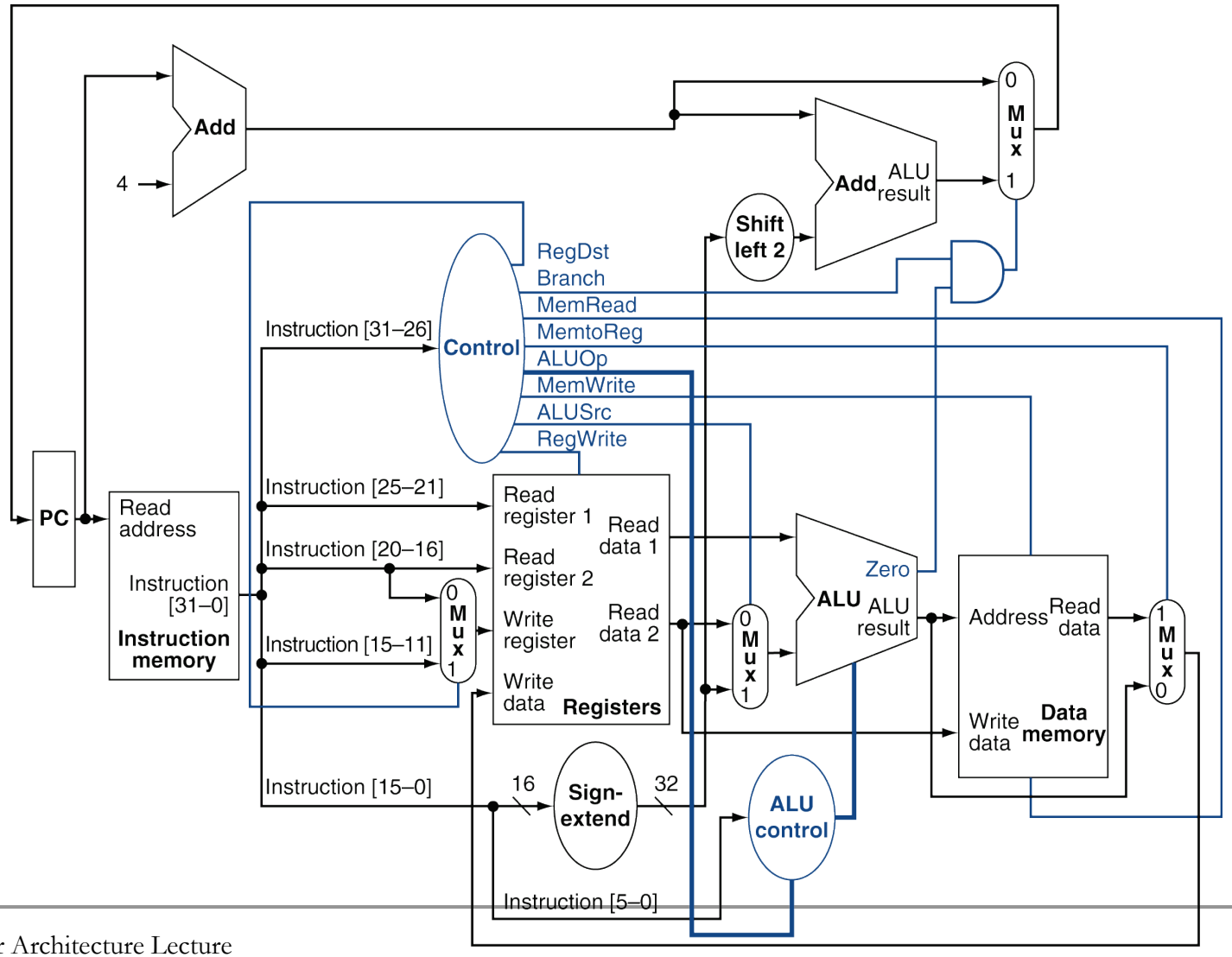


The Main Control Unit

- Control signals derived from instruction



Datapath With Control

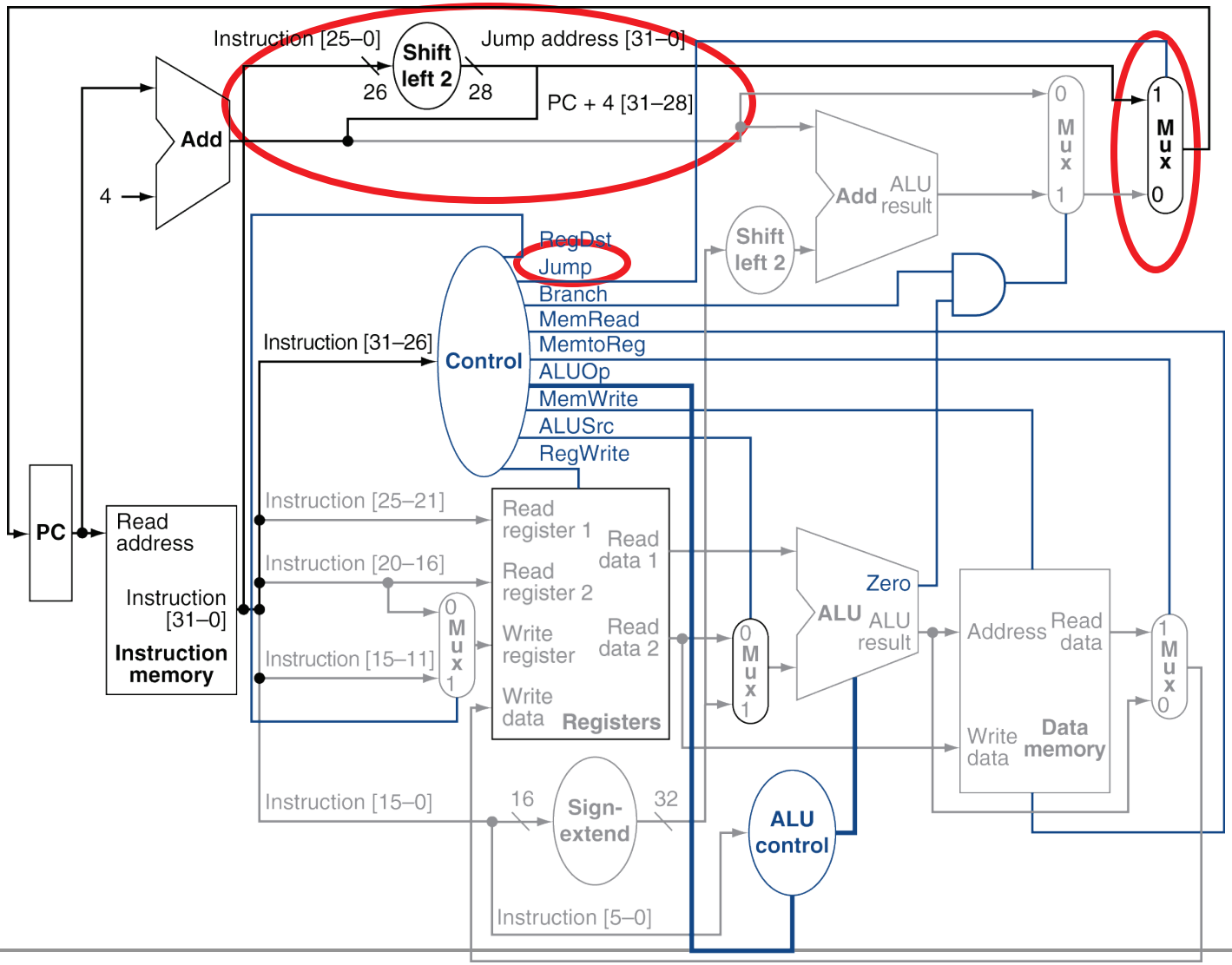


Adding `j` instruction

- J-type format

- Next PC = (PC+4)[31:28] combined with (target<<2)
- 3 choices for next PC
  - PC + 4
  - `beq` target address
  - `j` target address
- Need another multiplexer and control bit!

Datapath with jump support



Control signals

- PCWriteCond: conditionally enables write to PC if branch condition is true (if Zero flag = 1)
- PCWrite: enables write to PC; used for both normal PC increment and jump instructions
- IorD: indicates if memory address is for instruction (i.e., PC) or data (i.e., ALU output)
- MemRead, MemWrite: controls memory operation
- MemtoReg: determines if write data comes from memory or ALU output
- IRWrite: enables writing of instruction register

Control signals (cont.)

- PCSource: controls multiplexer that determines if value written to PC comes from ALU (PC = PC+4), branch target (stored ALU output), or jump target
- ALUOp: feeds ALU control module to determine ALU operation
- ALUSrcA: controls multiplexer that chooses between PC or register as A input of ALU
- ALUSrcB: controls multiplexter that chooses between register, constant ‘4’, sign-extended immediate, or shifted immediate as B input of ALU
- RegWrite: enables write to register file
- RegDst: determines if rt or rd field indicates destination register

Summary of execution phases

Phase	R-type	Memory (lw/sw)	Branch/Jump
Instruction fetch	IR = Mem[PC] PC = PC + 4		
Instruction decode/ register fetch	A = Reg[IR[25:21]] B = Reg[IR[20:16]] ALUout = PC + (sign-extend(IR[15:0]) << 2)		
Execution, address computation, branch and jump completion	ALUout = A op B	ALUout = A + sign-extend(IR[15:0])	Branch: if (A == B) PC = ALUout Jump: PC = {PC[31:28],IR[25:0],(00)}
Memory access, R-type completion	Reg[IR[15:11]] = ALUout	Load: MDR = Mem[ALUout] Store: Mem[ALUout] = B	
Memory read completion		Load: Reg[IR[20:16]] = MDR	

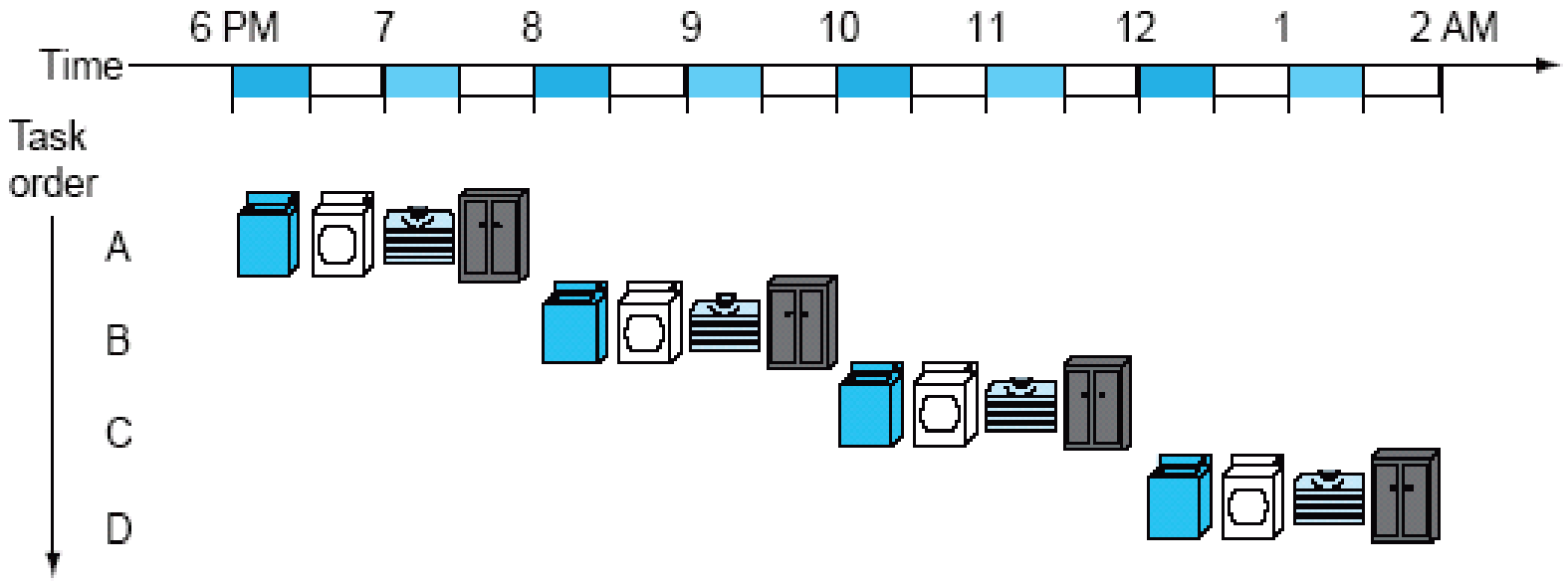
Motivating pipelining

- We've seen basic single-cycle datapath
  - Offers 1 CPI ...
  - ... but cycle time determined by longest instruction
    - Load essentially uses all stages
- We'd like both low CPI and a short cycle
- Solution: pipelining
  - Simultaneously execute multiple instructions
  - Use multi-cycle "assembly line" approach

Pipelining is like …

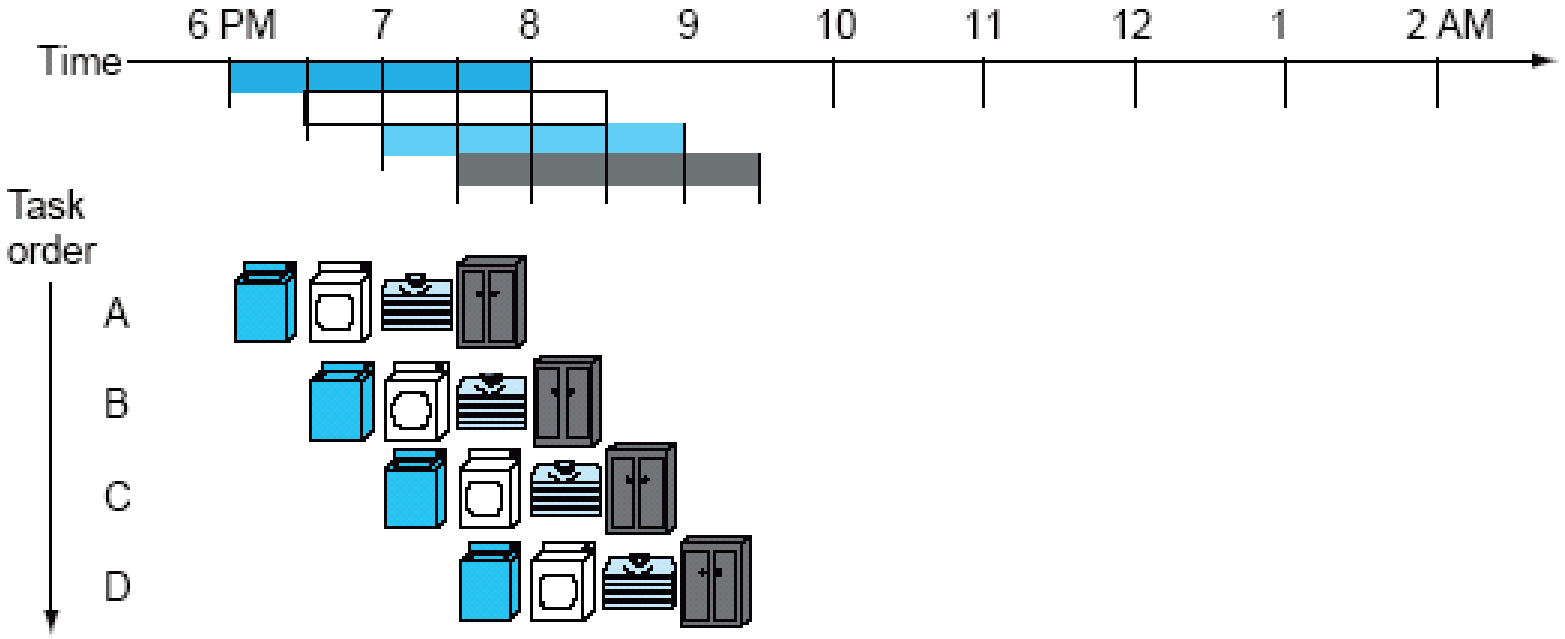
- … doing laundry (no, really)
- Say 4 people (Ann, Brian, Cathy, Don) want to use a laundry service that has four components:
  - Washer, which takes 30 minutes
  - Dryer, which takes 30 minutes
  - "Folder," which takes 30 minutes
  - "Storer," which takes 30 minutes

Sequential laundry service



- Each person starts when previous one finishes
- 4 loads take 8 hours

Pipelined laundry service



- As soon as a particular component is free, next person can use it
- 4 loads take 3 ½ hours

Pipelining questions

- Does pipelining improve latency or throughput?
  - Throughput—time for each instruction same, but more instructions per unit time
- What's the maximum potential speedup of pipelining?
  - The number of stages, *N*—before, each instruction took *N* cycles, now we can (theoretically) finish 1 instruction per cycle
- If one stage can run faster, how does that affect the speedup?
  - No effect—cycle time depends on longest stage, because you may be using hardware from all stages at once

Principles of pipelining

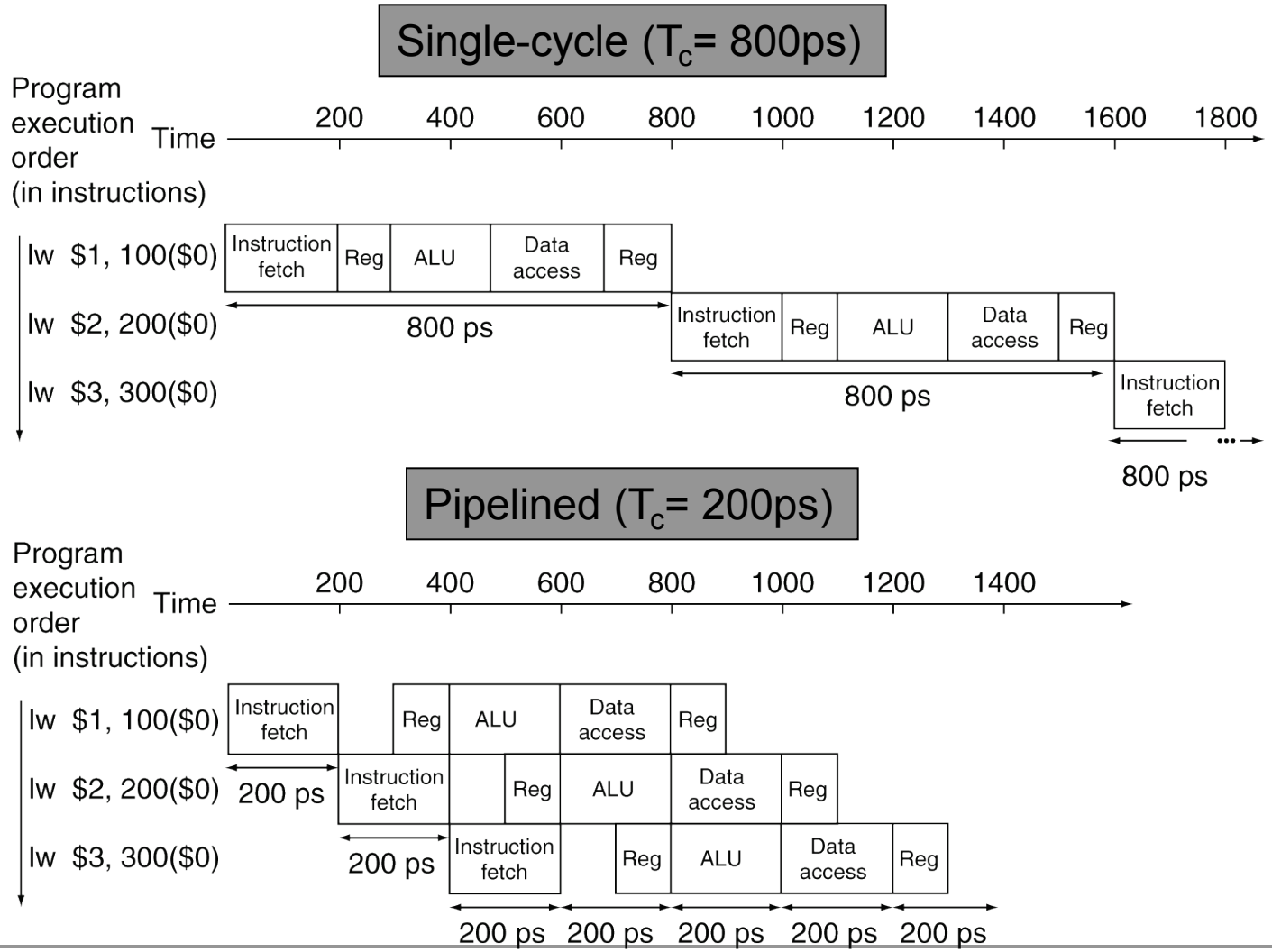
- Every instruction takes same number of steps
  - Pipeline stages
  - 1 stage per cycle (like multi-cycle datapath)
  - MIPS (like most simple processors) has 5 stages
    - IF: Instruction fetch
    - ID: Instruction decode and register read
    - EX: Execution / address calculation
    - MEM: Memory access
    - WB: Write back result to register

Pipeline Performance

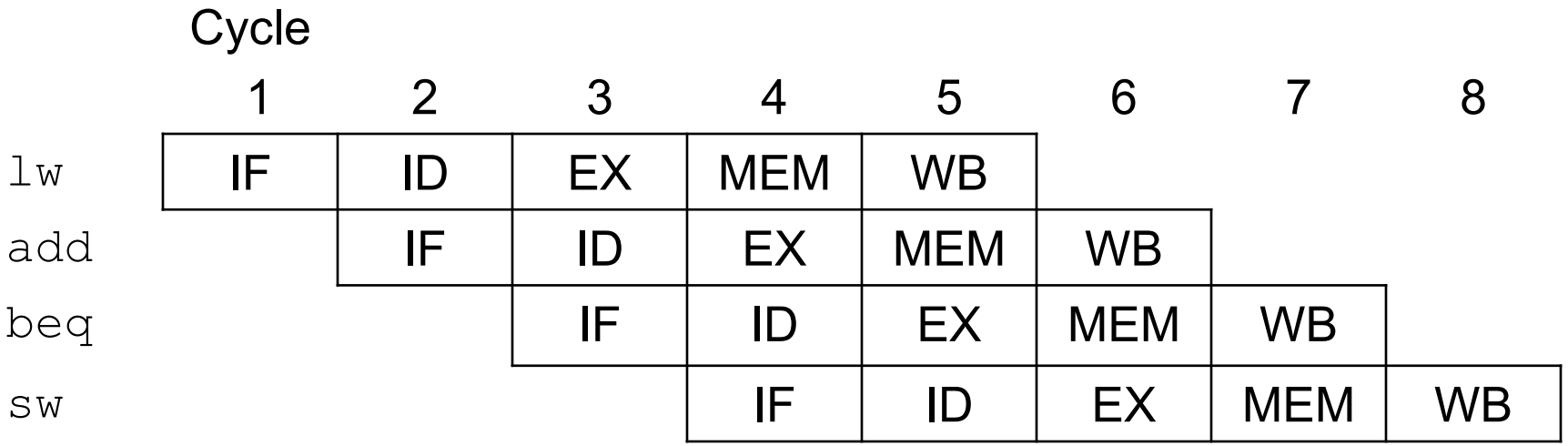
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath
  - How long does instruction take in single-cycle?

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

Pipeline Performance



Pipeline diagram



- Pipeline diagram shows execution of multiple instructions
  - Instructions listed vertically
  - Cycles shown horizontally
  - Each instruction divided into stages
  - Can see what instructions are in a particular stage at any cycle

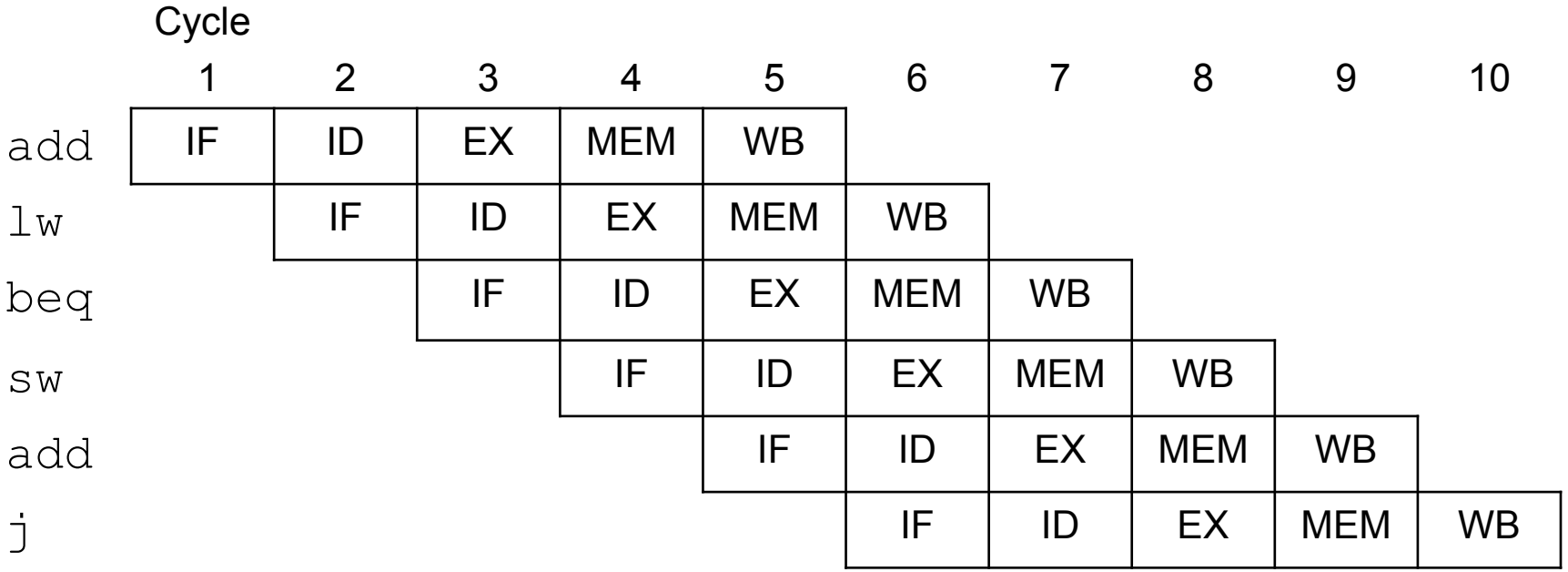
Performance example

- Say we have the following code:

```
loop:       add $t1, $t2, $t3
            lw  $t4, 0($t1)
            beq $t4, $t3, end
            sw  $t3, 4($t1)
            add $t2, $t2, 8
            j   loop
end:        ...
```

- Assume each pipeline stage takes 4 ns
- How long would one loop iteration take in an ideal pipelined processor (i.e., no delays between instructions)?
- How long would one loop iteration take in a single-cycle datapath?
- How much faster is the pipelined datapath than the single-cycle datapath? (Express your answer as a ratio, not an absolute difference.)

Solution

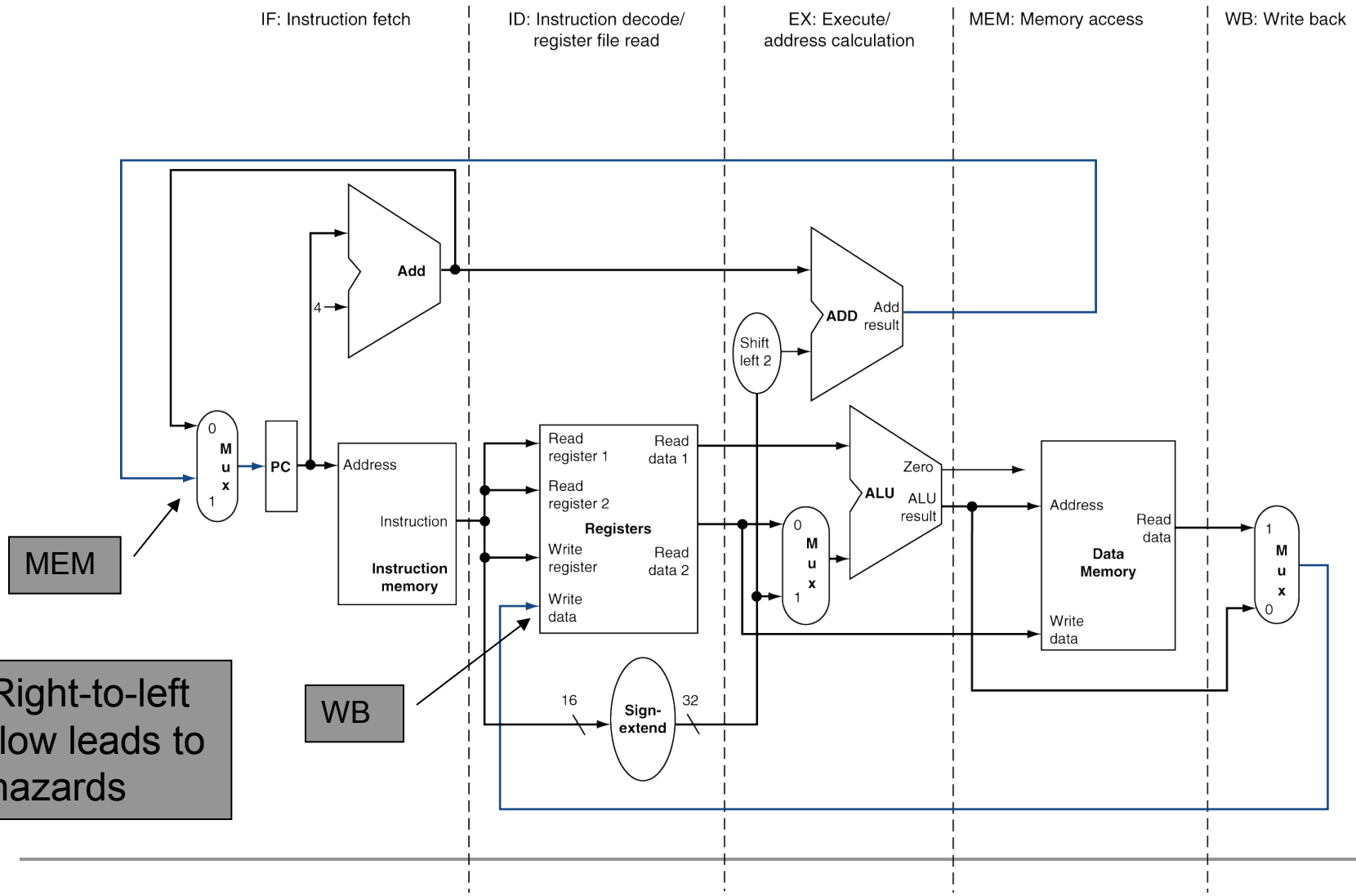


- Can draw pipelining diagram to show # cycles
- In ideal pipelining, with M instructions & N pipeline stages, total time = N + (M-1)
  - Here, M = 6, N = 5 → 5 + (6-1) = 10 cycles
  - Total time = (10 cycles) * (4 ns/cycle) = 40 ns

Solution (continued)

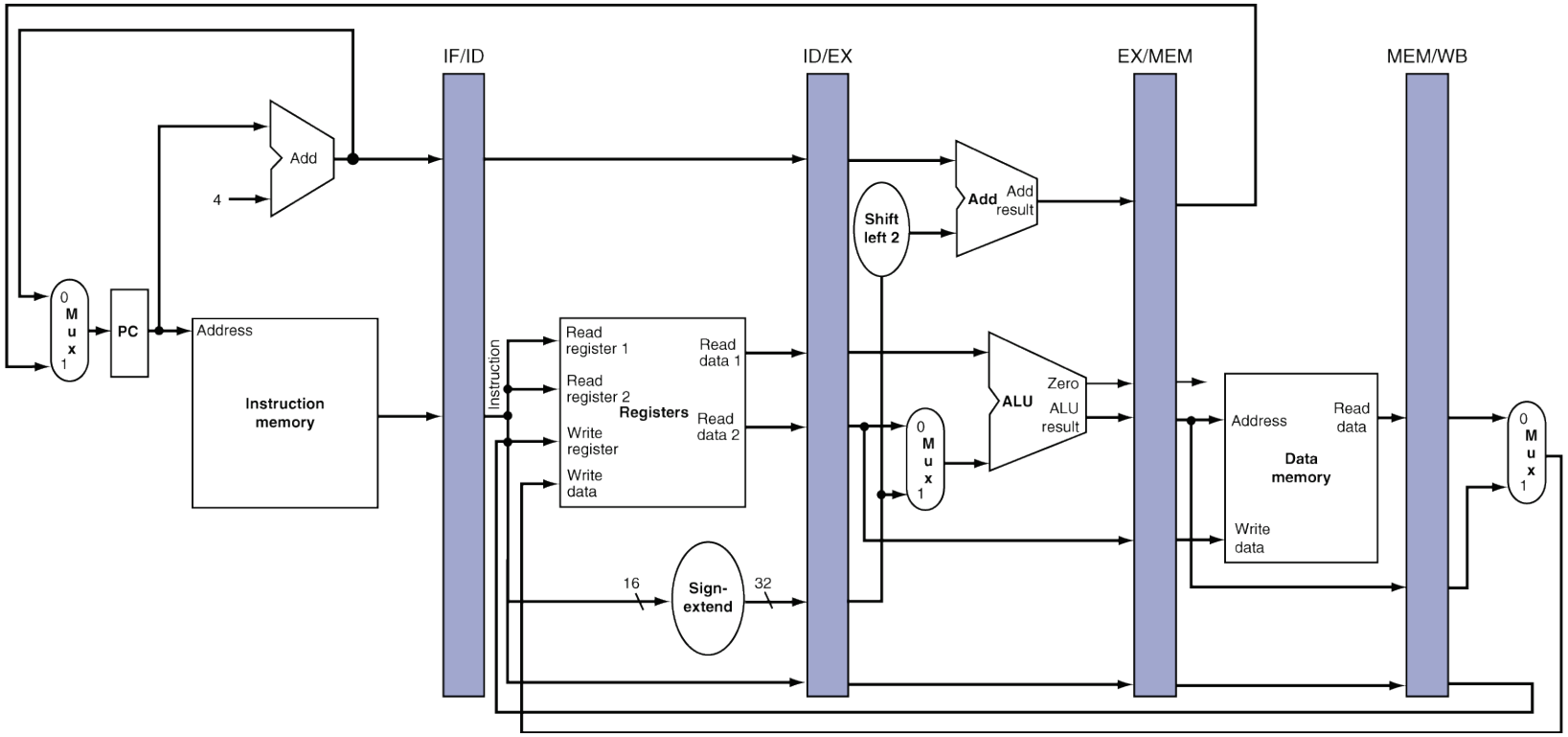
- Single-cycle datapath: longest instruction determines total instruction time
  - Longest instruction (lw) requires all 5 stages
  - 5 * (4 ns) = 20 ns per instruction
  - Total time = (6 instructions) * (20 ns) = **120 ns**
- Ratio of execution times (speedup):
  - 120 ns / 40 ns = 3 → pipelining is 3x faster

Pipelined datapath principles



Pipeline registers

- Need registers between stages for info from previous cycles
- Register must be able to hold all needed info for given stage
  - For example, IF/ID must be 64 bits—32 bits for instruction, 32 bits for PC+4
- May need to propagate info through multiple stages for later use
  - For example, destination reg. number determined in ID, but not used until WB

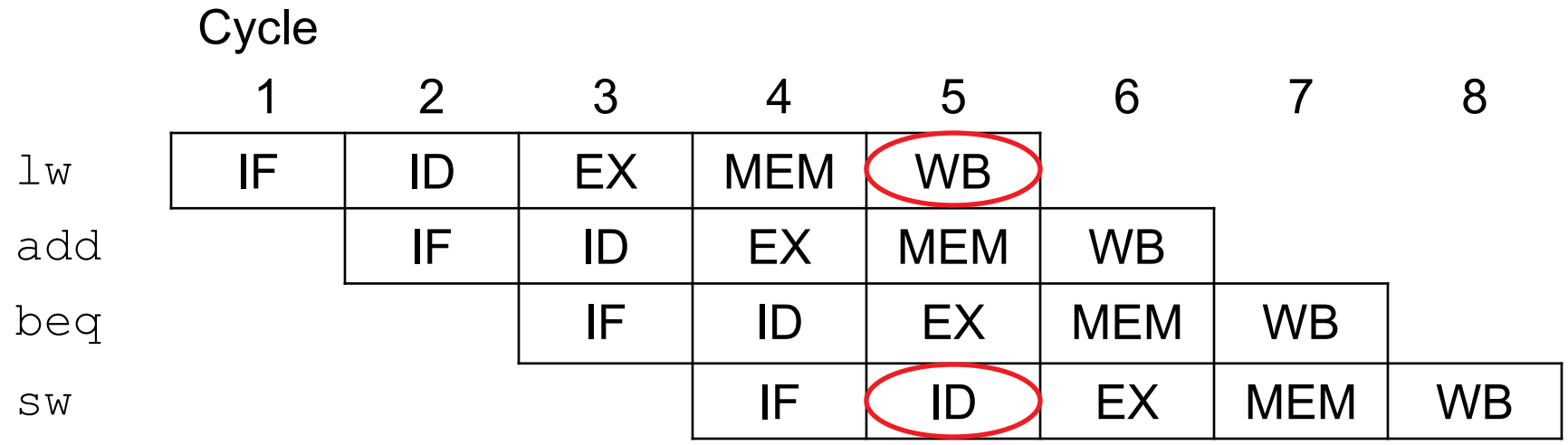


Pipeline hazards

- A hazard is a situation that prevents an instruction from executing during its designated clock cycle
- 3 types:
  - Structure hazards: two instructions attempt to simultaneously use the same hardware
  - Data hazards: instruction attempts to use data before it's ready
  - Control hazards: attempt to make a decision before condition is evaluated

Structure hazards

- Examples in MIPS pipeline
  - May need to calculate addresses and perform operations
    → need multiple adders + ALU
  - May need to access memory for both instructions and data
    → need instruction & data memories (caches)
  - May need to read and write register file
    → write in first half of cycle, read in second



Data Hazard Example

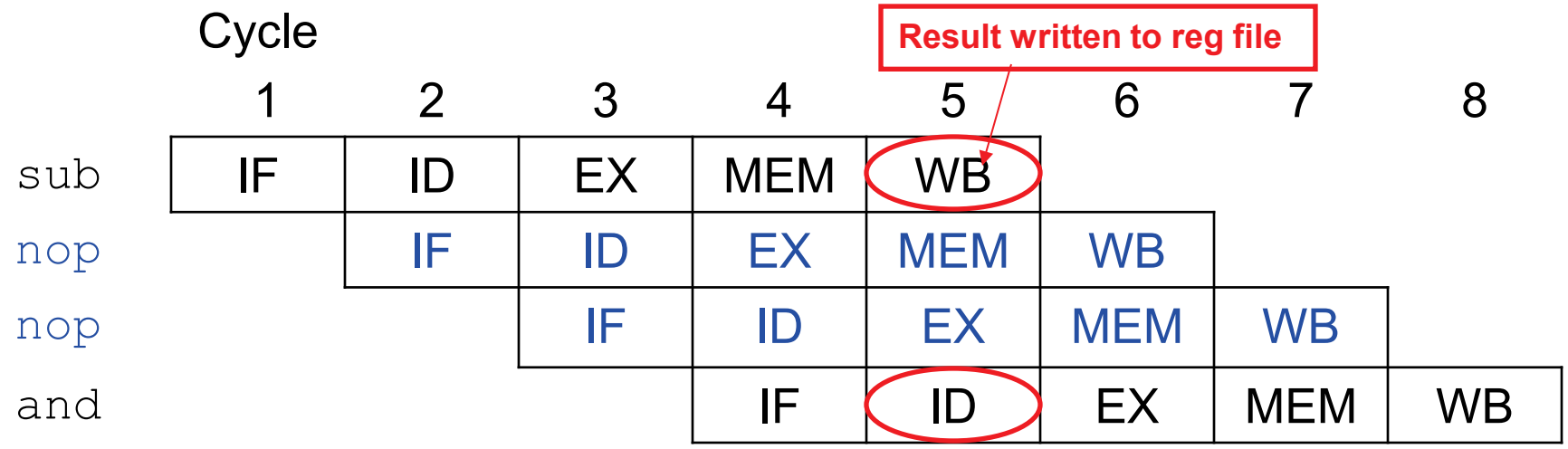
- Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

- Can't use value of $2 until it's actually computed and stored
  - No hazard for sw
  - Register hardware takes care of add
  - What about and, or?

Software solution: no-ops

- No-ops: instructions that do nothing
- Effectively "stalls" pipeline until data is ready
- Compiler can recognize hazards ahead of time and insert nop instructions



No-op example

- Given the following code, where are no-ops needed?

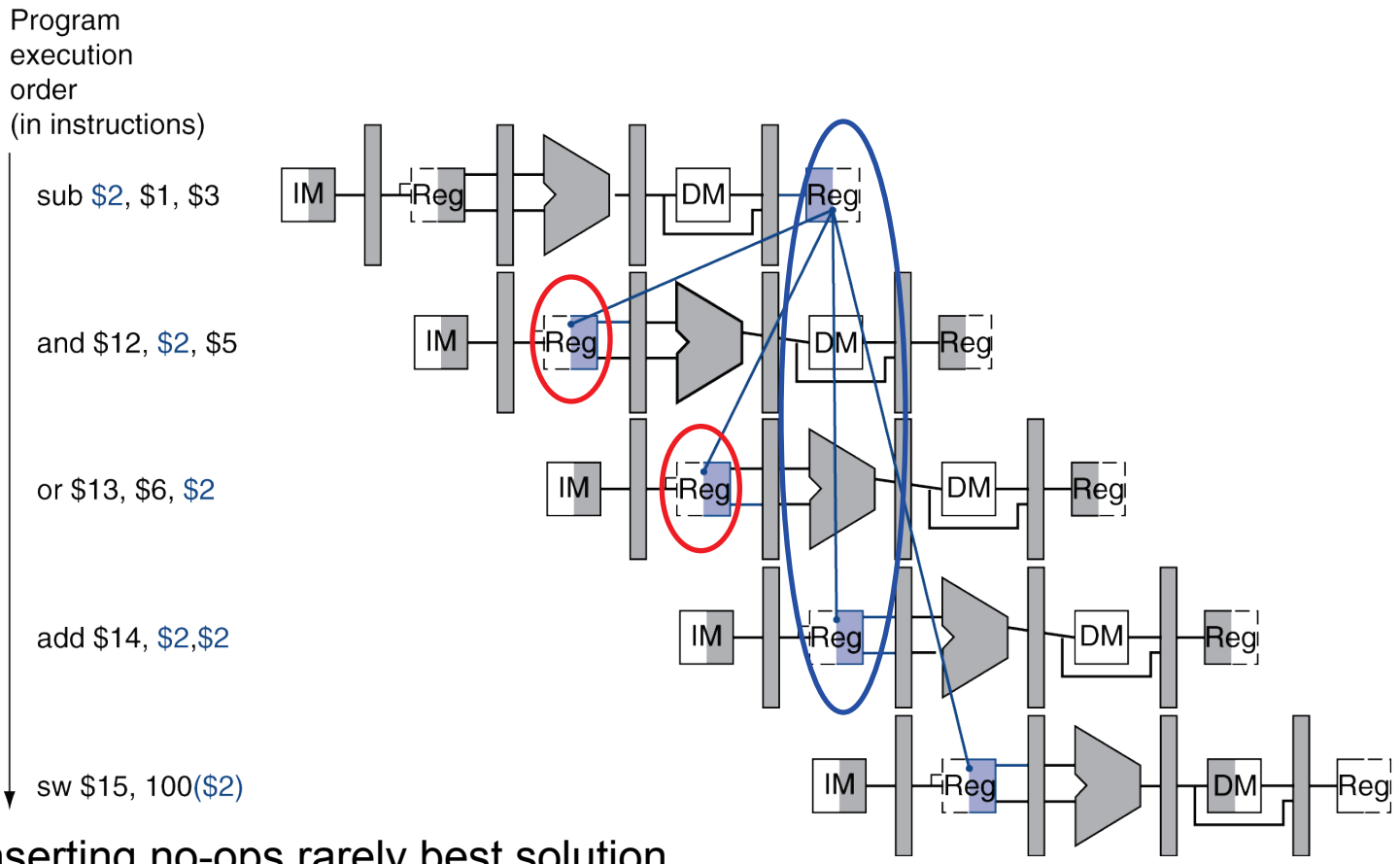
```
add $t2, $t3, $t4
sub $t5, $t1, $t2
or  $t6, $t2, $t7
slt $t8, $t9, $t5
```

Solution

- Given the following code, where are no-ops needed?

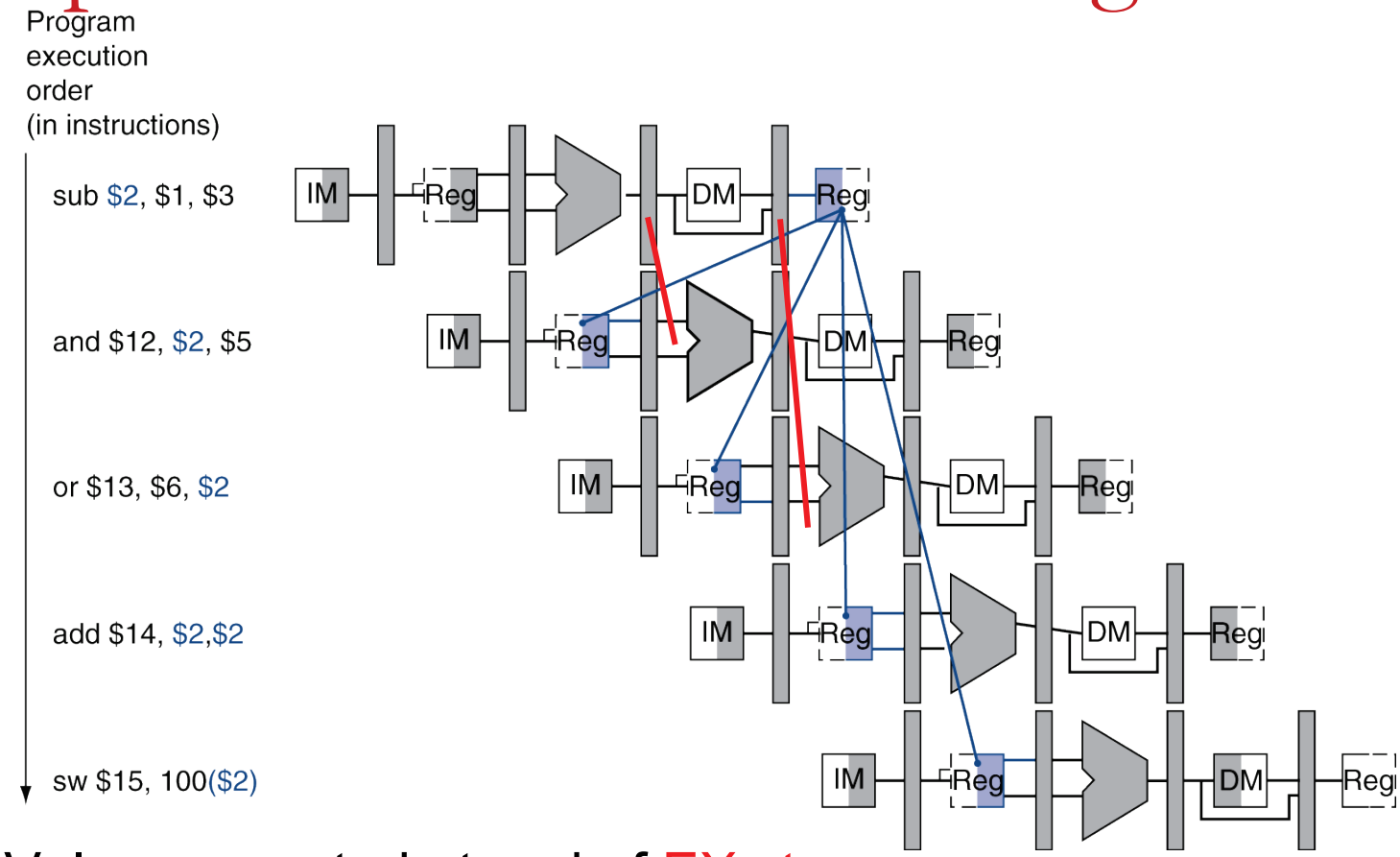
```
add $t2, $t3, $t4      ←$t2 used by sub, or
nop
nop
sub $t5, $t1, $t2      ← $t5 used by slt
or  $t6, $t2, $t7
nop                    ← could also be before or
slt $t8, $t9, $t5
```

Avoiding stalls



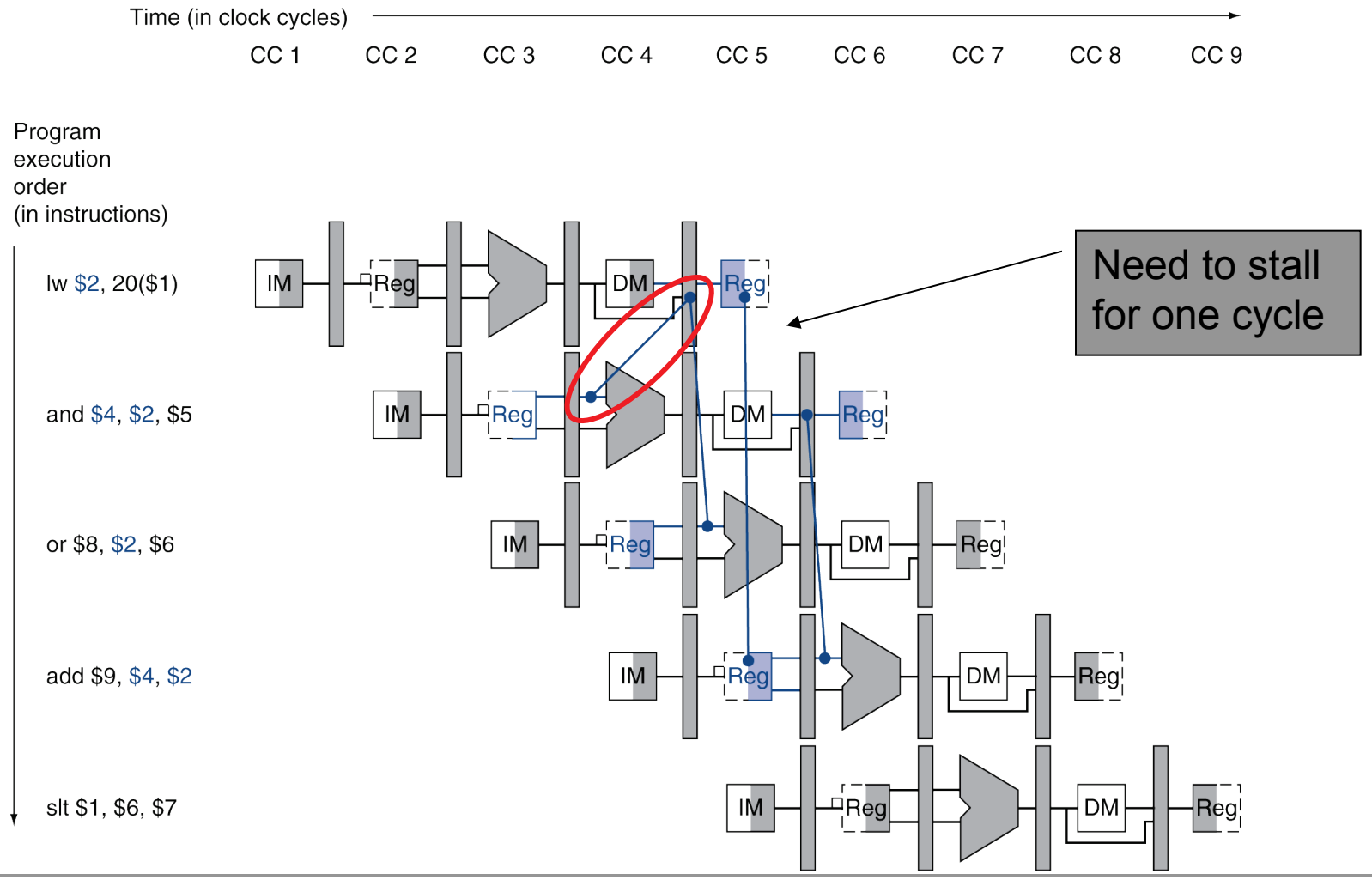
- Inserting no-ops rarely best solution
  - Complicates compiler
  - Reduces performance
- Can we solve problem in hardware? (**Hint:** when do we know value of $2?)

Dependencies & Forwarding



- Value computed at end of EX stage
- Use pipeline registers to forward
  - Add additional paths to ALU inputs from EX/MEM, MEM/WB

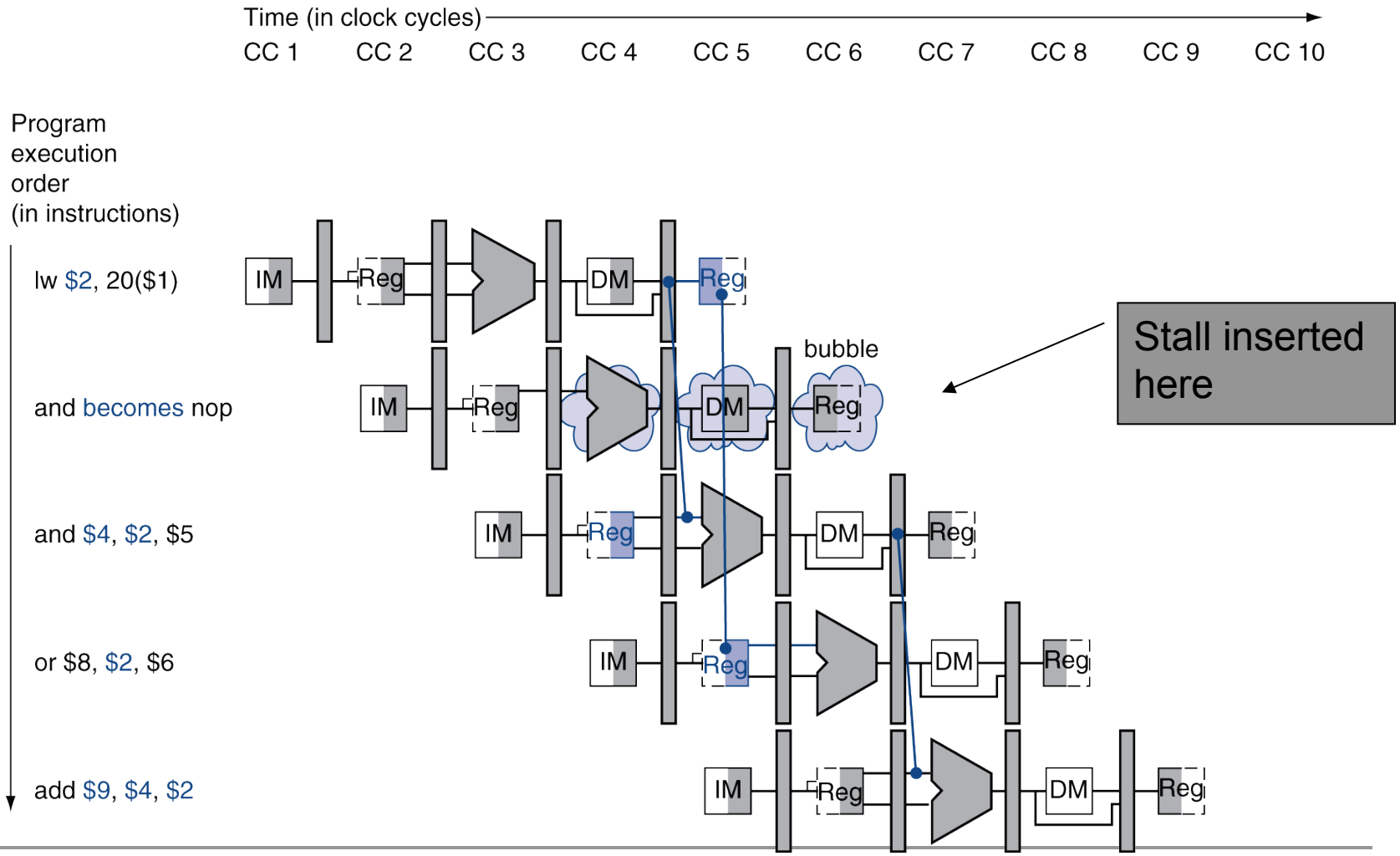
Load-Use Data Hazard



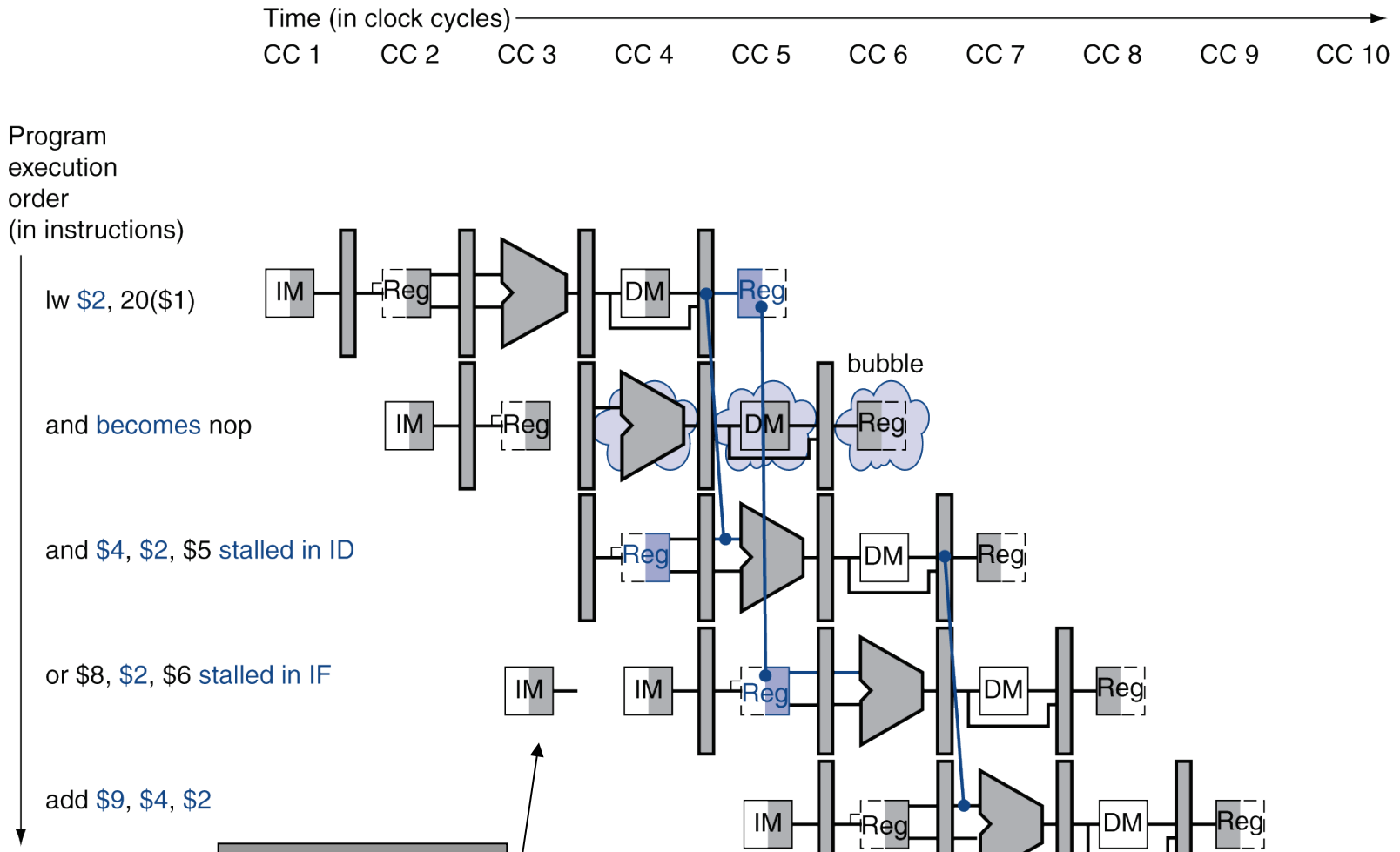
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

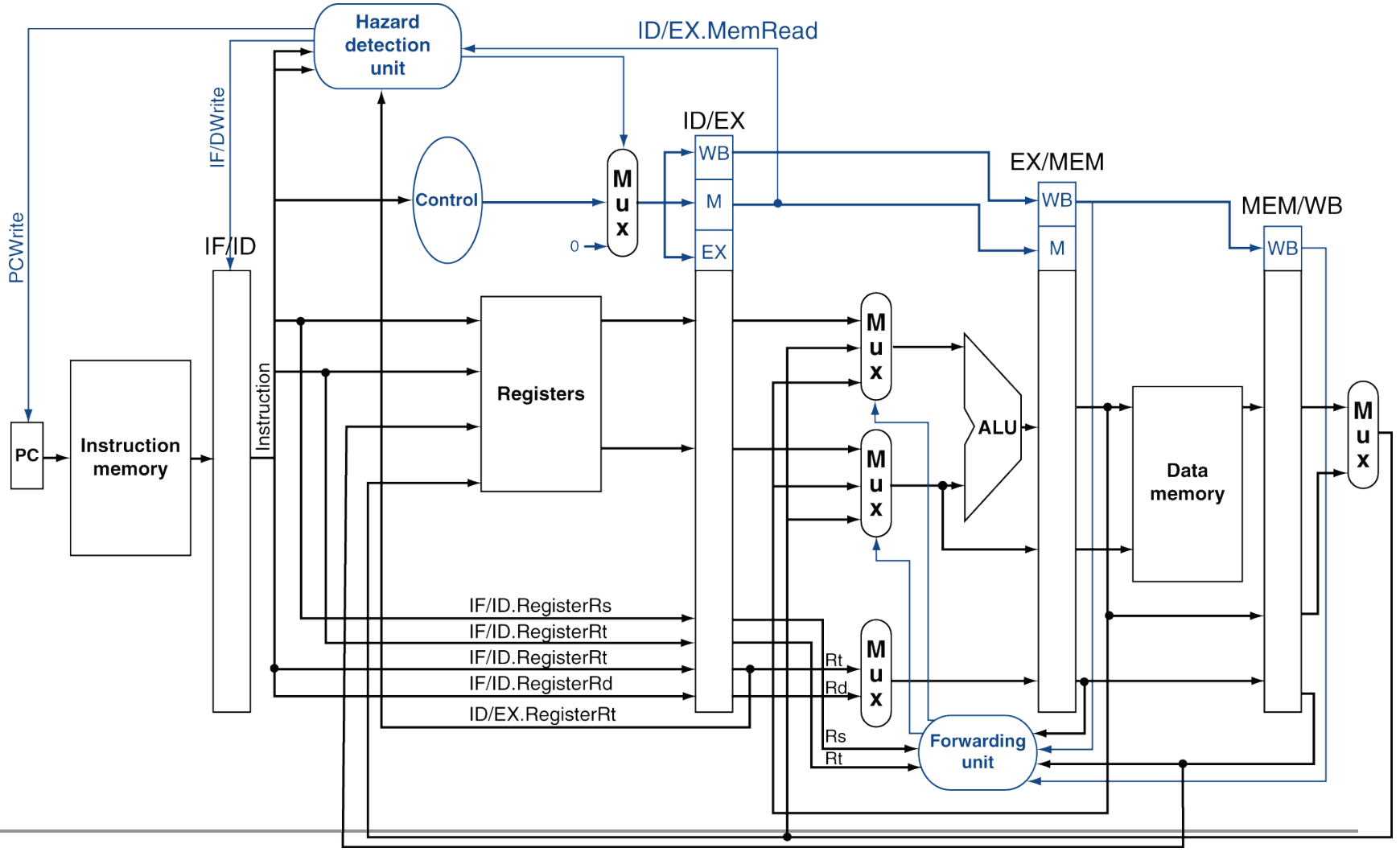


Stall/Bubble in the Pipeline



Or, more accurately…

Datapath with Hazard Detection



Performance example revisited

- Let's reevaluate the code we saw earlier:

```
loop:       add $t1, $t2, $t3
            lw  $t4, 0($t1)
            beq $t4, $t3, end
            sw  $t3, 4($t1)
            add $t2, $t2, 8
            j   loop
end:        ...
```

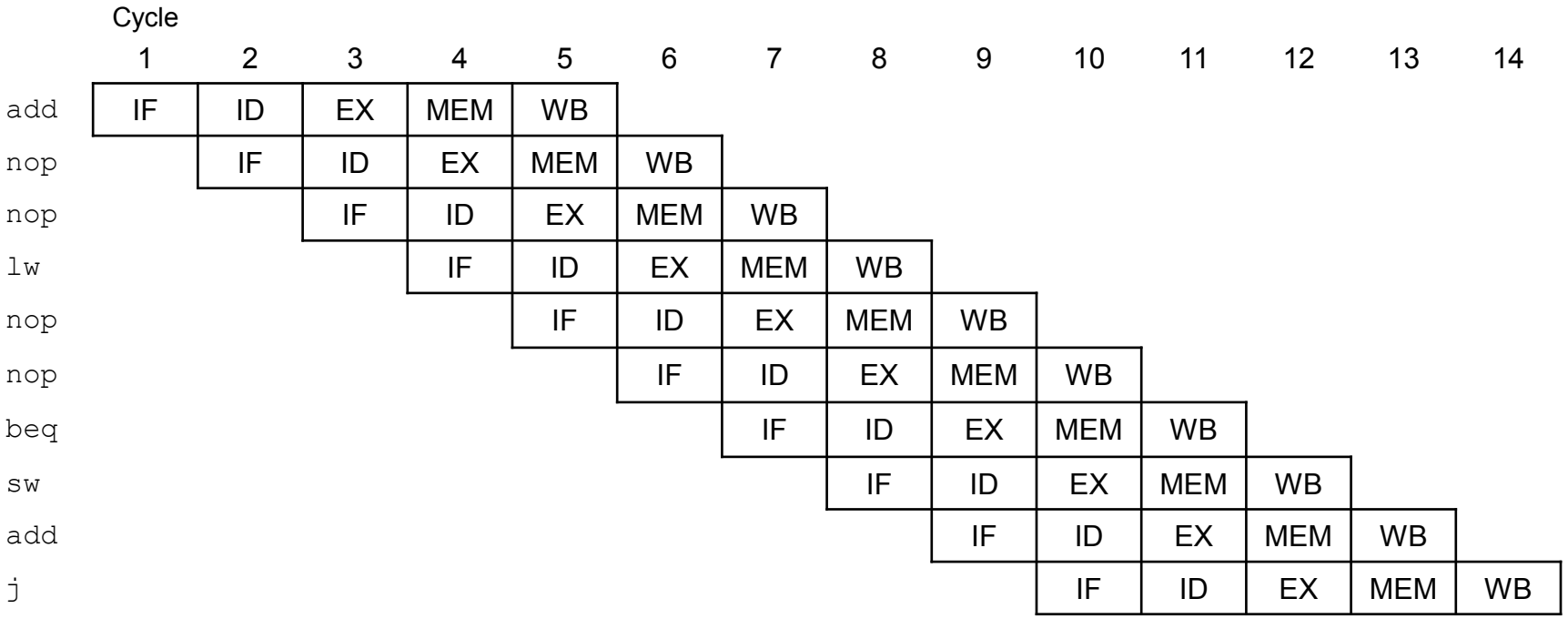
- Again, assume each pipeline stage takes 4 ns
- How long would one loop iteration take in a pipelined processor **without** forwarding?
- How long would one loop iteration take in a pipelined processor **with** forwarding?

Solution

- First, identify potential data hazards:
  - `lw` uses `$t1` generated in `add`
  - `beq` uses `$t4` generated in `lw`
  - If we had multiple loop iterations, first `add` would use `$t2` generated in second `add`

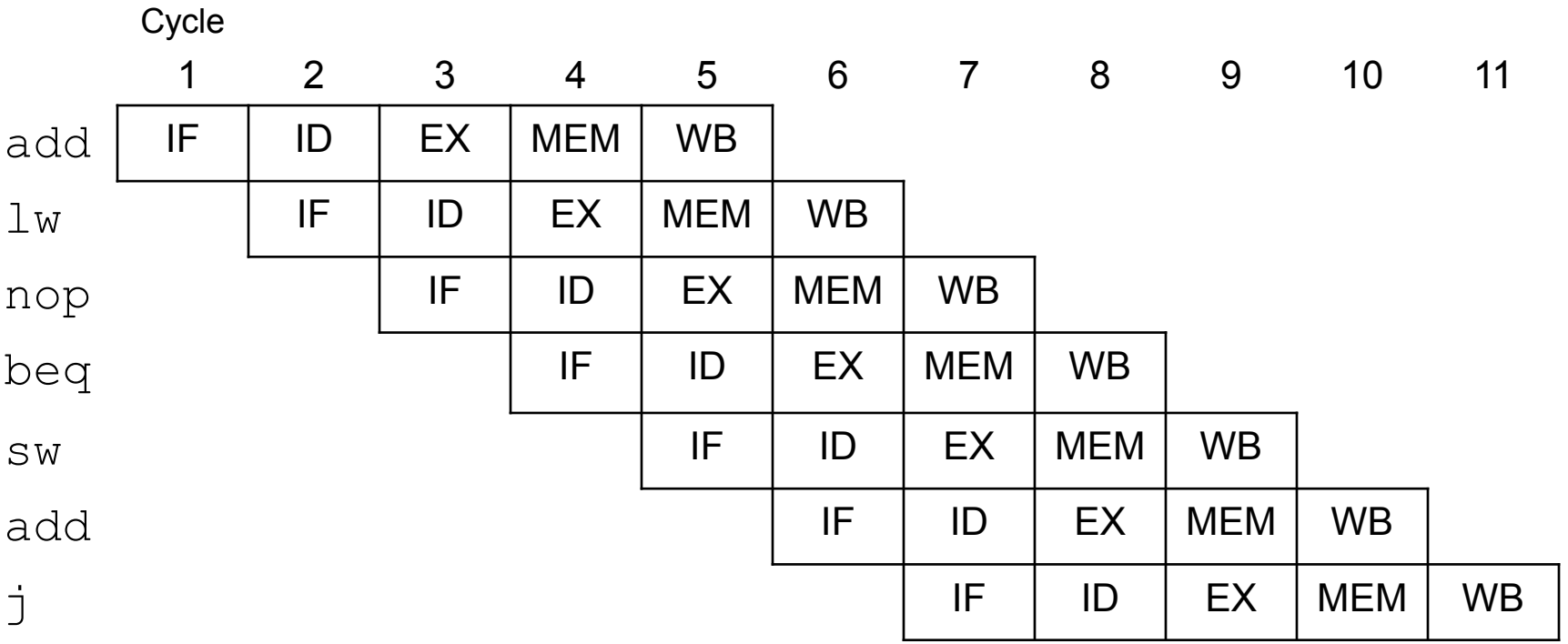
```
loop:   add $t1, $t2, $t3
        lw  $t4, 0($t1)
        beq $t4, $t3, end
        sw  $t3, 4($t1)
        add $t2, $t2, 8
        j   loop
end:    ...
```

Solution: no forwarding



- Need 2 cycles between dependent instructions in 5 stage pipeline without forwarding
- Diagram shows code takes 14 cycles
  - (14 cycles) * (4 ns/cycle) = 64 ns

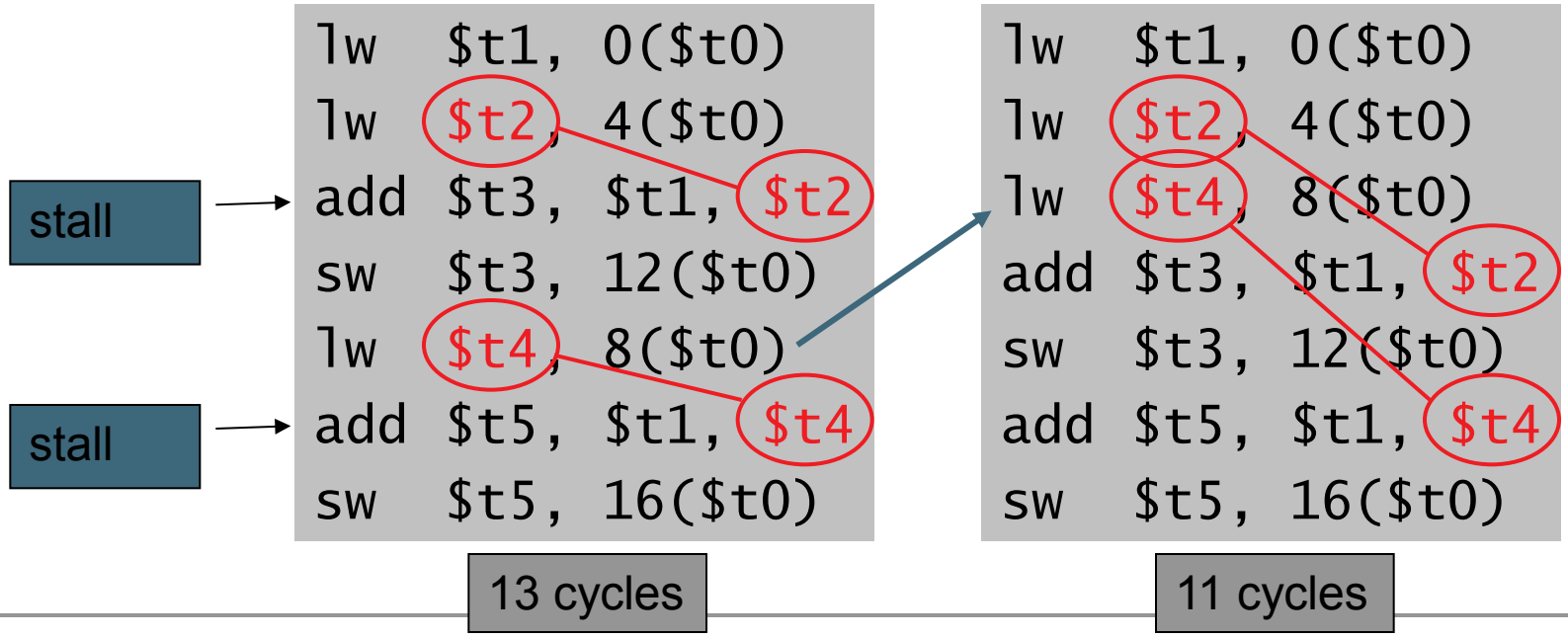
Solution: forwarding



- Forwarding resolves add → lw hazard
- Need 1 cycle to handle lw → beq hazard
- Total time: 11 cycles
  - (11 cycles) * (4 ns/cycle) = 44 ns

Code Scheduling to Avoid Stalls

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

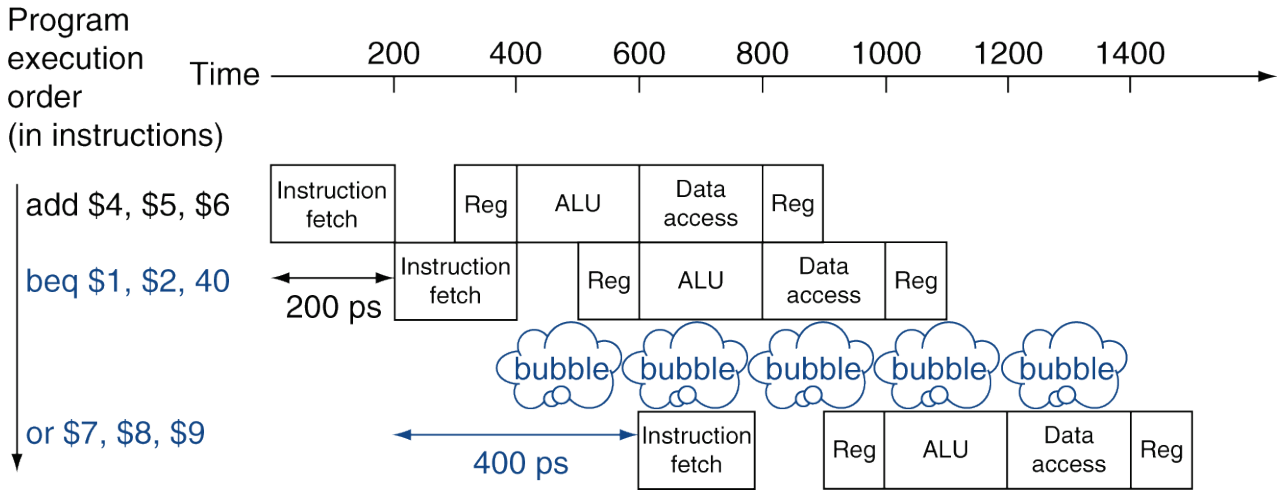


Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

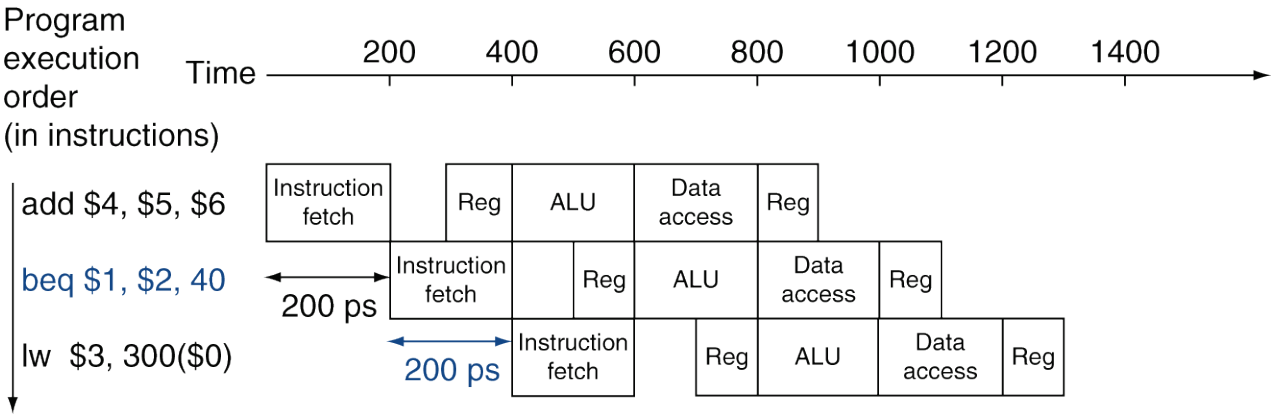


Branch Prediction

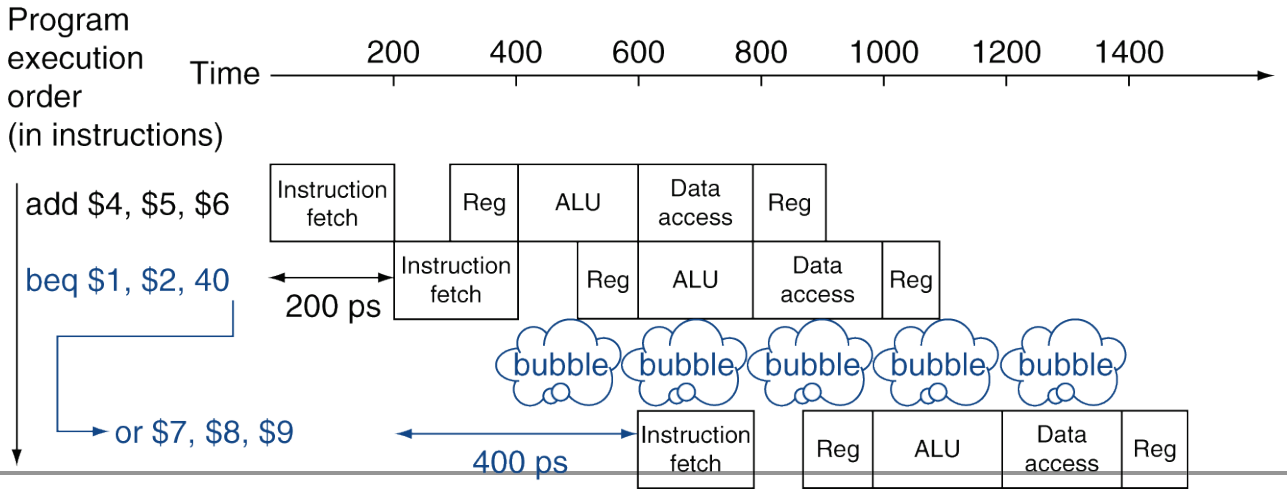
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Final notes

- Next time:
  - Instruction scheduling issues
    - Dynamic branch prediction
    - Dynamic scheduling
    - Multiple issue
- Announcements/reminders
  - HW 1 due 2/7
  - HW 2 to be posted; due 2/13