

# A single-cycle MIPS processor

---

- As previously discussed, an instruction set architecture is an *interface* that defines the hardware operations which are available to software.
- Any instruction set can be implemented in many different ways. Over the next few weeks we'll compare two important implementations.
  - In a basic **single-cycle implementation** all operations take the same amount of time—a single cycle.
  - In a **pipelined implementation**, a processor can overlap the execution of several instructions, potentially leading to big performance gains.

# Single-cycle implementation

---

- In lecture, we will describe the implementation a simple MIPS-based instruction set supporting just the following operations.

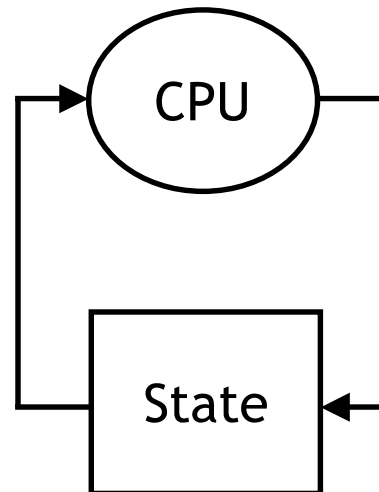
Arithmetic:	add	sub	and	or	slt
Data Transfer:	lw	sw			
Control:	beq				

- We use MIPS because it is significantly easier to implement than x86.
- Today we'll build a **single-cycle implementation** of this instruction set.
  - All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations.
  - We'll explain the datapath, and briefly describe the control unit.

# Computers are state machines

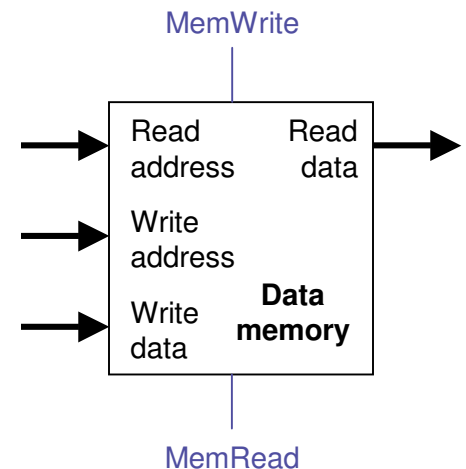
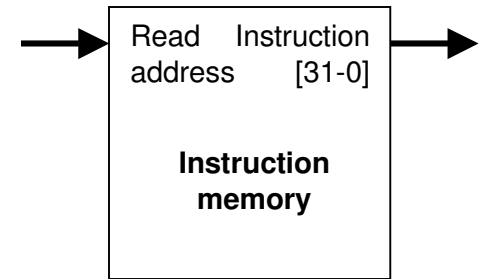
---

- A computer is just a big fancy **state machine**.
  - Registers, memory, hard disks and other storage form the state.
  - The processor keeps reading and updating the state, according to the instructions in some program.
- Theory classes like (the new) CS273 explicitly model computers as state machines or finite automata.



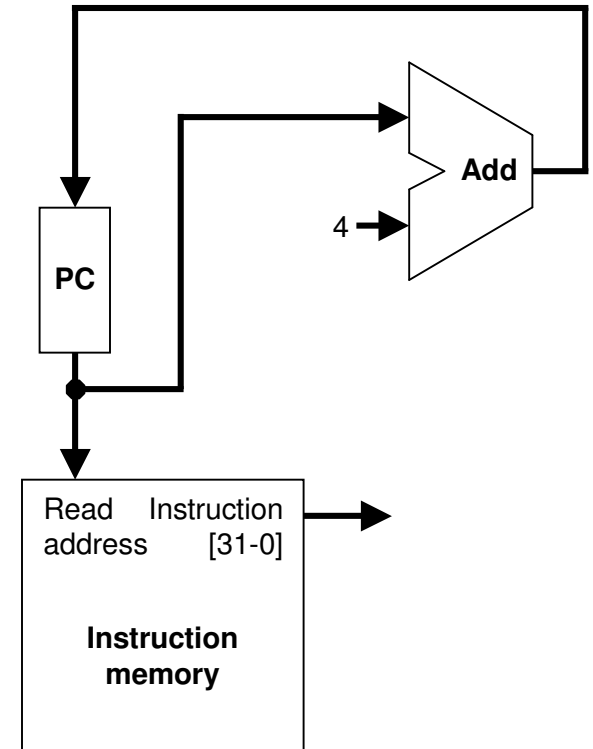
# Memories

- It's easier to use a **Harvard architecture** at first, with instructions and data stored in *separate* memories.
- To fetch instructions and read & write words, we need these memories to be 32-bits wide (buses are represented by dark lines here).
- Blue lines represent control signals. **MemRead** and **MemWrite** should be set to 1 if the **data memory** is to be read or written respectively, and 0 otherwise.
- For now, we will assume you cannot write to the **instruction memory**.
  - Pretend it's already loaded with a program, which doesn't change while it's running.



# Instruction fetching

- The CPU is always in an infinite loop, fetching instructions from memory and executing them.
- The **program counter** or **PC** register holds the address of the current instruction.
- MIPS instructions are each four bytes long, so the PC should be incremented by four to read the next instruction in sequence.



# Decoding instructions (R-type)

- A few weeks ago, we saw encodings of MIPS instructions as 32-bit values.
- Register-to-register arithmetic instructions use the **R-type** format.
  - **op** is the instruction opcode, and **func** specifies a particular arithmetic operation
  - **rs**, **rt** and **rd** are source and destination registers.

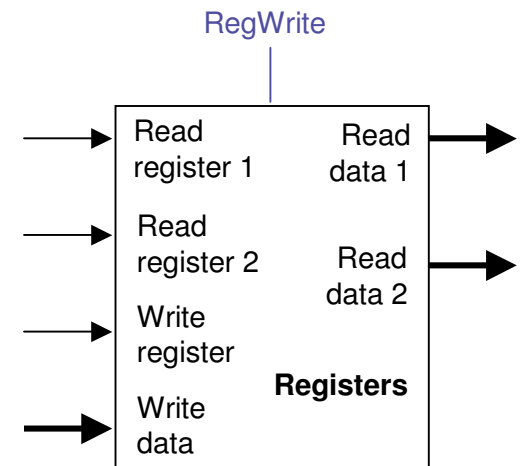
op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- An example instruction and its encoding:

add	\$s4	,	\$t1	,	\$t2	000000	01001	01010	10100	00000	1000000
-----	------	---	------	---	------	--------	-------	-------	-------	-------	---------

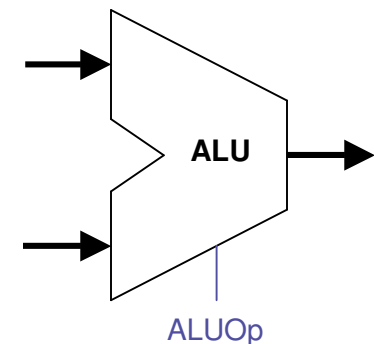
# Registers and ALUs

- R-type instructions must access registers and an ALU.
- Our **register file** stores thirty-two 32-bit values.
  - Each register specifier is 5 bits long.
  - You can read from two registers at a time.
  - **RegWrite** is 1 if a register should be written.



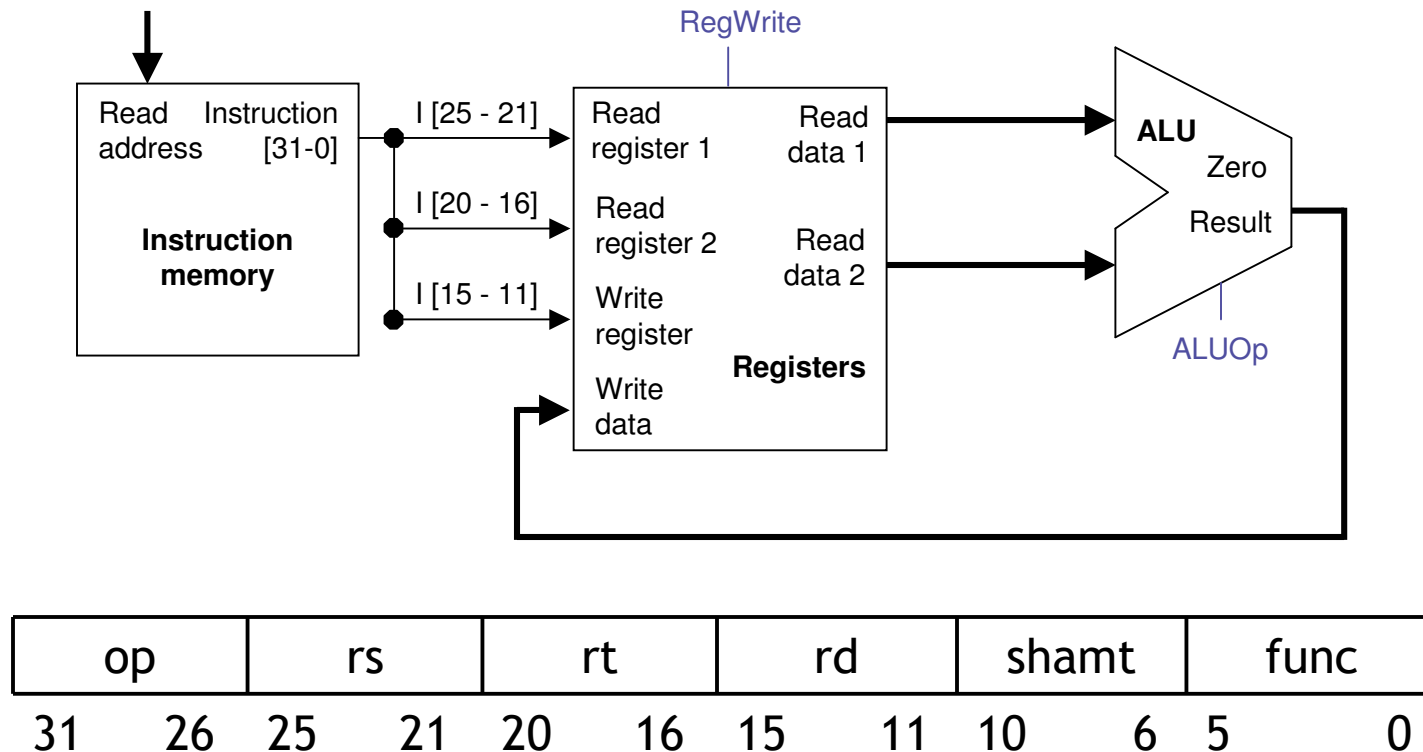
- Here's a simple **ALU** with five operations, selected by a 3-bit control signal **ALUOp**.

ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt



# Executing instructions (R-type)

1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields **rs** and **rt**, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field **rd** of the instruction word.





# Decoding I-type instructions

- The lw, sw and beq instructions all use the **I-type** encoding.
  - **rt** is the *destination* for lw, but a *source* for beq and sw.
  - **address** is a 16-bit signed constant.

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

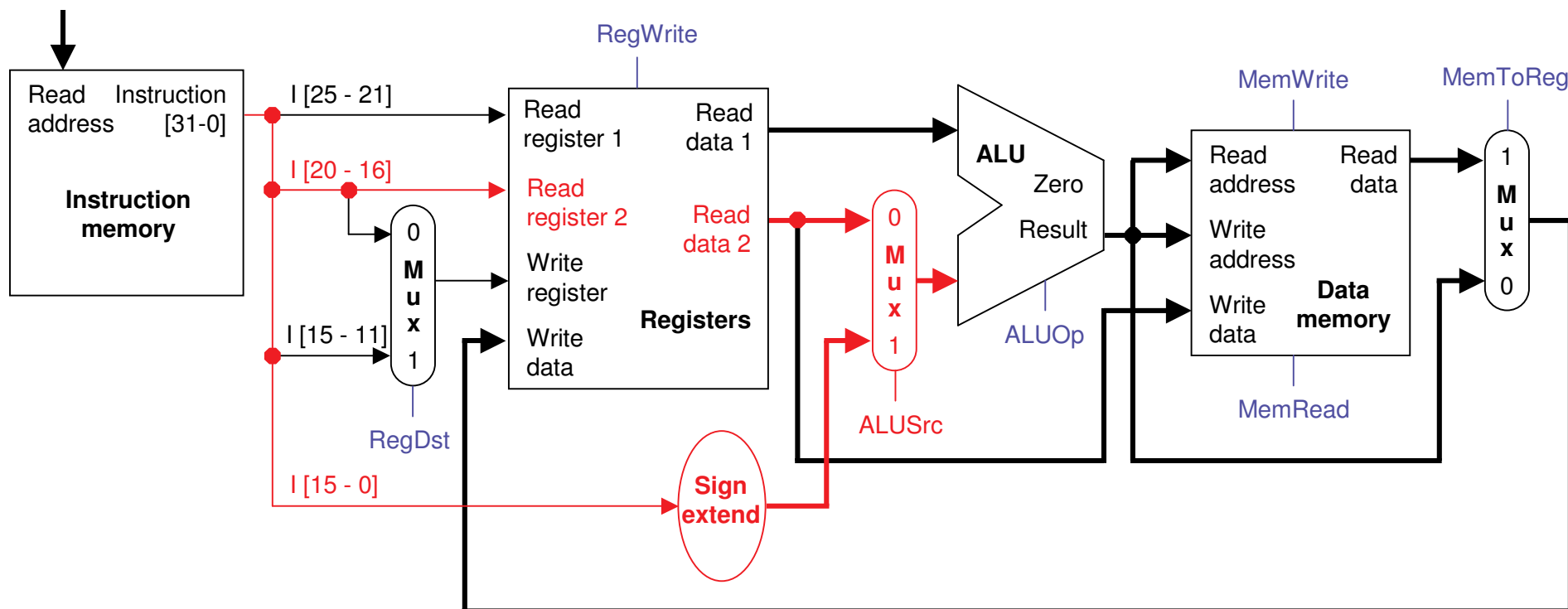
- Two example instructions:

lw	\$t0, -4(\$sp)	100011	11101	01000	1111 1111 1111 1100
----	----------------	--------	-------	-------	---------------------

sw	\$a0, 16(\$sp)	101011	11101	00100	0000 0000 0001 0000
----	----------------	--------	-------	-------	---------------------

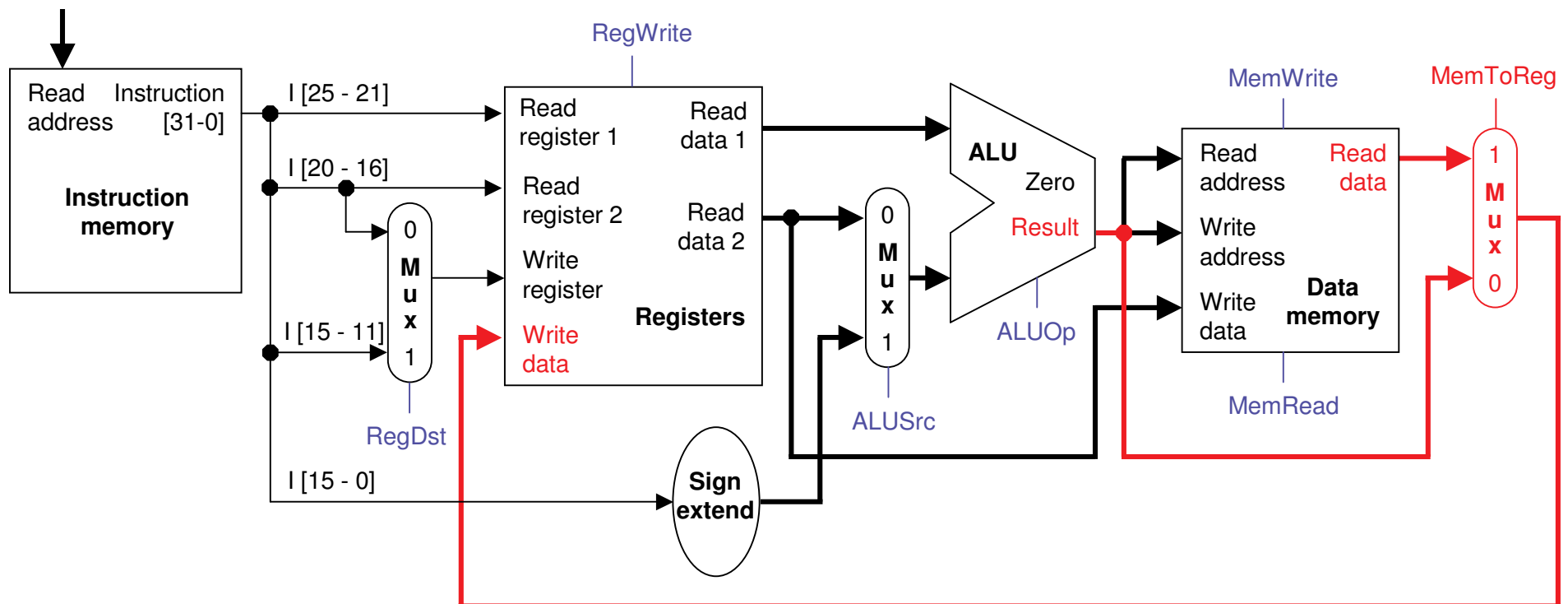
# Accessing data memory

- For an instruction like `lw $t0, -4($sp)`, the base register `$sp` is added to the *sign-extended* constant to get a data memory address.
- This means the ALU must accept *either* a register operand for arithmetic instructions, *or* a sign-extended immediate operand for `lw` and `sw`.
- We'll add a multiplexer, controlled by `ALUSrc`, to select either a register operand (0) or a constant operand (1).



# MemToReg

- The register file's "Write data" input has a similar problem. It must be able to store *either* the ALU output of R-type instructions, *or* the data memory output for lw.
- We add a mux, controlled by **MemToReg**, to select between saving the ALU result (0) or the data memory output (1) to the registers.



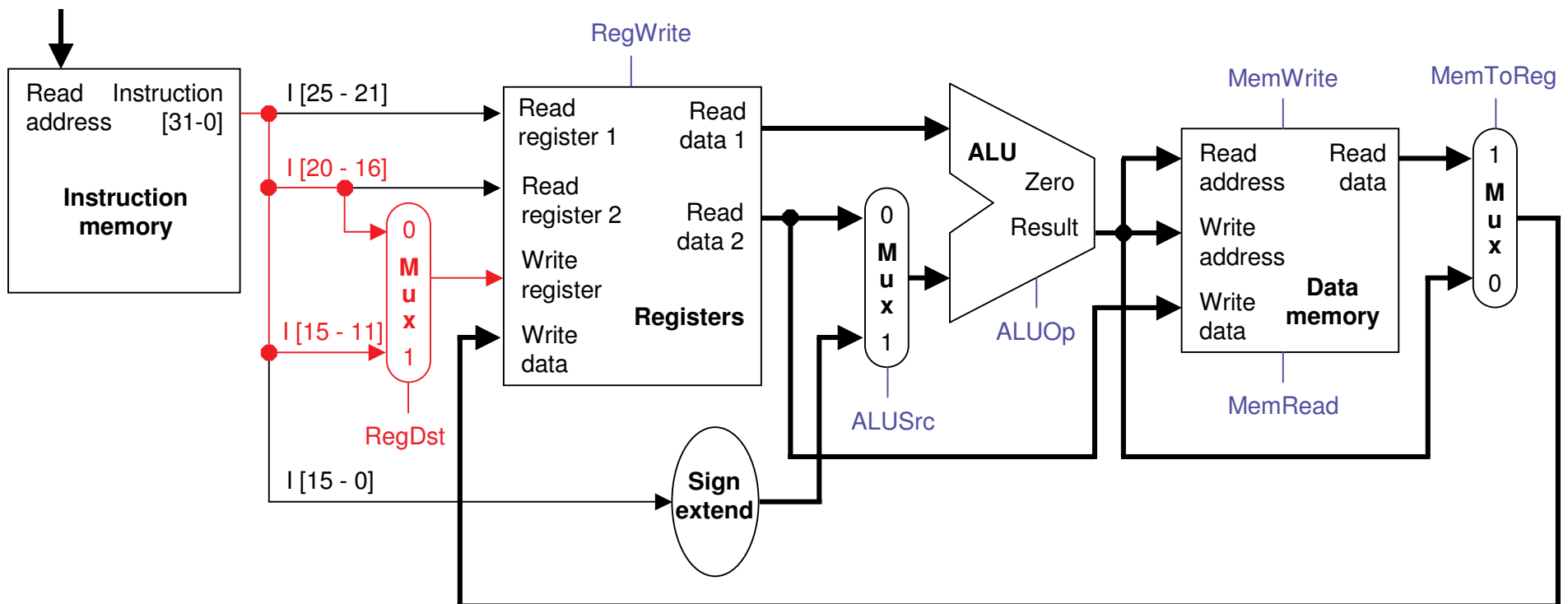
# RegDst

- A final annoyance is the destination register of lw is in *rt* instead of *rd*.

op	rs	rt	address
----	----	----	---------

lw \$rt, address(\$rs)

- We'll add one more mux, controlled by *RegDst*, to select the destination register from either instruction field *rt* (0) or field *rd* (1).

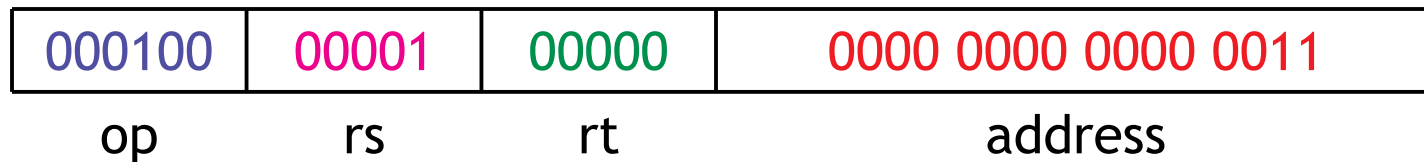


# Branches

- For branch instructions, the constant is not an address but an *instruction offset* from the current program counter to the desired address.

```
    beq  $a0, $0, L
    add  $v1, $v0, $0
    add  $v1, $v1, $v1
    j    Somewhere
L:      add  $v1, $v0, $v0
```

- The target address L is three *instructions* past the `beq`, so the encoding of the branch instruction has 0000 0000 0000 0011 for the address field.



- Instructions are four bytes long, so the actual memory offset is 12 bytes.

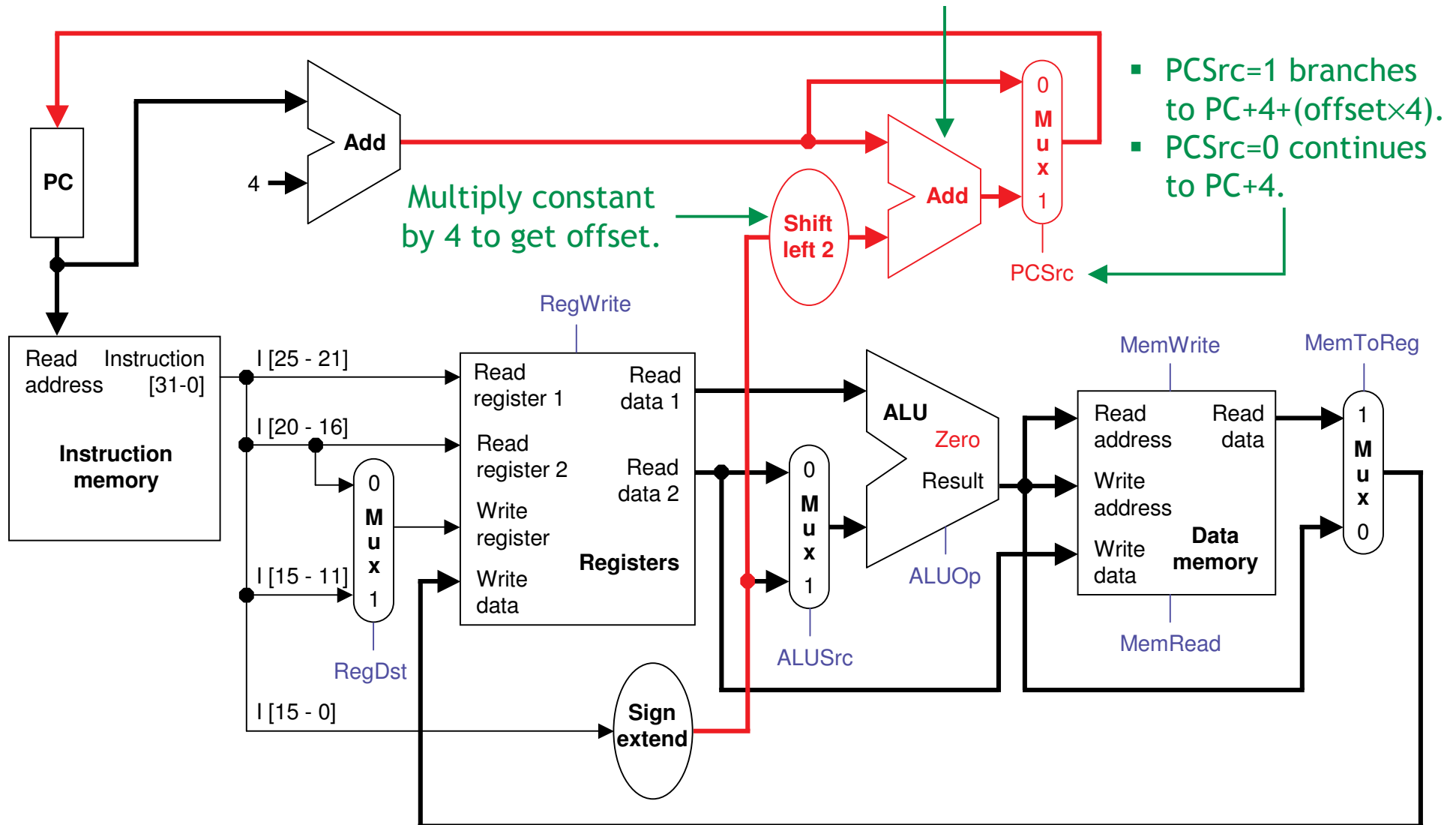
# The steps in executing a beq

---

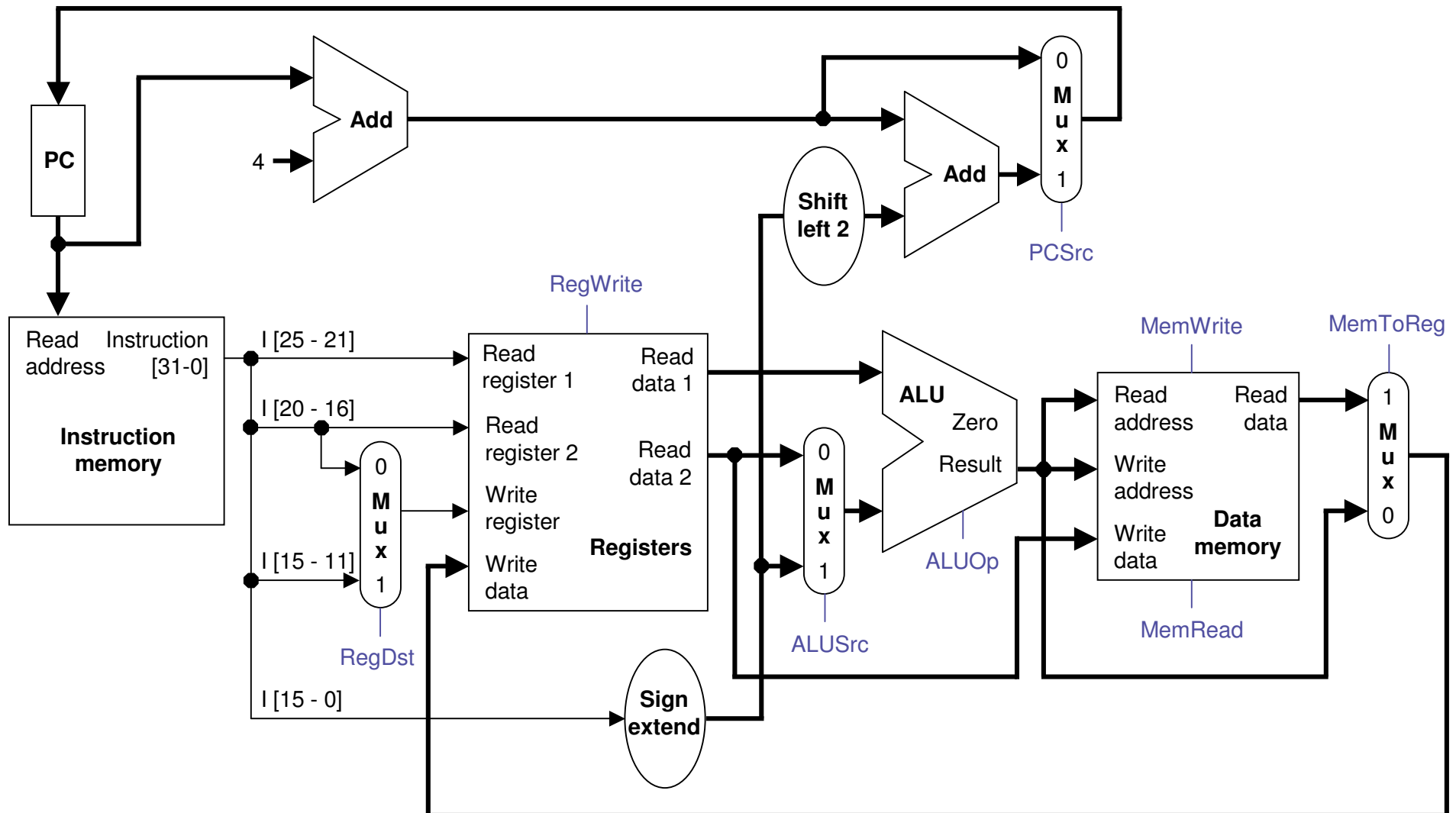
1. Fetch the instruction, like `beq $at, $0, offset`, from memory.
2. Read the source registers, `$at` and `$0`, from the register file.
3. Compare the values by subtracting them in the ALU.
4. If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address,  $PC + 4 + (offset \times 4)$ .
5. Otherwise the branch should not be taken, and the PC should just be incremented to  $PC + 4$  to fetch the next instruction sequentially.

# Branching hardware

We need a second adder, since the ALU is already doing subtraction for the beq.



# The final datapath





# Control

---

- The **control unit** is responsible for setting all the control signals so that each instruction is executed properly.
  - The control unit's input is the 32-bit instruction word.
  - The outputs are values for the blue control signals in the datapath.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.

# Summary

---

- A **datapath** contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our **single-cycle implementation**, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The **control unit** tells the datapath what to do, based on the instruction that's currently being executed.
  - Our processor has ten **control signals** that regulate the datapath.
  - The control signals can be generated by a combinational circuit with the instruction's 32-bit binary encoding as input.
- Next week, we'll see the performance limitations of this single-cycle machine and discuss how to improve upon it.