Single-cycle Datapath and Control

Processors

- 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

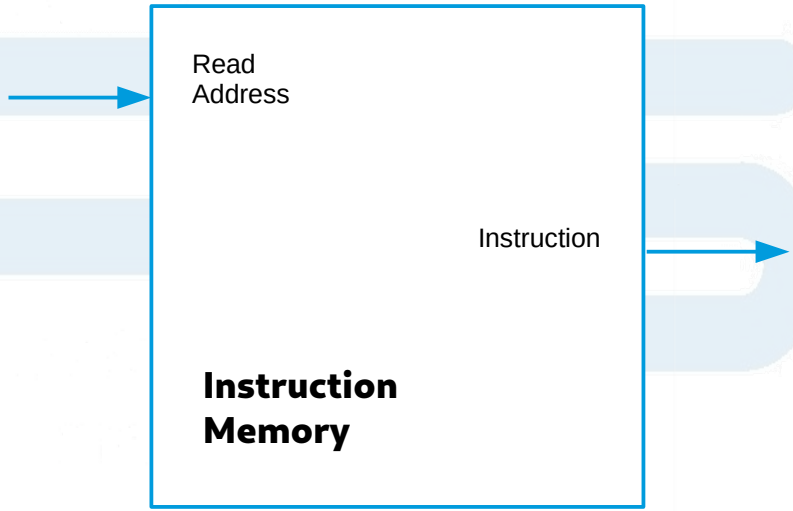
Datapath

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

- Code can also be written, e.g., self-modifying code|



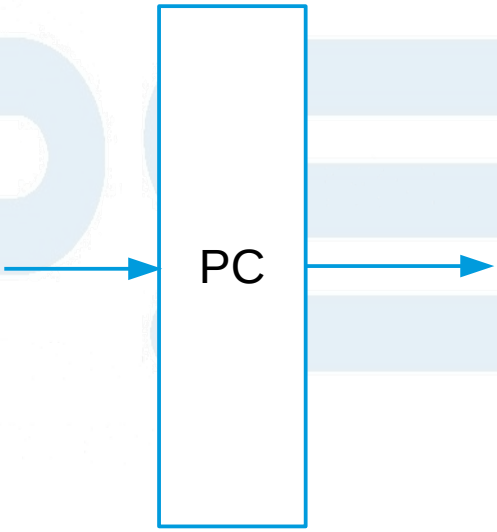
Datapath

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.

- Normally PC increments sequentially except for branch instructions

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

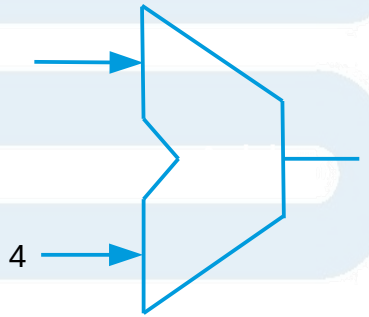


Datapath

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.



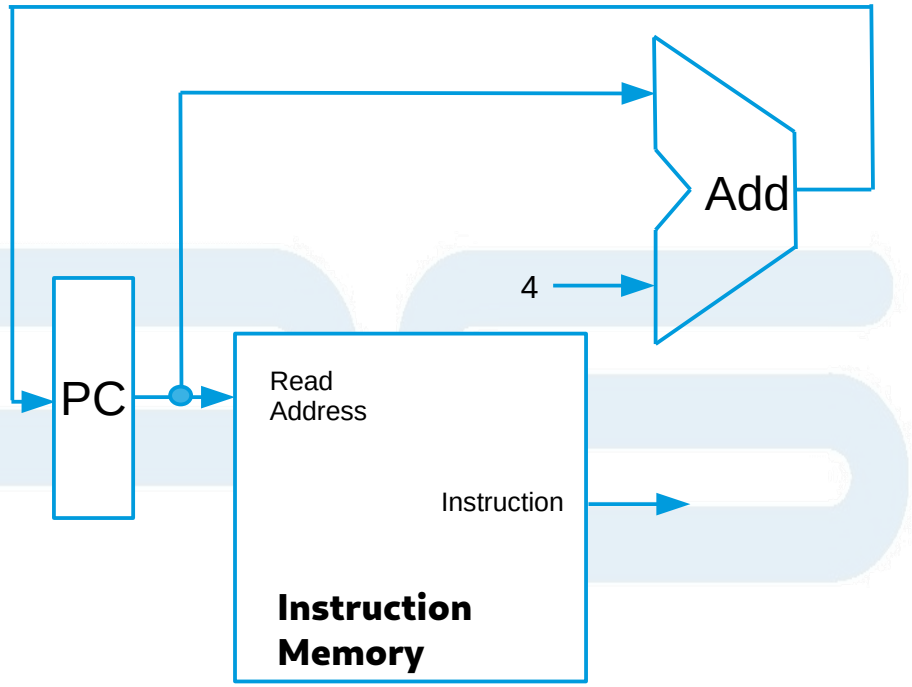
Datapath

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

- Instruction fetching: use the address in the PC to fetch the current instruction from instruction memory
- Instruction decoding: determine the fields within the instruction
- Instruction execution: perform the operation indicated by the instruction
- Update the PC to hold the address of the next instruction

Datapath

- Fetch the instruction at the address of the PC
- Decode the instruction
- Execute the instruction
- Update the PC to hold the address of the next instruction



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

MIPS instruction formats

- Different MIPS instruction formats
  - R: register, register, register
  - I: immediate, lw, sw
  - B: branches
  - J: jump

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

Name	Fields					
Field size	6	5	5	5	5	6
R format	opcode	rs	rt	rd	shamt	funct

opcode – instruction operation code

rs – first register operand

rt – second register operand

rd – register destination operand

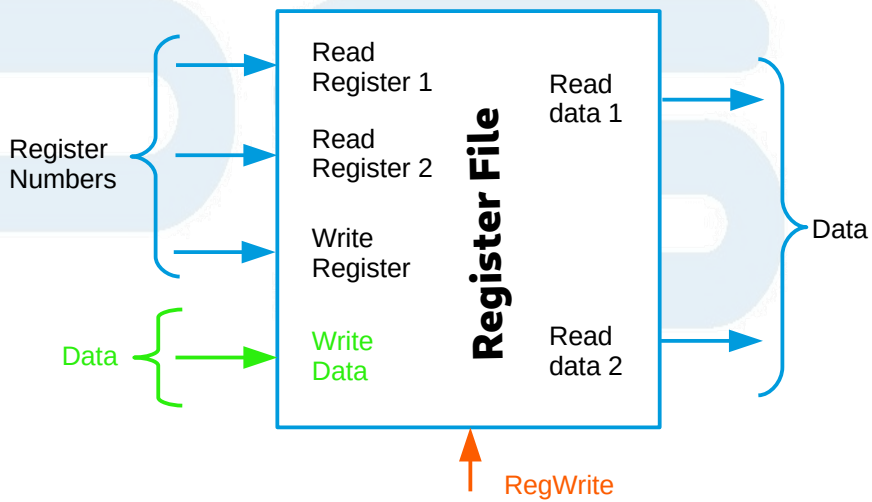
funct – additional opcodes

shamt – shift amount

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

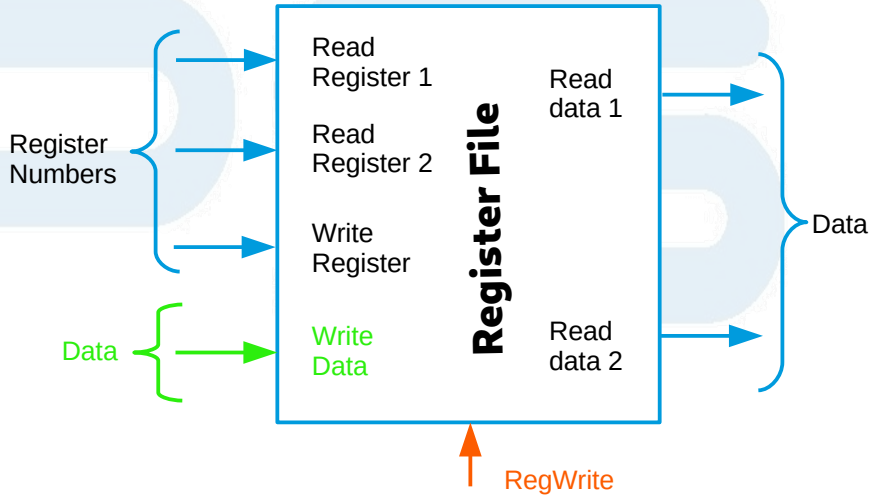


Datapath

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

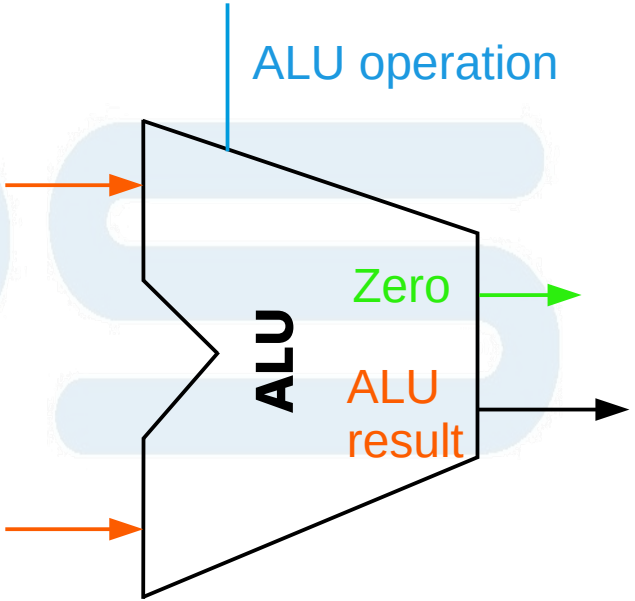
To actually execute R-format instructions, we need to include the ALU element.

The ALU performs the operation indicated by the instruction.

It takes two 32 bits operands, as well as a 4-bit wide operation selector value. The result of the operation is the *ALU result* 32 bits value.

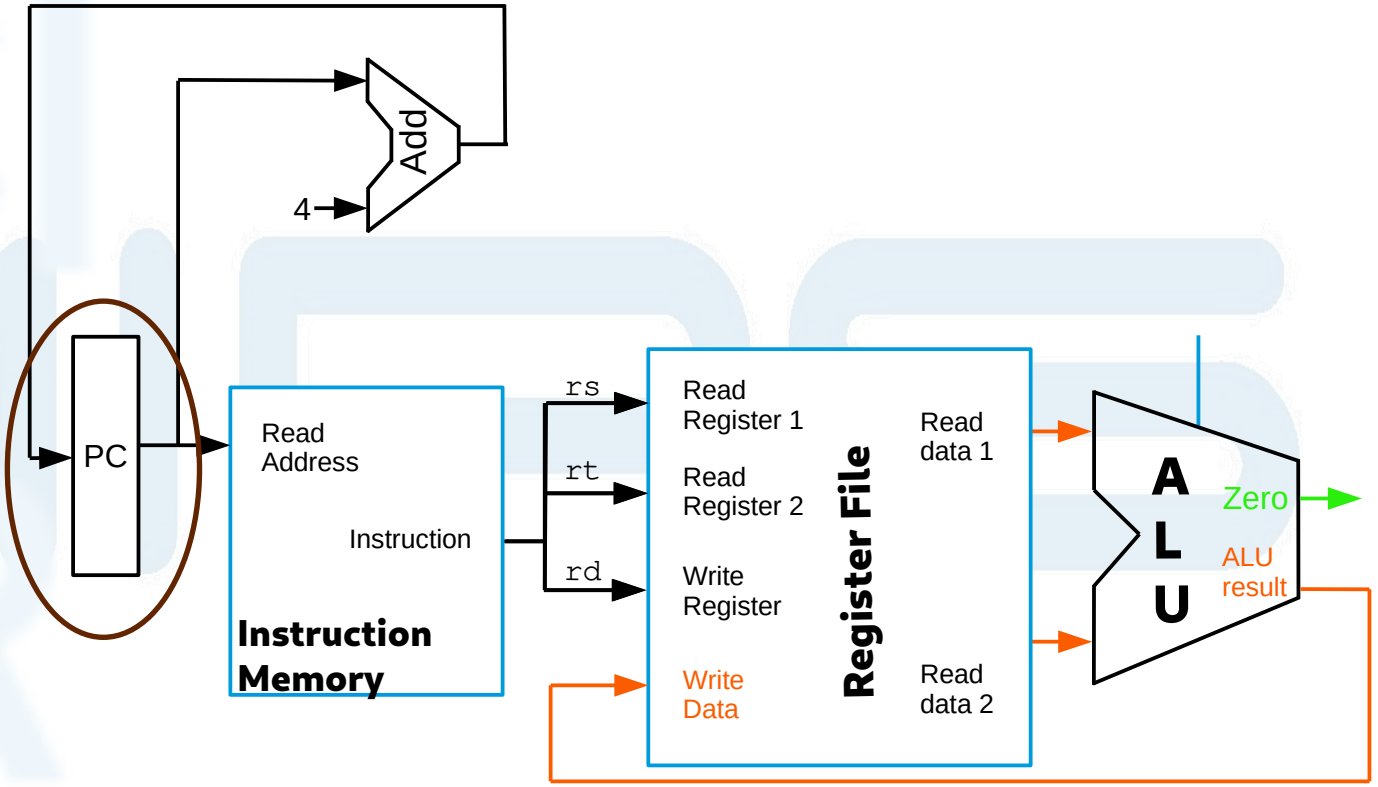
- ALU operation is a part of the control. We discuss datapath first

We have an additional (Zero) output specifically for branching – we will cover this in a minute.



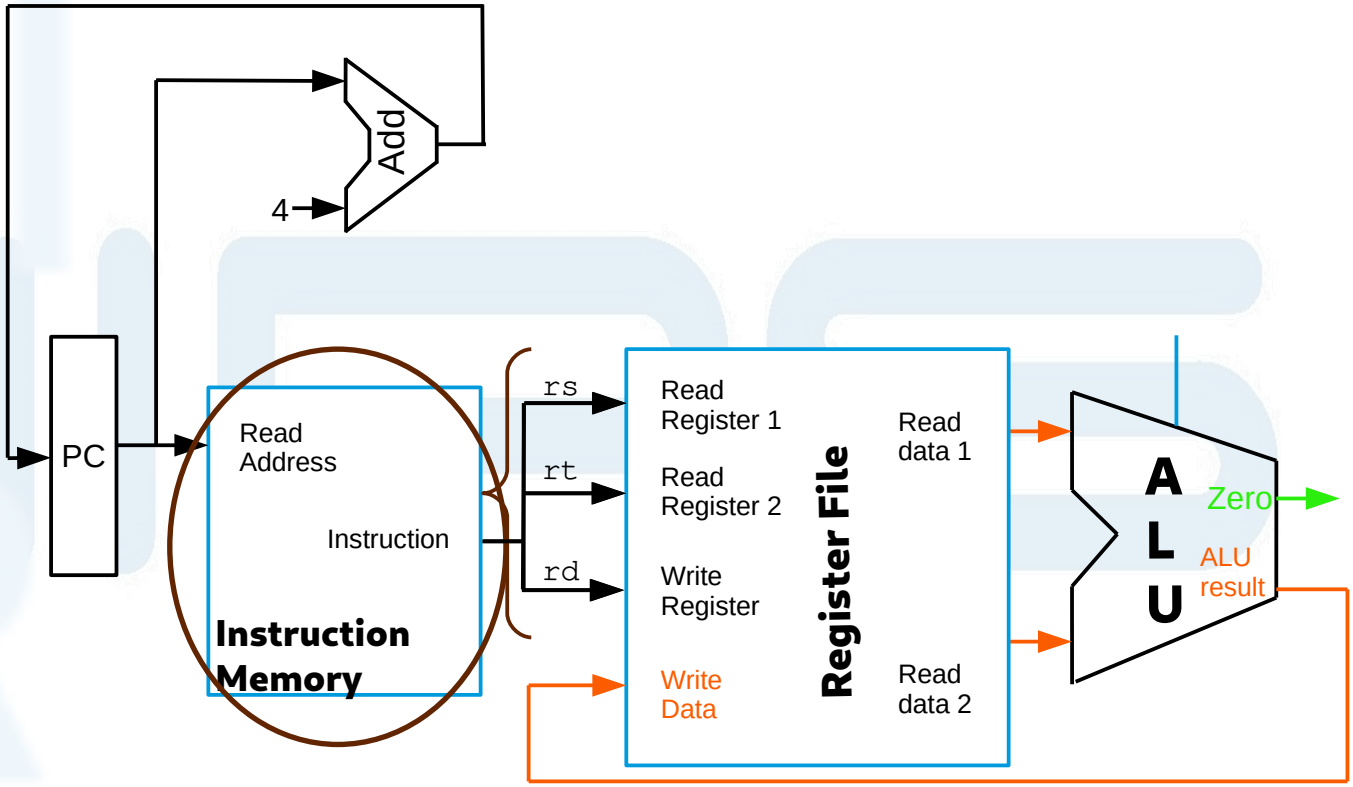
Datapath for R-format instructions

1. Grab the instruction address from PC



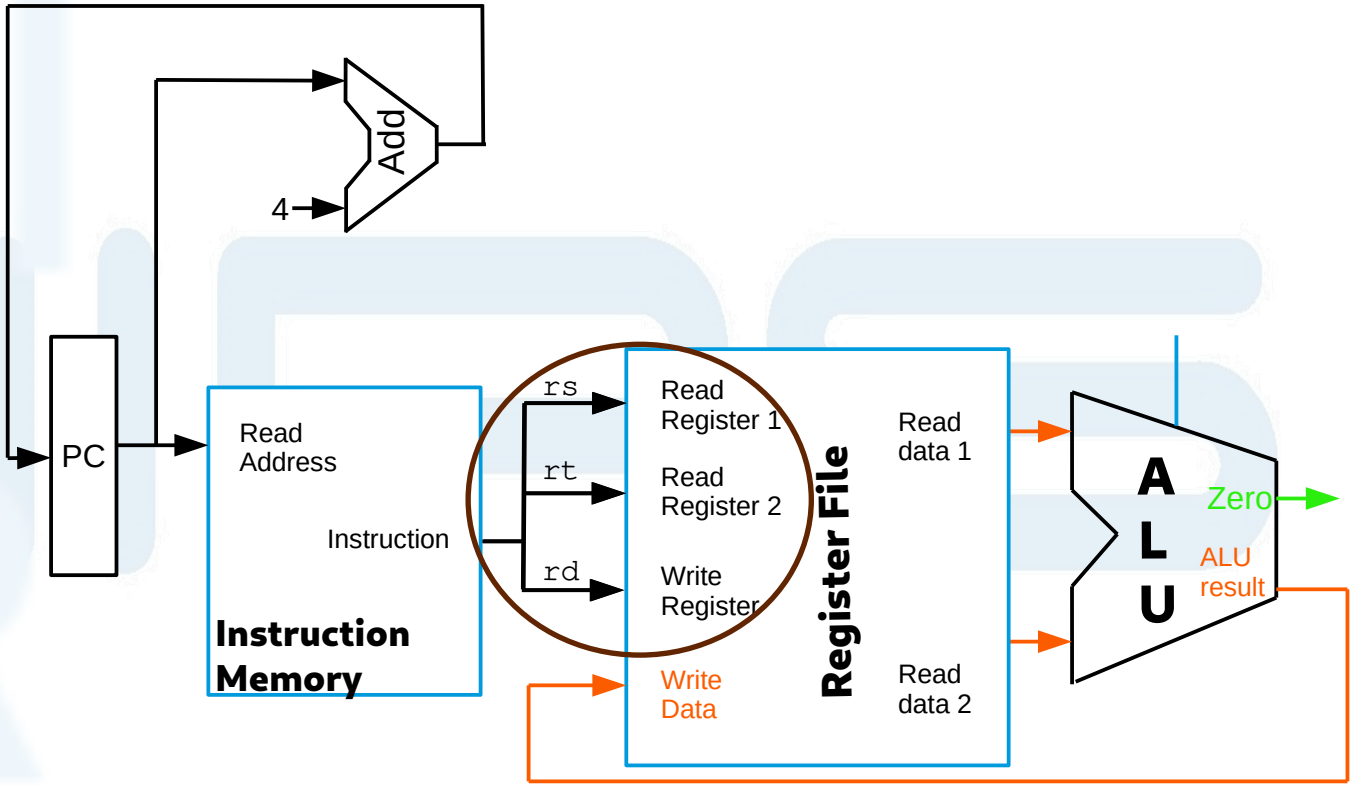
Datapath for R-format instructions

2. Fetch the instruction from instruction memory
3. Decode the instruction



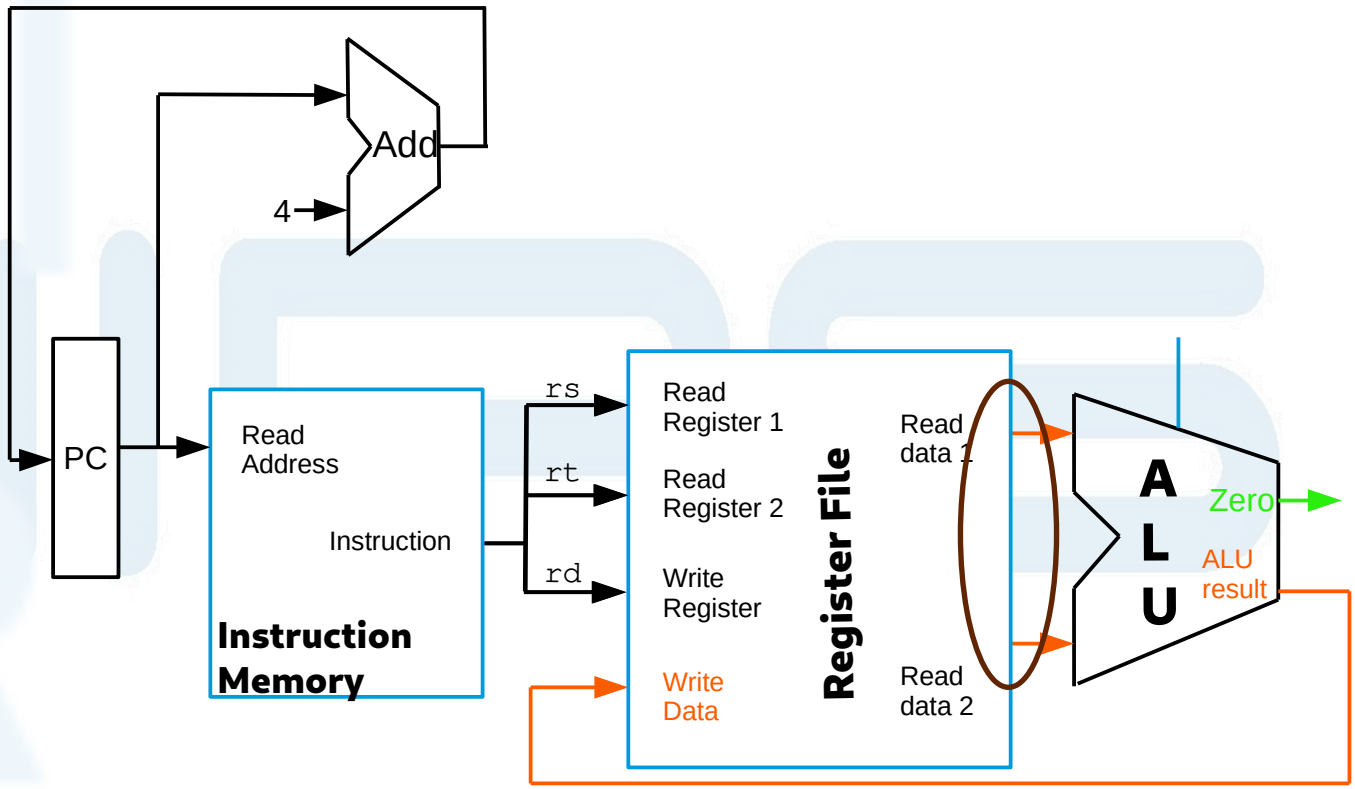
Datapath for R-format instructions

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



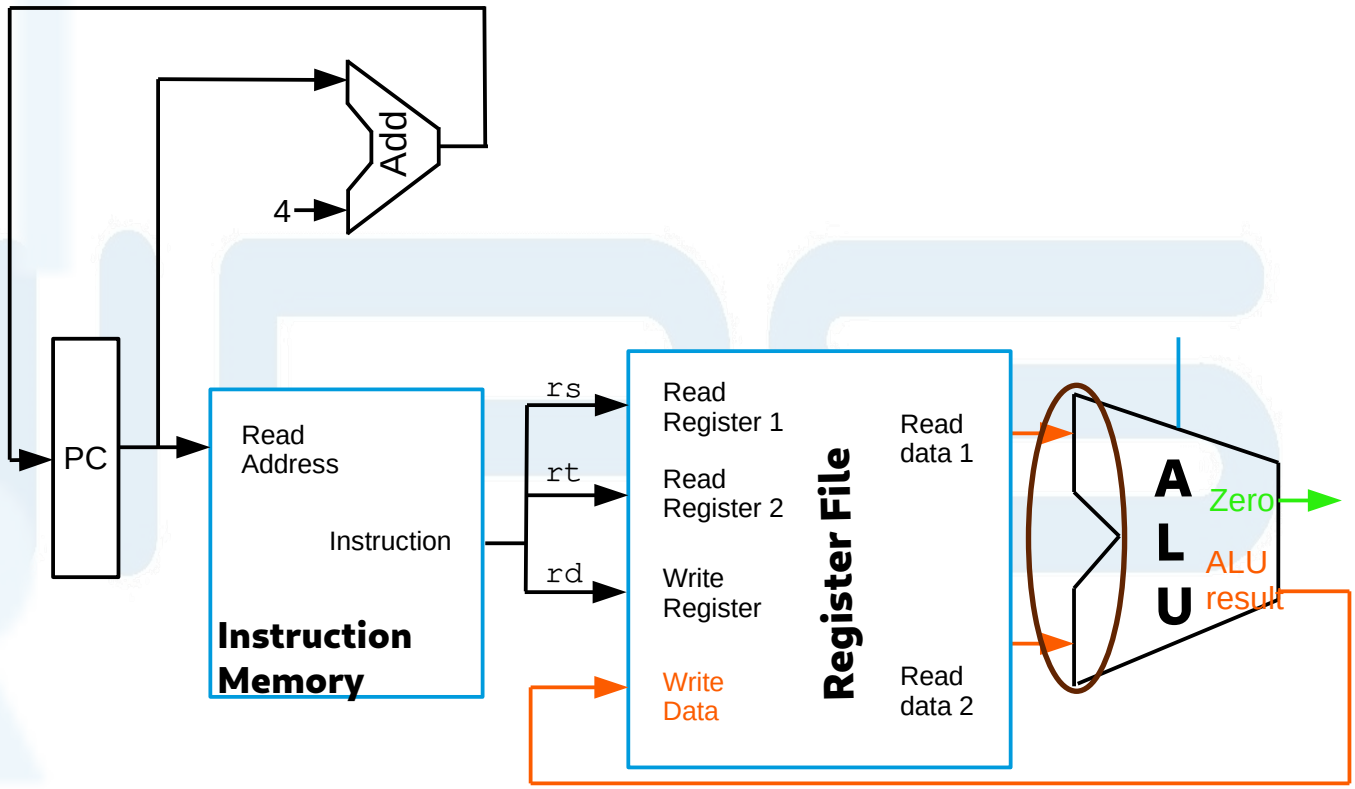
Datapath for R-format instructions

5. Retreive data from read register 1 and read register 2 (rs, rt)



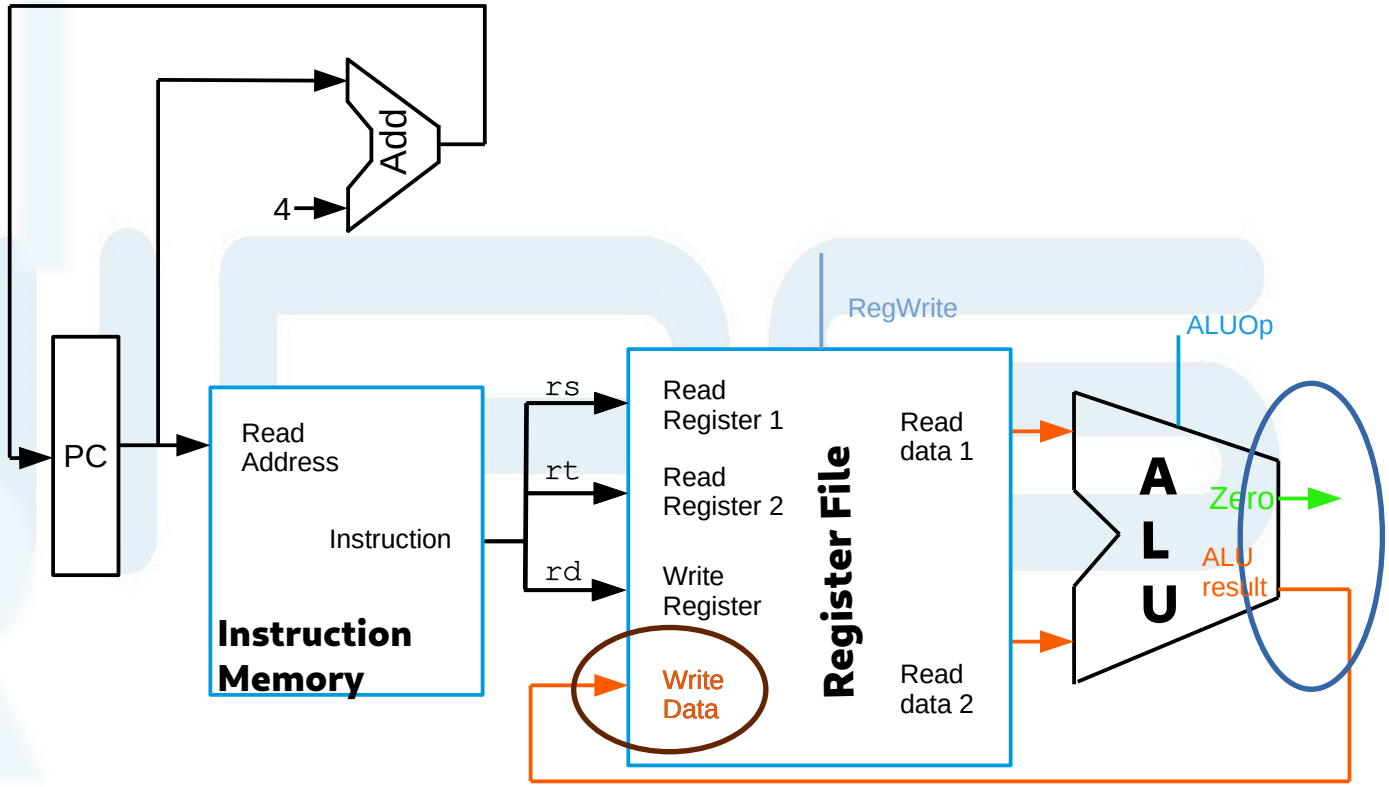
Datapath for R-format instructions

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



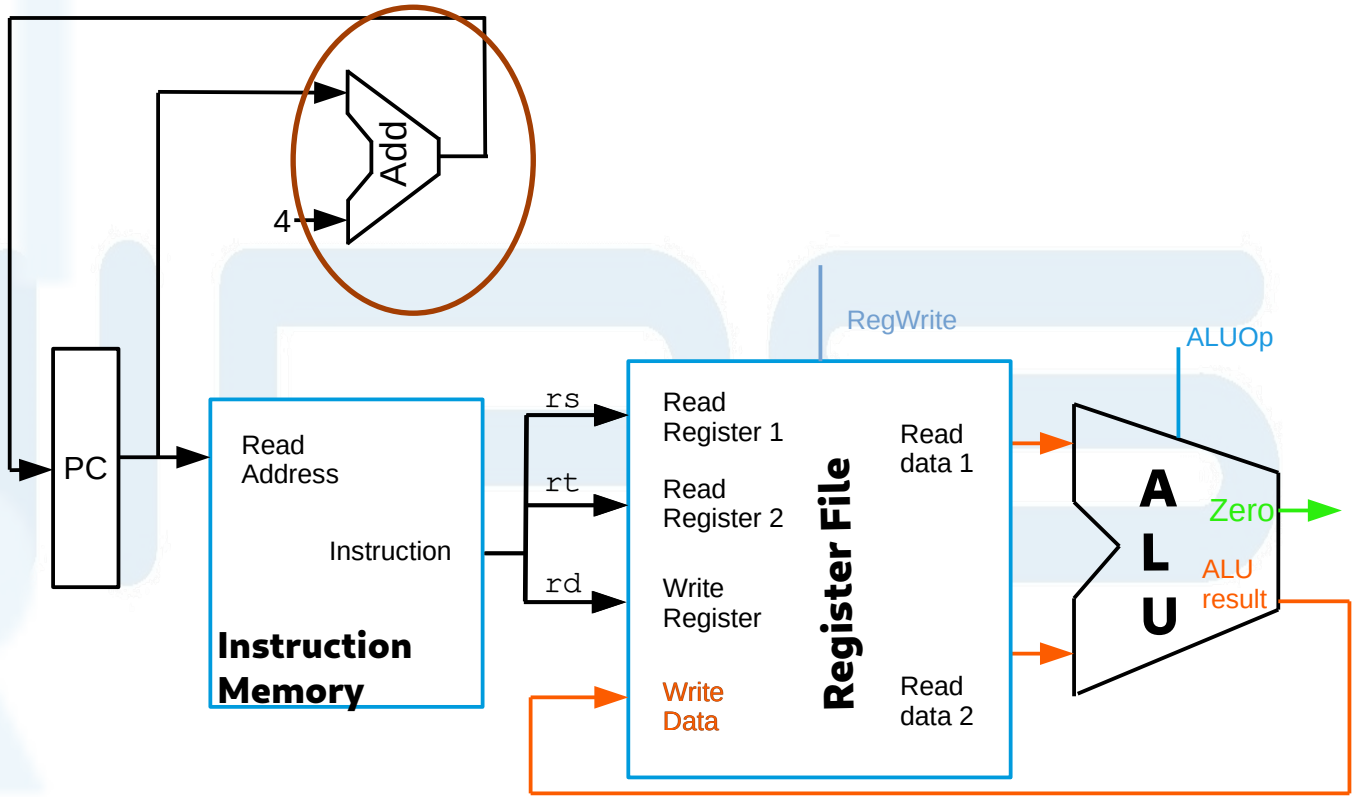
Datapath for R-format instructions

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



Datapath for R-format instructions

8. Add 4 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-type instructions. In our limited MIPS instruction set, these are lw, sw, and beq.

Name	Fields					
Field size	6	5	5	5	5	6
I format	opcode	rs	rt	immed		

opcode – instruction operation code

rs – first register operand

rd – register destination operand

immed – immediate operand, zero extended if logical operator, otherwise sign-extended

rd
opcode
rd
opcode
rd
opcode

Data transfer instructions

Let's start I-format instructions with accommodating the data transfer. For lw and sw, we have the following format:

- `lw $rd, immed($rs)`
- `sw $rd, immed($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, $rd represents the register that will be assigned the memory value
  In sw, $rd 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

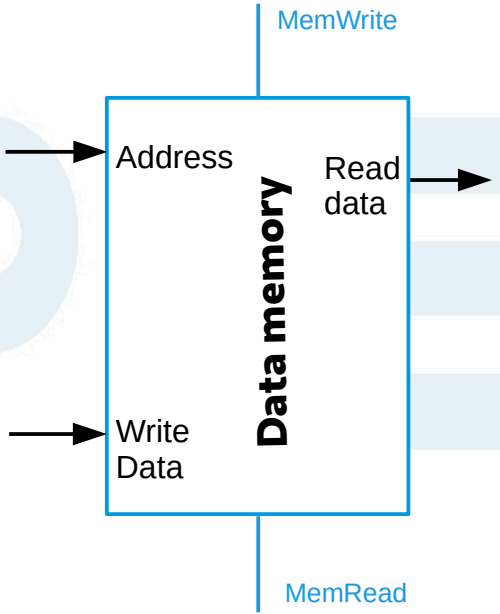
Datapath for I-format instructions

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.

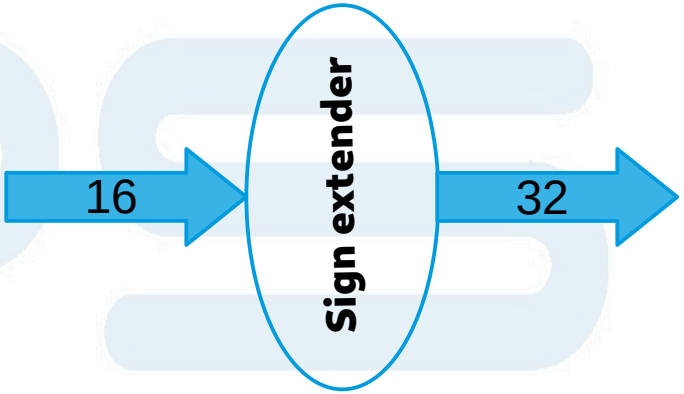


Datapath for I-format instructions

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.

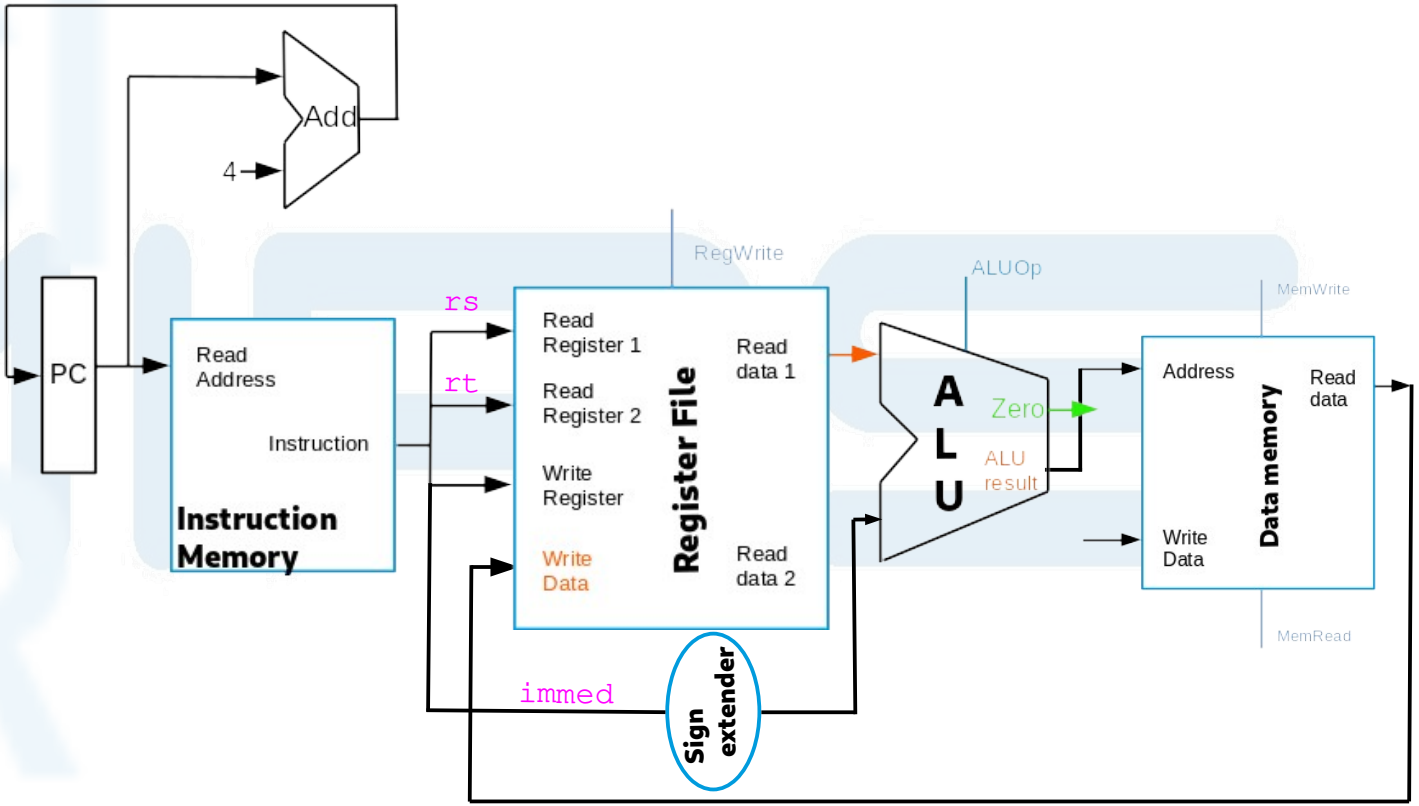


Datapath for `load word` instruction

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

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

The operands *rs*, *rt*, and *immed* have been added to the datapath for added clarity.

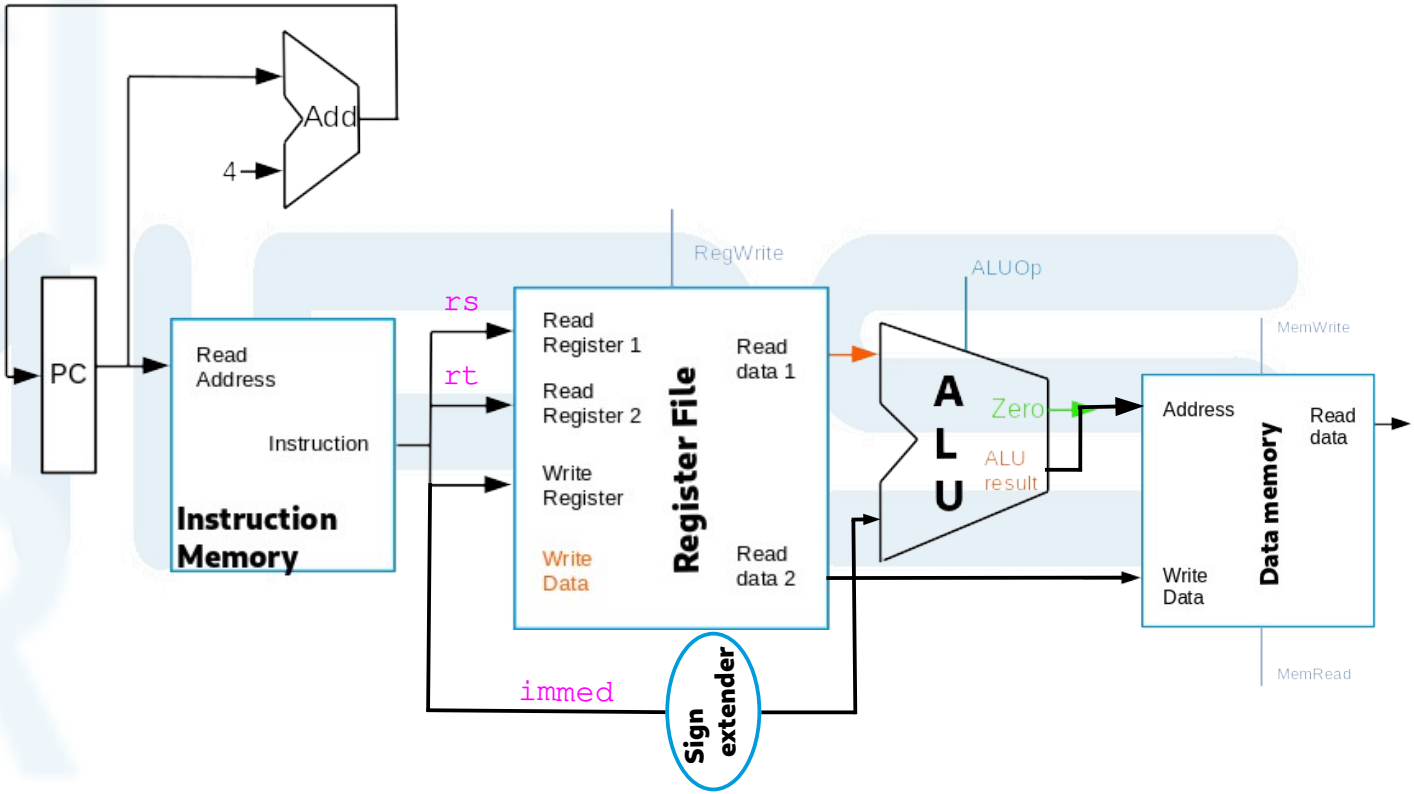


Datapath for `store word` instruction

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

`sw $rt, immed($rs)`

The operands *rs*, *rt*, and *immed* have been added to the datapath for added clarity.

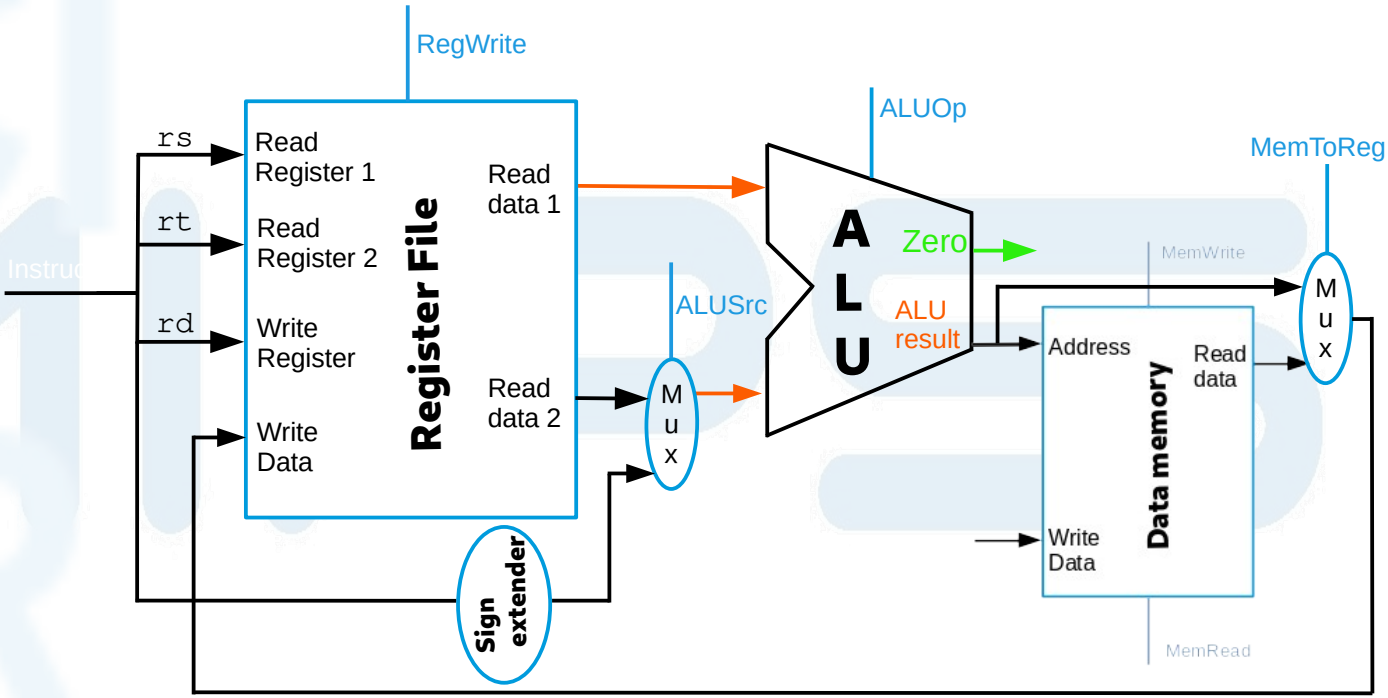


Datapath for R-format and memory access

Instructions

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

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

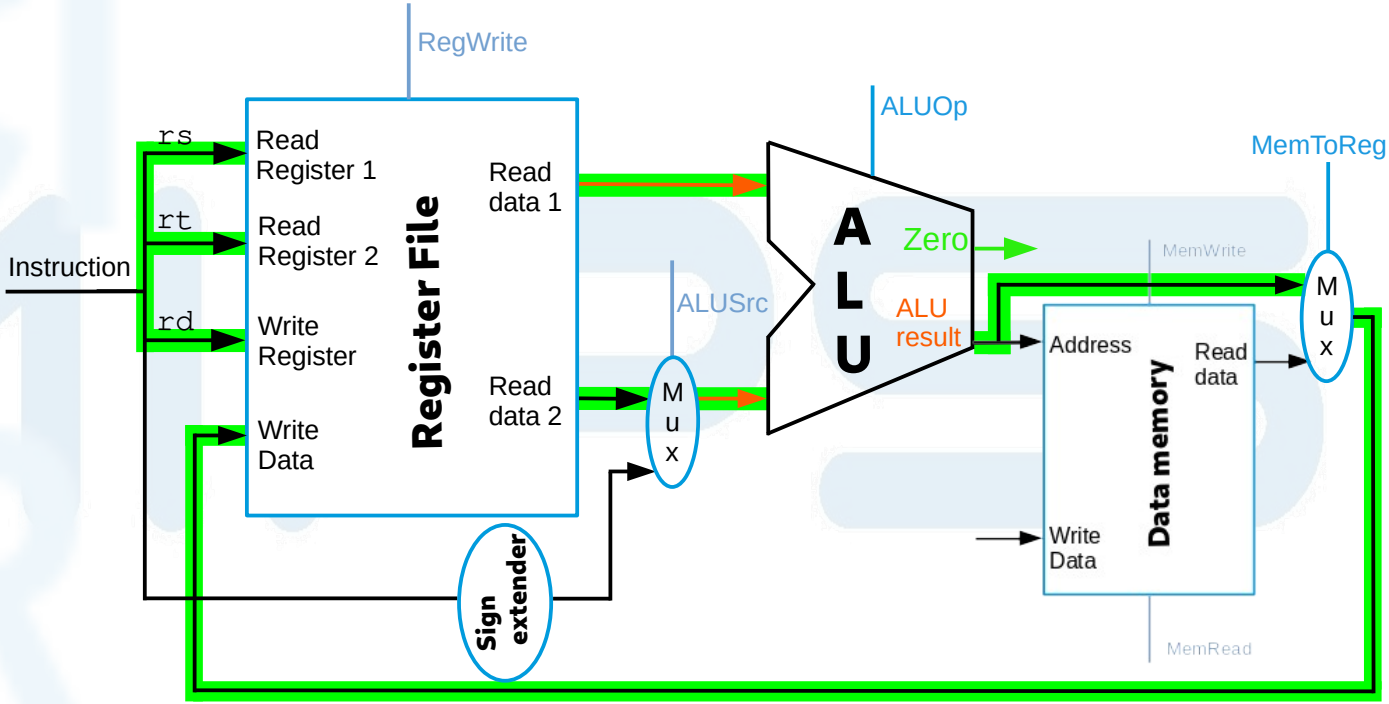


Datapath for R-format

Instructions

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

Cycles to complete: 4

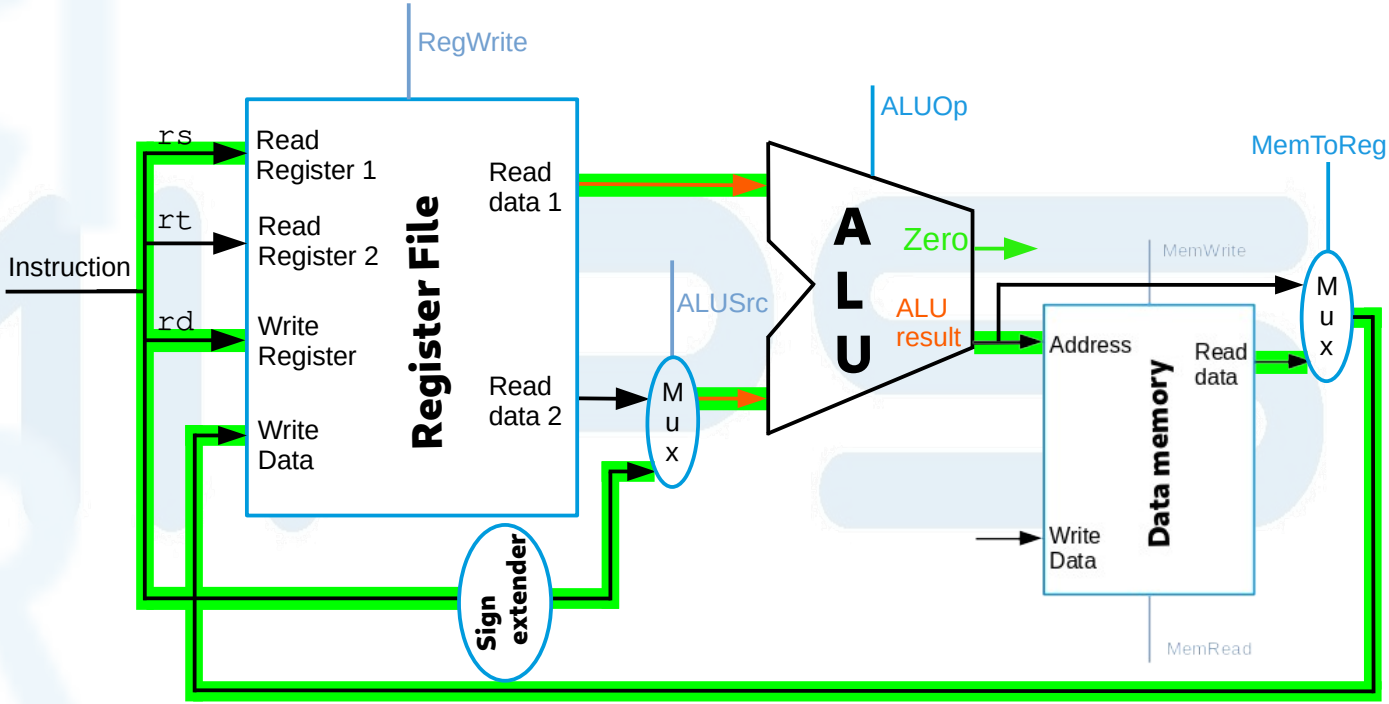


Datapath for load memory access

Instructions

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

Cycles to complete: 5

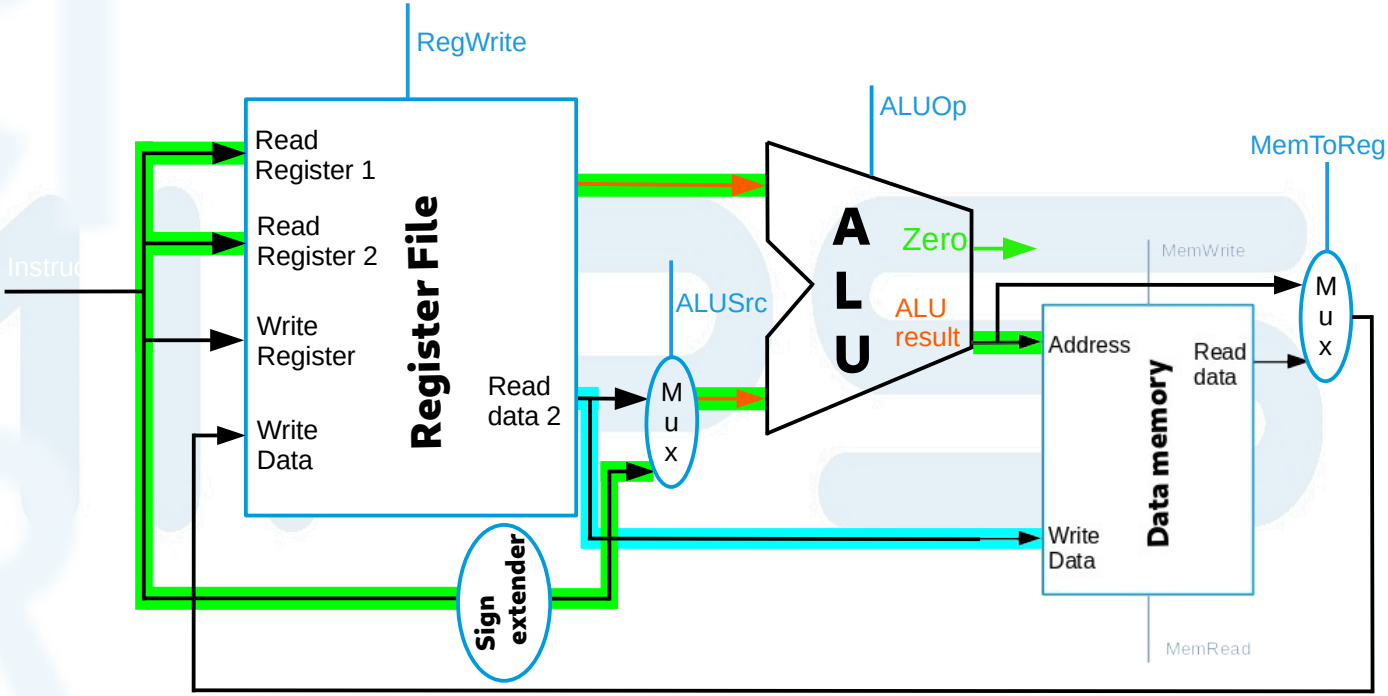


Datapath for store memory access

Instructions

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

Cycles to complete: 4



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 $rs, $rt, target
```

This instruction compares the contents of $rs and $rt 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	5	5	5	5	6
I format	opcode	rs	rt	immediate		

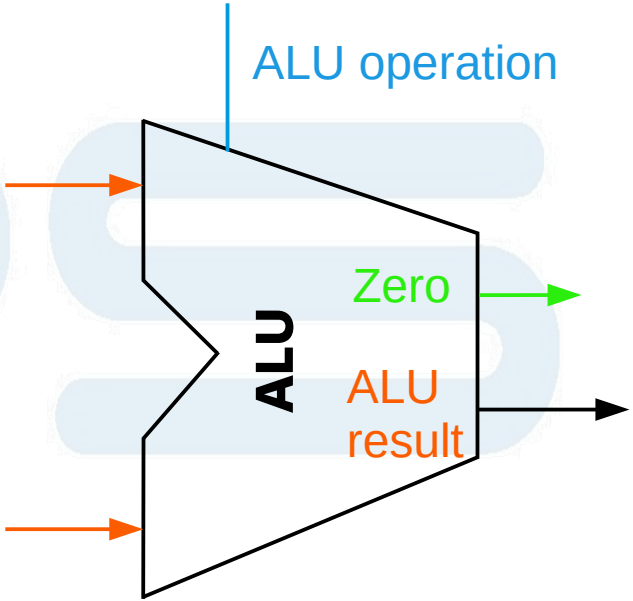
Branching instructions

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

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

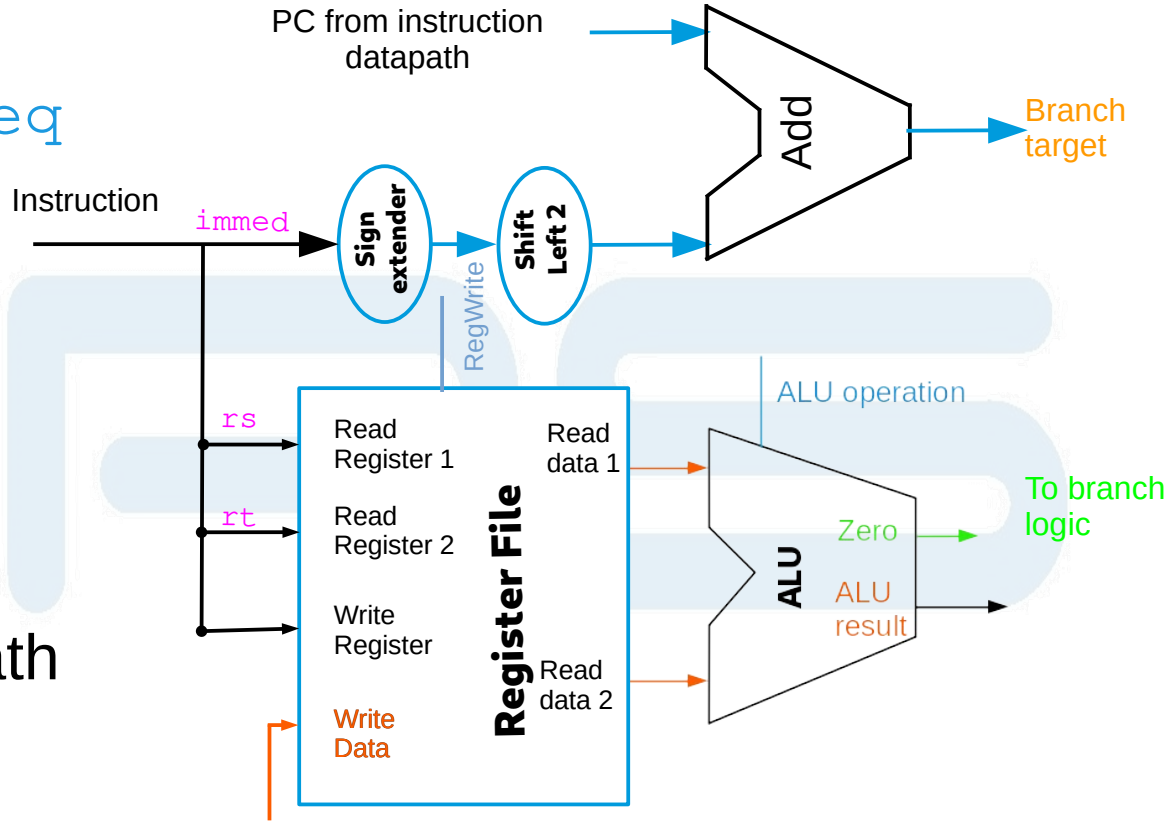


Datapath for BEQ

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

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

The registers *rs*, *rt* and *immed* have been added to the datapath for added clarity.

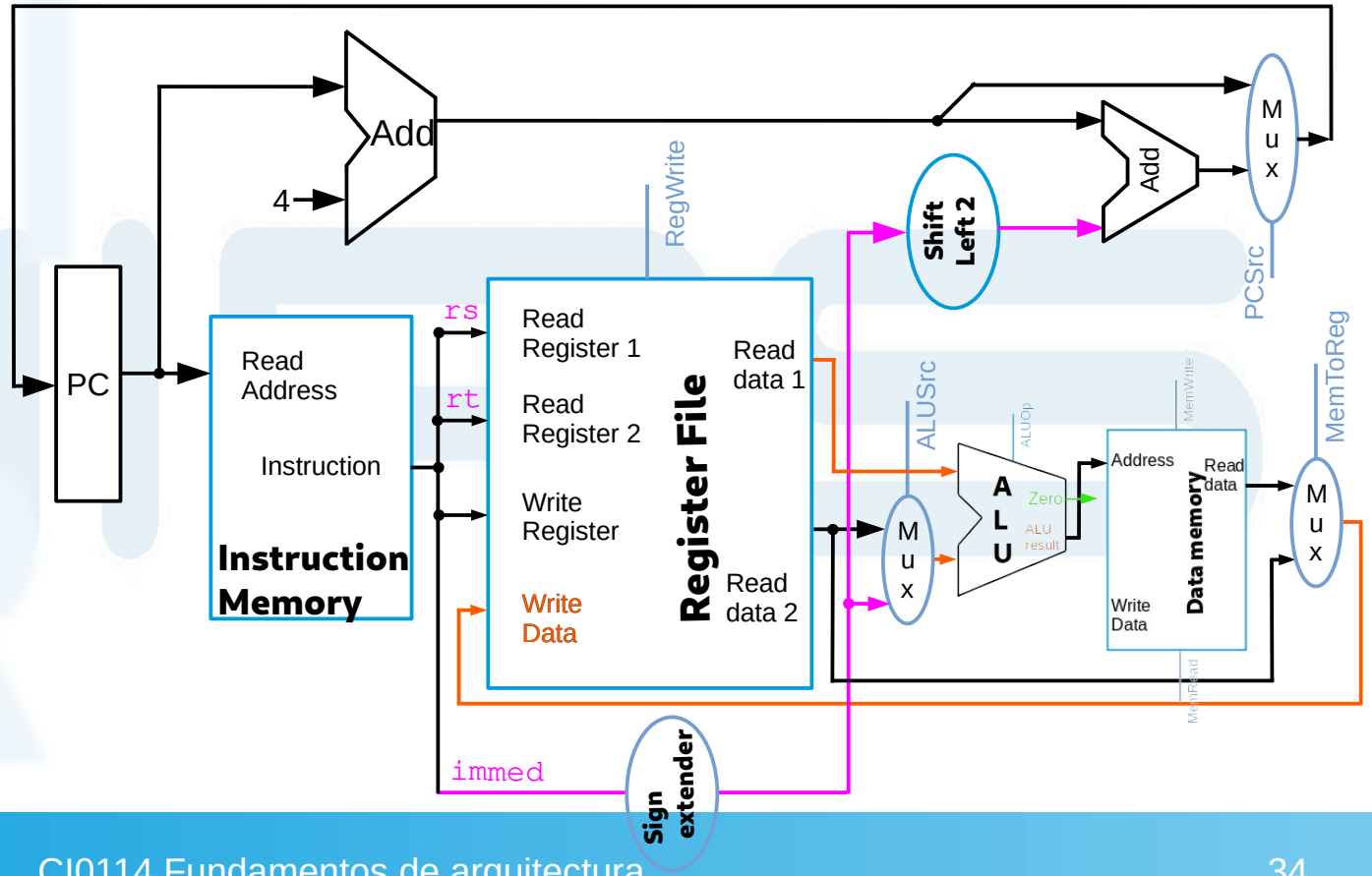


Datapath for beq instructions

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

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

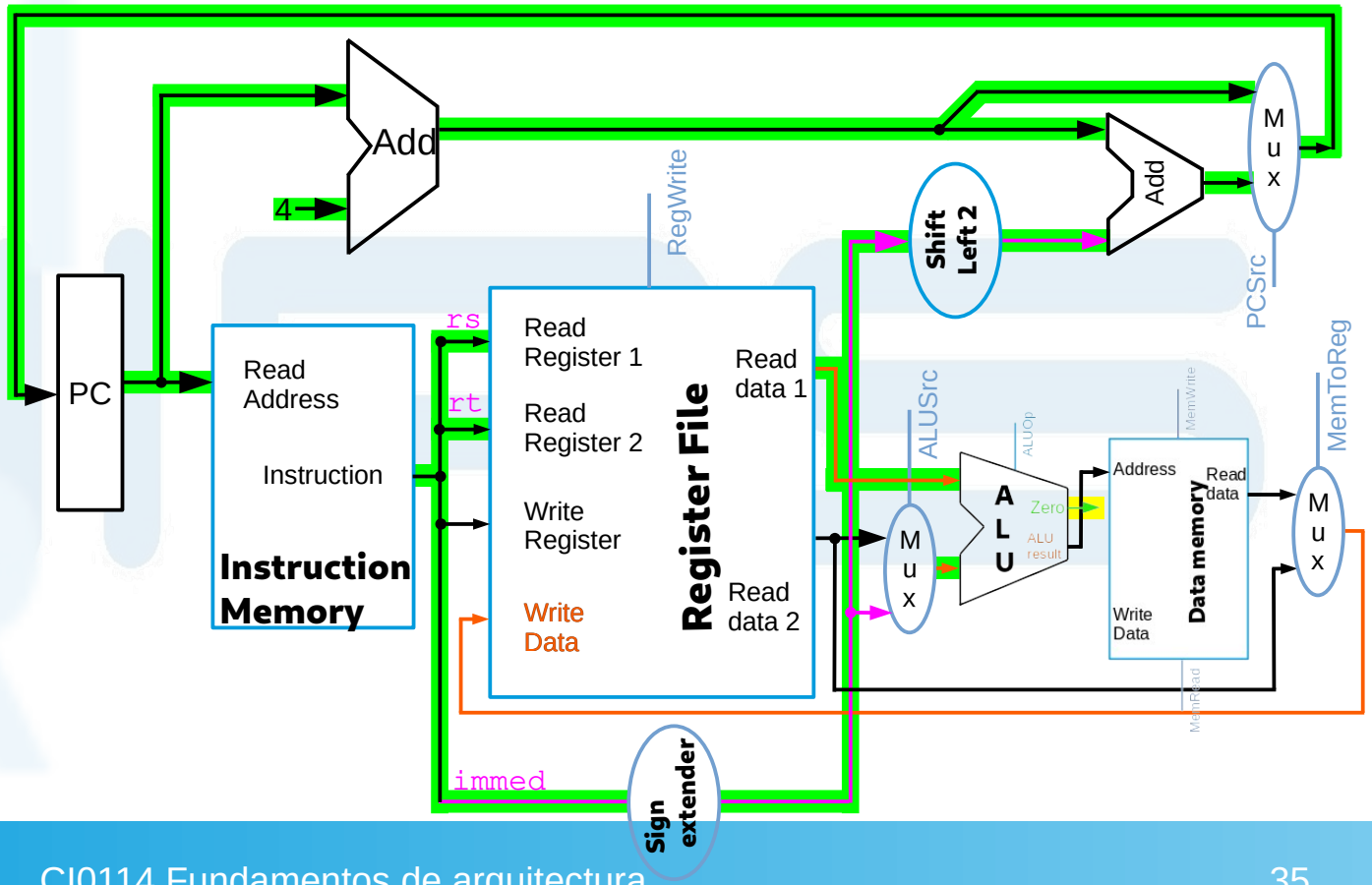
The operands *rs*, *rt*, and *immed* have been added to the datapath for added clarity.



Datapath for R and I format instructions

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

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



J-format instructions

The last instruction we have to implement in our simple MIPS subset is the *jump* instruction.

- `j target`

An example jump instruction is `j target`. This instruction indicates that the next instruction to be executed is at the address of label **target**.

Name	Fields	
Field size	6	26
J format	opcode	target address

J-format instructions

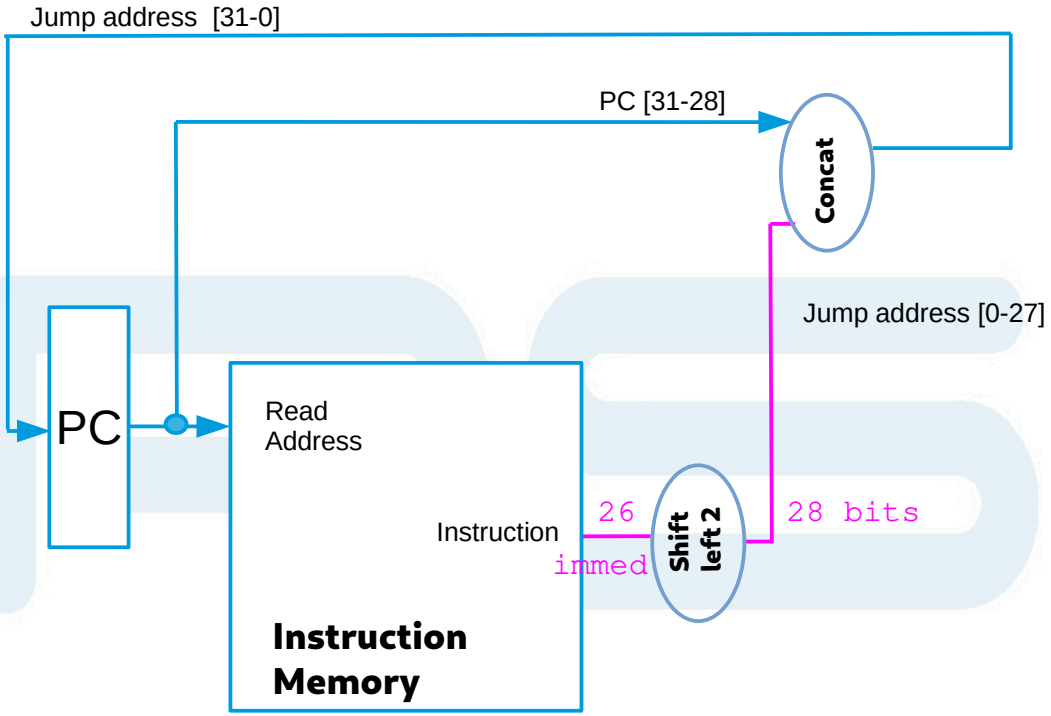
Note, we do not have enough space in the instruction to specify a full 32 bits target address.

- Jumps solve this problem by specifying a portion of an absolute address
  - Take the 26-bit target address field from the instruction, left shitf by 2 (instructions are word-aligned)
  - Concatenate the result with the upper 4 bits of PC + 4

Datapath for J-type instructions

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:

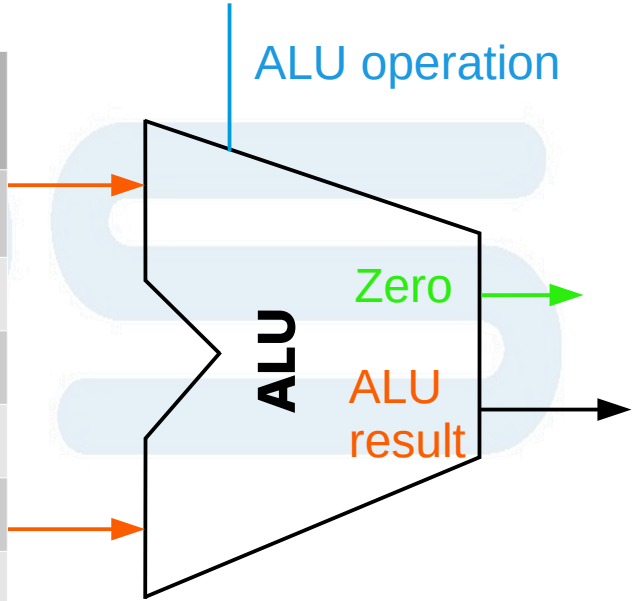
- Operation to be performed by ALU
- Whether register file needs to be written
- Signals for multiple intermediate multiplexors
- Whether data memory needs to be written

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.

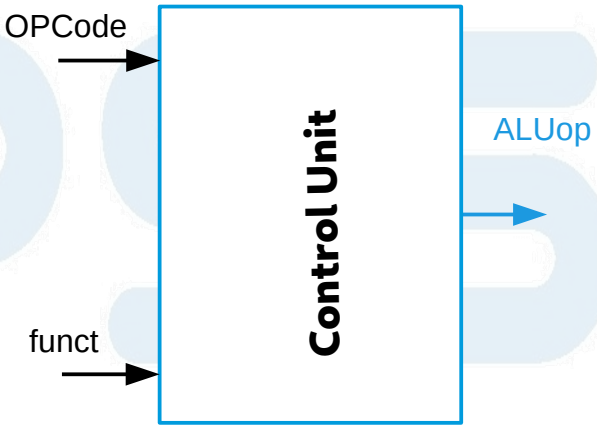
ALU control lines	Function
0000-0011	ADD, SUB, MULT, DIV
0100-0111	AND, OR, XOR, NOT
1000-1001	NAND, NOR
1010-1011	SLT, SLTU
1100-1101	SLL, SRL
1110	SRA
1111	NEG



Control unit

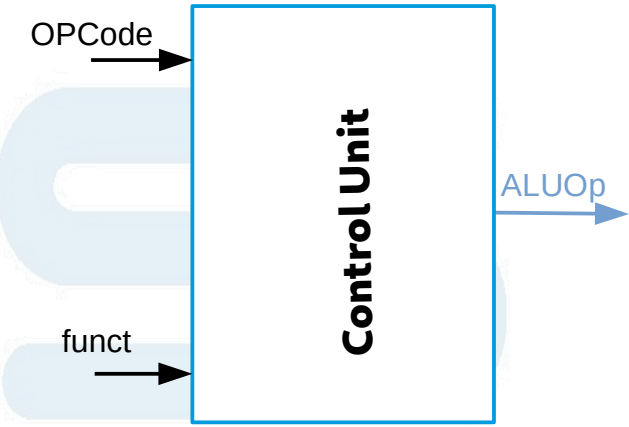
How do we set these control lines? Consider the control unit below.

- The 4-bit *ALUop* input indicates whether an operation should be add for loads and stores, subtract for beq, or others consideren on ISA
- The 6-bit *funct* is a field of R-format instructions. This field defines a unique set of ALU control input lines



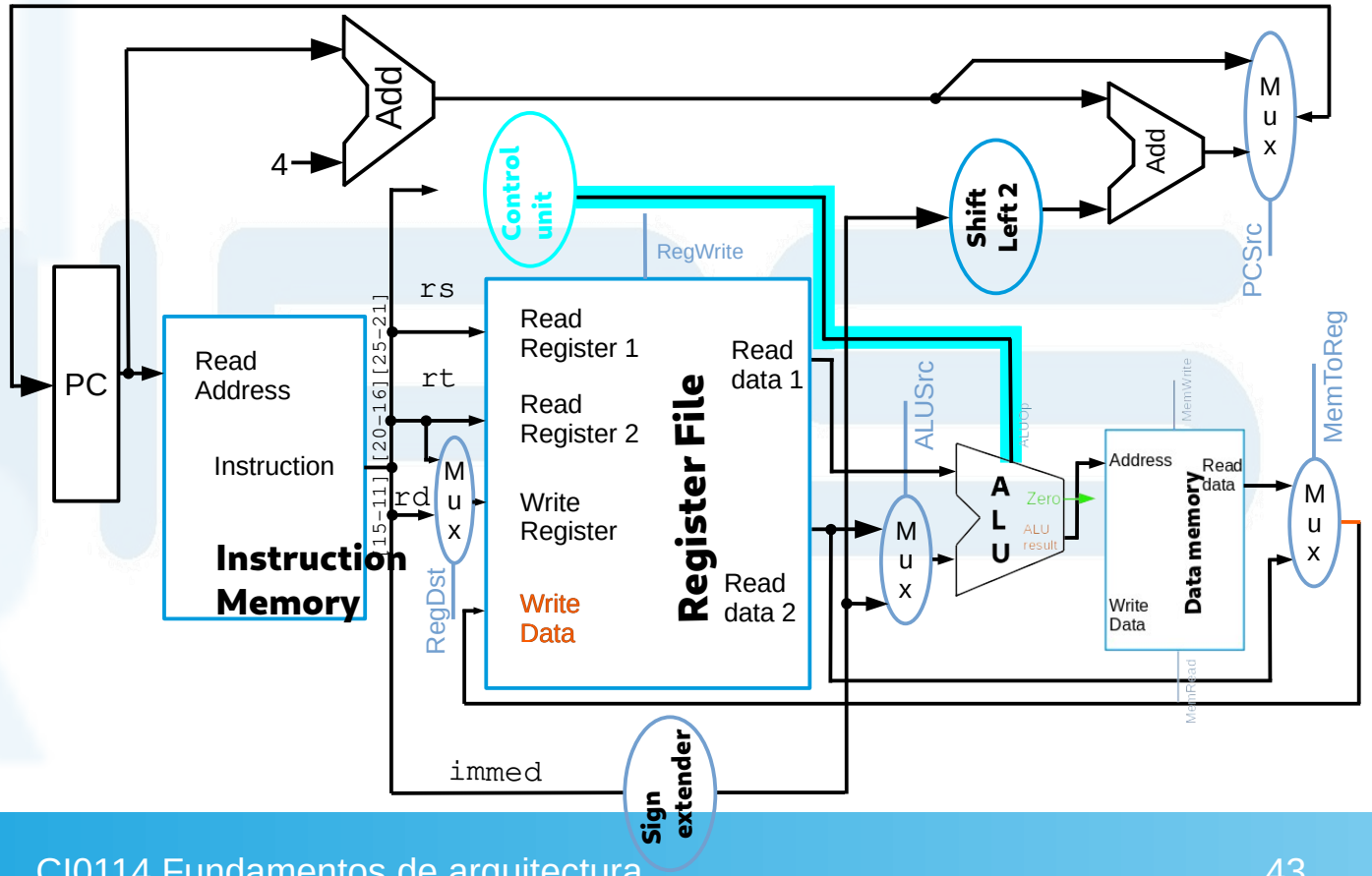
ALU operation (R-format)

OPCode	Operation	funct	ALU action	ALUOp
lw	Load word	N/A	add	0000
sw	Store word	N/A	add	0000
beq	Branch equal	N/A	subtract	0001
R-type	add	100000	add	0000
R-type	sub	100010	subtract	0001
R-type	and	100100	and	0100
R-type	or	100111	or	0101
R-type	Set on <	101010	slt	1010
R-type	Shift left	000000	sll	1100



CPU scheme for ALU control

Notice the added element for determining the ALU control input (ALUOp) from the funct field for R-format.



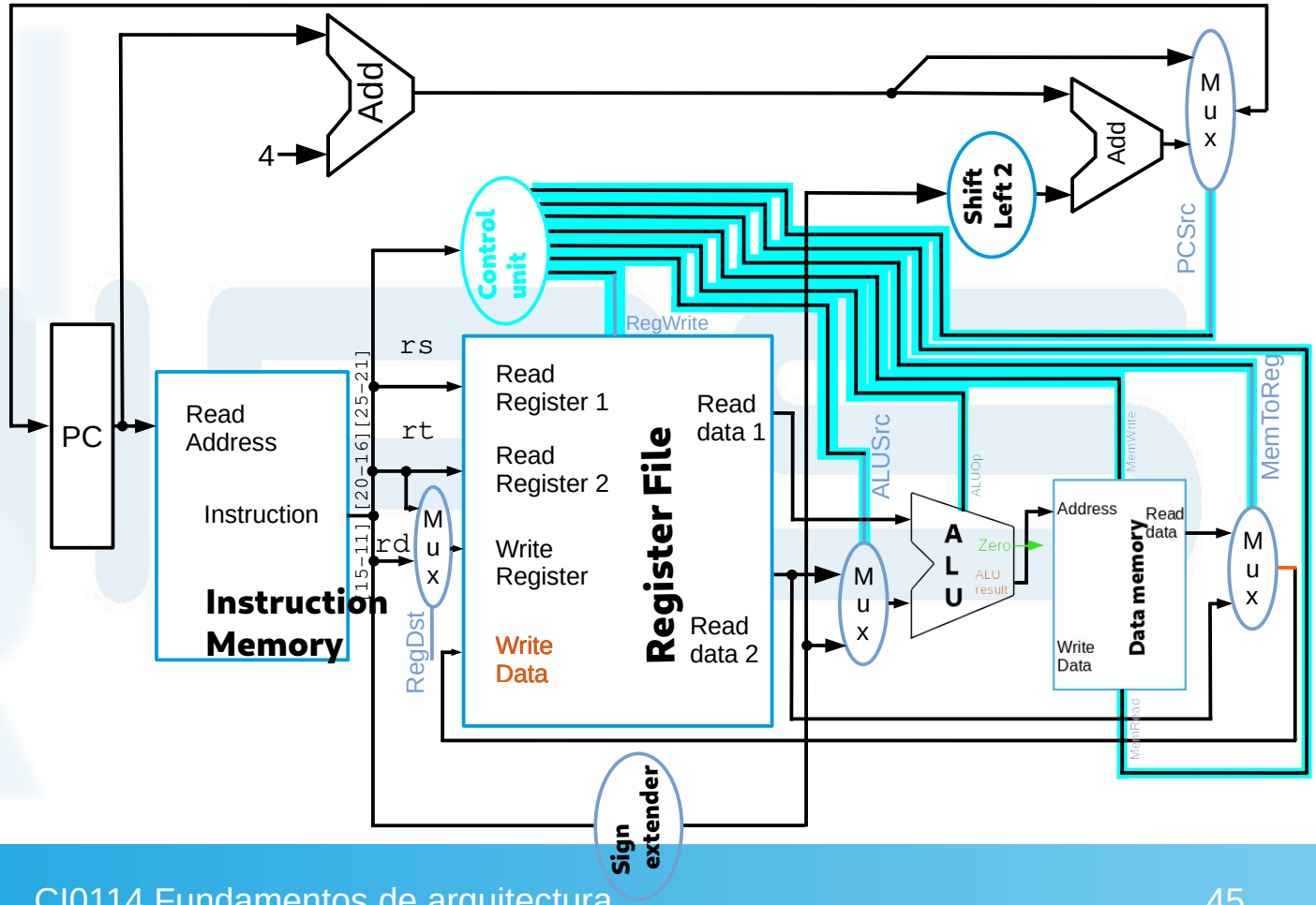
Control signals

Signal name	Effect when not set	Effect when set
RegDst	Destination register comes from *rt* field	Destination register comes from *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 is 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

Control logic

Here, we add control logic for every instruction except the jump instruction.

Notice how most of the control decisions can be decided using only the upper 7 bits of the instruction (opcode).



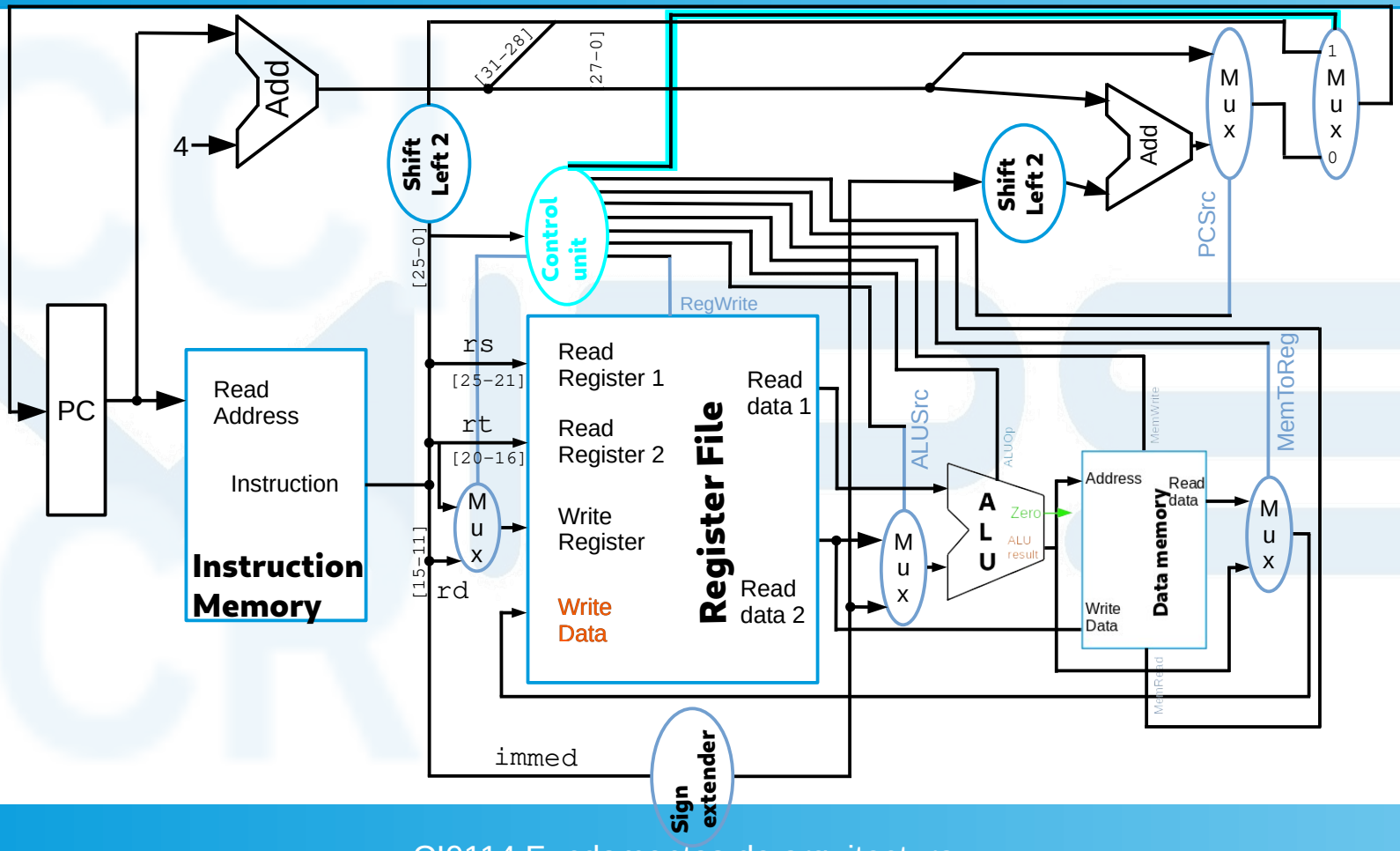
Control signals

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	PCSrc
R-type	1	0	0	1	0	0	0
lw	0	1	1	1	1	0	0
sw	x	1	x	0	0	1	0
beq	x	0	x	0	0	0	1

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

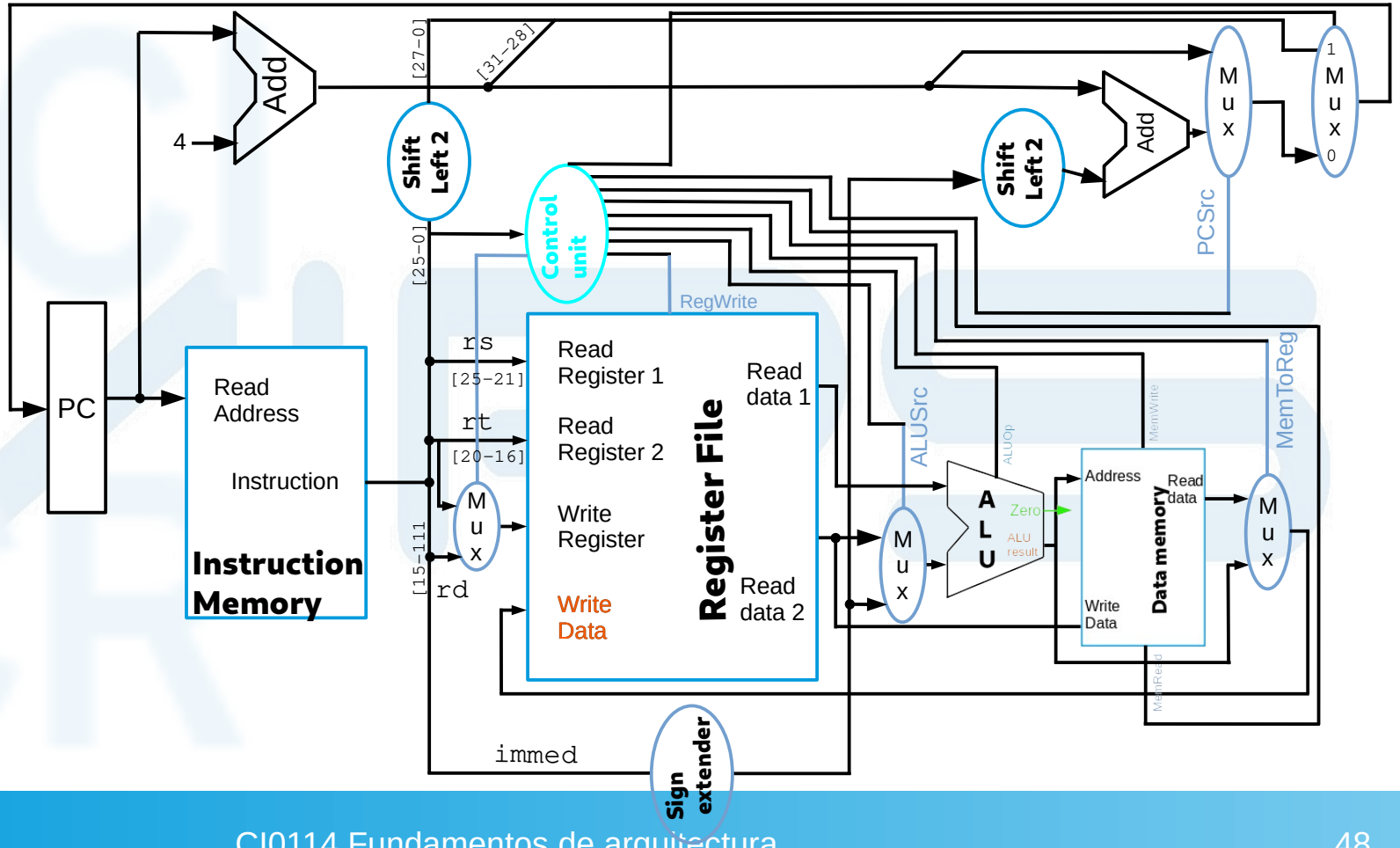
Additional control line for jump instruction



Quiz time!!!

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

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

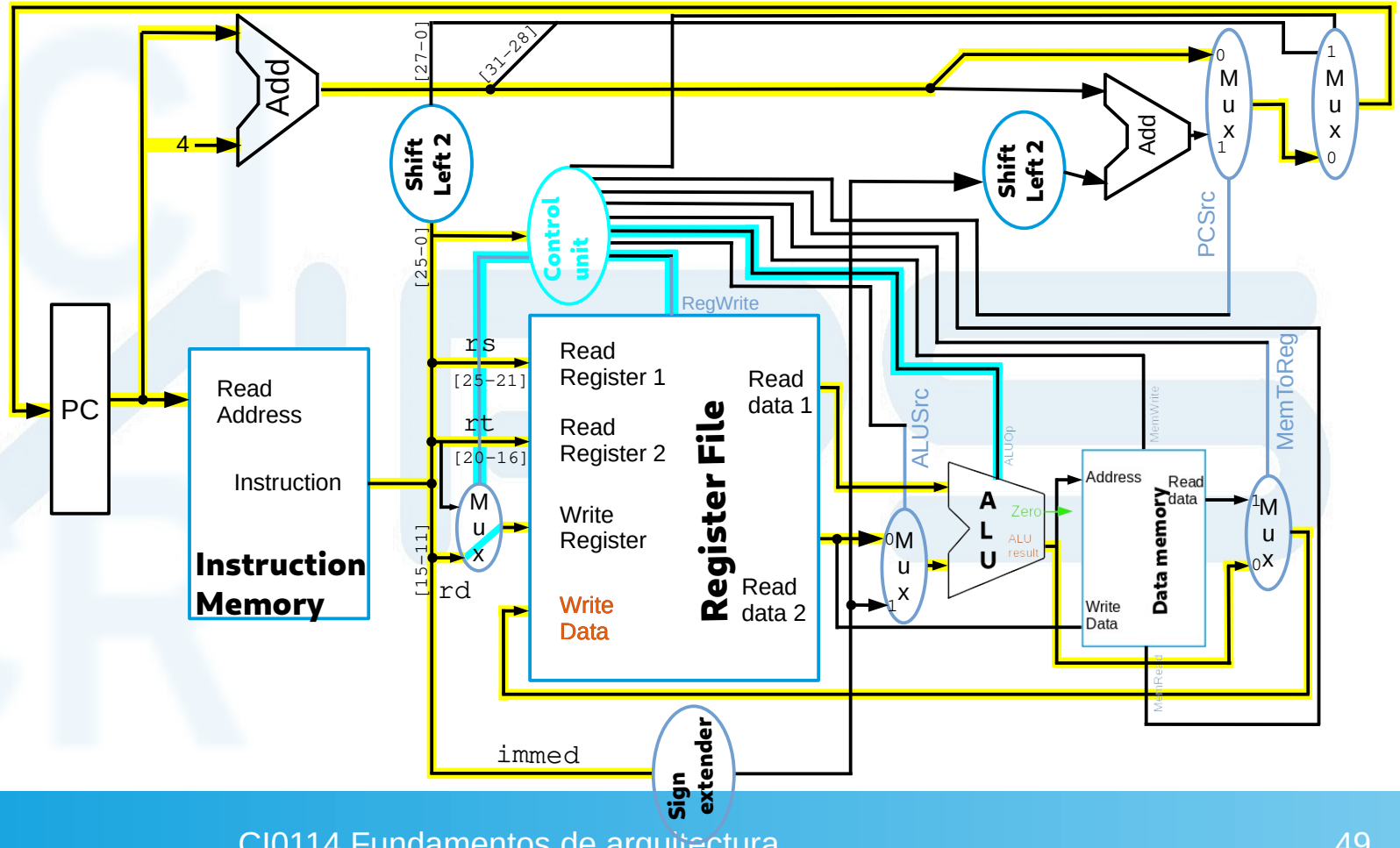


Quiz solution part 1

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

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

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

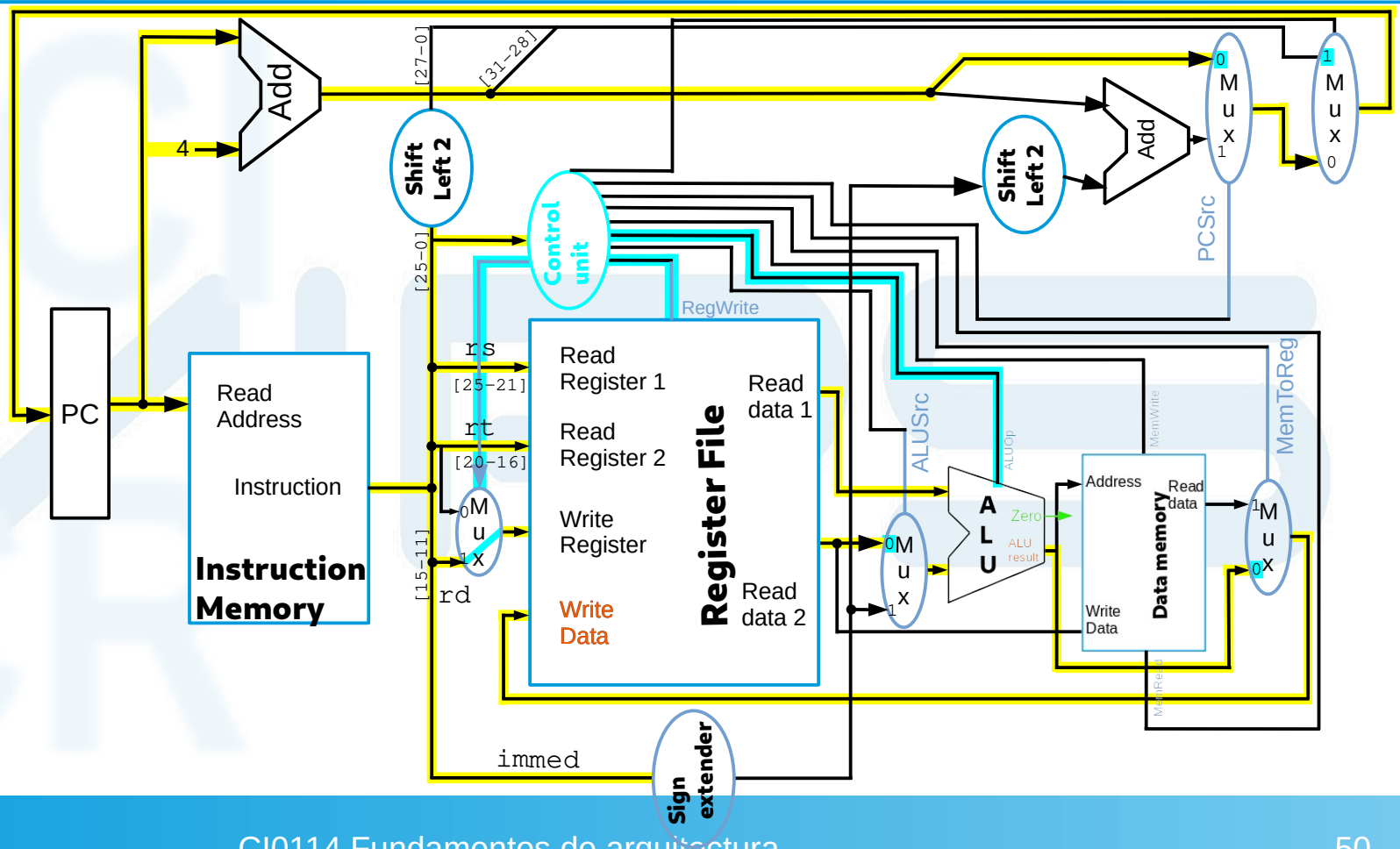


Quiz solution part 2

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

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

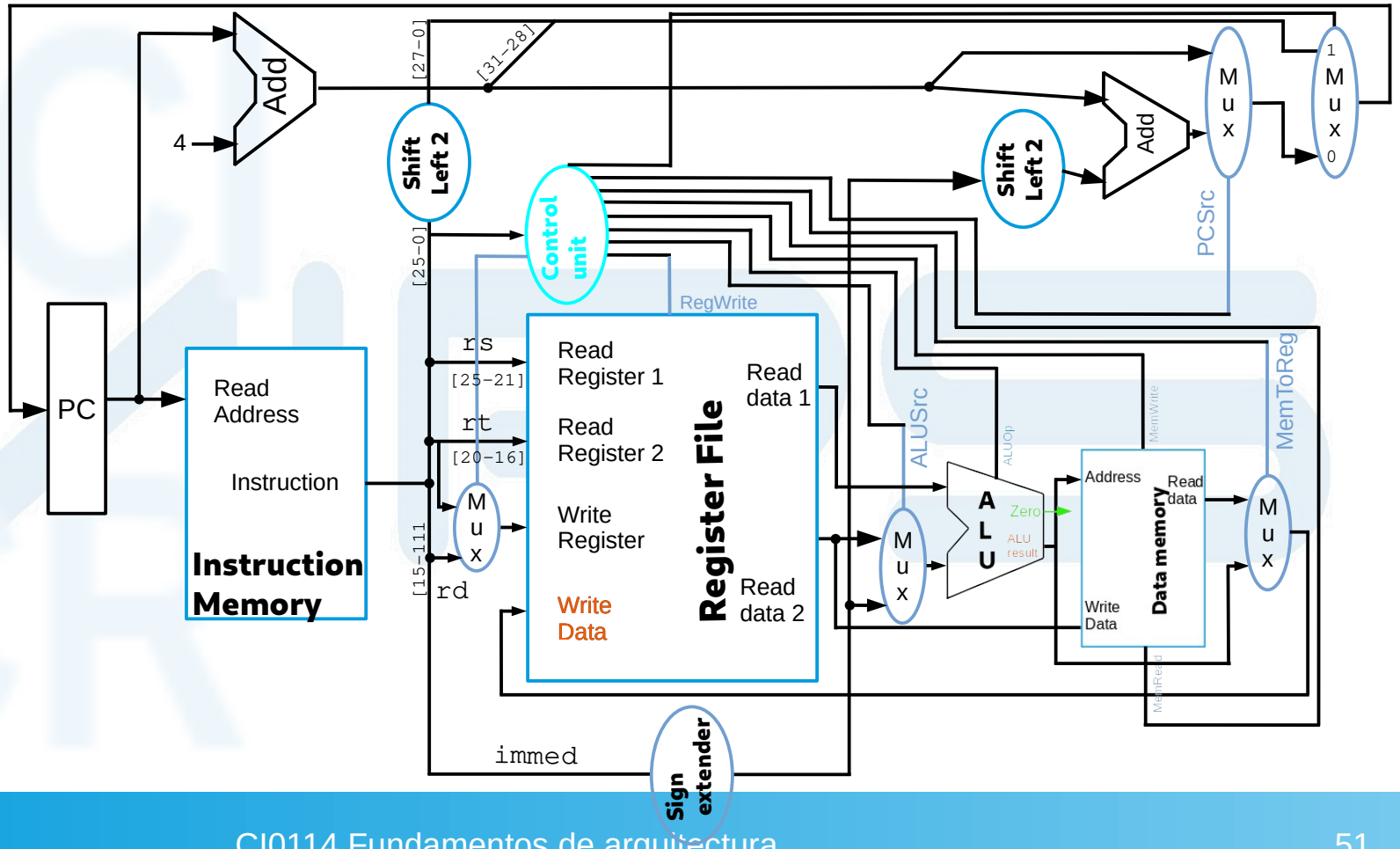
Datapath shown in yellow. Relevant control line assertions in blue.



Quiz time!!!

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

```
beq $rd, $rs, imm
```

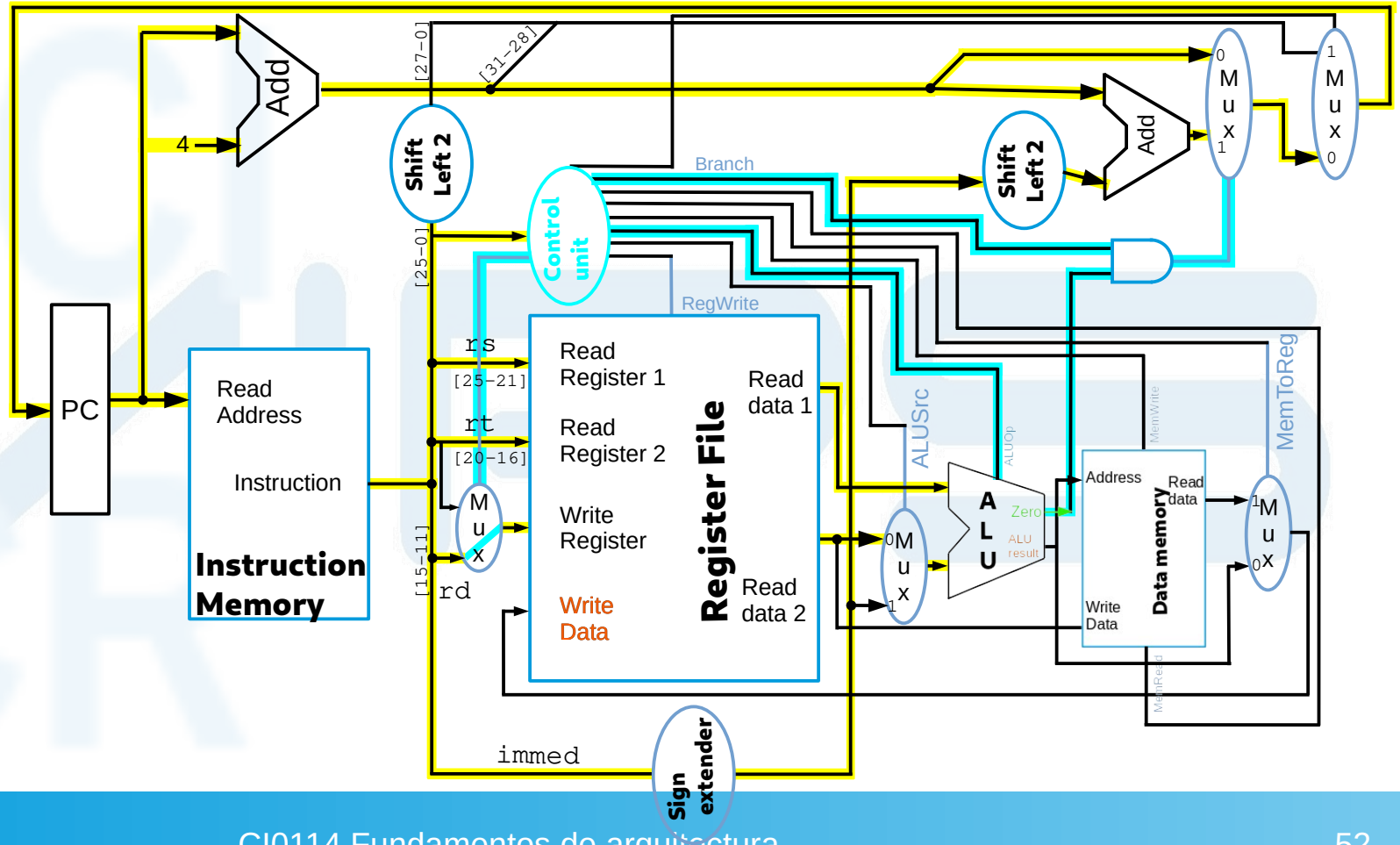


Quiz solution

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

```
beq $rd, $rs, $rt
```

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



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

Instr	Instruction Memory	Register Read	ALU operation	Data Memory	Register Write	Total
R-type	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 construction 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 construction violates the idea of making the common case fast
- May be wastefull with respect area since some functional units, such as adders, must be duplicated because they cannot be shared during a clock cycle