6 - Datapath Design

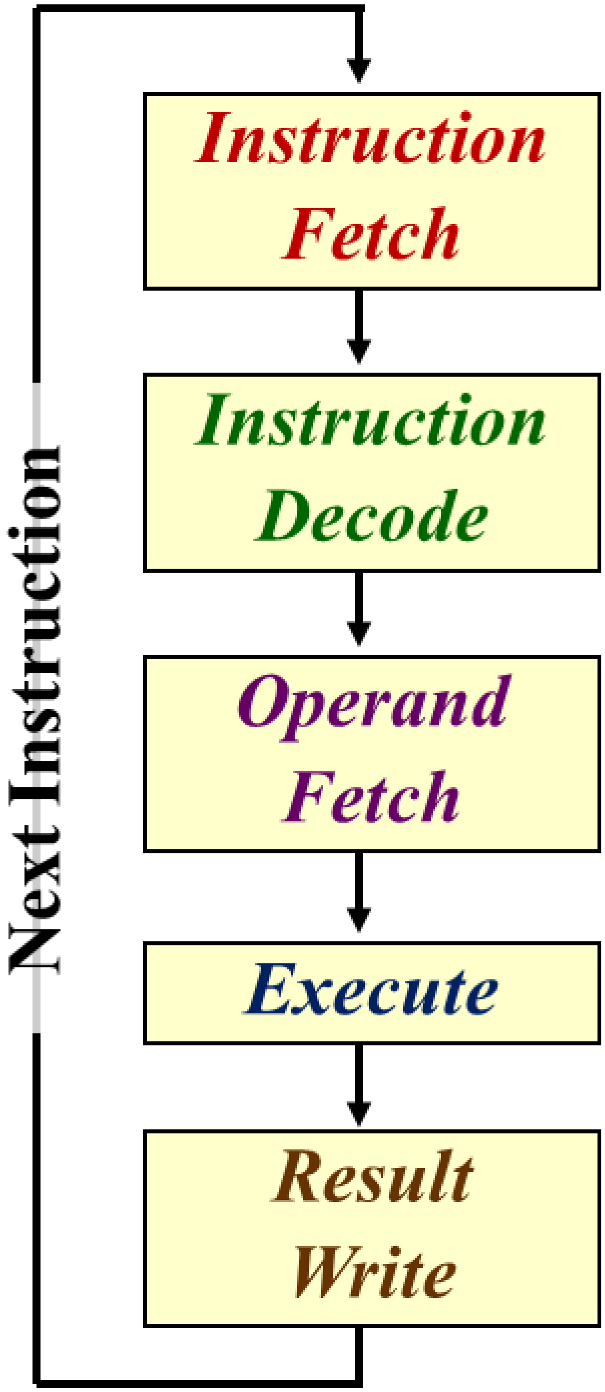
6.1 Building a Processor: Datapath & Control

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

6.2 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

6.3 Instruction Execution Cycle

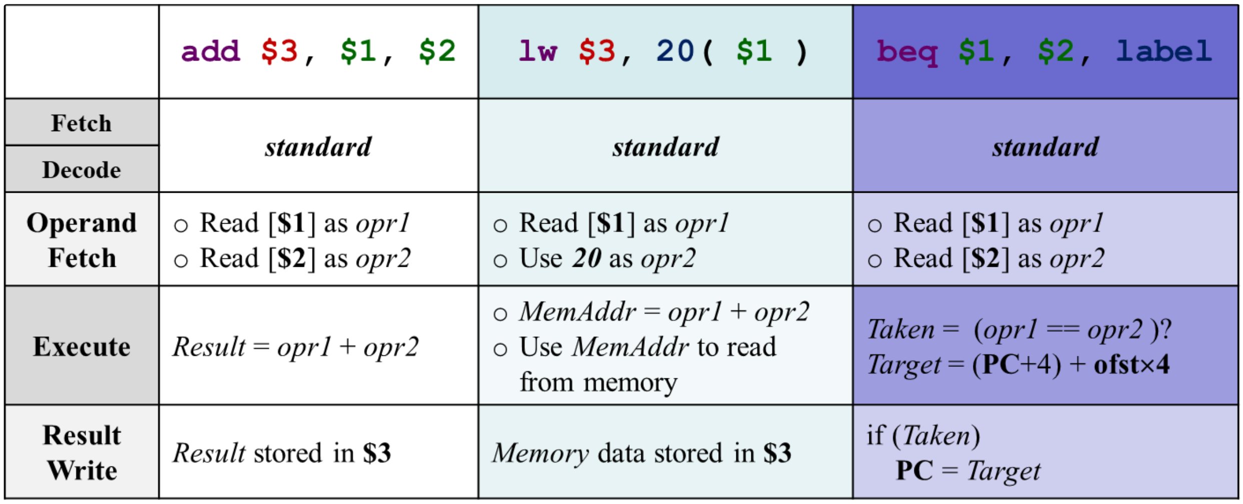


1. Fetch:
   - Get instruction from memory
   - Address is in Program Counter (PC) Register
2. Decode:
   - Find out the operation required
3. Operand Fetch
   - Get operand(s) needed for operation
4. Execute:
   - Perform the required operation

5. Result Write (Store):
   - Store the result of the operation

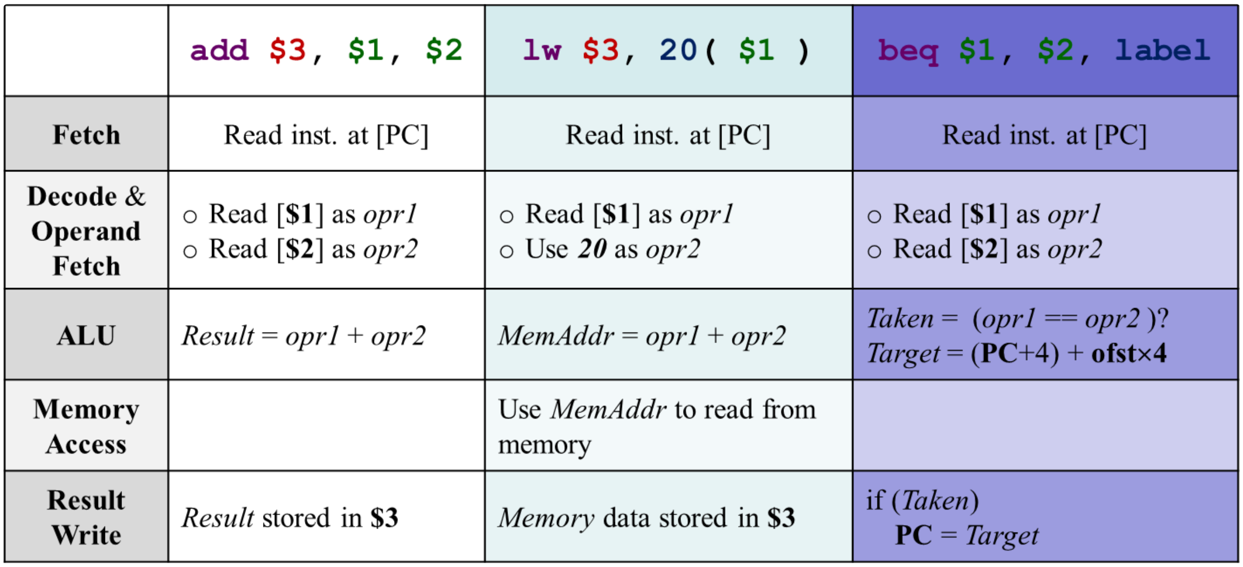
6.4 MIPS Instruction Execution

- Show the actual steps for 3 representative MIPS instructions
- Fetch and Decode stage not shown, the standard steps are performed



	add $3, $1, $2	lw $3, 20($1)	beq $1, $2, label
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 from memory	*Taken* = (*opr1* == *opr2*)? *Target* = (**PC**+4) + **ofst×4**
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

- `opr` = operand, `ofst` = offset, `MemAddr` = Memory Address
- 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, label
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) + **ofst×4**
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

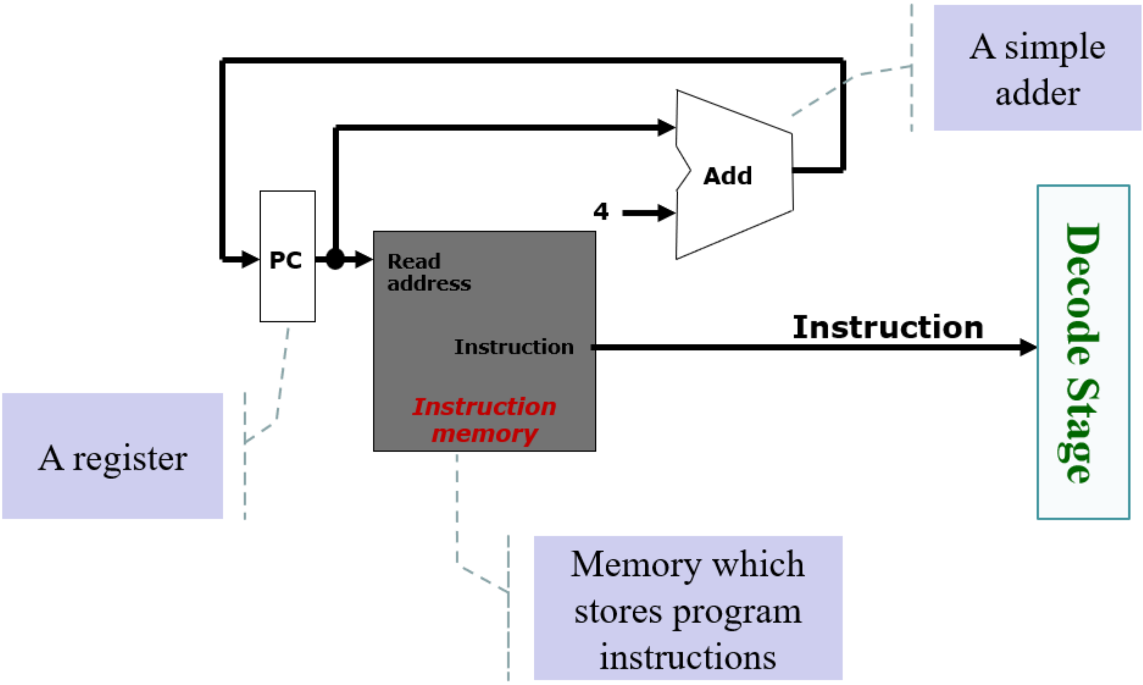
- `inst.` = instruction

6.5 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

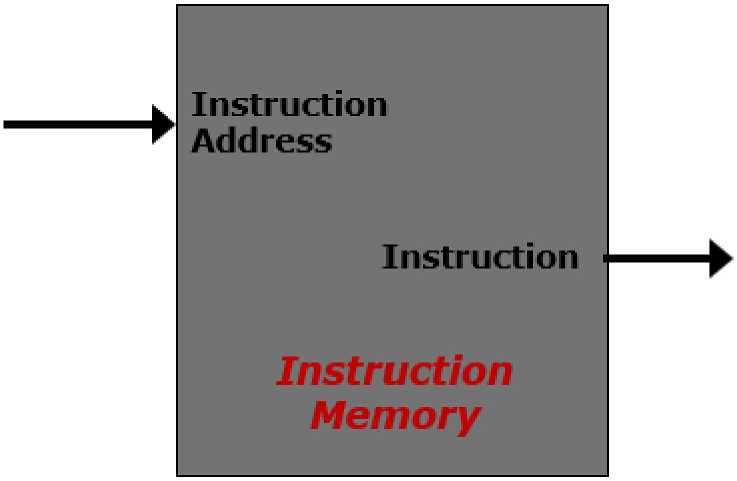
6.5.1 Fetch Stage

- 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:
     - Since each instruction is 32 bit, which is 4 bytes
     - Note the exception when branch/jump instruction is executed
- Output to the next stage (**Decode**)”
  - The instruction to be executed

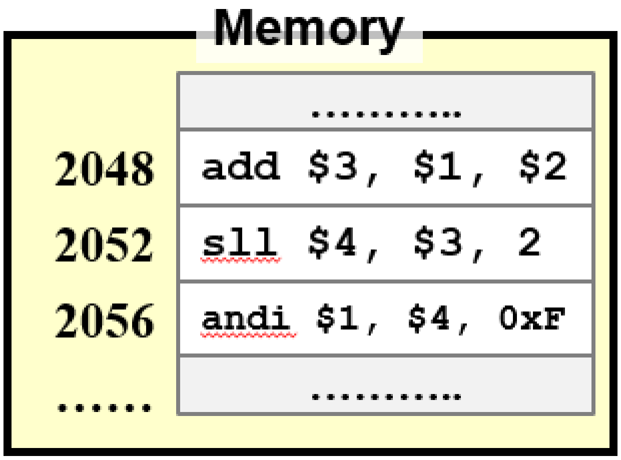


Element: Instruction Memory

- Storage element for the instructions
  - It is a sequential circuit
  - 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 below



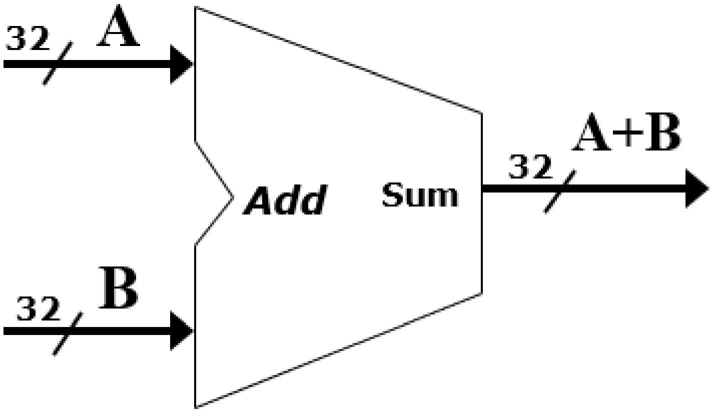
Memory

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

> 1. **存储元素**：这里提到的存储元素是指用于存储指令的硬件部件。在计算机中，这通常是指主存储器或RAM。
> 2. **它是一个顺序电路**：与组合电路不同，顺序电路有一定的内部状态，并且其输出不仅取决于当前的输入，还取决于之前的输入或状态。这意味着存储元素可以维持和存储信息。
> 3. **有内部状态用于存储信息**：这意味着该电路具有某种记忆能力，能够保存数据或状态信息，直到被更改或清除。
> 4. **时钟信号是假设的并且未显示**：顺序电路通常需要一个时钟信号来同步其操作。但是，在某些描述或图解中，为了简化表示，时钟信号可能不会明确显示。
> 5. **提供指令给定地址**：存储元素的主要功能之一是，当提供一个指令地址 `M` 时，它能够输出存储在地址 `M` 的内容。这就是计算机从内存中获取指令并执行它们的方式。

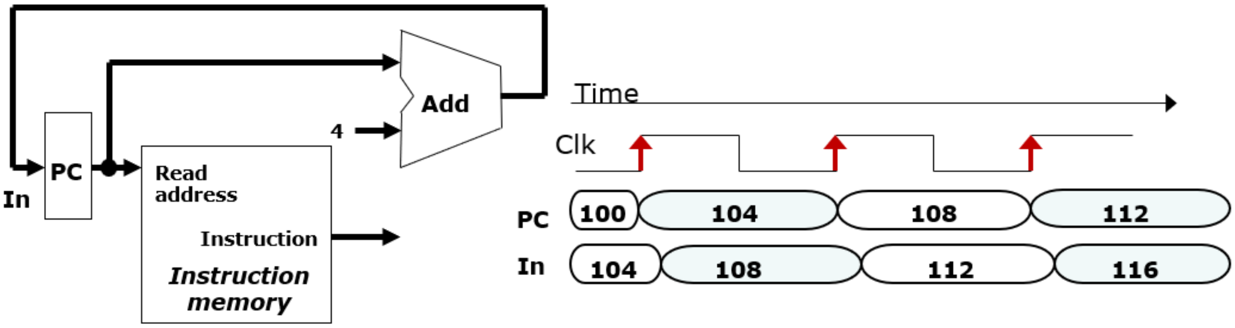
Element: Adder

- Combinational logic to implement the addition of two numbers
- Inputs:
  - Two 32-bit numbers A,B
- Output:
  - Sum of the input number A+B



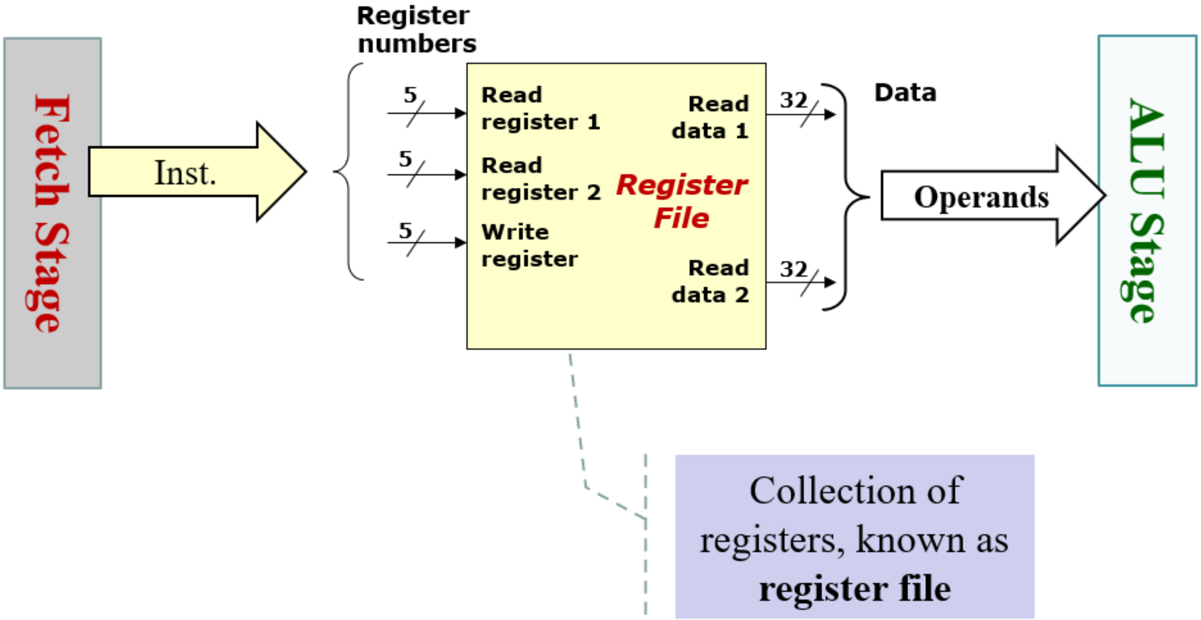
The idea of clocking

- PC is read during the first half of the clock period and it is updated with PC+4 at the next rising clock edge



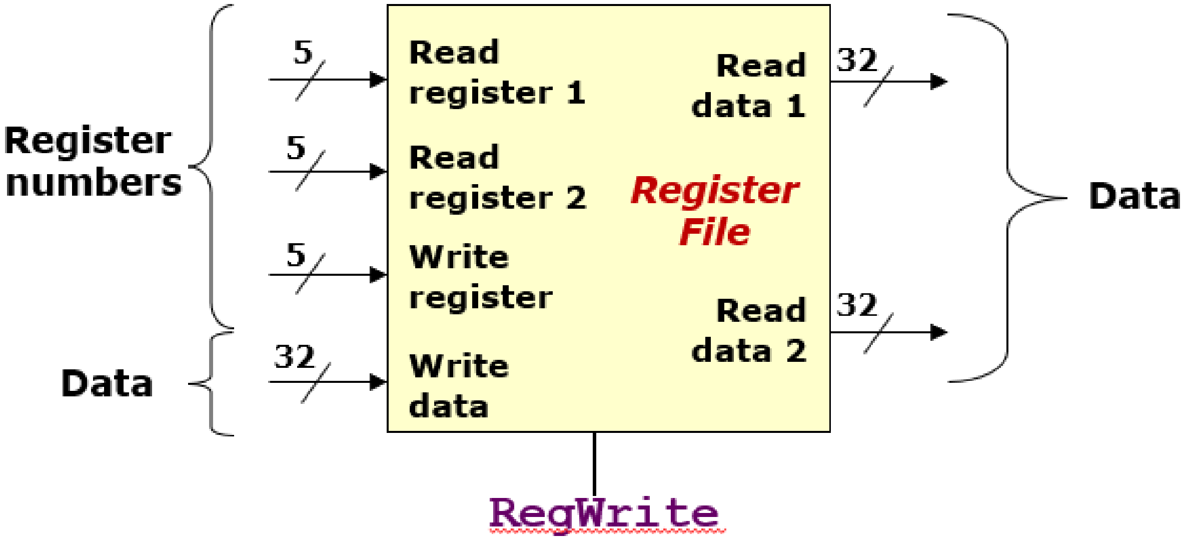
6.5.2 Decode Stage

- 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
- Input from previous stage (Fetch):
  - Instruction to be executed
- Output to the next stage (ALU):
  - Operation and the necessary operands

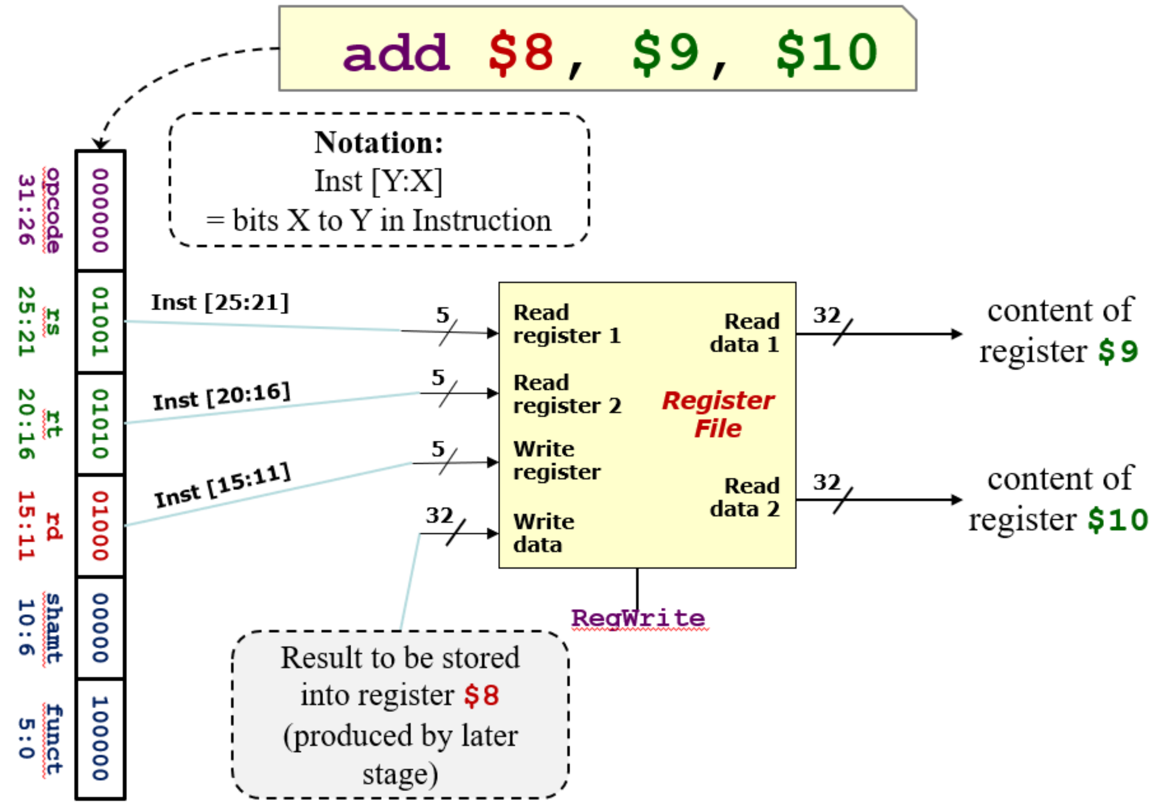


Element: Register File

- A collection of 32 registers
  - Each 32-bit wide; can be read/written by specifying register number
  - Read at most two register 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



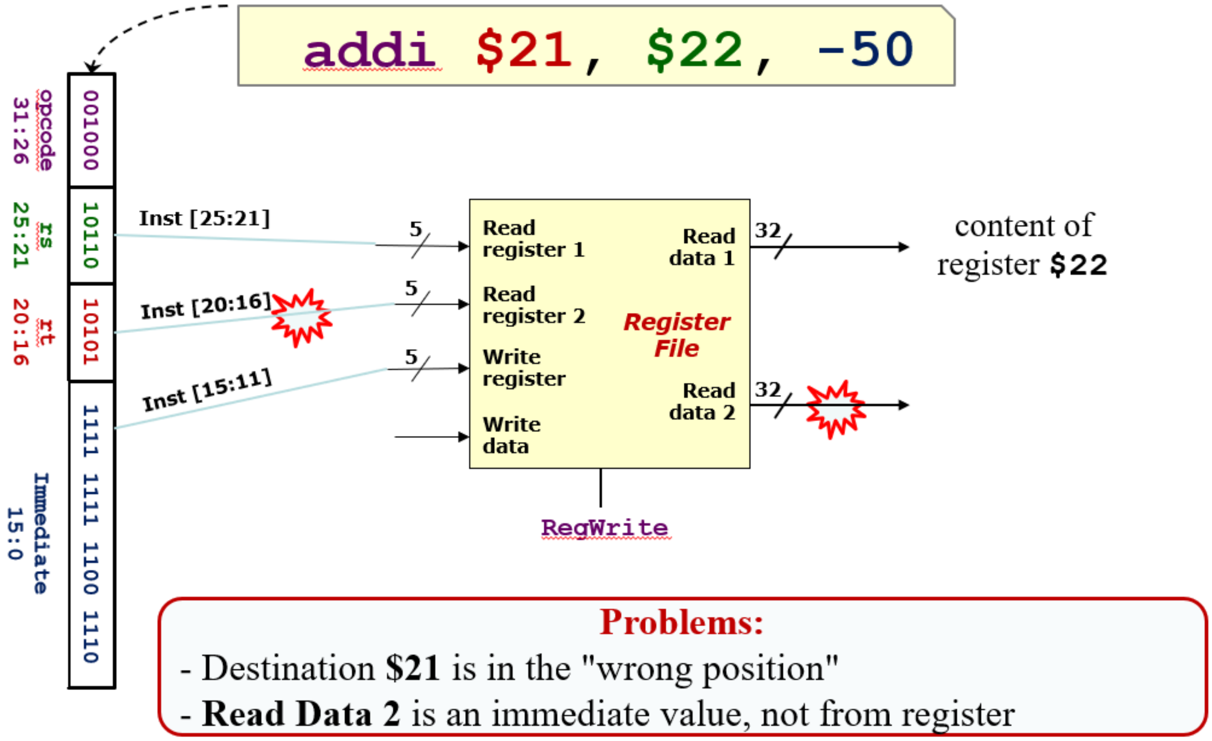
R-Format Instruction



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

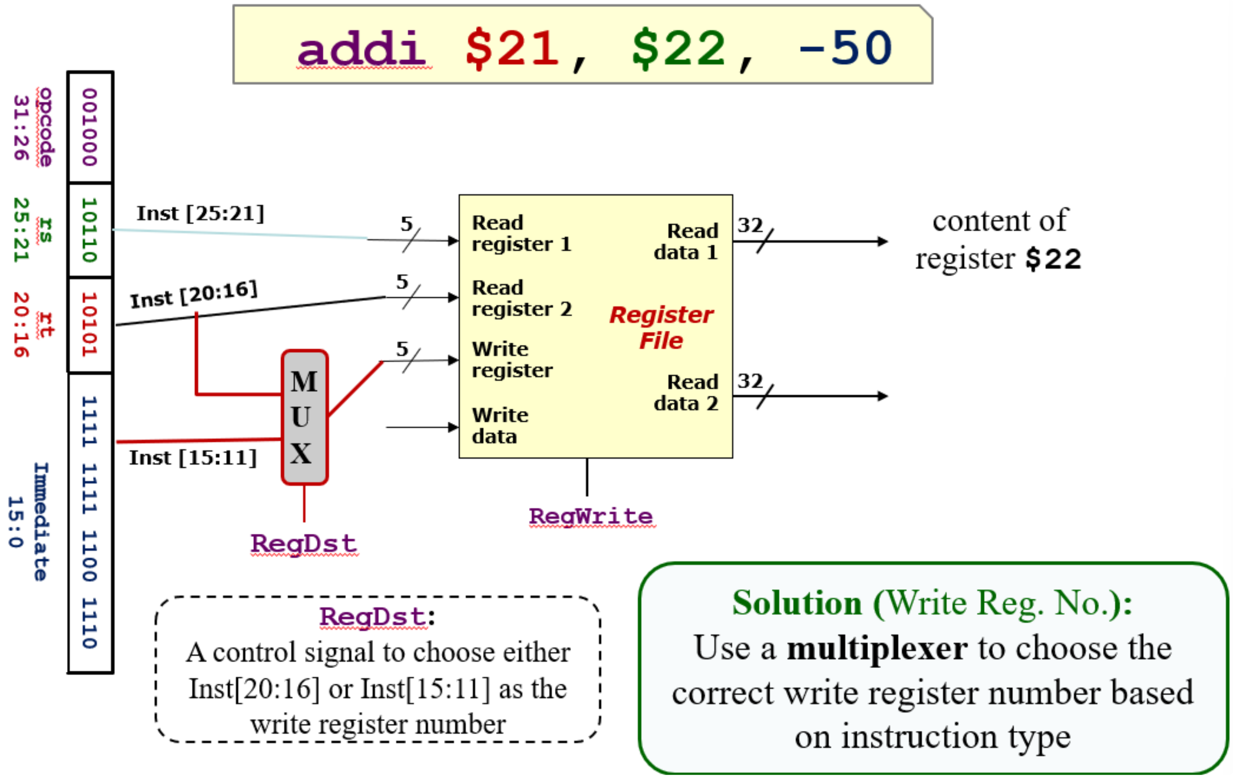
Result to be stored into register $8 (produced by later stage)

I-Format Instruction



Problems:

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



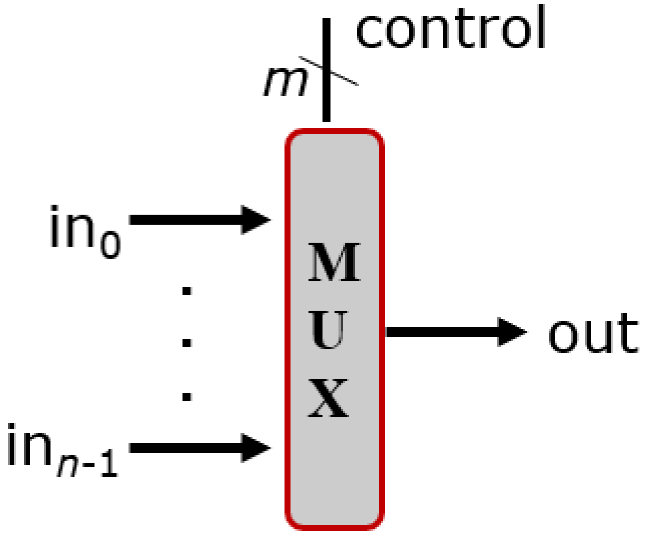
RegDst:
A control signal to choose either Inst[20:16] or Inst[15:11] as the write register number

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

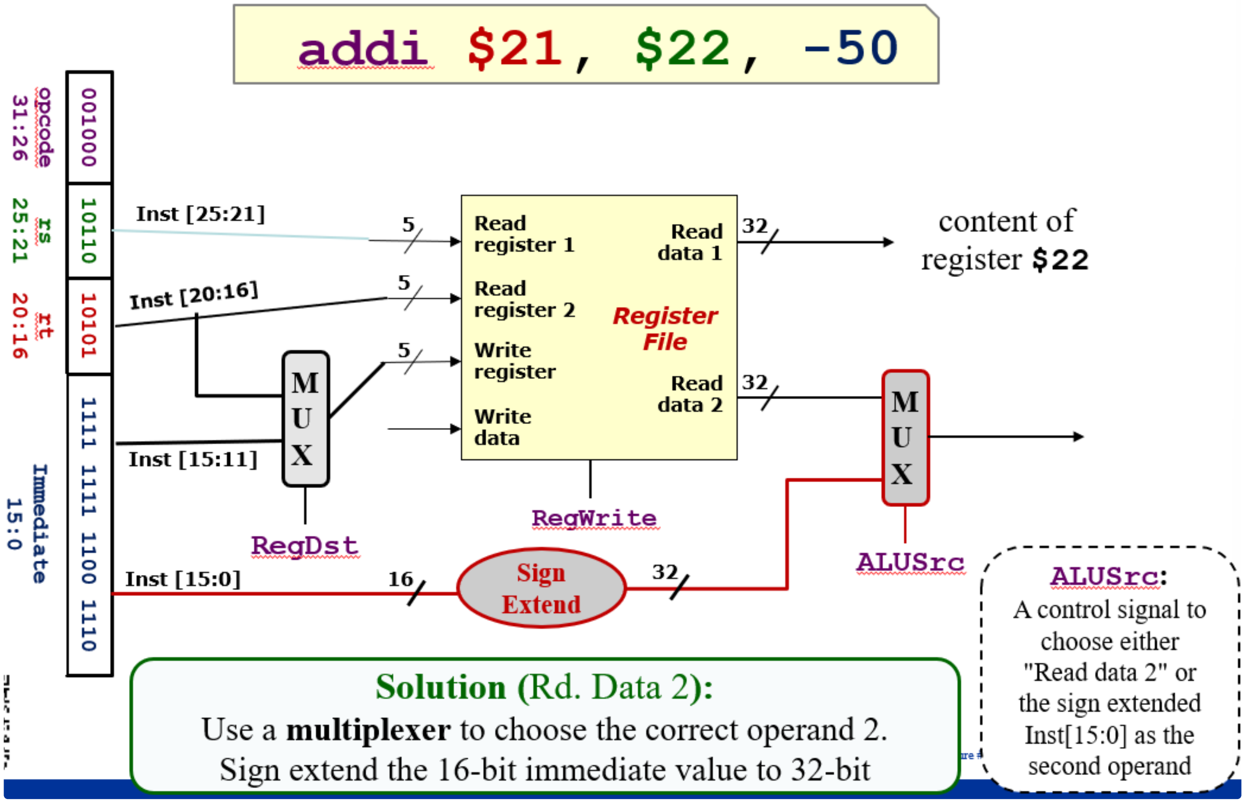
> 1. **不同的指令格式**：MIPS指令集包括R-format、I-format和J-format等不同格式的指令。这些指令格式中，寄存器的编号和位置可能会不同。
> 2. **写入寄存器的选择**：在R-format指令中，通常使用 `rd` 字段（也就是 `Inst[15:11]` ）来指定结果的写入寄存器。但是，在I-format指令（如 `addi` ）中，结果是写入 `rt` 字段指定的寄存器，也就是 `Inst[20:16]` 。
> 3. **multiplexer的作用**：为了能够处理这种不同，我们需要一个选择器，即 `multiplexer` 。根据指令的类型（R-format还是I-format）， `multiplexer` 决定应该使用 `Inst[20:16]` 还是 `Inst[15:11]` 来确定写入寄存器的编号。
> 4. **控制信号 `RegDst`** ：这是一个控制信号，用于告诉 `multiplexer` 该选择哪个输入。例如，对于I-format指令， `RegDst` 会设置为选择 `Inst[20:16]` ；对于R-format指令，它会设置为选择 `Inst[15:11]` 。

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$_0$ to out
Control=3 → select in$_3$ to out



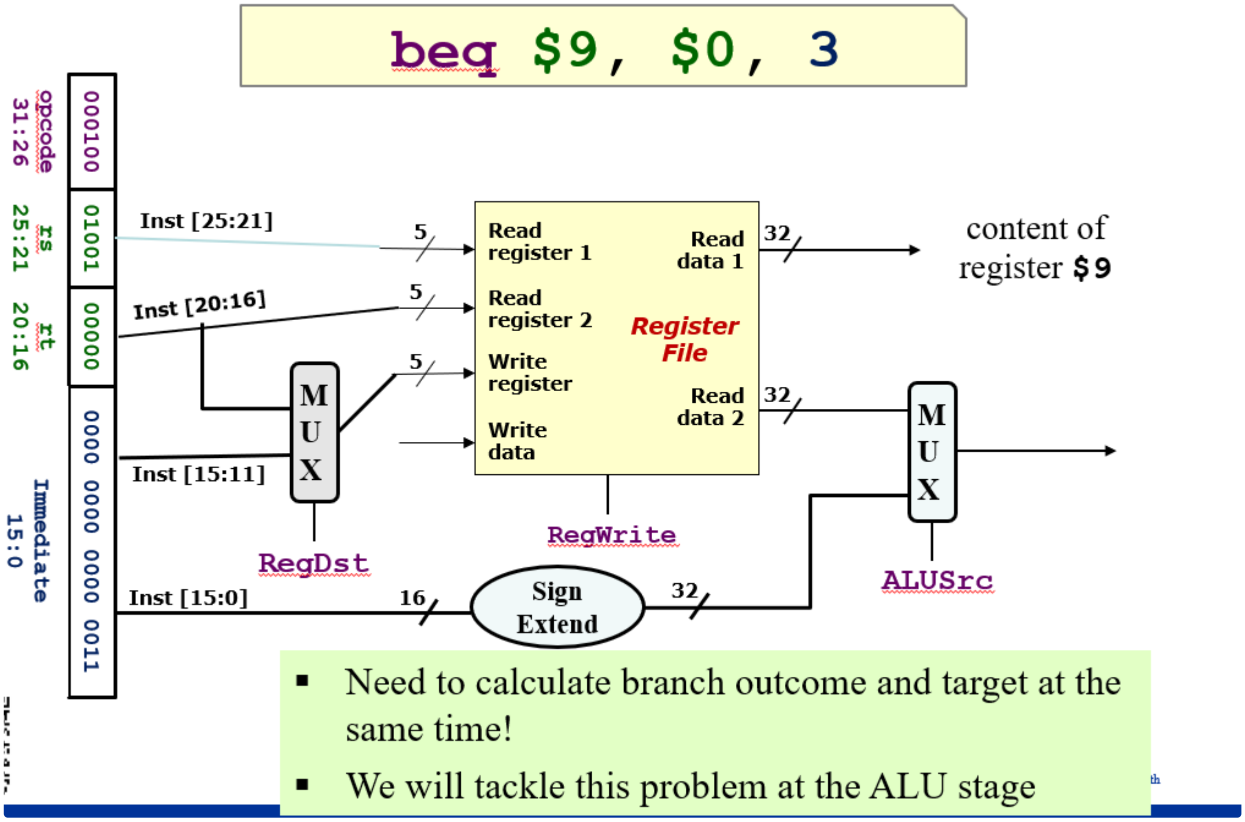
Solution (Rd. Data 2):
Use a **multiplexer** to choose the correct operand 2.
Sign extend the 16-bit immediate value to 32-bit

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

> 这张图中显示了完整的decoder处理I-format指令的流程。
>
> 首先在左侧的输入部分，由于I format指令少一个寄存器的输入，所以先加入一个MUX将rt部分和原先属于rd部分的寄存器输入，根据指令的类型选择输出一个到decoder中。最后，对immediate的后半部分（16位）加长到32位后连到data read 2后面的MUX中。
>
> sign extend操作是这样的：首先检查immediate的最高位（第15位，也称为符号位）。如果符号位是0（即该值是正数或零），那么在32位值的前16位都填充0。如果符号位是1（即该值是负数），那么在32位值的前16位都填充1。

> 由于MIPS的I format指令中的immediate字段一开始就被设计成16位，所以在decode阶段中截断成16位再延长至32位的操作不会导致数据的损失
>
> 在右半部分，如果指令是I format的，那么左侧的MUX会输入rt，右侧的MUX会选择immediate。最终decode阶段输出的是一个register值和一个immediate值

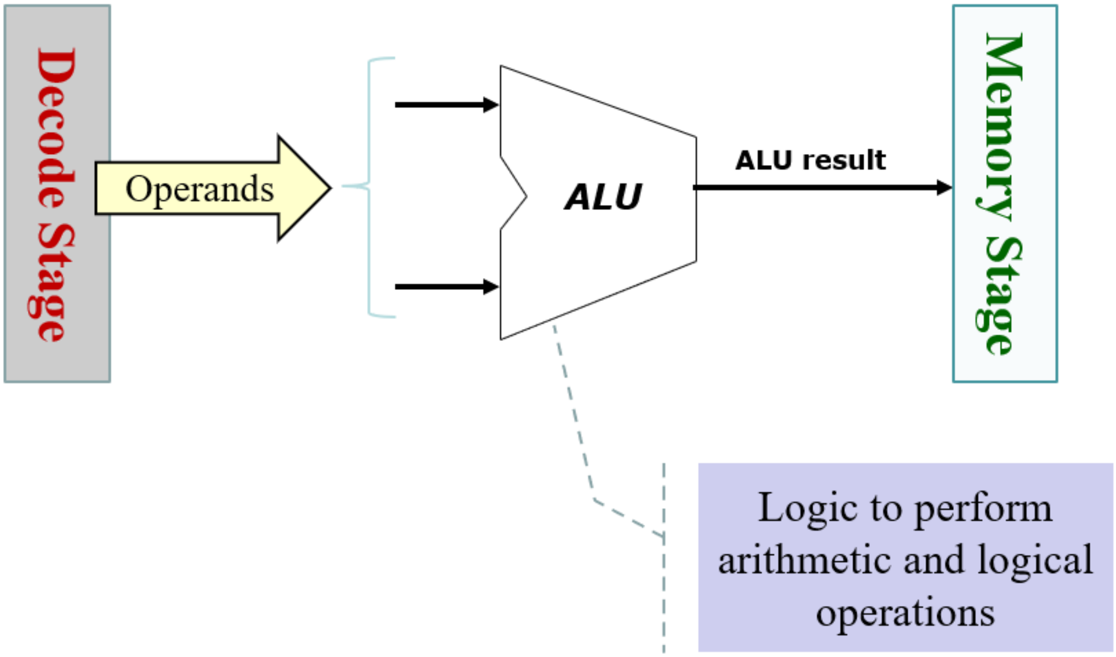
Branch Instruction


beq $9, $0, 3

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

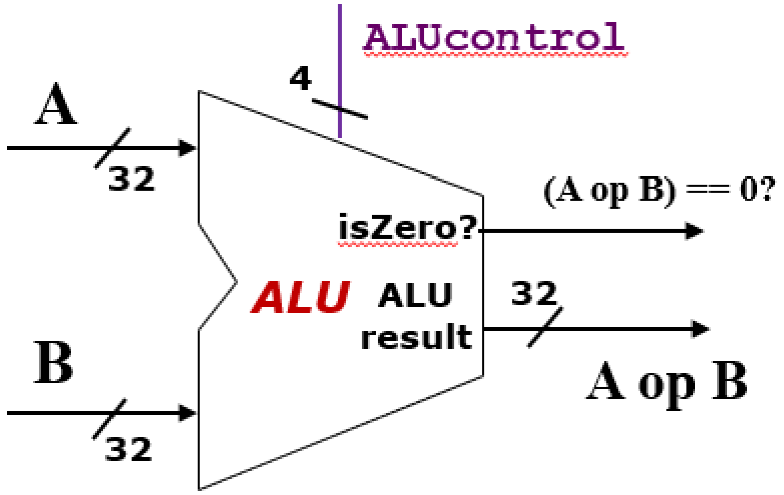
6.5.3 ALU Stage

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



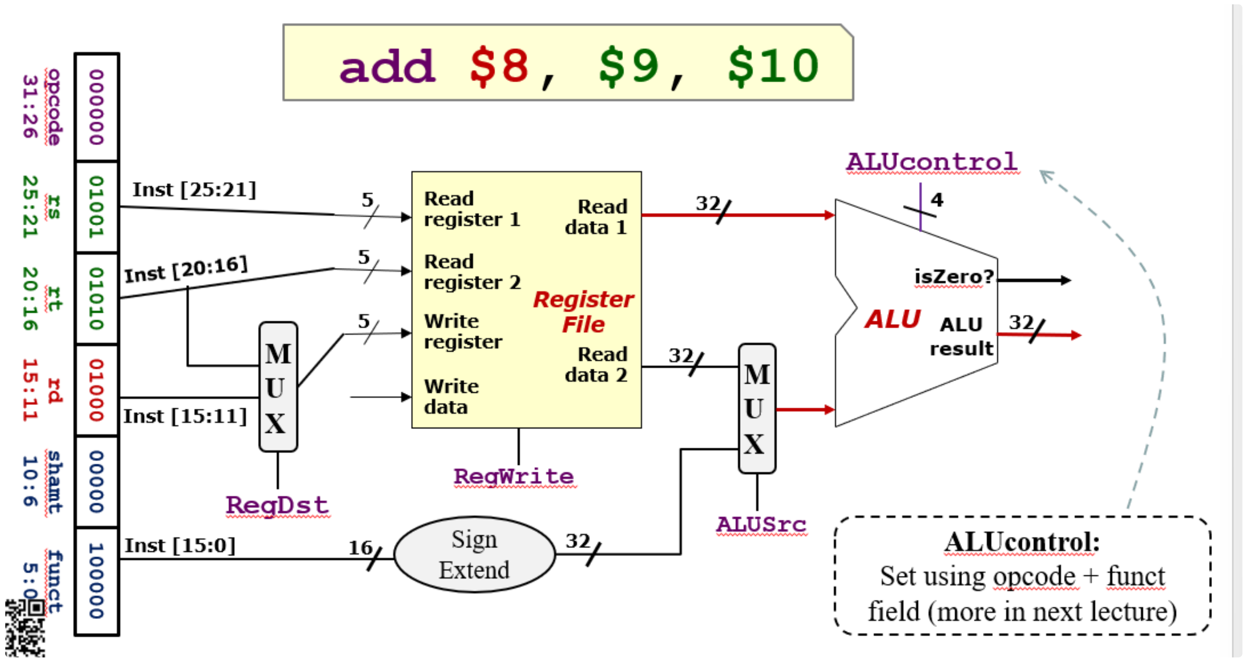
Element: Arithmetic Logic Unit

- ALU is a 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, and a 1-bit signal to indicate whether the result is zero



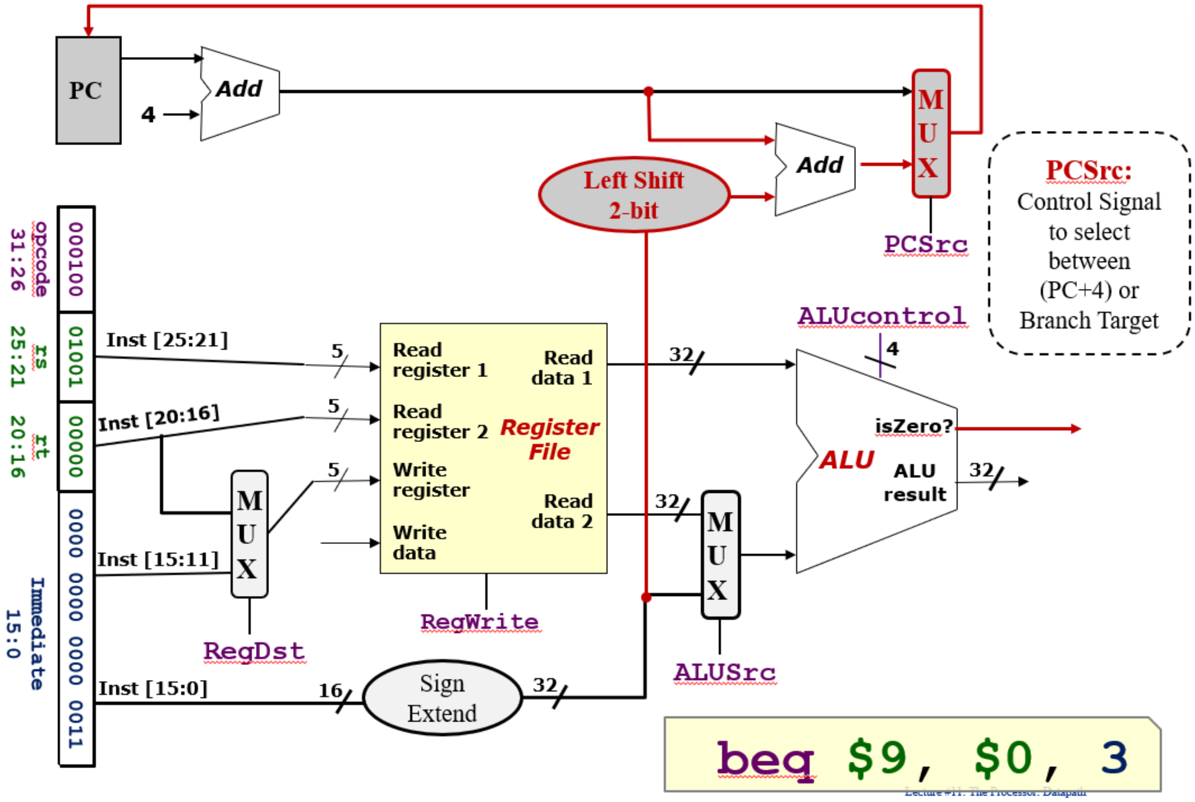
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Decode & ALU Stage: Non-Branch Instructions



ALU Stage: Branch Instructions

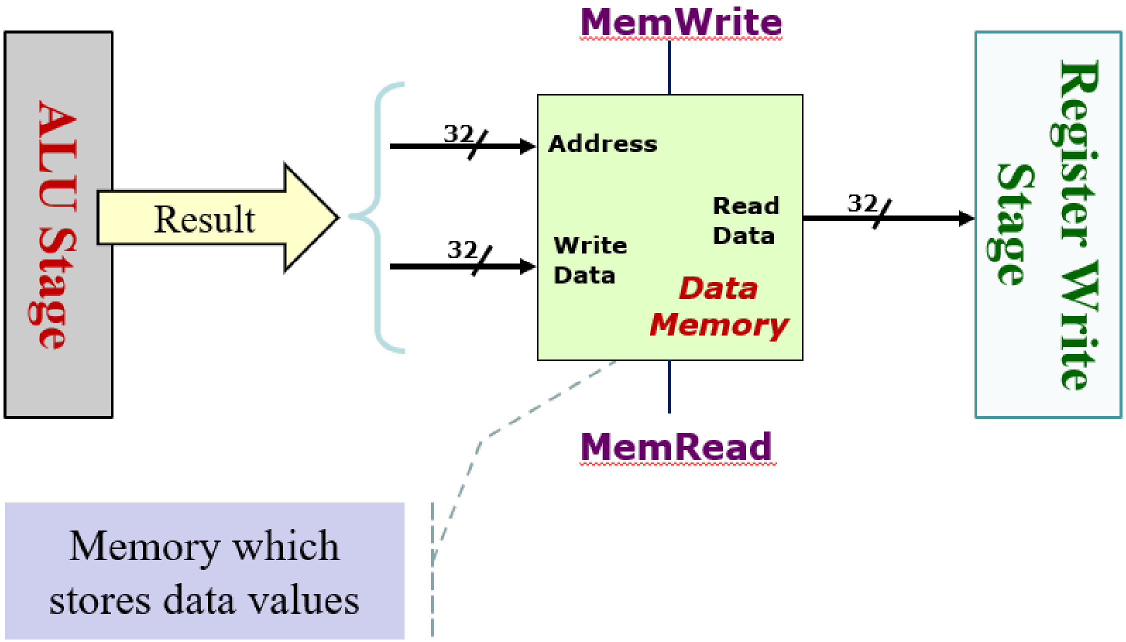
- Branch Instruction is harder as we need to perform two calculations, for example `beq $9, $0, 3`
  1. Branch Outcome
     - Use ALU to compare the register
     - The 1-bit `isZero` signal is enough to handle equal/not equal-check.
  2. Branch Target Address:
     - Introduce additional logic to calculate the address
     - Need PC (from Fetch Stage)
     - Need Offset (from Decode Stage)



6.5.4 Memory Stage

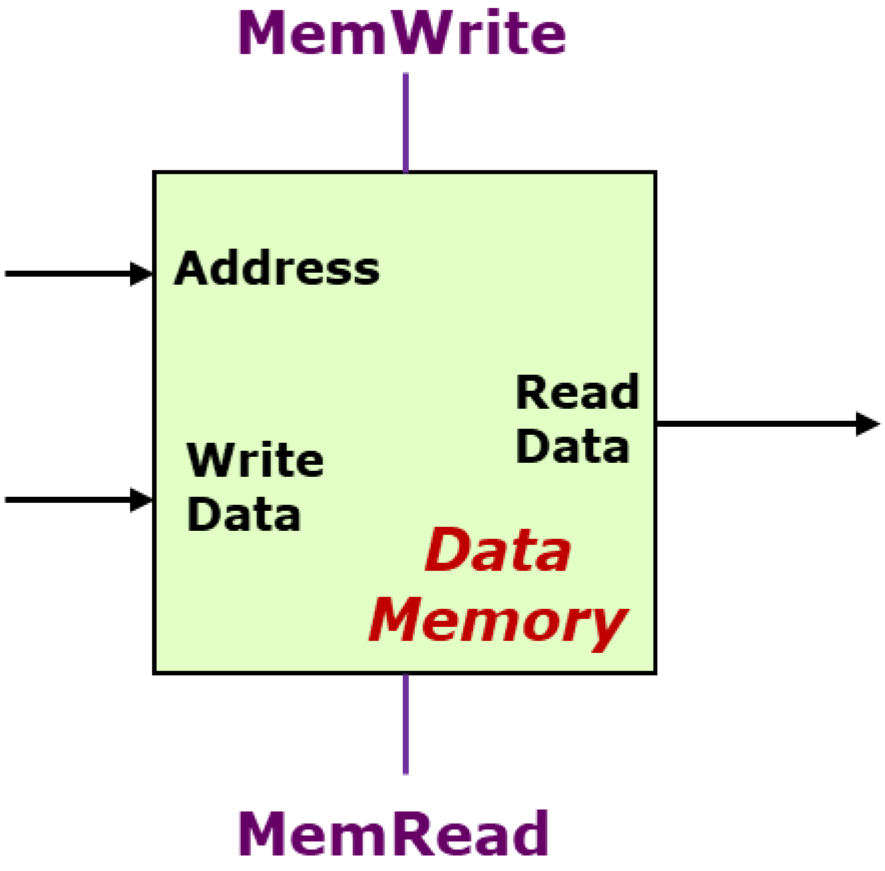
- Instruction Memory Access Stage:
  - Only the load and store instructions need to perform operation in this stage:
    - Use memory address calculated by ALU stage
    - Read from or write to data memory
  - All other instruction remain idle
    - Result from ALU stage will pass through to be used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

  - Result to be stored (if applicable)

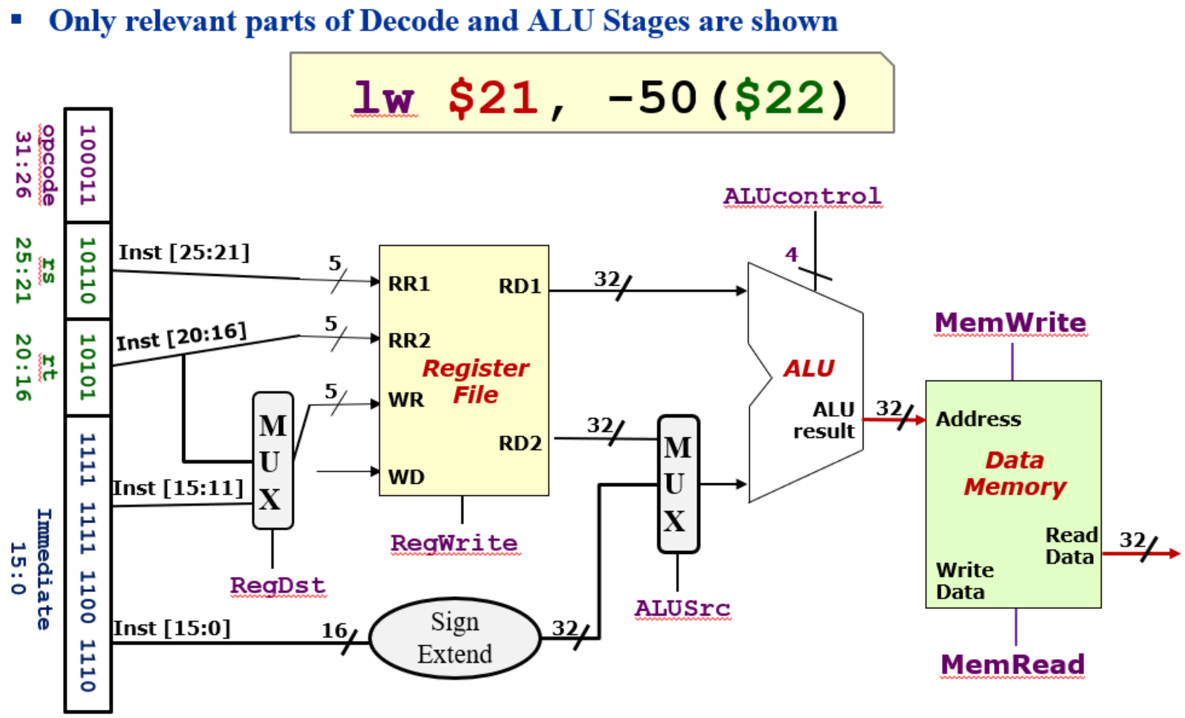


Element: Data Memory

- Storage element for the data of a program
- Inputs: (1) Memory address, (2) data to be written (Write Data) for store instructions
- Output: Data read from memory (Read Data) for load instructions



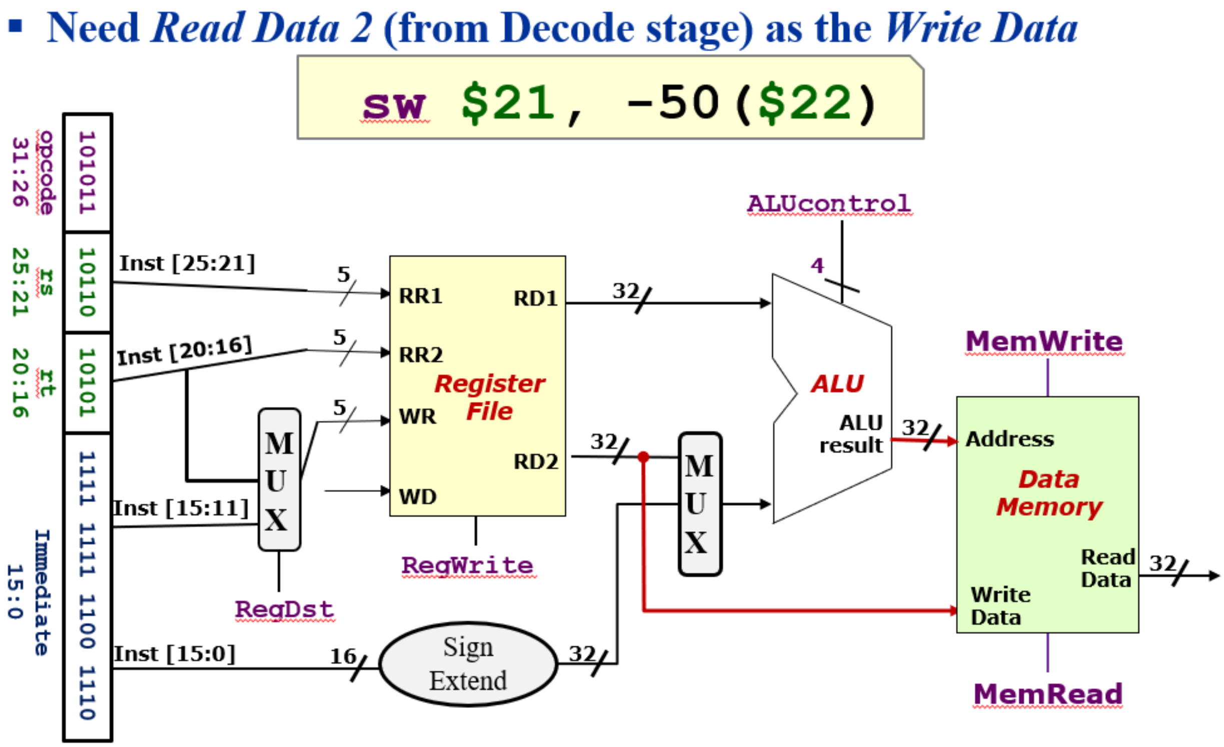
Load Instruction



- Only relevant parts of Decode and ALU Stages are shown

lw $21, -50($22)

Store Instruction

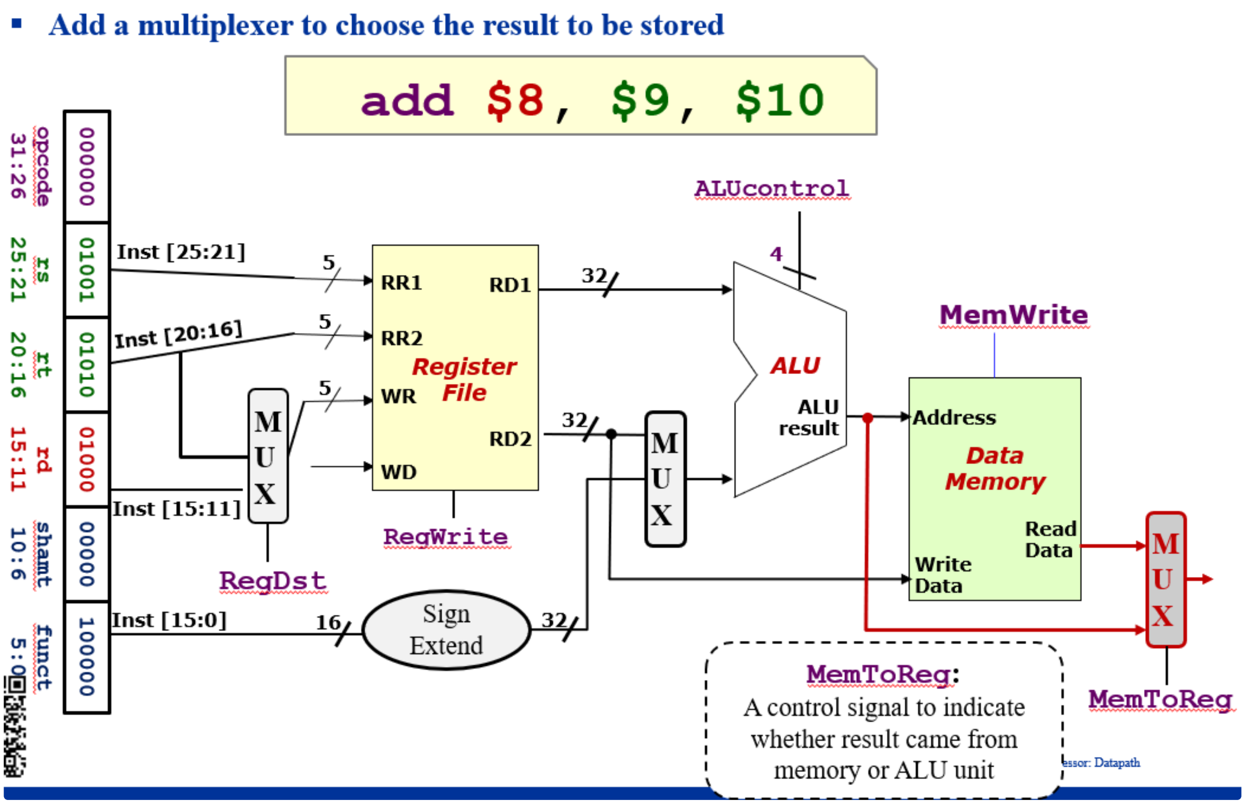


- Need *Read Data 2* (from Decode stage) as the *Write Data*

sw $21, -50($22)

> 1. **lw (Load Word) 指令**：它的功能是从内存中加载一个字（word，通常为32位）到寄存器中。该指令需要提供一个基址寄存器和一个偏移量来确定要读取的内存地址。例如，指令 `lw $t0, 4($t1)` 会从内存地址 `$t1 + 4` 加载一个字并存放到寄存器 `$t0` 中。
> 2. **sw (Store Word) 指令**：与 `lw` 指令相反，`sw` 的功能是将一个寄存器中的字存储到内存中。同样，这需要一个基址寄存器和一个偏移量来确定要写入的内存地址。例如，指令 `sw $t0, 4($t1)` 会将寄存器 `$t0` 中的内容存储到内存地址 `$t1 + 4` 。

For Non-Memory Instruction (`add`)



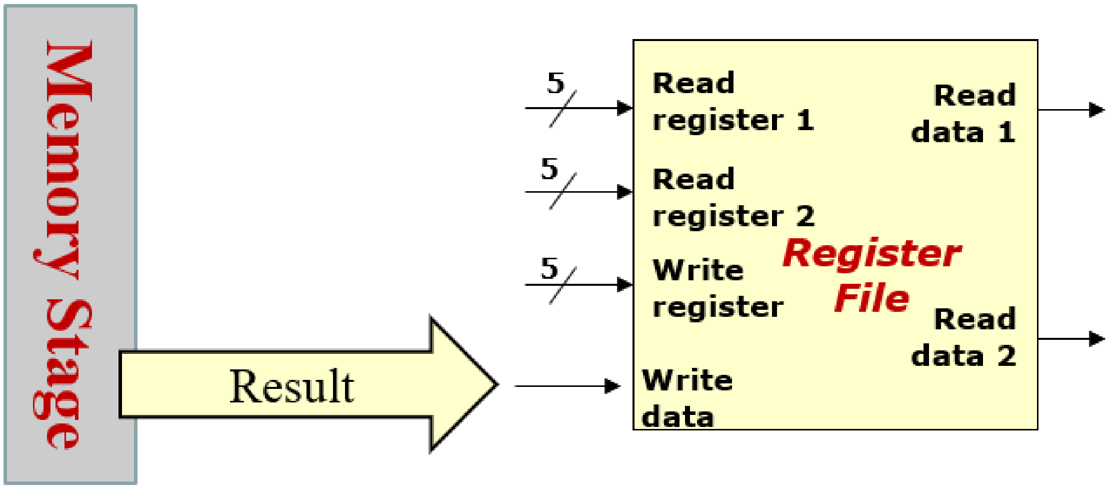
- Add a multiplexer to choose the result to be stored

add $8, $9, $10

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

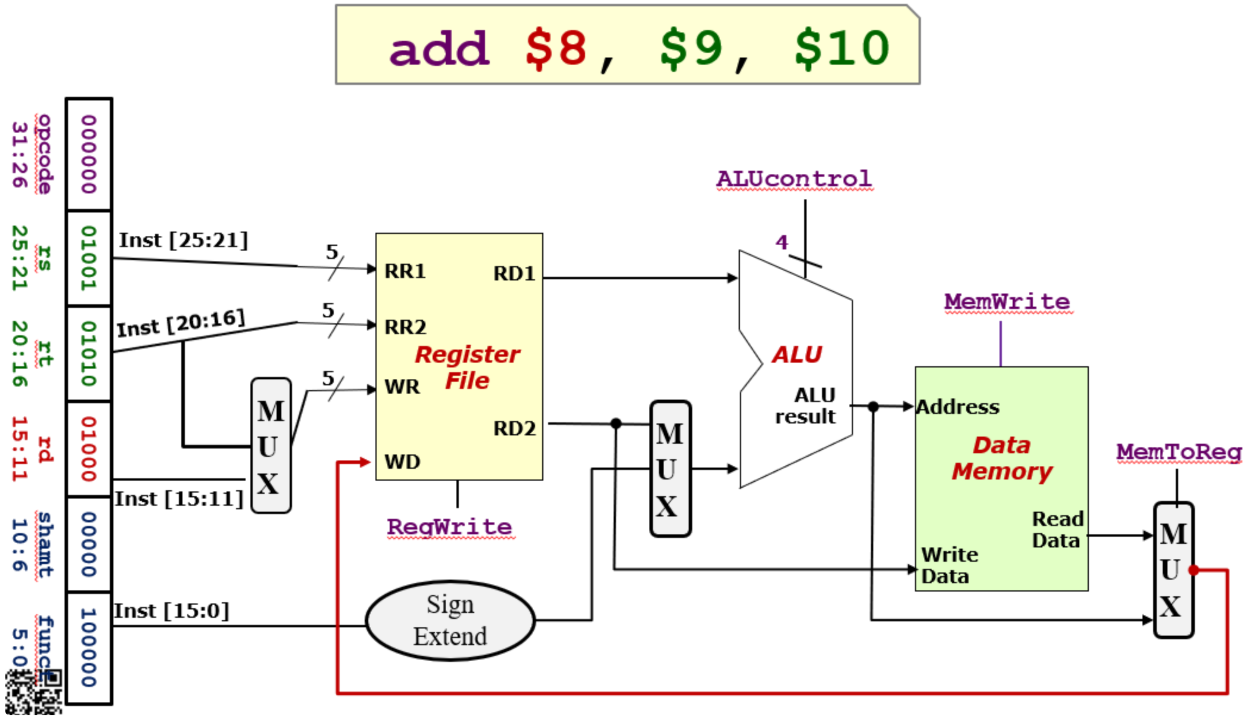
6.5.5 Register Write Stage

- 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
    - These are no result to be written
    - These instructions remain idle in this stage
- Input from previous stage (Memory):
  - Computation result either from memory or ALU

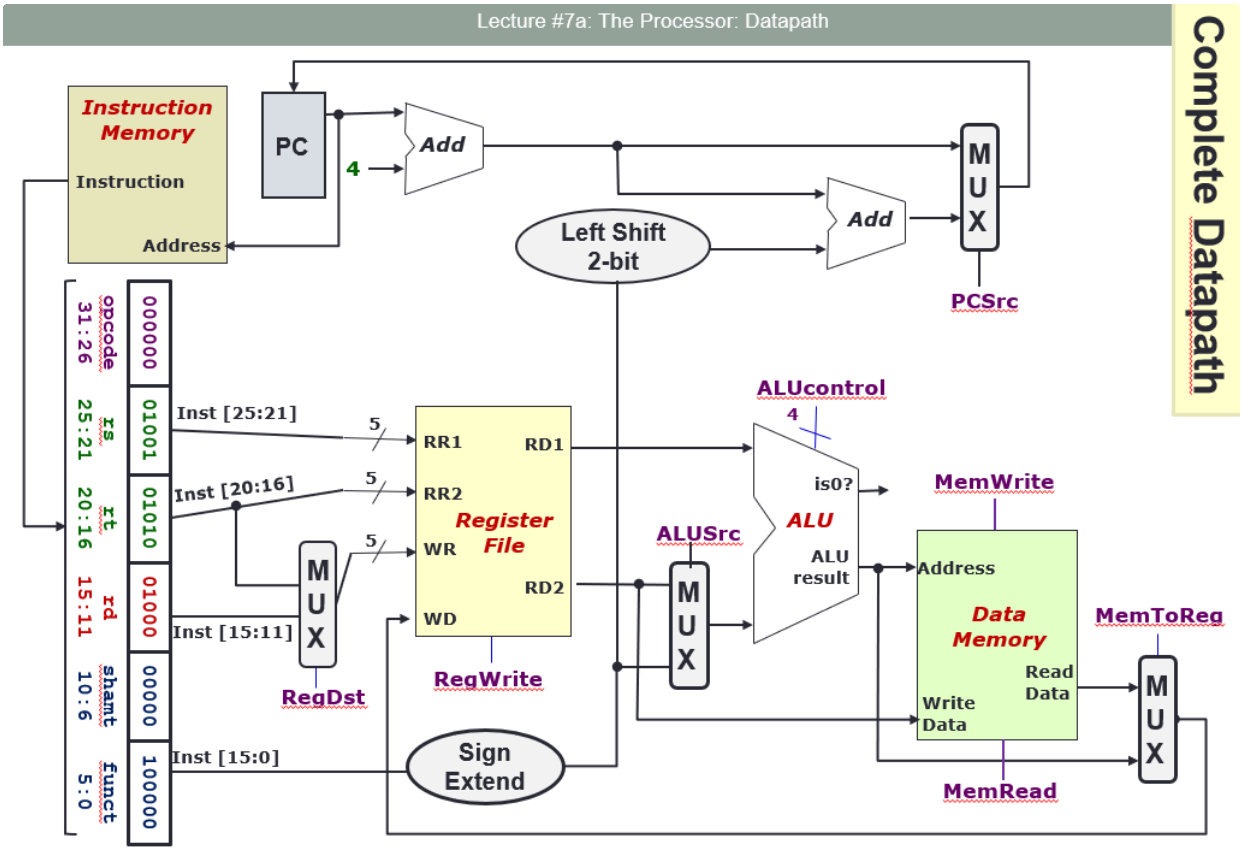


- Result Write stage has no additional element:
    - Basically just route the correct result into register file
    - The Write Register number is generated way back in the Decode Stage

Routing



6.6 Complete Datapath


Lecture #7a: The Processor: Datapath

Complete Datapath