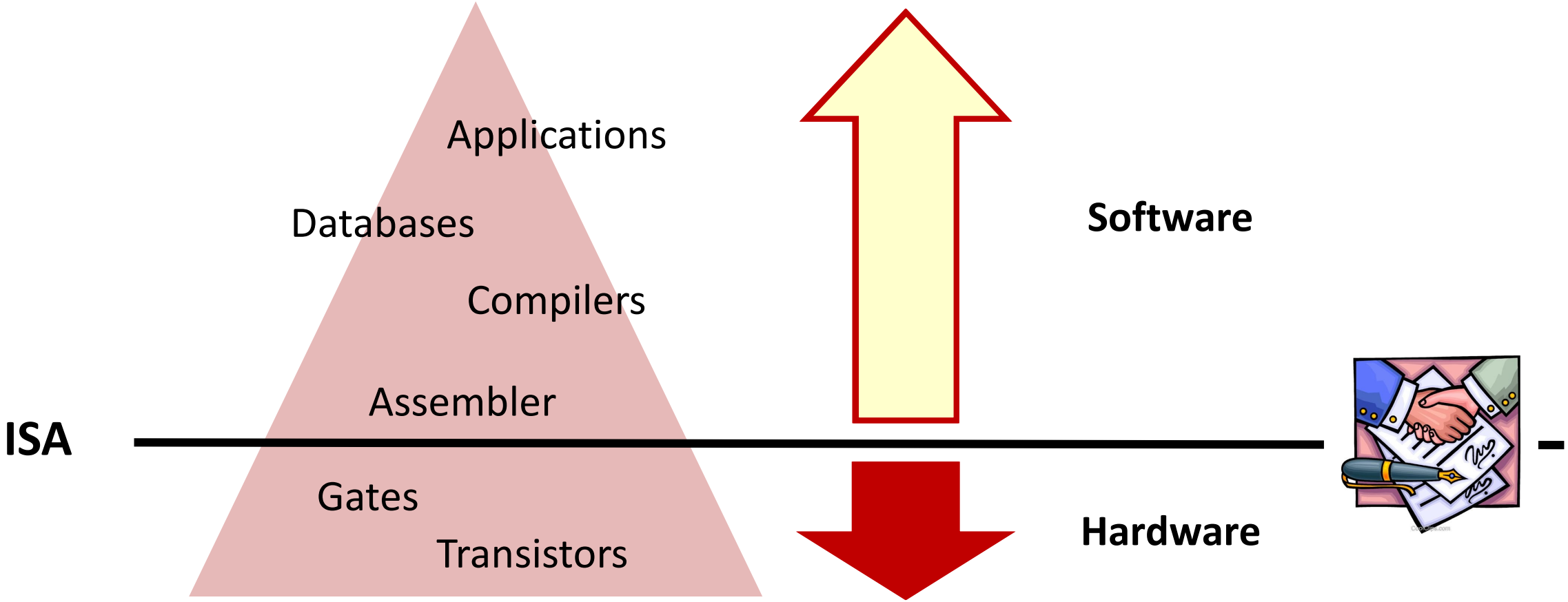
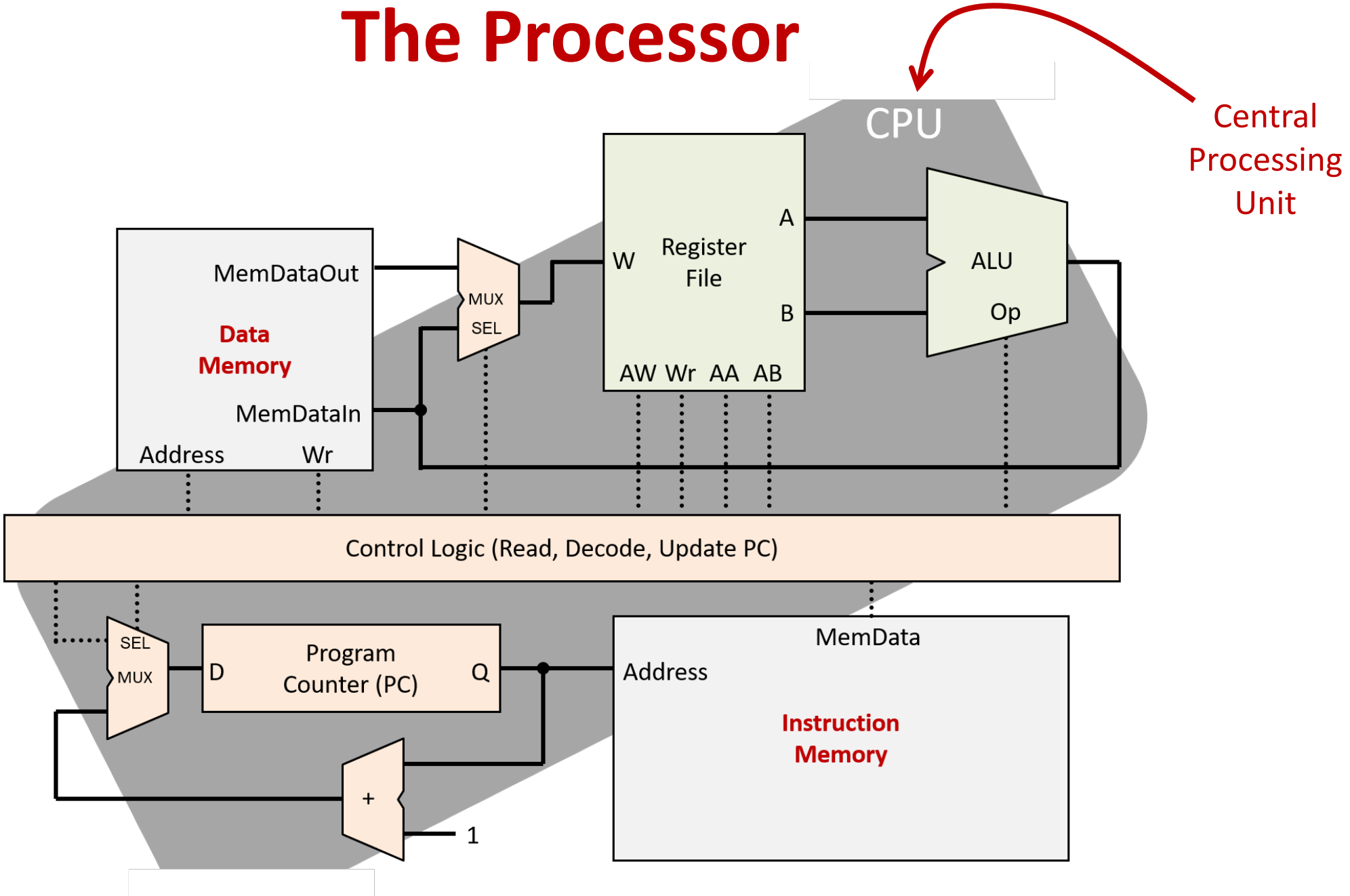
CS-200
Computer Architecture
—
Part 2a. Processor, I/Os, and Exceptions
Multicycle Processor

Paolo Ienne
<paolo.ienne@epfl.ch>

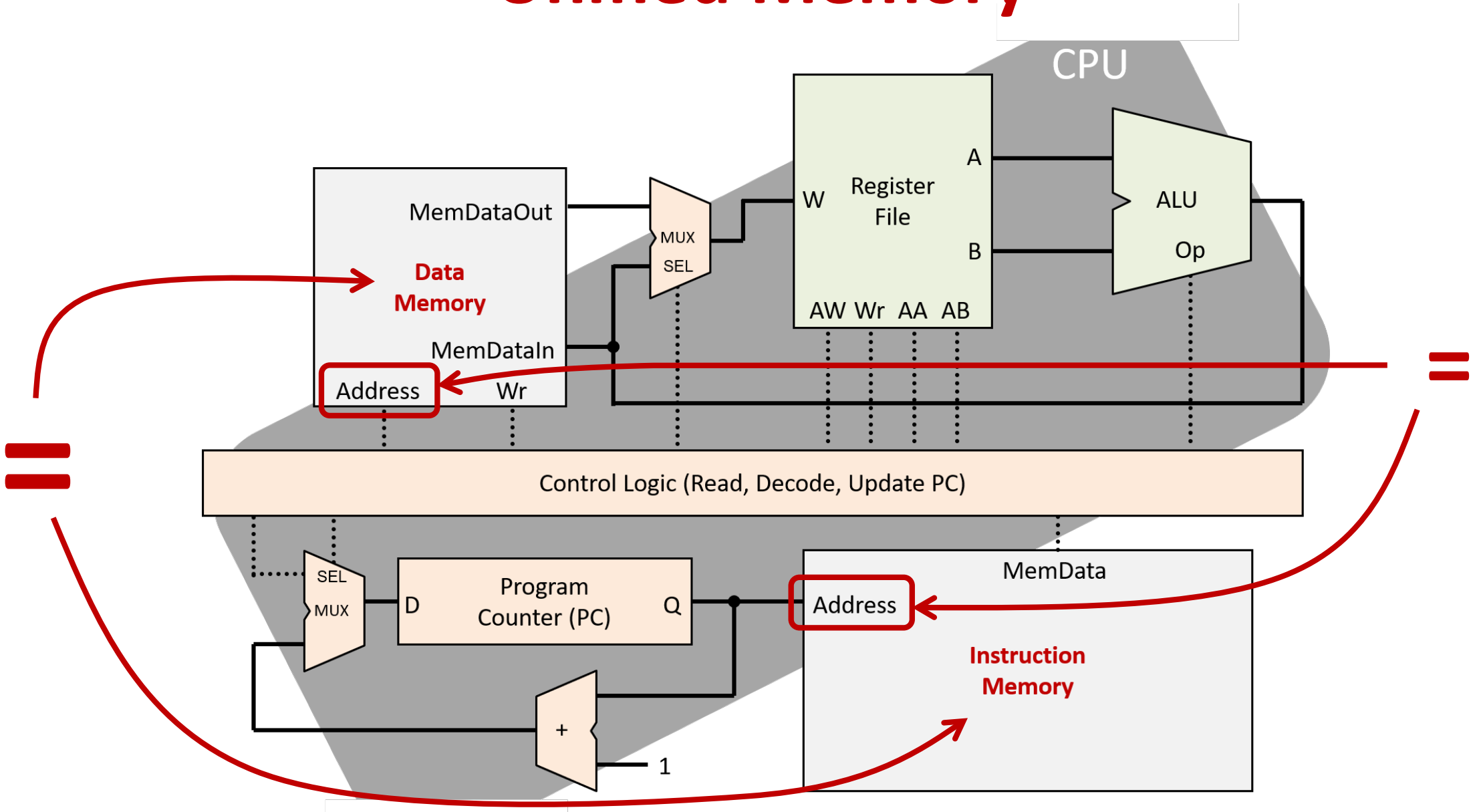
The Contract between HW and SW



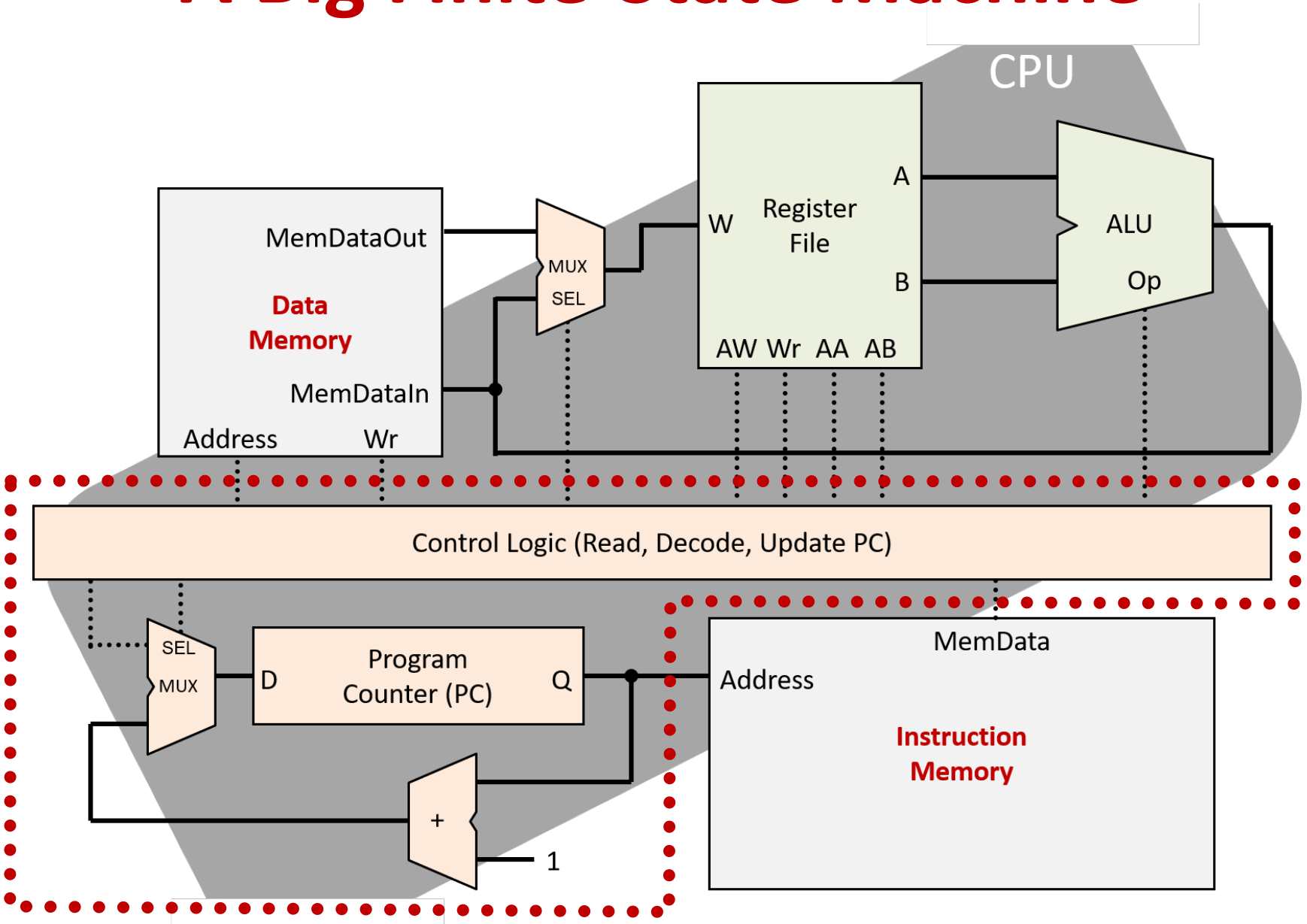
The Processor



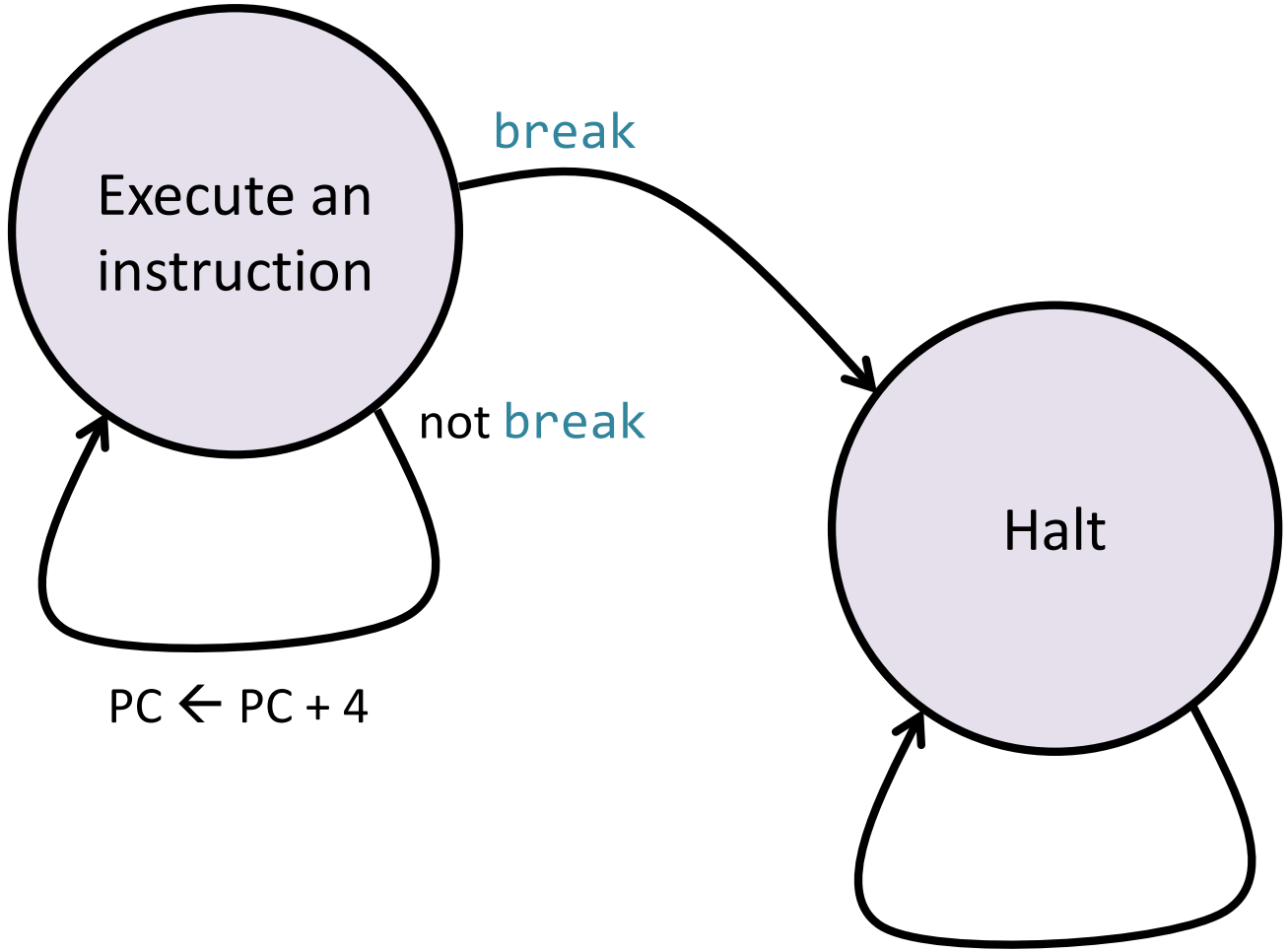
Unified Memory



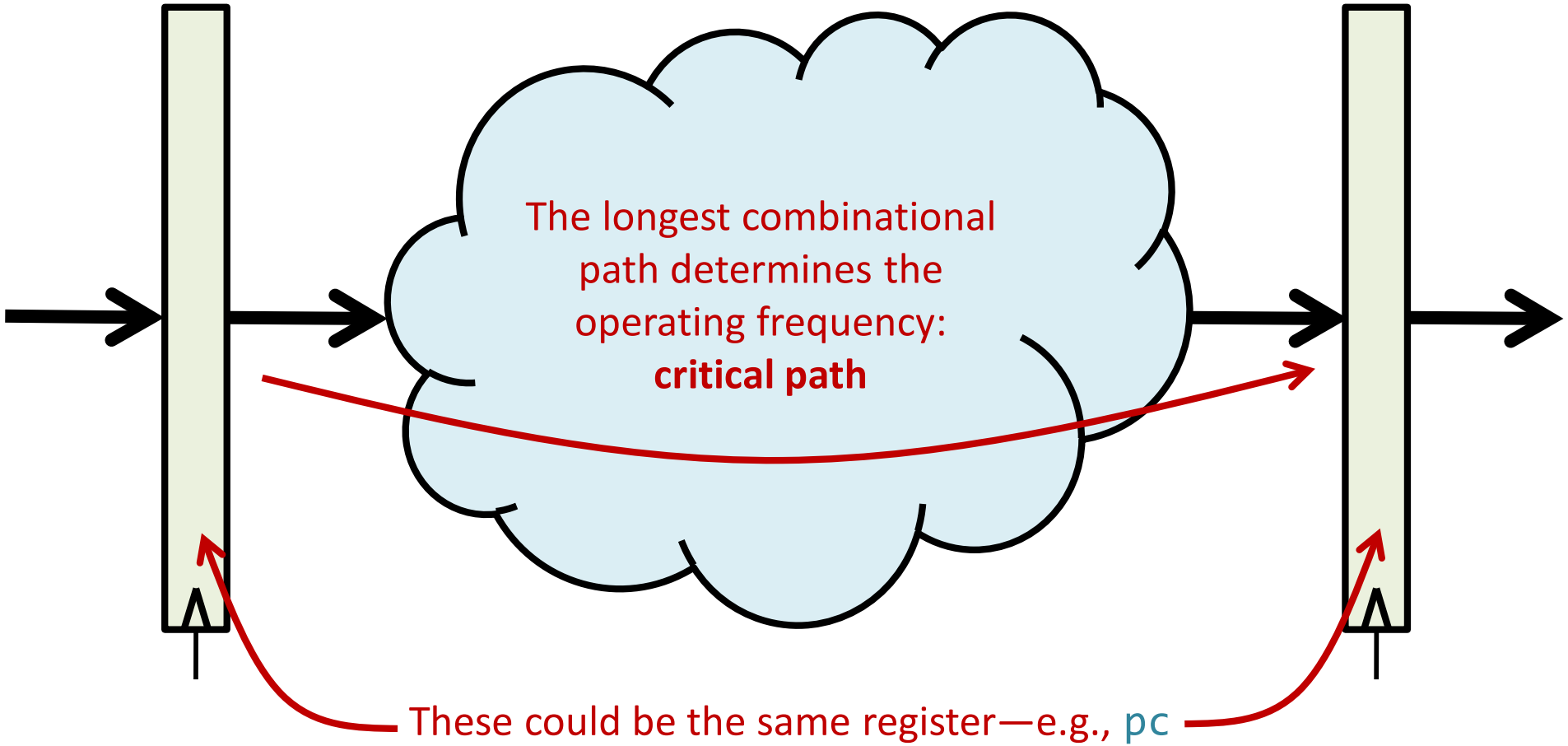
A Big Finite-State Machine



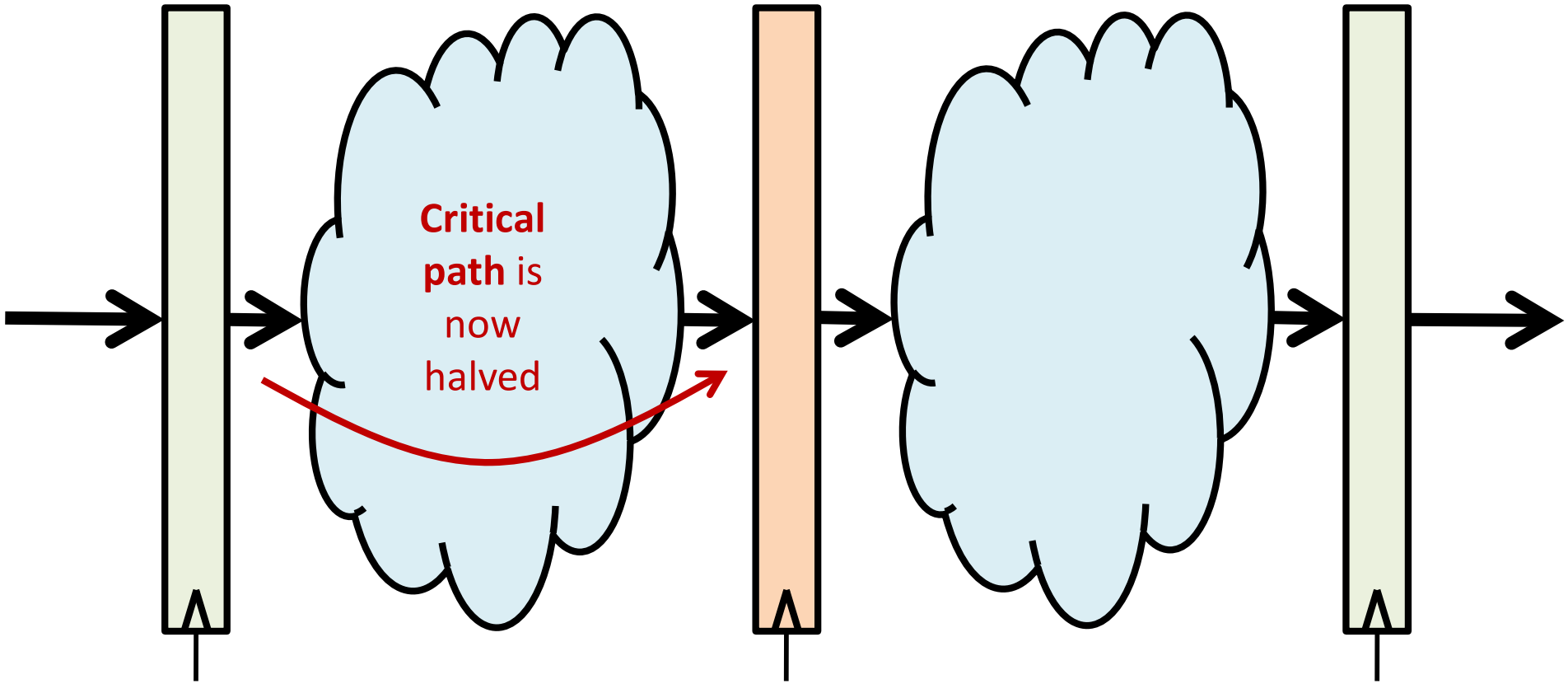
Single-Cycle Processor



Propagation Time



Increasing the Frequency?

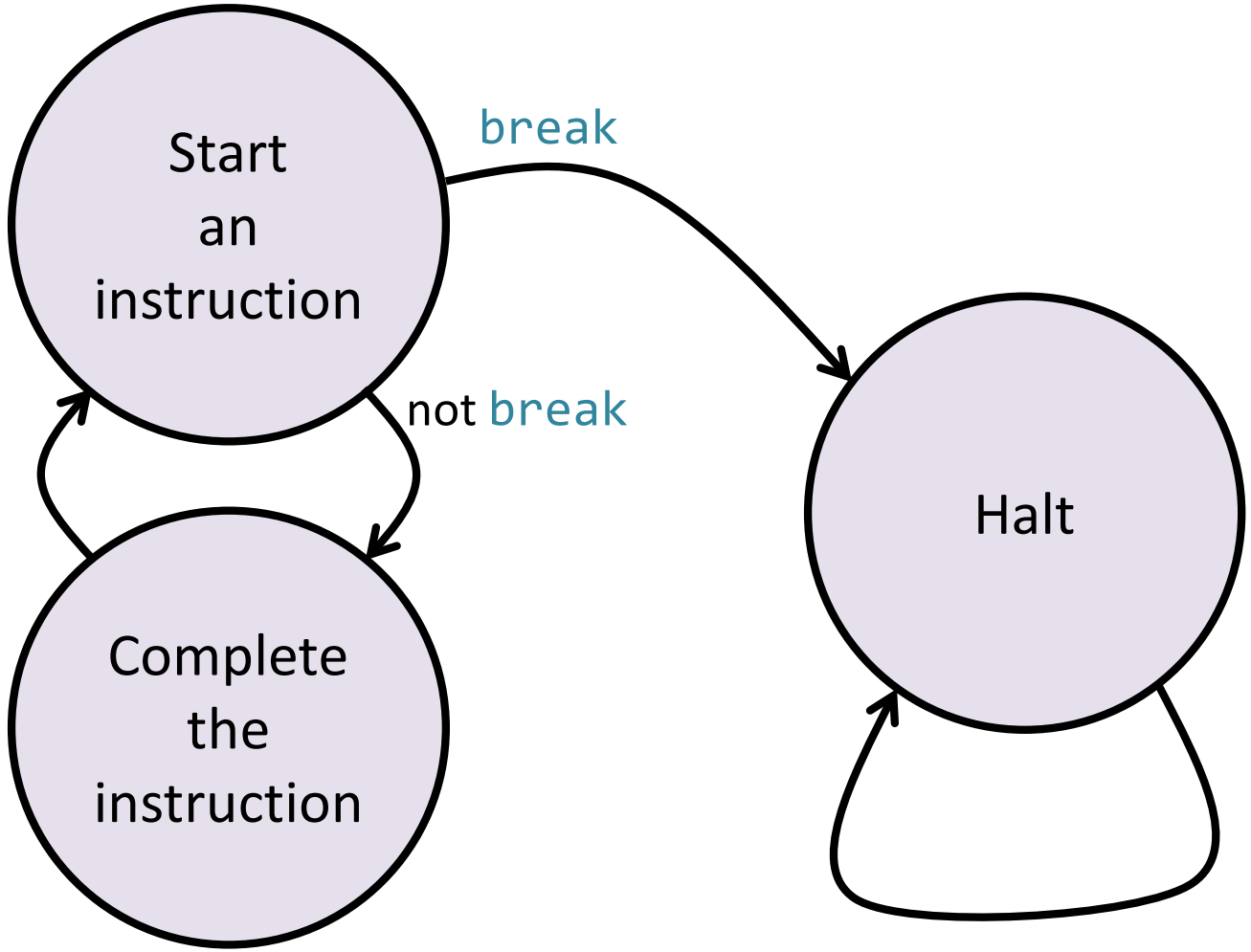


Two-Cycle Processor

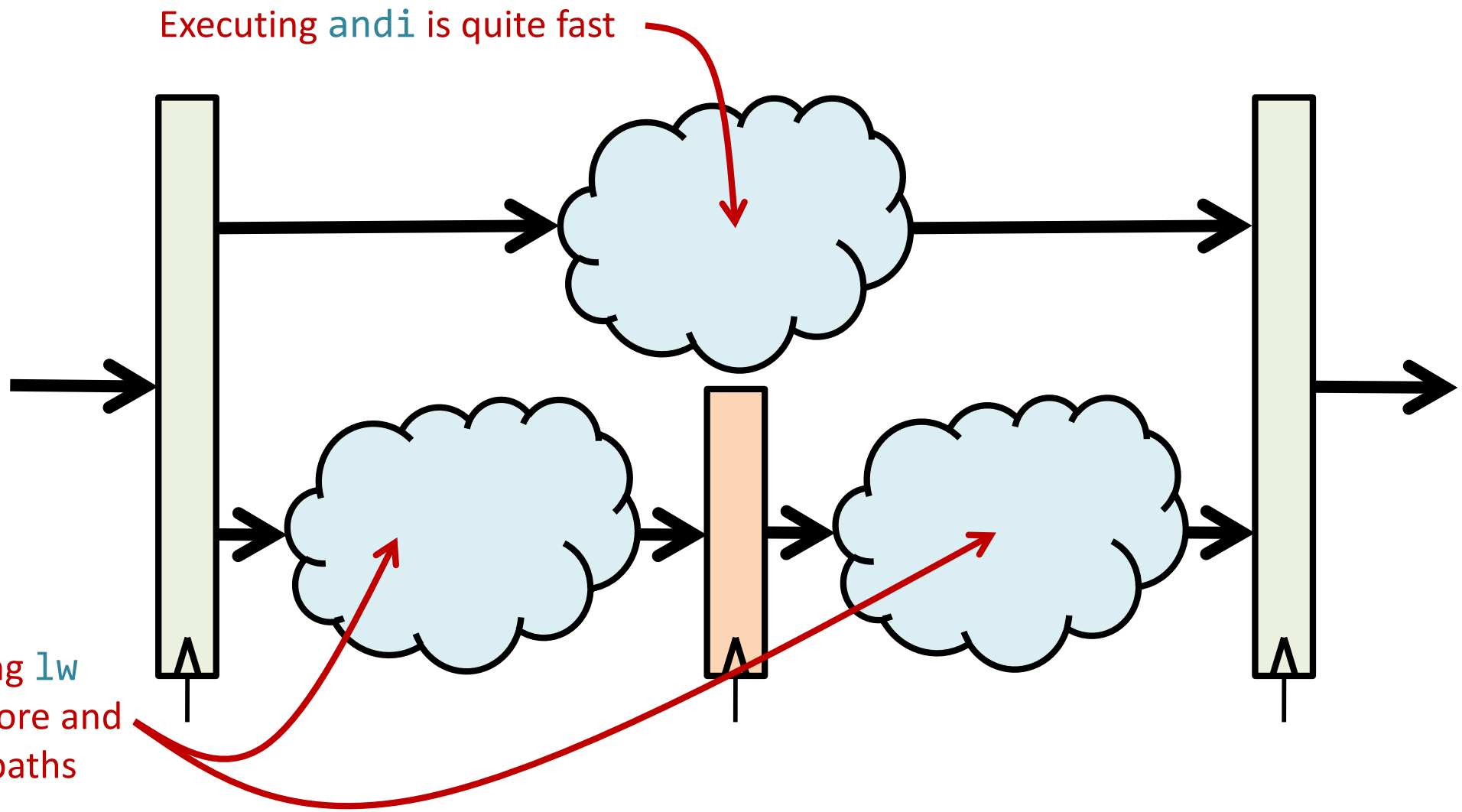
Did we gain anything?

1 instruction
per cycle
at frequency **F**

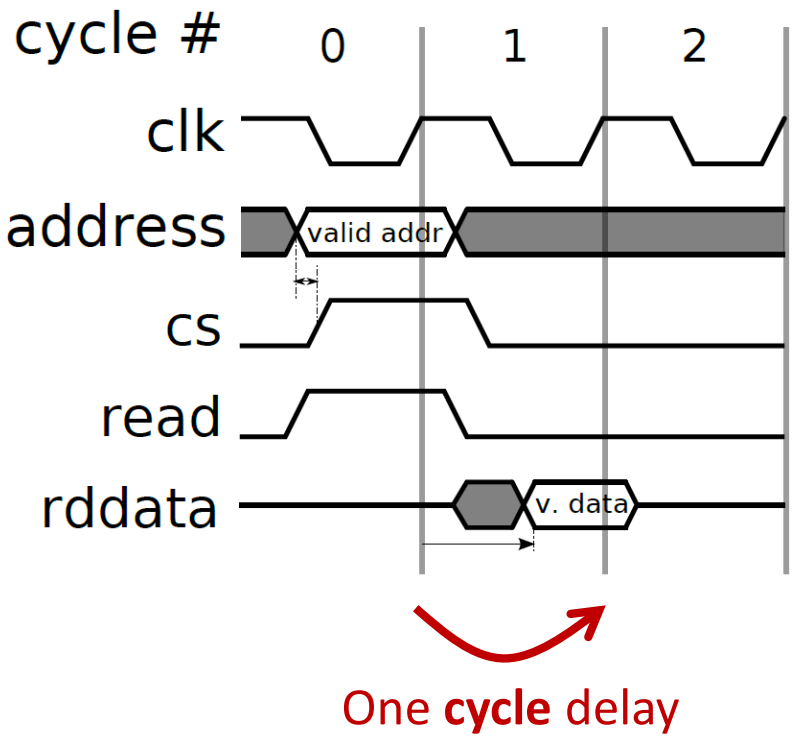
1 instruction
every **two cycles**
at frequency **2F**

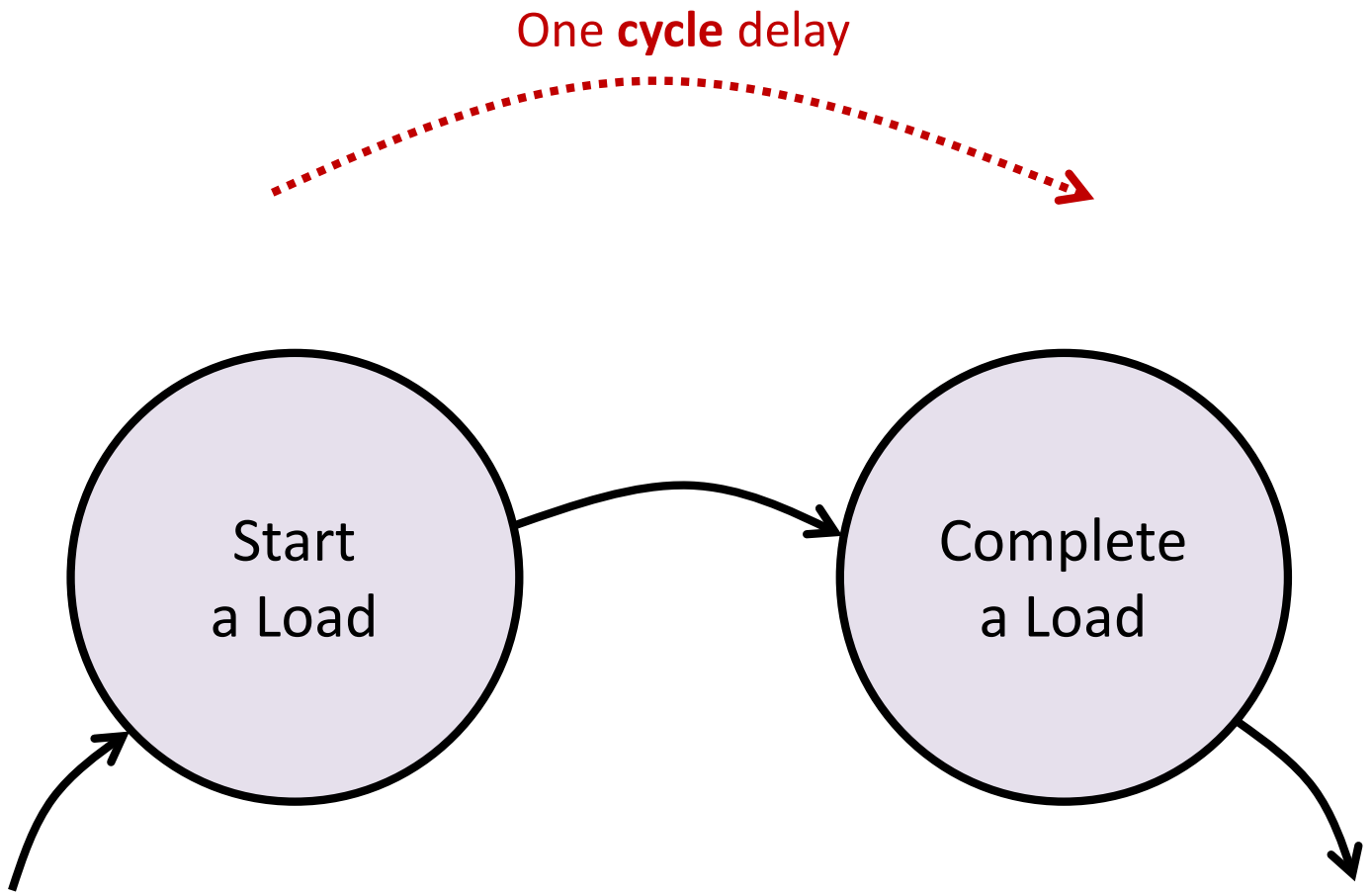


Not All Paths Are Born Equal!

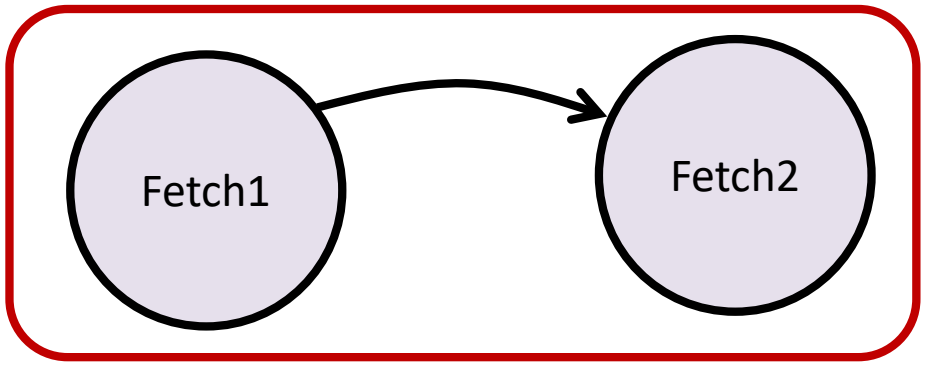


And Maybe Memories Are Sequential



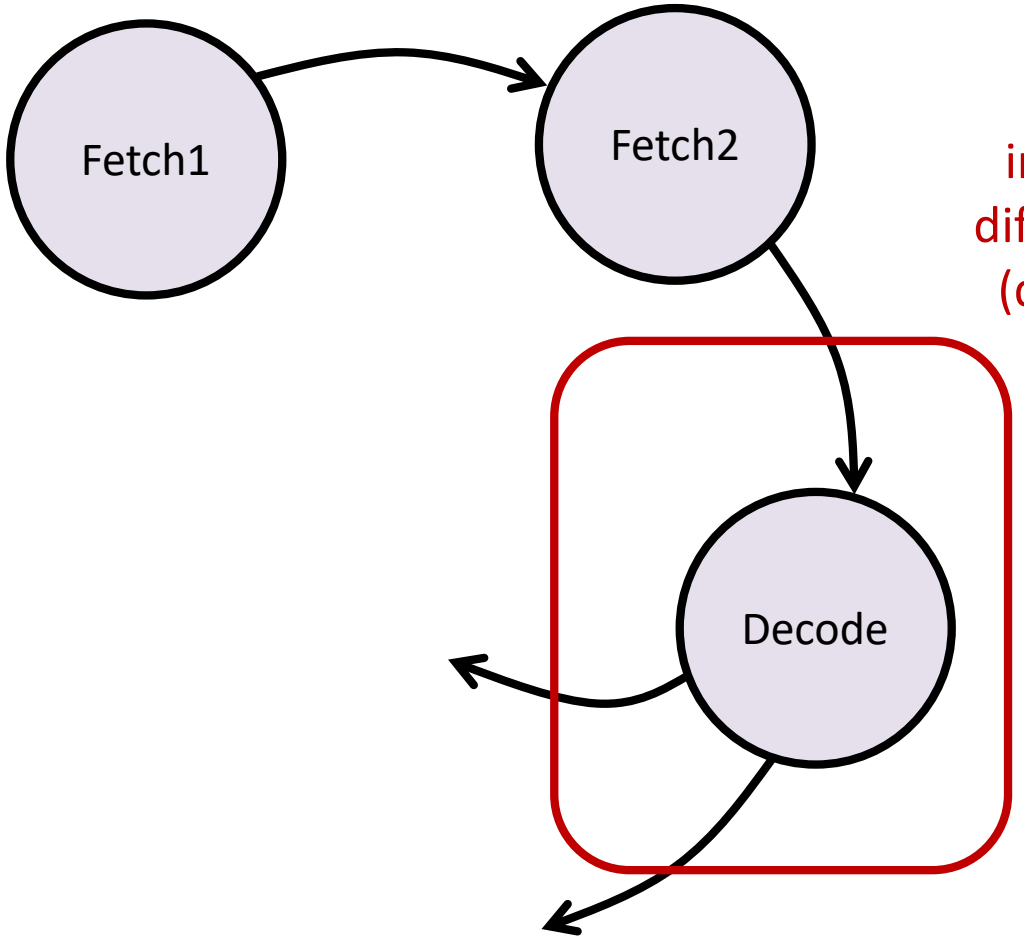


Multi-Cycle Processor



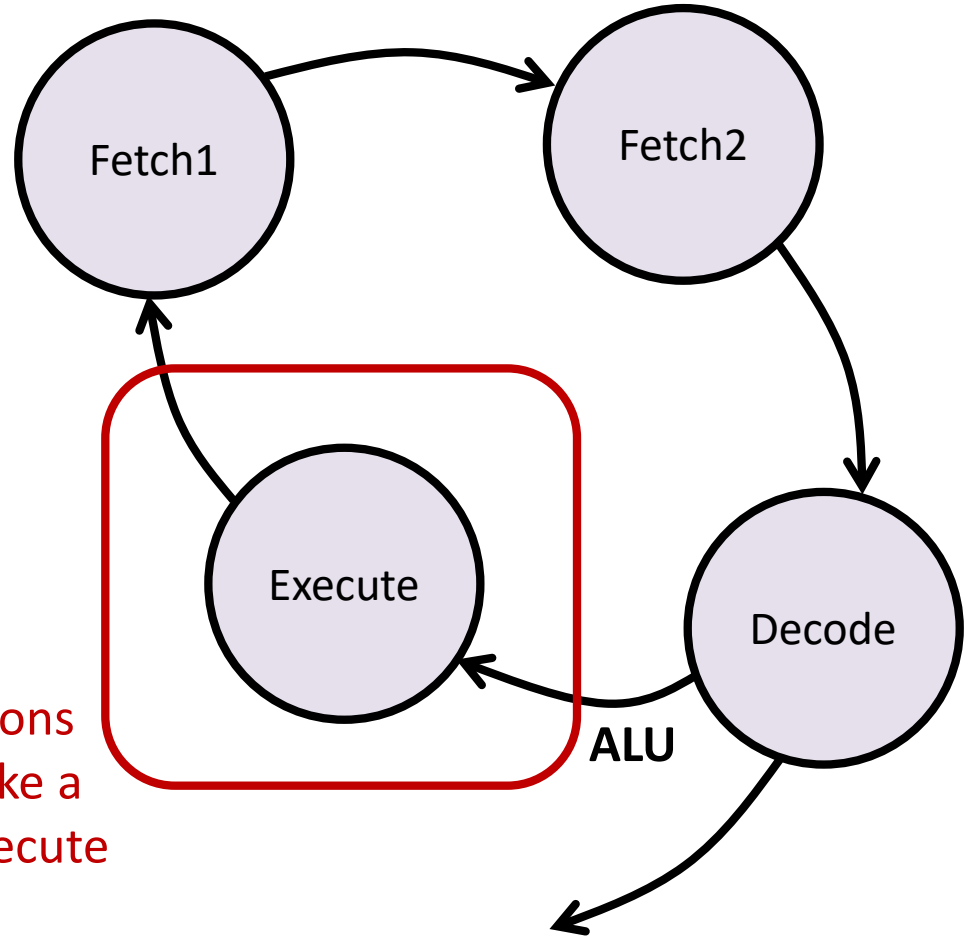
Get the instruction from memory

Multi-Cycle Processor



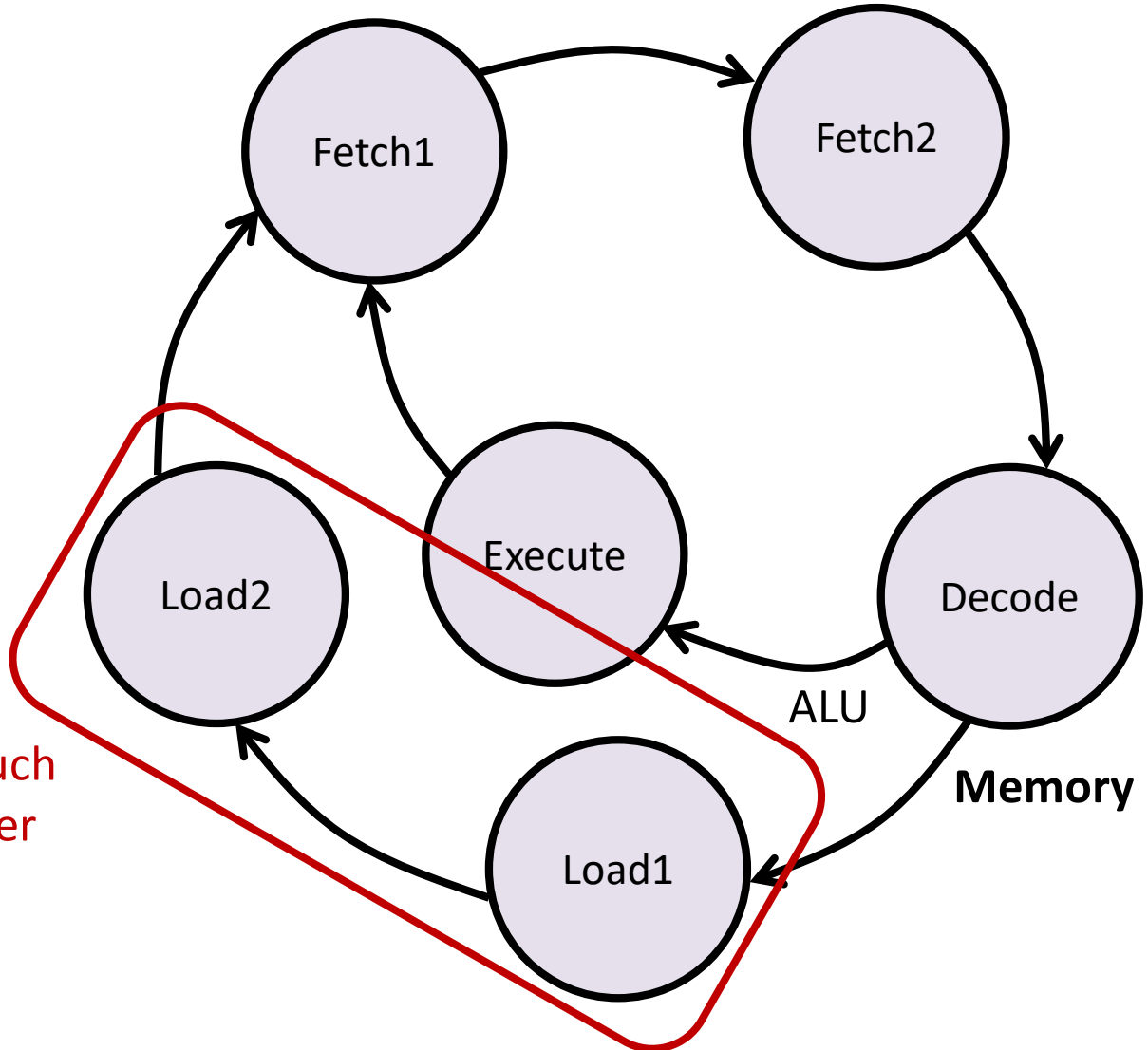
"Understand" the instruction and take a different path as needed (different resources to control, different complexity)

Multi-Cycle Processor



Simple instructions such as `andi` take a single cycle to execute

Multi-Cycle Processor



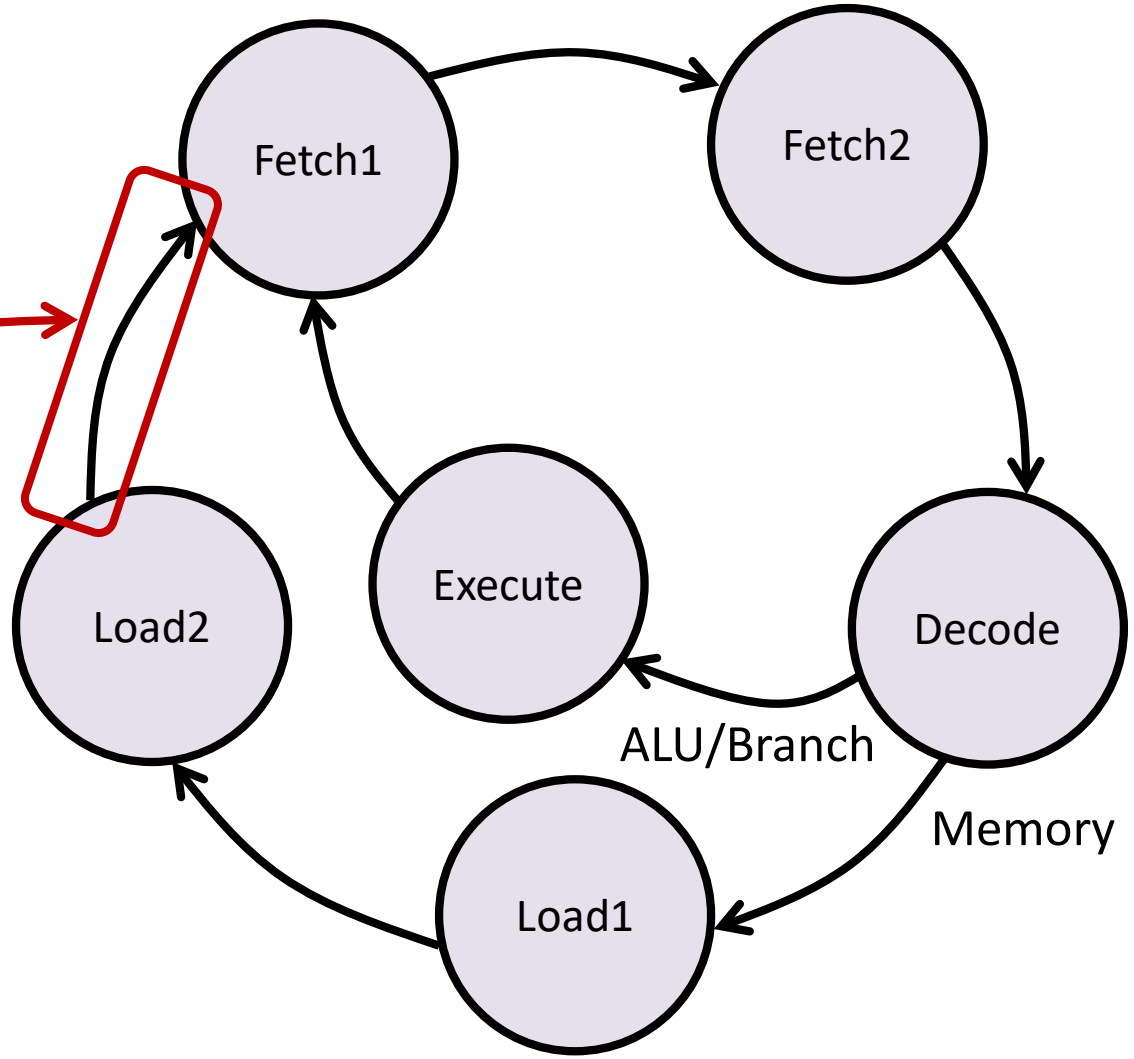
Complex ones such as `lw` take longer

Multi-Cycle Processor

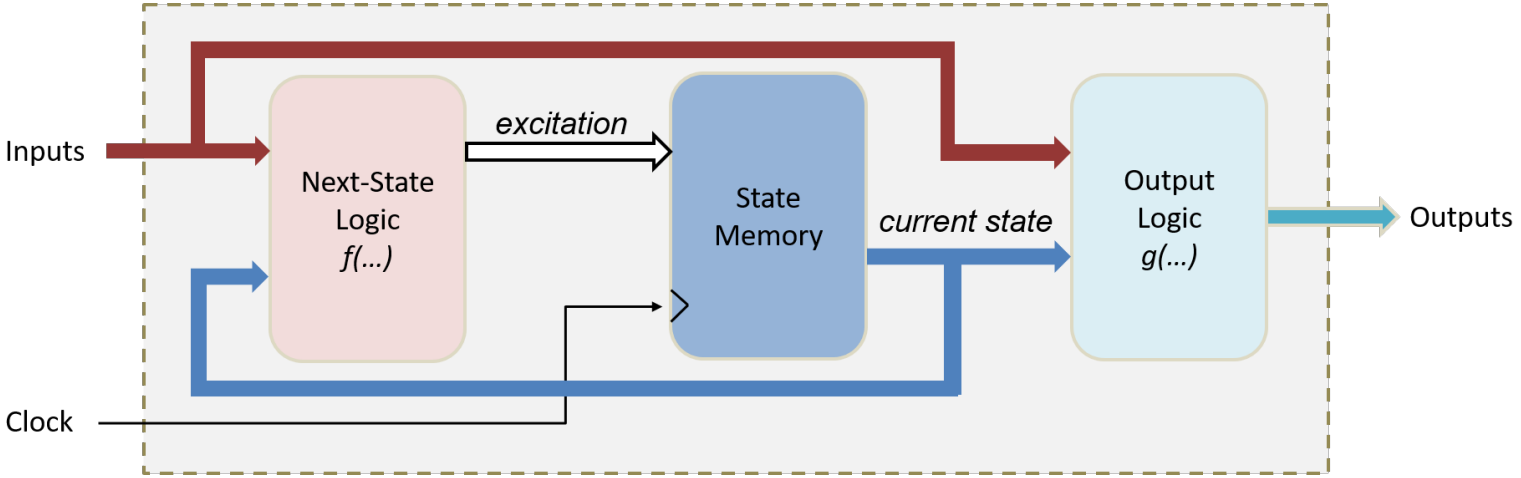
We will try

① **not** to have **too many** stages

② to have paths as **balanced** as possible

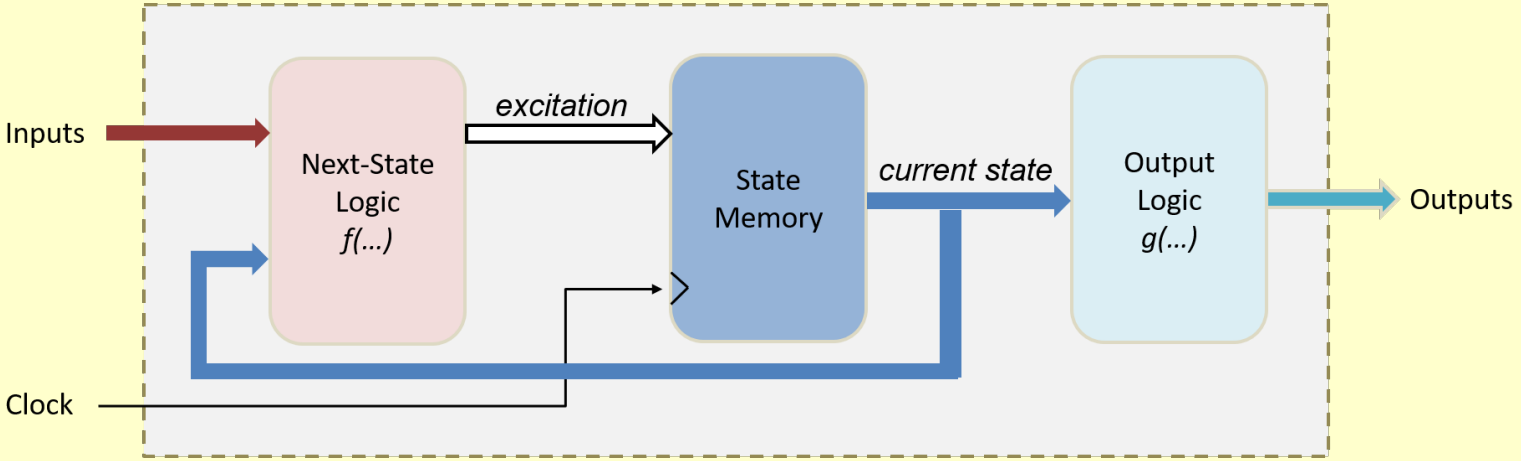


Mealy or Moore?



Mealy FSM

Output depends on **input and state**



Moore FSM

Output depends on **state only**

Sometimes this needs more cycles…

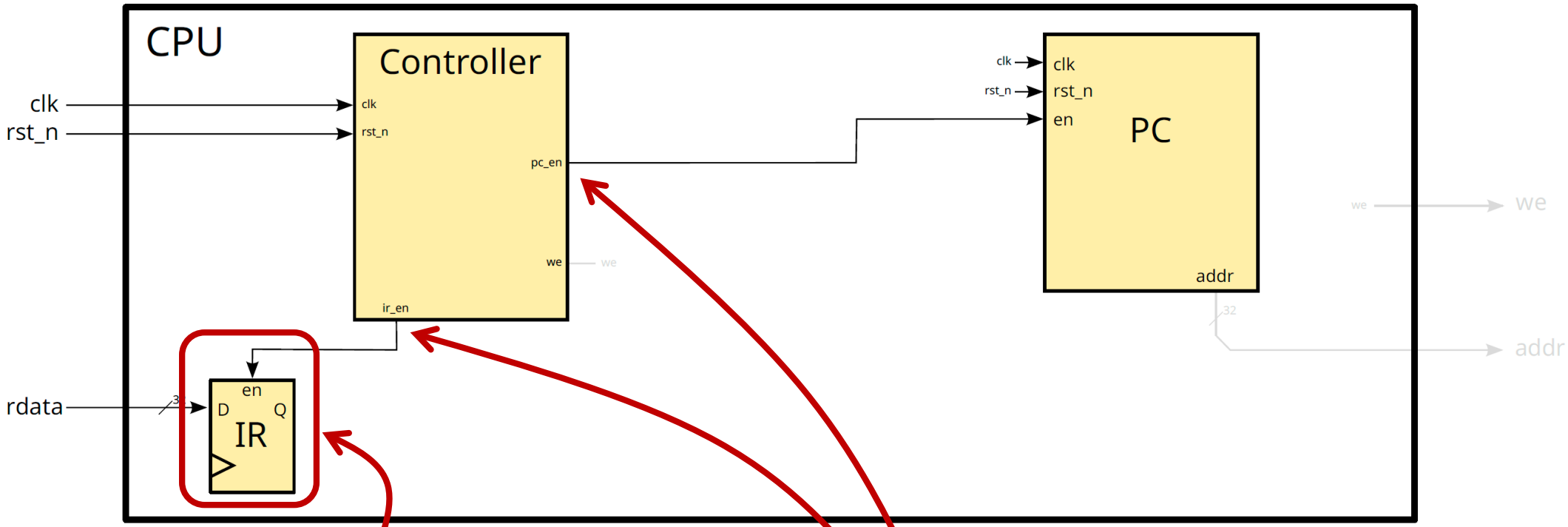
…and sometimes this is not acceptable

Building the Circuit

An “empty” FSM
(we still do not know
what we need to control…)

Basic registers
we need

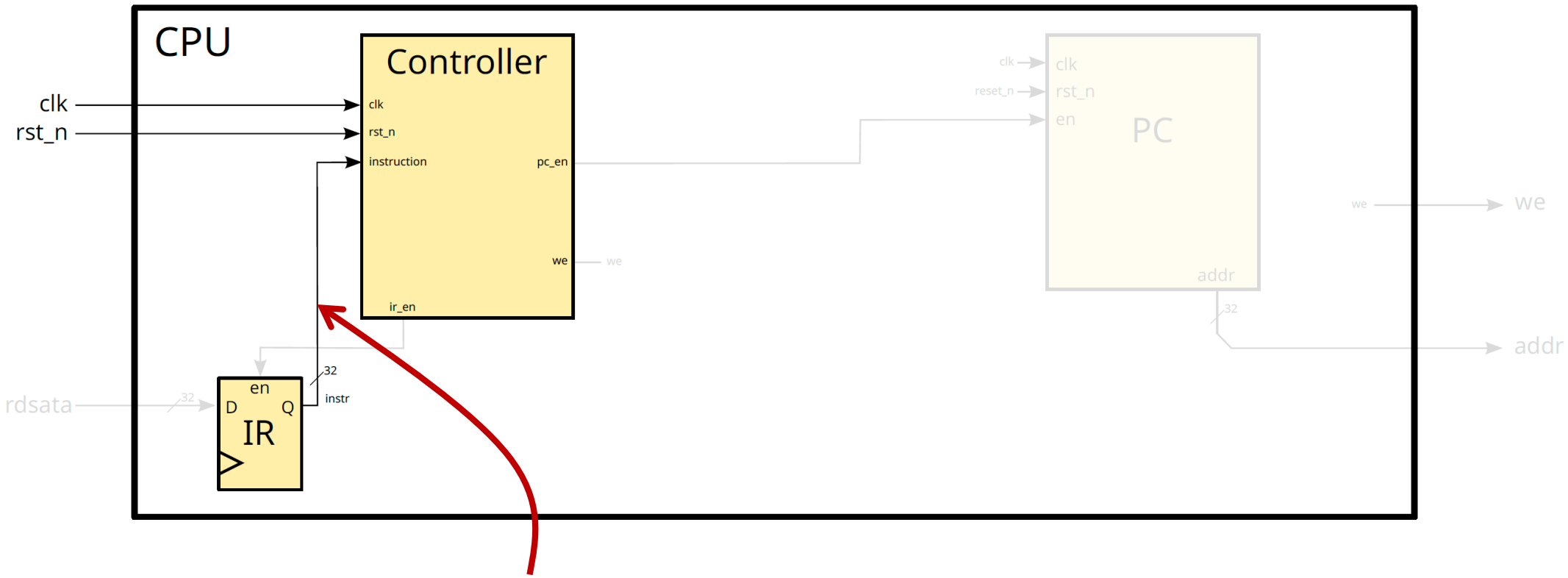
Add Progressively What We Need



We will need an **Instruction Register** to memorize the instruction coming from memory

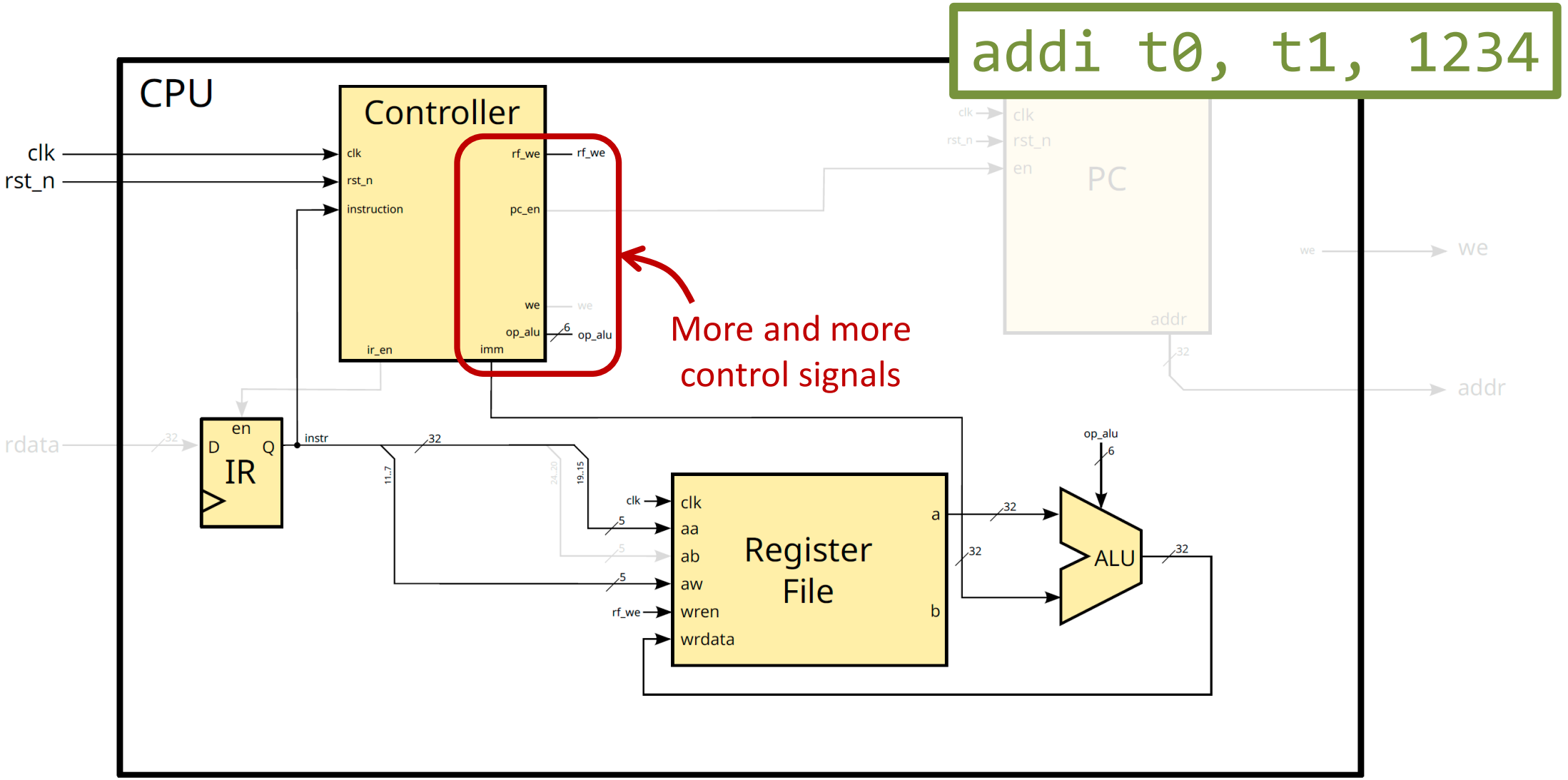
We will need to **enable** these registers when there is something new to record
→ we will have the **FSM generate them**

Follow the Functionality

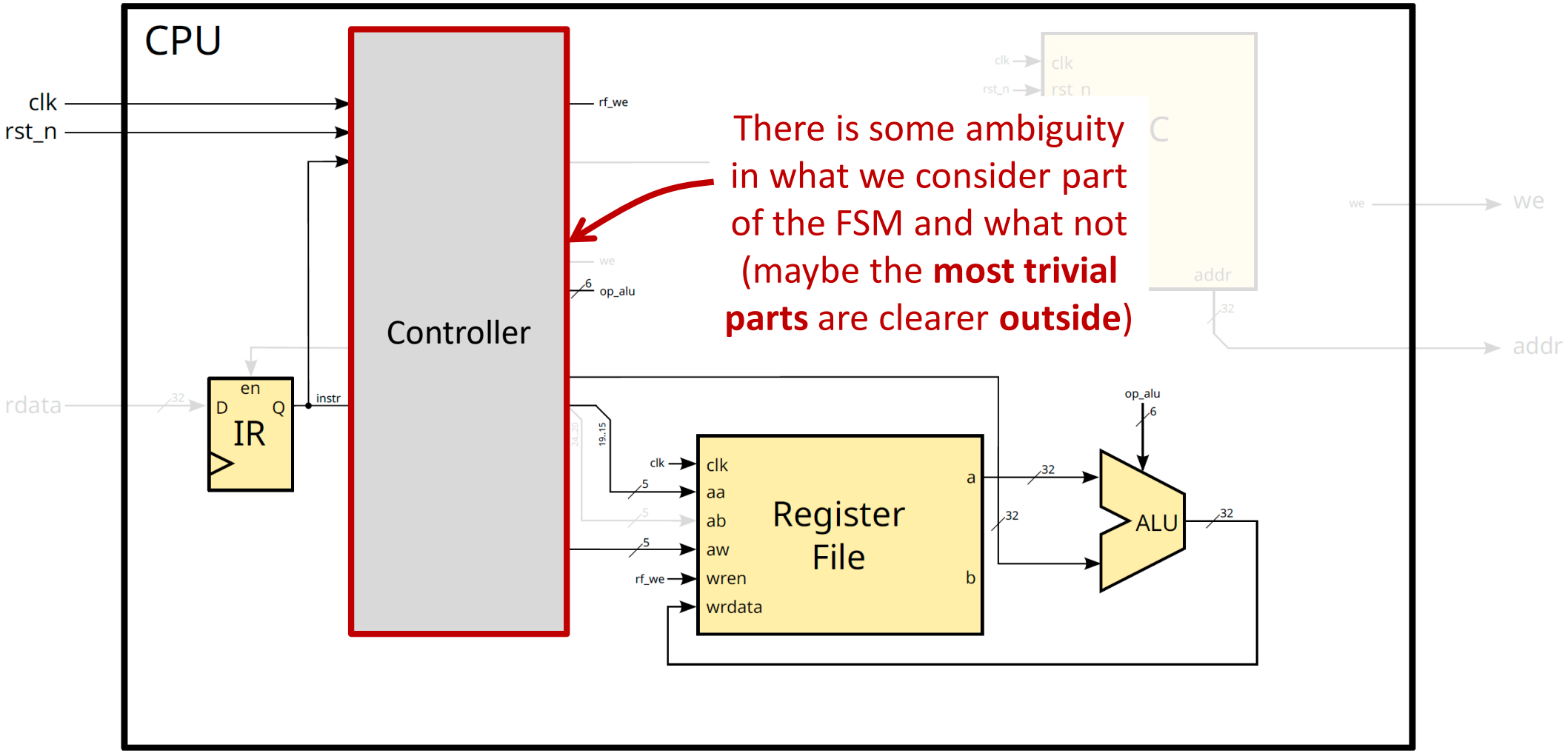


When an instruction arrives into IR,
the controller needs to know what it is
(the next state depends on it!)
→ **connect IR to the controller**

I-Type Instructions Need RF and ALU

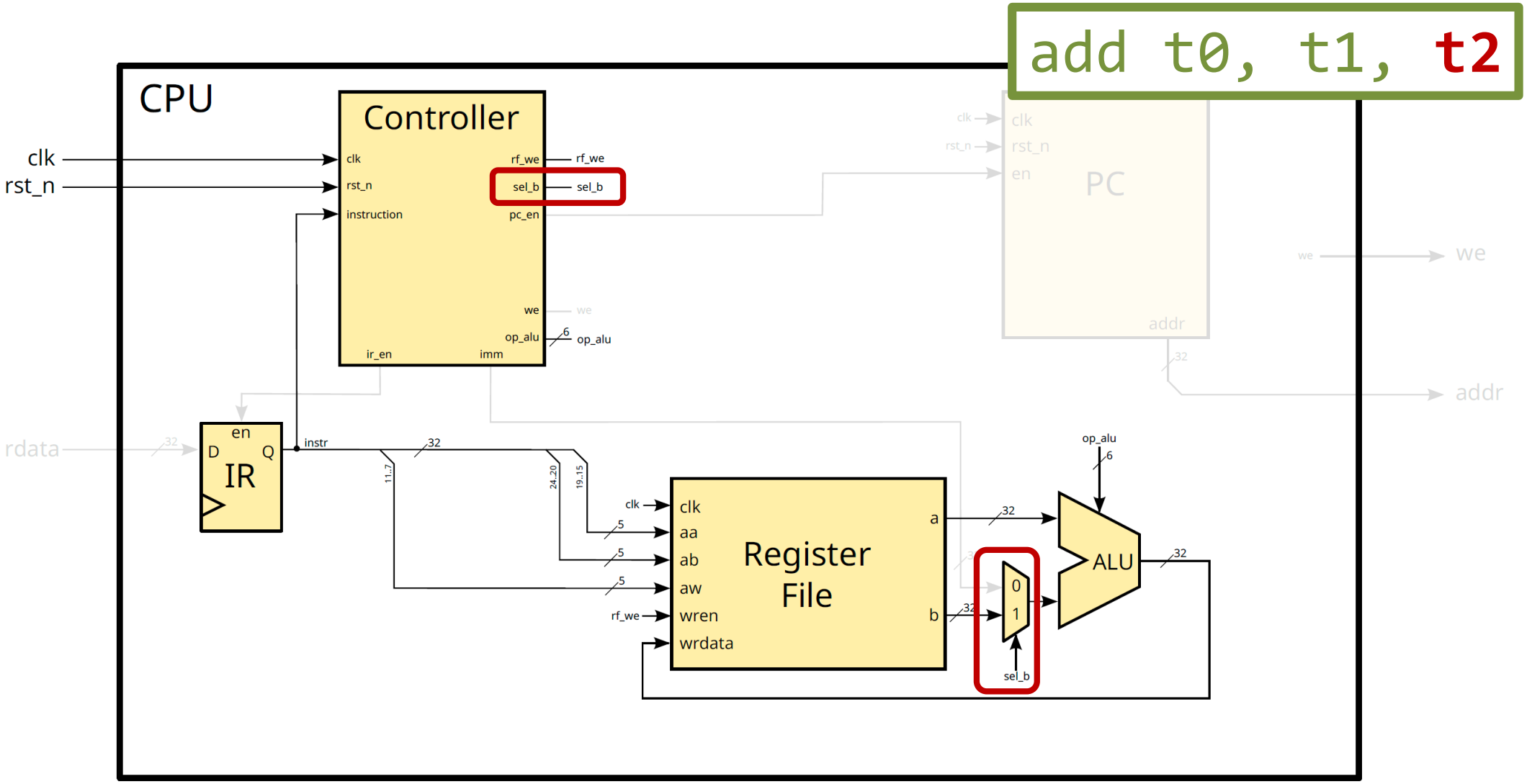


I-Type Instructions Need RF and ALU



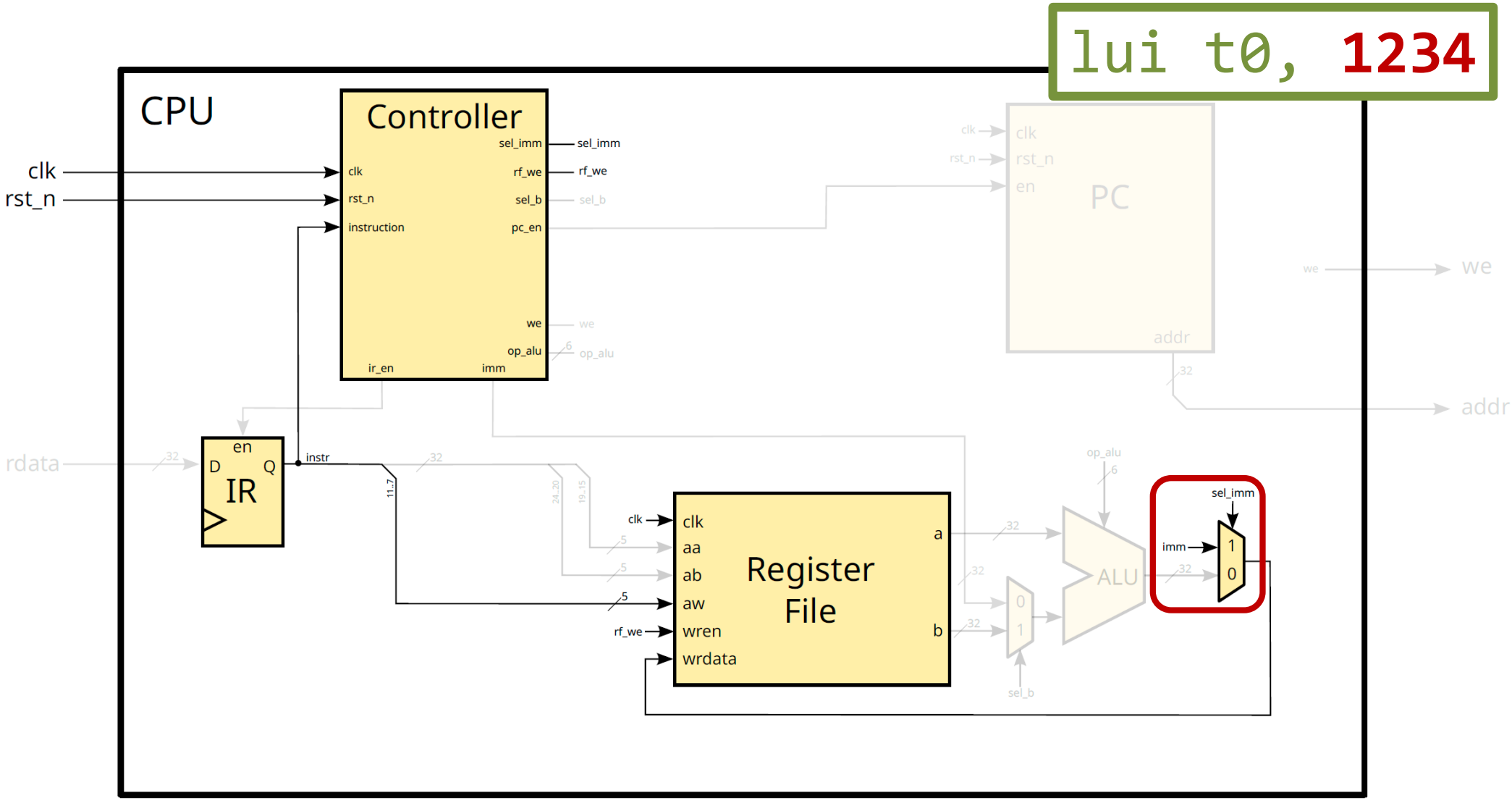
R-Type Instructions Need a Second Operand

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

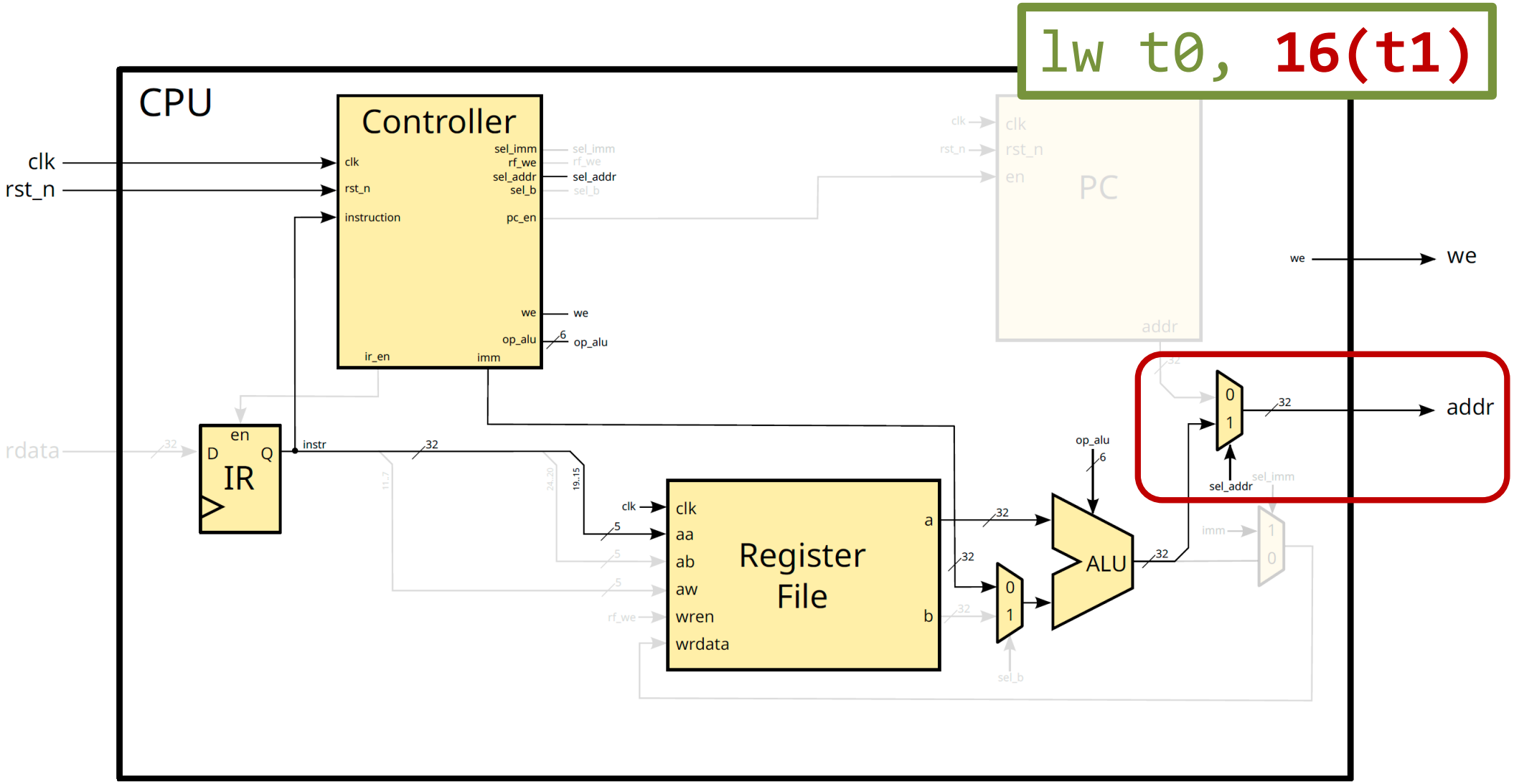


U-Type Instructions Write an Immediate

```
lui t0, 1234
```

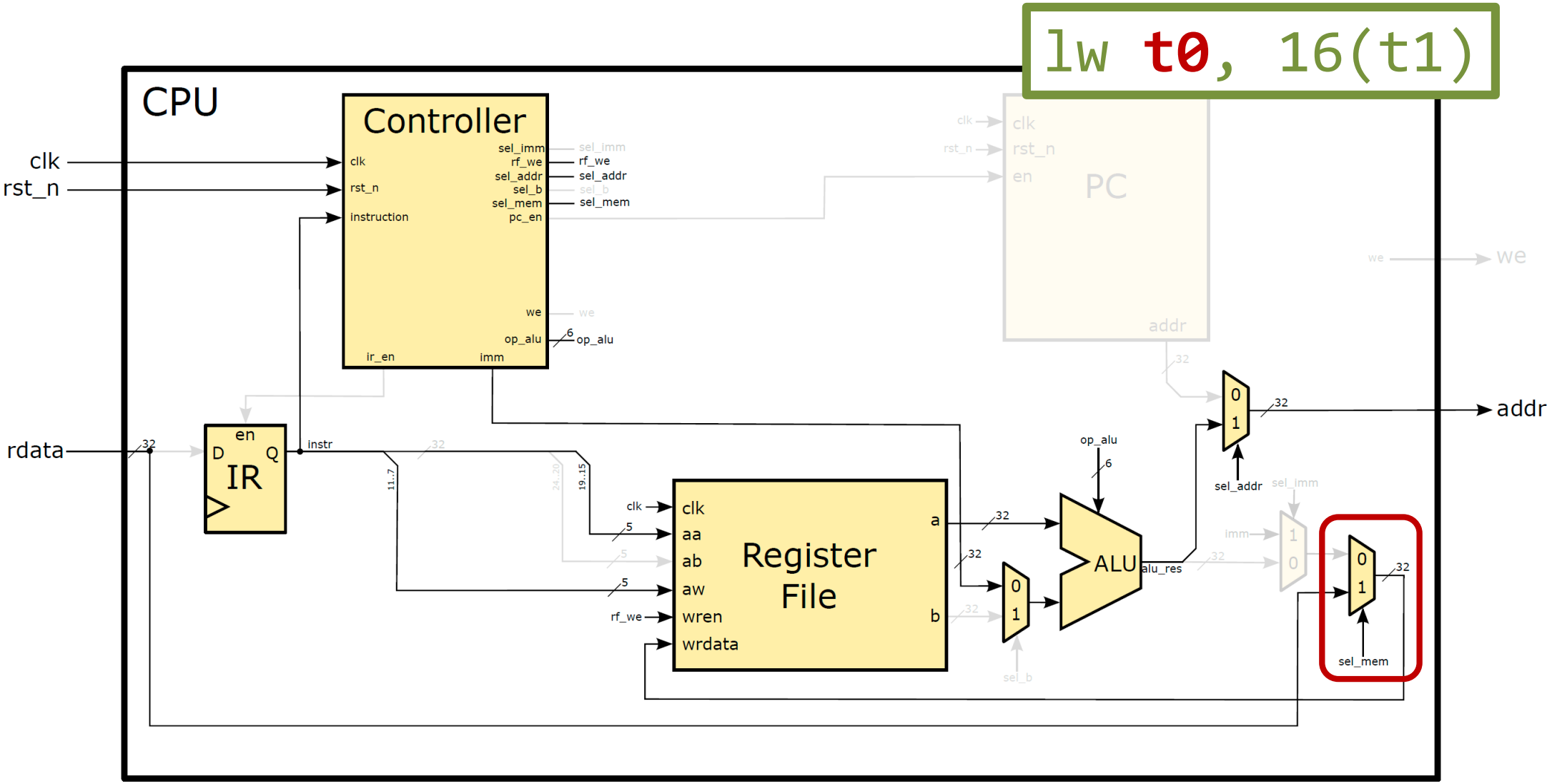


Load and Stores Produce a Memory Address



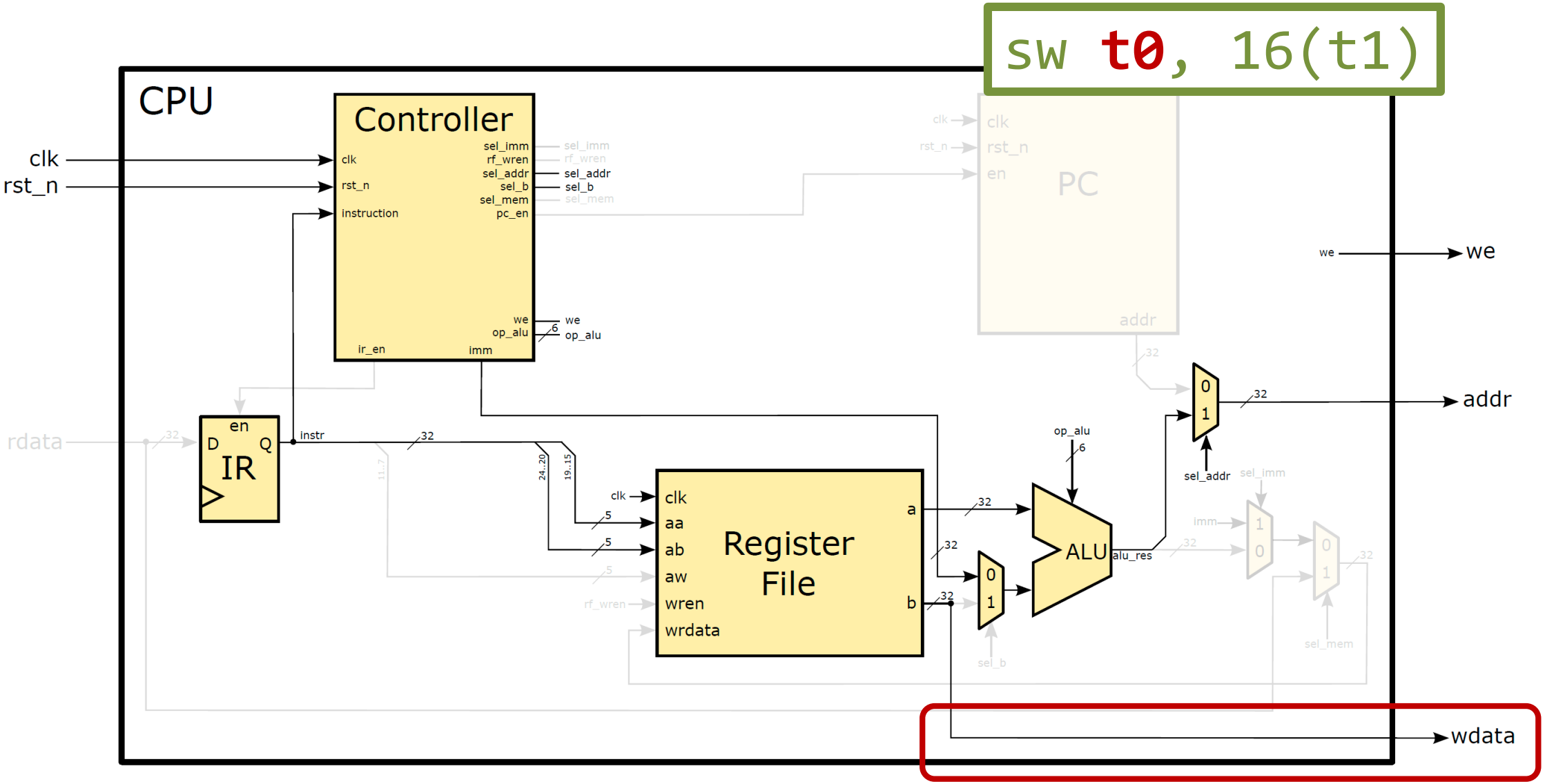
Loads Write the Read Data into the RF

```
lw t0, 16(t1)
```

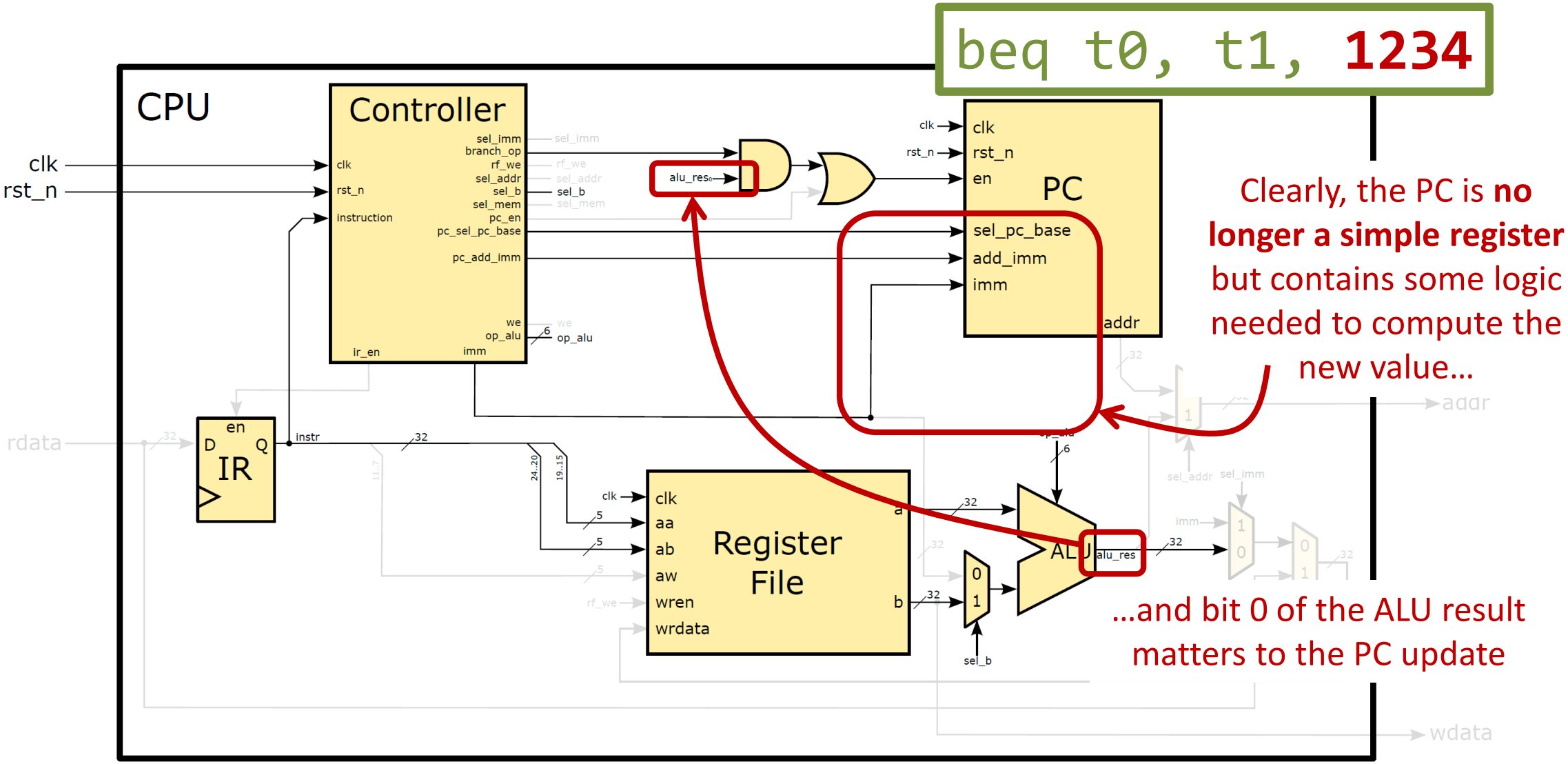


Stores Send an Operand to Memory

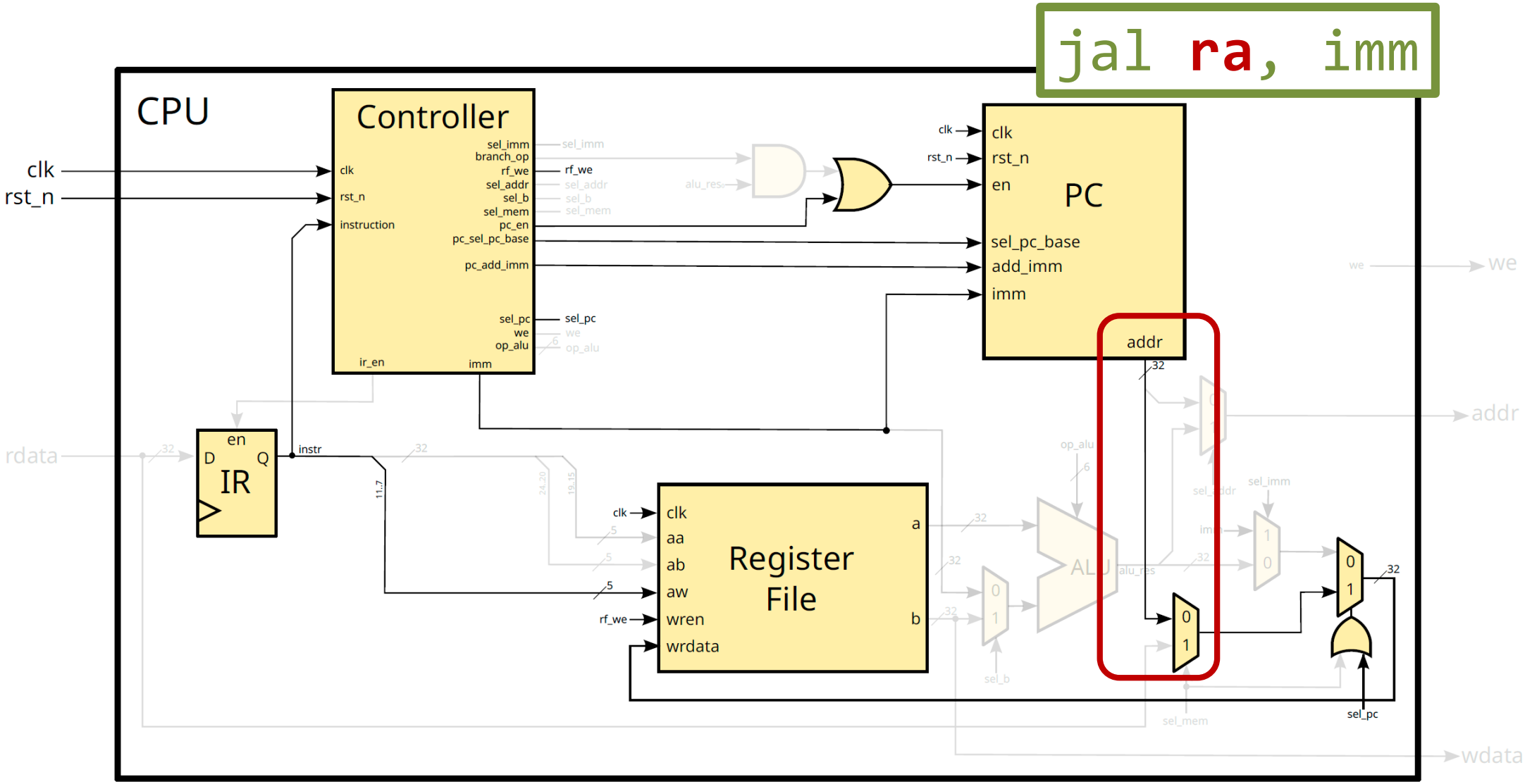
```
sw t0, 16(t1)
```



Branches Need to Write an Offset to the PC

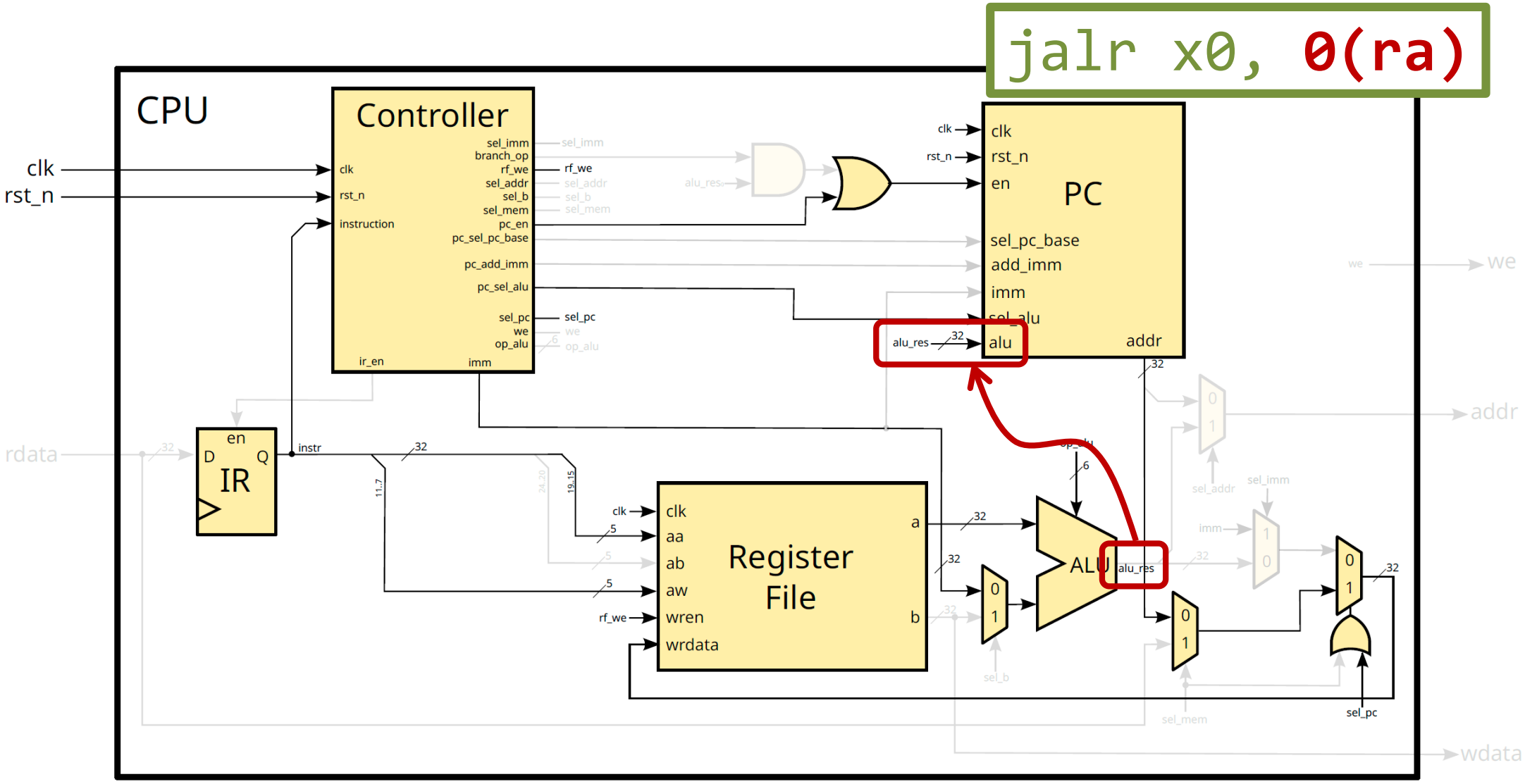


`jal` Needs to Store PC + 4 in the RF

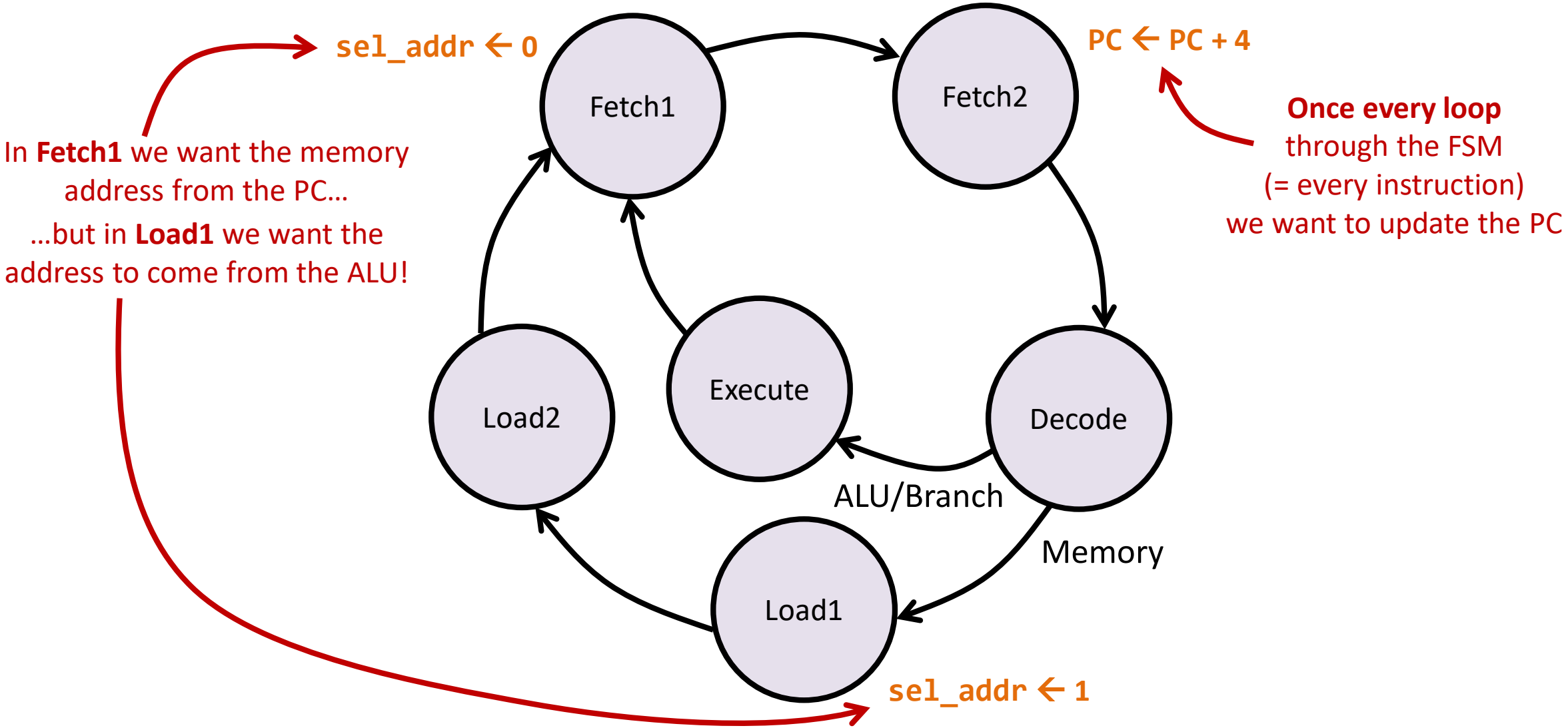


Jumps Need to Write an Address to the PC

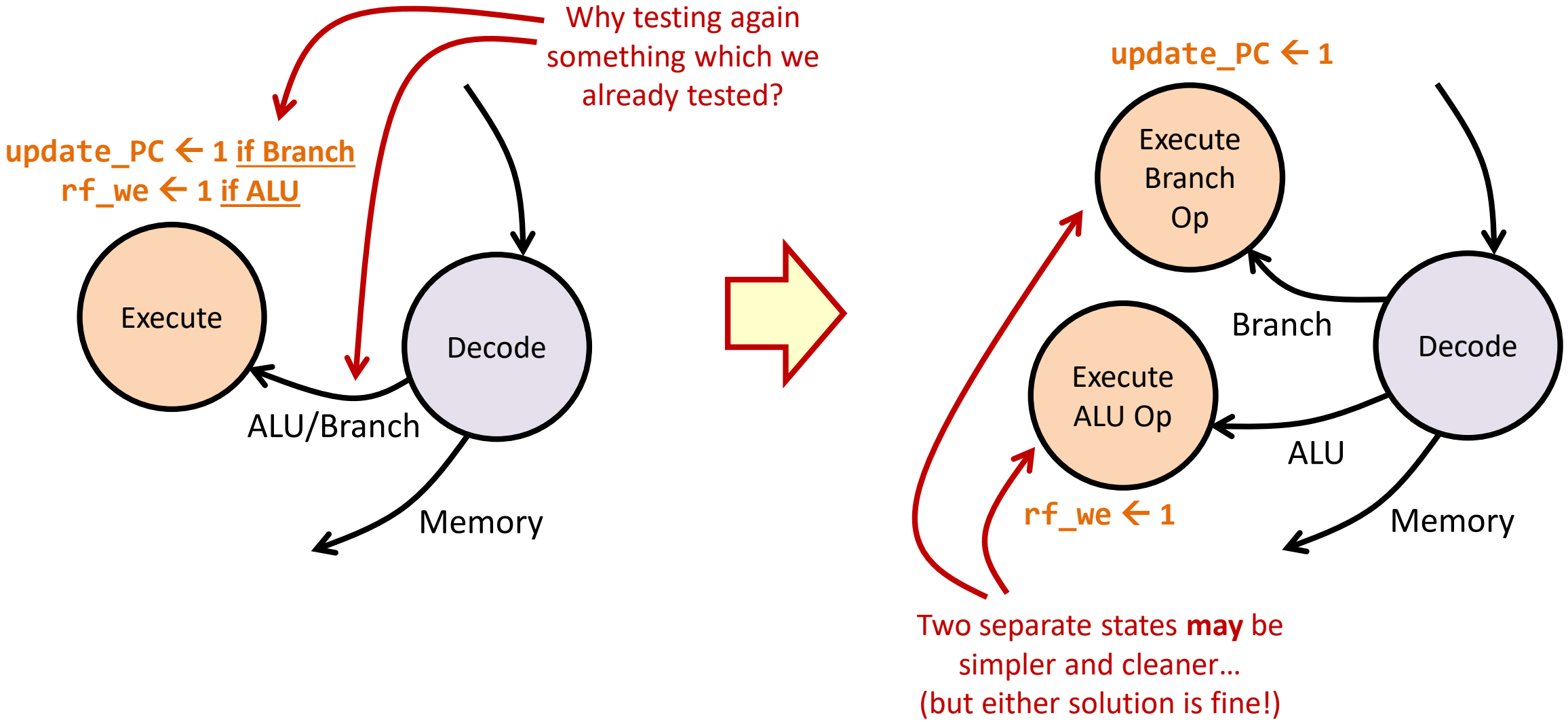
```
jalr x0, 0(ra)
```



Time to Go Back to the FSM!



And We Can Still Change the FSM…

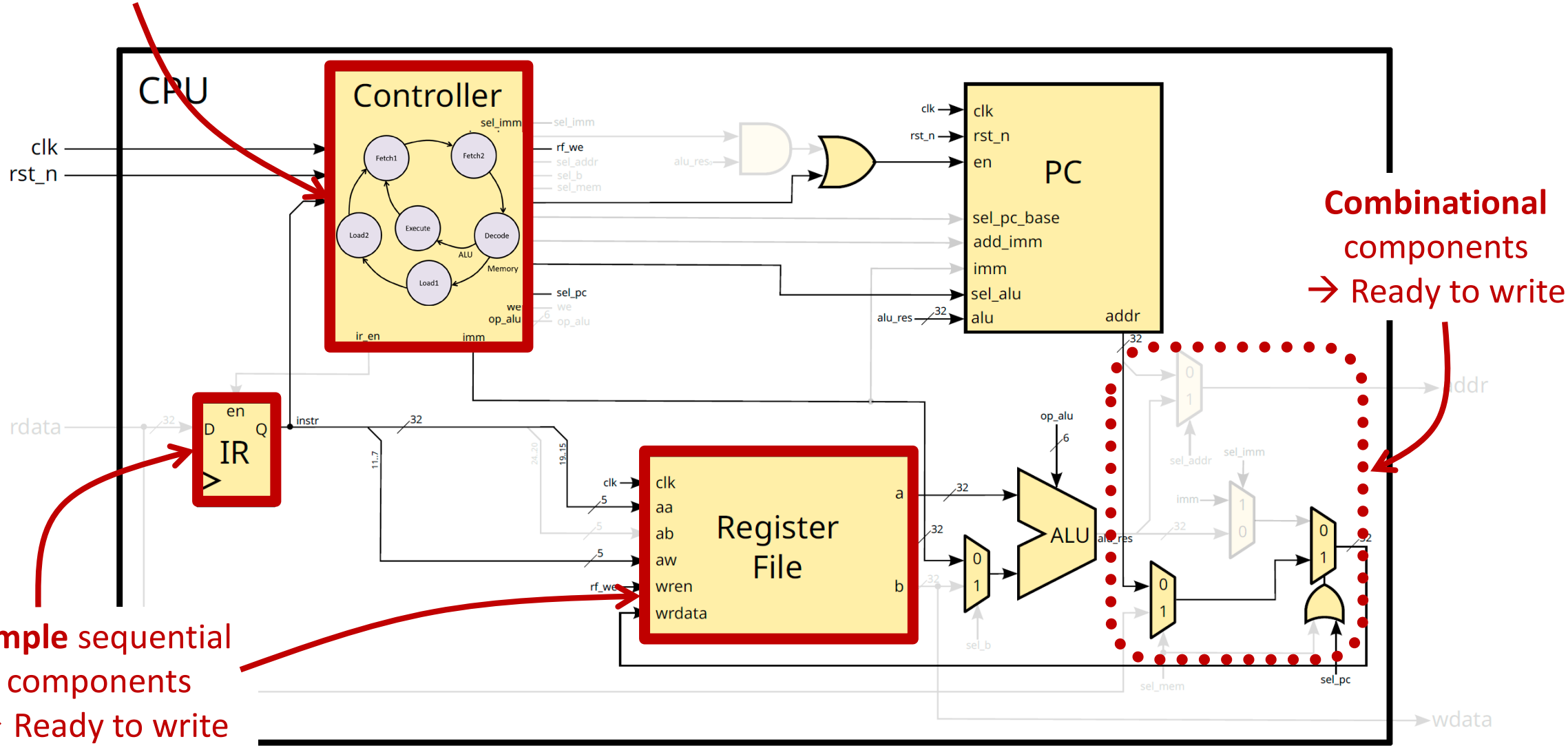


Do Not Write Verilog until Done!

- Verilog and VHDL are **HDL**s → Hardware **Description** Languages
- Describe something only when it is perfectly **clear**:
  - You have **drawn a diagram** like the one of the previous slide
  - You have clearly identified **combinational** and **sequential** blocks
- Always **decompose complex sequential** blocks
  - Describe **only** sequential blocks that are **simple registers** (e.g., IR)
  - Draw **hierarchical diagrams** until sequential blocks are trivial
- Use a **hierarchical approach** (much as in programming) and use your diagrams to guide the creation of modules (e.g., PC)

Write Only Simple Stuff

Formal **FSM**
→ Ready to write

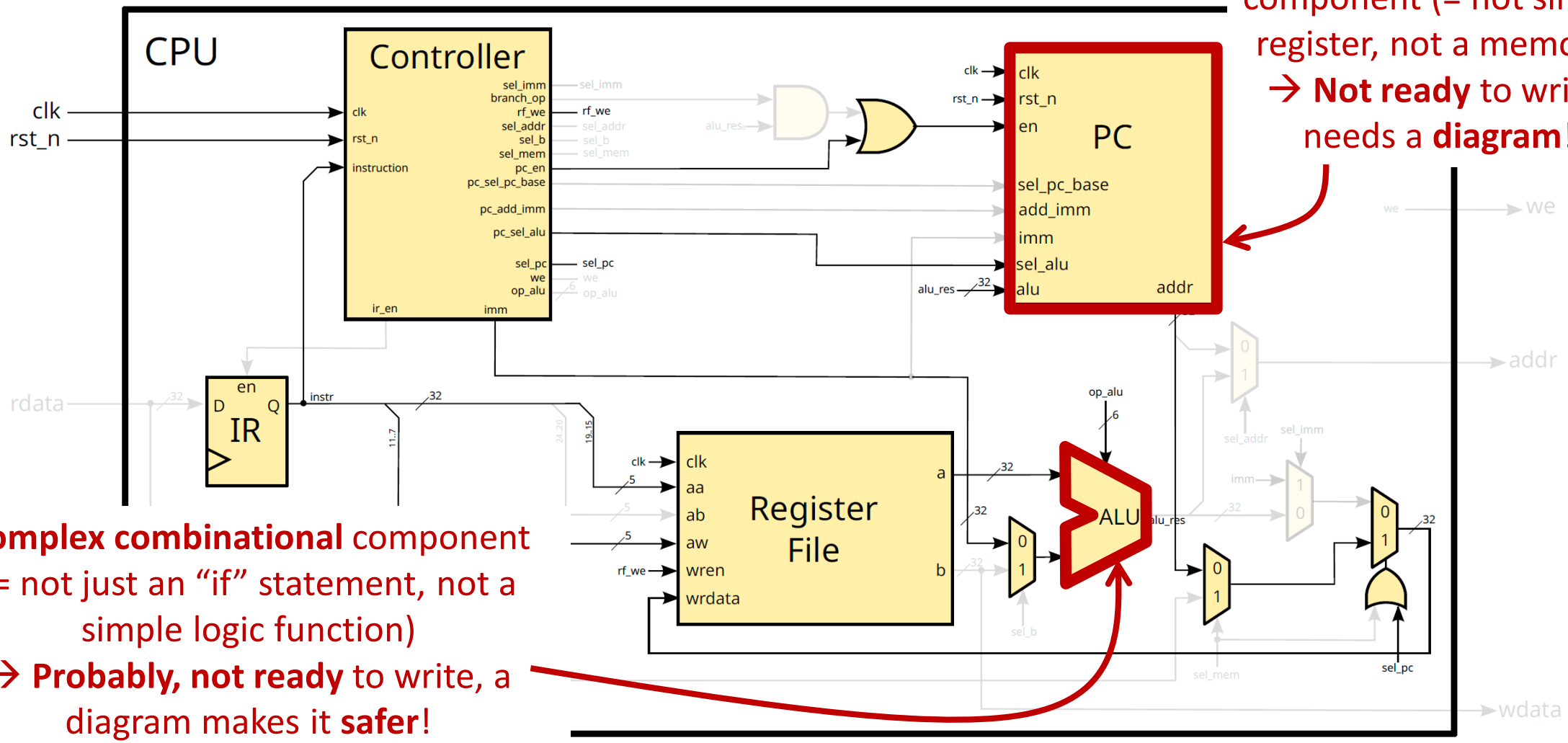


Combinational components
→ Ready to write

Simple sequential components
→ Ready to write

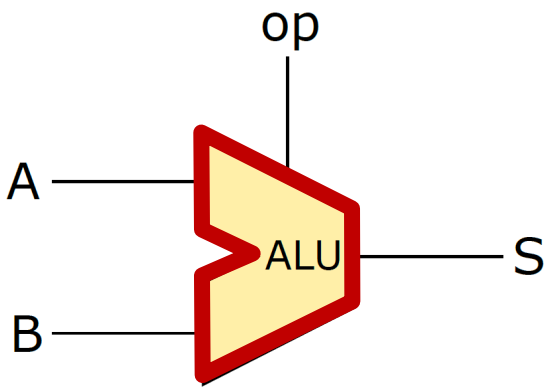
“Open the Box” of Complex Stuff

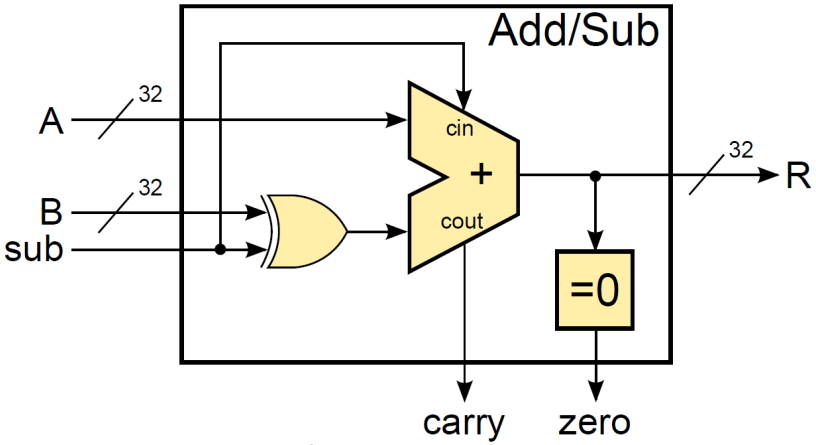
Complex sequential component (= not simple register, not a memory)
→ **Not ready** to write, needs a **diagram**!

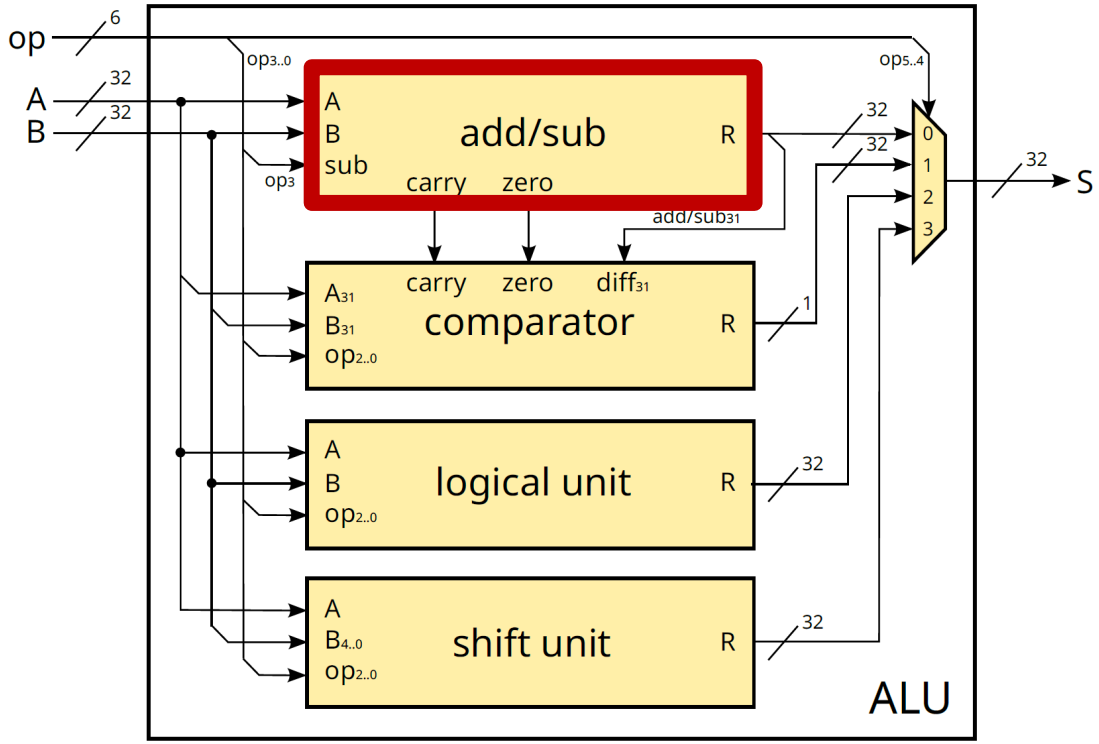


Complex combinational component (= not just an “if” statement, not a simple logic function)
→ **Probably, not ready** to write, a diagram makes it **safer**!

Detail Complex Combinational Modules







Write Verilog by Sticking to Basic Patterns

Combinational

- **`always@ (*)`** is used to describe a block with combinational logic. The * symbol is used in the sensitivity list to trigger the block whenever any of the inputs are changed; therefore, the outputs reflect the inputs change.

Verilog **guidelines** in Moodle

```
always @ (*)
  begin
    if (a)
      y = ~b;
    else
      y = b;
  end
```

You can write **much more complex combinational** blocks (e.g., next state in FSMs)…

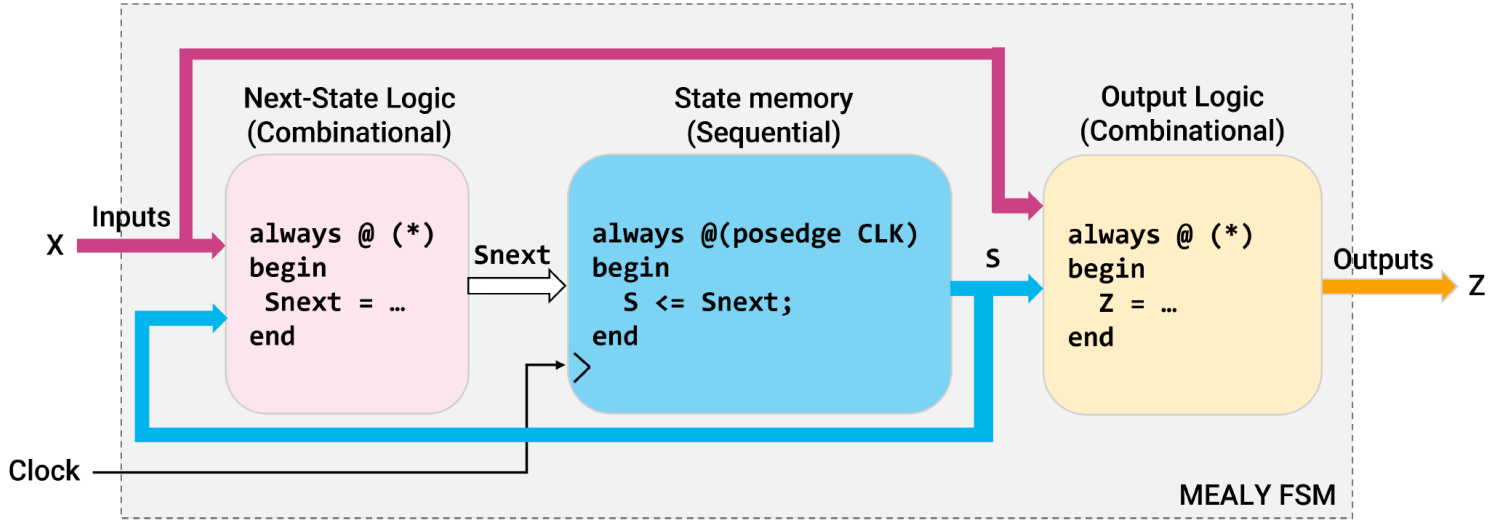
```
always @ (posedge clk)
  begin
    if (reset == 1)
      q <= 0;
    else if ((enable1 == 1) && (enable2 == 1))
      q <= d;
  end
```

…but this is just about the **most complex sequential** block you want to write! (only **registers** and **counters**)

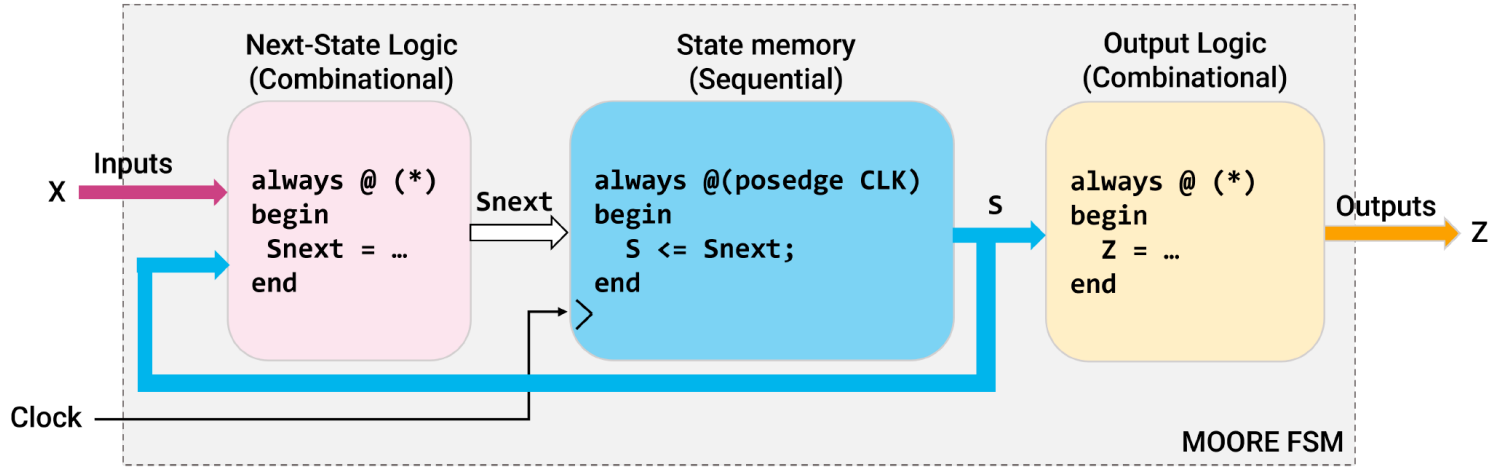
Sequential

- **`always@ (posedge clk)`** is used to describe a block with sequential logic i.e., has flip-flops. The two keywords **posedge** and **negedge** determine whether the active clock edge of the flip-flops is the rising or the falling clock edge respectively.

Three `always` Blocks per FSM, Always!

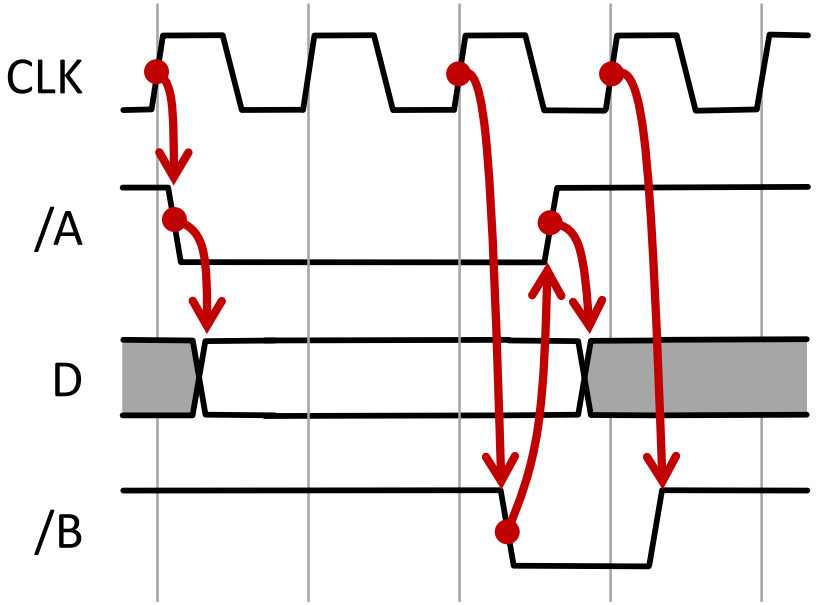


Mealy



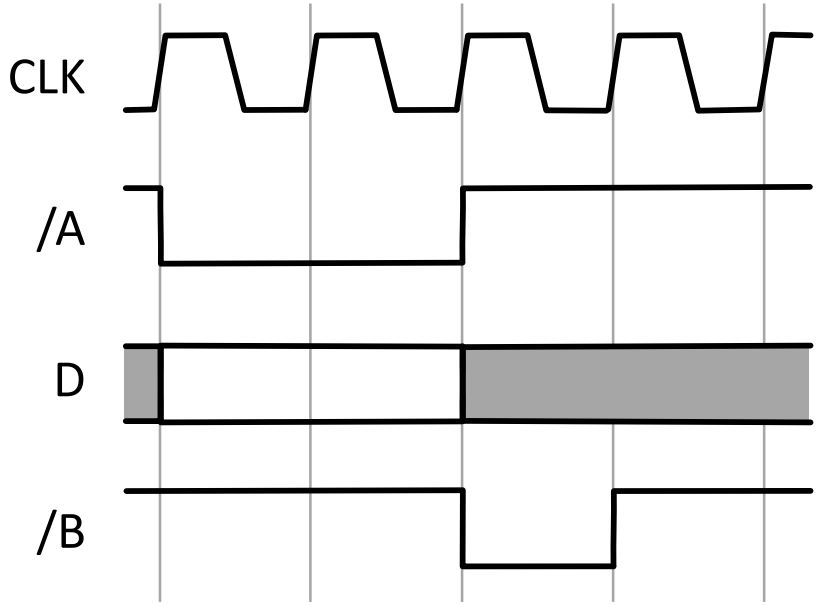
Moore

Beware of Zero-Delay Simulation!



Real delays suggest **causality**

A good timing diagram
even **shows causality explicitly**
with arrows



Alas, all this is **lost** in
zero-delay simulation

Consider redrawing by hand
in difficult debugging situations

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Chapter 4**; only **Section 4.1** to **4.5**