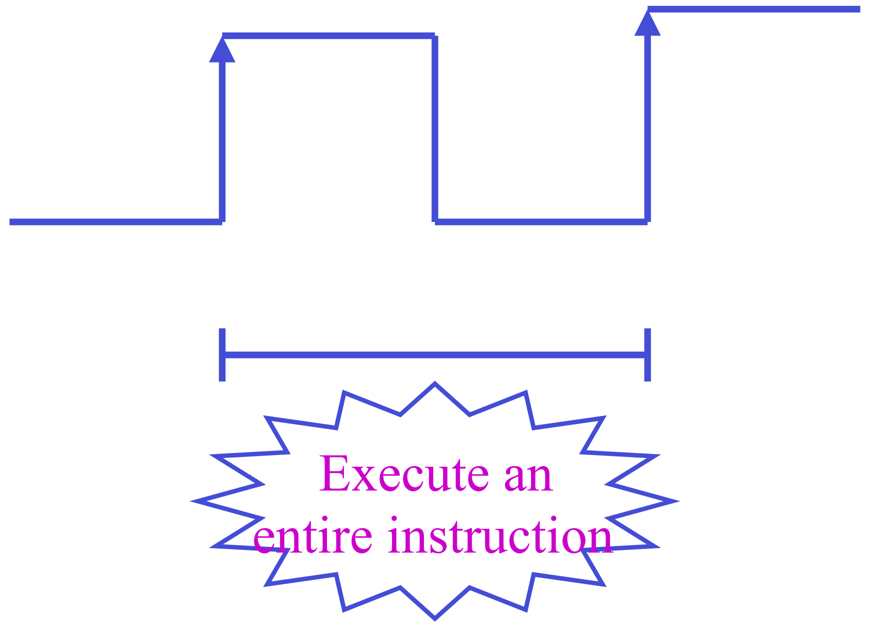
Single Cycle CPU

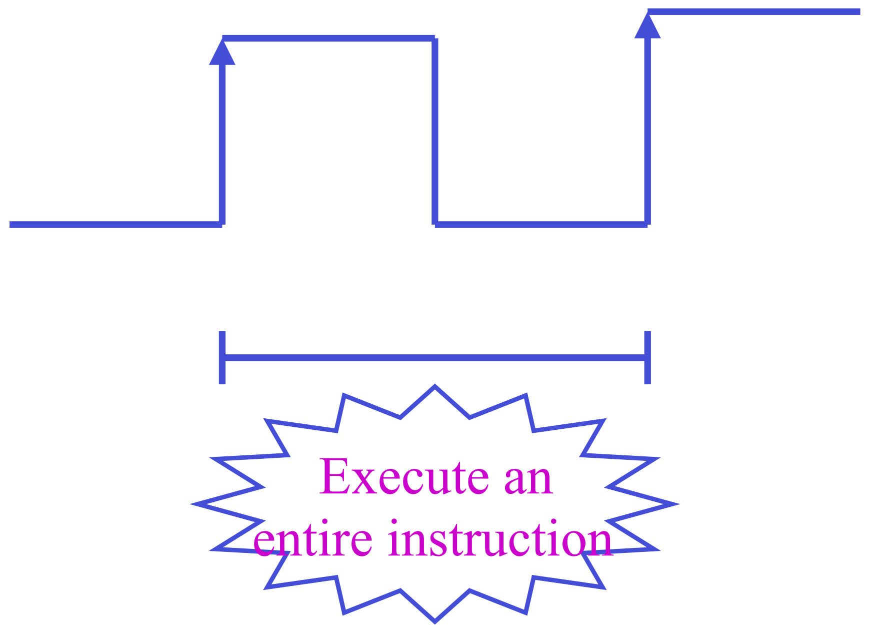
Jason Mars

The Big Picture: The Performance Perspective



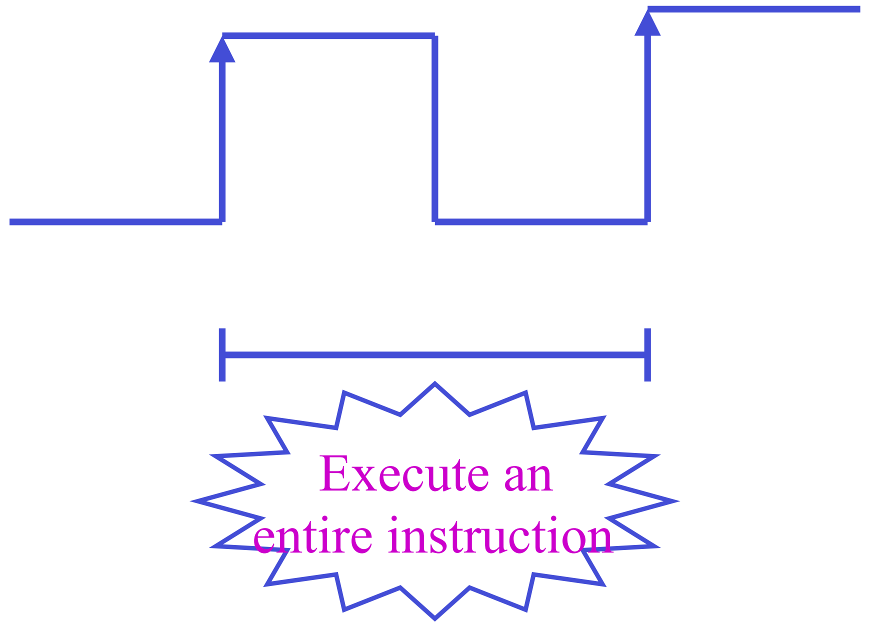
The Big Picture: The Performance Perspective

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction



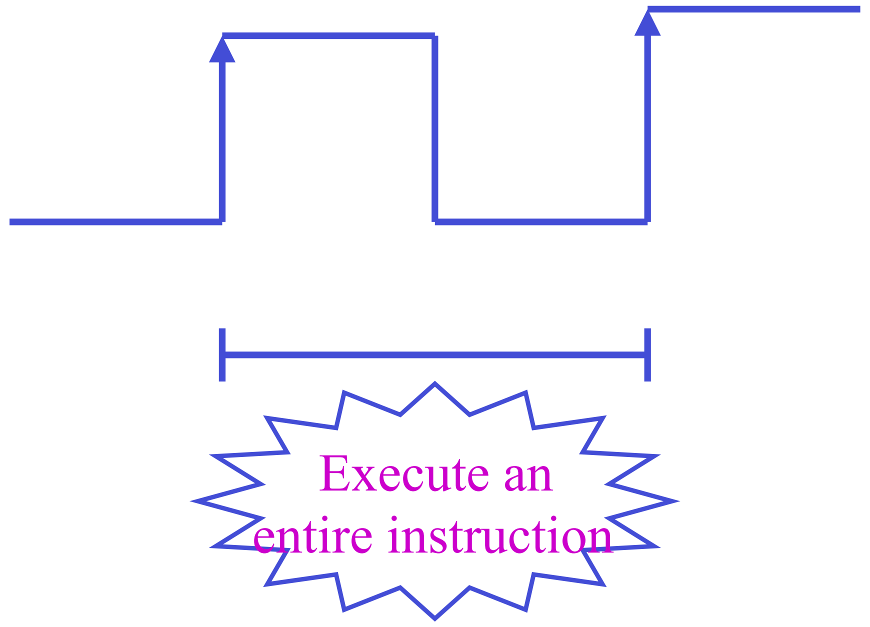
The Big Picture: The Performance Perspective

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Starting today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time



The Big Picture: The Performance Perspective

- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Starting today:
  - Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time
- ET = Insts * CPI * Cyc Time



Processor Datapath and Control

Processor Datapath and Control

- We're ready to look at an implementation of the MIPS simplified to contain only:
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add, sub, and, or, slt
  - control flow instructions: beq

Processor Datapath and Control

- We're ready to look at an implementation of the MIPS simplified to contain only:
    - memory-reference instructions: lw, sw
    - arithmetic-logical instructions: add, sub, and, or, slt
    - control flow instructions: beq
- Generic Implementation:
    - use the program counter (PC) to supply instruction address
    - get the instruction from memory
    - read registers
    - use the instruction to decide exactly what to do

Processor Datapath and Control

- We're ready to look at an implementation of the MIPS simplified to contain only:
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add, sub, and, or, slt
  - control flow instructions: beq
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers
  - memory-reference? arithmetic? control flow?

Review: MIPS Instruction Formats

- All instructions 32-bits long
- 3 Formats:

R-Type

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct

I-Type

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

J-Type

6 bits	26 bits
opcode	target

The MIPS Subset

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

The MIPS Subset

- R-Type
  - *add rd, rs, rt*
  - sub, and, or, slt

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

The MIPS Subset

- R-Type
  - *add rd, rs, rt*
  - sub, and, or, slt

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct

- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

The MIPS Subset

- R-Type
  - *add rd, rs, rt*
  - sub, and, or, slt

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct

- LOAD and STORE
  - lw rt, rs, imm16
  - sw rt, rs, imm16

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

- BRANCH:
  - beq rs, rt, imm16

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset

Basic Steps of Execution

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?
- Memory access
  - Where is my data?

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?
- Decode
  - What’s the incoming instruction?
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?
- Memory access
  - Where is my data?
- Write back results to registers
  - Where to write?

Basic Steps of Execution

- Instruction Fetch
    - Where is the instruction?
- Decode
    - What’s the incoming instruction?
    - Where are the operands in an instruction?
- Execution: ALU
    - What is the function that ALU should perform?
- Memory access
    - Where is my data?
- Write back results to registers
    - Where to write?
- Determine the next PC

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Instruction memory address: PC

- Decode
  - What's the incoming instruction?
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?
- Memory access
  - Where is my data?
- Write back results to registers
  - Where to write?
- Determine the next PC

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Instruction memory address: PC

- Decode
  - What's the incoming instruction?
  - Where are the operands in an instruction?

register file

- Execution: ALU
  - What is the function that ALU should perform?
- Memory access
  - Where is my data?
- Write back results to registers
  - Where to write?
- Determine the next PC

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Instruction memory
address: PC

- Decode
  - What's the incoming instruction?
  - Where are the operands in an instruction?

register file

- Execution: ALU
  - What is the function that ALU should perform?

ALU

- Memory access
  - Where is my data?
- Write back results to registers
  - Where to write?
- Determine the next PC

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Instruction memory
address: PC

- Decode
  - What's the incoming instruction?
  - Where are the operands in an instruction?

register file

- Execution: ALU
  - What is the function that ALU should perform?

ALU

- Memory access
  - Where is my data?

Data memory
address: effective address

- Write back results to registers
  - Where to write?
- Determine the next PC

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Instruction memory
address: PC

- Decode
  - What's the incoming instruction?
  - Where are the operands in an instruction?

register file

- Execution: ALU
  - What is the function that ALU should perform?

ALU

- Memory access
  - Where is my data?

Data memory
address: effective address

- Write back results to registers
  - Where to write?

register file

- Determine the next PC

Basic Steps of Execution

- Instruction Fetch
  - Where is the instruction?

Instruction memory

address: PC

- Decode
  - What's the incoming instruction?

    register file
  - Where are the operands in an instruction?
- Execution: ALU
  - What is the function that ALU should perform?

    ALU
- Memory access
  - Where is my data?

Data memory

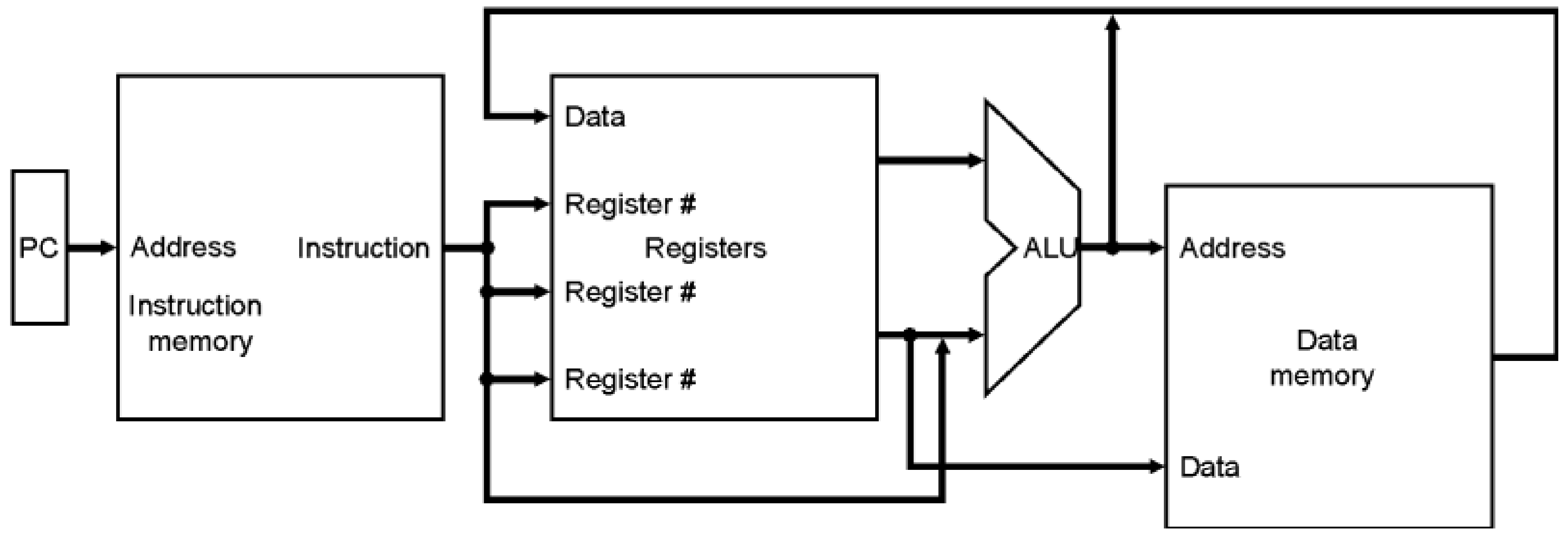
address: effective address

- Write back results to registers
  - Where to write?

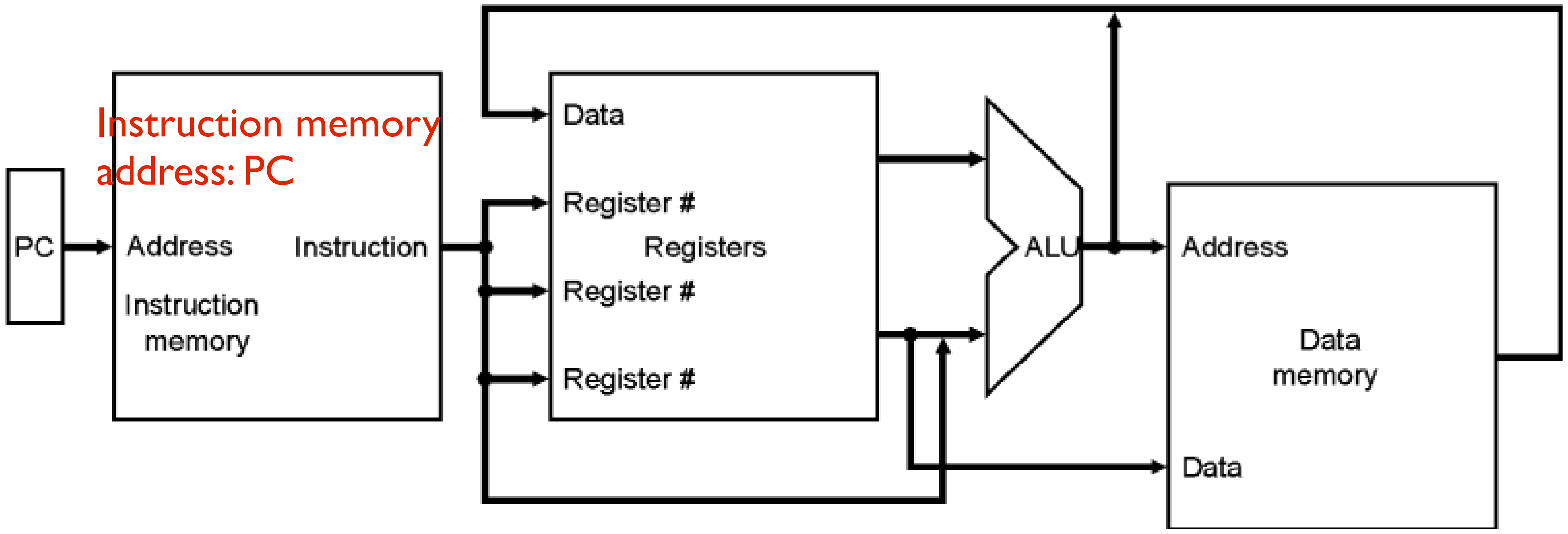
    register file
- Determine the next PC

  program counter

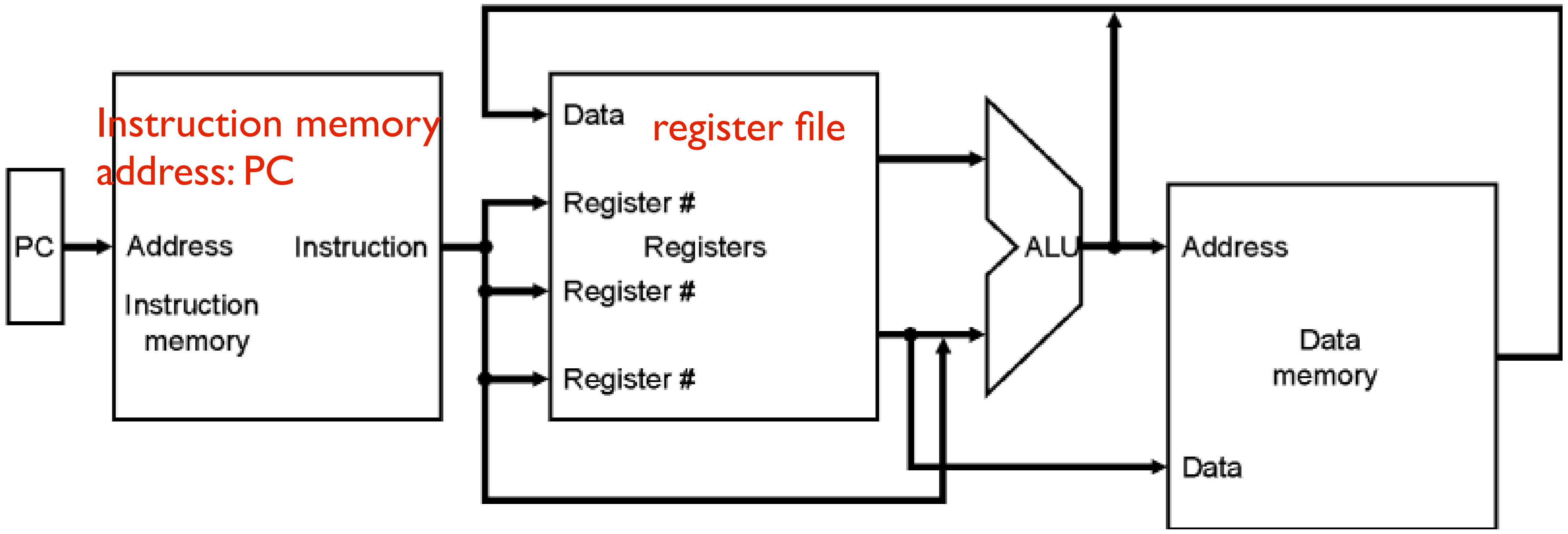
Where We're Going...



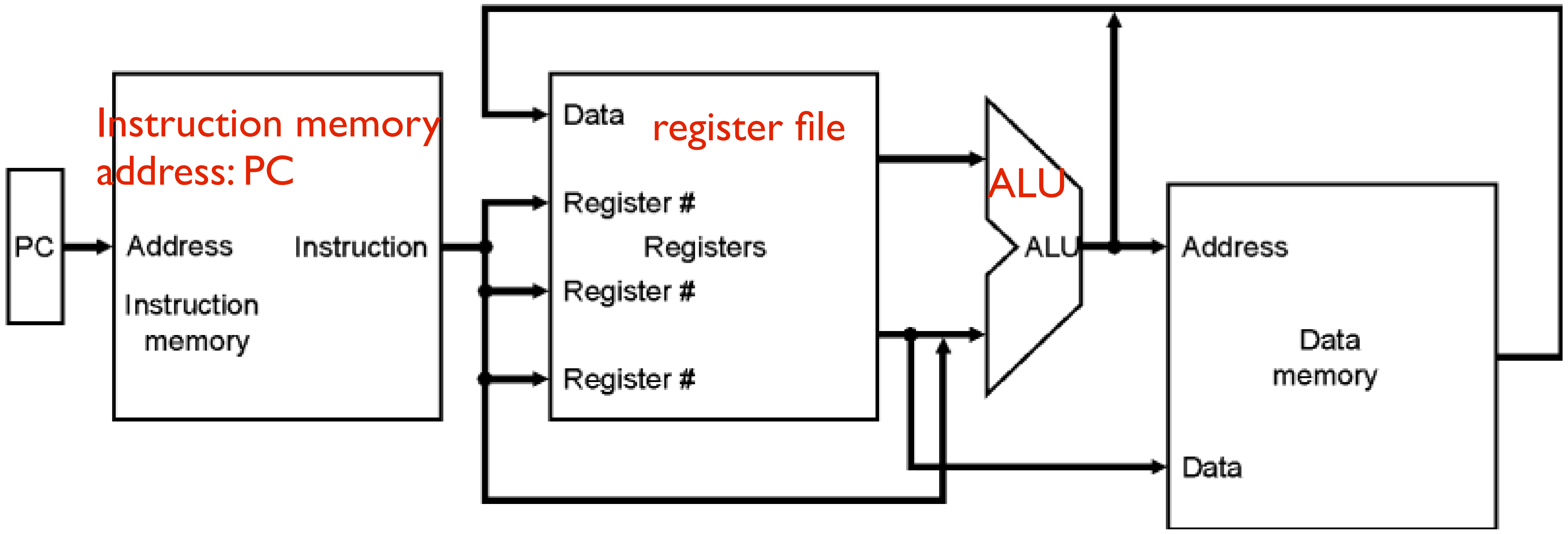
Where We're Going...



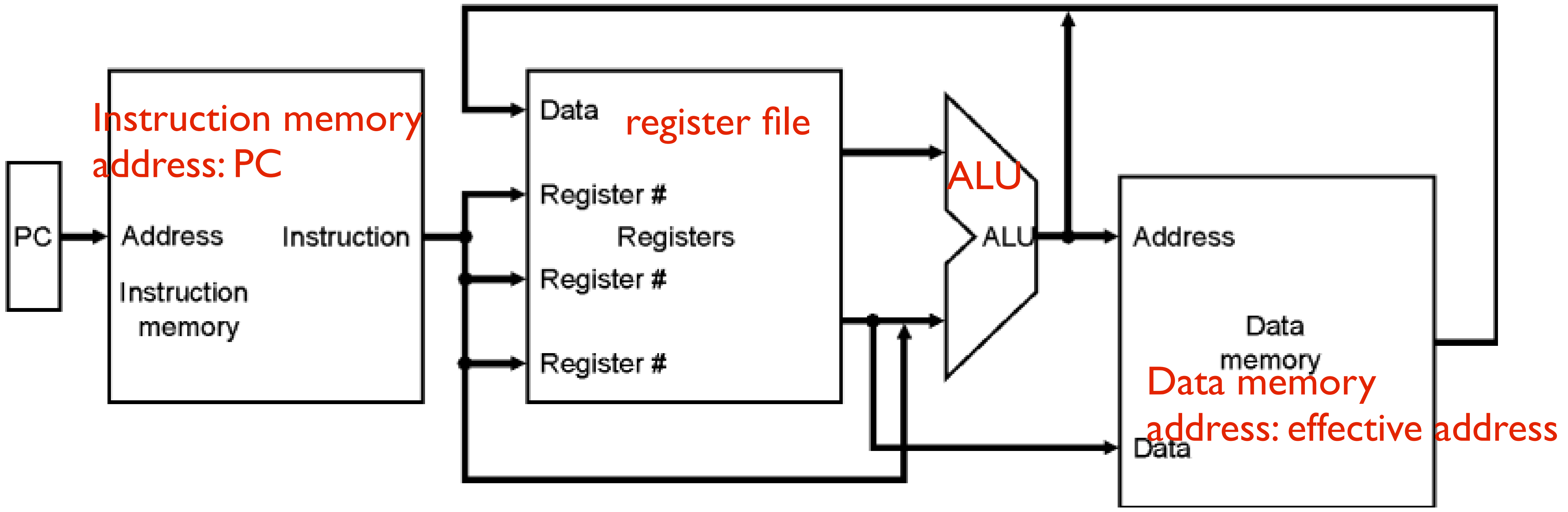
Where We're Going...



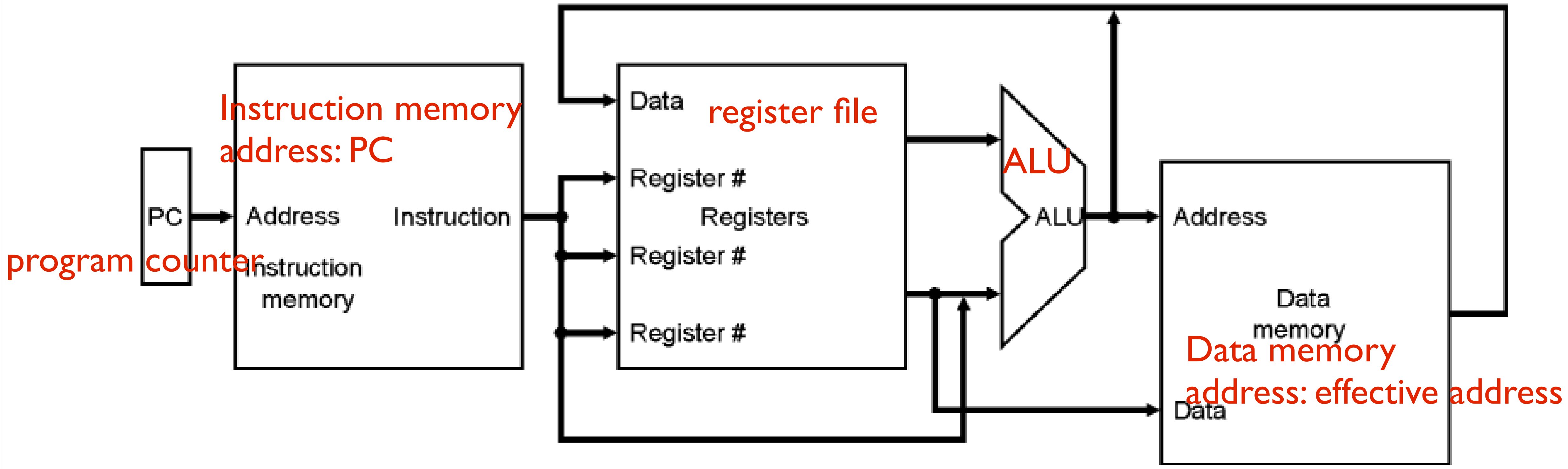
Where We're Going...



Where We're Going...

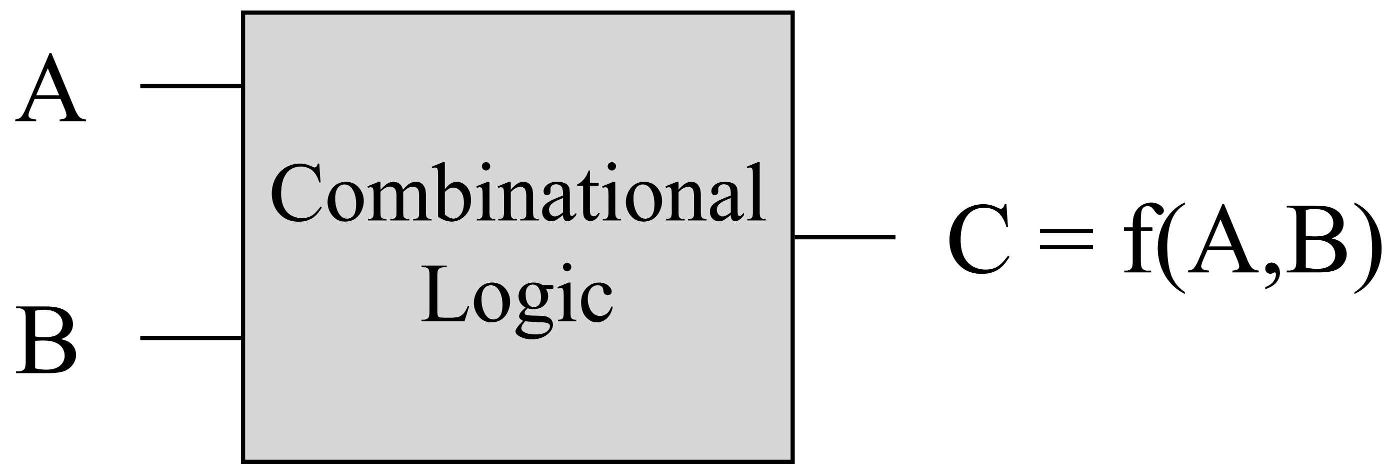


Where We're Going...

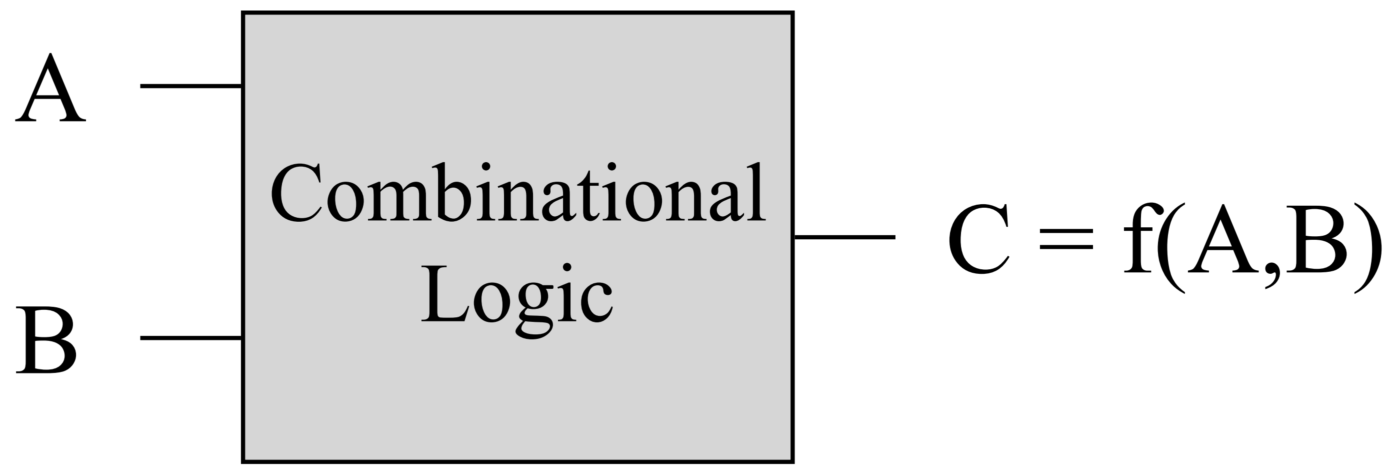


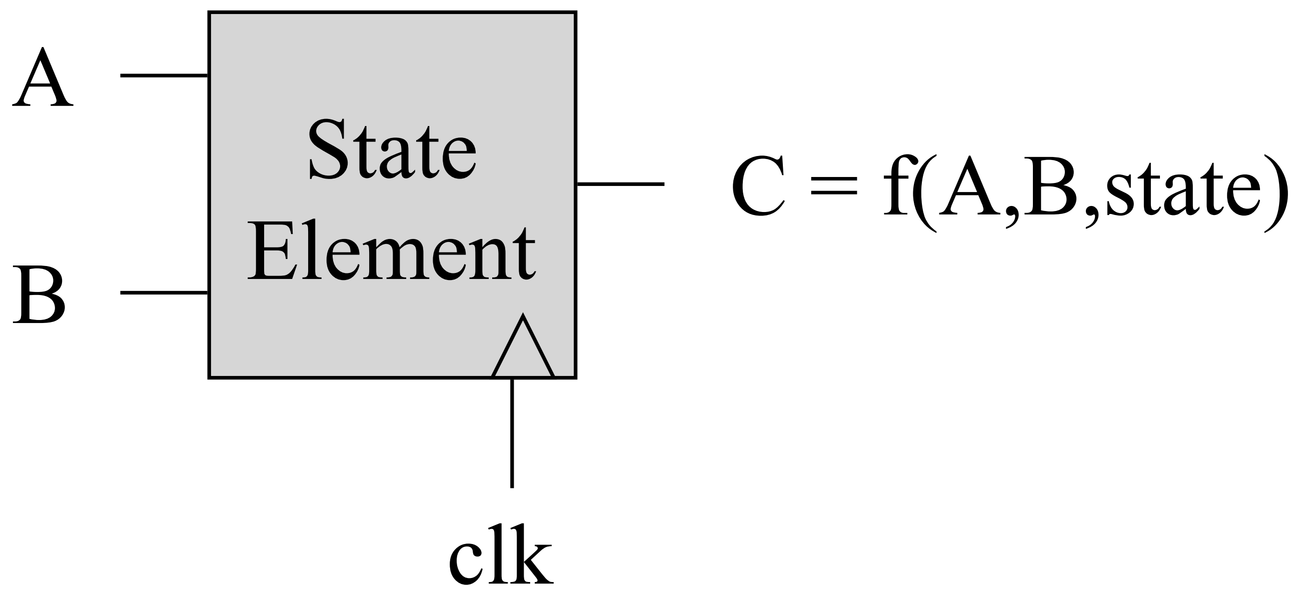
Review: Two Type of Logical Components

Review: Two Type of Logical Components

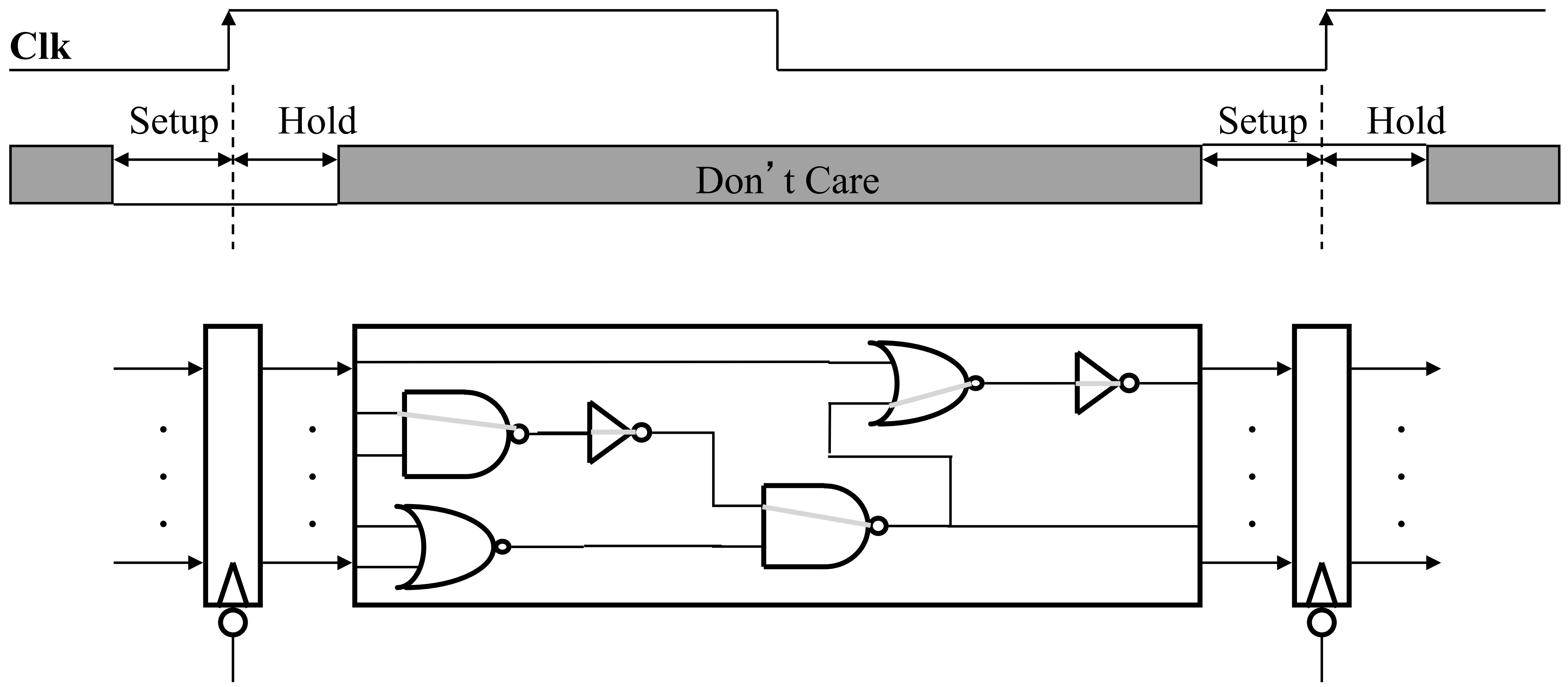


Review: Two Type of Logical Components





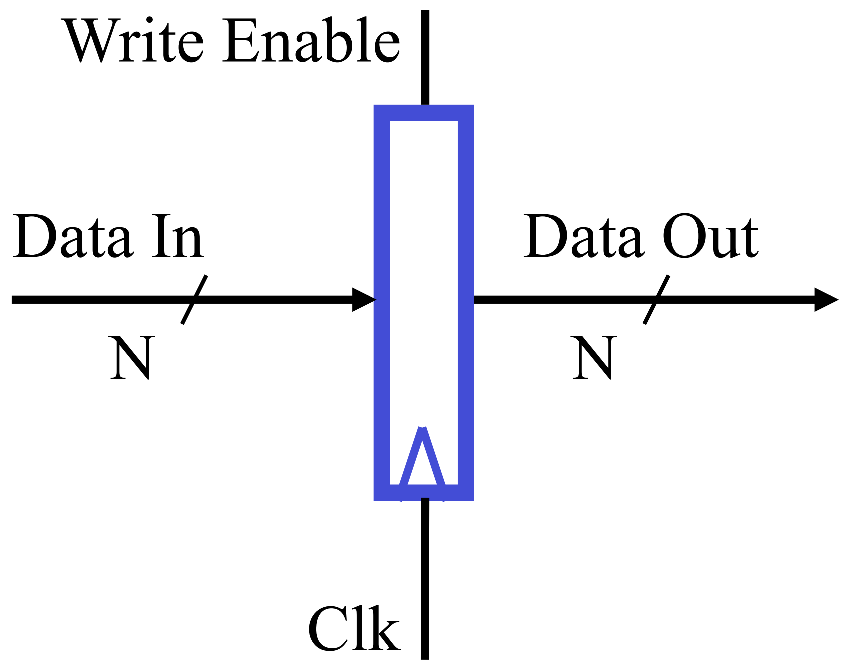
Clocking Methodology



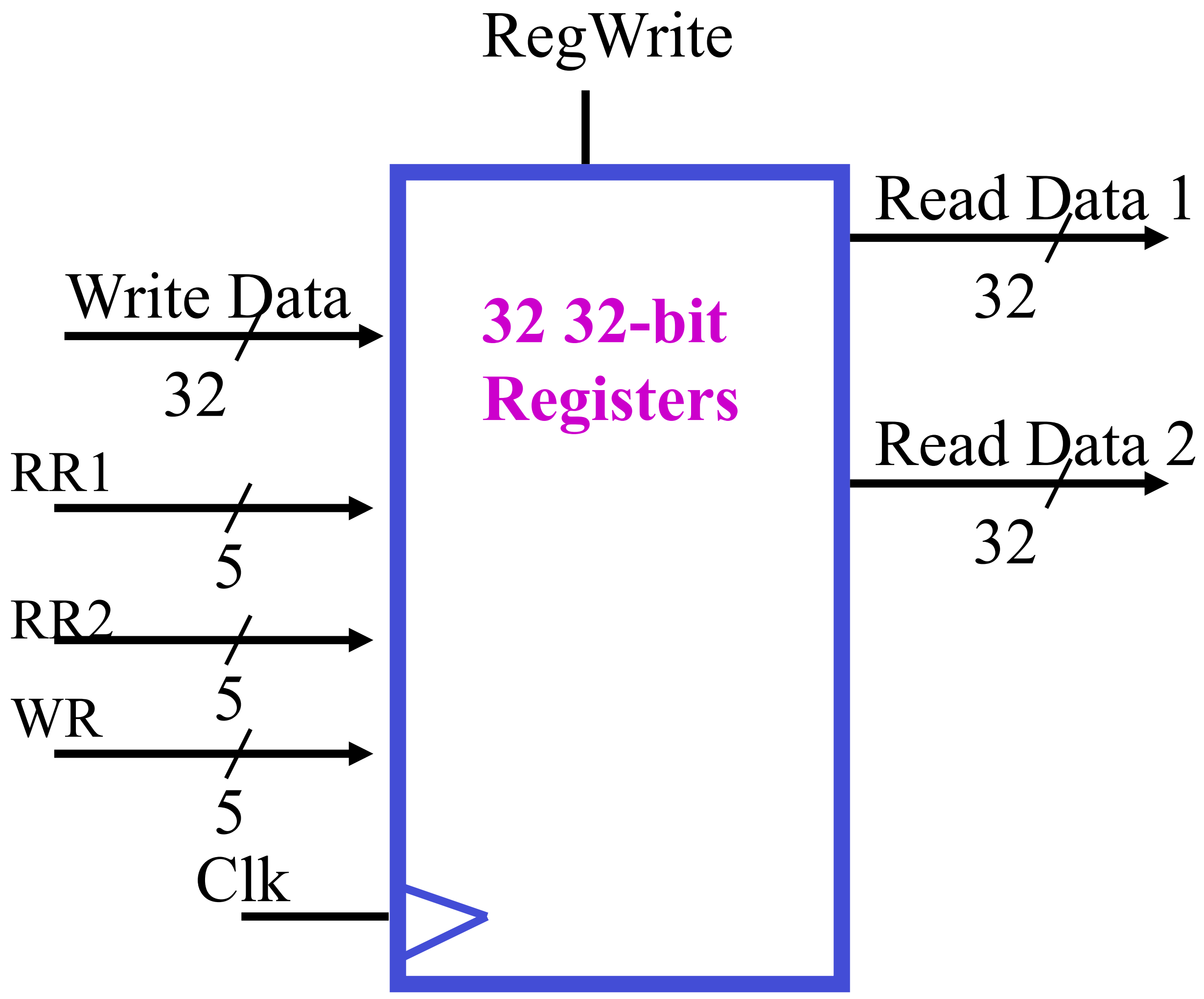
- All storage elements are clocked by the same clock edge

Storage Element: The Register

- Register
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
- Write Enable:
  - 0: Data Out will not change
  - 1: Data Out will become Data In (on the clock edge)

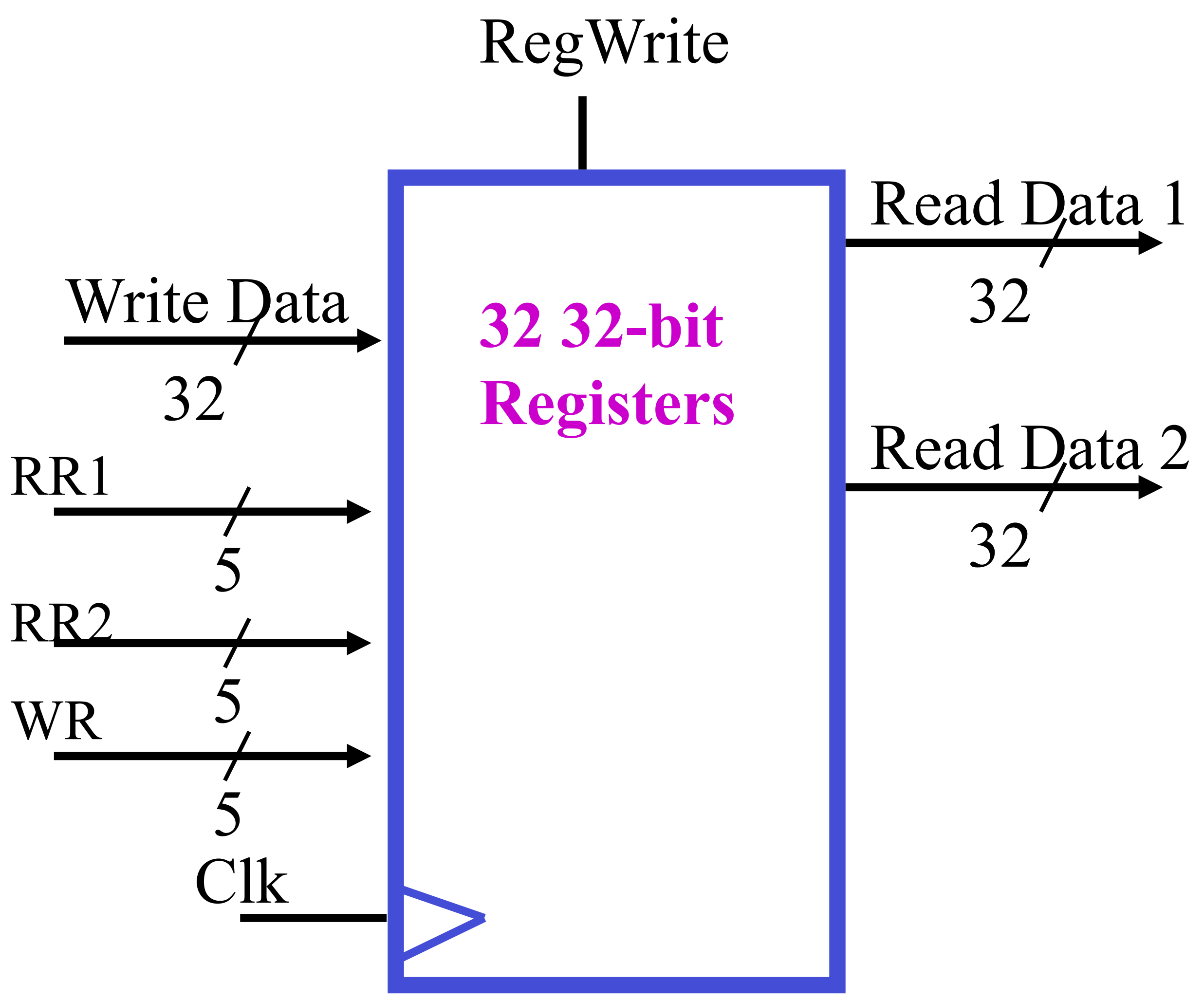


Storage Element: Register File



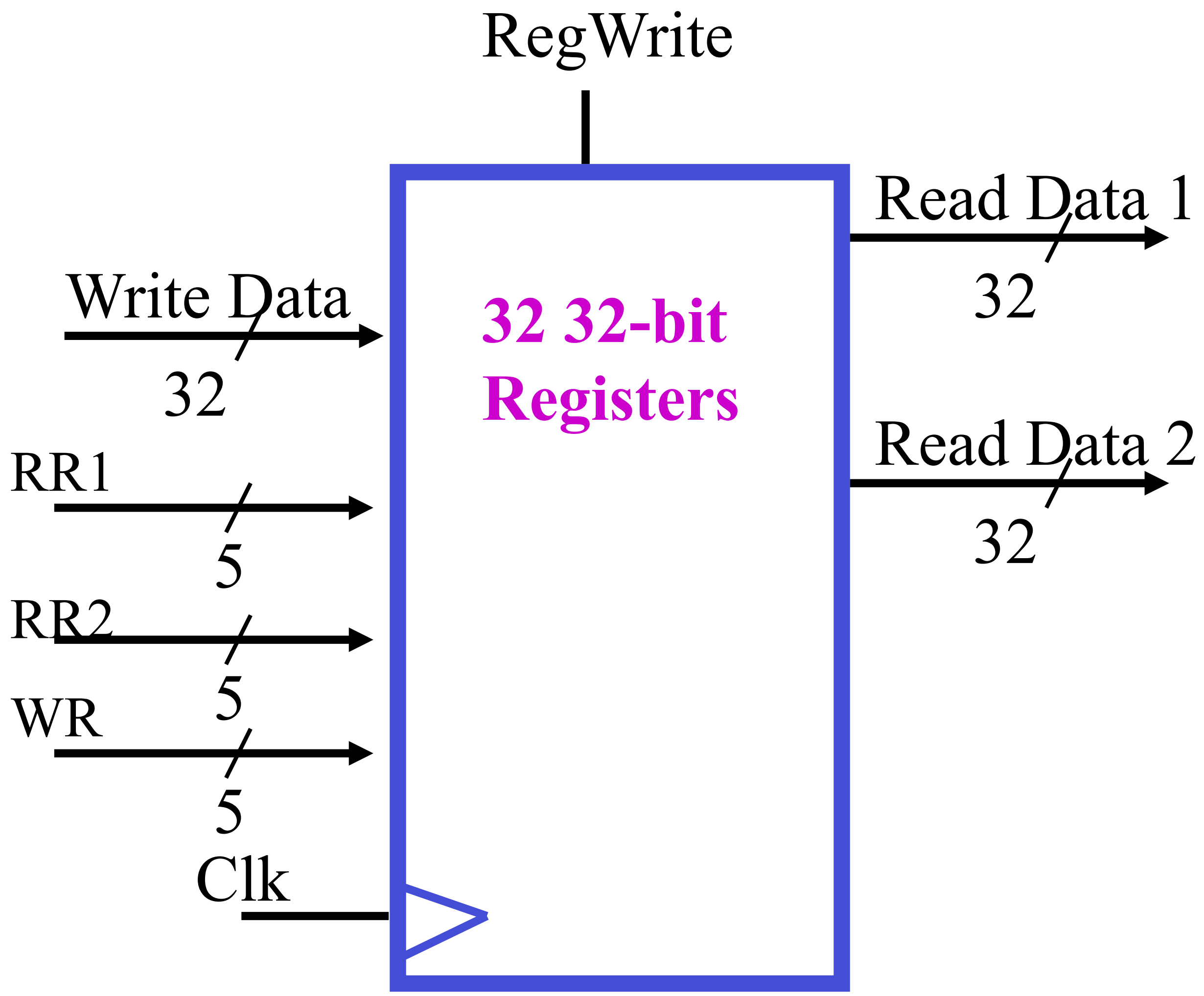
Storage Element: Register File

- Register File consists of (32) registers:



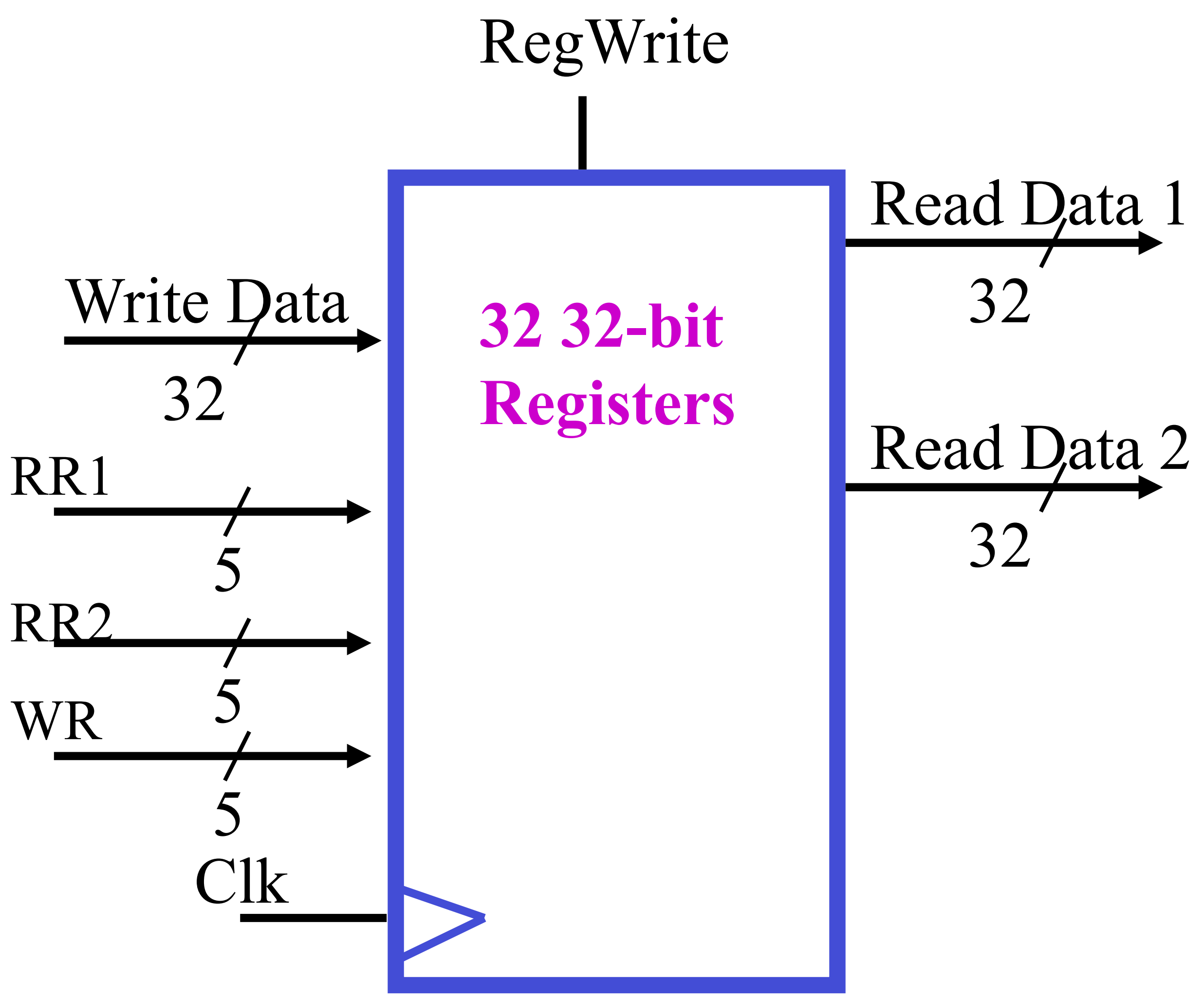
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses



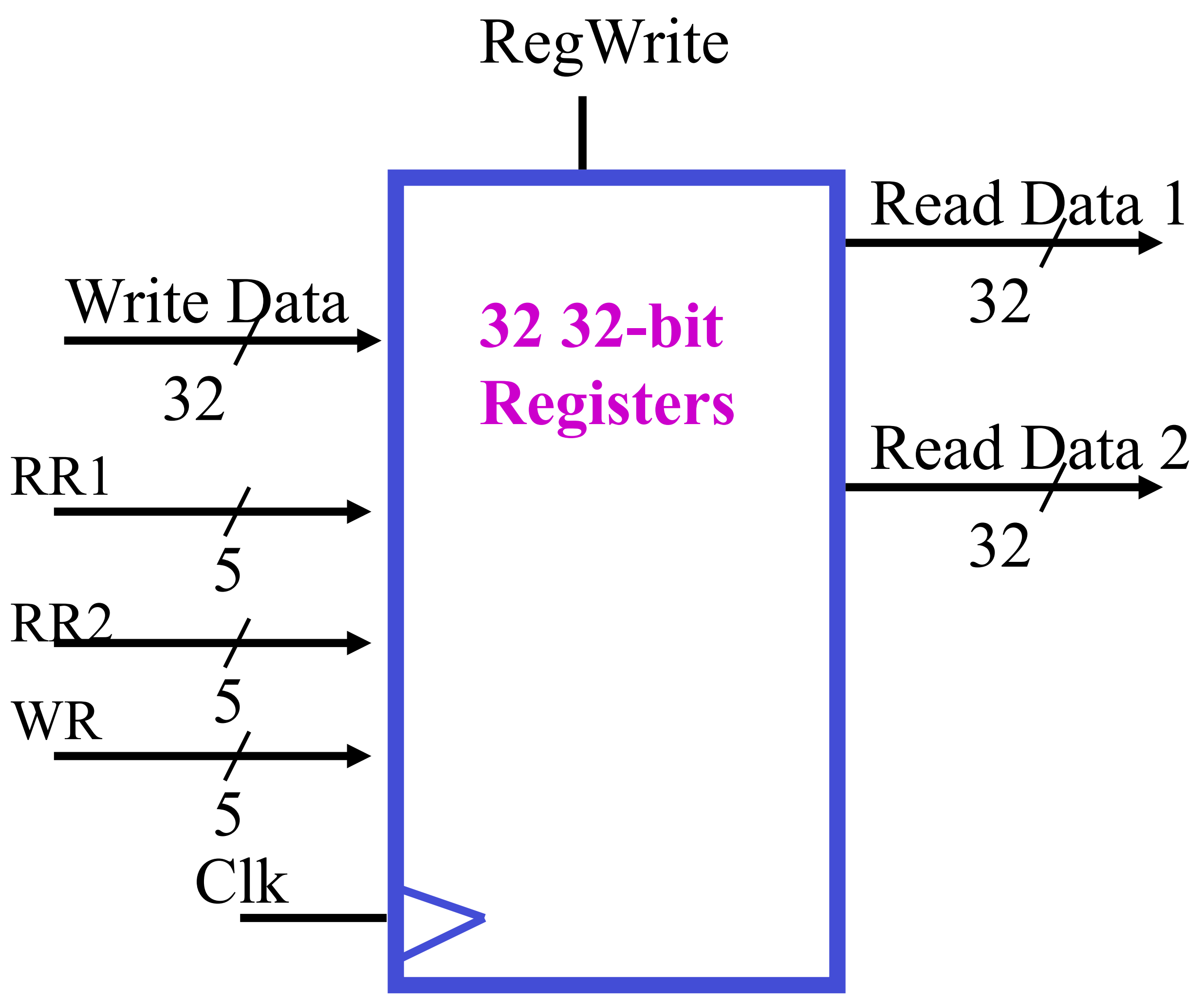
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus



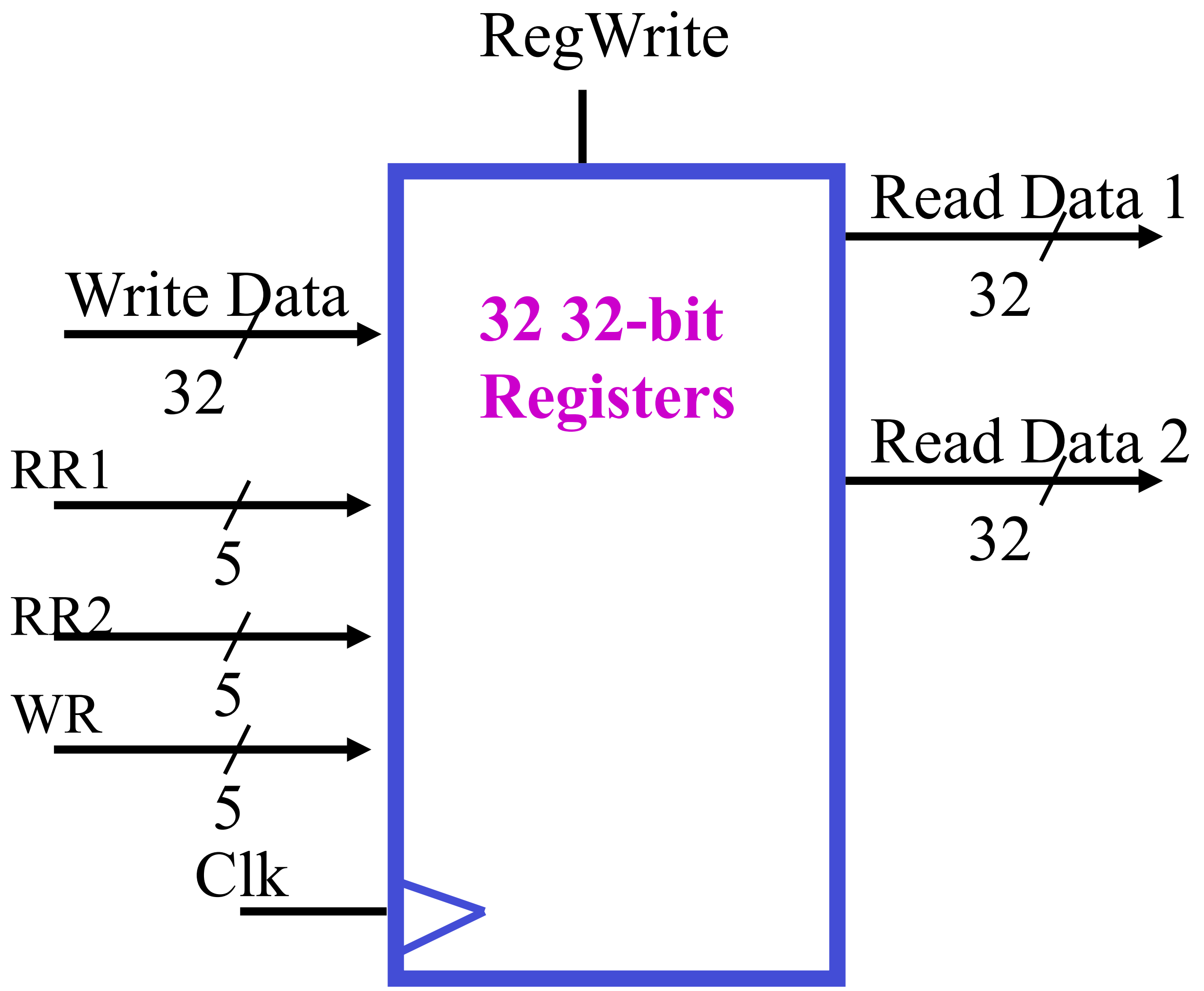
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:



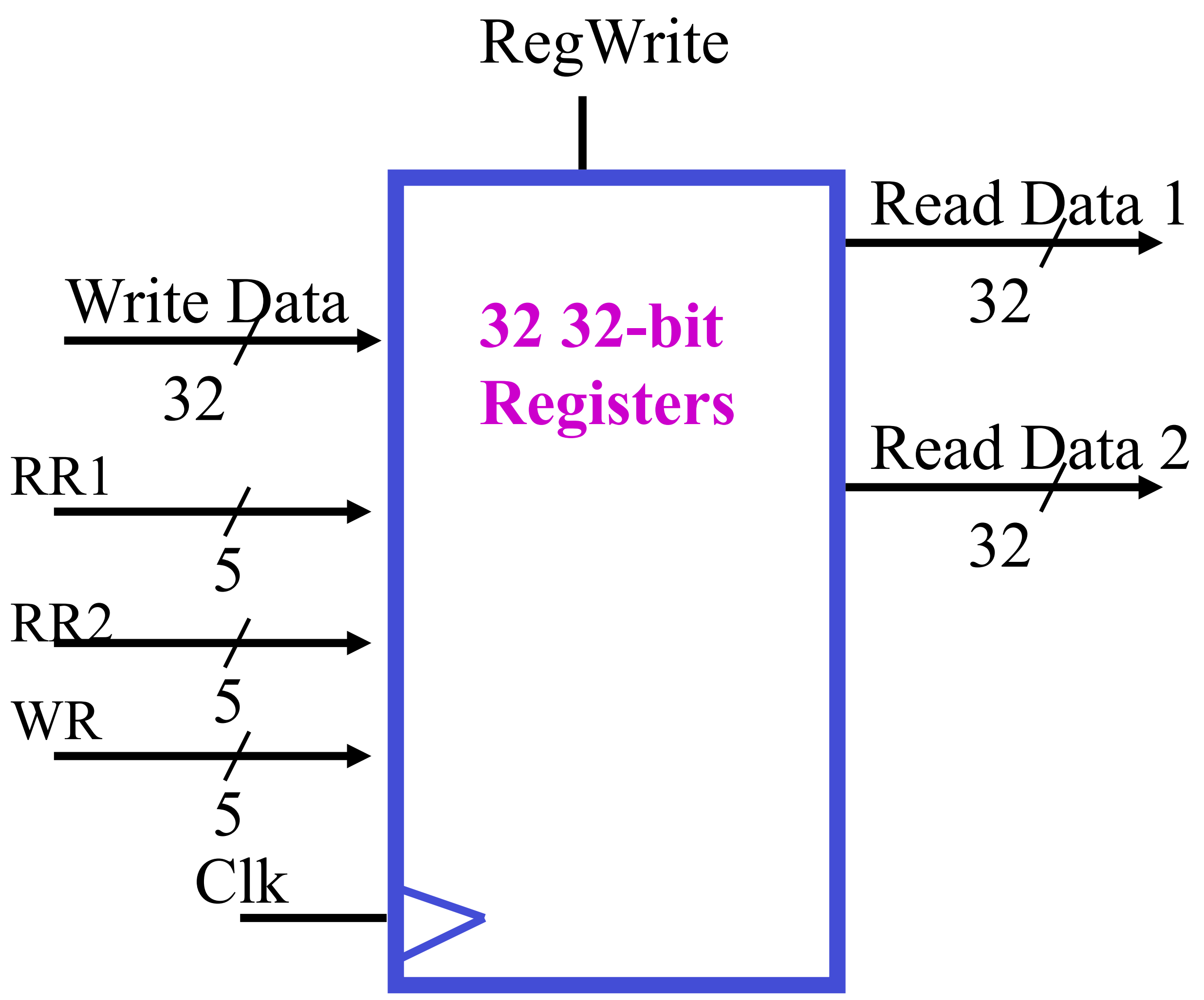
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:
  - RR1 selects the register to put on bus "Read Data 1"



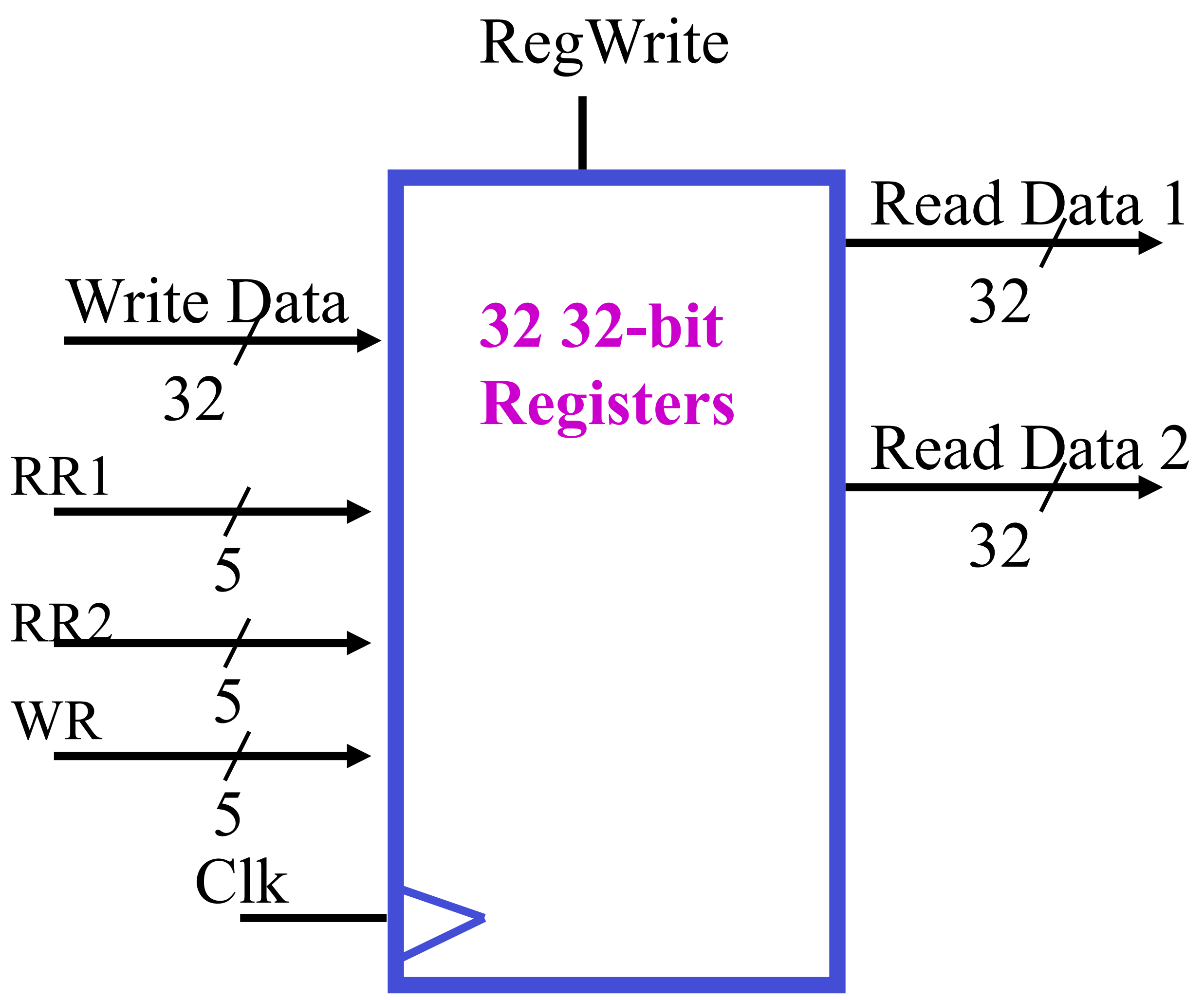
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:
  - RR1 selects the register to put on bus “Read Data 1”
  - RR2 selects the register to put on bus “Read Data 2”



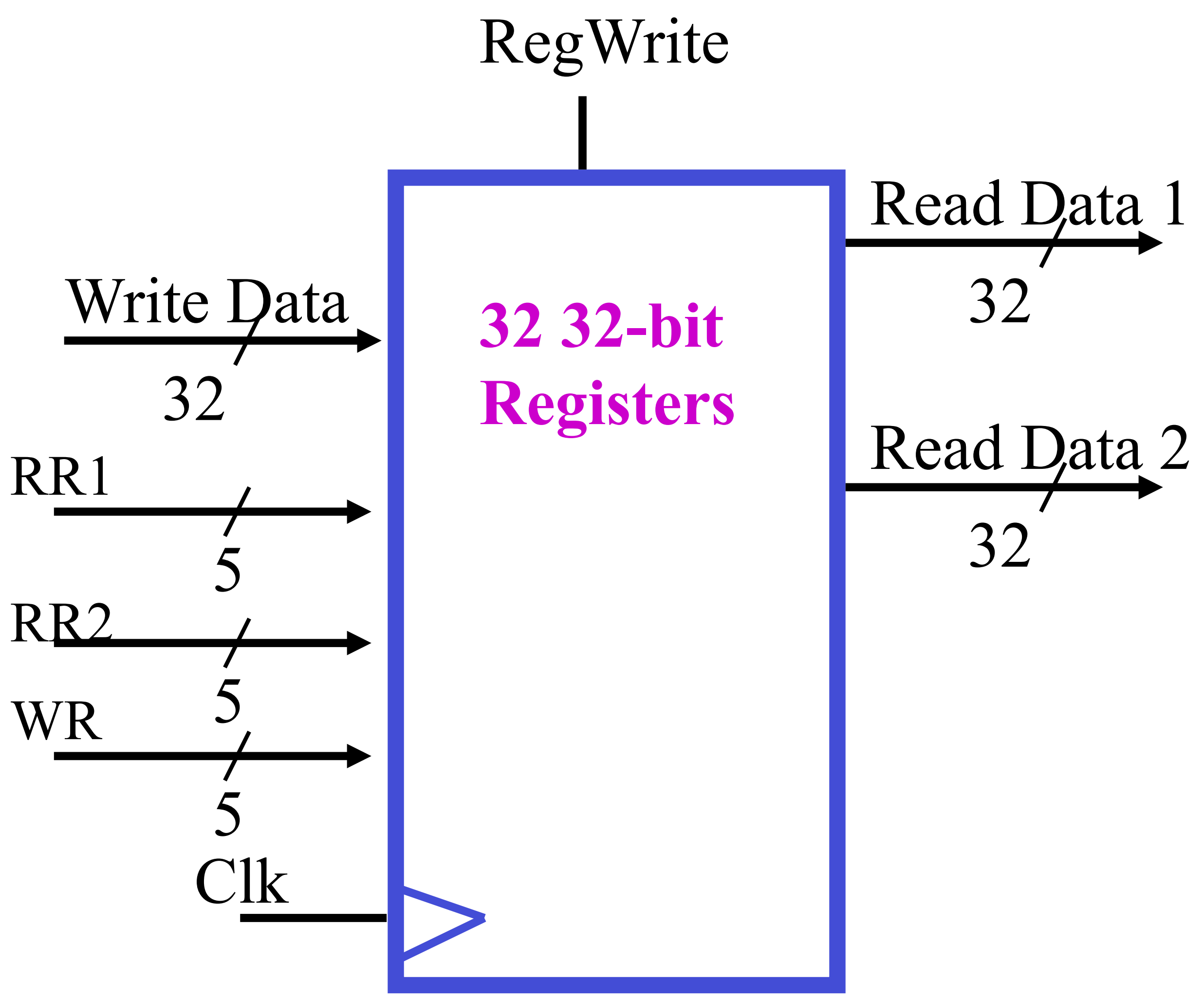
Storage Element: Register File

- Register File consists of (32) registers:
    - Two 32-bit output buses
    - One 32-bit input bus
- Register is selected by:
    - RR1 selects the register to put on bus "Read Data 1"
    - RR2 selects the register to put on bus "Read Data 2"
    - WR selects the register to be written



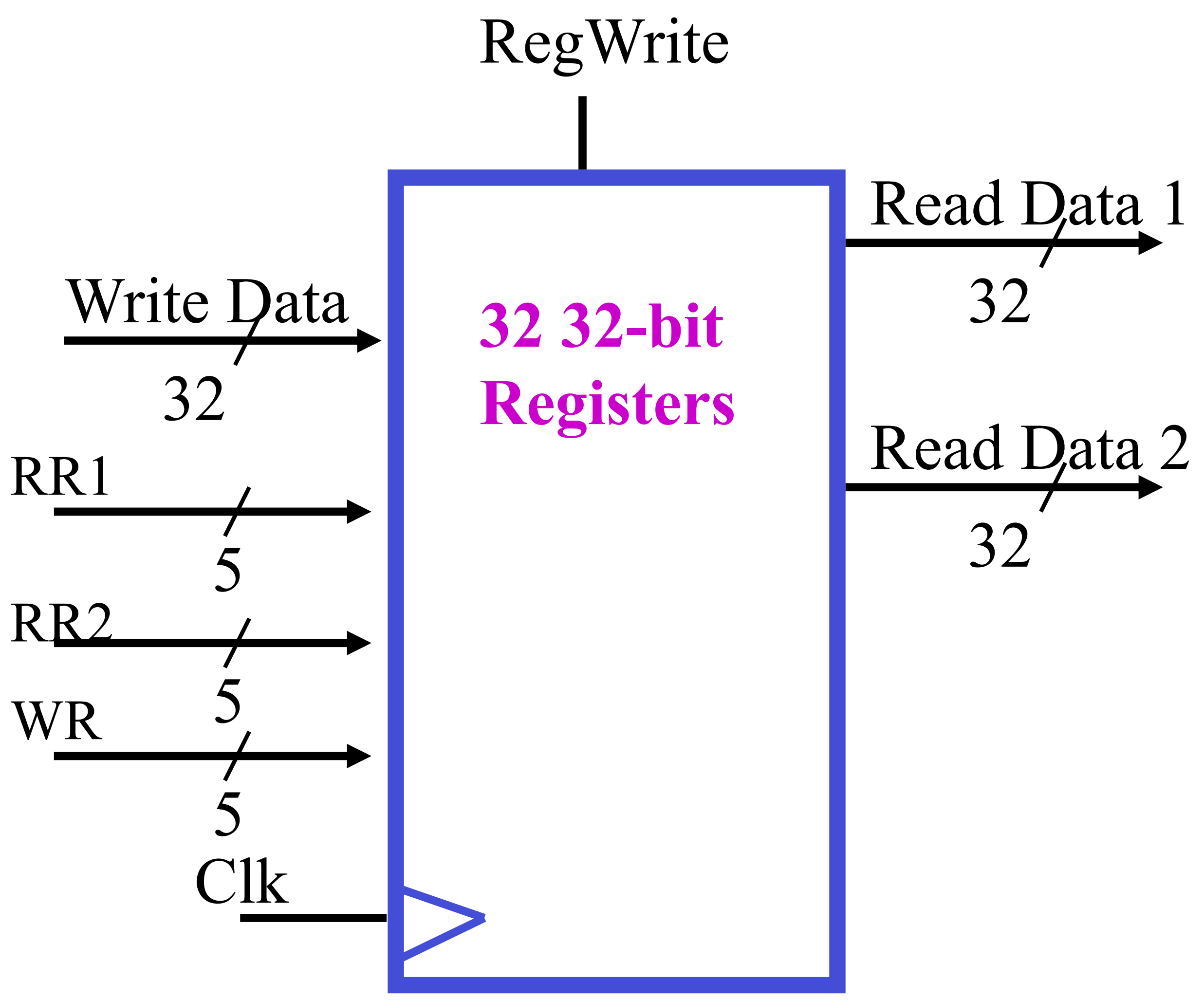
Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:
  - RR1 selects the register to put on bus "Read Data 1"
  - RR2 selects the register to put on bus "Read Data 2"
  - WR selects the register to be written
  - via WriteData when RegWrite is 1

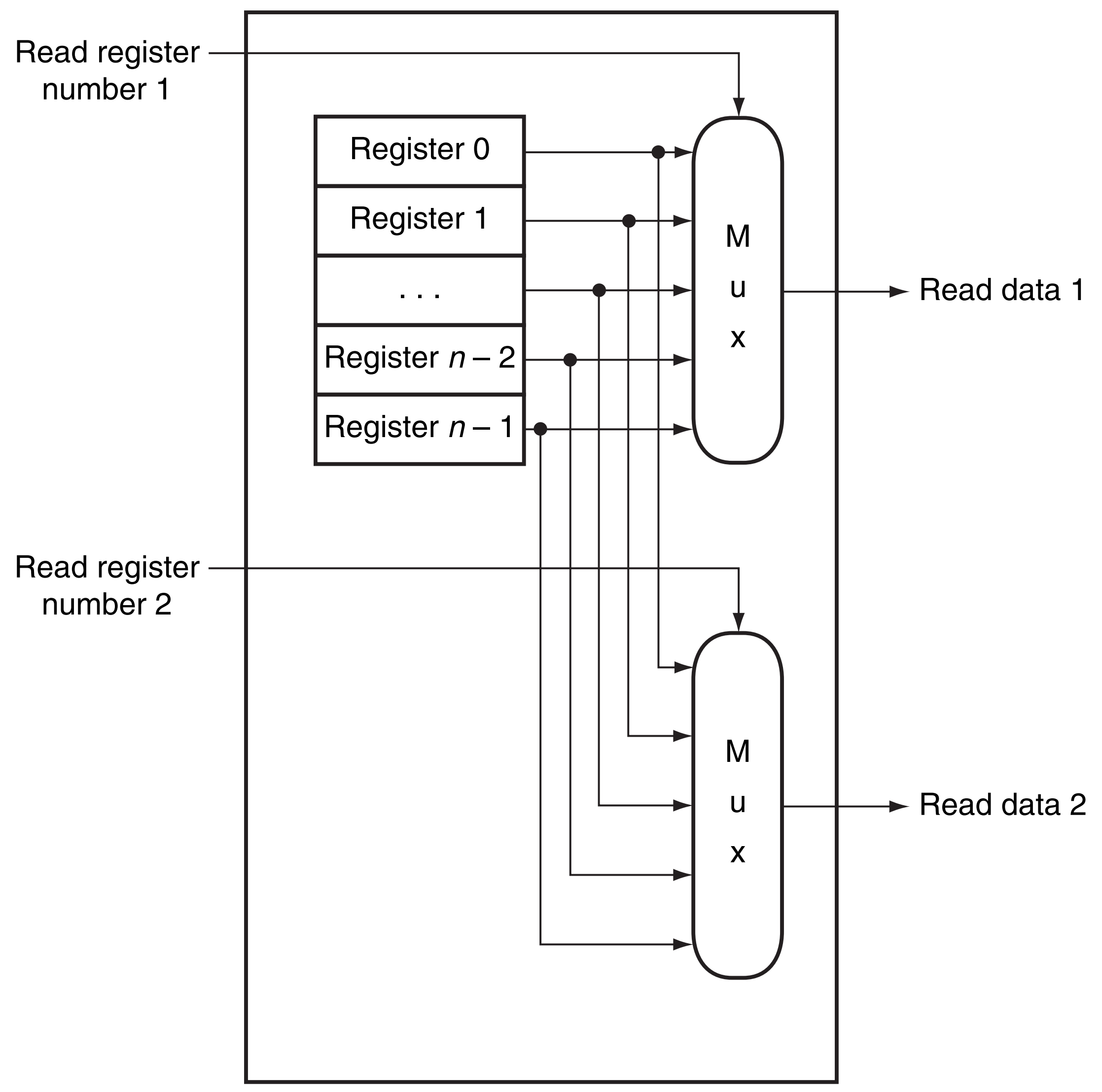


Storage Element: Register File

- Register File consists of (32) registers:
  - Two 32-bit output buses
  - One 32-bit input bus
- Register is selected by:
  - RR1 selects the register to put on bus “Read Data 1”
  - RR2 selects the register to put on bus “Read Data 2”
  - WR selects the register to be written
  - via WriteData when RegWrite is 1
- Clock input (CLK)

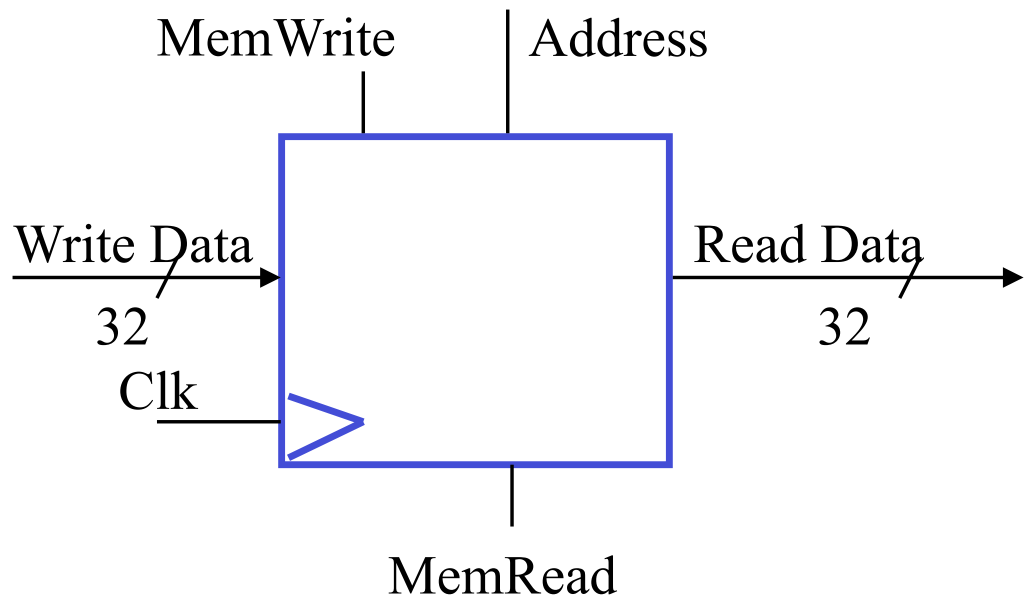


Inside the Register File



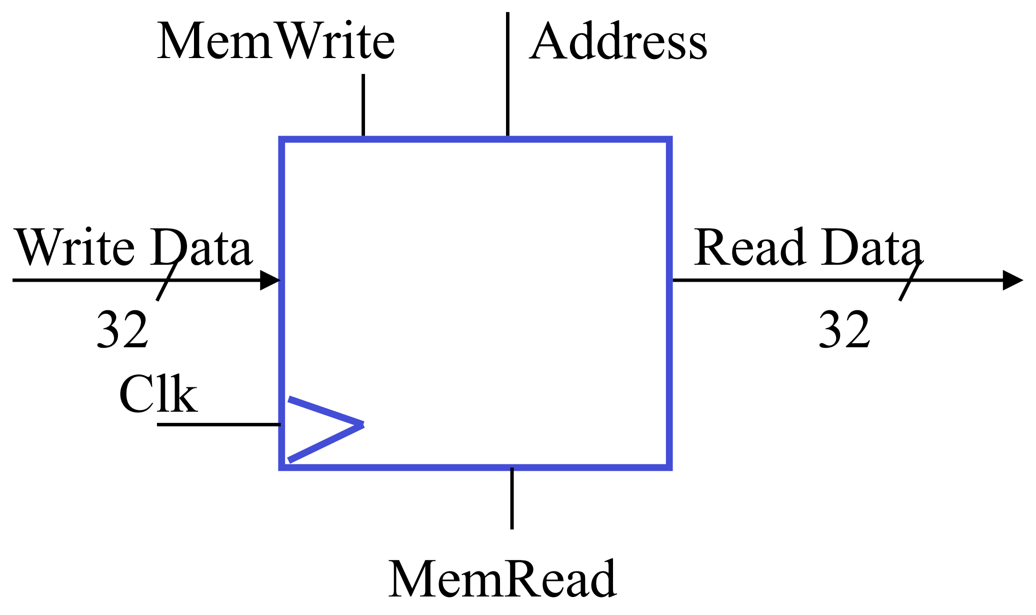
- The implementation of two read ports register file
  - n registers
  - done with a pair of n-to-1 multiplexors, each 32 bits wide.

Storage Element: Memory



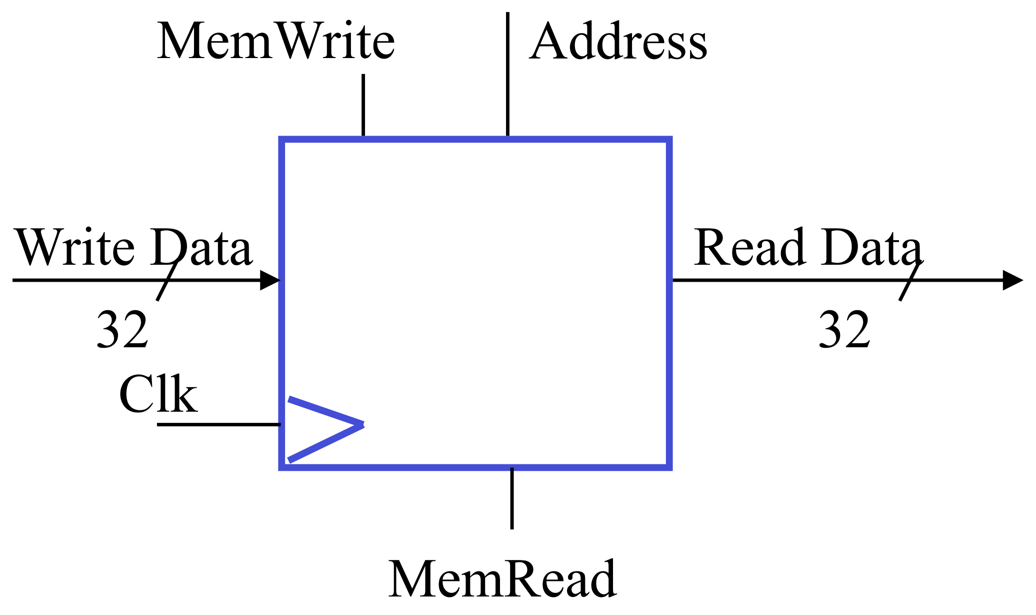
Storage Element: Memory

- Memory



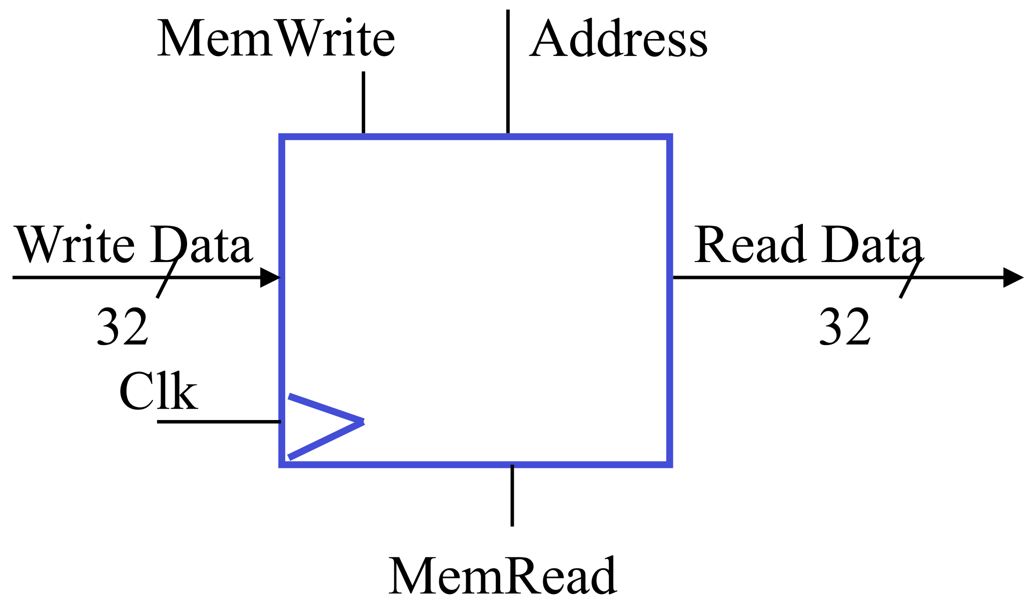
Storage Element: Memory

- Memory
    - Two input buses: WriteData, Address



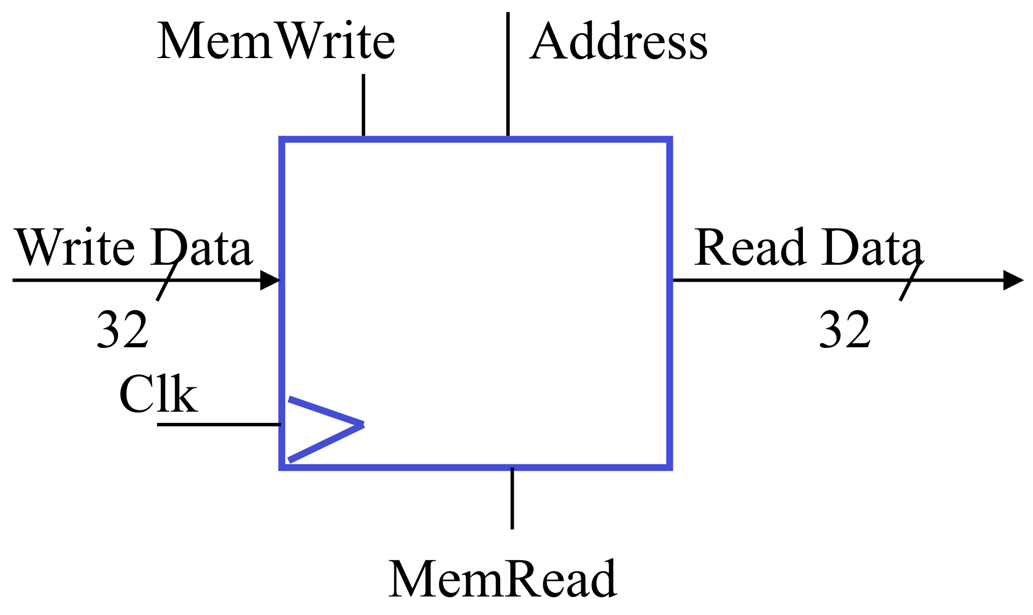
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData



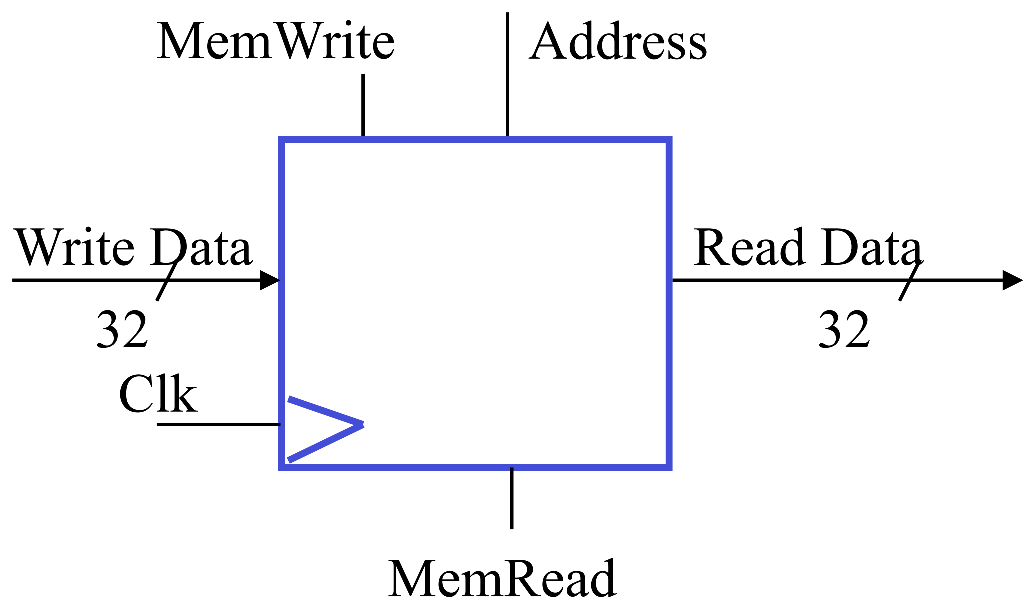
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:



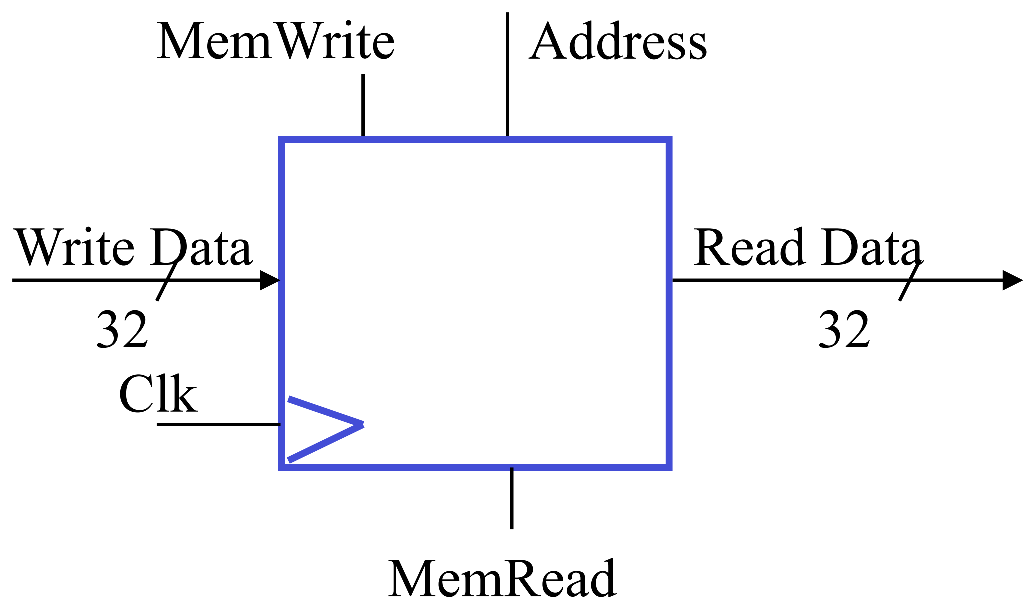
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
  - Address selects the word to put on ReadData bus



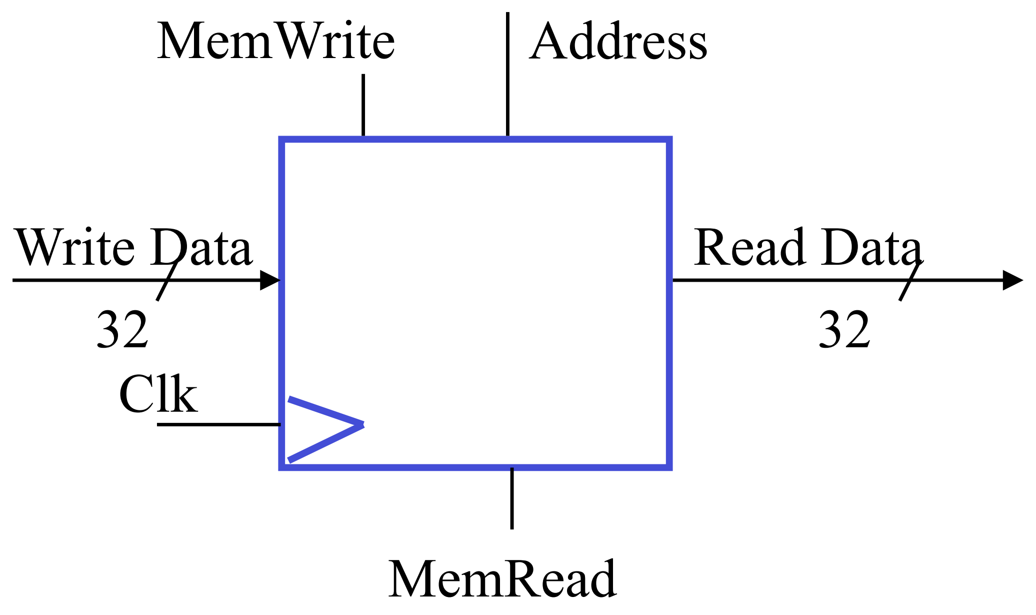
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
  - Address selects the word to put on ReadData bus
  - If MemWrite = 1: address selects the memory word to be written via the WriteData bus



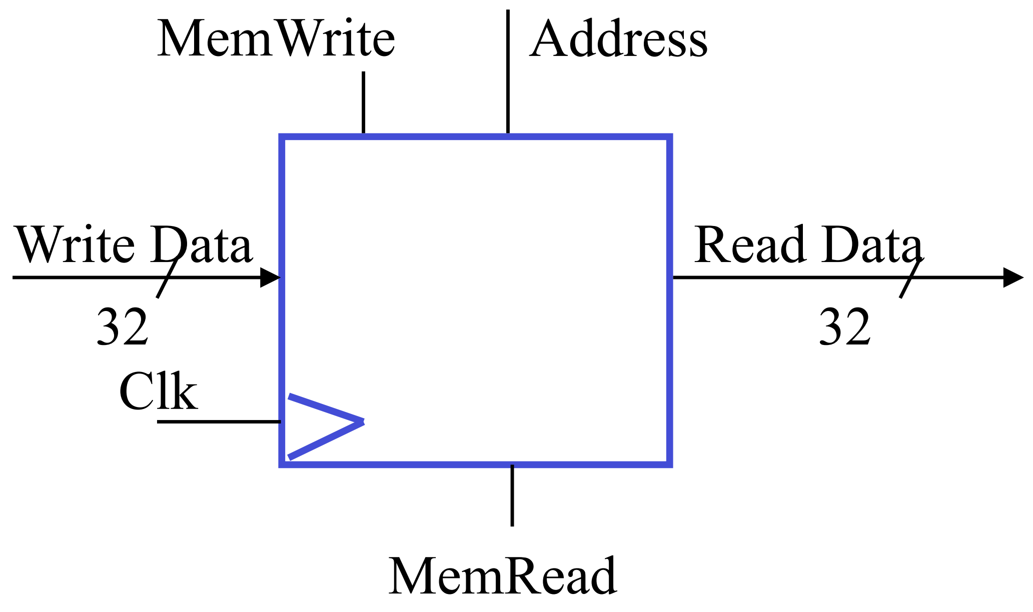
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
  - Address selects the word to put on ReadData bus
  - If MemWrite = 1: address selects the memory word to be written via the WriteData bus
- Clock input (CLK)



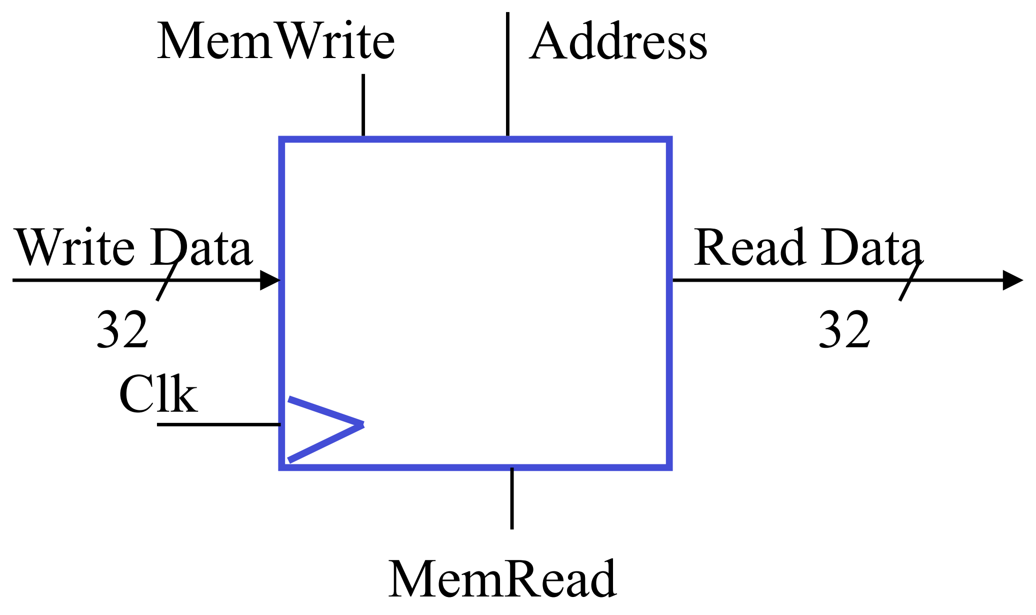
Storage Element: Memory

- Memory
    - Two input buses: WriteData, Address
    - One output bus: ReadData
- Memory word is selected by:
    - Address selects the word to put on ReadData bus
    - If MemWrite = 1: address selects the memory word to be written via the WriteData bus
- Clock input (CLK)
    - The CLK input is a factor ONLY during write operation



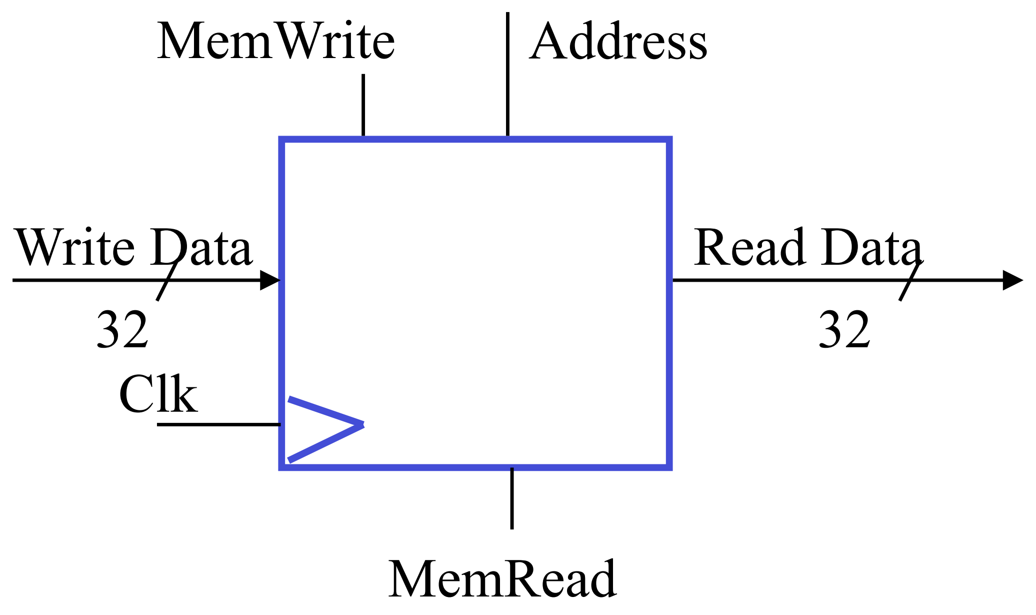
Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
  - Address selects the word to put on ReadData bus
  - If MemWrite = 1: address selects the memory word to be written via the WriteData bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:



Storage Element: Memory

- Memory
  - Two input buses: WriteData, Address
  - One output bus: ReadData
- Memory word is selected by:
  - Address selects the word to put on ReadData bus
  - If MemWrite = 1: address selects the memory word to be written via the WriteData bus
- Clock input (CLK)
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid => ReadData valid after "access time."

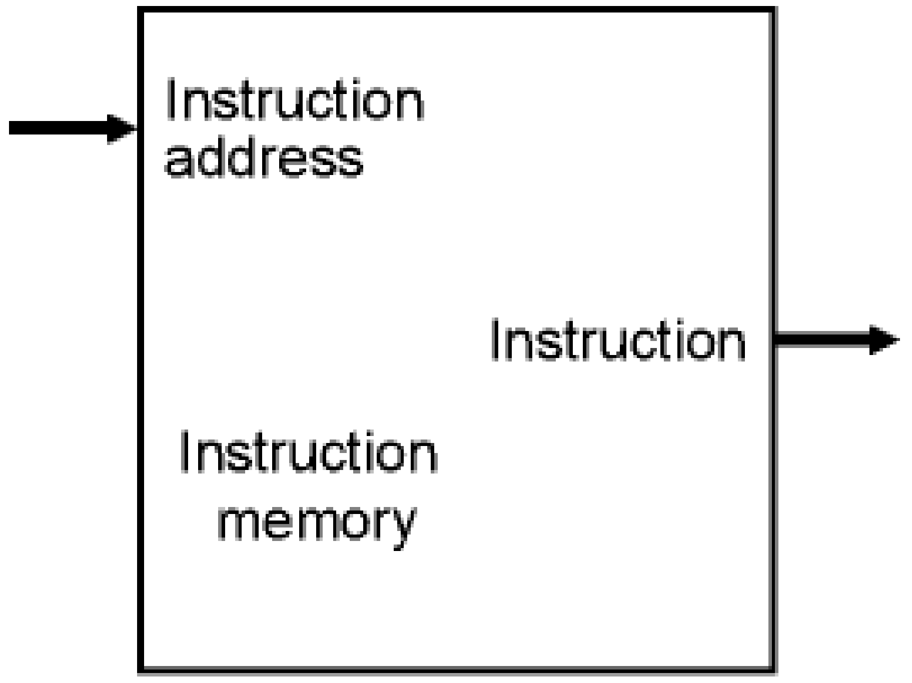


RTL: Register Transfer Language

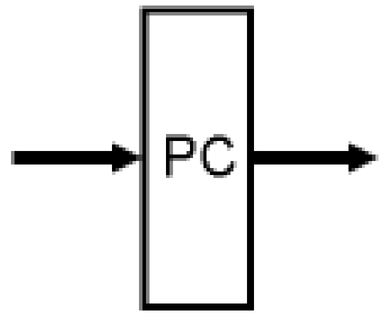
- Describes the movement and manipulation of data between storage elements:

```
R[3] <- R[5] + R[7]
PC <- PC + 4 + R[5]
R[rd] <- R[rs] + R[rt]
R[rt] <- Mem[R[rs] + immed]
```

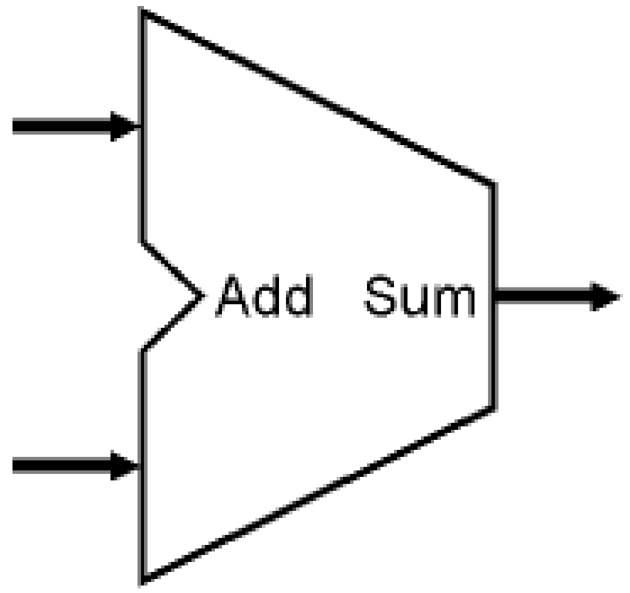
Instruction Fetch and Program Counter Management



a. Instruction memory



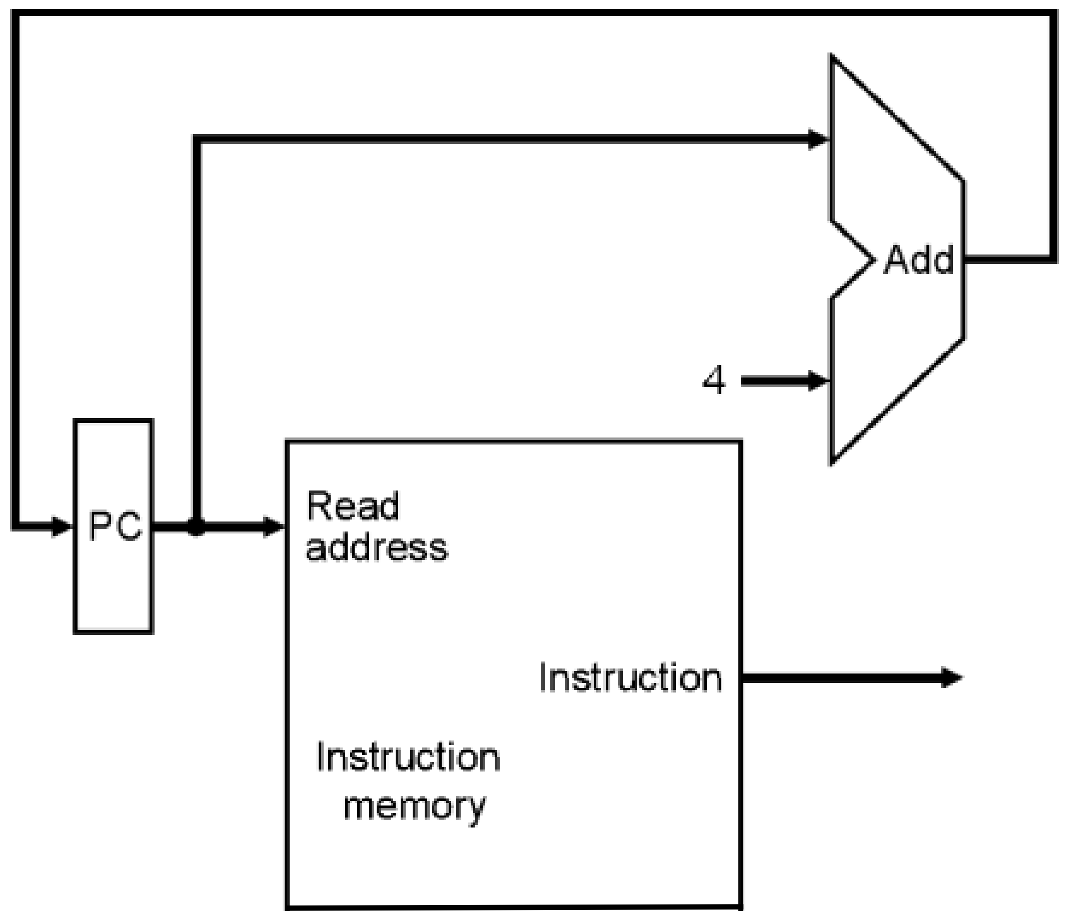
b. Program counter



c. Adder

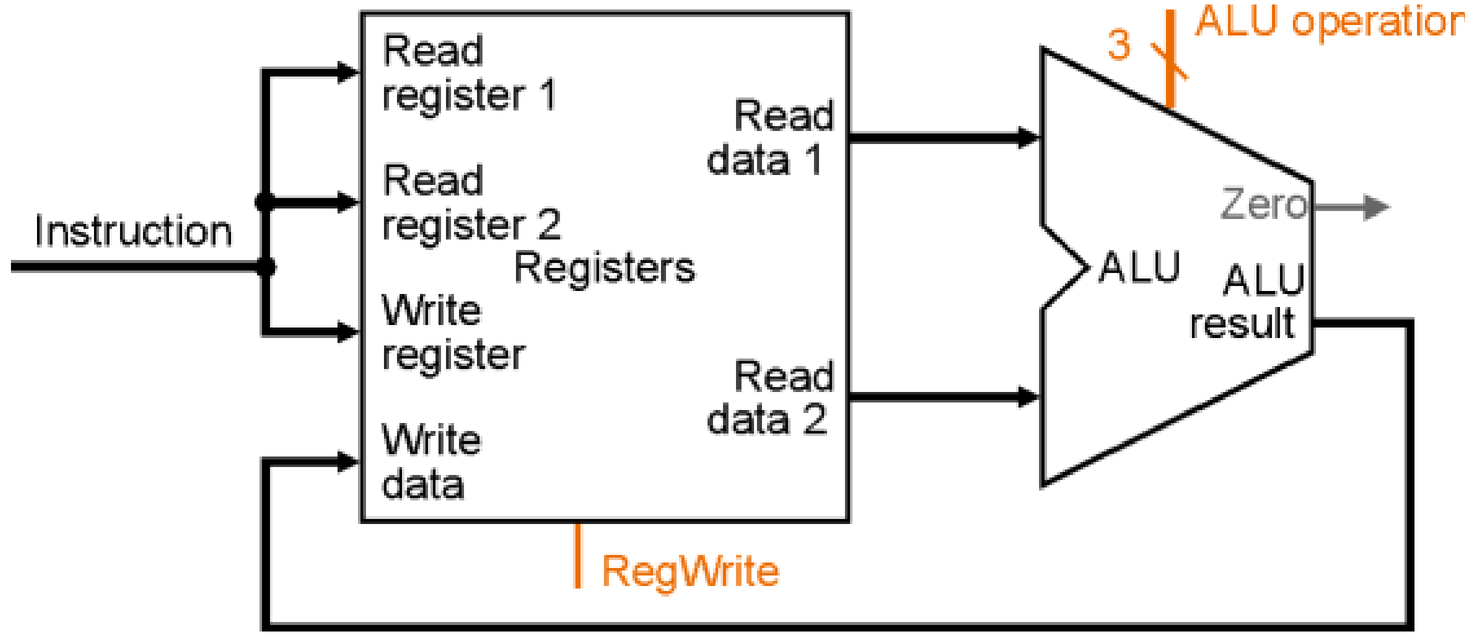
Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: inst <- mem[PC]
  - Update the program counter:
    - Sequential Code: PC <- PC + 4
    - Branch and Jump   PC <- “something else”



Datapath for Register-Register Operations

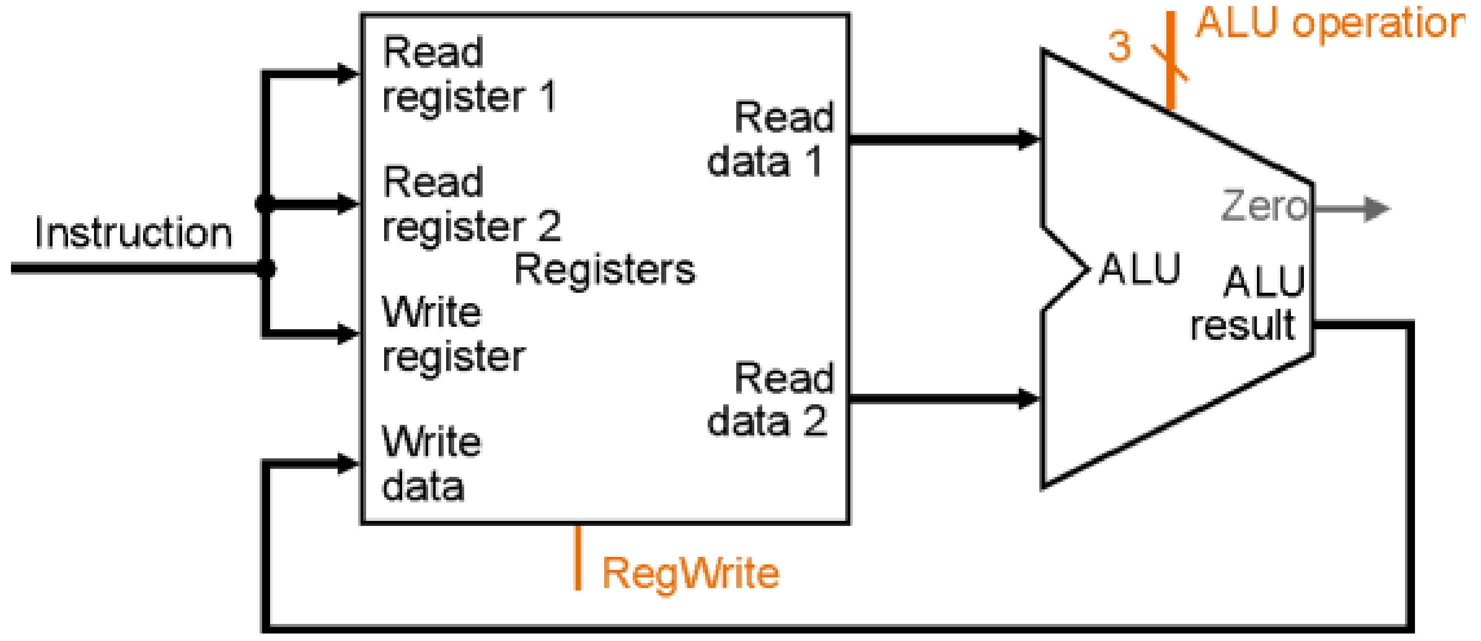
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct



Datapath for Register-Register Operations

- R[rd] <- R[rs] op R[rt] Example: *add   rd, rs, rt*

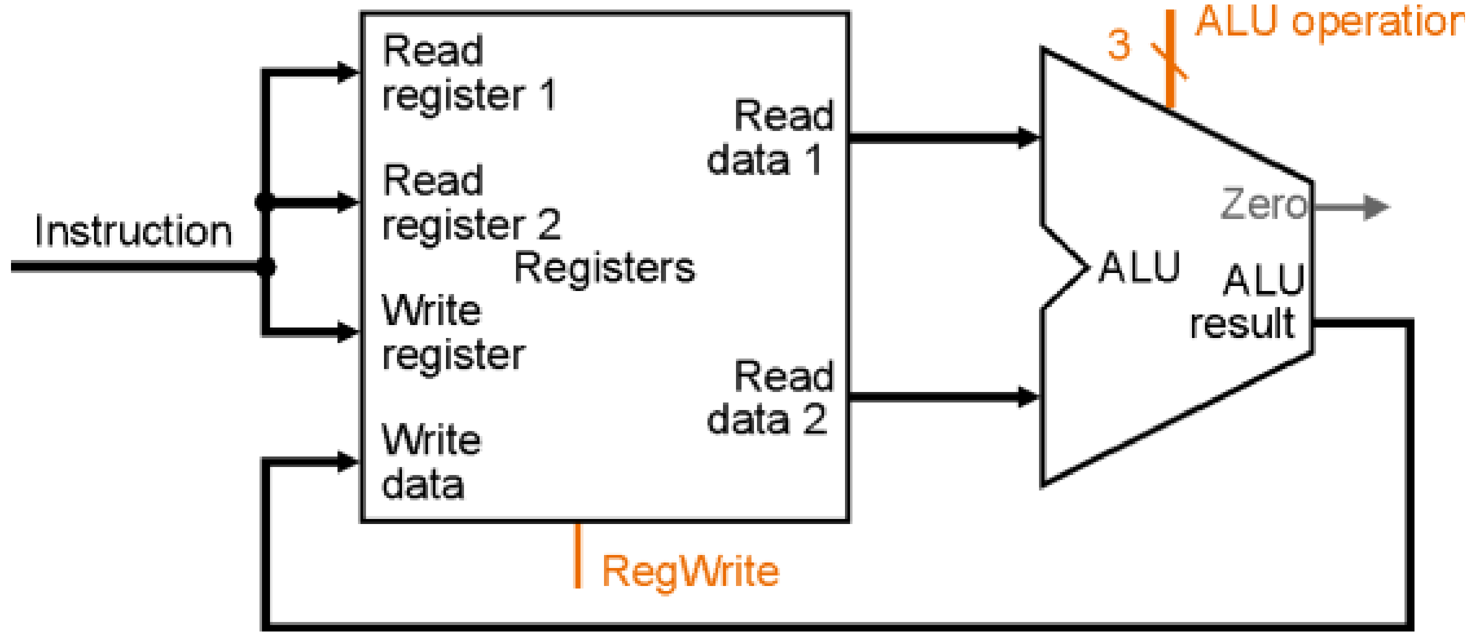
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct



Datapath for Register-Register Operations

- R[rd] <- R[rs] op R[rt] Example: *add   rd, rs, rt*
  - RR1, RR2, and WR comes from instruction's rs, rt, and rd fields

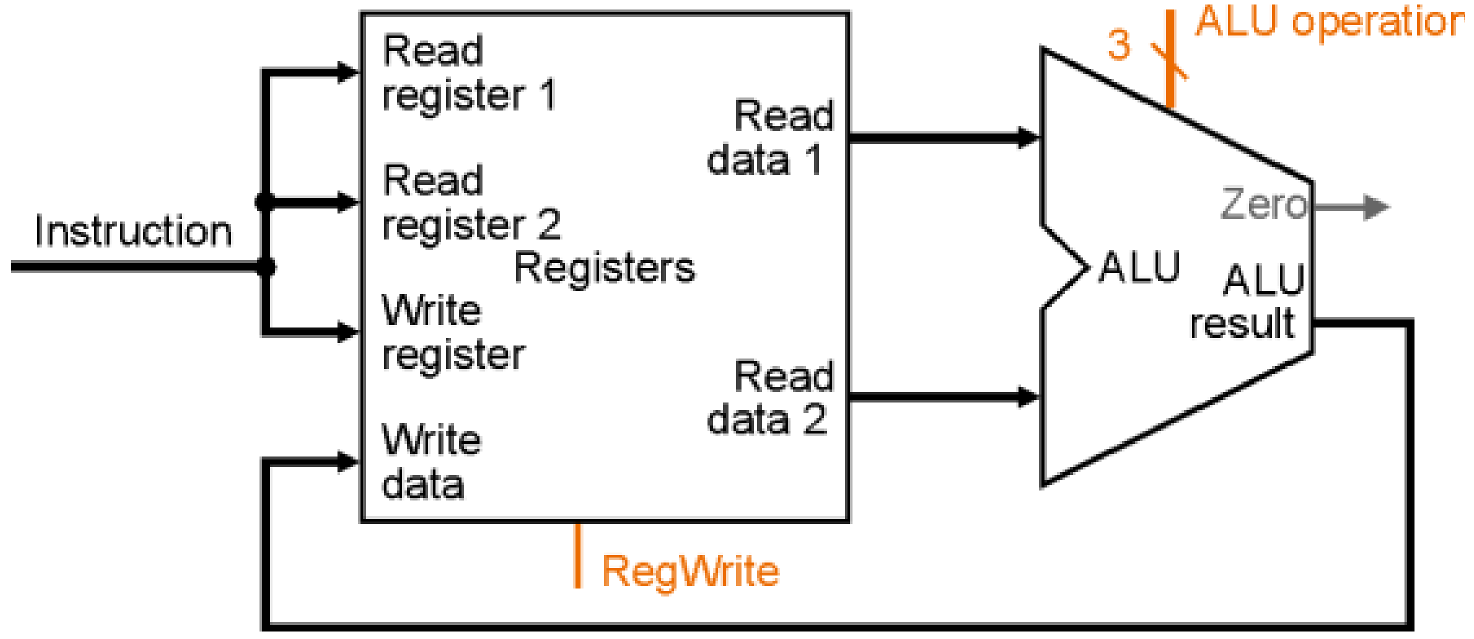
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct



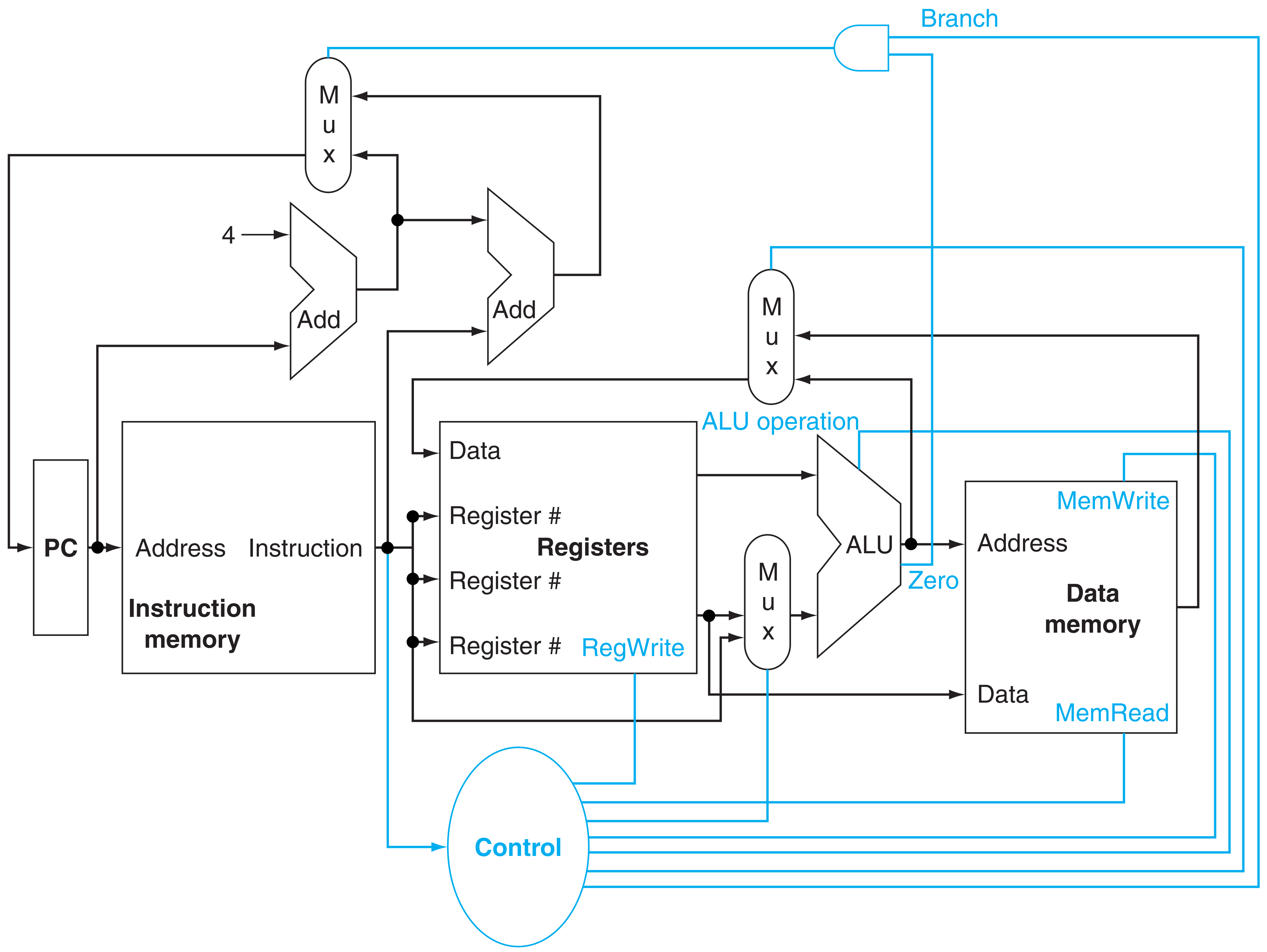
Datapath for Register-Register Operations

- R[rd] <- R[rs] op R[rt] Example: *add   rd, rs, rt*
  - RR1, RR2, and WR comes from instruction's rs, rt, and rd fields
  - ALUoperation and RegWrite: control logic after decoding instruction

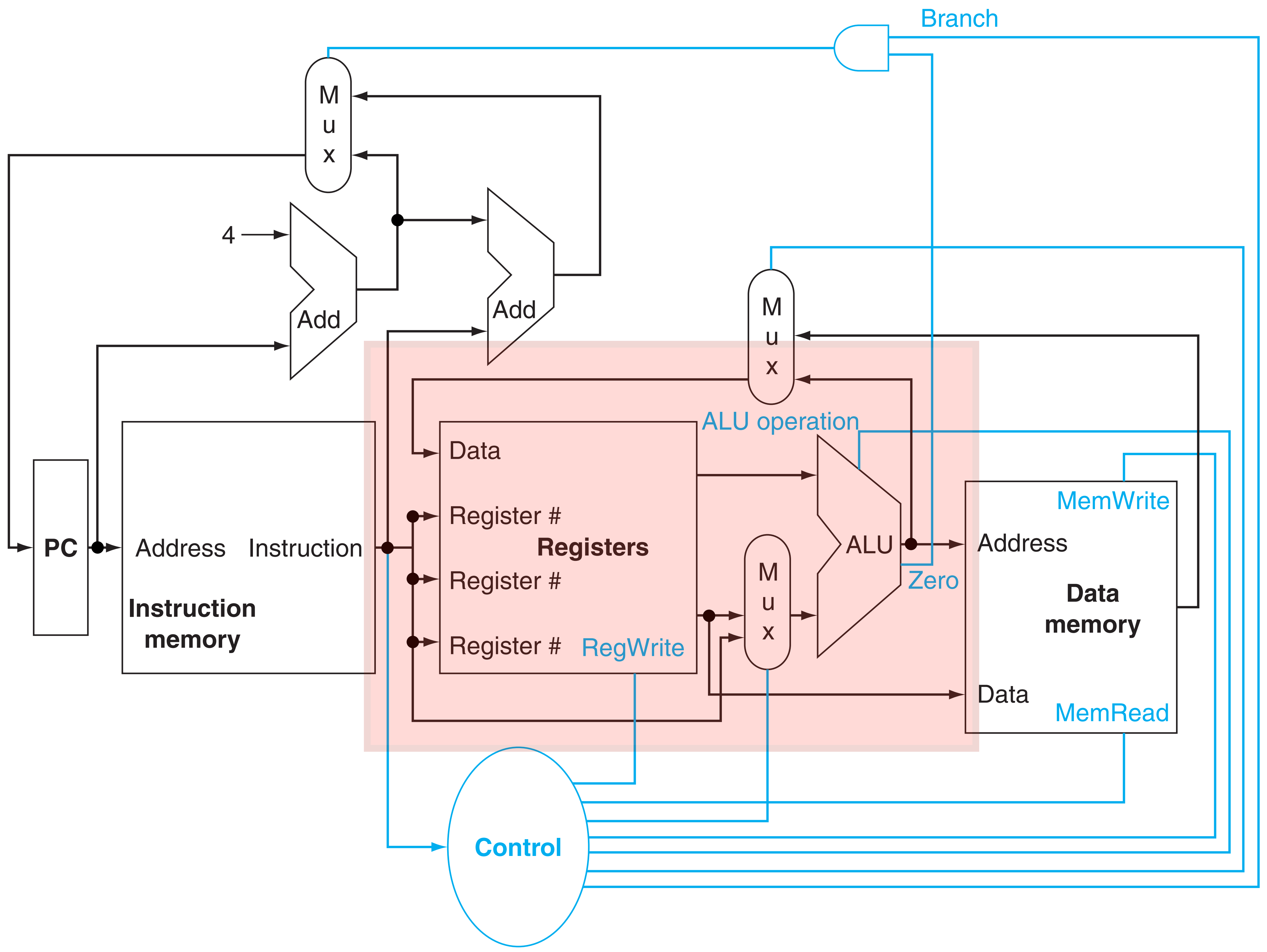
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
opcode	rs	rt	rd	shift amount	funct



Control Logic??



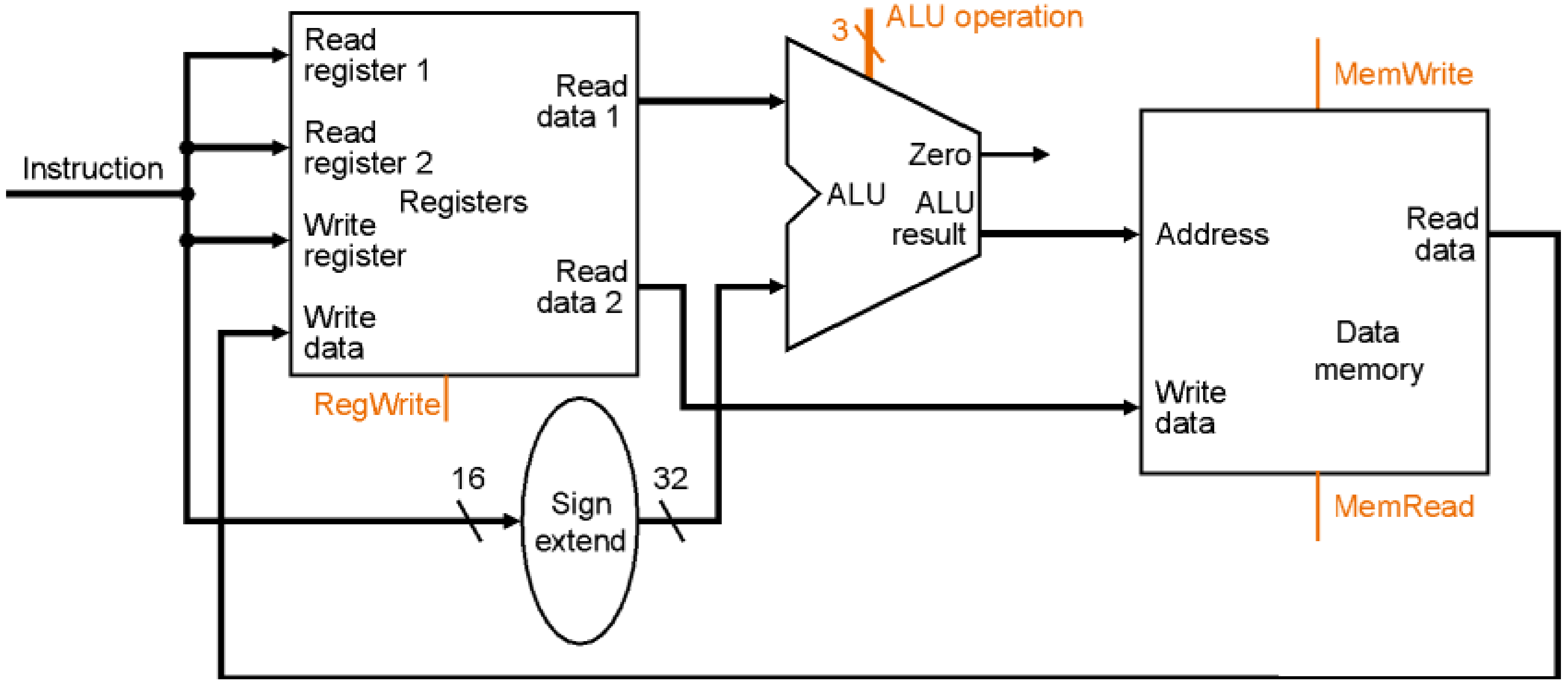
Control Logic??



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

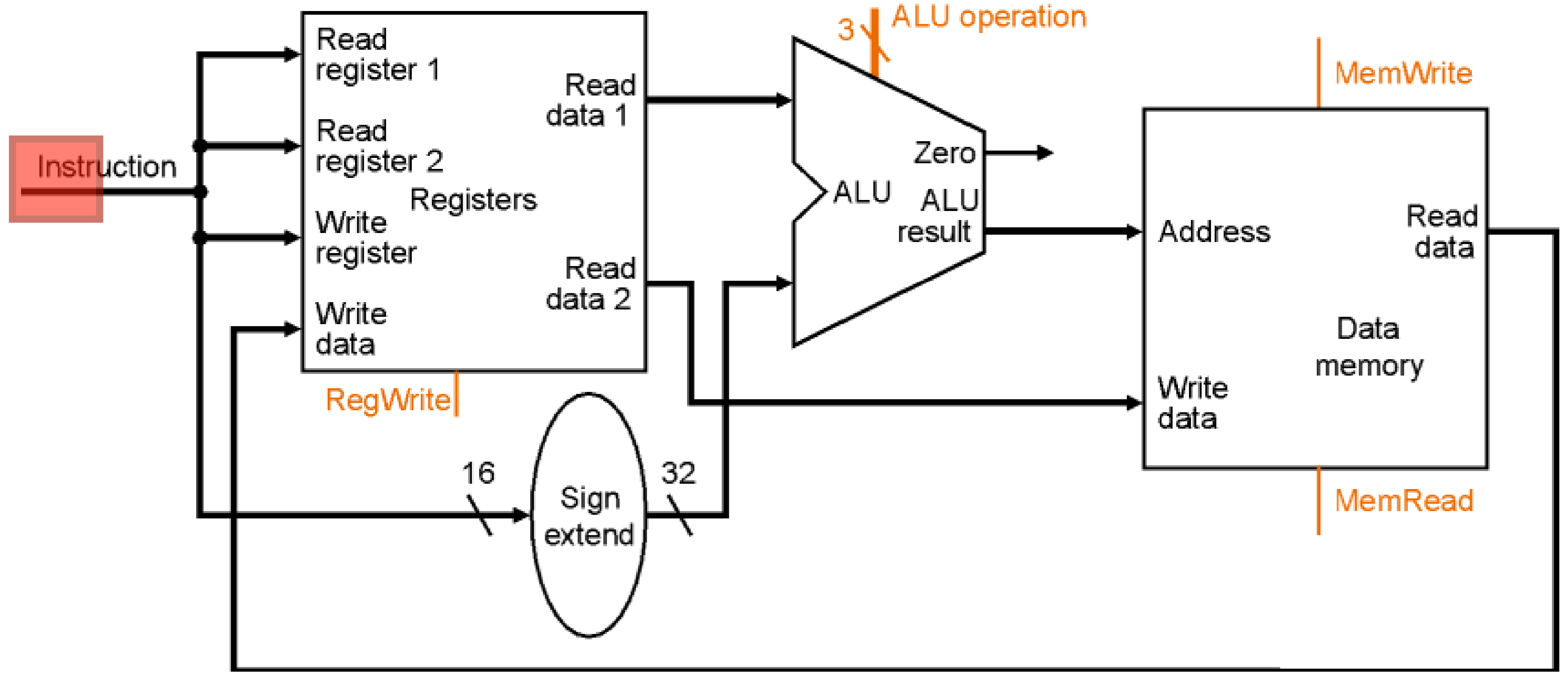
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

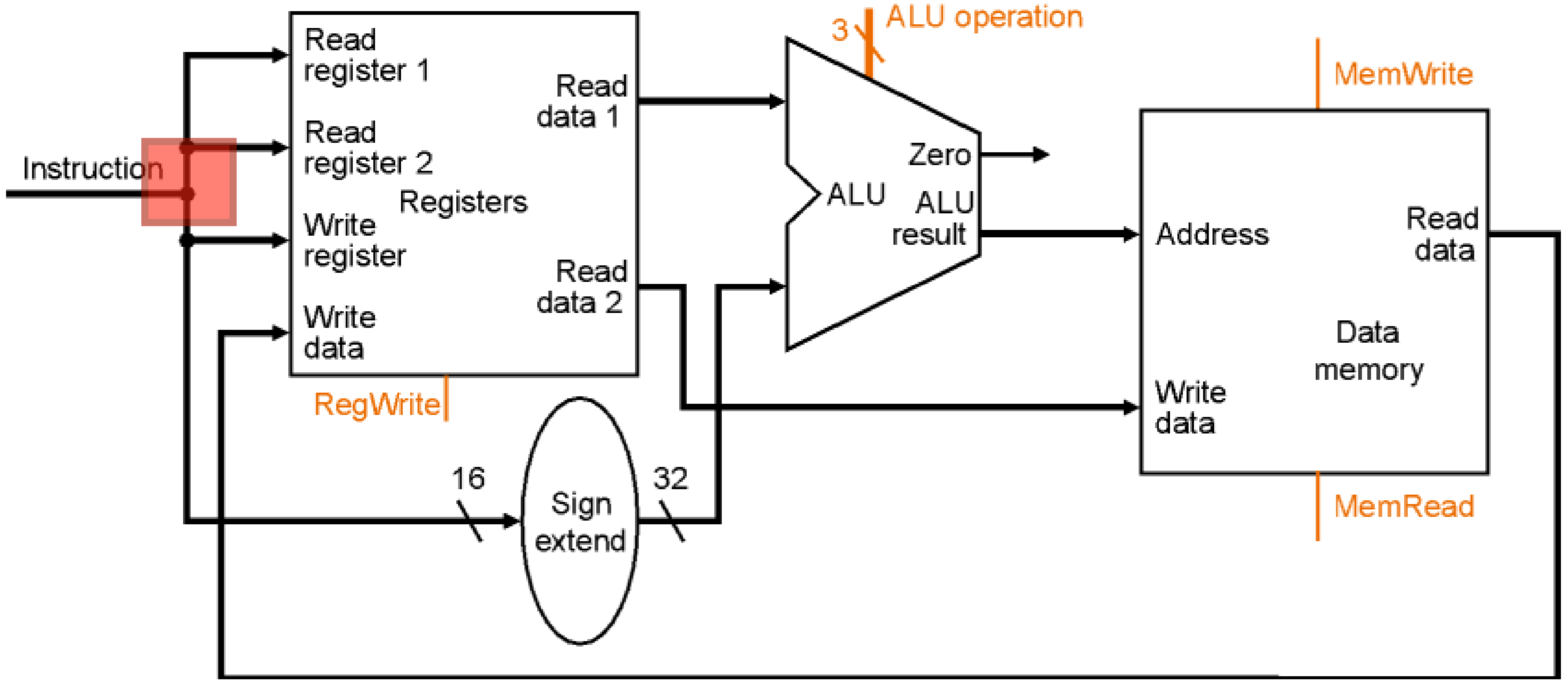
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

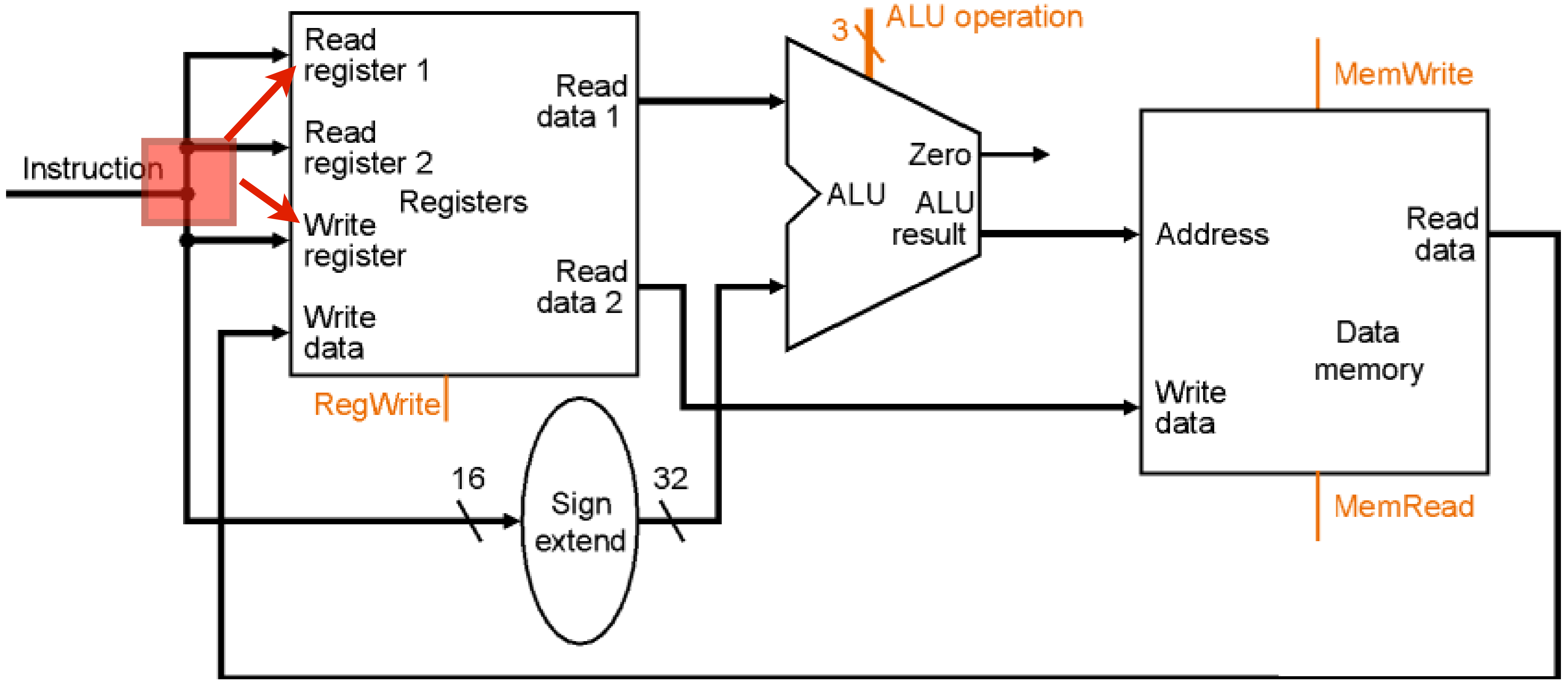
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw rt, rs, imm16*

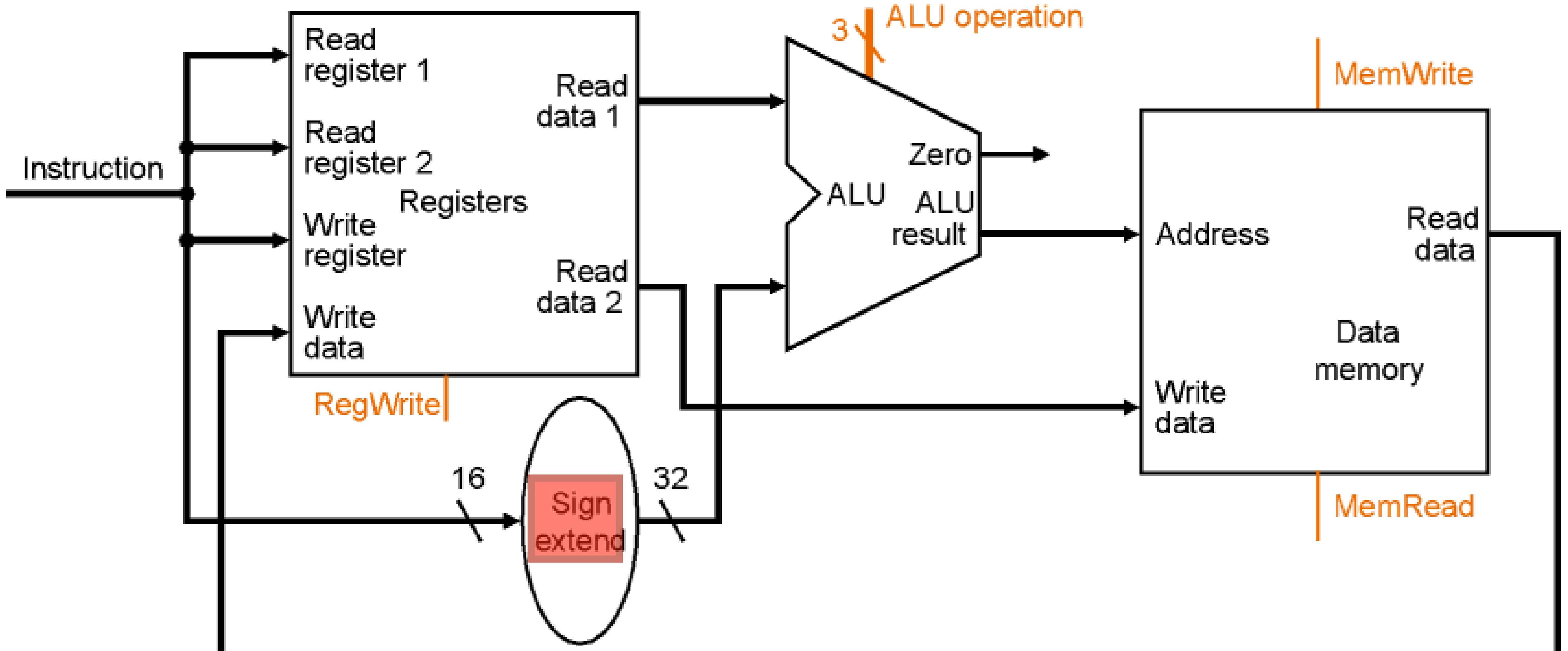
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

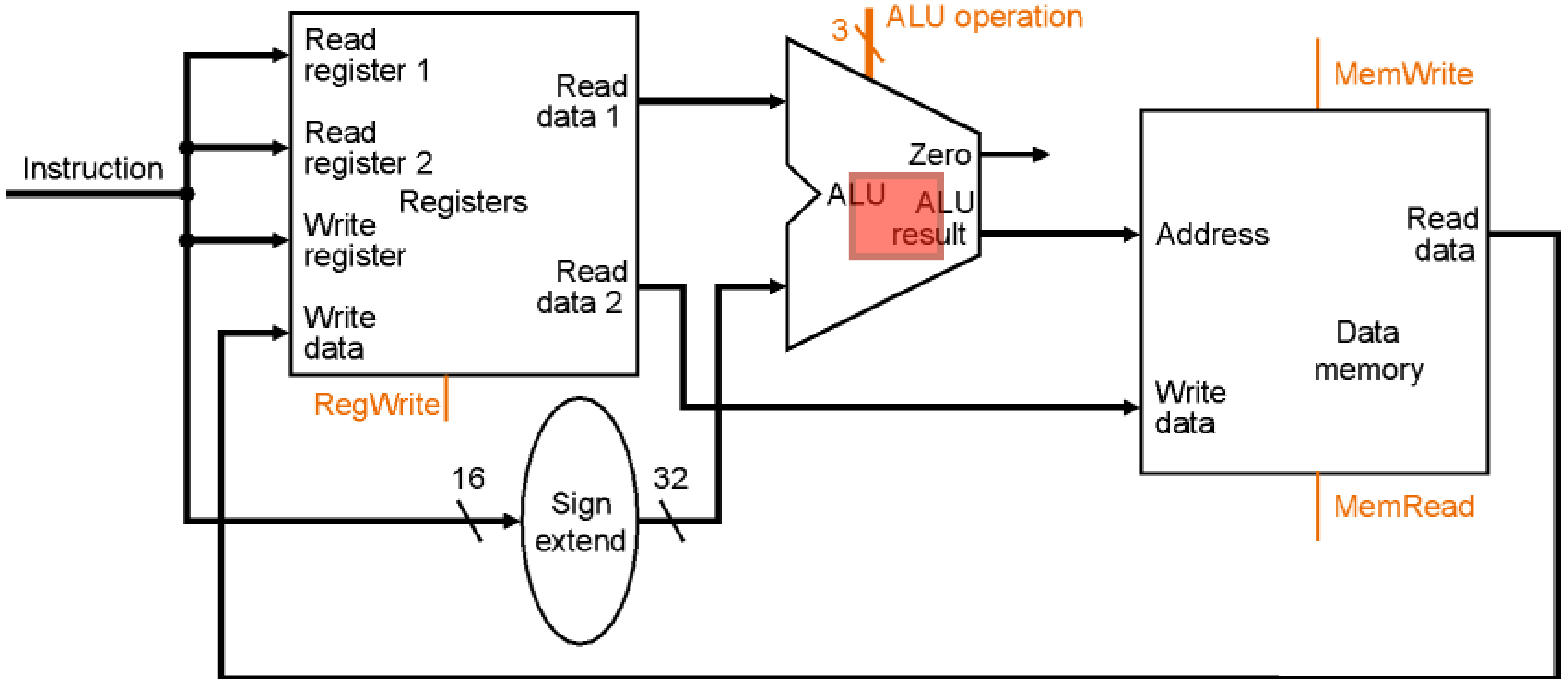
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

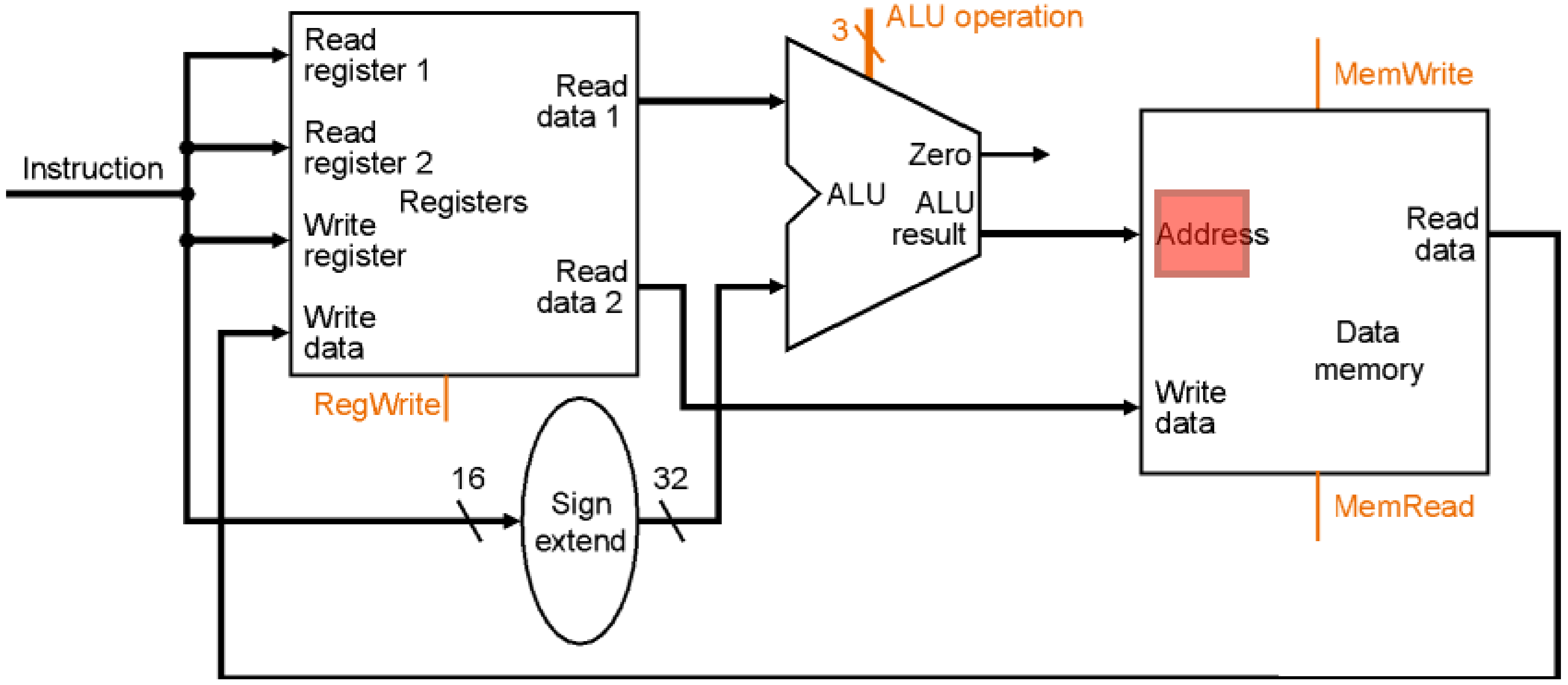
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

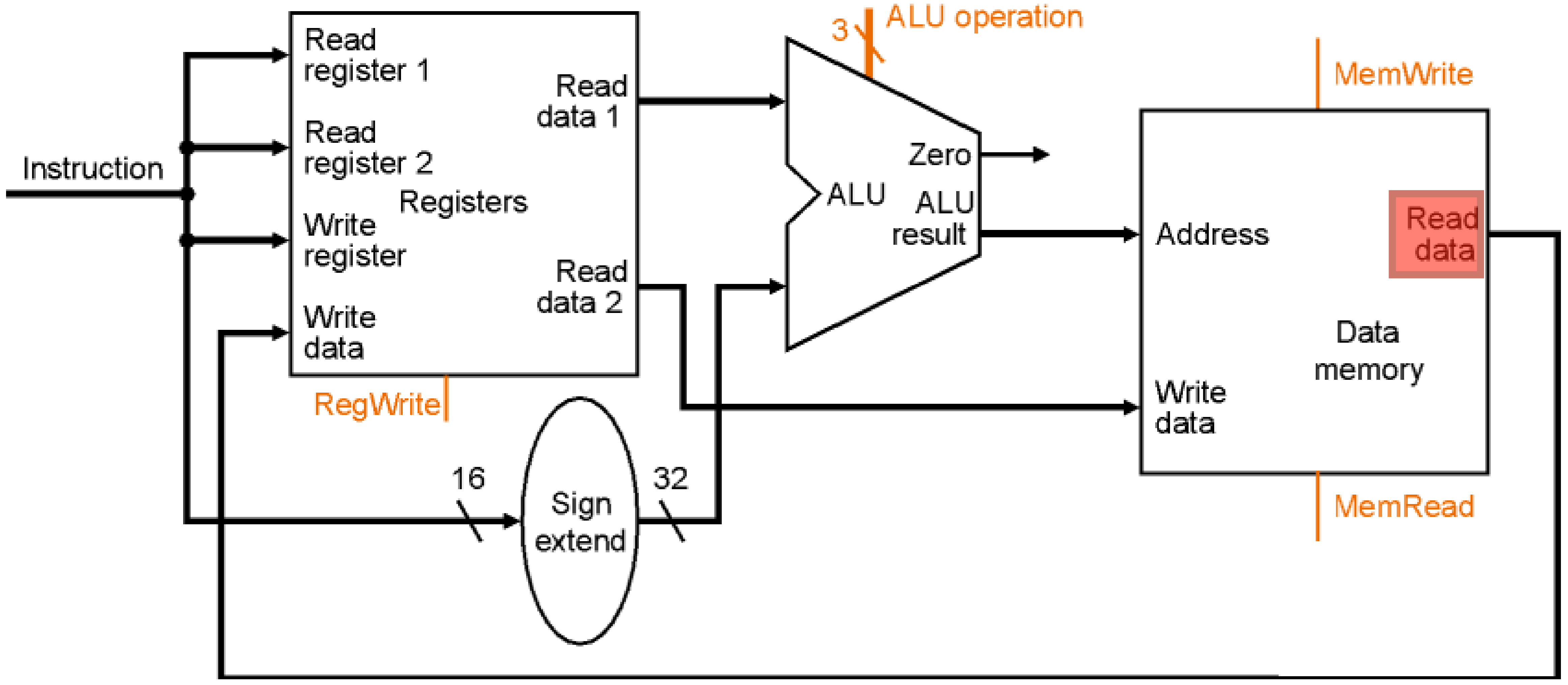
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

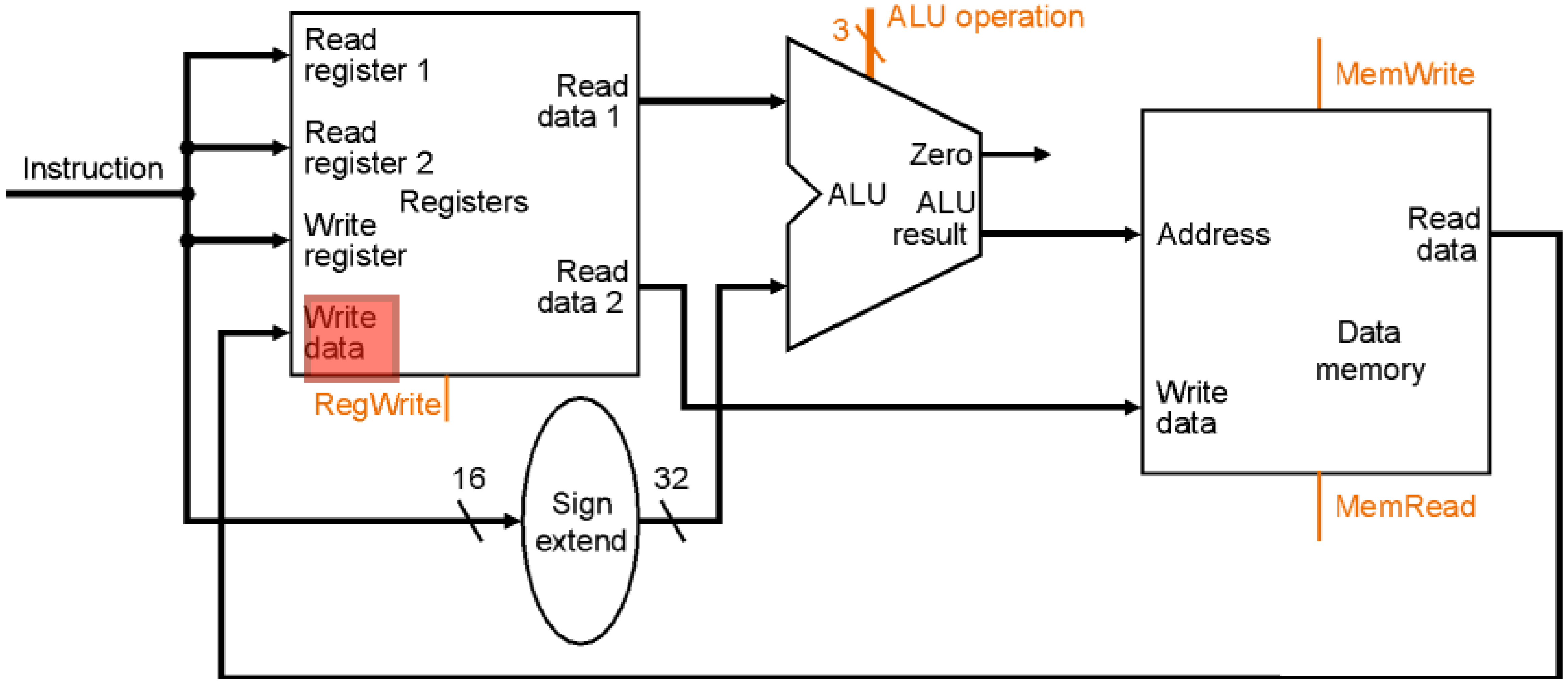
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Load Operations

- R[rt] <- Mem[R[rs] + SignExt[imm16]] Example: *lw   rt, rs, imm16*

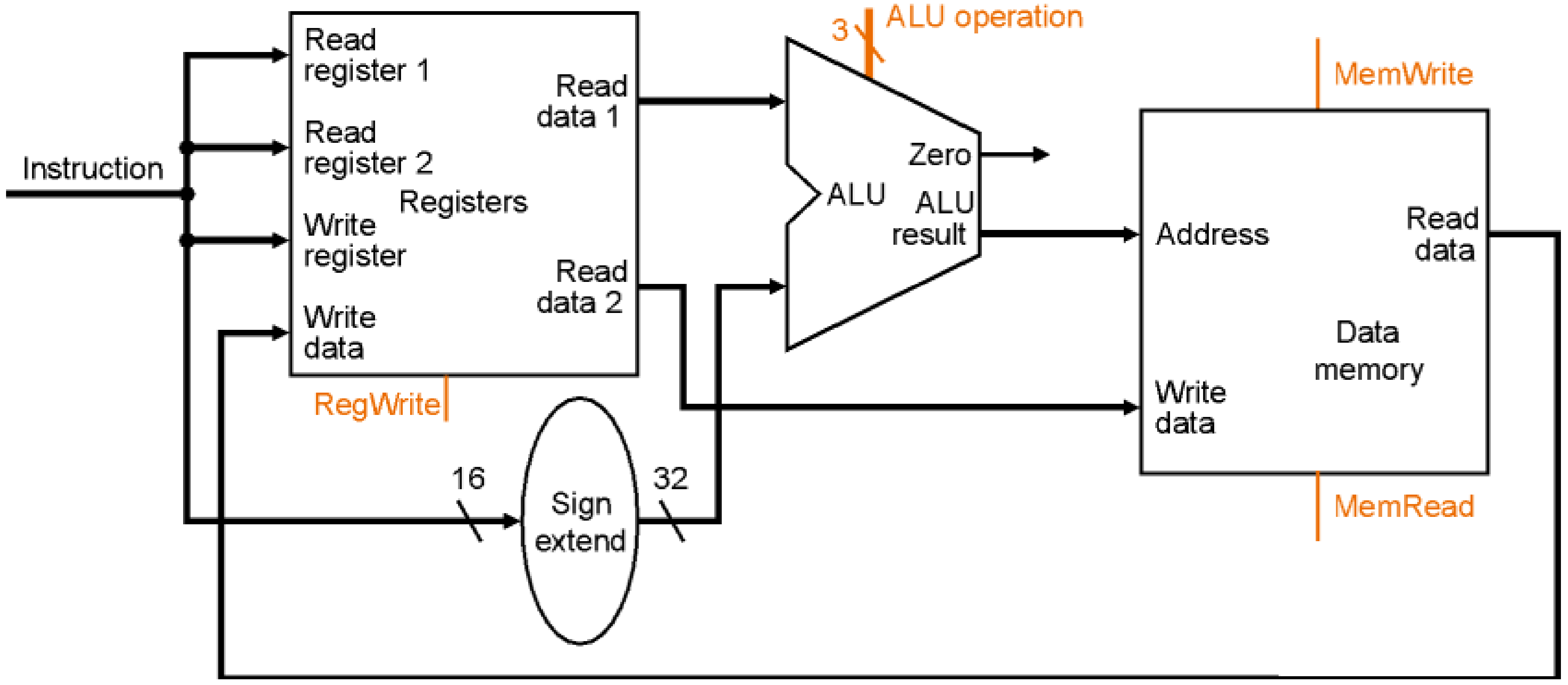
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

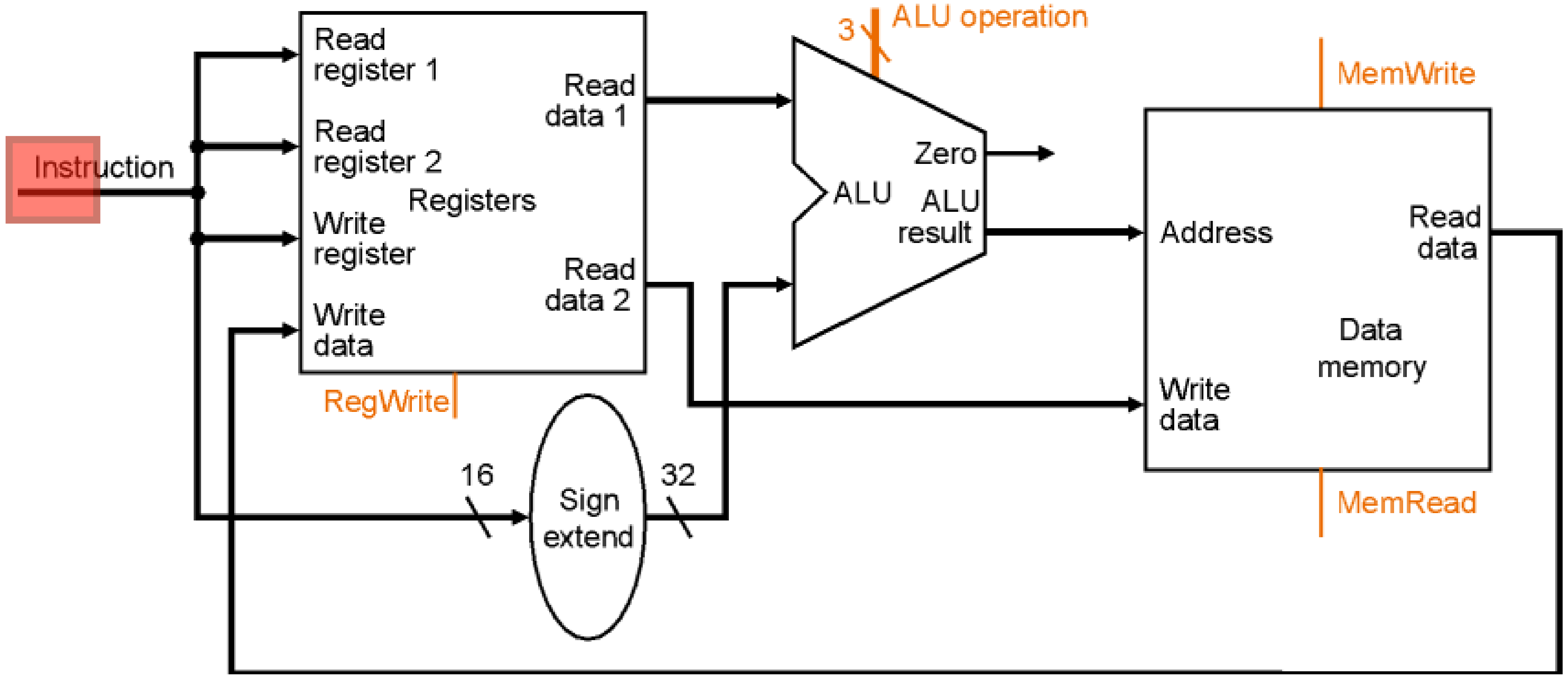
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

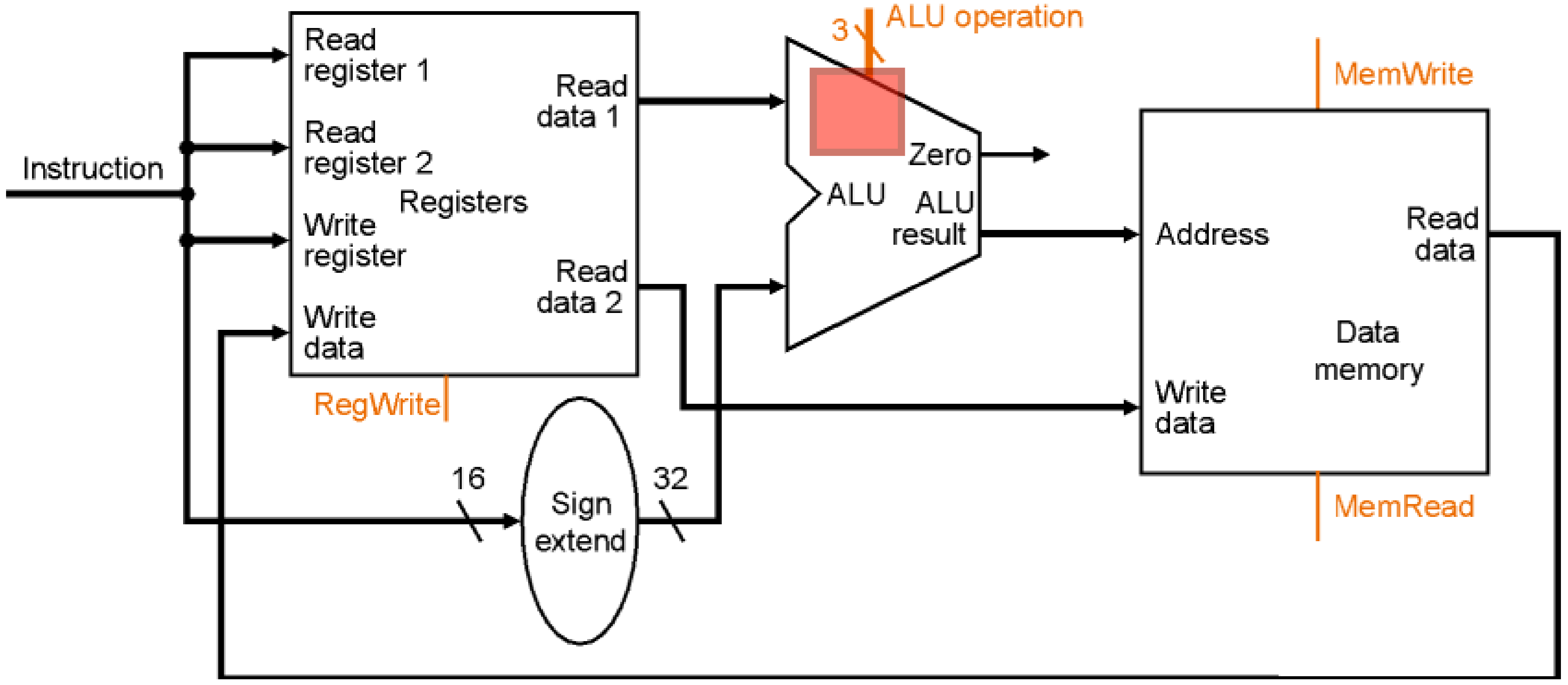
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

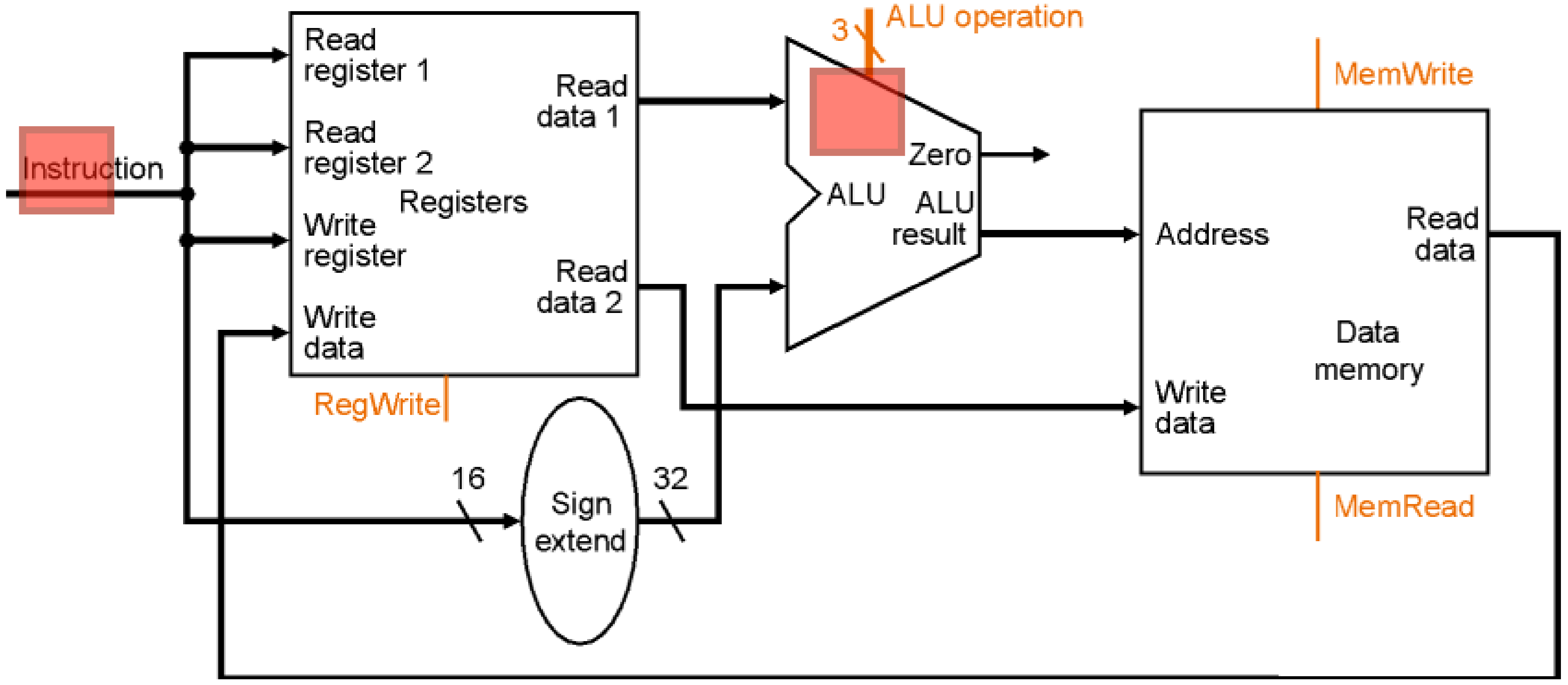
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

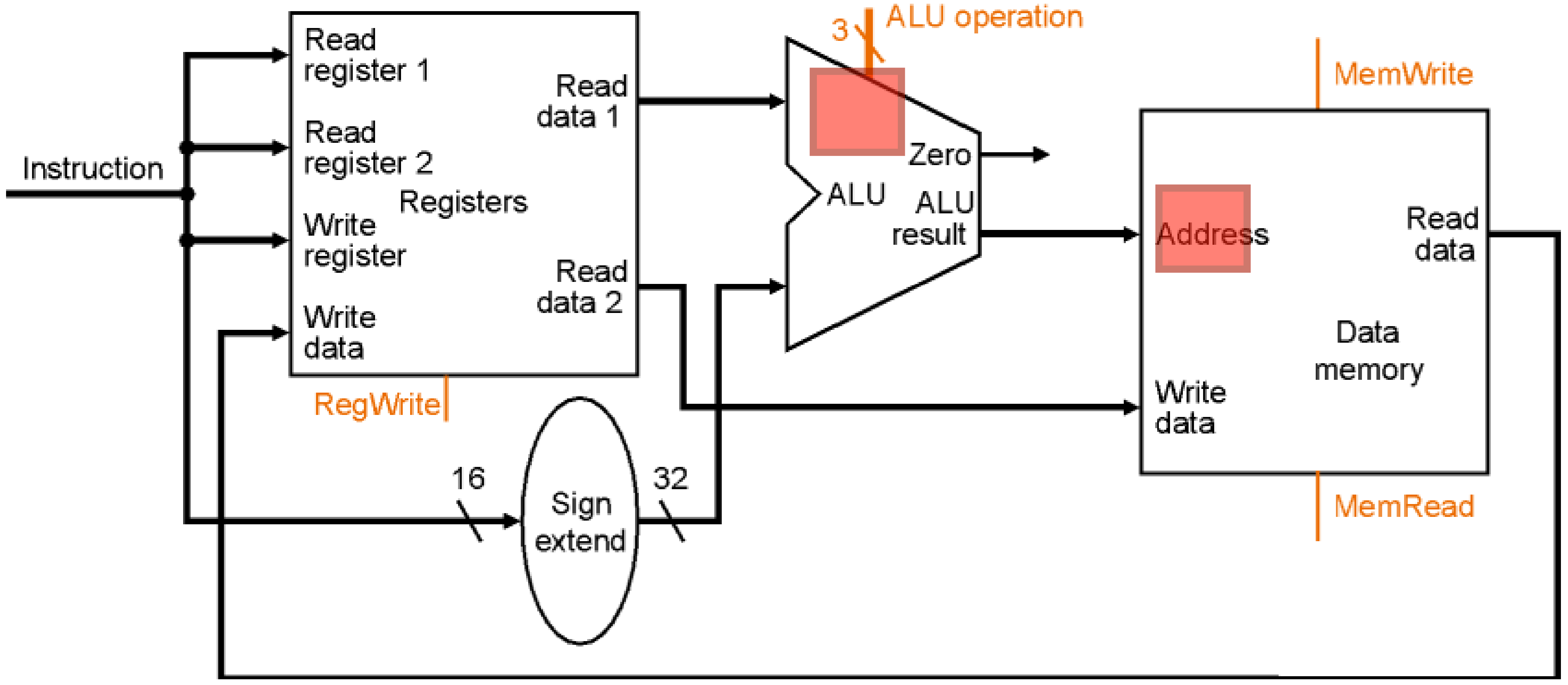
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

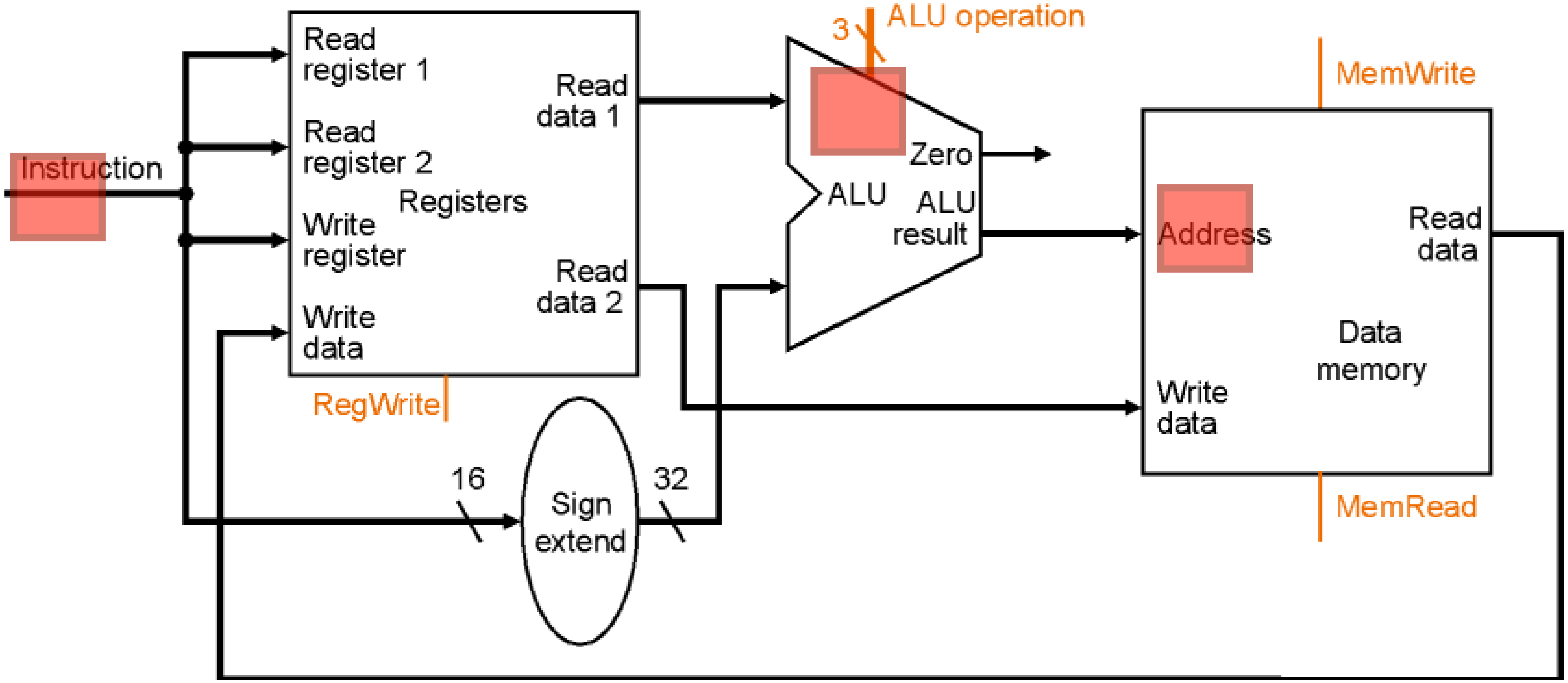
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

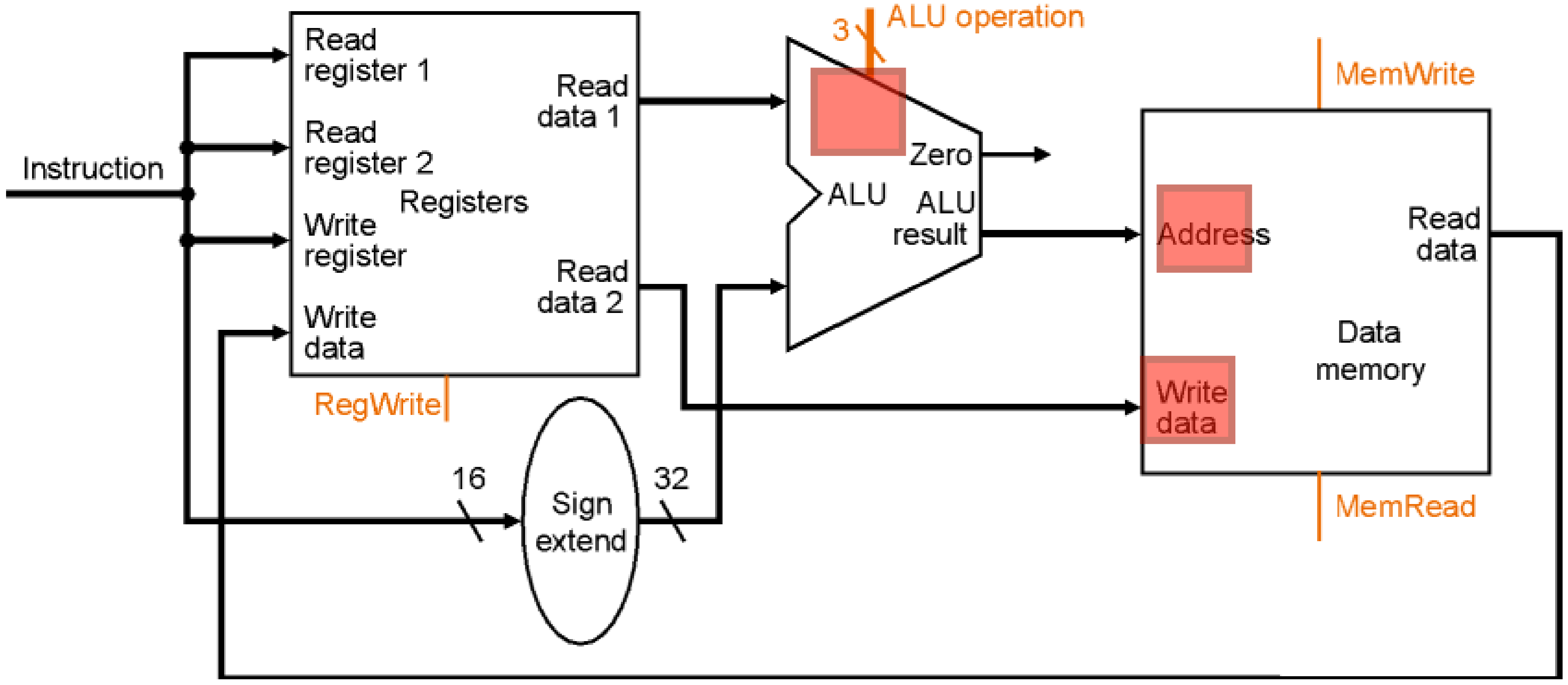
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Store Operations

- Mem[R[rs] + SignExt[imm16]] <- R[rt] Example: *sw   rt, rs, imm16*

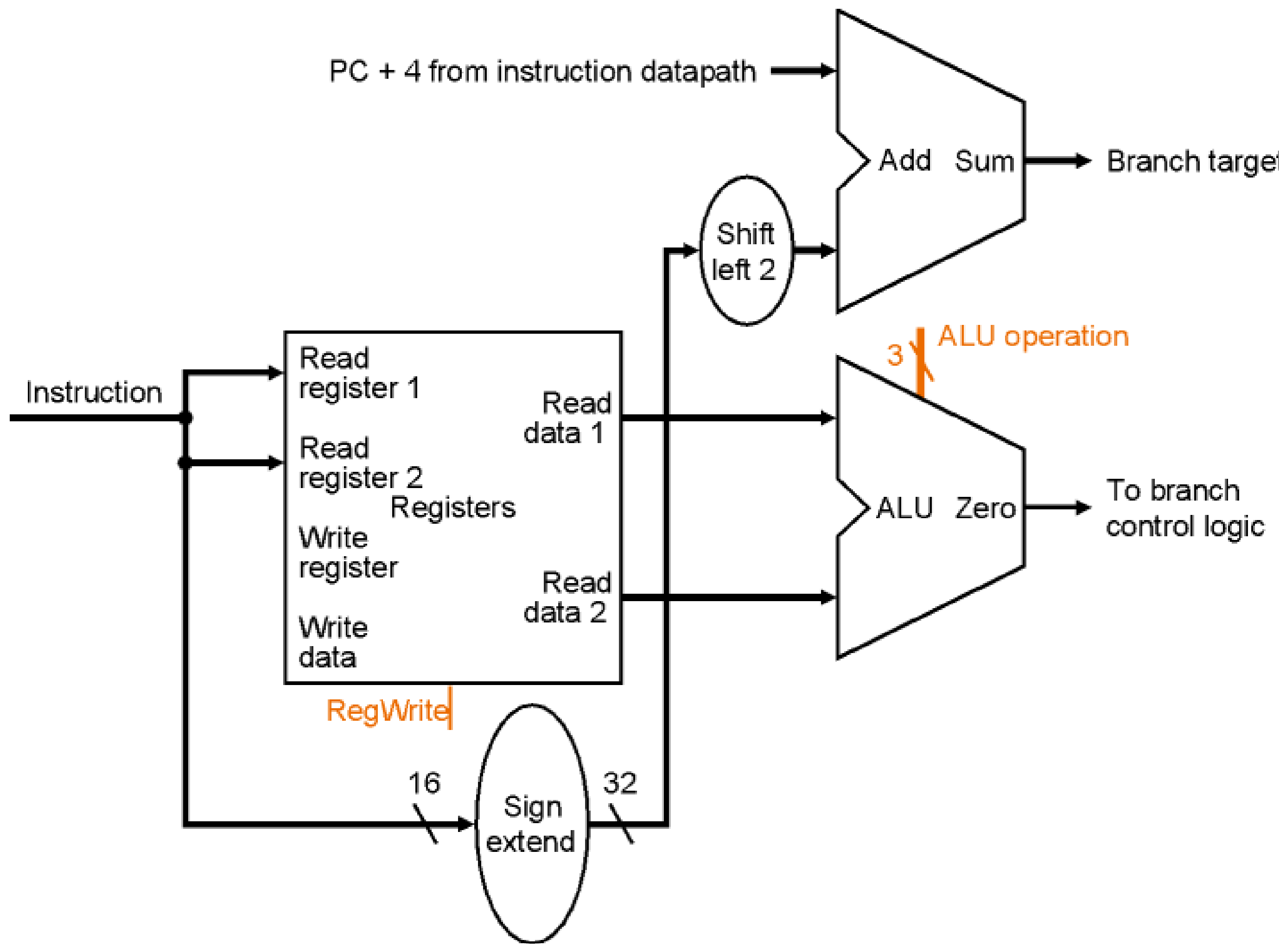
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

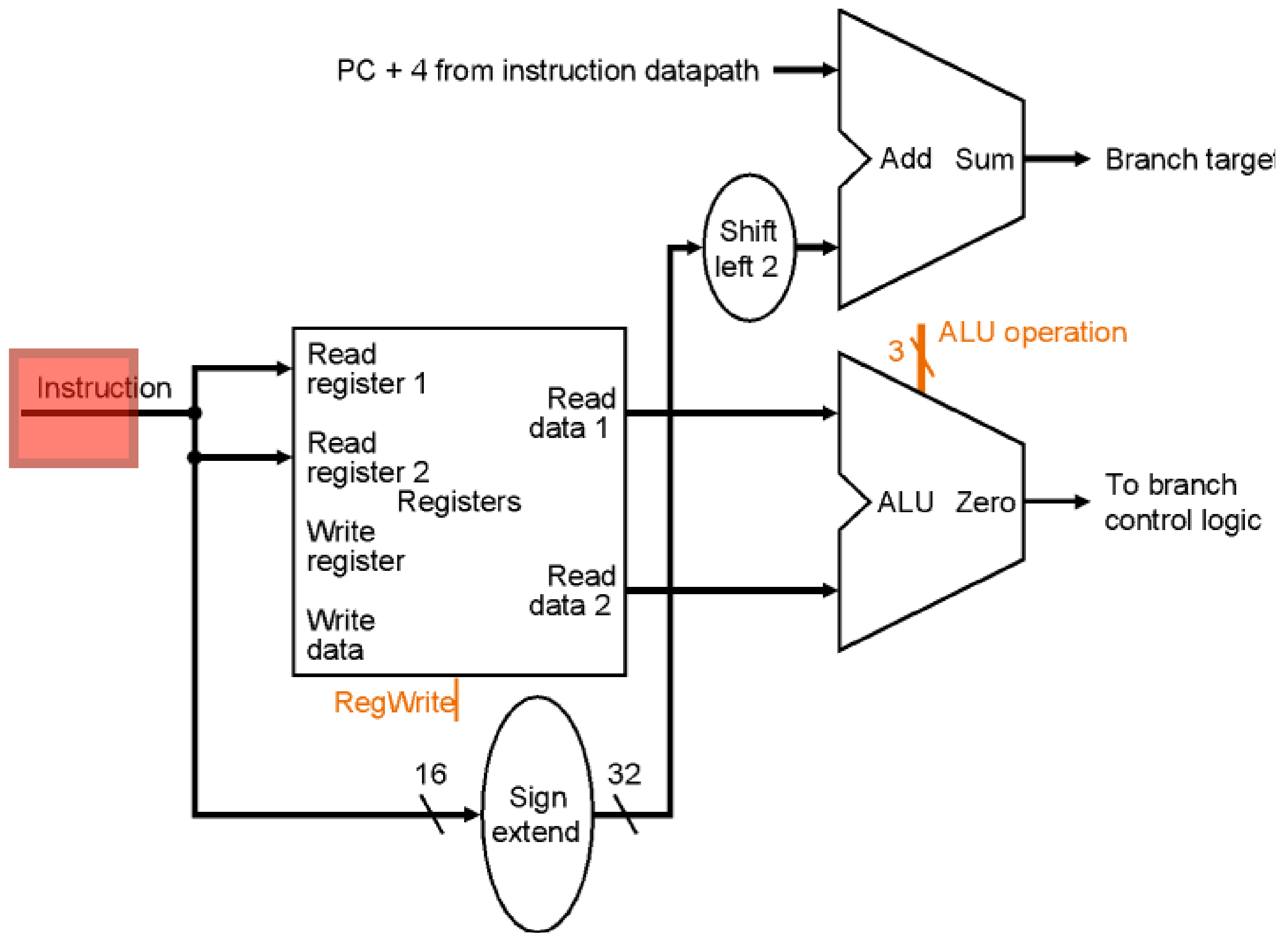
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

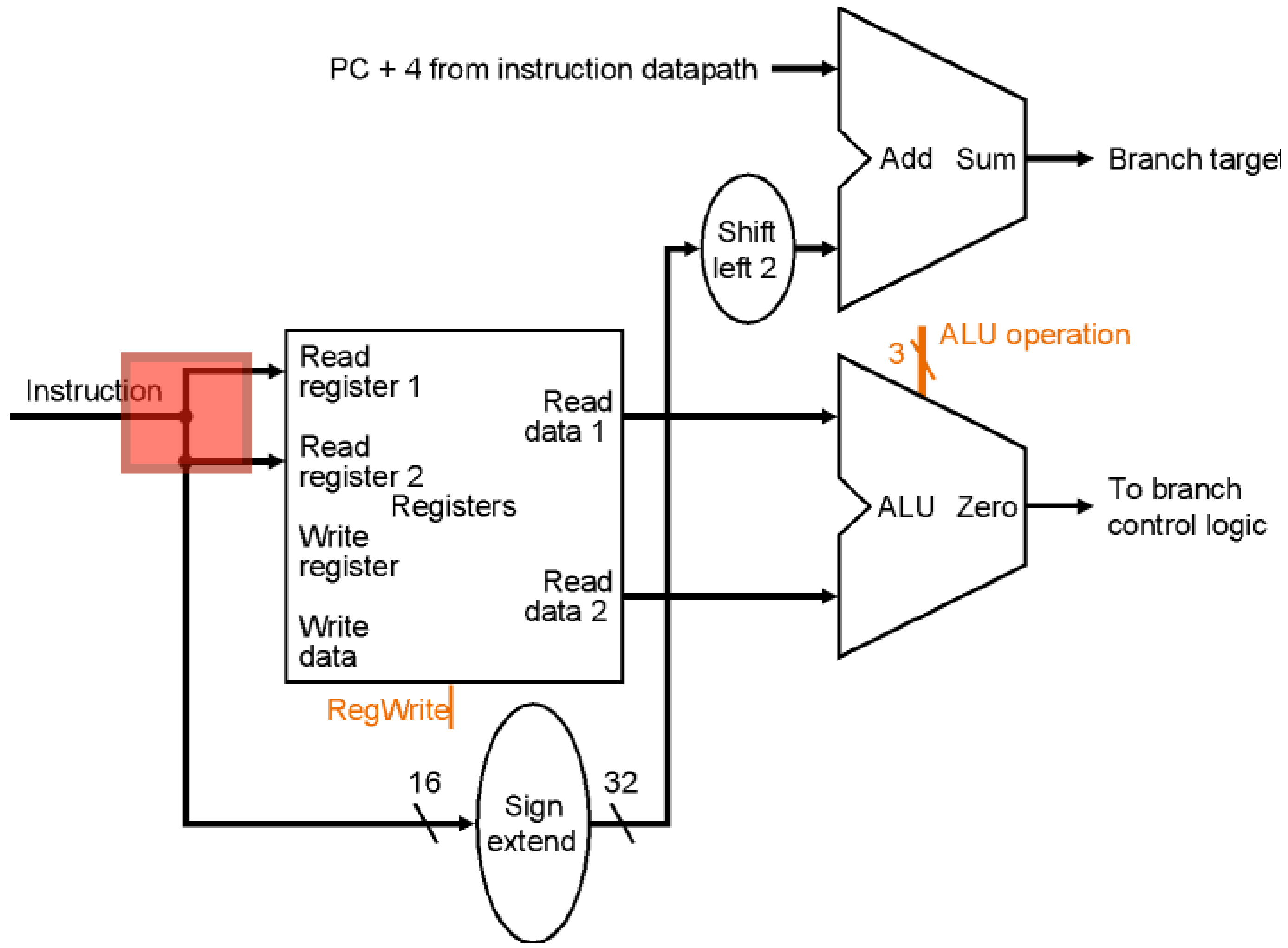
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

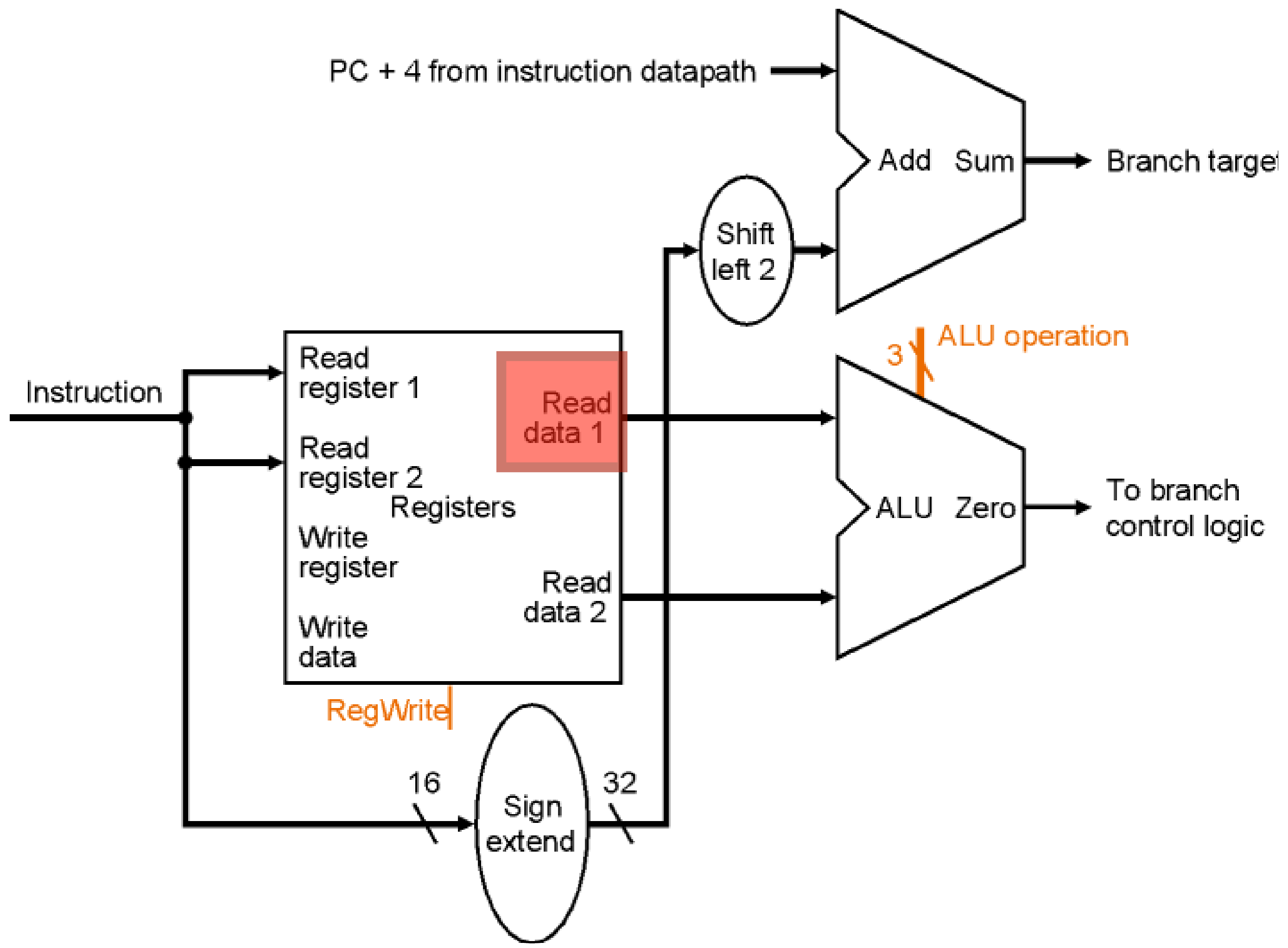
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq   rs, rt, imm16*

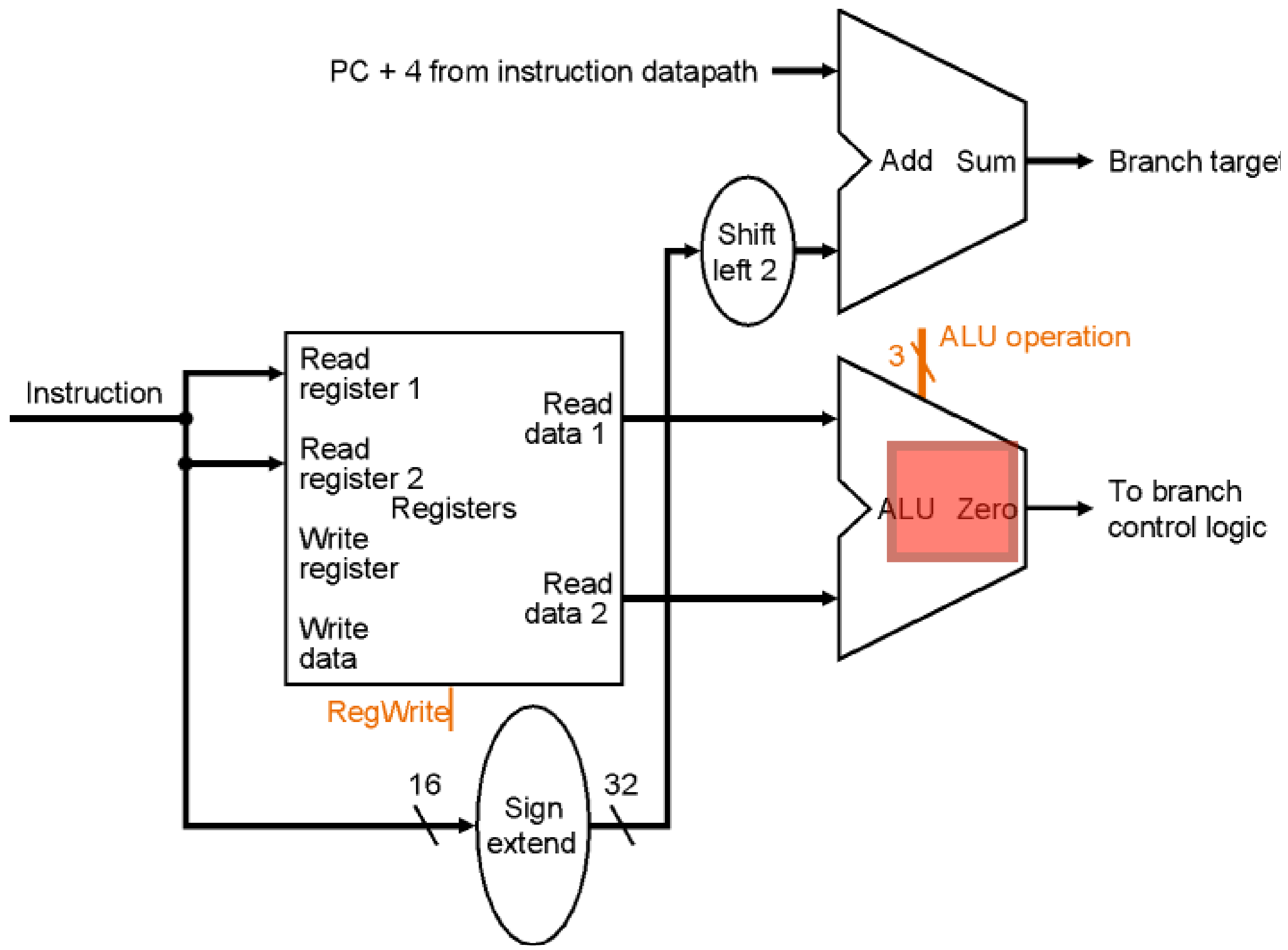
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

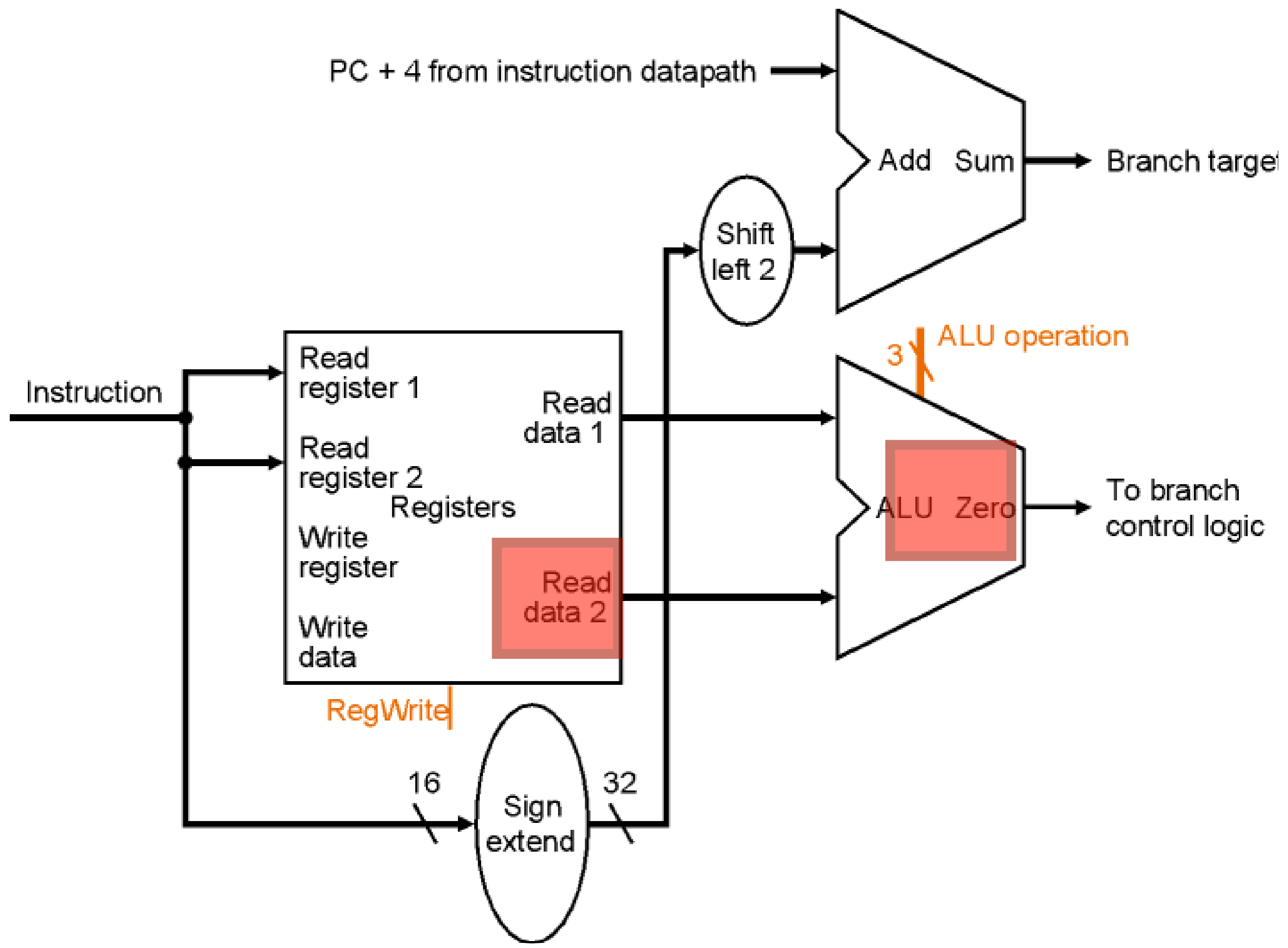
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq   rs, rt, imm16*

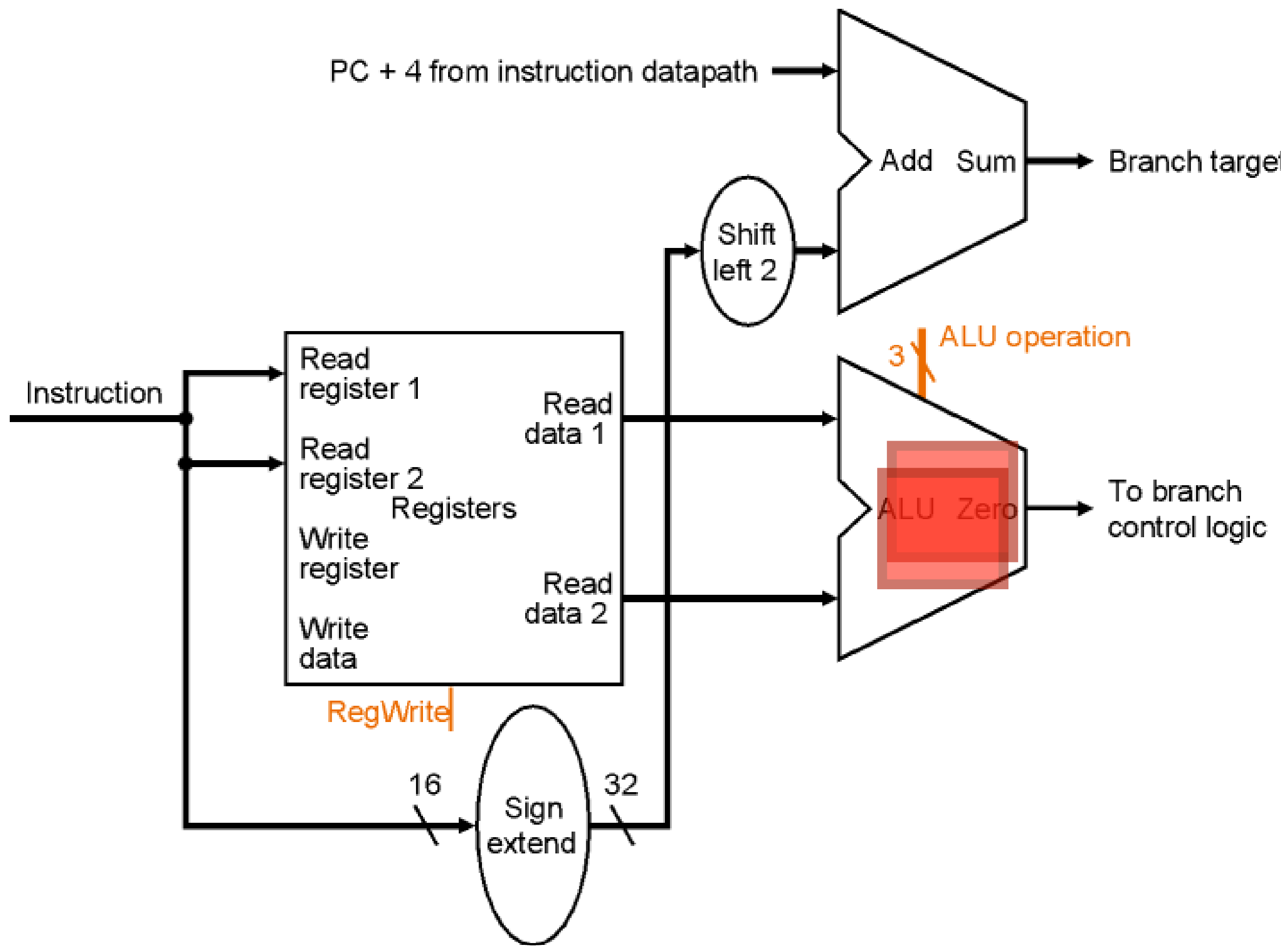
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

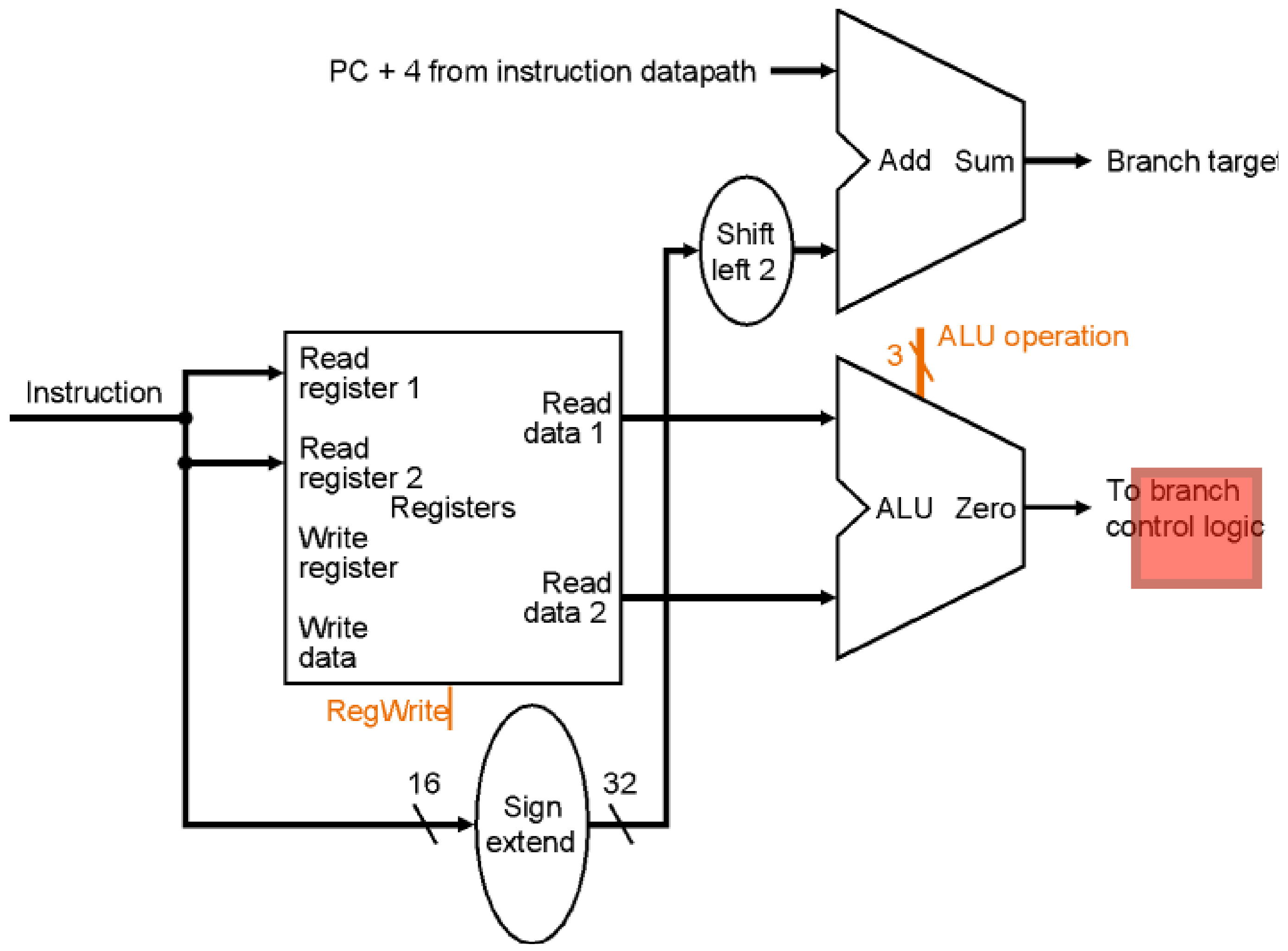
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

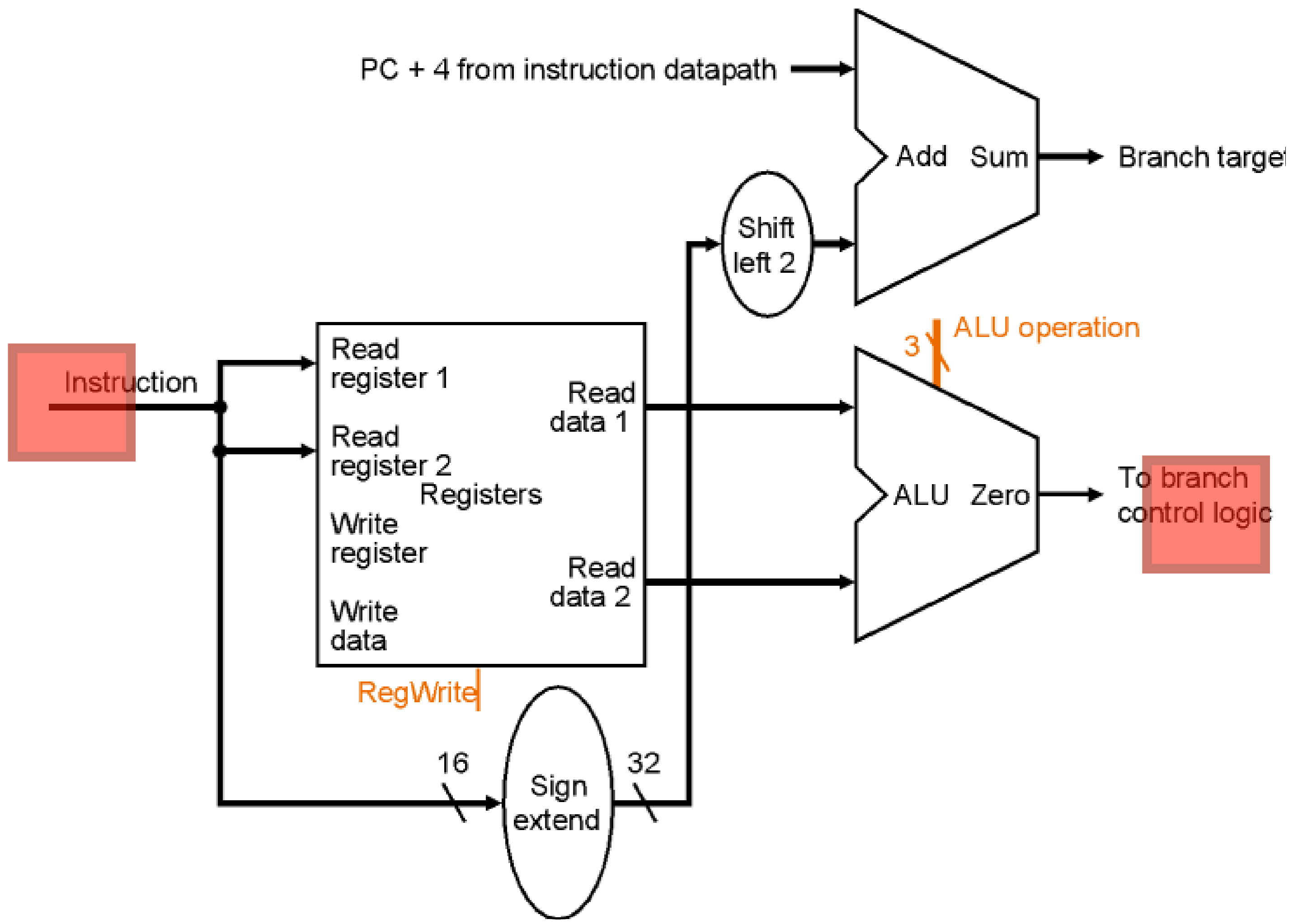
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

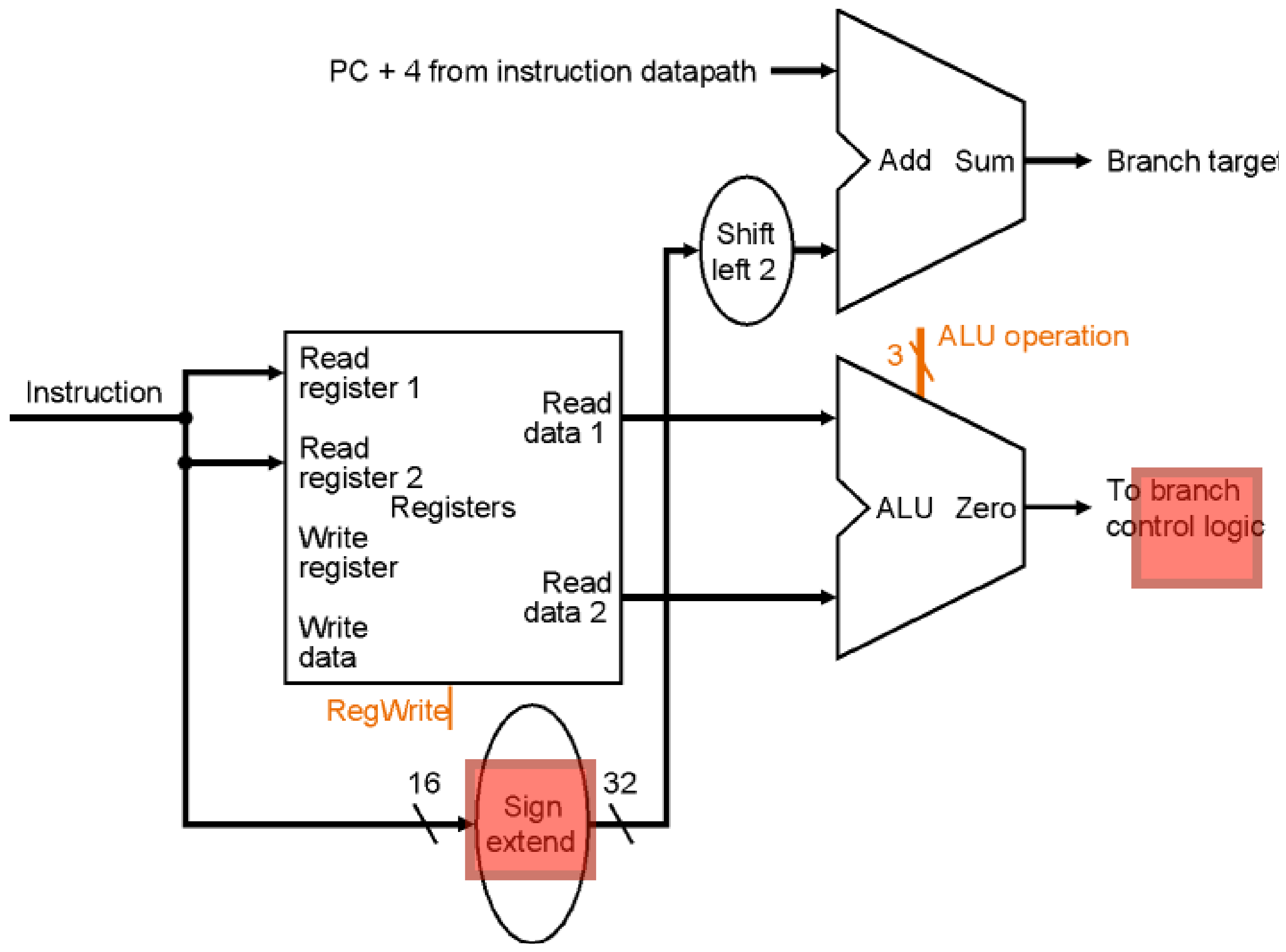
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

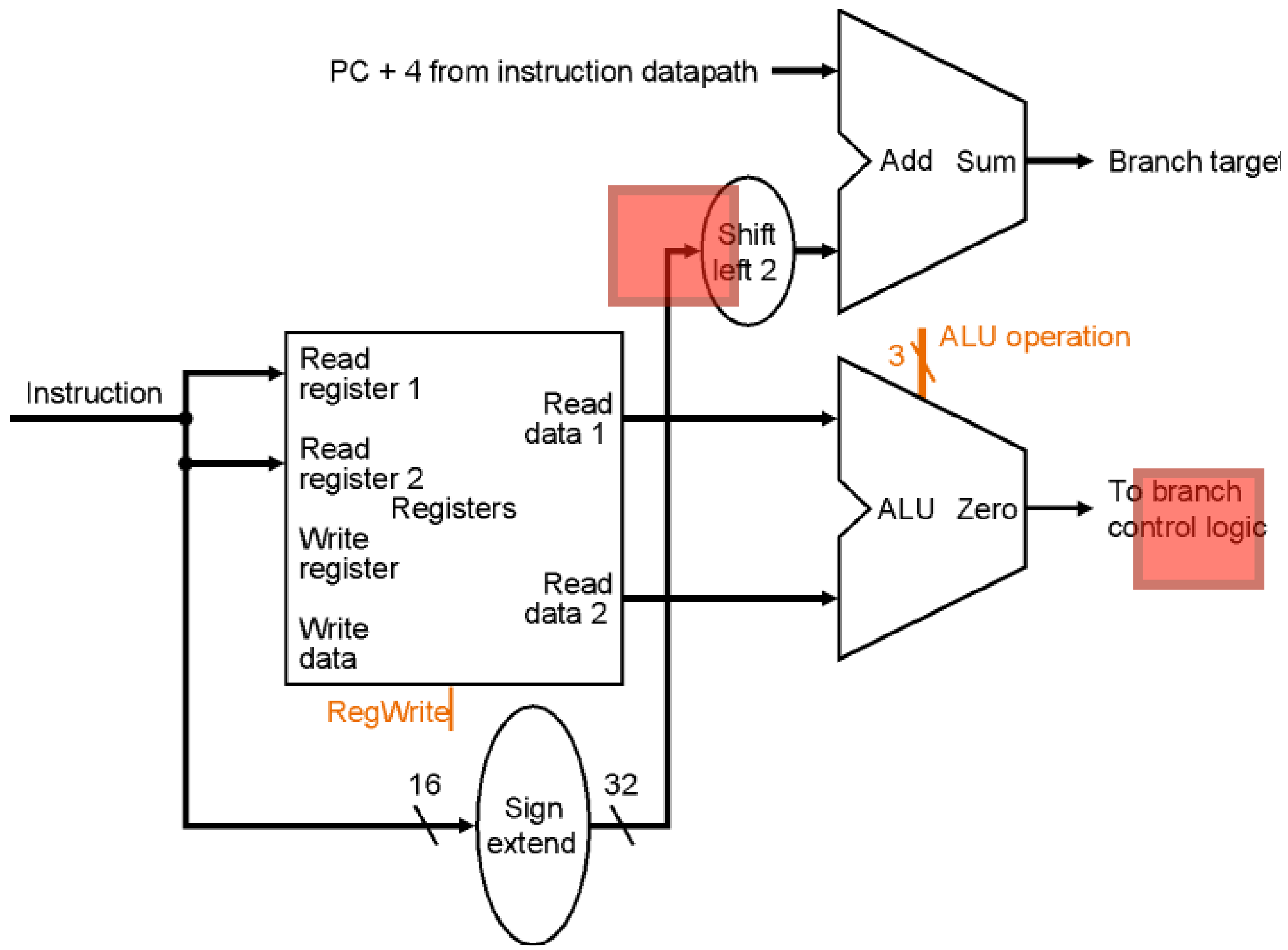
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq   rs, rt, imm16*

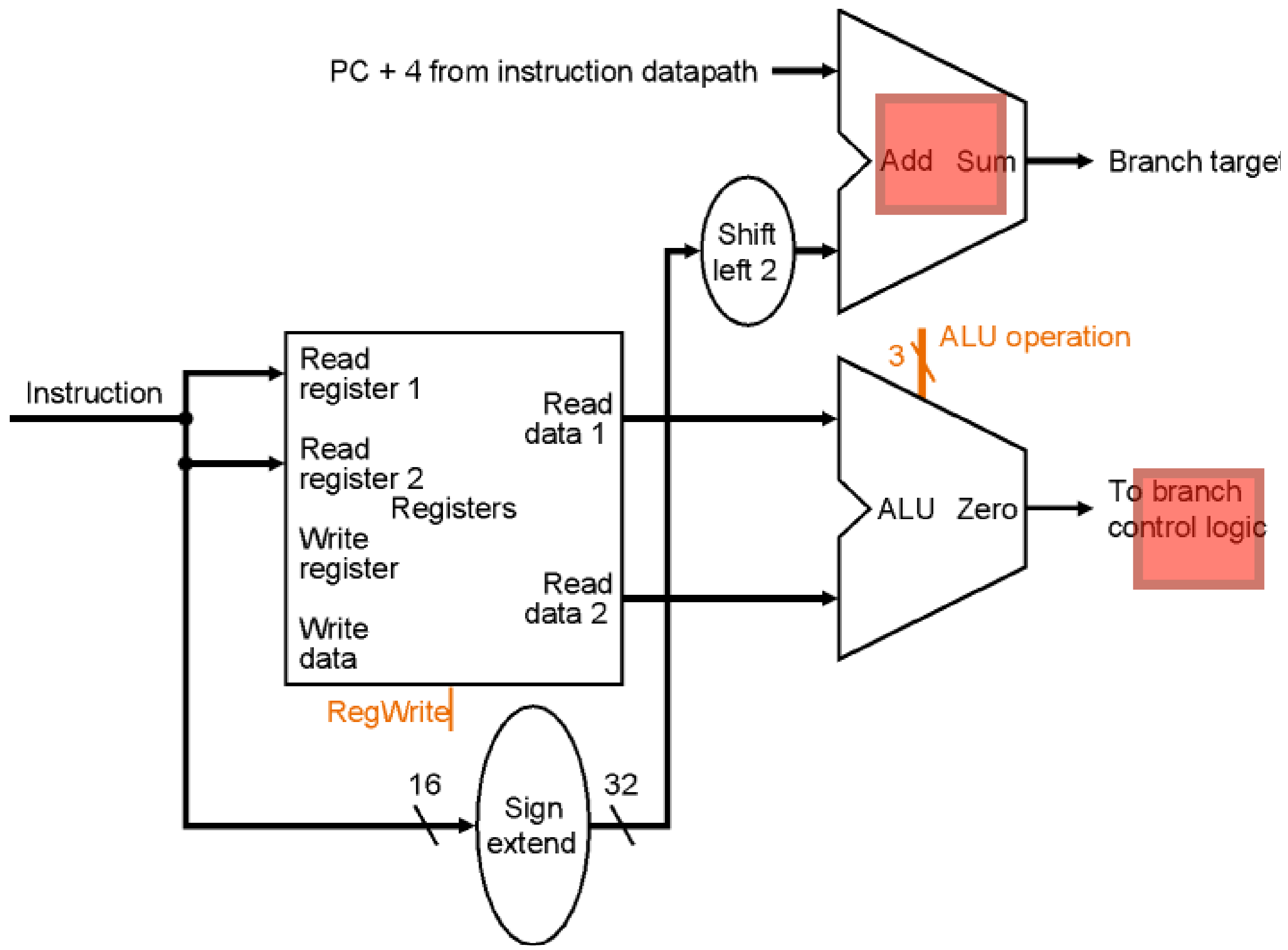
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

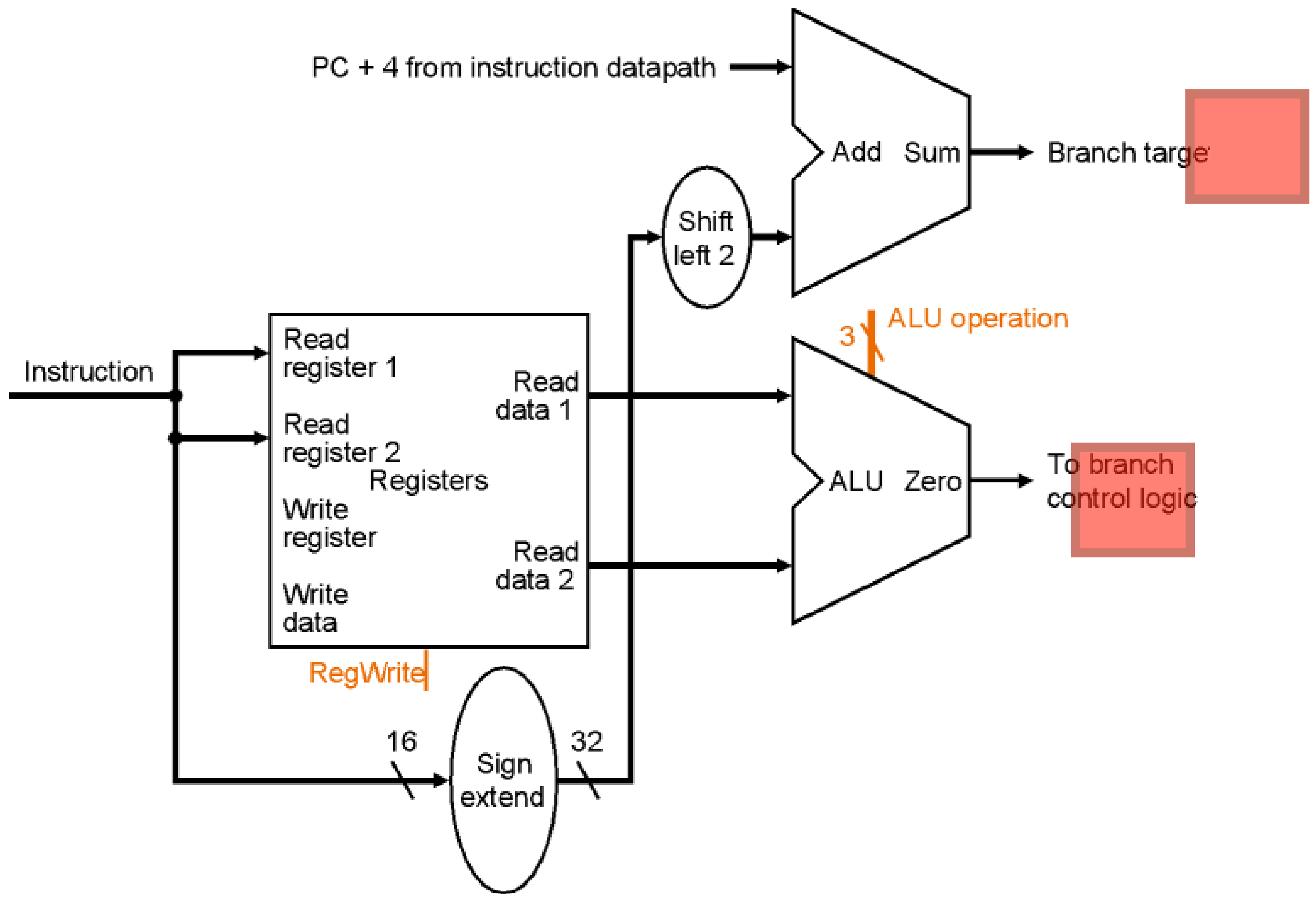
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



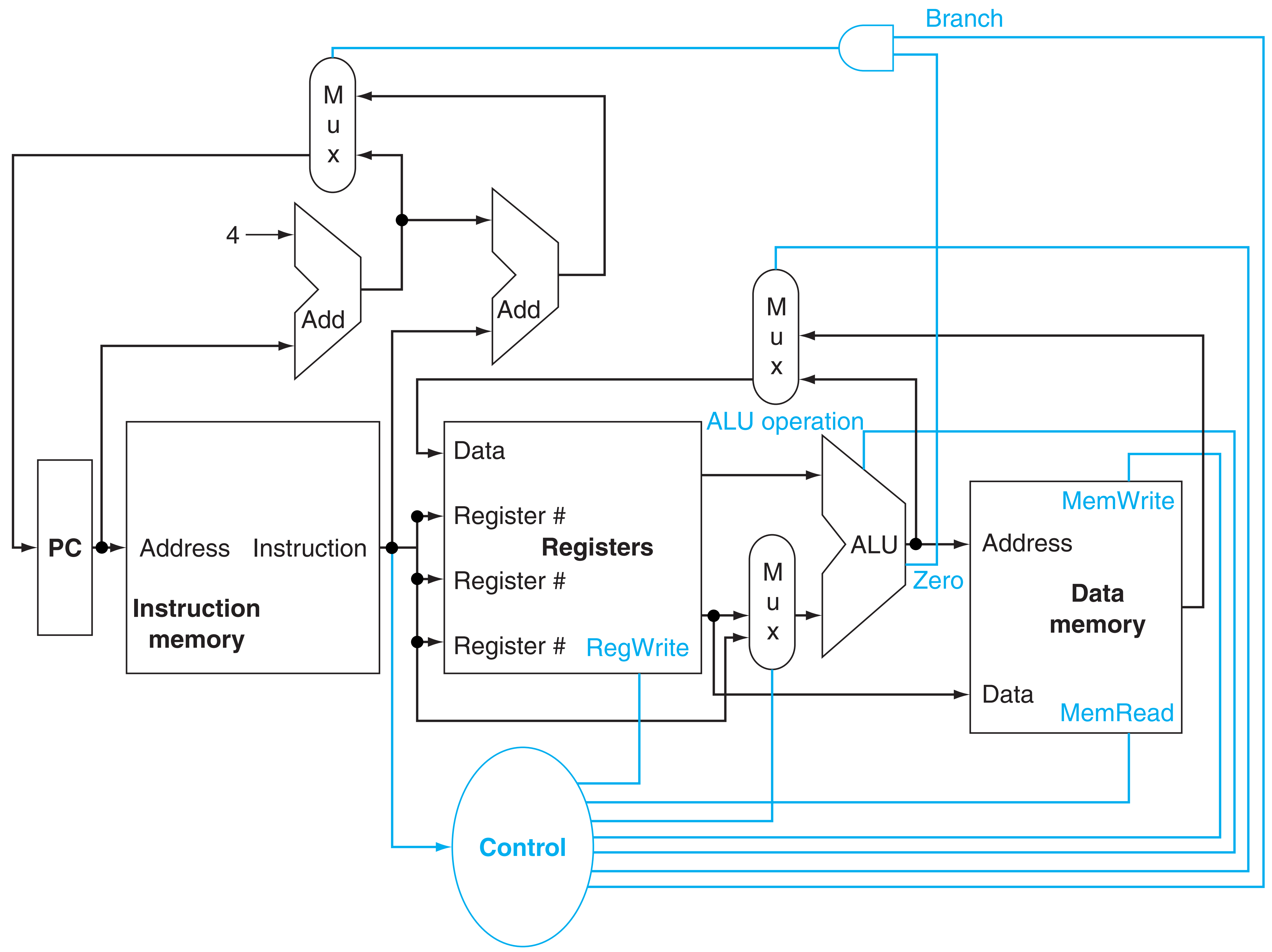
Datapath for Branch Operations

- Z <- (rs == rt); if Z, PC = PC+4+imm16; else PC = PC+4
- *beq rs, rt, imm16*

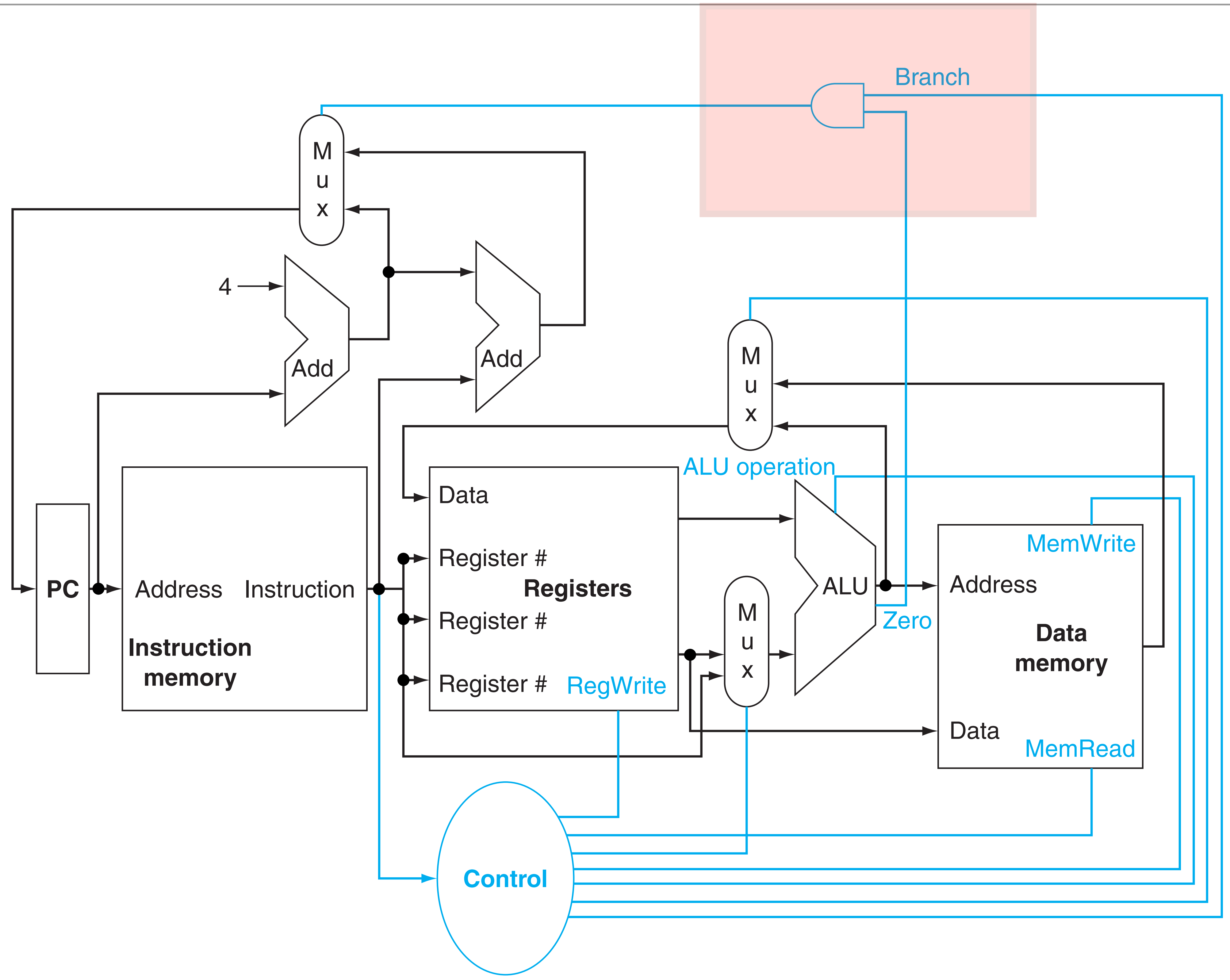
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	immediate / offset



Control Logic??



Control Logic??



Binary Arithmetic for the Next Address

Binary Arithmetic for the Next Address

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4

Binary Arithmetic for the Next Address

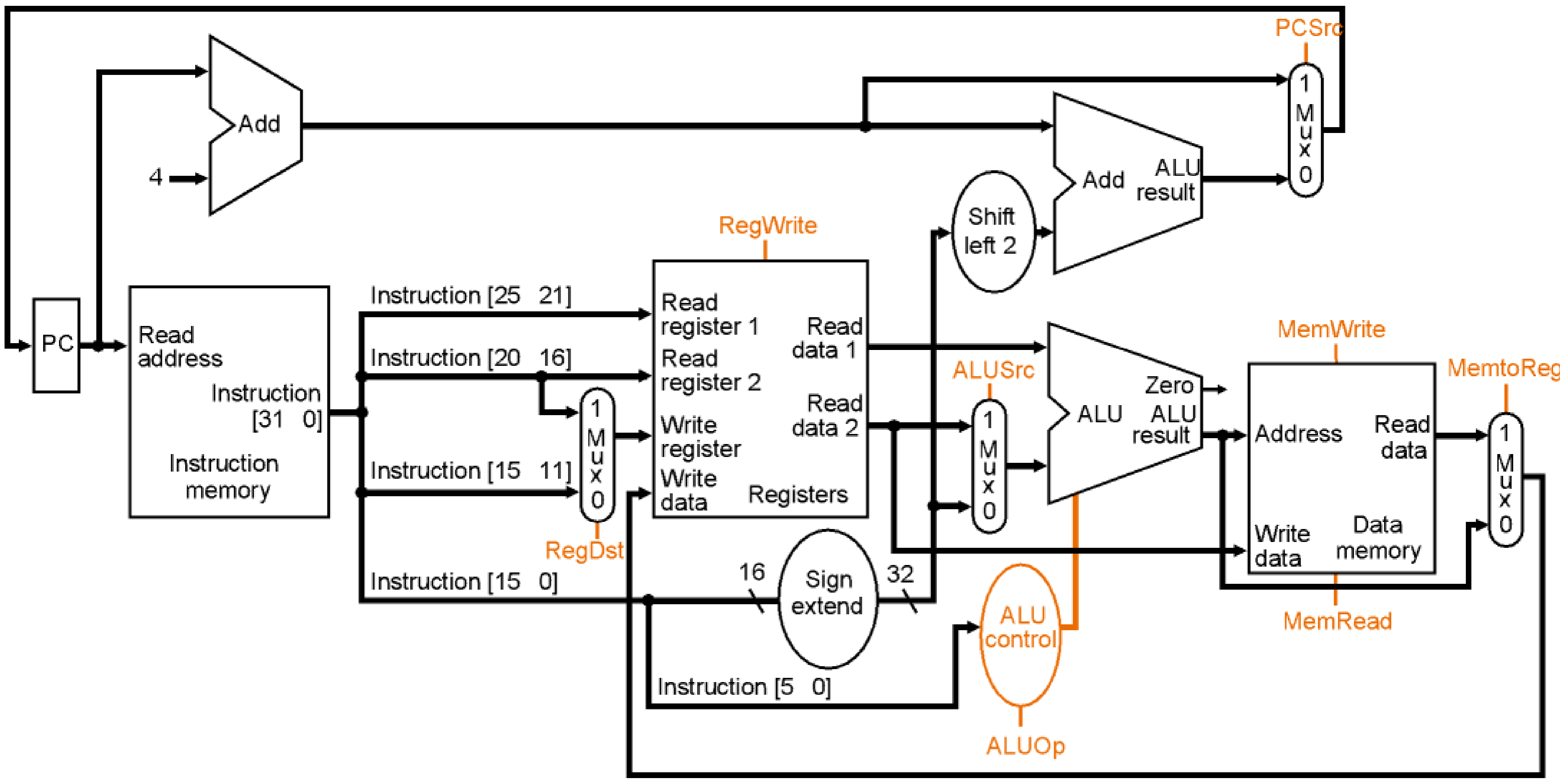
- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4
- The magic number "4" always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs

Binary Arithmetic for the Next Address

- In theory, the PC is a 32-bit byte address into the instruction memory:
  - Sequential operation: PC<31:0> = PC<31:0> + 4
  - Branch operation: PC<31:0> = PC<31:0> + 4 + SignExt[Imm16] * 4
- The magic number "4" always comes up because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
  - The 2 LSBs of the 32-bit PC are always zeros
  - There is no reason to have hardware to keep the 2 LSBs
- In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - In either case: Instruction Memory Address = PC<31:2> concat "00"

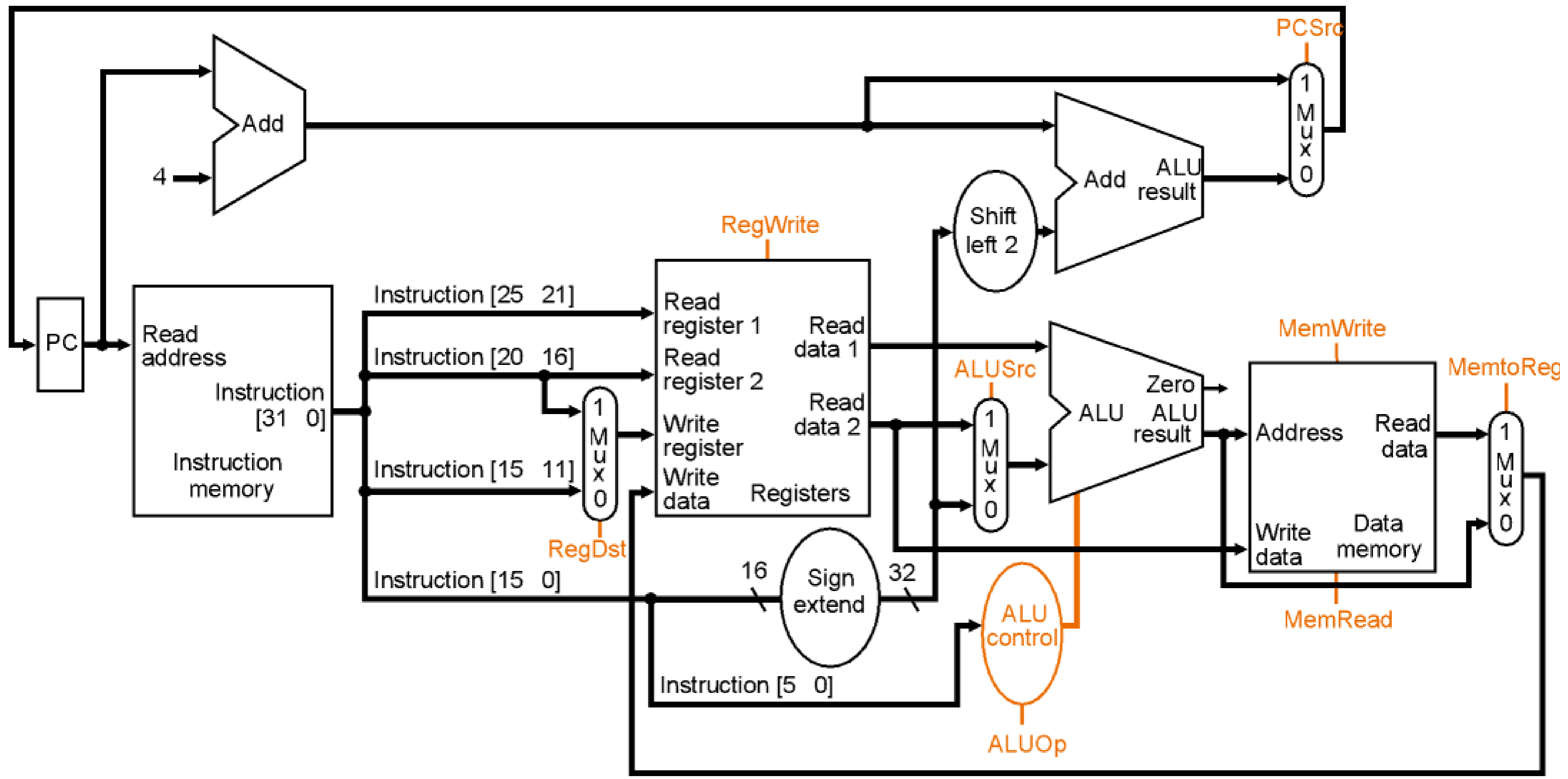
Putting it All Together: A Single Cycle Datapath

- We have everything except control signals



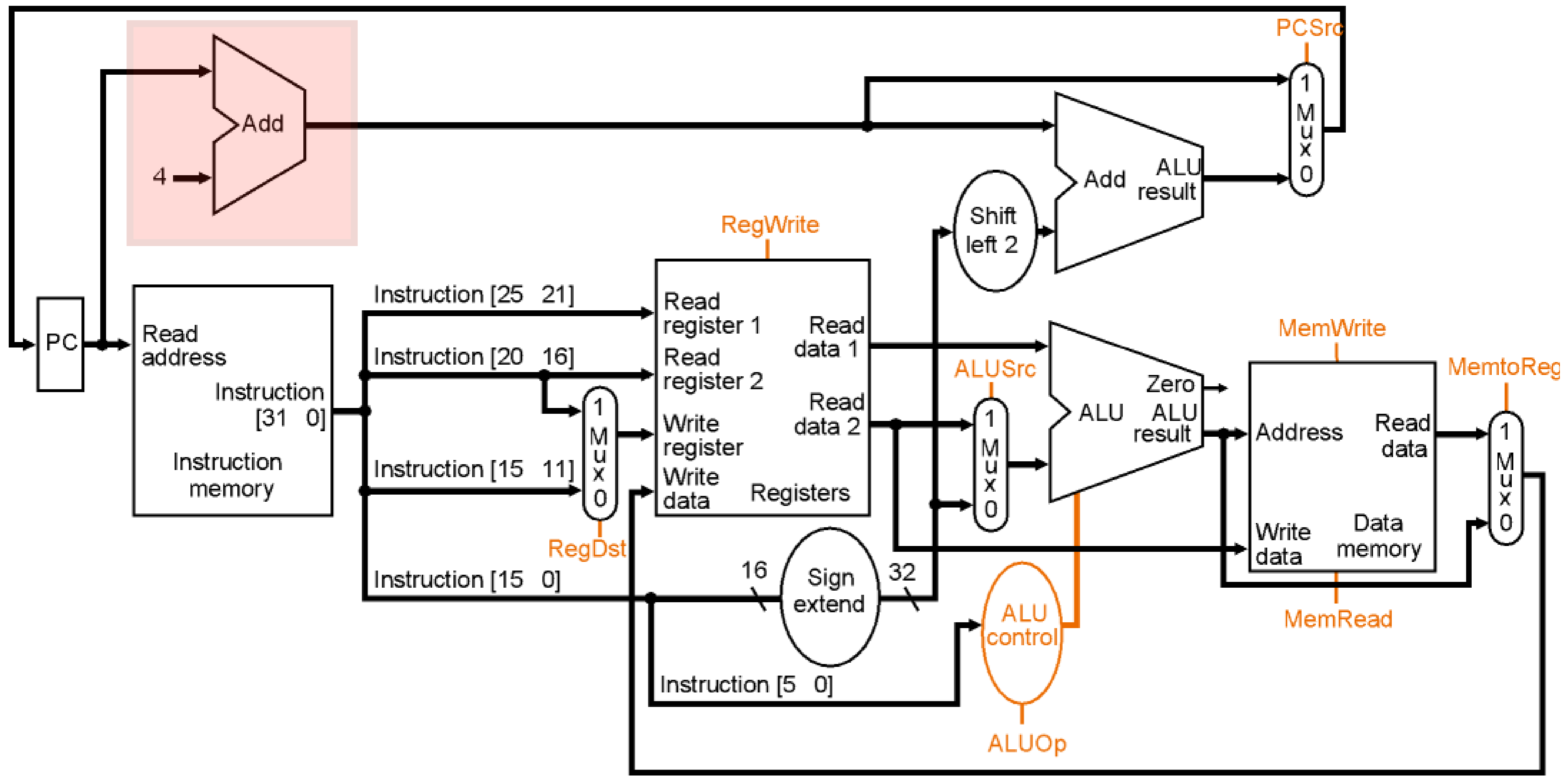
GAME: GUESS THE FUNCTION!!

- We have everything except control signals



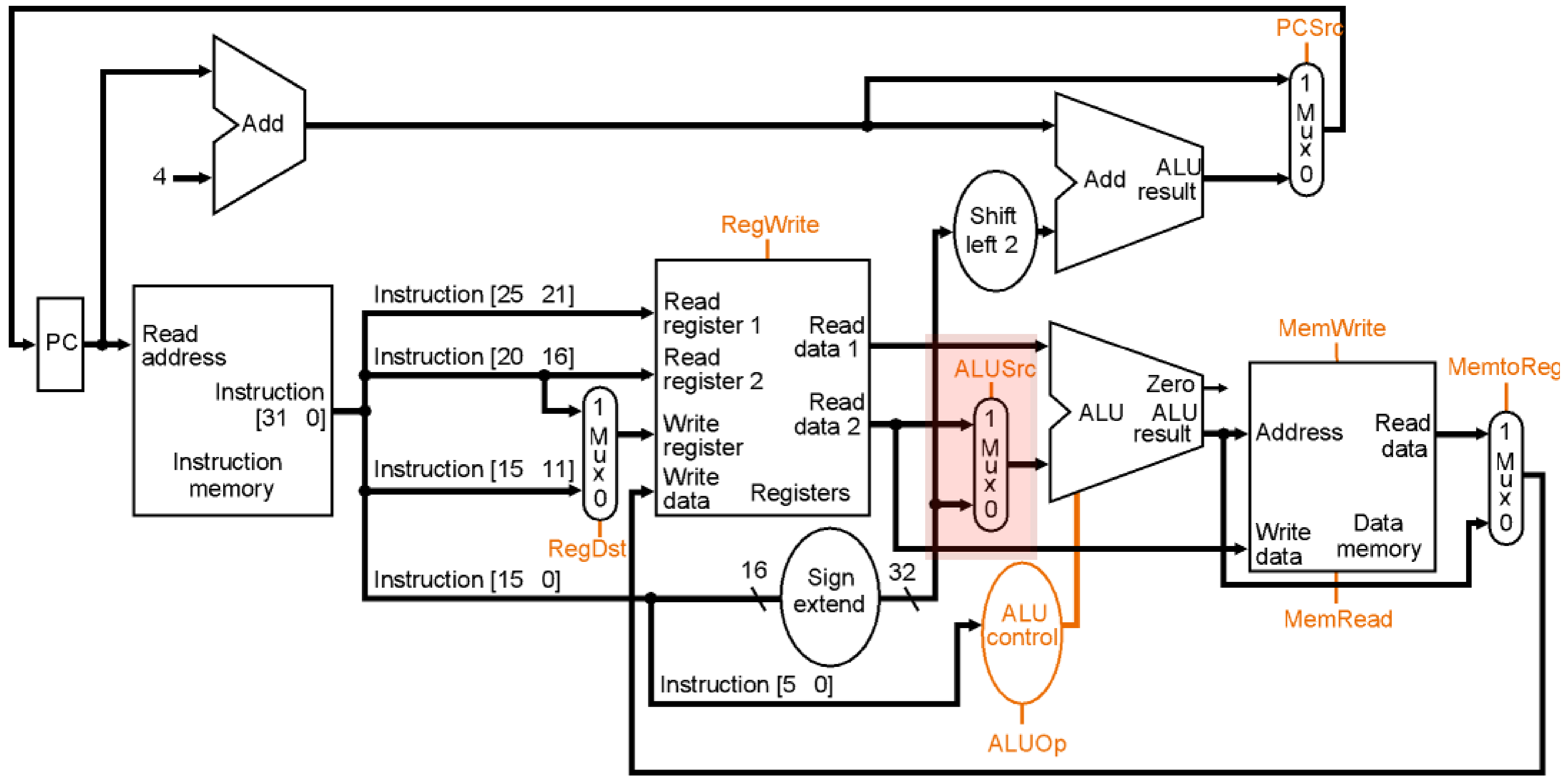
GAME: GUESS THE FUNCTION!!

- We have everything except control signals



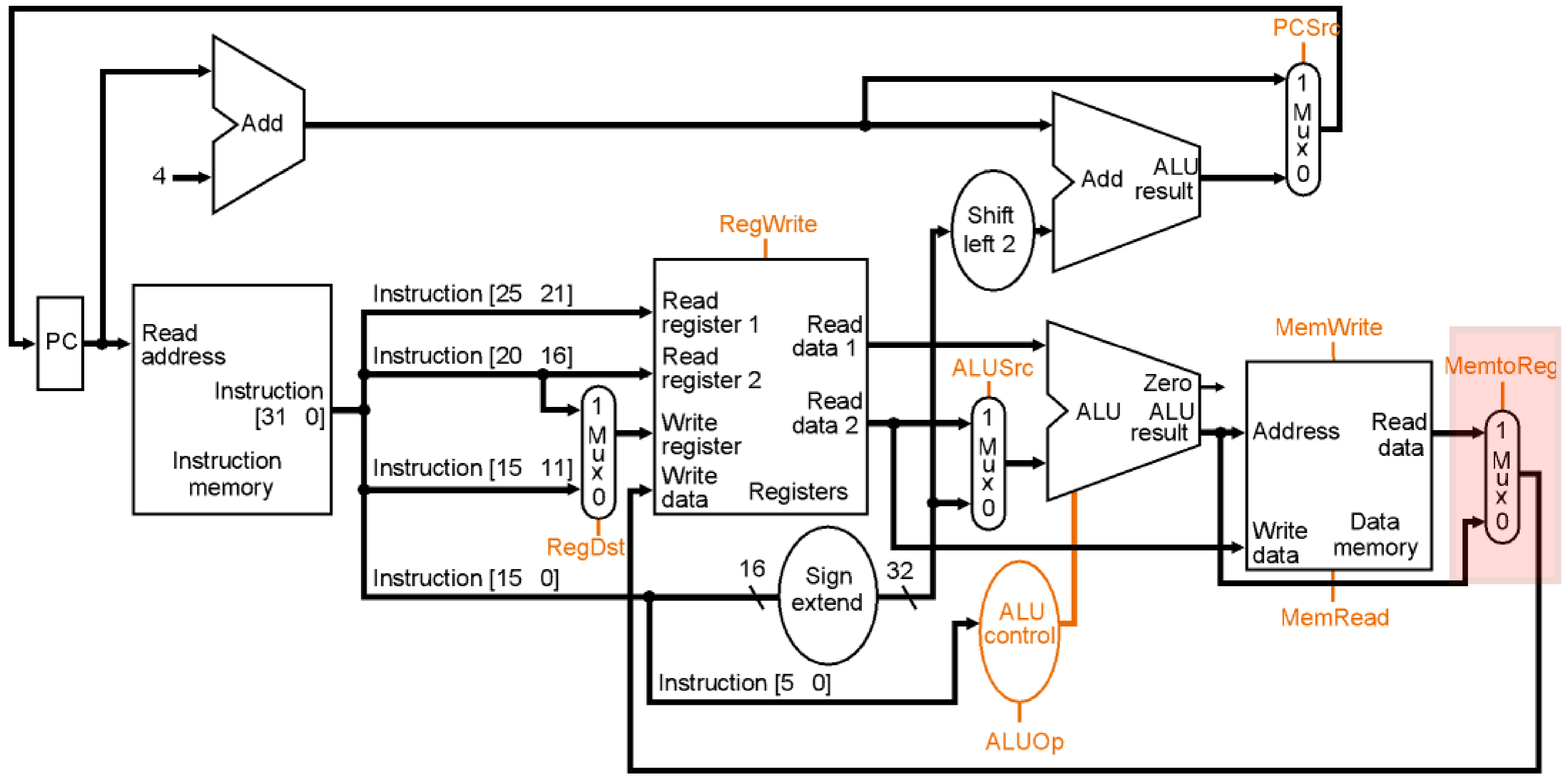
GAME: GUESS THE FUNCTION!!

- We have everything except control signals



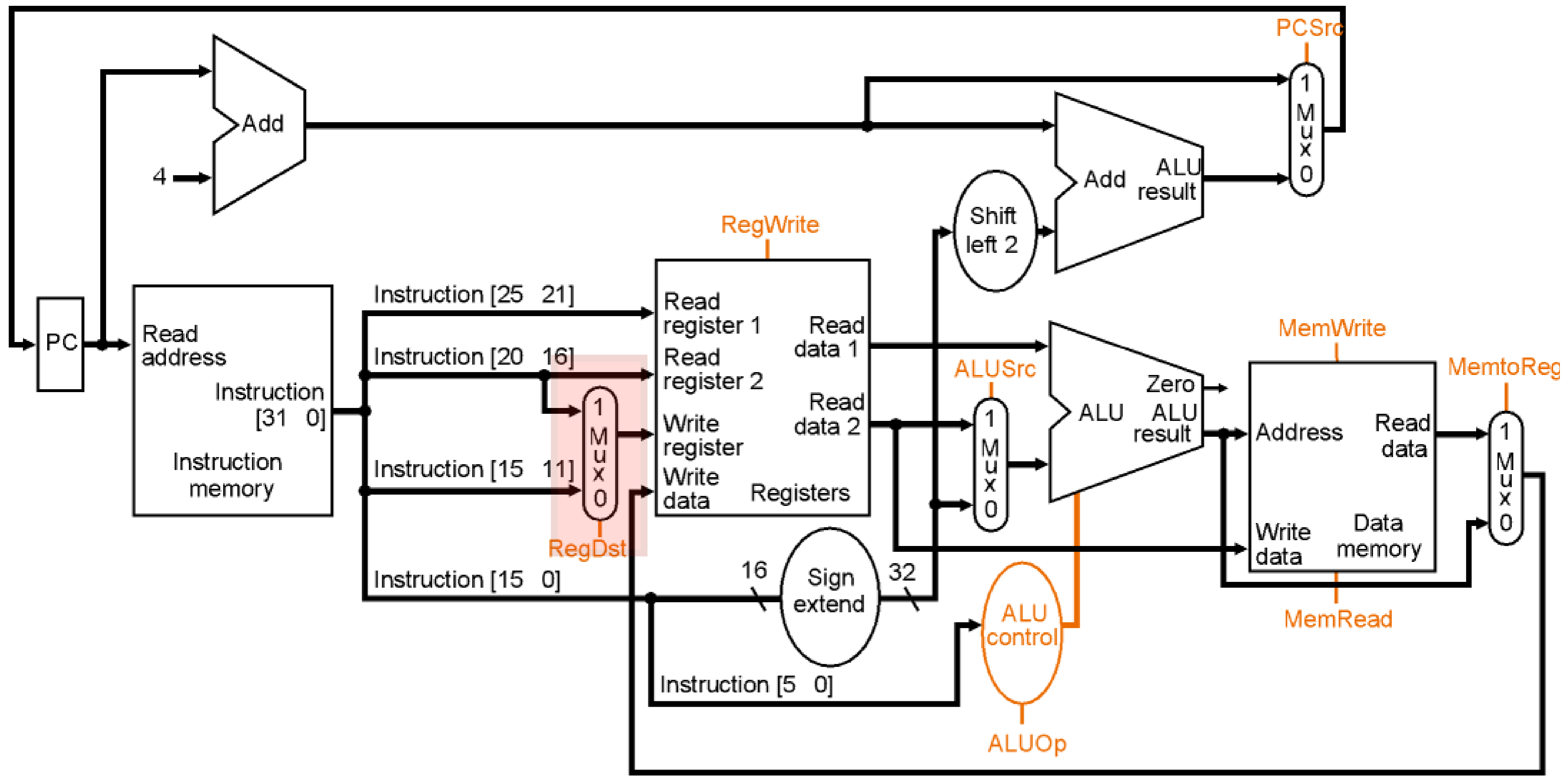
GAME: GUESS THE FUNCTION!!

- We have everything except control signals



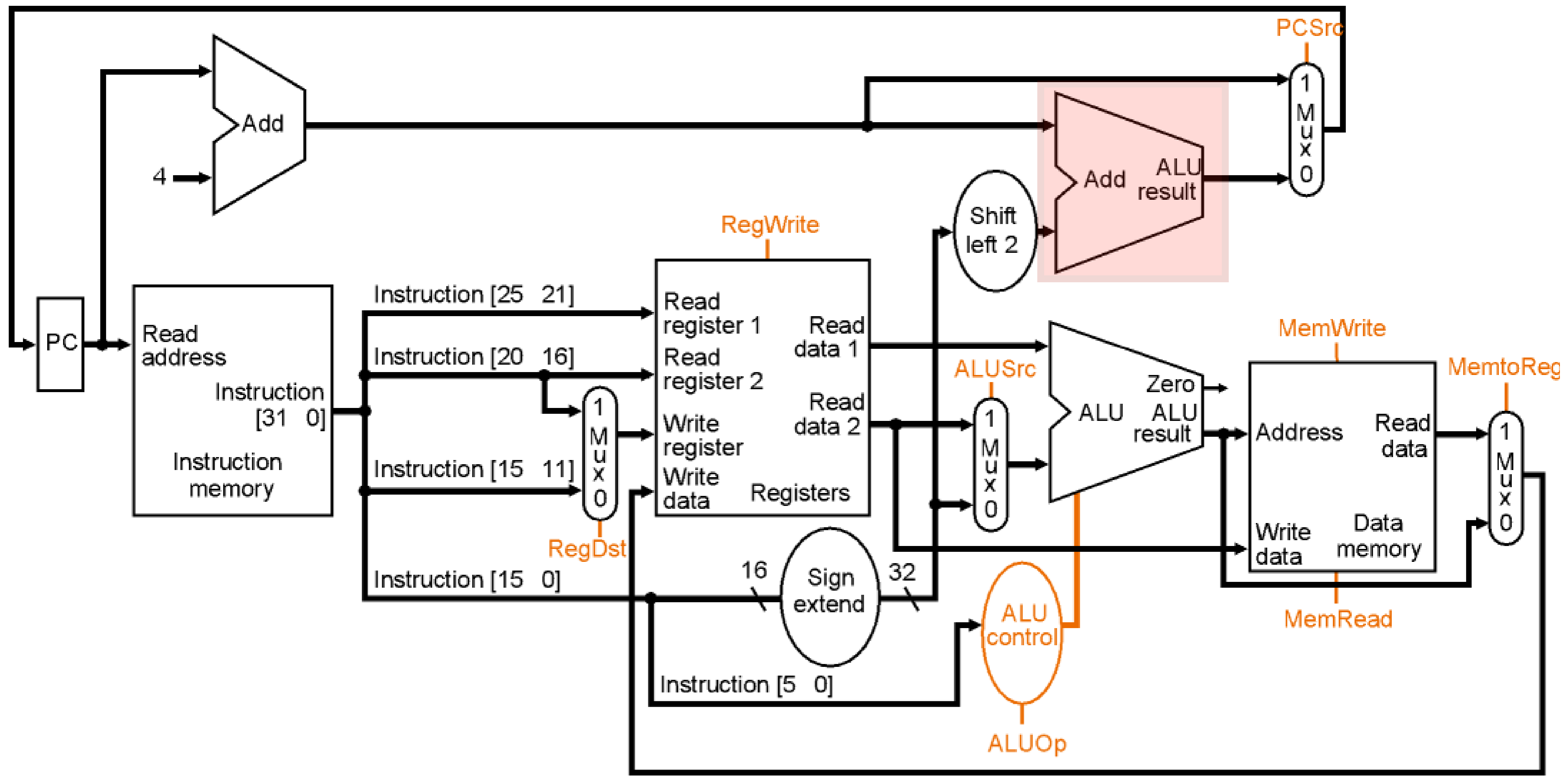
GAME: GUESS THE FUNCTION!!

- We have everything except control signals



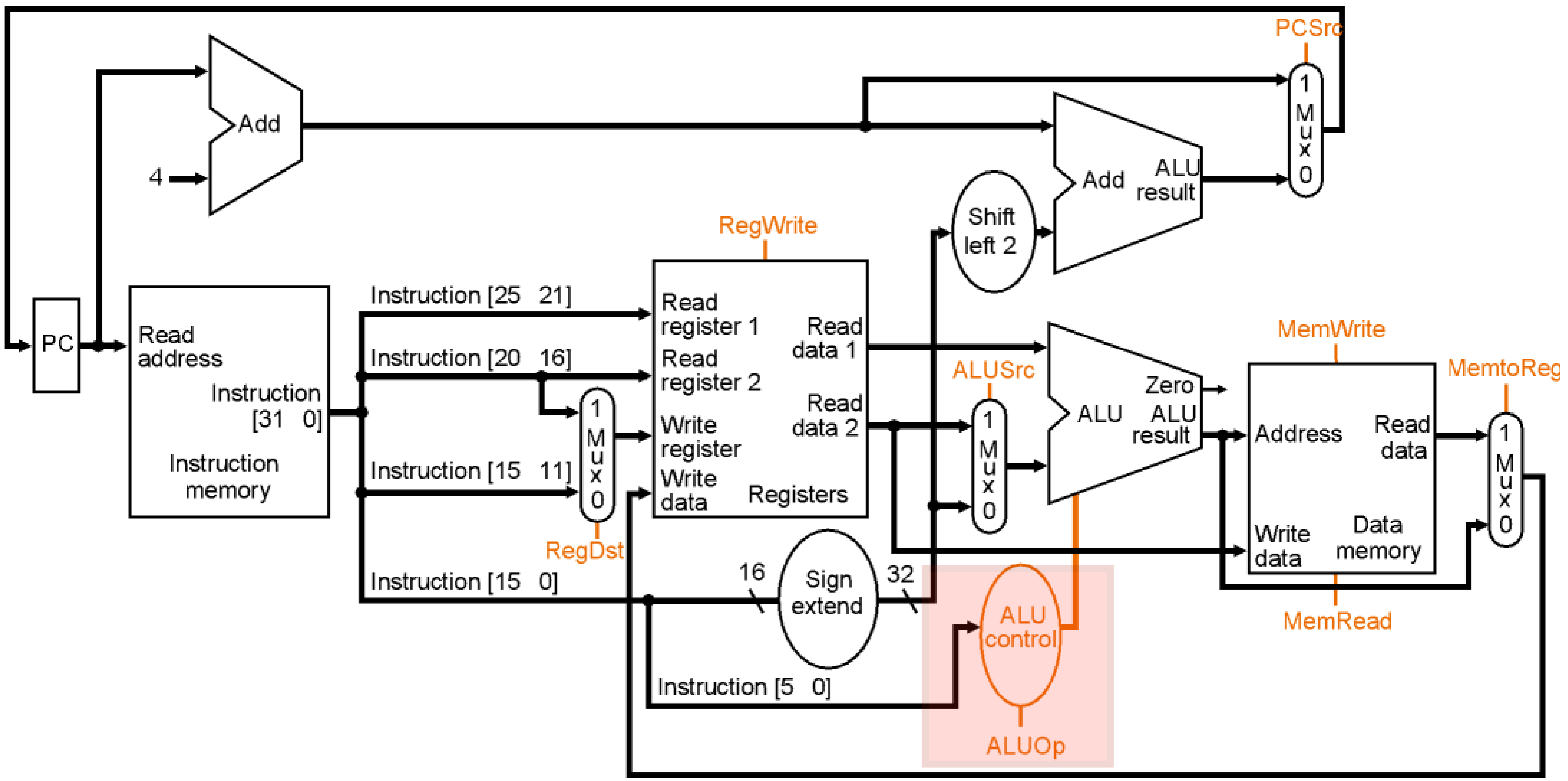
GAME: GUESS THE FUNCTION!!

- We have everything except control signals



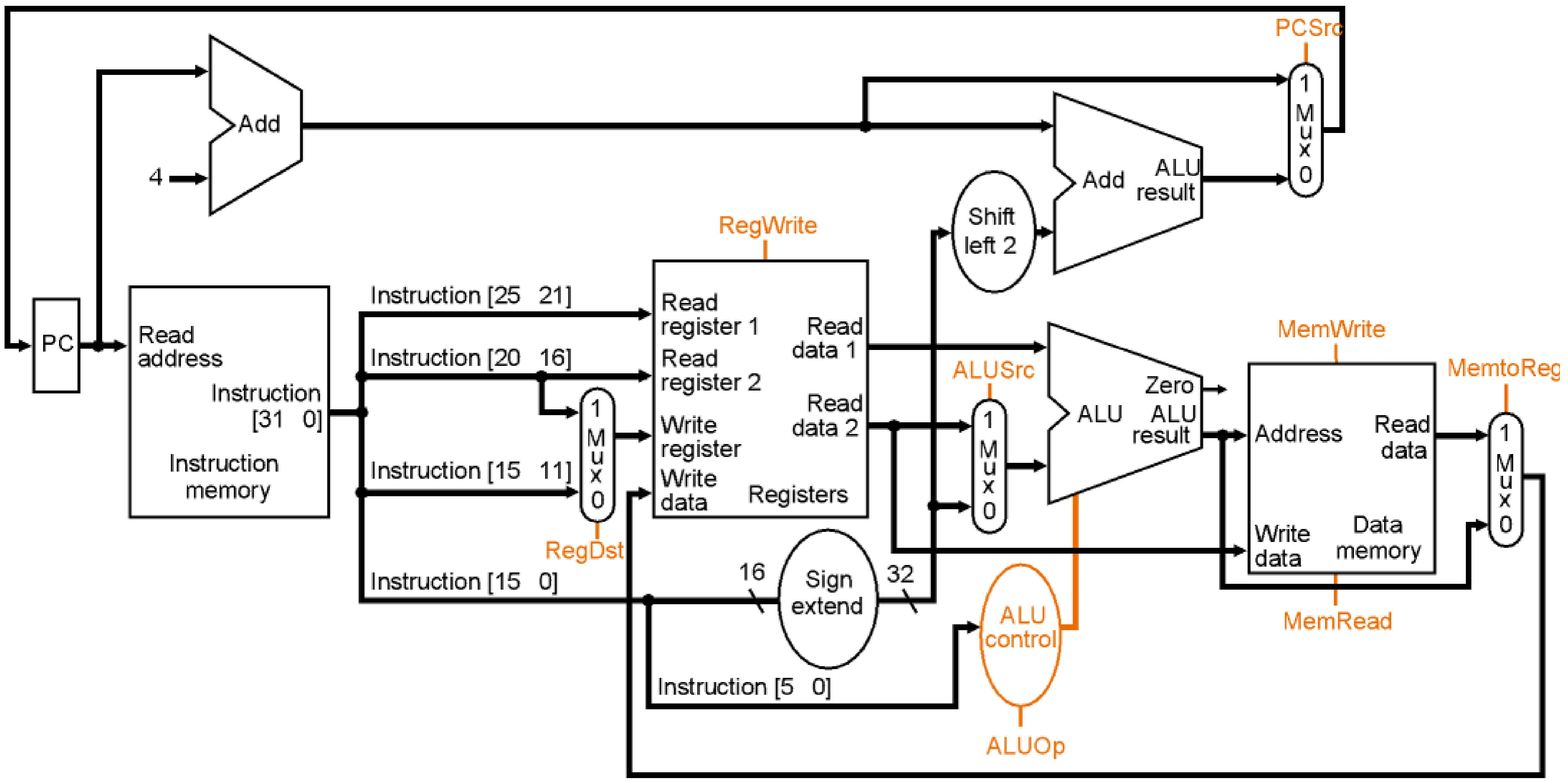
GAME: GUESS THE FUNCTION!!

- We have everything except control signals

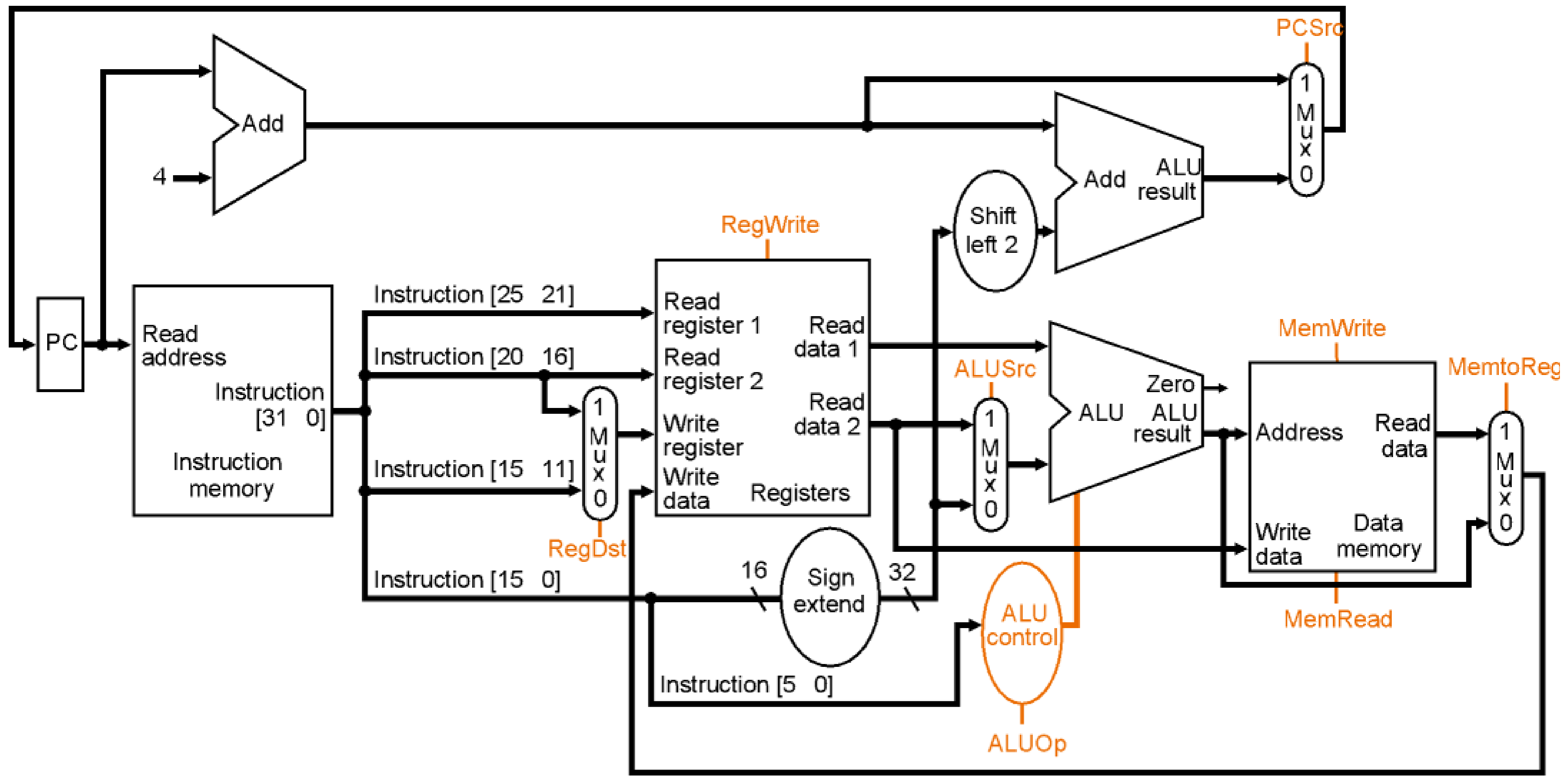


GAME: GUESS THE FUNCTION!!

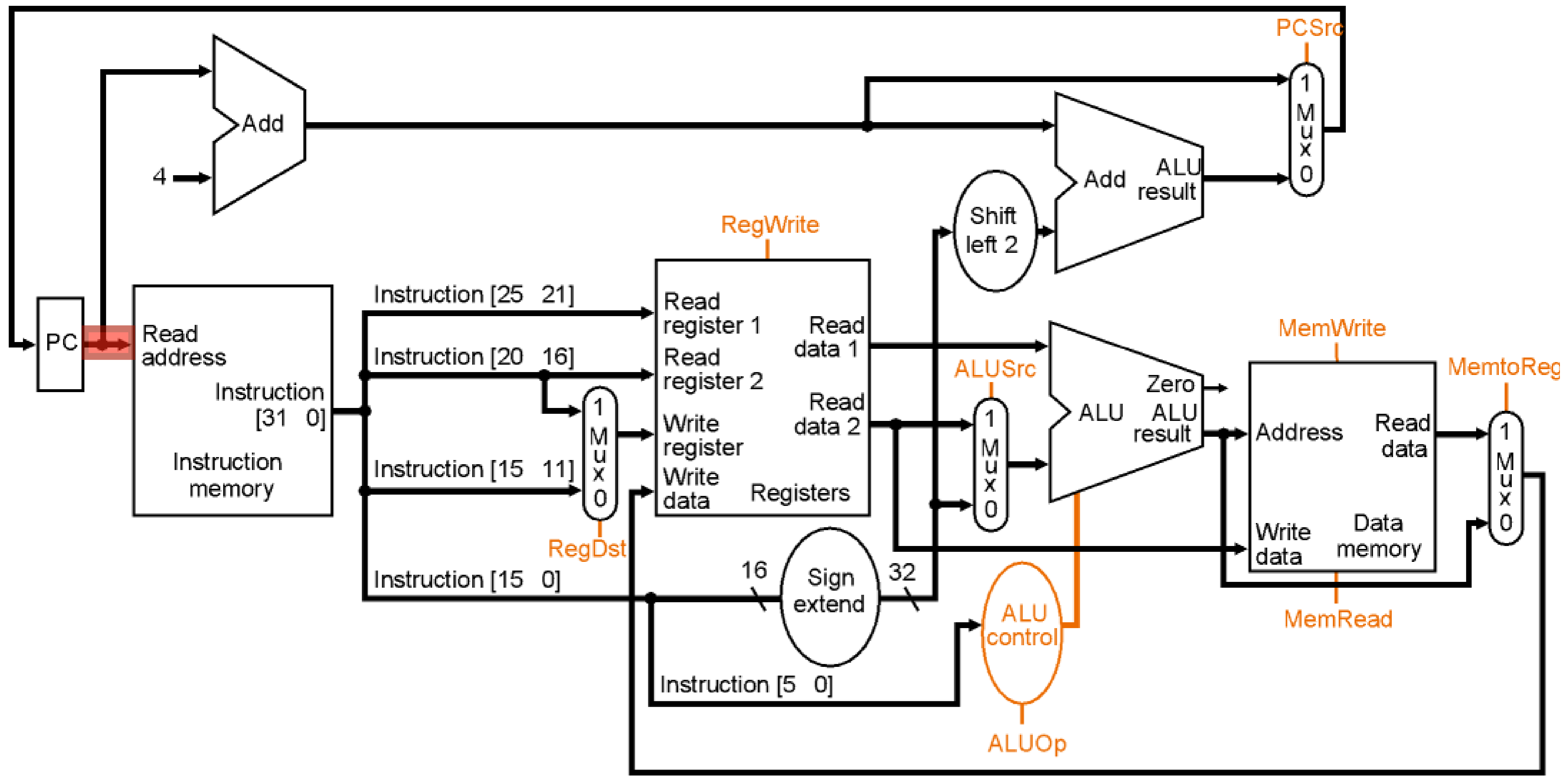
- We have everything except control signals



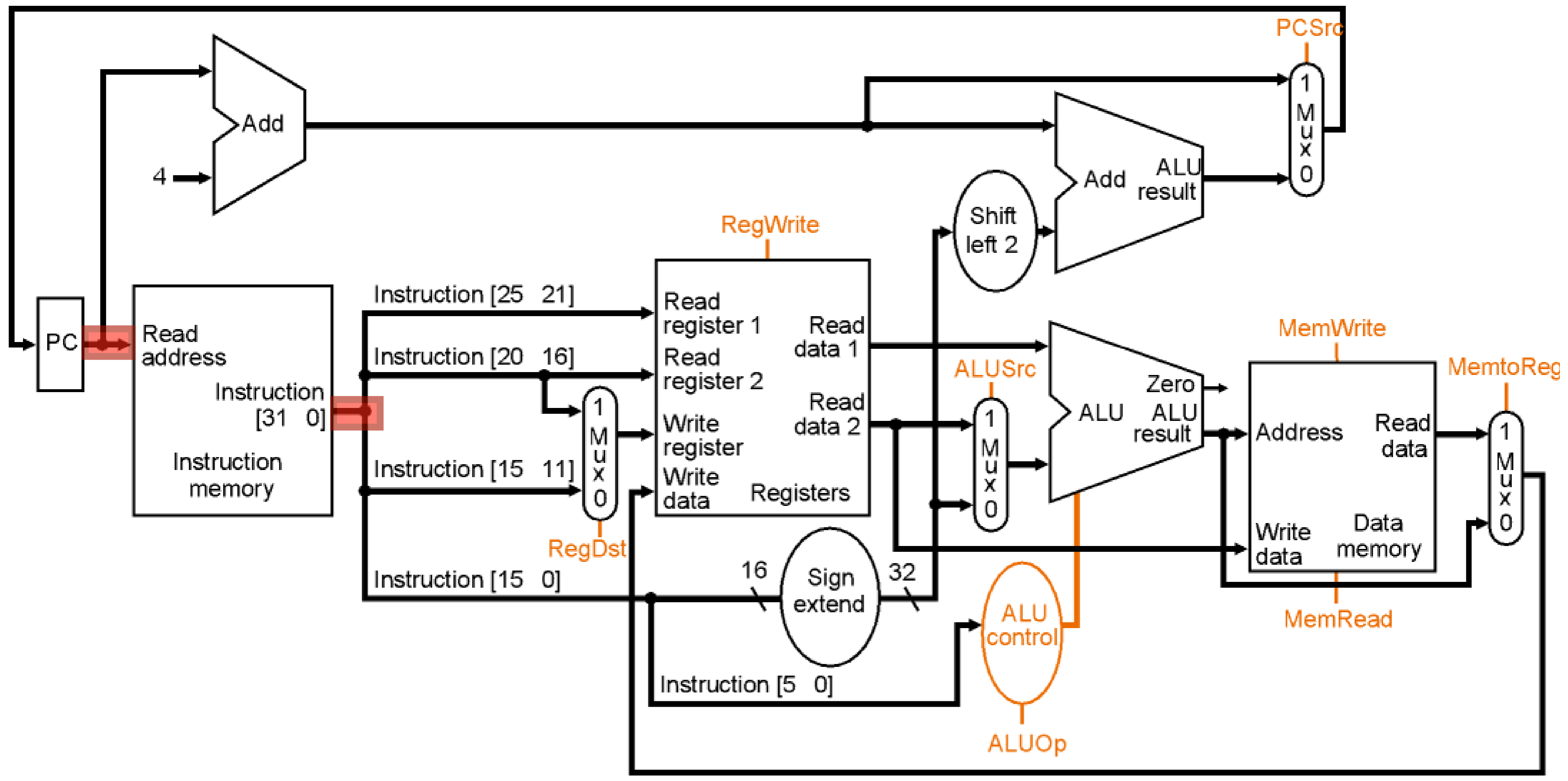
The R-Format (e.g. add) Datapath



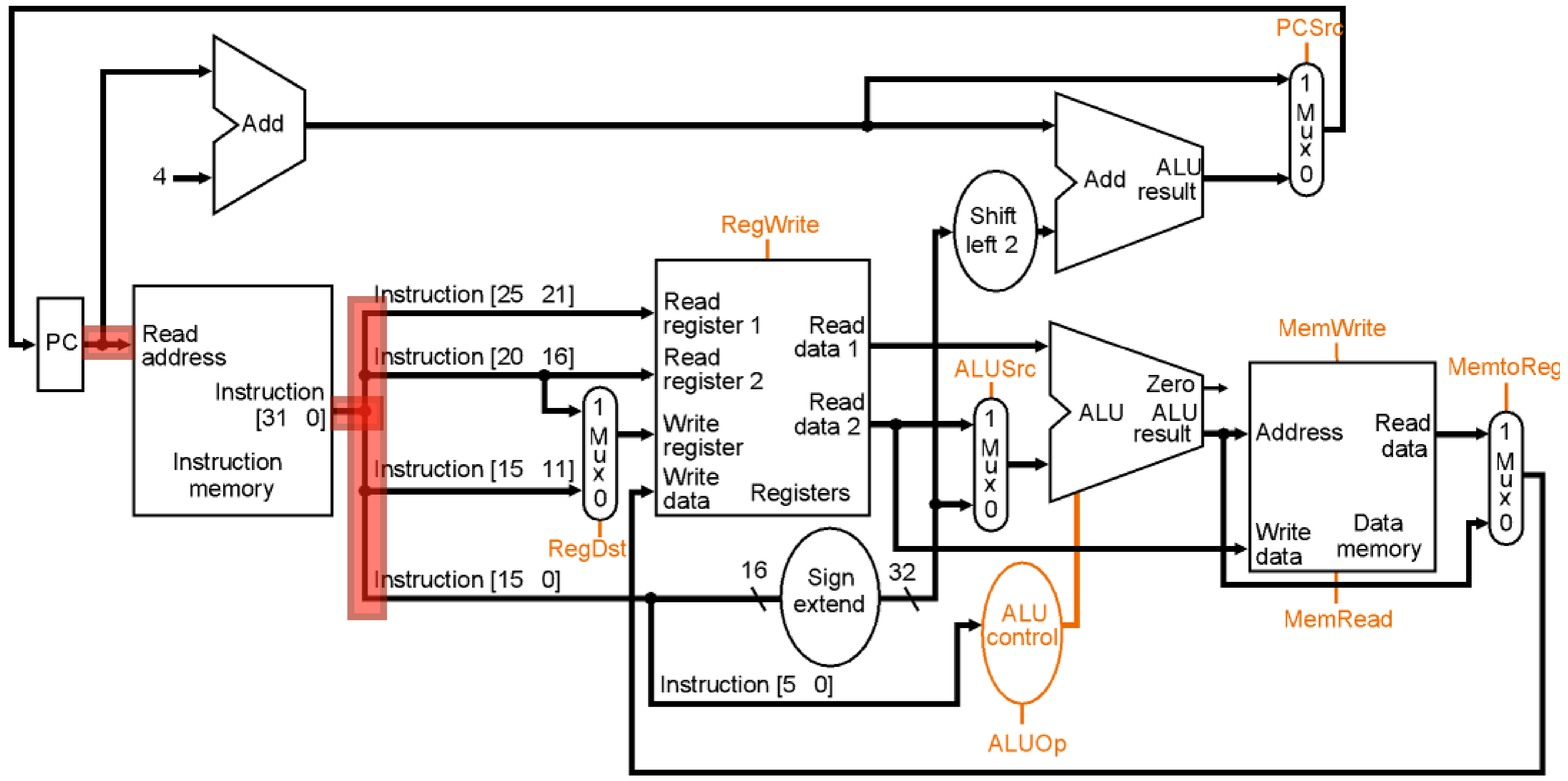
The R-Format (e.g. add) Datapath



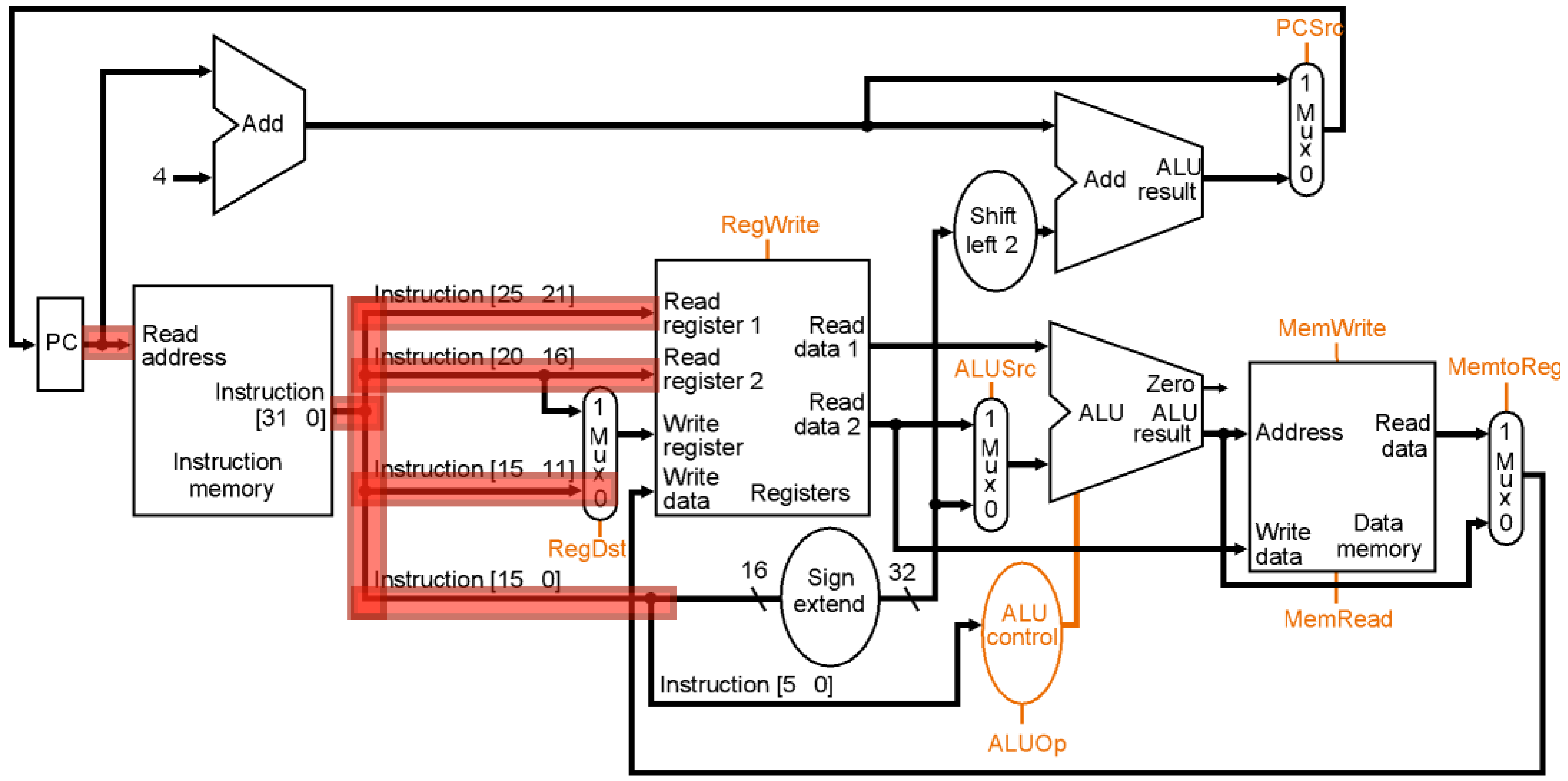
The R-Format (e.g. add) Datapath



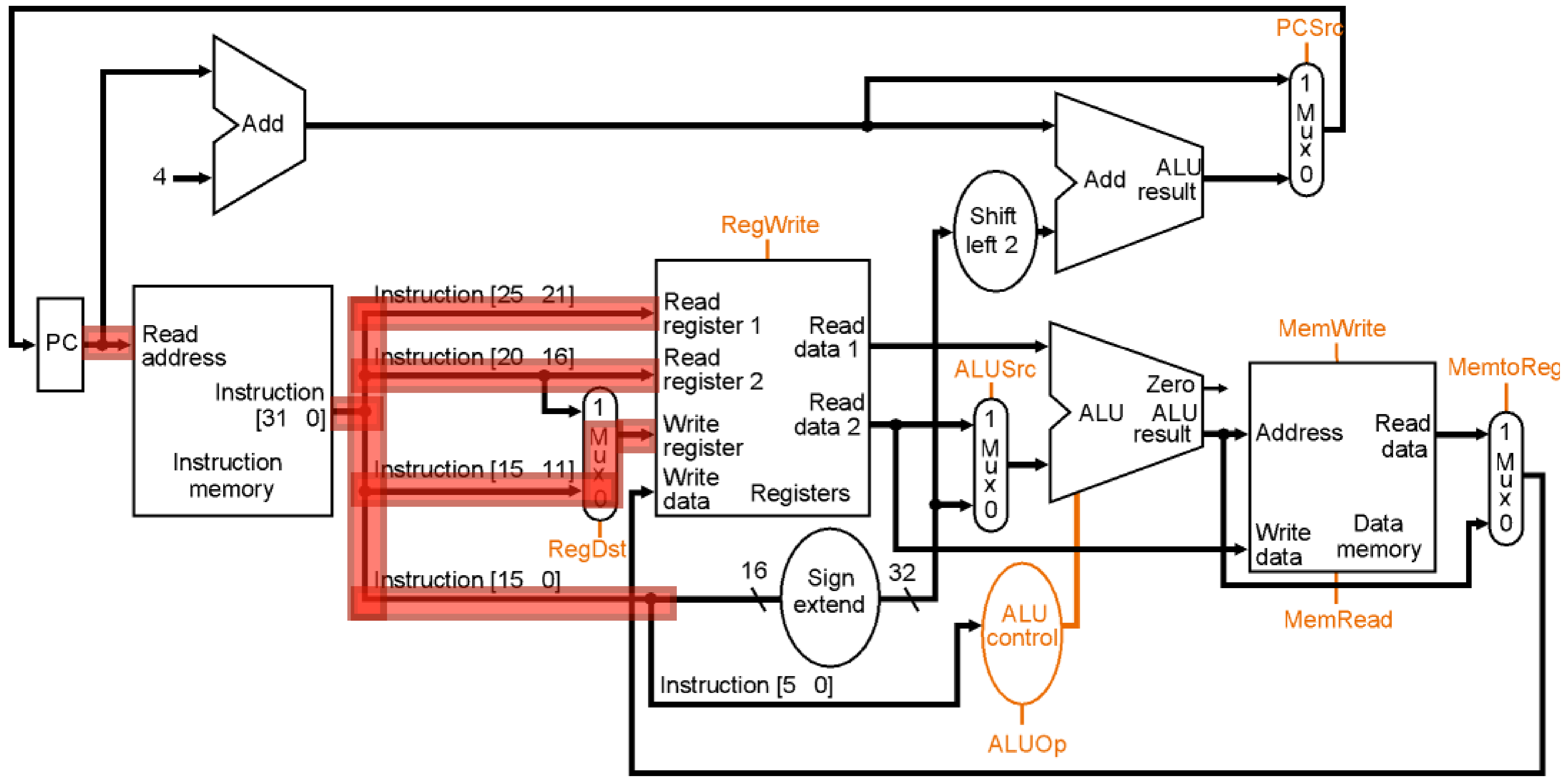
The R-Format (e.g. add) Datapath



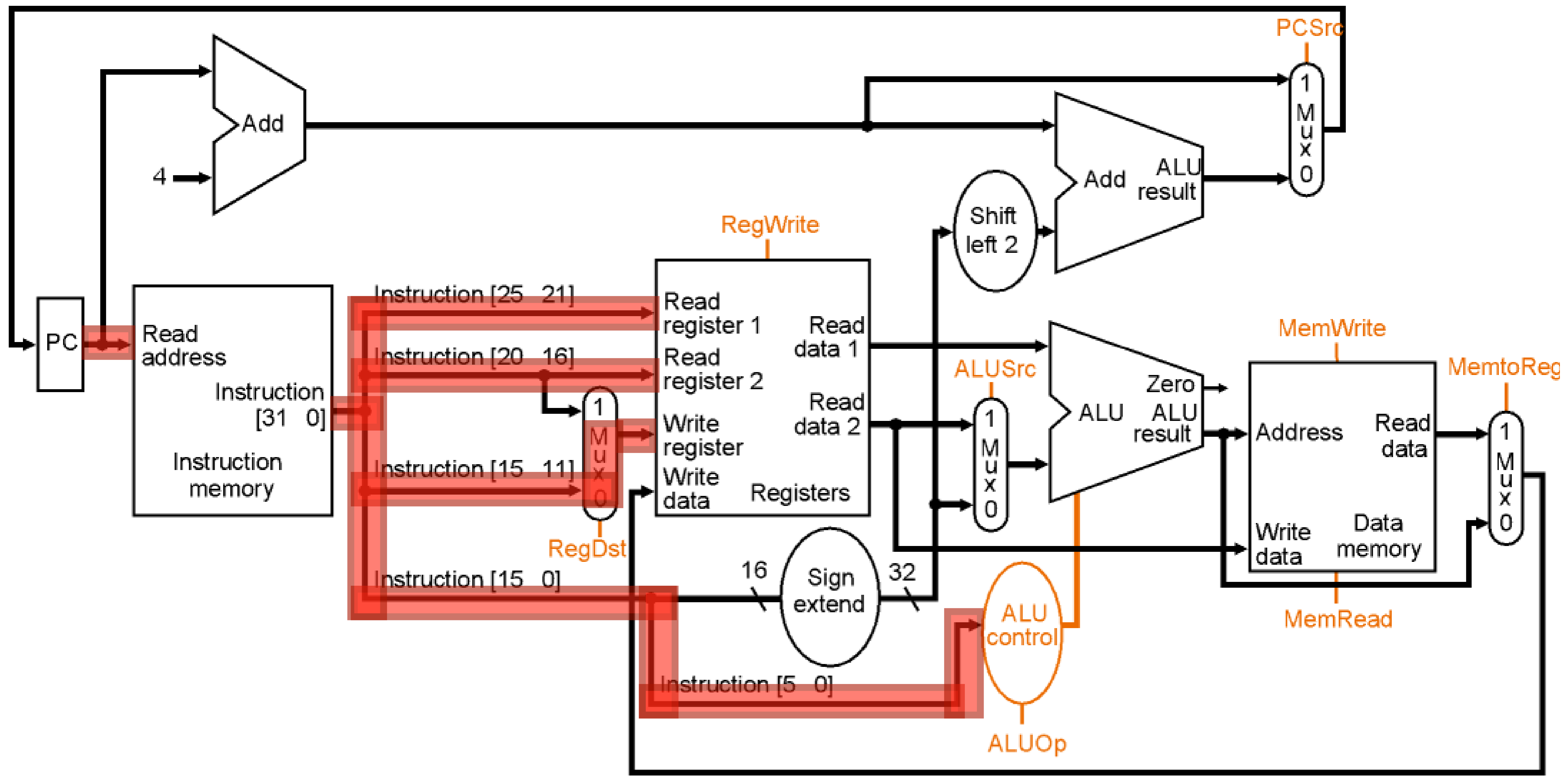
The R-Format (e.g. add) Datapath



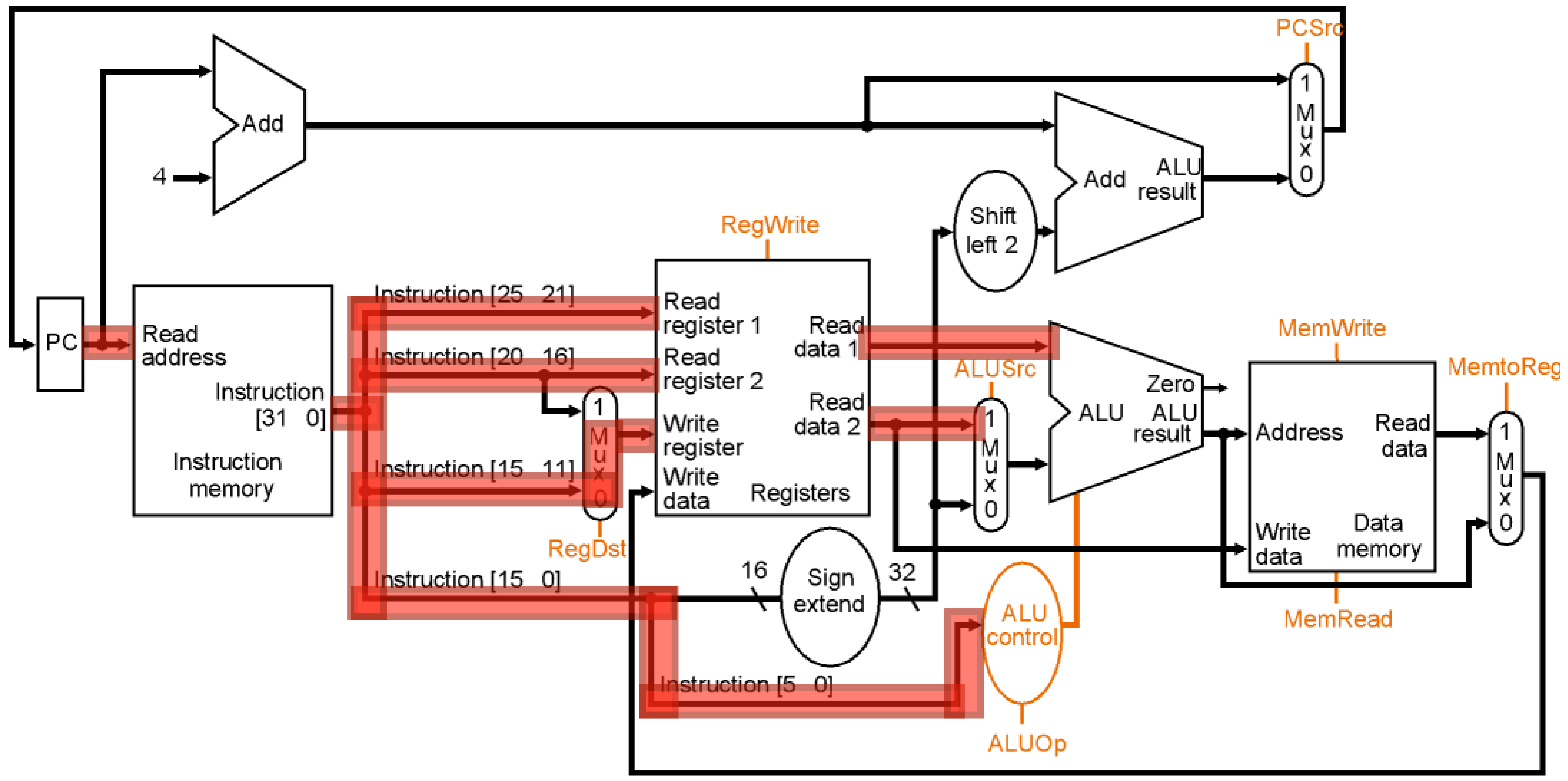
The R-Format (e.g. add) Datapath



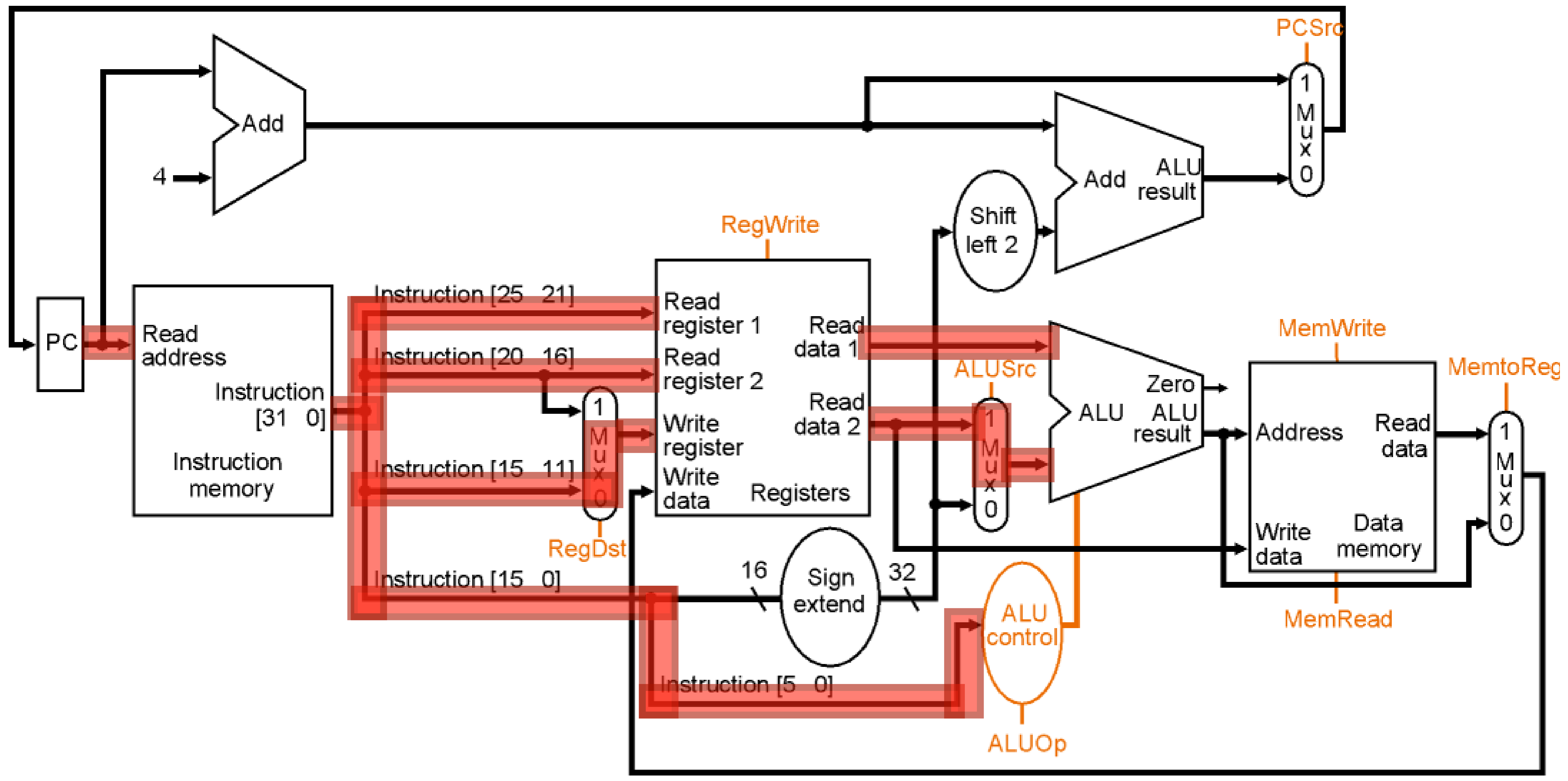
The R-Format (e.g. add) Datapath



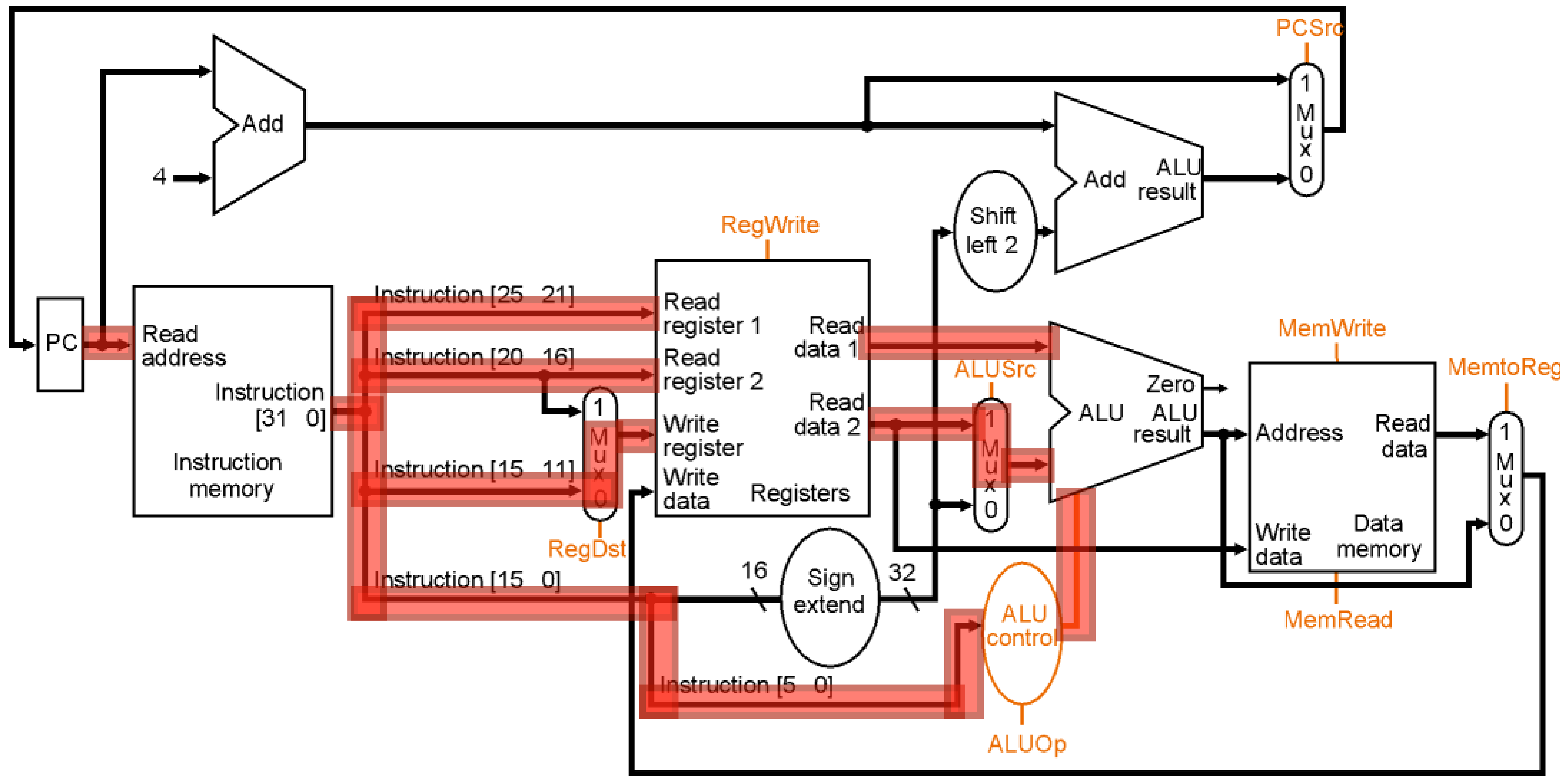
The R-Format (e.g. add) Datapath



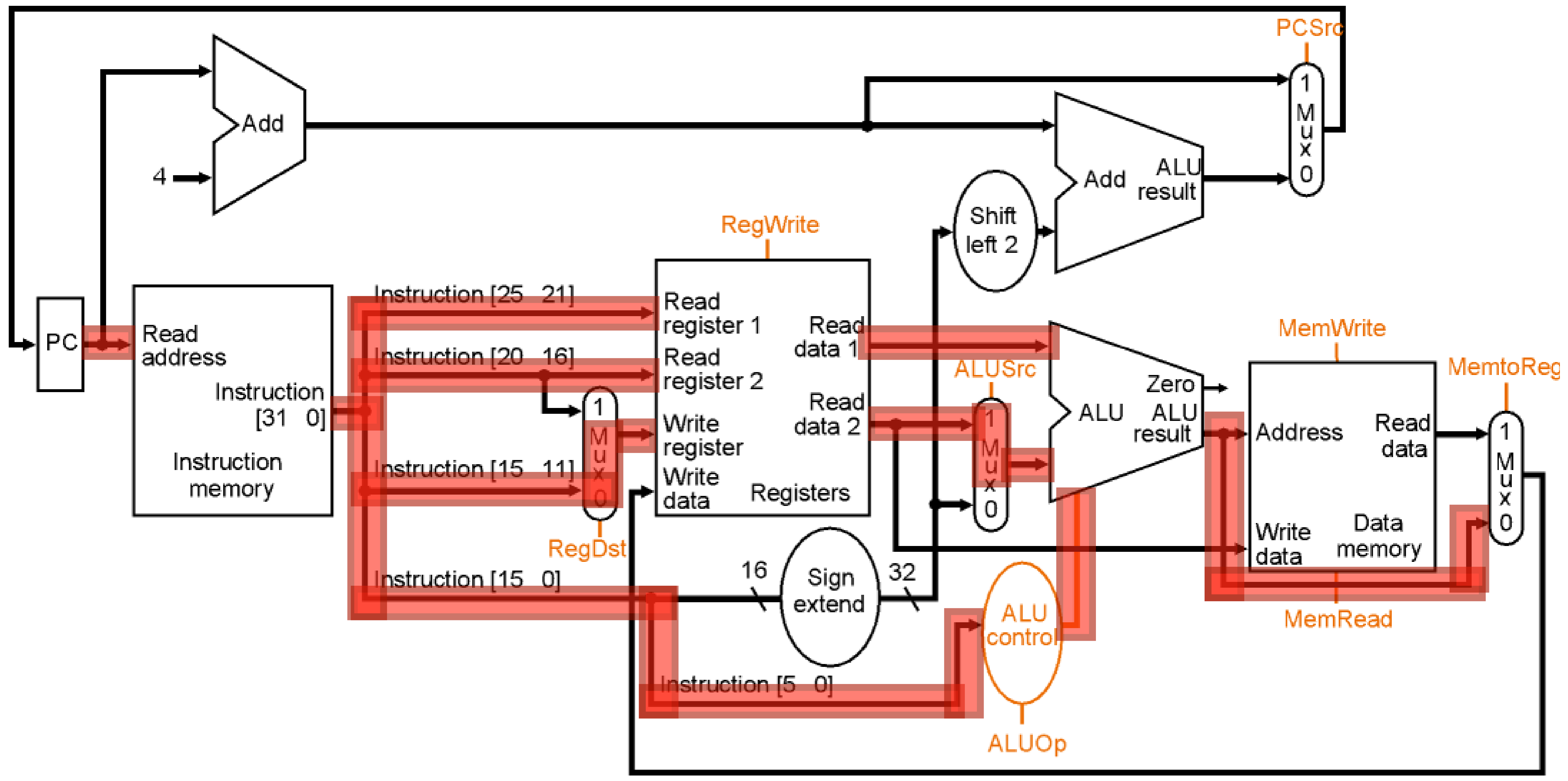
The R-Format (e.g. add) Datapath



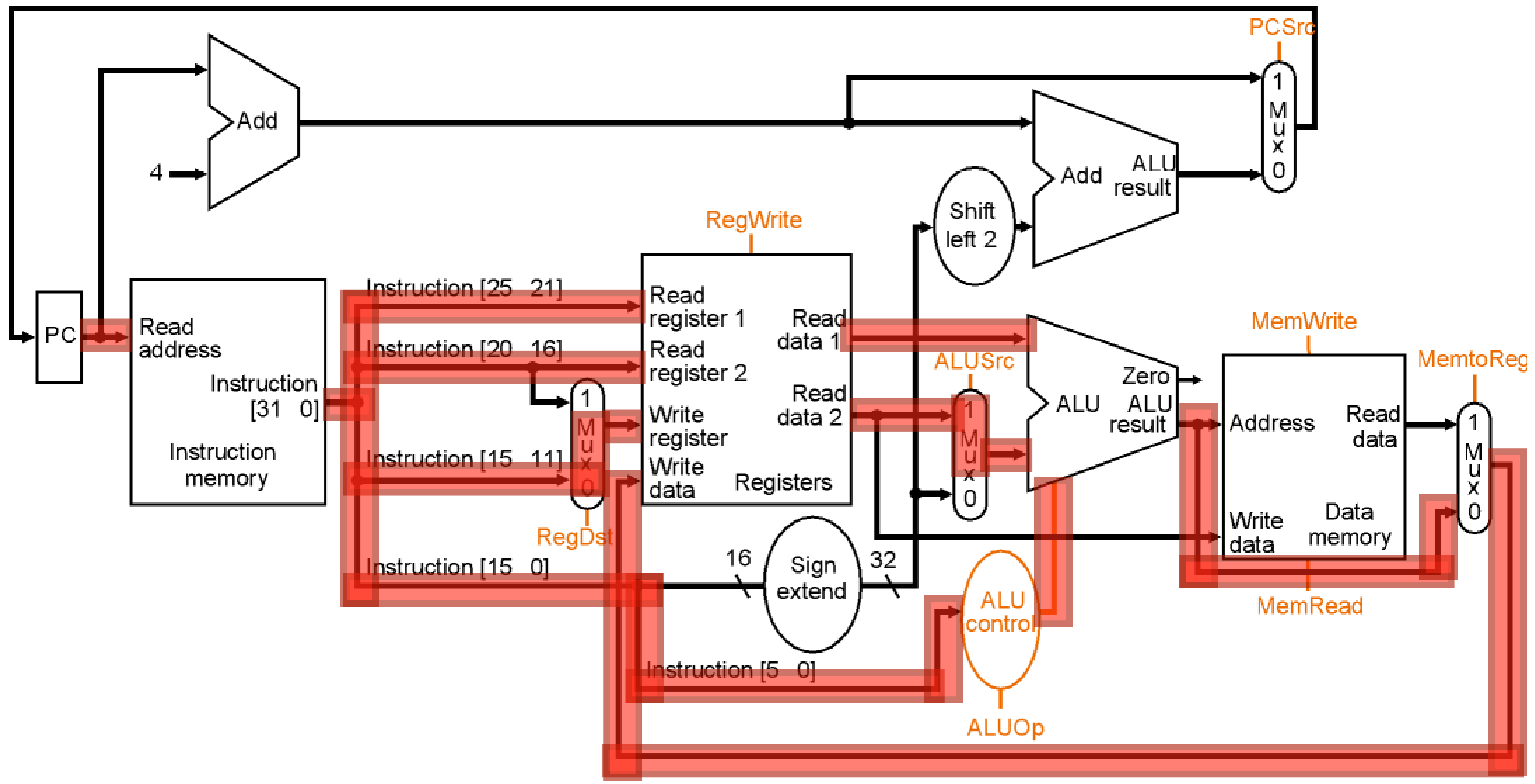
The R-Format (e.g. add) Datapath



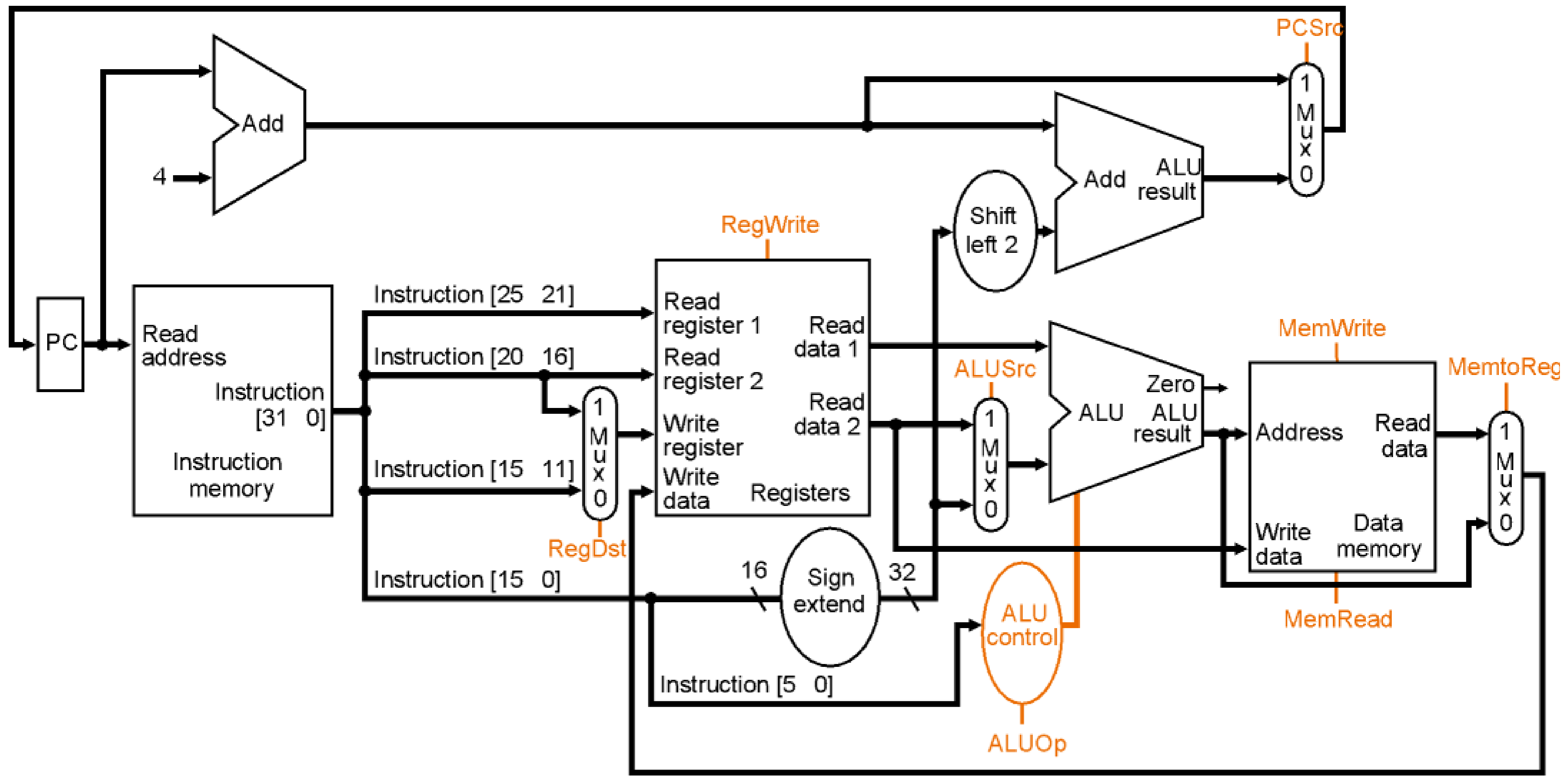
The R-Format (e.g. add) Datapath



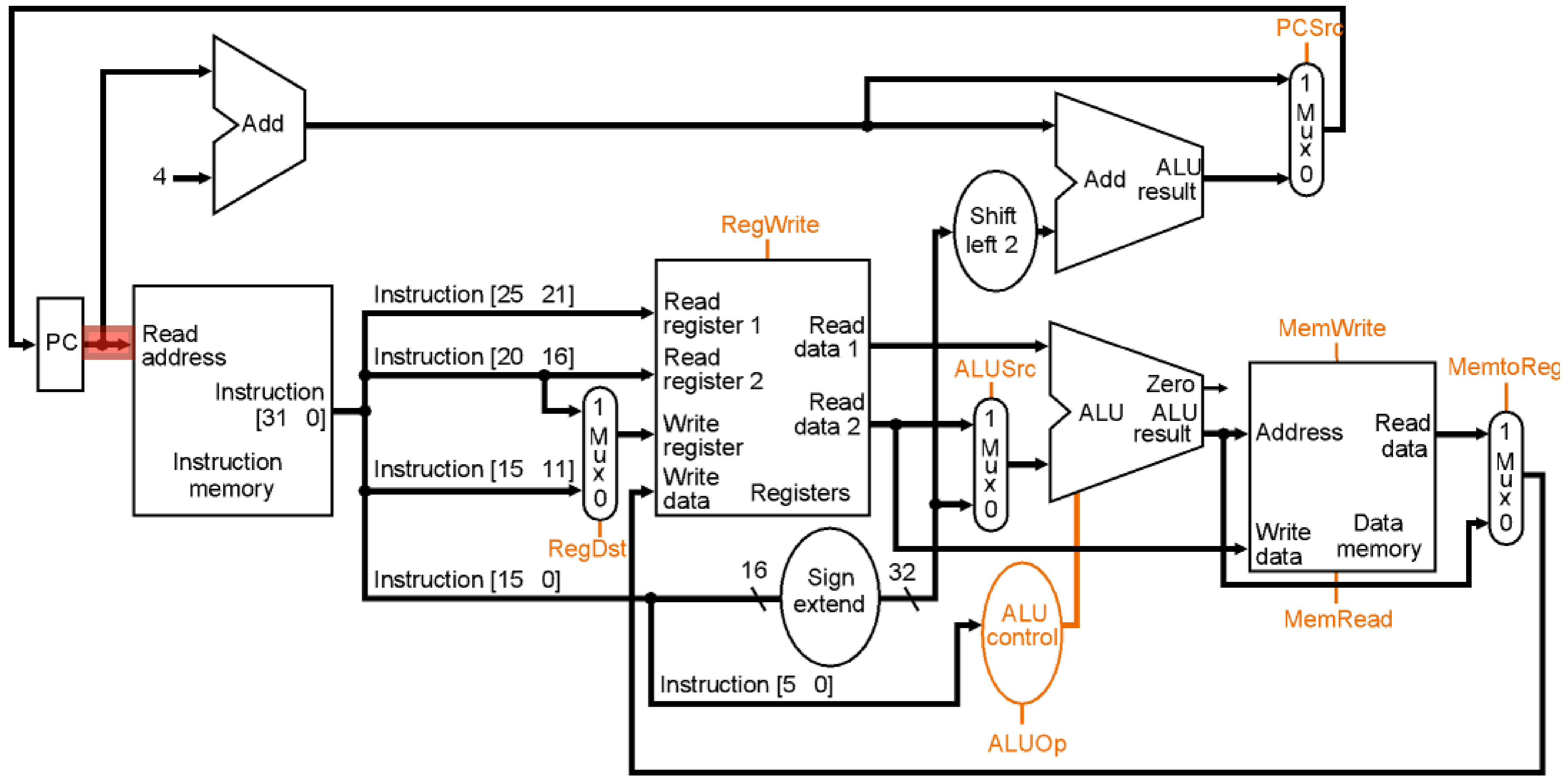
The R-Format (e.g. add) Datapath



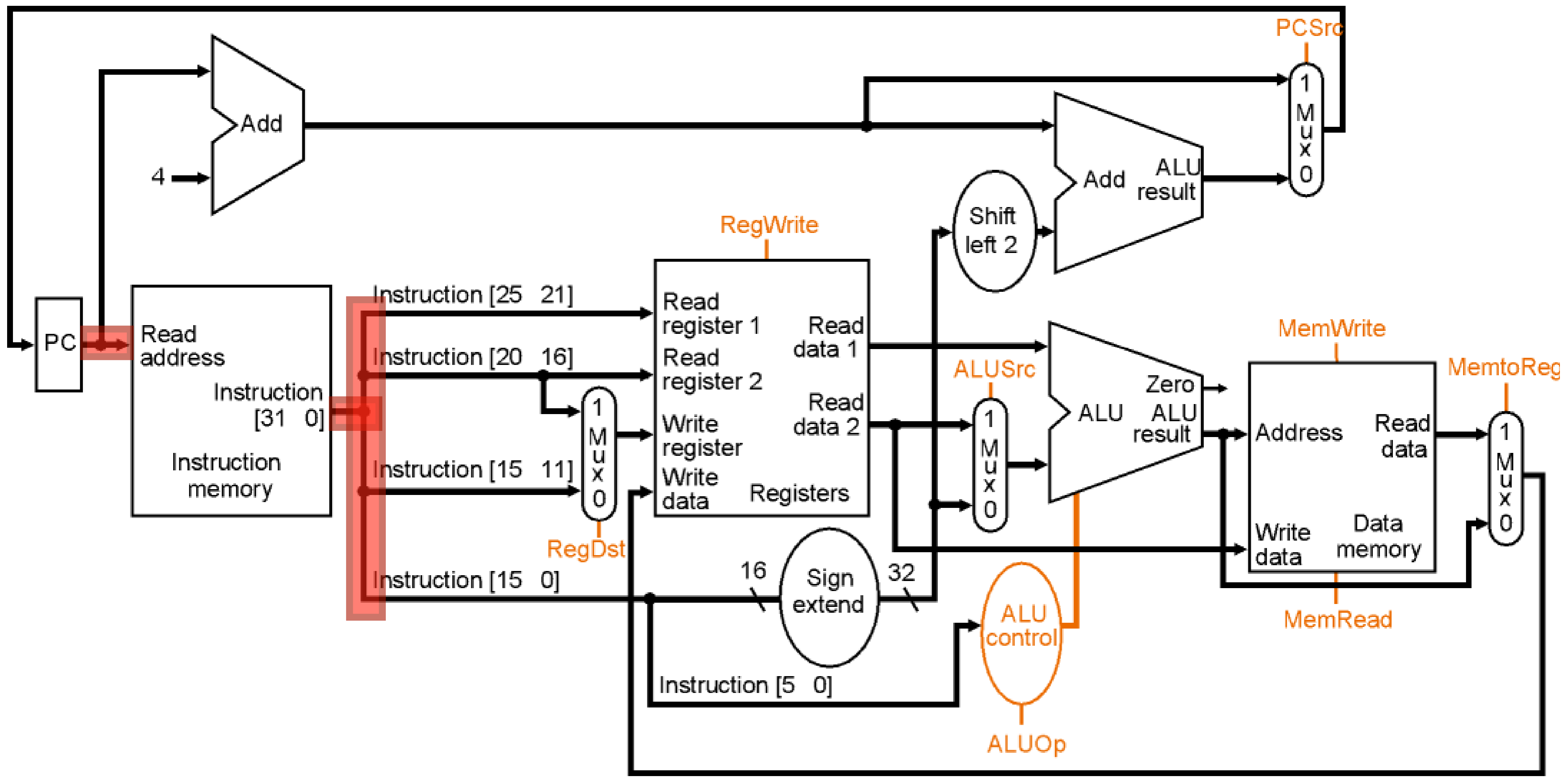
The Load Datapath



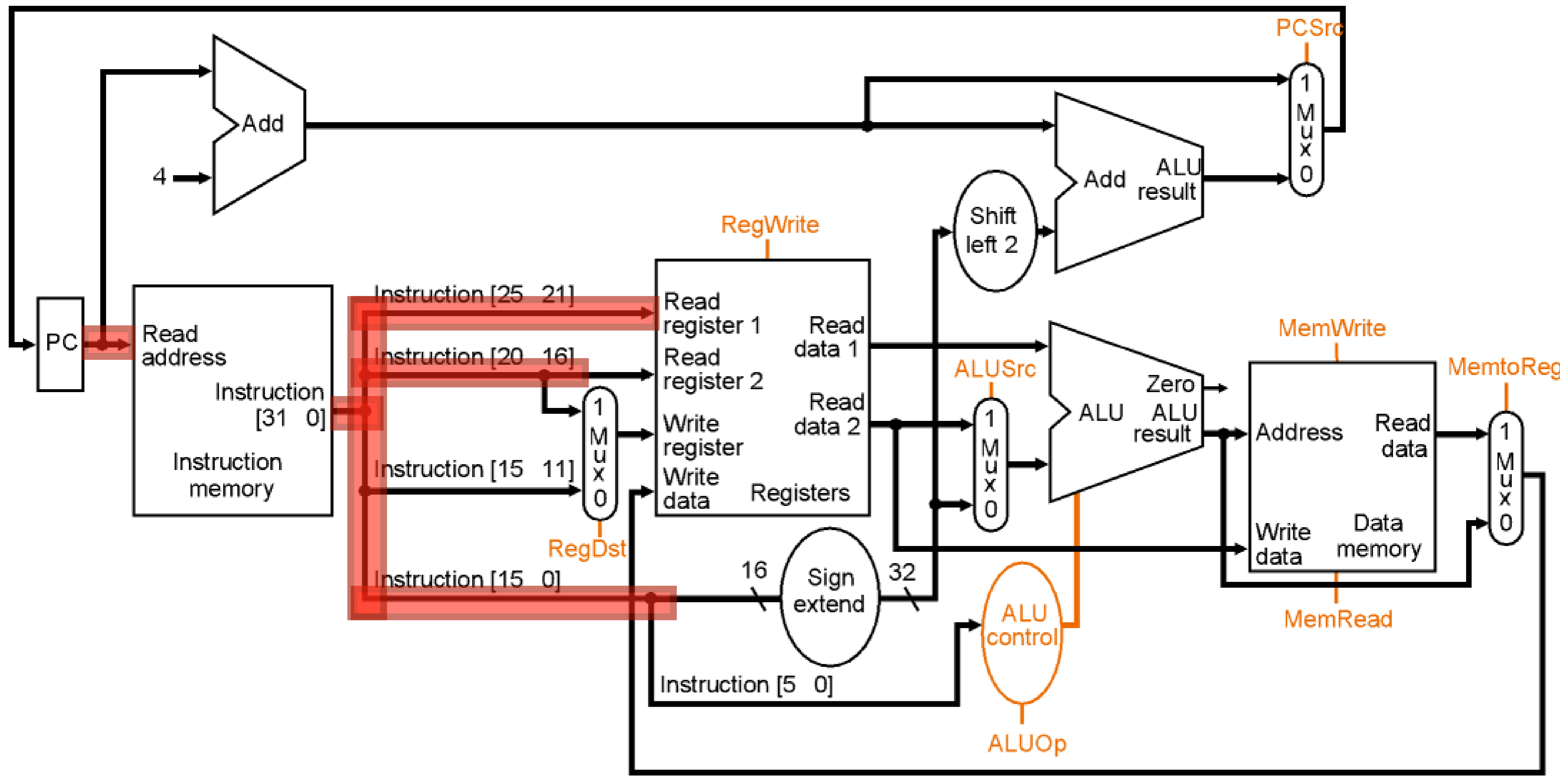
The Load Datapath



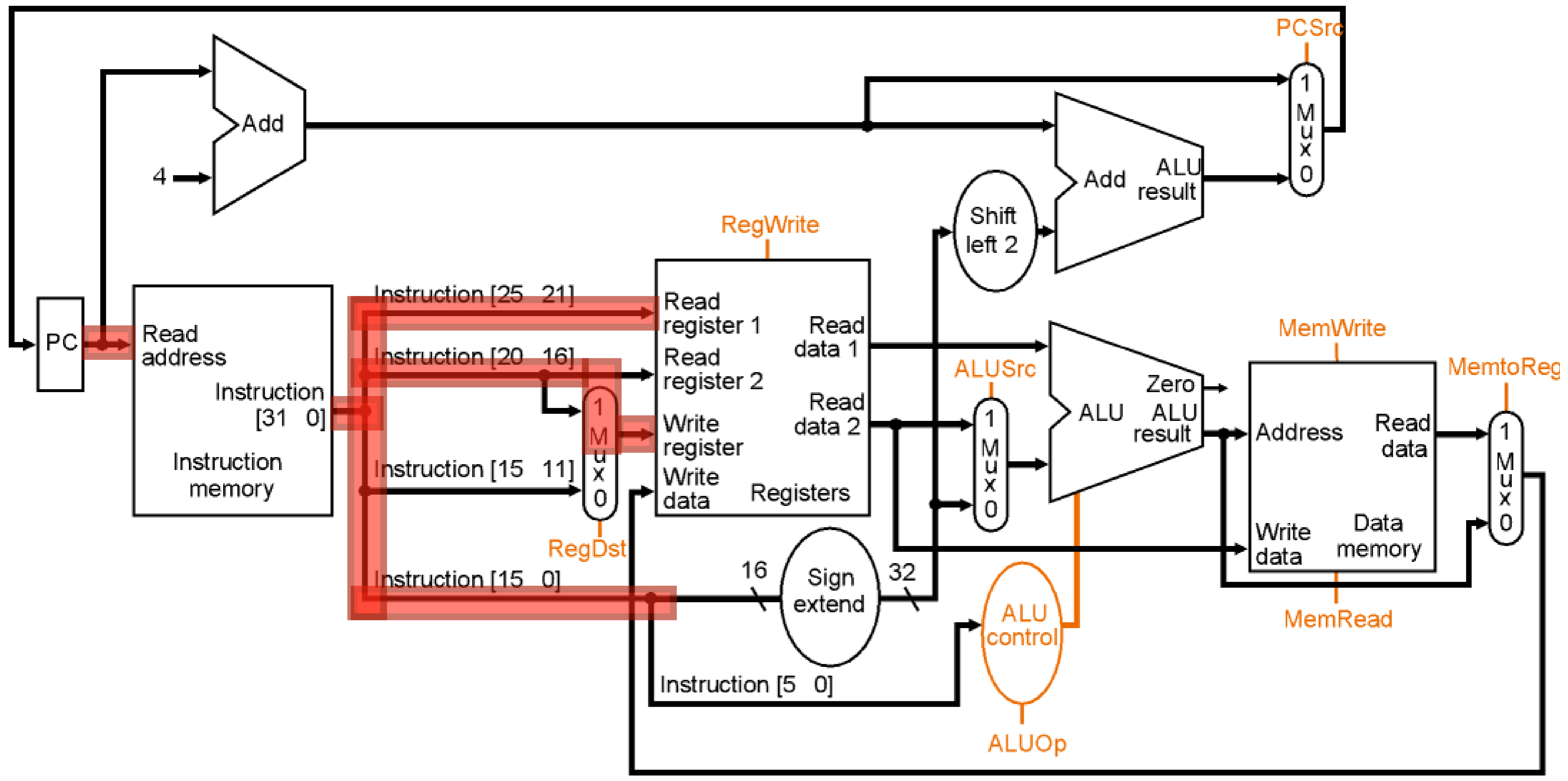
The Load Datapath



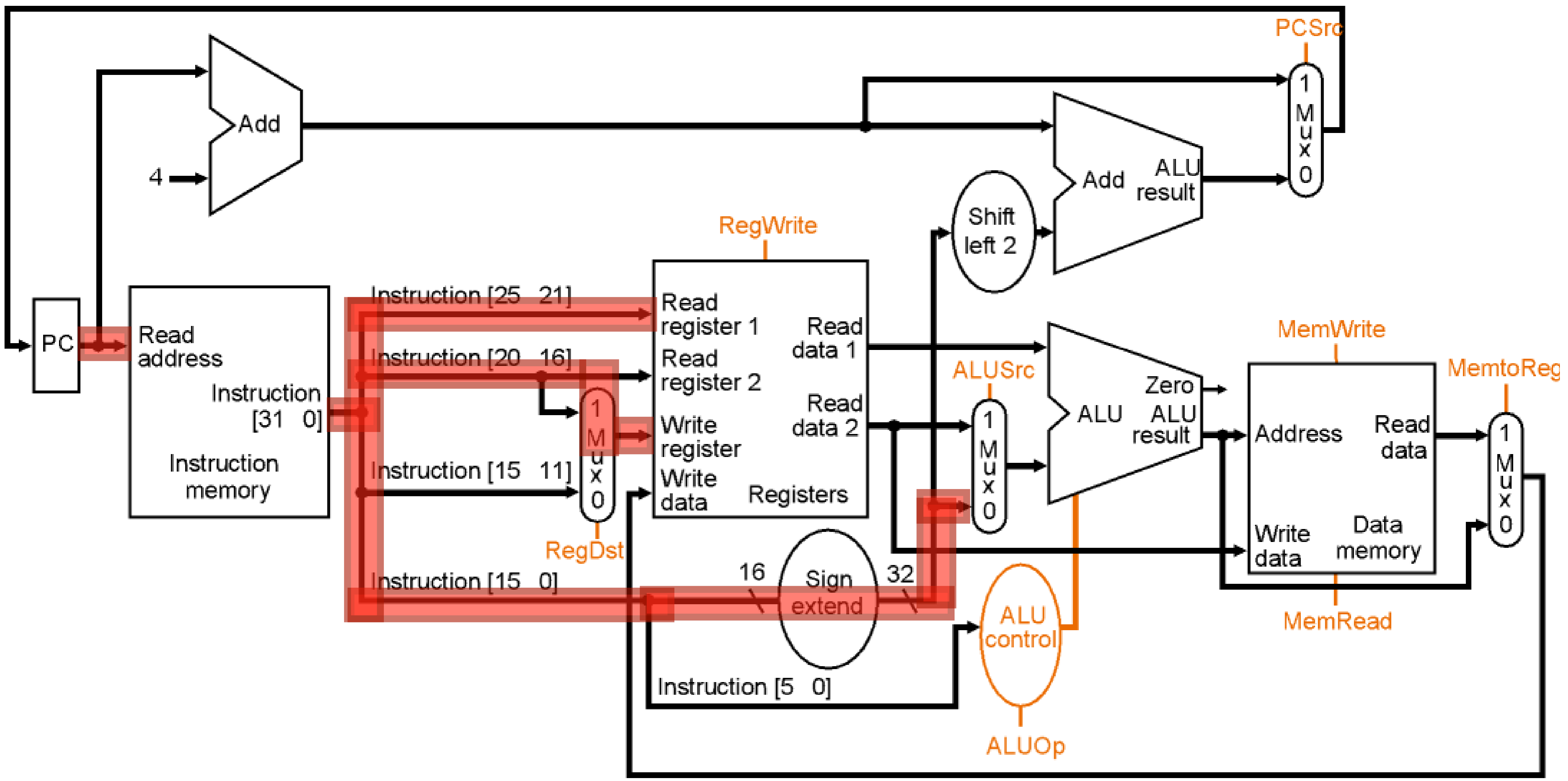
The Load Datapath



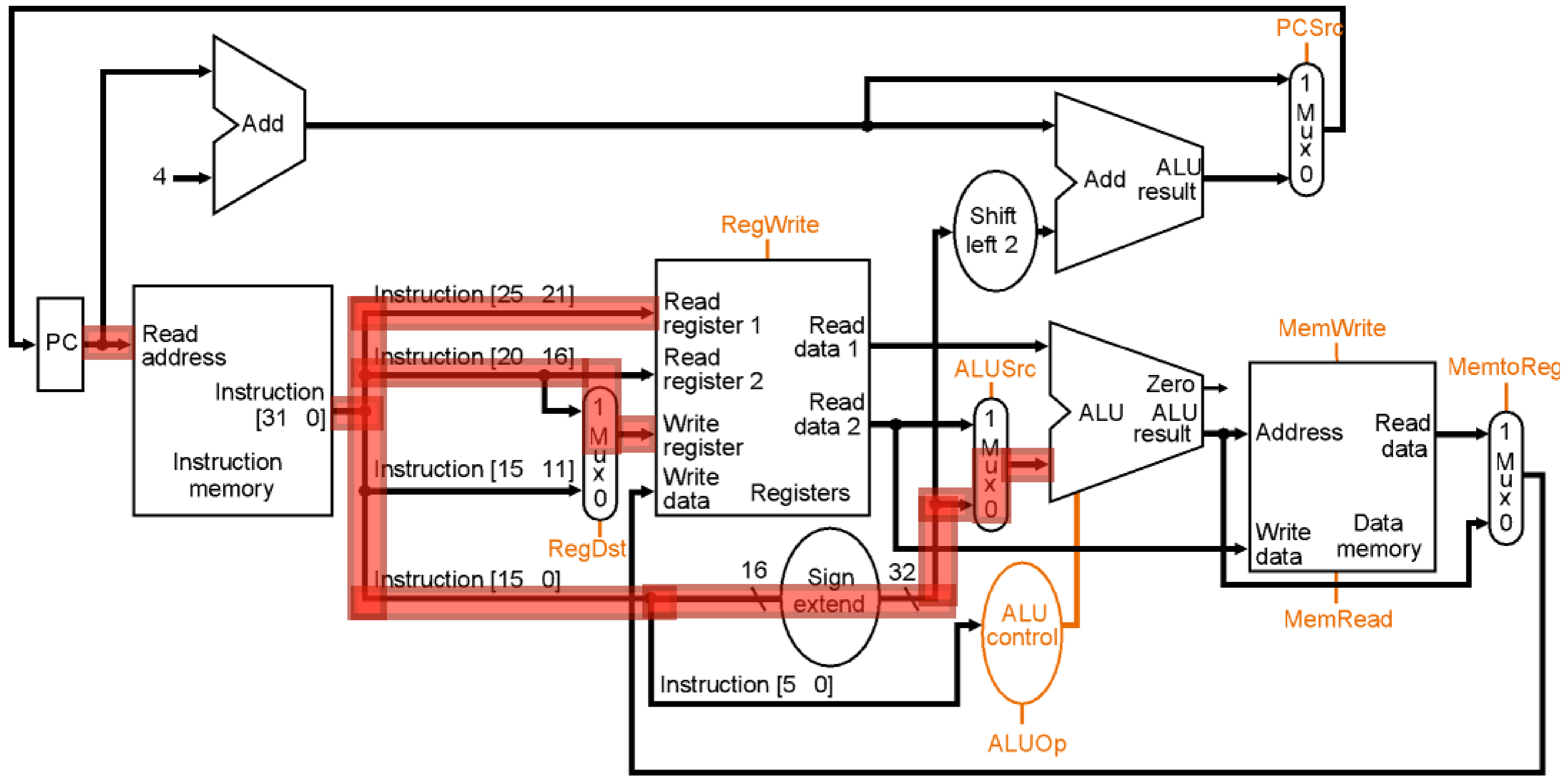
The Load Datapath



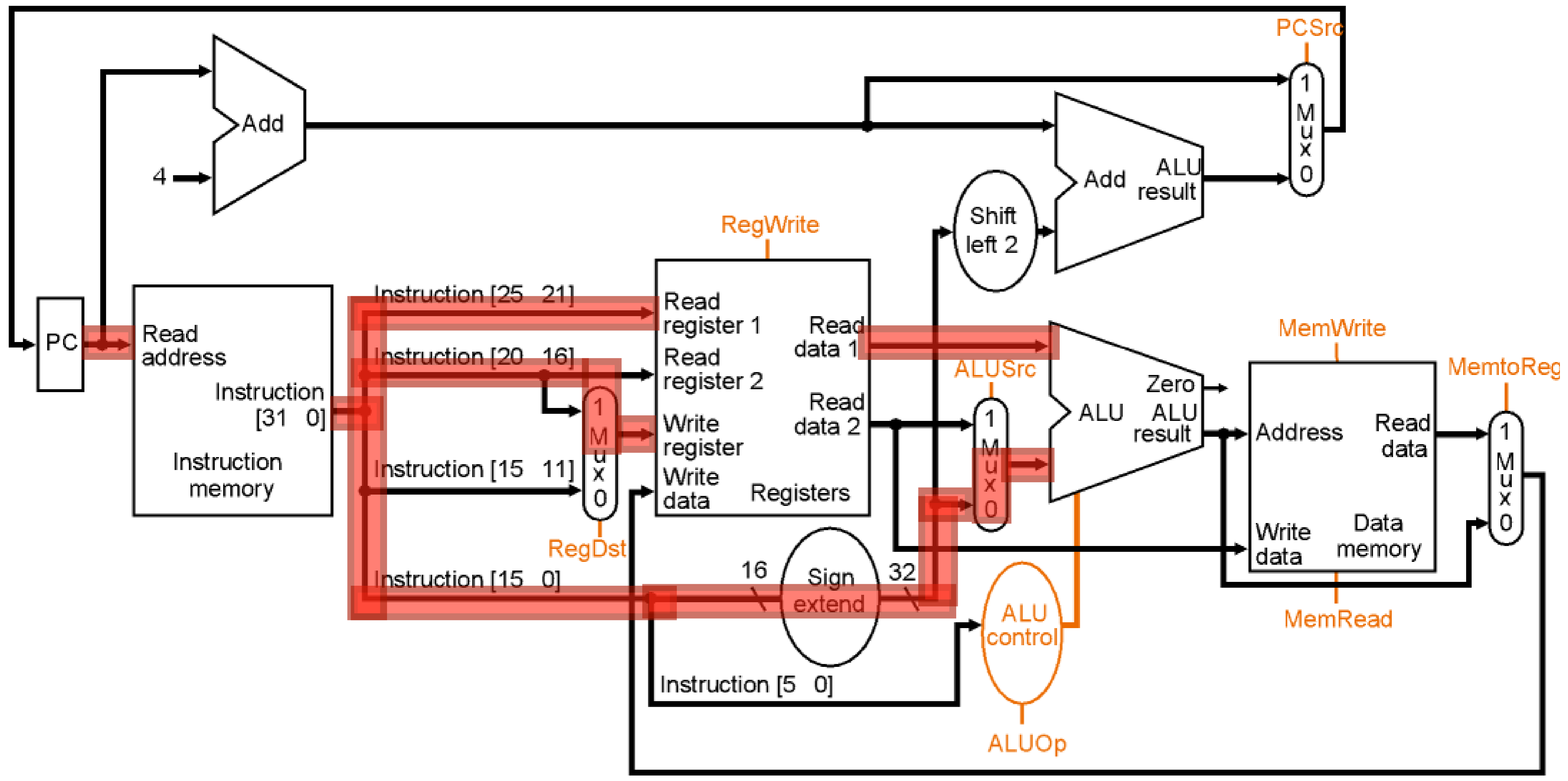
The Load Datapath



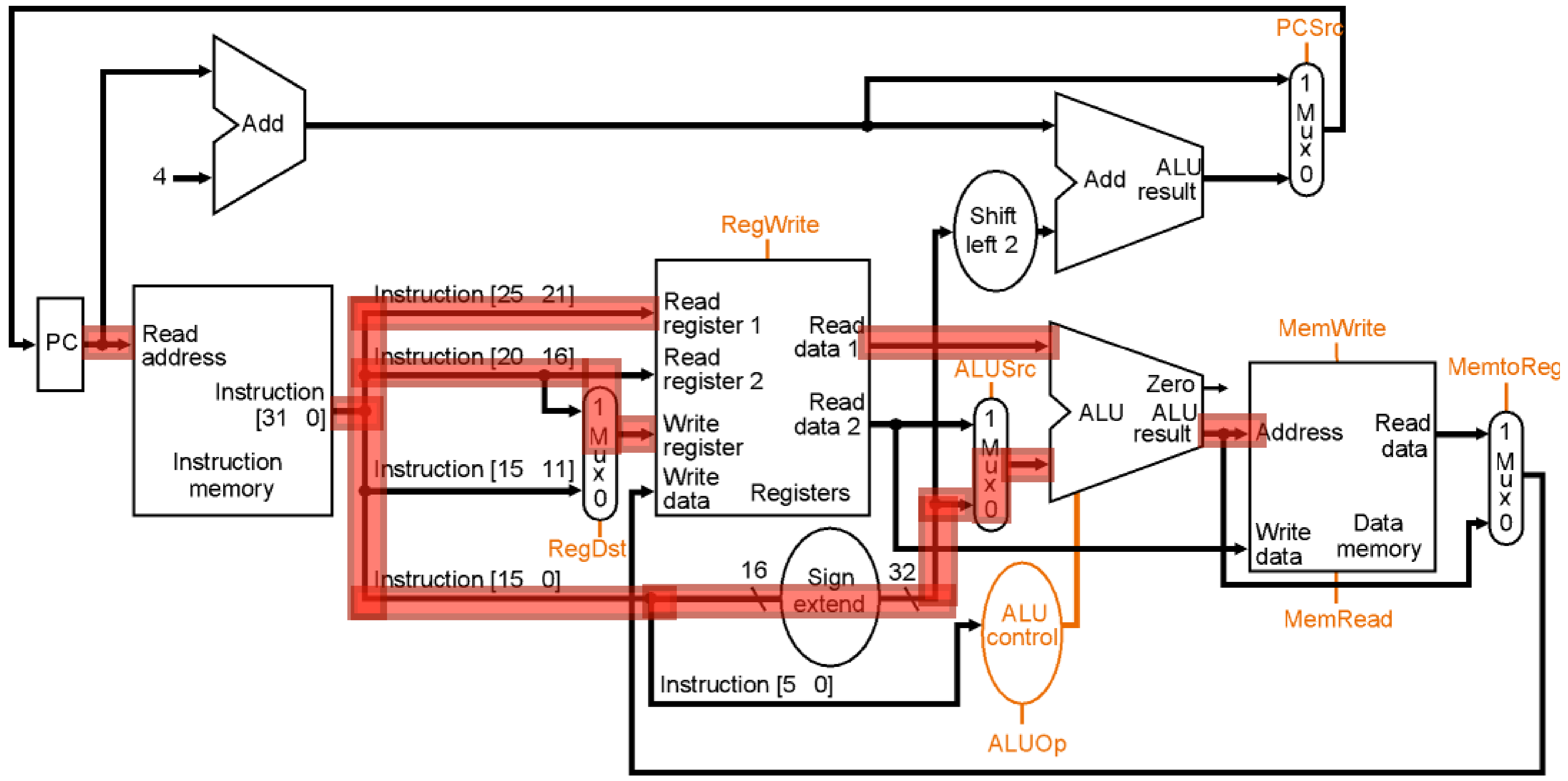
The Load Datapath



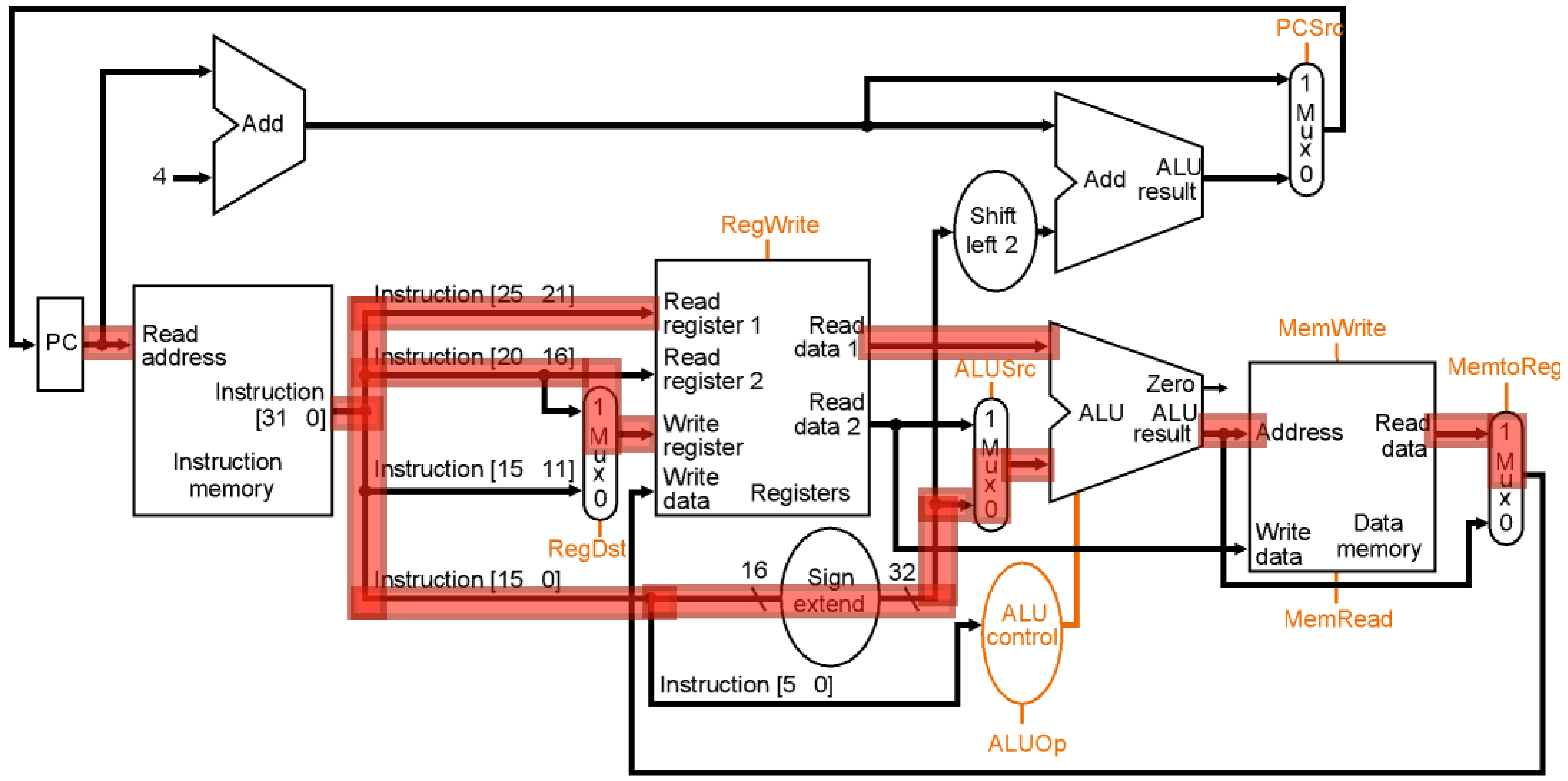
The Load Datapath



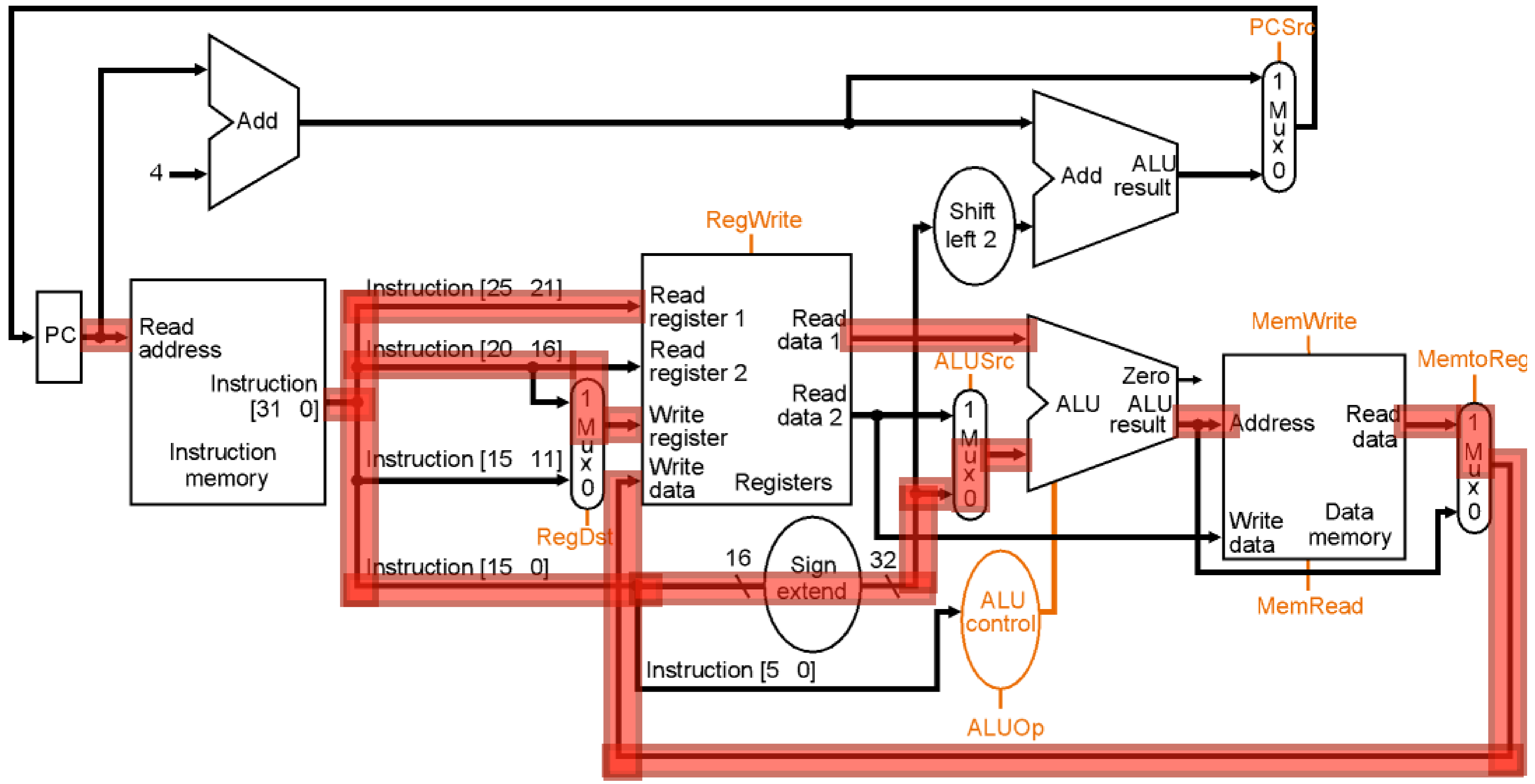
The Load Datapath



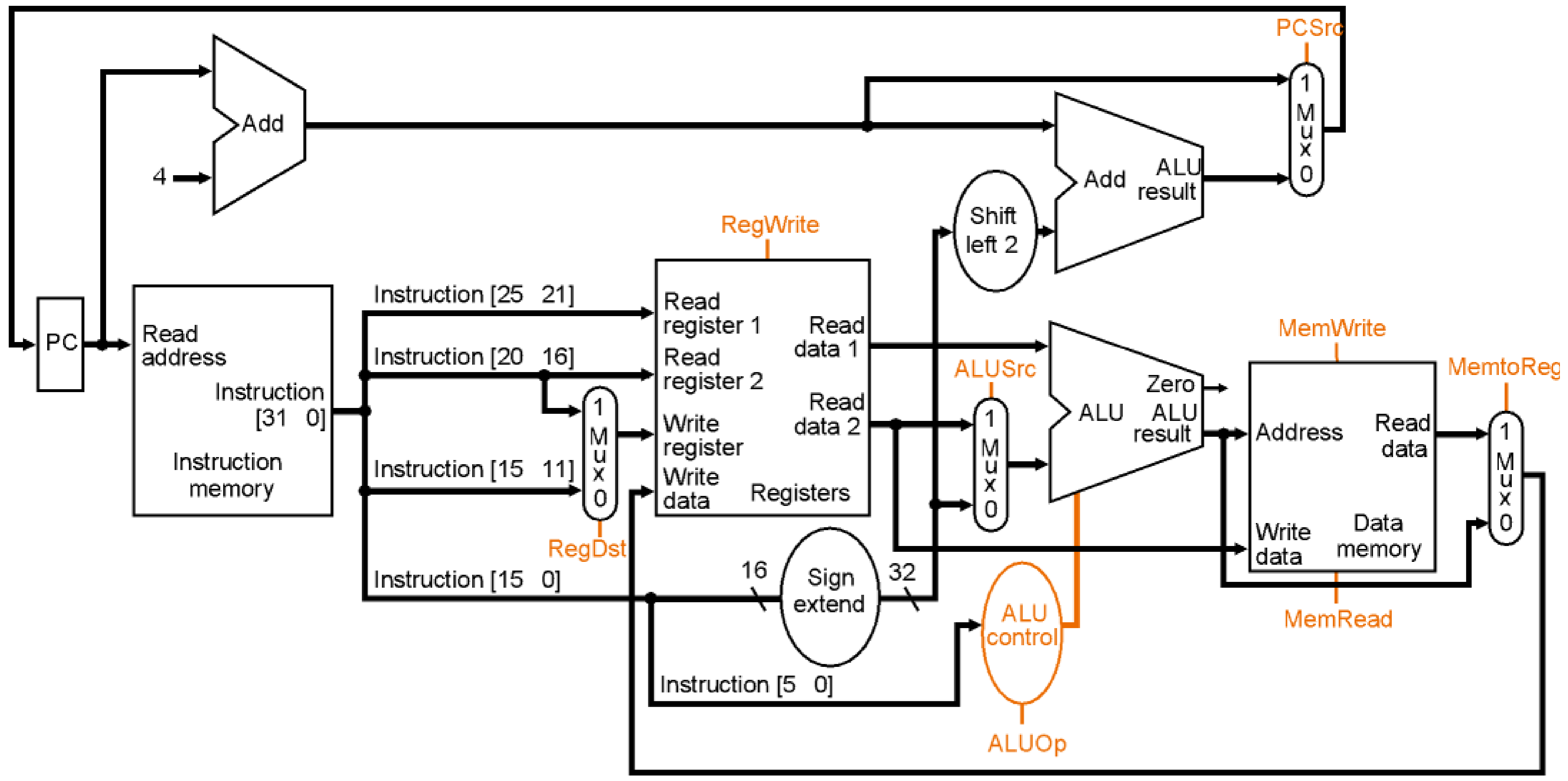
The Load Datapath



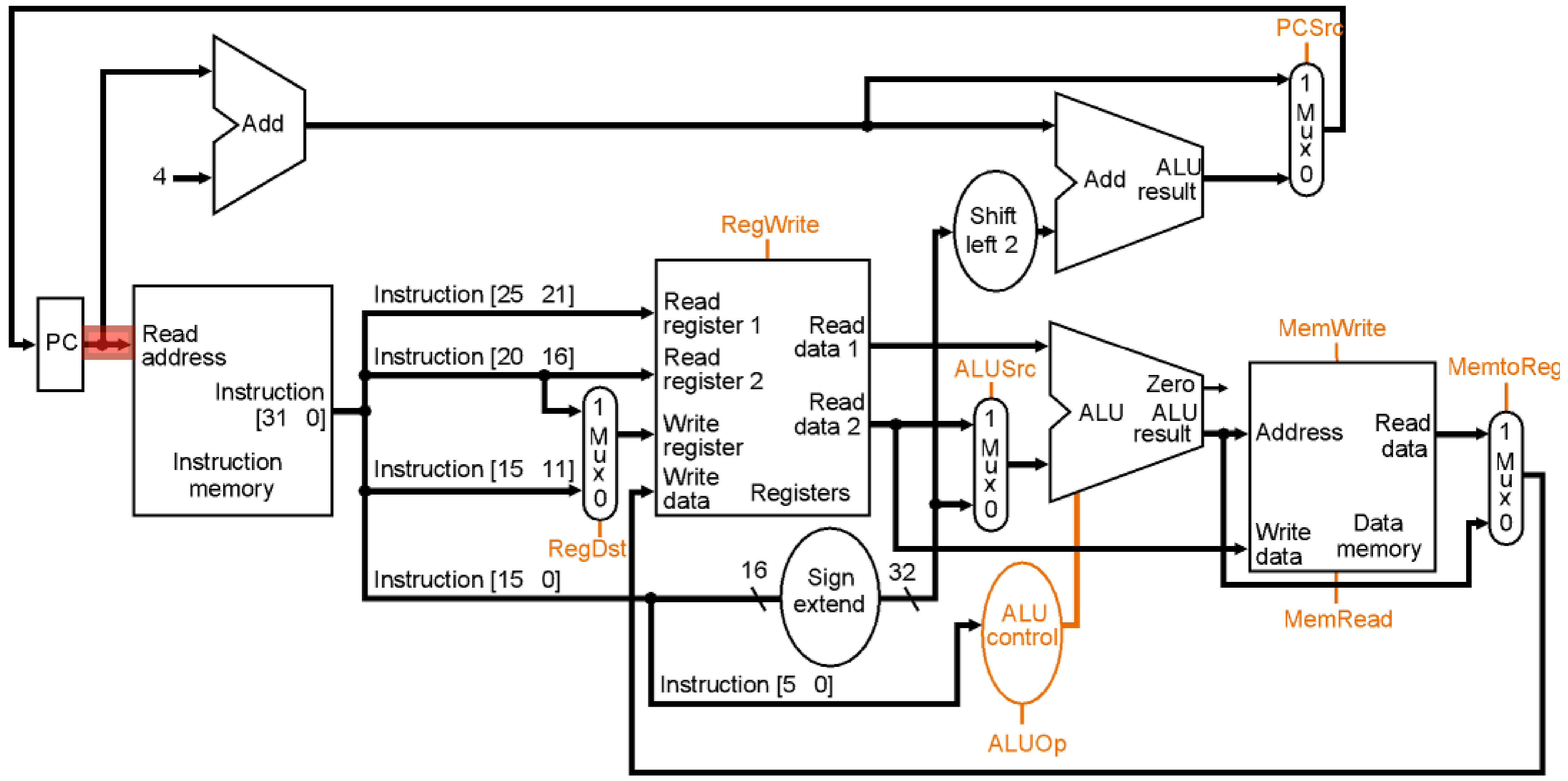
The Load Datapath



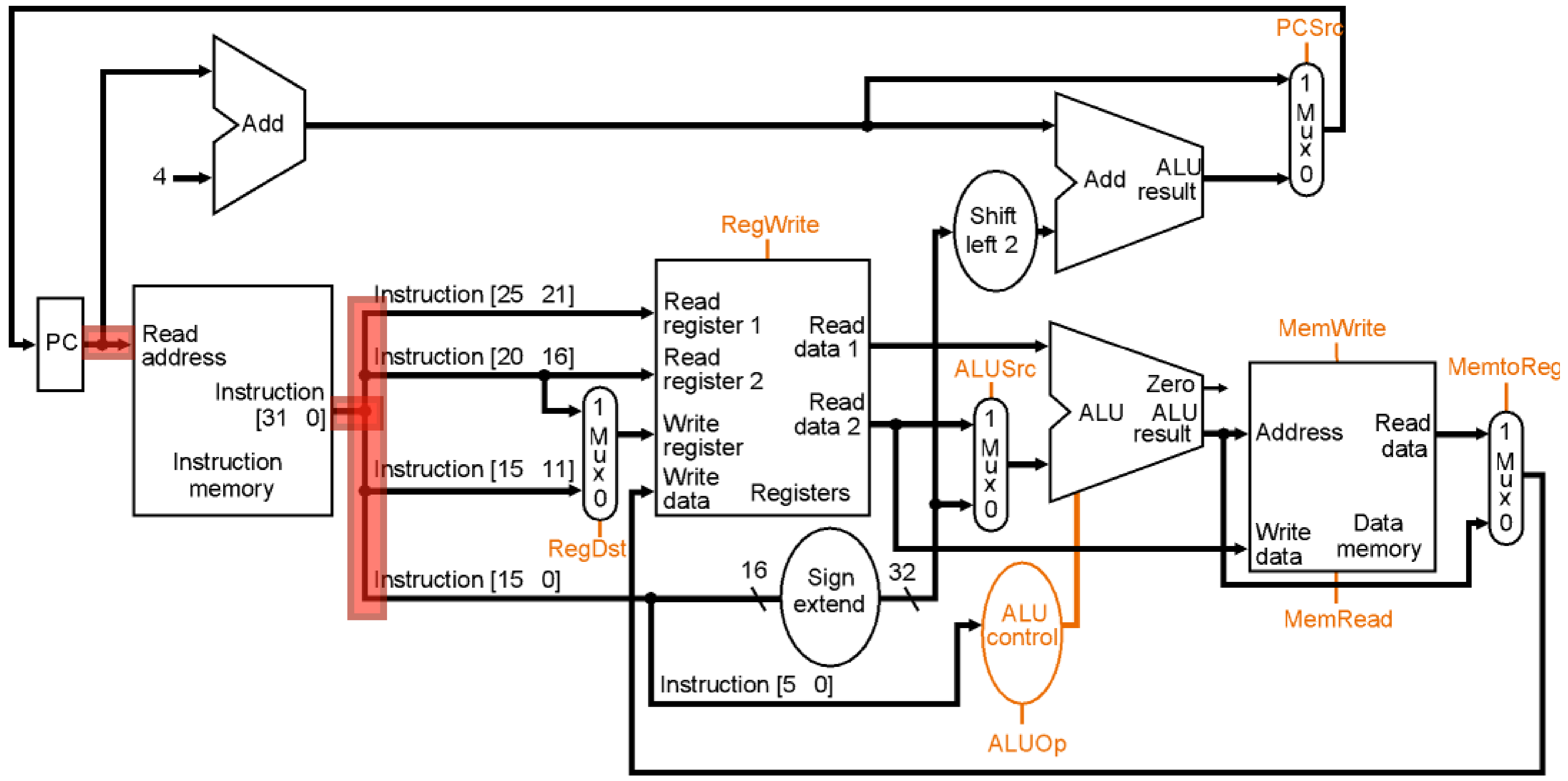
The Store Datapath



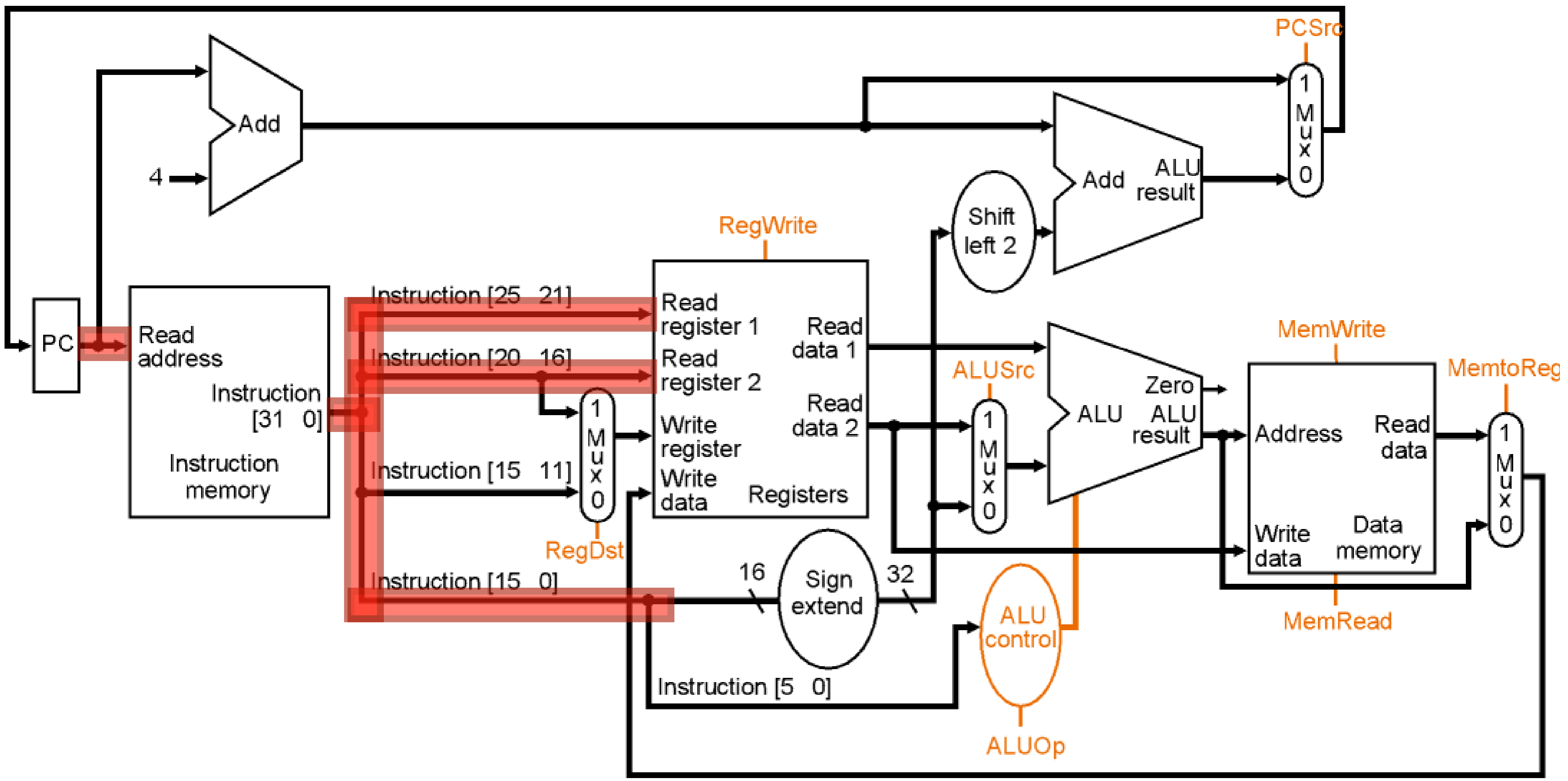
The Store Datapath



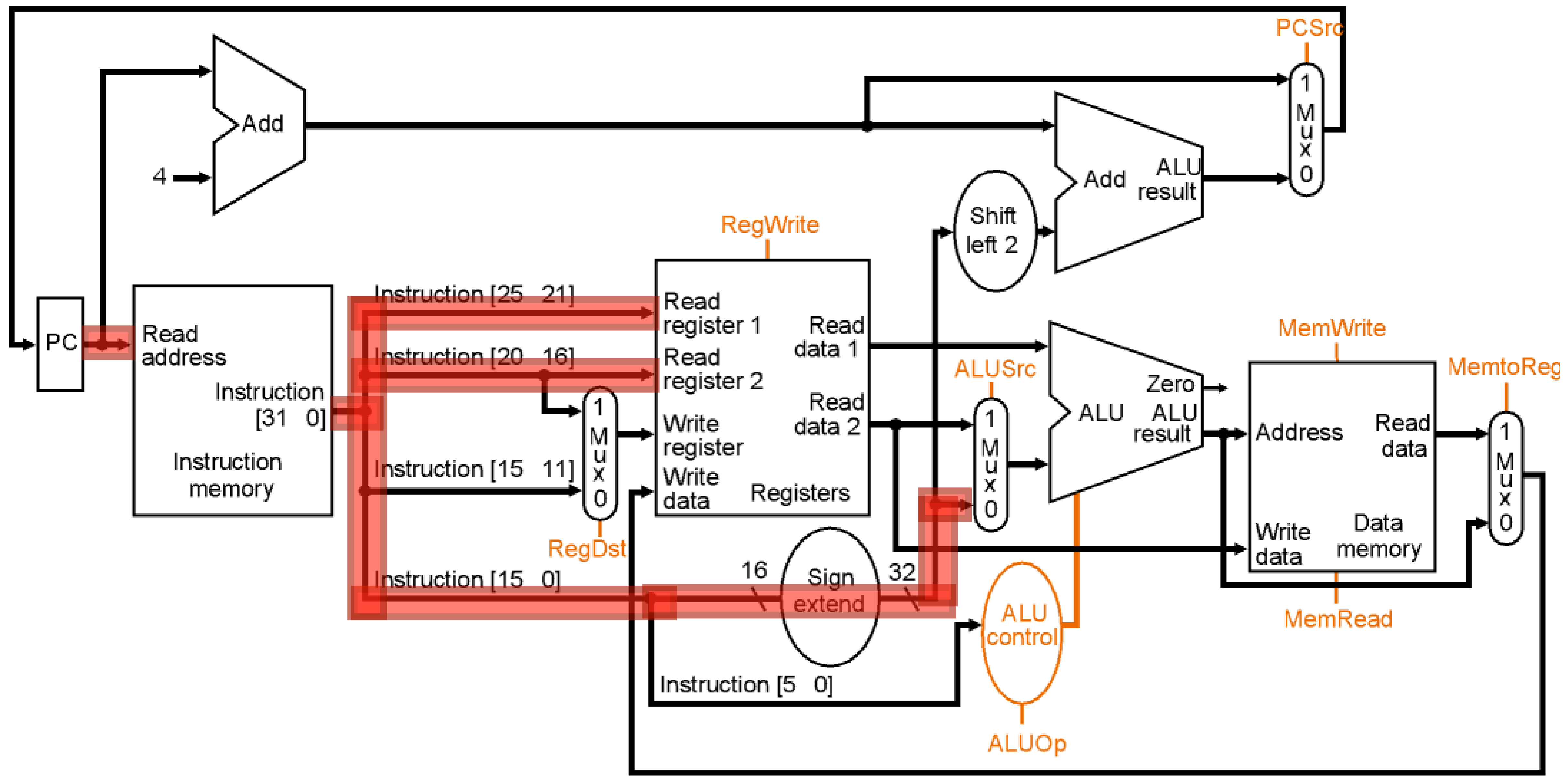
The Store Datapath



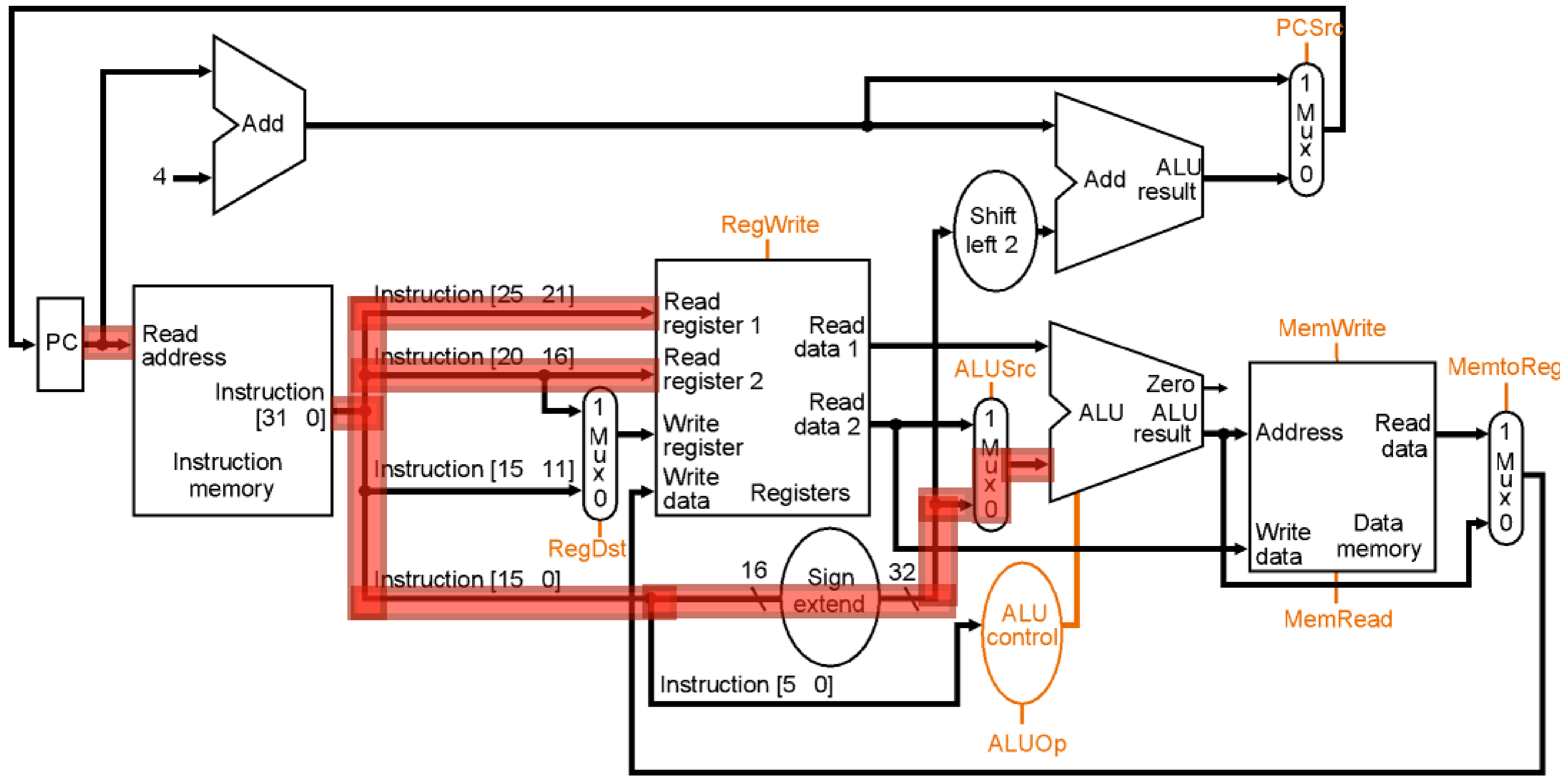
The Store Datapath



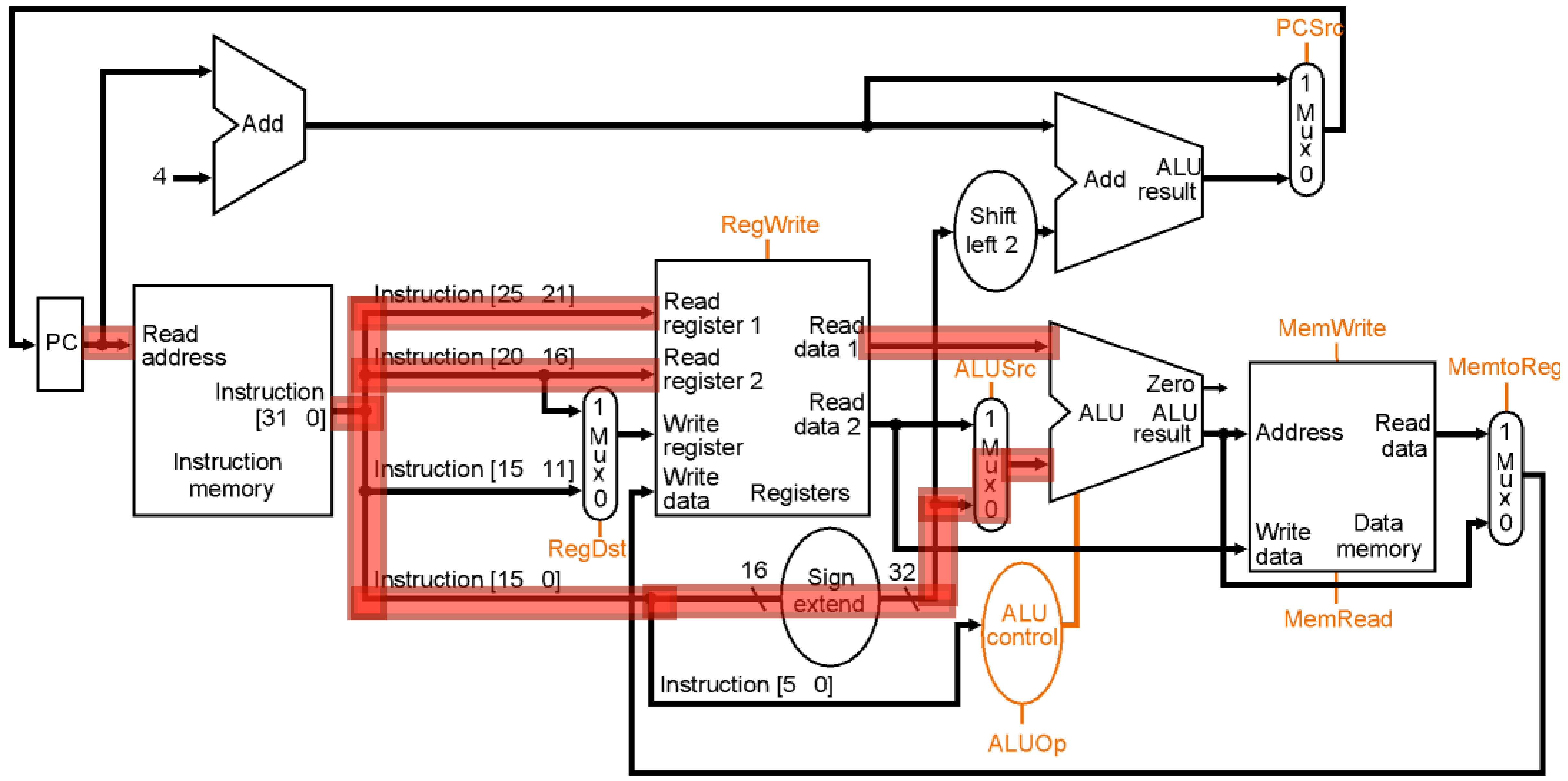
The Store Datapath



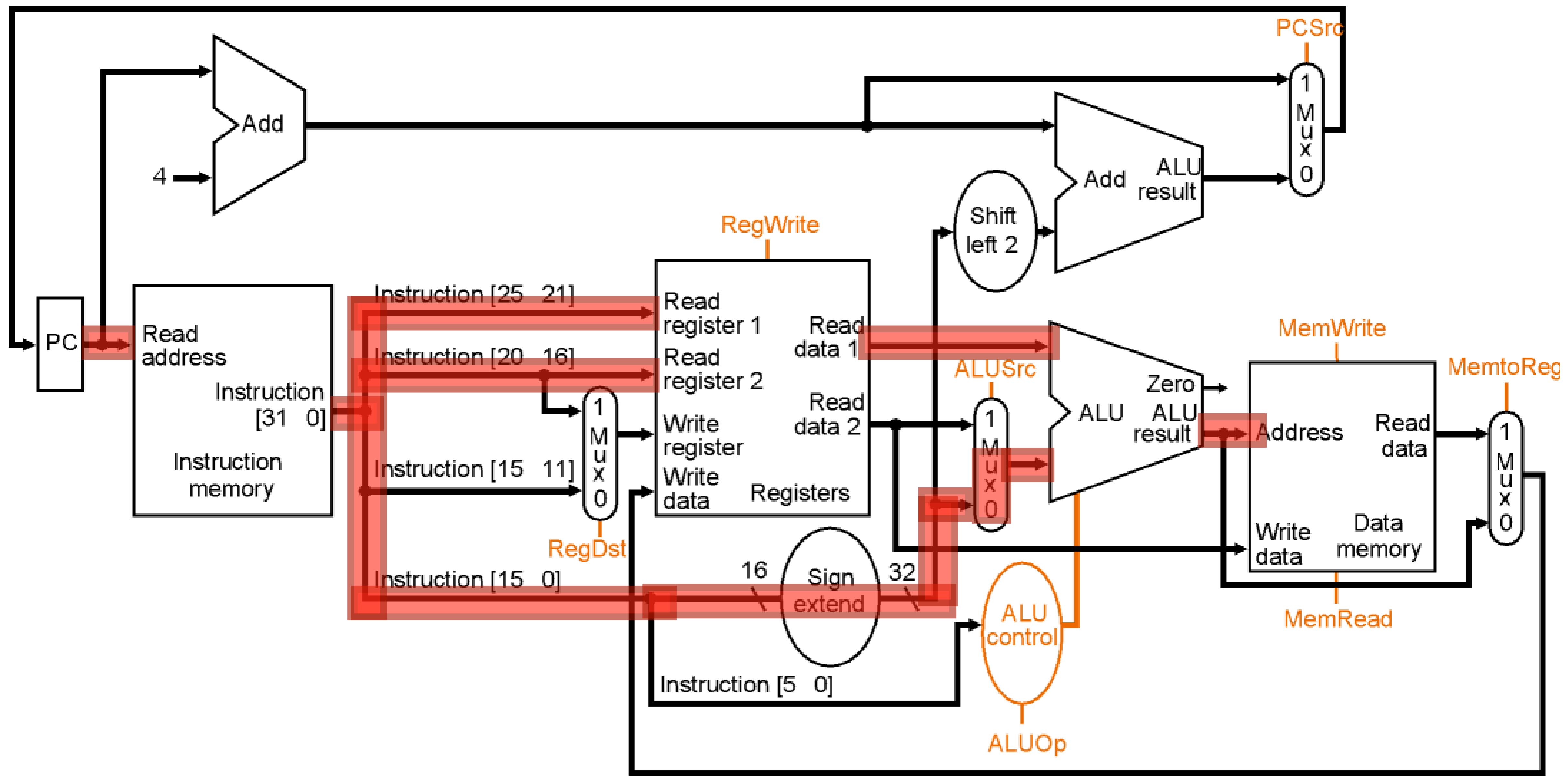
The Store Datapath



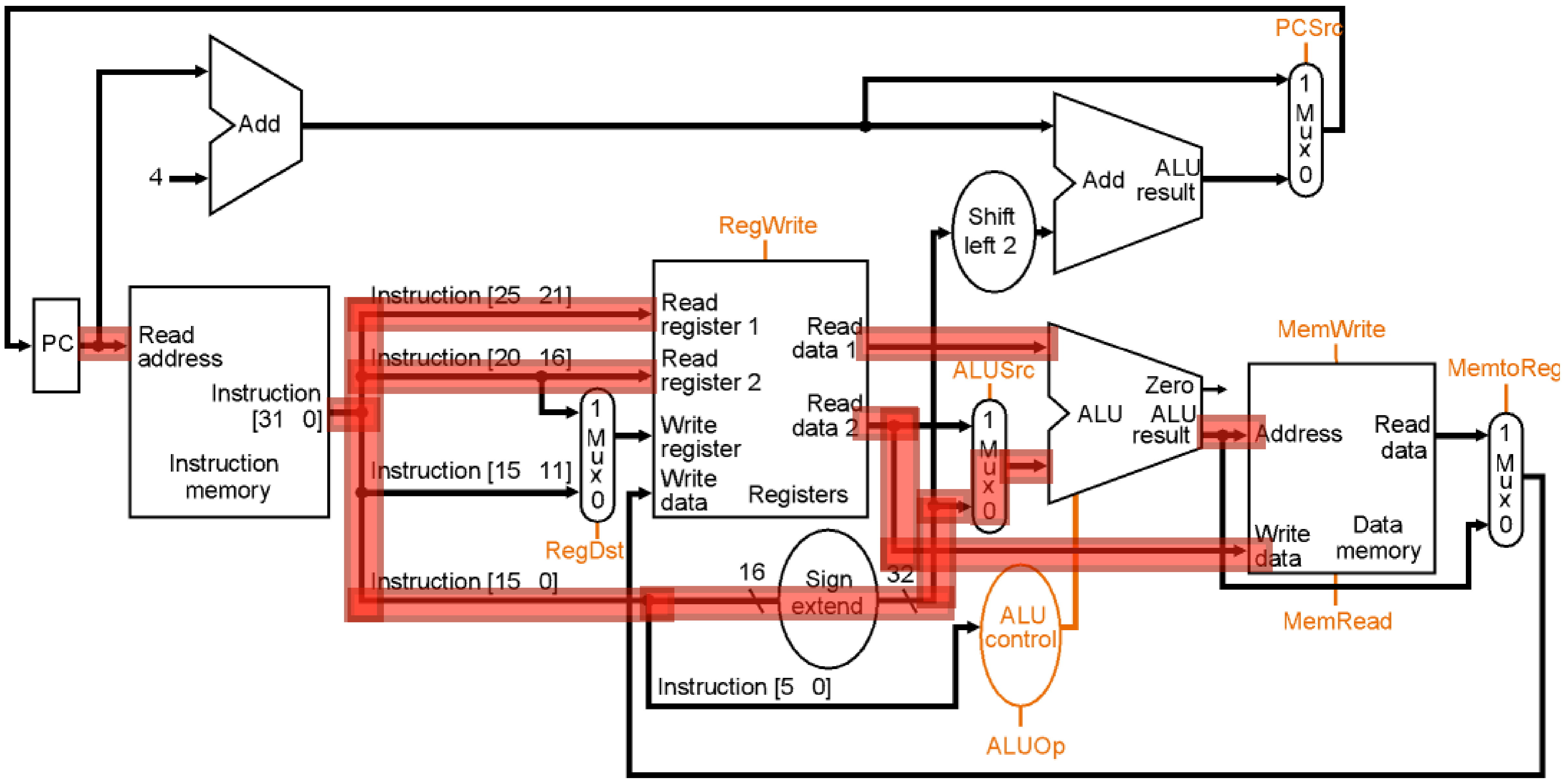
The Store Datapath



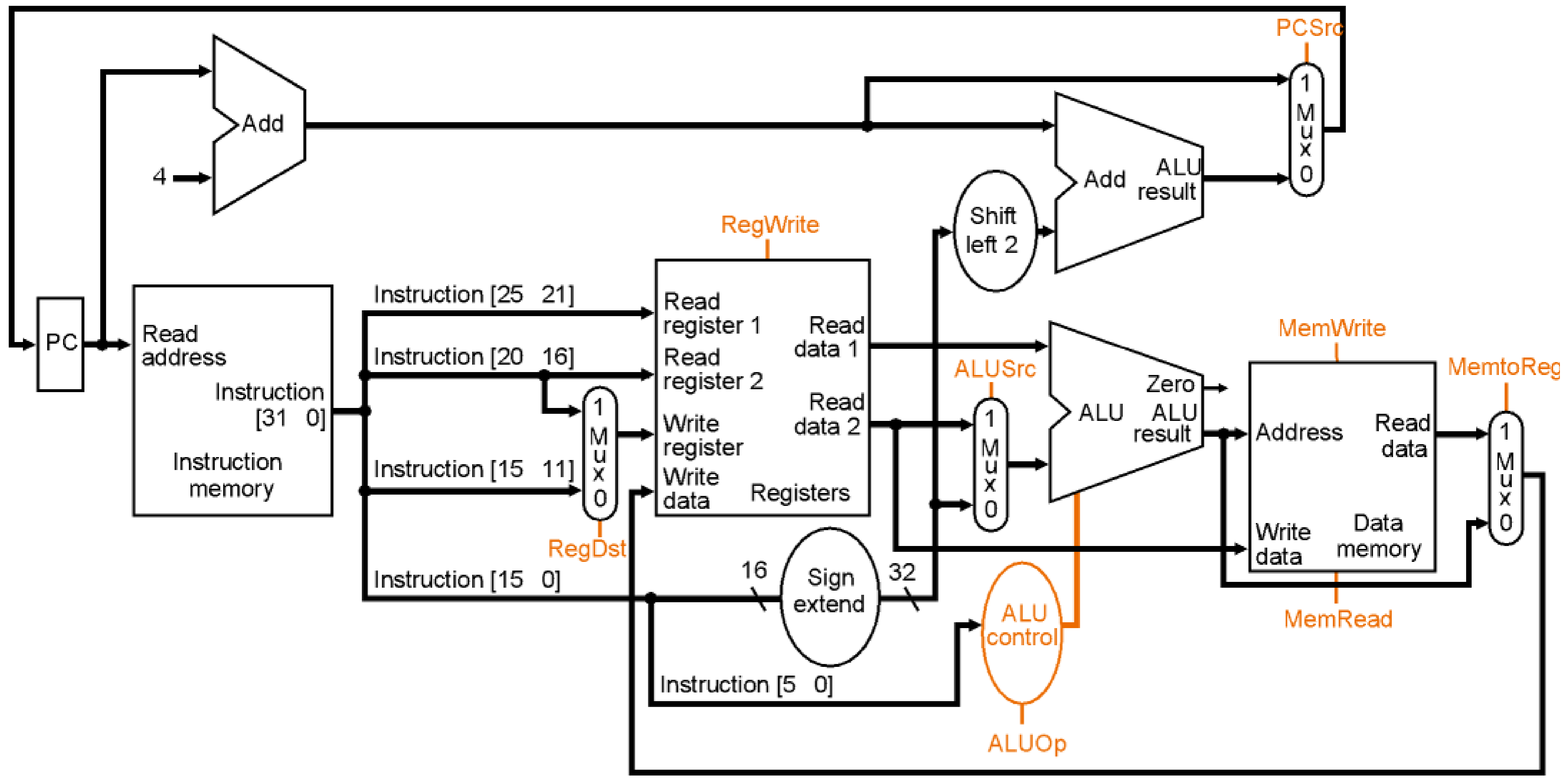
The Store Datapath



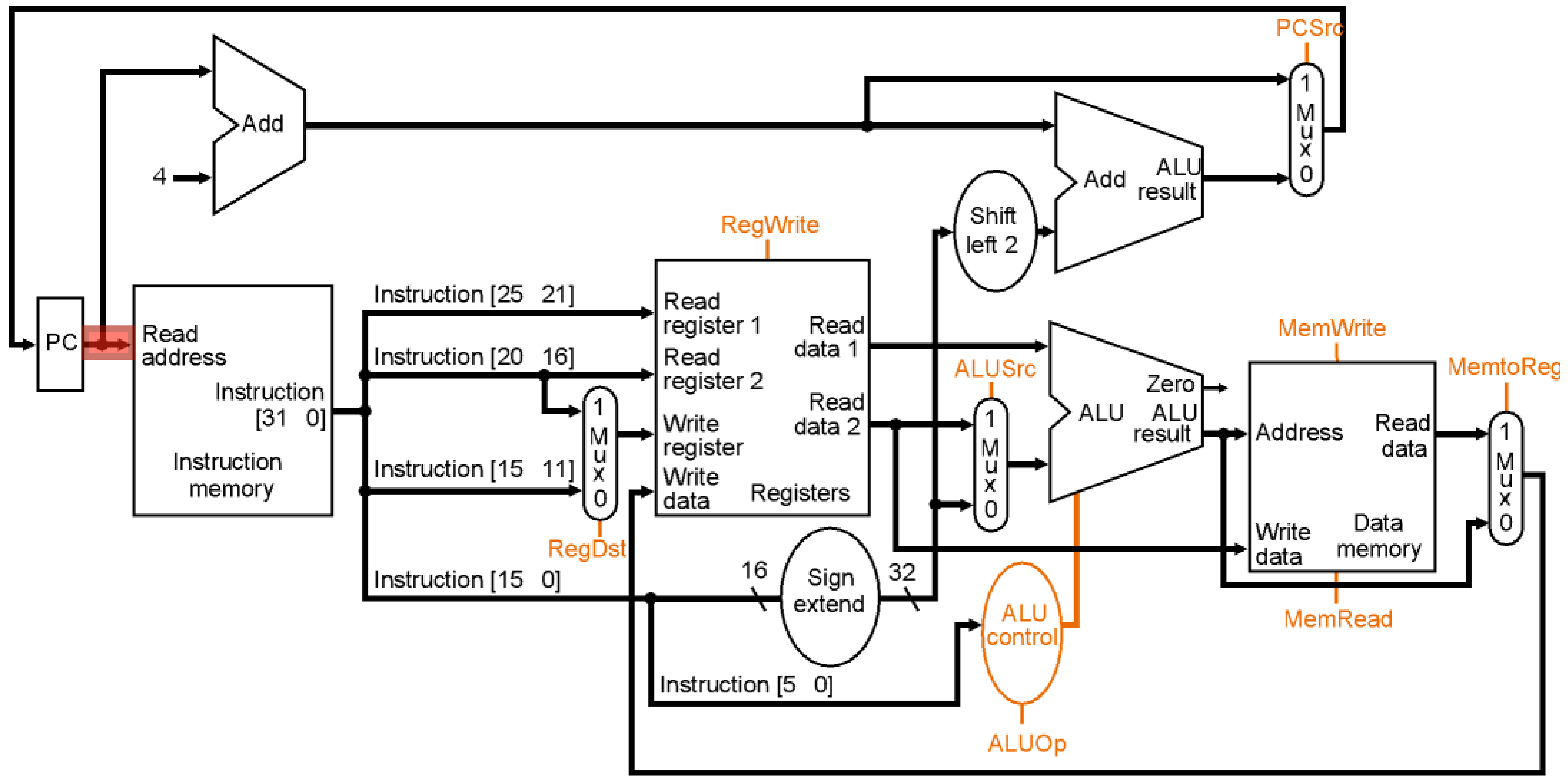
The Store Datapath



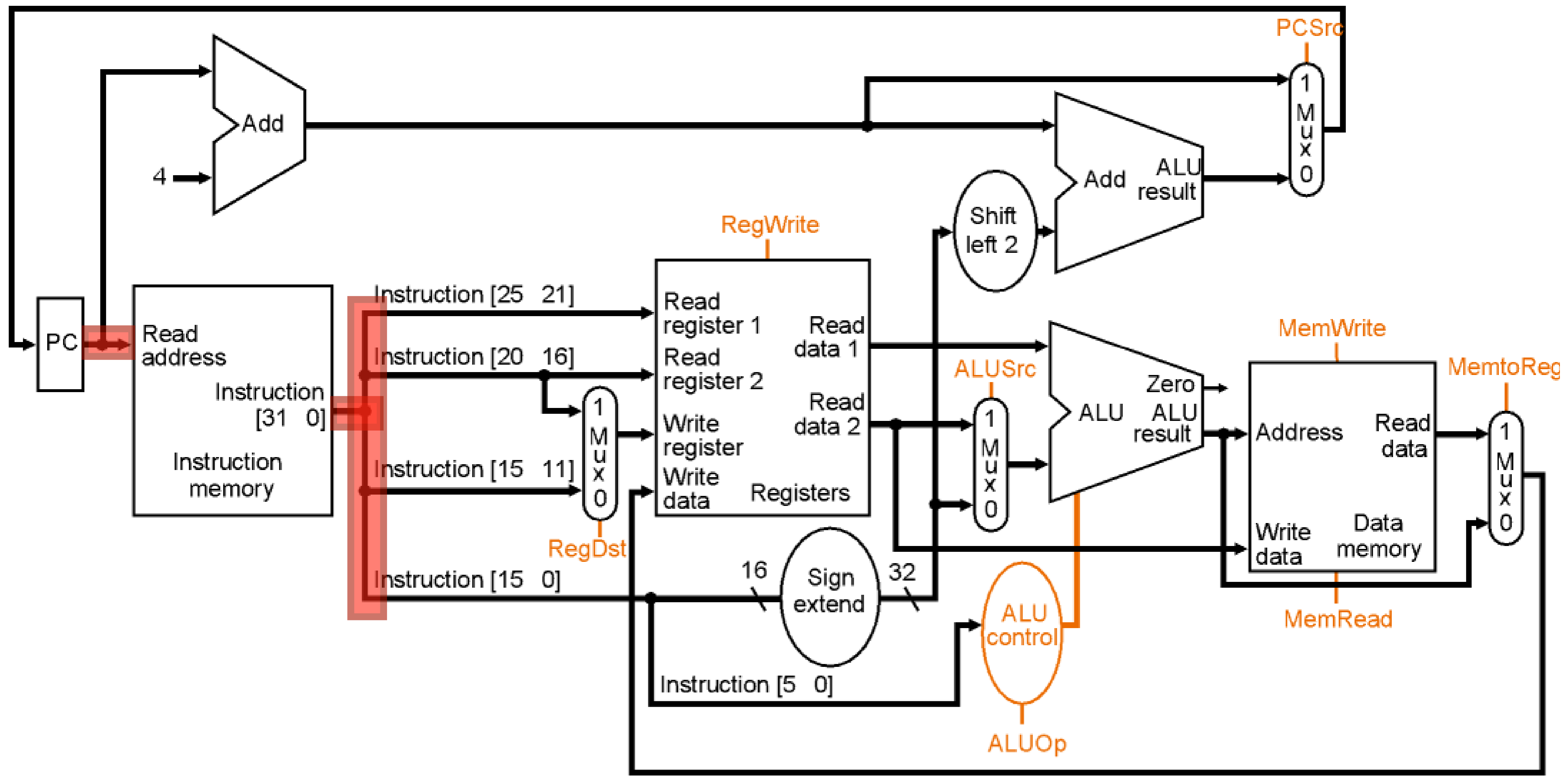
The Branch (beq) Datapath



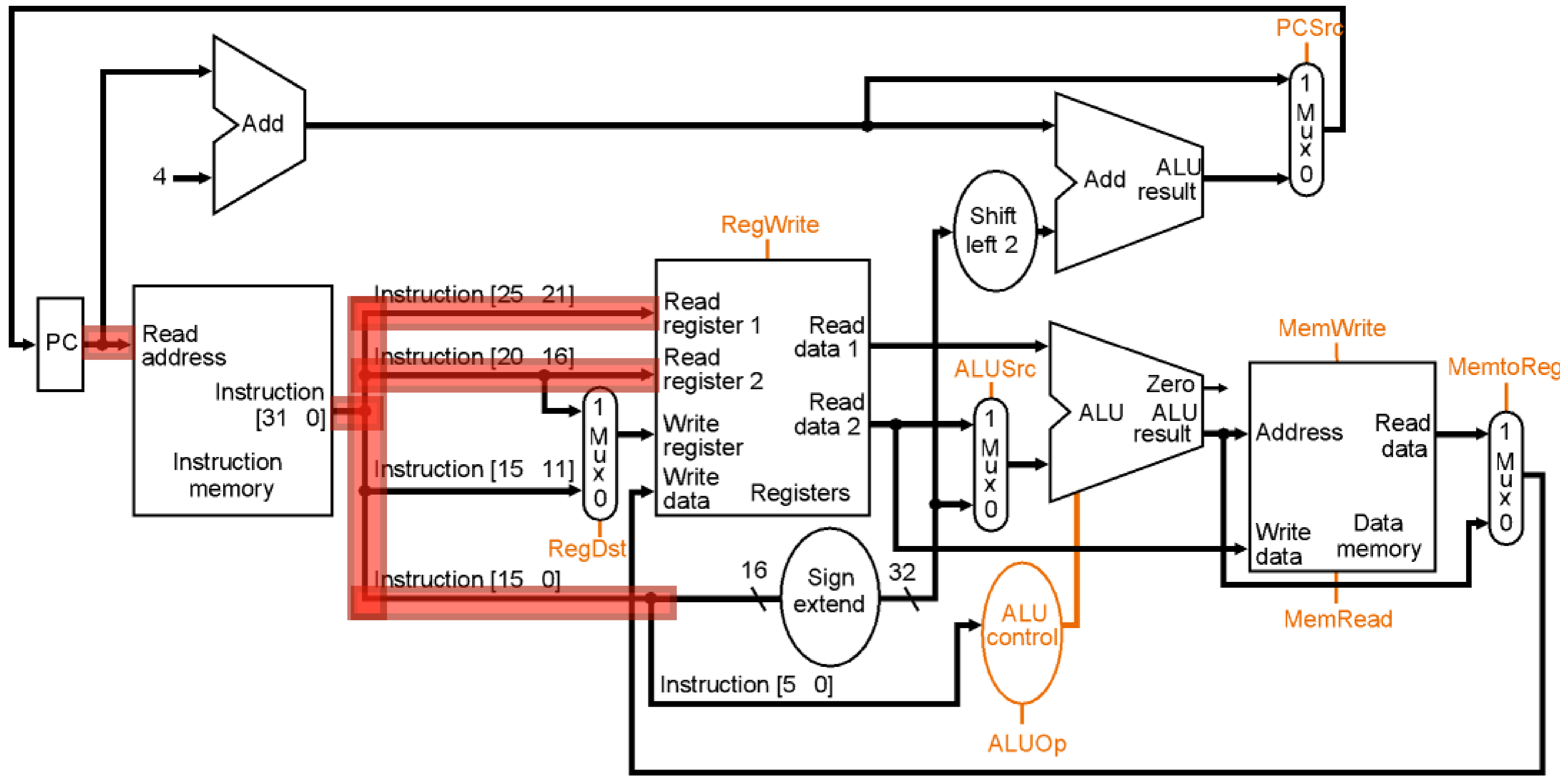
The Branch (beq) Datapath



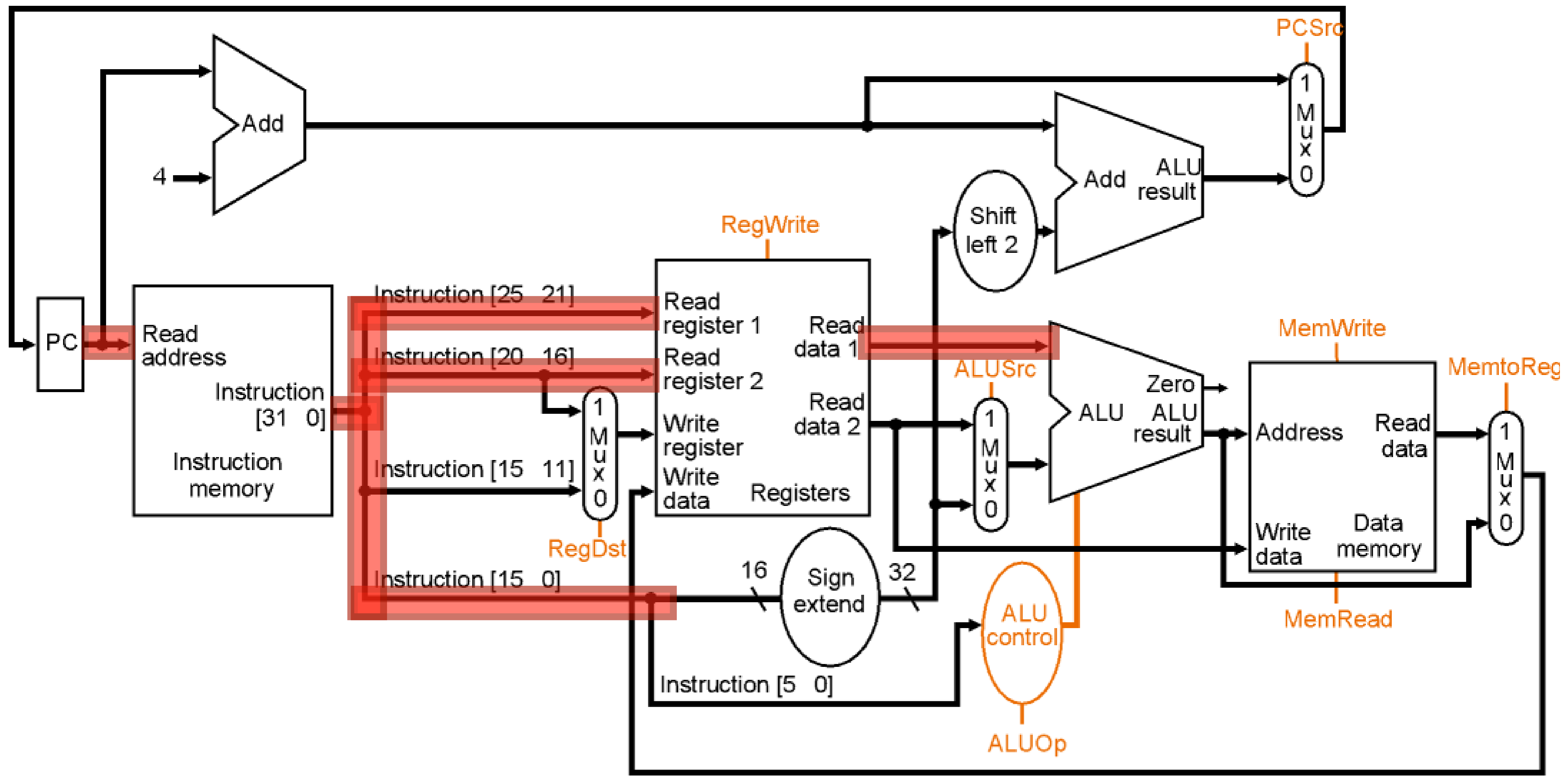
The Branch (beq) Datapath



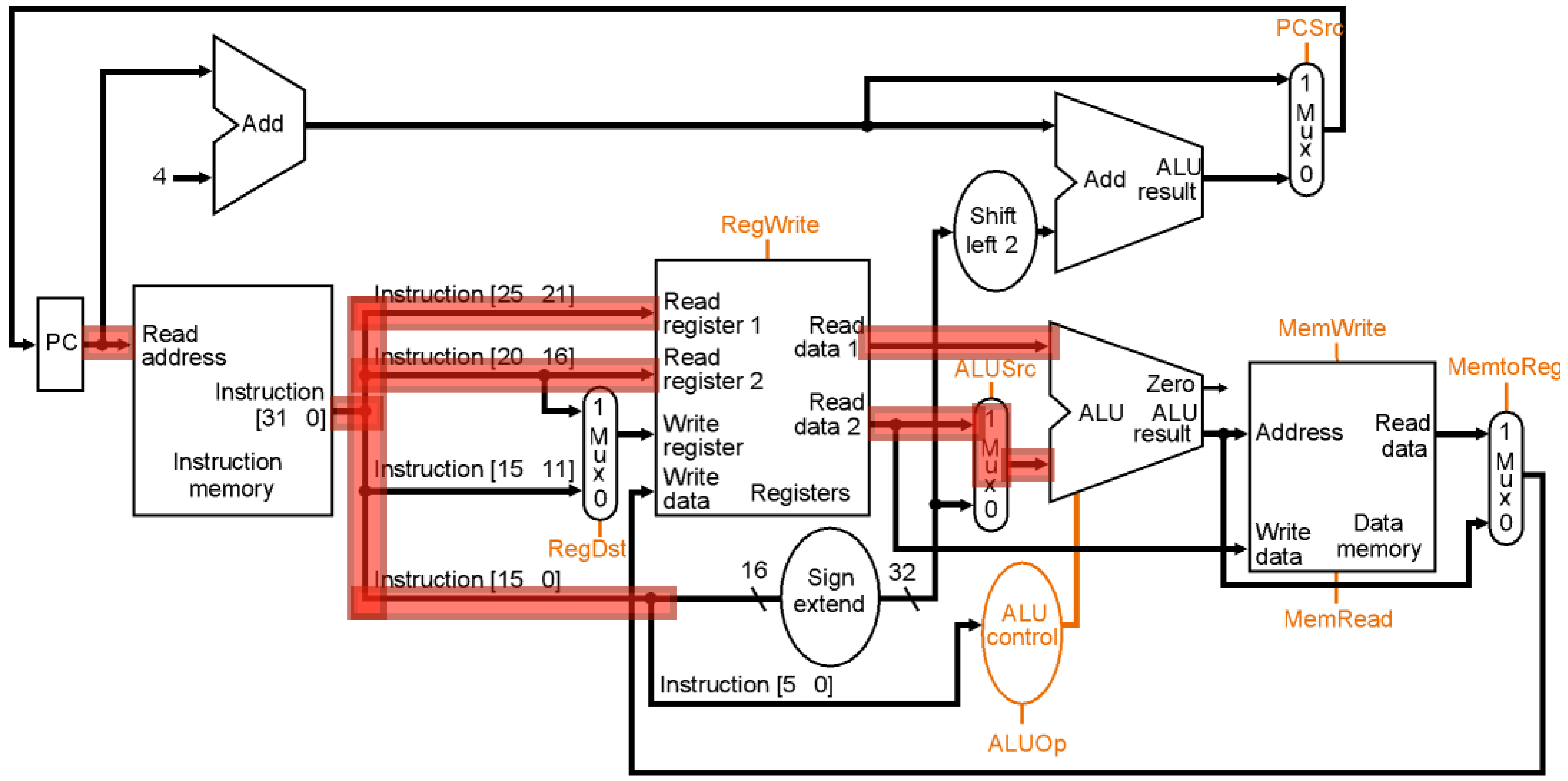
The Branch (beq) Datapath



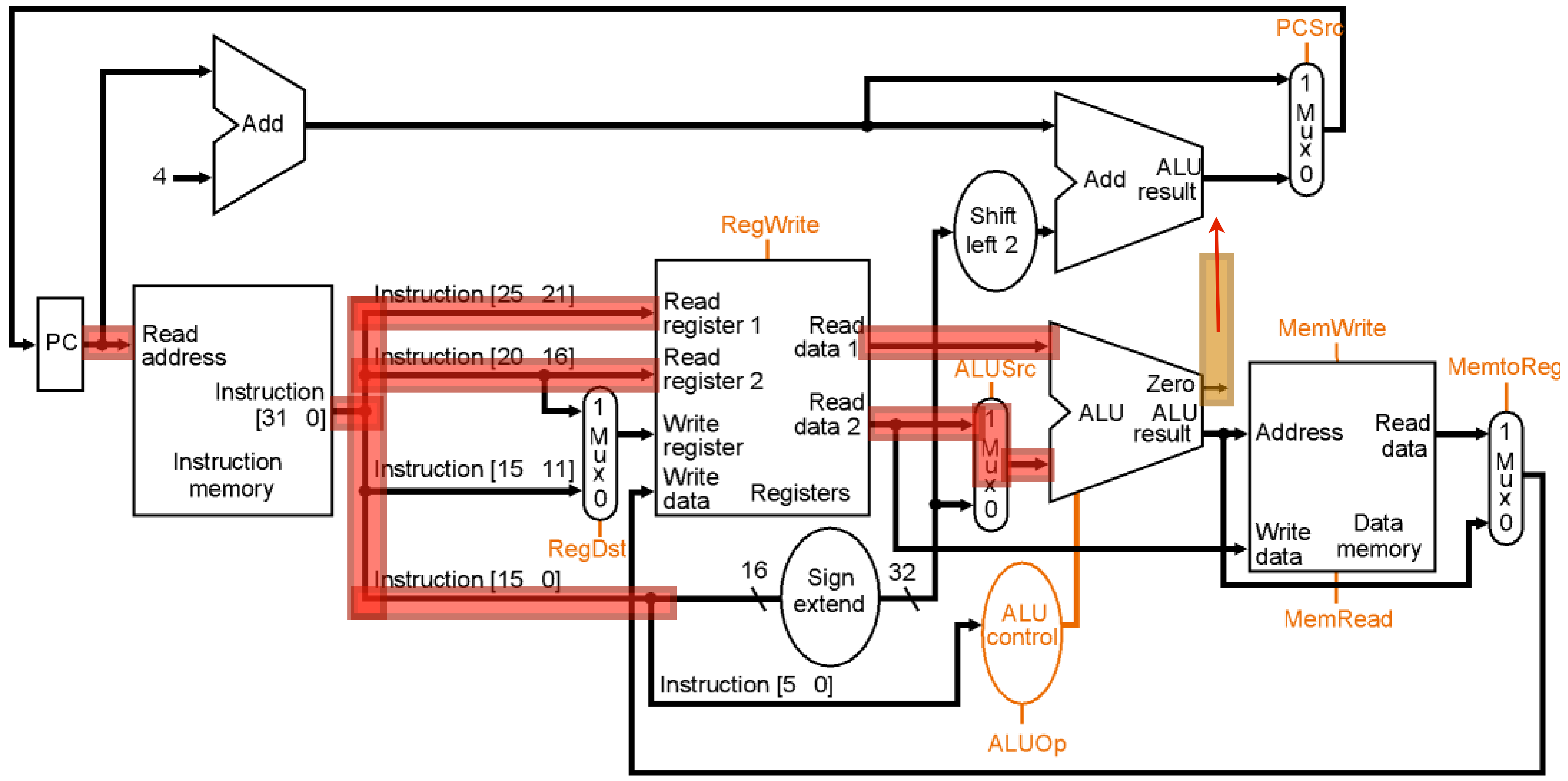
The Branch (beq) Datapath



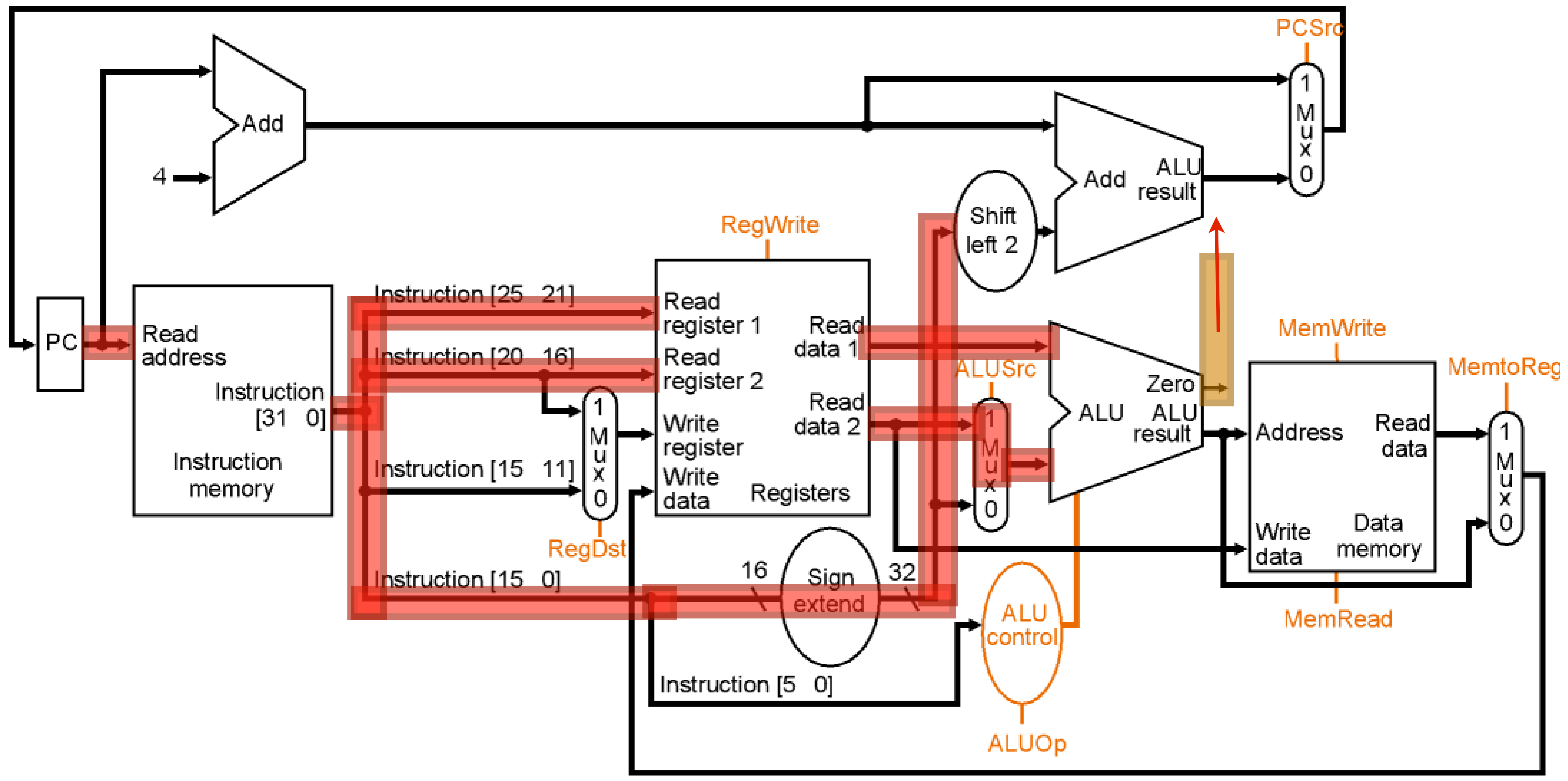
The Branch (beq) Datapath



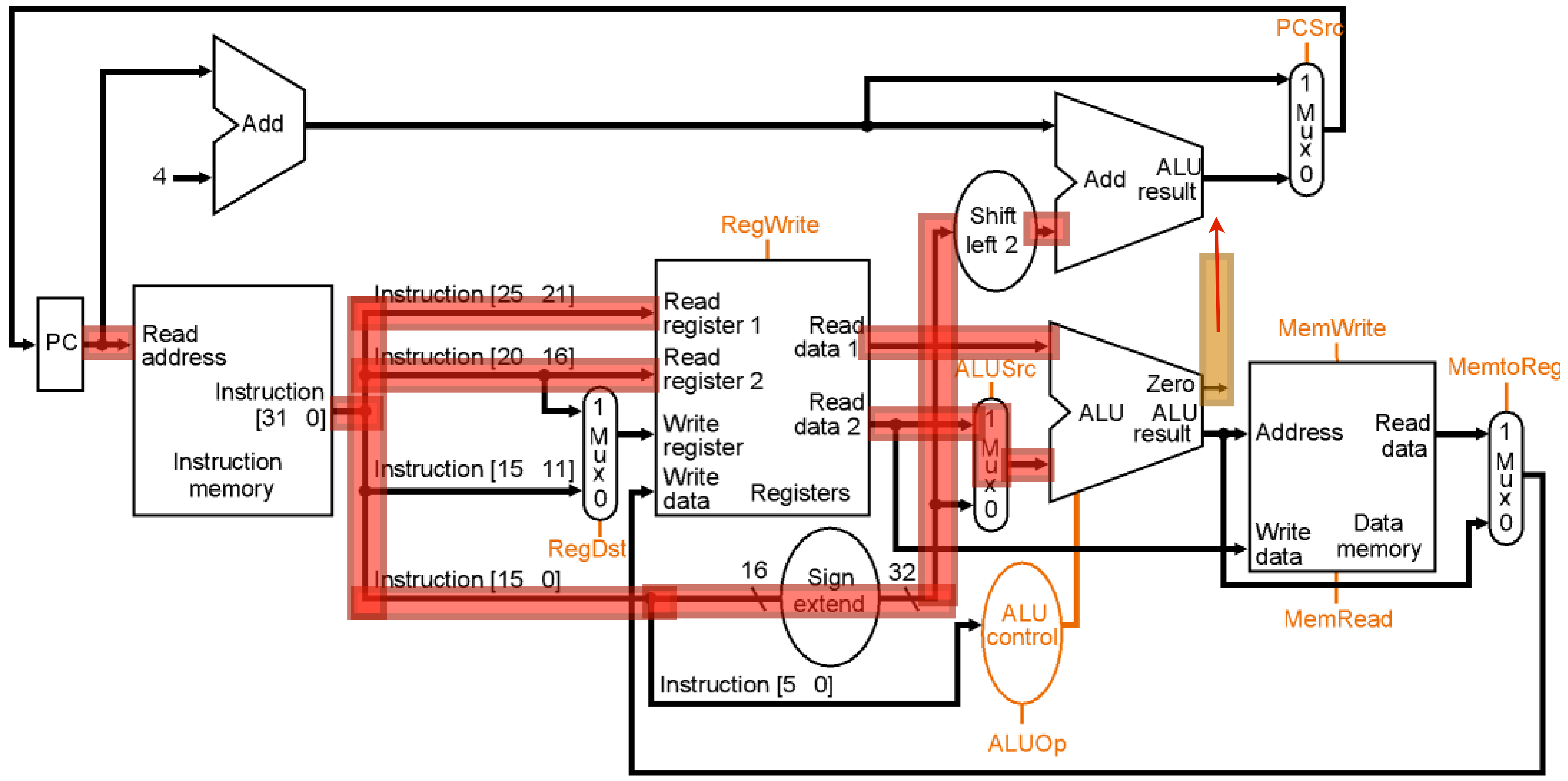
The Branch (beq) Datapath



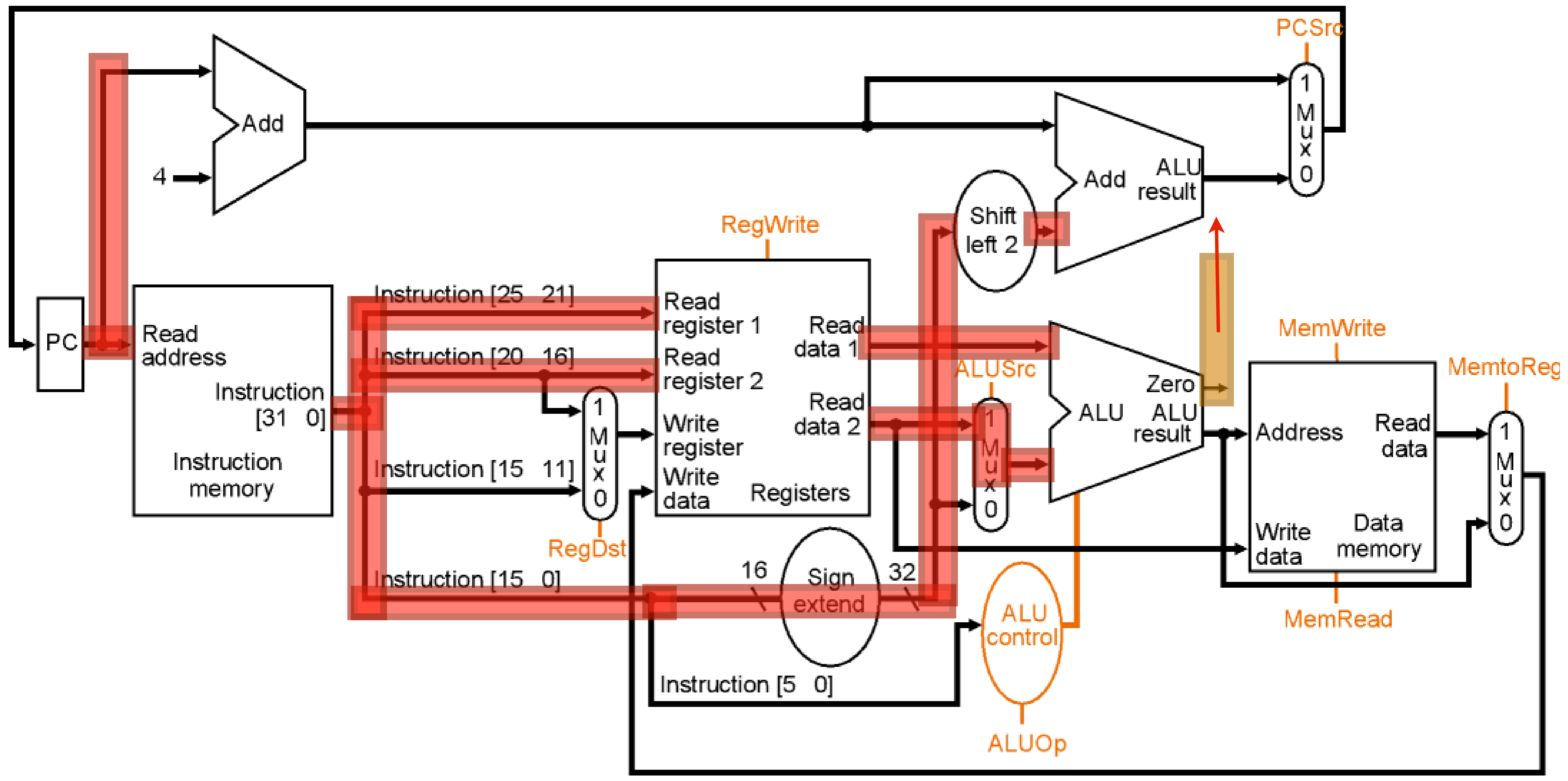
The Branch (beq) Datapath



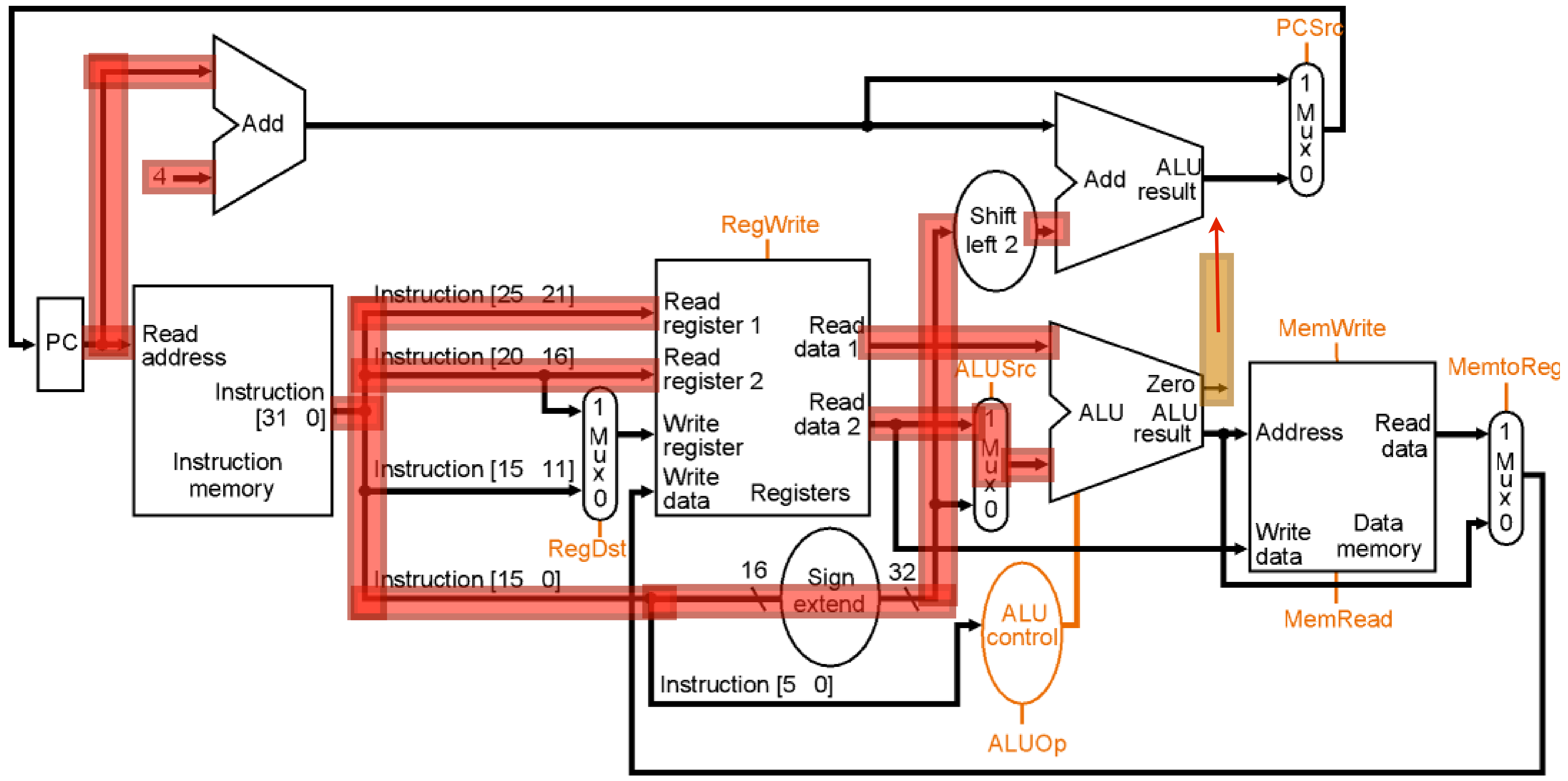
The Branch (beq) Datapath



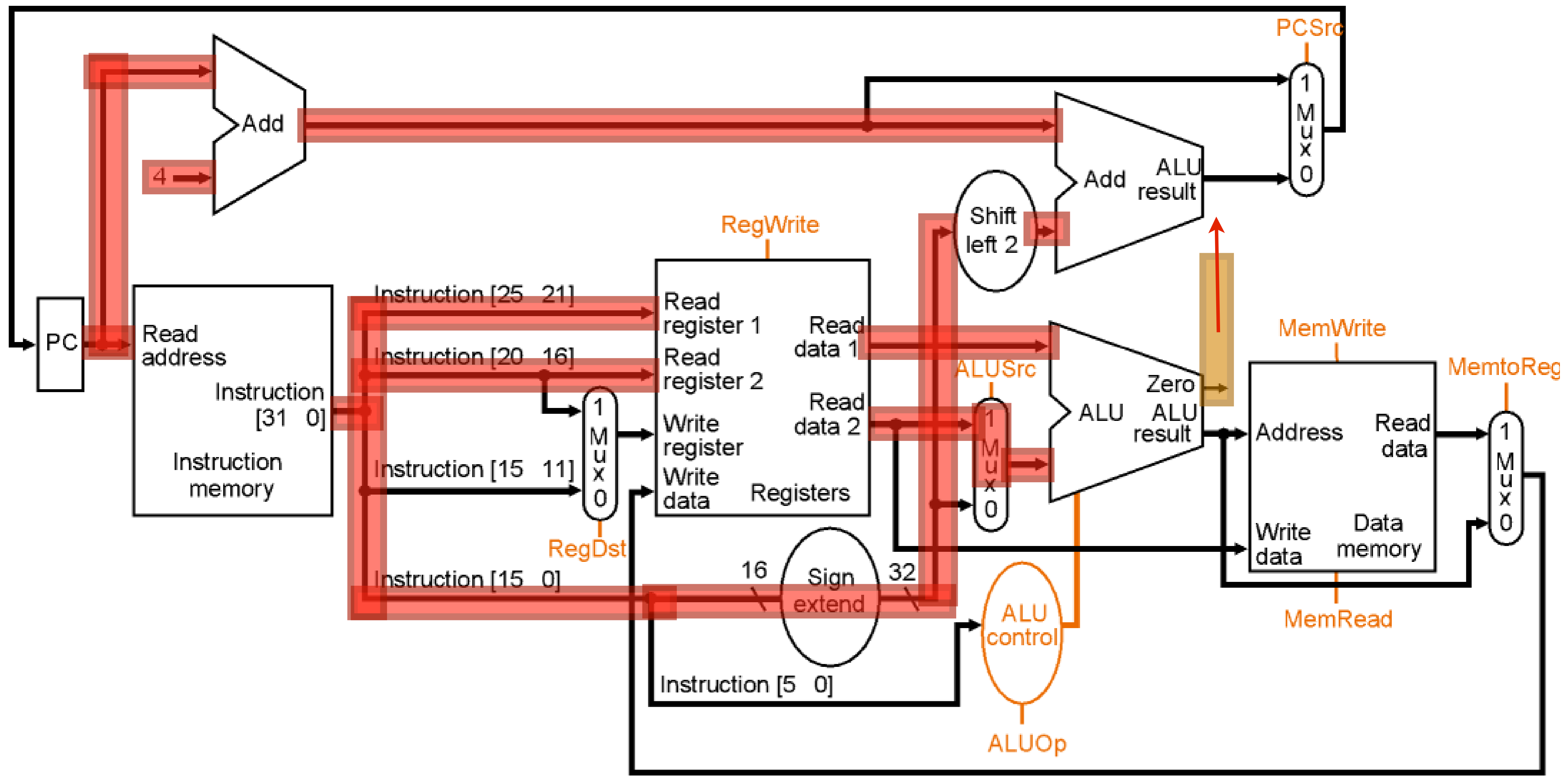
The Branch (beq) Datapath



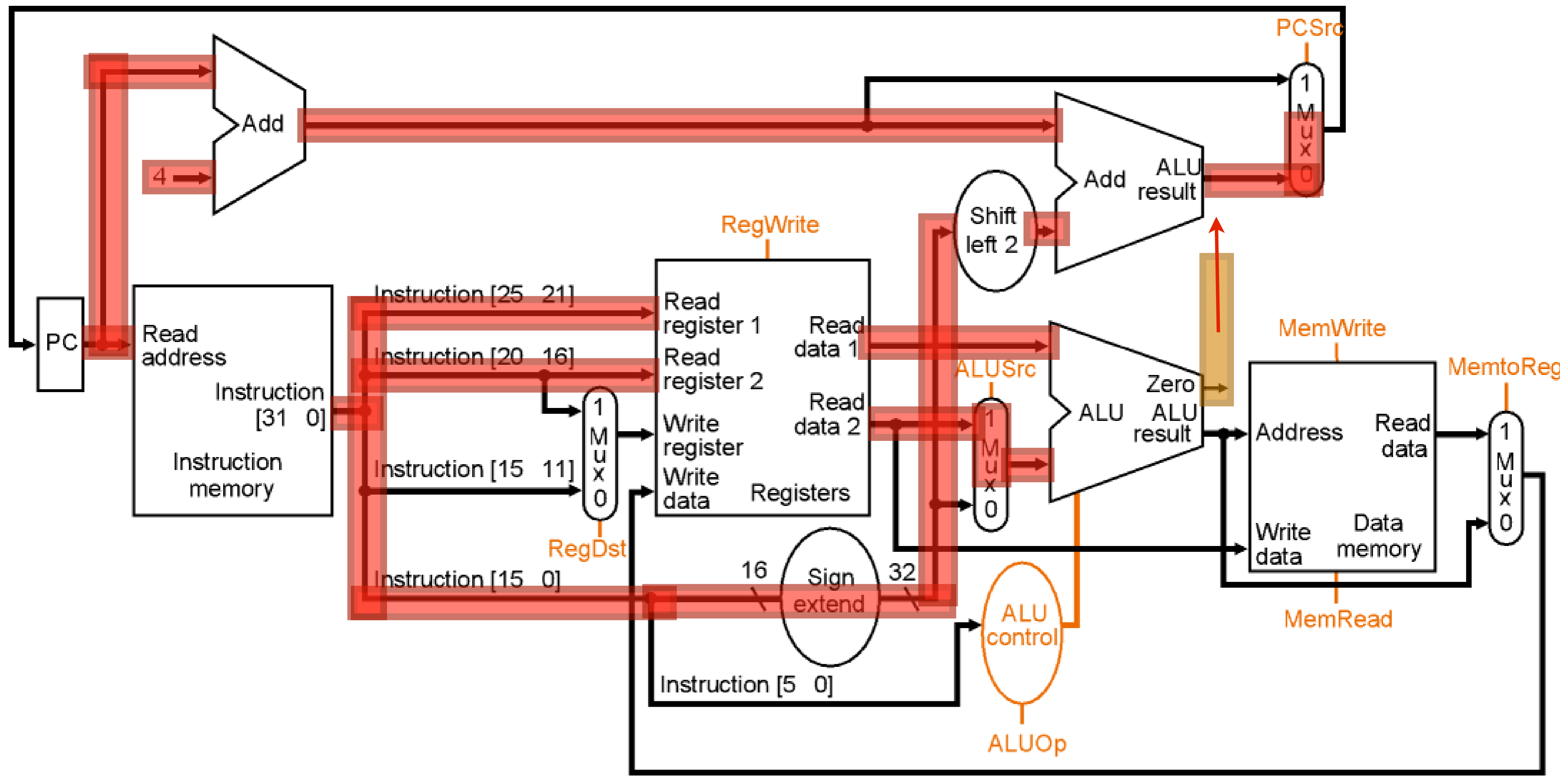
The Branch (beq) Datapath



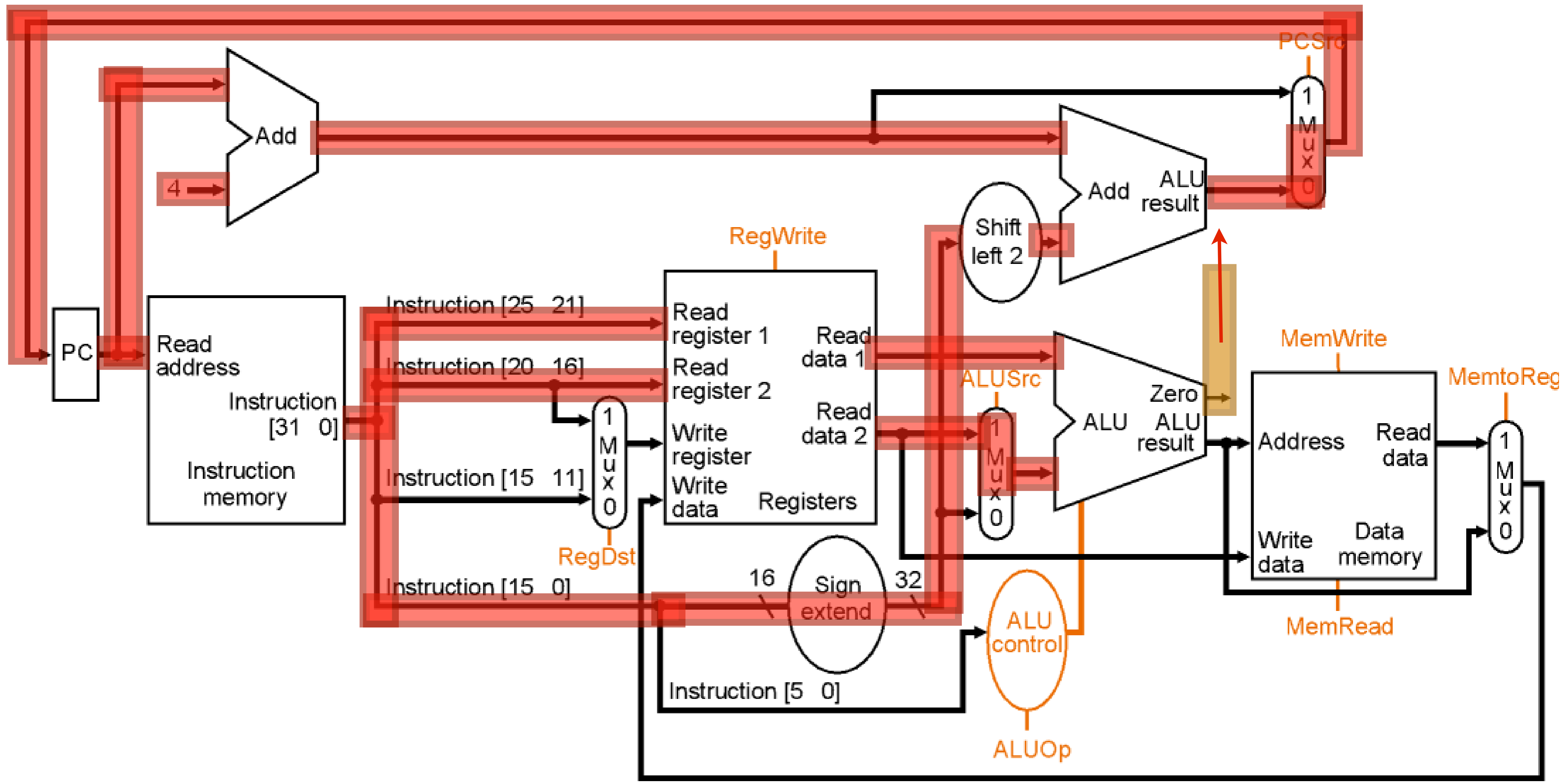
The Branch (beq) Datapath



The Branch (beq) Datapath

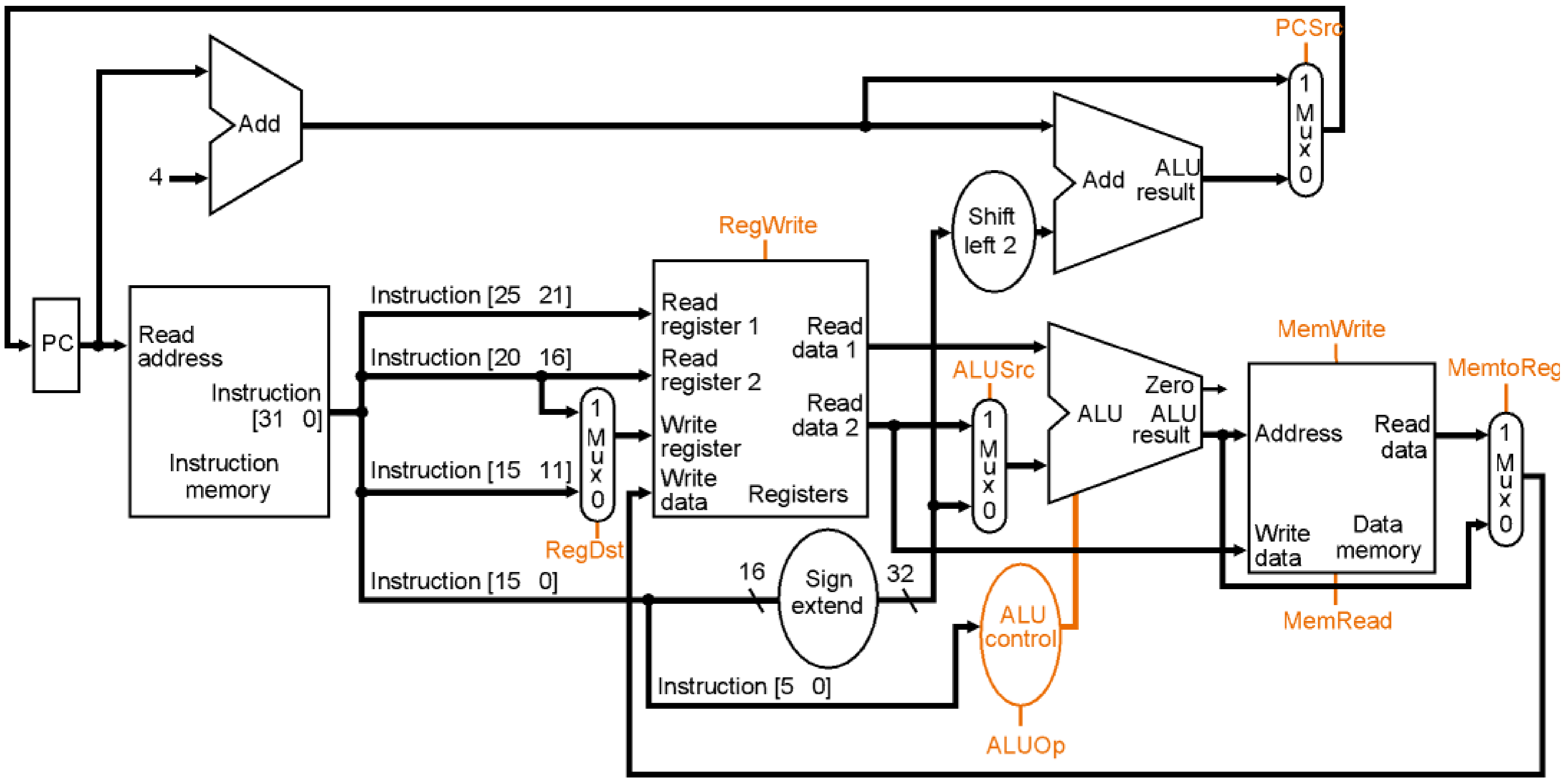


The Branch (beq) Datapath



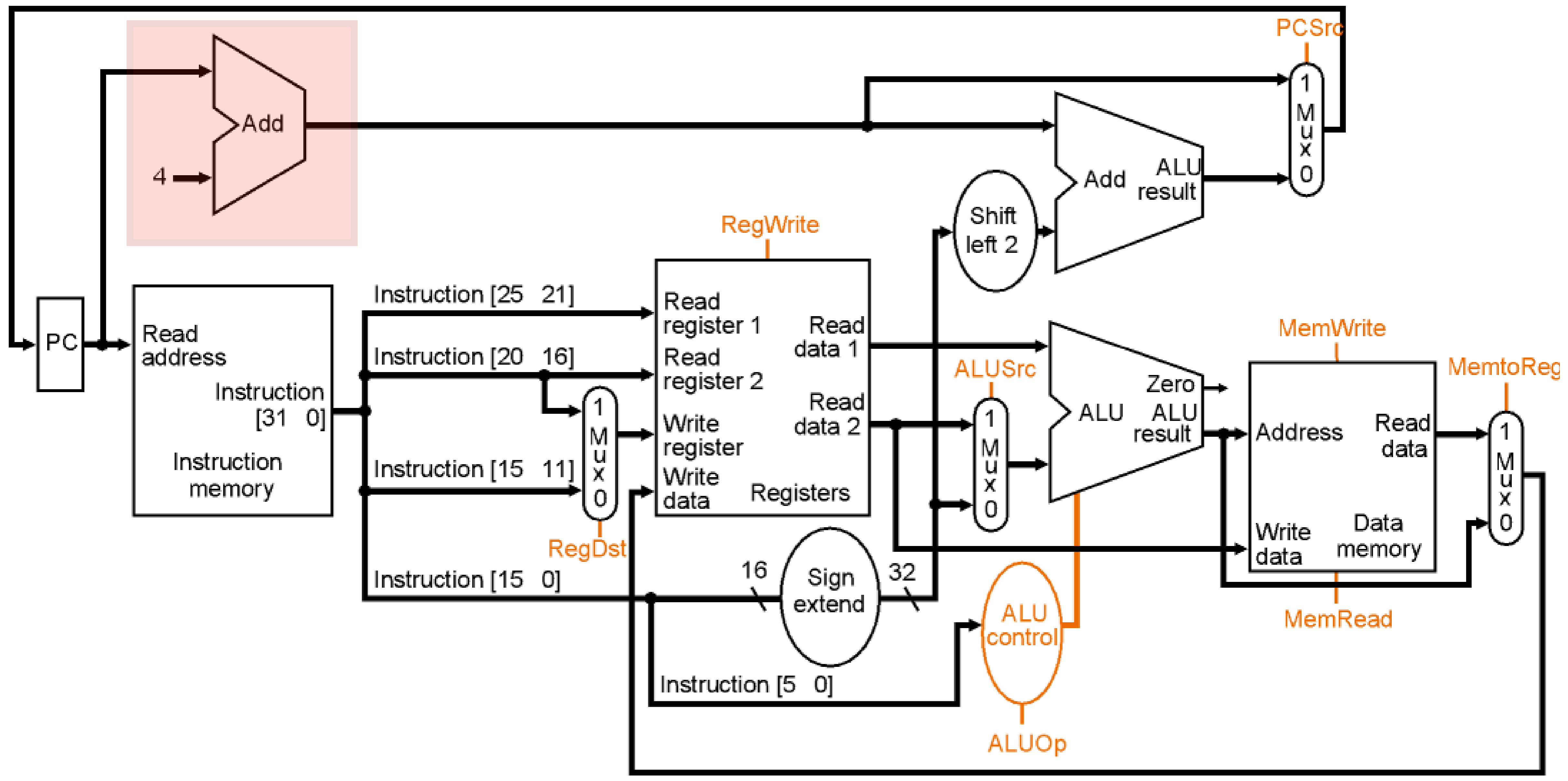
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



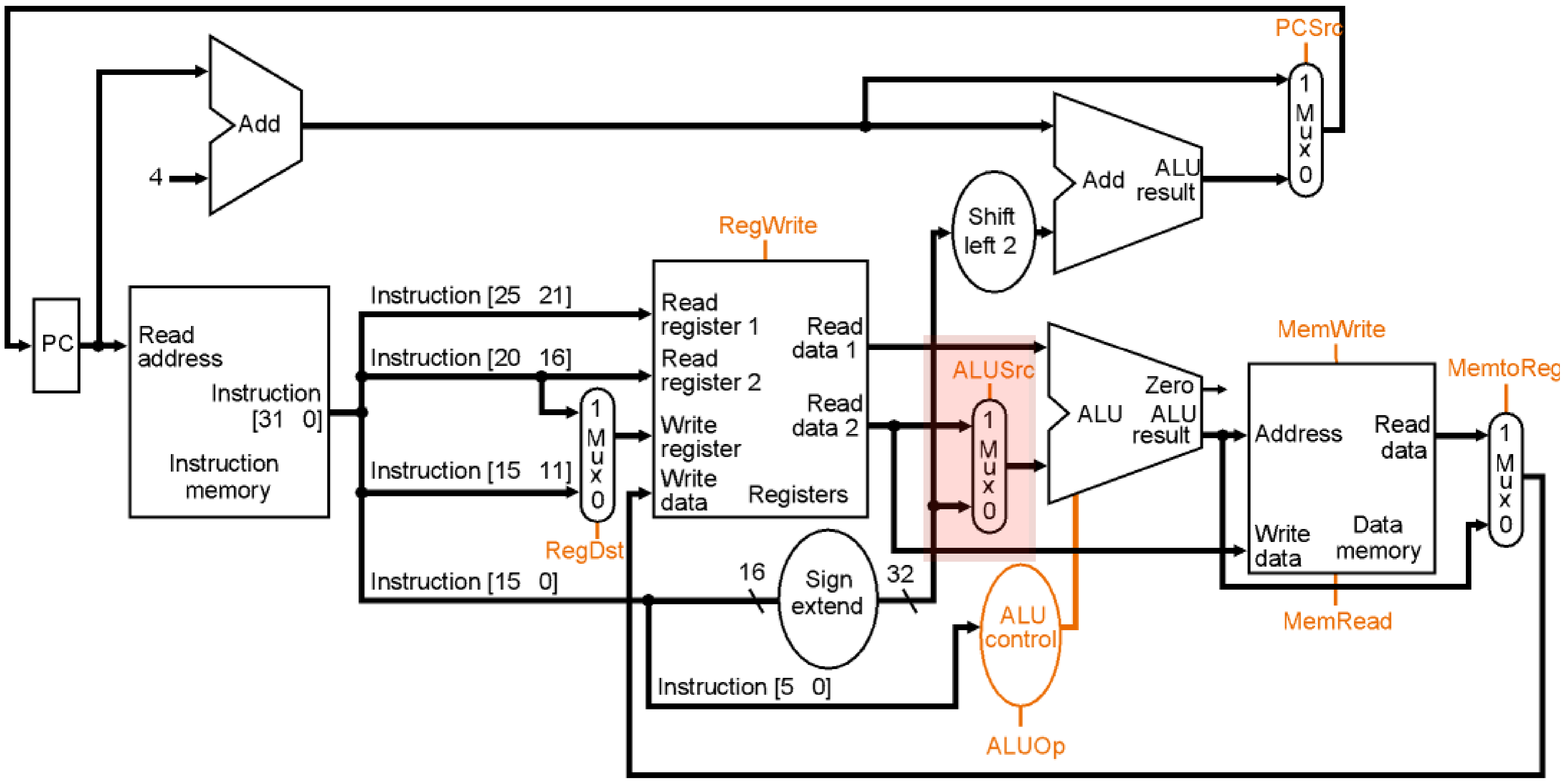
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



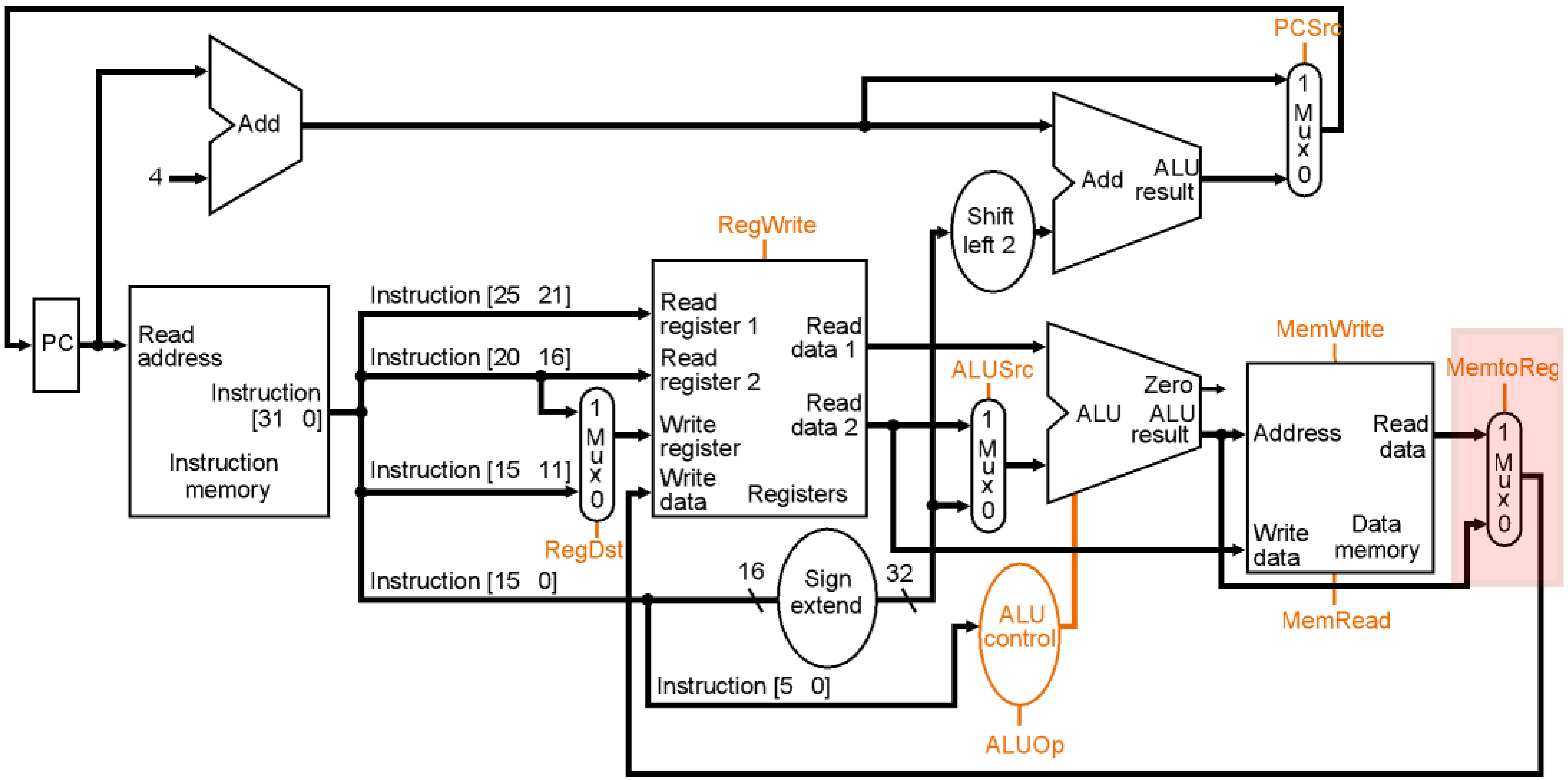
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



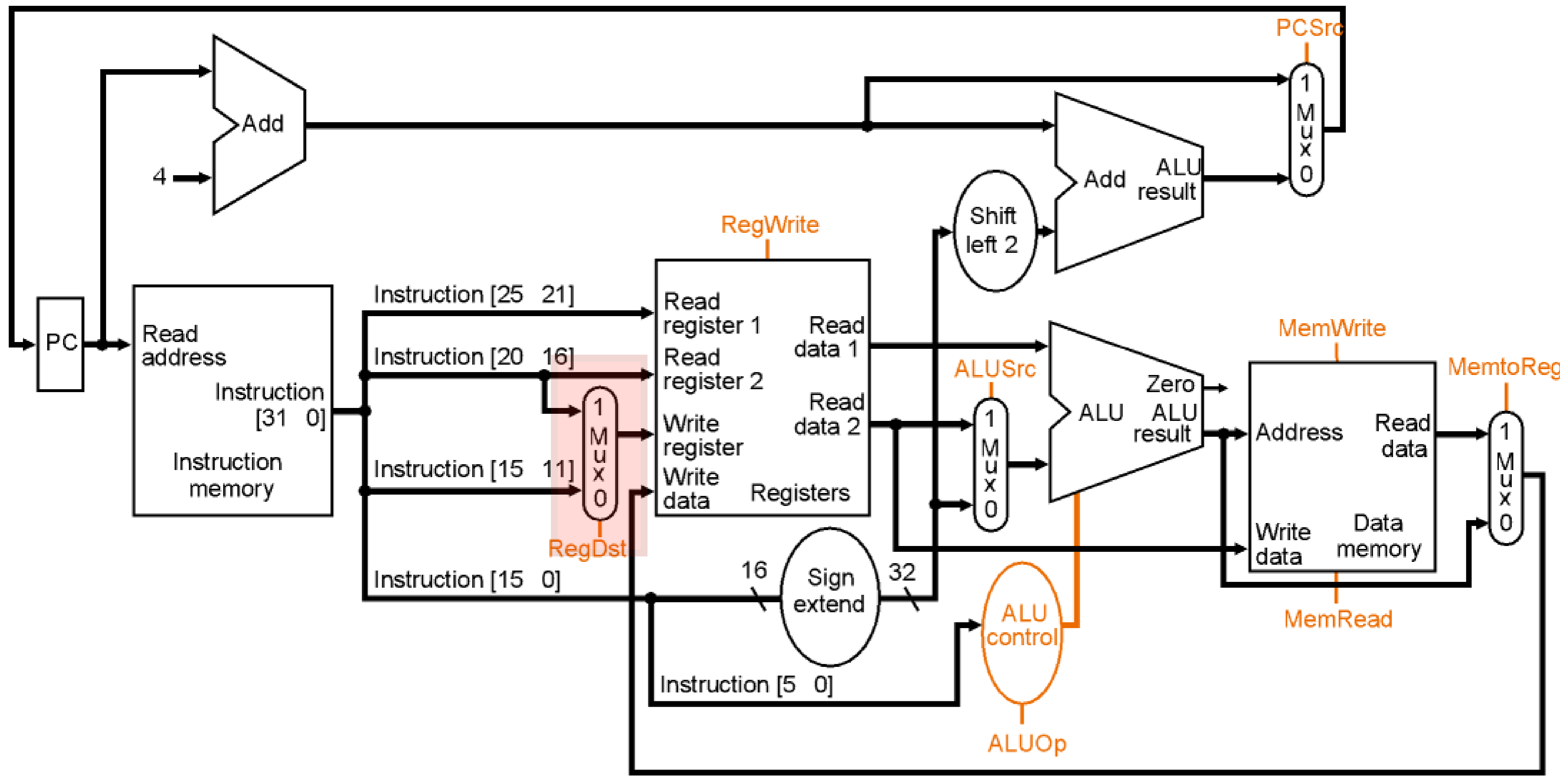
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



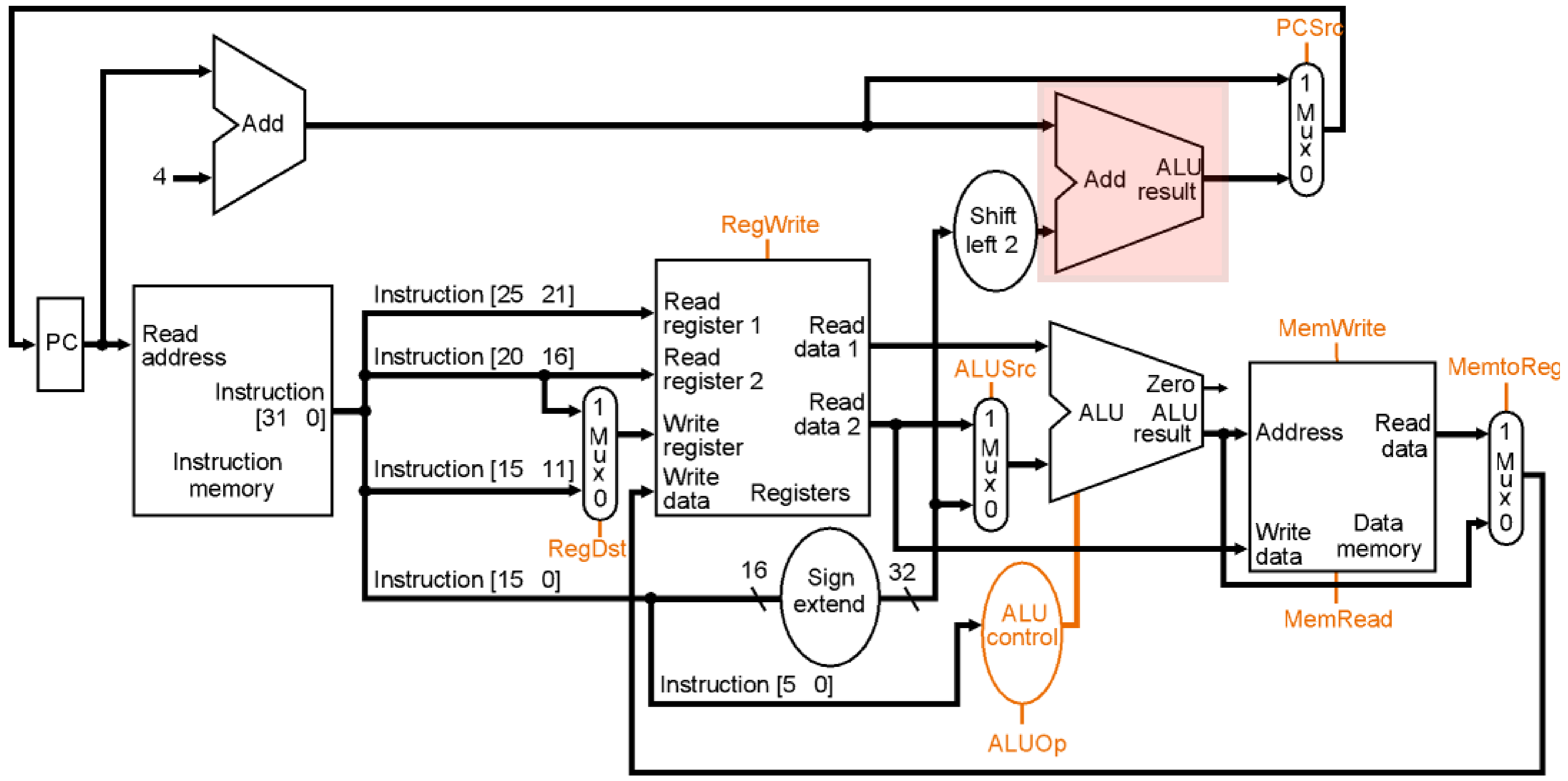
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



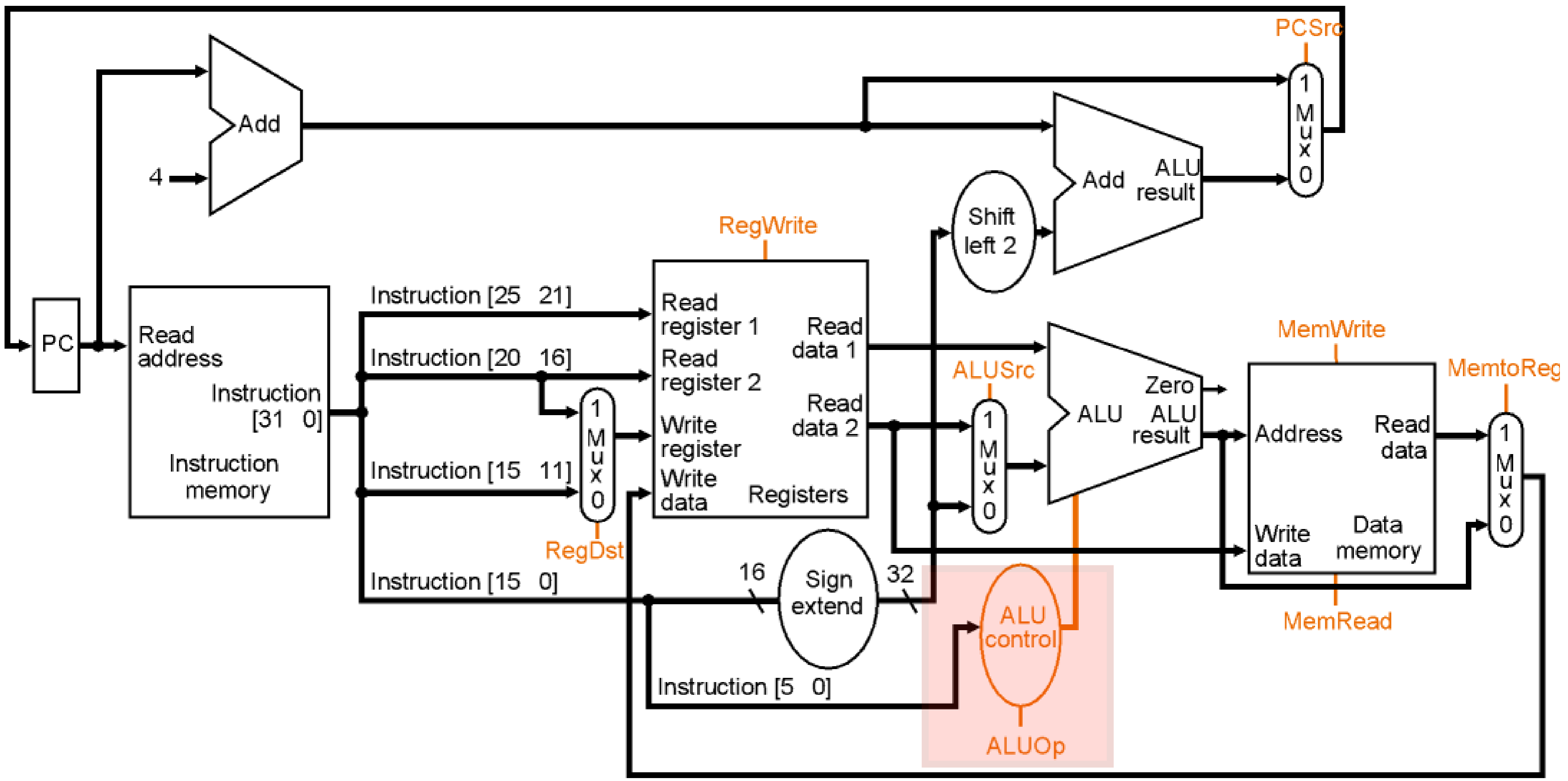
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



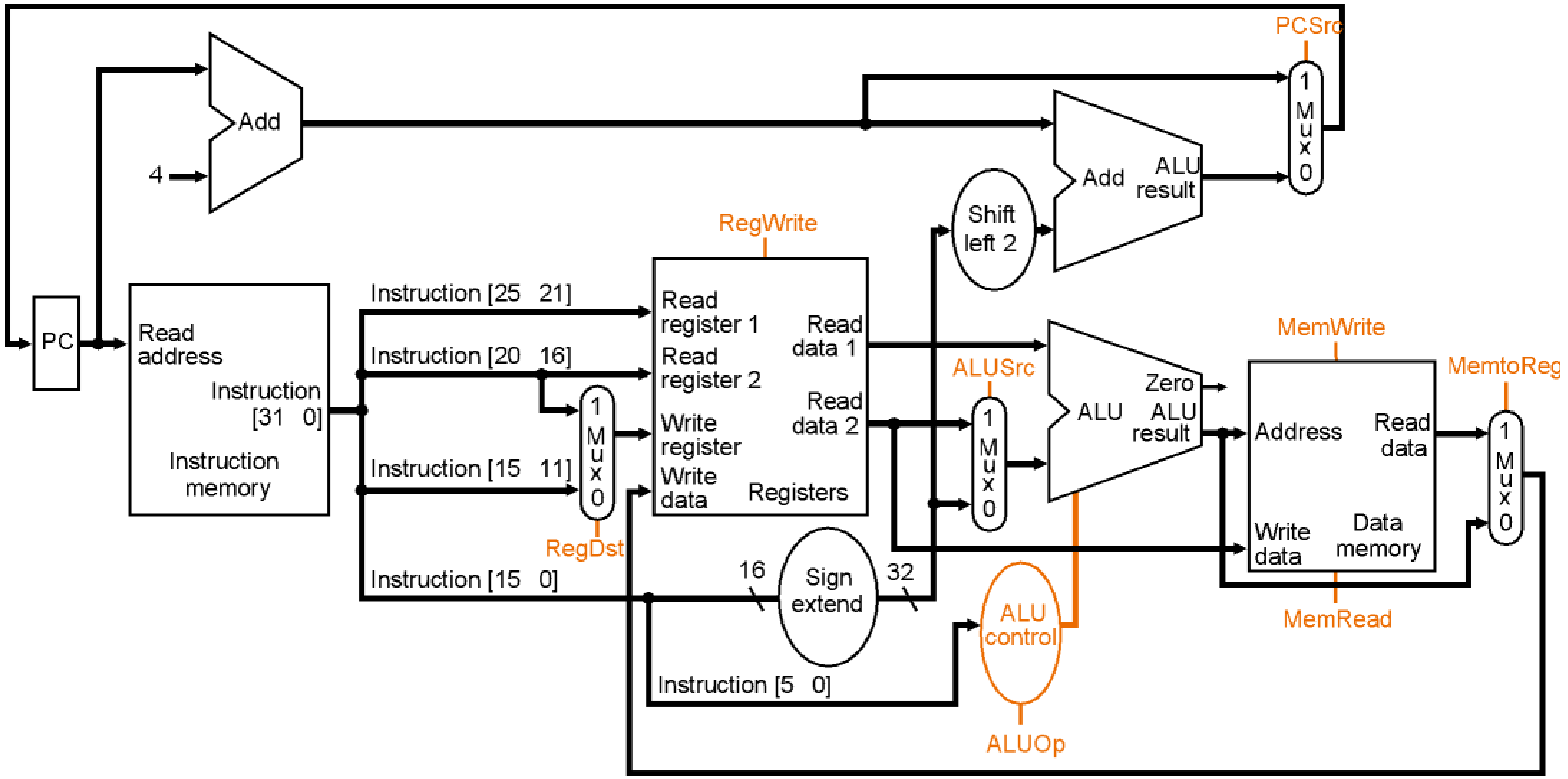
GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



GAME: GUESS THE FUNCTION!! (Review)

- We have everything except control signals



Key Points

Key Points

- CPU is just a collection of state and combinational logic

Key Points

- CPU is just a collection of state and combinational logic
- We just designed a very rich processor, at least in terms of functionality

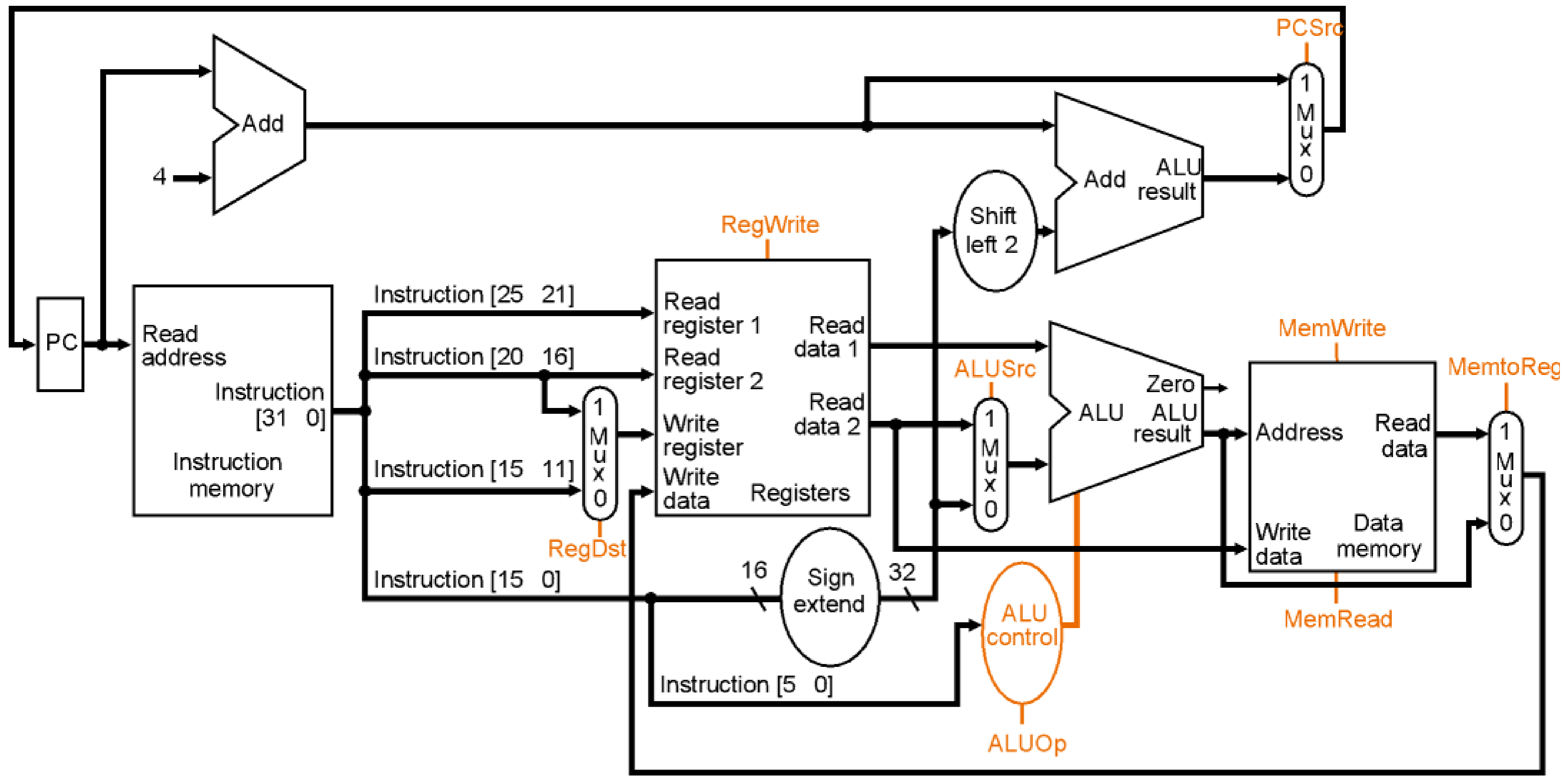
Key Points

- CPU is just a collection of state and combinational logic
- We just designed a very rich processor, at least in terms of functionality
- ET = IC * CPI * Cycle Time
    - where does the single-cycle machine fit in?

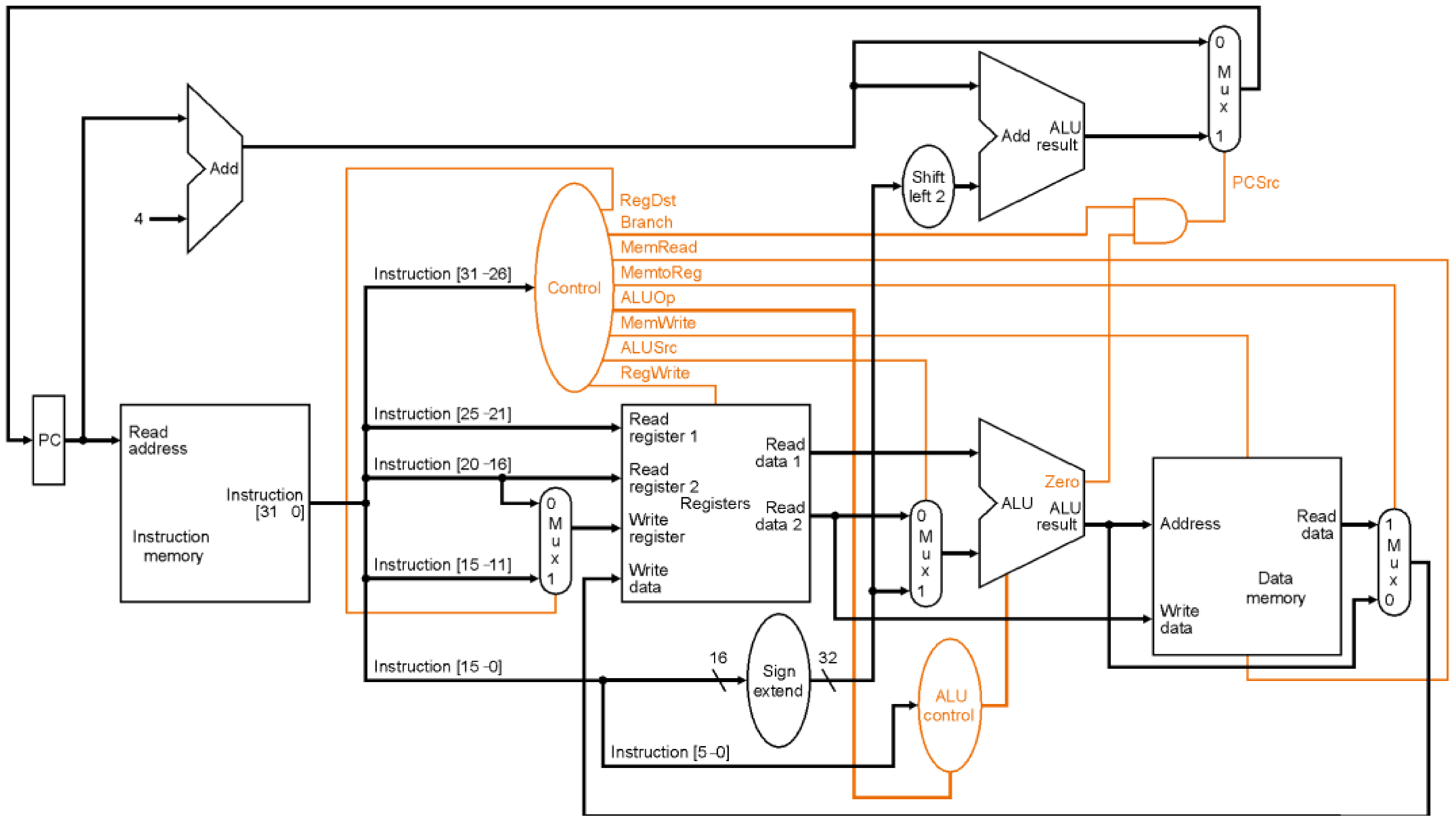
The Control Unit

Putting it All Together: A Single Cycle Datapath

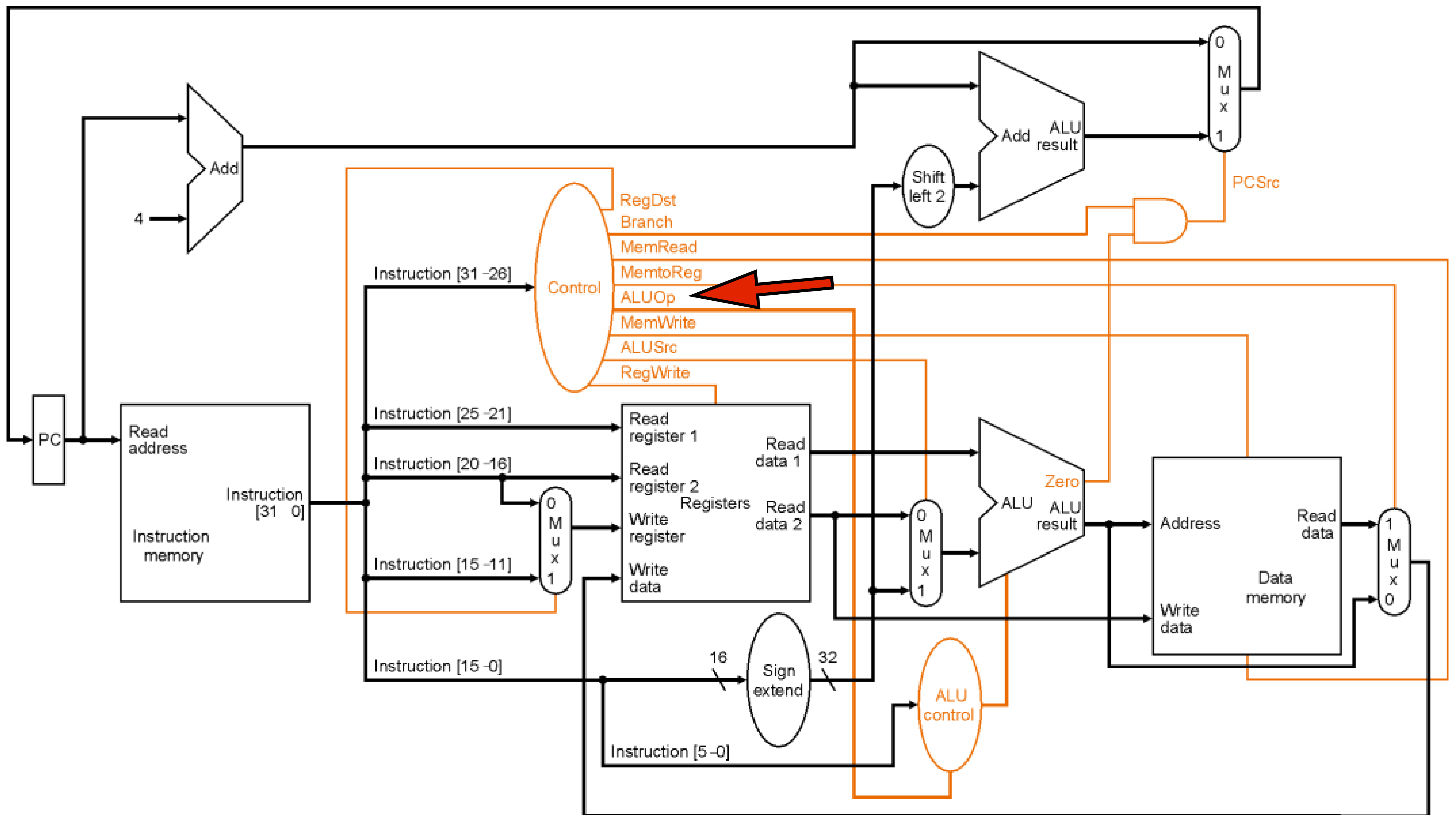
- We have everything except control signals



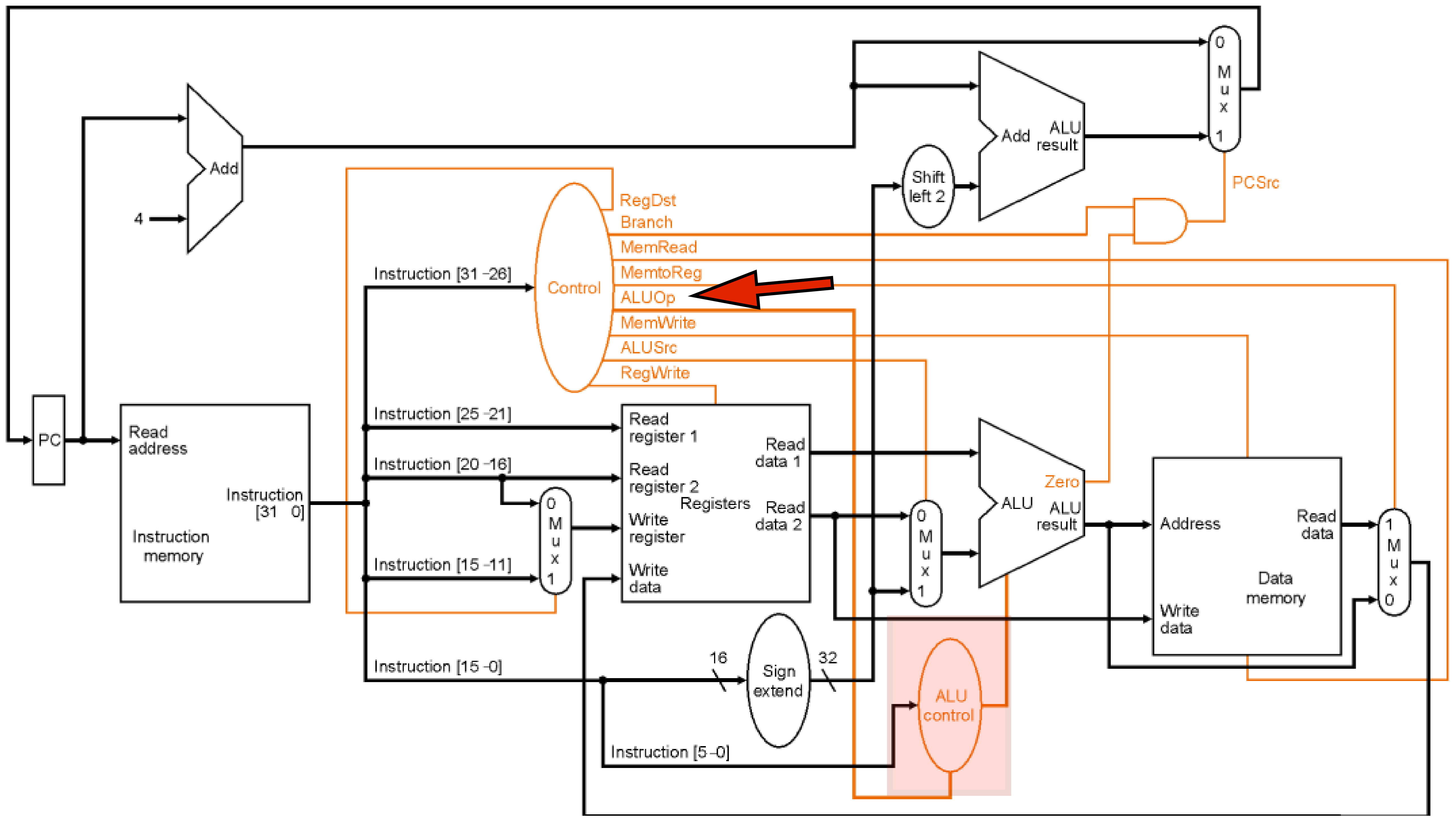
Putting it All Together: A Single Cycle Datapath



Putting it All Together: A Single Cycle Datapath



Putting it All Together: A Single Cycle Datapath



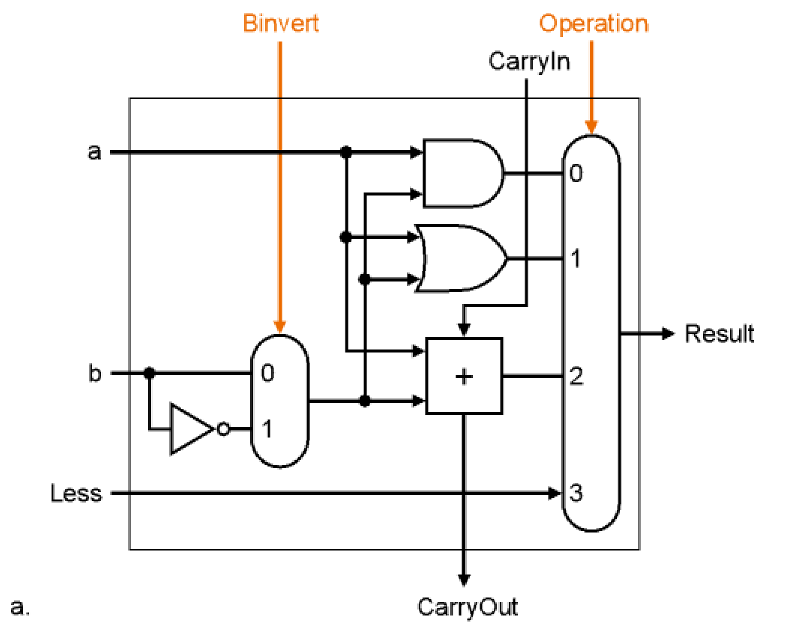
ALU Control Bits

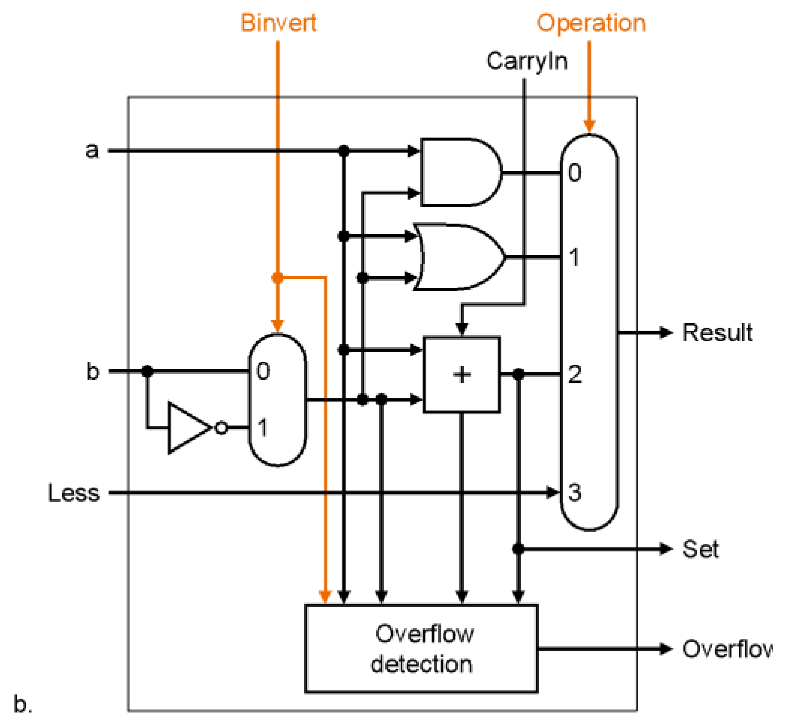
- 5-Function ALU

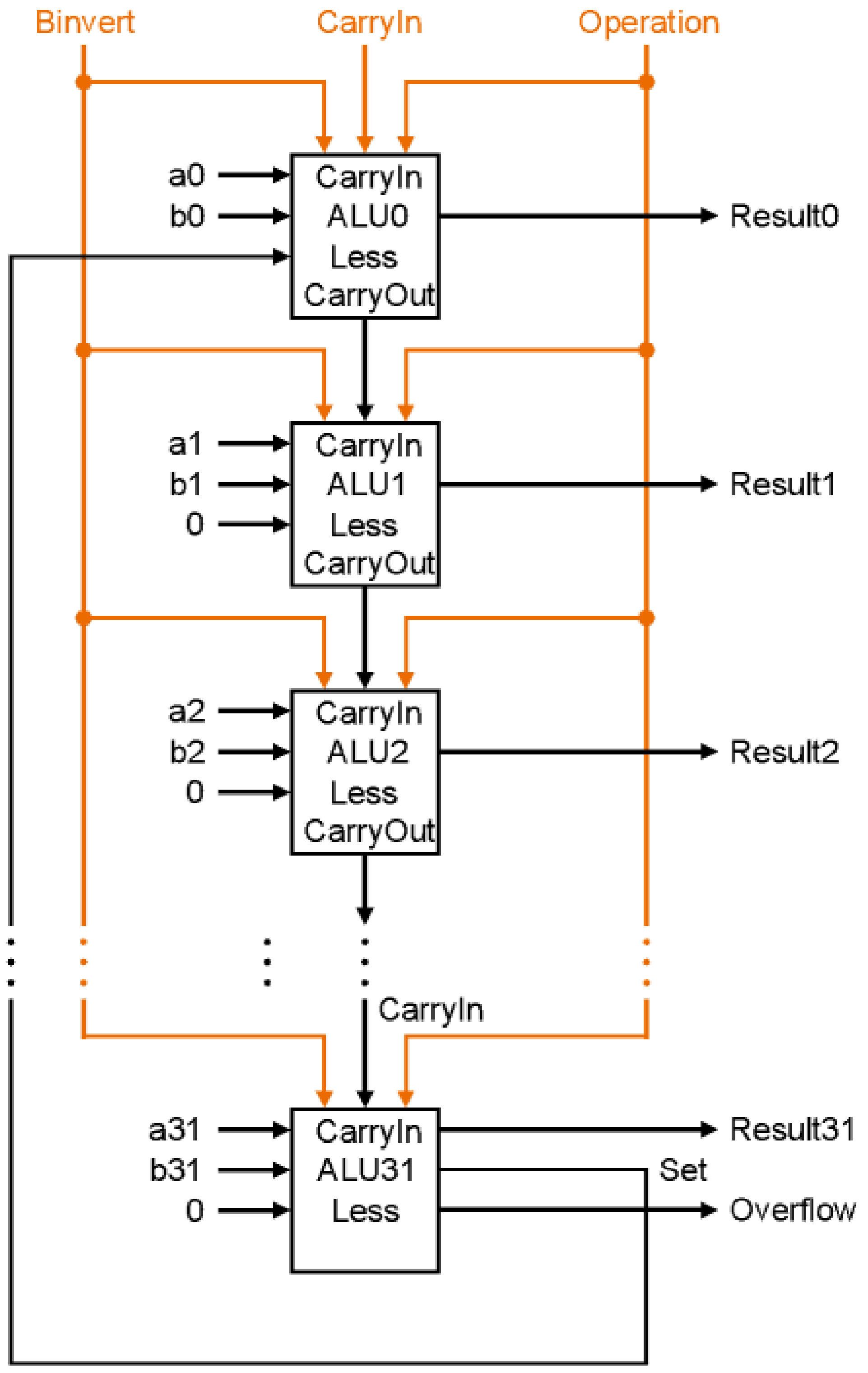
ALU control input	Function	Operations
000	And	and
001	Or	or
010	Add	add, lw, sw
110	Subtract	sub, beq
111	Slt	slt

- Note: book also has NOR, not used - and a forth bit, not used

Full ALU

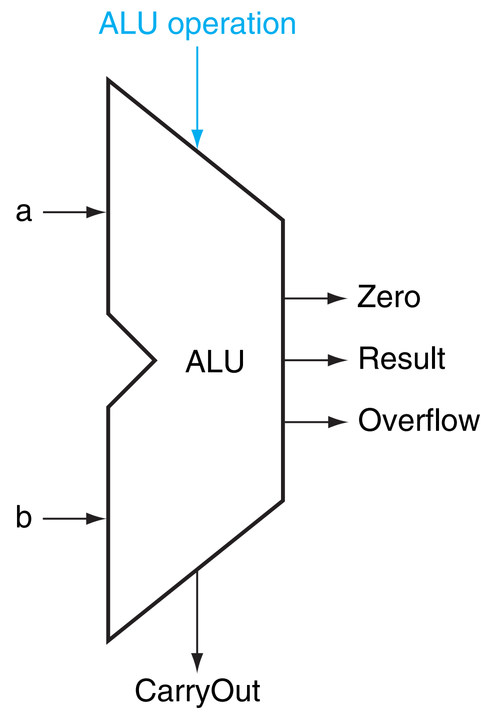




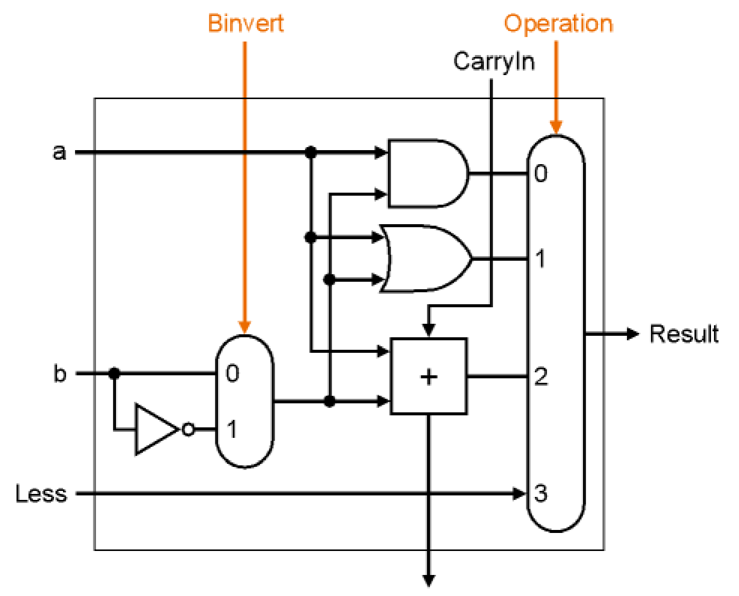


what signals accomplish:

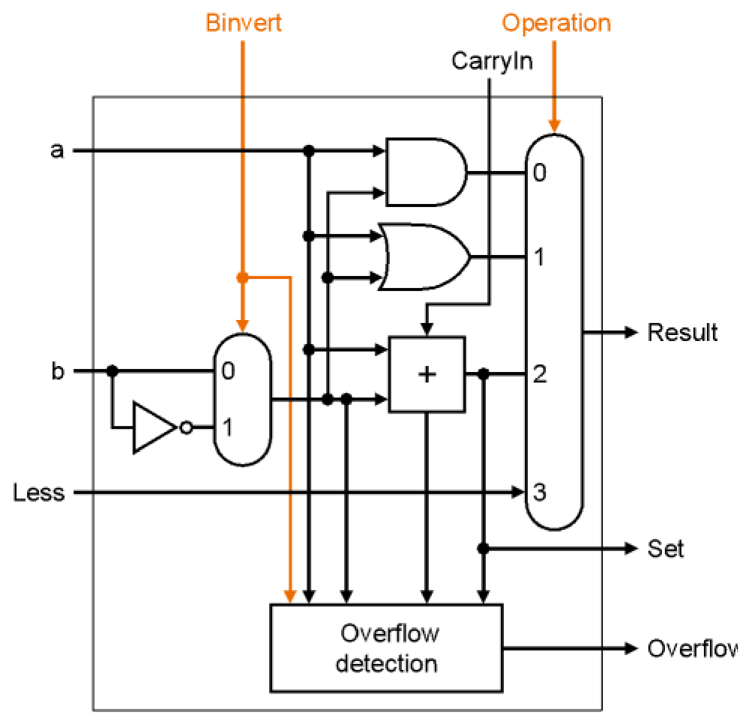
	Binvert	CIn	Oper
add?			
sub?			
and?			
or?			
beq?			
slt?			



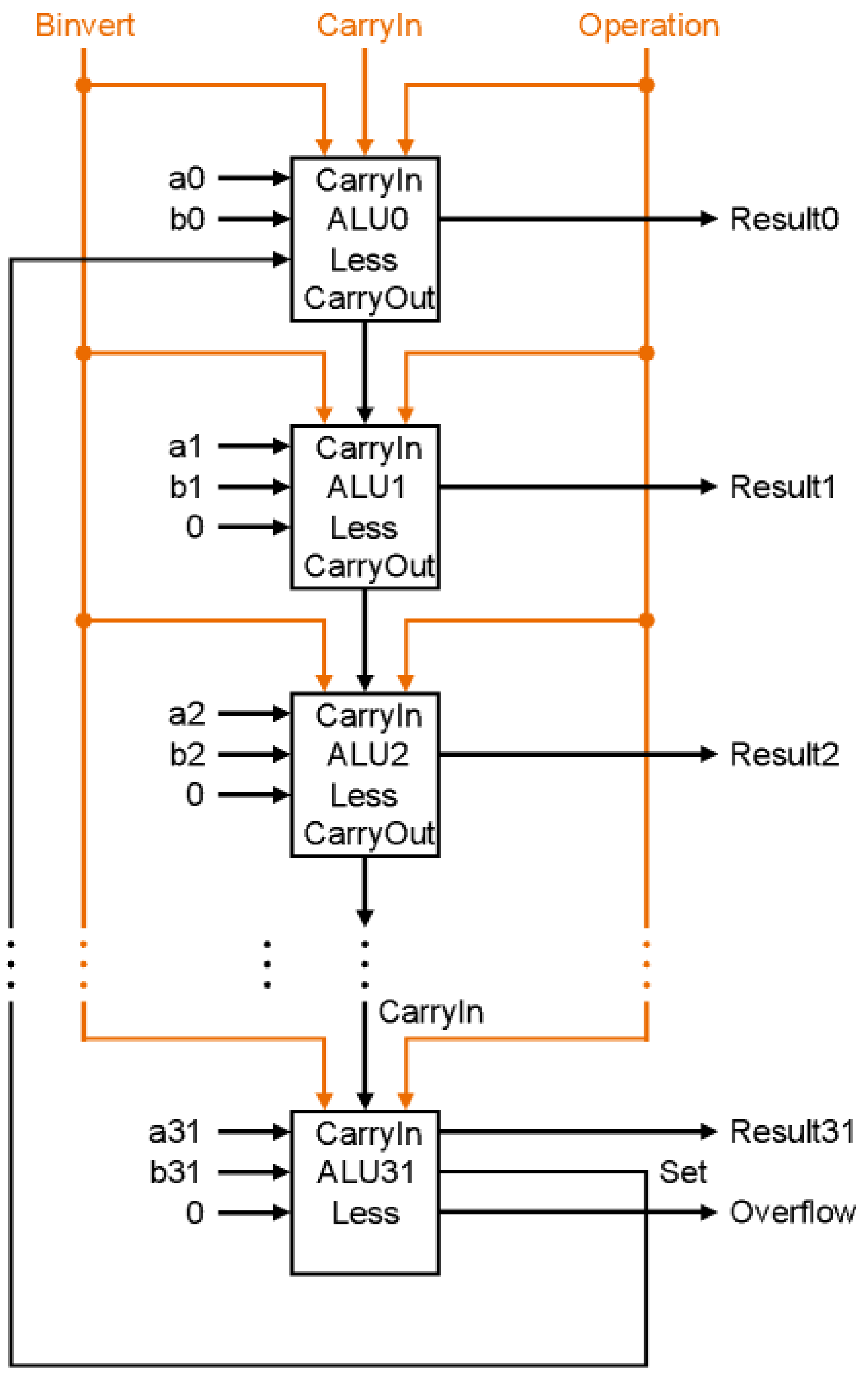
Full ALU



a.

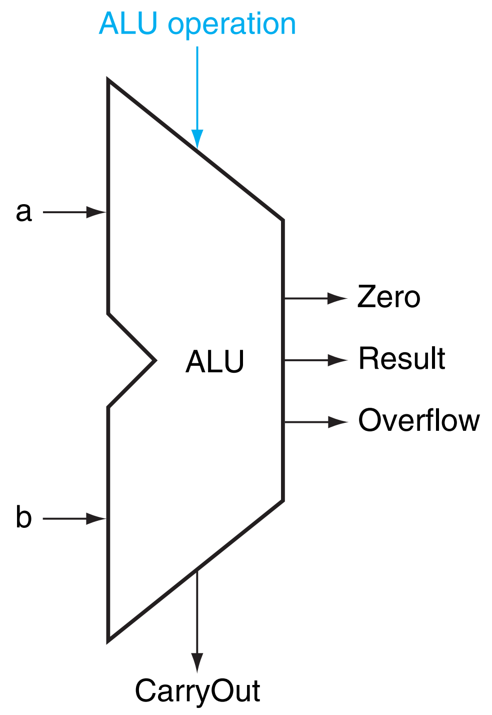


b.

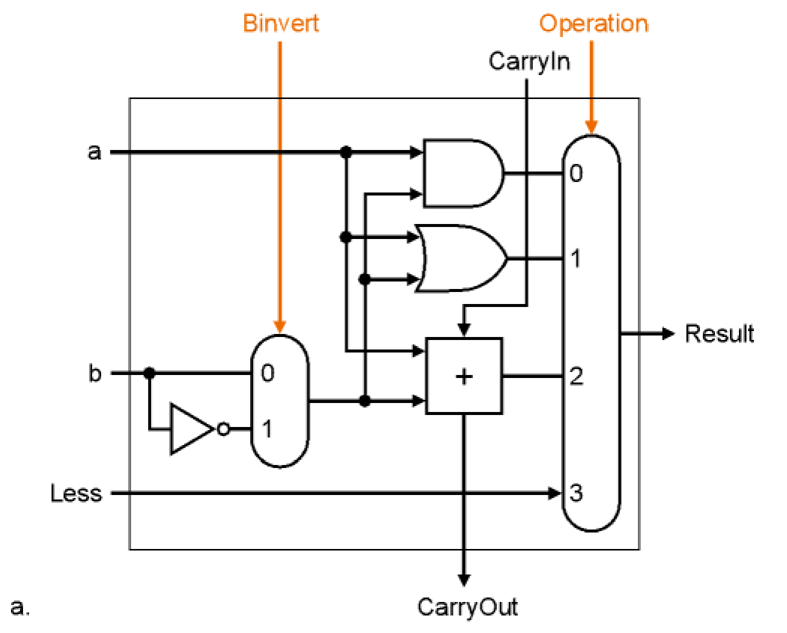


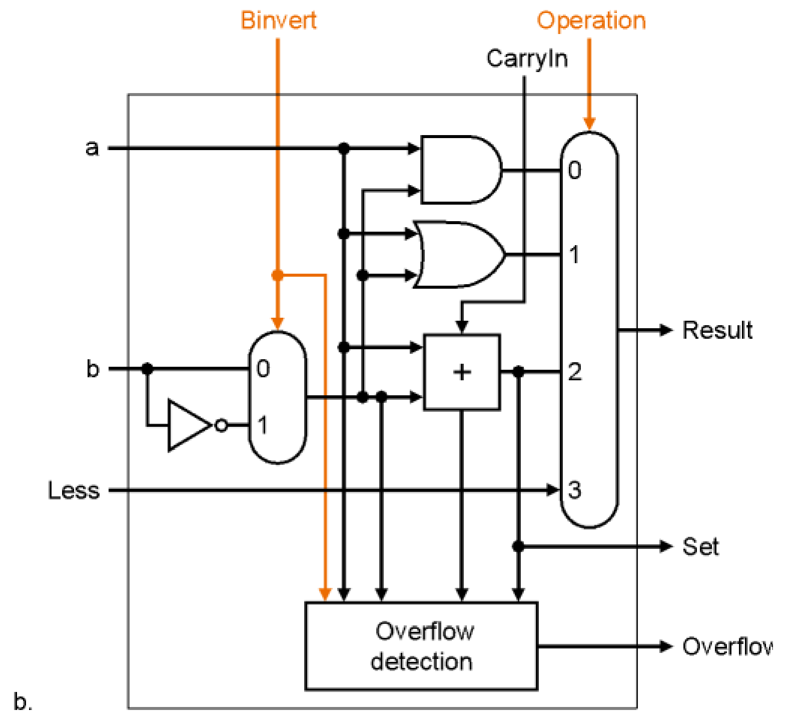
what signals accomplish:

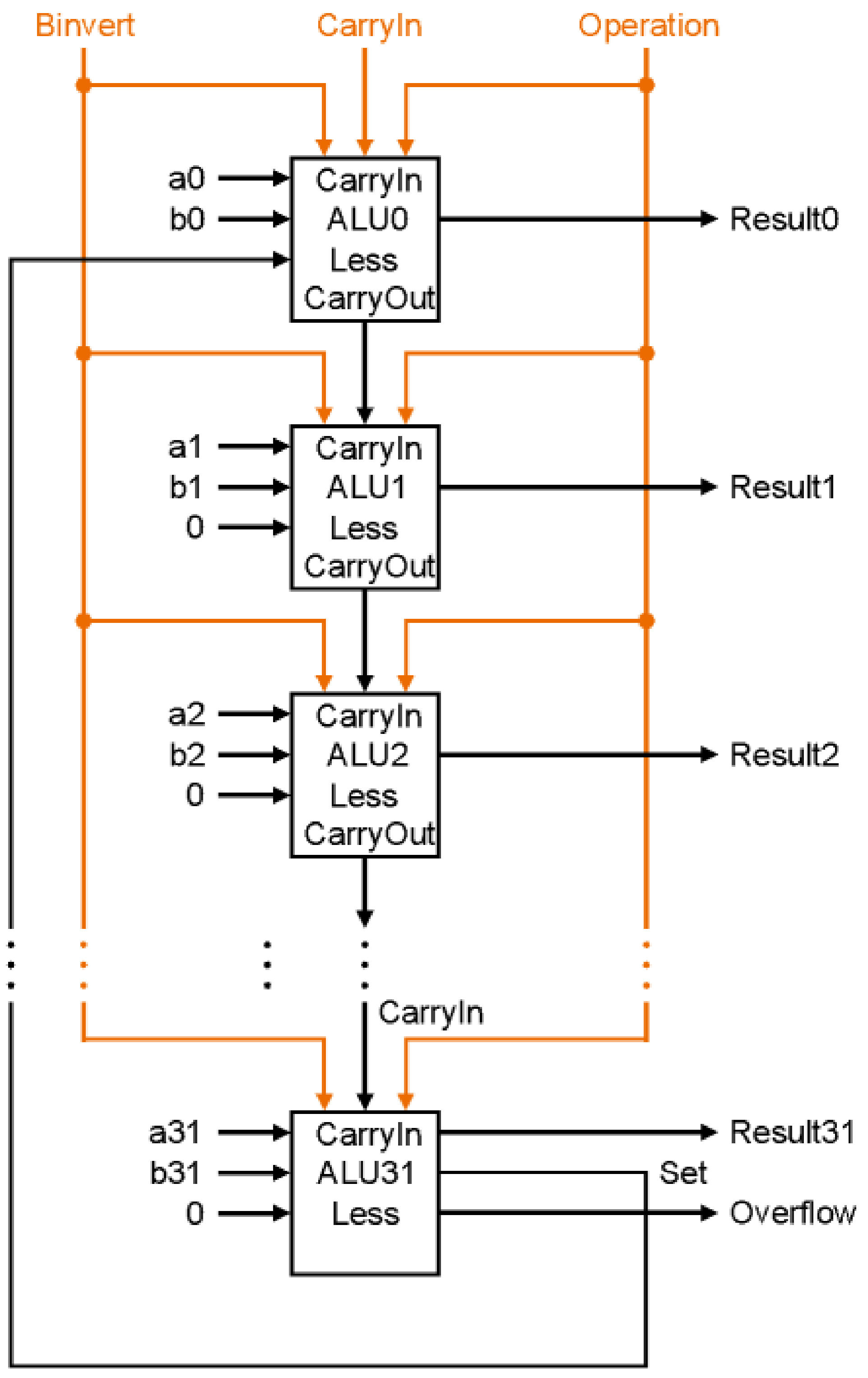
	Binvert	CIn	Oper
add?	0	0	10
sub?			
and?			
or?			
beq?			
slt?			



Full ALU

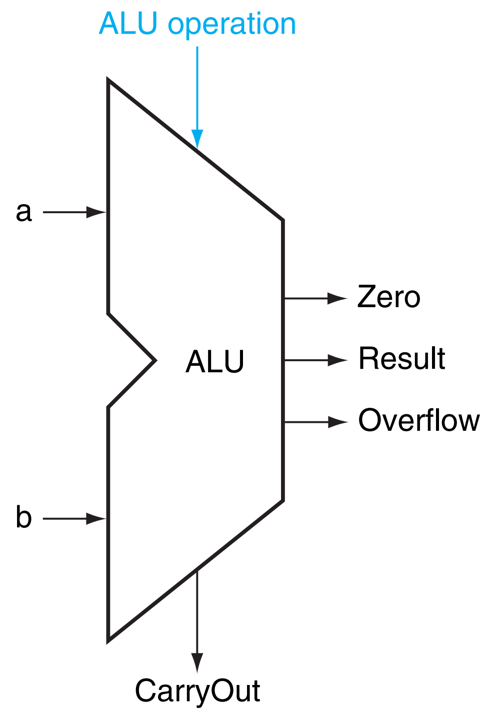




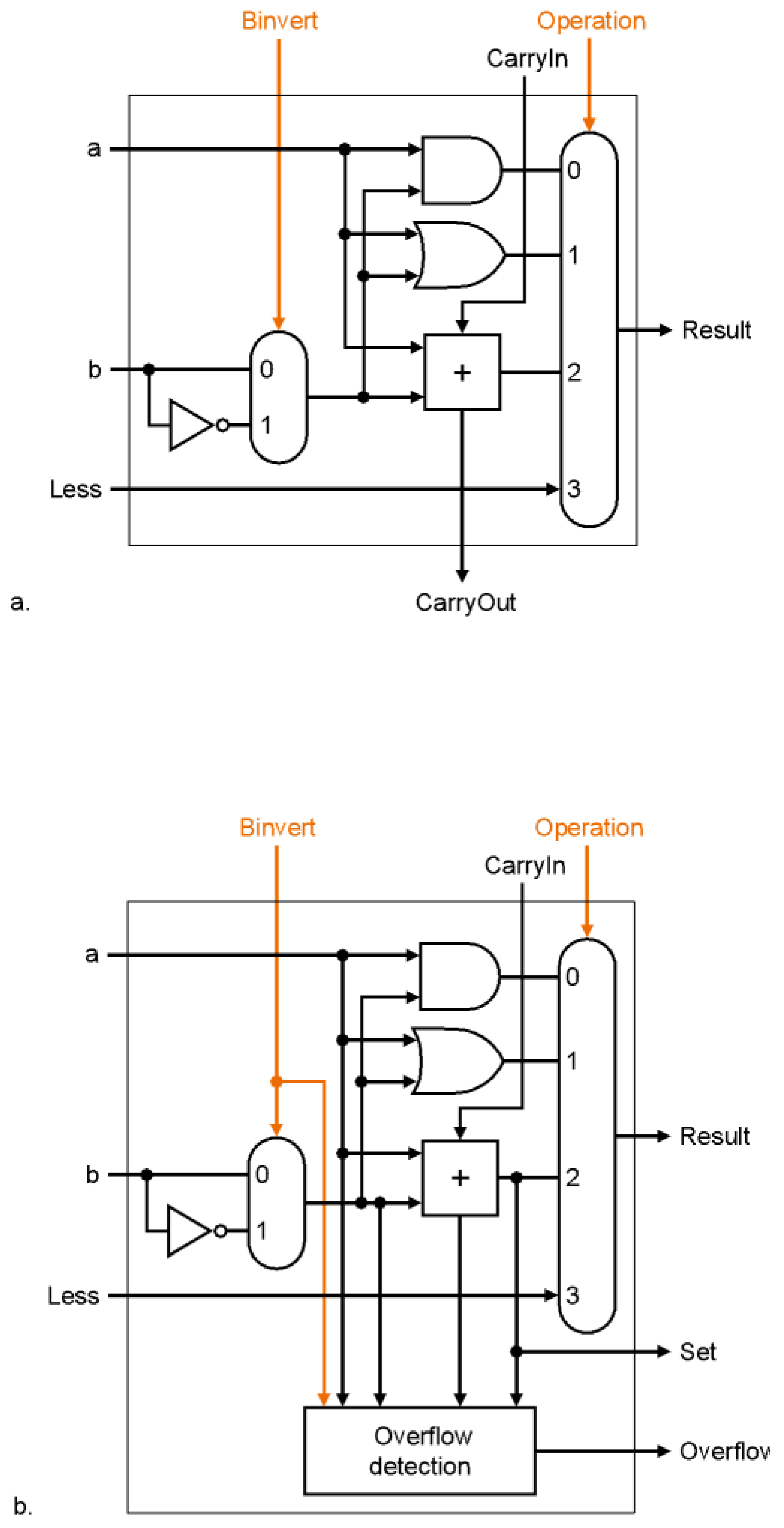


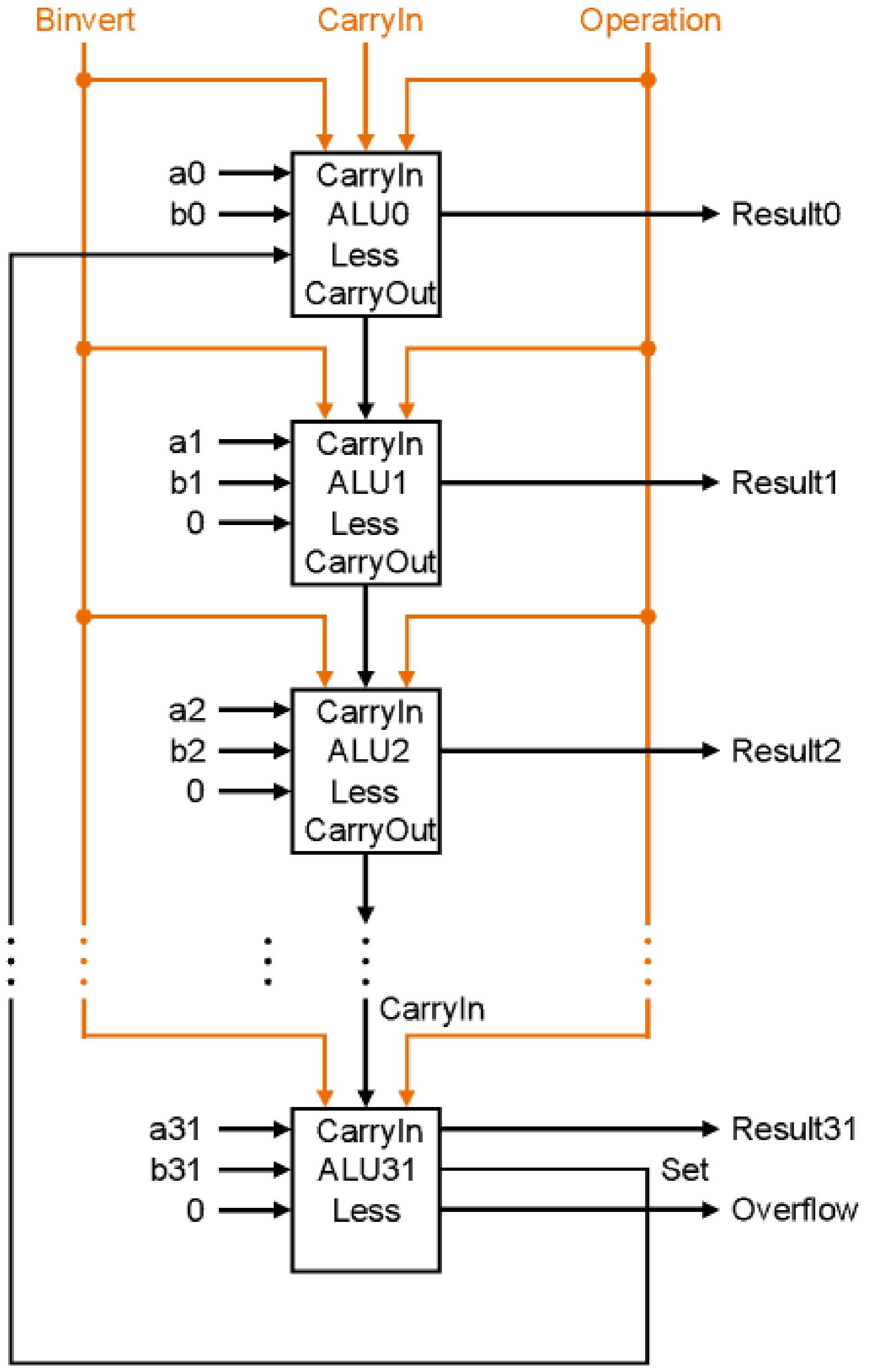
what signals accomplish:

	Binvert	CIn	Oper
add?	0	0	10
sub?	1	1	10
and?			
or?			
beq?			
slt?			



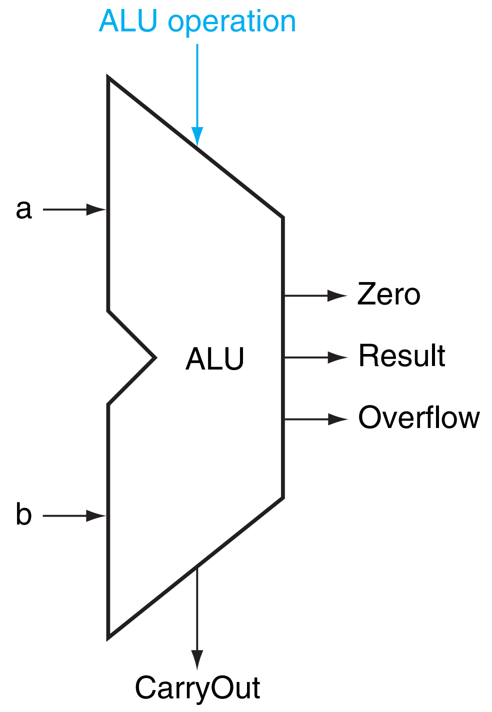
Full ALU



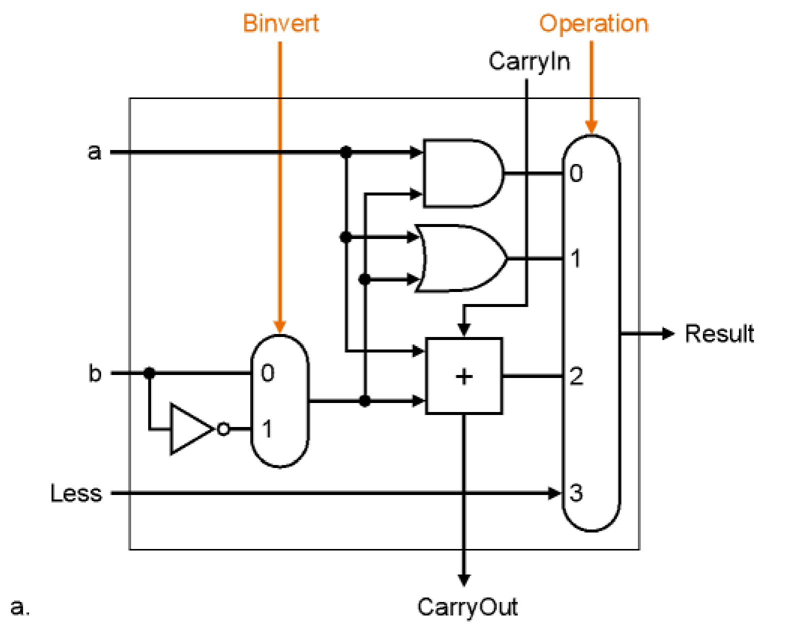


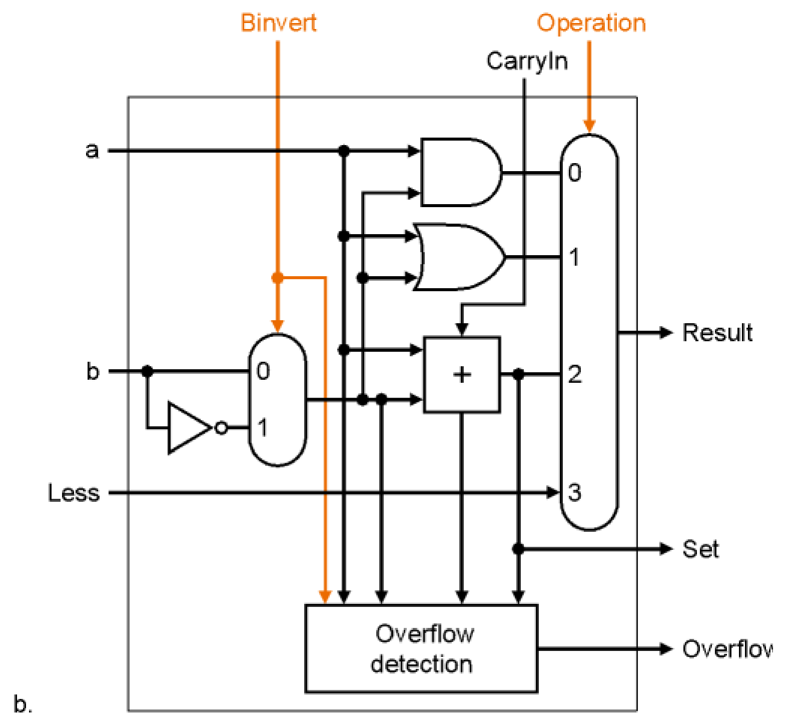
what signals accomplish:

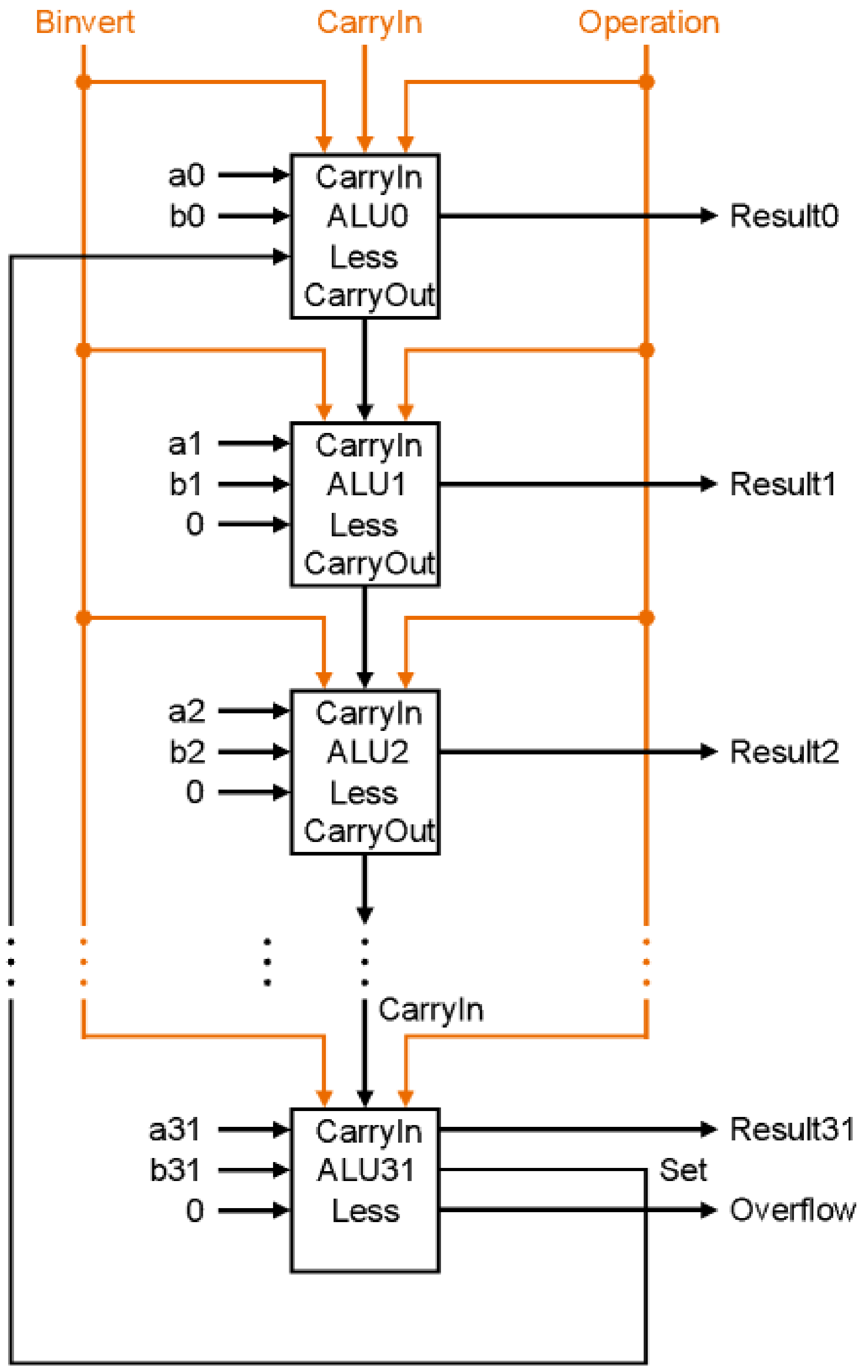
	Binvert	CIn	Oper
add?	0	0	10
sub?	1	1	10
and?	0	0	00
or?			
beq?			
slt?			



Full ALU

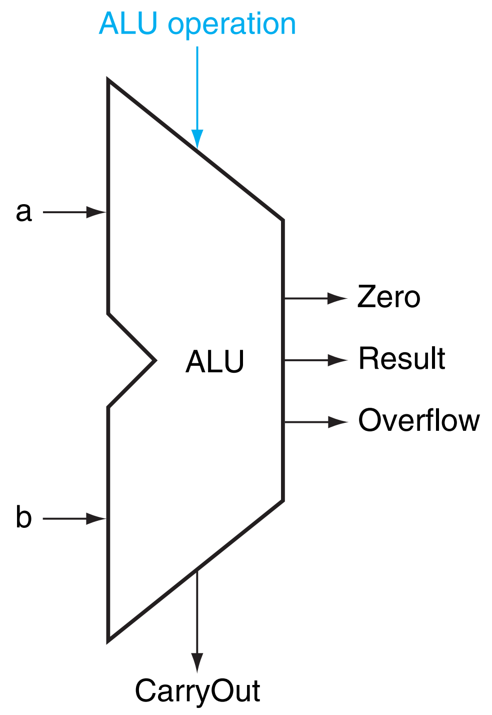




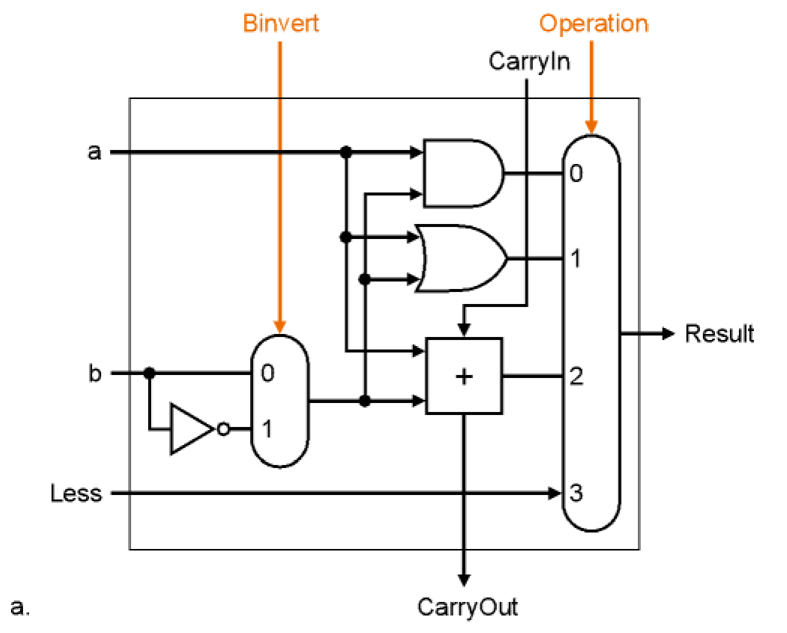


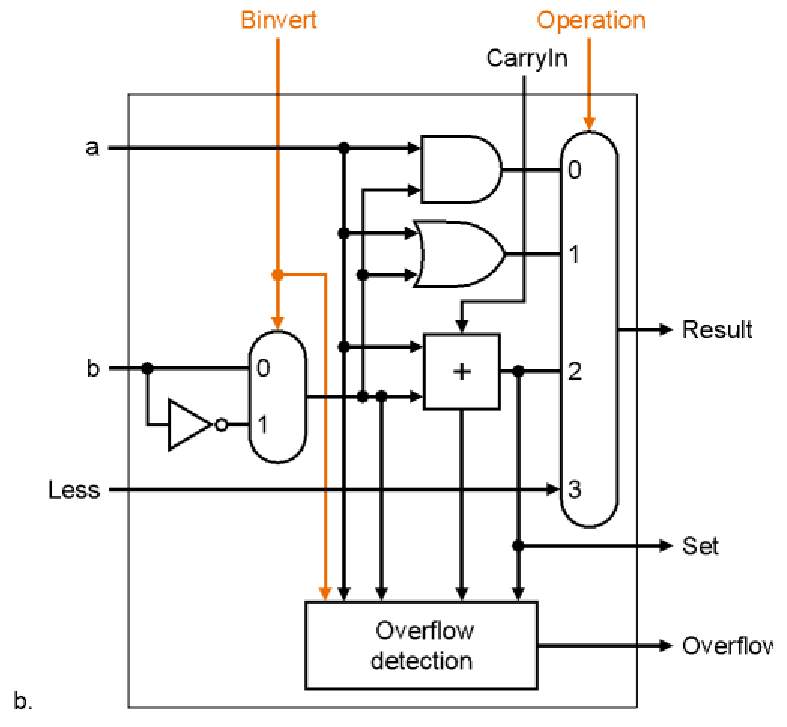
what signals accomplish:

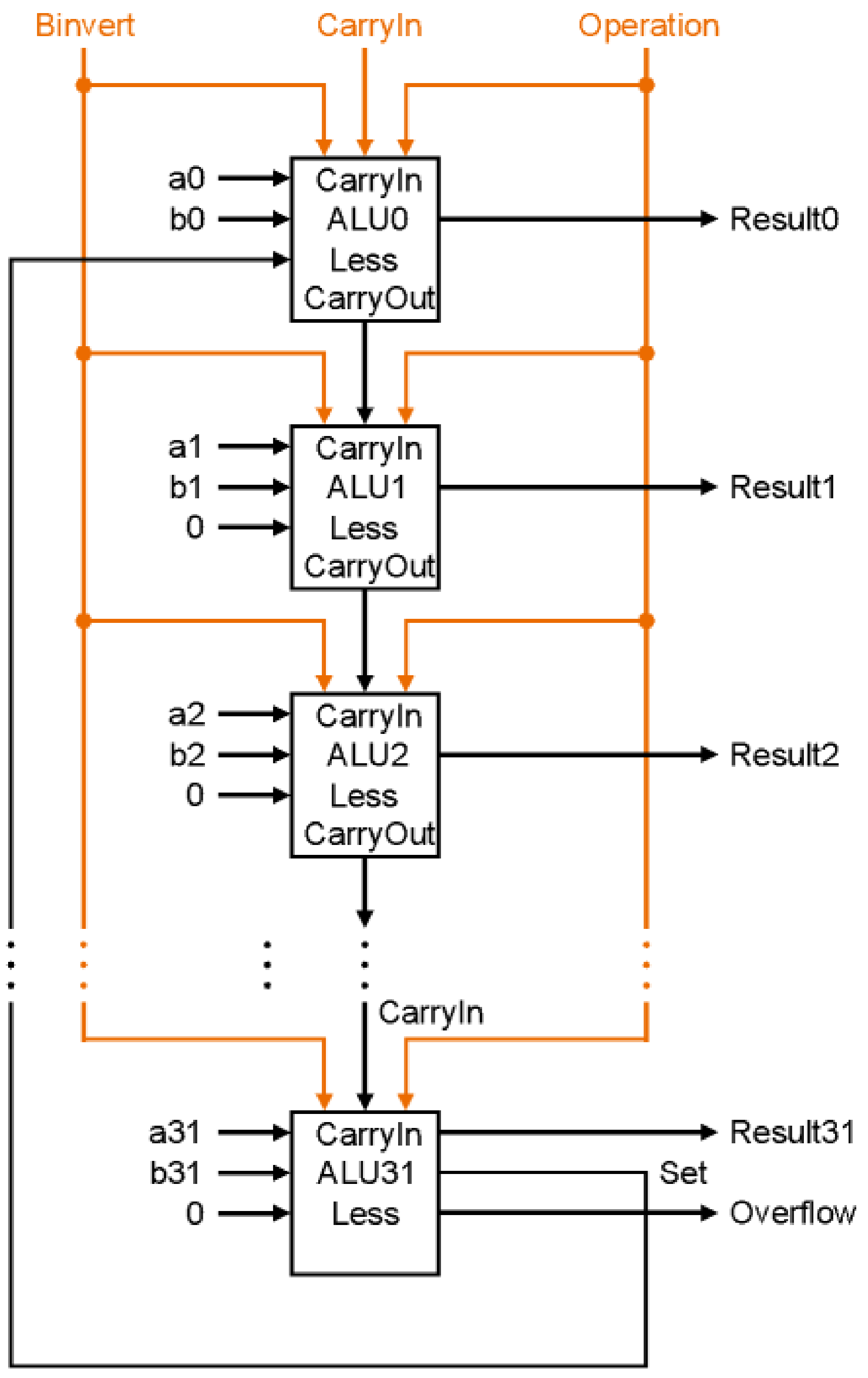
	Binvert	CIn	Oper
add?	0	0	10
sub?	1	1	10
and?	0	0	00
or?	0	0	01
beq?			
slt?			



Full ALU

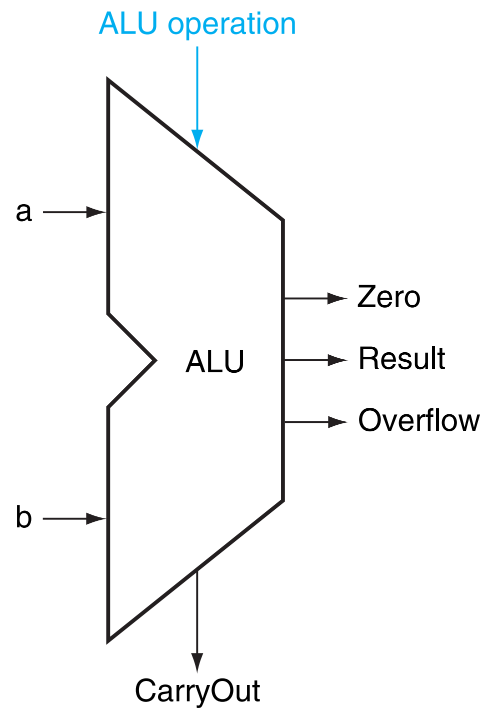




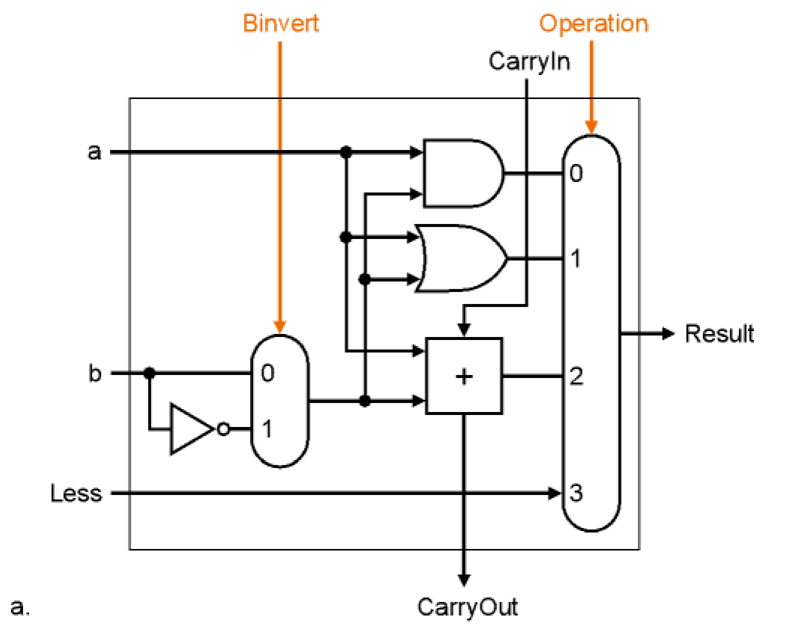


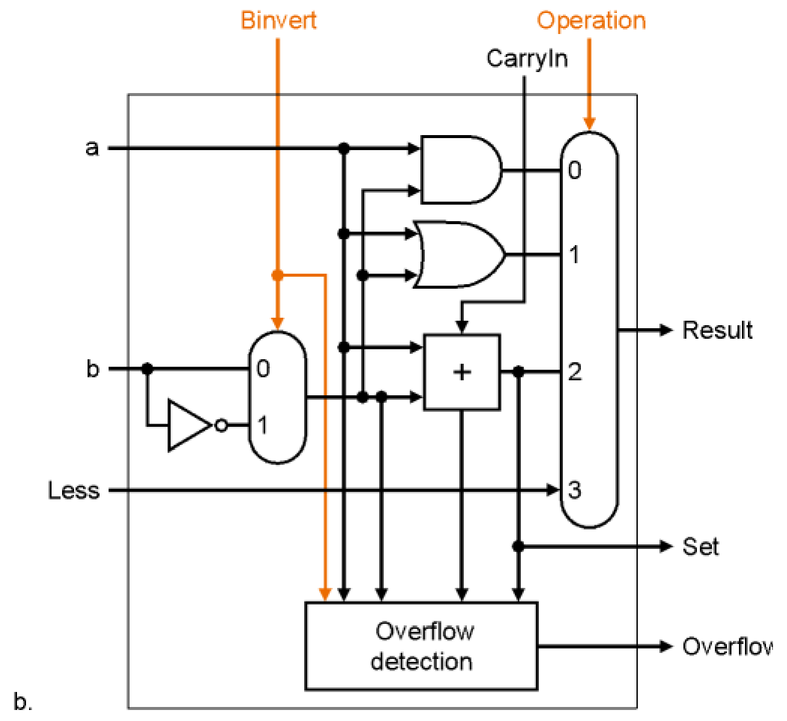
what signals accomplish:

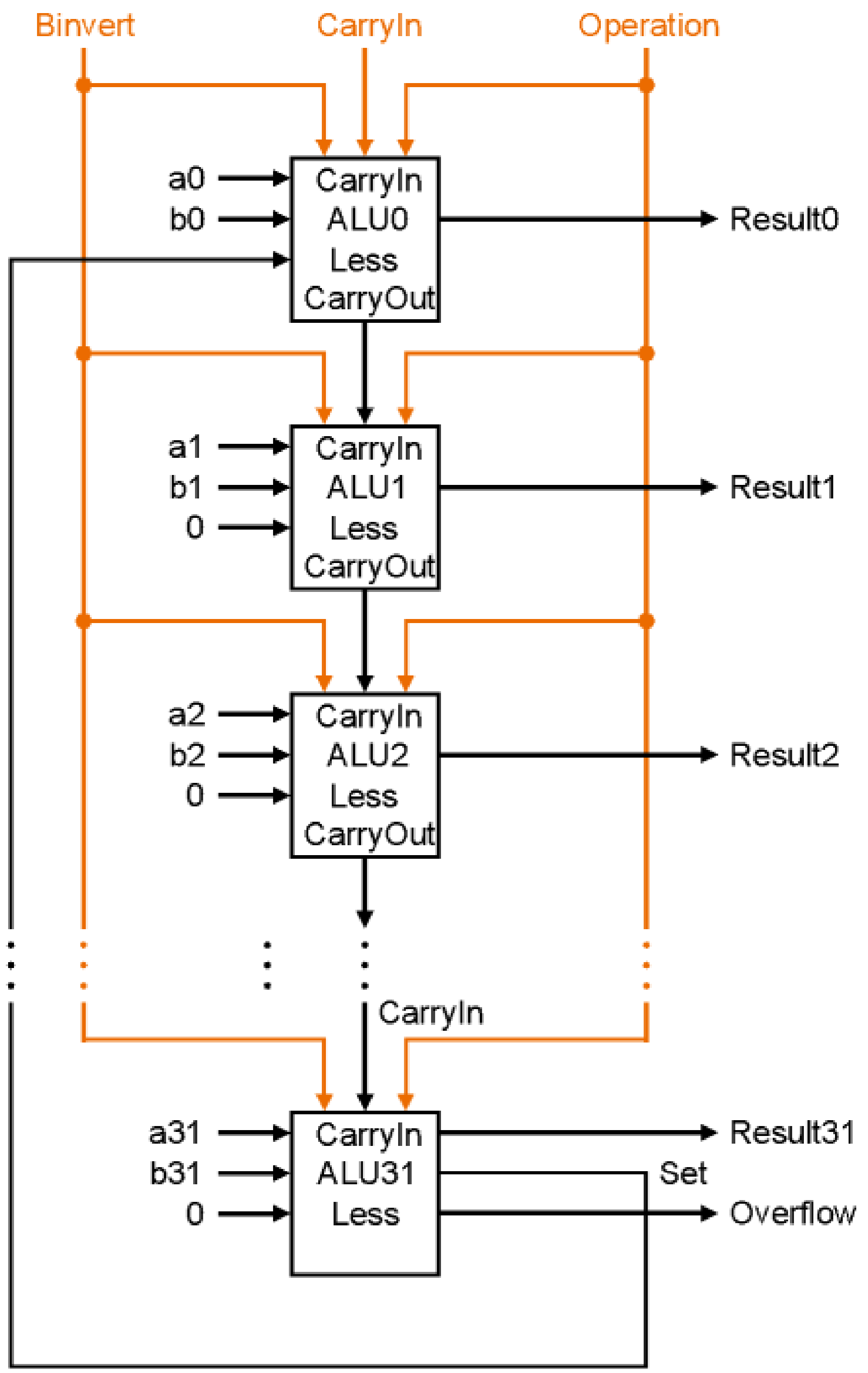
	Binvert	CIn	Oper
add?	0	0	10
sub?	1	1	10
and?	0	0	00
or?	0	0	01
beq?	1	1	10
slt?			



Full ALU

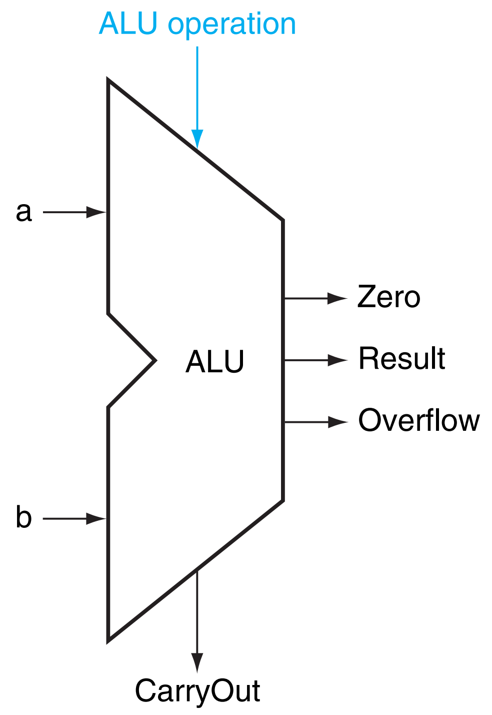






what signals accomplish:

	Binvert	CIn	Oper
add?	0	0	10
sub?	1	1	10
and?	0	0	00
or?	0	0	01
beq?	1	1	10
slt?	1	1	11



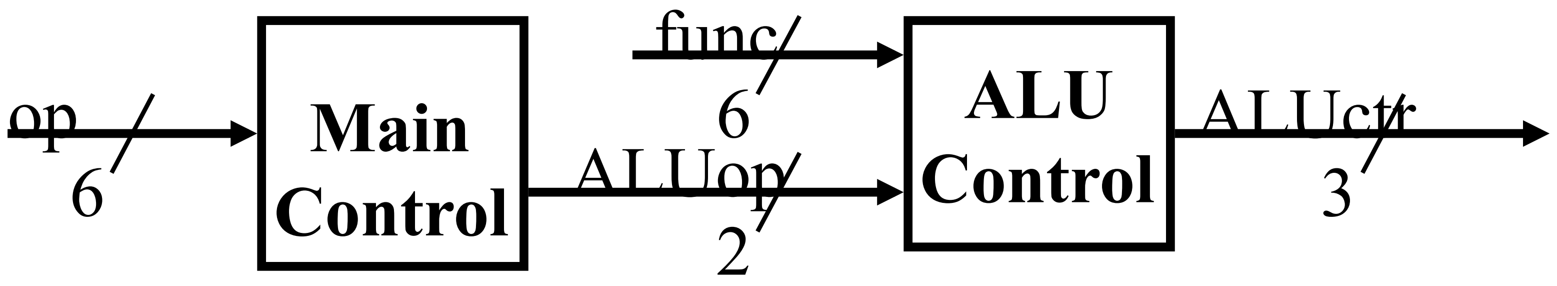
ALU Control Bits

- 5-Function ALU

ALU control input	Function	Operations
000	And	and
001	Or	or
010	Add	add, lw, sw
110	Subtract	sub, beq
111	Slt	slt

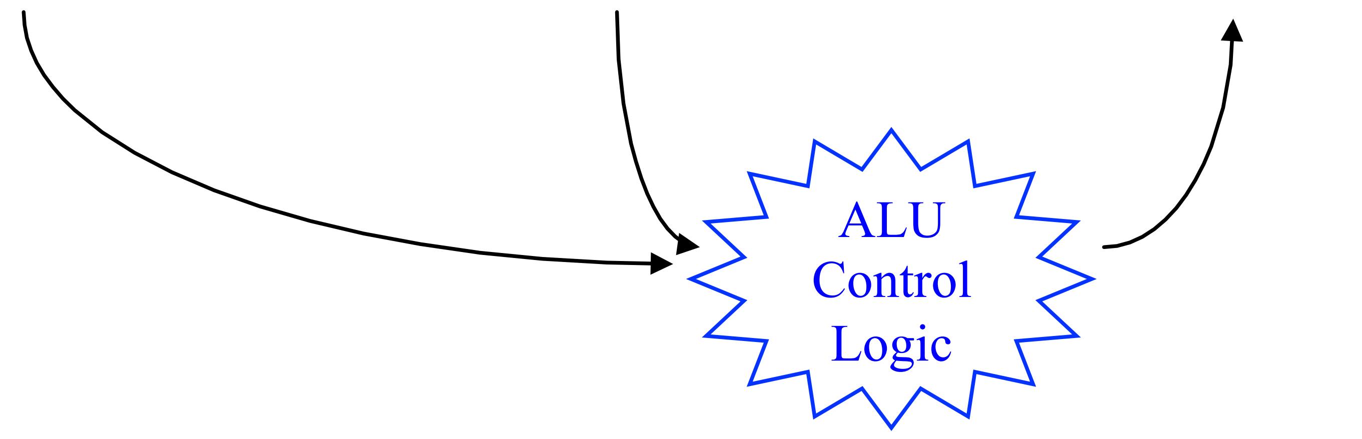
- Based on opcode (bits 31-26) and function code (bits 5-0) from instruction
- ALU doesn't need to know all opcodes--we will summarize opcode with ALUOp (2 bits):

00 - lw,sw   01 - beq   10 - R-format

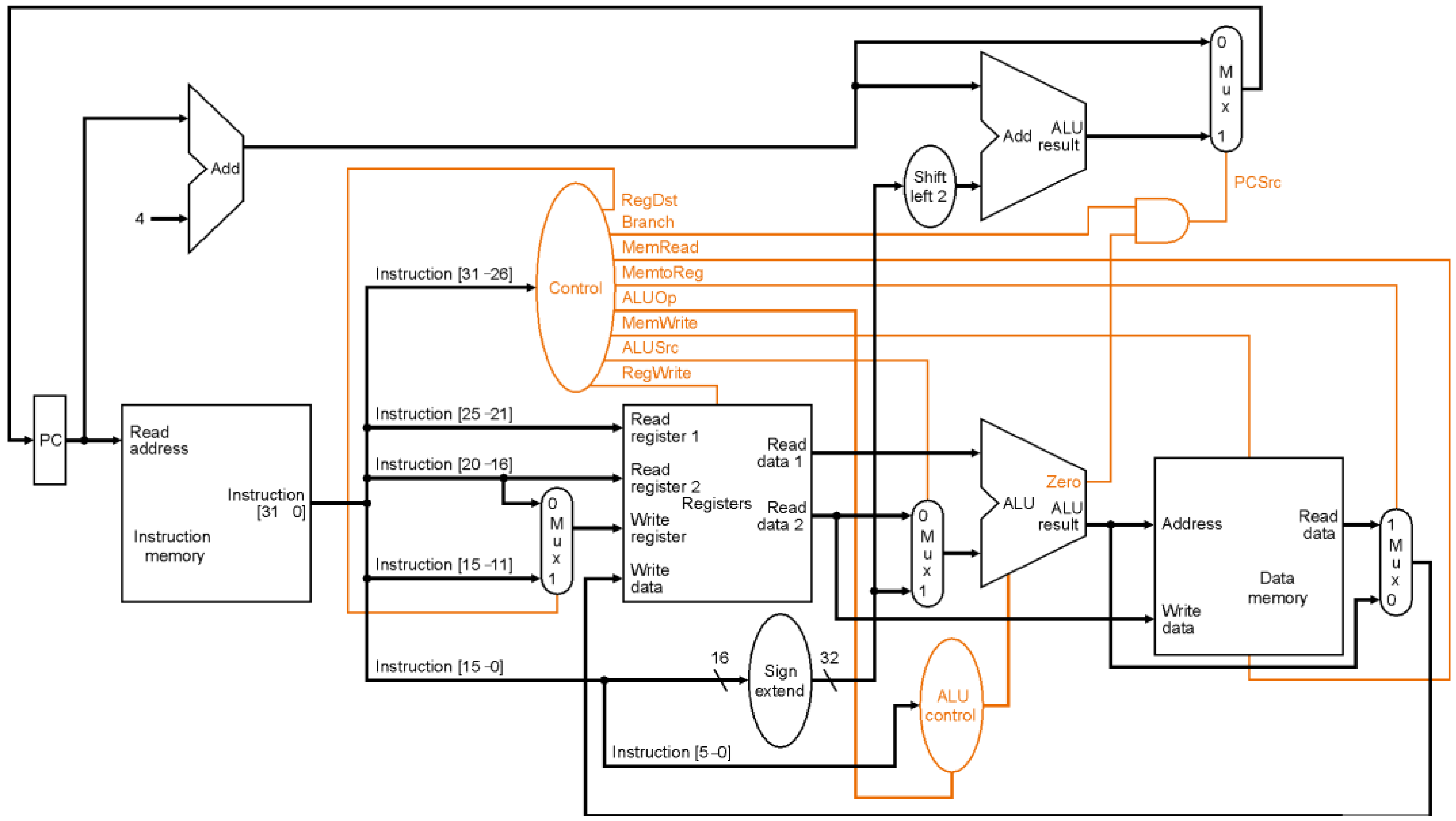


Generating ALU Control

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
lw	00	load word	xxxxxx	add	010
sw	00	store word	xxxxxx	add	010
beq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	slt	101010	slt	111

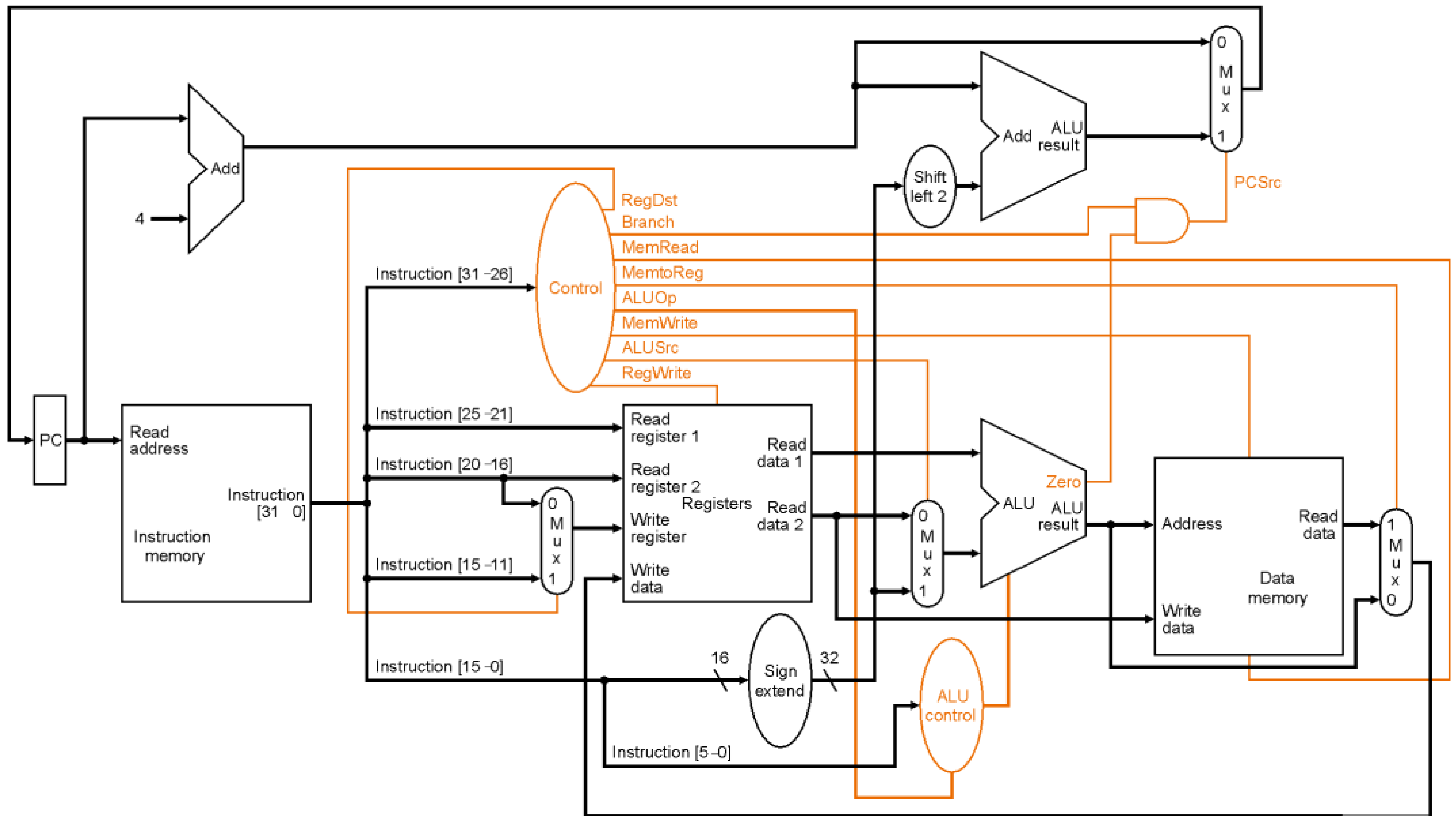


Controlling the CPU



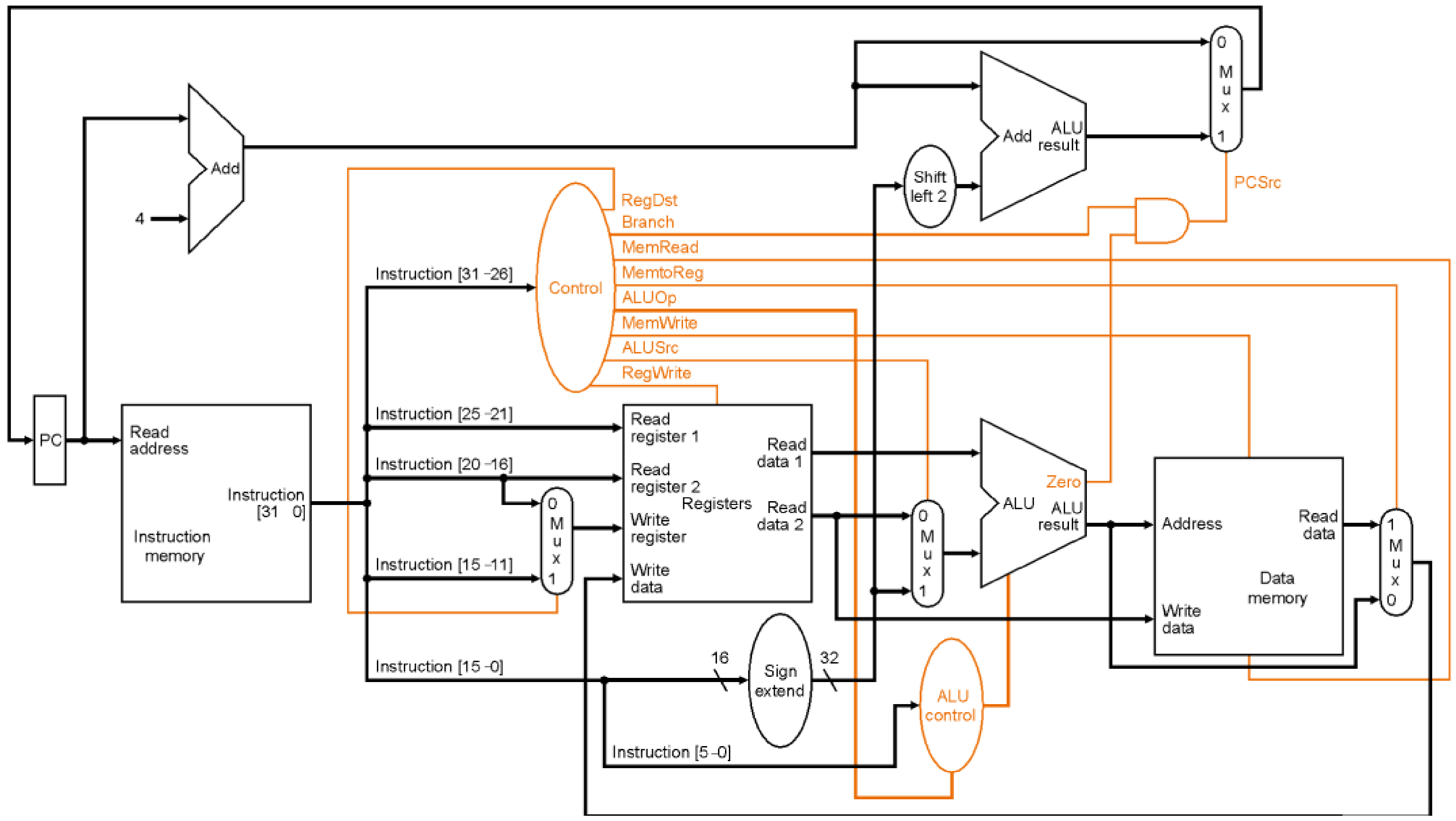
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format								1	0
`lw`								0	0
`sw`								0	0
`beq`								0	1

Controlling the CPU



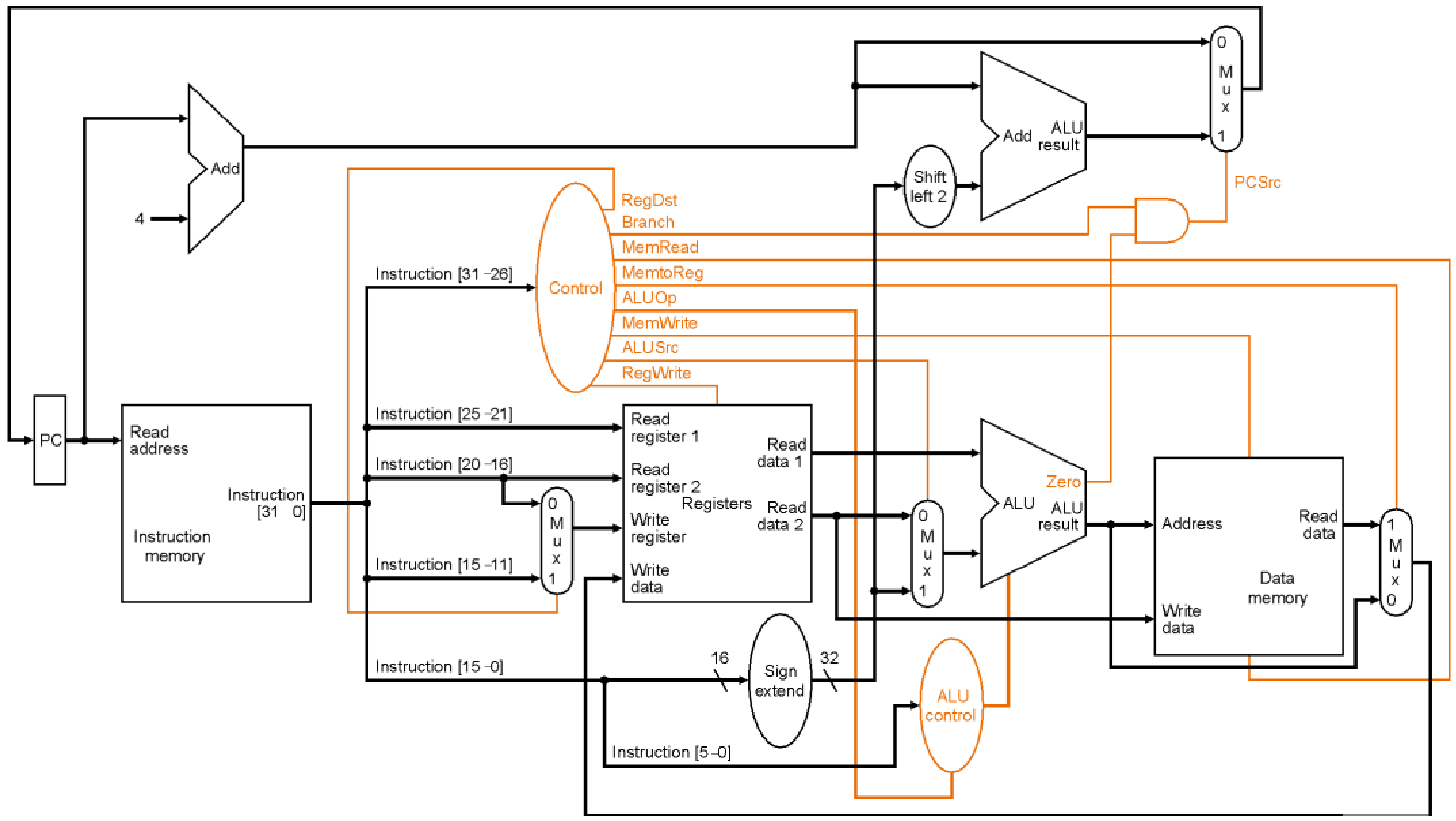
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1							1	0
lw								0	0
sw								0	0
beq								0	1

Controlling the CPU



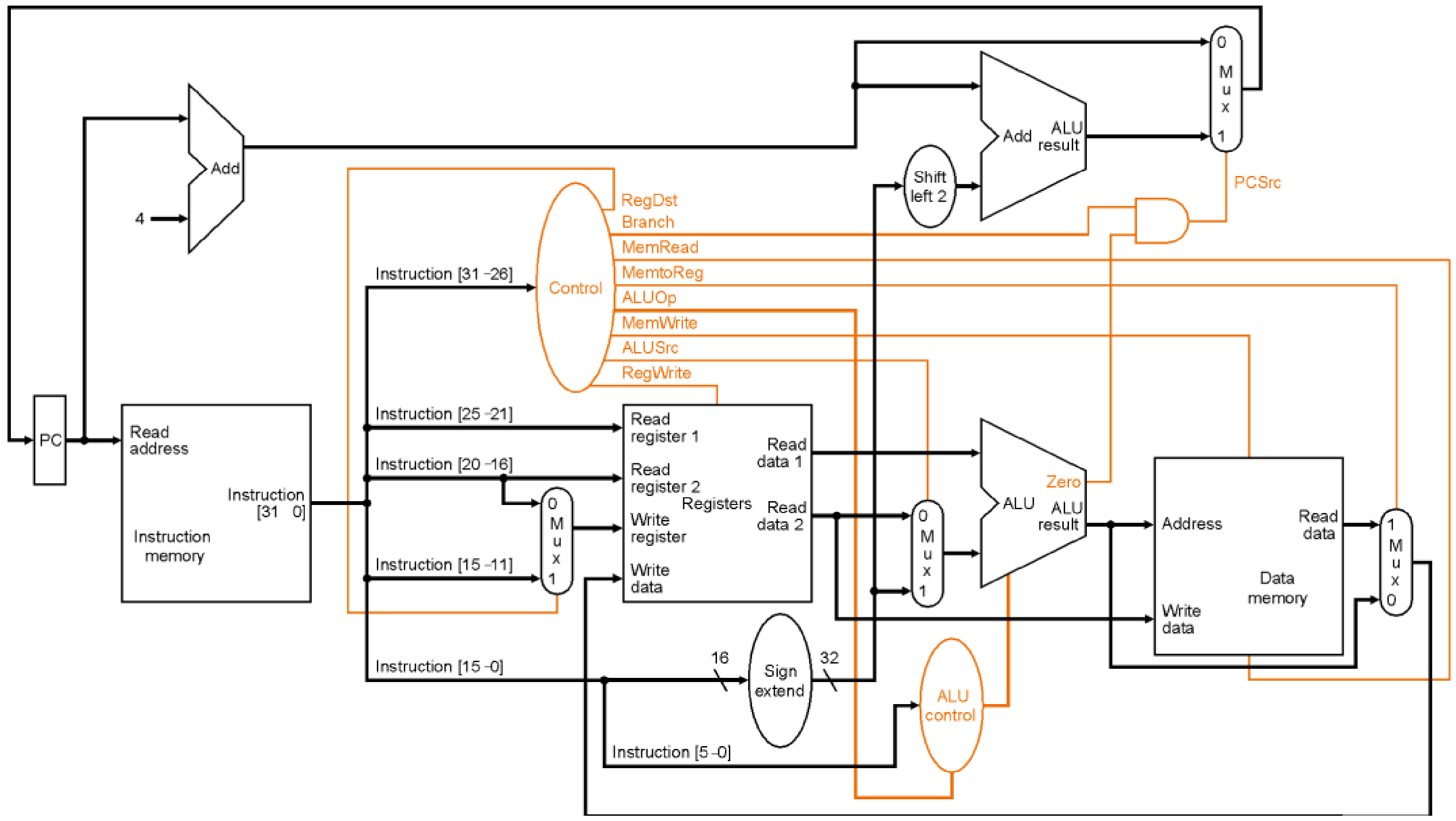
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0						1	0
`lw`								0	0
`sw`								0	0
`beq`								0	1

Controlling the CPU



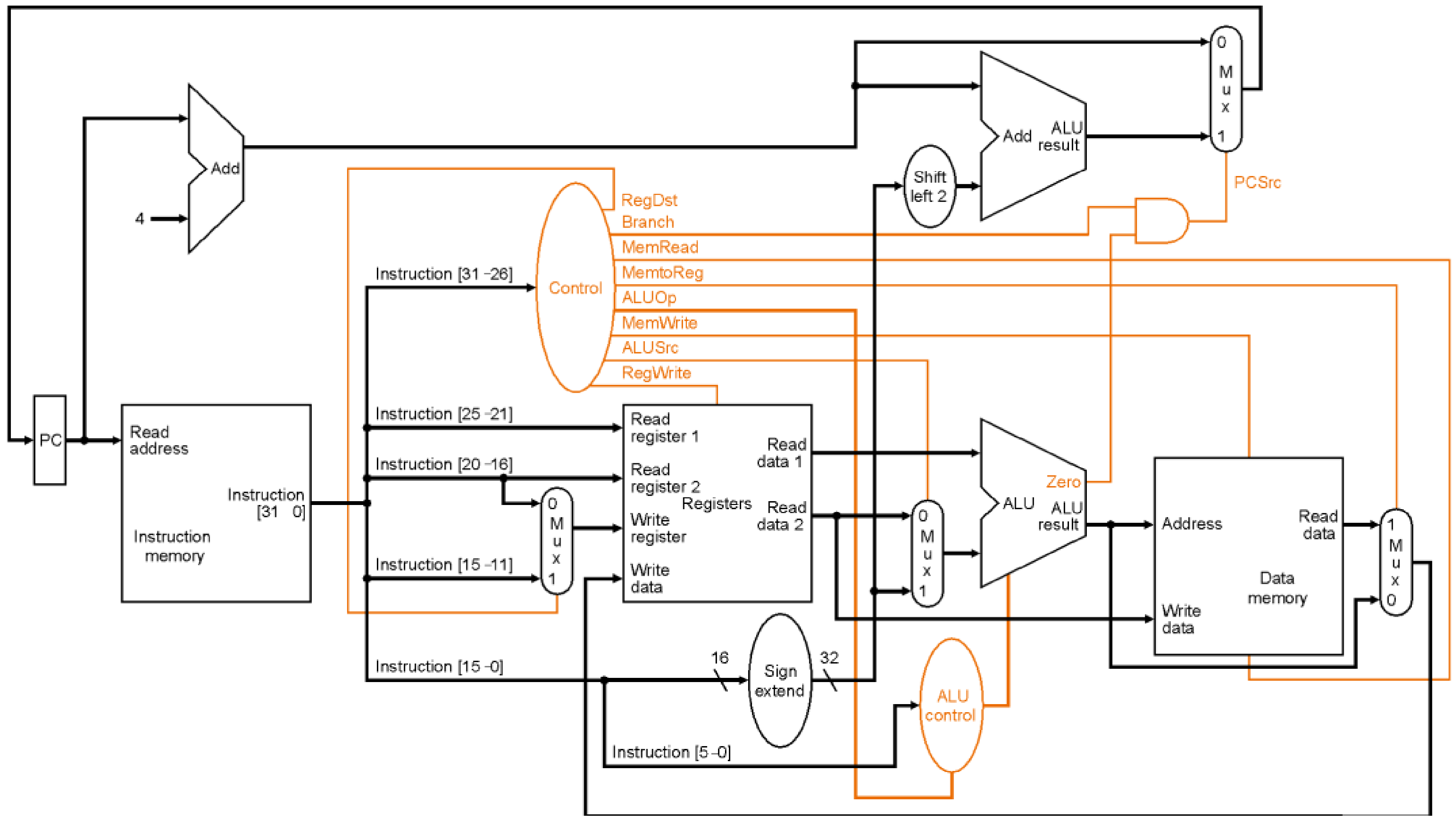
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0					1	0
lw								0	0
sw								0	0
beq								0	1

Controlling the CPU



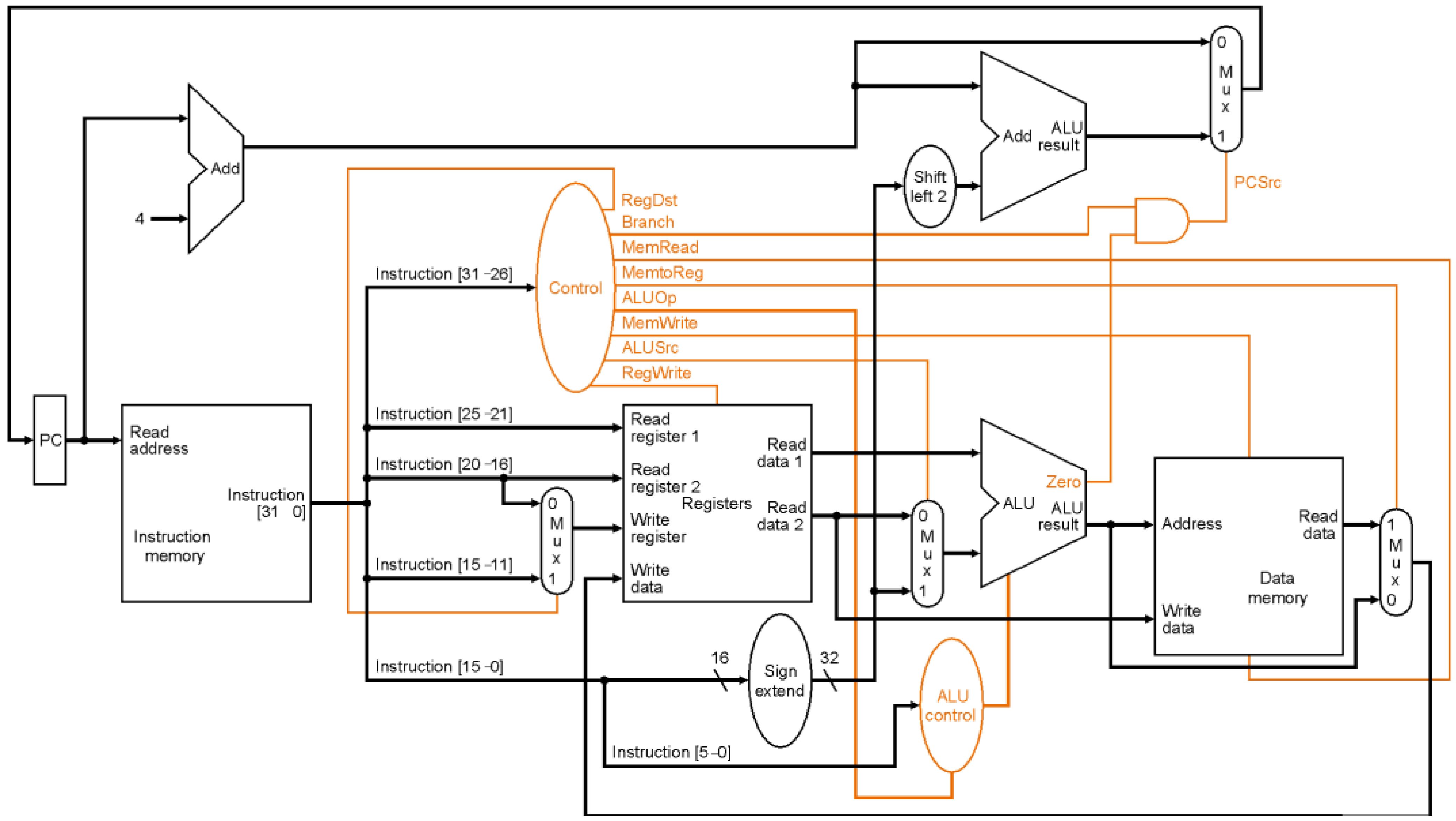
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1				1	0
lw								0	0
sw								0	0
beq								0	1

Controlling the CPU



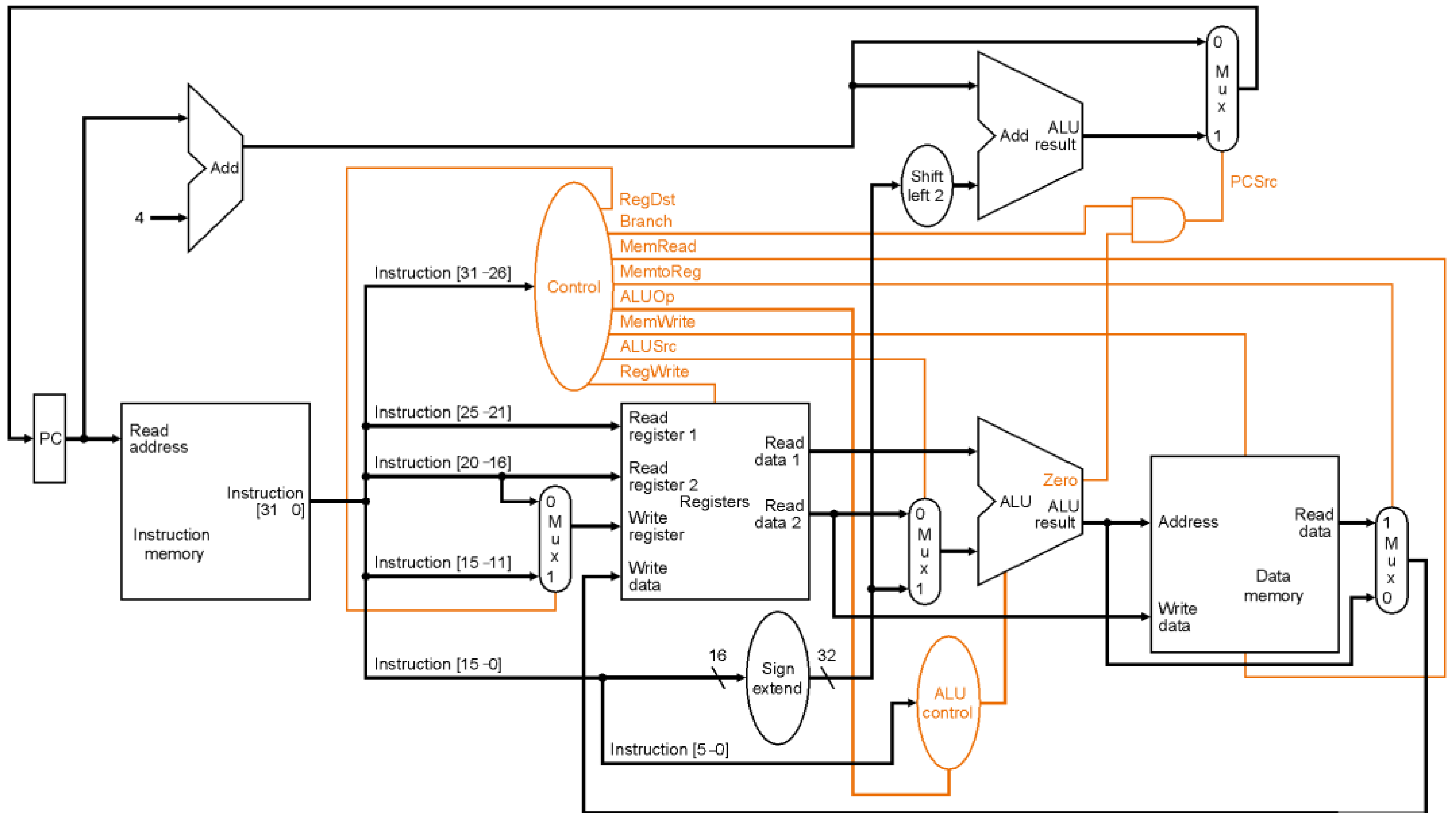
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0			1	0
lw								0	0
sw								0	0
beq								0	1

Controlling the CPU



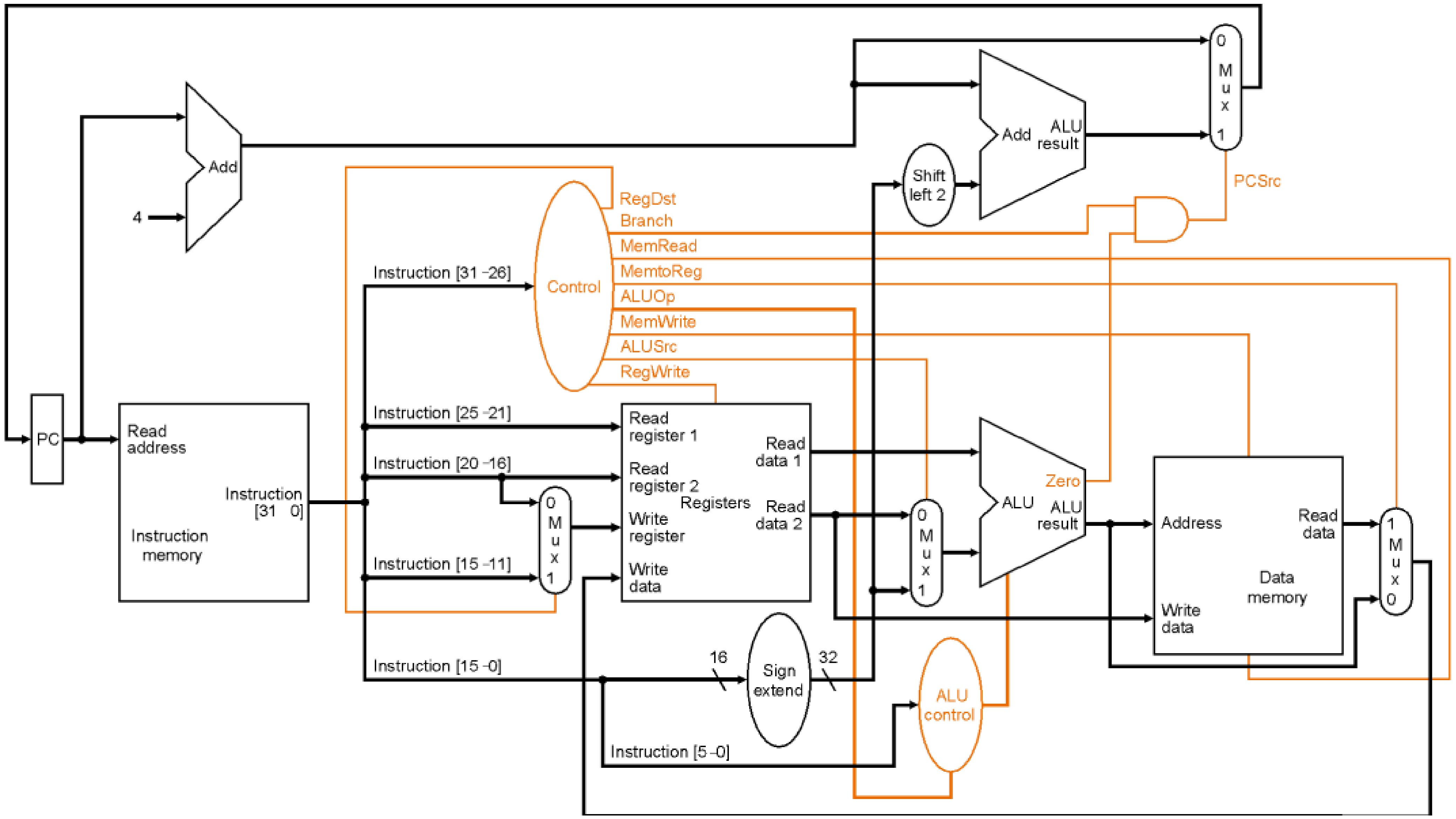
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0		1	0
lw								0	0
sw								0	0
beq								0	1

Controlling the CPU



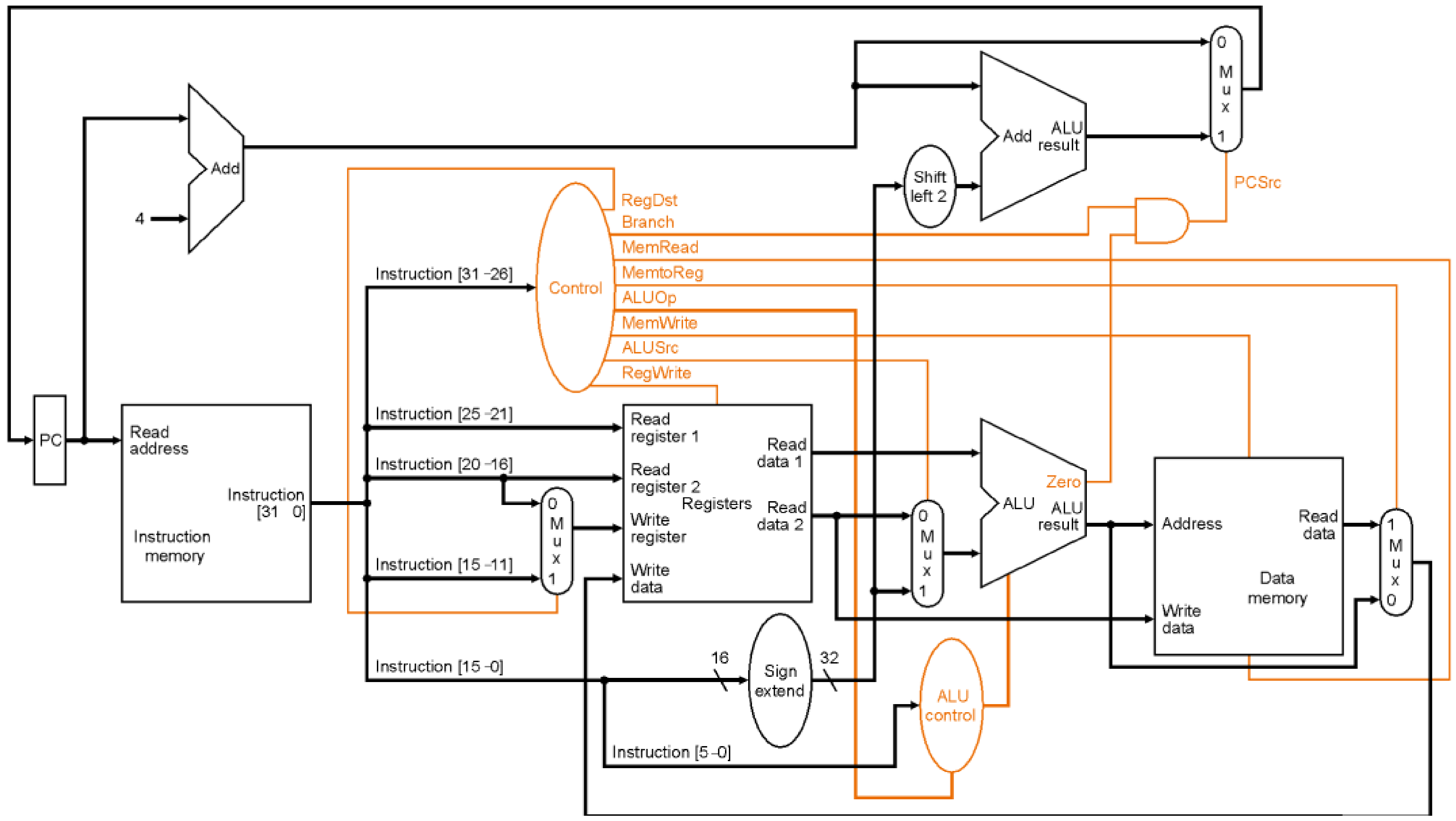
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`								0	0
`sw`								0	0
`beq`								0	1

Controlling the CPU



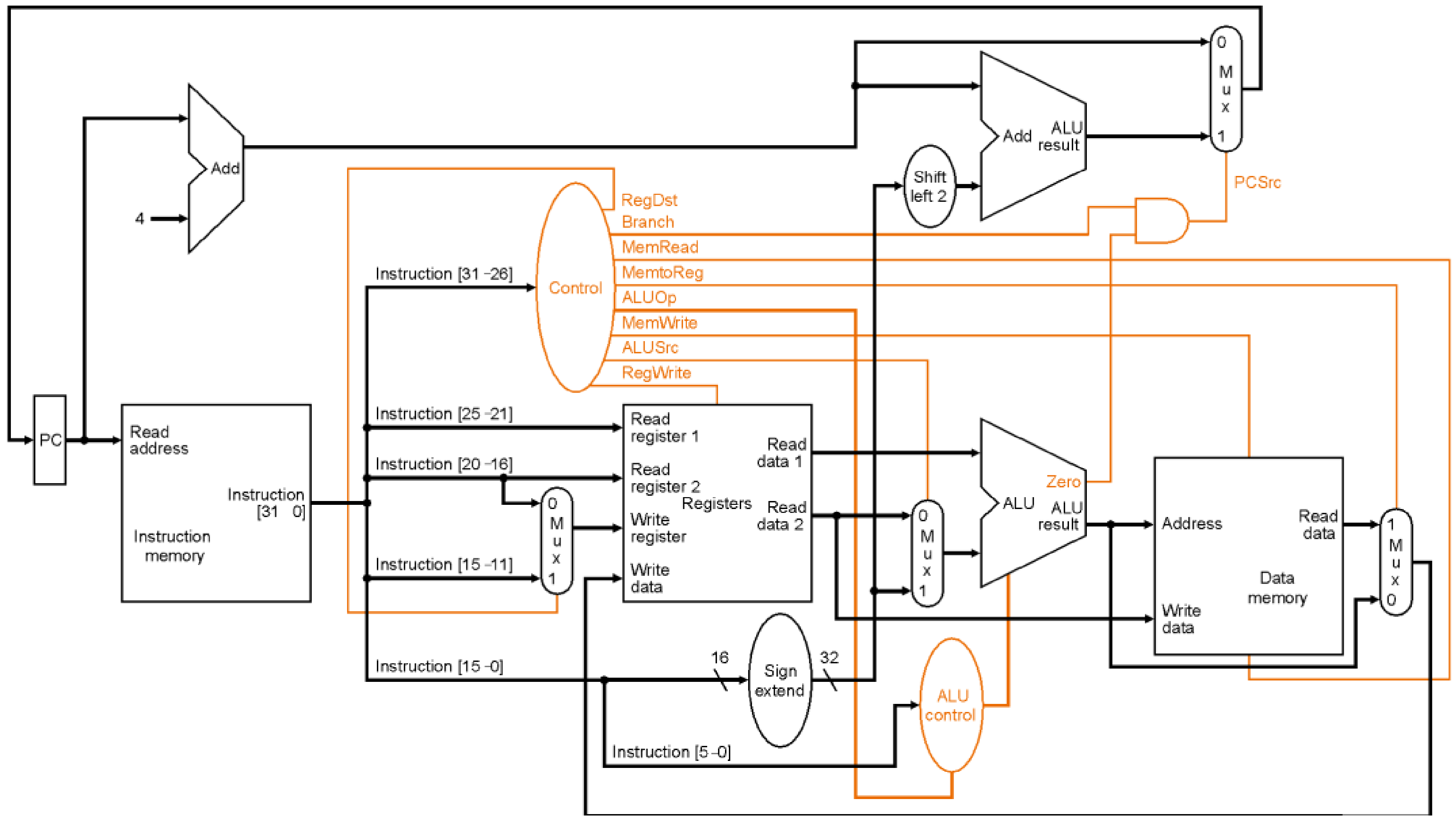
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0							0	0
sw								0	0
beq								0	1

Controlling the CPU



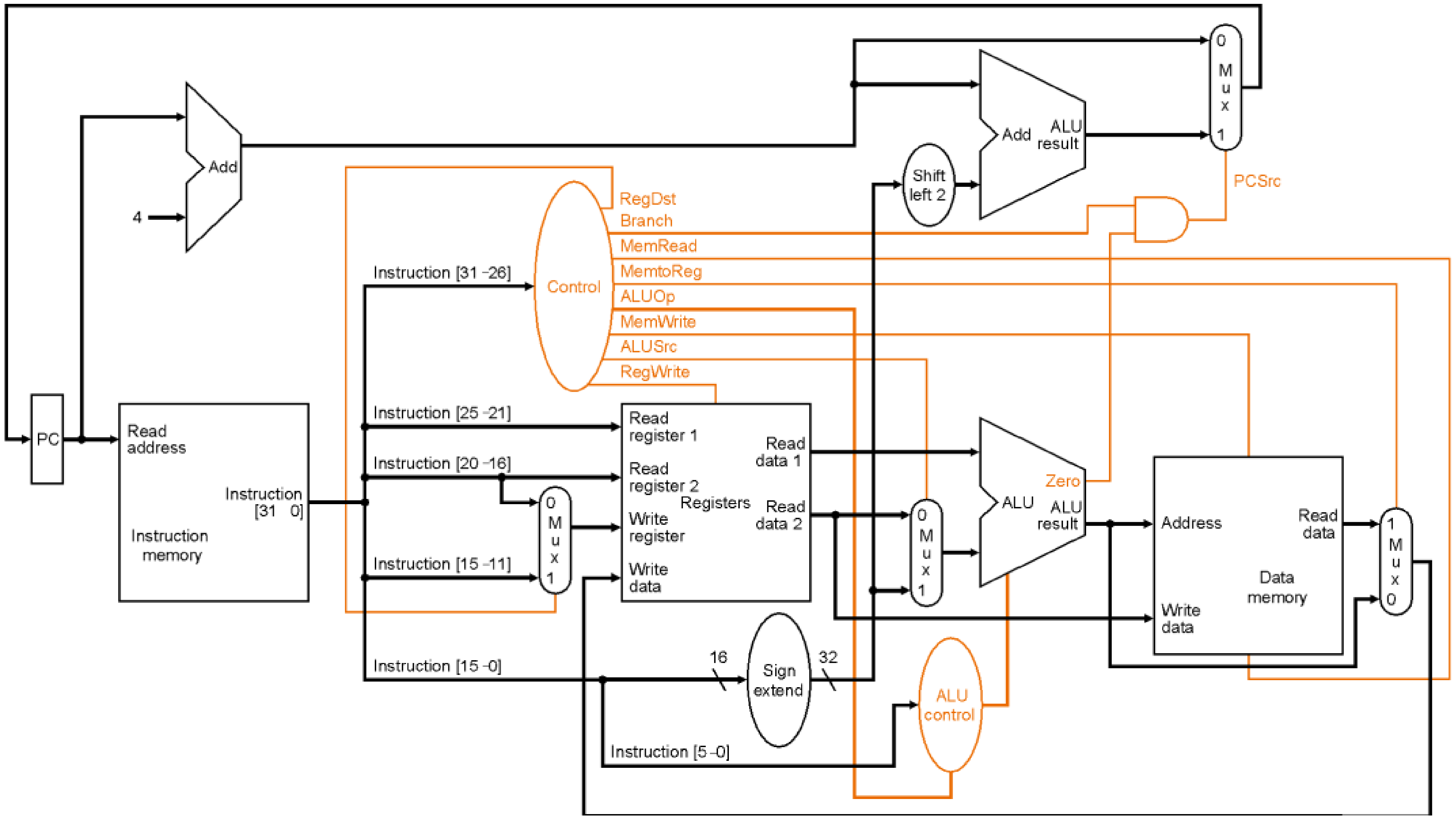
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1						0	0
sw								0	0
beq								0	1

Controlling the CPU



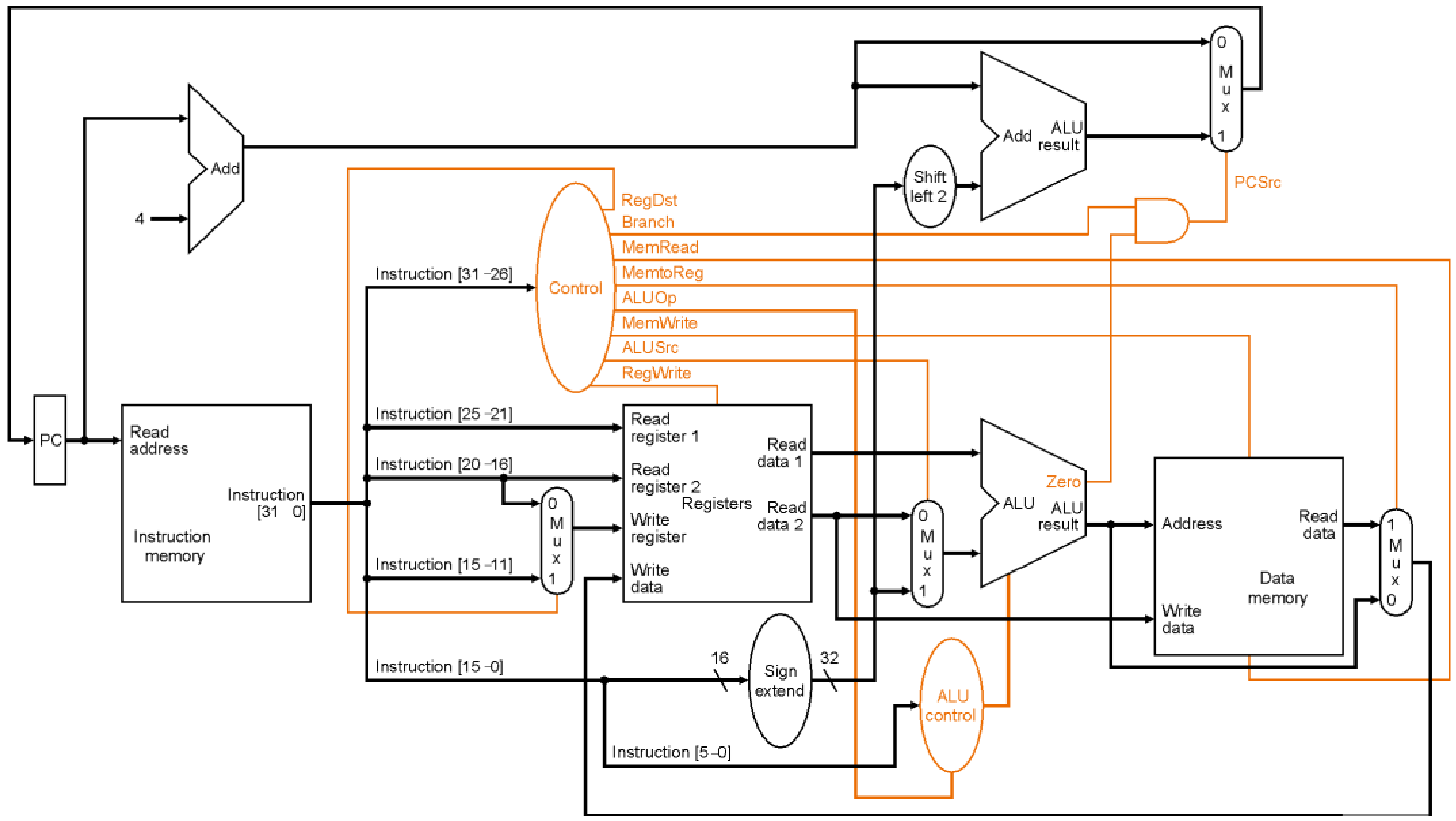
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1					0	0
`sw`								0	0
`beq`								0	1

Controlling the CPU



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1				0	0
`sw`								0	0
`beq`								0	1

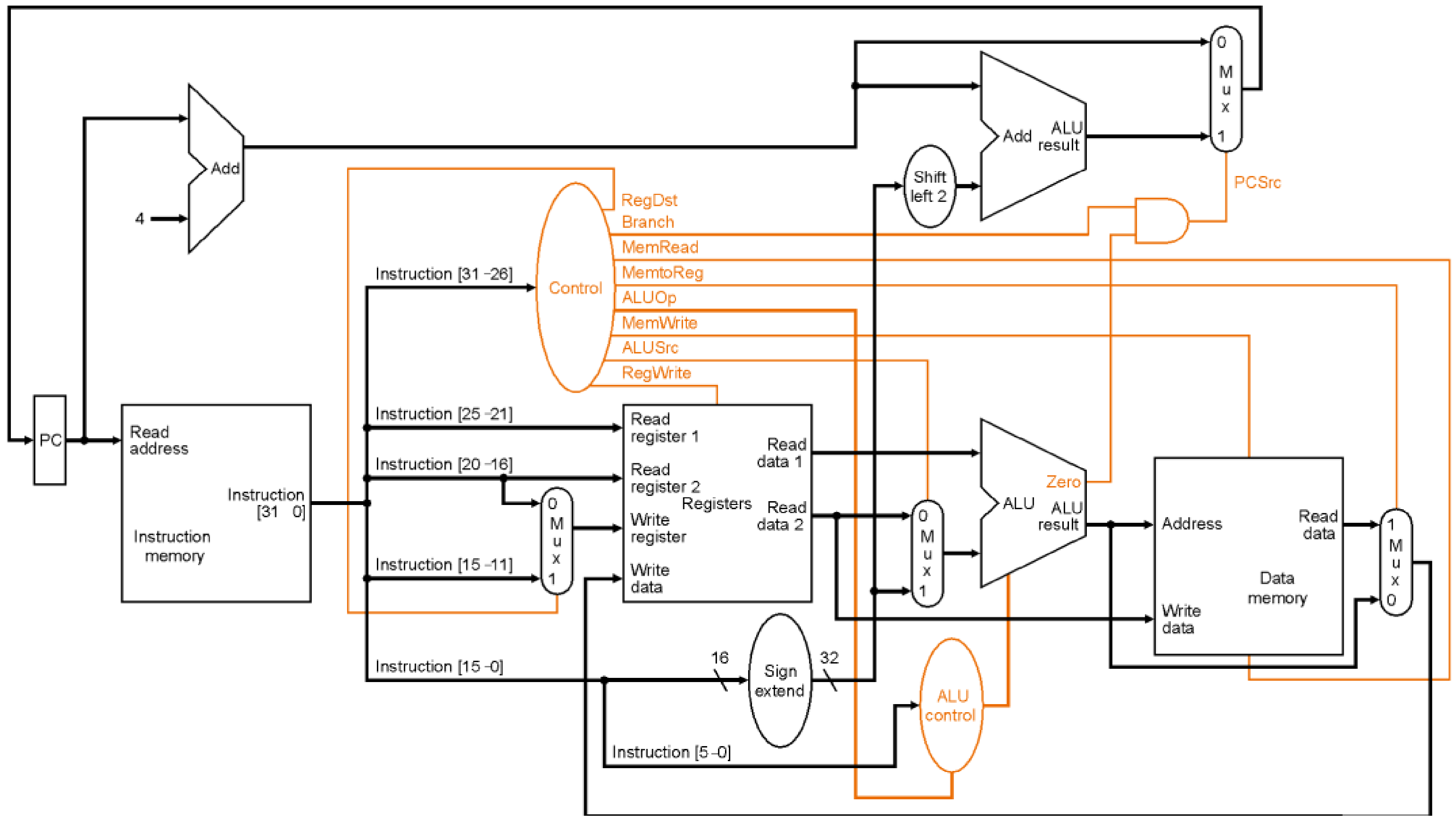
Controlling the CPU



Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1			0	0
`sw`								0	0
`beq`								0	1

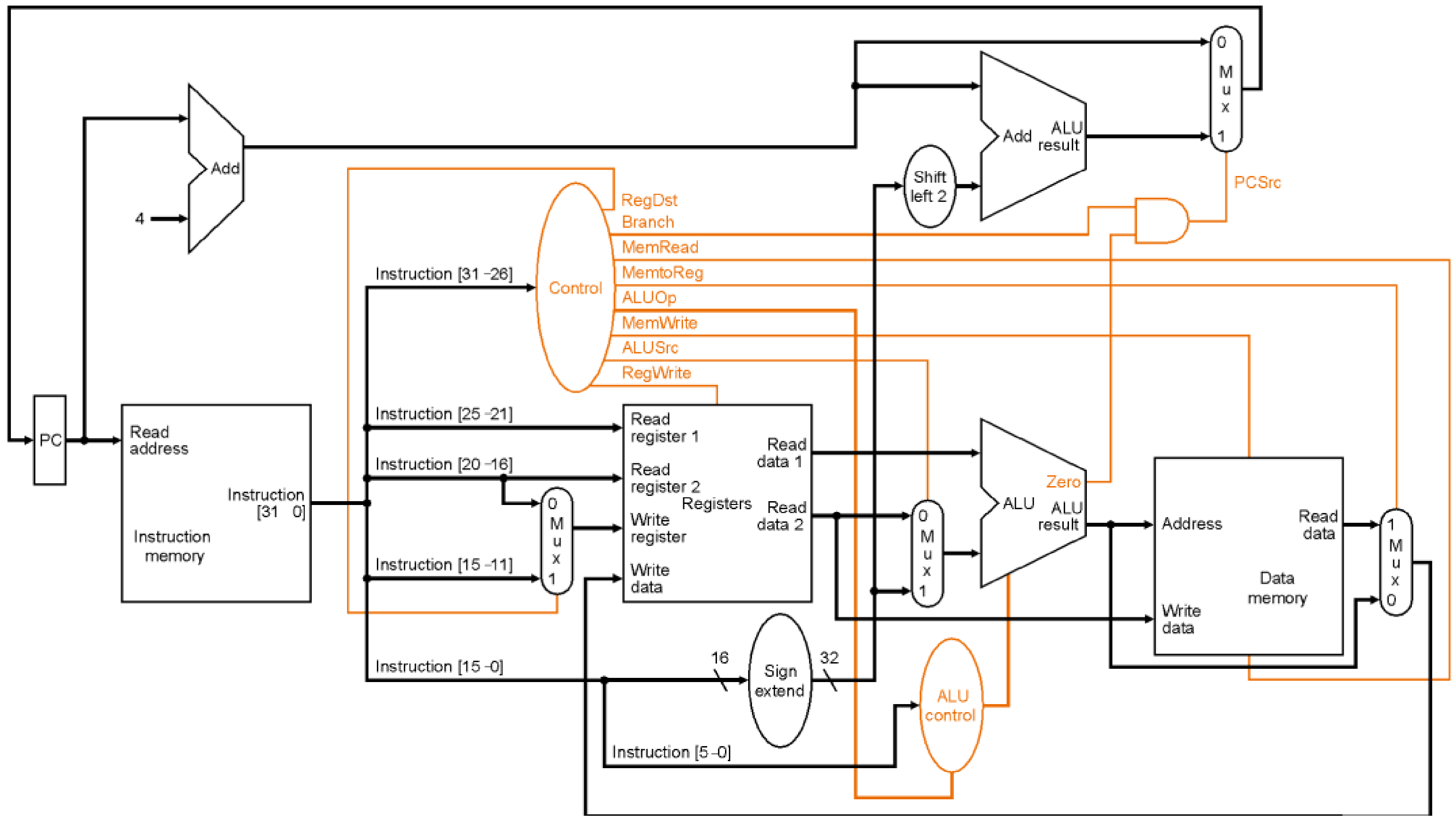
Tuesday, February 5, 13

Controlling the CPU



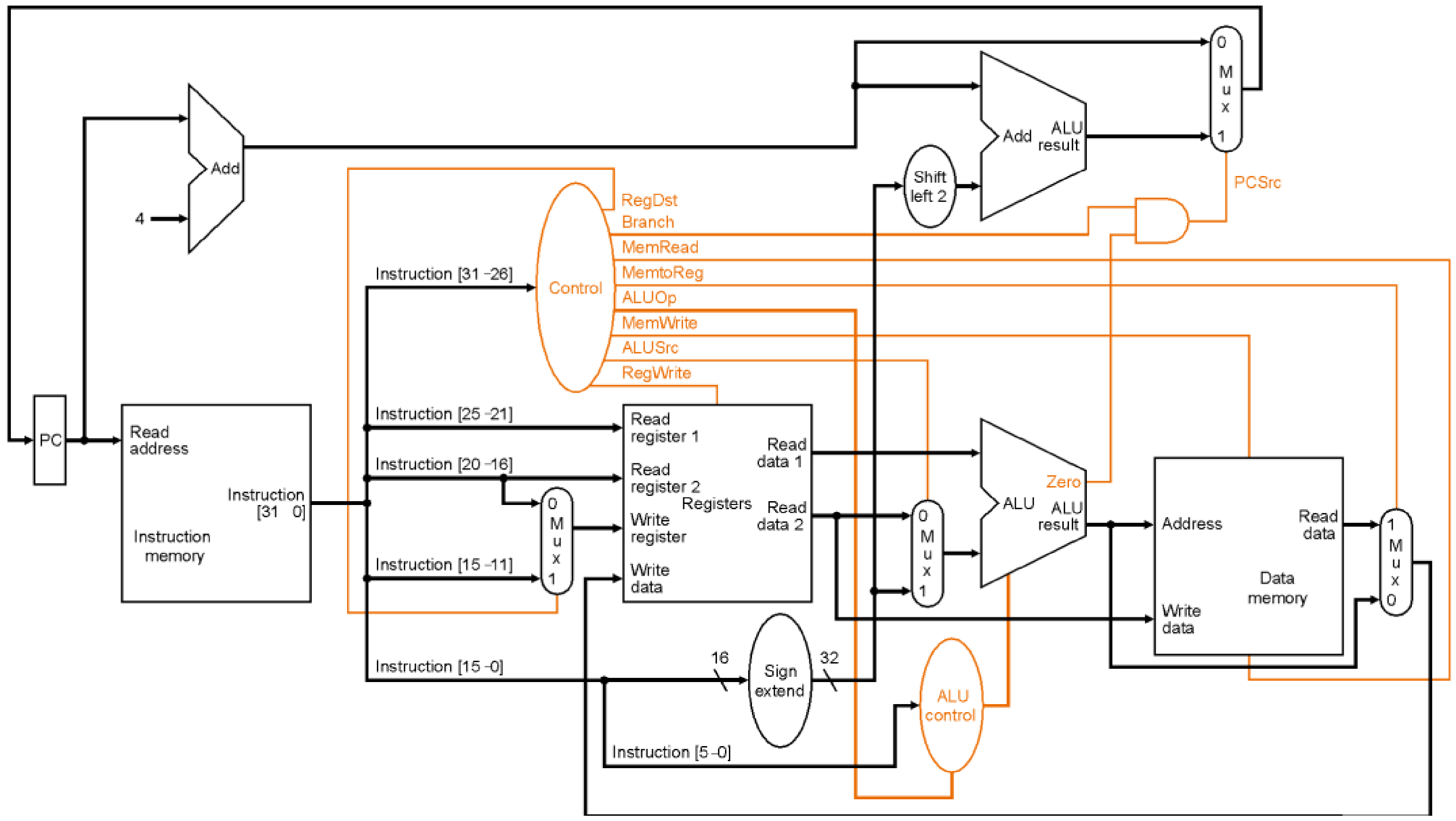
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0		0	0
`sw`								0	0
`beq`								0	1

Controlling the CPU



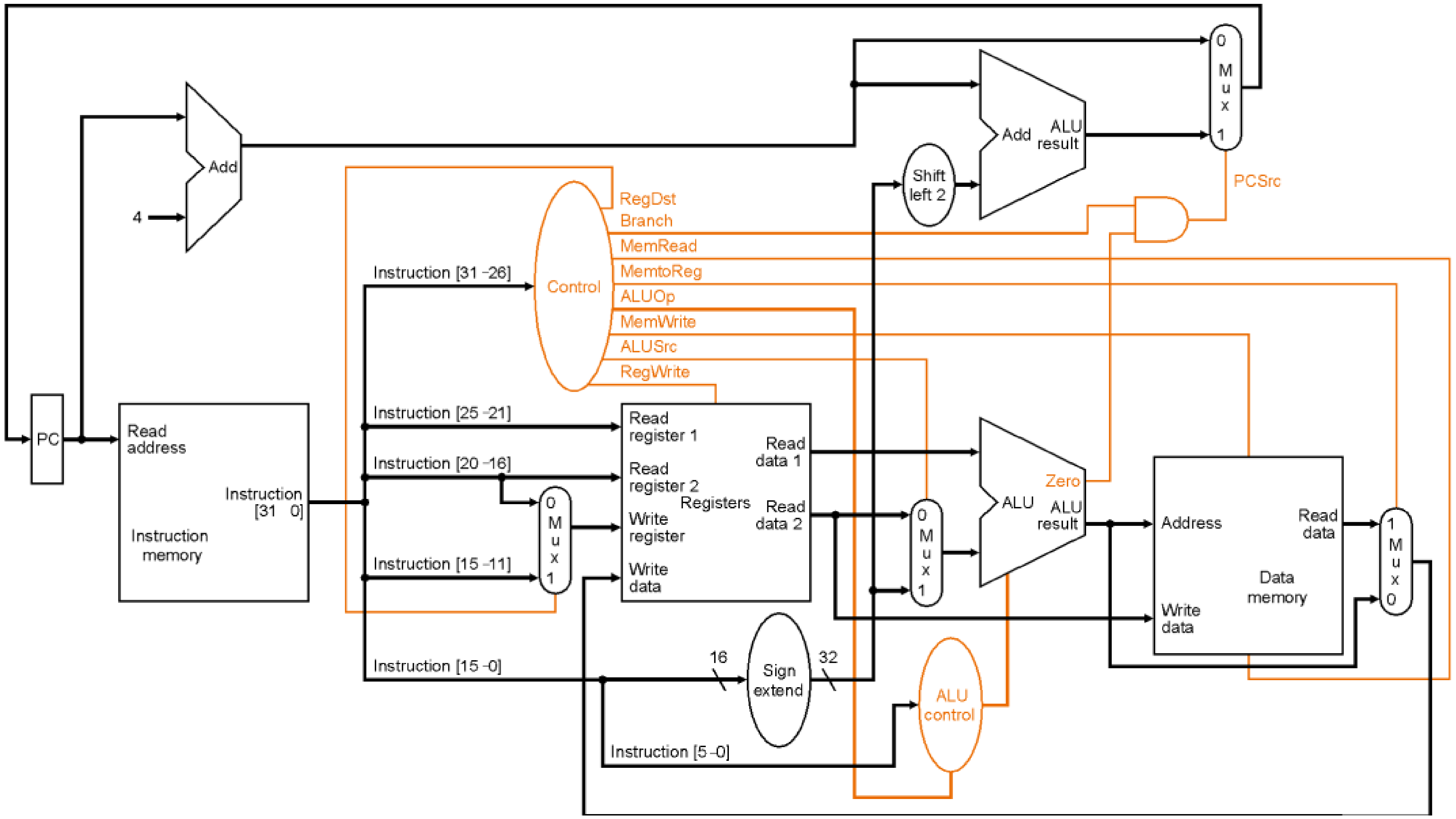
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`								0	0
`beq`								0	1

Controlling the CPU



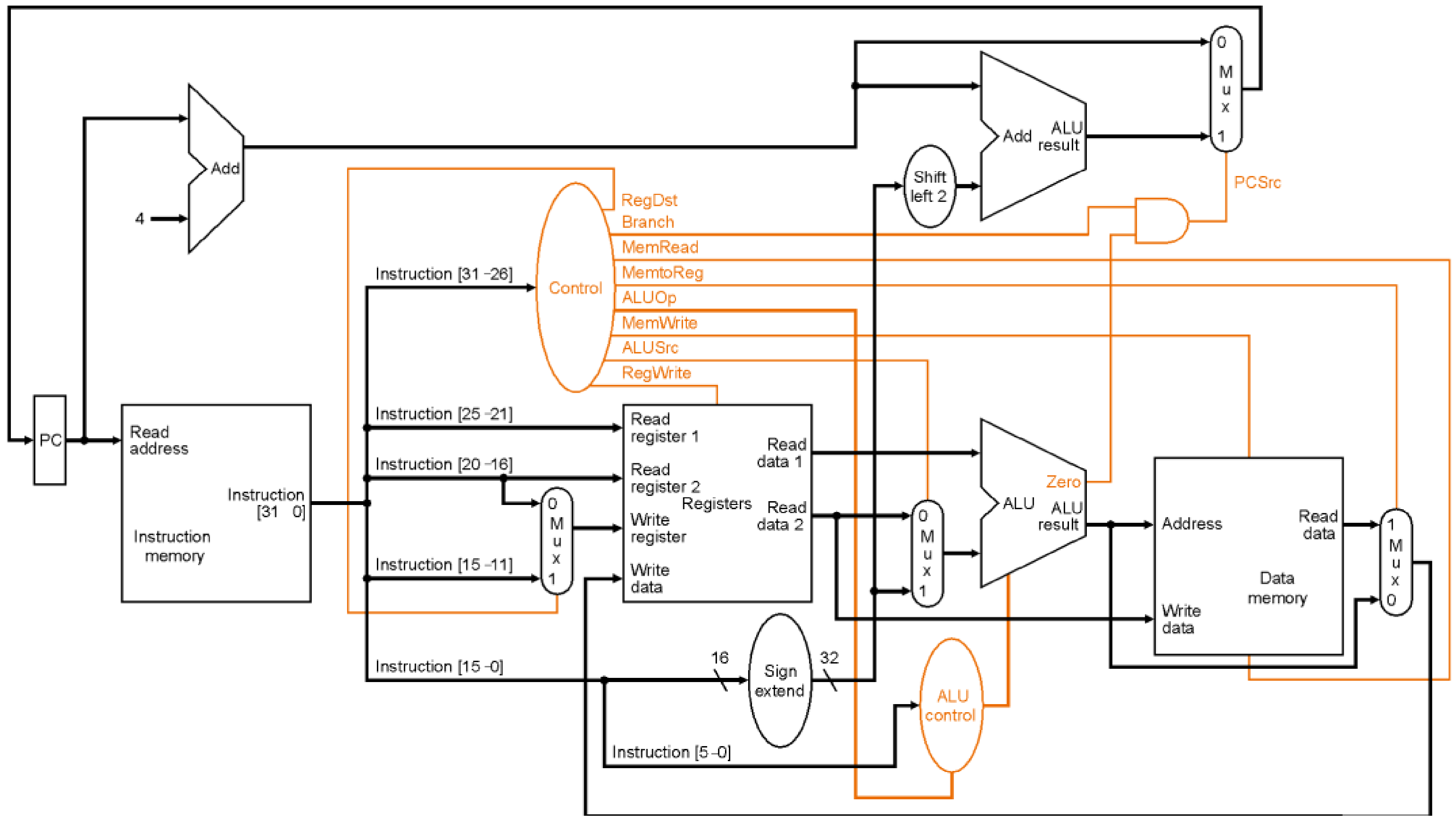
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X							0	0
beq								0	1

Controlling the CPU



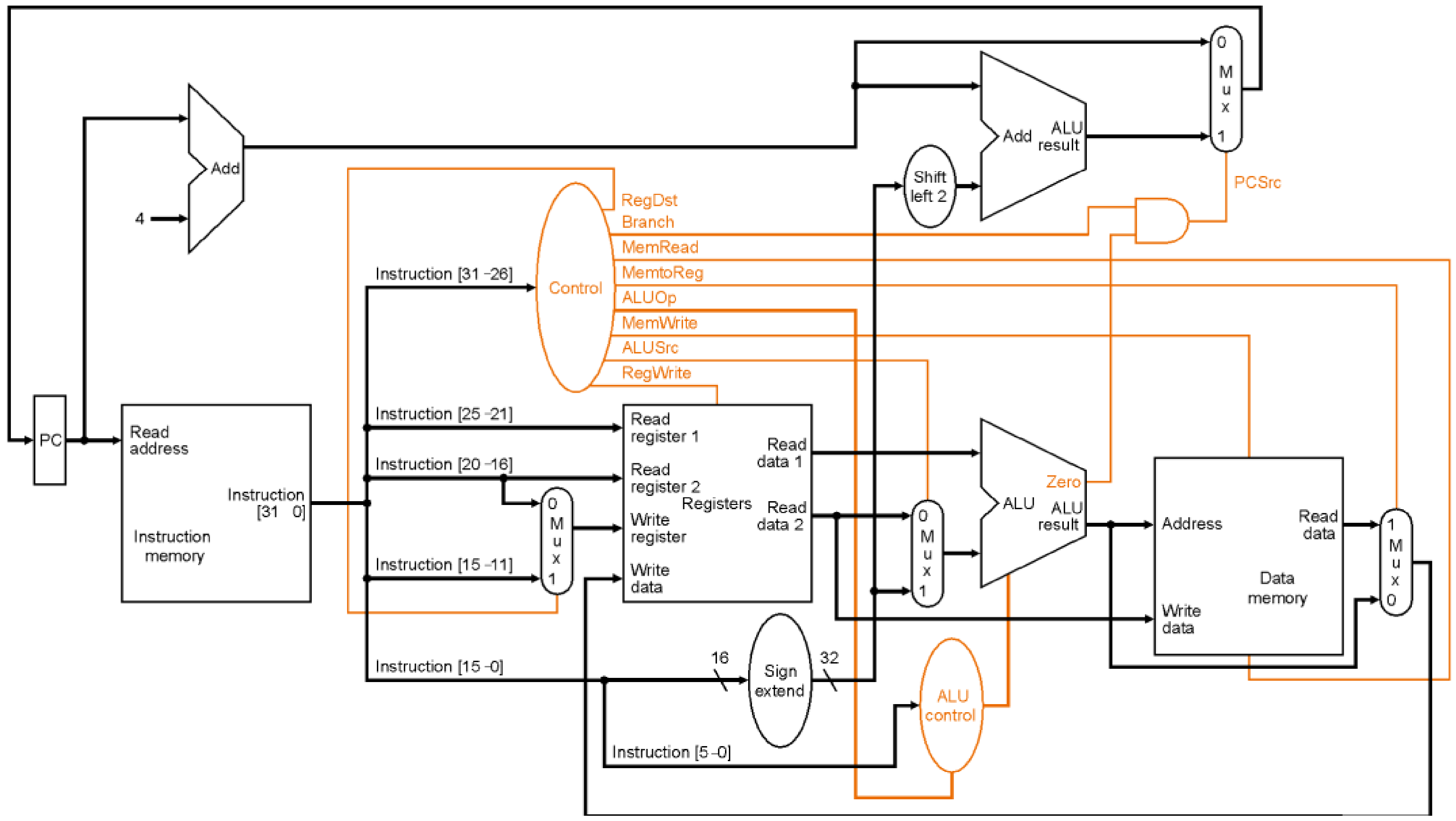
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1						0	0
beq								0	1

Controlling the CPU



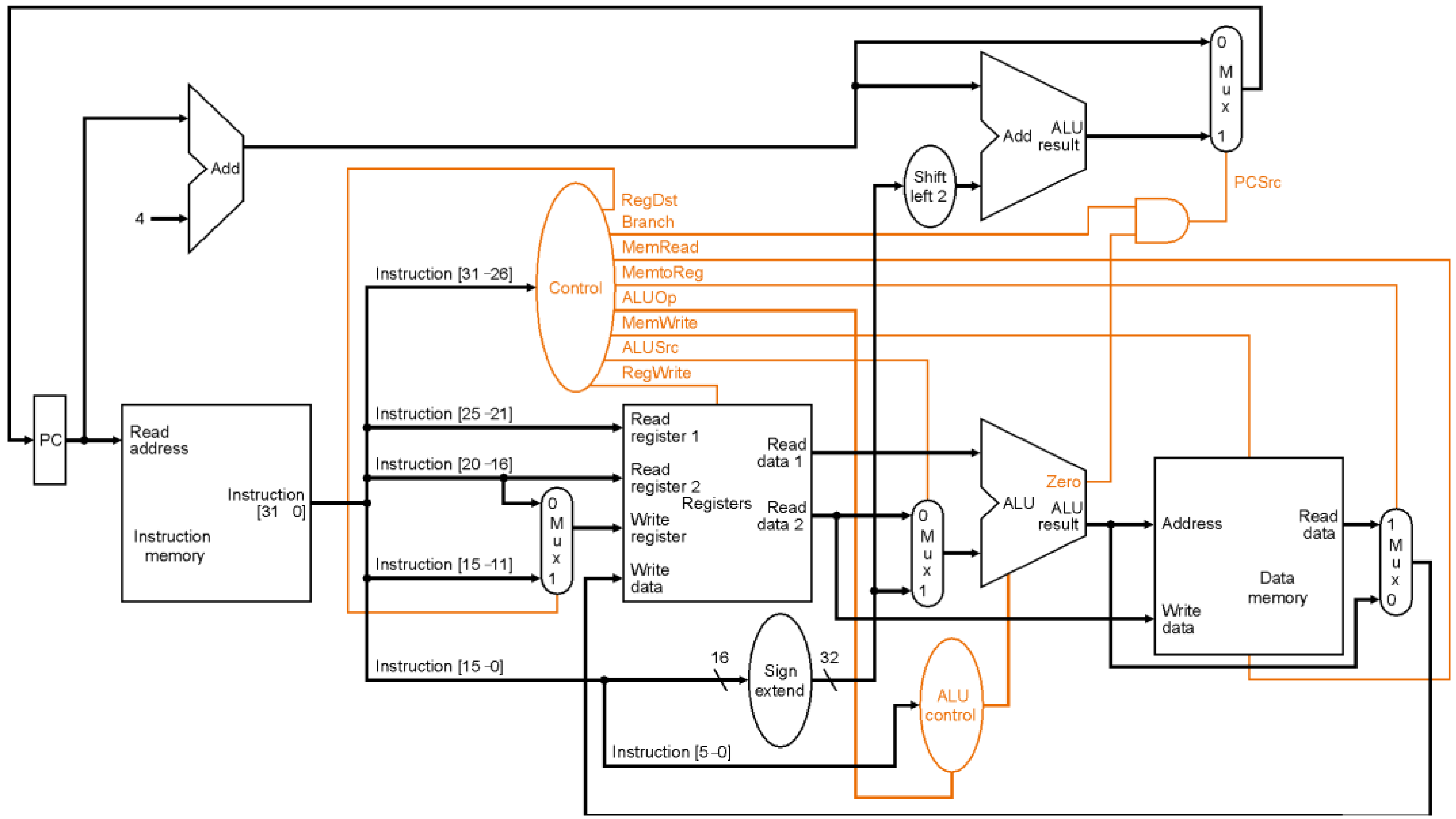
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X					0	0
`beq`								0	1

Controlling the CPU



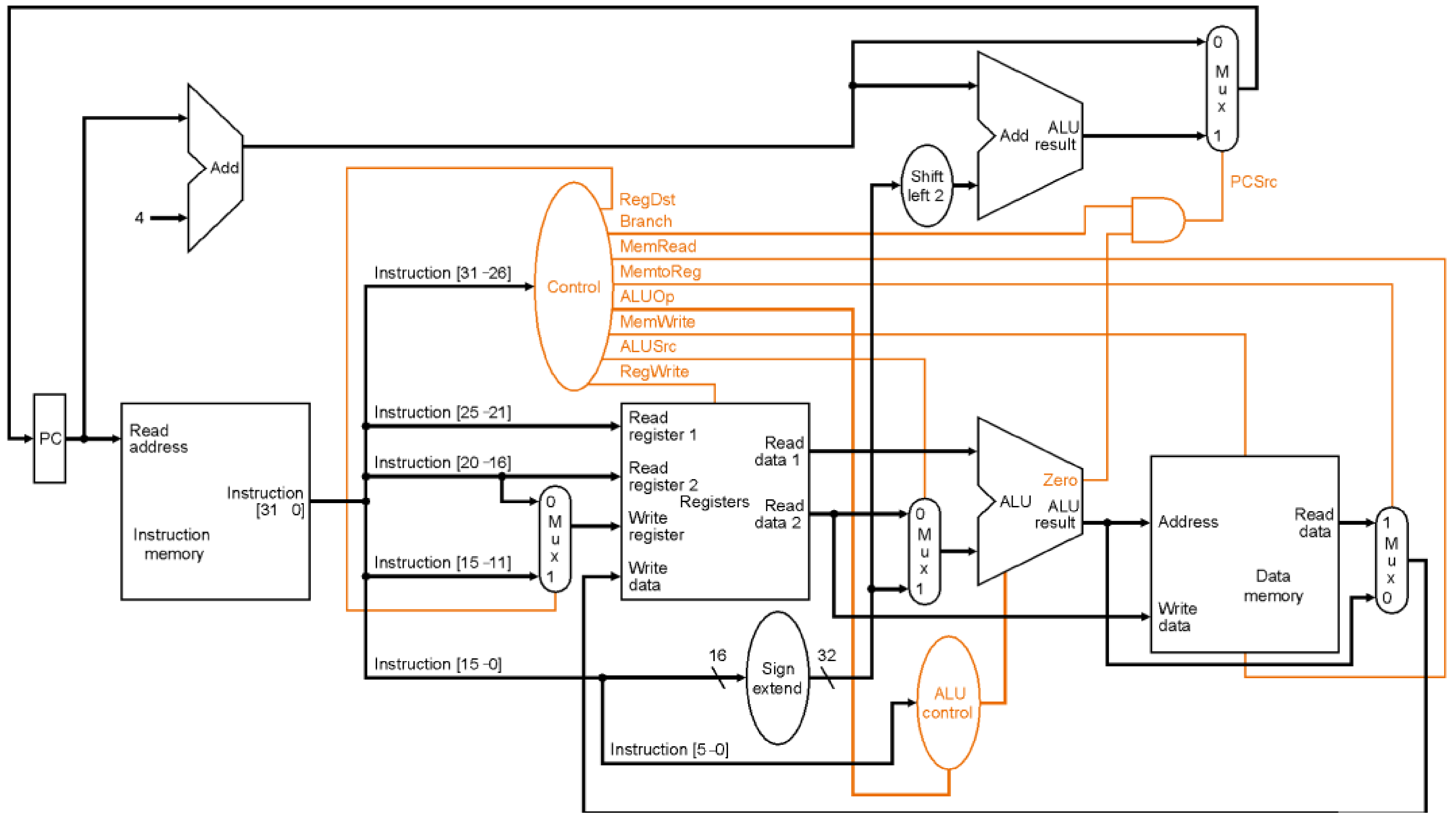
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0				0	0
beq								0	1

Controlling the CPU



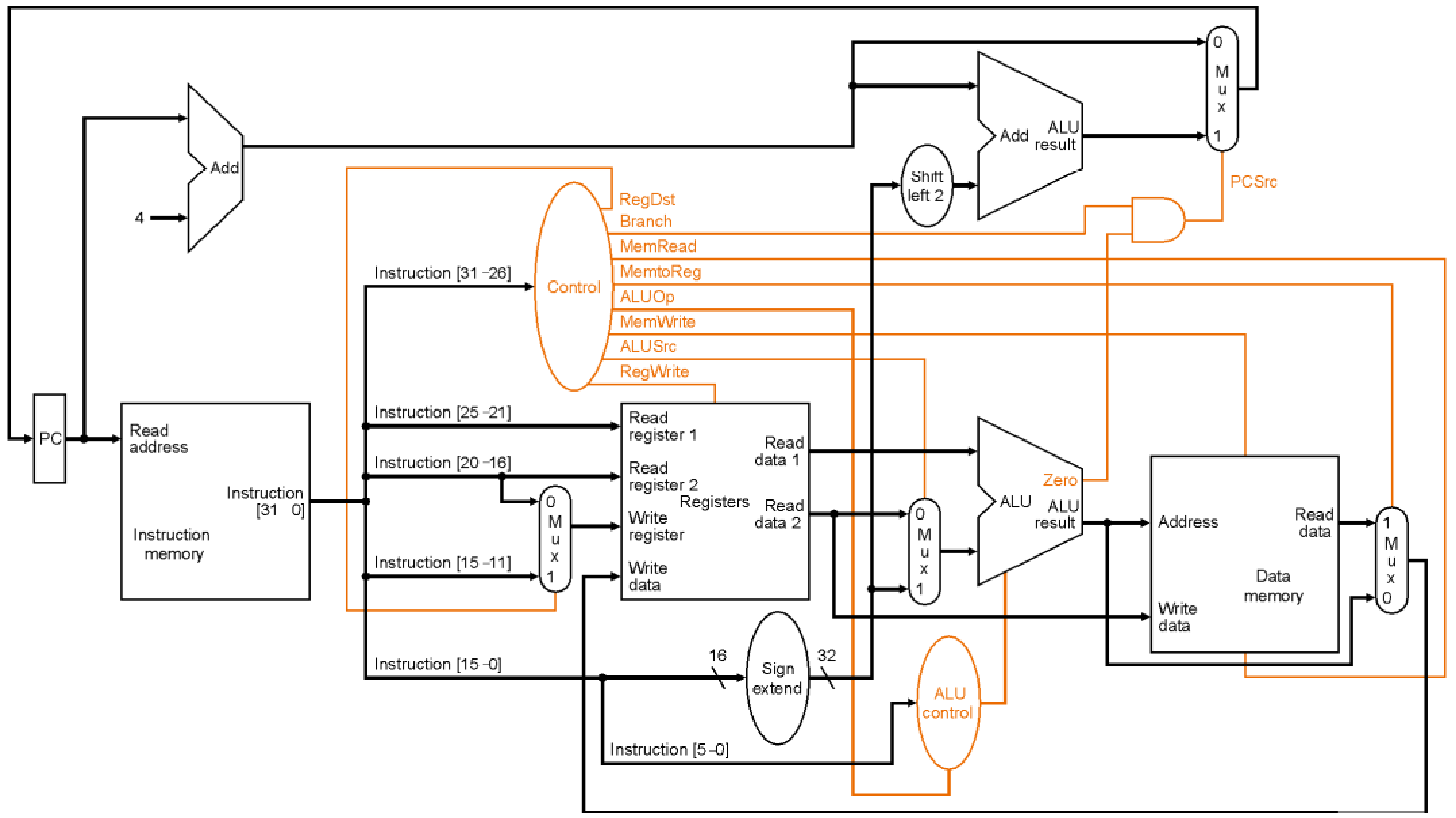
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0			0	0
beq								0	1

Controlling the CPU



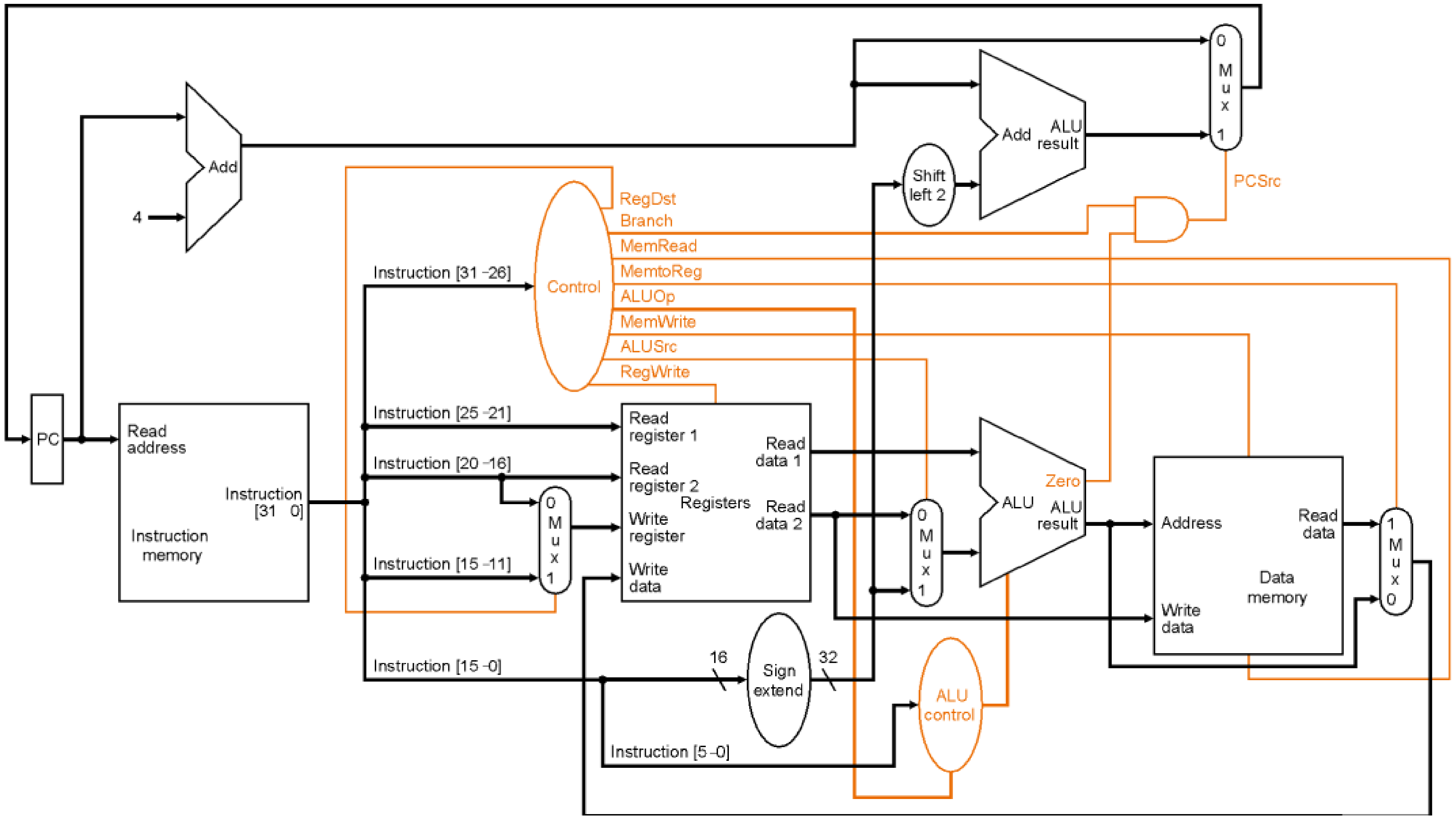
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1		0	0
beq								0	1

Controlling the CPU



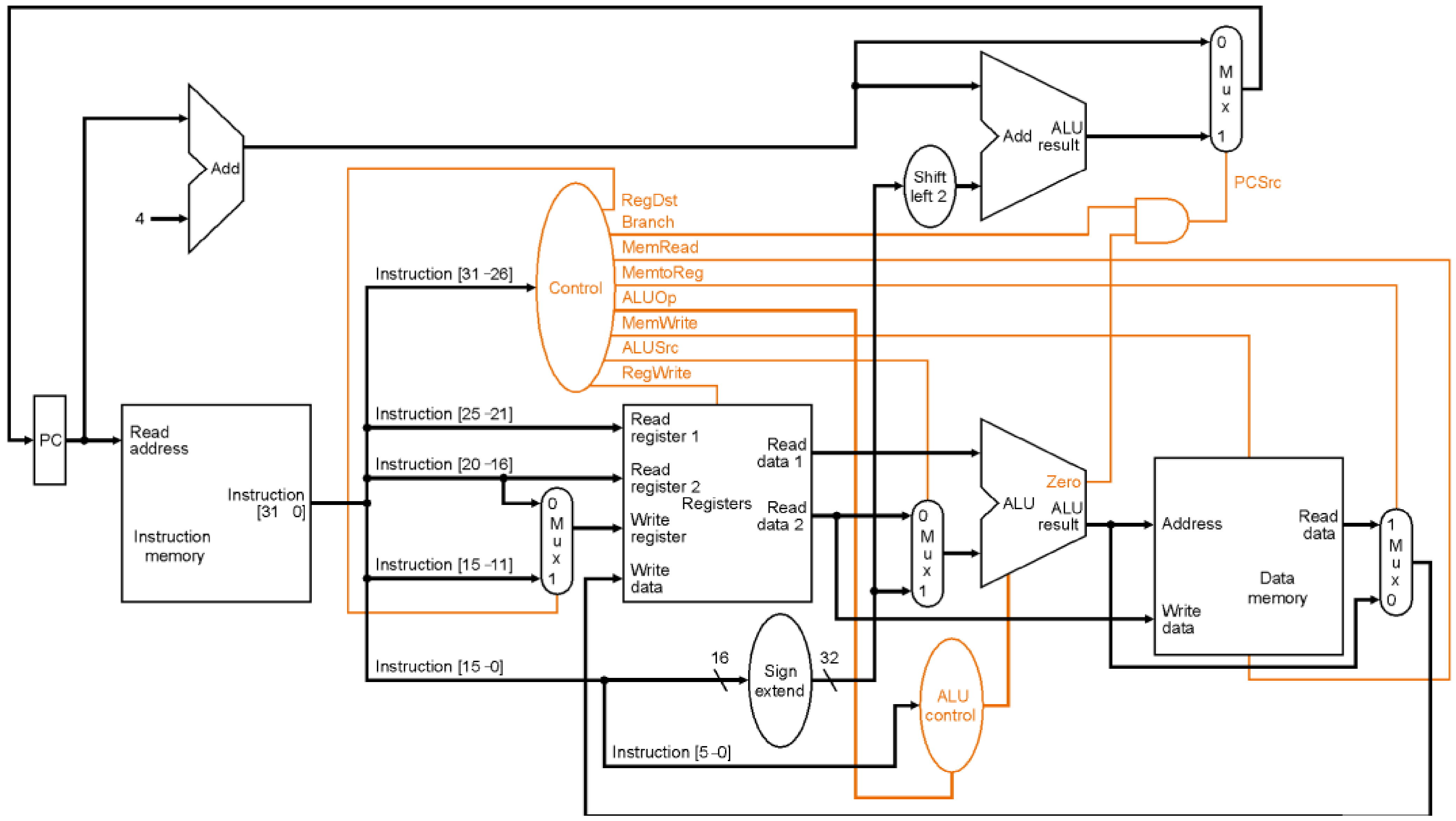
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`								0	1

Controlling the CPU



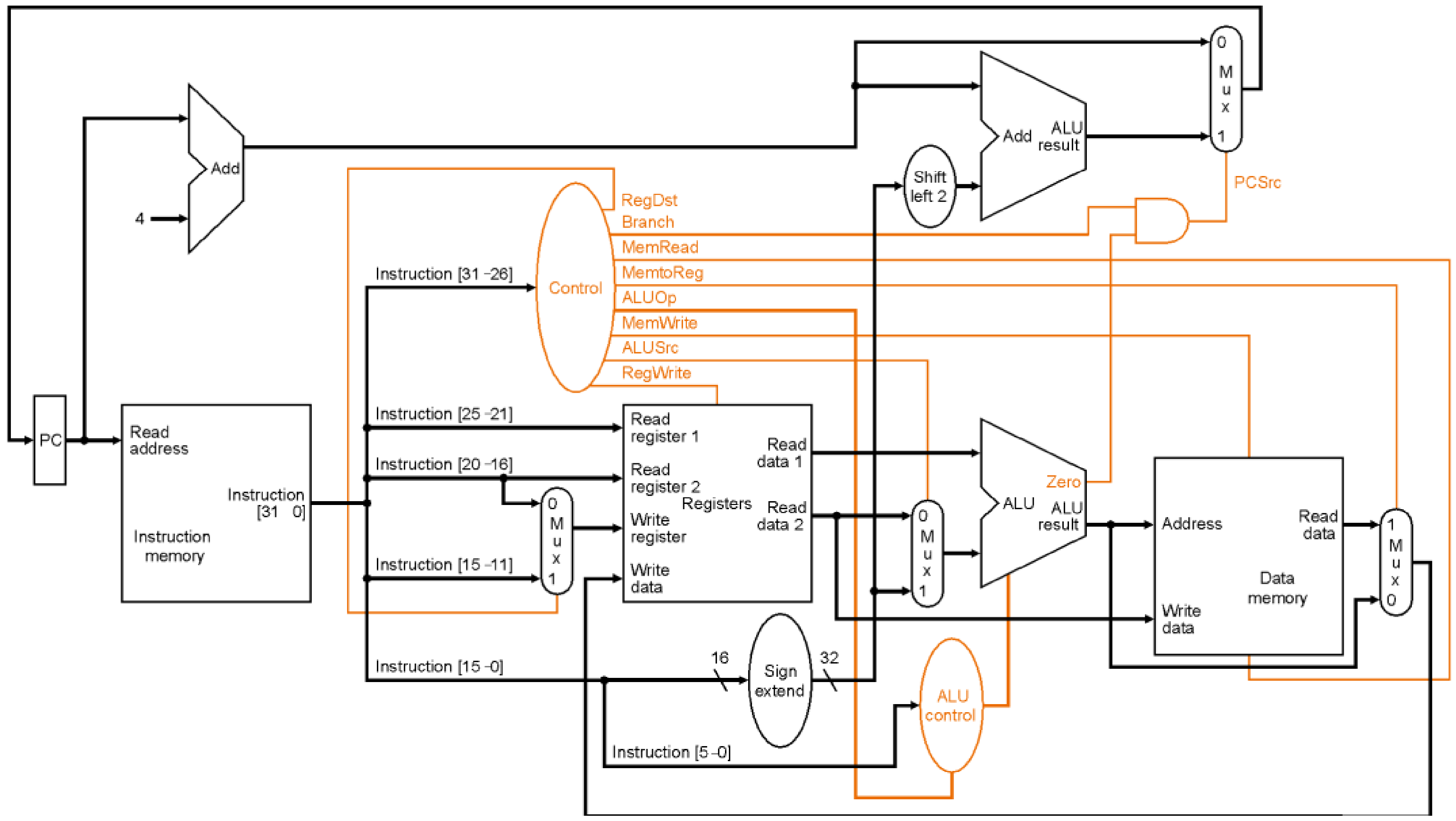
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X							0	1

Controlling the CPU



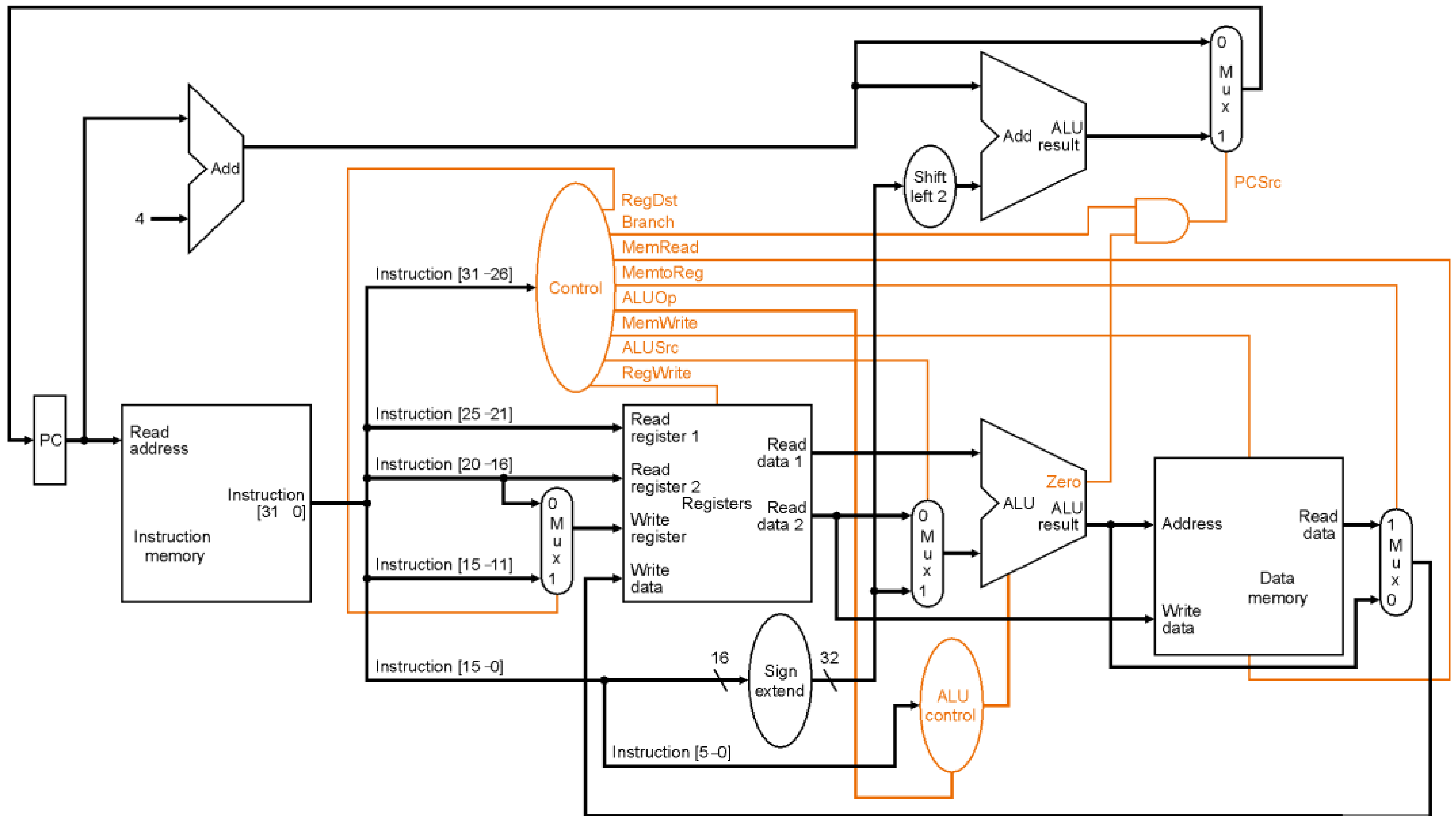
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0						0	1

Controlling the CPU



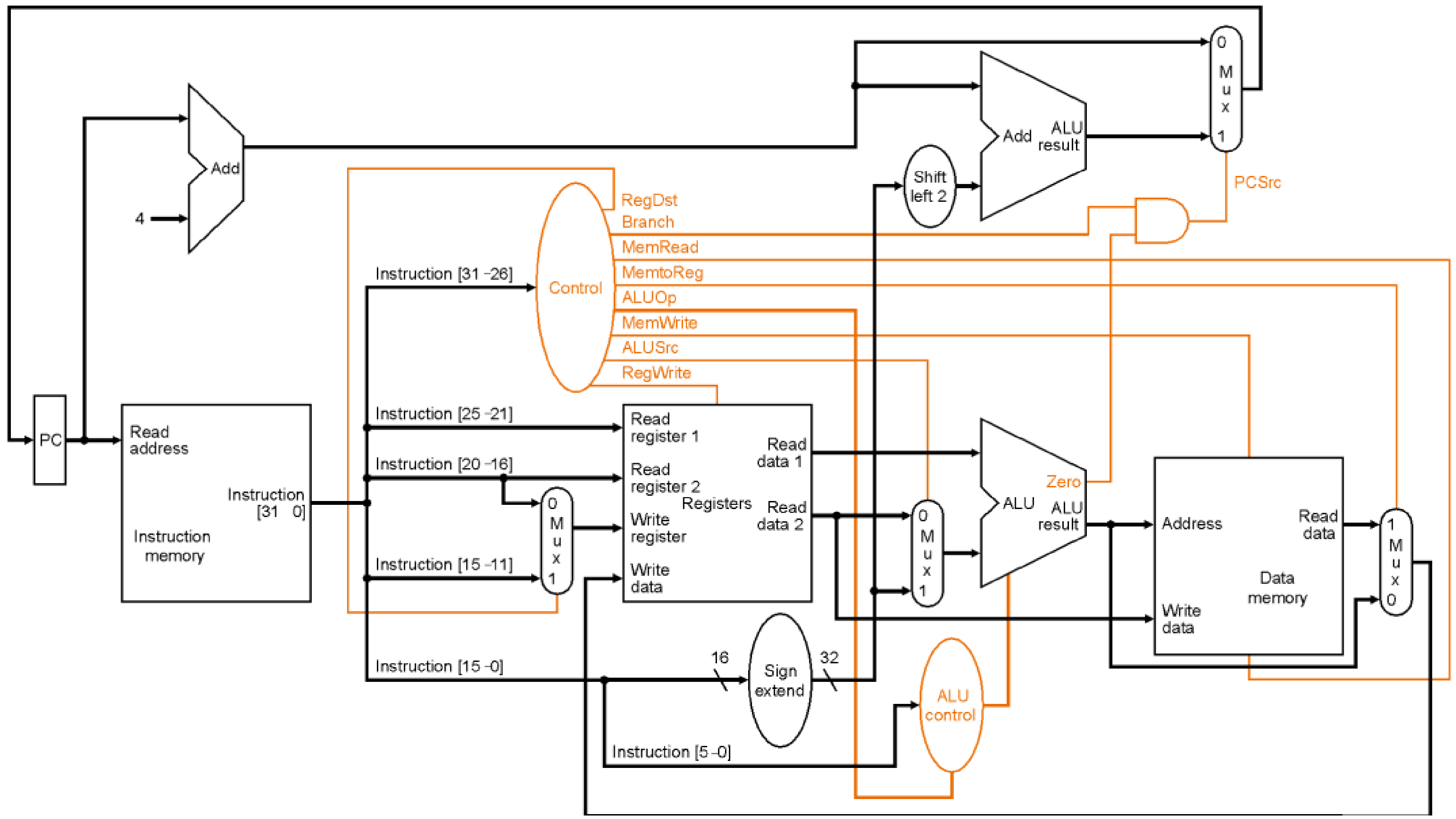
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X					0	1

Controlling the CPU



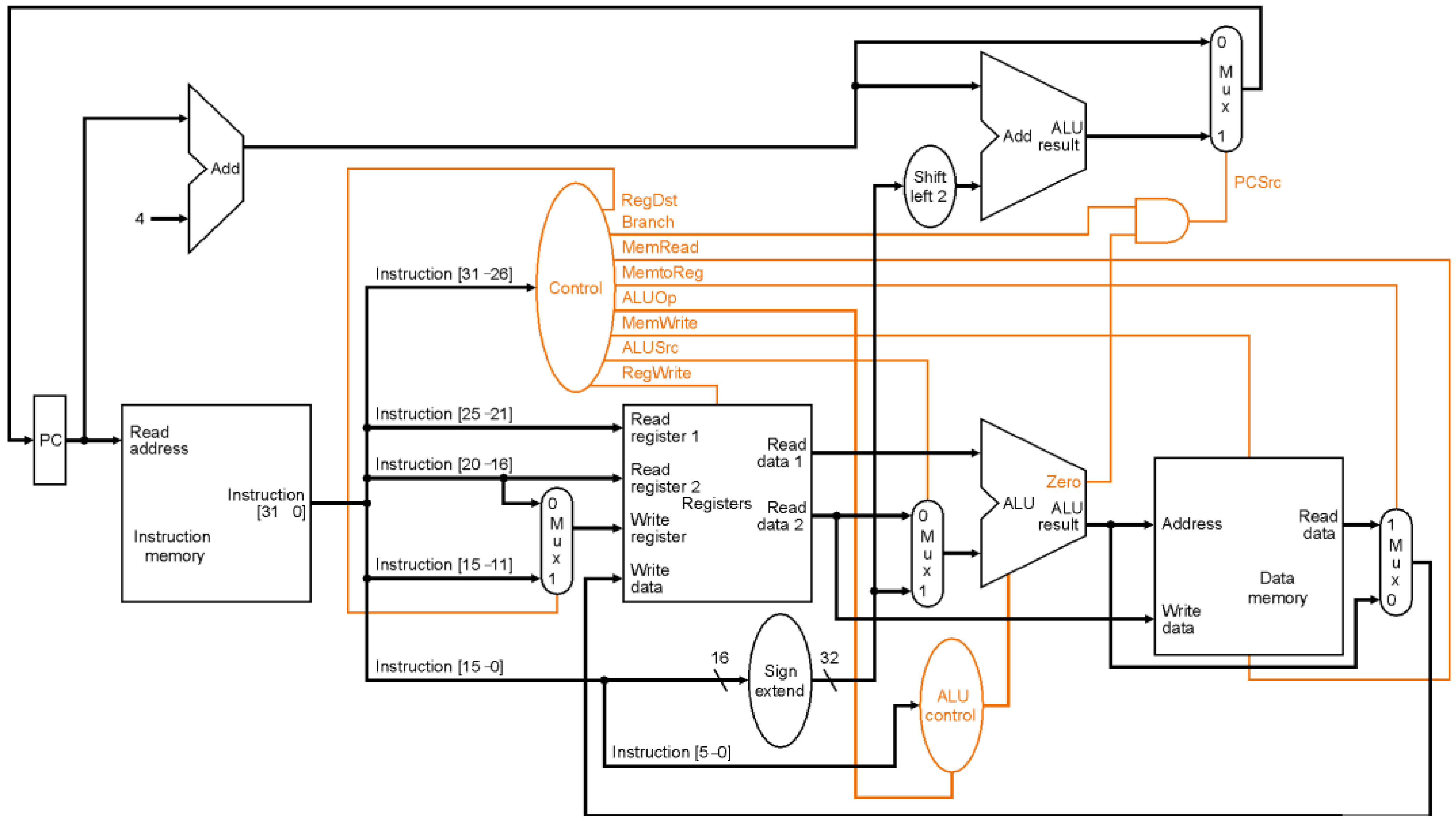
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0				0	1

Controlling the CPU



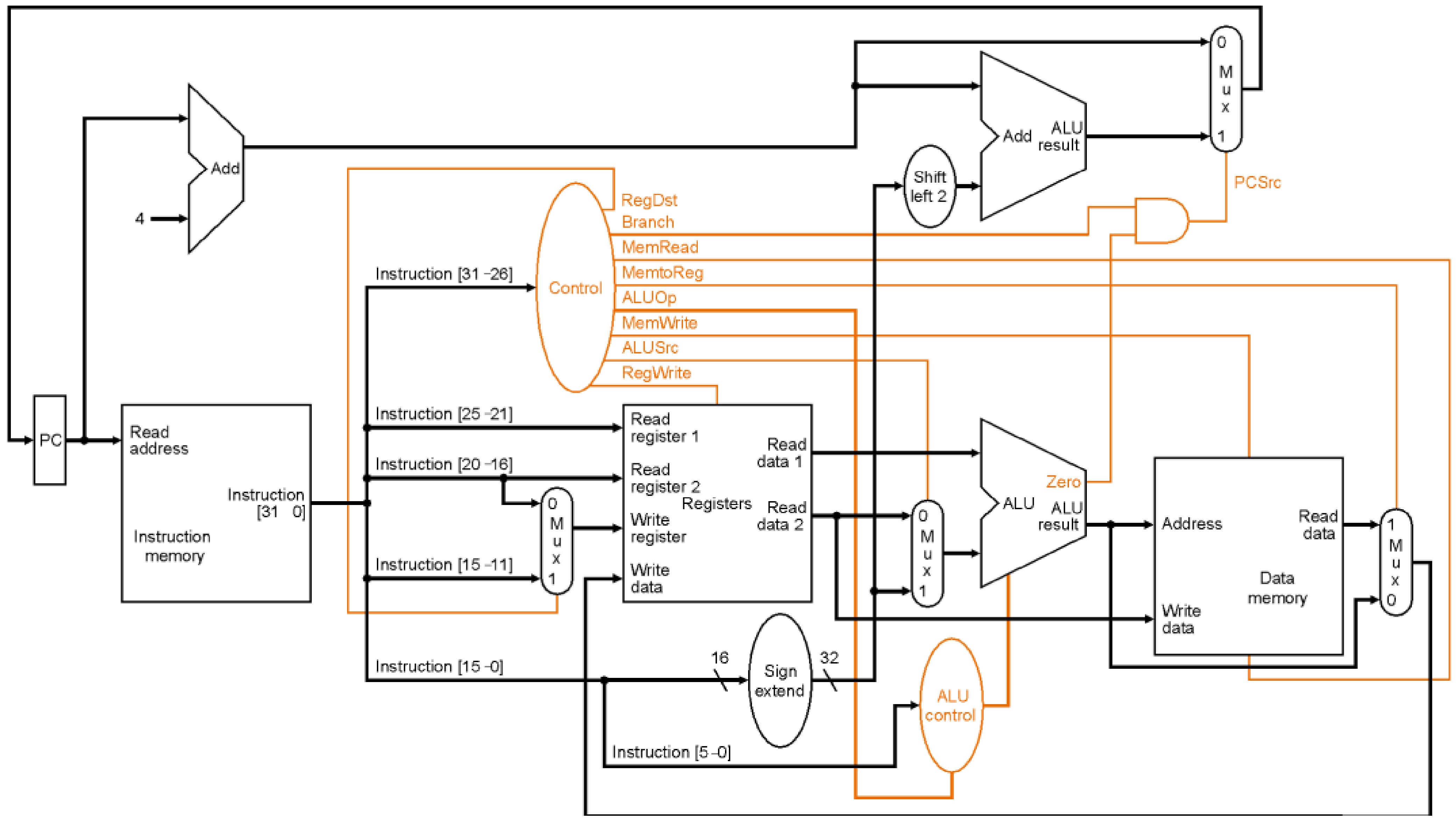
Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0			0	1

Controlling the CPU



Instruction	RegDst	ALUSrc	Mem to-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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		0	1

Controlling the CPU



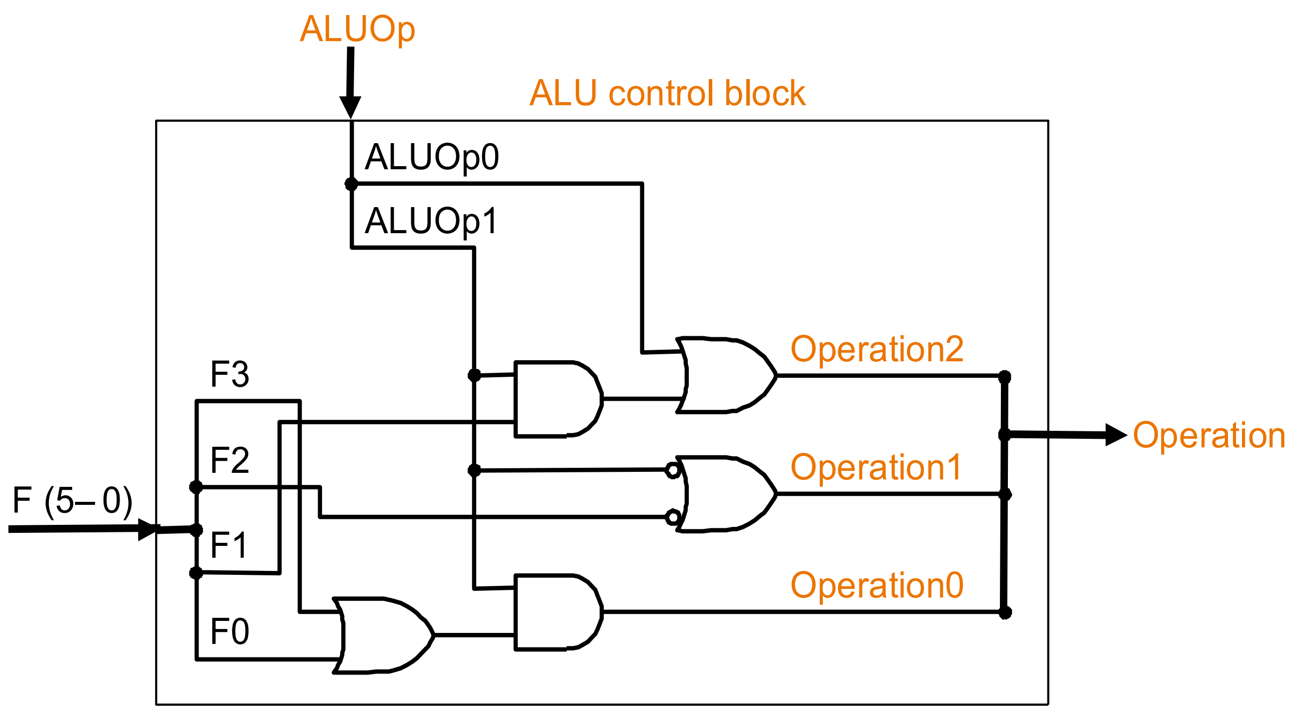
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

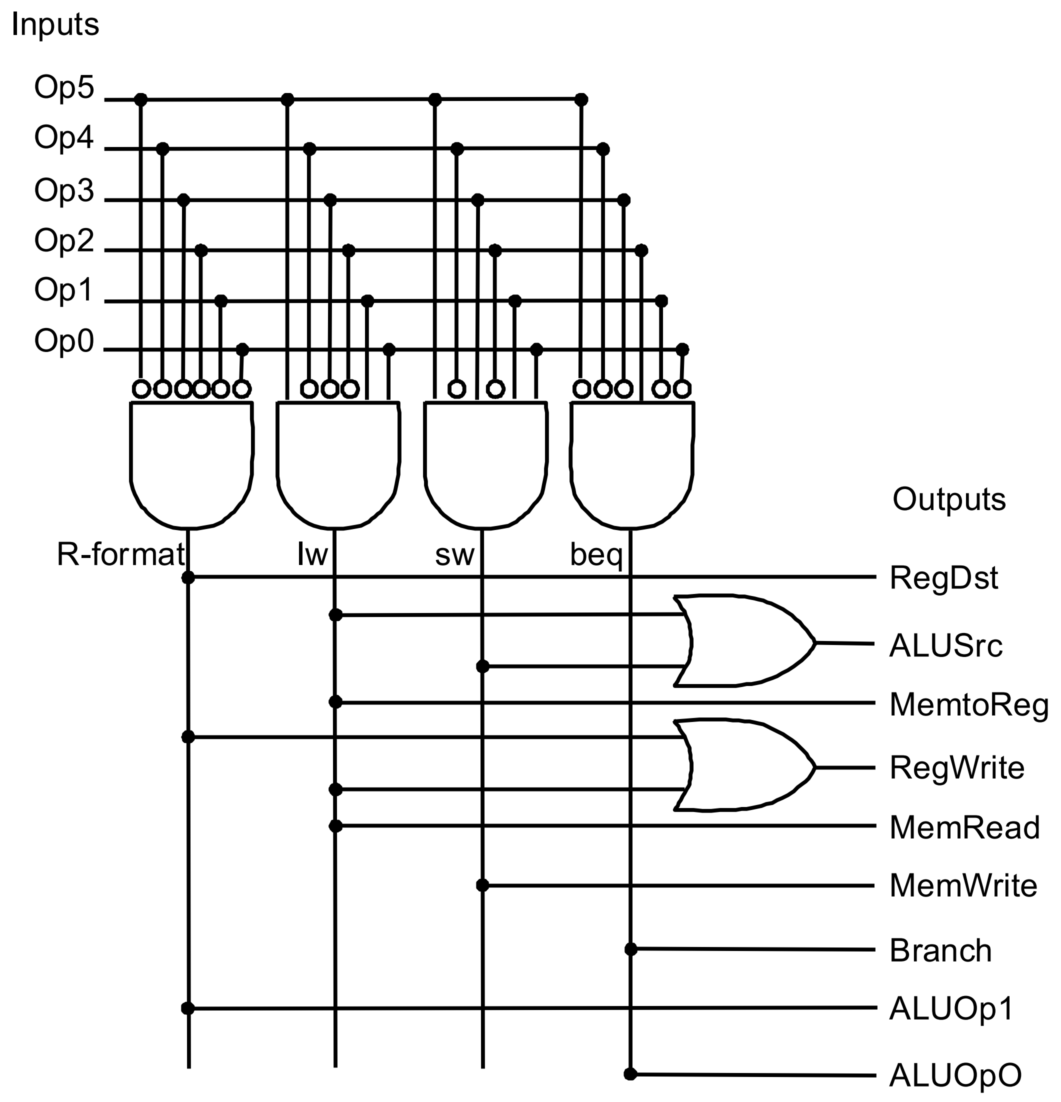
Control Truth Table

		R-format	**lw**	**sw**	**beq**
Opcode		000000	100011	101011	000100
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Control

- Simple Combinational Logic (truth tables)





Single Cycle CPU Summary

Single Cycle CPU Summary

- Easy, particularly the control

Single Cycle CPU Summary

- Easy, particularly the control
- Which instruction takes the longest? By how much? Why is that a problem?
  - ET = IC * CPI * CT

Single Cycle CPU Summary

- Easy, particularly the control
- Which instruction takes the longest? By how much? Why is that a problem?
    - ET = IC * CPI * CT
- What else can we do?

Single Cycle CPU Summary

- Easy, particularly the control
- Which instruction takes the longest? By how much? Why is that a problem?
  - ET = IC * CPI * CT
- What else can we do?
- When does a multi-cycle implementation make sense?
  - e.g., 70% of instructions take 75 ns, 30% take 200 ns?
  - suppose 20% overhead for extra latches

Single Cycle CPU Summary

- Easy, particularly the control
- Which instruction takes the longest? By how much? Why is that a problem?
  - ET = IC * CPI * CT
- What else can we do?
- When does a multi-cycle implementation make sense?
  - e.g., 70% of instructions take 75 ns, 30% take 200 ns?
  - suppose 20% overhead for extra latches
- Real machines have much more variable instruction latencies than this.