CS 31007 Autumn 2021

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

Rajat Subhra Chakraborty (*RSC*)
Bhargab B. Bhattacharya (*BBB*)

Lecture #26: Processor Design
27 September 2021

Indian Institute of Technology Kharagpur
Computer Science and Engineering

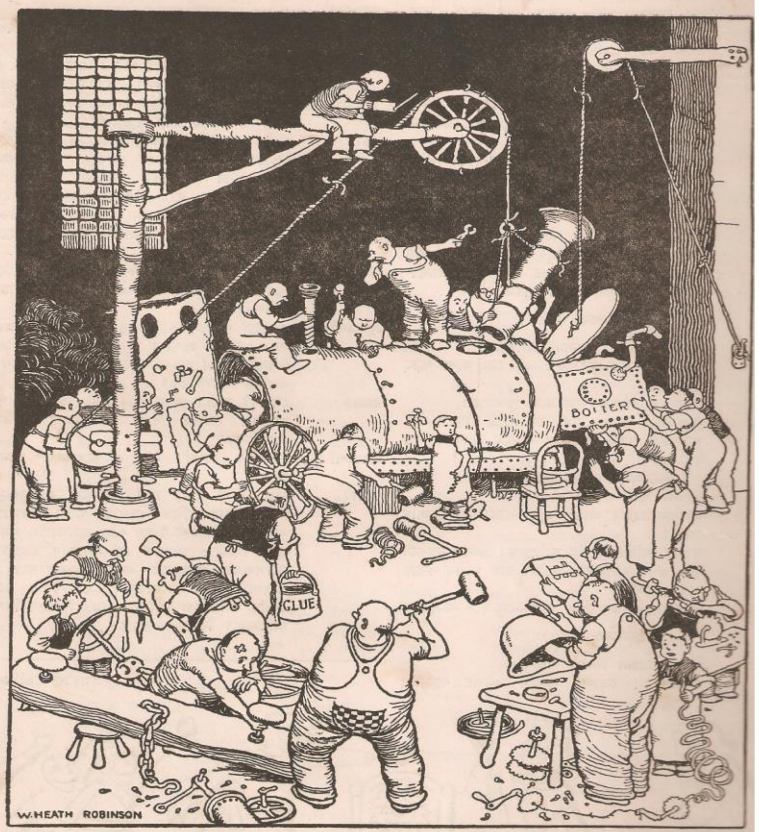
So far covered …

- Introduction, CPU Performance Equation
- MIPS Assembly Language
- Number Systems: Integers, Floating-Point
- Computer Arithmetic: Algorithms and Hardware

- Today: Moving into Processor Design ..

Building a Computer System ……

রেলগাড়ির আদিপর্ব।১



কত লোক খাটে কারখানাতে কত মাথা করে ঝিম ঝিম
কত কাঠখড় পোড়ে বানাতে সেকালের রেলের ইনজিন।

Courtesy: Satyajit Ray

Building a Computer System ……

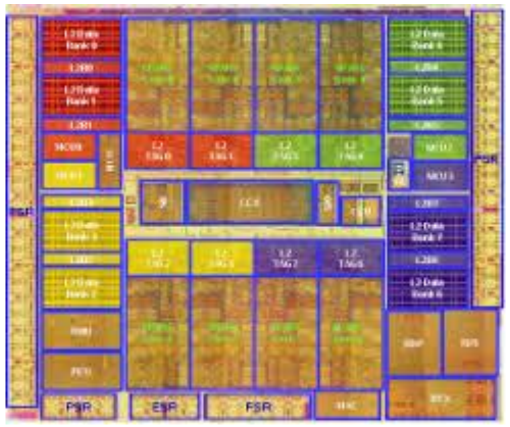
CPU:

- Steering wheel of a computer system – Hardware realization
- Should be able to execute each instruction
- Arithmetic, data transfer, branch, etc. ..
- Coordinate among all modules
- Overall control of computing processes

Challenges

- What are the issues in processor design?

  cost, speed, capability, yield, portability, testing and fault-tolerance, security, IP-protection, market demand



- What are the inputs?
- What are the goals?
- What are the constraints?
  - cost, power, heat dissipation, silicon area, time-to-market
- How one can automate the entire design process?

Inputs

- ISA: Instruction Set Architecture
- -- instruction sets, formats, addressing modes, register description, word length;
- Components: Basic ALU blocks: adders, multipliers ..
- Other logic blocks – MUX-es, Flip-Flops (FF)
- Register files, Program Counter (PC)
- Memory
- Clock for synchronizing FFs

Output

- Hardware design that is capable of executing instructions under program control

Issues

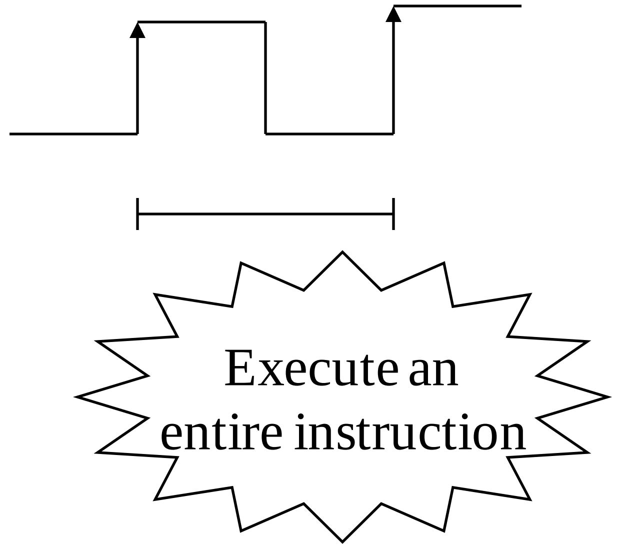
- CPU performance: IC, CPI, CCT
  - Instruction count (IC)
    - Determined by ISA and compiler
  - CPI and Clock Cycle time (CCT)
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version (single-cycle execution)
  - A more realistic and faster pipelined version
- Simple subset of ISA, covering most aspects
  - Memory reference: *lw*, *sw*
  - Arithmetic/logical: *add*, *sub*, *and*, *or*, *slt*
  - Control transfer: *beq*, *j*

Performance

- Execution Time = IC * CPI * CCT
- Processor design (datapath and control) will determine:
  - Clock cycle time (CCT)
  - Clock cycles per instruction (CPI)

Single-cycle processor:

- Advantage: CPI = 1
- Disadvantage: long cycle time

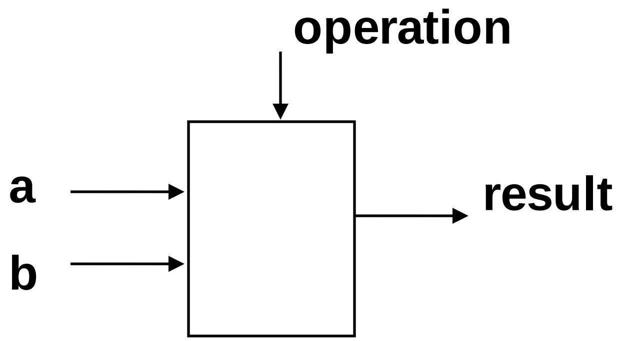


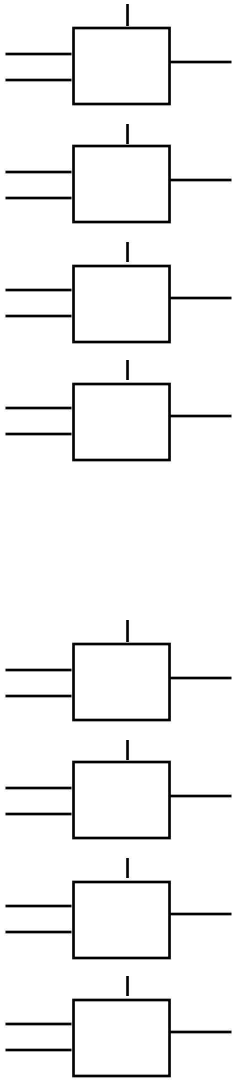
Start with Designing a Simple ALU

- 1-bit ALU
- 32-bit ALU

An ALU (arithmetic logic unit)

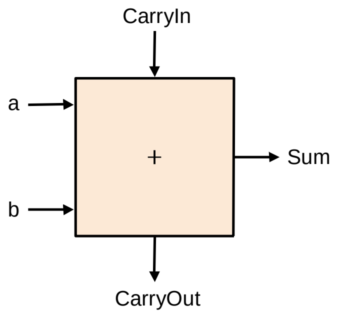
- Let's build an ALU to support `and, or, add, sub` (for integers) instructions
  - just build a 1-bit ALU, and use 32 of them





Implementation

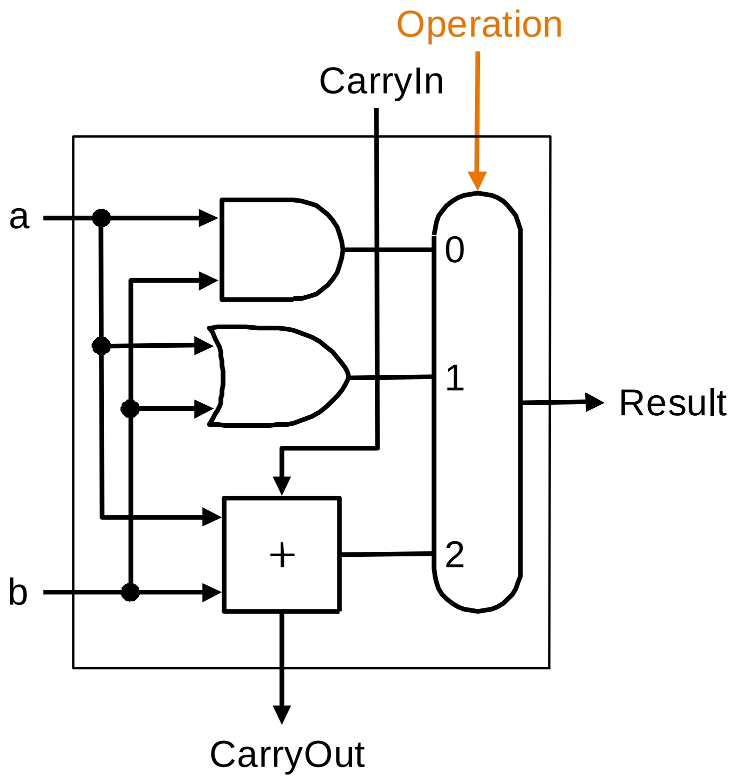
- Let's look at a 1-bit ALU for addition:



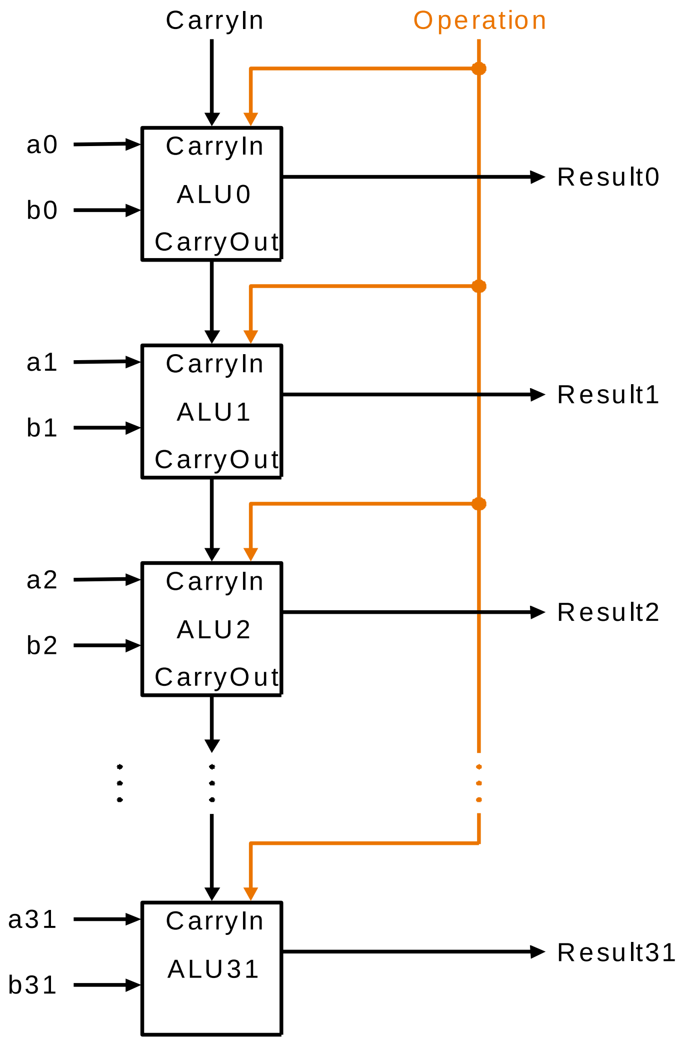
$$c_{out} = ab + ac_{in} + bc_{in}$$
$$sum = a \oplus b \oplus c_{in}$$

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

Building a 32-bit ALU (recall RCA)



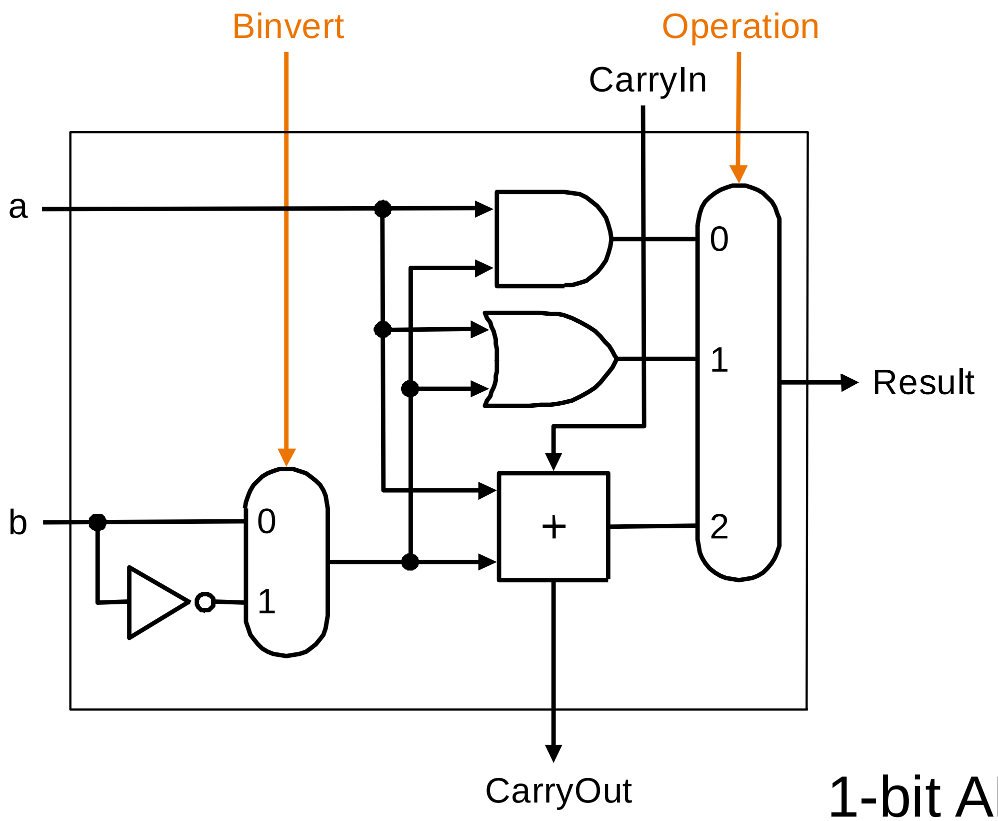
1-bit ALU for performing AND, OR, ADD



32-bit ALU for performing AND, OR, ADD

Subtraction (a – b)

- Two's complement approach: just negate *b* and add
  => set *Binvert* = 1; *Carry-in* = 1



1-bit ALU for performing AND, OR, ADD, SUB

Adding other MIPS instructions to ALU

- Support the *set-on-less-than* instruction (*slt*)

- `slt rd, rs, rt`

  => if (rs < rt), set rd = 1; else rd = 0;

  => use subtraction: (rs – rt) < 0 implies rs < rt

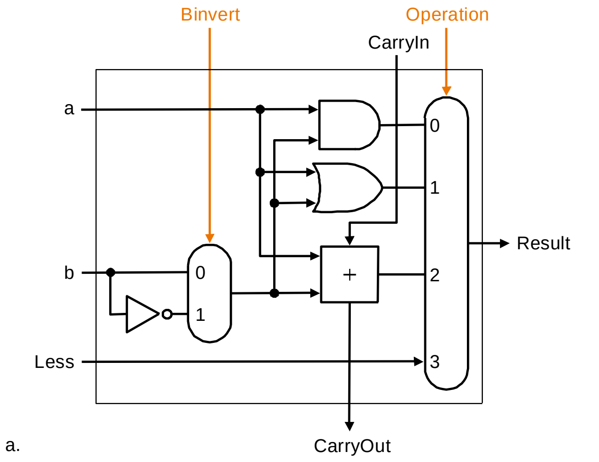
  Support test for equality (beq $t1, $t2, Label)

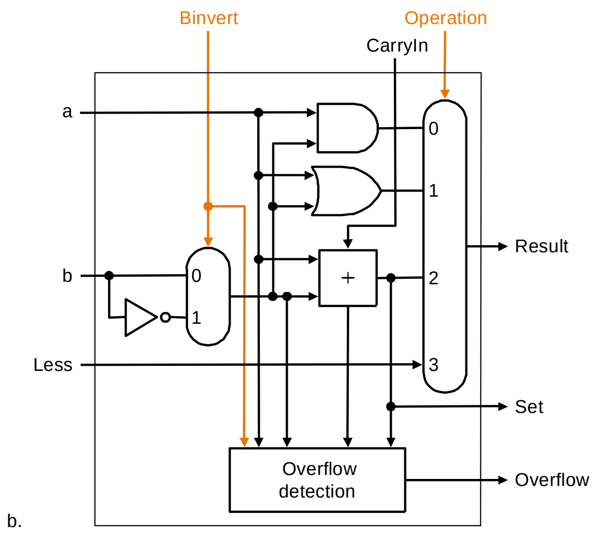
  => use subtraction: ($t1 - $t2) = 0 implies $t1 = $t2

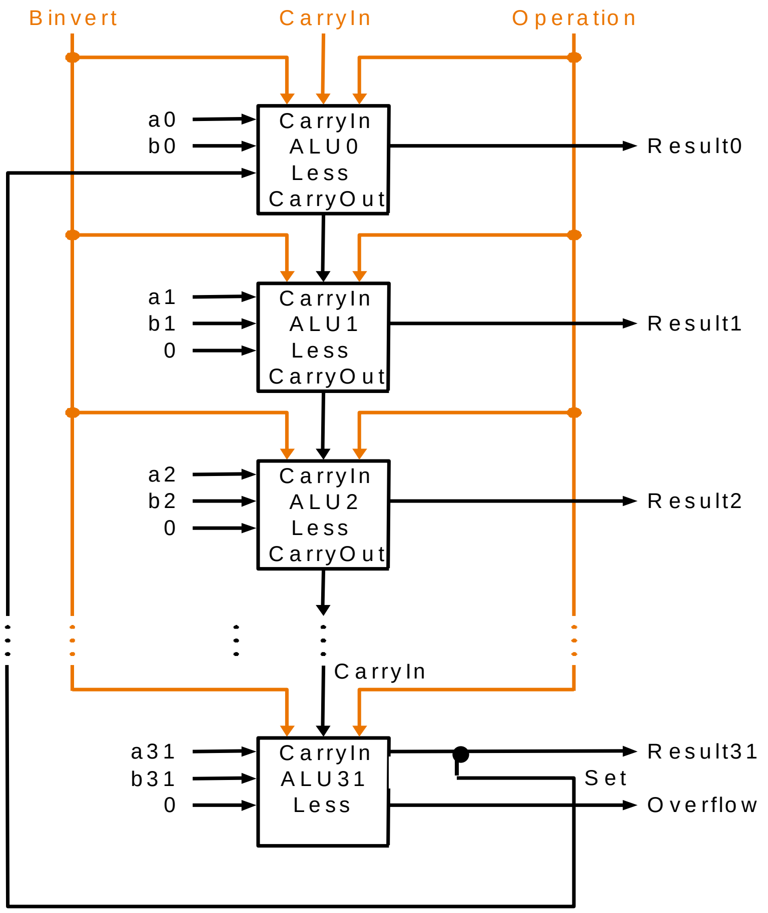
Supporting *slt rd, rs, rt*

slt:
if (rs < rt), set rd = 1; else rd = 0

When the condition is true, i.e.
rs – rt < 0 (negative) ⇒
rd: 0000 …………..0001







Implementing *beq* $t1, $t2, Label

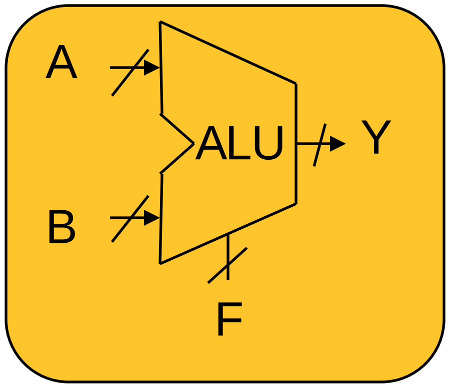
- **Notice control lines:**

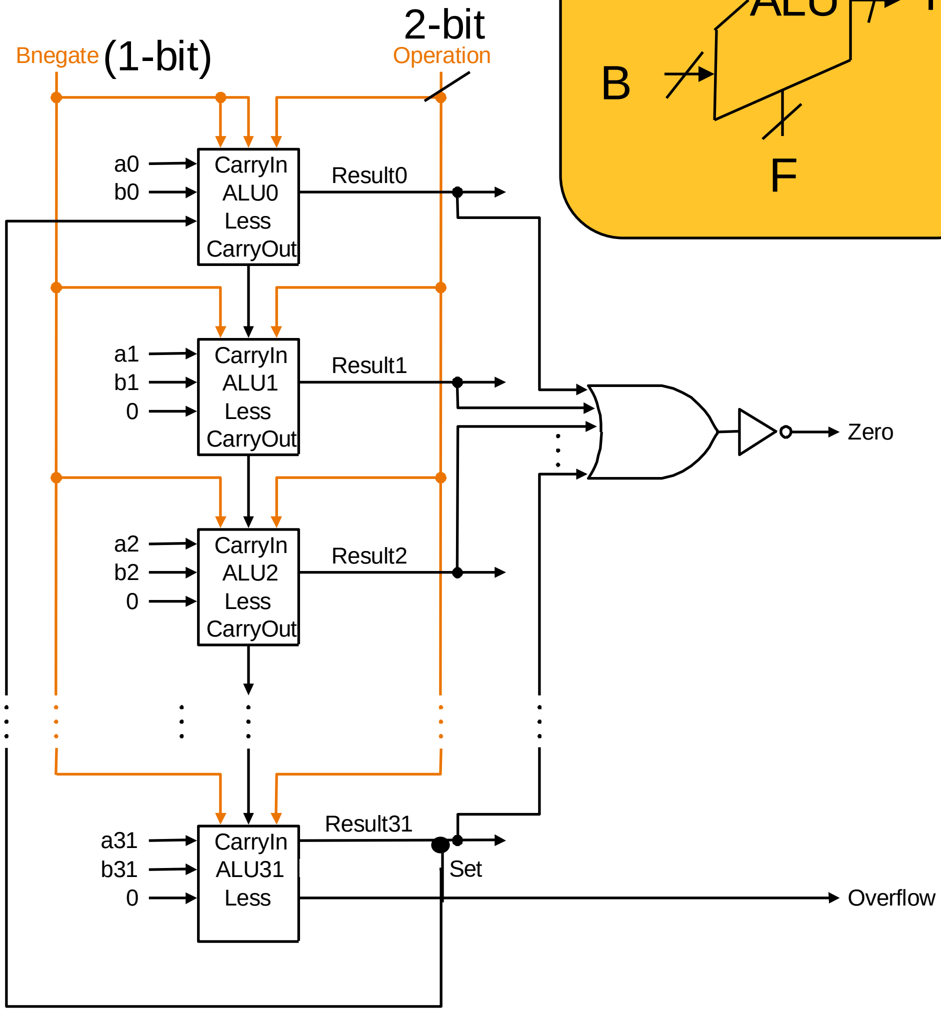
```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

output **Zero** becomes 1

when *beq* is taken;

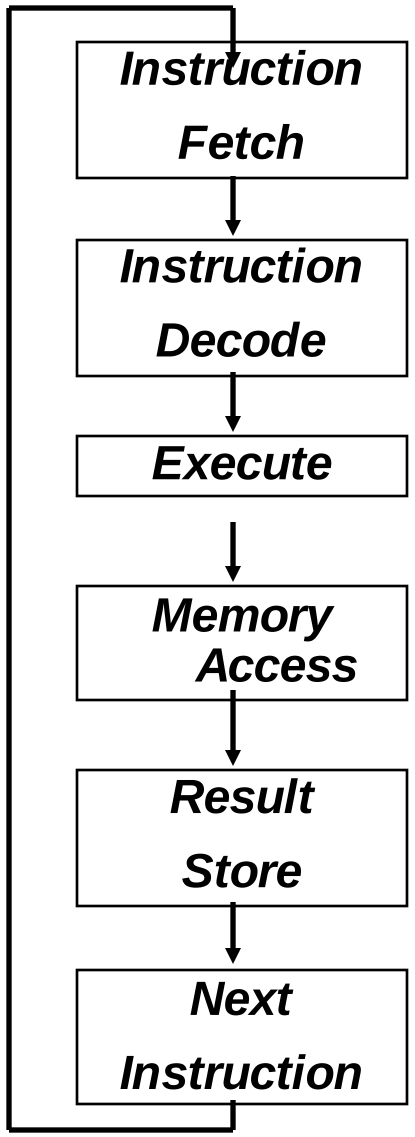
PC needs to updated …





This ALU implements MIPS *add*, *sub*, *and*, *or*, *slt,* and *beq*

Recap: Execution Cycle (MIPS)



Obtain instruction from memory **(IF)**

Instruction-Decode and Register-Read **(ID/RR)**

CPU Design: Needs hardware implementation of all steps

Execute the instruction **(Ex)**

Locate and obtain operand data from memory/register **(Mem)**

Write-Back results in register/memory for later use **(WB)**

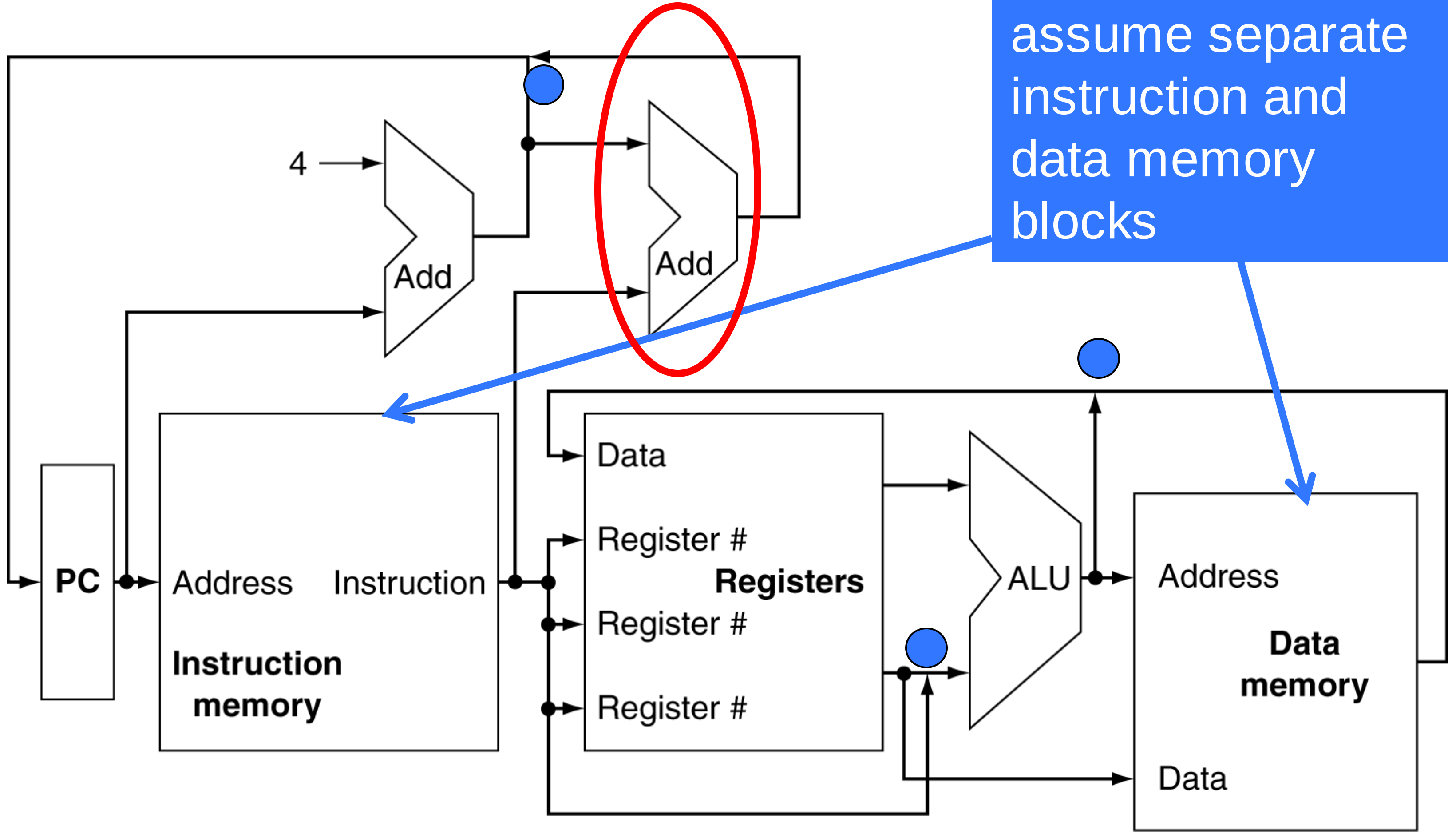
Determine successor instruction

IF → ID/RR → EX → Mem → WB

Instruction Execution

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

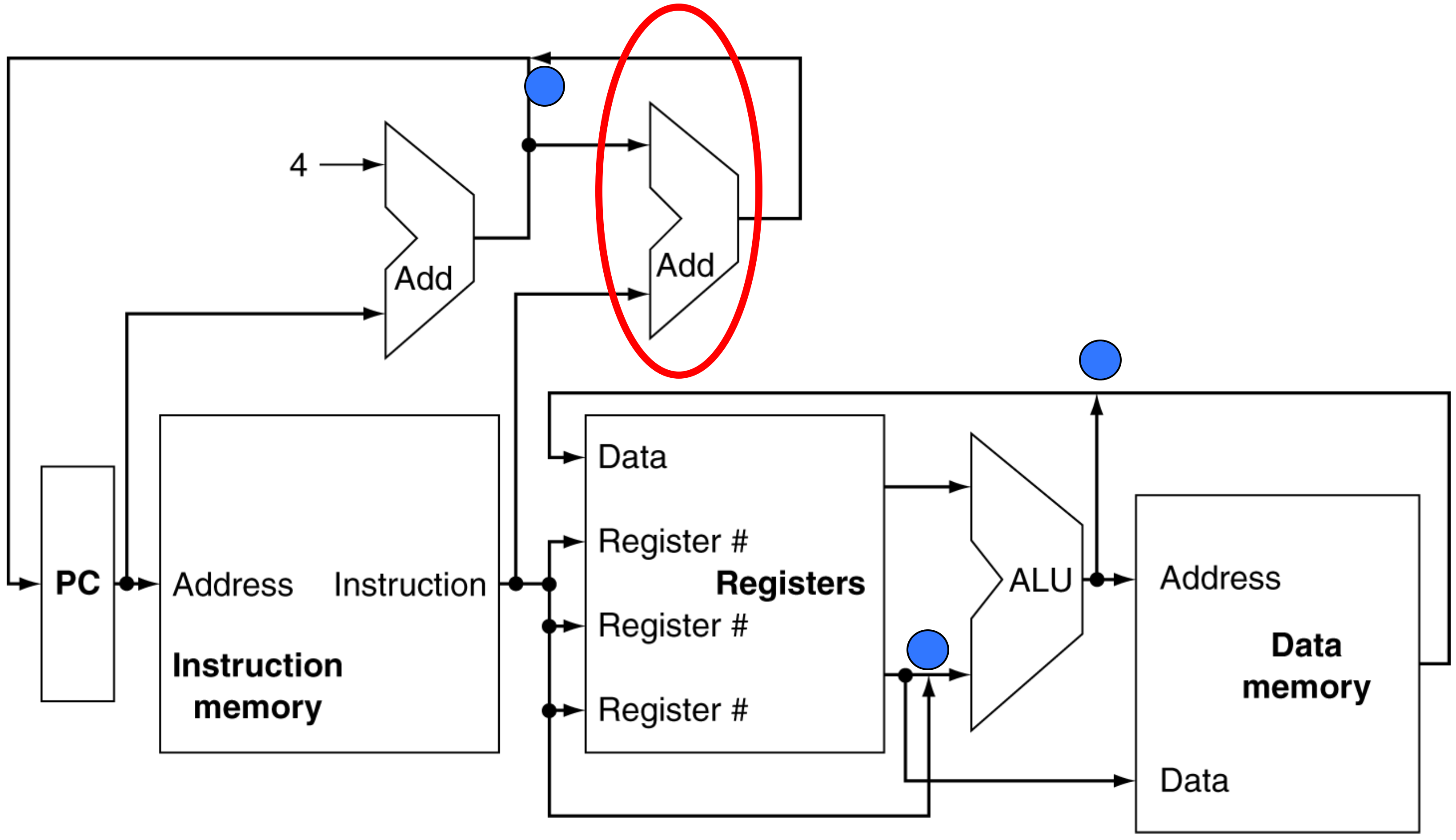
CPU Overview: Fixing data and control paths



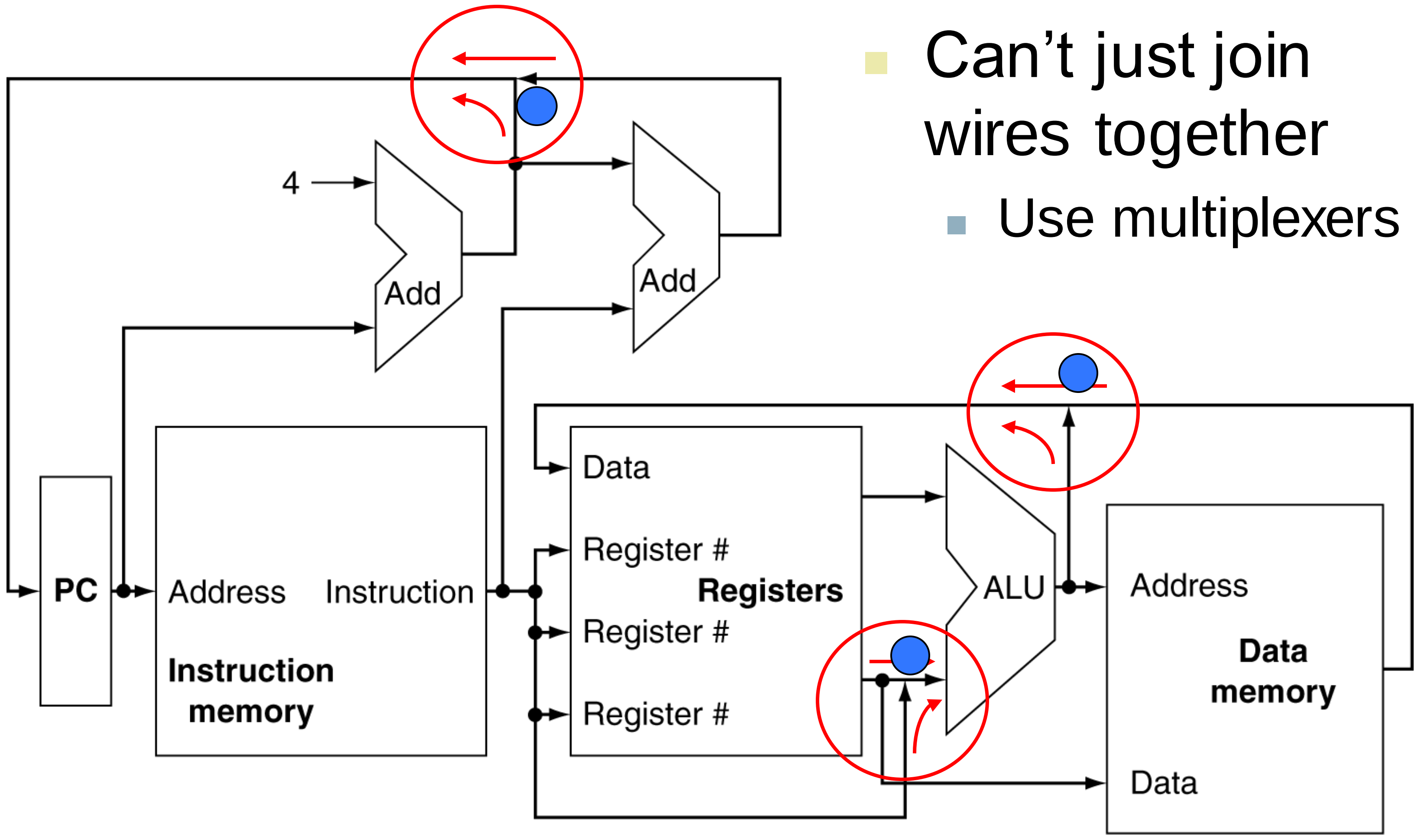
For simplicity, assume separate instruction and data memory blocks

Two types of functional units:

- elements that operate on data values (combinational)
- elements that contain state (sequential)



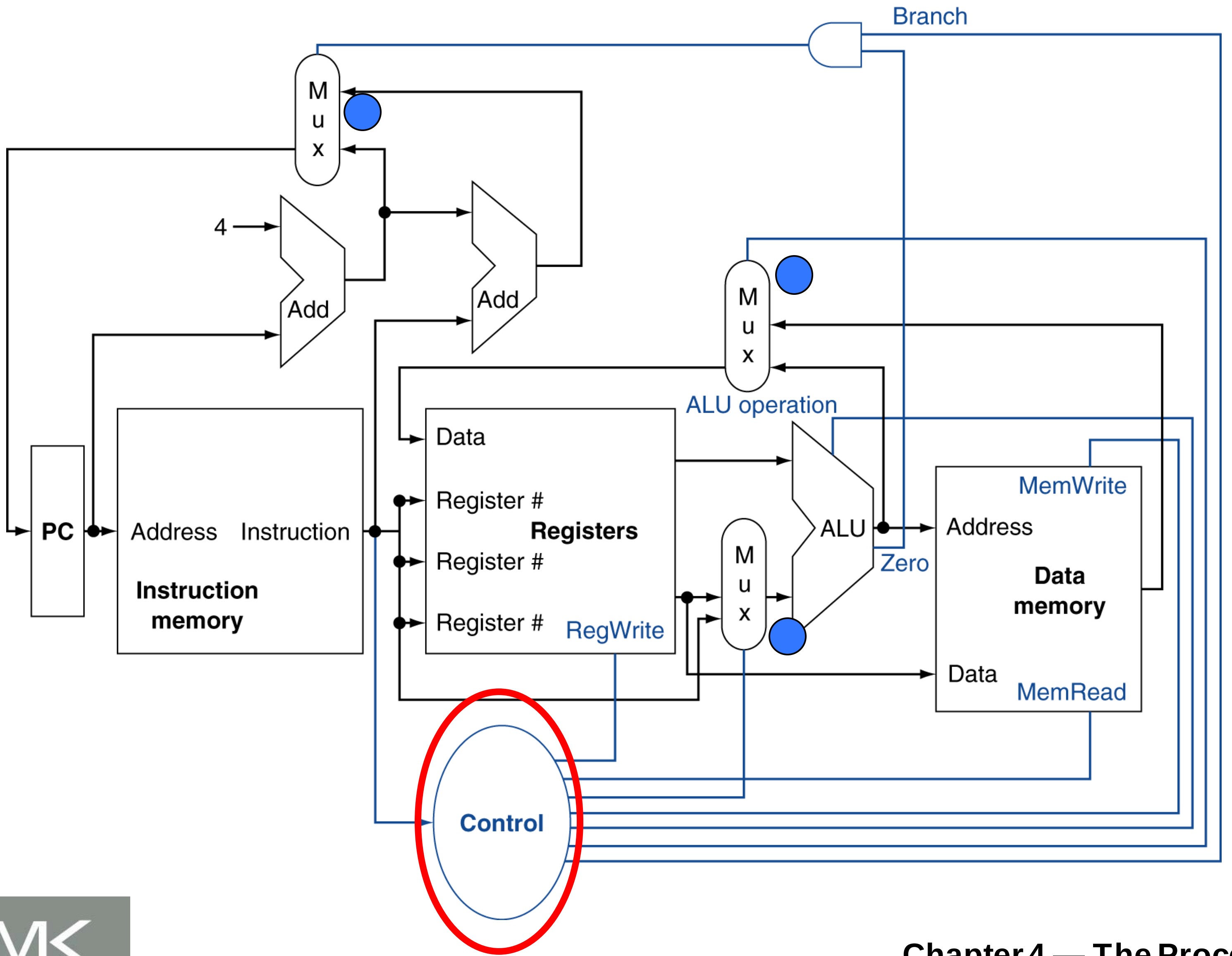
Multiplexers for stitching multiple paths



- Can’t just join wires together
  - Use multiplexers

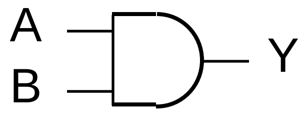
Control

IF → ID/RR → EX → Mem → WB

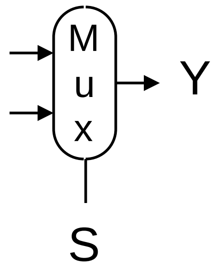


Combinational Elements

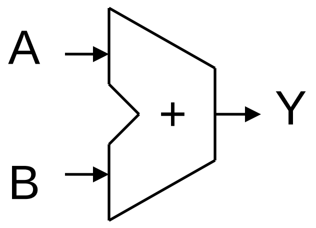
- AND-gate
  - Y = A & B



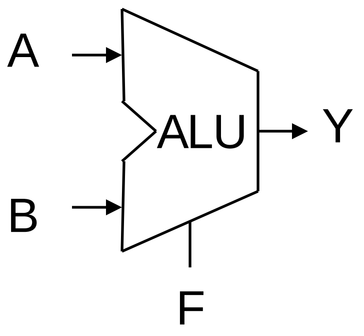
- Multiplexer



- Adder
  - Y = A + B

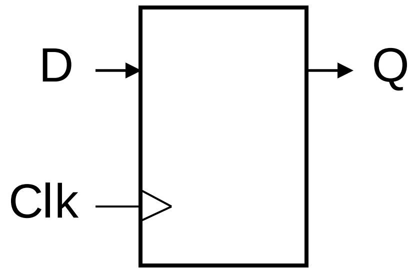


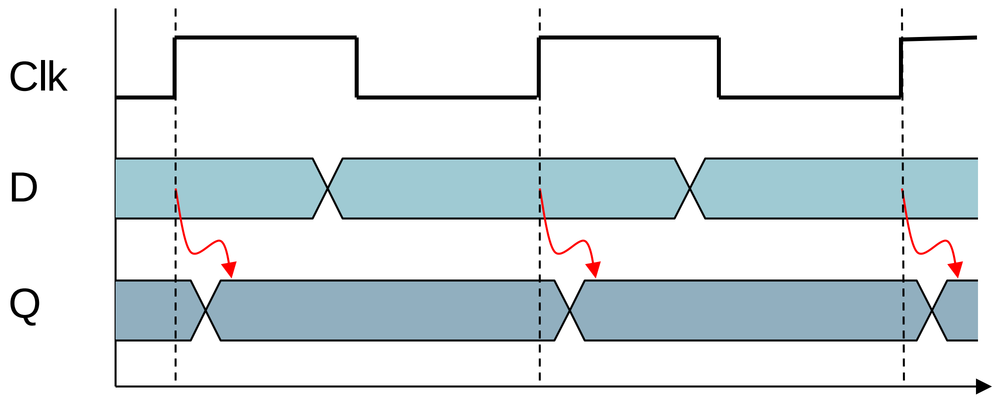
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

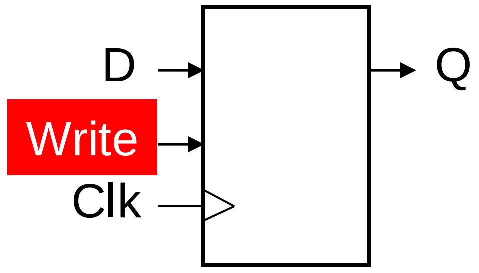
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

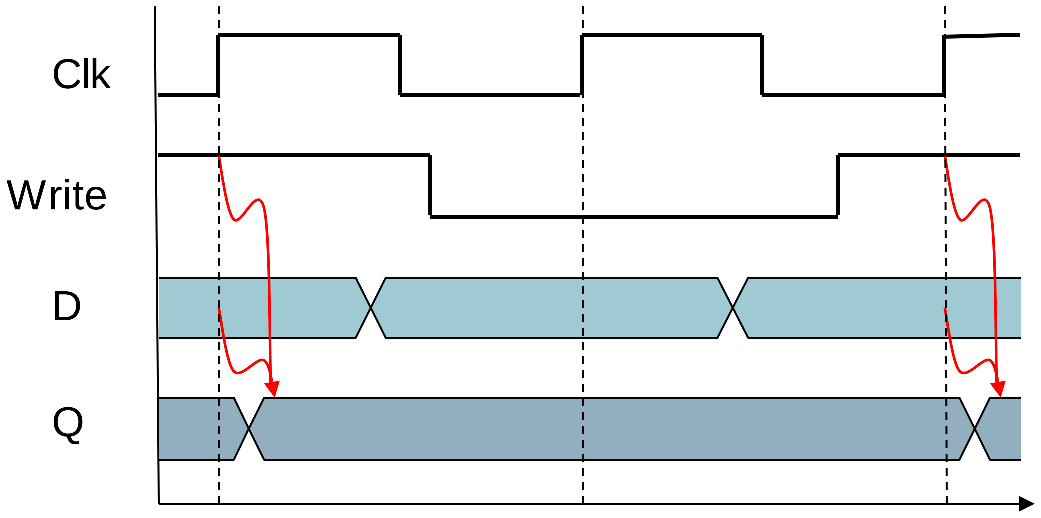




Sequential Elements

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



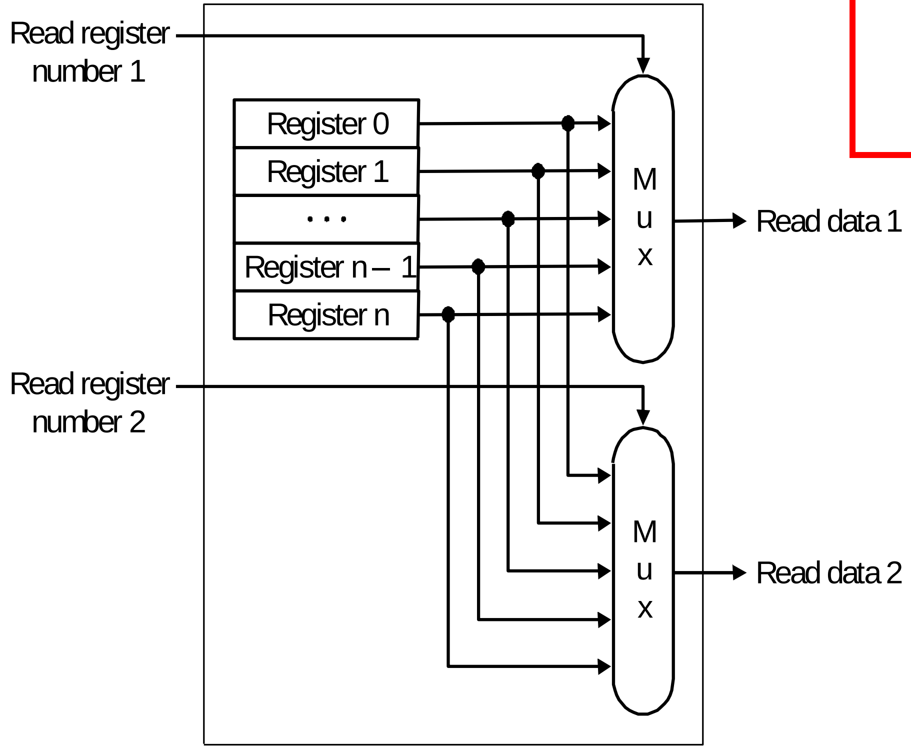


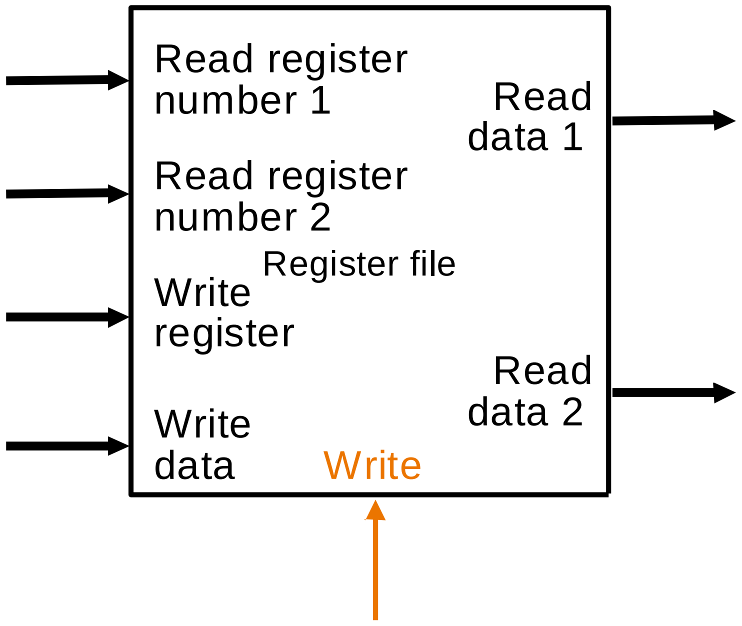
Register File: Read

- Built using D flip-flops

MIPS has a 32 × 32-bit register file
    Use for frequently accessed data
    Numbered 0 to 31

Assembler names
    $t0, $t1, …, $t9 for temporary values
    $s0, $s1, …, $s7 for saved variables

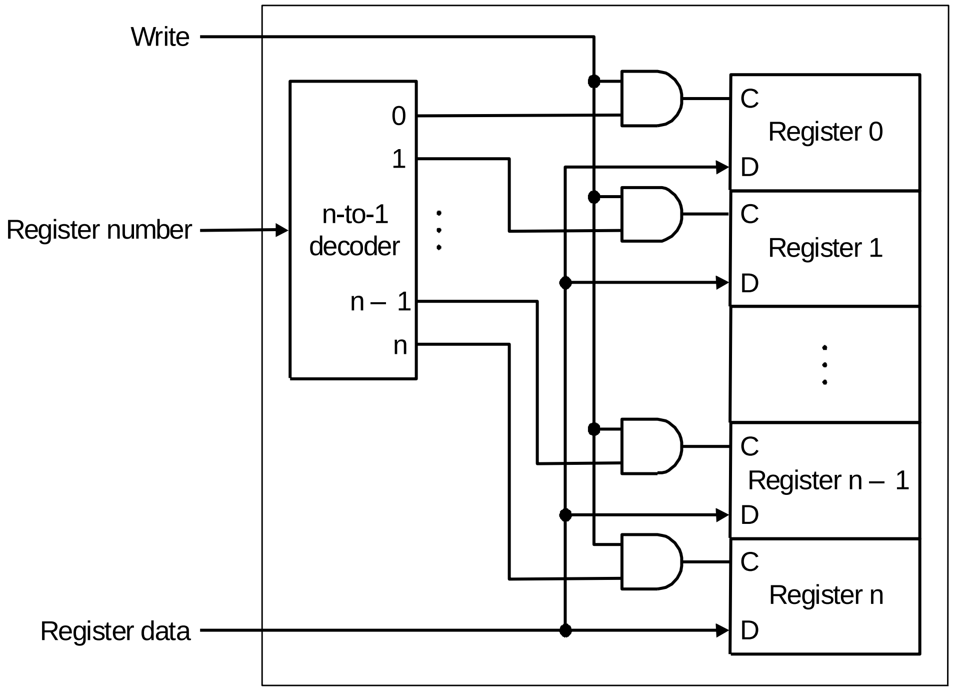




add rd, rs, rt

Register File: Write

- use the clock to determine when to write, provided *Write* is enabled



add rd, rs, rt

CS 31007 Autumn 2021

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors

Rajat Subhra Chakraborty (*RSC*)

Bhargab B. Bhattacharya (*BBB*)

Lecture #27, #28

Single-Cycle Processor Design

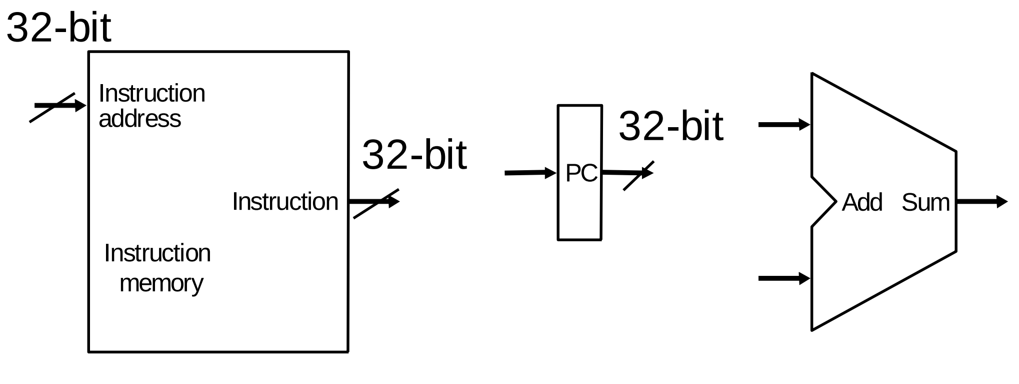
28 September 2021

Indian Institute of Technology Kharagpur

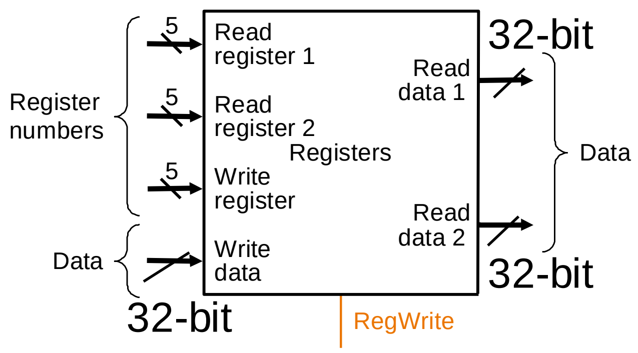
Computer Science and Engineering

Simple Hardware Implementation of a CPU

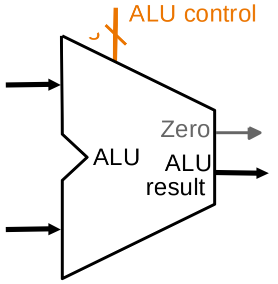
- Functional units we need for implementing instructions



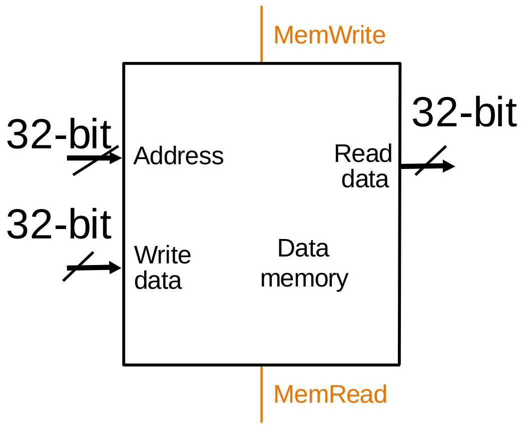
a. Instruction memory b. Program counter c. Adder



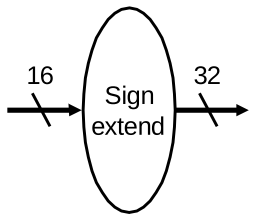
a. Registers



b. ALU



a. Data memory unit



b. Sign-extension unit

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

Processor Design

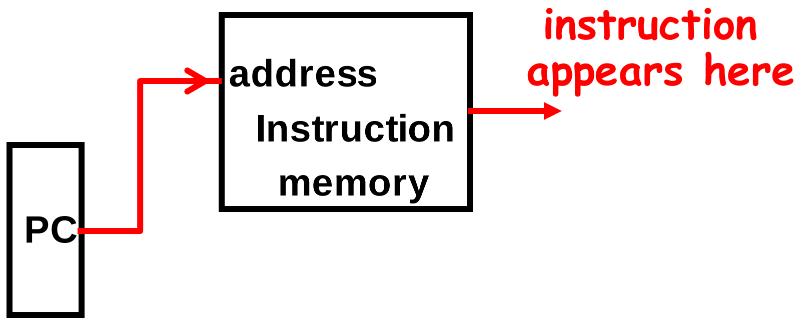
- We're ready to implement the MIPS "core"
  - load-store instructions: `lw, sw`
  - reg-reg instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq`

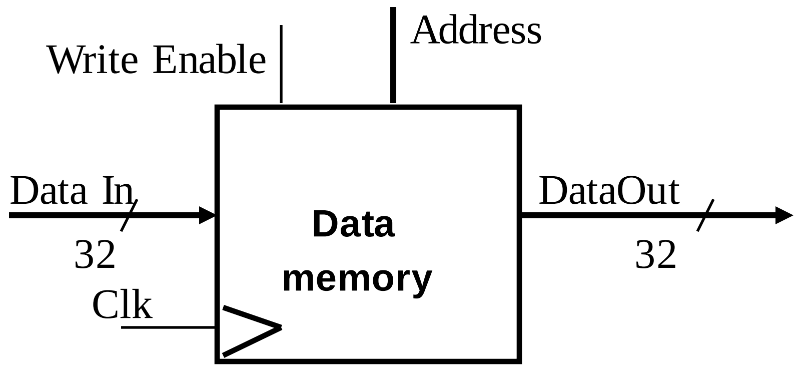
First, we need to fetch an instruction into processor

- program counter (PC) supplies instruction address
- retrieve the instruction from memory

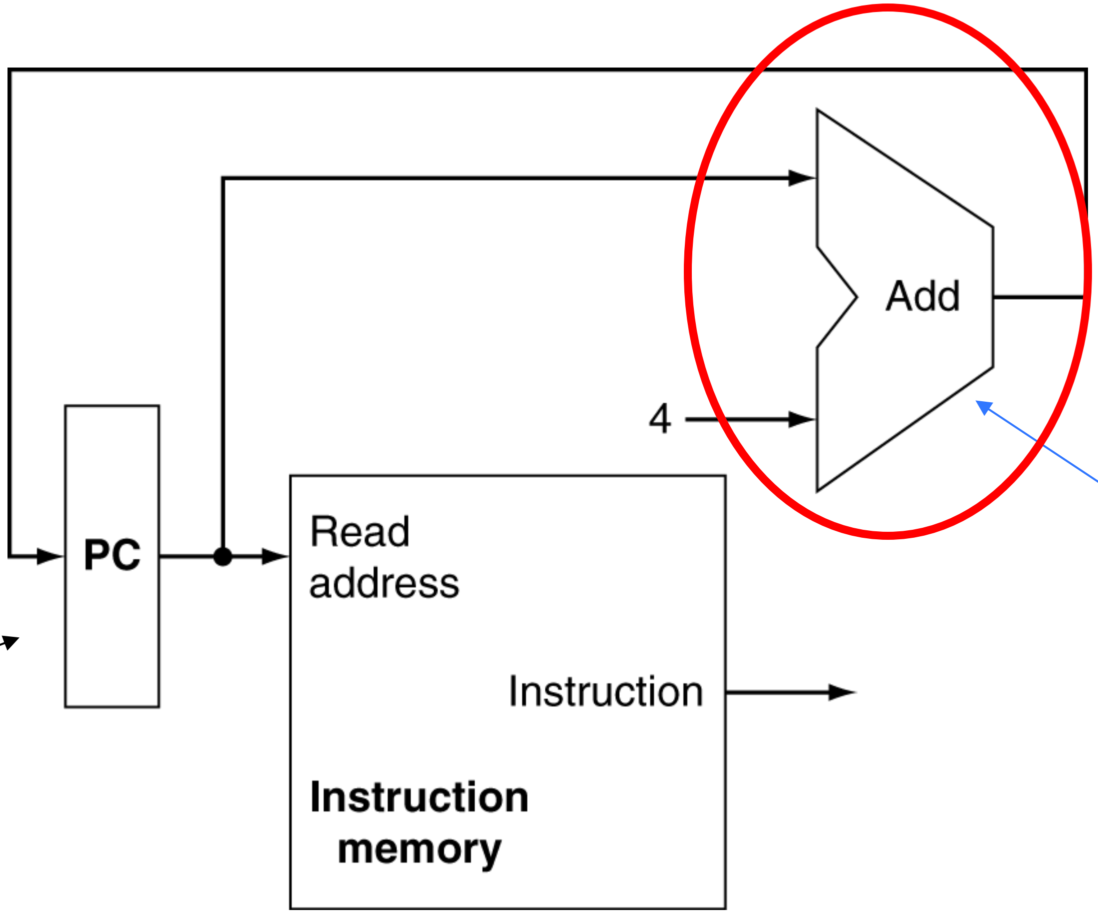
Accessing Instruction & Data in the Same Cycle?

Solution: Use separate instruction and data memory





Instruction Fetch



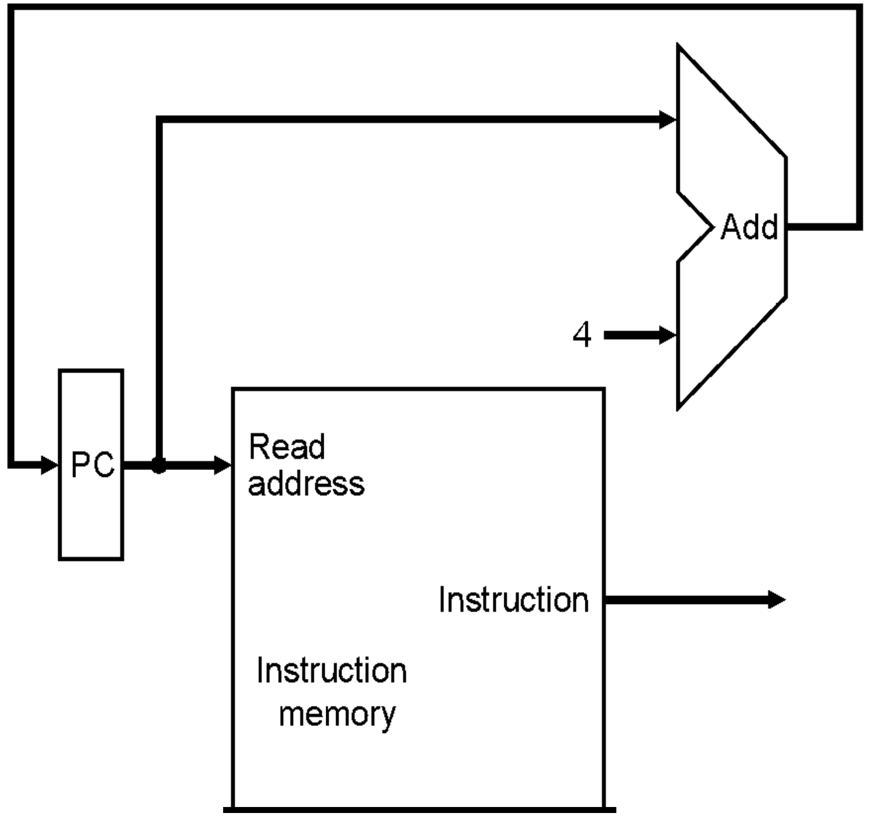
32-bit register

Increment by 4 for next instruction

Instruction Fetch Unit

Updating the PC for next instruction

- Sequential Code: PC <- PC + 4
- **Branch and Jump: PC <- "target address"**
  (to be fixed later)



MIPS Instruction Subset

- <u>**R-type**</u>
  - ***add rd, rs, rt***
  - **sub, and, or, slt**

31 26	21	16	11	6	0
op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

1. Read registers rs and rt
2. Feed them to ALU
3. Update register file

- <u>**LOAD**</u> **and** <u>**STORE**</u>
  - **lw rt, rs, imm**
  - **sw rt, rs, imm**

31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits

1. Read register rs (and rt for store)
2. Feed rs and immed to ALU
3. Move data between mem and reg

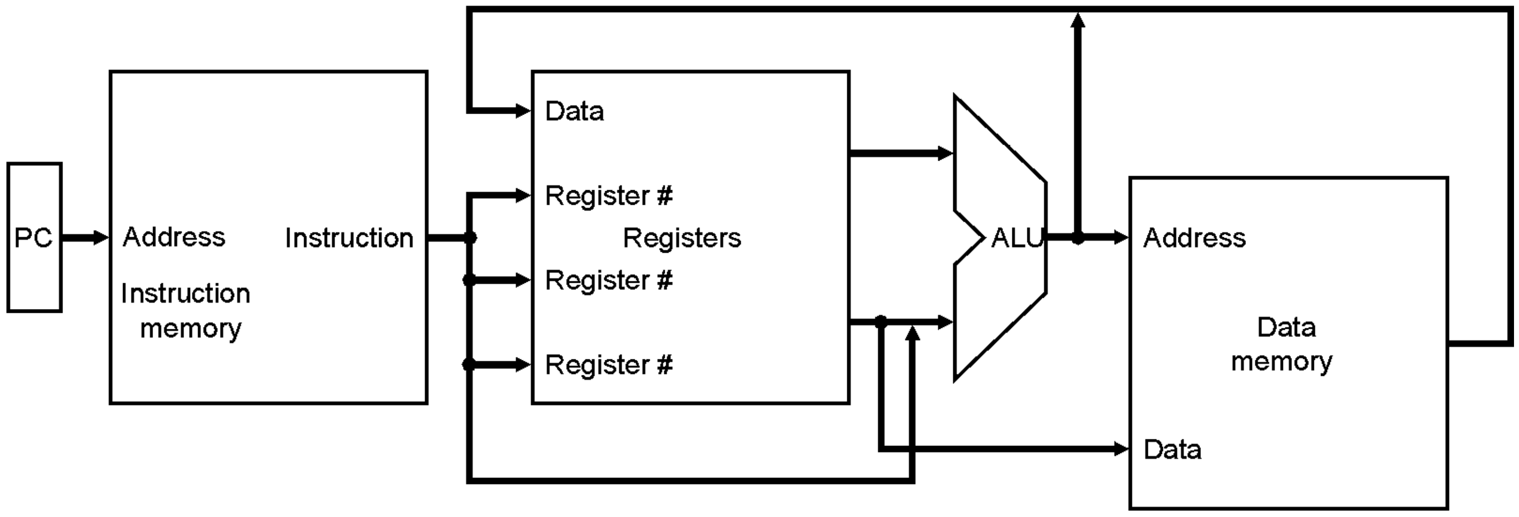
- <u>**BRANCH**</u>**:**
  - **beq rs, rt, imm**

31 26	21	16	0
op	**rs**	**rt**	**displacement**
6 bits	5 bits	5 bits	16 bits

1. Read registers rs and rt
2. Feed to ALU to compare
3. Add PC to disp; update PC

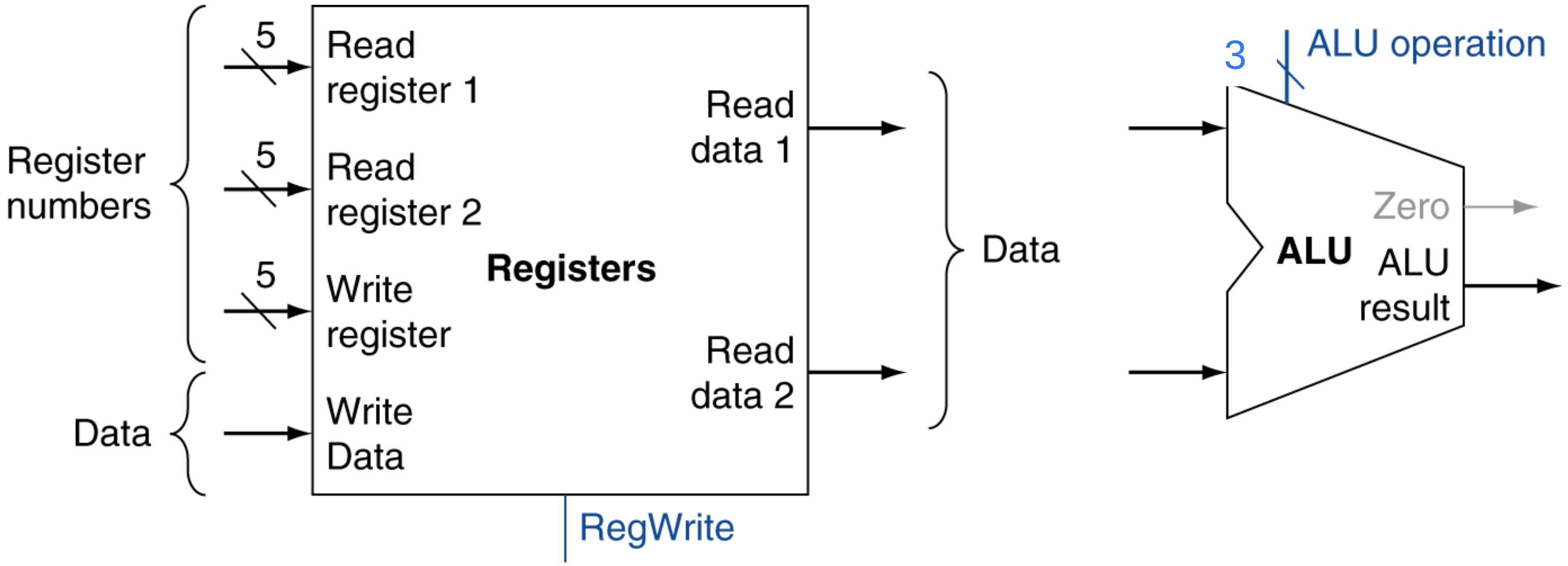
Processor Design

- **Generic Implementation:**
  - all instructions read some registers
  - all instructions use the ALU after reading registers
  - memory accessed & registers updated after ALU
- **Suggests basic design:**



R-Format Instructions

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

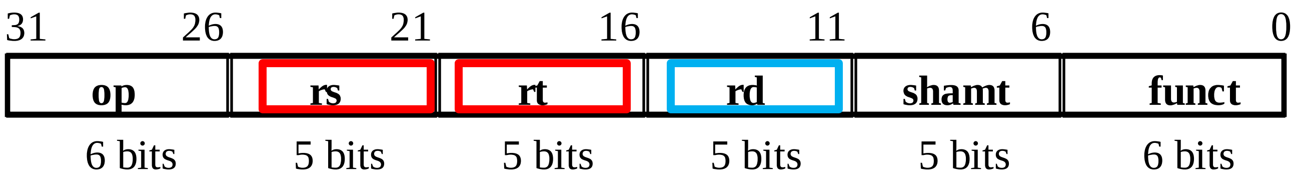


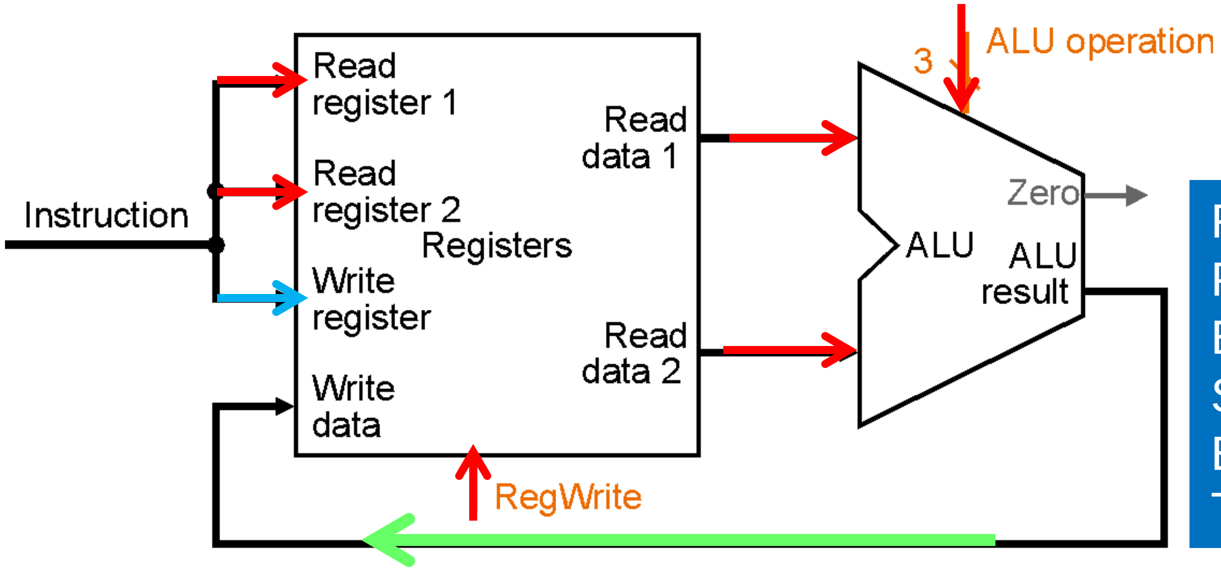
a. Registers

b. ALU

Datapath for Reg-Reg Operations

- R[rd] <- R[rs] op R[rt]   Example: *add rd, rs, rt*
  - ID's of three registers obtained from *rs, rt,* and *rd* fields of the instruction (two sources and one destination)
  - ALU-operation signal depends on *op* and *funct*

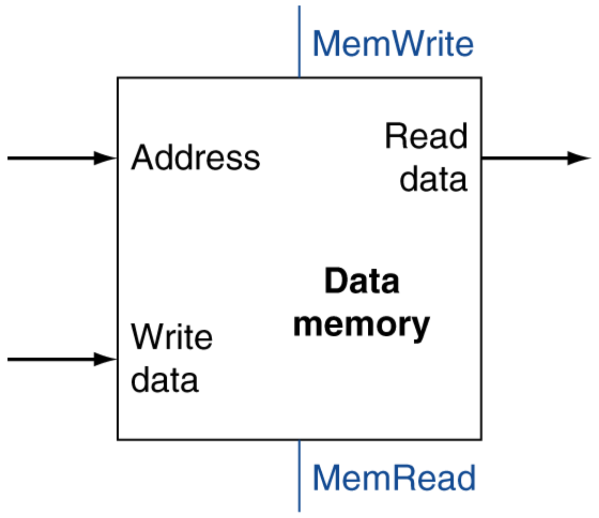




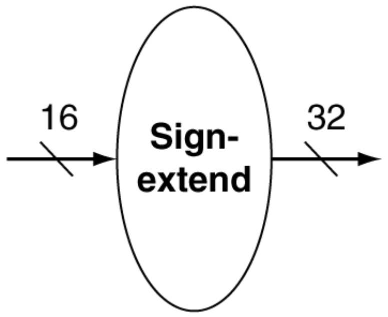
Read contents of *rs* and *rt;*
Pass the two operands to ALU;
Enable addition;
Select destination-register *rd;*
Enable RegWrite;
Transfer result from ALU to *rd;*

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



a. Data memory unit

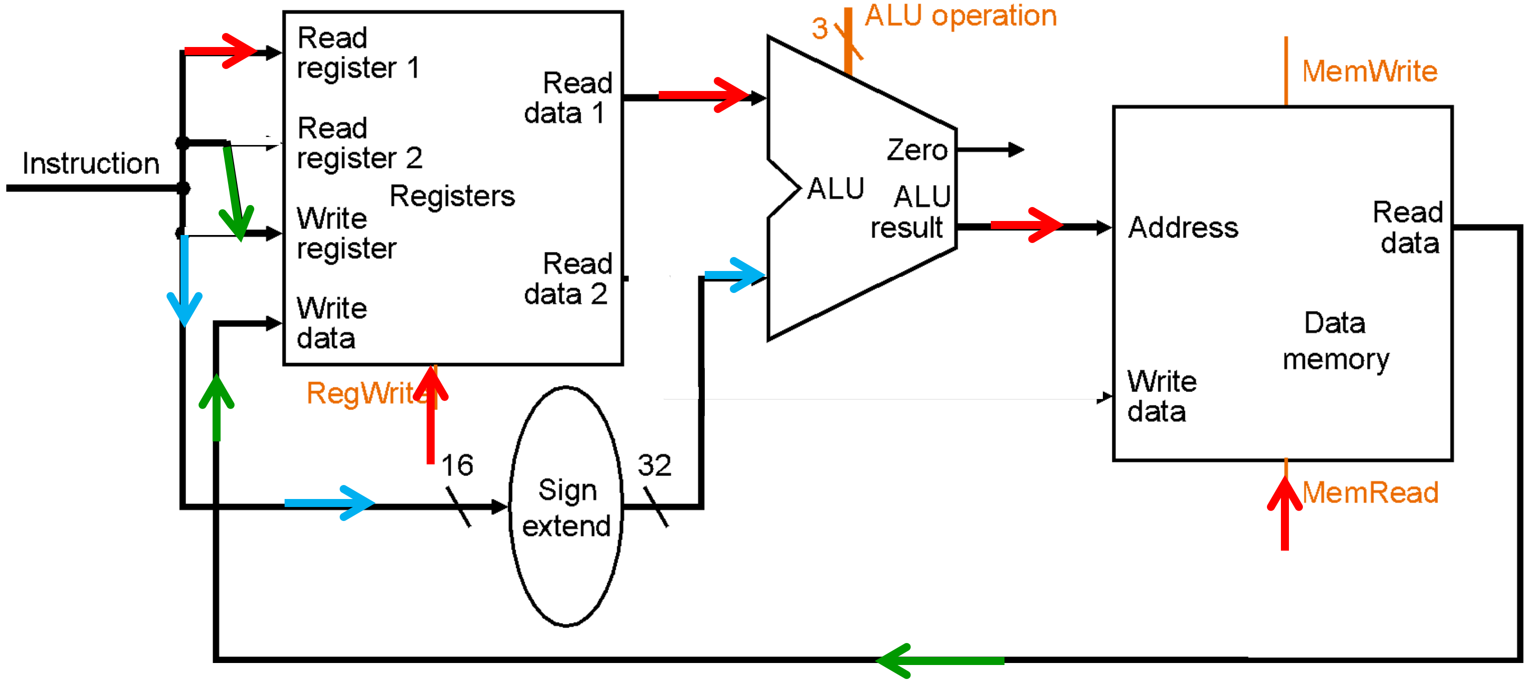


b. Sign extension unit

Datapath for Load Operations

R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: ***lw rt, rs, imm16***

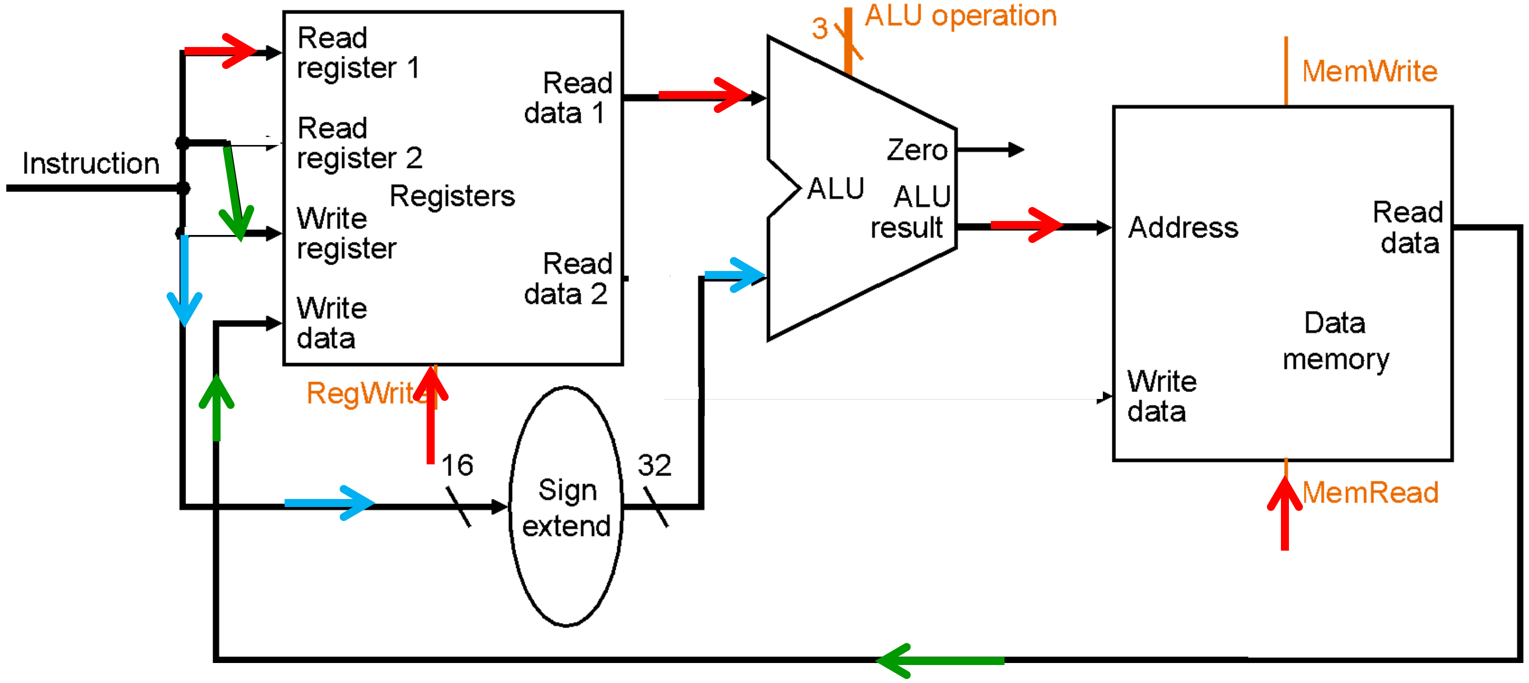
31 26	21	16	0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



Datapath for Load Operations

R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw rt, rs, imm16*

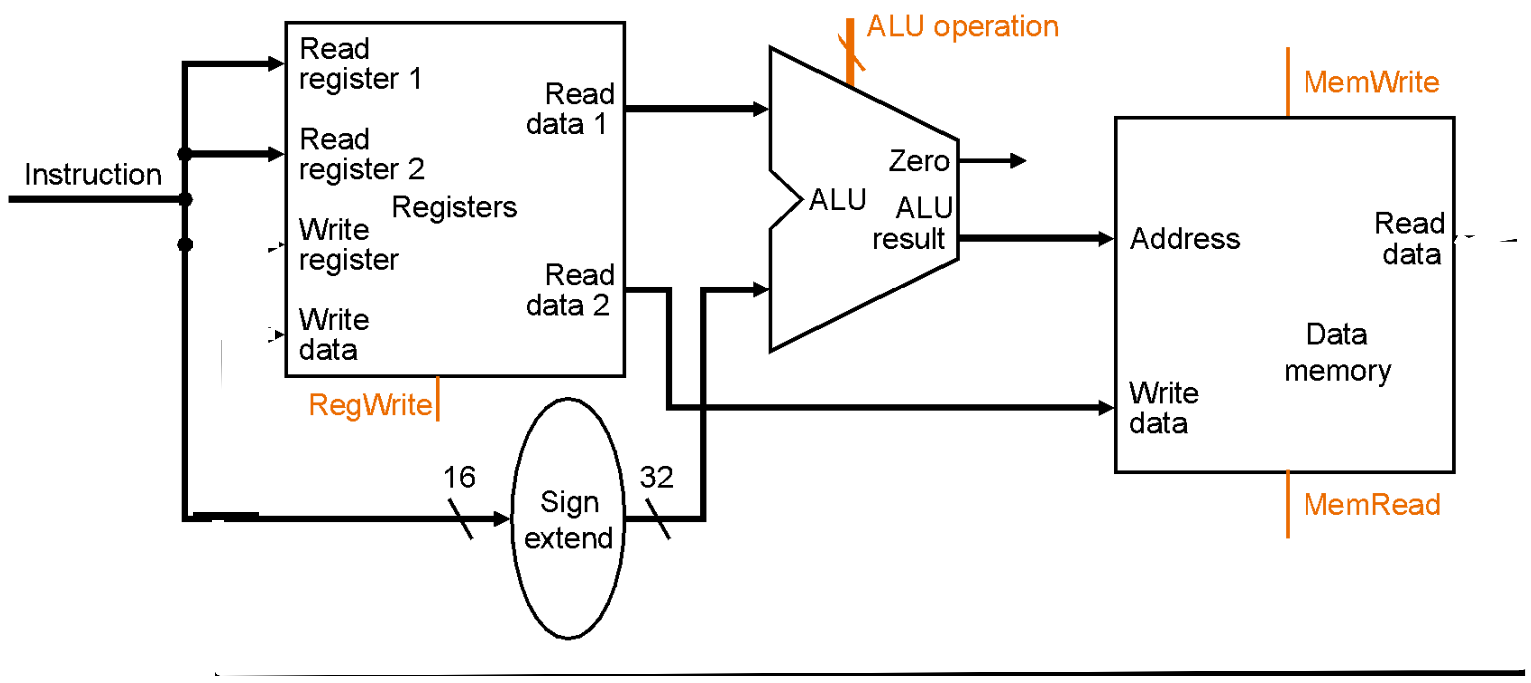
Read contents of *rs*
Pass (*rs*) and sign-ext (imm) to ALU;
Enable addition to compute address;
Select dest register *rt*
Enable MemRead
Read memory location pointed by the computed address;
Transfer data from Mem to *rt;*



Datapath for Store Operations

Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw rt, rs, imm16*

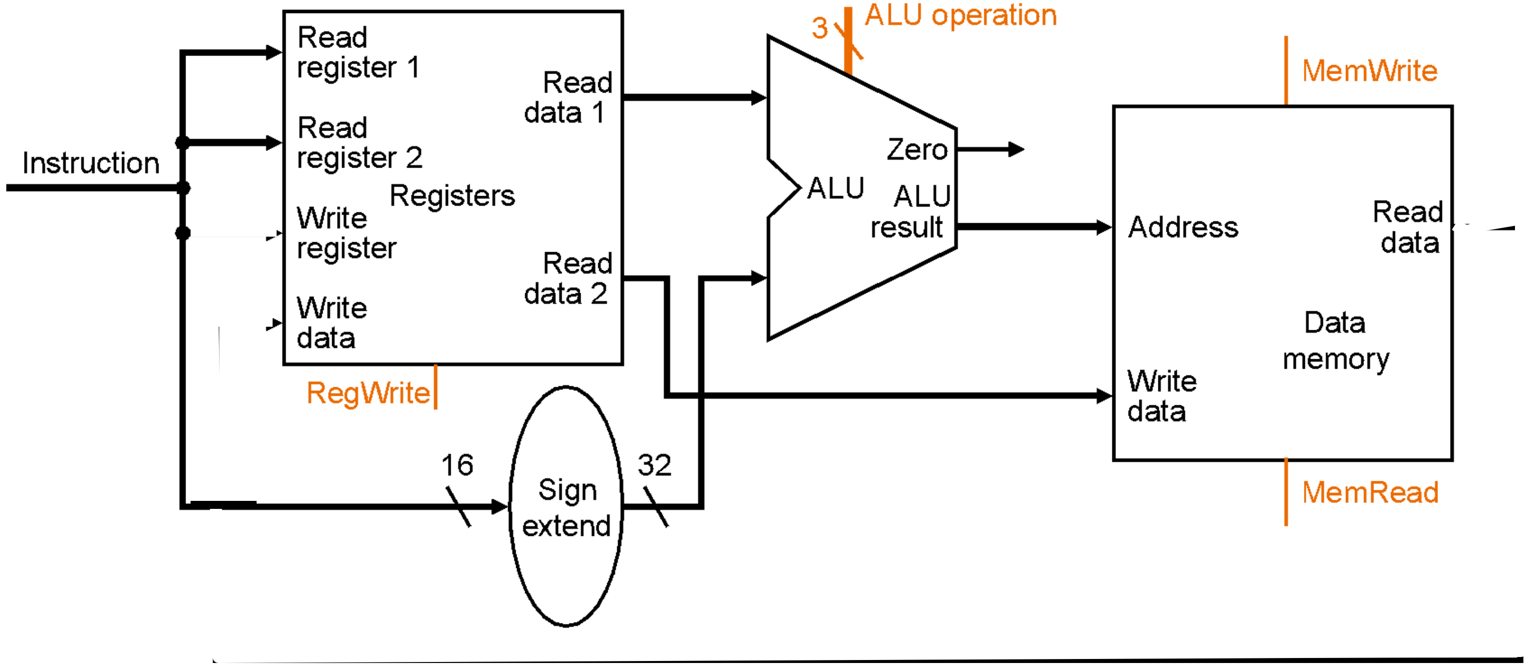
31 … 26	25 … 21	20 … 16	15 … 0
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



Datapath for Store Operations

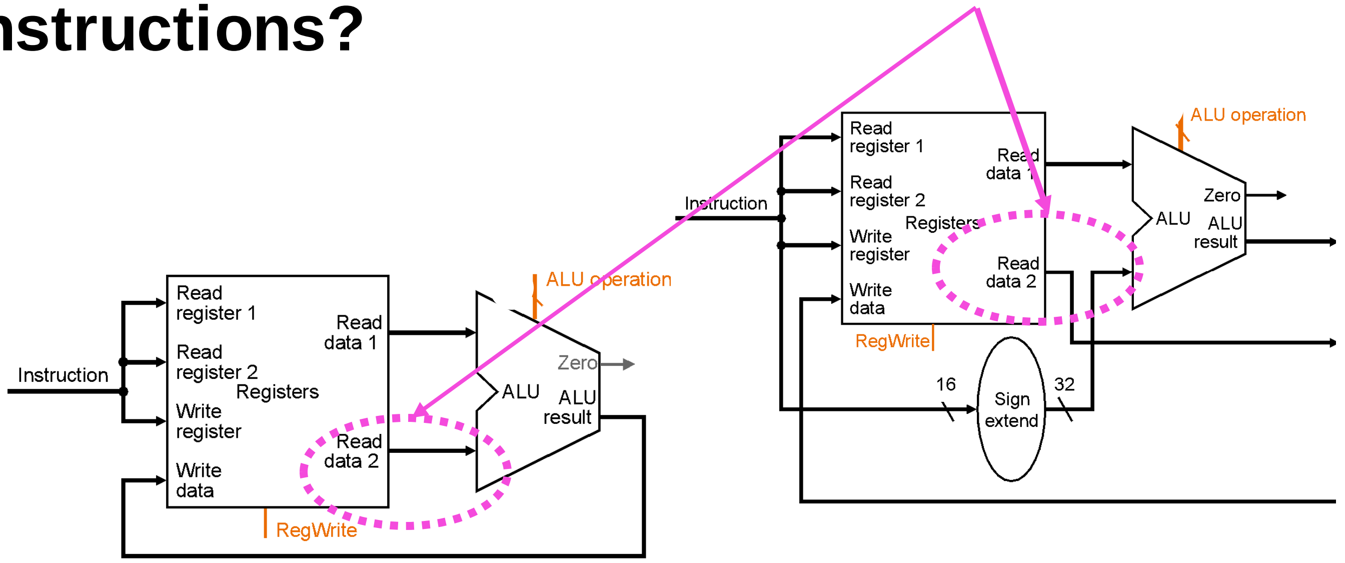
Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

Read contents of *rs*
Pass (*rs*) and sign-ext (imm) to ALU;
Enable addition to compute address;
Select and read dest register *rt*
Enable MemWrite
Transfer (*rt*) to Mem location pointed by the computed address



Combining datapaths

- **How do we allow different datapaths for different instructions?**

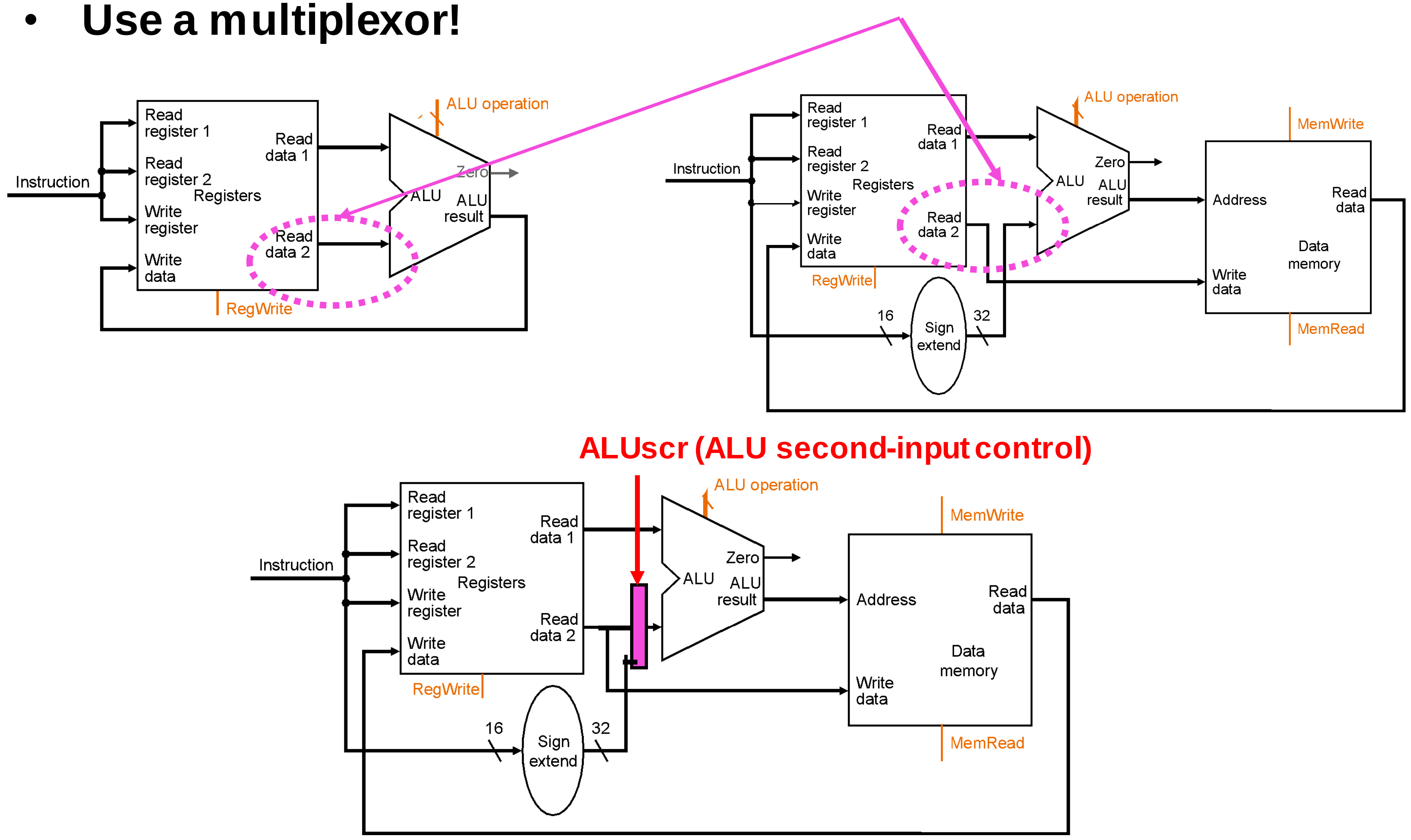


R-type

Store

Combining datapaths

- How do we allow different datapaths for different instructions?
- Use a multiplexor!



Branch Instructions

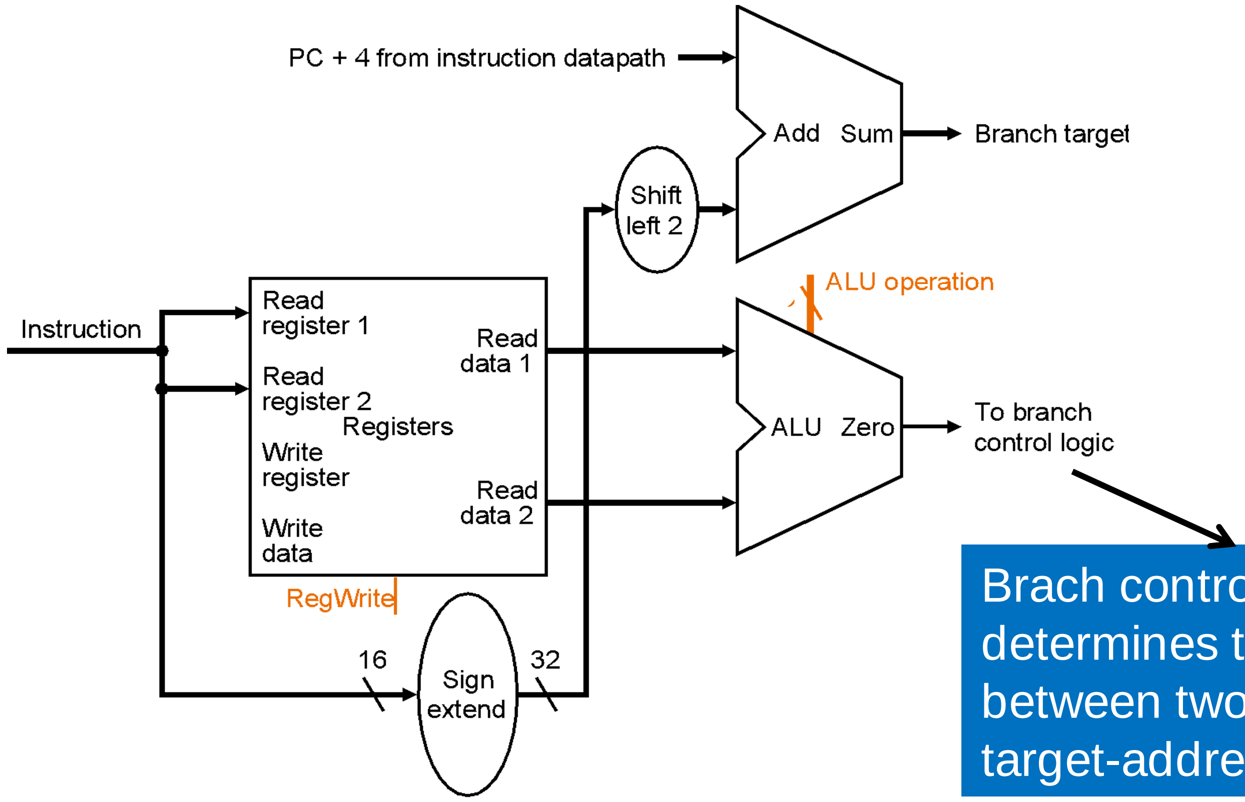
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch

Datapath for Branch Operations,,

beq rs, rt, imm16 **We need to compare *rs* and *rt***

31 26 21 16 0

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits



Brach control logic determines the choice between two possible target-addresses

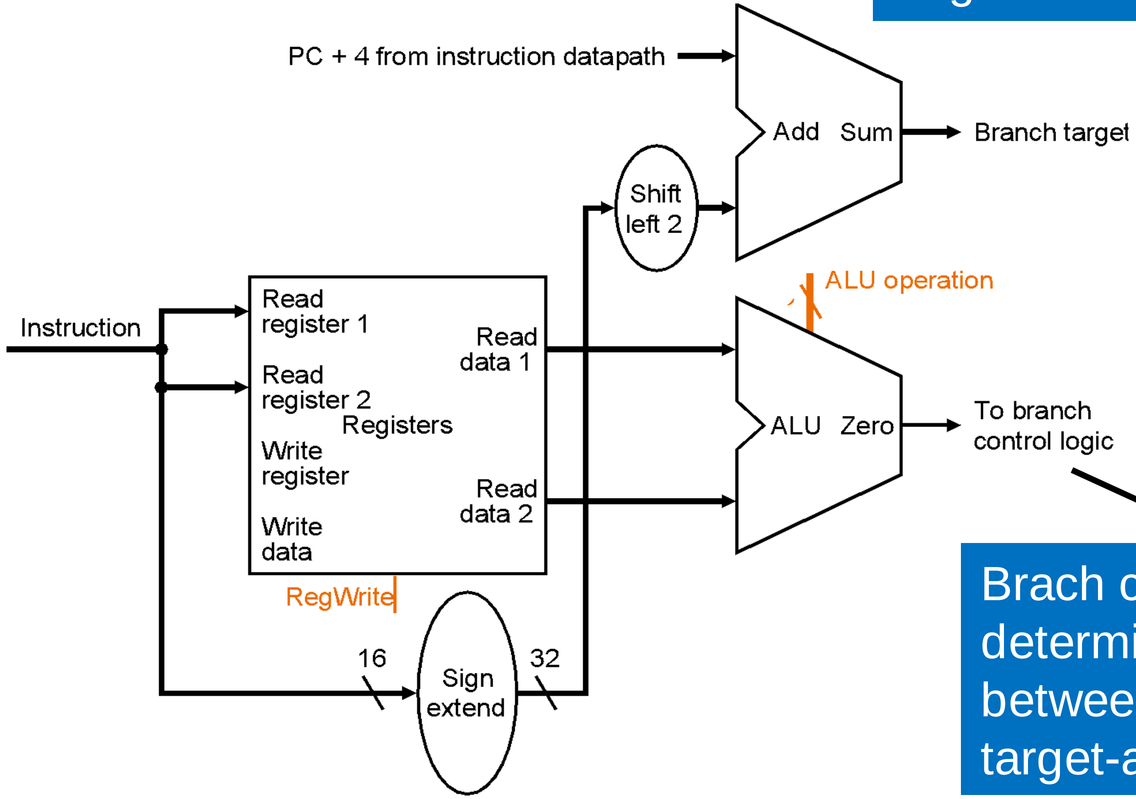
Datapath for Branch Operations

beq rs, rt, imm16

We need to

Read contents of *rs* and *rt;*
Compare them in ALU, by enabling ALU-op;
Read *imm* and sign-extend;
Shift-left by 2-bits to compute word-offset;
Add it with (PC+4) to compute target ddress;

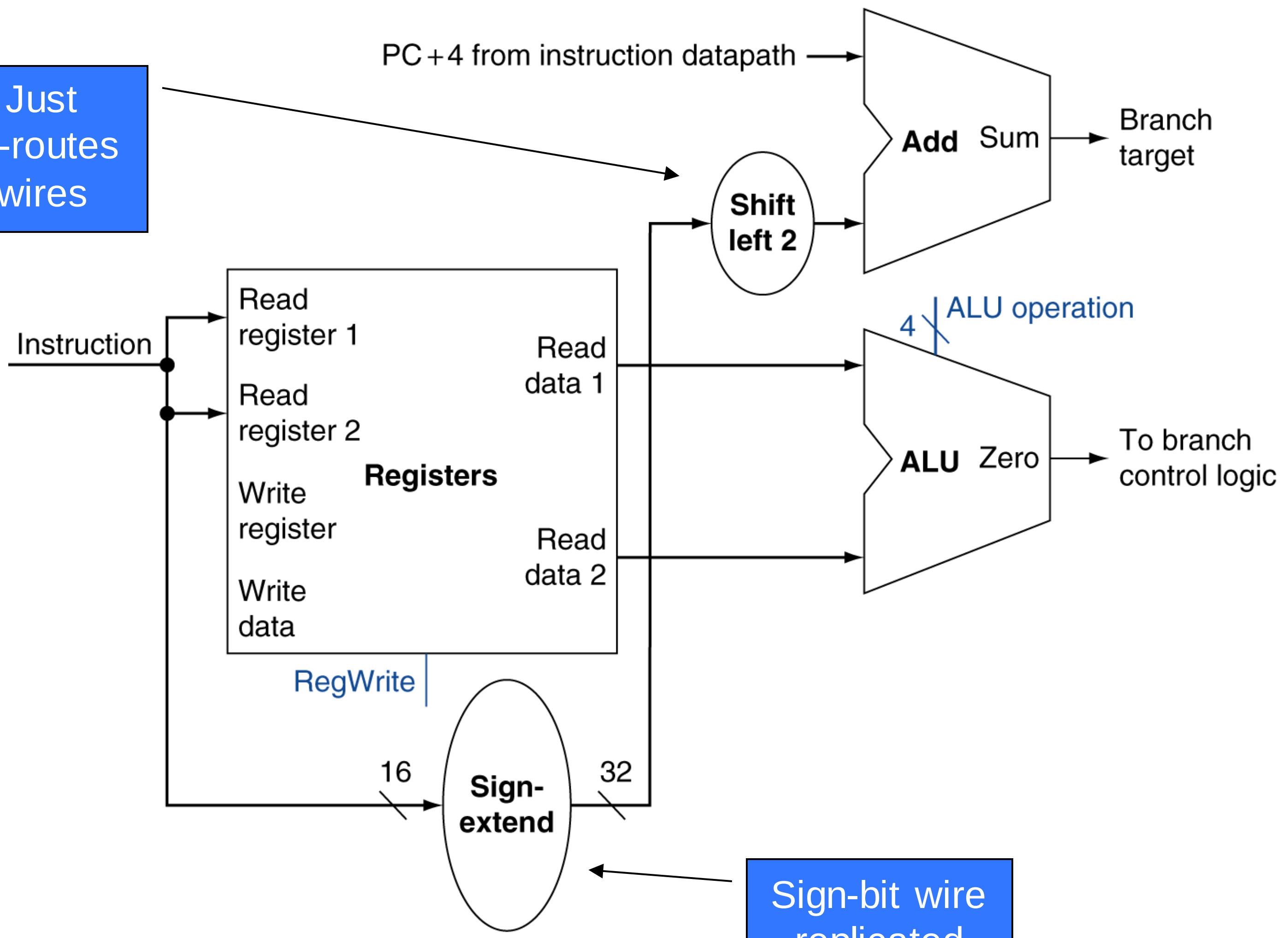
31	26	21	16
op	**rs**	**rt**	**immediate**
6 bits	5 bits	5 bits	16 bits



Brach control logic determines the choice between two possible target-addresses

Branch Instructions

Just re-routes wires

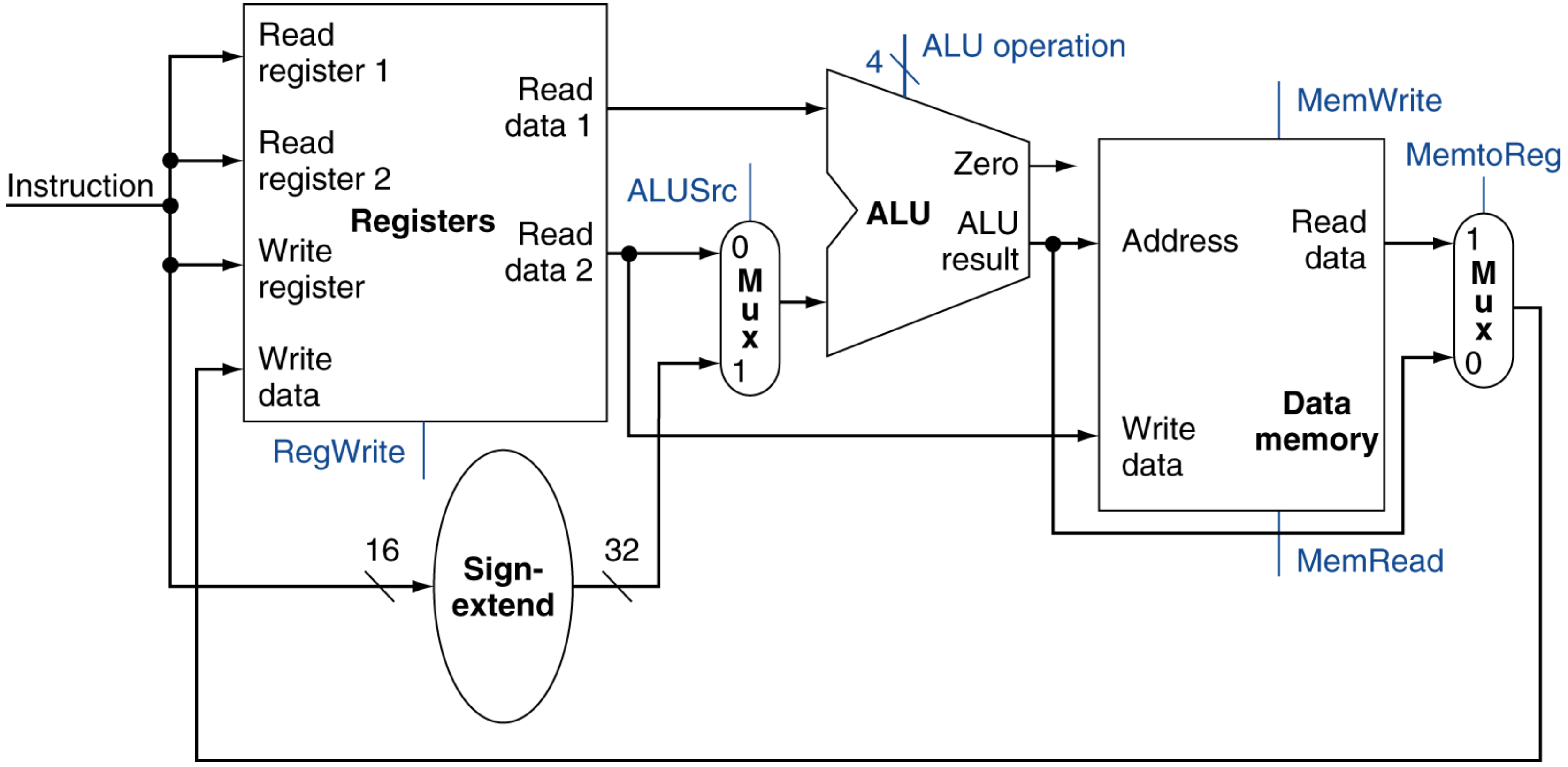


Sign-bit wire replicated

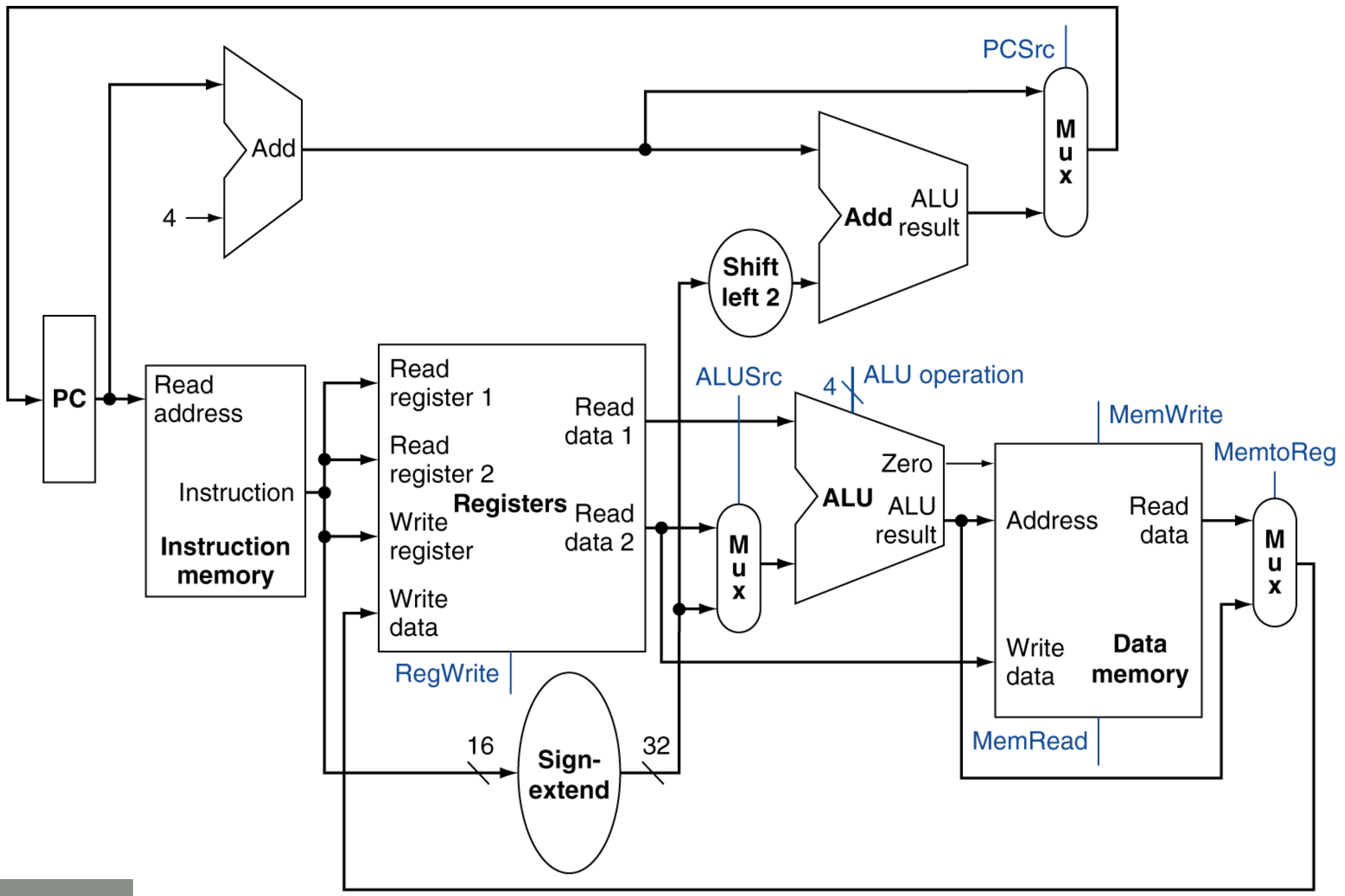
Composing the Elements

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

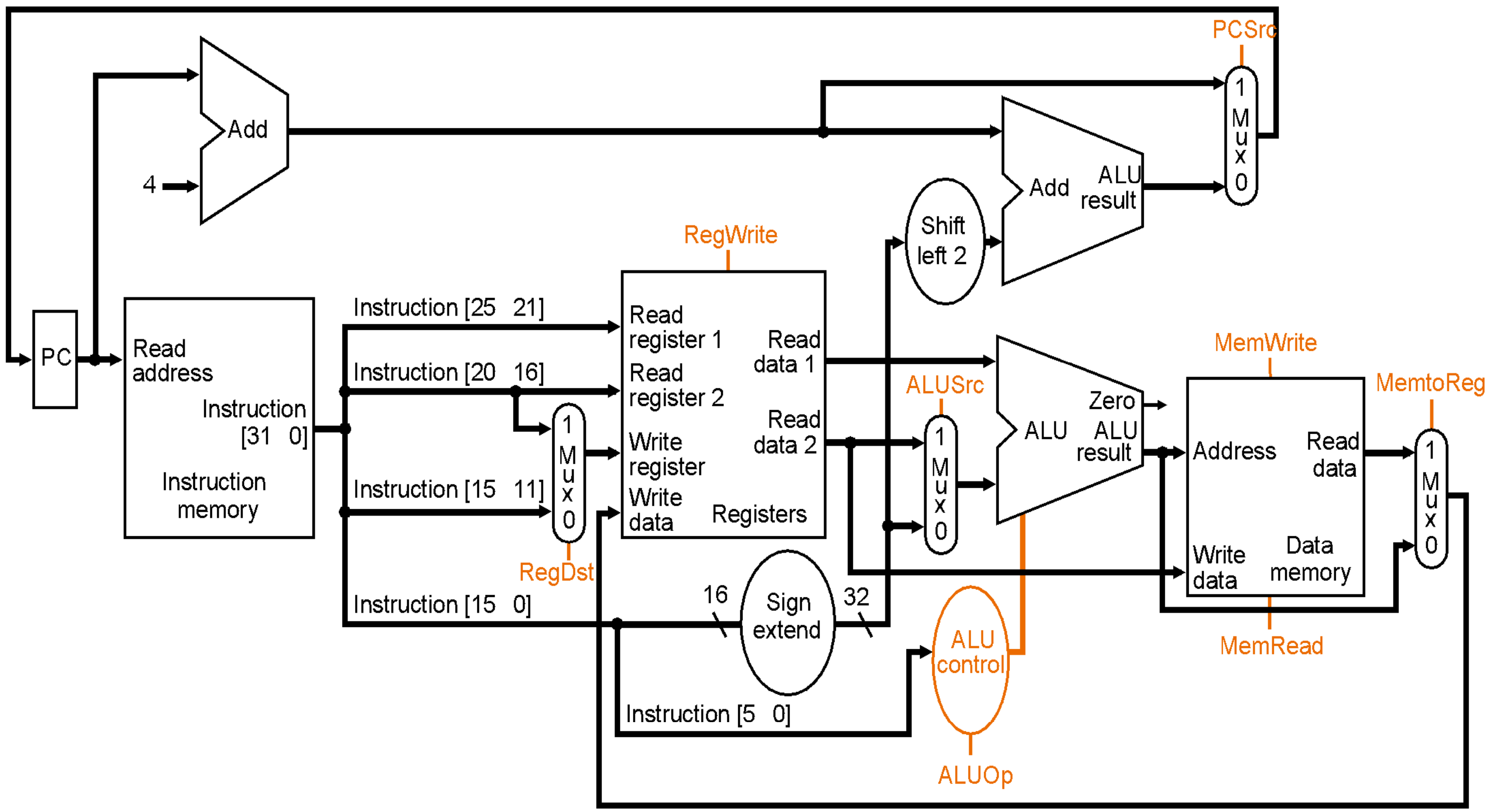
R-Type/Load/Store Datapath



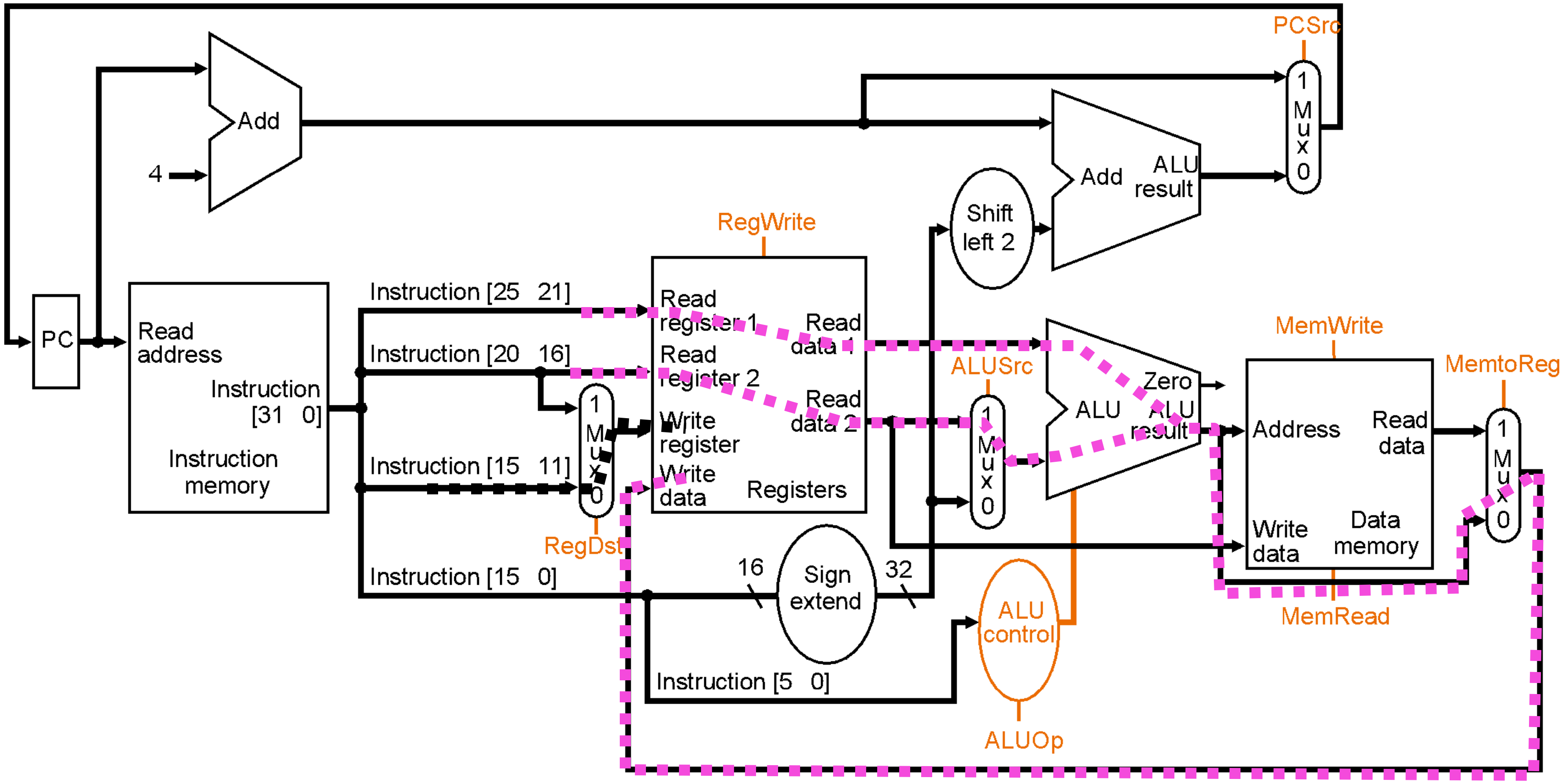
Full Datapath



All together: the single cycle datapath

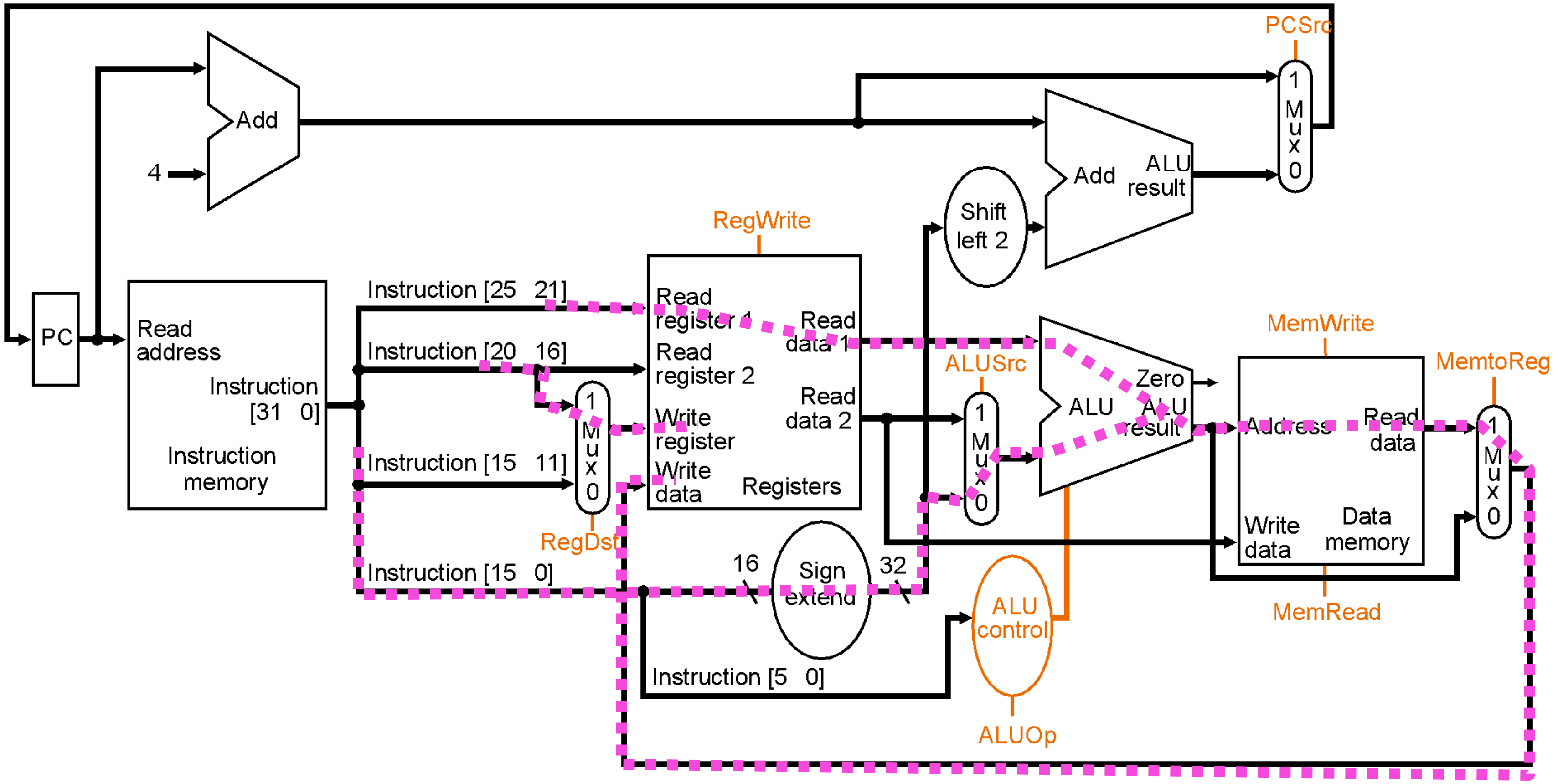


The R-Format (e.g. *add*) Datapath



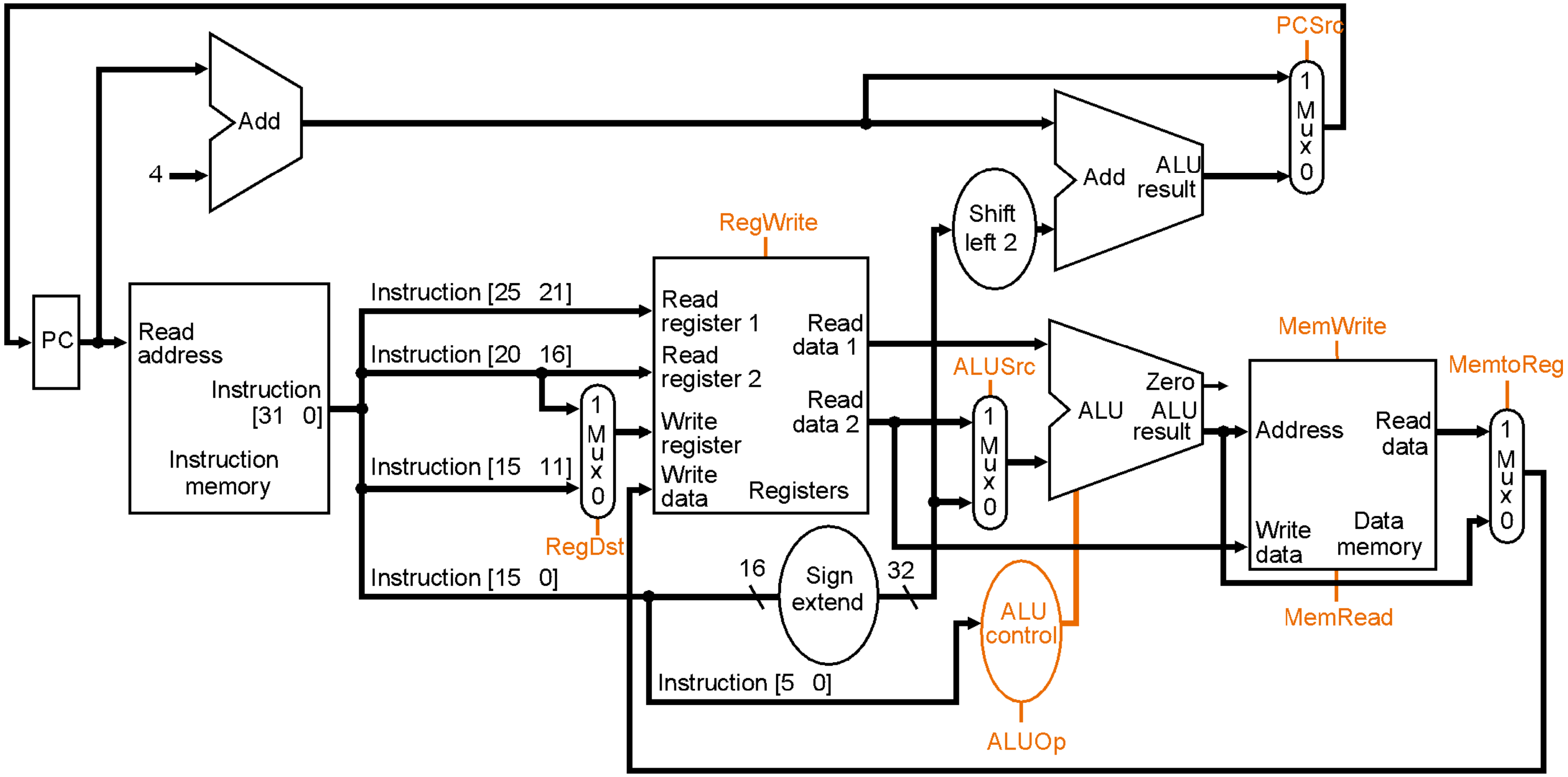
Need ALUsrc=1, ALUop="add", MemWrite=0, MemToReg=0, RegDst = 0, RegWrite=1 and PCsrc=1.

The Load Datapath

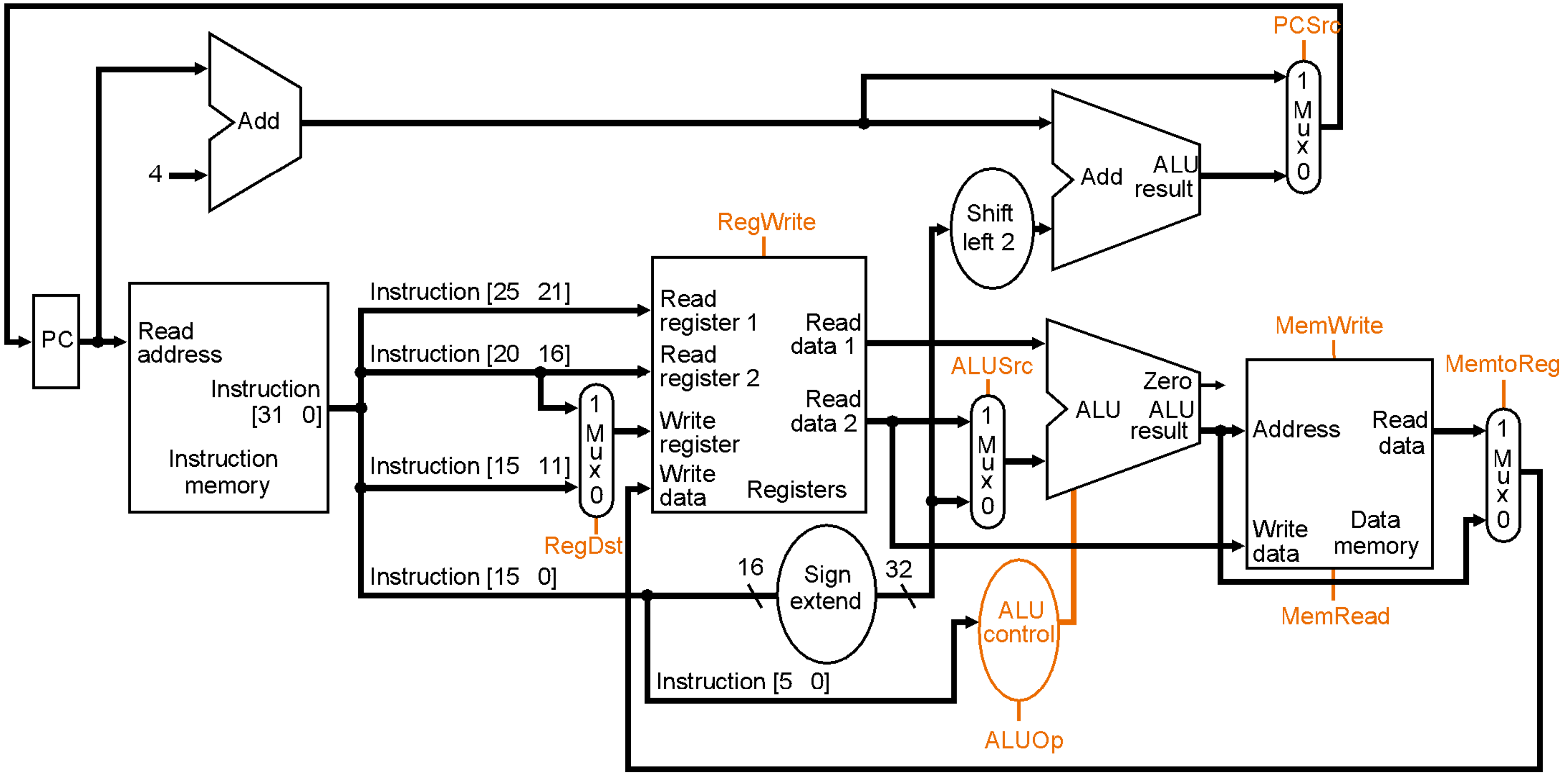


What control signals do we need for load??

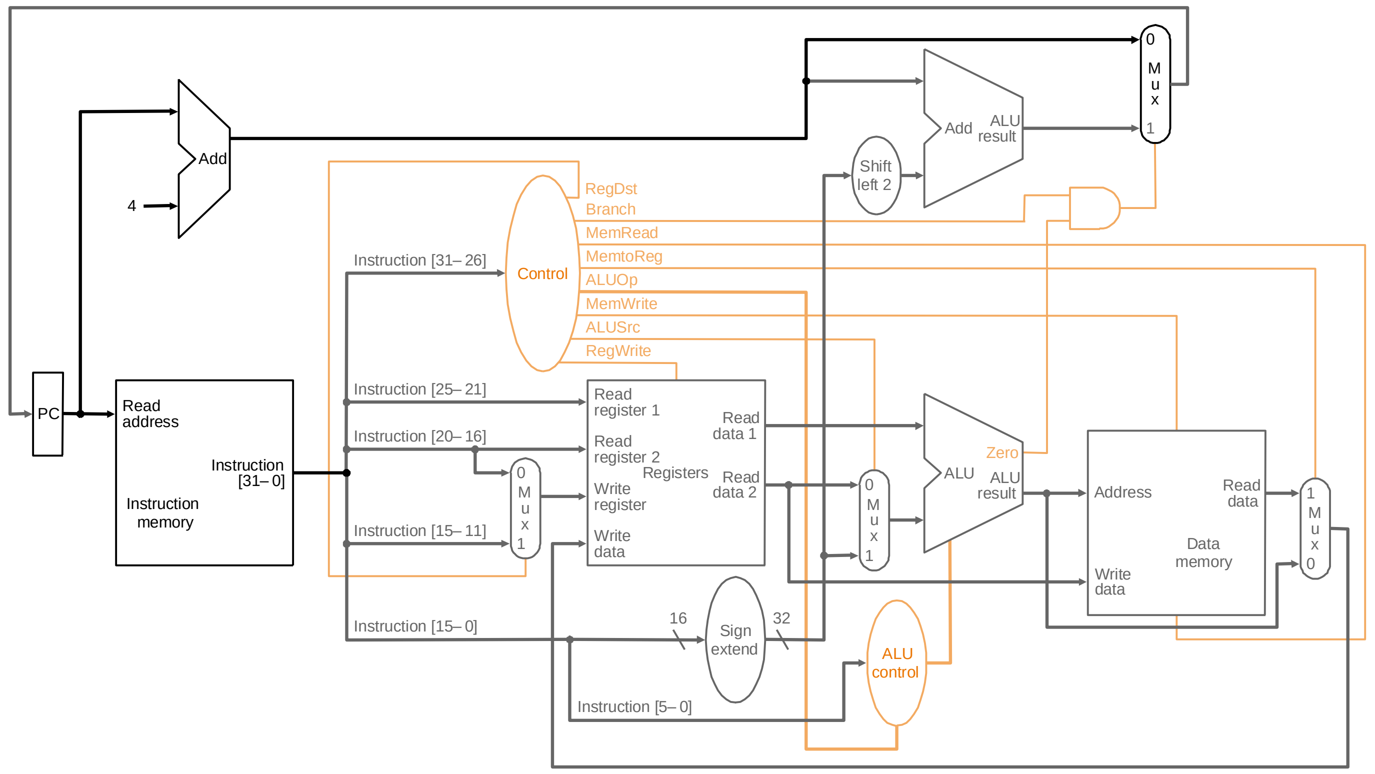
The Store Datapath



The beq Datapath



Control Signal Requirement



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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

How to design the control unit?

- Input:

  --- Op-code and function-field bits from instructions

- Output:

  --- Desired control signals needed during the execution of an instruction

- For single-cycle implementation, the controller can be designed by a combinational circuit

Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

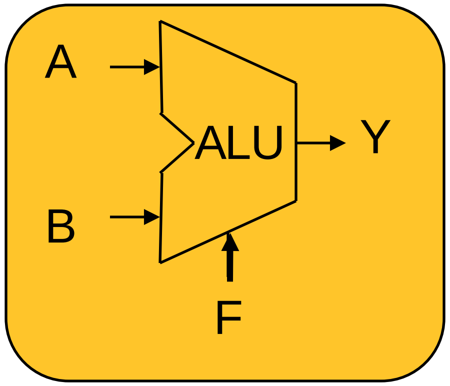
add $8, $17, $18 **Instruction Format:**

`000000`	`10001`	`10010`	`01000`	`00000`	`100000`
`op`	**`rs`**	**`rt`**	**`rd`**	**`shamt`**	**`funct`**

- ALU's operation based on instruction type and function code

ALU Control

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



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

ALU Control

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

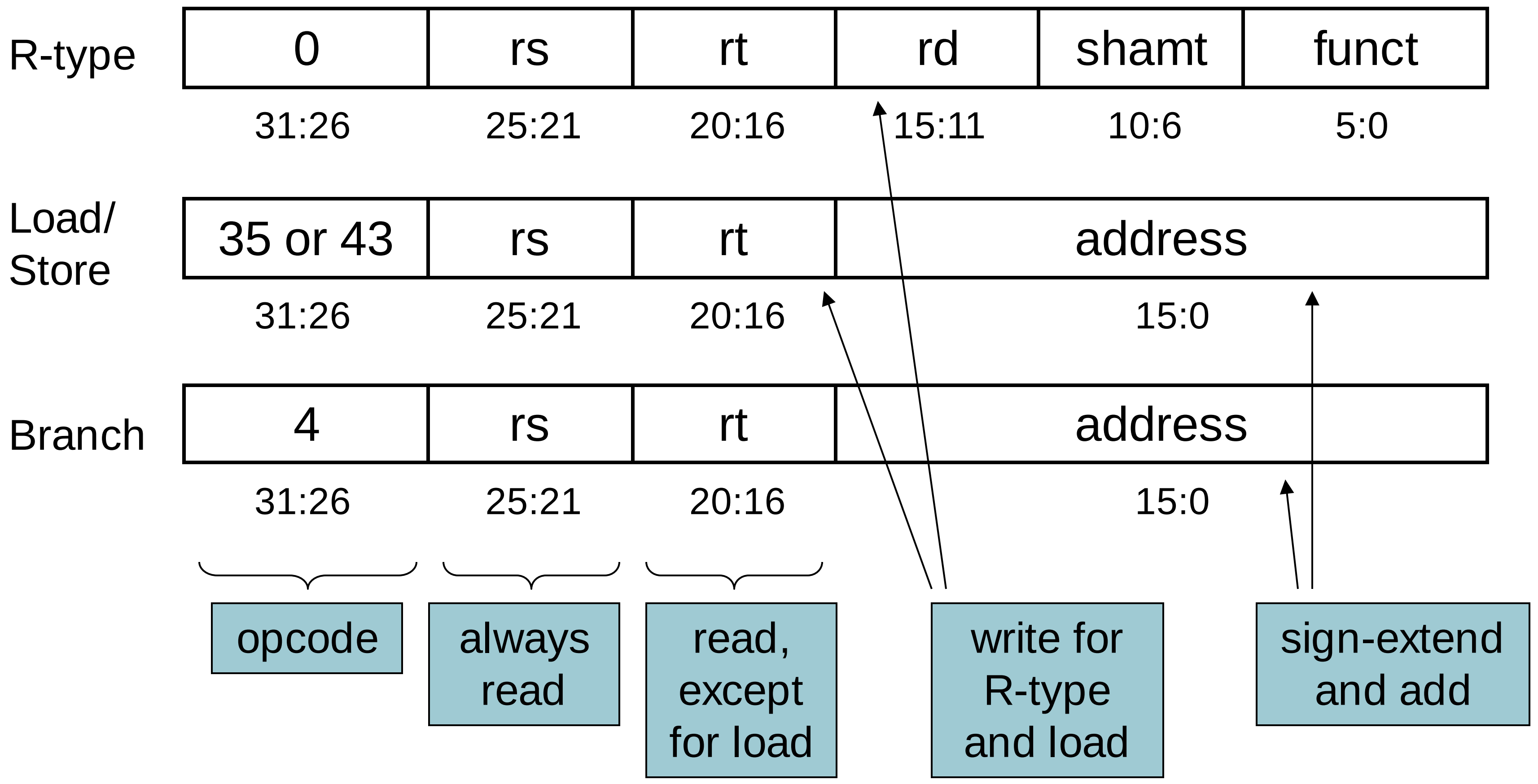
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than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

Inputs to control logic

outputs of control logic

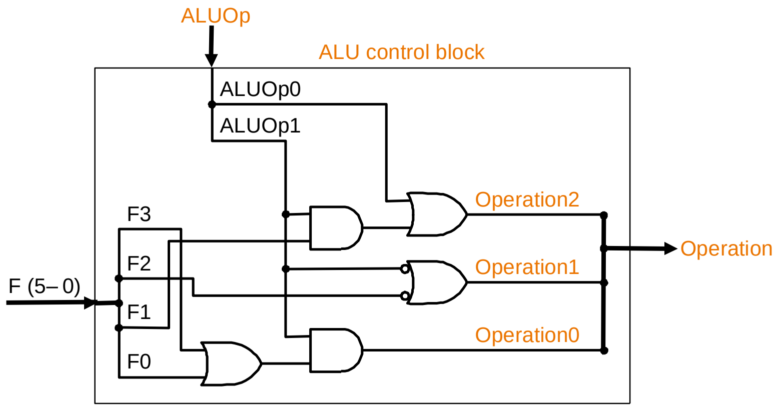
The Main Control Unit

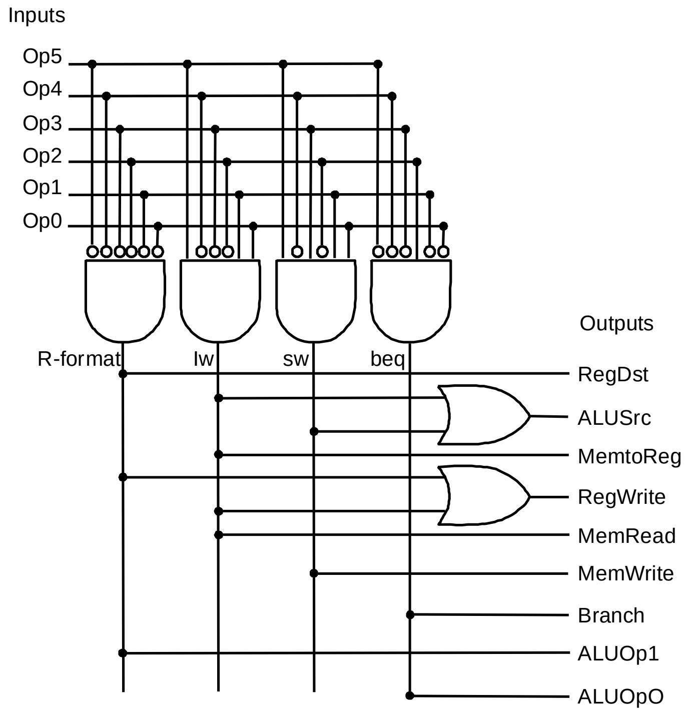
- Control signals derived from instruction



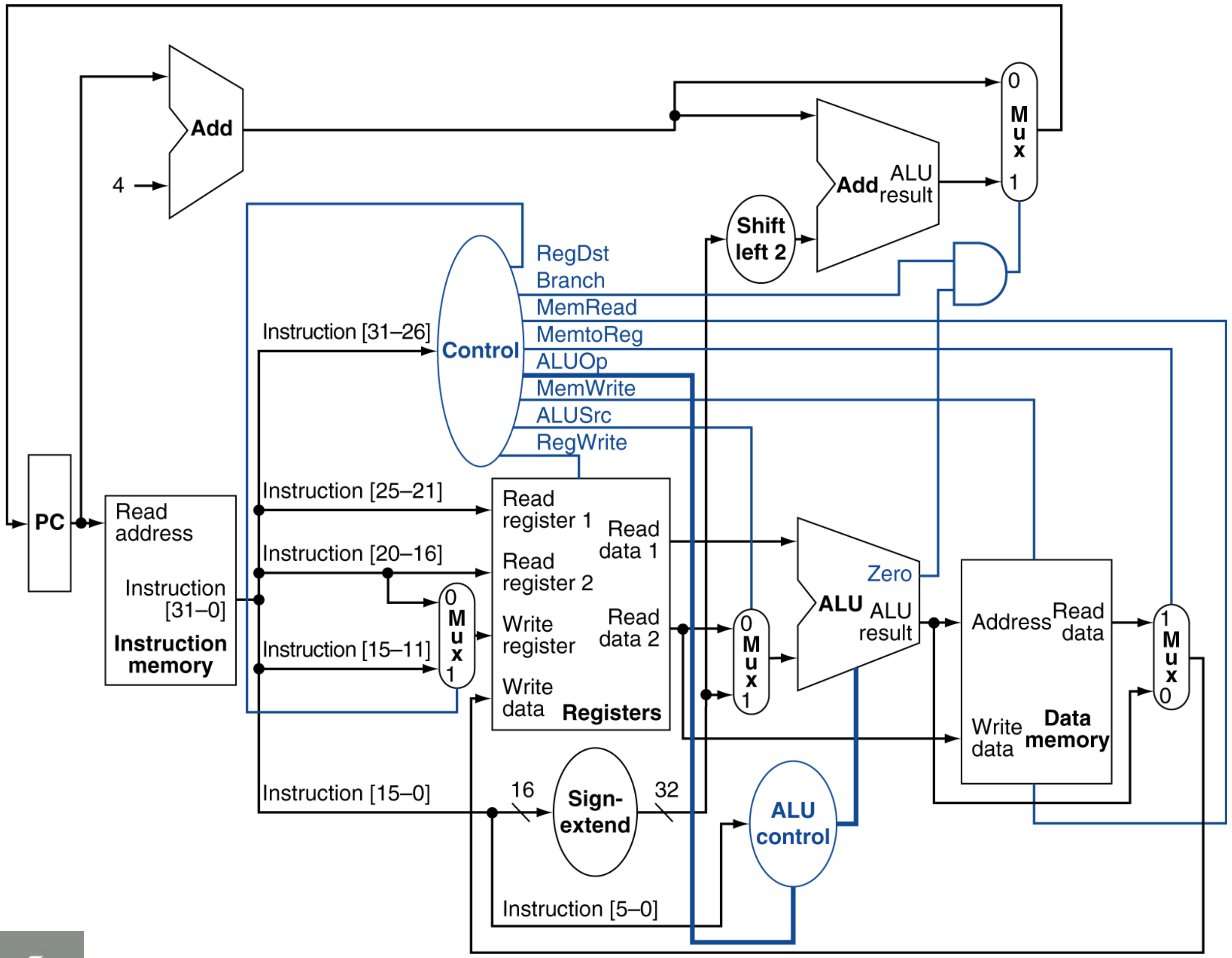
Control

- **Simple combinational logic needed**

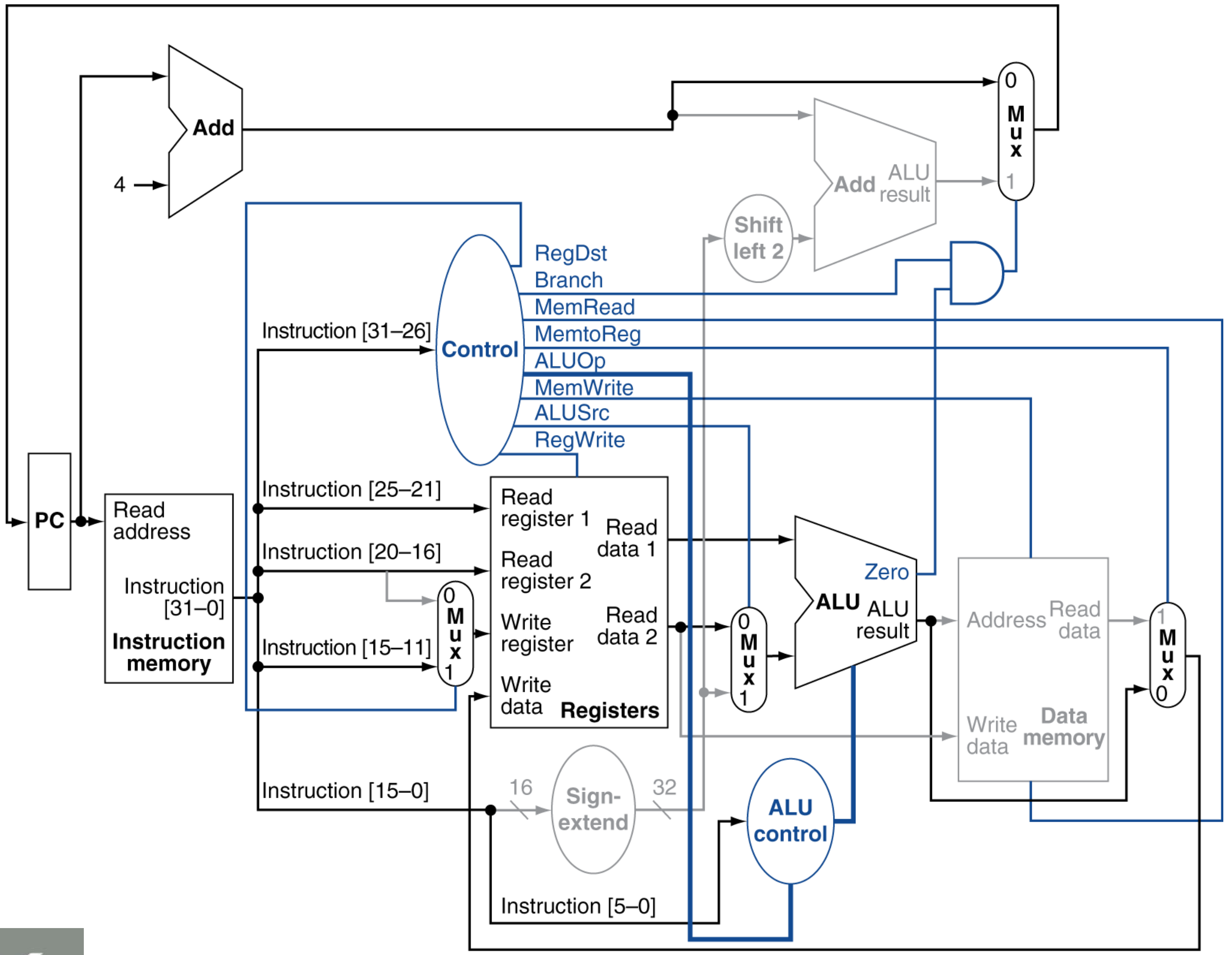




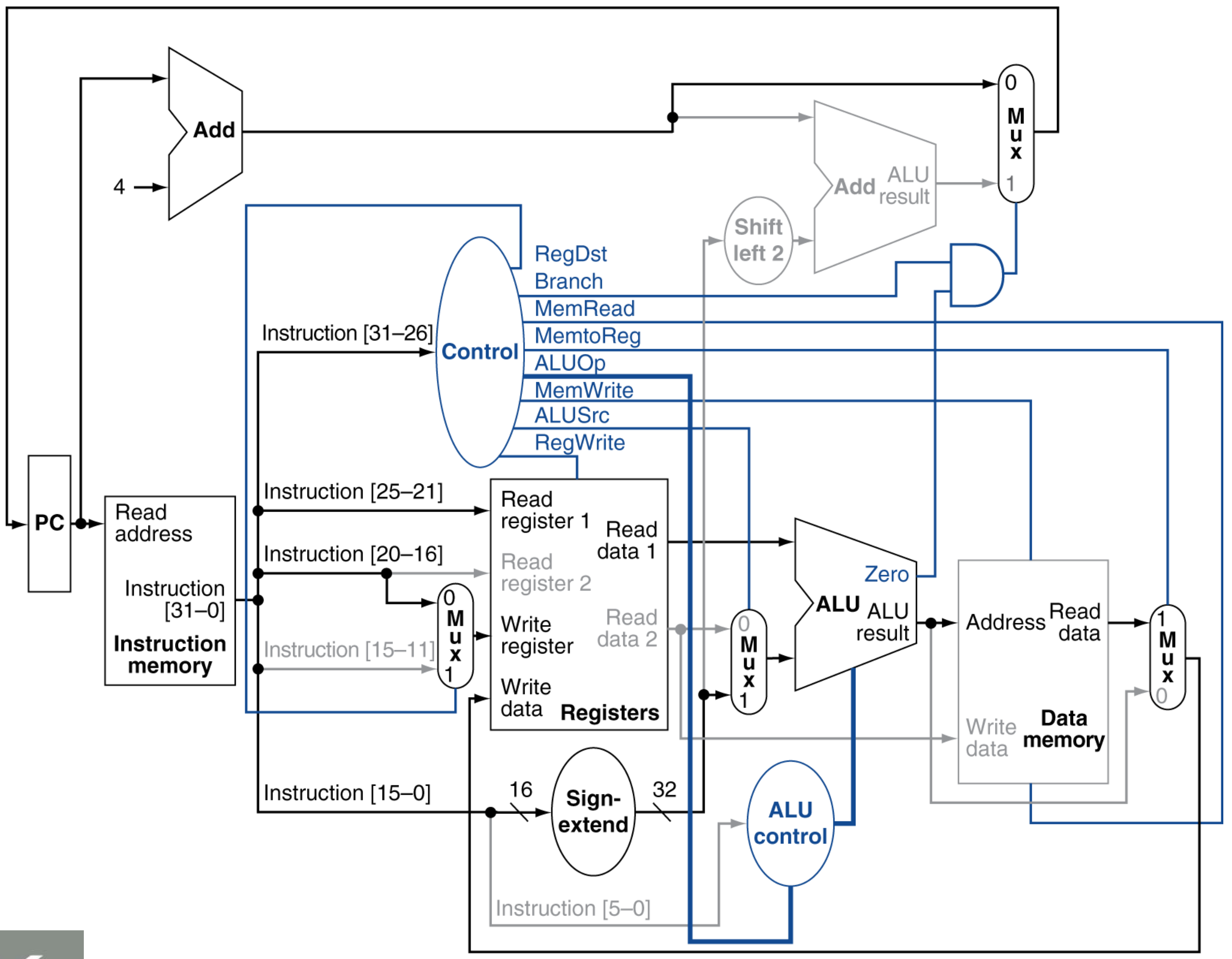
Datapath With Control



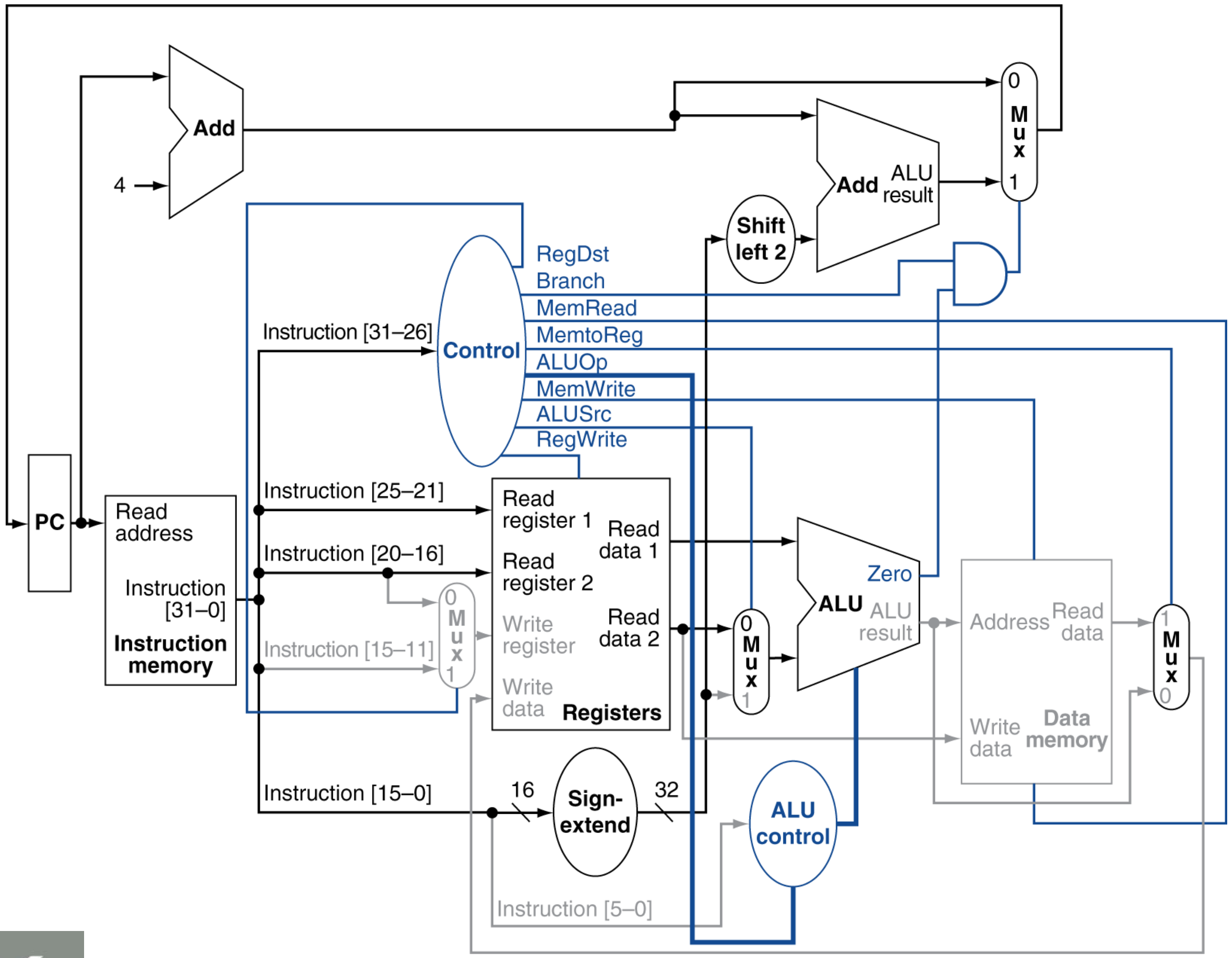
R-Type Instruction



Load Instruction



Branch-on-Equal Instruction

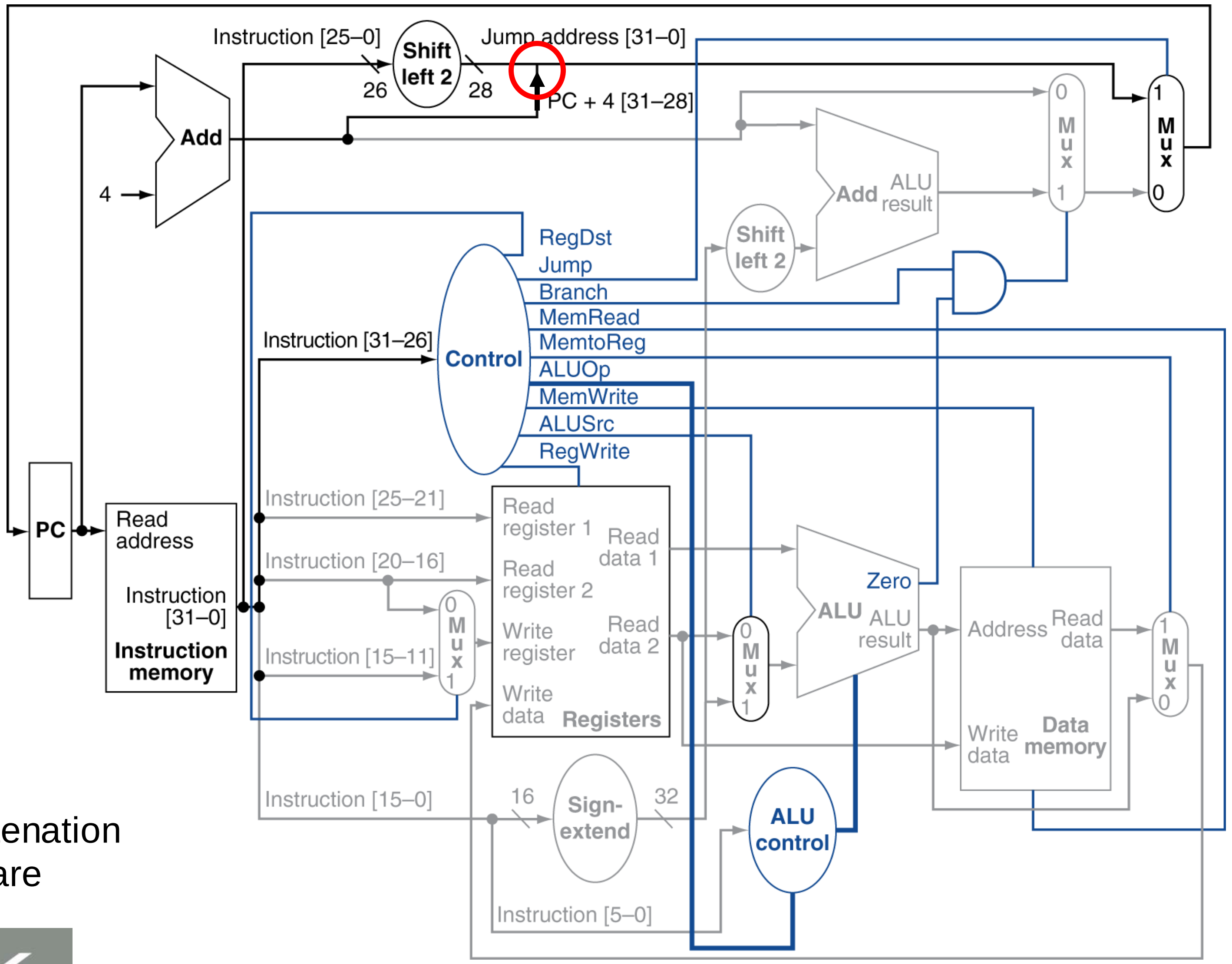


Implementing Jumps

Jump	2	address
	31:26	25:0

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - Least significant two bits: 00
- Need an extra control signal decoded from op-code

Datapath with “Jumps” Added



○ concatenation hardware

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

Key Points

- CPU is just a collection of state and combinational logic
- We have just designed a processor, at least in terms of functionality
- CPU time = IC * CPI * CCT
  - single-cycle machine degrades performance
  - Needs improved techniques such as pipelining

COA Tutorial – Processor Design

Lecture #29

30 September 2021

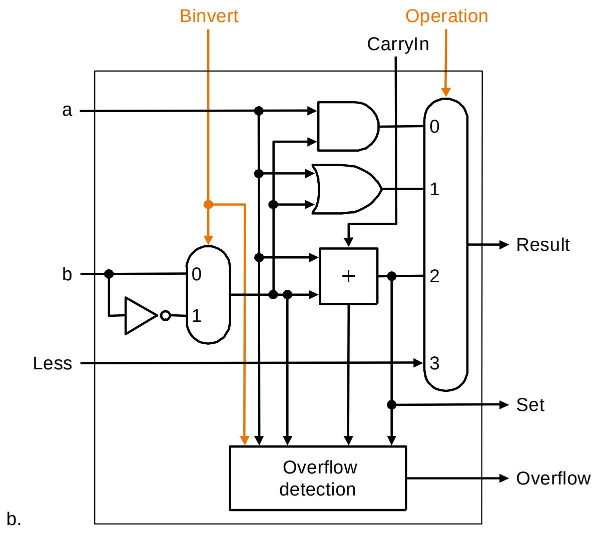
Processor Design

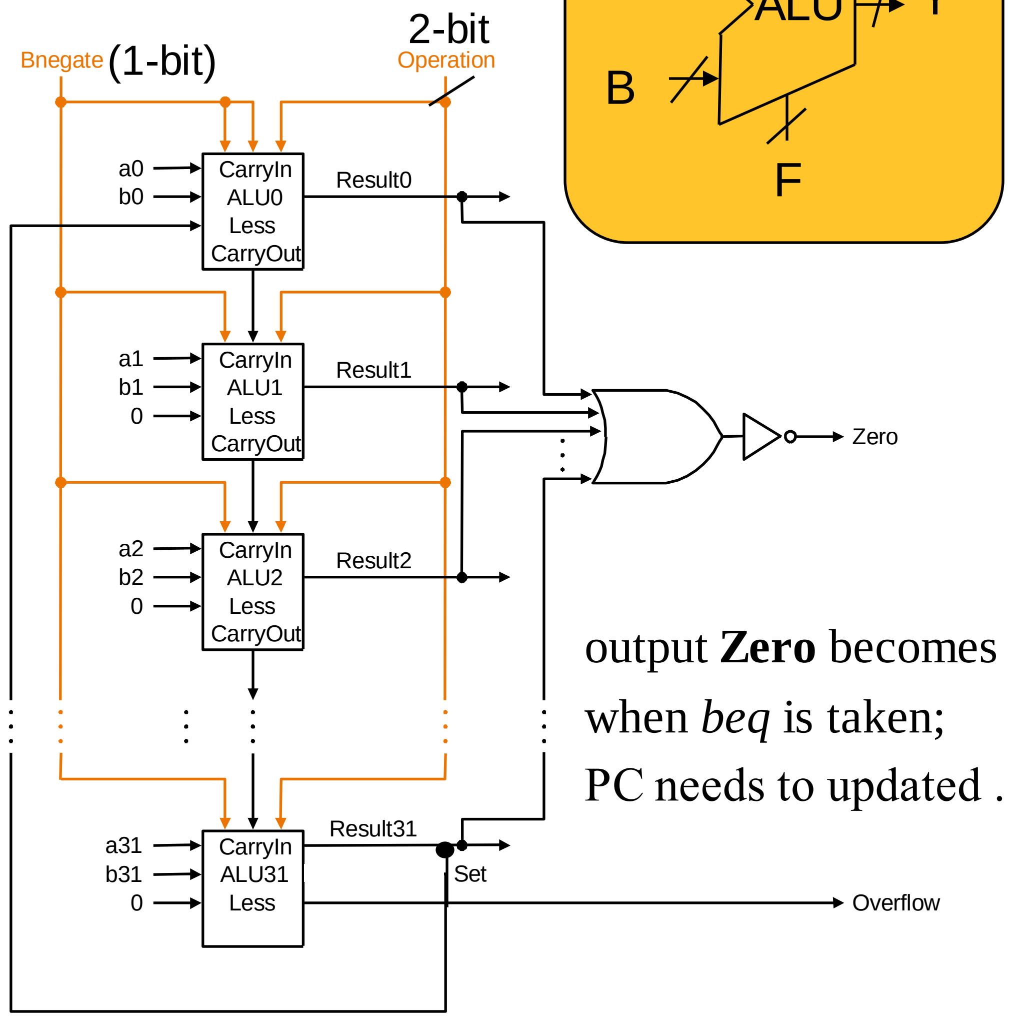
- Goal: To implement the MIPS "core" processor
  - load-store instructions: `lw, sw`
  - reg-reg instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq`

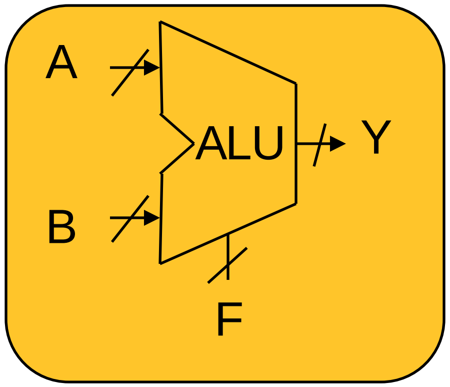
Implementing ALU

Control lines (Bnegate Op1 Op0):

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```





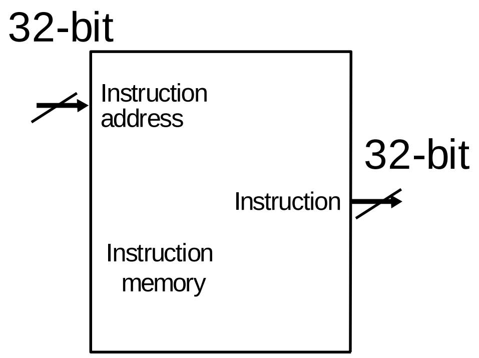


output **Zero** becomes 1 when *beq* is taken; PC needs to updated …

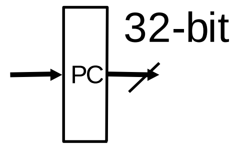
This ALU implements MIPS *add*, *sub*, *and*, *or*, *slt*, and *beq*

Simple Hardware Implementation of a CPU

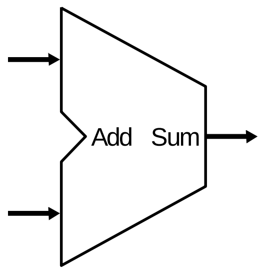
- Functional units we need for implementing instructions



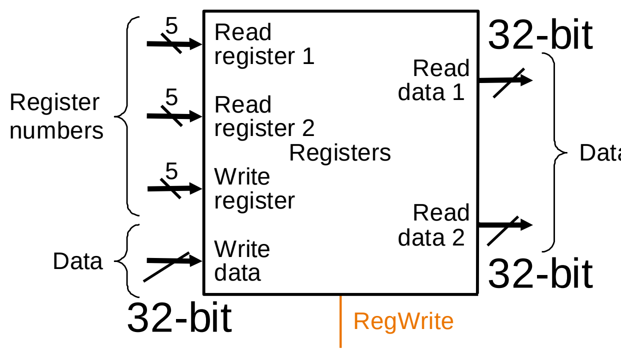
a. Instruction memory



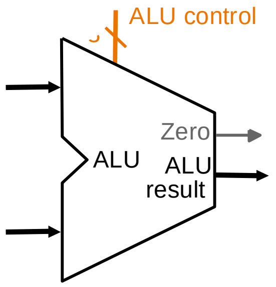
b. Program counter



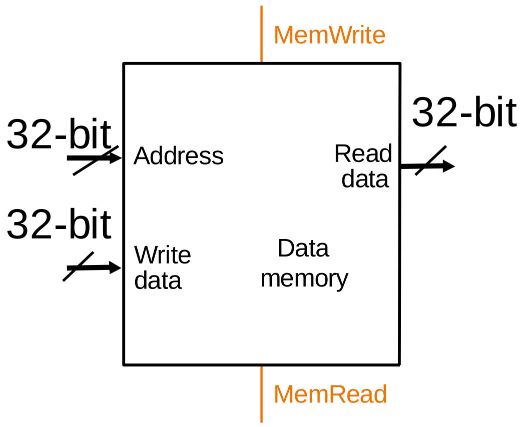
c. Adder



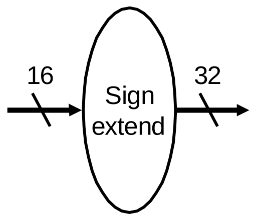
a. Registers



b. ALU



a. Data memory unit



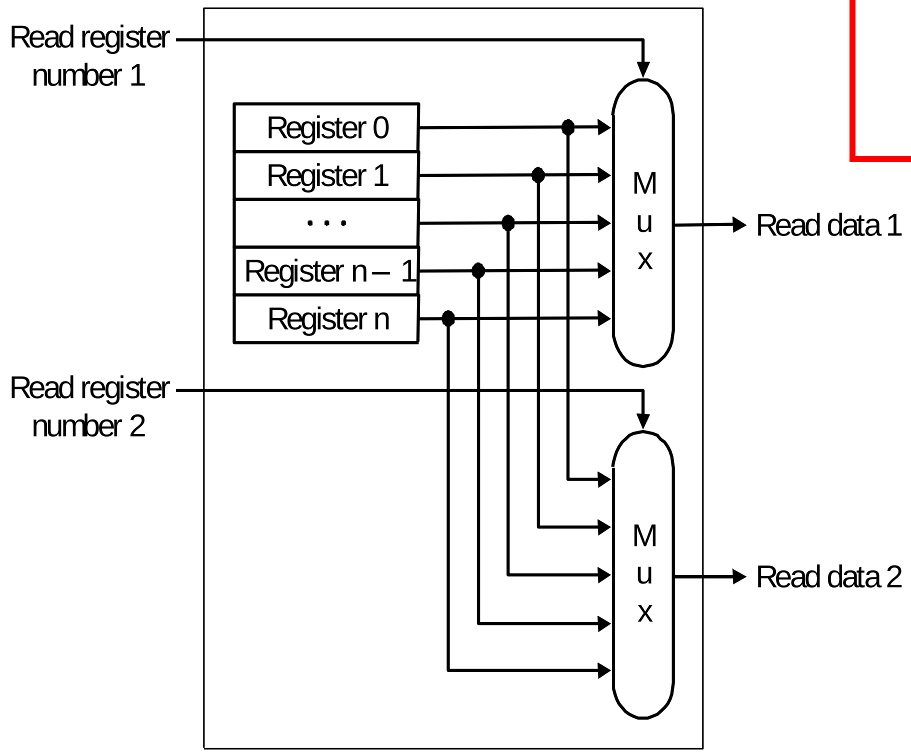
b. Sign-extension unit

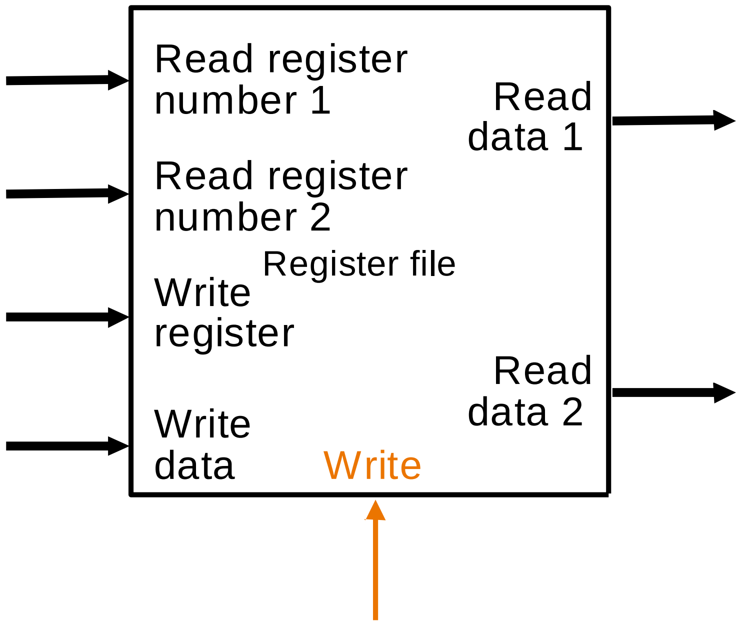
Register File: Read

- Built using D flip-flops

MIPS has a 32 × 32-bit register file
    Use for frequently accessed data
    Numbered 0 to 31

Assembler names
    $t0, $t1, …, $t9 for temporary values
    $s0, $s1, …, $s7 for saved variables

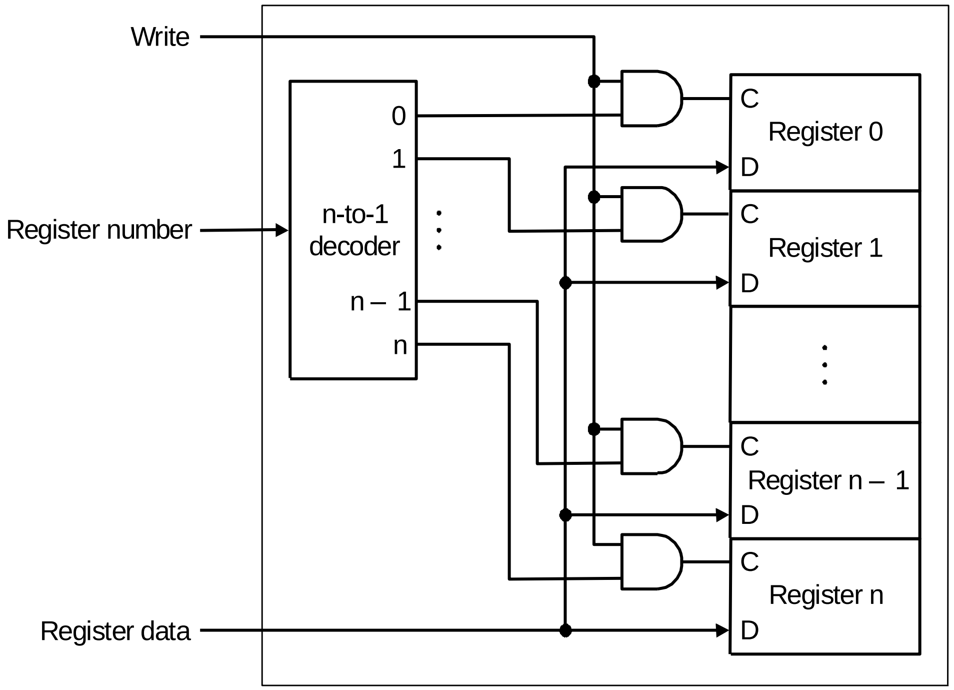




add rd, rs, rt

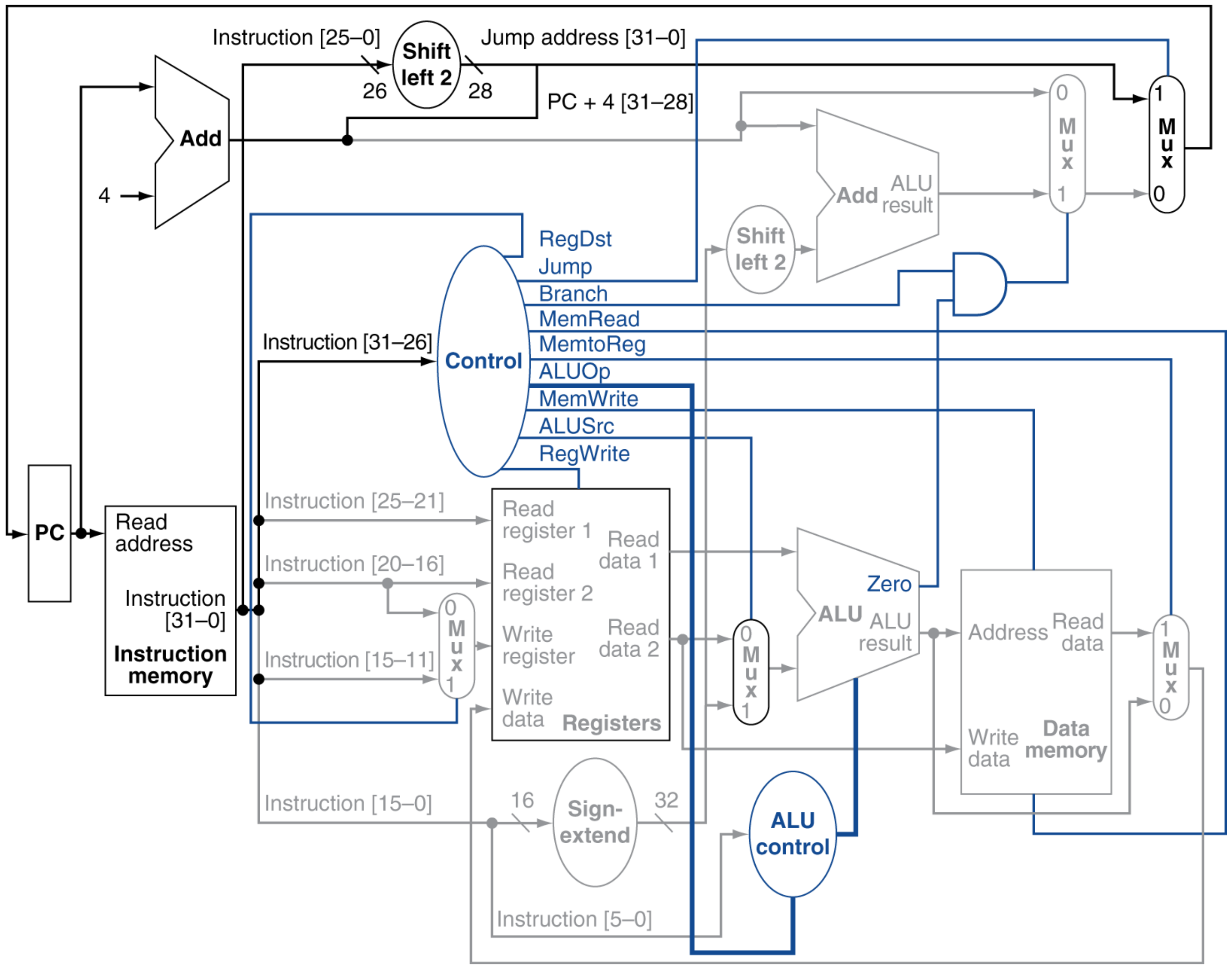
Register File: Write

- use the clock to determine when to write, provided *Write* is enabled



add rd, rs, rt

Control Signal Requirement

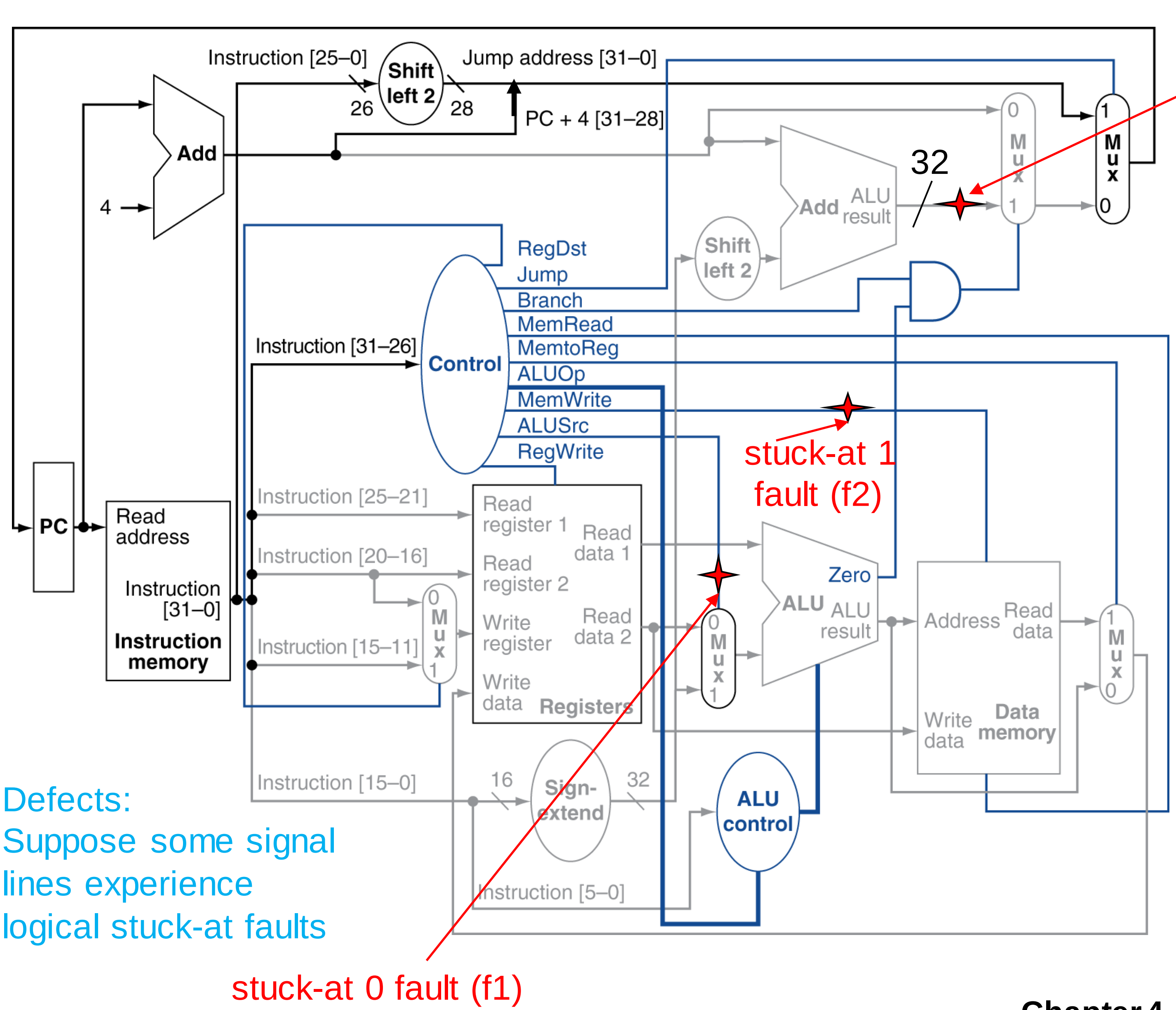


JUMP = 0

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	ADD/SUB/Logical	
`lw`	0	1	1	1	1	0	0	ADD	
`sw`	X	1	X	0	0	1	0	ADD	
`beq`	X	0	X	0	0	0	1	SUB	

Logic Faults in MIPS Processor

In MIPS, programs typically start at memory address 0x00400000



In the presence of f1: ADD, SUB, AND, OR, SLT √

BEQ, JUMP √

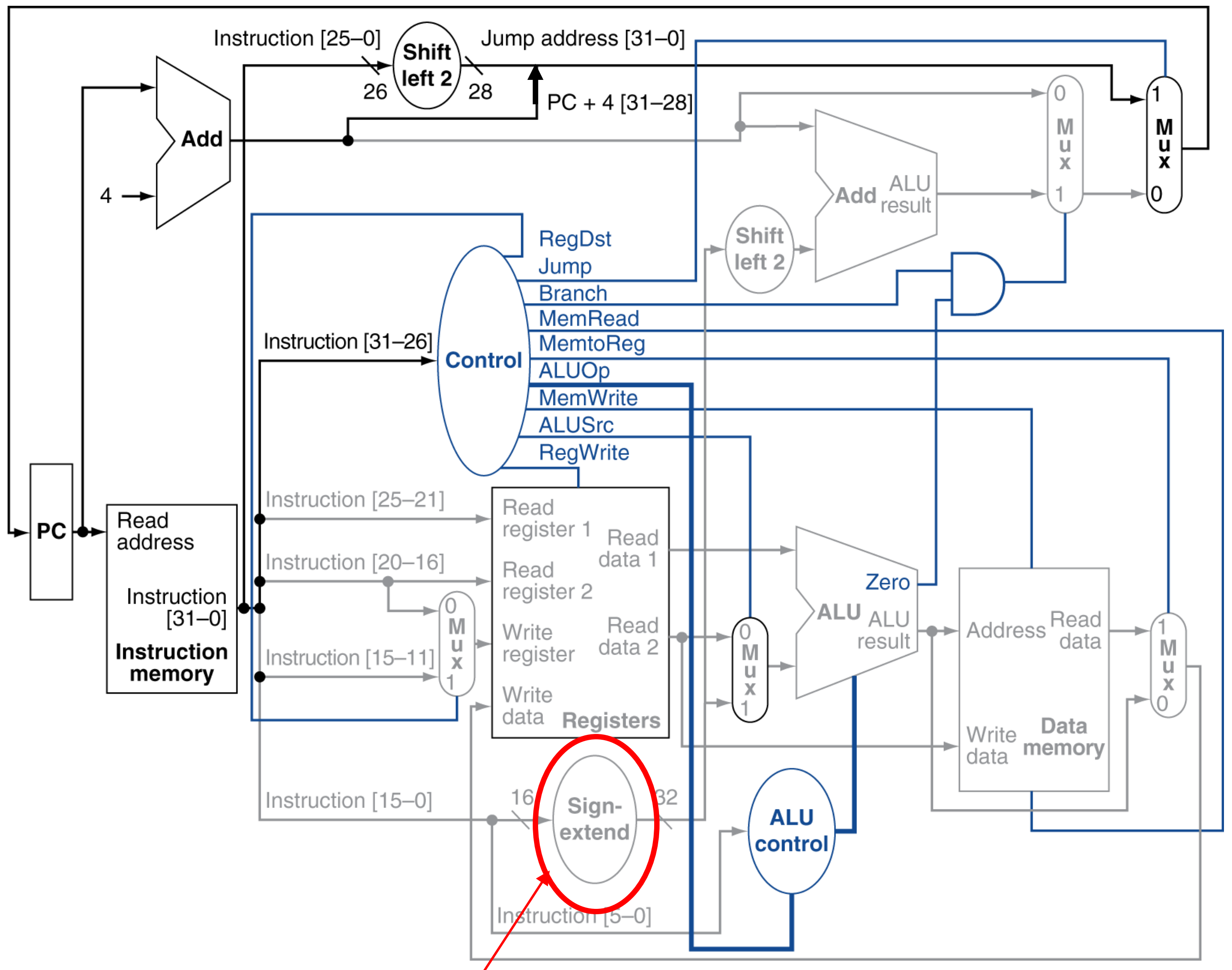
ADDI, SUBI, LW, SW ×

In the presence of f2: ?

In the presence of f3: ? Catastrophic! Why?

Defects: Suppose some signal lines experience logical stuck-at faults

Timing Faults in MIPS Processor



delay fault

Defects:
Suppose Sign-Ext. Unit has become slower than usual;
How to test if it is impacting outputs?

May be we can try:

```
AND  $r1, $r2, $zero
ADDI  $r1, $zero, 5
li $v0, 1
MOVE  $a0, $r1
Syscall
```

Check output to ascertain
if the SE-block is slow beyond tolerance