CprE 3810: Computer Organization and Assembly Level Programming

Processor Design

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Processor Design Approach

- Break operation down to steps even logic gates can understand
- Generally decompose into two kinds of operations:
  - Things that deal with the real data (datapath)
  - Things that control the stuff operating on the real data (control)
- Find a decomposition that is simple and efficient
- We will start simple
  - Later (in lecture and lab), we'll add features to improve performance

The Processor: Datapath & Control

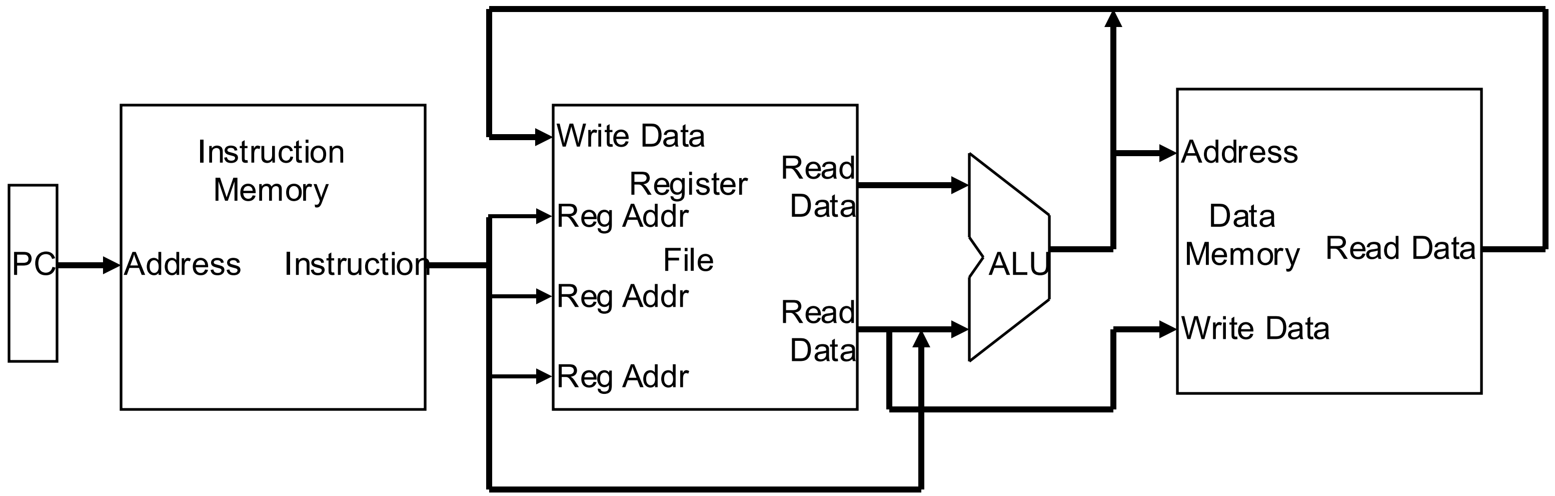
- Textbook RISC-V implementation simplified to contain only (~~added in slides~~):
  - Memory-reference instructions: `lw, sw`
  - Arithmetic-logical instructions: `add, sub`~~`, and, or, xor, slt, sltu`~~
  - Arithmetic-logical immediate instructions: ~~`addi, andi, ori, xori, slti, sltiu`~~
  - Control flow instructions: `beq,`~~`j`~~

- You lab version will require more instructions and thus modifications to this base design!

- Generic implementation:
  - Use the program counter (**PC**) to supply the instruction address and **fetch** the instruction from memory (and update the PC)
  - **Decode** the instruction (and read registers)
  - **Execute** the instruction (already built in lab)

Abstract Implementation View

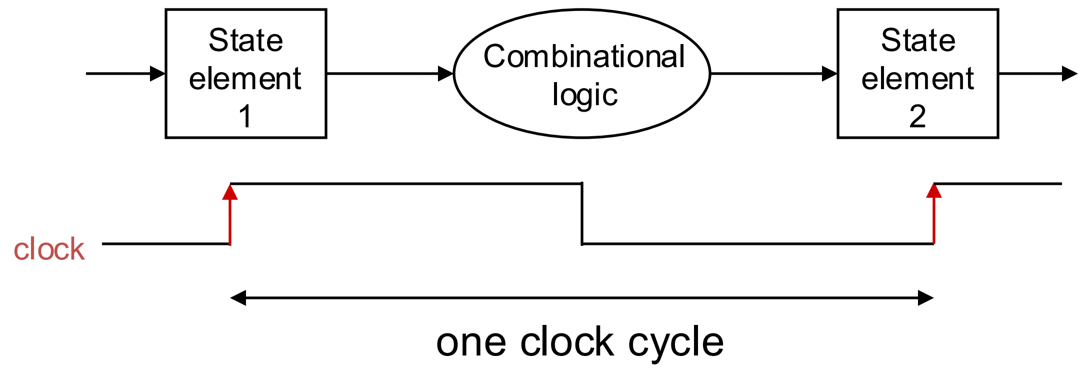
- Two types of blocks:
  - Elements that operate on data values (combinational)
  - Elements that contain state (sequential)



- **Single cycle** operation
- Split memory (Harvard) model - one memory for instructions and one for data

Edge-triggered Implementation

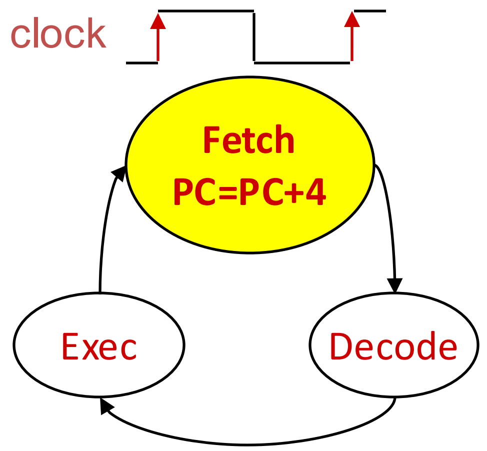
- An edge-triggered methodology, typical execution
  - Read contents of some state elements (combinational activity, so no clock control signal needed)
  - Send values through some combinational logic
  - Write results to one or more state elements on clock edge

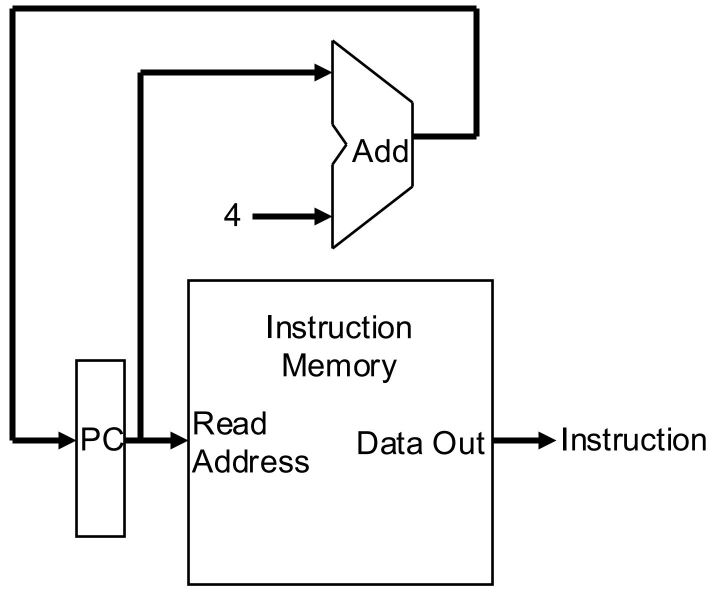


- Assumes state elements are written on every clock cycle; if not, need explicit write control signal
  - Write occurs only when both the write control is asserted and the clock edge occurs

Fetching Instructions

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next (sequential) instruction

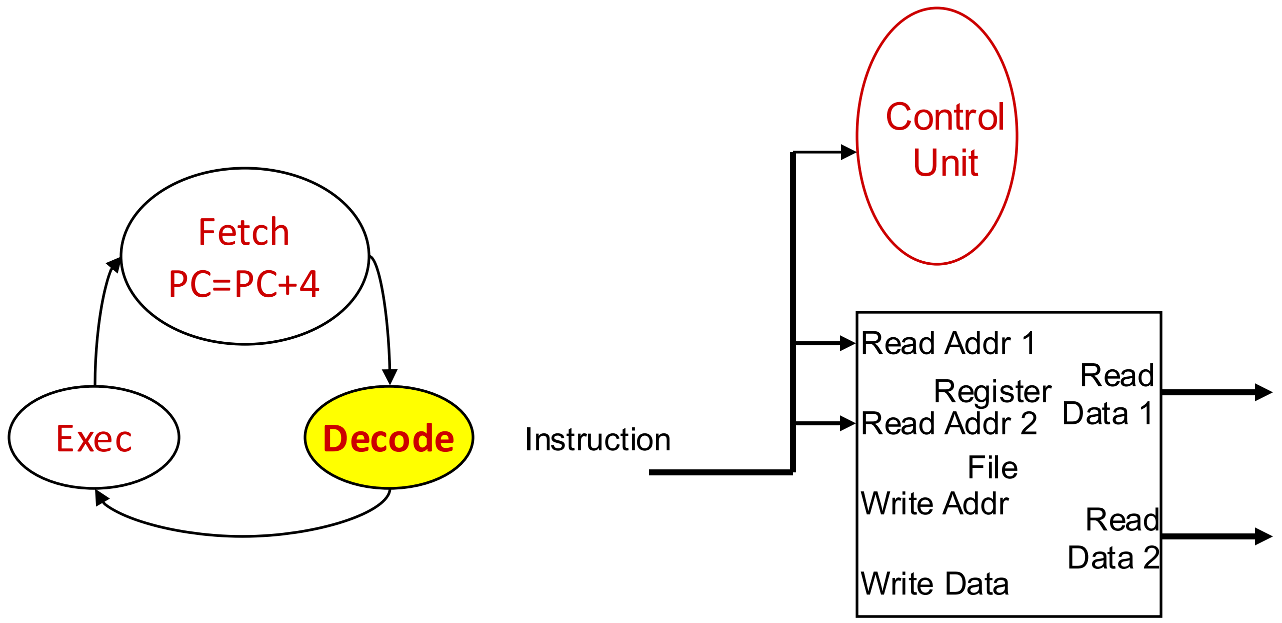




  - PC is updated every clock cycle, so it does not need an explicit write control signal – just a clock signal
  - Reading from the Instruction Memory is a combinational activity

Decoding Instructions

- Decoding instructions involves
  - Sending the fetched instruction's opcode and function field bits to the control unit



  - Reading two values from the Register File
    - Register File addresses are contained in the instruction

Reading Registers "Just in Case"

- Note that both RegFile read ports are active for all instructions during the Decode cycle using the `rs1` and `rs2` instruction field addresses
  - Since we haven't decoded the instruction yet, we don't know what the instruction is!
  - *Just in case* the instruction uses values from the RegFile do "work ahead" by reading the two source operands

Which instructions *do* make use of the RegFile values?

Reading Registers "Just in Case"

- Note that both RegFile read ports are active for all instructions during the Decode cycle using the `rs1` and `rs2` instruction field addresses
  - Since we haven't decoded the instruction yet, we don't know what the instruction is!
  - *Just in case* the instruction uses values from the RegFile do "work ahead" by reading the two source operands

Which instructions *do* make use of the RegFile values?

- Also, all instructions (except **`jal`**) use the ALU after reading the registers

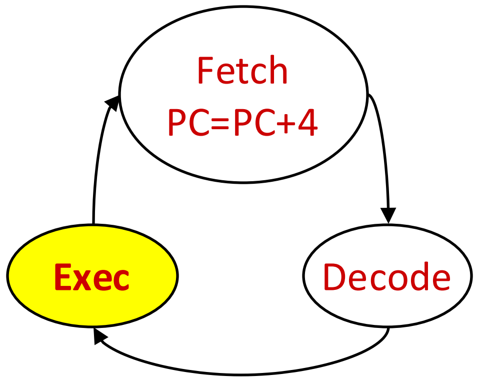
For what purpose? arithmetic? memory? control flow?

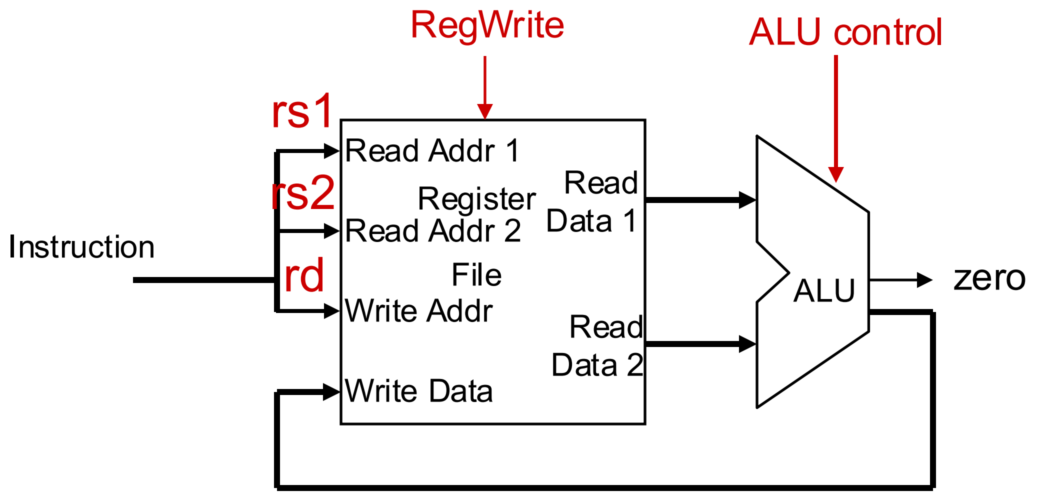
Executing R Format Operations

- R format operations (`add, sub, slt, and, or`)

Instruction Bits	7	5	5	3	5	7
	funct7	rs2	rs1	funct3	rd	OP=0110011
	Function Code	Second Source Register	First Source Register	Function Code	Result Register	Operation Code

  – Perform operation (op and funct) on values in rs1 and rs2
  – Store the result back into the Register File (into location rd)





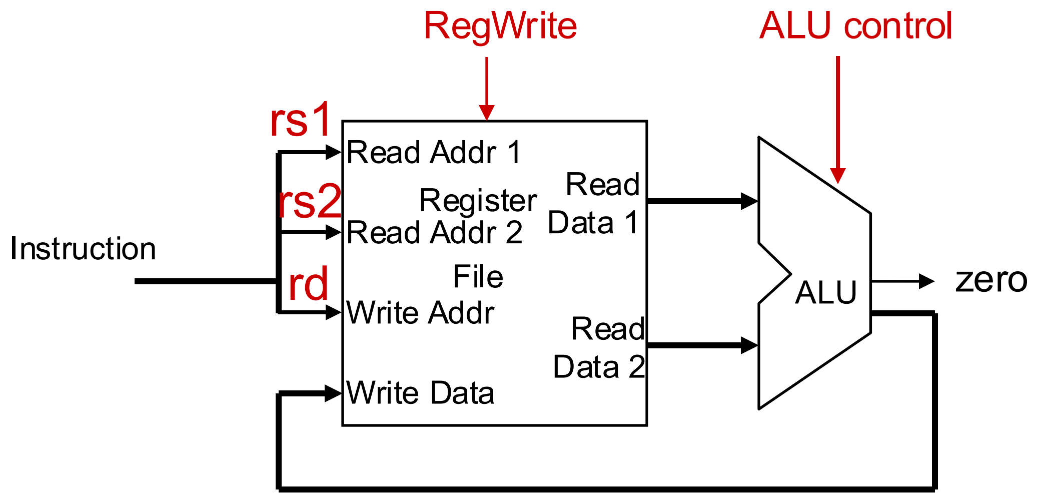
  – Note that Register File is not written every cycle (e.g., `sw` or `beq`), so we need an explicit write control signal for the Register File

Consider the `slt` Instruction

- Remember the R format instruction `slt`

```
slt t0, s0, s1   # if s0 < s1
                 #    then  t0 = 1
                 # else  t0 = 0
```

  – Where does the 1 (or 0) come from to store into t0 in the Register File at the end of the execute cycle?



Executing Load Operations

- Load operations have to:

	12	5	3	5	7
Instruction Bits	**imm**	**rs1**	**func3**	**rd**	**OP**
	Immediate Value	First Source Register	Function Code	Destination Register	Operation Code

  – Compute a memory address by adding the base register (in rs1) to the 12-bit signed offset field in the instruction
    - Base register was read from the Register File during decode
    - Offset value in upper 12 bits of the instruction must be sign extended to create a 32-bit signed value

  – Load value, read from the Data Memory, must be stored in the Register File

Executing Load and Store Operations

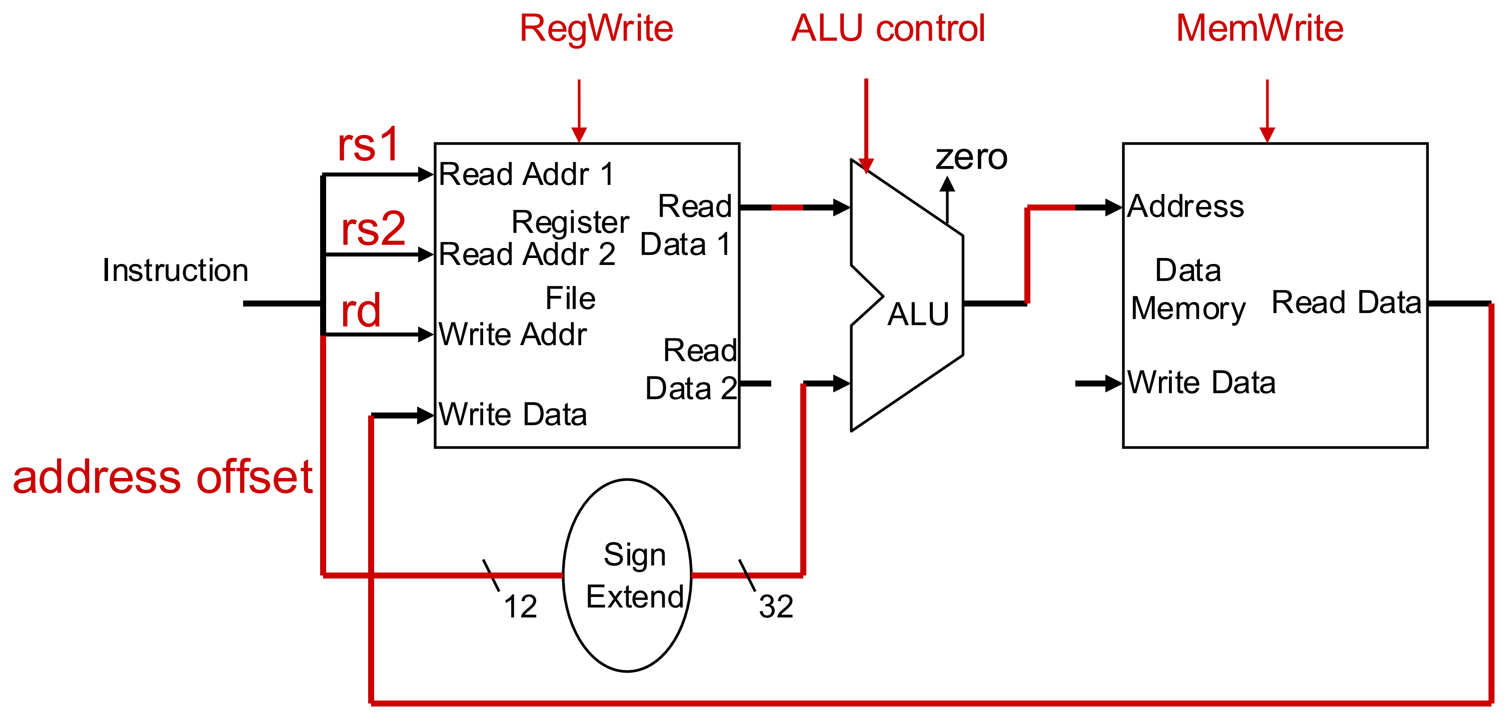
- Load and store operations have to:

	7	5	5	3	5	7
Instruction Bits	**imm[11:5]**	**rs2**	**rs1**	**func3**	**imm[4:0]**	**OP**
	Immediate Value	Second Source Register	Second Source Register	Function Code	Destination Register	Operation Code

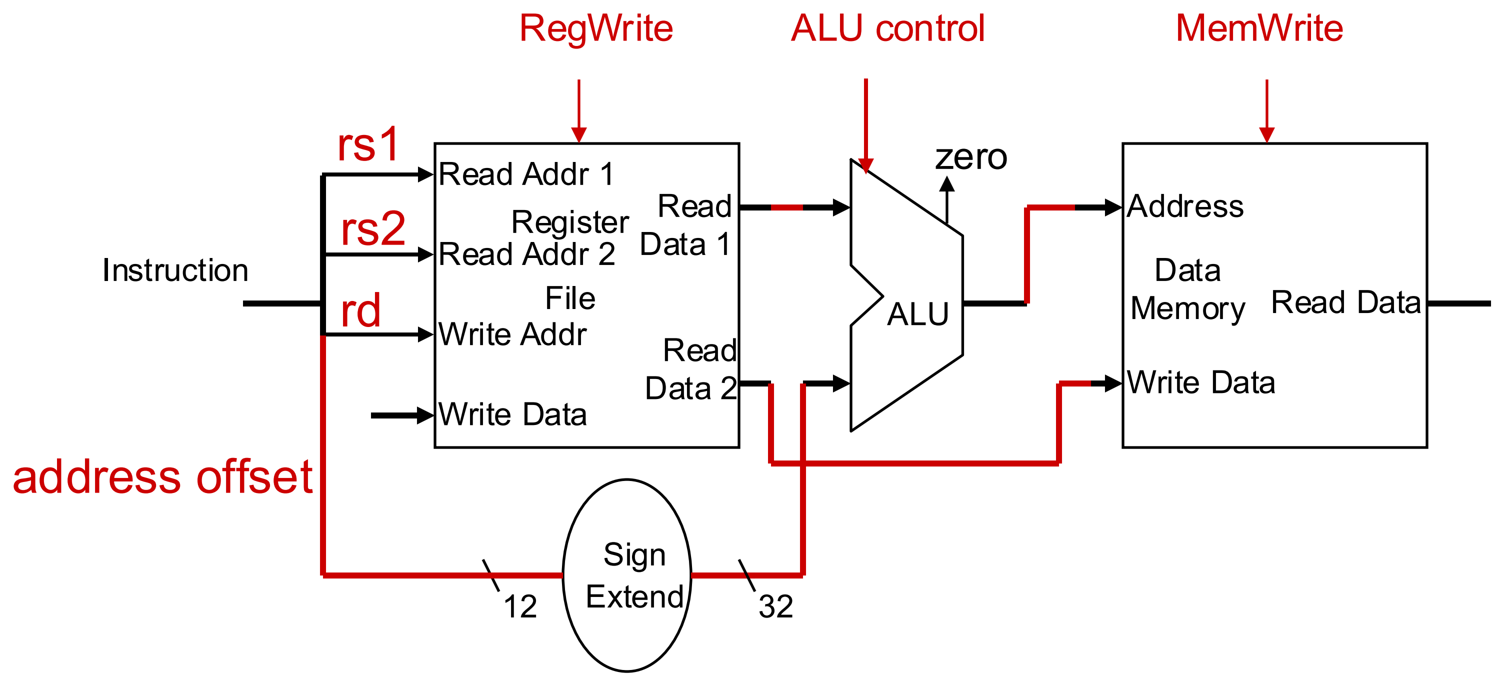
  – Compute a memory address by adding the base register (in rs1) to the 12-bit signed offset field in the instruction
    - Base register was read from the Register File during decode
    - Offset value in the upper 7 bits concatenated with bits 12:7 of the instruction must be sign extended to create a 32-bit signed value

  – Store value, read from the Register File during decode (in rs2), must be written to the Data Memory

Executing Load Operations (cont.)



Executing Store Operations (cont.)



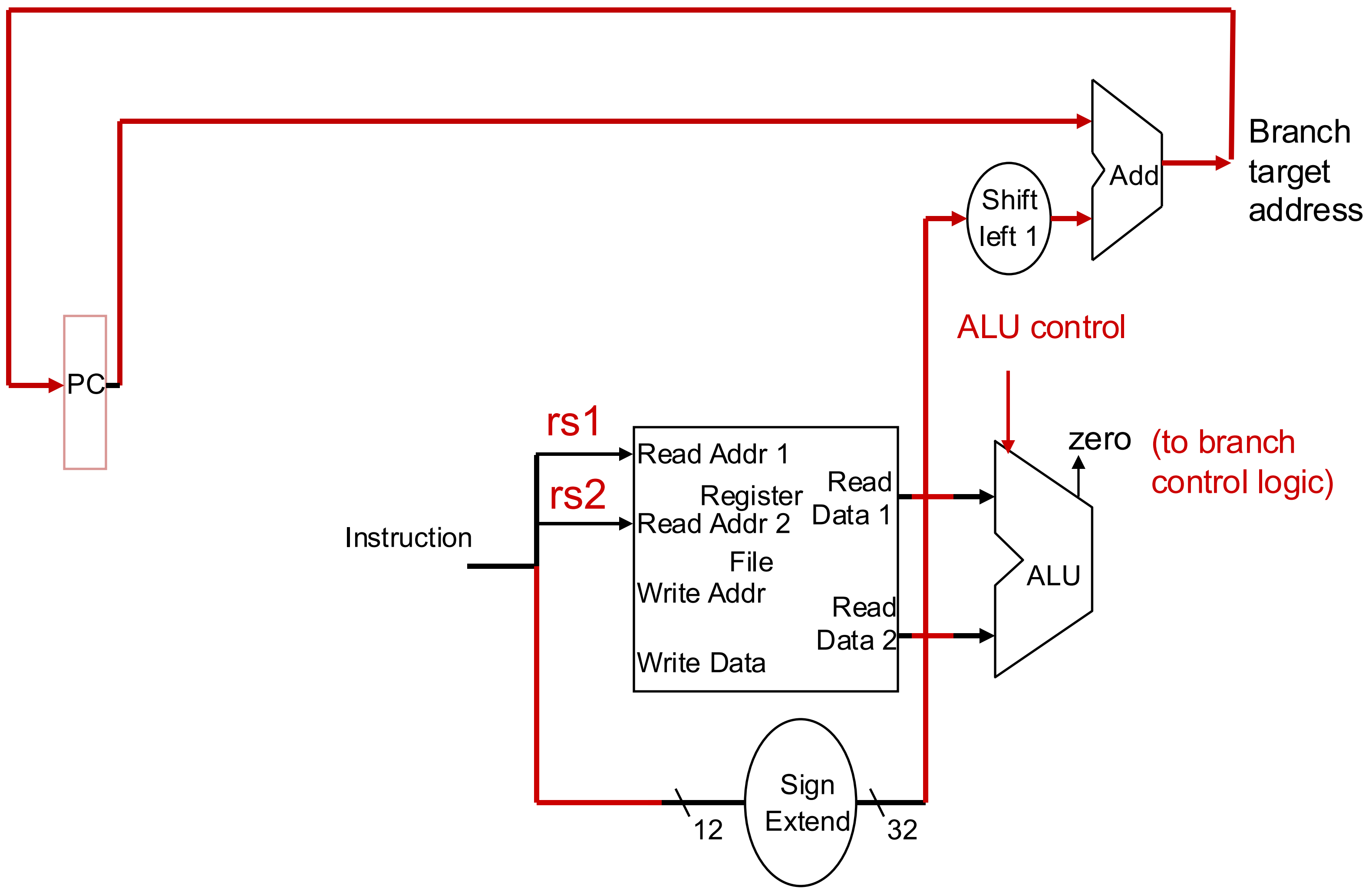
Executing Branch Operations

- Branch operations have to

	7	5	5	3	5	7
Instruction Bits	**imm[12|10:5]**	**rs2**	**rs1**	**func3**	**imm[4:1|11]**	**OP**
	Immediate Value	Second Source Register	Second Source Register	Function Code	Destination Register	Operation Code

  - Compare the operands read from the Register File during decode (rs1 and rs2 values) for equality (`zero` ALU output)
  - Compute the branch target address by adding the PC to the12-bit signed immediate field in the instruction that is concatenated with 1 bit of zero and sign extended
    - The “base register” is the PC
    - Offset value must be appropriately selected from the instruction and sign extended to create a 32-bit signed value and then shifted left 1 bits

Executing Branch Operations (cont.)



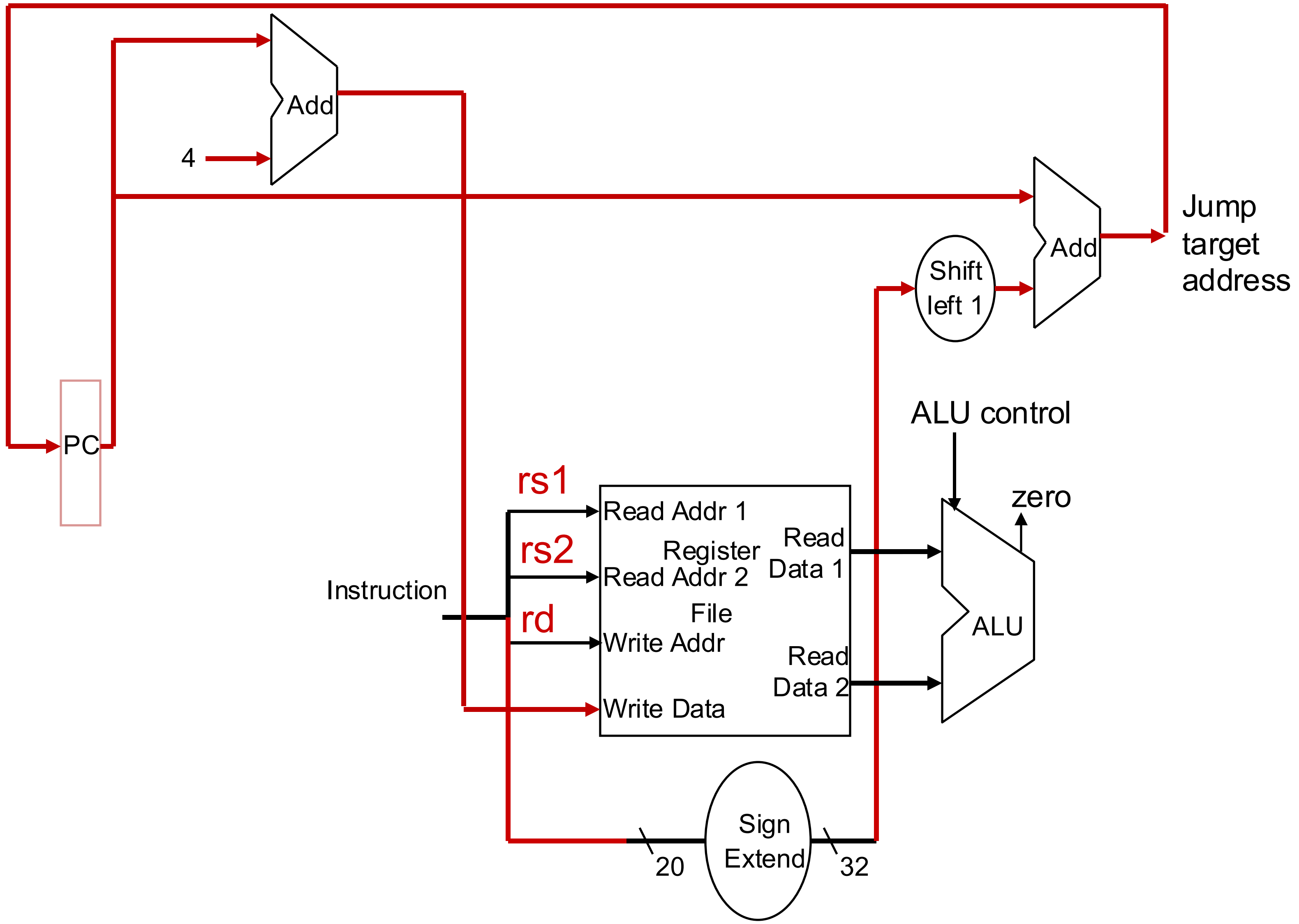
Executing Jump Operations

- Jump (`jal` really) operations have to

	20	5	7
Instruction Bits	**imm[20|10:1|11|19:12]**	**rd**	**OP**
	Immediate	Destination Register	Operation Code

  - Compare the operands read from the Register File during decode (rs1 and rs2 values) for equality (`zero` ALU output)
  - Compute the branch target address by adding the PC to the12-bit signed immediate field in the instruction that is concatenated with 1 bit of zero and sign extended
    - The “base register” is the PC
    - Offset value must be appropriately selected from the instruction and sign extended to create a 32-bit signed value and then shifted left 1 bits

Executing Jump Operations (cont.)



Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)