

**CSE 260M / ESE 260**  
**Intro. To Digital Logic & Computer Design**

Bill Siever  
&  
Jim Feher

# This week

- Thursday:  
    Studio — Here / Seigle 301  
    Bring kits (1 per group)— will be used briefly
- Hw #7 — Soon. Watch for announcement
- Tuesday (Nov. 26): Wrap up lecture topics; Exam 2 Review
- Thurs (Nov 28): Thanksgiving Break
- Tues (Dec. 3): Exam 2. IN *Hillman 70*

# Studio Review

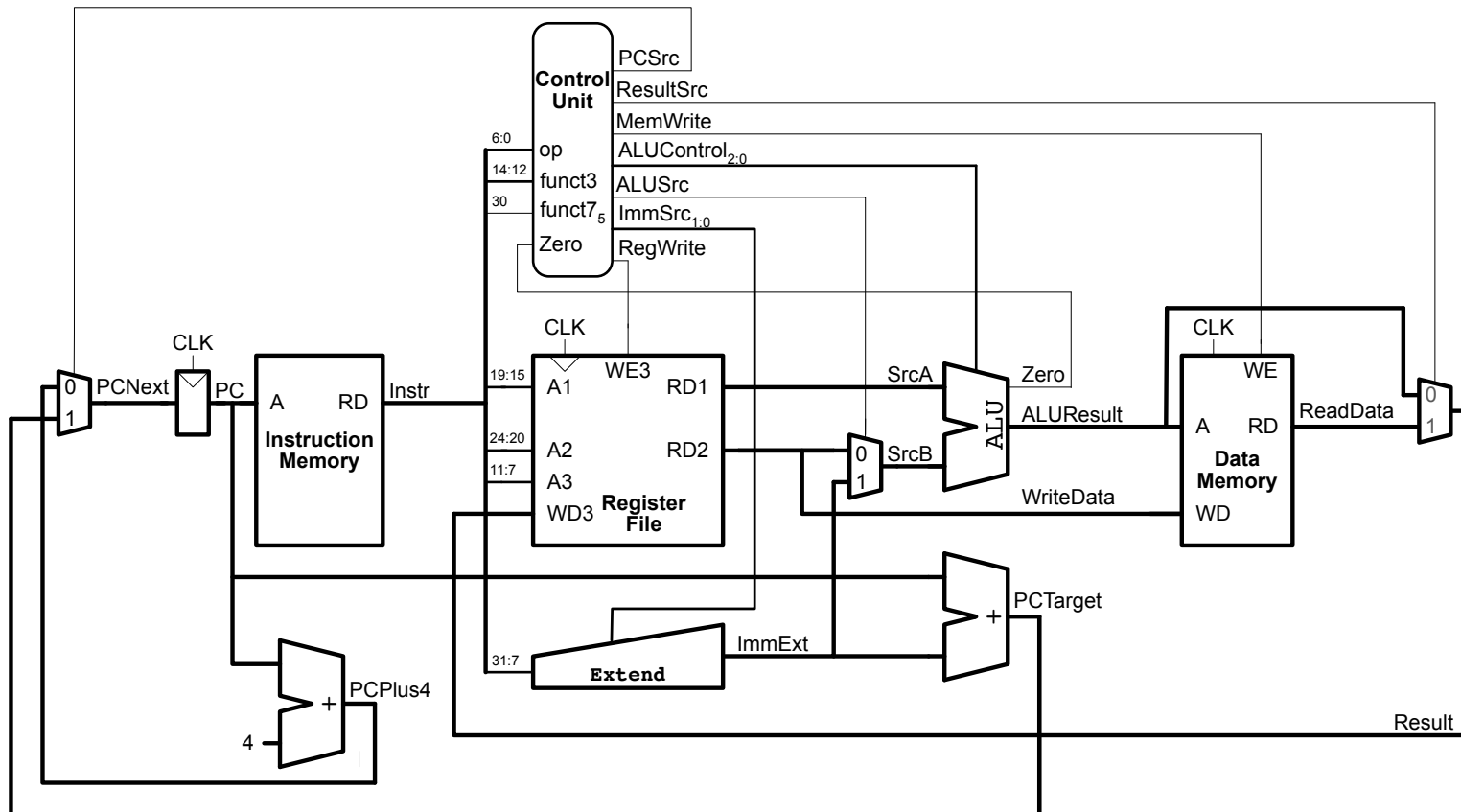
- Led.sv: Creating an “active high” signal
  - Because LEDs are built to share the +V
- Switches.sv: More “active high” stuff (pull up)
  - Because iCE40 is built to pull-up (but not down)
- Fulladder project / parts (fulladder.sv / top.sv)
- Blink
  - Multiple modules / hierarchy
  - Multiple processes / counting

# Chapter 7

# Architectures: RISC-V

- Three major kinds of things instructions do:
  - Computation (use the ALU; do basic math/logic): `add`, `addi`, `or`, ...
  - Move data (between registers or registers to/from memory):  
`mv*`, `lw`, `sw`, ...
  - Control program flow (which instruction happens next): `beq`, `blt`, `cal`,  
...

# Simple RISC-V Computer



# The Problem

**Find  $x$  such that  $2^x = 128$**

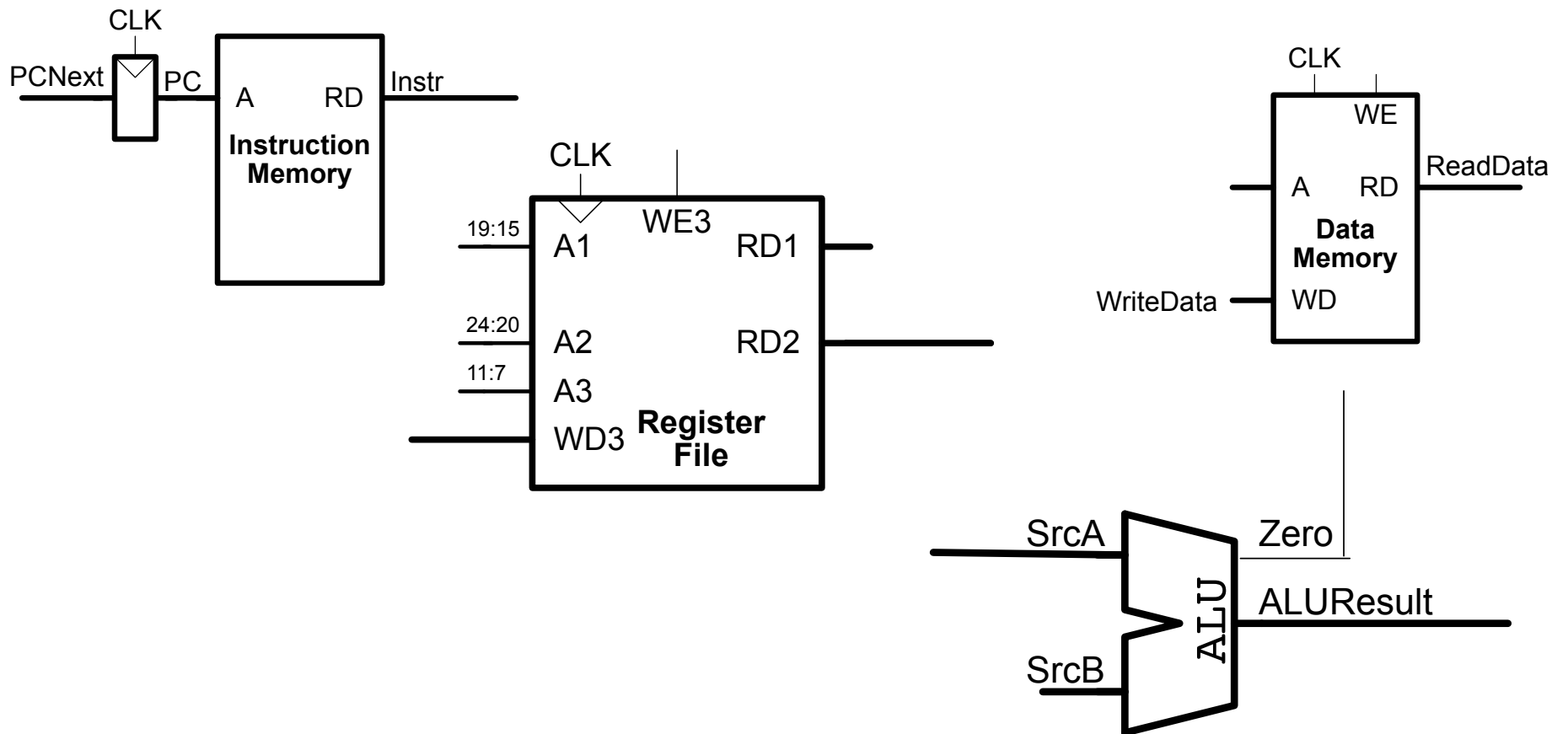
**Problem: Find x such that  $2^x = 128$**

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x = 0;

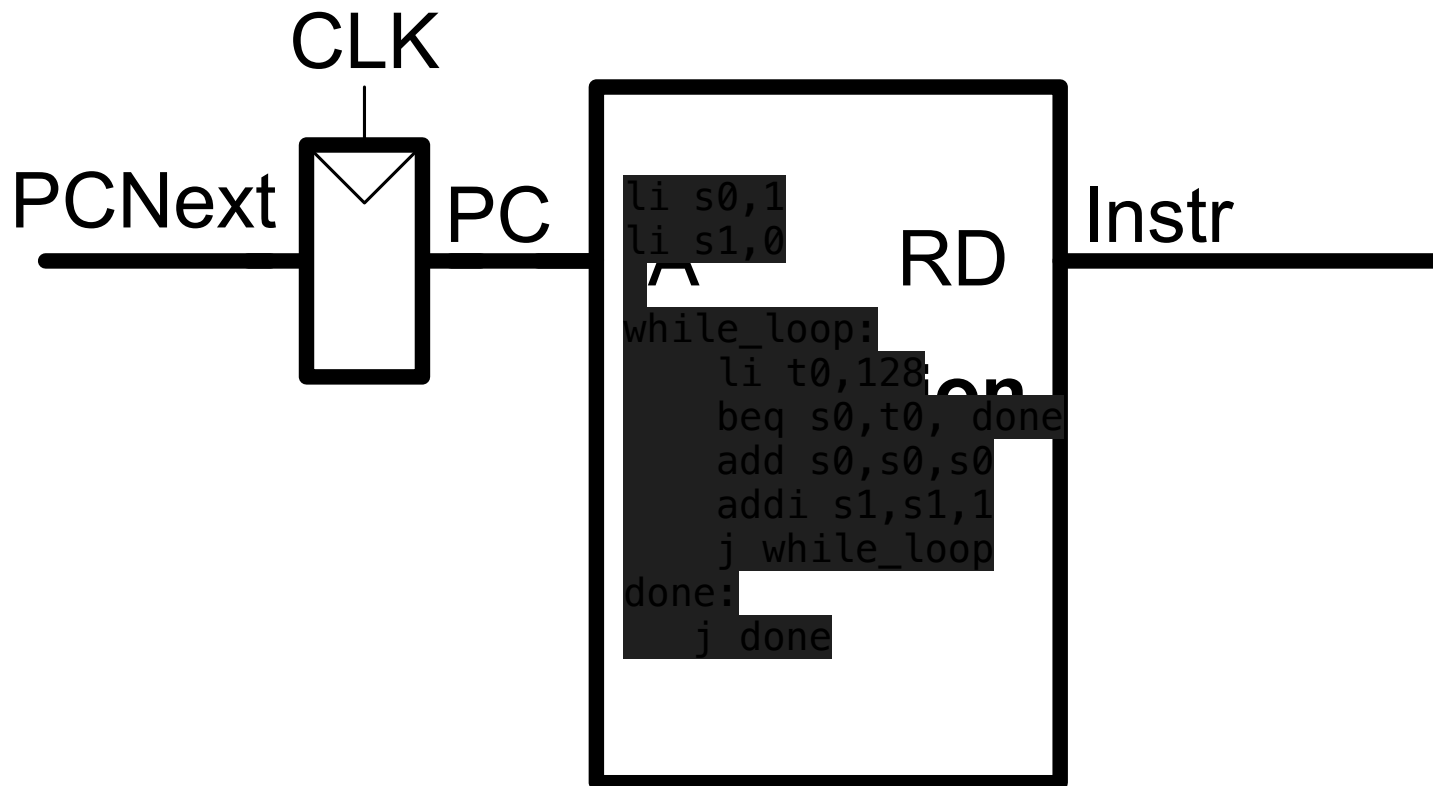
while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```



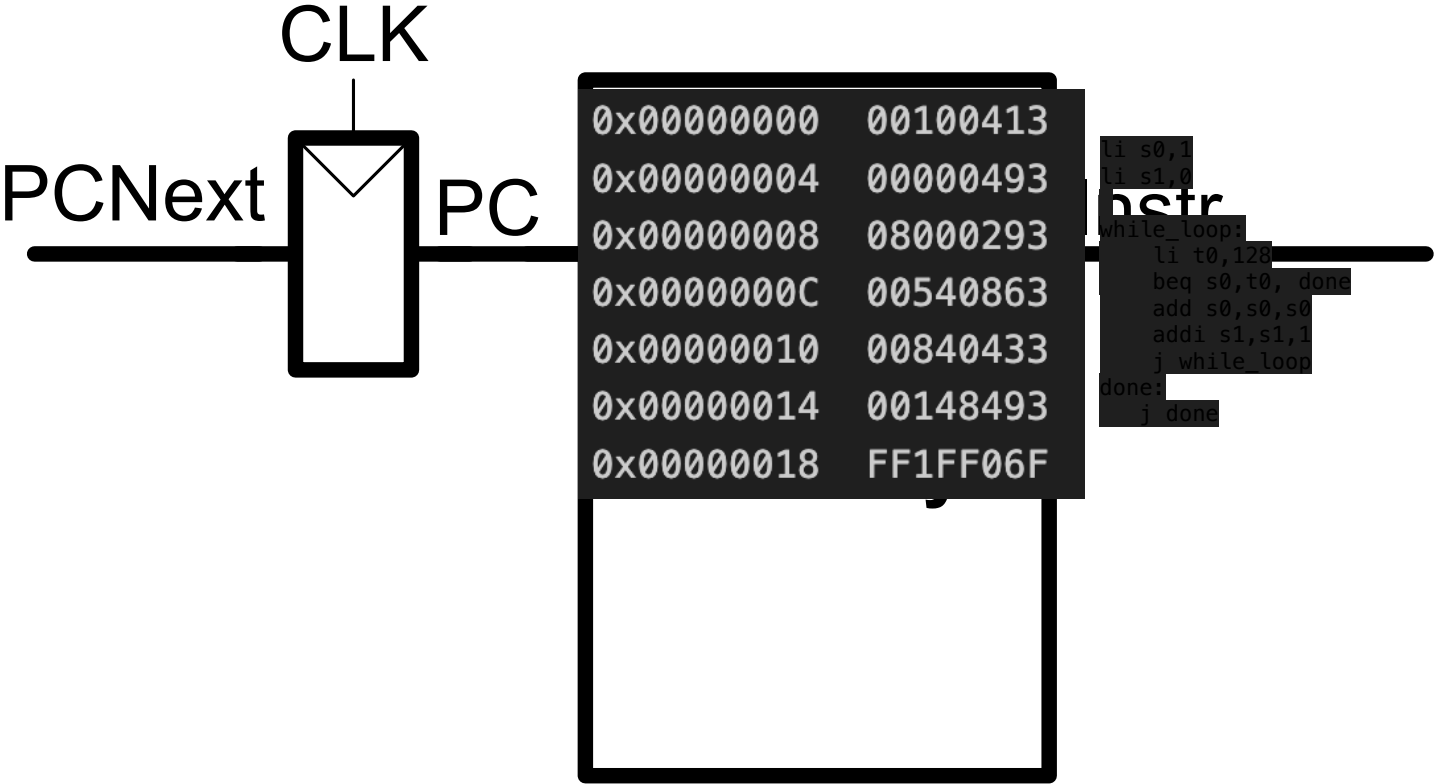
# Behavior: Parts of CPU Model



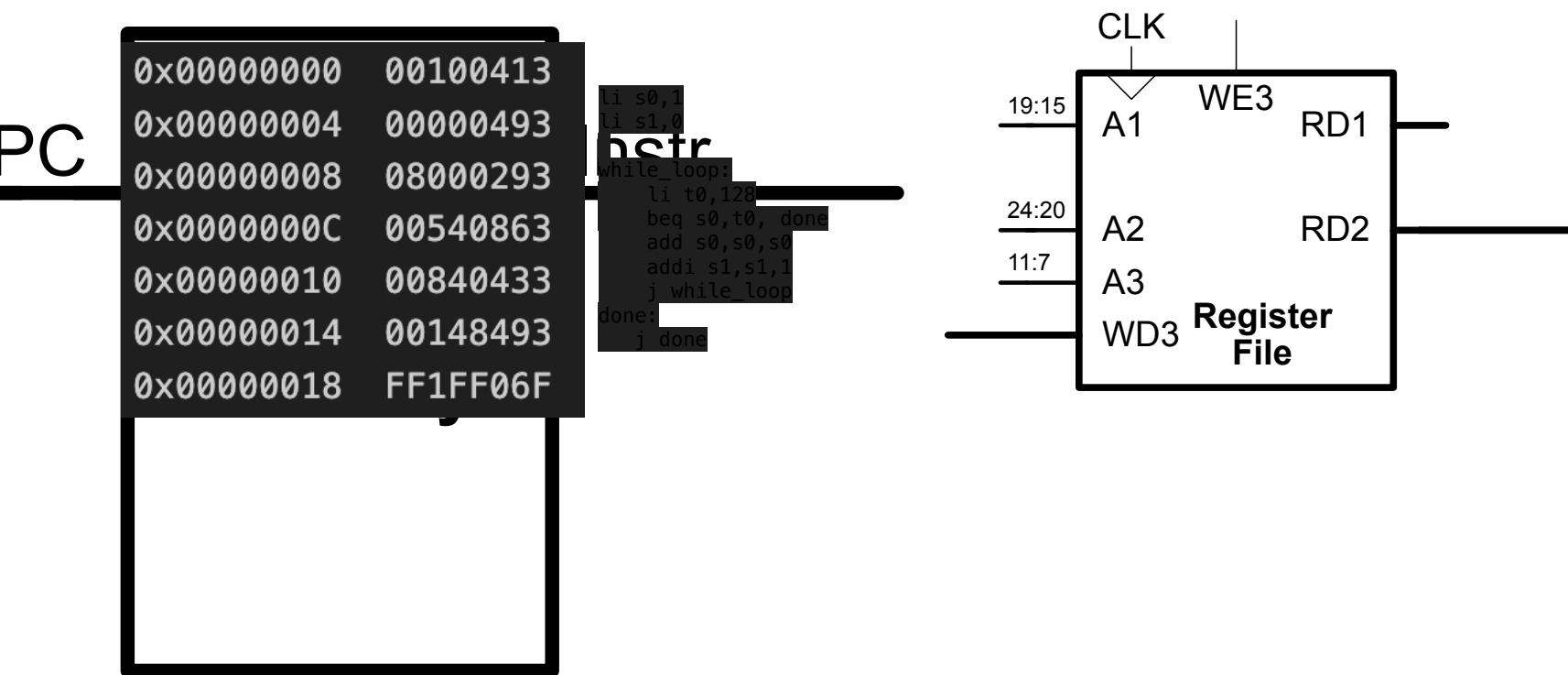
# Behavior: Parts of CPU Model



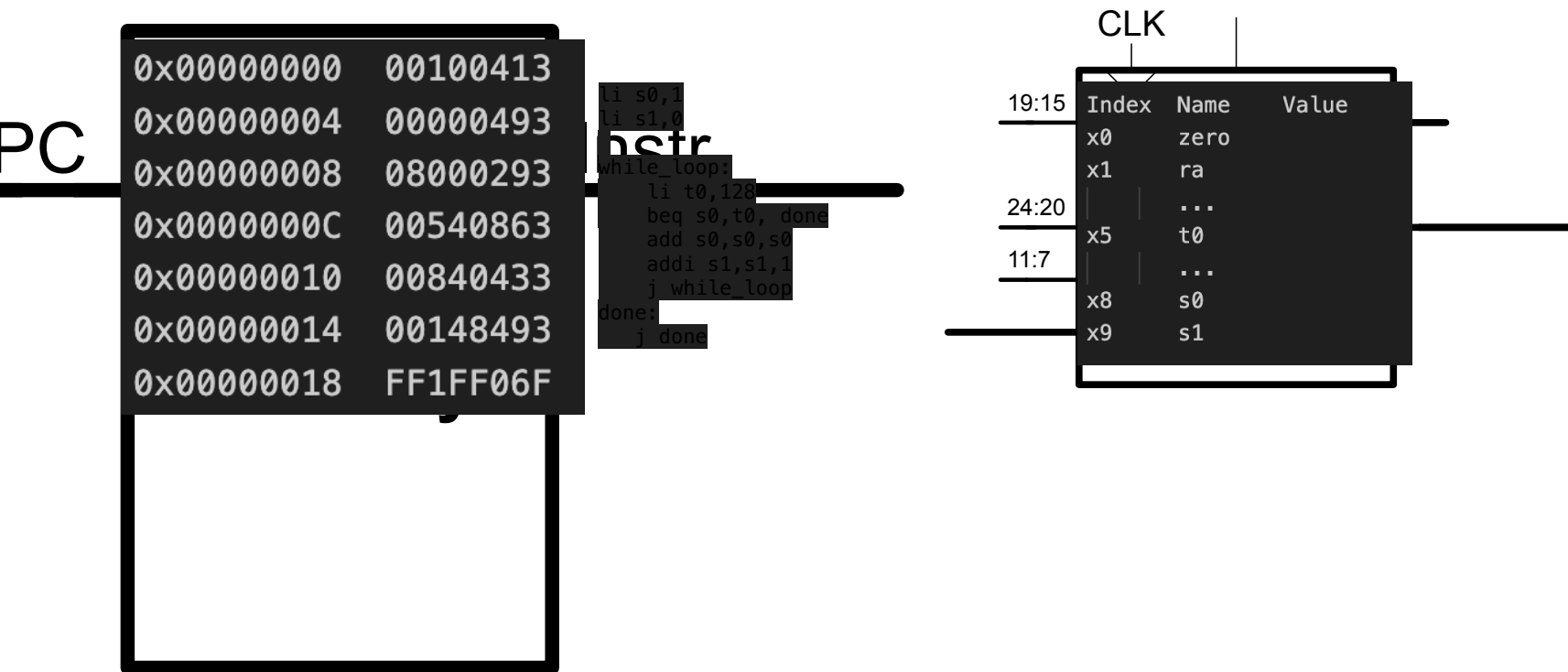
# Behavior: Parts of CPU Model



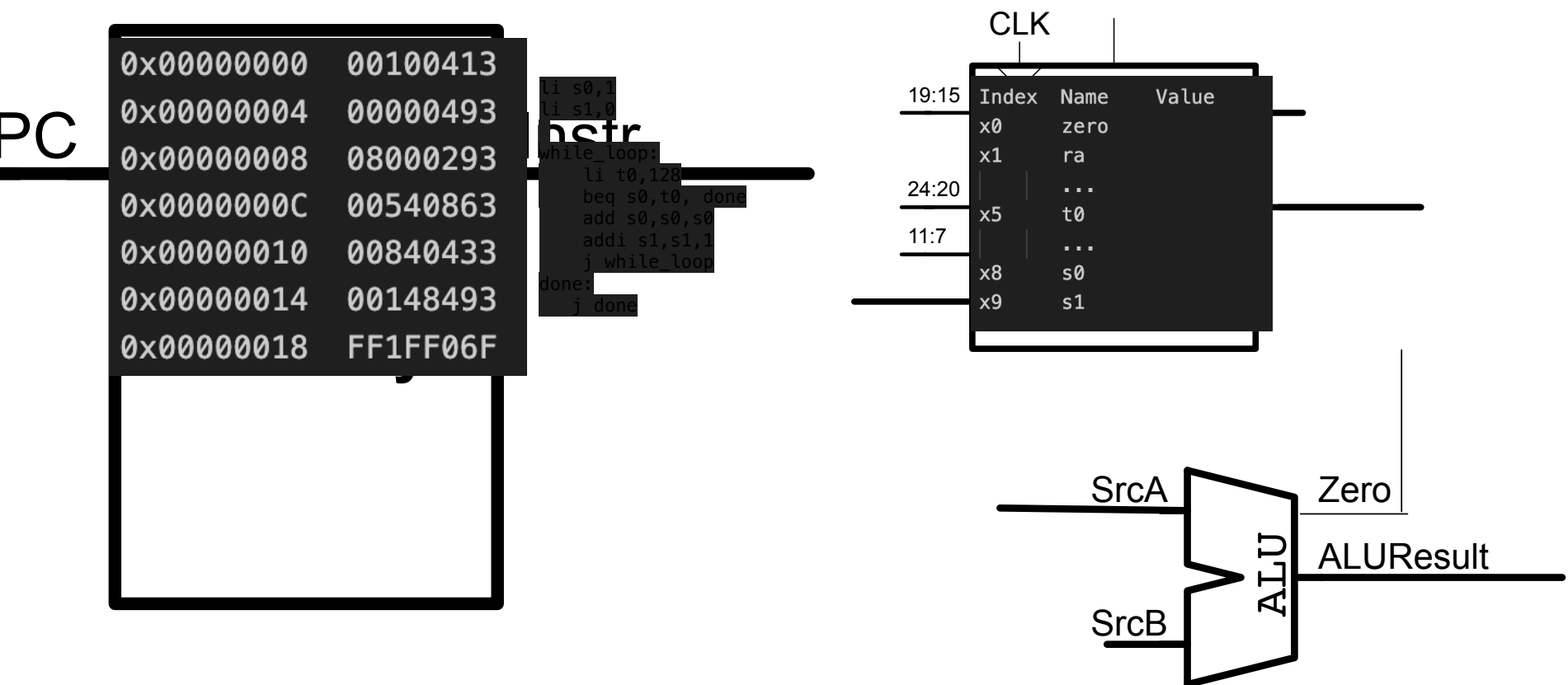
# Behavior: Parts of CPU Model



# Behavior: Parts of CPU Model



# Behavior: Parts of CPU Model

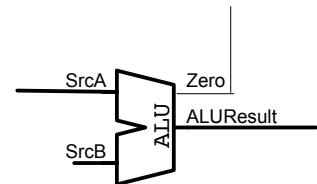


# Parts of CPU Model

```
0x00000000 00100413
0x00000004 00000493
0x00000008 08000293
0x0000000C 00540863
0x00000010 00840433
0x00000014 00148493
0x00000018 FF1FF06F
```

```
li s0,1
li s1,0
while_loop: li t0,128
            beq s0,t0, done
            add s0,s0,s0
            addi s1,s1,1
            j while_loop
done: Clk done
```

Index	Name	Value
x0	zero	
x1	ra	
...	...	
x5	t0	
...	...	
x8	s0	
x9	s1	



# RAM

- RISC-V: “Load-Store Architecture”
  - Transfer from RAM to register (load data)
  - Transfer from register to RAM (store data)
- Alternative architectures (x86) able to directly combine data from memory with register data



# RISC-V Load/Store

- Mnemonics: load word ( $lw$ ) / store word ( $sw$ )
- Memory Address Format:  $4(s0)$   
offset(base)
- Address calculation: RAM address (index) = add base address to the offset  
Ex: ( $s0 + 4$ ). (Read as “4 past  $s0$ ”)
- Array notation:  $RAM[s0+4]$
- In terms of “register” array:  $RAM[REG[s0]+4]$   
Since  $s0$  is x8 / index 8 in registers:  $RAM[REG[8]+4]$

# Big Picture

- Operations can be represented as numbers
- Operations manipulate memory, registers, and the “Program Counter”
  - Program Counter mostly “counts” (by 4s here, since each instruction is 4 bytes)
- Labels are addresses
  - An address is also an index into something, like RAM

# Big Picture

- Stereotypical “Instruction Cycle”
  - Fetch an instruction (from memory)
  - Decode it (wait for combinational logic to setup for execution)
  - Execute (wait for combinational logic to complete and for memory elements to update)
- Example Program & Animation: <https://eseo-tech.github.io/emulsiV/>
  - Test2.s: Add up some numbers...

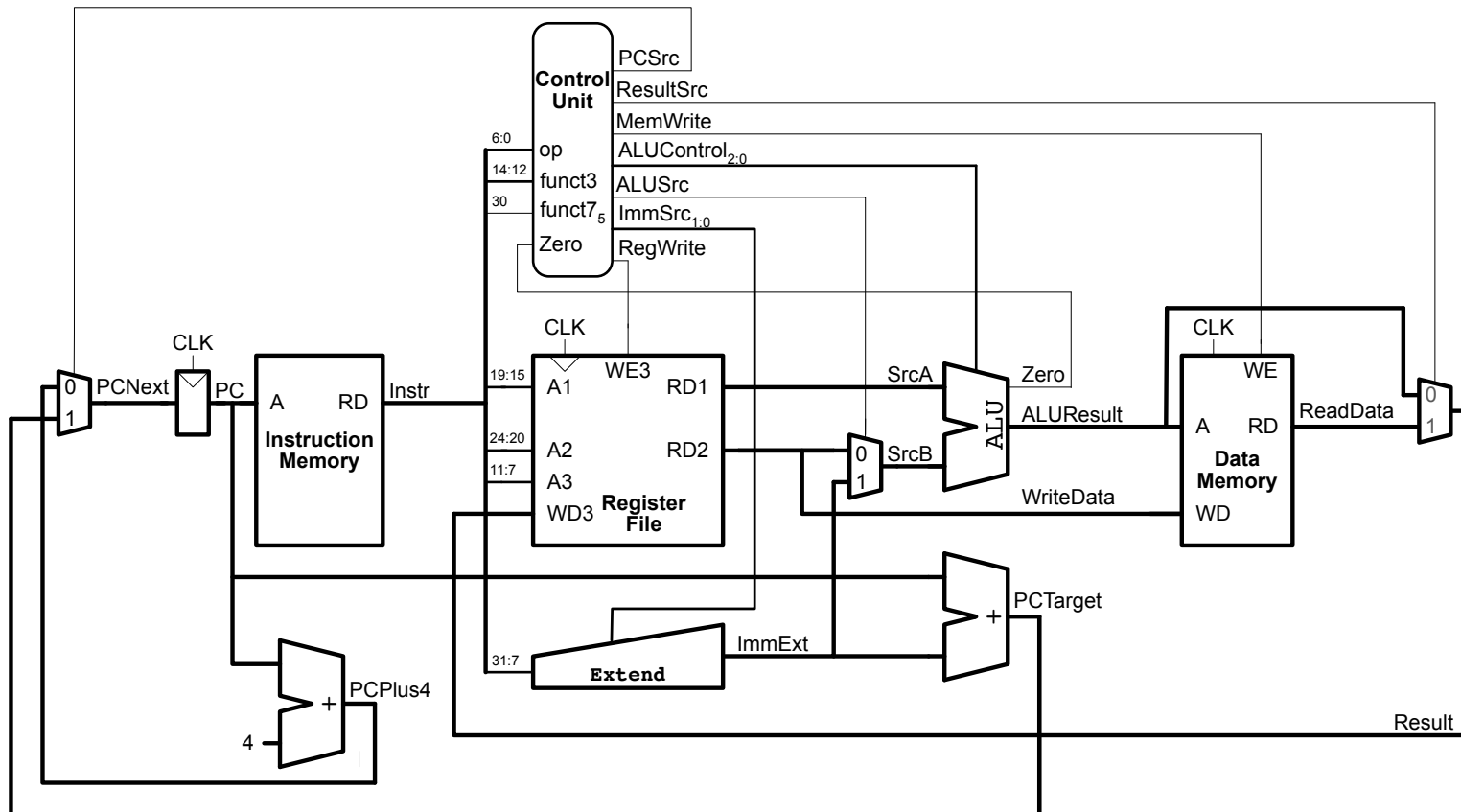
# Test: Add numbers up to 123

```
.text
li t0, 123
li t1, 0
loop:
    add t1, t1, t0
    addi t0, t0, -1
    bne zero, t0, loop
end:
    j end
```

# Architecture & MicroArchitecture

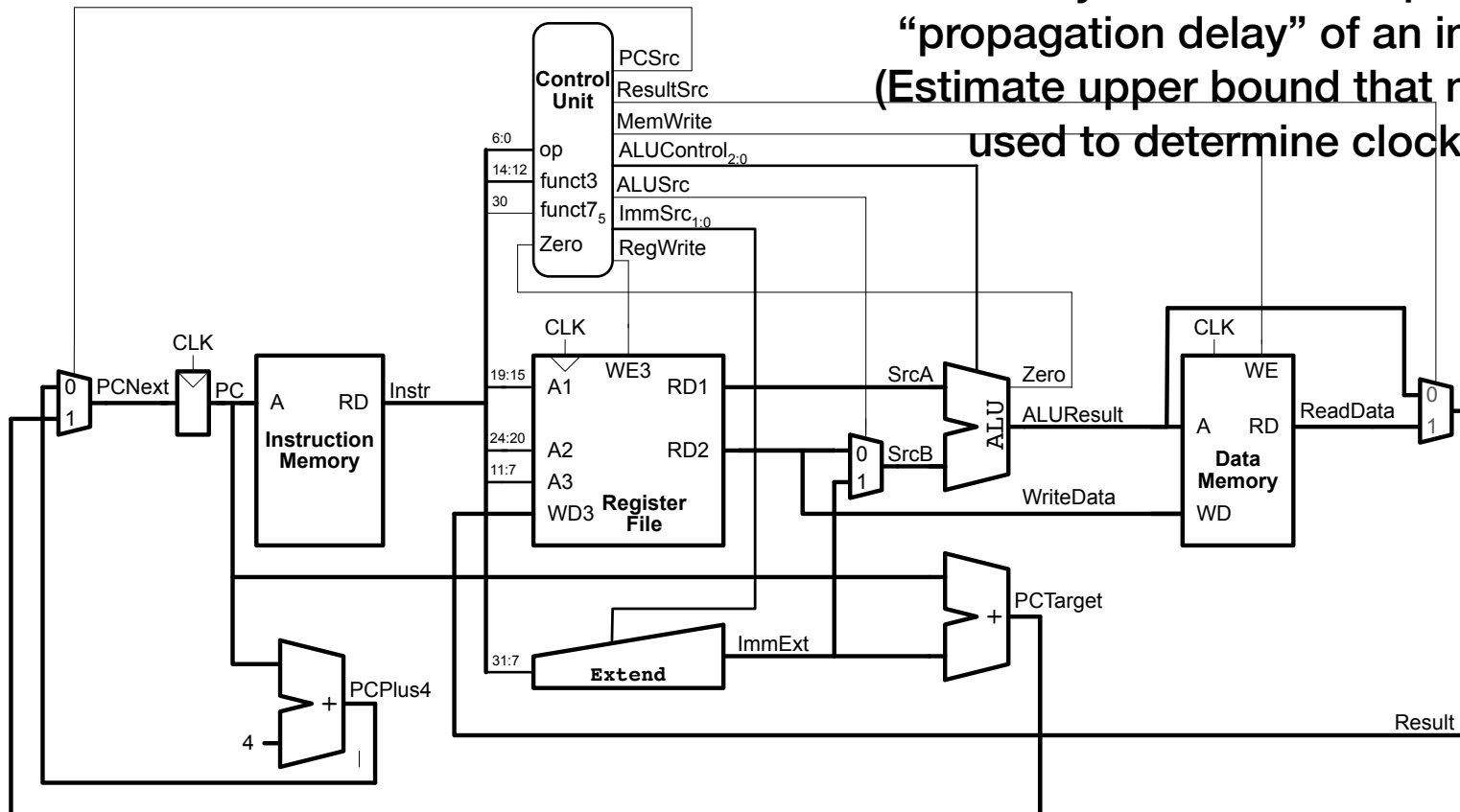
- Architecture: Blueprint of behavior based on assembly language
- Microarchitecture: Specific implementation(s)
  - Lots of variations possible with cost/performance tradeoffs
  - Ex: “Single Cycle” version vs. “Pipelined”
    - Single-Cycle: Each instruction is 1 (long) cycle
    - Multi-Cycle: Instructions take more than one cycle, but cycles are shorter and overall performance is better.

# Simple, Single-Cycle RISC-V Computer



# Simple, Single-Cycle RISC-V Computer

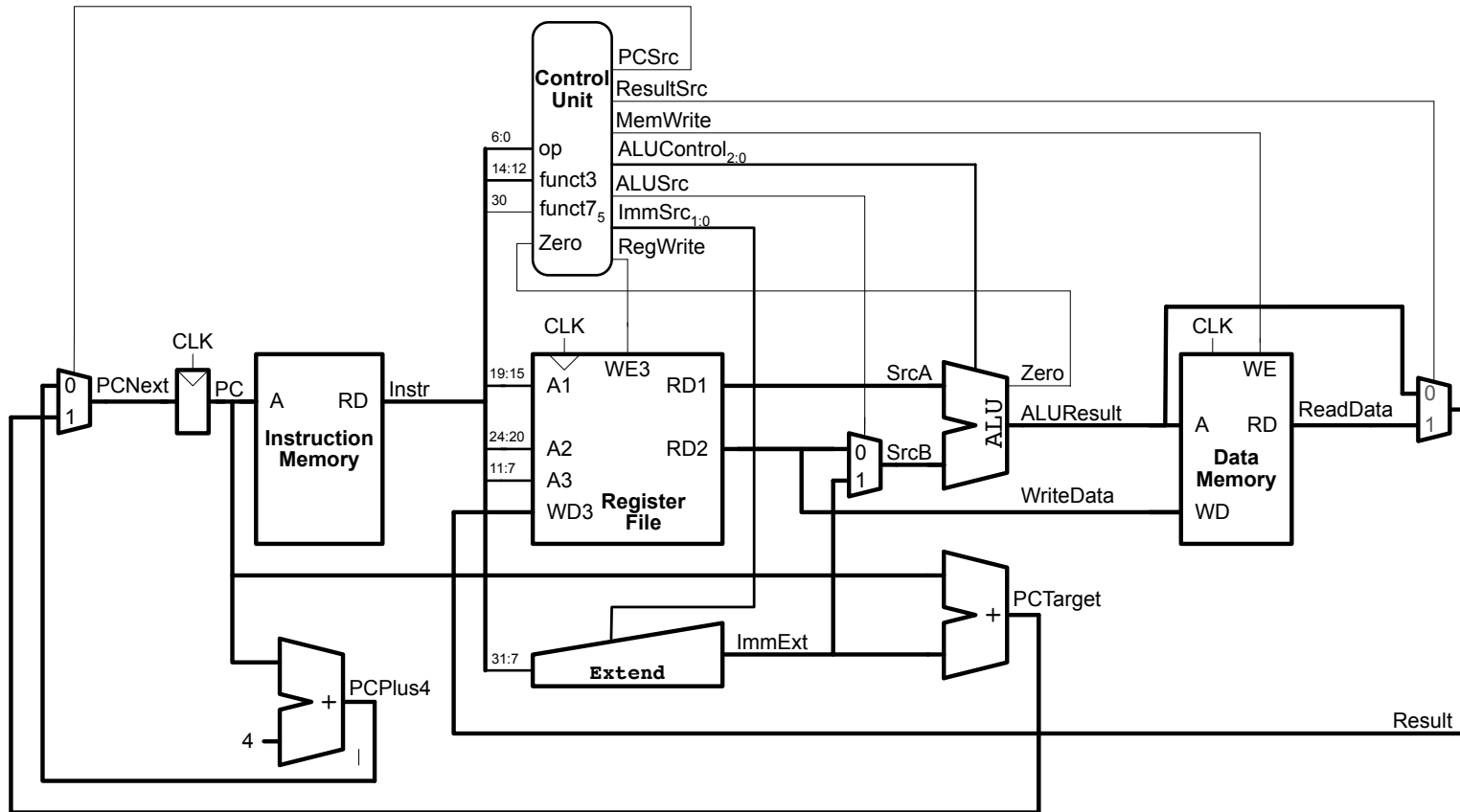
Identify items that are part of the “propagation delay” of an instruction.  
(Estimate upper bound that needs to be used to determine clock cycle)



# Simple, Single-Cycle RISC-V Computer

Describe behavior of all elements and any required control signals

add t0, t1, t1



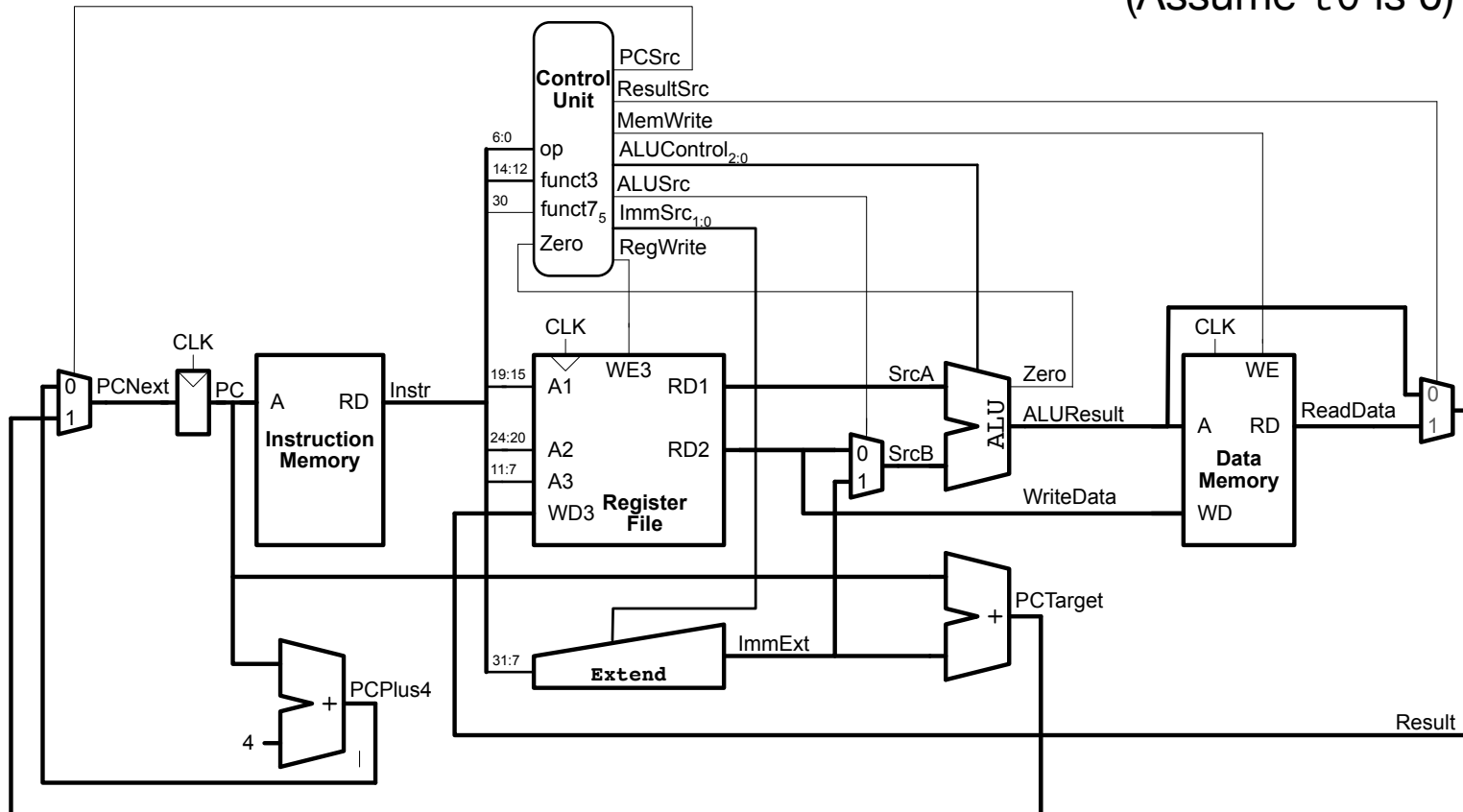


# Simple, Single-Cycle RISC-V Computer

Describe behavior of all elements and any

required control signals for  
 beq to, Zero, Loop

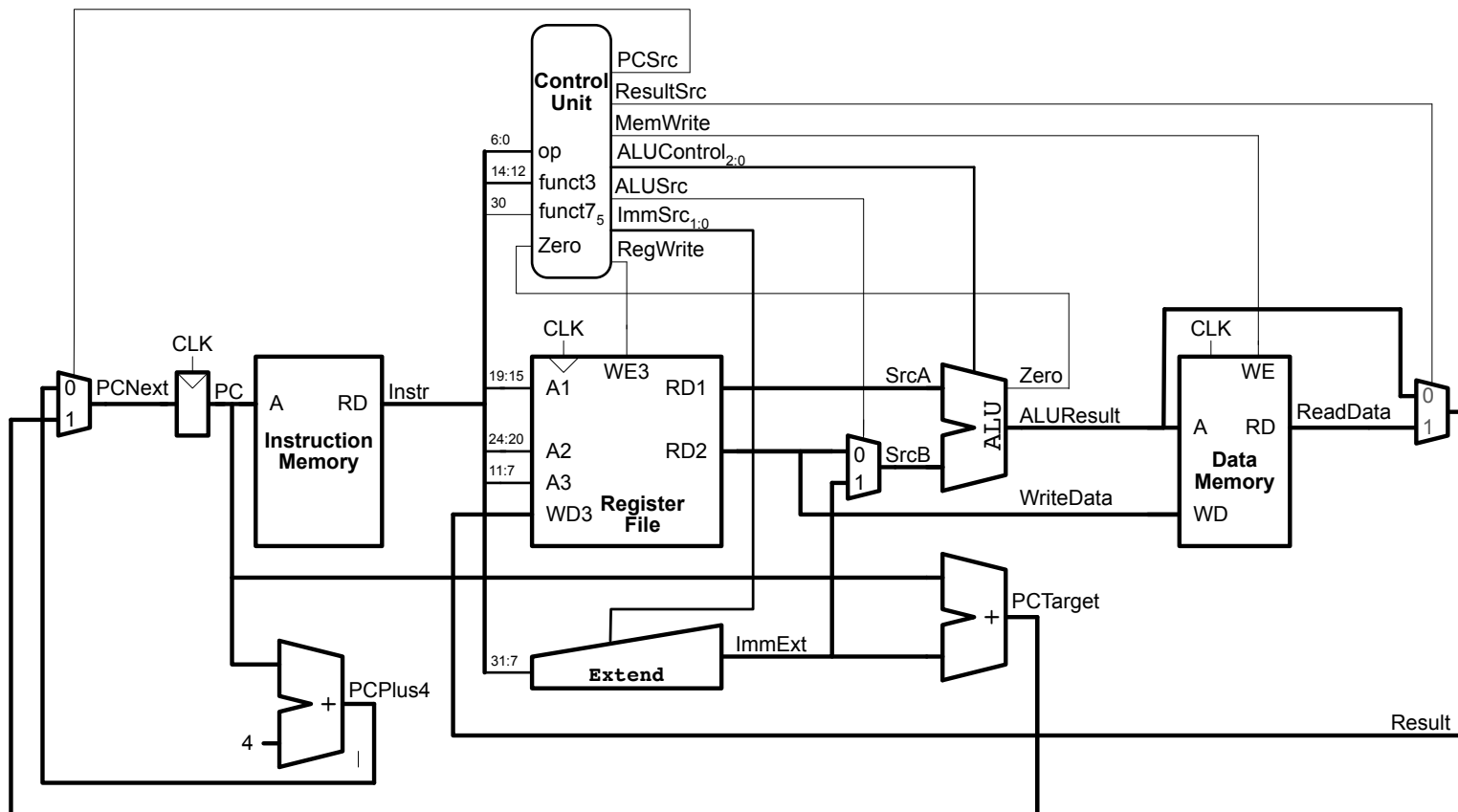
(Assume t0 is 0)



Describe behavior of all elements and any

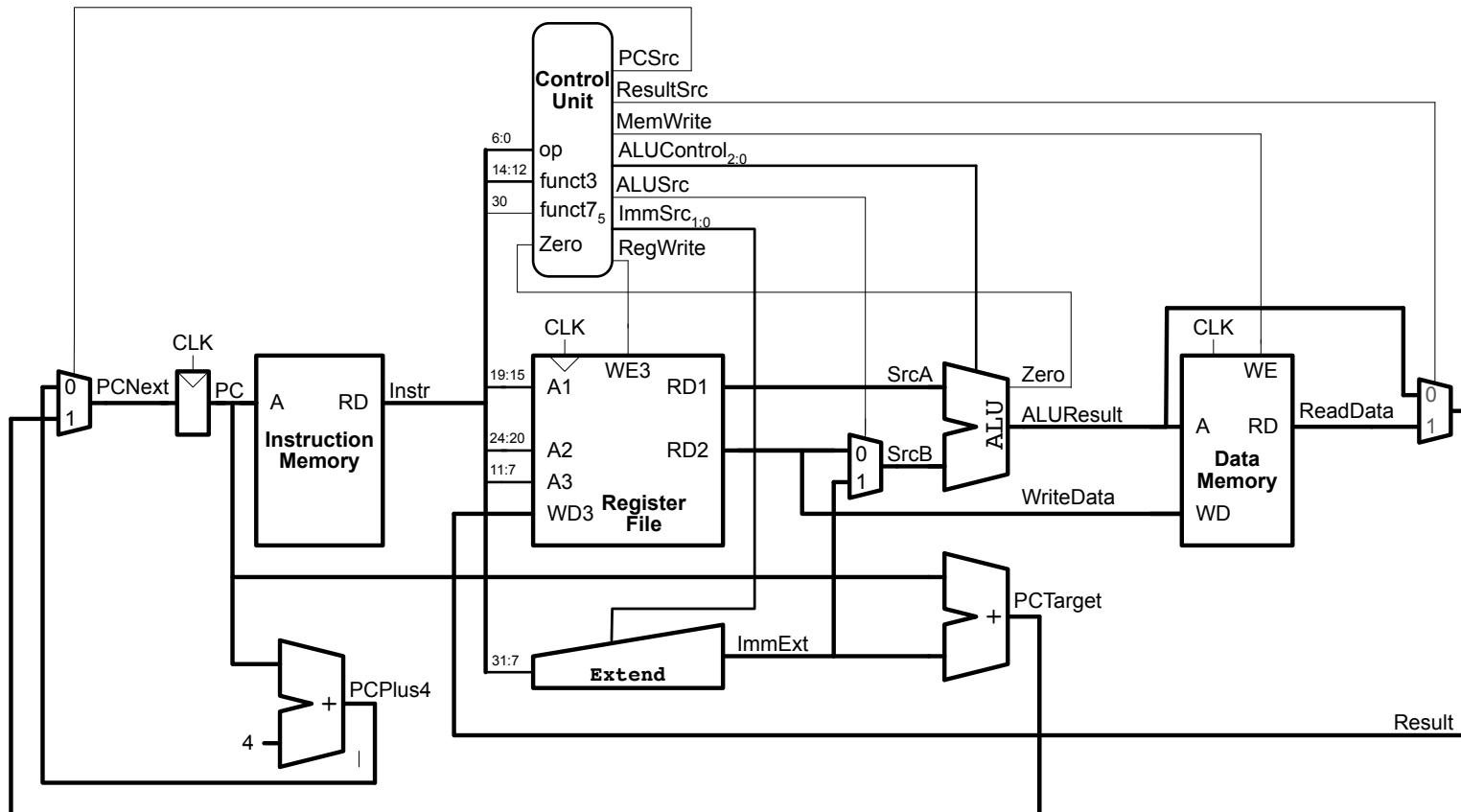
# Simple, Single-Cycle RISC-V Computer

required control signals for  
SW t0, 4(a0)



# Simple, Single-Cycle RISC-V Computer

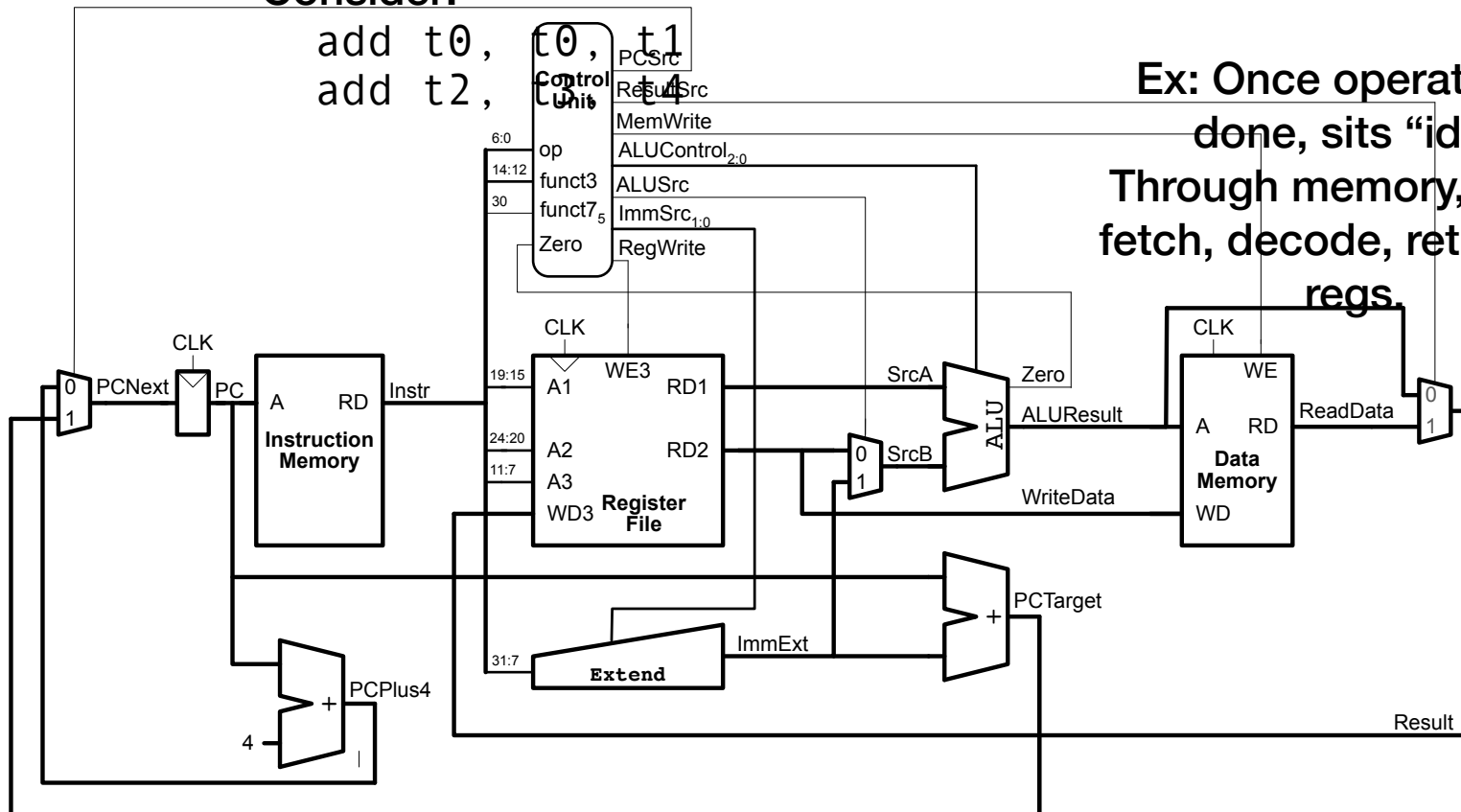
Describe behavior of all elements and any required control signals for  
 LW t0, 4(a0)



# Simple, Single-Cycle: Inefficient!

Consider:

```
add t0, t0, t1  
add t2, t2, t4
```



# Laundry

- Laundry machines
  - Washer takes 30 minutes
  - Dryer takes 1 hour (ugh)
- How long does it take to do 1 load of laundry all the way through?
- What about 2 loads?
- What's the approx. average for 50 loads of laundry?

**Pipelining: A sequence of operations  
(and overlapping a single “instruction” (load  
of laundry))**

# Next Time

- Studio
  - Bring full kits!