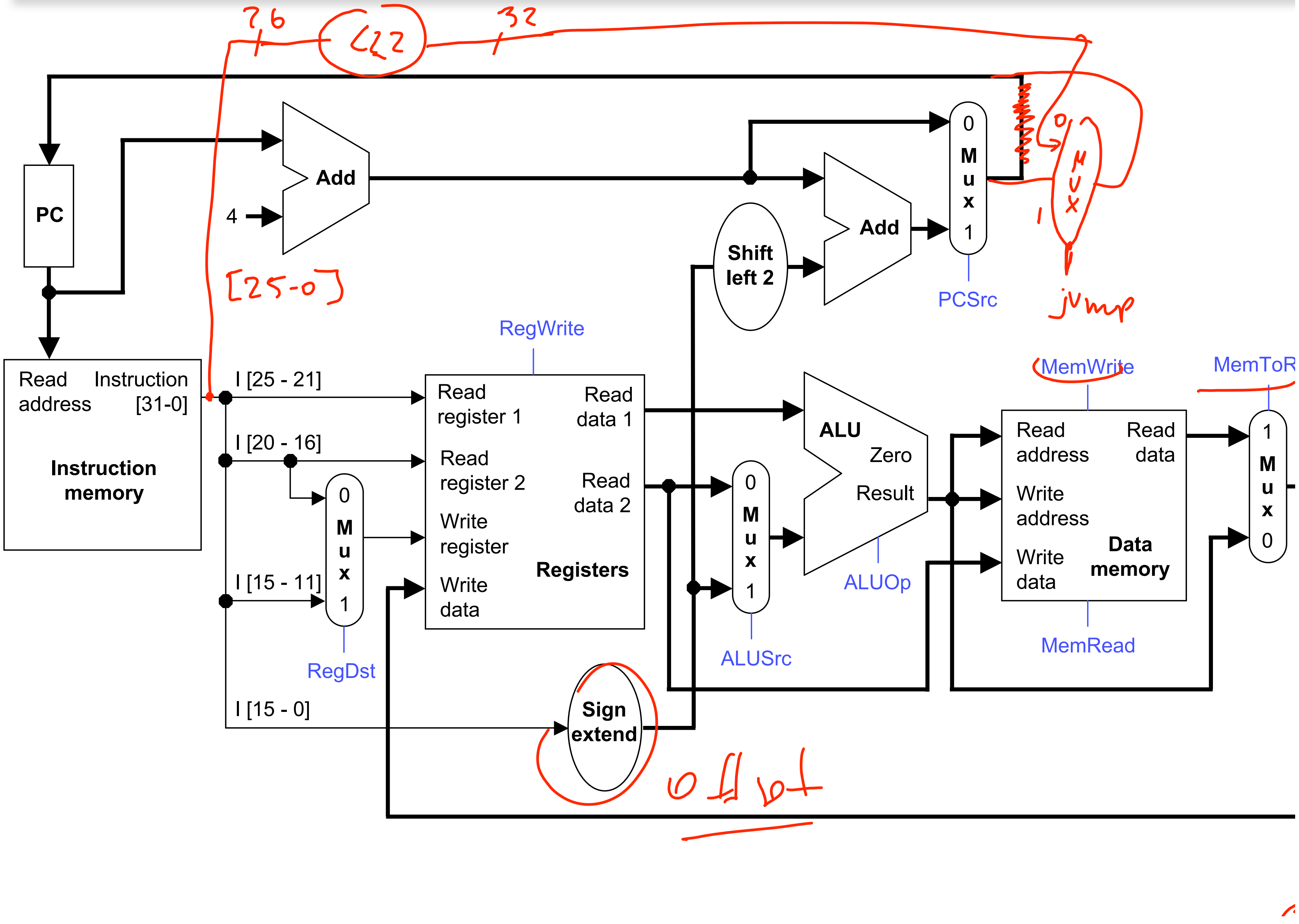
j label

op 26 bits

28 bits

The final datapath

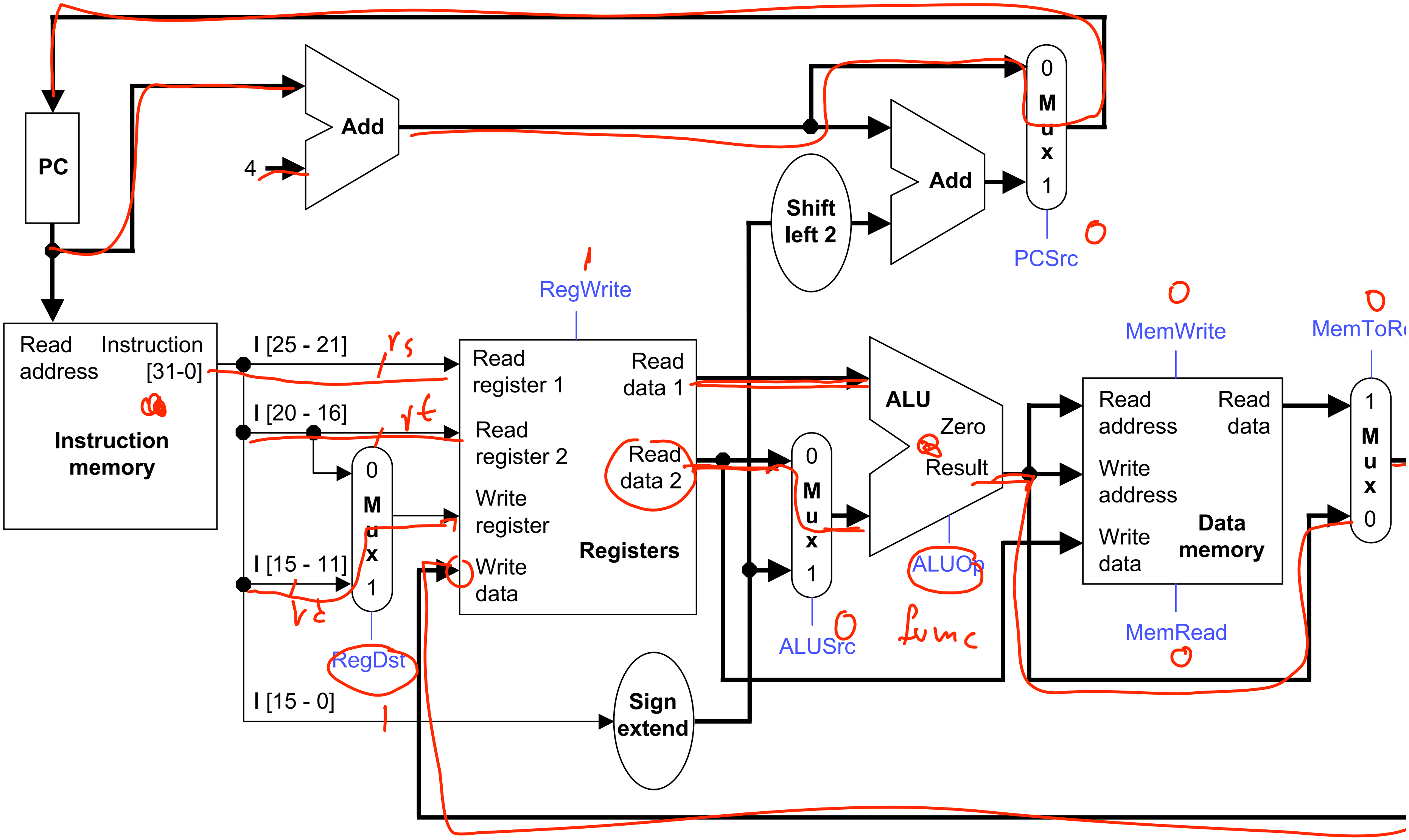


Control

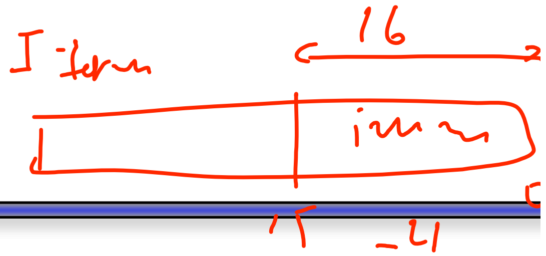
- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
    - The control unit's input is the 32-bit instruction word.
    - The outputs are values for the blue control signals in the datapath.
- Most of the signals can be generated from the 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

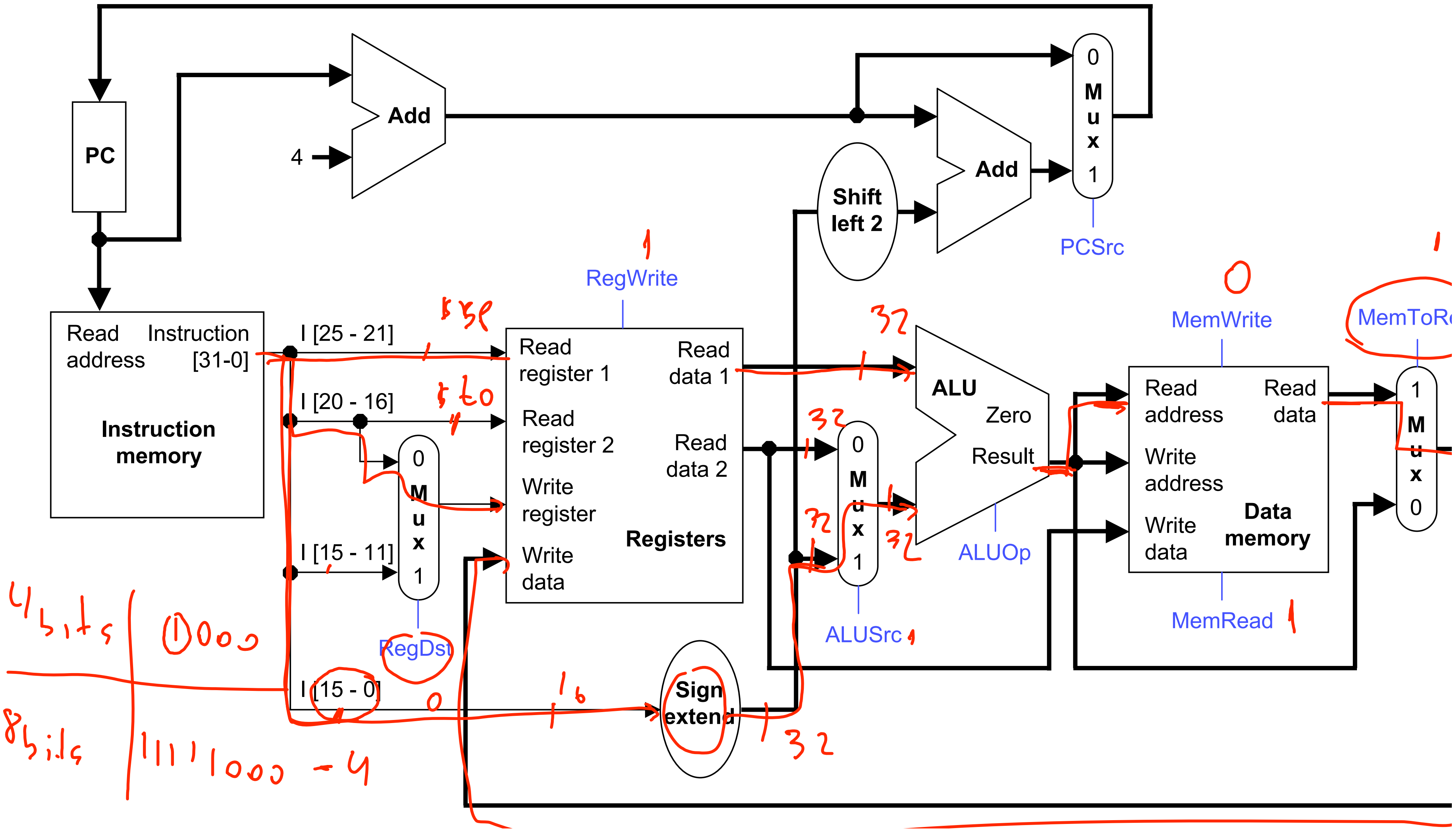
- The R-type instructions include add, sub, and, or, and slt.
- The ALUOp is determined by the instruction’s “func” field.



lw instruction path

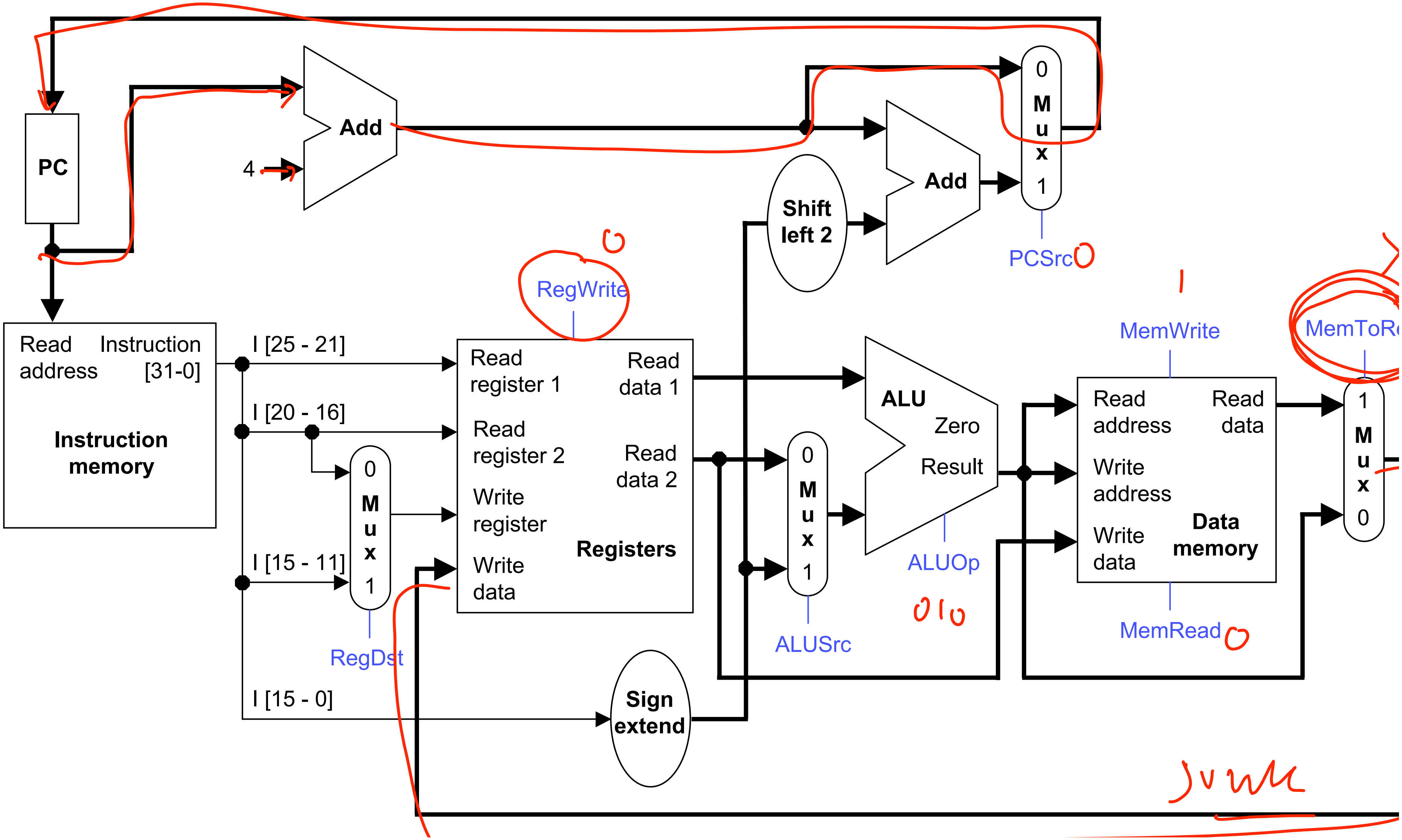


- An example load instruction is lw $t0, -4($sp).
- The ALUOp must be 010 (add), to compute the effective address.



sw instruction path

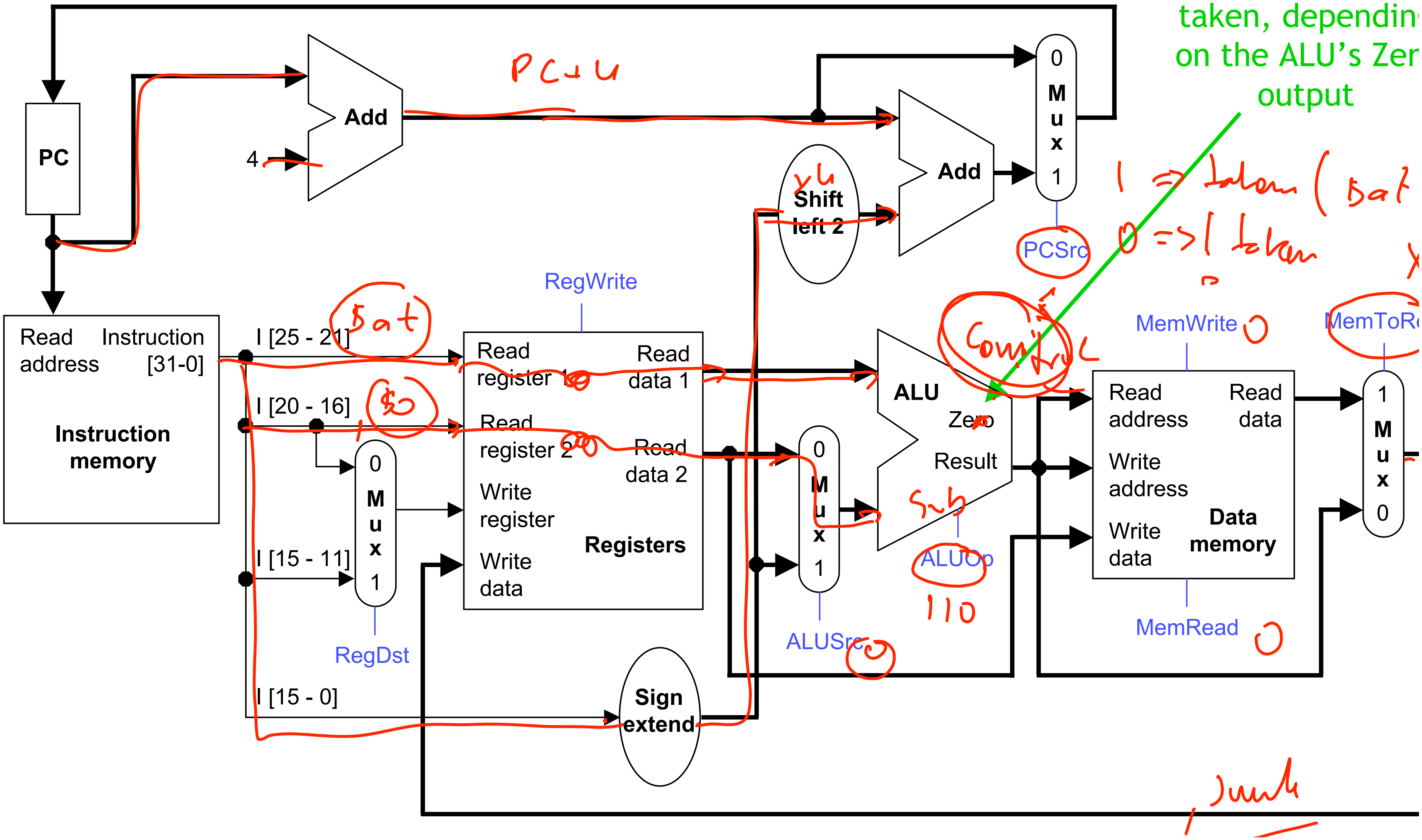
- An example store instruction is sw $a0, 16($sp).
- The ALUOp must be 010 (add), again to compute the effective address.



Op1 - OP2 = 0 ?

beq instruction path

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


The branch may or may not be taken, dependin on the ALU's Zer output

Control signal table

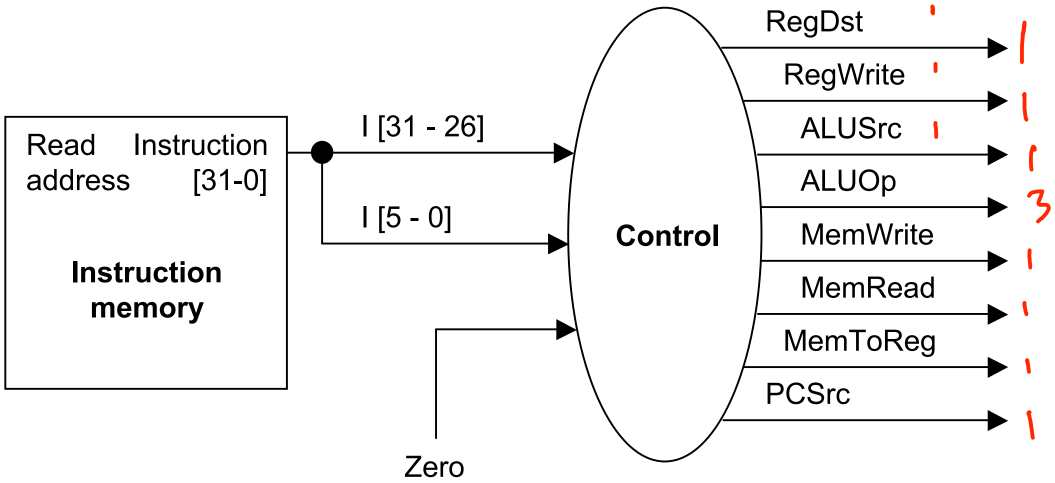
Operation	RegDst	RegWrite	ALUSrc	ALUOp func	MemWrite	MemRead	MemToRe
add ✓	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1 imm	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- ✓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 be *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. 6
  - Six bits come from the instruction's func field. 6
  - It also needs the Zero output of the ALU. 1

    13
- 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.
- The textbook presents a slightly different control unit.



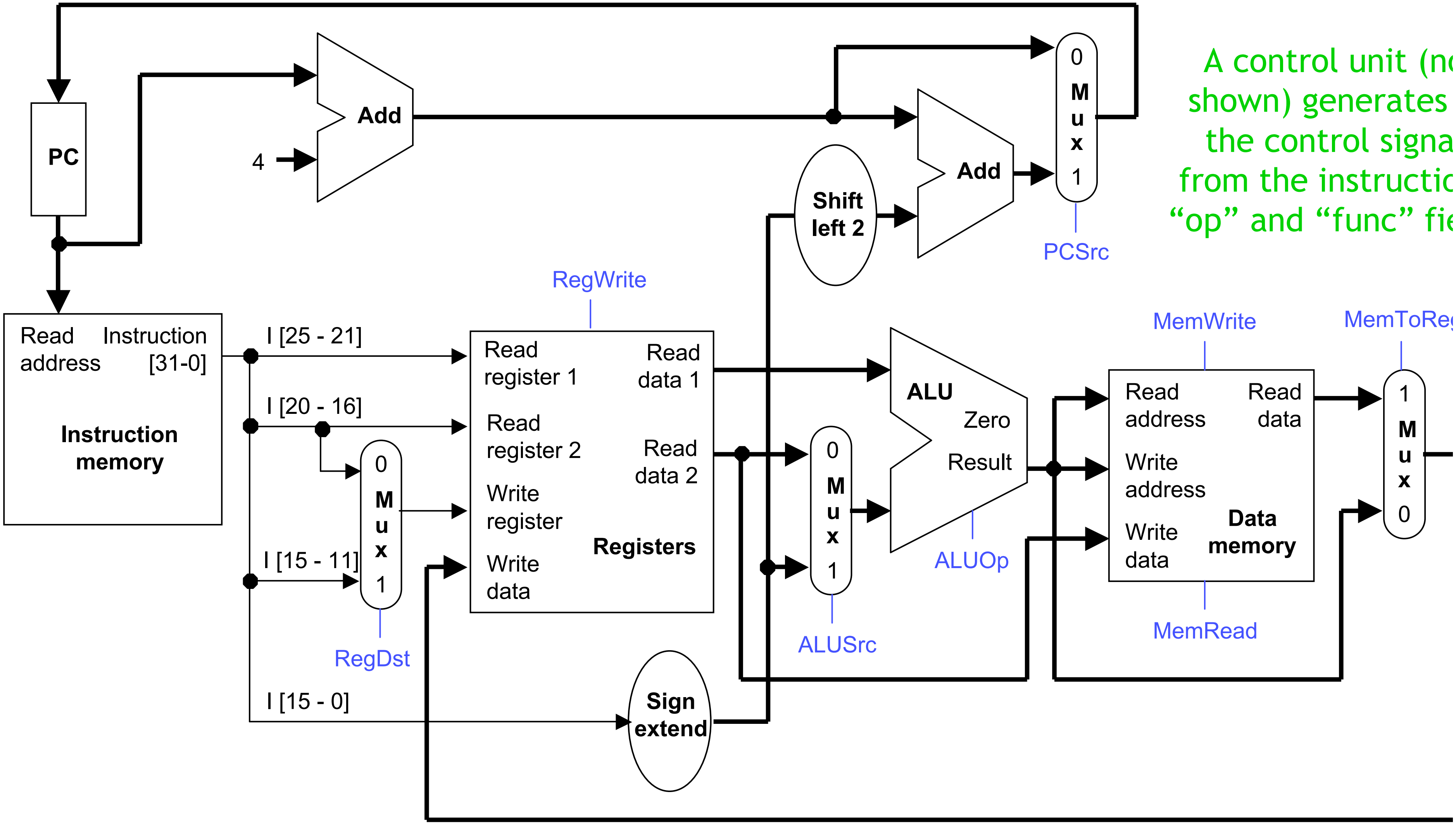
Summary - Single Cycle Datapath

- 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.
- Now we'll see the performance limitations of this single-cycle machine and try to improve upon it.

Multicycle datapath

- We just saw a single-cycle datapath and control unit for our simple MIPS-based instruction set.
- A multicycle processor fixes some shortcomings in the single-cycle CPU.
  - Faster instructions are not held back by slower ones.
  - The clock cycle time can be decreased.
  - We don’t have to duplicate any hardware units.
- A multicycle processor requires a somewhat simpler datapath which we’l see today, but a more complex control unit that we’ll see later.

The single-cycle design again…



A control unit (nc
shown) generates
the control signa
from the instructic
“op” and “func” fie

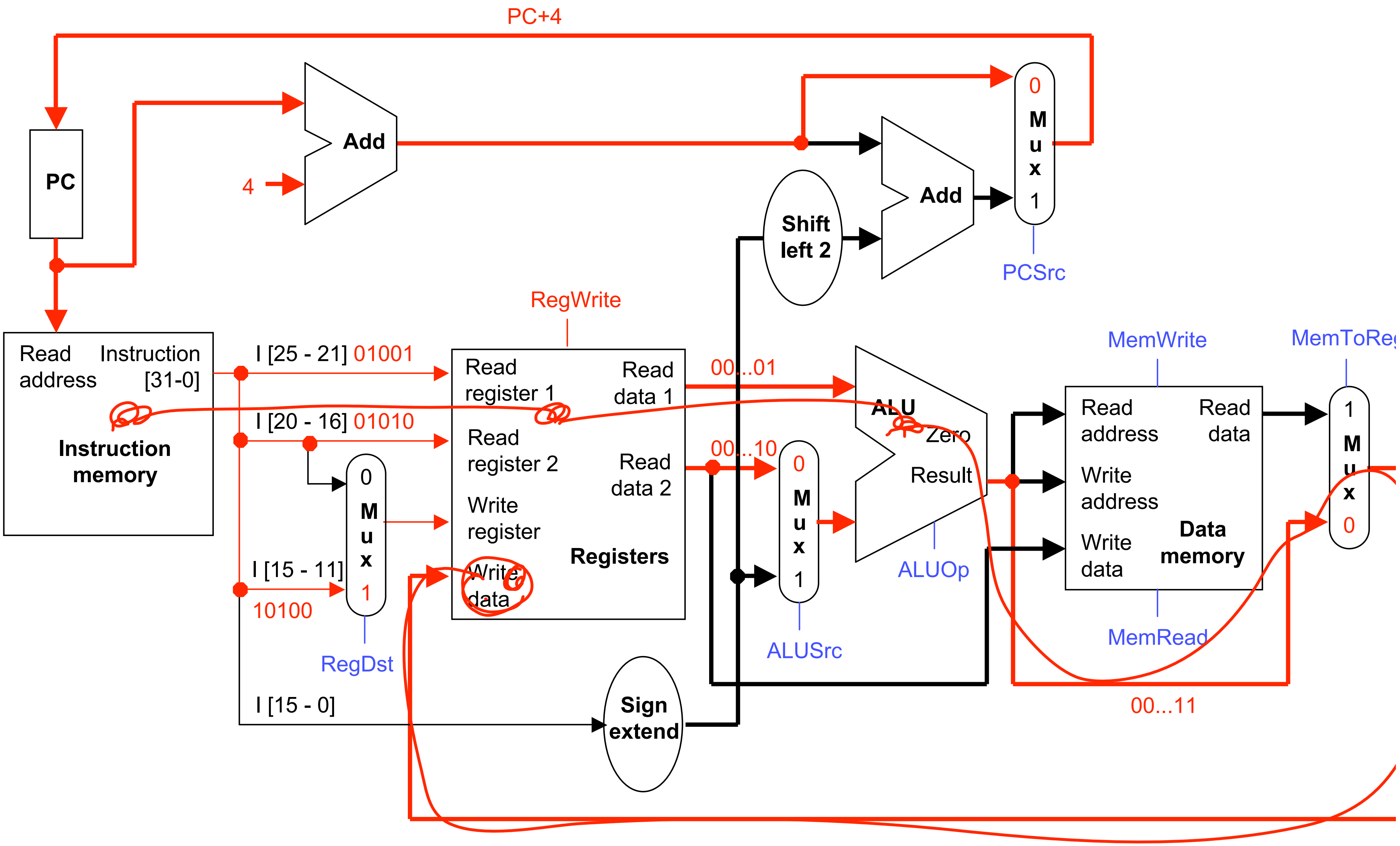
The example add from last time

- Consider the instruction add $s4, $t1, $t2.

000000	01001	01010	10100	00000	100000
op	rs	rt	rd	shamt	func

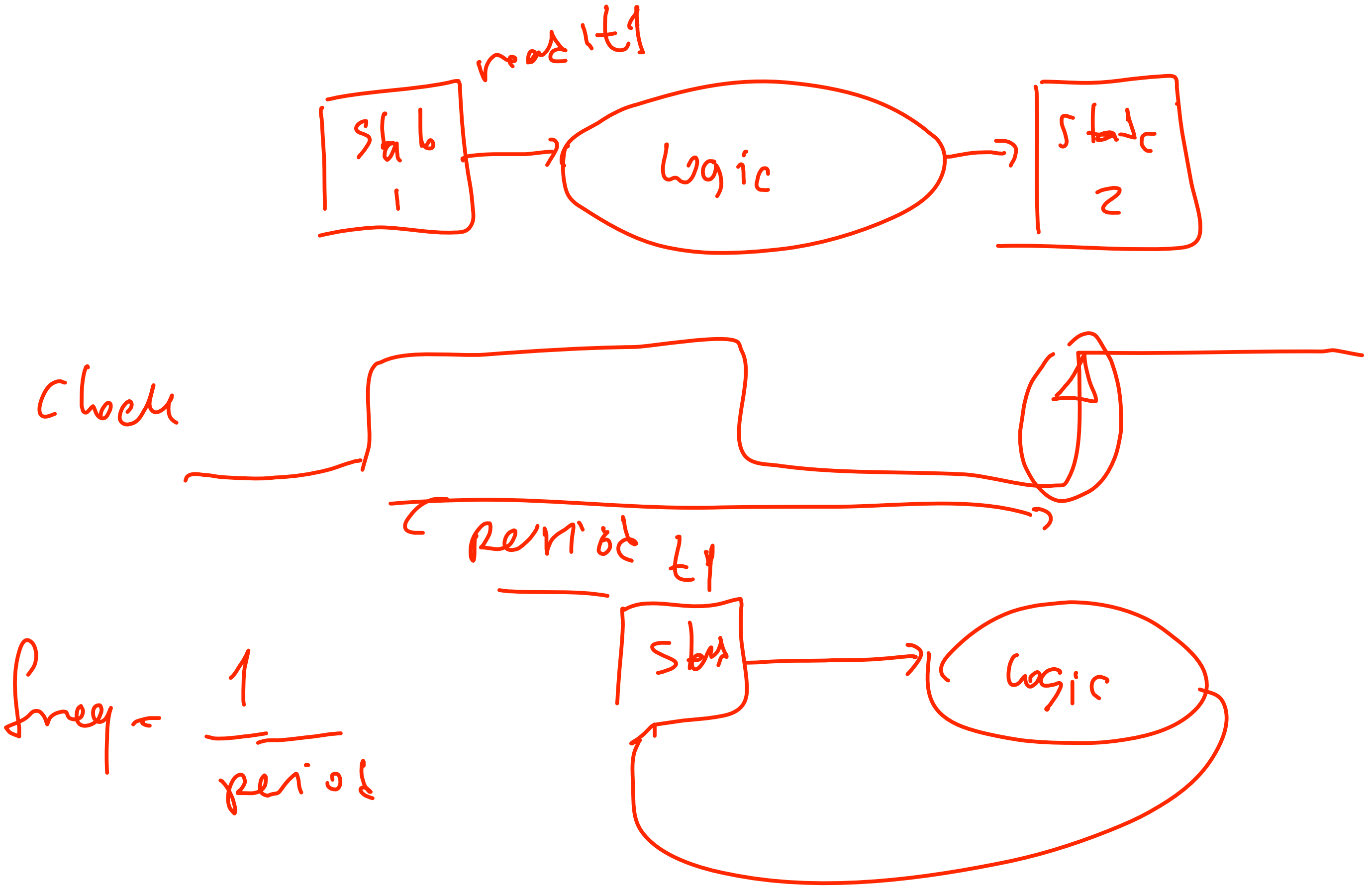
- Assume $t1 and $t2 initially contain 1 and 2 respectively.
- Executing this instruction involves several steps.
  1. The instruction word is read from the instruction memory, and the program counter is incremented by 4.
  2. The sources $t1 and $t2 are read from the register file.
  3. The values 1 and 2 are added by the ALU.
  4. The result (3) is stored back into $s4 in the register file.

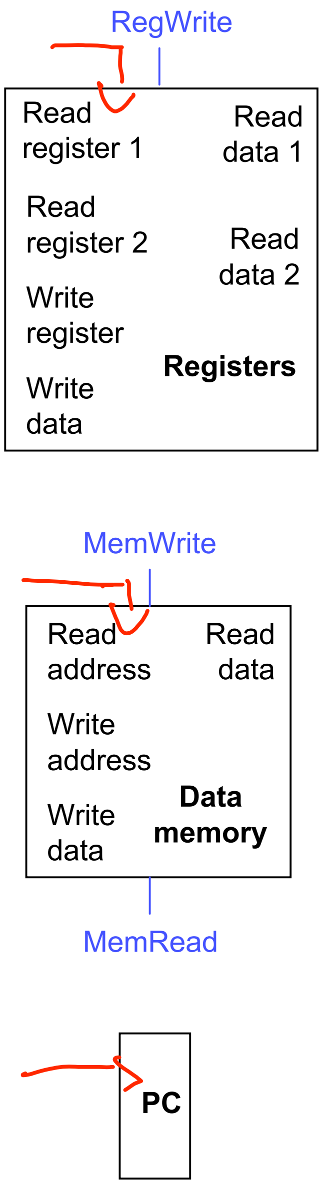
How the add goes through the datapath



State elements

- In an instruction like add $t1, $t1, $t2, how do we know $t1 is not updated until *after* its original value is read?





The datapath and the clock

STEP 1: A new instruction is loaded from memory. The control unit sets the datapath signals appropriately so that

- registers are read, ✓
- ALU output is generated, ✓
- data memory is read and ✓
- branch target addresses are computed. ✓

STEP 2:

- The register file is updated for arithmetic or lw instructions.
- Data memory is written for a sw instruction.
- The PC is updated to point to the next instruction.

- In a single-cycle datapath everything in Step 1 must complete within one clock cycle.

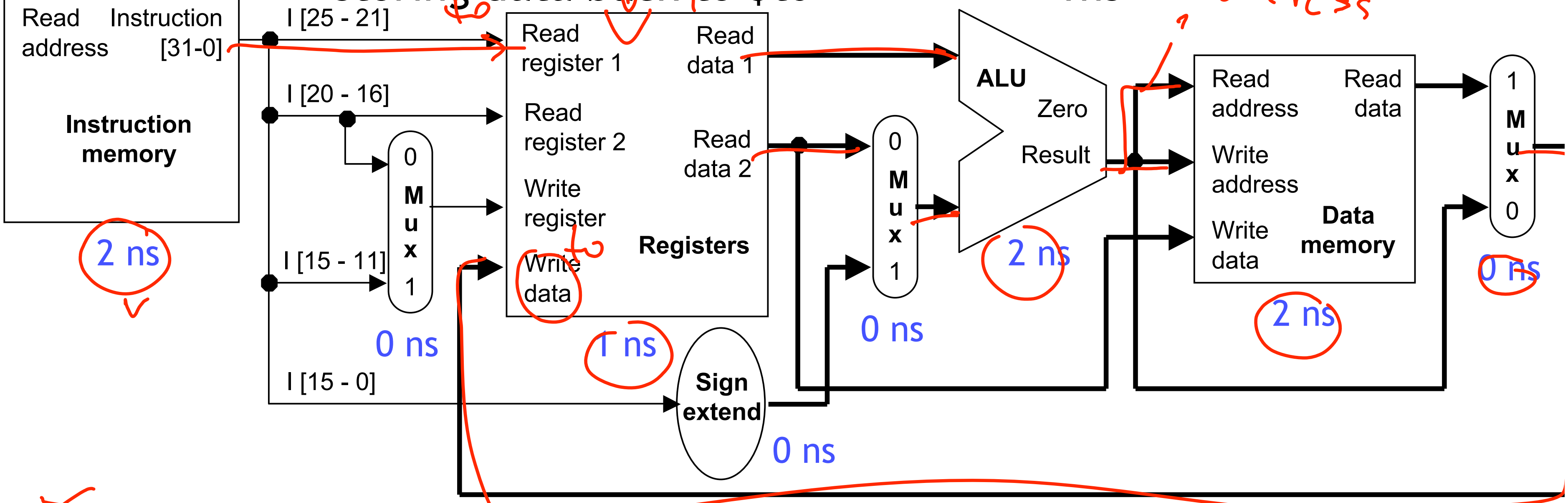
The slowest instruction...

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, lw $t0, –4($sp) needs 8ns, assuming the delays shown here.

reading the instruction memory	2ns
reading the base register $sp	1ns
computing memory address $sp-4 2ns	
reading the data memory	2ns
storing data back to $t0	1ns

8ns

effective address



...determines the clock cycle time

- If we make the cycle time 8ns then *every* instruction will take 8ns, even if they don’t need that much time.
- For example, the instruction add $s4, $t1, $t2 really needs just 6ns.

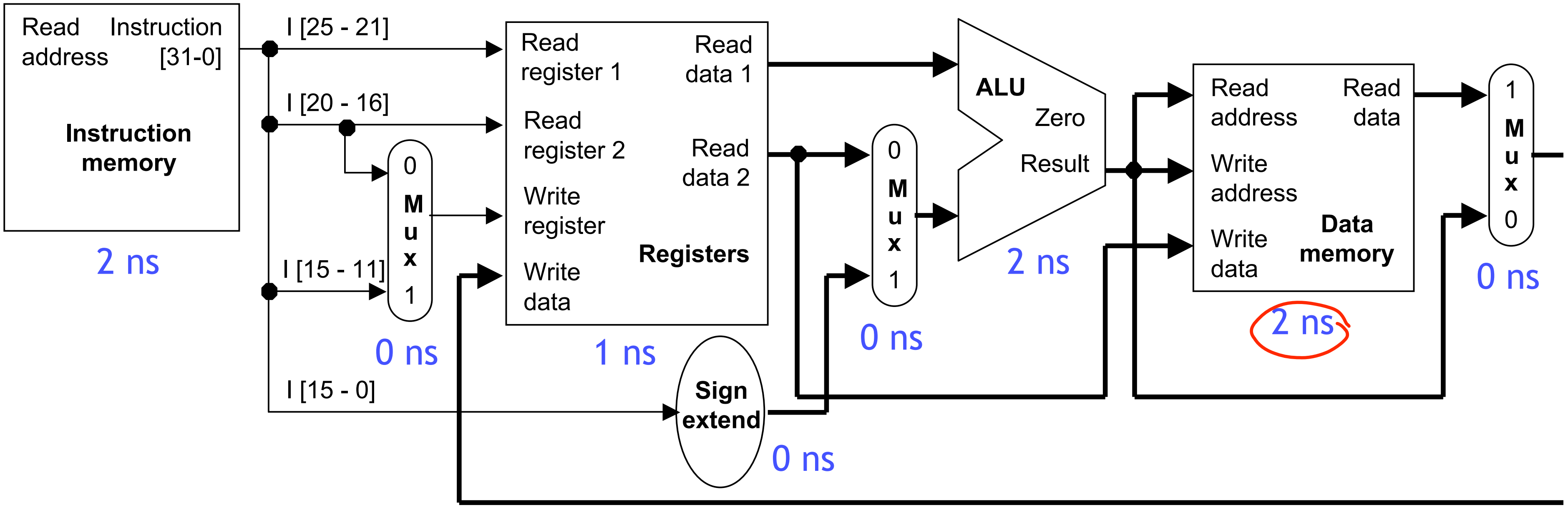
reading the instruction memory	2ns
reading registers $t1 and $t2	1ns
computing $t1 + $t2	2ns
storing the result into $s0	1ns

6ns

add = 6ns

sf = 7ns

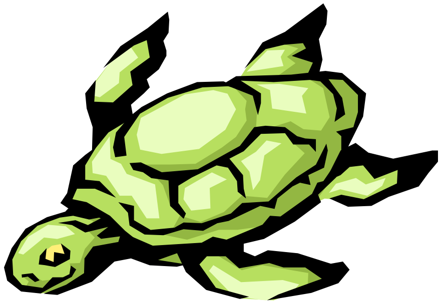
beq = 5ns



How bad is this?

- With these same component delays, a sw instruction would need 7ns, and beq would need just 5ns.
- Let's consider the **`gcc`** instruction mix from ~~p. 189~~ of the textbook.

Instruction	Frequency
Arithmetic	48% ✓
Loads	22%
Stores	11%
Branches	19%



- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be:

$$(48\% \times 6\text{ns}) + (22\% \times 8\text{ns}) + (11\% \times 7\text{ns}) + (19\% \times 5\text{ns}) = 6.36\text{ns}$$

$\frac{8}{6.36} \approx$

- The single-cycle datapath is about 1.26 times slower!

It gets worse...

- We've made <u>very</u> optimistic assumptions about memory latency:
  - Main memory accesses on modern machines is >50ns.
    - For comparison, an ALU on the Pentium4 takes ~0.3ns.
- Our worst case cycle (loads/stores) includes 2 memory accesses
  - A modern single cycle implementation would be stuck at <10Mhz.
  - Caches will improve common case access time, not worst case.
- Tying frequency to worst case path violates first law of performance!!

$< \frac{1}{100ns} = \frac{10}{}$

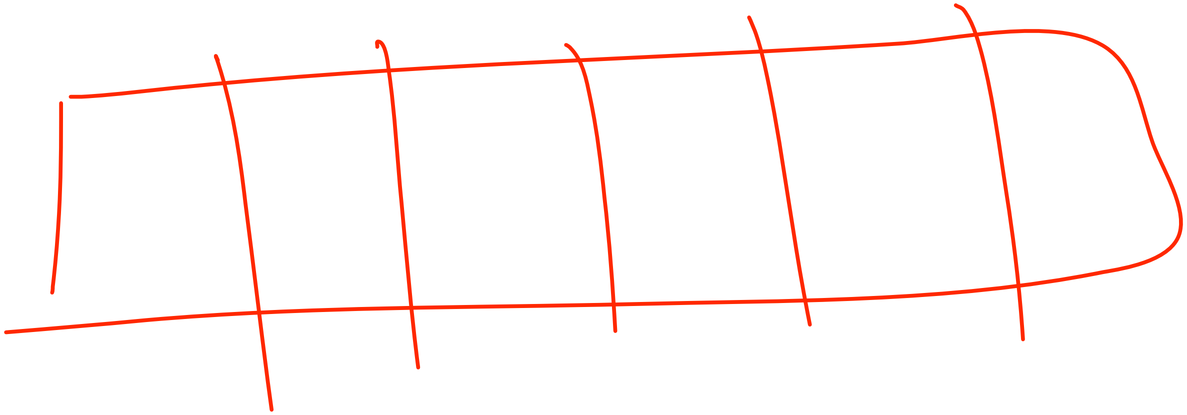
3G

A multistage approach to instruction execution

- We've informally described instructions as executing in several steps.

  1. Instruction fetch and PC increment. ✓
  2. Reading sources from the register file. ✓
  3. Performing an ALU computation. ✓
  4. Reading or writing (data) memory. ✓
  5. Storing data back to the register file. ✓

- What if we made these stages *explicit* in the hardware design?



Performance benefits

- Each instruction can execute only the stages that are necessary.
  - Arithmetic 1 2 3 5
  - Load 1 2 3 4 5
  - Store 1 2 3 4
  - Branches 1 2 3
- This would mean that instructions complete as soon as possible, instead of being limited by the slowest instruction.

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

The  clock  cycle

- Things are simpler if we assume that each “stage” takes one clock cycle.
  - This means instructions will require multiple clock cycles to execute.
  - But since a single stage is fairly simple, the cycle time can be low.
- For the proposed execution stages below and the sample datapath delays shown earlier, each stage needs 2ns at most.
  - This accounts for the slowest devices, the ALU and data memory.
  - A 2ns clock cycle time corresponds to a 500MHz clock rate!

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

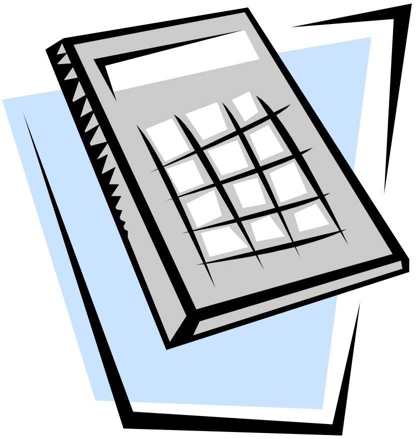
Cost benefits

- As an added bonus, we can eliminate some of the extra hardware from the single-cycle datapath.
  - We will restrict ourselves to using each functional unit once per cycle just like before.
  - But since instructions require multiple cycles, we could reuse some units in a *different* cycle during the execution of a single instruction.
- For example, we could use the same ALU:
  - to increment the PC (first clock cycle), and
  - for arithmetic operations (third clock cycle).

Proposed execution stages

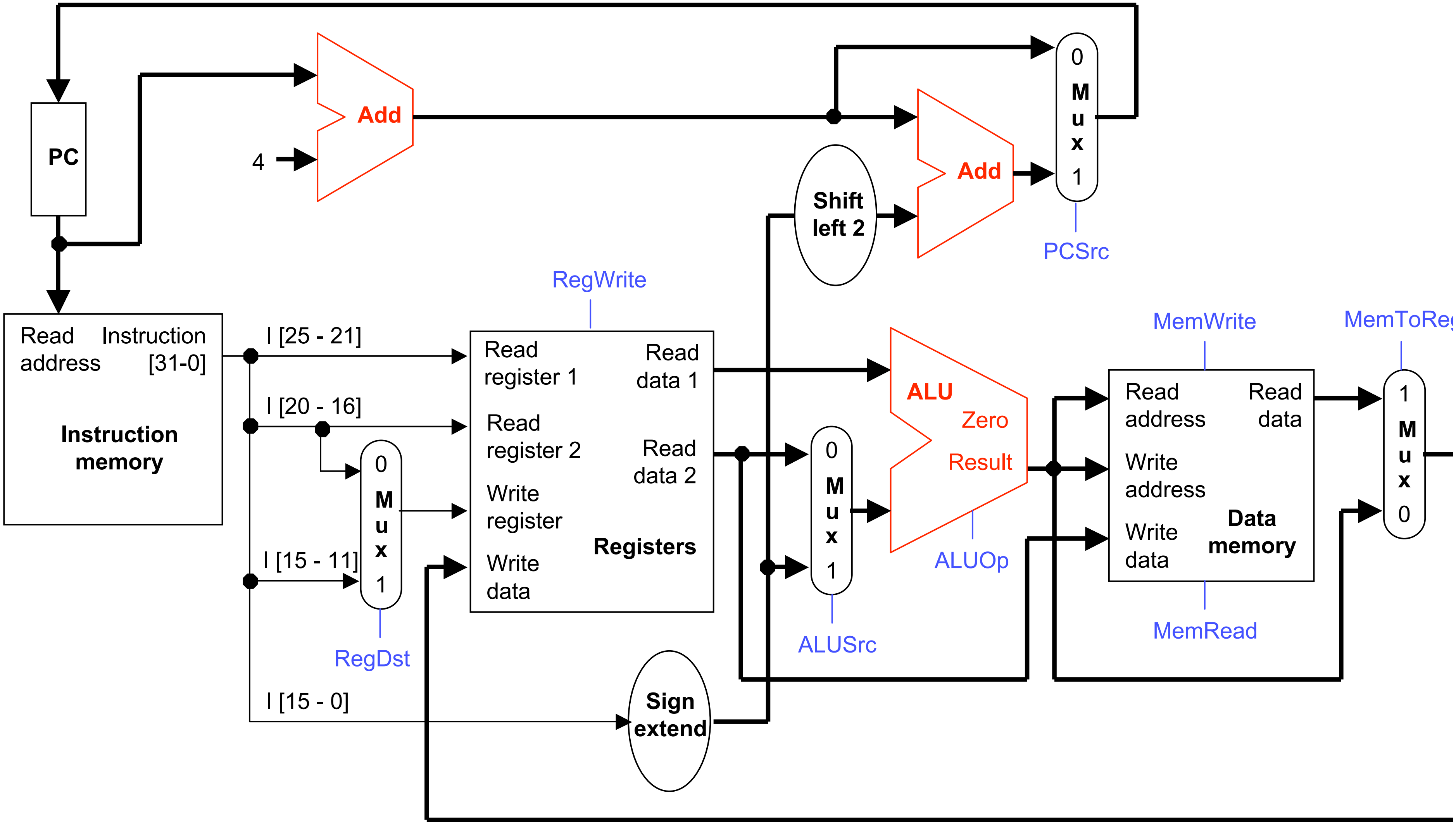
1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

Two extra adders



- Our original single-cycle datapath had an ALU and two adders.
- The arithmetic-logic unit had two responsibilities.
  - Doing an operation on two registers for arithmetic instructions.
  - Adding a register to a sign-extended constant, to compute effective addresses for lw and sw instructions.
- One of the extra adders incremented the PC by computing PC + 4.
- The other adder computed branch targets, by adding a sign-extended, shifted offset to (PC + 4).

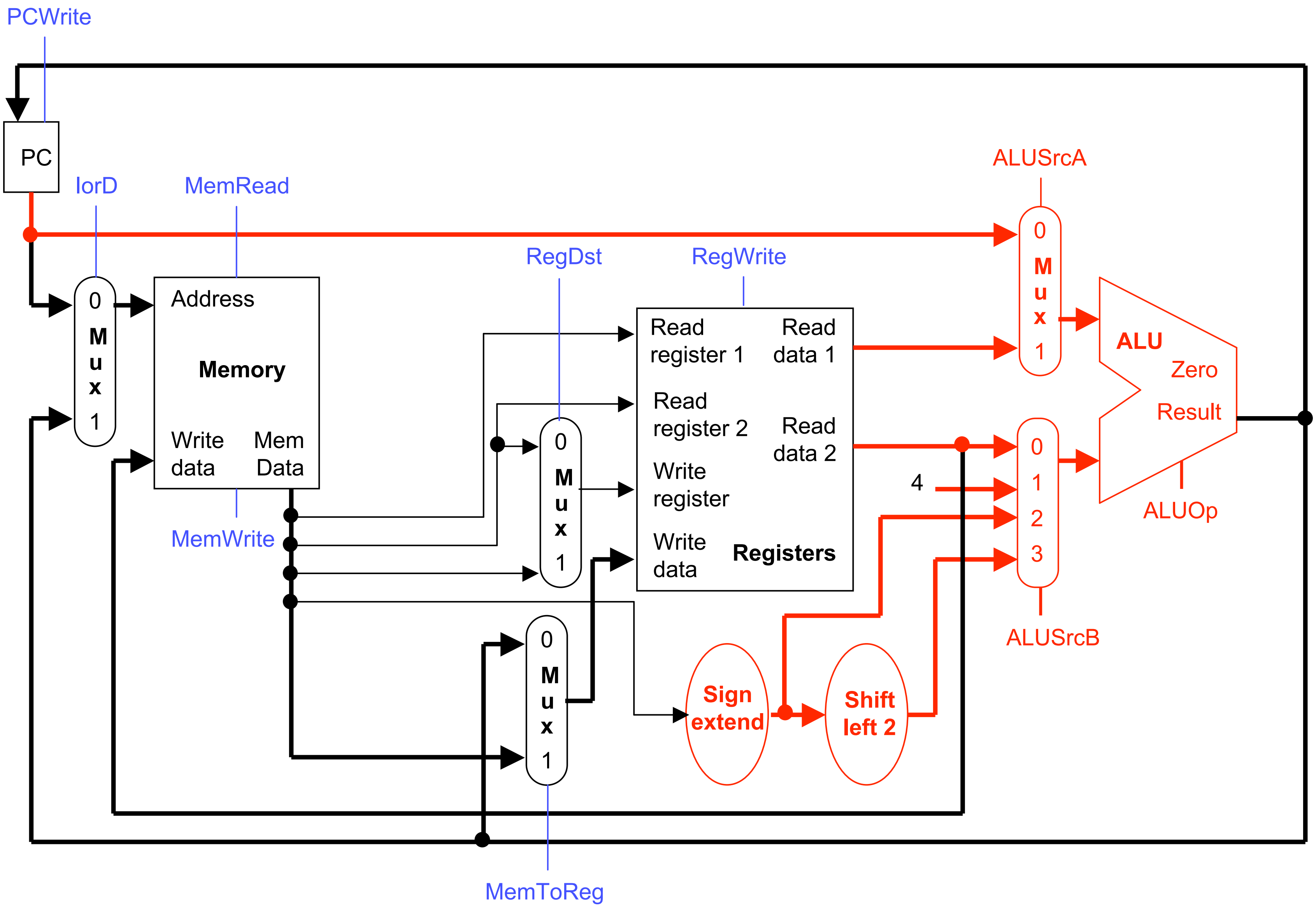
The extra single-cycle adders



Our new adder setup

- We can eliminate *both* extra adders in a multicycle datapath, and instea use just one ALU, with multiplexers to select the proper inputs.
- A 2-to-1 mux ALUSrcA sets the first ALU input to be the PC or a register.
- A 4-to-1 mux ALUSrcB selects the second ALU input from among:
  - the register file (for arithmetic operations),
  - a constant 4 (to increment the PC),
  - a sign-extended constant (for effective addresses), and
  - a sign-extended and shifted constant (for branch targets).
- This permits a single ALU to perform all of the necessary functions.
  - Arithmetic operations on two register operands.
  - Incrementing the PC.
  - Computing effective addresses for lw and sw.
  - Adding a sign-extended, shifted offset to (PC + 4) for branches.

The multicycle adder setup highlighted



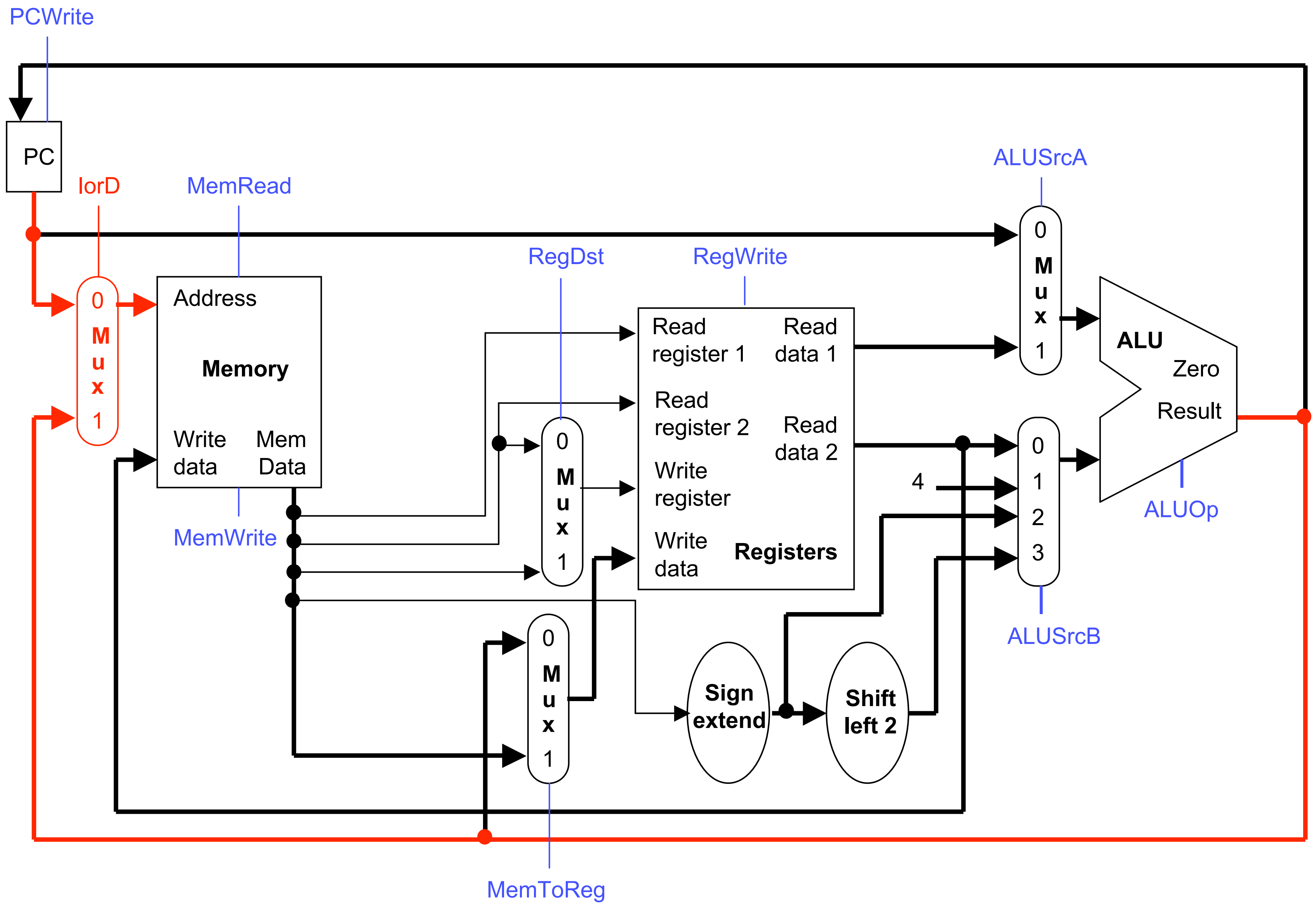
Eliminating a memory

- Similarly, we can get by with one unified memory, which will store *both* program instructions *and* data. (a Princeton architecture)
- This memory is used in both the instruction fetch and data access stages, and the address could come from either:
  - the PC register (when we're fetching an instruction), or
  - the ALU output (for the effective address of a lw or sw).
- We add another 2-to-1 mux, IorD, to decide whether the memory is being accessed for instructions or for data.

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

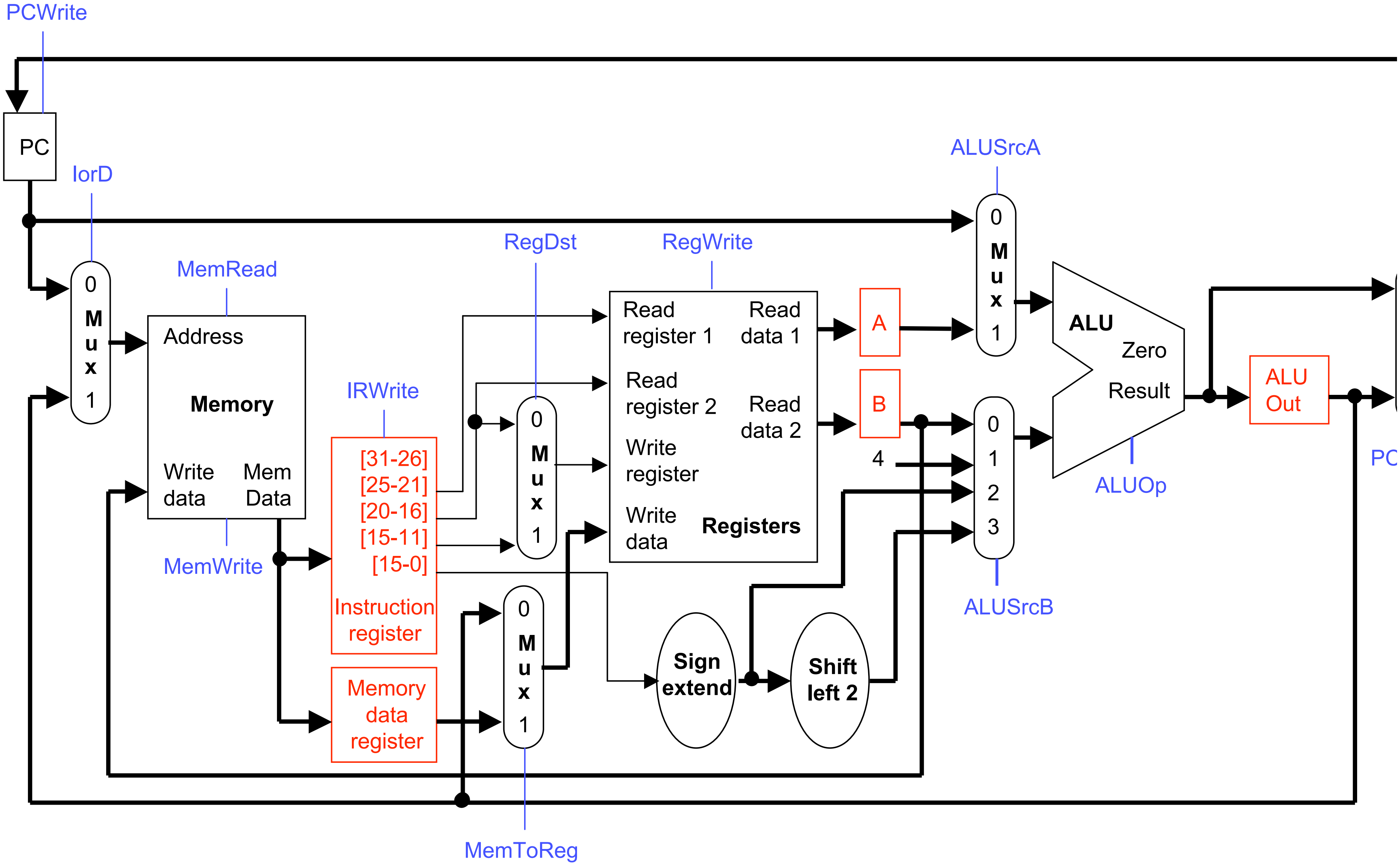
The new memory setup highlighted



Intermediate registers

- Sometimes we need the output of a functional unit in a later clock cycle during the execution of one instruction.
  - The instruction word fetched in stage 1 determines the destination of the register write in stage 5.
  - The ALU result for an address computation in stage 3 is needed as the memory address for lw or sw in stage 4.
- These outputs will have to be stored in intermediate registers for future use. Otherwise they would probably be lost by the next clock cycle.
  - The instruction read in stage 1 is saved in Instruction register.
  - Register file outputs from stage 2 are saved in registers A and B.
  - The ALU output will be stored in a register ALUOut.
  - Any data fetched from memory in stage 4 is kept in the Memory data register, also called MDR.

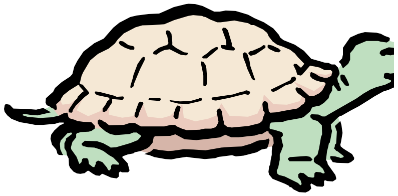
The final multicycle datapath



Register write control signals

- We have to add a few more control signals to the datapath.
- Since instructions now take a variable number of cycles to execute, we cannot update the PC on each cycle.
  - Instead, a PCWrite signal controls the loading of the PC.
  - The instruction register also has a write signal, IRWrite. We need to keep the instruction word for the duration of its execution, and must explicitly re-load the instruction register when needed.
- The other intermediate registers, MDR, A, B and ALUOut, will store data for only one clock cycle at most, and do not need write control signals.

Summary



- A single-cycle CPU has two main disadvantages.
  - The cycle time is limited by the worst case latency.
  - It requires more hardware than necessary.
- A multicycle processor splits instruction execution into several stages.
  - Instructions only execute as many stages as required.
  - Each stage is relatively simple, so the clock cycle time is reduced.
  - Functional units can be reused on different cycles.
- We made several modifications to the single-cycle datapath.
  - The two extra adders and one memory were removed.
  - Multiplexers were inserted so the ALU and memory can be used for different purposes in different execution stages.
  - New registers are needed to store intermediate results.
- Next time, we'll look at controlling this datapath.