CS250
Computer Architecture

Fall 2012

Part 6: A Single-Cycle MIPS Processor

Acknowledgment for Lecture Notes Authorship

- Prof. Craig Zilles
- Mr. Howard Huang

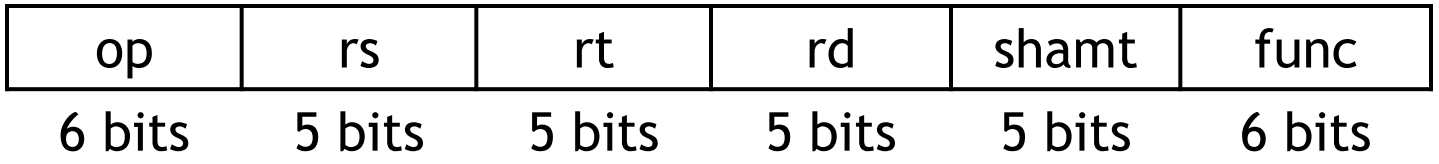
  Both with the Department of Computer Science at UIUC

Assembly vs. machine language

- So far we've been using assembly language.
  - We assign names to operations (e.g., add) and operands (e.g., $t0).
  - Branches and jumps use labels instead of actual addresses.
  - Assemblers support many pseudo-instructions.
- Programs must eventually be translated into machine language, a binary format that can be stored in memory and decoded by the CPU.
- MIPS machine language is designed to be easy to decode.
  - Each MIPS instruction is the same length, 32 bits.
  - There are only three different instruction formats, which are very similar to each other.
- Studying MIPS machine language will also reveal some restrictions in the instruction set architecture, and how they can be overcome.

R-type format

- Register-to-register arithmetic instructions use the R-type format.



- This format includes six different fields.
  - op is an operation code or opcode that selects a specific operation.
  - rs and rt are the first and second source registers.
  - rd is the destination register.
  - shamt is only used for shift instructions.
  - func is used together with op to select an arithmetic instruction.

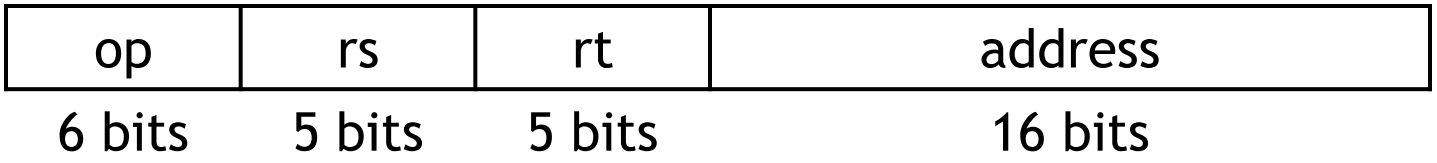
About the registers

- We have to encode register names as 5-bit numbers from 00000 to 11111.
  - For example, $t8 is register $24, which is represented as 11000.

- The number of registers available affects the instruction length.
  - Each R-type instruction references 3 registers, which requires a total of 15 bits in the instruction word.
  - We can't add more registers without either making instructions longer than 32 bits, or shortening other fields like op and possibly reducing the number of available operations.

I-type format

- Load, store, branch and immediate instructions all use the I-type format.



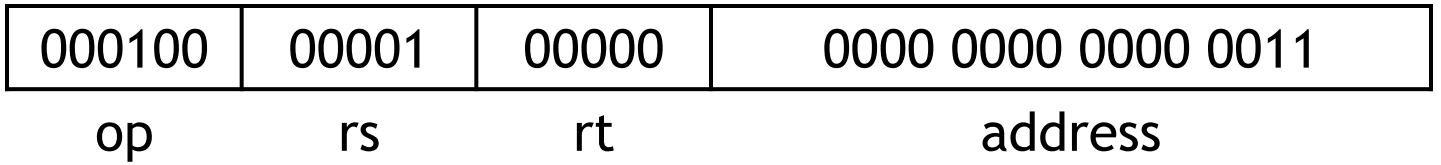
- For uniformity, op, rs and rt are in the same positions as in the R-format.
- The meaning of the register fields depends on the exact instruction.
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions.
  - rt is a source register for branches and stores, but a destination register for the other I-type instructions.
- The address is a 16-bit signed two's-complement value.
  - It can range from -32,768 to +32,767

Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current program counter (PC) to the target address.

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

- Since the branch target L is three *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:



Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if "Far" is very far away, then the effect of:

```
        beq  $s0, $s1, Far
        ...
```

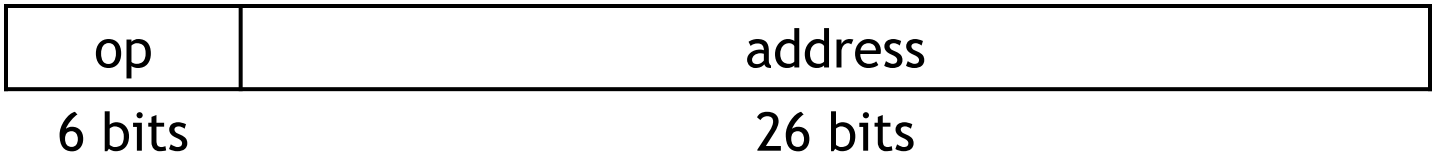
can be simulated with the following actual code.

```
        bne  $s0, $s1, Next
        j    Far
Next:   ...
```

- Again, the MIPS designers have taken care of the common case first.

J-type format

- Finally, the jump instruction uses the J-type instruction format.



- The jump instruction contains a *word* address, not an offset
  - Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - So instead of saying "jump to address 4000," it's enough to just say "jump to instruction 1000."
  - A 26-bit address field lets you jump to any address from 0 to 2^{28}.

- For even longer jumps, the jump register, or jr, instruction can be used.

```
jr  $ra      # Jump to 32-bit address in register $ra
```

Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

Null-terminated Strings

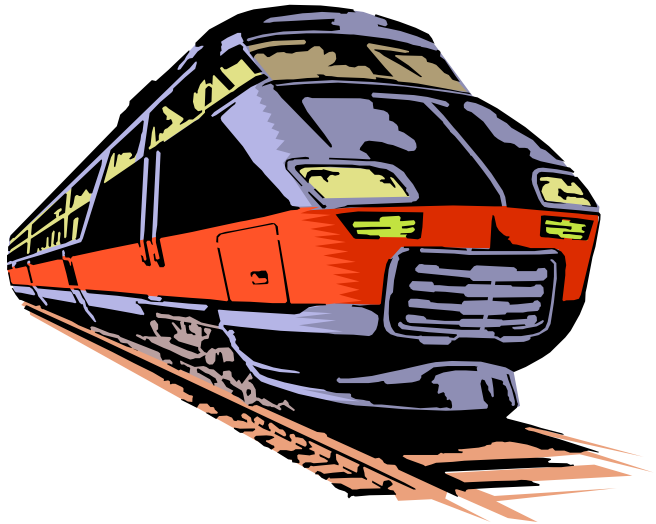
- For example, “Harry Potter” can be stored as a 13-byte array.

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- Since strings can vary in length, we put a 0, or null, at the end of the string.
  - This is called a null-terminated string

A single-cycle MIPS processor

- As previously discussed, an instruction set architecture is an *interface* that defines the hardware operations that 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.

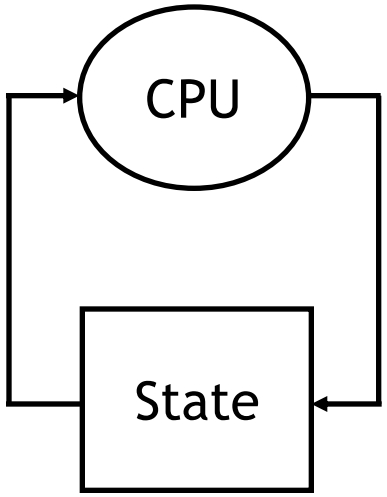
Arithmetic:	add	sub	and	or	slt
Data Transfer:	lw	sw			
Control:	beq				

  - 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 first, and then make the control unit.

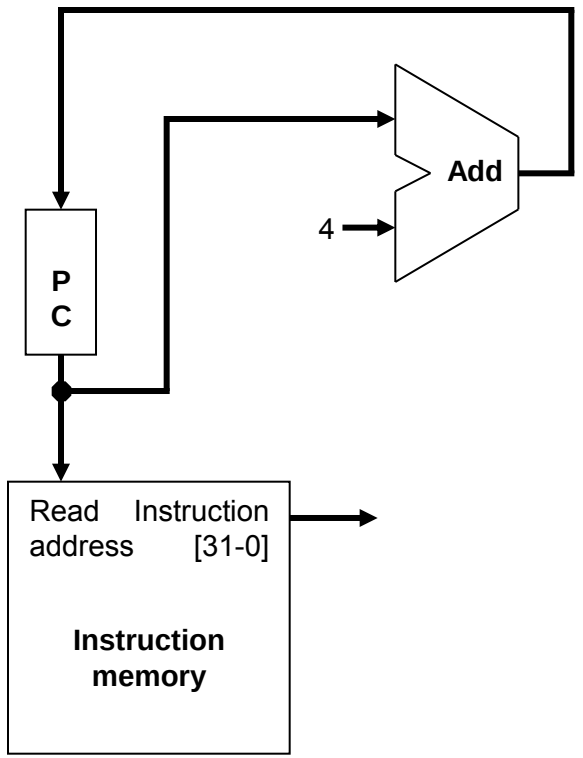
Computers are state machines

- 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.



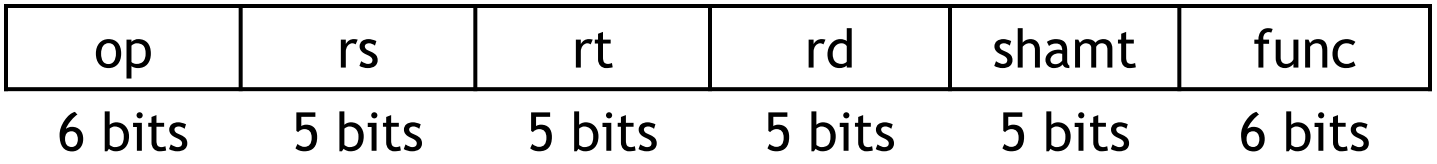
Instruction fetching

- 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.



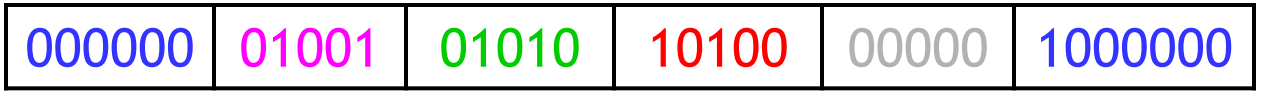
Encoding R-type instructions

- MIPS instructions are 32-bit words.
- 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.



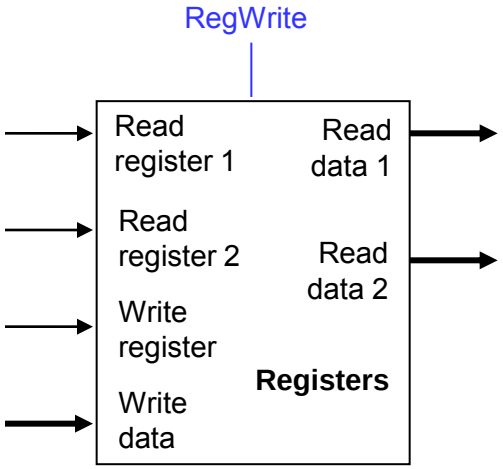
- An example instruction and its encoding:

add $s4, $t1, $t2



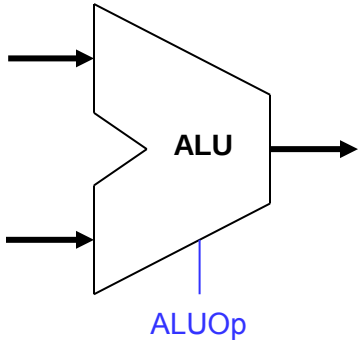
Registers and ALUs

- 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.

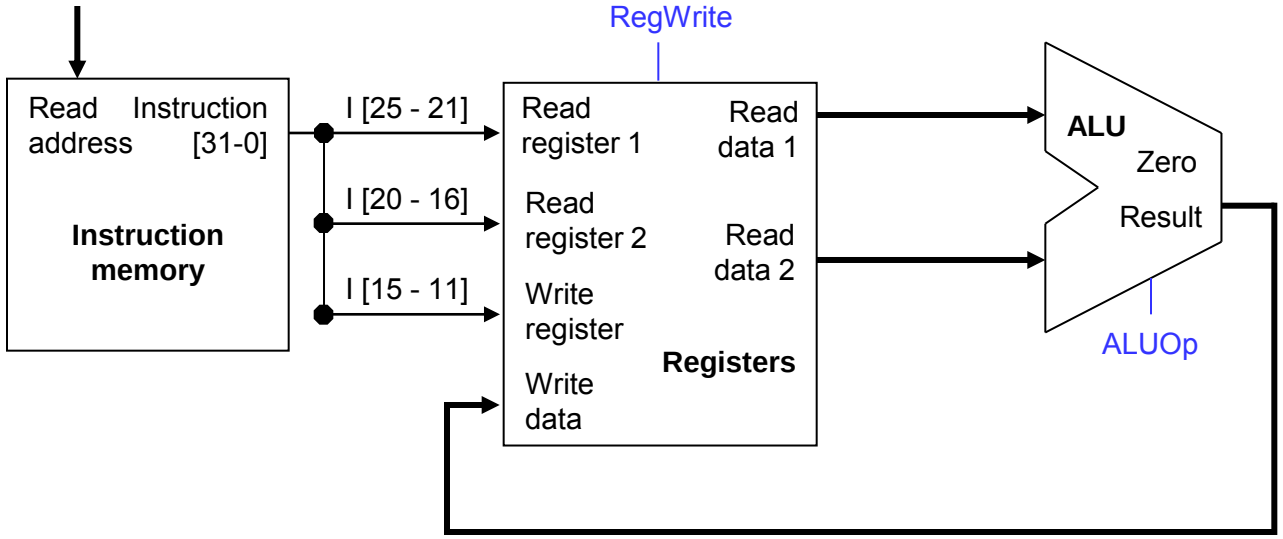
- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.

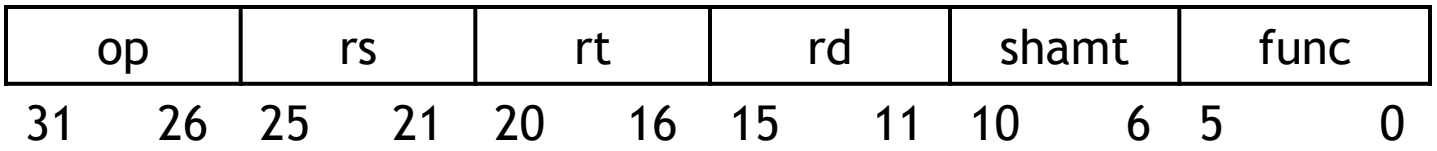


ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt

Executing an R-type instruction

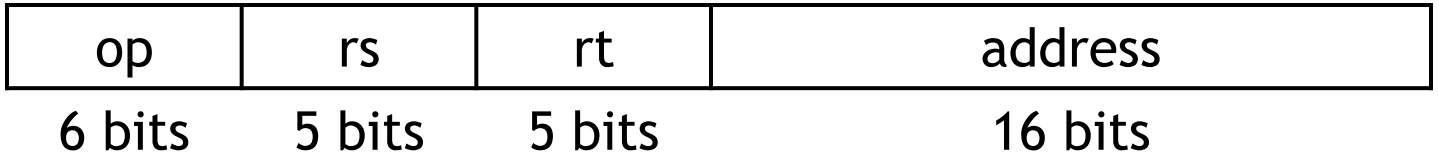
1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction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





Encoding 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.

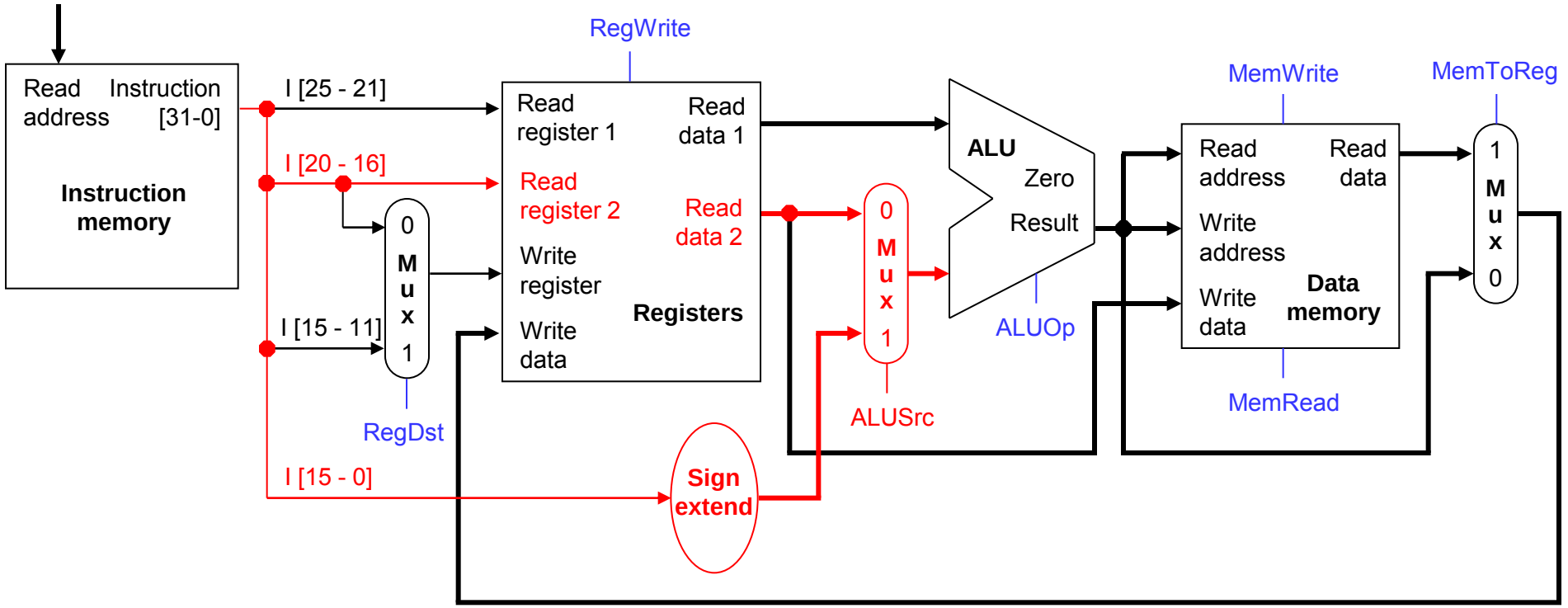


- Two example instructions:

lw $t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100
sw $a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

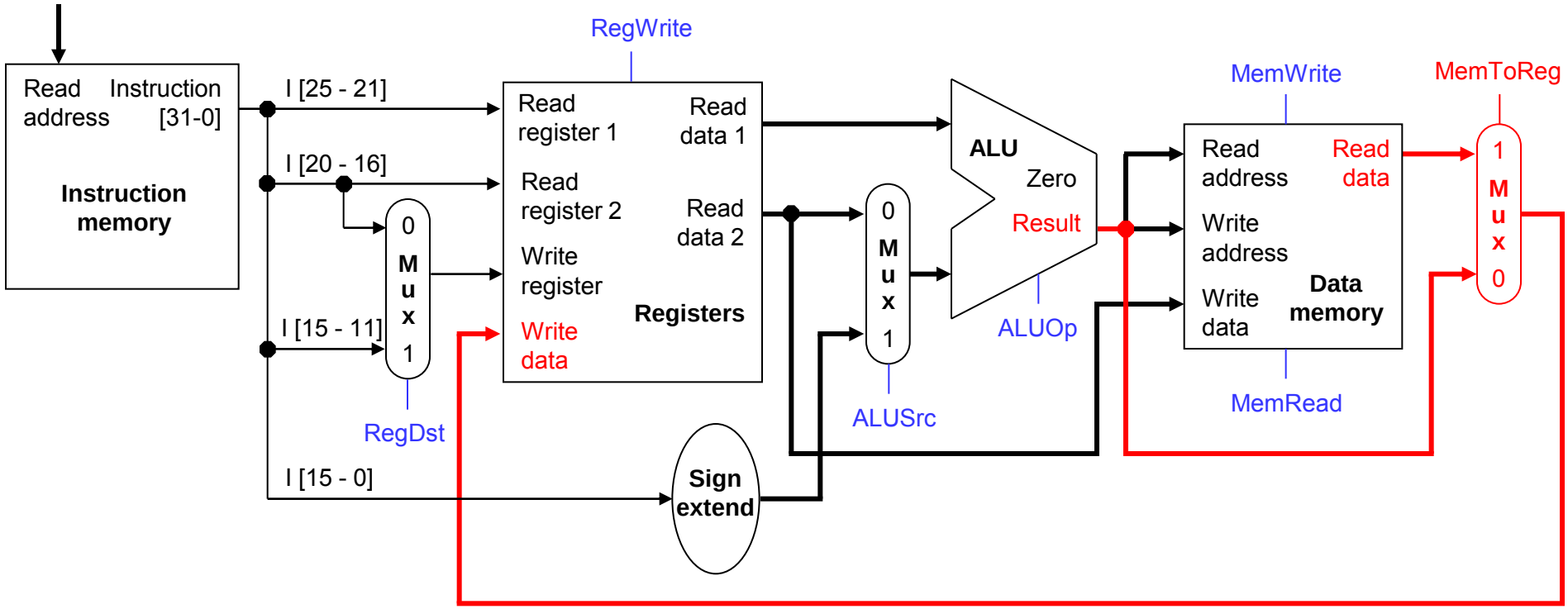
Accessing data memory

- 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.



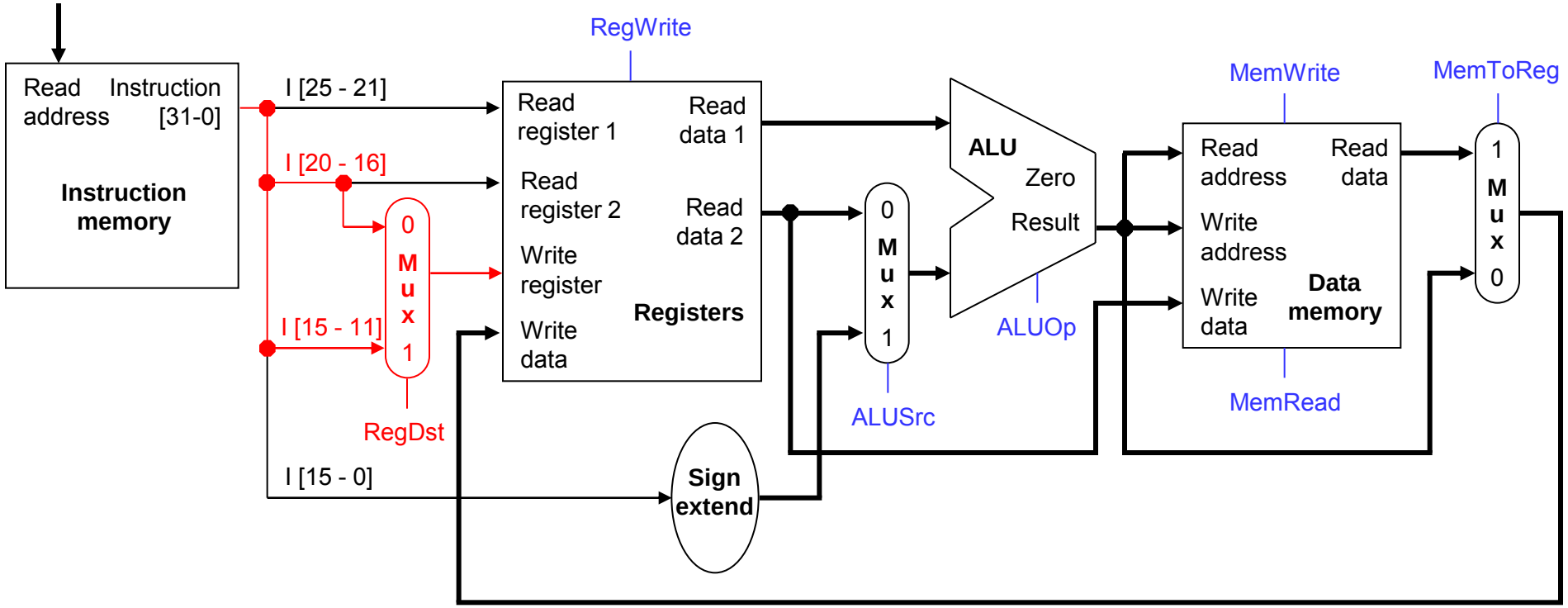
RegDst

- 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).

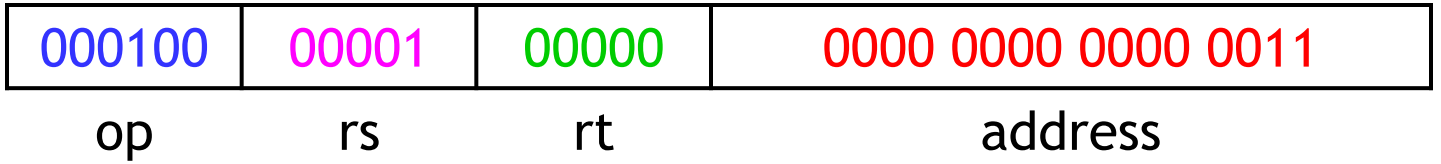


Branches

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

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

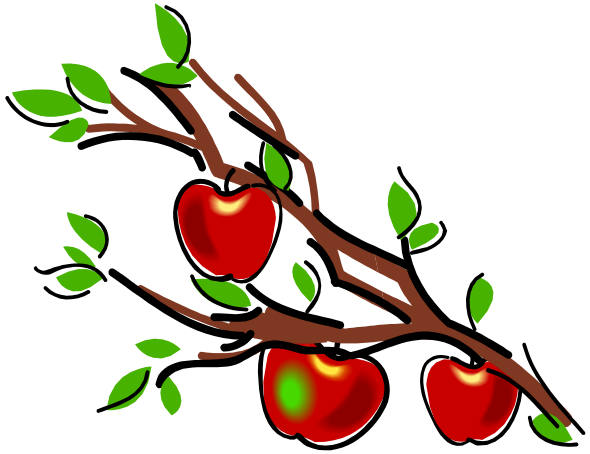
- The target address L is three *instructions* past the beq, so the encoding of the branch instruction has 0000 0000 0000 0011 for the address field.



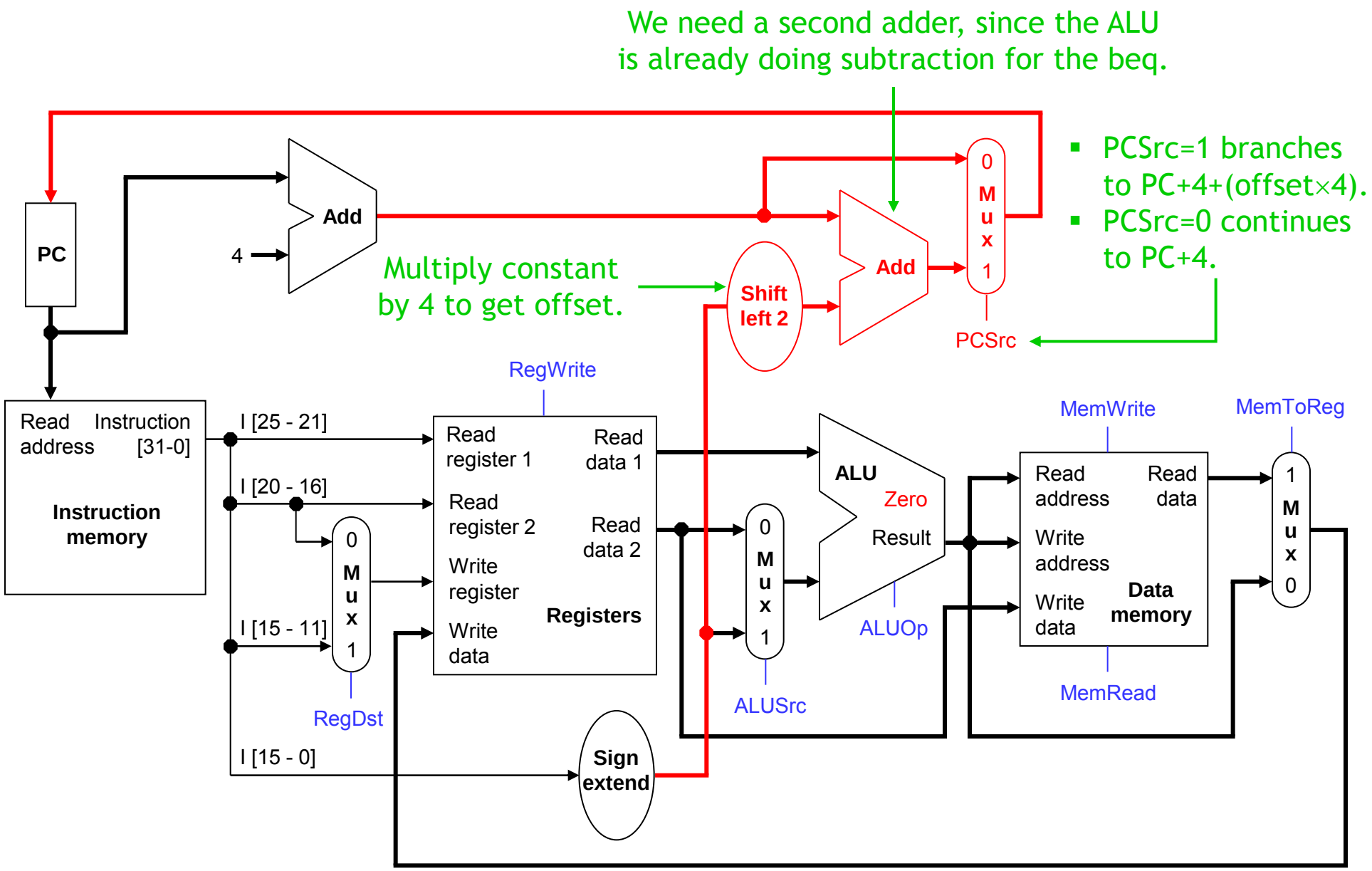
- Instructions are four bytes long, so the actual memory offset is 12 bytes.

The steps in executing a beq

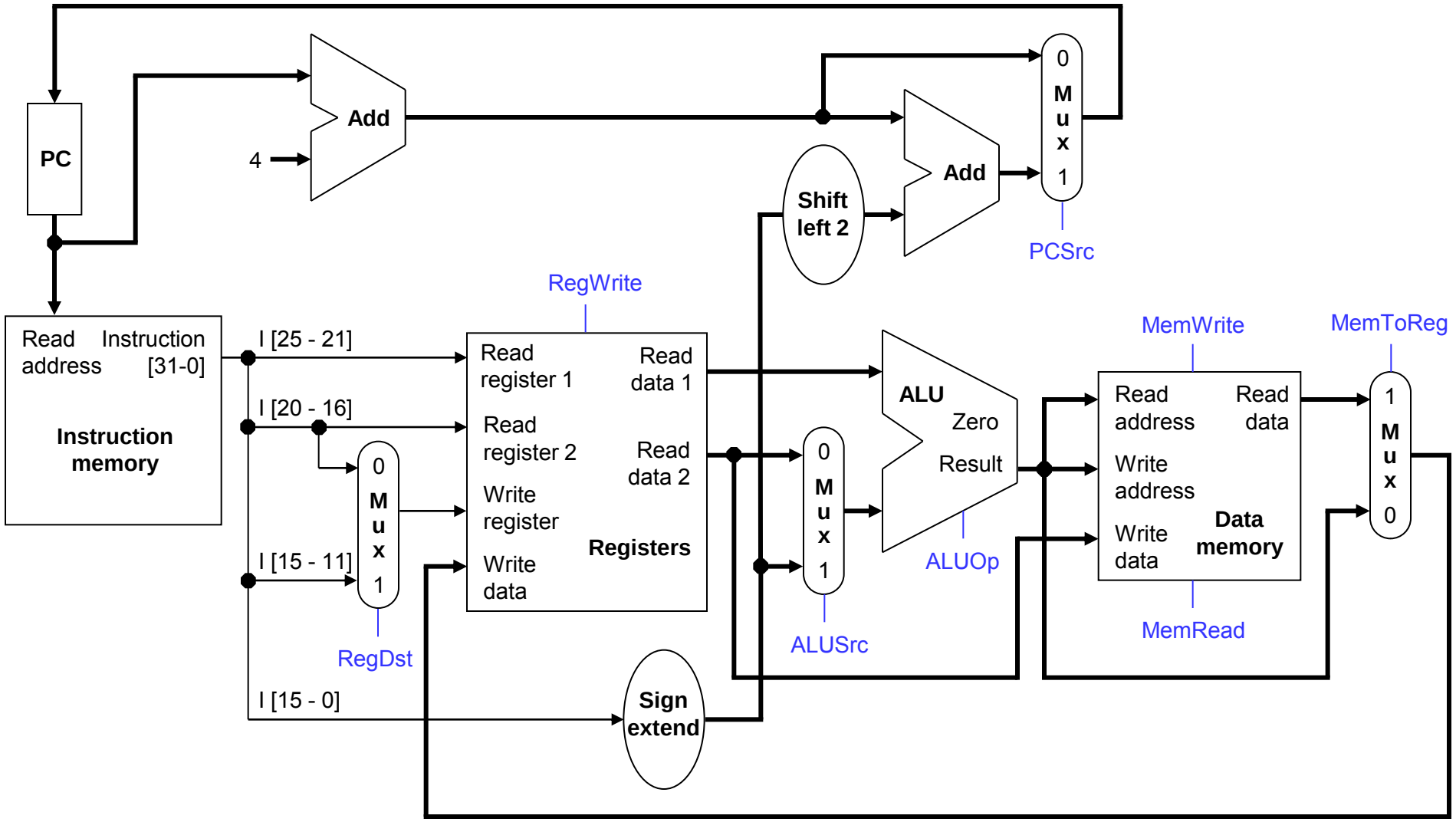
1. Fetch the instruction, like beq $at, $0, offset, from memory.
2. Read the source registers, $at and $0, from the register file.
3. Compare the values by subtracting them in the ALU.
4. If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address, PC + 4 + (offset x 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



The final datapath

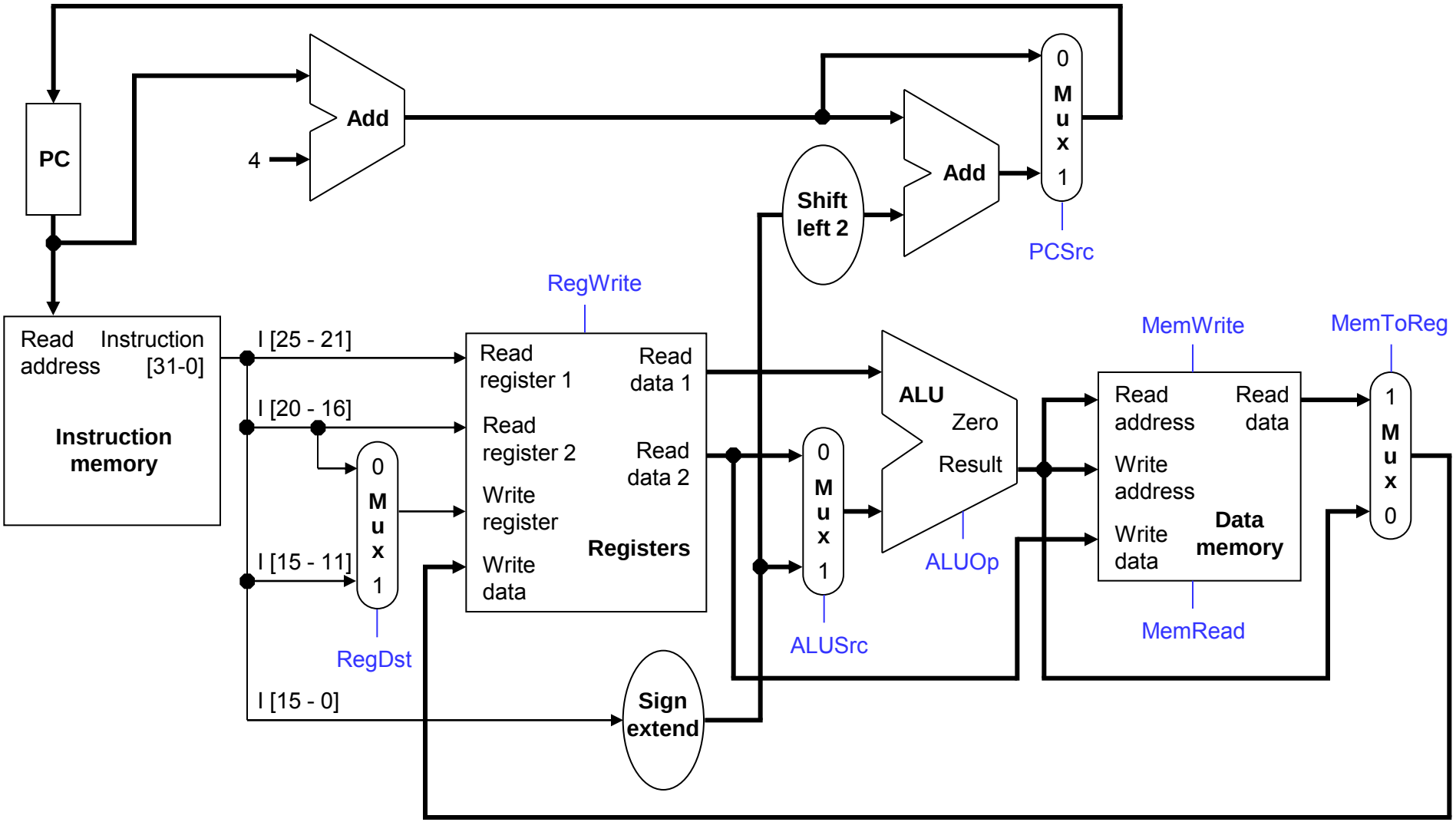


Control

- 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.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

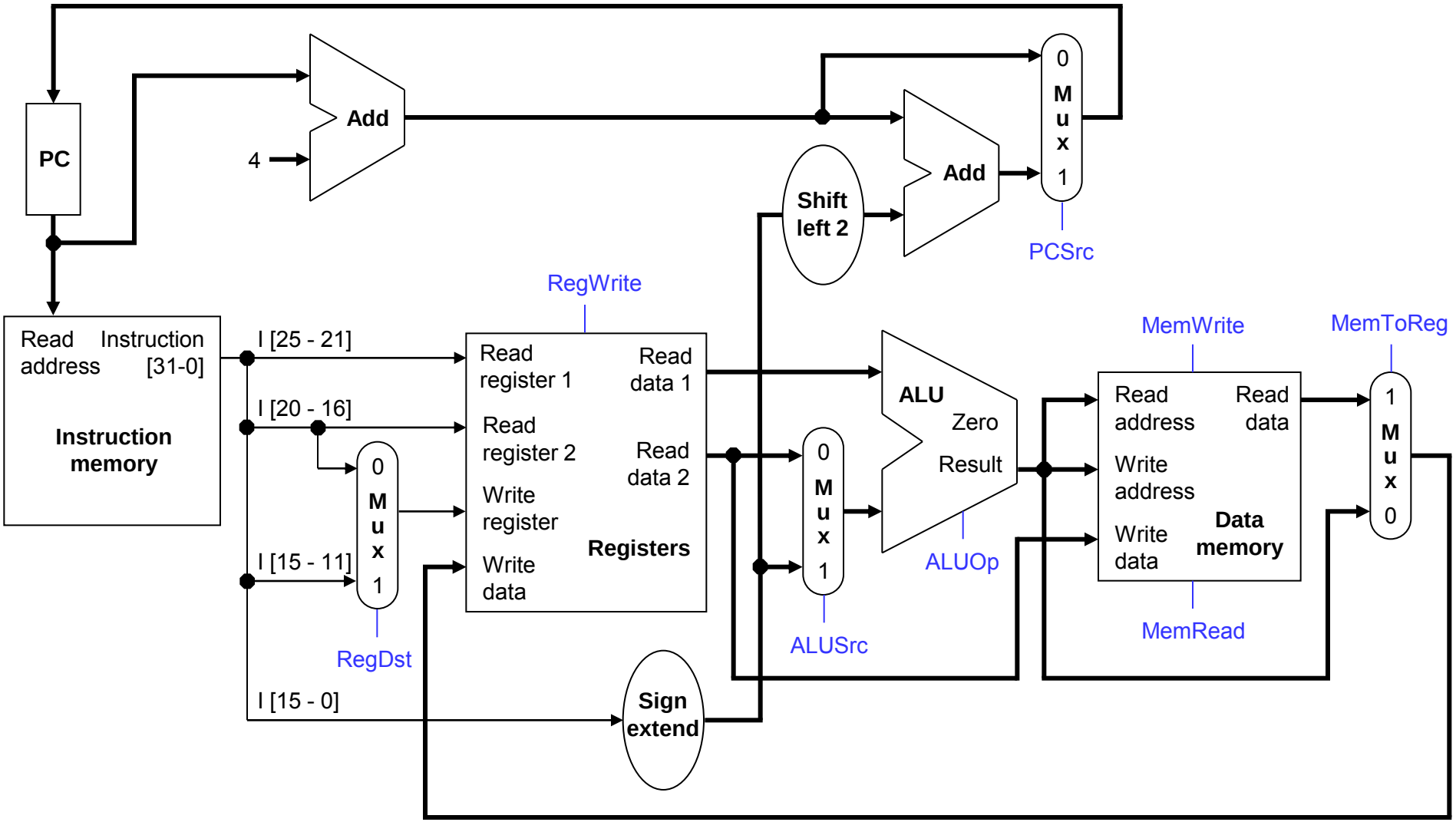
R-type instruction path

- The R-type instructions include add, sub, and, or, and slt.
- The ALUOp is determined by the instruction's "func" field.



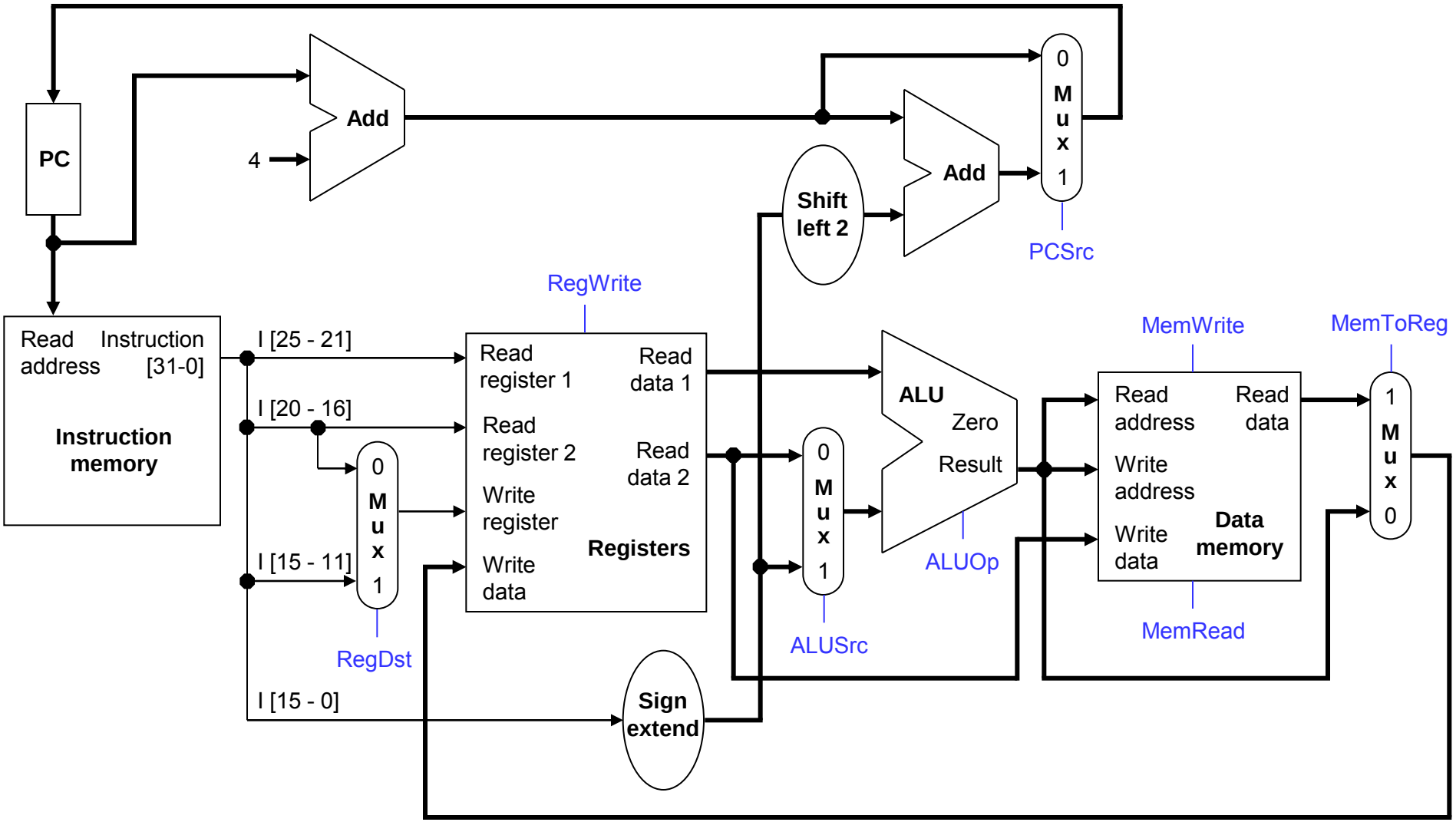
lw instruction path

- An example load instruction is lw $t0, -4($sp).
- The ALUOp must be 010 (add), to compute the effective address.



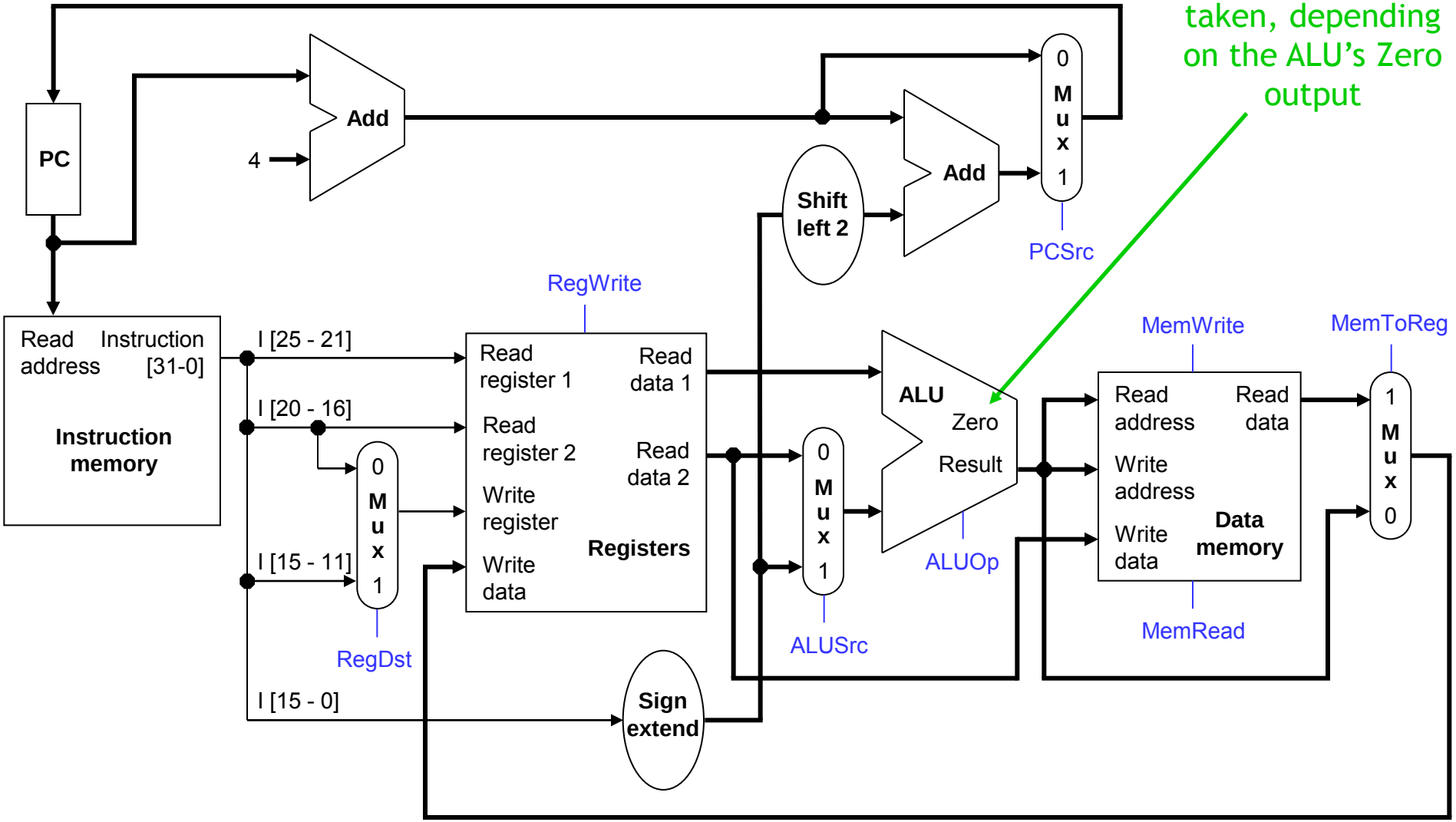
sw instruction path

- An example store instruction is sw $a0, 16($sp).
- The ALUOp must be 010 (add), again to compute the effective address.



beq instruction path

- One sample branch instruction is beq $at, $0, offset.
- The ALUOp is 110 (subtract), to test for equality.



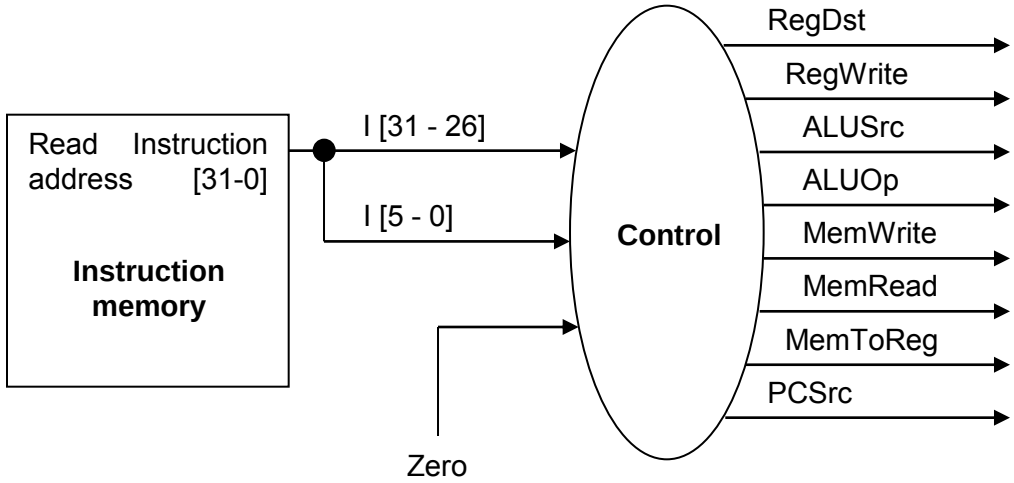
Control signal table

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq *and* the ALU's Zero output is true.

Generating control signals

- The control unit needs 13 bits of inputs.
  - Six bits make up the instruction's opcode.
  - Six bits come from the instruction's func field.
  - It also needs the Zero output of the ALU.
- The control unit generates 10 bits of output, corresponding to the signals mentioned on the previous page.
- You can build the actual circuit by using big K-maps, big Boolean algebra, or big circuit design programs.



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.
- Up next, we'll see the performance limitations of this single-cycle machine and discuss how to improve upon it.