Chapter 4

The Processor

Introduction

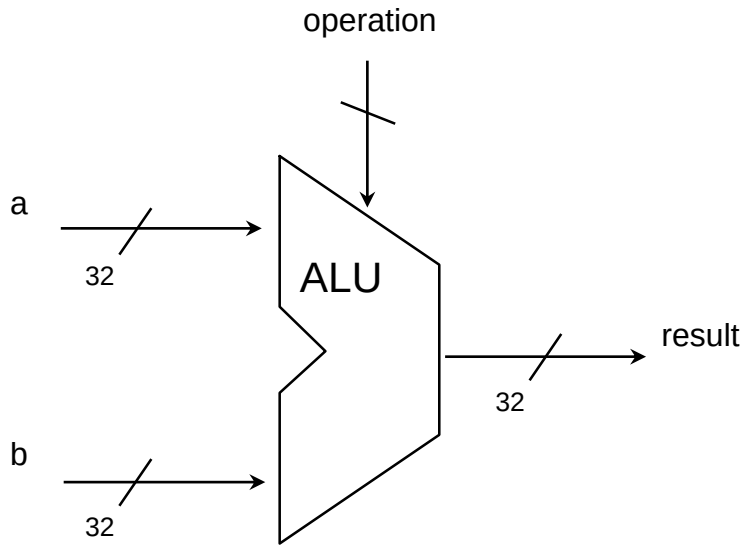
- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version (single-cycle CPU)
  - A more realistic version (pipelined CPU)

Introduction

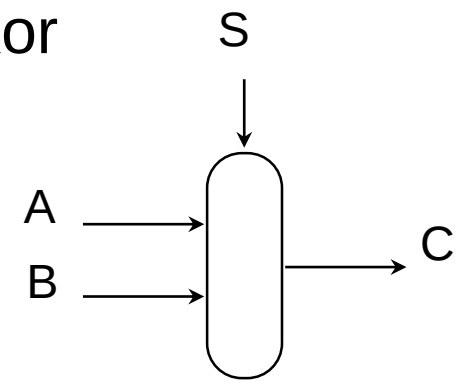
- The simplified CPU (Single-cycle CPU)
  - ALU (Arithmetic Logic Unit)
  - Datapath
  - Control Signals
    - ALU control
    - Main control
- Instructions supported
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`
- Simple subset, shows most aspects

The ALU (Appendix)

- A 32-bit ALU



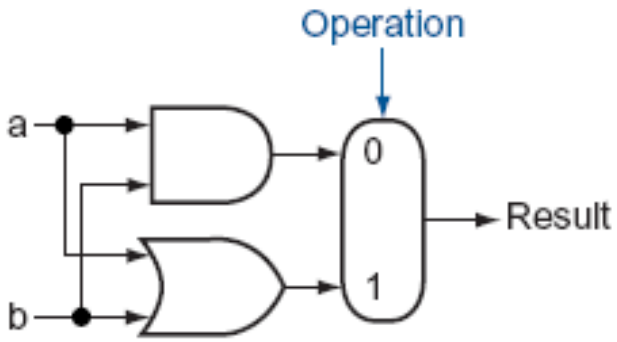
- First, let's review Boolean Logic we'll need
  - Multiplexor



note: we call this a 2-input mux even though it has 3 inputs!

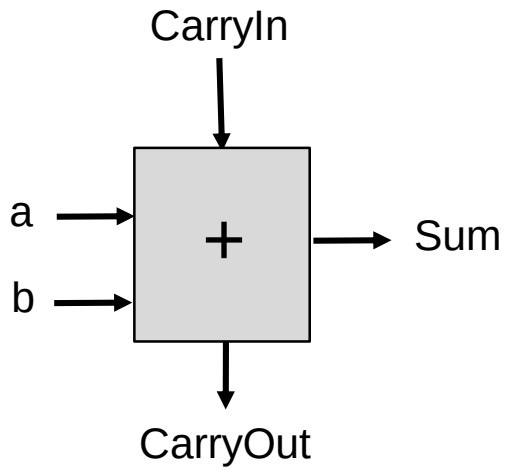
The ALU

- 1-bit and/or



- 1-bit addition:

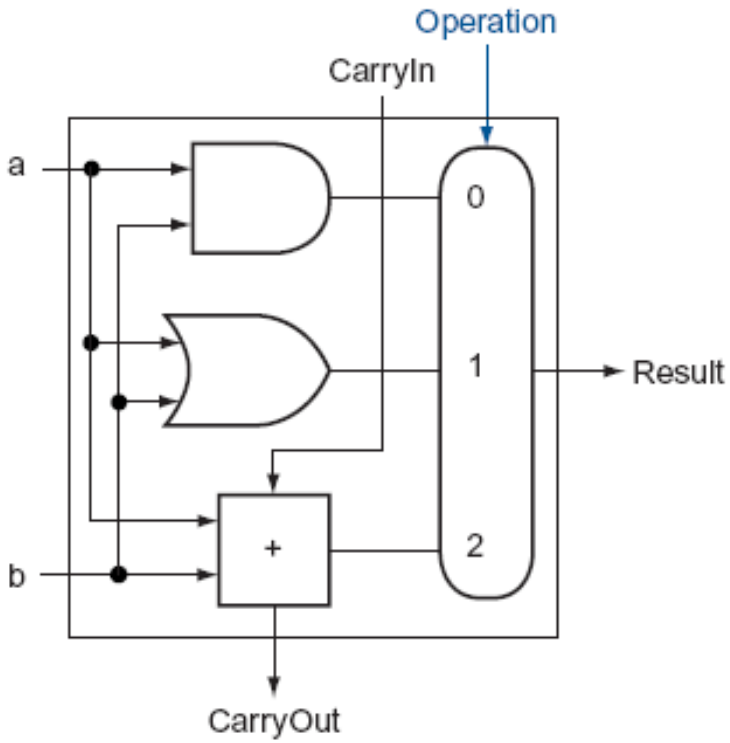
$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a\ xor\ b\ xor\ c_{in}$$



- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

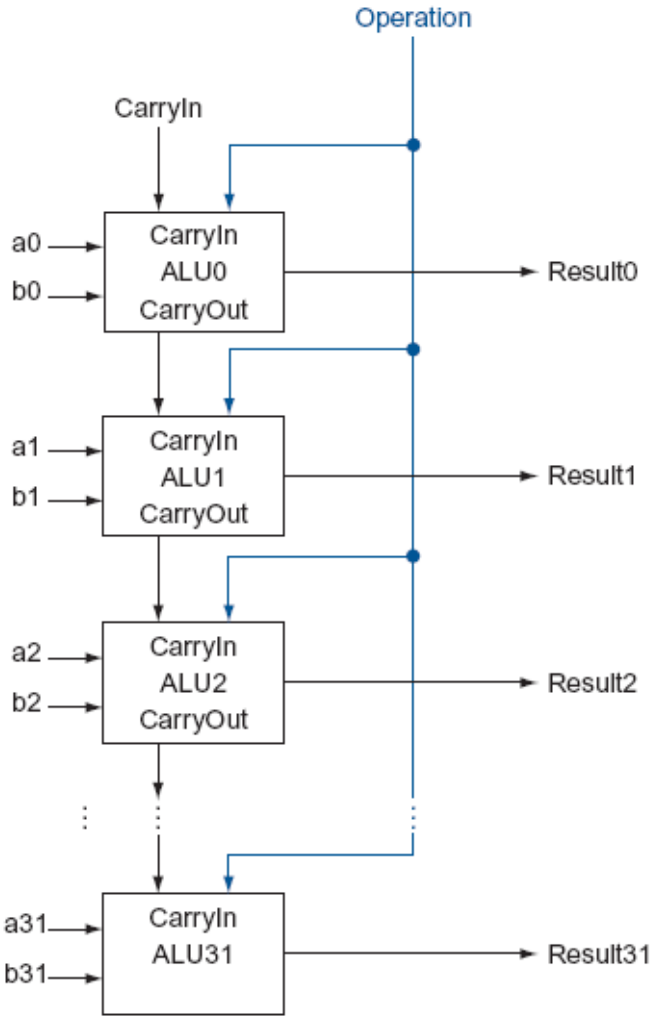
The ALU – ADD function

1-bit ALU



Control lines	
Operation	Function
00	and
01	or
10	add

32-bit ALU



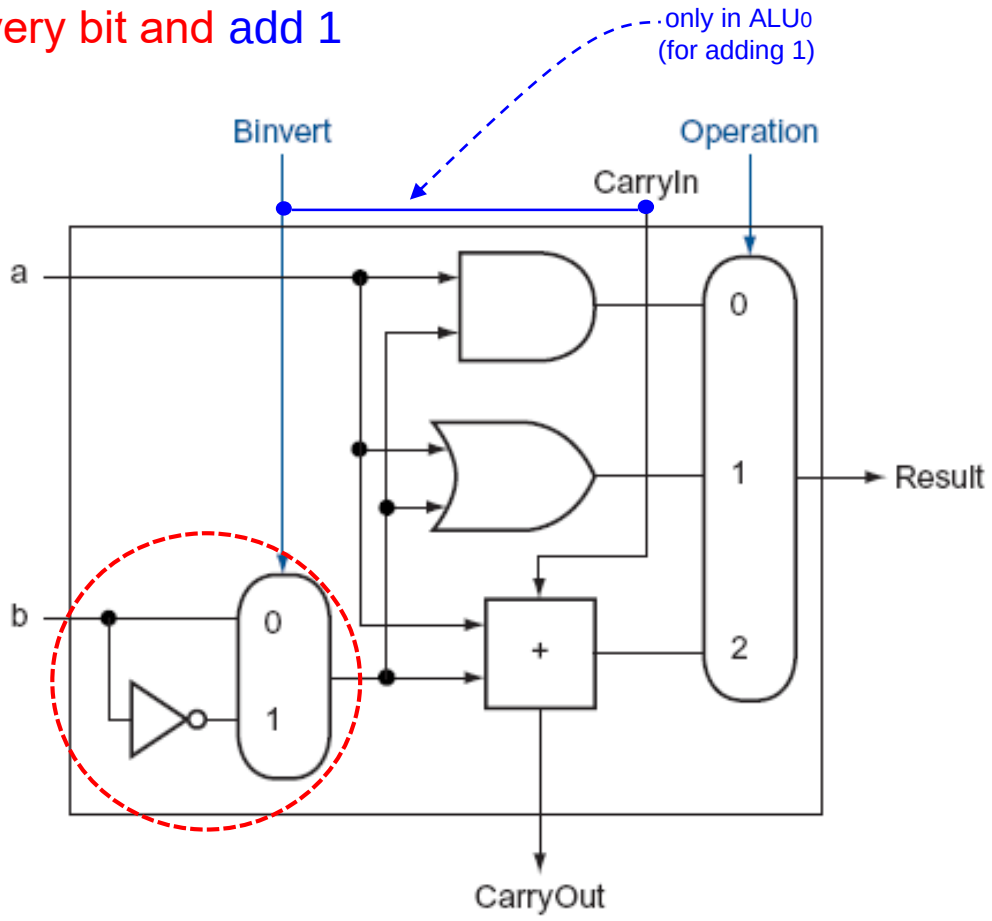
The ALU – SUB function

- What about subtraction (a-b)? → a + (-b)
  - How to negate b?
    - 2's complement: invert every bit and add 1
- A very clever solution:

Control lines		
Binvert	Operation	Function
0	00	and
0	01	or
0	10	add
1	10	sub

How to do "add 1" for **sub** ?
(need to set **C**in0 = 1)

Combining Binvert and Cin0

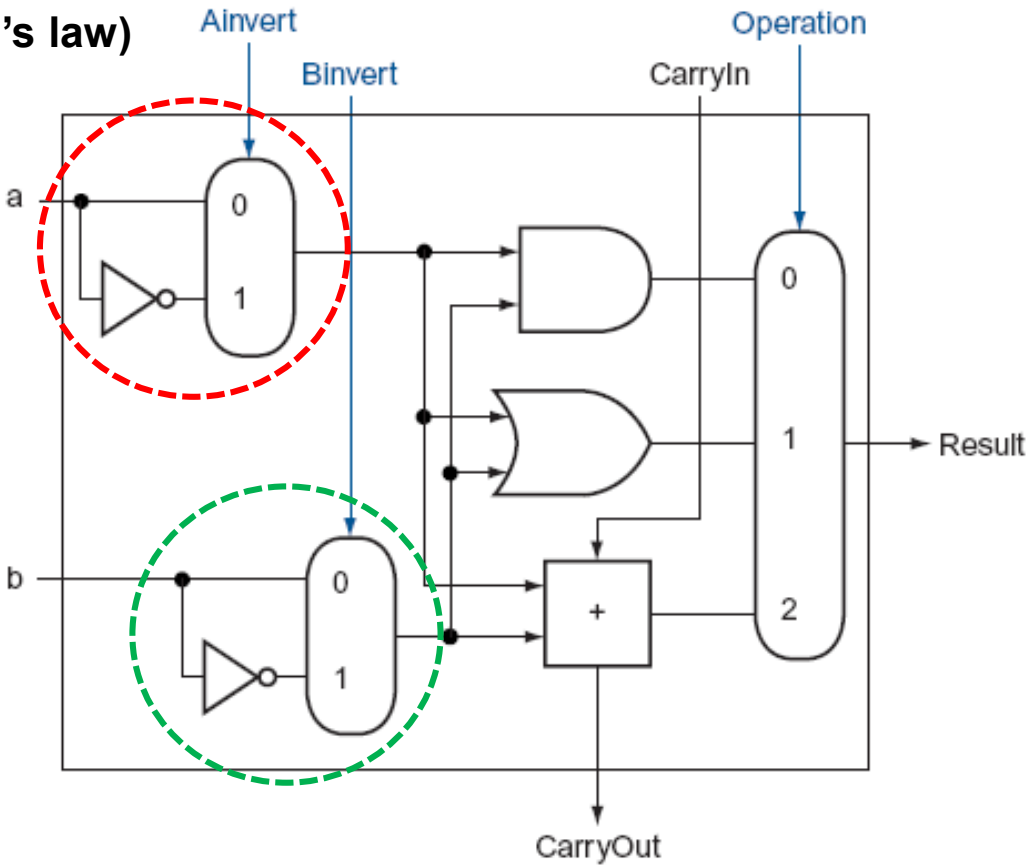


The ALU - NOR function

- NOR function
  - How do we get “a NOR b” ?

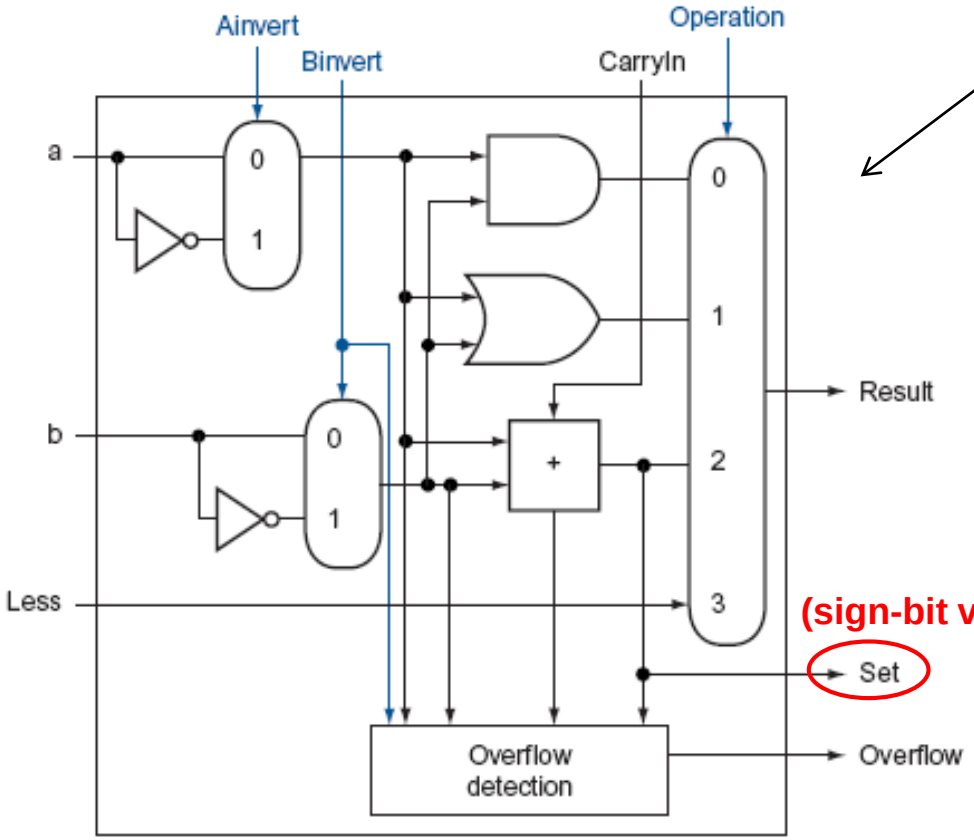
$$\overline{(a + b)} = \bar{a} \cdot \bar{b}$$ **(DeMorgan’s law)**

Control lines			
Ainv	Binv	Operation	Function
0	0	00	and
0	0	01	or
0	0	10	add
0	1	10	sub
1	1	00	nor



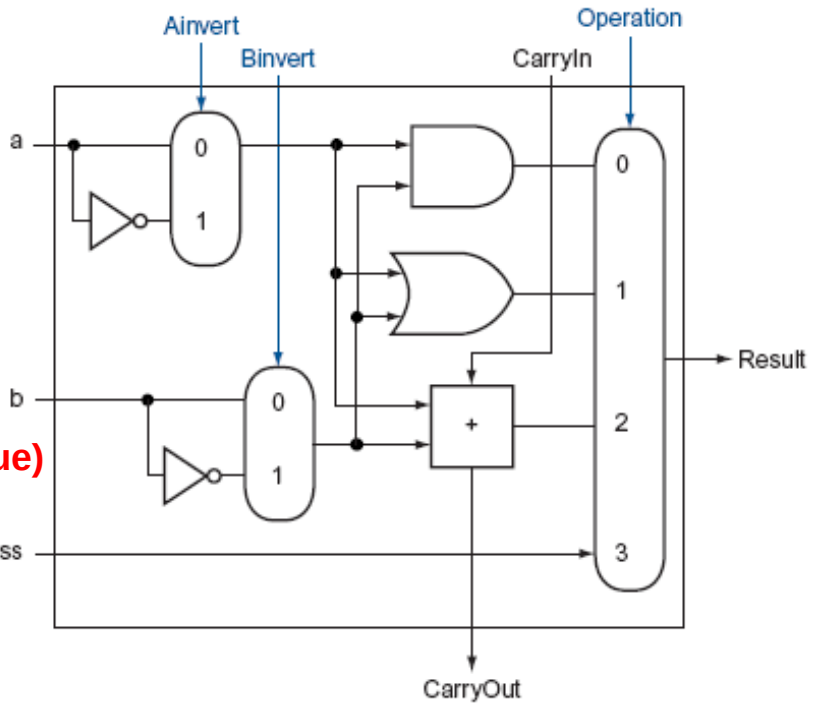
The ALU – SLT function

- Slt produces a 1 if rs < rt and 0 otherwise;
- use subtraction: (a-b) < 0 implies a < b



most significant bit

all other bits



(sign-bit value)

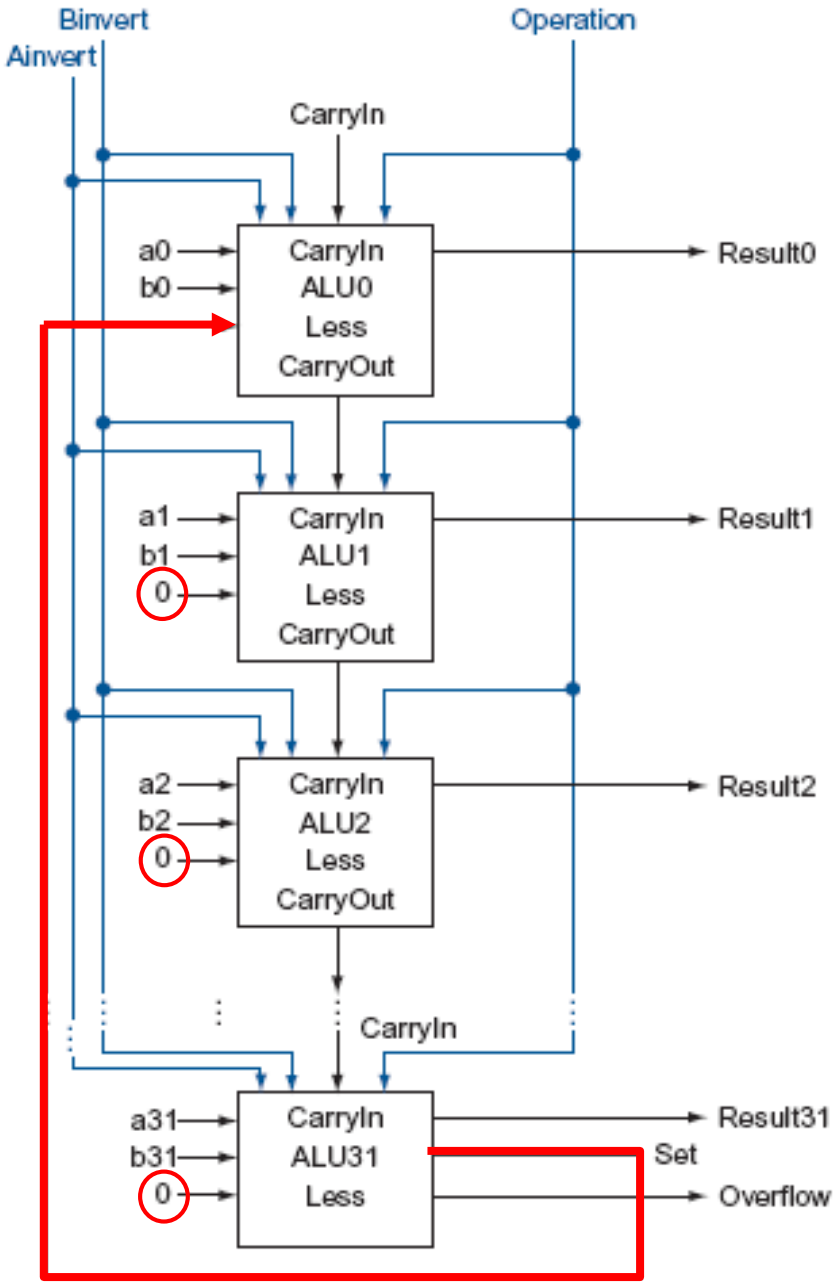
The ALU - SLT

if (a – b) < 0, Result = 0…01;

else Result = 0…00

Control lines

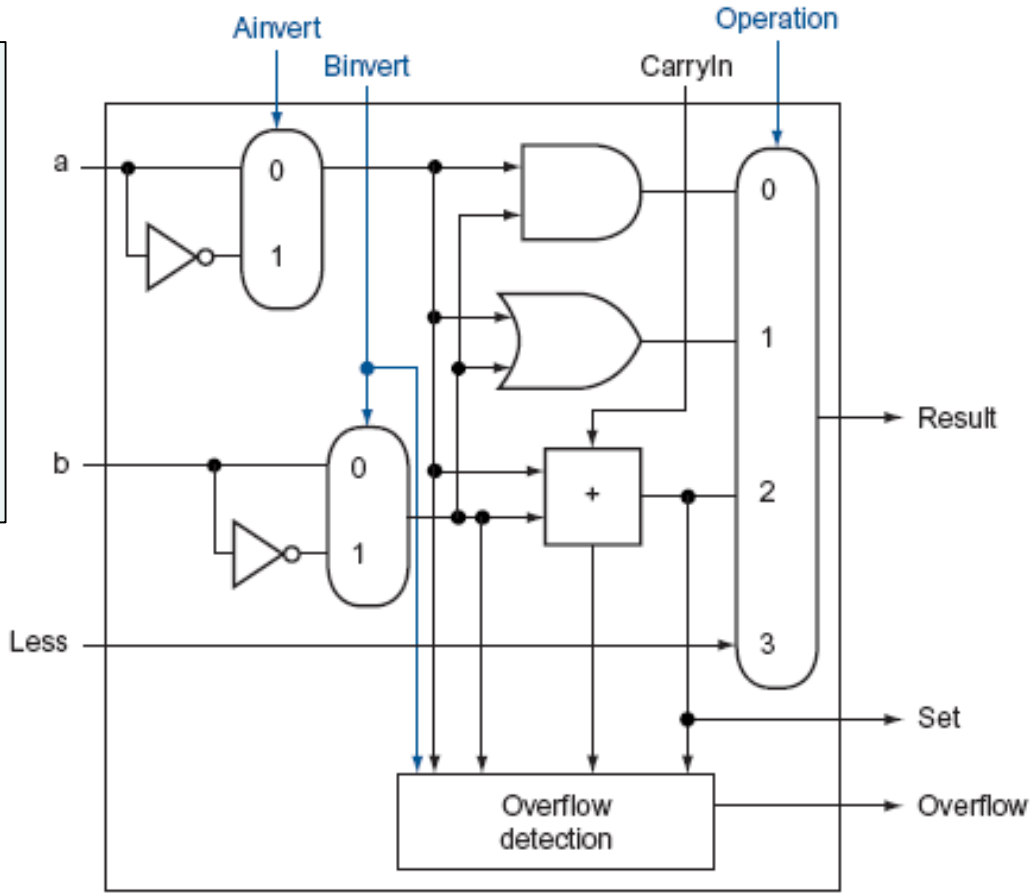
Ainv	Binv	Operation	Function
0	0	0 0	and
0	0	0 1	or
0	0	1 0	add
0	1	1 0	sub
0	1	1 1	slt
1	1	0 0	nor



The ALU

Control lines

Ainv	Binv	Operation	Function
0	0	0 0	and
0	0	0 1	or
0	0	1 0	add
0	1	1 0	sub
0	1	1 1	slt
1	1	0 0	nor



ALU- Test for equality (for beq)

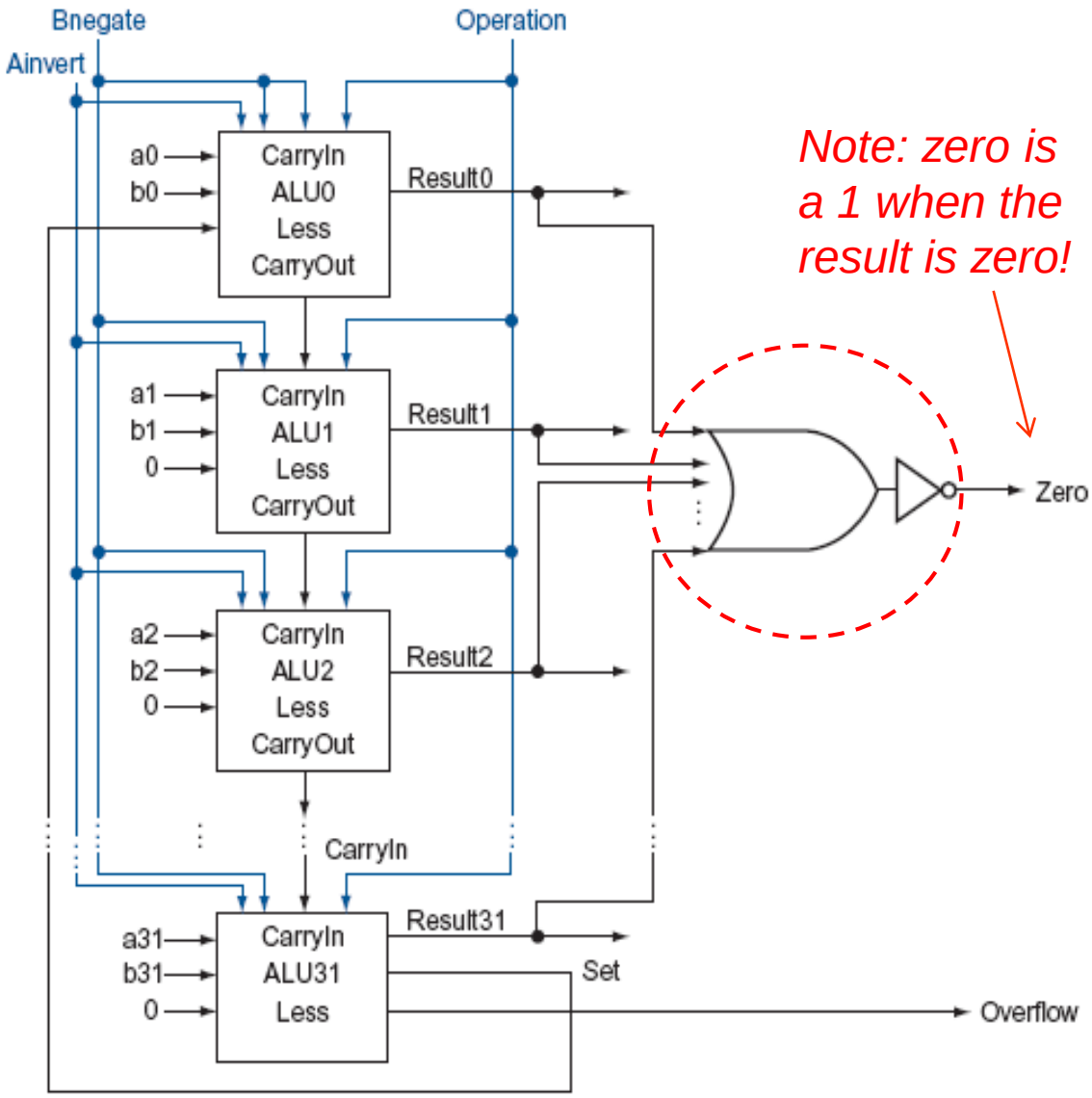
- Use subtraction:

  (a-b) = 0 implies a = b

  - Notice control lines:

```
0000 = and
0001 = or
0010 = add
0110 = subtract
0111 = slt
1100 = NOR
```

Note: we only perform the first five functions in the implementation later



ALU: Overflow

- **4-bit signed addition**

Ex.1 7 + 3 = 10, but …

	(0)	(1)	(1)	(1)	
	0	1	1	1	7
+	0	0	1	1	3
	1	0	1	0	-6

Ex.2 (- 4) + (- 5) = -9, but …

	(1)	(0)	(0)	(0)	
	1	1	0	0	-4
+	1	0	1	1	-5
	0	1	1	1	7

Overflow !

4 bits

(10-based)	(Binary)
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
-8	1 0 0 0
-7	1 0 0 1
-6	1 0 1 0
-5	1 0 1 1
-4	1 1 0 0
-3	1 1 0 1
-2	1 1 1 0
-1	1 1 1 1

Signed binary (2's complement)

ALU: Overflow

- **Overflow**: result too big/small to represent
  - For addition
    - Adding two +ve operands: overflow if result sign is 1
    - Adding two –ve operands: overflow if result sign is 0
    - Adding +ve and –ve operands: No overflow
  - How about subtraction?

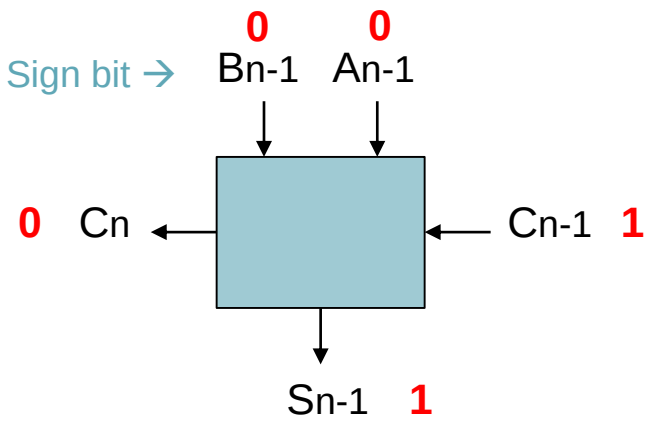
Operation	Operand A	Operand B	Result indicating overflow
A+B	≥ 0	≥ 0	< 0
A+B	< 0	< 0	≥ 0
A-B	≥ 0	< 0	< 0
A-B	< 0	≥ 0	≥ 0

  - **Overflow detection:** **Carry into MSB ≠ Carry out of MSB**

Overflow Detection

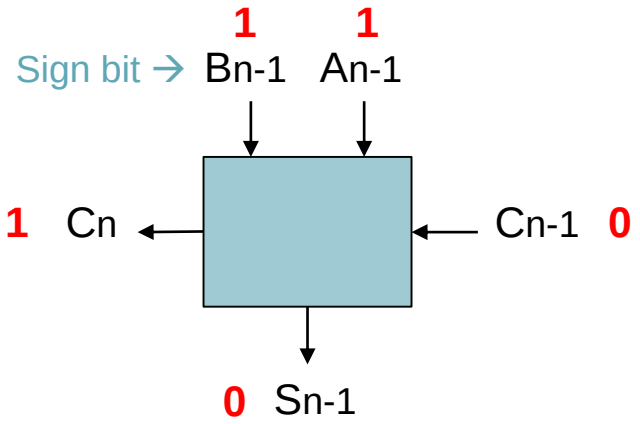
- Signed n-bit: $A_{n-1}\ A_{n-2}\ \ldots\ A_0$ → Sign bit : A_{n-1}
- For n-bit adder, overflow occurs if

1. Add two positive numbers, result is negative



→ **$C_{n-1} = 1$ and $C_n = 0$**

2. Add two negative numbers, result is positive

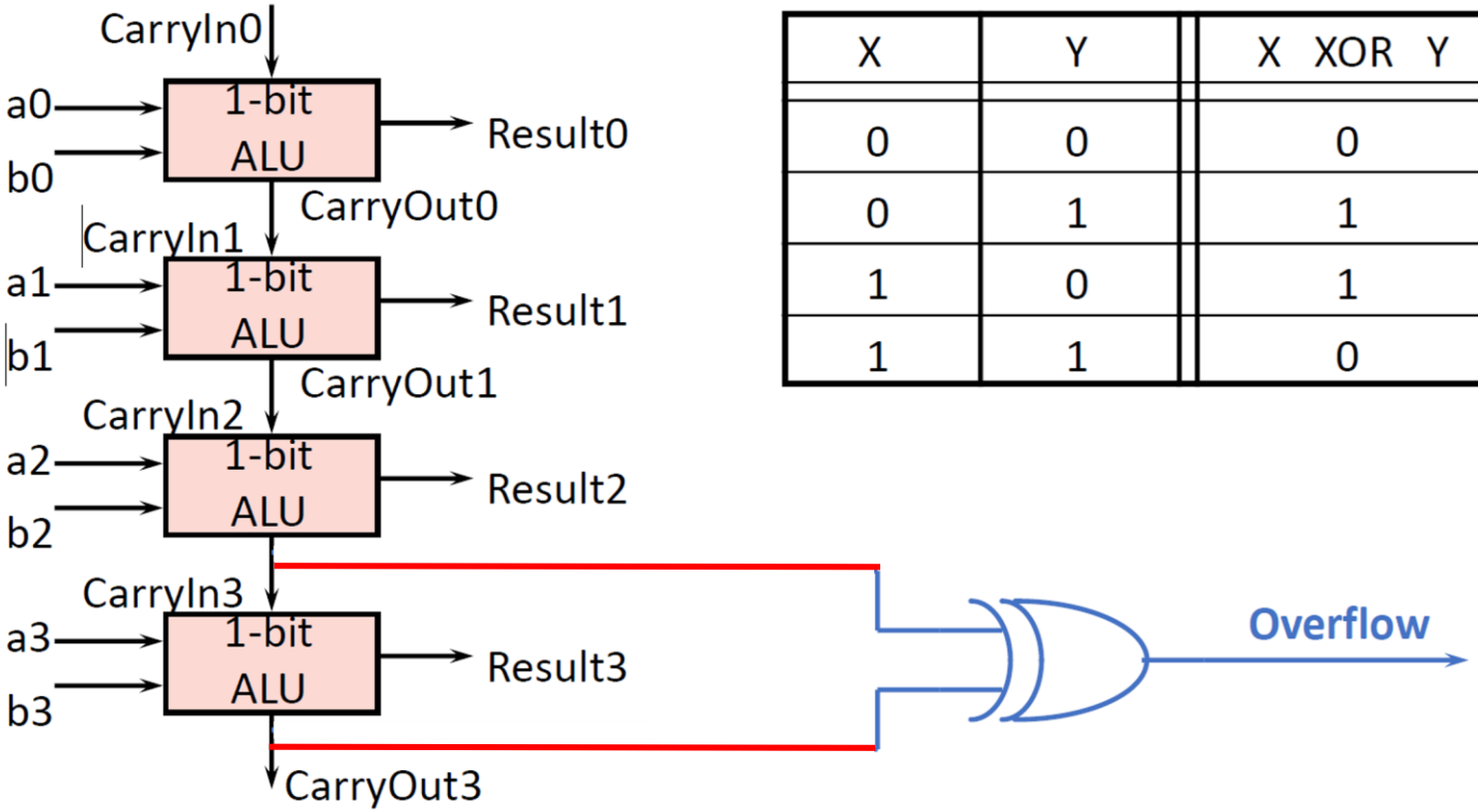


→ **$C_{n-1} = 0$ and $C_n = 1$**

C_{n-1} XOR $C_n = 1$

Overflow Detection

- Detection logic
  - Overflow = CarryIn[N-1] **XOR** CarryOut[N-1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Introduction

- The simplified CPU (Single-cycle CPU)
  - ALU (Arithmetic Logic Unit)
  - Datapath
  - Control Signals

  - We have seen the design of a simple 32-bit ALU that perform and/or/add/sub/slt/nor
  - Now we are ready to design the whole processor: Datapath + Control signals

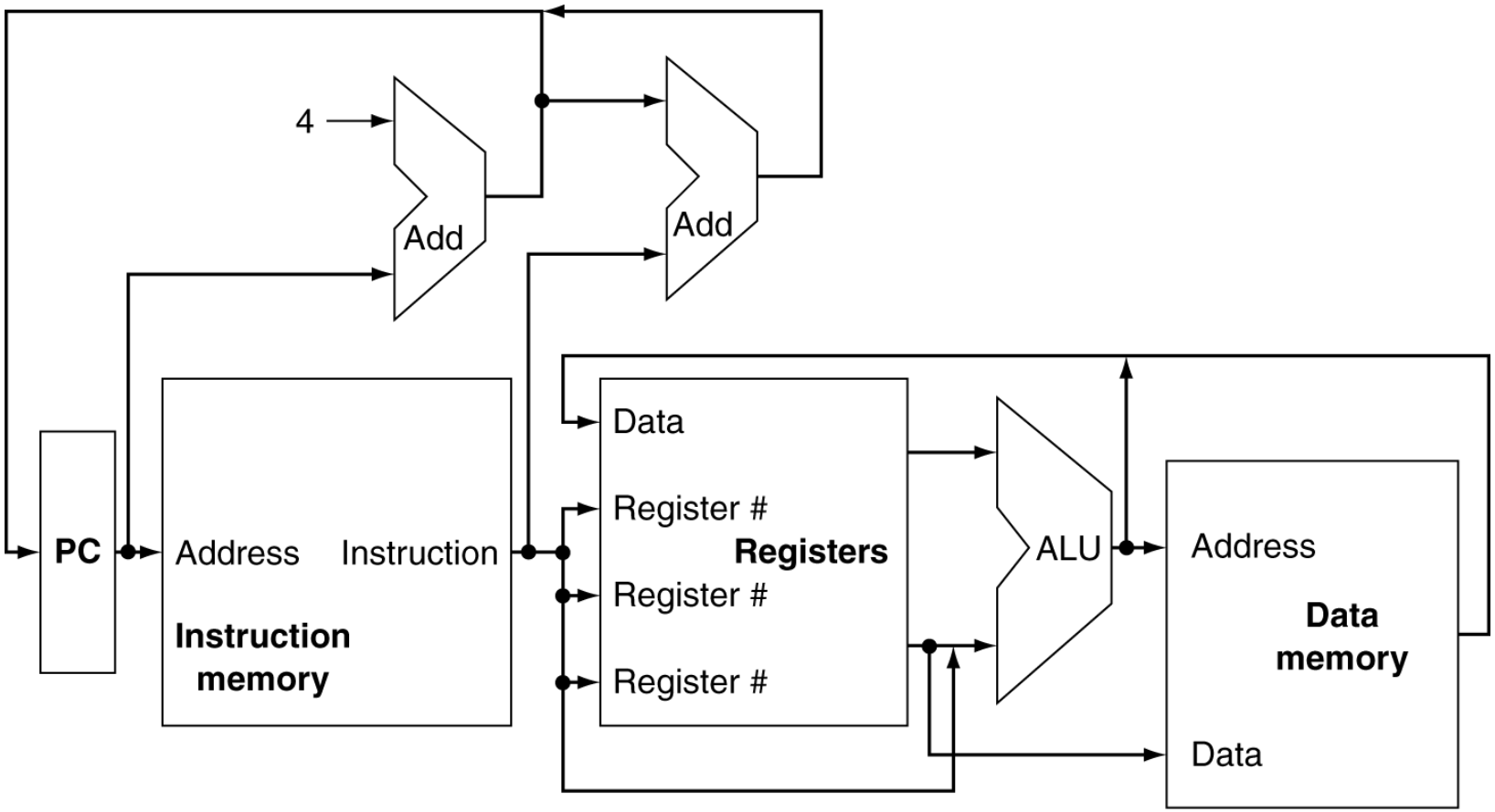
CPU Overview

Instruction Execution

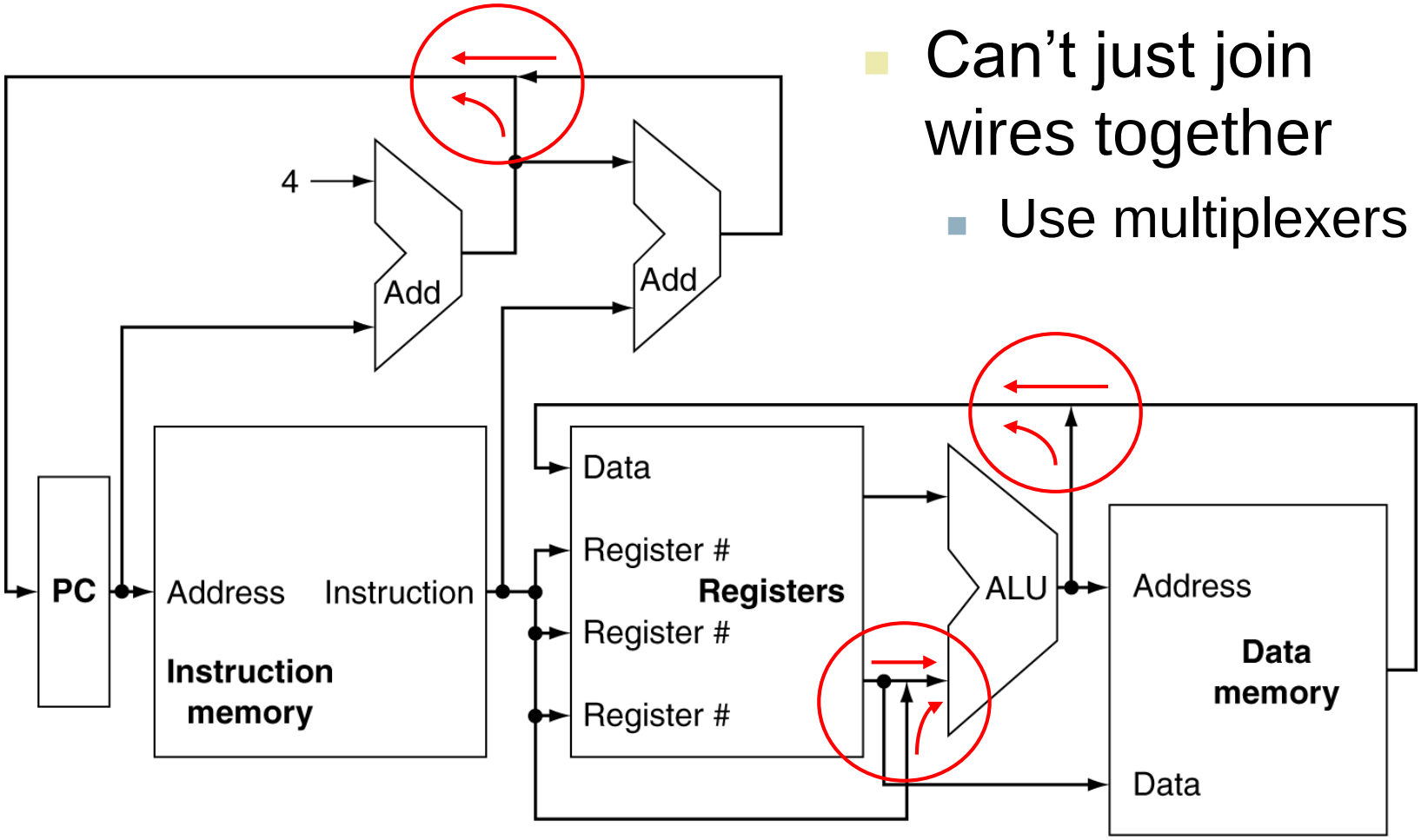
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic/logic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target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

CPU Overview

- The datapath

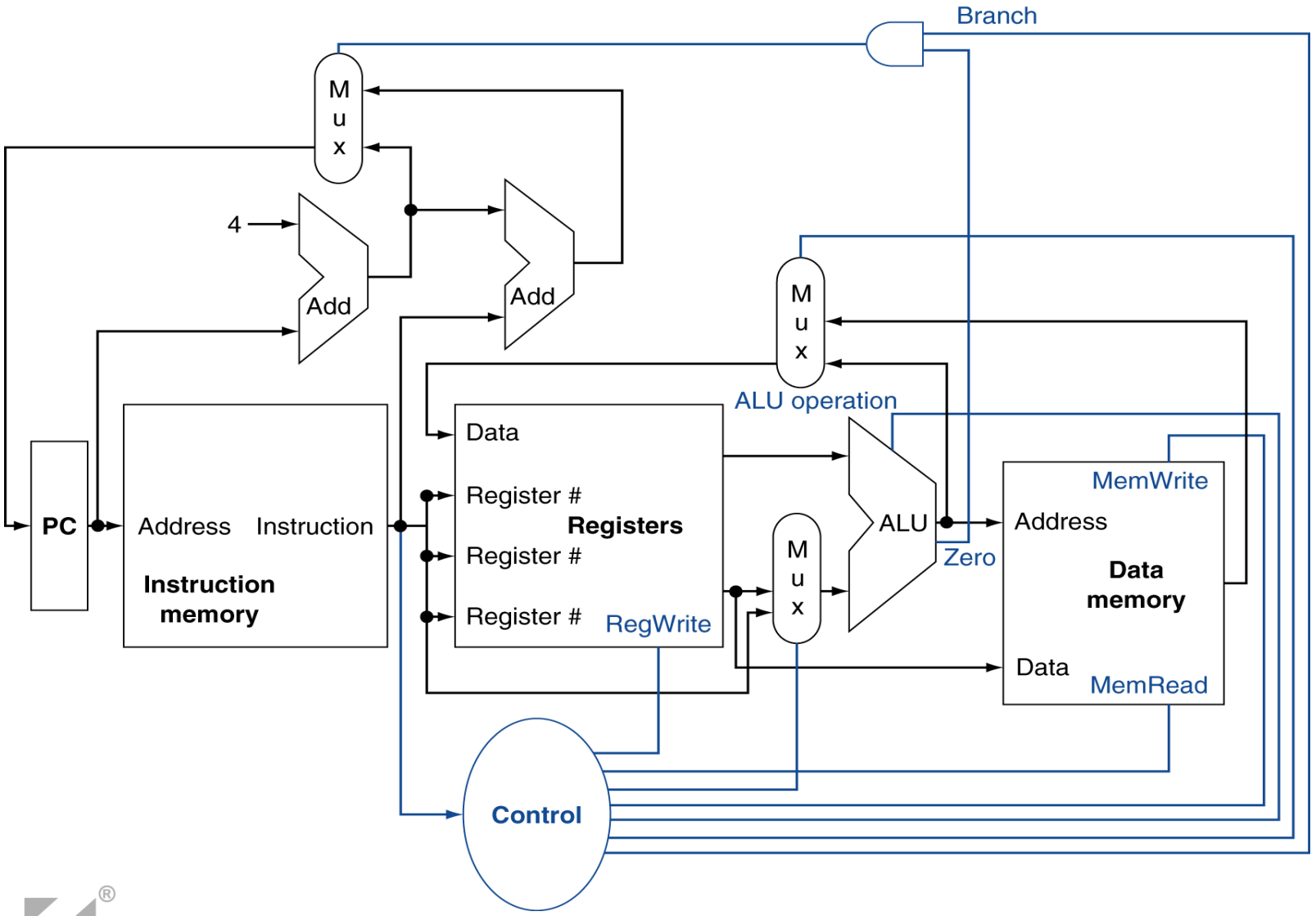


Multiplexers



- Can't just join wires together
  - Use multiplexers

Control

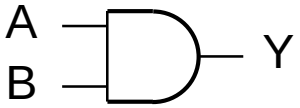

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Basics

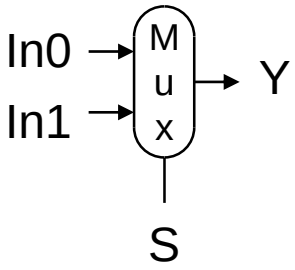
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

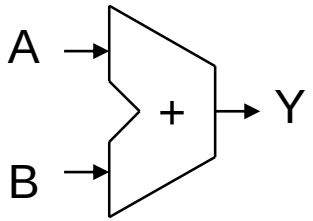
- AND-gate
  - Y = A & B



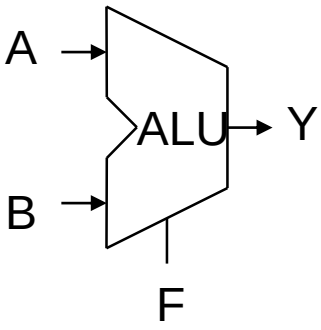
- Multiplexer
  - Y = S ? In1 : In0



- Adder
  - Y = A + B

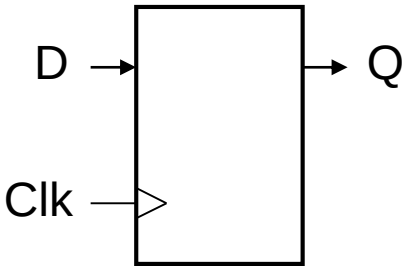


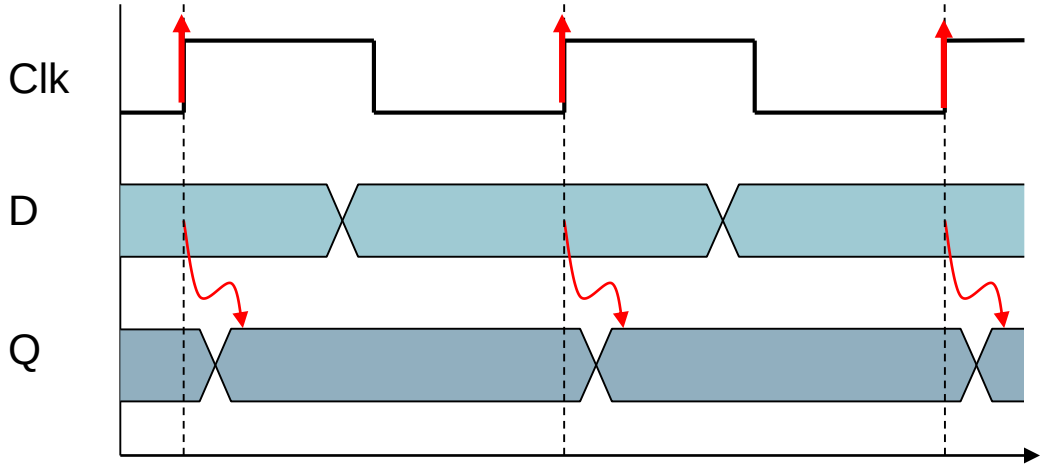
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

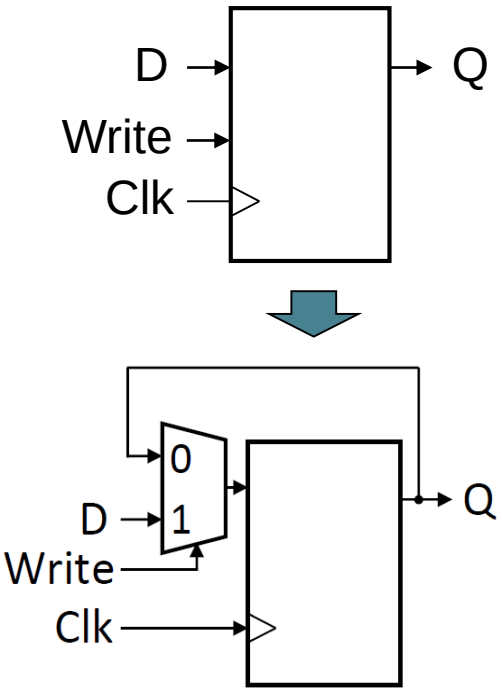
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

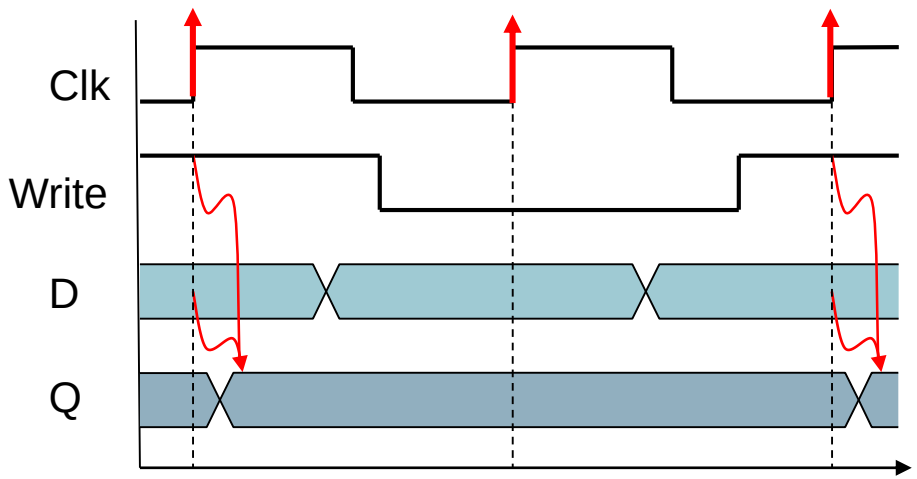




Sequential Elements

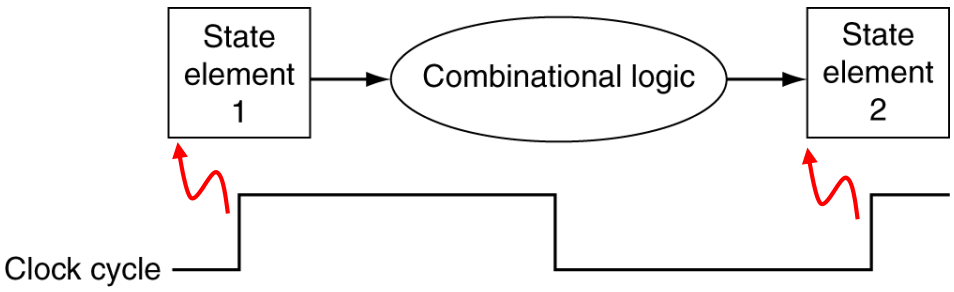
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

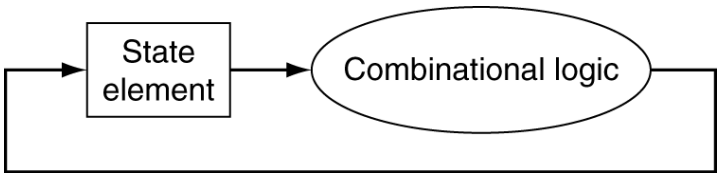




Clocking Methodology

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

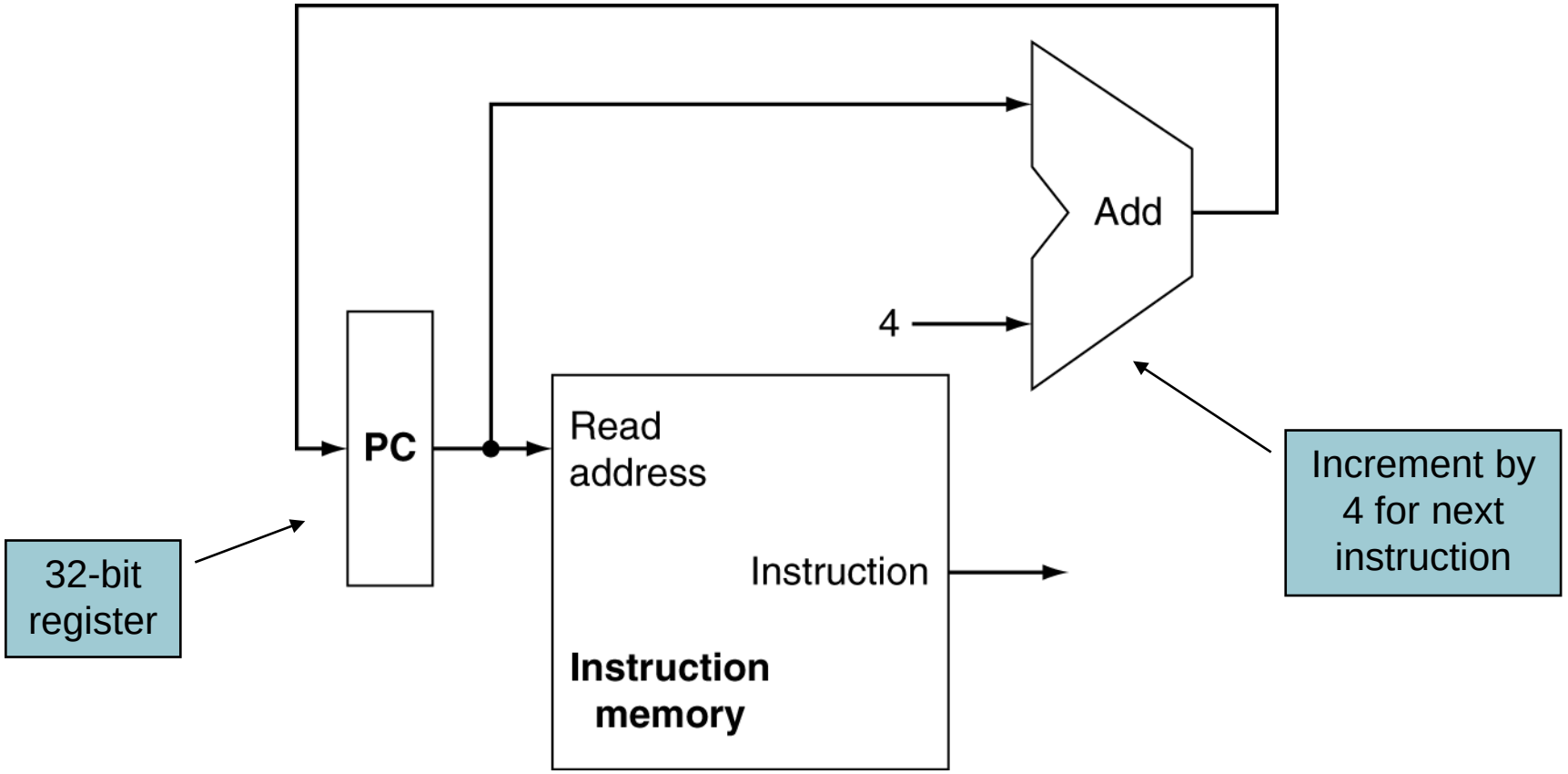




Building a Datapath

- 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

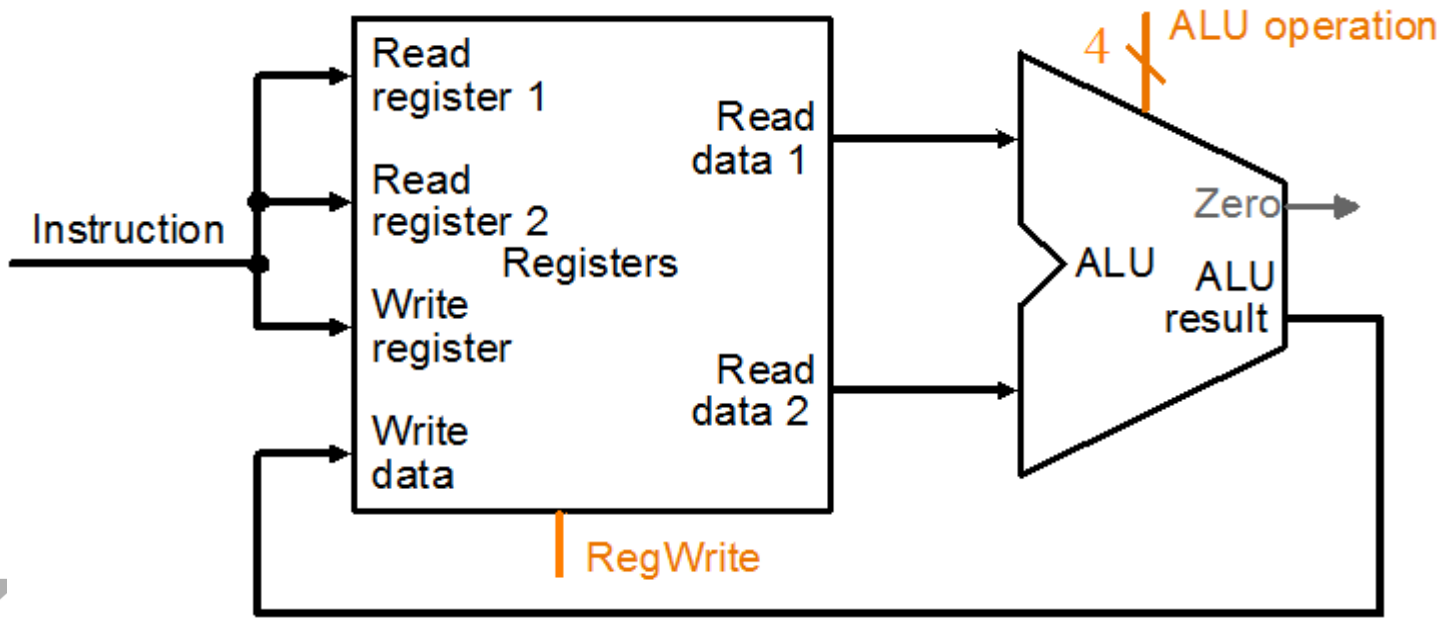


R-Format Instructions

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

add $rd, $rs, $rt

op	rs	rt	rd	shamt	funct

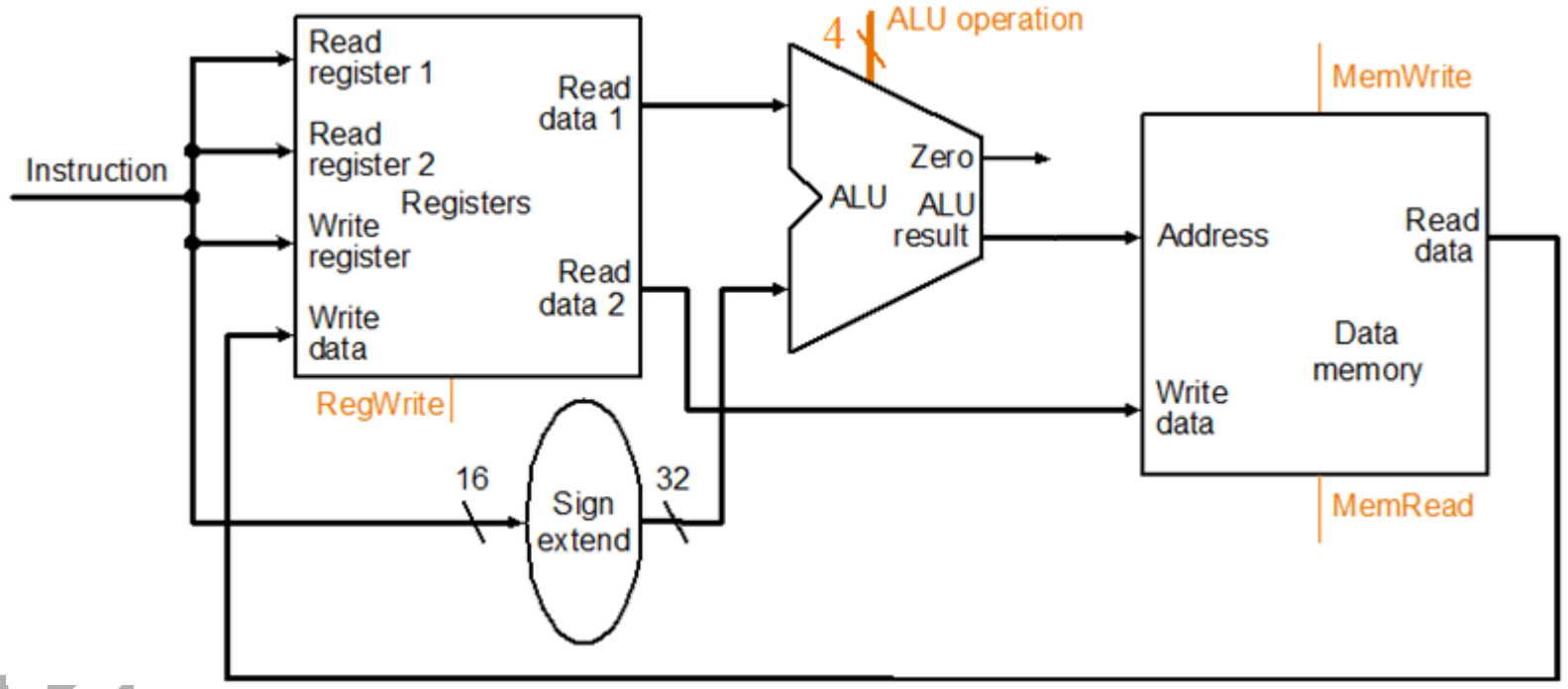


Load/Store Instructions

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

lw $rt, offset($rs)

sw $rt, offset($rs)

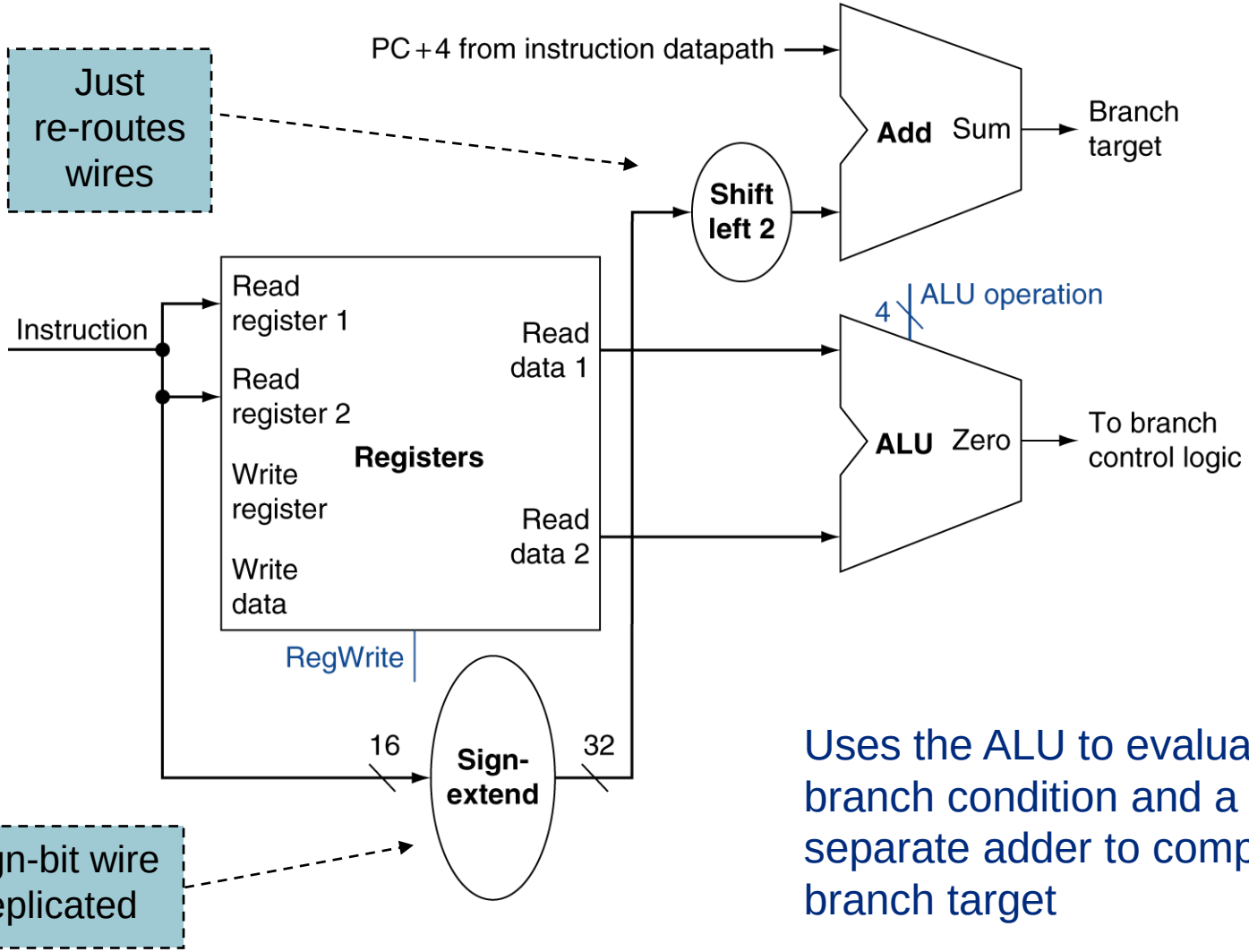


Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

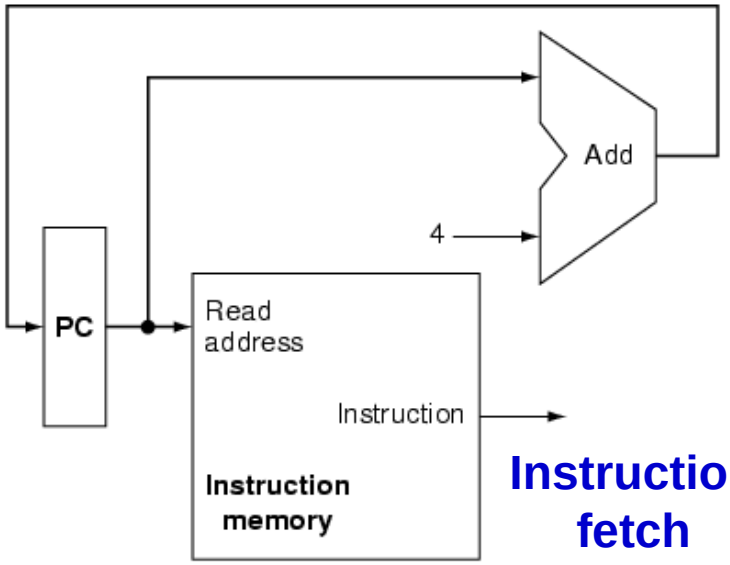
Beq $rs, $rt, L

Branch Instructions

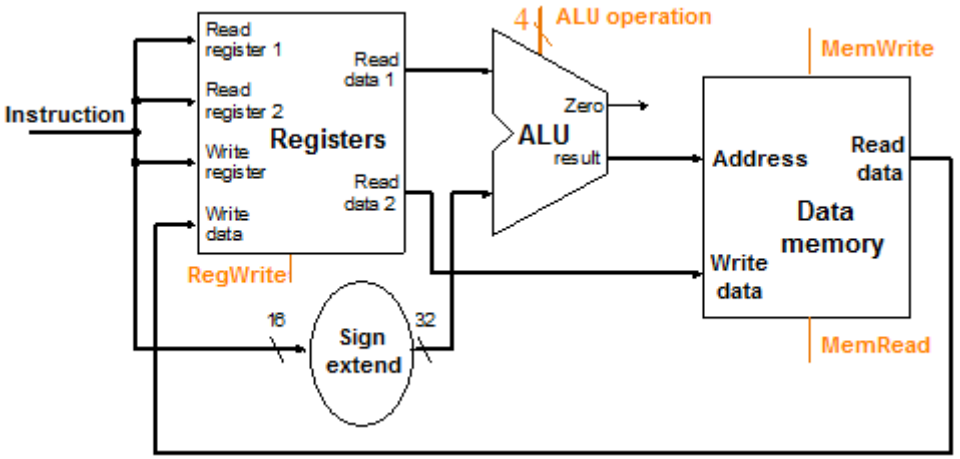


Uses the ALU to evaluate the branch condition and a separate adder to compute the branch target

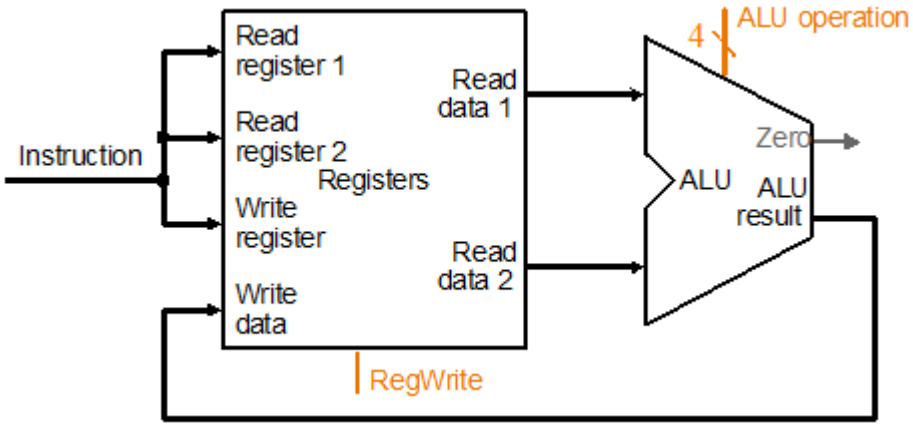
How to combines these together?



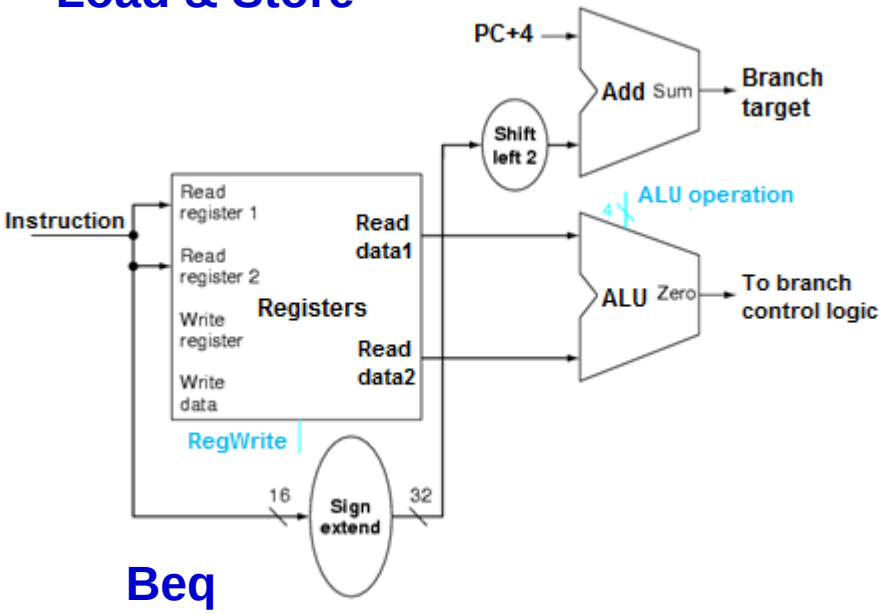
Instruction fetch



Load & Store



R-type

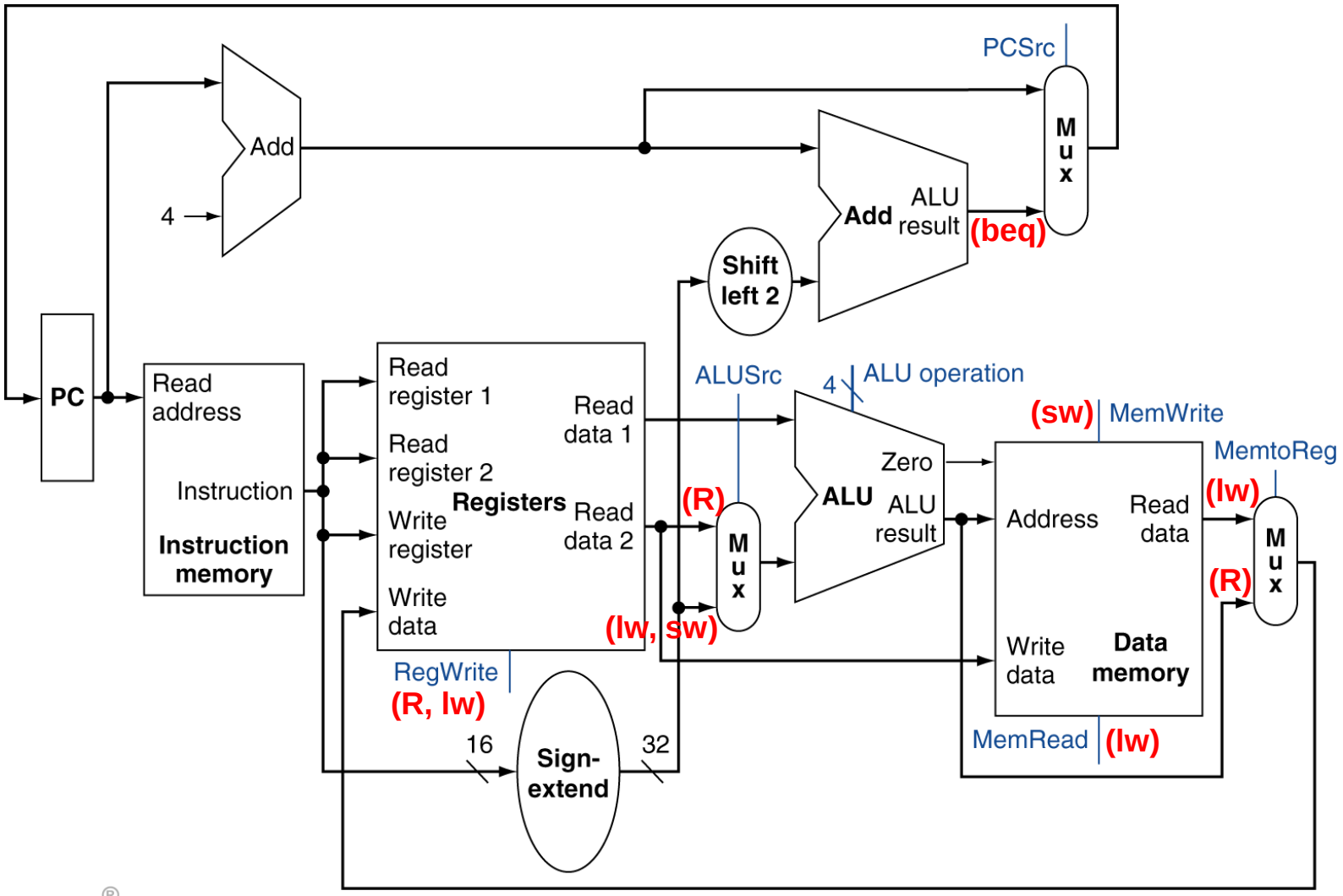


Beq

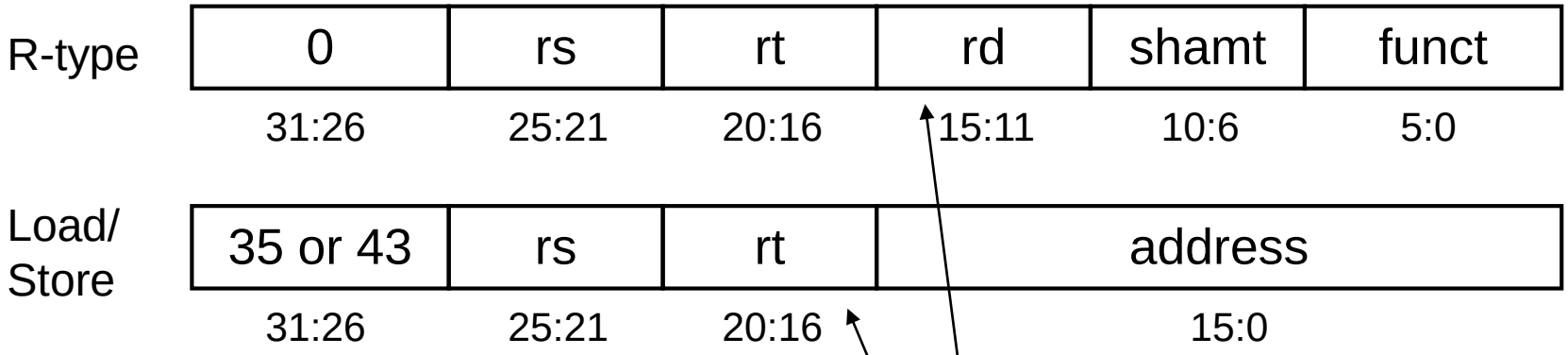
Composing the Elements

- 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

The Datapath with MUXs and Control Signals



Difference in the write-register field



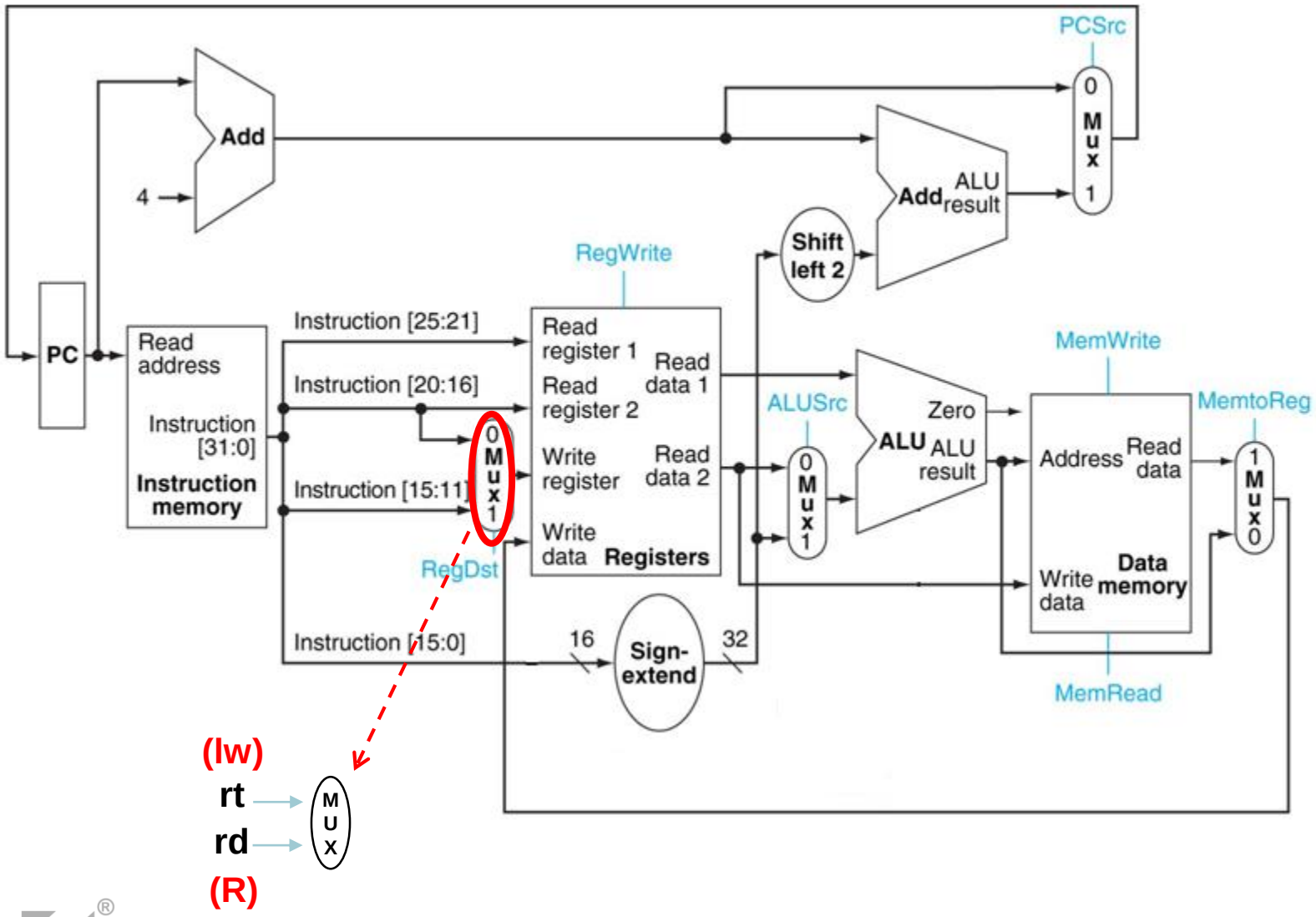
add $t0, $t1, $t2 (rd, rs, rt)

lw $t0, 4($t1) (rt, rs)

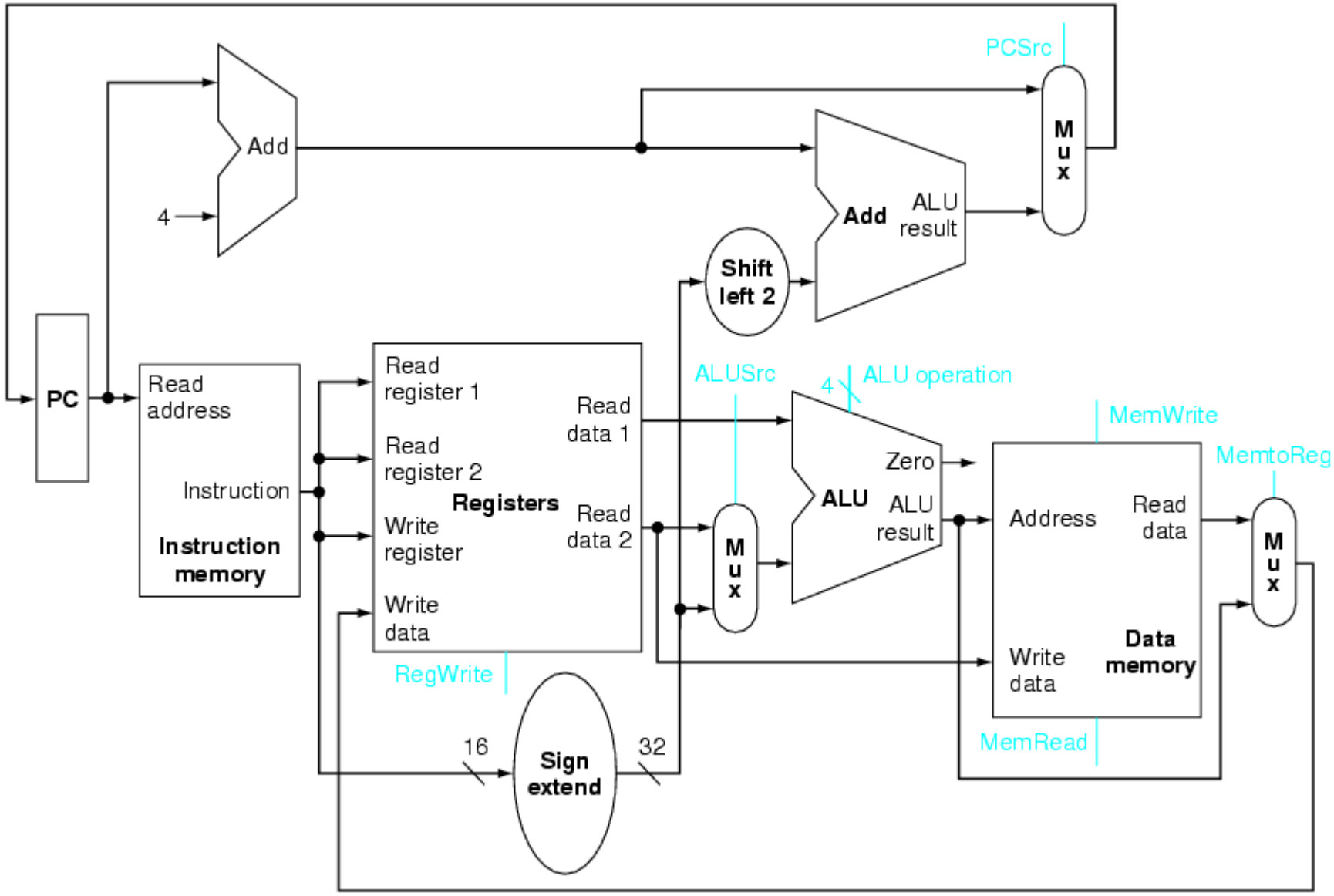
beq $t0, $t1, L (rs, rt)

write for R-type and load

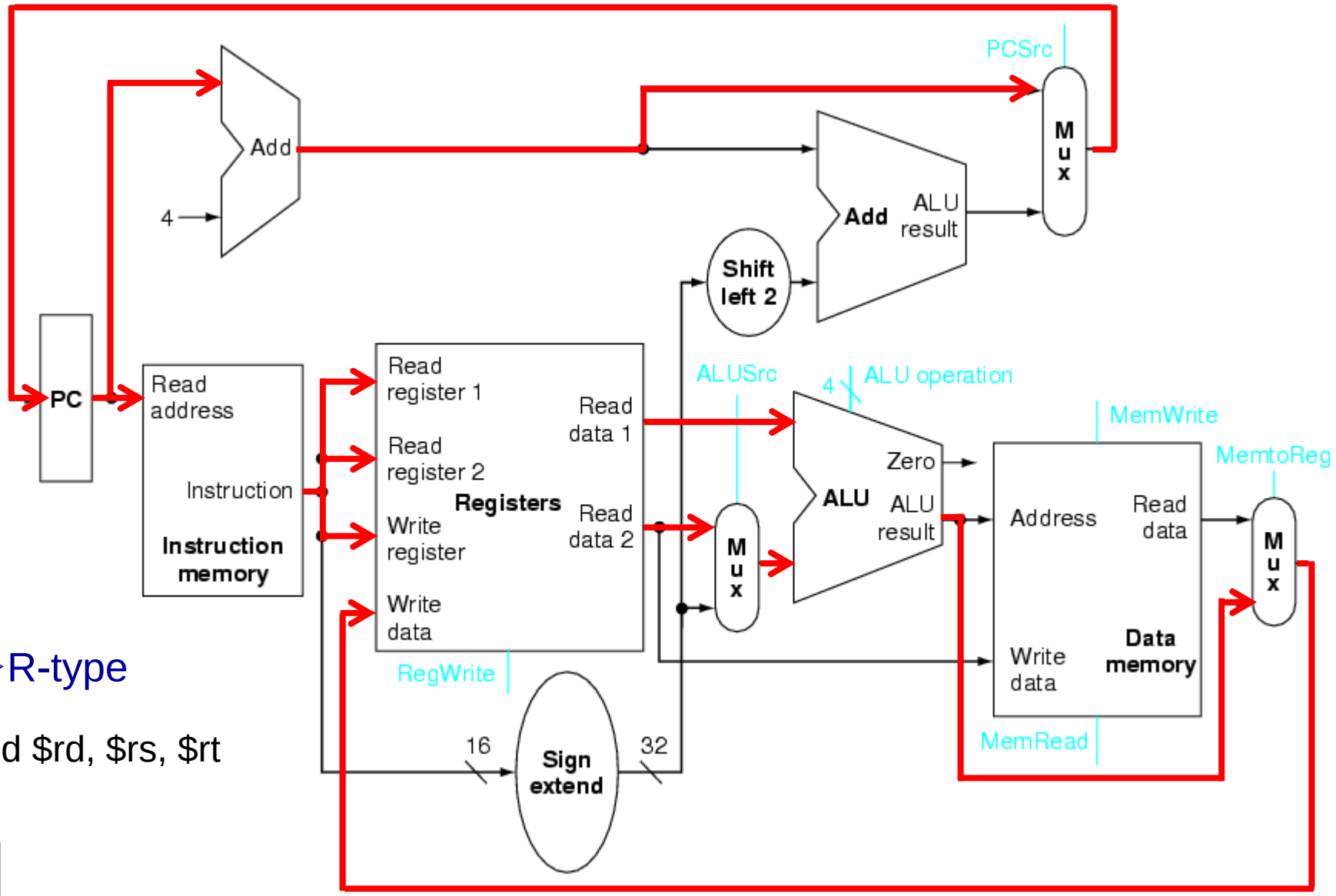
Full Datapath



A Simple Datapath for MIPS Inst. Set



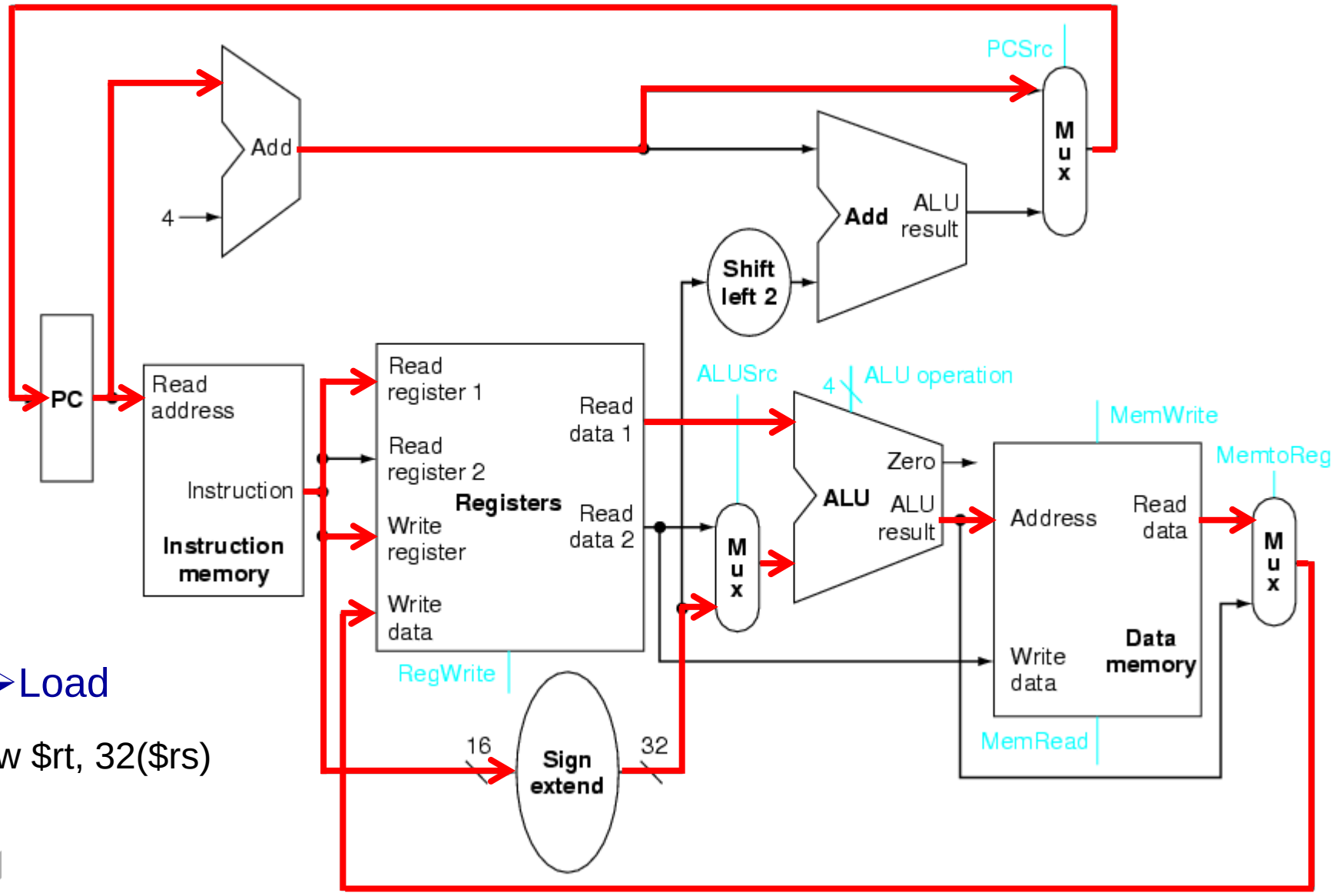
A Simple Datapath for MIPS Inst. Set



- R-type

add $rd, $rs, $rt

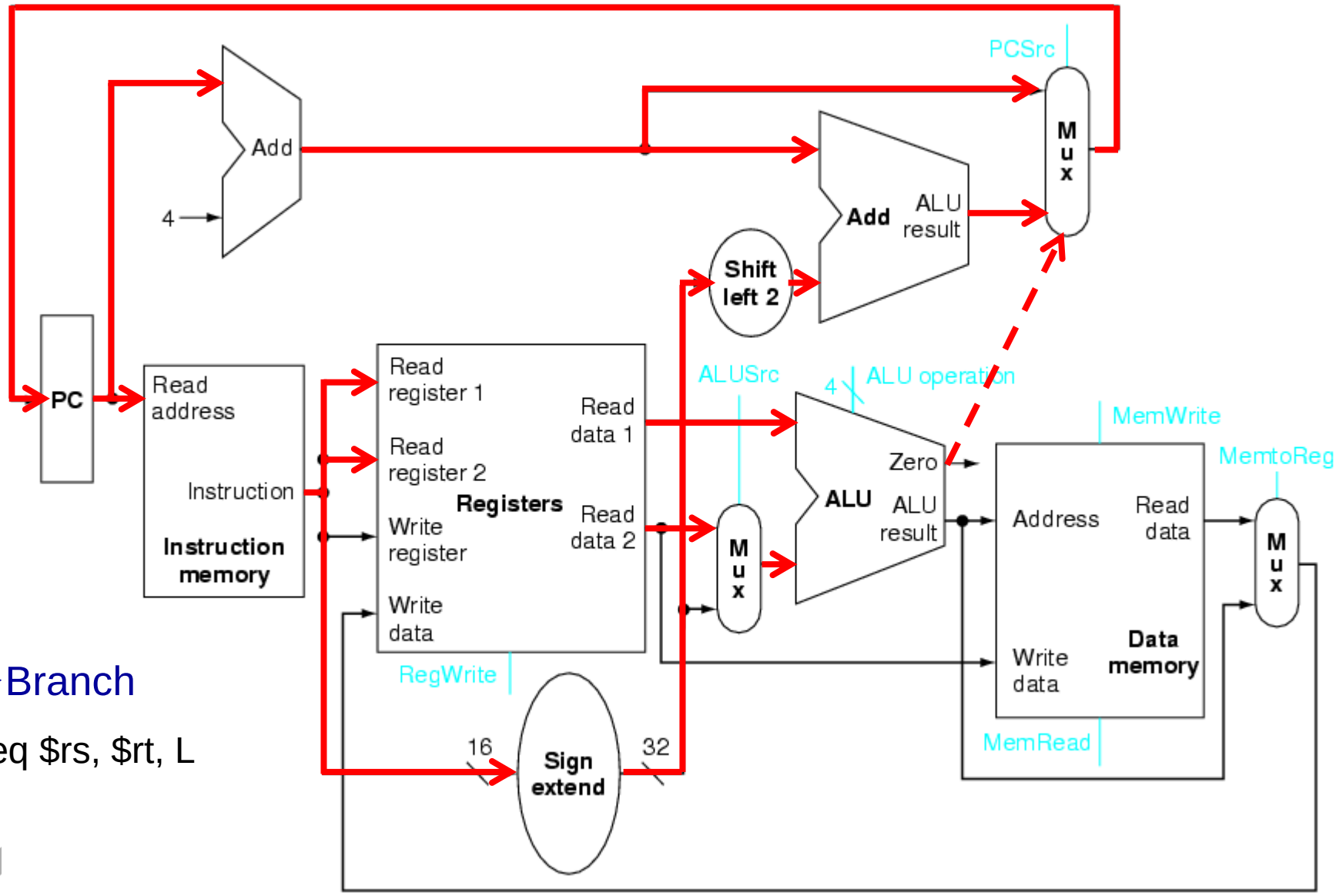
A Simple Datapath for MIPS Inst. Set



- Load

lw $rt, 32($rs)

A Simple Datapath for MIPS Inst. Set



- Branch

beq $rs, $rt, L

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 (slt)
1100	NOR

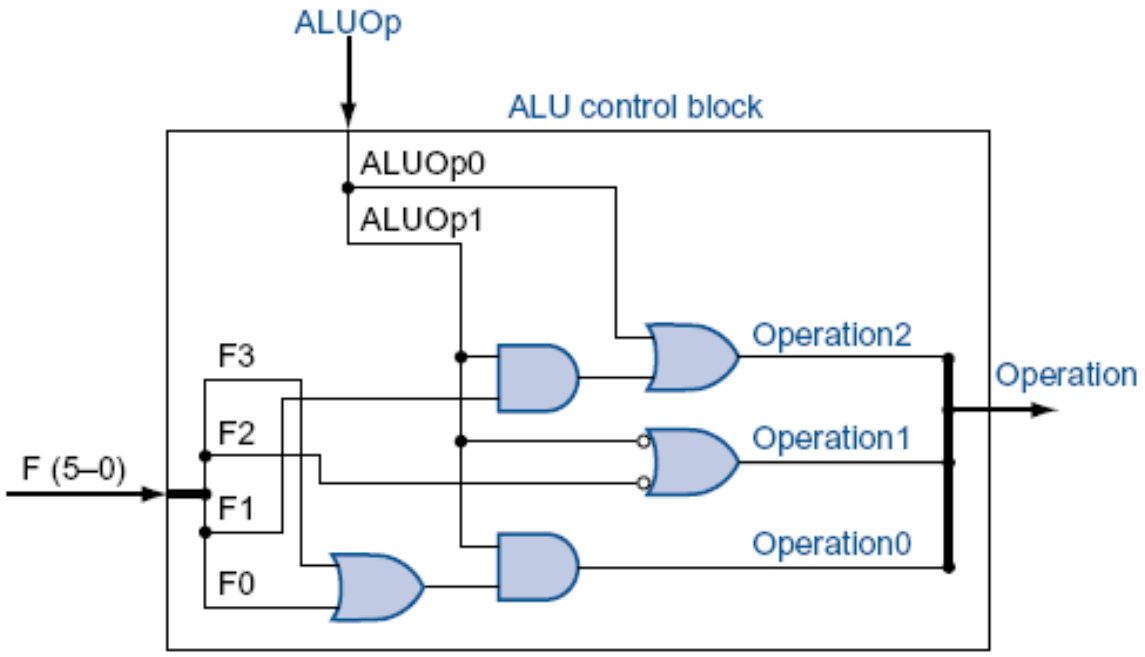
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

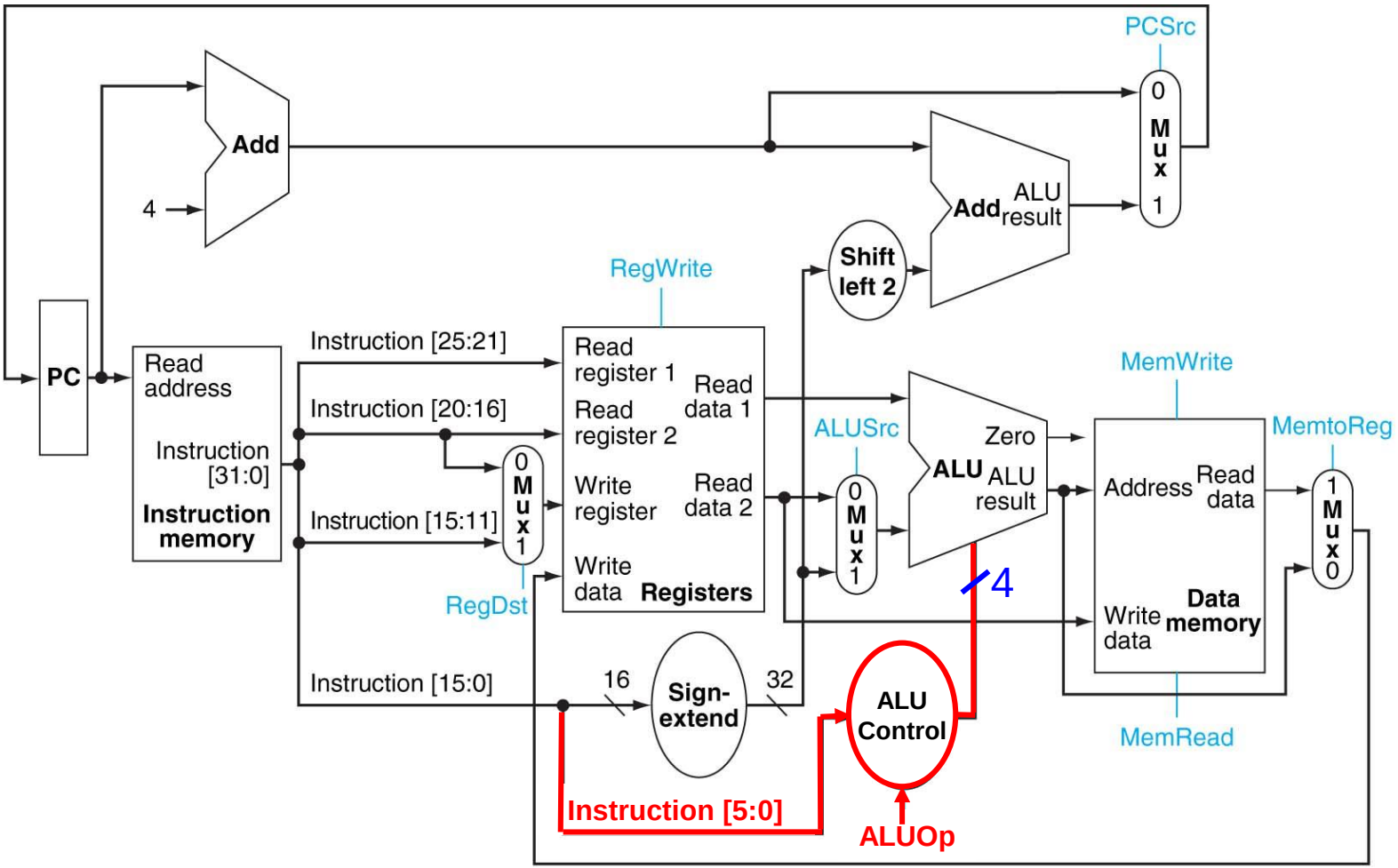
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

ALU Control (Appendix)

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	C_3 C_2 C_1 C_0
0	0	X	X	X	X	X	X	0010 (lw/sw)
X	1	X	X	X	X	X	X	0110 (beq)
1	X	X	X	0	0	0	0	0010 (add)
1	X	X	X	0	0	1	0	0110 (sub)
1	X	X	X	0	1	0	0	0000 (AND)
1	X	X	X	0	1	0	1	0001 (OR)
1	X	X	X	1	0	1	0	0111 (slt)

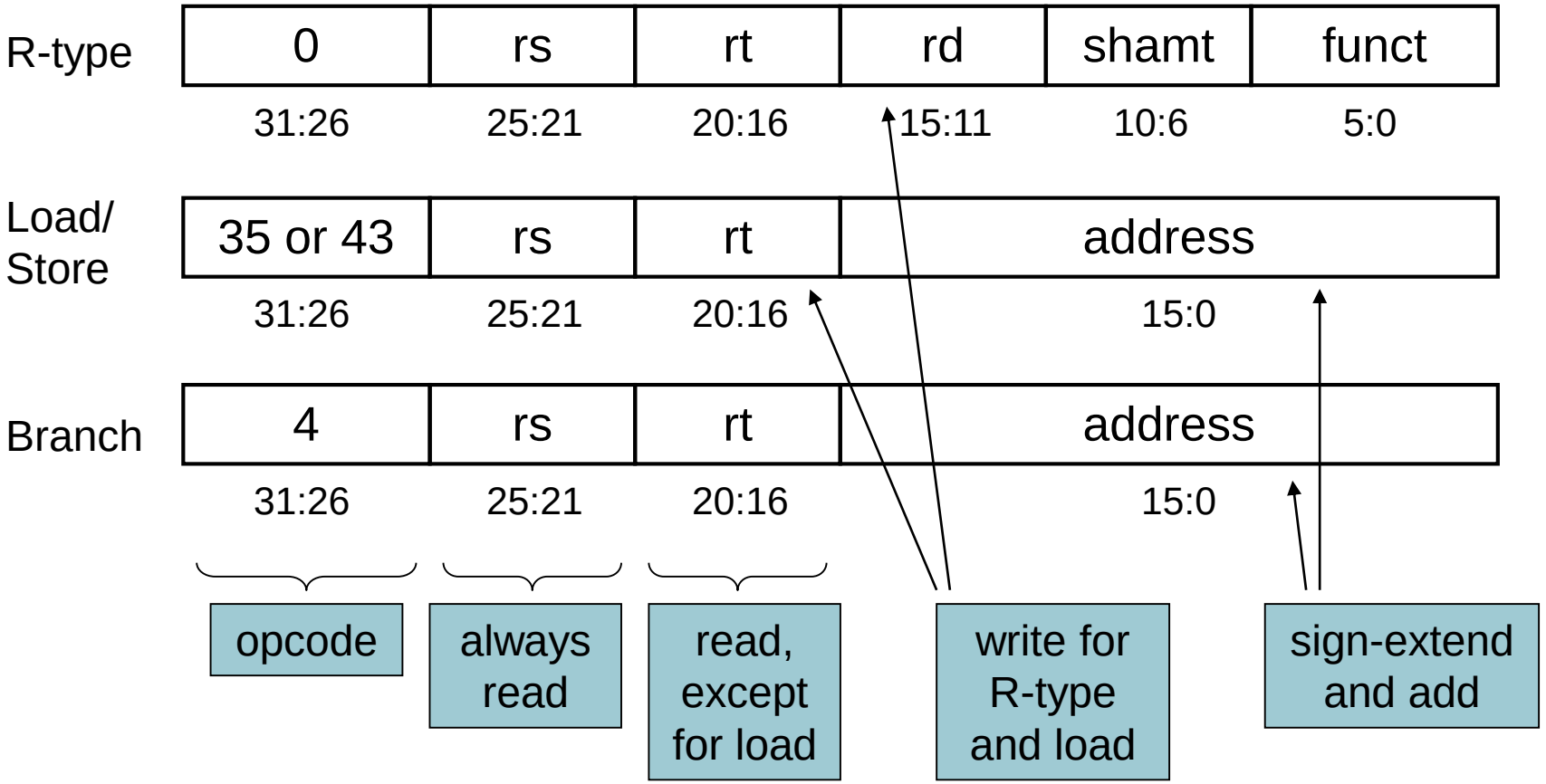


The ALU Control



The Main Control Unit

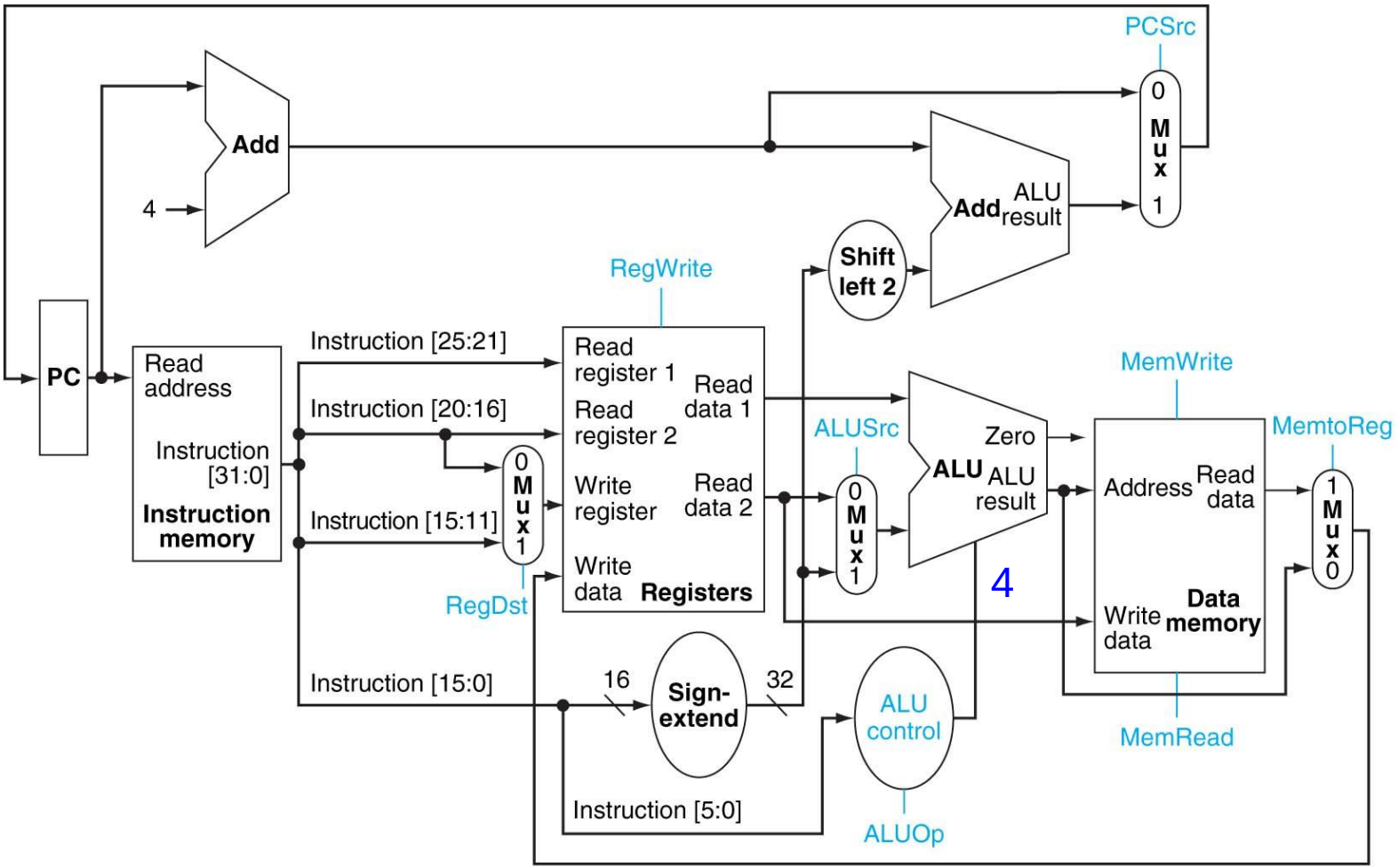
- Control signals derived from instruction



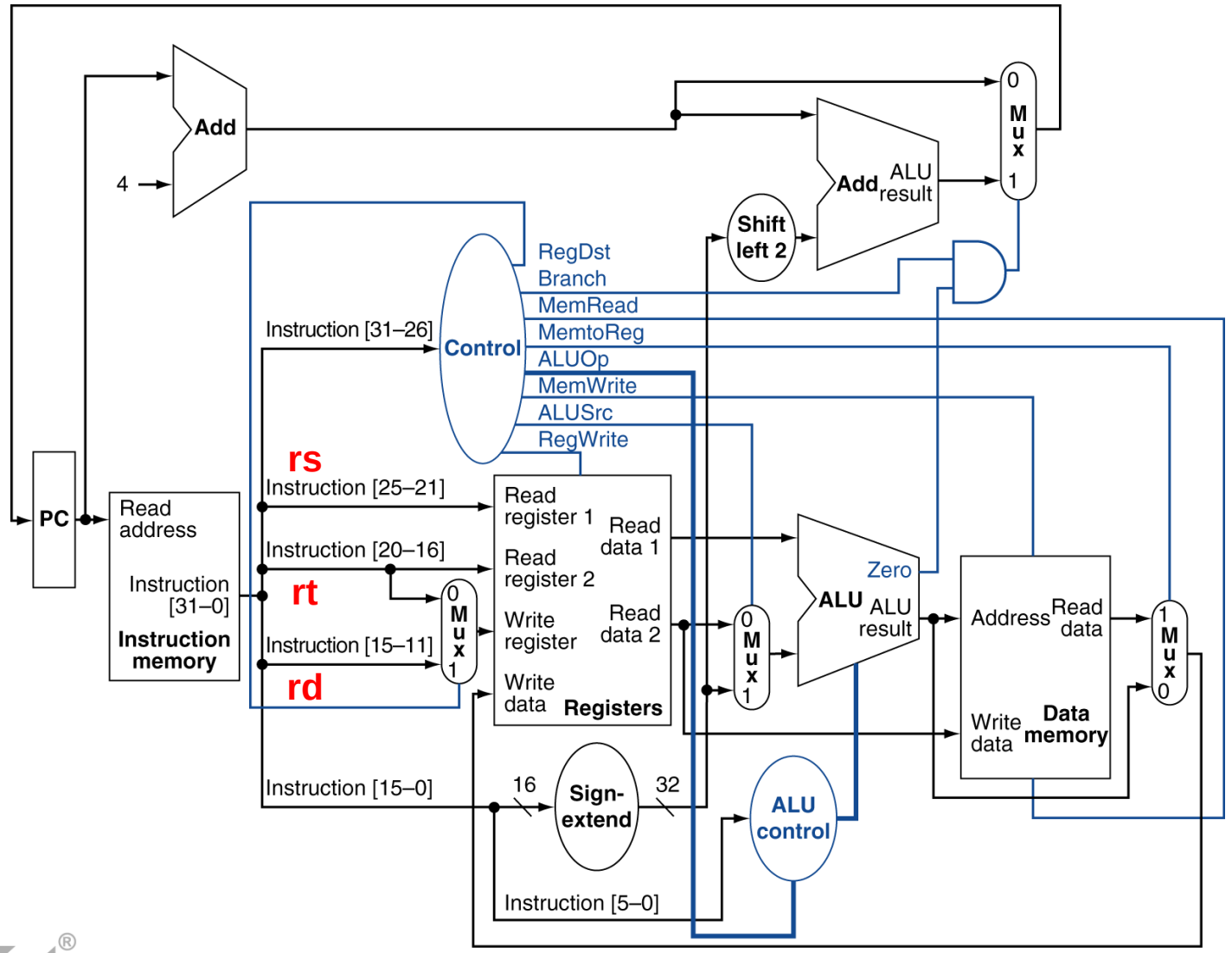
The Main Control Unit

Signal name	(set to 0) Effect when deasserted	(set to 1) Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

The ALU Control

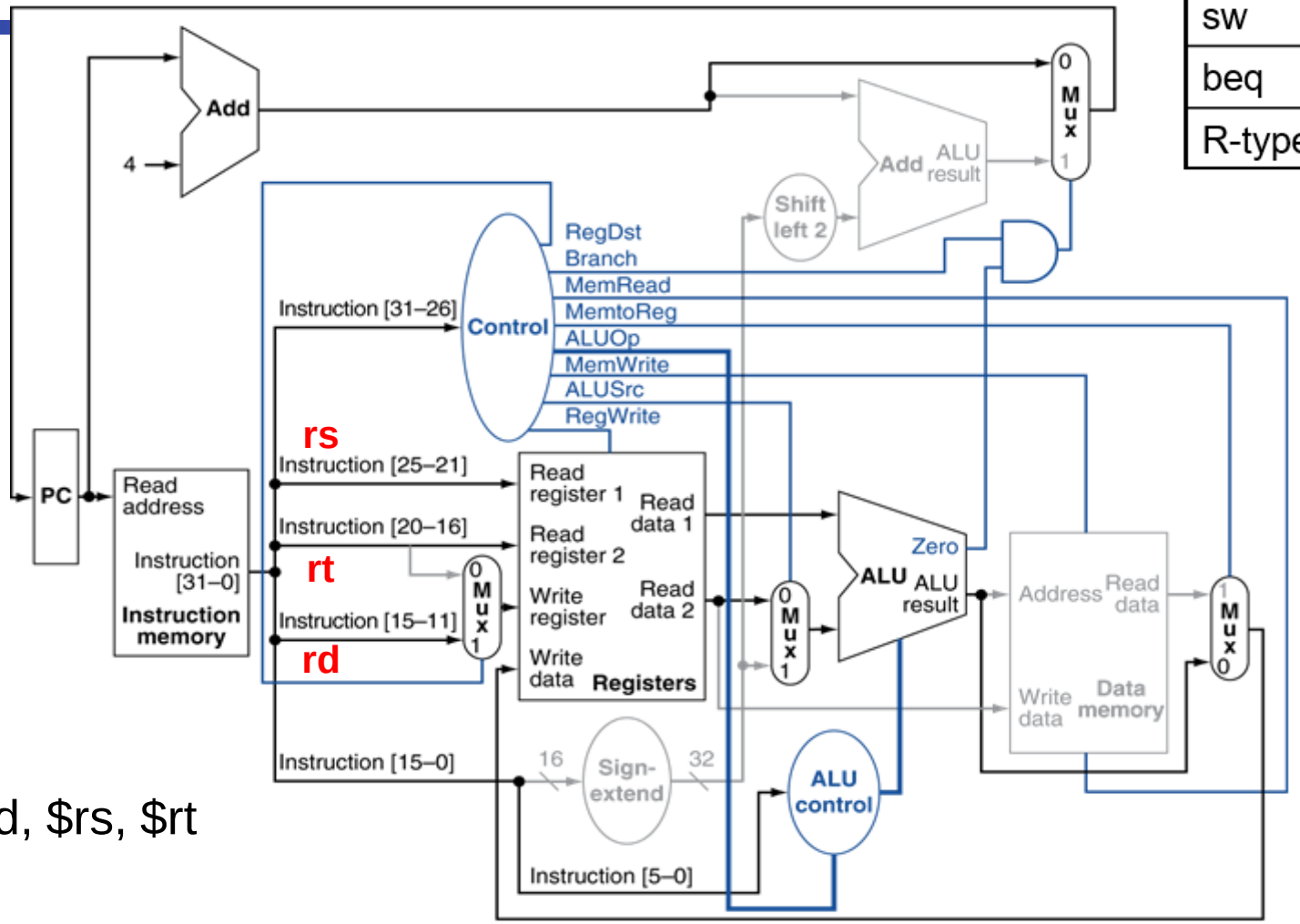


Datapath With Control



R-Type Instruction

opcode	ALUOp
lw	00
sw	00
beq	01
R-type	10

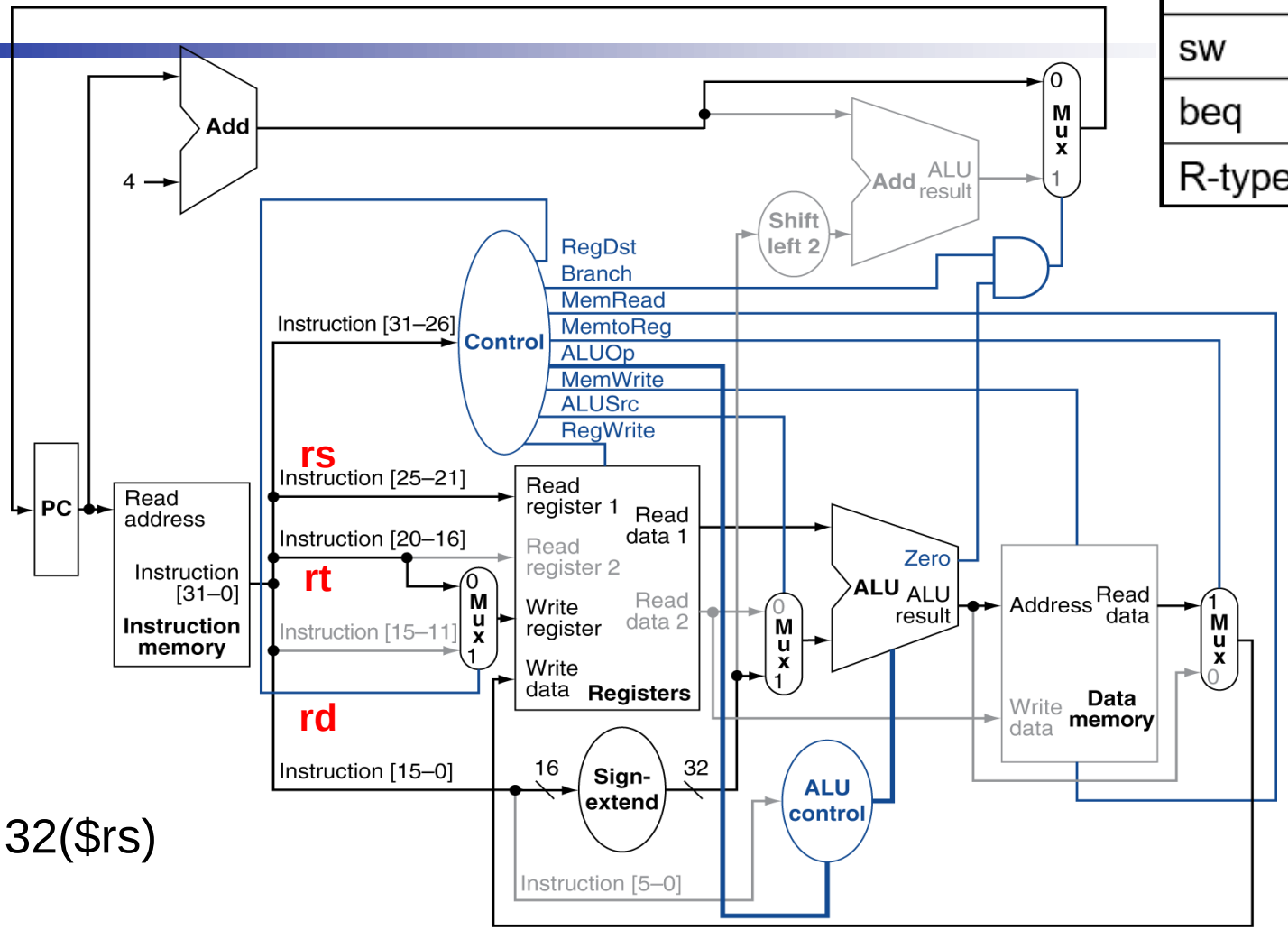


add $rd, $rs, $rt

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0

Load Instruction

opcode	ALUOp
lw	00
sw	00
beq	01
R-type	10

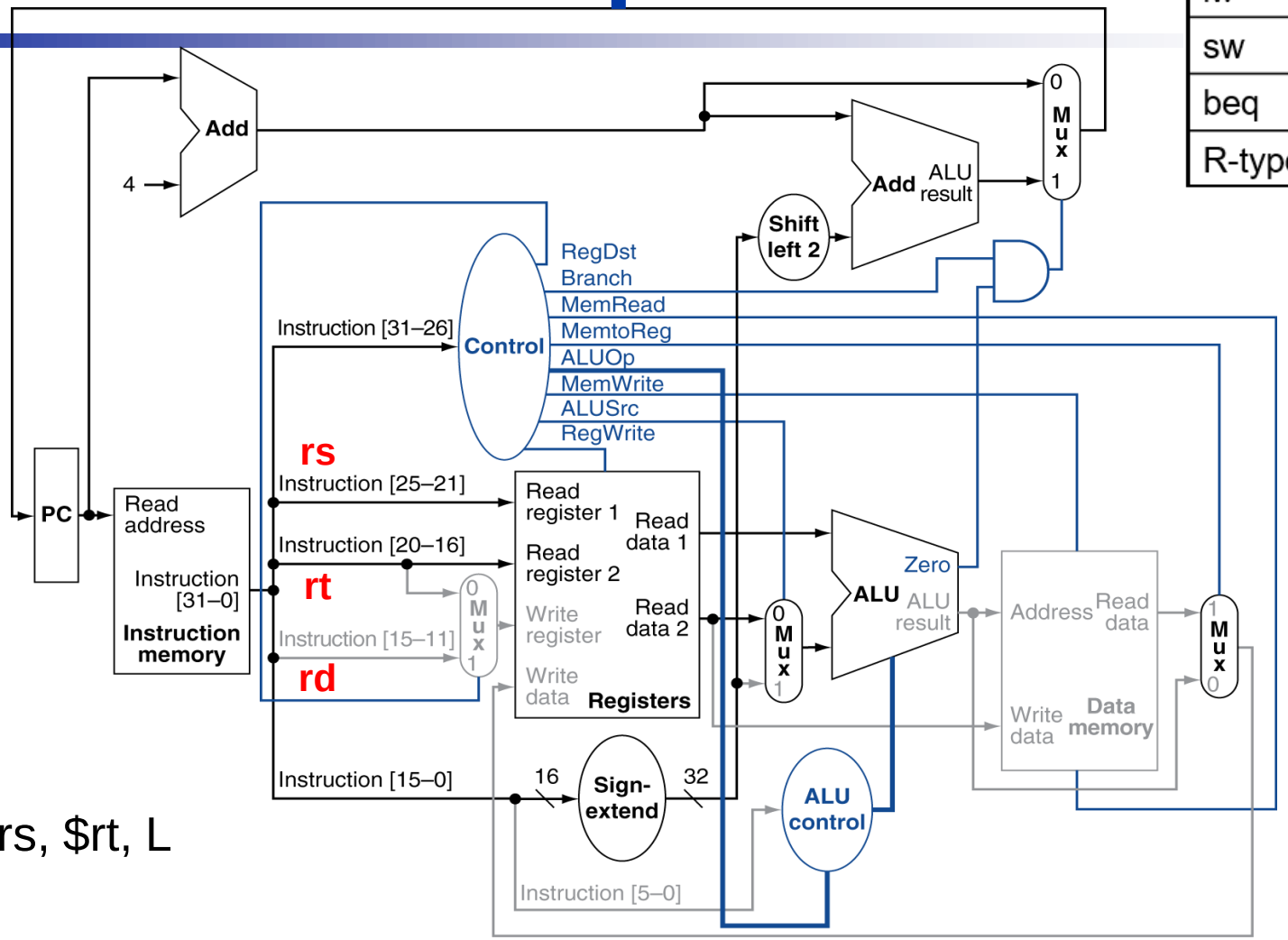


lw $rt, 32($rs)

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
lw	0	1	1	1	1	0	0	0	0

Branch-on-Equal Instr.

opcode	ALUOp
lw	00
sw	00
beq	01
R-type	10



beq $rs, $rt, L

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
beq	x	0	x	0	0	0	1	0	1

The Main Control Unit

- The setting of the control lines:

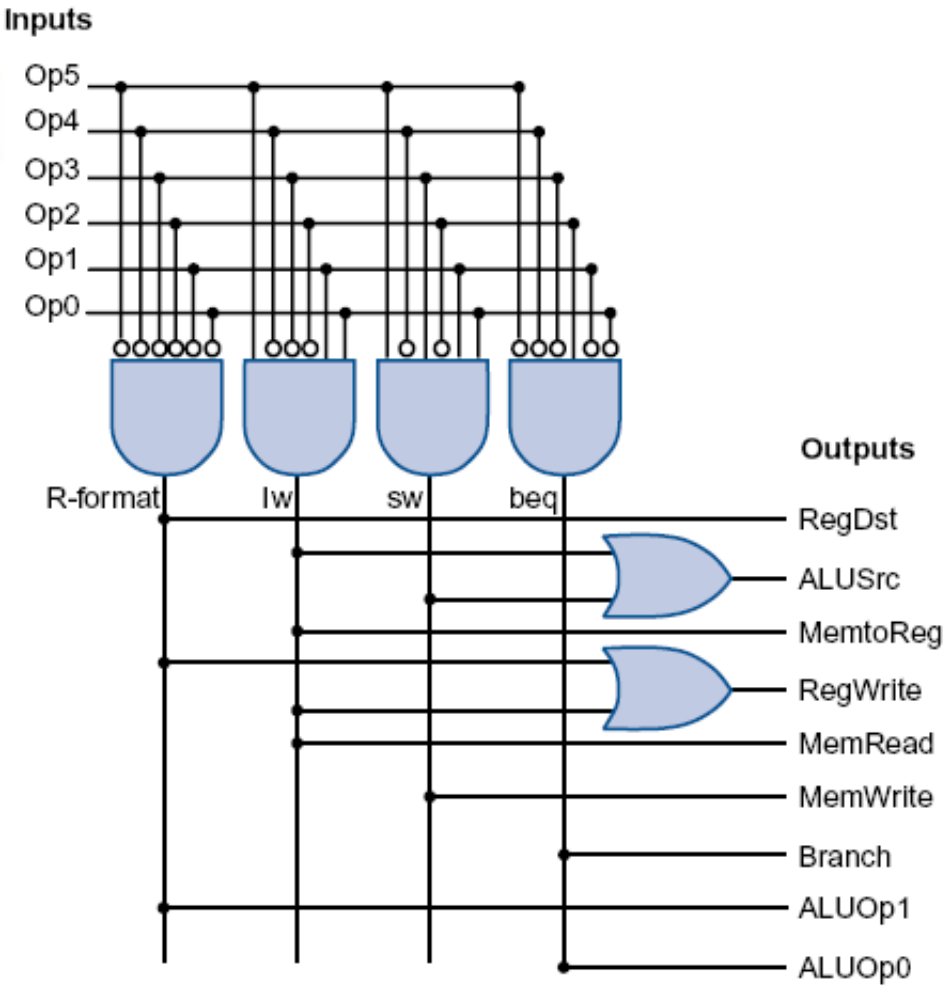
Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- The encoding for each of the opcodes of interest:

Name	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0_{ten}	0	0	0	0	0	0
lw	35_{ten}	1	0	0	0	1	1
sw	43_{ten}	1	0	1	0	1	1
beq	4_{ten}	0	0	0	1	0	0

Main Control Unit (Appendix)

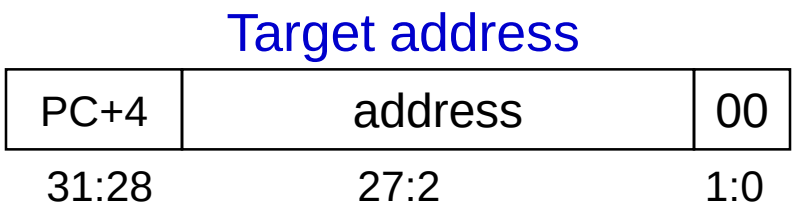
	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



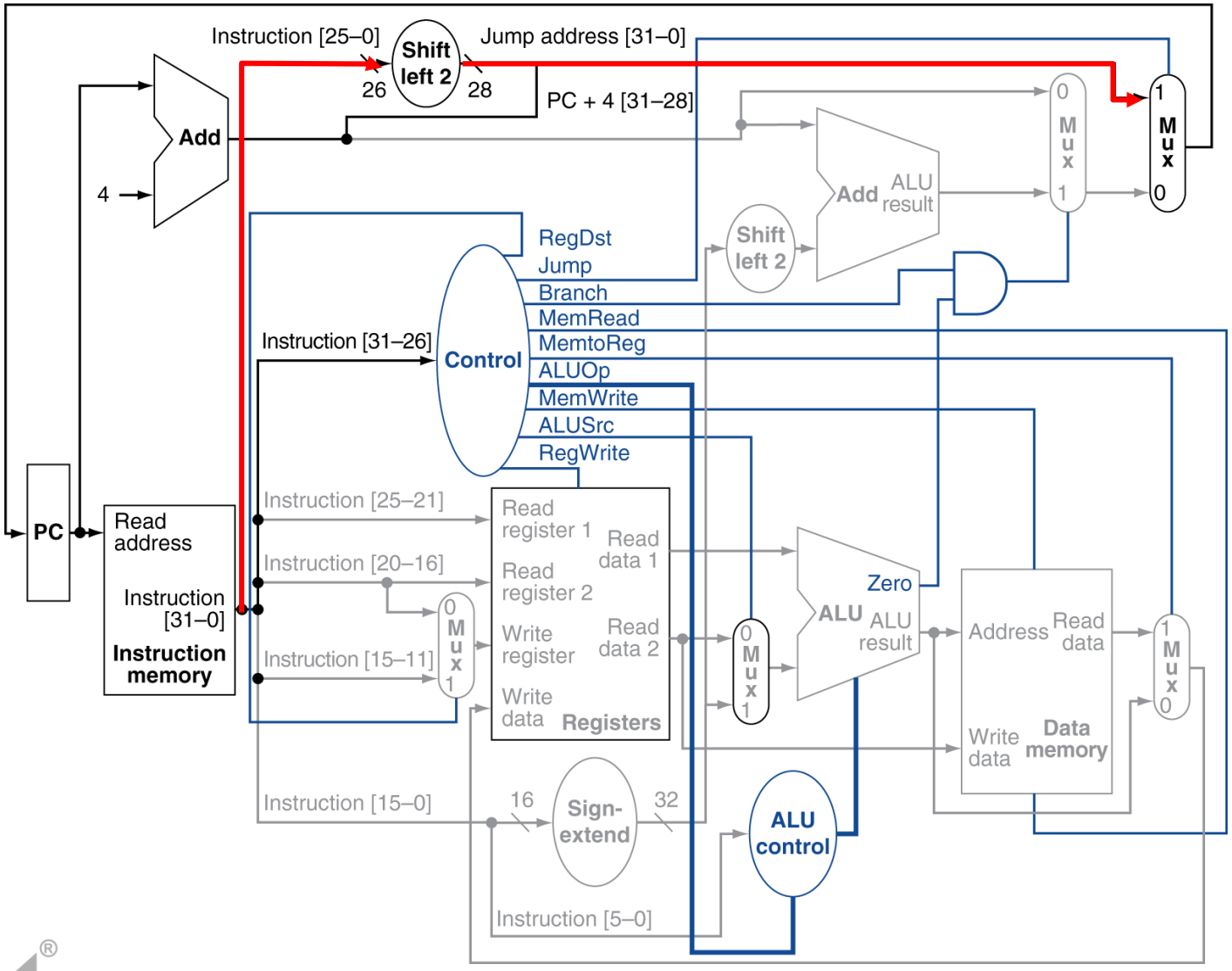
Implementing Jumps

Jump Instruction (opcode=2)

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode



Datapath With Jumps Added

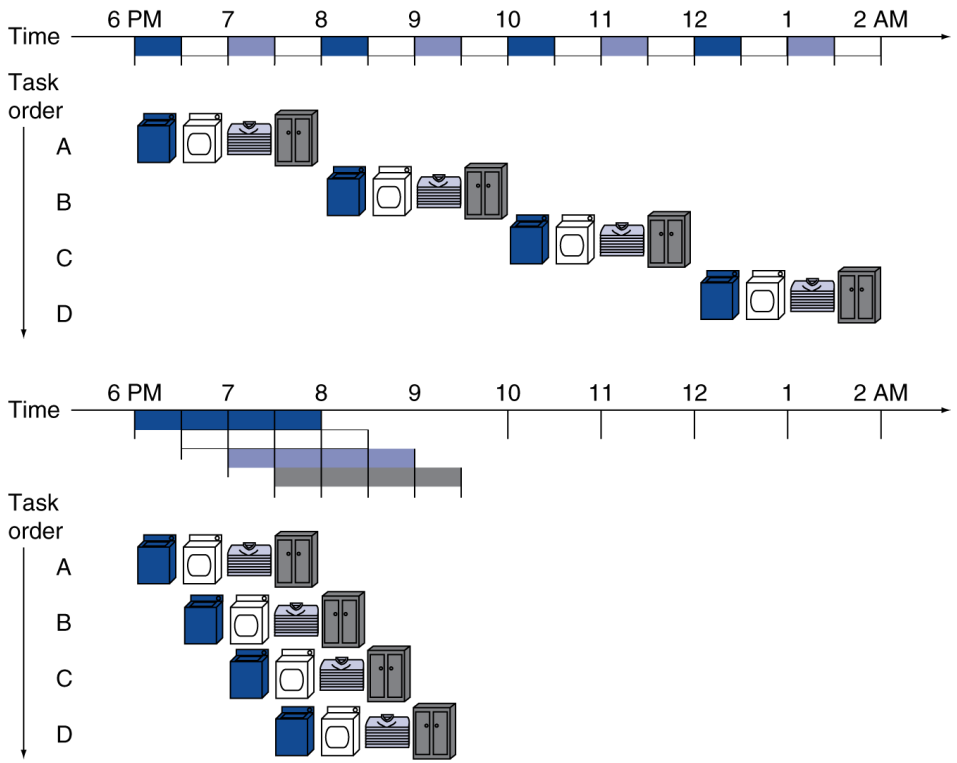


Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → 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:
  - Speedup
    = 2n/(0.5n + 1.5) ≈ 4
    = number of stages

MIPS Pipeline

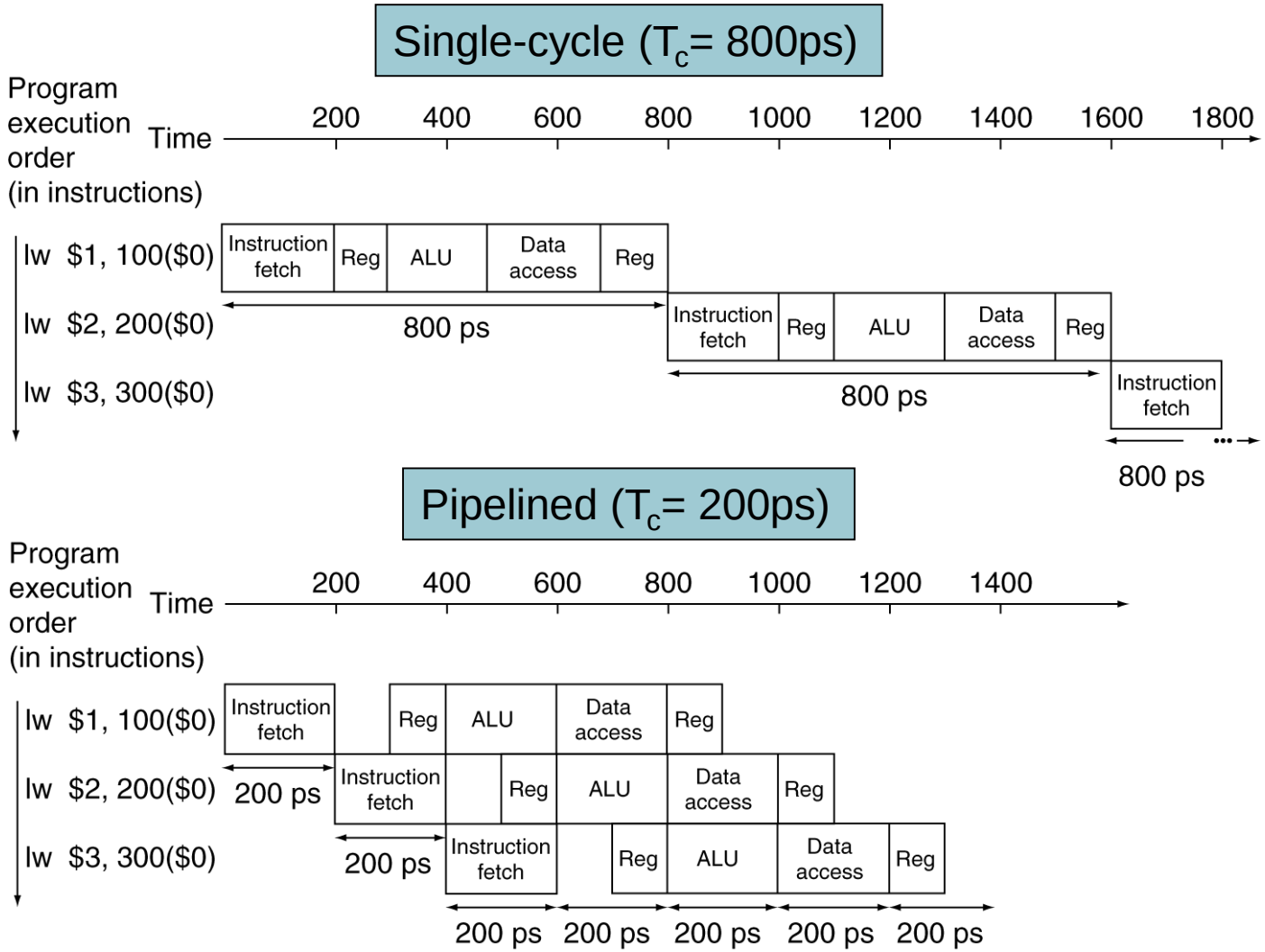
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

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

	IF	ID	EX	MEM	WB	
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 Speedup

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

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch 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

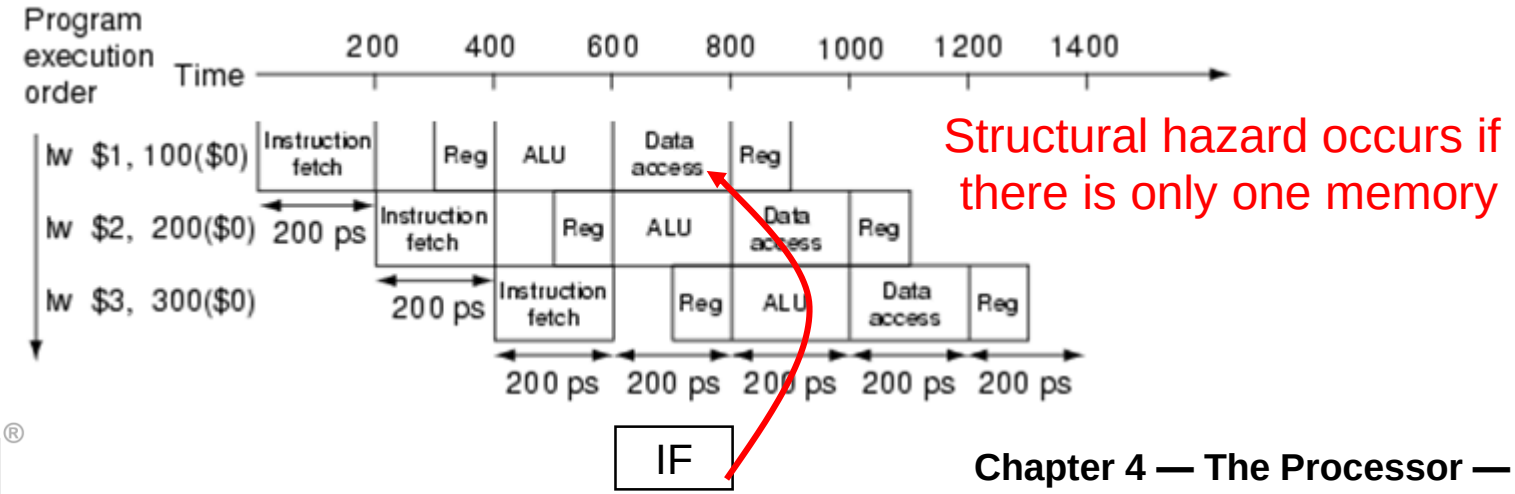
Hazards

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

Structure Hazards

- Conflict for use of a resource
  - hardware cannot support the combination of instruction in the same clock.
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle

➔ require separate instr./data memories (or caches)



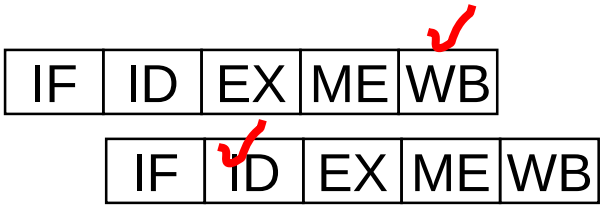
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - data that is needed to execute the instruction is still in the pipeline, not yet available.

- Ex. 1

  add $s0, $t0, $t1

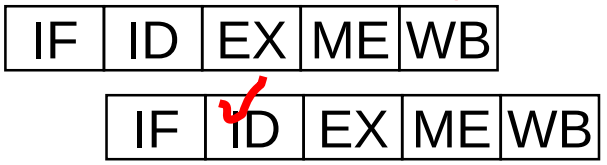
  sub $t2, $s0, $t3



Ex. 2 (load-use data hazard)

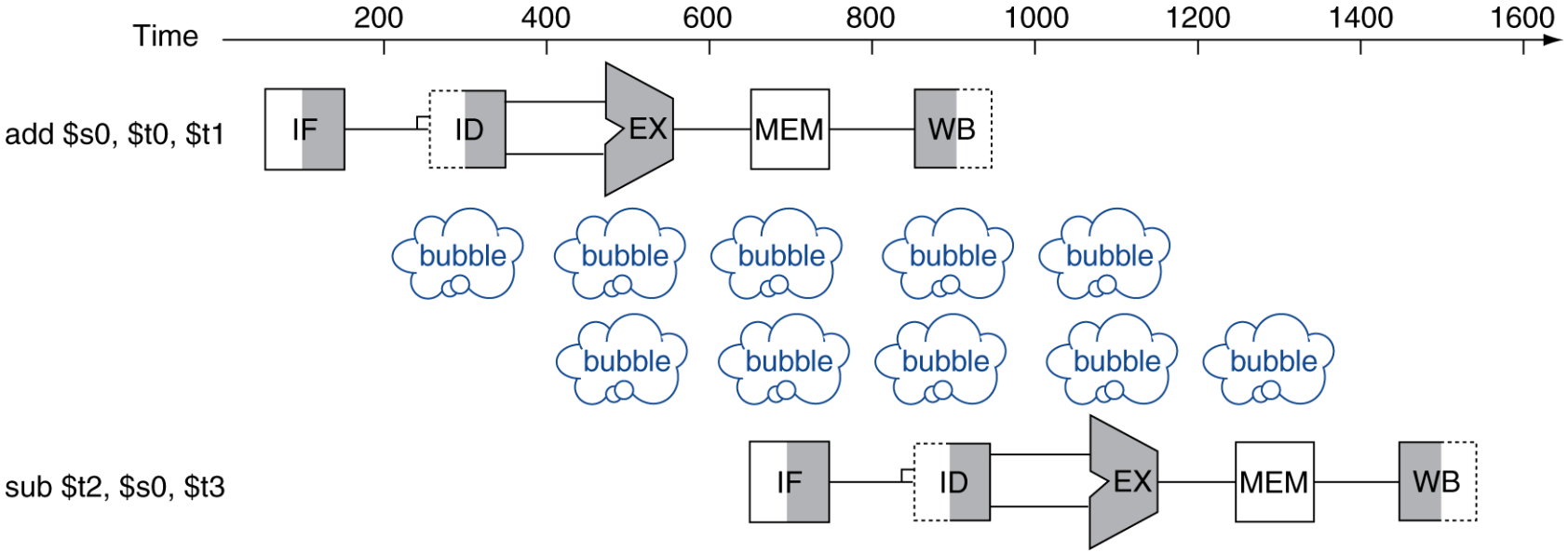
lw $s0, 4($t1)

sub $t2, $s0, $t3



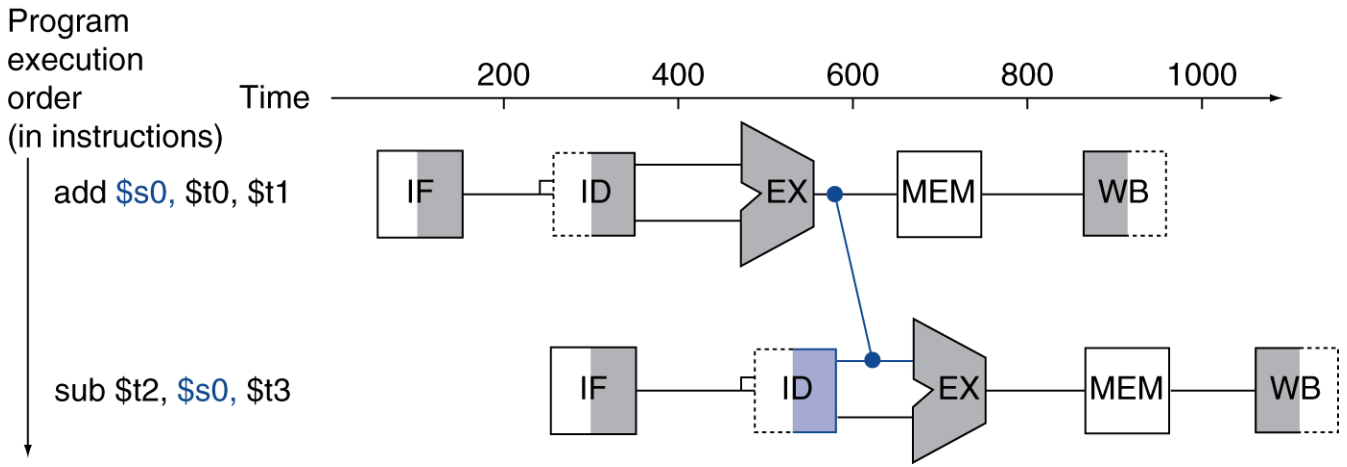
Data Hazards - stall

- Stall the instructions (bubble)
  - `add  $s0, $t0, $t1`
    `sub  $t2, $s0, $t3`



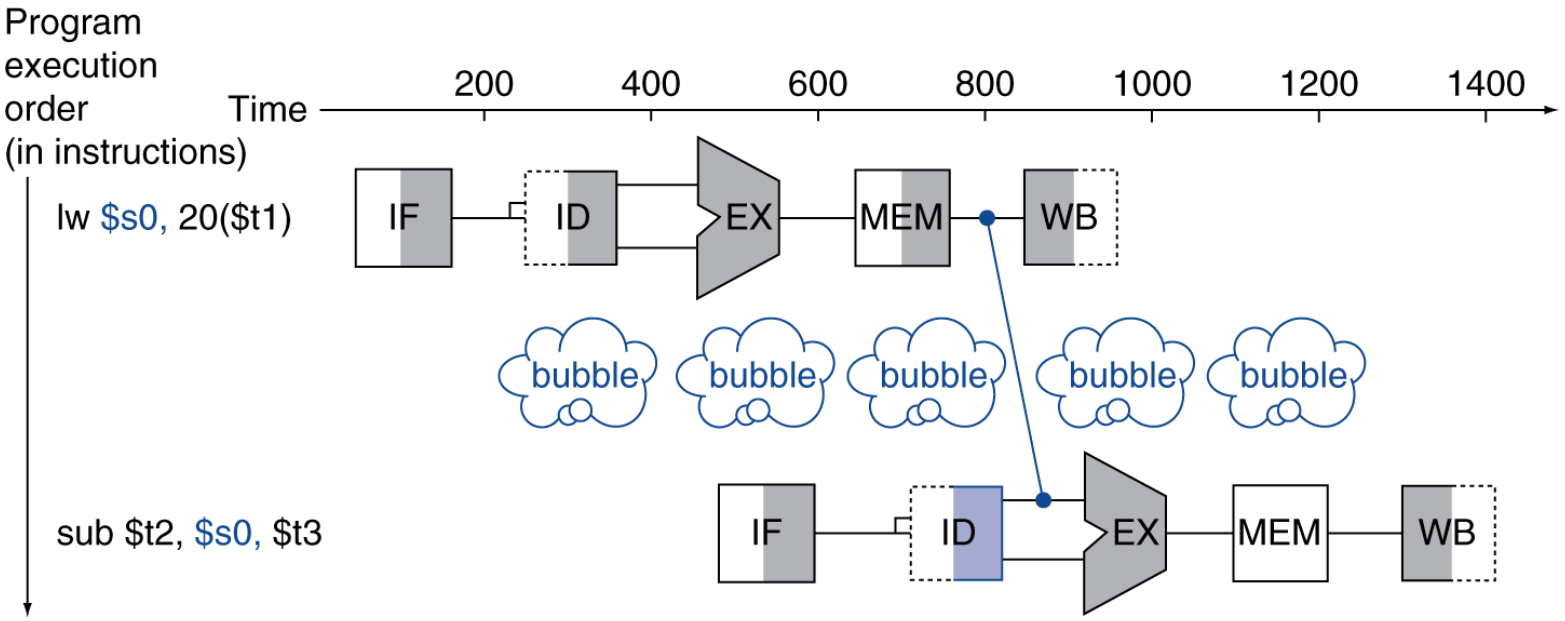
Data Hazards - Forwarding

- 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

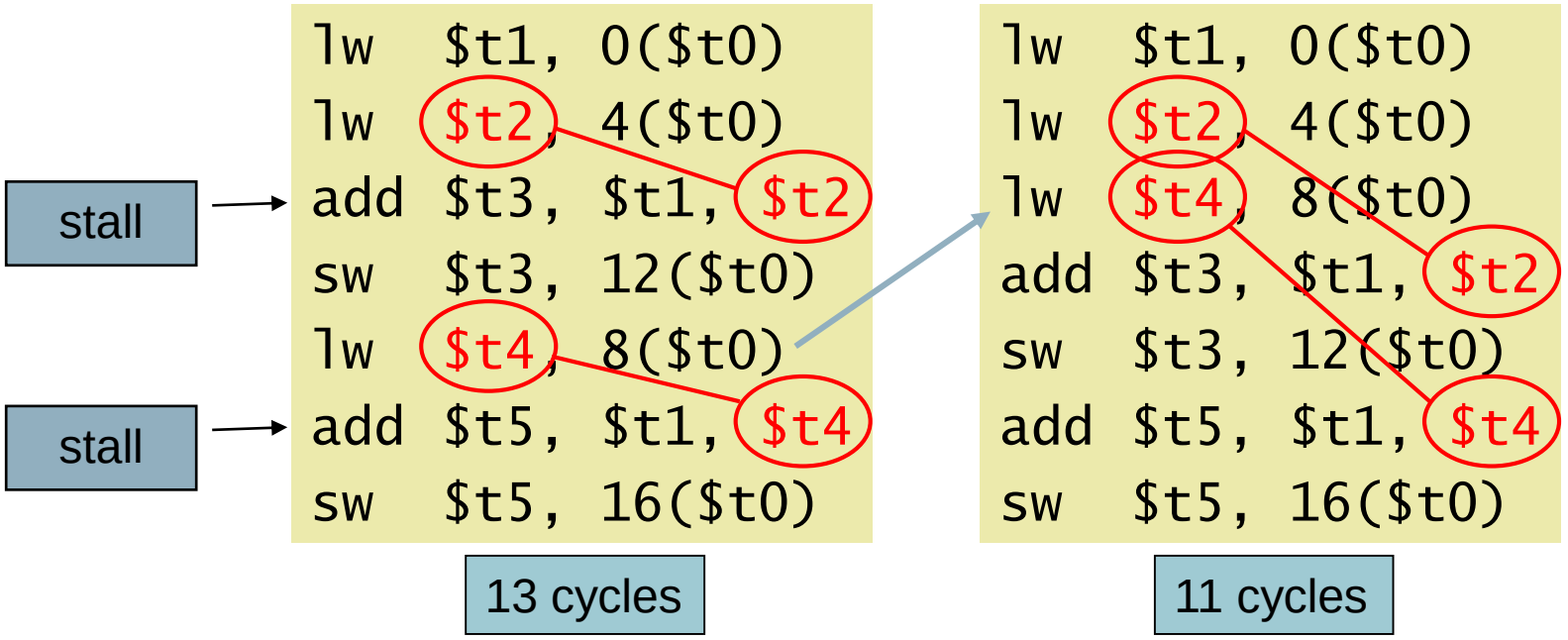
- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!



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;`

`A` → [3], `B` → [0], `E` → [1], `C` → [4], `B` → [0], `F` → [2]

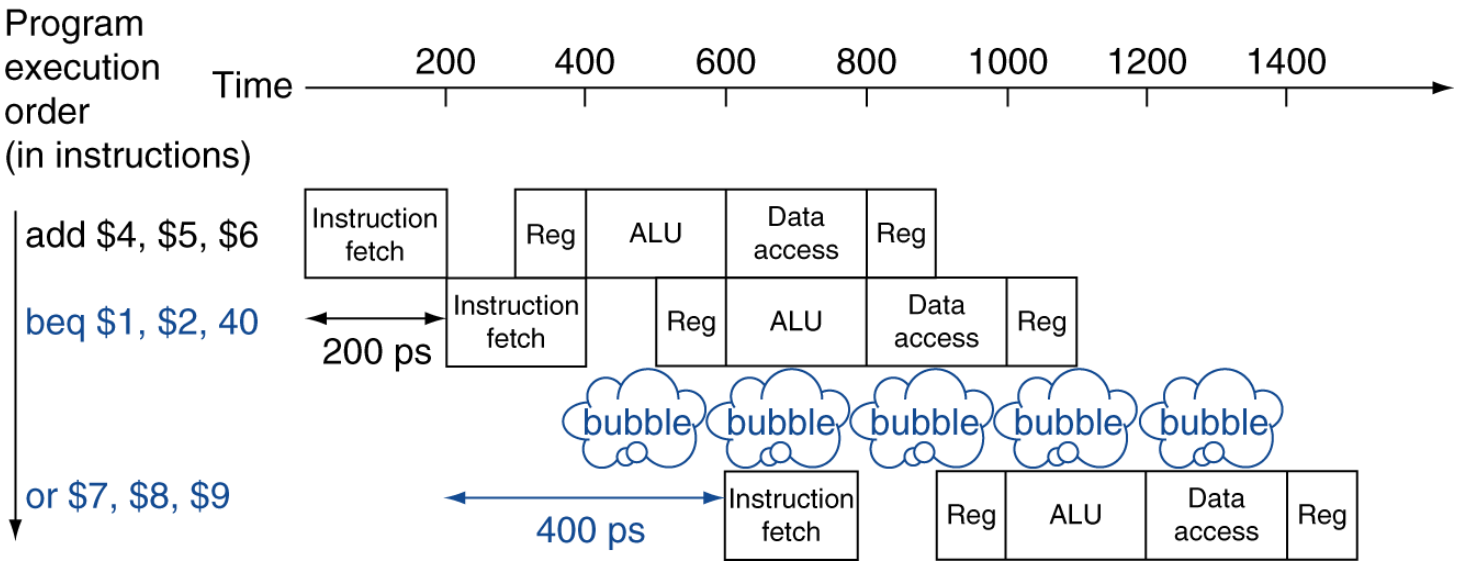


Control Hazards

- 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 (however, the branch outcome is determined at EX stage.)
- 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
  - Assume extra hardware has been added to test registers and calculate branch address at **ID stage**

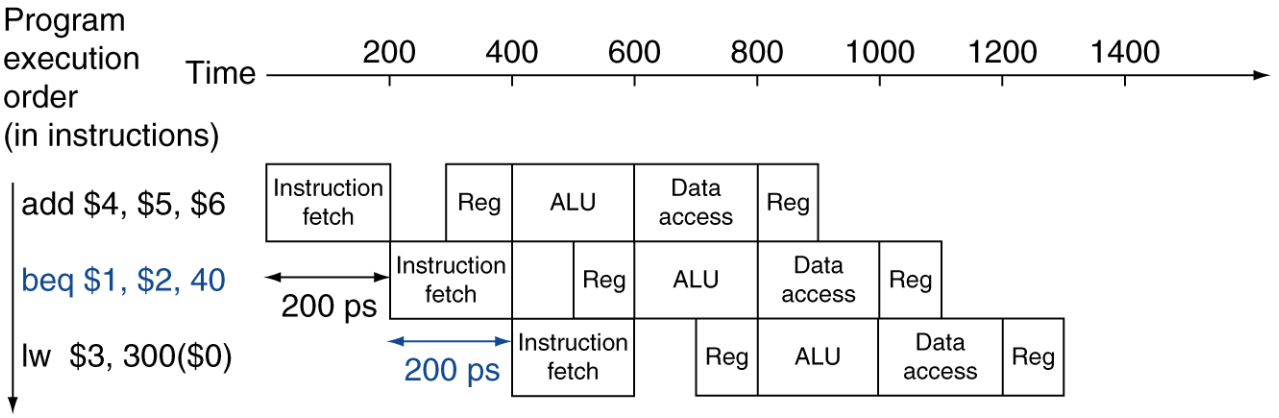


Branch Prediction

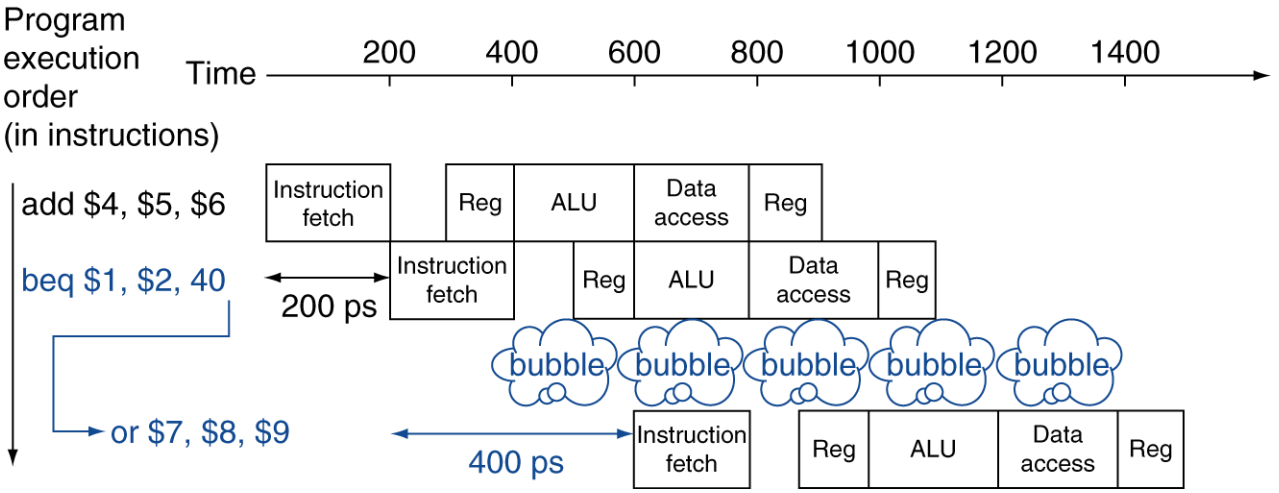
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



Prediction incorrect



More-Realistic Branch Prediction

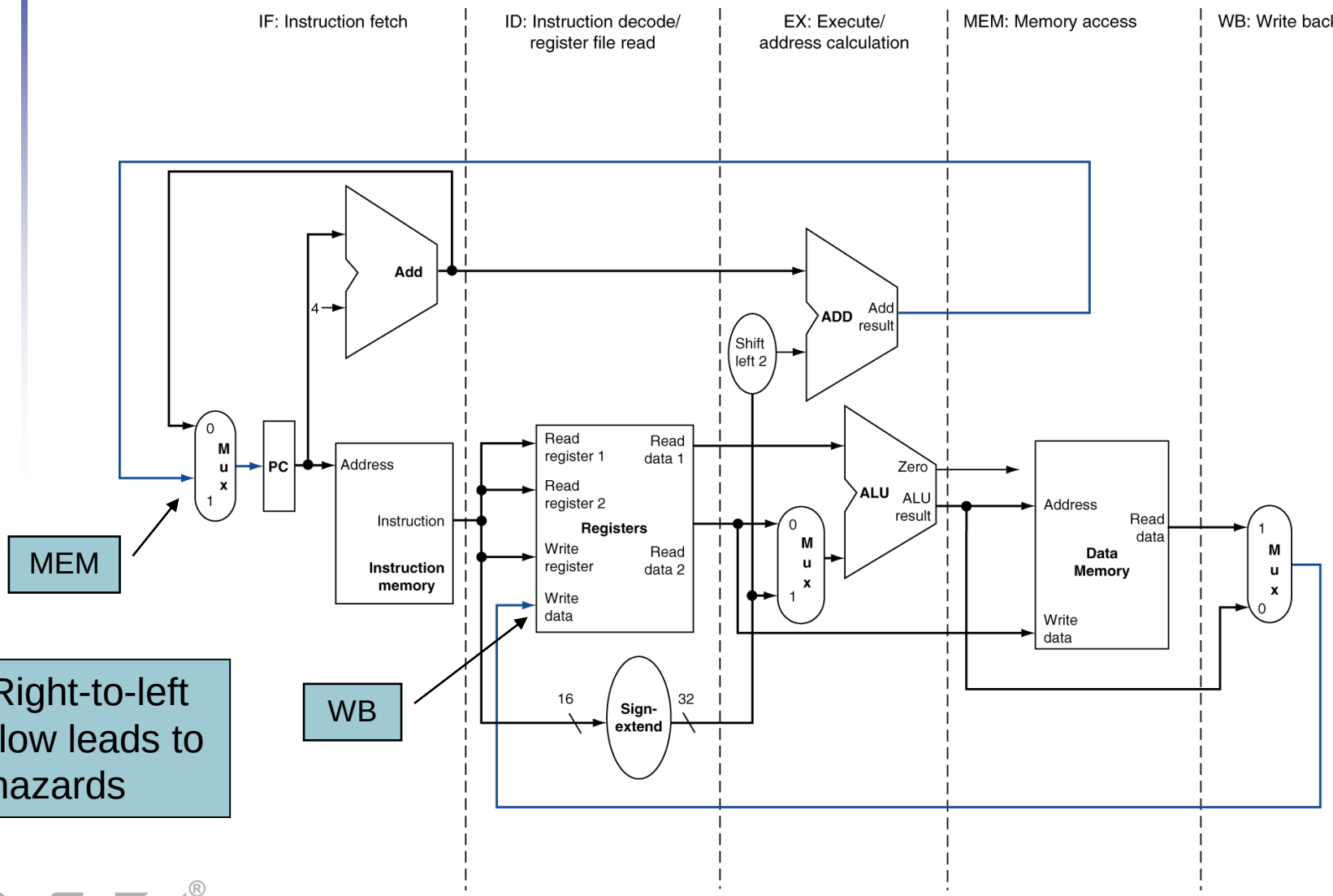
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each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

Pipeline Summary

The BIG Picture

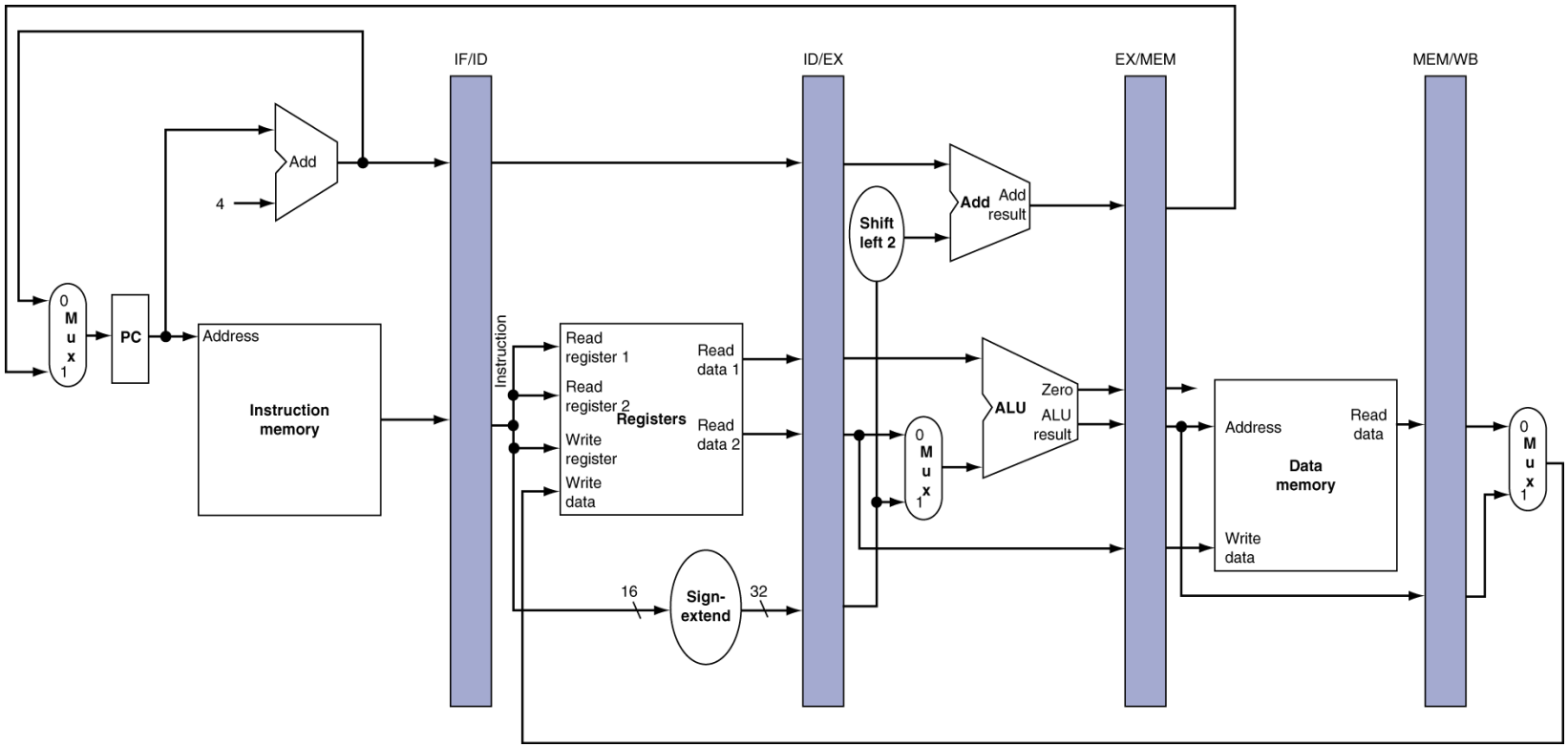
- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

MIPS Pipelined Datapath



Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle

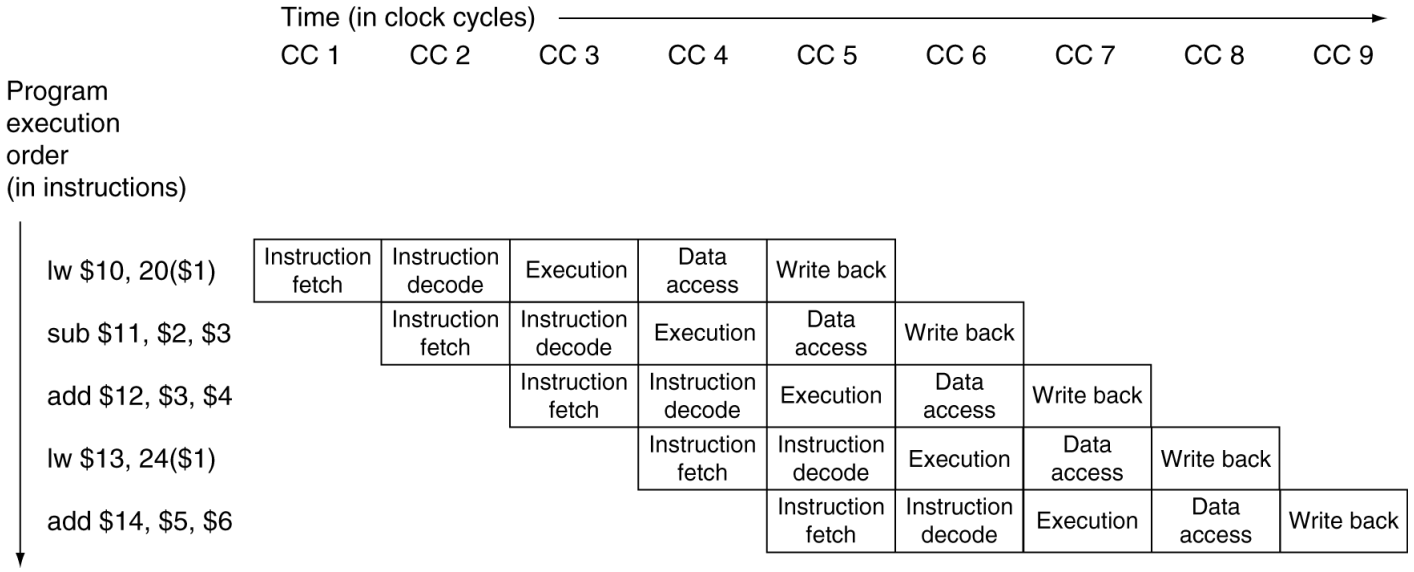


Pipeline Operation Diagram

- 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 illustrations.

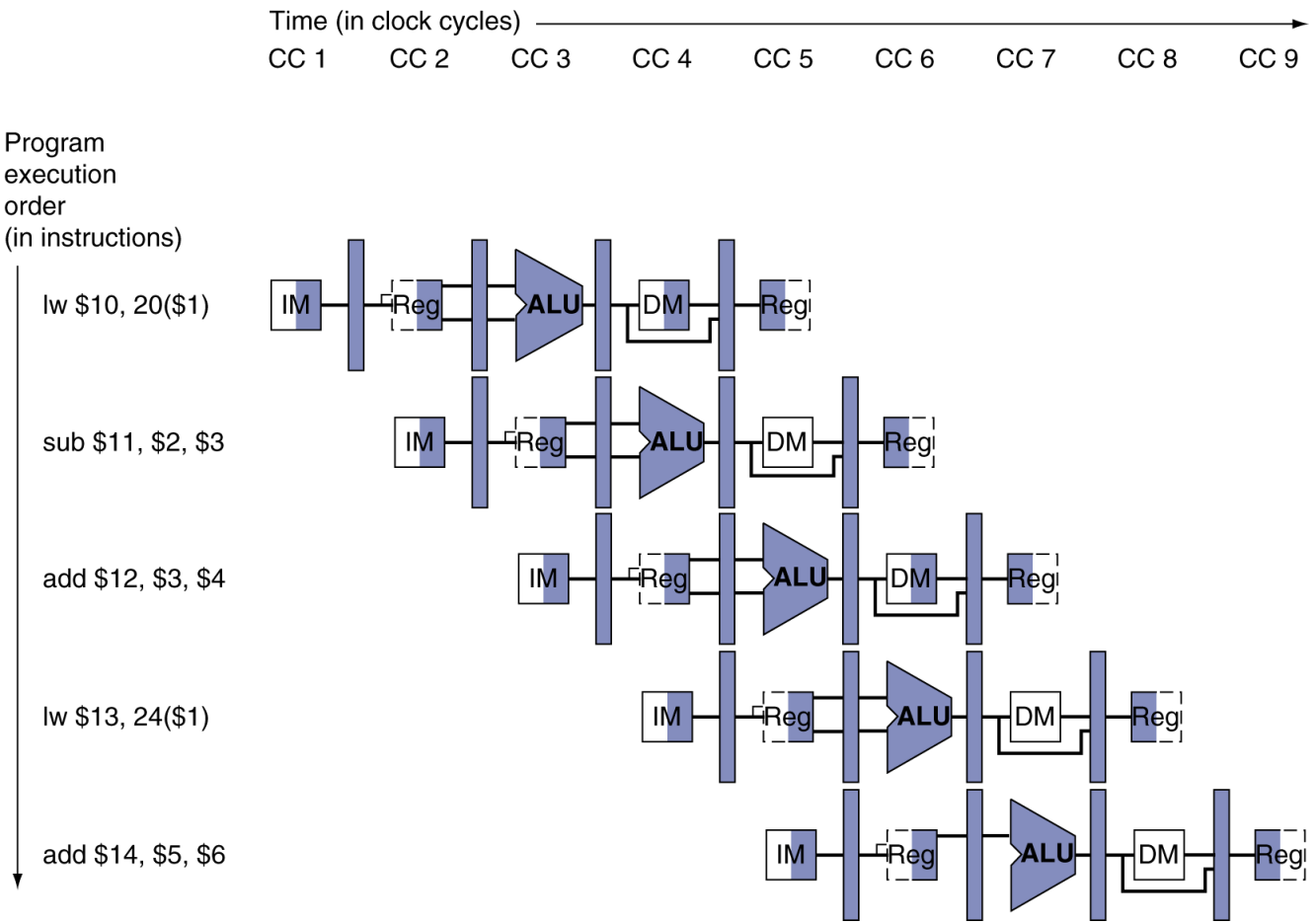
Multi-Cycle Pipeline Diagram

- Traditional form



Multi-Cycle Pipeline Diagram

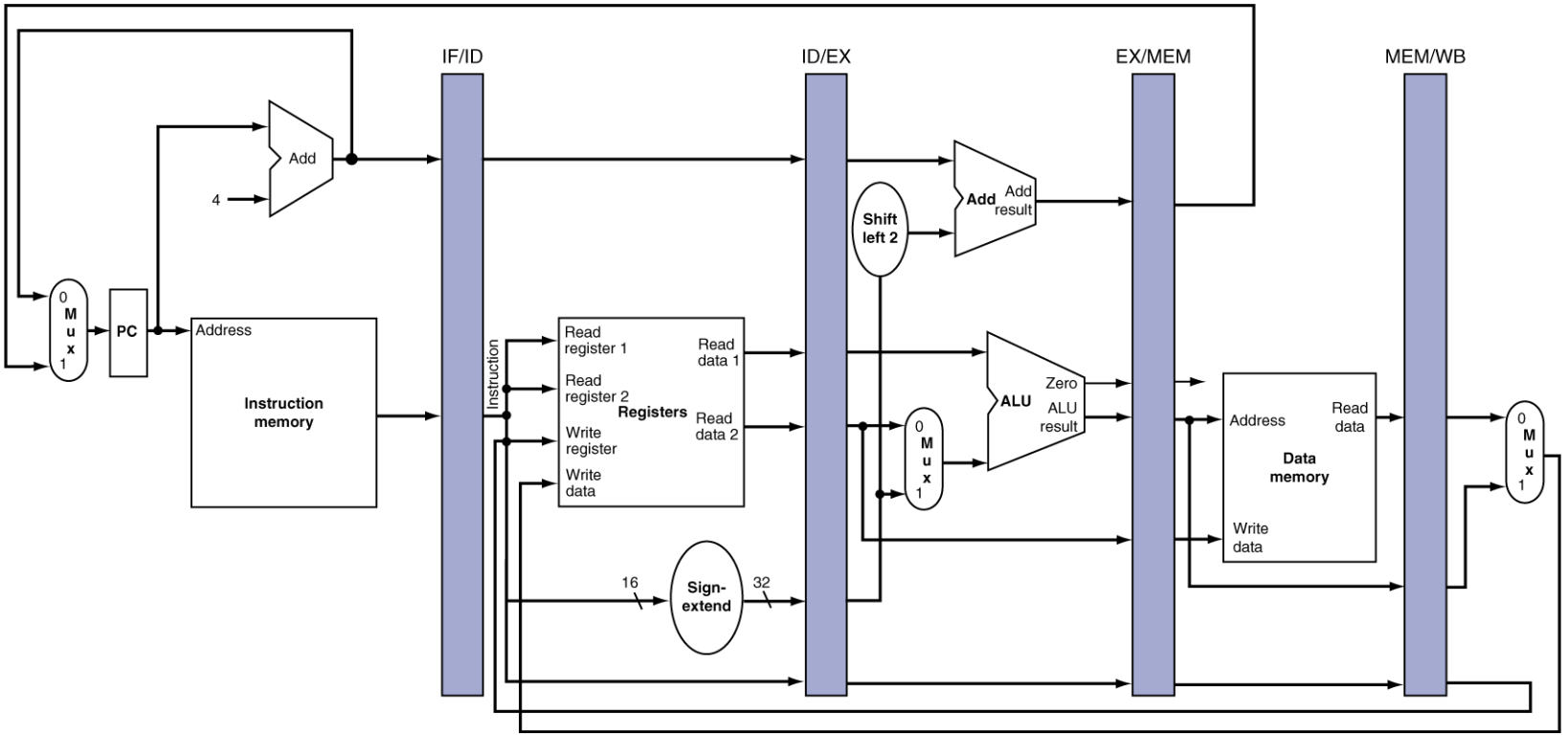
- Form showing resource usage



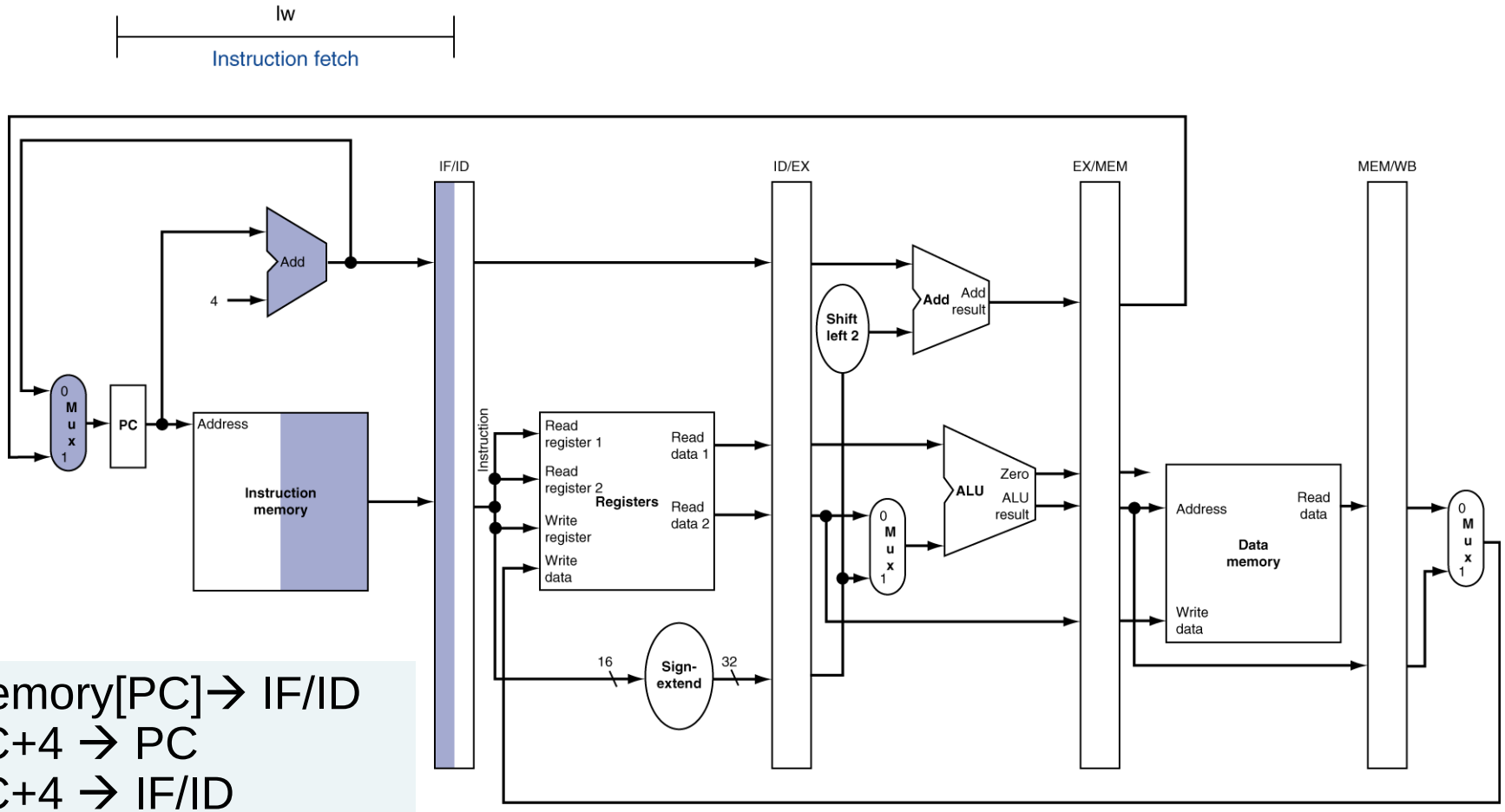
Single-Cycle Pipeline Diagram

- 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



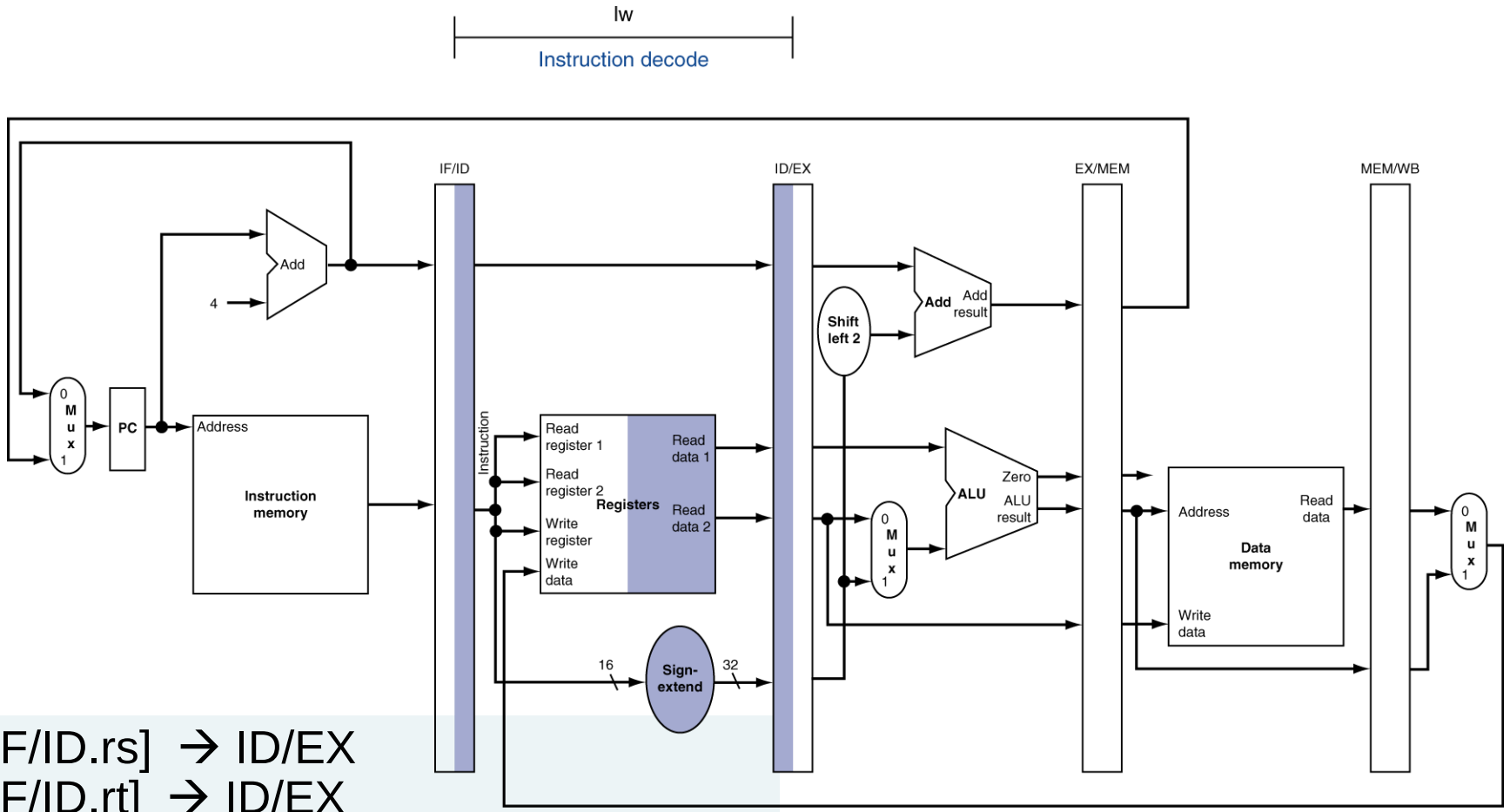
IF for Load, Store, …



Memory[PC]→ IF/ID
PC+4 → PC
PC+4 → IF/ID

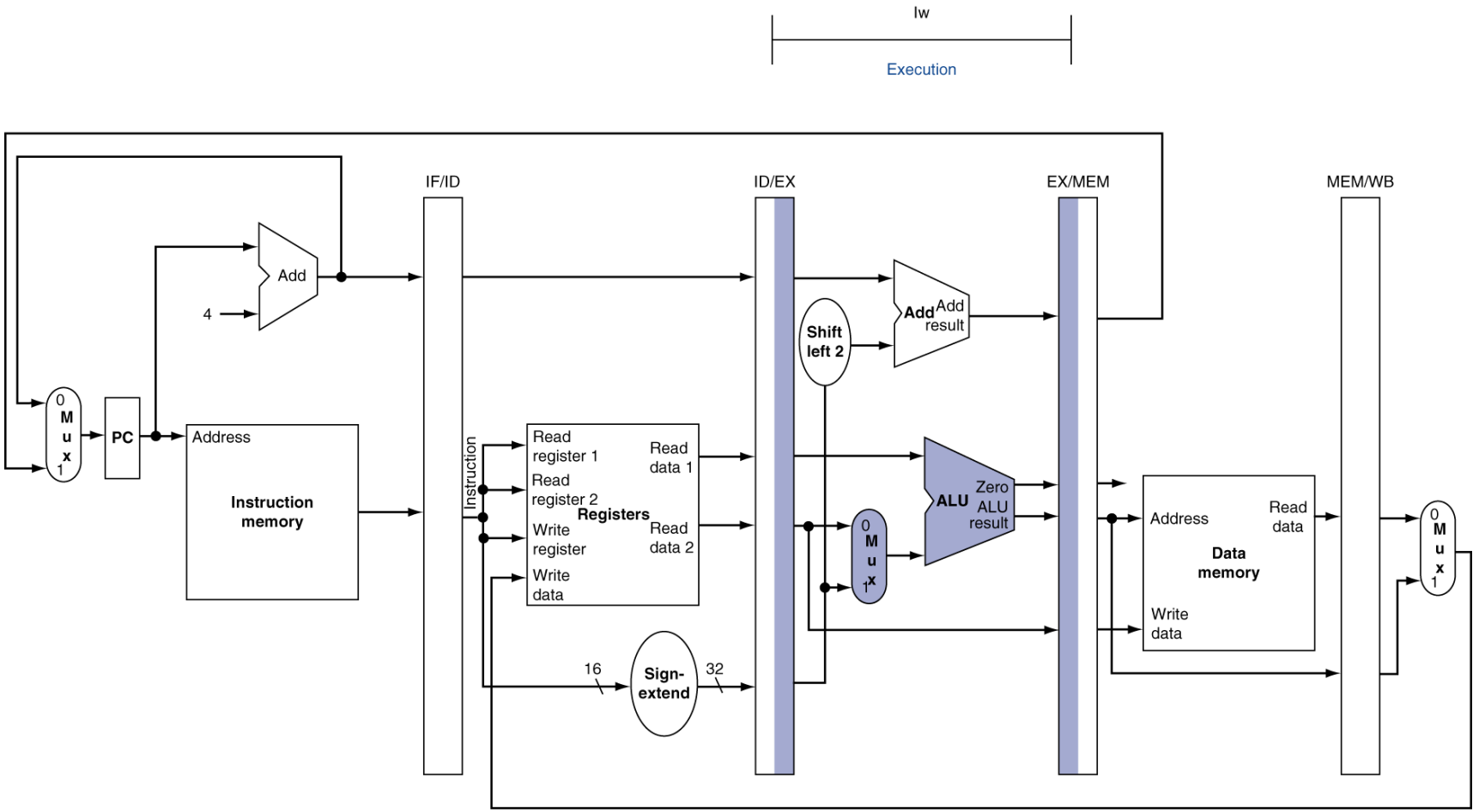
Note: Shaded left-half: Write Shaded right-half: Read

ID for Load, Store, …



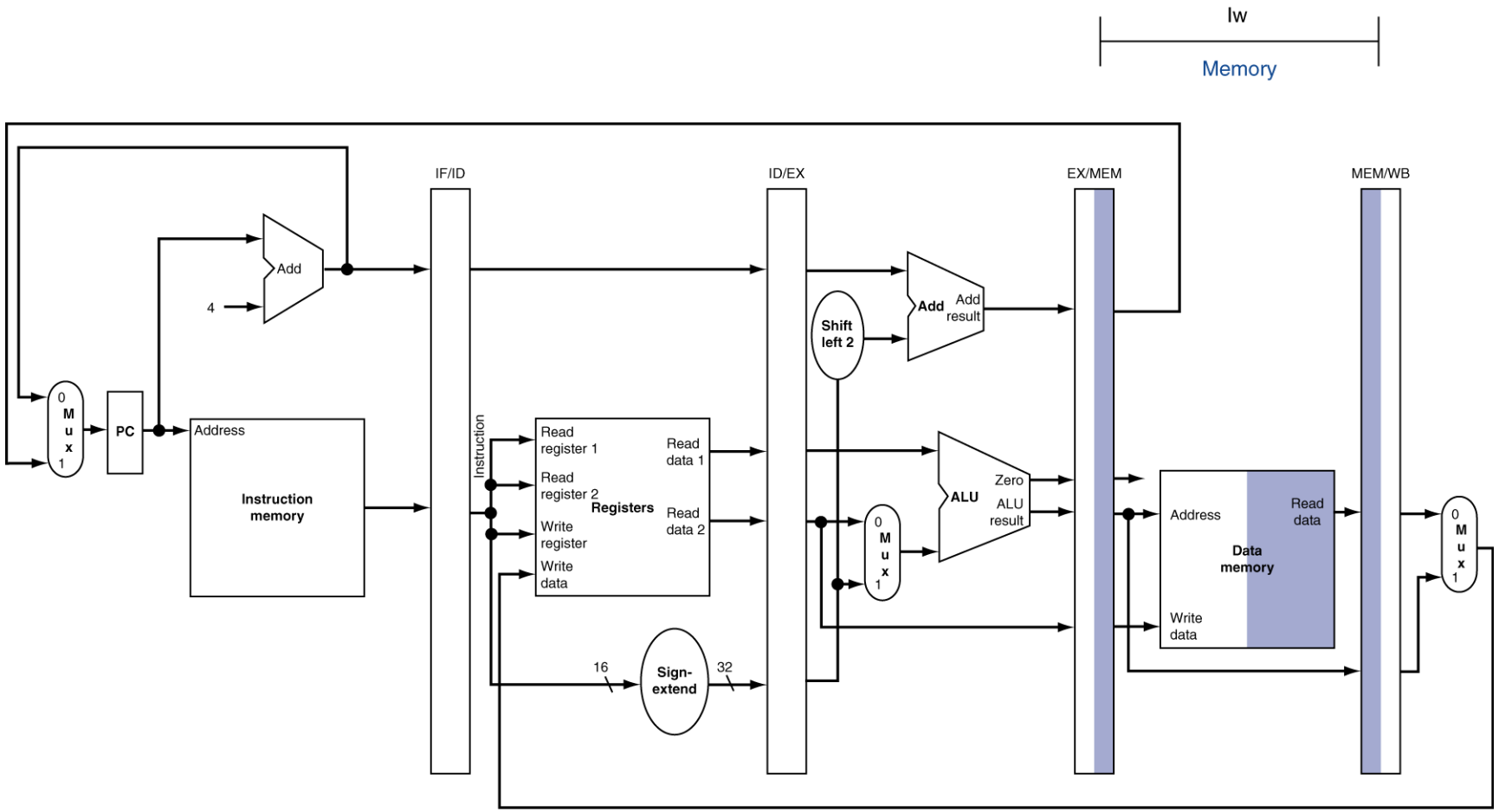
Reg[IF/ID.rs] → ID/EX
Reg[IF/ID.rt] → ID/EX
Sign-extend(IF/ID.Instr[15:0]) → ID/EX
IF/ID.pc+4 → ID/EX

EX for Load



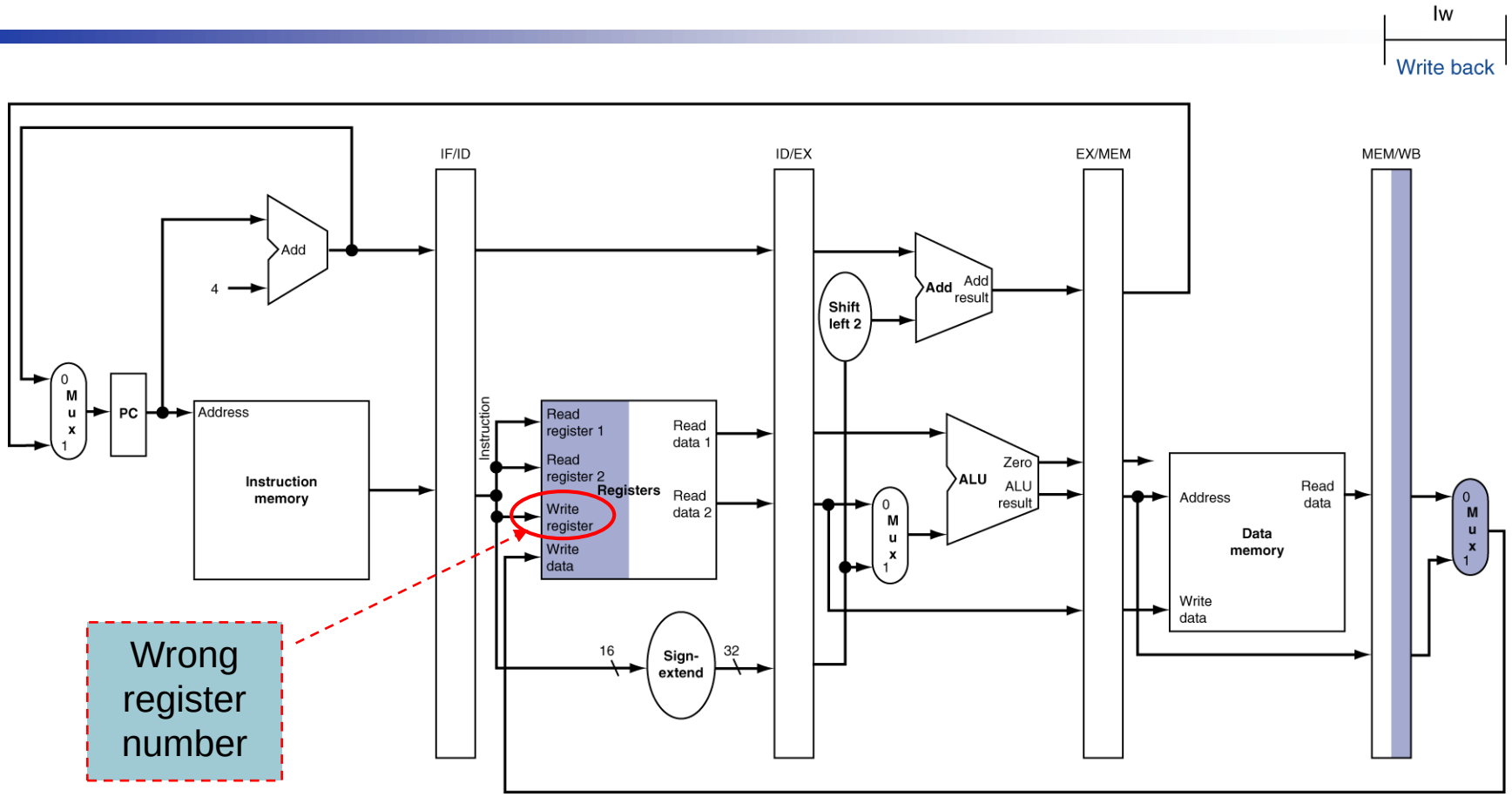
mem-addr = ID/EX.reg[rs]+ID/EX.sign-ext32 → EX/MEM

MEM for Load



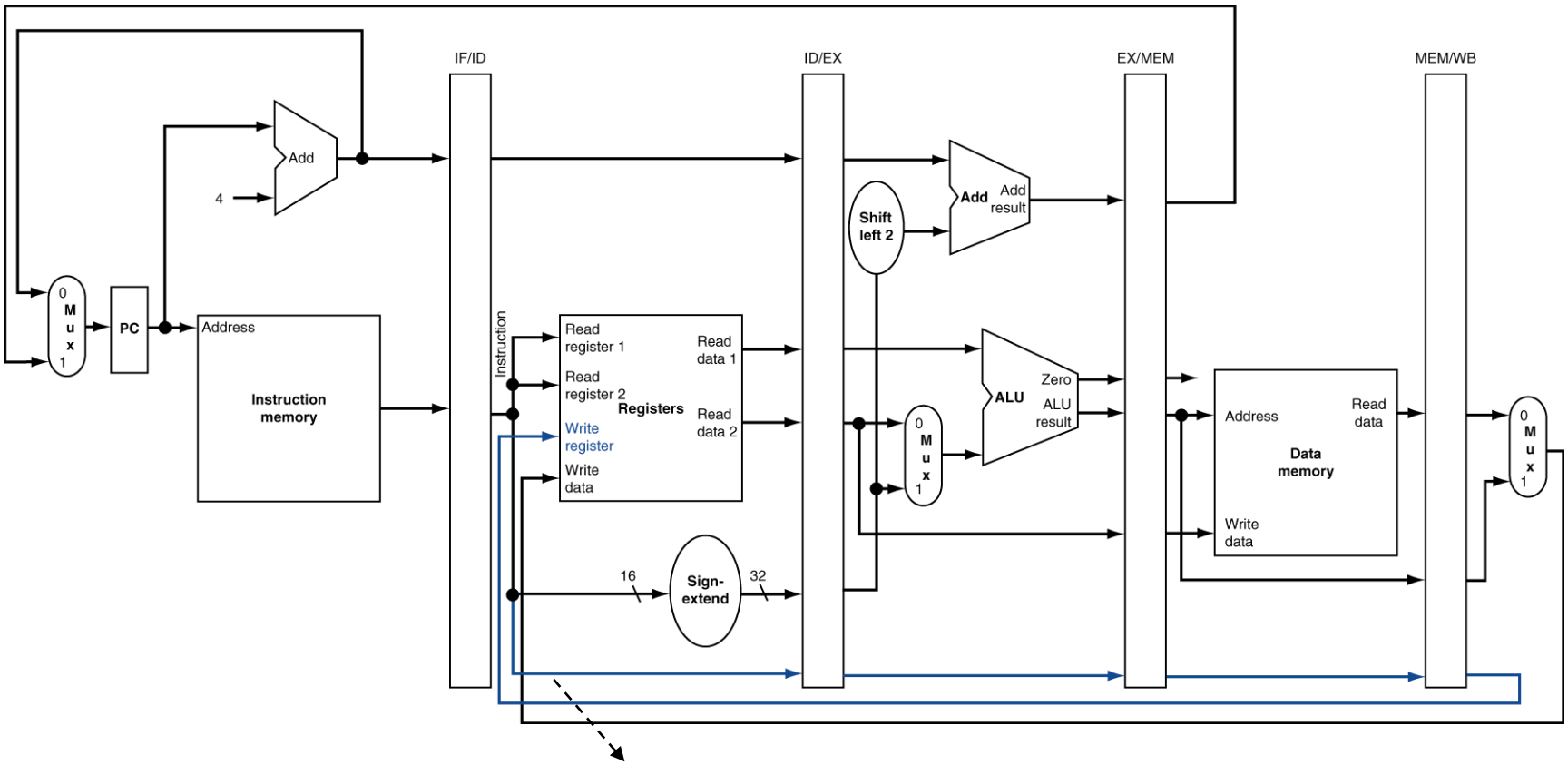
mem-data= Memory[EX/MEM. mem-addr] → MEM/WB

WB for Load



MEM/WB.mem-data → Reg[rt]

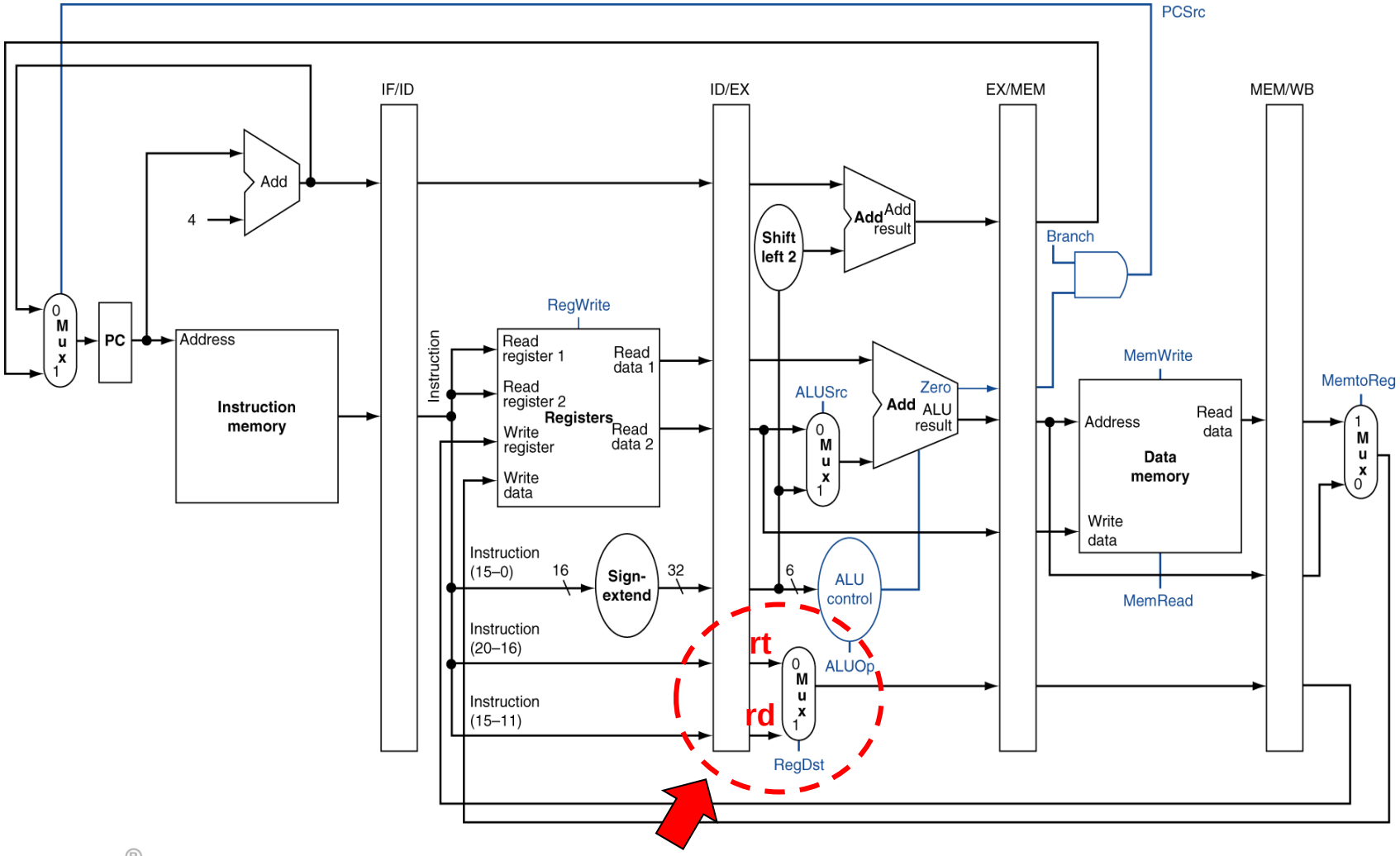
Corrected Datapath for Load



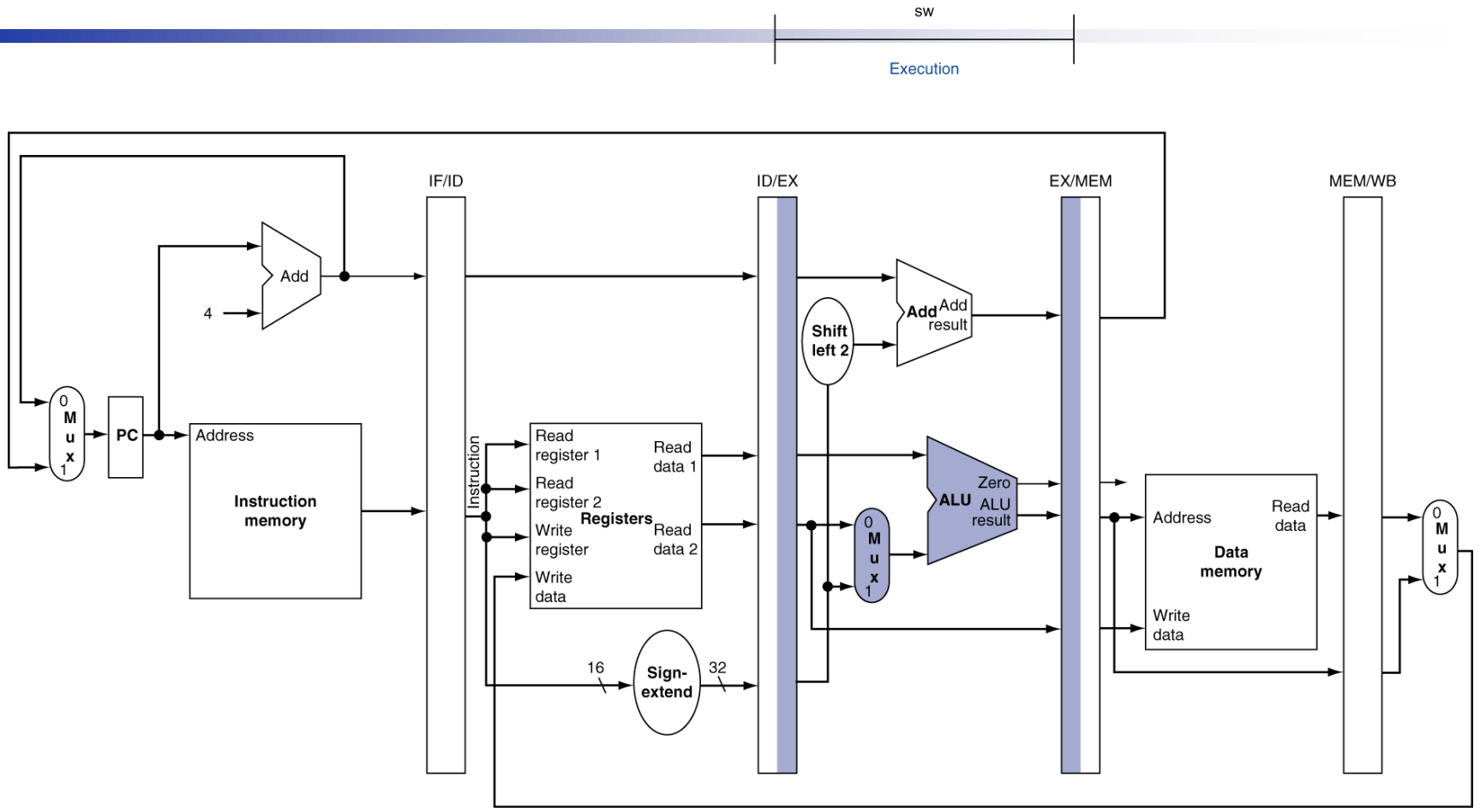
IF/ID.rt -> ID/EX -> EX/MEM -> MEM/WB for lw

How about R-type instruction ?

Corrected Datapath for Load and R-type



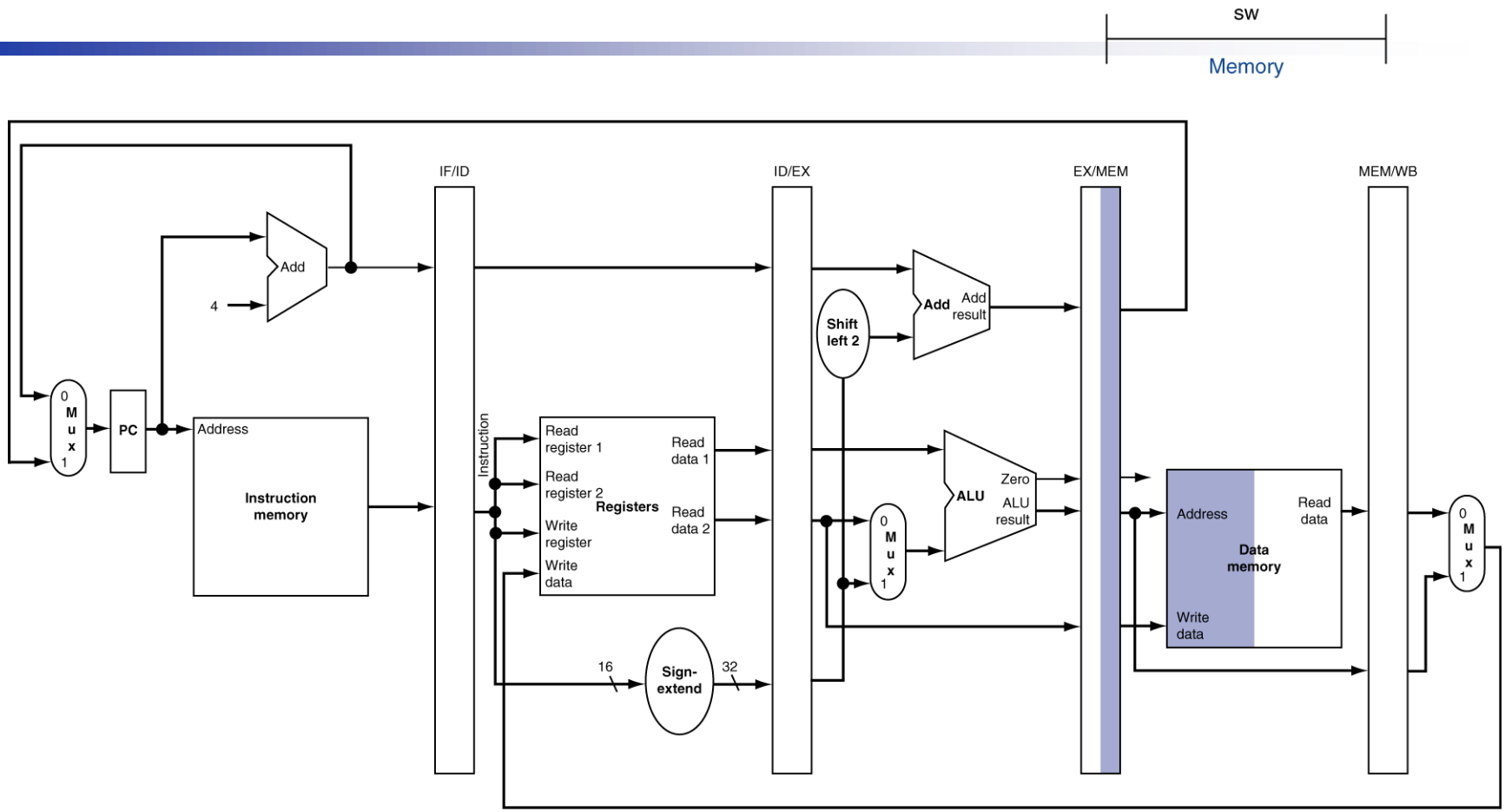
EX for Store



mem-addr = ID/EX.reg[rs]+ID/EX.sign-ext32 → EX/MEM

Anything else ? See next page

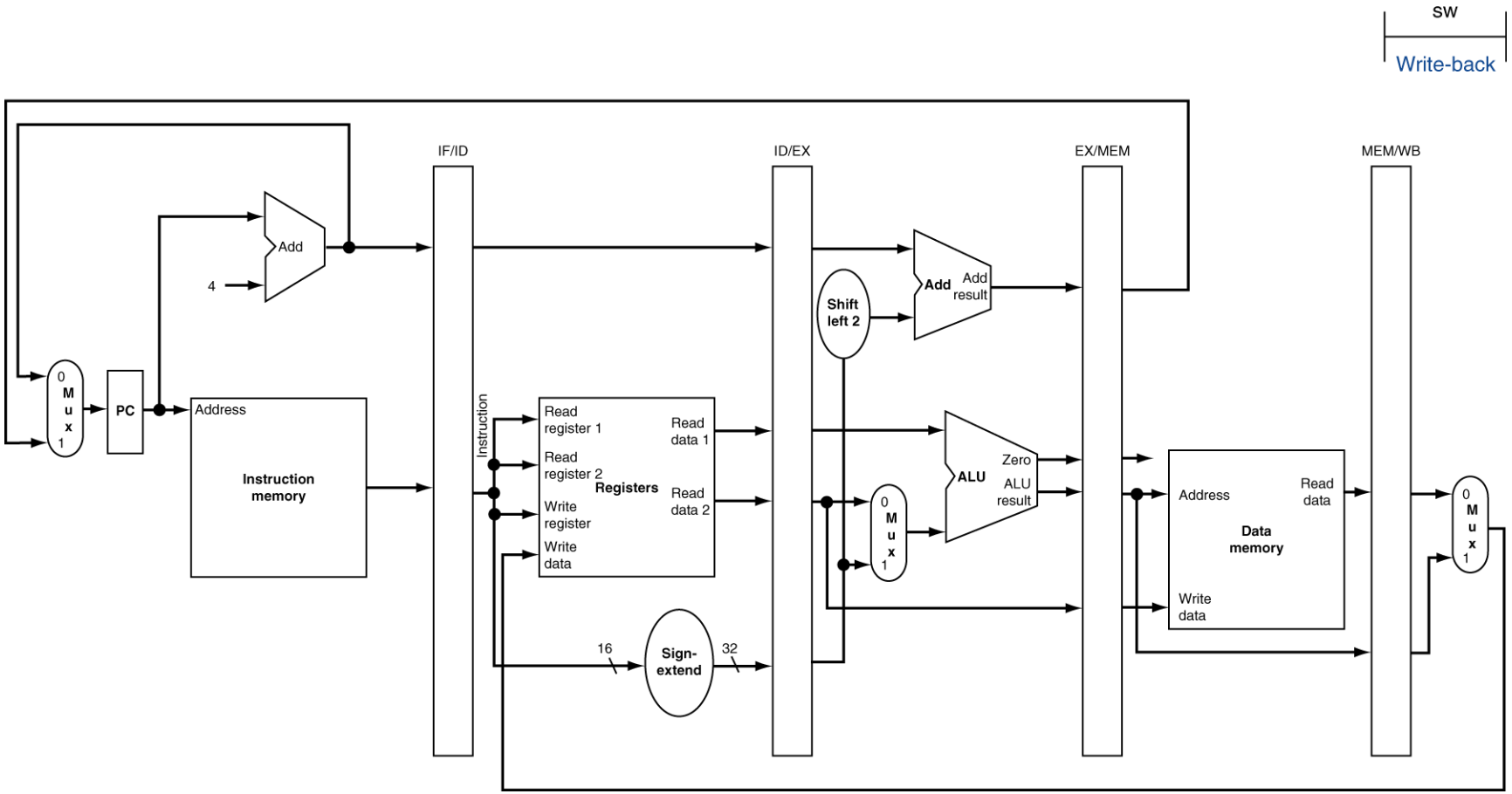
MEM for Store



Reg[rt] → MEM[EX/MEM. mem-addr]

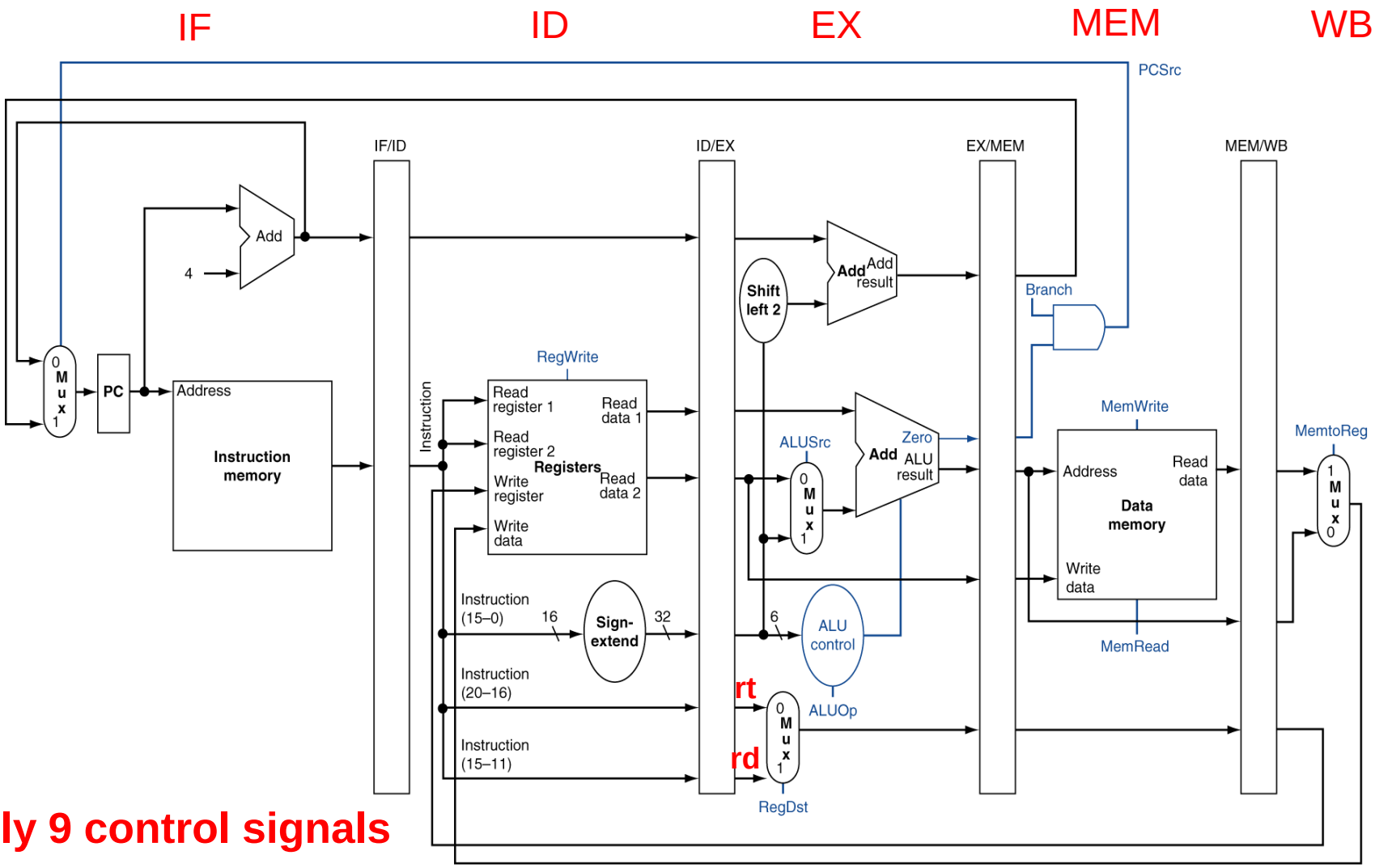
How to get **Reg[rt]**? add "ID/EX.Reg[rt] → EX/MEM" to EX stage

WB for Store



Do nothing at this stage.

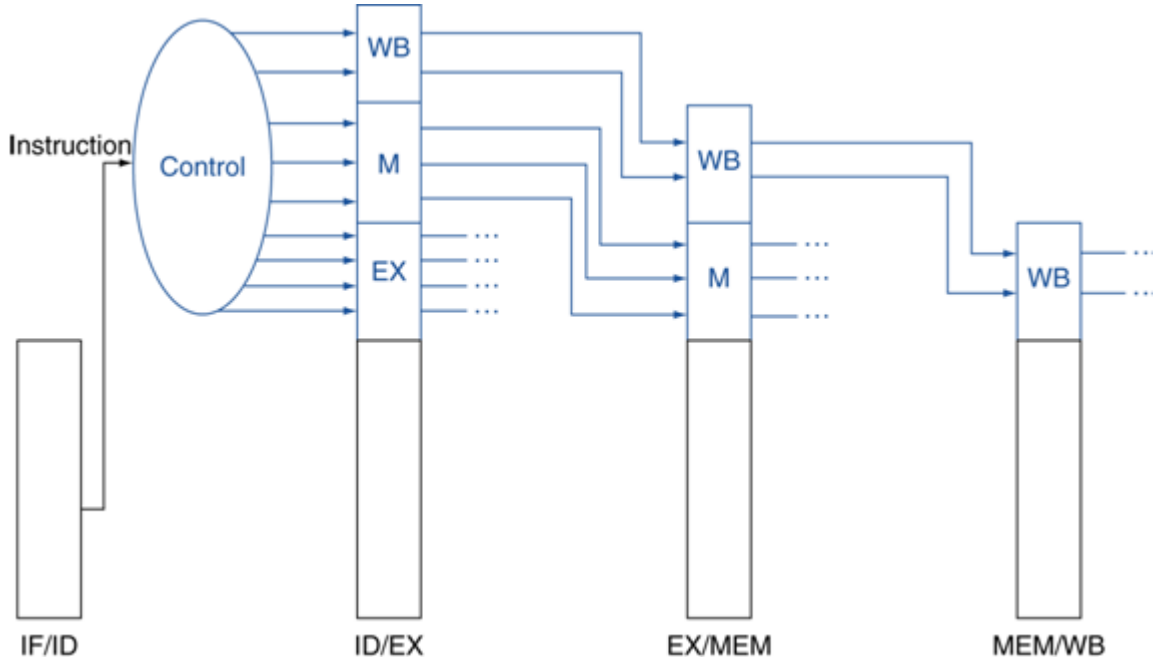
Pipelined Control (Simplified)



Totally 9 control signals

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
- Pass control signals along just like the data
  - Create control info. at ID stage and then used in appropriate stage as pipeline move down

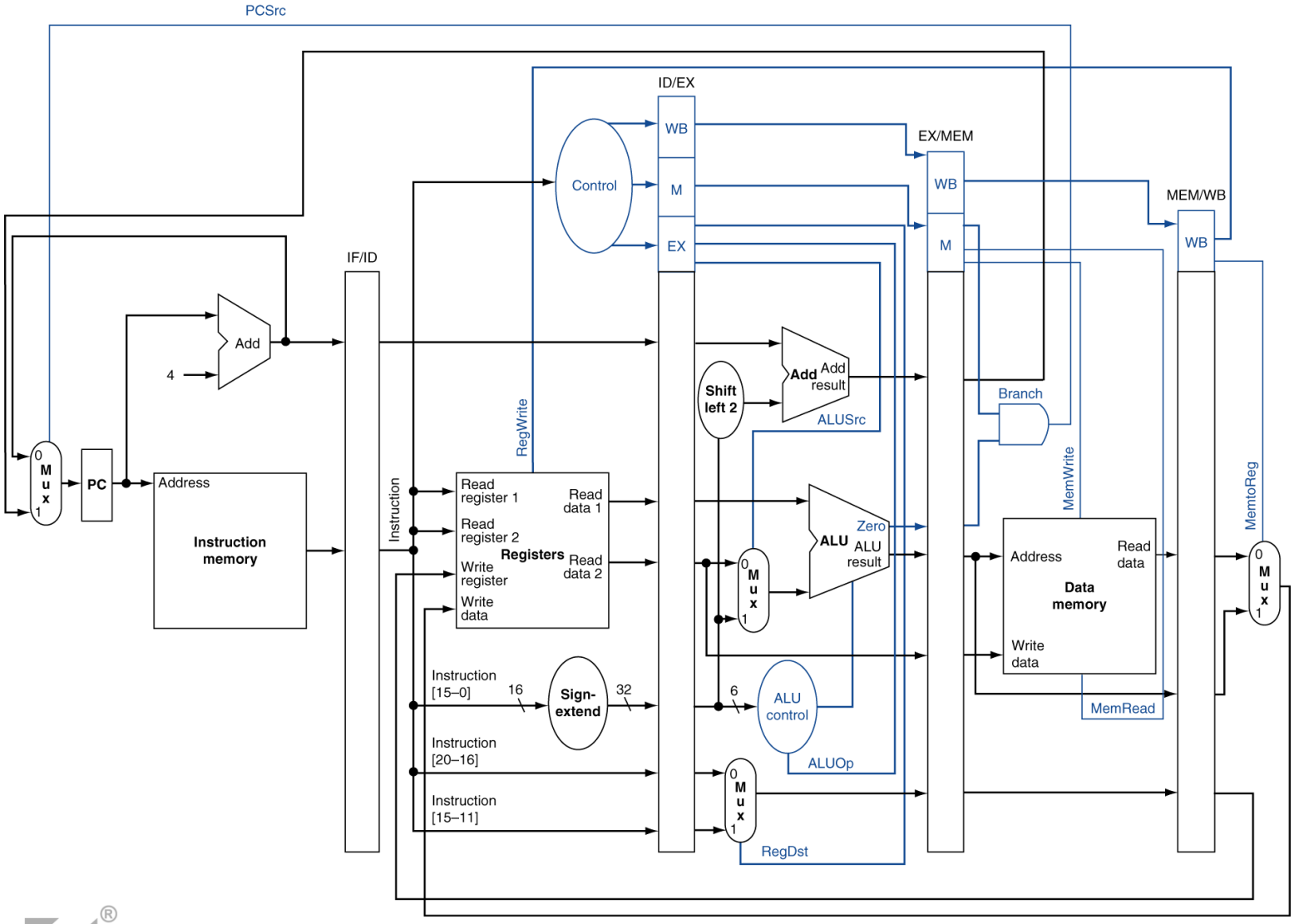


Pipelined Control

Instr.	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	x

- **IF**: the control signals to read instruction memory and to write PC are always asserted for at each cycle, so there is nothing special to control at this stage.
- **ID**: As in IF, the same thing happens at every cycle, so no optional control signals to set.

Pipelined Control



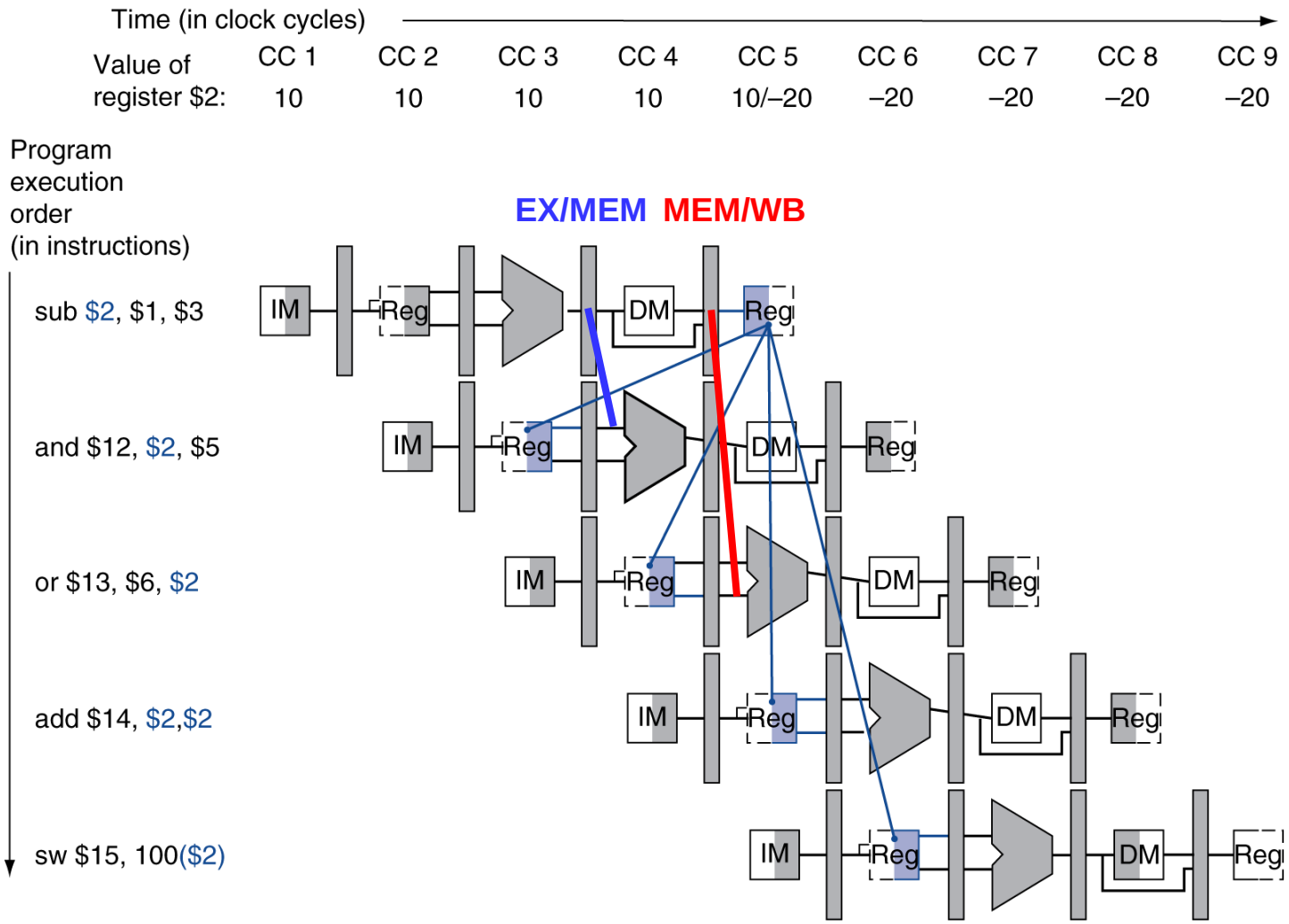
Data Hazards in ALU Instructions

- 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



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

Detecting the Need to Forward

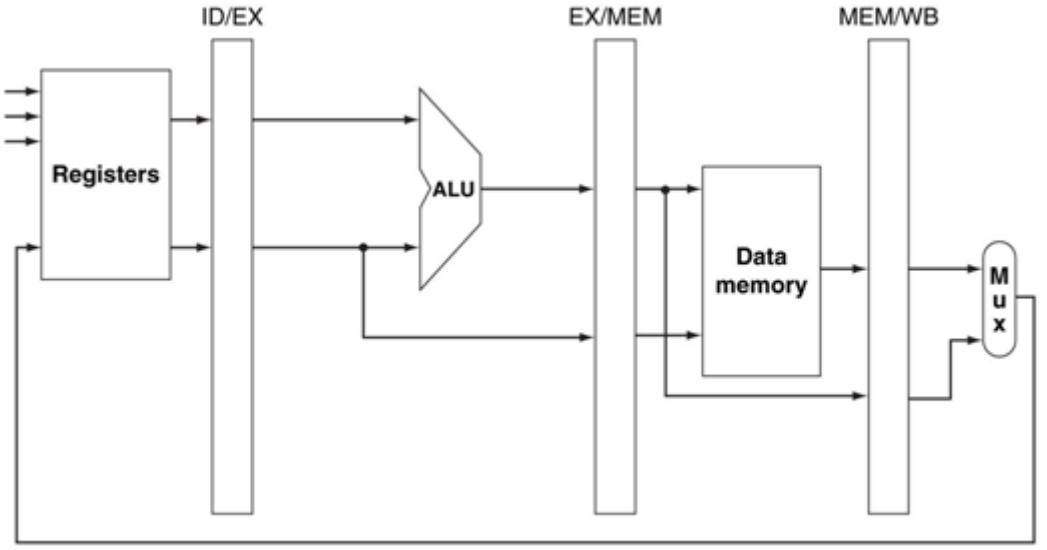
- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Conditions

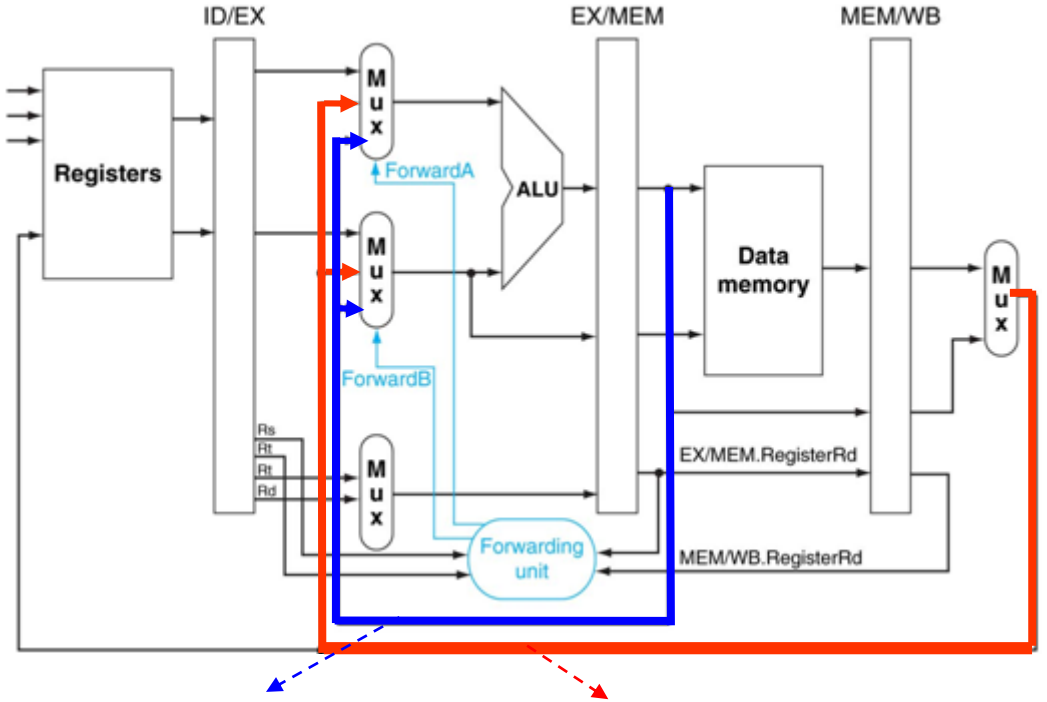
- EX hazard (the *sub-and* in p.83)
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = **10**
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = **10**
- MEM hazard (the *sub-or* in p.83)
  - 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**

Forwarding

Without forwarding paths



With forwarding paths



1a.EX/MEM.RegRd = ID/EX.RegRs
1b.EX/MEM.RegRd = ID/EX.RegRt

2a.MEM/WB.RegRd = ID/EX.RegRs
2b.MEM/WB.RegRd = ID/EX.RegRt

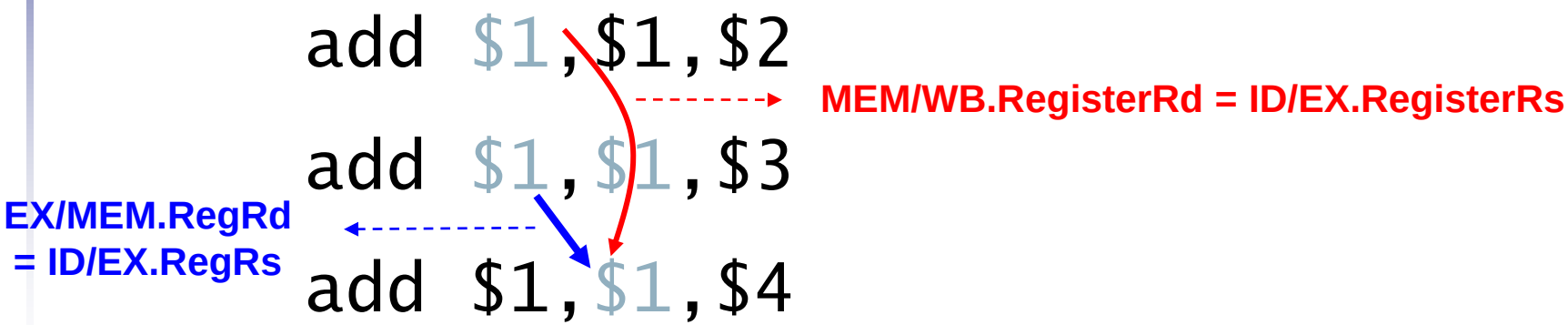
Forwarding Control

- The control values for the forwarding multiplexors

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

Double Data Hazard

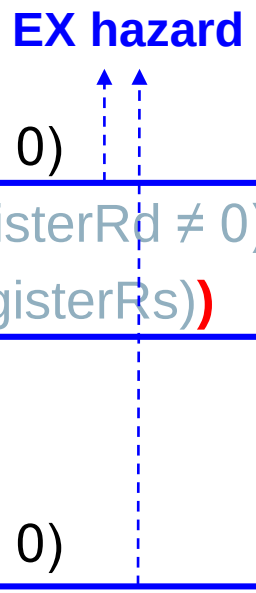
- Consider the sequence:



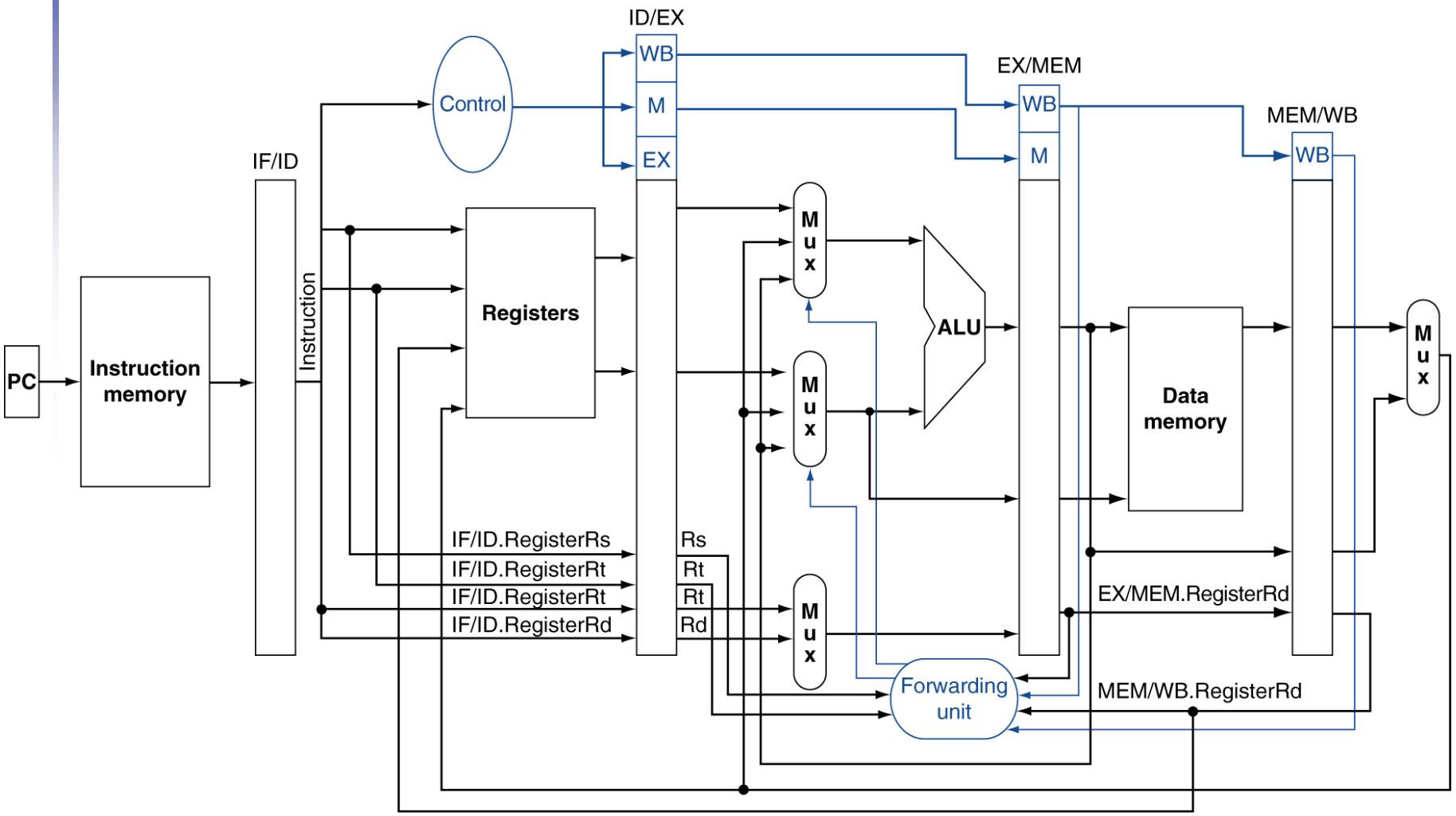
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

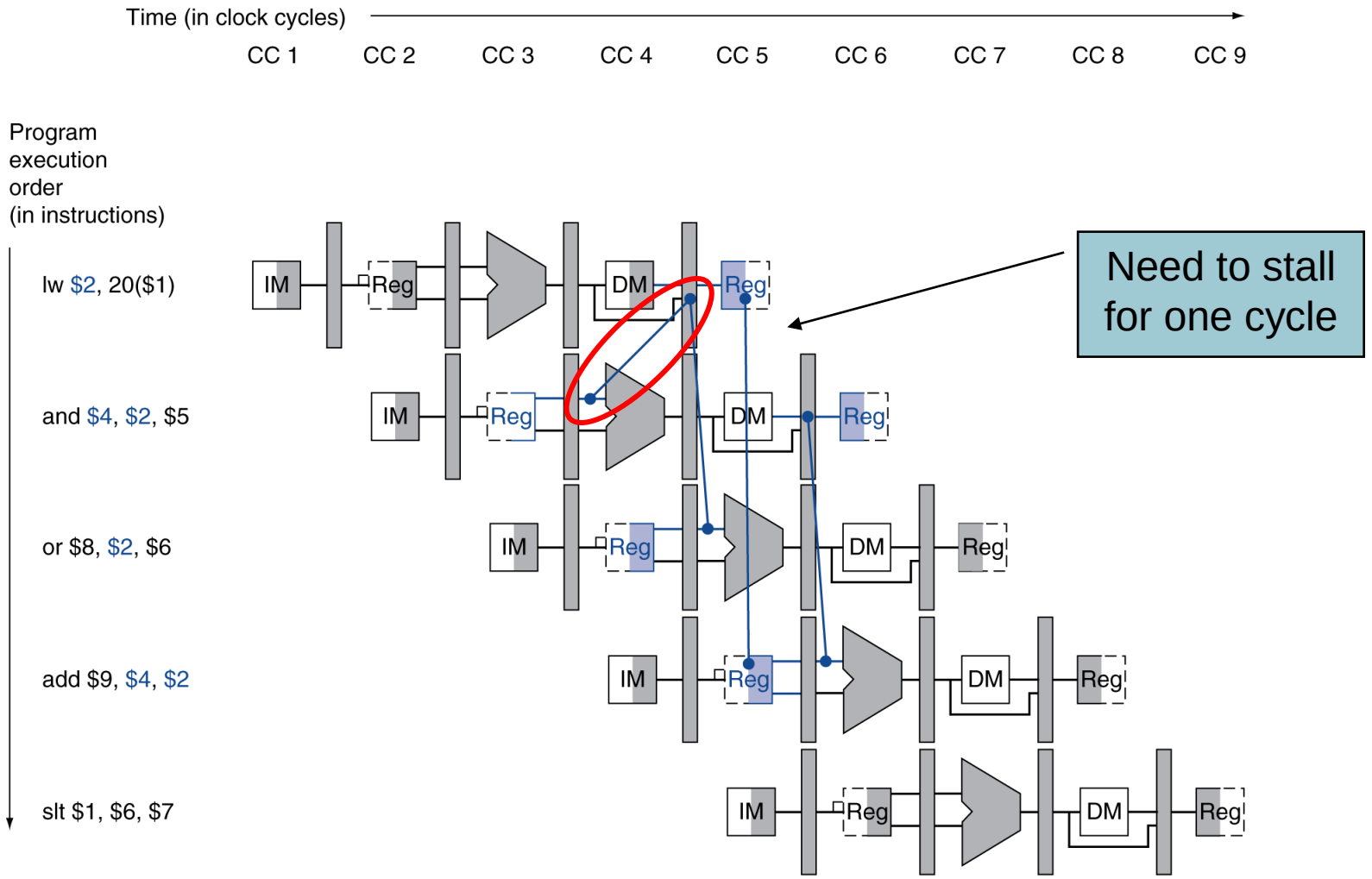
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and **not** **(**(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs)**)**
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    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))

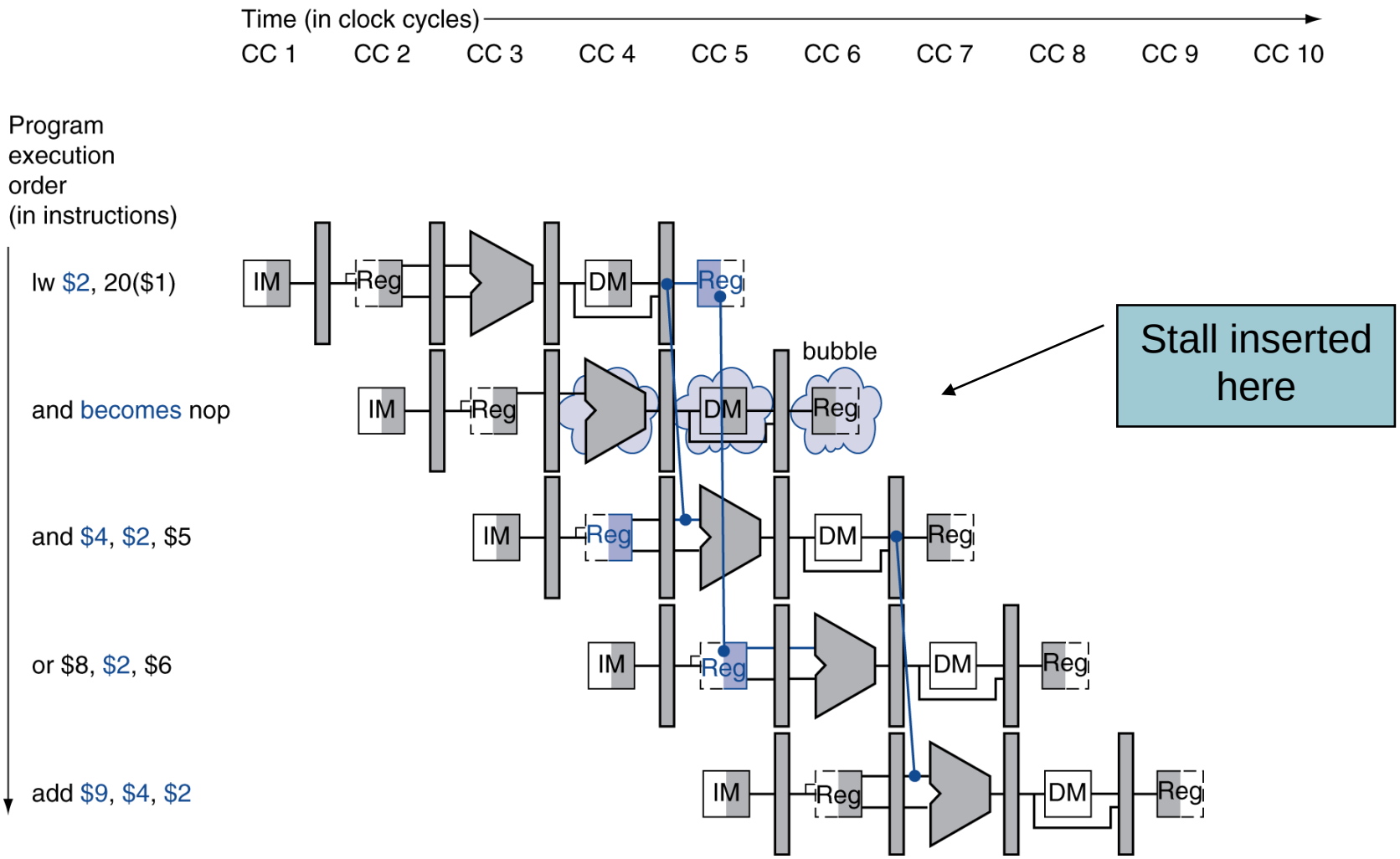
A load instr.

- If detected, stall and insert bubble

How to Stall the Pipeline

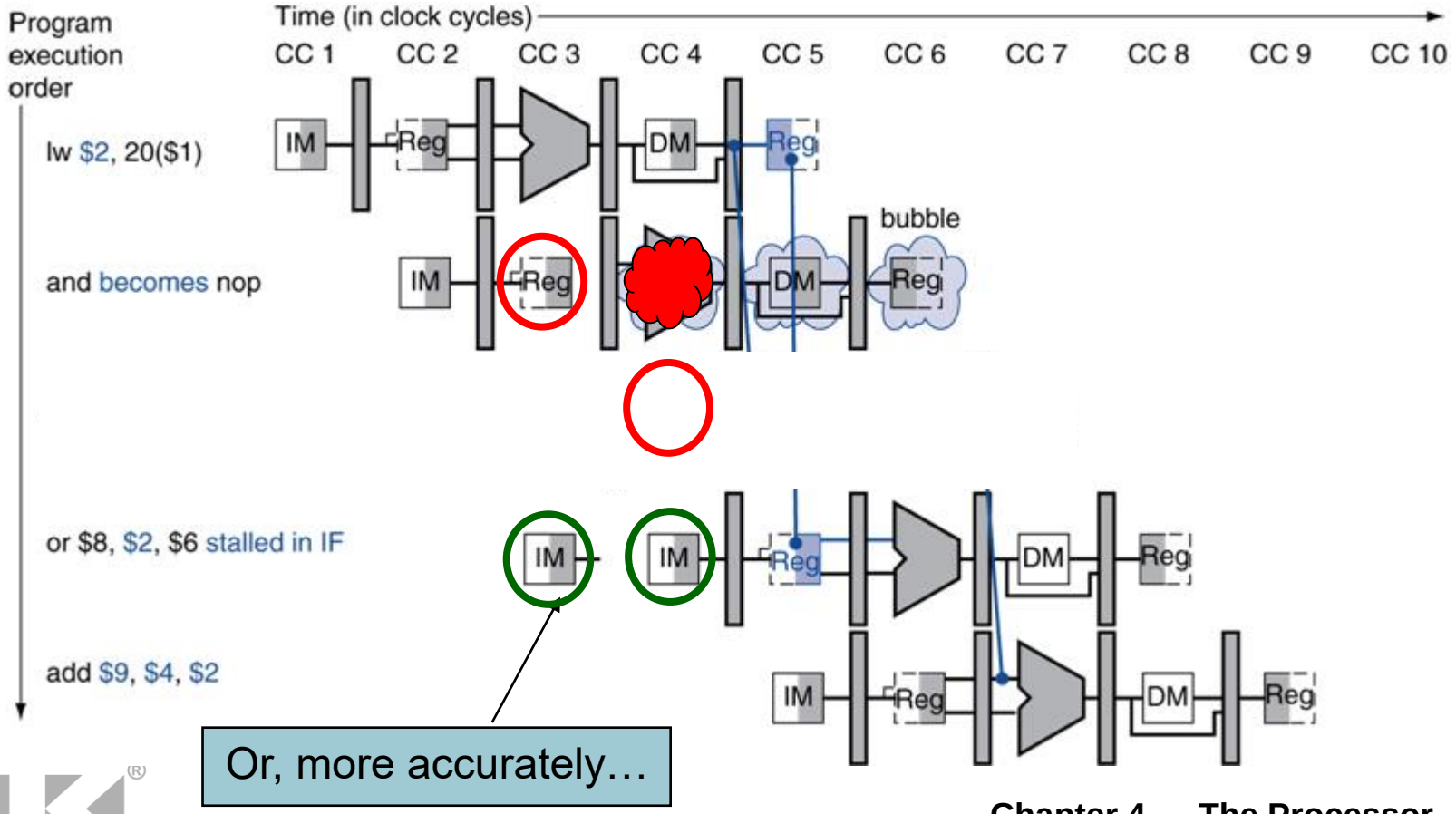
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline



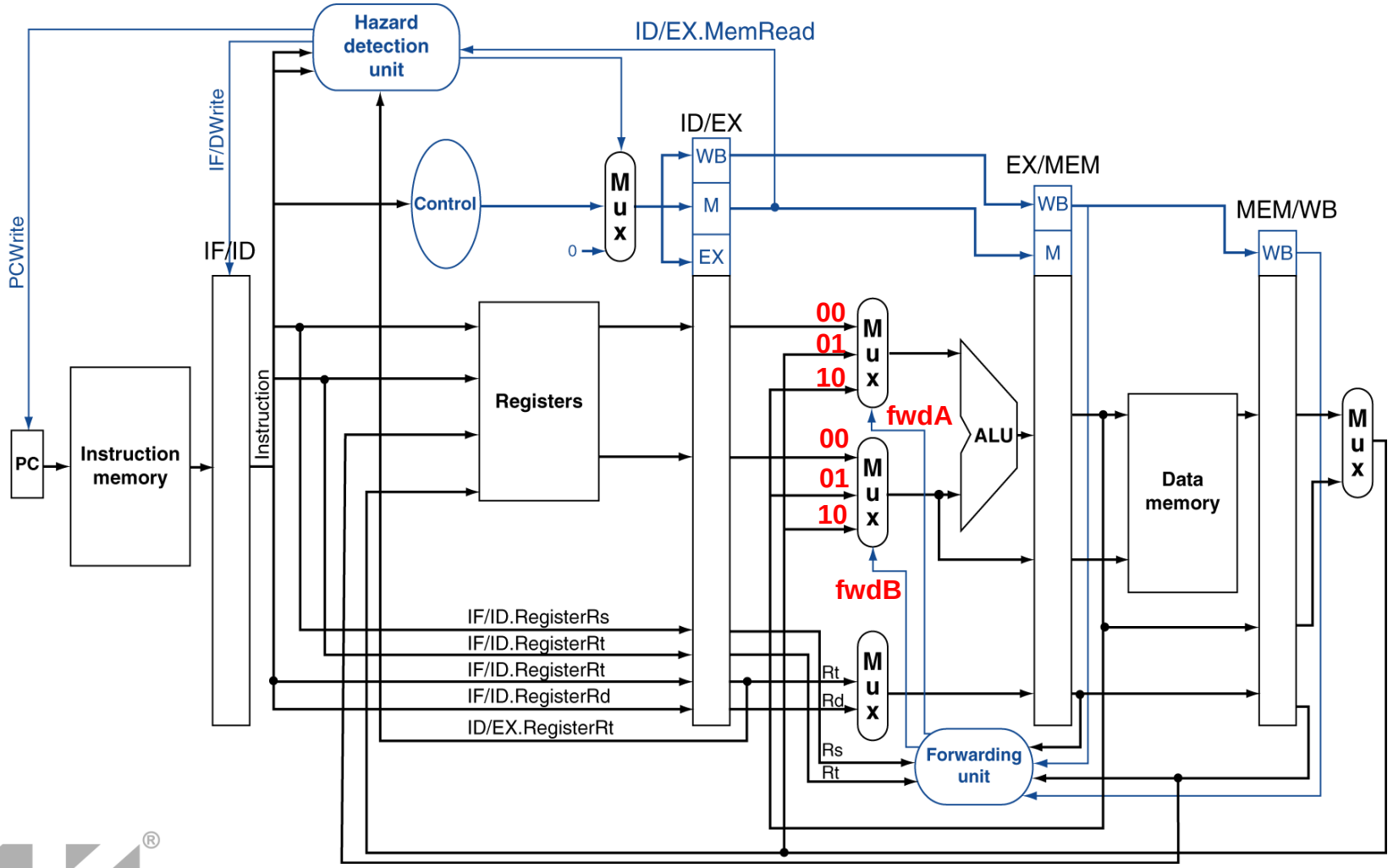
Stall/Bubble in the Pipeline

- stall the pipeline: keep an instruction in the same stage
  - bubble – change the EX, MEM, WB controls fields of the ID/EX to 0
  - Keep the instr. in IF and ID for one more cycle (*and* and *or* below)



Datapath with Hazard Detection

Hazard detected: 0 as mux input (*and* → bubble), PCWrite = 0, IF/Dwrite = 0



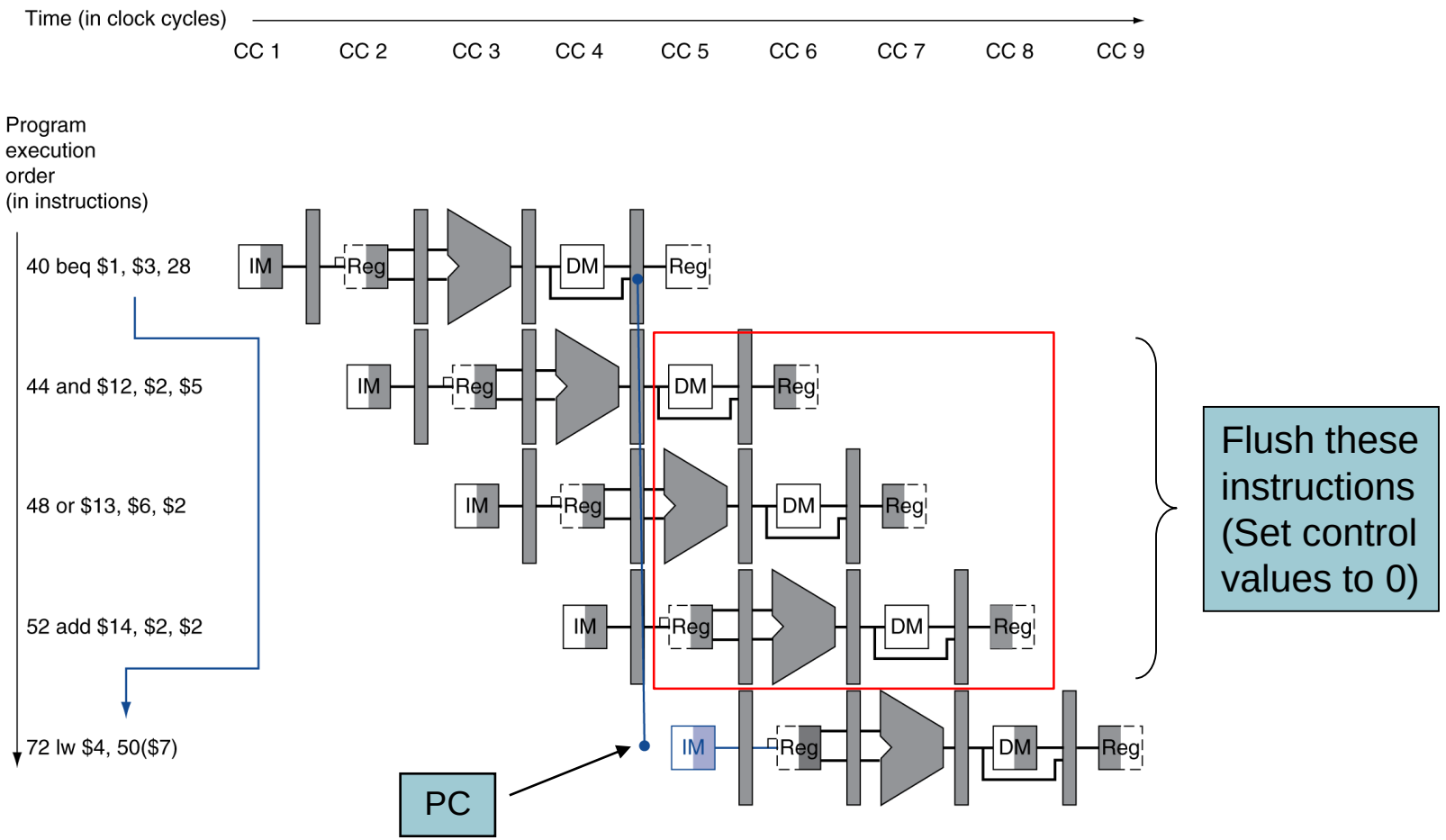
Stalls and Performance

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



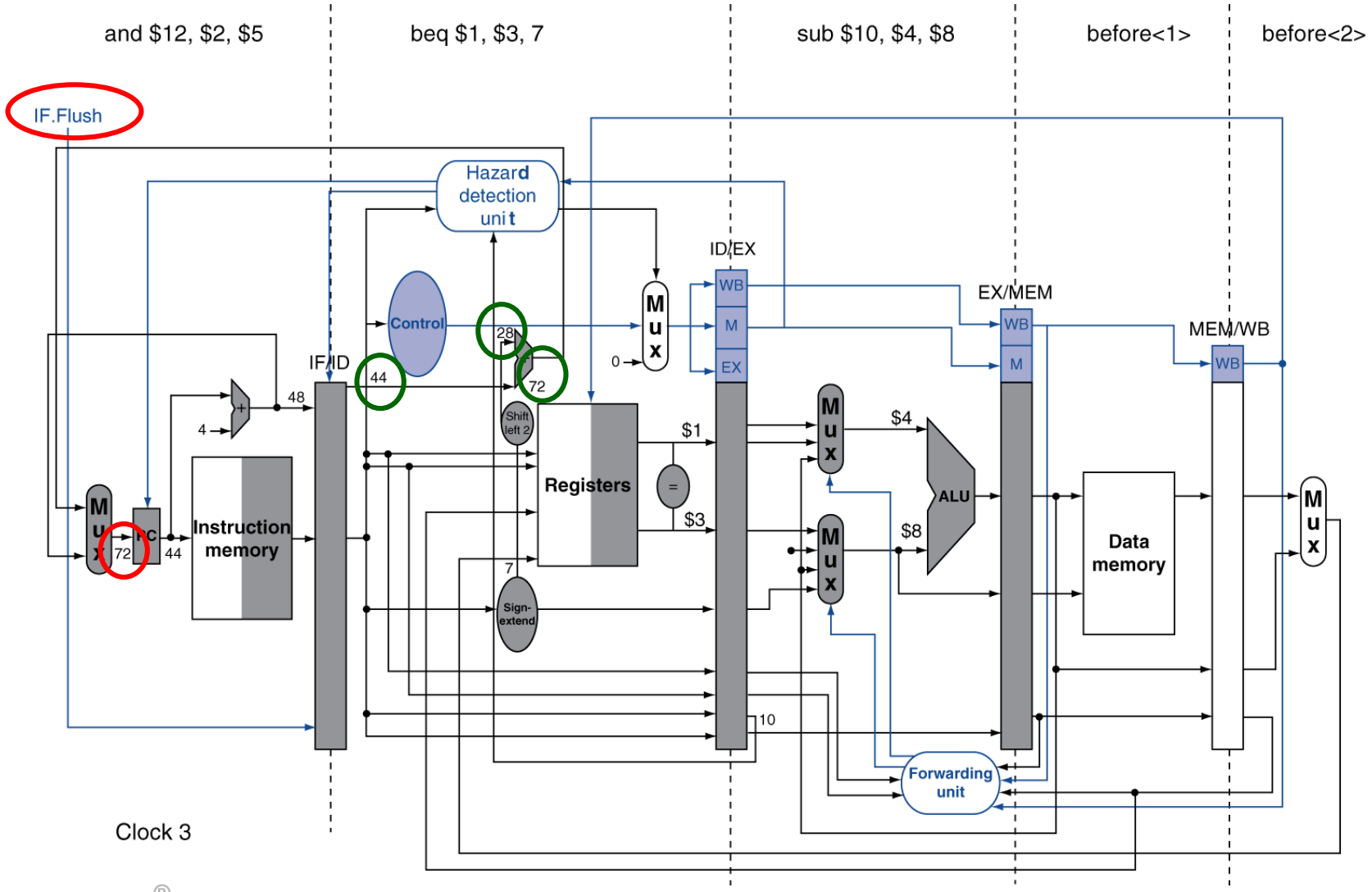
Reducing Branch Delay

- Add hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator (XOR two registers and OR the result)
- Example: branch taken

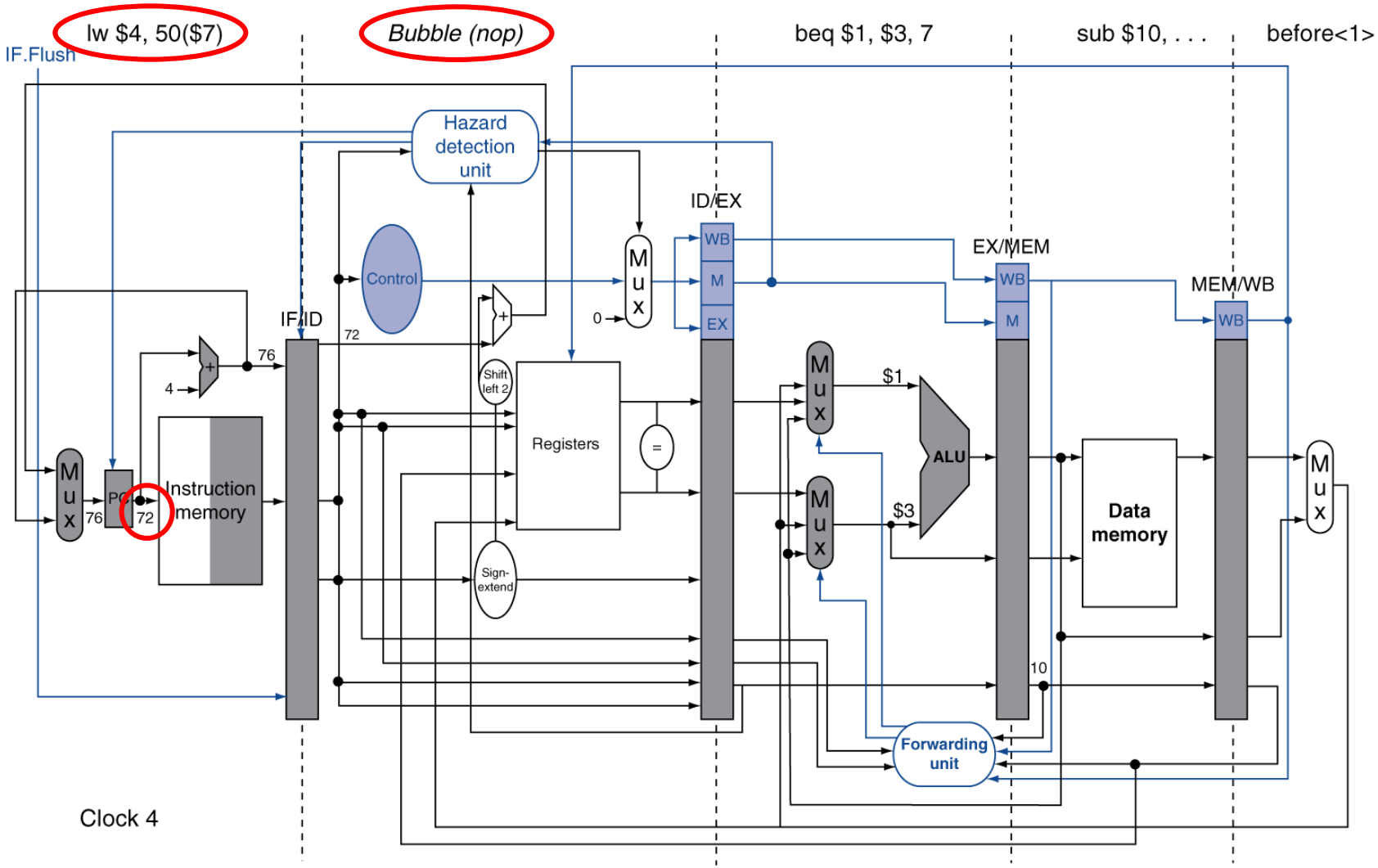
```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

For wrong branch decision:
flush the instr. at IF stage
(use IF.Flush to clear IF/ID.instr.)

Example: Branch Taken

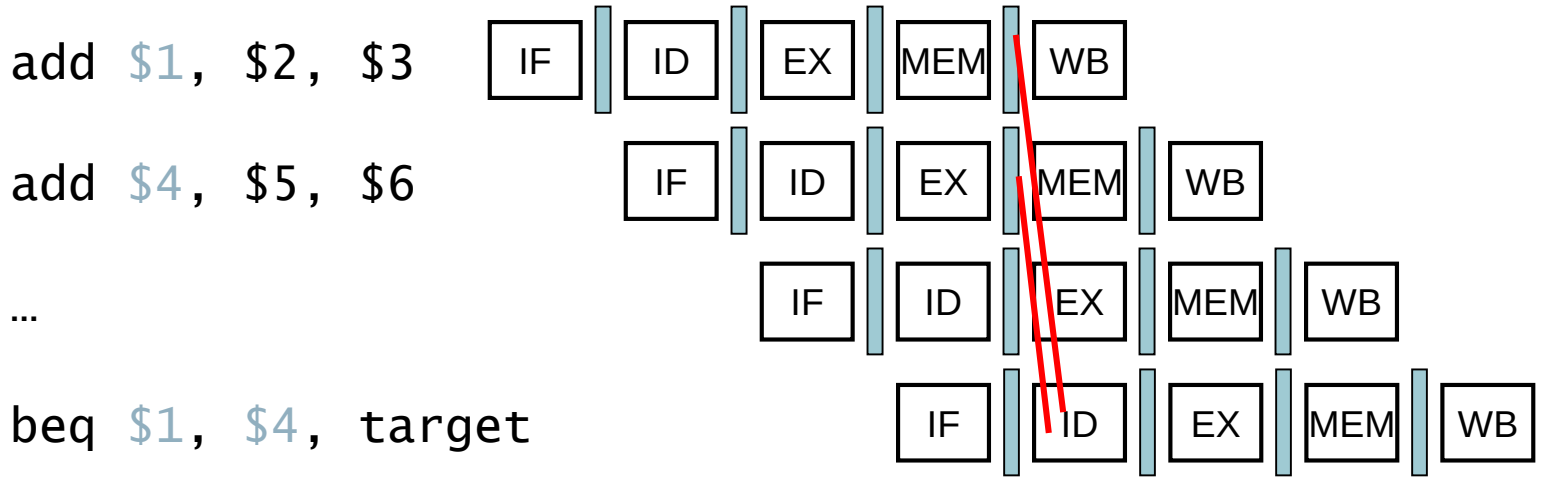


Example: Branch Taken



Data Hazards for Branches

- Since we add hardware to determine branch outcome at ID stage, we need to compare two registers at ID stage
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction ➔ resolve with forwarding

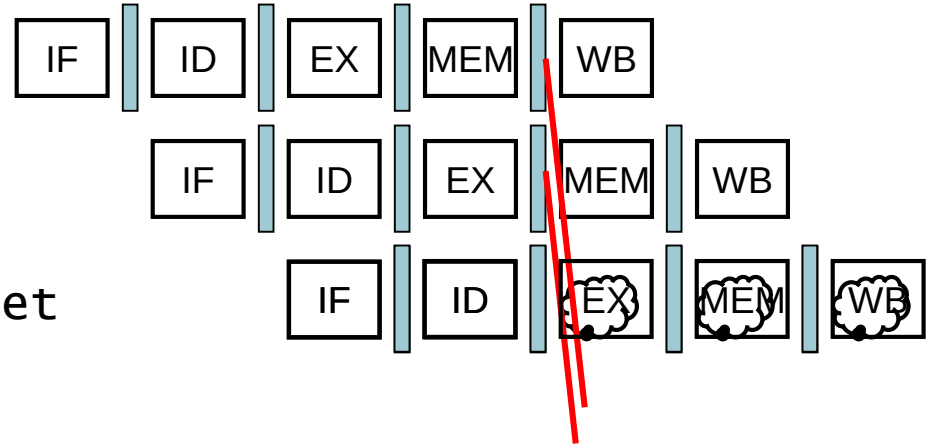


- Can resolve using forwarding

Data Hazards for Branches

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

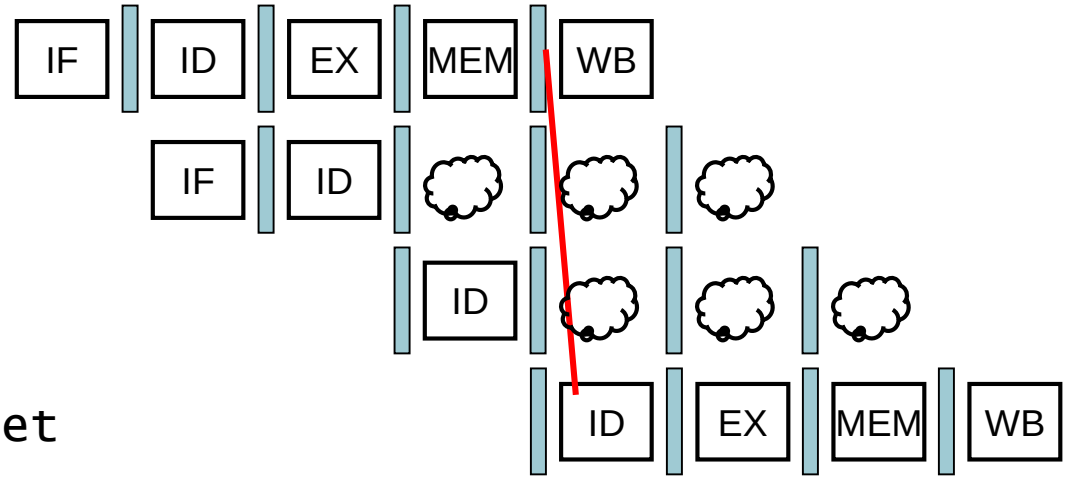
```
lw   $1, addr
add  $4, $5, $6
beq  $1, $4, target
```



Data Hazards for Branches

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

```
lw   $1, addr
beq  stalled
beq  stalled
beq  $1, $0, target
```

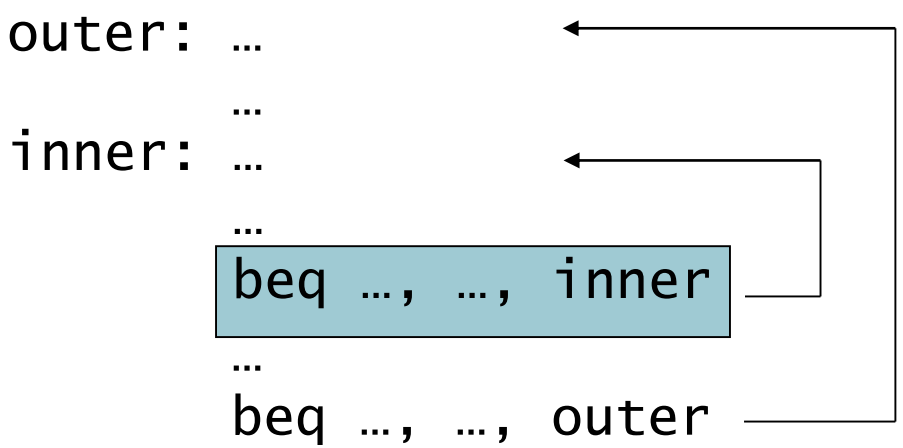


Dynamic Branch Prediction

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

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!



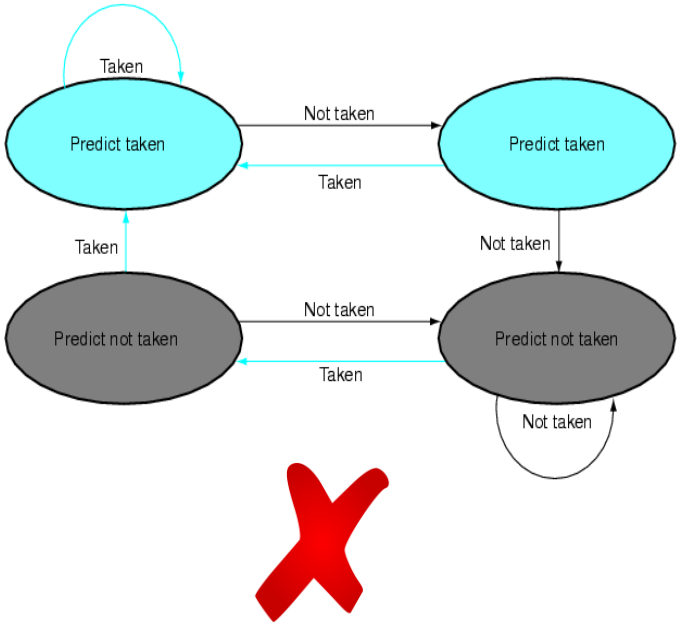
  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as *not taken* on first iteration of inner loop next time around

T T T T T N T T T T T N T T T T T N TTT

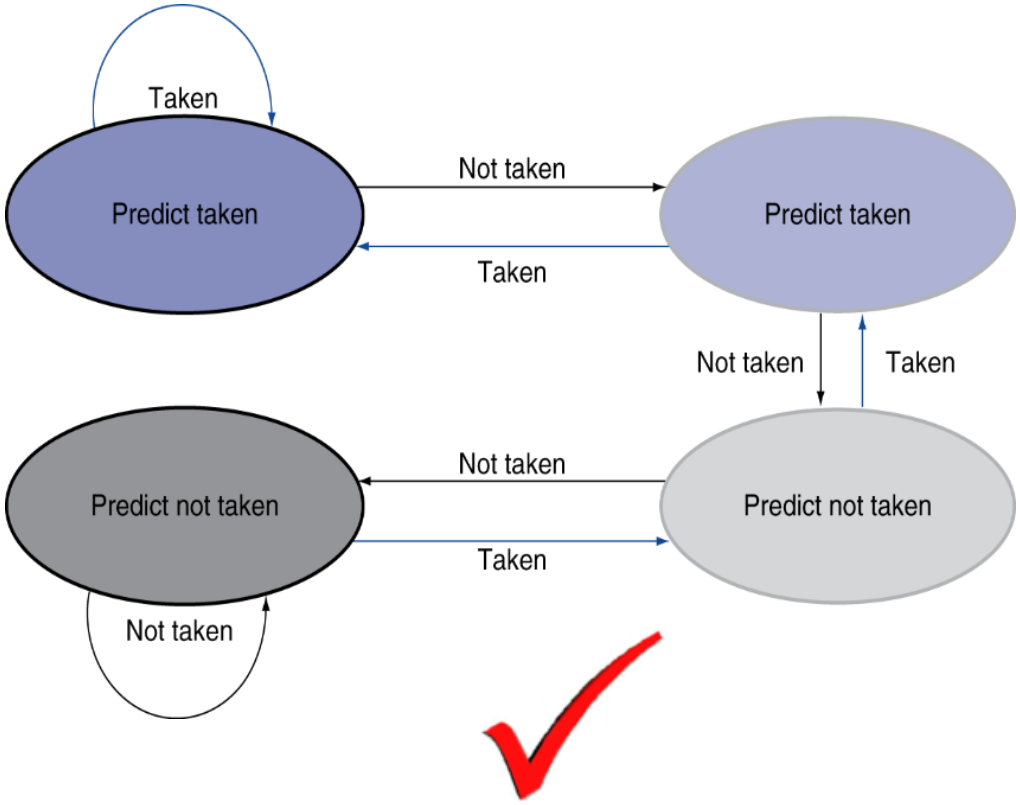
T T T T T T N T T T T T N T T T T T N TT

2-Bit Predictor

- Only change prediction on two successive mispredictions



The 4th, 5th Edition



Branch predictor

- Branch: **T – N – T – N – N – T – N**
- Predictors are initialized to **taken**.

- 1-bit: **T – N – T – N – N – T – N**
  T → T → N → T → N → N → T
- 2-bit: **T – N – T – N – N – T – N**
  T → T → T → T → T → N → T
- 2-bit: **T – N – T – N – N – N – T – N**
  T → T → T → T → T → N → N → N

Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

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, managed by a System Control Coprocessor (CP0)
- Save PC of 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 0180
  - In MIPS, a single entry point for all exceptions
  - OS decodes the status register to find the cause
  - Alternative: vectored Interrupts
    - Handler address determined by the cause
    - E.g.: Undefined opcode: C000 0000
    - Overflow: C000 0020
    - …: C000 0040

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

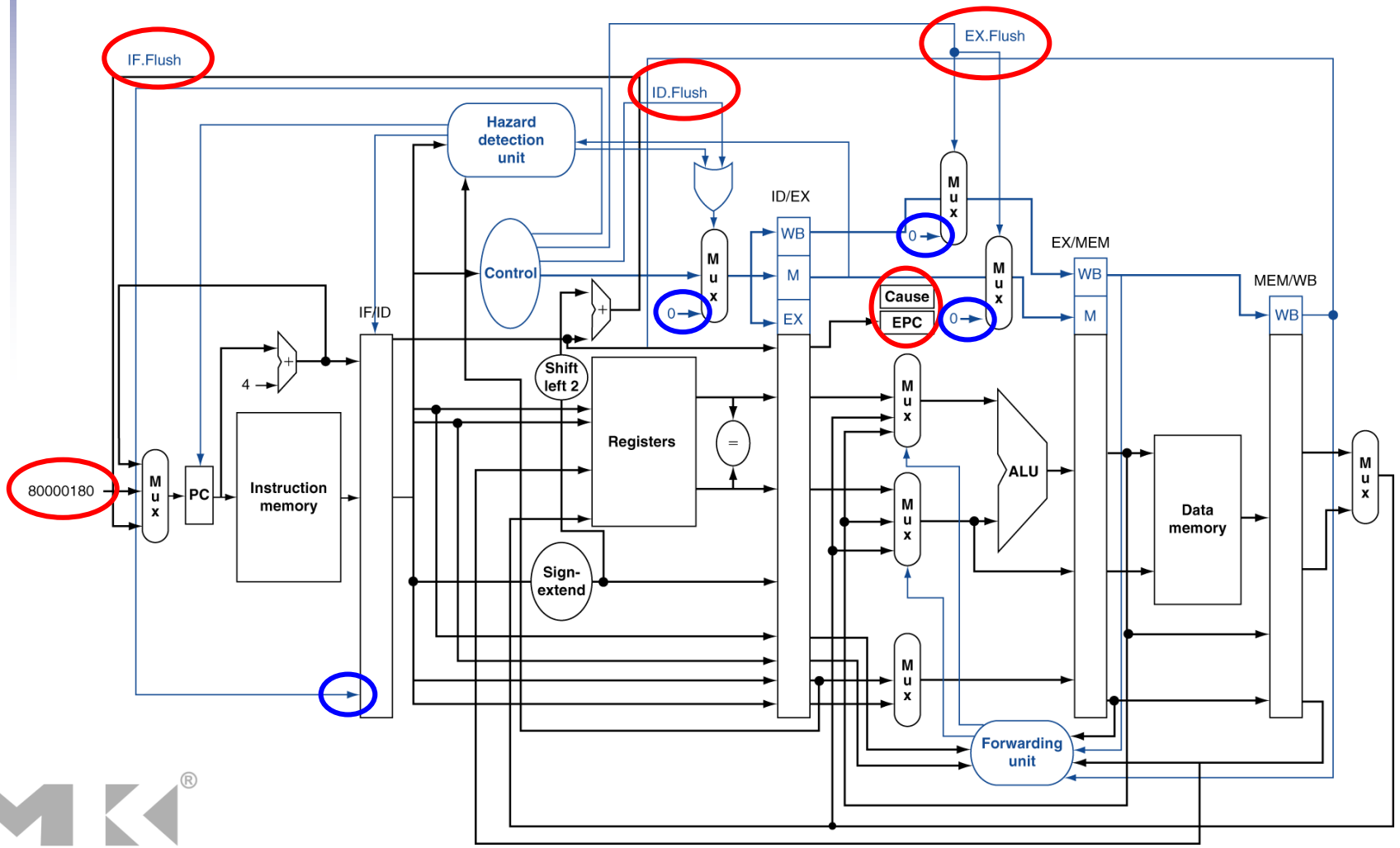
- Another form of control hazard
- Consider overflow on add in EX stage

  `add $1, $2, $1`

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

Pipeline with Exceptions

- Flush instructions at IF, ID, EX when an exception is detected at EX.
- Supply 8000 0180 to PC. Add **Cause, EPC, IF.Flush, ID.Flush, EX.Flush**



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

Exception Example

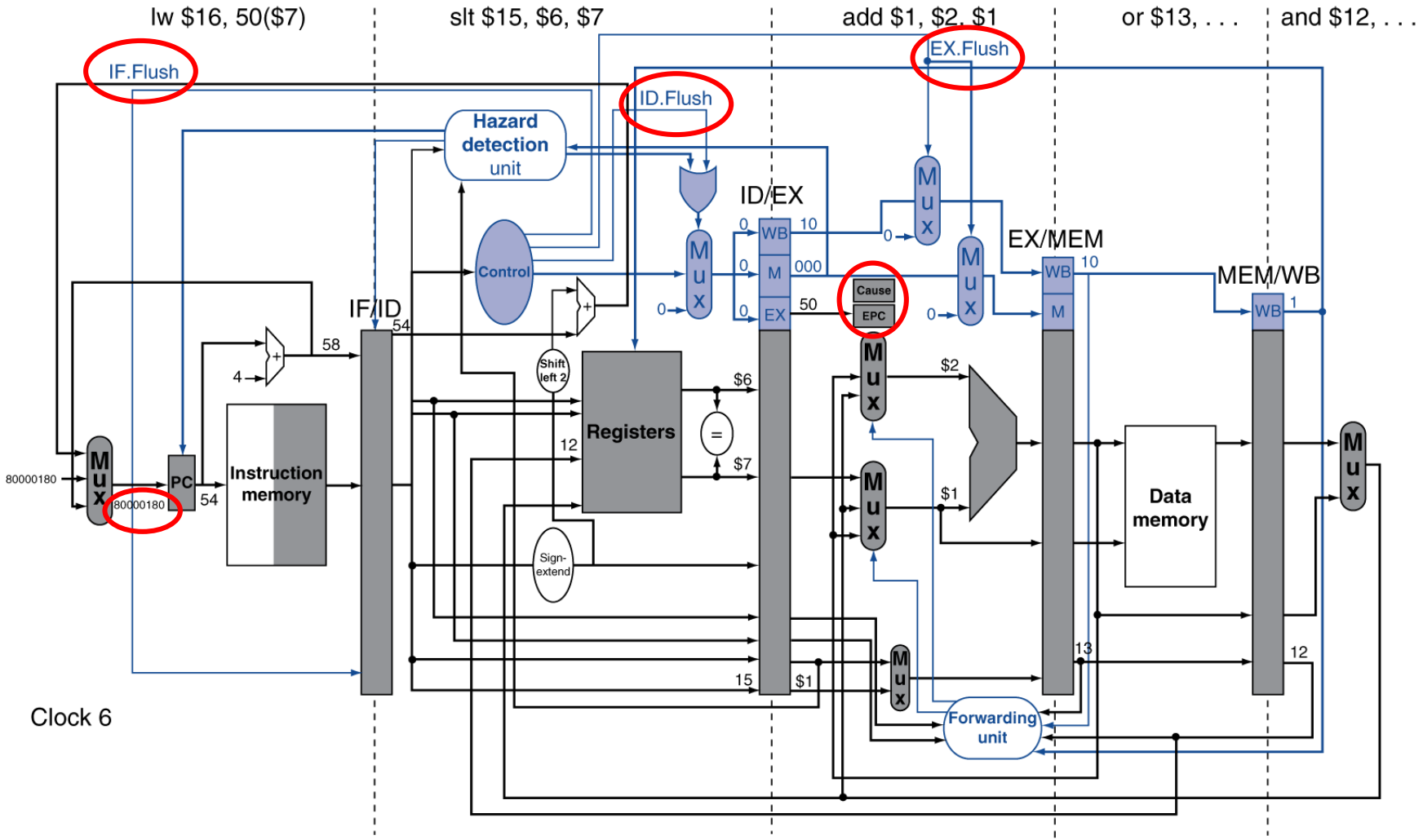
- Exception on add in

```
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)
…
```

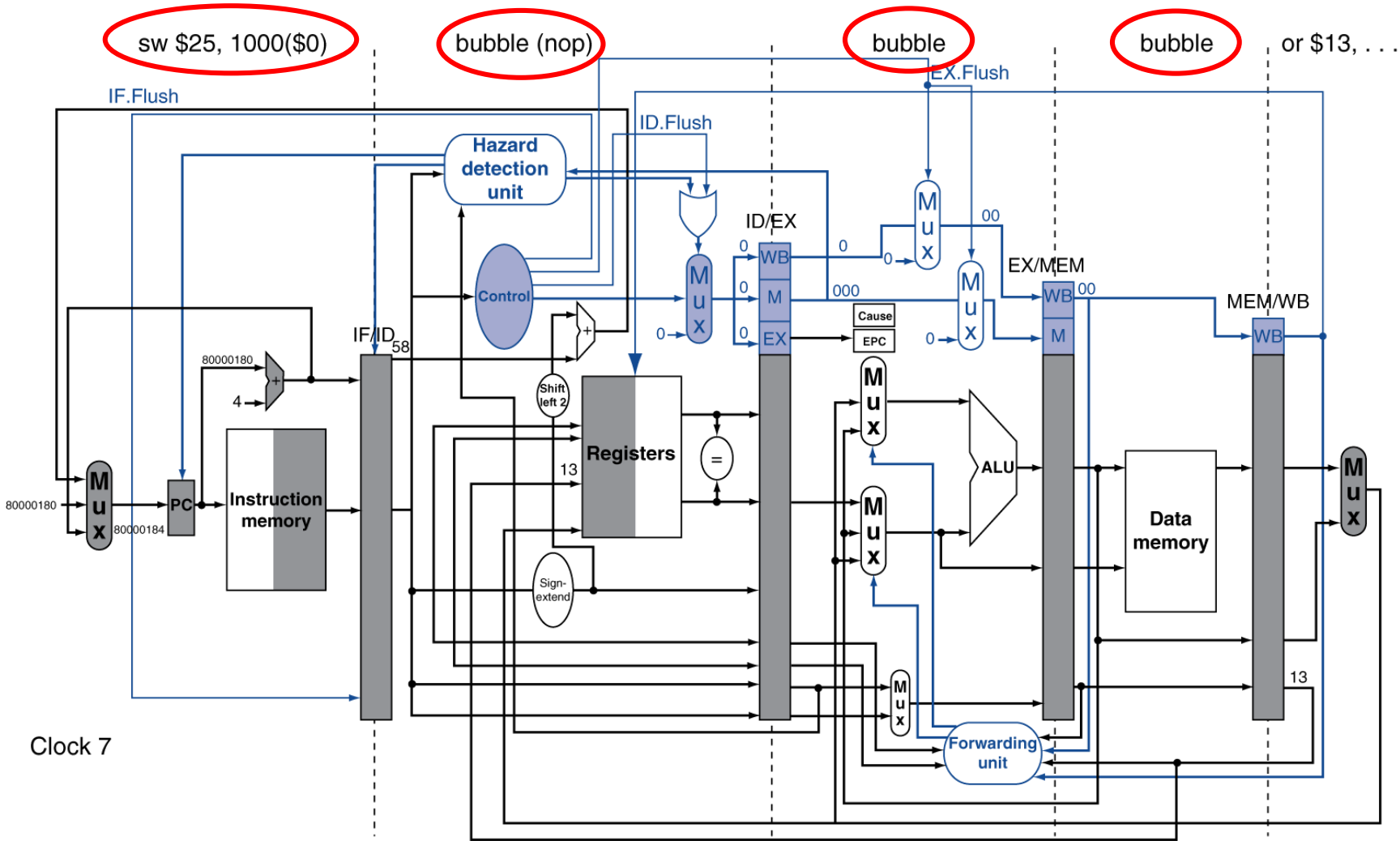
- Handler

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

Exception Example



Exception Example



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 ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ 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
  - ⇒ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

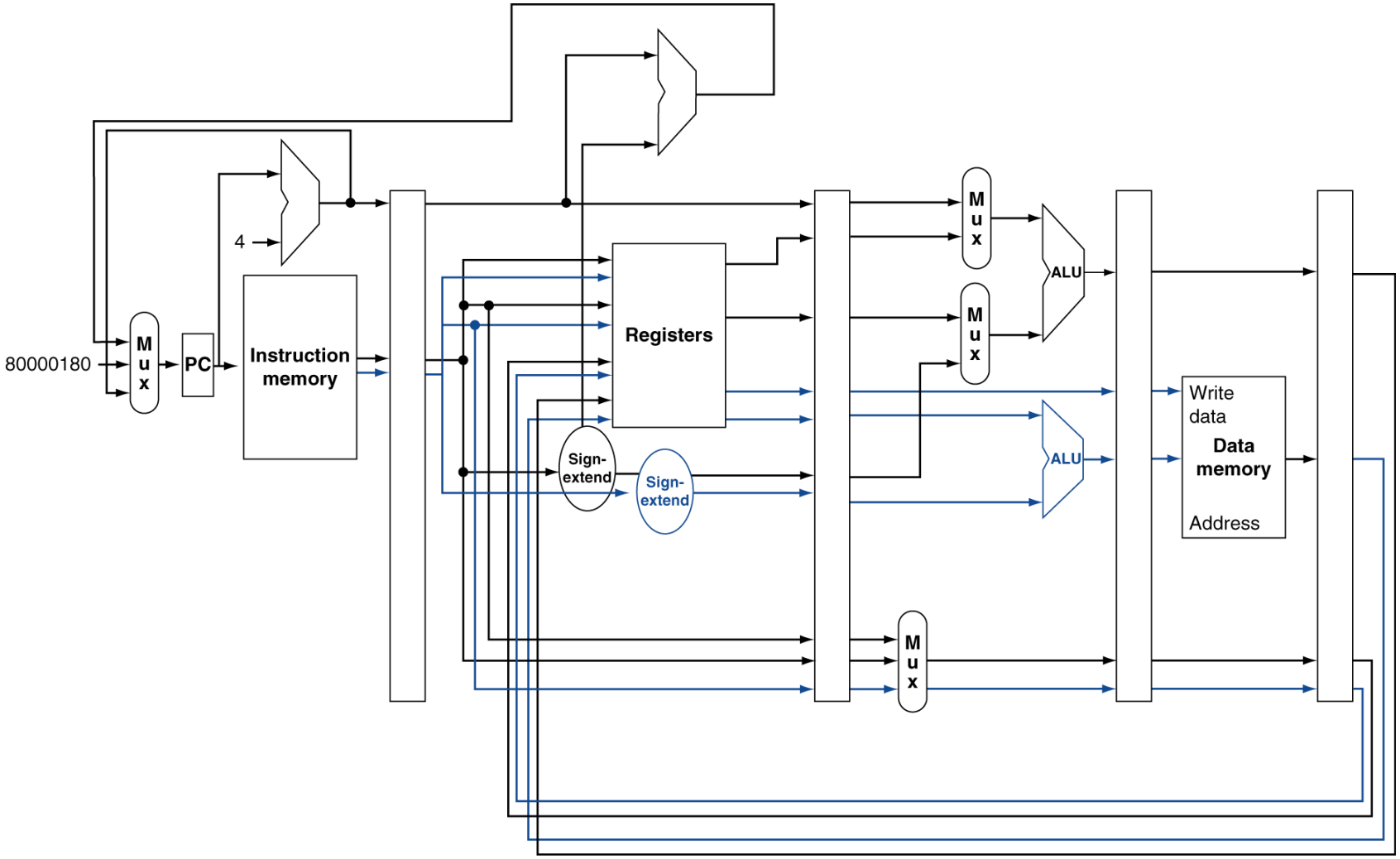
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

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

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      bne  $s1, $zero, Loop
```

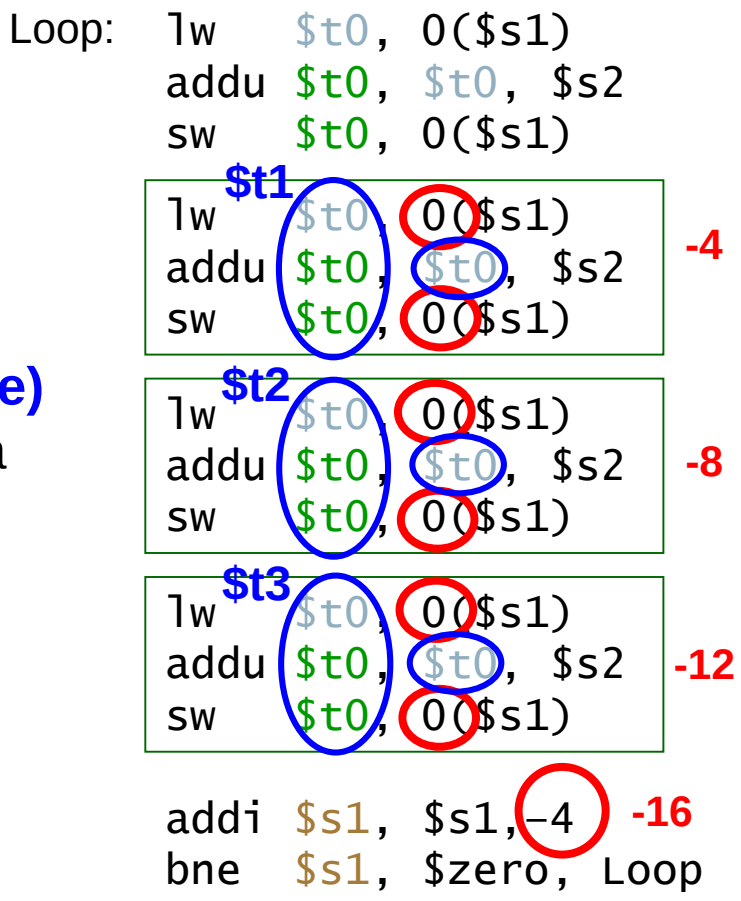
Antidependence (name dependence)

-The ordering forced by the reuse of a name (register), ***not*** true data dependence.

Register Renaming:

- to remove anti-dependence.

- e.g., $t0 ➔ $t0, $t1, $t2, $t3



```
lw   $t0, 0($s1)
addu $t0, $t0, $s2
sw   $t0, 0($s1)
```

```
lw   $t1, -4($s1)
addu $t1, $t0, $s2
sw   $t1, -4($s1)
```

```
lw   $t2, -8($s1)
addu $t2, $t0, $s2
sw   $t2, -8($s1)
```

```
lw   $t3, -12($s1)
addu $t3, $t0, $s2
sw   $t3, -12($s1)
```

```
addi $s1, $s1,-16
bne  $s1, $zero, Loop
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
		nop	2
	addu $t0, $t0, $s2	nop	3
		sw $t0, 0($s1)	4
			5

Loop Unrolling Example

	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, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

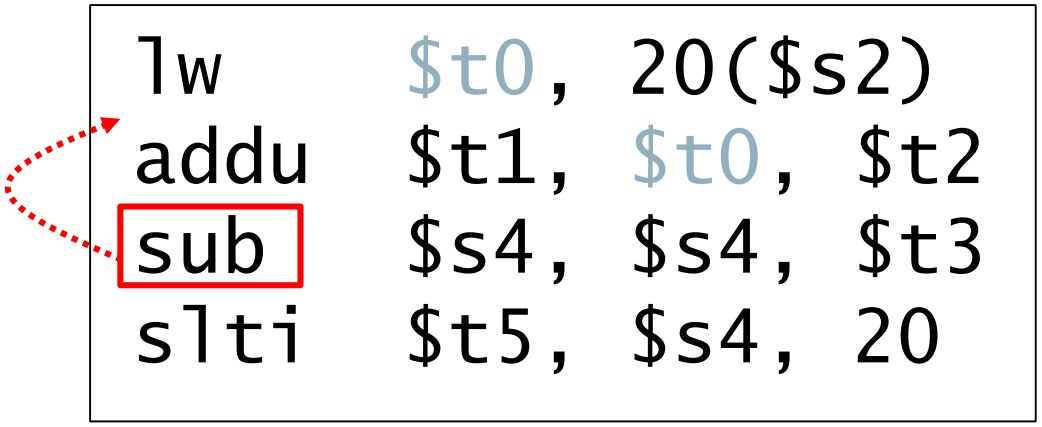
- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Dynamic Multiple Issue

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

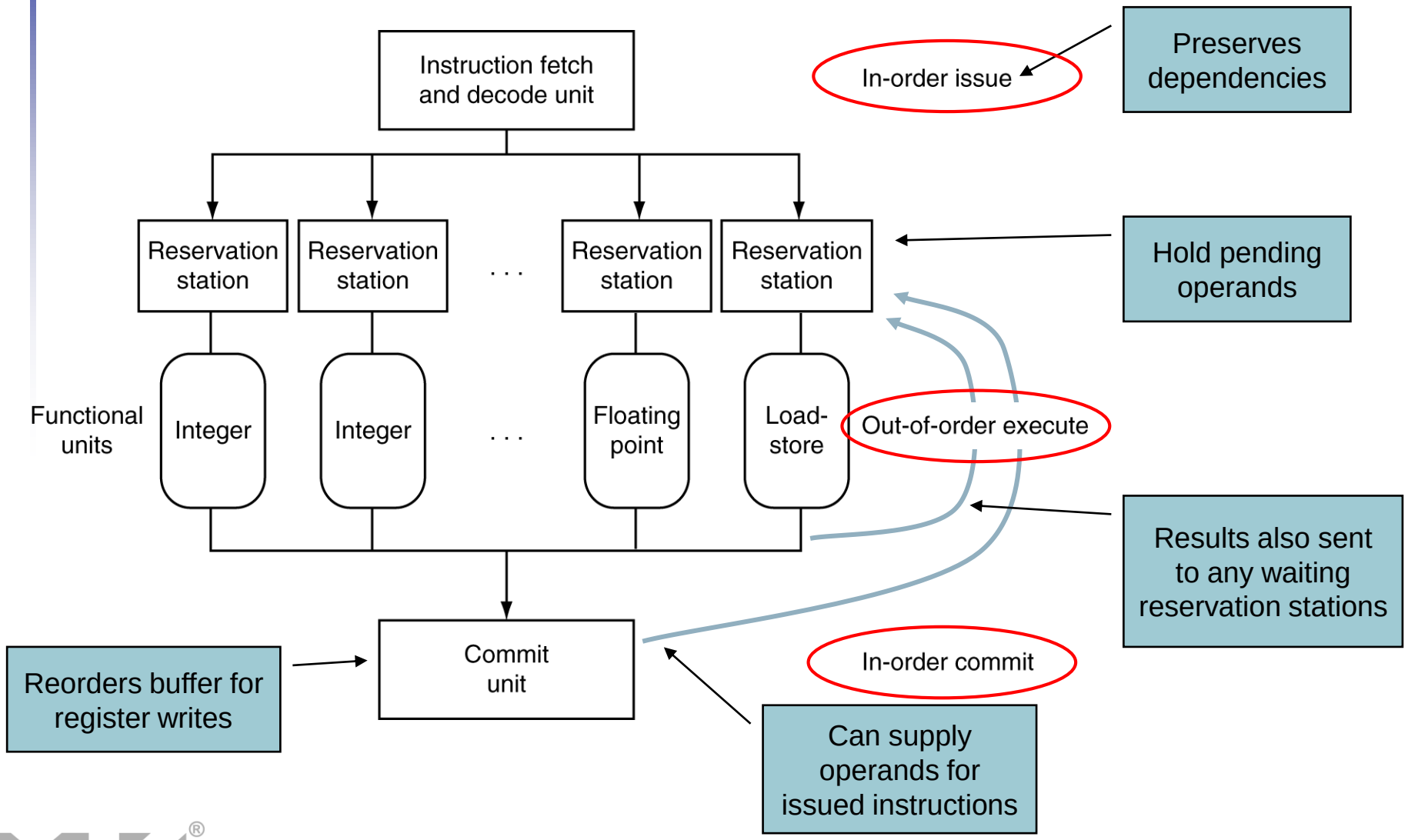
Dynamic Pipeline Scheduling

- It is hardware support
- Allow the CPU to execute instructions out of order to avoid stalls (out-of-order execution)
  - But commit result to registers in order (in-order commit)



  - Can start sub while addu is waiting for lw

Dynamically Scheduled CPU



Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by another function unit directly.
    - Register update may not be required

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code? (Note: dynamic scheduling is hardware support)
- 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

Power Efficiency

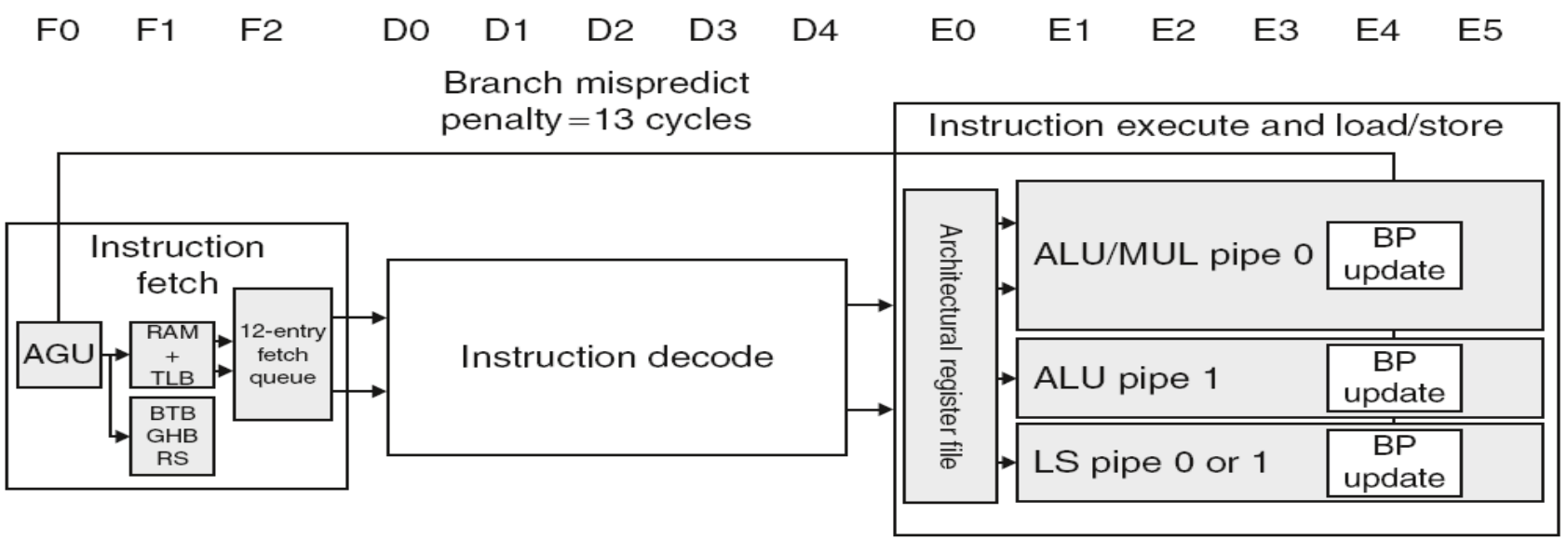
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

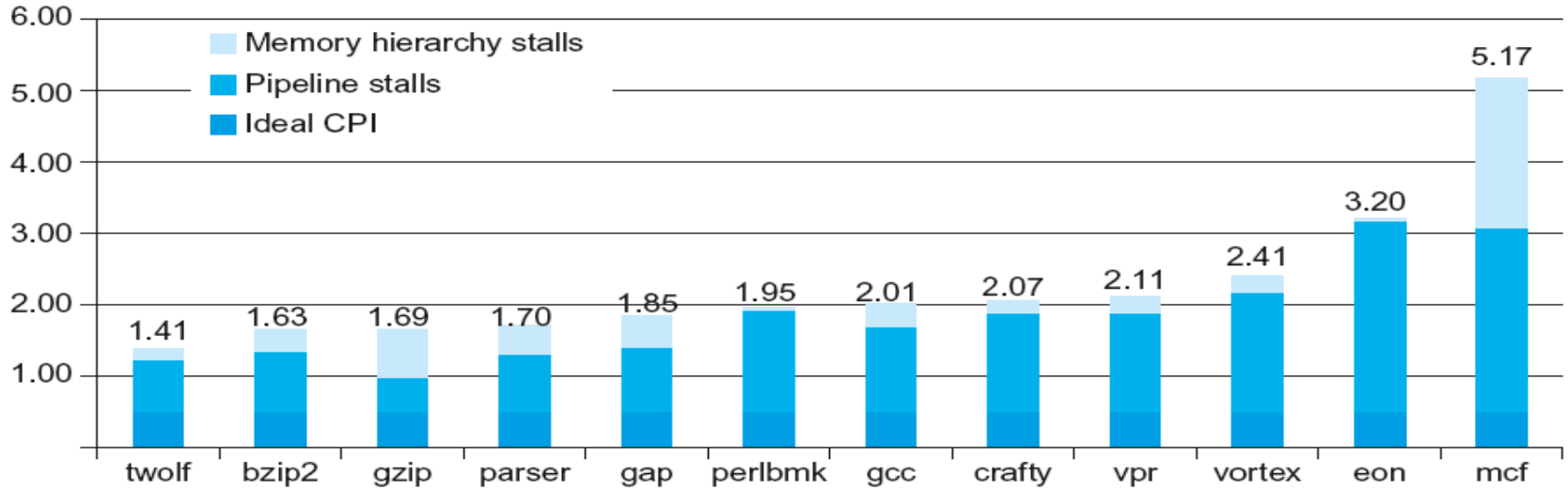
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

Cortex A8 and Intel i7

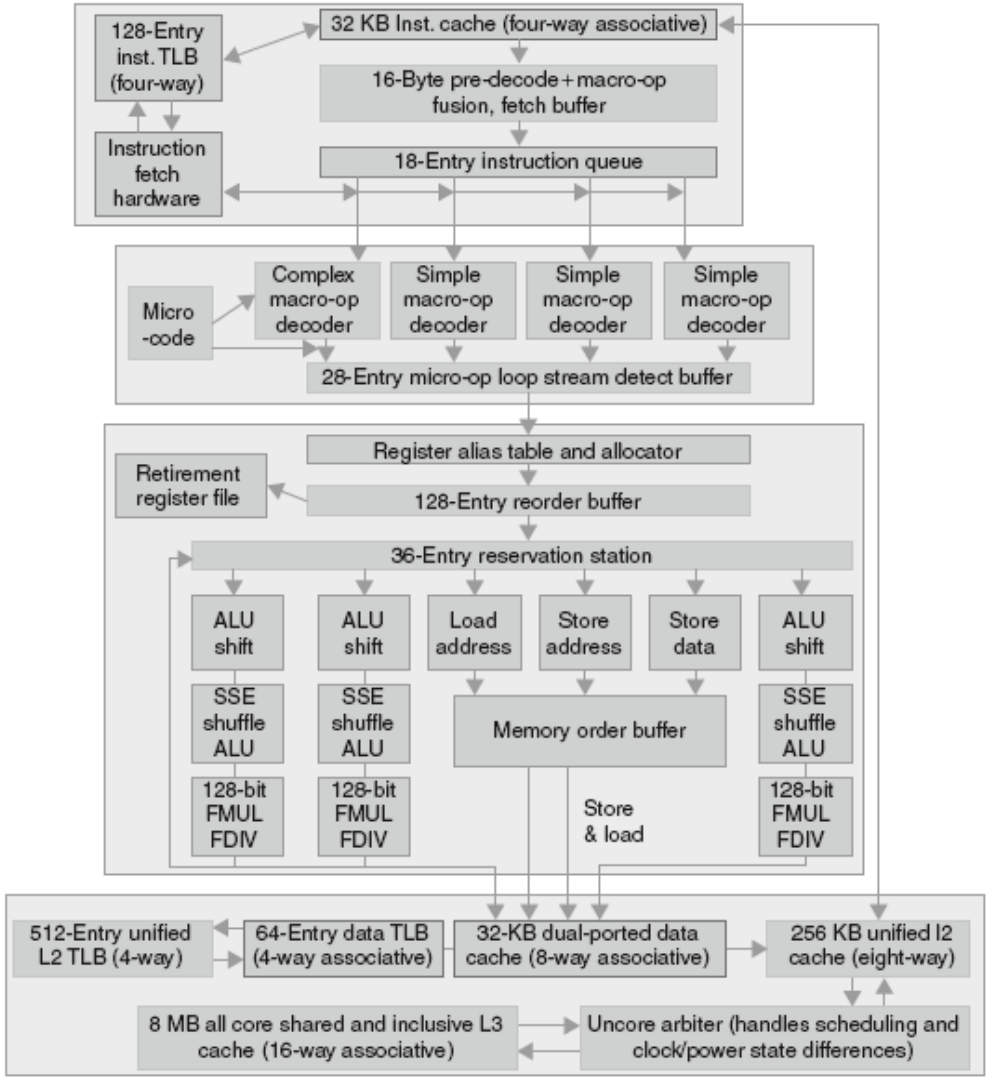
Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

ARM Cortex-A8 Pipeline & Performance

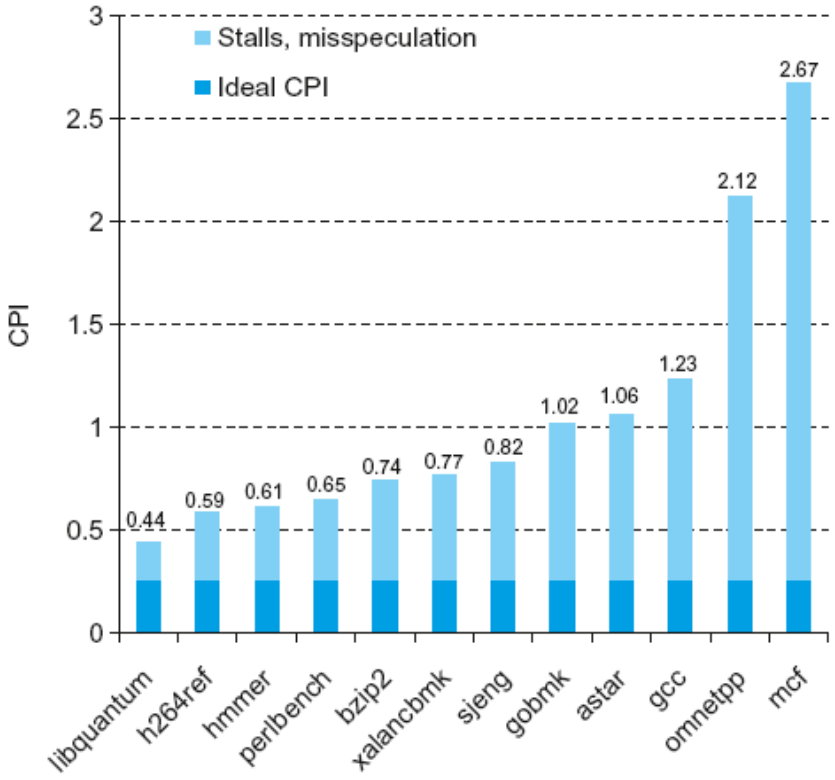


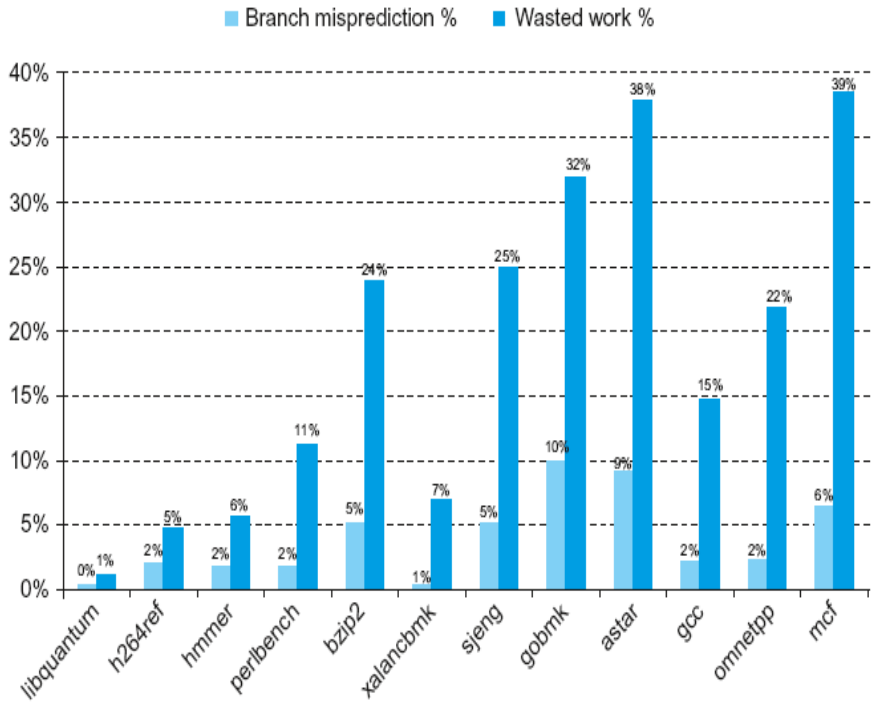


Core i7 Pipeline

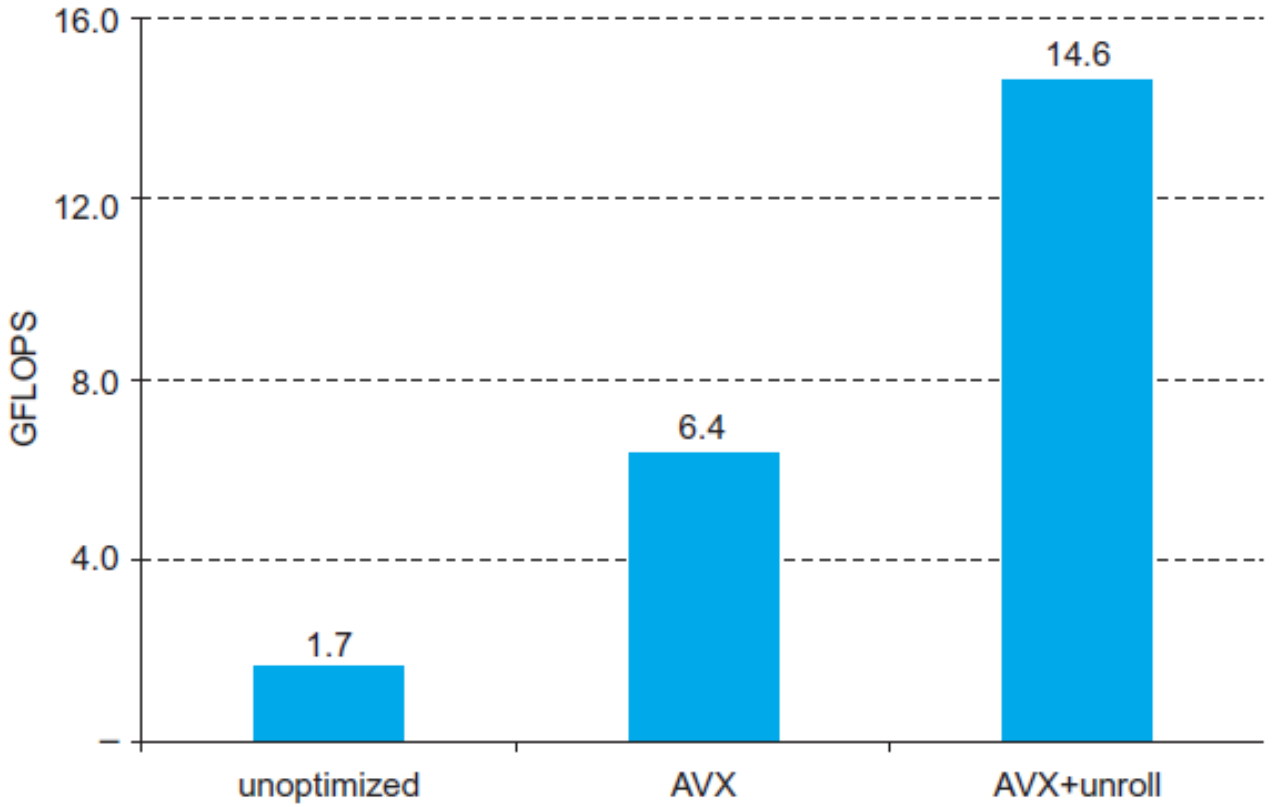


Core i7 Performance

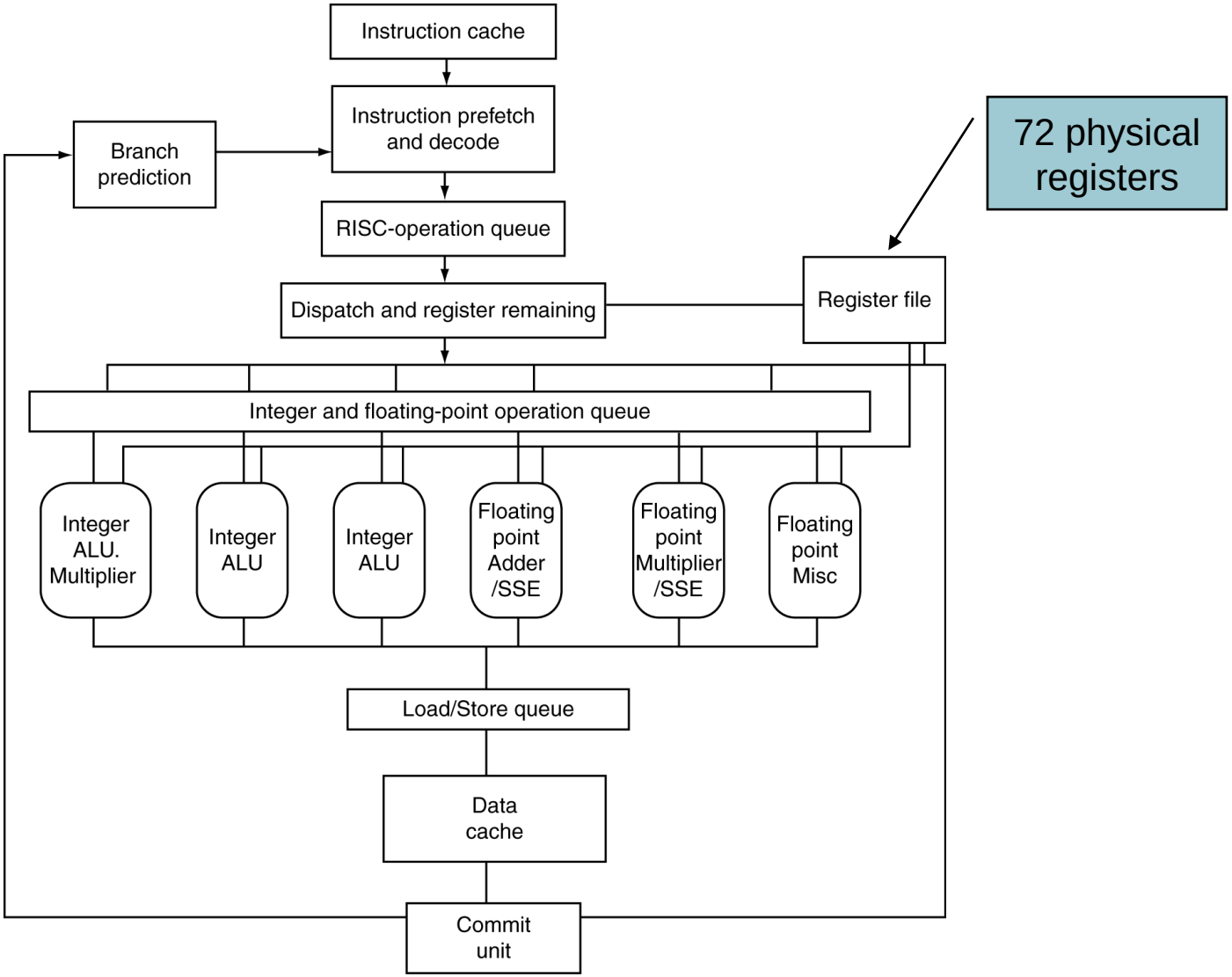




Performance Impact

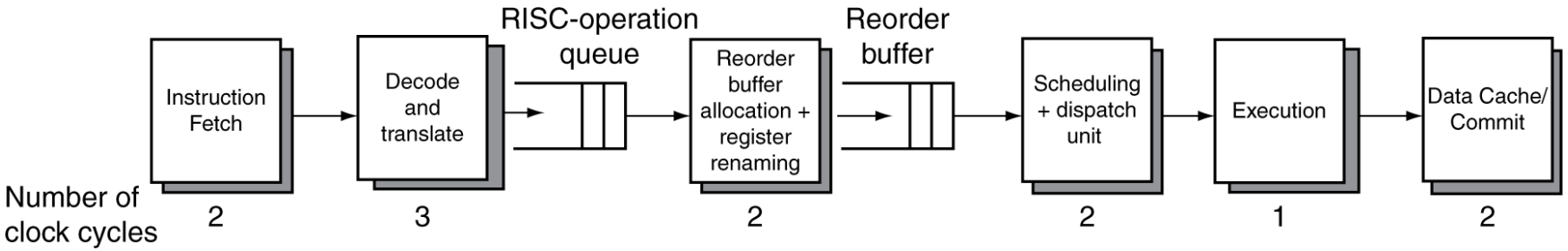


The Opteron X4 Microarchitecture



The Opteron X4 Pipeline Flow

- For integer operations



  - FP is 5 stages longer
  - Up to 106 RISC-ops in progress
- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays

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

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