Multi-cycle
Datapath and control

Single-cycle implementation

- As we've seen, single-cycle implementation, although easy to implement, could potentially be very inefficient
- In single-cycle, we define a clock cycle to be the length of time needed to execute a single instruction. So, our lower bound on the clock period is the length of the most-time consuming instruction
- In our previous example, our jump instruction needs only 4ns but our clock period must be 13ns to accommodate the load word instruction!

Multi-cycle implementation

- We can get around some of the disadvantages by introducing a little more complexity to our datapath
- Instead of viewing the instruction as one big task that needs to be performed, in multi-cycle the instructions are broken up into smaller fundamental steps
- As a result, we can shorten the clock period and perform the instructions incrementally across multiple cycles
- What are these fundamental steps? Well, let's take a look at what our instructions actually need to do…

R format steps

- An instruction is fetched from instruction memory and the PC is incremented
- Read two source register values from the register file.
- Perform the ALU operation on the register data operands
- Write the result of the ALU operation to the register file

LOAD steps

- An instruction is fetched from instruction memory and the PC is incremented
- Read a source register value from the register file and sign-extend the 16 least significant bits of the instruction
- Perform the ALU operation that computes the sum of the value in the register and the sign-extended immediate value from the instruction
- Access data memory at the address given by the result from the ALU
- Write the result of the memory value to the register file

STORE steps

- An instruction is fetched from instruction memory and the PC is incremented
- Read two source register values from the register file and sign-extend the 16 least significant bits of the instruction
- Perform the ALU operation that computes the sum of the value in the register and the sign-extended immediate value from the instruction
- Update data memory at the address given by the result from the ALU

BRANCH EQUAL (BEQ) steps

- An instruction is fetched from instruction memory and the PC is incremented
- Read two source register values from the register file and sign-extend the 16 least significant bits of the instruction and then left shifts it by two
- The ALU performs a subtract on the data values read from the register file. The value of PC+4 is added with the sign-extended left-shifted-by-two immediate value from the instruction, which results in the branch target address
- The Zero result from the ALU is used to decide which adder result should be used to update the PC

JUMP steps

- An instruction is fetched from instruction memory and the PC is incremented
- Concatenate the four most significant bits of PC+4, the 26 least significant bits of the instruction, and two zero bits. Assign the result to the PC

General steps

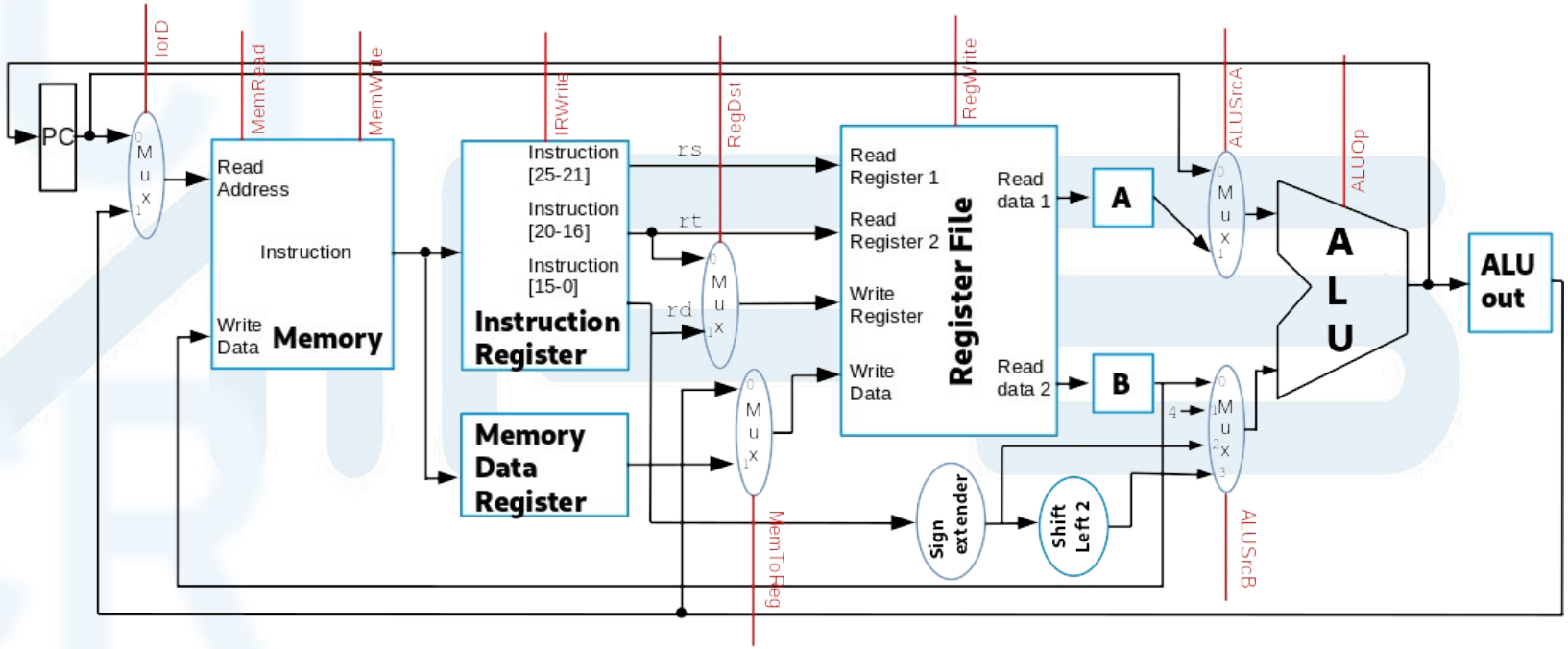
So, generally, we can say we need to perform the following steps:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execution, memory address computation, branch completion, or jump completion
4. Memory access or R-type instruction completion
5. Memory read completion

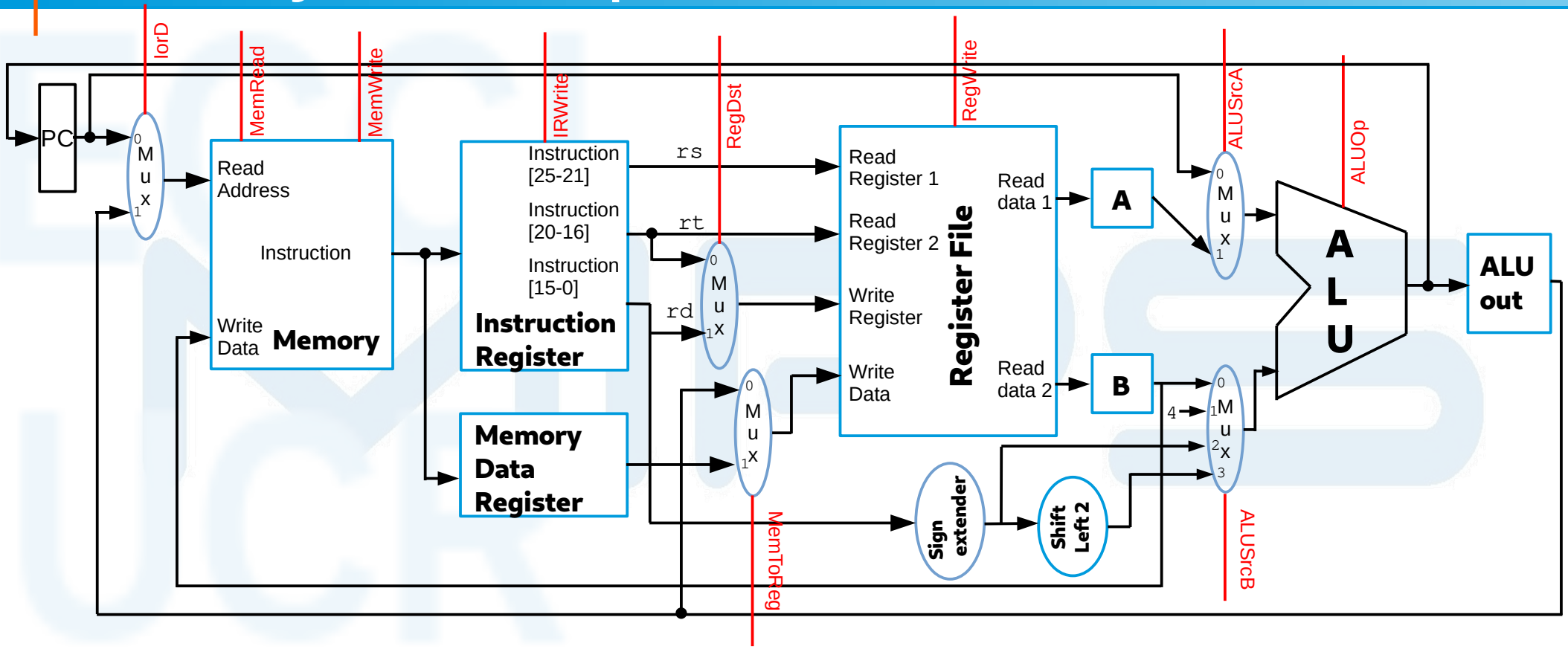
Multi-cycle datapath

Here is a general overview of our new multi-cycle datapath

- We now have a single memory element that interacts with both instructions and data
- Single ALU unit, no dedicated adders
- Several temporary registers

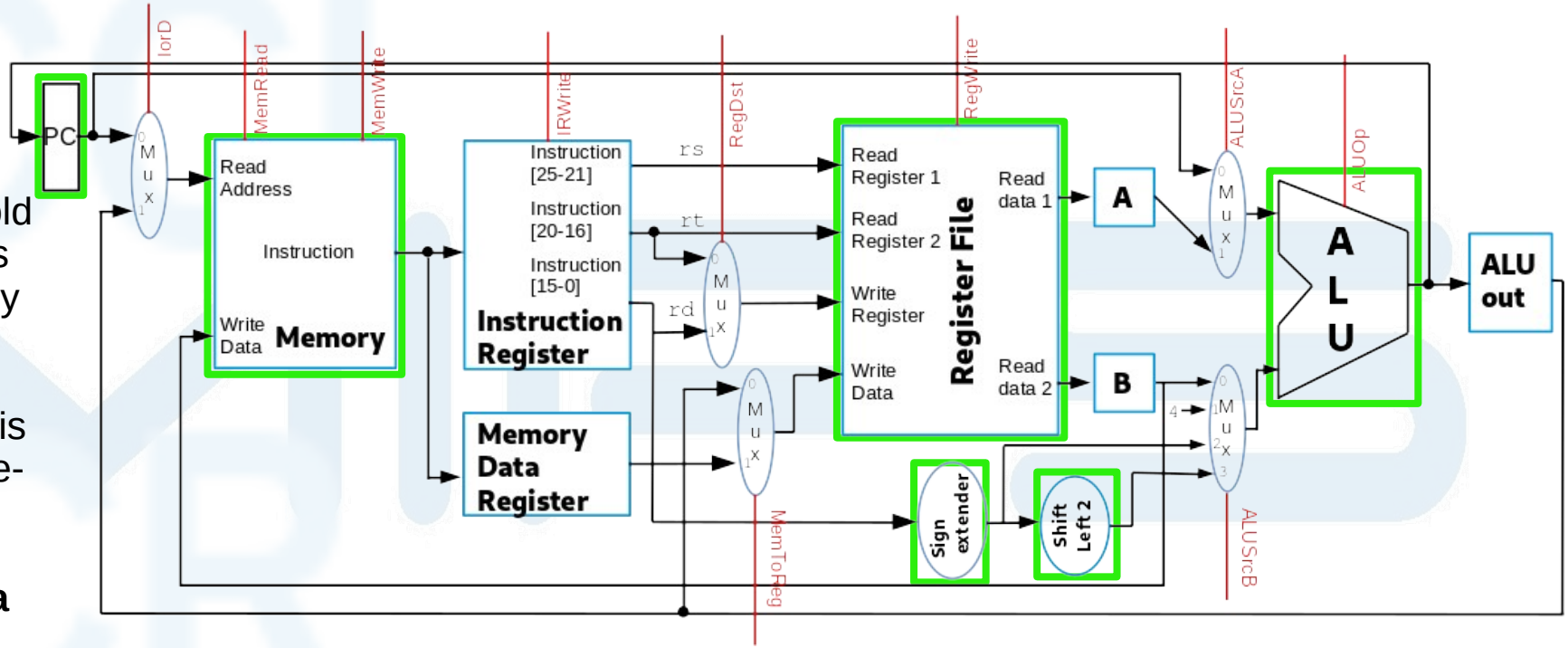


Multi-cycle datapath



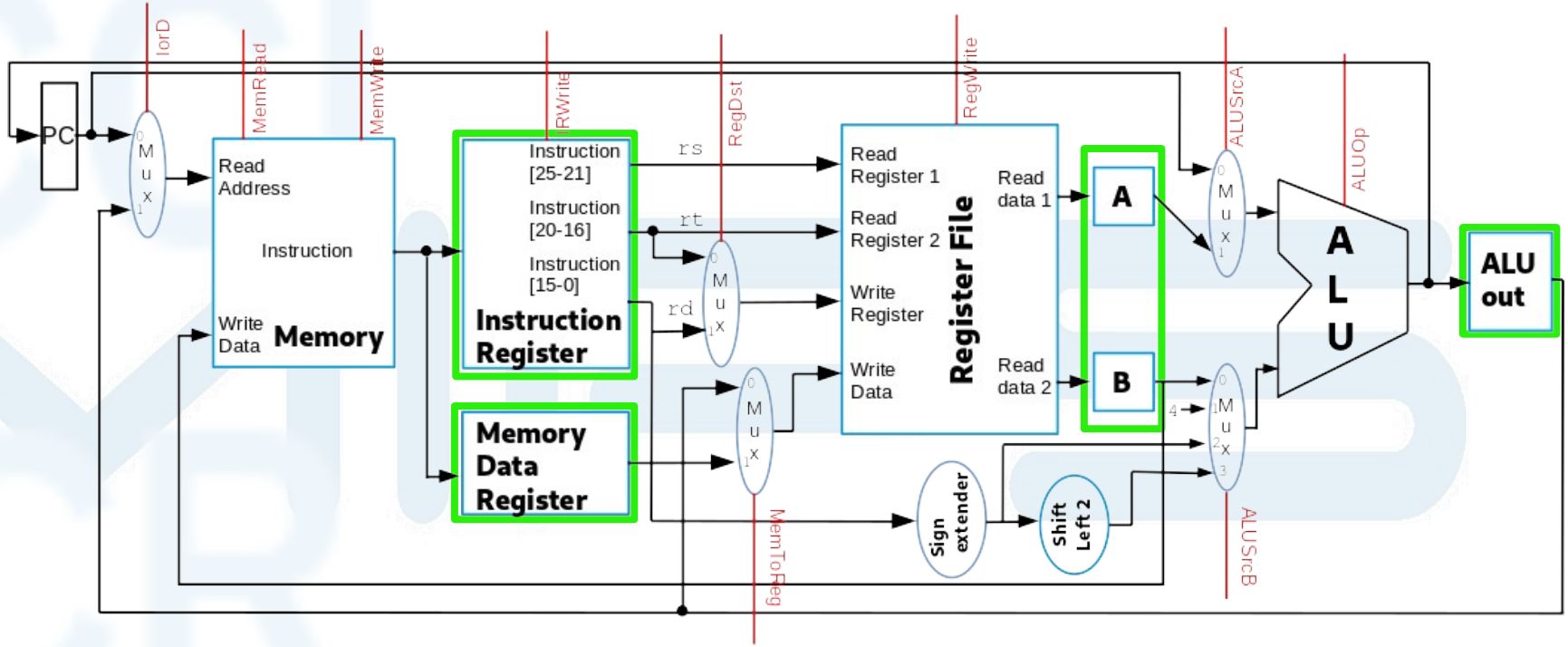
Multi-cycle datapath

These are some old datapath elements that we are already used to. Note, however, that the **Memory** element is now pulling double-duty as both the **Instruction Memory** and **Data Memory** element.



Multi-cycle datapath

New added datapath elements.

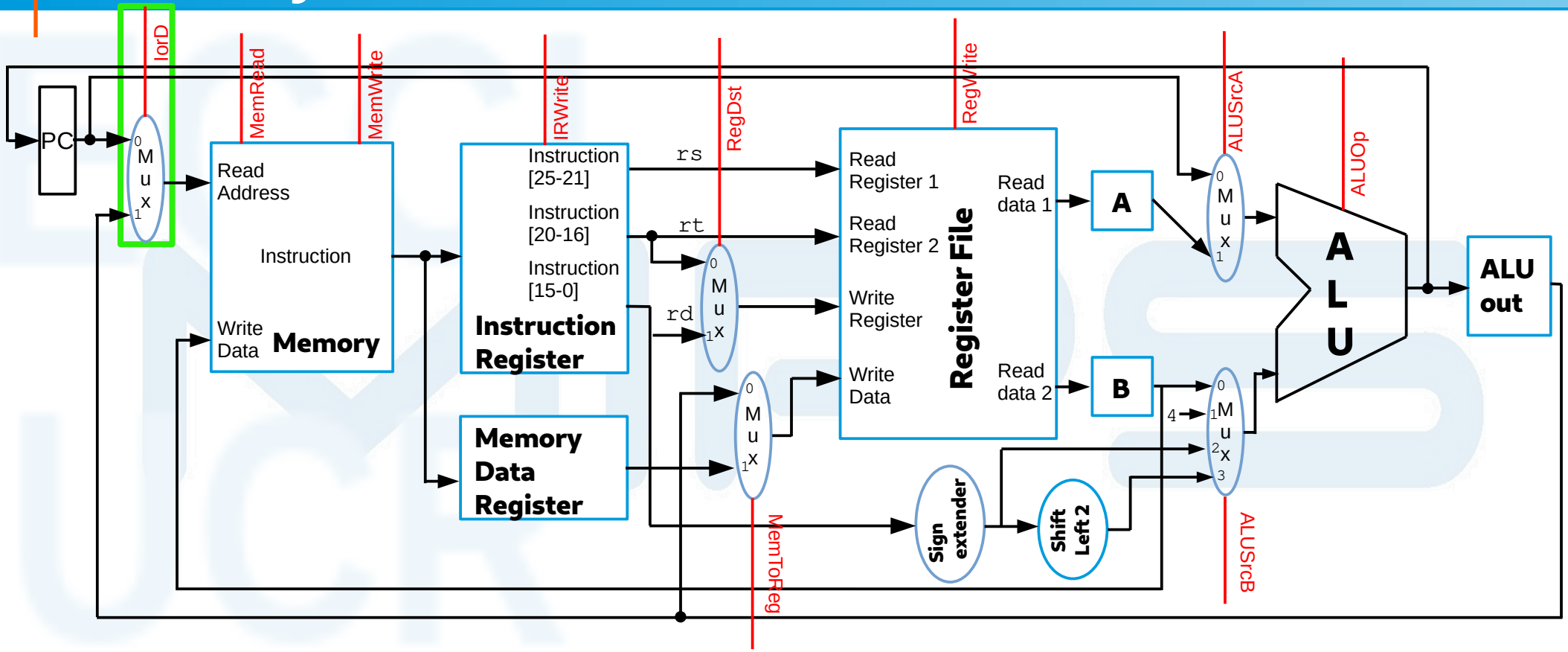


Multi-cycle datapath

New temporary registers:

- **Instruction register** (IR) – holds the instruction after its been pulled from memory
- **Memory data register** (MDR) – temporarily holds data grabbed from memory until the next cycle
- **A** – temporarily holds the contents of read register 1 until the next cycle
- **B** – temporarily holds the contents of read register 2 until the next cycle
- **ALUOut** – temporarily holds the contents of the ALU until the next cycle
- Note: every register is written on every cycle except for the instruction register

Multi-cycle control *IorD*

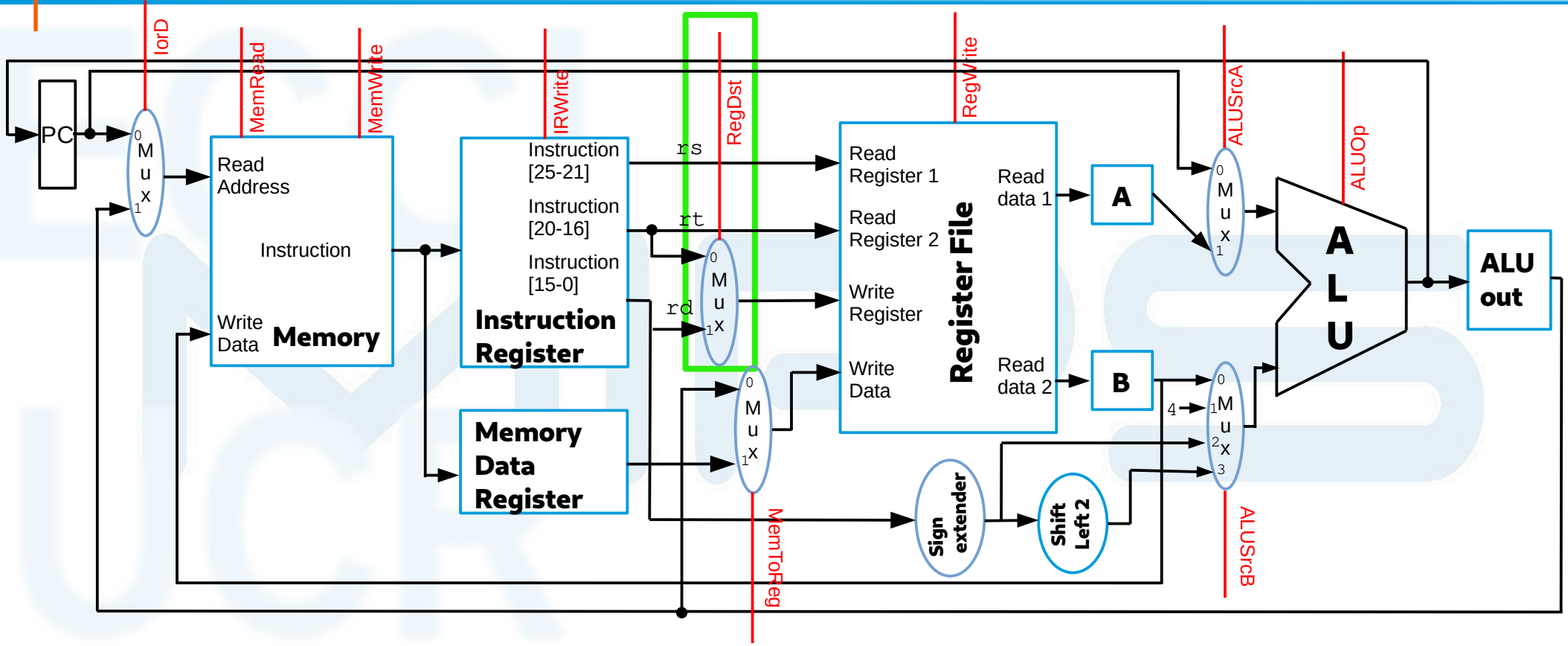


Multi-cycle control

The **IorD** control signal.

- Deasserted (0): the contents of PC is used as the address for the memory unit
- Asserted (1): The contents of ALUOut is used as the address for the memory unit

Multi-cycle control *RegDst*

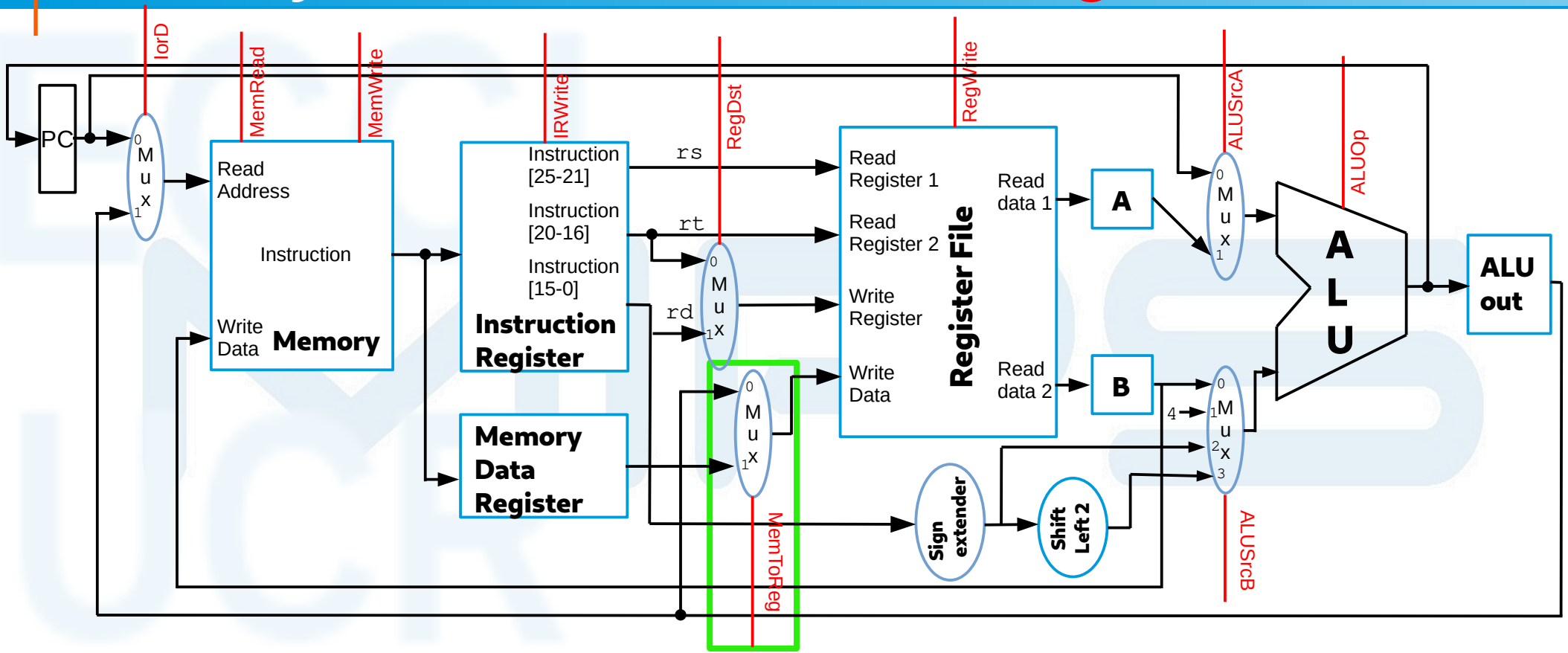


Multi-cycle control

The RegDst control signal:

- Deasserted (0): the register file destination number for the Write register comes from the **rt** field
- Asserted (1): the register file destination number for the Write register comes from the **rd** field

Multi-cycle control *MemToReg*

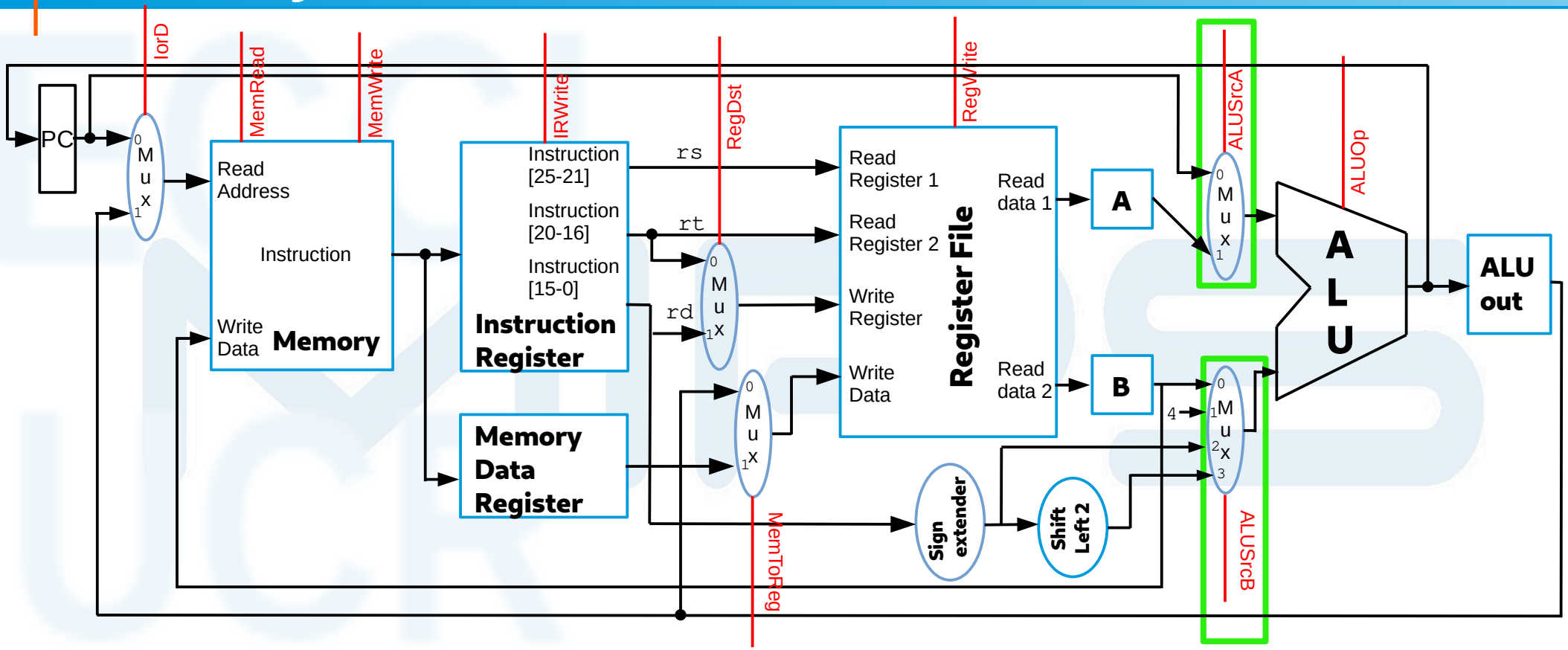


Multi-cycle control

The MemToReg control signal:

- Deasserted (0): the value fed to the register file input comes from ALUout
- Asserted (1): the value fed to the register file input comes from MDR

Multi-cycle control *ALUSrc*

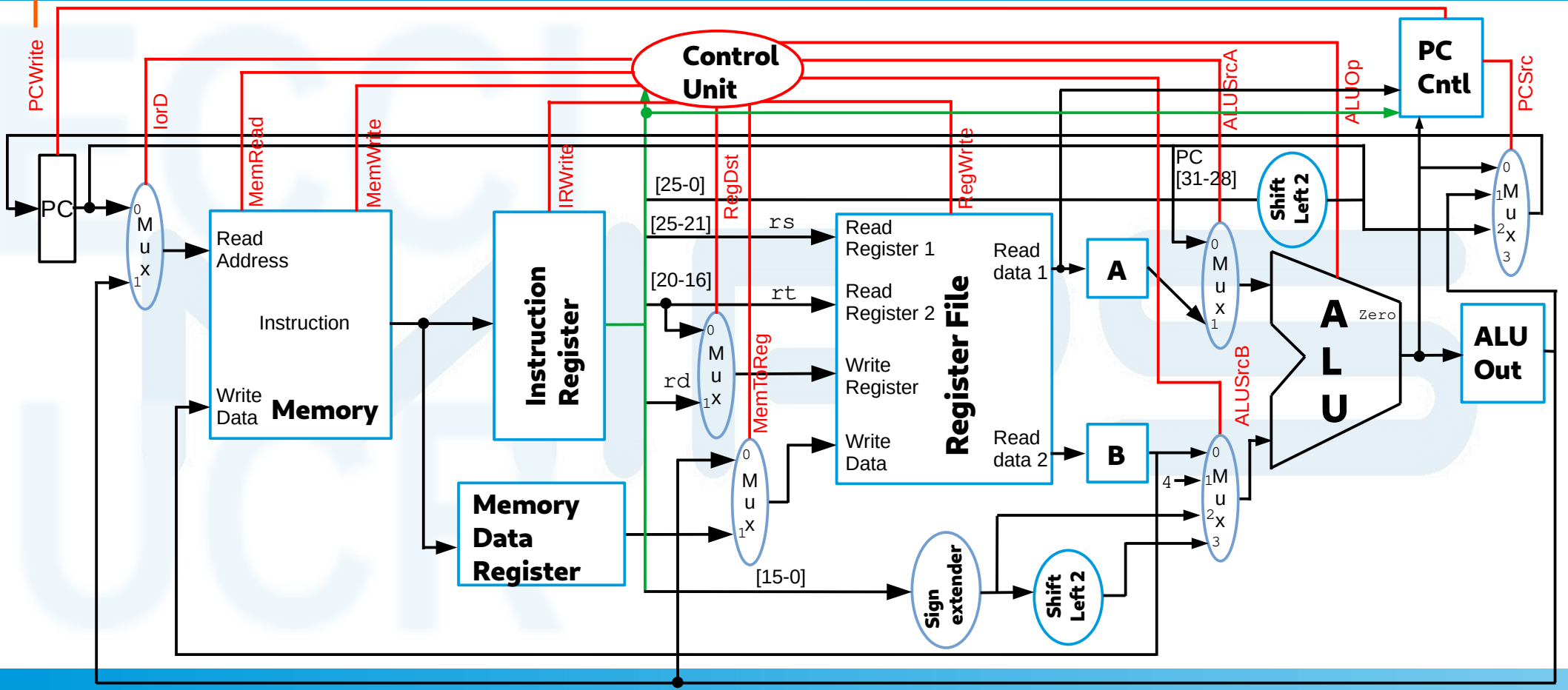


Multi-cycle control

One of the changes we've made is that we're using only a single ALU. We have no dedicated adders on the side. To implement this change, we need to add some multiplexors.

- **ALUSrcA** multiplexor chooses between the contents of PC or the contents of temporary register A as the first operand
- **ALUSrcB** multiplexor chooses between the contents of temporary register B, the constant 4, the immediate field, or the left-shifted immediate field as the second operand

Multi-cycle datapath and control



Multi-cycle 1-bit control signal

1-Bit signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field	The register file destination number for the Write register comes from the rd field
RegWrite	None	Write register is written with the value of the *Write data* input
ALUSrcA	The first ALU operand is PC	The first ALU operand is A register
MemRead	None	Content of memory at the location specified by the Address input is put on the Memory data output
MemWrite	None	Memory contents of the location specified by the Address input is replaced by the value on the Write data input

Multi-cycle 1-bit control signal

1-Bit signal name	Effect when deasserted	Effect when asserted
MemToReg	The value fed to the register file input is ALUout	The value fed to the register file input comes from Memory data register
IorD	None	ALUOut is used to supply the address to the memory unit
IRWrite	None	The output of the memory is written into the Instruction Register (IR)
PCWrite	None	The PC is written; the source is controlled by PC-Source
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active

Multi-cycle 2-bit control signal

2-Bit signal name	Value	Effect
ALUSrcB	00	The second input to ALU comes from the B register
	01	The second input to ALU is 4
	10	The second input to the ALU is the sign-extended, lower 16 bits of the Instruction Register (IR).
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left by 2 bits
PCSource	00	Output of the ALU (PC+4) is sent to the PC for writing
	01	The contents of ALUOut (the 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

Multi-cycle 4-bit control signal ALU

4-Bit signal name	Value	The ALU performs
ALUOp	0000	Add operation
	0001	
	0010	Substrac operation
	0011	Shift right arithmetic
	0100	And operation
	0101	Or operation
	0110	Xor operation
	0111	Nor operation
	1000	Multiplication operation
	1010	Division operation

Multi-cycle datapath and control

Ok, so we already observed that our instructions can be roughly broken up into the following steps:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execution, memory address computation, branch completion, or jump completion
4. Memory access or R-type instruction completion
5. Memory read completion

   Instructions take 3-5 of the steps to complete. The first two are performed identically in all instructions

Instruction fetch step (IF)

First step for all instructions types

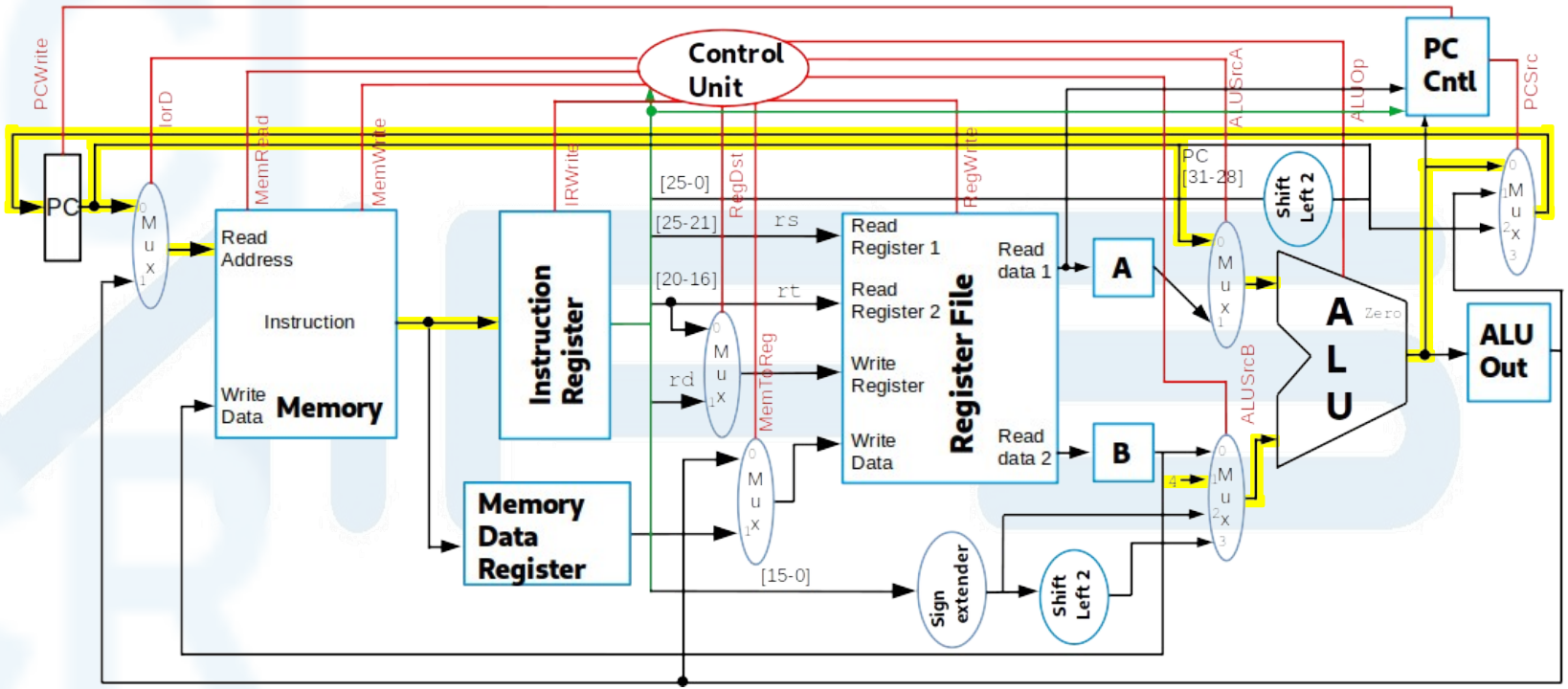
```
IR = Memory[PC];
PC = PC + 4;
```

Operations:

- Send contents of PC to the memory element as the address
- Read instruction from memory
- Write instruction into IR for use in next cycle
- Increment PC by 4

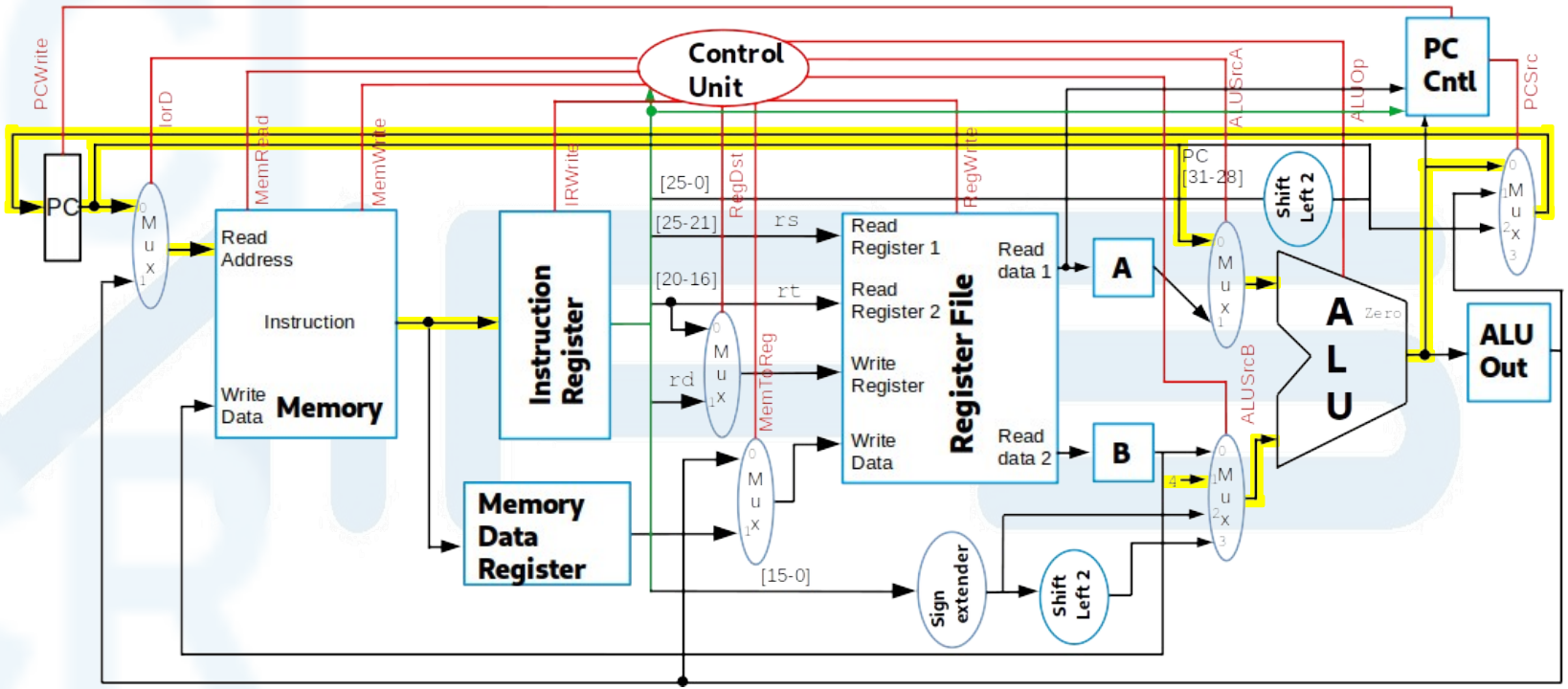
Instruction fetch step (quiz time!!!)

Signal	Value
PCWrite	
IorD	
MemRead	
MemWrite	
IRWrite	
PCSource	
ALUOp	
ALUSrcA	
ALUSrcB	
RegWrite	



Instruction fetch step

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	+
ALUSrcA	0
ALUSrcB	01
RegWrite	0



Instruction decode + register fetch step

Second step for all instructions types

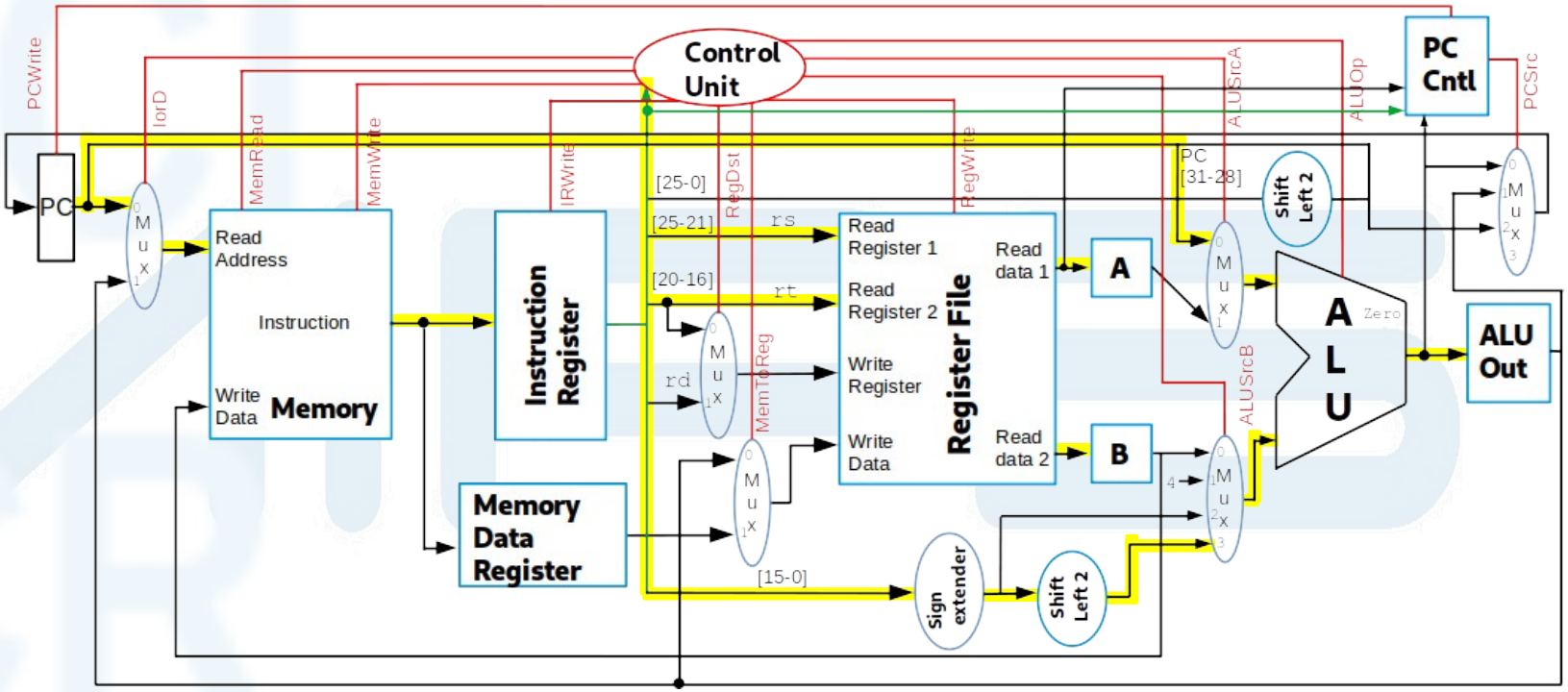
```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

Operations:

- Decode instruction
- Optimistically read registers
- Optimistically compute branch target

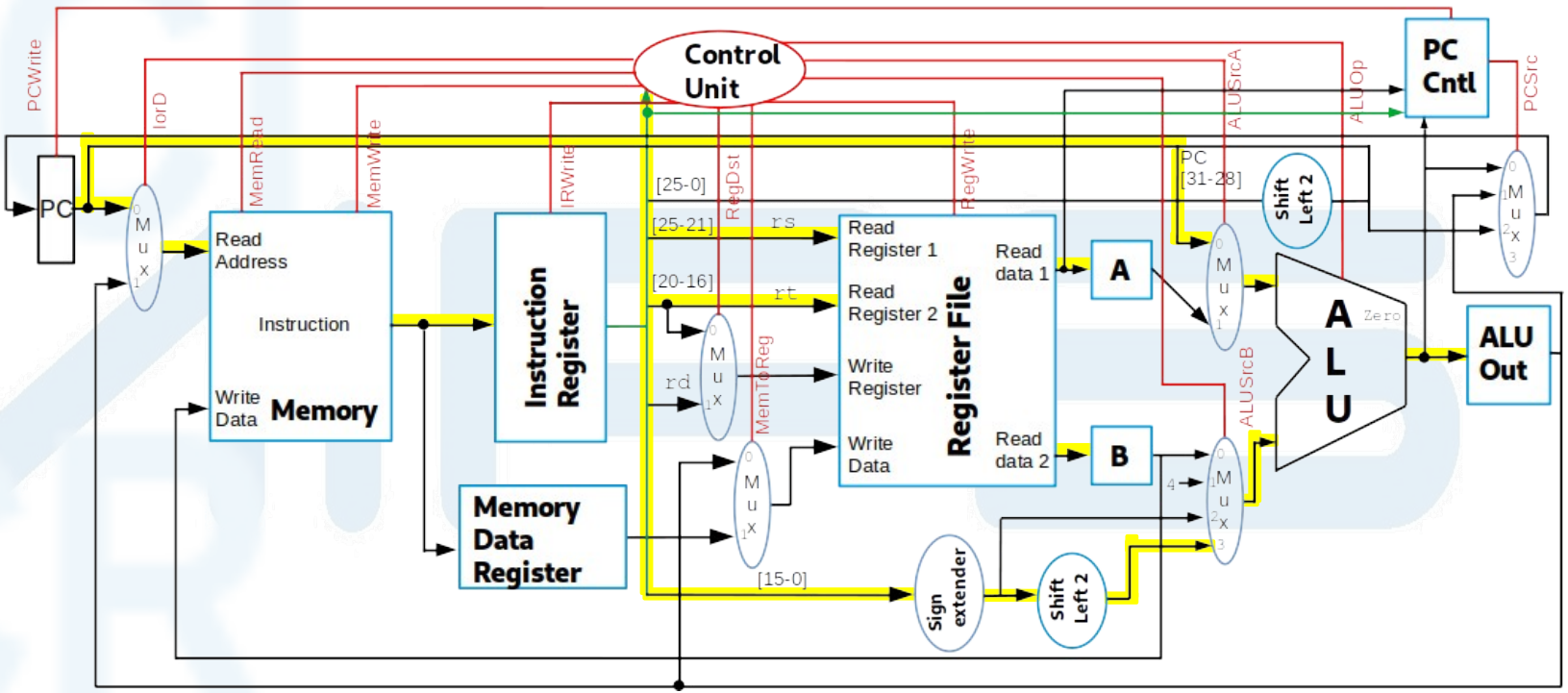
Instruction decode + register fetch step

Signal	Value
ALUOp	
ALUSrcA	
ALUSrcB	



Instruction decode + register fetch step

Signal	Value
ALUOp	0000
ALUSrcA	0
ALUSrcB	11



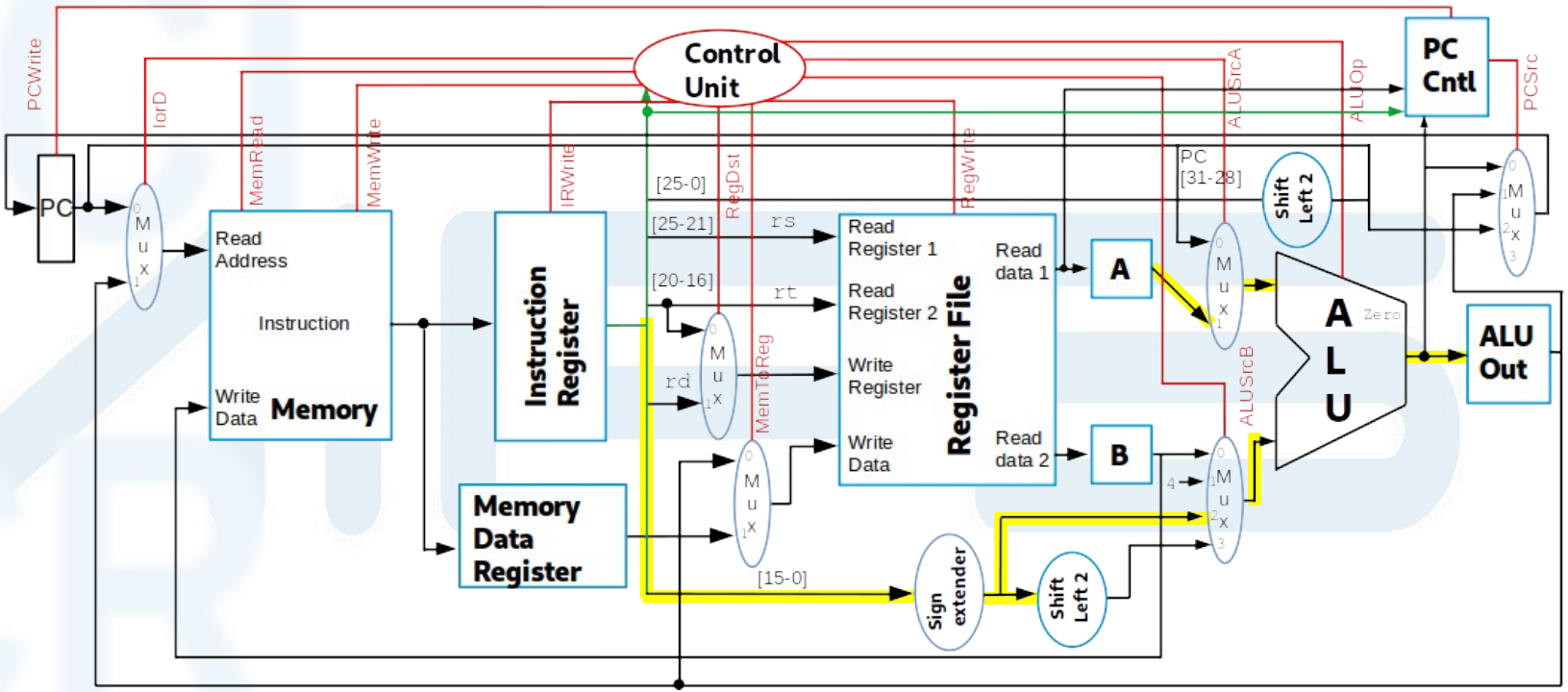
Execution step

Here is where our instructions **diverge**

- Memory reference (load and store):
  - `ALUOut = A + sign-extend(IR[15-0]);`
- Arithmetic-logical operation:
  - `ALUOut = A op B;`
- Branch:
  - `if (A == B) PC = ALUOut;`
- Jump
  - `PC = PC[31-28] || (IR[25-0] << 2);`

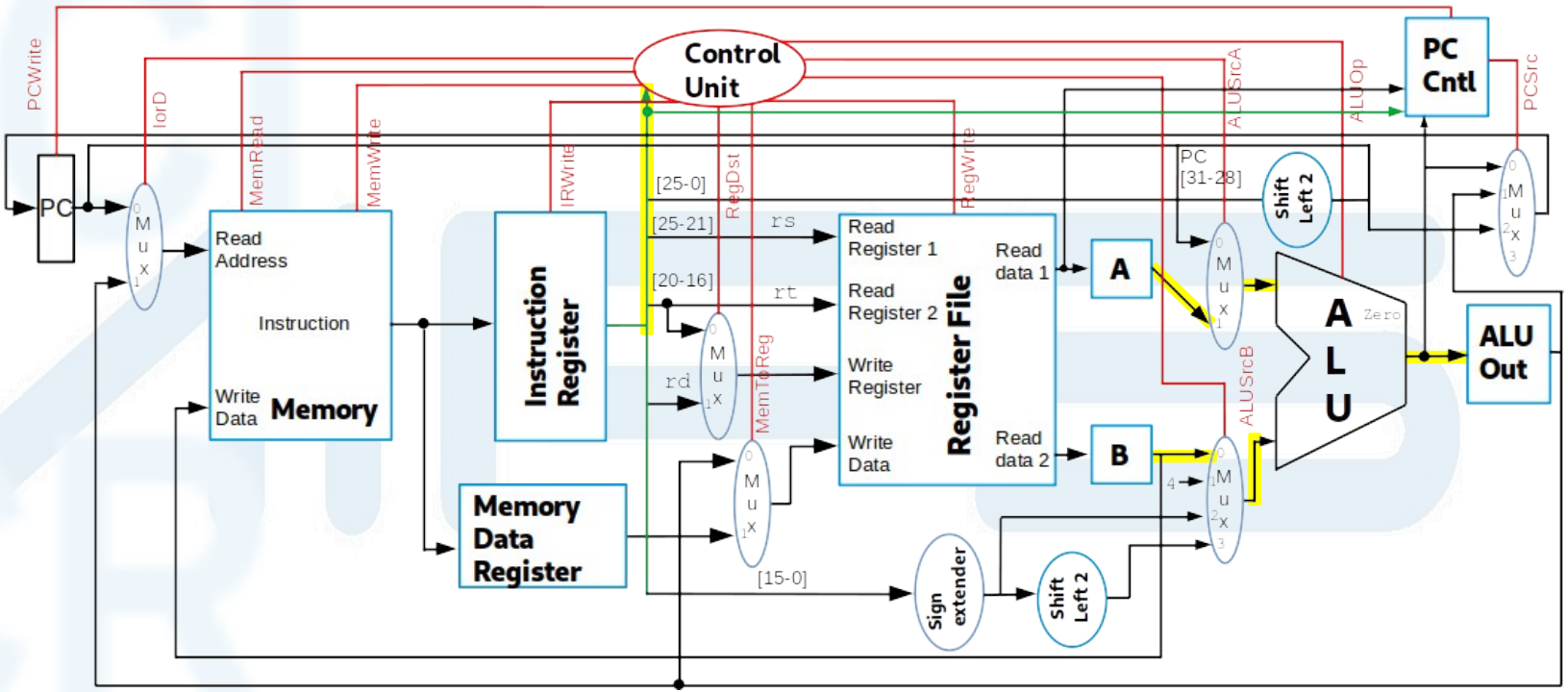
Execution: memory reference

Signal	Value
ALUOp	0000
ALUSrcA	1
ALUSrcB	10



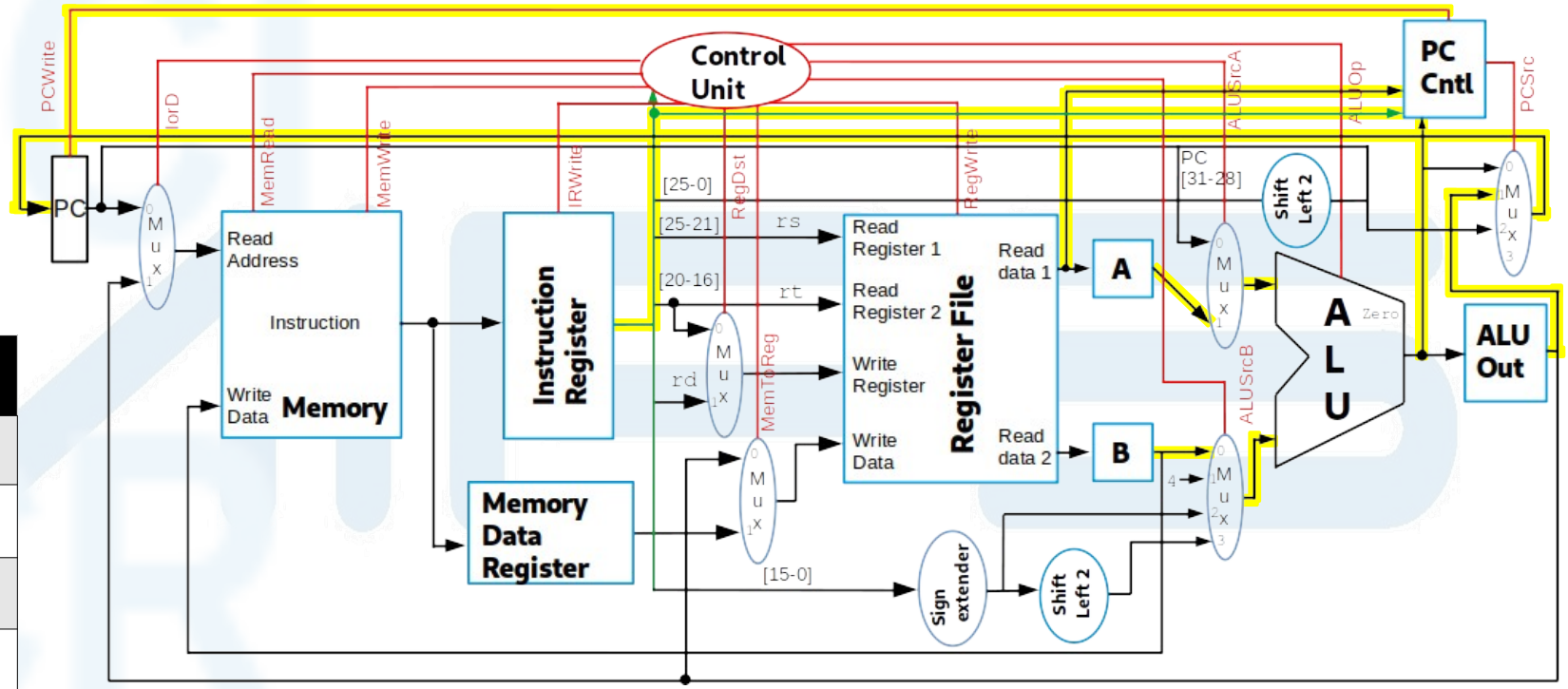
Execution: arithmetic/logic operation

Signal	Value
ALUOp	xxxx
ALUSrcA	1
ALUSrcB	10

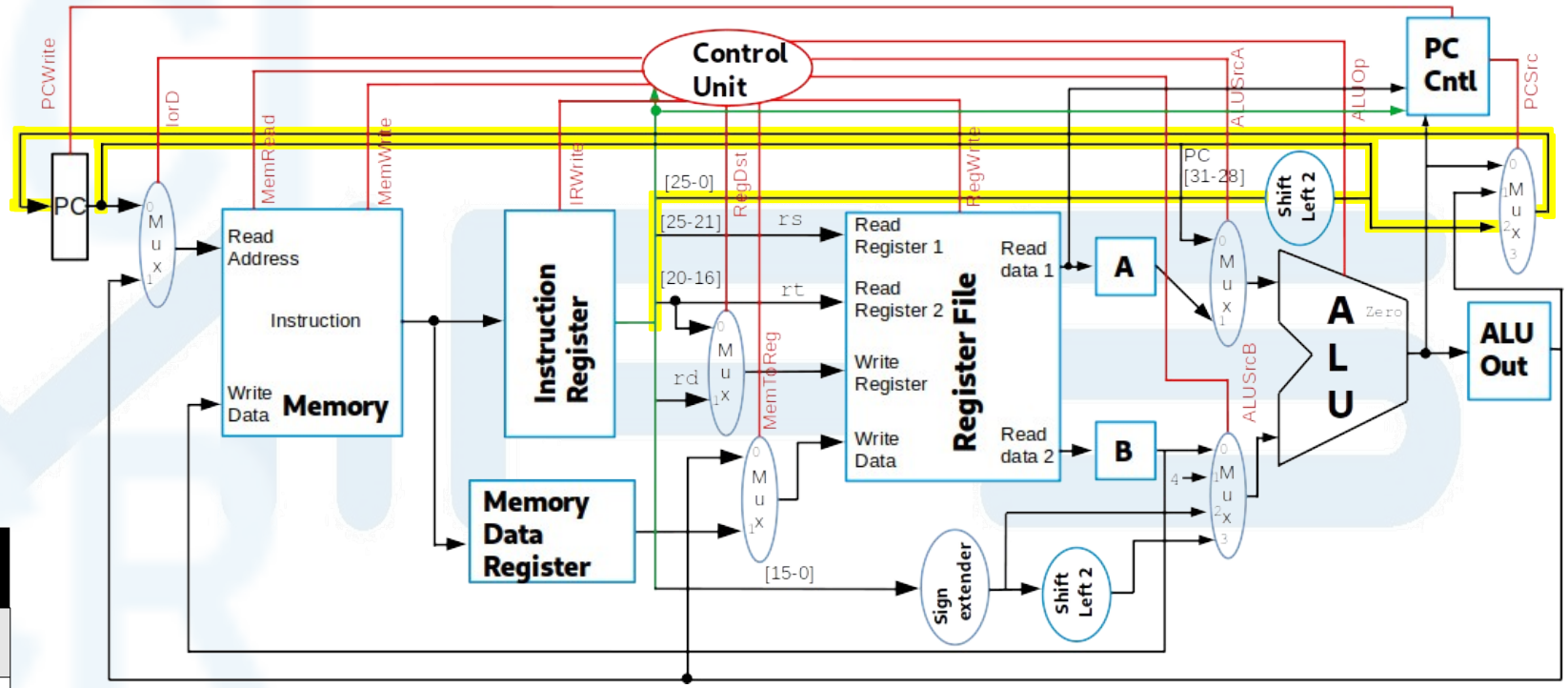


Execution: branch

Signal	Value
ALUOp	0010
ALUSrcA	1
ALUSrcB	00
PCSource	01



Execution: jump



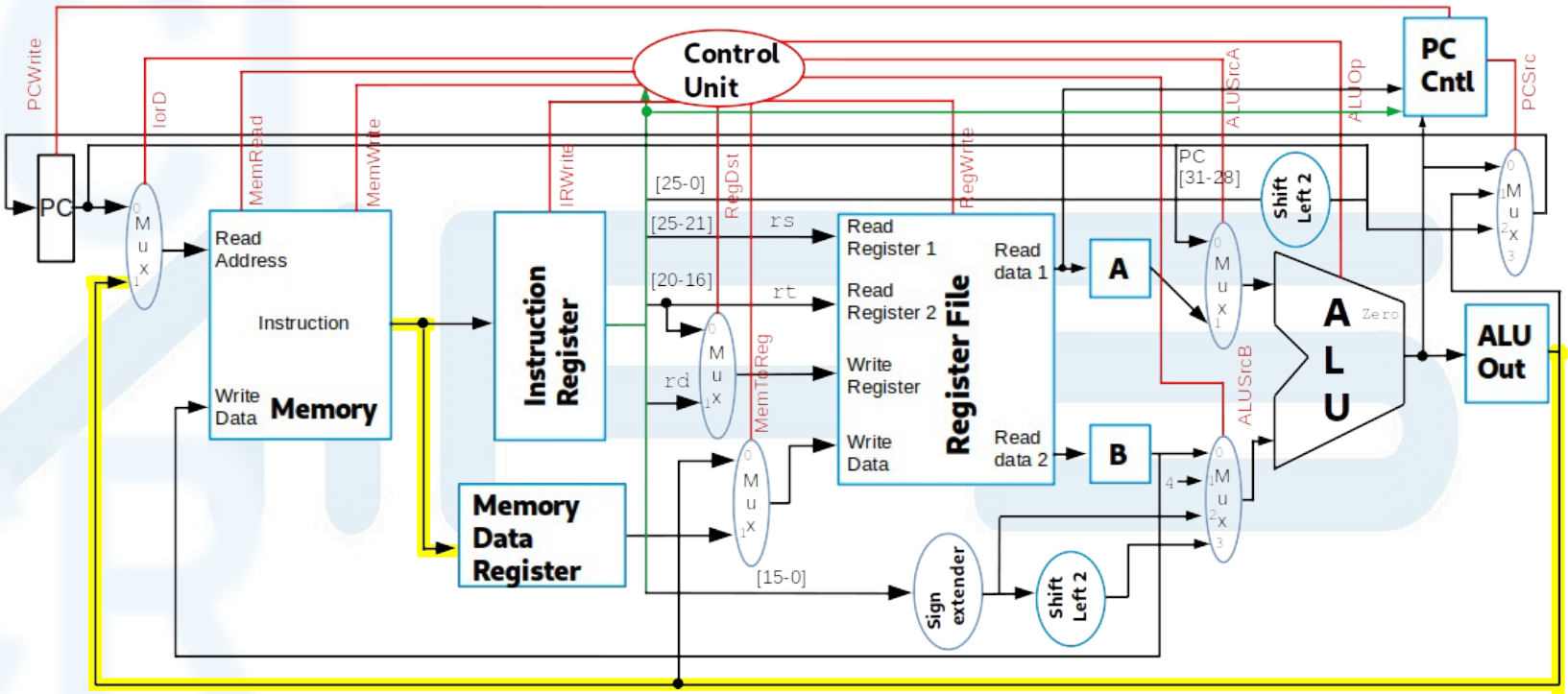
Signal	Value
PCSource	10
PCWrite	1

Memory access/ R-type completion step

- Memory reference:
  - `Load: MDR = Memory[ALUOut];`
  - `Store: Memory[ALUOut] = B;`
- R-type instruction:
  - `Reg[IR[15-11]] = ALUOut;`

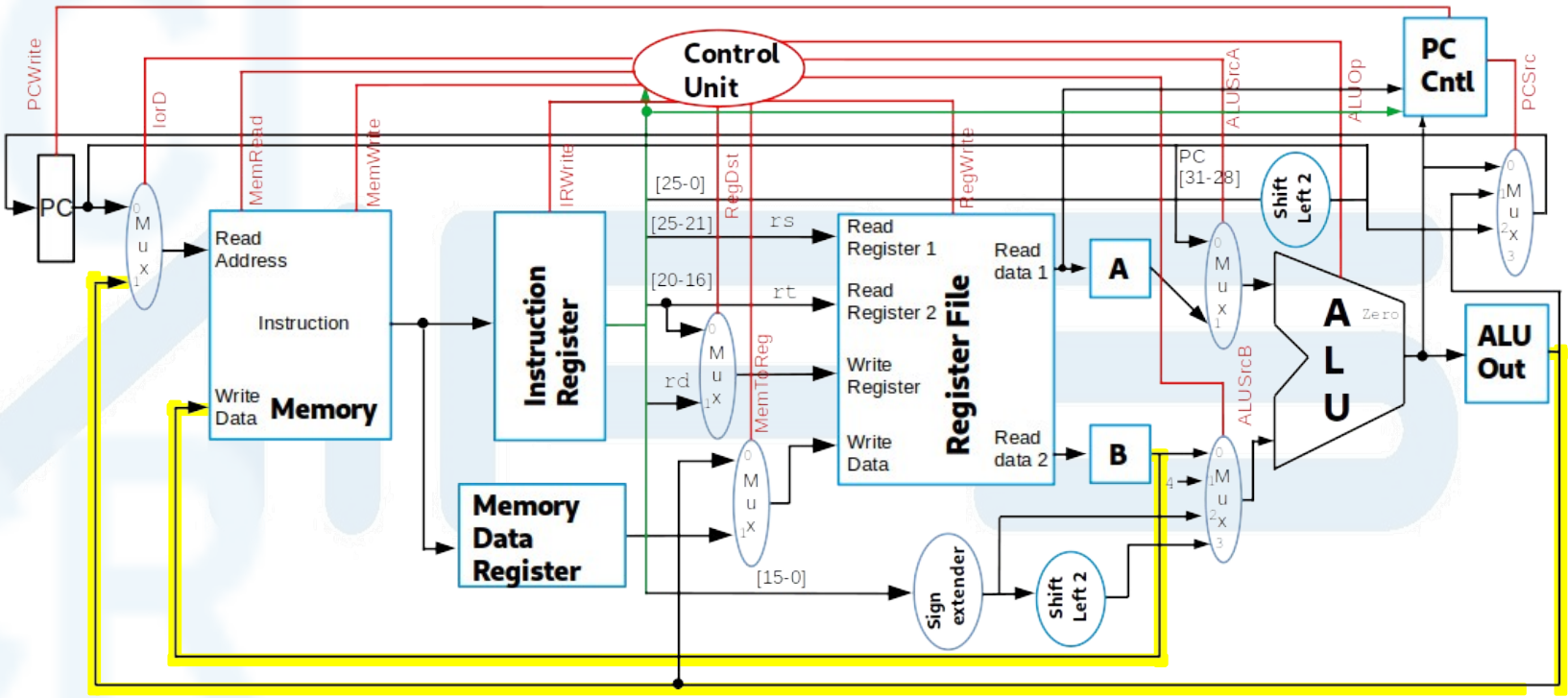
Memory access: LOAD

Signal	Value
MemRead	1
IorD	1
IRWrite	0



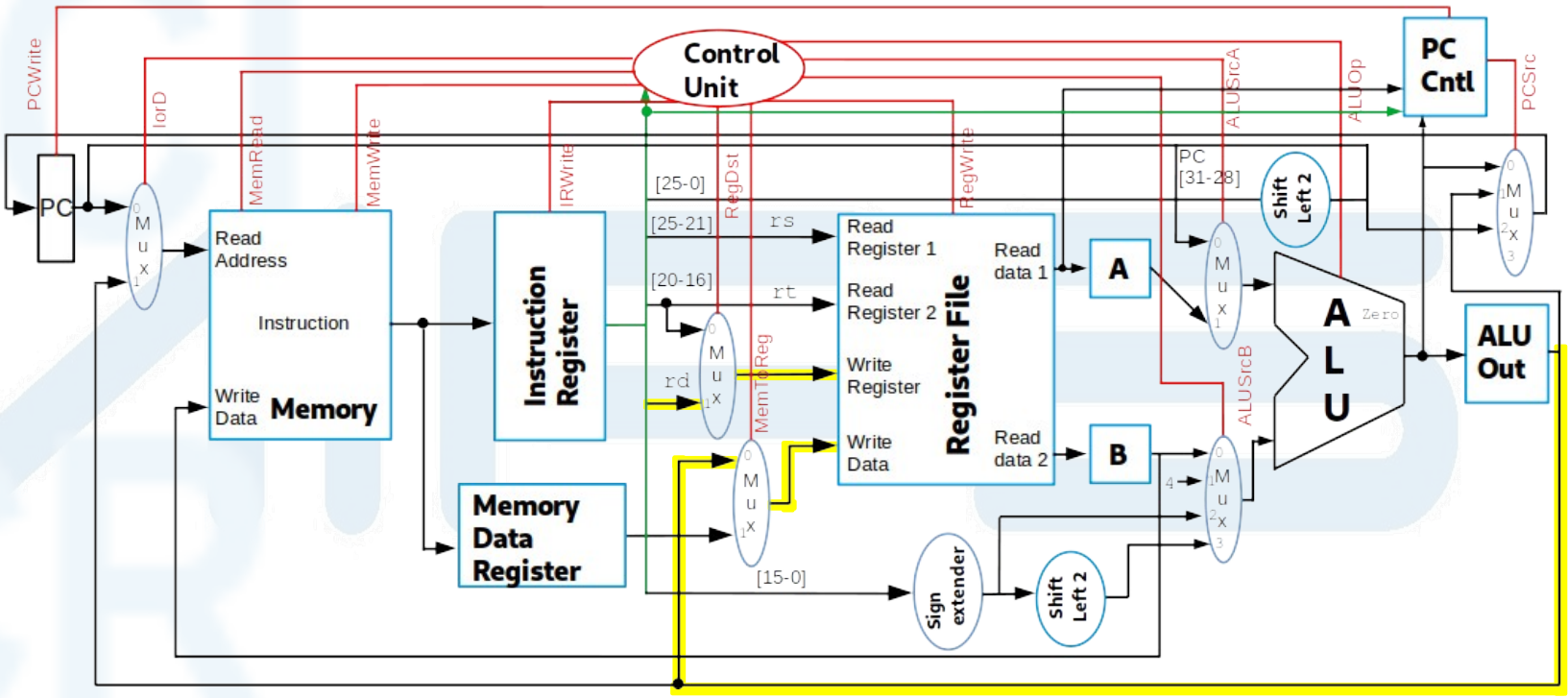
Memory access: STORE

Signal	Value
MemWrite	1
IorD	1



R-type completion

Signal	Value
MemToReg	0
RegWrite	1
RegDst	1

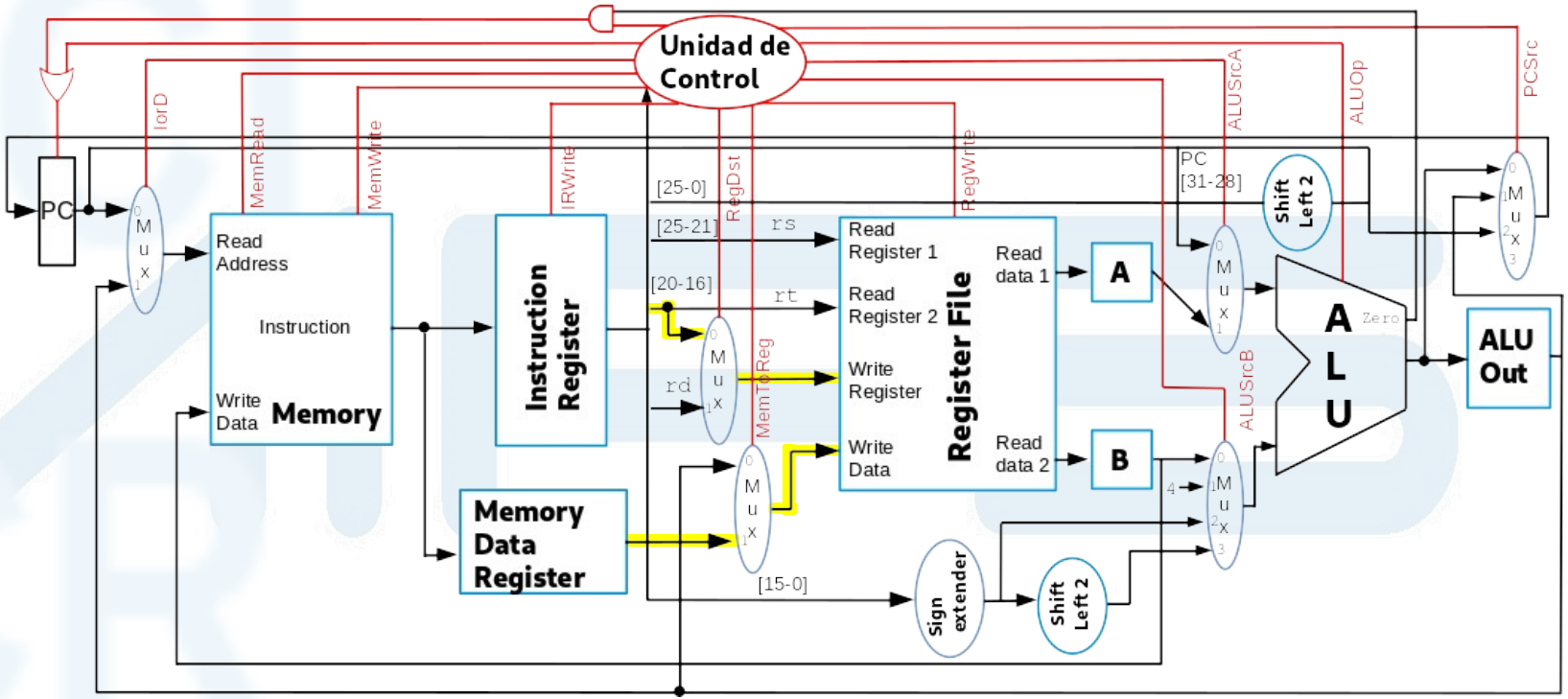


Read completion step

- Load operation:
  - `Reg[IR[20-16]] = MDR;`

Read completion

Signal	Value
MemToReg	1
RegWrite	1
RegDst	0



Multi-cycle datapath and control

So, now we know what the steps are and what happens in each step for each kind of instruction in our mini-MIPS instruction set

To make things clearer, let’s investigate how multi-cycle works for a particular instruction at a time

Execution of R-format **add** instruction

R-format instructions require 4 cycles to complete. Let's imagine that we're executing and *add* instruction

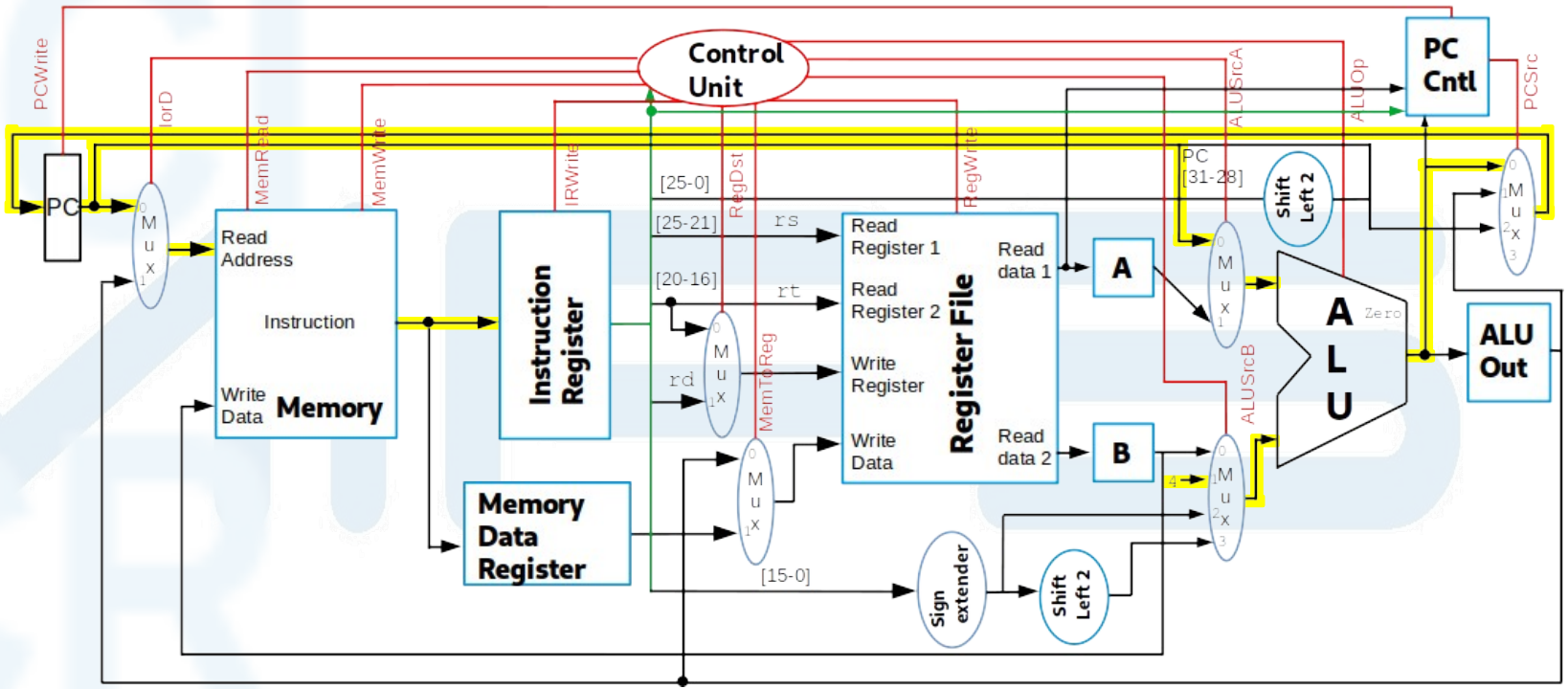
```
add $s0, $s1, $s2
```

which has the following example fields:

Op code	rs	rt	rd	shmat	funct
000000	10001	10010	10000	00000	100000

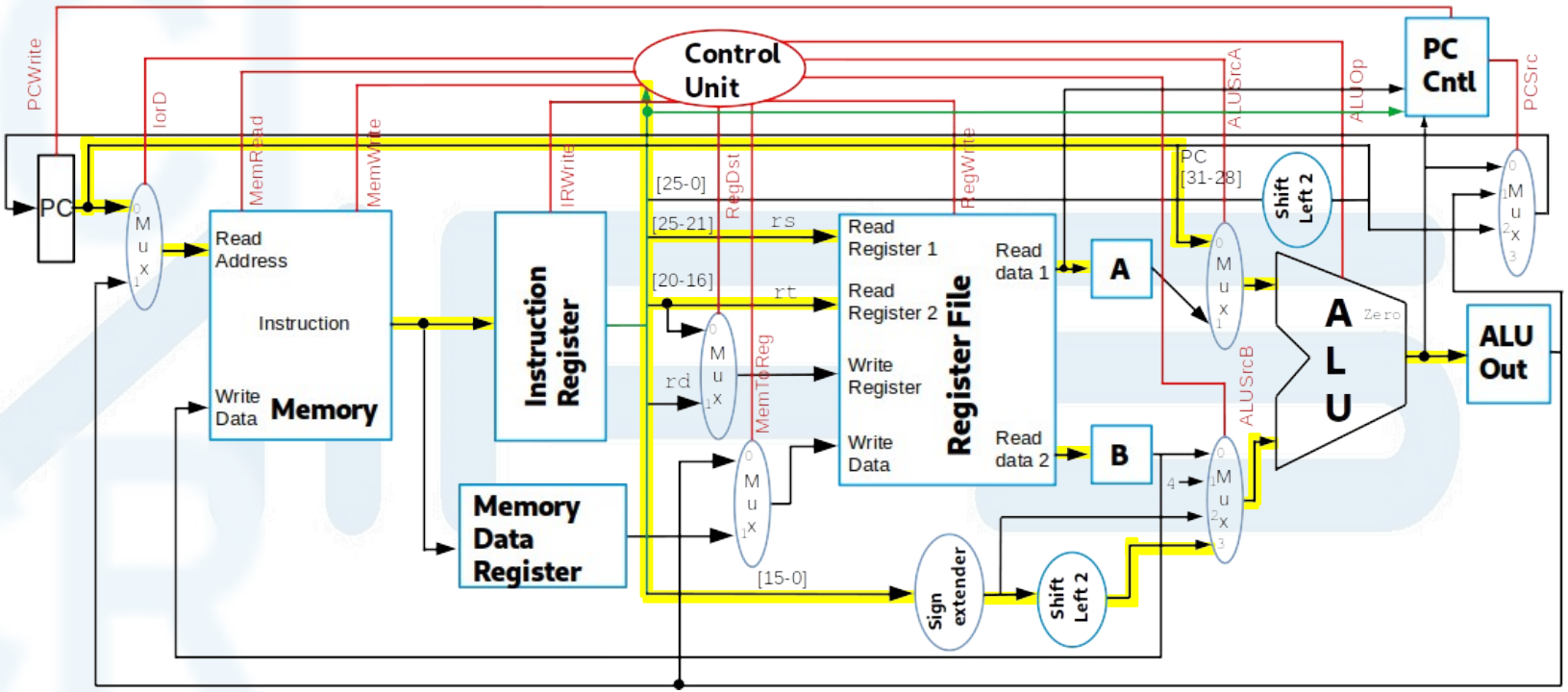
Instruction **add**: step 1

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	0000
ALUSrcA	0
ALUSrcB	01
RegWrite	0



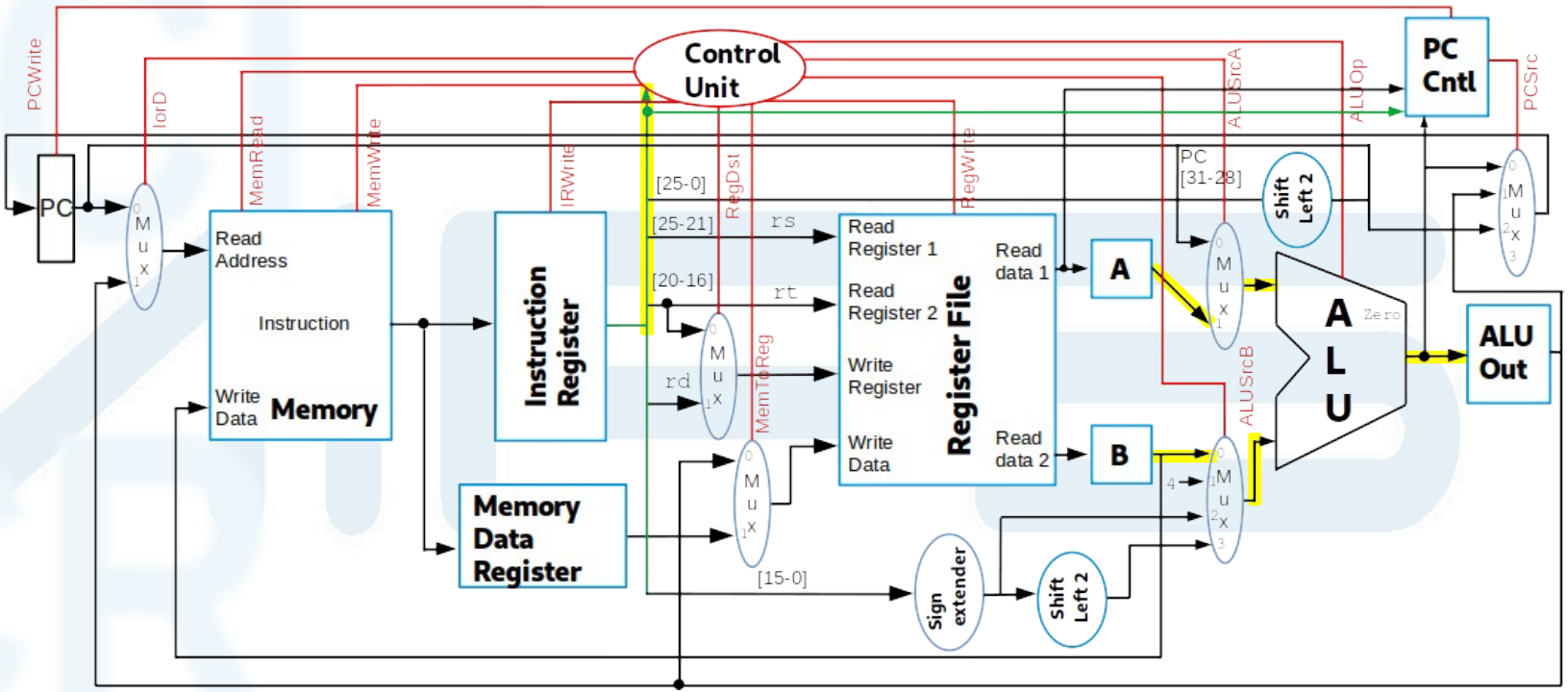
Instruction **add**: step 2

Signal	Value
ALUOp	0000
ALUSrcA	0
ALUSrcB	11



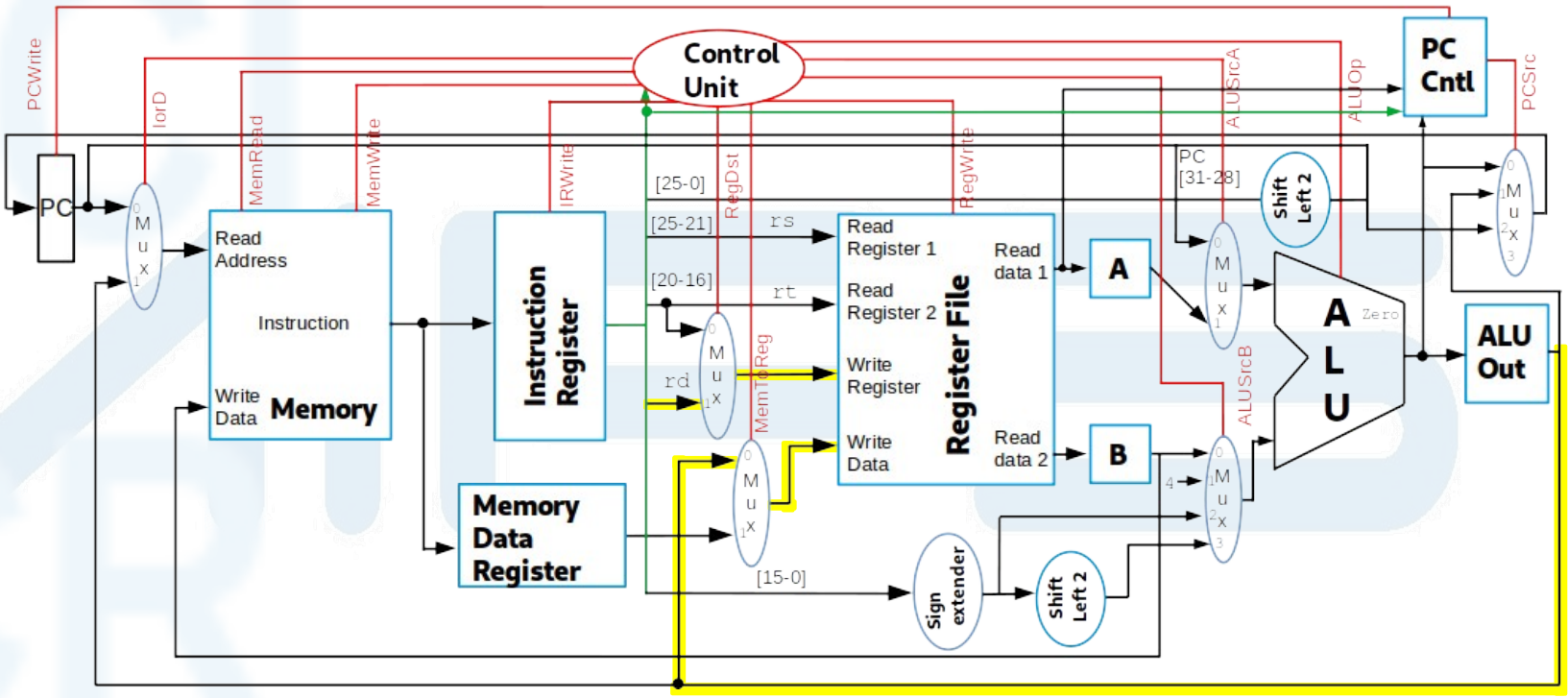
Instruction **add**: step 3

Signal	Value
ALUOp	0000
ALUSrcA	1
ALUSrcB	10



Instruction **add**: step 4

Signal	Value
MemToReg	0
RegWrite	1
RegDst	1



Branch

Branch instructions require 3 cycles to complete. Let's imagine that we're executing a *beq* instruction

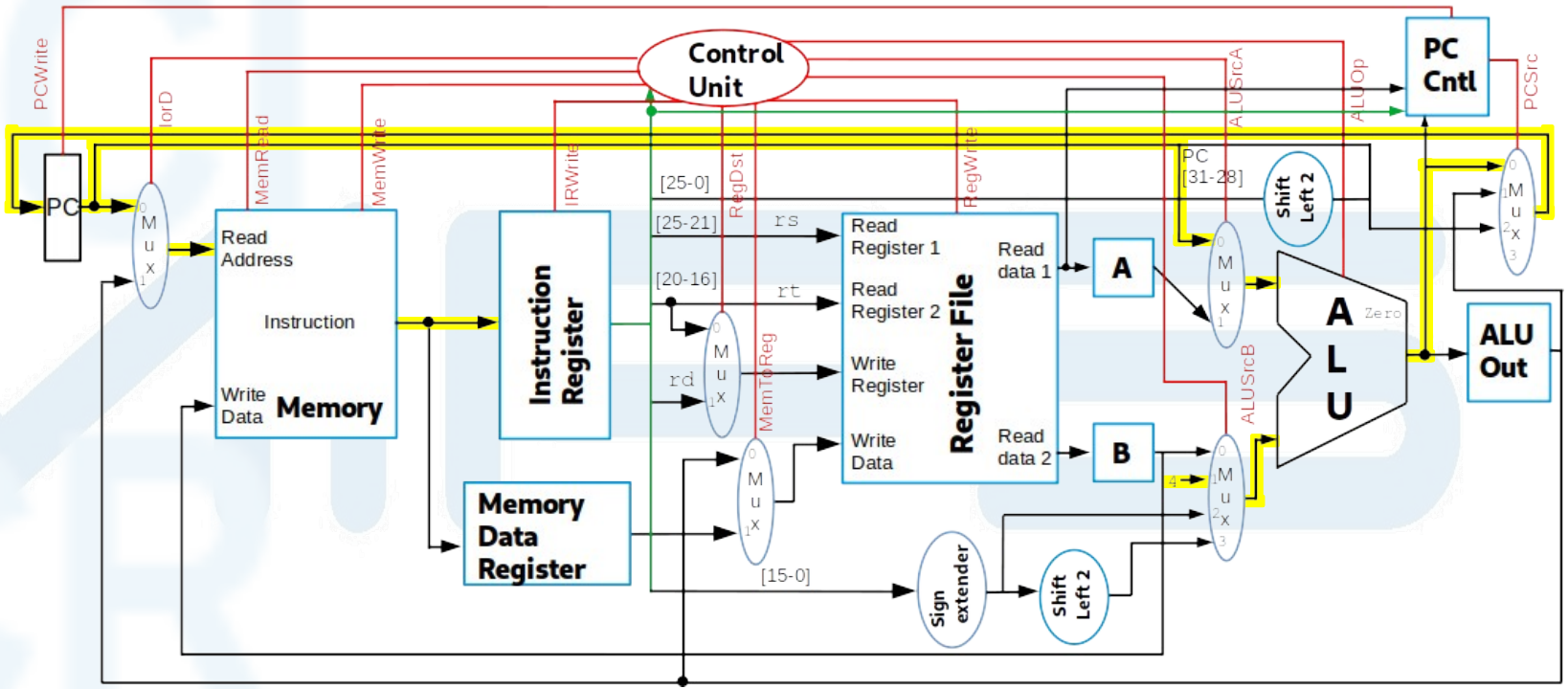
```
beq $s0, $s1, L1
```

- which has the following fields:

Op code	rs	rt	Immediate
000100	10001	10010	XXXXXXXXXXXXXXXX

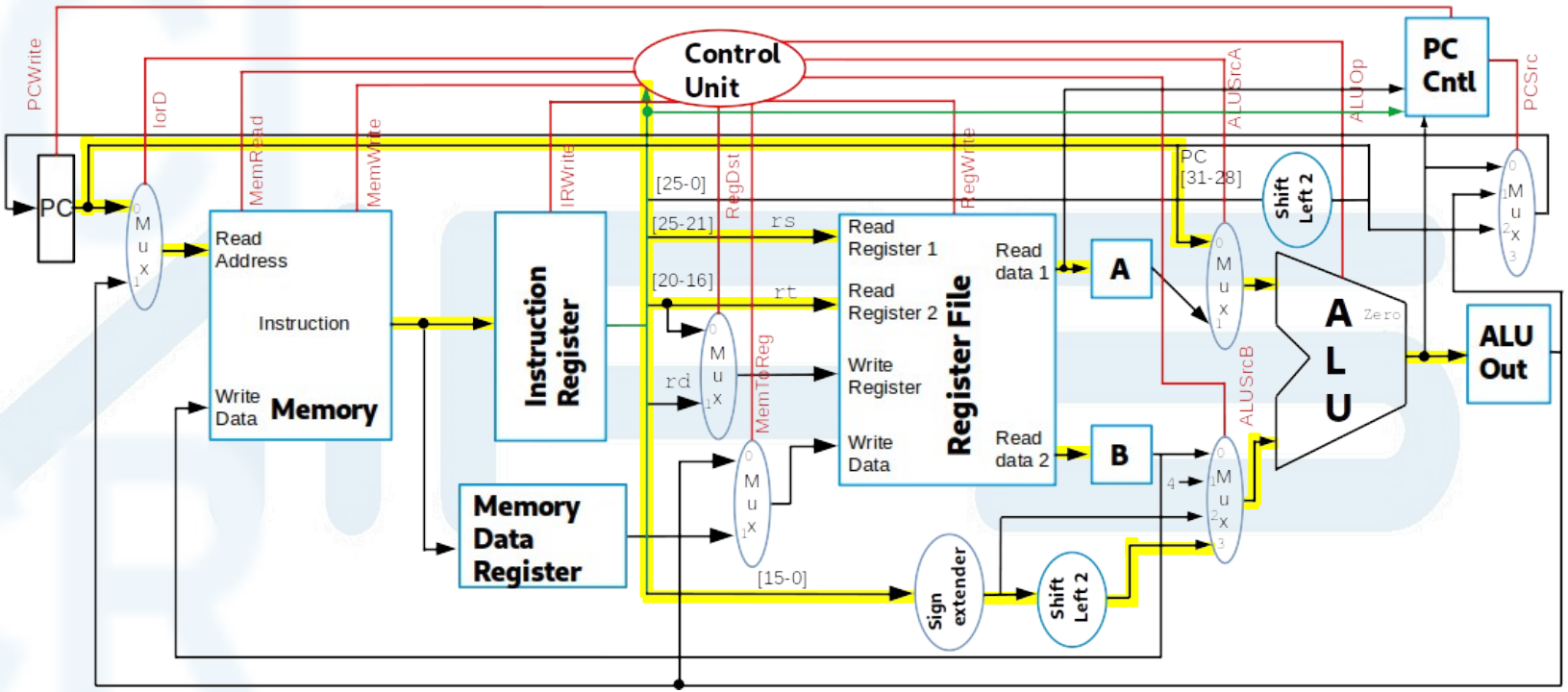
Instruction **branch**: step 1

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	0000
ALUSrcA	0
ALUSrcB	01
RegWrite	0



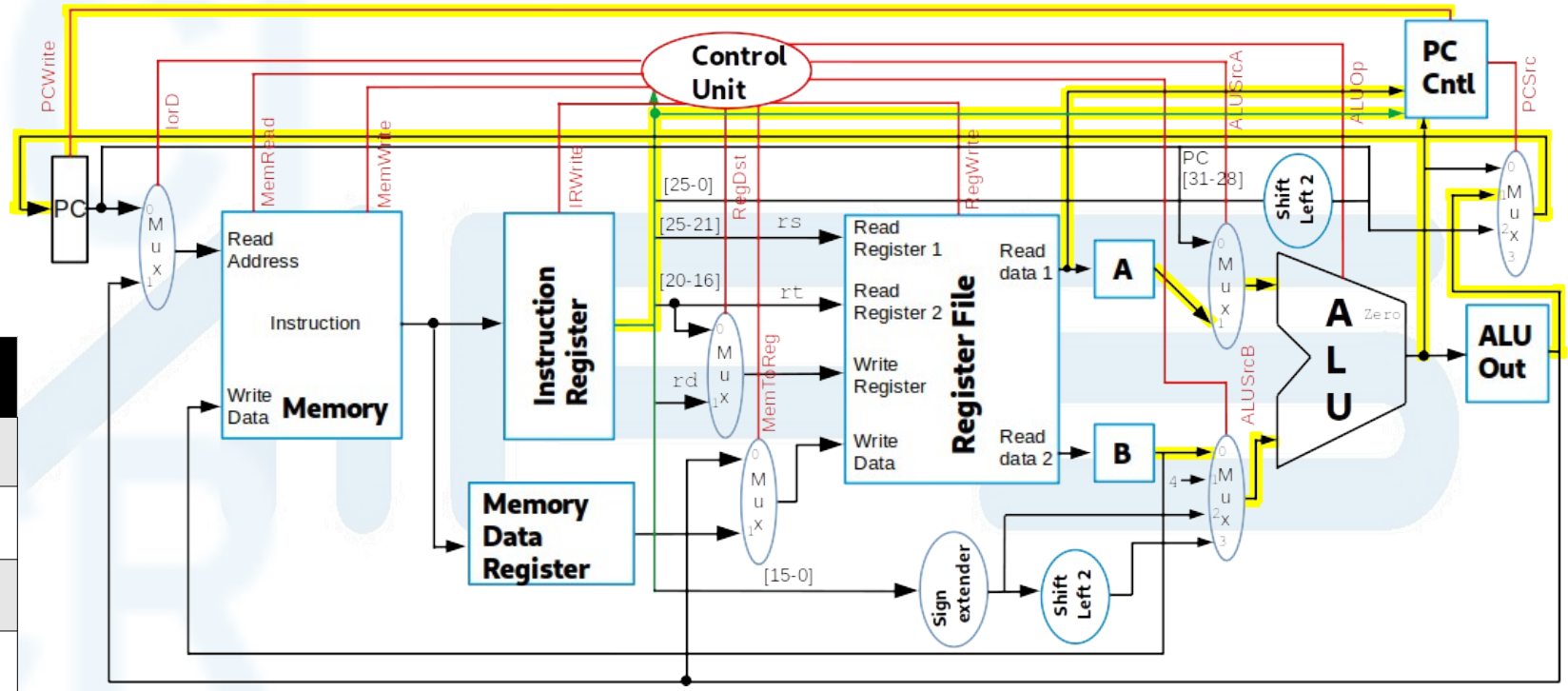
Instruction **branch**: step 2

Signal	Value
ALUOp	0000
ALUSrcA	0
ALUSrcB	11



Instruction **branch**: step 3

Signal	Value
ALUOp	0010
ALUSrcA	1
ALUSrcB	00
PCSource	01
PCWriteCond	1



Store

Store instructions require 4 cycles to complete. Let's imagine that we're executing a *sw* instruction

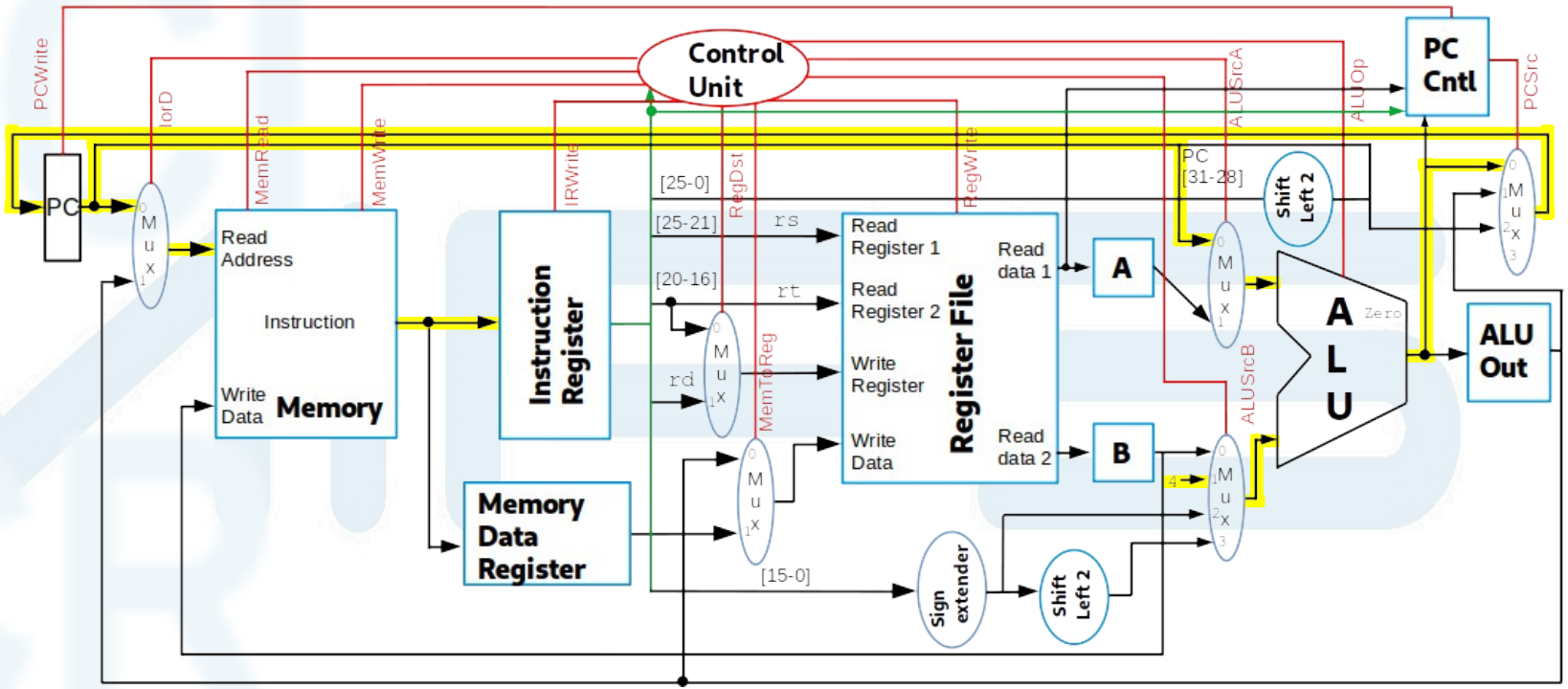
```
sw $rt, immed($rs)
```

- which has the following fields:

Op code	rs	rt	Immediate
101011	10111	10010	XXXXXXXXXXXXXXXX

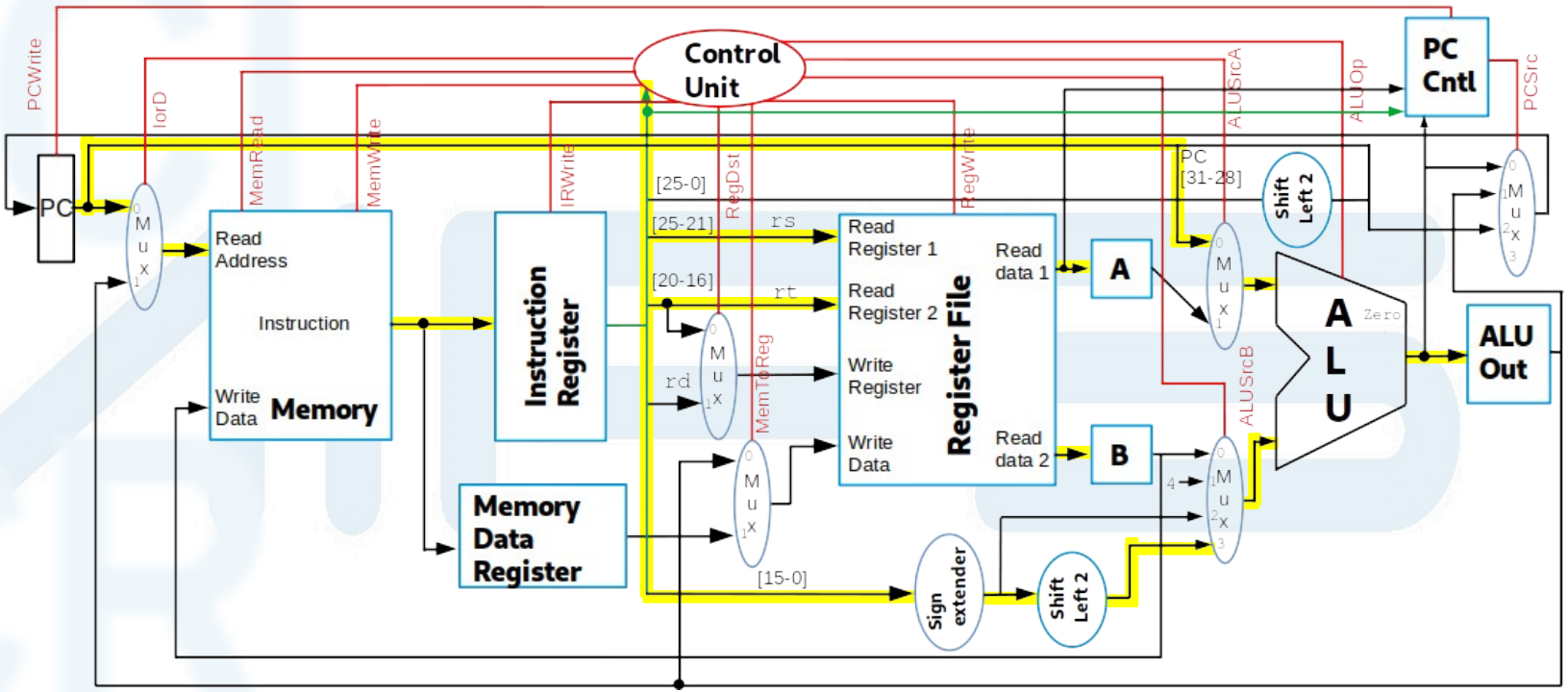
Instruction **store**: step 1

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	0000
ALUSrcA	0
ALUSrcB	01
RegWrite	0



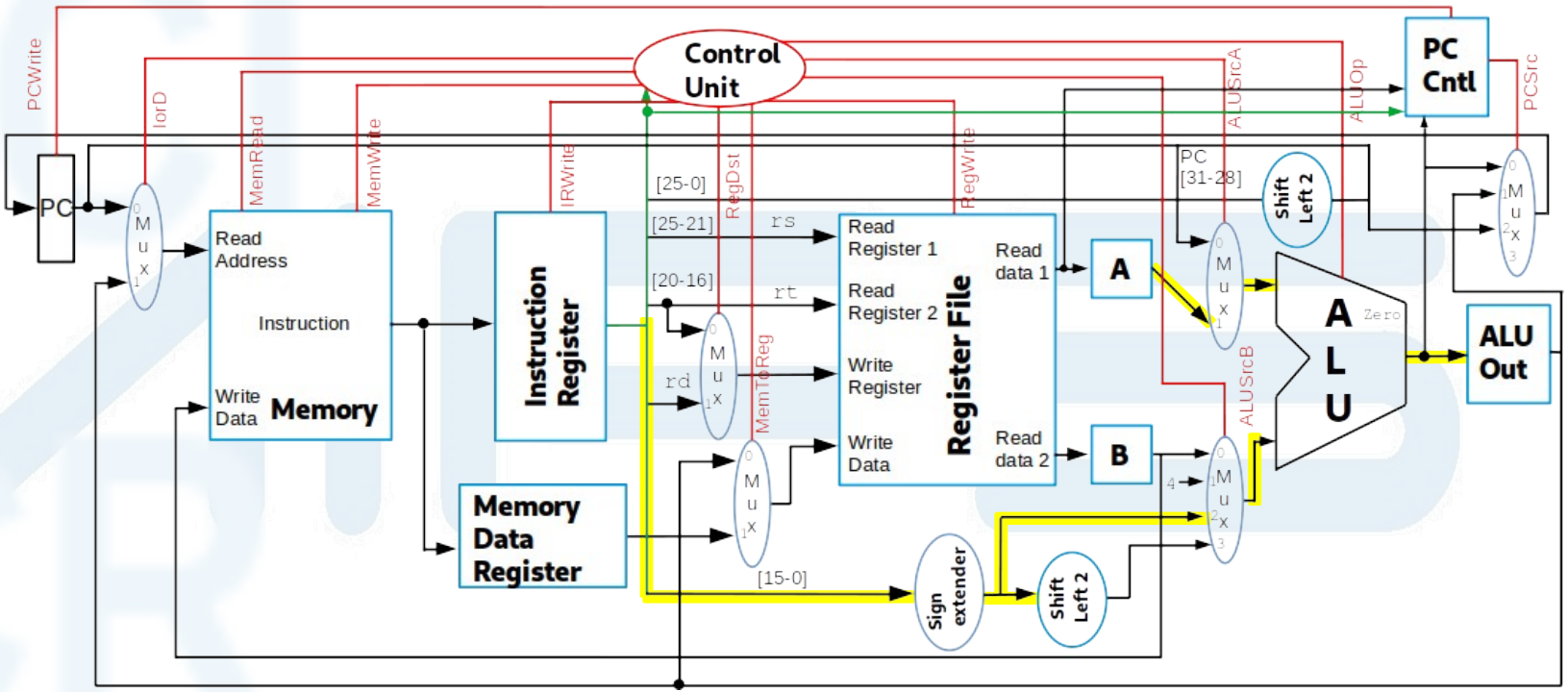
Instruction **store**: step 2

Signal	Value
ALUOp	0000
ALUSrcA	0
ALUSrcB	11



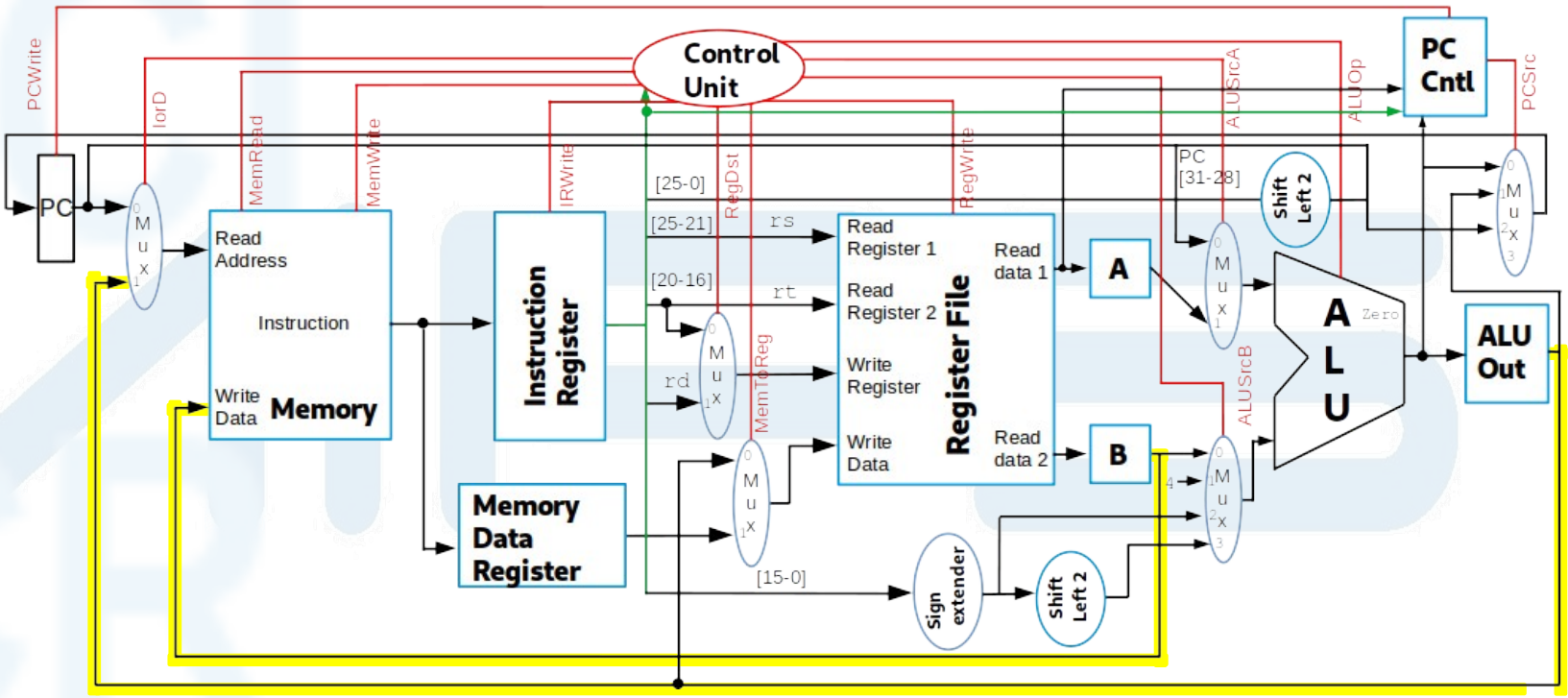
Instruction **store**: step 3

Signal	Value
ALUOp	0000
ALUSrcA	1
ALUSrcB	10



Instruction store: step 4

Signal	Value
MemWrite	1
IorD	1



Load

Load instructions require 5 cycles to complete. Let's imagine that we're executing a *lw* instruction

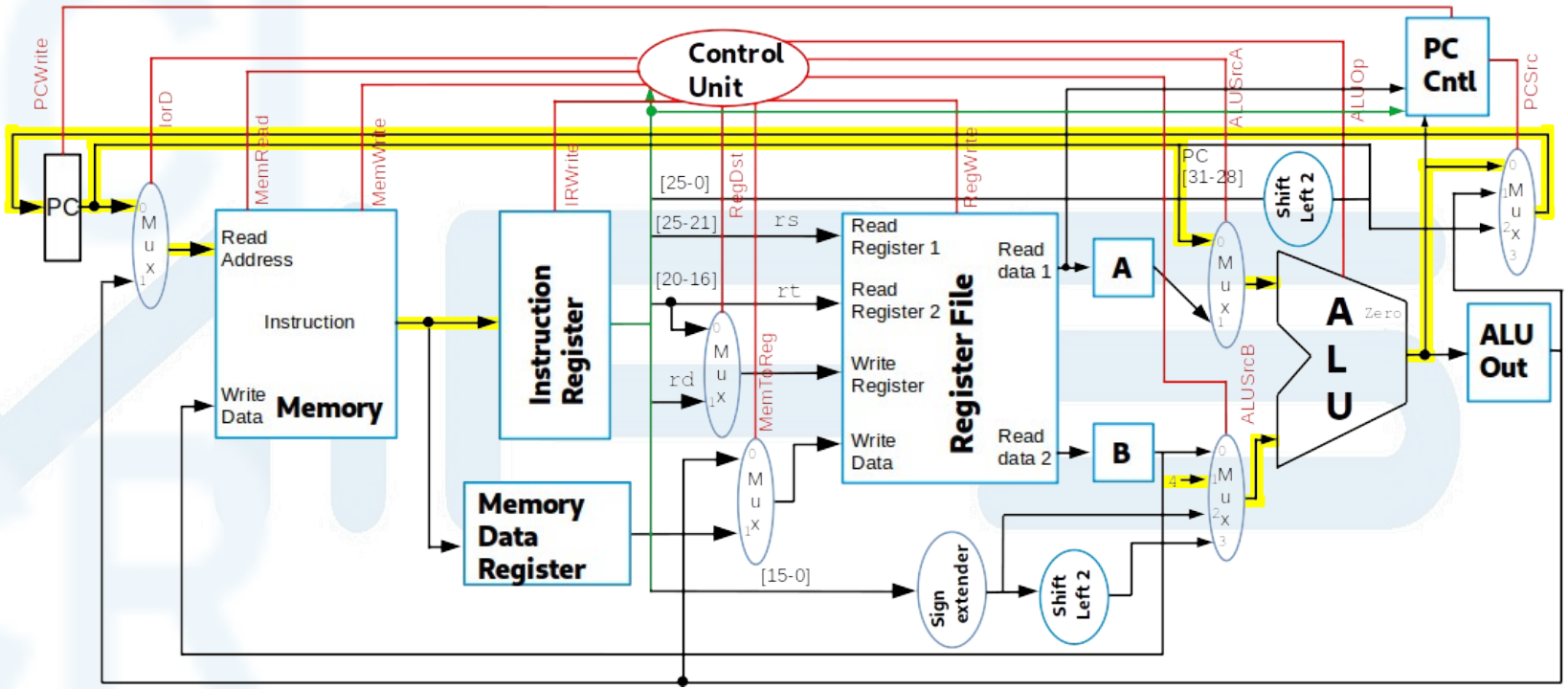
```
lw $rt, immed($rs)
```

- which has the following fields:

Op code	rs	rt	Immediate
100011	10111	10010	XXXXXXXXXXXXXXXX

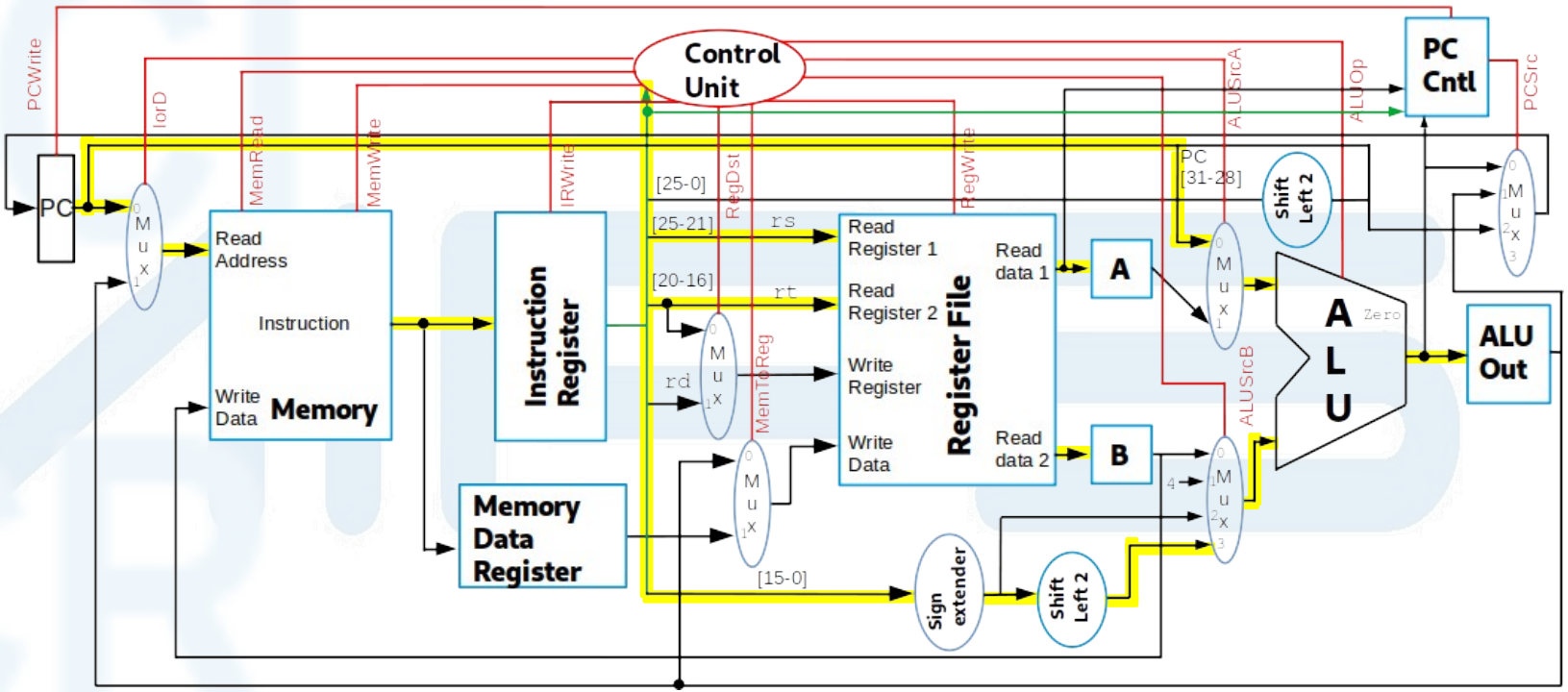
Instruction **load**: step 1

Signal	Value
PCWrite	1
IorD	0
MemRead	1
MemWrite	0
IRWrite	1
PCSource	00
ALUOp	0000
ALUSrcA	0
ALUSrcB	01
RegWrite	0



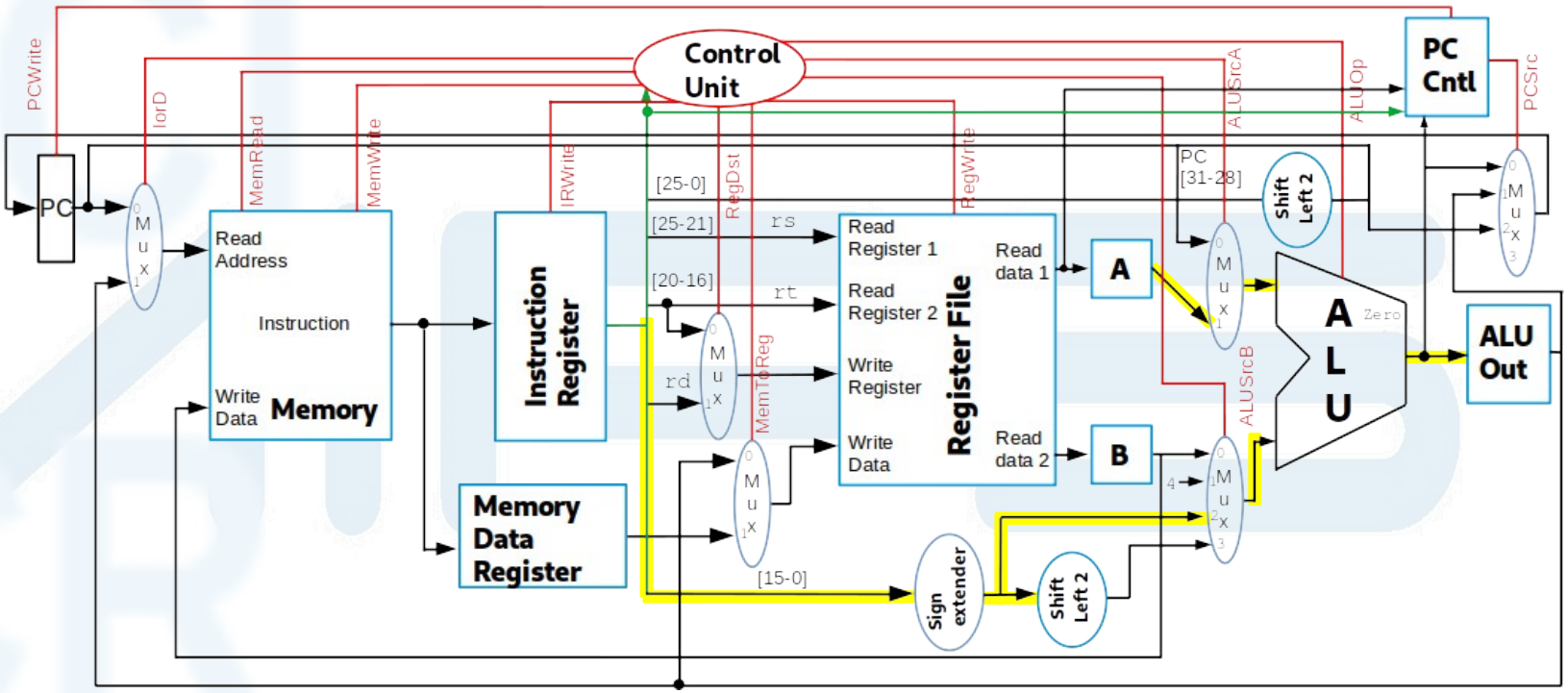
Instruction **load**: step 2

Signal	Value
ALUOp	0000
ALUSrcA	0
ALUSrcB	11



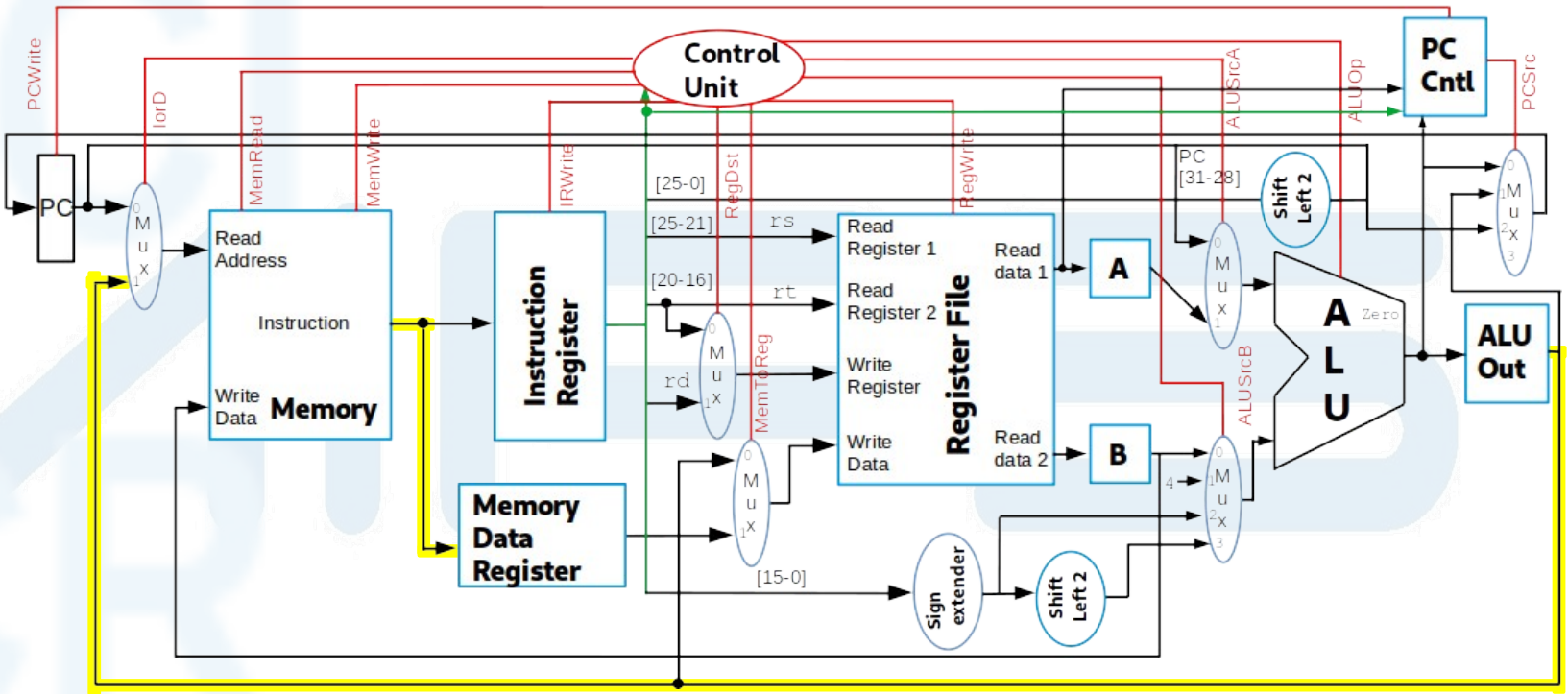
Instruction **load**: step 3

Signal	Value
ALUOp	0000
ALUSrcA	1
ALUSrcB	10



Instruction **load**: step 4

Signal	Value
MemRead	1
IorD	1
IRWrite	0



Instruction **load**: step 5

Signal	Value
MemToReg	1
RegWrite	1
RegDst	0