# LECTURE 5

Single-Cycle
Datapath and Control

# PROCESSORS

In lecture 1, we reminded ourselves that the *datapath* and *control* are the two components that come together to be collectively known as the processor.

- Datapath consists of the functional units of the processor.
  - Elements that hold data.
    - Program Counter, Register File, Instruction Memory, etc.
  - Elements that operate on data.
    - ALU, adders, etc.
  - Buses for transferring data between elements.
- Control commands the datapath regarding when and how to route and operate on data.

# MIPS

To showcase the process of creating a datapath and designing a control, we will be using a subset of the MIPS instruction set. Our available instructions include:

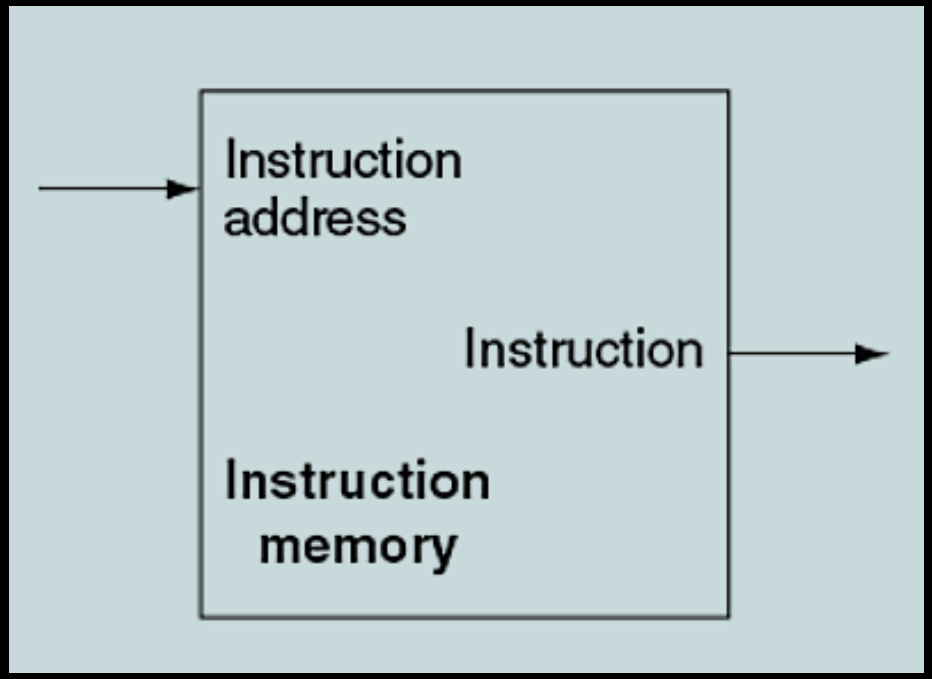
- `add, sub, and, or, slt`
- `lw, sw`
- `beq, j`

# DATAPATH

To start, we will look at the datapath elements needed by every instruction.

First, we have *instruction memory*.

Instruction memory is a state element that provides read-access to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address.

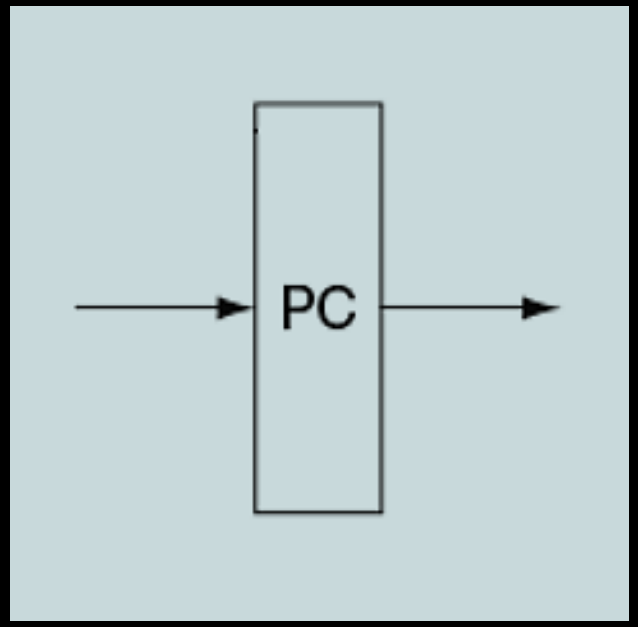


# DATAPATH

Next, we have the *program counter* or *PC*.

The PC is a state element that holds the address of the current instruction. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.

The arrows on either side indicate that the PC state element is both readable and writeable.

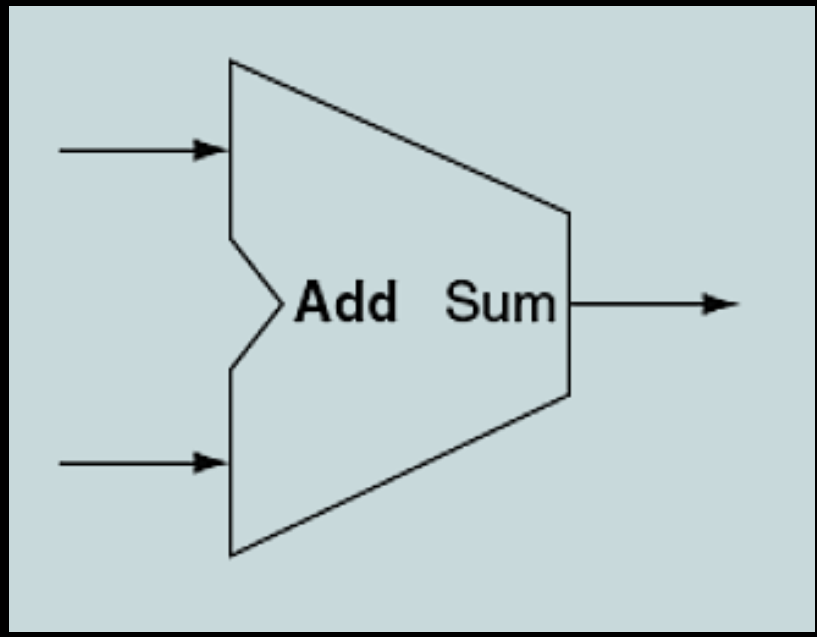


# DATAPATH

Lastly, we have the *adder*.

The adder is responsible for incrementing the PC to hold the address of the next instruction.

It takes two input values, adds them together and outputs the result.



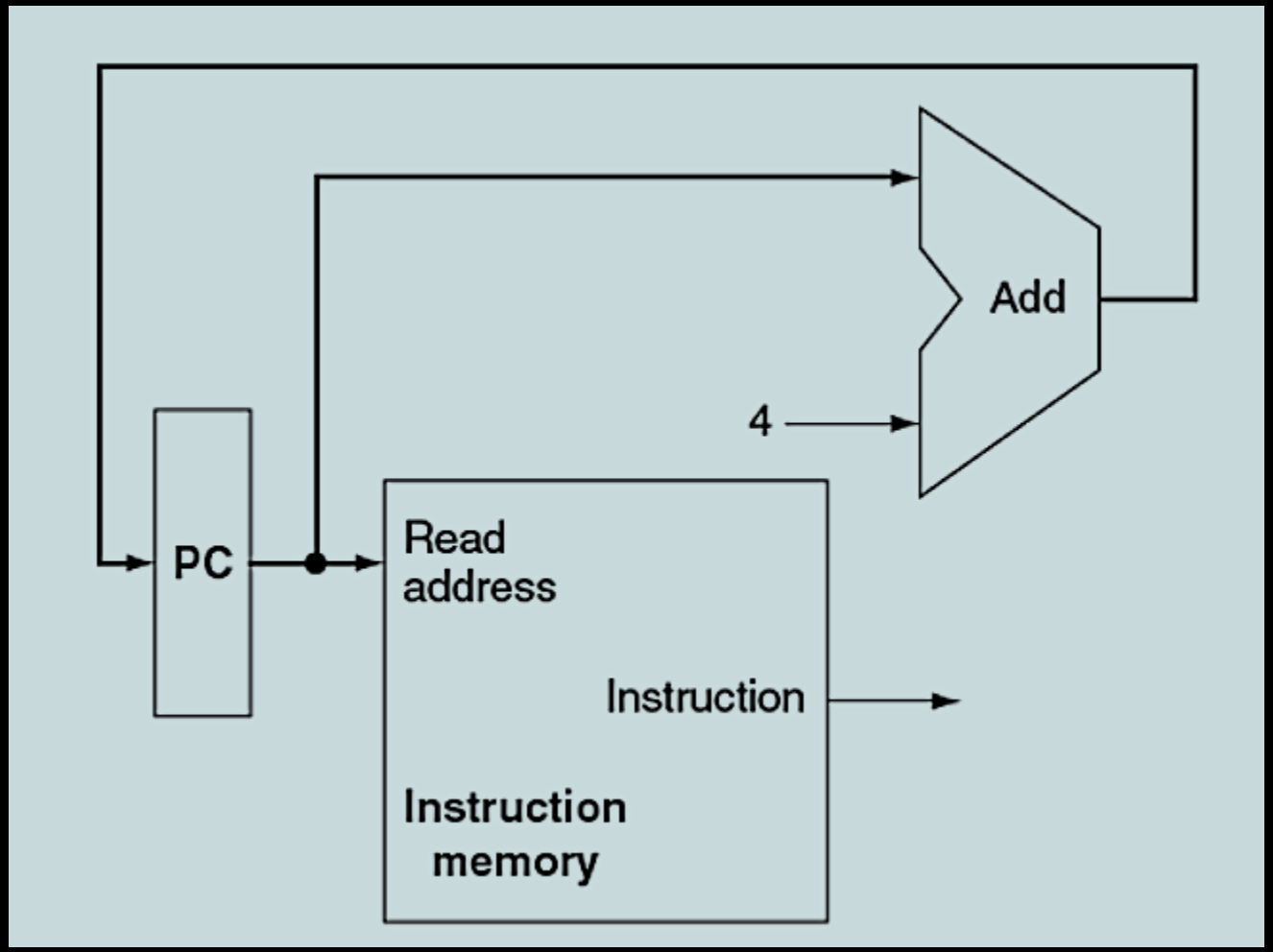
# DATAPATH

So now we have instruction memory, PC, and adder datapath elements. Now, we can talk about the general steps taken to execute a program.

- Use the address in the PC to fetch the current instruction from instruction memory.
- Determine the fields within the instruction (*decode* the instruction).
- Perform the operation indicated by the instruction.
- Update the PC to hold the address of the next instruction.

# DATAPATH

- Use the address in the PC to fetch the current instruction from instruction memory.
- Determine the fields within the instruction (*decode* the instruction).
- Perform the operation indicated by the instruction.
- Update the PC to hold the address of the next instruction.




Note: we perform PC+4 because instructions are word-aligned.

# R-FORMAT INSTRUCTIONS

Now, let's consider R-format instructions. In our limited MIPS instruction set, these are *add*, *sub*, *and*, *or* and *slt*.

All R-format instructions read two registers, *rs* and *rt*, and write to a register *rd*.

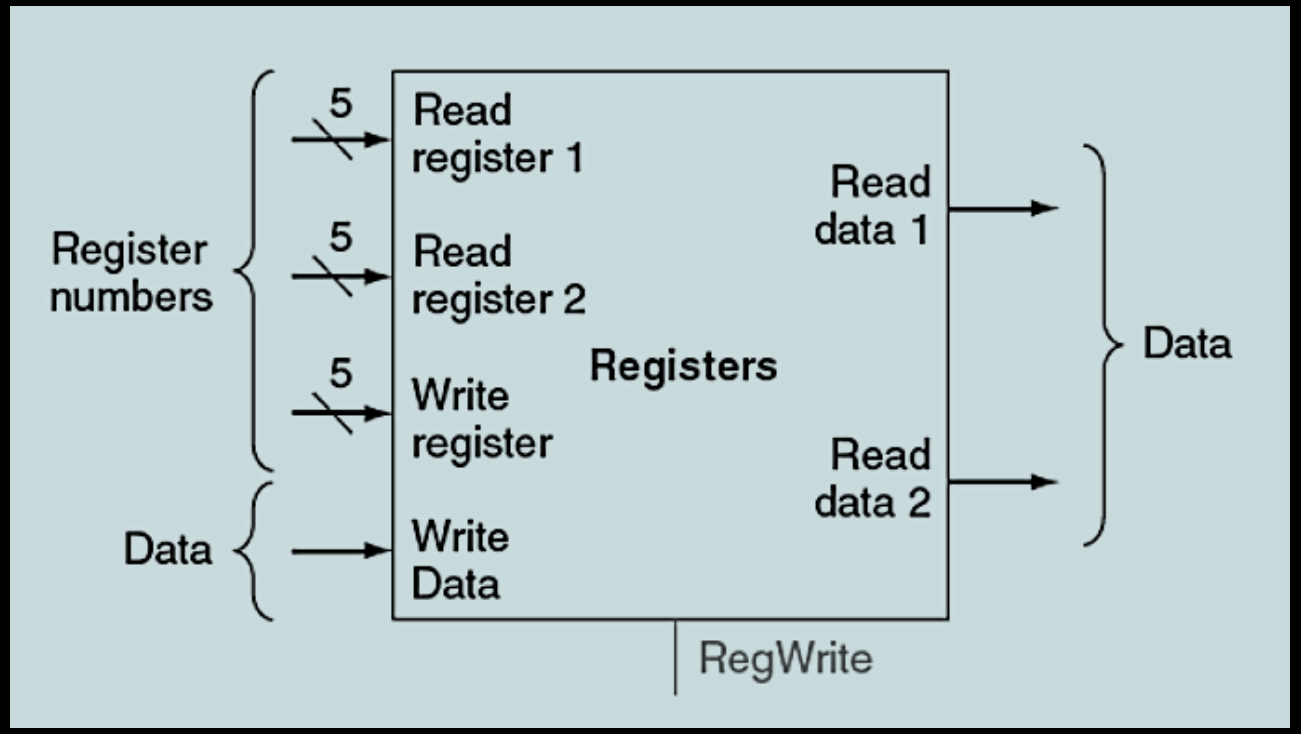
| **Name** | **Fields** | | | | | |
|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| R format | op | rs | rt | rd | shamt | funct |

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.
shamt – shift amount.
funct – additional opcodes.

# DATAPATH

To support R-format instructions, we'll need to add a state element called a *register file*. A register file is a collection readable/writeable registers.

- `Read register 1` – first source register. 5 bits wide.
- `Read register 2` – second source register. 5 bits wide.
- `Write register` – destination register. 5 bits wide.
- `Write data` – data to be written to a register. 32 bits wide.

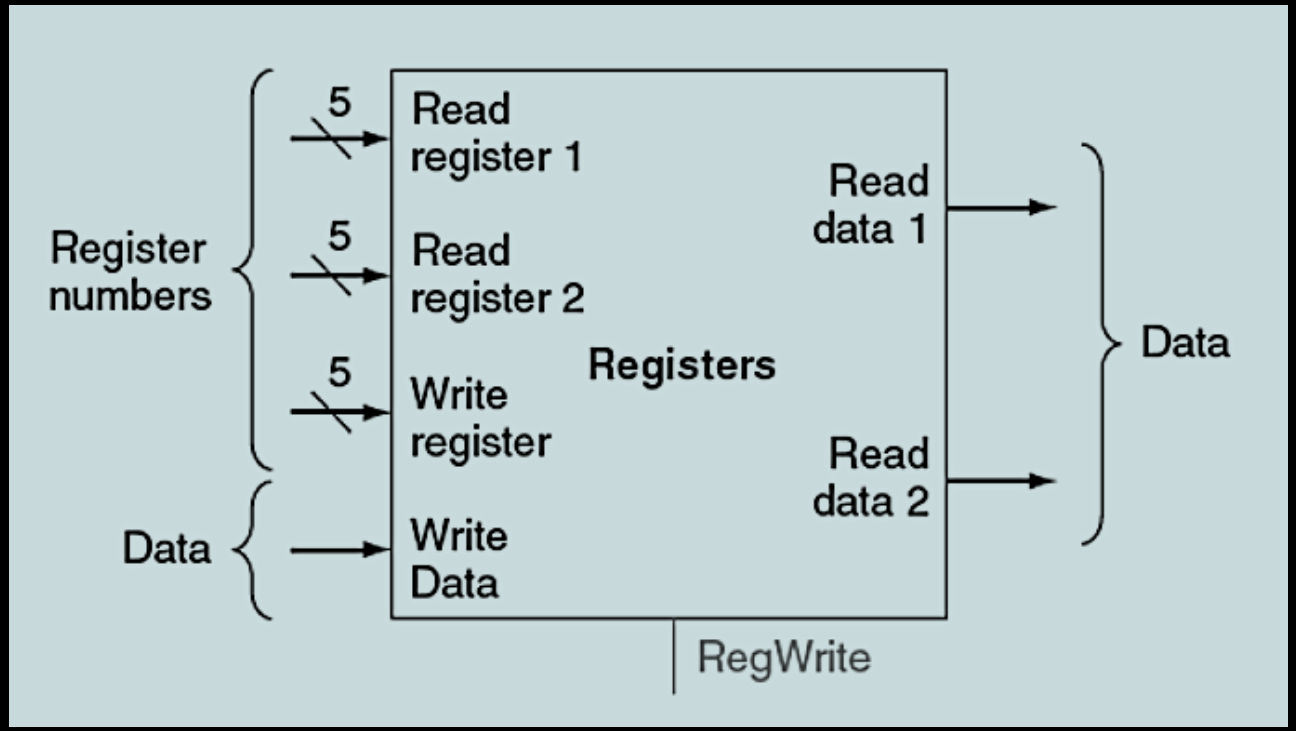


# DATAPATH

At the bottom, we have the `RegWrite` input. A writing operation only occurs when this bit is set.

The two output ports are:

- `Read data 1` – contents of source register 1.
- `Read data 2` – contents of source register 2.

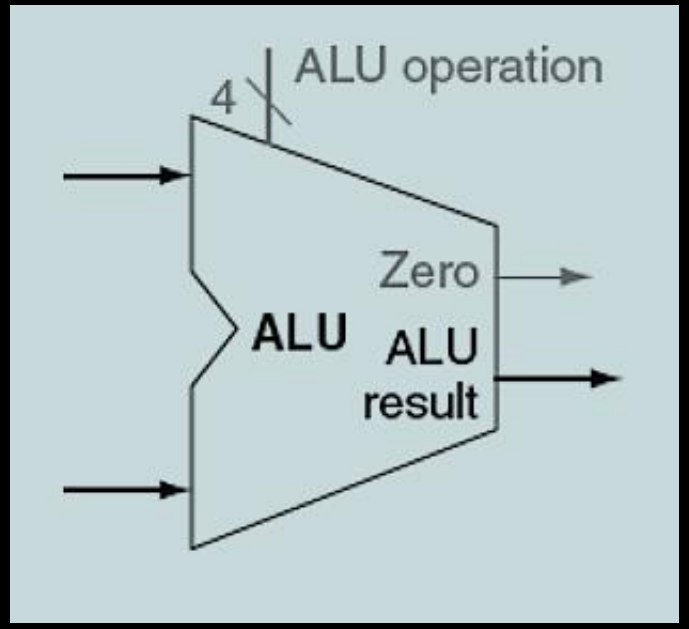


# DATAPATH

To actually perform R-format instructions, like `add` for example, we need to include the ALU element.

The ALU performs the operation indicated by the instruction. It takes two inputs, the operands to perform the operation on, as well as a 4-bit wide operation selector value.

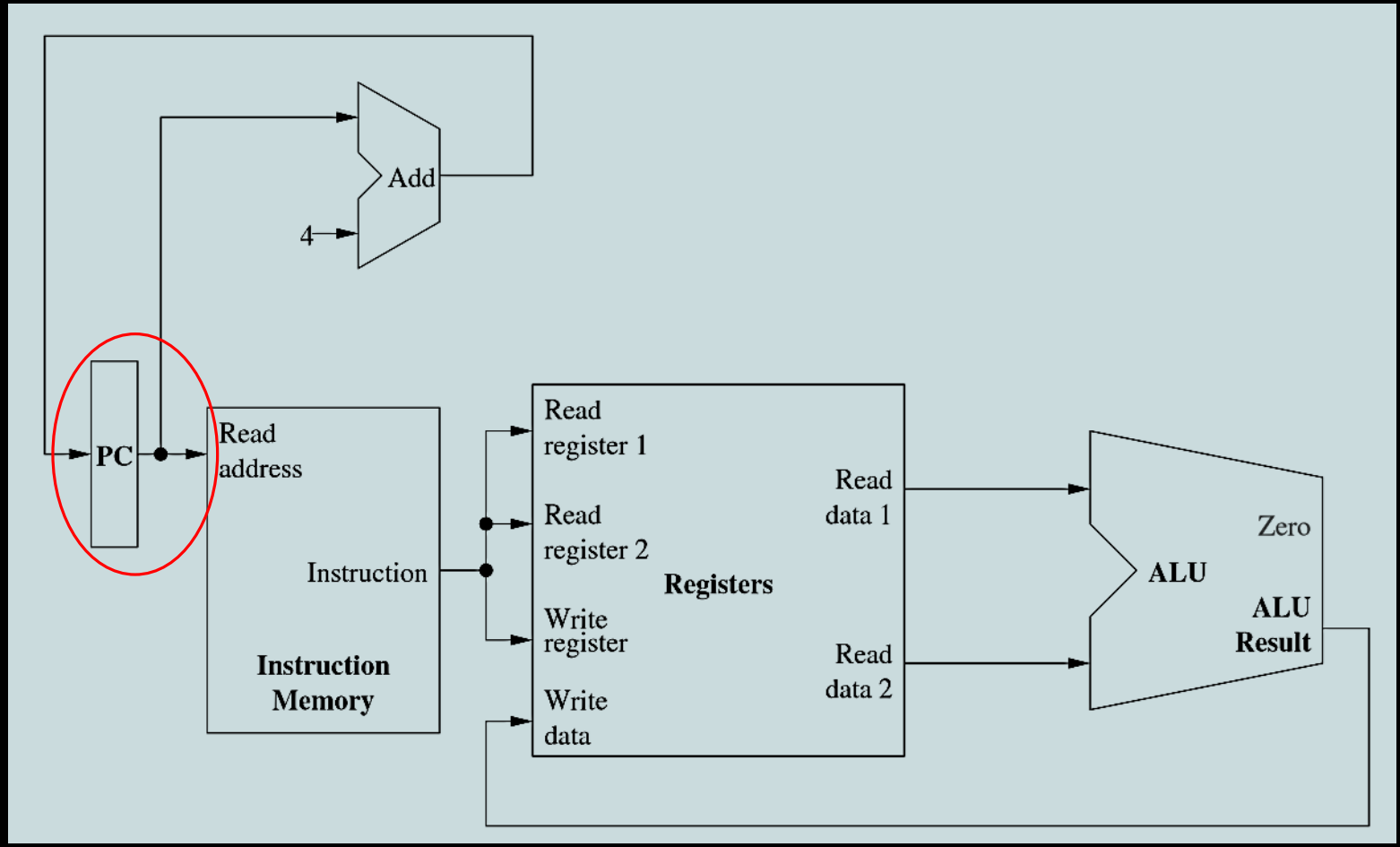
The result of the operation is the output value. We have an additional output specifically for branching – we will cover this in a minute.



# DATAPATH

Here is our datapath for R-format instructions.

1. Grab instruction address from PC.

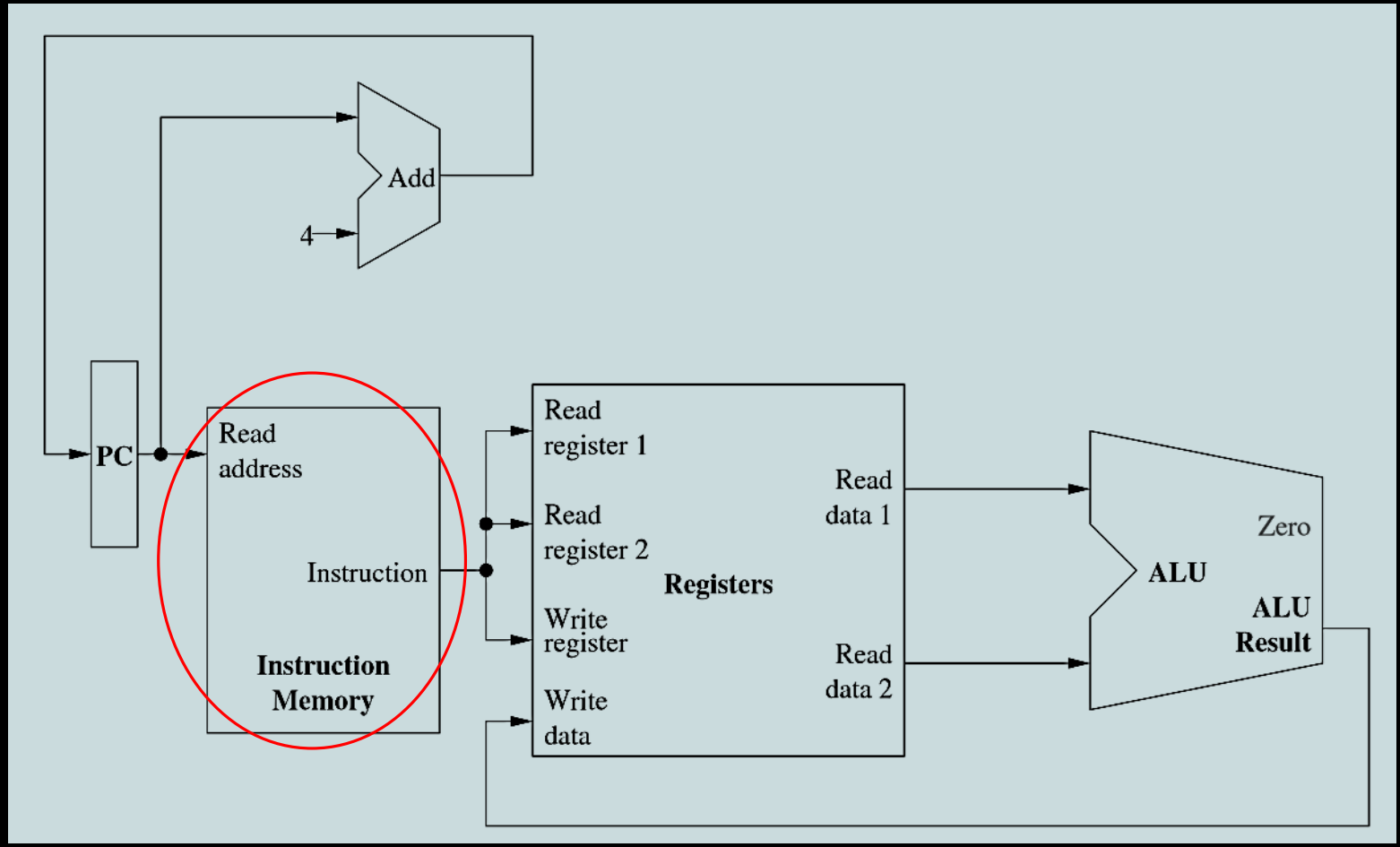


# DATAPATH

Here is our datapath for R-format instructions.

2. Fetch instruction from instruction memory.

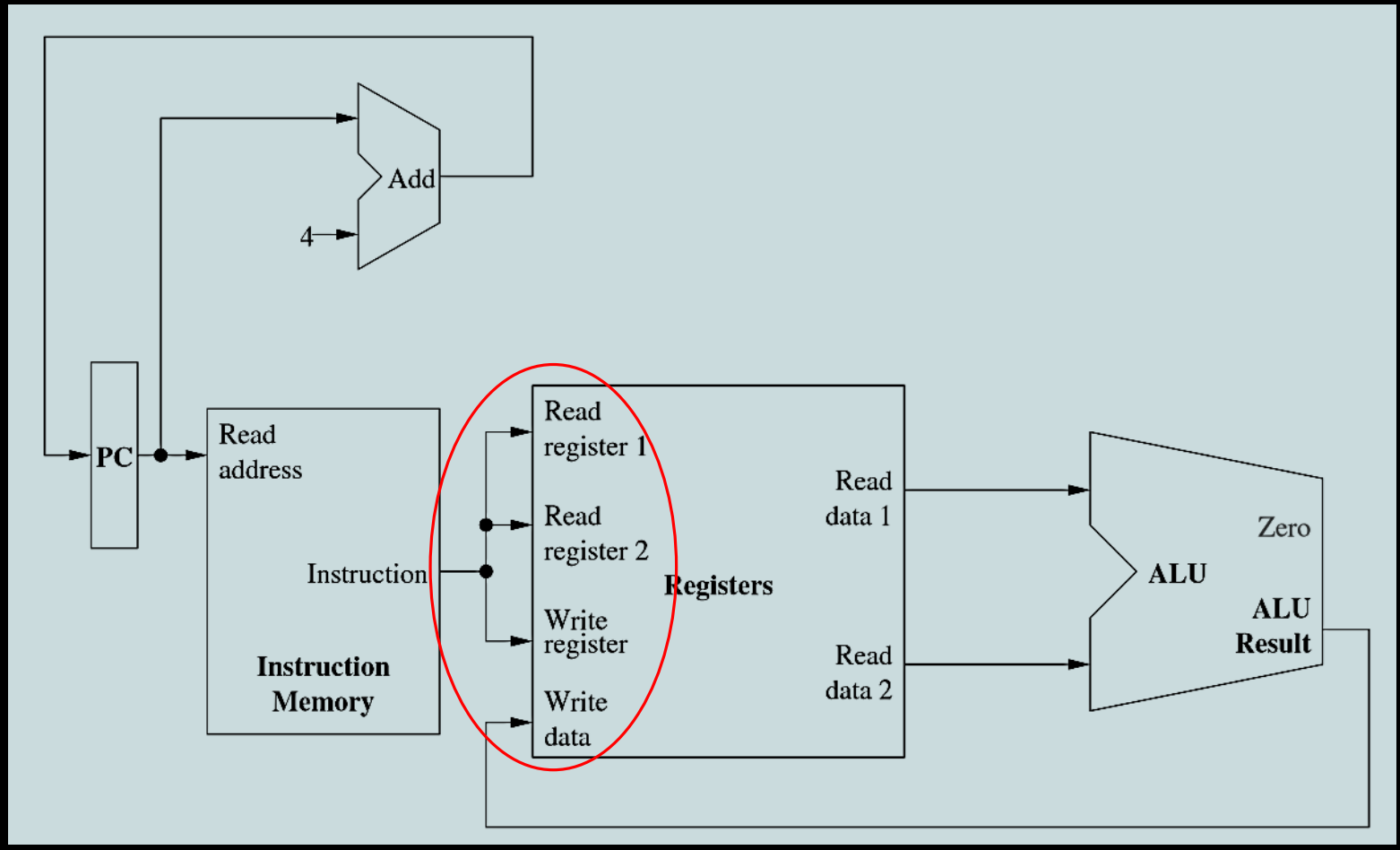
3. Decode instruction.



# DATAPATH

Here is our datapath for R-format instructions.

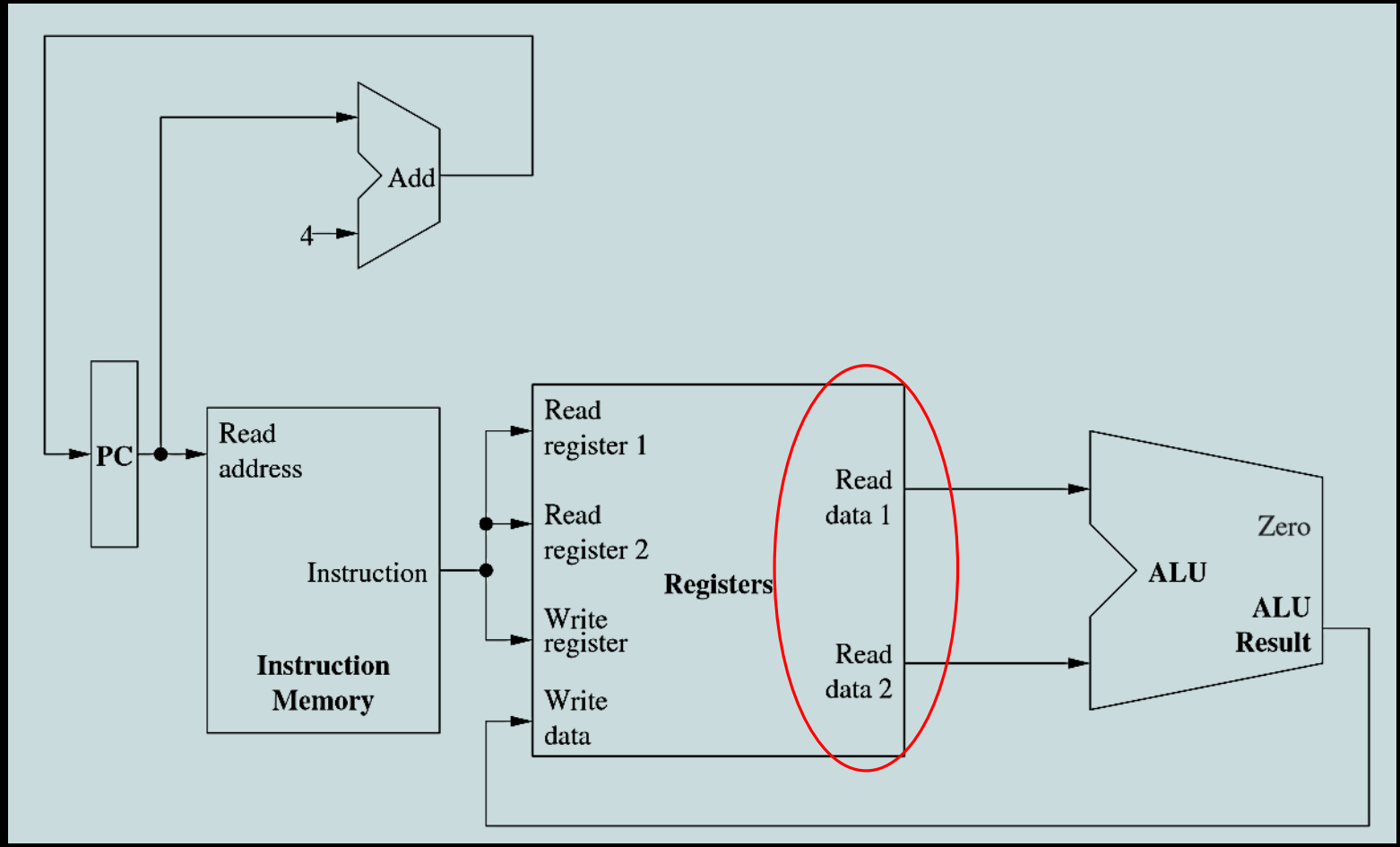
4. Pass `rs`, `rt`, and `rd` into read register and write register arguments.



# DATAPATH

Here is our datapath for R-format instructions.

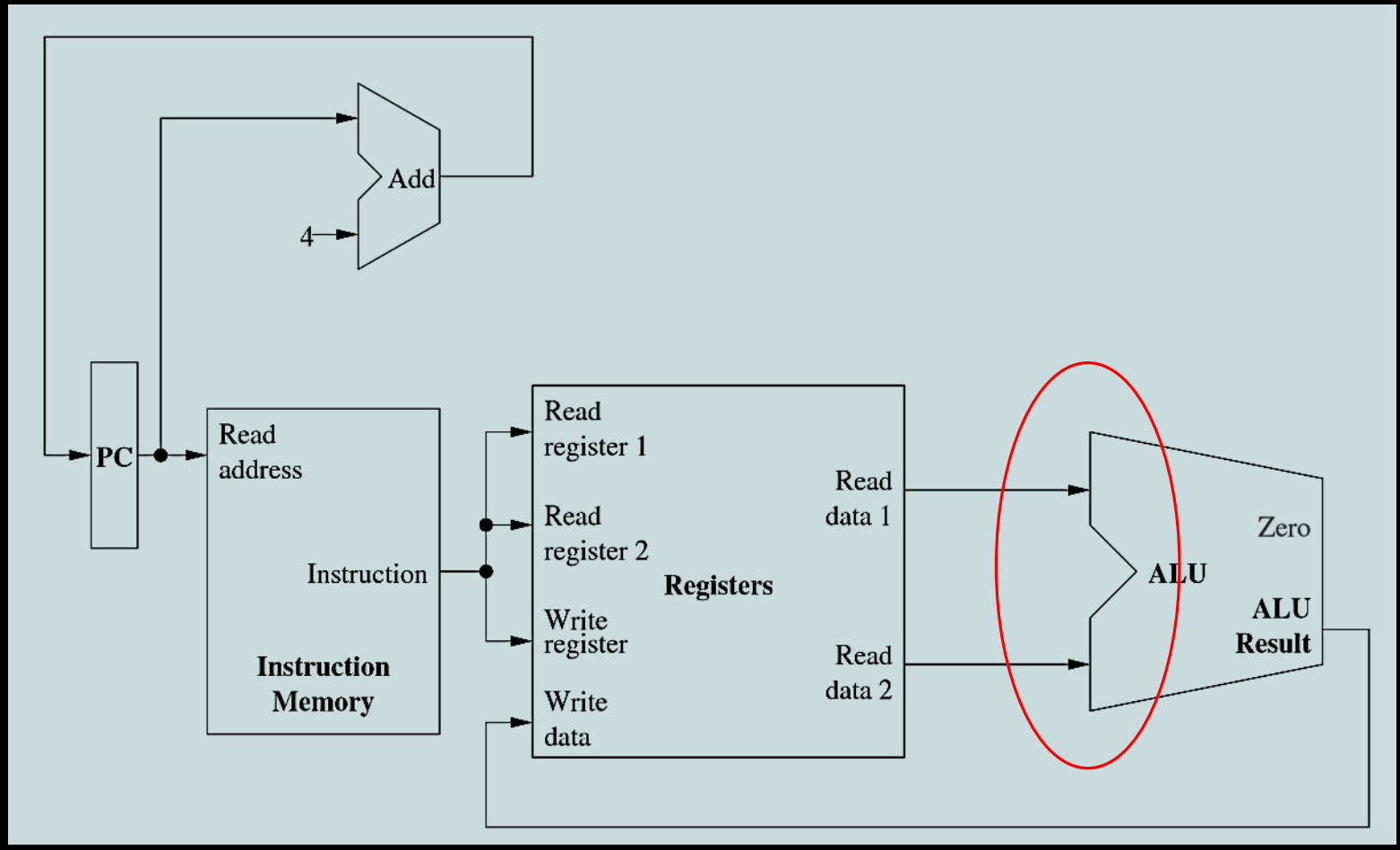
5. Retrieve contents of read register 1 and read register 2 (`rs` and `rt`).



# DATAPATH

Here is our datapath for R-format instructions.

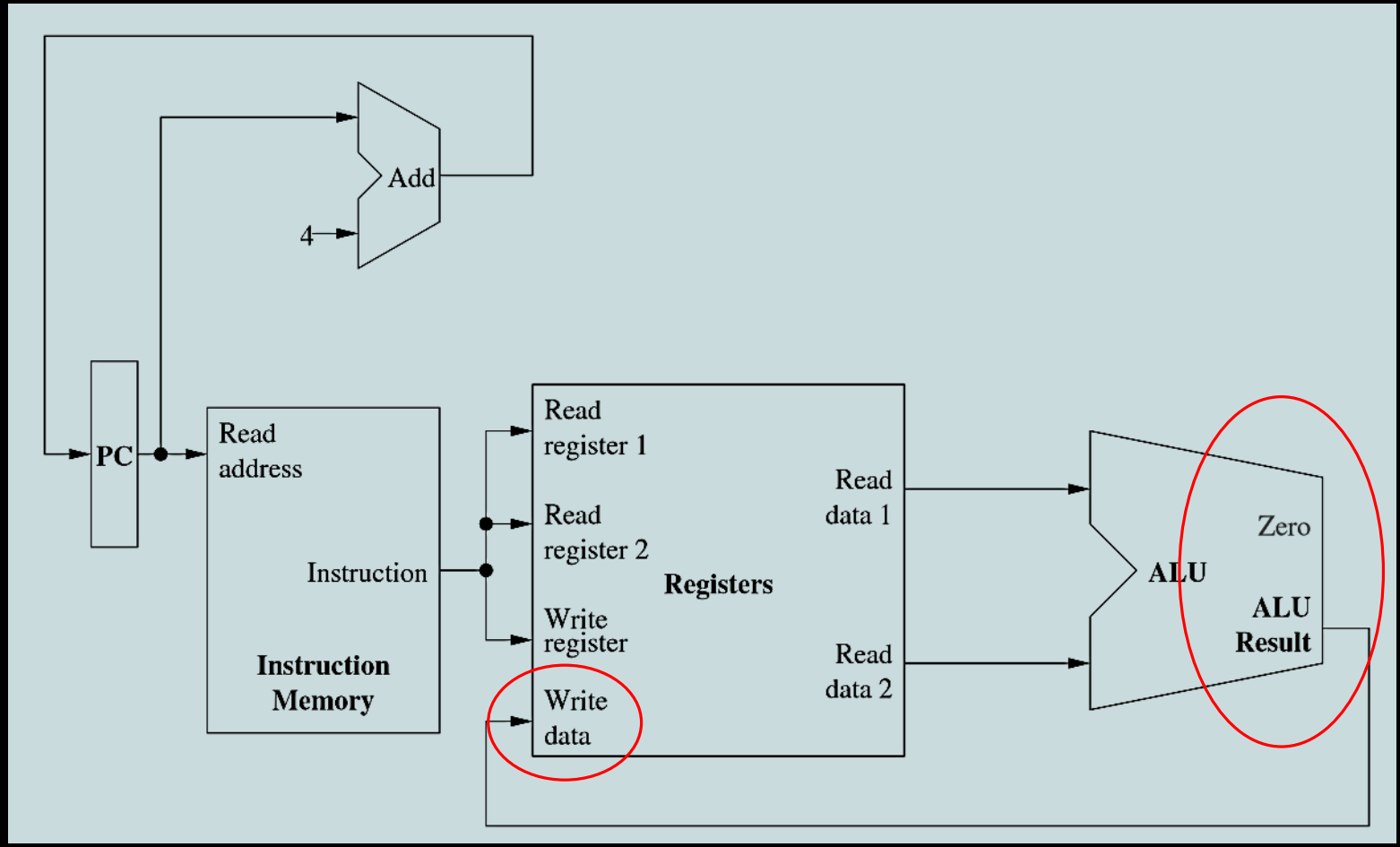
6. Pass contents of `rs` and `rt` into the ALU as operands of the operation to be performed.



# DATAPATH

Here is our datapath for R-format instructions.

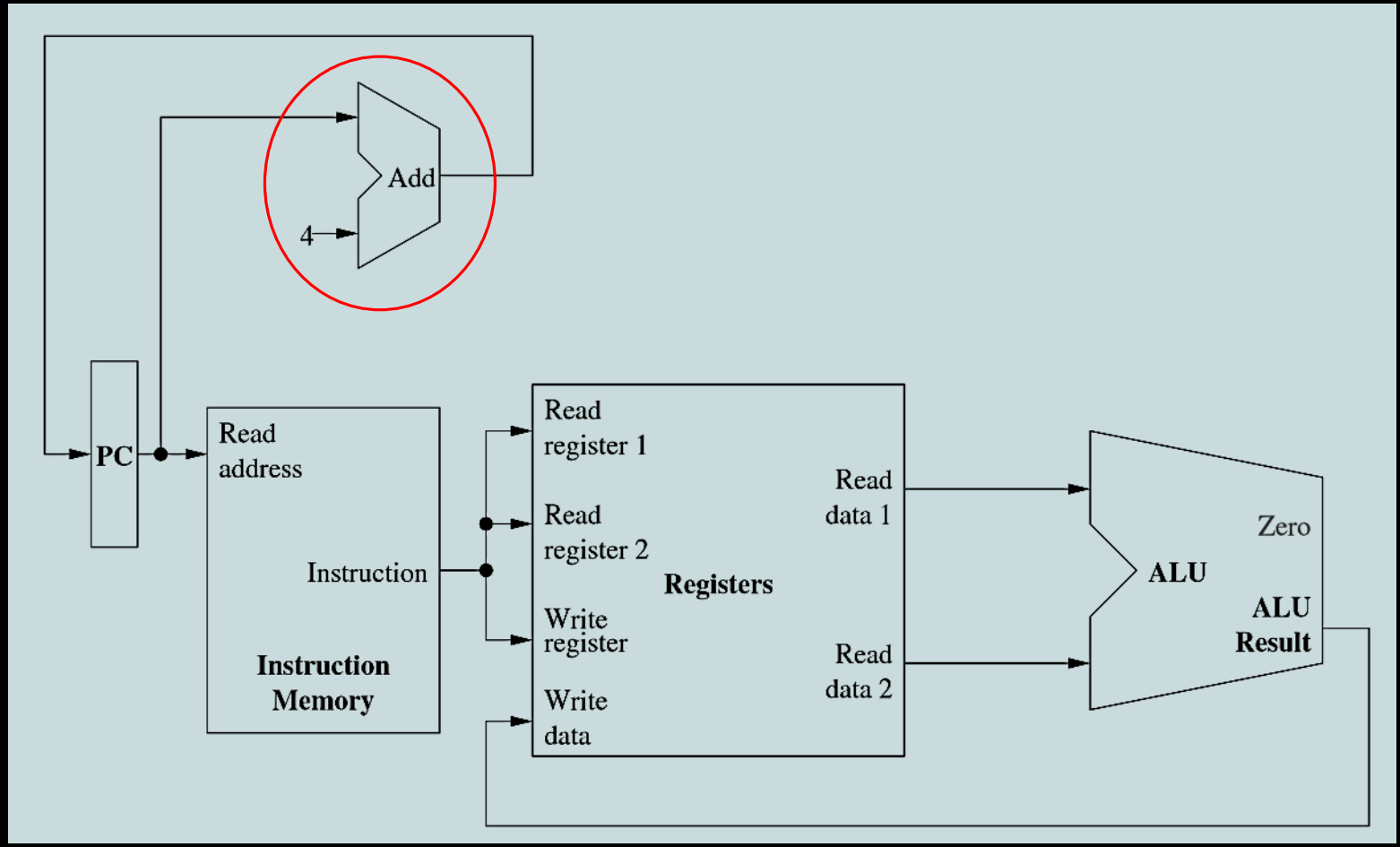
7. Retrieve result of operation performed by ALU and pass back as the write data argument of the register file (with the `RegWrite` bit set).



# DATAPATH

Here is our datapath for R-format instructions.

8. Add 4 bytes to the PC value to obtain the word-aligned address of the next instruction.



# I-FORMAT INSTRUCTIONS

Now that we have a complete datapath for R-format instructions, let's add in support for I-format instructions. In our limited MIPS instruction set, these are `lw`, `sw`, and `beq`.

- The *op* field is used to identify the type of instruction.
- The *rs* field is the source register.
- The *rt* field is either the source or destination register, depending on the instruction.
- The *immed* field is zero-extended if it is a logical operation. Otherwise, it is sign-extended.

| **Name** | **Fields** | | | | | |
|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| I format | op | rs | rt | immed | | |

# DATA TRANSFER INSTRUCTIONS

Let's start with accommodating the data transfer instructions – we'll get to `beq` in a bit. For `lw` and `sw`, we have the following format:

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

- The memory address is computed by *sign-extending* the 16-bit immediate to 32-bits, which is added to the contents of $rs.
- In lw, $rt represents the register that will be assigned the memory value. In sw, $rt represents the register whose value will be stored in memory.

Bottom line: we need two more datapath elements to access memory and perform sign-extending.

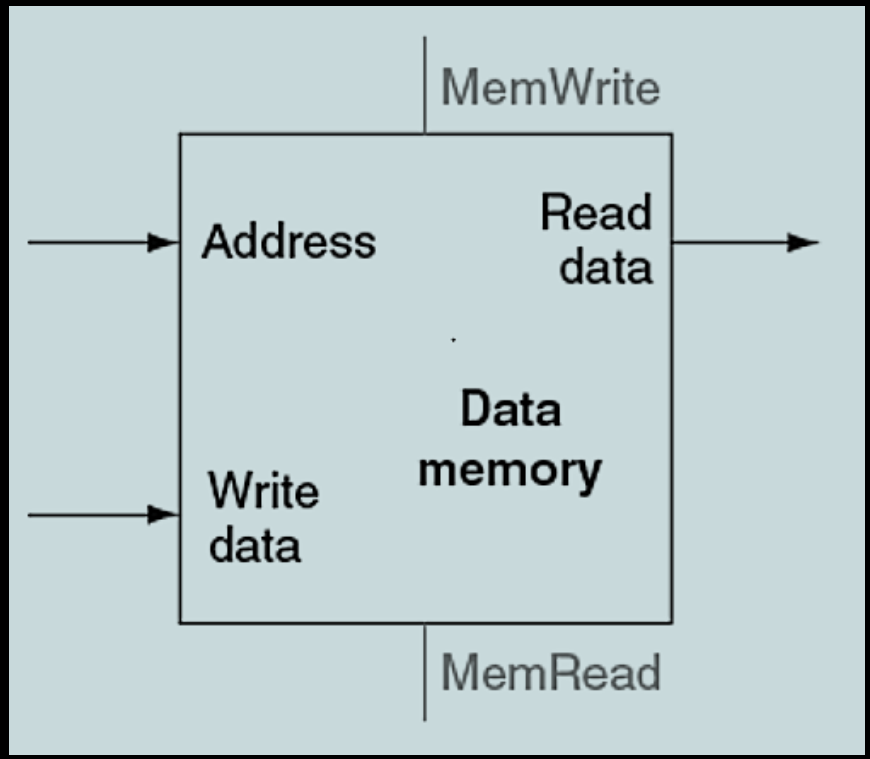
# DATAPATH

The *data memory* element implements the functionality for reading and writing data to/from memory.

There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.

The output is the data read from the memory location accessed, if applicable.

Reads and writes are signaled by MemRead and MemWrite, respectively, which must be asserted for the corresponding action to take place.

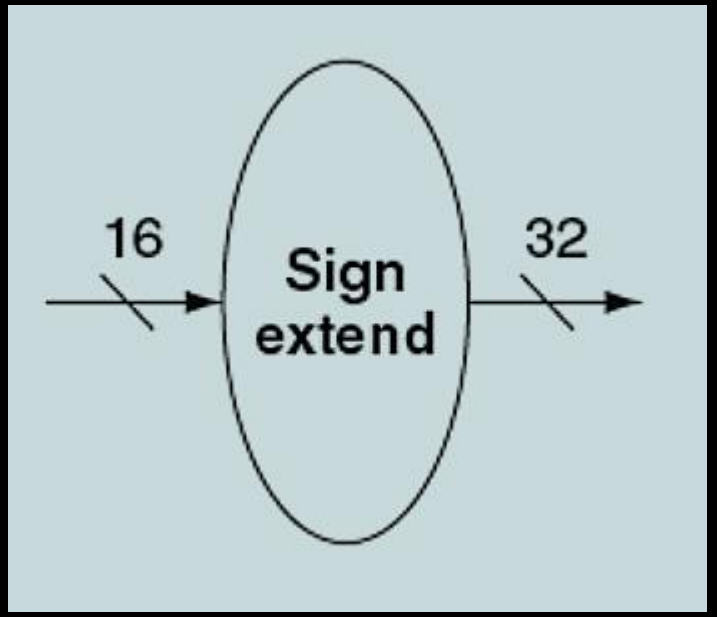


# DATAPATH

To perform sign-extending, we can add a sign extension element.

The sign extension element takes as input a 16-bit wide value to be extended to 32-bits.

To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.

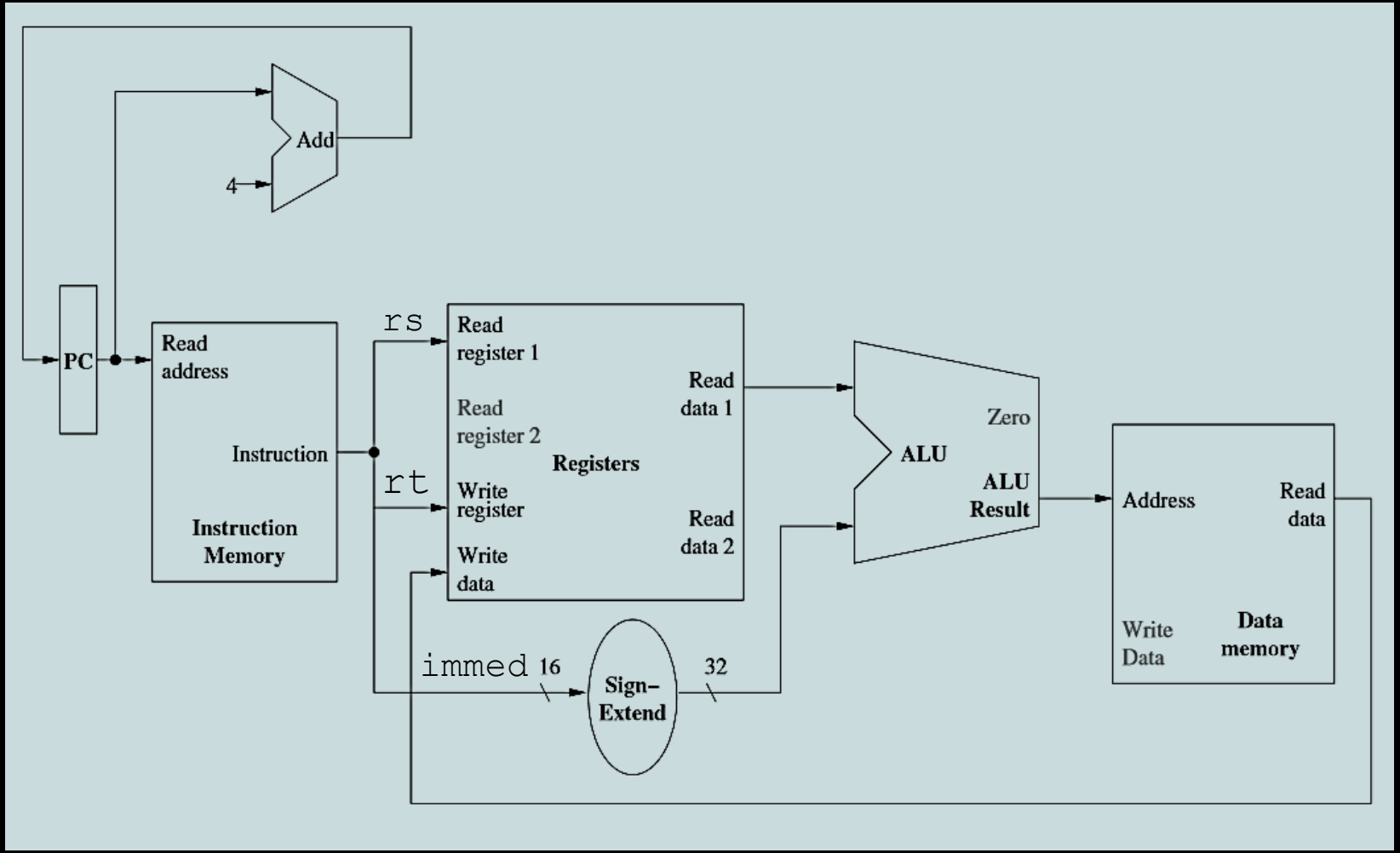


# DATAPATH FOR LOAD WORD

Here, we have modified the datapath to work only for the `lw` instruction.

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

The registers have been added to the datapath for added clarity.

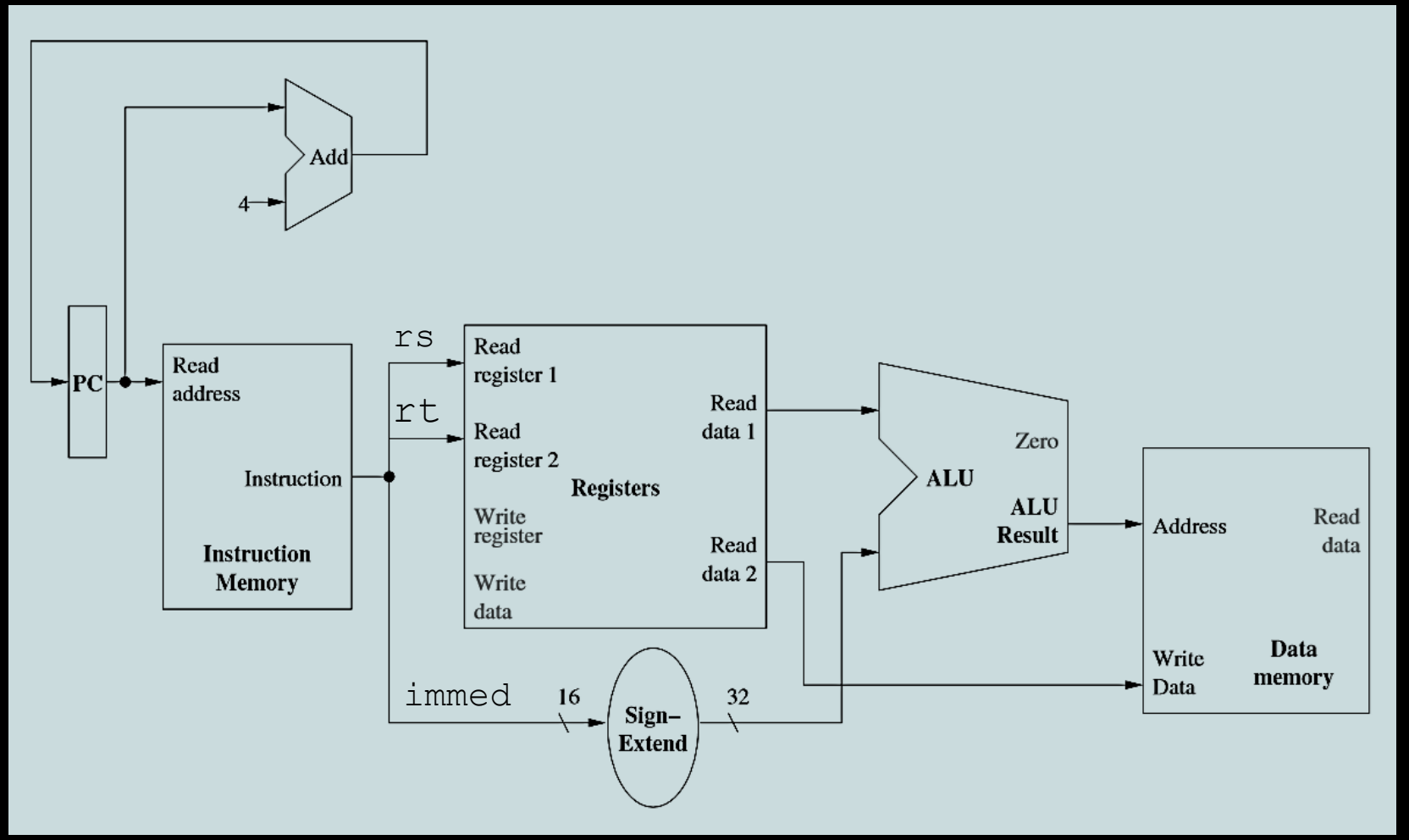


# DATAPATH FOR STORE WORD

Here, we have modified the datapath to work only for the `sw` instruction.

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

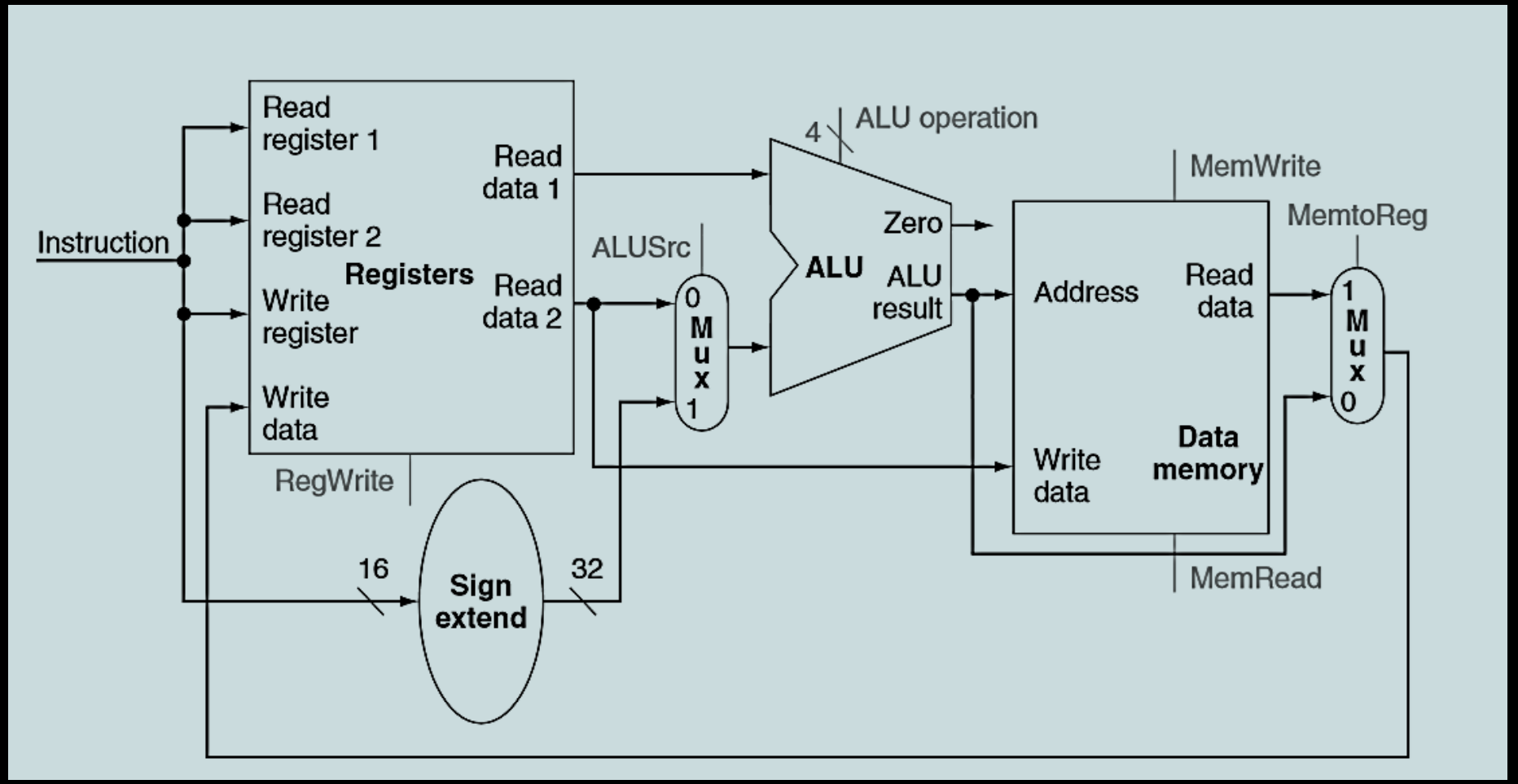
The registers have been added to the datapath for added clarity.



# DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

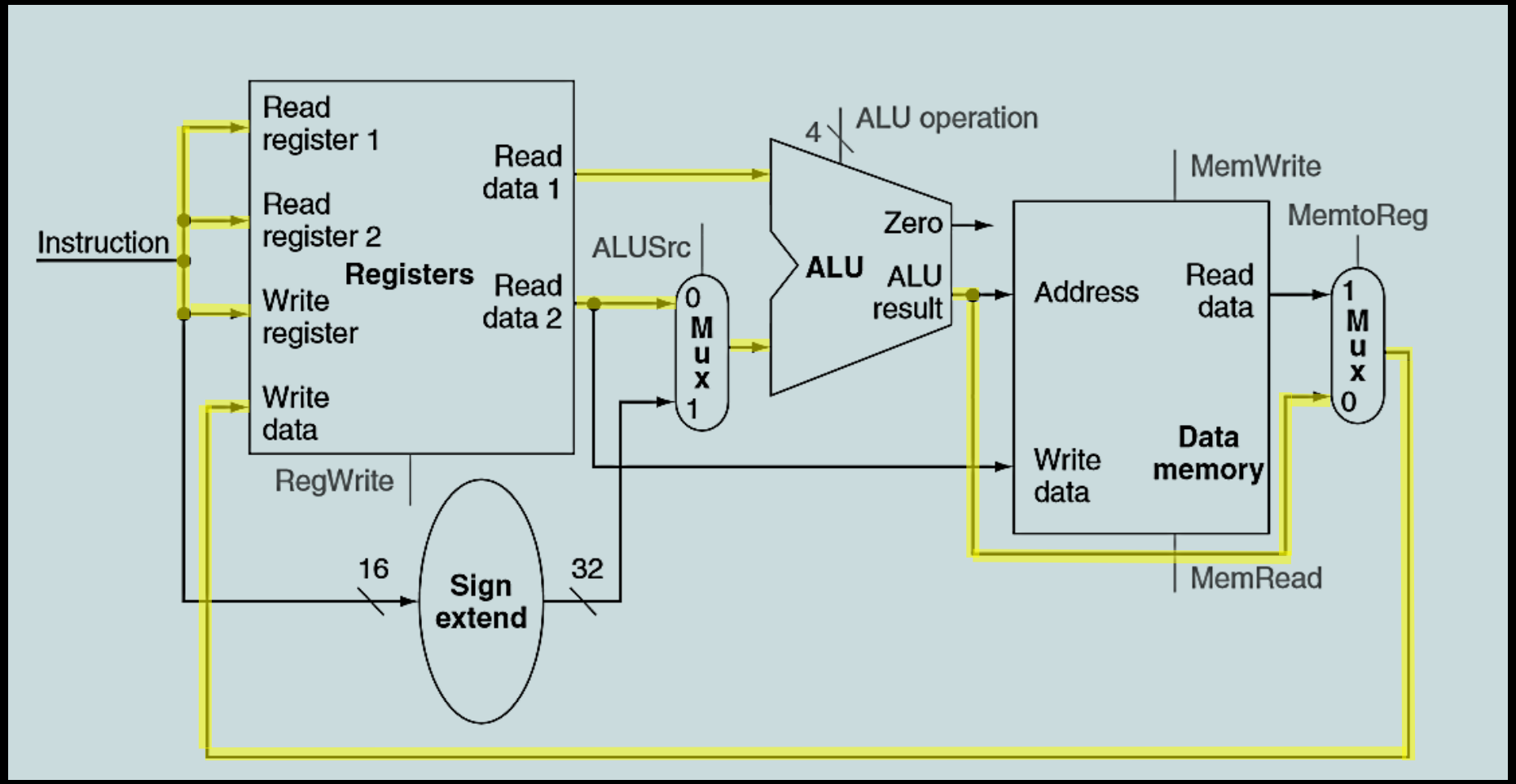
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



# DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

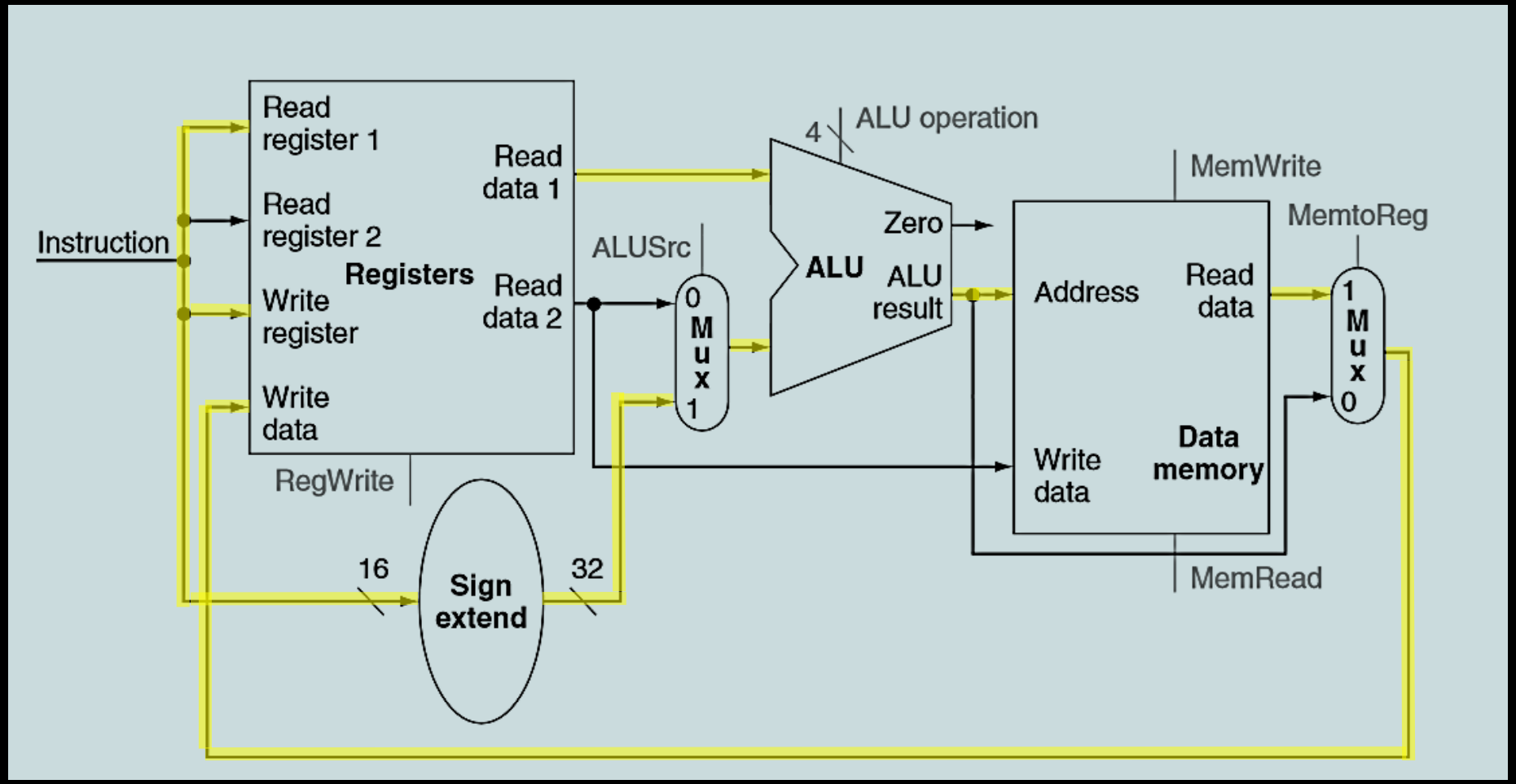
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



# DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

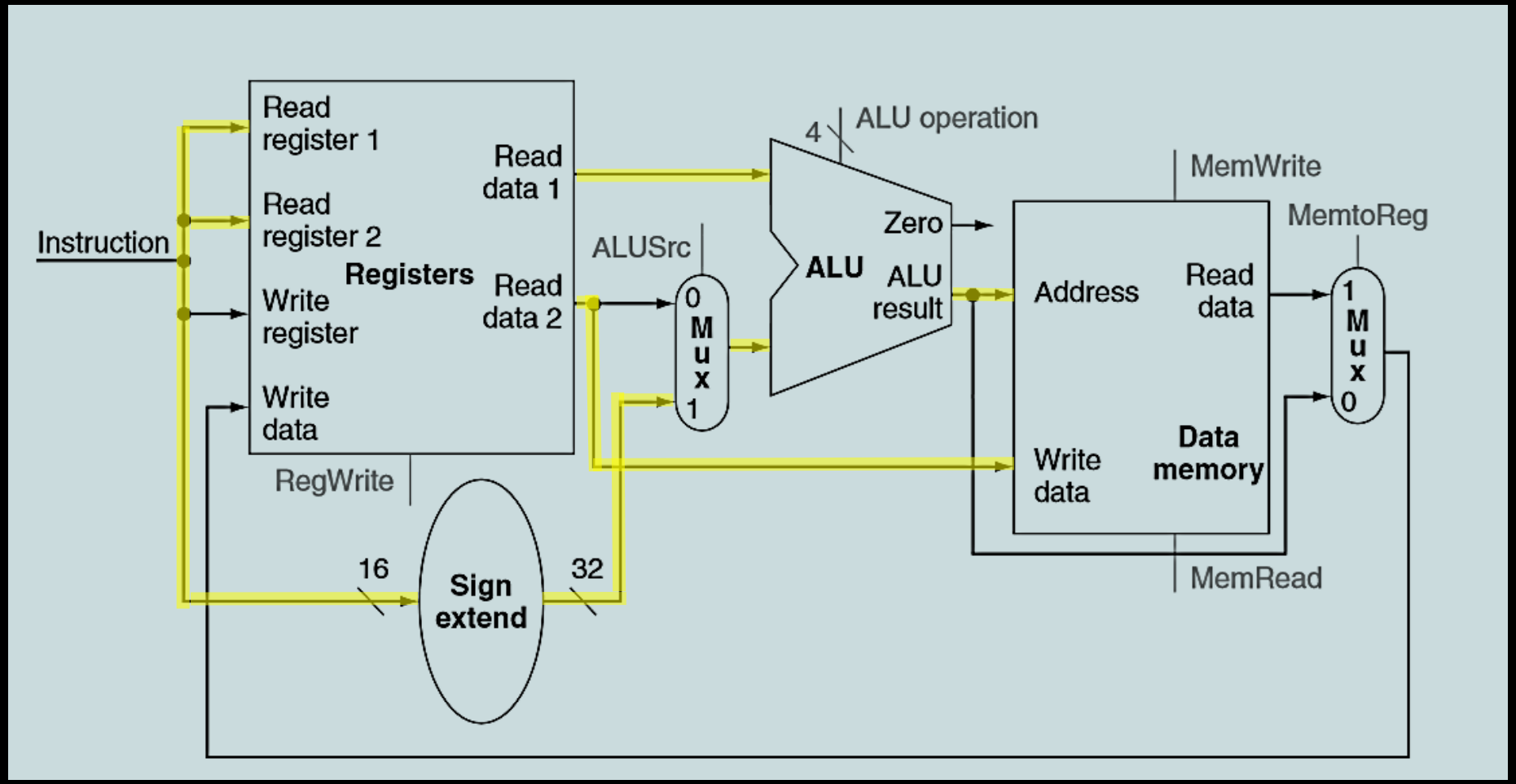
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



# DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



# BRANCHING INSTRUCTIONS

Now we'll turn out attention to a branching instruction. In our limited MIPS instruction set, we have the beq instruction which has the following form:

```
beq $t1, $t2, target
```

This instruction compares the contents of $t1 and $t2 for equality and uses the 16-bit immediate field to compute the target address of the branch relative to the current address.

| **Name** | **Fields** | | | | | |
|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| I format | op | rs | rt | immed | | |

# BRANCHING INSTRUCTIONS

Note that our immediate field is only 16-bits so we can't specify a full 32-bit target address. So we have to do a few things before jumping.

- The immediate field is left-shifted by two because the immediate represents the number of *words* offset from PC+4, not the number of bytes (and we want to get it in number of bytes!).
- We sign-extend the immediate field to 32-bits and add it to PC+4.

```
beq $t1, $t2, target
```

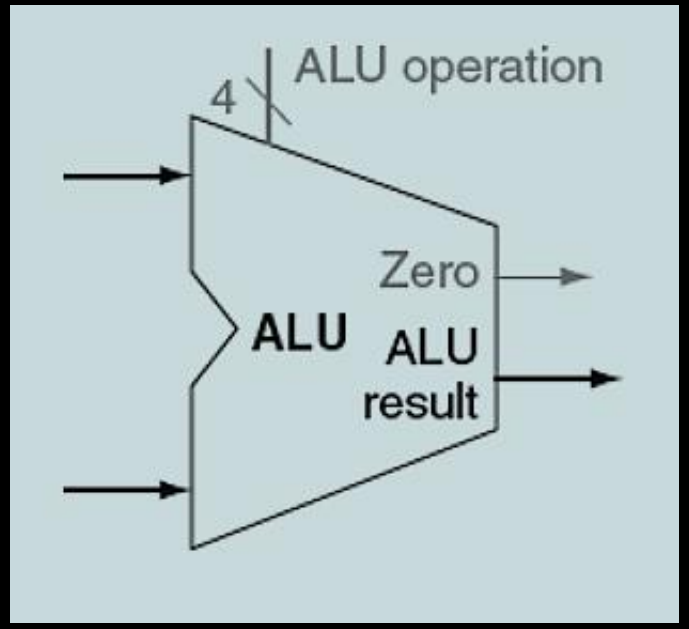
| **Name** | **Fields** | | | | | |
|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| I format | op | rs | rt | immed | | |

# BRANCHING INSTRUCTIONS

Besides computing the target address, a branching instruction also has to compare the contents of the source registers.

As stated before, the ALU has an output line denoted as Zero. This output is specifically hardwired to be set when the result of an operation is zero.

To test whether a and b are equal, we can set the ALU to perform a subtraction operation. The Zero output line is only set if a – b is 0, indicating a and b are equal.

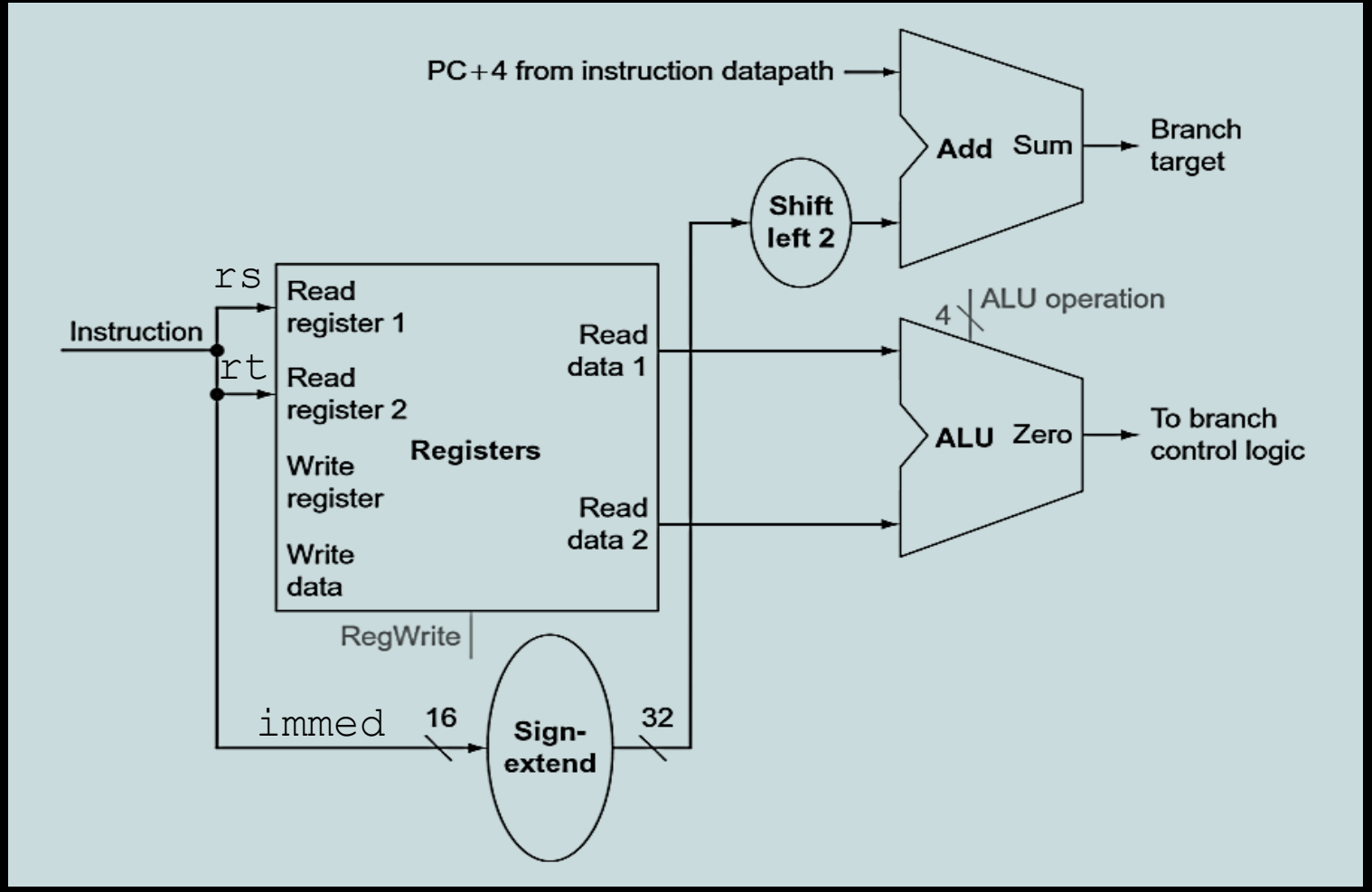


# DATAPATH FOR BEQ

Here, we have modified the datapath to work only for the `beq` instruction.

```
beq $rs, $rt, immed
```

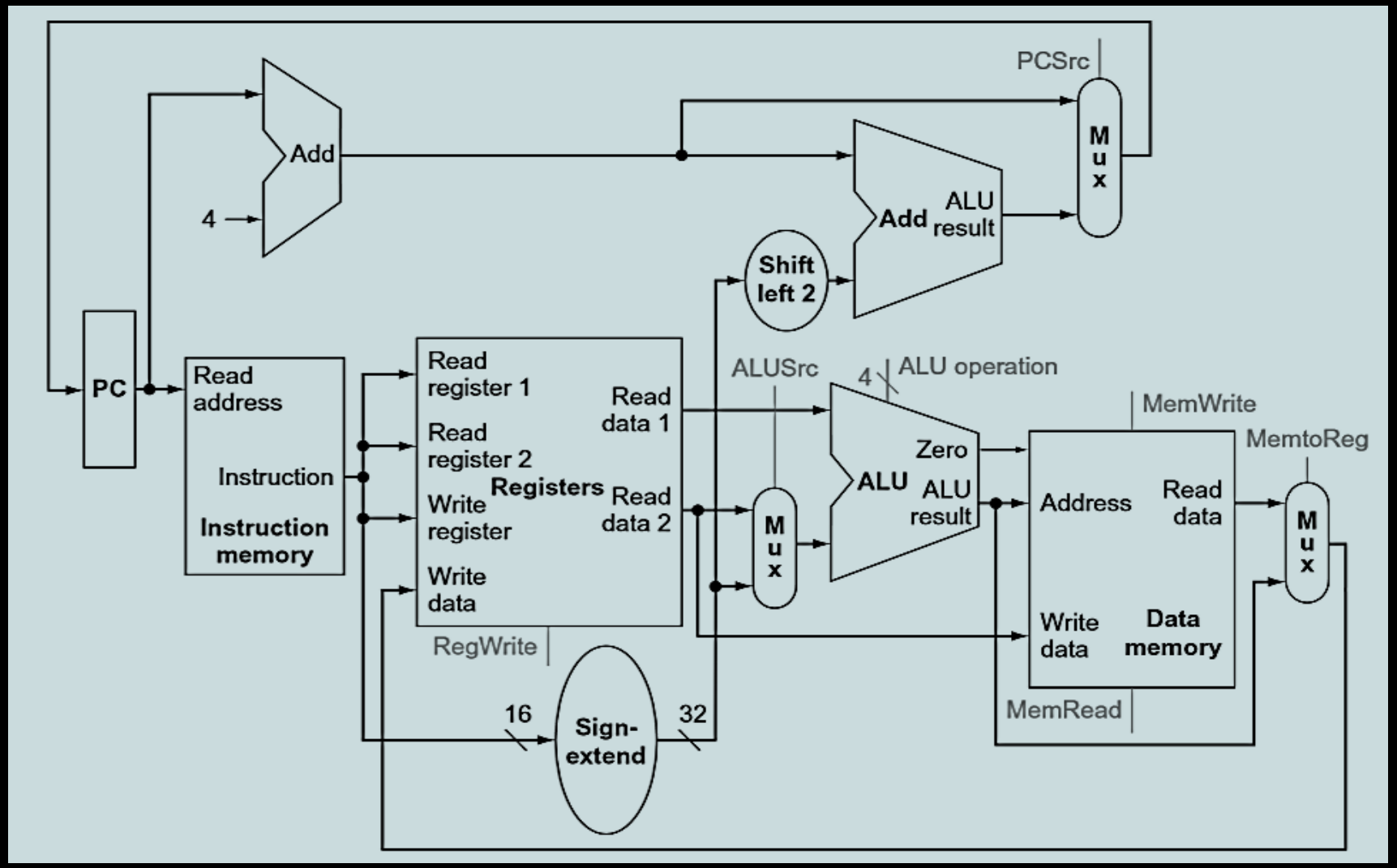
The registers have been added to the datapath for added clarity.



# DATAPATH FOR R AND I FORMAT

Now we have a datapath which supports all of our R and I format instructions.

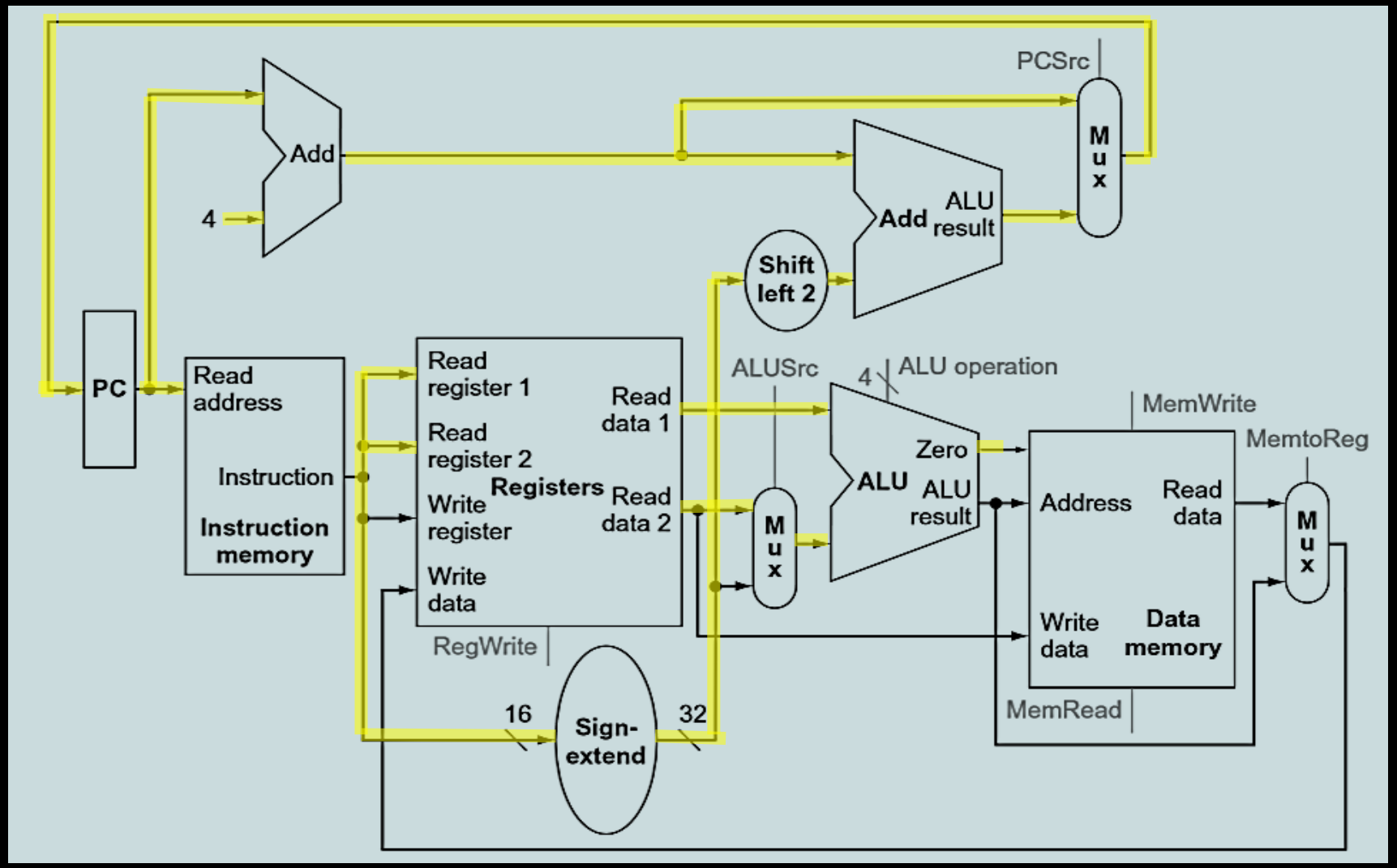
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
beq $rs, $rt, immed
```



# DATAPATH FOR R AND I FORMAT

Now we have a datapath which supports all of our R and I format instructions.

```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
beq $rs, $rt, immed
```



# J-FORMAT INSTRUCTIONS

The last instruction we have to implement in our simple MIPS subset is the jump instruction. An example jump instruction is `j L1`. This instruction indicates that the next instruction to be executed is at the address of label L1.

- We have 6 bits for the opcode.
- We have 26 bits for the target address.

| **Name** | **Fields** | | | | | |
|---|---|---|---|---|---|---|
| Field Size | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
| J format | op | targaddr | | | | |

# J-FORMAT INSTRUCTIONS

Note, as in the branching example, we do not have enough space in the instruction to specify a full target address. Branching solves this problem by specifying an offset in words.

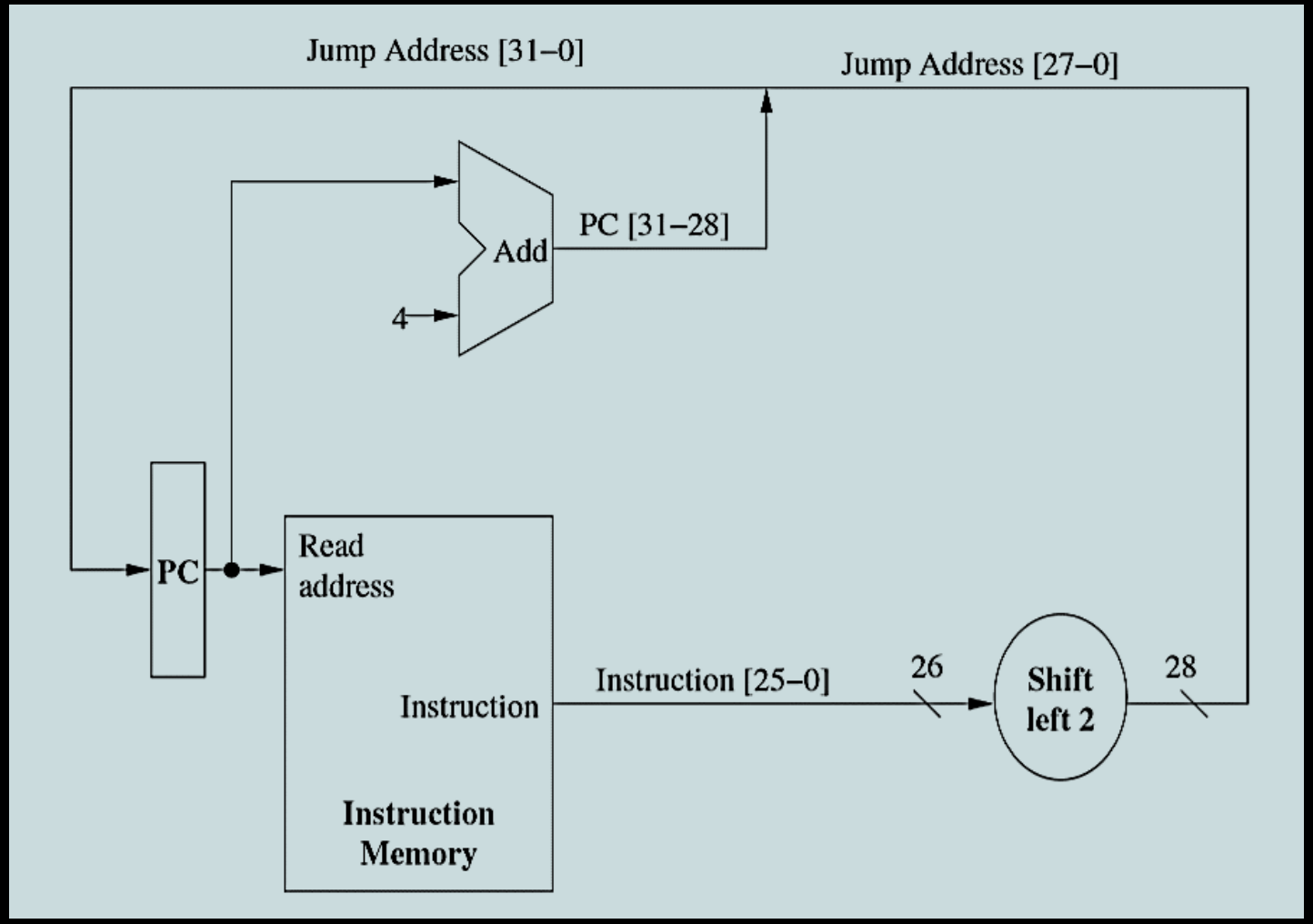
Jump instructions solve this problem by specifying *a portion of an absolute address, not an offset.*

We take the 26-bit target address field of the instruction, left-shift by two (because we are word-aligned), and concatenate the result with the upper 4 bits of PC+4.

# DATAPATH FOR J-FORMAT

Here, we have modified the datapath to work only for the `j` instruction.

```
j targaddr
```



# SINGLE-CYCLE CONTROL

Now we have a complete datapath for our simple MIPS subset – we will show the whole diagram in just a couple of minutes. Before that, we will add the control.

The *control unit* is responsible for taking the instruction and generating the appropriate signals for the datapath elements.

Signals that need to be generated include

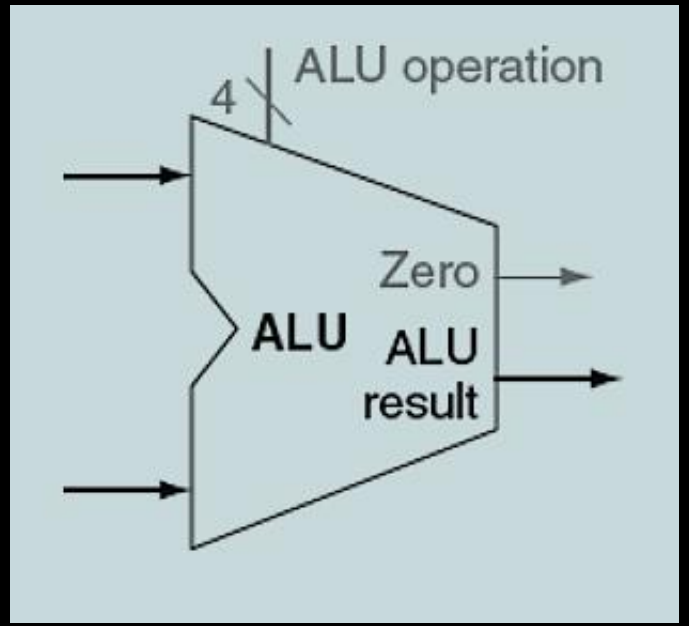
- Operation to be performed by ALU.
- Whether register file needs to be written.
- Signals for multiple intermediate multiplexors.
- Whether data memory needs to be written.

For the most part, we can generate these signals using only the *opcode* and *funct* fields of an instruction.

# ALU CONTROL LINES

Note here that the ALU has a 4-bit control line called ALU operation. The first two bits indicate whether a and b need to be inverted, respectively. The last two bits indicate the operation.

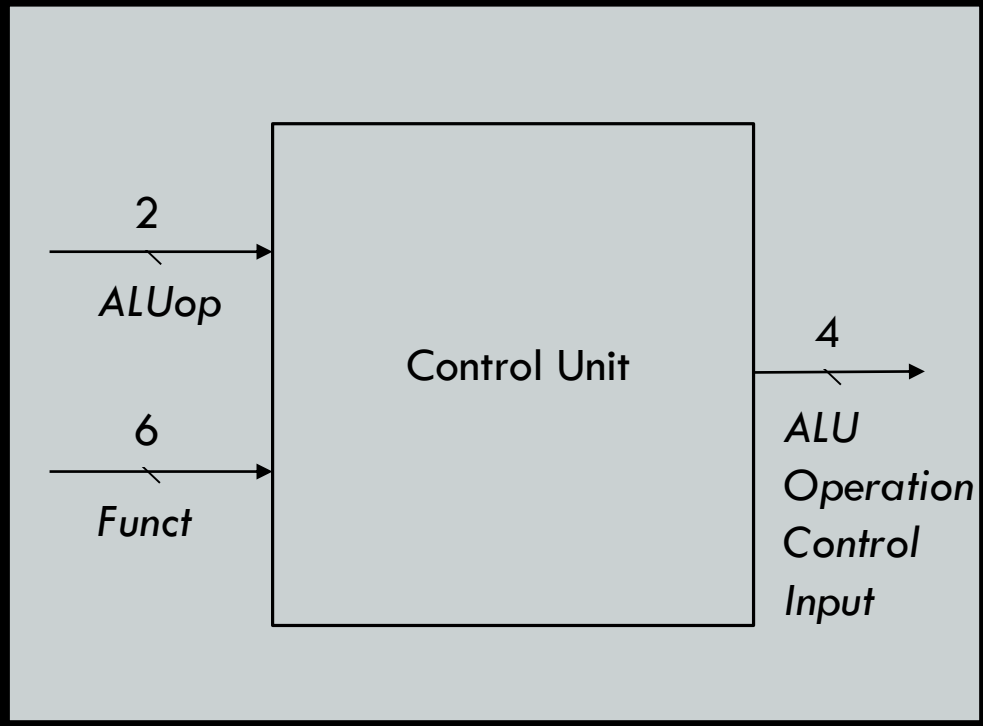
| ALU Control Lines | Function |
|---|---|
| 0000 | AND |
| 0001 | OR |
| 0010 | Add |
| 0110 | Subtract |
| 0111 | Set on less than |
| 1100 | NOR |



# ALU CONTROL LINES

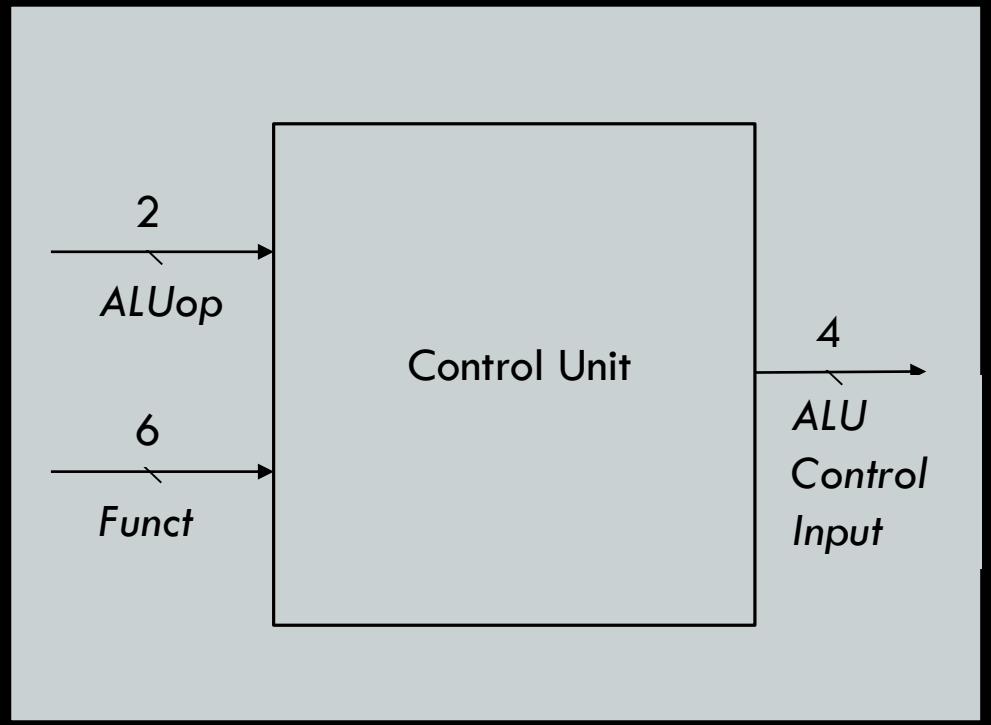
How do we set these control lines? Consider the control unit below.

- The 2-bit *ALUop* input indicates whether an operation should be add (00) for loads and stores, subtract (01) for beq, or determined by the funct input (10).
- The 6-bit *Funct* input corresponds to the funct field of R-format instructions. Each unique funct field corresponds to a unique set of ALU control input lines.



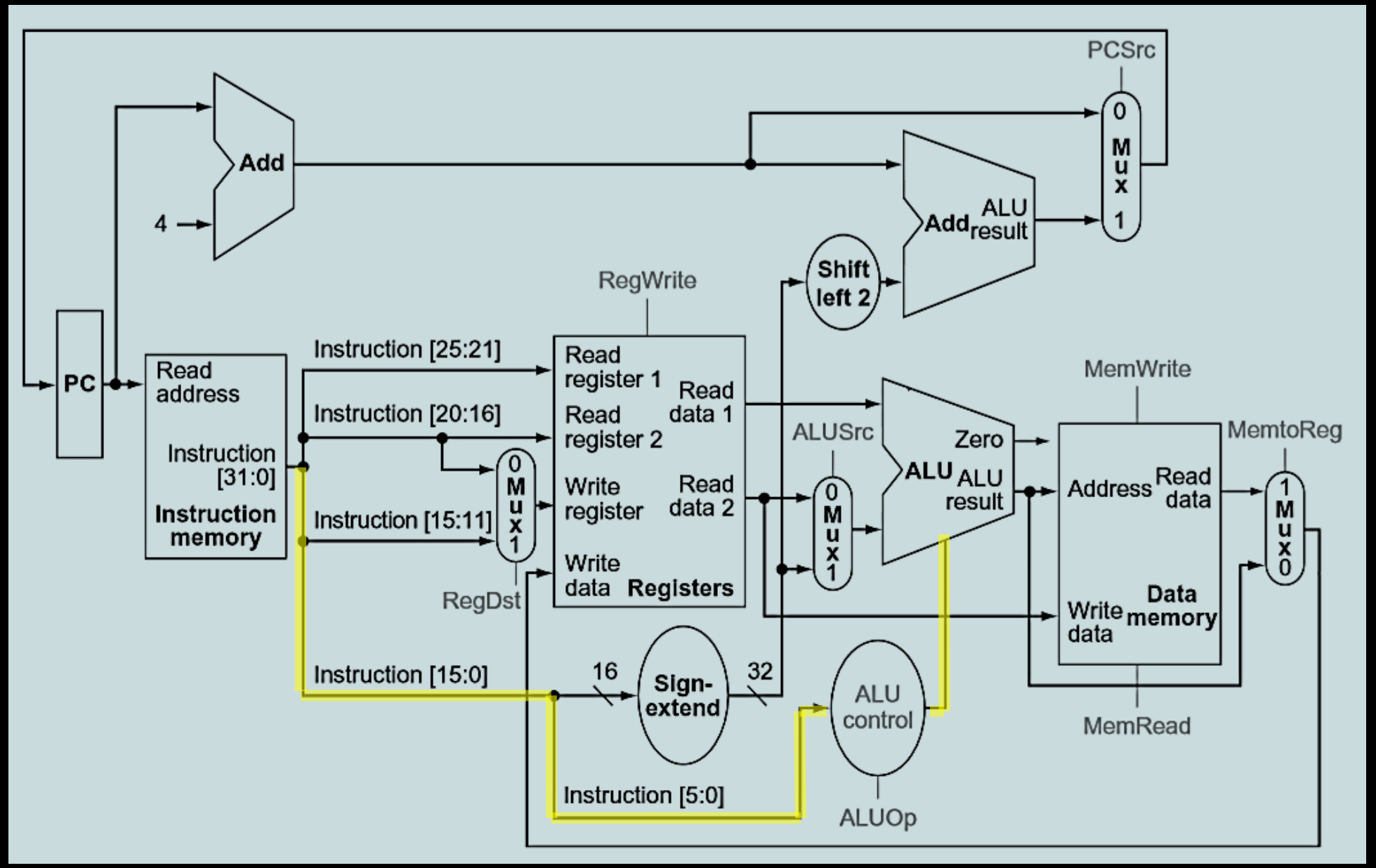
# ALU CONTROL LINES

| Opcode | ALU op | Operation | Funct | ALU action | ALU Control Input |
|---|---|---|---|---|---|
| lw | 00 | Load word | N/A | add | 0010 |
| sw | 00 | Store word | N/A | add | 0010 |
| beq | 01 | Branch equal | N/A | subtract | 0110 |
| R-type | 10 | Add | 100000 | add | 0010 |
| R-type | 10 | Subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | AND | 0000 |
| R-type | 10 | OR | 100101 | OR | 0001 |
| R-type | 10 | Set on less than | 101010 | slt | 0111 |



Here, we have modified the datapath to work with every instruction except the jump instruction.

Notice the added element for determining the ALU control input from the funct field for R-types.



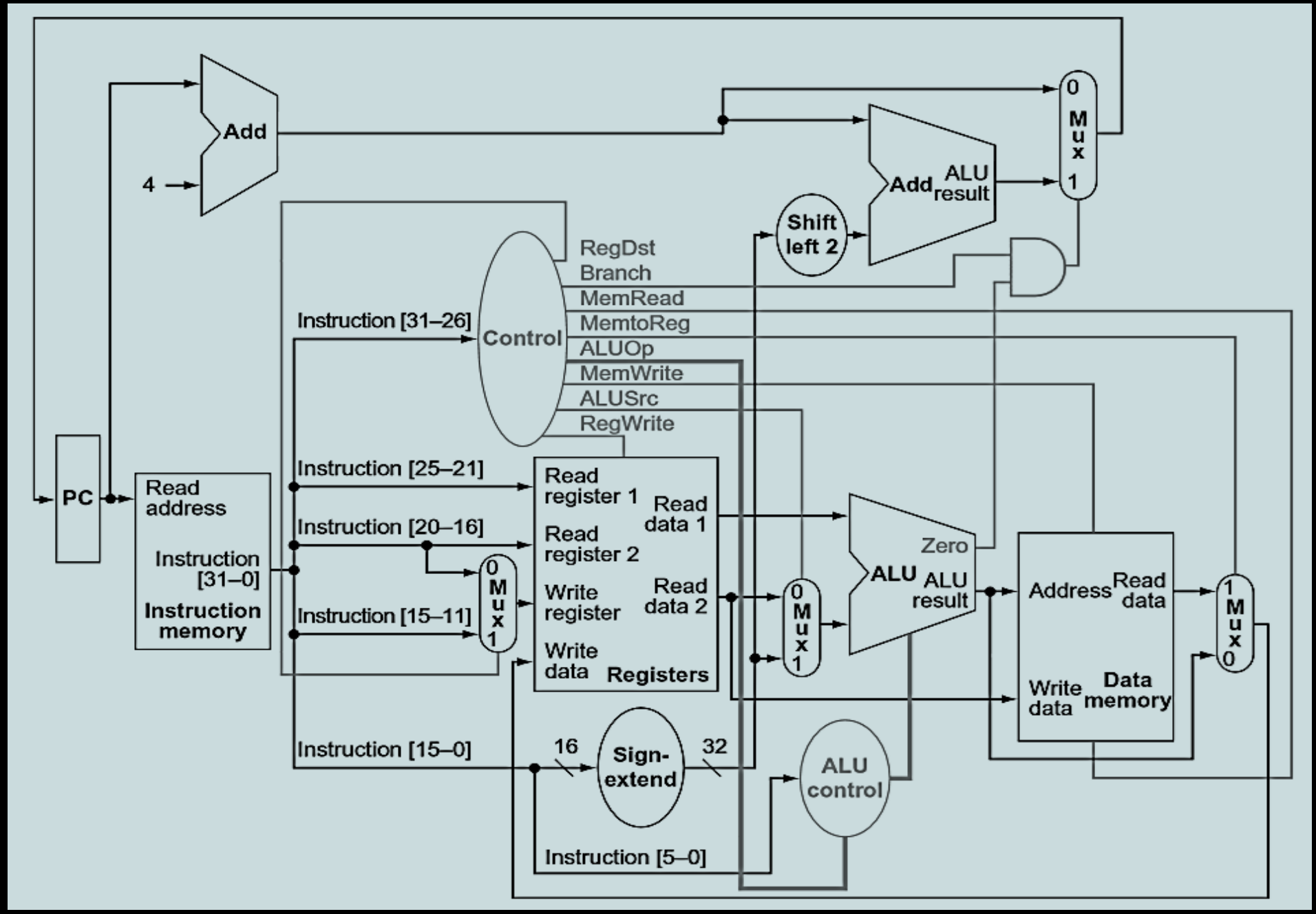
# CONTROL SIGNALS

As we can see from the previous slide, we also need to use the instruction to set control signals other than the ALU.

| Signal Name | Effect when not set | Effect when set |
|---|---|---|
| RegDst | Destination register comes from rt field. | Destination register comes from the rd field. |
| RegWrite | None. | Write Register is written to with Write Data. |
| ALUSrc | Second ALU operand is Read Data 2. | Second ALU operand is immediate field. |
| PCSrc | PC → PC + 4 | PC → Branch target |
| MemRead | None. | Contents of Address input are copied to Read Data. |
| MemWrite | None. | Write Data is written to Address. |
| MemToReg | Value of register Write Data is from ALU. | Value of register Write Data is memory Read Data. |

Here, we have modified the datapath to work with every instruction except the jump instruction.

Notice how most of the control decisions can be decided using only the upper 6 bits of the instruction (opcode).



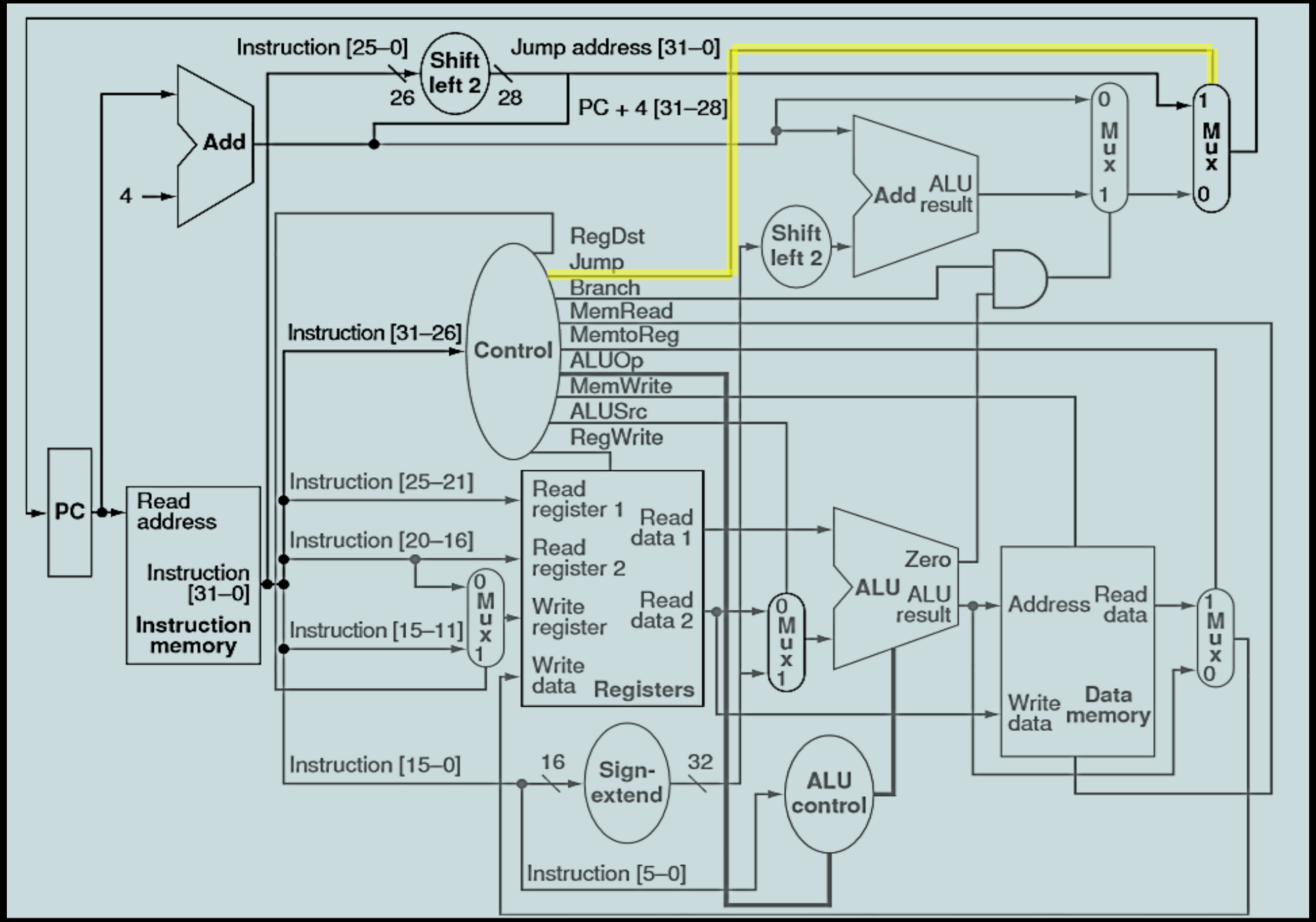
# CONTROL SIGNALS

From the previous slide, we can see that the control signals are chosen based on the upper 6 bits of the instruction. That is, the *opcode* is used to set the control lines.

| Instr. | RegDst | ALUSrc | MemToReg | RegWrite | MemRead | MemWrite | Branch | ALUOp1 | ALUOp2 |
|---|---|---|---|---|---|---|---|---|---|
| R | 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 |

Furthermore, as we saw before, the ALU control input lines are also dictated by the funct fields of applicable instructions.

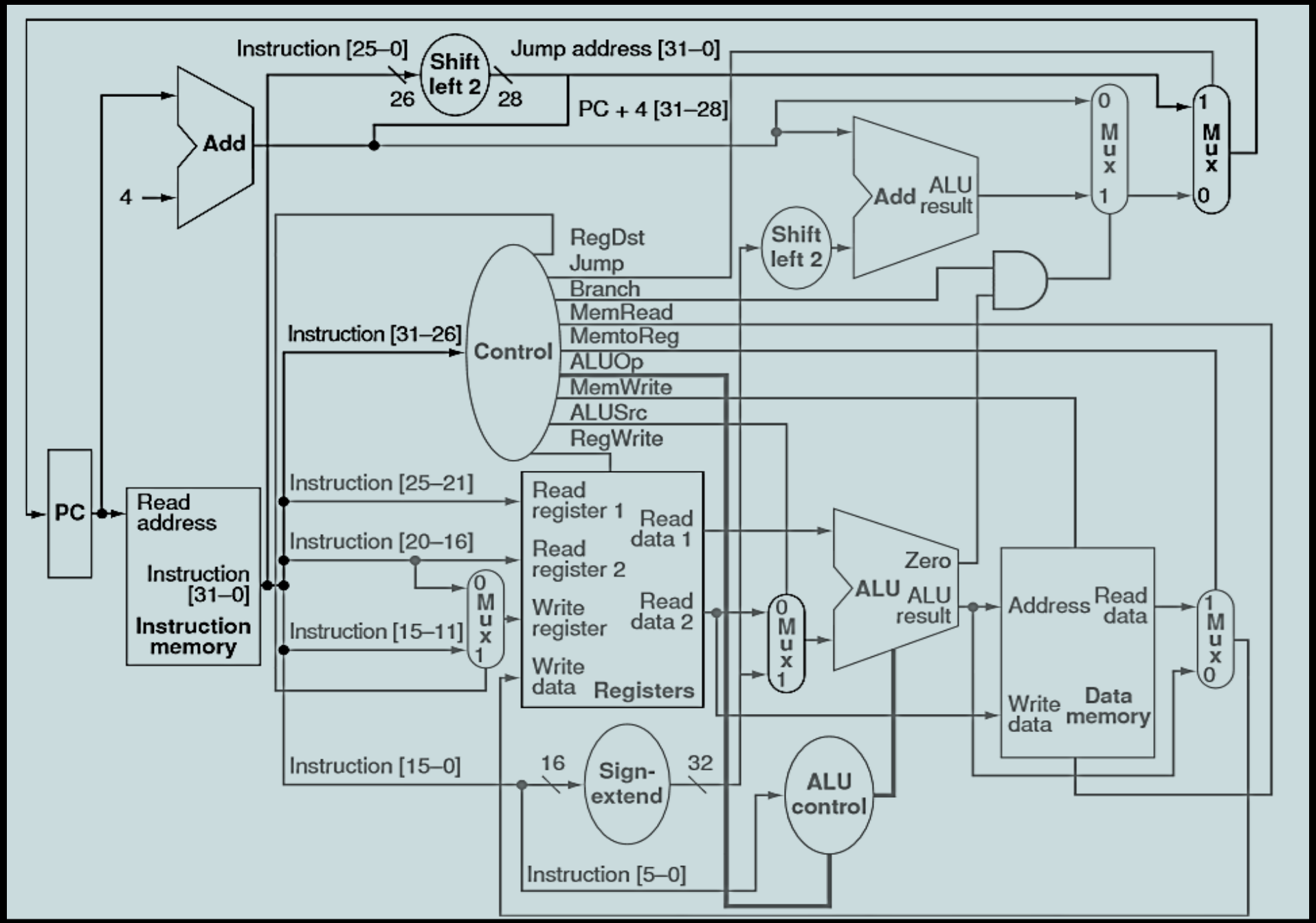
Here we add in an additional control line for jump instructions.



Quiz Time!

What are the relevant datapath lines for the add instruction and what are the values of each of the control lines?

```
add $rd, $rs, $rt
```

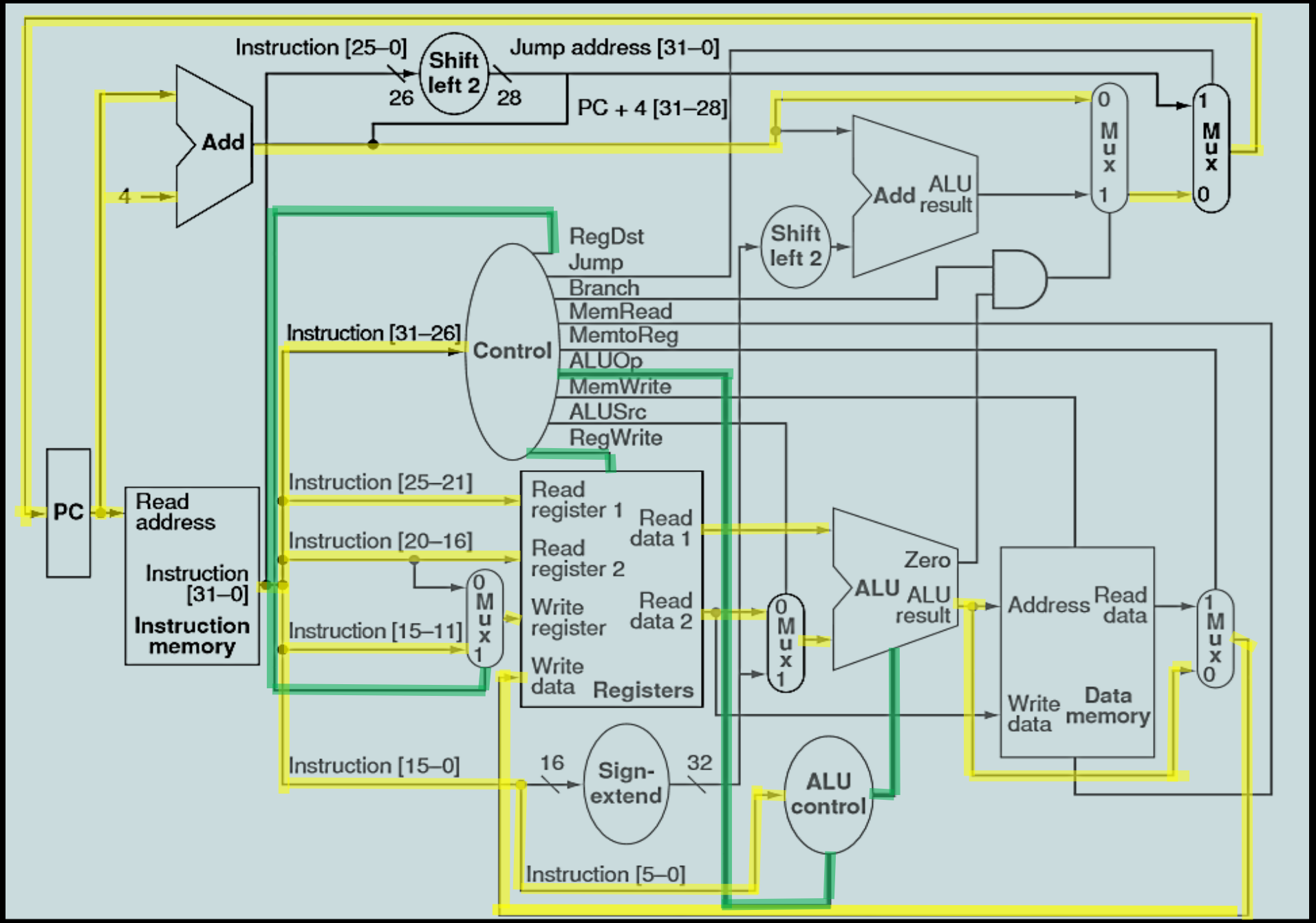


Quiz Time!

What are the relevant datapath lines for the add instruction and what are the values of each of the control lines?

```
add $rd, $rs, $rt
```

Datapath shown in yellow. Relevant control line assertions in green.

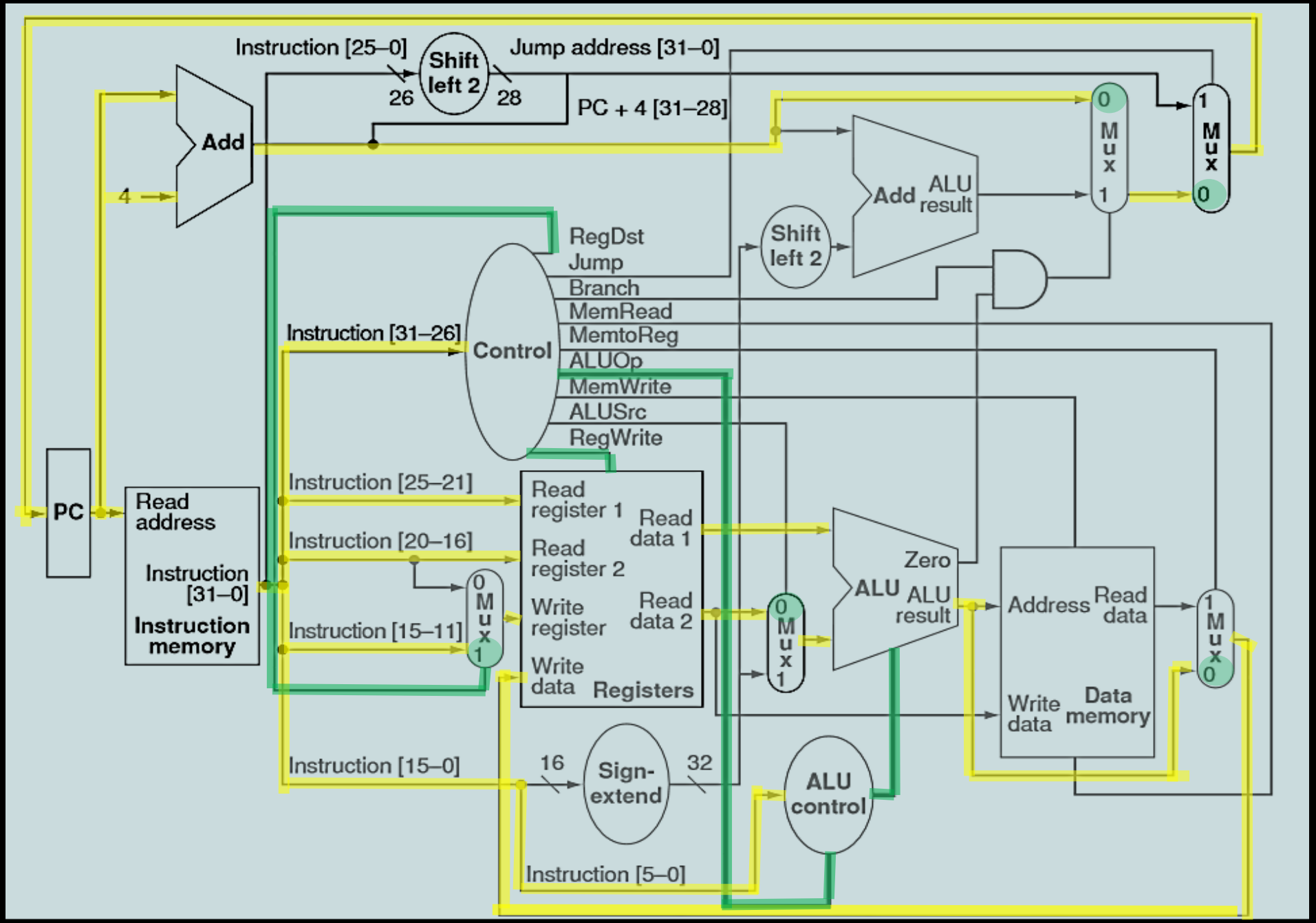


Quiz Time!

What are the relevant datapath lines for the add instruction and what are the values of each of the control lines?

```
add $rd, $rs, $rt
```

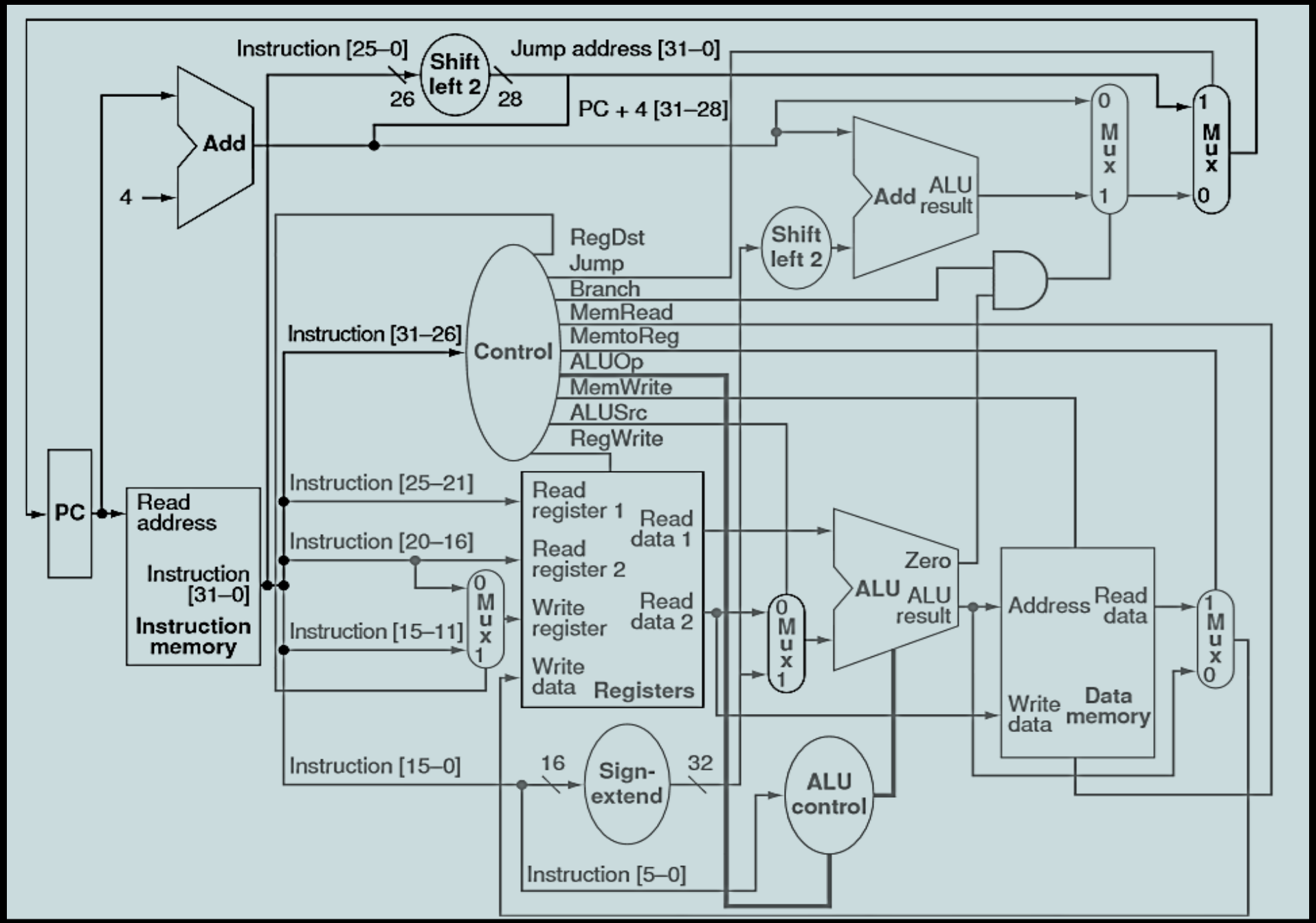
Datapath shown in yellow. Relevant control line assertions in green.



Quiz Time!

What are the relevant datapath lines for the beq instruction and what are the values of each of the control lines?

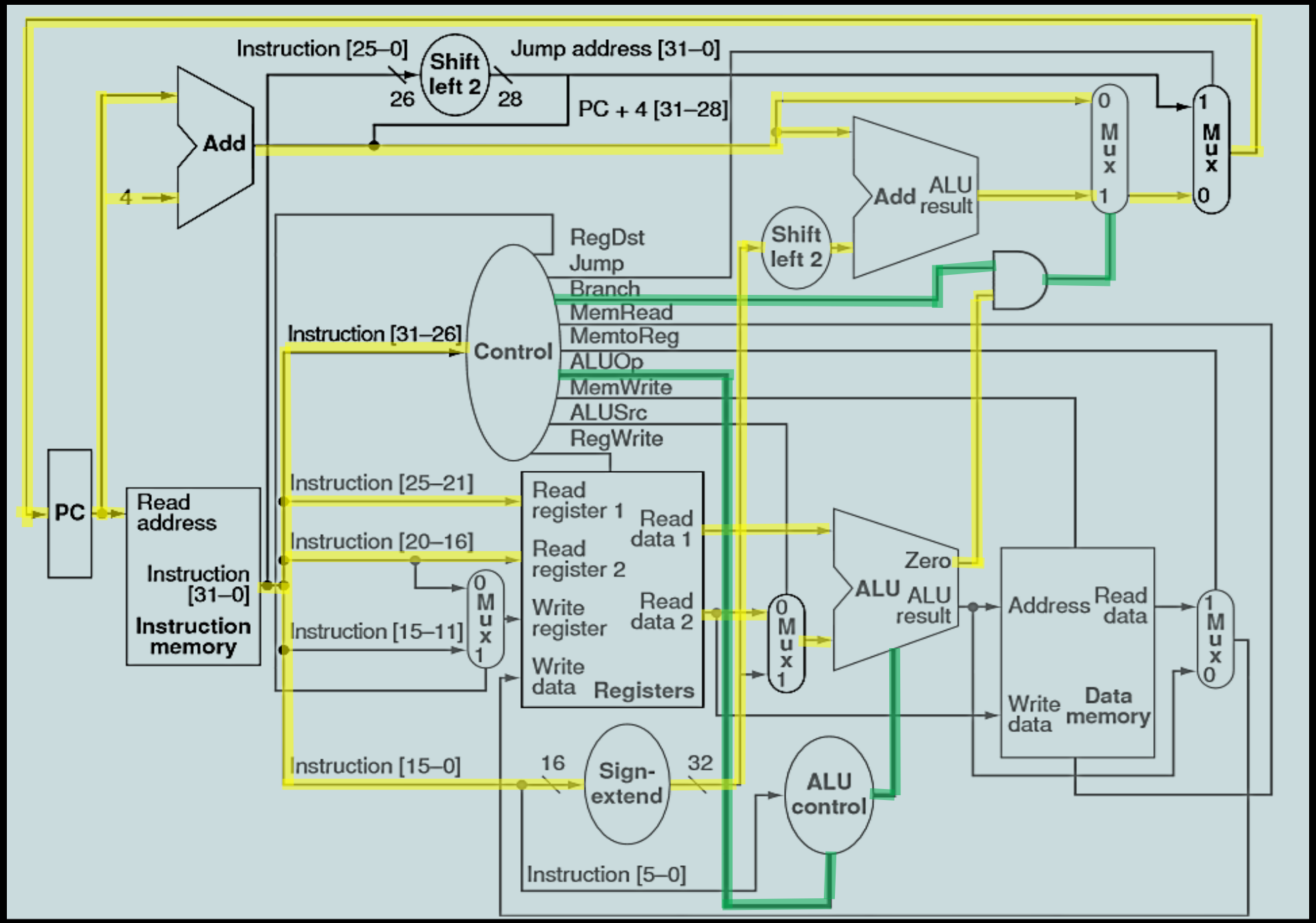
```
beq $rs, $rt, imm
```



Quiz Time!

What are the relevant datapath lines for the beq instruction and what are the values of each of the control lines?

```
beq $rs, $rt, imm
```



# RELATIVE CYCLE TIME

What is the longest path (slowest instruction) assuming 4ns for instruction and data memory, 3ns for ALU and adders, and 1ns for register reads or writes? Assume negligible delays for muxes, control unit, sign extend, PC access, shift left by 2, routing, etc

| Type | Instruction Memory | Register Read | ALU Operation | Data Memory | Register Write | Total |
|---|---|---|---|---|---|---|
| R-format | 4 | 1 | 3 | 0 | 1 | 9 |
| lw | 4 | 1 | 3 | 4 | 1 | 13 |
| sw | 4 | 1 | 3 | 4 | 0 | 12 |
| beq | 4 | 1 | 3 | 0 | 0 | 8 |
| j | 4 | 0 | 0 | 0 | 0 | 4 |

# SINGLE-CYCLE IMPLEMENTATION

The advantage of single cycle implementation is that it is simple to implement.

Disadvantages

- The clock cycle will be determined by the longest possible path, which is not the most common instruction. This type of implementation violates the idea of making the common case fast.
- May be wasteful with respect to area since some functional units, such as adders, must be duplicated since they cannot be shared during a single clock cycle