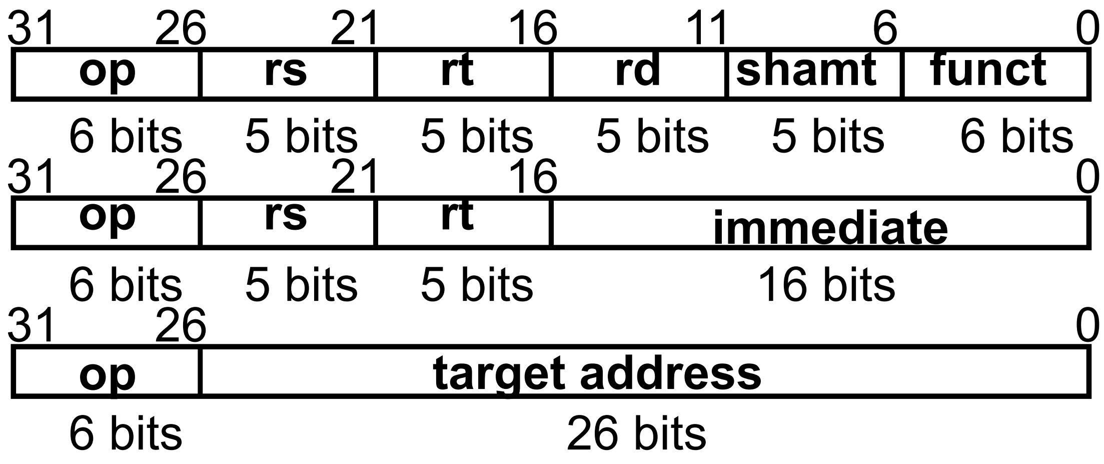
PROCESSOR DATAPATH

Introduction

- MIPS ISA
  - 3 Instruction Types
    - R-type, I-type, J-type
- Datapath and Control Unit
  - Simplified
  - Pipelined
- Simple subset
  - Memory reference: lw, sw
  - Arithmetic/logical: add, addi, sub, and, or, slt
  - Control transfer: beq, j

Instruction Format Review

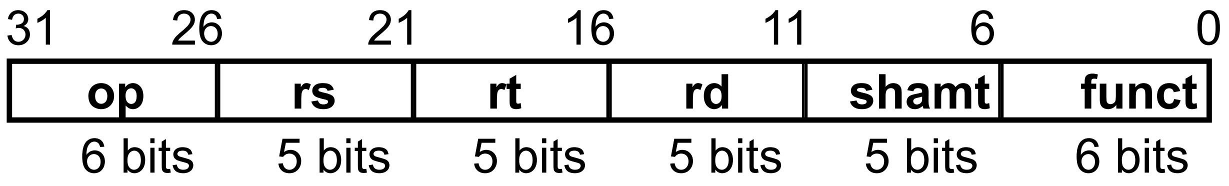
- Three instruction formats
  - R-type
  - I-type
  - J-type



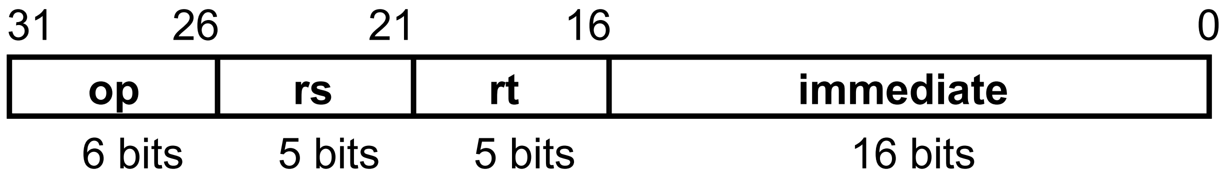
- Fields:
  - op: operation of the instruction
  - rs, rt, rd: source/destination register specifiers
  - shamt: shift amount
  - funct: selects variant of the operation in the “op” field
  - address/immediate: address offset or immediate value
  - target address: target address of the jump instruction

MIPS Subset

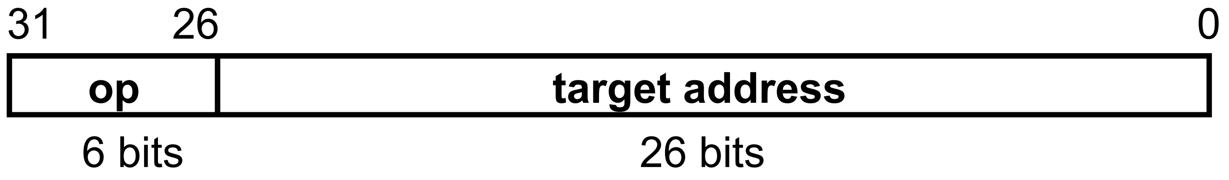
- Add, Subtract, AND, OR, SLT
  - add rd, rs, rt
  - sub rd, rs, rt
  - and rd, rs, rt
  - or rd, rs, rt
  - slt rd, rs, rt



- ADD Immediate
  - addi rt, rs, imm16
- Load, Store
  - lw rt, rs, imm16
  - sw rt, rs, imm16



- BRANCH
  - beq rs, rt, imm16
- JUMP:
  - j target

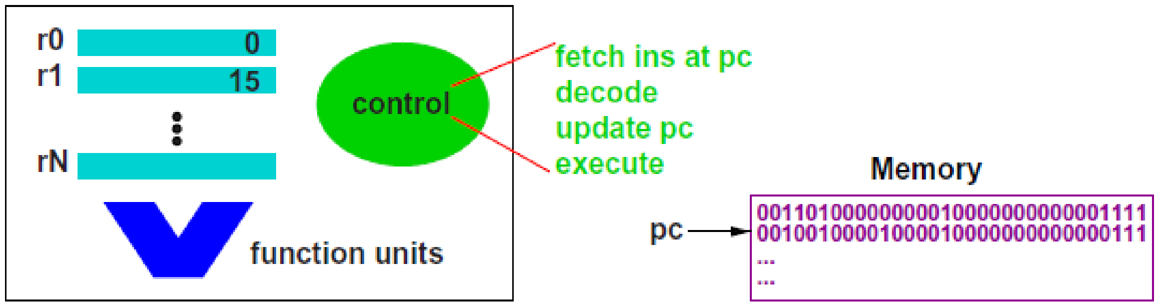


Instruction Execution

- For every instruction:
  1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.

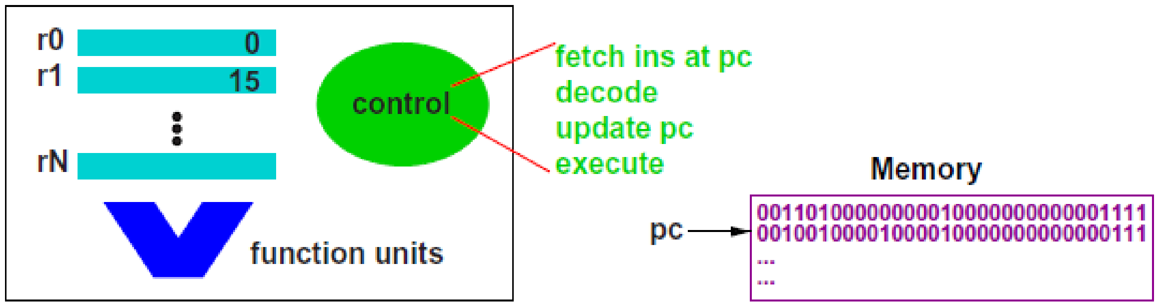
  1. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require that we read two registers.

Instruction Execution



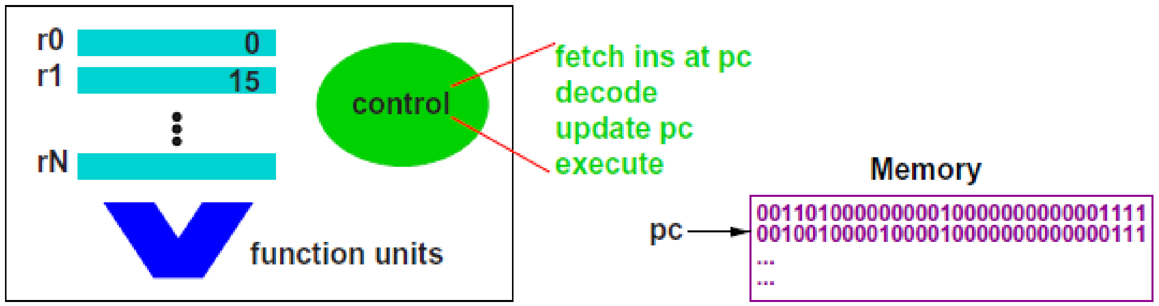
- Fetch:
  - Get the next instruction: stored at PC

Instruction Execution



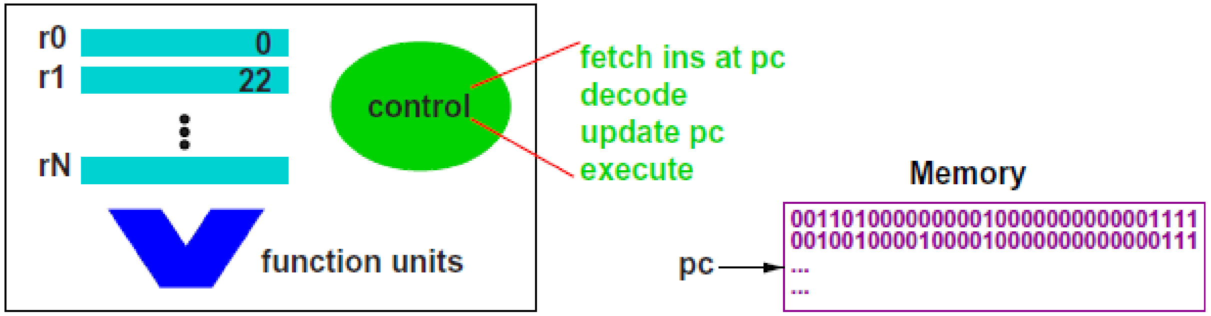
- Decode:
  - Opcode determines if the instruction is arithmetic/logical, memory logical, or branch.

Instruction Execution



- Update PC: PC ← target address or PC + 4
  - If the instruction is a branch, we have to calculate the target address
  - If the instruction is not a branch, the new PC will be PC + 4

Instruction Execution



- Update PC: PC ← target address or PC + 4
  - If the instruction is a branch, we have to calculate the target address
  - If the instruction is not a branch, the new PC will be PC + 4

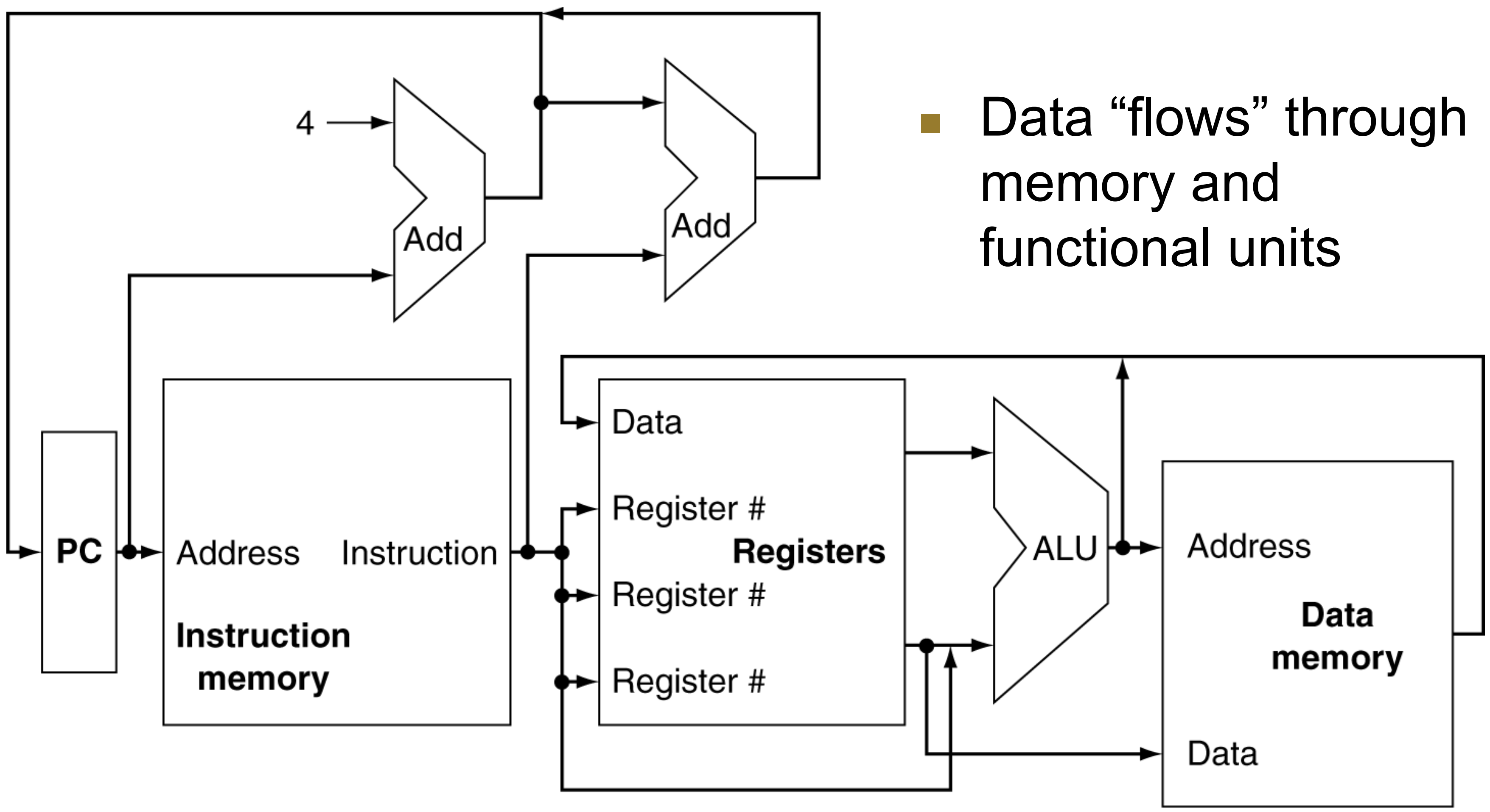
Instruction Execution

- Execute:
  - Depending on instruction class, we use the ALU to calculate:
    - Arithmetic result
    - Memory address for load/store
    - Branch target address

Instruction Execution

- To read and write registers, we need a register file
  - R-types read two registers and write one register
  - Memory-reference instructions read one register
    - Also access data memory

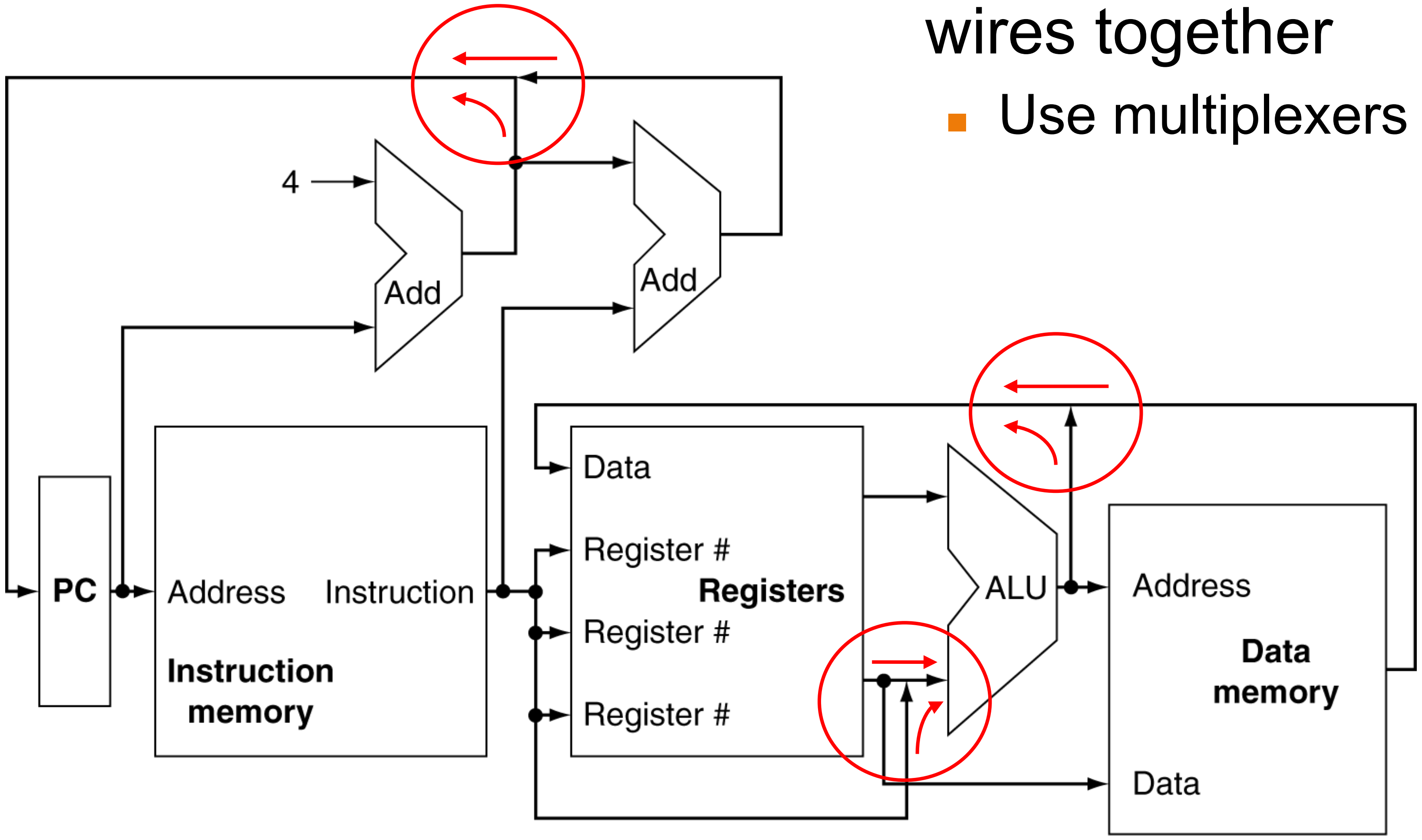
Processor Overview



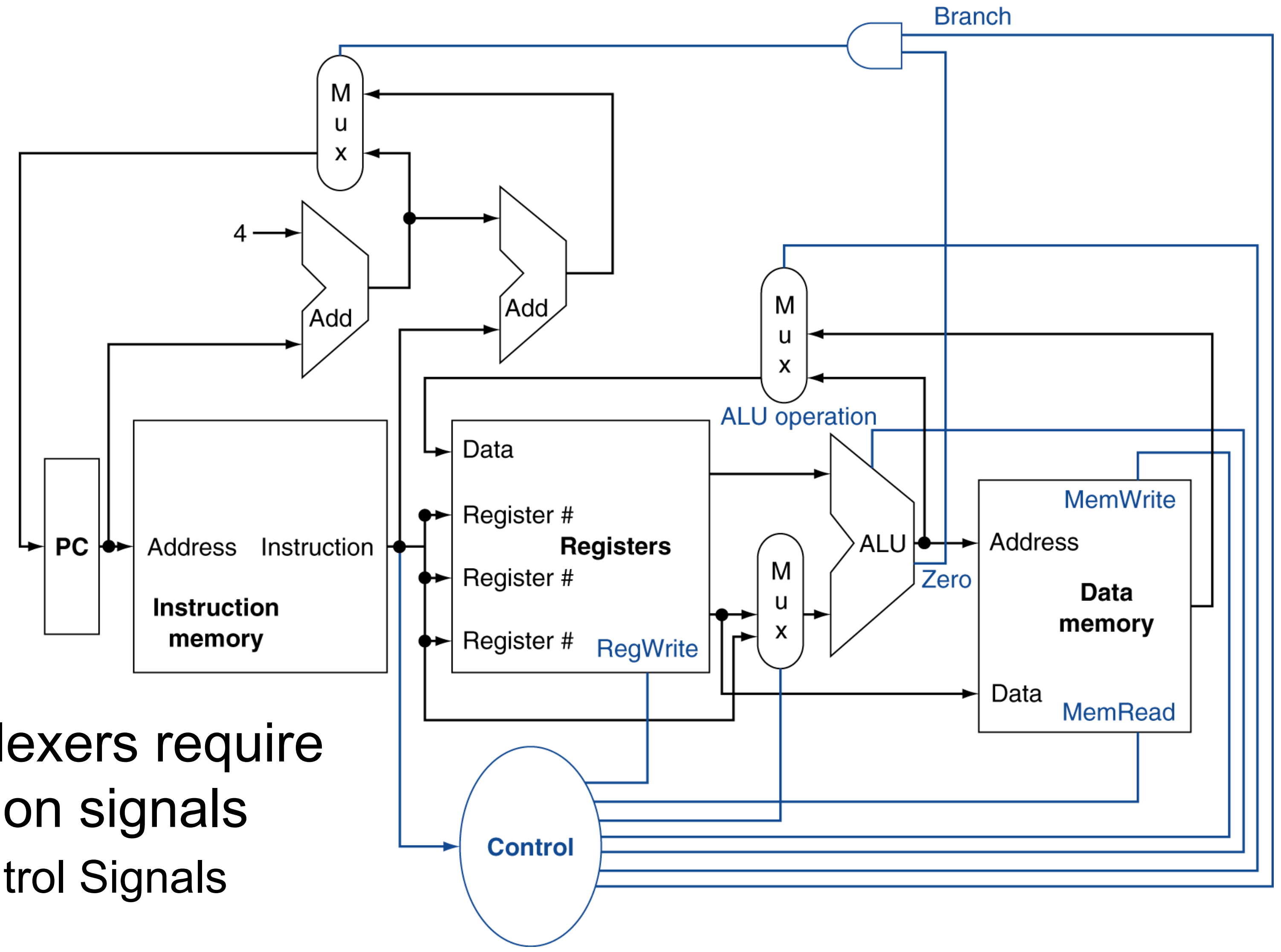
- Data "flows" through memory and functional units

Processor Overview

- Can't just join wires together
  - Use multiplexers



Processor Overview



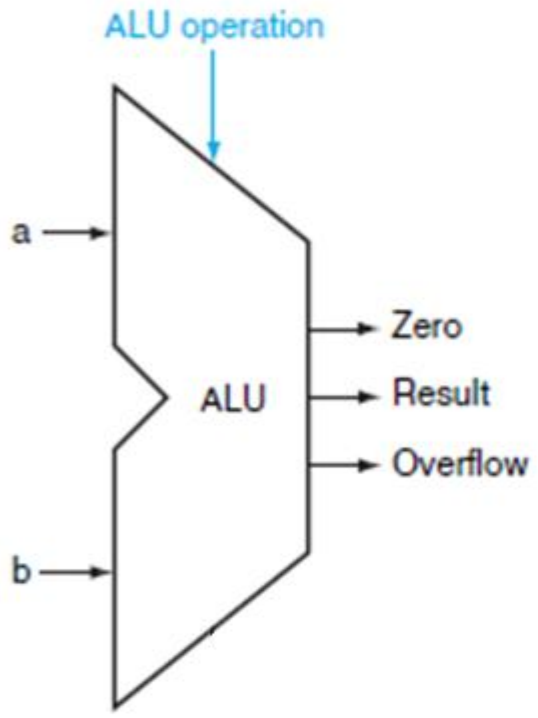
- Multiplexers require selection signals
  - Control Signals

Signals

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses

Logic Design Review

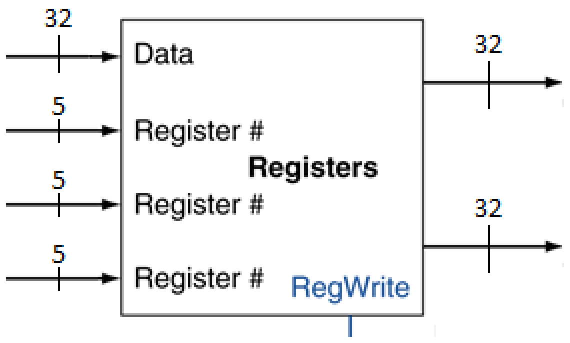
- Combinational Circuits
  - Output depends only on current input
  - Operate on data
- Example: ALU
  - Given the same input a combinational circuit will always produce the same output



Logic Design Review

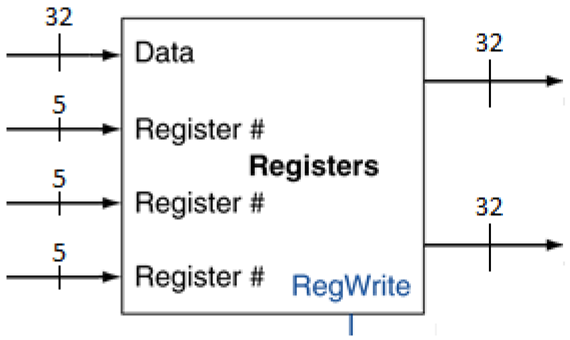
- Sequential Circuits
  - Have “state”
  - Output depends on current input and previous outputs

- Example: Register File
  - Output depends on the contents of the registers in the file



MIPS Register File

- 32 32-bit registers
- One write bus
- Two read buses
- Selection Inputs
  - RegWrite (write enable signal)
  - Reg. Source A
  - Reg. Source B
  - Reg. Destination



Write Control Signals

Every Clock Cycle

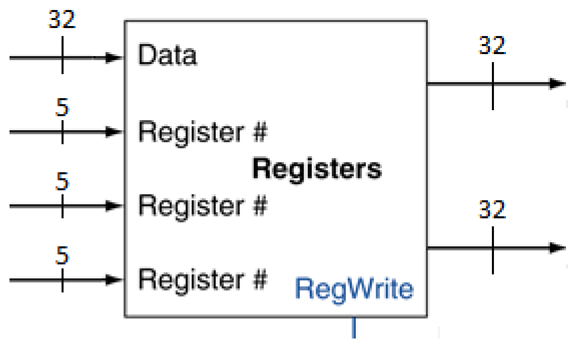
- We do not show a write control signal when a state element is written on every active clock edge.
  - Program Counter

When Necessary

- If a state element is not updated on every clock, then an explicit write control signal is required.
  - Register File

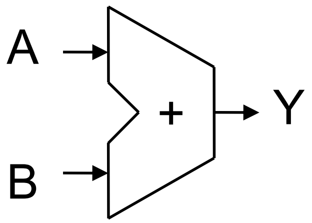
Datapath Elements

- Most elements have 32-bit wide inputs and outputs
- Buses labeled with their width

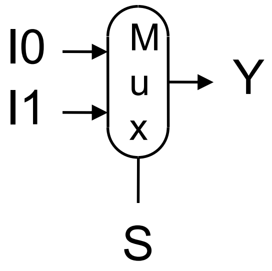


Datapath Elements - Combinational

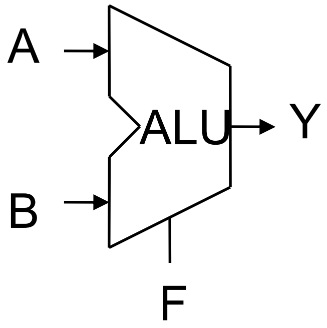
- Adder
  - Y = A + B



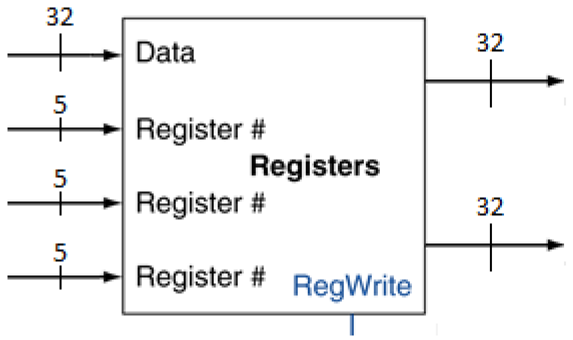
- Multiplexer
  - Y = S ? I1 : I0

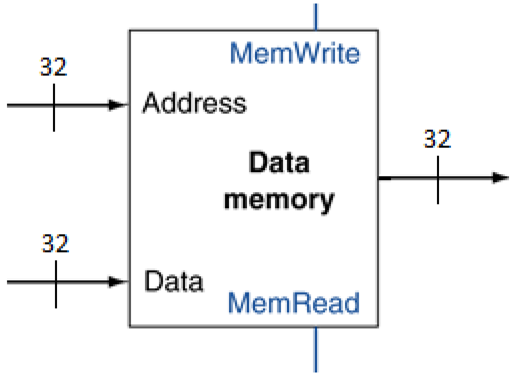


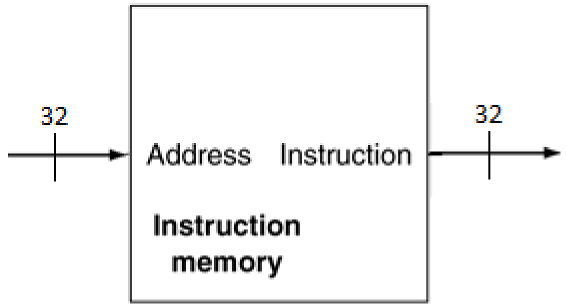
- Arithmetic/Logic Unit
  - Y = F(A, B)



Datapath Elements - Sequential

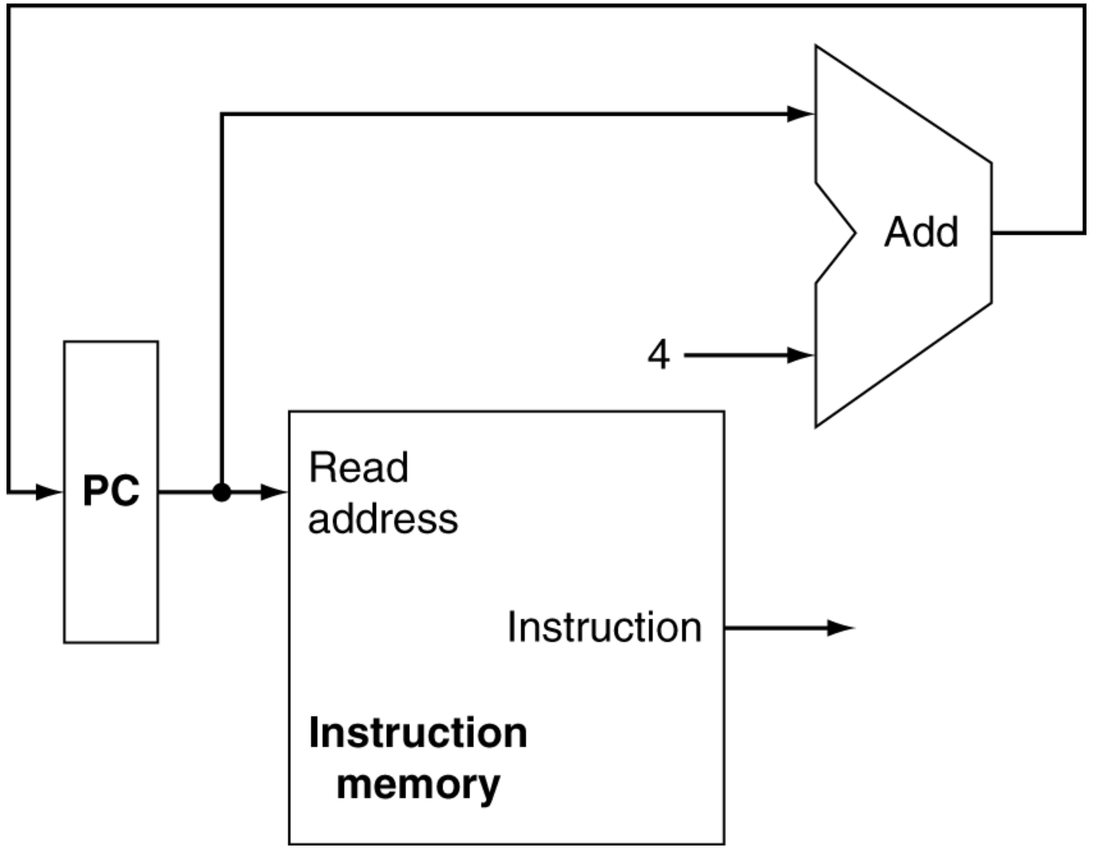






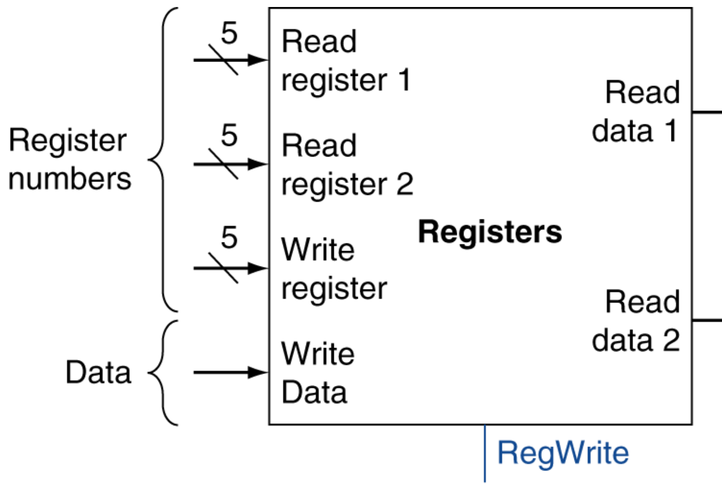
Fetch Elements

- Memory Unit
- Program Counter
- Adder

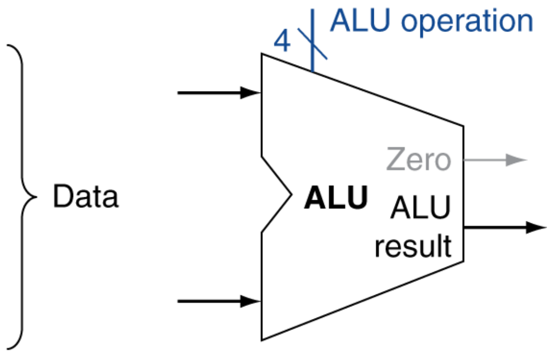


R-Type Instructions

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



a. Registers

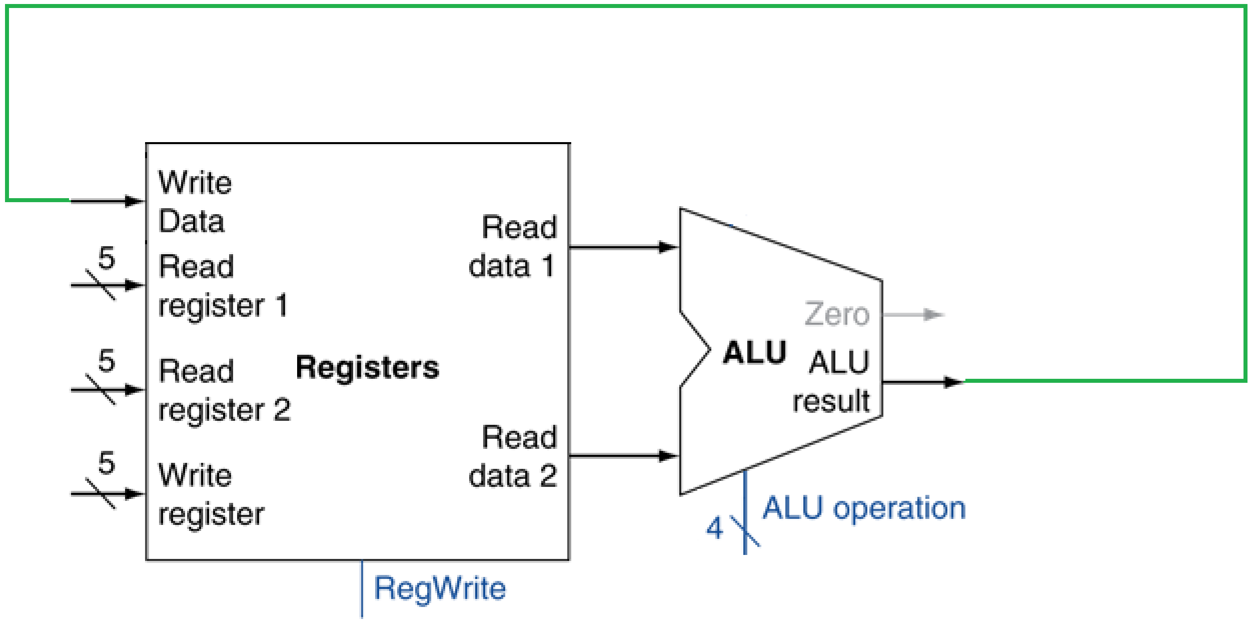


b. ALU

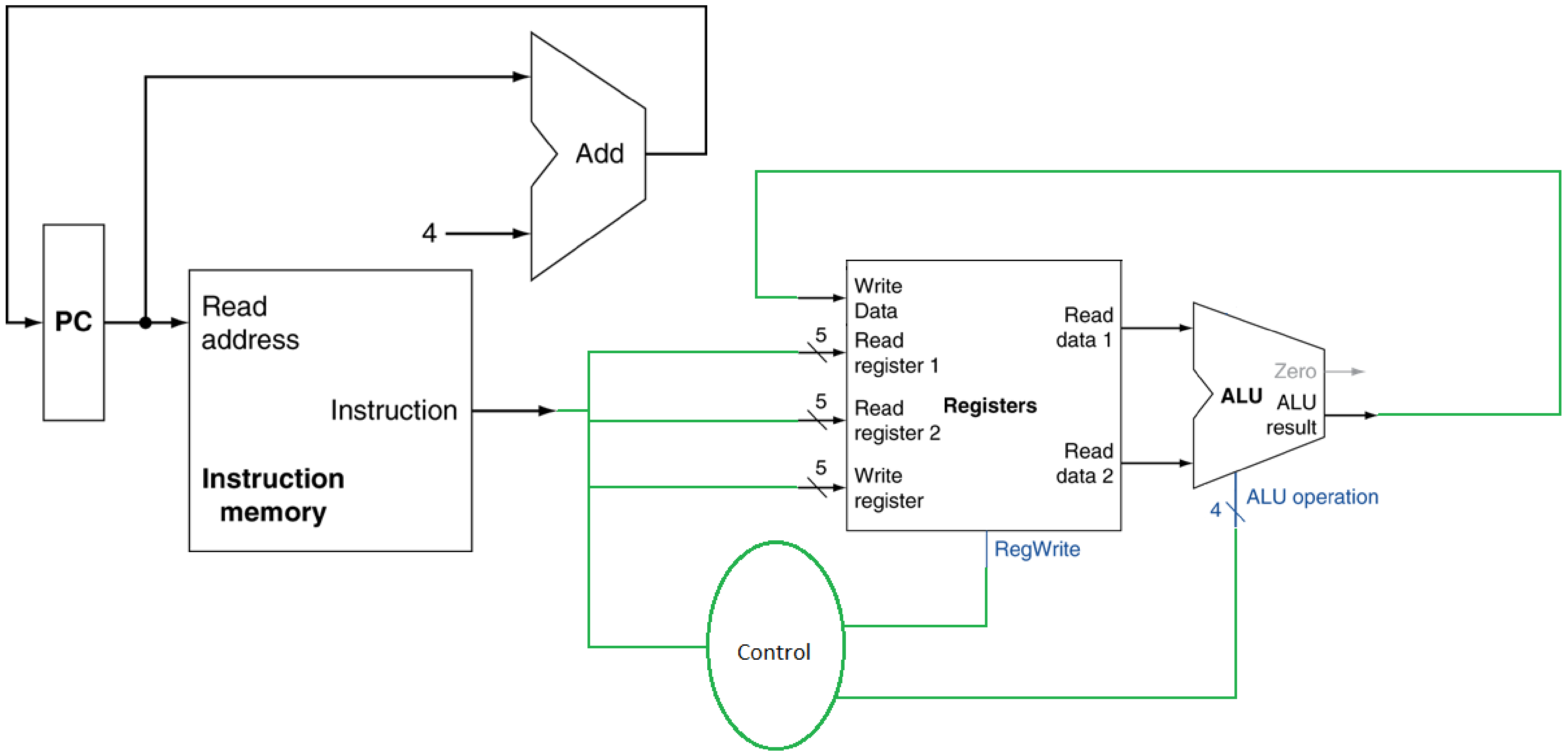
R-Type Instructions

add $t1, $t2, $t3

- RW = 9
- RA = 10
- RB = 11
- ALUop = “add”



Datapath: R-Type Instructions

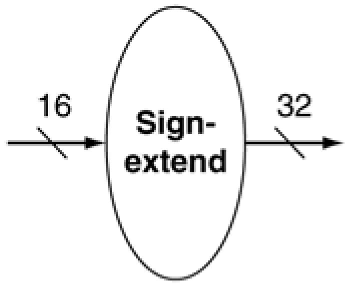


I-Type Instructions

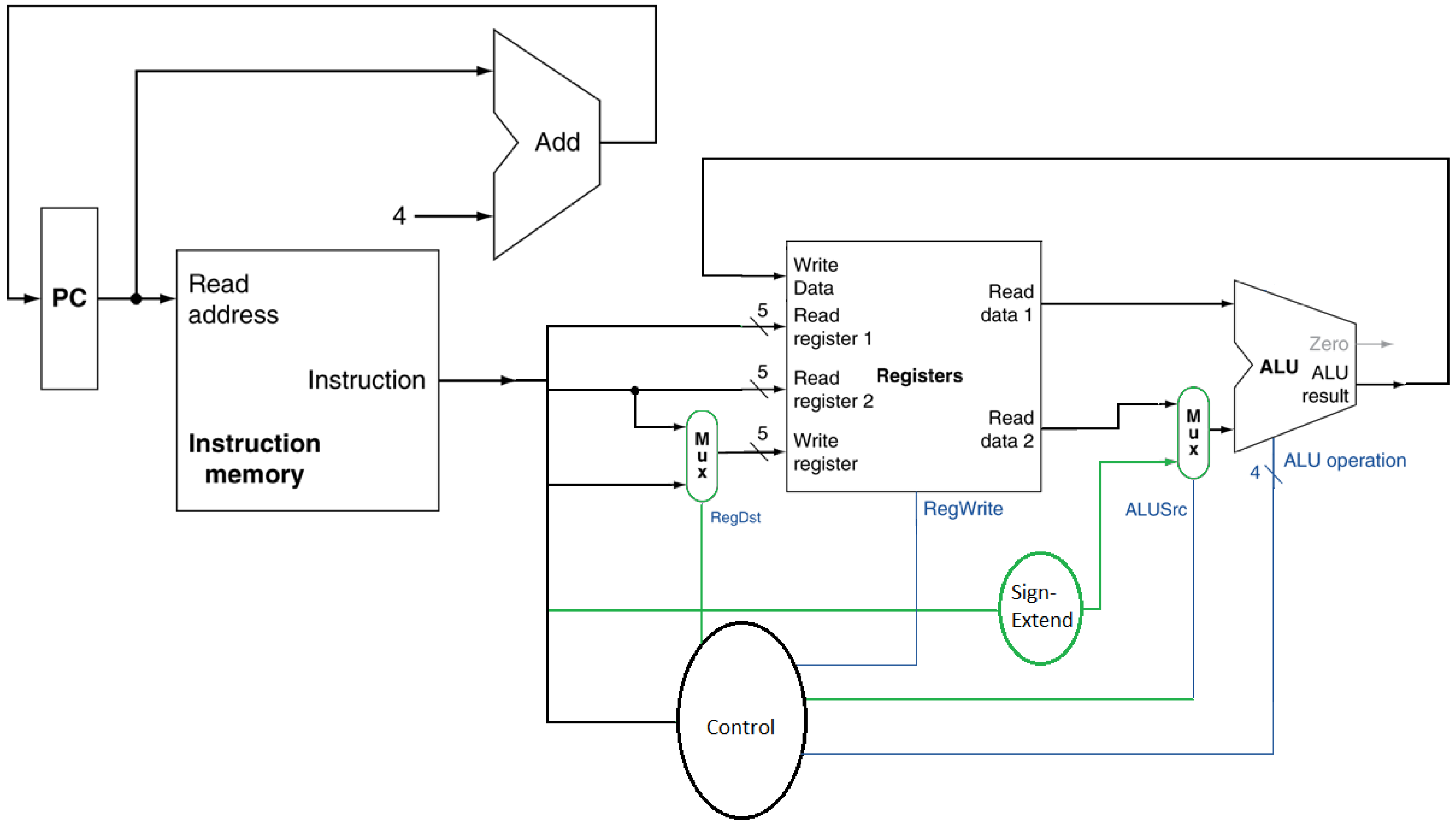
- Replace one read register, shamt, and funct with 16 bit constant

- ALU requires 32-bit inputs
  - Sign-extend the 16 bit immediate
    - Fill with 0s if the constant is positive
    - Fill with 1s if the constant is negative

- Choose between the second read registers and the sign extended constant.



Datapath: I-Type Instructions



Load/Store Instructions

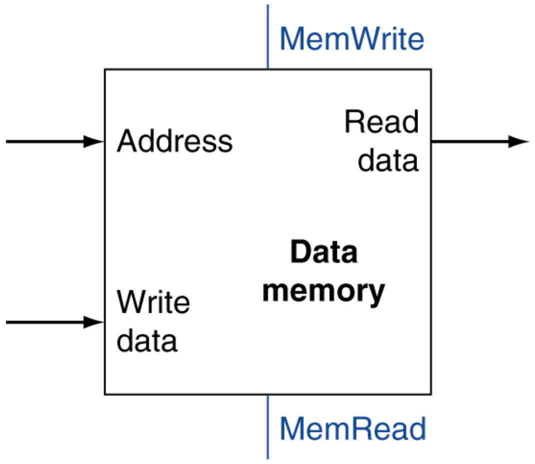
- lw $t1, offset($t2)
- sw $t1, offset($t2)

- Compute memory address: $t2 + offset
- Since offset is 16 bits, it needs to be extended to 32 bits

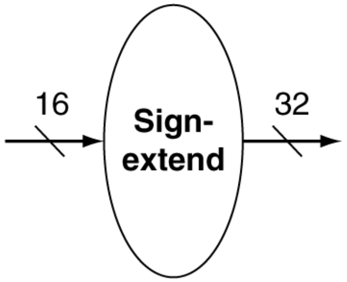
Load/Store Instructions

- Load: Read memory and update register
- Store: Write register value to memory

- Elements:
  - Register file
  - Sign extension unit
  - Data memory unit

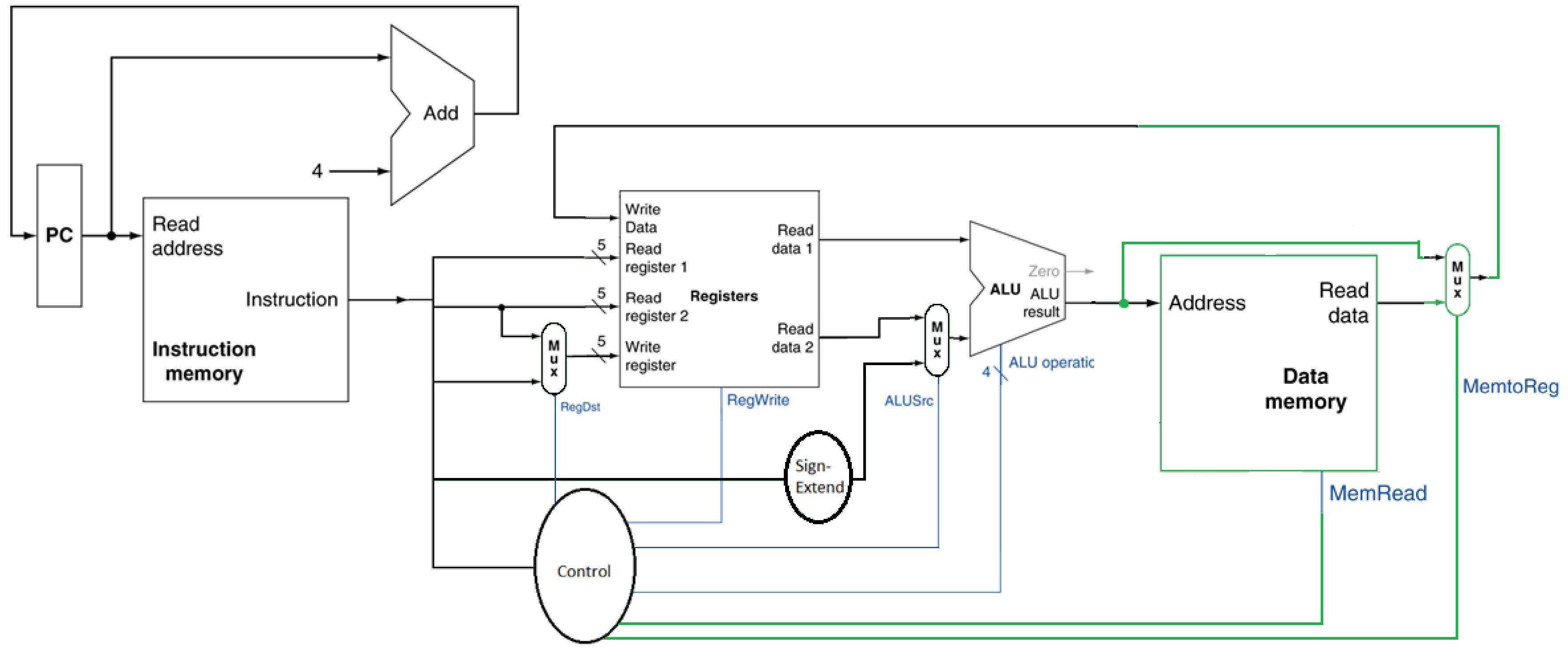


a. Data memory unit

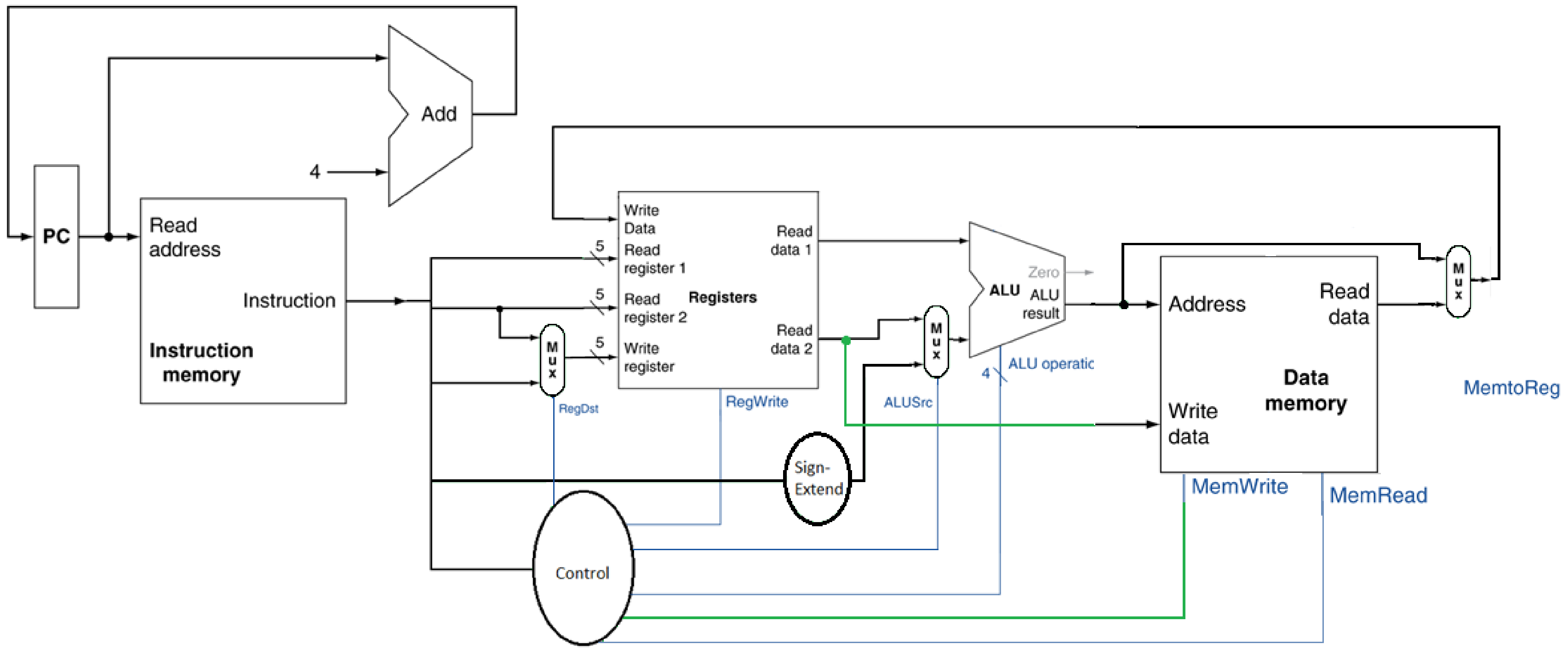


b. Sign extension unit

Datapath: Load Instruction



Datapath: Store Instruction



Branch Instructions

```
beq    $t1, $t2, offset
```

- 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

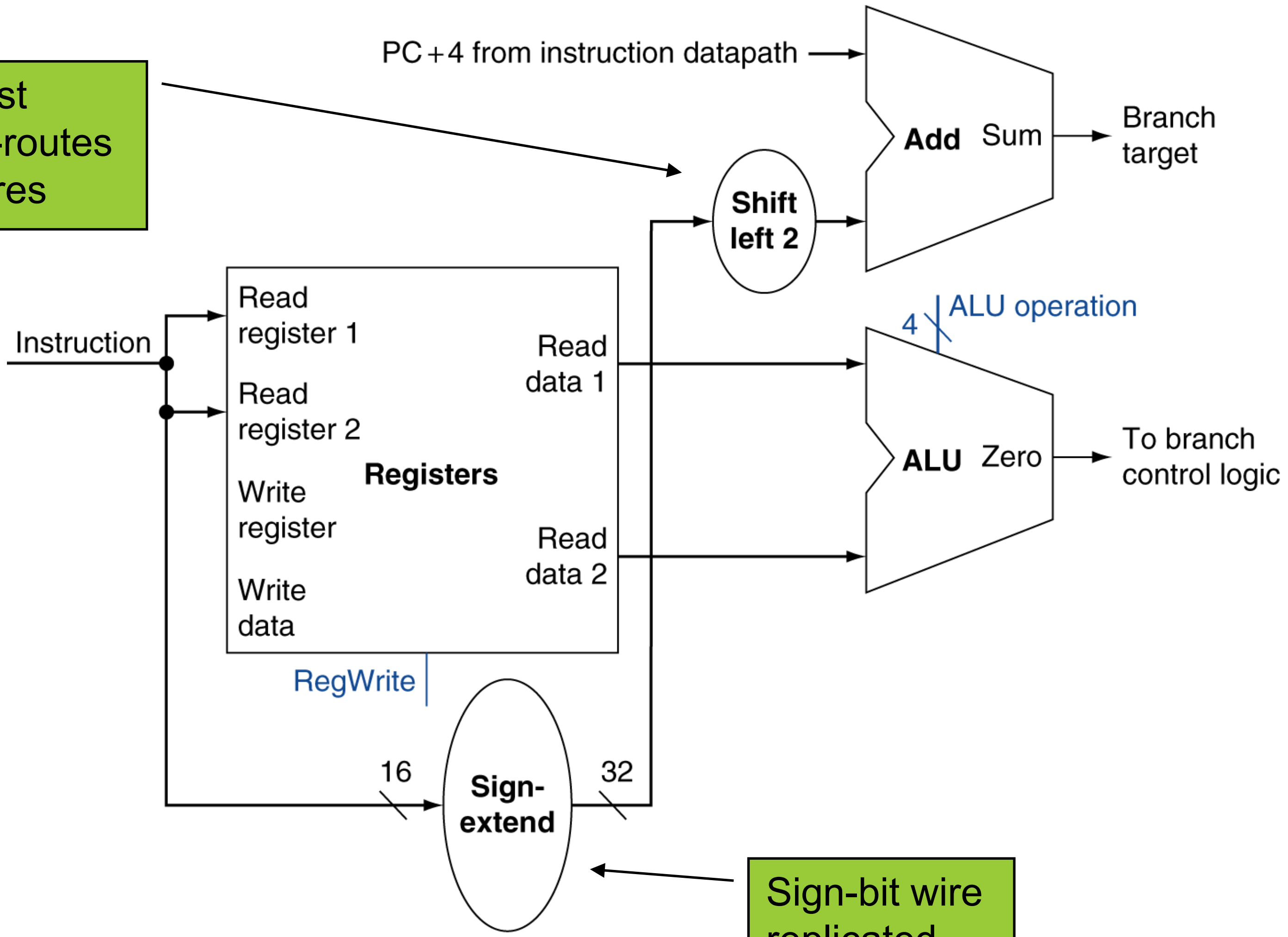
Target Address Details

- The base for the branch address calculation is the address of the instruction following the branch.
  - PC+4

- Memory is byte addressed
  - The offset field must be shifted left 2 bits

Branch Instructions

Just re-routes wires



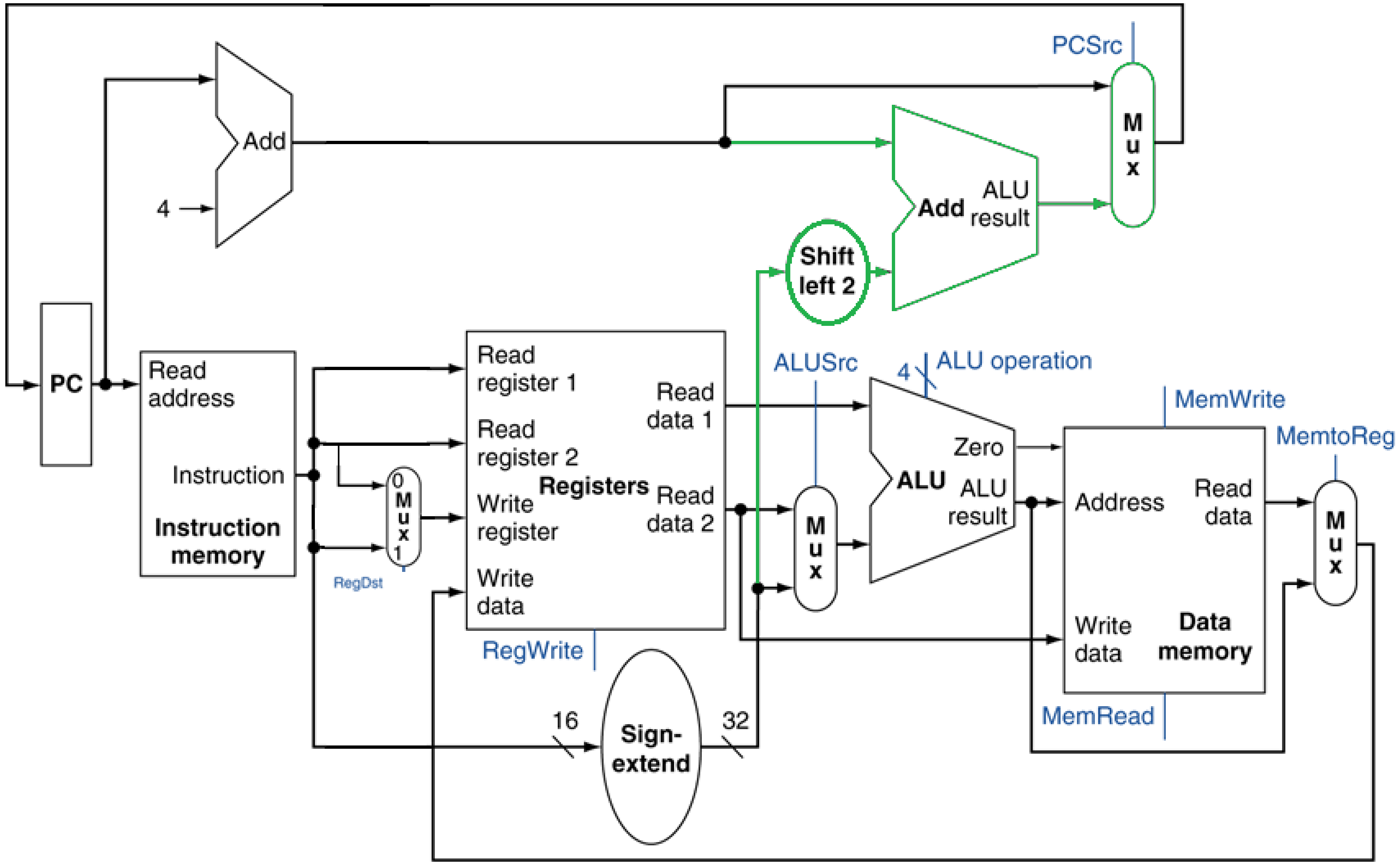
Sign-bit wire replicated

Branching

beq $t1, $t2, offset

- Branch is taken
  - When $t1 - $t2 = 0, the zero signal from the ALU
  - PC = PC + 4 + 4*offset
- Branch is not taken
  - PC = PC + 4

Datapath: Branch



Jump Instructions

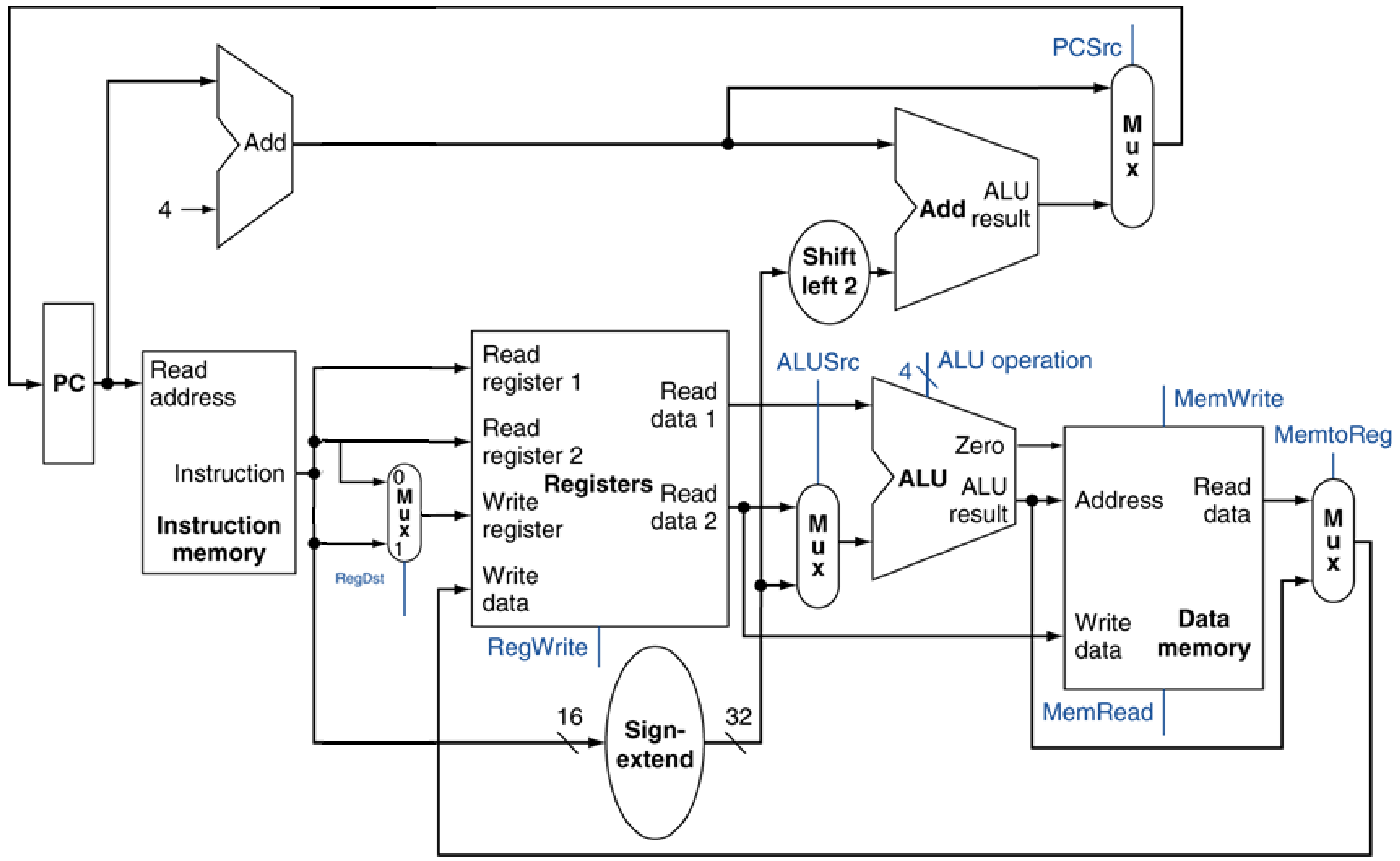
- PC = PC[31-28] : Offset << 2

Single Datapath

- All instructions executed in one clock cycle
  - Each datapath element can only do one function at a time
  - Any element needed more than once must be duplicated
  - Hence, we need separate instruction and data memories

- Use multiplexers where alternate data sources are used for different instructions

Datapath



PCSrc
Add
4
M u x
Add
ALU result
Shift left 2
PC
Read address
Instruction
Instruction memory
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
0 M u x 1
RegDst
RegWrite
ALUSrc
4
ALU operation
M u x
ALU
Zero
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
M u x
16
Sign-extend
32

Control Unit

- Take in input
- Generate signals for each state element
- Generate selection signals for each multiplexor
- Generate function signals for ALU
  - ALU Control

ALU Control

- MIPS subset: lw, sw, beq, add, addi, sub, and, or, slt

- ALU has 4 control inputs
  - 16 possible functions

ALU Control	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set-on-less-than
1100	NOR

ALU Control

- Load/Store: add
- Branch: subtract
- R-type: depends on funct field

ALU Control	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set-on-less-than
1100	NOR

ALU Control

- Opcode determines which type of instruction
- The ALUOp is a 2-bit signal derived from this Opcode
- ALUOp and the funct field will determine 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	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

- Multiple levels of decoding:
  - Main Control generates ALUOp
  - ALUOp and funct bits determine ALU Control

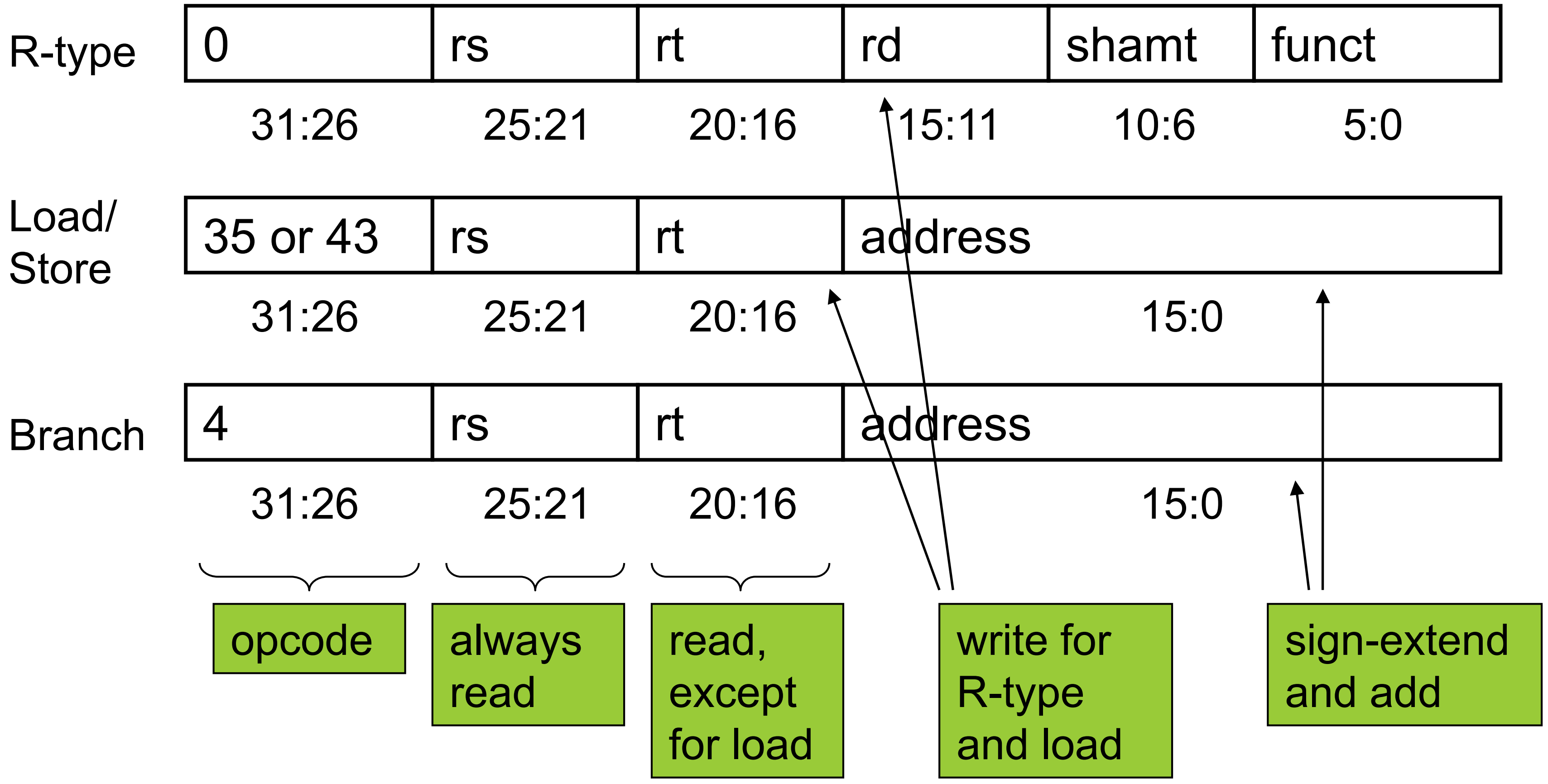
ALU Control

- ALUOp and funct bits determine ALU Control

ALUOp1	ALUOp2	Funct						Operation
0	0	X	X	X	X	X	X	0010 (add)
X	1	X	X	X	X	X	X	0110 (subtract)
1	X	X	X	0	0	0	0	0010 (add)
1	X	X	X	0	0	1	0	0110 (subtract)
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 Main Control Unit

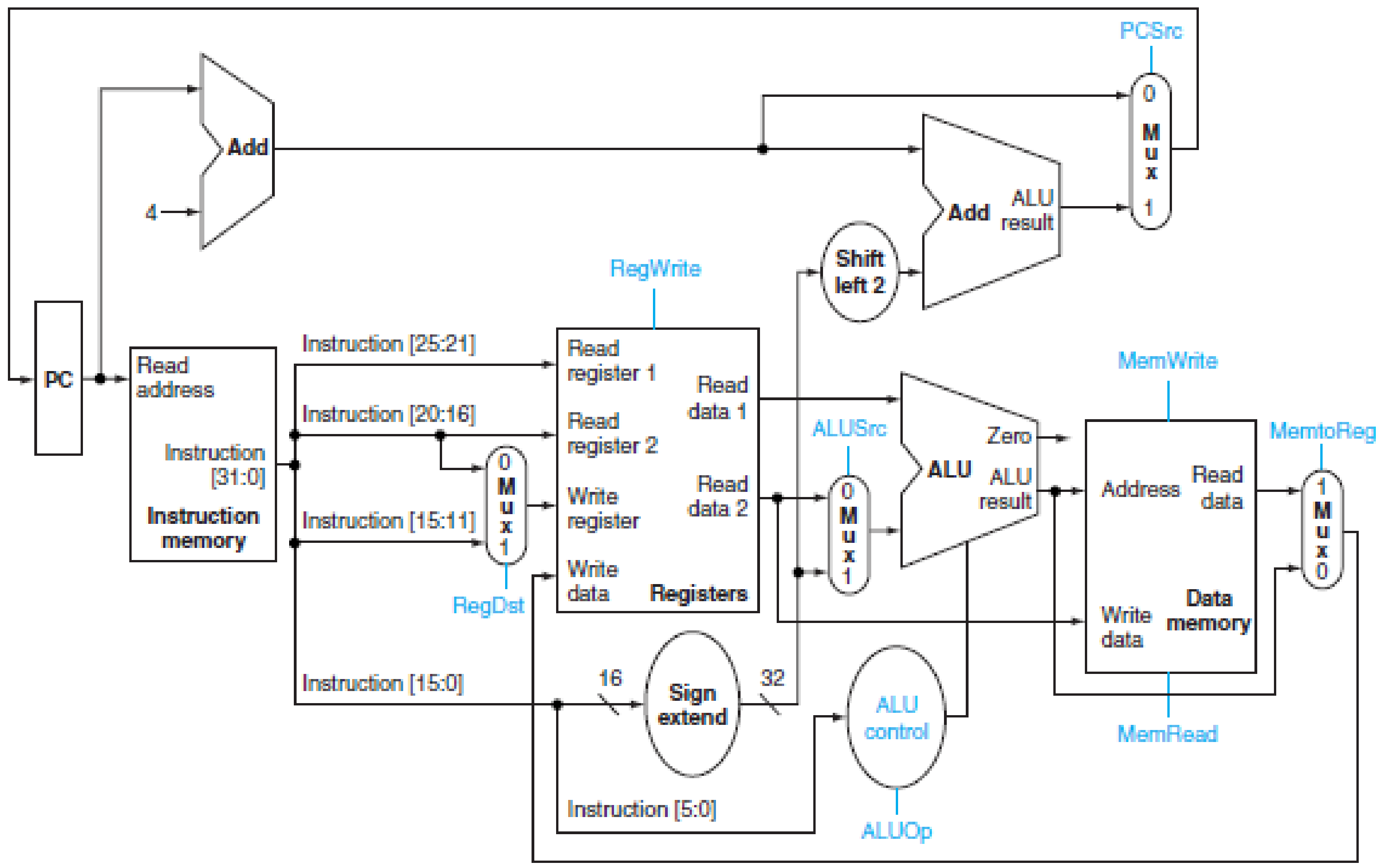
- Control signals derived from instruction



Main Control Unit

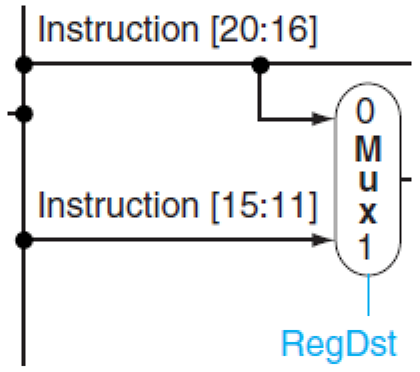
- Observations
  - The opcode is always contained in bits 31:26. We will refer to this field as Op[5:0].
  - The two registers to be read are always specified by the rs and rt fields, at positions 25:21 and 20:16. This is true for the R-type instructions, branch equal, and for store.
  - The base register for load and store instructions is always in bit positions 25:21 (rs).
  - The 16-bit offset for branch equal, load, and store is always in positions 15:0.
  - The destination register is in one of two places.
    - For a load it is in bit positions 20:16 (rt)
    - For an R-type instruction it is in bit positions 15:11 (rd)
      - We will need to add a multiplexor to select which field of the instruction is used to indicate the register number to be written.

Datapath with ALU Control



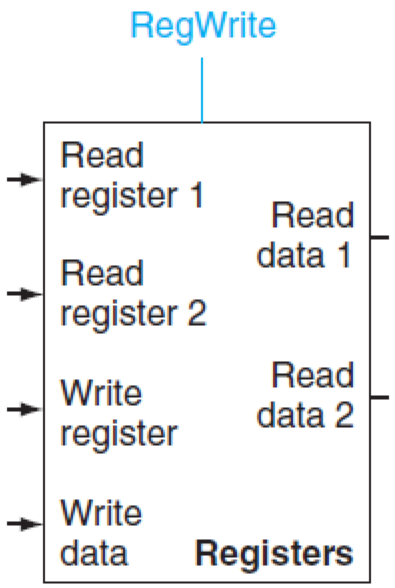
Control Signals – Register Destination

- RegDst
  - Deasserted: the register destination number for the write register comes from the rt field (bits 20:16)
  - Asserted: the register destination number for the write register comes from the rd field (bits 15:11)



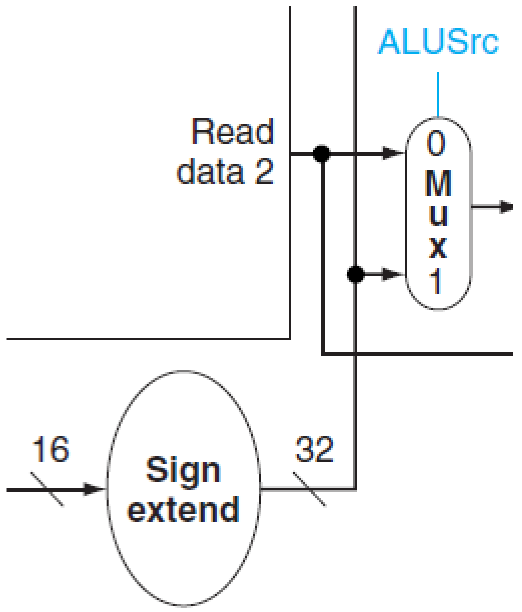
Control Signals – Register Write

- RegWrite
  - Asserted: the register destination specified by the write register input is written with the value from the write data input



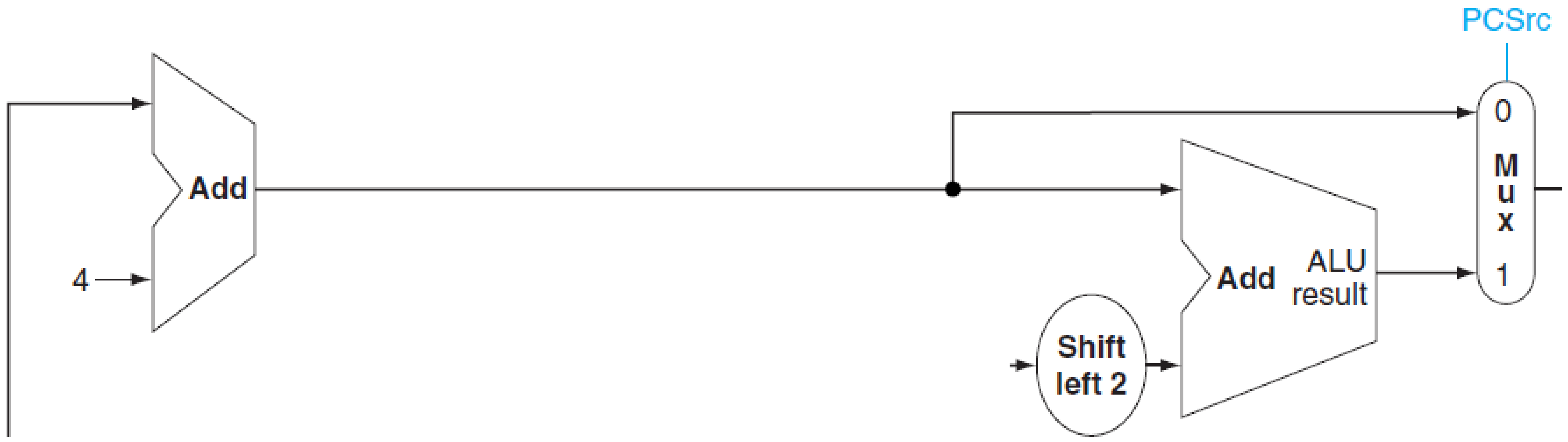
Control Signals – ALU Source

- ALUSrc
  - Deasserted: The second ALU operand comes from the second register file output
  - Asserted: The second ALU operand is the sign-extended lower 16 bits of the instruction



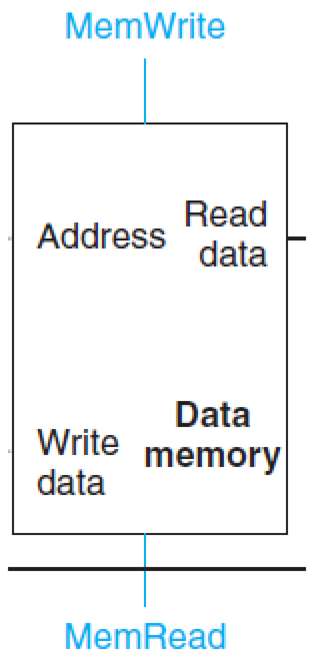
Control Signals - PCSrc

- Program Counter Source
  - Deasserted: The PC is replaced by PC + 4
  - Asserted: The PC is replaced by a branch target address



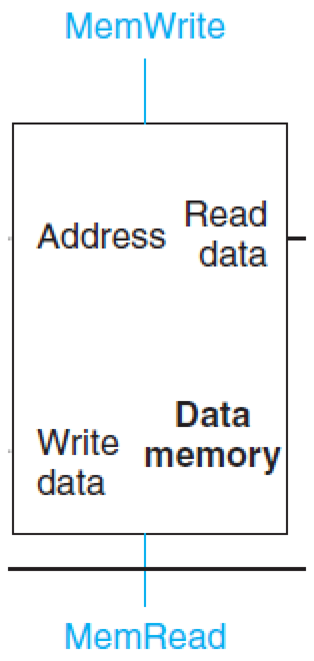
Control Signals – Memory Read

- MemRead
  - Asserted: Data memory contents designated by the address input are put on the read data output.



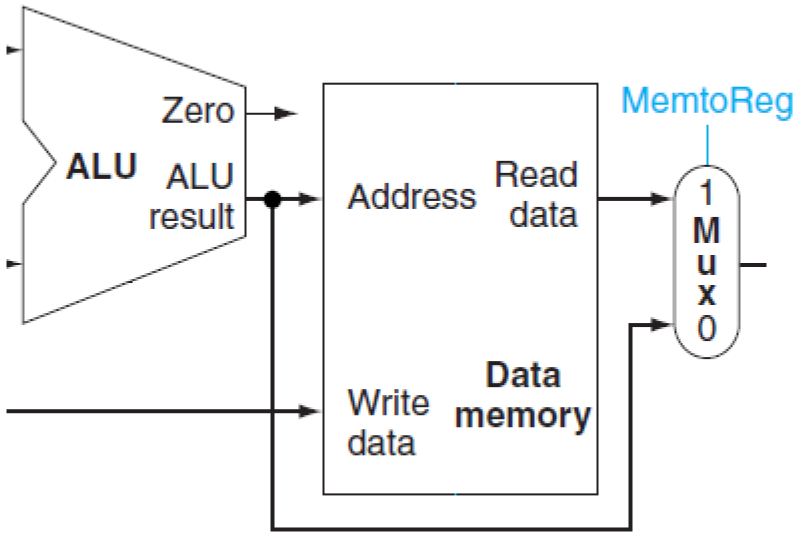
Control Signals – Memory Write

- MemWrite
  - Asserted: Data memory contents designated by the address input are replaced by the value on the write data input.



Control Signals – Memory to Register

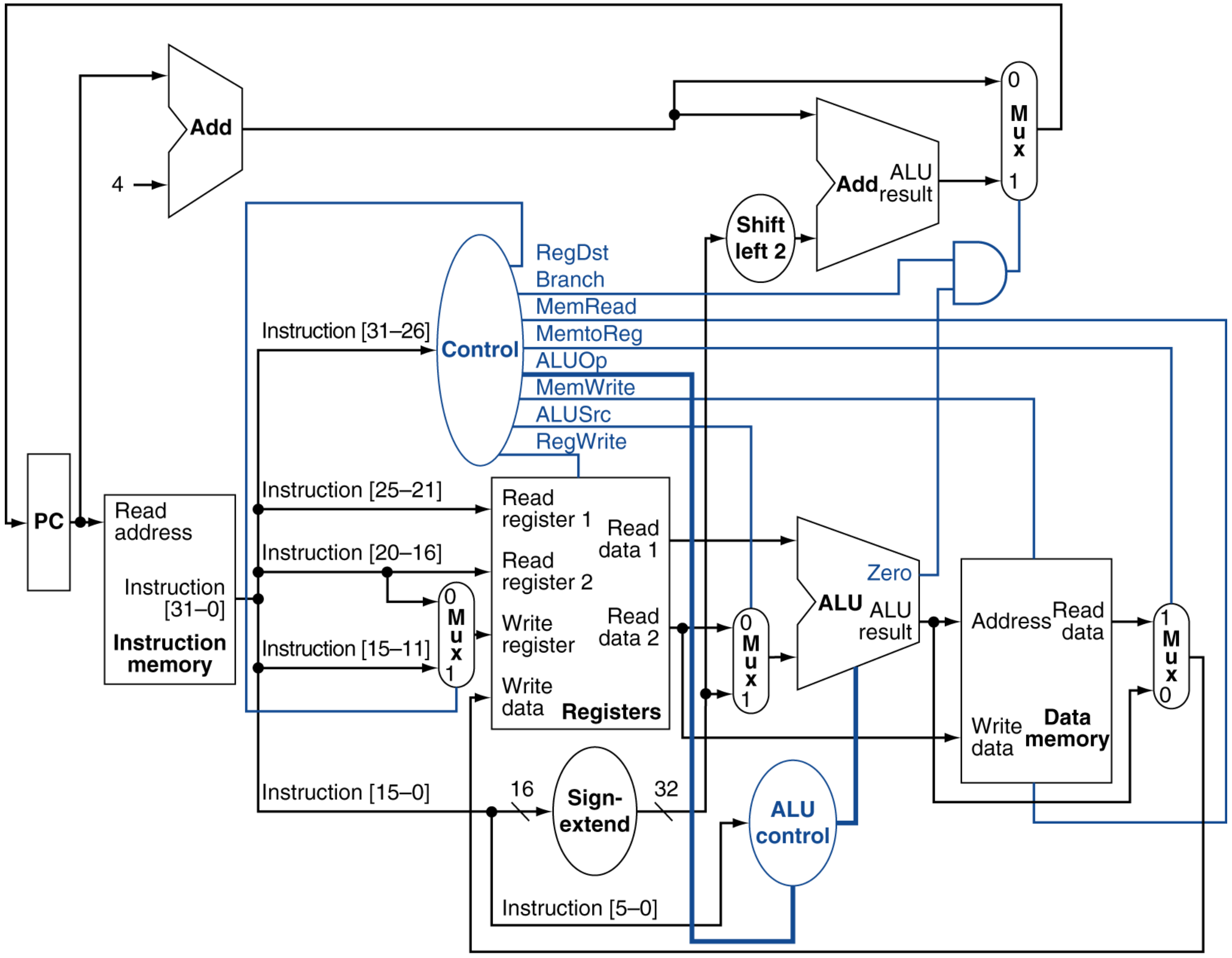
- MemtoReg
  - Deasserted: The value fed to the register write data input comes from the ALU
  - Asserted: The value fed to the register write data input comes from the data memory.



Control Signals

- All but one signal can be set based only on the opcode
- PCSrc is the exception
  - Relies on the result of a branch
  - PCSrc should be set if the instruction is beq and the zero output of the ALU is asserted

Datapath With Control

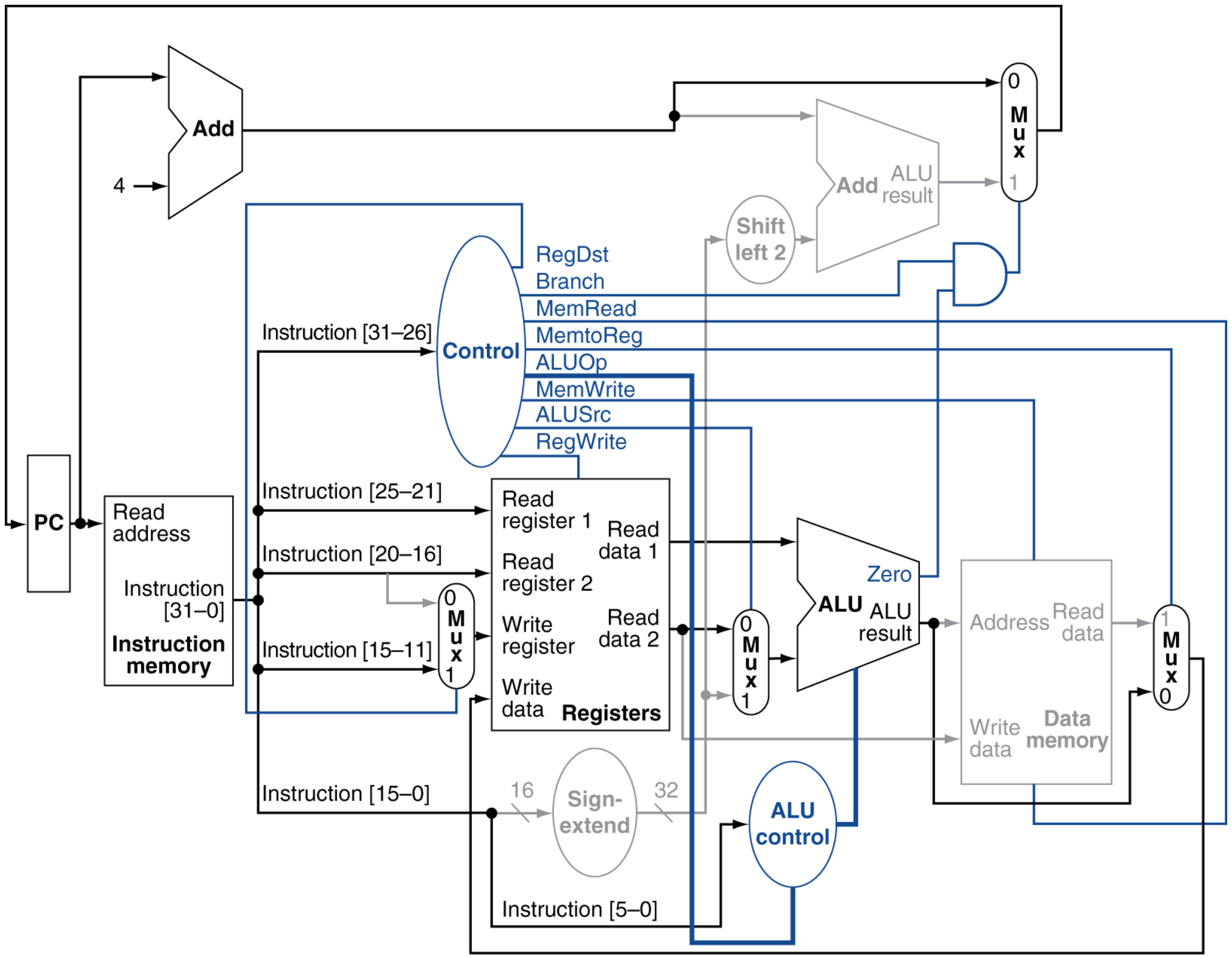


Opcode to Control

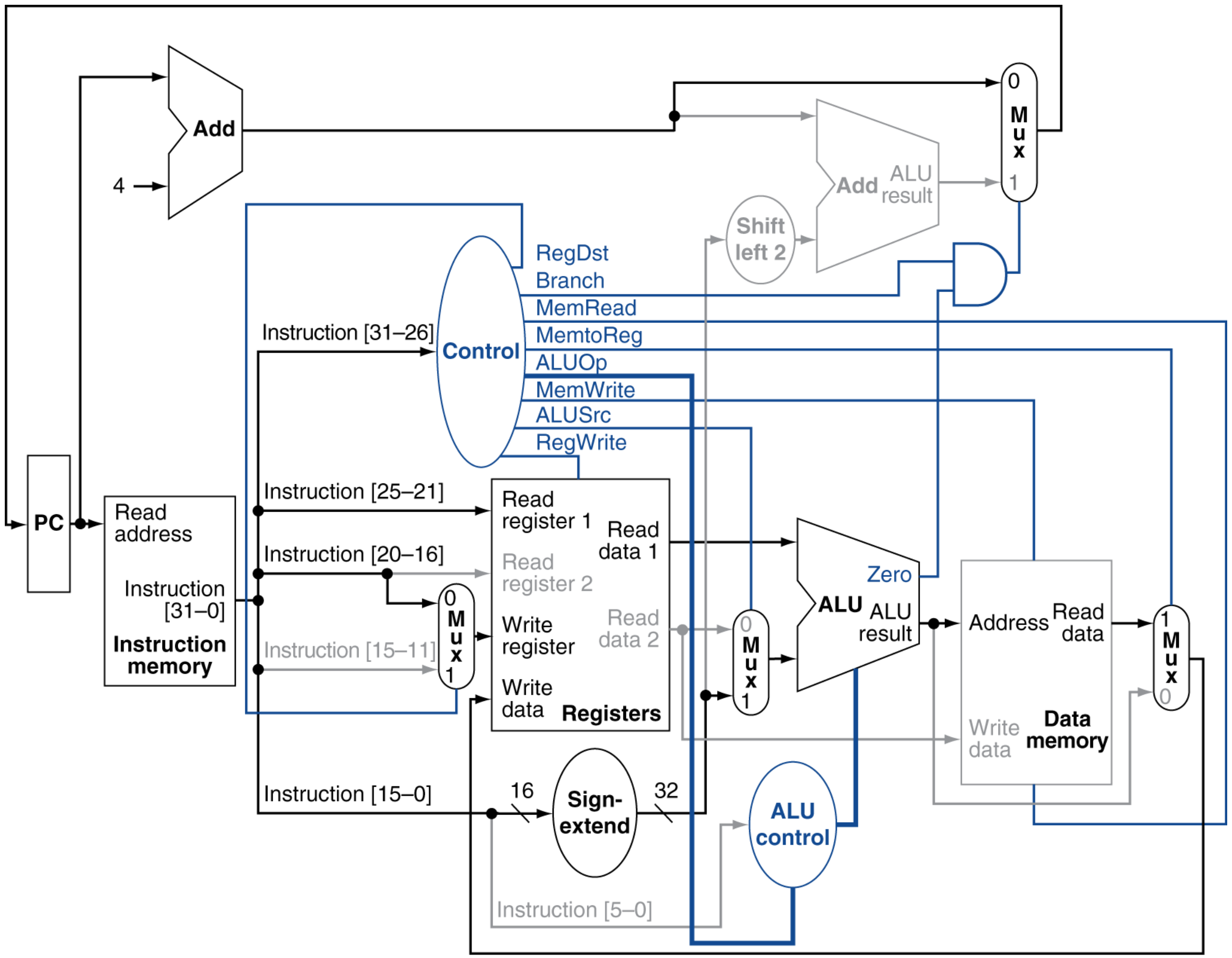
Instruction	RegDst	ALUSrc	MemtoReg	RegWrite
R-Format	1	0	0	1
Lw	0	1	1	1
Sw	X	1	X	0
Beq	X	0	X	0

Instruction	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-Format	0	0	0	1	0
Lw	1	0	0	0	0
Sw	0	1	0	0	0
Beq	0	0	1	0	1

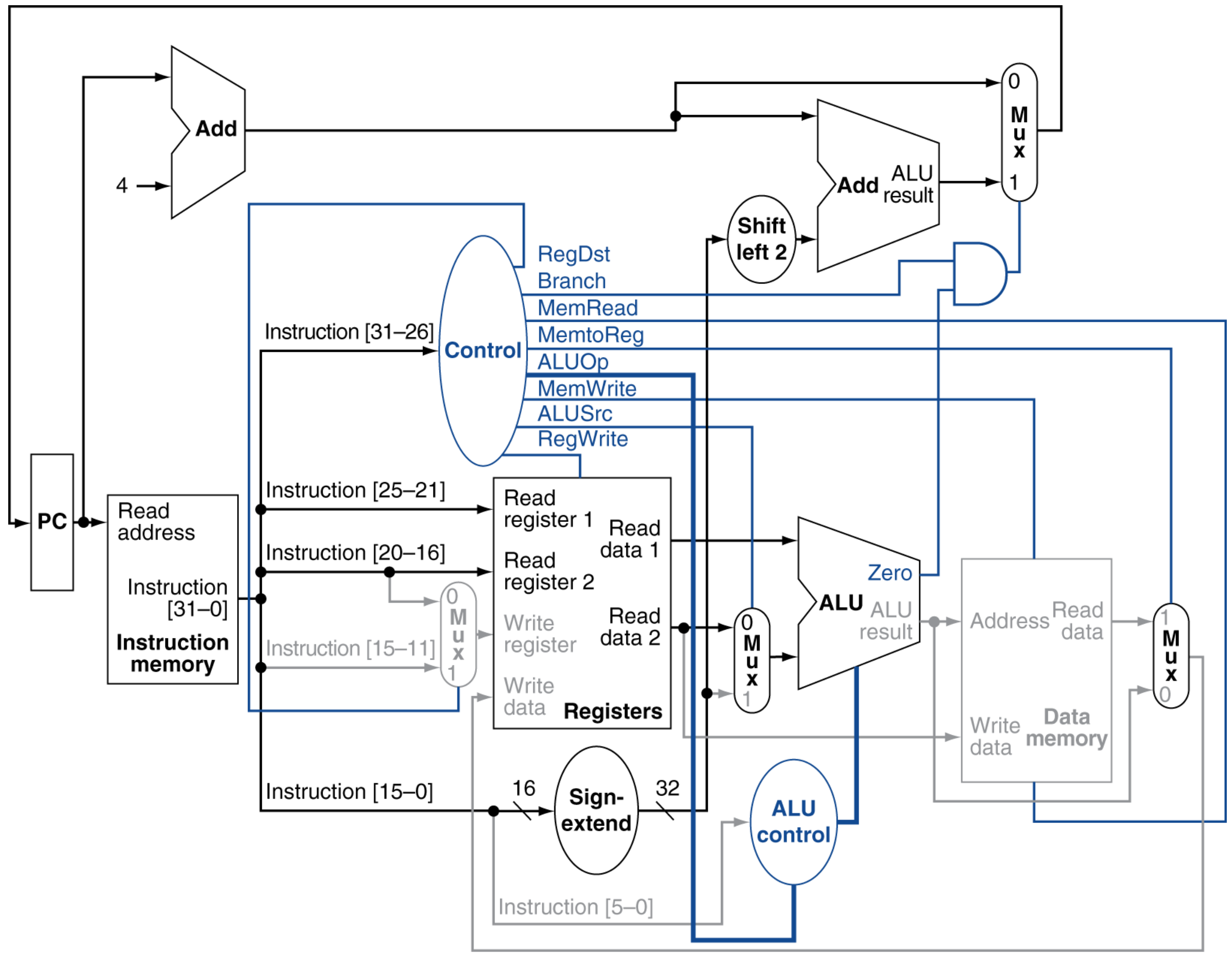
R-Type Instruction



Load Instruction



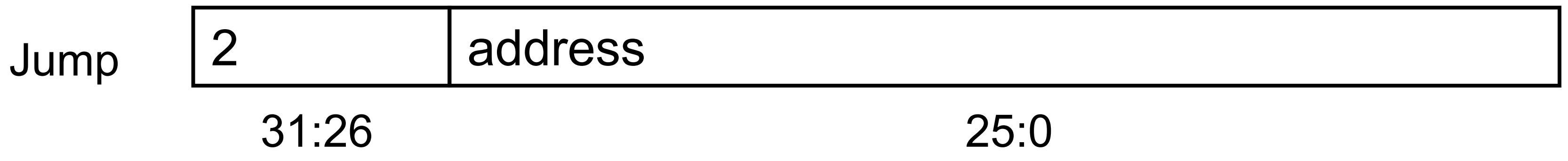
Branch-on-Equal Instruction



Opcode to Control

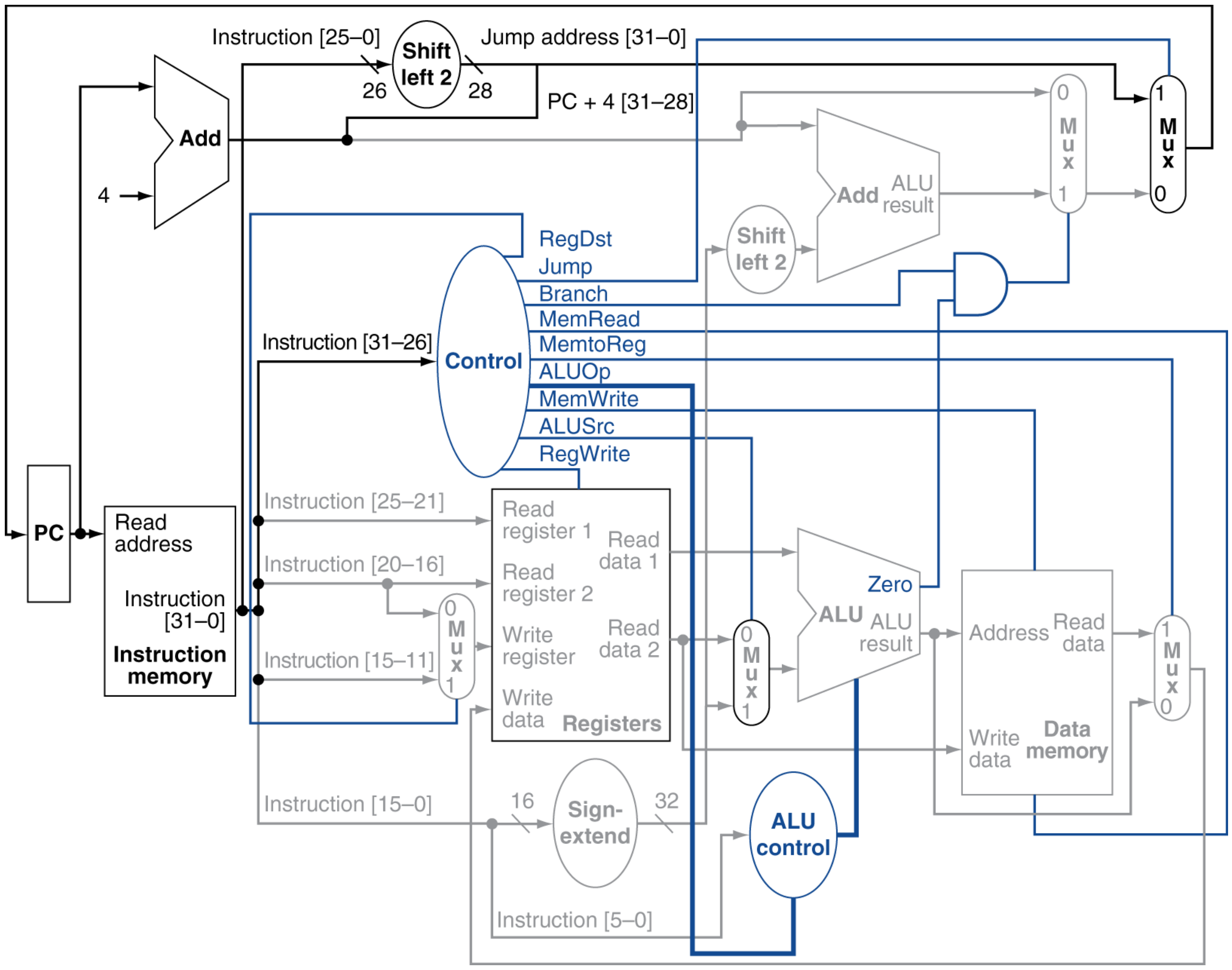
In/Out	Signal	R-Format	Lw	Sw	Beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemToReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Implementing Jumps



- Jump 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



Single-Cycle Implementation

- Every instruction begins execution on one clock edge and completes execution on the next clock edge.

- Clock cycle time must be at least as long as the longest instruction (load word).
    - Cycle time =

      PC’s propagation time +
      Instruction Memory Access Time +
      Register File Access Time +
      ALU Delay (address calculation) +
      Data Memory Access Time +
      Register File Setup Time +
      Clock Skew

Single-Cycle Implementation

- Every instruction begins execution on one clock edge and completes execution on the next clock edge.

- Clock cycle time must be at least as long as the longest instruction (load word).
  - The cycle time for load is much longer than any other instruction

Single-Cycle Implementation

- Single-Cycle implementation is not practical
  - Unable to implement more complex instructions
  - Cannot make improves that will speed up the system unless those improvements are to load
  - Violates the design principle of making the common case fast
  - Some functional units must be duplicated, increasing hardware cost

Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
  - Pipelining is nearly universal.

Analogy for Pipelining: Laundry

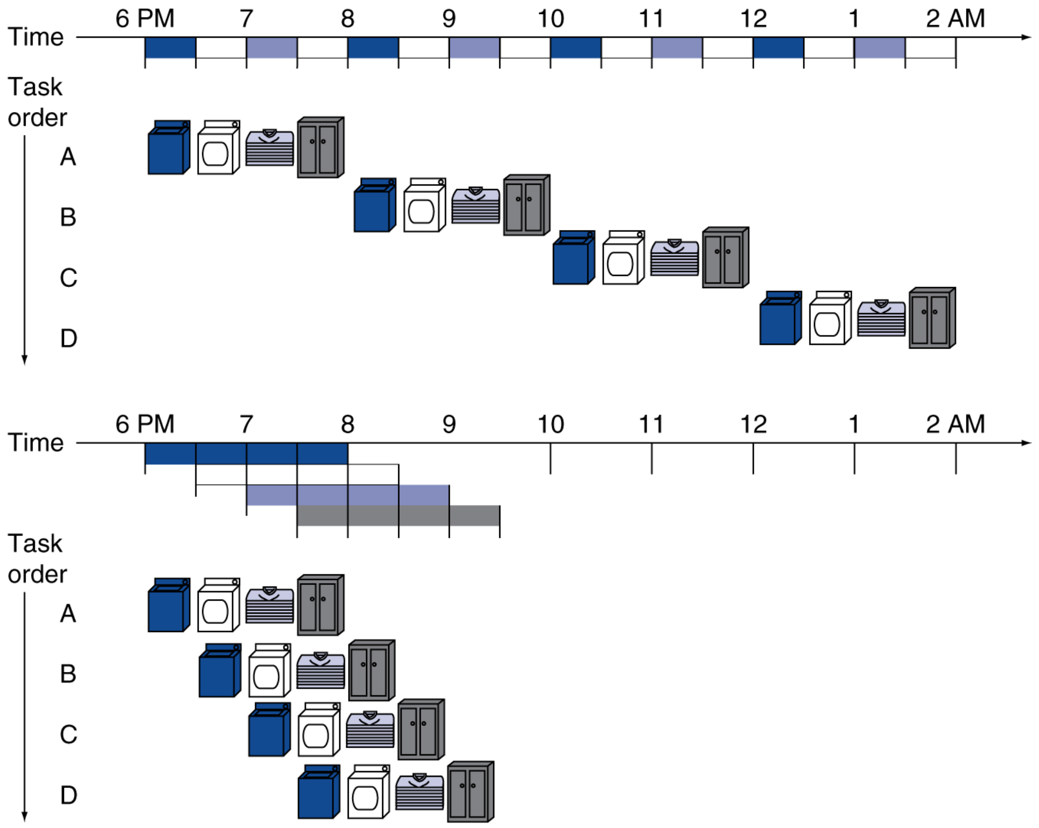
- Nonpipelined approach:
  1. Place one dirty load of clothes in the washer.
  2. When the washer is finished, place the wet load in the dryer.
  3. When the dryer is finished, place the dry load on a table and fold.
  4. When folding is finished, put the clothes away.

- When the clothes are put away, the next load can begin.

Analogy for Pipelining: Laundry

- Pipelined approach:
  1. Place one dirty load (load A) of clothes in the washer.
  2. When the washer is finished, place load A in the dryer and start a new load (load B) in the washer.
  3. When the dryer is finished, place load A on the table to fold, place load B in the dryer, and start a new load (load C) in the washer.
  4. When folding is finished, put away load A, fold load B, place load C in the dryer, start load D in the washer.

Analogy for Pipelining: Laundry



Pipelining Paradox

- One load of laundry still takes the same amount of time.
- The amount of time it takes to do many loads of laundry is shorter with pipelining.

- Pipelining improves throughput.

Pipelining Speedup

- If all the stages take about the same amount of time and there is enough work to do, then the speedup due to pipelining is roughly equal to the number of stages in the pipeline.

- Laundry Analogy:
- 4 stages (washing, drying, folding, putting away)
  - 20 loads pipelined would take about 5 times as long as 1 load
  - 20 loads of sequential laundry takes 20 times as long as 1 load

MIPS Pipeline

1. Fetch instruction from memory.	• Instruction Fetch	(IF)
1. Read registers while decoding the instruction.	• Instruction Decode	(ID)
1. Execute the operation or calculate an address.	• Execution	(EX)
1. Access an operand in data memory.	• Memory Access	(MEM)
1. Write the result into a register.	• Write Back	(WB)

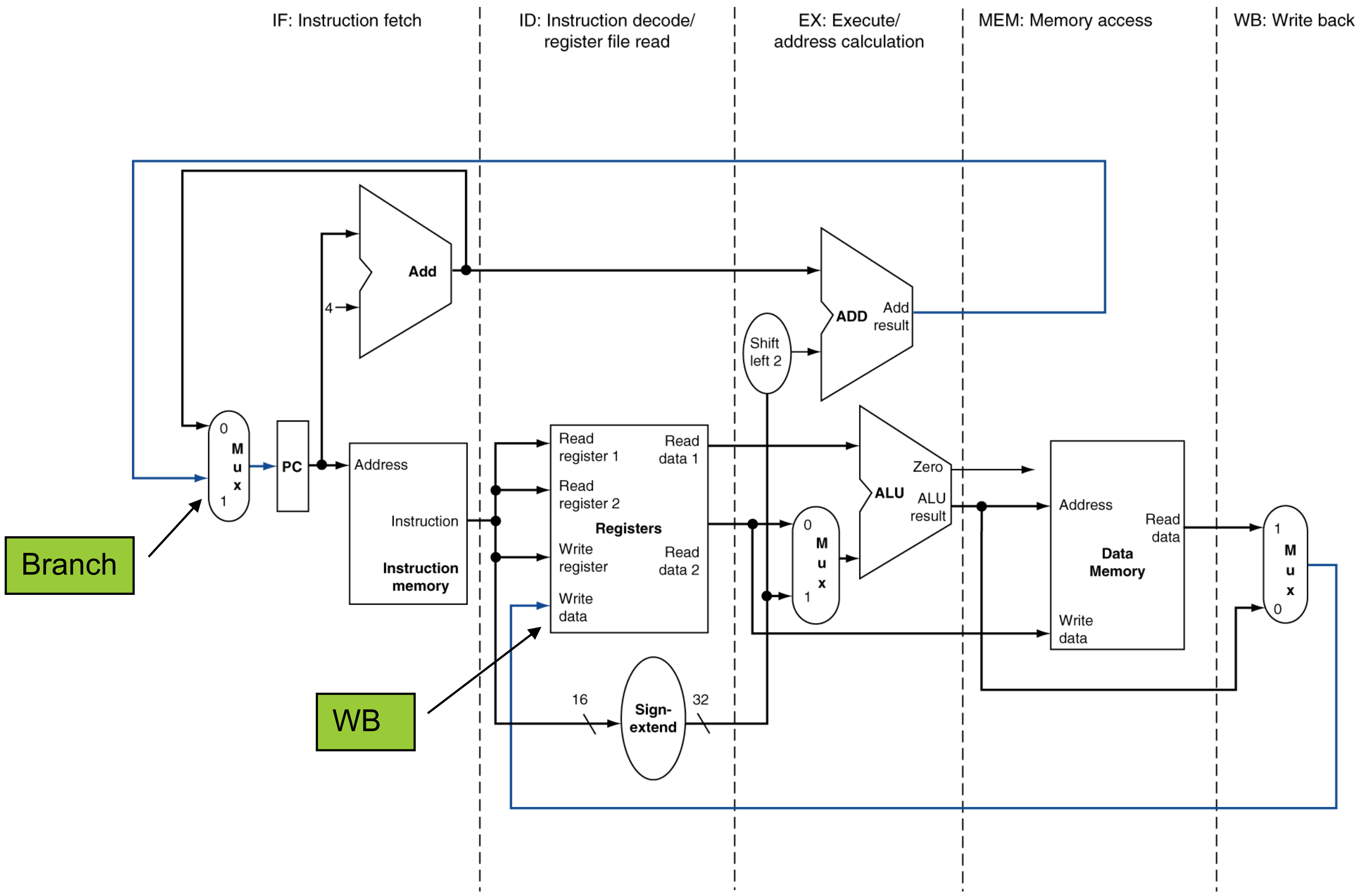
Pipeline Speedup

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

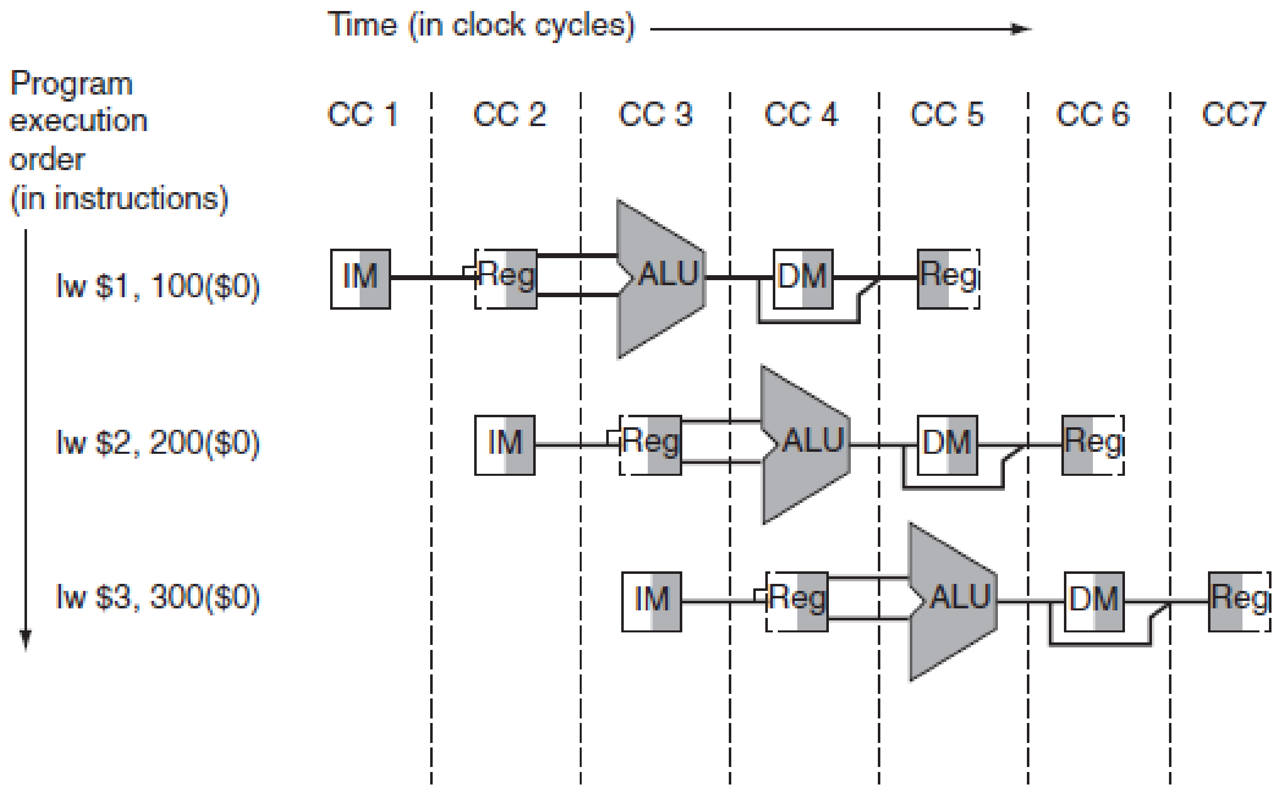
Pipelining and ISA Design

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

MIPS Pipelined Datapath

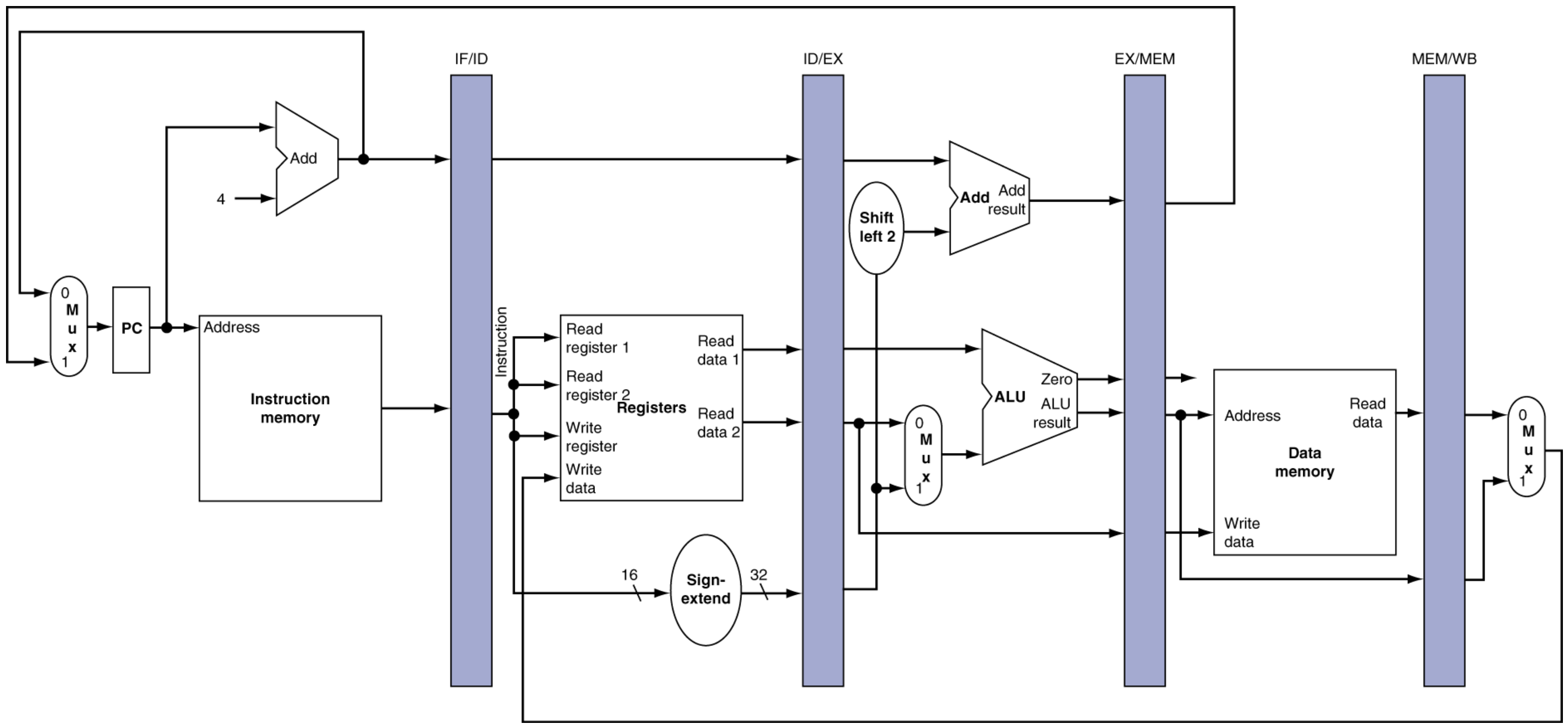


Pipelined Execution

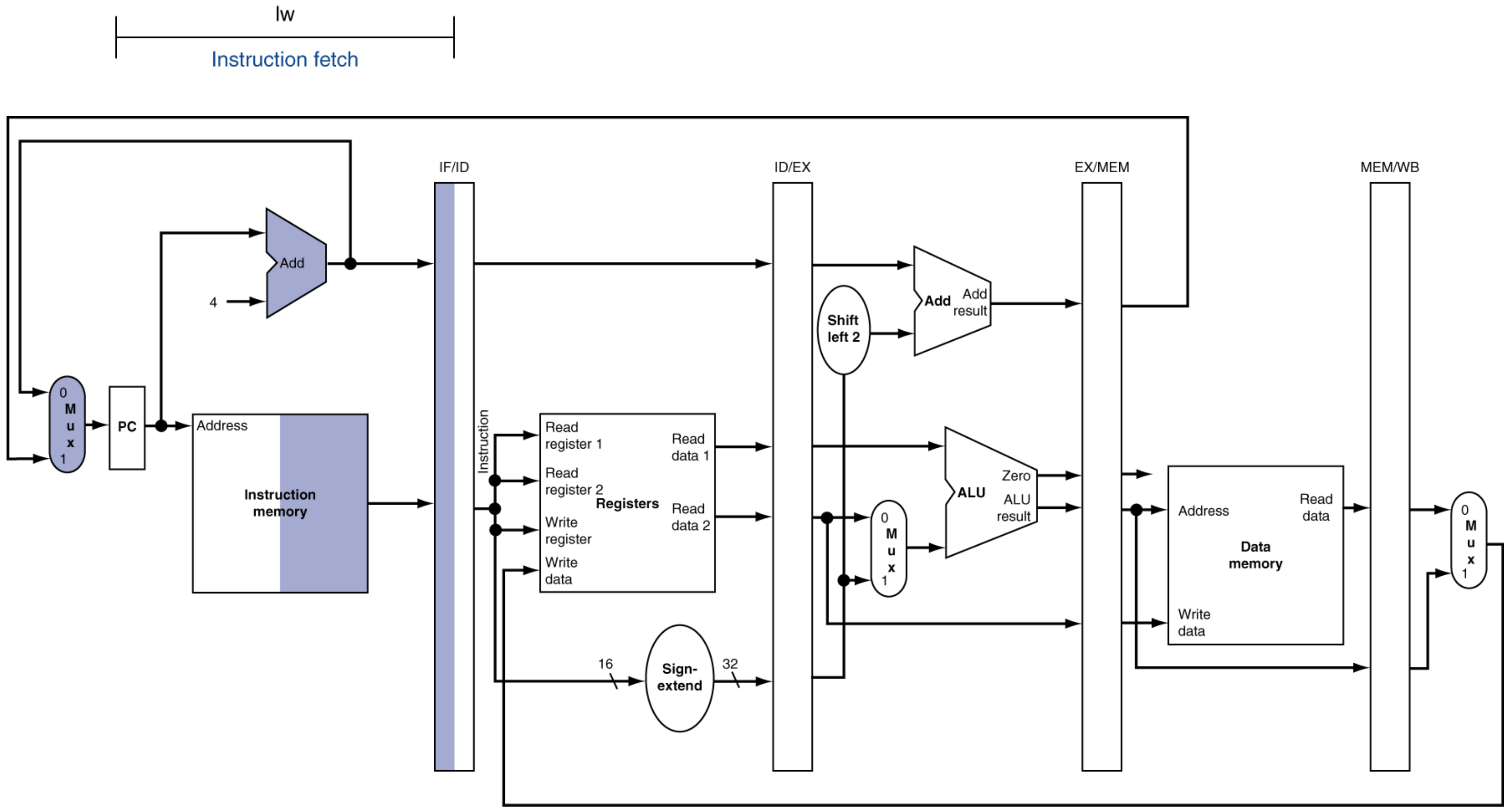


Pipeline registers

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

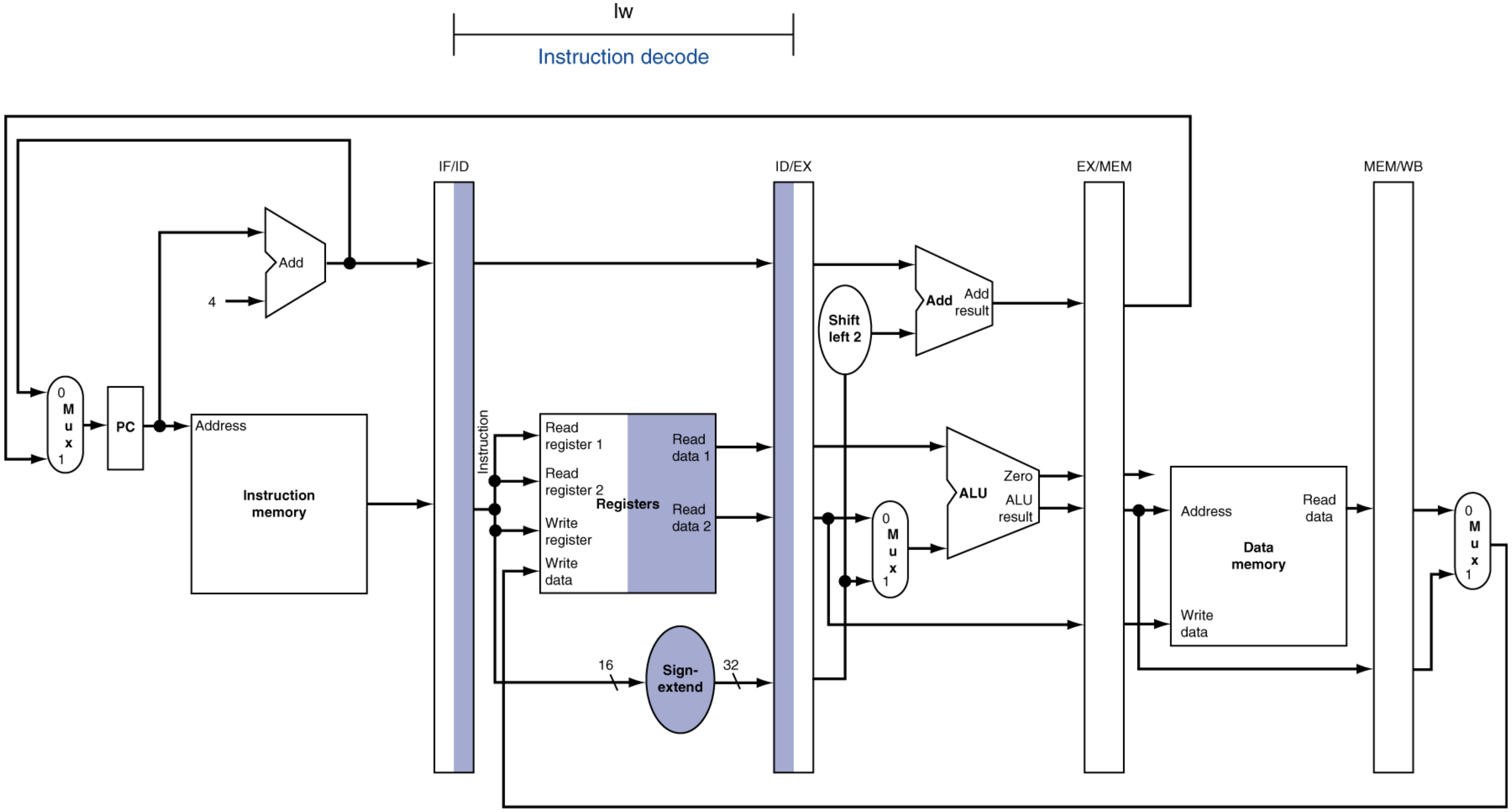


IF for Load

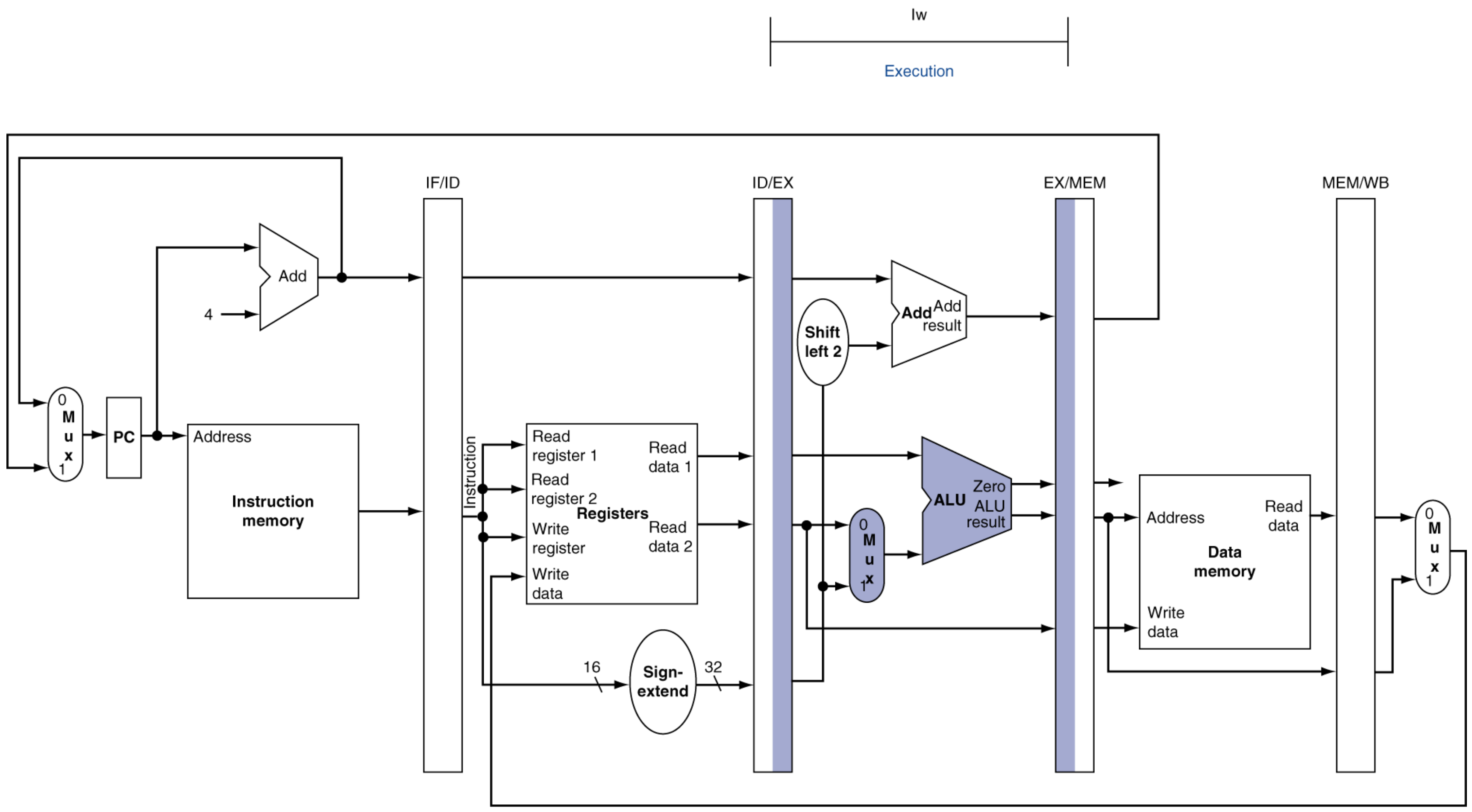


lw
Instruction fetch
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Write data
Read data

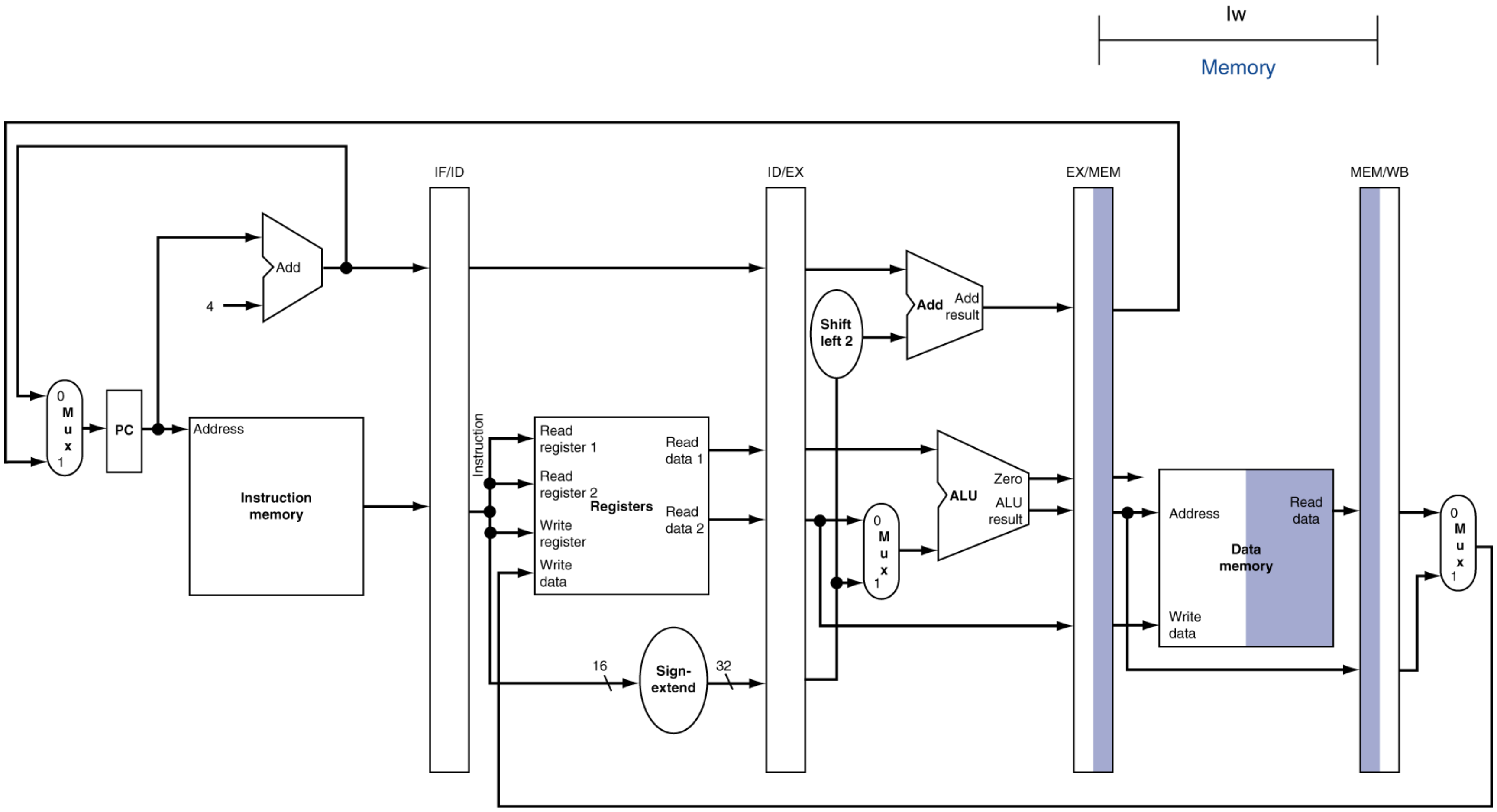
ID for Load


lw
Instruction decode
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
0 M u x 1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
0 M u x 1

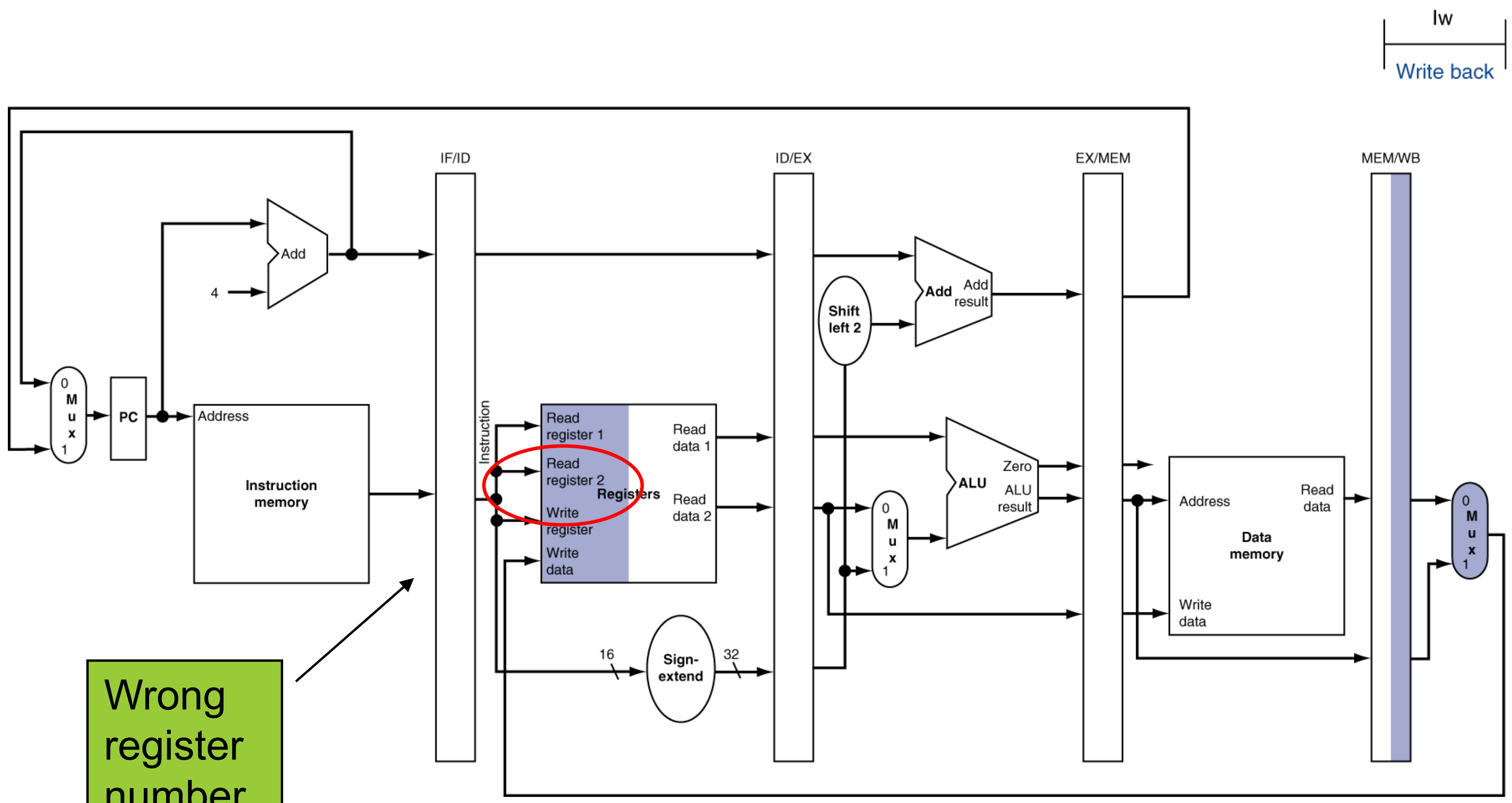
EX for Load


lw
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
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
0
M
u
x
1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
0
M
u
x
1

MEM for Load

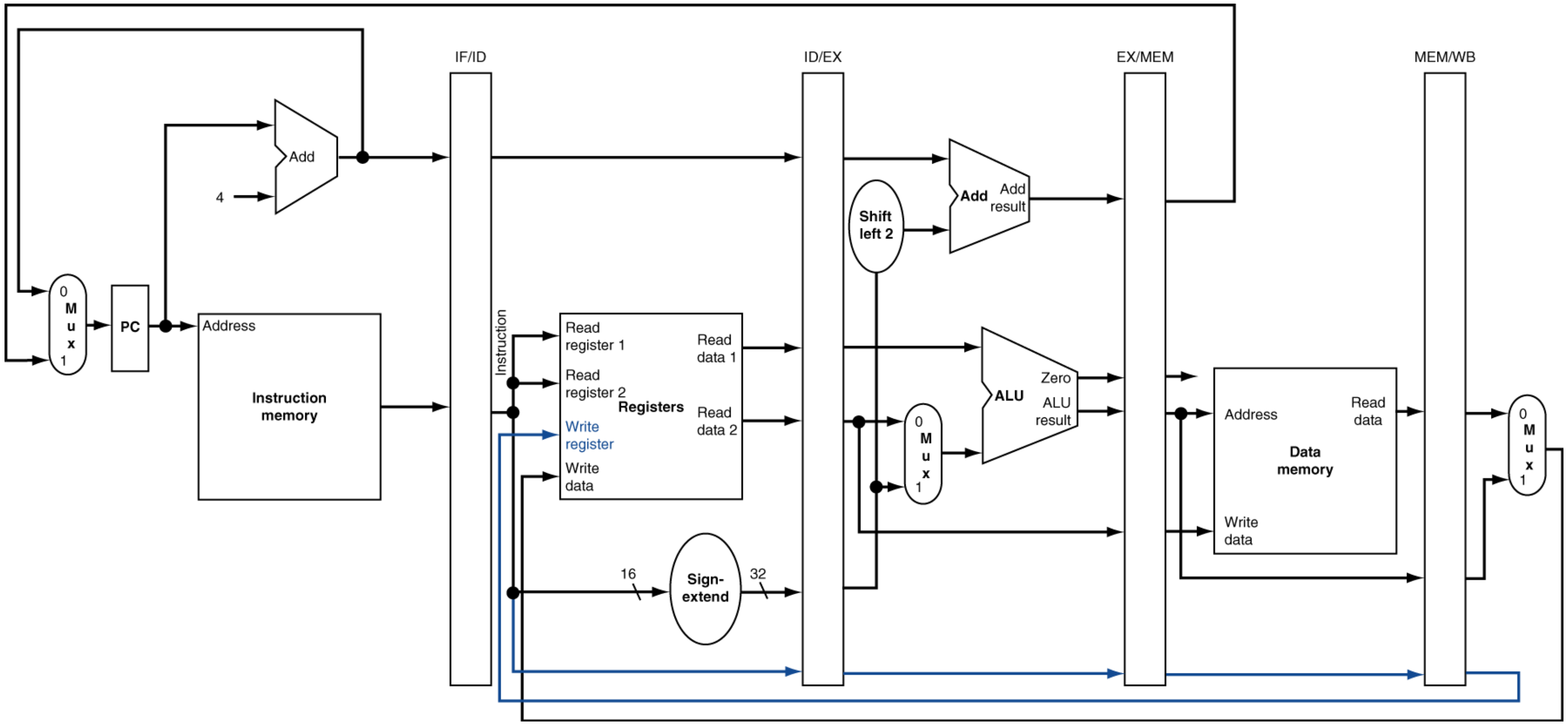

lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
PC
0 M u x 1
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
0 M u x 1
Address
Read data
Data memory
Write data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

WB for Load

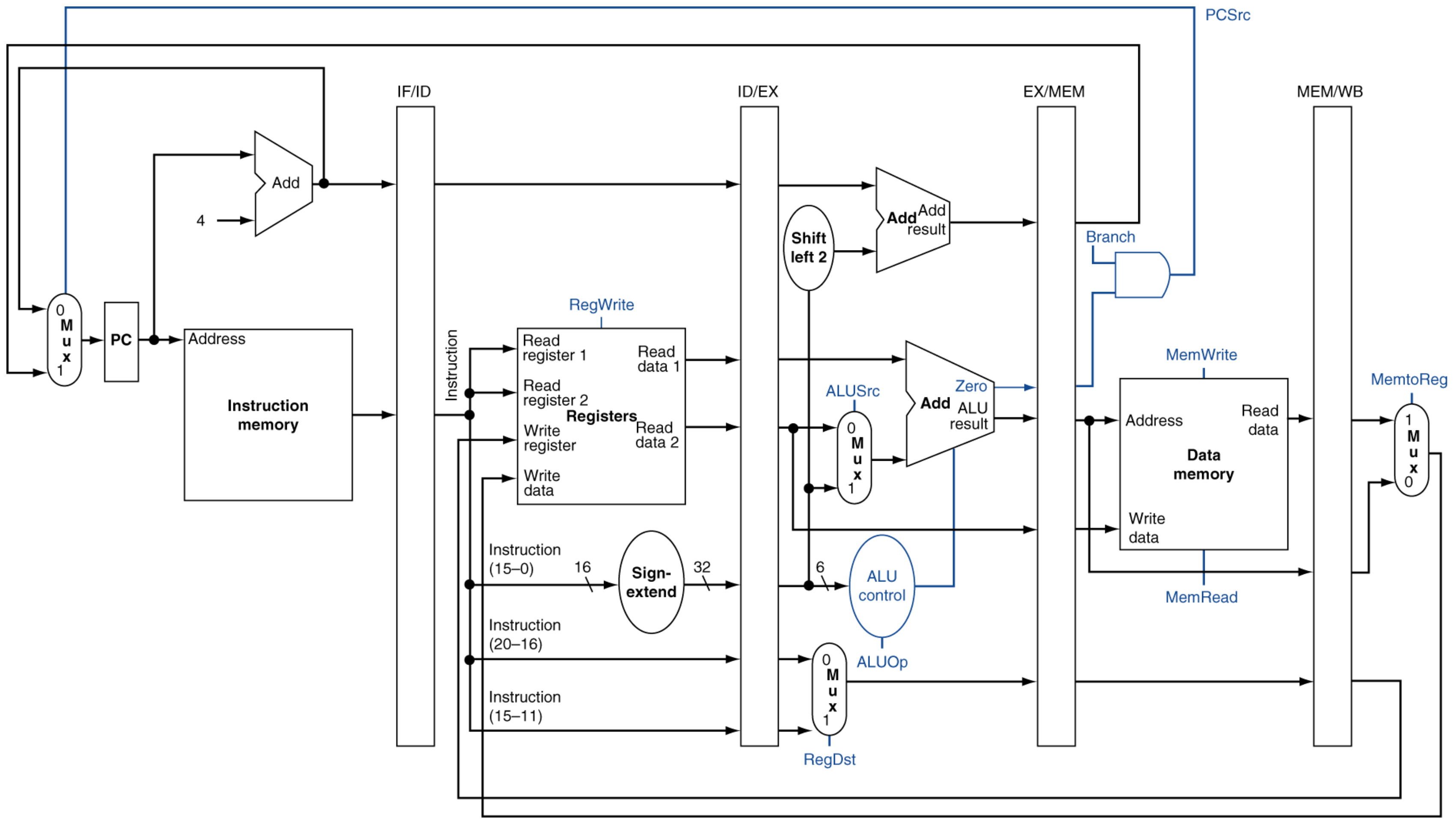


Wrong register number

Corrected Datapath for Load



Pipelined Control



Pipelined Control

- Instruction fetch:
  - PC written on every clock cycle
  - Instruction memory read on every clock cycle
  - no optional control lines
- Instruction decode/register file read
  - Register file read on every clock cycle
  - no optional control lines
- Execution:

Set RegDst:	select the Result register
Set ALUOp:	select the ALU operation
Set ALUSrc.	select either Read data 2 or a sign-extended immediate for the ALU.

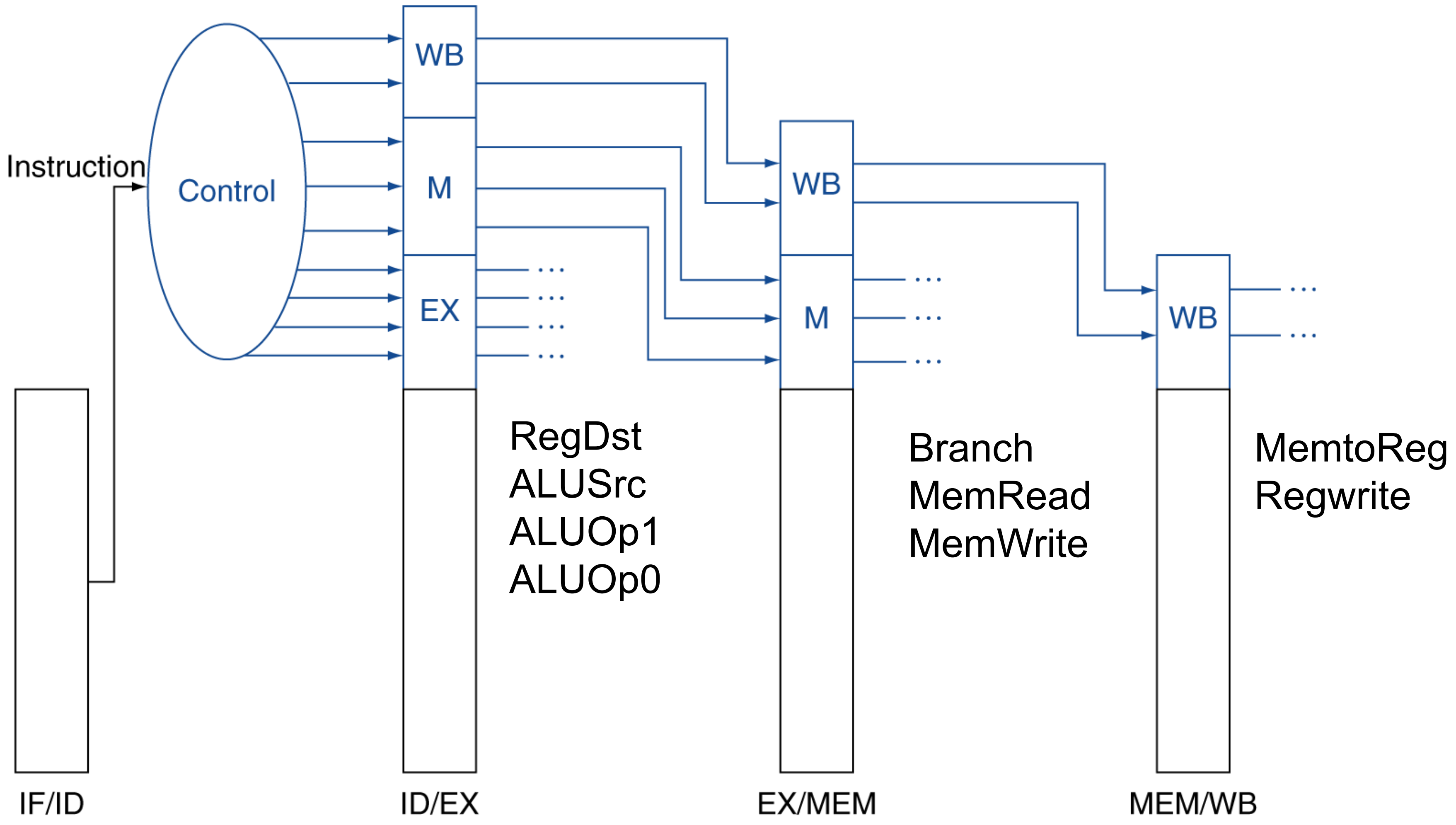
- Memory access:

Set Branch:	affects PCSrc
Set MemRead	
Set MemWrite.	

- Write back:

Set MemtoReg:	decides between sending the ALU result or the memory value to the register file
Set Reg-Write:	specifies whether the register file can be written

Pipelined Control

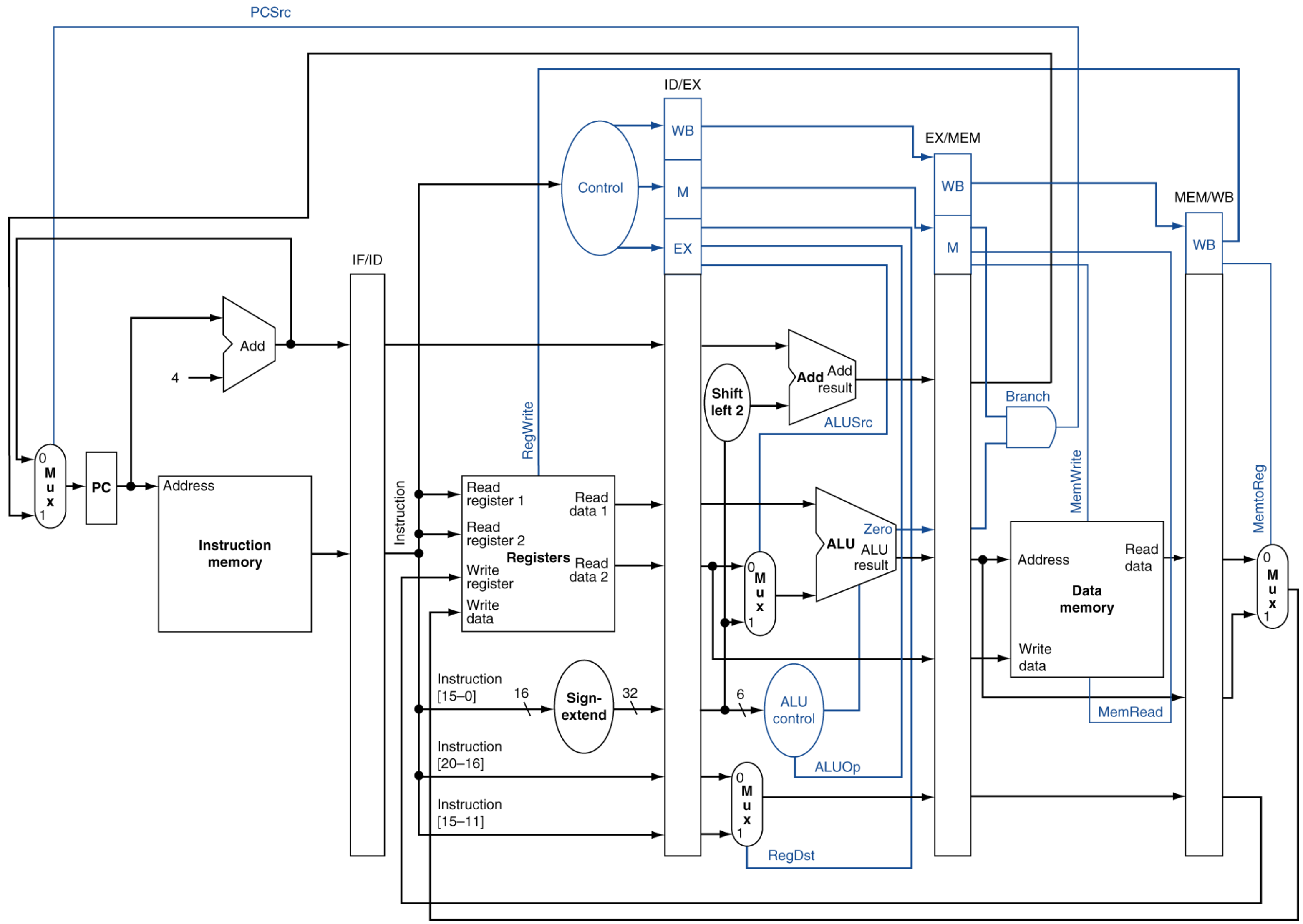


RegDst
ALUSrc
ALUOp1
ALUOp0

Branch
MemRead
MemWrite

MemtoReg
Regwrite

Pipelined Control



Hazards

- Situations that prevent starting the next instruction in the next cycle

- Structure hazard
  - 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

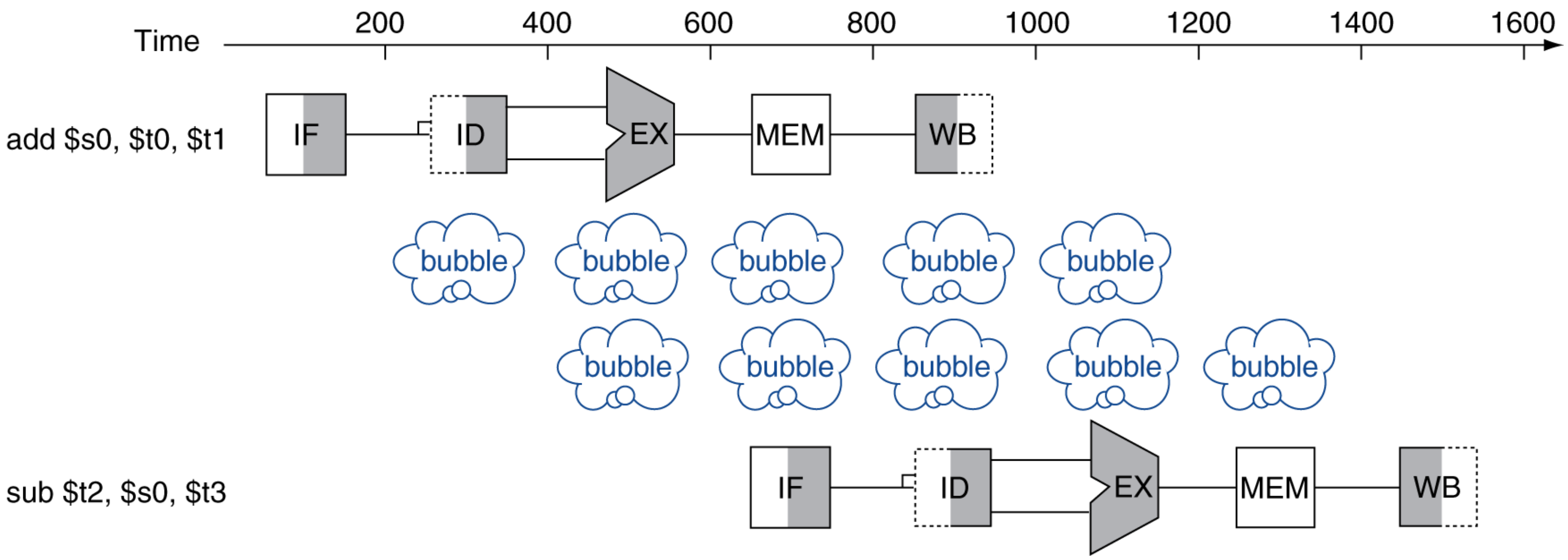
- Conflict for use of a resource

- 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 separate instruction/data caches

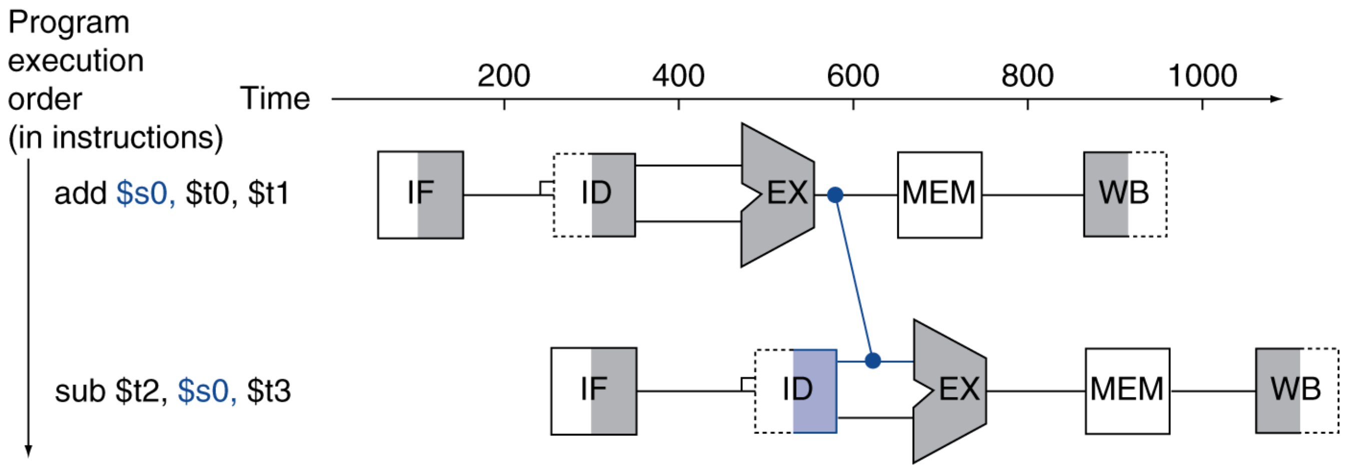
Data Hazards

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



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

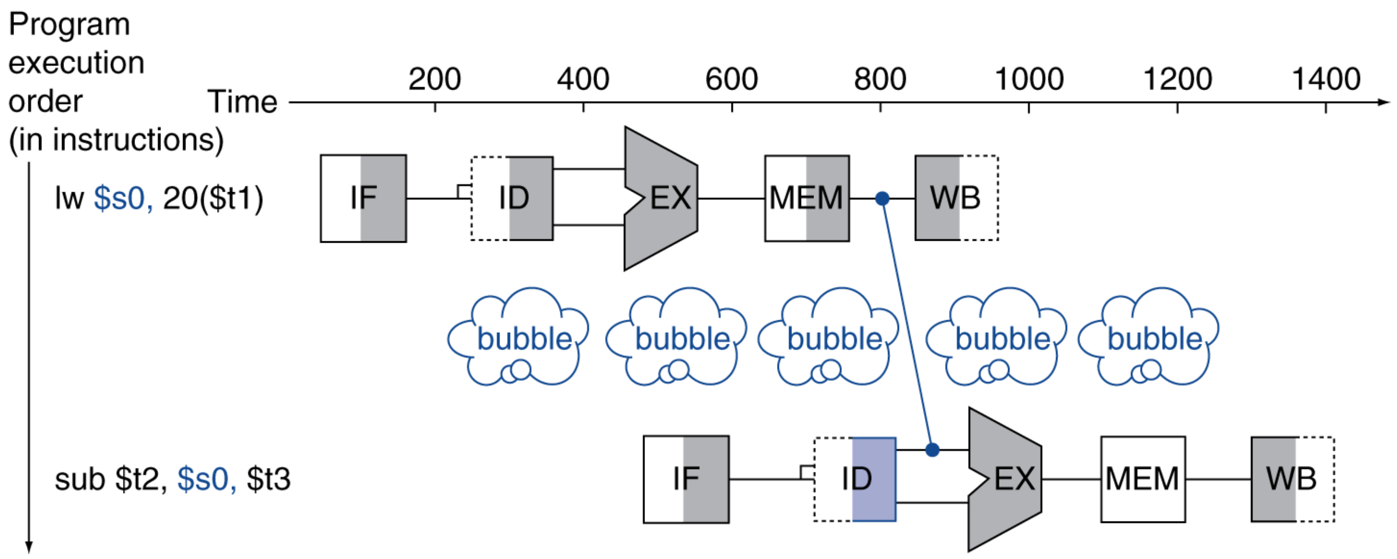


Forwarding

- Forwarding: the result is passed forward from an earlier instruction to a later instruction.

Load-Use Data Hazard

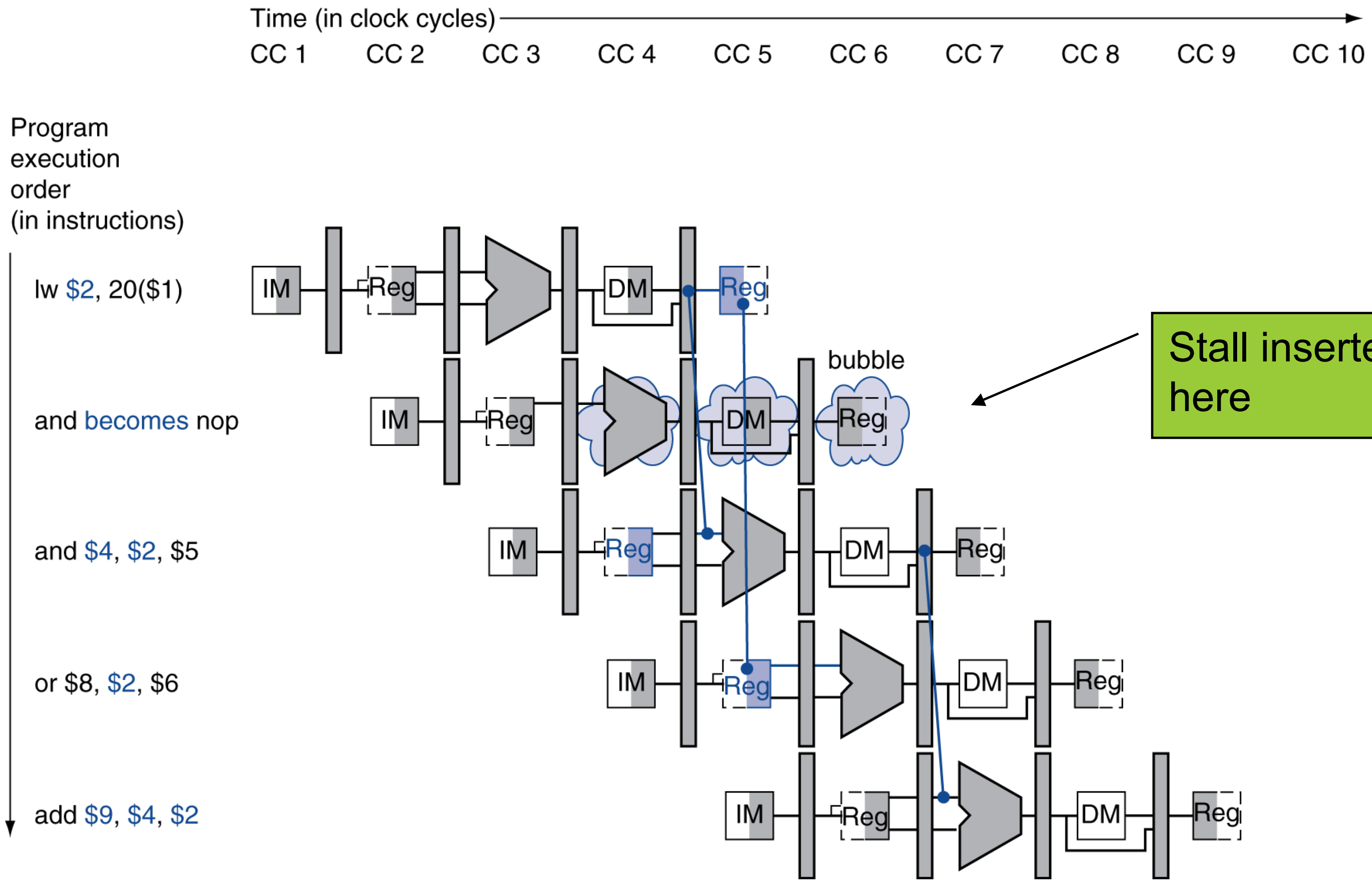
- Can’t always avoid stalls by forwarding



Pipeline Stalls

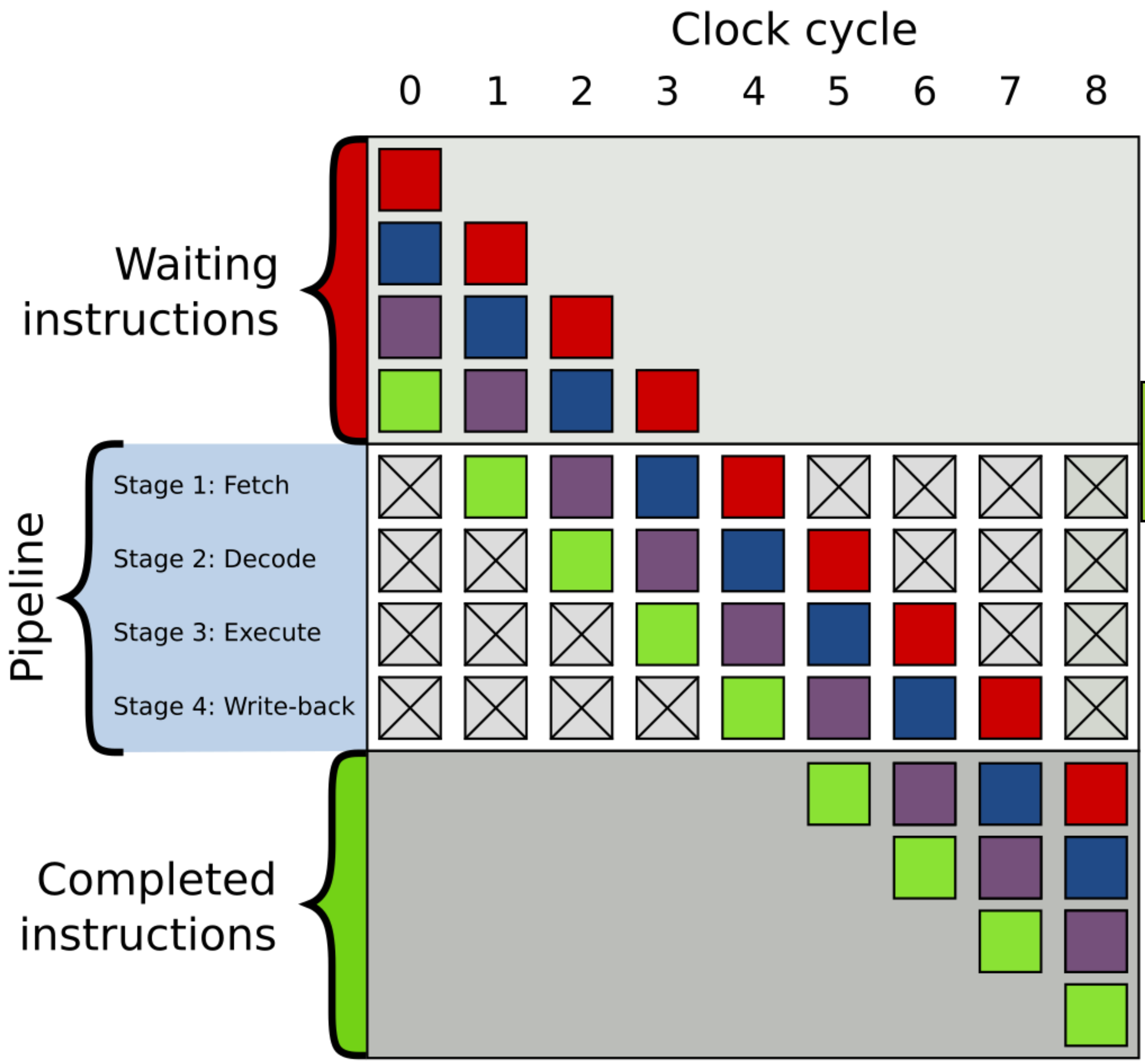
- A stall (bubble) is implemented with a nop
  - “No operation”
- Force control values in ID/EX register to 0

Stall/Bubble in the Pipeline



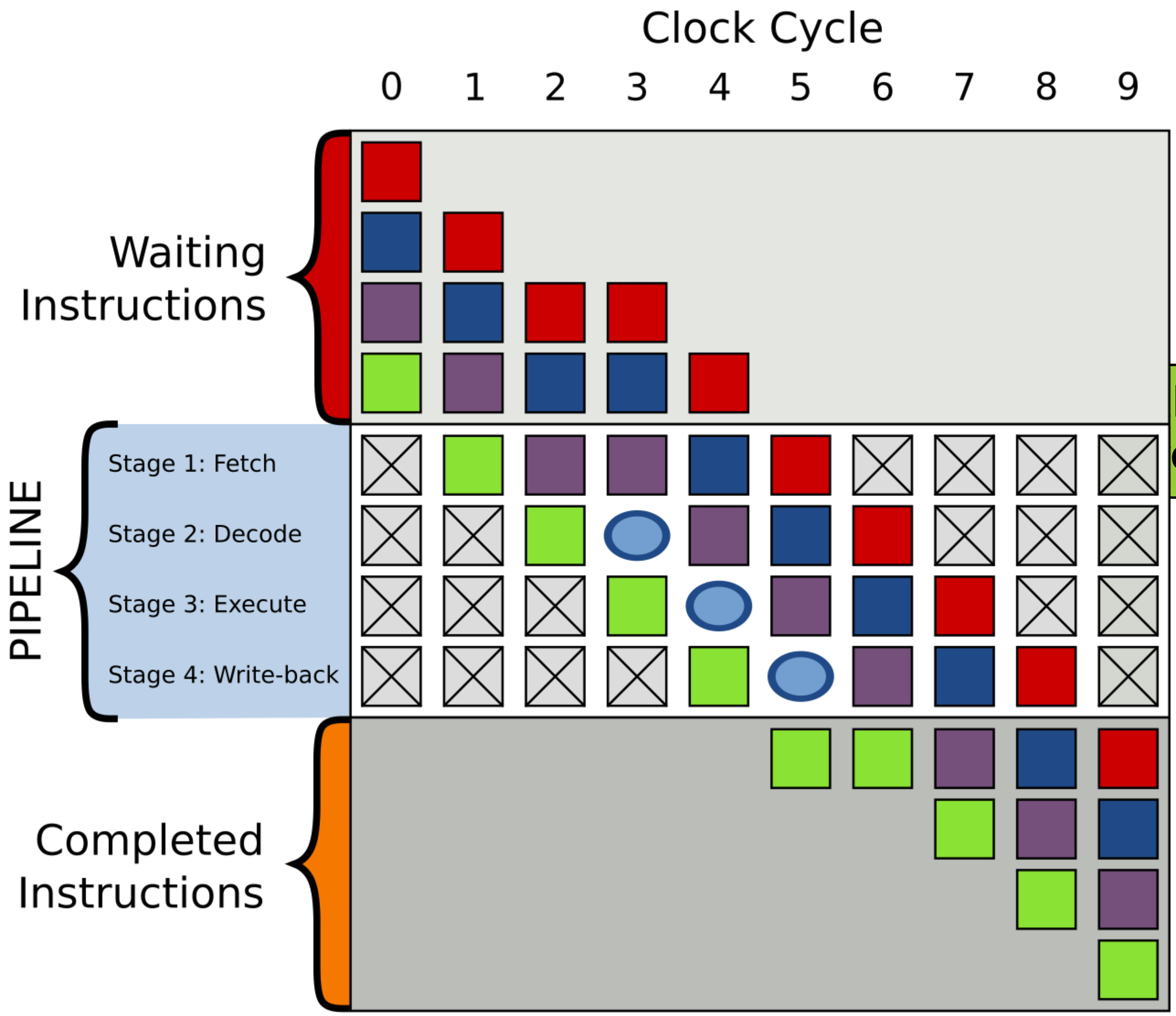
Stall inserted here

Another view (no bubble)...



Stall inserted here

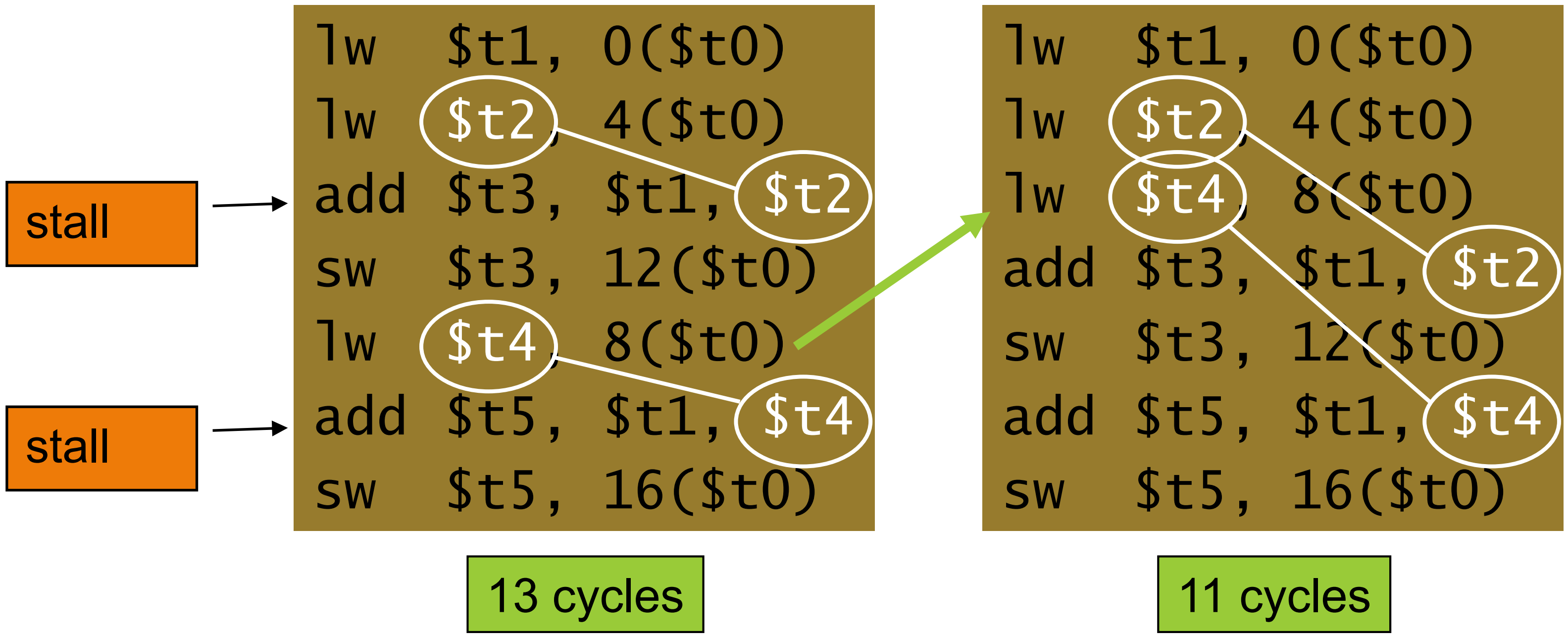
Another view (with bubble)...



ll inserted
e

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;`



```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

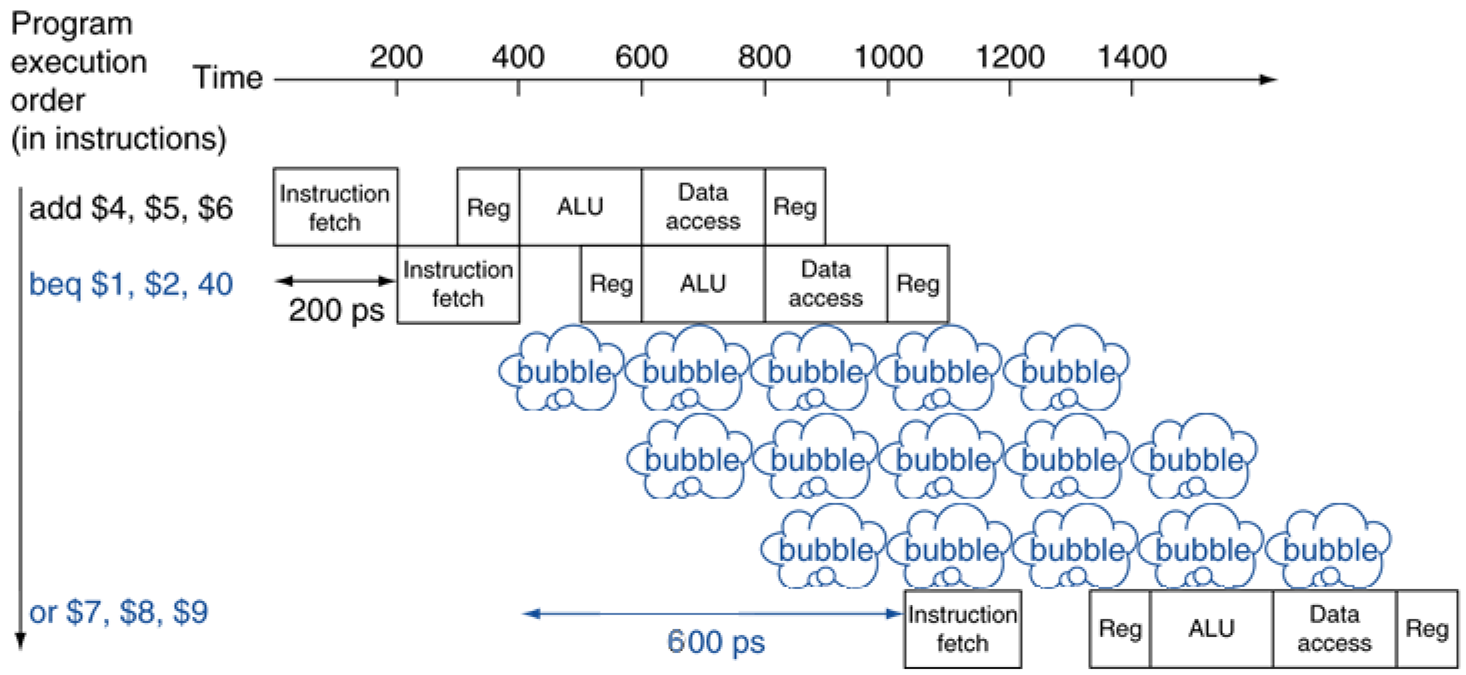
Control Hazards

- Branch 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

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

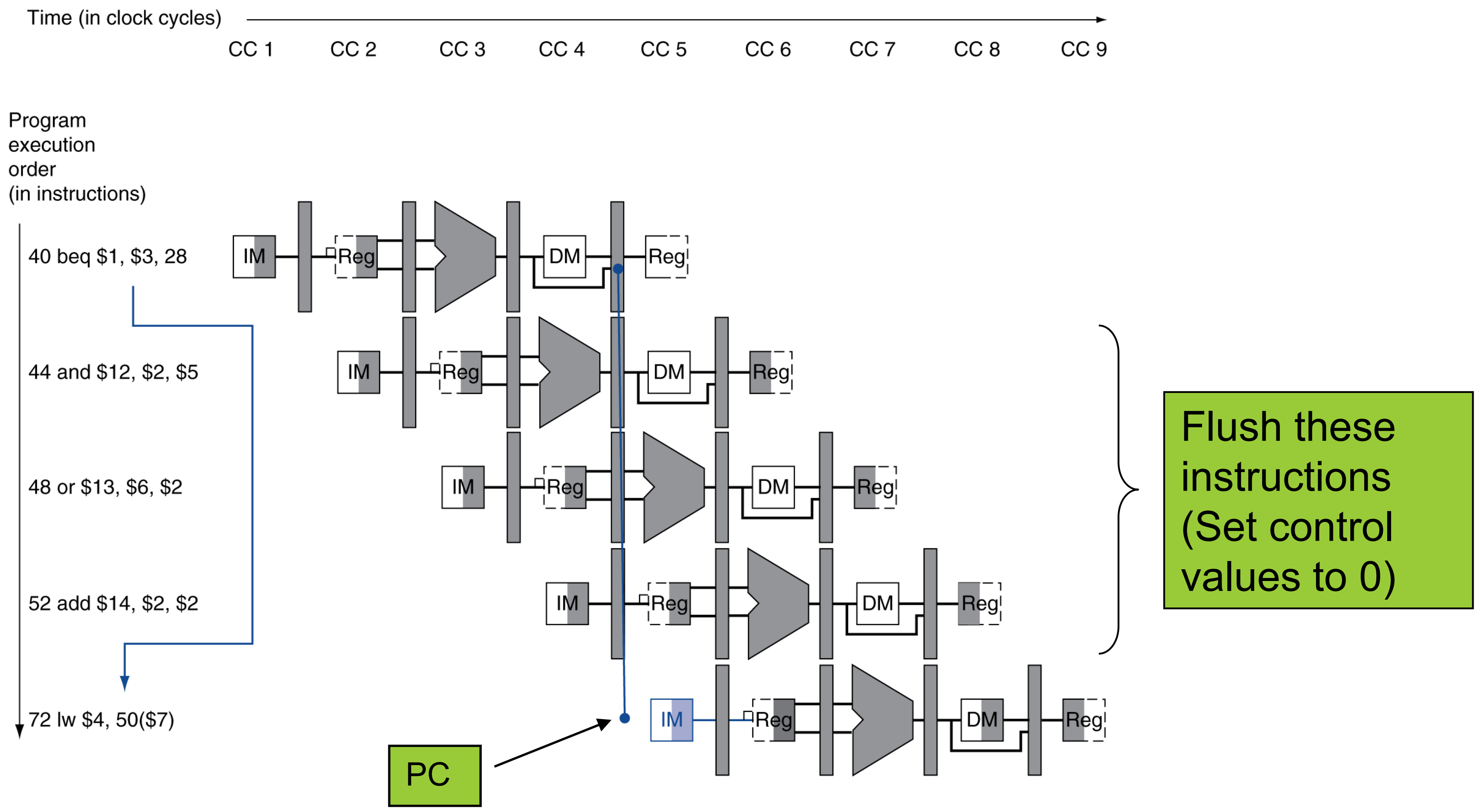


Branch Prediction

- 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

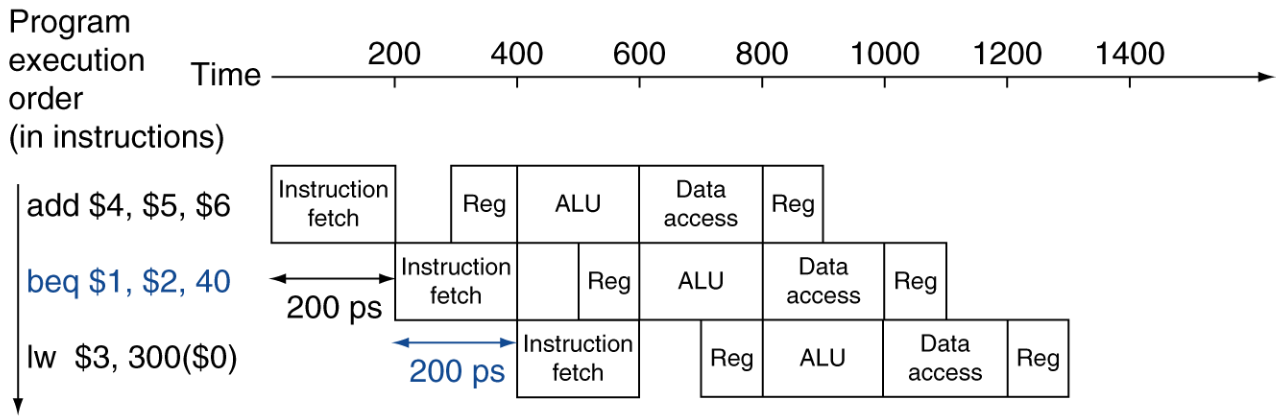
Predict Branch Not Taken

- If branch is determined as taken in MEM:

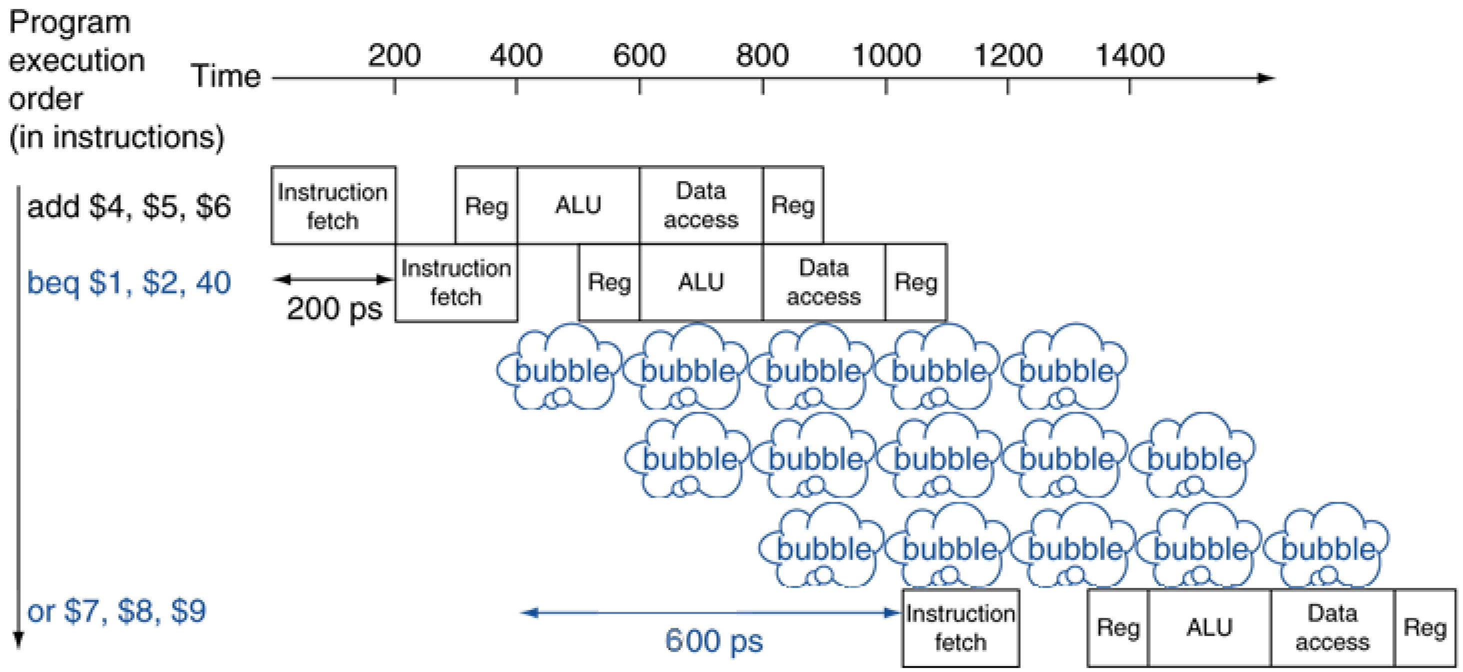


Predict Branch Not Taken

Prediction correct



Prediction incorrect



Predict Branch Not Taken

- If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

Branch Prediction

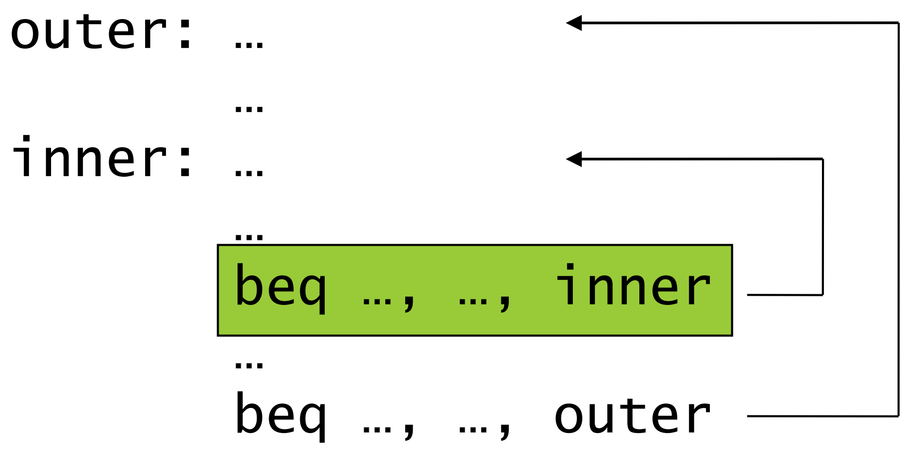
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch History Table
  - Indexed by recent branch instruction addresses
  - 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!

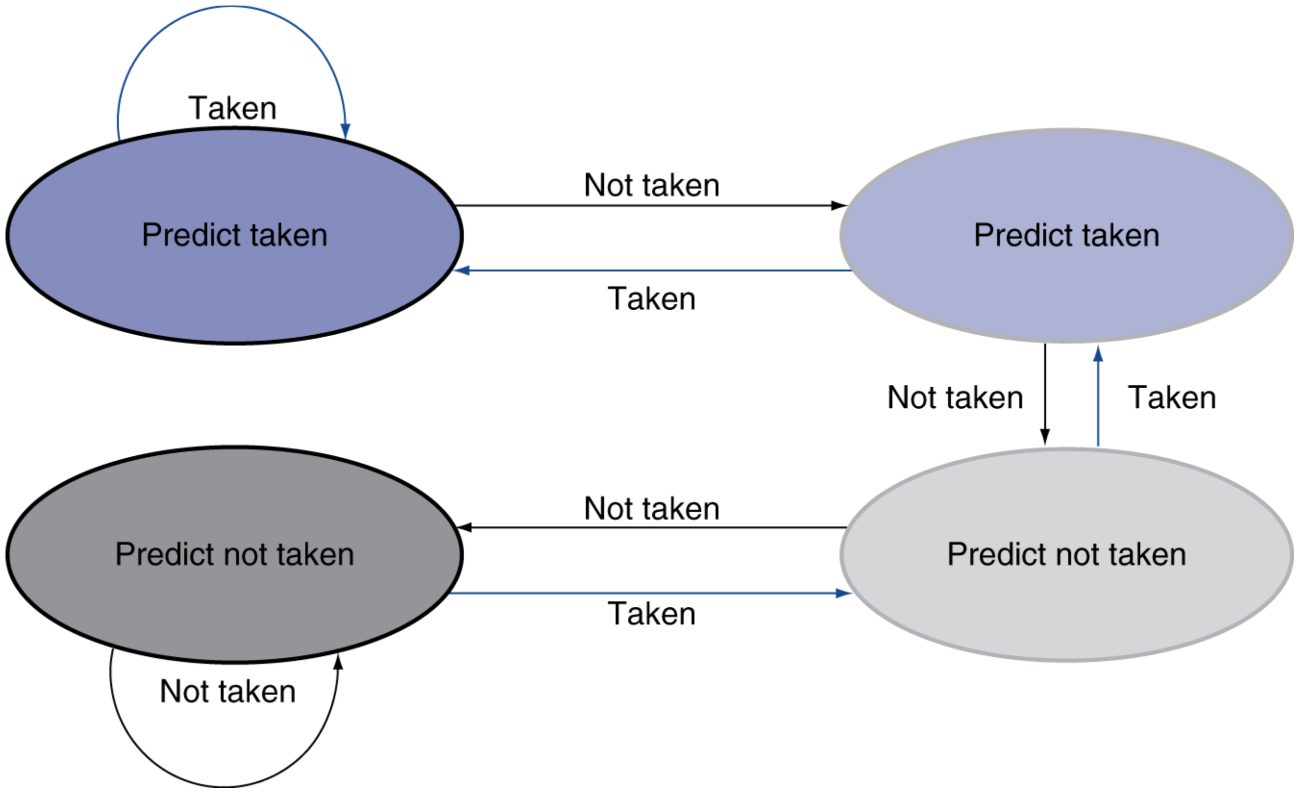


```
outer: …
       …
inner: …
       …
       beq …, …, inner
       …
       beq …, …, outer
```

  - 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



Delayed Decision

- Branch Instruction
- Execute the next sequential instruction
- Execute the instruction that resulted from the branch

• add $4, $5, $6	->	beq $1, $2, 40
• beq $1, $2, 40	->	add $4, $5, $6
• <branch result>		<branch result>

Cycle Time Matters

- Delayed branches work when branches are short
  - no processor uses a delayed branch of more than 1 cycle
- Longer branch delays rely on hardware based branch prediction

Reducing Branch Delay

- Insight:
  - many branches rely only on simple tests
    - can be implemented with a few gates over an ALU
  - more complex branches take two instructions
    - slt followed by beq/bne
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Reduces the penalty of a branch to only one instruction if the branch is taken

Summary

- 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

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control

- Exception
  - Arises within the CPU
  - Examples: undefined opcode, overflow, syscall

- Interrupt
  - From an external I/O controller

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: Status register
    - 0000 – undefined instruction
    - 0180 – arithmetic overflow

- Jump to handler

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

Exceptions in a Pipeline

- Another form of control hazard

- Consider overflow on add in EX stage

  ```
  add $1, $2, $1
  ```

  - Prevent $1 from being overwritten
  - 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

Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions

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

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
  - 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
  - More instructions completed per second
  - Latency for each instruction not reduced

- Hazards: structural, data, control

- Exceptions are handled by a outside handler and control hazard logic.