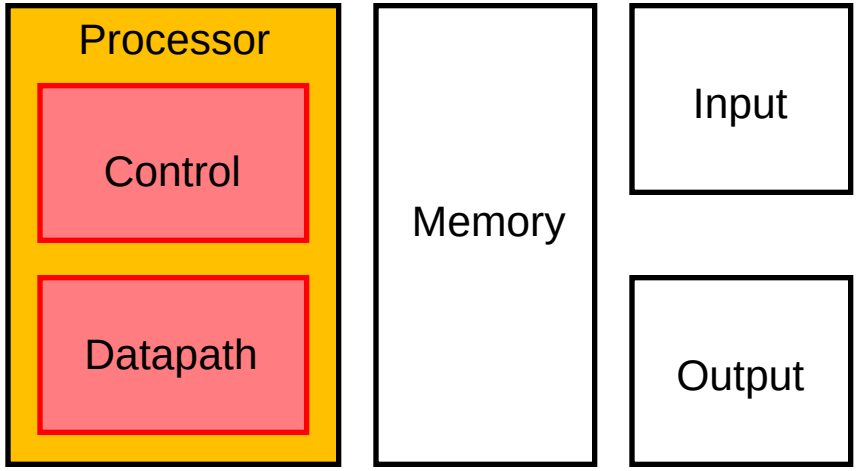
Chapter 4

The Processor

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



- Designing the Single Cycle Processor: Data Path + Control

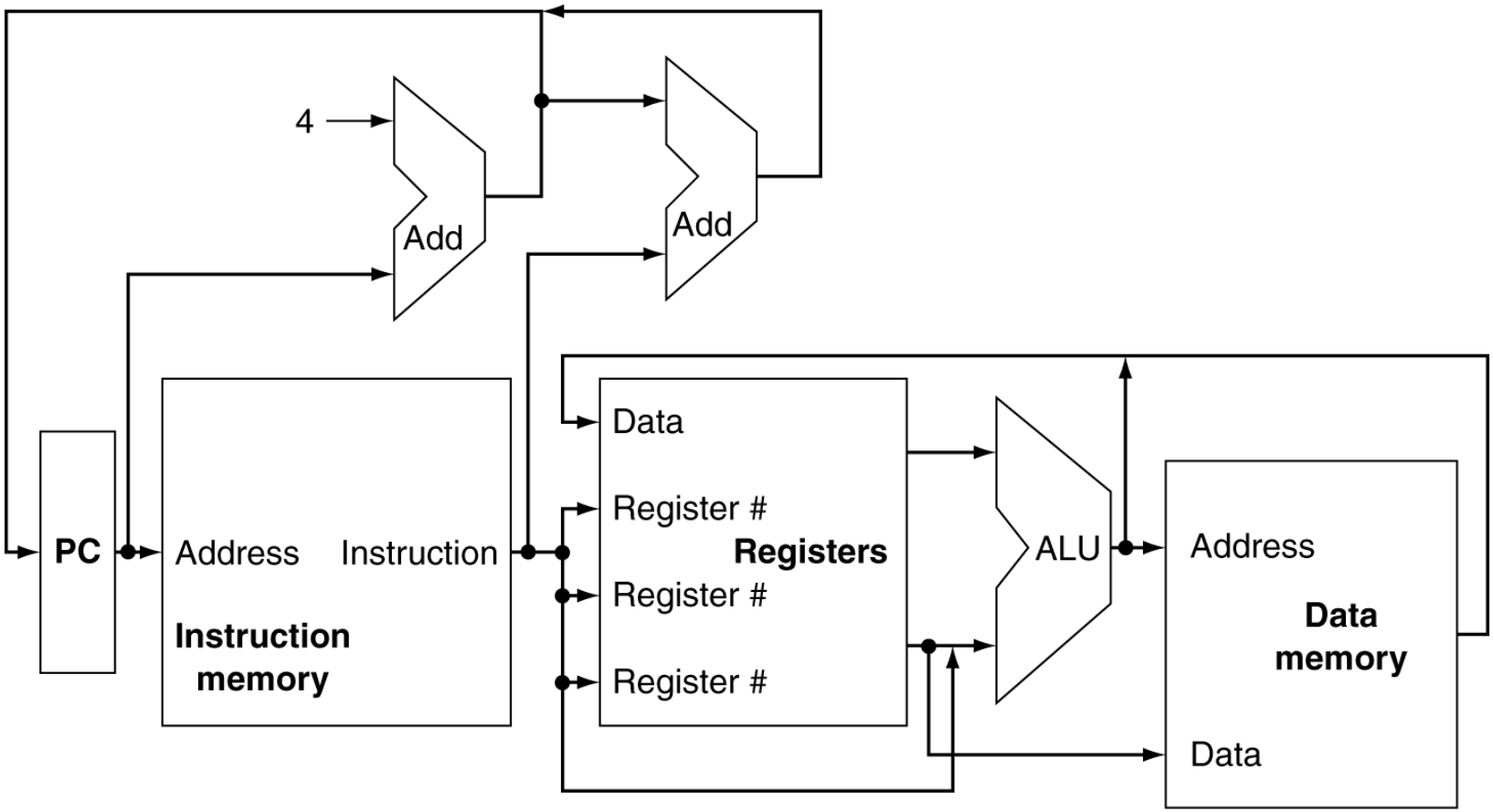
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Simple subset of ISA, shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `andi`, `ori`, `slt`
  - Control transfer: `beq`, `j`

Instruction Execution

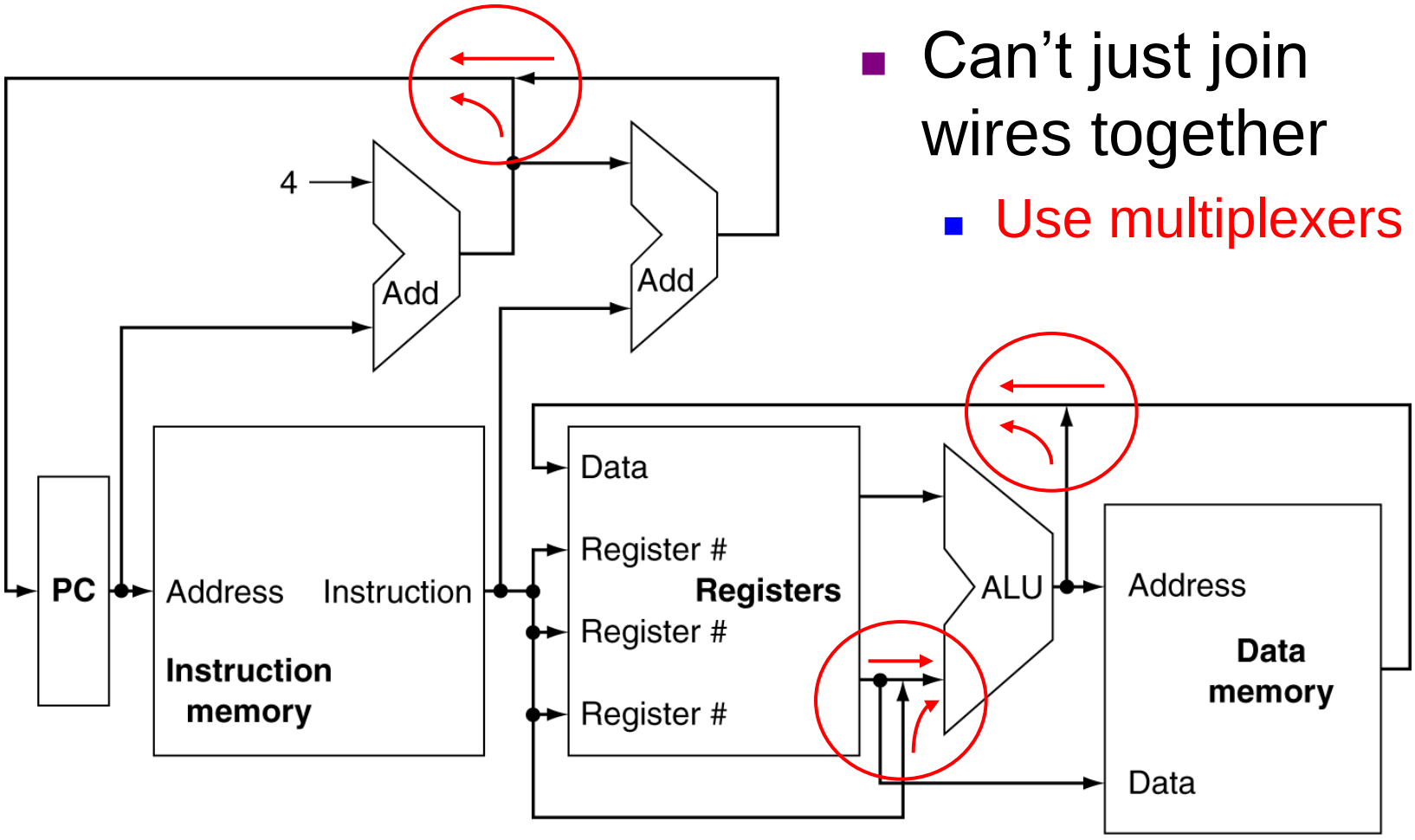
- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC $\leftarrow$ branch target address or PC + 4

CPU Overview – basic data path

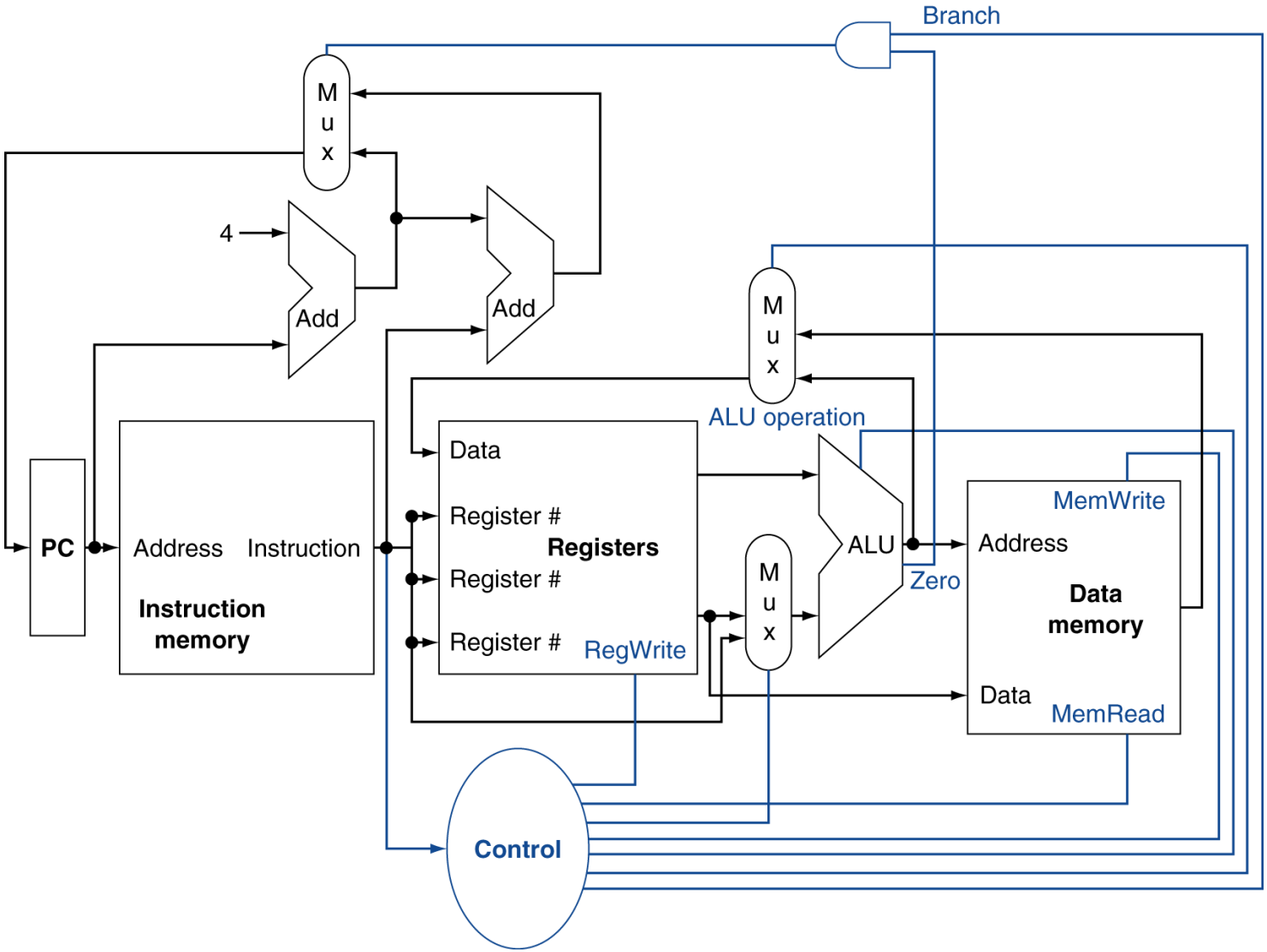


Multiplexers

- Can't just join wires together
  - Use multiplexers



Control – blue lines

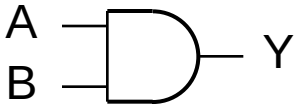


Logic Design Basics

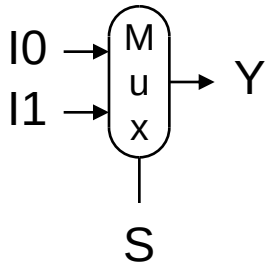
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

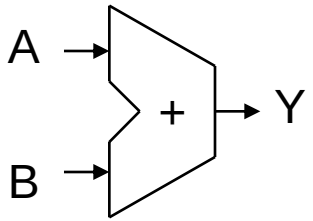
- AND-gate
  - Y = A & B



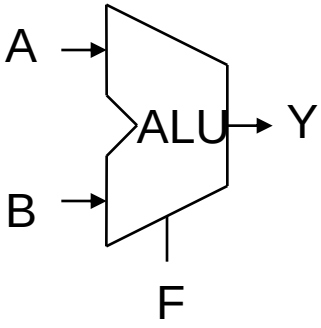
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

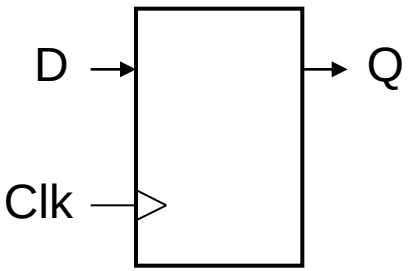


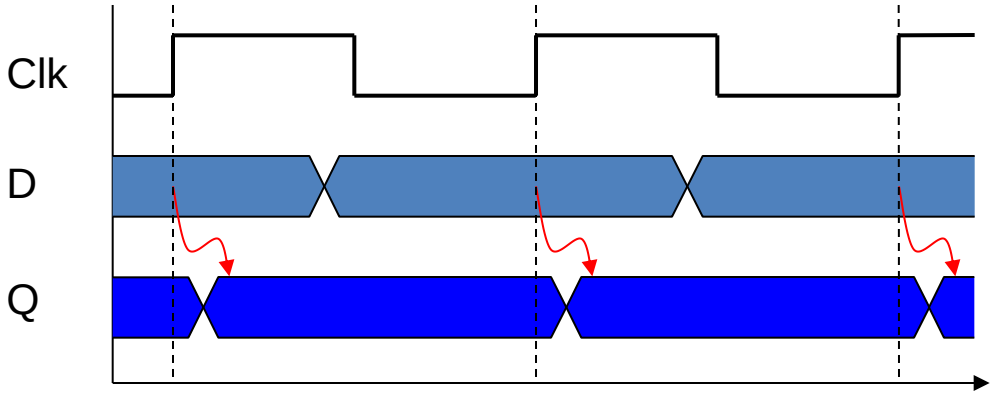
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

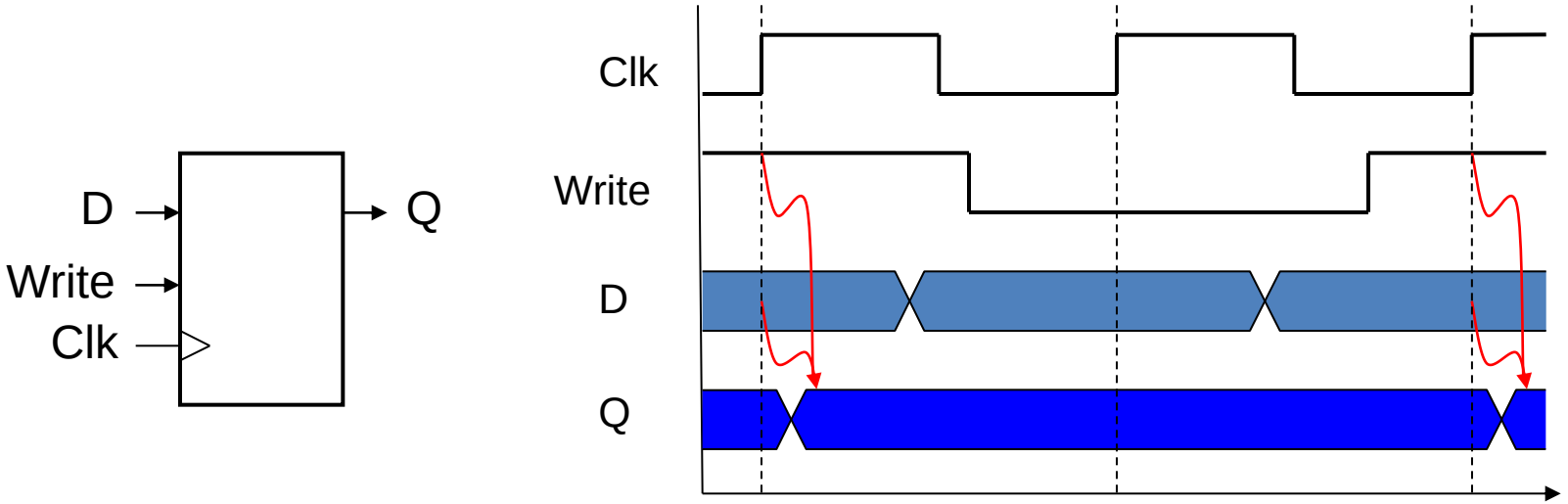
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1





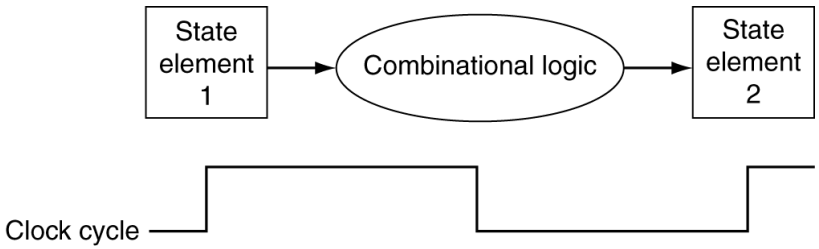
Sequential Elements

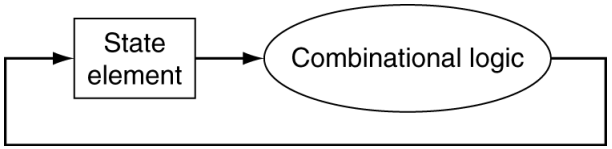
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later



Clocking Methodology

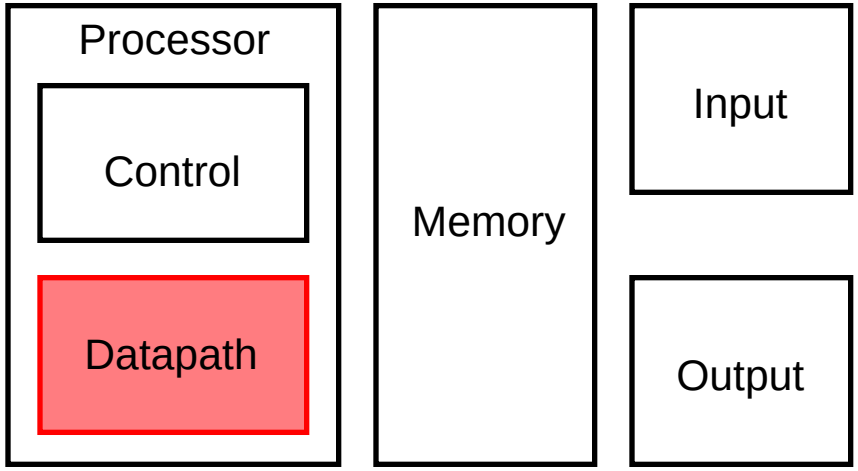
- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period





The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

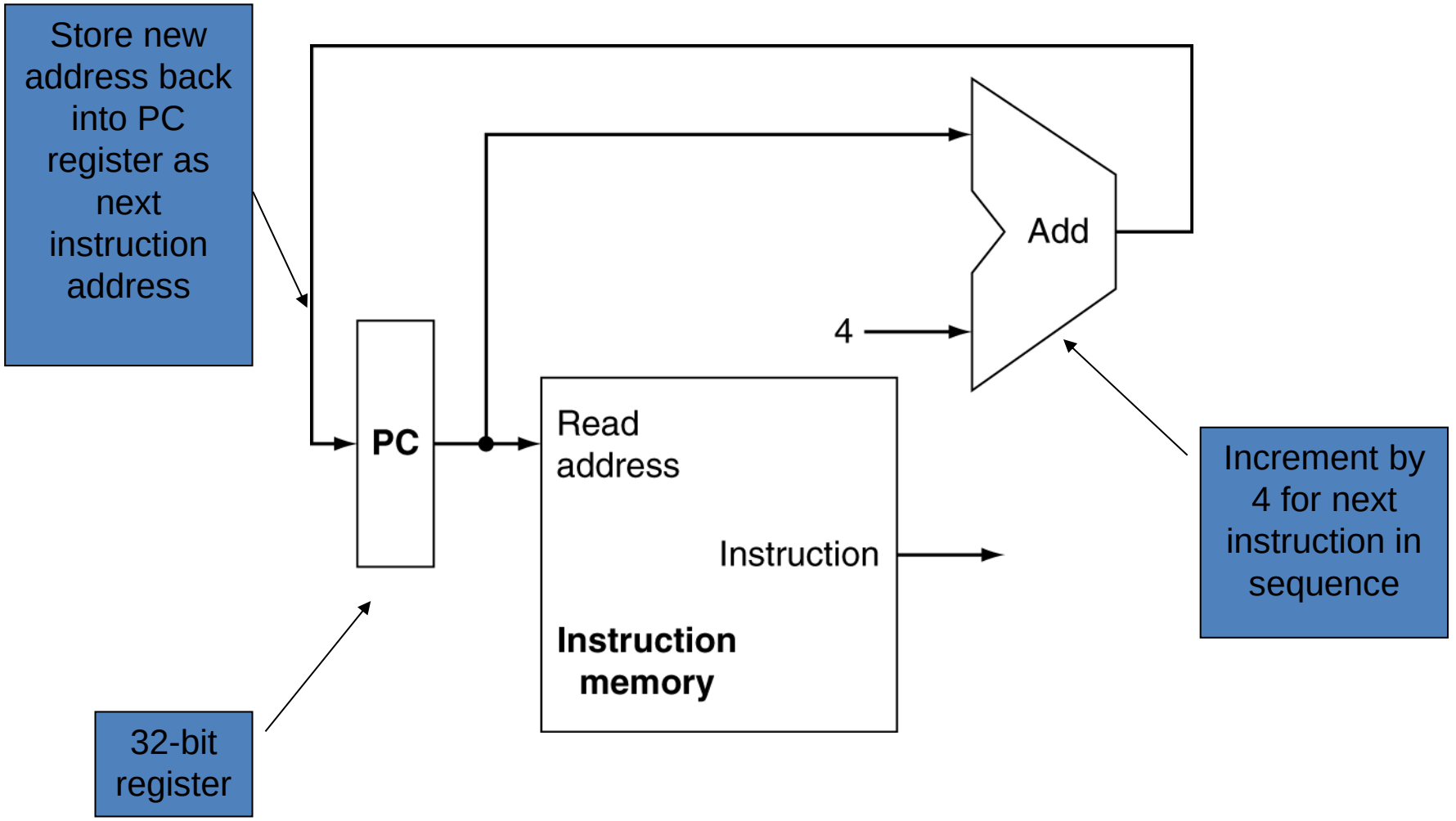


- Designing the Single Cycle Datapath

Building a Datapath

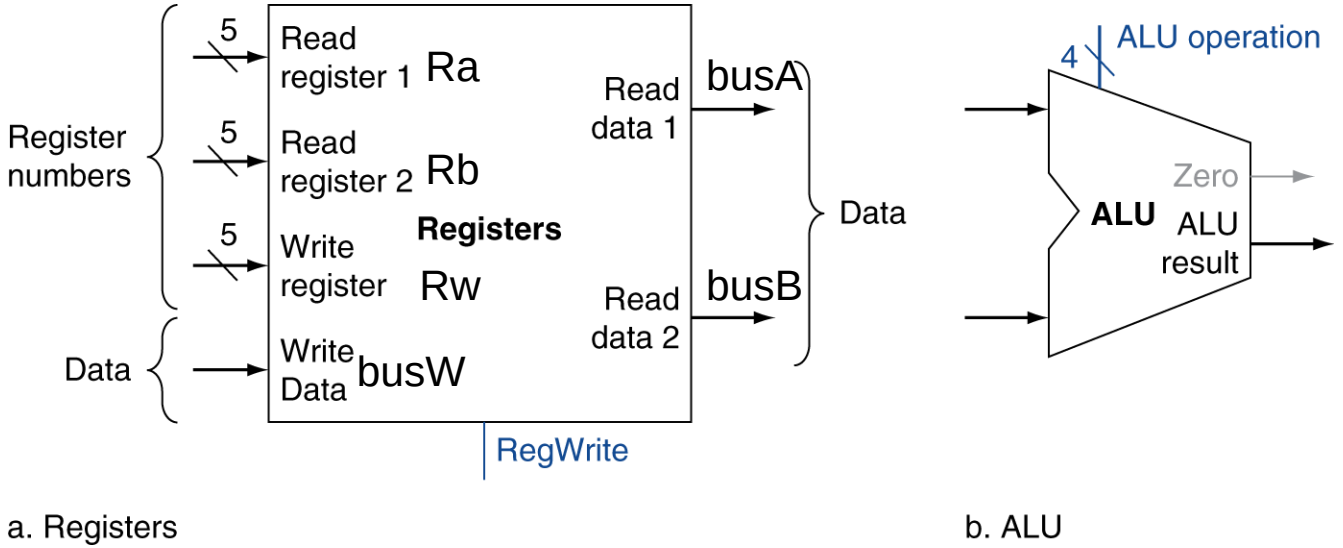
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a MIPS datapath incrementally
  - Refining the overview design

Instruction Fetch



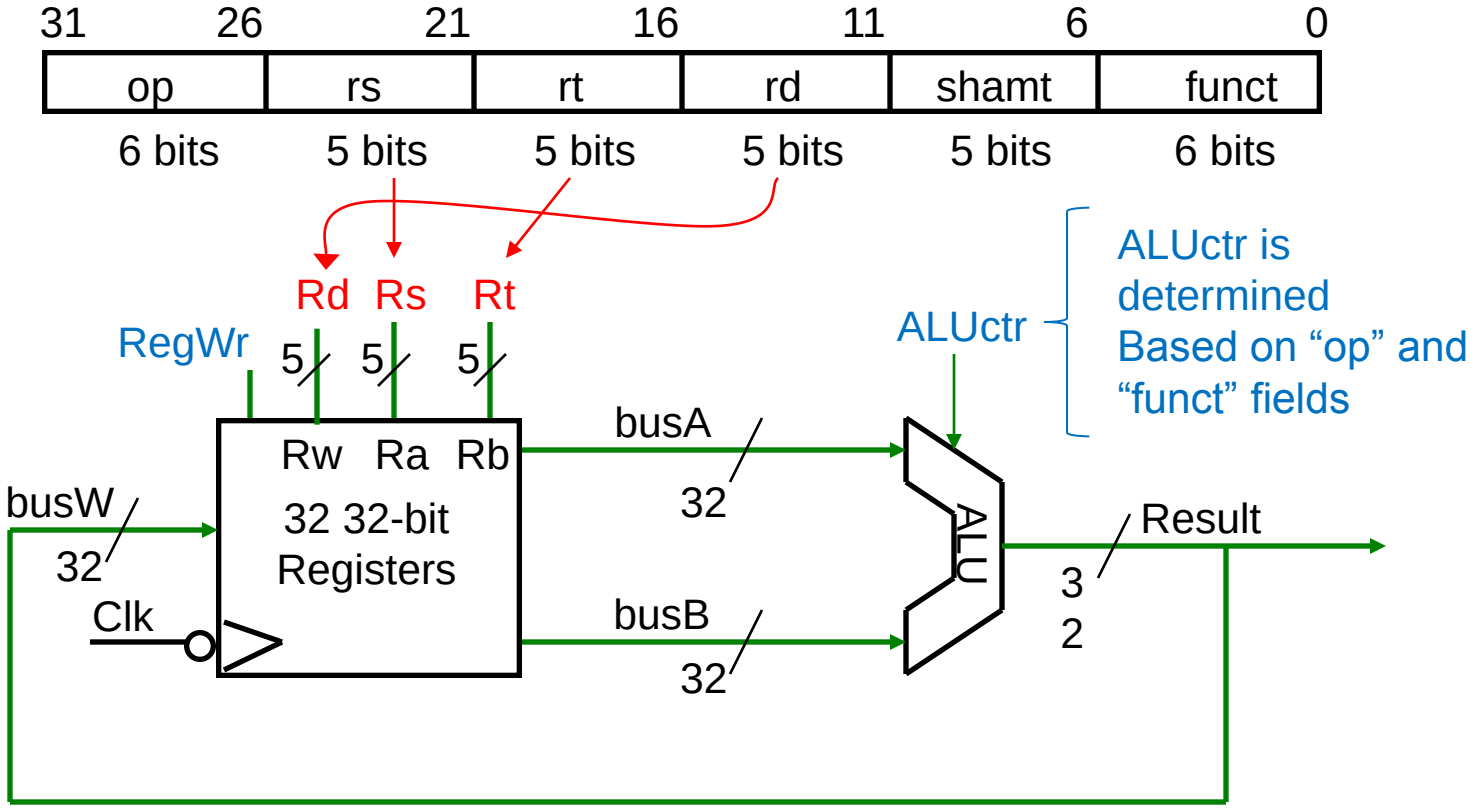
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result

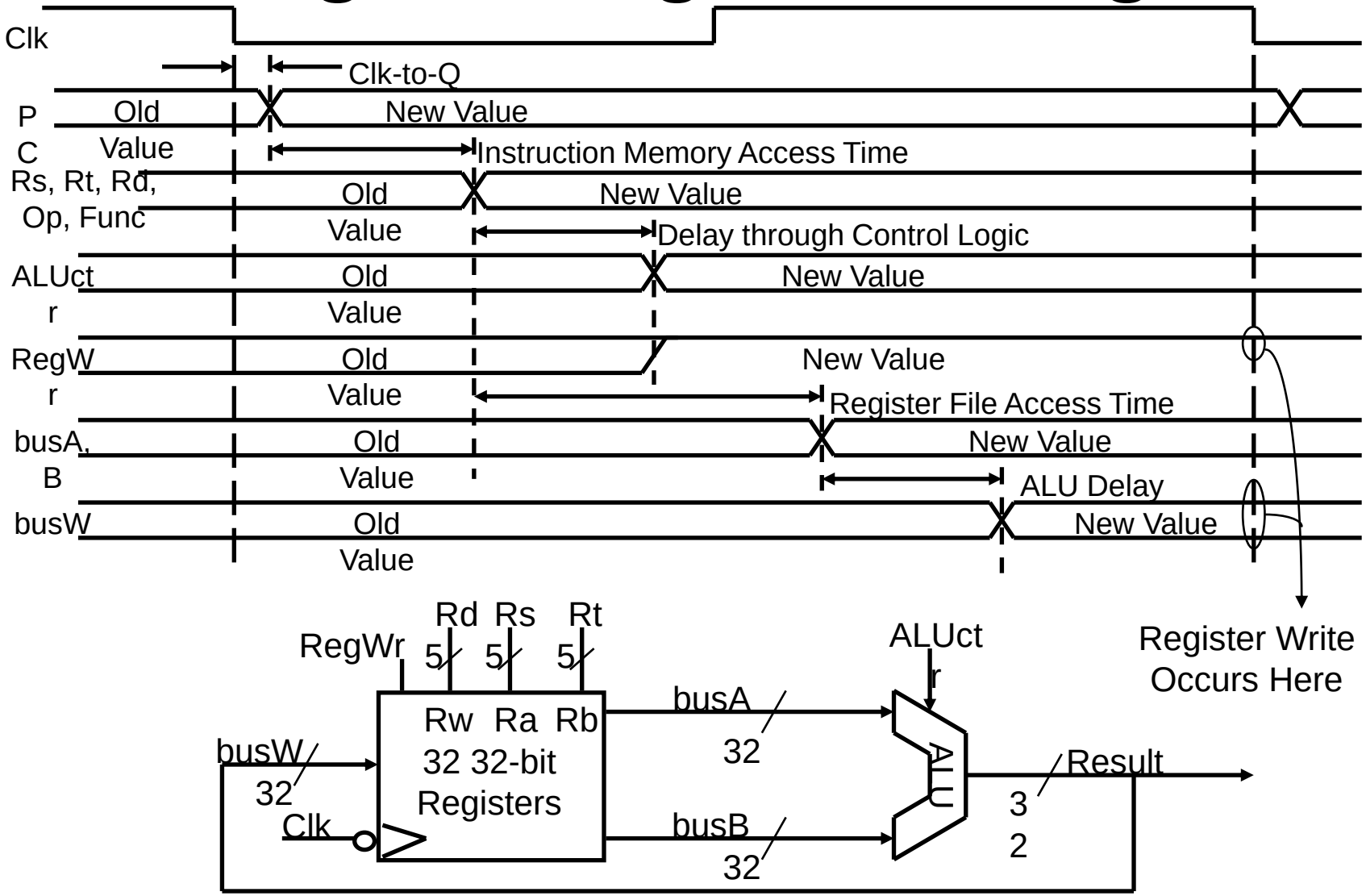


Data path for R-type Instructions

- Example: addU rd, rs, rt
  - ALUctr and RegWr: control logic after decoding the instruction (we will look at the design of control logic later)

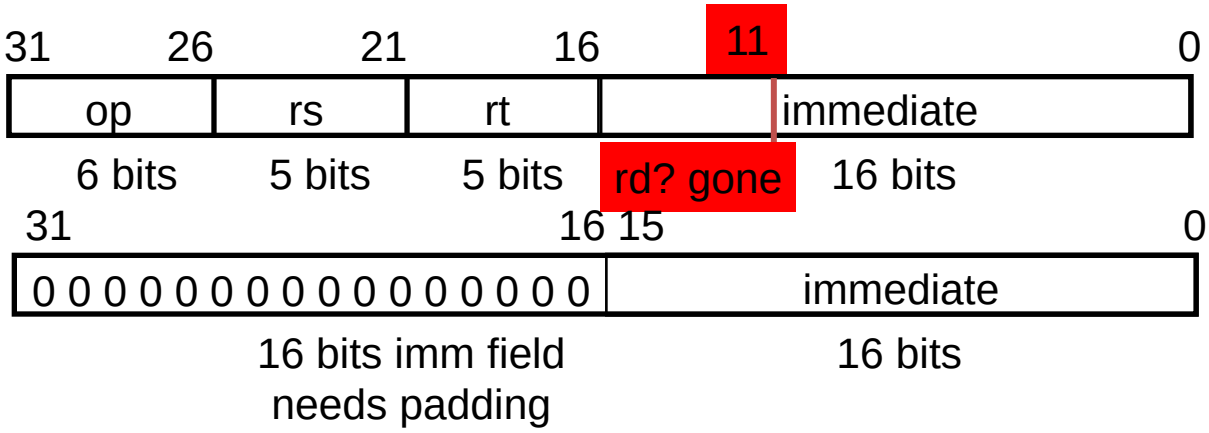


Register-Register Timing



Logical Operations with Immediate (path for I-type)

Some modifications needed to existing paths and logic



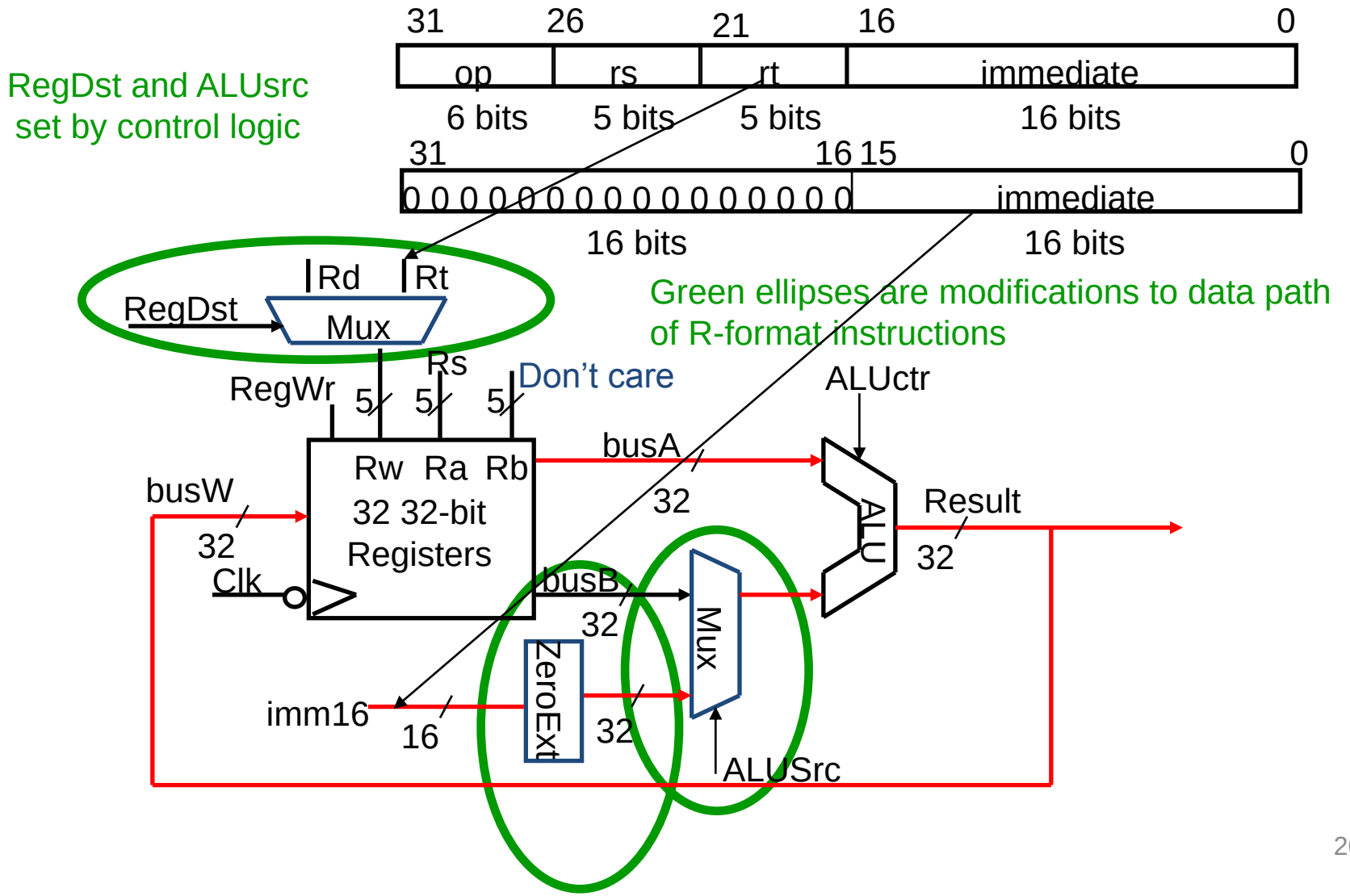
- Two issues:
  - Instead of rd field in R-format, we have rt field in I-format instructions as the destination register address.

    ➔ we need a way to select between rt and rd fields as the destination address.
  - Second argument of the ALU comes from immediate field of the instruction instead of busB.

    ➔ we need a way to select between the immediate field and the busB to feed in to the second input of ALU.

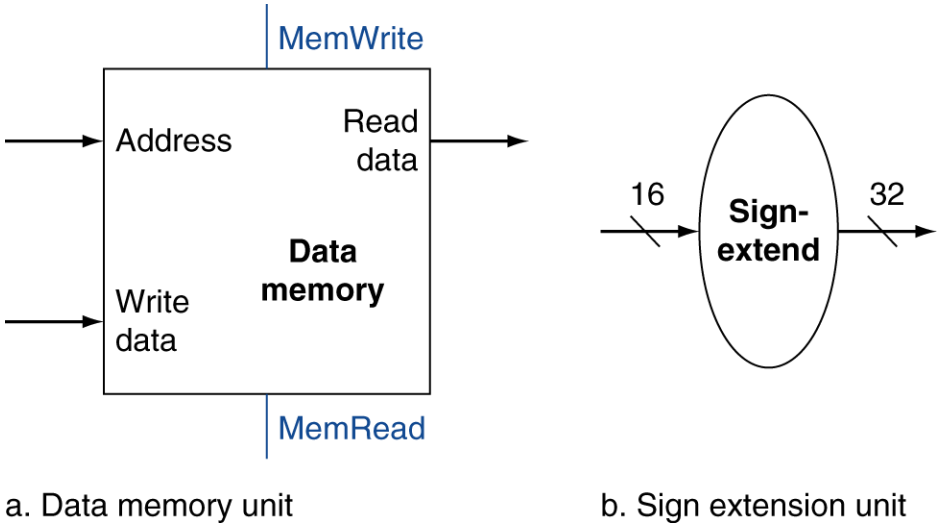
Logical Operations with Immediate

Example: ori rs, rt, 0xffff



Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend immediate field instead of zero-extend
- Load: Read memory and update register
- Store: Write register value to memory

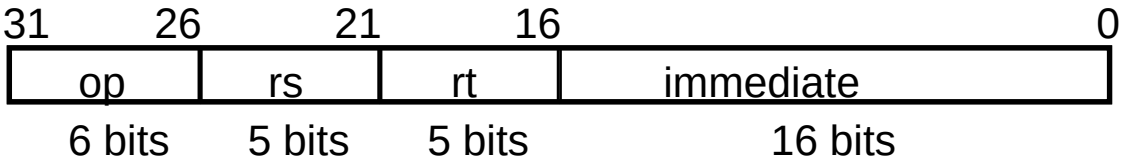


a. Data memory unit

b. Sign extension unit

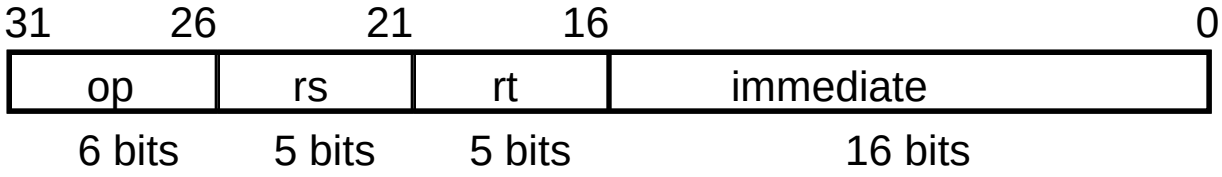
Load Operations

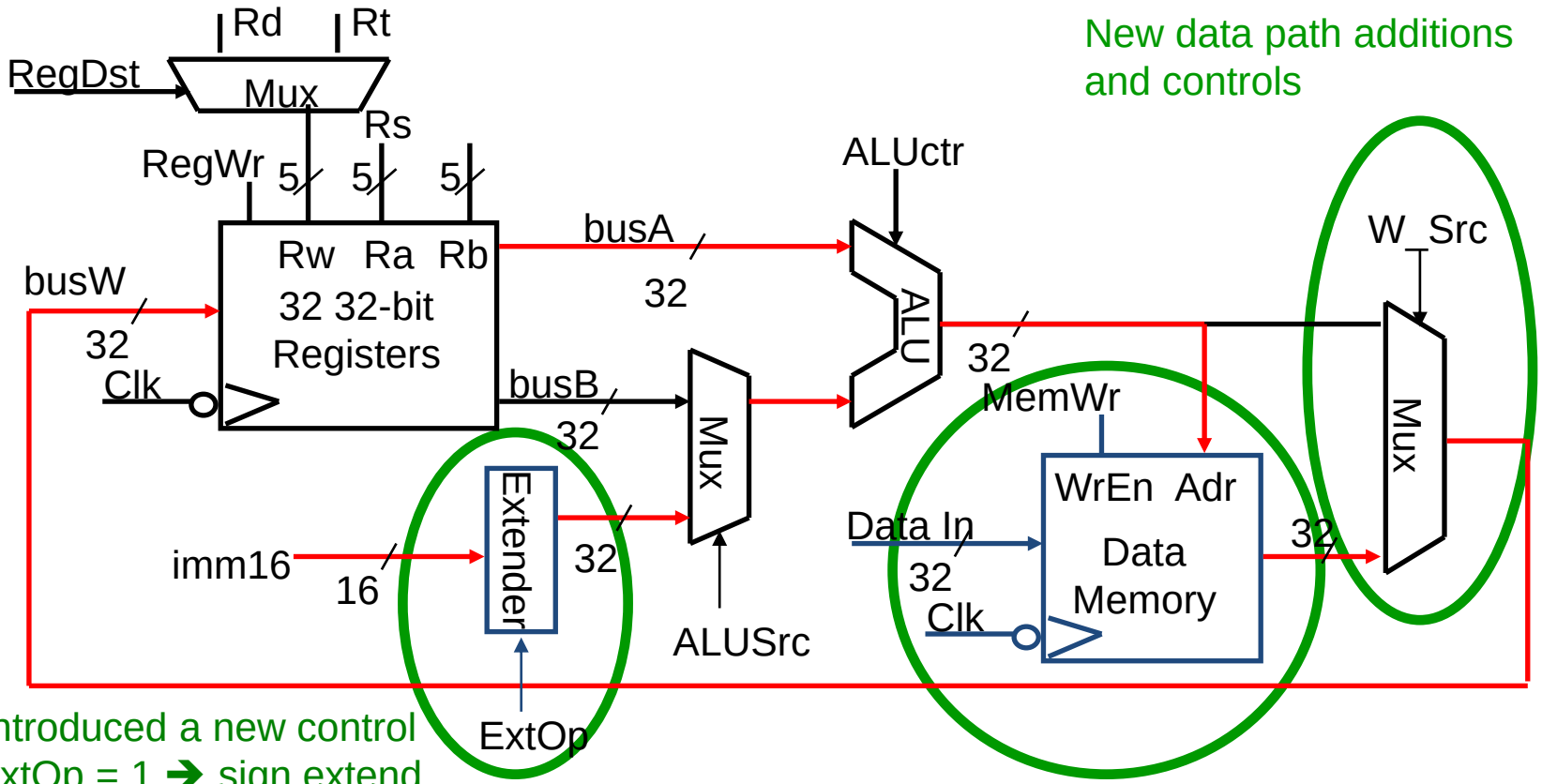
- Example: lw rt, rs, imm16



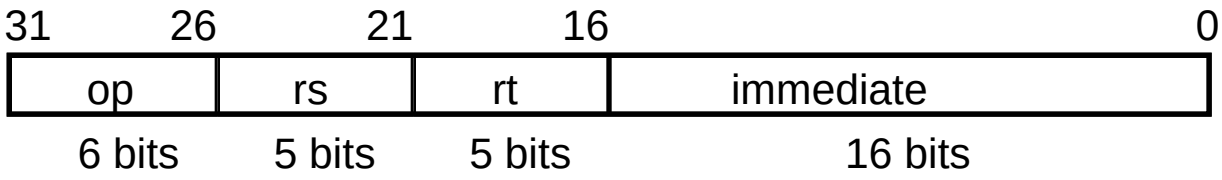
- Cannot use instruction's Rd field for register file's Rw input
  - Because lw is an I-type instruction and there is no Rd field.
  - Instead Rt field is used for destination
- First input of ALU comes from Rs field of instruction and second input comes from immediate field
- Extender that can do sign extension is used for extending the immediate field. The immediate field for address computation is a signed number.
- ALU output goes to address input of data memory
- Data to be written in register file can come from ALU result or data read from memory → we need a MUX to select one.

Load Operations





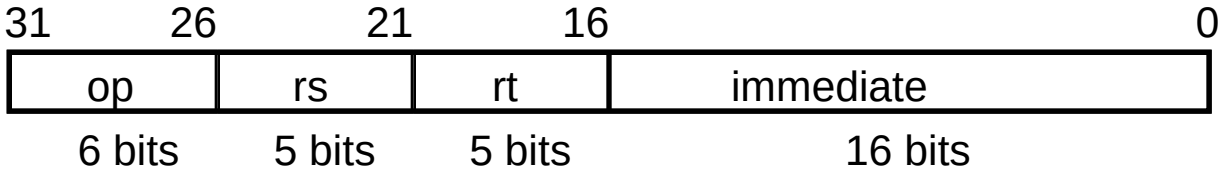
Store Operations

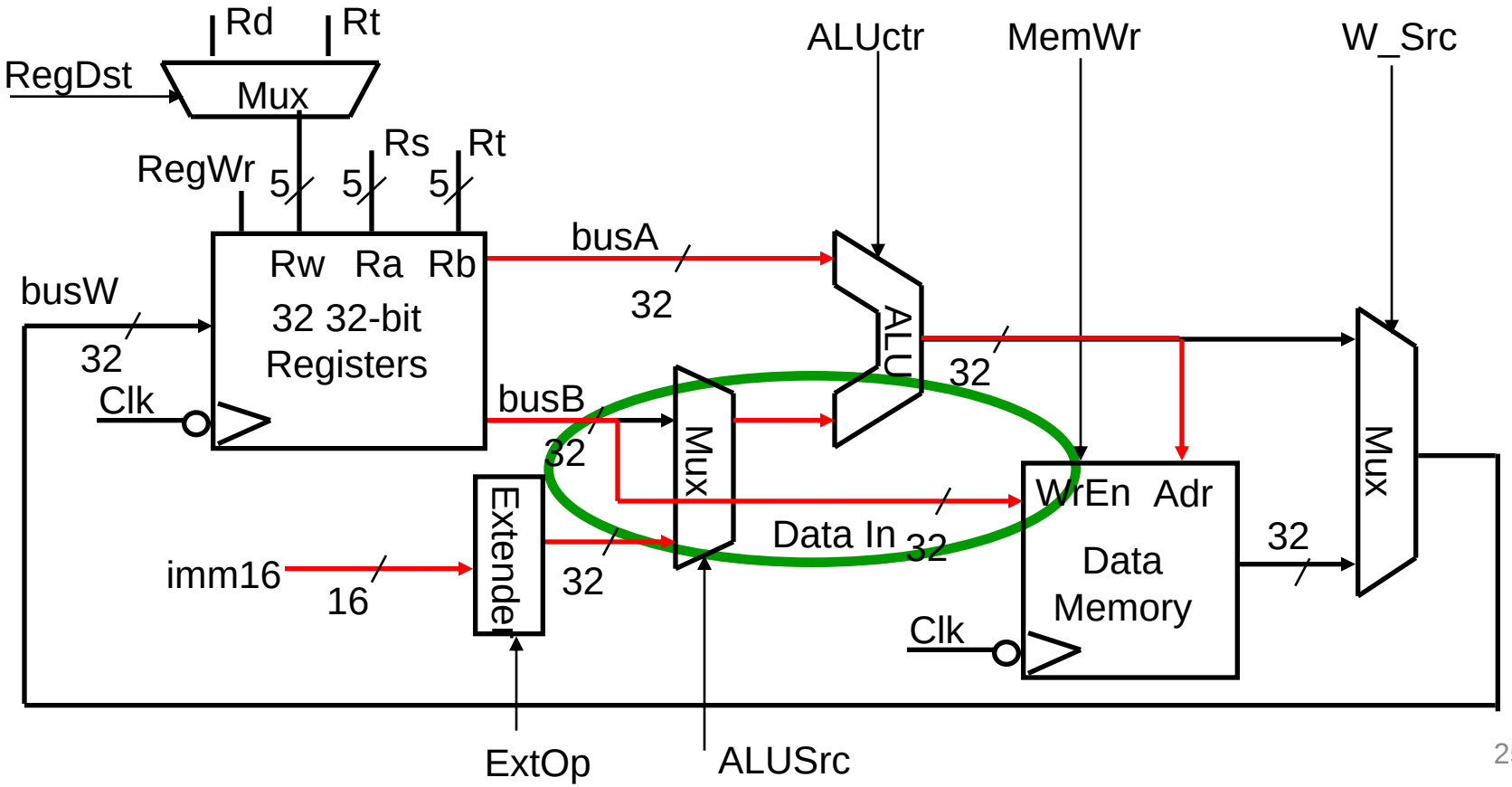


- Register file, ALU and extender are the same as for lw instruction
- New datapath that takes BusB from register file and connects it to DataIn of data memory.
- Sw is the first instruction that does not write a result into the register file (it writes a result in memory).

  → control unit must set RegWr to 0 while executing sw instruction.
- Also, MemWr must be asserted by the control unit.

Store Operations

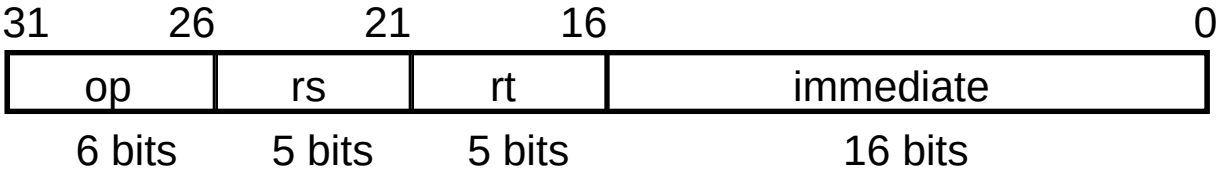




Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (i.e., multiply by 4)
    - (immediate in # of words)
  - Add to PC + 4
    - Already calculated at the instruction fetch stage

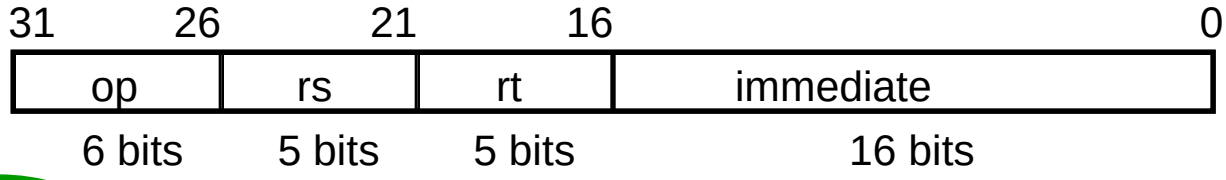
The Branch Instruction

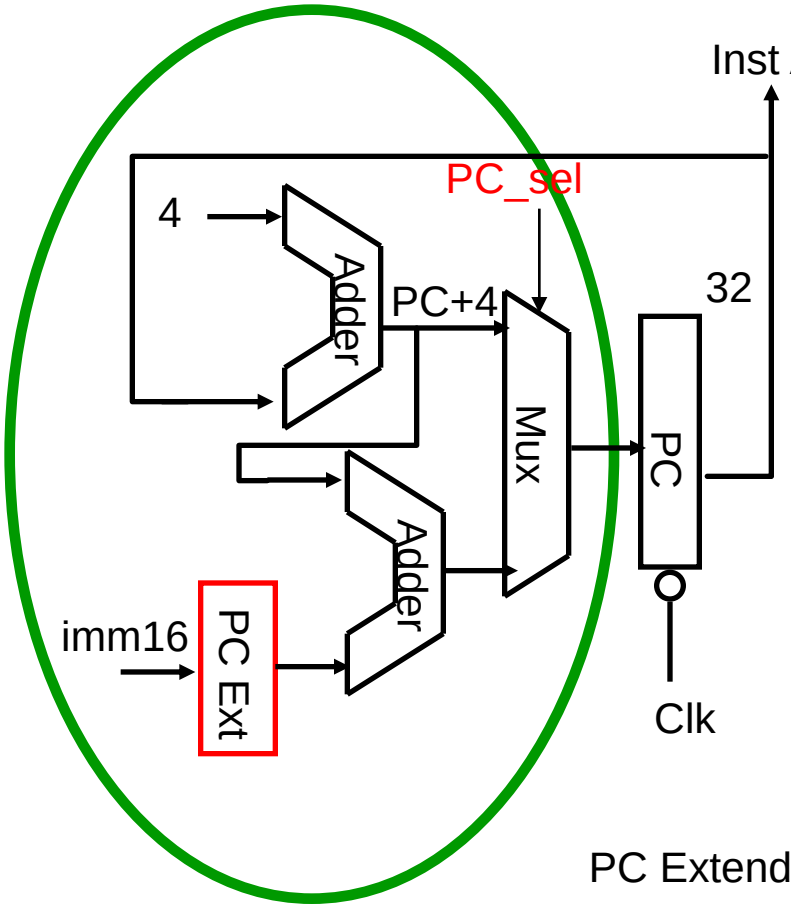


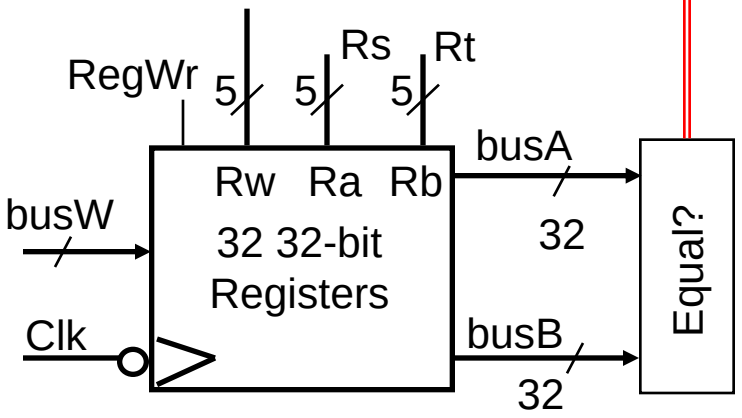
- beq rs, rt, imm16
  - mem[PC] — Fetch the instruction from memory
  - COND ← (R[rs] == R[rt]) — Calculate the branch condition
  - if (COND == 1) — Calculate the next instruction's address
    - PC ← PC + 4 + (SignExt(imm16) * 4)
  - else
    - PC ← PC + 4

PC extend

Datapath for Branch Operations







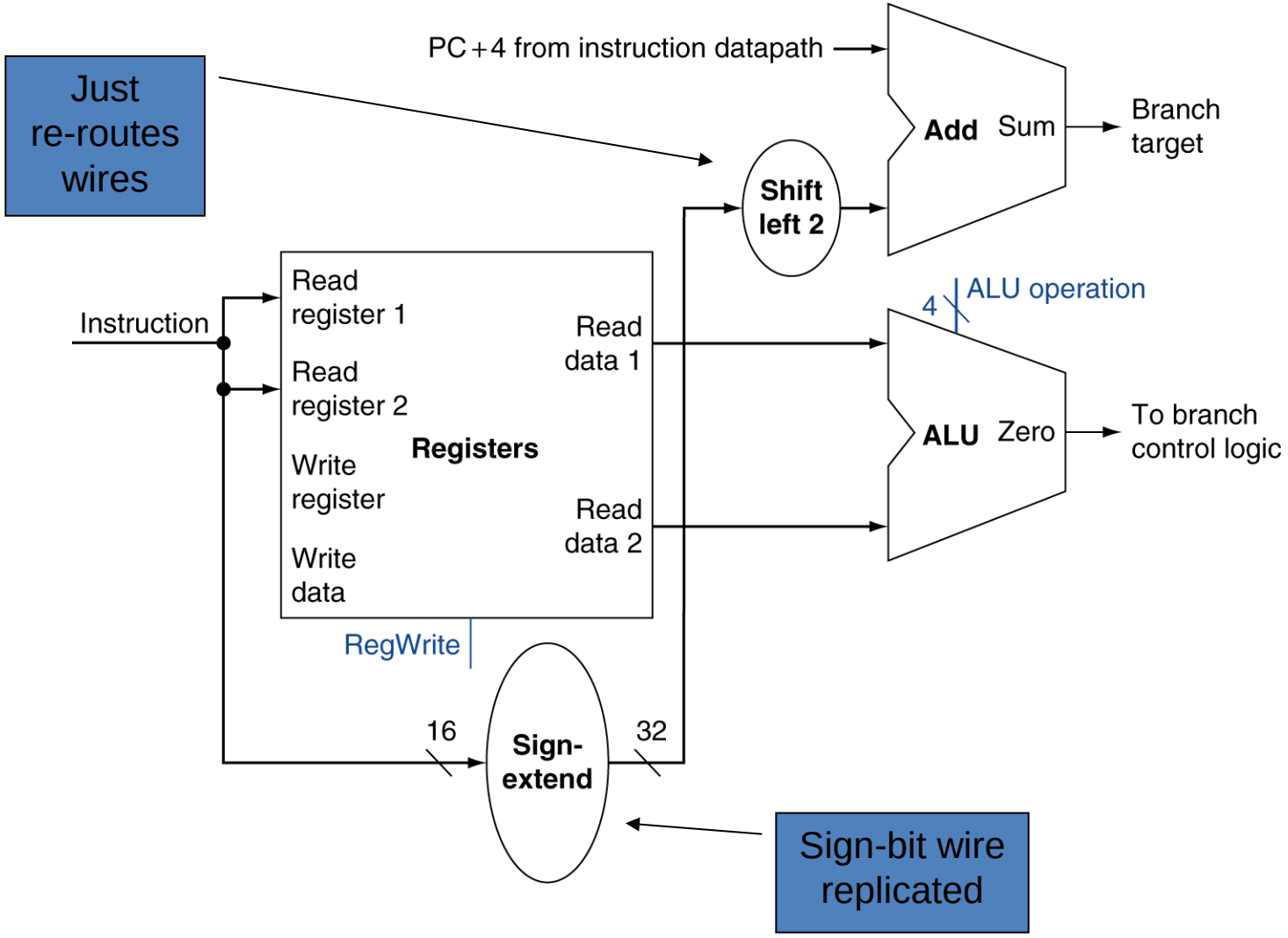
Cond is the zero test output of ALU that we designed.

Cond

Equal = zero after (R[Rs]-R[Rt])

PC Extend=sign extend and mult by 4 (or shift left 2 bits)

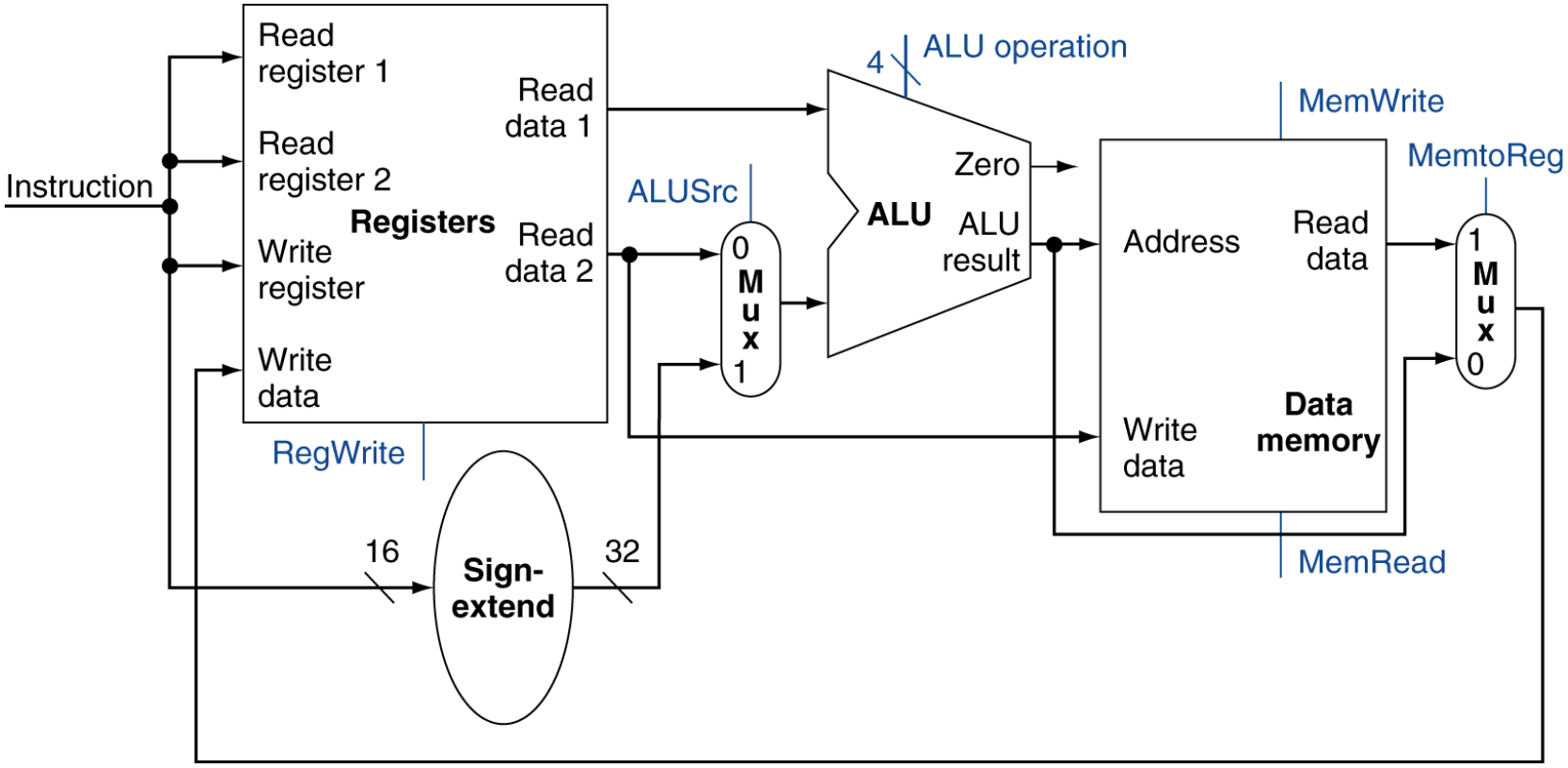
Branch Instructions



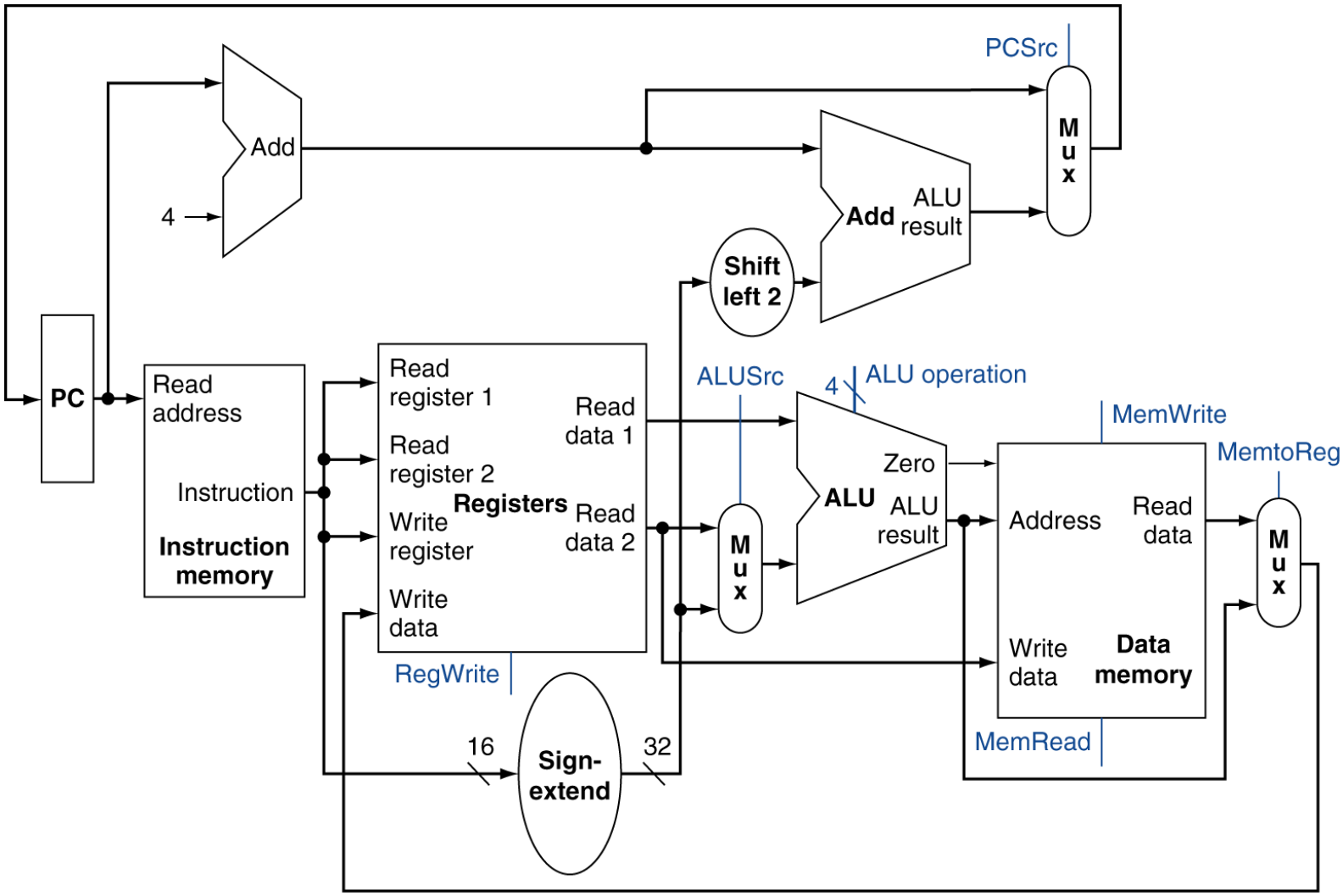
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath

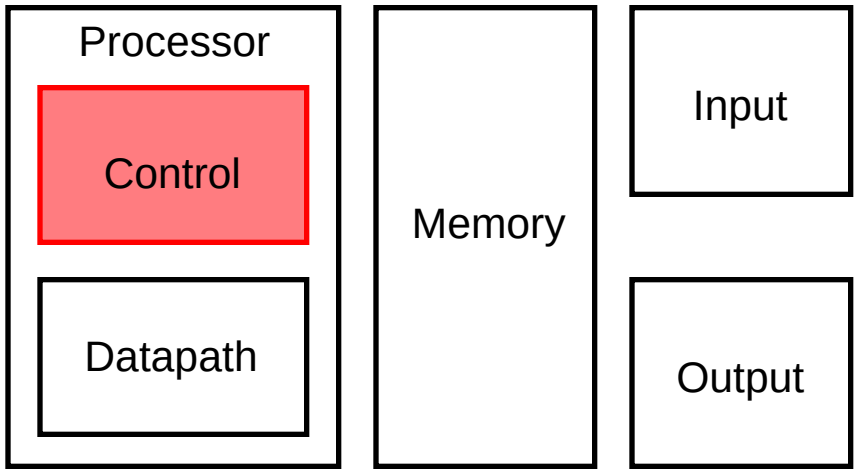


Full Datapath



The Big Picture: Where are We Now?

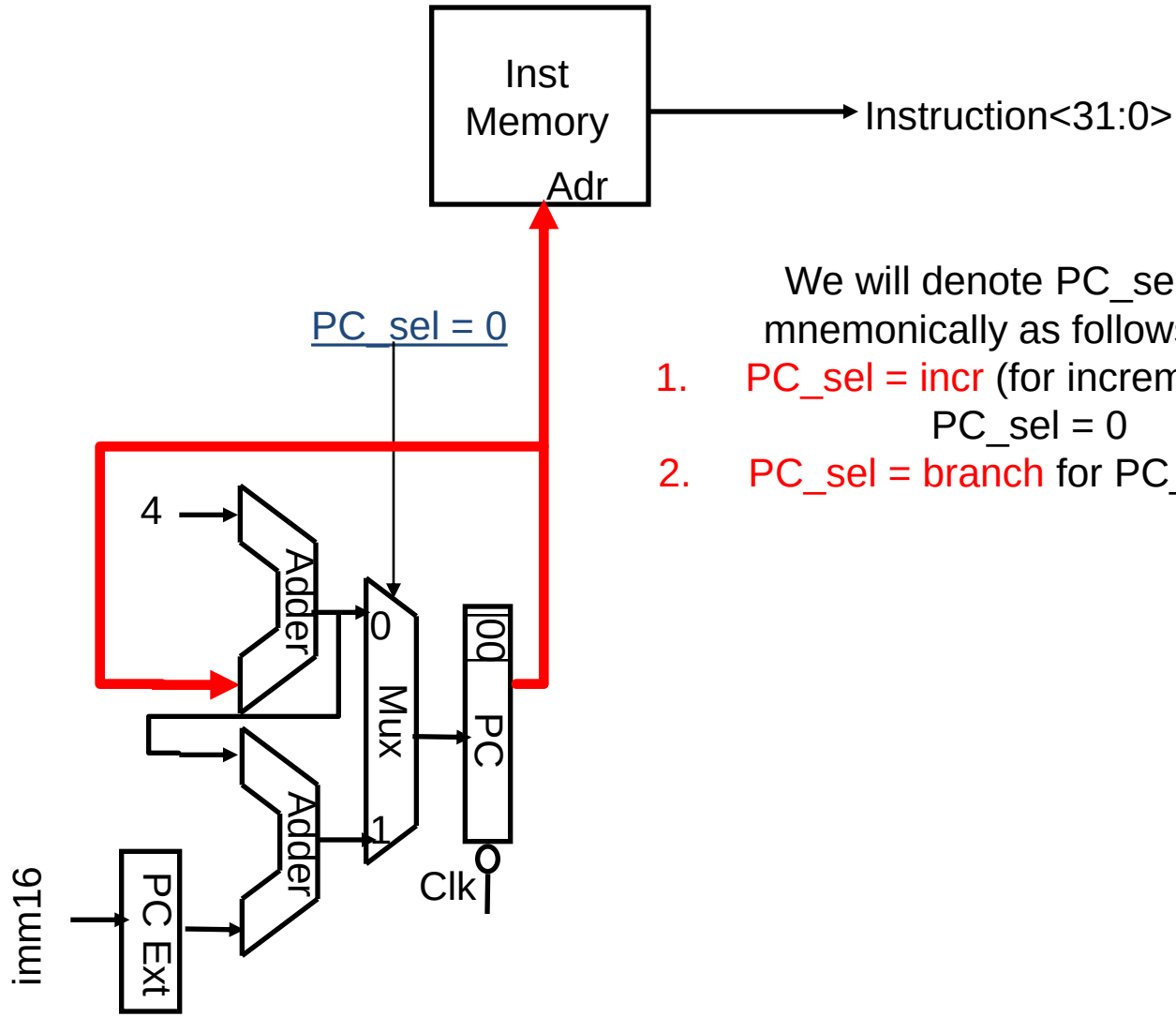
- We've done the datapath
- The Five Classic Components of a Computer



- Designing the Control for the Single Cycle Datapath

Instruction Fetch Unit at the Beginning of `Add`

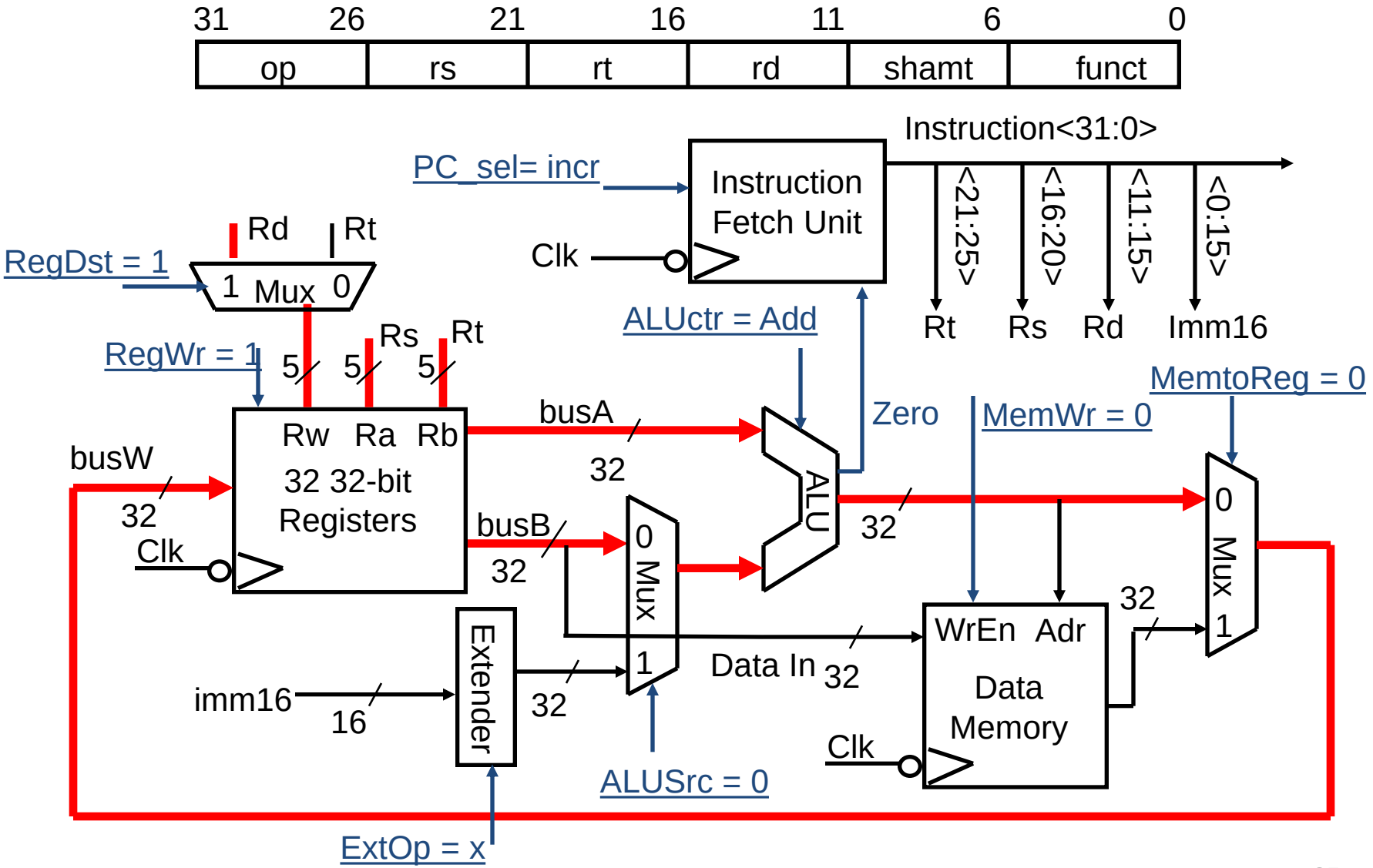
– This is the same for all instructions



We will denote PC_sel mnemonically as follows:

1. PC_sel = incr (for increment) for PC_sel = 0
2. PC_sel = branch for PC_sel = 1

The Single Cycle Datapath during `Add`



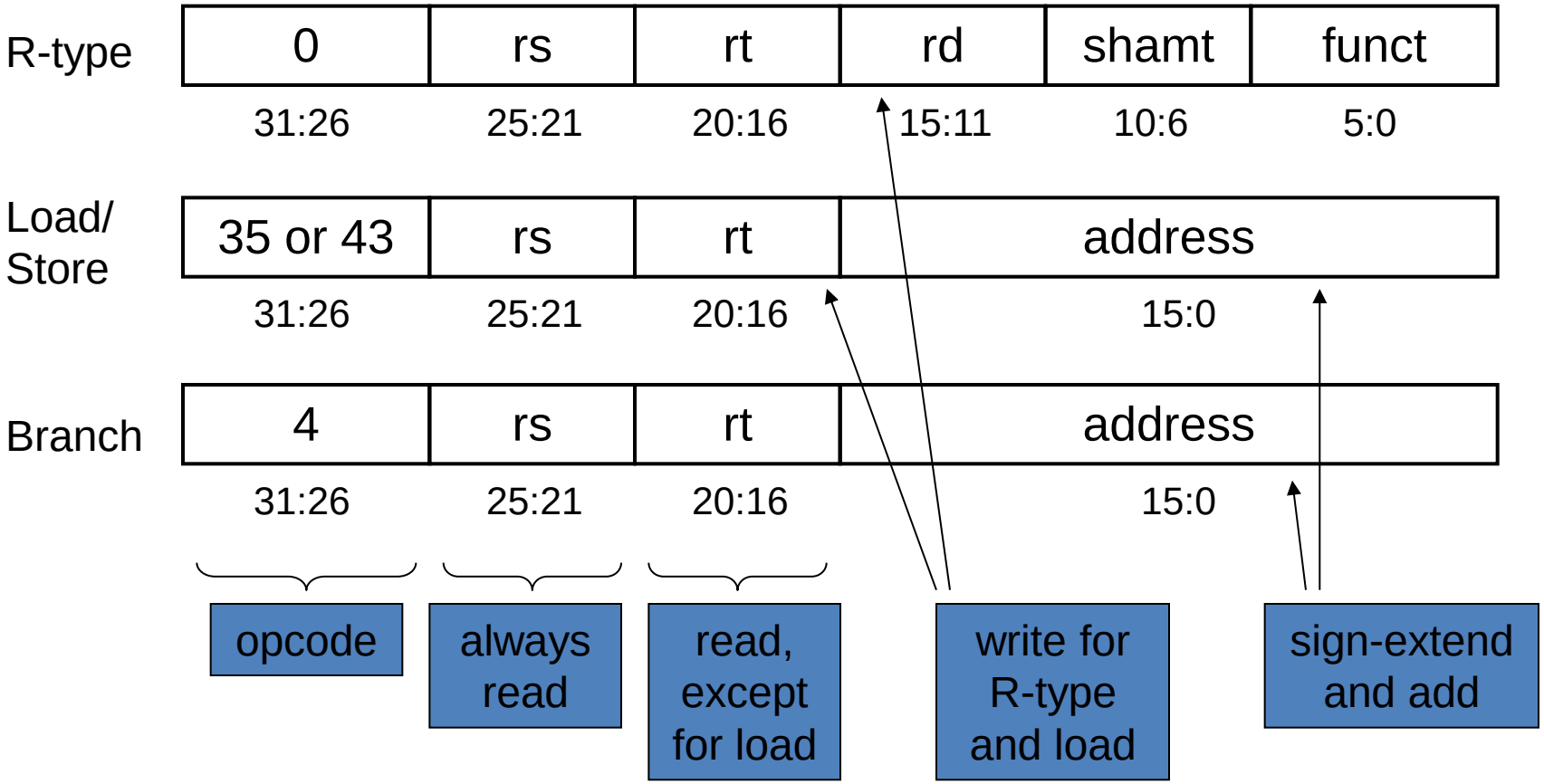
Mnemonics for control signals

- Instead of keeping track of 0 or 1 values for control signals, we will use mnemonics.

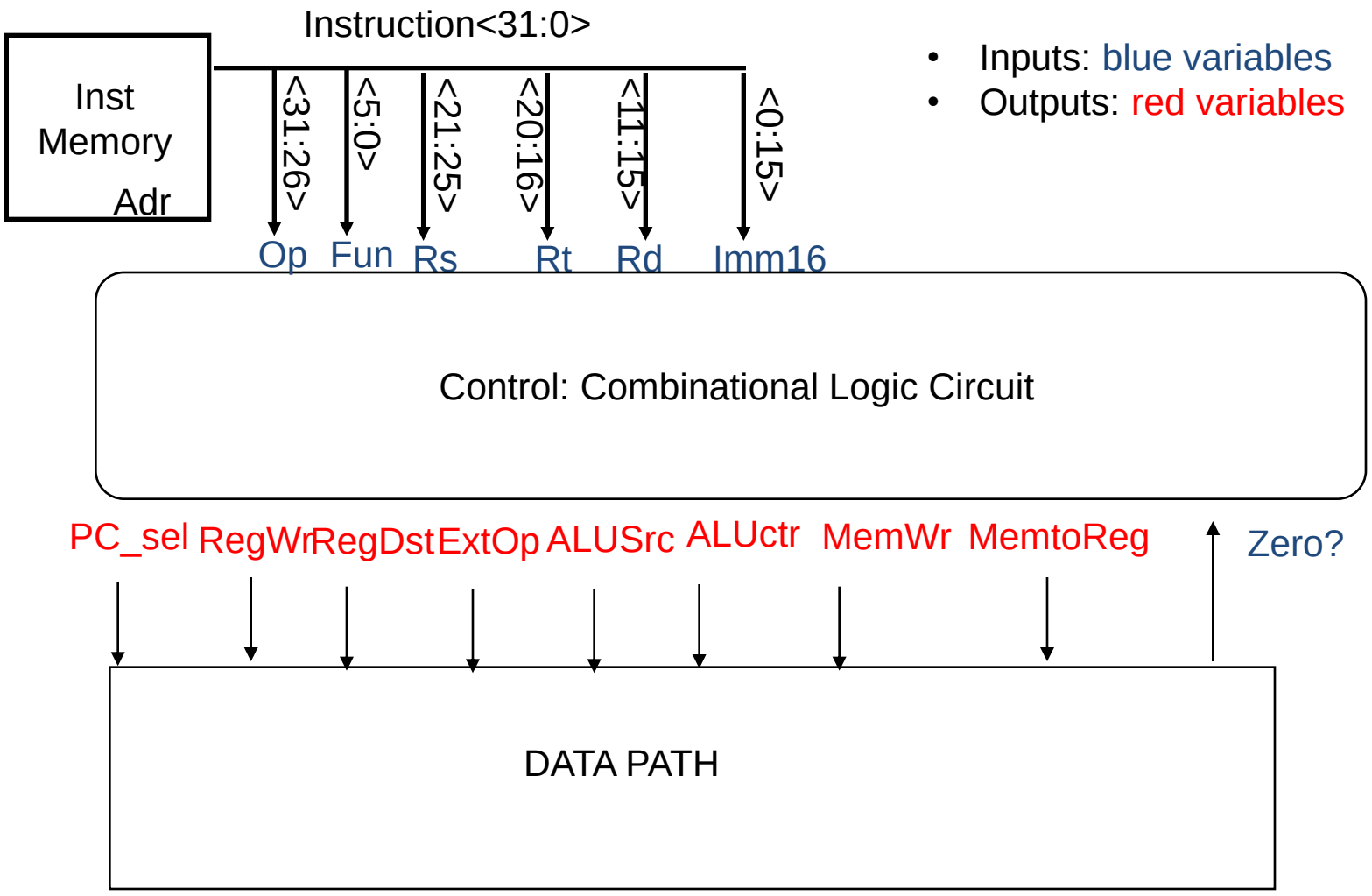
1. We already saw
   - PC_sel = incr (for increment) for PC_sel = 0
   - PC_sel = branch (for branch target address) PC_sel = 1
- Other signals:
   - ALUsrc = RegB or Imm
   - ALUctr = “add” or “sub” etc.
   - ExtOp = “Sn” for sign or “Z” for zero
   - Other controls (RegWr), we will just list the control if they are 1 and omit them if they are 0. (We’ll see example in the next few slides).

The Main Control Unit

- Control signals derived from instruction



Datapath → Control



- Inputs: blue variables
- Outputs: red variables

A Summary of Control Signals (mnemonics used)

inst	Register Transfer	
ADD	R[rd] ← R[rs] + R[rt];	PC ← PC + 4
	ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, PC_sel = incr	
SUB	R[rd] ← R[rs] – R[rt];	PC ← PC + 4
	ALUsrc = RegB, ALUctr = “sub”, RegDst = rd, RegWr, PC_sel = incr	
ORi	R[rt] ← R[rs] + zero_ext(Imm16);	PC ← PC + 4
	ALUsrc = Im, Extop = “Z”, ALUctr = “or”, RegDst = rt, RegWr, PC_sel = incr	
LOAD	R[rt] ← MEM[R[rs] + sign_ext(Imm16)];	PC ← PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemtoReg, RegDst = rt, RegWr,	PC_sel = incr
STORE	MEM[R[rs] + sign_ext(Imm16)] ← R[rs];	PC ← PC + 4
	ALUsrc = Im, Extop = “Sn”, ALUctr = “add”, MemWr, PC_sel = incr	
BEQ	if (R[rs] == R[rt]) then PC ← PC + 4 + sign_ext(Imm16)]*4 else PC ← PC + 4	
	PC_sel = branch, ALUctr = “sub”	

ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

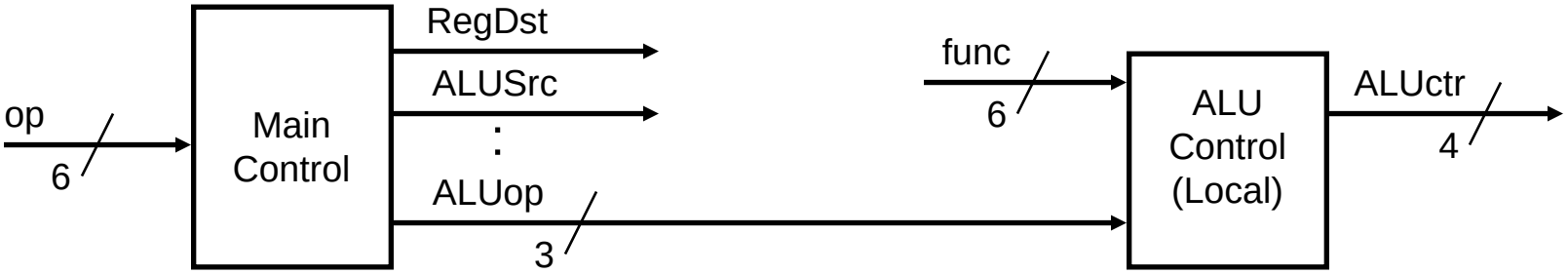
ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Logic for each control signal

We're done with designing local ALU ctrl. Now we look at the datapath control signals:

- PC_sel ← if (OP == BEQ) then EQUAL else 0
- ALUsrc ← if (OP == "Rtype") then "regB" else "immed"
- ALUctr ← if (OP == "Rtype") then funct
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp ← if (OP == ORi) then "zero" else "sign"
- MemWr ← (OP == Store)
- MemtoReg ← (OP == Load)
- RegWr: ← if ((OP == Store) || (OP == BEQ)) then 0 else 1
- RegDst: ← if ((OP == Load) || (OP == ORi)) then 0 else 1

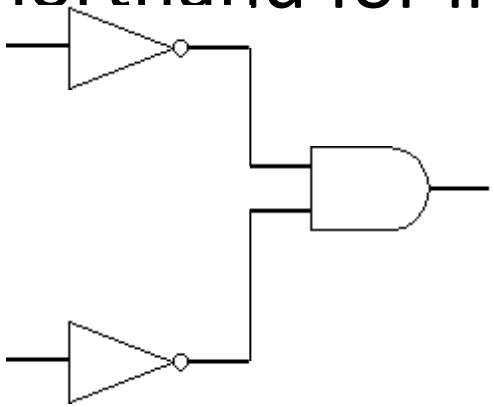
The "Truth Table" for the Main Control



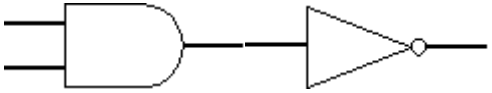
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

Notational explanation for next slide

- Shorthand for inverted input



is denoted by

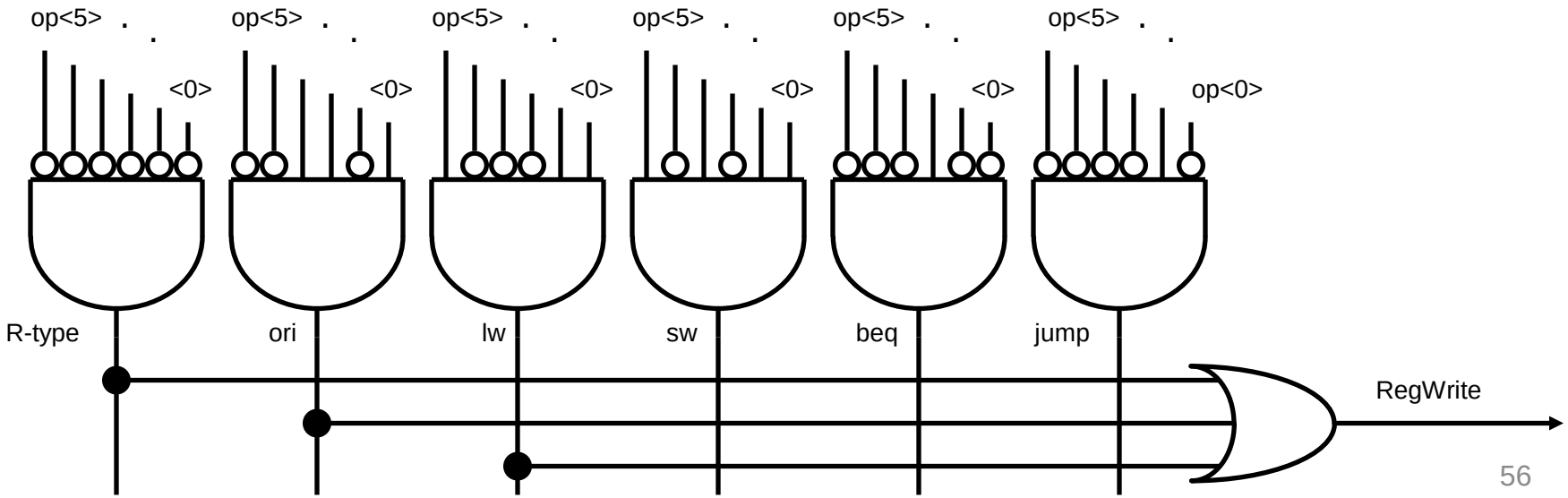


is denoted by

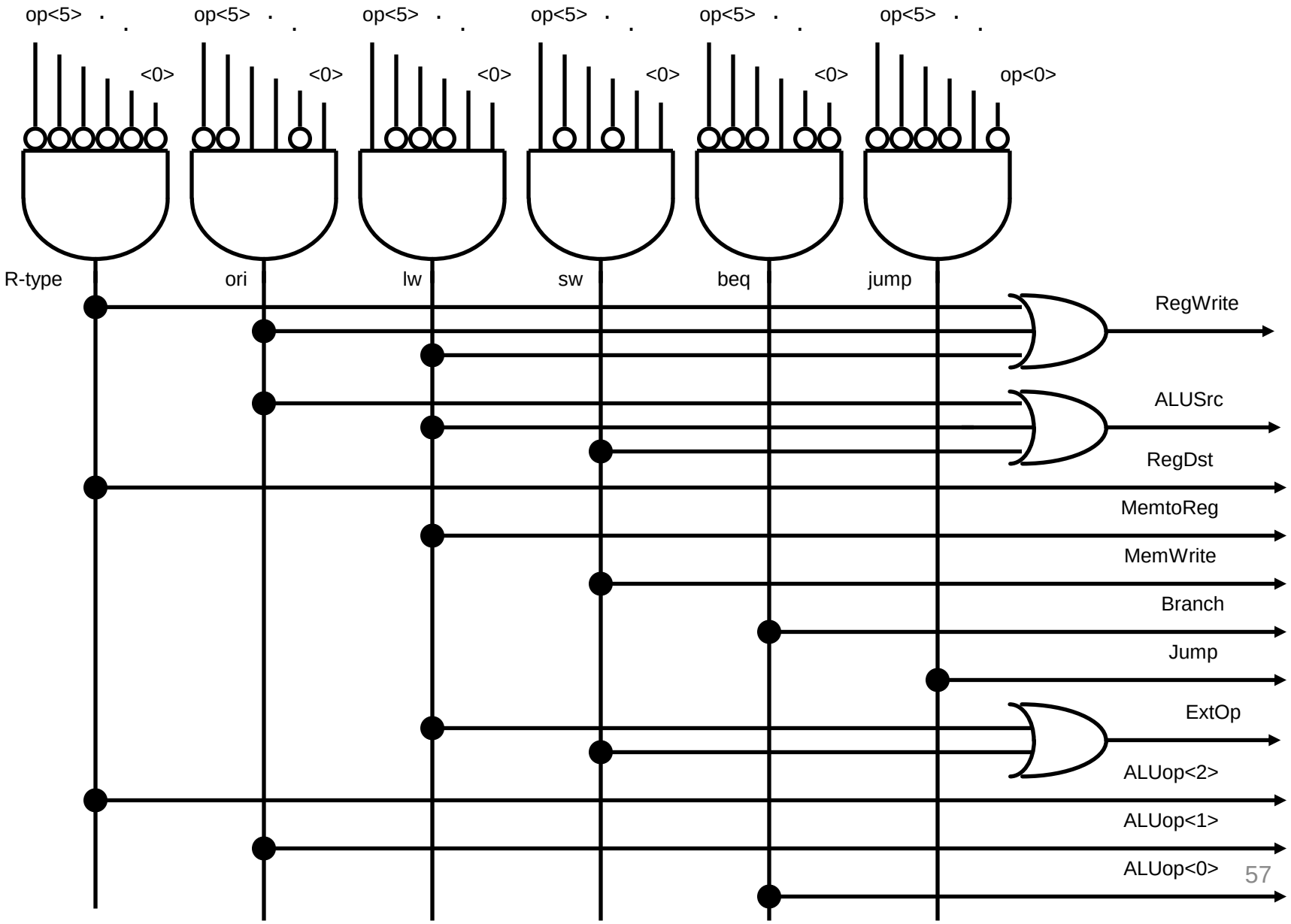
The “Truth Table” for RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

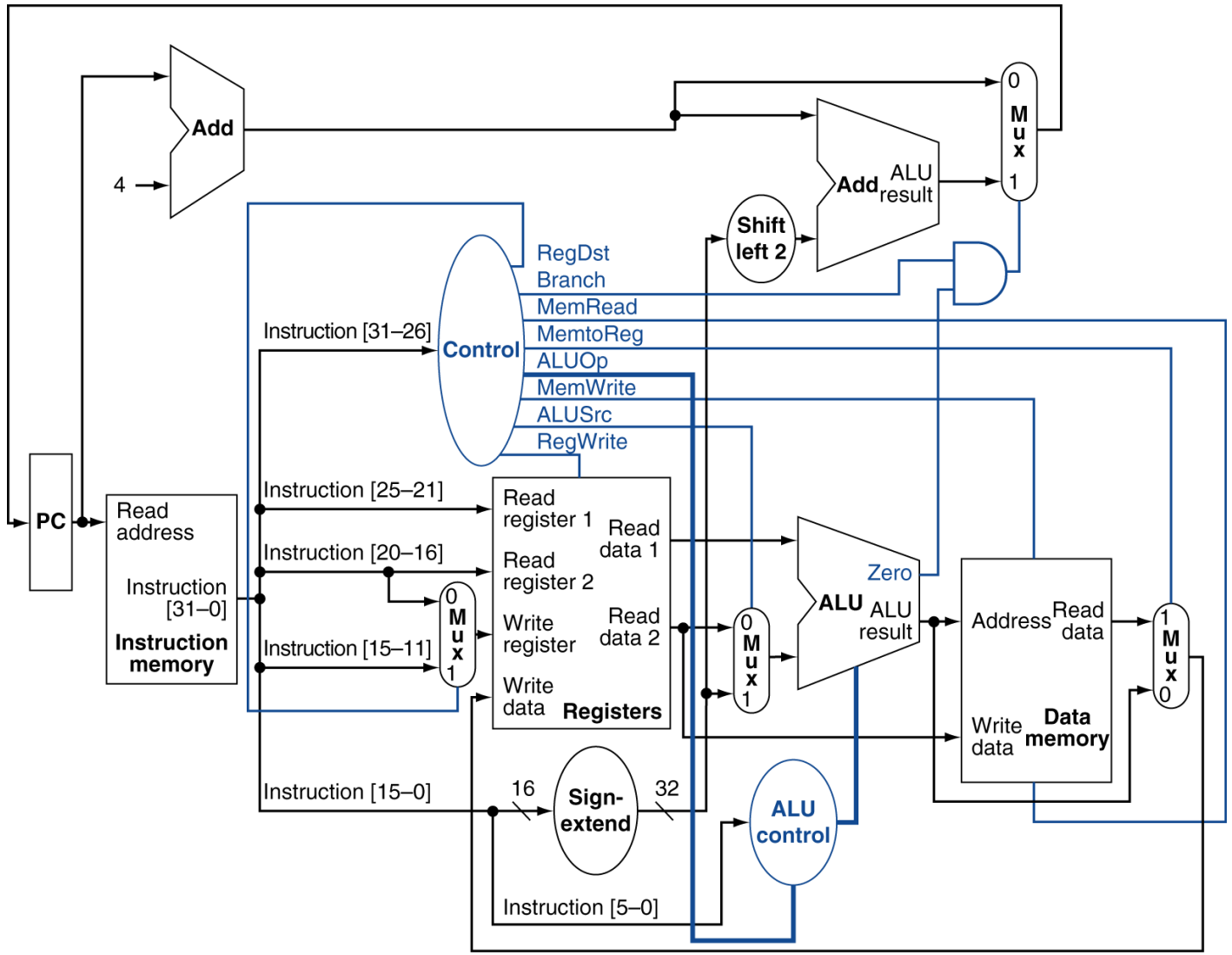
- RegWrite = R-type + ori + lw
  - = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  - + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)
  - + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



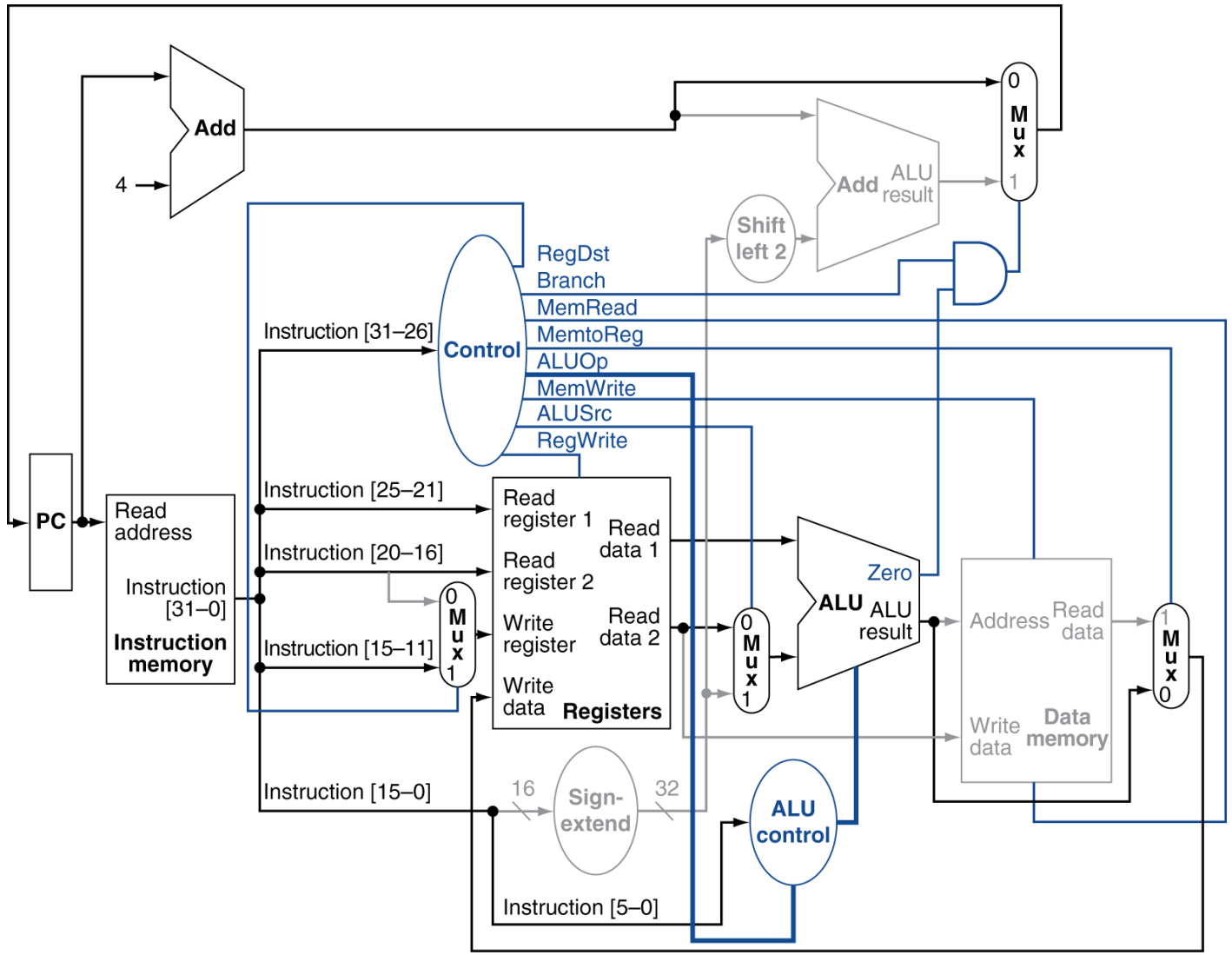
PLA Implementation of the Main Control



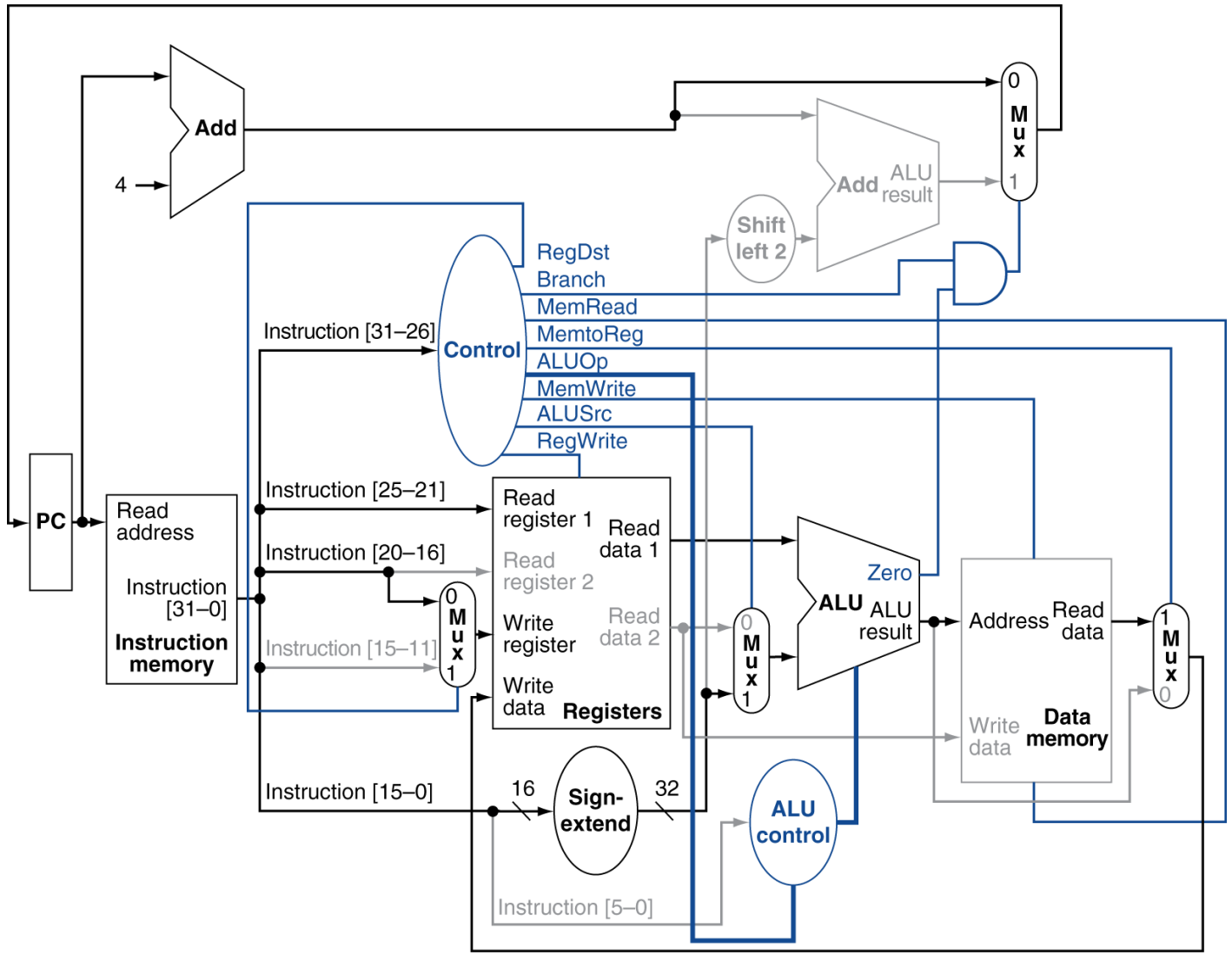
Datapath With Control



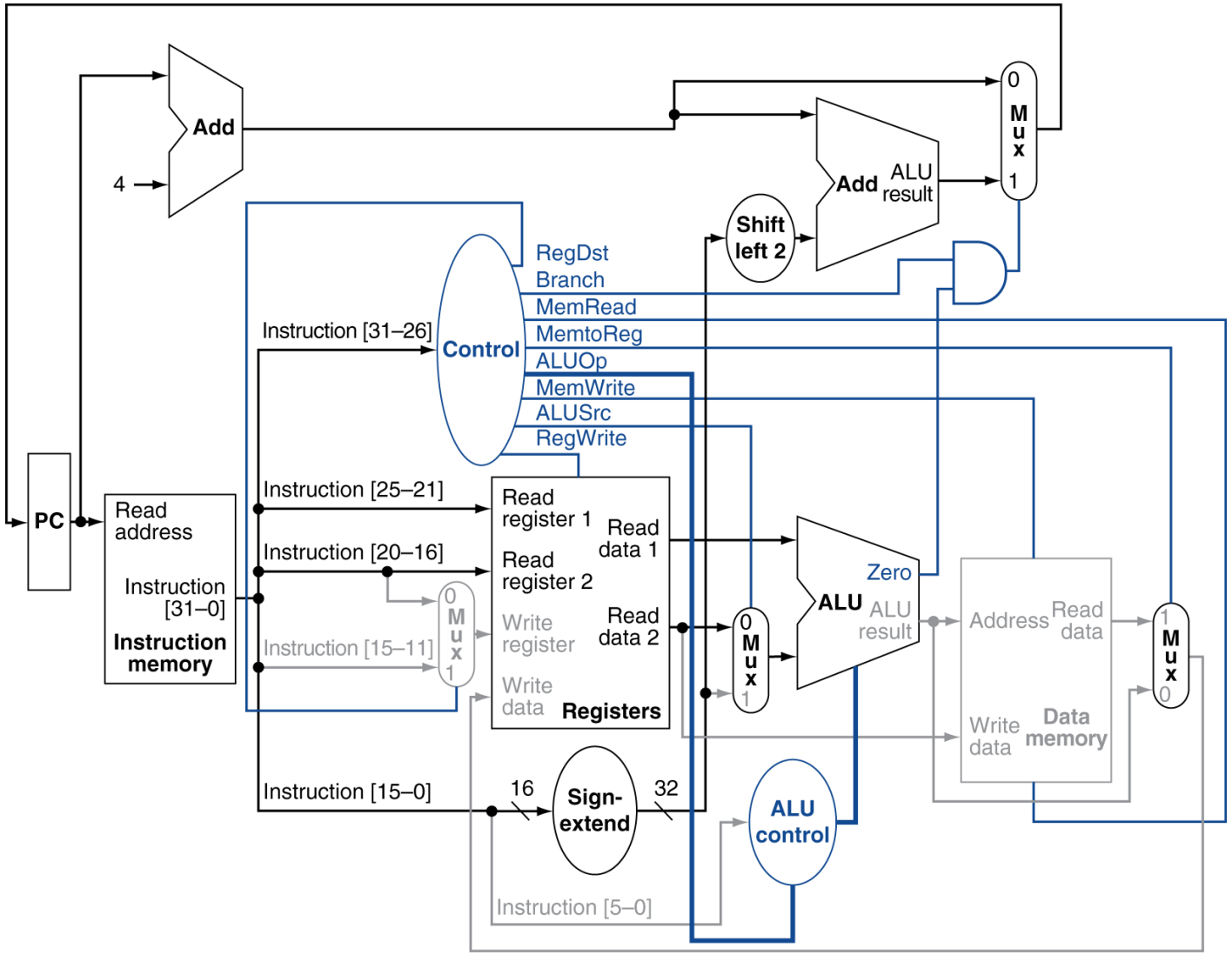
R-Type Instruction



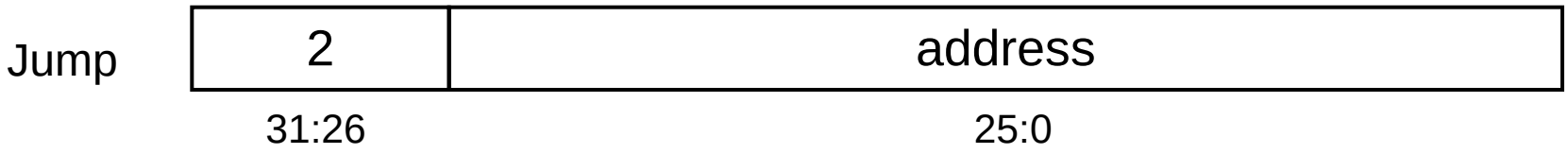
Load Instruction



Branch-on-Equal Instruction

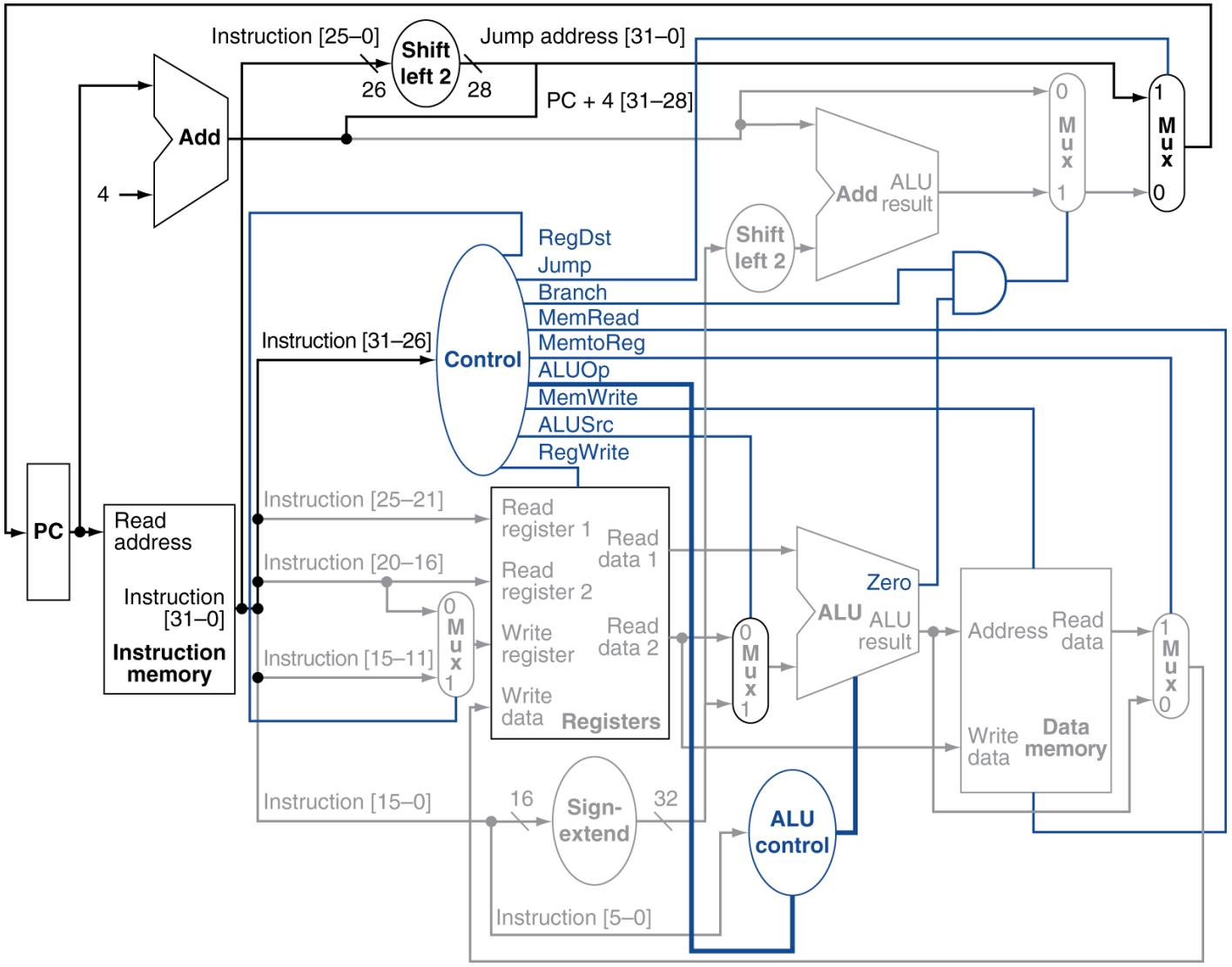


Implementing Jumps



- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00 (shift left by 2 bits to get byte address)
- Need an extra control signal decoded from opcode (Jump in the next slide)

Datapath With Jumps Added

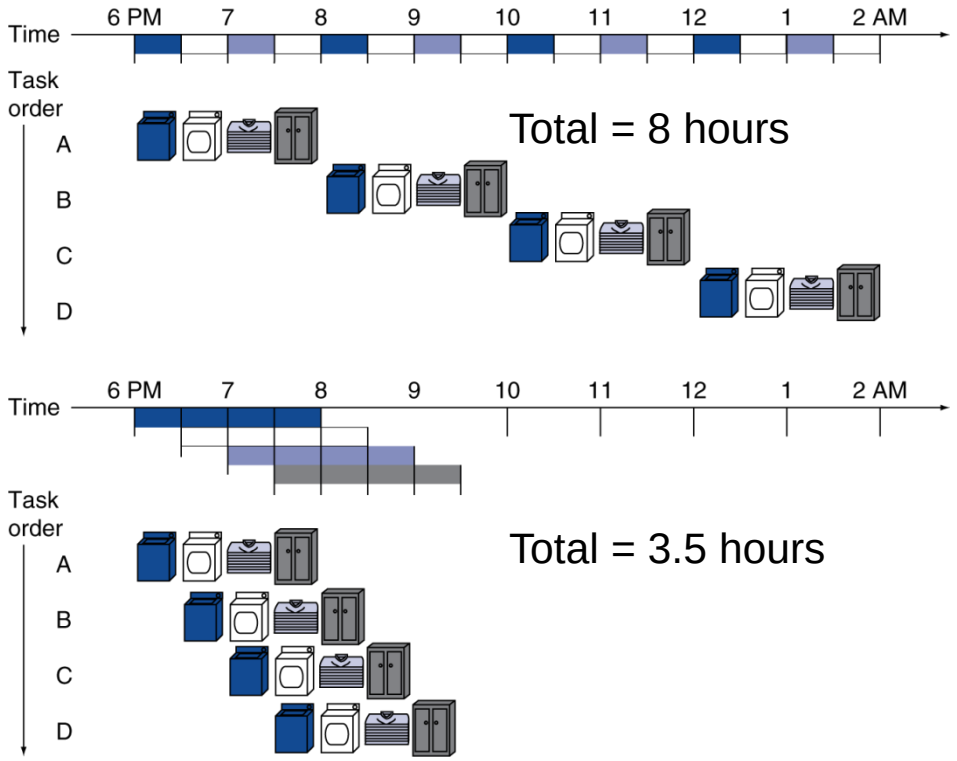


Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

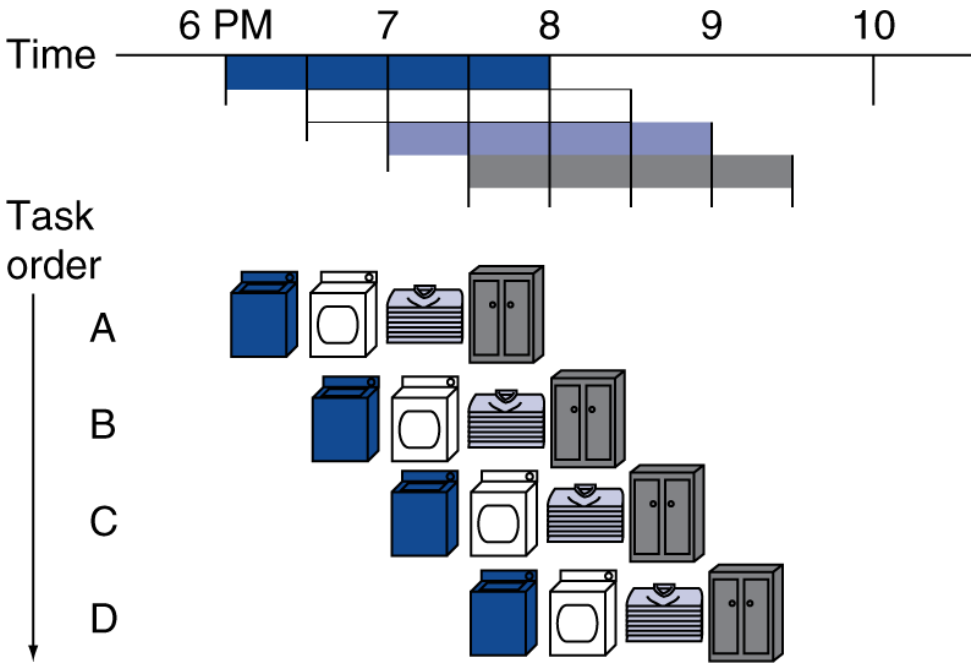
Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



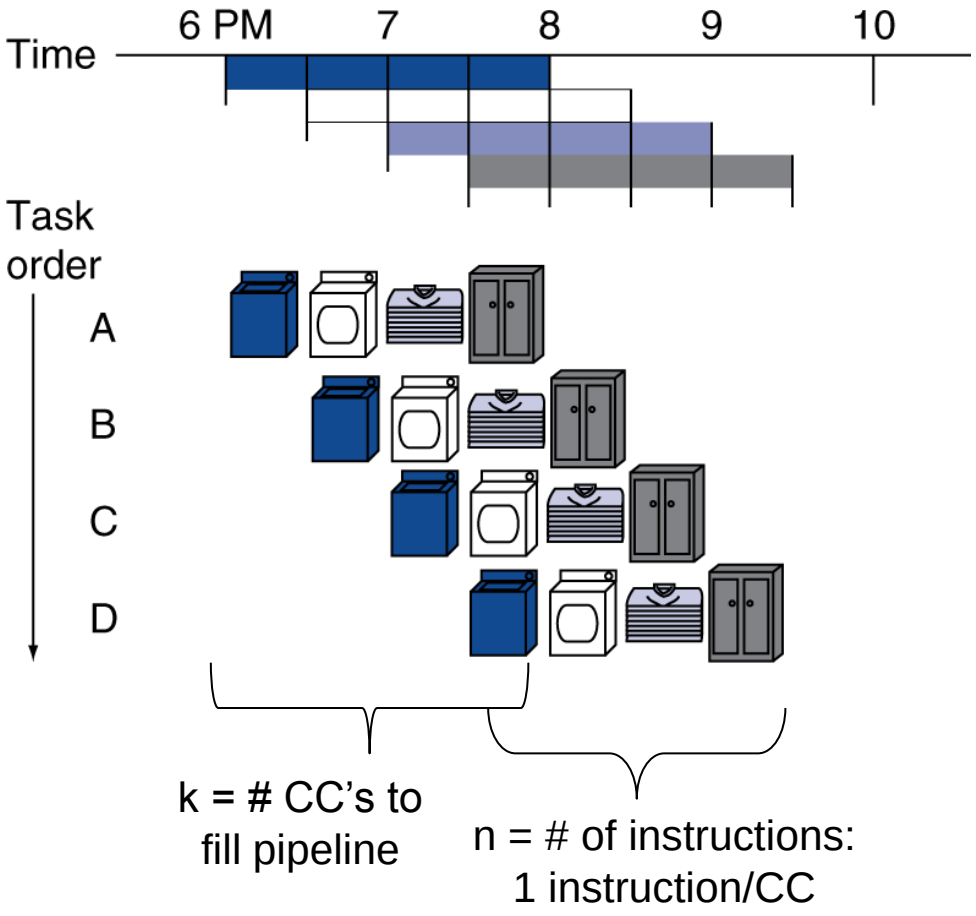
- Four loads:
  - Speedup
    = 8/3.5 = 2.3
- Non-stop (steady state):
  - Speedup
    ≈ 4
    = number of stages

Pipelining Lessons



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

Pipelining Lessons

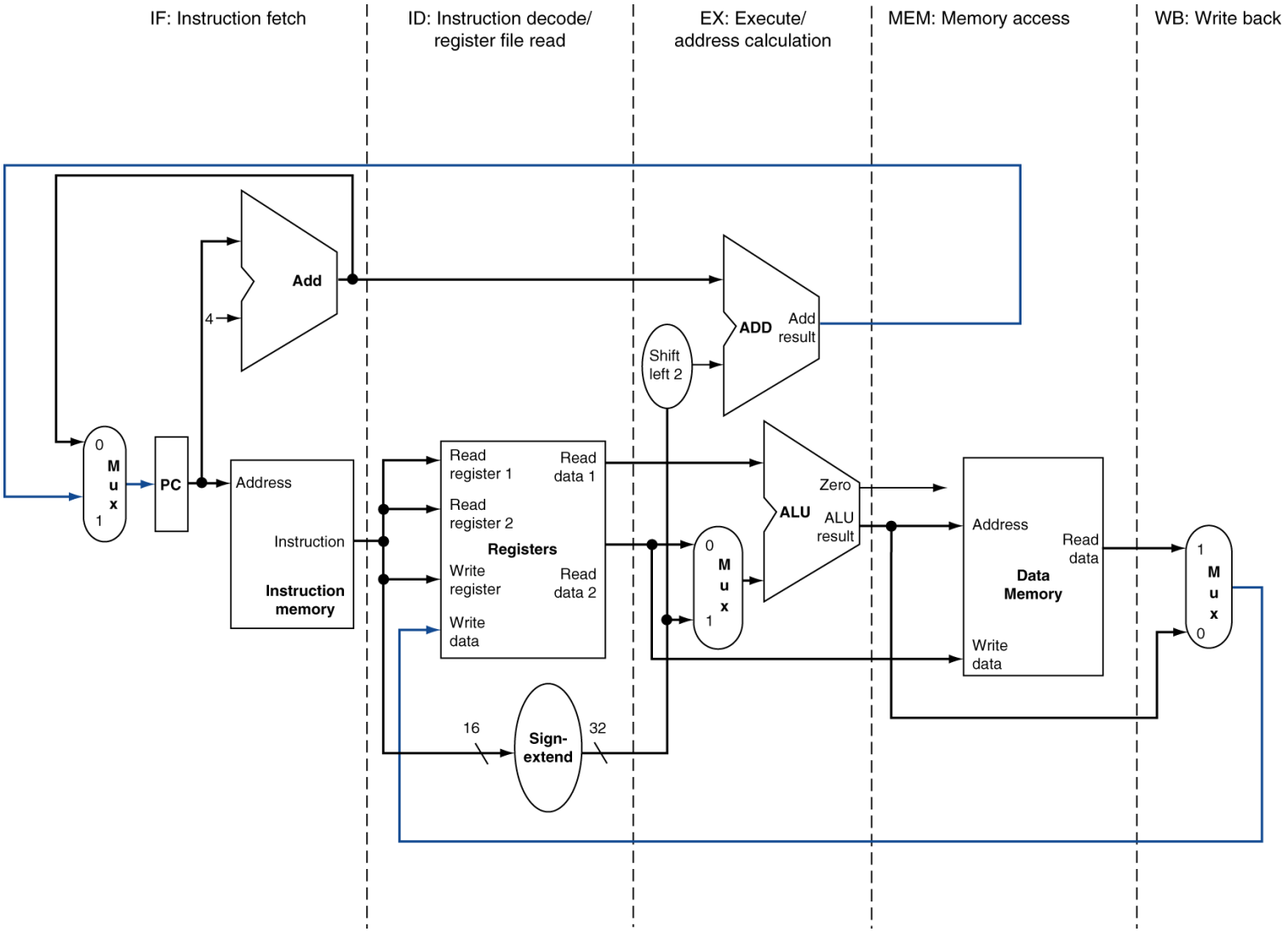


- # stages: k (=4 in this case)
- # instructions: n (=4 in this case)
- Total # of clock cycles = k+n-1 (=4+4-1=7 in this case).
- Without pipeline: n*k
- With pipeline: k+n-1
- Speedup $= \frac{n*k}{k+n-1}$
- As $n \to \infty$,
- $\lim_{n\to\infty} \frac{n*k}{k+n-1} = k$

MIPS Pipeline

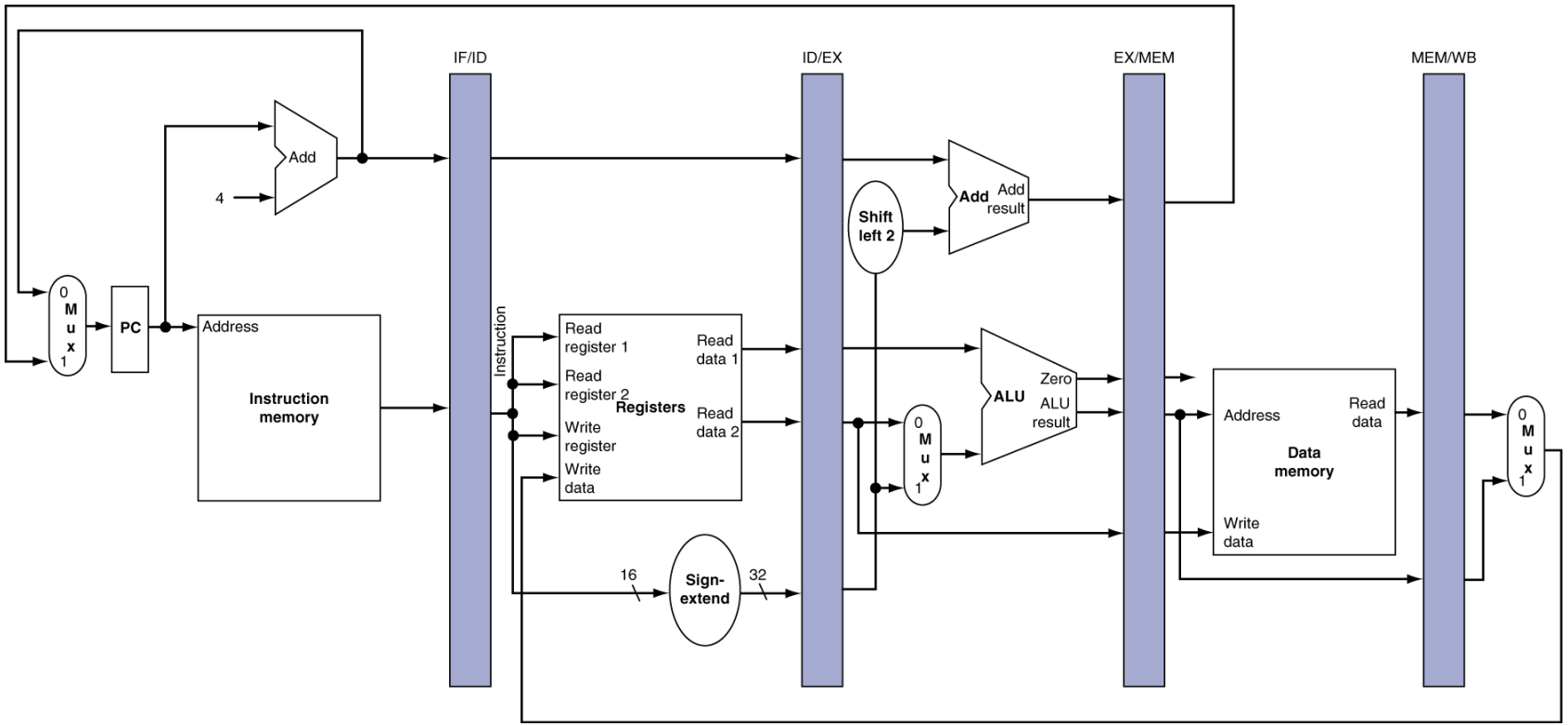
- Five stages, one step per stage
  1. IF: Instruction Fetch from memory
  2. ID: Instruction Decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

MIPS Pipelined Datapath (start with single cycle data path)



Pipeline registers

- Separate pipeline stages by inserting registers along the data path between stages.
  - To hold information produced in previous cycle

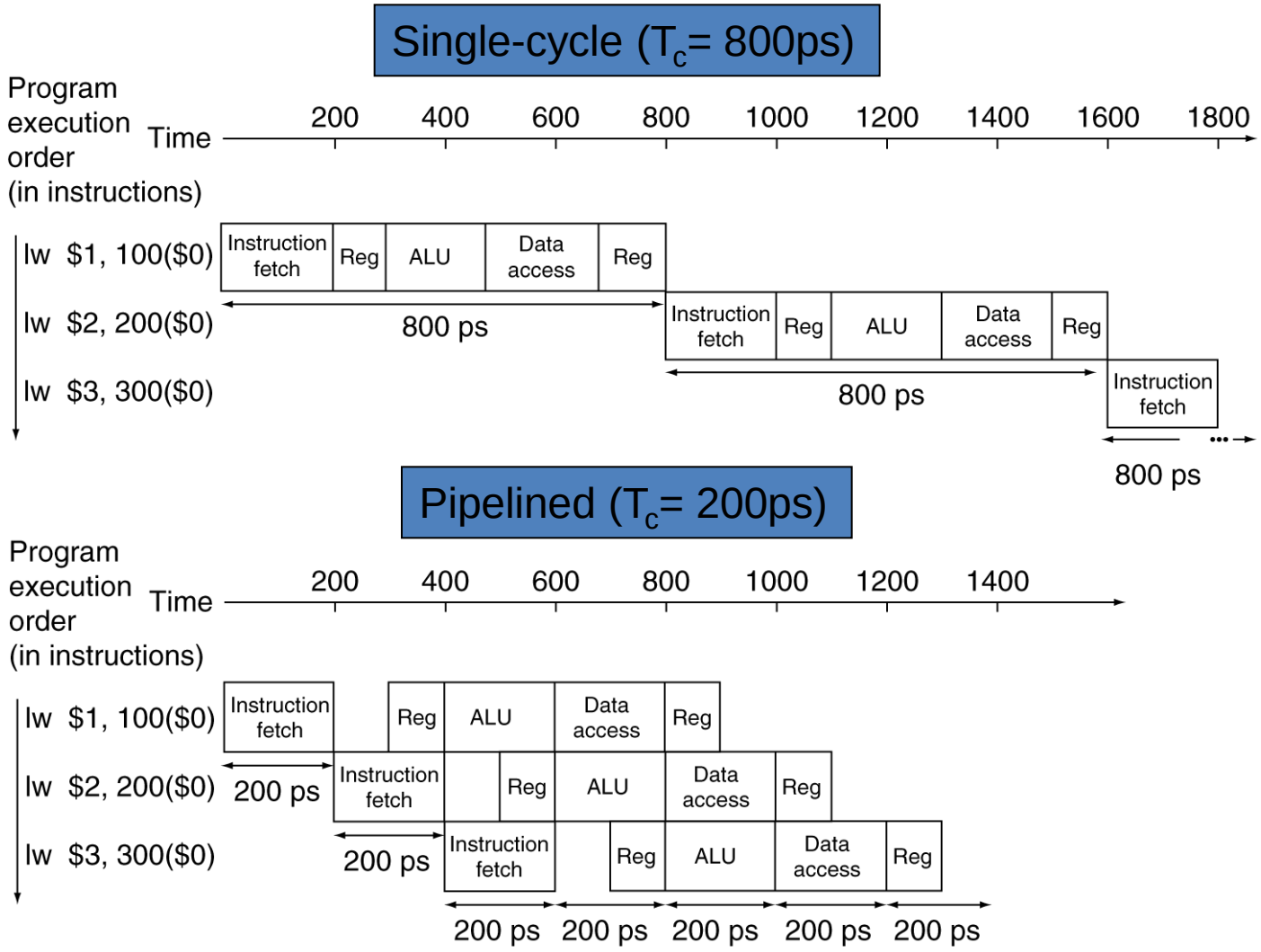


Pipeline Performance

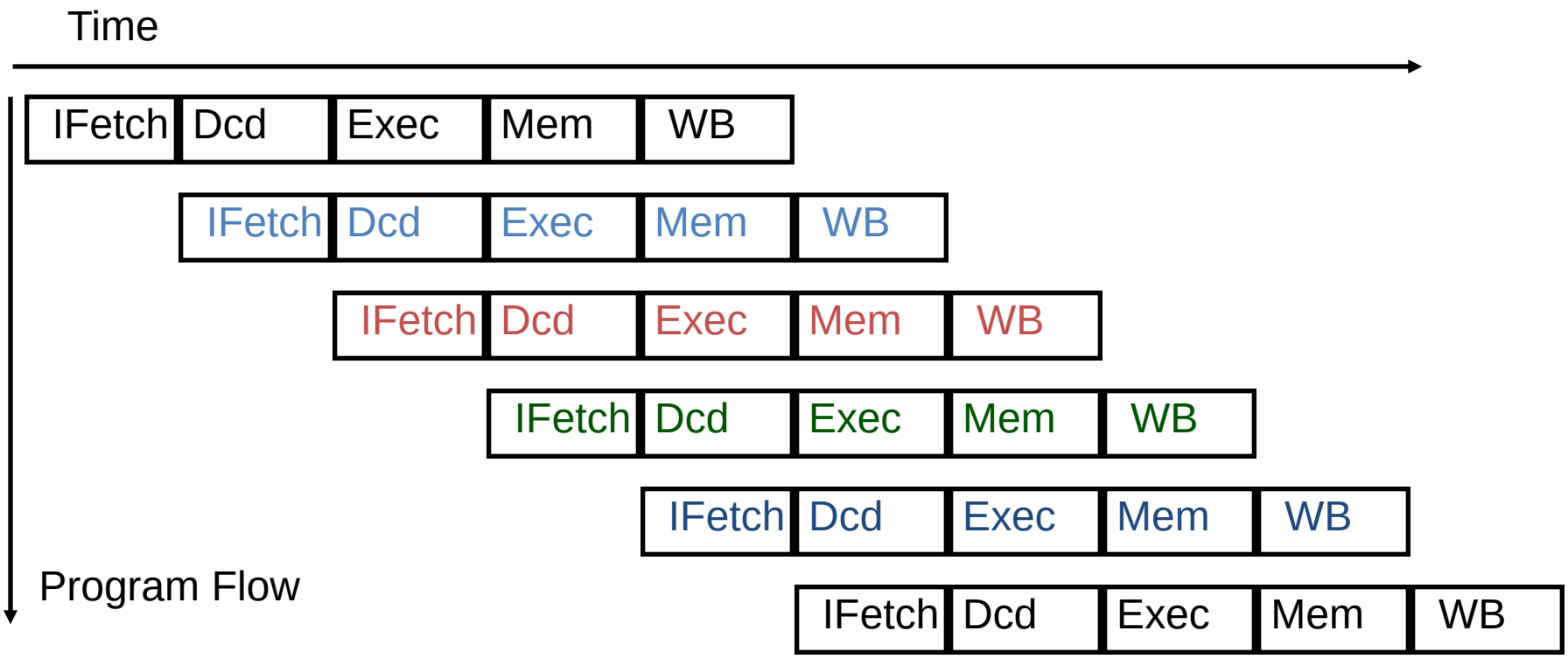
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

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

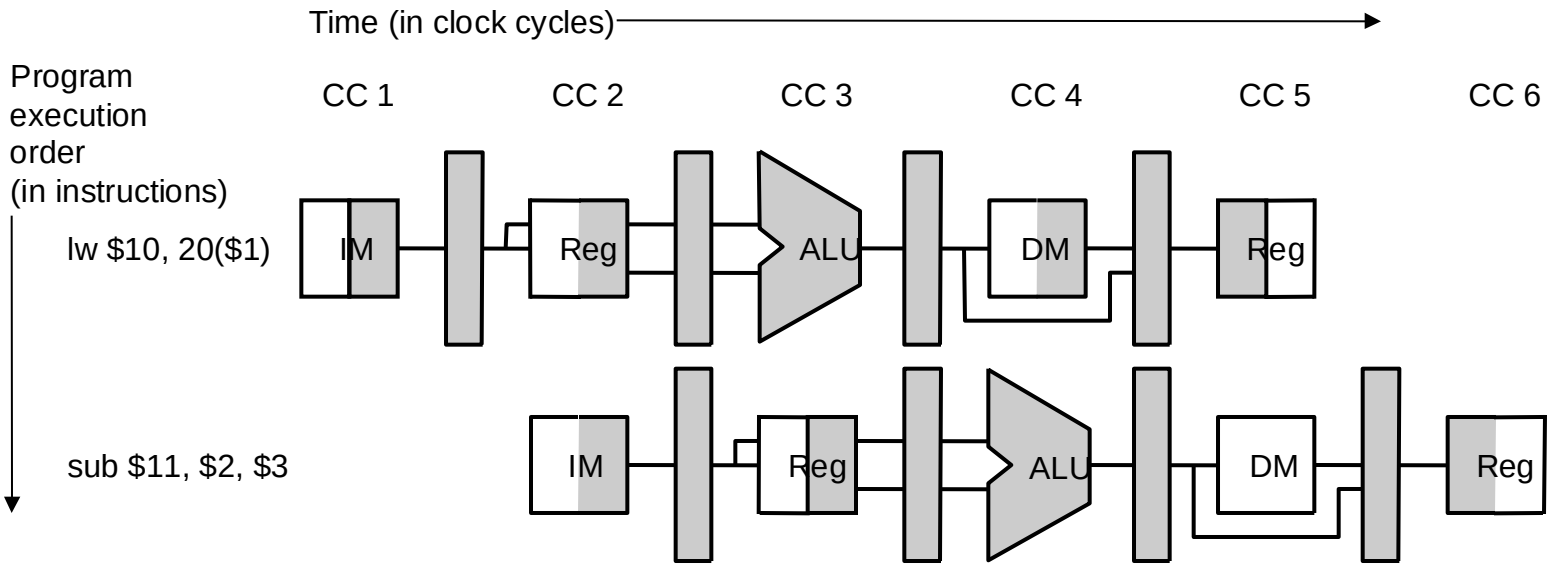
Pipeline Performance



Conventional Pipelined Execution Representation



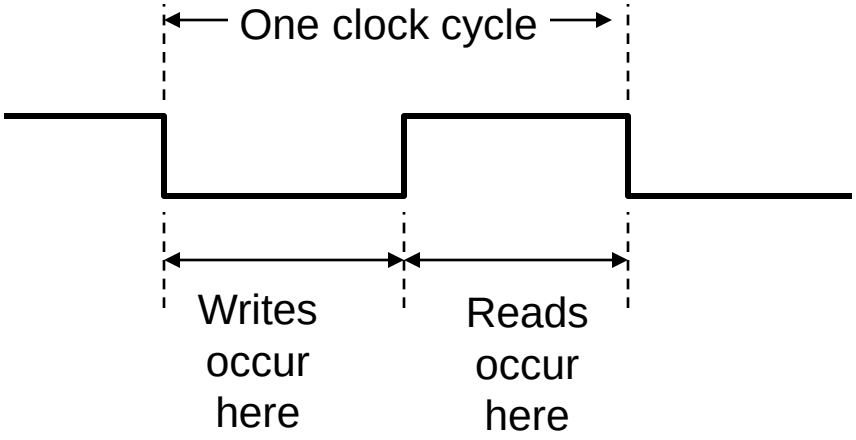
Graphically Representing Pipelines



- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths
  - Note: the assumption is that the register file is written in the first half of the clock cycle and read in the second half of the clock cycle (see appendix B and two-phase clocking techniques for this) [this will be important when we are looking at pipeline hazard conditions]

Two-phased clocking technology

- Writes occur in first half of the clock cycle, reads occur in the second half.



- Read Appendix B Section B.7
  - In particular, at the end of the section read “Elaboration” paragraph.

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time

$$t_{pipelined} = \frac{t_{nonpipelined}}{\#\text{ of stages}}$$

- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not necessarily decrease!

Pipelining and ISA Design

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

Hazards

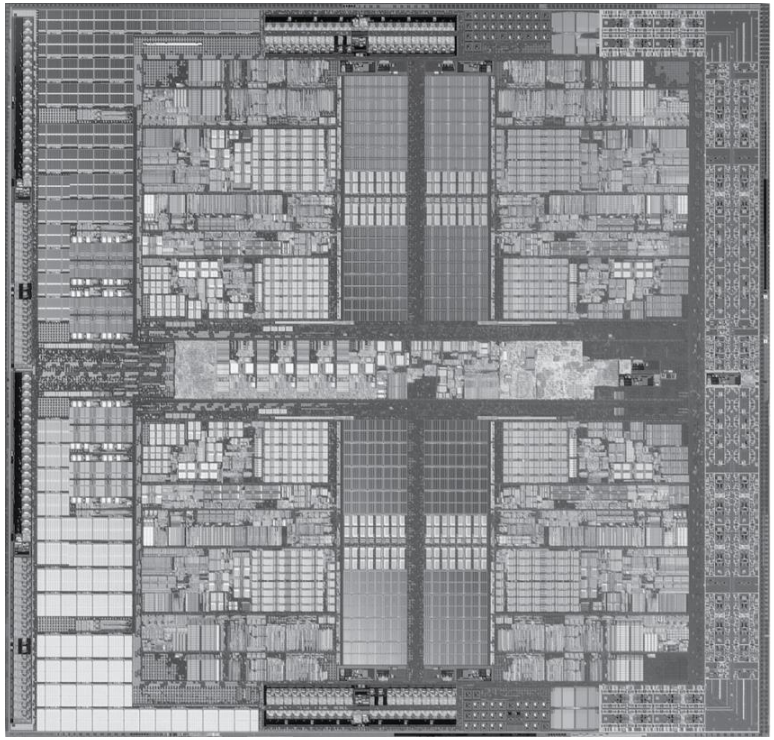
- Situations that prevent starting the next instruction in the next cycle
- Structural hazards
  - A required resource (e.g., memory) is busy
- Data hazards
  - Need to wait for previous instruction to complete its data read/write
- Control hazards
  - Deciding on control action depends on previous instruction (e.g., beq)

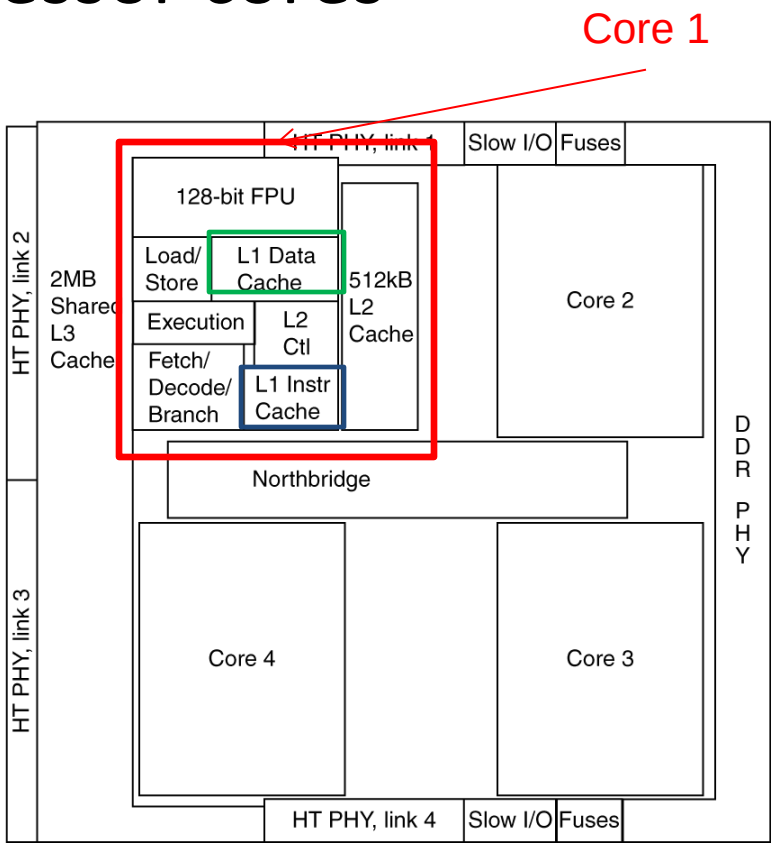
Structural Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or more realistically *separate instruction/data caches*

Recall from first lecture

- AMD Barcelona: 4 processor cores



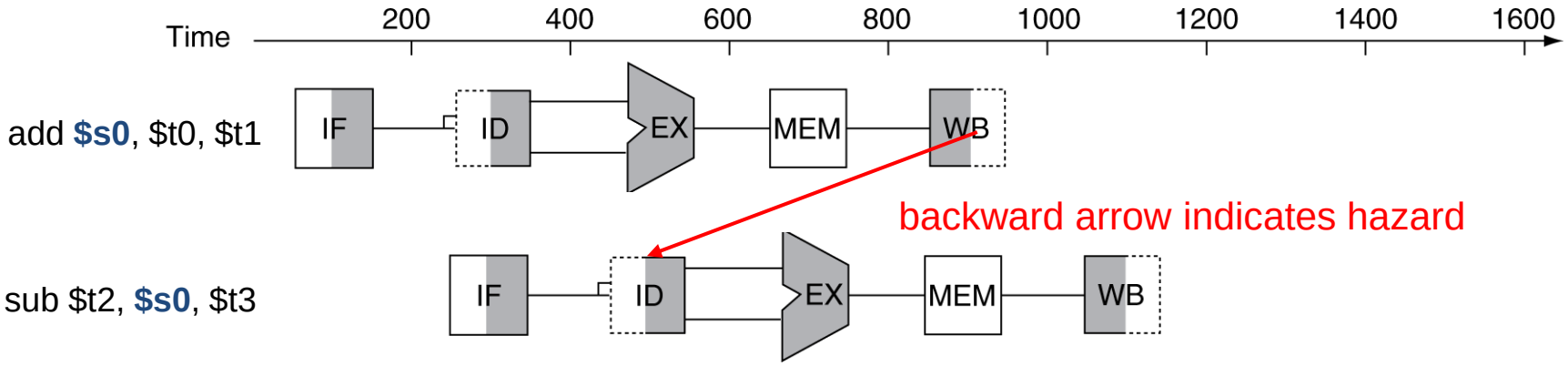


Data Dependencies

- Three types of data dependencies exist:
  - Read After Write (RAW)
    - Example:
    - ```
      add  $s0, $t0, $t1
      sub  $t2, $s0, $t3
      ```
  - Write After Read (WAR)
    - Example:
    - ```
      sub  $t2, $s0, $t3
      ```
    - ```
      add  $s0, $t0, $t1
      ```
  - Write After Write (WAW)
    - Example:
    - add $s0, $s1, $s2
    - sub $s0, $t1, $t2
      ```

Data Hazards

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

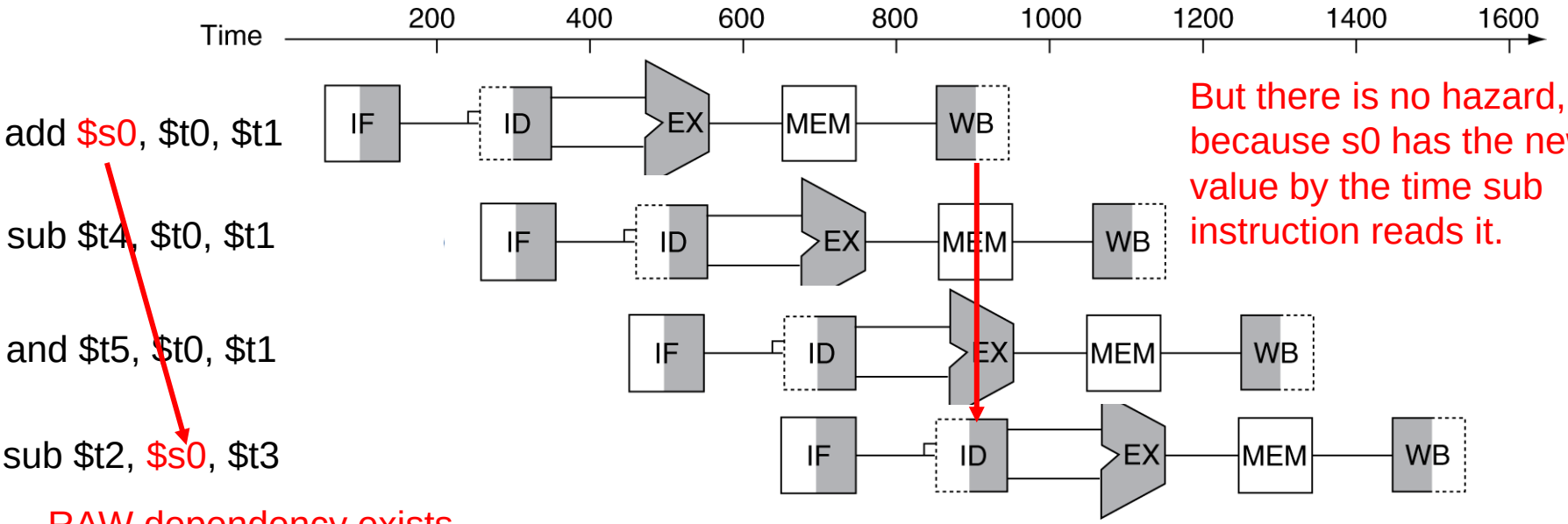


Data dependency vs. hazard

- Data dependencies may result in hazards
- … But they do not have to.
- One may have data dependency without a data hazard.
- How?
  - Example:
    - ```
      add   $s0, $t0, $t1
      sub   $t4, $t0, $t1
      and   $t5, $t0, $t1
      sub   $t2, $s0, $t3
      ```
      ```

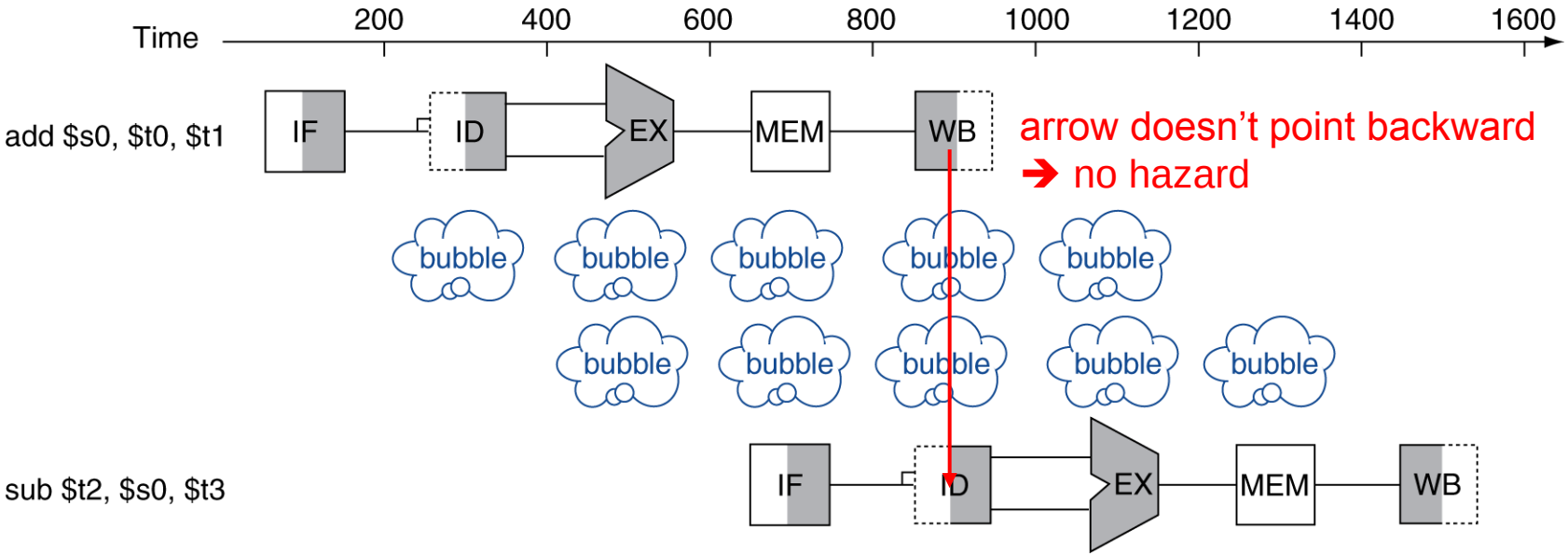
Data Hazards vs dependency

- add $s0, $t0, $t1
  add $t2, $t3, $t4
  add $t5, $t6, $t7
  sub $t8, $s0, $t9



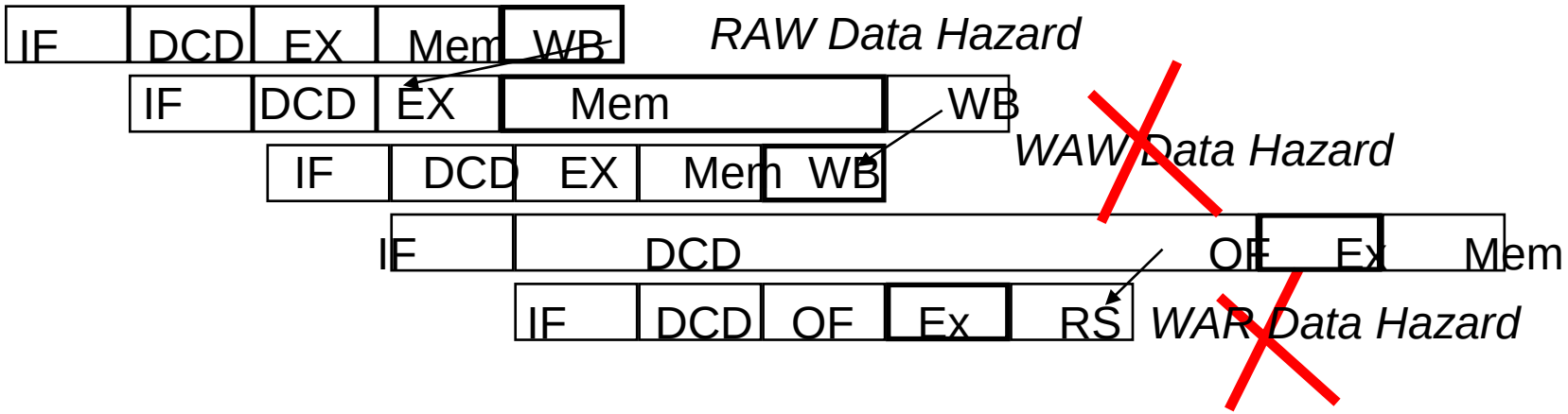
Data Hazards

- Or we can eliminate the hazard by appropriate stalls of instructions ➔ wasted clock cycles
  - `add  $s0, $t0, $t1`
    `sub  $t2, $s0, $t3`



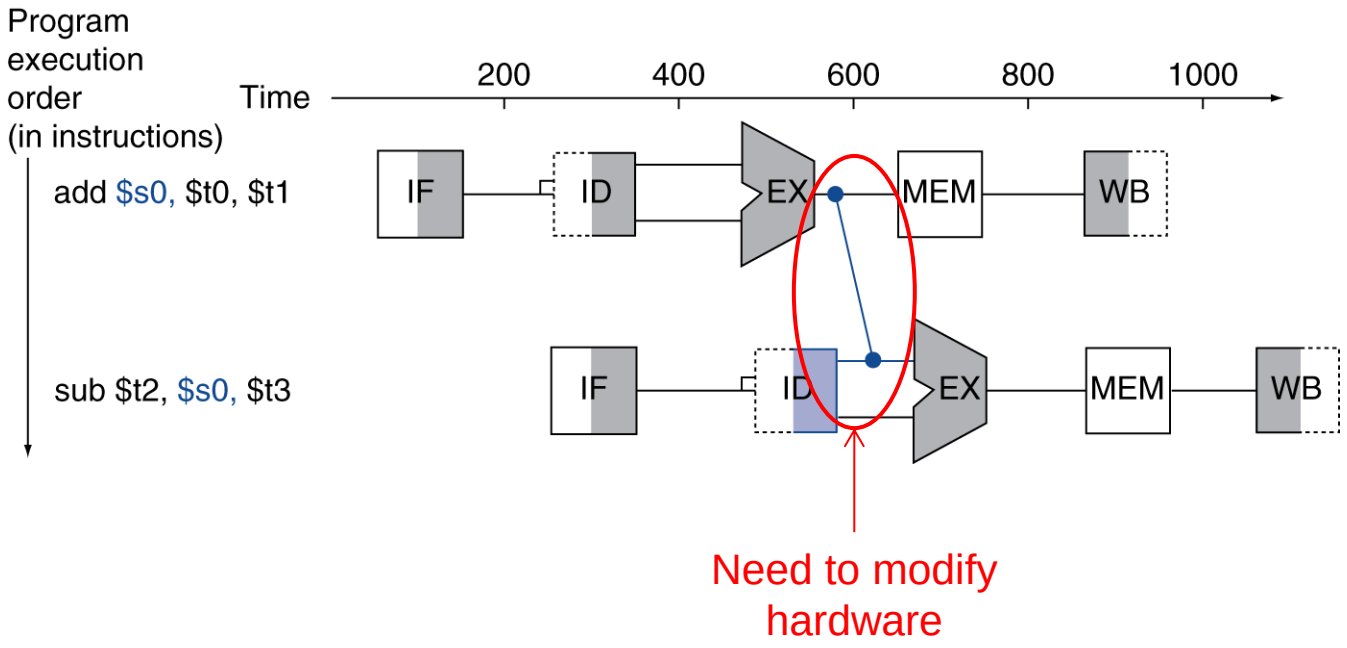
Data Hazards

- Avoid some “by design”
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eliminate WAW by doing all WBs in order (always at the last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)



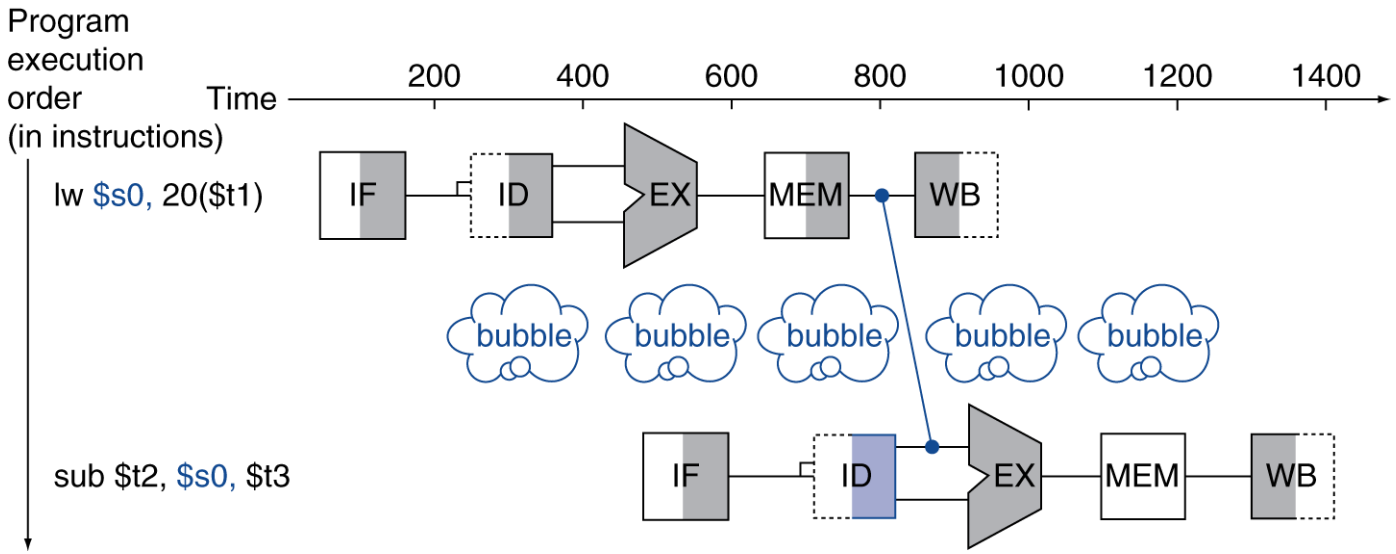
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



Load-Use Data Hazard

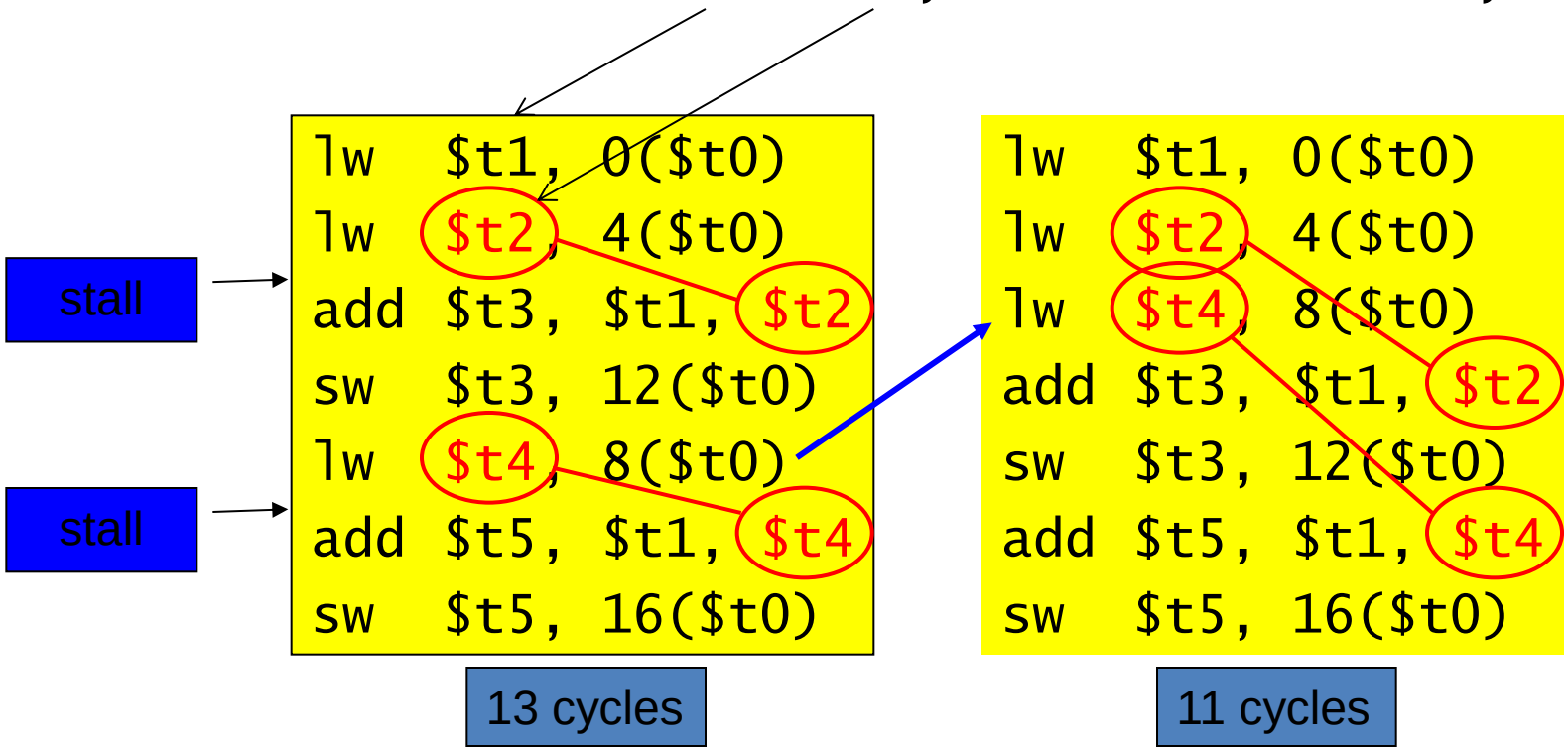
- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Need to modify hardware

Code Scheduling to Avoid Stalls: software solution if possible

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

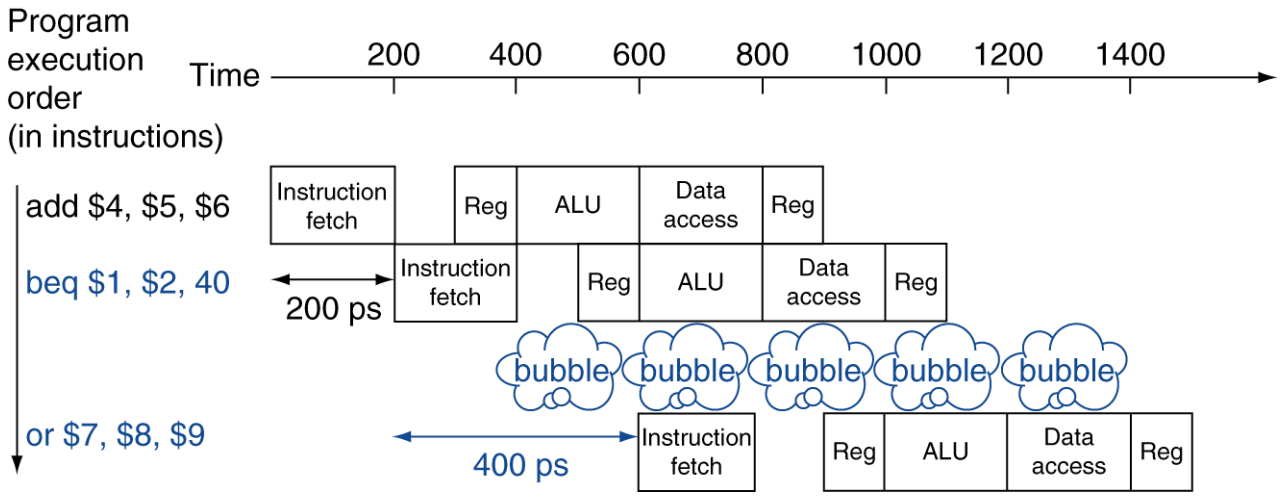


Control Hazards

- 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

Example code

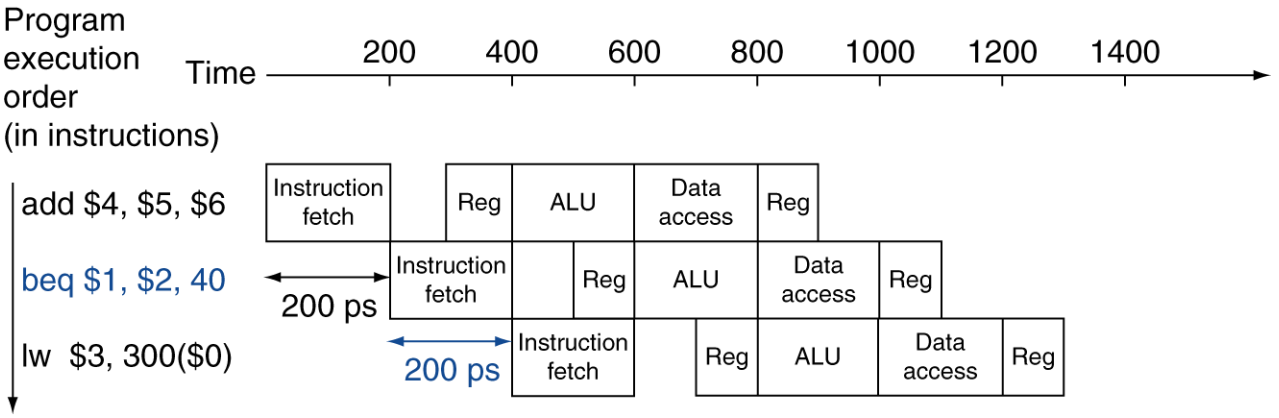
```
        add $4, $5, $6
        beq $1, $2, 40
        lw $3, 300($0)
        ...
40:     or $7, $8, $9
```

If beq condition = False, this instruction should be fetched (branch not taken)

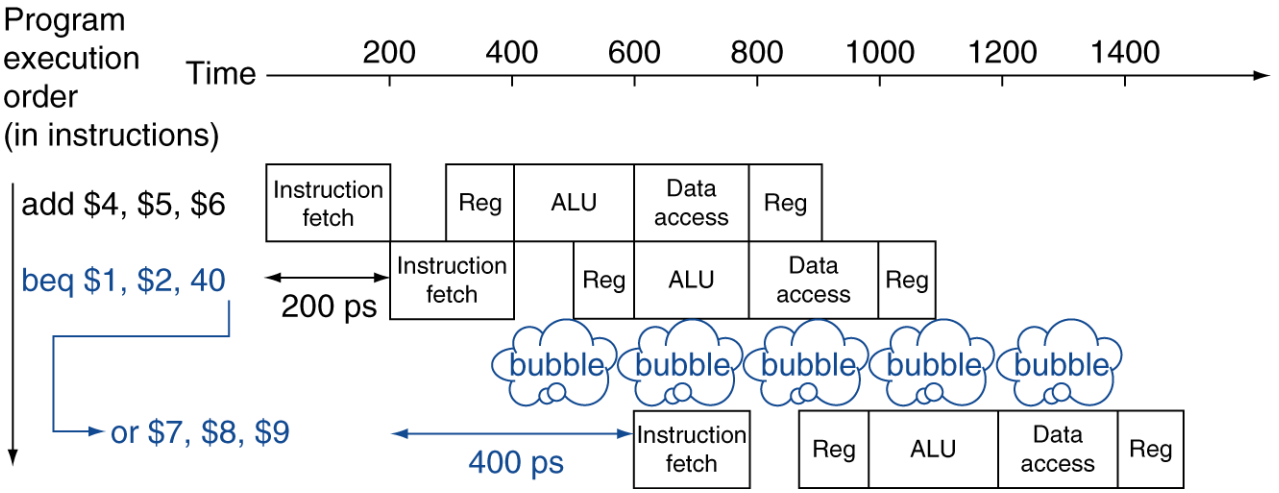
If beq condition = true, this instruction should be fetched (branch taken)

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 (we will see more)
  - 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

Pipeline Summary

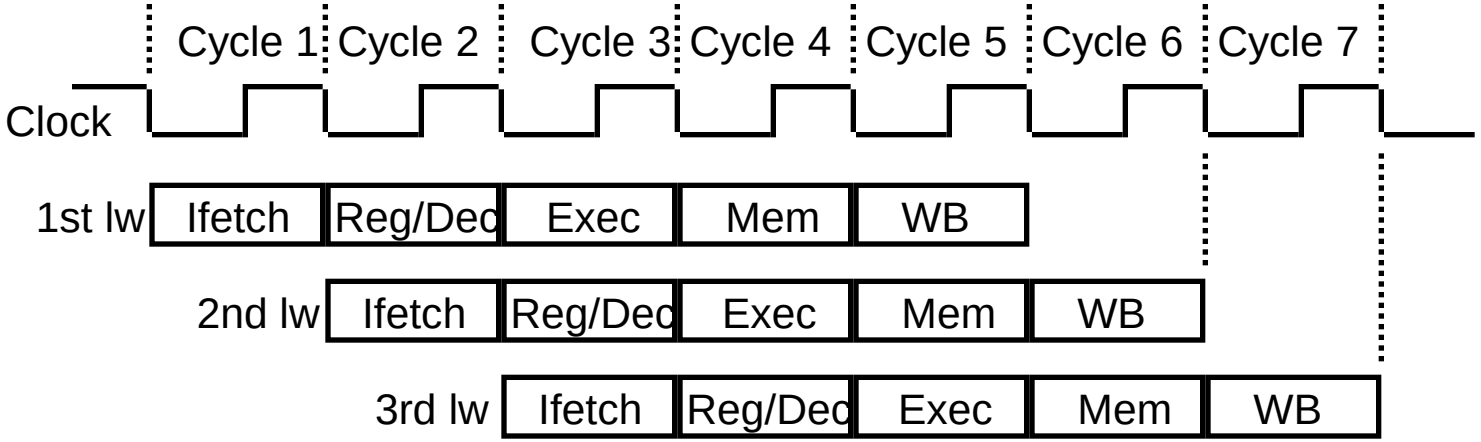
The BIG Picture

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

Review

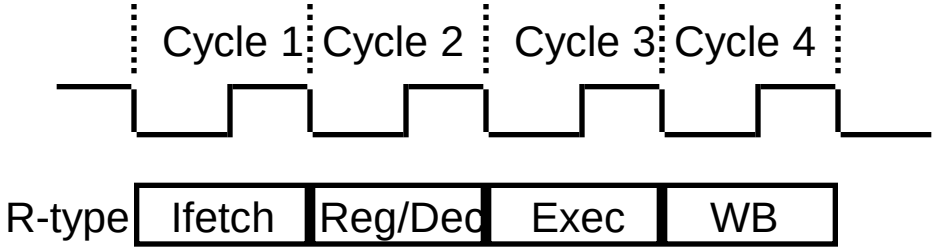
- Why do we keep all 5 stages in pipeline for all instructions?

Pipelining the Load Instruction



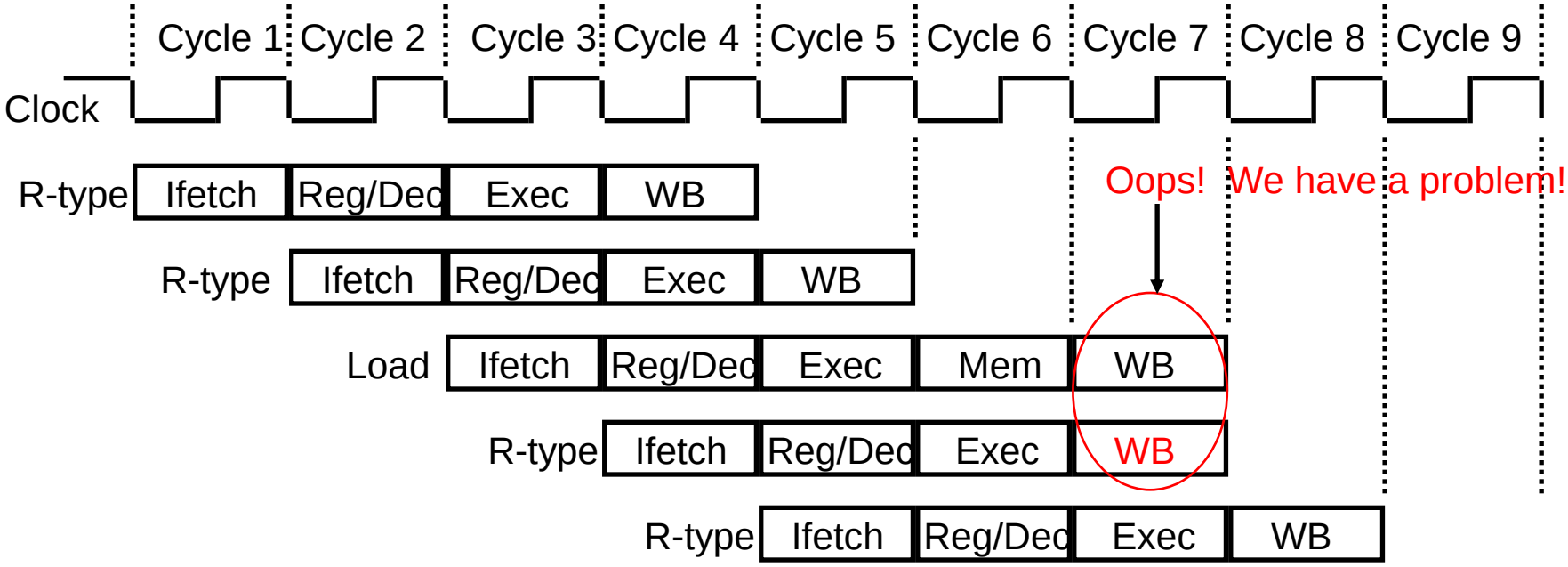
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the WB stage

The Four Stages of R-type



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
  - ALU operates on the two register operands
  - Update PC
- Wr: Write the ALU output back to the register file

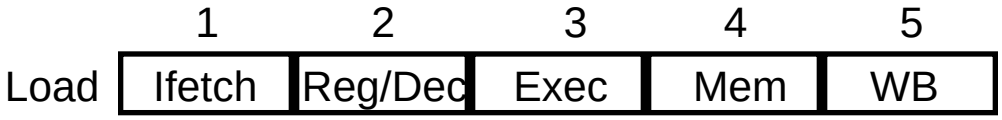
Pipelining the R-type and Load Instruction



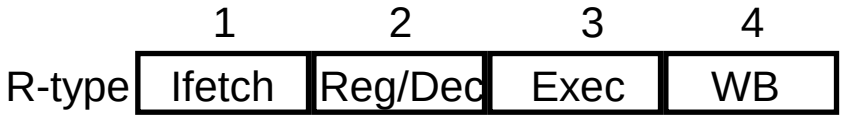
- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Important Observation

- First condition for pipeline to work: Each functional unit can only be used once per instruction (necessary but not sufficient condition)
- Second condition for pipeline to work: Each functional unit must be used at the same stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage

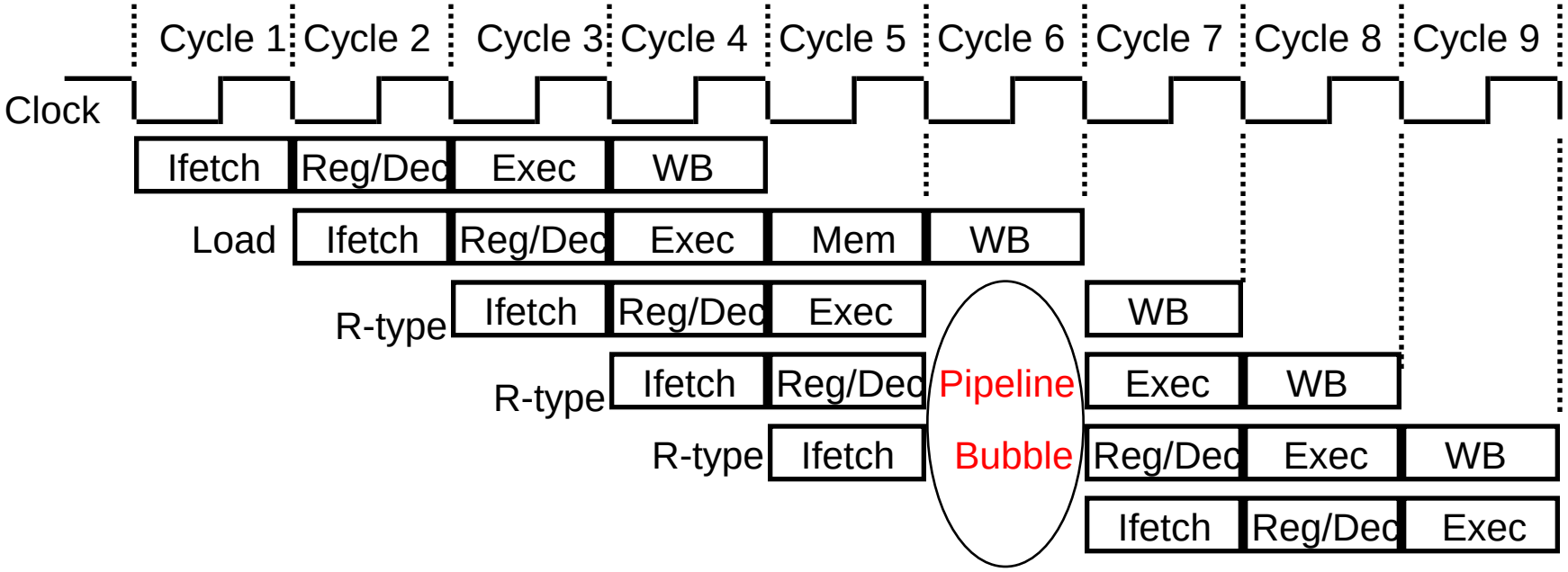


  - R-type uses Register File's Write Port during its 4th stage



- 2 ways to solve the problem.

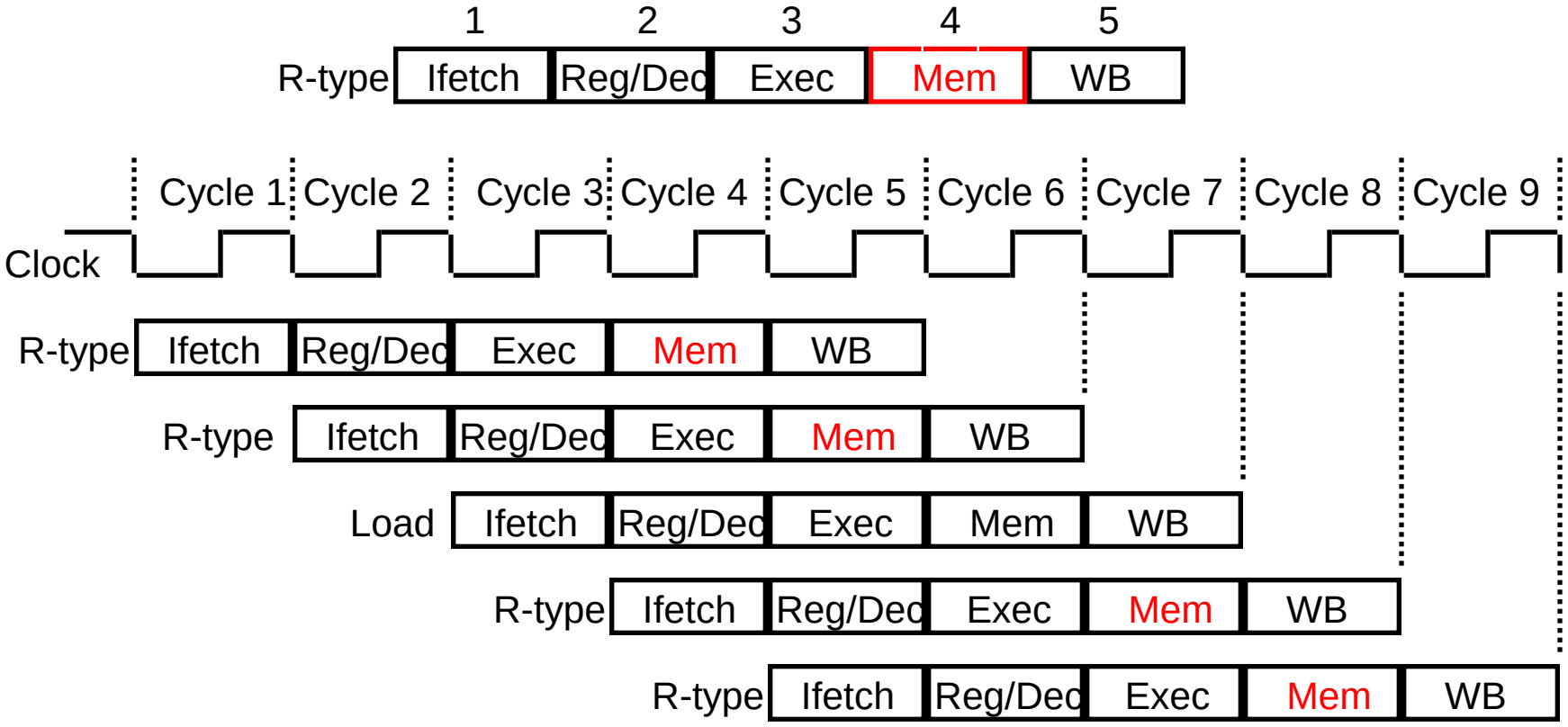
Solution 1: Insert "Bubble" into the Pipeline



- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - We lose one cycle due to bubble
  - A mix of load and R-type instructions will have an effective CPI > 1 instead of 1 (because we will not be able to finish one instruction per cycle in the steady state)

Solution 2: Delay R-type's Write by One Cycle

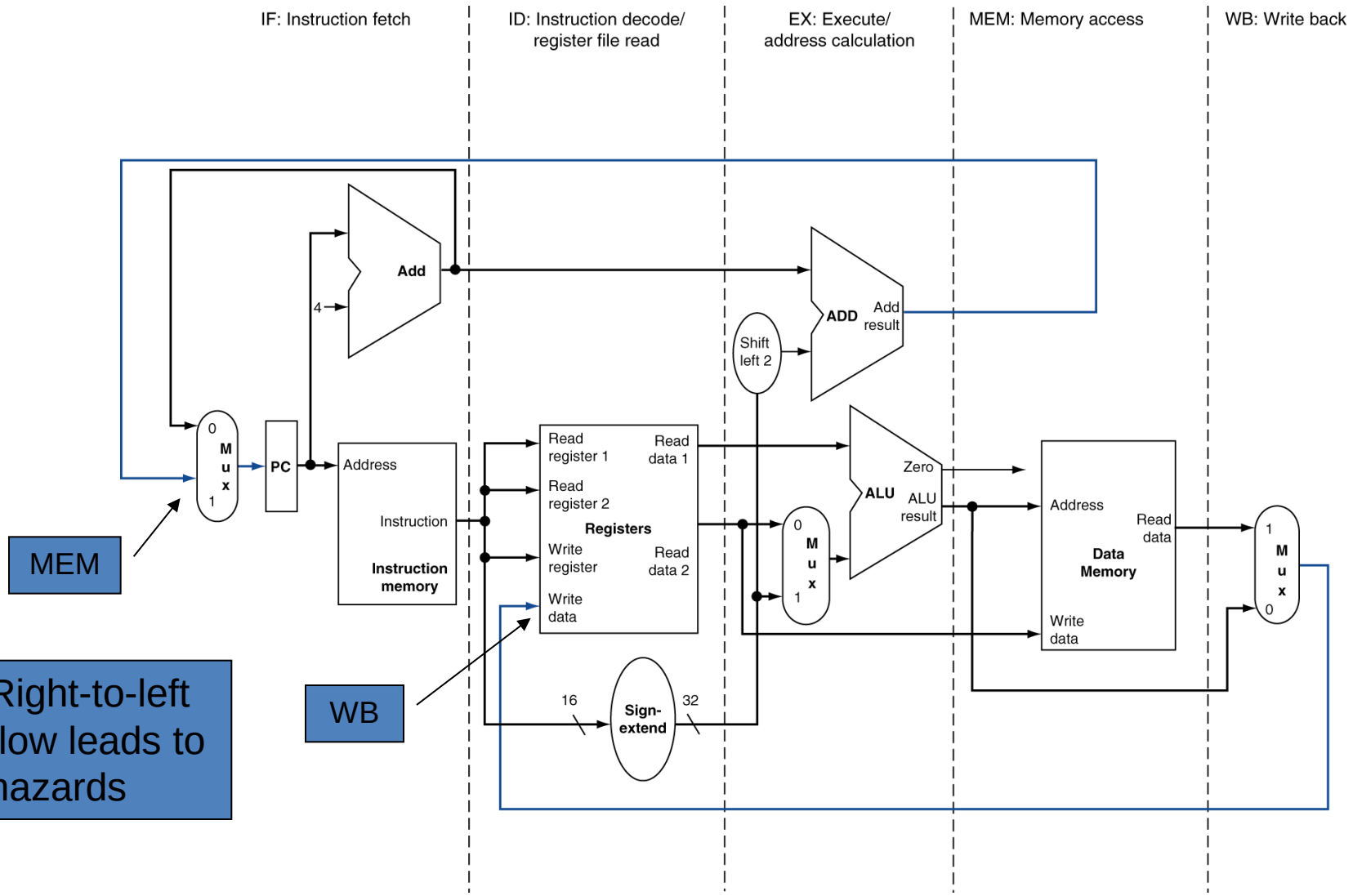
- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.



Solution 2: Delay R-type's Write by One Cycle (cont)

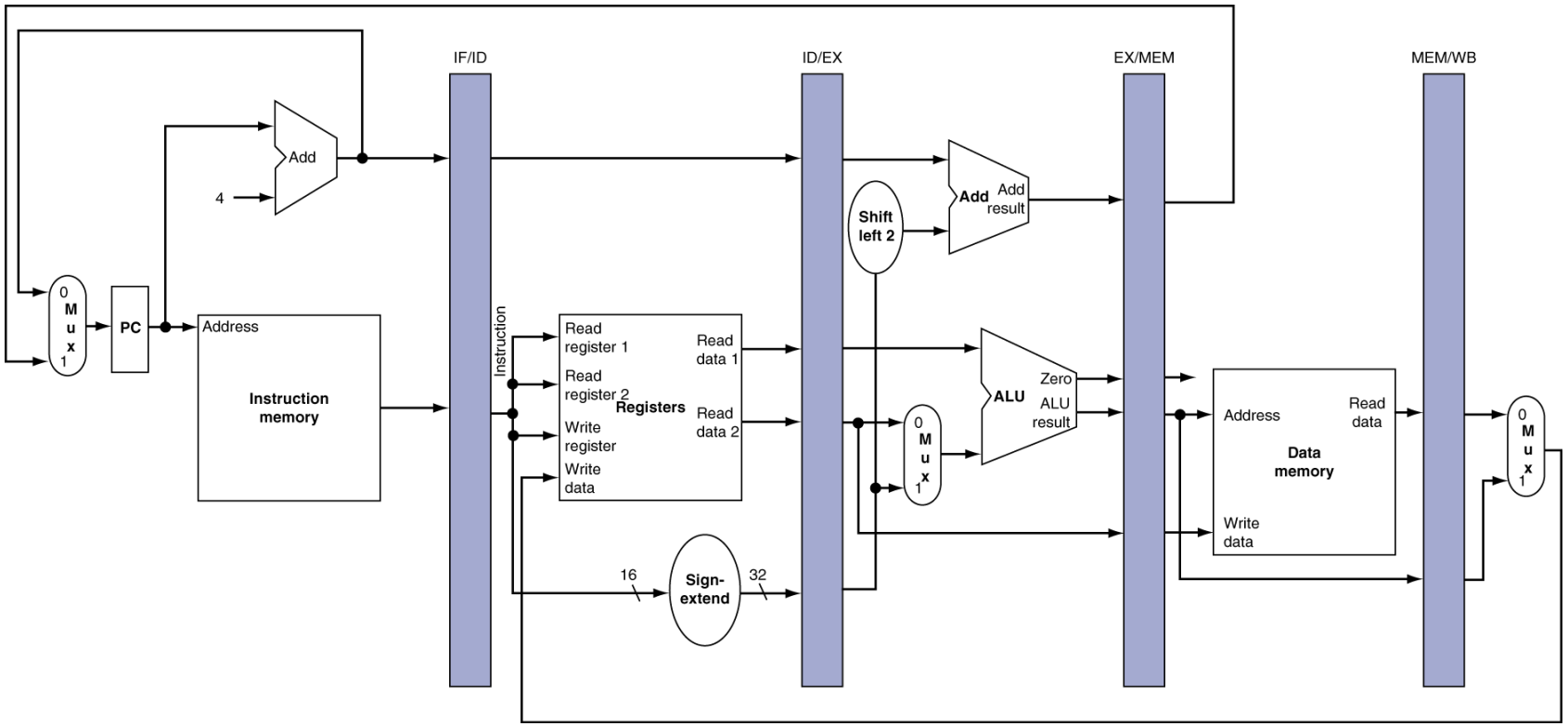
- Delay R-type's register write by one cycle:
  - Much simpler solution as far as control circuit is concerned.
  - Better solution as far as performance: back to completing one instruction per cycle
  - By adding 1 extra cycle to R-type instructions, we pay a price in individual R-type instructions, but the overall performance of the system improves.
  - The reason is that we have a more efficient pipeline.

MIPS Pipelined Datapath revisited (start with single cycle data path)



Pipeline registers

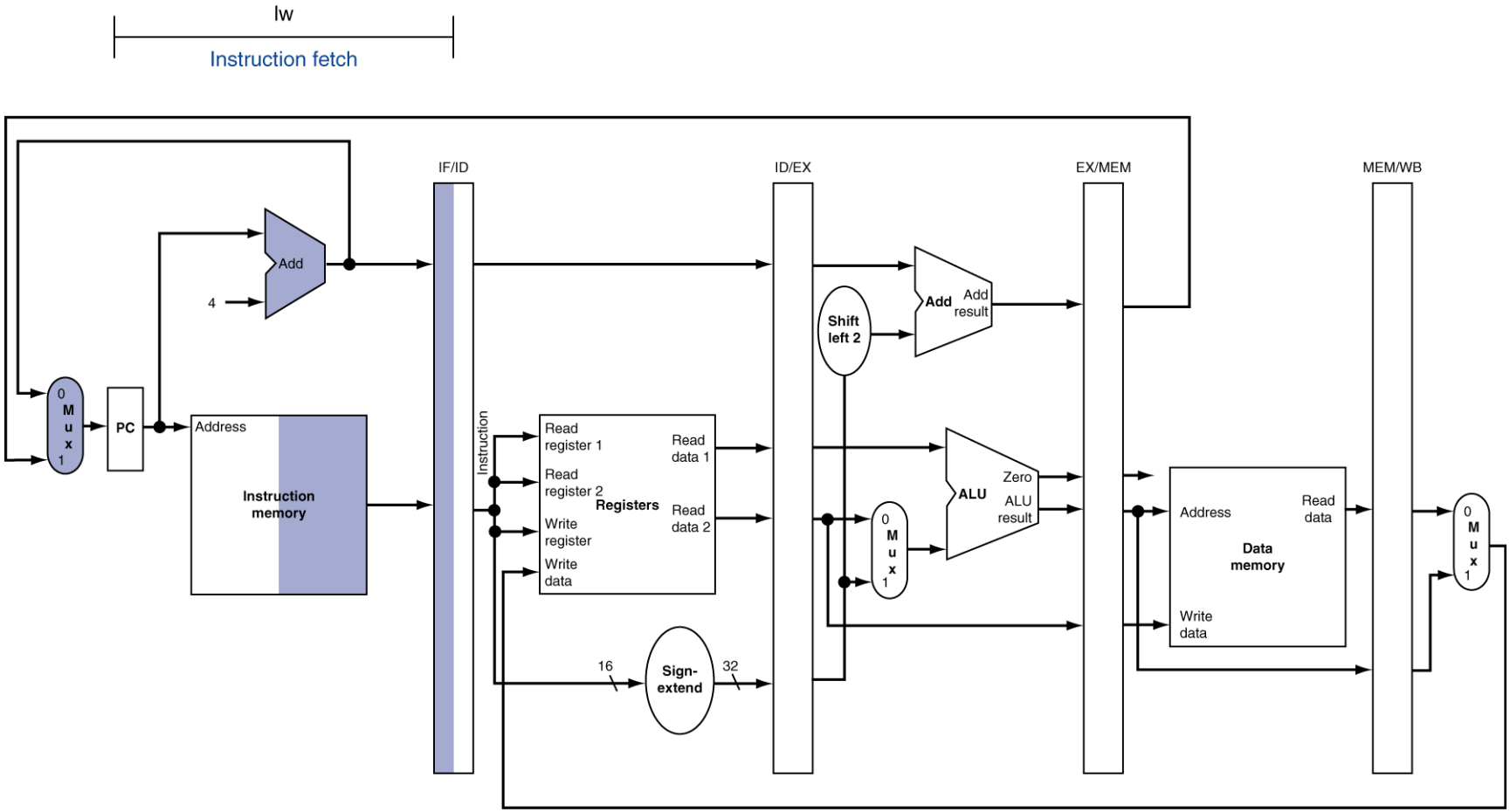
- Separate pipeline stages by inserting registers along the data path between stages.
  - To hold information produced in previous cycle



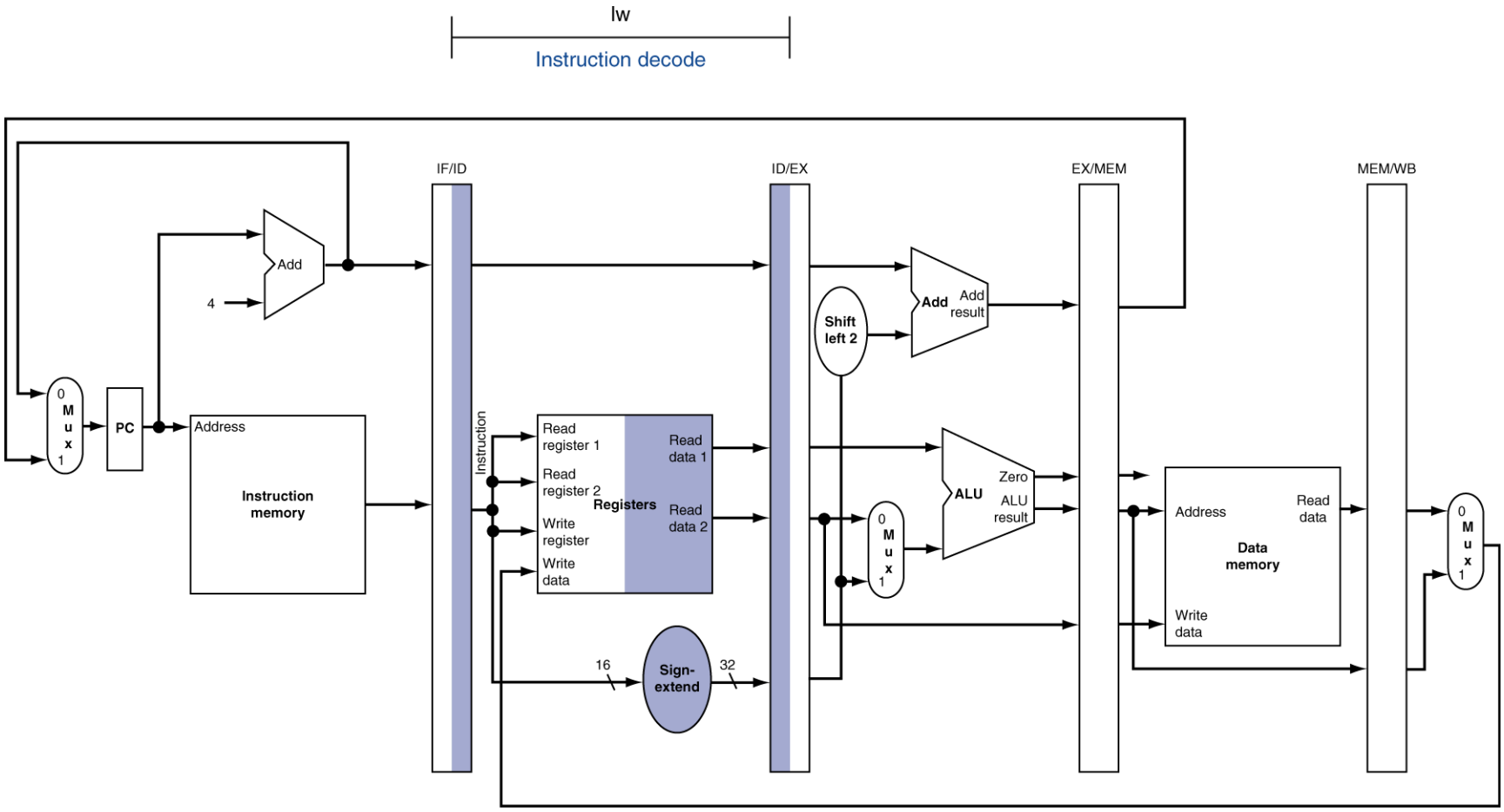
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

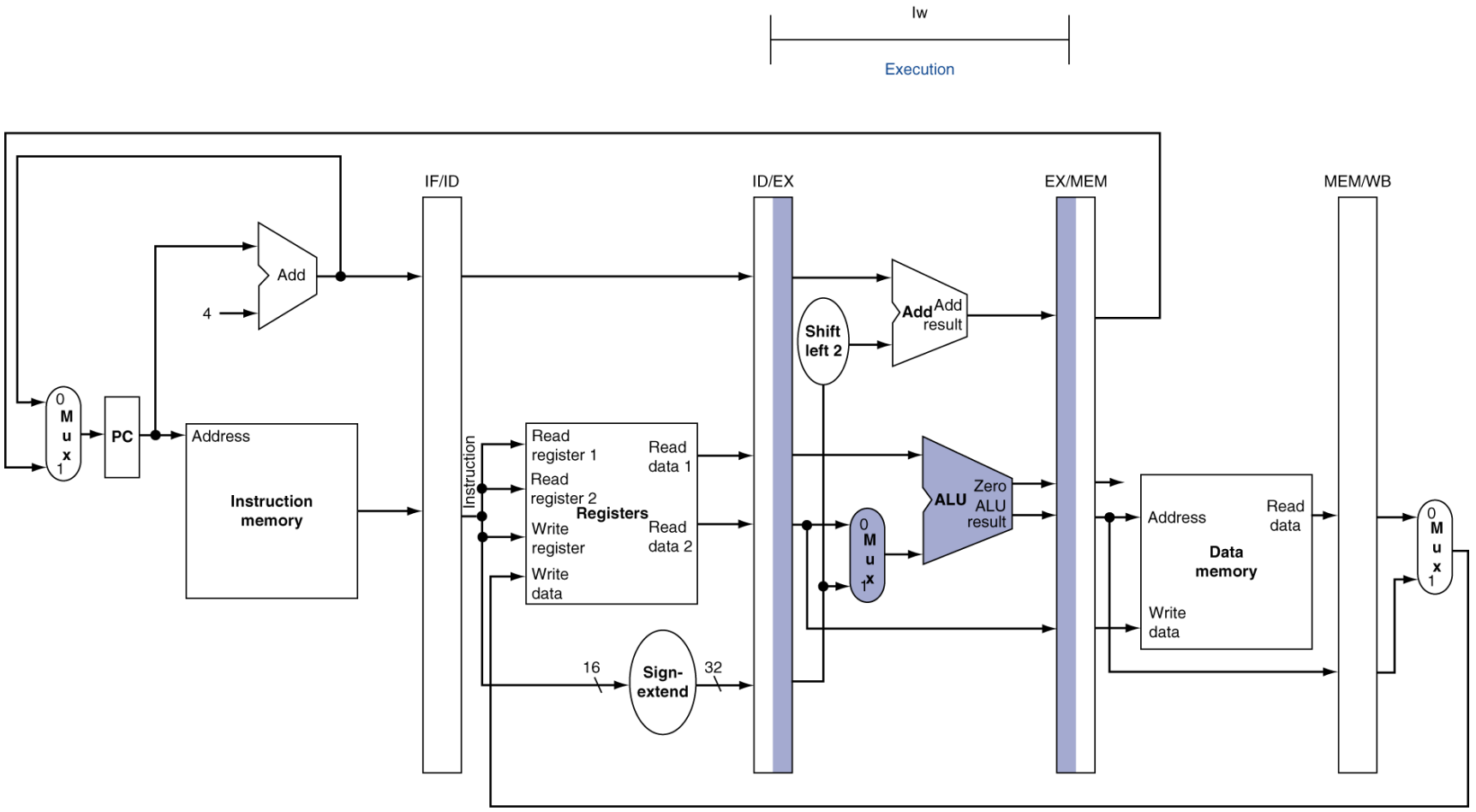
IF for Load, Store, …



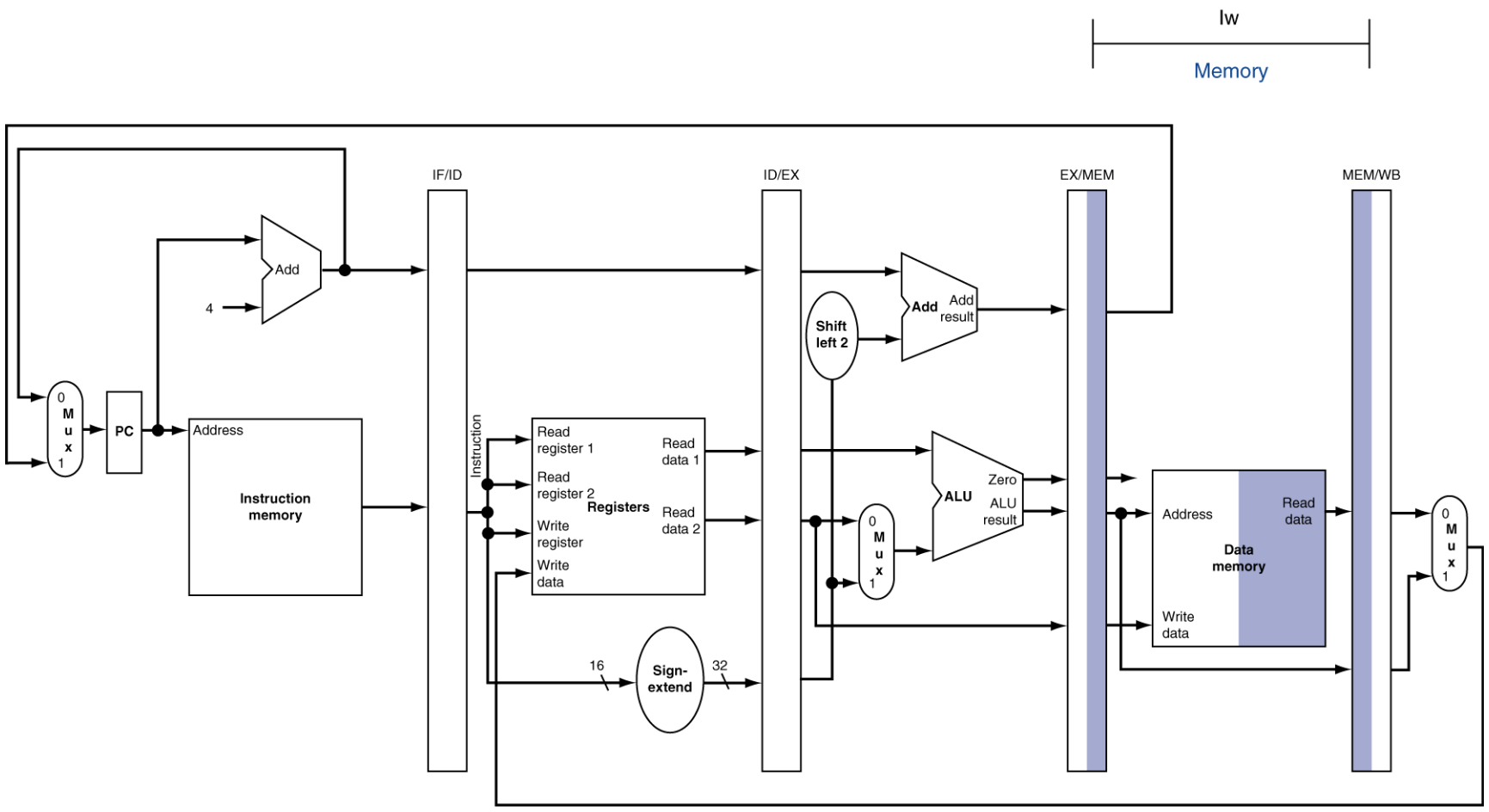
ID for Load, Store, …



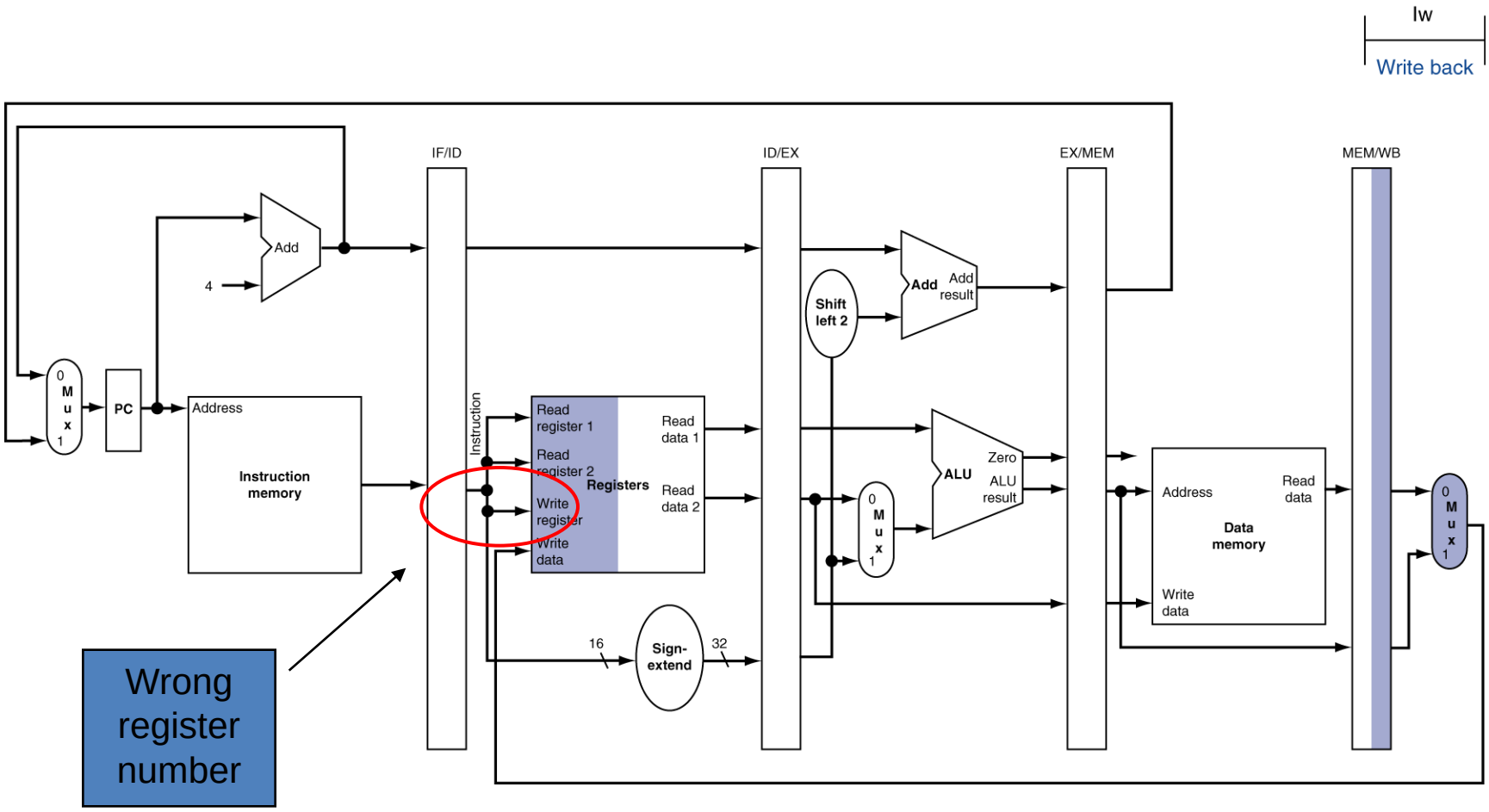
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32

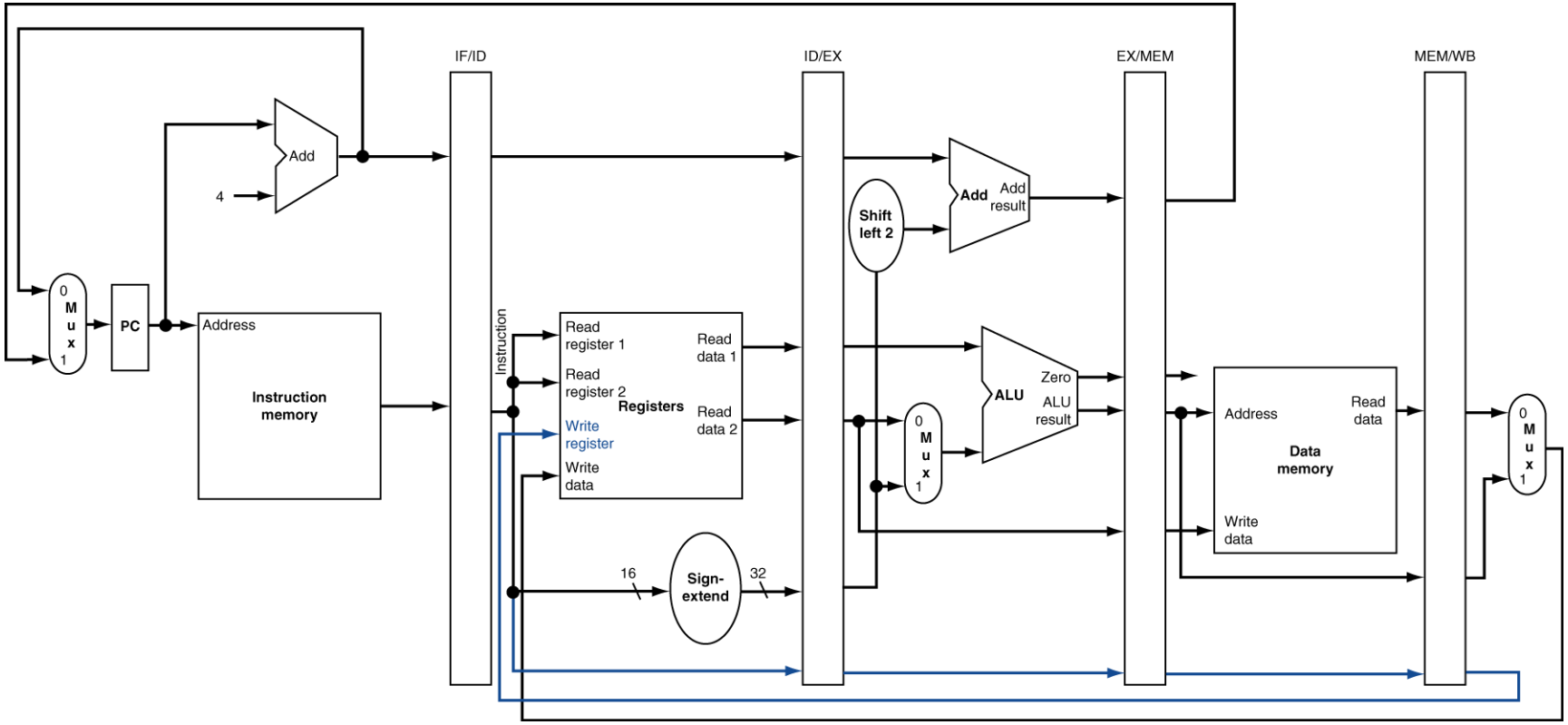
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

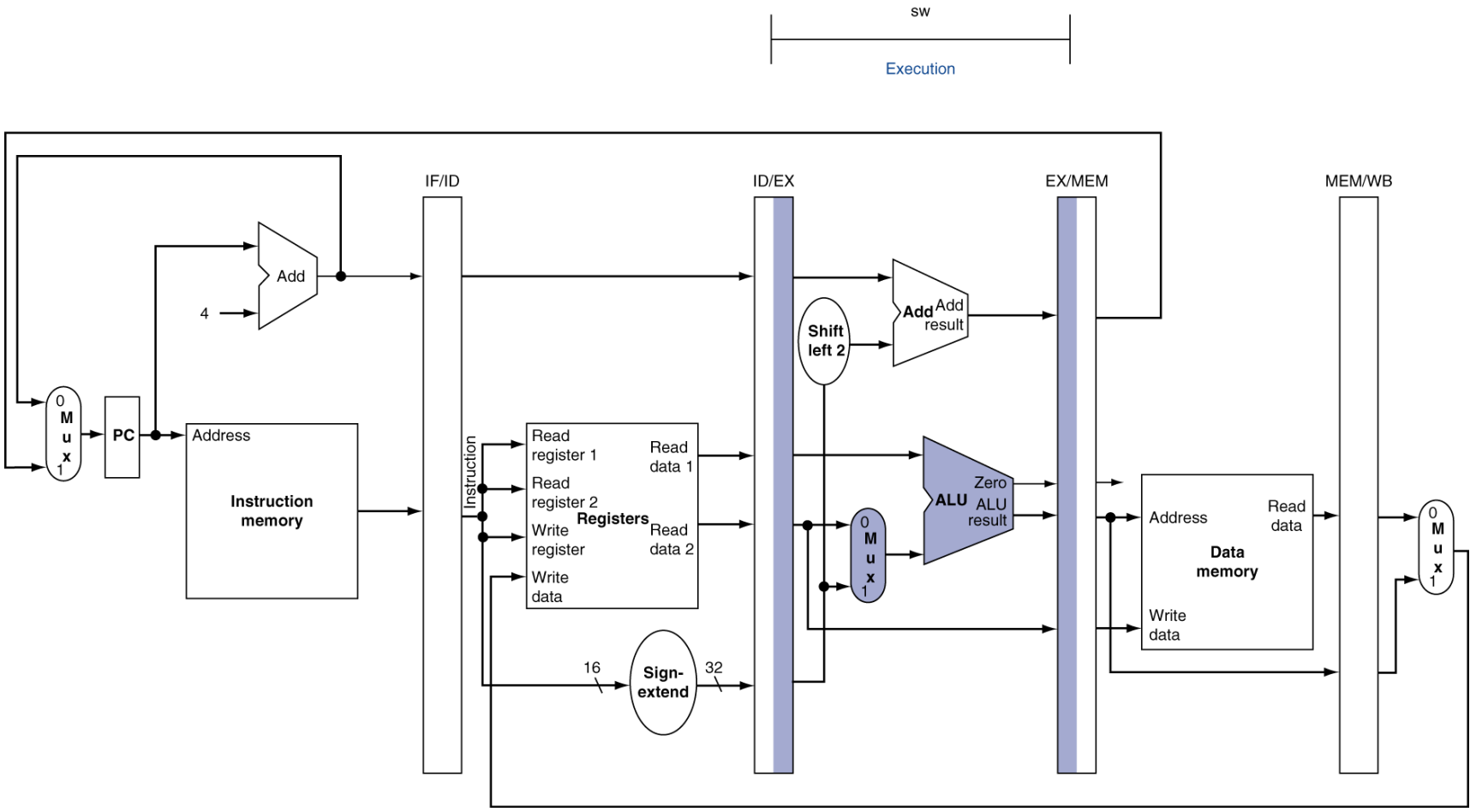
WB for Load


lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Wrong register number

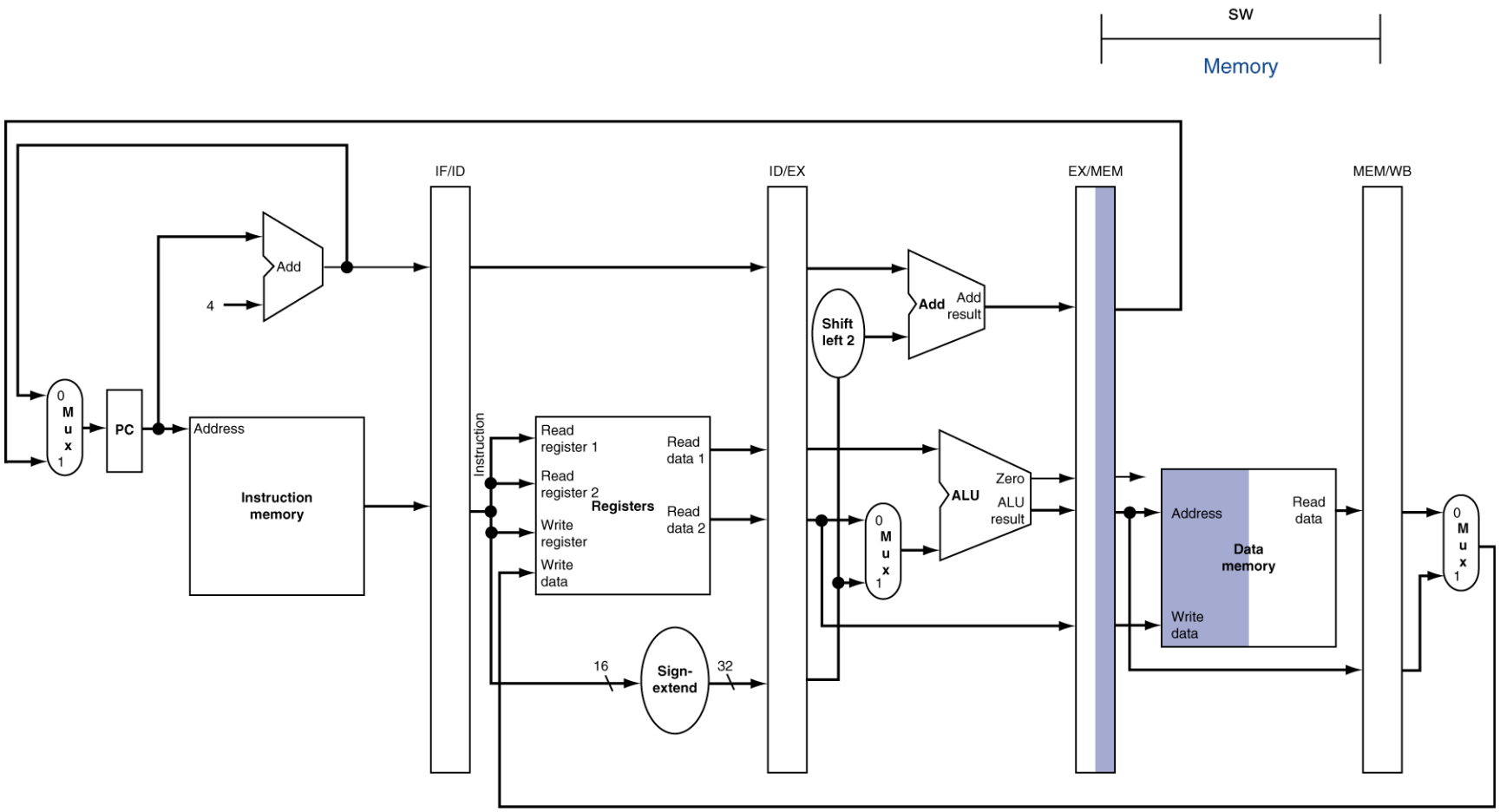
Corrected Datapath for Load



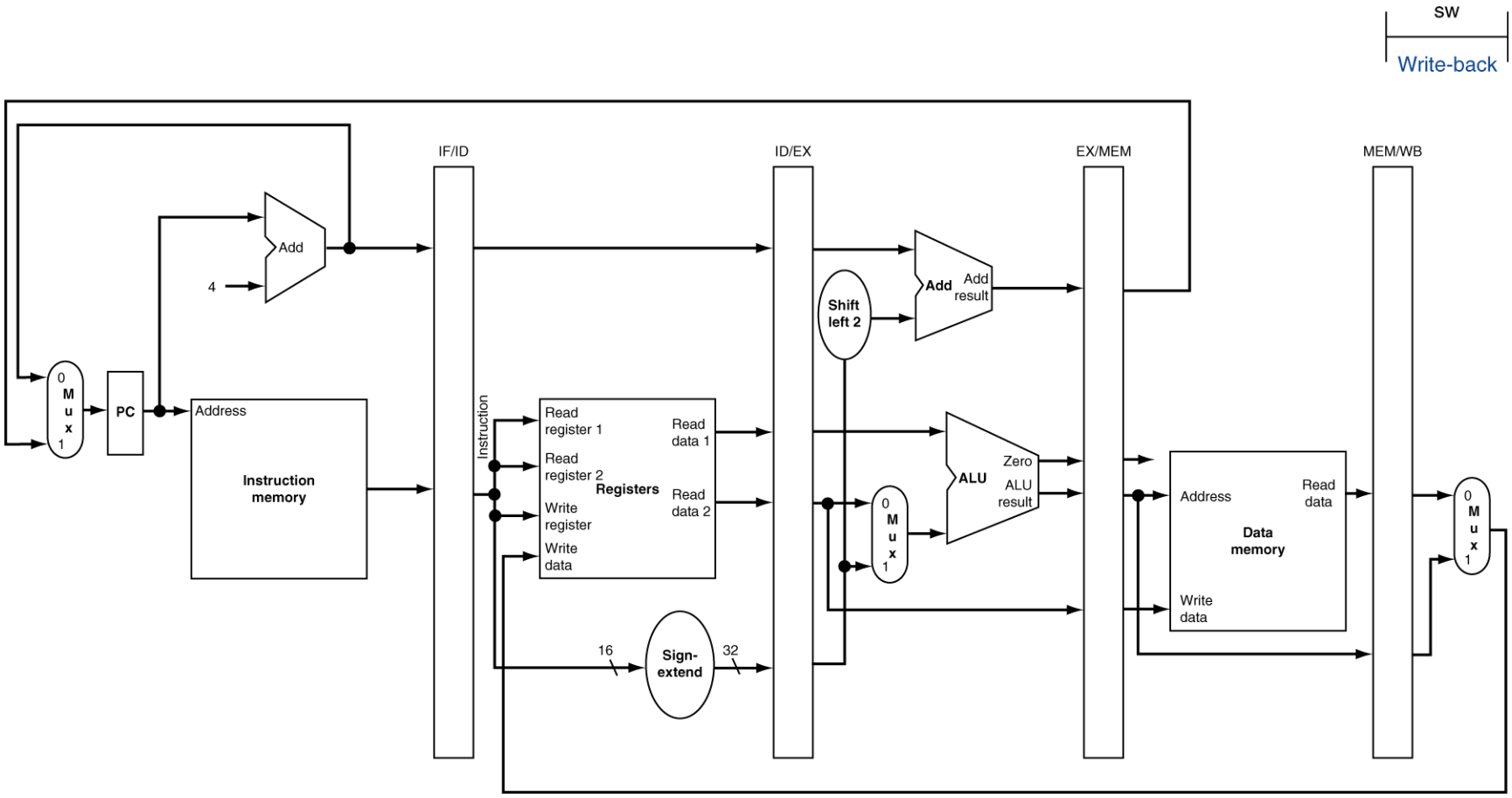
EX for Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

MEM for Store

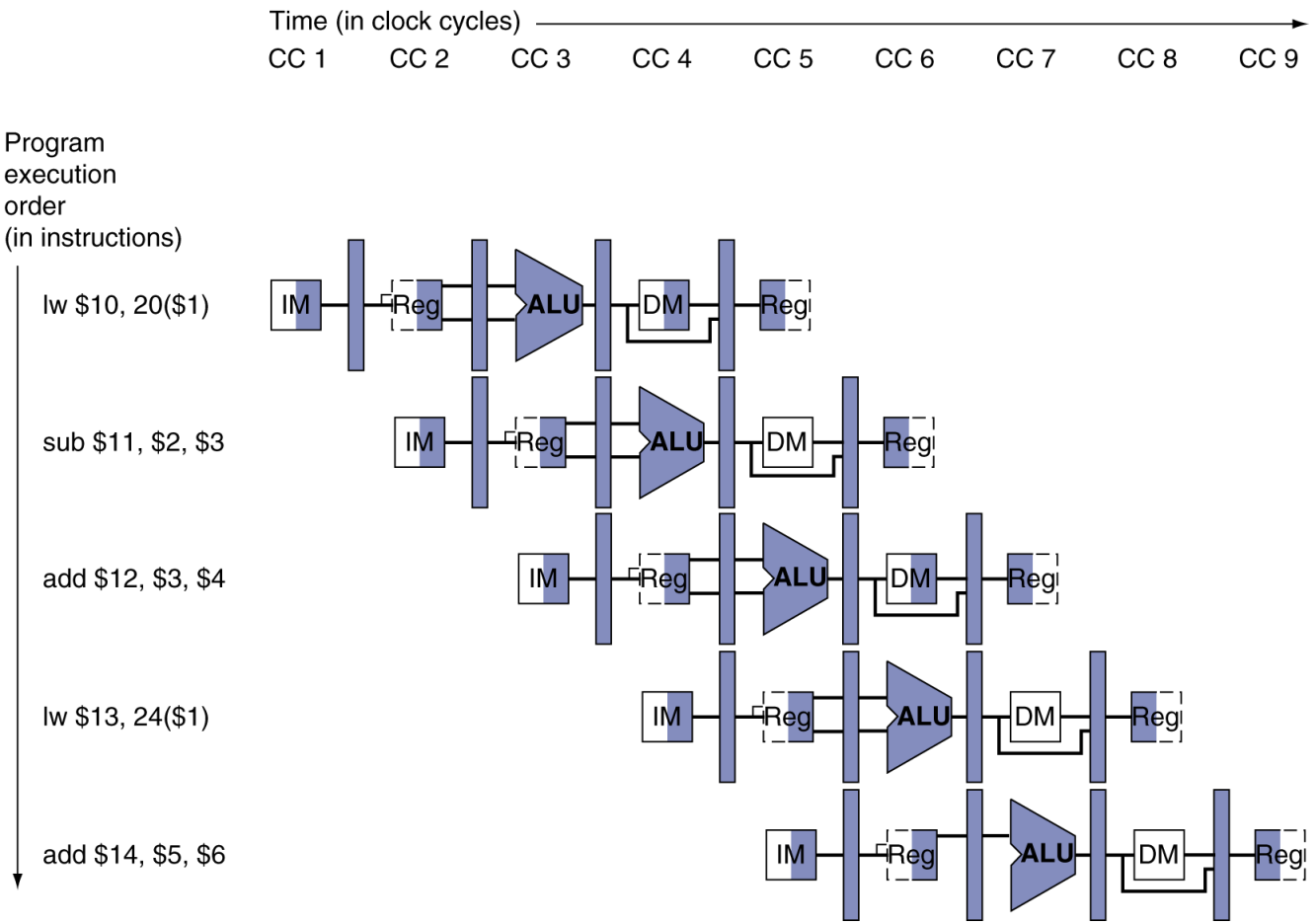


WB for Store


SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

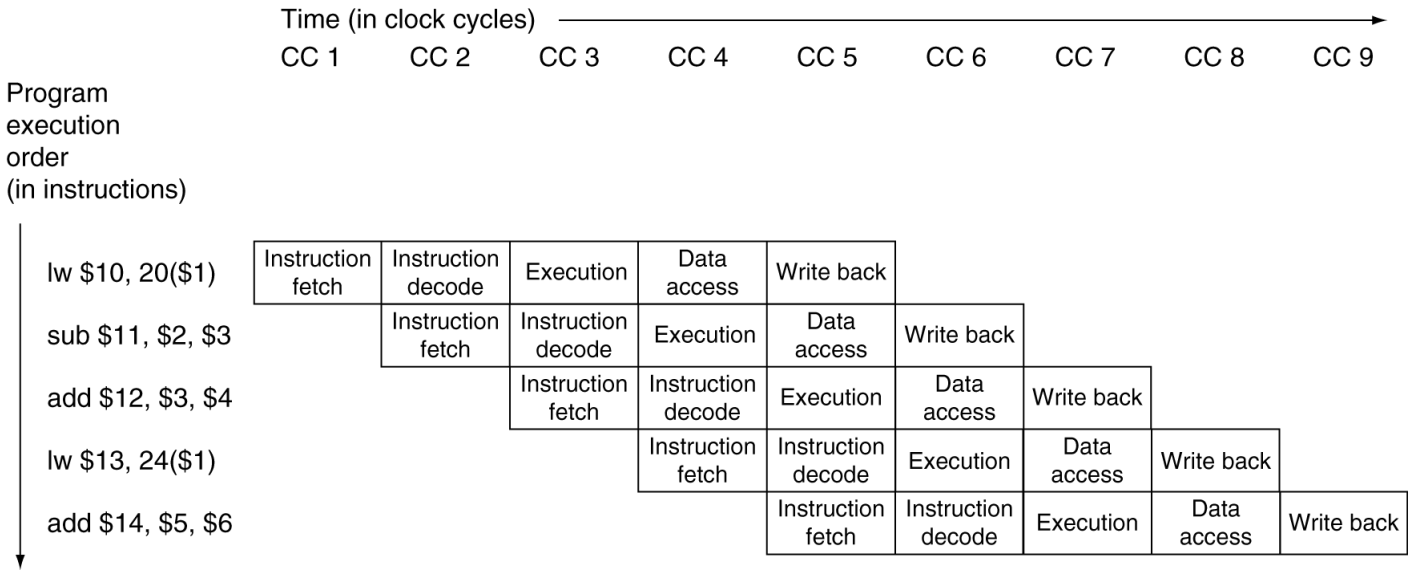
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

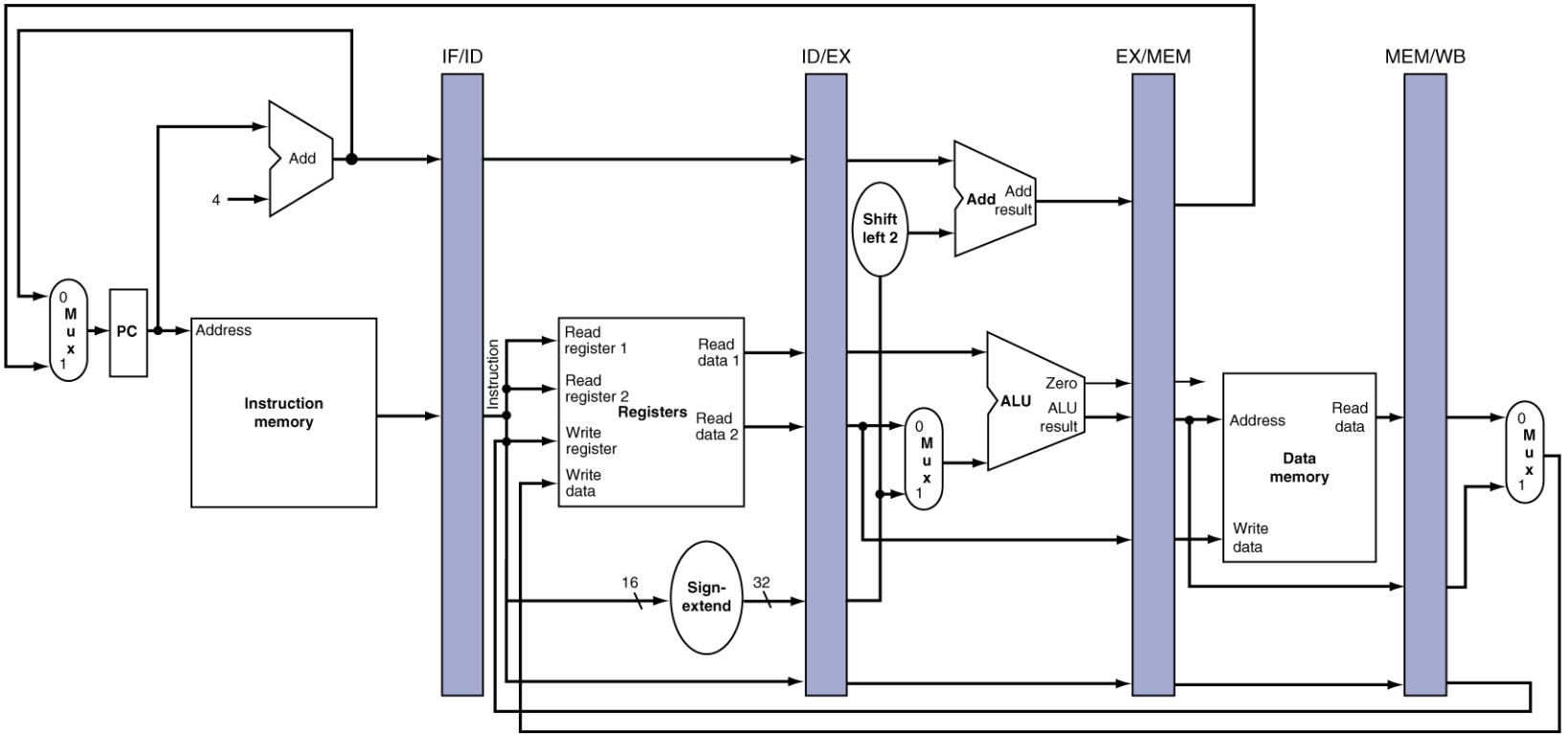
- Traditional form



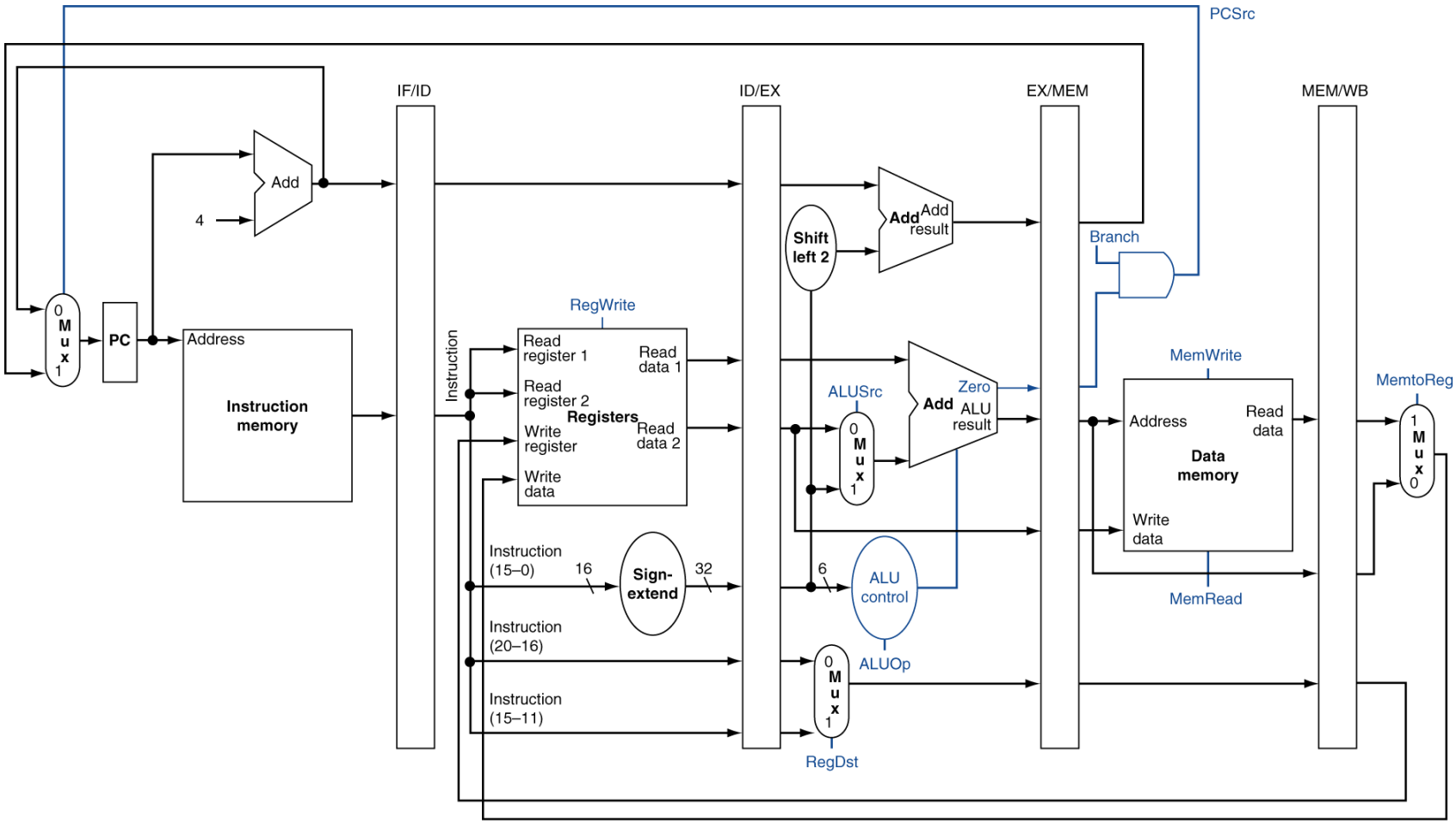
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

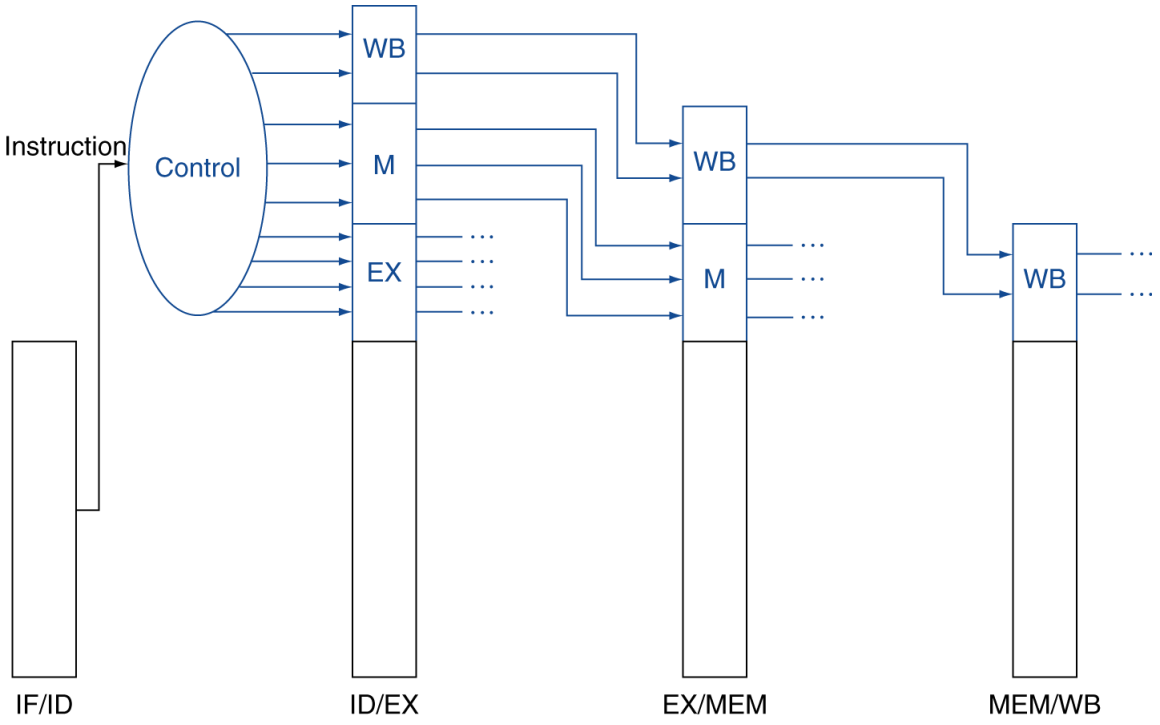


Pipelined Control (Simplified)

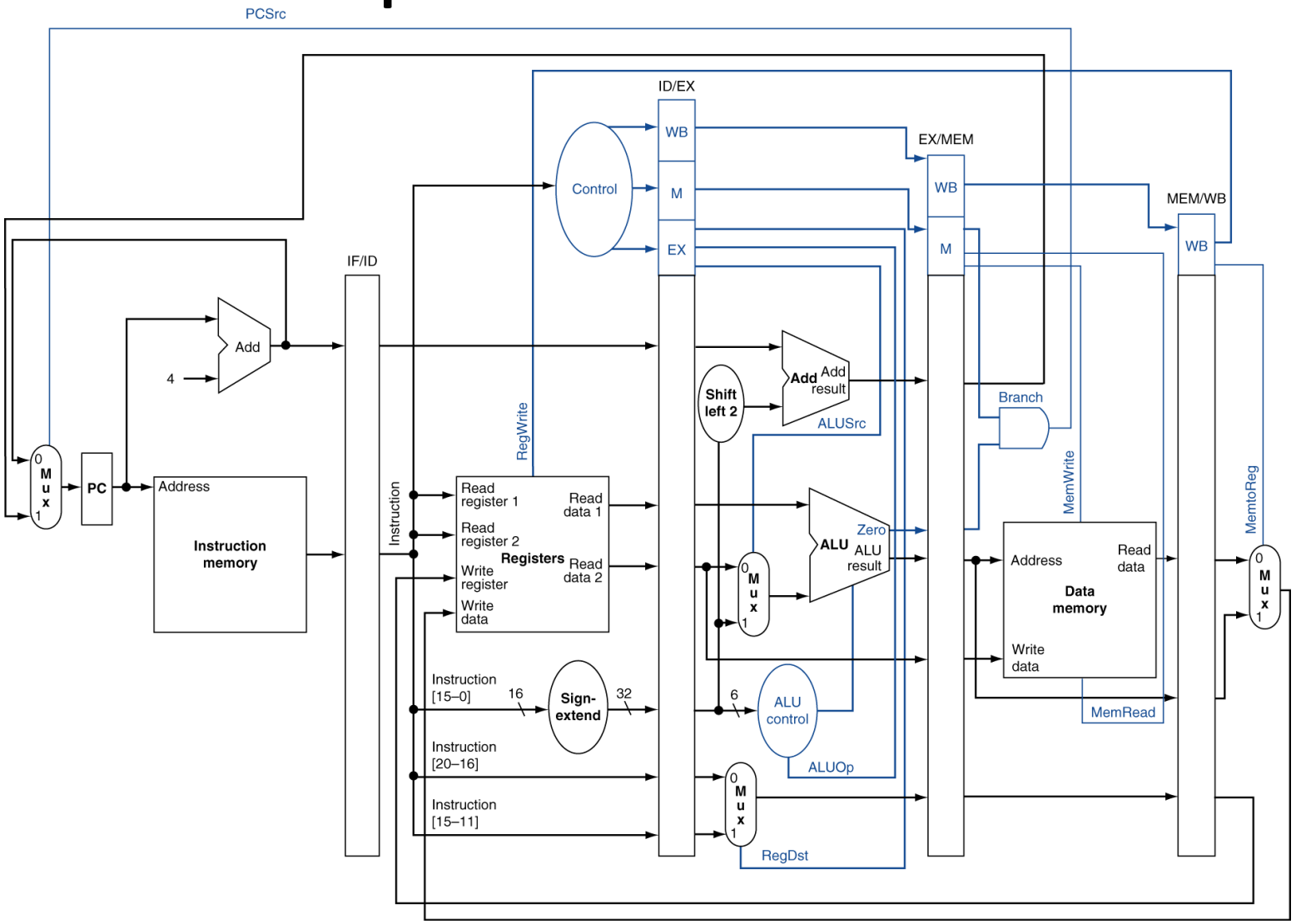


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



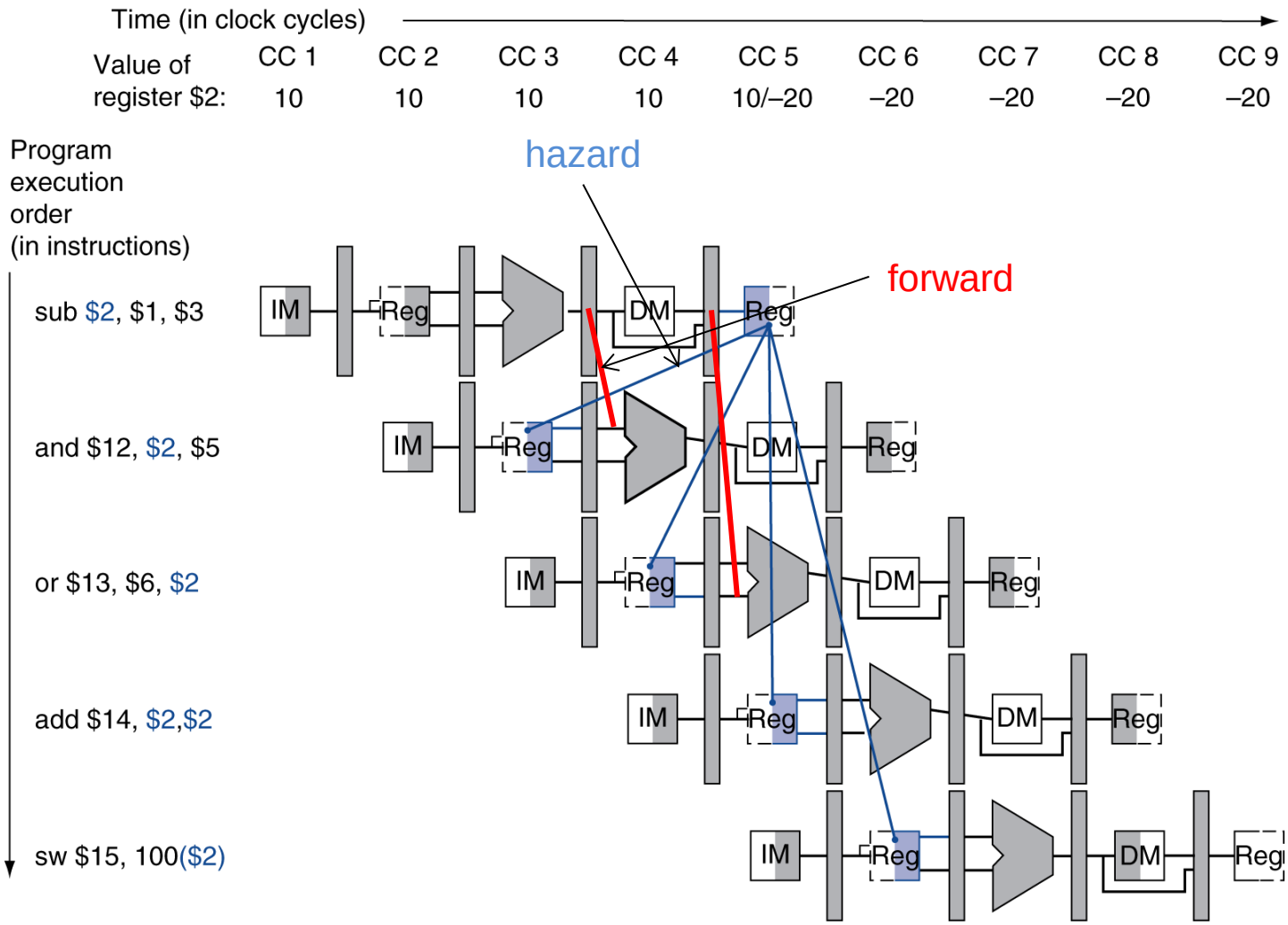
Data Hazards in ALU Instructions

- Consider this sequence:

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

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding (Bypassing)

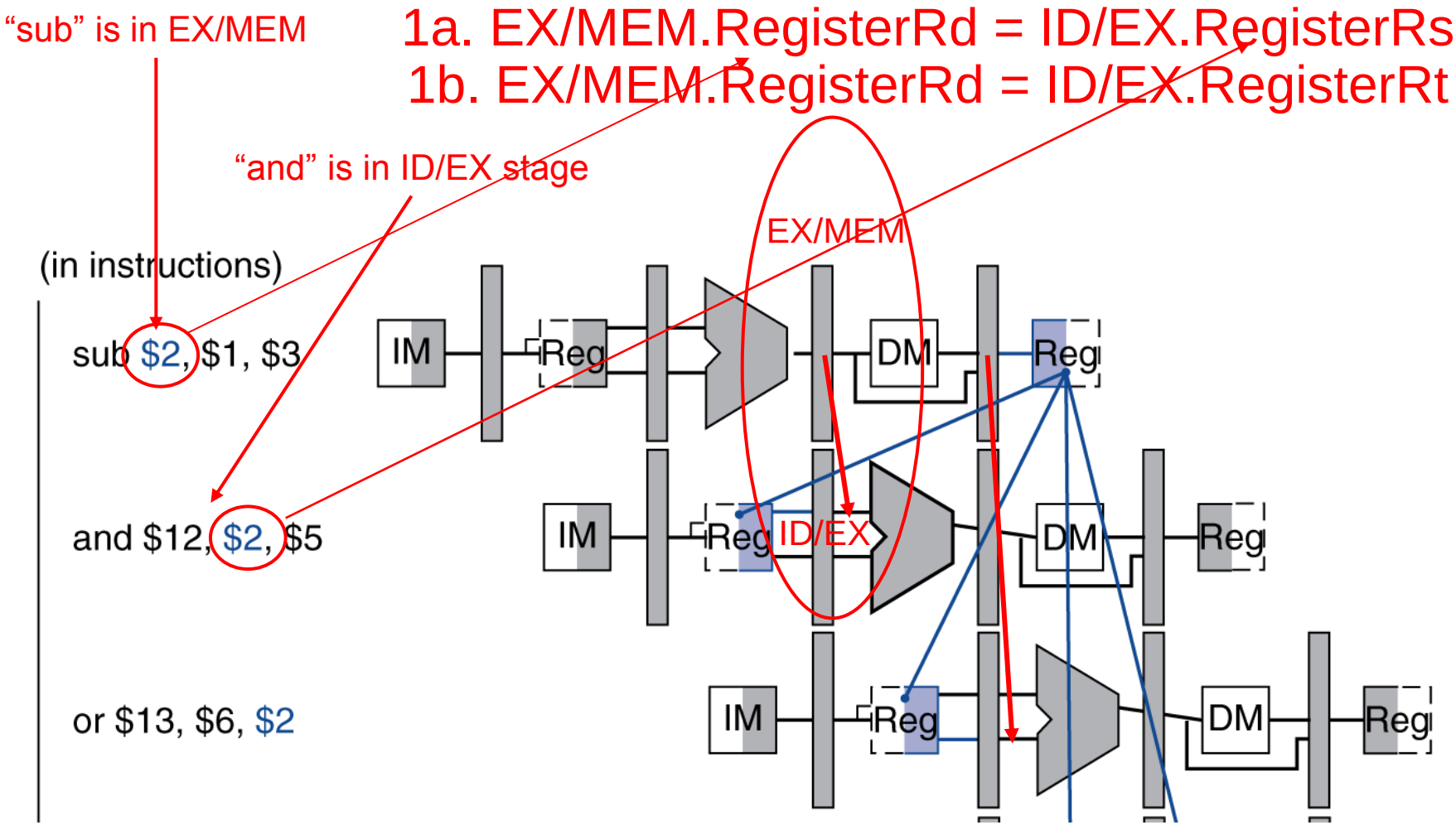


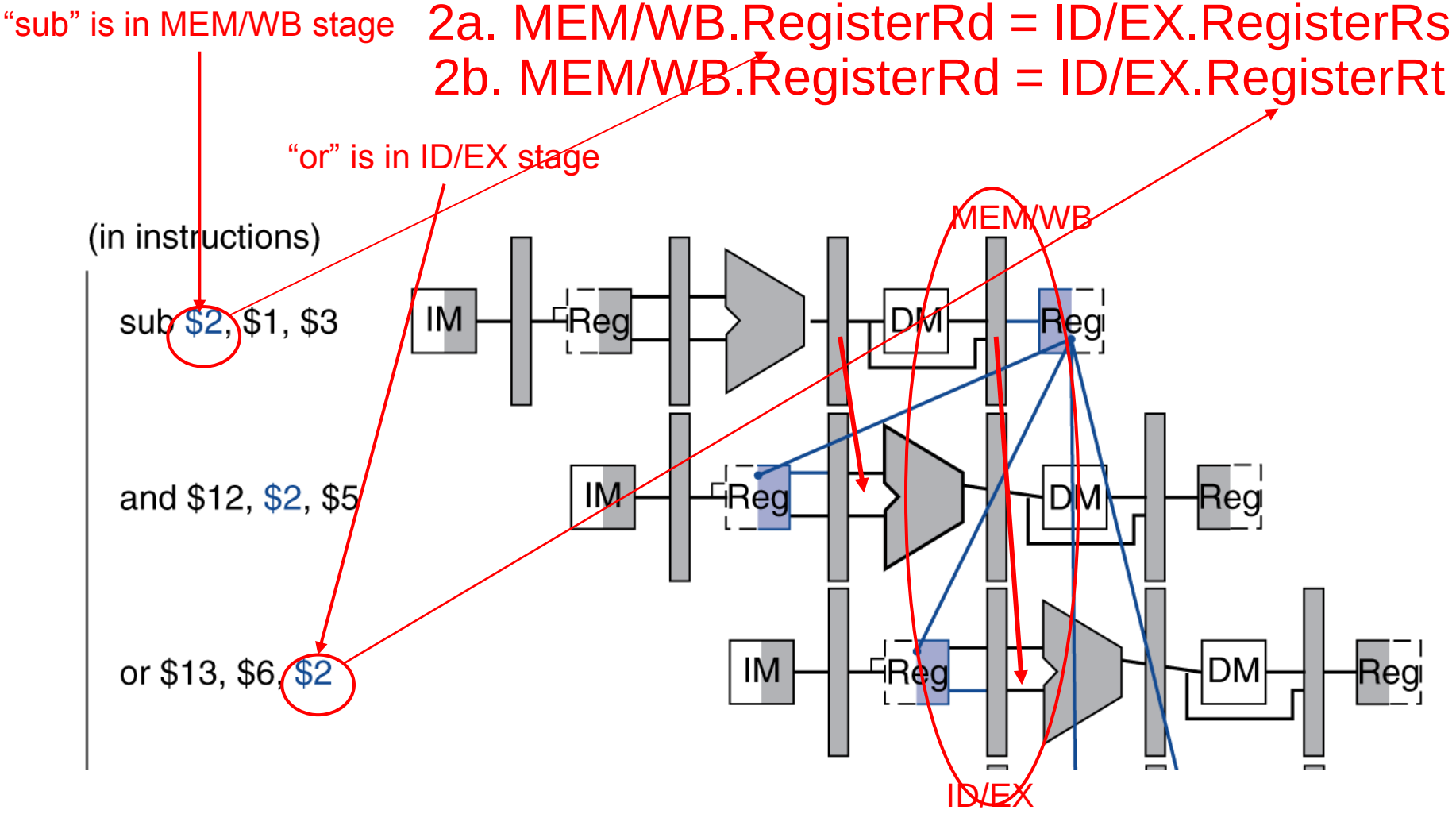
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

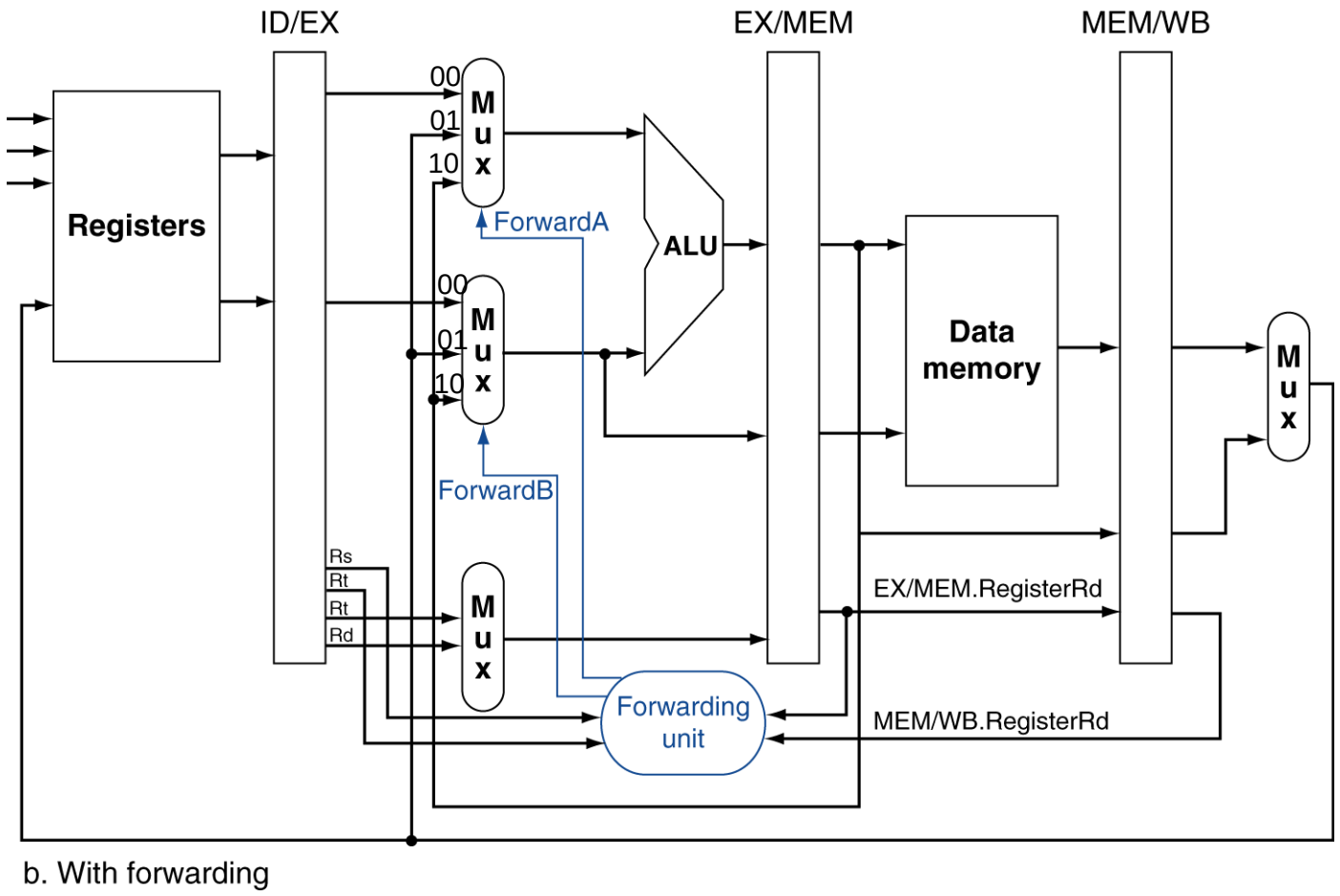

"sub" is in EX/MEM
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
"and" is in ID/EX stage
EX/MEM
(in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
IM
Reg
ID/EX
DM
Reg


"sub" is in MEM/WB stage
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
"or" is in ID/EX stage
(in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
IM
Reg
DM
Reg
MEM/WB
ID/EX

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Paths



b. With forwarding

Forwarding Conditions

- EX hazard
  - 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
- MEM hazard
  - 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

Double Data Hazard

- Consider the sequence:

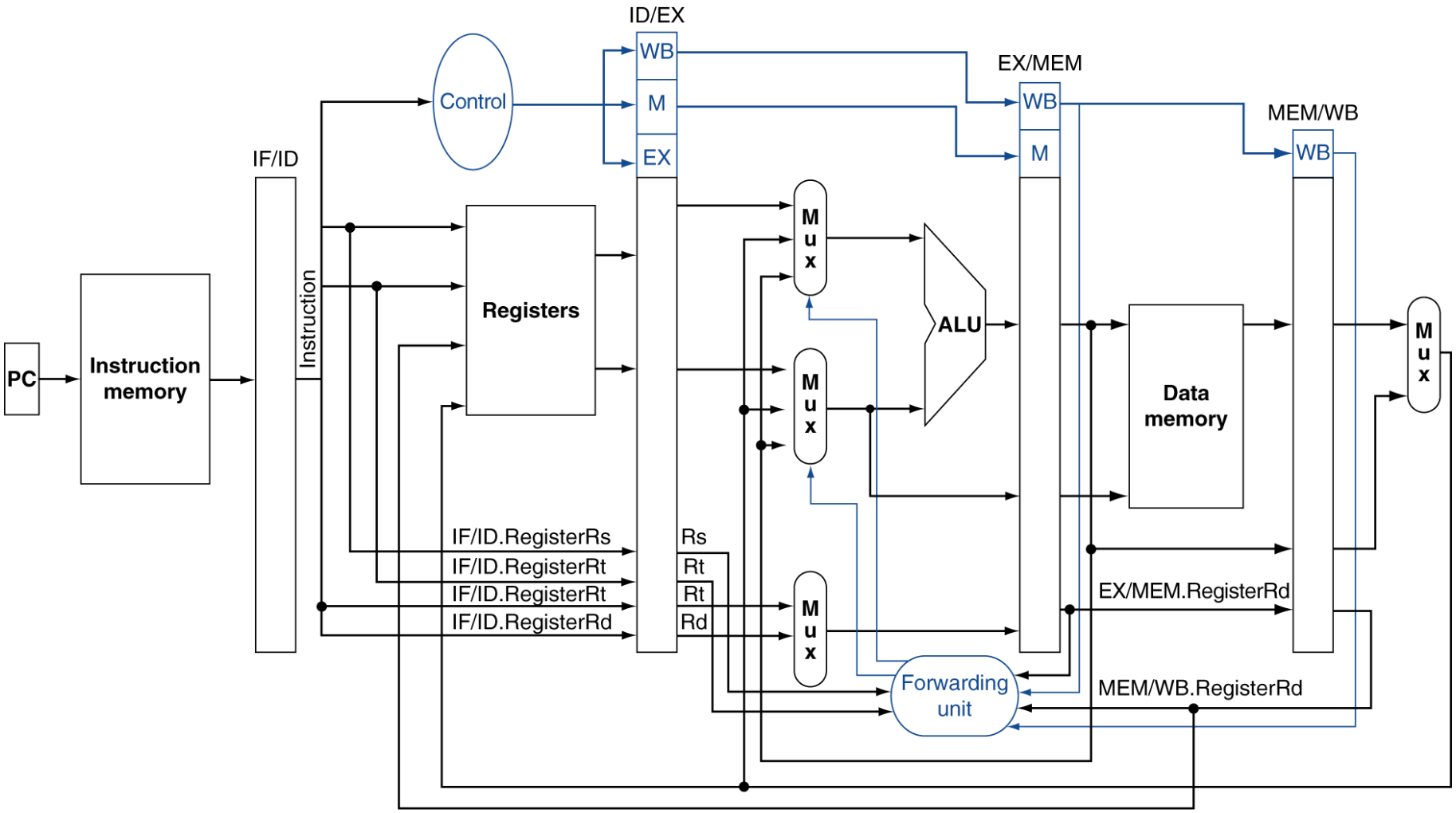
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

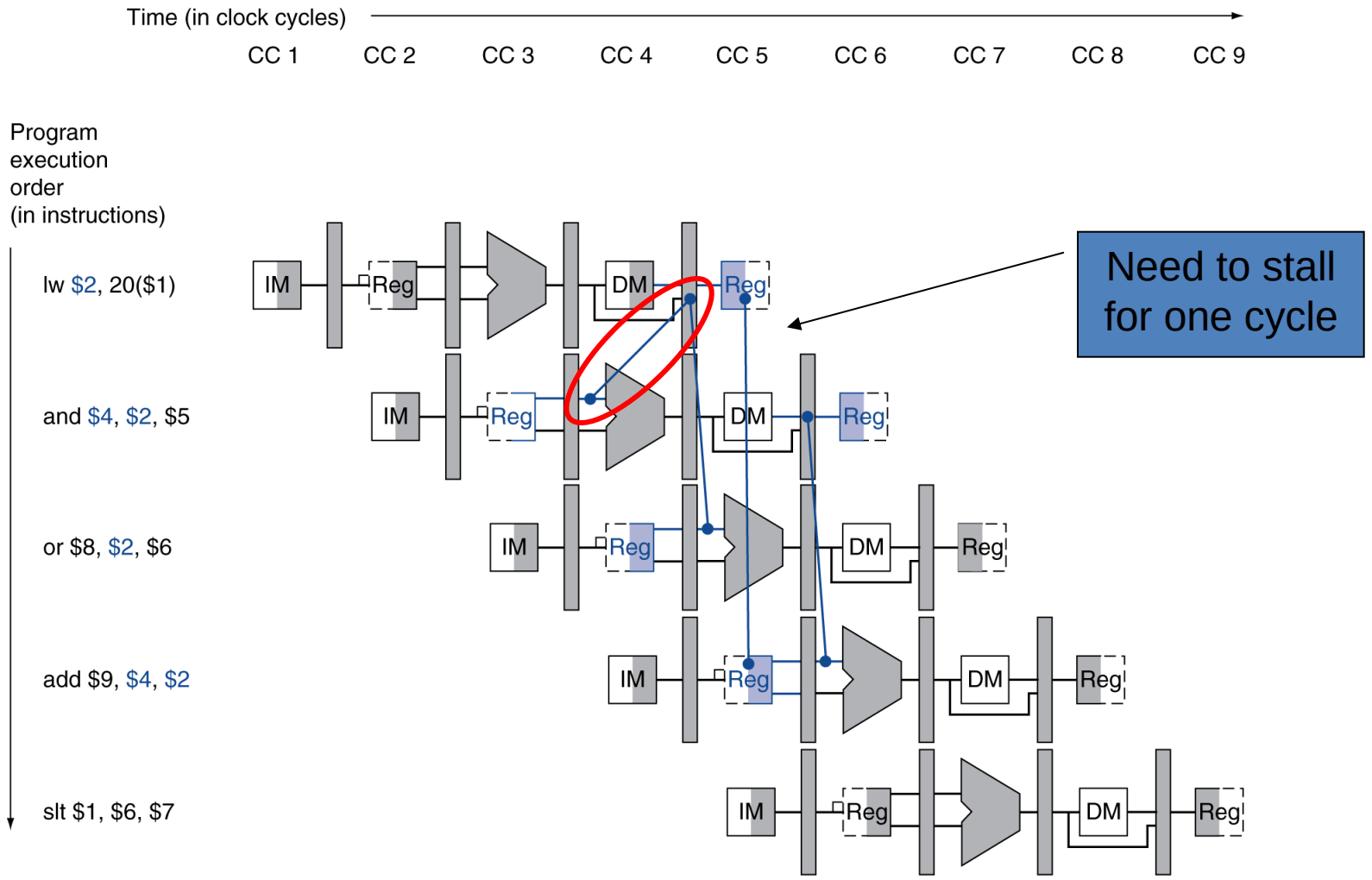
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.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 not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Datapath with Forwarding



Load-Use Data Hazard



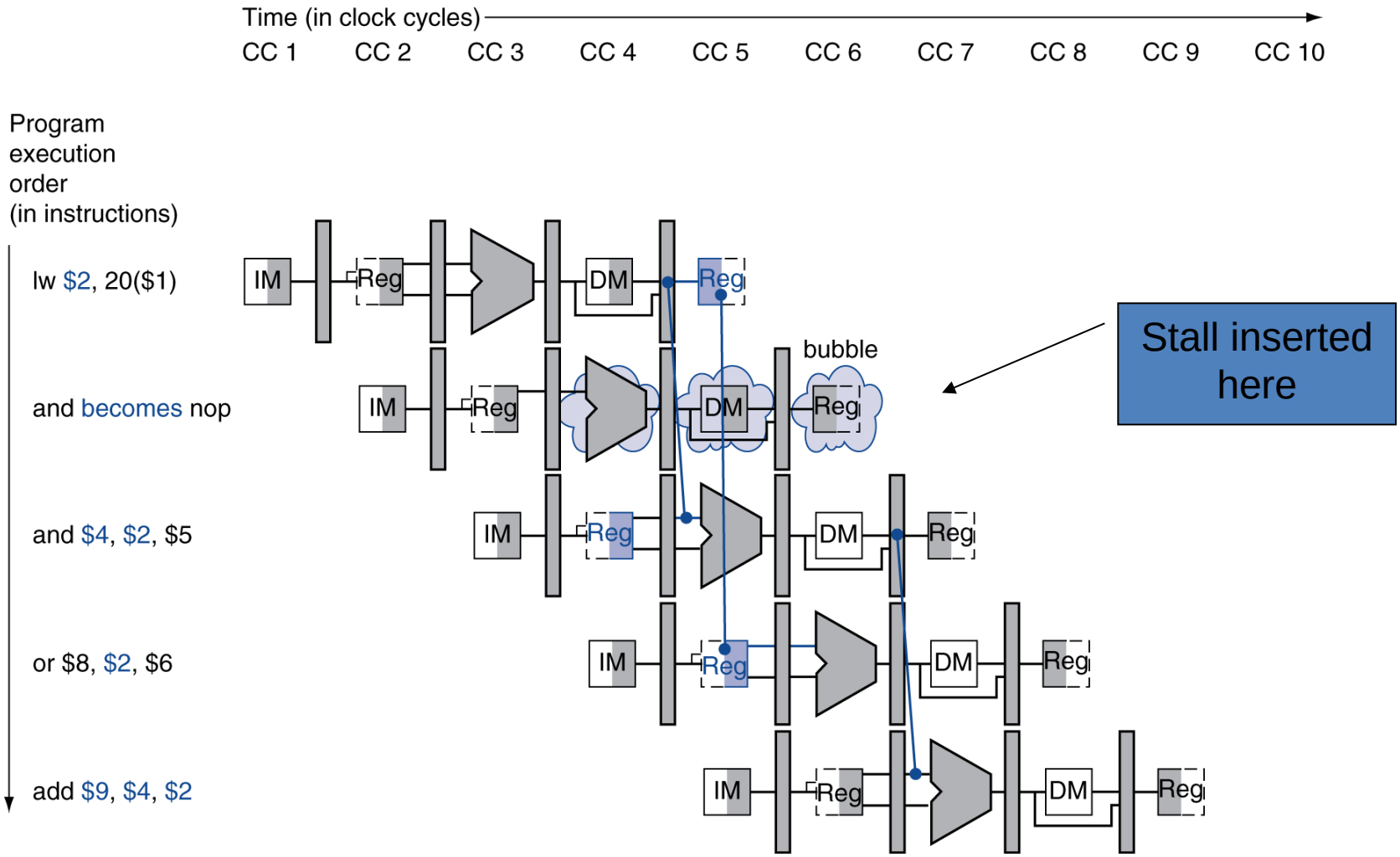
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

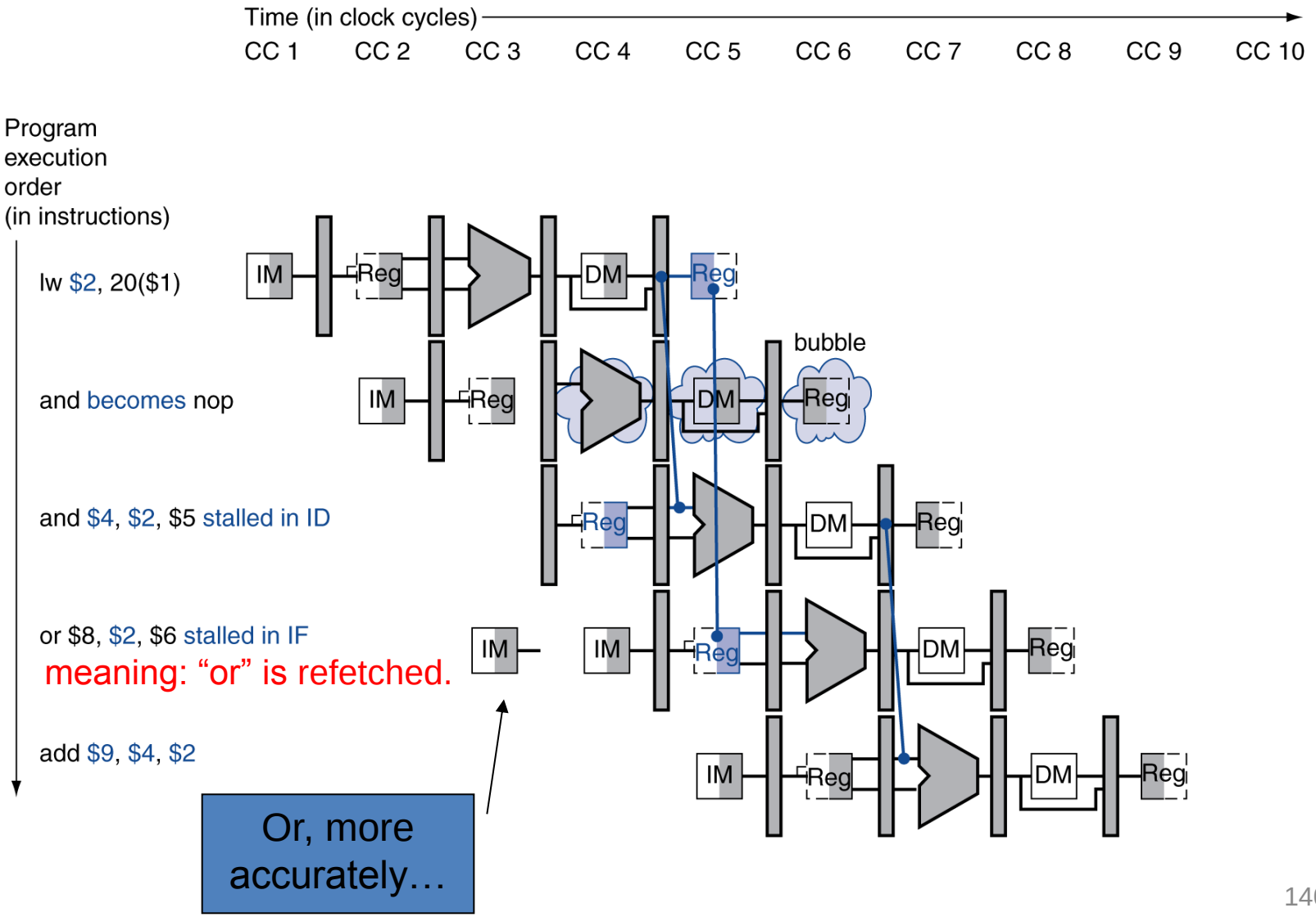
How to Stall the Pipeline

- 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

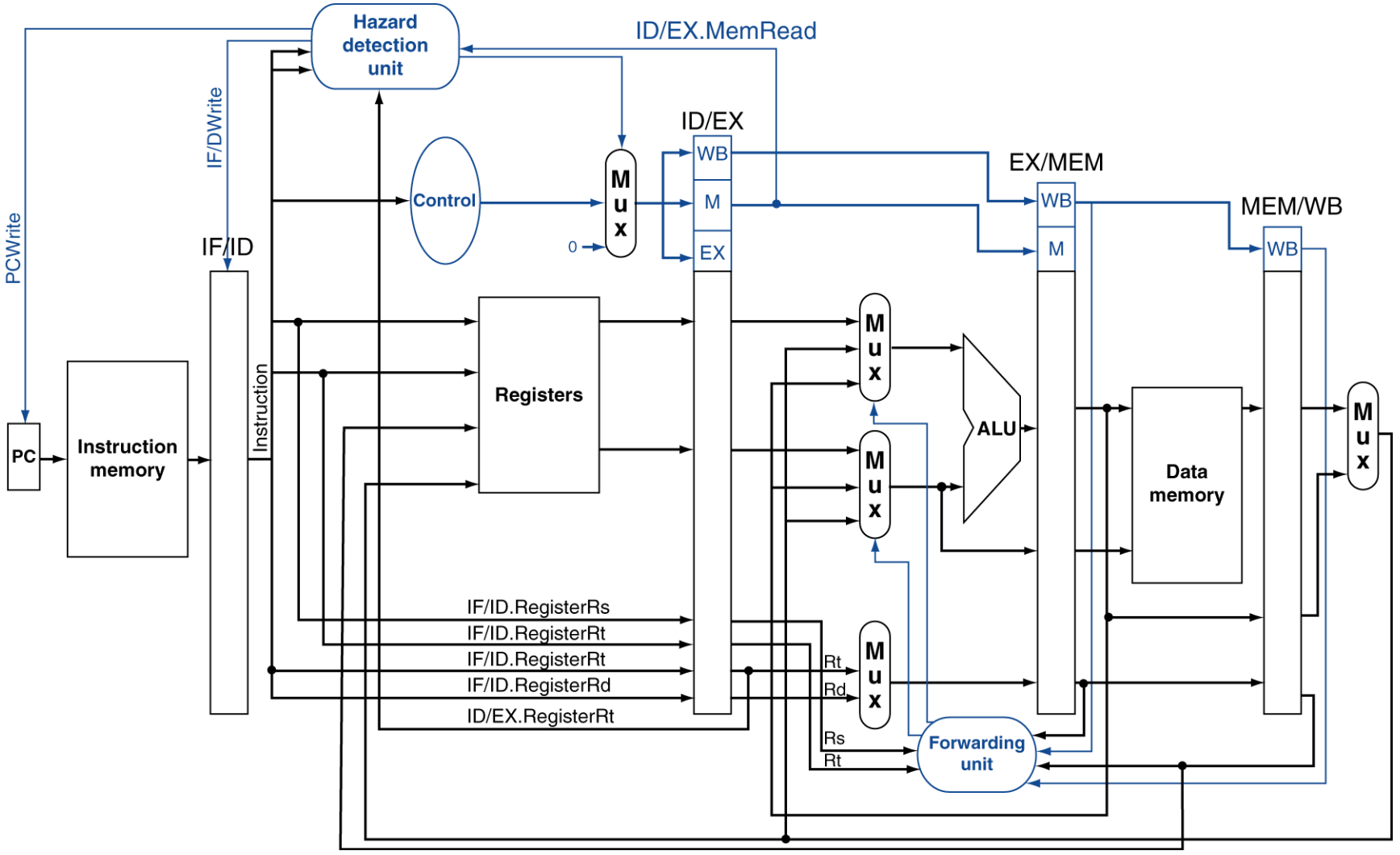
Stall/Bubble in the Pipeline



Stall/Bubble in the Pipeline



Datapath with Hazard Detection



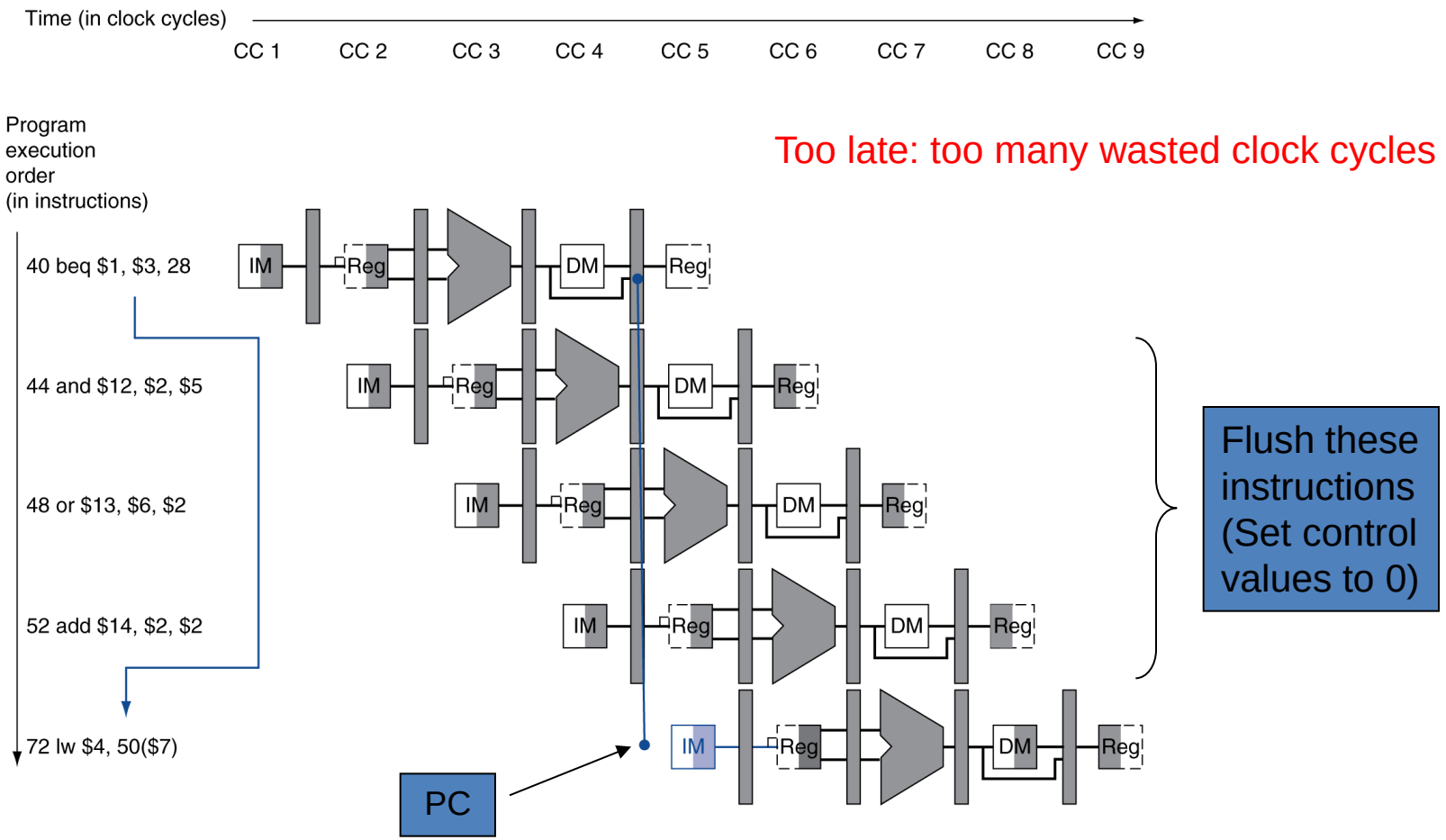
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Branch Hazards

- If branch outcome determined in MEM

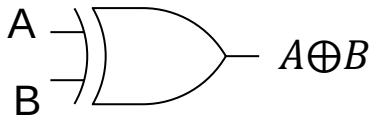


Reducing Branch Delay

- Move hardware to determine outcome to ID stage. We need two additional pieces of hardware:
  1. Target address adder (32 bit adder) so we can caculate branch target address early if branch taken
  2. Register comparator (32 bit comparitor using XNOR and AND gates) so we can do the branch test early.
- Example: branch taken

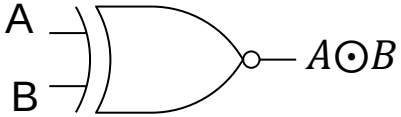
```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Exclusive or



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

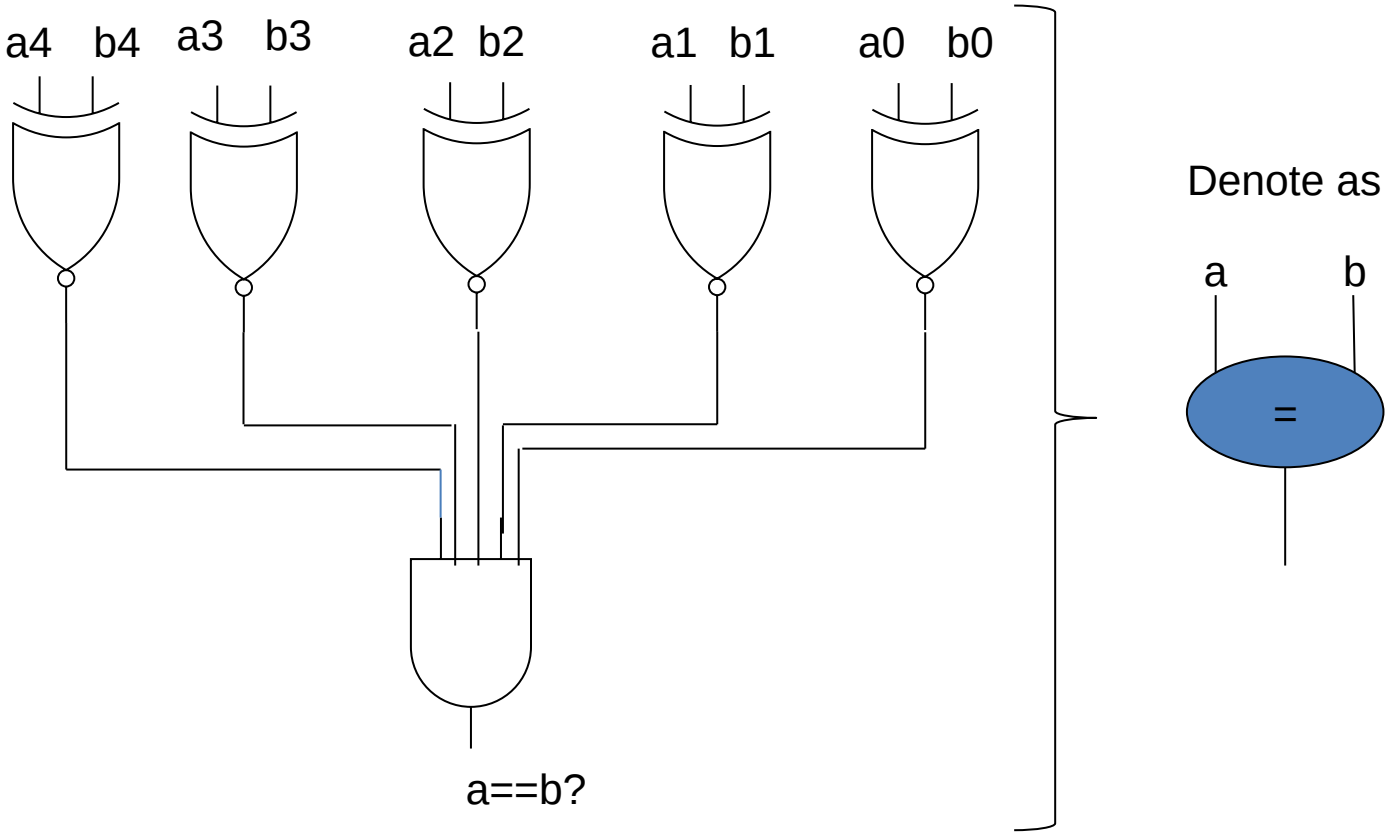
Exclusive Nor



A	B	$A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

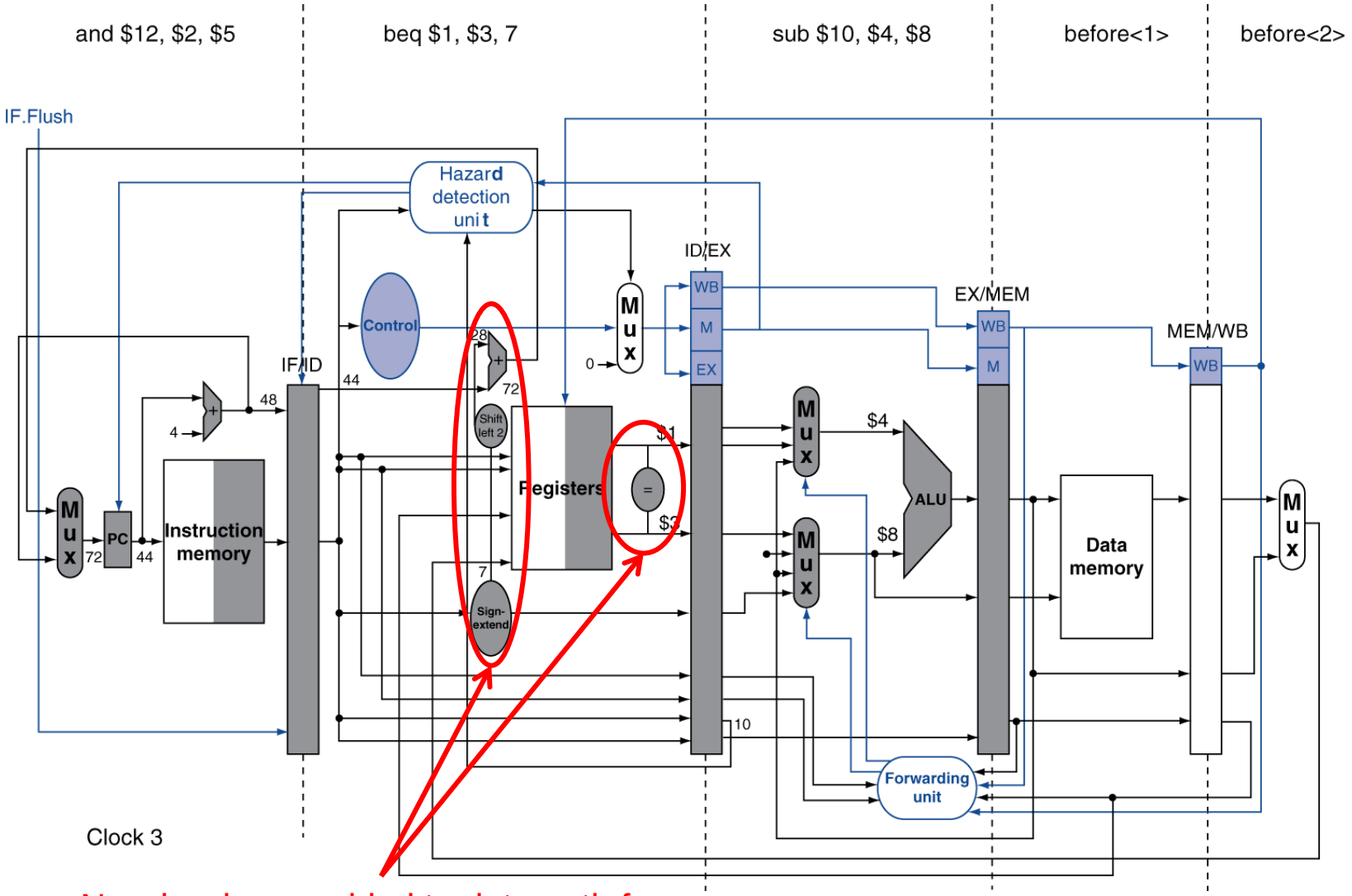
Also called equivalence
Can be used to check equality

How to check if two 5 bit numbers are equal?



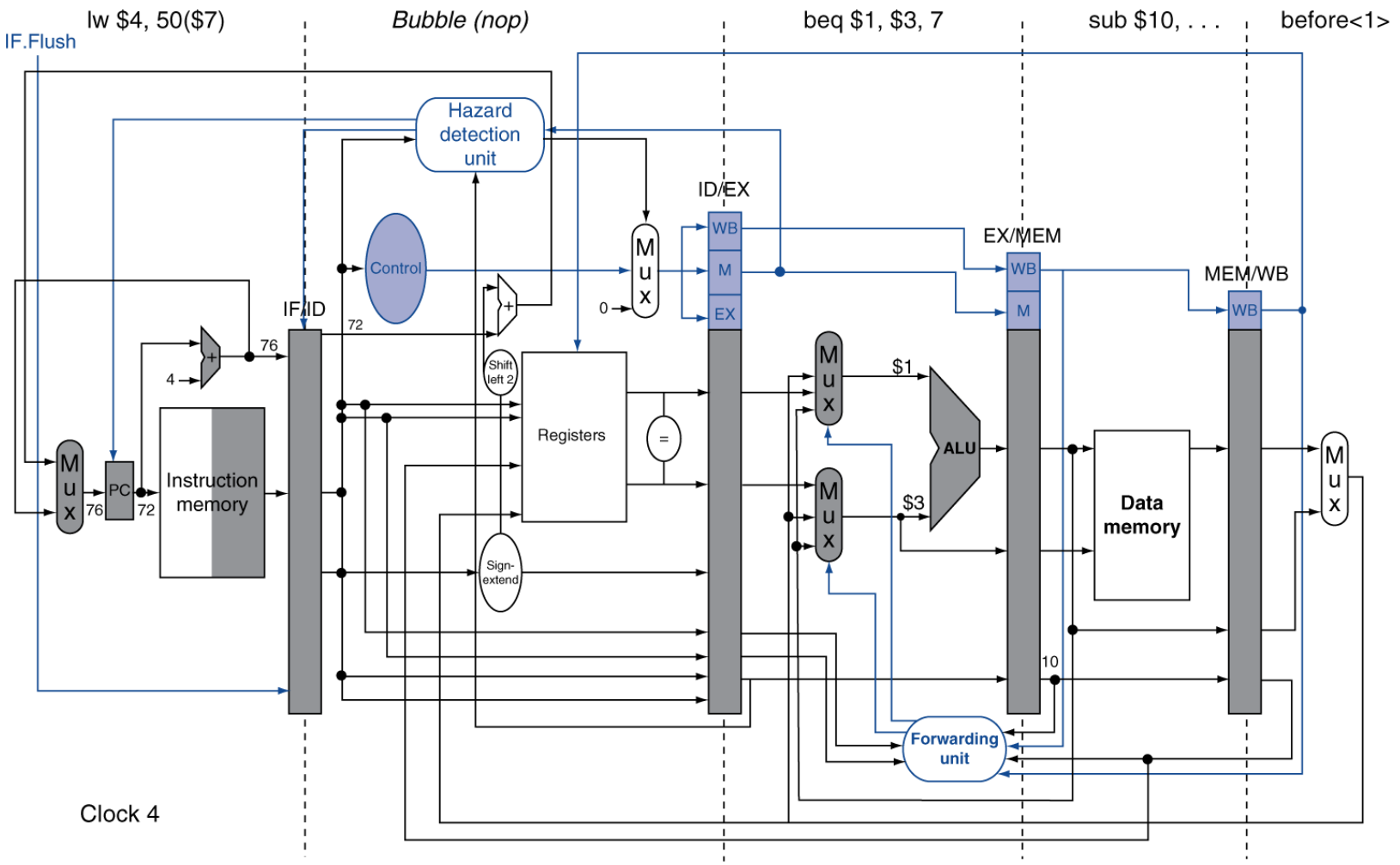
Can do it for any number of bits: e.g., 32 bit numbers.

Example: Branch Taken



New hardware added to data path for earlier branch outcome determination

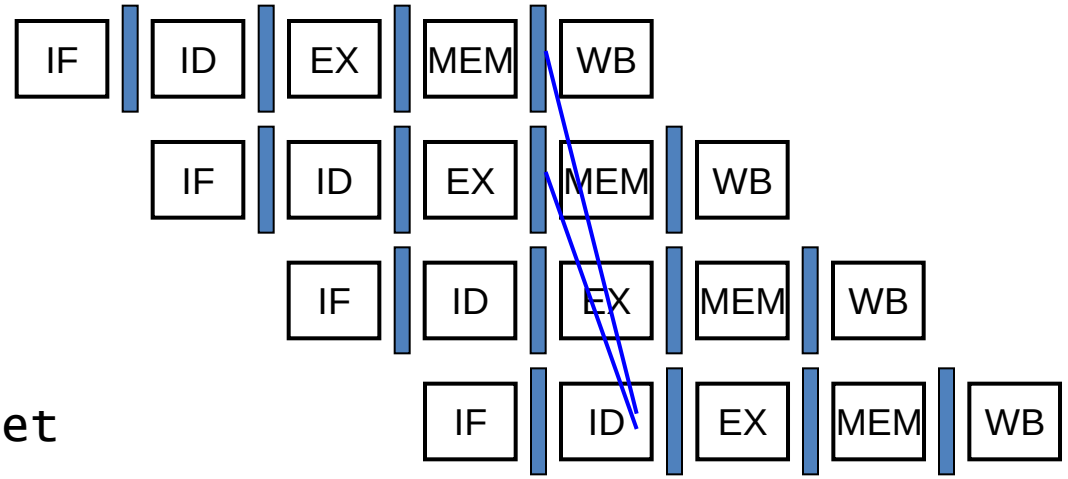
Example: Branch Taken



Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

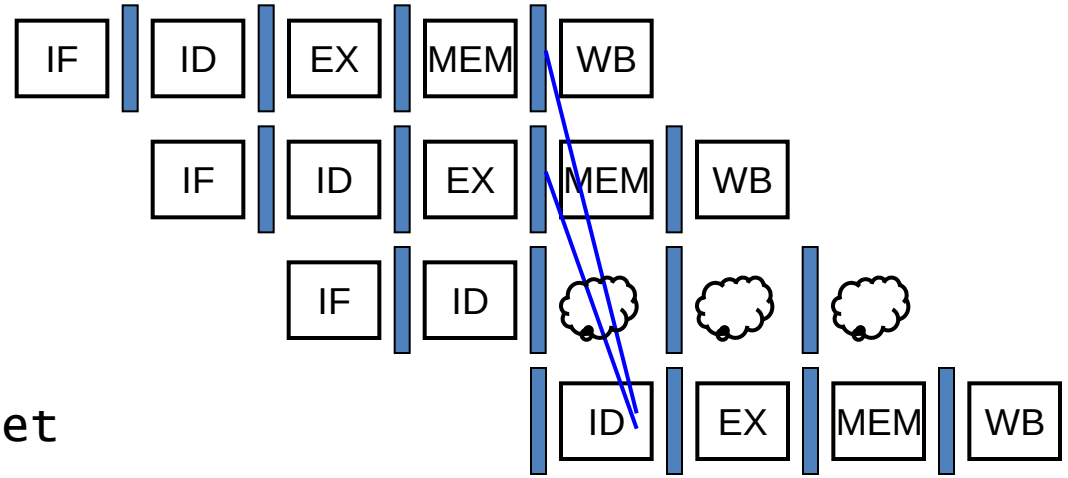


- Can resolve using forwarding

Data Hazards for Branches

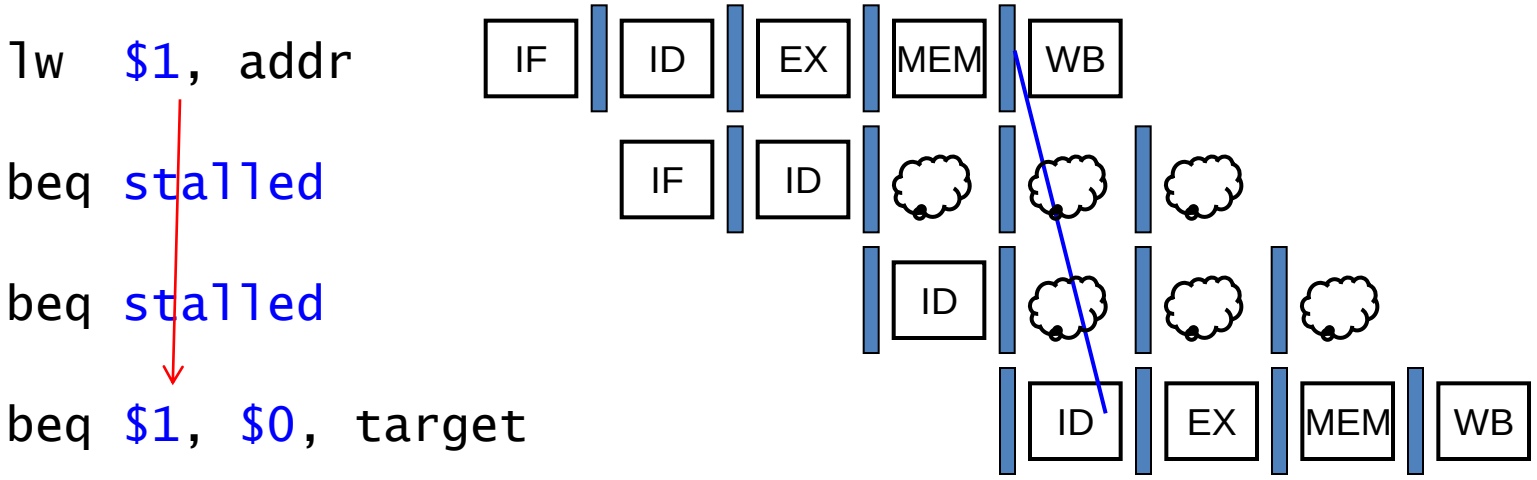
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

```
lw   $1, addr
add  $4, $5, $6
beq  stalled
beq  $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles



Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken: taken=1,not taken=0)
  - To execute a branch
    - Check table, expect/predict the same outcome
    - Start fetching from fall-through or target depending on prediction
    - If wrong, flush pipeline and flip prediction

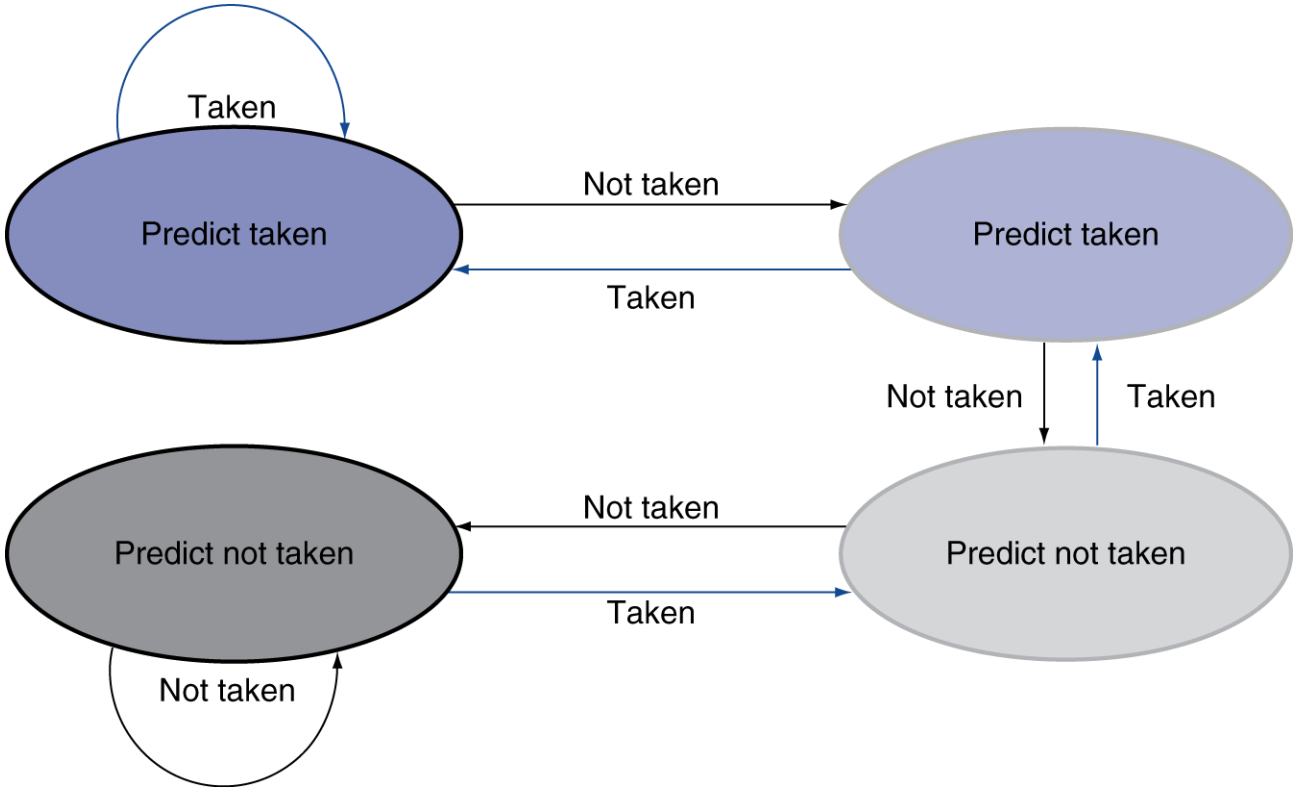
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

2-Bit Predictor

- Only change prediction on two successive mispredictions



Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

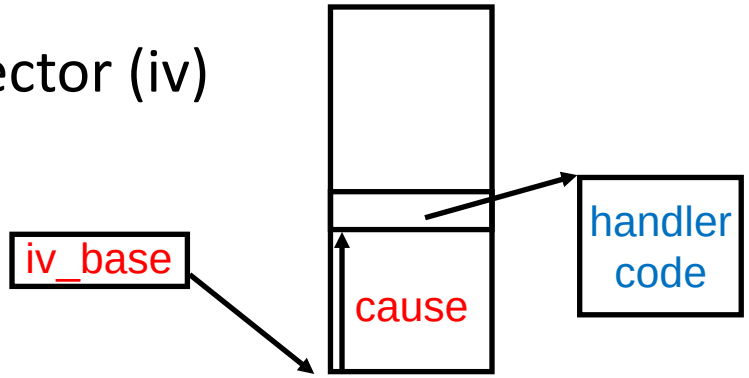
- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

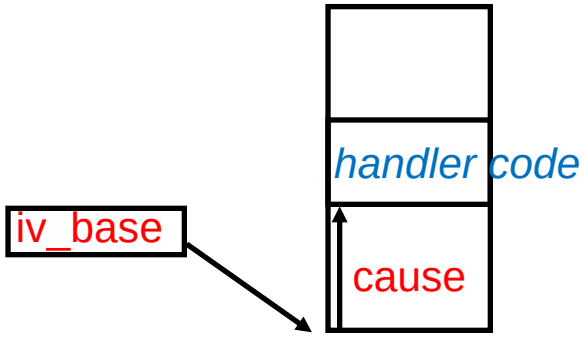
- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Addressing the Exception Handler

- Traditional Approach: Interrupt Vector (iv)
  - PC ← MEM[iv_base + cause || 00]
  - 370, 68000, Vax, 80x86, . . .



- RISC Handler Table
  - PC ← iv_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M88K, . . .



- MIPS Approach: fixed entry
  - PC ← EXC_addr
  - Let handler sort it out

Cause register: stores the cause of exception

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

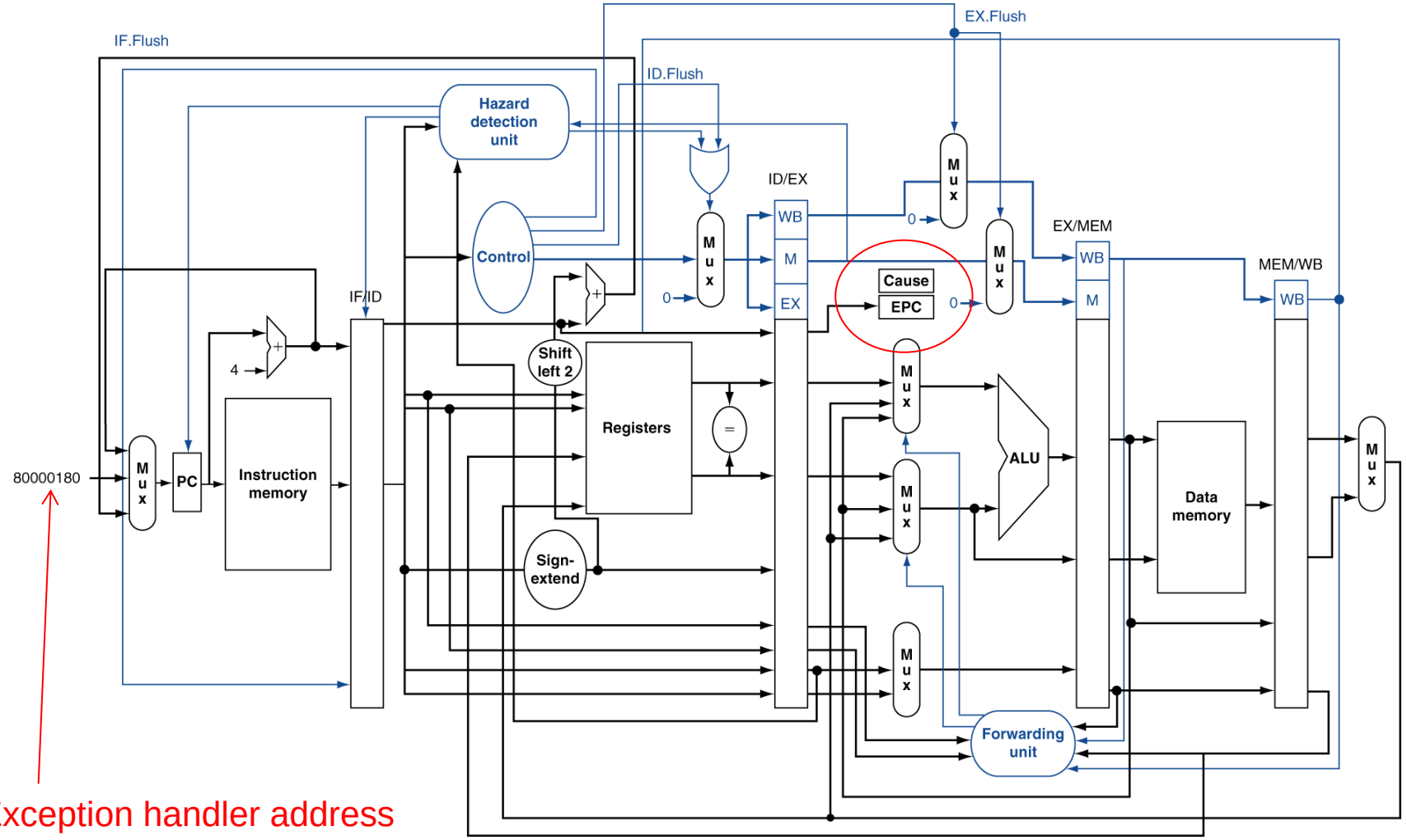
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`

  – Prevent $1 (Rd) from being clobbered
  – Complete previous instructions in pipeline
  – Flush add and subsequent instructions in pipeline
  – Set Cause and EPC register values
  – Transfer control to handler
- Similar to mispredicted branch
  – Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust and save PC

Exception Example

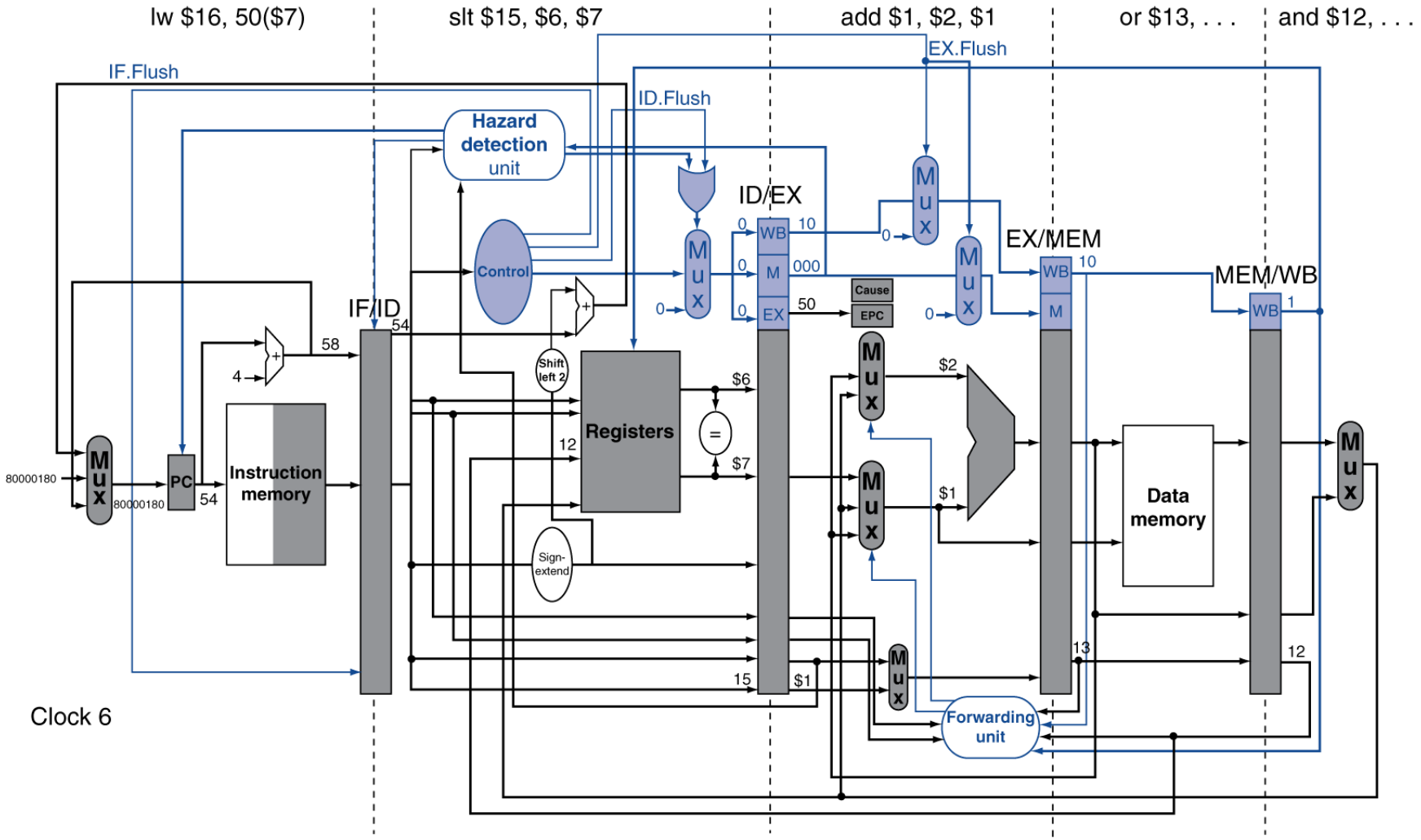
- Exception on add in

```
40      sub   $11, $2, $4
44      and   $12, $2, $5
48      or    $13, $2, $6
4C      add   $1,  $2, $1
50      slt   $15, $6, $7
54      lw    $16, 50($7)
…
```

- Handler

```
80000180    sw    $25, 1000($0)
80000184    sw    $26, 1004($0)
…
```

Exception Example

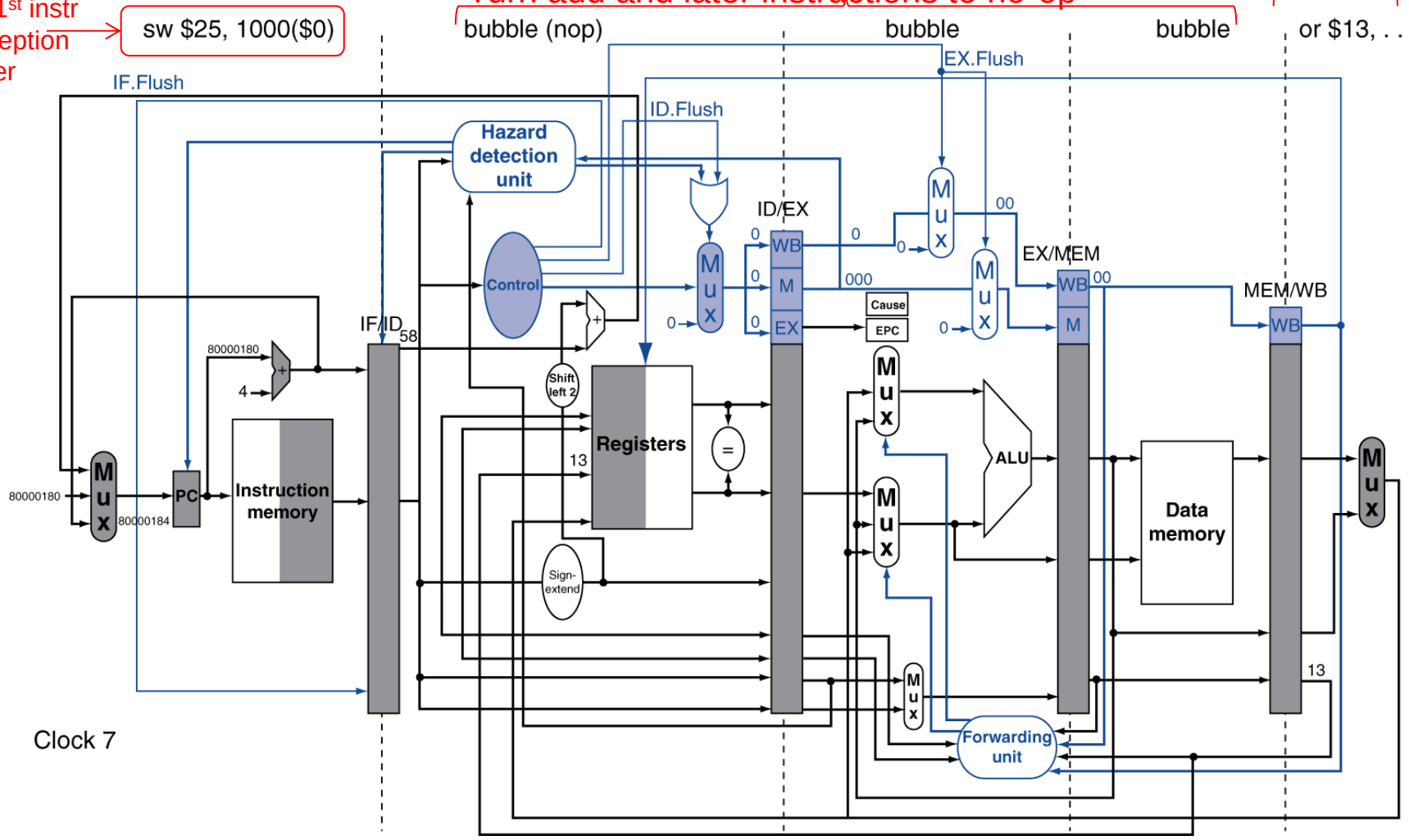


Exception Example

Complete previous instr.

Turn add and later instructions to no-op

Load 1st instr of exception handler



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

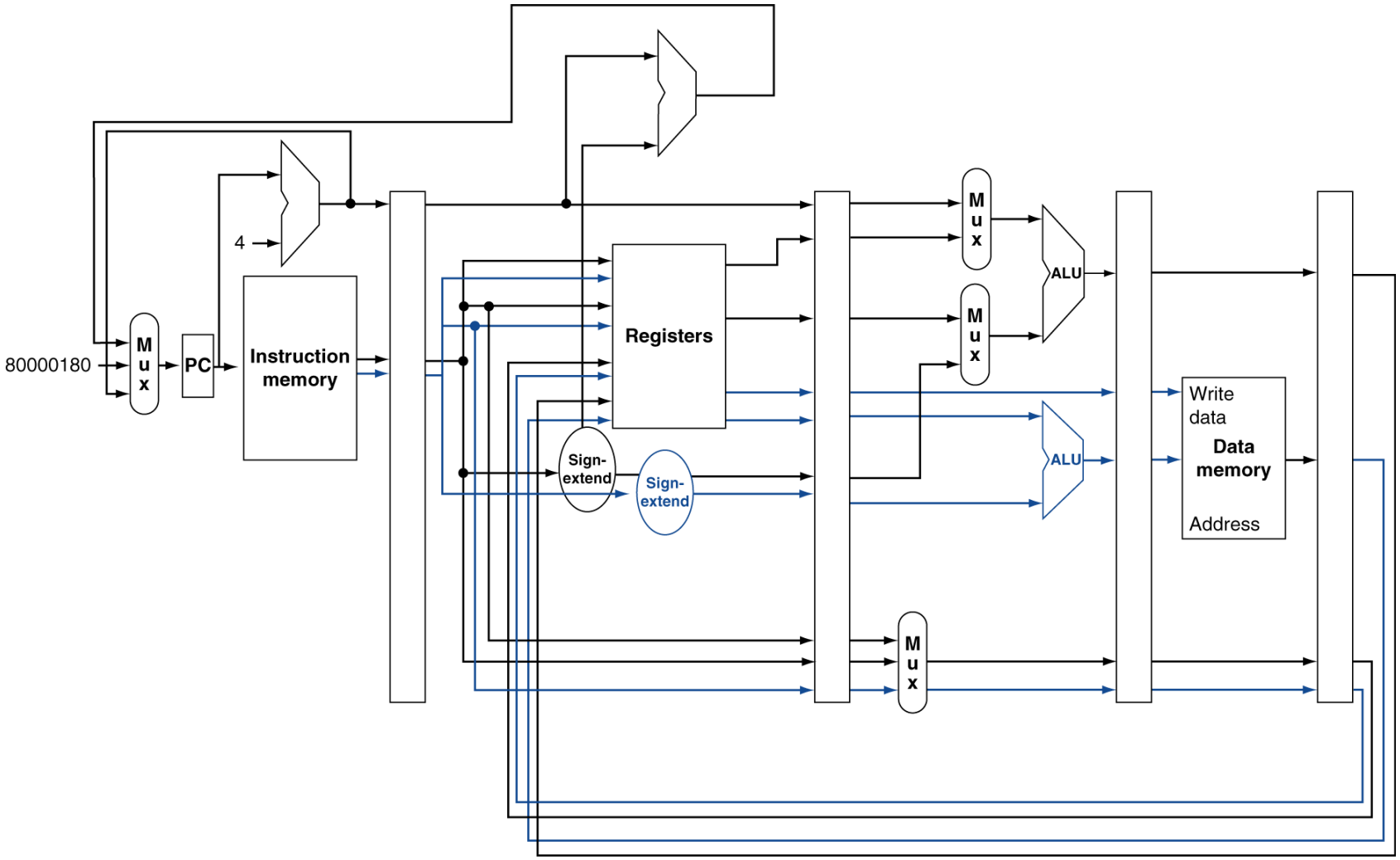
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

MIPS with Static Dual Issue



Hazards in the Dual-Issue MIPS

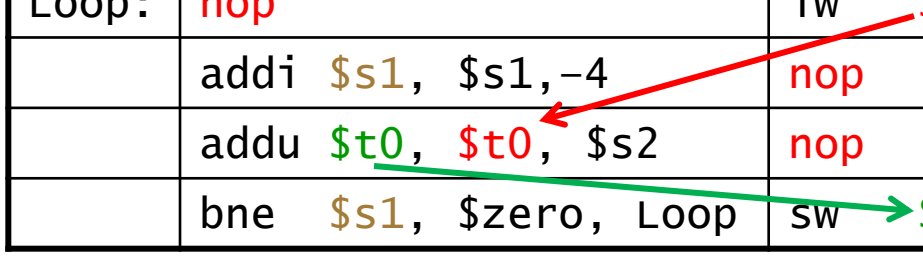
- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet
    - ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
      ```

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: 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, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4



- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Superscalar pipelines

- Example: Let's unroll the loop once (I.e, two copies)

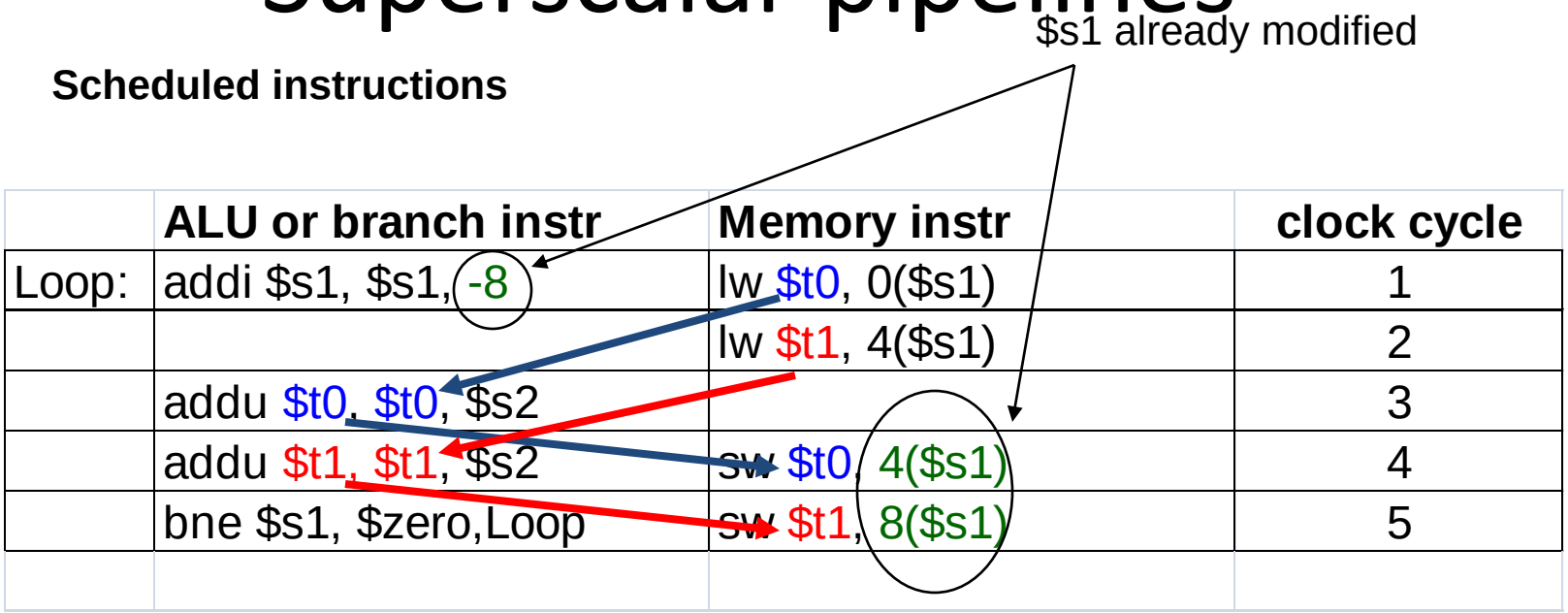
```
Loop:   lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        lw      $t1, –4($s1)      # get next element
        addu    $t1, $t1, $s2
        sw      $t1, –4($s1)      # store in correct place
        addi    $s1, $s1, –8      # decrement by 8
        bne     $s1, $zero, Loop
```

Register renamed

- t1: additional temp register needs to be used.
- Proper address arithmetic must be used.
- How is this scheduled?

Superscalar pipelines

Scheduled instructions



	ALU or branch instr	Memory instr	clock cycle
Loop:	addi $s1, $s1, -8	lw $t0, 0($s1)	1
		lw $t1, 4($s1)	2
	addu $t0, $t0, $s2		3
	addu $t1, $t1, $s2	sw $t0, 4($s1)	4
	bne $s1, $zero,Loop	sw $t1, 8($s1)	5

Now we are running in superscalar mode 6 out of 8 instructions..
6/8 = 75% utilization.
IPC=8/5=1.6
Can we do better?

Loop Unrolling Example

Unroll 4 copies of the loop body

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	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
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

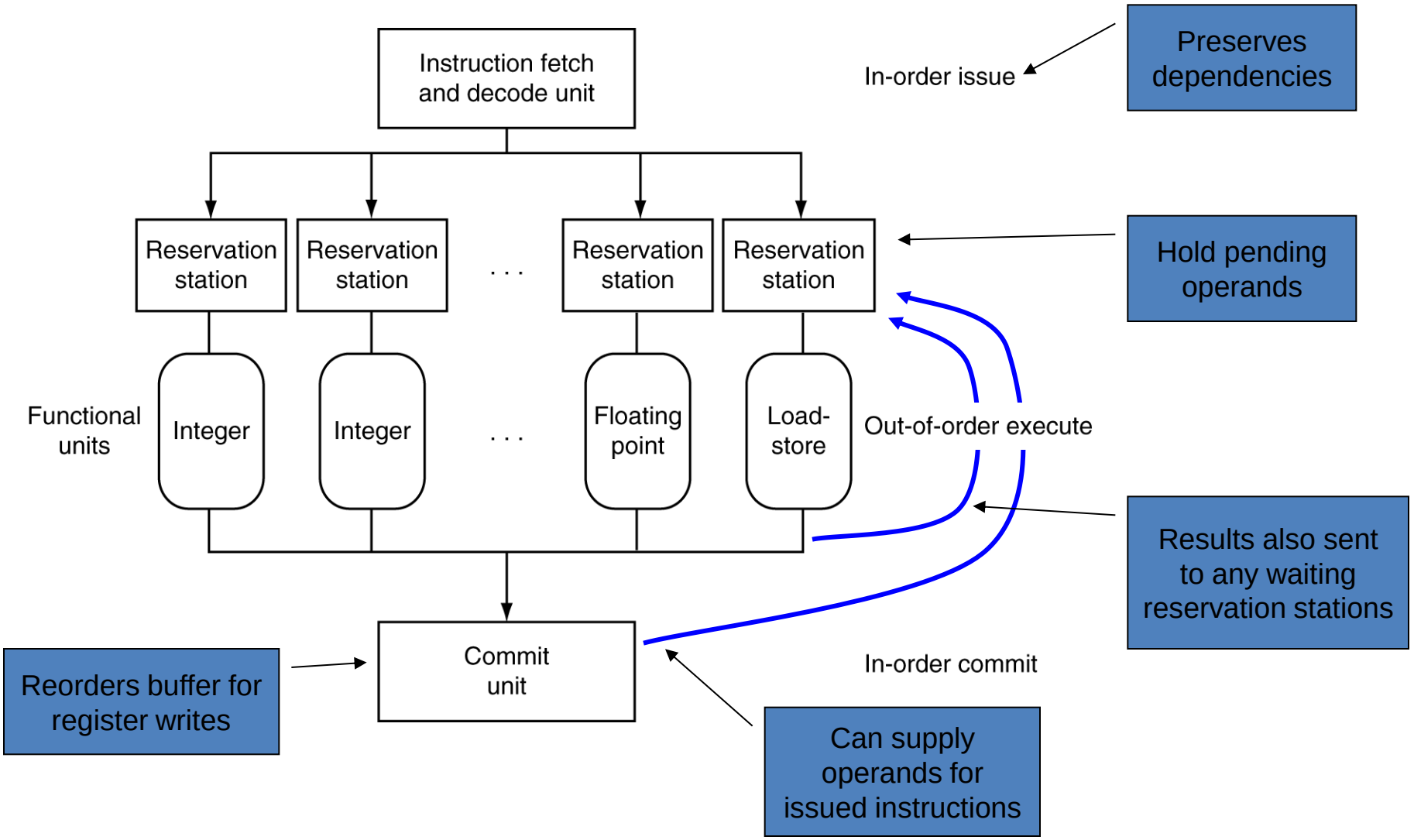
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start `sub` while `addu` is waiting for lw

Dynamically Scheduled CPU



Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions
      - movez $8, $11, $4 # $8 ← $11 if ($4 == 0) else nop
      - moven $8, $11, $4 # $8 ← $11 if ($4 != 0) else nop
      - Always next instruction fetched. (no speculation needed, no control hazard)

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall