

CS2100

<http://www.comp.nus.edu.sg/~cs2100/>

COMPUTER ORGANISATION

Lecture #11

The Processor: Datapath



NUS
National University
of Singapore

School of
Computing

Lecture #11: Processor: Datapath

1. Building a Processor: Datapath & Control
2. MIPS Processor: Implementation
3. Instruction Execution Cycle (Recap)
4. MIPS Instruction Execution
5. Let's Build a MIPS Processor
 - 5.1 Fetch Stage
 - 5.2 Decode Stage
 - 5.3 ALU Stage
 - 5.4 Memory Stage
 - 5.5 Register Write Stage
6. The Complete Datapath!

1. Building a Processor: Datapath & Control

- Two major components for a processor

Datapath

- Collection of components that process data
- Performs the arithmetic, logical and memory operations

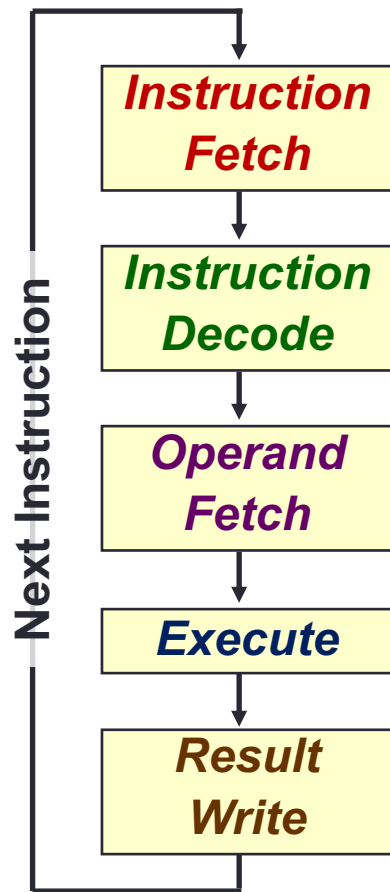
Control

- Tells the datapath, memory and I/O devices what to do according to program instructions

2. MIPS Processor: Implementation

- Simplest possible implementation of a subset of the core MIPS ISA:
 - **Arithmetic and Logical operations**
 - `add, sub, and, or, addi, andi, ori, slt`
 - **Data transfer instructions**
 - `lw, sw`
 - **Branches**
 - `beq, bne`
- Shift instructions (`sll, srl`) and J-type instructions (`j`) will not be discussed:
 - Left as exercises 😊

3. Instruction Execution Cycle (Basic)



1. **Fetch:**

- Get instruction from memory
- Address is in **P**rogram **C**ounter (PC) Register

2. **Decode:**

- Find out the operation required

3. **Operand Fetch:**

- Get operand(s) needed for operation

4. **Execute:**

- Perform the required operation

5. **Result Write (Store):**

- Store the result of the operation

4. MIPS Instruction Execution (1/2)

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stages not shown:
 - The standard steps are performed

	<code>add \$3, \$1, \$2</code>	<code>lw \$3, 20(\$1)</code>	<code>beq \$1, \$2, ofst</code>
Fetch	<i>standard</i>	<i>standard</i>	<i>standard</i>
Decode			
Operand Fetch	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Use 20 as <i>opr2</i> 	<ul style="list-style-type: none"> ○ Read [\$1] as <i>opr1</i> ○ Read [\$2] as <i>opr2</i>
Execute	<i>Result = opr1 + opr2</i>	<ul style="list-style-type: none"> ○ <i>MemAddr</i> = <i>opr1</i> + <i>opr2</i> ○ Use <i>MemAddr</i> to read from memory 	<i>Taken</i> = (<i>opr1</i> == <i>opr2</i>)? <i>Target</i> = (PC +4) + ofst ×4
Result Write	<i>Result</i> stored in \$3	<i>Memory</i> data stored in \$3	if (<i>Taken</i>) PC = <i>Target</i>

- **opr** = operand
- **MemAddr** = Memory Address
- **ofst** = offset

4. MIPS Instruction Execution (2/2)

- Design changes:
 - Merge *Decode* and *Operand Fetch* – Decode is simple for MIPS
 - Split *Execute* into **ALU** (Calculation) and **Memory Access**

	<code>add \$3, \$1, \$2</code>	<code>lw \$3, 20(\$1)</code>	<code>beq \$1, \$2, ofst</code>
Fetch	Read inst. at [PC]	Read inst. at [PC]	Read inst. at [PC]
Decode & Operand Fetch	<ul style="list-style-type: none"> Read [\$1] as <i>opr1</i> Read [\$2] as <i>opr2</i> 	<ul style="list-style-type: none"> Read [\$1] as <i>opr1</i> Use 20 as <i>opr2</i> 	<ul style="list-style-type: none"> Read [\$1] as <i>opr1</i> Read [\$2] as <i>opr2</i>
ALU	$Result = opr1 + opr2$	$MemAddr = opr1 + opr2$	$Taken = (opr1 == opr2) ?$ $Target = (PC+4) + ofst \times 4$
Memory Access		Use <i>MemAddr</i> to read from memory	
Result Write	<i>Result</i> stored in \$3	<i>Memory data</i> stored in \$3	if (<i>Taken</i>) PC = <i>Target</i>

5. Let's Build a MIPS Processor

- What we are going to do:
 - Look at each stage closely, figure out the requirements and processes
 - Sketch a high level block diagram, then zoom in for each elements
 - With the simple starting design, check whether different type of instructions can be handled:
 - Add modifications when needed
- ➔ Study the design from the viewpoint of a designer, instead of a "tourist" 😊

5.1 Fetch Stage: Requirements

1. **Fetch**
2. Decode
3. ALU
4. Memory
5. RegWrite

- Instruction **Fetch Stage**:

1. Use the **Program Counter (PC)** to fetch the instruction from **memory**

- PC is implemented as a special register in the processor

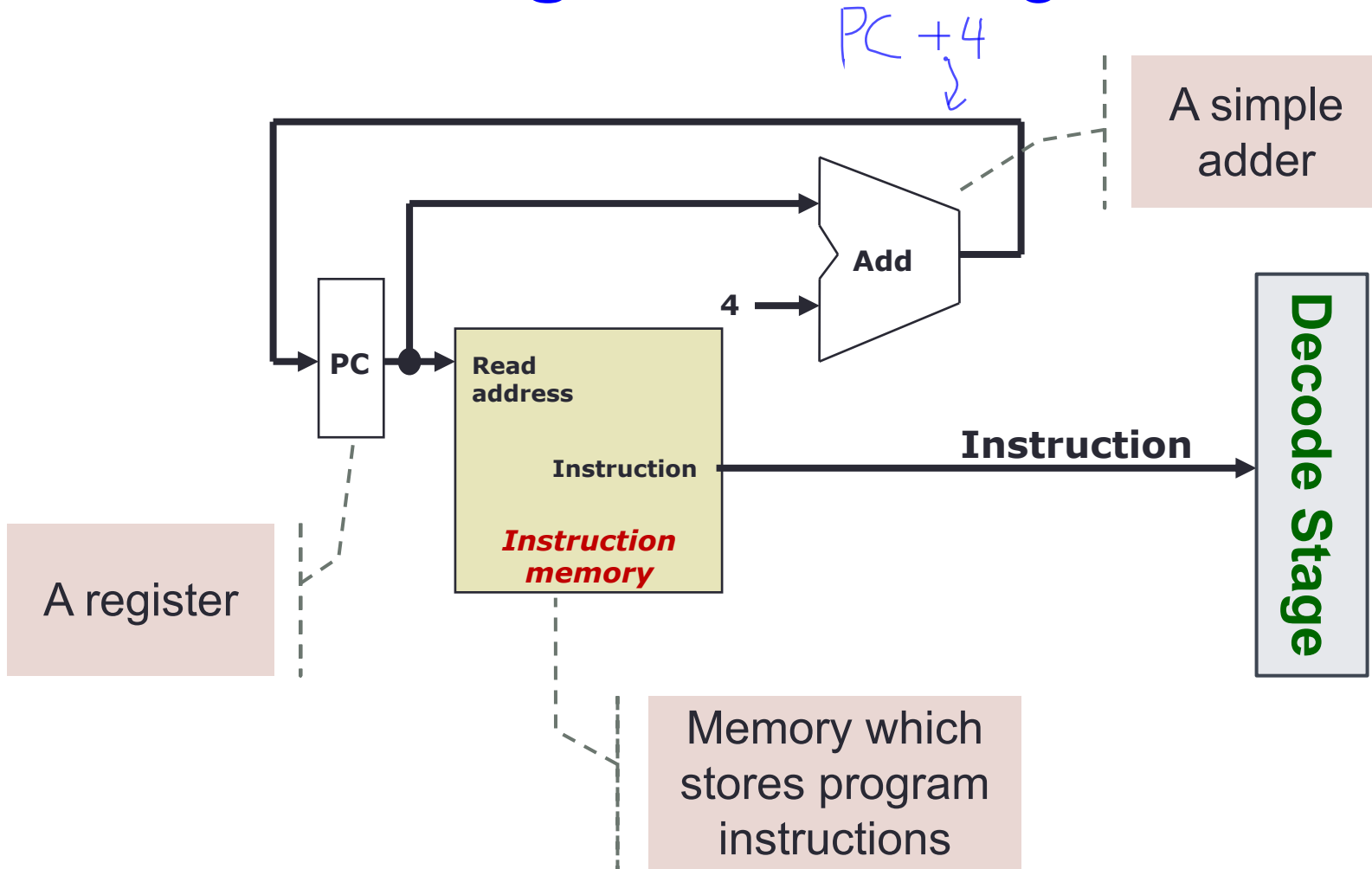
2. **Increment** the PC by 4 to get the address of the next instruction:

- How do we know the next instruction is at PC+4?
- Note the exception when branch/jump instruction is executed


- Output to the next stage (**Decode**):

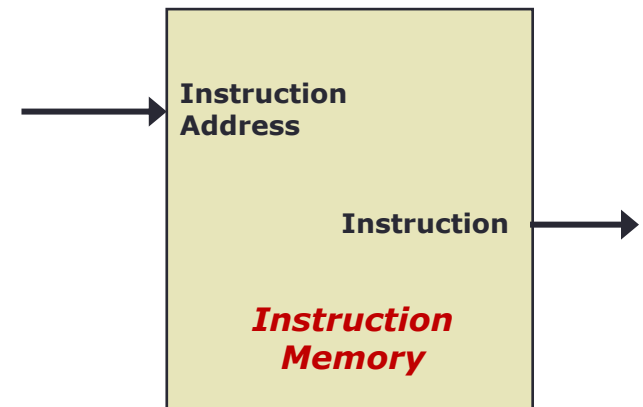
- The instruction to be executed

5.1 Fetch Stage: Block Diagram



5.1 Element: Instruction Memory

- Storage element for the instructions
 - It is a **sequential circuit** (to be covered later)
 - Has an internal state that stores information 
 - Clock signal is assumed and not shown



- Supply instruction given the address
 - Given instruction address M as input, the memory outputs the content at address M
 - Conceptual diagram of the memory layout is given on the right →

Memory	
.....
2048	add \$3, \$1, \$2
2052	sll \$4, \$3, 2
2056	andi \$1, \$4, 0xF
.....

5.1 Element: Adder

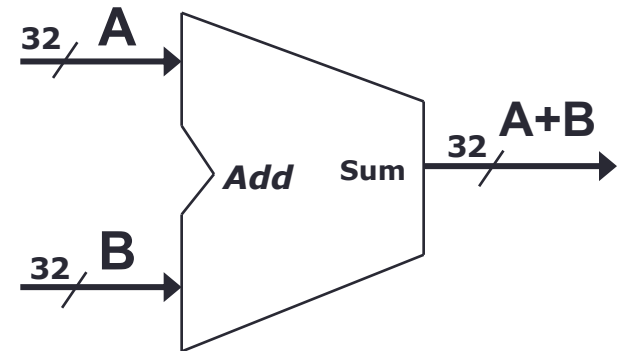
- Combinational logic to implement the addition of two numbers

- **Inputs:**

- Two 32-bit numbers **A**, **B**

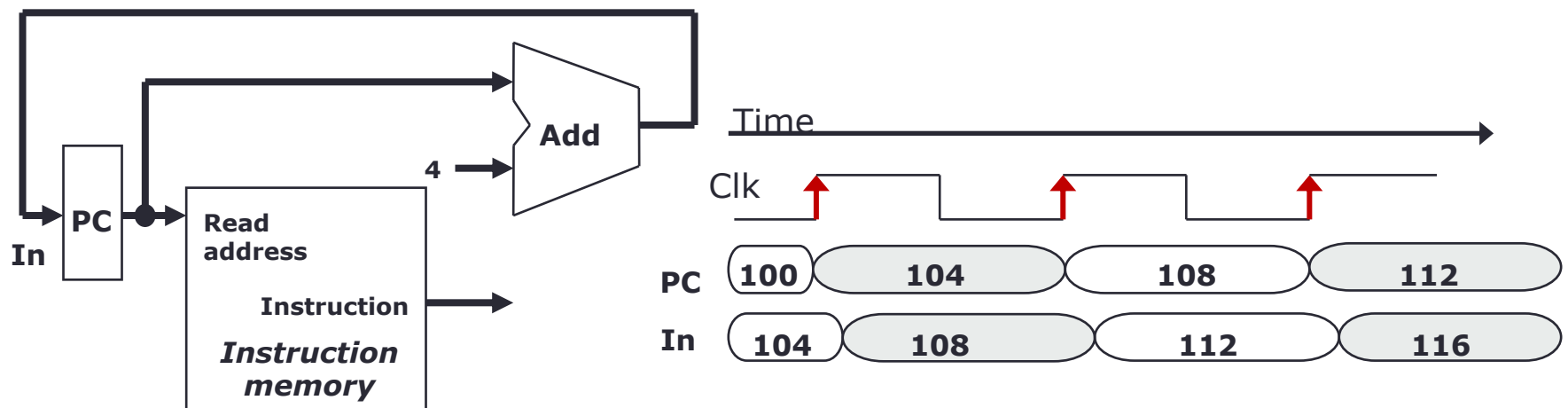
- **Output:**

- Sum of the input numbers, **A + B**



5.1 The Idea of Clocking

- It seems that we are reading and updating the PC at the same time:
 - How can it work properly?
- **Magic of clock:**
 - PC is read during the first half of the clock period and it is updated with PC+4 at the **next rising clock edge**



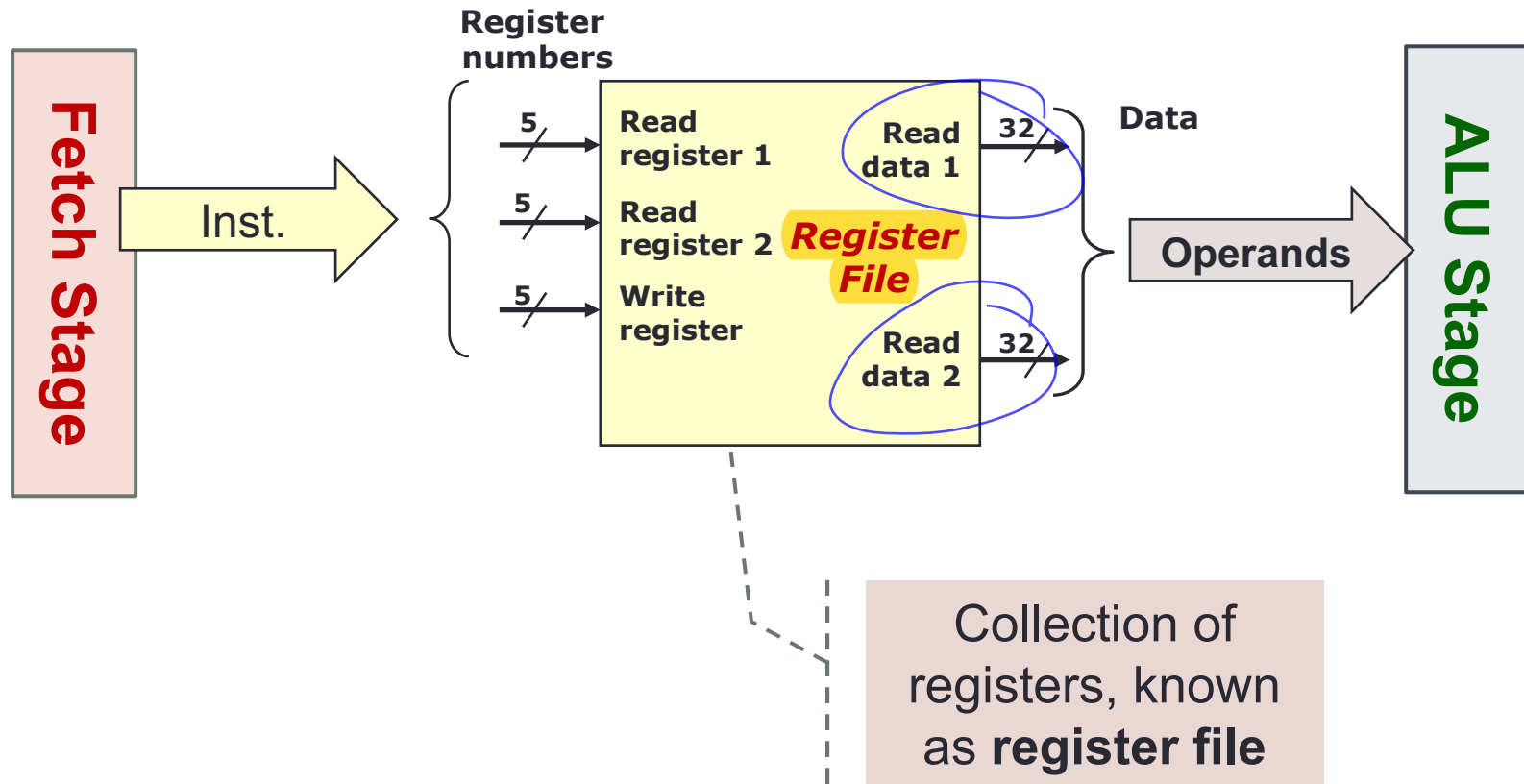
5.2 Decode Stage: Requirements

1. Fetch
2. **Decode**
3. ALU
4. Memory
5. RegWrite

- Instruction **Decode Stage**:
 - Gather data from the instruction fields:
 1. Read the **opcode** to determine instruction type and field lengths
 2. Read data from all necessary registers
 - Can be two (e.g. **add**), one (e.g. **addi**) or zero (e.g. **j**)
- Input from previous stage (**Fetch**):
 - Instruction to be executed
- Output to the next stage (**ALU**):
 - Operation and the necessary operands

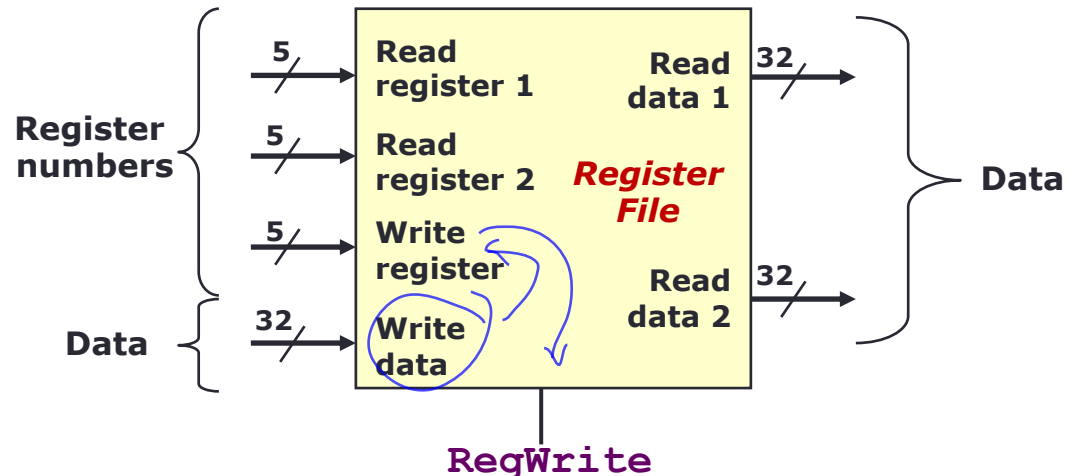
5.2 Decode Stage: Block Diagram

$\$1, \2

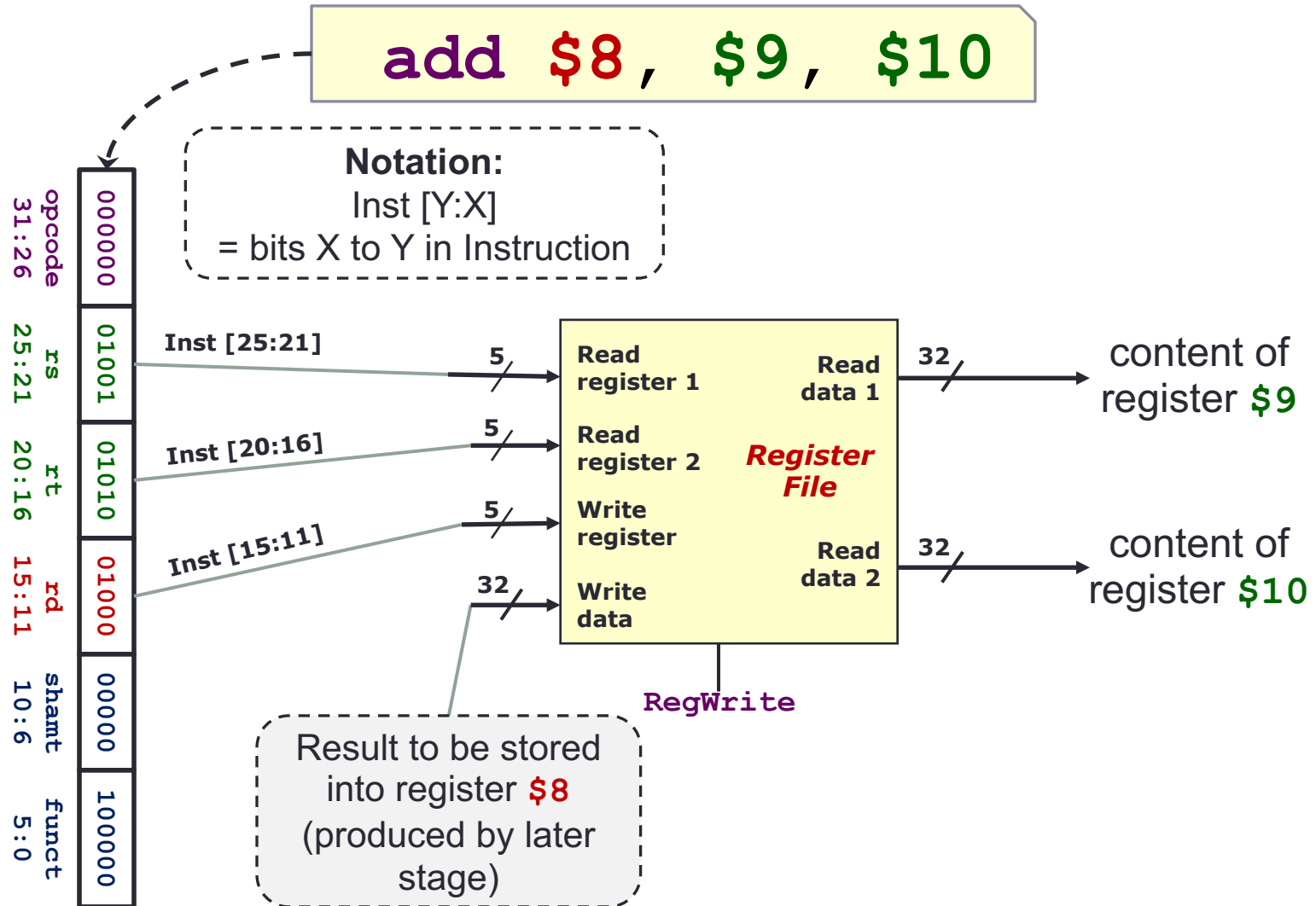


5.2 Element: Register File

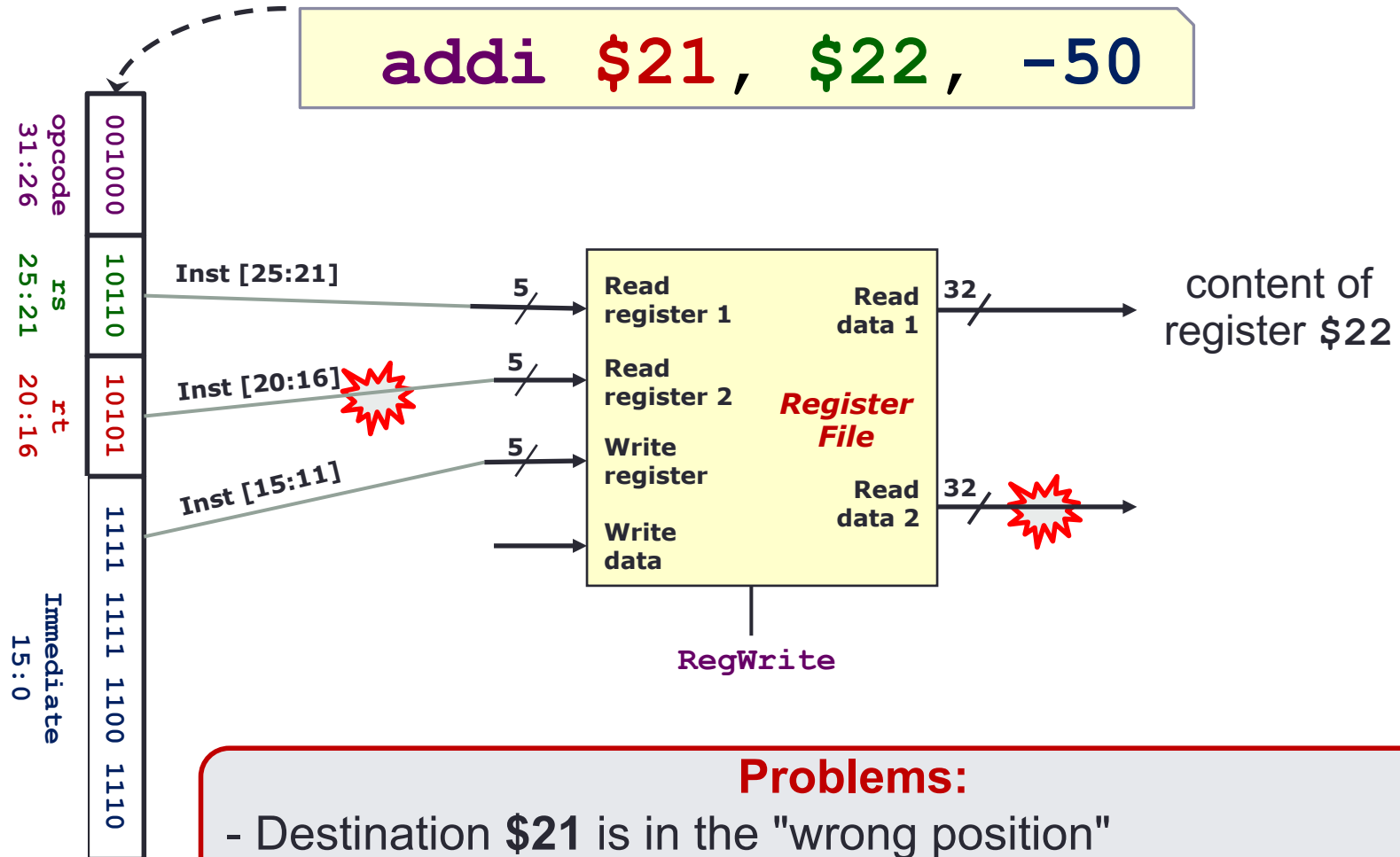
- A collection of 32 registers:
 - Each 32-bit wide; can be read/written by specifying register number
 - Read at most two registers per instruction
 - Write at most one register per instruction
- **RegWrite** is a control signal to indicate:
 - Writing of register
 - 1(True) = Write, 0 (False) = No Write



5.2 Decode Stage: R-Format Instruction



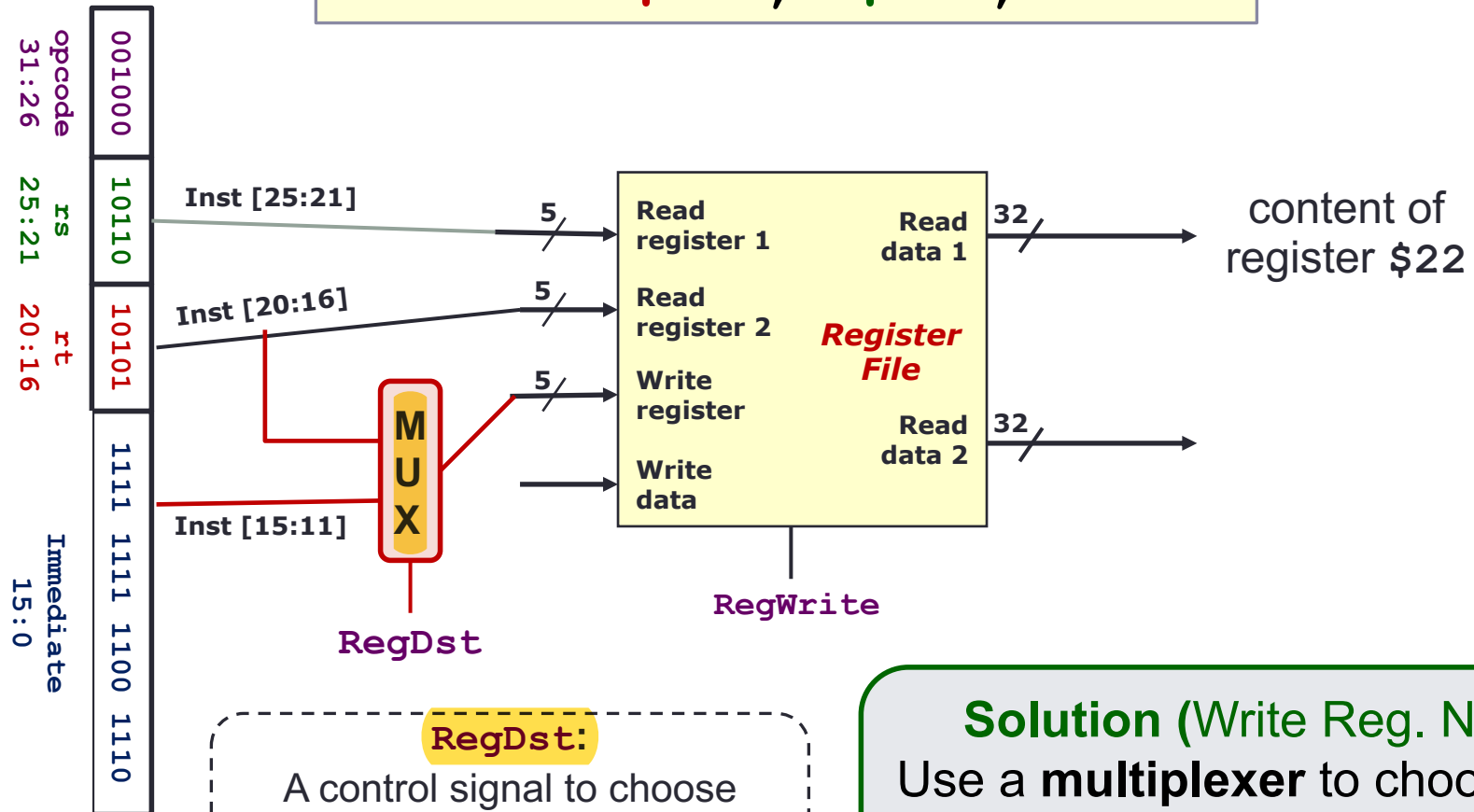
5.2 Decode Stage: I-Format Instruction



- Problems:**
- Destination **\$21** is in the "wrong position"
 - **Read Data 2** is an immediate value, not from register

5.2 Decode Stage: Choice in Destination

addi \$21, \$22, -50



Solution (Write Reg. No.):
Use a **multiplexer** to choose the correct write register number based on instruction type

5.2 Multiplexer

- **Function:**

- Selects one input from multiple input lines

- **Inputs:**

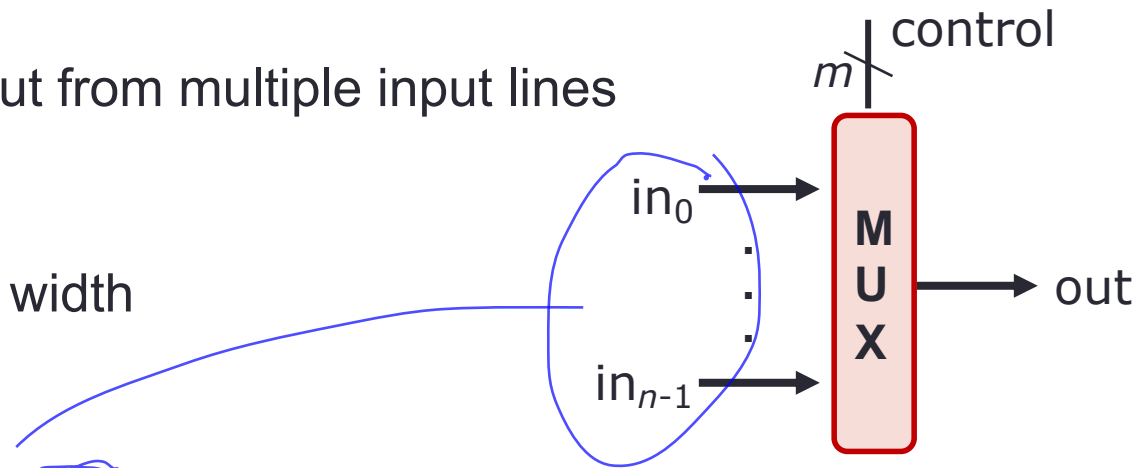
- n lines of same width

- **Control:**

- m bits where $n = 2^m$

- **Output:**

- Select i^{th} input line if control = i

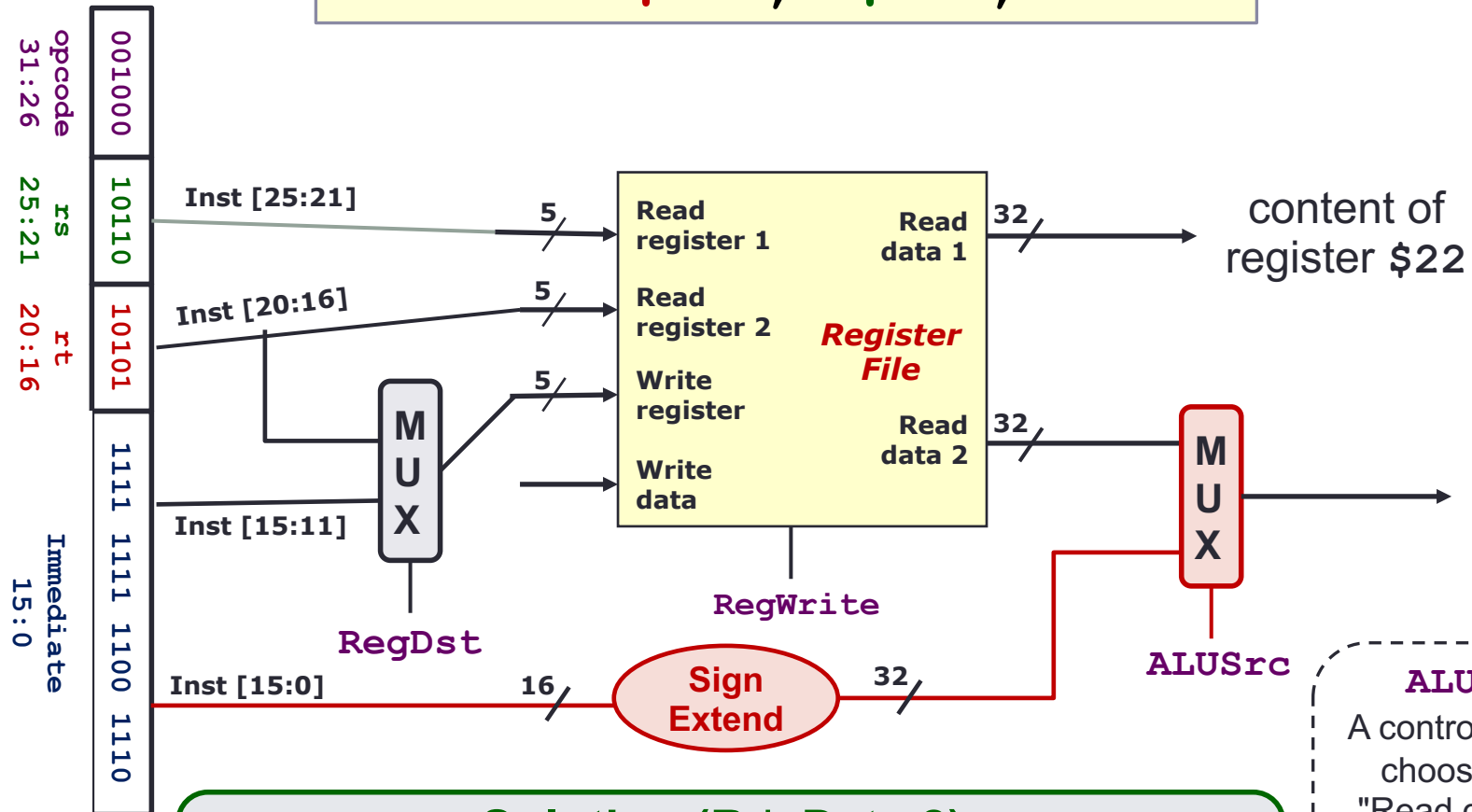


Control=0 \rightarrow select in_0 to out

Control=3 \rightarrow select in_3 to out

5.2 Decode Stage: Choice in Data 2

addi \$21, \$22, -50



Solution (Rd. Data 2):

Use a **multiplexer** to choose the correct operand 2.
Sign extend the 16-bit immediate value to 32-bit

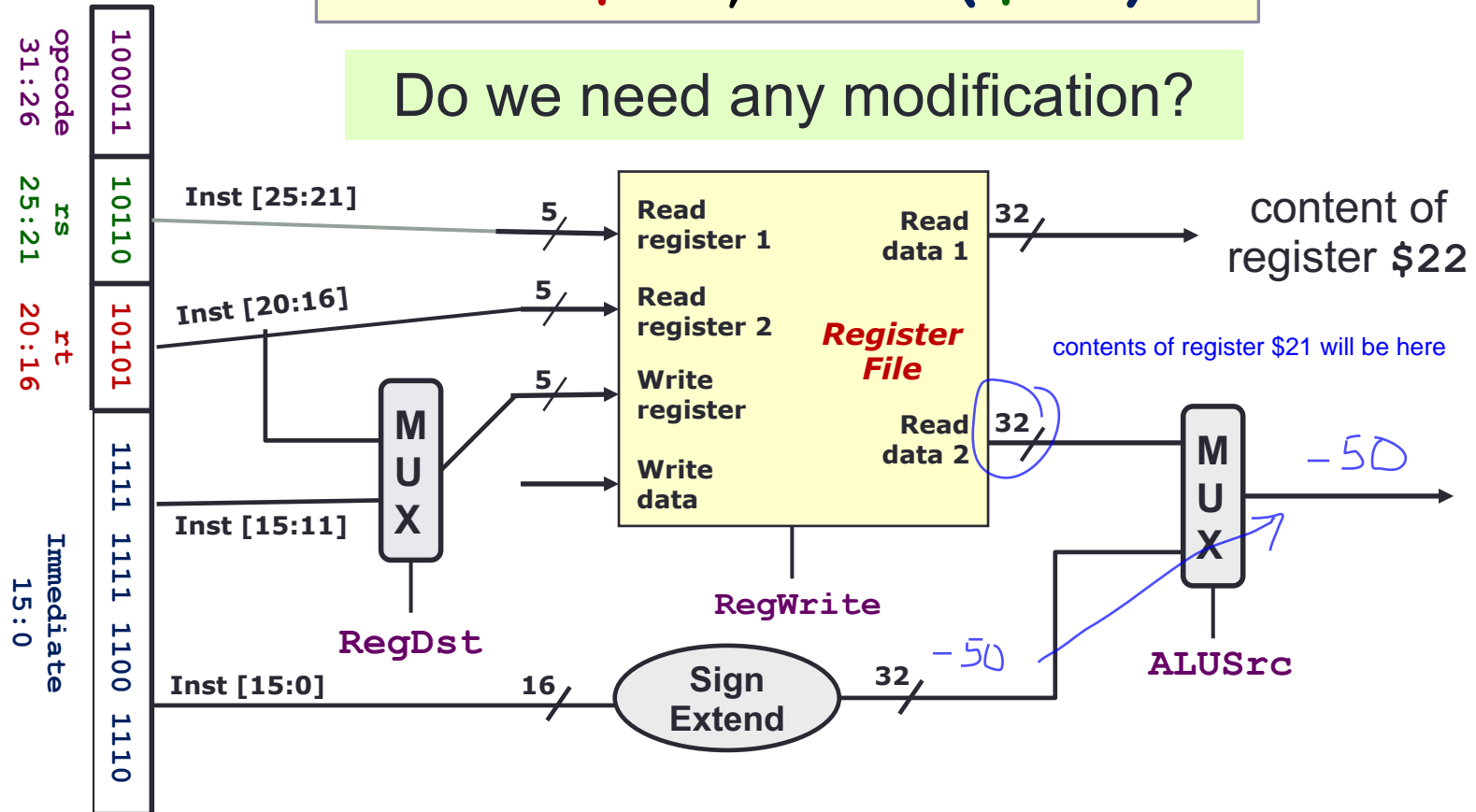
ALUSrc:

A control signal to choose either "Read data 2" or the sign extended **Inst[15:0]** as the second operand

5.2 Decode Stage: Load Word Instruction

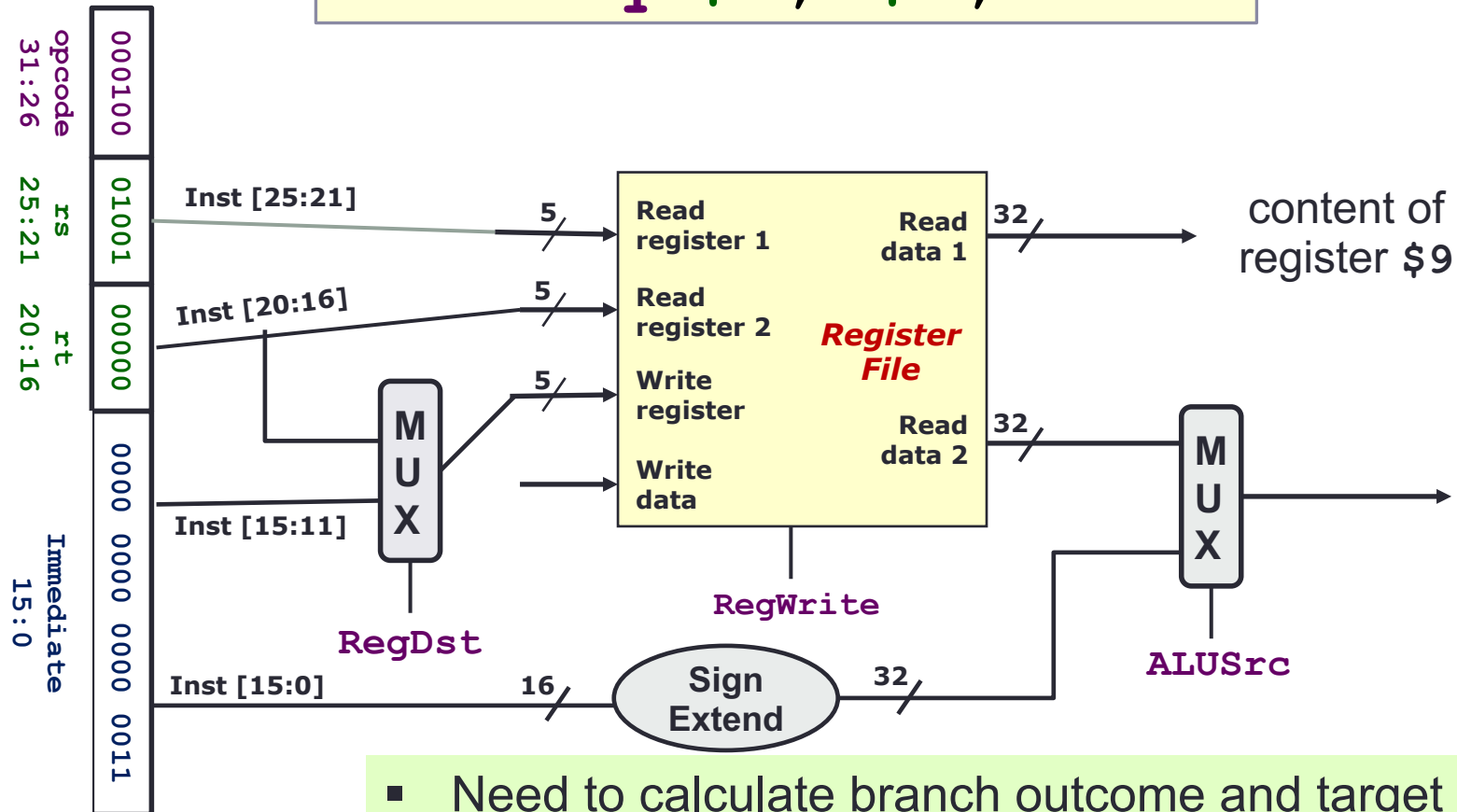
lw \$21, -50(\$22)

Do we need any modification?



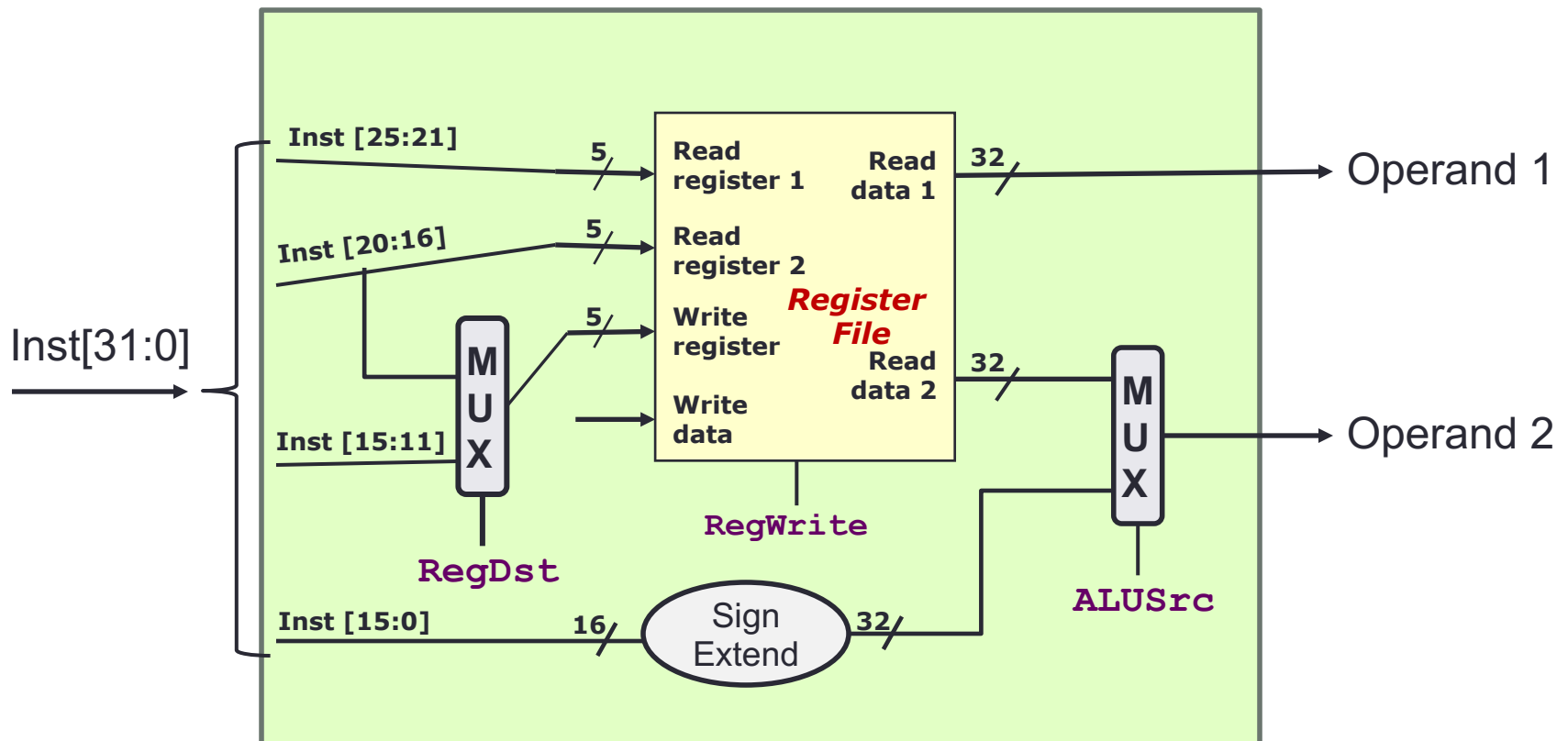
5.2 Decode Stage: Branch Instruction

beq \$9, \$0, 3



- Need to calculate branch outcome and target at the same time!
- We will tackle this problem at the ALU stage

5.2 Decode Stage: Summary

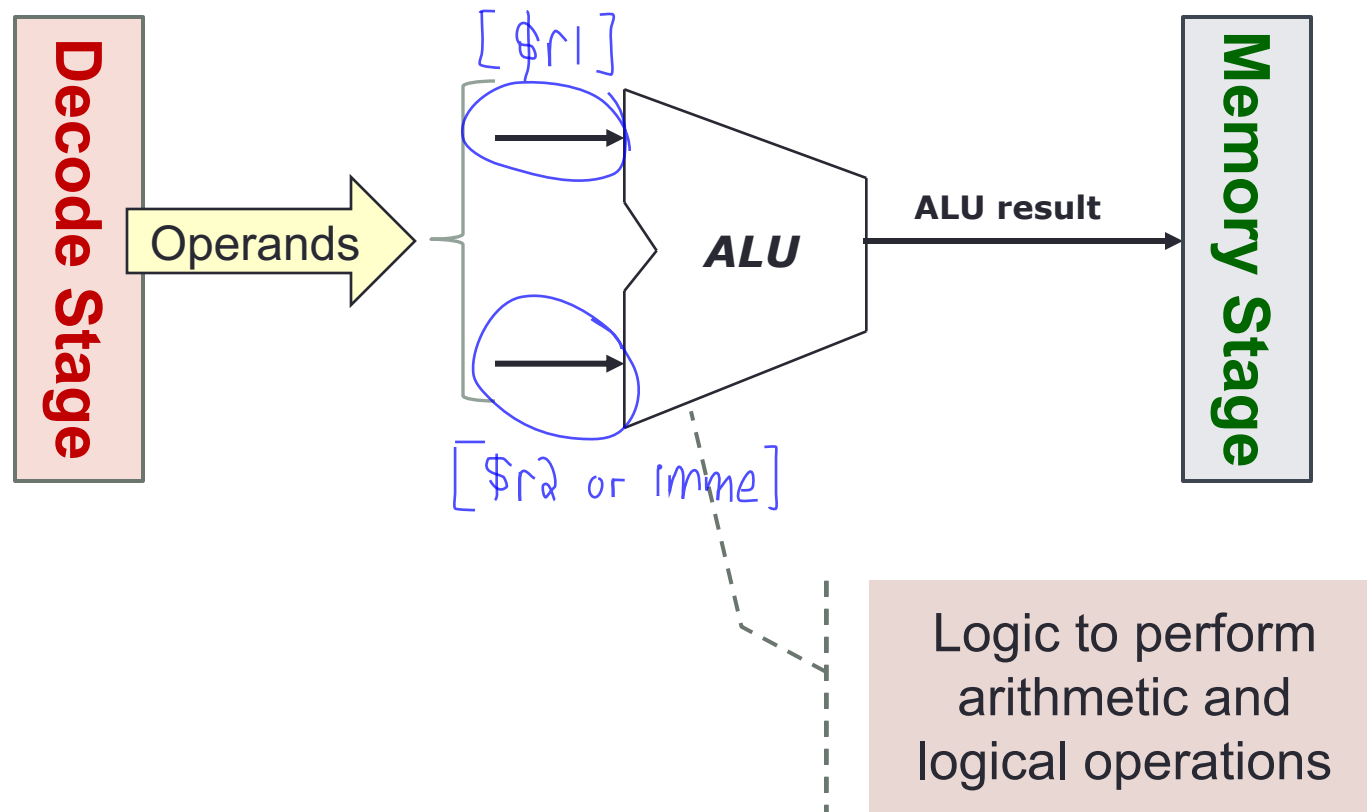


5.3 ALU Stage: Requirements

1. Fetch
2. Decode
3. **ALU**
4. Memory
5. RegWrite

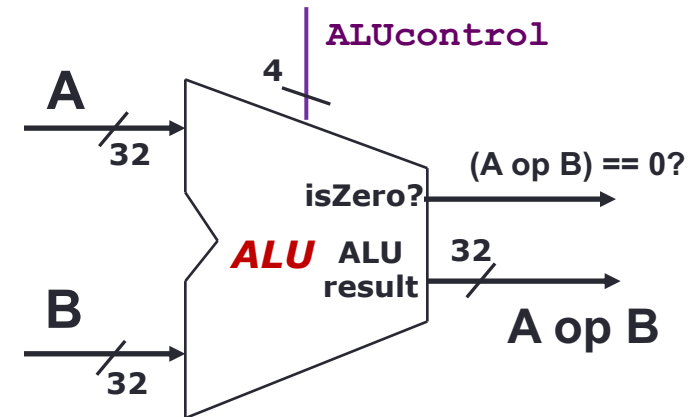
- Instruction **ALU Stage**:
 - ALU = Arithmetic-Logic Unit
 - Also called the **Execution stage**
 - Perform the real work for most instructions here
 - Arithmetic (e.g. **add**, **sub**), Shifting (e.g. **sll**), Logical (e.g. **and**, **or**)
 - Memory operation (e.g. **lw**, **sw**): Address calculation
 - Branch operation (e.g. **bne**, **beq**): Perform register comparison and target address calculation
- Input from previous stage (**Decode**):
 - Operation and Operand(s)
- Output to the next stage (**Memory**):
 - Calculation result

5.3 ALU Stage: Block Diagram



5.3 Element: Arithmetic Logic Unit

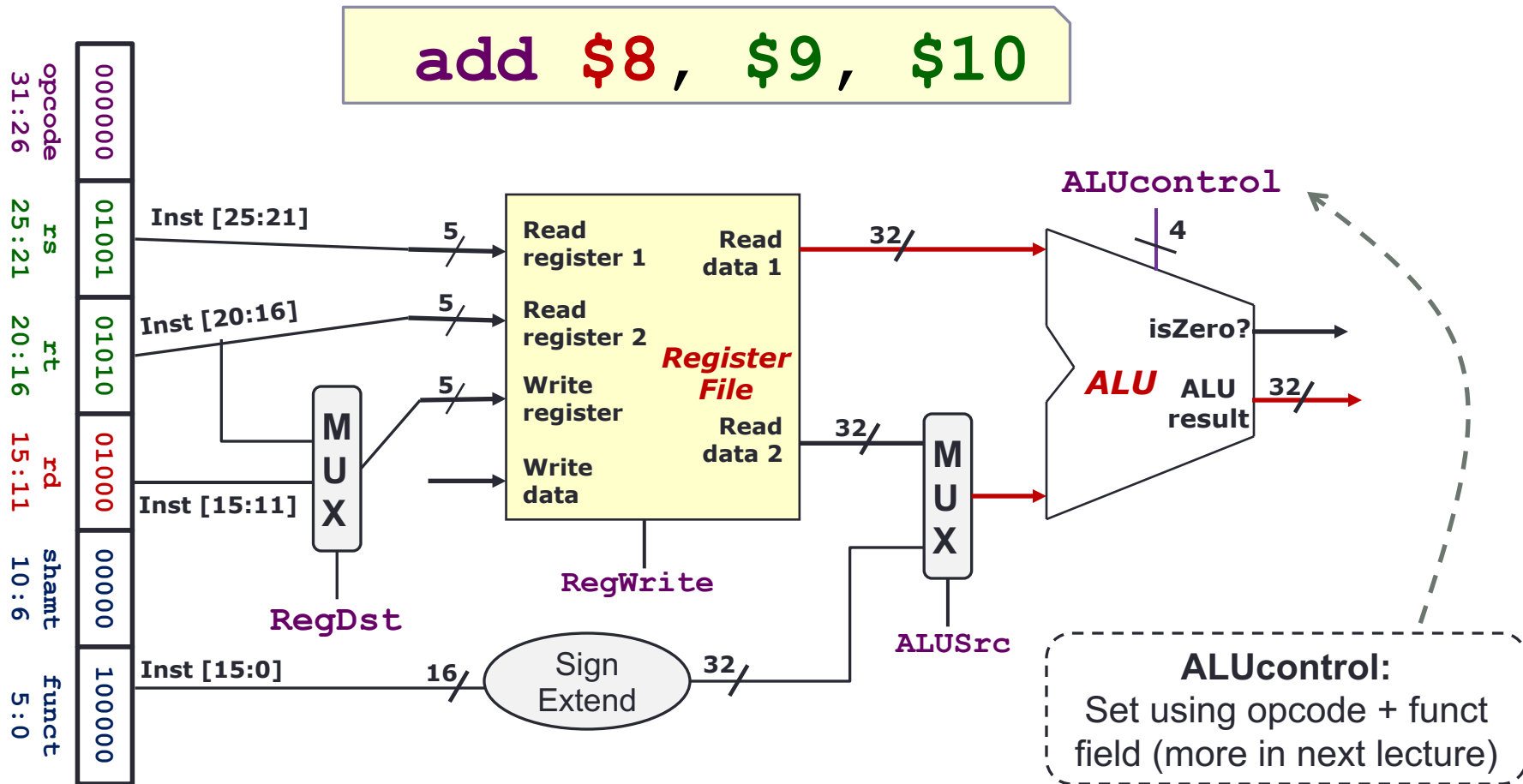
- **ALU (Arithmetic Logic Unit)**
 - Combinational logic to implement arithmetic and logical operations
- **Inputs:**
 - Two 32-bit numbers
- **Control:**
 - 4-bit to decide the particular operation
- **Outputs:**
 - Result of arithmetic/logical operation
 - A 1-bit signal to indicate whether result is zero



ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

5.3 ALU Stage: Non-Branch Instructions

- We can handle non-branch instructions easily:



5.3 ALU Stage: Branch Instructions

- Branch instruction is harder as we need to perform two calculations:

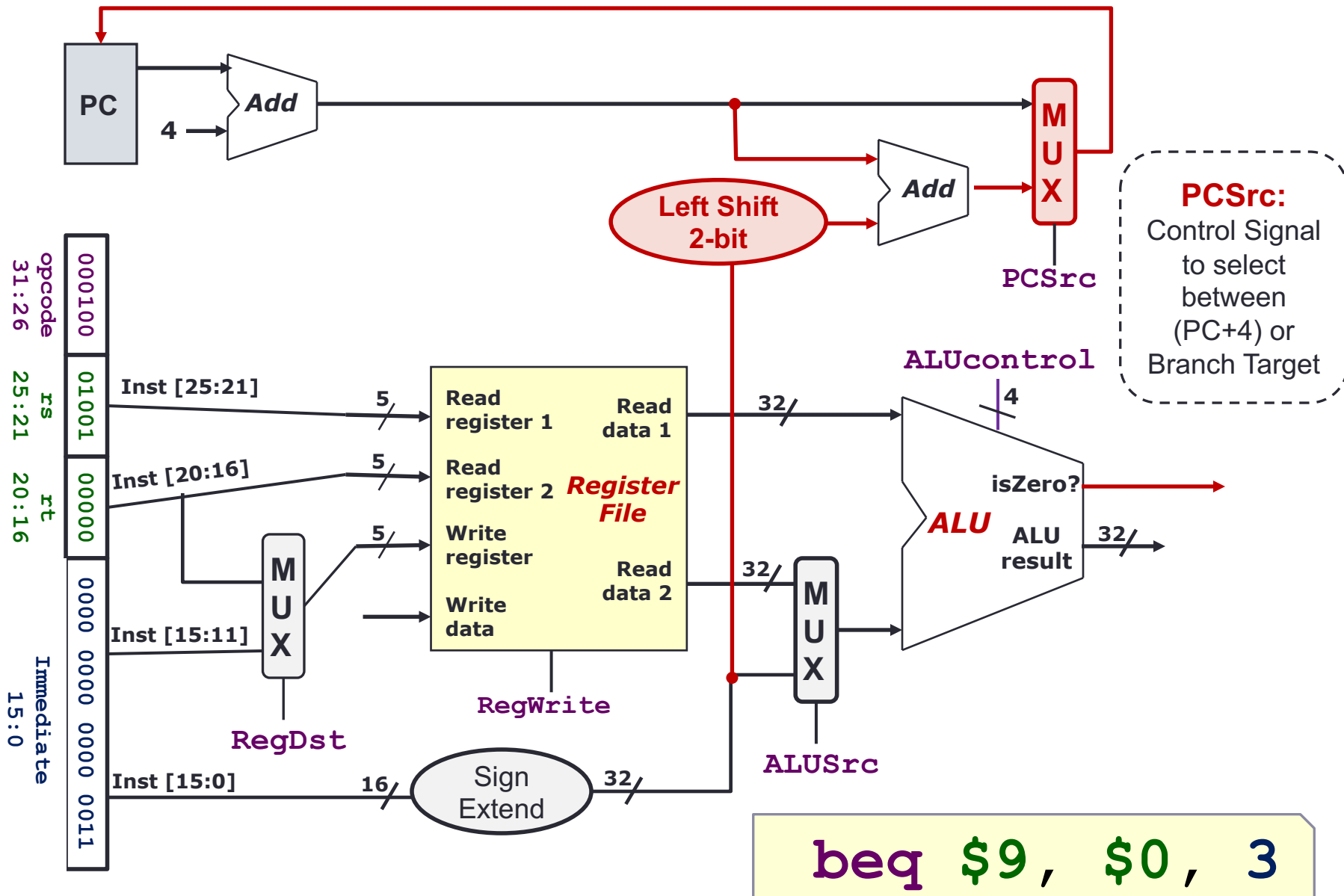
- Example: "**beq** \$9, \$0, 3"

1. Branch Outcome:

- Use ALU to compare the register
- The 1-bit "**isZero?**" signal is enough to handle equal/not equal check (how?)

2. Branch Target Address:

- Introduce additional logic to calculate the address
- Need **PC** (from **Fetch Stage**)
- Need **Offset** (from **Decode Stage**)

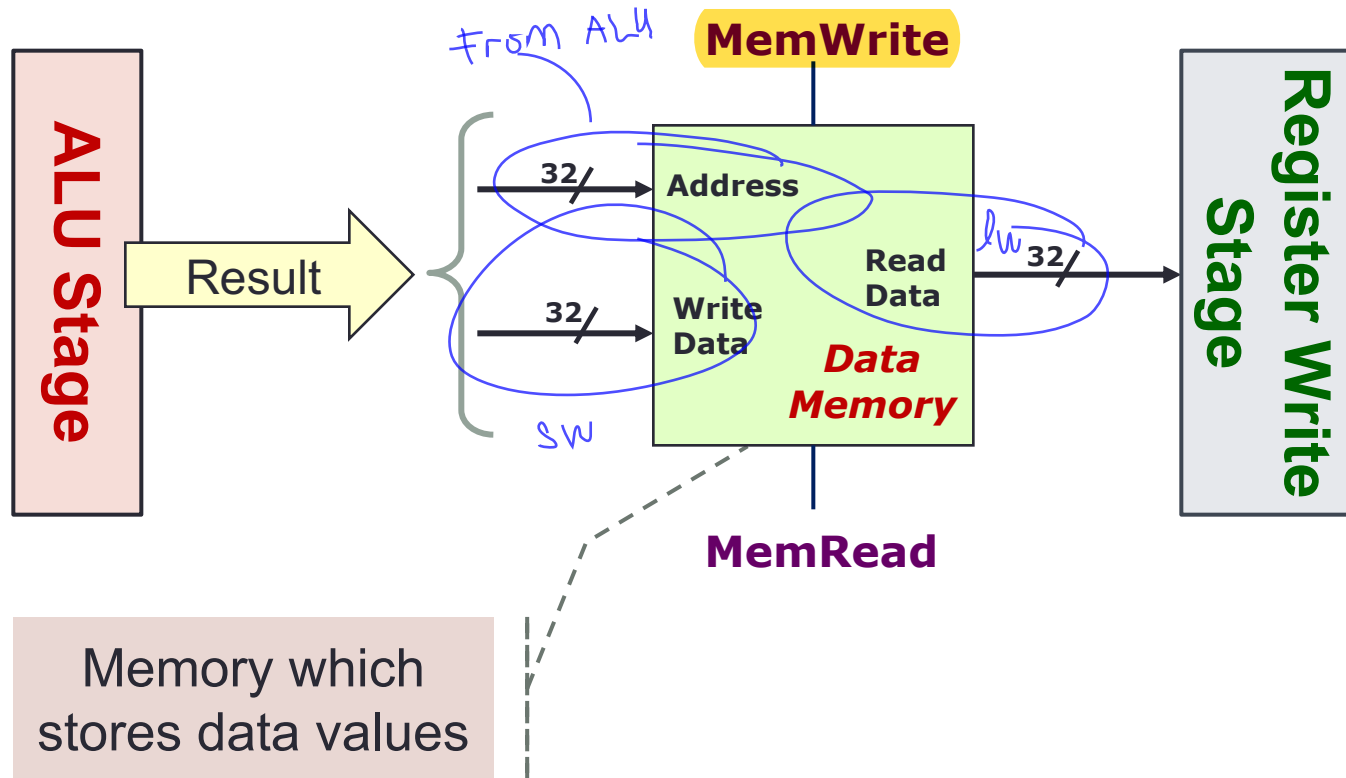


5.4 Memory Stage: Requirement

1. Fetch
2. Decode
3. ALU
- 4. Memory**
5. RegWrite

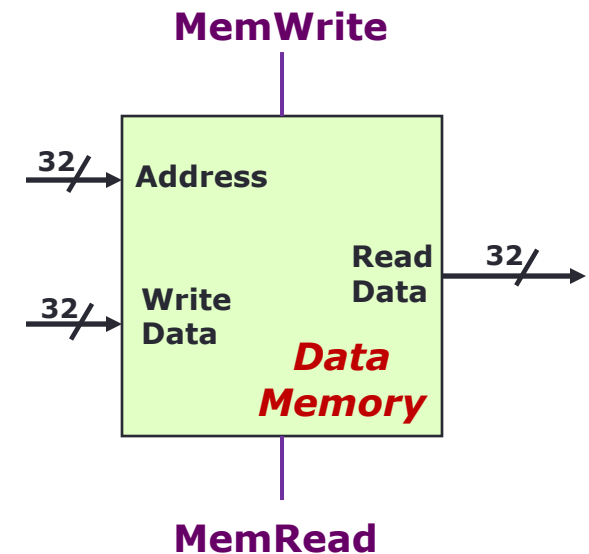
- Instruction **Memory Access Stage**:
 - Only the **load** and **store** instructions need to perform operation in this stage:
 - Use memory address calculated by ALU Stage
 - Read from or write to data memory
 - All other instructions remain idle
 - Result from ALU Stage will pass through to be used in Register Write stage (see section 5.5) if applicable
- Input from previous stage (**ALU**):
 - Computation result to be used as memory address (if applicable)
- Output to the next stage (**Register Write**):
 - Result to be stored (if applicable)

5.4 Memory Stage: Block Diagram



5.4 Element: Data Memory

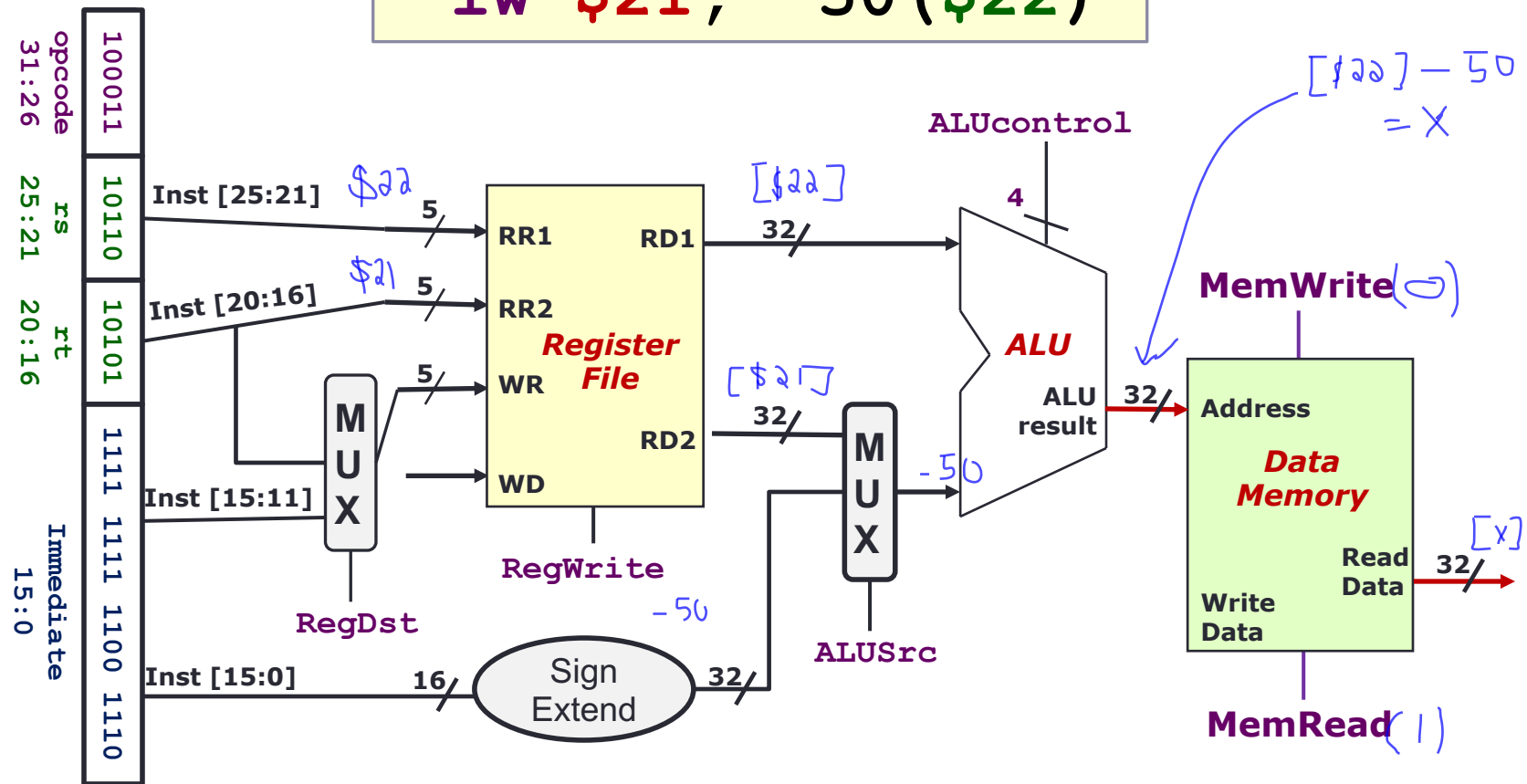
- Storage element for the data of a program
- **Inputs:**
 - Memory Address
 - Data to be written (Write Data) for store instructions
- **Control:**
 - Read and Write controls; only one can be asserted at any point of time
- **Output:**
 - Data read from memory (Read Data) for load instructions



5.4 Memory Stage: Load Instruction

- Only relevant parts of Decode and ALU Stages are shown

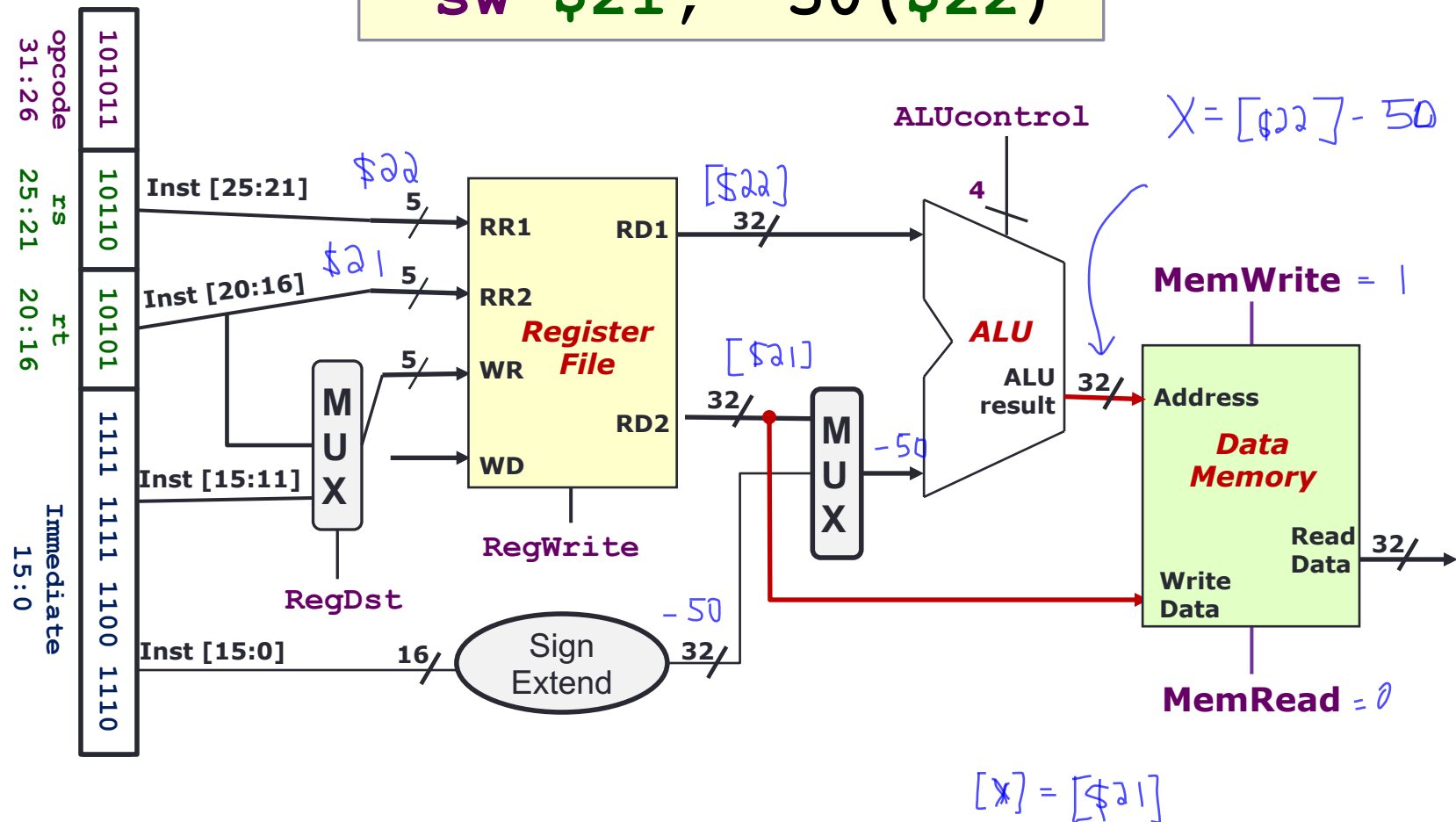
lw \$21, -50(\$22)



5.4 Memory Stage: Store Instruction

- Need **Read Data 2** (from Decode stage) as the **Write Data**

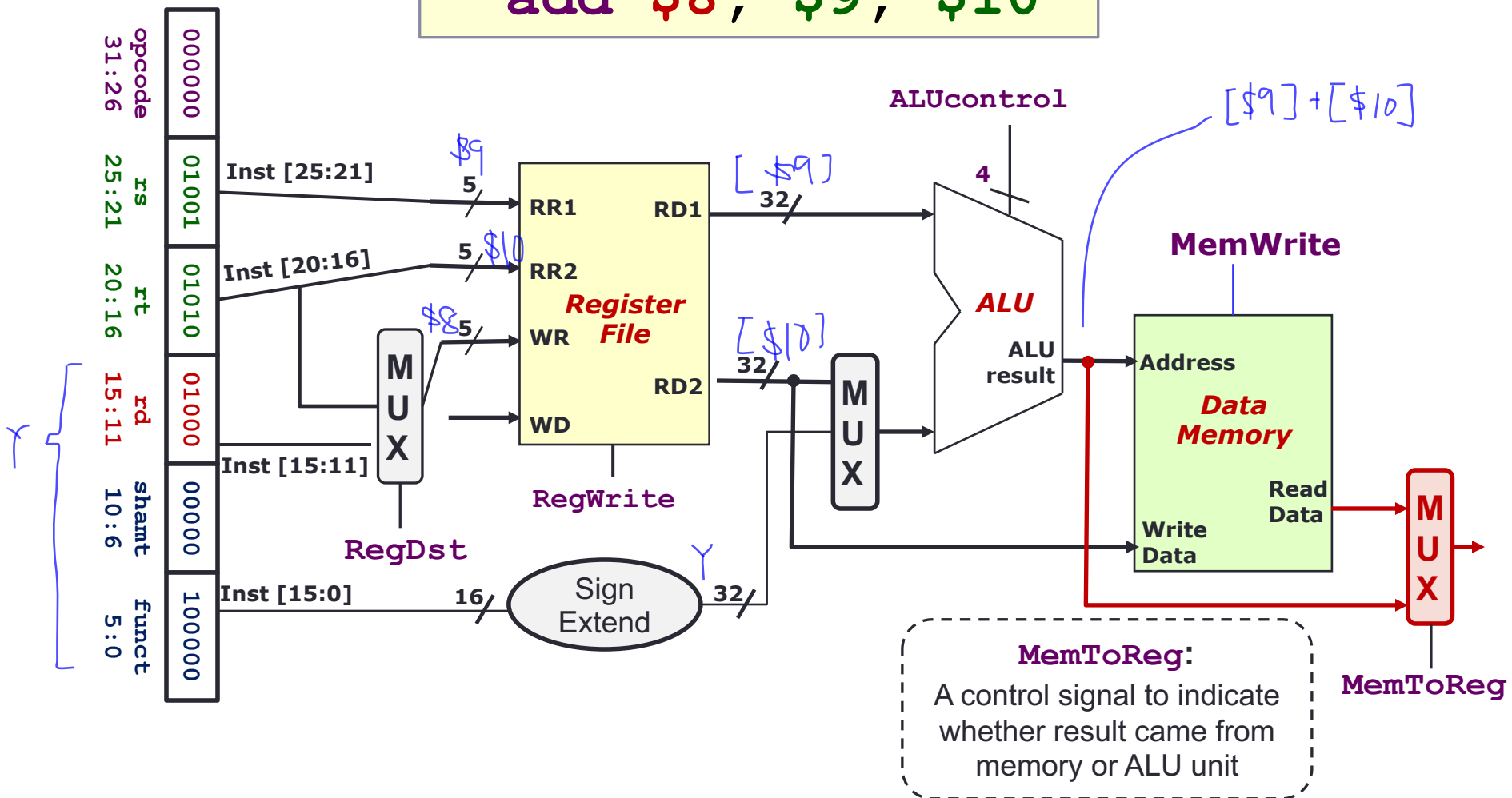
SW \$21, -50 (\$22)



5.4 Memory Stage: Non-Memory Inst.

- Add a multiplexer to choose the result to be stored

add \$8, \$9, \$10

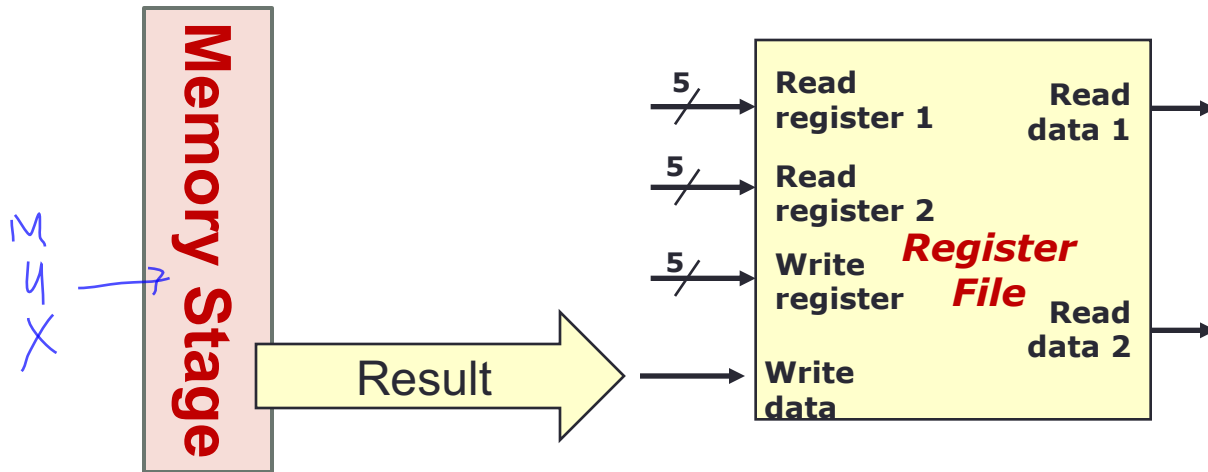


5.5 Register Write Stage: Requirements

1. Fetch
2. Decode
3. ALU
4. Memory
5. **RegWrite**

- **Instruction Register Write Stage:**
 - Most instructions write the result of some computation into a register
 - Examples: arithmetic, logical, shifts, loads, set-less-than
 - Need destination register number and computation result
 - Exceptions are stores, branches, jumps:
 - There are no results to be written
 - These instructions remain idle in this stage
- **Input from previous stage (Memory):**
 - Computation result either from memory or ALU

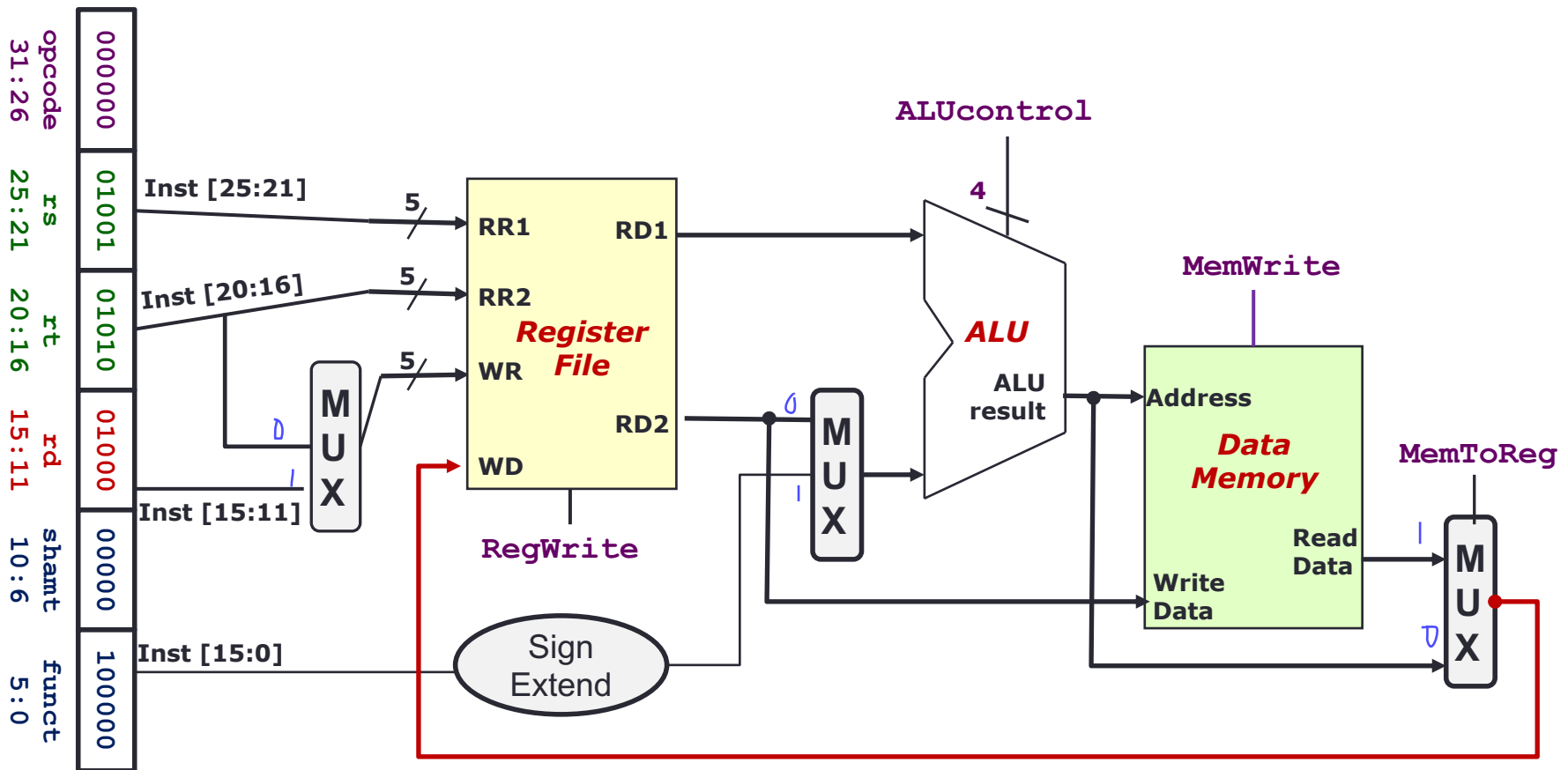
5.5 Register Write Stage: Block Diagram



- Result Write stage has no additional element:
 - Basically just route the correct result into register file
 - The **Write Register** number is generated way back in the **Decode Stage**

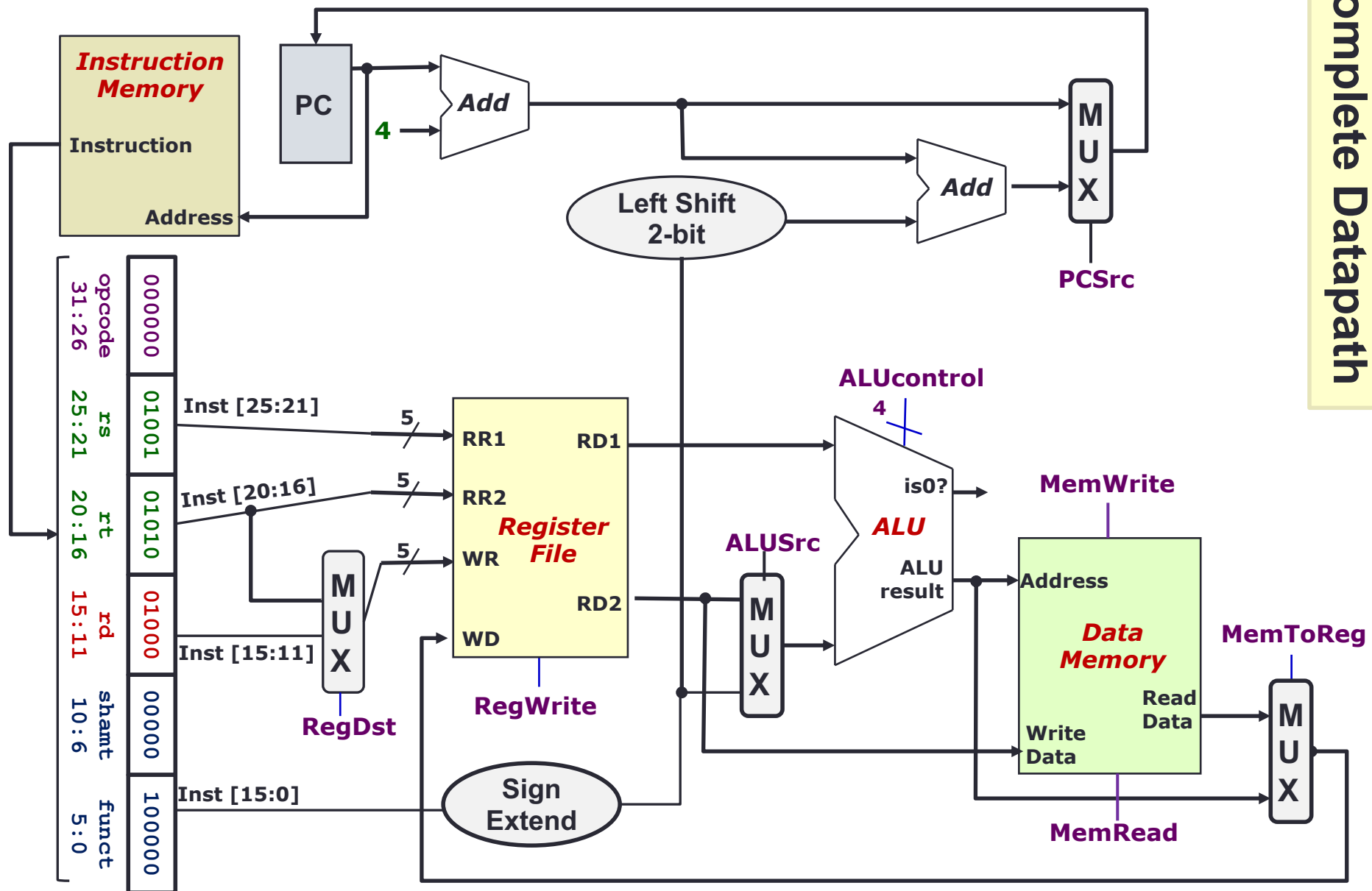
5.5 Register Write Stage: Routing

add \$8, \$9, \$10



6. The Complete Datapath!

- We have just finished “designing” the datapath for a subset of MIPS instructions:
 - Shifting and Jump are not supported
- Check your understanding:
 - Take the complete datapath and play the role of controller:
 - See how supported instructions are executed
 - Figure out the correct control signals for the datapath elements
- Coming up next: **Control**



Reading

- **The Processor: Datapath and Control**
 - COD Chapter 5 Sections 5.1 – 5.3 (3rd edition)
 - COD Chapter 4 Sections 4.1 – 4.3 (4th edition)



End of File