CO20-320241

# Computer Architecture and Programming Languages

CAPL

## Lecture 18 & 19

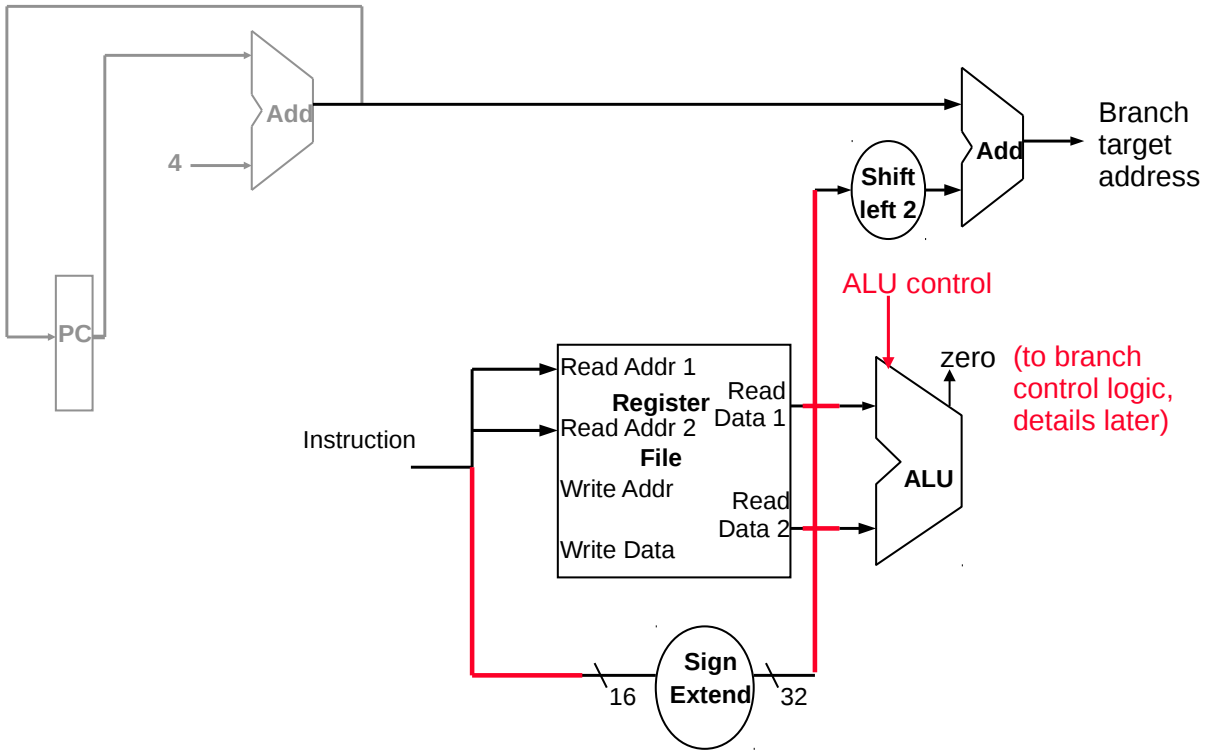
Dr. Kinga Lipskoch

Fall 2019

# Executing Branch Operations

Branch operations involve:

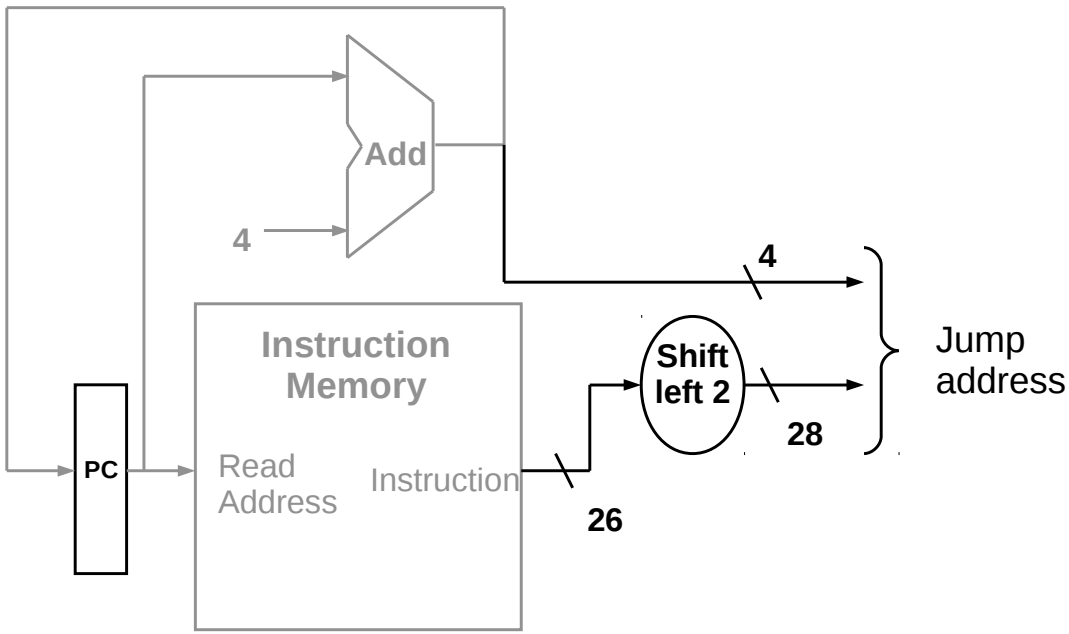
- compare the operands read from the Register File during decode for equality (zero ALU output)
- compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instruction



# Executing Jump Instructions

Jump operations involve:

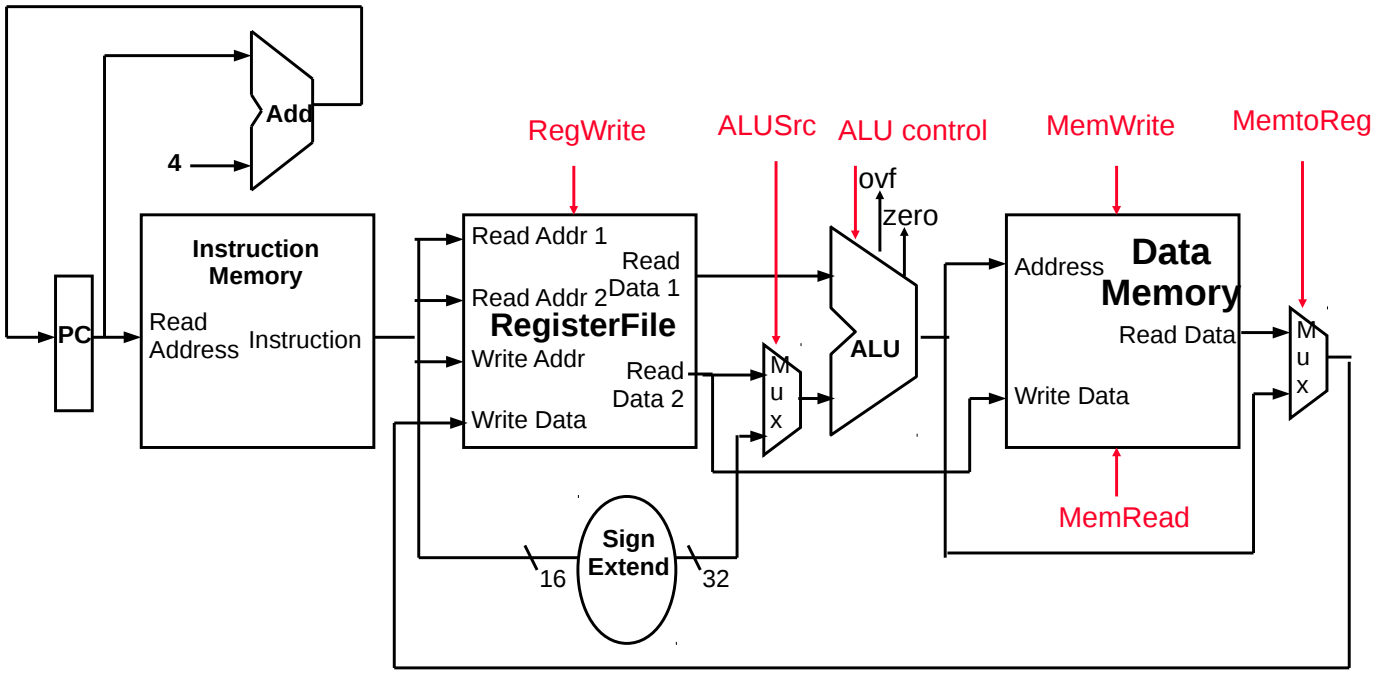
- replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



# Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instruction in one clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - multiplexers needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- Cycle time is determined by length of the longest path

# Fetch, R, and Memory Access Portions




- arithmetic-logical instr. use ALU with input from two registers
- memory instr., second input is sign-extended 16-bit offset field
- value in destination register comes either from ALU (R-type) of memory (load)

# ALU Control

Four bits used for control inputs:

- Only 6 of 16 bit combinations are used

| ALU control lines | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set on less than |
| 1100 | NOR |

- R-type: AND, OR, subtract or set on less than
- Memory: add
- Branch equal: subtraction

# ALU Control Bits

| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 0010 |
| SW | 00 | save word | XXXXXX | add | 0010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | and | 0000 |
| R-type | 10 | OR | 100101 | or | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

- Opcode determines setting of ALUOp bits
- When ALUOp is 00 or 01, ALU action does not depend on function field
- When ALUOp is 10, function code is used to set ALU control input

# Truth Table for the Four ALU Control Bits

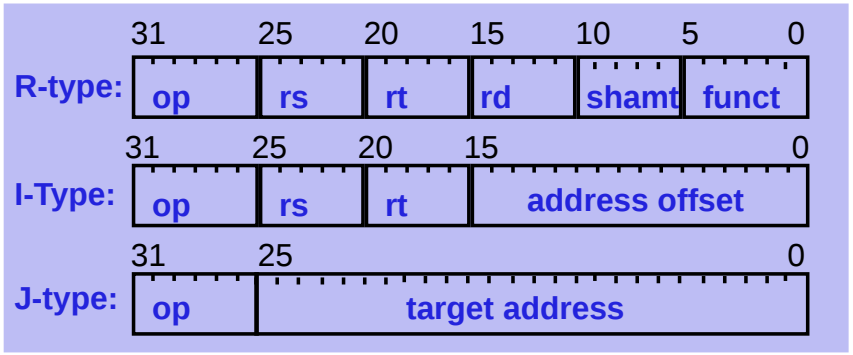
| ALUOp | | Funct field | | | | | | Operation |
|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp2 | F5 | F4 | F3 | F2 | F1 | F0 | |
| 0 | 0 | X | X | X | X | X | X | 0010 |
| X | 1 | X | X | X | X | X | X | 0110 |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 0010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 |
| 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 |

- X are "don't care items"
- Such a truth table can be easily turned into gates (therefore encoded in hardware)

# Adding the Control

- Selecting the operations to perform (ALU, Register File and Memory read/write)
- Controlling the flow of data (multiplexor inputs)
- Observations:

- op field always in bits $31-26$
- addr. of registers to be read are always specified by the `rs` field (bits $25-21$) and `rt` field (bits $20-16$); for `lw` and `sw`, `rs` is the base register
  - addr. of register to be written is in one of two places – in `rt` (bits $20-16$) for `lw`; in `rd` (bits $15-11$) for R-type instructions
  - offset for `beq`, `lw`, and `sw` always in bits $15-0$



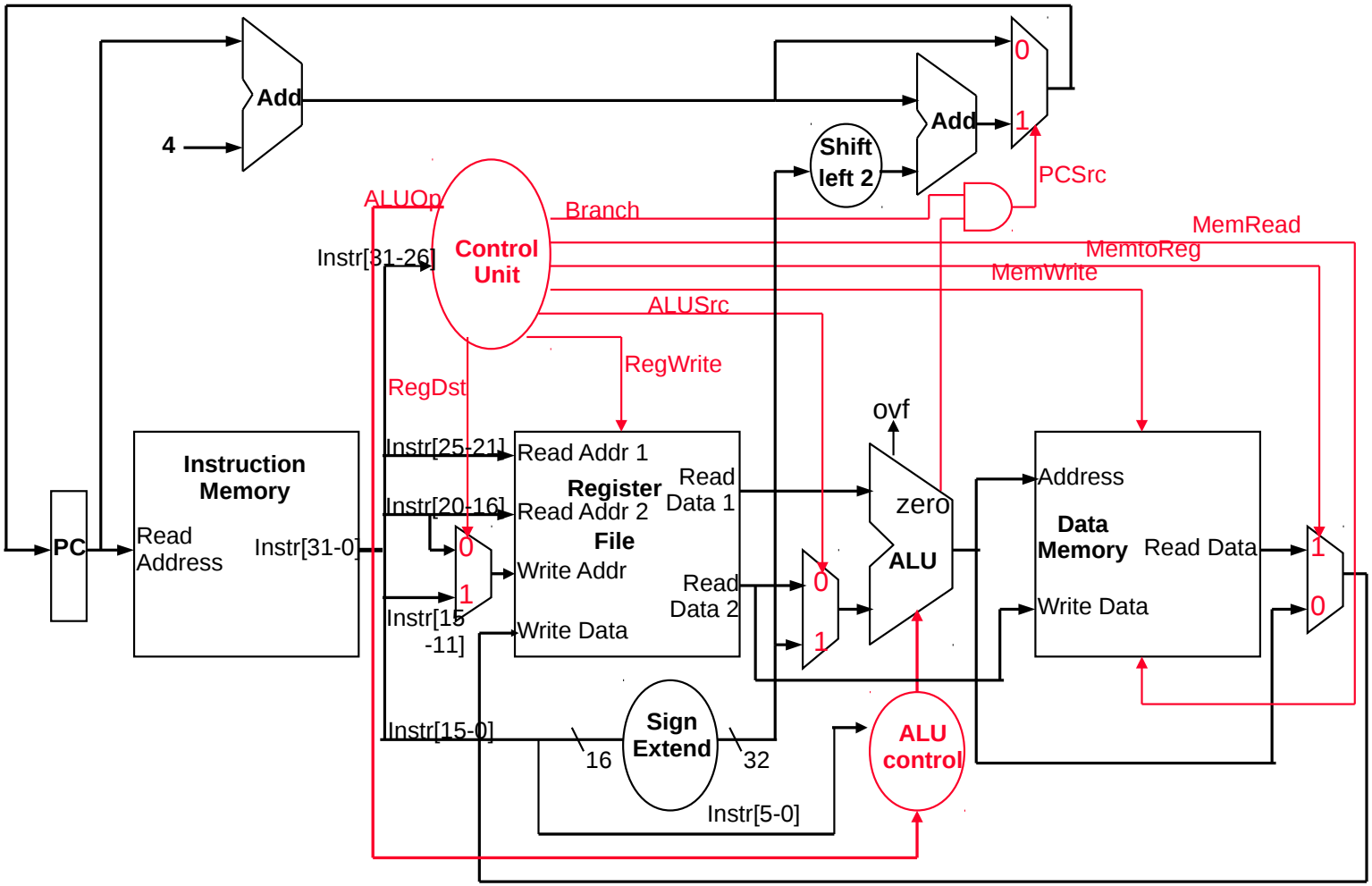
# Summary of Control Lines

- RegDest
  Source of the destination register for the operation
- RegWrite
  Enables writing a register in the register file
- ALUsrc
  Source of second ALU operand, can be a register or part of the instruction
- PCsrc
  Source of the PC (increment [PC + 4] or branch)
- MemRead/MemWrite
  Reading/Writing from memory
- MemtoReg
  Source of write register contents

# The Seven Control Signals

| Signal name | Effect when deasserted | Effect when asserted |
|---|---|---|
| RegDst | The register destination number for the Write register comes from the rt field (bits 20:16) | The register destination number for the Write register comes from the rd field (bits 15:11). |
| RegWrite | None. | The register on the Write register input is written with the value on the Write data input. |
| ALUSrc | The second ALU operand comes from the second register file output (Read data 2) | The second ALU operand is the sign-extended, lower 16 bits of the instruction. |
| PCSrc | The PC is replaced by the output of the adder that computes the value of PC + 4. | The PC is replaced by the output of the adder that computes the branch target. |
| MemRead | None. | Data memory contents designated by the address input are put on the Read data output. |
| MemWrite | None. | Data memory contents designated by the address input are replaced by the value on the Write data input. |
| MemtoReg | The value fed to the register Write data input comes from the ALU. | The value fed to the register Write data input comes from the data memory. |

# Single Cycle Datapath with Control Unit

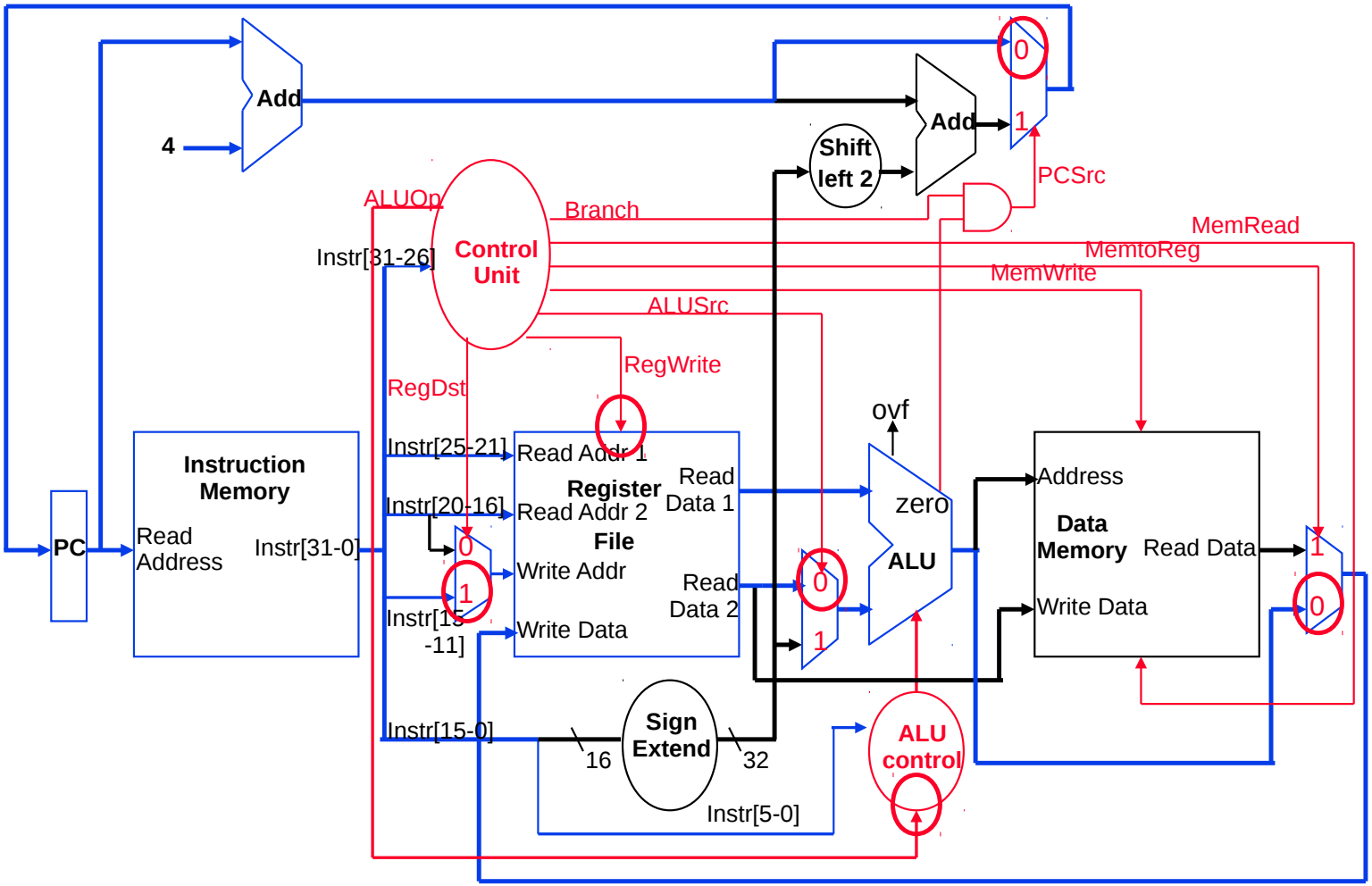


# Control Lines

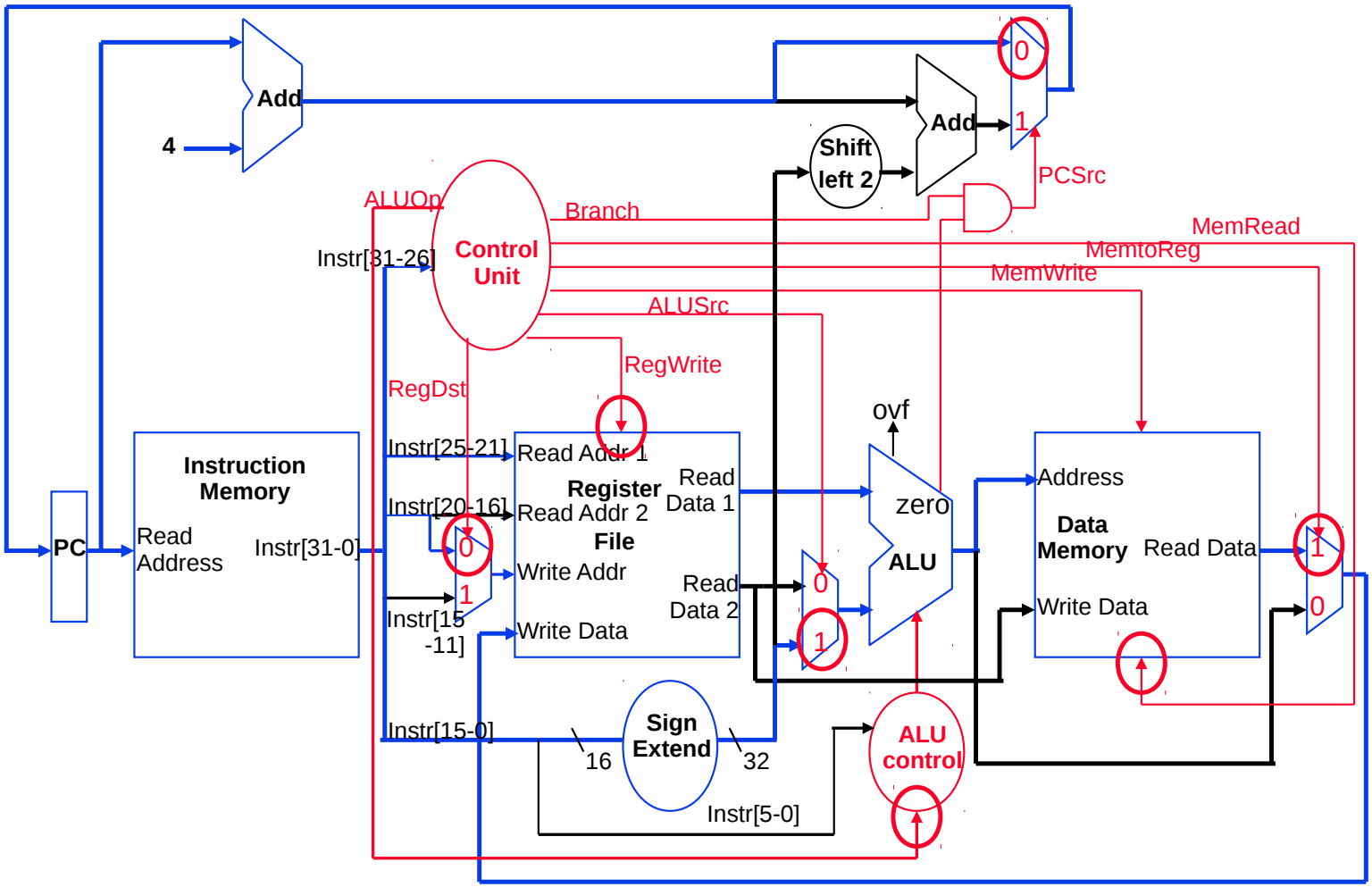
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp1 | ALUOp2 |
|---|---|---|---|---|---|---|---|---|---|
| R-Format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| `lw` | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| `sw` | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| `beq` | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

The setting of the control lines is completely determined by the opcode fields of the instruction

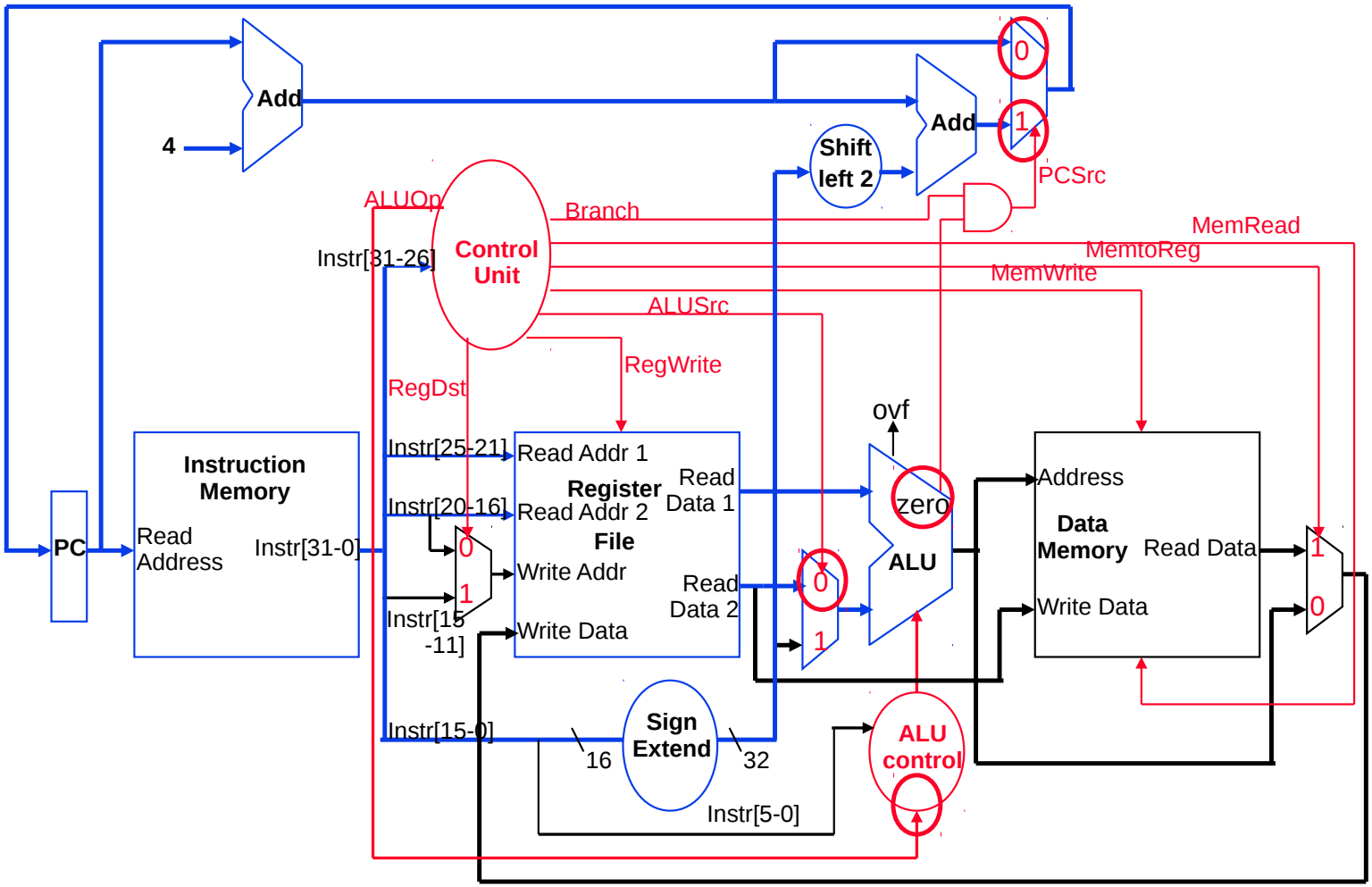
# R-type Instruction Data/Control Flow



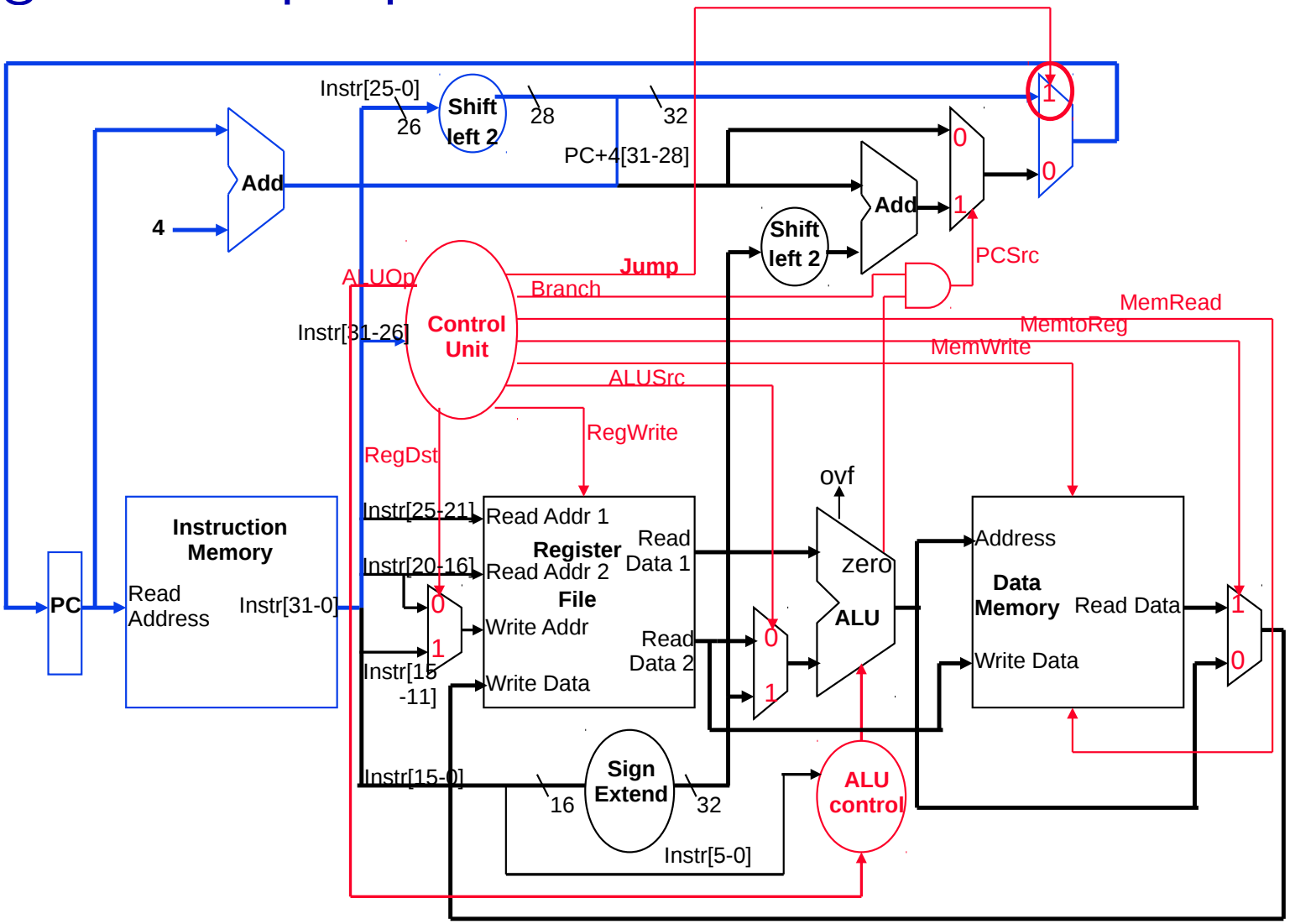
# Load Word Instruction Data/Control Flow



# Branch Instruction Data/Control Flow



# Adding the Jump Operation



# Latencies in the Datapath (1)

Latency = a measure of the time delay experienced by a system
First a few notations:

- I-Mem = Instruction Memory
- D-Mem = Data Memory
- ALU = Arithmetic Logic Unit
- Add = Adder
- Mux1, Mux2, Mux3, Mux4 = 4 different multiplexors (which are in the datapath figure)
- Sign-Extent
- Shift-left-2
- RegF = Register File
- WBack = Write Back to the Register File

# Latencies in the Datapath (2)

All possible paths for different instruction classes:

- ALU operations like `add`, etc.
- Possible paths for ALU operations:
  1. I-Mem $\rightarrow$ RegF $\rightarrow$ ALU $\rightarrow$ Mux3 $\rightarrow$ WBack to RegF
  2. I-Mem $\rightarrow$ RegF $\rightarrow$ Mux2 $\rightarrow$ ALU $\rightarrow$ Mux3 $\rightarrow$ WBack to RegF
  3. I-Mem $\rightarrow$ Mux1 $\rightarrow$ RegF (Write address)
  4. Add $\rightarrow$ Mux4 $\rightarrow$ PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

# Latencies in the Datapath (3)

- The `sw` instruction:
- Possible paths for `sw`:
  1. I-Mem → RegF → ALU → D-Mem
  2. I-Mem → RegF → D-Mem
  3. I-Mem → Sign-Extend → Mux2 → ALU → D-Mem
  4. Add → Mux4 → PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

# Latencies in the Datapath (4)

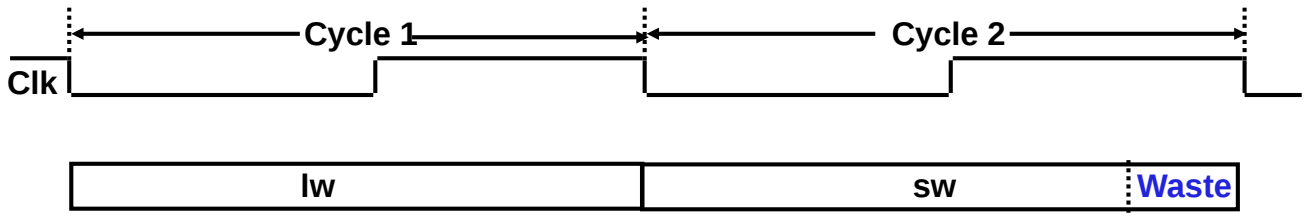
- The `lw` instruction:
- Possible paths for `lw`:
  1. I-Mem → RegF → ALU → D-Mem → Mux3 → WBack to RegF
  2. I-Mem → Mux1 → RegF (Write address)
  3. I-Mem → Sign-Extend → Mux2 → ALU → D-Mem → Mux3 → WBack to RegF
  4. Add → Mux4 → PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

# Latencies in the Datapath (5)

- Branching instructions, for example `beq`:
- Possible paths for `beq`:
  1. I-Mem → RegF → ALU → Mux4 → PCWrite (update PC)
  2. I-Mem → RegF → Mux2 → ALU → Mux4 → PCWrite (update PC)
  3. I-Mem → Sign-Extend → Shift-left-2 → Add → Mux4 → PCWrite (update PC)
  4. Add → Mux4 → PCWrite (update PC)
- Depending the latencies given, you need to select the longest path in terms of time

# Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
- Especially problematic for more complex instructions like floating point multiply




- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

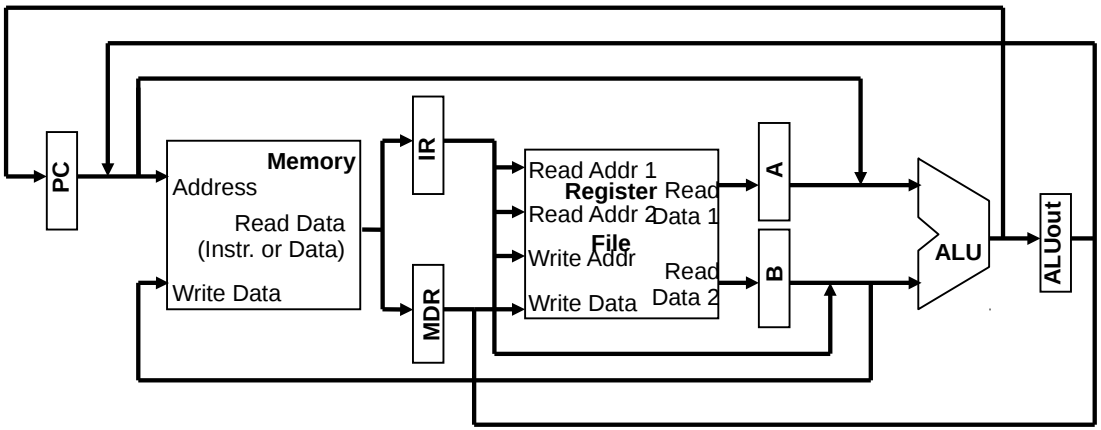
But, it is simple and easy to understand

# Multicycle Datapath Approach (1)

- Let an instruction take more than 1 clock cycle to complete
  - Break up instructions into steps where each step takes a cycle while trying to
    - balance the amount of work to be done in each step
    - restrict each cycle to use only one major functional unit
  - Not every instruction takes the same number of clock cycles
- In addition to faster clock rates, multicycle allows functional units that can be used more than once per instruction as long as they are used on different clock cycles, as a result
  - only need one memory – but only one memory access per cycle
  - need only one ALU/adder – but only one ALU operation per cycle

# Multicycle Datapath Approach (2)

- At the end of a cycle
  - Store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer). All (except IR) hold data only between a pair of adjacent clock cycles (no write control signal needed)




IR – Instruction Register
A, B – regfile read data registers

MDR – Memory Data Register
ALUout – ALU output register

- Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

# Additional Registers Needed

- Data used by the same instruction must be stored in additional registers
  - Position is determined by two factors
    - which units fit into same clock cycle
    - what data is needed for later cycles
  - Instruction register (IR) and Memory data register (MDR) added to save output of the memory for instruction read and data read
  - A and B registers added to hold register operand values read from register file
  - ALUOut register holds the output of ALU
- All registers (except IR) will hold data just between a pair of adjacent cycles, thus do not need write control signal

# Multicycle Datapath

- Functional units shared for different purposes
  - Multiplexer needed for Memory access
    - PC or ALUOut
  - Three ALUs replaced by one
    - additional multiplexer for first ALU input (A or PC) added
    - 4-way multiplexer for second ALU input
- More registers and multiplexers, but
- Less memory units (1 instead of 2)
- Fewer adders (2)
- Reduced hardware cost

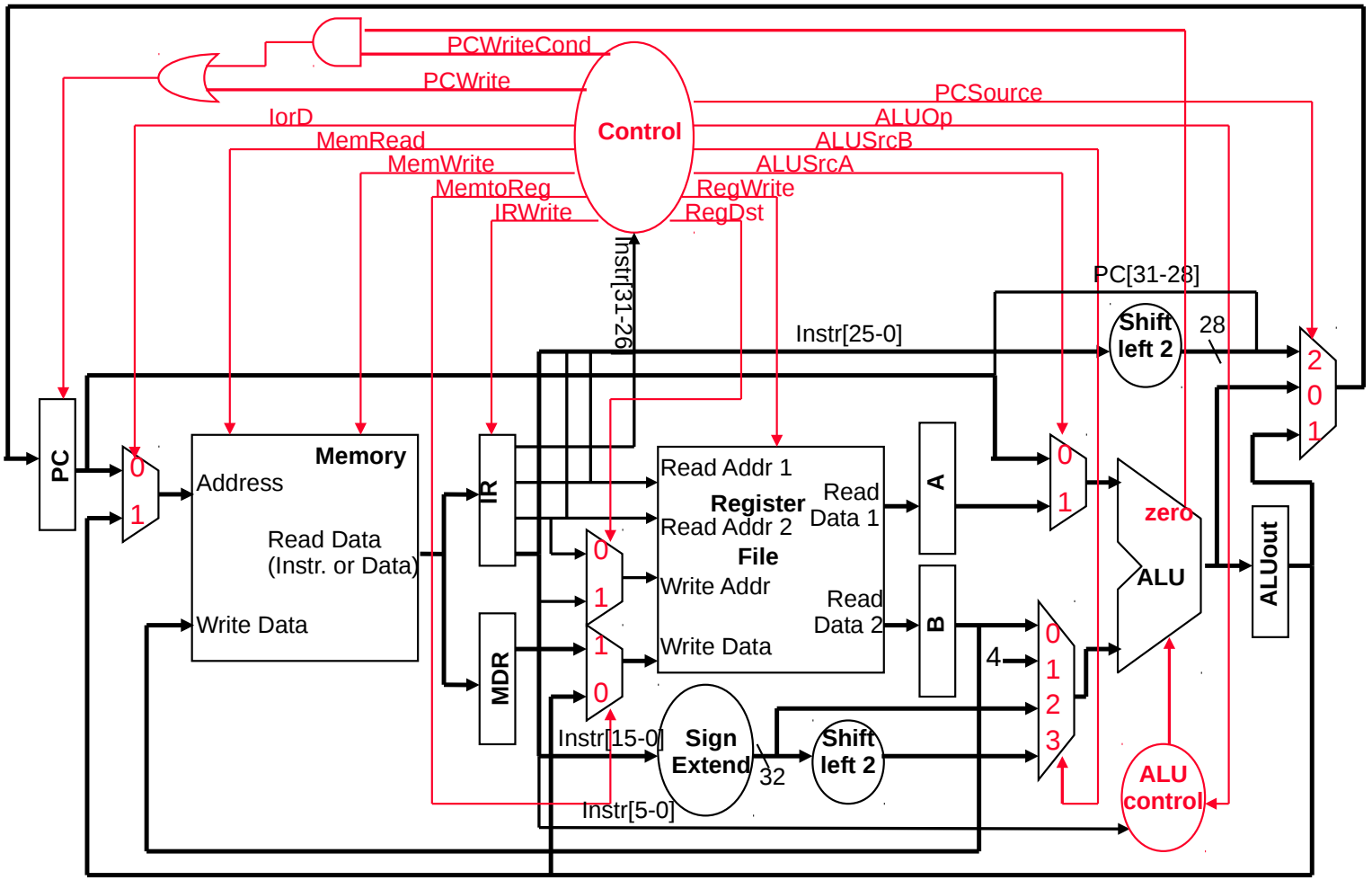
# More Control Lines Needed

- Multiple clock cycles per instruction
  - State units (PC, memory, registers) need write control lines
  - Memory needs read signal
  - Additional multiplexers need control line, 4-way needs 2
- PC has three possible sources
  - PCWrite, unconditional write of PC
  - PCWriteCond, cause write of PC, if branch is true

| Actions of the 1-bit control signals | | |
|---|---|---|
| **Signal name** | **Effect when deasserted** | **Effect when asserted** |
| RegDst | The register destination number for the Write register comes from the rt field (bits 20:16) | The register destination number for the Write register comes from the rd field (bits 15:11). |
| RegWrite | None. | The register on the Write register input is written with the value on the Write data input. |
| ALUSrcA | The first ALU operand comes from the PC. | The first ALU operand comes from the A register. |
| MemRead | None. | Content of memory at the location specified by the Address input is put on Memory data output. |
| MemWrite | None. | Memory contents at the location specified by the Address input is replaced by value on Write data input. |
| MemtoReg | The value fed to the register Write data input comes from the ALUOut. | The value fed to the register Write data input comes from the MDR. |
| IorD | The PC is used to supply the address to the memory unit. | ALUOut is used to supply the address to the memory unit. |
| IRWrite | None. | The output of the memory is written into the IR. |
| PCWrite | None. | The PC is written; the source is controlled by PCSource. |
| PCWriteCond | None. | The PC is written if the Zero output from the ALU is also active. |

| Actions of the 2-bit control signals | | |
|---|---|---|
| **Signal name** | **Value (binary)** | **Effect** |
| ALUOp | 00 | The ALU performs an add operation. |
| | 01 | The ALU performs an subtract operation. |
| | 10 | The funct field of the instruction determines the ALU operation |
| ALUSrcB | 00 | The second input to the ALU comes from the B register. |
| | 01 | The second input to the ALU is the constant 4. |
| | 10 | The second input to the ALU is the sign-extended, lower bits of the IR. |
| | 11 | The second input to the ALU is the sign-extended, lower bits of the IR shifted left 2 bits. |
| PCSource | 00 | Output of the ALU (PC+4) is sent to the PC for writing. |
| | 01 | The contents of ALUOut (branch target address) are sent to the PC for writing. |
| | 10 | The jump target address (IR[25:0]) shifted left 2 bits and concatenated with PC+4[31:28] is sent to the PC for writing. |

# The Multicycle Datapath with Control Signals



# Instructions from ISA Perspective

- Move from one-cycle to multi-cycle
  - Identifying steps that take one cycle
  - Equal distribution of execution time
  - At most one operation for each of the modules
    - ALU
    - Register file
    - Memory
- New registers if
  - The signal is computed in one cycle and used in another cycle
  - The inputs of the block generating the signal may change in the second cycle

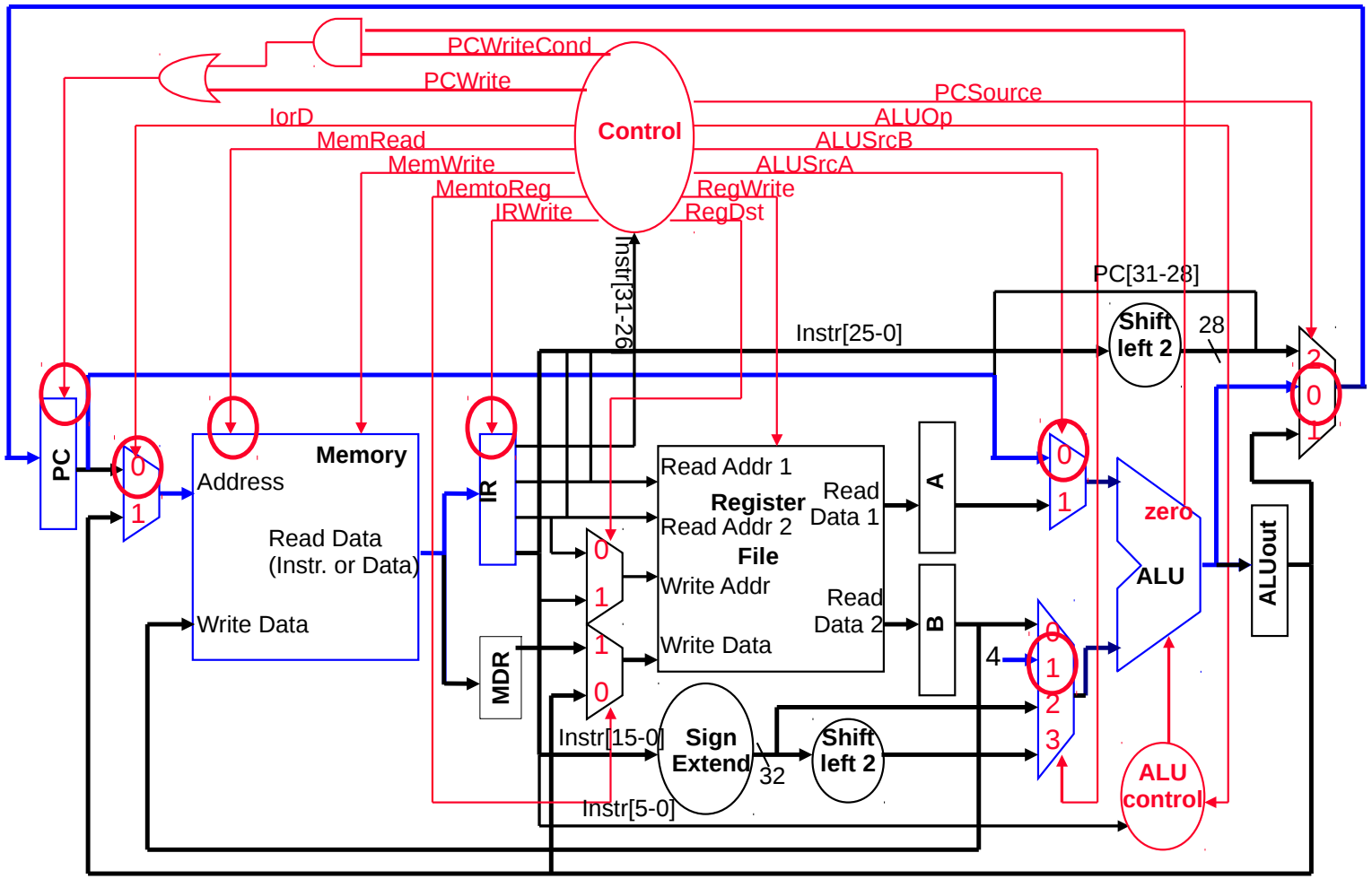
# The Five Execution Steps

1. Instruction fetch
   - Move the instruction from the instruction memory to the instruction register IR
2. Instruction decode and register fetch
   - Provide the register contents for the ALU
3. Execution, memory address computation or branch completion
4. Memory access or R-type instruction completion
5. Write back step

# Step 1: Instruction Fetch (1)

- Load instruction from memory
  IR = Memory [PC]
  - Set Read address mux (IorD) = 0 select instruction
  - Set MemRead = 1
  - Set IRWrite = 1
- Increment PC
  PC = PC + 4
  - Set ALUSrcA = 0 get operand from IR
  - Set ALUSrcB = 01 get operand '4'
  - Set ALUOp = 00 add
  - Allow storing new PC in PC register

# Step 1: Instruction Fetch (2)



# Step 2: Instruction Decode & Register Fetch (1)

- Switch registers to the output of the register block

```
A <= register [IR [25:21]]      rs
B <= register [IR [20:16]]      rt
```

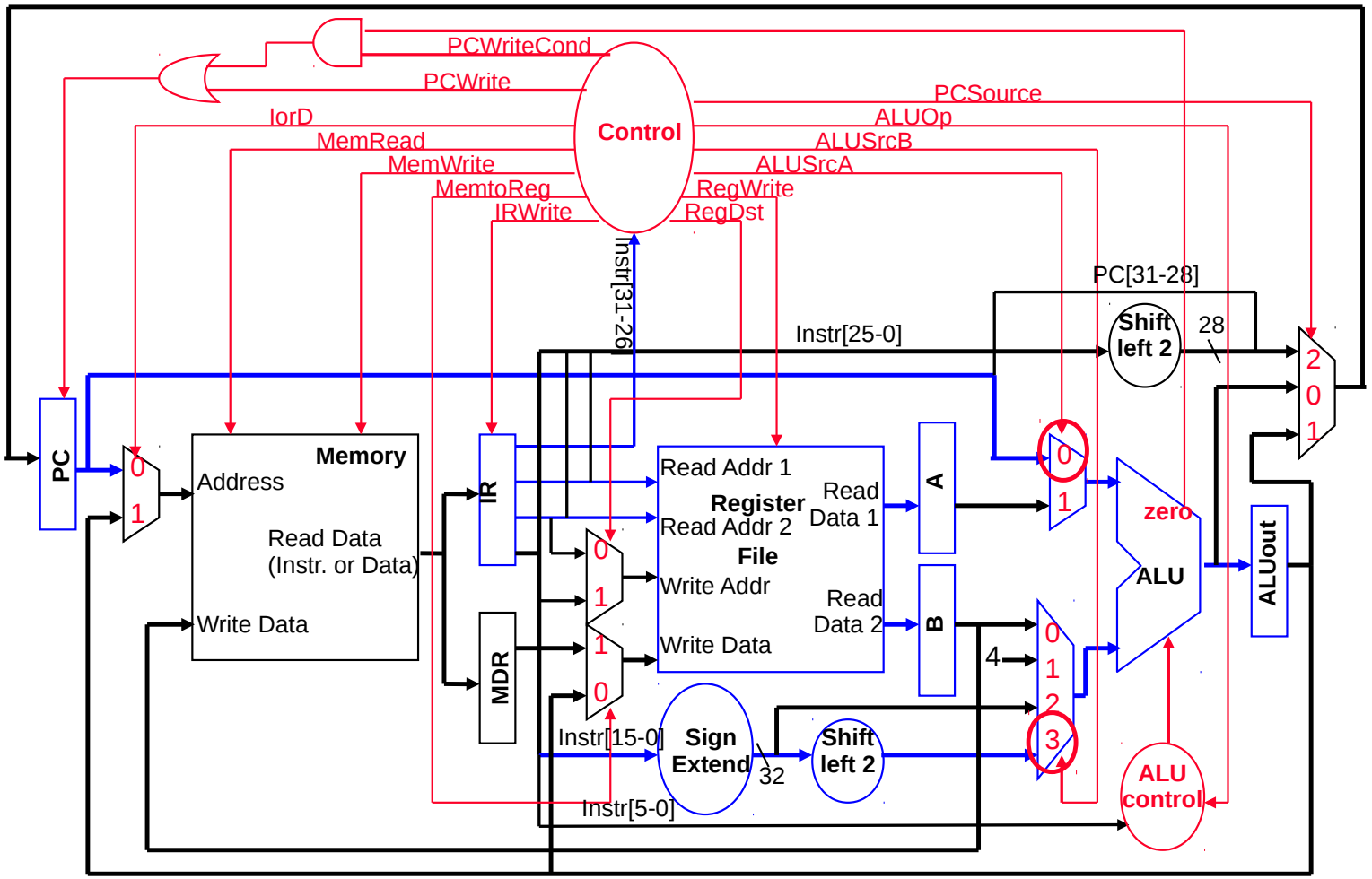
  - No signal setting required

- (Always) calculate the branch target address

```
ALUOut <= PC + (sign-ext. (IR [15:0]) << 2)
```

  - Value can just be ignored if instruction is not branch
  - Stored in the ALUOut register
  - Set ALUSrcB = 11
  - Set ALUOp = 00 add

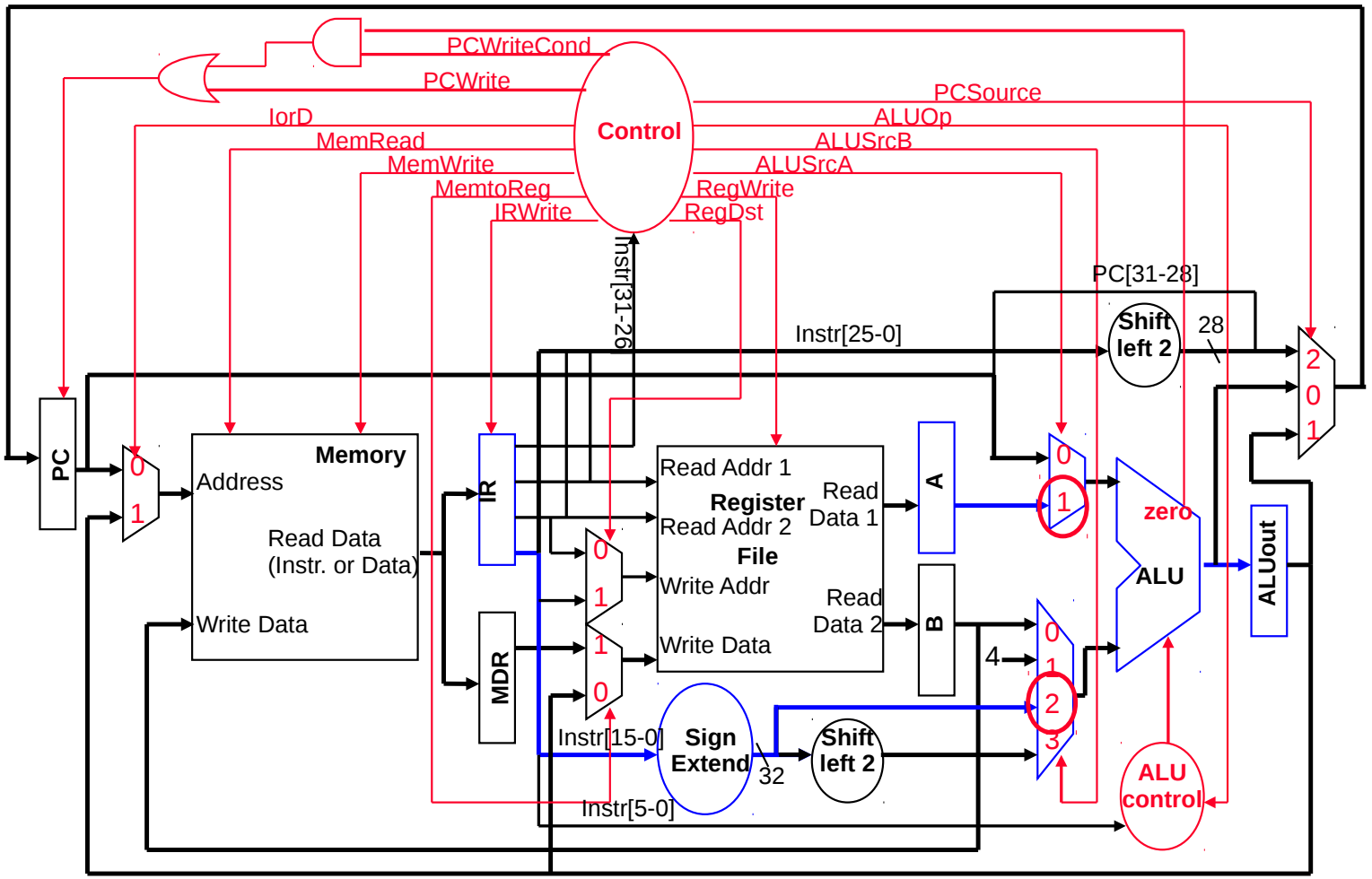
# Step 2: Instruction Decode & Register Fetch (2)



# Step 3: Execution, Memory Address Computation or Branch Completion

- First cycle where step depends on the instruction
- Selection performed by interpretation of the op + function field of the instruction
- Memory reference
  - calculate address
    `ALUOut <= A + sign-extend(IR[15:0])`
  - Set ALUSrcA = 1 get operand from A
  - Set ALUSrcB = 10 get operand from sign extension unit
  - Set ALUOp = 00 add

# Step 3: Memory Reference



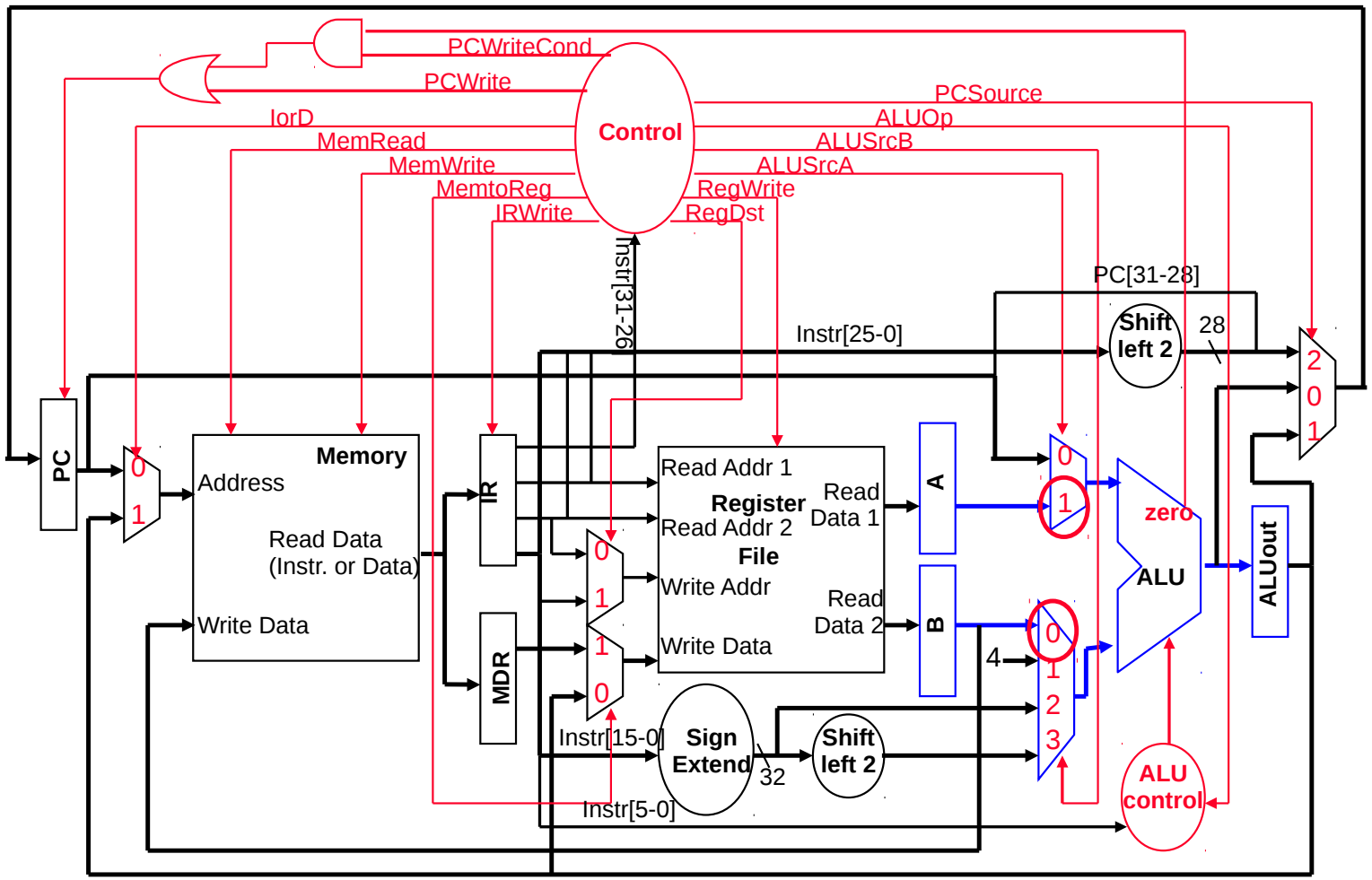
# Step 3: Execution, Memory Address Computation or Branch Completion

Arithmetic-logical instruction (R-type):

`ALUOut = A op B`

- Set ALUSrcA = 1 get operand from A
- Set ALUSrcB = 00 get operand from B
- Set ALUOp = 10 code from IR

# Step 3: Arithmetic-Logical Instruction



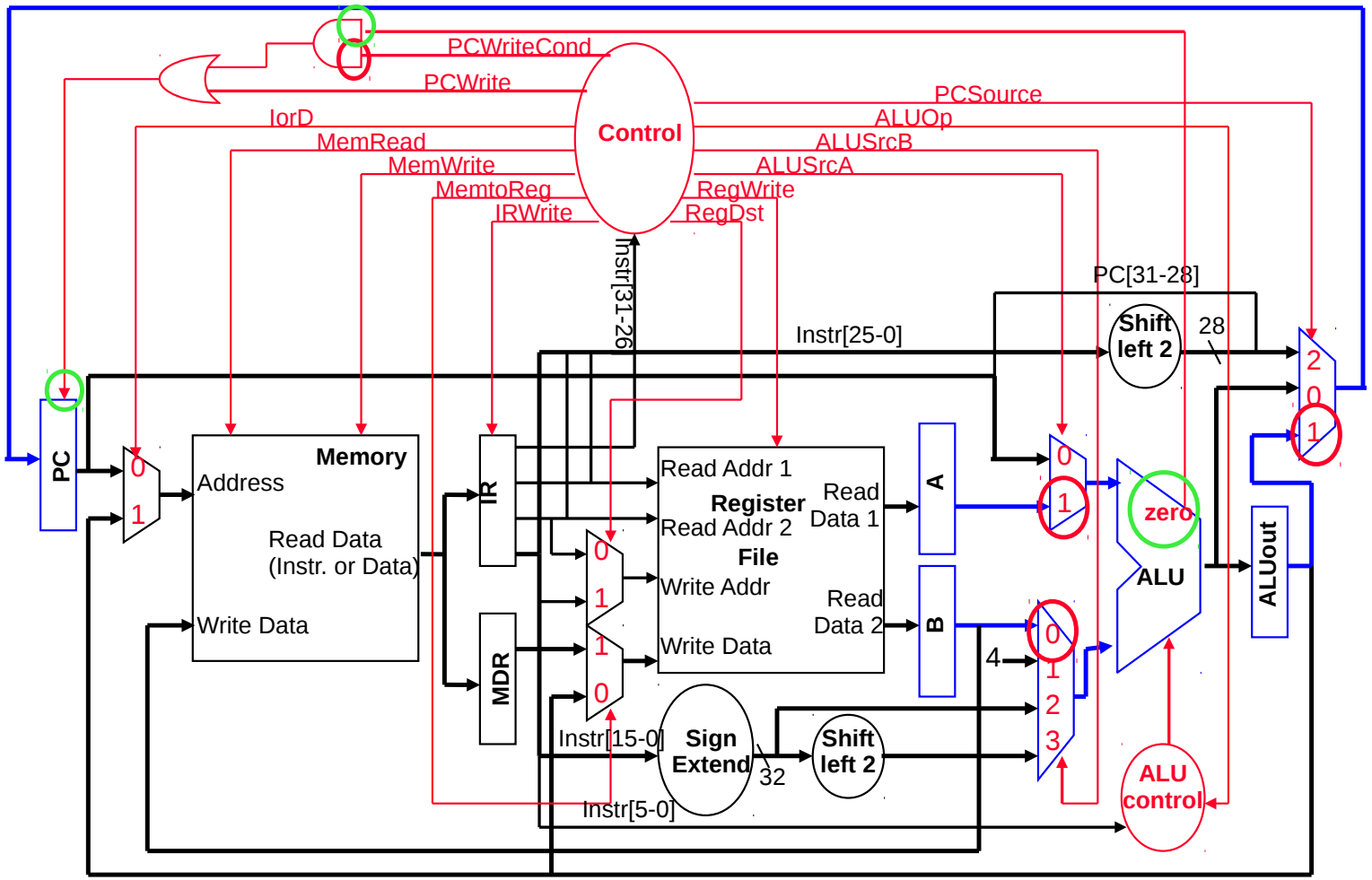
# Step 3: Execution, Memory Address Computation or Branch Completion

Branch:

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

- Set ALUSrcA = 1 get operand from A
- Set ALUSrcB = 00 get operand from B
- Set ALUOp = 01 subtraction
- Write ALUOut to PC register

# Step 3: Branch

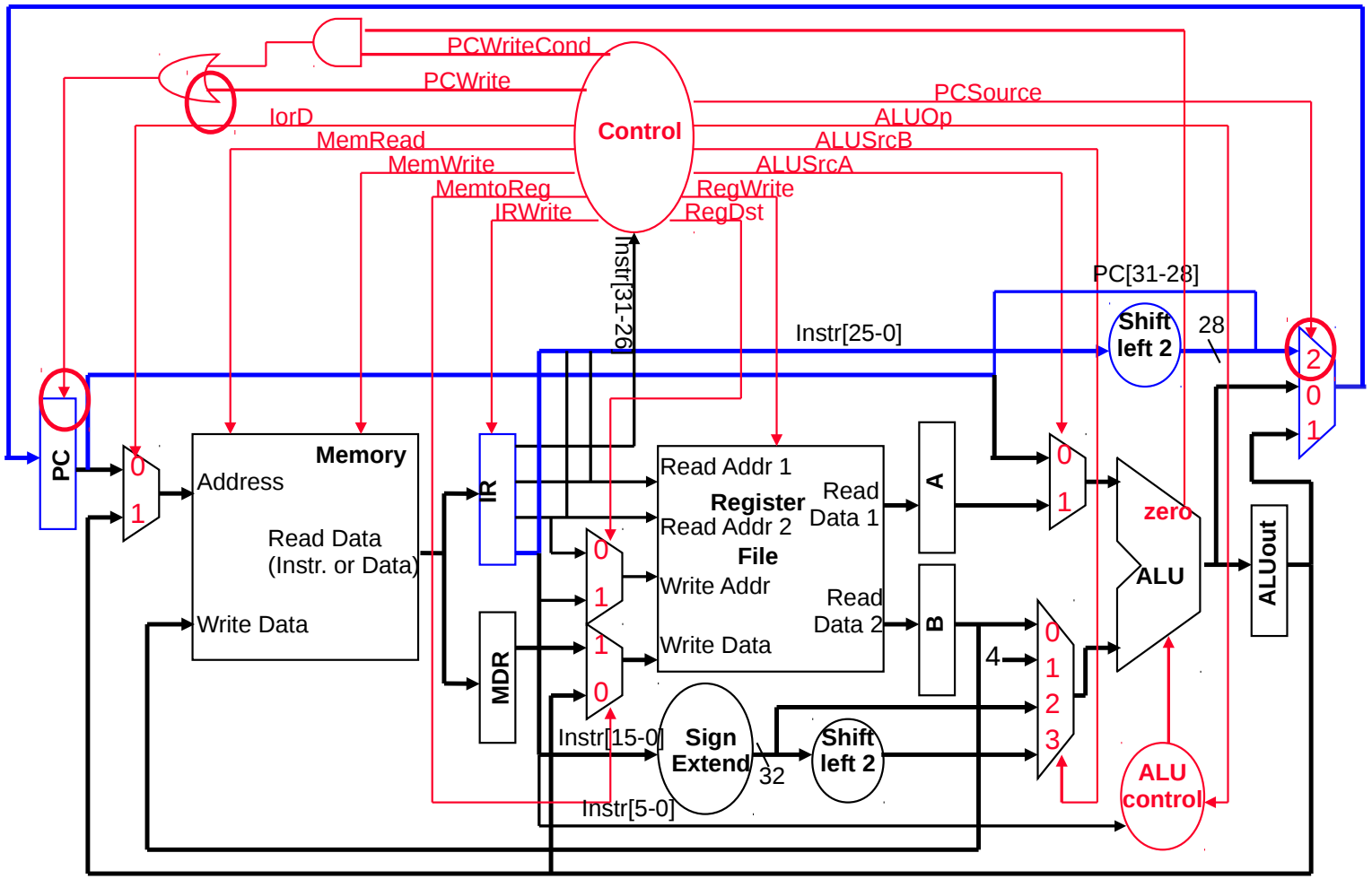


# Step 3: Execution, Memory Address Computation or Branch Completion

Jump:

```
PC <= {PC [31:28], (IR[25:0] << 2)}
```

# Step 3: Jump

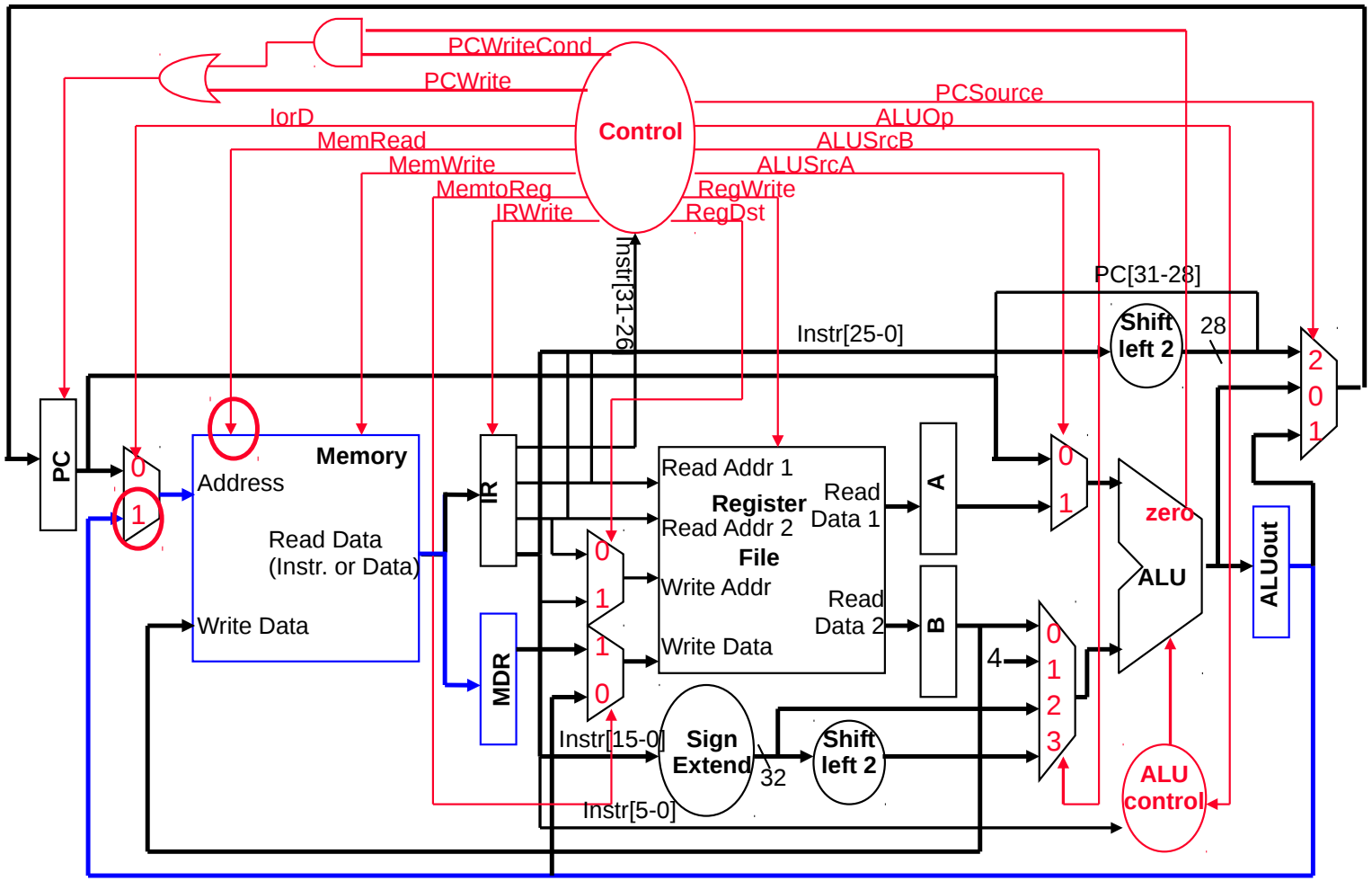

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Control
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instr[31-26]
PC[31-28]
Instr[25-0]
Shift left 2
28
PC
Memory
Address
Read Data (Instr. or Data)
Write Data
IR
MDR
Read Addr 1
Register File
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
A
B
4
zero
ALU
ALUout
Instr[15-0]
Sign Extend
32
Shift left 2
Instr[5-0]
ALU control

# Step 4: Memory Access or R-type Instruction Completion

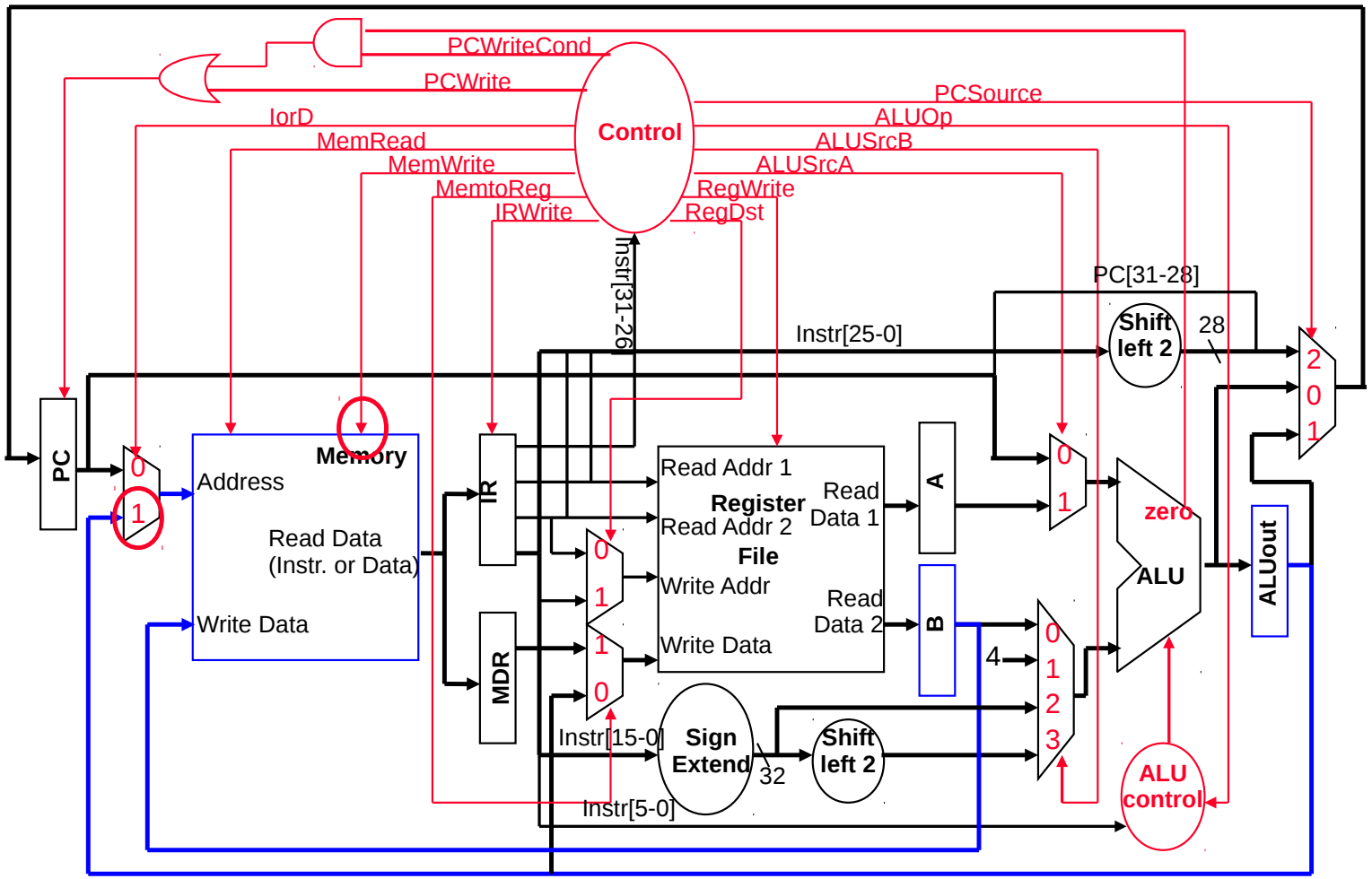
Memory reference:

- ALU controls must remain stable
- Set IorD $= 1$ address from ALU
- load from memory
  `MDR <= memory[ALUOut]`
  - Set MemRead $= 1$
- store to memory
  `memory[ALUOut] <= B`
  - Set MemWrite $= 1$

# Step 4: Memory Reference (load word)



# Step 4: Memory Reference (save word)



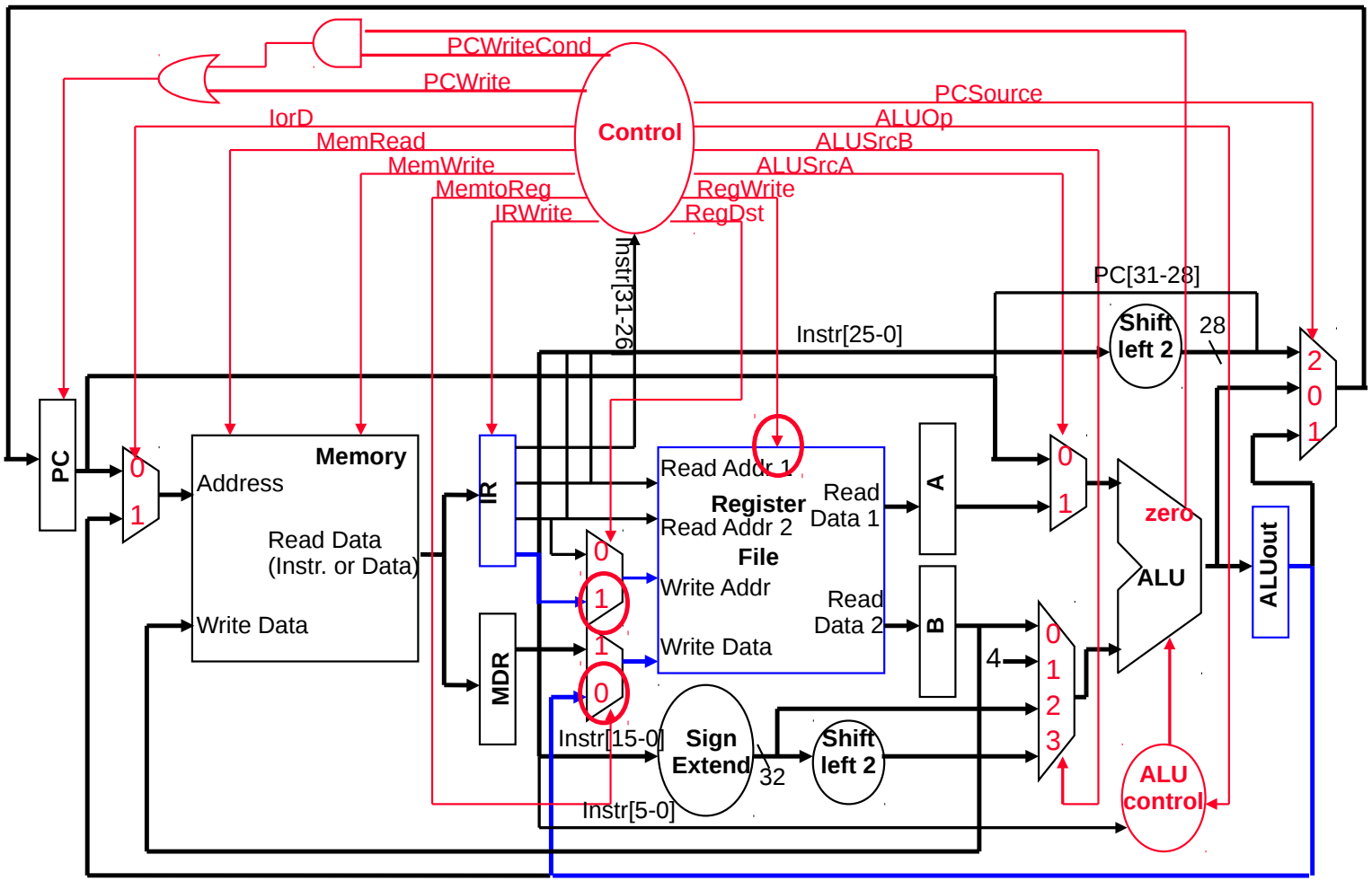
# Step 4: Memory Access or R-type Instruction Completion

Arithmetic-logical instruction completion:

```
Register[IR[15:11]] <= ALUOut
```

- Set RegDst = 1 Select write register
- Set RegWrite = 1 Allow write operation
- Set MemToReg = 0 Select ALU data
- ALUOp, ALUSrcA, ALUSrcB = constant

# Step 4: Arithmetic-Logical Instruction Completion



# Step 5: Write Back

Write data from memory to the register:

```
Register[IR[20:16]] <= MDR
```

- Set RegDst = 0 Select write rt as target register
- Set RegWrite = 1 Allow write operation
- Set MemToReg = 1 Select Memory data
- ALUOp, ALUSrcA, ALUSrcB = constant

# Step 5: Memory Reference (load word)