

CSE 2600

Intro. To Digital Logic & Computer Design

Bill Siever
&
Michael Hall

Review Reading (from Mem. to Reg): Load

- Mnemonic: load word (lw)
- Format: lw t1, 8(s0)
lw destination, offset(base)
- Address calculation: RAM index = add base address (s0) to the offset (8) = (s0 + 8)
- Result: t1 holds the data value at address (s0 + 8)
I.e. t1 = RAM[s0+8]
- In terms of “register” array:
REG[t1] = RAM[REG[s0]+8]
REG[6] = RAM[REG[8]+8]
- Units: Memory is usually “byte addressable” and words are 4 bytes.
So data is actually retrieved from RAM[REG[s0]+8] to RAM[REG[s0]+11]

Reading (from Reg to Mem.): Store

- Mnemonic: store word (sw)
- Format: sw t1, 8(s0)
sw source, offset(base)
- Address calculation: RAM index = add base address (s0) to the offset (8) = (s0 + 8)
- Result: t1 holds the data value at address (s0 + 8)
I.e. $\text{RAM}[\text{s0}+8] = \text{t1}$
- In terms of “register” array:
 $\text{RAM}[\text{REG}[\text{s0}]+8]] = \text{REG}[\text{t1}]$
 $\text{RAM}[\text{REG}[8]+8]] = \text{REG}[6]$
- ***NOTE: Stores are the one case where the “thing on the left” is not the destination!***

Studio 6B: More Assembly (Functions & Memory)

Chapter 6 & 7

Architectures: RISC-V

- “Architecture”: Programmer’s view of CPU
- Fundamental data size: 32-bit “word”. CPU/ALU designed for 32-bit operations (Multiple 32-bit operations can be done to do larger operations)
- Memories
 - RAM: Big array of numbers; Uses 32-bit addresses
 - Registers: Array with 32, 32-bit values. Special names correspond to intended uses (Ex: a-registers, like a0, are for “arguments” to functions)
 - Instructions: Also 32-bit values; May be stored in RAM or separate memory

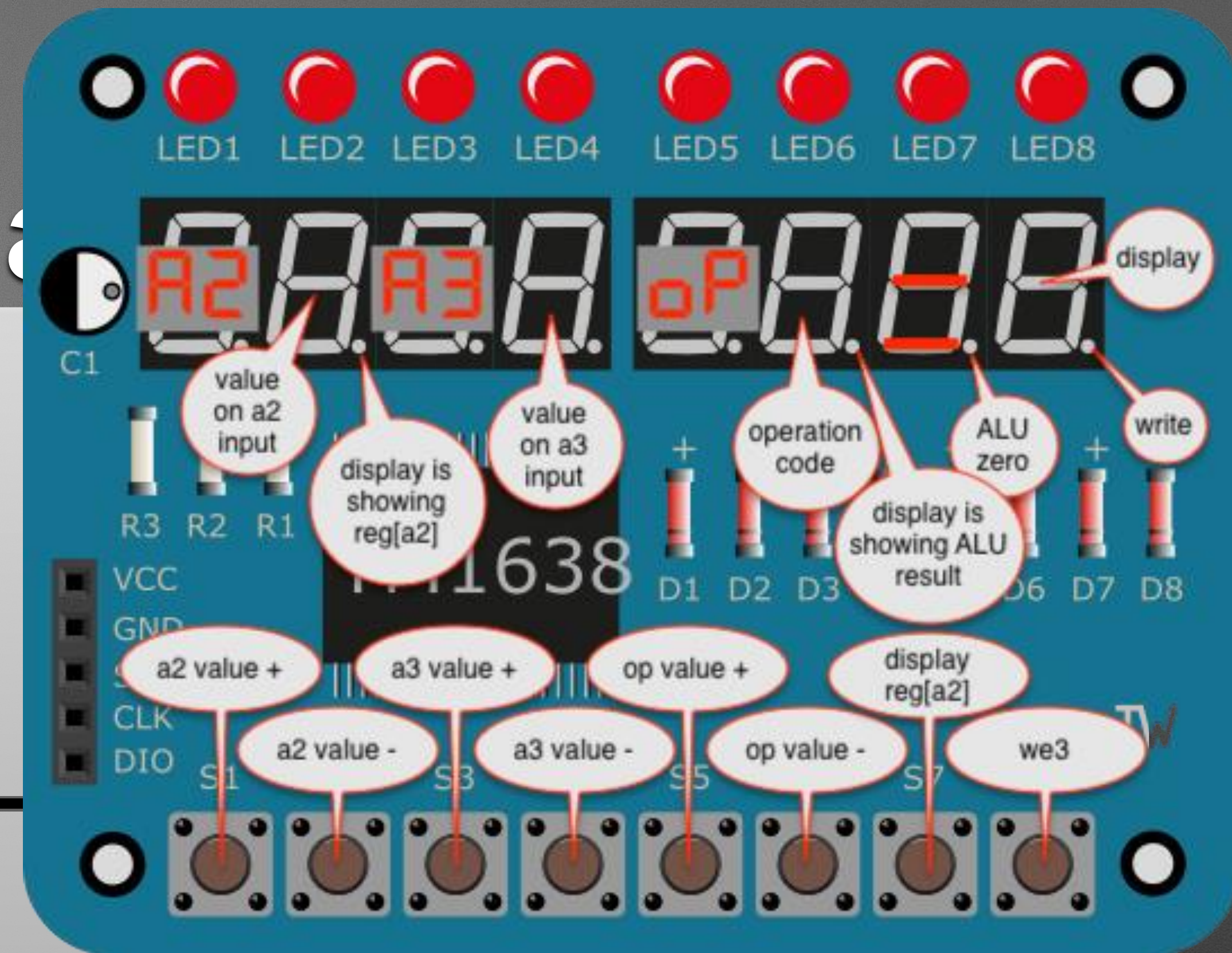
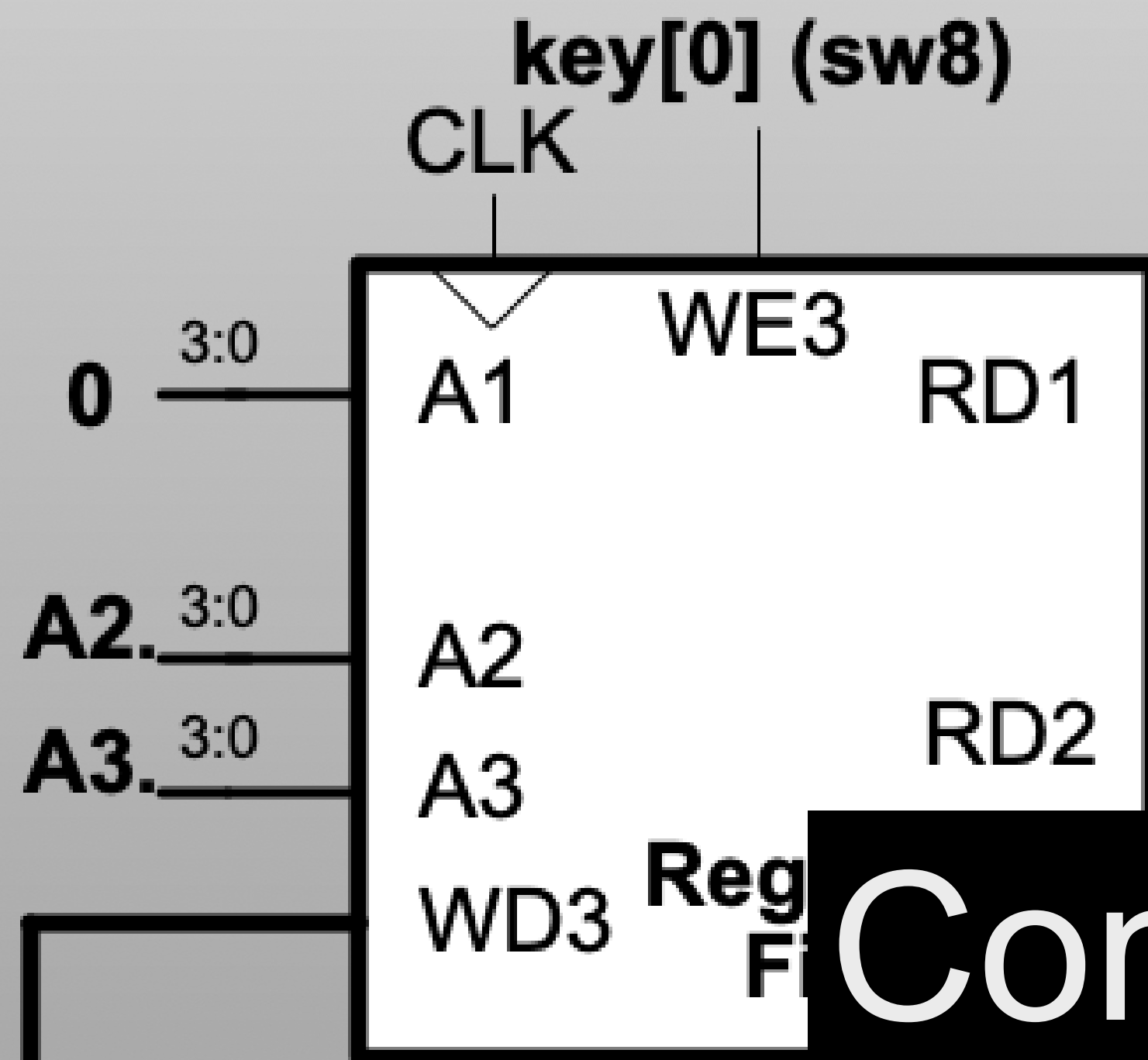
Architectures: RISC-V

- Machine codes
 - “[Substitution code](#)”: Numbers represent concepts
 - RISC-V “[Instruction Set Architecture](#)” (ISA):
 - Formats are about data locations
 (“addressing” / location of information needed)

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.

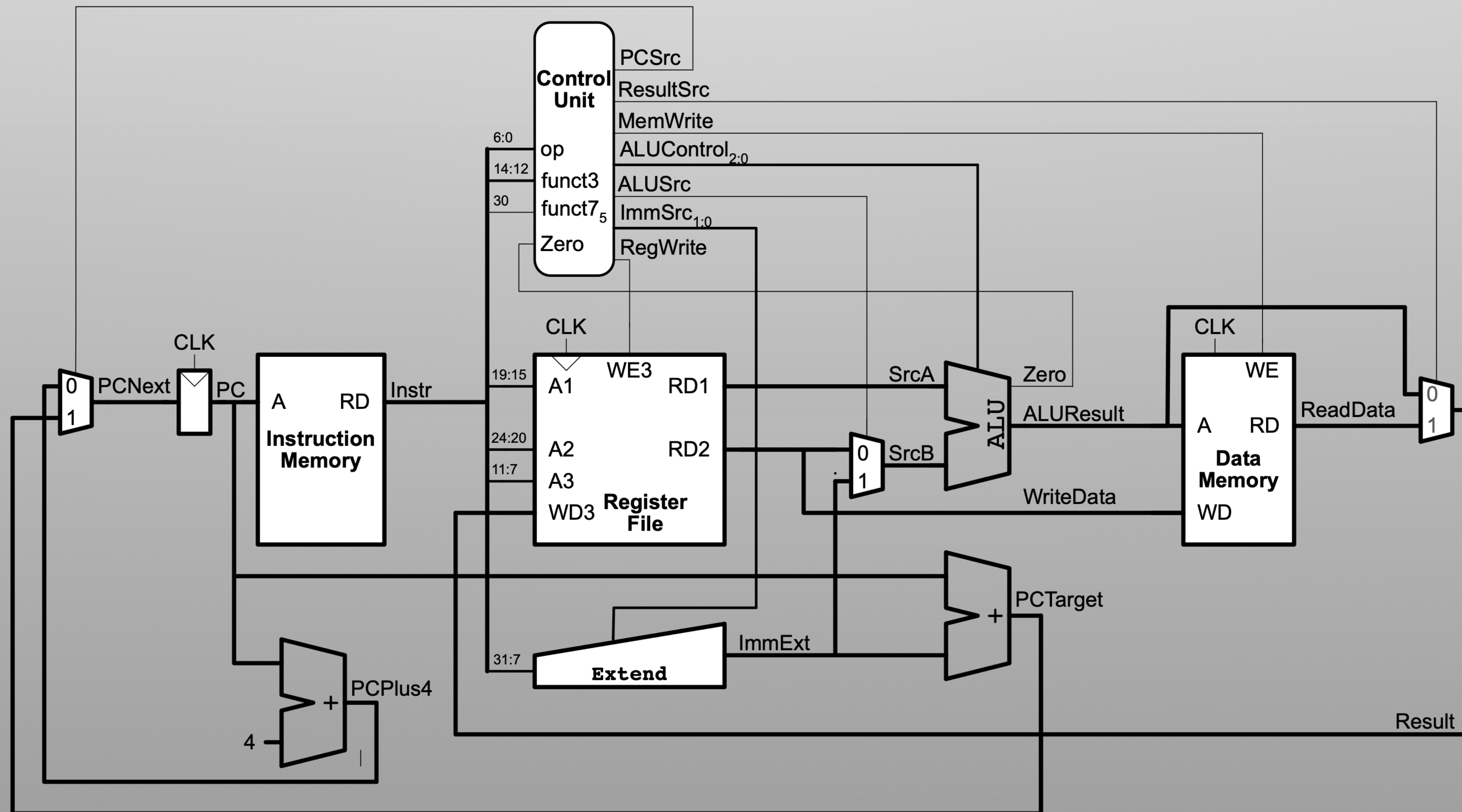
Homework 5A: Partia



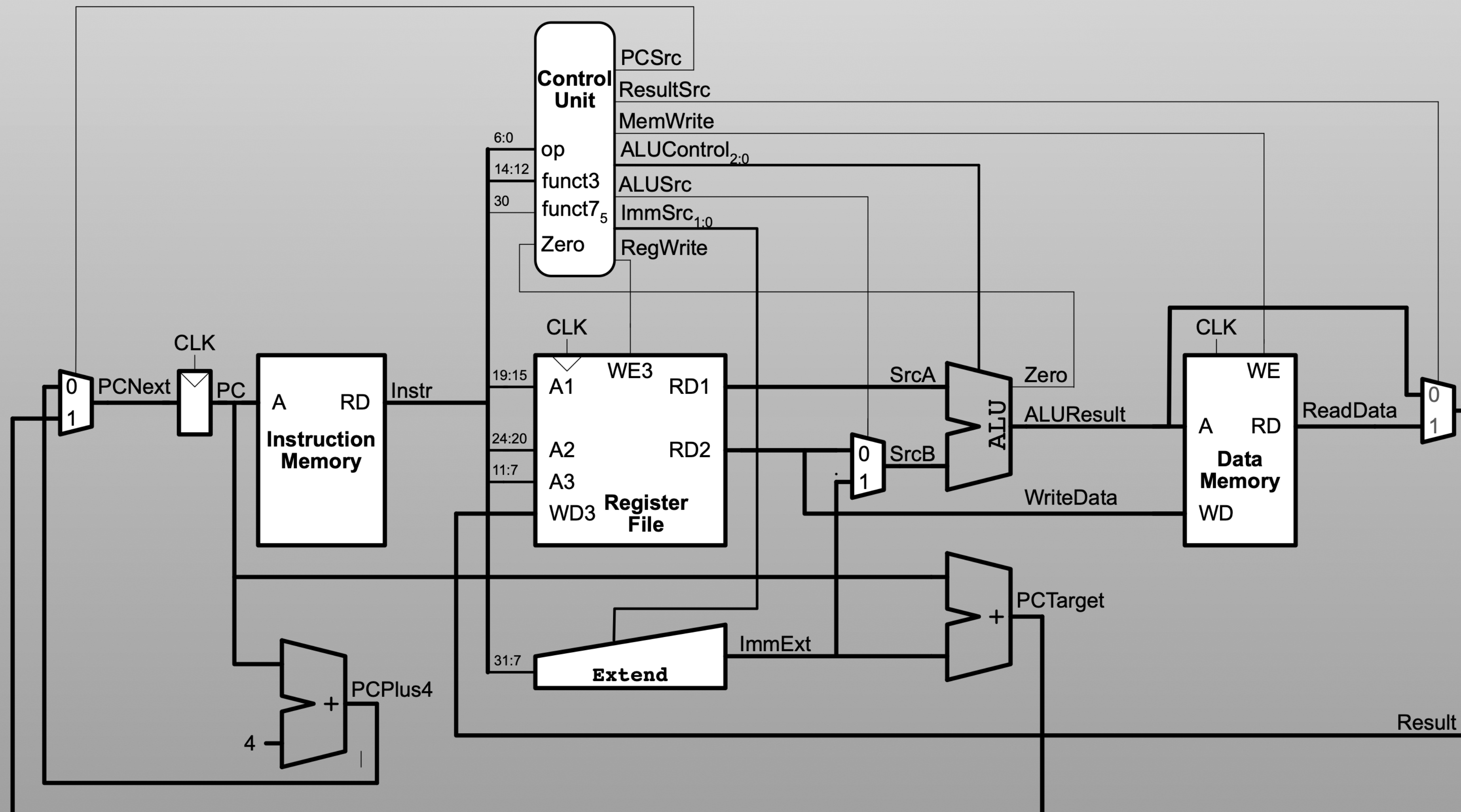
Control: You

Data path: Hardware

Simple (Single-Cycle) RISC-V Computer



Simple (Single-Cycle) Control vs. Datapath



The Problem

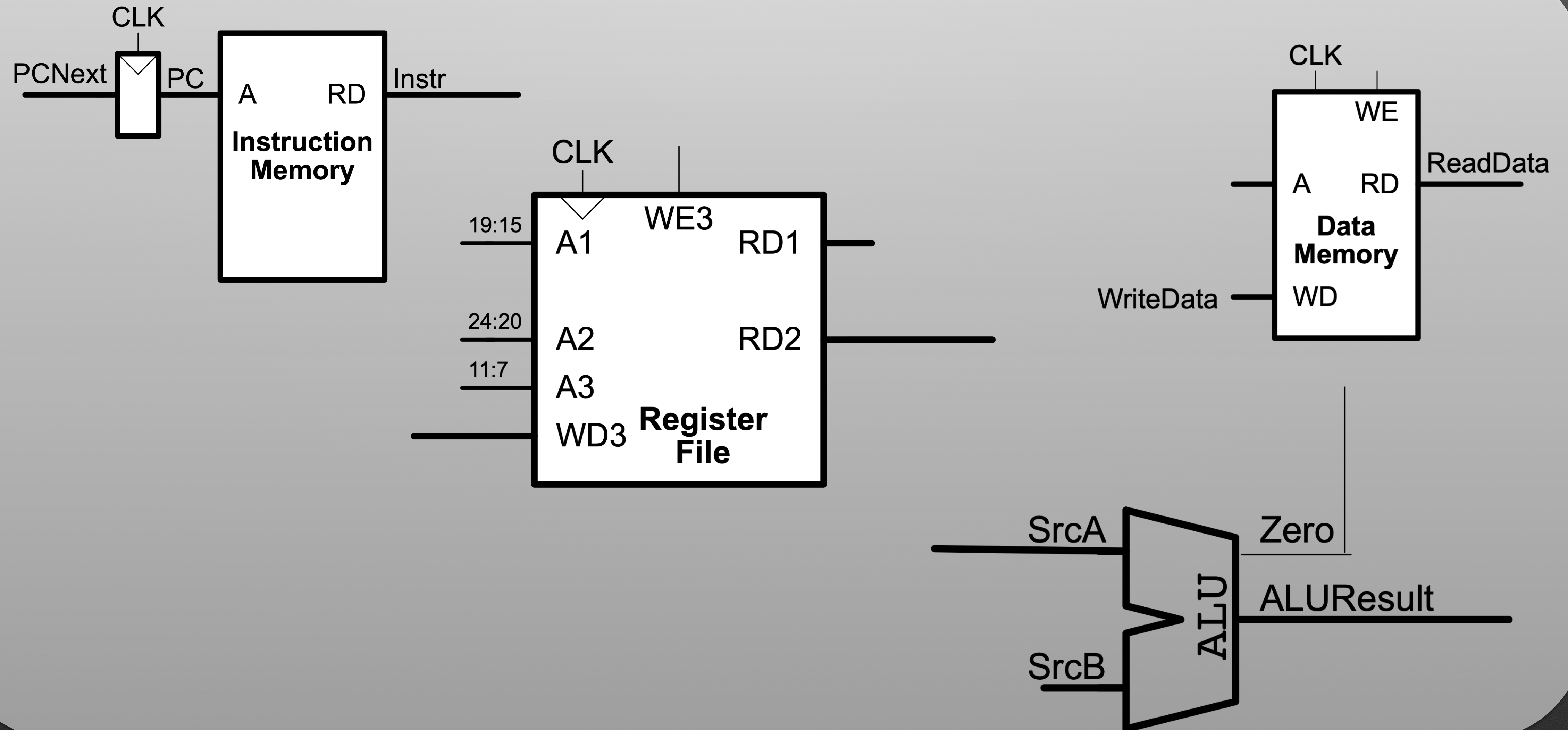
Find x such that $2^x = 128$

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

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

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

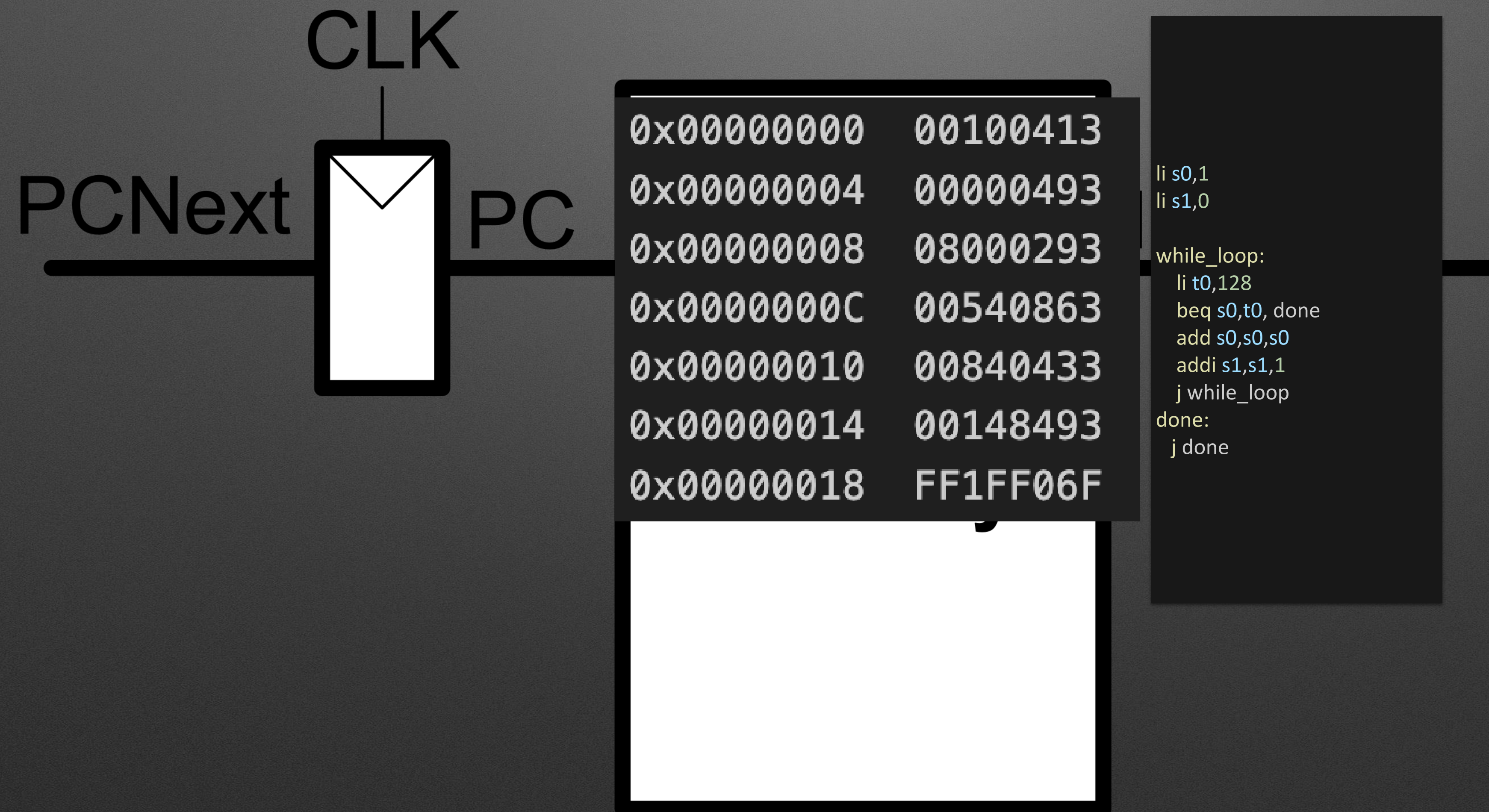
Behavior: Parts of CPU Model



Behavior: Parts of CPU Model



Behavior: Parts of CPU Model



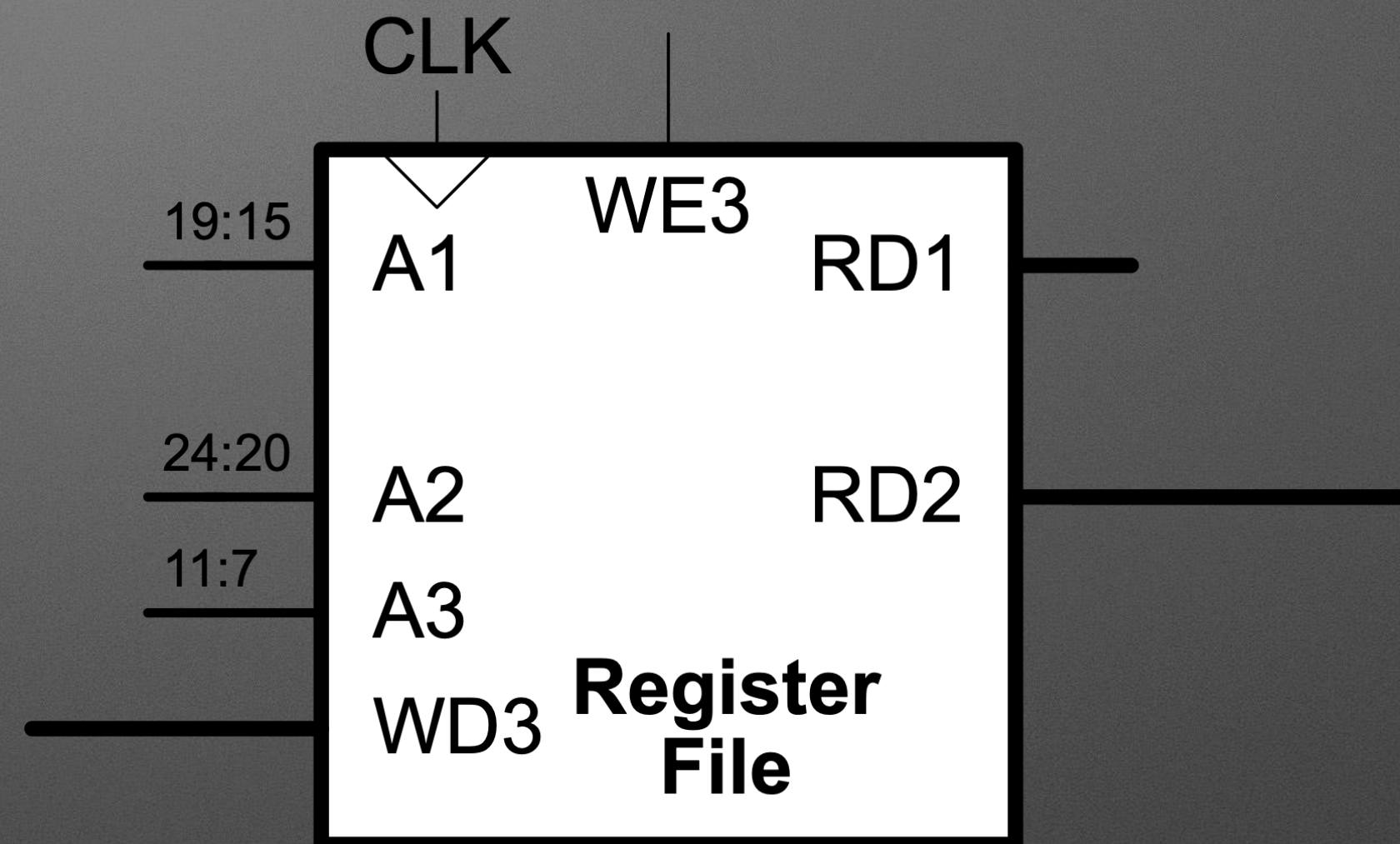
Behavior: Parts of CPU Model

PC

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:
  j done
```



Behavior: Parts of CPU Model

PC

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:
  j done
```

CLK

19:15

Index

Name

Value

x0

zero

x1

ra

24:20

...

x5

t0

11:7

...

x8

s0

x9

s1

Behavior: Parts of CPU Model

PC

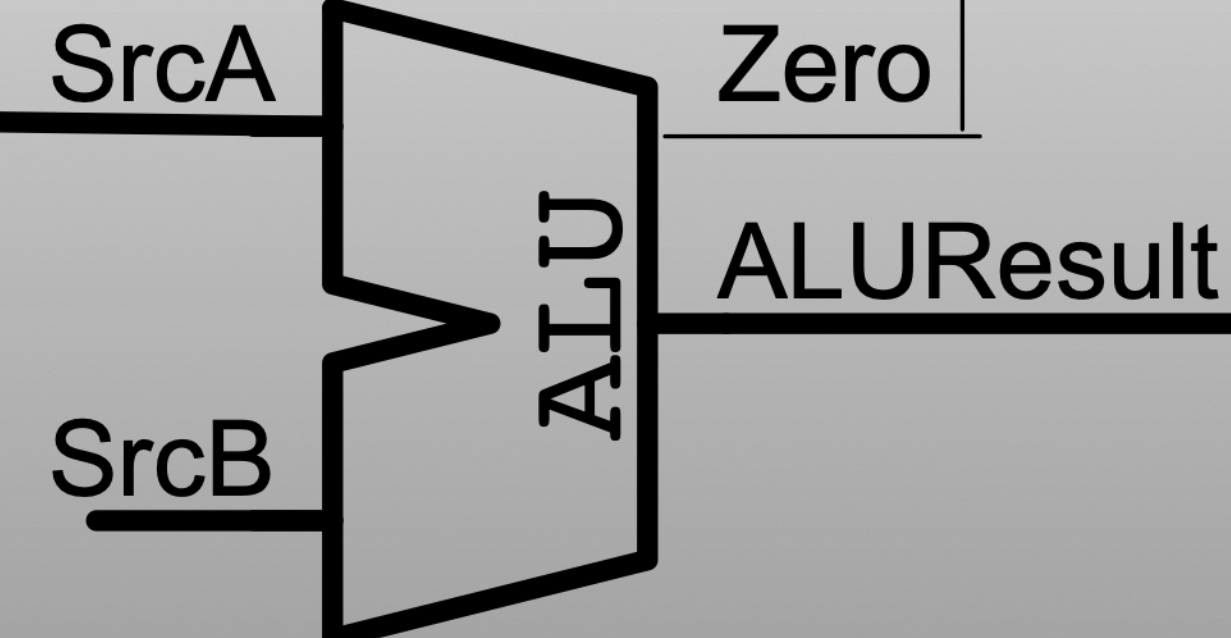
0x00000000	00100413
0x00000004	00000493
0x00000008	08000293
0x0000000C	00540863
0x00000010	00840433
0x00000014	00148493
0x00000018	FF1FF06F

```
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:
  j done
```

CLK

	Index	Name	Value
19:15	x0	zero	
	x1	ra	
24:20		...	
	x5	t0	
11:7		...	
	x8	s0	
	x9	s1	



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

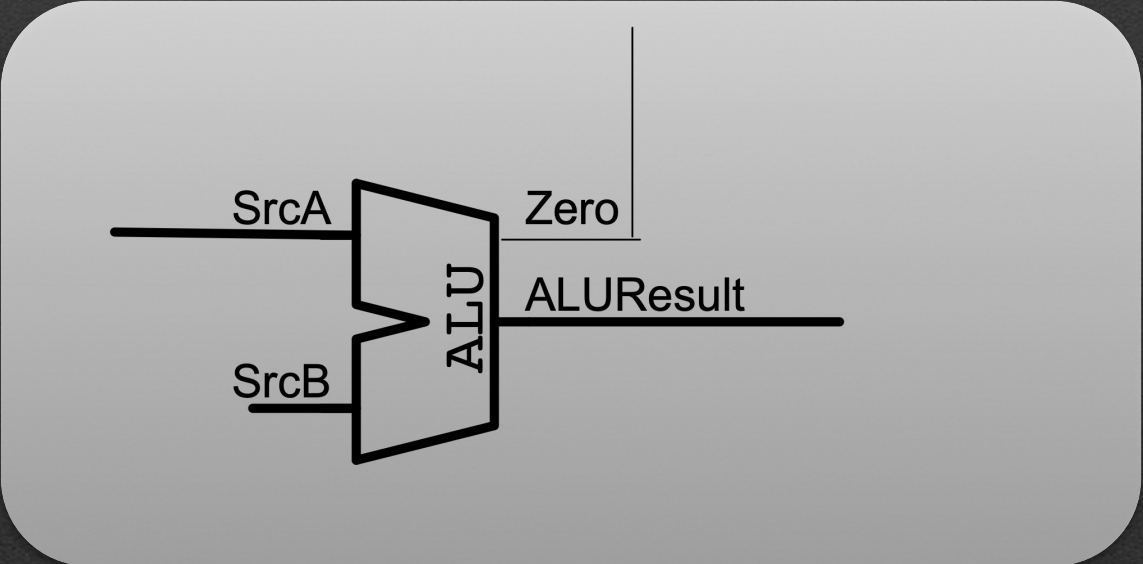


```
li s0,1  
li s1,0  
while_loop: li t0,128  
             beq s0,t0, done  
             add s0,s0,s0  
             addi s1,s1,1  
             j while_loop  
done: i done
```

19:15

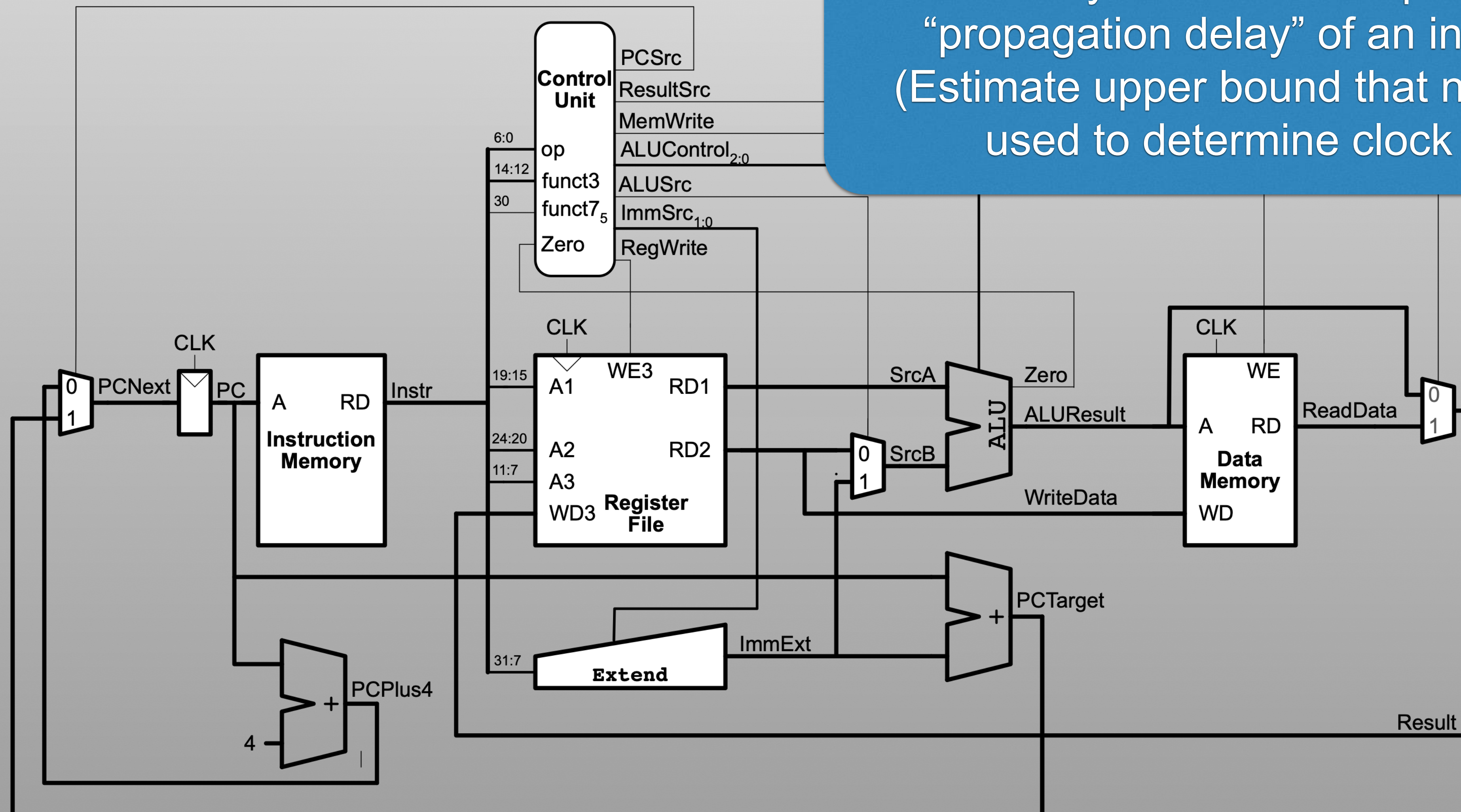
Index		Name	Value
x0		zero	
x1		ra	
		...	
x5		t0	
		...	
x8		s0	
x9		s1	

Model



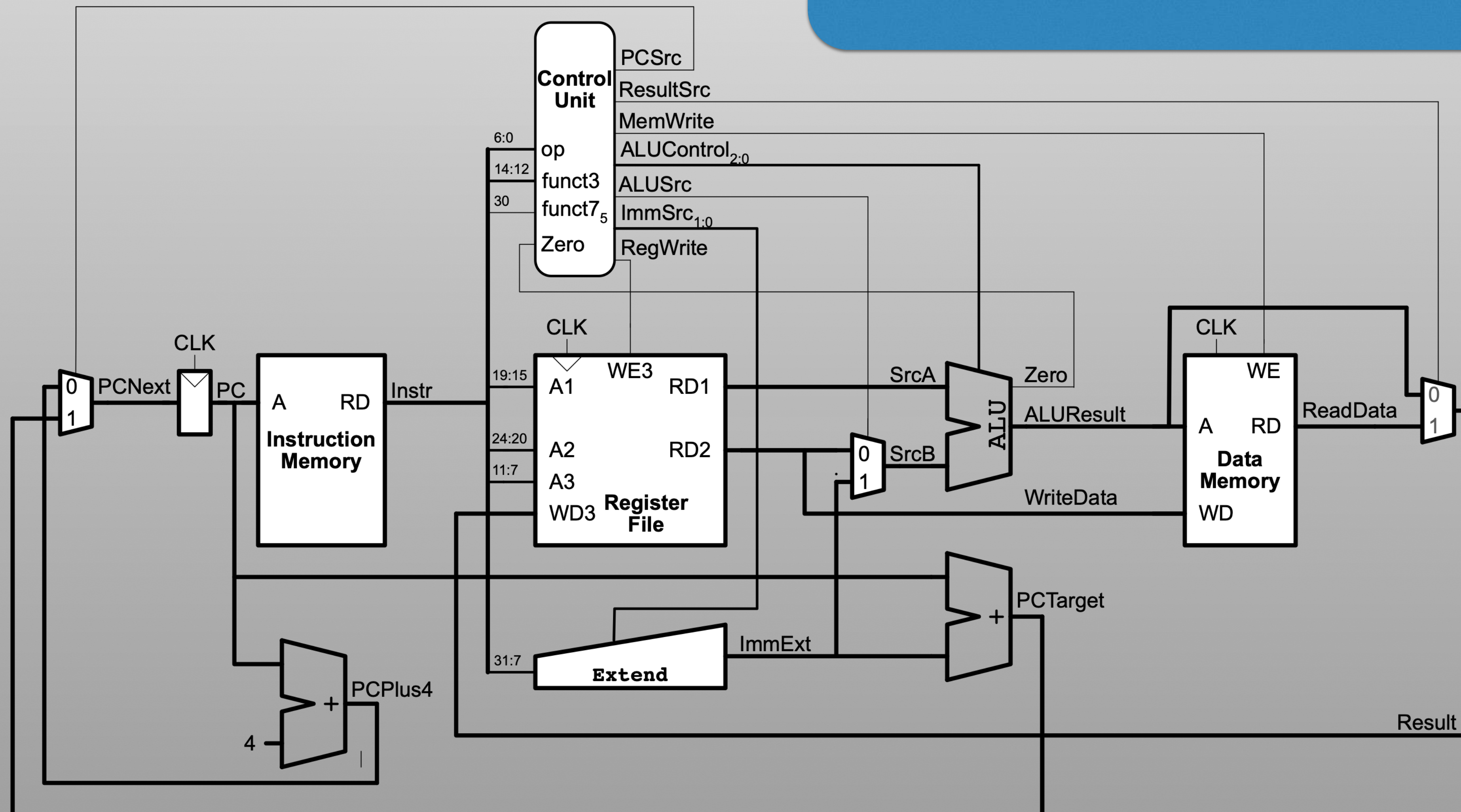
Simple, Single-Cycle RISC-V Computer

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



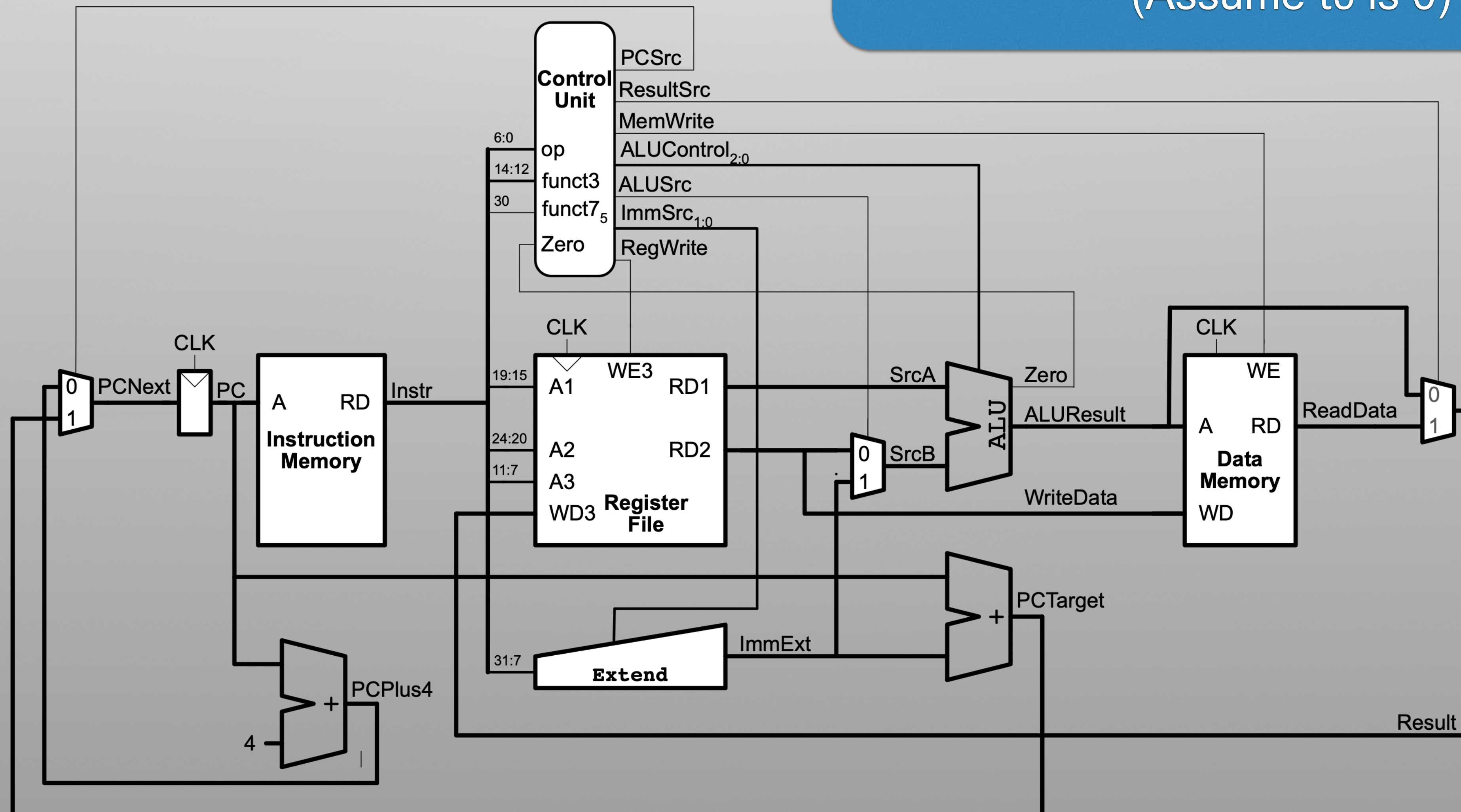
Simple, Single-Cycle

Describe behavior of all elements and any required control signals for
add t0,t1,t1



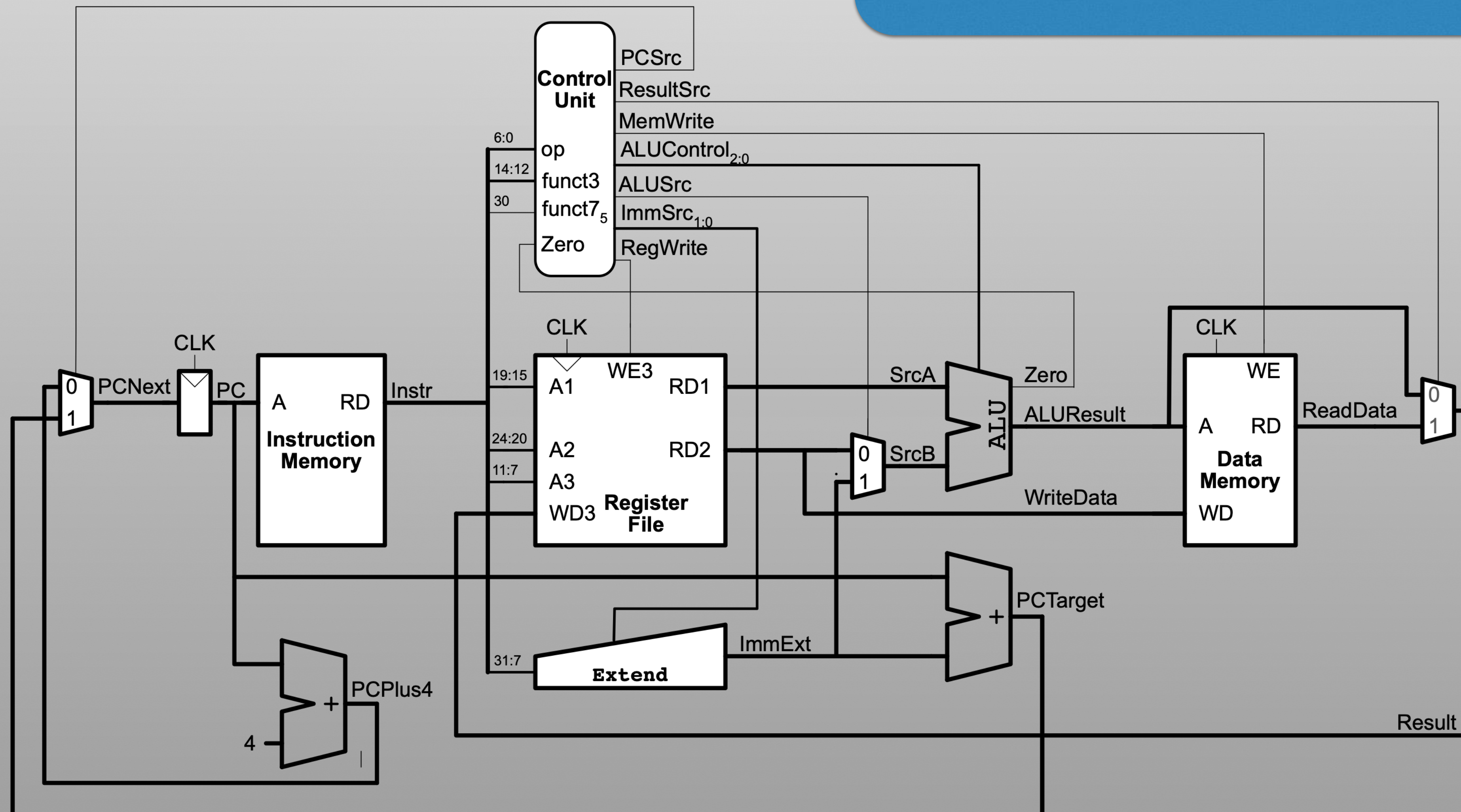
Simple, Single-Cycle

Describe behavior of all elements and any required control signals for
beq t0,zero,loop
(Assume t0 is 0)



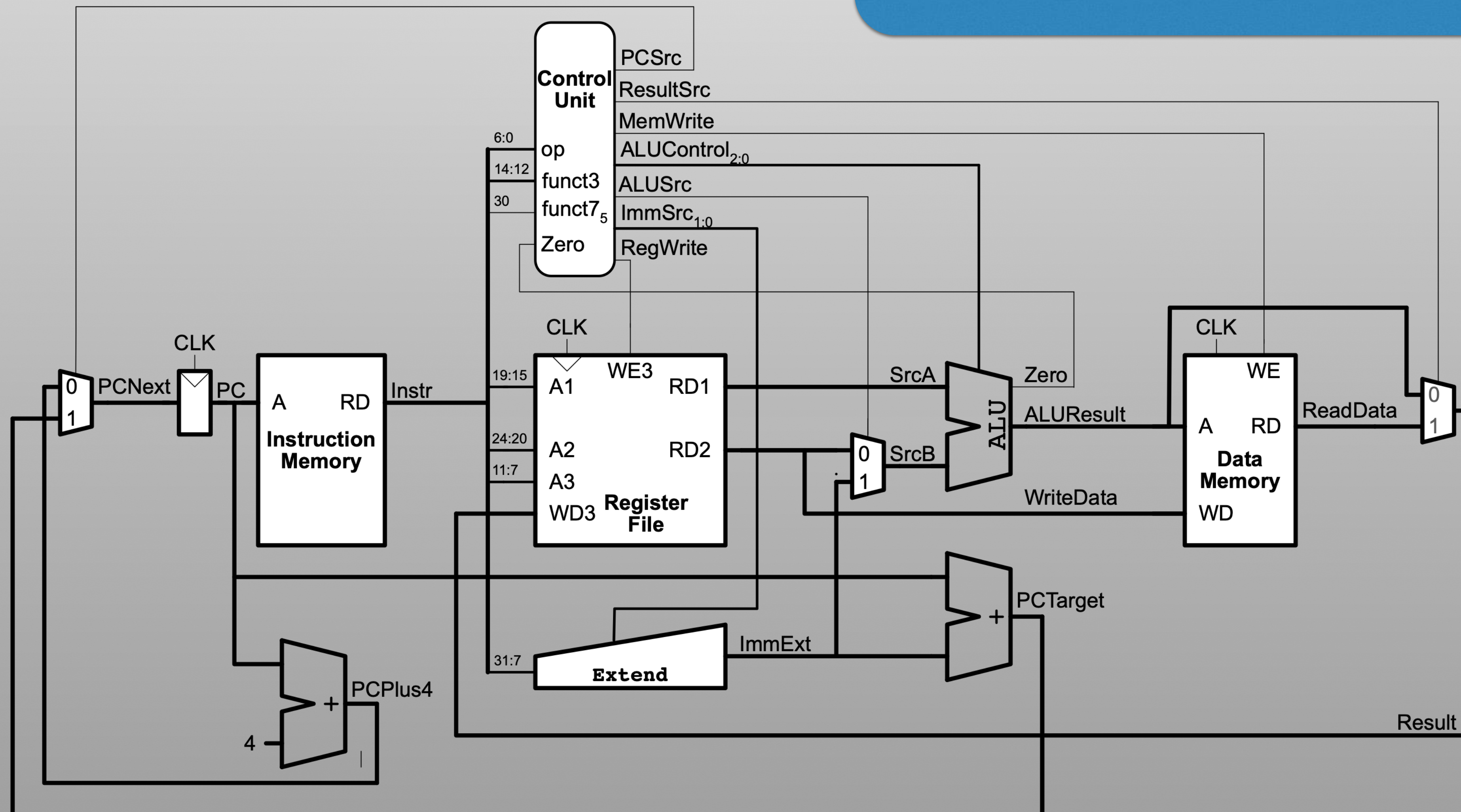
Simple, Single-Cycle R

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



Simple, Single-Cycle R

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



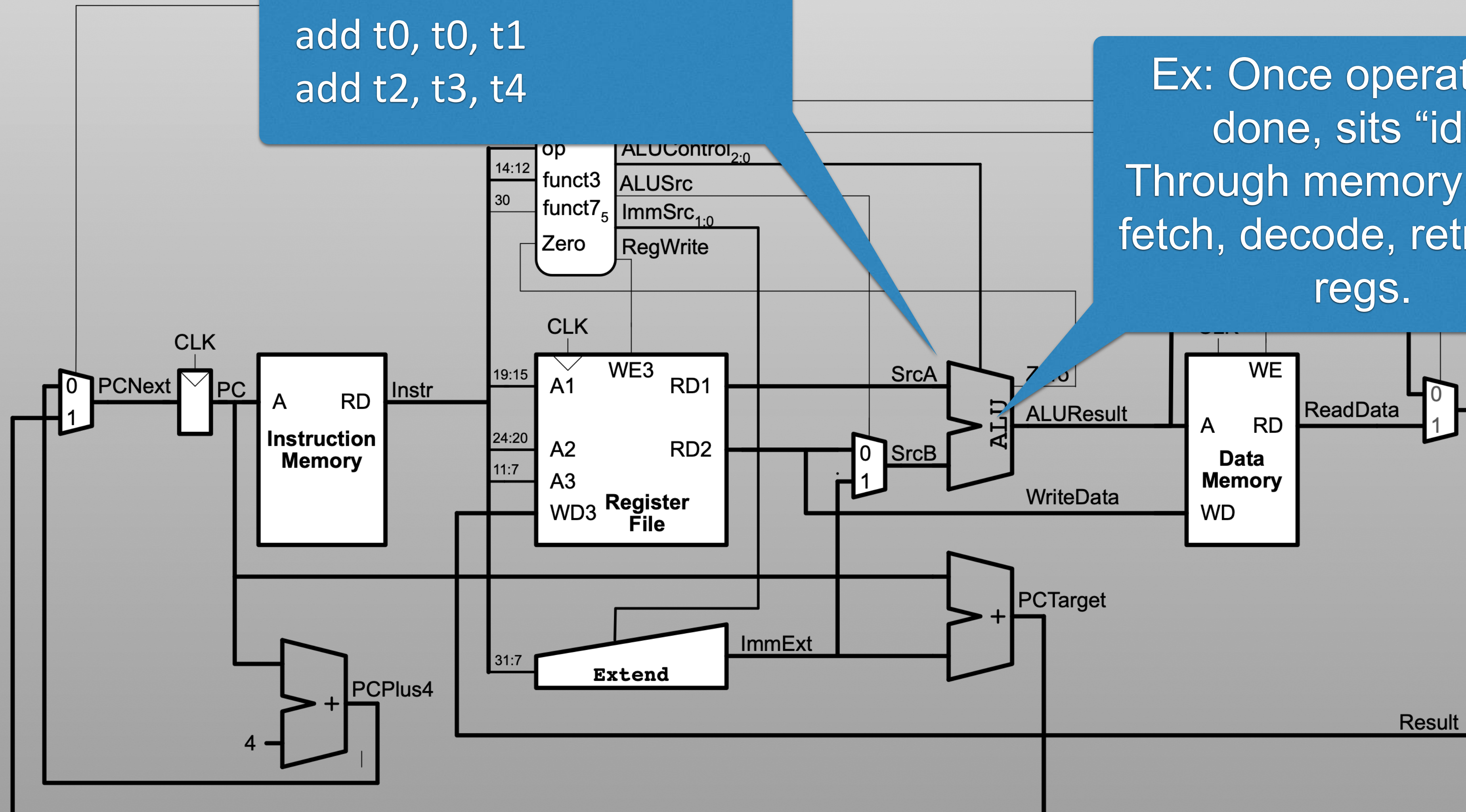
Simple Single-Cycle: Inefficient!

Consider:

add t0, t0, t1

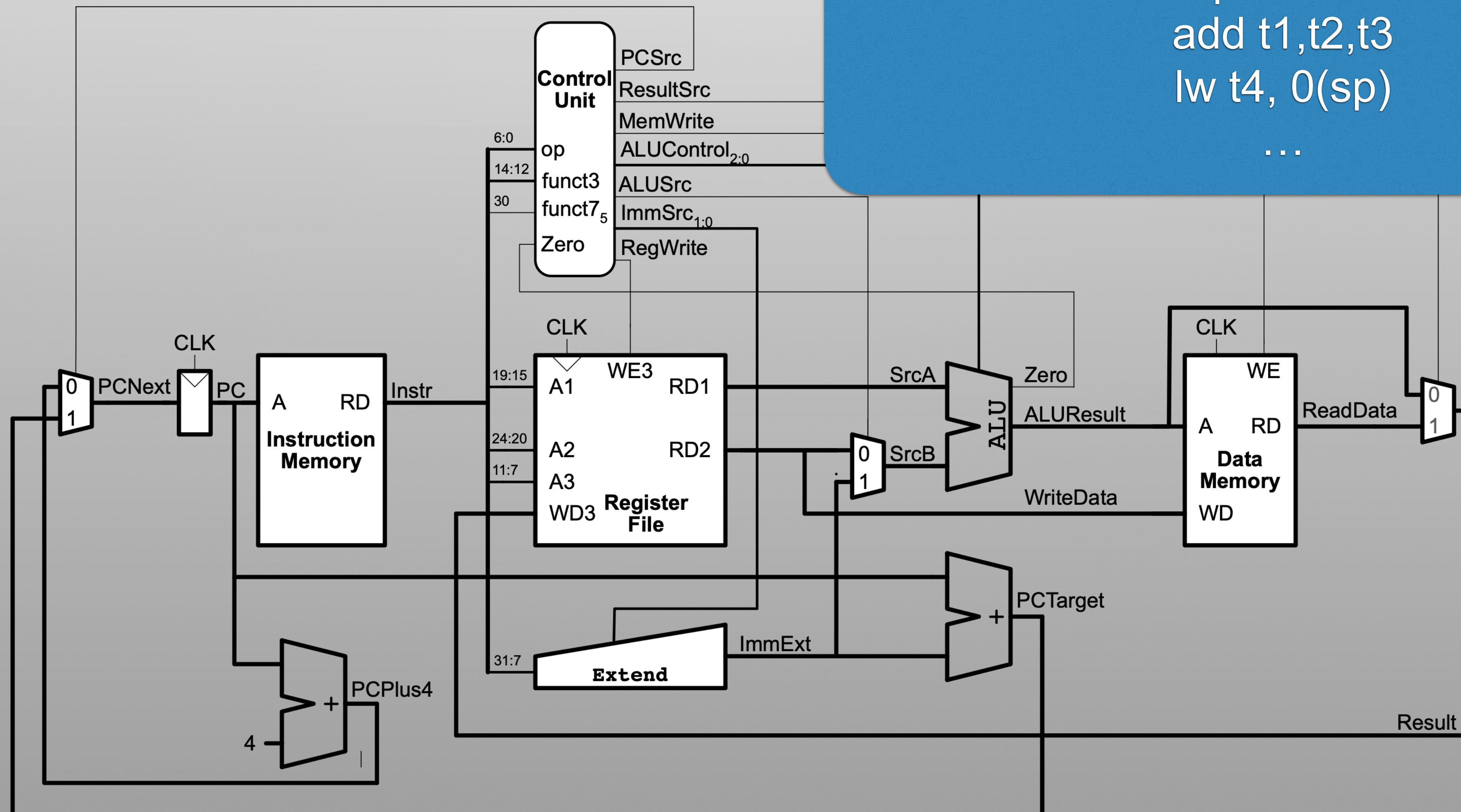
add t2, t3, t4

Ex: Once operation is done, sits "idle" Through memory, clock, fetch, decode, retrieve of regs.



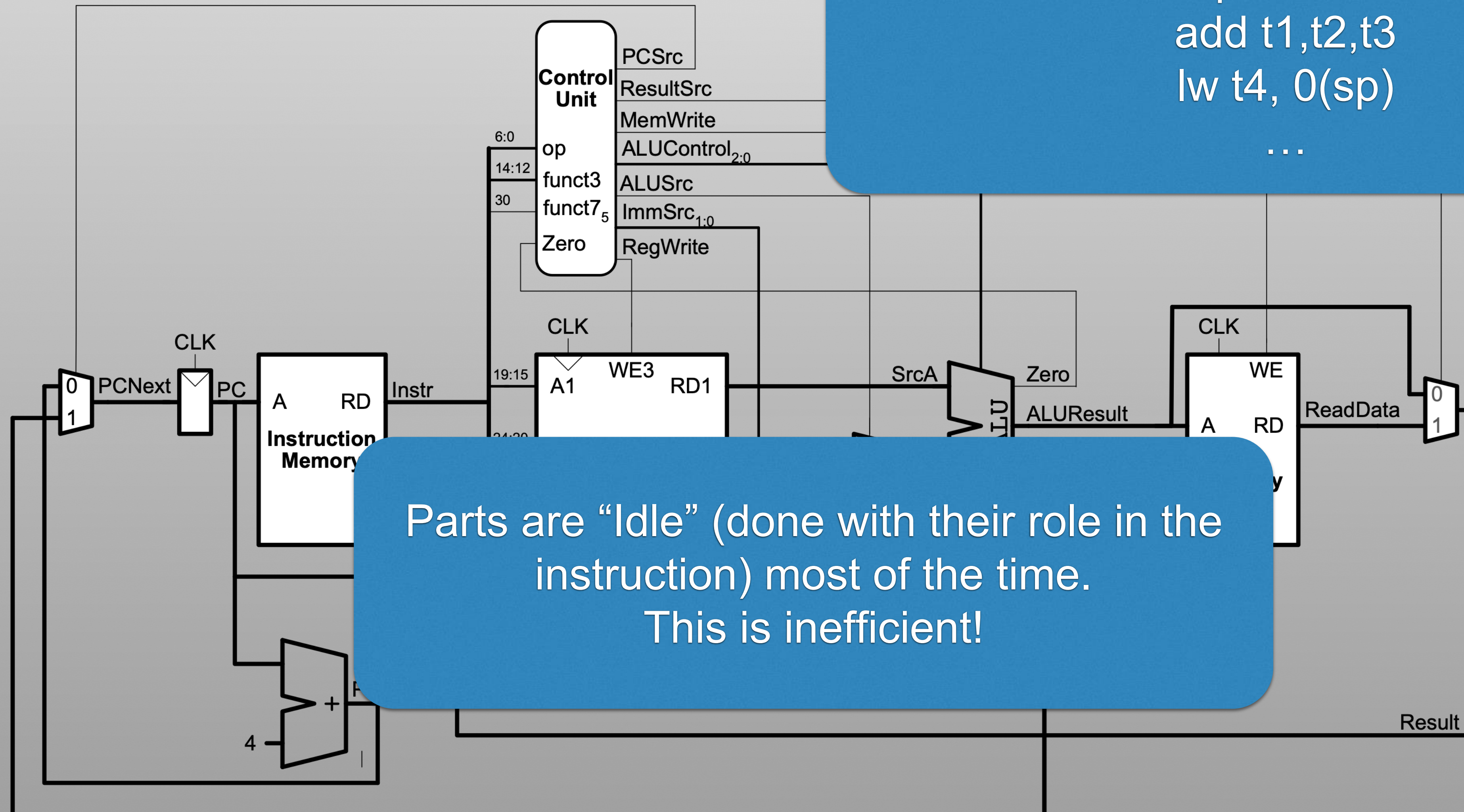
Simple, Single-Cycle RISC-V Computer

Consider a sequence of instructions:
add t1,t2,t3
lw t4, 0(sp)
...



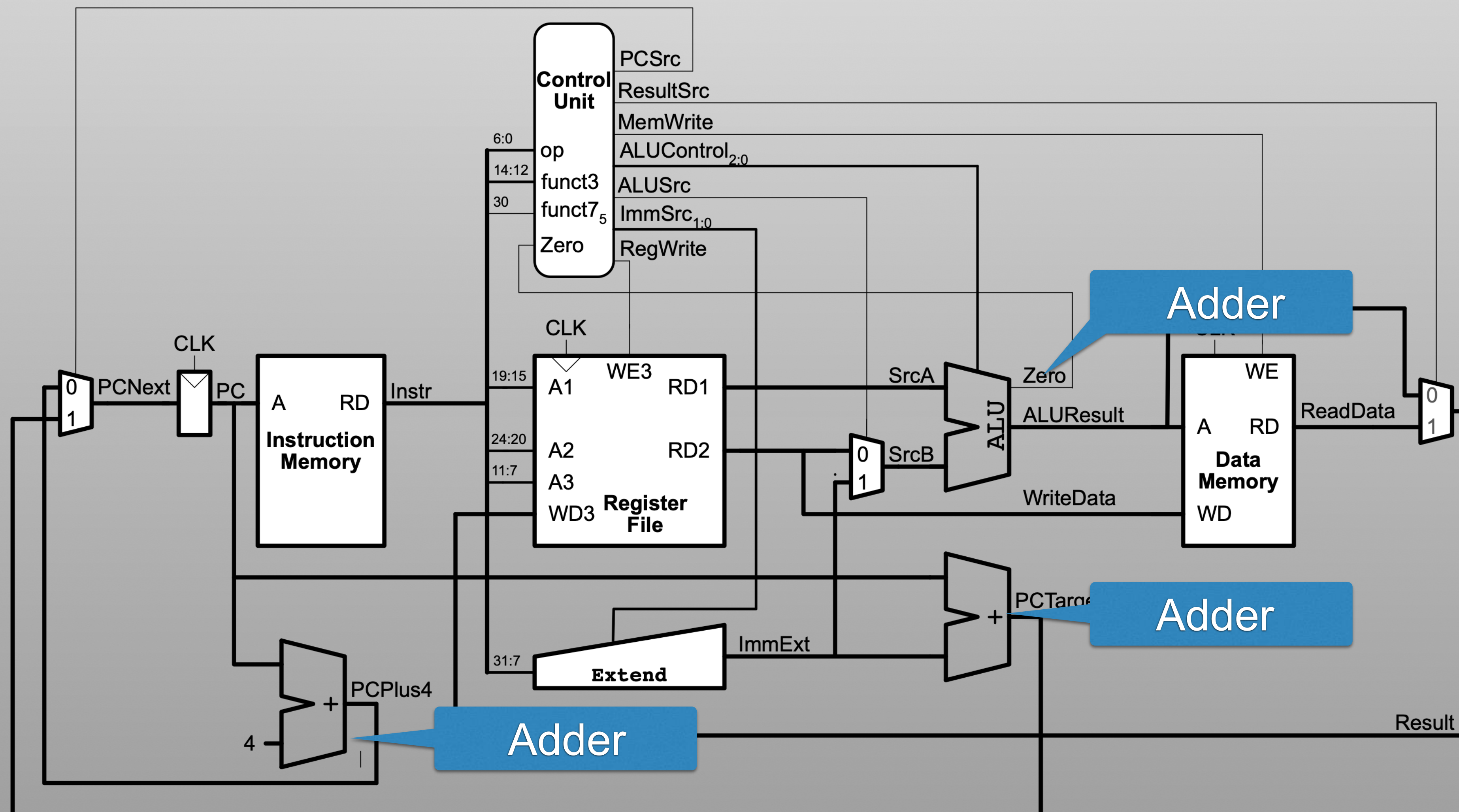
Simple, Single-Cycle RISC-V Computer

Consider a sequence of instructions:
add t1,t2,t3
lw t4, 0(sp)
...

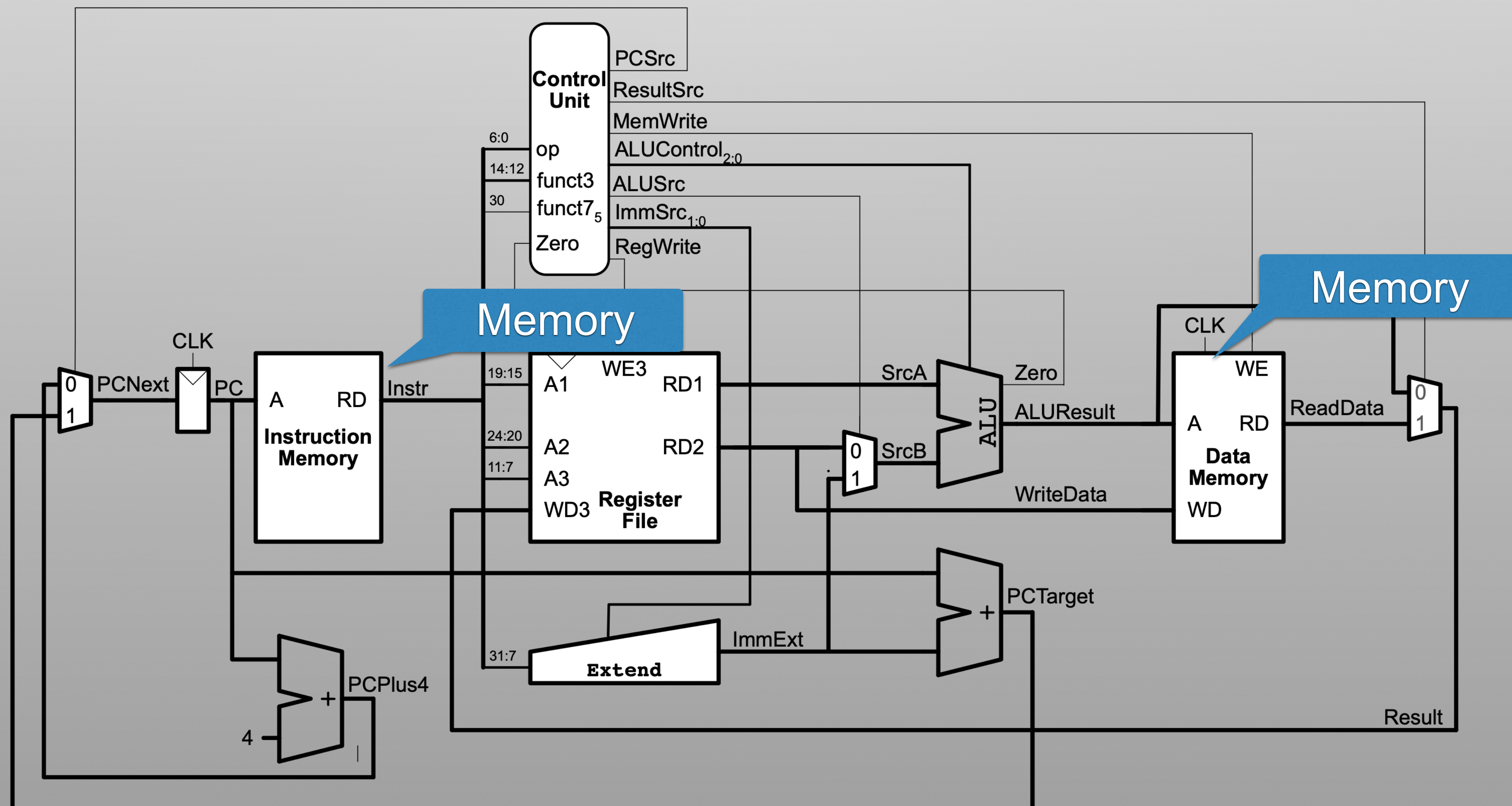


Parts are “Idle” (done with their role in the instruction) most of the time.
This is inefficient!

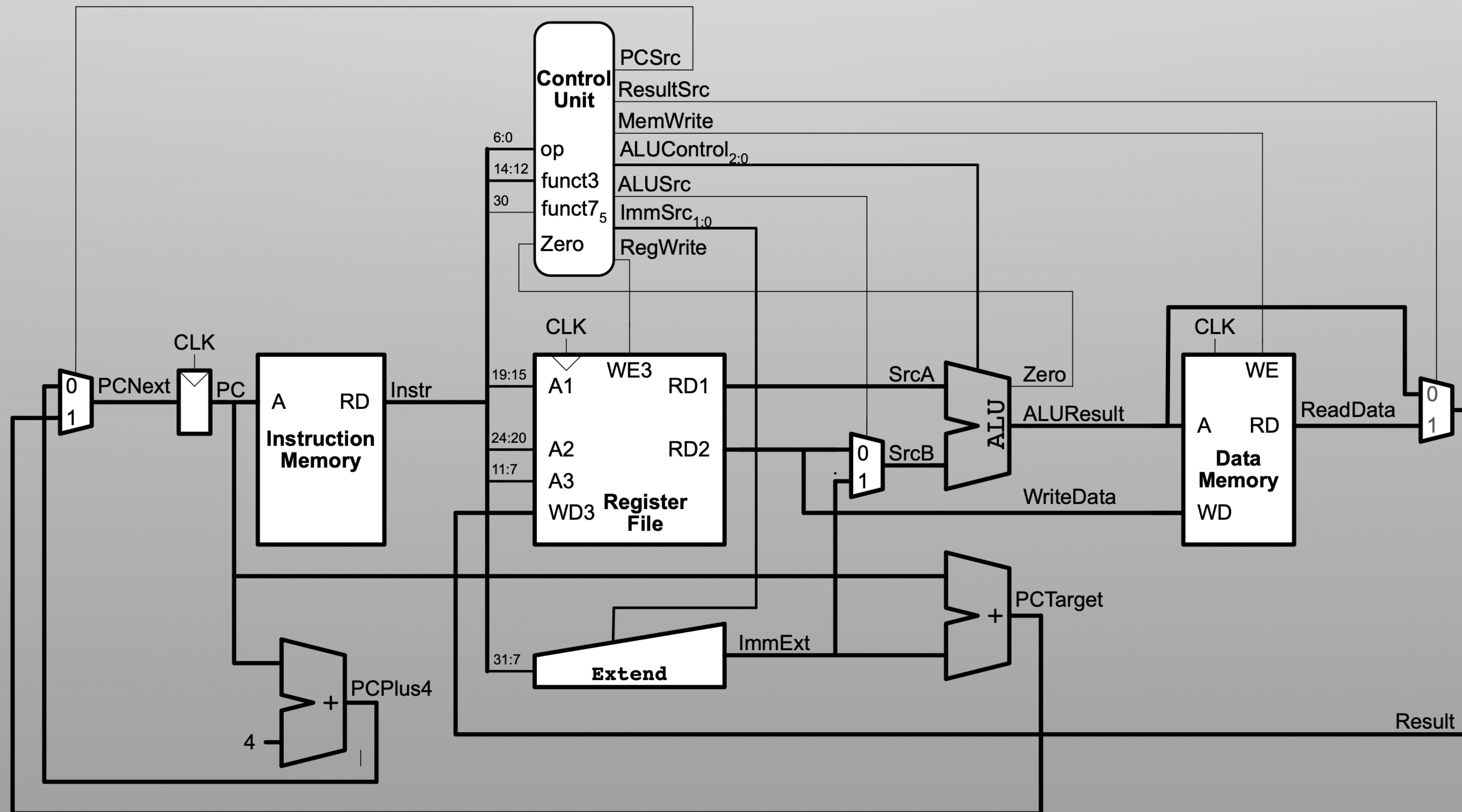
Simple, Single-Cycle: Inefficient!



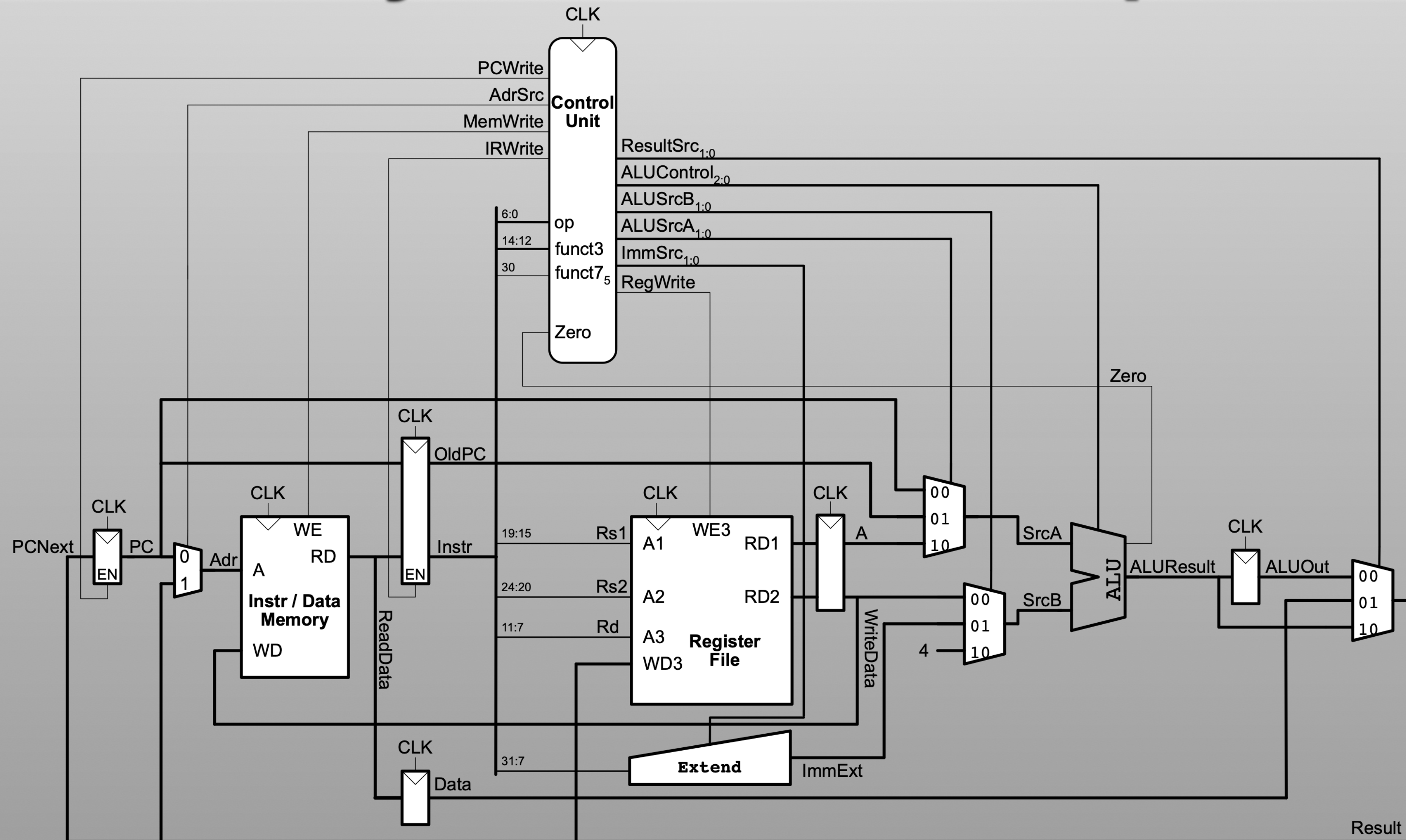
Simple, Single-Cycle: Inefficient!



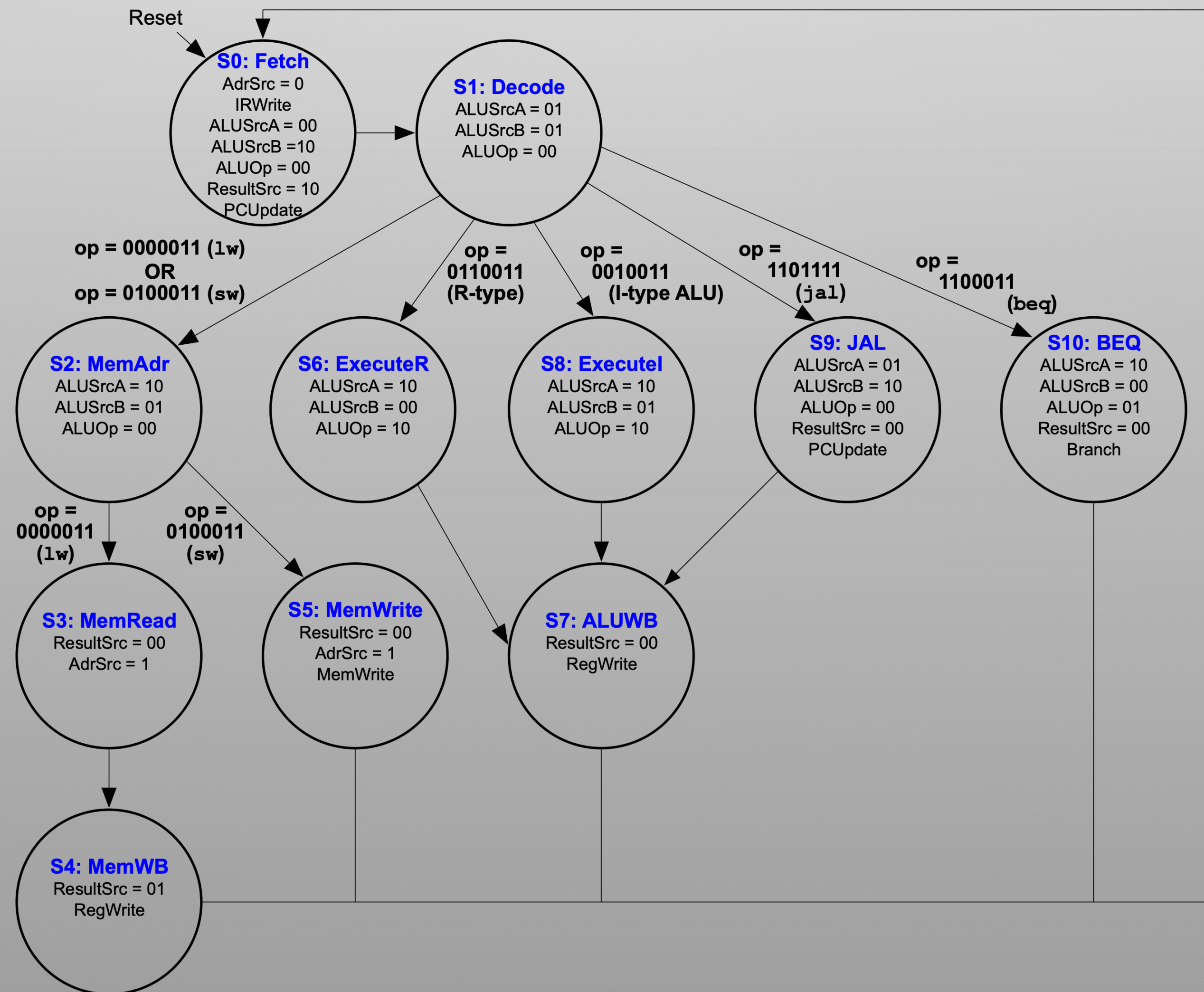
Simple, Single-Cycle RISC-V Computer



Multi-Cycle RISC-V Computer



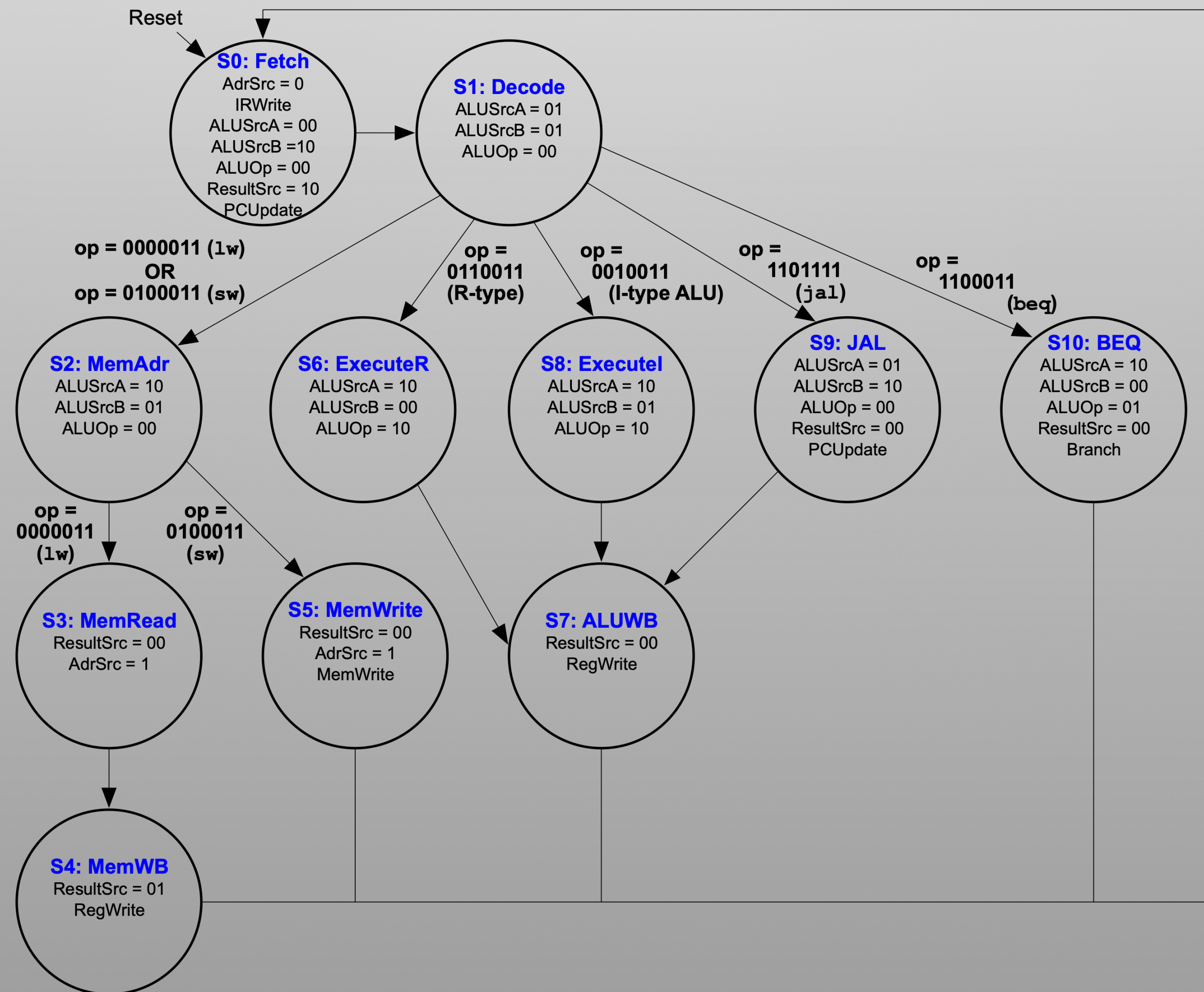
Process



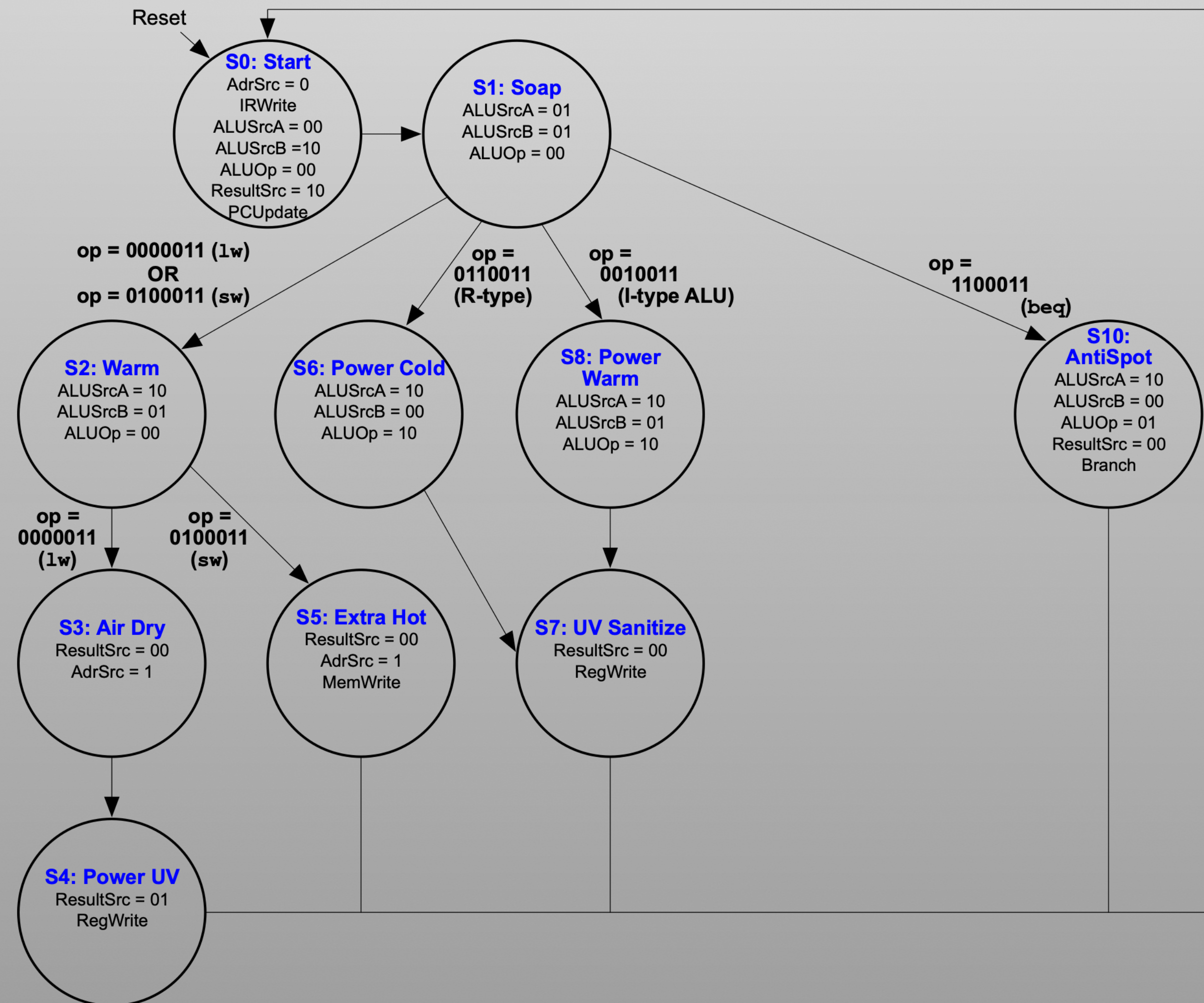
Pros/Cons of Multi-Cycle

- Instructions take only required time: Not constrained by the slowest instruction!
- A little more complex

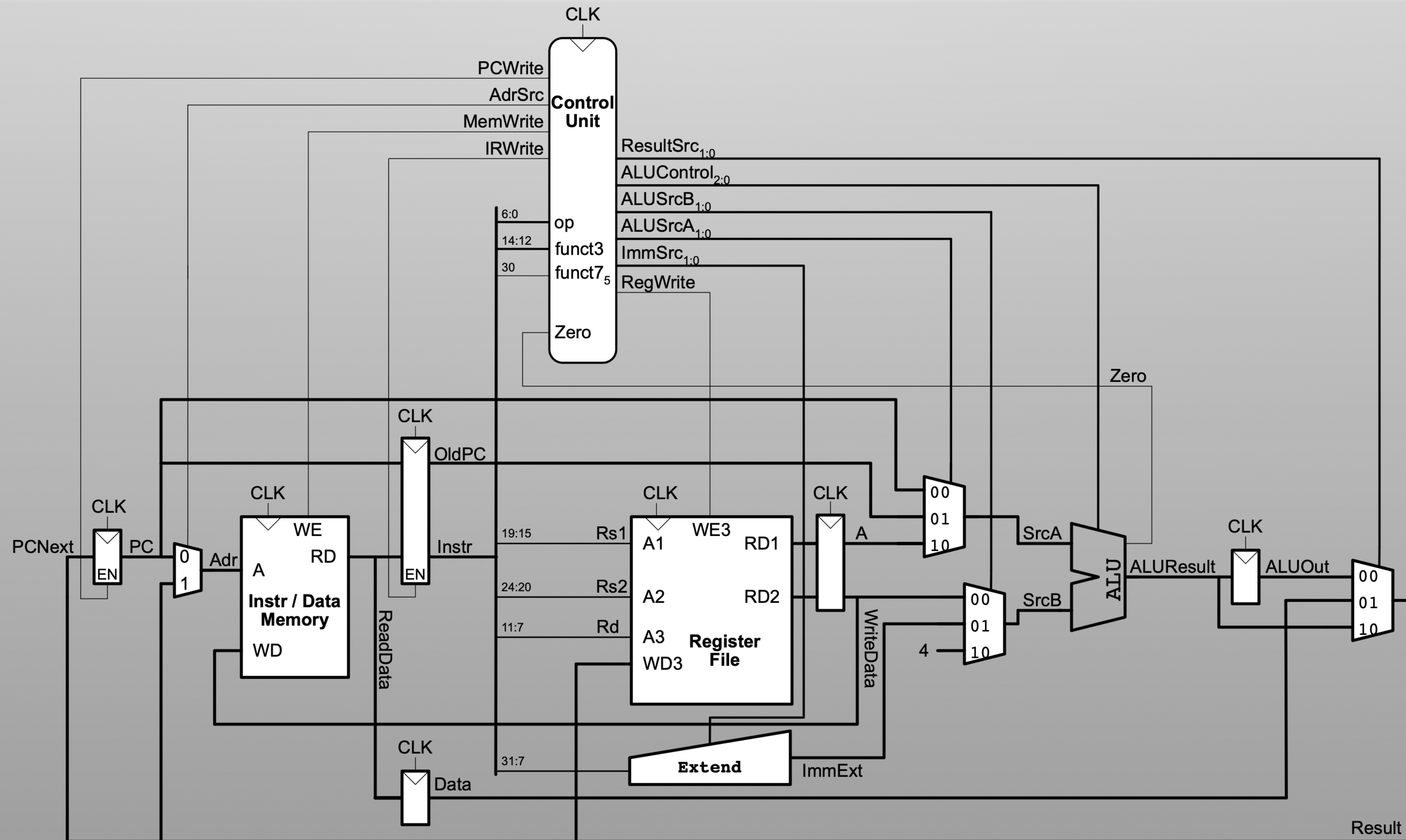
Process



Process: Hw 3B & 4B - Washer



Ex: add t0,t1,t2

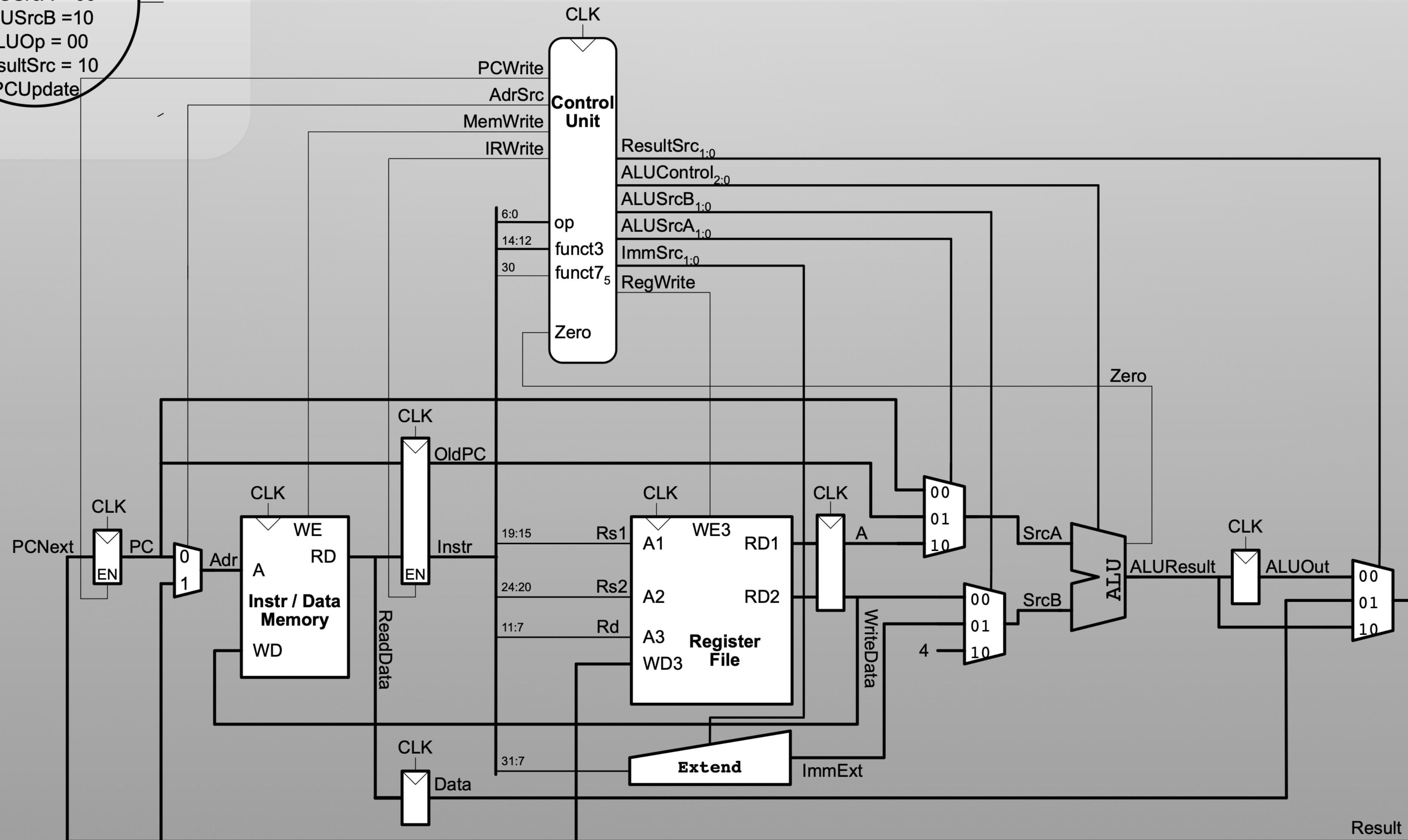


Ex: add t0,t1,t2

Reset

S0: Fetch

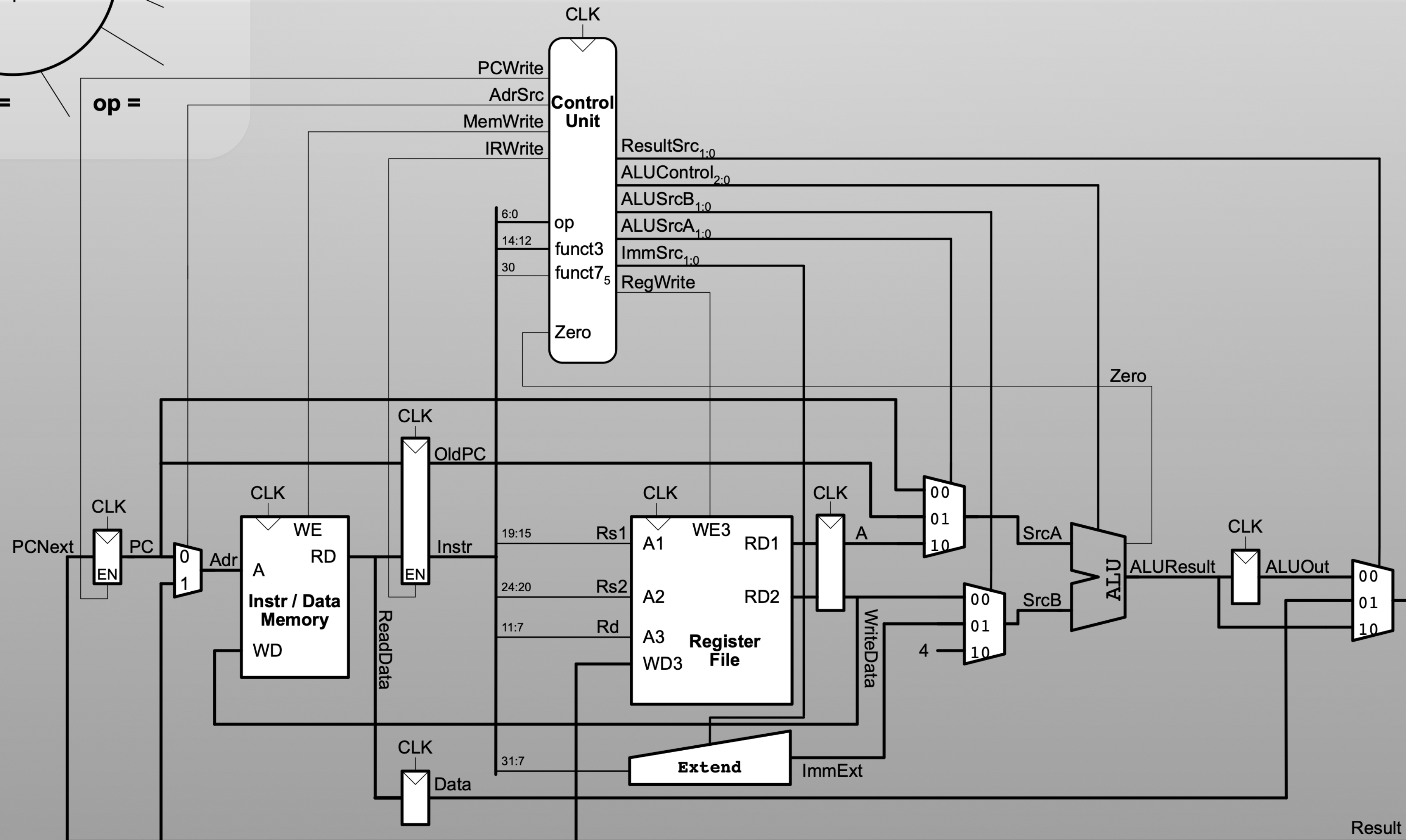
AdrSrc = 0
IRWrite
ALUSrcA = 00
ALUSrcB = 10
ALUOp = 00
ResultSrc = 10
PCUpdate



S1: Decode

ALUSrcA = 01
ALUSrcB = 01
ALUOp = 00

Ex: add t0,t1,t2

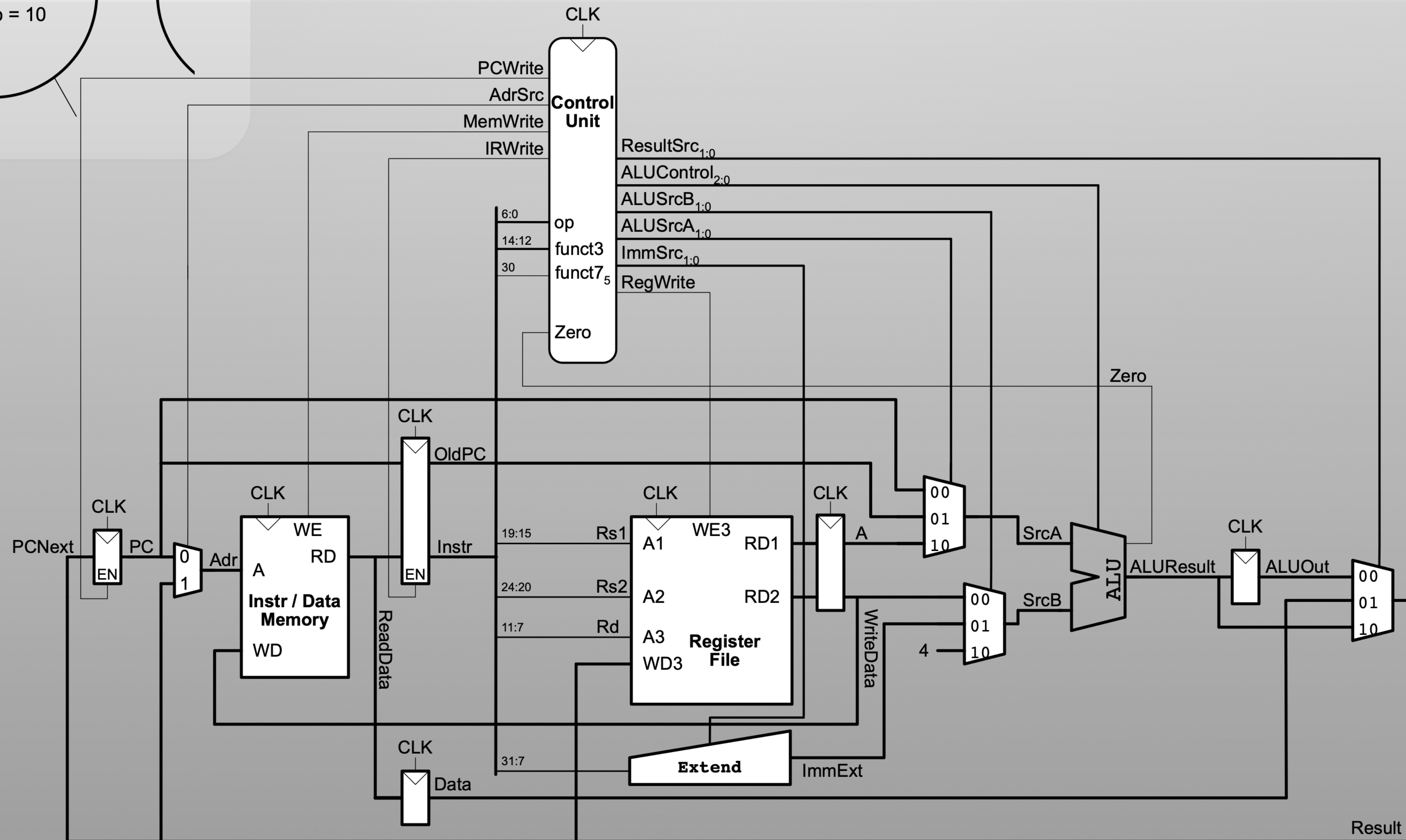


op =
0110011
(R-type)

S6: ExecuteR

ALUSrcA = 10
ALUSrcB = 00
ALUOp = 10

Ex: add t0,t1,t2

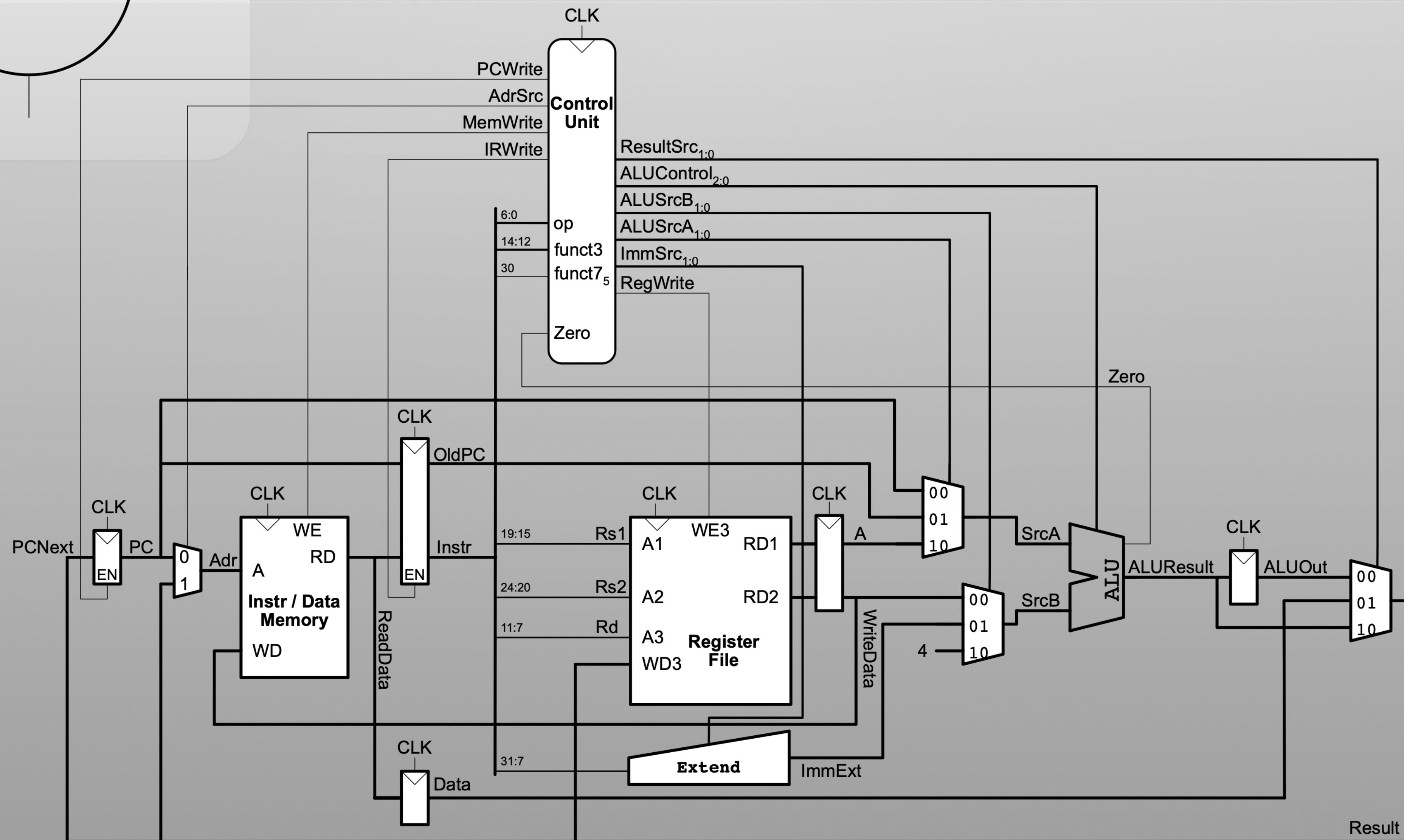


S7: ALUWB

ResultSrc = 00

RegWrite

Ex: add t0,t1,t2



Questions

- The register file inputs and outputs are a bit confusing to me: A1-A3, Rd1, Rd2.
- How exactly do control signals get generated from an instruction step-by-step (not just conceptually, but concretely in hardware)?
- If multicycle solves the problem of long clock cycles but incurs many CPI, which cause slower overall performance, how do we evaluate the trade-offs between multicycle and single-cycle?
- Are there still any single-cycle and multicycle architectural hardware selling now?
- How are hazards handled and how much pipelining actually improves performance in real cases?