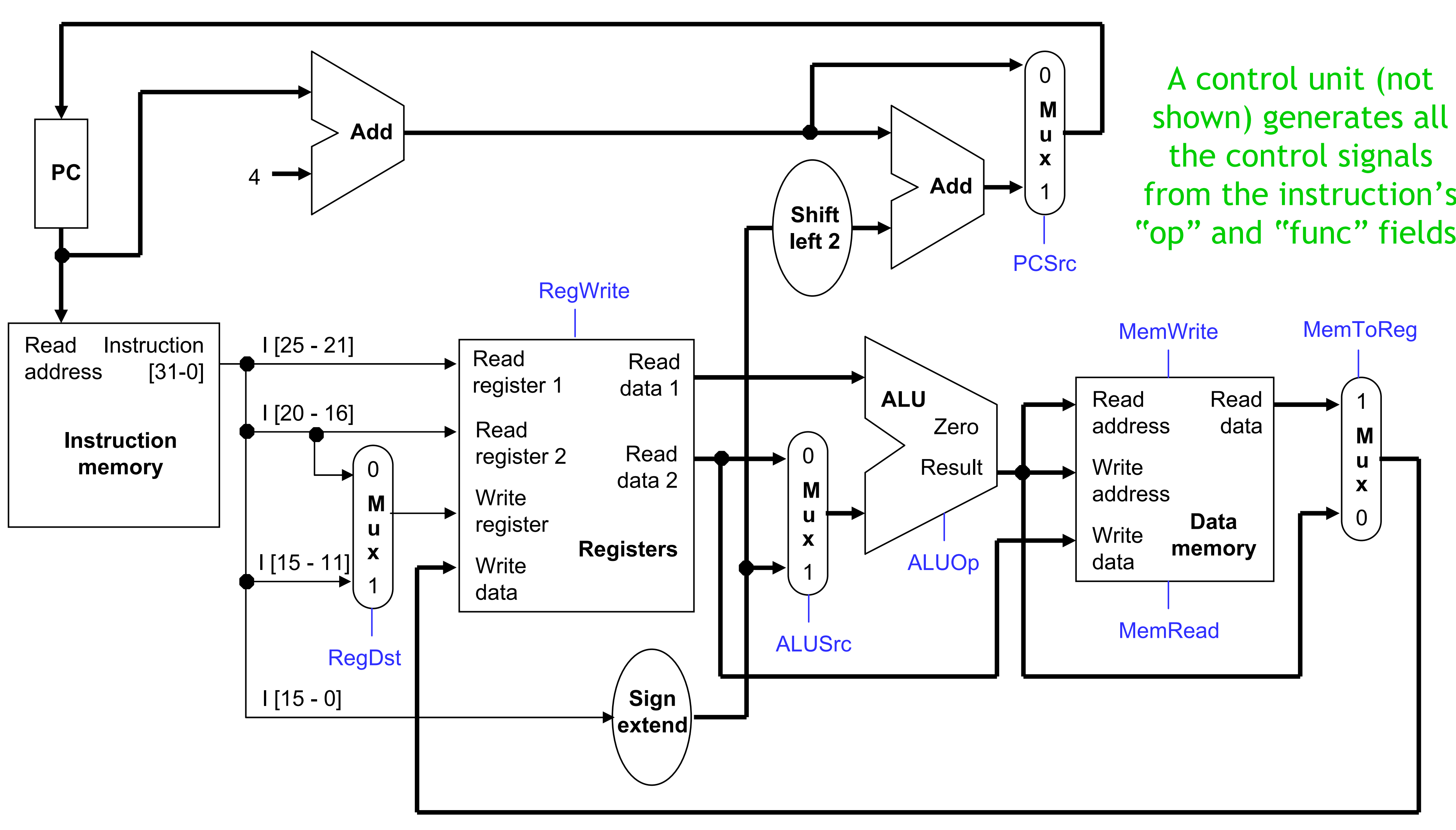
# Multicycle datapath

- Last time we 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'll see today, but a more complex control unit that we'll save for next week.


©2001-2003 Howard Huang

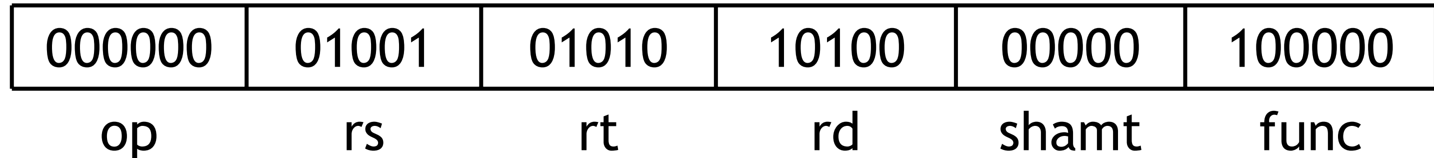
# The single-cycle design from last time




A control unit (not shown) generates all the control signals from the instruction's "op" and "func" fields.

# The example add from last time

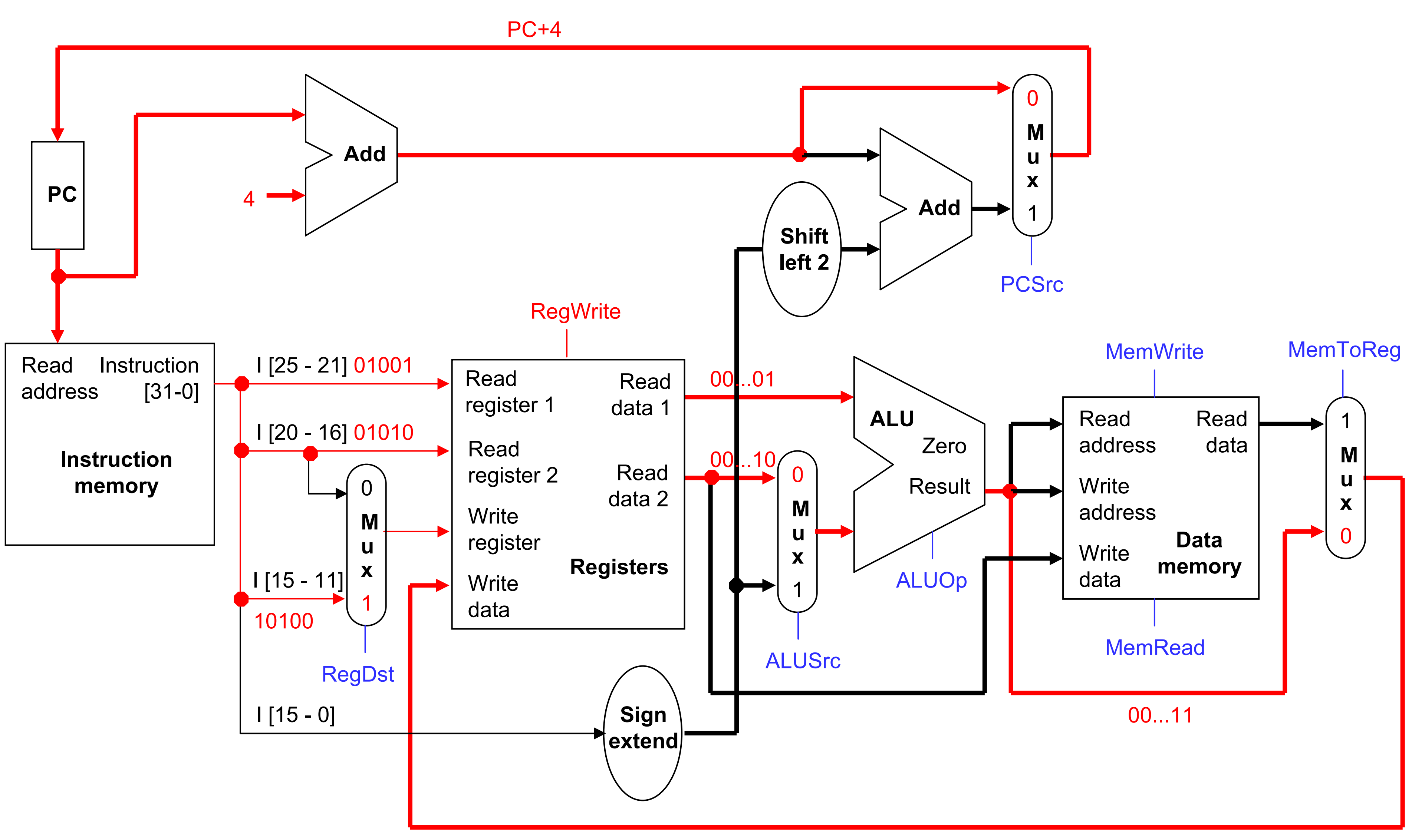
- Consider the instruction add $s4, $t1, $t2.



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

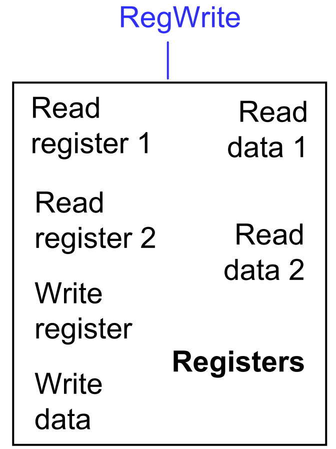
- Assume $t1 and $t2 initially contain 1 and 2 respectively.
- Executing this instruction involves several steps.
  1. The instruction word is read from the instruction memory, and the program counter is incremented by 4.
  2. The sources $t1 and $t2 are read from the register file.
  3. The values 1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

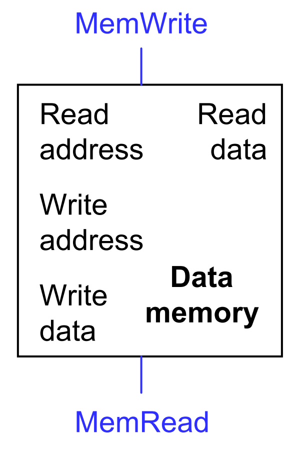
# How the add goes through the datapath



# Edge-triggered 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?
- We'll assume that our state elements are positive edge triggered, and can be updated only on the positive edge of a clock signal.
  - The register file and data memory have explicit write control signals, RegWrite and MemWrite. These units can be written to only if the control signal is asserted *and* there is a positive clock edge.
  - In a single-cycle machine the PC is updated on each clock cycle, so we don't bother to give it an explicit write control signal.



# The datapath and the clock

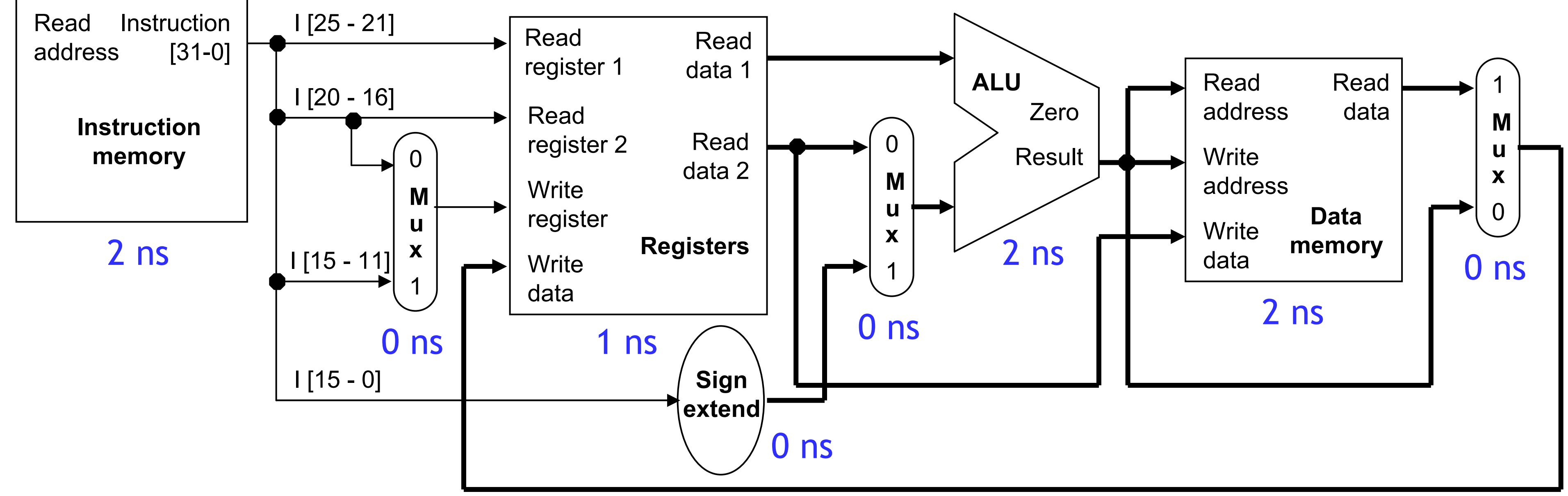
1. On a positive clock edge, the PC is updated with a new address.
2. A new instruction can then be loaded from memory. The control unit sets the datapath signals appropriately so that
   - registers are read,
   - ALU output is generated,
   - data memory is read or written, and
   - branch target addresses are computed.
3. Several things happen on the *next* positive clock edge.
   - 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 2 must complete within one clock cycle, before the next positive clock edge.

# 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.

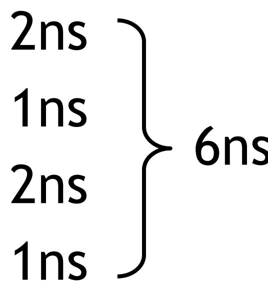
| | | |
|---|---|---|
| reading the instruction memory | 2ns | 8ns |
| reading the base register $sp | 1ns | |
| computing memory address $sp-4 | 2ns | |
| reading the data memory | 2ns | |
| storing data back to $t0 | 1ns | |

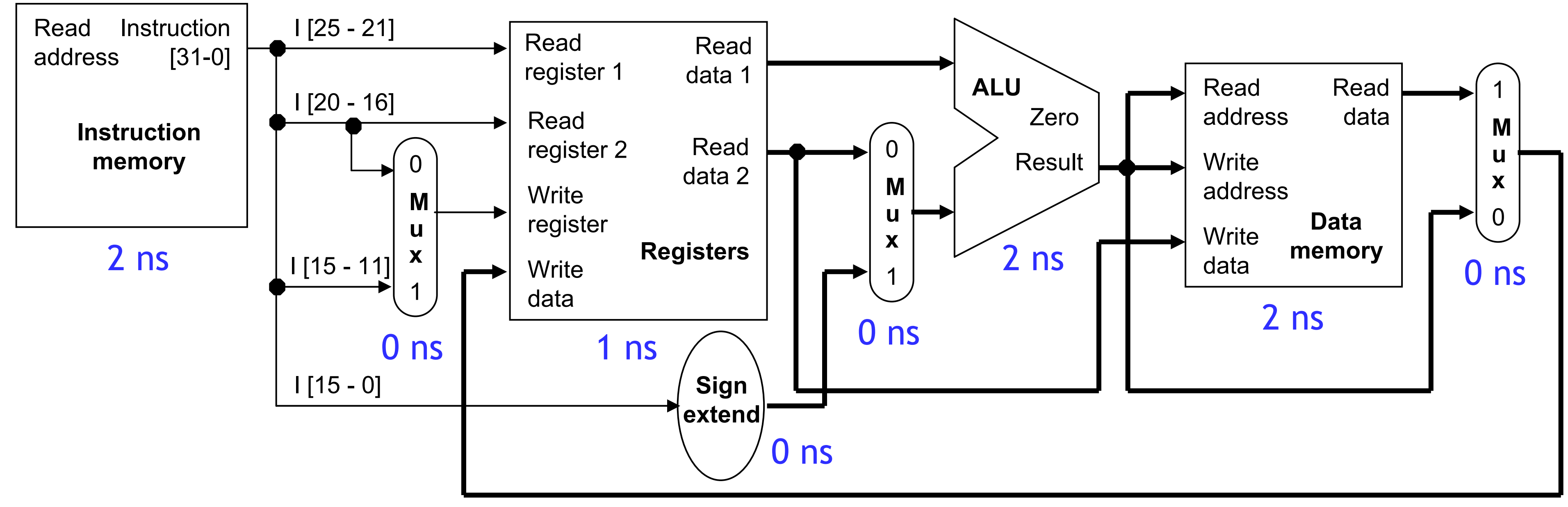


# ...determines the clock cycle time

- 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 |



# 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, which we also used in Homework 2.

| Instruction | Frequency |
|---|---|
| Arithmetic | 48% |
| Loads | 22% |
| Stores | 11% |
| Branches | 19% |

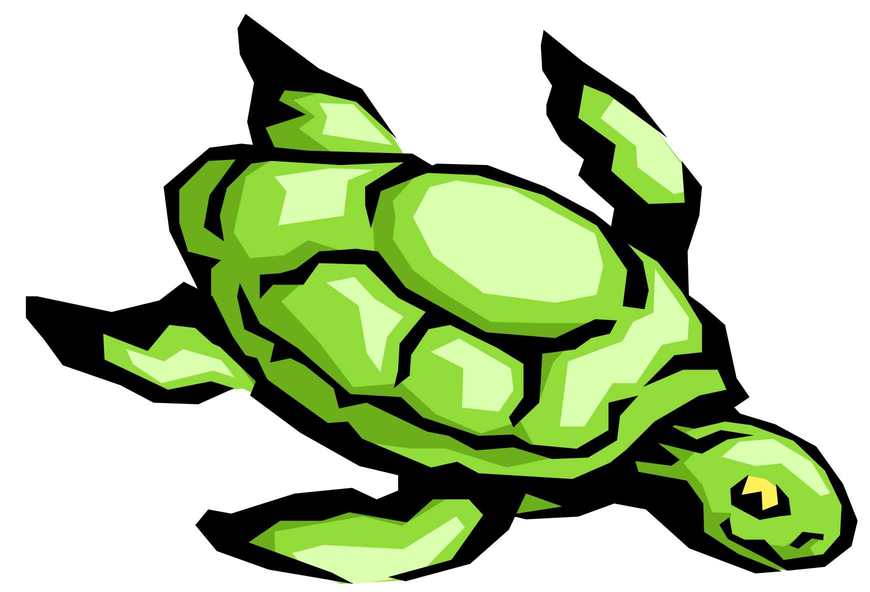
- With a single-cycle datapath, each instruction would require 8ns.
- But if we could execute instructions as fast as possible, the average time per instruction for gcc would be:

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

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

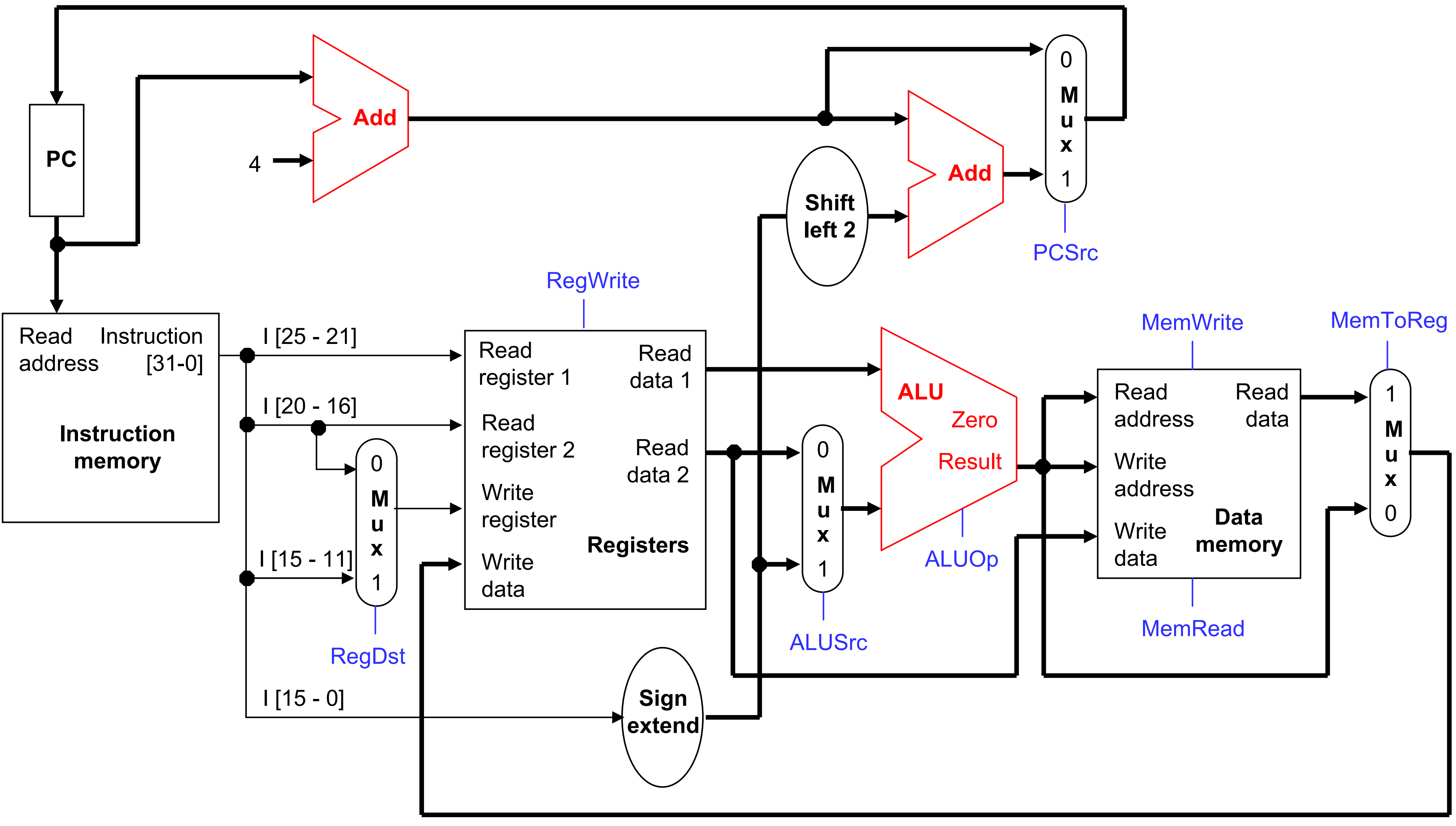
# It gets worse...

- Our small instruction set includes only very simple operations.
- If we supported more complex, time-consuming instructions, then the performance penalty of a single-cycle machine could be much lower.
  - Integer multiplication and division, or floating-point operations
  - Complex addressing modes like the 8086
  - Vector-based, SIMD instructions like MMX, SSE, 3DNow, or AltiVec



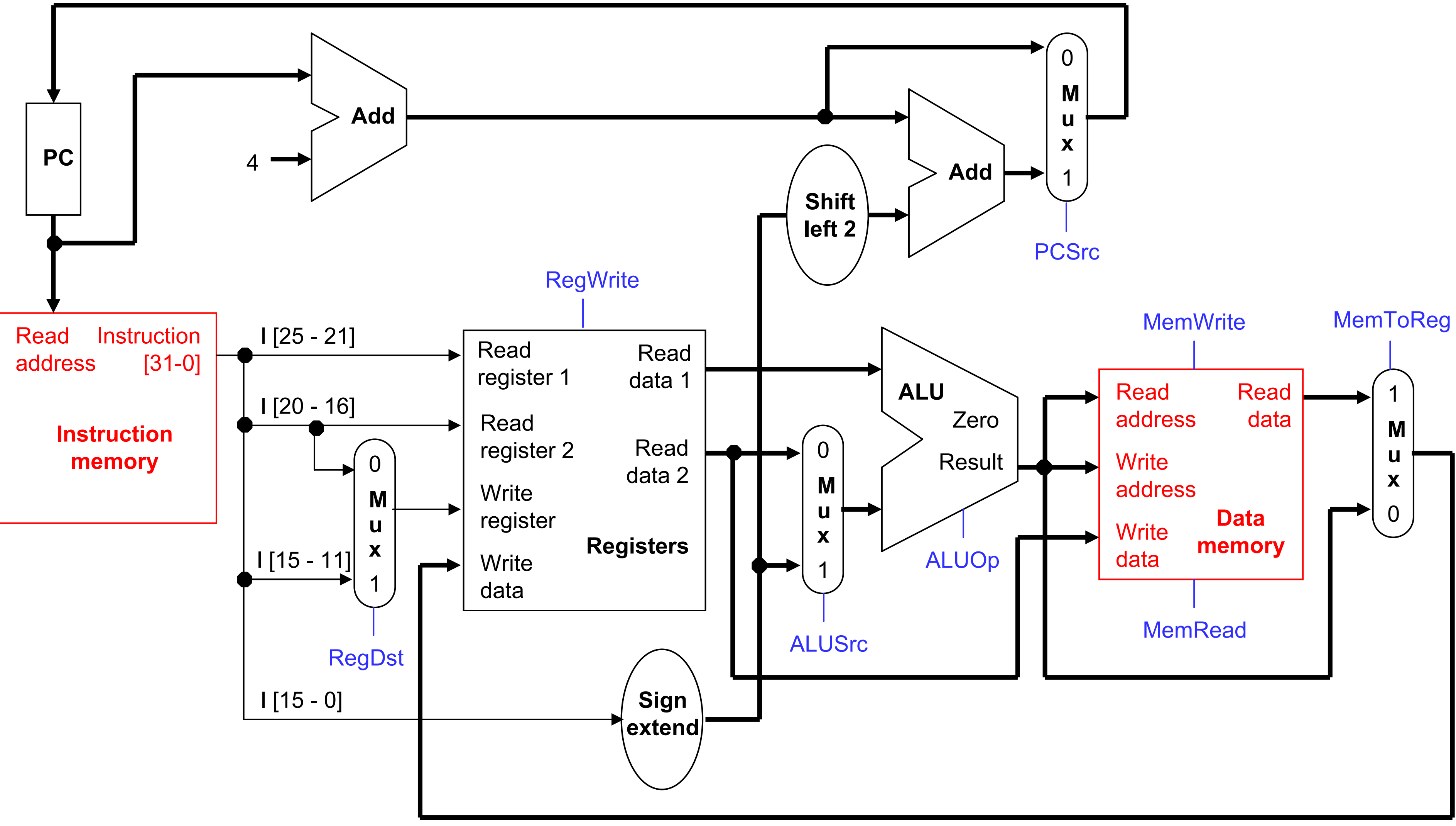
# ...and worse...

- A single-cycle datapath also uses extra hardware—one ALU is not enough, since we must do up to three calculations in one clock cycle for a beq.



# ...and worse

- This is also why we used a Harvard architecture with two memories; you can’t easily read two addresses from the same memory in one cycle.



# 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?

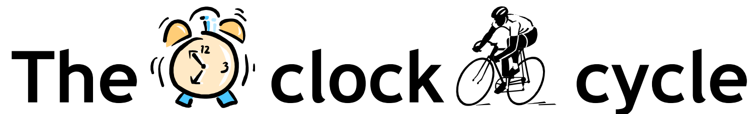
# Performance benefits

- Each instruction can execute only the stages that are necessary.
  - Arithmetic operations never read or write data memory.
  - A sw instruction does not save anything to the register file.
  - Branches neither access memory nor write to the registers.
- 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

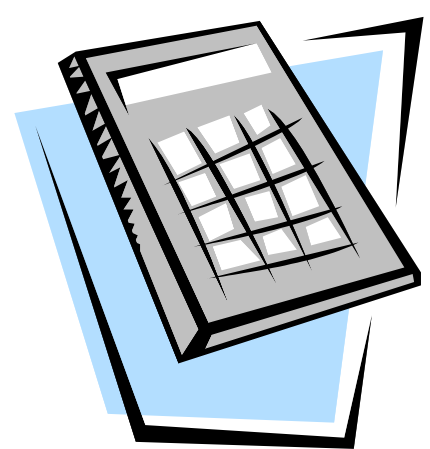
# Cost benefits

- As an added bonus, we can eliminate some of the extra hardware from the single-cycle datapath.
  - We are still restricted 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 an ALU to increment the PC in the first clock cycle of an instruction execution, and then reuse that ALU for arithmetic operations in the third cycle.

Proposed execution stages

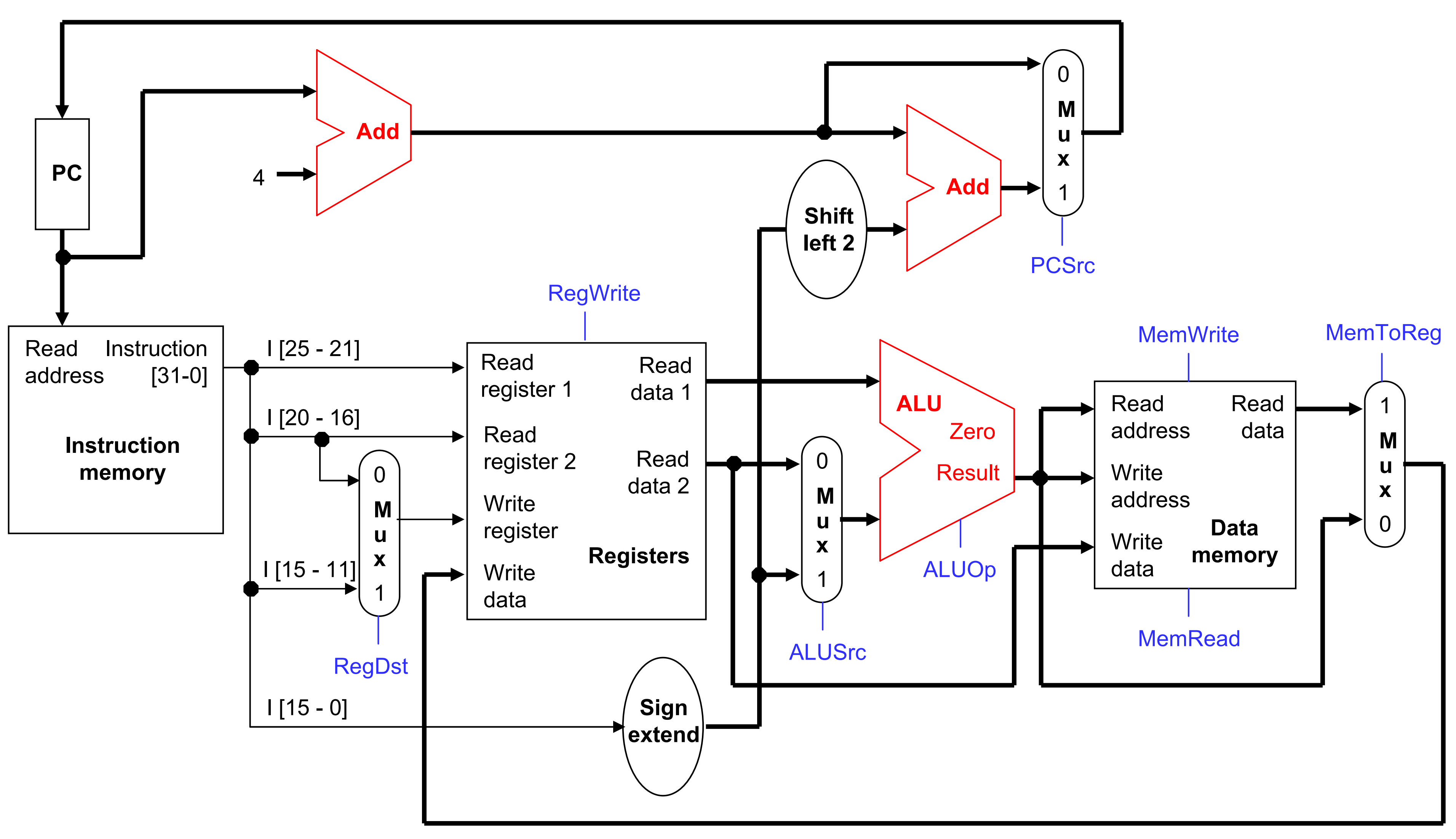
1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

# 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).

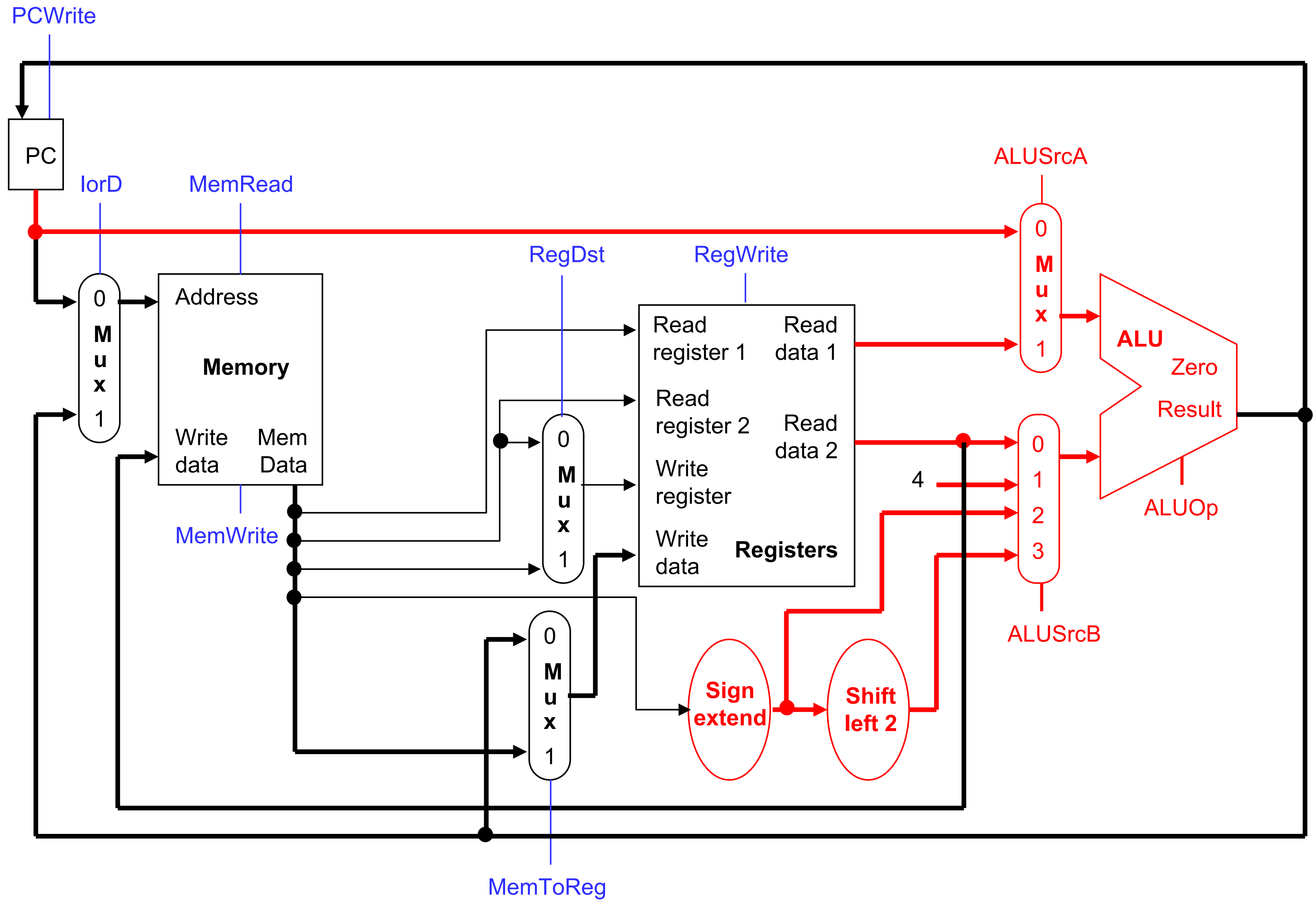
# The extra single-cycle adders



# Our new adder setup

- We can eliminate *both* extra adders in a multicycle datapath, and instead 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



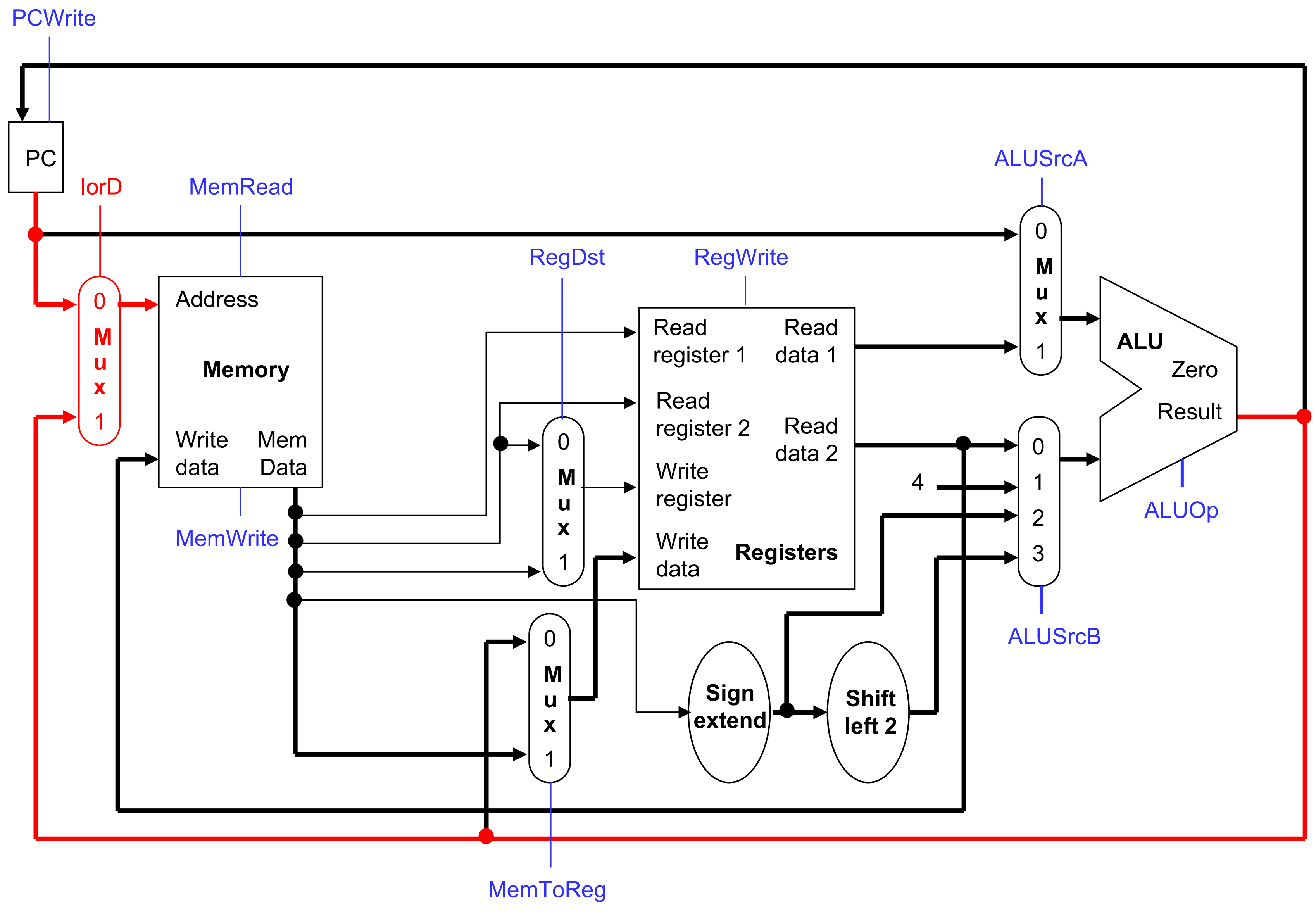
# Eliminating a memory

- Similarly, we can get by with one unified memory, which will store *both* program instructions *and* data.
- 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

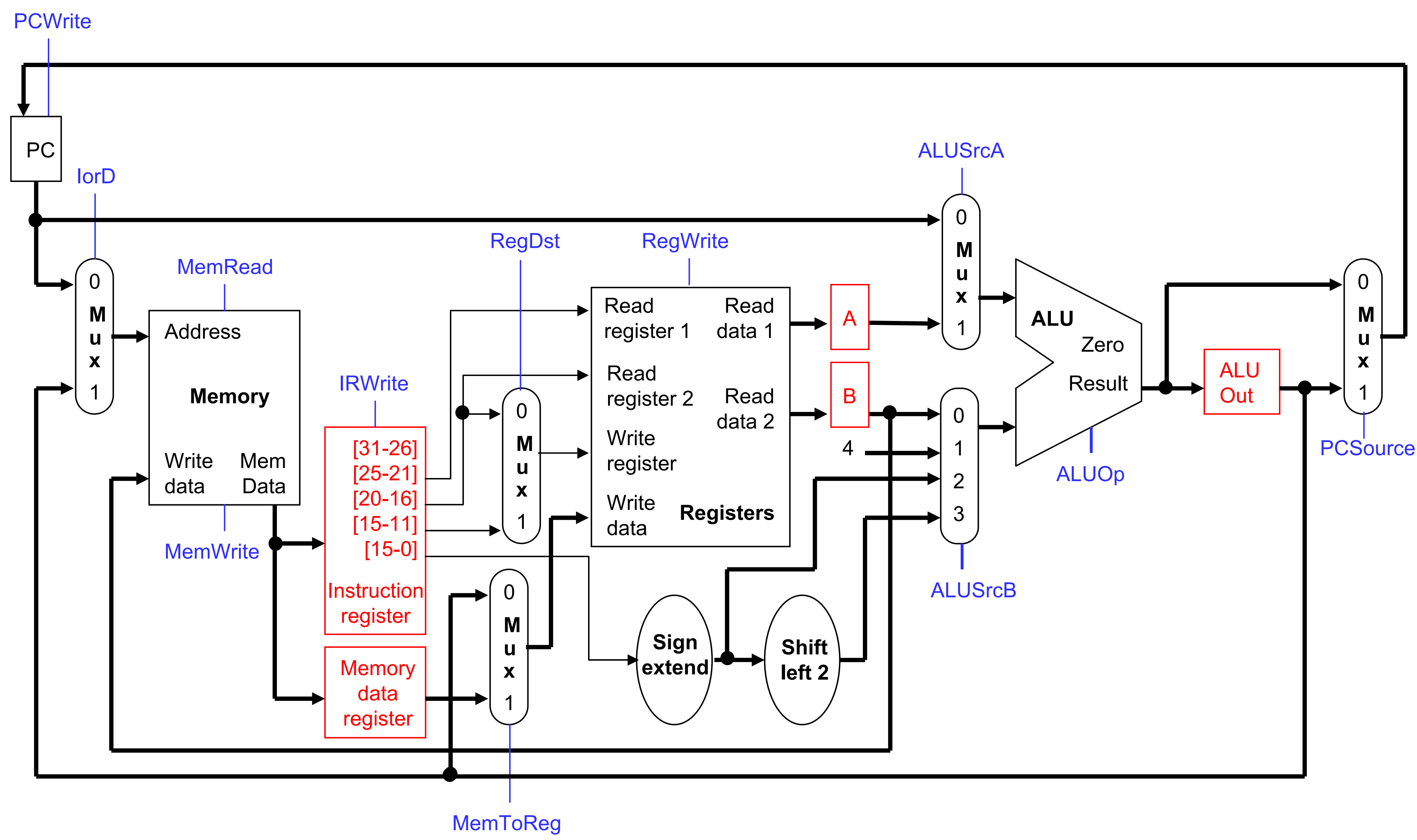
# The new memory setup highlighted



# 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 slowest instruction.
  - 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 Monday we’ll look at controlling this beast, which will also help us understand how this datapath works.