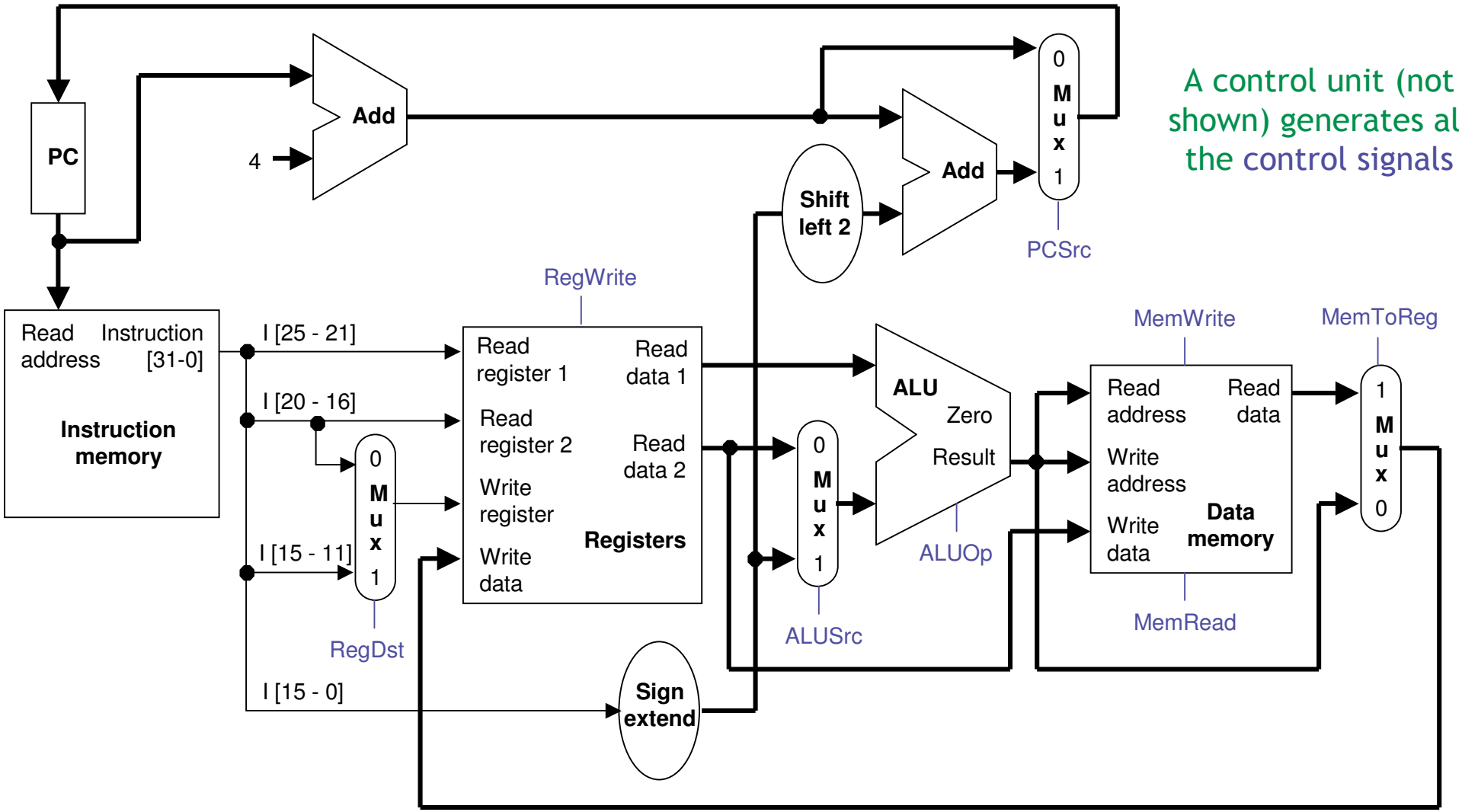
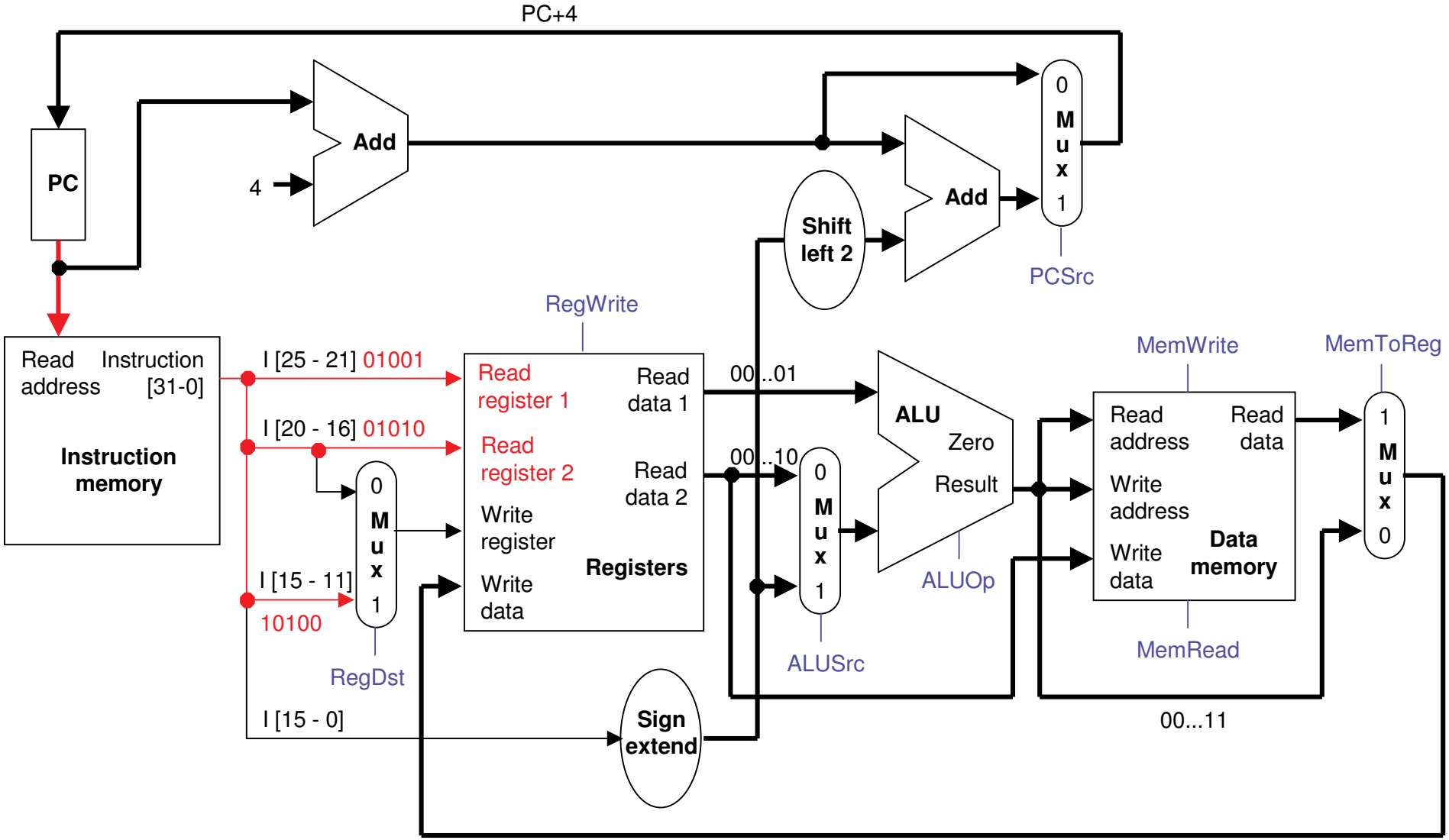
The single-cycle design from last time



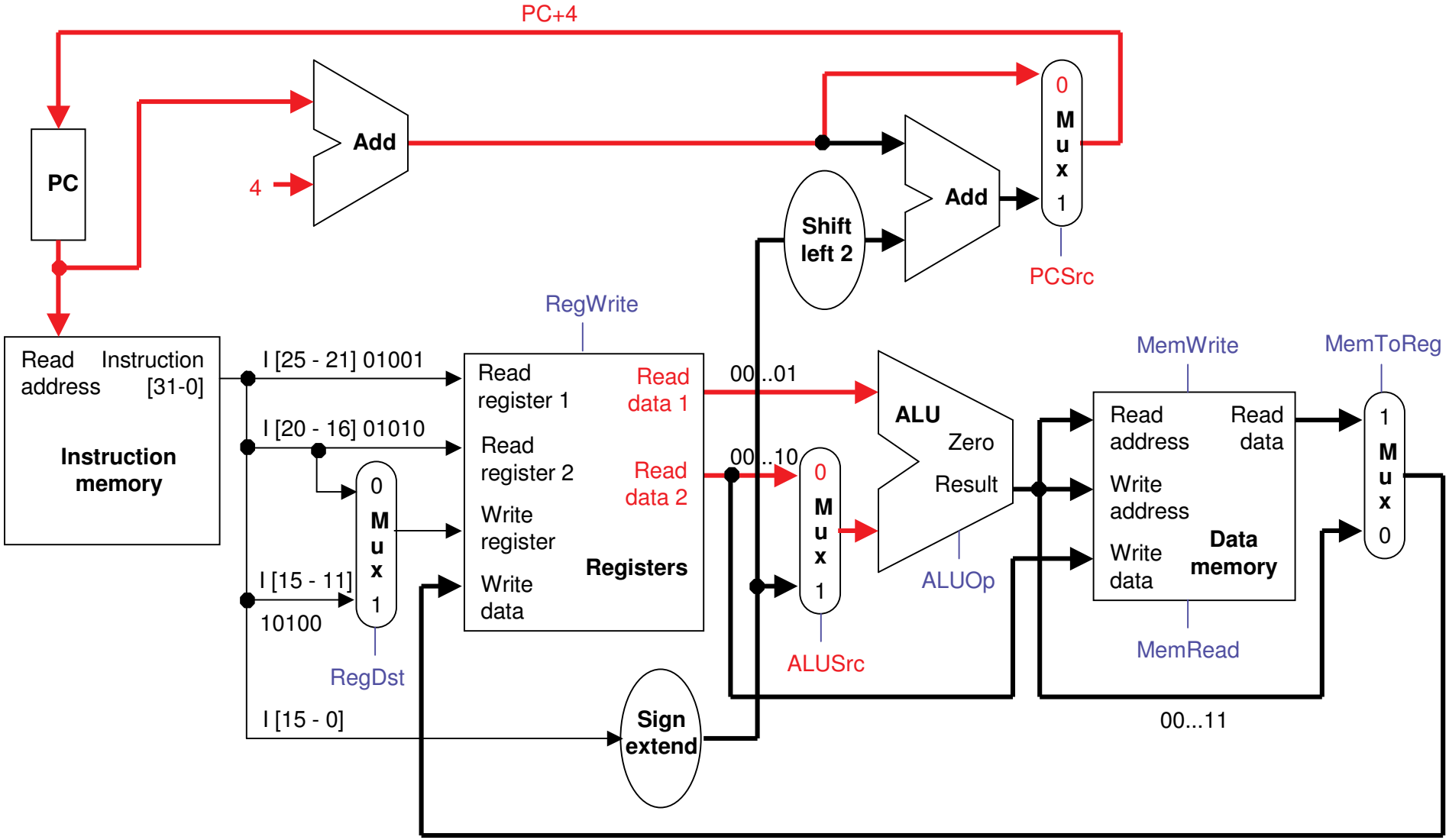
A control unit (not shown) generates all the control signals

How add $s4, $t1, $t2 goes through the datapath



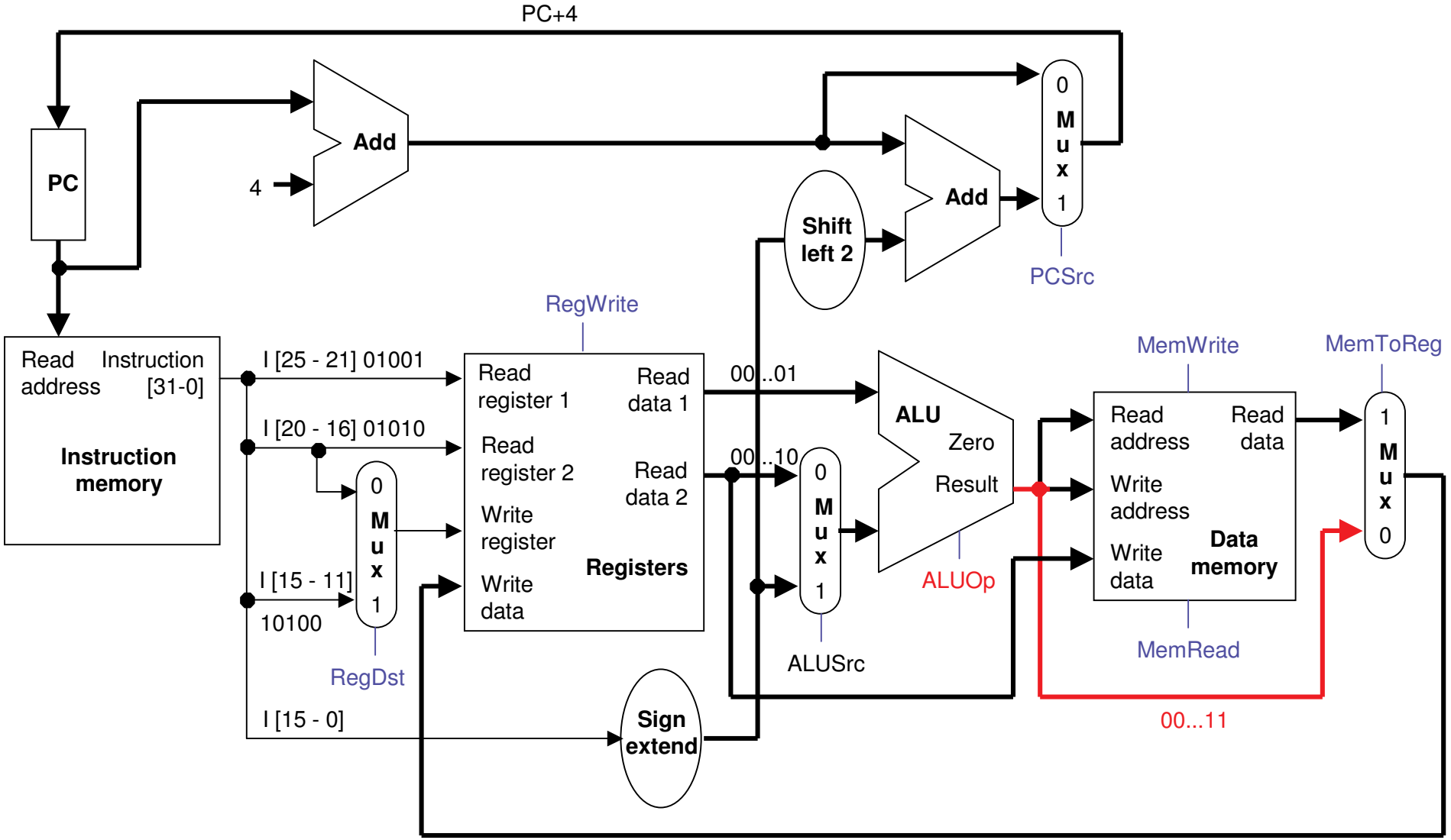
Read instruction from memory

How add $s4, $t1, $t2 goes through the datapath



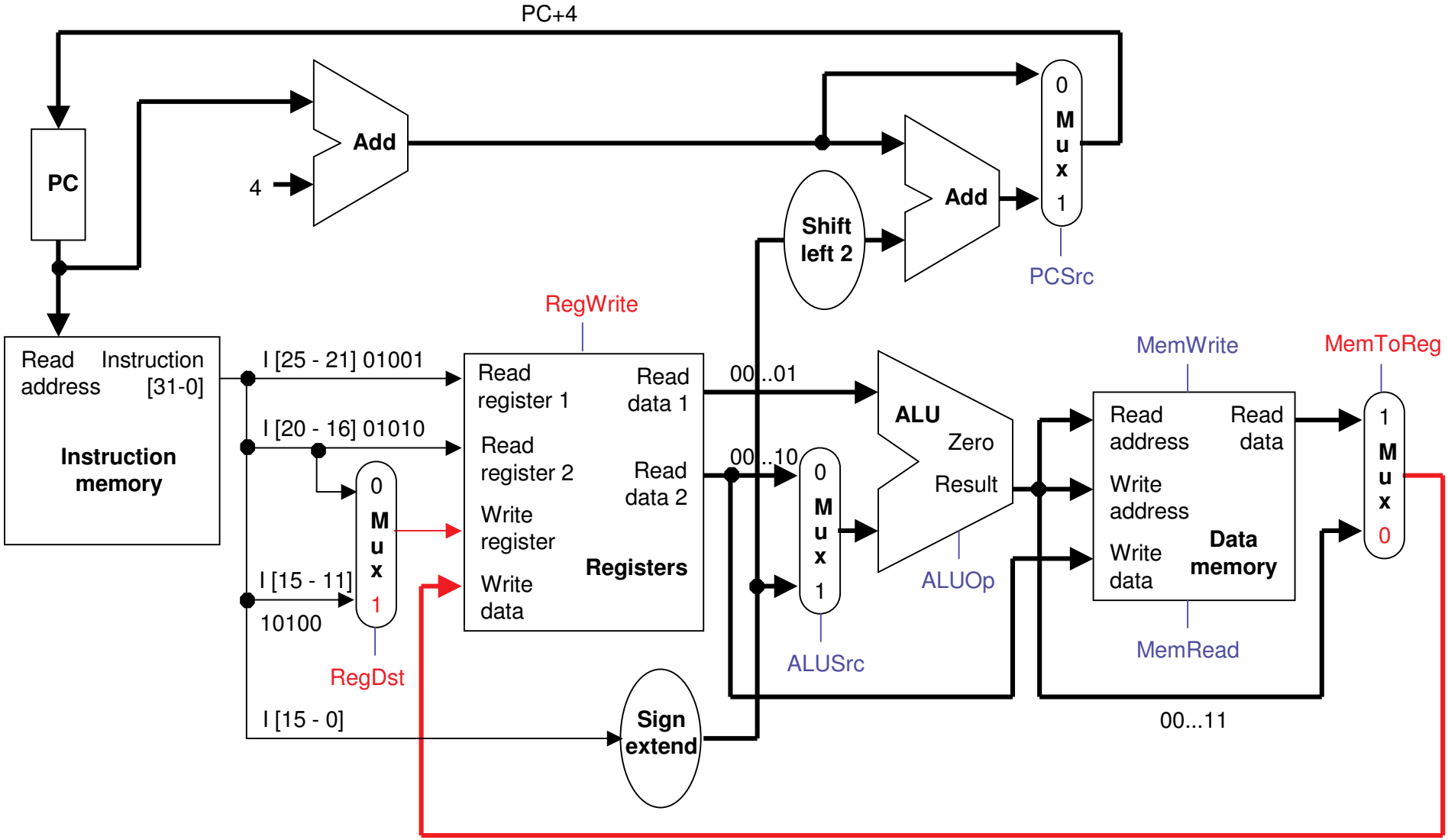
Read registers, increment PC + 4, set new PC

How add $s4, $t1, $t2 goes through the datapath



Perform addition in ALU

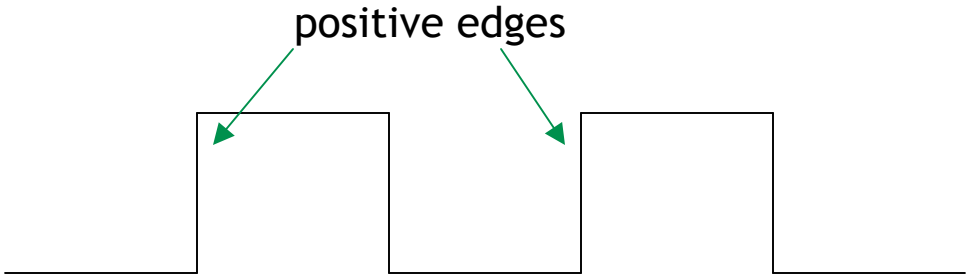
How add $s4, $t1, $t2 goes through the datapath

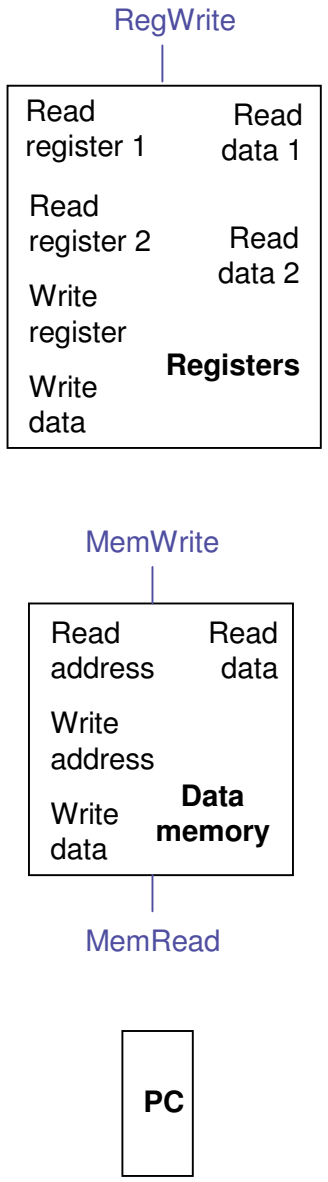


Write result back to target register

Timing issues: clock edges

- 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.





Performance of Single-cycle Design

$$\text{CPU time}_{X,P} = \text{Instructions executed}_P * \text{CPI}_{X,P} * \text{Clock cycle time}_X$$

CPI = 1 for a single-cycle design

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 for a lw instruction, 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.

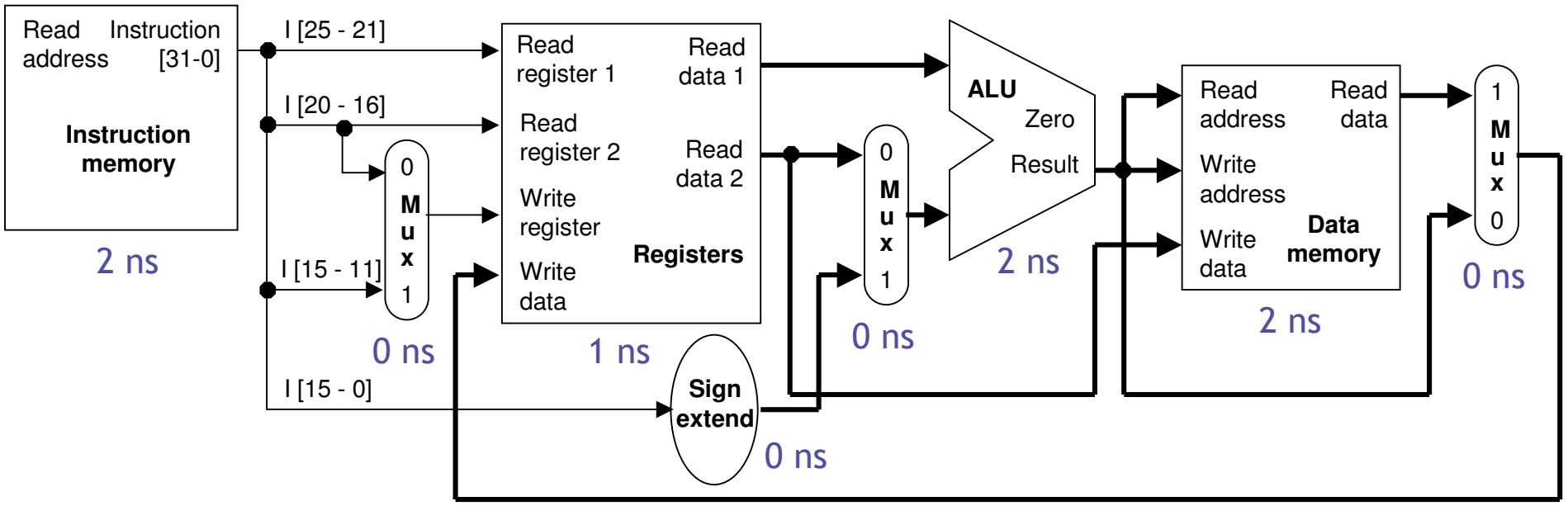
- In a single-cycle datapath everything in Step 2 must complete within one clock cycle, before the next positive clock edge.

How long is that clock cycle?

Components of the data-path

- If all instructions must complete within one clock cycle, then the cycle time has to be large enough to accommodate the *slowest* instruction.
- Each component of the datapath has an associated *delay* (latency)

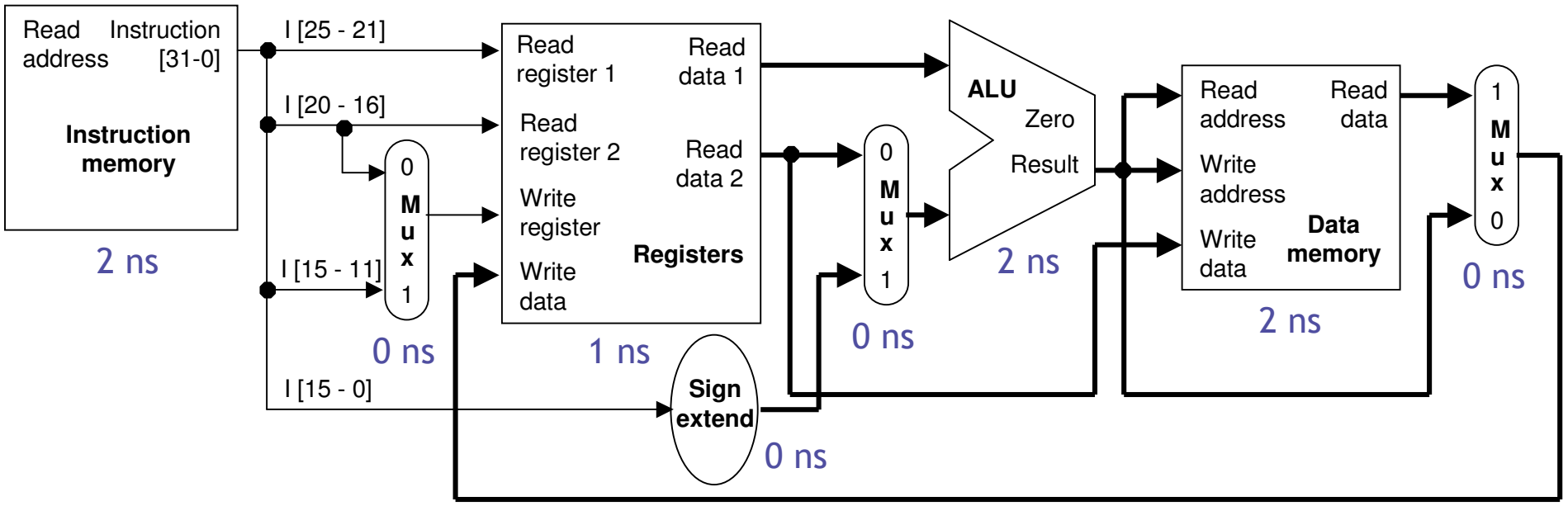
reading the instruction memory	2ns	8ns
reading the register file	1ns	
ALU computation	2ns	
accessing data memory	2ns	
writing to the register file	1ns	



The slowest instruction...

- A lw instruction uses *all* components of the datapath
- For example, lw $t0, -4($sp) needs 8ns, assuming the delays shown here.

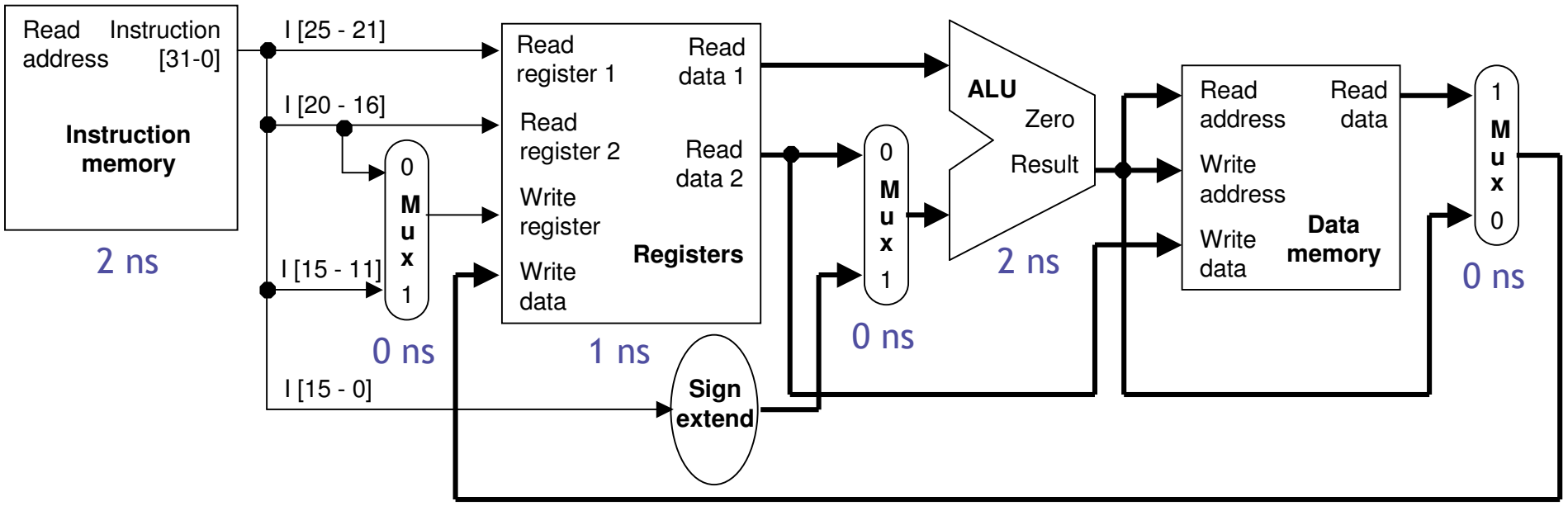
reading the instruction memory	2ns	8ns
reading the base register $sp	1ns	
computing memory address $sp-4	2ns	
reading the data memory	2ns	
storing data back to $t0	1ns	



...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	6ns
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.

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% x 6ns) + (22% x 8ns) + (11% x 7ns) + (19% x 5ns) = 6.36ns

- 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 >50ns.
  - For comparison, an ALU operation 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!!

Improving performance

- Two ideas for improving performance:

  1. Spilt each instruction into multiple steps, each taking 1 cycle
     - steps: IF (instruction fetch), ID (instruction decode), EX (execute ALU operation), MEM (memory access), WB (register write-back)
     - slow instructions take more cycles than fast instructions
     - known as a *multi-cycle* implementation

  2. Crucial observation: each instruction uses only a *portion* of the datapath in each step
     - can *overlap* instructions; each uses one portion of the datapath
     - known as a *pipelined* implementation

- Examples of pipelining: any assembly process (cars, sandwiches), multiple loads of laundry (washer + dryer can be pipelined), etc.

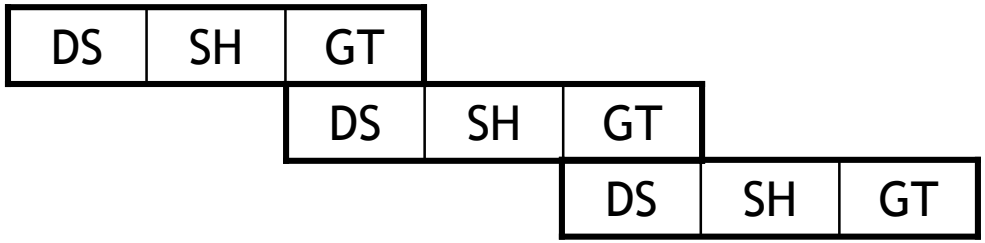
Pipelining: a SPIMbot example

- The basic operation can be broken into three steps:
  - Do Scan (DS), which initiates a scan
  - Scan Handler (SH), which processes the scan
  - Get Tokens (GT), which picks the tokens up
- Our current implementation is "multi-cycle"

- Key observation:

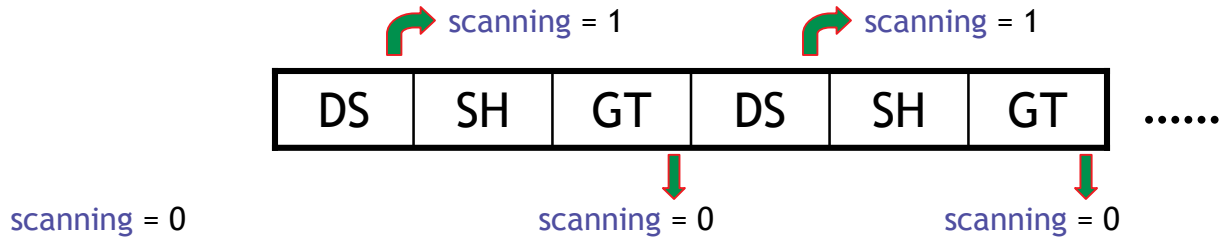
  SH must finish before next SH starts (since both write to the same memory address)…

  … but DS can start as soon as previous SH is complete
- Thus, we can overlap DS and GT:

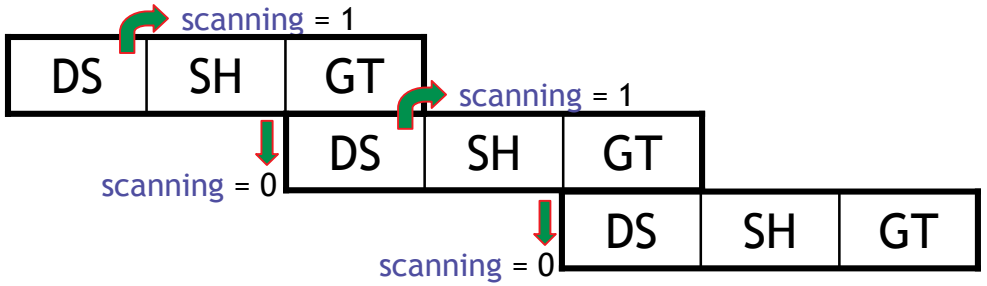


Implementing simple pipeline

- Multi-cycle implementation:



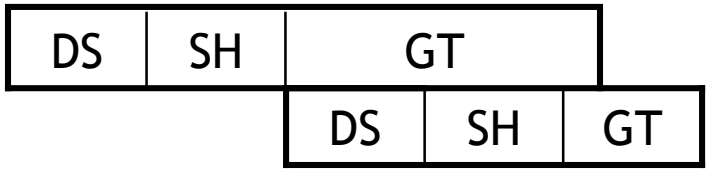
- **Pipelined implementation:**



A complication: GT can be slow

- In SH, we determine `scan_length` and set `index` to zero

- So, the following situation is bad since SH interferes with GT



- How can we avoid this problem?

  Change the code in SH so that each new token found is placed at the *back of a queue*. The easiest way to implement queues is as arrays. Treat `xcoord` and `ycoord` as queues by adding new tokens to the end (and increase `scan_length` by 1). In GT, once `index` equals `scan_length` (i.e. all tokens in the queue are found), simply set `index` and `scan_length` to 0.

- We can now potentially pipeline even better…