

# Lecture 8

---

- Today
  - Finish single-cycle datapath/control path
  - Look at its performance and how to improve it.

# Single-Cycle Performance

---



- Last time we saw a MIPS single-cycle datapath and control unit.
- Today, we'll explore factors that contribute to a processor's **execution time**, and specifically at the performance of the single-cycle machine.
- Next time, we'll explore how to improve on the single cycle machine's performance using pipelining.

# Three Components of CPU Performance

---

$$\text{CPU time}_{x,p} = \text{Instructions executed}_p * \text{CPI}_{x,p} * \text{Clock cycle time}_x$$

Cycles Per Instruction

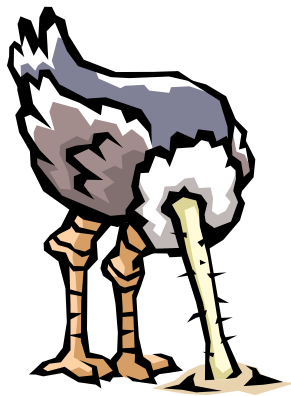


# Instructions Executed

---

- Instructions executed:
  - We are not interested in the **static instruction count**, or how many lines of code are in a program.
  - Instead we care about the **dynamic instruction count**, or how many instructions are actually executed when the program runs.
- There are three lines of code below, but the number of instructions executed would be 2001.

```
li    $a0, 1000
Ostri ch: sub  $a0, $a0, 1
      bne  $a0, $0, Ostri ch
```



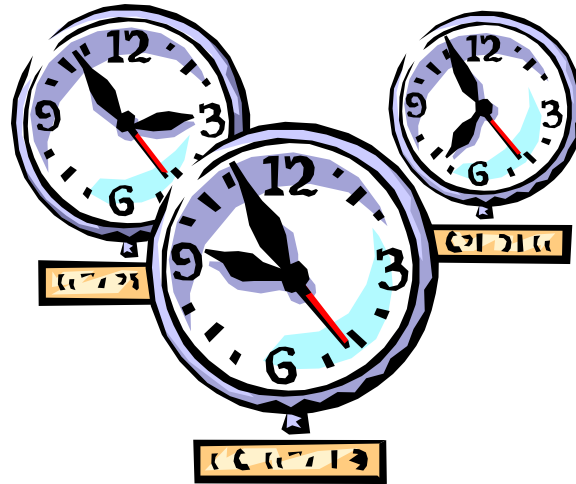
# CPI

---

- The average number of clock cycles per instruction, or **CPI**, is a function of the machine and program.
  - The CPI depends on the actual instructions appearing in the program—a floating-point intensive application might have a higher CPI than an integer-based program.
  - It also depends on the CPU implementation. For example, a Pentium can execute the same instructions as an older 80486, but faster.
- So far we assumed each instruction took one cycle, so we had  $CPI = 1$ .
  - The CPI can be  $>1$  due to memory stalls and slow instructions.
  - The CPI can be  $<1$  on machines that execute more than 1 instruction per cycle (superscalar).

# Clock cycle time

---



- One “cycle” is the minimum time it takes the CPU to do any work.
  - The **clock cycle time** or clock period is just the length of a cycle.
  - The **clock rate**, or frequency, is the reciprocal of the cycle time.
- Generally, a higher frequency is better.
- Some examples illustrate some typical frequencies.
  - A 500MHz processor has a cycle time of 2ns.
  - A 2GHz (2000MHz) CPU has a cycle time of just 0.5ns (500ps).

# Execution time, again

$$\text{CPU time}_{x,p} = \text{Instructions executed}_p * \text{CPI}_{x,p} * \text{Clock cycle time}_x$$

- The easiest way to remember this is match up the units:

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Clock cycles}}{\text{Instructions}} * \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Make things faster by making any component smaller!!

	Program	Compiler	ISA	Organization	Technology
Instruction Executed					
CPI					
Clock Cycle Time					

- Often easy to reduce one component by increasing another

# Example 1: ISA-compatible processors

---

- Let's compare the performances two x86-based processors.
  - An 800MHz AMD Duron, with a CPI of 1.2 for an MP3 compressor.
  - A 1GHz Pentium III with a CPI of 1.5 for the same program.
- Compatible processors implement identical instruction sets and will use the same executable files, with the same number of instructions.
- But they implement the ISA differently, which leads to different CPIs.

$$\begin{aligned} \text{CPU time}_{\text{AMD,P}} &= \text{Instructions}_P * \text{CPI}_{\text{AMD,P}} * \text{Cycle time}_{\text{AMD}} \\ &= \\ &= \end{aligned}$$

$$\begin{aligned} \text{CPU time}_{\text{P3,P}} &= \text{Instructions}_P * \text{CPI}_{\text{P3,P}} * \text{Cycle time}_{\text{P3}} \\ &= \\ &= \end{aligned}$$

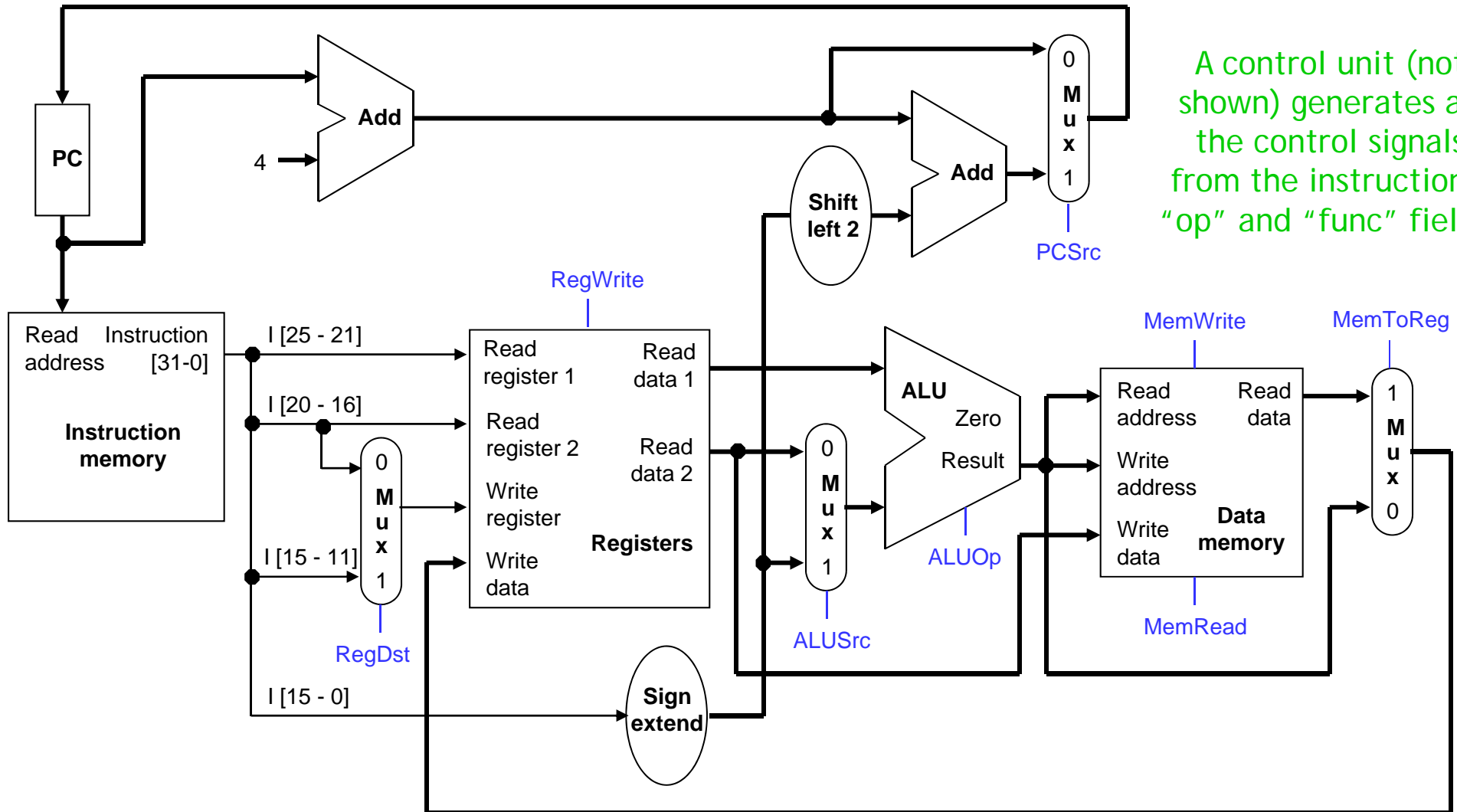


## Example 2: Comparing across ISAs

---

- Intel's Itanium (IA-64) ISA is designed facilitate executing multiple instructions per cycle. If an Itanium processor achieves an average CPI of .3 (3 instructions per cycle), how much faster is it than a Pentium4 (which uses the x86 ISA) with an average CPI of 1?
  - a) Itanium is three times faster
  - b) Itanium is one third as fast
  - c) Not enough information

# The single-cycle design from last time



# The example add from last time

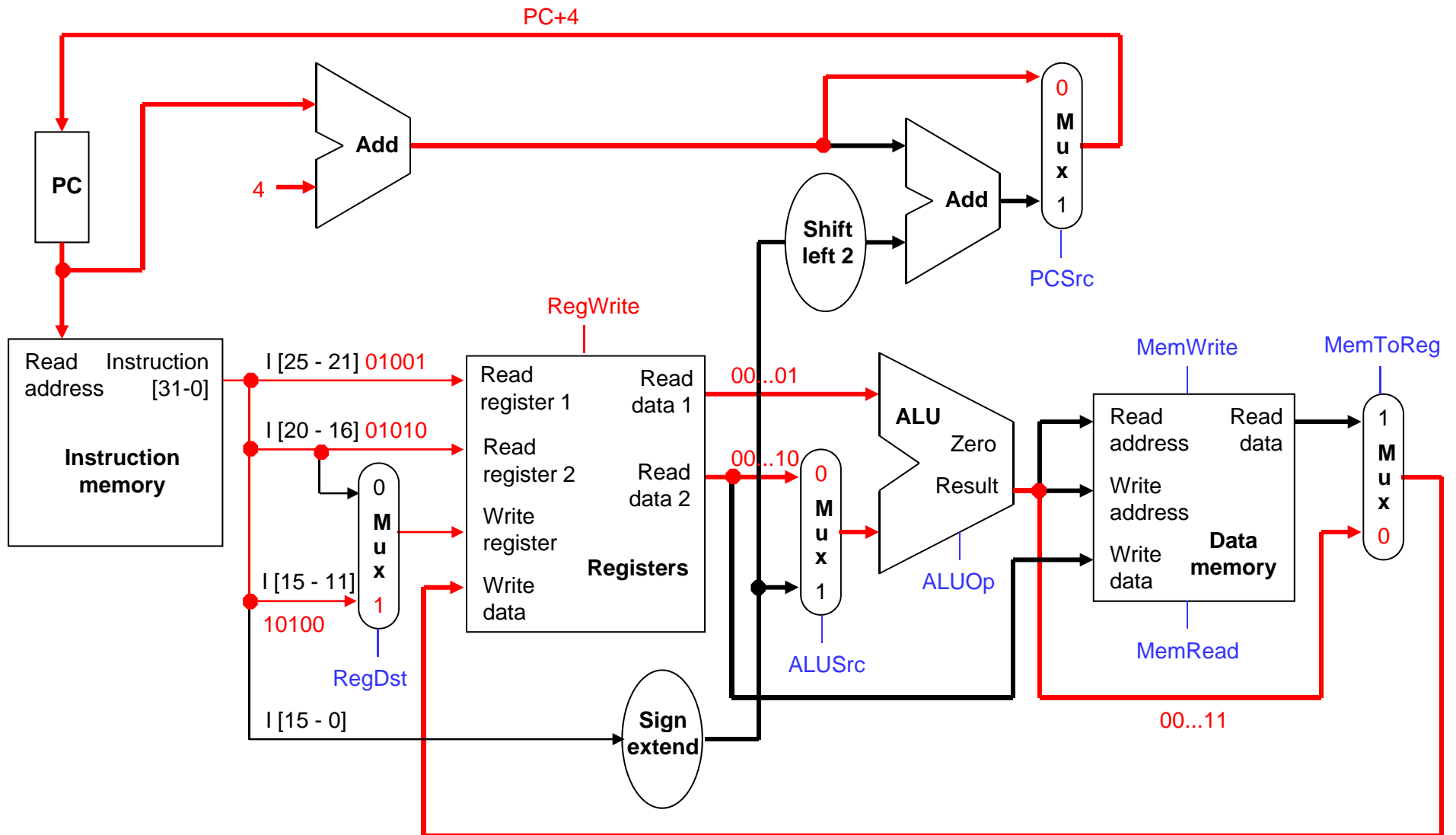
---

- 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.
  - The instruction word is read from the instruction memory, and the program counter is incremented by 4.
  - The sources `$t1` and `$t2` are read from the register file.
  - The values 1 and 2 are added by the ALU.
  - The result (3) is stored back into `$s4` in the register file.

# How the add goes through the datapath



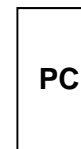
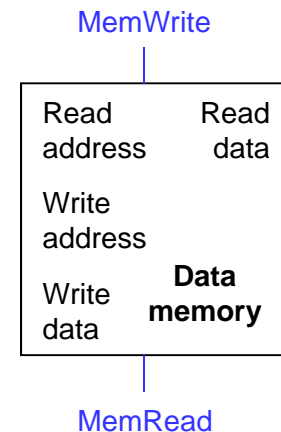
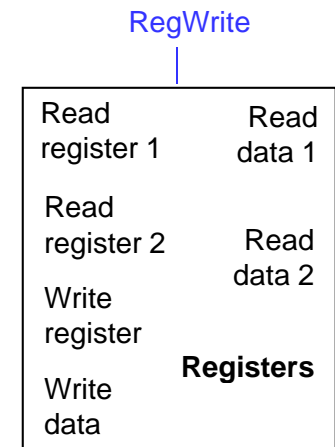
# Performance of Single-cycle Design

---

$$\text{CPU time}_{x,p} = \text{Instructions executed}_p * \text{CPI}_{x,p} * \text{Clock cycle time}_x$$

# 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 are 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

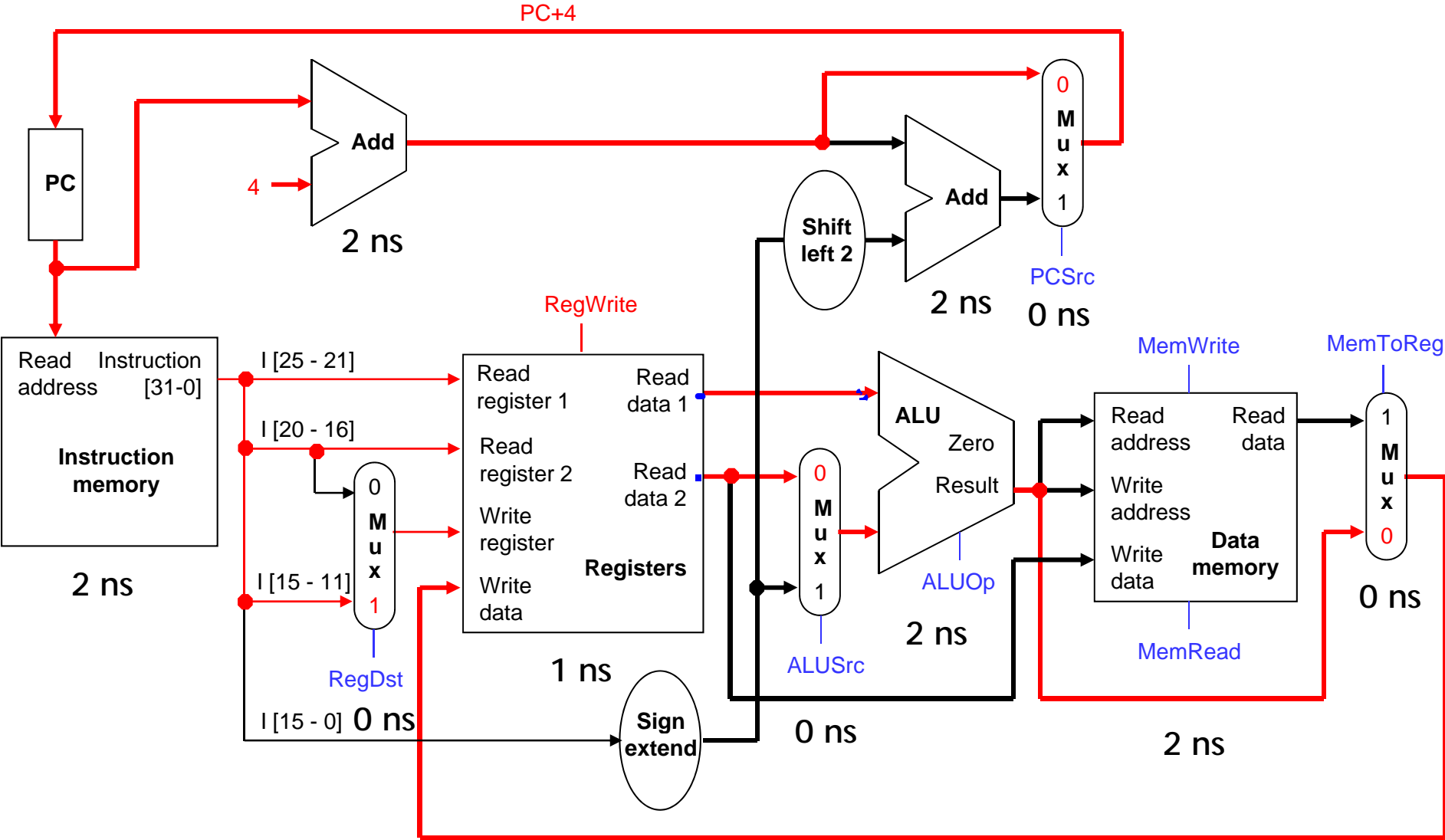
---

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.



*How long is that clock cycle?*

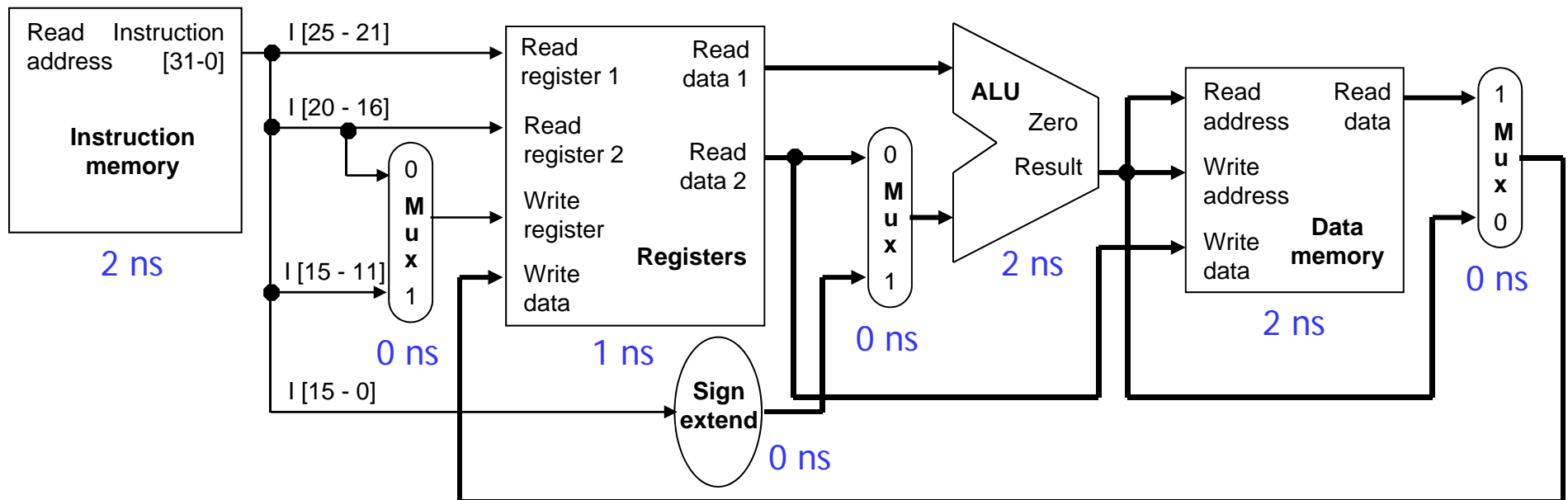
# Compute the longest path in the add instruction





# The slowest instruction...

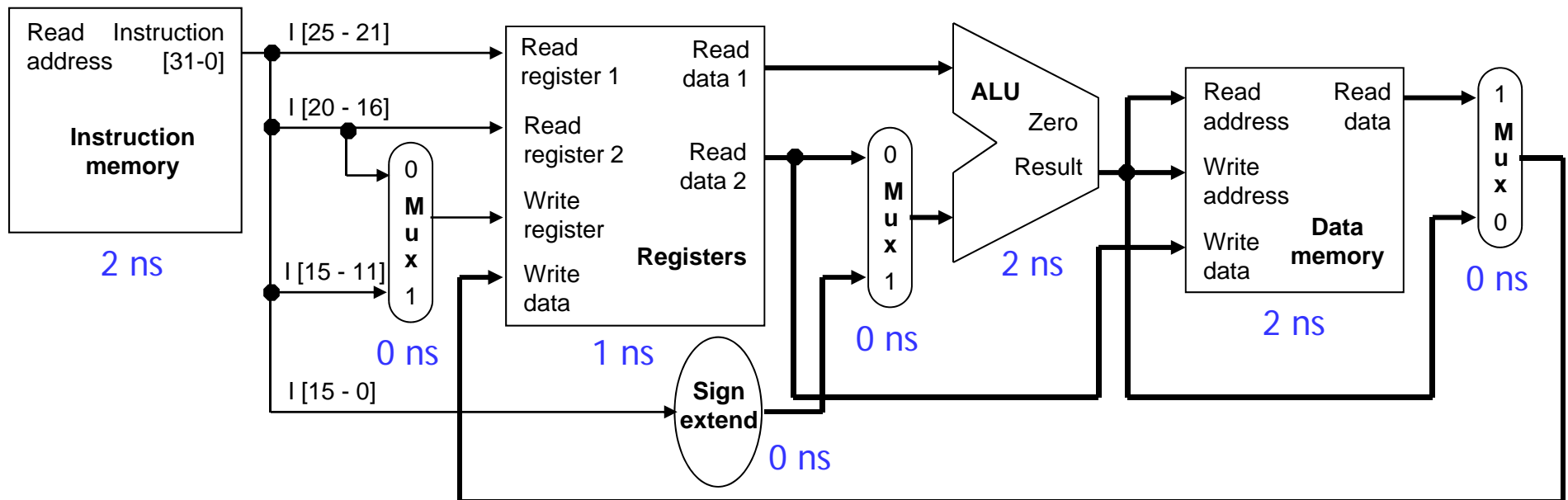
- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- For example, `lw $t0, -4($sp)` is the slowest instruction needing \_\_ns.
  - Assuming the circuit latencies below.



# 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	

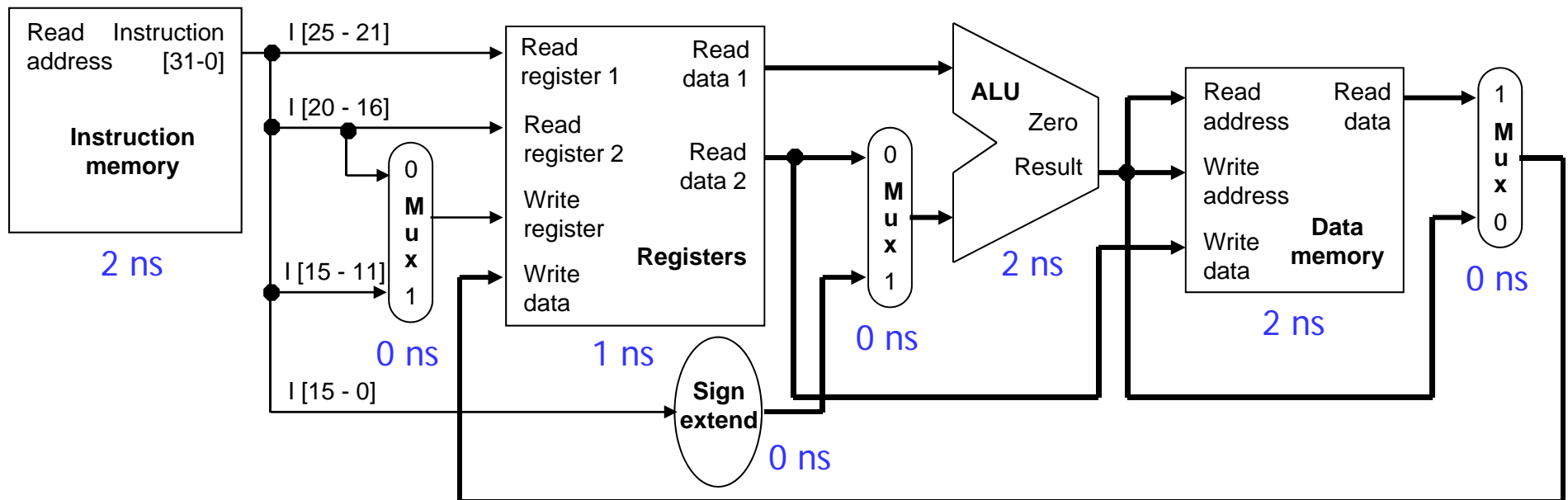


## ...determines the clock cycle time

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

} 6 ns



# 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:

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

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

## It gets worse...

---

- We've made very optimistic assumptions about memory latency:
  - Main memory accesses on modern machines is  $>50\text{ns}$ .
    - For comparison, an ALU on an AMD Opteron takes  $\sim 0.3\text{ns}$ .
- Our worst case cycle (loads/stores) includes 2 memory accesses
  - A modern single cycle implementation would be stuck at  $<10\text{Mhz}$ .
  - Caches will improve common case access time, not worst case.
- Tying frequency to worst case path violates first law of performance!!
  - “Make the common case fast” (we'll revisit this often)



# Summary

---

- **Performance** is one of the most important criteria in judging systems.
  - Here we'll focus on **Execution time**.
- Our main performance equation explains how performance depends on several factors related to both hardware and software.

$$\text{CPU time}_{x,p} = \text{Instructions executed}_p * \text{CPI}_{x,p} * \text{Clock cycle time}_x$$

- It can be hard to measure these factors in real life, but this is a useful guide for comparing systems and designs.
- A single-cycle CPU has two main disadvantages.
  - The cycle time is limited by the worst case latency.
  - It isn't efficiently using its hardware.
- Next time, we'll see how this can be rectified with pipelining.

