ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

COMPUTER ORGANISATION
(TỔ CHỨC MÁY TÍNH)

Processor: Datapath

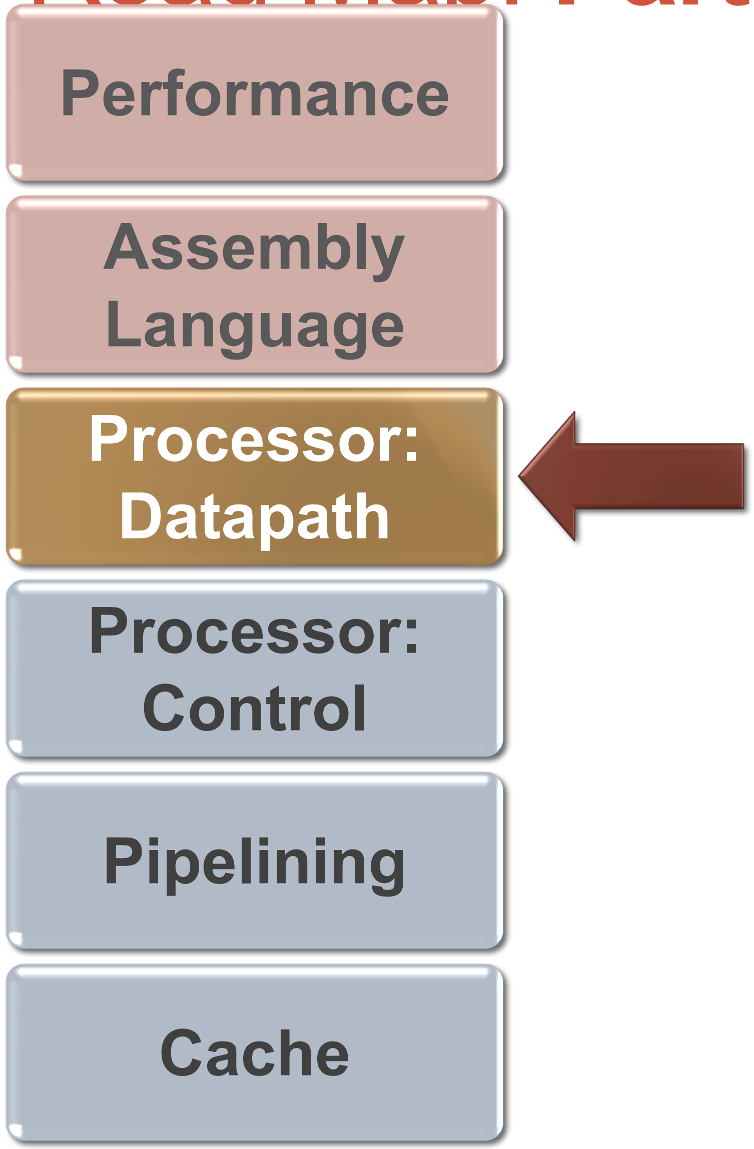
Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Road Map: **Part II**



- **Processor Datapath**
  - Generic Execution Stages
  - MIPS Execution Stages
  - Constructing Datapath

Building a Processor: Datapath & Control

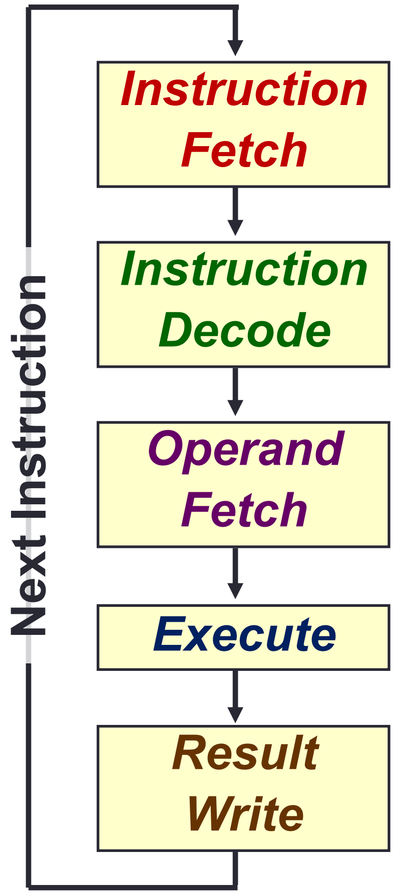
- Two major components for a processor:
- **Datapath**
  - Collection of components that process data
  - Performs the arithmetic, logical and memory operations
- **Control**
  - Tells the datapath, memory, and I/O devices what to do according to program instructions

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 here
  - Left as exercises ☺

Recap: Instruction Execution Cycle

Important!



- **Fetch:**
  - Get instruction from memory
  - Address is in **P**rogram **C**ounter (PC) Register
- **Decode:**
  - Find out the operation required
- **Operand Fetch:**
  - Get operand(s) needed for operation
- **Execute:**
  - Perform the required operation
- **Result Write (WriteBack):**
  - Store the result of the operation

MIPS Instruction Executions

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

	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, ofst`
Fetch / **Decode**	***standard***	***standard***	***standard***
Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
Execute	*Result = opr1 + opr2*	○ *MemAddr* = opr1 + opr2 ○ Use *MemAddr* to read data from memory	*Taken = (opr1 == opr2)?* *Target* = (**PC**+4) or (**PC**+4) + **ofst**
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	**PC** = *Target*

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

5-STAGE MIPS EXECUTION

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

	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, ofst`
Fetch	Read inst. at [**PC**]	Read inst. at [**PC**]	Read inst. at [**PC**]
Decode & Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
ALU	*Result = opr1 + opr2*	*MemAddr* = opr1 + opr2	*Taken* = (*opr1* == *opr2*)? *Target* = (**PC**+4) or (**PC**+4) + **ofst**
Memory Access		Use *MemAddr* to read datefrom memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	**PC** = *Target*

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

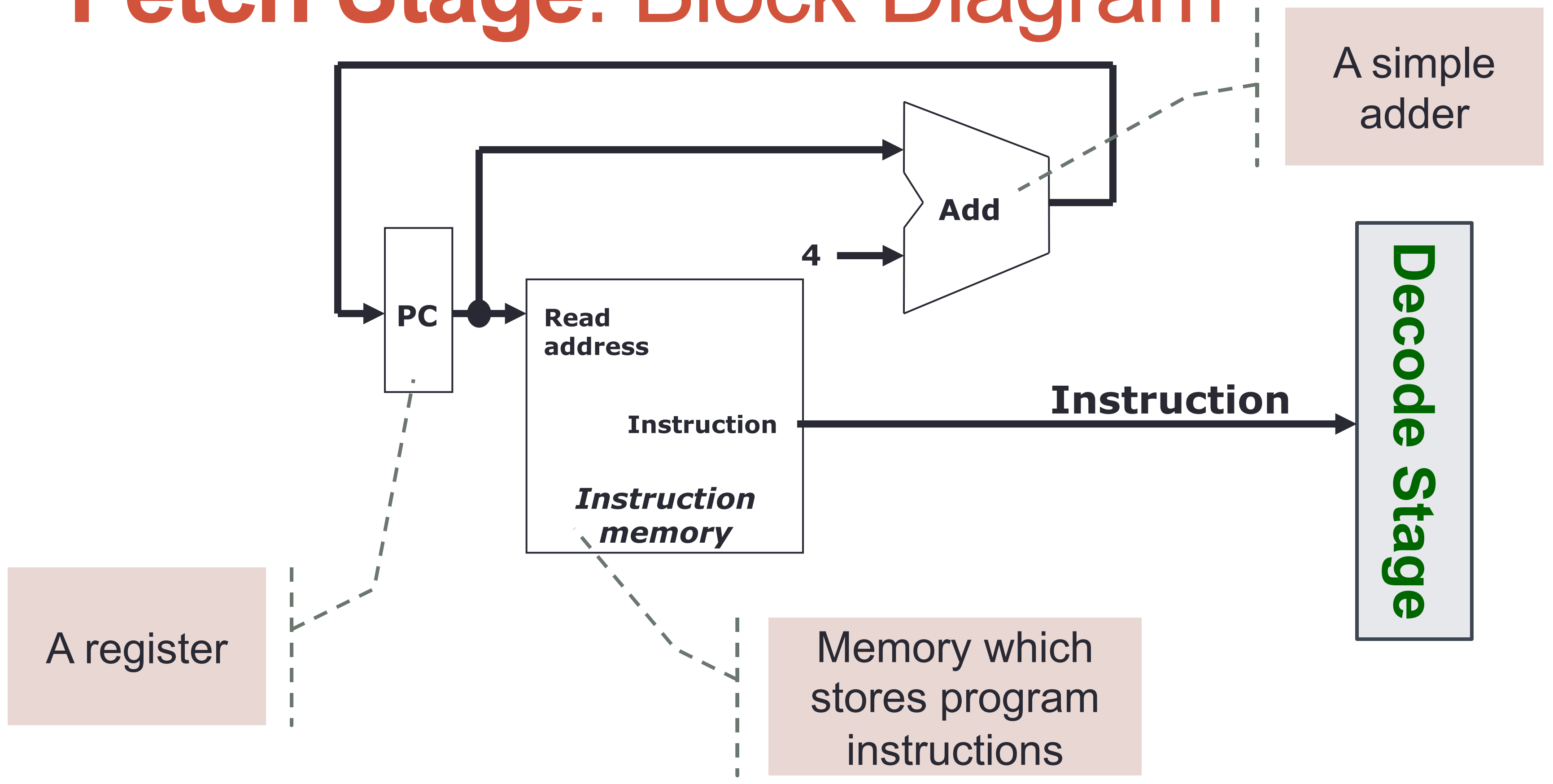
➔ Study the design from the viewpoint of a designer, instead of a “tourist” ☺

Fetch Stage: Requirements

- Instruction **Fetch Stage**:
  1. Use the **P**rogram **C**ounter (**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

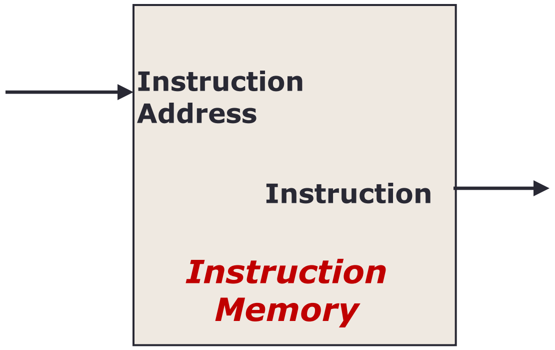
- Output to the next stage (**Decode**):
  - The instruction to be executed

Fetch Stage: Block Diagram

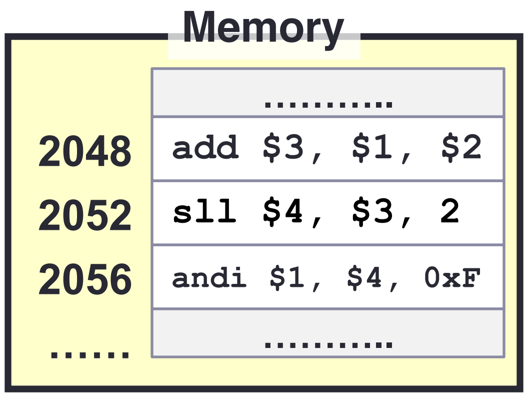


Element: **Instruction Memory**

- Storage element for the instructions
  - Recall: **sequential circuit**
  - Has an internal state that stores information
  - Clock signal is assumed and not shown



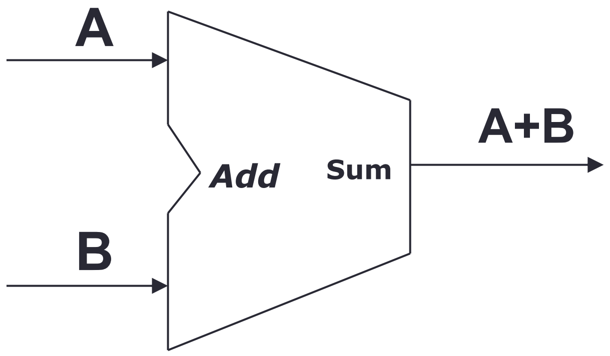
- Supply instructions given the address
  - Given instruction address M as input, the memory outputs the content at address M
  - Conceptual diagram of the memory layout is given on the right ➔



Element: **Adder**

- Combinational logic to implement the addition of two numbers

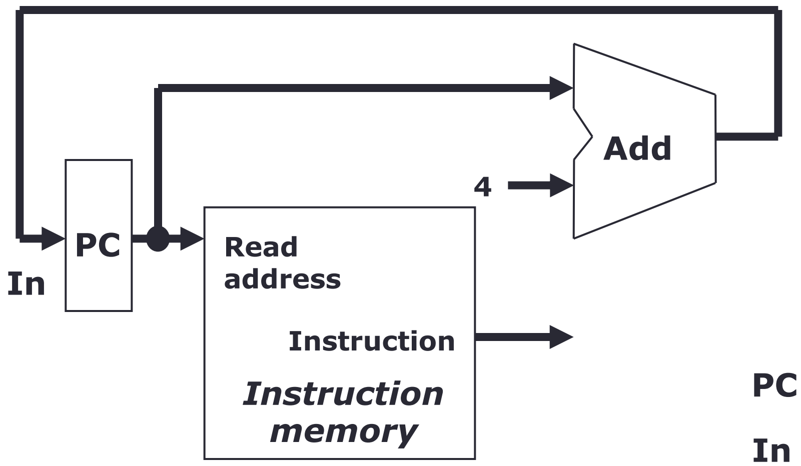
- **Inputs:**
  - Two 32-bit numbers **A**, **B**
- **Output:**
  - Sum of the input numbers, **A + B**

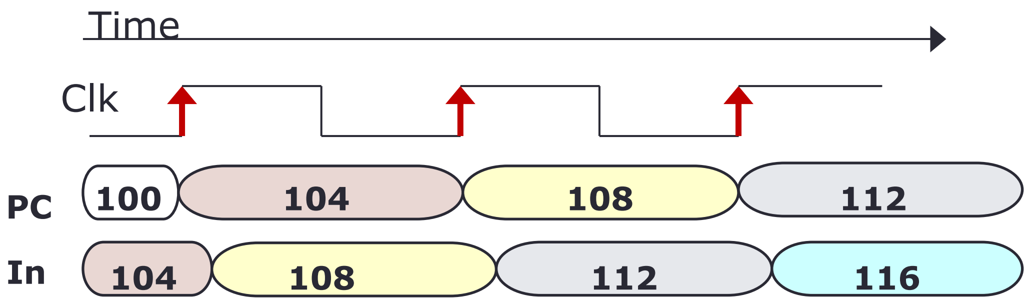


- Just a 32-bit version of the adder discussed in first part of the course ☺

The Idea of Clocking

- It seems that we are reading and updating the PC at the same time:
  - How can it works 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**

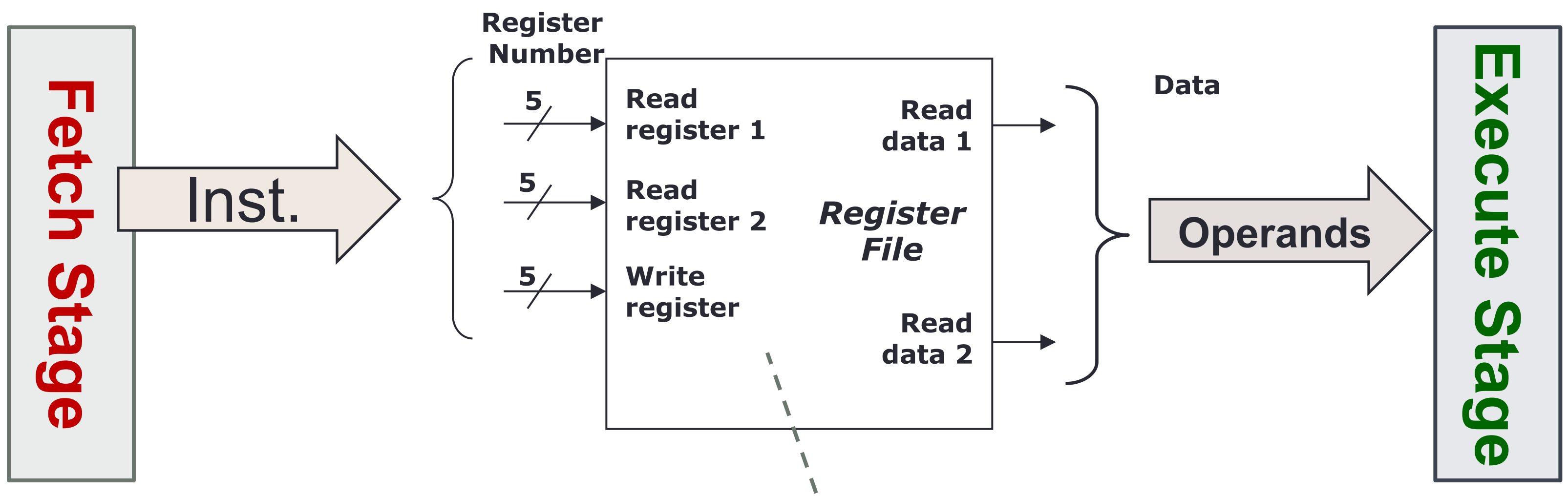




Decode Stage: Requirement

- 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

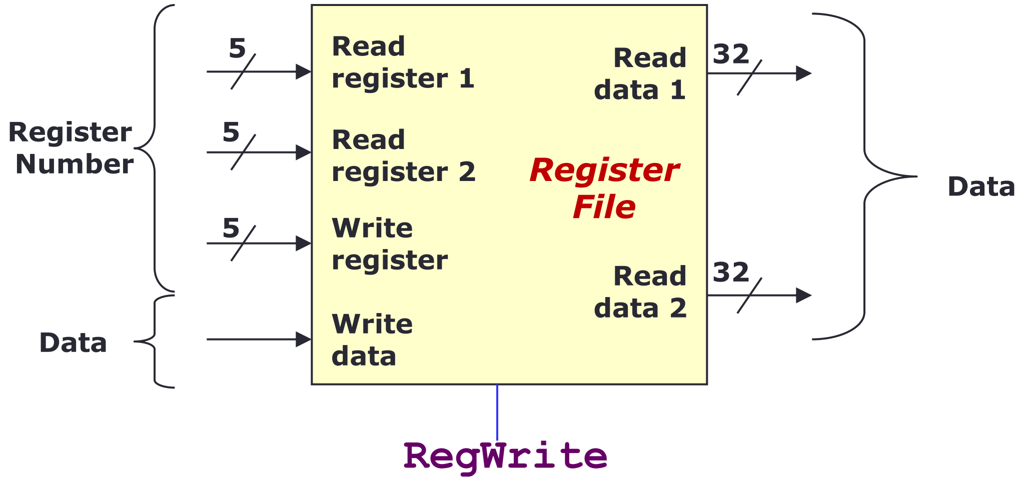
Decode Stage: Block Diagram



Collection of registers, known as **register file**

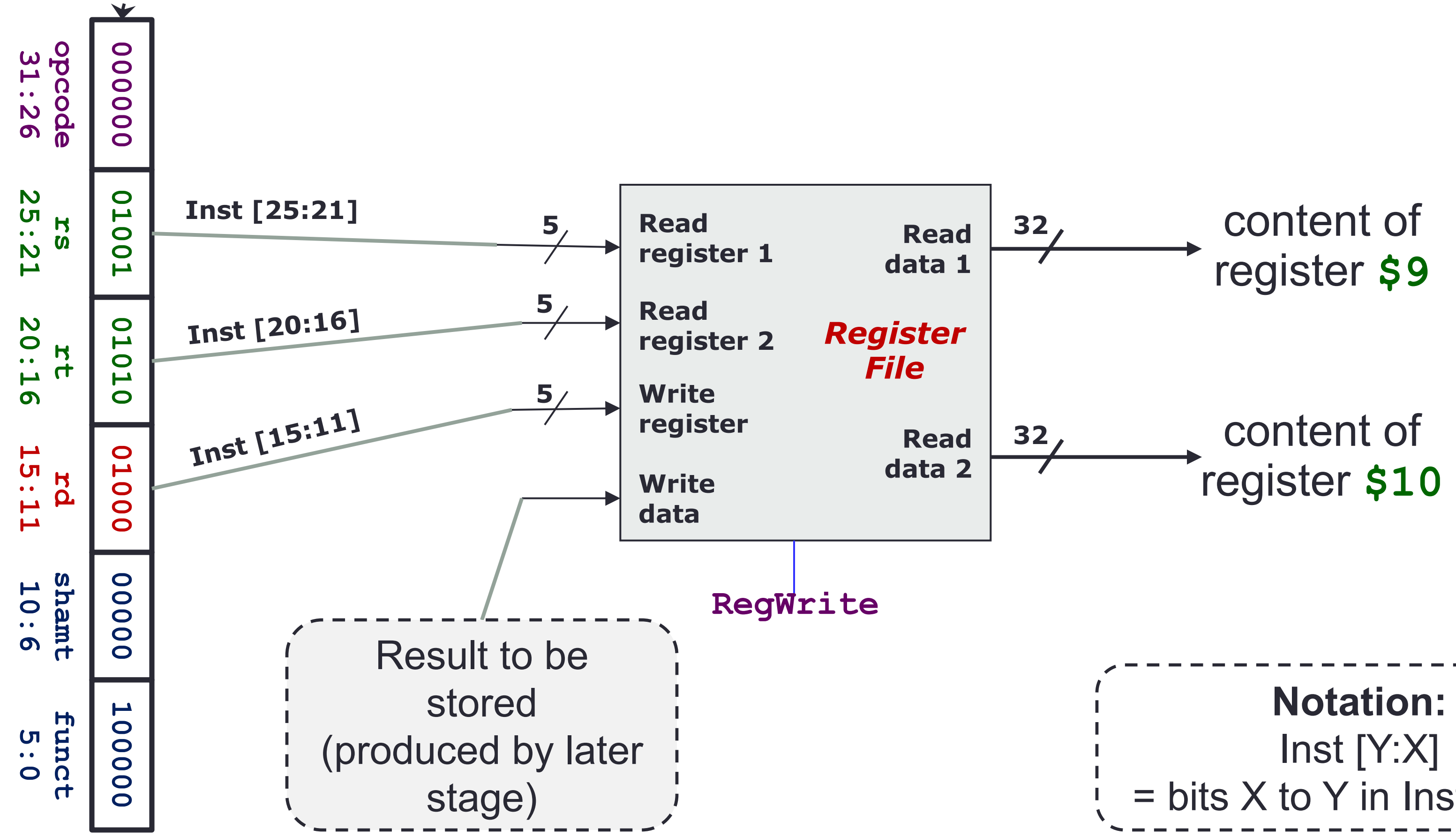
Element: **Register File**

- A collection of 32 registers:
  - Each is 32-bit wide and 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



Decode Stage: **R-Type Instruction**

```
add $8, $9, $10
```

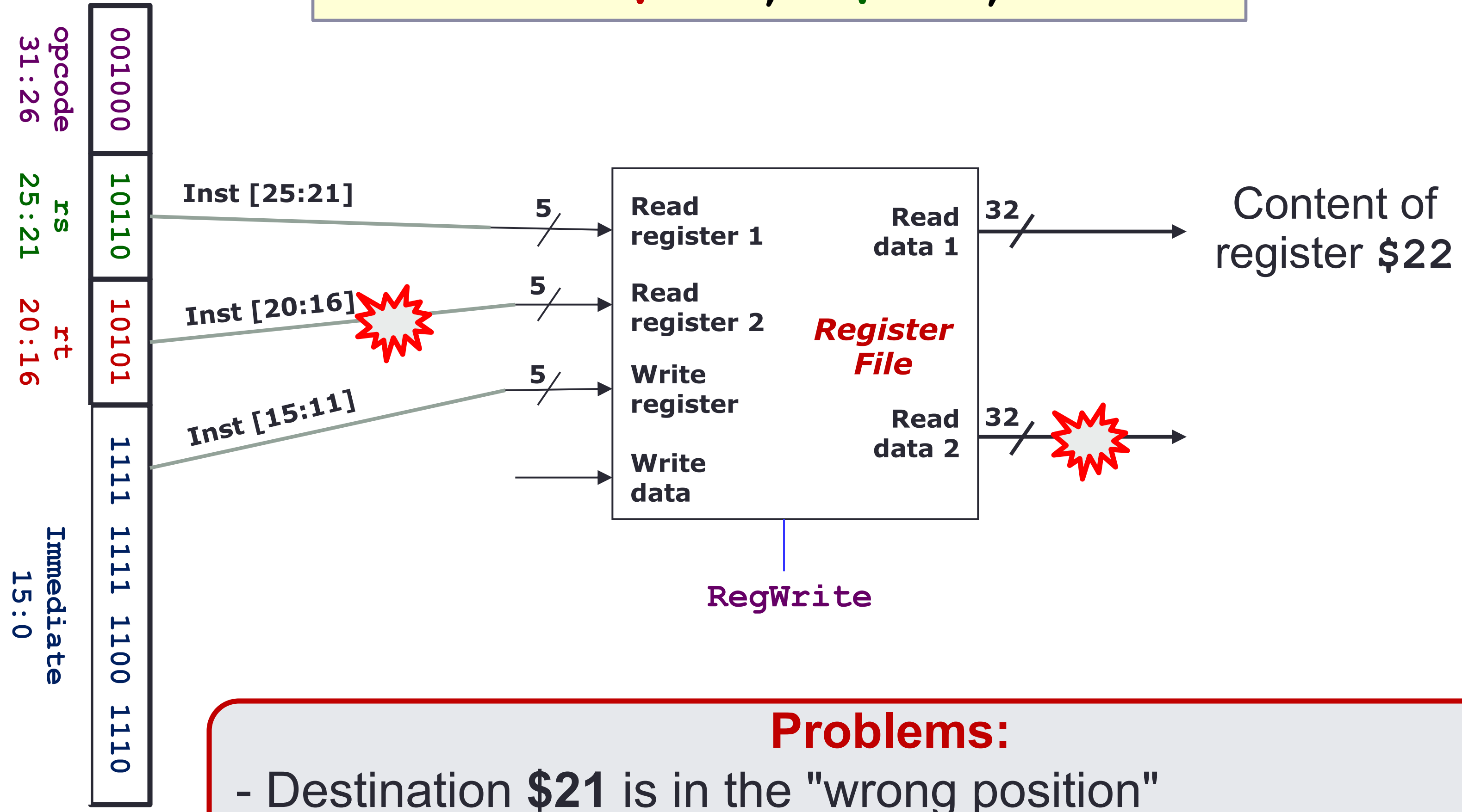


Result to be stored (produced by later stage)

Notation:
Inst [Y:X]
= bits X to Y in Instruction

Decode Stage: **I-Type Instruction**

```
addi $21, $22, -50
```

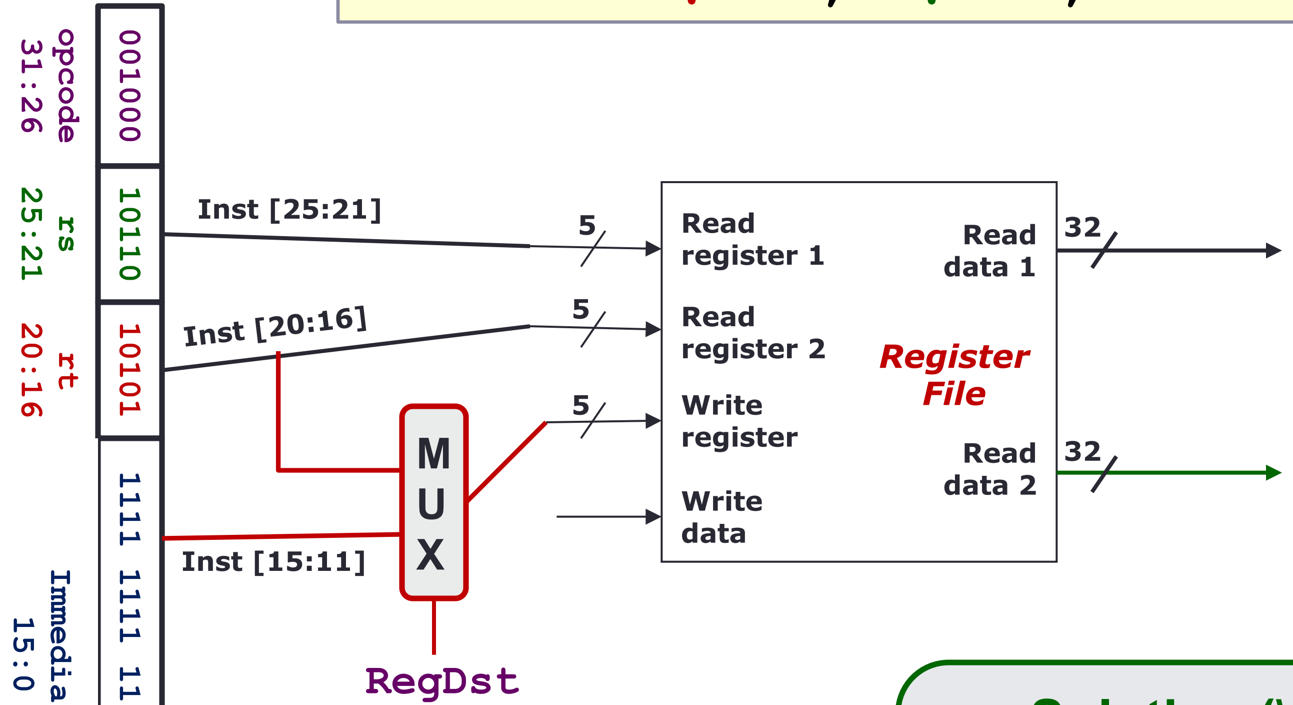


Problems:

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

Decode Stage: **Choice in Destination**

```
addi $21, $22, -50
```

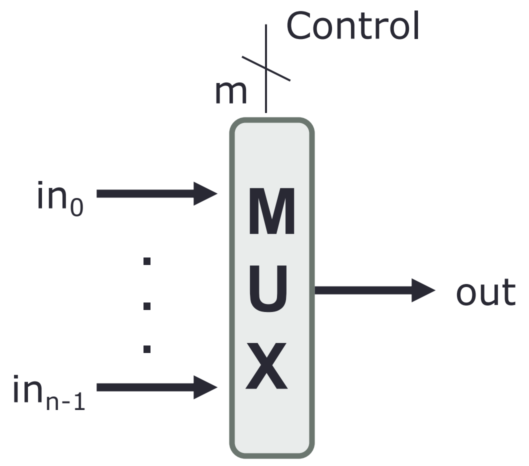


RegDst:
A control signal to choose either Inst[20:16] or Inst[15:11] as the write register number

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

Recap: **Multiplexer**

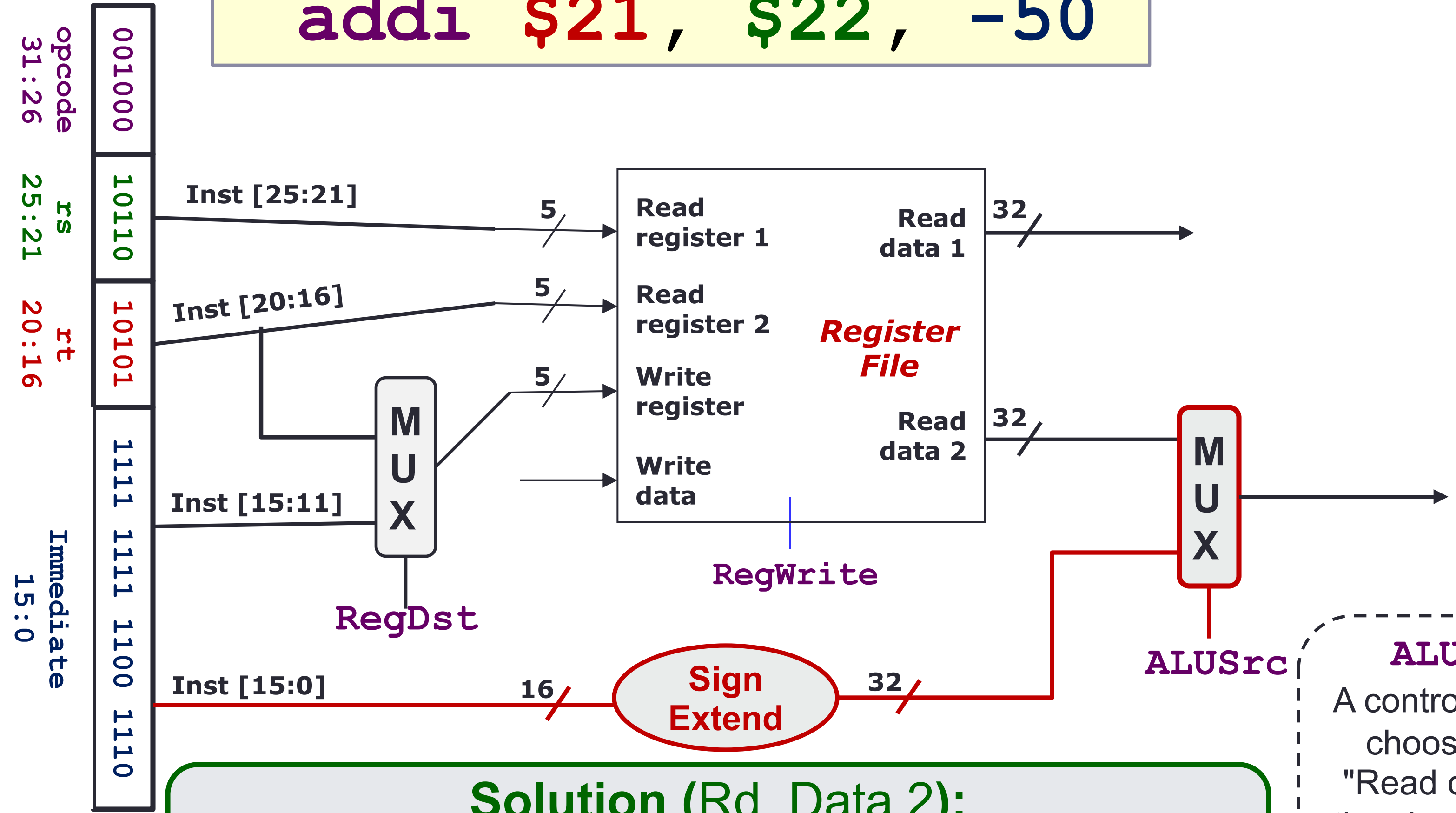
- **Function:**
  - Selects one input from multiple input lines
- **Inputs:**
  - ***n*** lines of same width
- **Control:**
  - ***m*** bits where n = 2^m
- **Output:**
  - Select i^th input line if control=i



Control=0 → select in_0
Control=3 → select in_3

Decode Stage: **Choice in Data 2**

```
addi $21, $22, -50
```

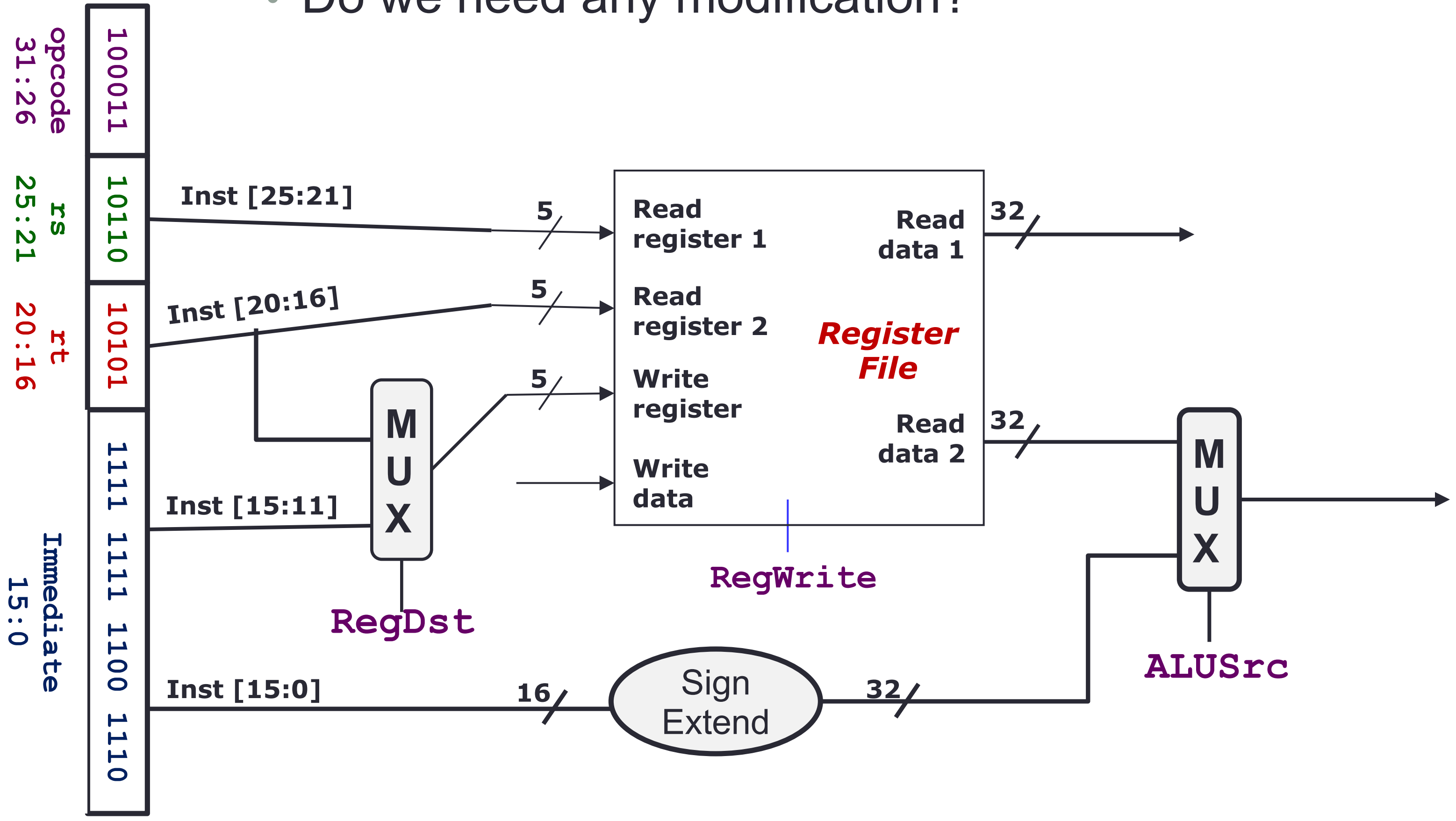


Solution (Rd. Data 2):
Use a **multiplexer** to choose the correct operand 2.
Sign extend the 16-bit immediate value to 32-bit

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

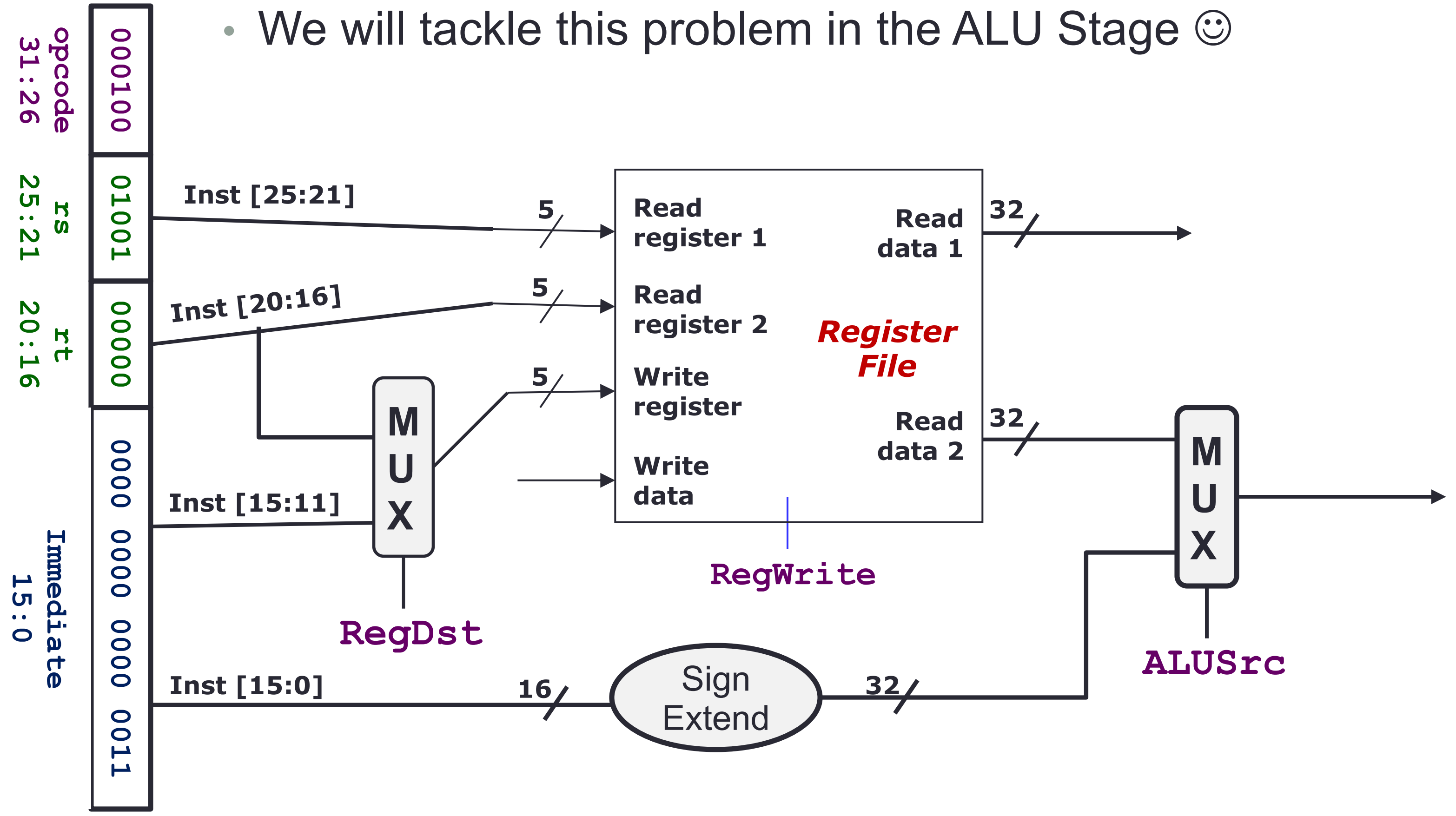
Decode Stage: Load Word Instruction

- Try it out: "`lw $21, -50($22)`"
  - Do we need any modification?

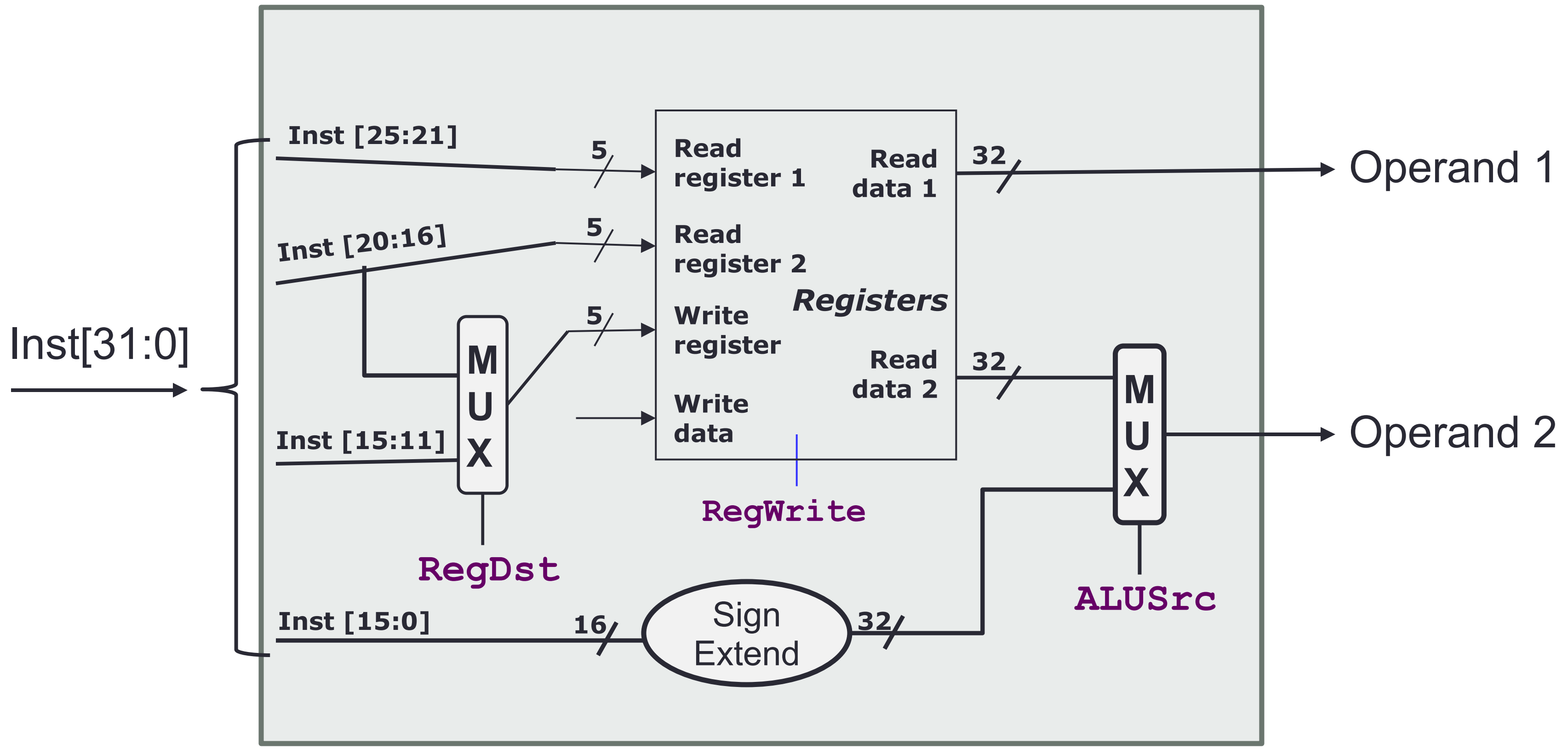


Decode Stage: Branch Instruction

- Example: "`beq $9, $0, 3`"
  - Need to calculate branch outcome and target at the same time!
  - We will tackle this problem in the ALU Stage ☺



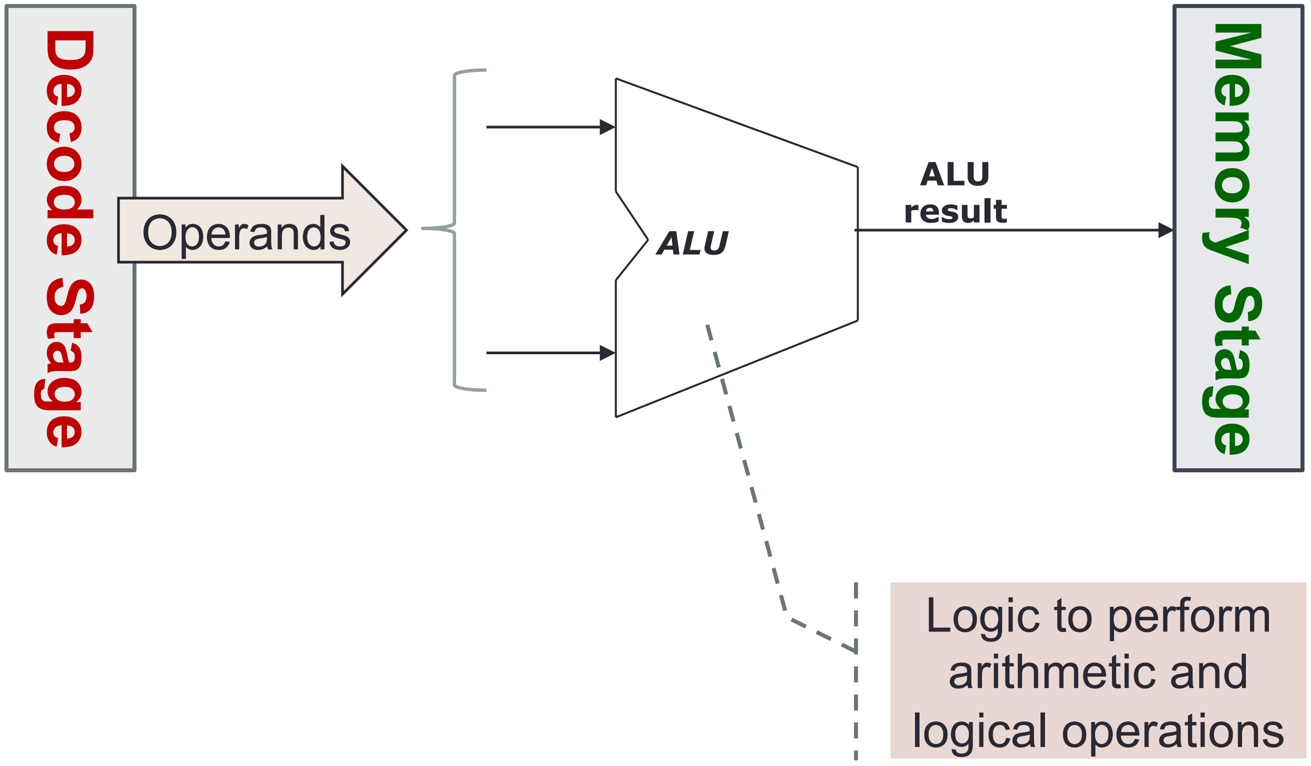
Decode Stage: Summary



ALU Stage: Requirement

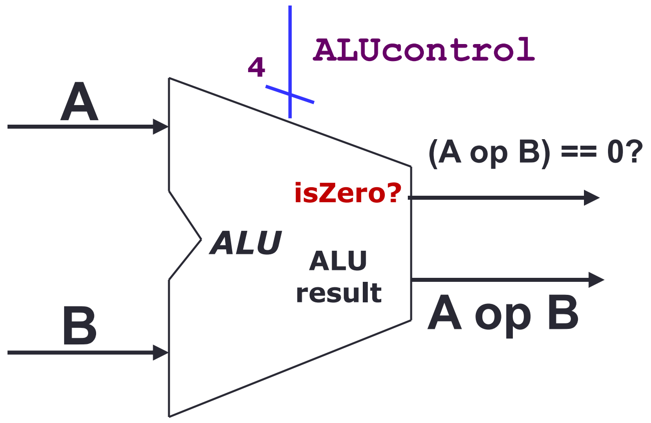
- Instruction **ALU Stage**:
  - ALU = Arithmetic-Logic Unit
  - Perform the real work for most instructions here
    - Arithmetic (e.g. **`add`**, **`sub`**), Shifting (e.g. **`sll`**), Logical (e.g. **`and`**, **`or`**)
    - Memory operation (e.g. **`lw`**, **`sw`**): Address calculation
    - Branch 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

ALU Stage: Block Diagram



Element: **Arithmetic Logical Unit**

- **ALU (Arithmetic-logical unit)**
  - Combinational logic to implement arithmetic and logical operations
- **Inputs:**
  - Two 32-bit numbers



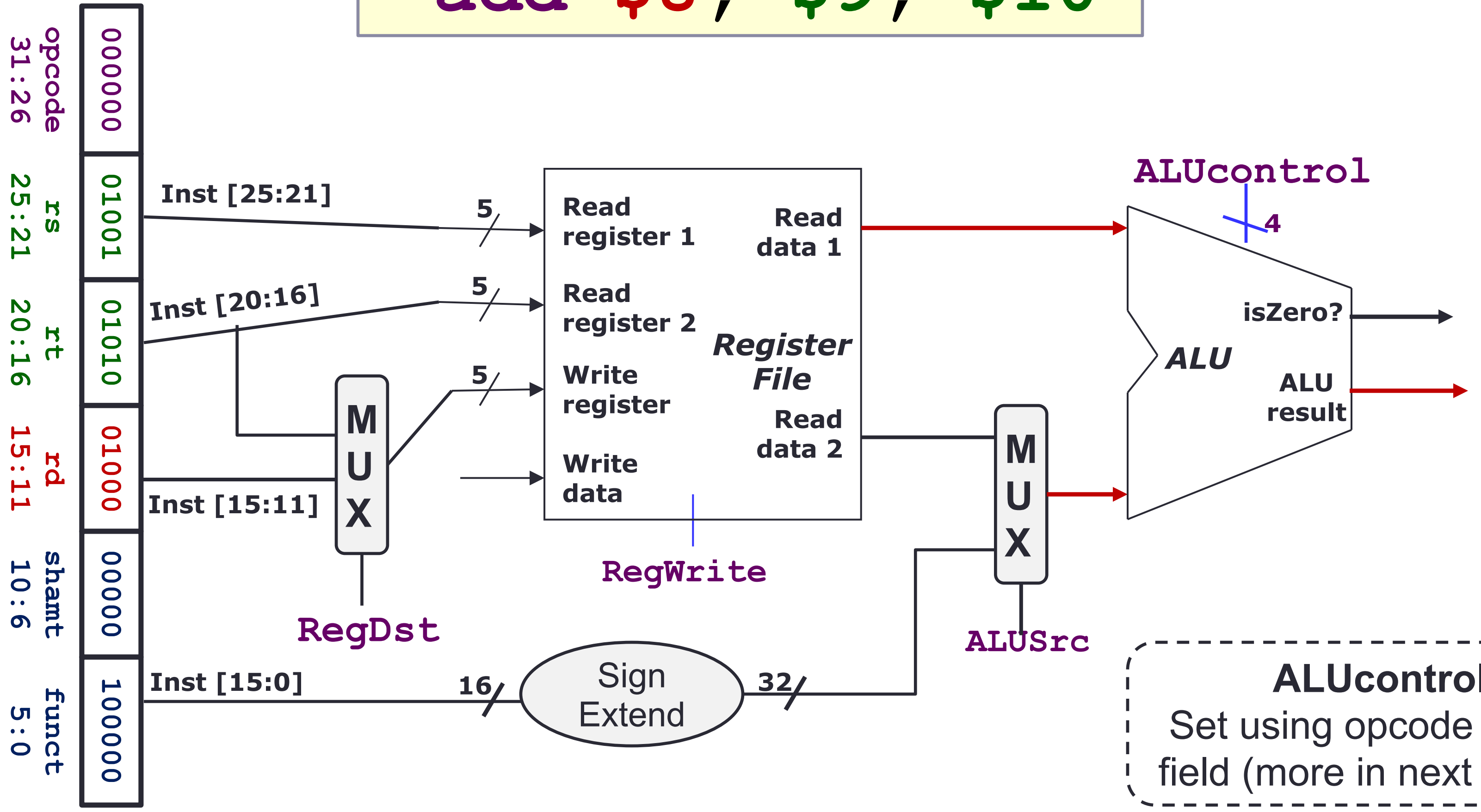
- **Control:**
  - 4-bit to decide the particular operation
- **Outputs:**
  - Result of arithmetic/logical operation
  - A 1-bit signal to indicate whether result is zero

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

ALU Stage: **Non-Branch Instructions**

- We can handle non-branch instructions easily:

`add $8, $9, $10`

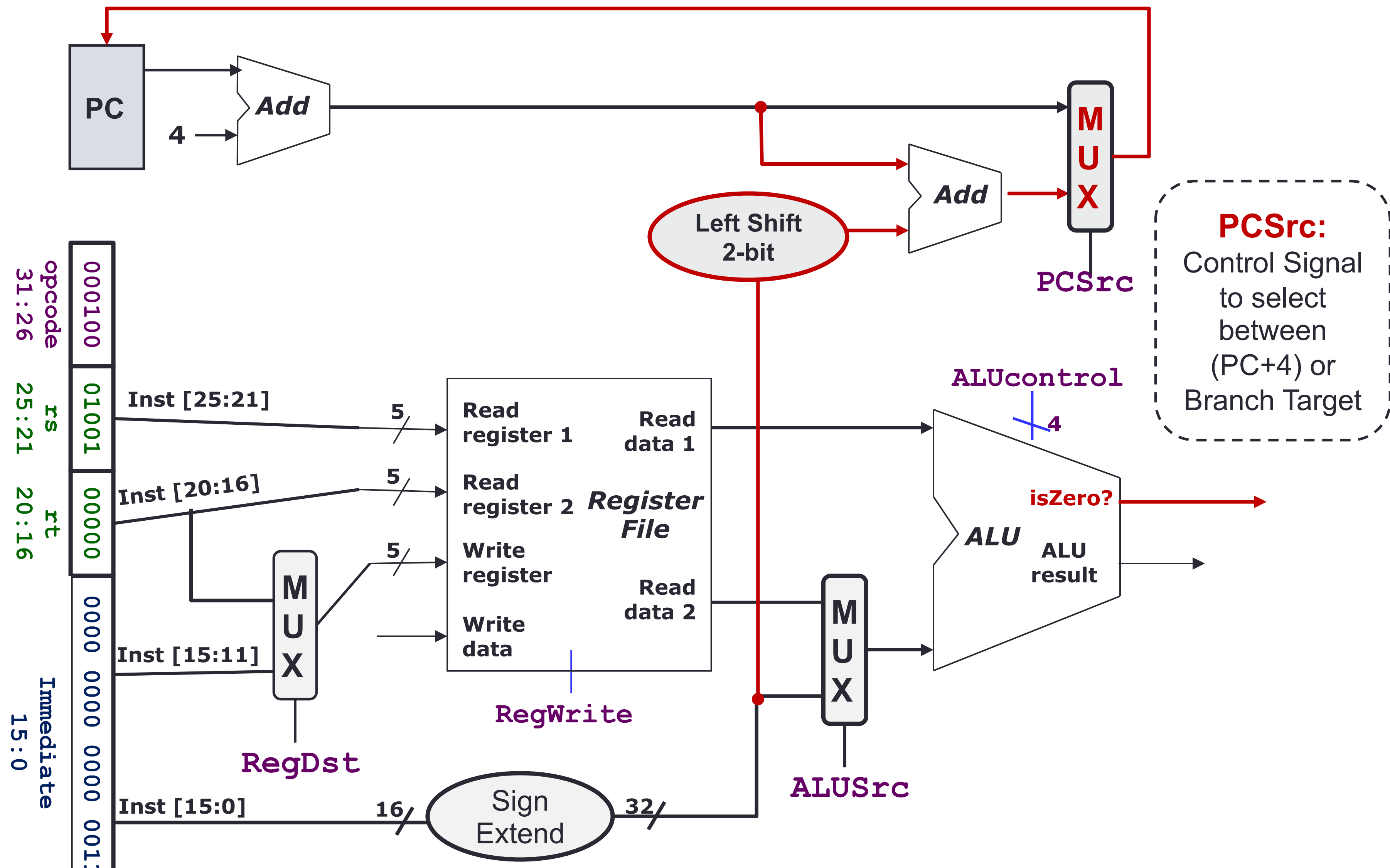


ALUcontrol:
Set using opcode + funct field (more in next lecture)

ALU Stage: **Brach Instructions**

- Branch instruction is harder as we need to perform two calculations:
- Example: "`beq $9, $0, 3`"
  1. **Branch Outcome:**
     - Use ALU unit 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)

Complete ALU Stage



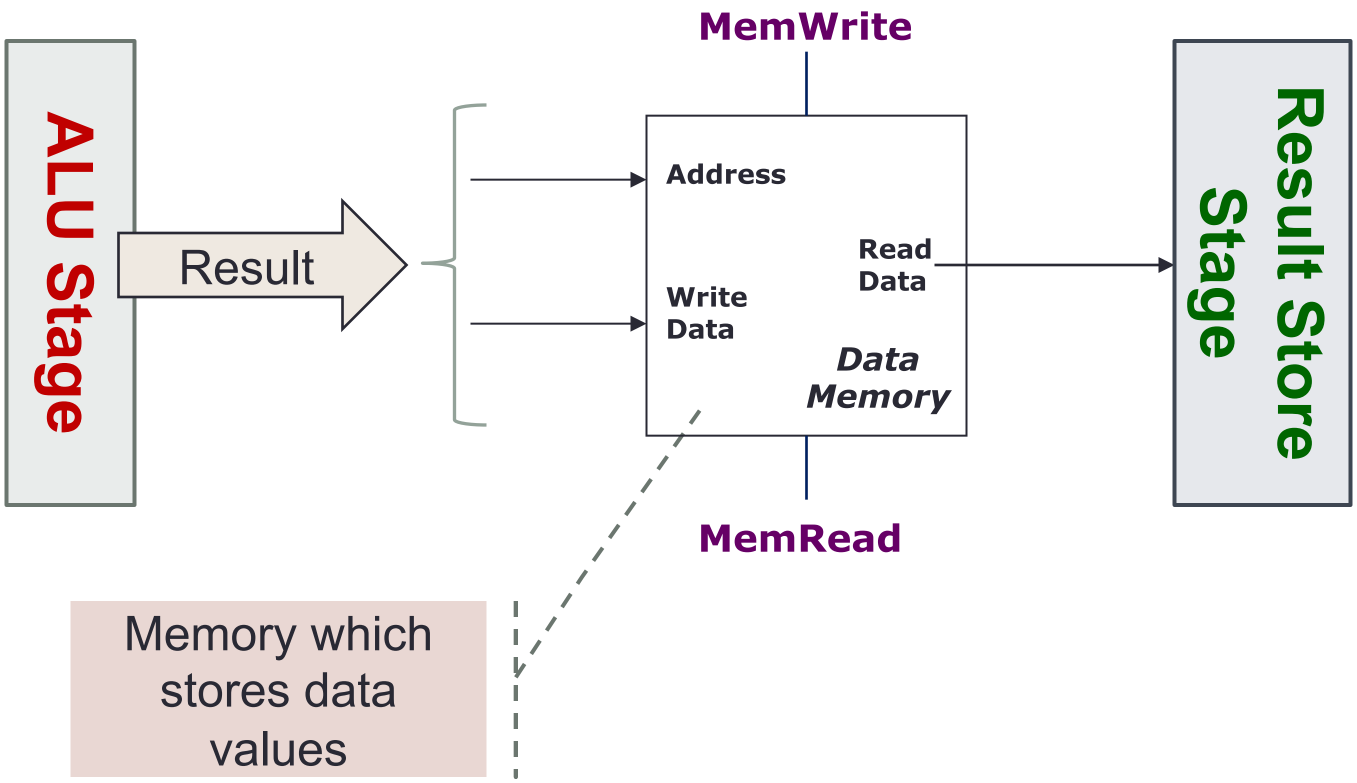
PCSrc: Control Signal to select between (PC+4) or Branch Target

E.g. "beq $9, $0, 3"

Memory Stage: Requirement

- 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 Result Store (Writeback) stage if applicable
- Input from previous stage (**ALU**):
  - Computation result to be used as memory address (if applicable)
- Output to the next stage (**Writeback**):
  - Result to be stored (if applicable)

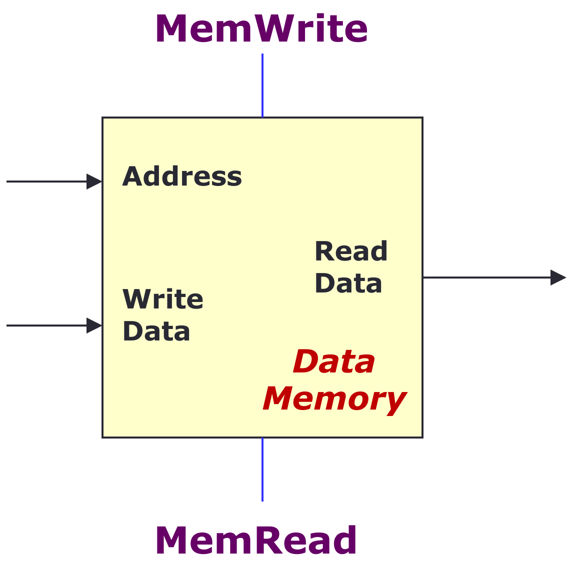
Memory Stage: Block Diagram



Element: **Data Memory**

- Storage element for the data of a program

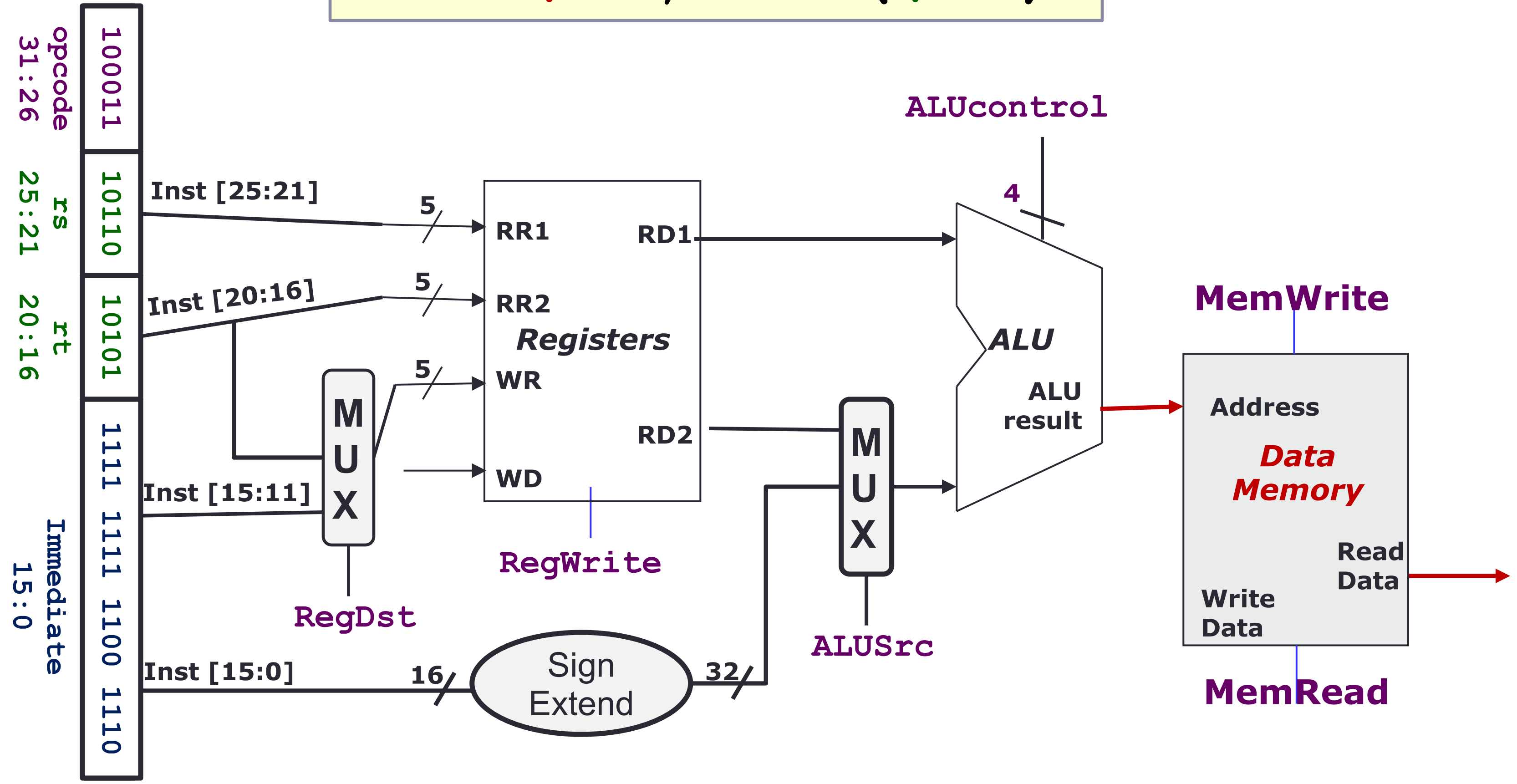
- **Inputs:**
  - Memory Address
  - Data to be written (Write Data) for store instructions
- **Control:**
  - Read and Write controls; only one can be asserted at any point of time
- **Output:**
  - Data read from memory (Read Data) for load instructions



Memory Stage: **Load Instructions**

- Only relevant parts of Decode & ALU Stages are shown

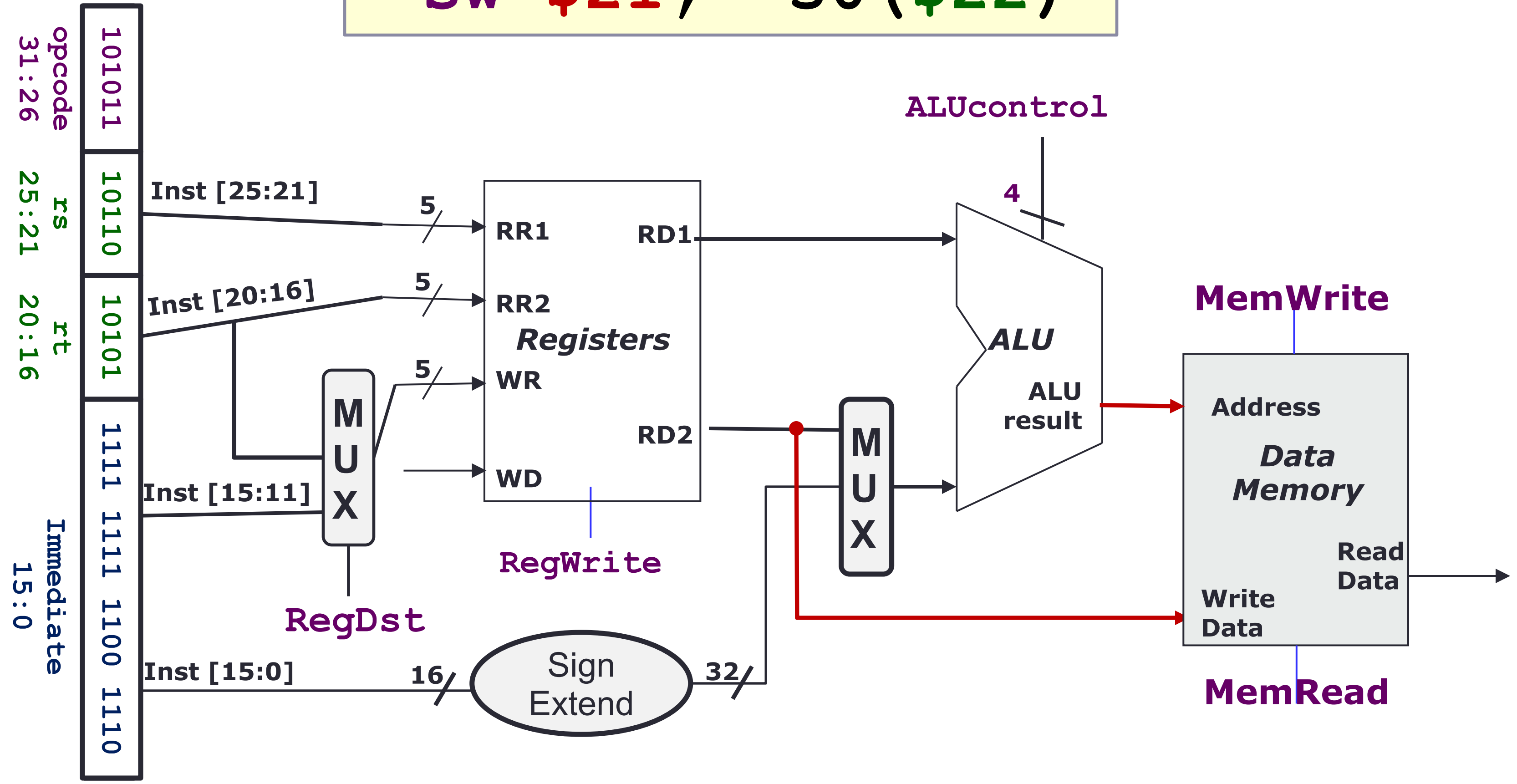
```
lw $21, -50($22)
```



Memory Stage: **Store Instructions**

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

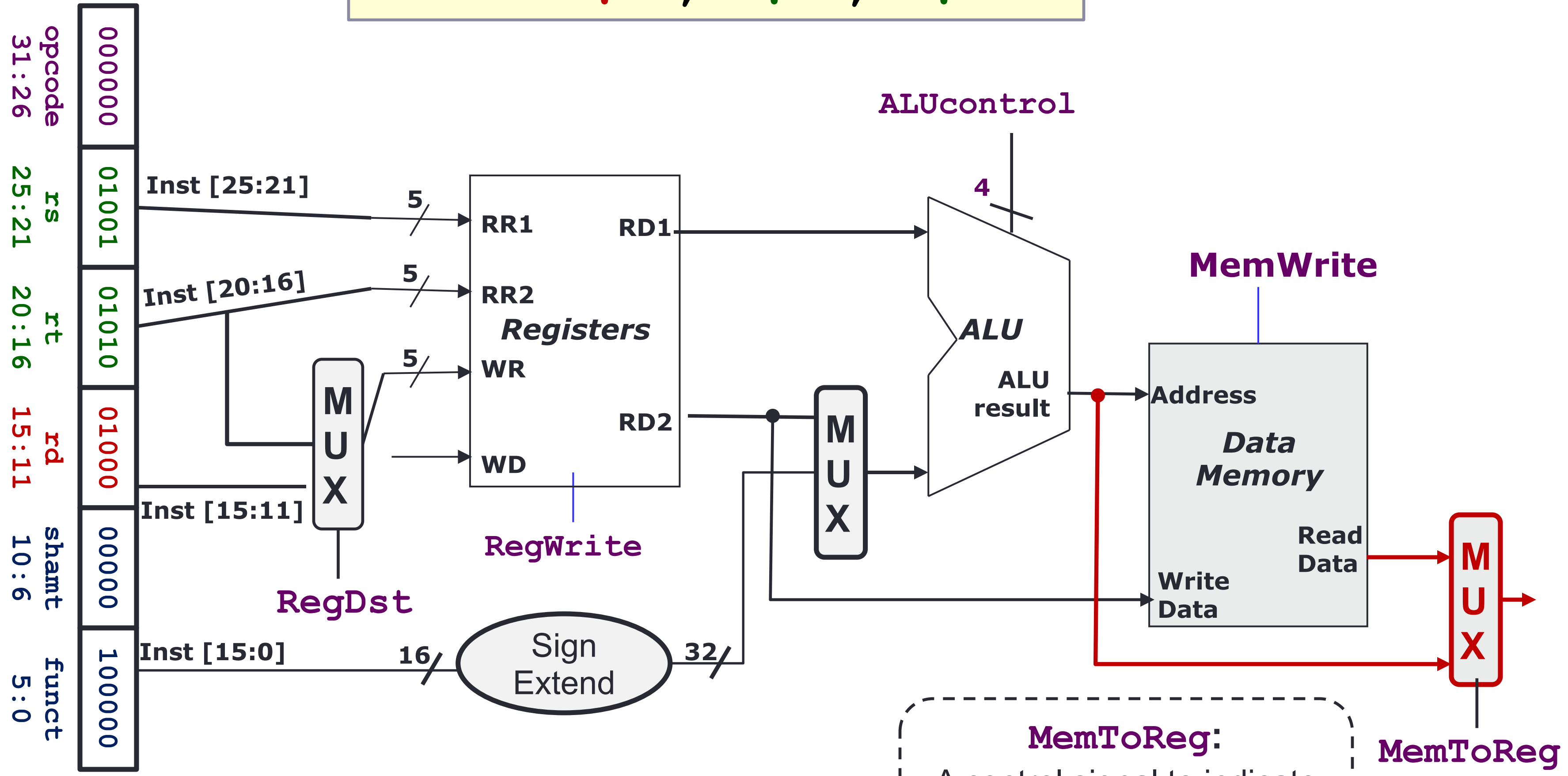
```
sw $21, -50($22)
```



Memory Stage: **Non-Memory Instructions**

- Add a multiplexer to choose the result to be stored

```
add $8, $9, $10
```



MemToReg: A control signal to indicate whether result came from memory or ALU unit

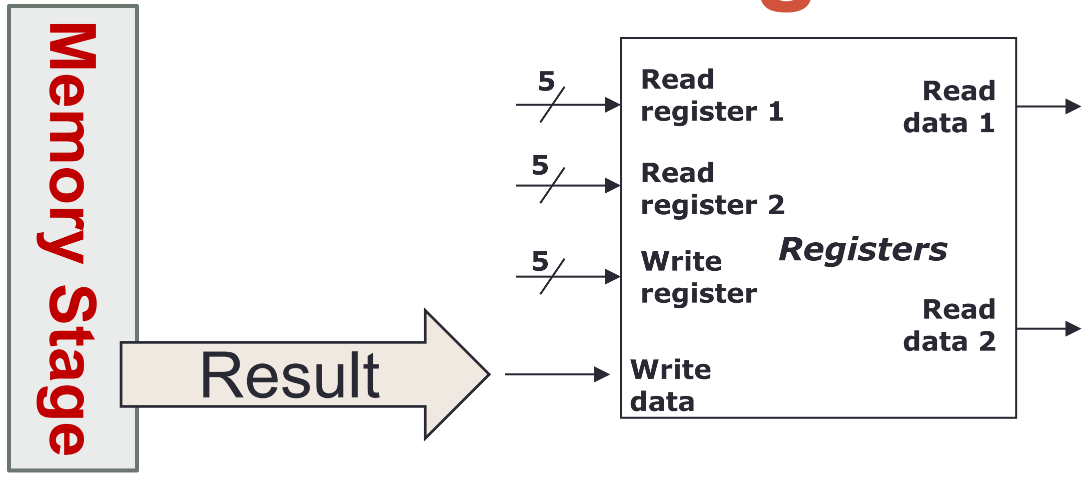
Result Write Stage: Requirement

- 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

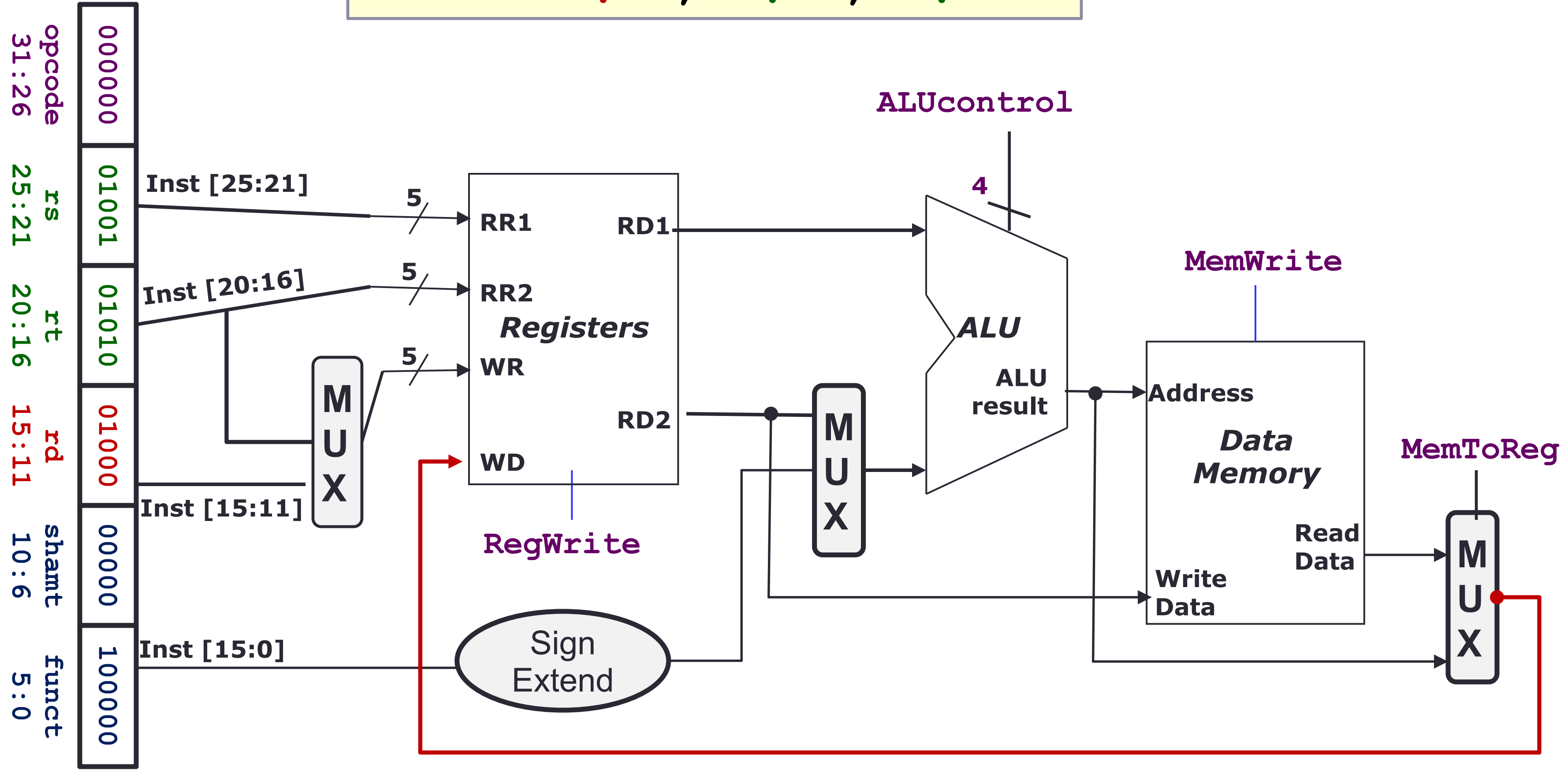
Result Write Stage: Block Diagram



- 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

Result Write Stage: Routing

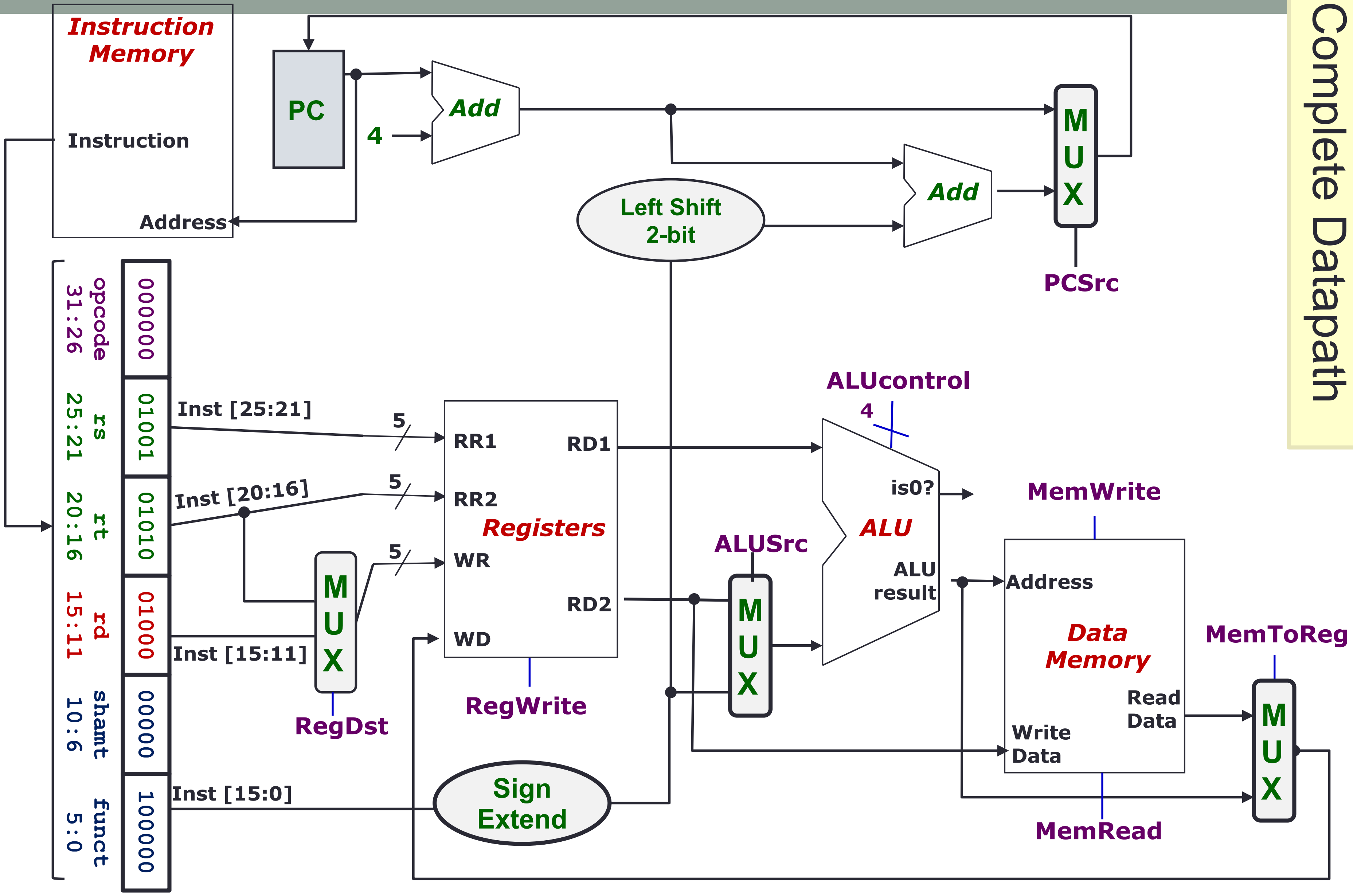
```
add $8, $9, $10
```



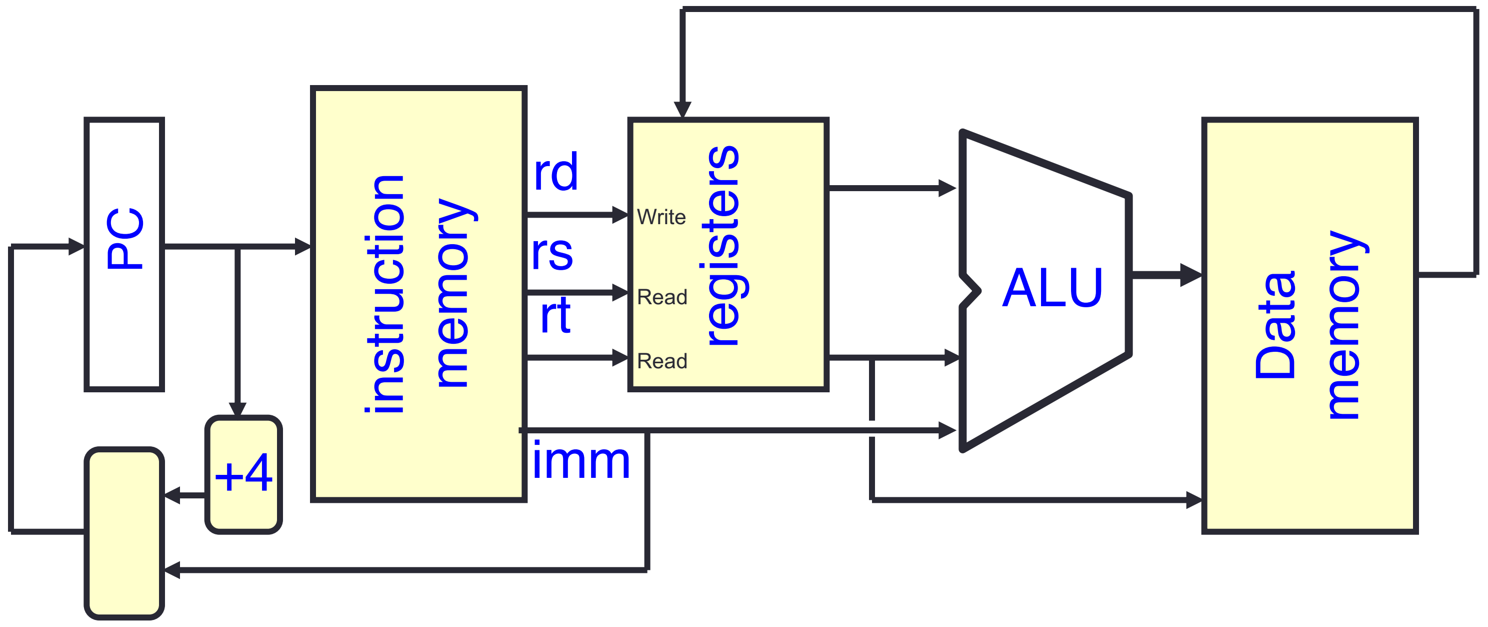
The Complete Datapath!

- We have just finished “designing” the datapath for a subset of MIPS instructions:
  - Shifting and Jumping 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** (Lecture #16)

Complete Datapath



Datapath: Generic Steps



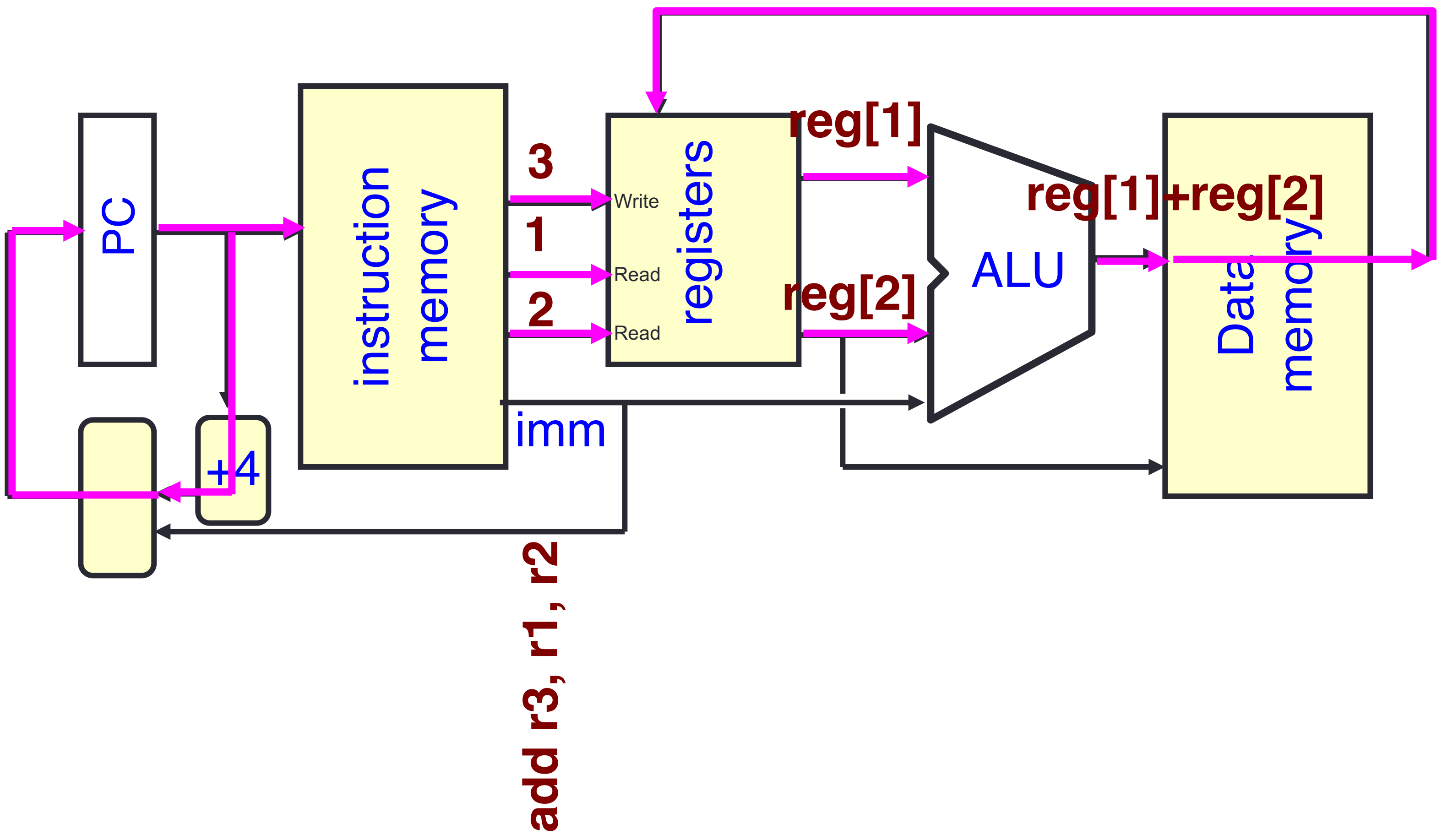
1. Instruction Fetch
2. Decode/ Register Read
3. ALU
4. Memory
5. Write Back

Datapath Walkthroughs: ADD (1/2)

- `add $r3,$r1,$r2 # r3 = r1+r2`
  - Stage 1: Fetch this instruction, increment PC.
  - Stage 2: Decode to find that it is an `add` instruction, then read registers `$r1` and `$r2`.
  - Stage 3: Add the two values retrieved in stage 2.
  - Stage 4: Idle (nothing to write to memory).
  - Stage 5: Write result of stage 3 into register `$r3`.

Datapath Walkthroughs: ADD (2/2)

`add $r3, $r1, $r2`

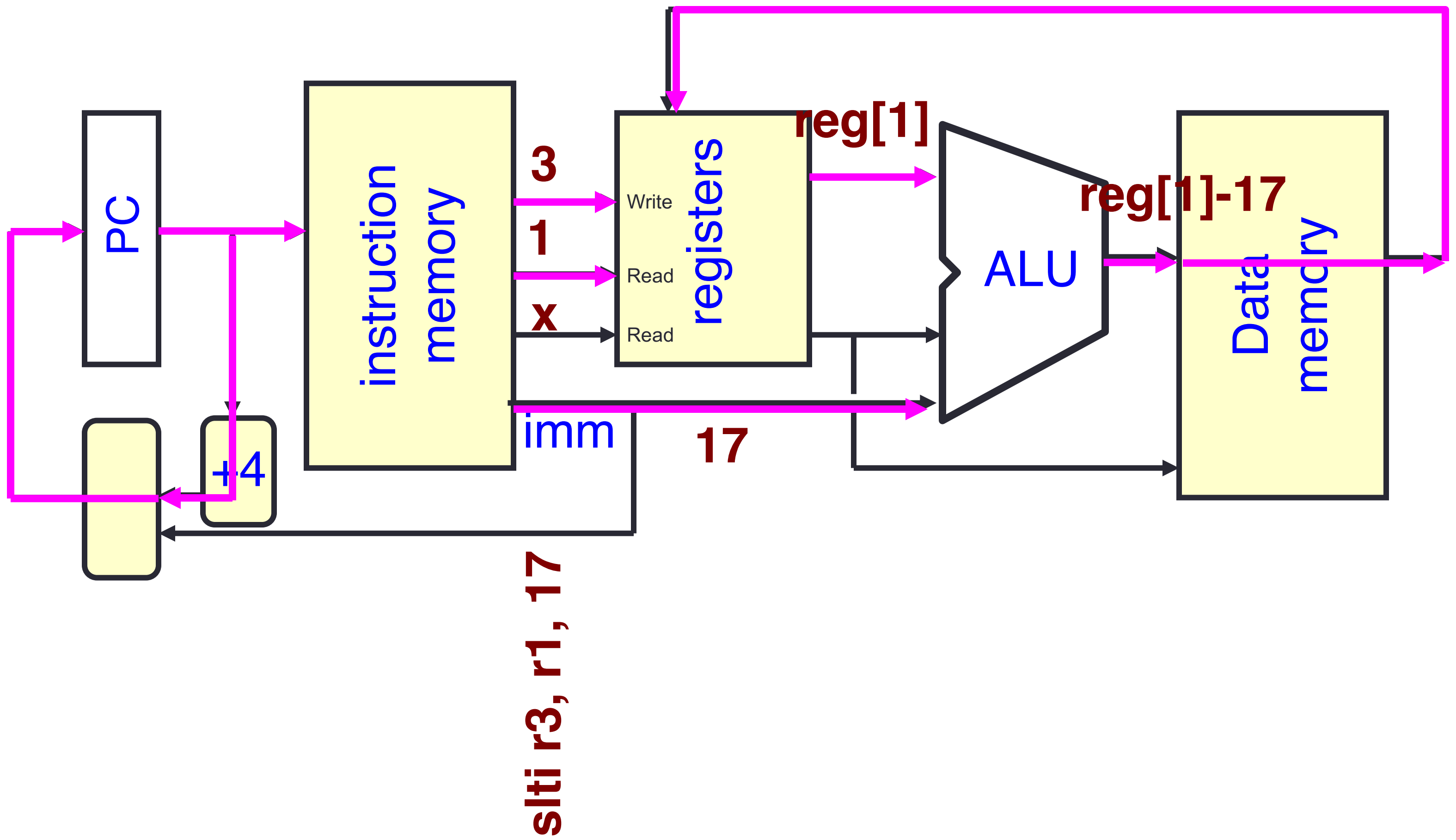


Datapath Walkthroughs: SLTI (1/2)

- **`slti $r3,$r1,17`**
  - Stage 1: Fetch this instruction, increment PC.
  - Stage 2: Decode to find it is an **`slti`**, then read register **`$r1`**.
  - Stage 3: Compare value retrieved in stage 2 with the integer 17.
  - Stage 4: Go idle.
  - Stage 5: Write the result of stage 3 in register **`$r3`**.

Datapath Walkthroughs: SLTI (2/2)

```
slti $r3, $r1, 17
```

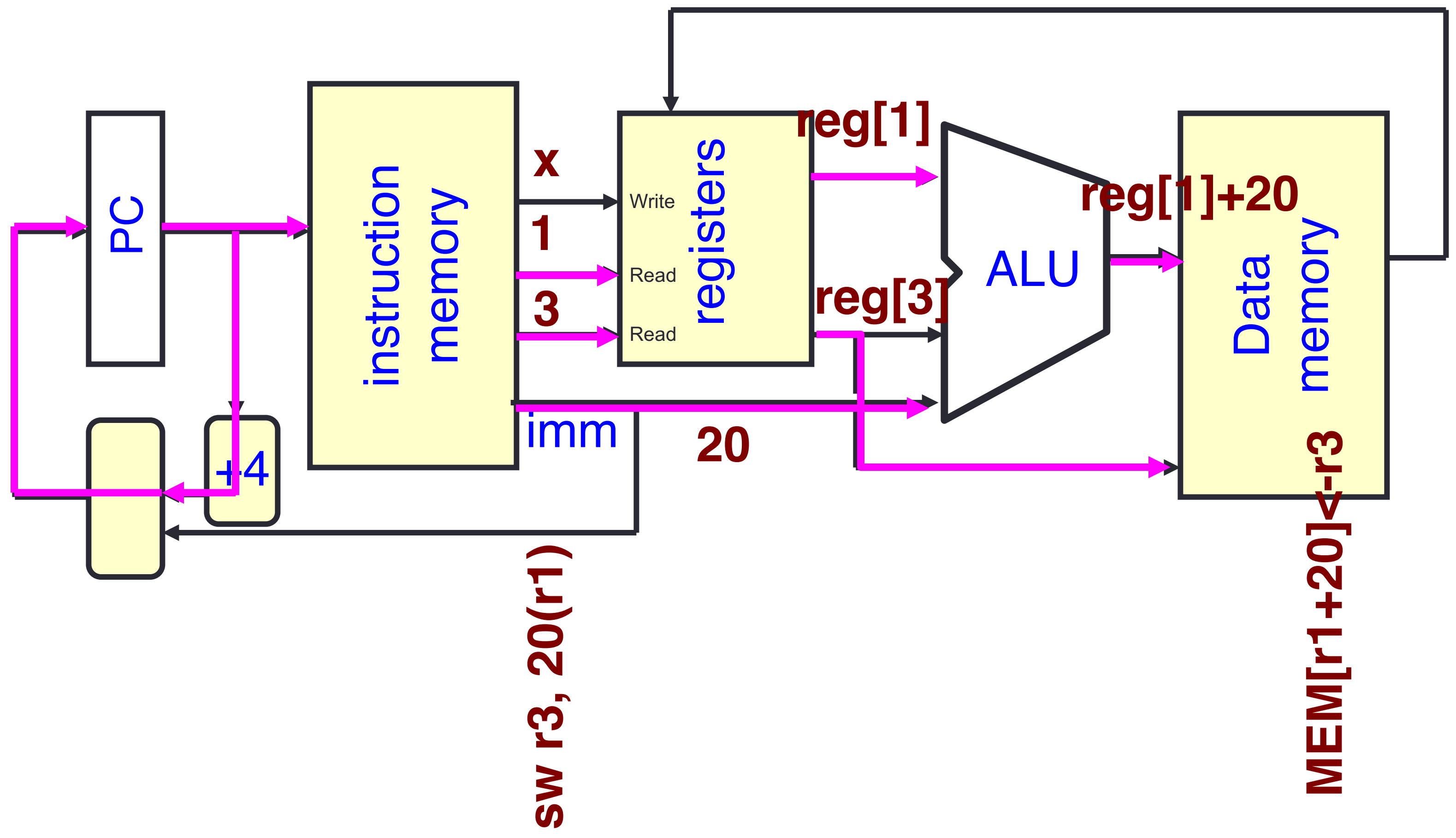


Datapath Walkthroughs: SW (1/2)

- `sw $r3, 20($r1)`
  - Stage 1: Fetch this instruction, increment PC.
  - Stage 2: Decode to find it is an `sw`, then read registers `$r1` and `$r3`.
  - Stage 3: Add 20 to value in register `$r1` (retrieved in stage 2).
  - Stage 4: Write value in register `$r3` (retrieved in stage 2) into memory address computed in stage 3.
  - Stage 5: Go idle (nothing to write into a register).

Datapath Walkthroughs: SW (2/2)

`sw $r3, 20($r1)`



Why Five Stages?

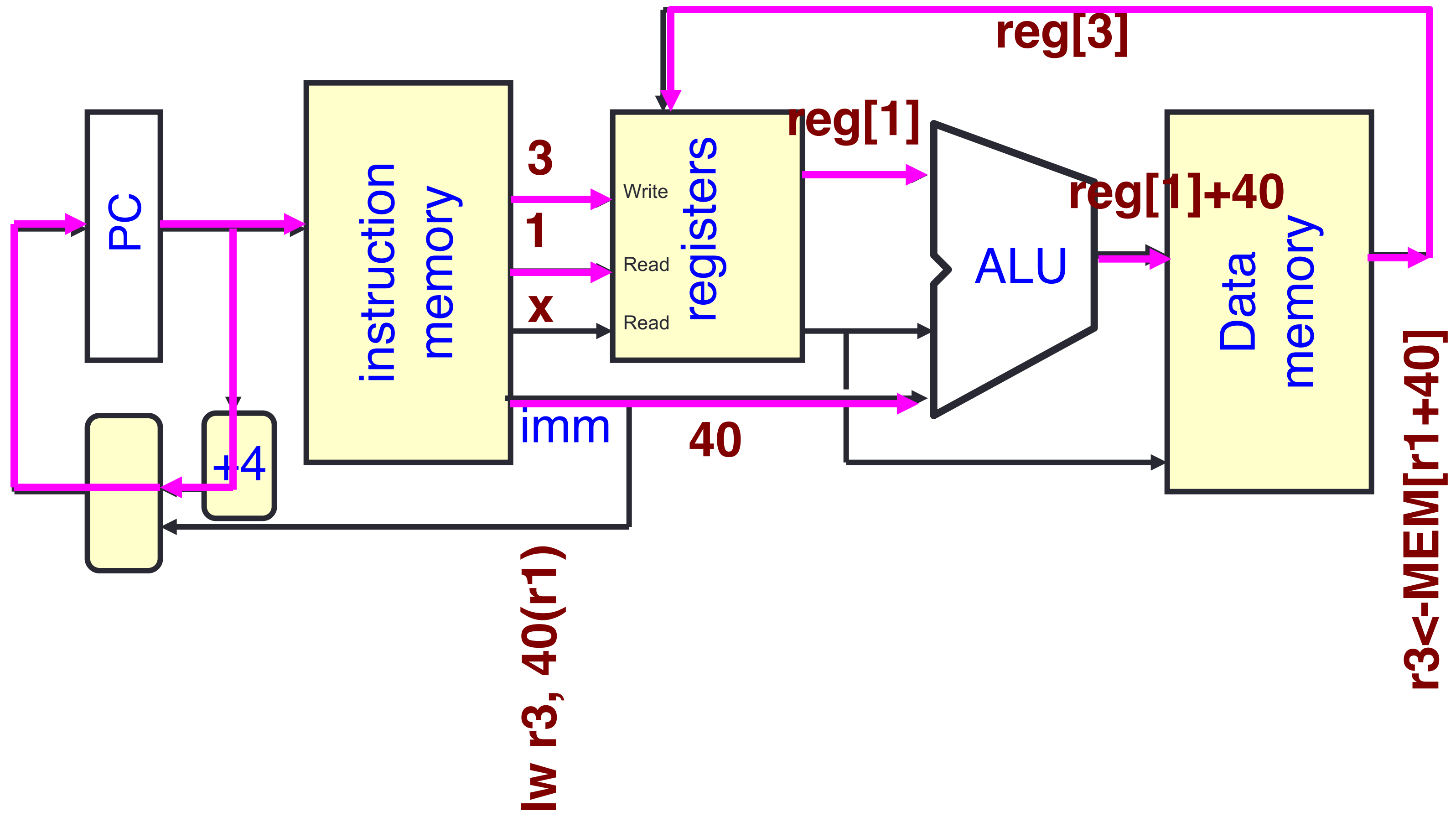
- Could we have a different number of stages?
  - Yes, and other architectures do.
- So why does MIPS have five stages, if instructions tend to go idle for at least one stage?
  - There is one instruction that uses all five stages: the load.

Datapath Walkthroughs: LW (1/2)

- **`lw $r3, 40($r1)`**
  - Stage 1: Fetch this instruction, increment PC.
  - Stage 2: Decode to find it is a **`lw`**, then read register **`$r1`**.
  - Stage 3: Add 40 to value in register **`$r1`** (retrieved in stage 2).
  - Stage 4: Read value from memory address compute in stage 3.
  - Stage 5: Write value found in stage 4 into register **`$r3`**.

Datapath Walkthroughs: LW (2/2)

```
lw $r3, 40($r1)
```

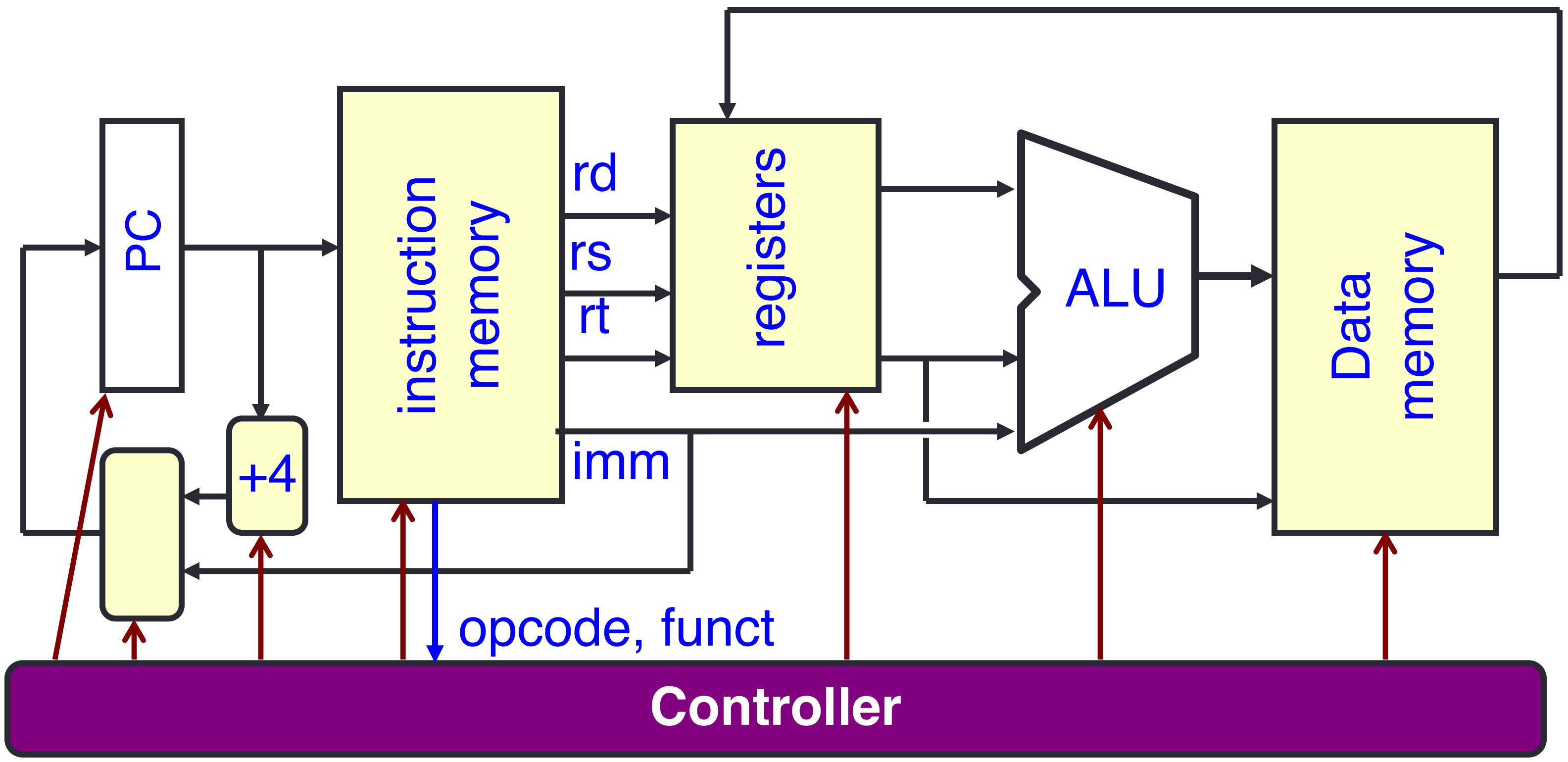


What Hardware is Needed?

- PC: a register which keeps track of address of the next instruction.
- General Purpose Registers
  - Used in stage 2 (read registers) and stage 5 (writeback).
- Memory
  - Used in stage 1 (fetch) and stage 4 (memory).
  - Cache system makes these two stages as fast as the others, on average.

Datapath: Summary

- Construct datapath based on register transfers required to perform instructions.
- Control part causes the right transfers to happen.



Reading Assignment

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

Q&A