COMP2611: Computer Organization

The Processor: Datapath & Control

Major Goals

- To present the design of MIPS processor
  - **Single-cycle implementation**
  - **Pipelined single-cycle implementation**

- To illustrate to datapath & control for both implementations

What Do We Study in this Chapter?

- Focus on implementing of a subset of the core MIPS instruction set
  - **Memory-reference instructions**: `lw`, `sw`
  - **Arithmetic-logical instructions**: `add`, `sub`, `and`, `or`, `slt`
  - **Branch and jump instructions**: `beq`, `j`

- Instructions not included:
  - Integer instructions such as those for multiplication and division
  - Floating-point instructions

How Is an Instruction Executed?

1. Fetch the instruction from memory location indicated by program counter (PC)

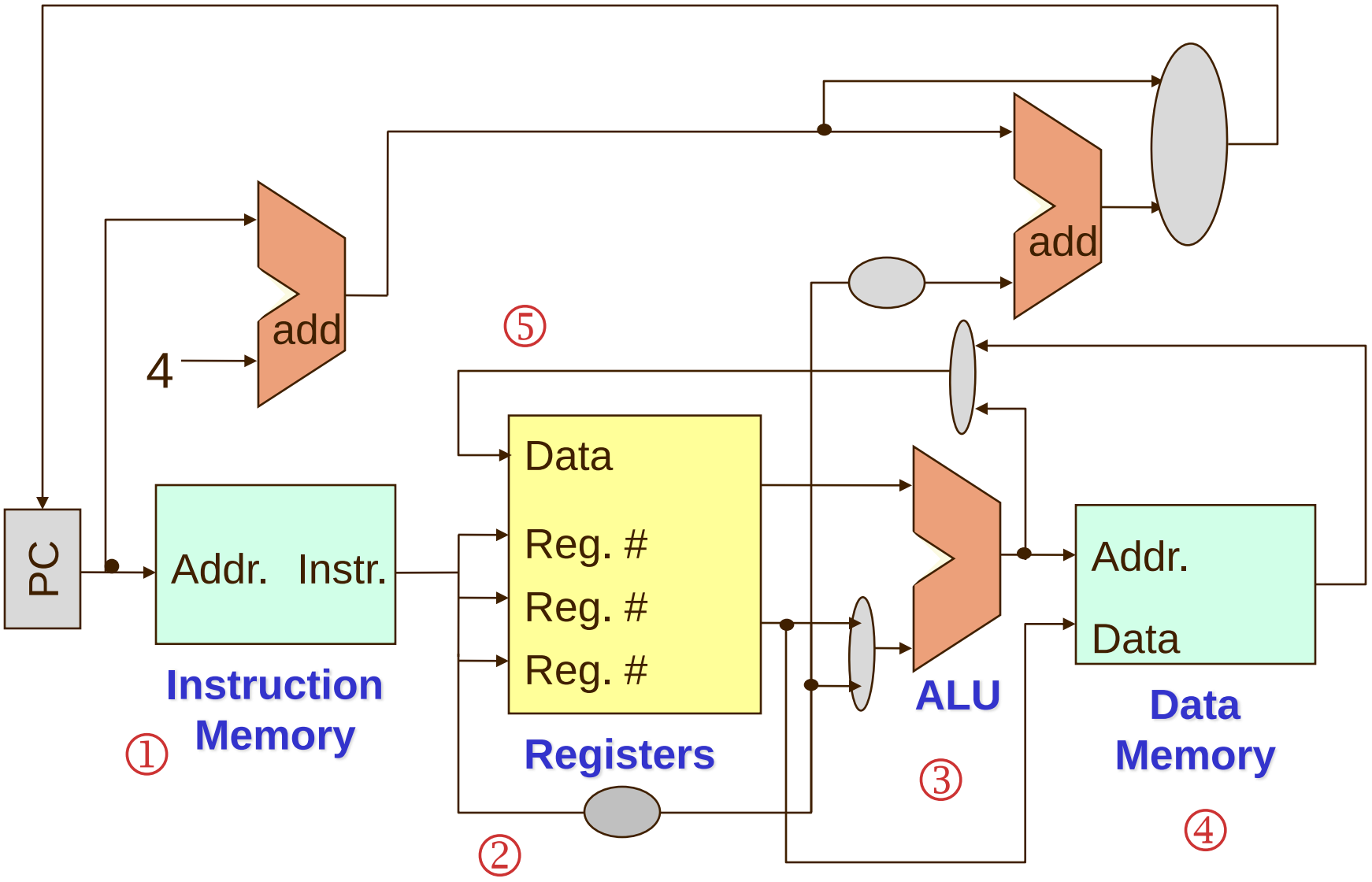
2. Decode the instruction – to find out what to perform
   Meanwhile, read source registers specified in the instruction fields
   - `lw` instruction require reading only one register
   - most other instructions require reading two registers

3. Perform the operation required by the instruction using the ALU
   - Arithmetic & logical instructions: execute
   - Memory-reference instructions: use ALU for address calculation
   - Conditional branch instructions: use ALU for comparison

4. Memory access: `lw` and `sw` instructions

5. Write back the result to the destination register
   Increment PC by 4 or change PC to branch target address

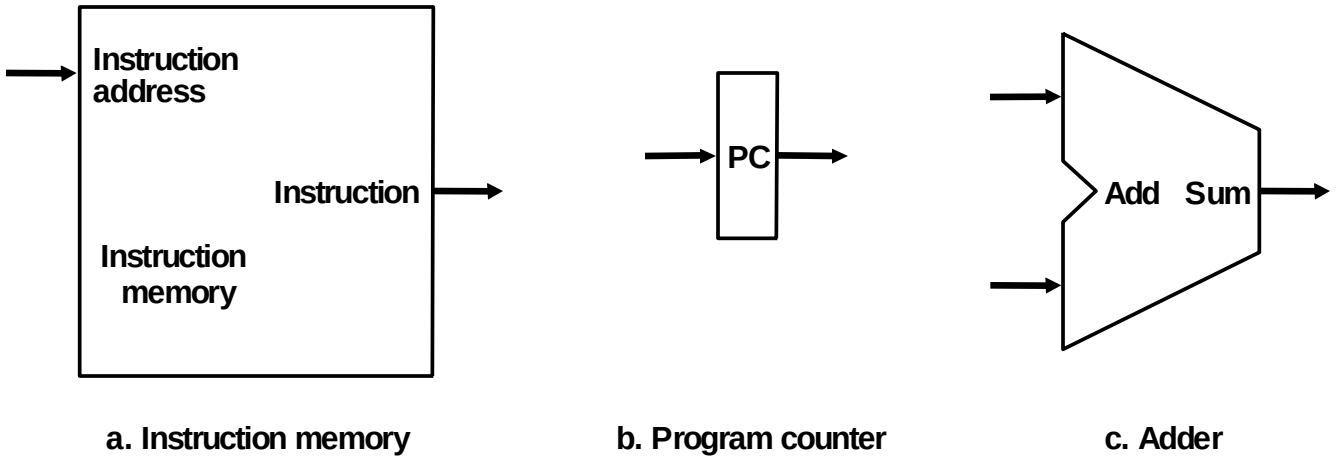
Brief Overview of MIPS Processor Implementation



1. Building a Datapath

Basic Building Blocks

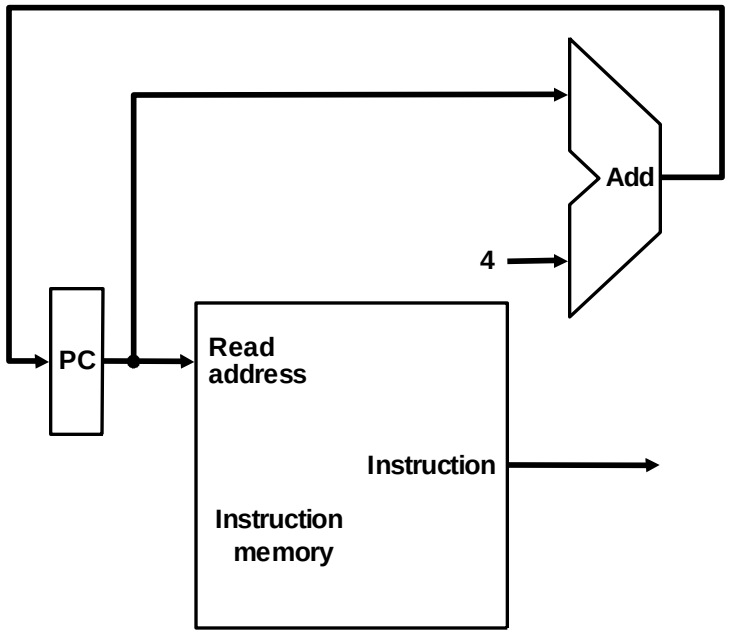
- **Instruction memory**:
  - A memory unit that stores the instructions of a program
  - Supplies an instruction given its address
- **Program counter**:
  - A register storing the address of the instruction being executed
- **Adder**:
  - A unit that increments PC to form the address of next instruction



a. Instruction memory b. Program counter c. Adder

Executing Instruction

1. Fetch the current instruction from memory using PC
2. Prepare for the next instruction
   - By incrementing PC by 4 to point to next instruction (base case)
   - Will worry about the branches later

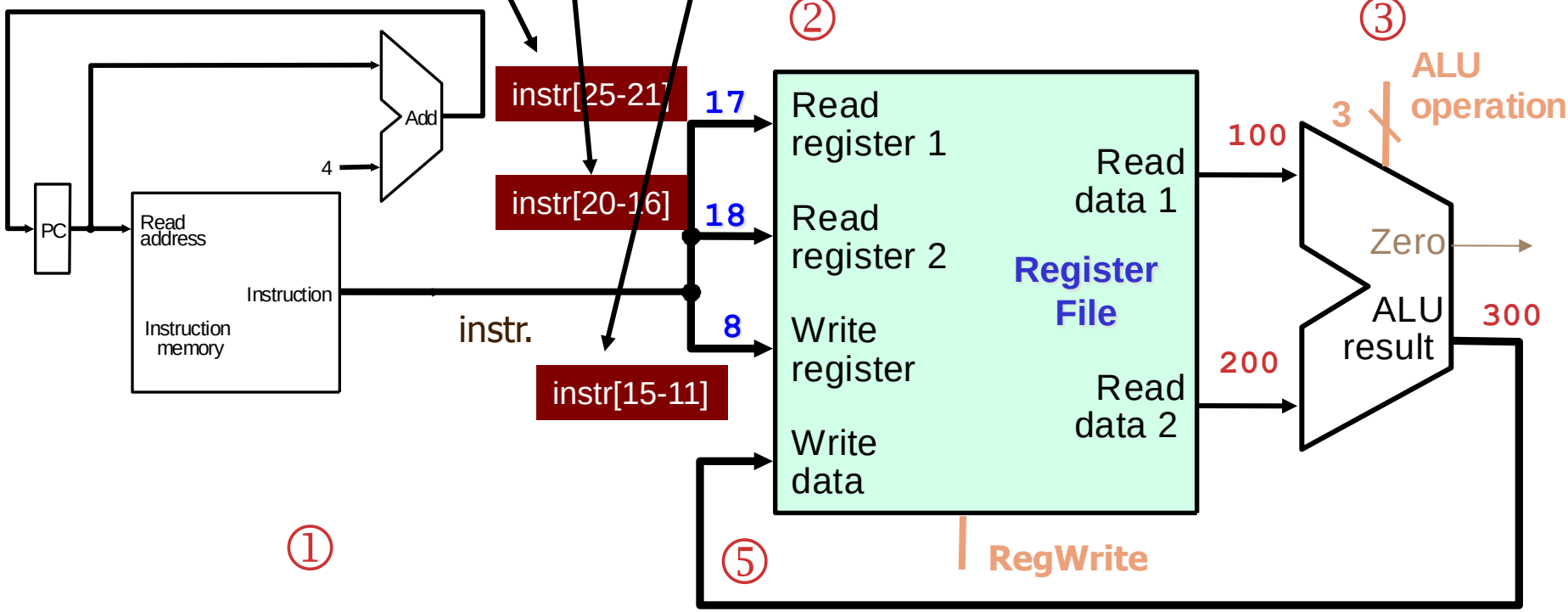


Datapath for Arithmetic/Logical (R-Type) Instr.

- **E.g.:** `add $t0, $s1, $s2`

0	17	18	8	0	32
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Register	Value
s1	100
s2	200



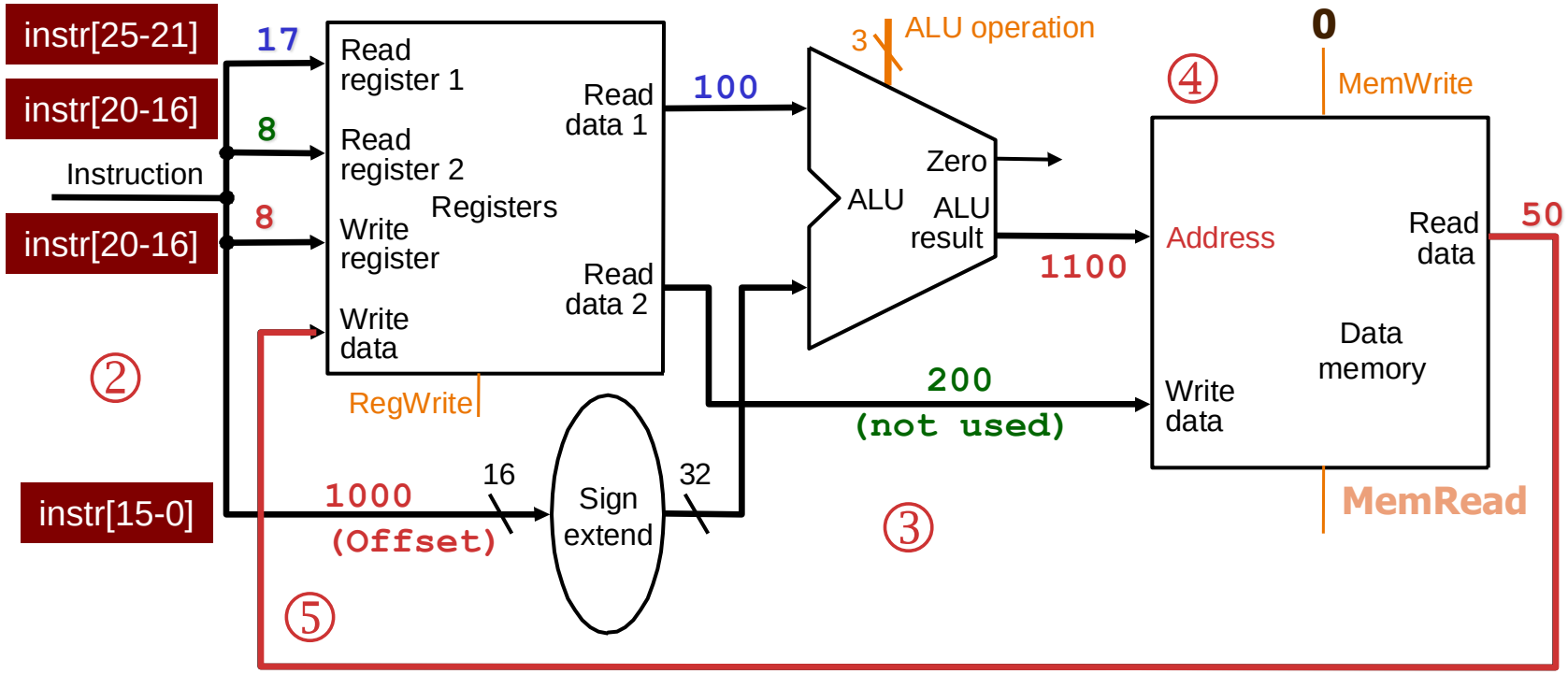
Datapath for Load/Store (I-Format) Instr.

- **E.g.:** `lw $t0, 1000($s1)`

35	17 (s1)	8 (t0)	1000 (immediate)
100011	10001	01000	0000 0011 1110 1000
6 bits	5 bits	5 bits	16 bits

Register	Value
t0	200
s1	100

Memory[1100]
50



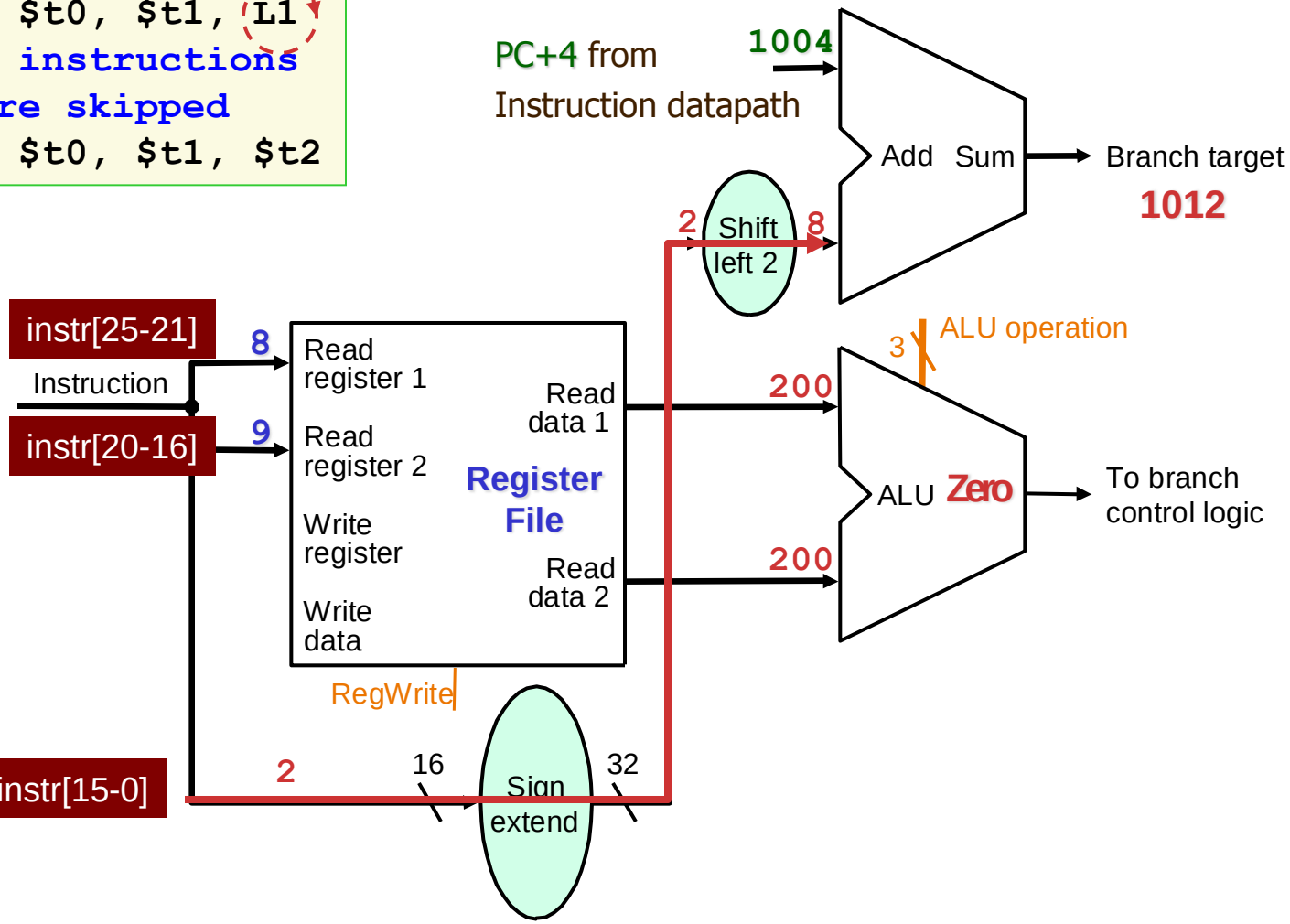
Note: For load word instruction, **MemWrite** has to be de-asserted so that the memory will not be modified by incoming write data.

Datapath for Branch (I-Format) Instr.

- **E.g.:** `beq $t0, $t1, 2`

```
1000:      beq $t0, $t1, L1
1004:      . 2 instructions
1008:      . are skipped
1012: L1:  add $t0, $t1, $t2
```

Register	Value
t0	200
t1	200
PC	1000



A Simple Single-Cycle Implementation

- We have already built a datapath for each instruction separately
- Now, we need to combine them into a **single datapath**
- **Key to combine**

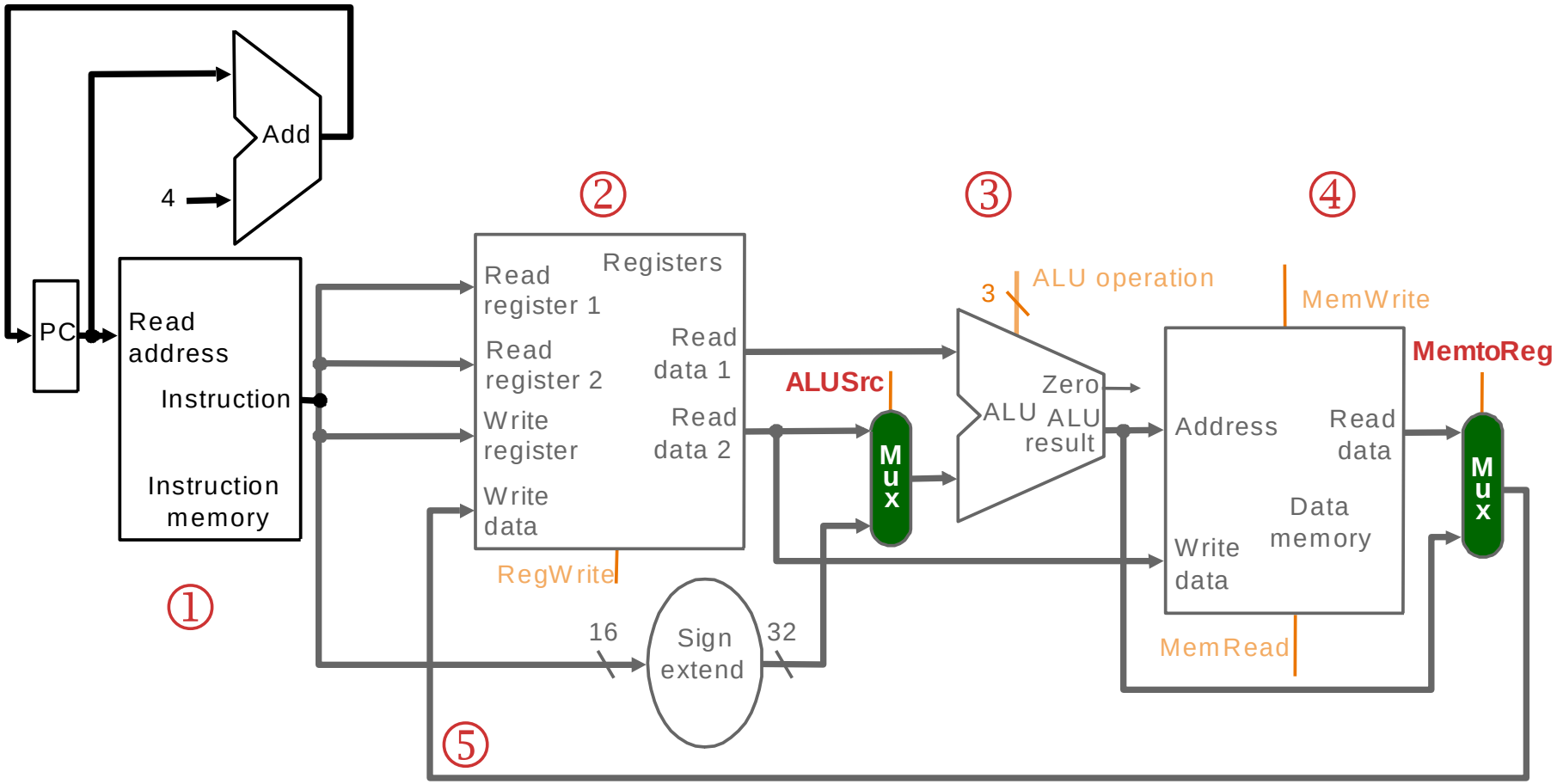
> Share some of the resources (e.g., ALU) among different instructions

Note:

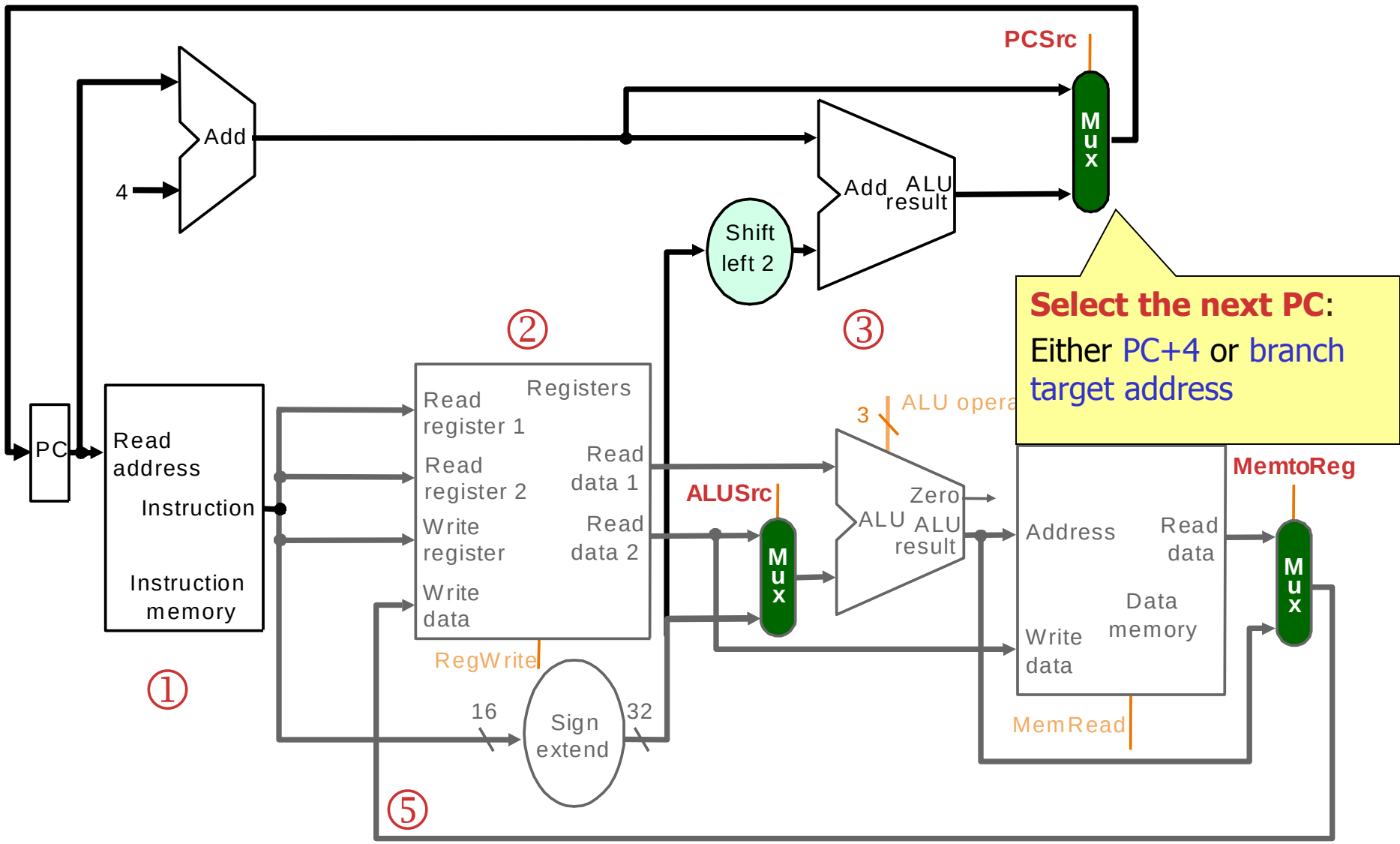
- This simple implementation is based on the (unrealistic) assumption
  - i.e. all instructions take just one clock cycle each to complete
- Implication:
  - No datapath resource can be used more than once per instruction
  - Any element needed more than once must be duplicated
  - Instructions and data have to be stored in separate memories

Combined Datapath for R-Type & Memory Instr.

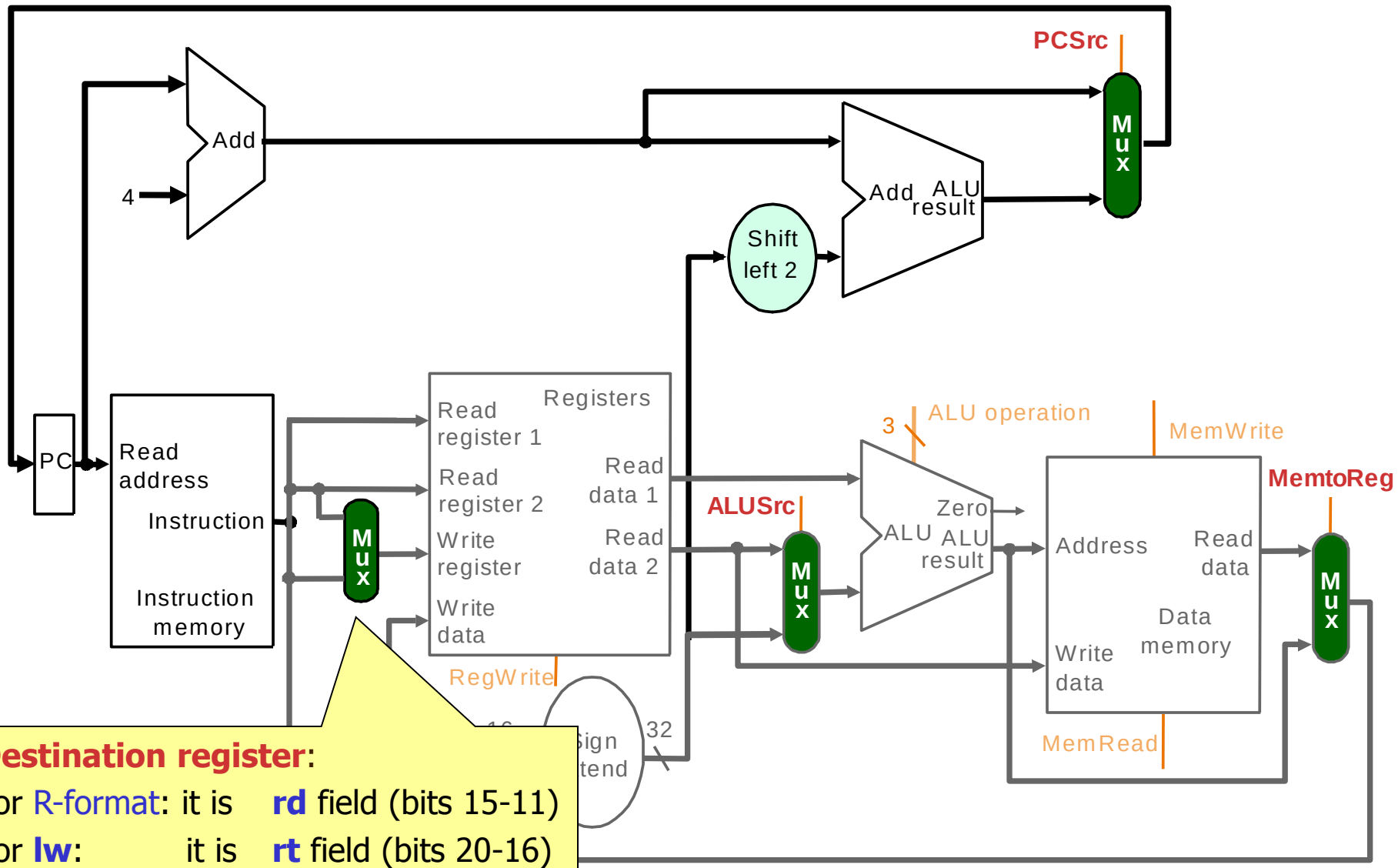
- Use **ALUSrc** to decide which source will be sent to the ALU
- Use **MemtoReg** to choose the source of output back to dest. register



Combined Datapath for Different Instr. Classes



Muxing Two Possible Destination Registers



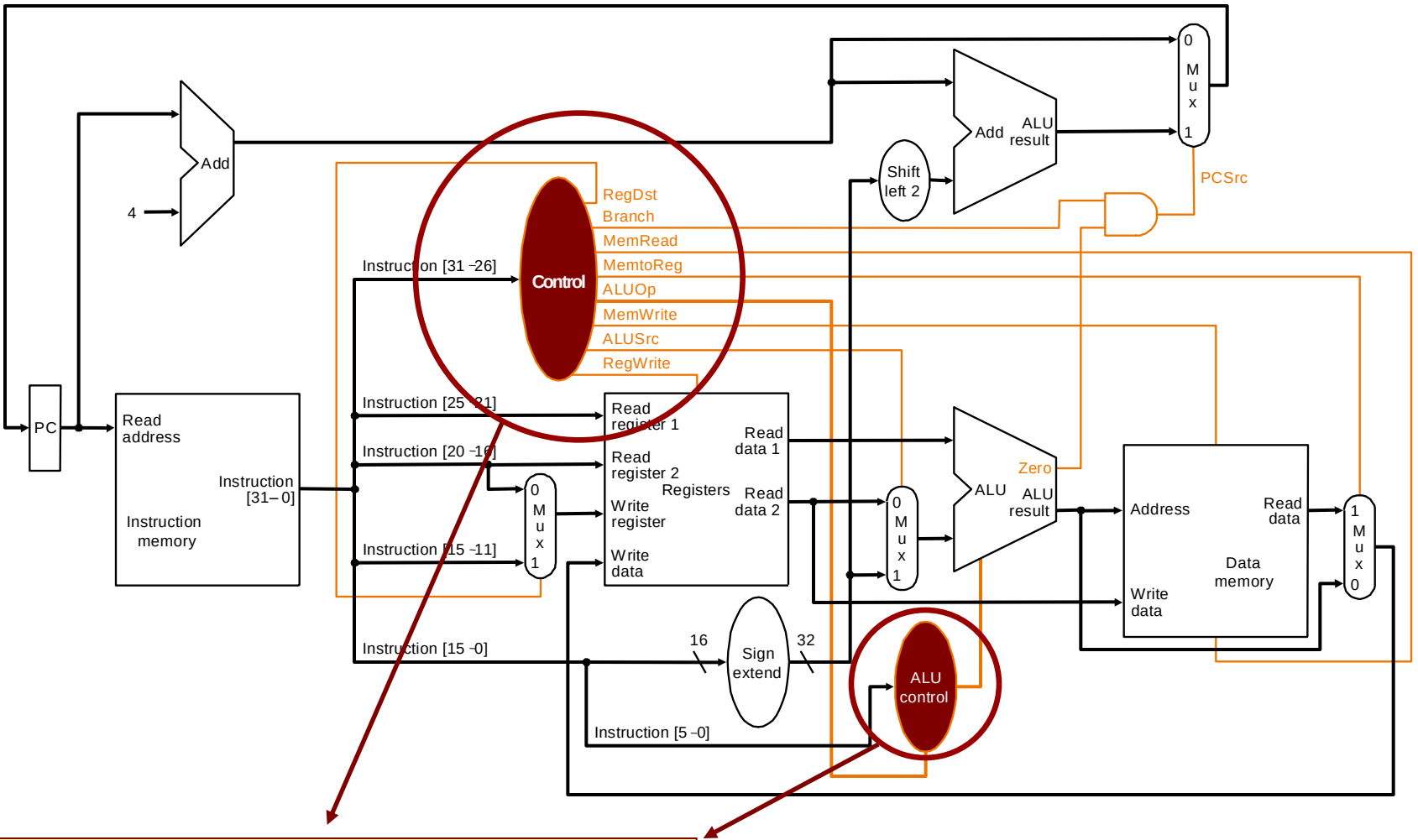
Destination register:

For R-format: it is **rd** field (bits 15-11)

For **lw**: it is **rt** field (bits 20-16)

2. Control

Overview: Datapath + Control Unit



Topic we are going to discuss next

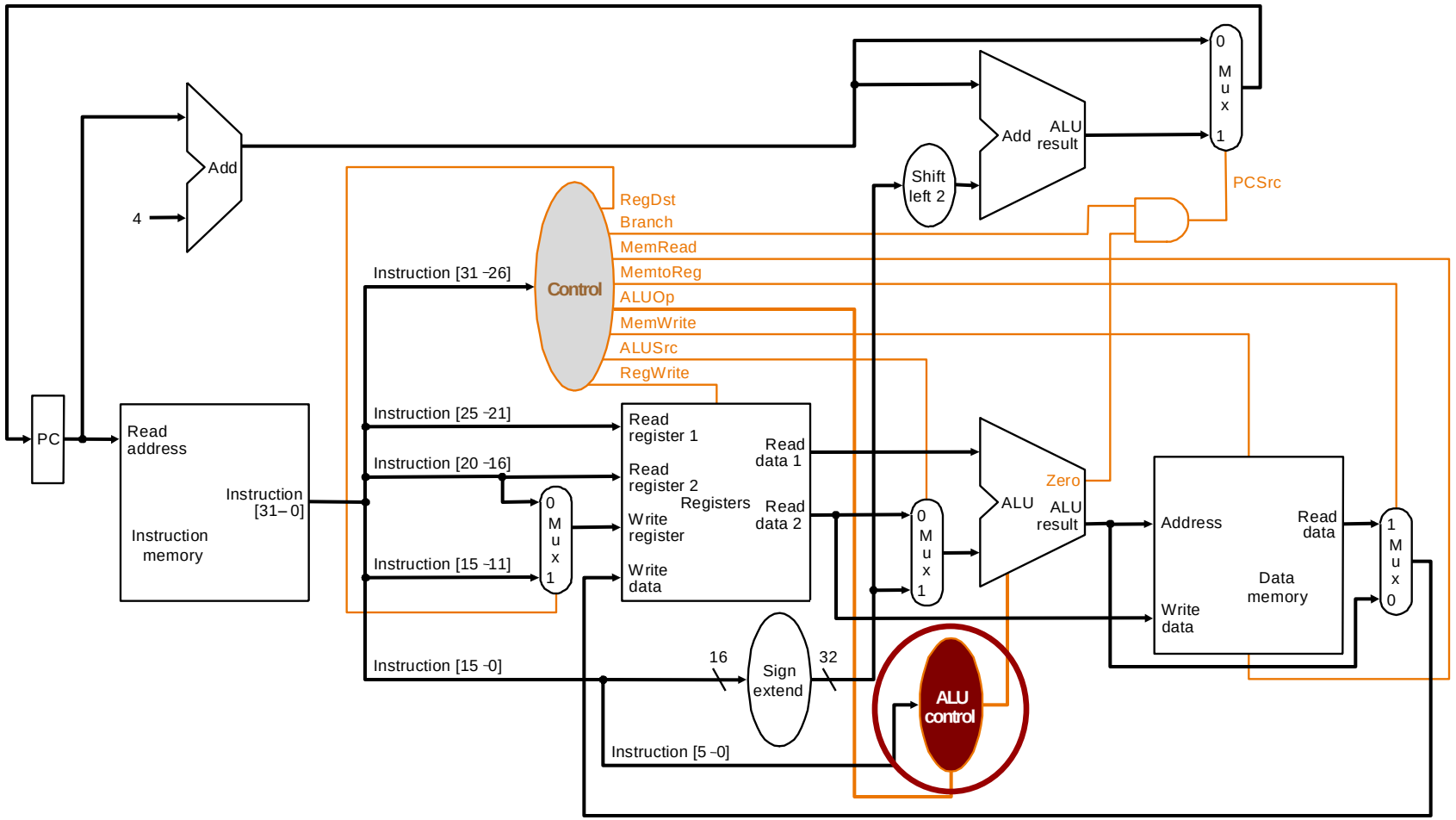
Datapath control unit controls the whole operation of the datapath

- How? Through **control signals**, e.g.
  - read/write signals for state elements: RegWrite, MemWrite, MemRead
  - selector inputs for multiplexors: ALUSrc, MemtoReg, PCSrc, RegDst
  - ALU control inputs (updated to 4 bits) for proper operations

ALU Control Input	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set on less than
1100	NOR

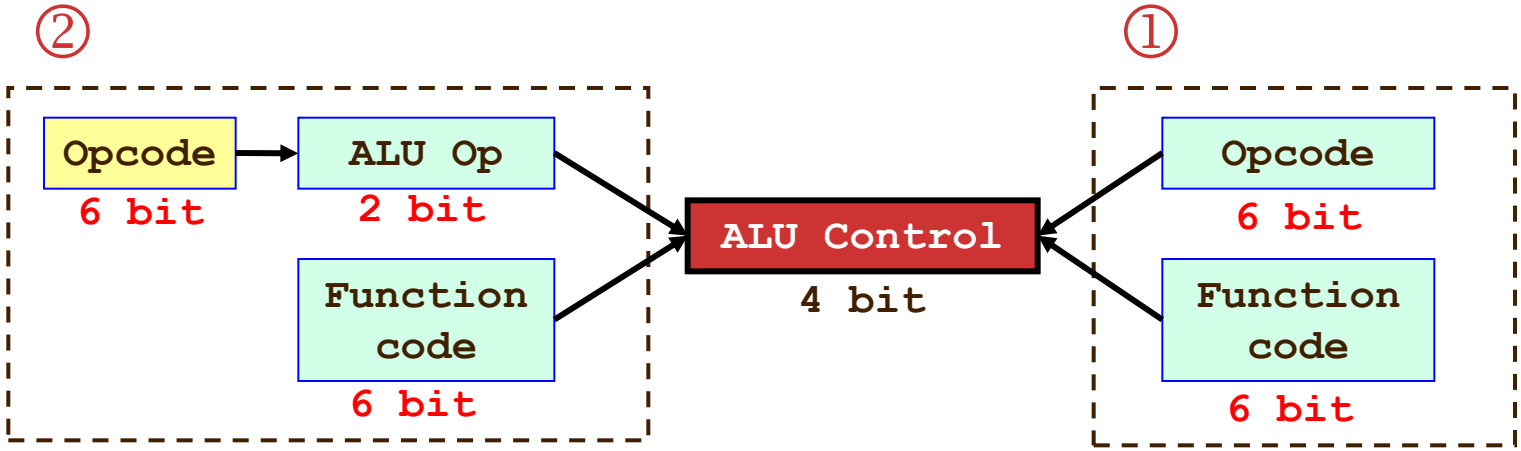
- The **ALU control** is part of the **datapath control unit**

Next: ALU Control



Generation of ALU Control Input Bits

- Two common implementation techniques:
  1. 1-level decoding
     - more input bits
  2. 2-level decoding
     + less input bits, less complicated => potentially faster logic



2 levels of decoding: only 8 inputs are used to generate 3 outputs in 2nd level

1 level only, a logic circuit with 12 inputs is needed

ALU Control Input Bits

- **Inputs** used by control unit to generate ALU control input bits:
  - **ALUOp** (2 bits)
  - **Function code** of instruction (6 bits)

Instruction operation	Desired ALU action	Instruction opcode	ALUOp	Function code	ALU control input
lw	add	load word	**00**	**XXXXXX**	**0010**
sw	add	store word	**00**	**XXXXXX**	**0010**
beq	subtract	branch equal	**01**	**XXXXXX**	**0110**
add	add	R-type	**10**	**100000**	**0010**
sub	subtract	R-type	**10**	**100010**	**0110**
and	and	R-type	**10**	**100100**	**0000**
or	or	R-type	**10**	**100101**	**0001**
slt	set on less than	R-type	**10**	**101010**	**0111**

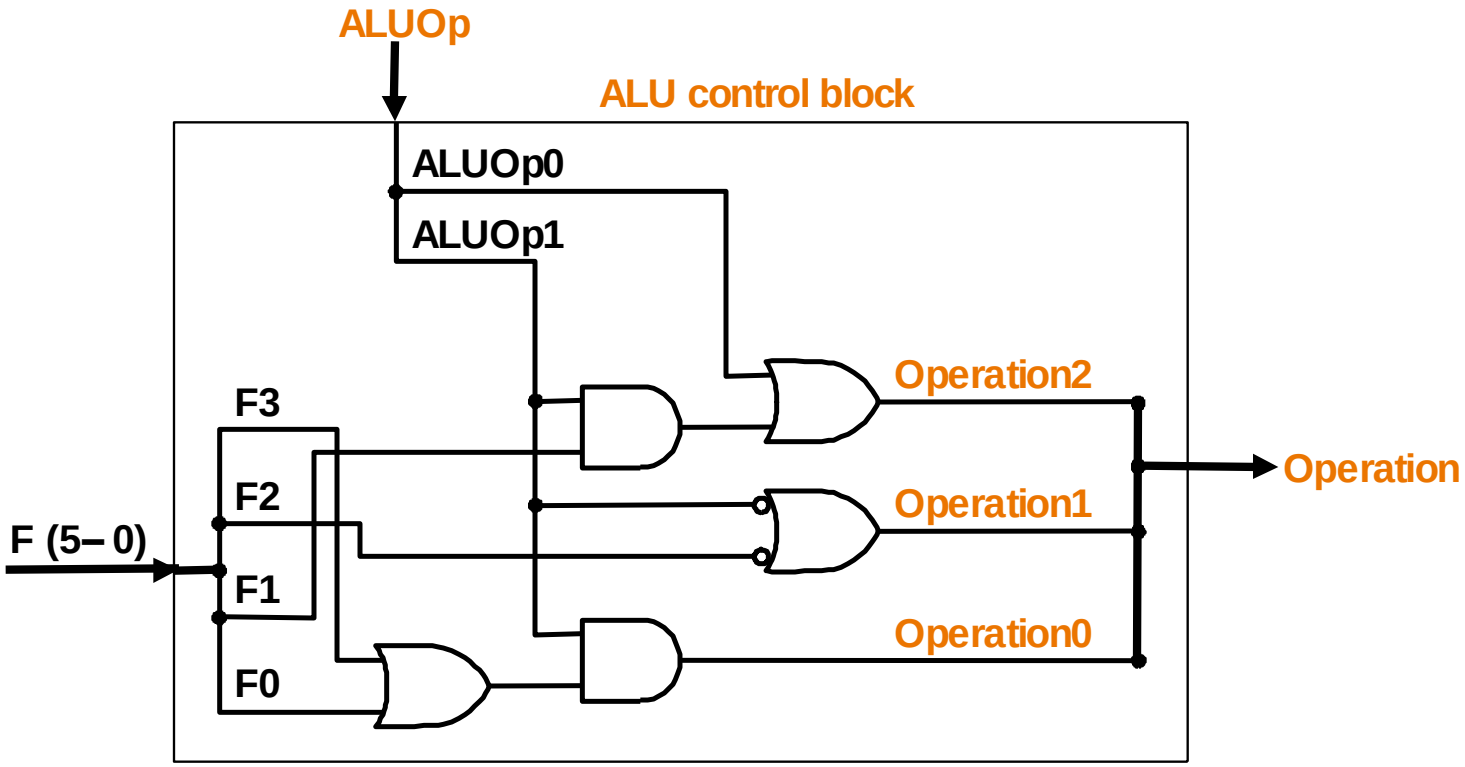
Implementing ALU Control Block

- Start from truth table
- Smart design converts many entries in the table to don't-care terms, leading to a simplified hardware implementation

ALUOp		Function code						Operation	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
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	R-type Instr.
1	X	X	X	0	0	1	0	0110	
1	X	X	X	0	1	0	0	0000	
1	X	X	X	0	1	0	1	0001	
1	X	X	X	1	0	1	0	0111	

- **Why we can come up with some many don't care?**

Hardware Implementation of ALU Control Block



Effects of Control Signals

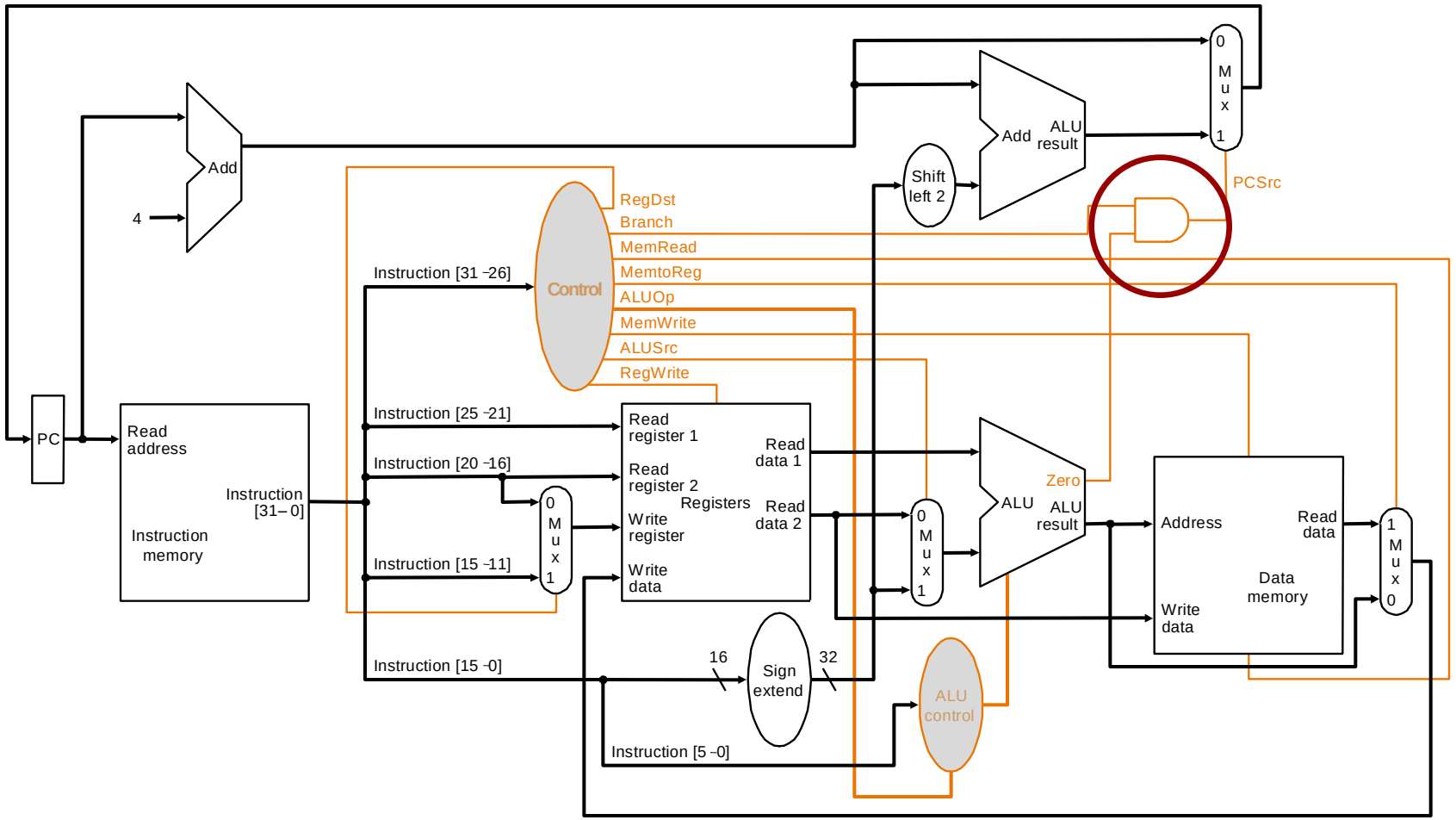
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from **rt** field (bits 20-16)	The register destination number for the Write register comes from **rd** field (bits 15-11)
RegWrite	None	Enable data write to the register specified by the register destination number
ALUSrc	The second ALU operand comes from the second register file output (Read data port 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The next PC picks up the output of the adder that computes PC+4	The next PC picks up the output of the adder that computes the branch target
MemRead	None	Enable read from memory. Memory contents designated by the address are put on the Read data output
MemWrite	None	Enable write to memory. Overwrite the memory contents designated by the address with the value on the Write data input
MemtoReg	Feed the Write data input of the register file with output from ALU	Feed the Write data input of the register file with output from memory

Setting of Control Signals

- The 9 control signals (7 from previous table + 2 from ALUOp) can be set based entirely on the 6-bit opcode, with the exception of PCSrc

- PCSrc control line is set if both conditions hold simultaneously:
  a. Instruction is a branch, e.g. **`beq`**
  b. Zero output of ALU is true (i.e., two source operands are equal)

Checking the Conditions for PCSrc



Setting of Control Signals (Cont'd)

- Setting of control lines (output of control unit):

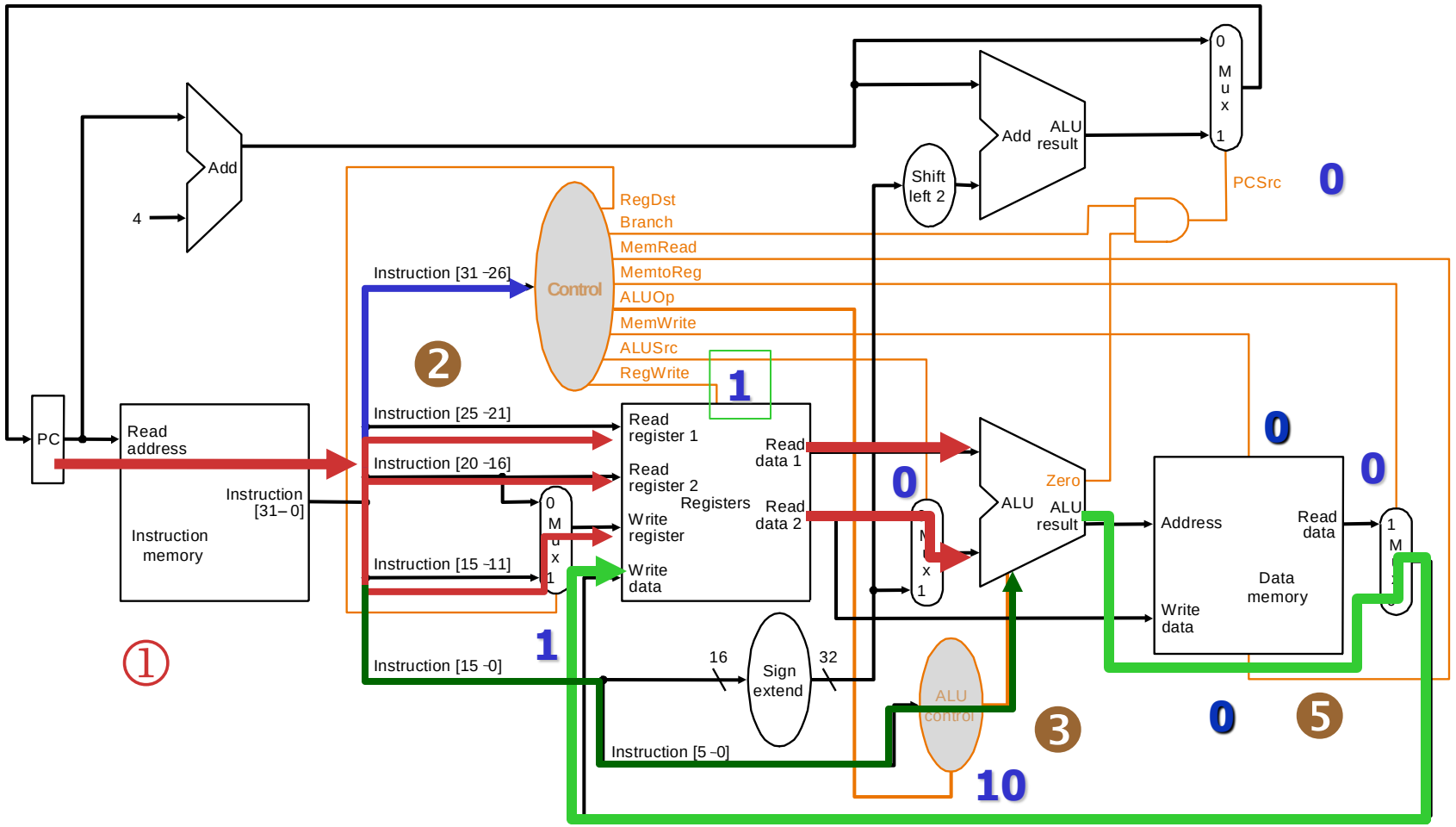
Instruction	Reg-Dst	ALU-Src	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

`sw` & `beq` will not modify any register, it is ensured by making RegWrite to 0
So, we don't care what write register & write data are

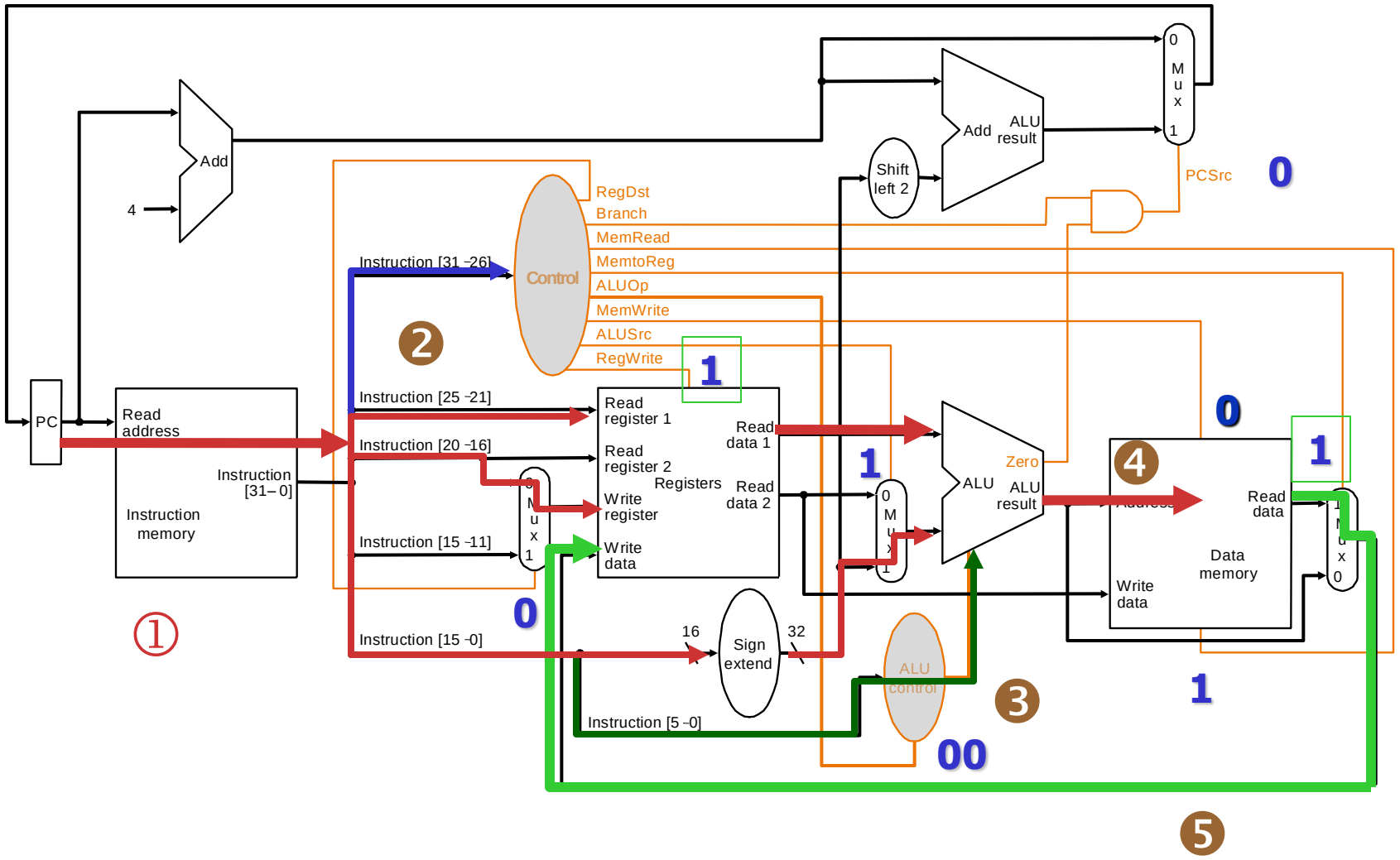
- Input to control unit (i.e. opcode determines setting of control lines):

Instruction	Opcode in decimal	Opcode in binary					
		Op5	Op4	Op3	Op2	Op1	Op0
R-format	0	0	0	0	0	0	0
`lw`	35	1	0	0	0	1	1
`sw`	43	1	0	1	0	1	1
`beq`	4	0	0	0	1	0	0

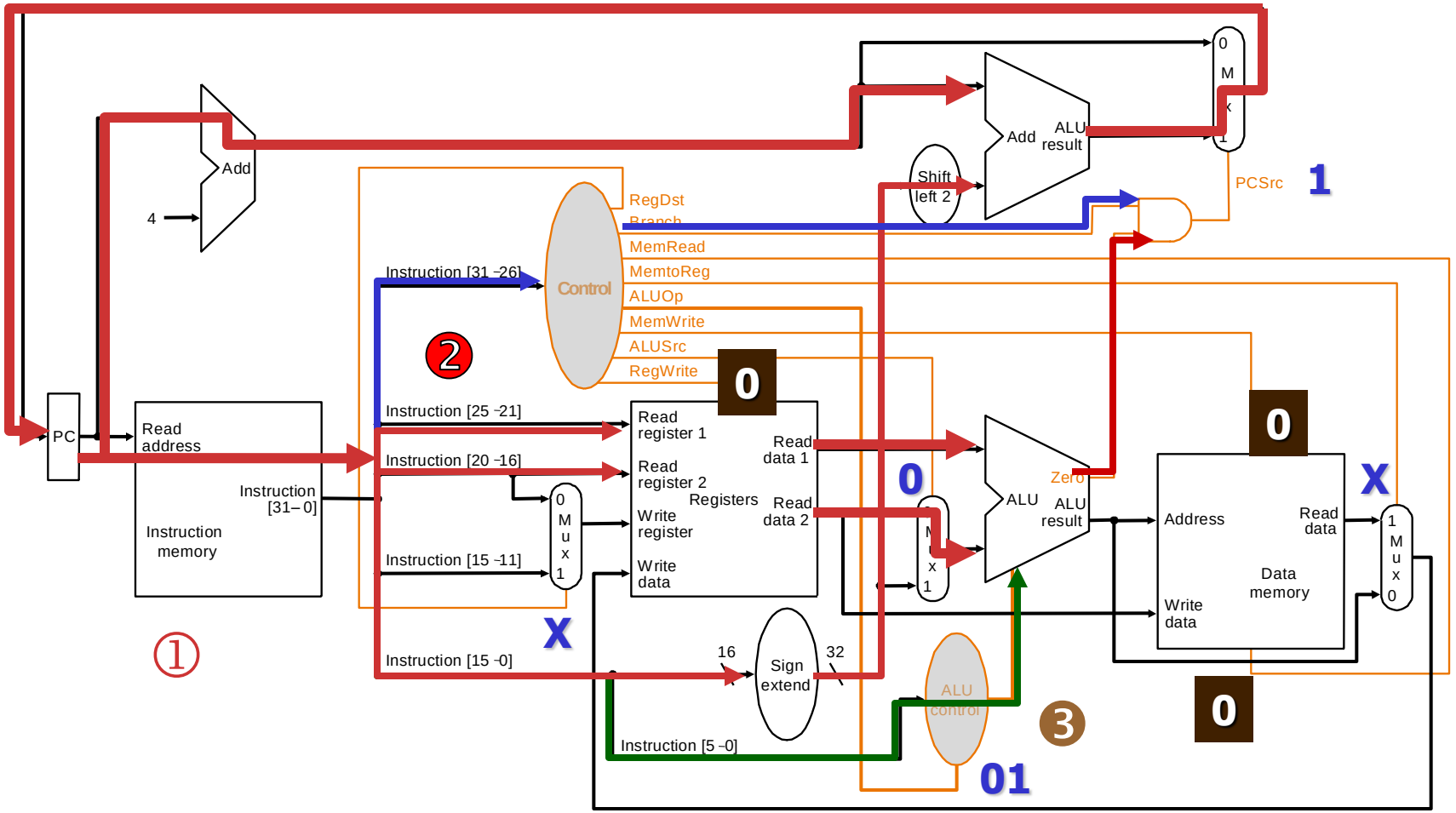
Review Datapath for R-Type Instr.



Review Datapath for Load Instr.



Review Datapath for Branch Instr.

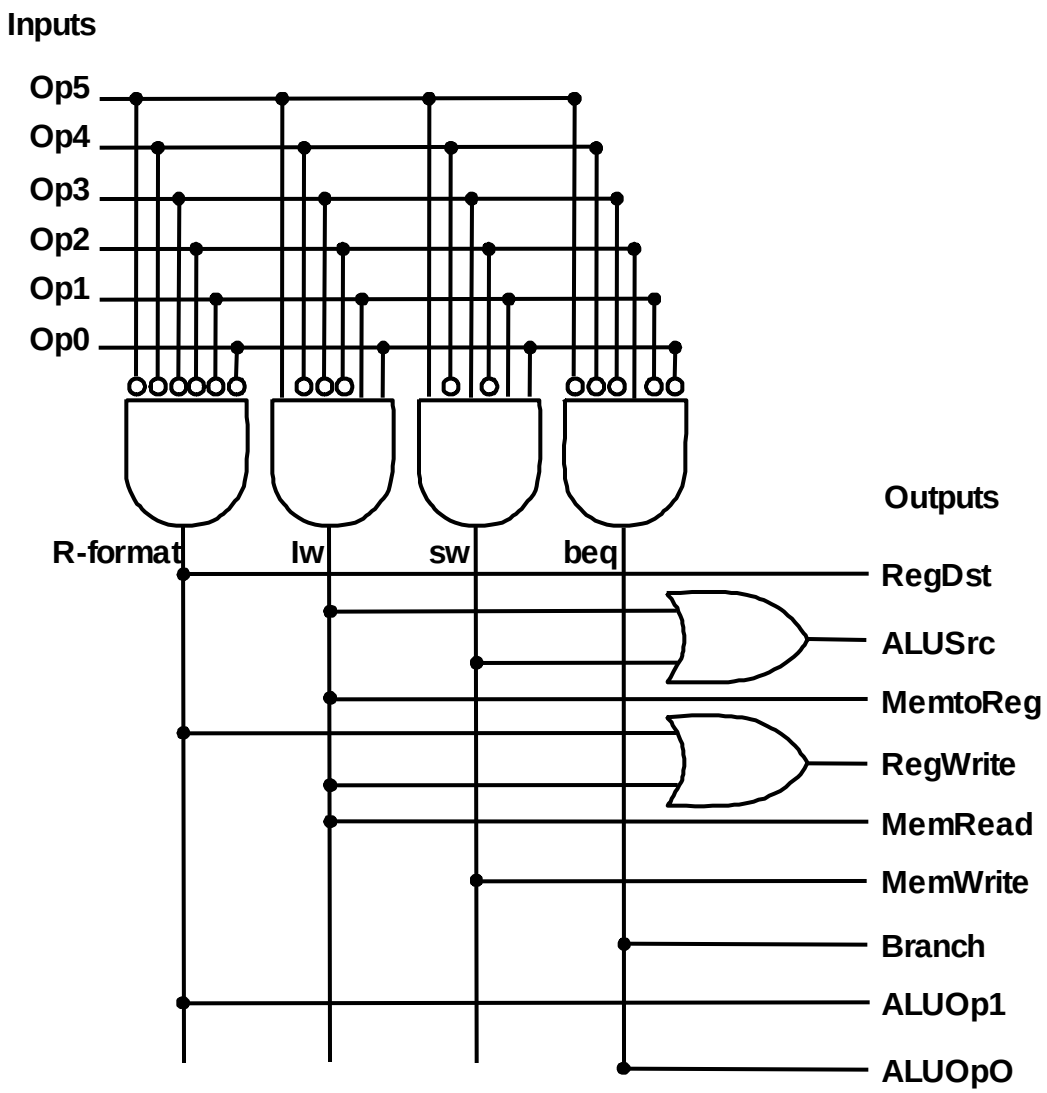


Implementing Datapath Control Unit

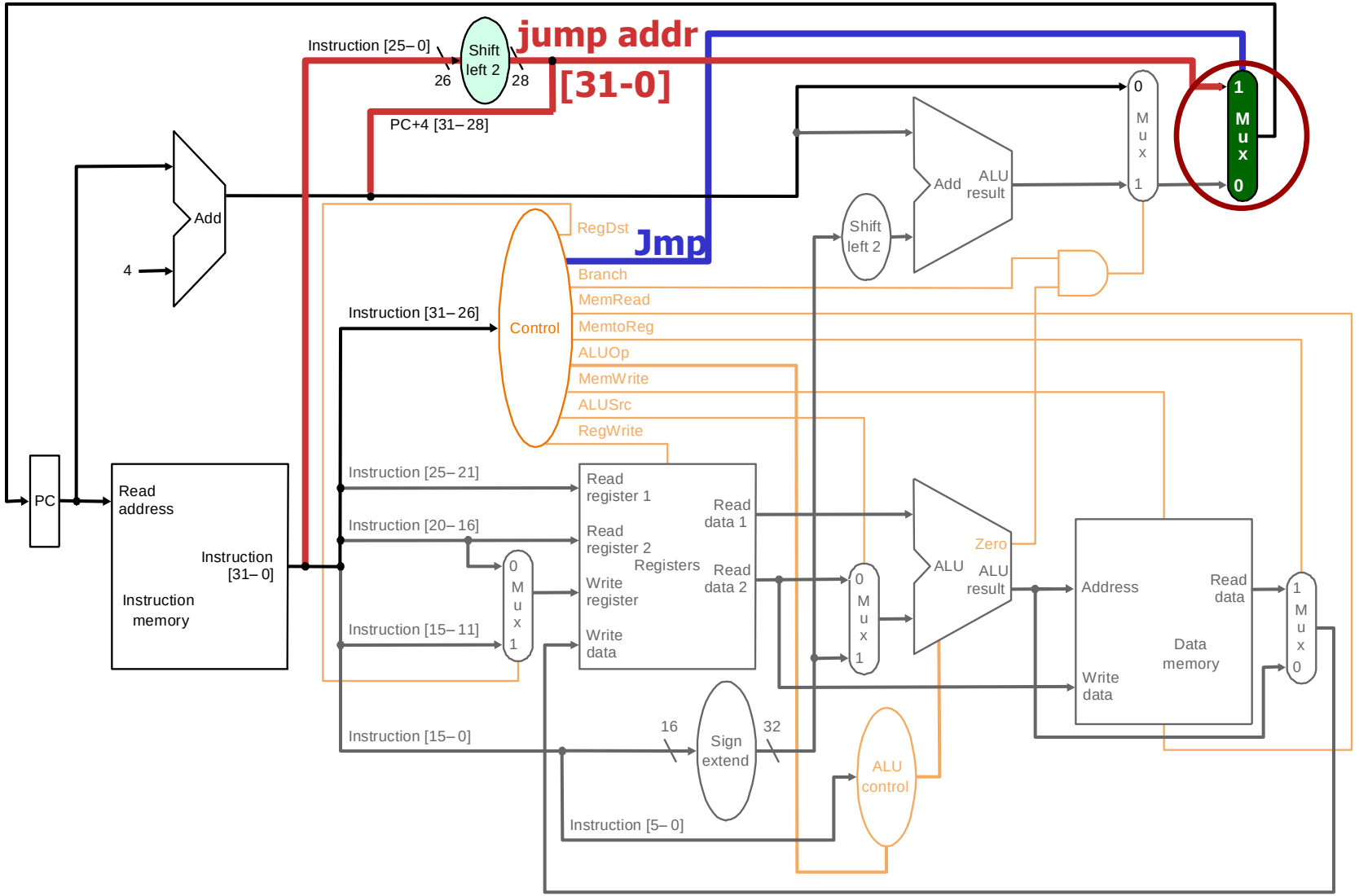
- Start with truth table

Input or output	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

Hardware Implementation of Datapath Control Unit



Extend Datapath & Control to Handle Jump Instr.



Single-cycle implementation can't run very fast

Why?

- **Every instruction takes one clock cycle (CPI = 1), and**
- **Clock cycle is determined by the longest path in the machine**
  - i.e. clock cycle is expected to be large, resulting in poor performance

What we have seen so far, the longest path is for a load instruction

- Load involves five functional units in series
- i.e. instruction mem., register file, ALU, data mem., register file

It is more severe when considering other computational instructions

- e.g. multiplication, division, and floating-point operations, etc.

Other issues

- Sharing of hardware functional units is NOT possible within a cycle