Introduction to MIPS Architecture with its data path

Why MIPS?

Architecture: Defined by the instruction set. Examples: MIPS, x86 family, 68k family, ARM, PowerPC, VAX, SPARC,etc.
MIPS - (Microprocessor without Interlocking Pipeline Stages)archiceture developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

The **MIPS** is an example of **RISC (Reduced Instruction Set Computing)**

- RISC uses simplified instruction set (as opposed to a complex set which is used in CISC)
- Provides higher performance architecture capable of executing those instructions using fewer microprocessor cycles per instruction

MIPS: Application

The **MIPS** is used in

- Embedded systems,
- Cisco routers
- Sony PlayStation
- Small computing devices
- Small consumer electronics and appliances
- Google's Honeycomb (Android 3) tablet

- There are multiple versions of MIPS: including MIPS I, II, III, IV, and V
- the current version of MIPS is MIPS32/64
- MIPS is a load/store architecture (also known as a register-register architecture); except for the load/store instructions used to access memory, all instructions operate on the registers.
- **MIPS I**
- **MIPS I has thirty-two 32-bit general-purpose registers (GPR). Register $0 is hardwired to zero and writes to it are discarded. Register $31 is the link register.**
- **HI and LO, are provided. There is a small set of instructions for copying data between the general-purpose registers and the HI/LO registers.**
- **The program counter has 32 bits. The two low-order bits always contain zero since MIPS I instructions are 32 bits long and are aligned to their natural word boundaries.**

MIPS Register Files

- Although called a "file", a register file is not related to disk files. A register file is a small set of high-speed storage cells inside the CPU. There are special-purpose registers such as the IR and PC, and also general-purpose registers for storing operands of instructions such as add, sub, mul, etc.
- A CPU register can generally be accessed in a single clock cycle, whereas main memory may require dozens of CPU clock cycles to read or write.
- MIPS processor has 32 general-purpose registers, so it takes 5 bits to specify which one to use.
- MIPS is a load-store architecture, which means that only load and store instructions can access memory. All other instructions (add, sub, mul, div, and, or, etc.) must get their operands from registers and store their results in a register.
- The MIPS processor has one standard register file containing 32 32-bit registers for use by integer and logic instructions. These registers are called $0 through $31.
- The MIPS processor has a separate register file for floating point instructions, which contains another 32 32-bit registers called $f0 through $f31.

Register Number	Conventional Name	Usage
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions - not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address
$f0 - $f3	-	Floating point return values
$f4 - $f10	-	Temporary registers, not preserved by subprograms
$f12 - $f14	-	First two arguments to subprograms, not preserved by subprograms
$f16 - $f18	-	More temporary registers, not preserved by subprograms
$f20 - $f31	-	Saved registers, preserved by subprograms

MIPS Instruction format

- Instructions are divided into three types: **R, I and J.**
- Every instruction starts with a 6-bit opcode.
- In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field;
- I-type instructions specify two registers and a 16-bit immediate value;
- J-type instructions follow the opcode with a 26-bit jump target

Type	-31-	format (bits)				-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

MIPS Instructions set : Examples

TABLE 9-1: Arithmetic Instructions in the MIPS ISA

Instruction	Assembly Code		Operation	Comments
add	add	$s1, $s2, $s3	$s1 = $s2 + $s3	Overflow detected
subtract	sub	$s1, $s2, $s3	$s1 = $s2 − $s3	Overflow detected
add immediate	addi	$s1, $s2, k	$s1 = $s2 + k	k, a 16-bit constant, is sign-extended and added; 2's complement overflow detected
add unsigned	addu	$s1, $s2, $s3	$s1 = $s2 + $s3	Overflow not detected
subtract unsigned	subu	$s1, $s2, $s3	$s1 = $s2 − $s3	Overflow not detected
add immediate unsigned	addiu	$s1, $s2, k	$s1 = $s2 + k	Same as addi except no overflow
move from co-processor register	mfc0	$s1, $epc	$s1 = $epc	epc is exception program counter
multiply	mult	$s2, $s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
multiply unsigned	multu	$s2, $s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
divide	div	$s2, $s3	Lo = $s2 / $s3 Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
divide unsigned	divu	$s2, $s3	Lo = $s2 / $s3 Hi = $s2 mod $s3	Unsigned quotient and remainder
move from Hi	mfhi	$s1	$s1 = Hi	Copy Hi to $s1
move from Lo	mflo	$s1	$s1 = Lo	Copy Lo to $s1

TABLE 9-2: Logical Instructions in the MIPS ISA

Instruction	Assembly Code	Operation	Comments
and	and $s1, $s2, $s3	$s1 = $s2 AND $s3	logical AND
or	or $s1, $s2, $s3	$s1 = $s2 OR $s3	logical OR
and immediate	andi $s1, $s2, k	$s1 = $s2 AND k	*k* is a 16-bit constant; *k* is 0-extended first
or immediate	ori $s1, $s2, k	$s1 = $s2 OR k	*k* is a 16-bit constant; *k* is 0-extended first
shift left logical	sll $s1, $s2, k	$s1 = $s2 << k	Shift left by 5-bit constant *k*
shift right logical	srl $s1, $s2, k	$s1 = $s2 >> k	Shift right by 5-bit constant *k*

TABLE 9-3: Memory Access Instructions in the MIPS ISA

Instruction	Assembly Code	Operation	Comments
load word	lw $s1, k($s2)	$s1 = Memory[$s2 + k]	Read 32 bits from memory; memory address = register content + *k*; *k* is 16-bit offset
store word	sw $s1, k($s2)	Memory[$s2 + k] = $s1	Write 32 bits to memory; memory address = register content + *k*; *k* is 16-bit offset;
load halfword	lh $s1, k($s2)	$s1 = Memory[$s2 + k]	Read 16 bits from memory; sign-extend and load into register
store halfword	sw $s1, k($s2)	Memory[$s2 + k] = $s1	Write 16 bits to memory
load byte	lb $s1, k($s2)	$s1 = Memory[$s2 + k]	Read byte from memory; sign-extend and load to register
store byte	sb $s1, k($s2)	Memory[$s2 + k] = $s1	Write byte to memory
load byte unsigned	lbu $s1, k($s2)	$s1 = Memory[$s2 + k]	Read byte from memory; byte is 0-extended
load upper immediate	lui $s1, k	$s1 = k * 2^{16}	Loads constant *k* to upper 16 bits of register

J Instructions: Examples

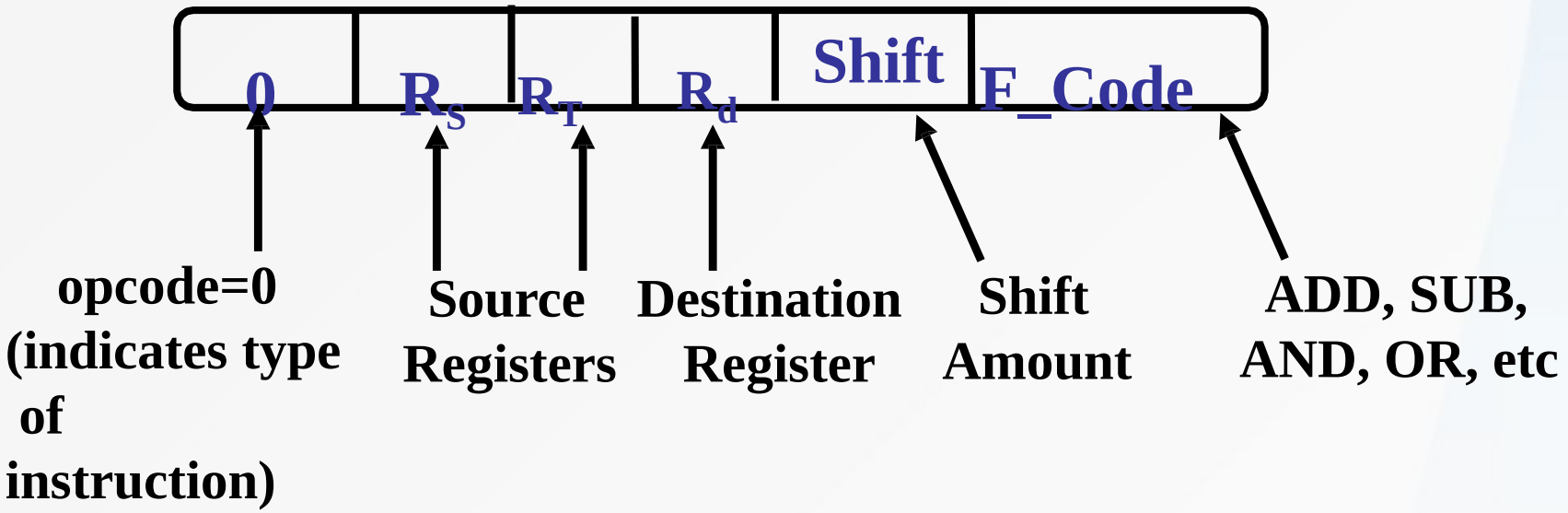
TABLE 9-5: Unconditional Control Transfer Instructions in the MIPS ISA

Instruction	Assembly Code	Operation	Comments
jump	j addr	Go to addr * 4; i.e., PC = addr * 4	Target address = Imm offset * 4; addr is 26 bits
jump register	jr $reg	Go to $reg; i.e., PC = $reg	$reg contains 32-bit target address
jump and link	jal addr	return address = PC + 4; go to addr * 4	For procedure call, return address saved in the link register $31

Mnemonic	Meaning	Type	Opcode	Functioncode
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
bgtz	Branch on Greater Than Zero	I	0x07	NA
blez	Branch if Less Than or Equal to Ze	I	0x06	NA
bne	Branch if Not Equal	I	0x05	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
jr	Jump to Address in Register	R	0x00	0x08
lb	Load Byte	I	0x20	NA
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
mfc0	Move from Coprocessor 0	R	0x10	NA
mfhi	Move from HI Register	R	0x00	0x10
mflo	Move from LO Register	R	0x00	0x12
mthi	Move to HI Register	R	0x00	0x11
mtlo	Move to LO Register	R	0x00	0x13
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
or	Bitwise OR	R	0x00	0x25
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA
sh	Store Halfword	I	0x29	NA
sll	Logical Shift Left	R	0x00	0x00
slt	Set to 1 if Less Than	R	0x00	0x2A

R Instructions

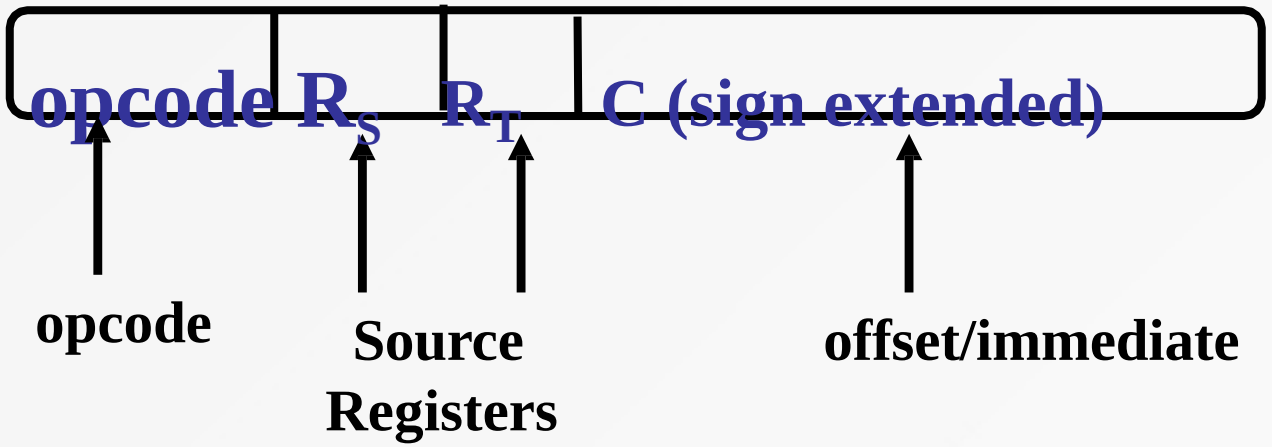
R-Format (32-bits): ALU (core) Instructions



Format	Fields						Used by
	6 bits 31–26	5 bits 25–21	5 bits 20–16	5 bits 15–11	5 bits 10–6	6 bits 5–0	
R-format	opcode	rs	rt	rd	shamt	F_code (funct)	ALU instructions except immediate, Jump Register (JR)

I Instructions

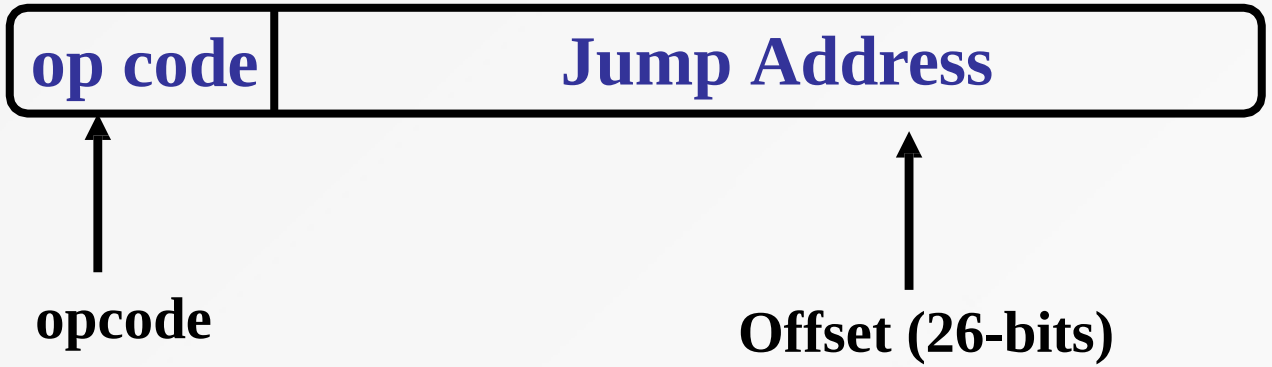
I-Format (32-bits): Load/Store, Immediate ALU, Branch



Format	Fields						Used by
	6 bits 31–26	5 bits 25–21	5 bits 20–16	5 bits 15–11	5 bits 10–6	6 bits 5–0	
I-format	opcode	rs	rt	offset/immediate			Load, store, Immediate ALU, beq, bne

JUMP Instructions

J-Format (32-bits)



Format	Fields						Used by
	6 bits 31–26	5 bits 25–21	5 bits 20–16	5 bits 15–11	5 bits 10–6	6 bits 5–0	
J-format	opcode	target address					Jump (J), Jump and Link (JAL)

DATAPATH IN MIPS Archiceture

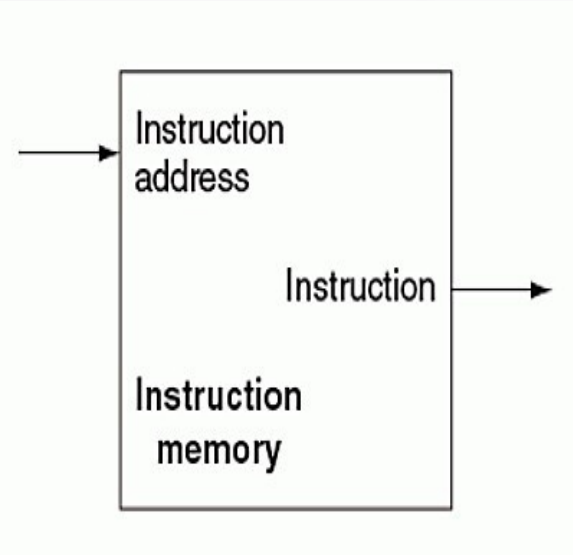
The datapath and control are the two components that come together to be collectively known as the processor.

- 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.

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.



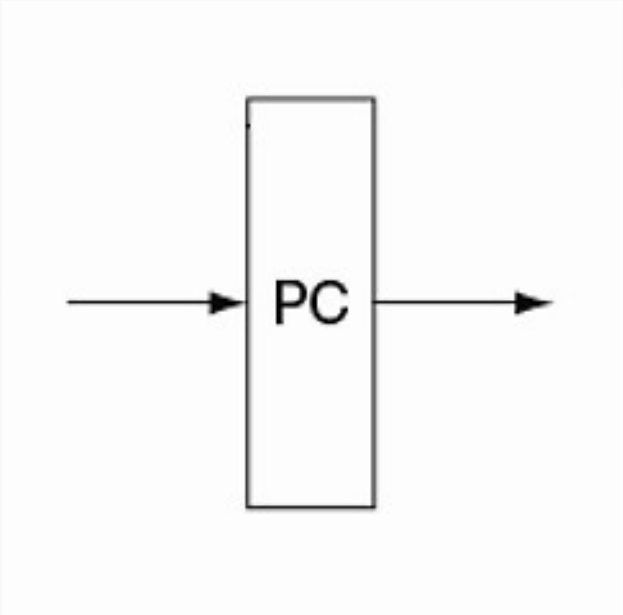
DATAPATH IN MIPS Archiceture

Next, we have the program counter or PC.

The PC is a state element that holds the address of the next 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

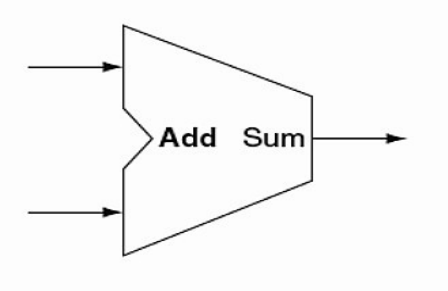
The arrows on either side indicate that the PC state element is both readable and writeable.



Lastly, we have the adder.

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

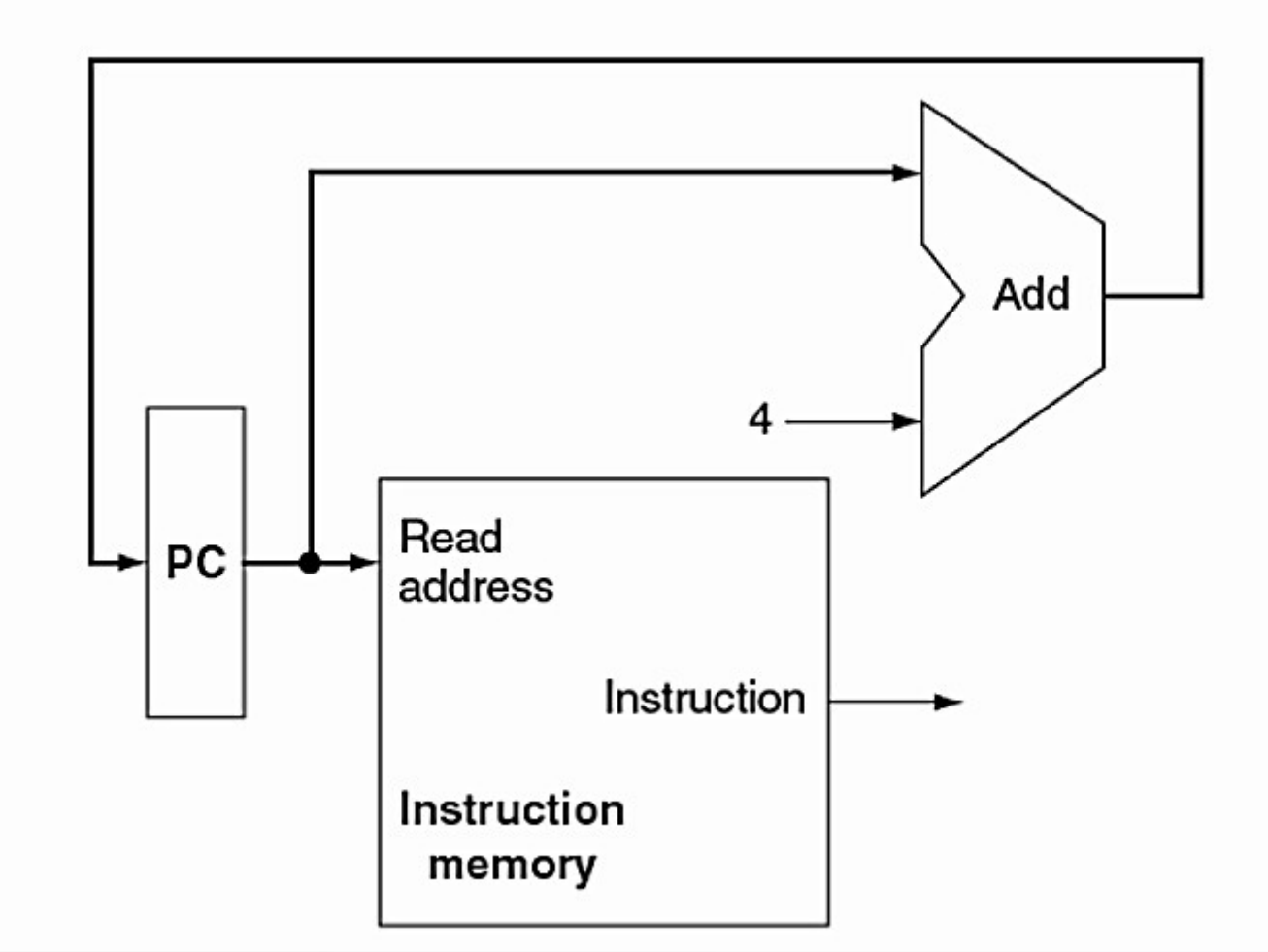
It takes two input values, adds them together and outputs the result.



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 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.

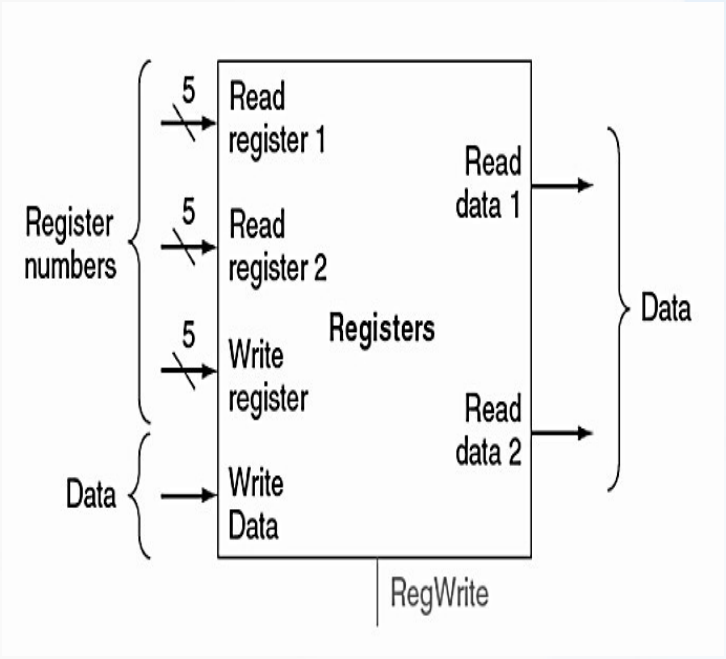
- *Fetch the instruction at the address in PC.*
- *Decode the instruction.*
- *Execute the instruction.*
- *Update the PC to hold the address of the next instruction.*



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

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.



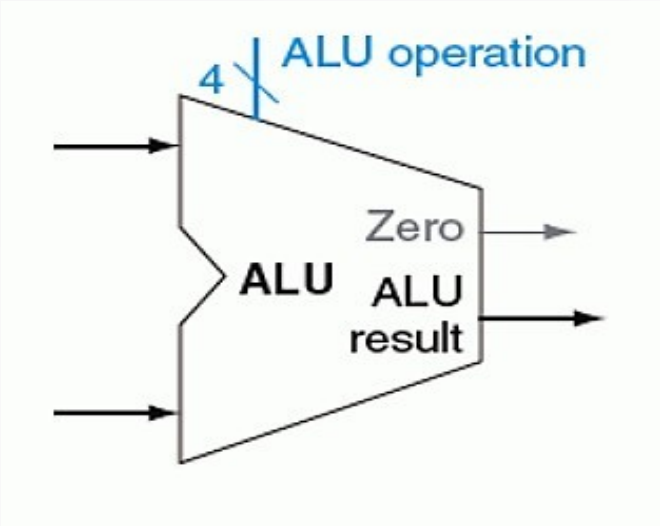
At the bottom, we have the RegWrite input. A writing operation only occurs when this bit is set.

The two output ports are:

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

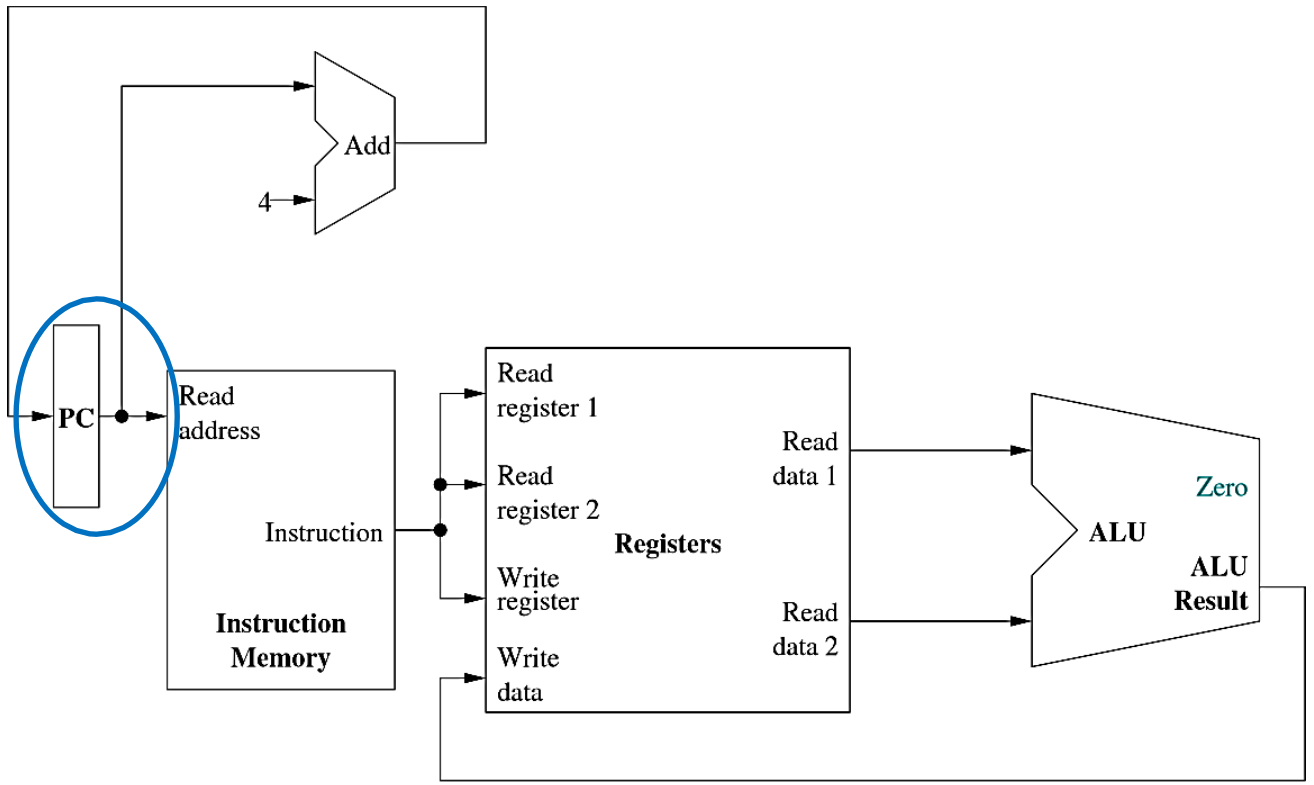
To execute R-format instructions, we need to include the **ALU element.**

The ALU performs the operation indicated by the instruction. It takes two operands, as well as a 4-bit wide operation selector value. The result of the operation is the output value.

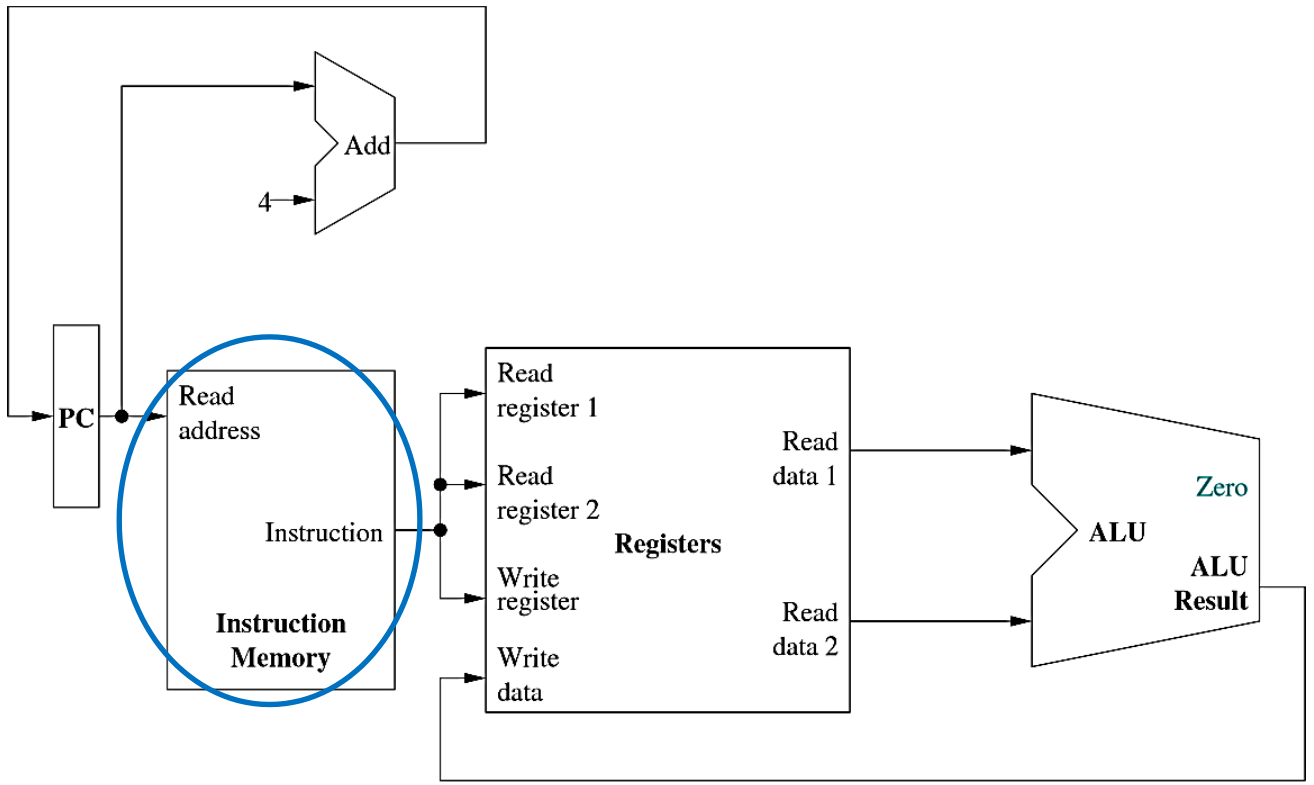


datapath for R-format instructions.

1. Grab instruction address from PC.

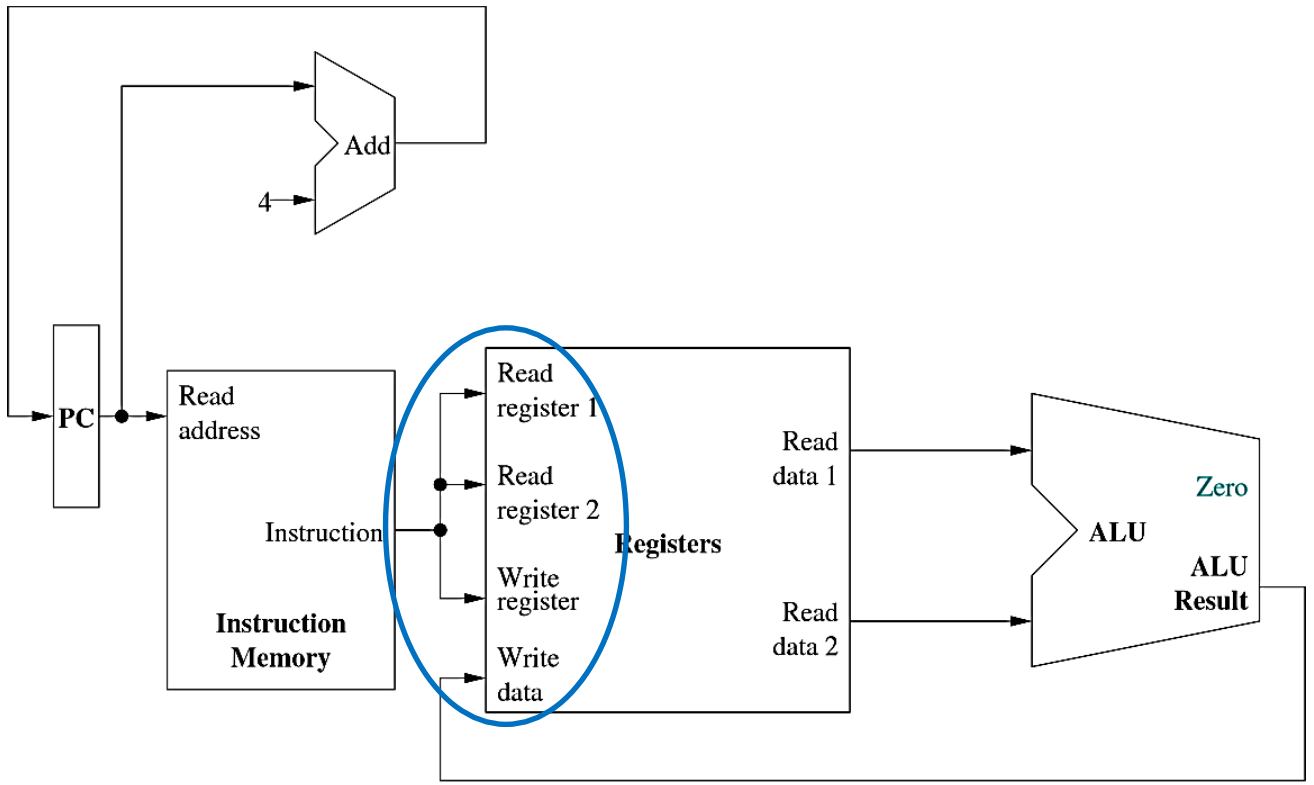


2. Fetch instruction from instruction memory.
3. Decode instruction.

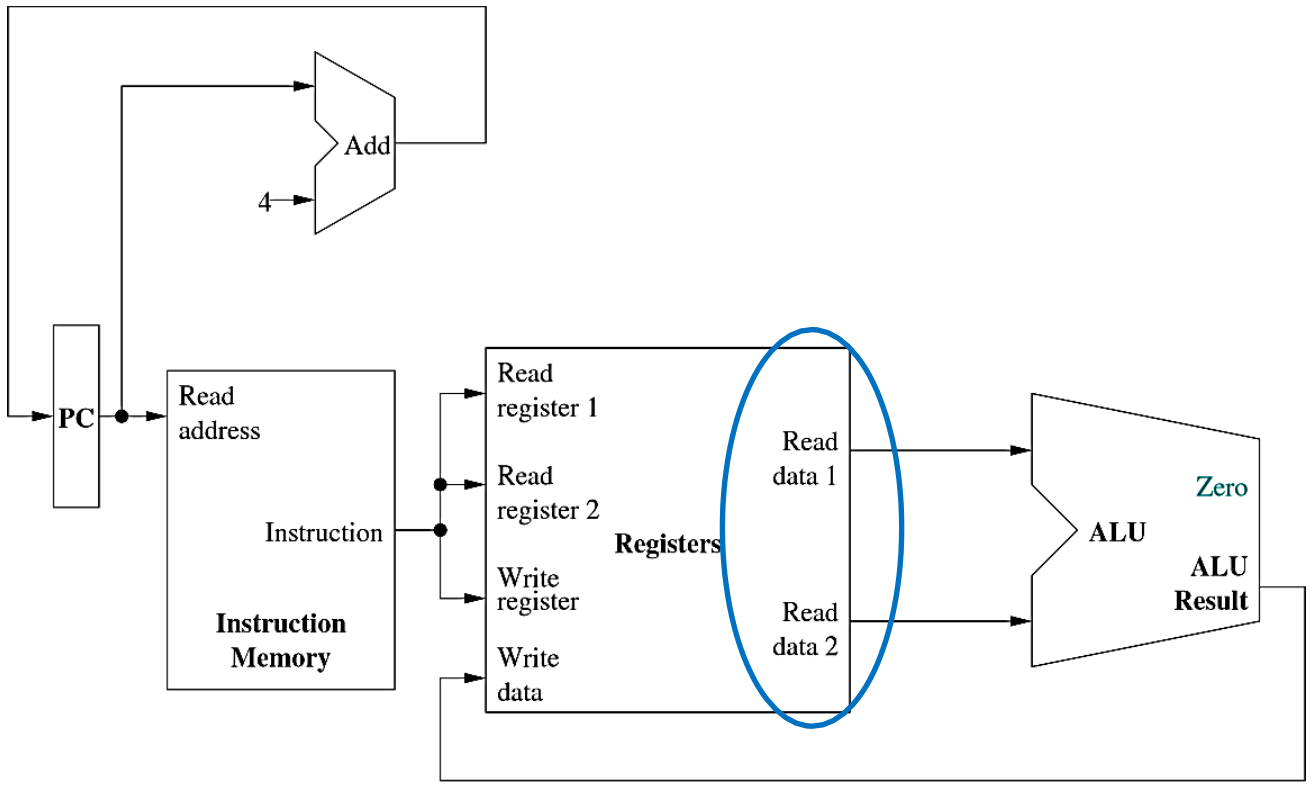


.

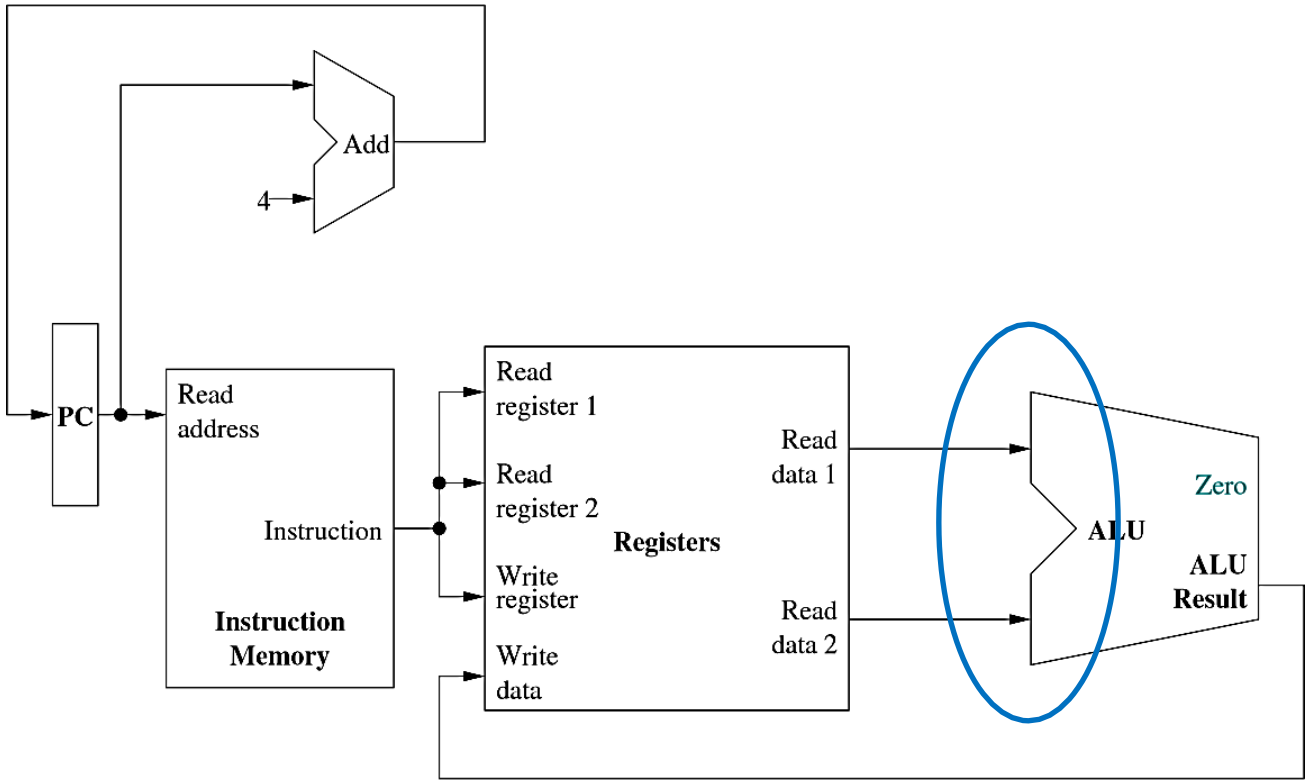
4. Pass `rs`, `rt`, and `rd` into *read register and write register arguments.*



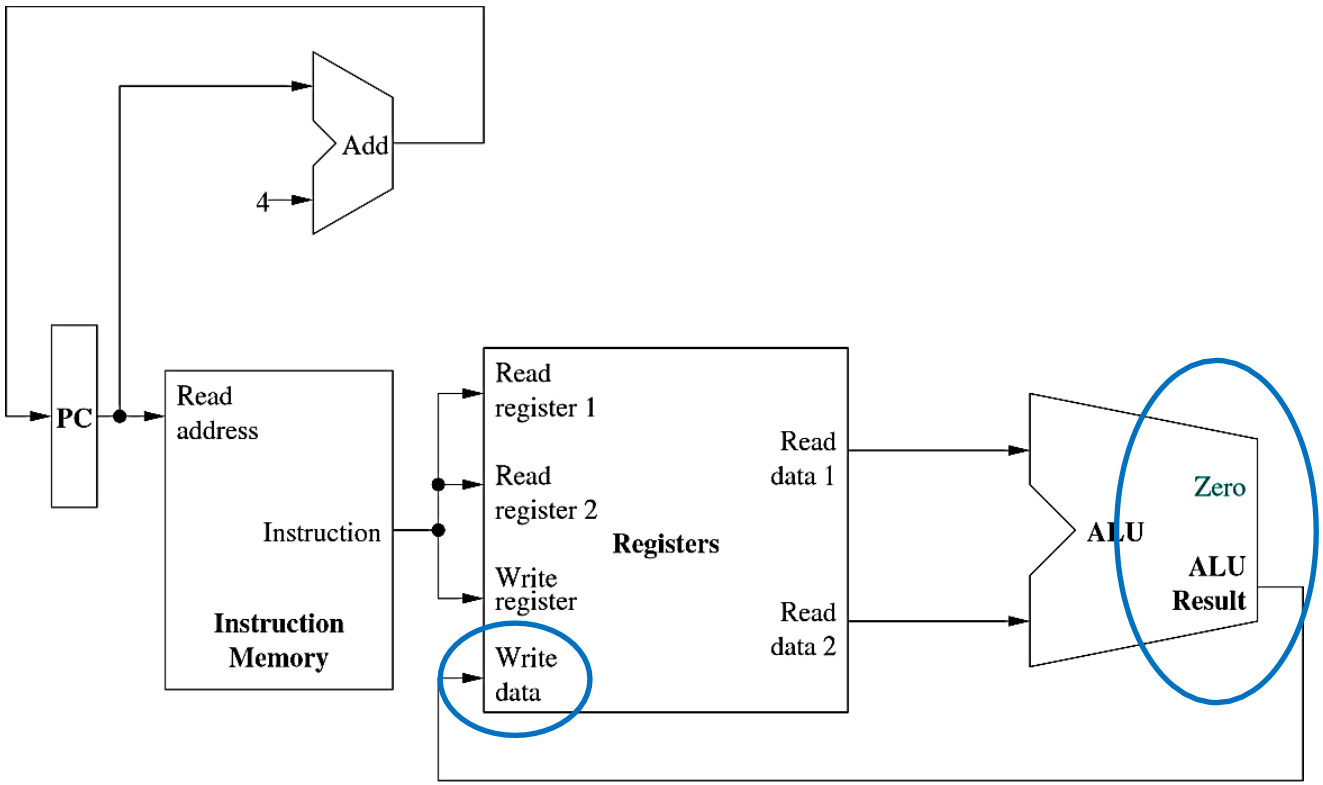
5. Retrieve data from read register 1 and read register 2 (`rs` and `rt`).



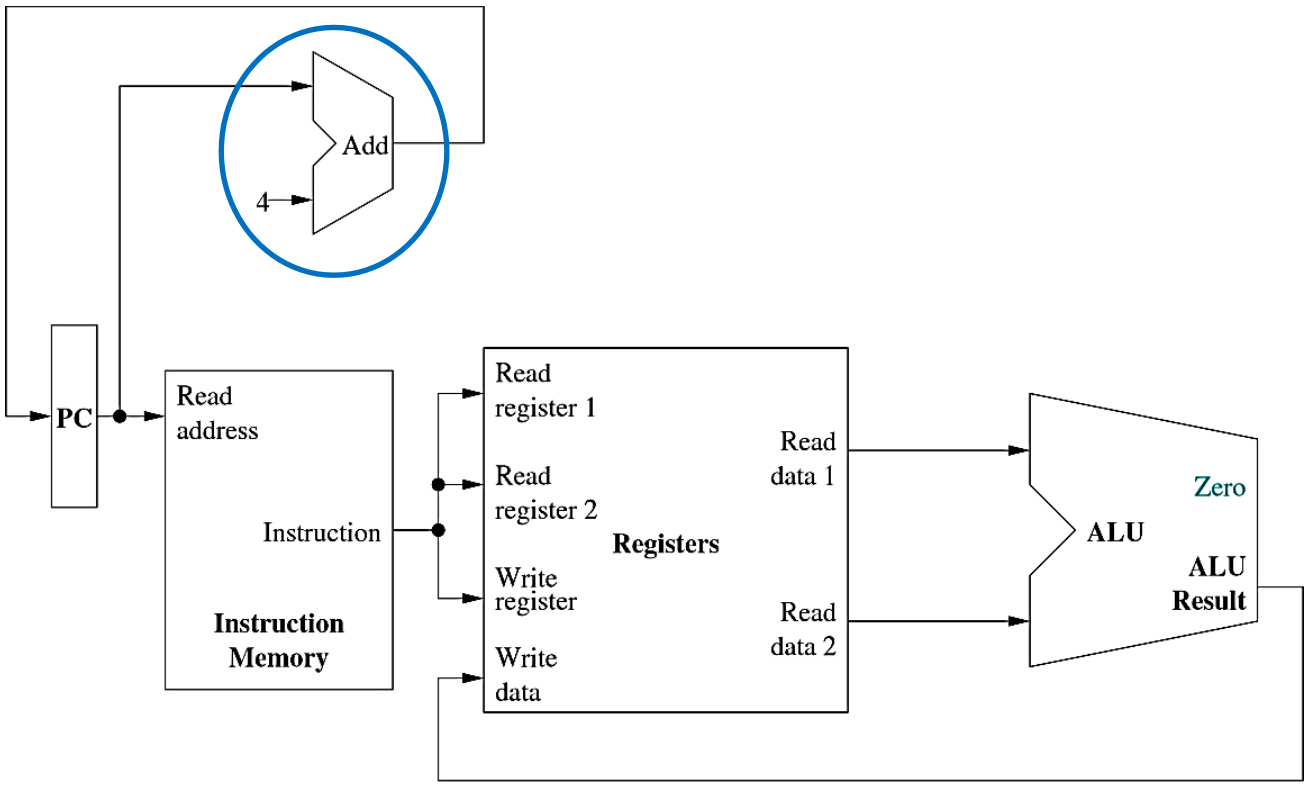
6. *Pass contents of rs and rt into the ALU as operands of the operation to be performed.*



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).*



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



I format Instruction

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		

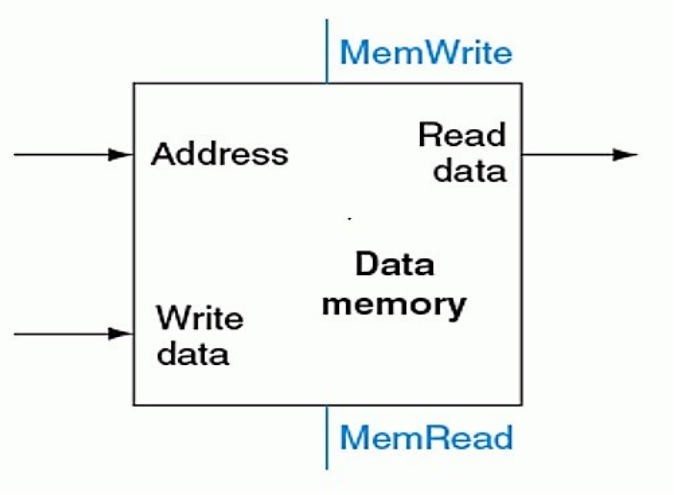
Data Transfer Instruction

```
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.

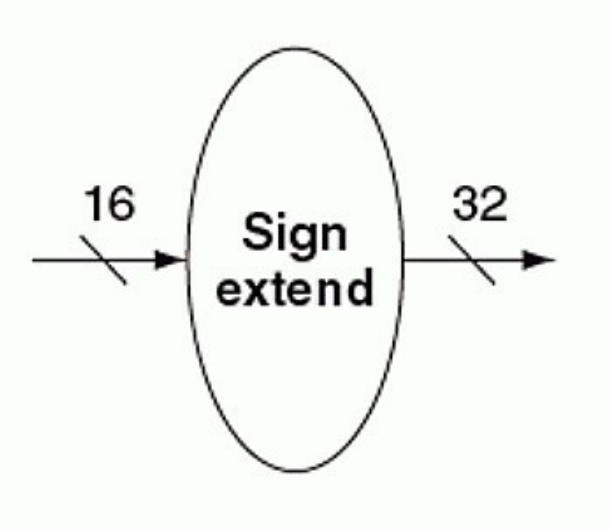
- The *data memory* element implements the functionality for *reading and writing data to/from memory*.
- There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.
- The output is the data read from the memory location accessed, if applicable.
- Reads and writes are signaled by *MemRead* and *MemWrite*, respectively, which must be asserted for the corresponding action to take place.



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.

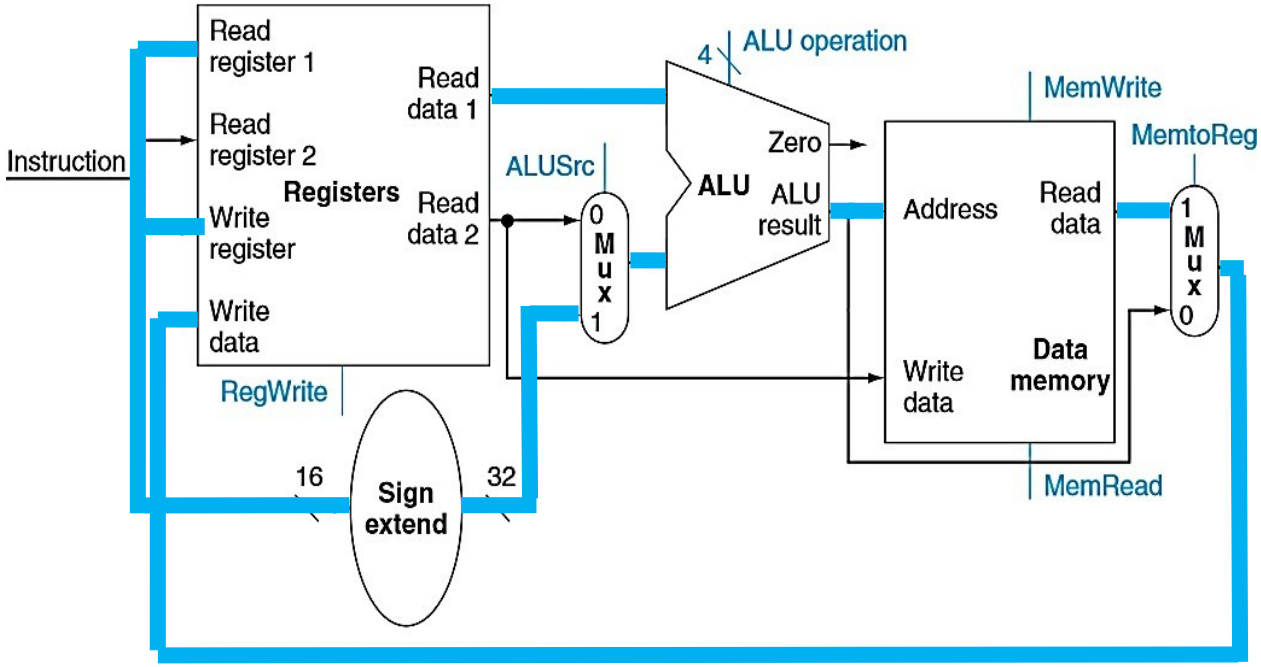
To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.



datapath format and memory access

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

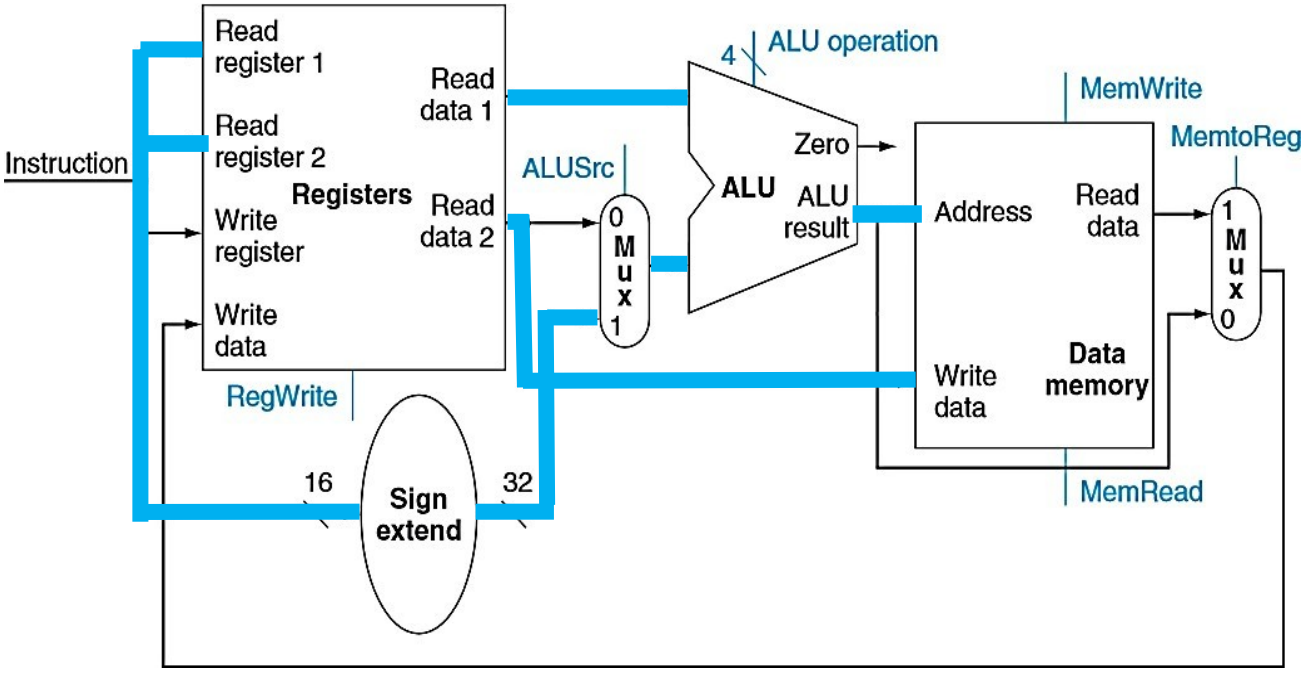
lw $rt, immed($rs)
sw $rt, immed($rs)



datapath format and memory access

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

lw $r t , immed($r s)
sw $r t , immed($r s)



branch Instruction

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		

branch instruction

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!).*
- *jump to=pc+4+(offset*4) to multiply 4 do two times left shift*
- *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		

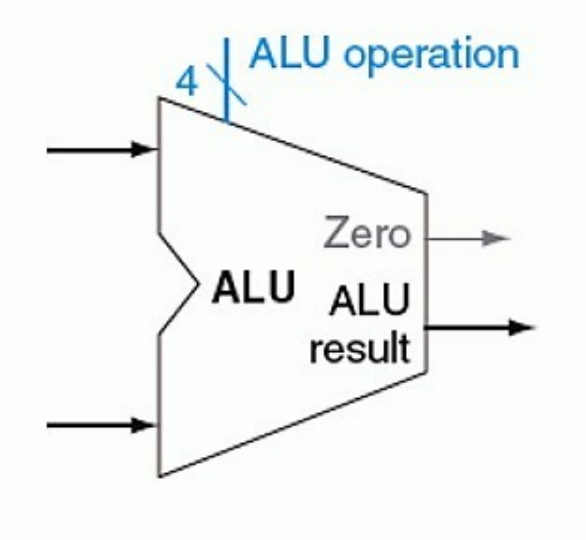
branch Instruction

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.*

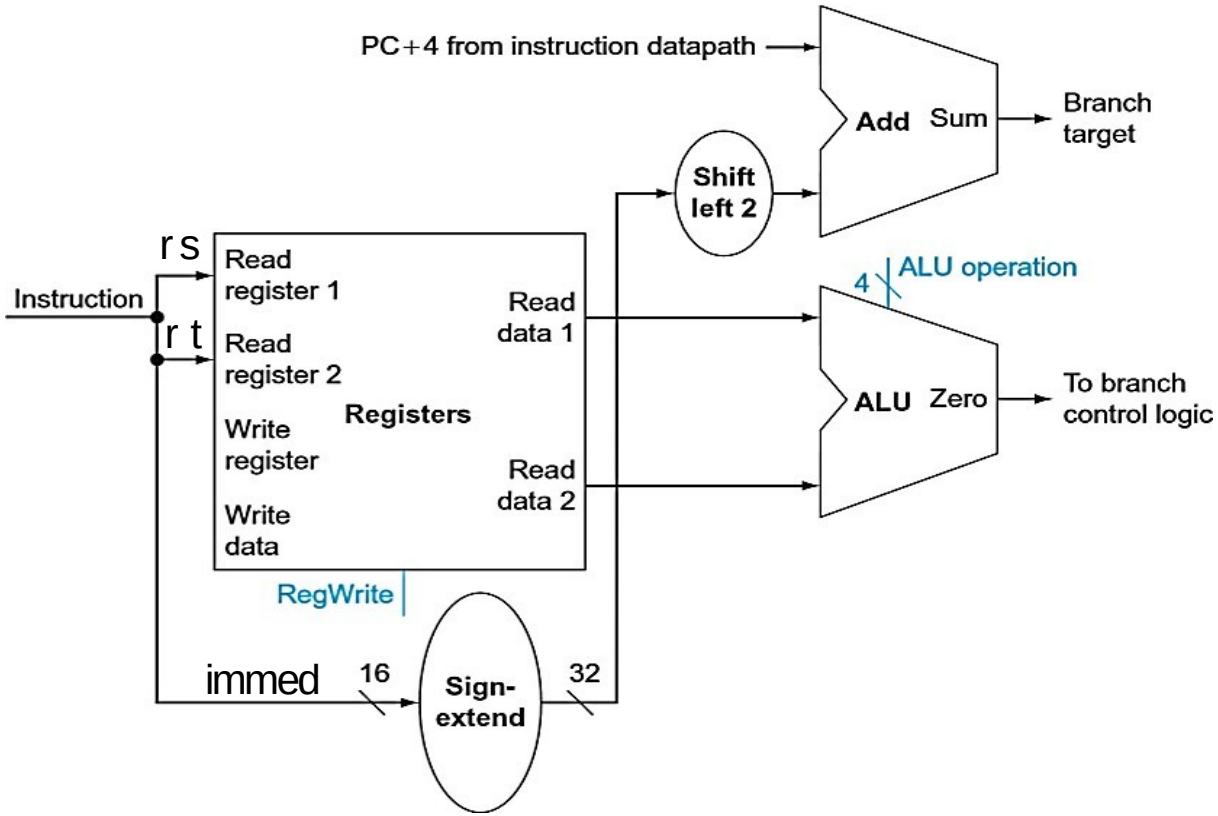


datapath for branch instruction

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

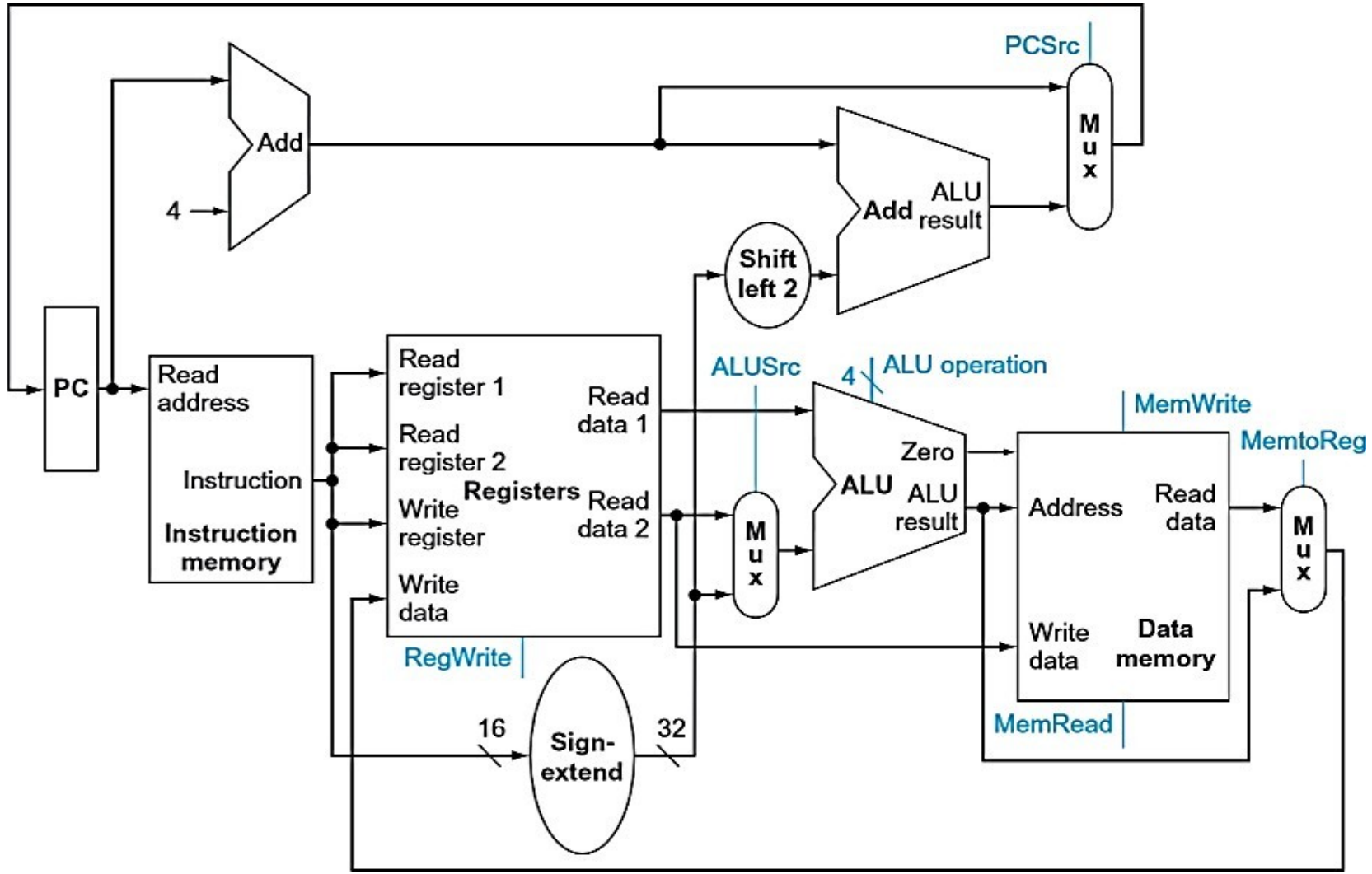
beq $rs, $rt, immed

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



datapath for R and I format

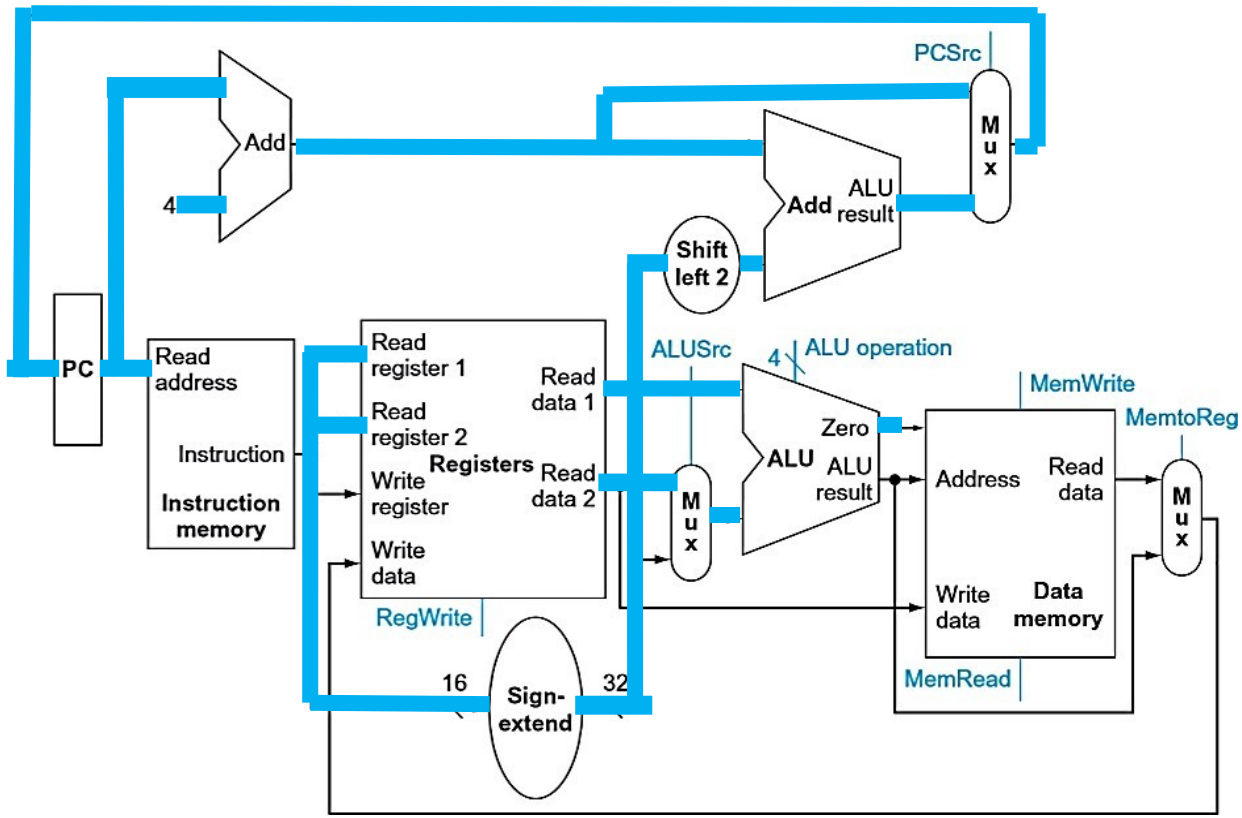
beq $r s , $r t , immed



datapath

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

beq $rs, $rt, immed



J format Instruction

The last instruction we have to implement in our simple MIPS subset is the jump instruction. An example jump instruction is Jump 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	target_addr				

J format Instruction

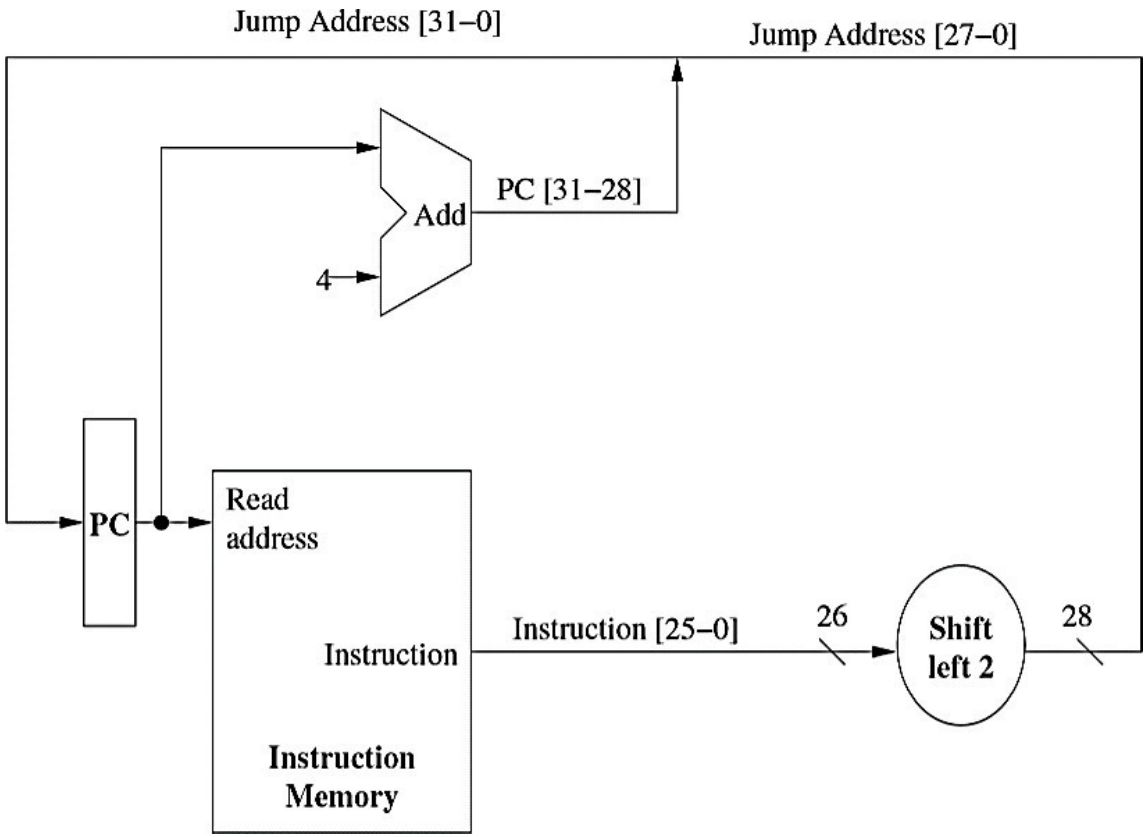
Note, we do not have enough space in the instruction to specify a full target address.

- *Branching solves this problem by specifying an offset in words.*
- *Jump instructions solve this problem by specifying a portion of an absolute address*
  - *Take the 26-bit target address field of the instruction, left-shift by two (instructions are word-aligned), concatenate the result with the upper 4 bits of PC+4.*

datapath for J format

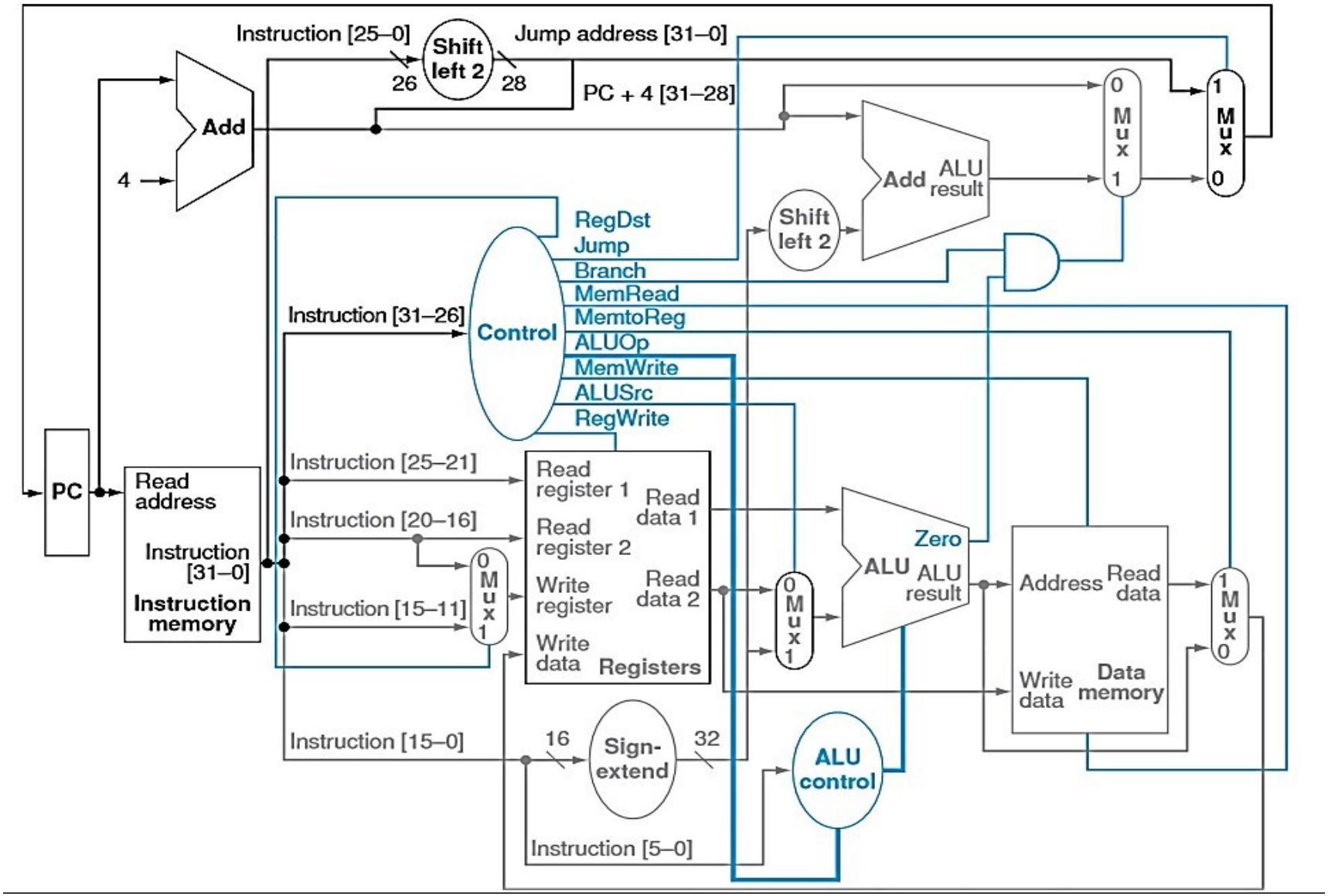
Here, we have modified the datapath to work only for the j instruction.

```
j targaddr
```



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

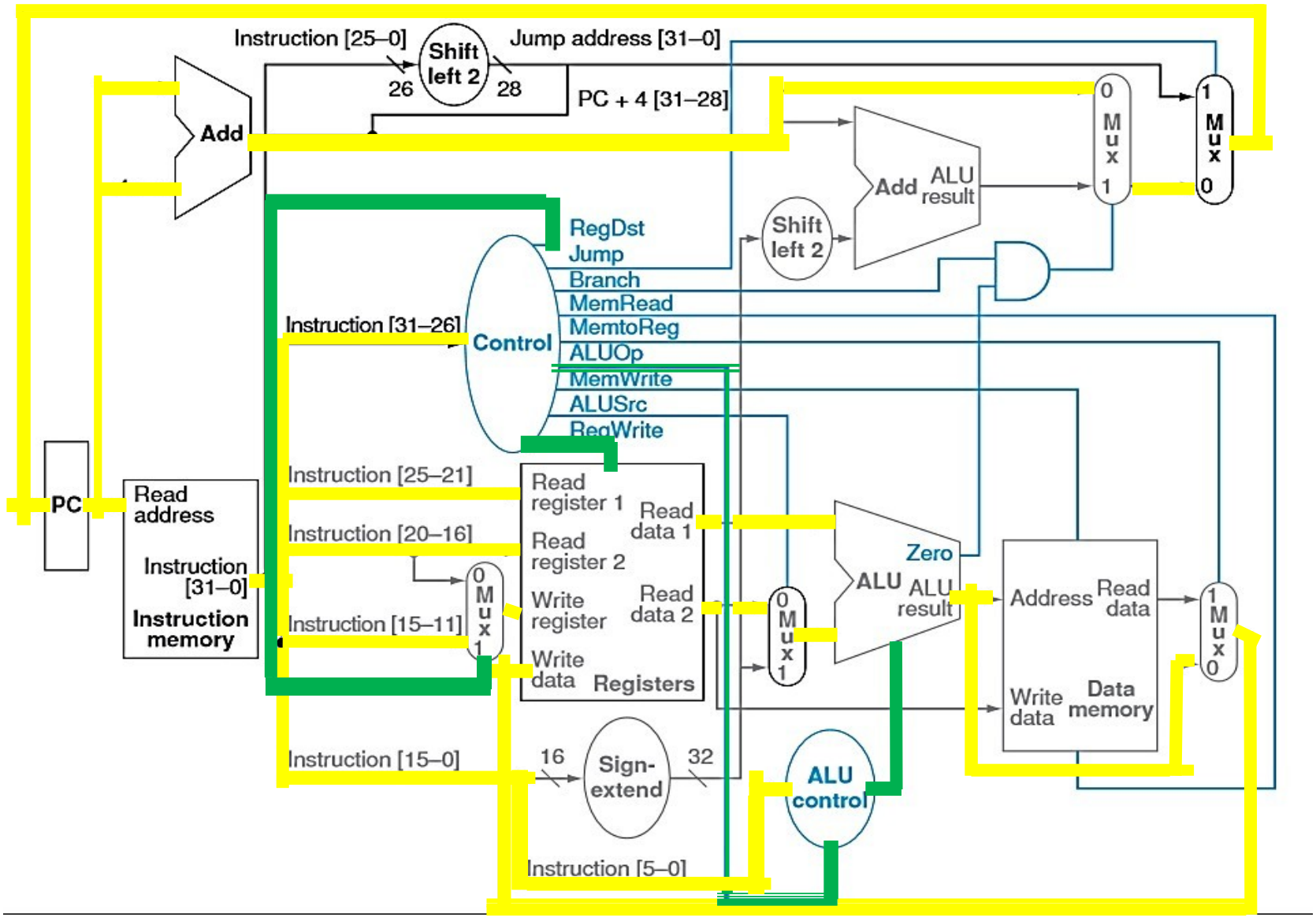
add $rd, $rs, $rt



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`

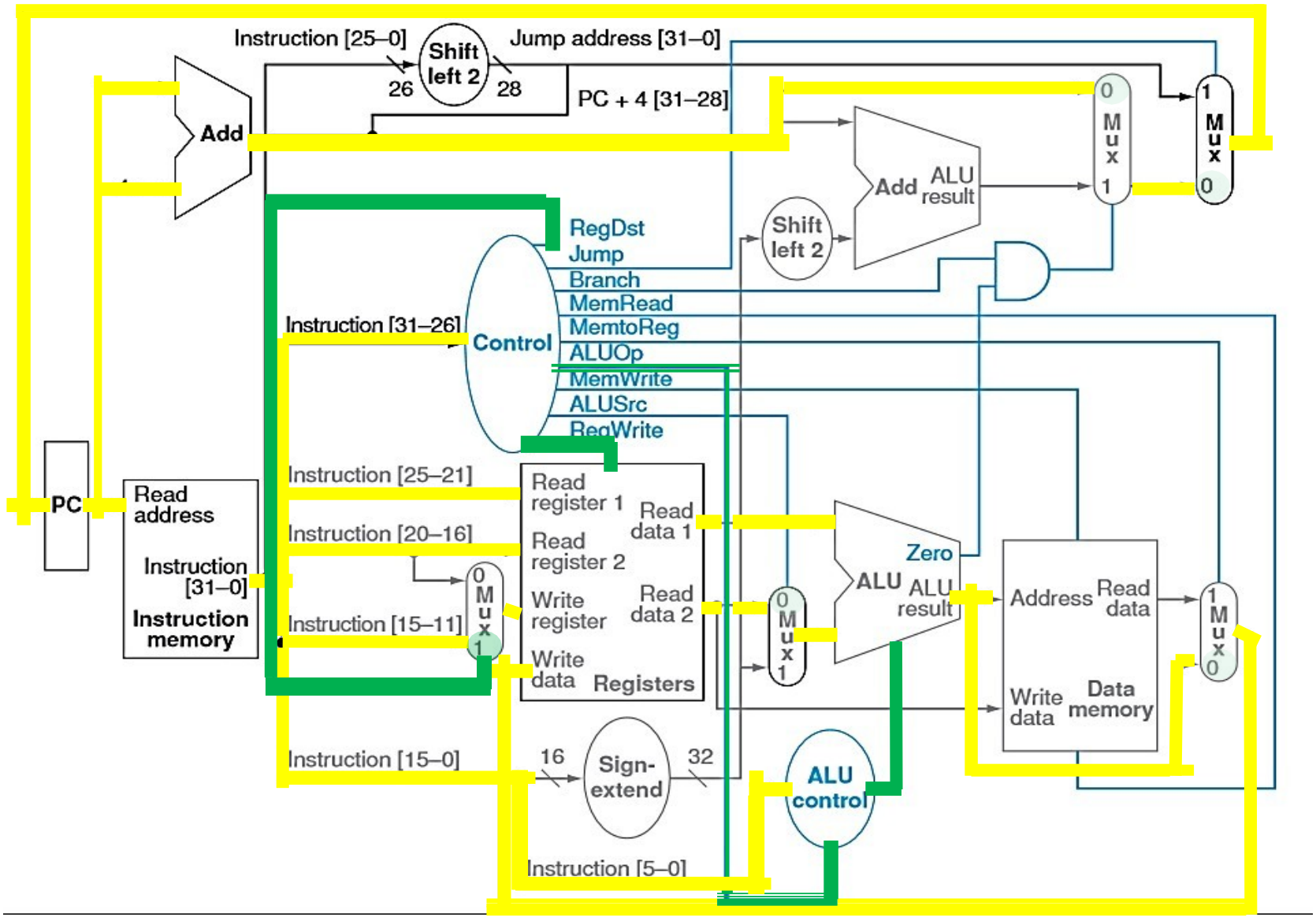
Datapath shown in yellow.
Relevant control line assertions in green.



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

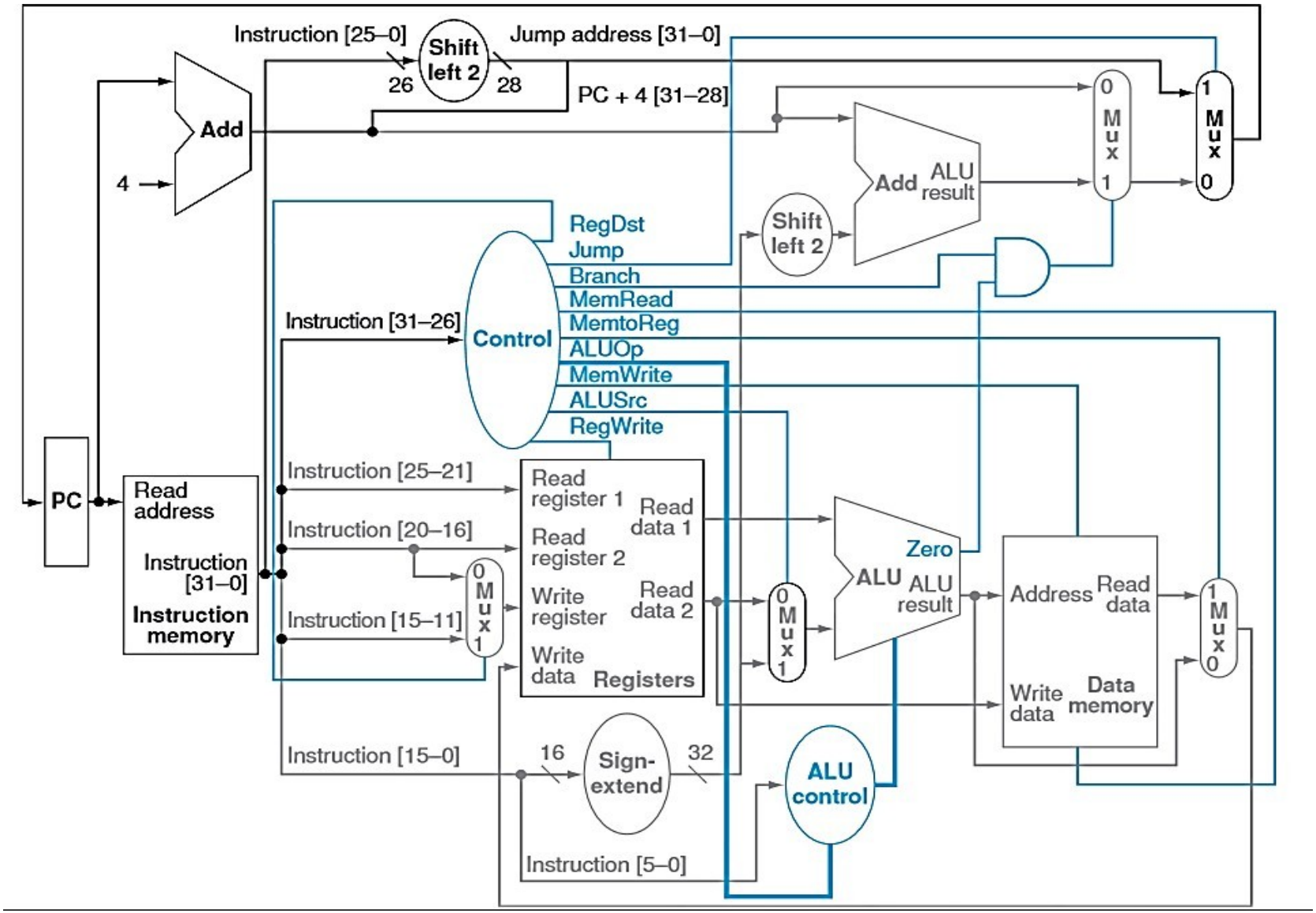
`add $rd, $rs, $rt`

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



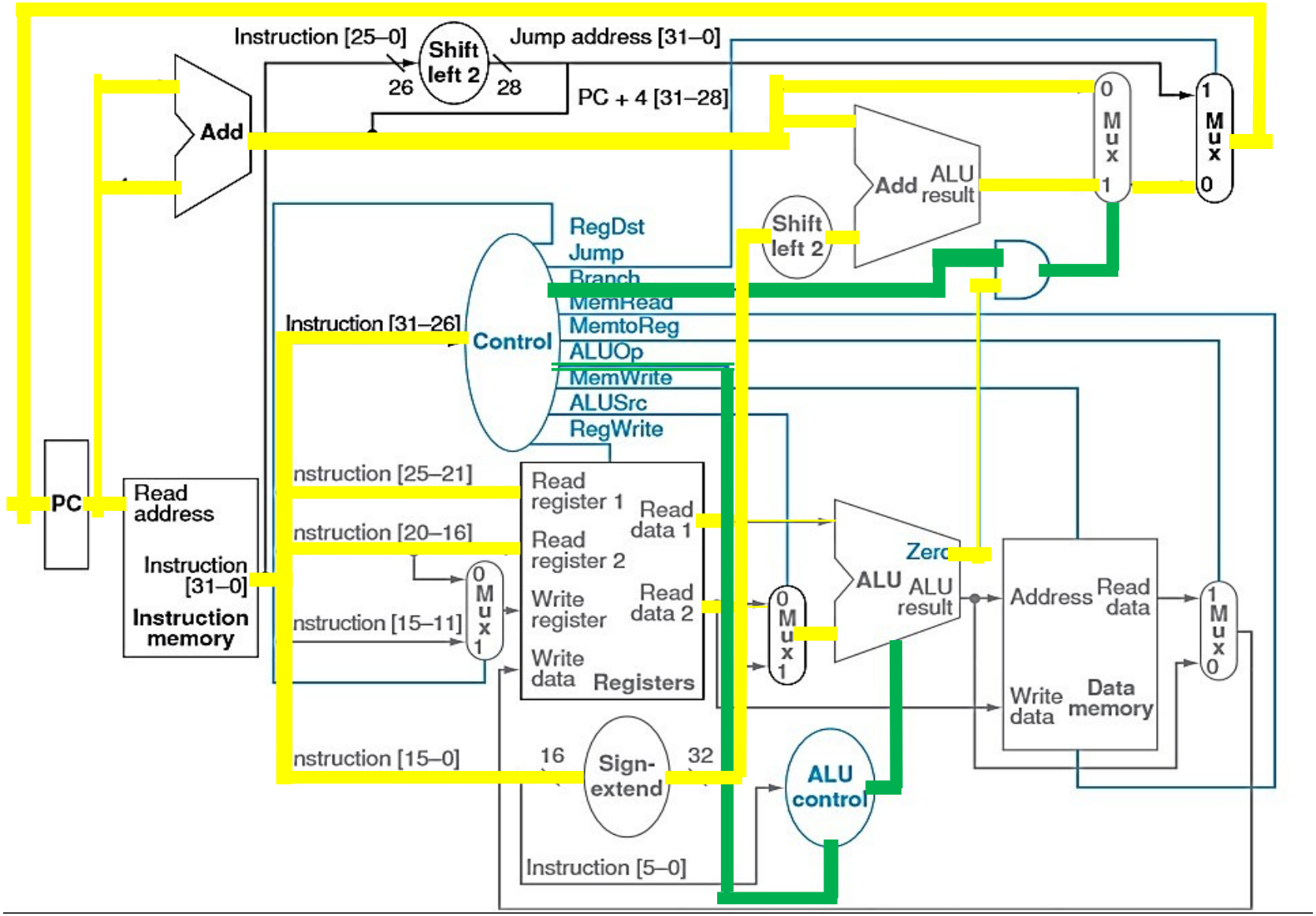
What are the relevant datapath lines for the beq instruction and what are the values of each of the control lines?

beq $rs, $rt, imm

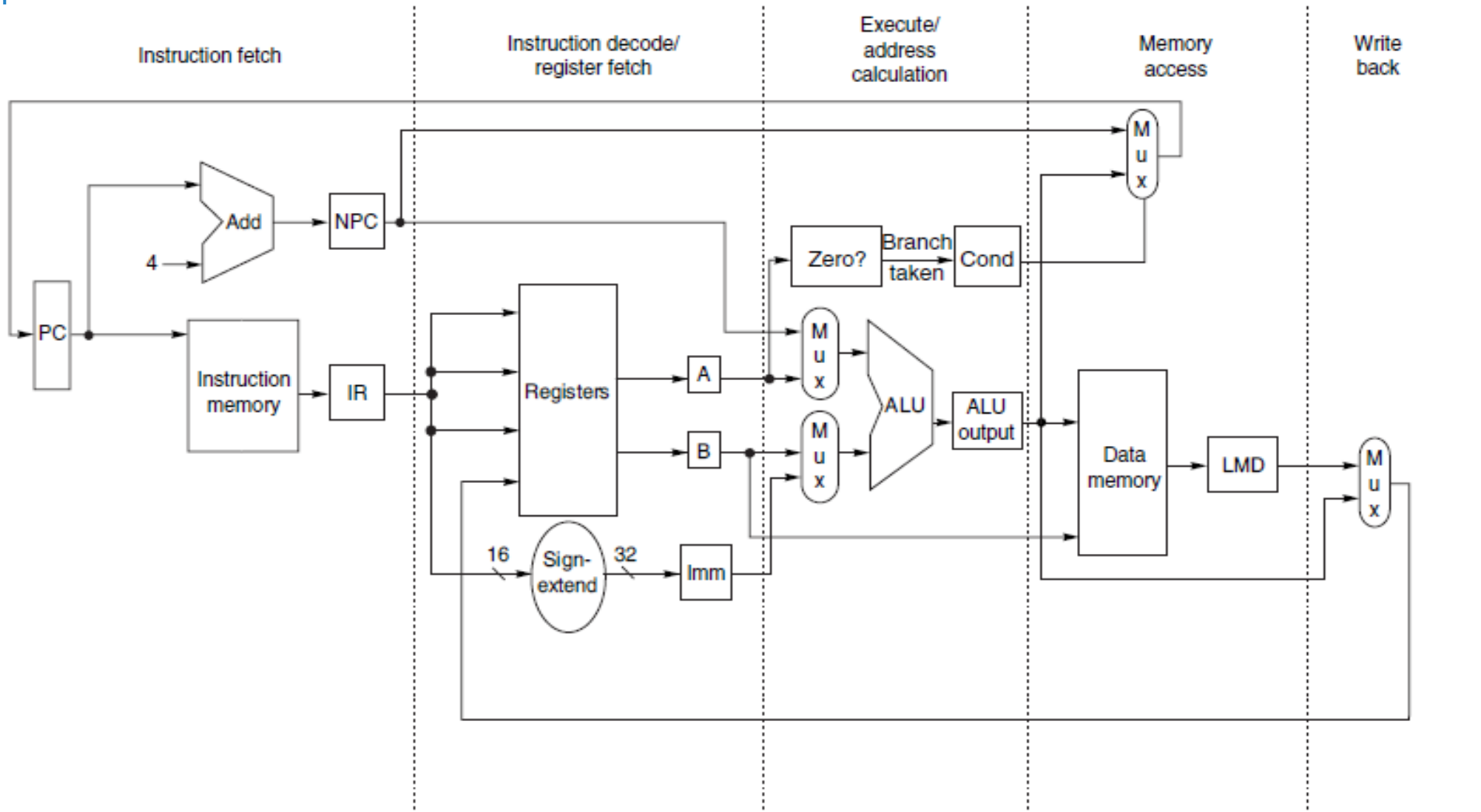


What are the relevant datapath lines for the beq instruction and what are the values of each of the control lines?

beq $rs, $rt, imm



MIPS Data path



Instruction Execution

1. **Instruction Fetch (IF)** - Get the instruction to be executed.

IR ← M[PC]
NPC ← PC + 4

IR – Instruction register
NPC – Next program counter

2. **Instruction Decode/Register Fetch (ID)** – Figure out what the instruction is supposed to do and what it needs.

A ← Register File[Rs]
B ← Register File[Rt]
Imm ← {(IR16)16, IR15..0}

A & B & Imm are temporary registers that hold inputs to the ALU which is in the Execute Stage

3. **Execution (EX)** -The instruction has been decoded, so execution can be split according to instruction type.

Reg-Reg : ALUout A op B
Reg-Imm: ALUout A op Imm
Branch: ALUout NPC + Imm

Cond (A {==, !=} 0)
LD/ST: ALUout A op Imm to form effective Address

4. **Memory Access/Branch Completion (MEM)** – Besides the IF stage this is the only stage that access the memory to load and store data.

Load:	LMD = Mem[ALUout]
Store:	Mem[ALUout] ← B
Branch:	if (cond) PC ← ALUout else PC ← NPC
Jump:	PC ← ALUout

LMD=Load Memory Data Register

5. **Write-Back (WB)** – Store all the results and loads back to registers.

Reg-Reg : Rd ← ALUoutput

Load: Rd ← LMD

Reg-Imm: Rt ← ALUoutput