COMP2611: Computer Organization

The Processor: Datapath & Control

Major Goals

- To present the design of MIPS processor
  - **Single-cycle implementation**
  - **Multiple-cycle implementation**
  - **Pipelined Processor**
- To illustrate to datapath & control for both implementations
- To introduce two control unit design approaches
  - Finite State Machine representation
  - Microprogramming

How Does Processor Design Impact Performance?

- Review the three components in the iron law of performance analysis
  - **Instruction count**
    - Determined by the compiler and the ISA
  - **Clock cycle time** &
  - **Clock cycles per instruction** (**CPI**)
    - Determined by the implementation of the processor

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

Big Picture

- First, understand how an instruction is executed before the design
- Next, split the execution of an instruction into multiple steps common to all instructions
  - To share the hardware resources as much as we can
- Next, implement each part separately
- Finally, put all these parts back together

How Is an Instruction Executed?

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

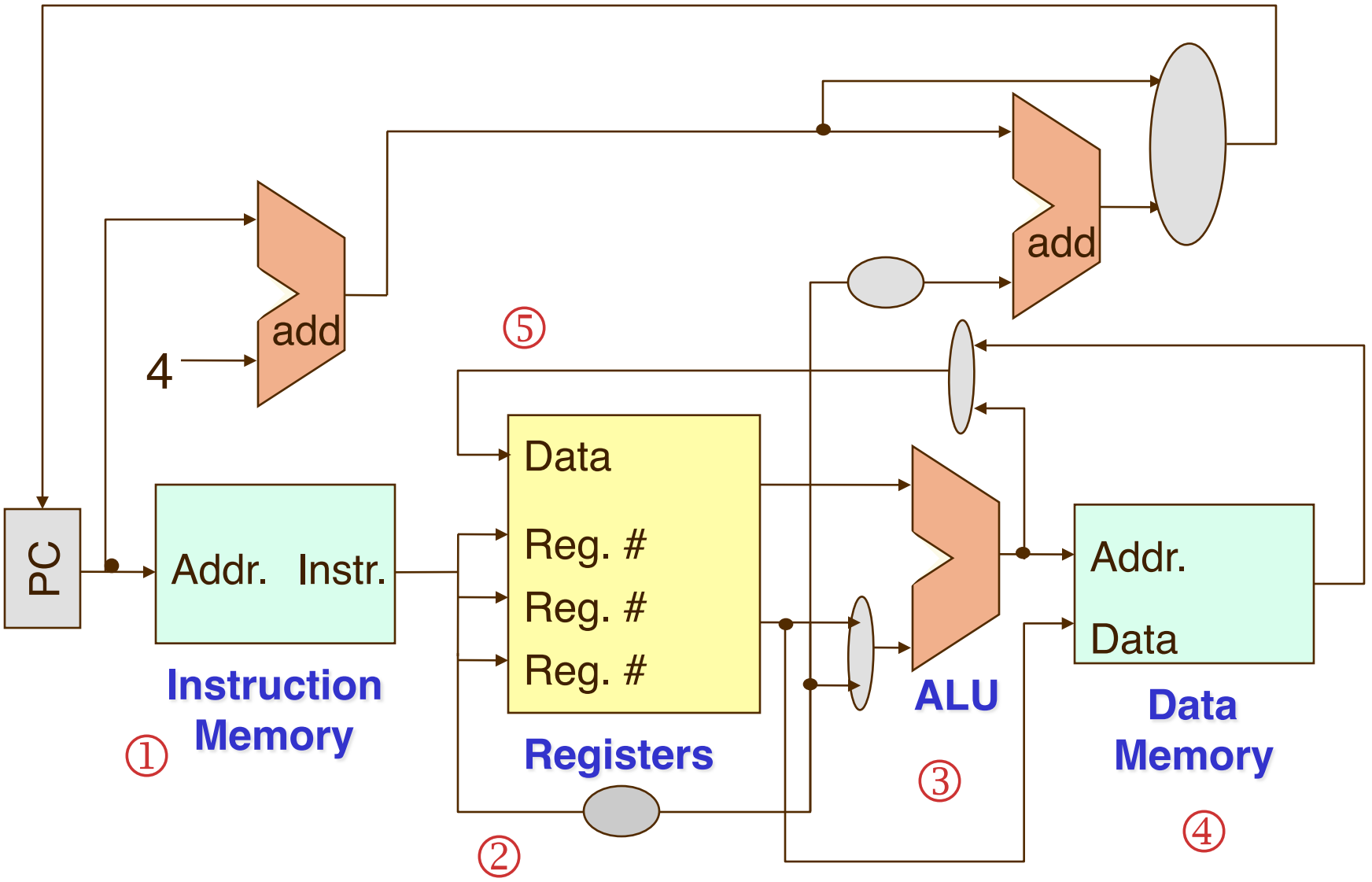
2. Decode the instruction – to find out what to perform
   Meanwhile, read source registers specified in the instruction fields
   - `lw` instruction require reading only one register
   - most other instructions require reading two registers

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

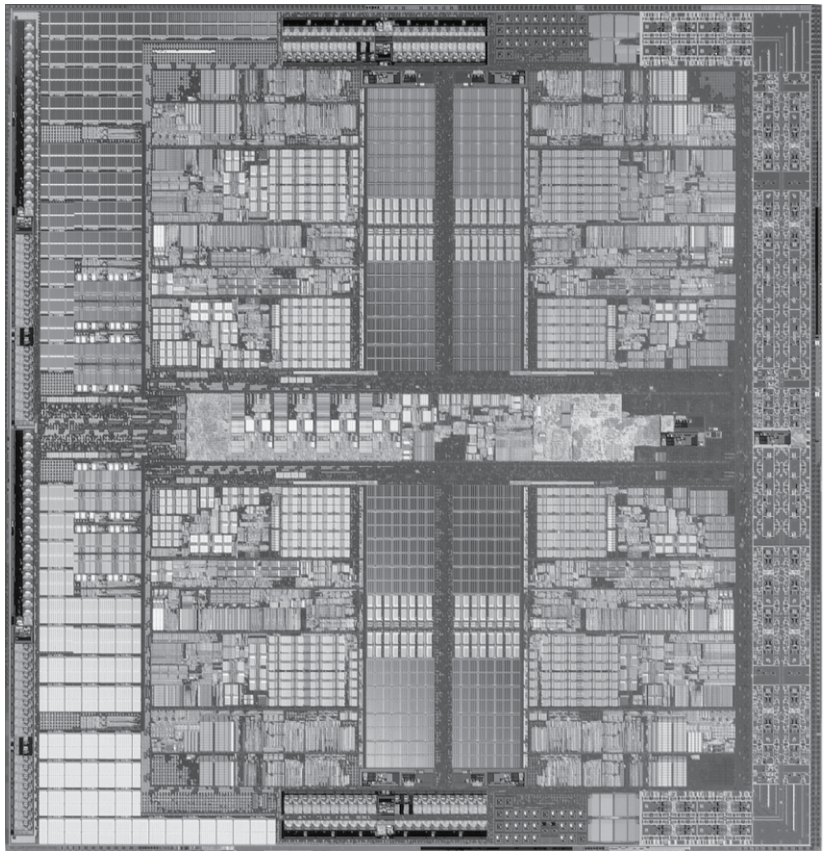
4. Memory access: `lw` and `sw` instructions

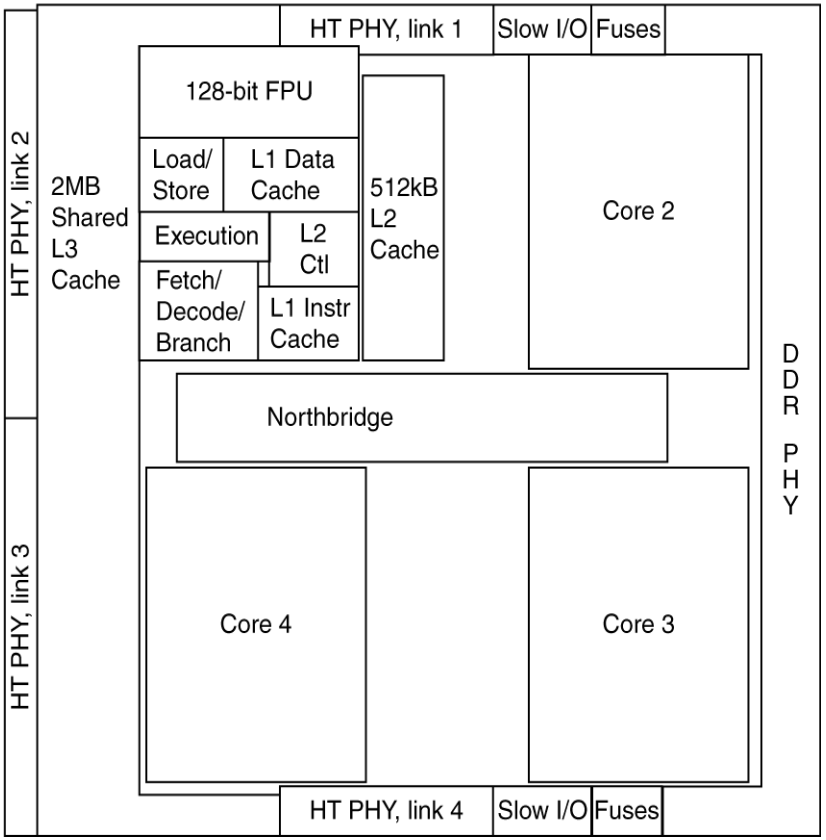
5. Write back the result to the destination register
   Increment PC by 4 or change PC to branch target address

Brief Overview of MIPS Processor Implementation



- AMD Barcelona: 4 processor cores

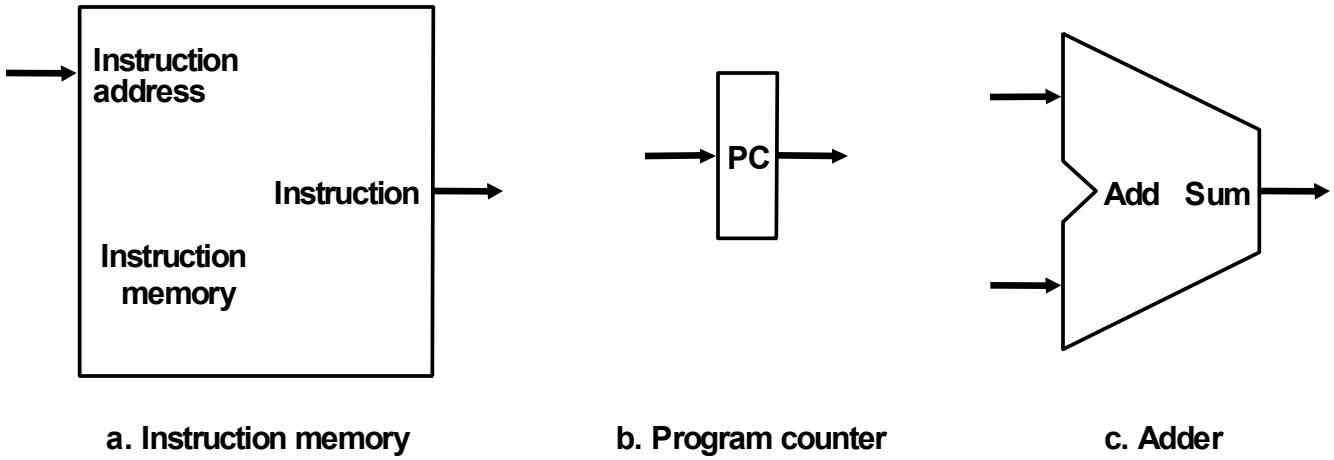




1. Building a Datapath

Basic Building Blocks

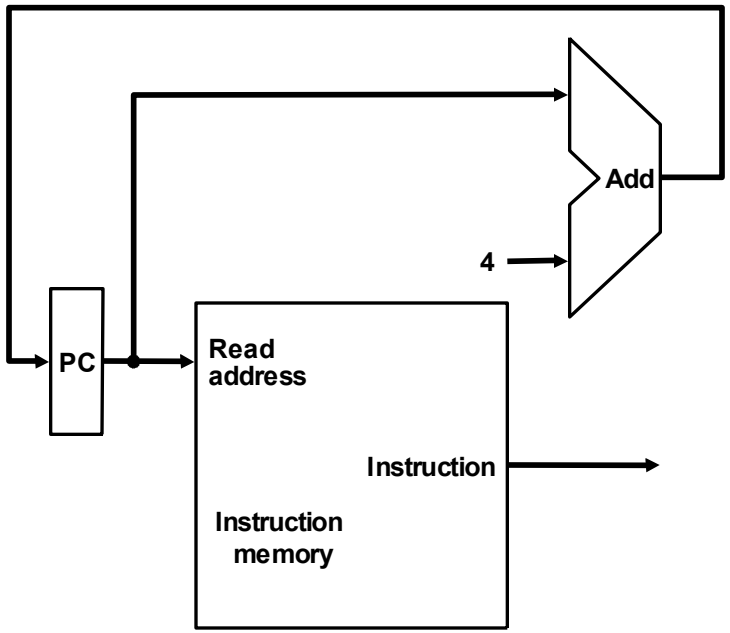
- **Instruction memory**:
  - A memory unit that stores the instructions of a program
  - Supplies an instruction given its address
- **Program counter**:
  - A register storing the address of the instruction being executed
- **Adder**:
  - A unit that increments PC to form the address of next instruction



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

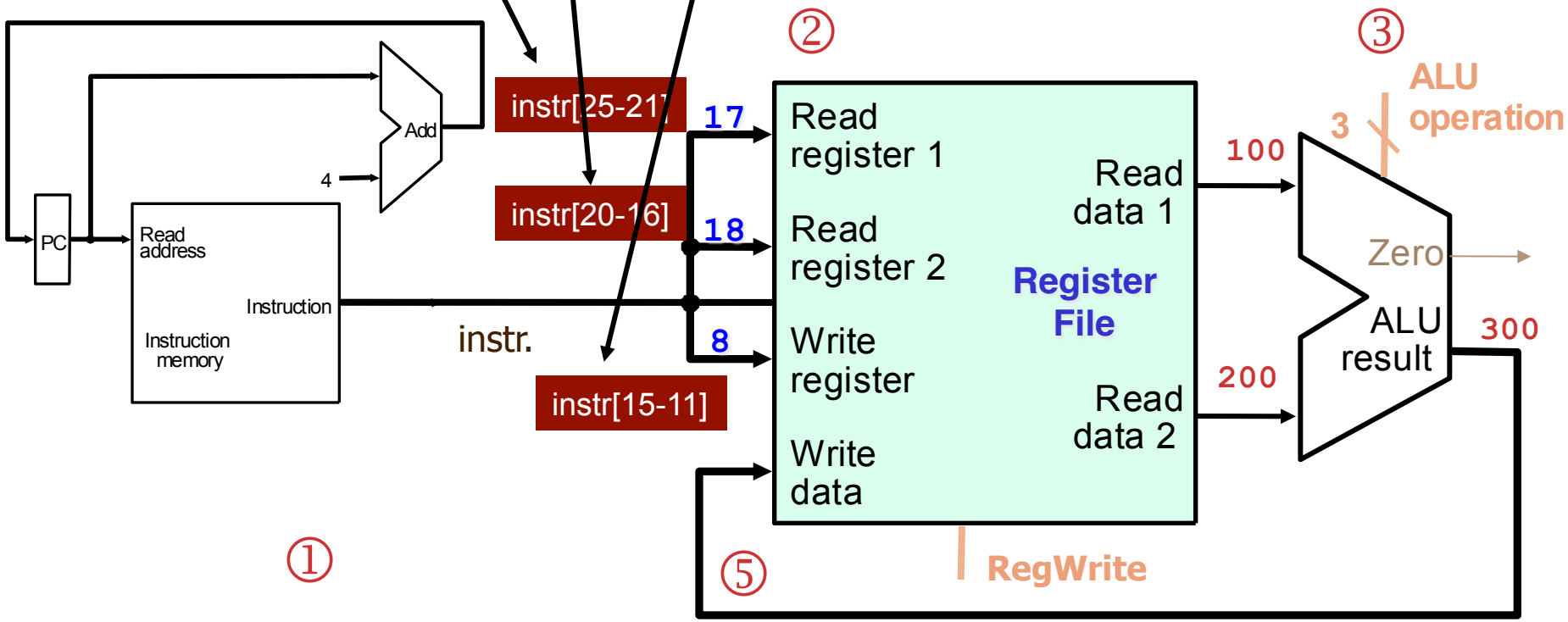


Datapath for Arithmetic/Logical (R-Type) Instr.

- **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



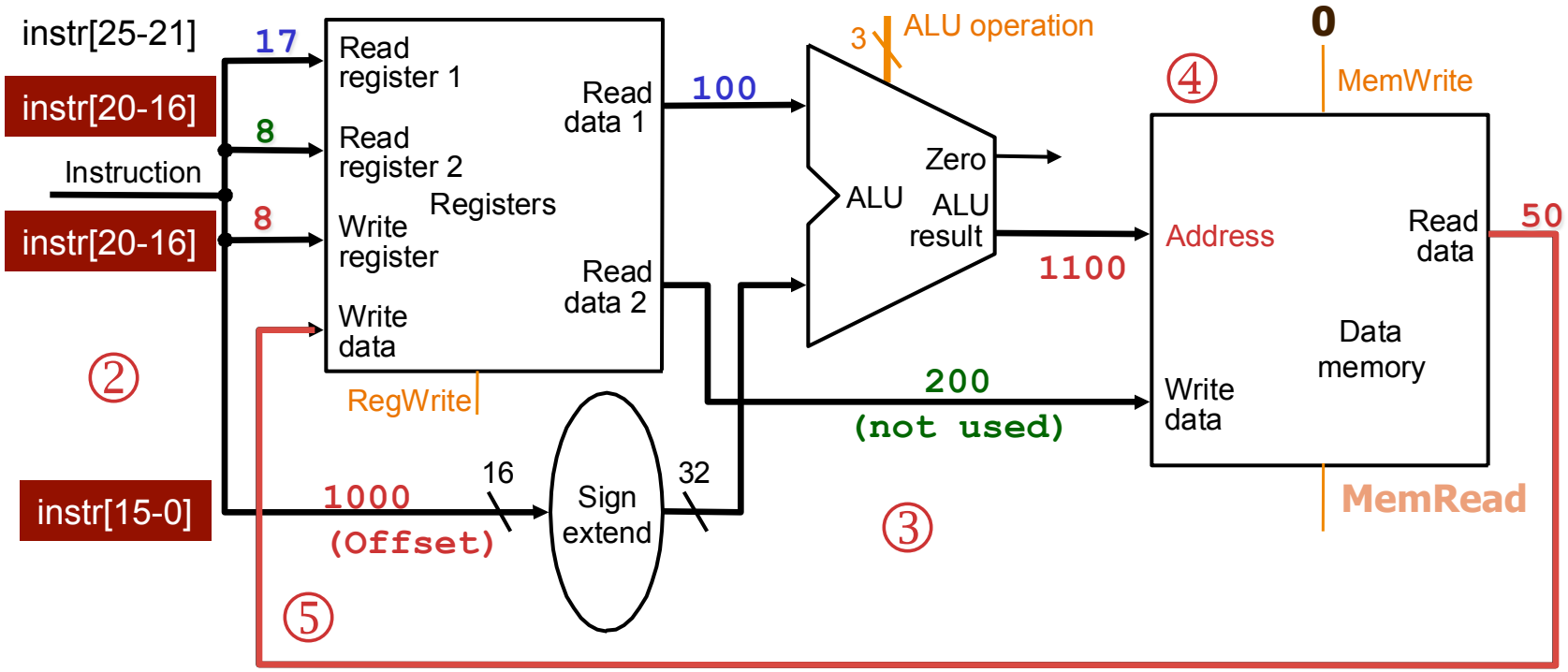
Datapath for Load/Store (I-Format) Instr.

- **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



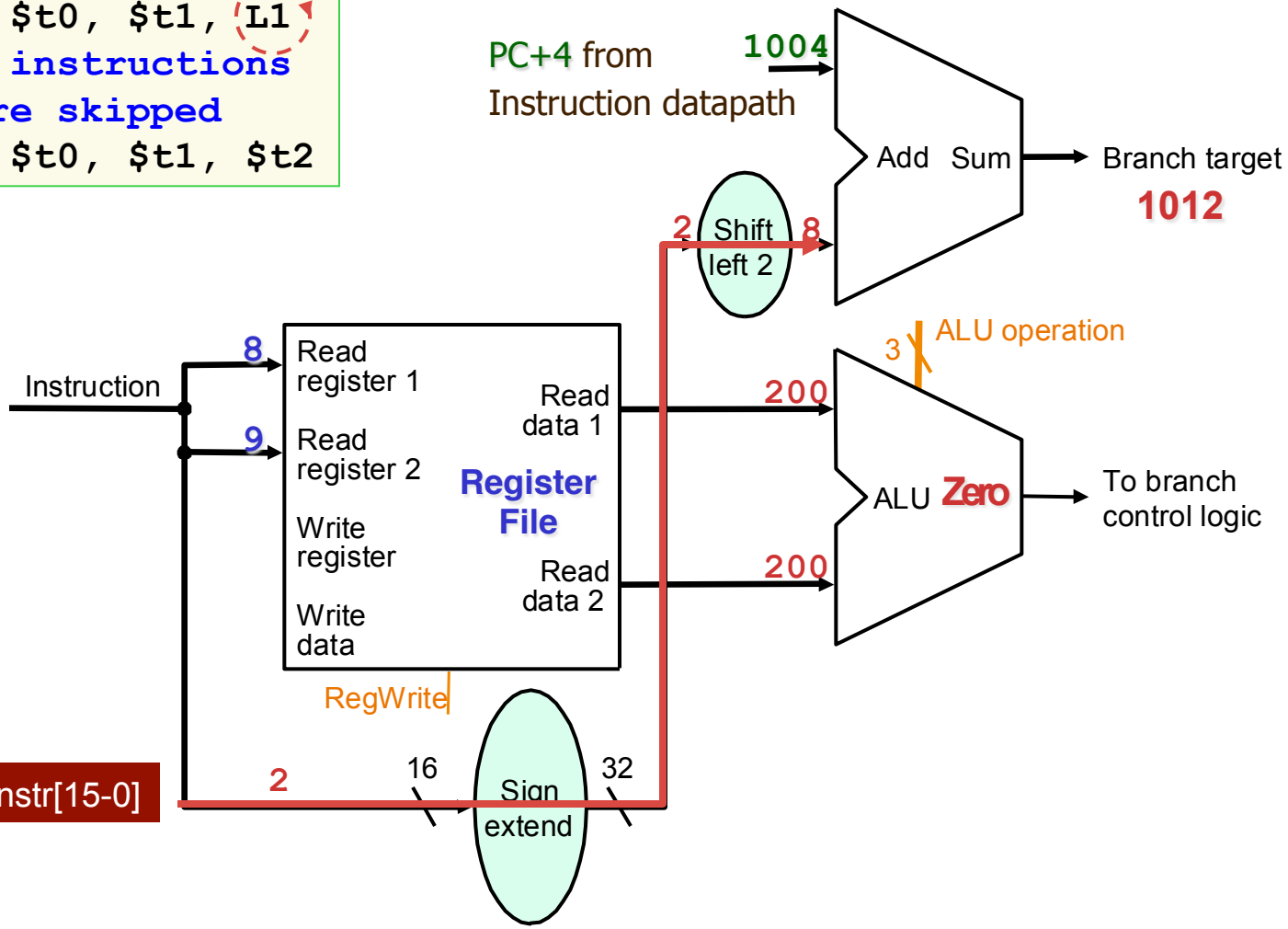
Note: For load word instruction, **MemWrite** has to be de-asserted so that the memory will not be modified by incoming write data.

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**

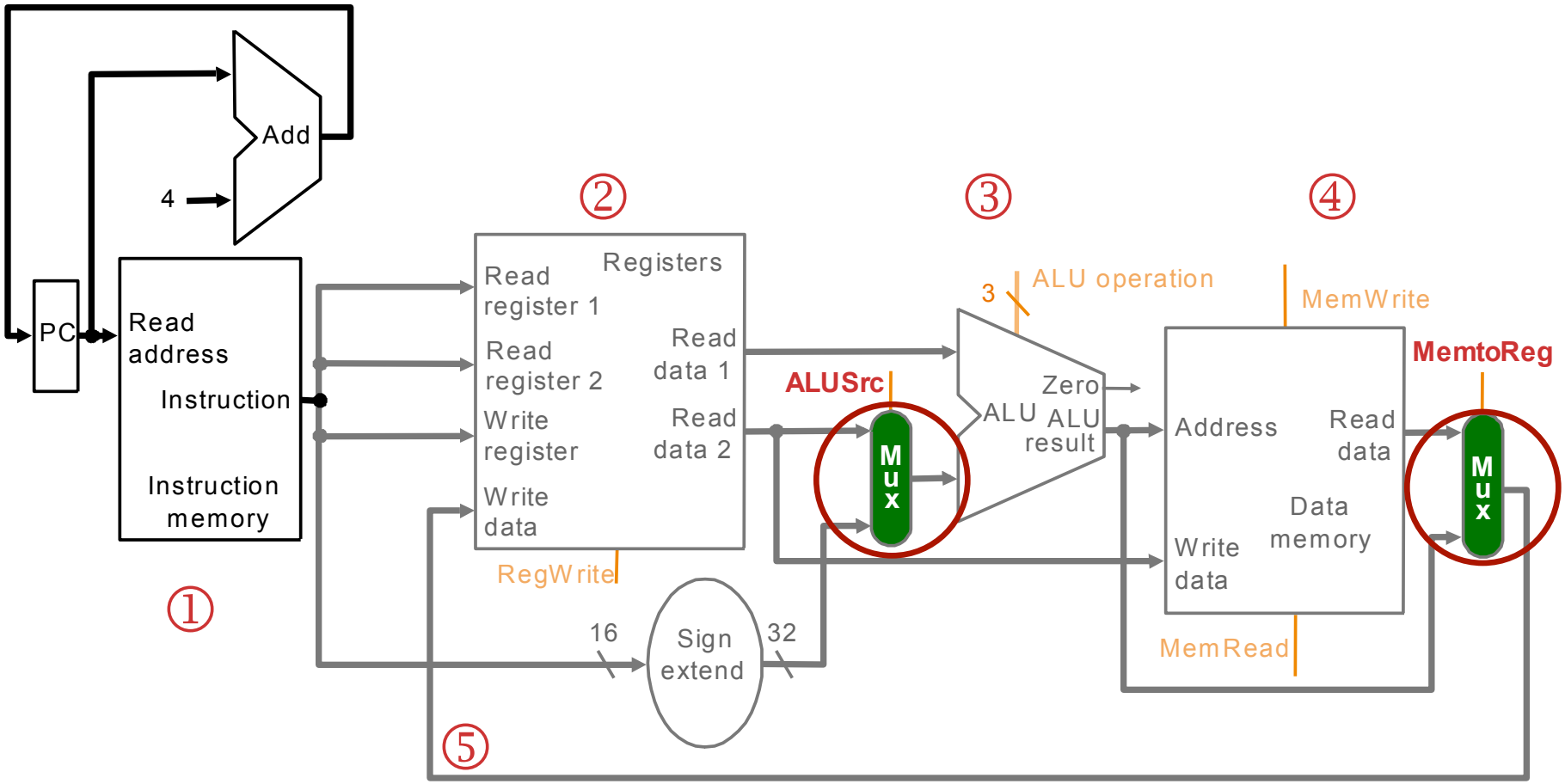
> Share some of the resources (e.g., ALU) among different instructions

Note:

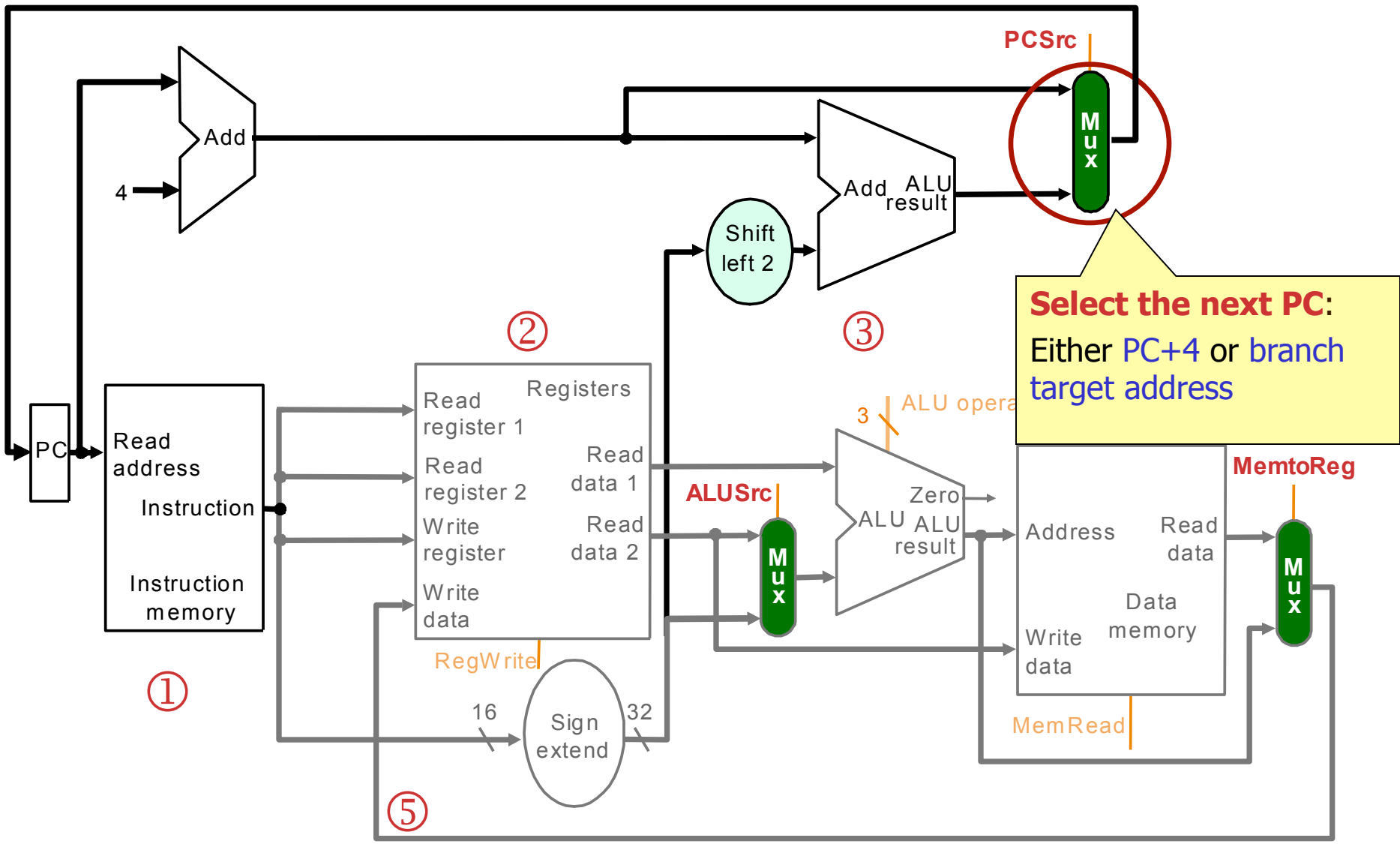
- This simple implementation is based on the (unrealistic) assumption
  - i.e. all instructions take just one clock cycle each to complete
- Implication:
  - No datapath resource can be used more than once per instruction
  - Any element needed more than once must be duplicated
  - ⇨ Instructions and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

Combined Datapath for R-Type & Memory Instr.

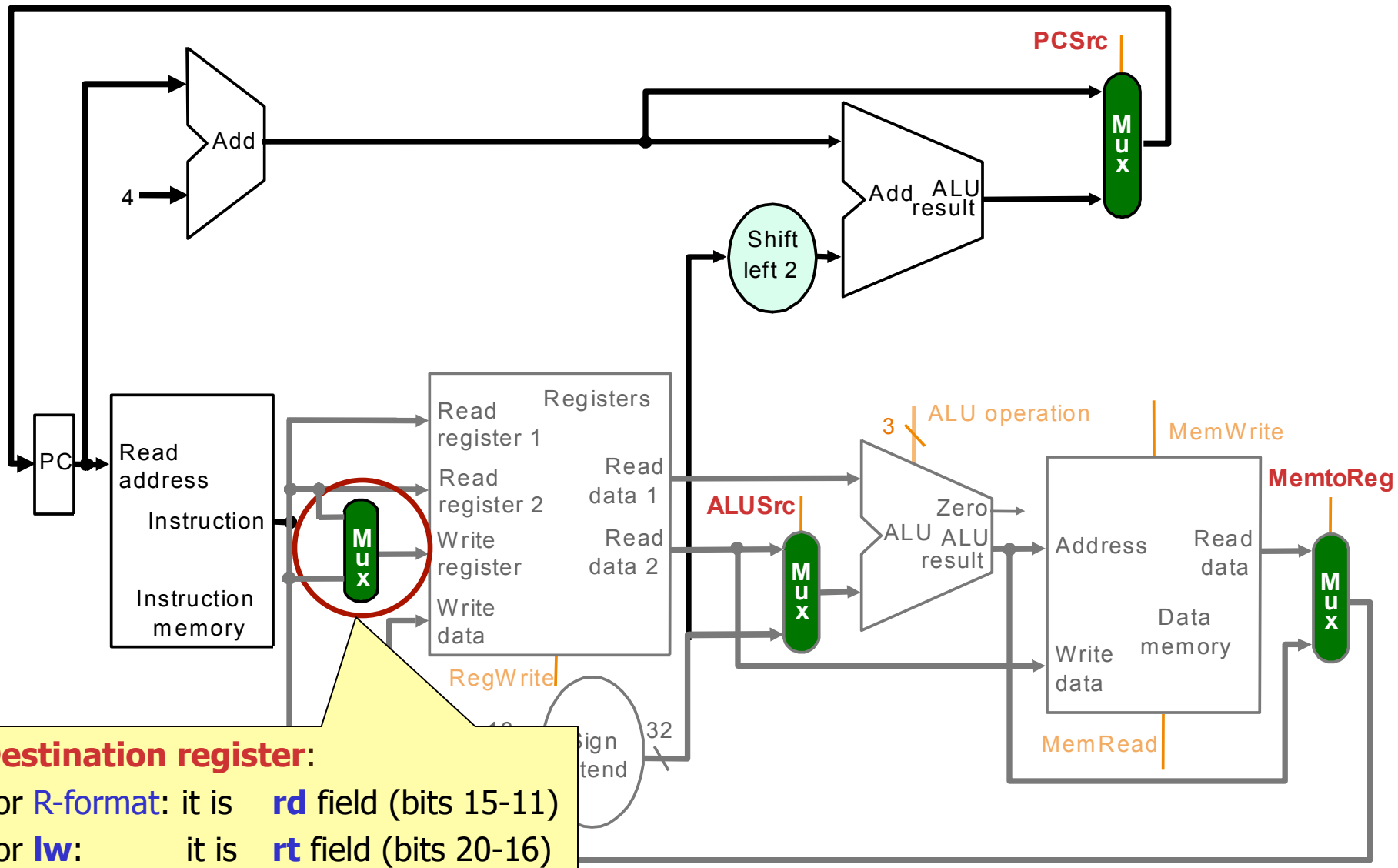
- Use **ALUSrc** to decide which source will be sent to the ALU
- Use **MemtoReg** to choose the source of output back to dest. register



Combined Datapath for Different Instr. Classes



Muxing Two Possible Destination Registers



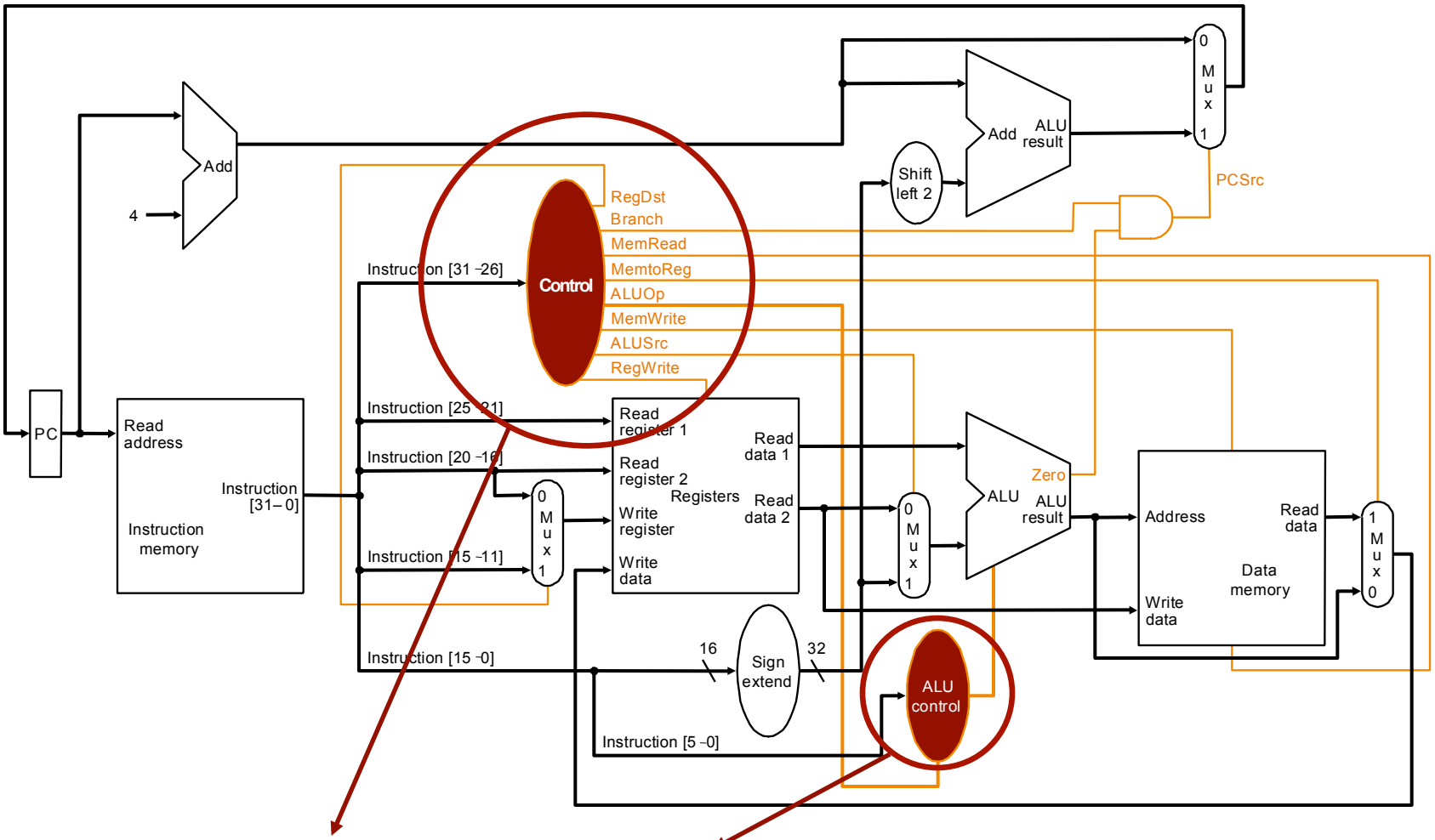
Destination register:

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

Control Unit

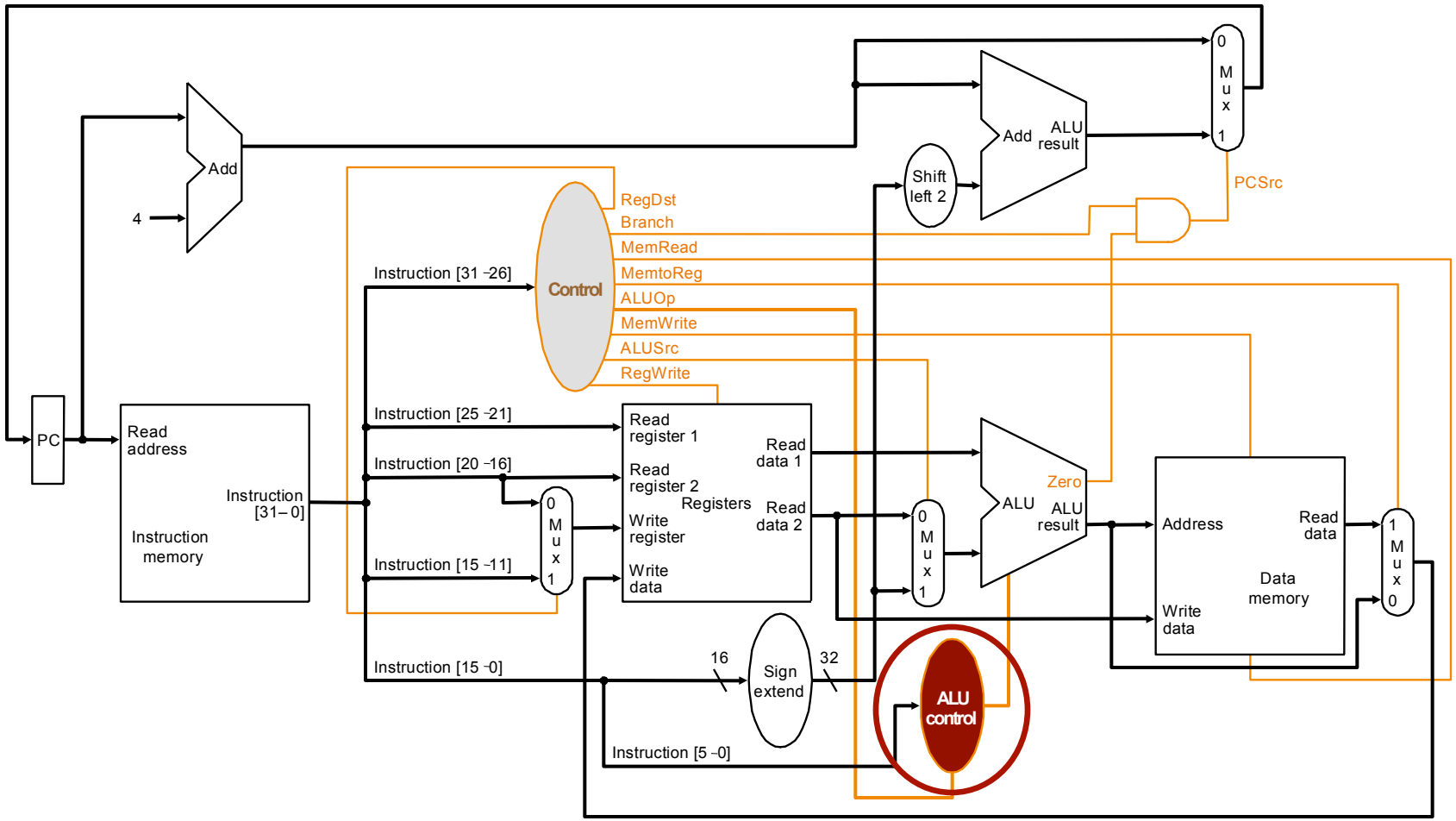
Datapath control unit controls the whole operation of the datapath

- How? Through **control signals**, e.g.
  - read/write signals for state elements: RegWrite, MemWrite, MemRead
  - selector inputs for multiplexors: ALUSrc, MemtoReg, PCSrc
  - 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

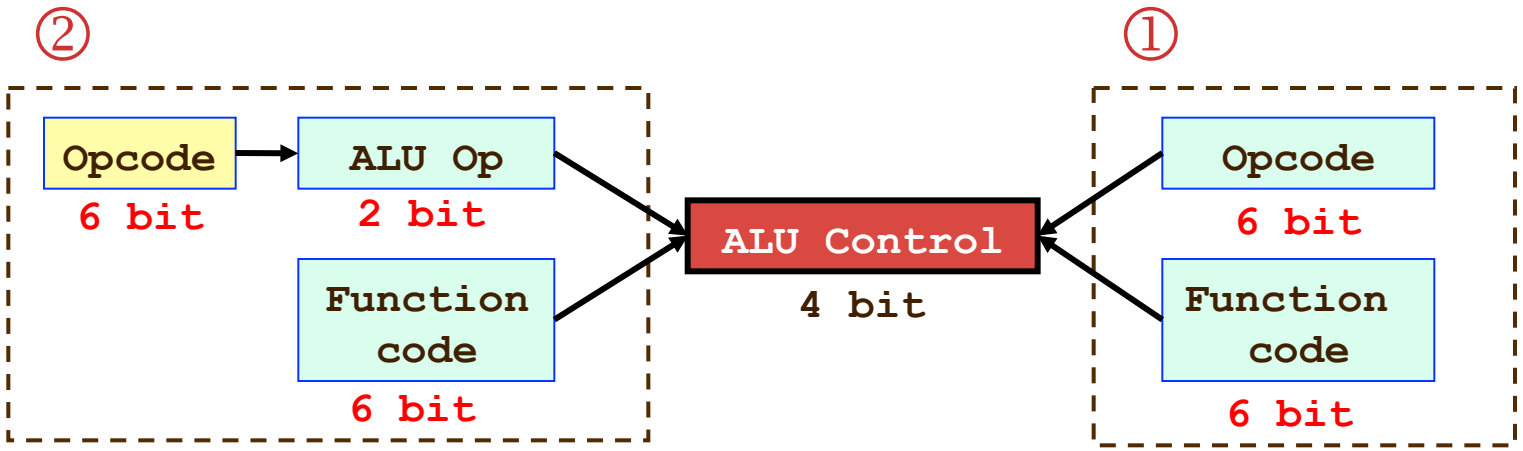
- The **ALU control** is part of the **datapath control unit**

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**

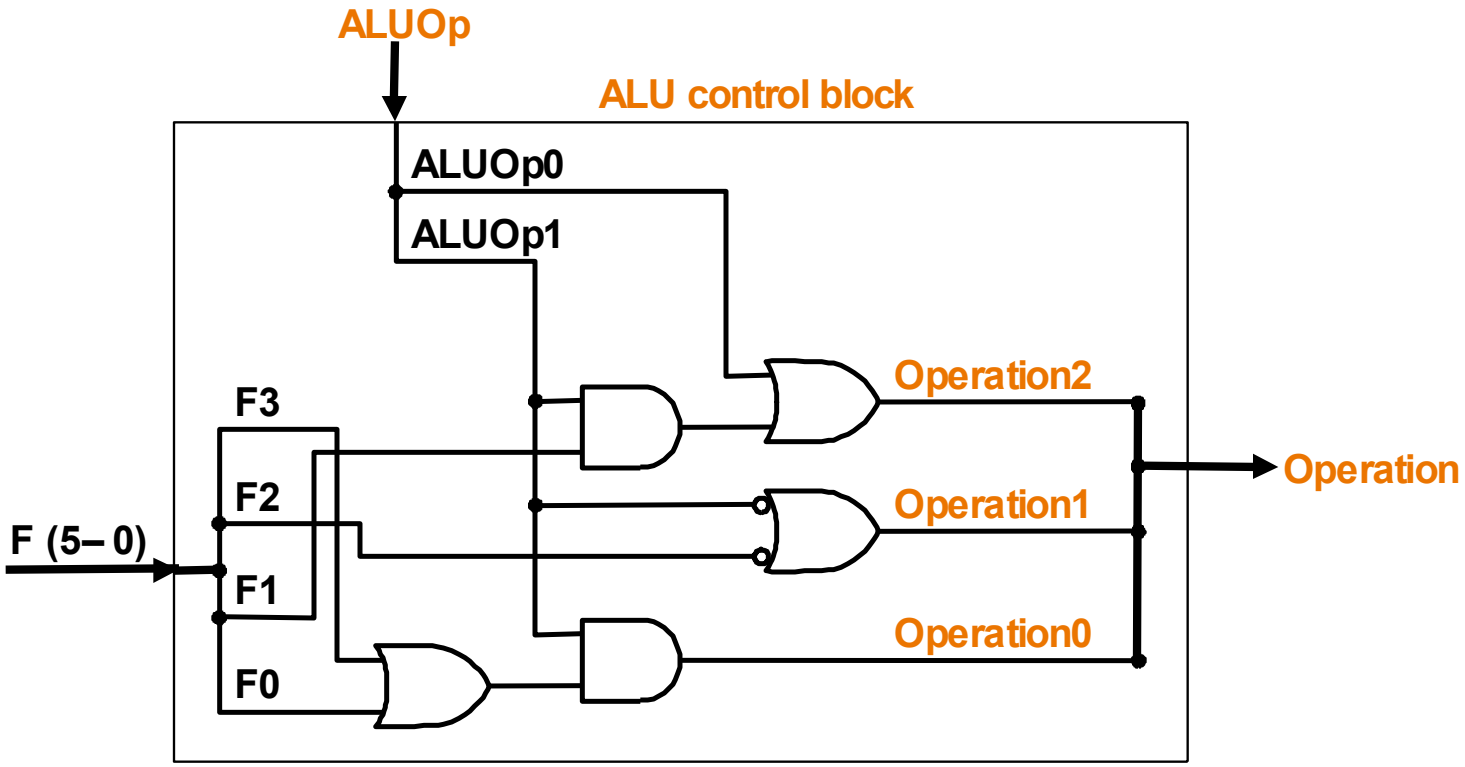
Implementing ALU Control Block

- 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

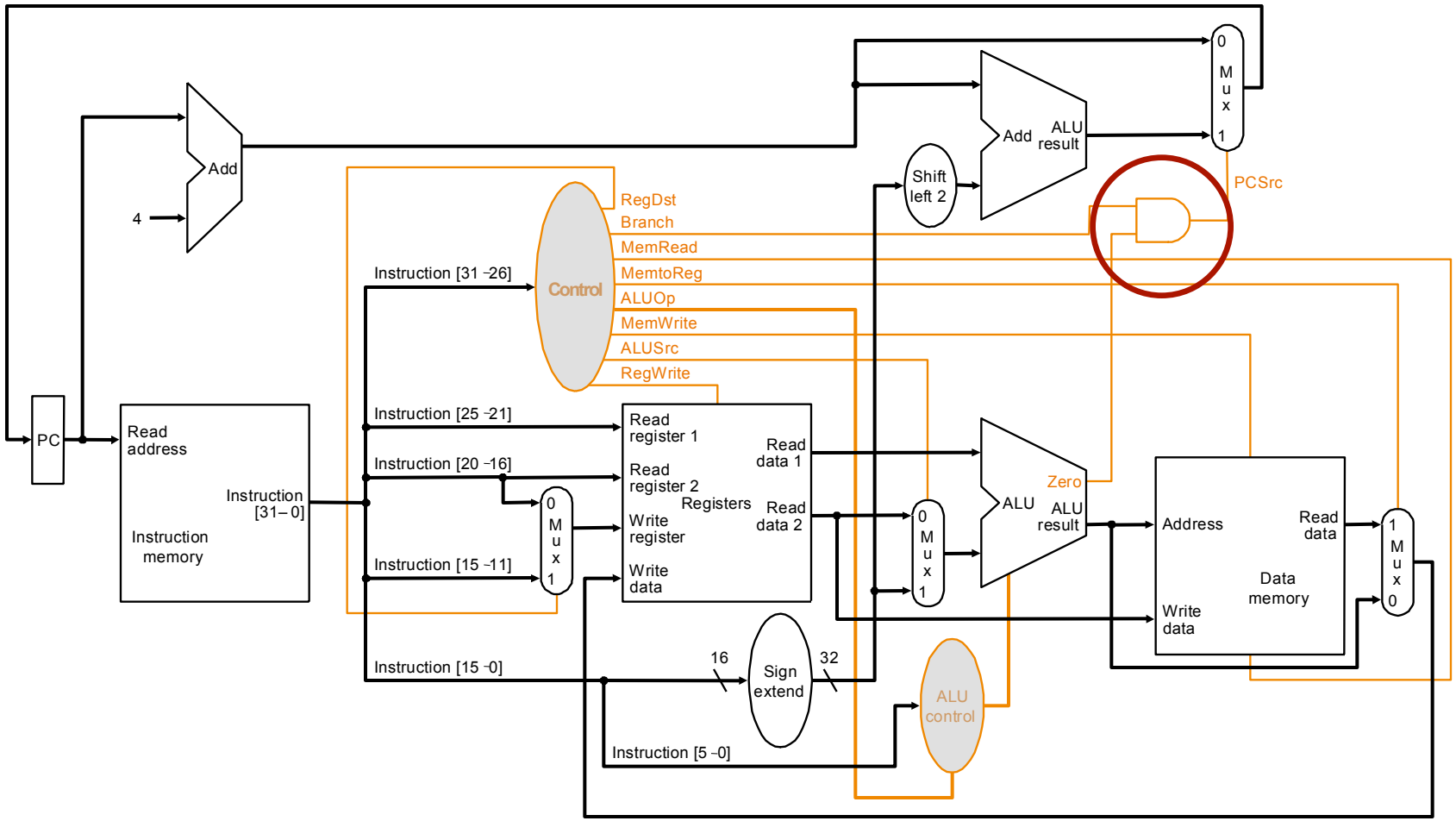
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from **rt** field (bits 20-16)	The register destination number for the Write register comes from **rd** field (bits 15-11)
RegWrite	None	Enable data write to the register specified 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):

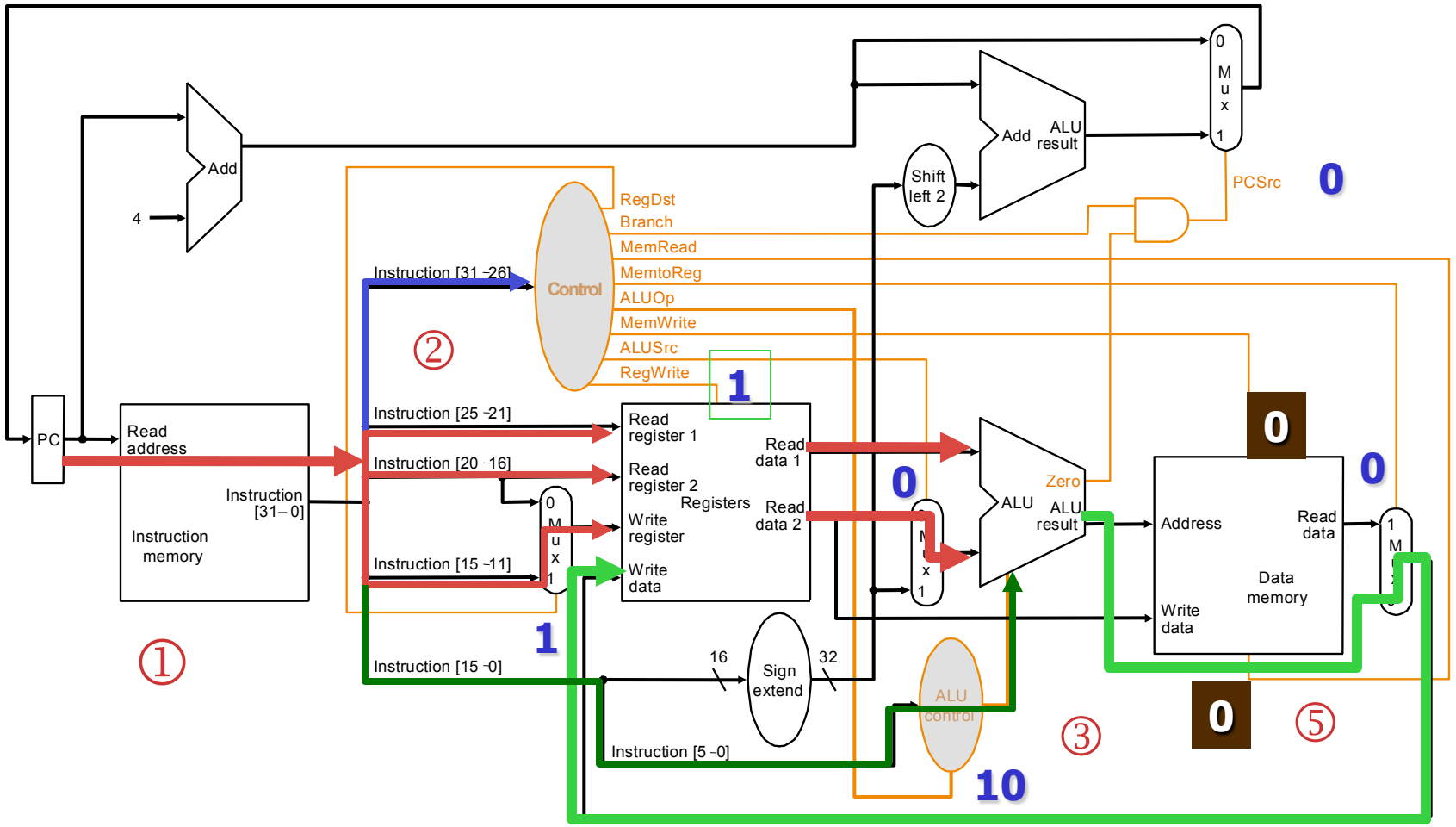
Instruction	Reg-Dst	ALU-Src	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

`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

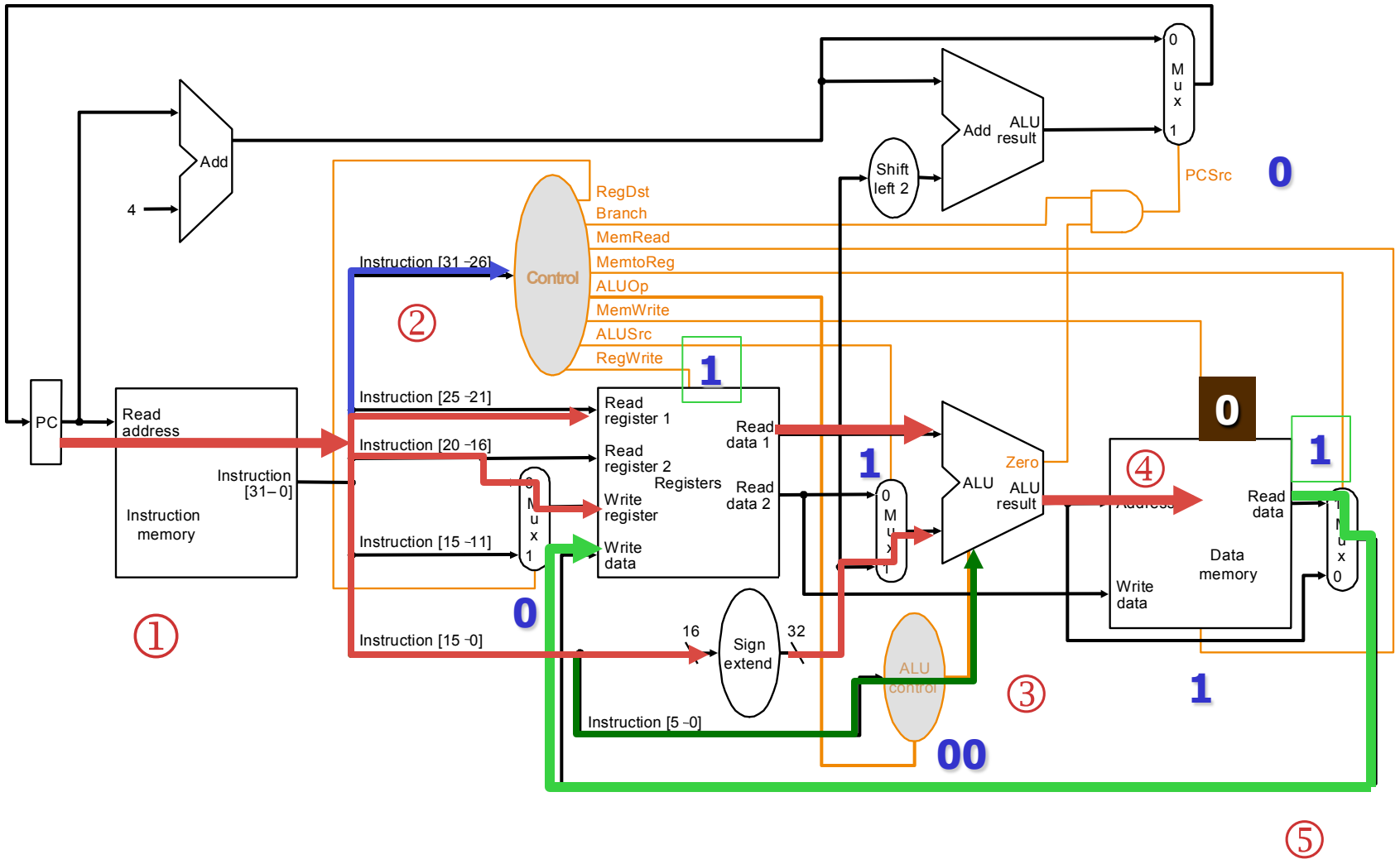
- Input to control unit (i.e. opcode determines setting of control lines):

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

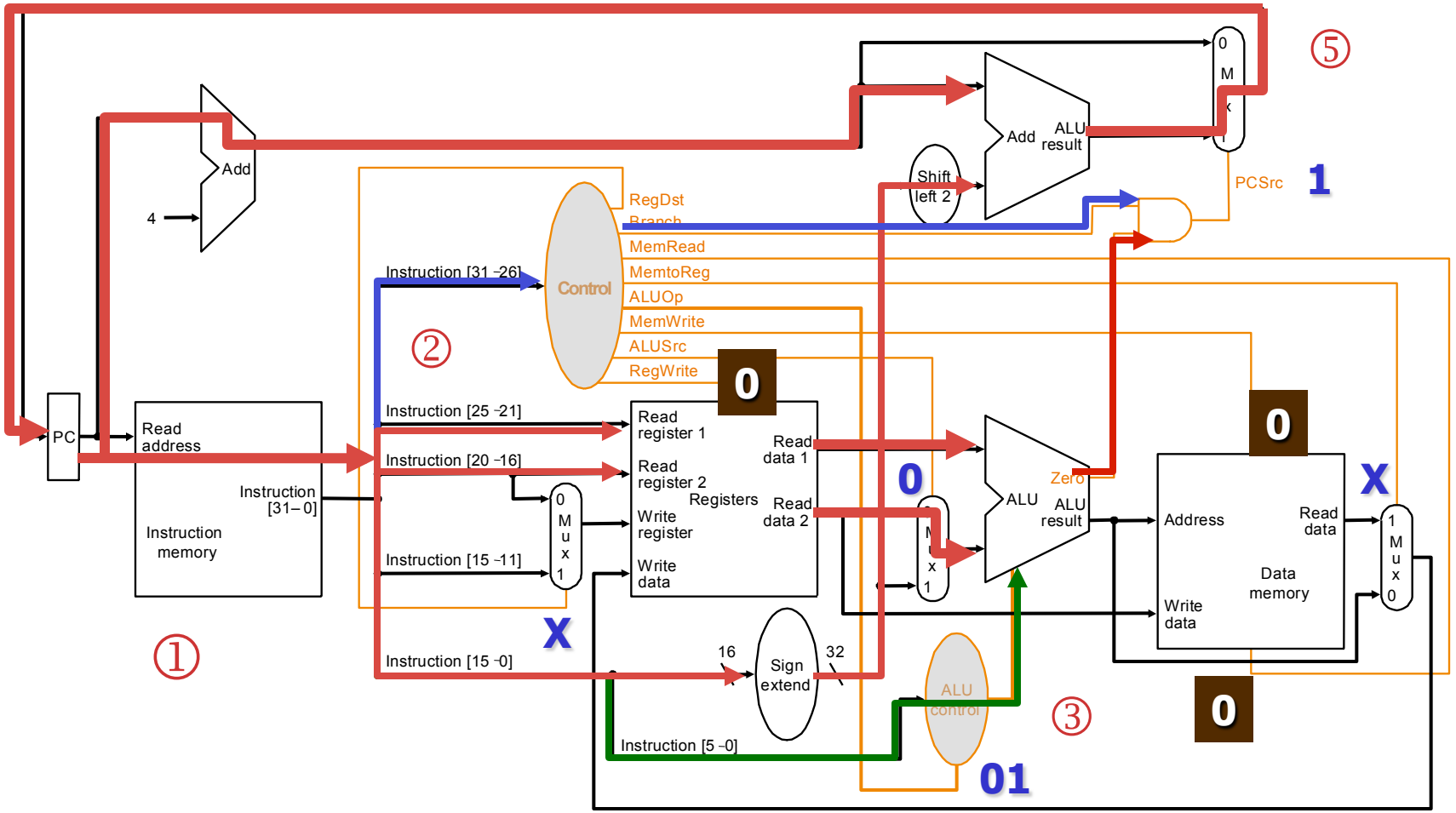
Review Datapath for R-Type Instr.



Review Datapath for Load Instr.



Review Datapath for Branch Instr.

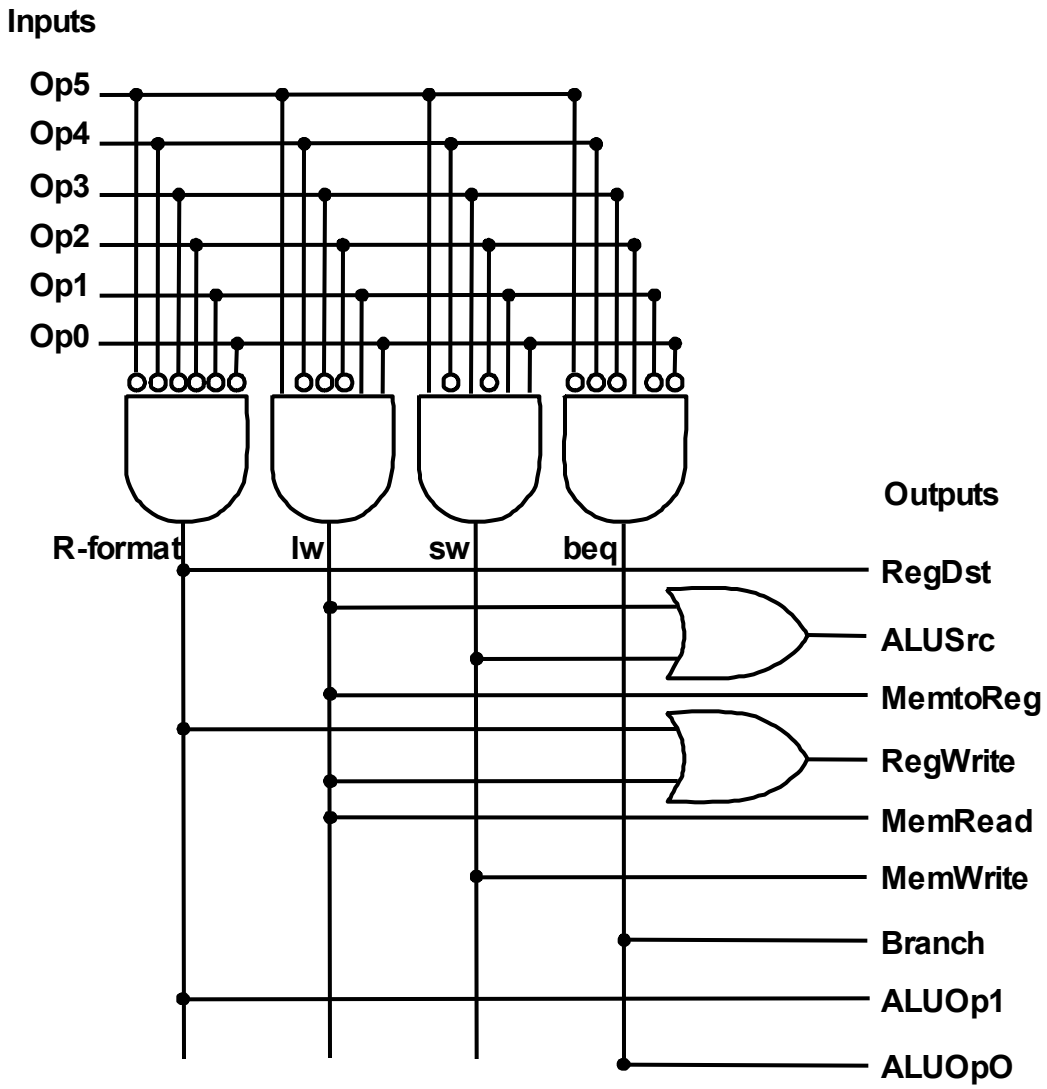


Implementing Datapath Control Unit

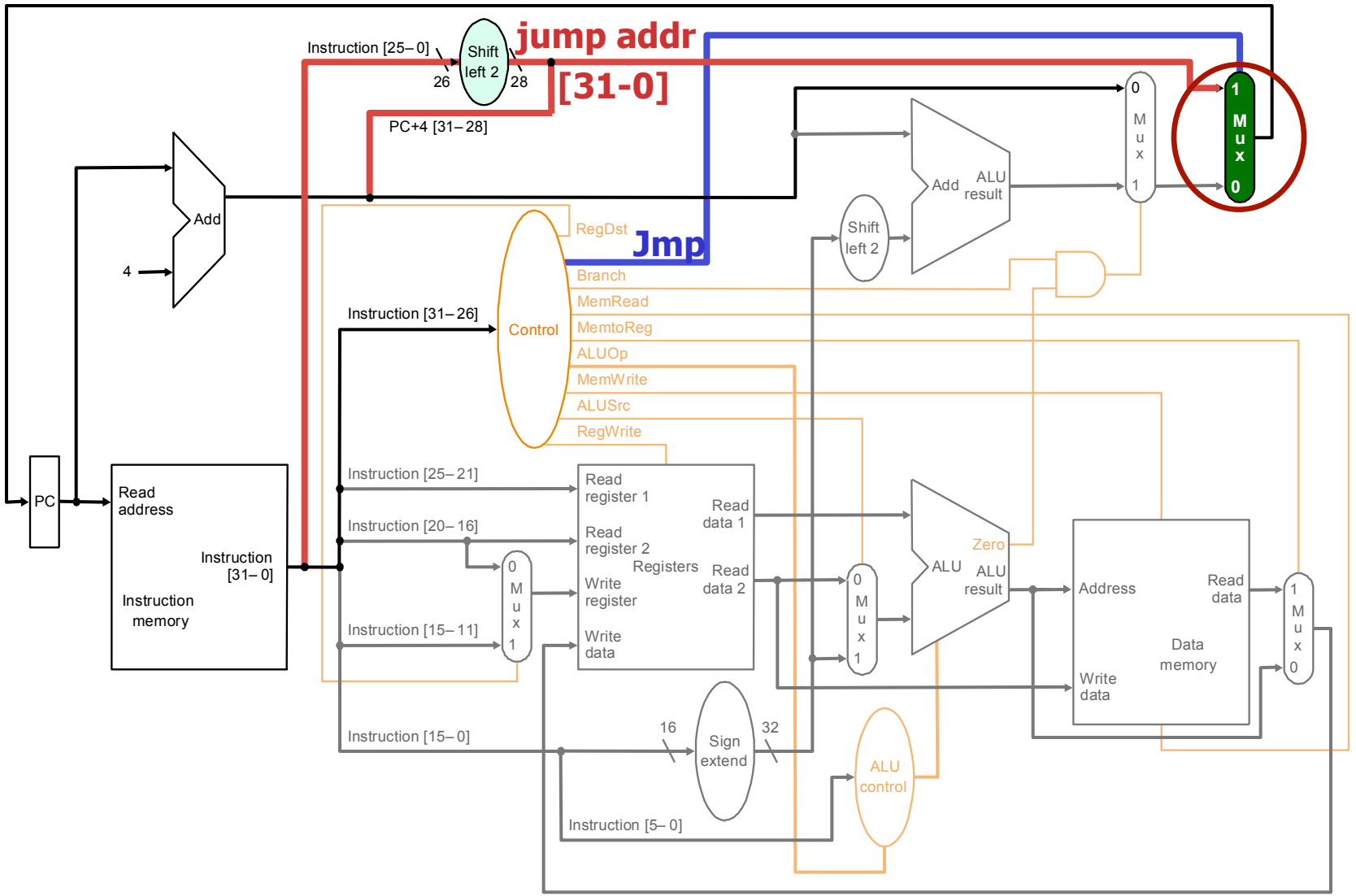
- Start with truth table

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.



Problems with Single-Cycle Datapath & Control

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

Can We Adopt Variable Clock Implementation?

Motivation:

- Many instructions do not involve all five functions units

Question:

- Why don't vary the clock for instructions with shorter execution time?

Answer:

- A variable-speed clock is very difficult to implement in hardware

The better solution:

Consider a **shorter clock cycle** and allow different types of instructions to take **different numbers of clock cycles**

(clock cycle is derived from the basic functional unit delays)

Multicycle Implementation

What does it mean by **multicycle**?

- An instruction takes multiple cycles to execute
- Execution of each instruction is broken into a series of steps
  1. Fetch the instruction
  2. Decode instruction & read the registers
  3. Perform ALU operation
  4. Memory access (if necessary)
  5. Write back the result
- Each step takes equal amount of time (one cycle) to complete

Why Multicycle?

Key advantage:

- Different types of instructions can have different number of cycles
  - Short instructions can complete early
    - e.g. R-type instruction can execute in 4 cycles instead of 5
  - Better performance as compared to the single-cycle implementation

Other advantage:

- Better sharing of function units ⇒ less hardware ⇒ cheaper
  - A single functional unit can be used more than once per instruction (as long as it is used at different clock cycles)
    - e.g. a single memory unit is used for both instructions and data
    - e.g. need only a single ALU, rather than one ALU and two adders

Trade-off

As compared to single-cycle implementation, multicycle approach needs

a. More temporary storage to hold values between clocks

   i.e. one or more registers added after every major functional unit

   - Instruction registers (IR)
   - Memory data registers (MDR)
   - Registers A & B
   - ALUOut

   But, why? (will explain in next couple of slides)

a. More complex control

   - Multiplexors
   - Control signals

Question: Worth adding all these?

Example: Single-Cycle CPU vs. Multicycle CPU

The instruction mix of a program:

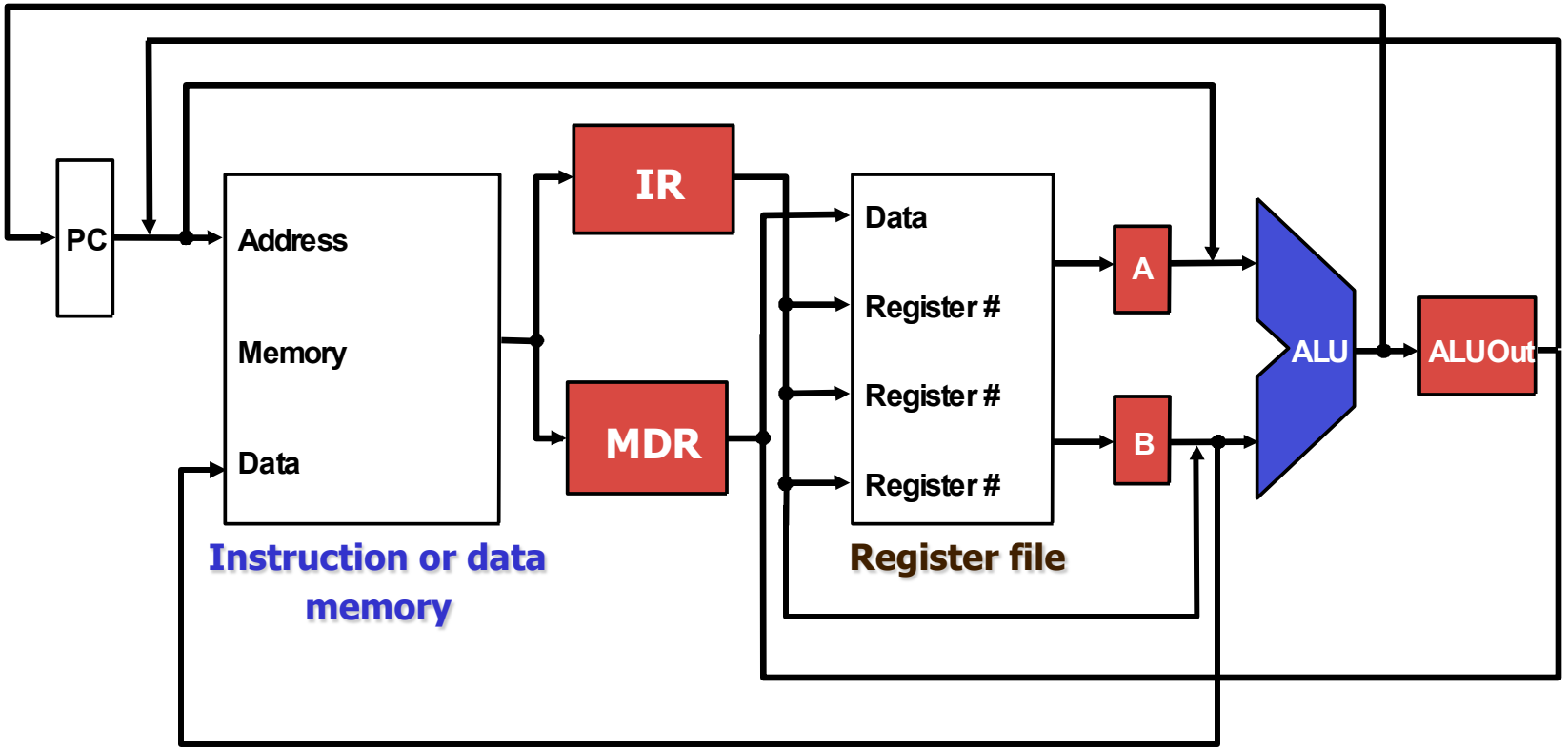
	Load	Store	Branch	Jump	ALU
# of instructions	25%	10%	11%	2%	52%
# of clocks/instr.	5	4	3	3	4

- Clock cycle of single-cycle CPU is **5x** of clock cycle of multicycle CPU
- For multicycle CPU (assuming the clock cycle is **R**)
  - CPI_{multi} = 0.25*5 + 0.10*4 + 0.11*3 + 0.02*3 + 0.52*4 = 4.12
  - $Time_{multi}$ = CPI_{multi} * $clock\ cycle_{multi}$ = 4.12 * **R** = **4.12R**
- For single-cycle CPU
  - CPI_{single} = 1.0
  - $Time_{single}$ = CPI_{single} * $clock\ cycle_{single}$ = 1.0 * **5R** = **5R**

5R / 4.12R = 1.21 ⇒ multicycle CPU speeds up by 21%

Overview of the Multicycle Datapath

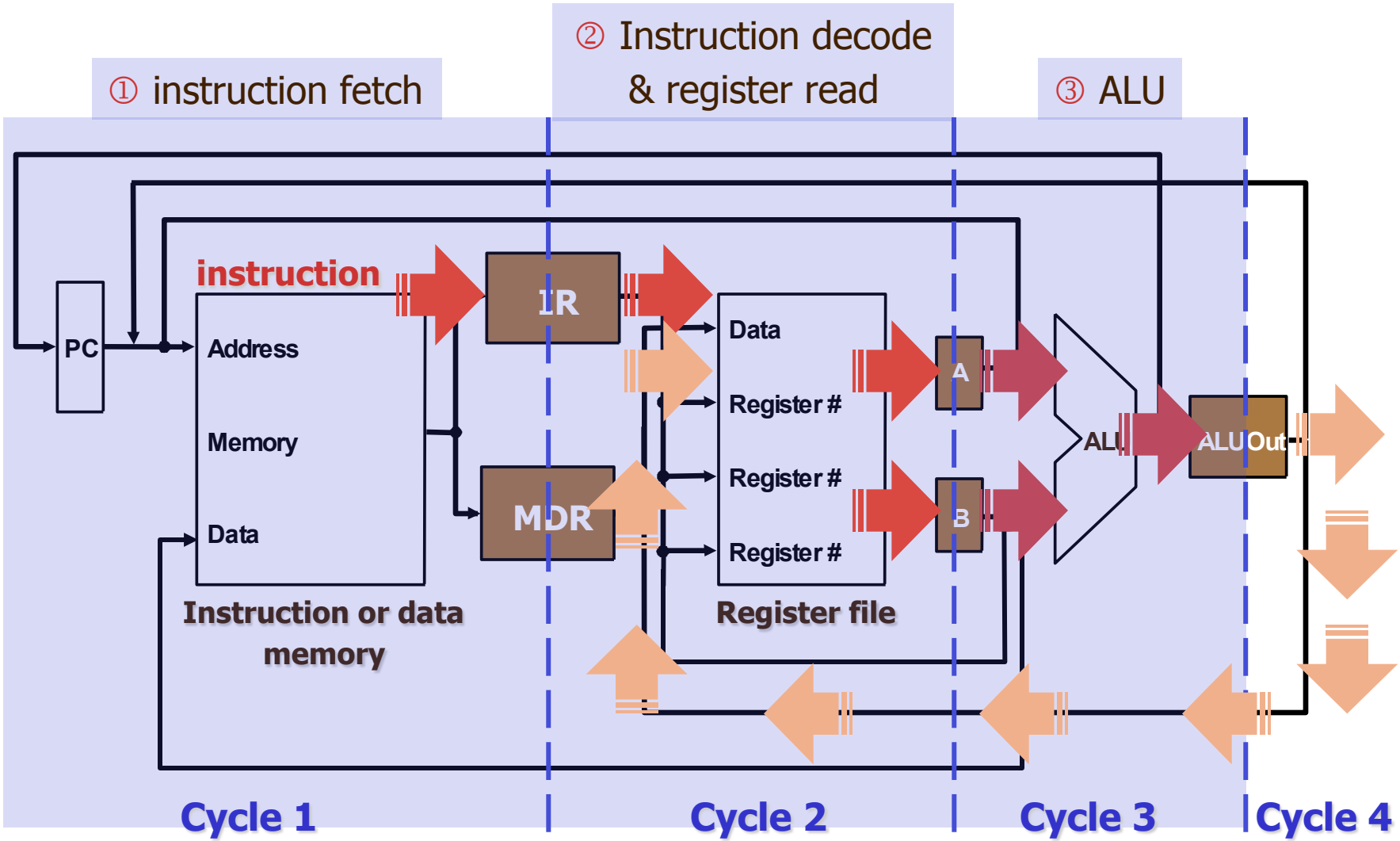
- **IR** ≡ Instruction Register; to hold instruction from memory
- **MDR** ≡ Memory Data Register; to hold data from memory
- **A, B**: registers to hold operand values from register file
- **ALUOut**: register to hold the output of the ALU



Blue: hardware being shared, Red: hardware added

Overview: How Do These Buffer Work?

- Let's use R-type instruction as an example



More About Temporary Storage

Memory is connected to both IR and MDR

- Output of the memory goes to both of them at the same time
- It seems like
  - Instruction read will destroy the value stored in MDR, or
  - Data read will destroy the instruction stored in IR
- Does it matter? How to resolve these problems?

ALUOut

- Holds the ALU's output (which will serve as an input for next cycle)
- The ALU's output can be either
  - Data (to be written to register or memory), or
  - Branch or jump target address

Need for Additional Control Signals

- **All except IR** hold data only between a pair of adjacent clock cycles
  - Thus, **MDR**, **A**, **B**, **ALUOut** **do not need a write control signal**

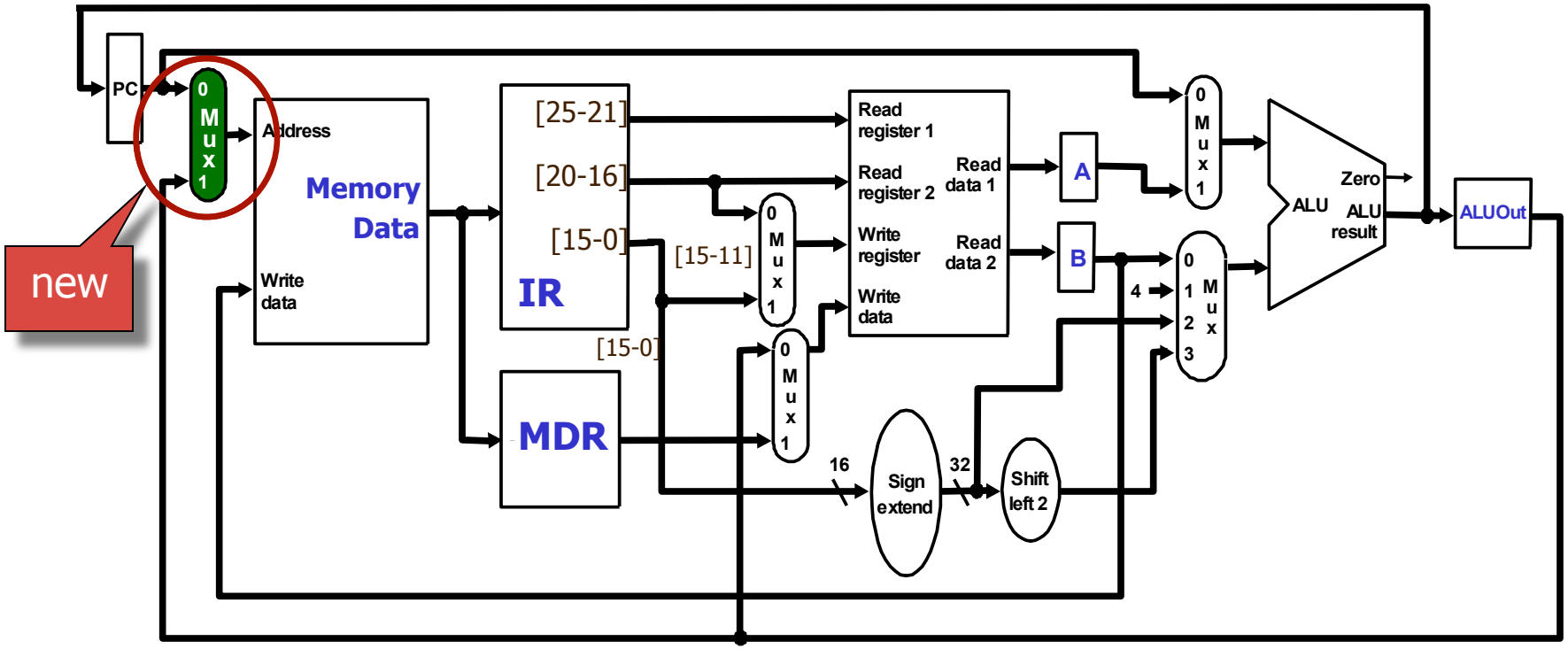
- **IR** needs to hold the instruction until the end of instruction execution
  - Thus, it **requires a write control signal**

- Since several functional units are shared for different purposes,
  - **Multiplexors** have to be added or expanded
  - Thus, **additional control signals are needed**
    - See next couple of slides for new multiplexors and control signals

Multiplexing Addresses to Read the Memory

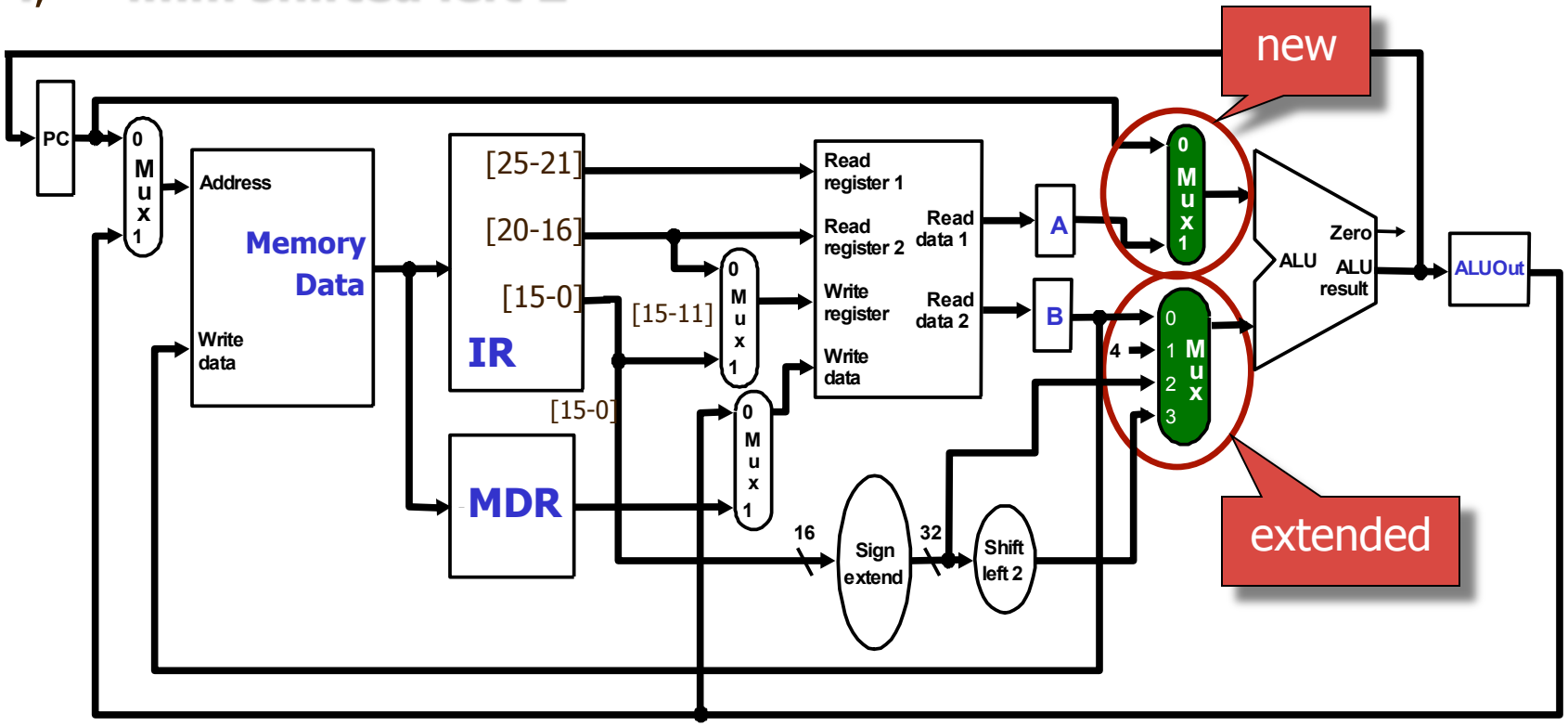
- Memory is shared by both instructions and data
  - The address to the memory module is no longer just **PC**
  - It can be either **instruction address** (PC) or **data address**

⇒ a **multiplexor** is needed to choose the source



Computation of Next PC Using the ALU

- The adders to compute next PC in single-cycle approach are removed
- So, generation of next PC and execution of operation share the ALU
  - The first source to the ALU can be either **A** or **PC**
  - The second source to the ALU can be either **B**, **sign-extended imm**, **4**, or **imm shifted left 2**



Constraints for Multicycle Datapath

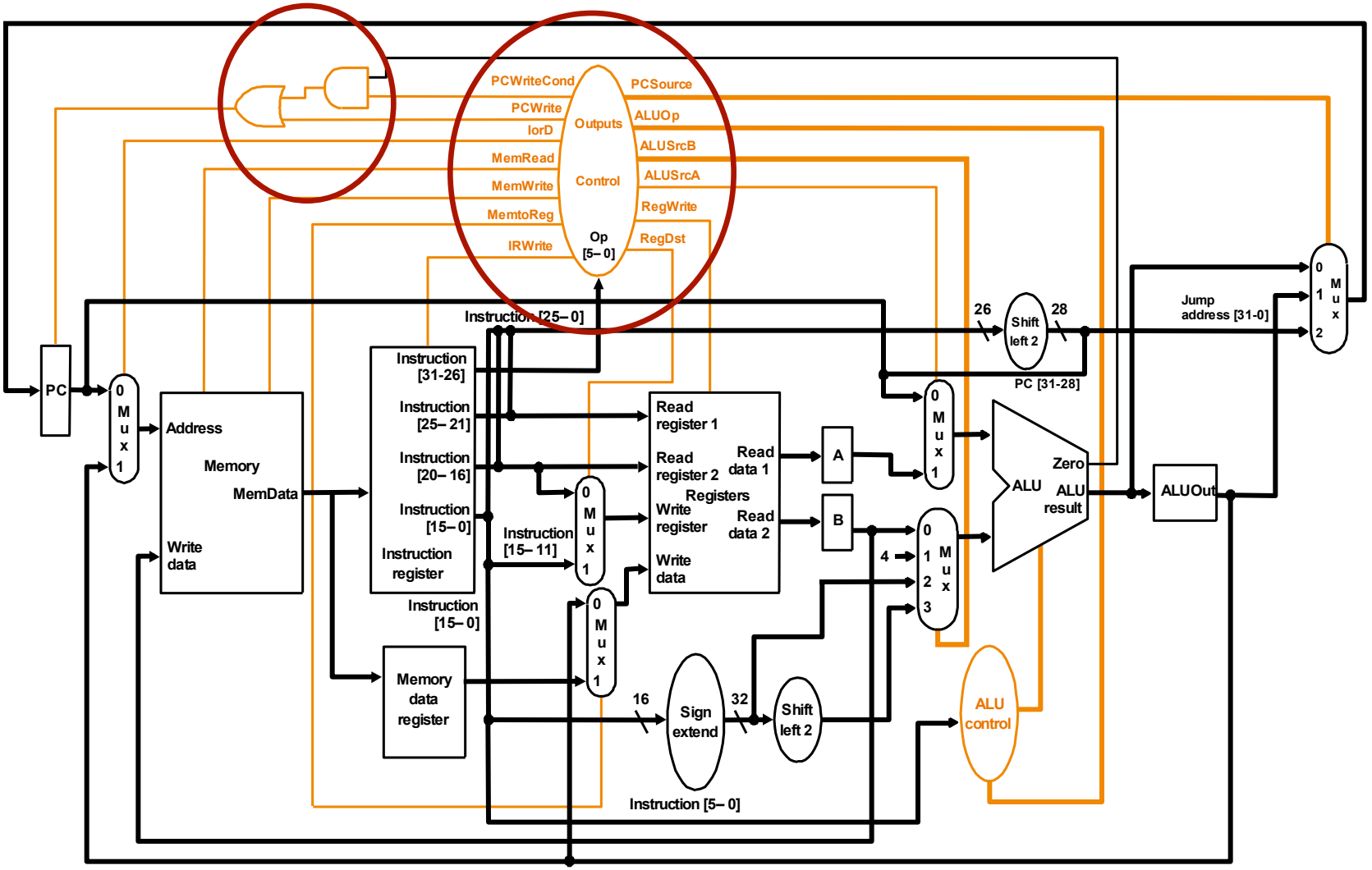
Only ONE ALU operation can be performed at each cycle

- Will calculation of next PC conflict with other ALU operation?
  - If yes, what should we do then?
  - If no, why?

Only ONE register file access can be performed at each cycle

- Will read from **src** registers conflict with write to **dest** register?
  - If yes, what should we do then?
  - If no, why?

Datapath & Control for Multicycle Implementation



3. Multicycle Implementation

1. Instruction Fetch

Hardware operations:

```
IR = Memory[PC];
PC = PC + 4;
```

- Send the PC to the memory as address
- Read an instruction from memory
- Write the instruction into the IR
- Increment the PC by 4 without violating the ALU usage constraints
  - Hence, hide the time required to compute PC+4
  - But, it seems like we are reading & writing PC in the same cycle!
  - Would there be any race condition? How to resolve it?

Hardware operations:

(Assume the existence of two registers and an offset field,
no harm to do the computation early even if they do not exist)

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- Read **rs** & **rt** from register file and store them to registers **A** & **B**
- Compute the branch target address (sign extension and left shift)
- Store the branch target address in ALUOut
  - If the instruction is actually a branch, ALUOut will be ready to provide the branch target address during next cycle
  - Again, we hide the time (i.e. avoid extra cycle) required to find the branch target address

Hardware operations:

a) ALU operation, or

b) Memory address computation, or

c) Branch completion

We'll look at them one by one in subsequent slides

Hardware operations (for R-format):

```
ALUOut = A op B;
```

- Send both registers A and B to the ALU
- Perform ALU operation specified by the function code on A & B
- Store the result in ALUOut

Hardware operations (for lw and sw):

```
ALUOut = A + sign-extend(IR[15-0]);
```

- Sign-extend the 16-bit offset to a 32-bit value
- Send both register A and the 32-bit offset to the ALU
- Store the result in ALUOut

Hardware operations: (for condition branch)

```
if (A == B) PC = ALUOut;
```

- Send both registers A and B to the ALU for comparison
  - Comparison is done by subtraction
- Set the PC to the branch target address if A and B are equal
  - Look at the output Zero from ALU to tell if equal
- If Zero = 1, write ALUOut computed during instruction decode to PC

Hardware operations: (for unconditional branch)

```
PC = PC[31-28] || (IR[25-0] << 2);
```

- Compute the jump address
  - Left-shift the 26-bit address field by 2 bits to give a 28-bit value
- Concatenate the 4 leftmost bits of the PC with the 28-bit value to form a 32-bit jump address
- Write the jump address to the PC

4. Memory Access or Completion

Hardware operations:

a) Memory access (load / store), or

b) Instruction completion (for all R-type and some I-type)

We'll look at them one by one in subsequent slides

Hardware operations:

```
MDR = Memory[ALUOut];     // for load instruction
   or
Memory[ALUOut] = B;       // for store instruction
```

- Address was computed and stored in ALUOut during previous step
- For a `load` instruction:
  - Retrieve a data word from memory with the specified address
  - Then, put the data word in `MDR`
- For a `store` instruction:
  - Write data word stored in `B` to memory at the specified address

Hardware operations: (for R-type instructions)

```
Reg[IR[15-11]] = ALUOut;
```

- Get the ALUOut value computed during previous step
- Write the value into destination register in the register file

5. Memory Read Completion

Hardware operations:

```
Reg[IR[20-16]] <= MDR;
```

- Get the data value stored in MDR during previous step
- Write the value into destination register in the register file

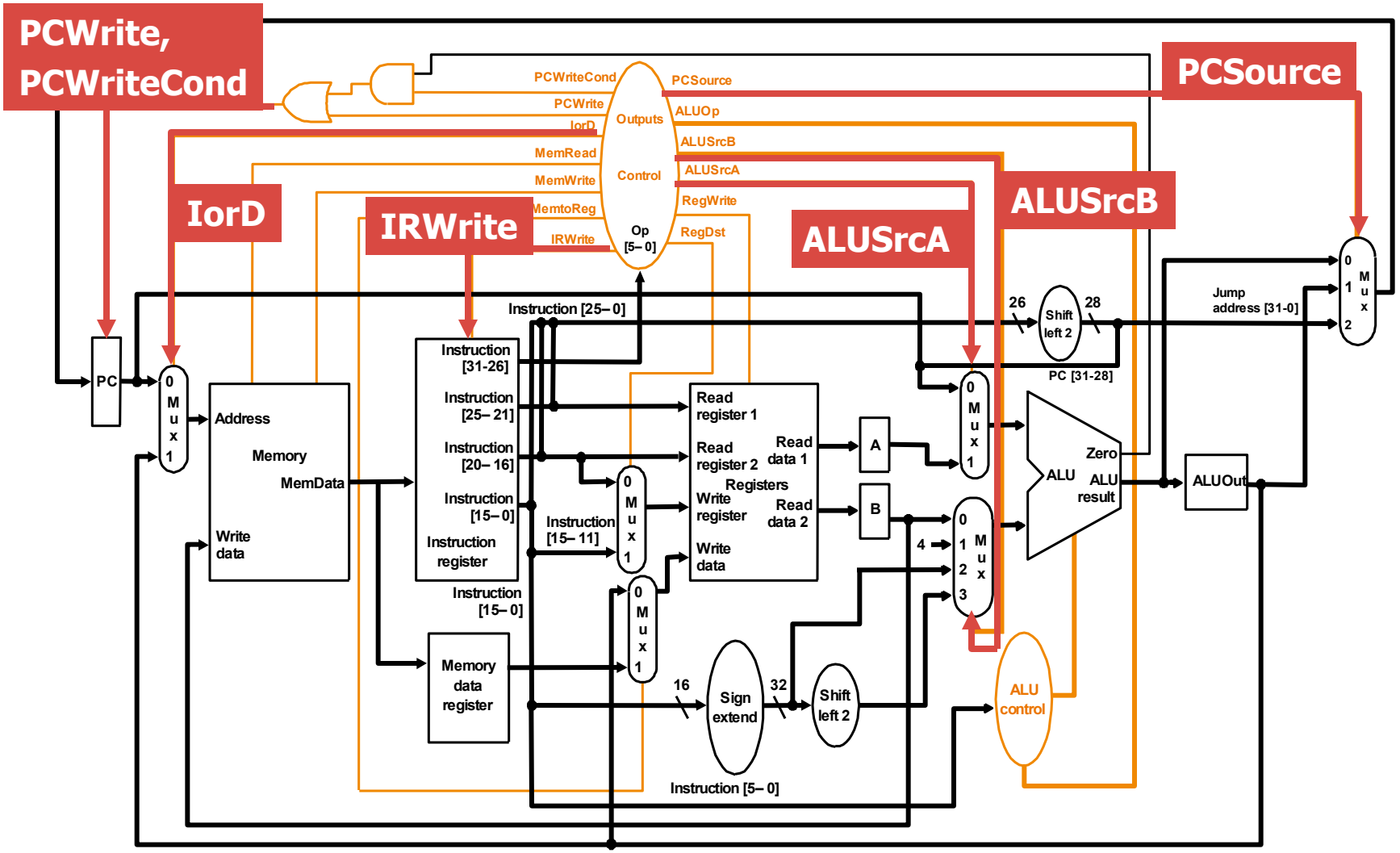
Summary of Execution Steps

	Actions for instructions			
Step name	**R-type**	**Memory references**	**Branches**	**Jumps**
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode / register fetch	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC + (sign-extend(IR[15-0]) << 2)			
Execution, addr comp., branch/jump completion	ALUOut = A op B	ALUOut = A + sign-extend(IR[15-0])	If (A == B) PC = ALUOut	PC = PC[31-28] || (IR[25-0] << 2)
Memory access, R-type completion	Reg[IR[15-11]] = ALUOut	Load: MDR=Memory[ALUOut]		
		Store: Memory[ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		
Total steps	**4**	**load: 5; store: 4**	**3**	**3**

Note: PC can be potentially modified at either step 1 or 3

4. Control Unit Design

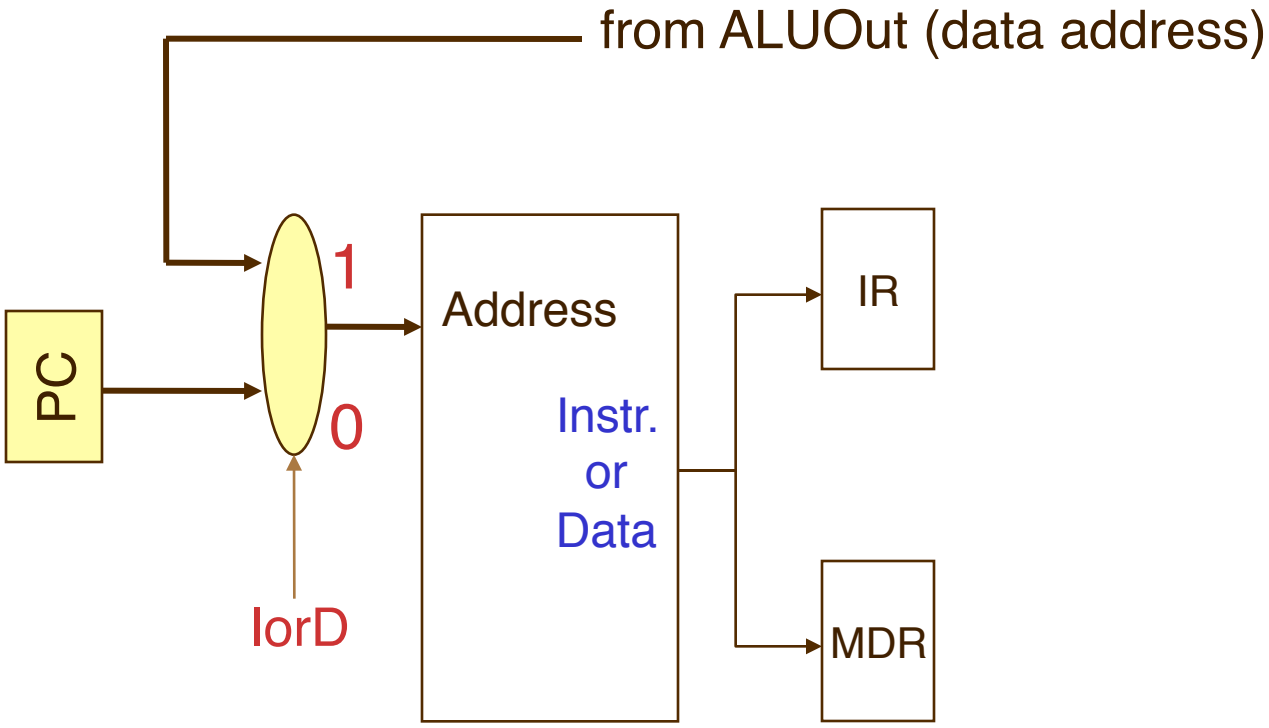
Control Signals Added in Multicycle Implementation



Control: IorD

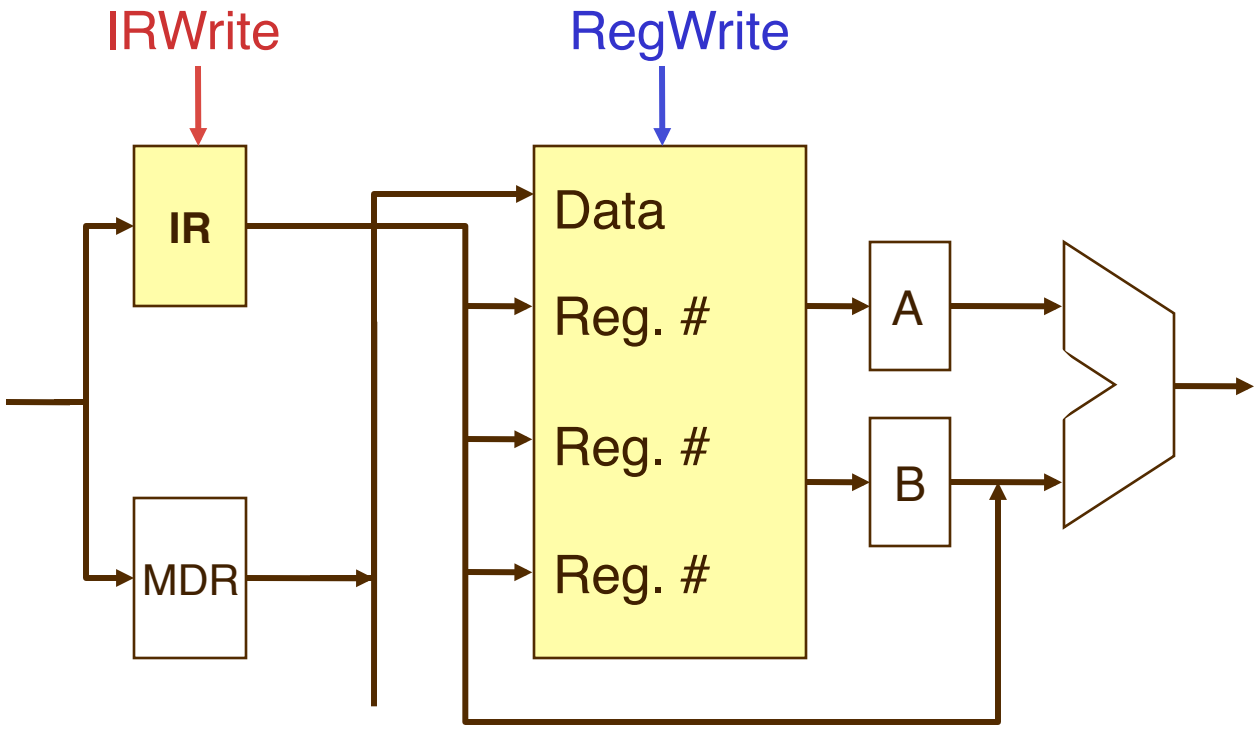
To select the source of address sent to the memory

- IorD = 0 ⇒ instruction, IorD = 1 ⇒ data



Control: IRWrite

- Enable write instruction register
  - IRWrite = 1 only upon instruction fetch; IRWrite = 0 otherwise
- Why no control for MDR?
  - Already have RegWrite to control when MDR being written to register



Control: ALUSrcA, ALUSrcB

To save hardware resources

- ALU operations and additions for branches/jumps share one ALU

What do we need for branches/jumps?

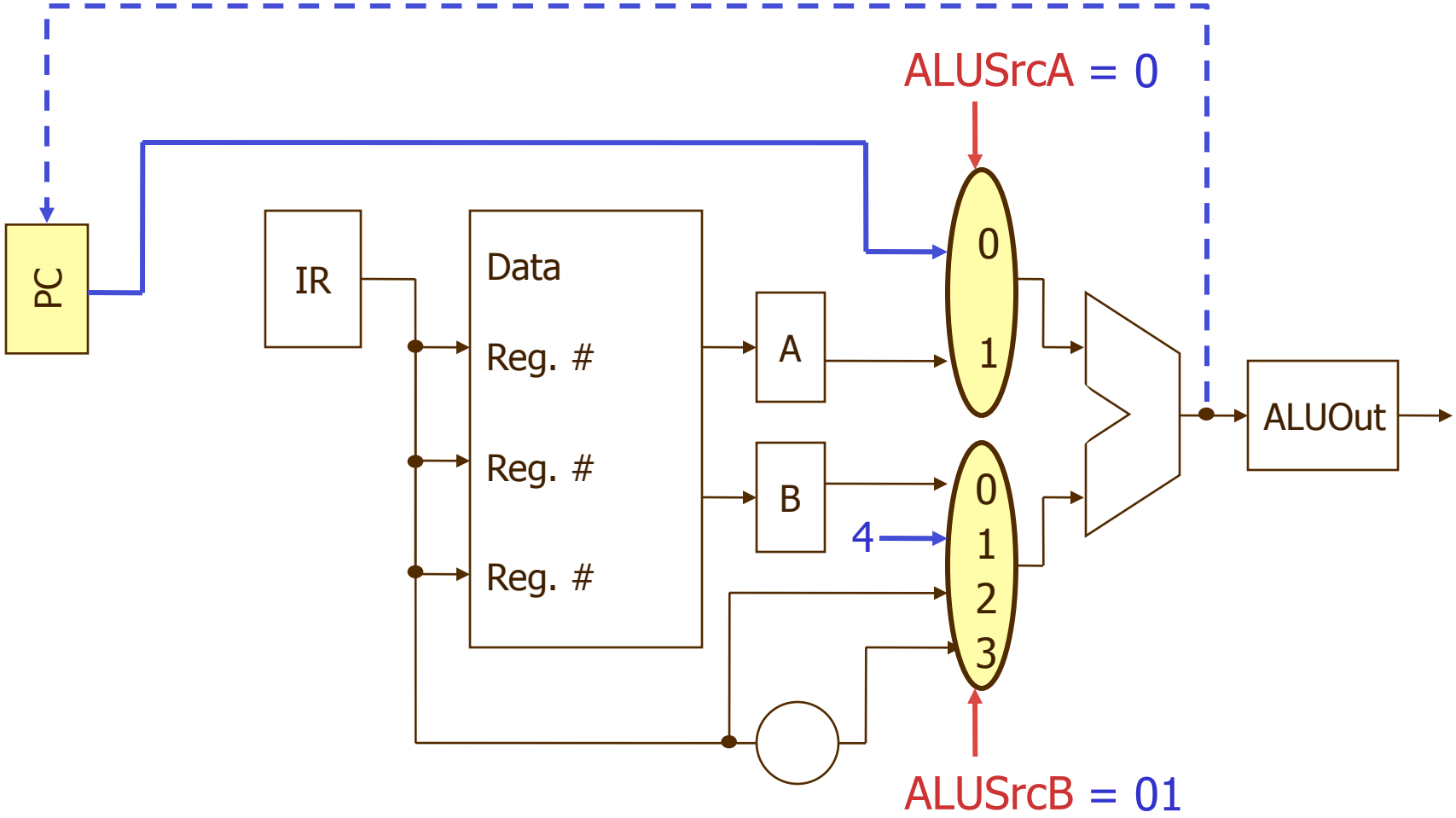
- (Taken) target, PC = PC + immediate << 2;
- (Non-taken) target, PC = PC + 4;
  - Also, the outcome of comparison (Zero?) for conditional branches
- (Unconditional) target, PC = PC[31-28] || (IR[25-0] << 2)

MIPS distributes all the above (target computations and comparison) to different cycles to fully utilize ALU without incurring unneeded cycle

Control: ALUSrcA, ALUSrcB by Example (1)

Base case: PC + 4

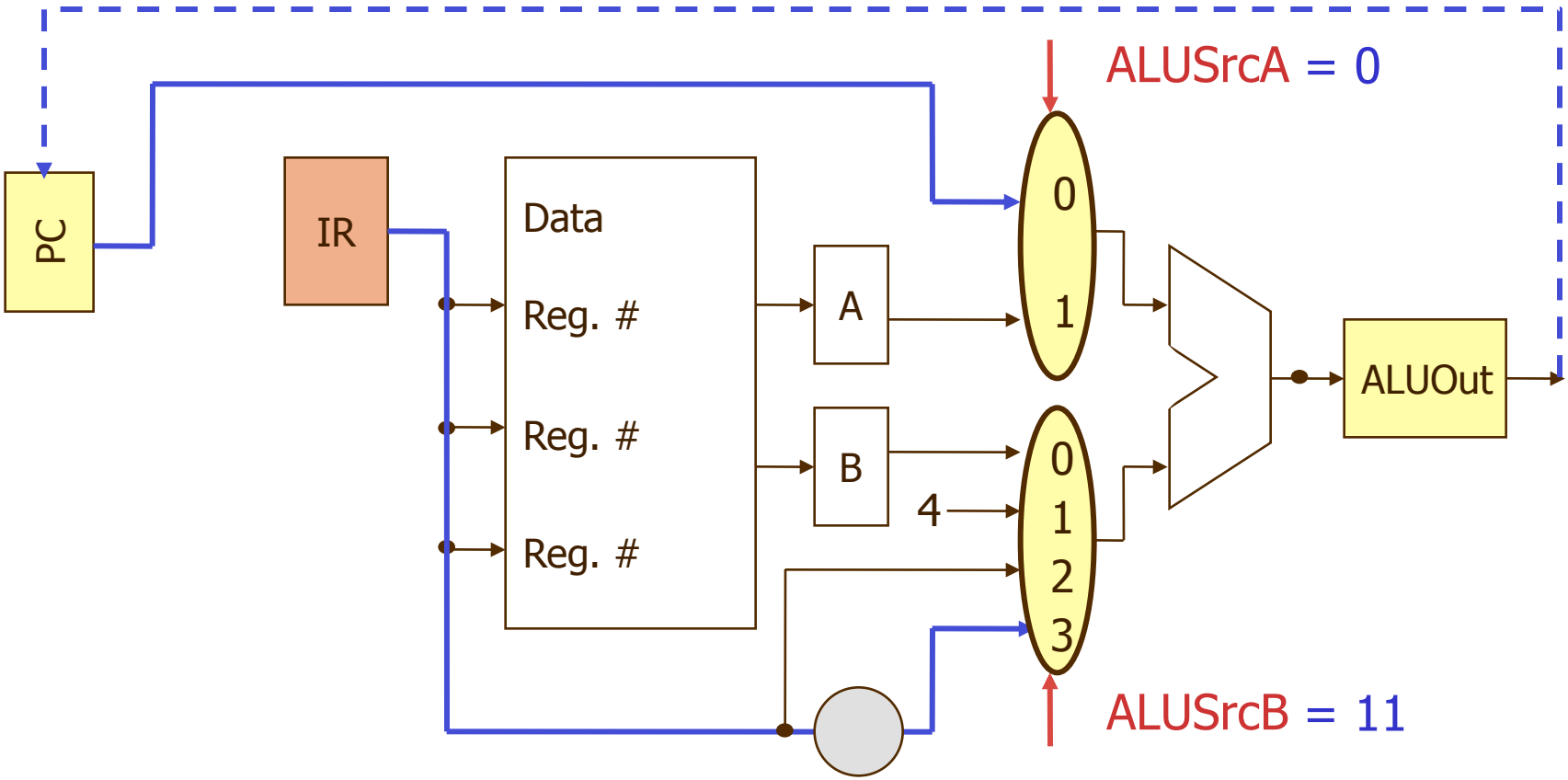
- Happens at step 1, while fetching instruction



Control: ALUSrcA, ALUSrcB by Example (2)

Conditional branch: e.g. beq $1, $2, target

- Calculation happens at step 2, while decoding instruction
- **ALUOut** stores the target address
  - Will be written to PC at step 3 if the comparison produces **Zero = 1**



Control: PCWriteCond, PCWrite, PCSource

Unconditional branch: e.g. j target

- Concatenation happens at step 3, does not involve ALU

Putting all together

- Writing to PC needs a MUX to select PC+4 or target address
  - Signals controlling the MUX come from PCWriteCond & PCWrite
    - PCWriteCond = 1, when the instruction is a conditional branch
    - PCWrite = 1, when the instruction is an unconditional branch
  - Signal = Zero • PCWriteCond + PCWrite
- PCSource selects the source of next PC
  - 00 ⇒ PC + 4, come from ALU
  - 01 ⇒ PC + offset, come from ALUOut
  - 10 ⇒ Jump address, come from concatenation

5b. Control Unit Design
(Finite state machine)

Control Unit Design for Multicycle Implementation

- Specifying control signals for single-cycle implementation is simple
  - Because everything happens within one cycle
  - Only require a set of truth tables based on instruction classes
- Specifying control signals for multicycle implementation is complex
  - Because an instruction is executed in a series of steps
  - Hence, have to specify signals for both the current and next steps

Two techniques for specifying the control:

1. **Finite state machine** representation
2. **Microprogramming** (will be explained if time permits)

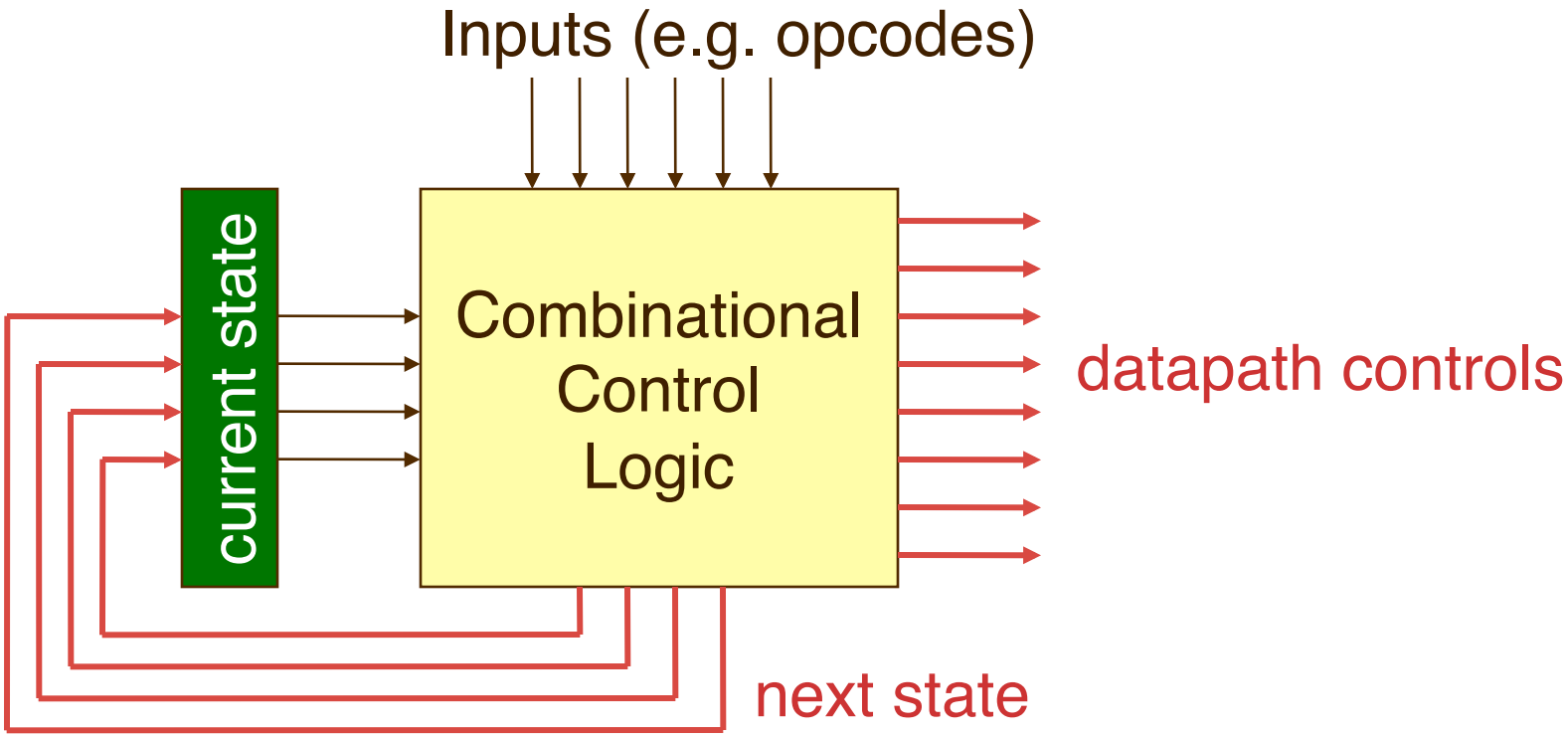
Finite State Machine (FSM)

A FSM consists of

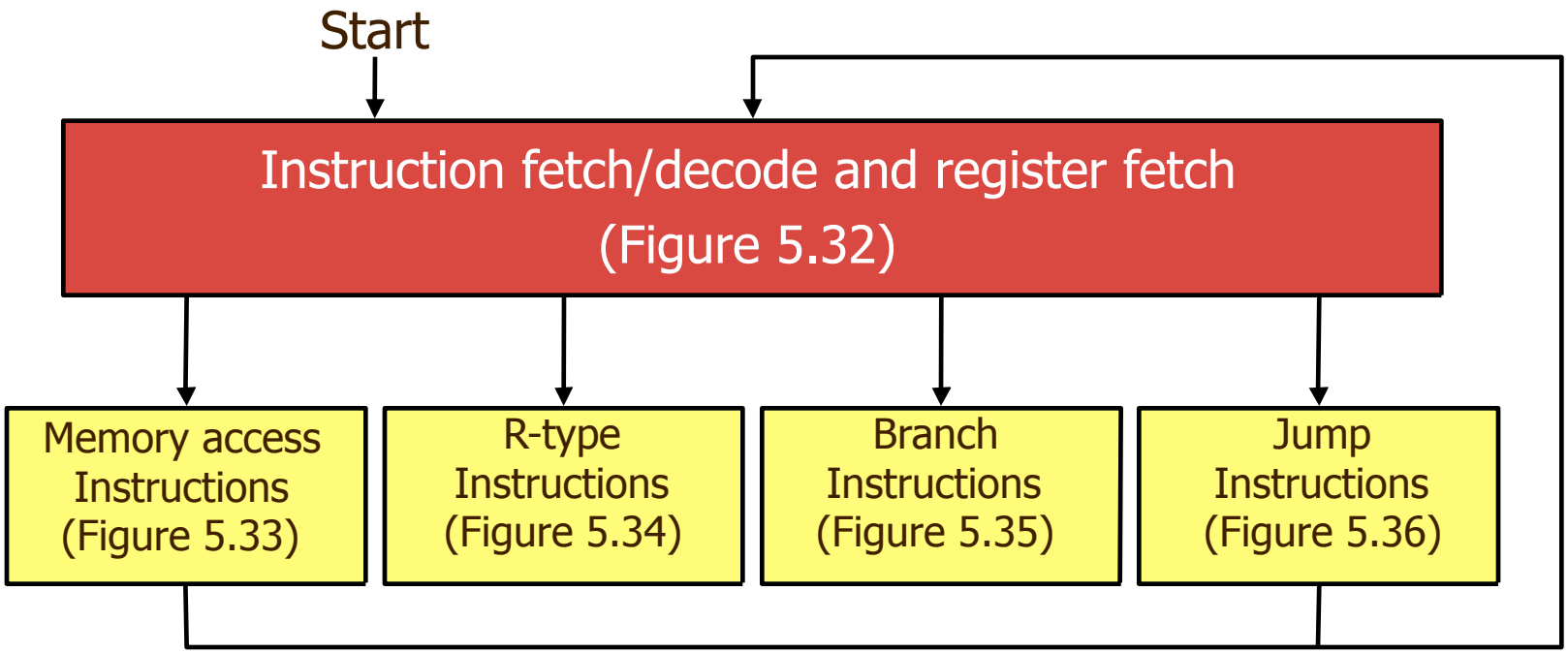
- A set of states & directions on how these states change

Directions

- Functions mapping (current state, inputs) to (new state)

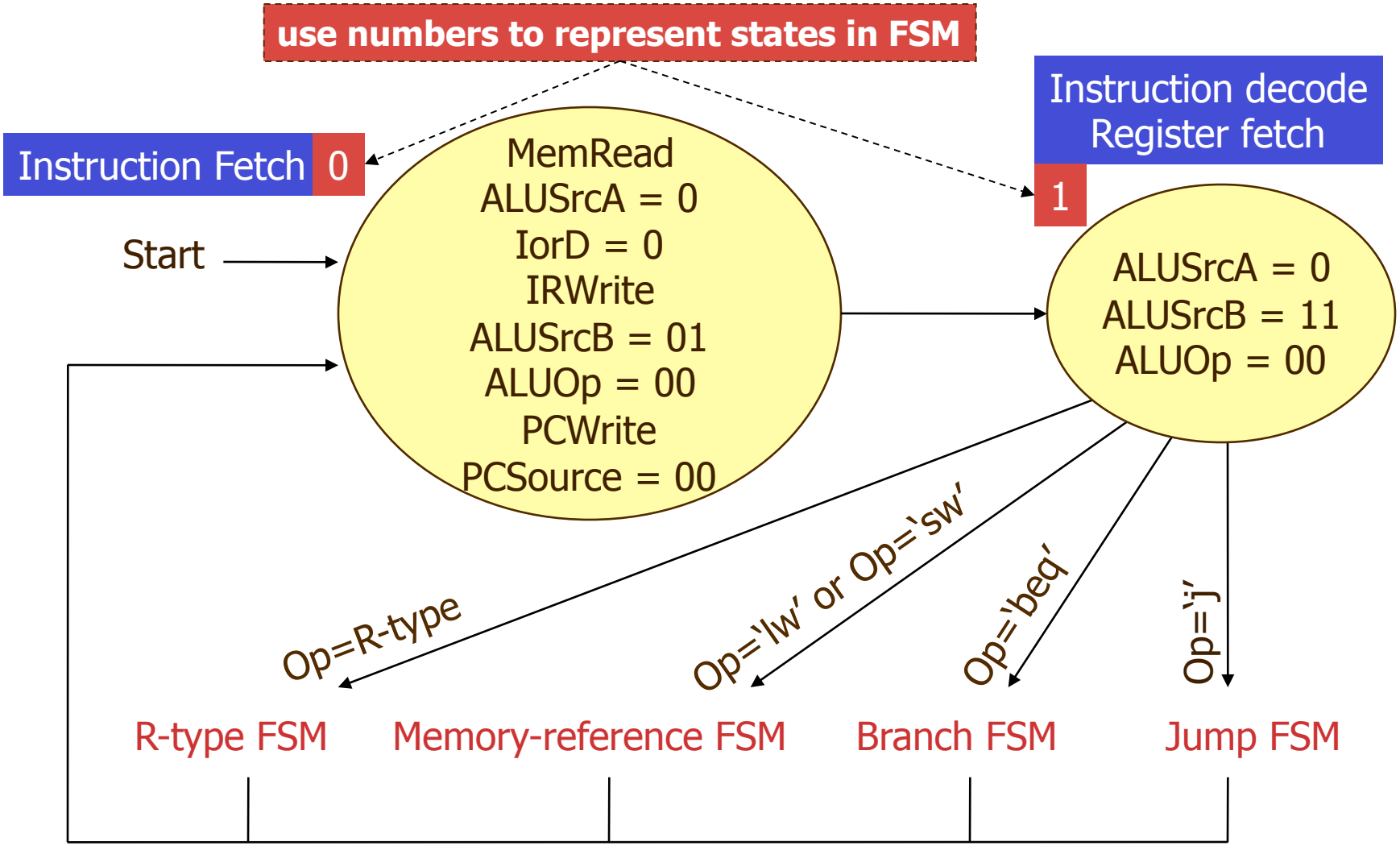


High-Level View of Finite State Machine Control

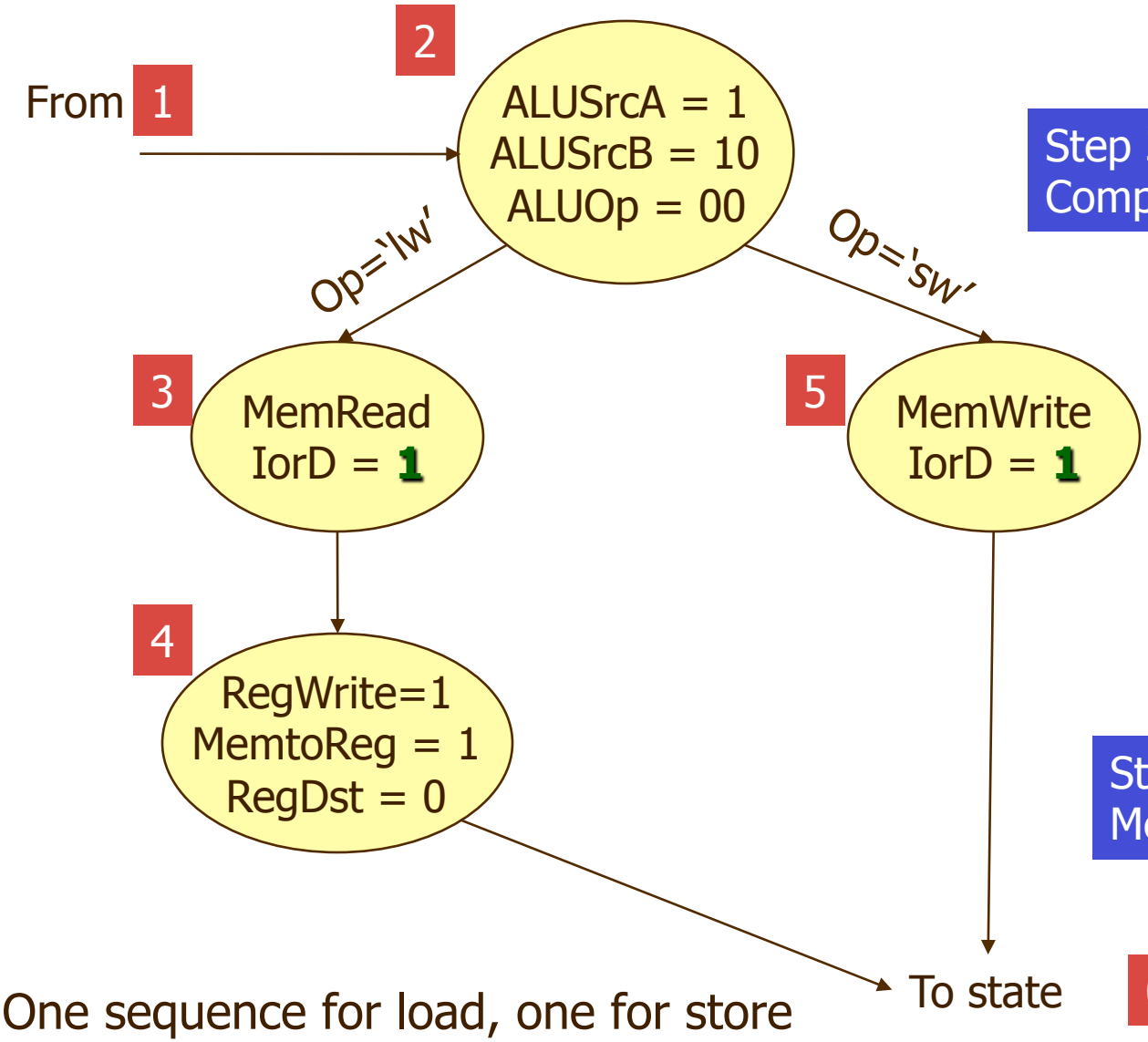


Instruction Fetch & Decode

- This part is common to all instructions



Memory Reference FSM



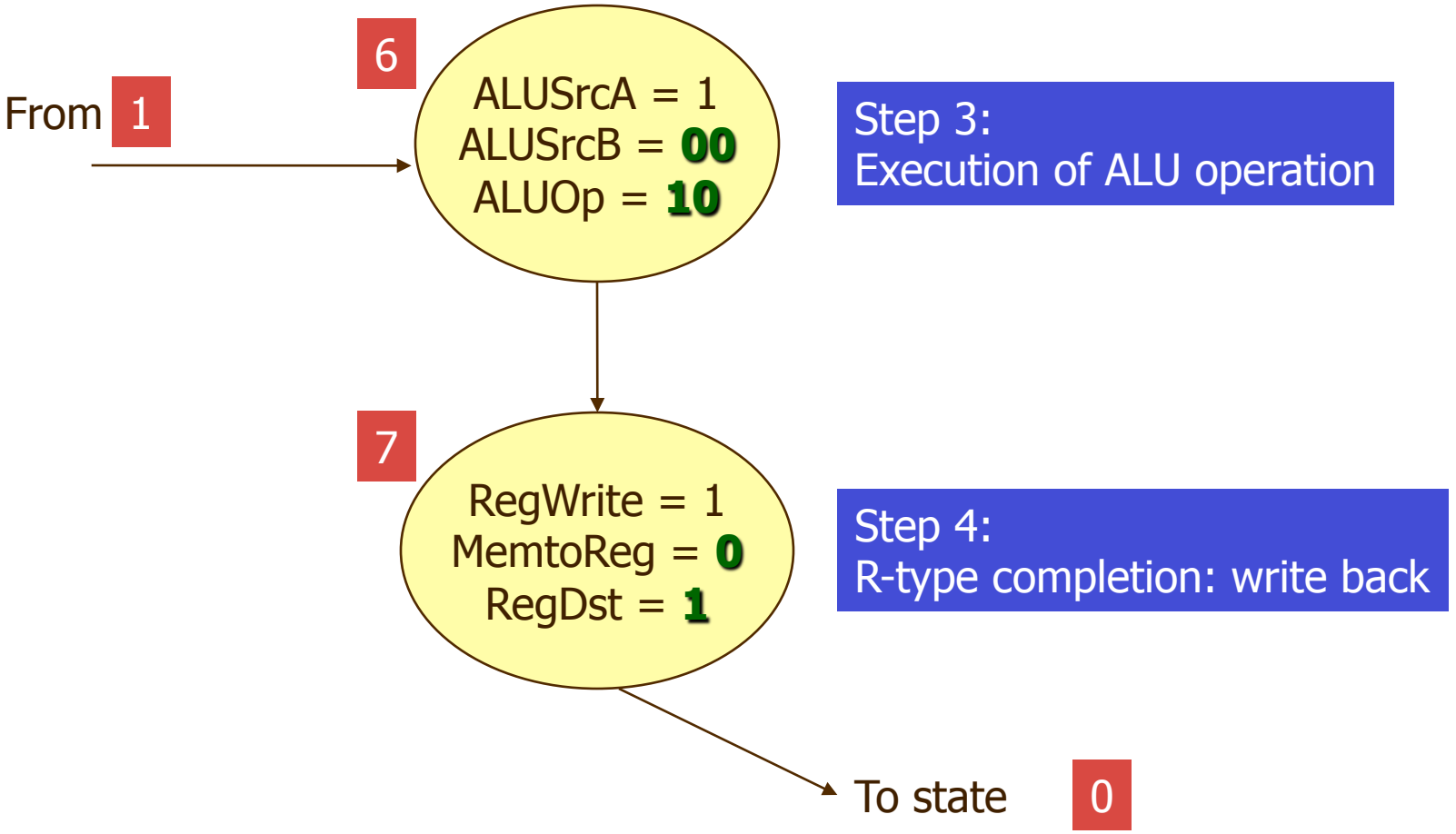
Step 3:
Computing memory address

Step 4:
Access memory

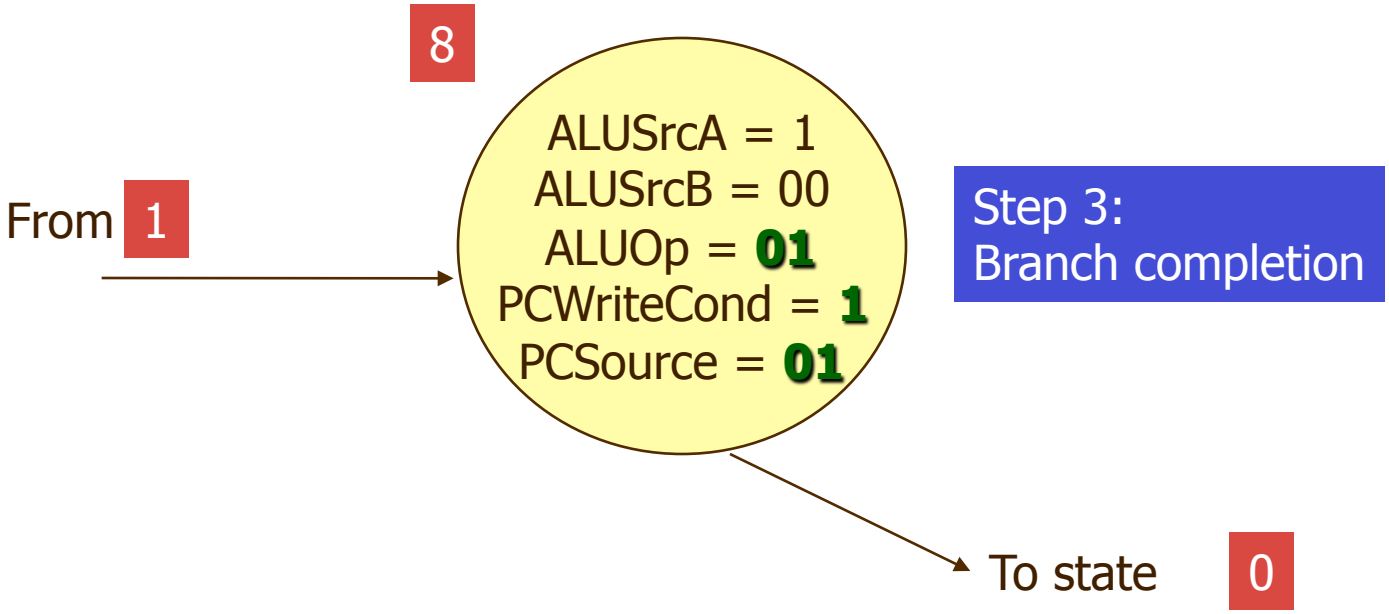
Step 5:
Memory read completion

- One sequence for load, one for store

R-Type FSM

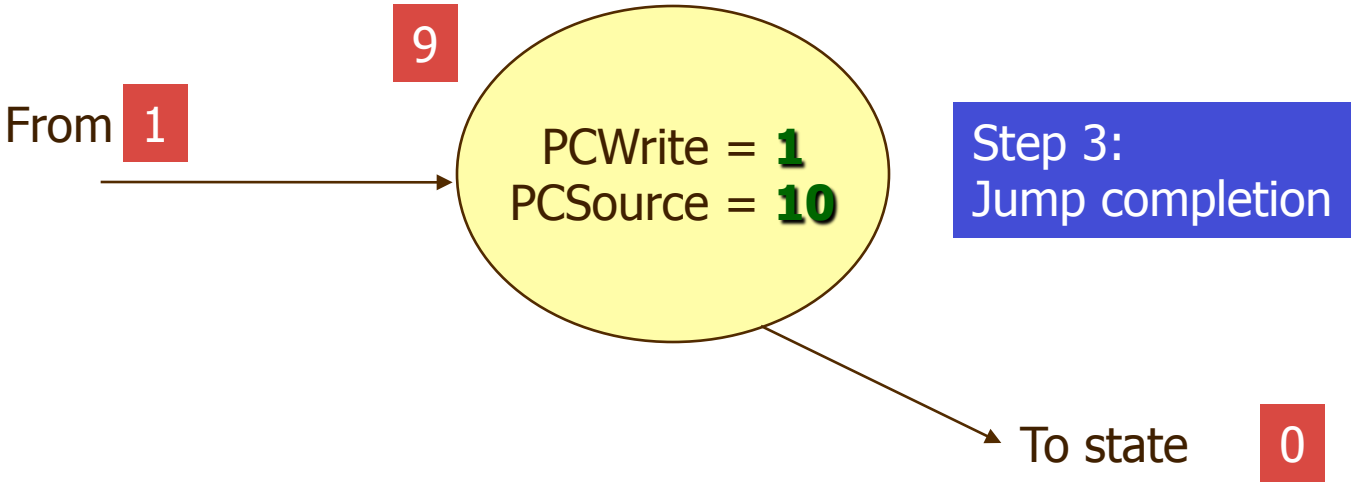

From 1
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Step 3:
Execution of ALU operation
7
RegWrite = 1
MemtoReg = 0
RegDst = 1
Step 4:
R-type completion: write back
To state 0

Branch FSM



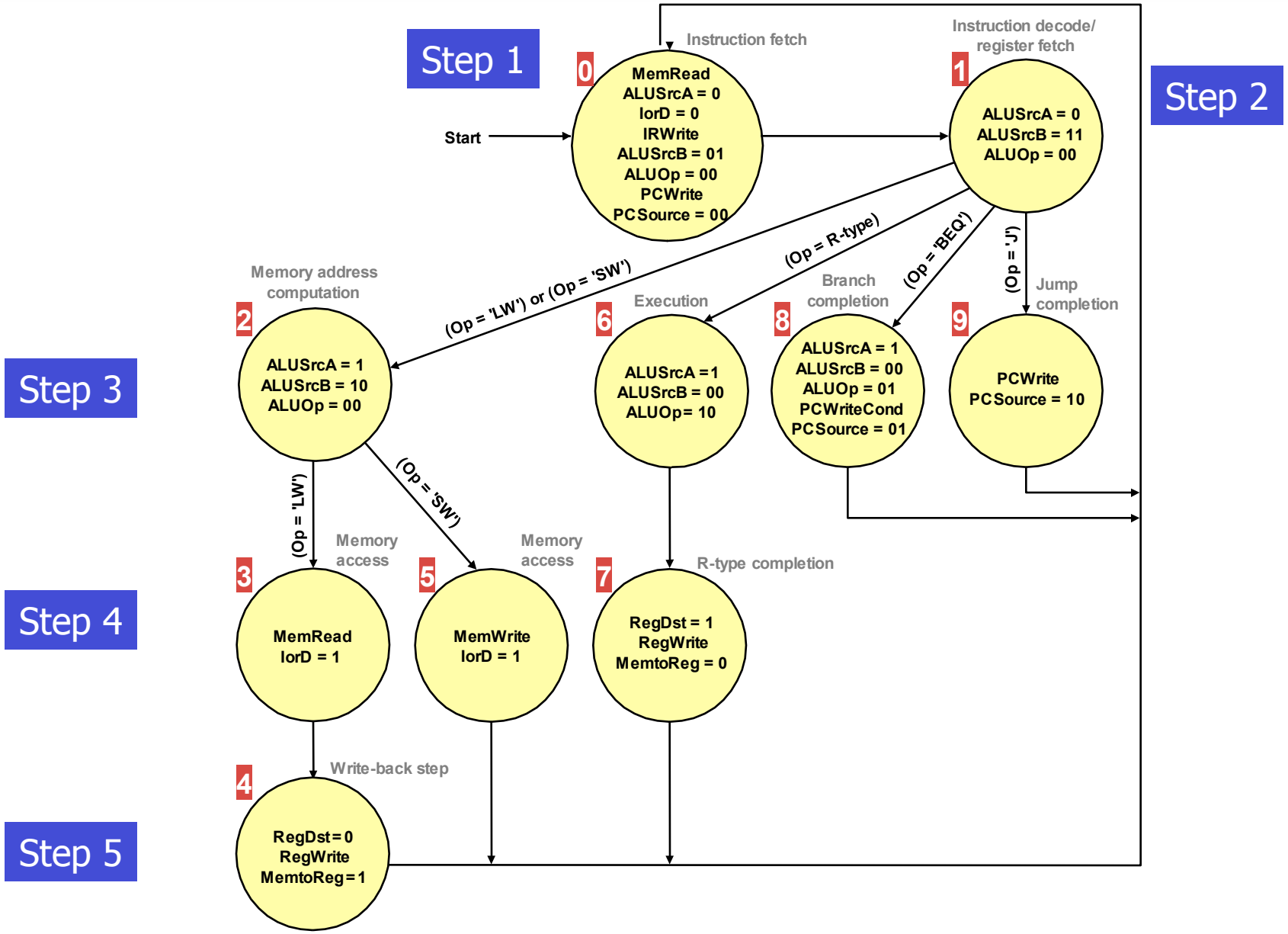
- ALUSrcA, ALUSrcB, ALUOp cause ALU to compare the registers
- PCSource, PCWriteCond enable change of PC if branch condition is true
  - Zero output of the ALU gives the branch condition

Jump FSM



- Concatenation is executed all the times
- Concatenation = `PC[31-28] || (IR[25-0] << 2)`
  - But, the result of concatenation is used only when PCSource = 10
  - Otherwise, dropped

Putting All Together



5c. Control Unit Design
(Microprogramming)

Simplifying Control Design by Microprogramming

- Use a FSM to represent the MIPS control is relatively easy & feasible

However,

- In case of a large instruction set and a large number of complex addressing modes (e.g., Intel x86/Pentium instruction set), there will be thousands of states in the FSM, which becomes too large and cumbersome to handle

As programs become large, we need a good structuring techniques to keep the programs comprehensible

The solution is called **Microprogramming**

- It basically treats the control design as a programming problem
  i.e. use microinstructions to '*program controls we need*" in each step
- Note that:
  - Datapath remains the same
  - Only control is specified differently

Microprogramming

- The set of controls to be asserted during a cycle are considered as an "control instruction"
- To distinguish such low-level control instructions from MIPS' instructions, we call them **microinstructions or microcodes**
- Executing a microinstruction has the effect of asserting the control signals specified by the microinstruction

- Values asserted on the control lines are represented symbolically in a **microprogram**, a representation of the microinstructions

In brief, you can say that each instruction

- Consists of a sequence of microinstructions
- # of microinstructions for the instruction =
  # of cycles to execute the instruction

Microinstructions

Microinstructions contain fields that specify

- **ALU Control, SRC1 and SRC2**
  - ALUop and the sources
- **Register control**
  - Read or write to the register file, and
  - Select the source of the value to write
- **Memory**
  - Read or write, and
- **PCWrite Control**
  - Specify the writing on the PC
- **Sequencing**
  - How to choose the next microinstruction

Example of Microinstruction

Label	ALU Control	SRC1	SRC2	Register Control	Memory	PCWrite Control	Sequencing
FETCH	ADD	PC	4		Read PC	ALU	Seq
	ADD	PC	Extshift	READ			DISPATCH 1
Mem1	ADD	A	Extend				DISPATCH 2
LW2					Read ALU		Seq
				Write MDR			FETCH
SW2					Write ALU		FETCH
Rformat1	Func Code	A	B				Seq
				Write ALU			FETCH
BEQ1	Subt	A	B			ALUOut cond	FETCH
JUMP1						Jump Addr	FETCH

- We can think of the **dispatch operation** as a ***case*** or ***switch*** statement with the opcode field and the dispatch table **T** used to select one of ***n*** different microinstruction sequences with one of ***n*** different labels (all ending in "**T**")

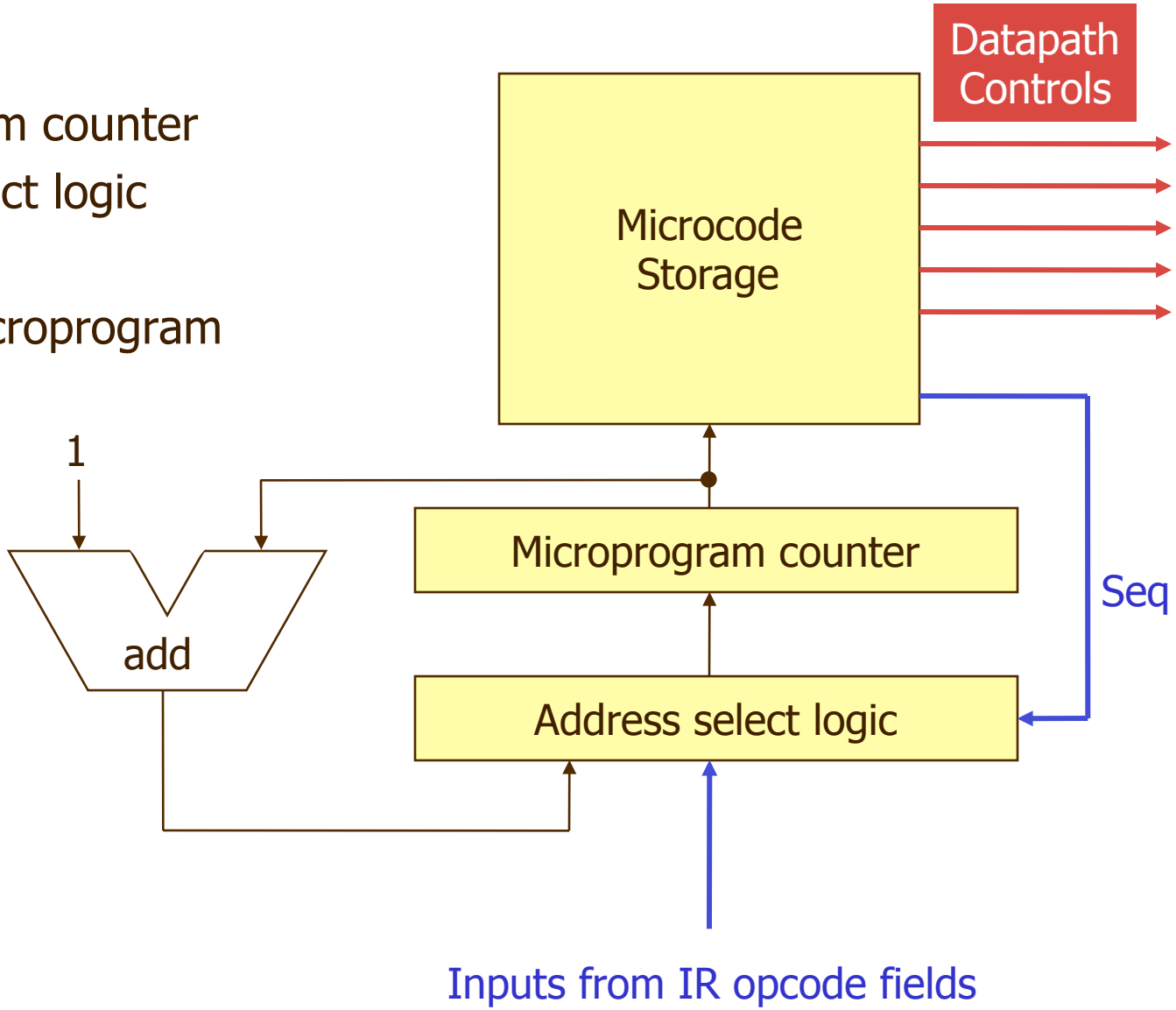
Implementing Microprogramming

We need

- Microprogram counter
- Address select logic

Storage for microprogram

- ROMs
- PLAs



Microinstructions and their fields

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.