EECS 370 - Lecture 11

Multi-Cycle Data Path

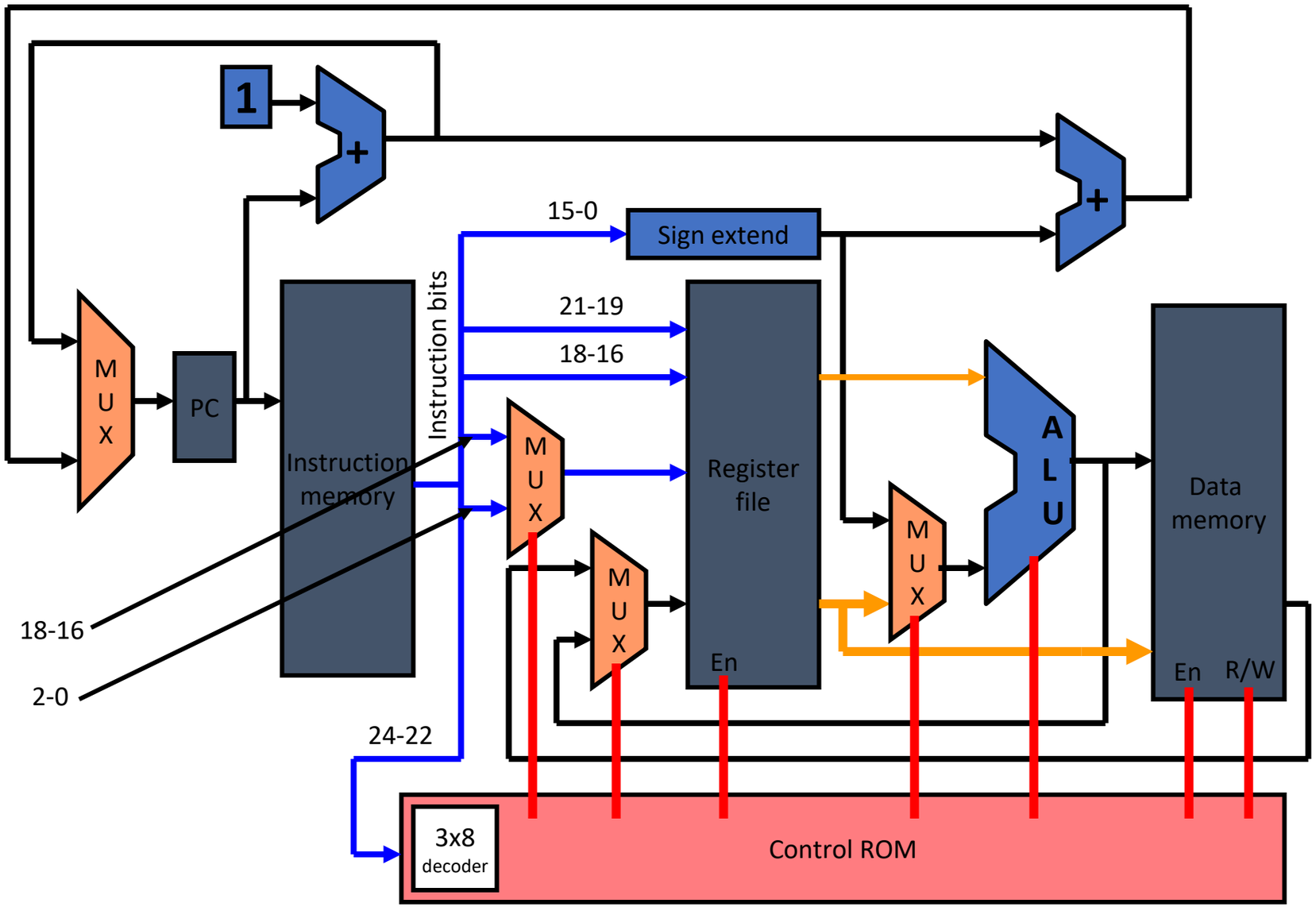
Announcements

- P2
  - Three parts: part a is due **Thursday**
- HW 2
  - Posted on website, due next **Mon**
- Midterm exam **Thu 10/9 6-8 pm**
  - Multi-cycle (covered Thursday) will be last topic covered
  - Sample exams on website
  - You can bring 1 sheet (double sided is fine) of notes
  - We will provide LC2K encodings + ARM cheat sheet
  - Calculator that doesn't connect to internet is recommended
- **Prof. Bleier will be giving Thursday's lecture as I will be out of town**

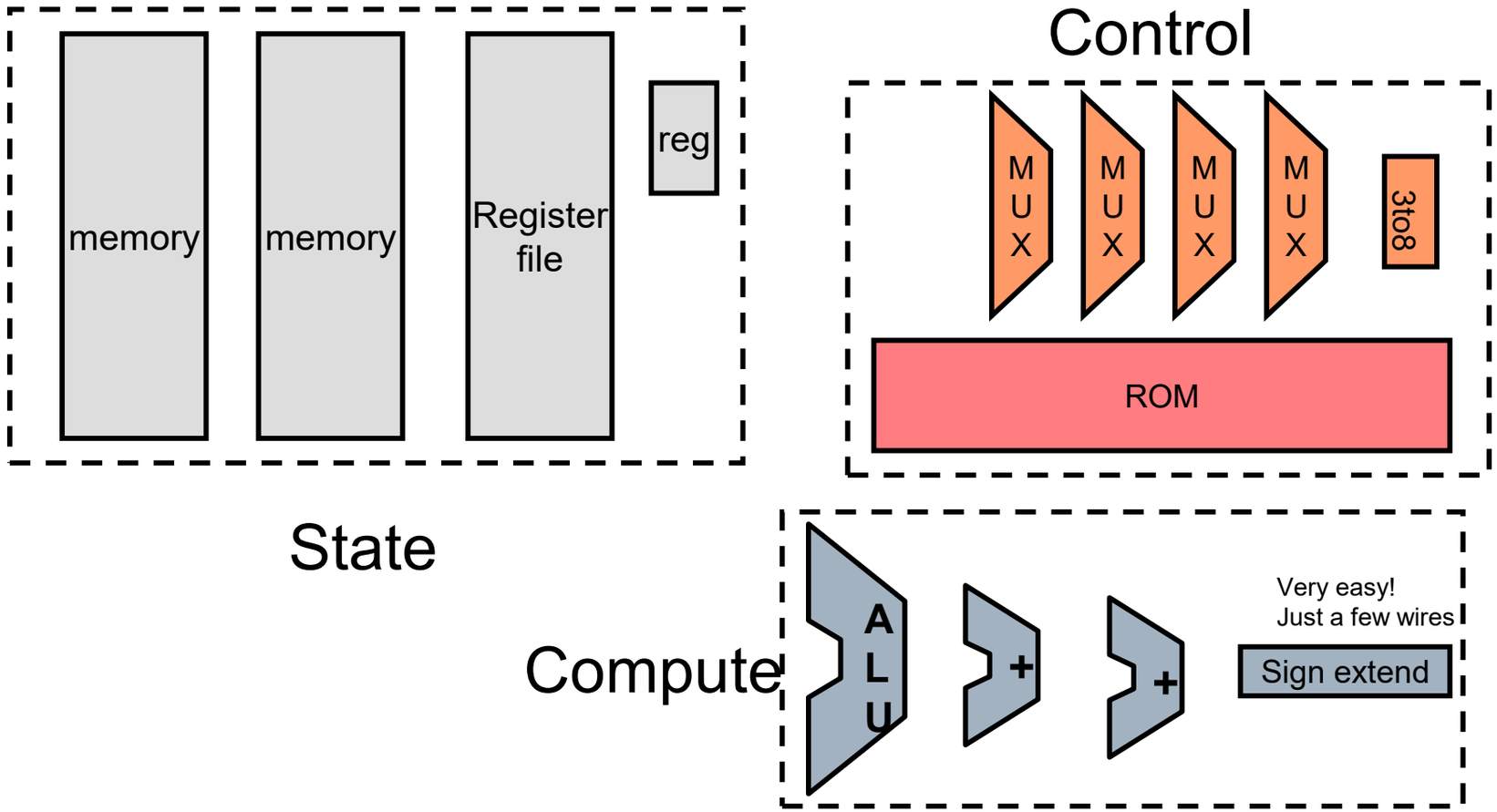
Single-Cycle Processor Design

- General-Purpose Processor Design
  - Fetch Instructions
  - Decode Instructions
    - Instructions are input to control ROM
  - ROM data controls movement of data
    - Incrementing PC, reading registers, ALU control
  - Clock drives it all
  - Single-cycle datapath: Each instruction completes in one clock cycle

LC2K Datapath Implementation

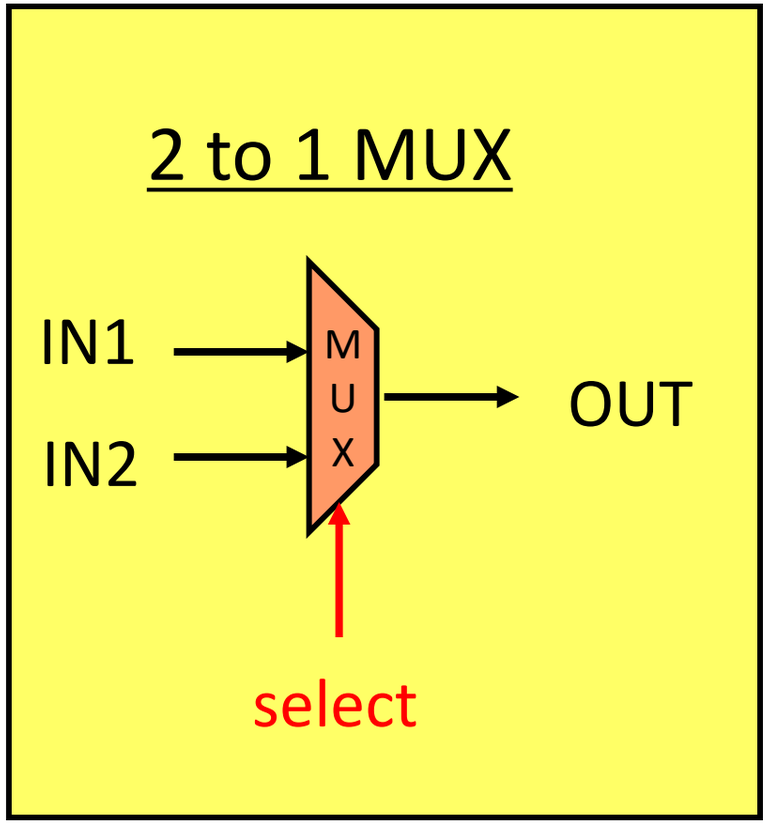


Building Blocks for the LC2K



Here are the pieces, go build yourself a processor!

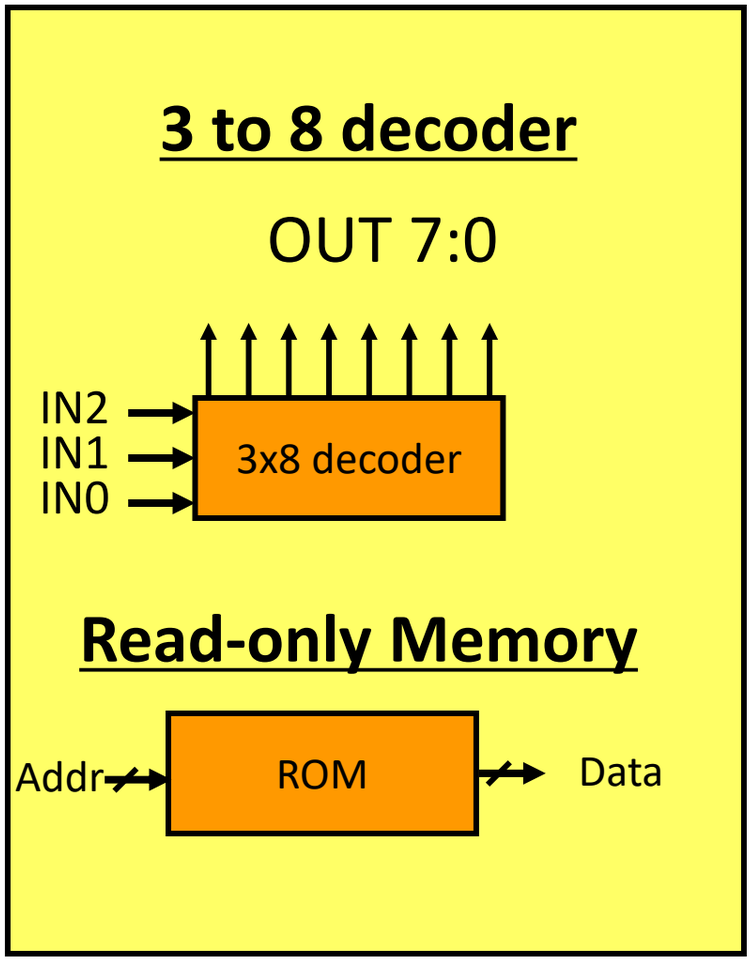
Control Building Blocks (1)



Connect one of the inputs to OUT based on the value of select

```
If (! select)
    OUT = IN1
Else
    OUT = IN2
```

Control Building Blocks (2)



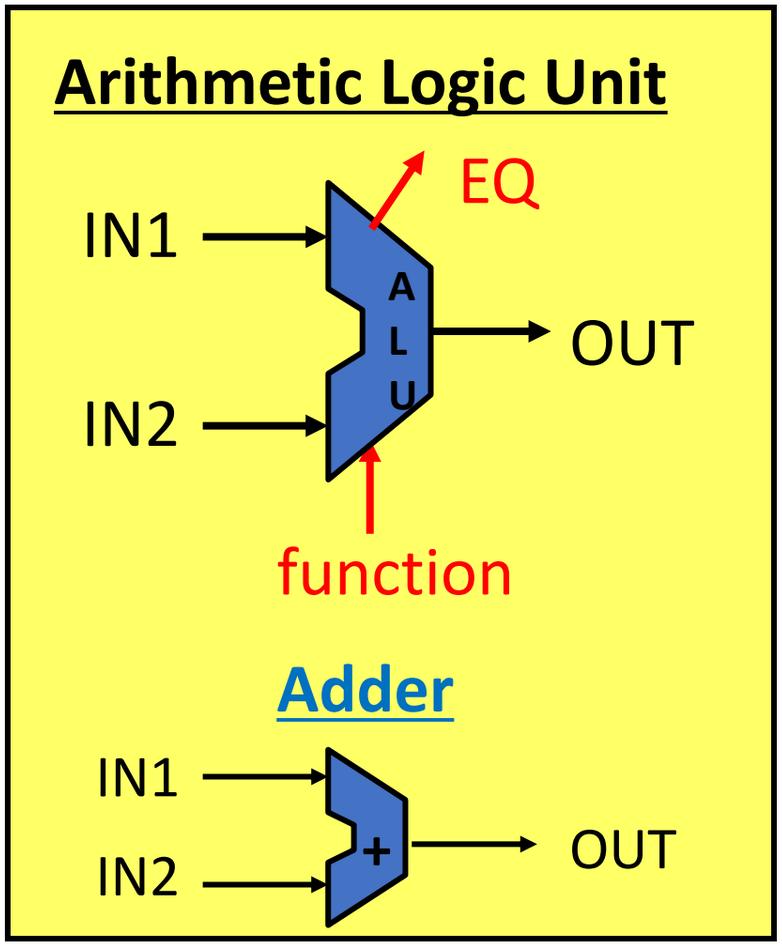
Decoder activates one of the output lines based on the input

IN	OUT
210	76543210
000	00000001
001	00000010
010	00000100
011	00001000
etc.	

ROM stores preset data in each location

- Give address, get data.

Compute Building Blocks (1)



Perform basic arithmetic functions

OUT = f(IN1, IN2)
EQ = (IN1 == IN2)

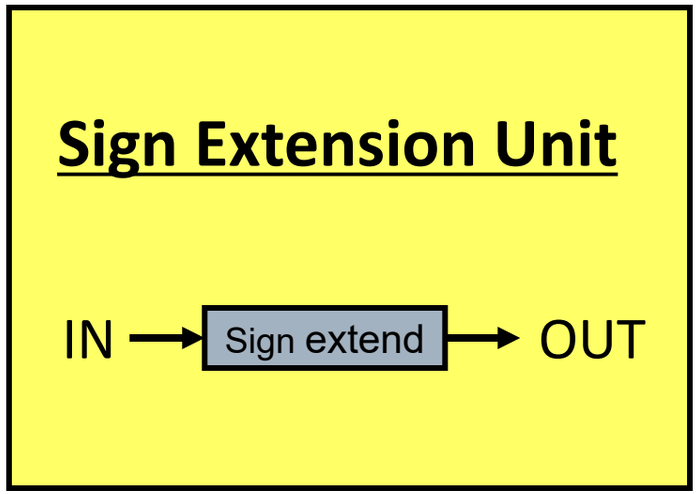
For LC2K:

f=0 is add
f=1 is nor

For other processors, there are many more functions.

Just adds

Compute Building Blocks (2)



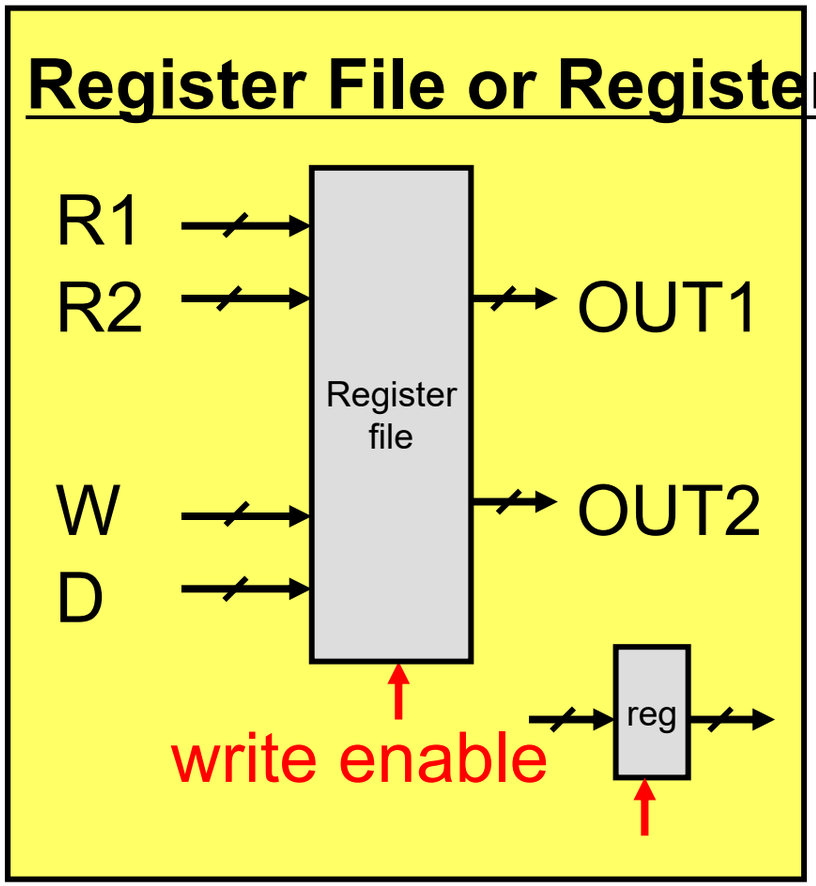
Sign extend (SE) input by replicating the MSB to width of output

OUT(31:0) = SE(IN(15:0))

OUT(31:16) = IN(15)
OUT(15:0) = IN(15:0)

Useful when compute unit is wider than data

State Building Blocks (1)



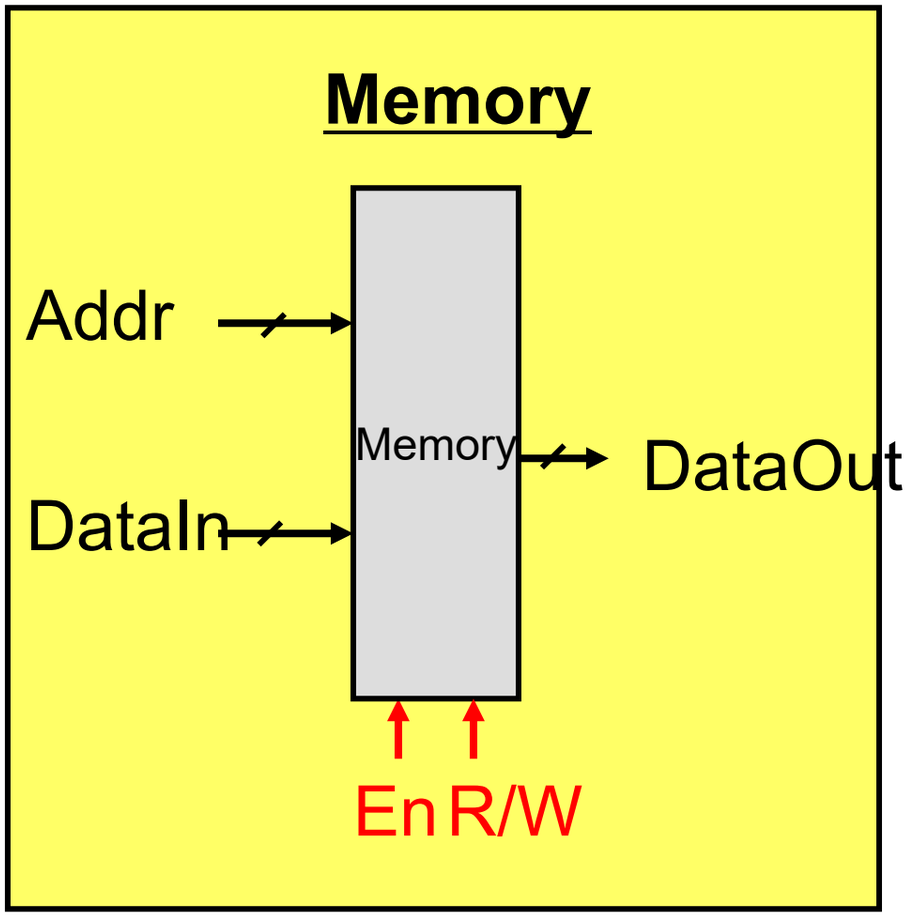
Small/fast memory to store temporary values

- **n** entries (LC2 = 8)
- **r** read ports (LC2 = 2)
- **w** write ports (LC2 = 1)

* Ri specifies register number to read
* W specifies register number to write
* D specifies data to write

Poll: How many bits are Ri and W in LC2K?

State Building Blocks (2)



Slower storage structure to hold large amounts of stuff.

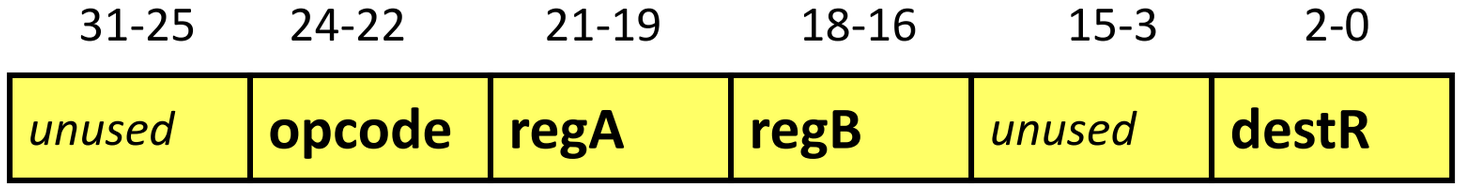
Use 2 memories for LC2K

* Instructions
* Data
* 65,536 total words

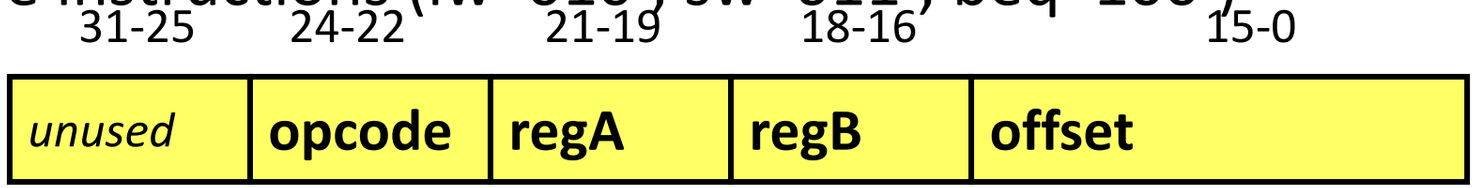
Recap: LC2K Instruction Formats

- Tells you which bit positions mean what

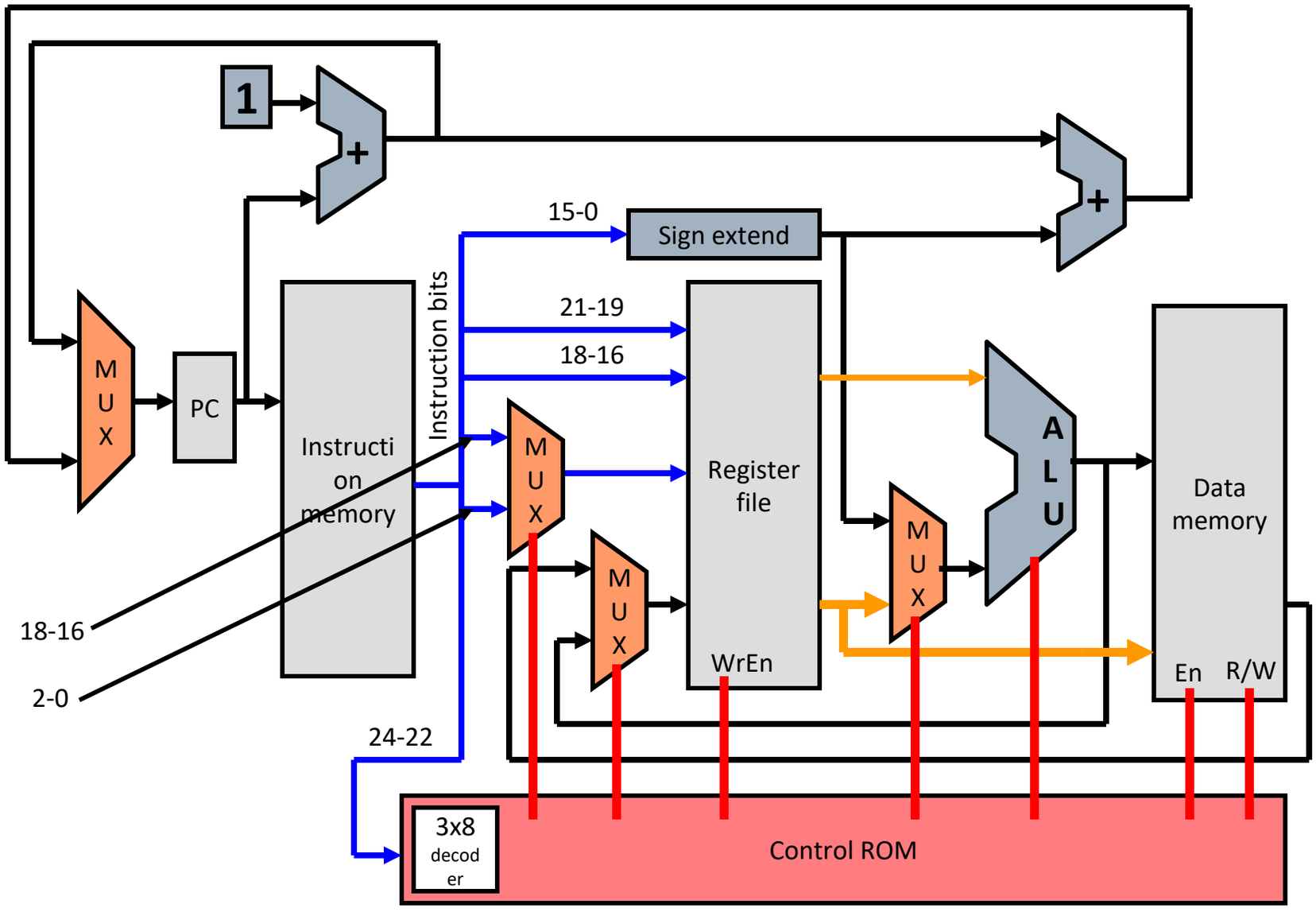
- R type instructions (add ‘000’, nor ‘001’)



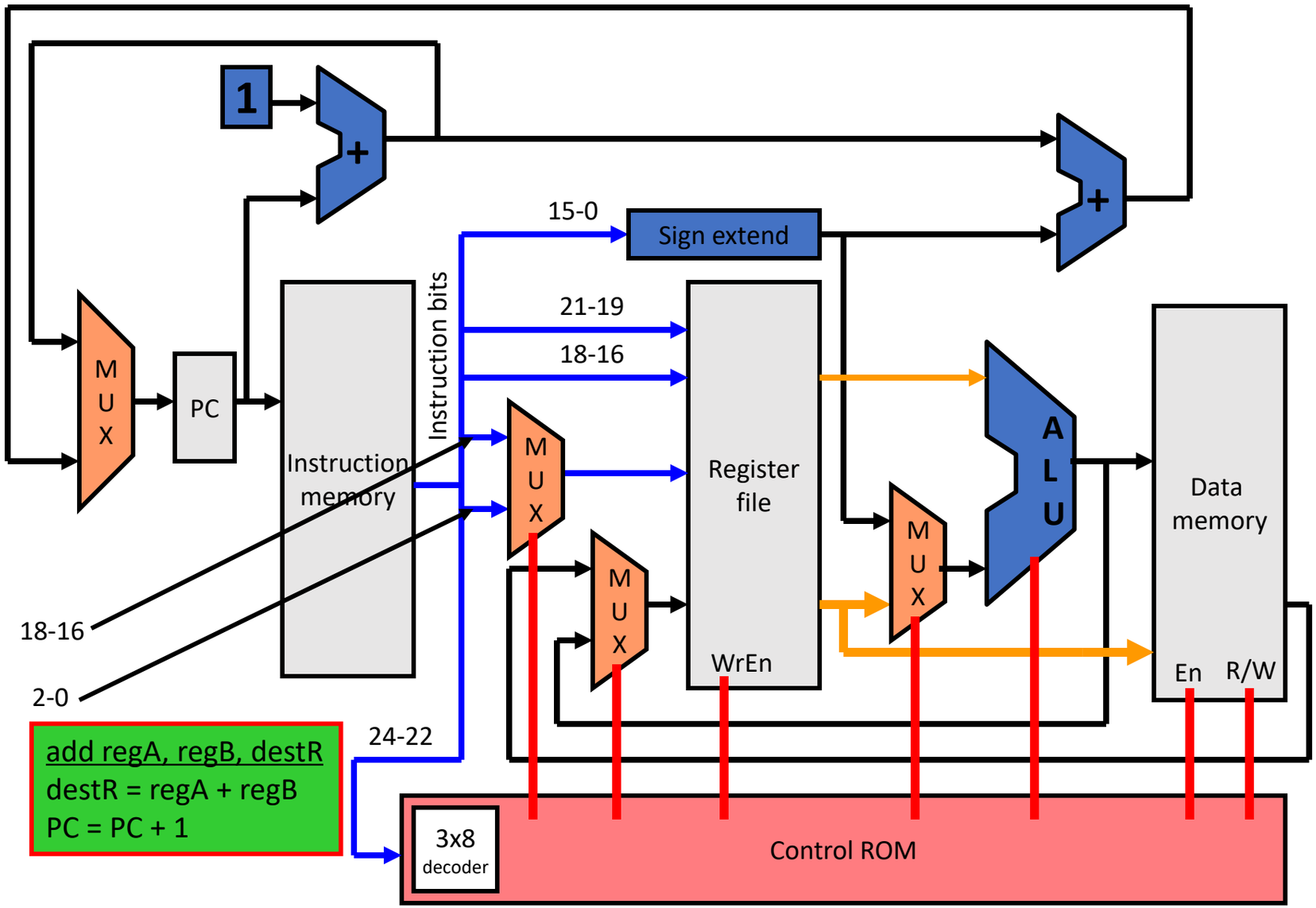
- I type instructions (lw ‘010’, sw ‘011’, beq ‘100’)



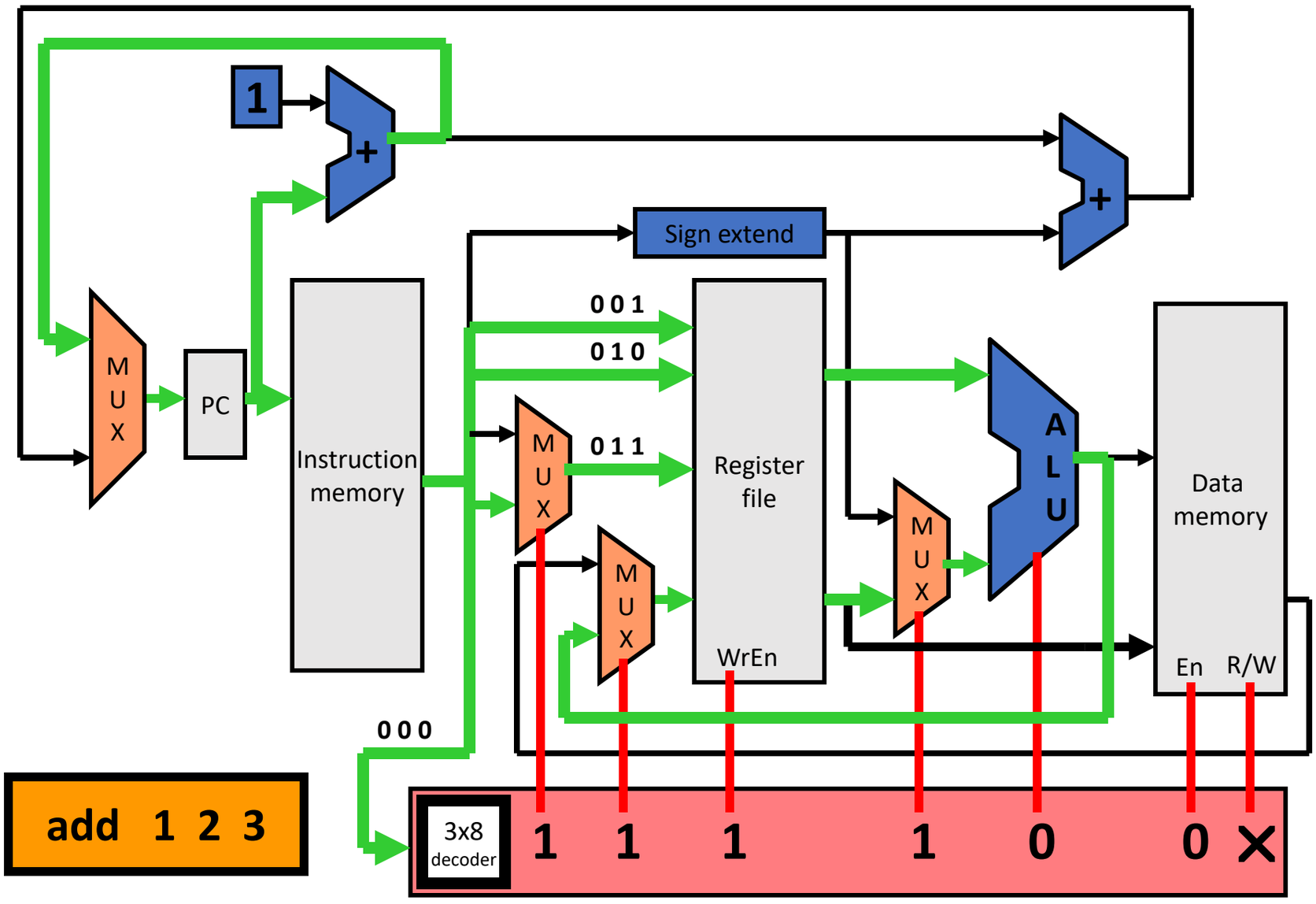
LC2K Single-Cycle Datapath Implementation



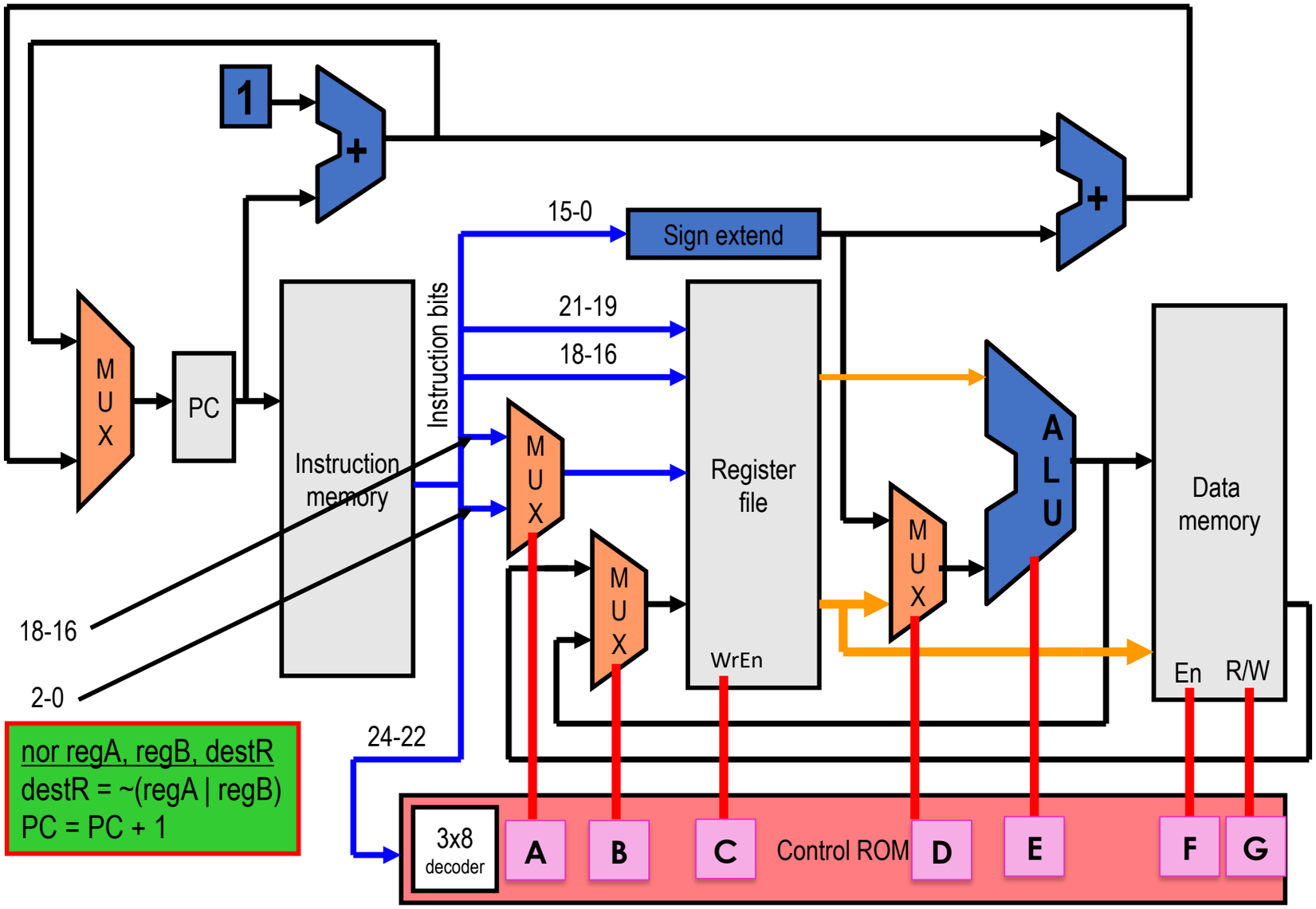
Executing an ADD Instruction



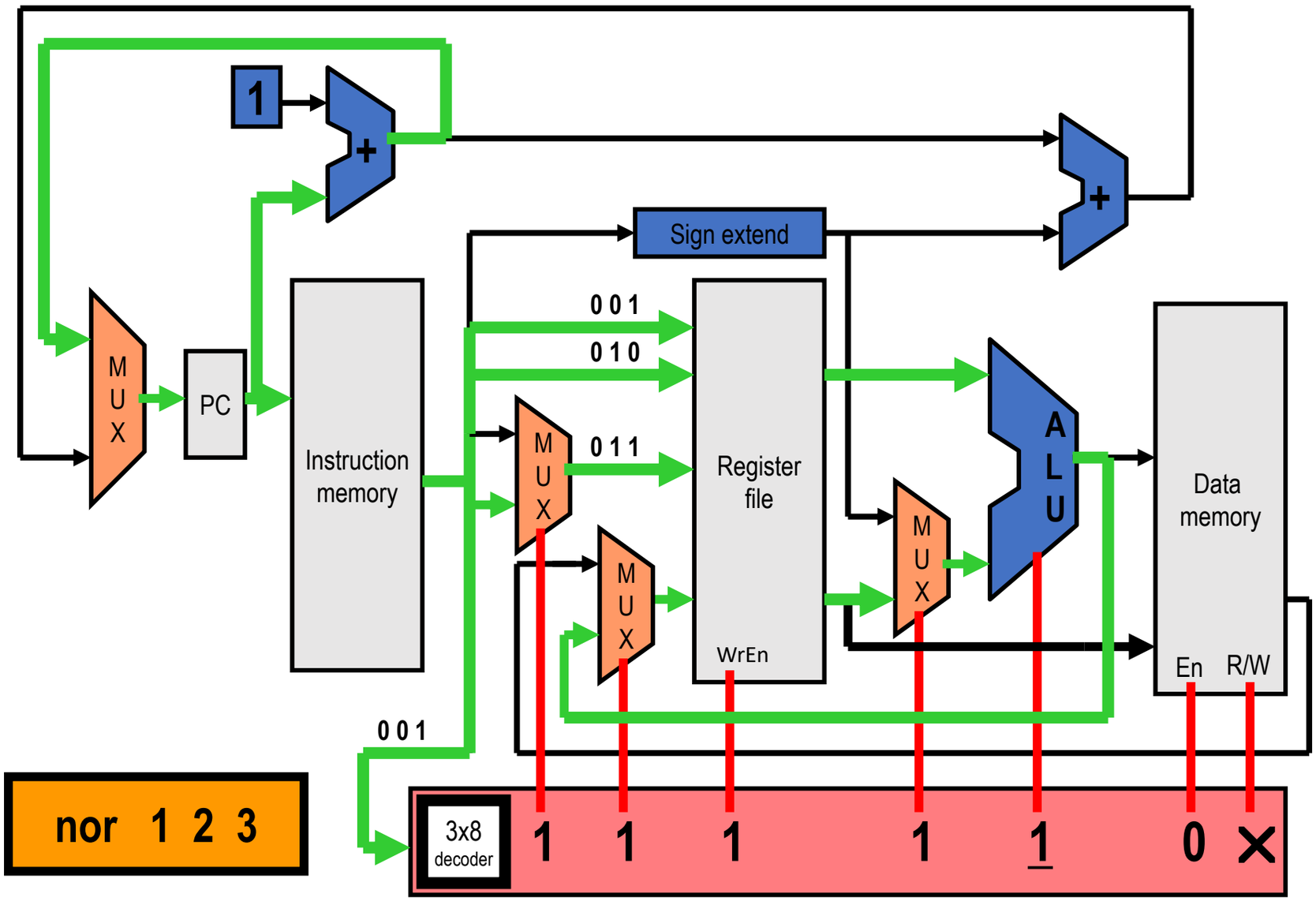
Executing ADD Instruction on LC2K



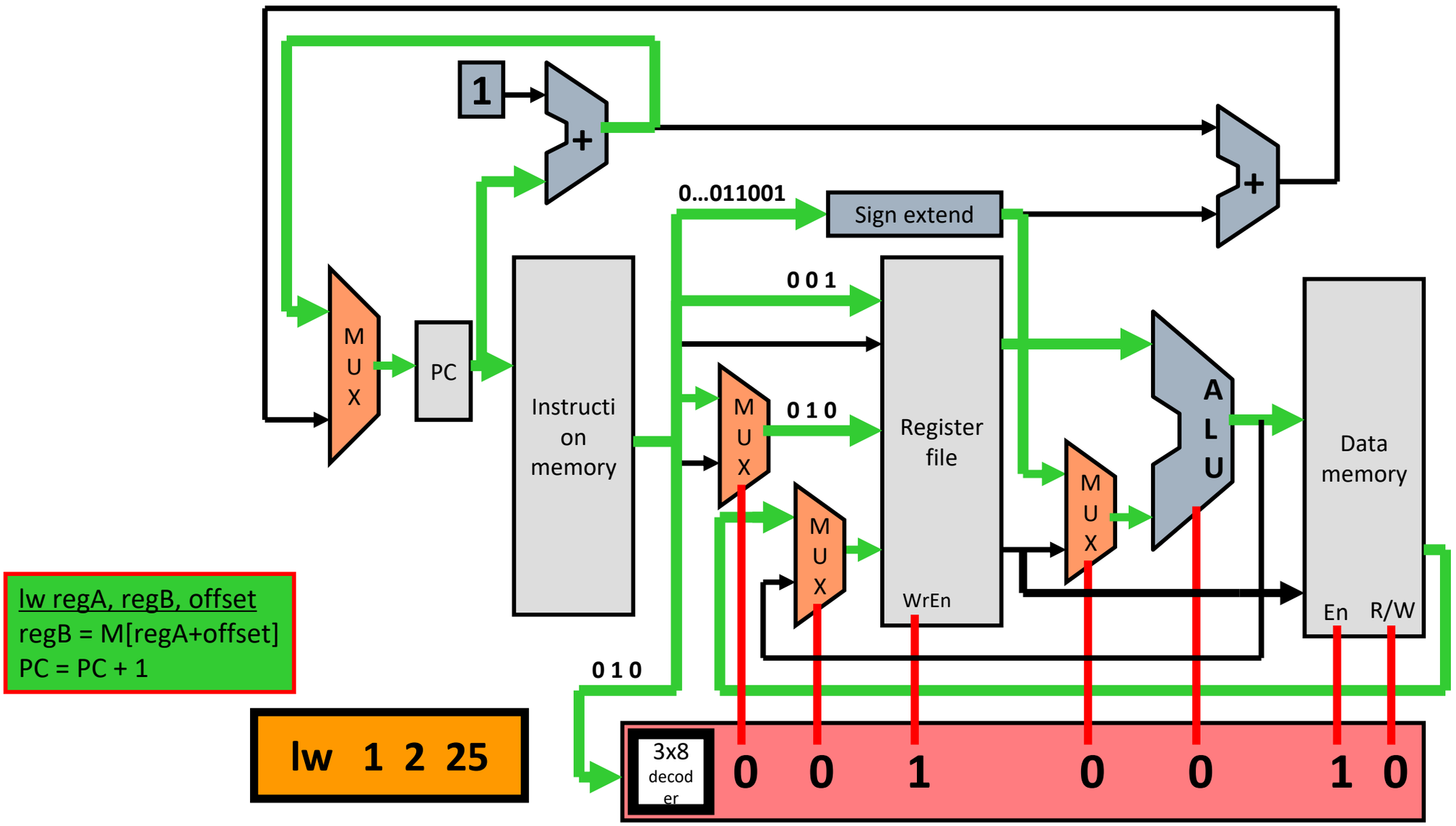
Executing NOR Instruction on LC2K



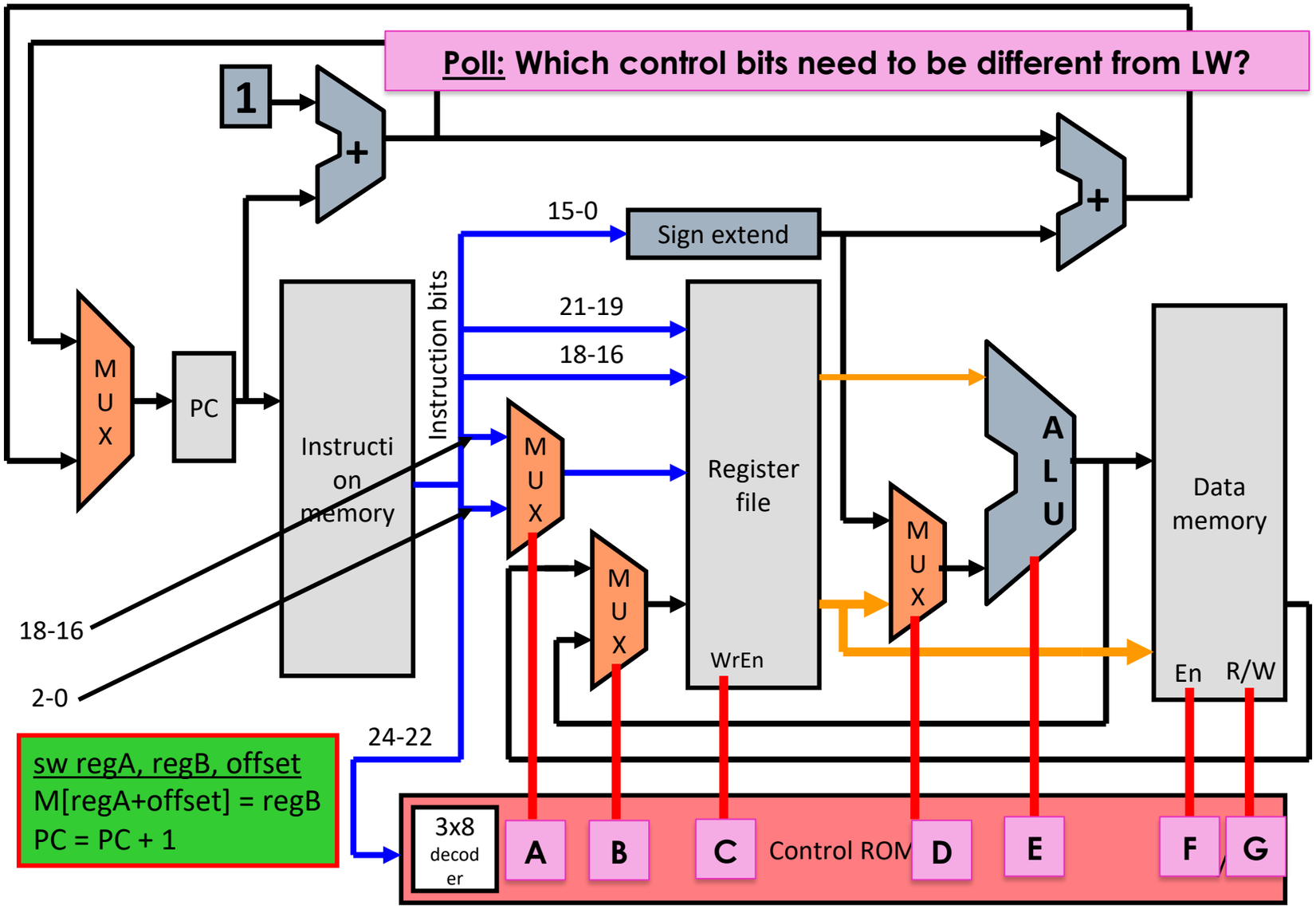
Executing NOR Instruction on LC2K



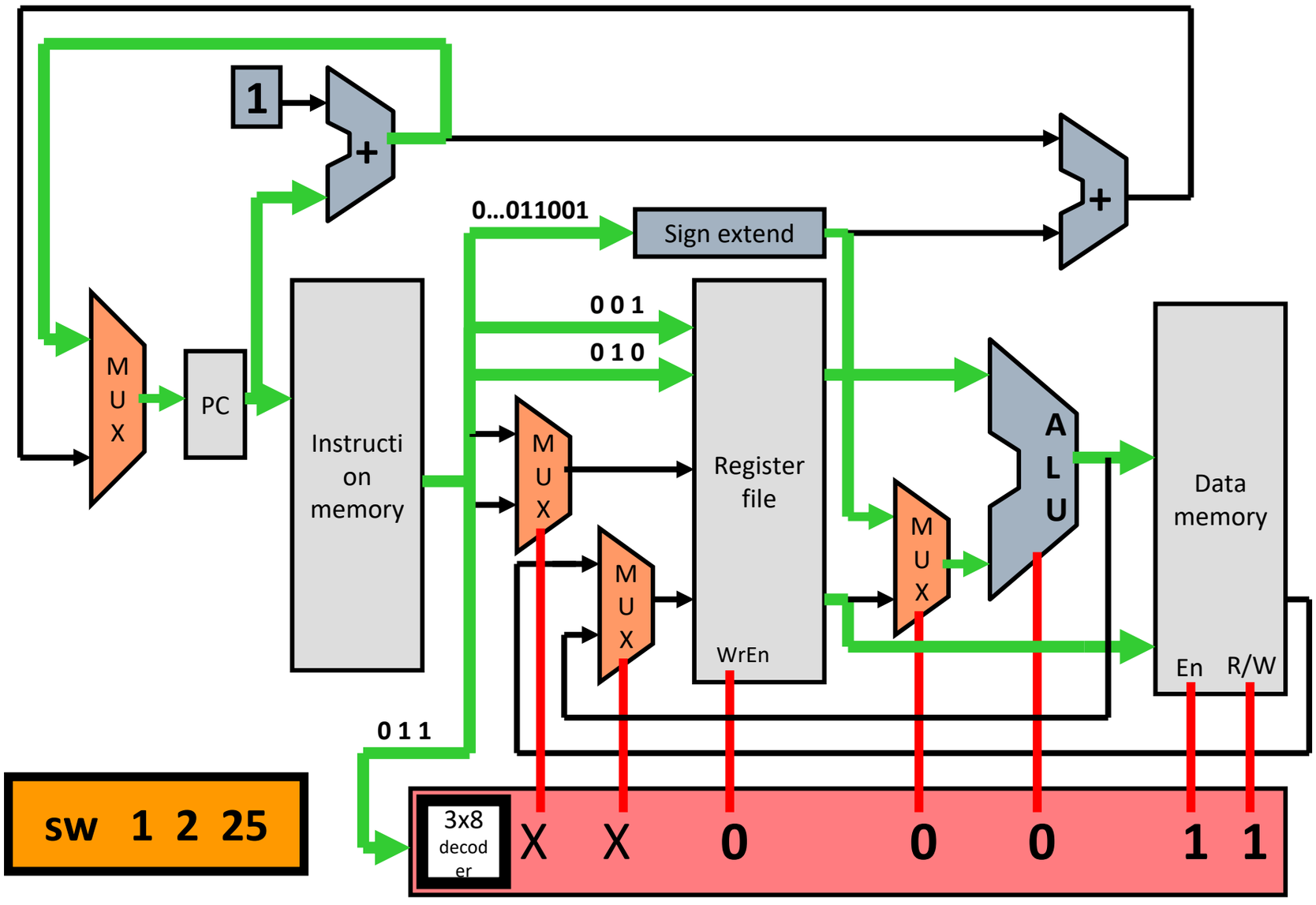
Executing LW Instruction on LC2K



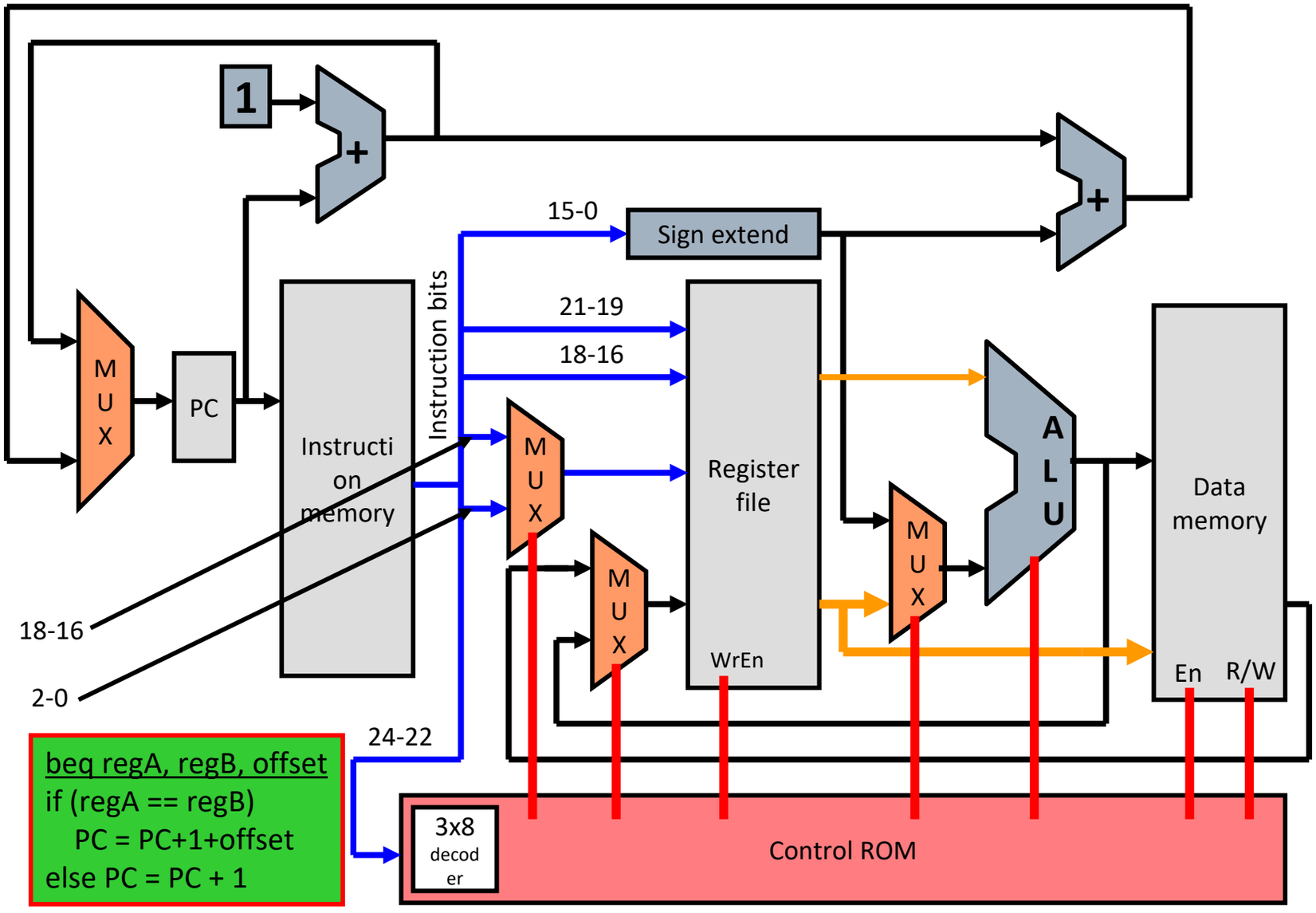
Executing SW Instruction on LC2K



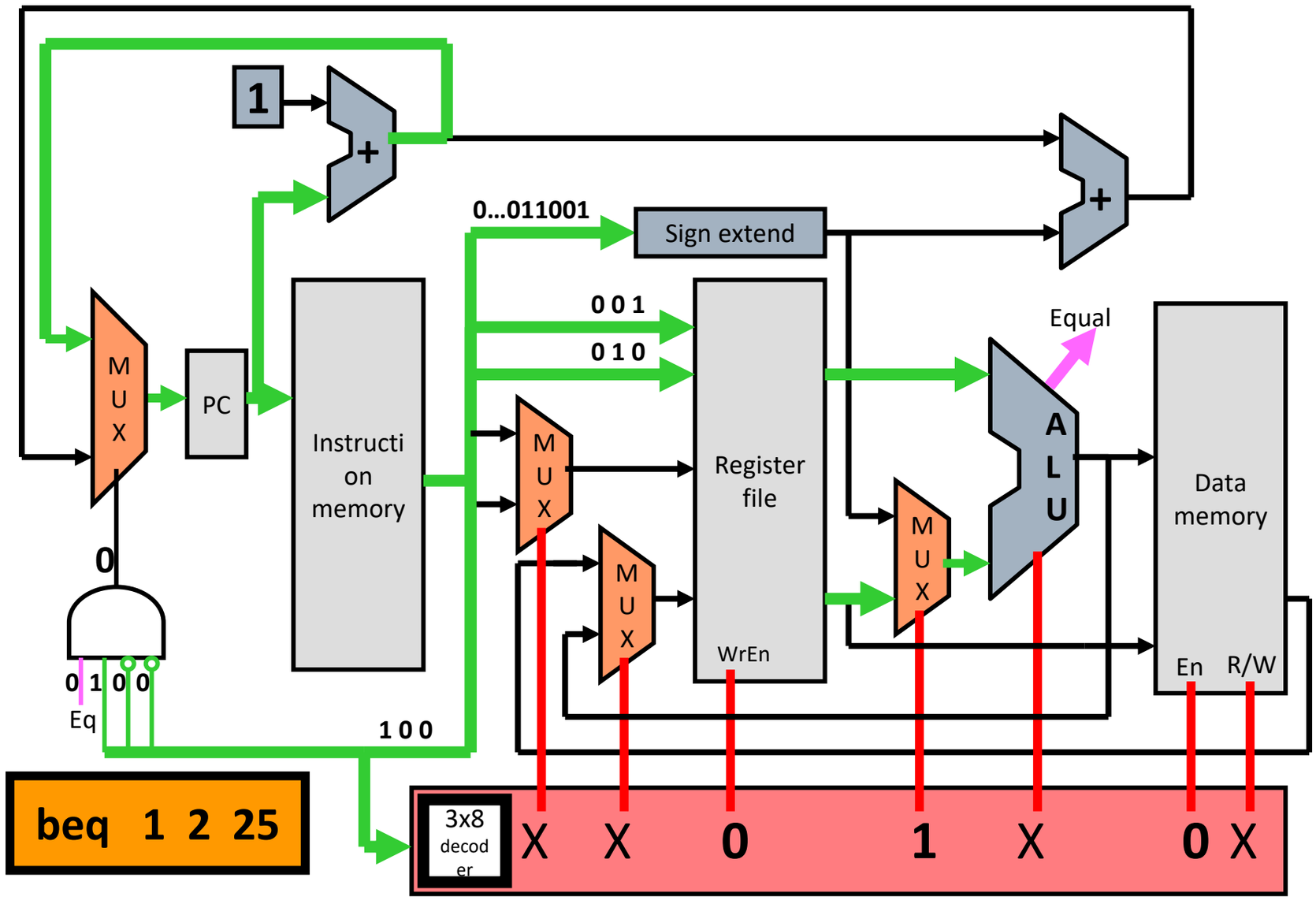
Executing SW Instruction on LC2K



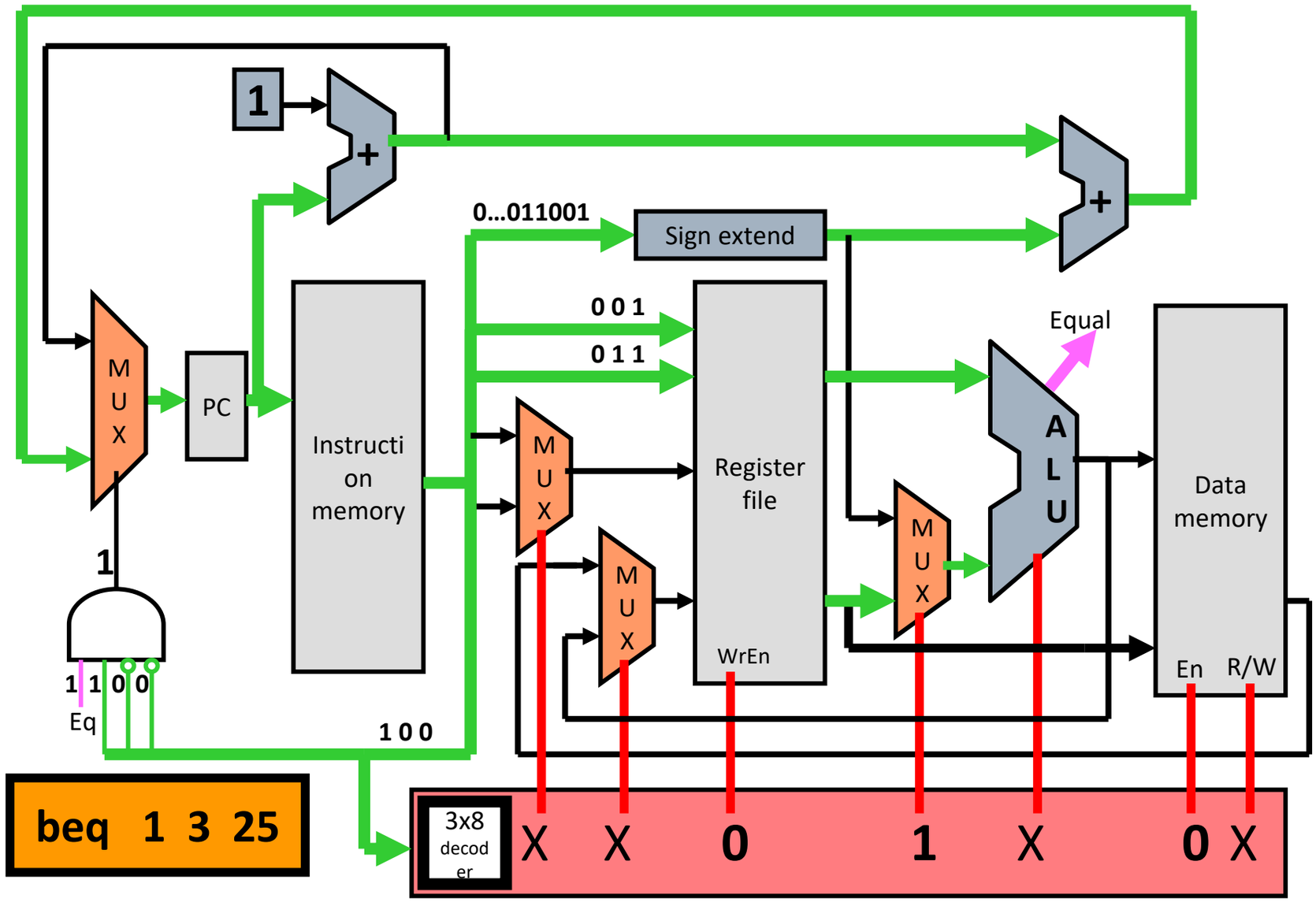
Executing BEQ Instruction on LC2K



Executing “not taken” BEQ Instruction



Executing “taken” BEQ Instruction



So Far, So Good

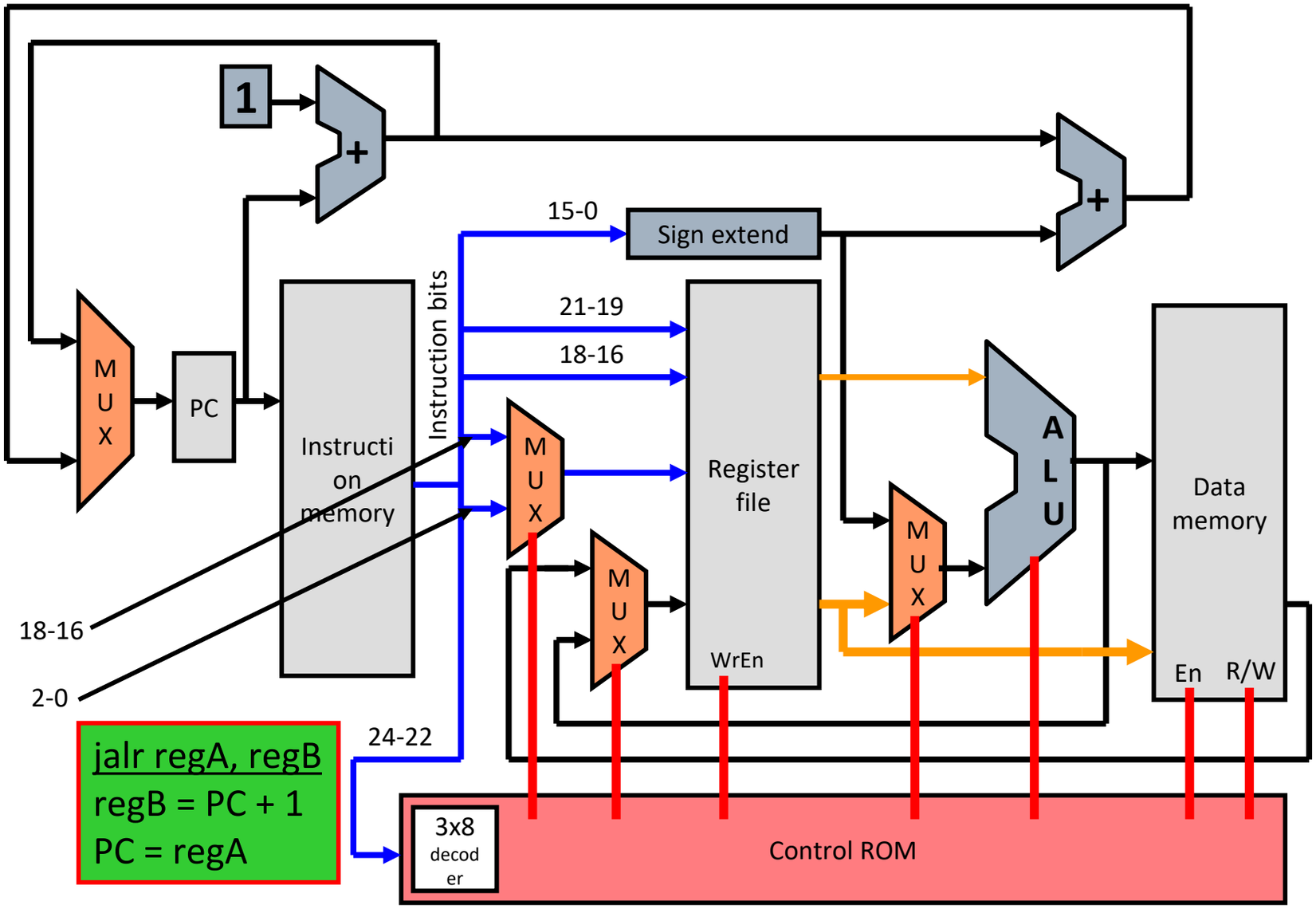
- Every architecture seems to have at least one "ugly" instruction
  - Something that doesn't elegantly fit in with the hardware we've already included

- For LC2K, that ugly instruction is JALR
  - It doesn't fine into our nice clean datapath

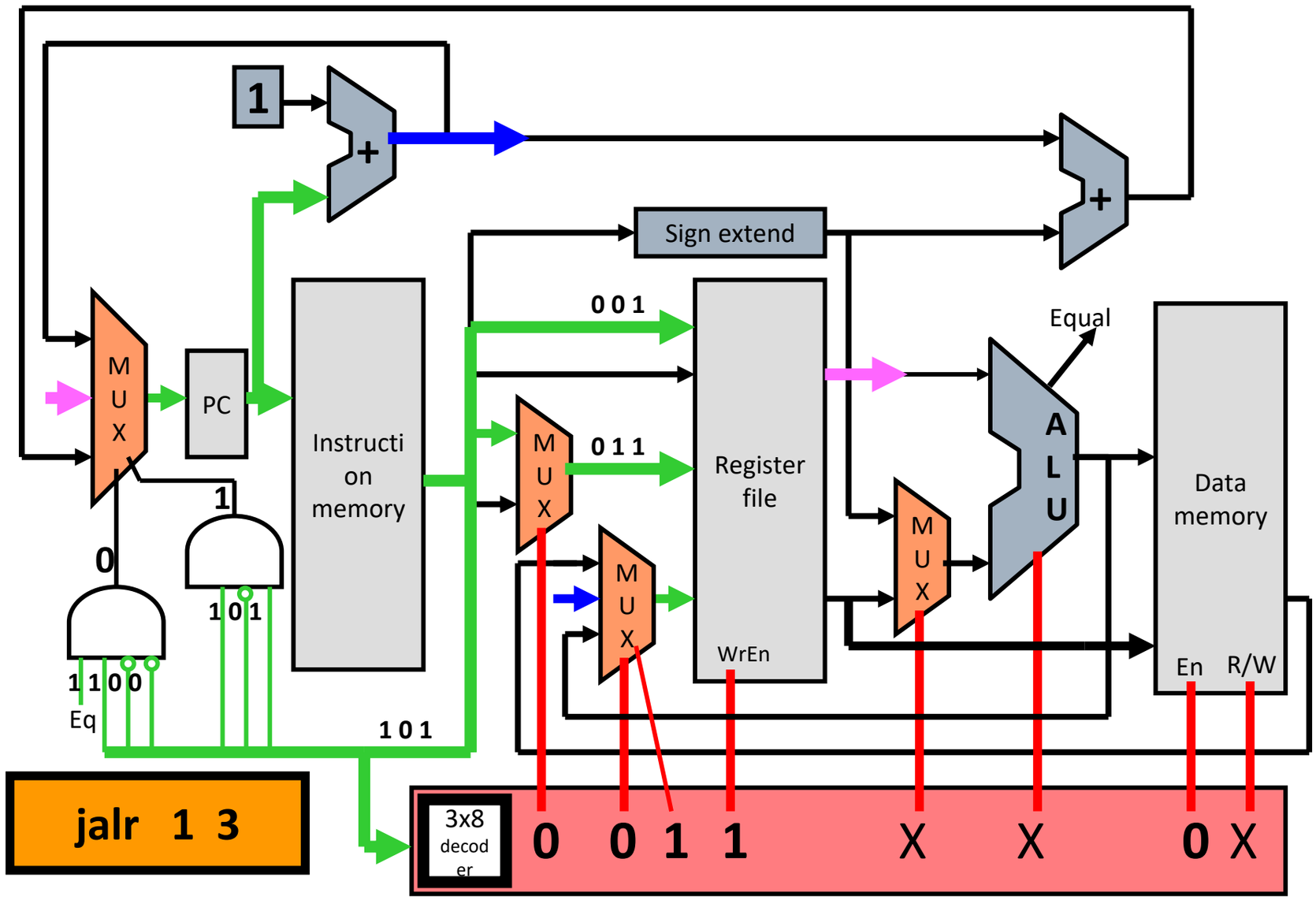
- To implement JALR we need to:
  - Write PC+1 into regB
  - Move regA into PC

- Right now there is:
  - No path to write PC+1 into a register
  - No path to write a register to the PC

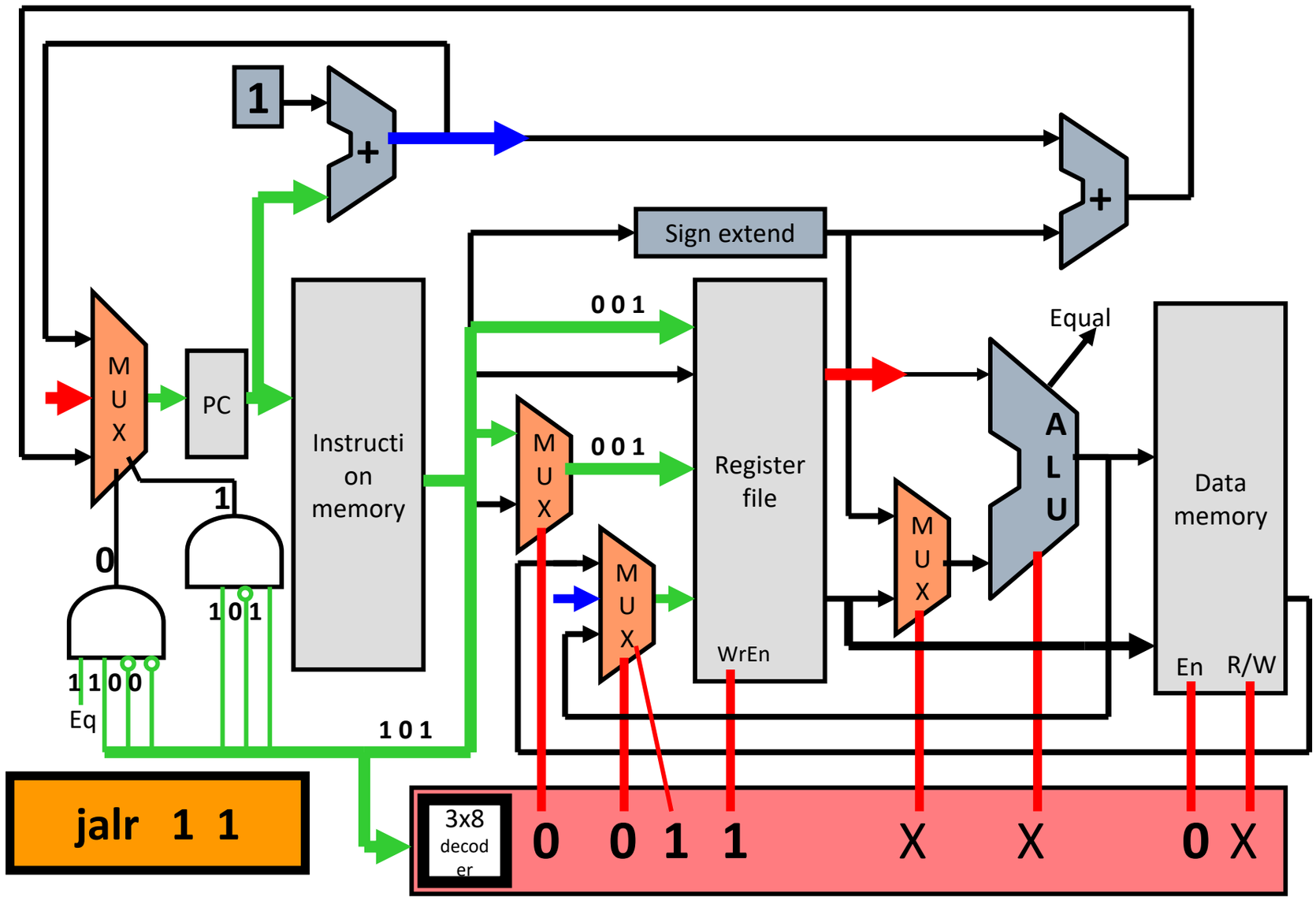
Executing a JALR Instruction



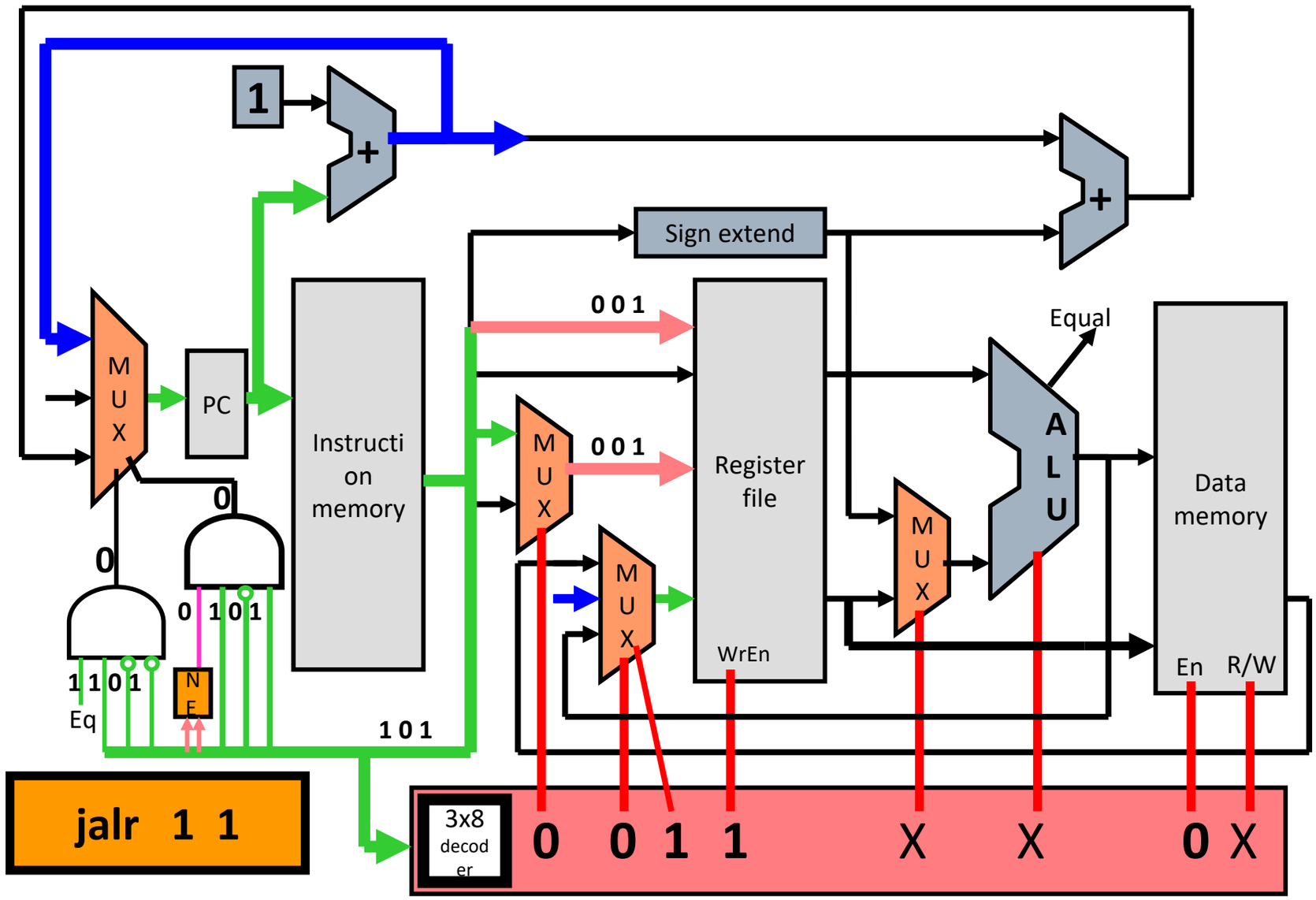
Executing a JALR Instruction



What if regA = regB for JALR?



Changes for JALR 1 1 Instruction



What's Wrong with Single-Cycle?

- **All instructions run at the speed of the slowest instruction.**
- Adding a long instruction can hurt performance
  - What if you wanted to include multiply?
- You cannot reuse any parts of the processor
  - We have 3 different adders to calculate PC+1, PC+1+offset and the ALU
- No benefit in making the common case fast
  - Since every instruction runs at the slowest instruction speed
    - This is particularly important for loads as we will see later

What's Wrong with Single-Cycle?

- 1 ns – Register read/write time
- 2 ns – ALU/adder
- 2 ns – memory access
- 0 ns – MUX, PC access, sign extend, ROM

What is the latency of lw?

	Get Instr	**read reg**	**ALU oper.**	**mem**	**write reg**	
• add:	2ns	+ 1ns	+ 2ns		+ 1 ns	= 6 ns
• beq:	2ns	+ 1ns	+ 2ns			= 5 ns
• sw:	2ns	+ 1ns	+ 2ns	+ 2ns		= 7 ns
• lw:	2ns	+ 1ns	+ 2ns	+ 2ns	+ 1ns	= 8 ns

Computing Execution Time

Assume: 100 instructions executed

- 25% of instructions are loads,
- 10% of instructions are stores,
- 45% of instructions are adds, and
- 20% of instructions are branches.

Single-cycle execution:

??

Optimal execution:

??

What is the single-cycle execution time?

How fast could this run if we weren't limited by a single-clock period?

Computing Execution Time

Assume: 100 instructions executed

- 25% of instructions are loads,
- 10% of instructions are stores,
- 45% of instructions are adds, and
- 20% of instructions are branches.

Single-cycle execution:

100 * 8ns = **800** ns

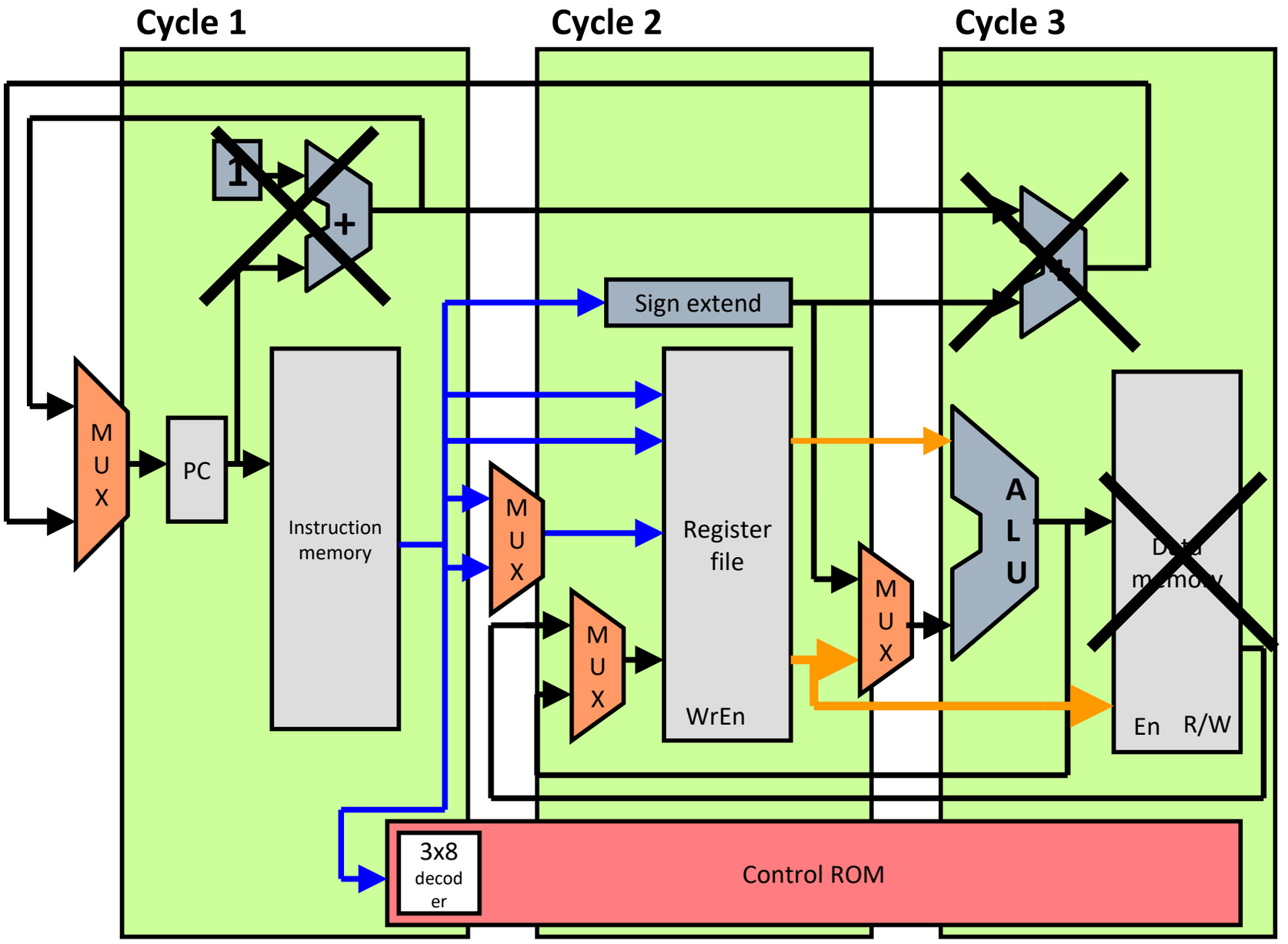
Optimal execution:

25*8ns + 10*7ns + 45*6ns + 20*5ns = **640** ns

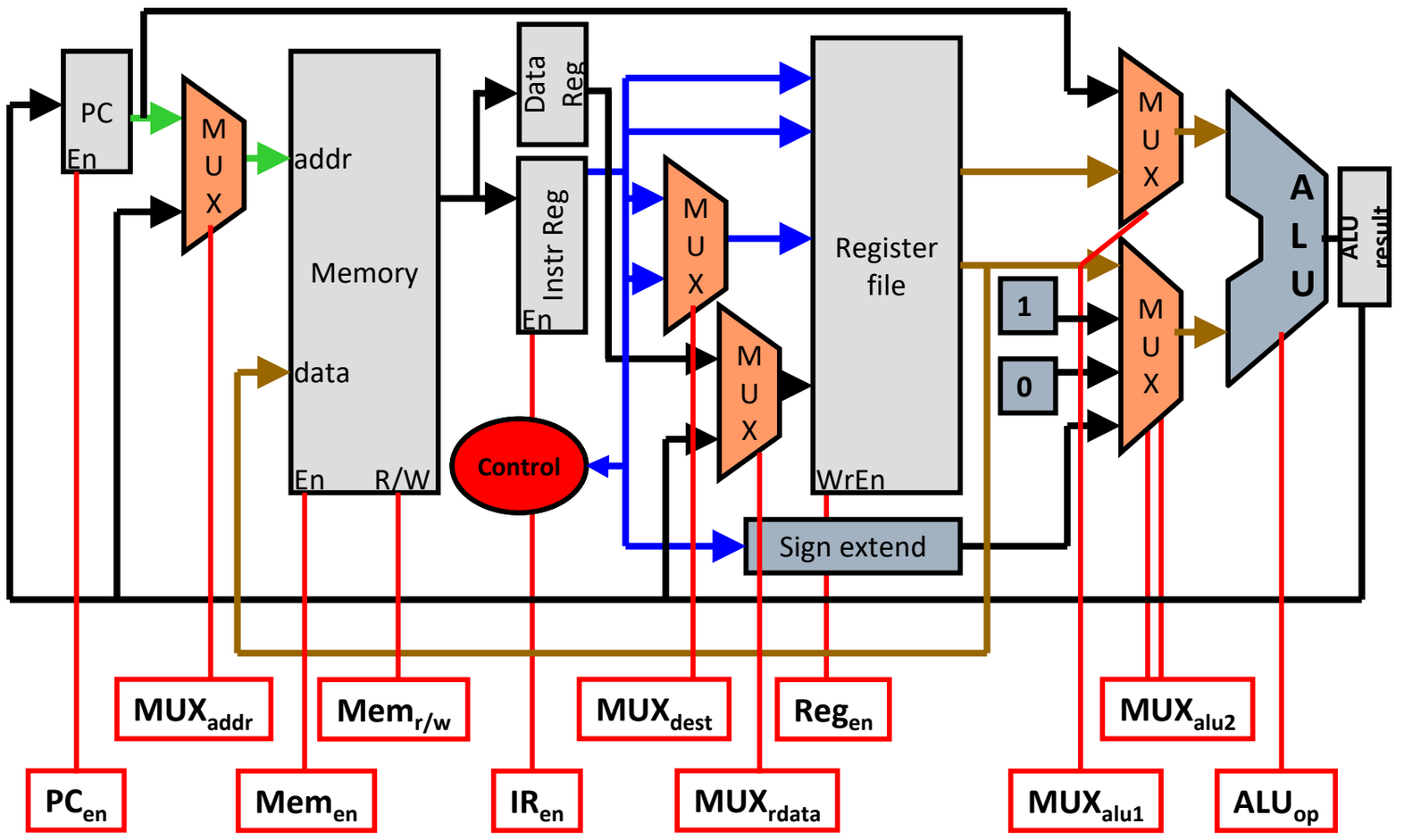
Multiple-Cycle Execution

- Each instruction takes multiple cycles to execute
  - Cycle time is reduced
  - Slower instructions take more cycles
  - Faster instruction take fewer cycles
    - We can start next instruction earlier, rather than just waiting
  - Can reuse datapath elements each cycle
- What is needed to make this work?
  - Since you are re-using elements for different purposes, you need more and/or wider MUXes.
  - You may need extra registers if you need to remember an output for 1 or more cycles.
  - Control is more complicated since you need to send new signals on each cycle.

LC2K Datapath – cycle groups



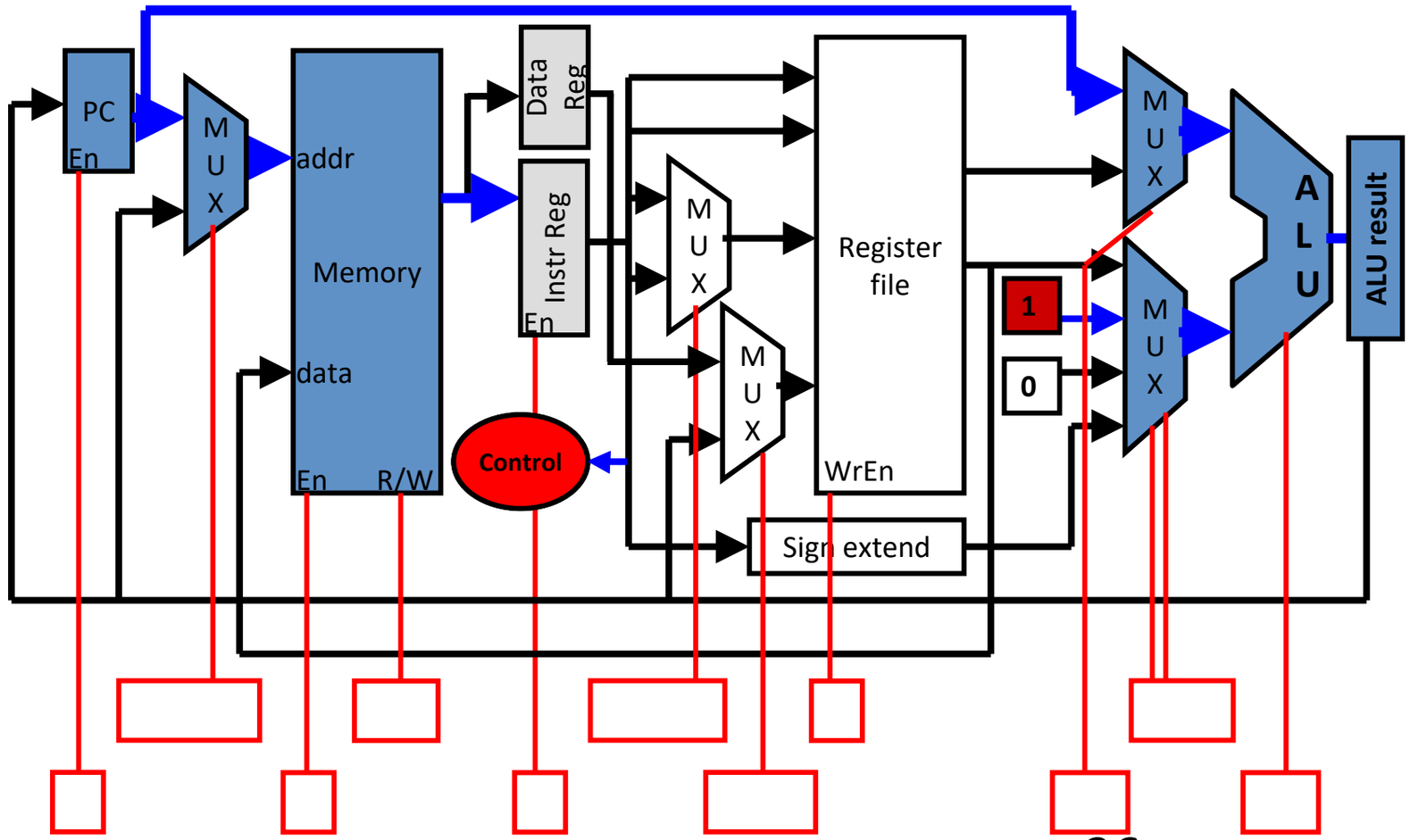
Multi-cycle LC2 Datapath



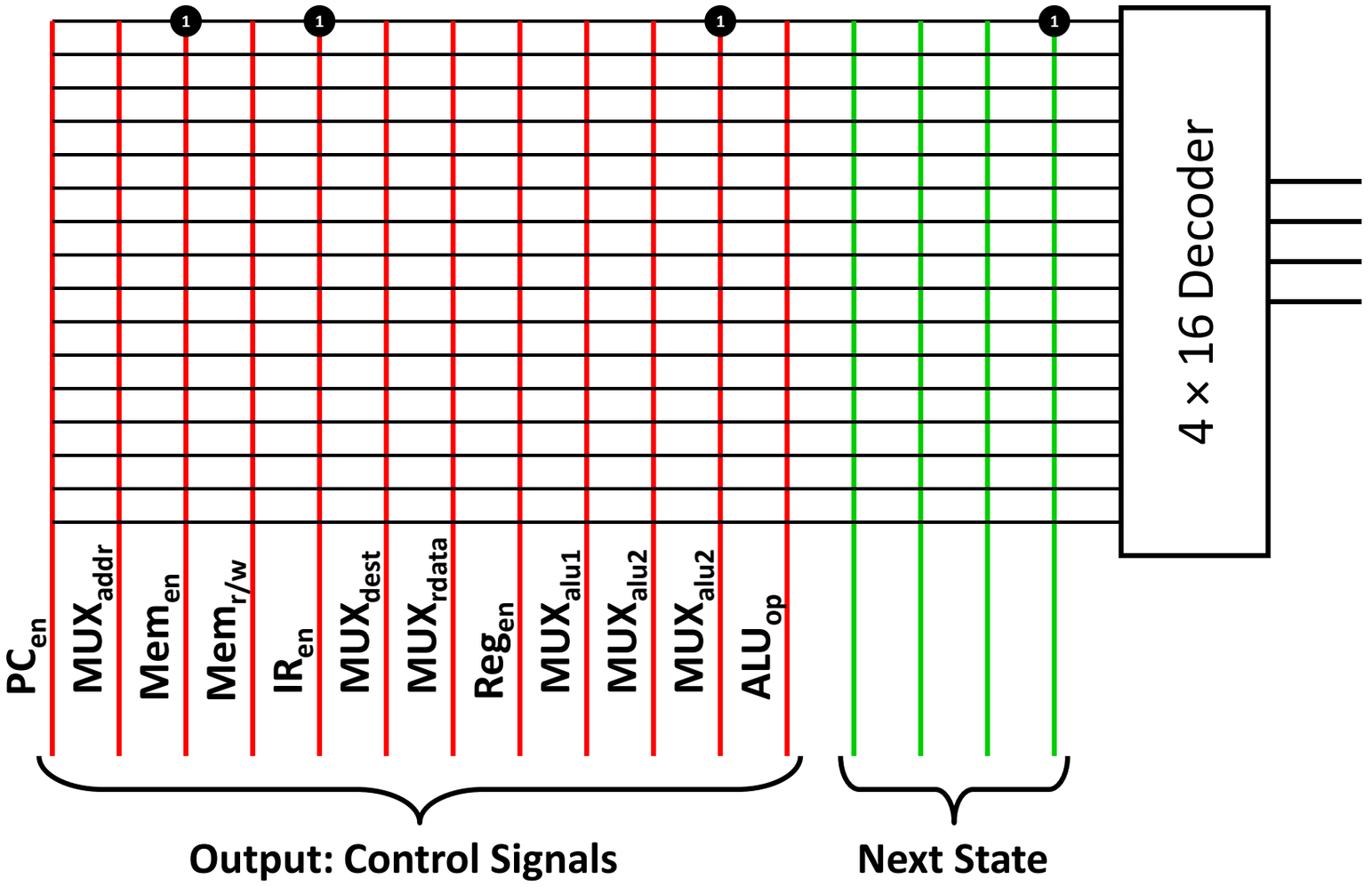
Each red signal comes from "Control" (implemented via ROM as before)

First Cycle (State 0) Fetch Instr

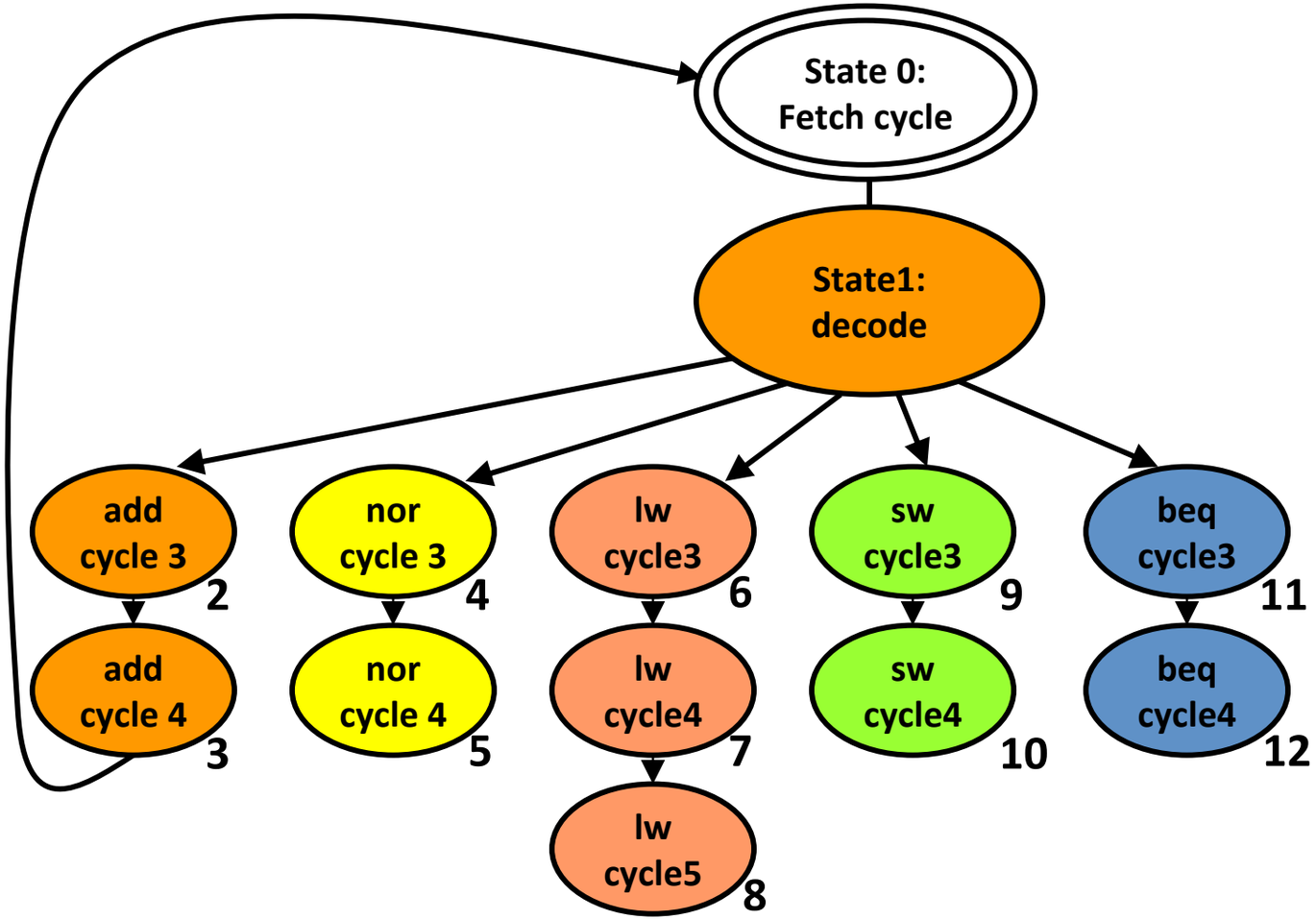
This is the same for all instructions
(since we don't know the instruction yet!)



Building the Control ROM



State 1: instruction decode

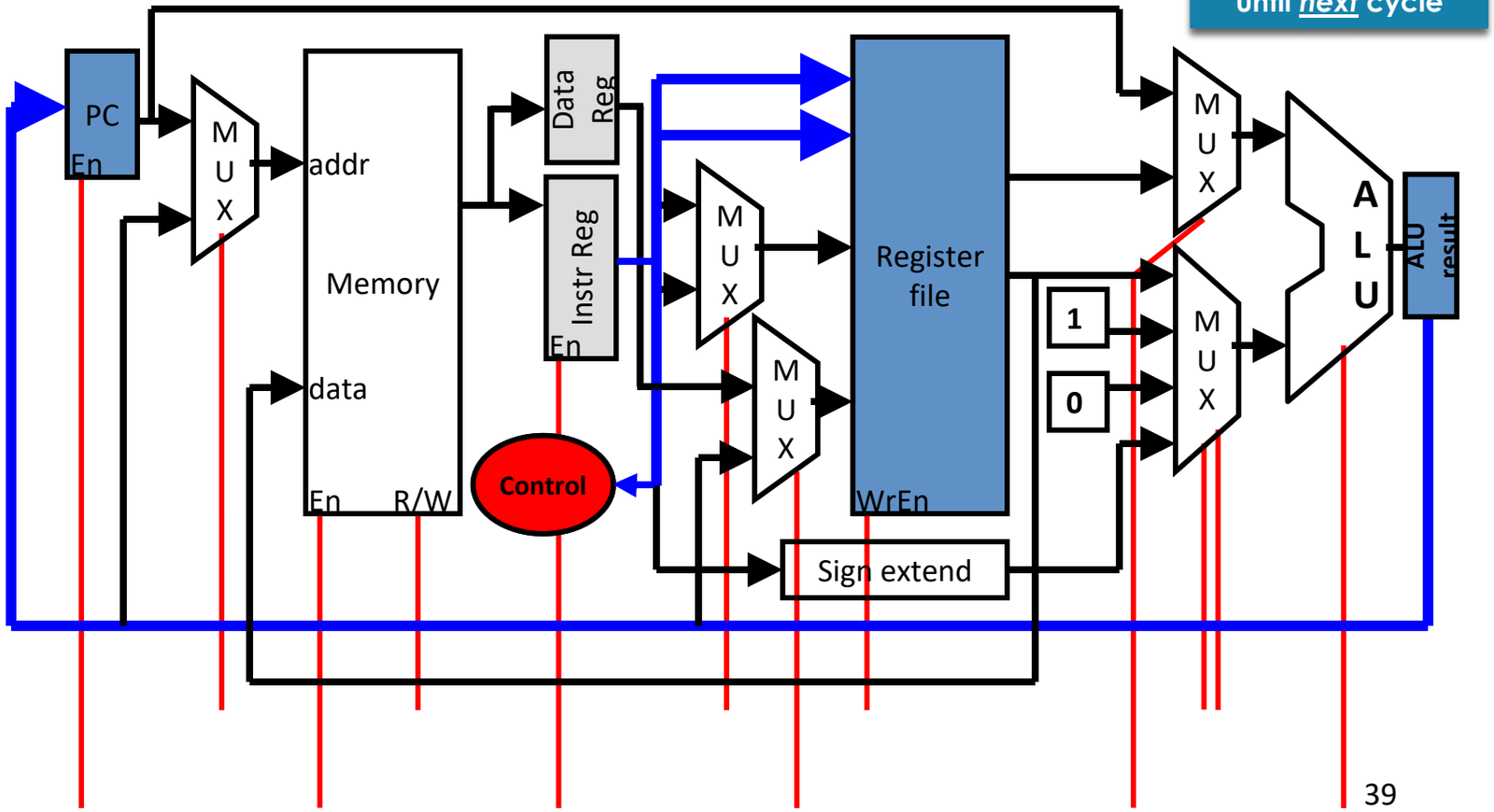


State 1: output function

Poll: What will the control bits be?

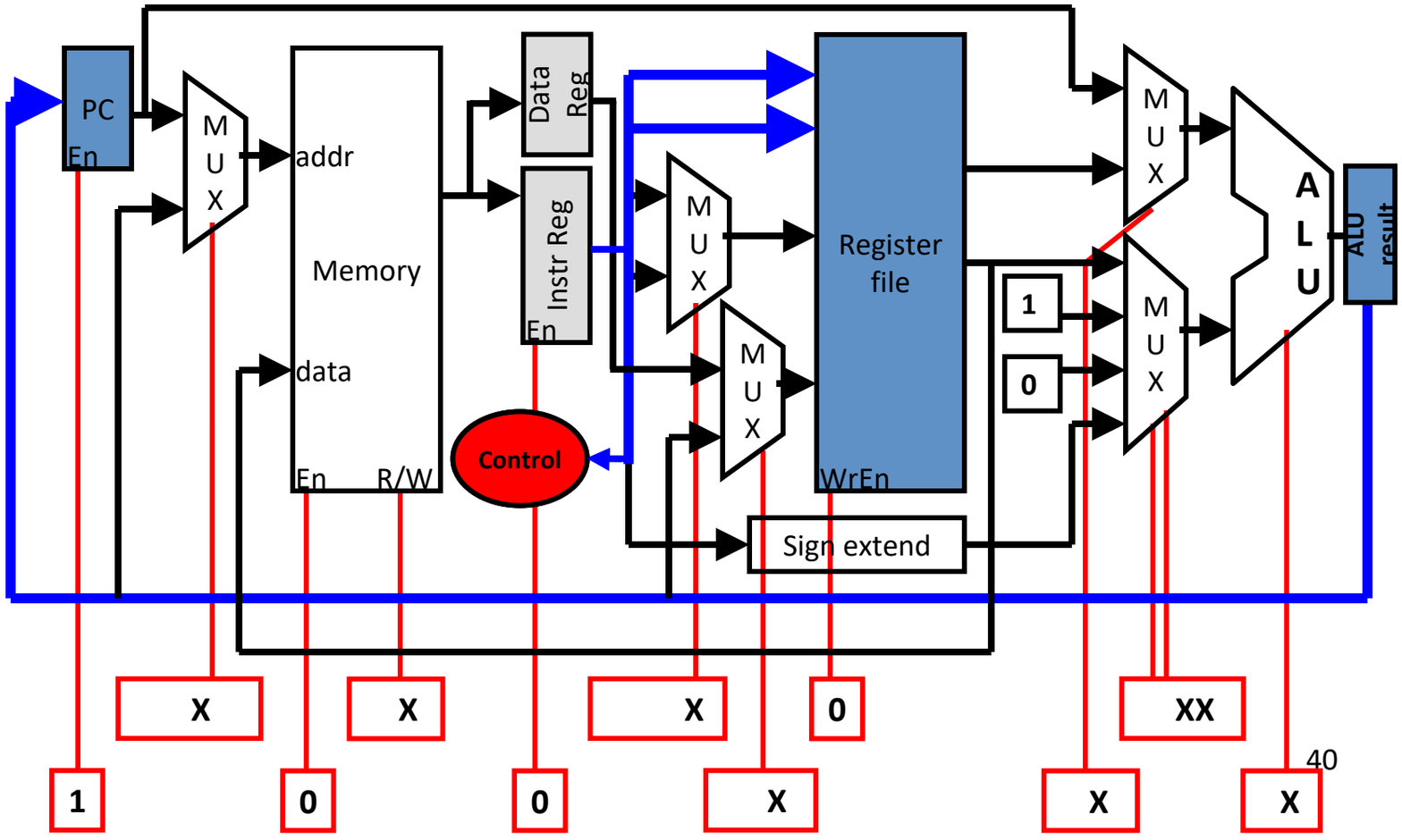
Update PC; read registers (regA and regB); use opcode to determine next state

Note: since RF read latency is same as clock period, RF data isn't available until *next* cycle



State 1: output function

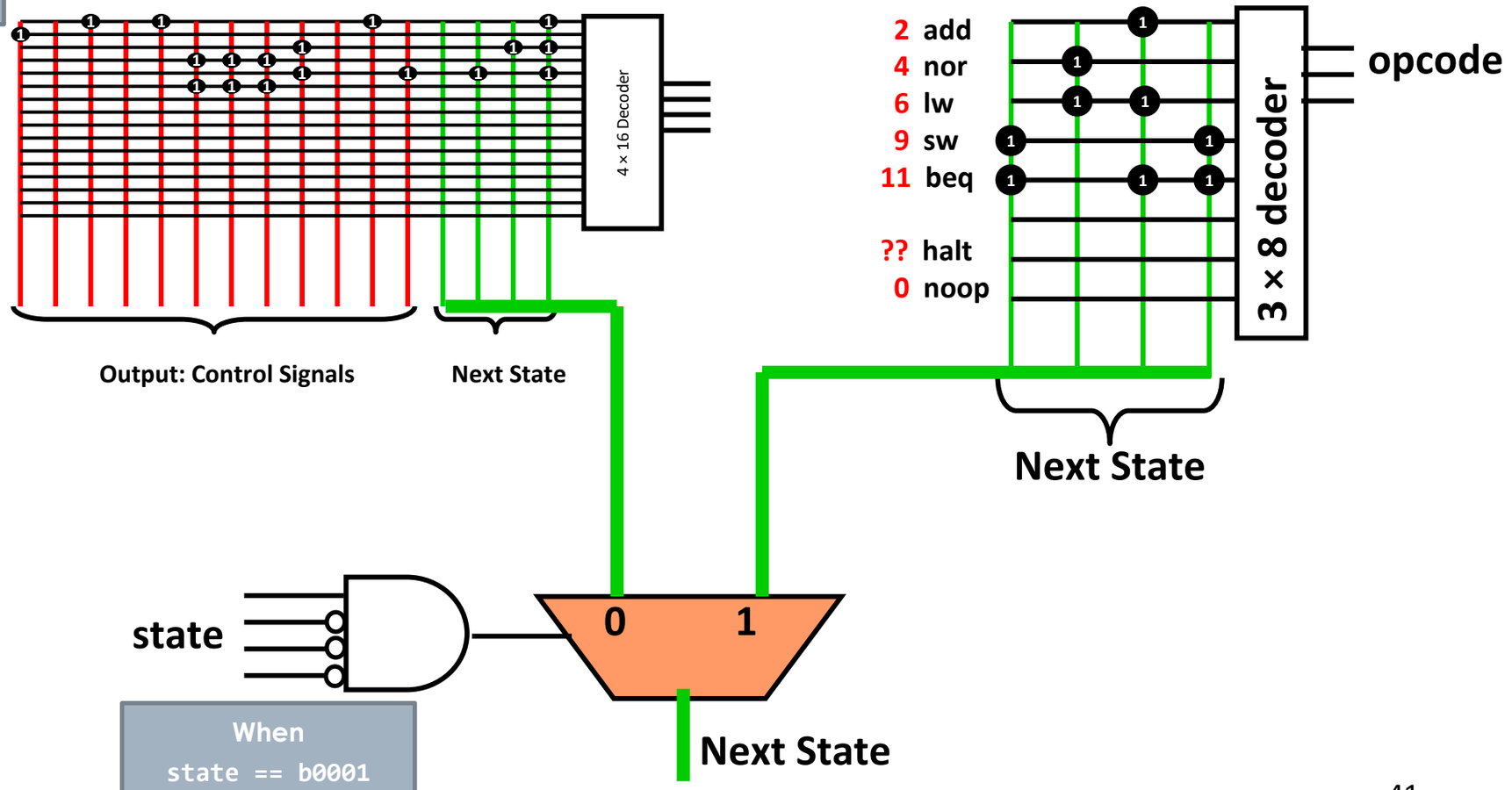
Update PC; read registers (regA and regB);
use opcode to determine next state



Transitioning from Decode State

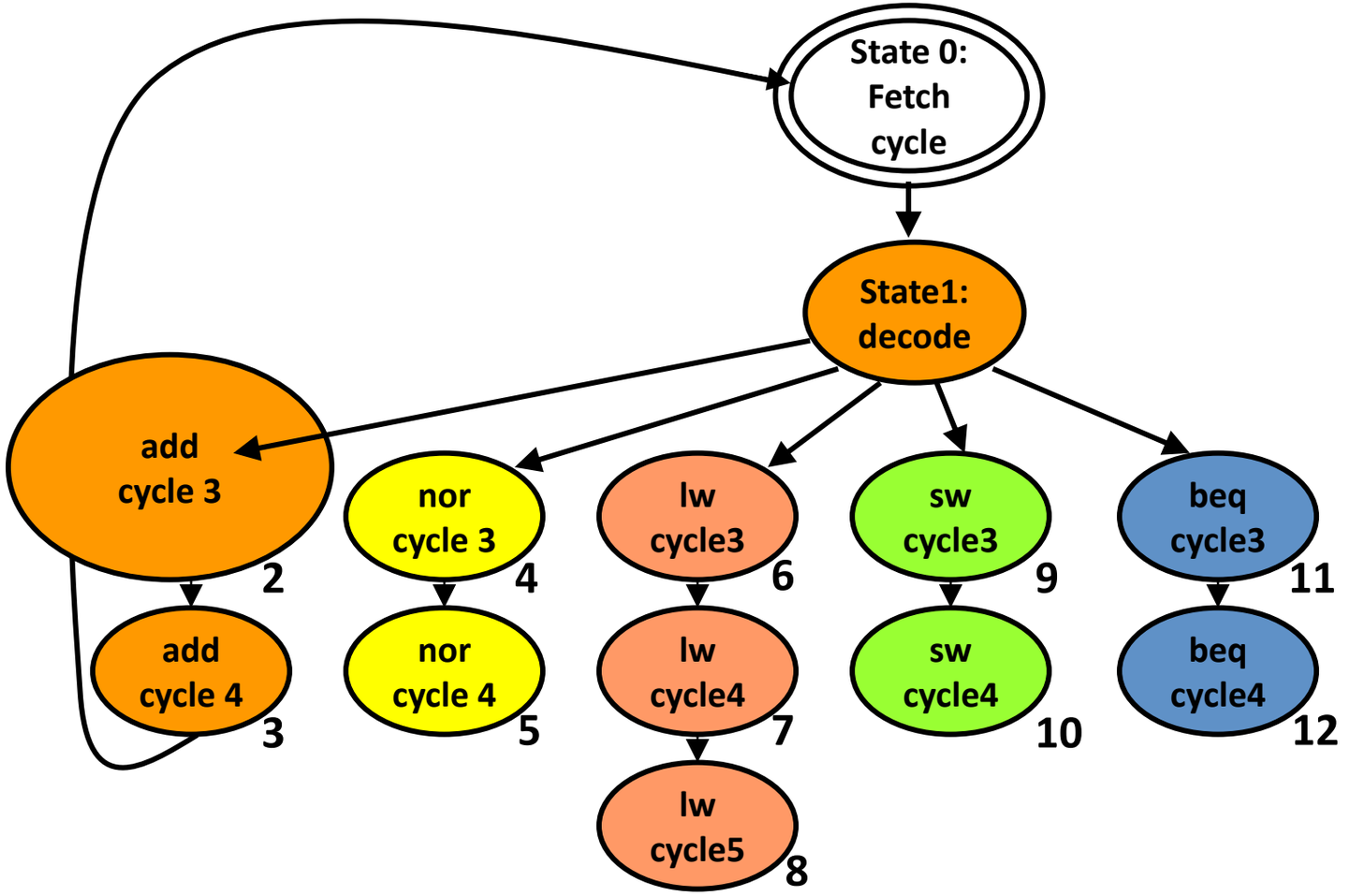
Which state we go to depends on opcode, can't just look at current state like before

Secondary ROM stores which state we should branch to after decode for each opcode



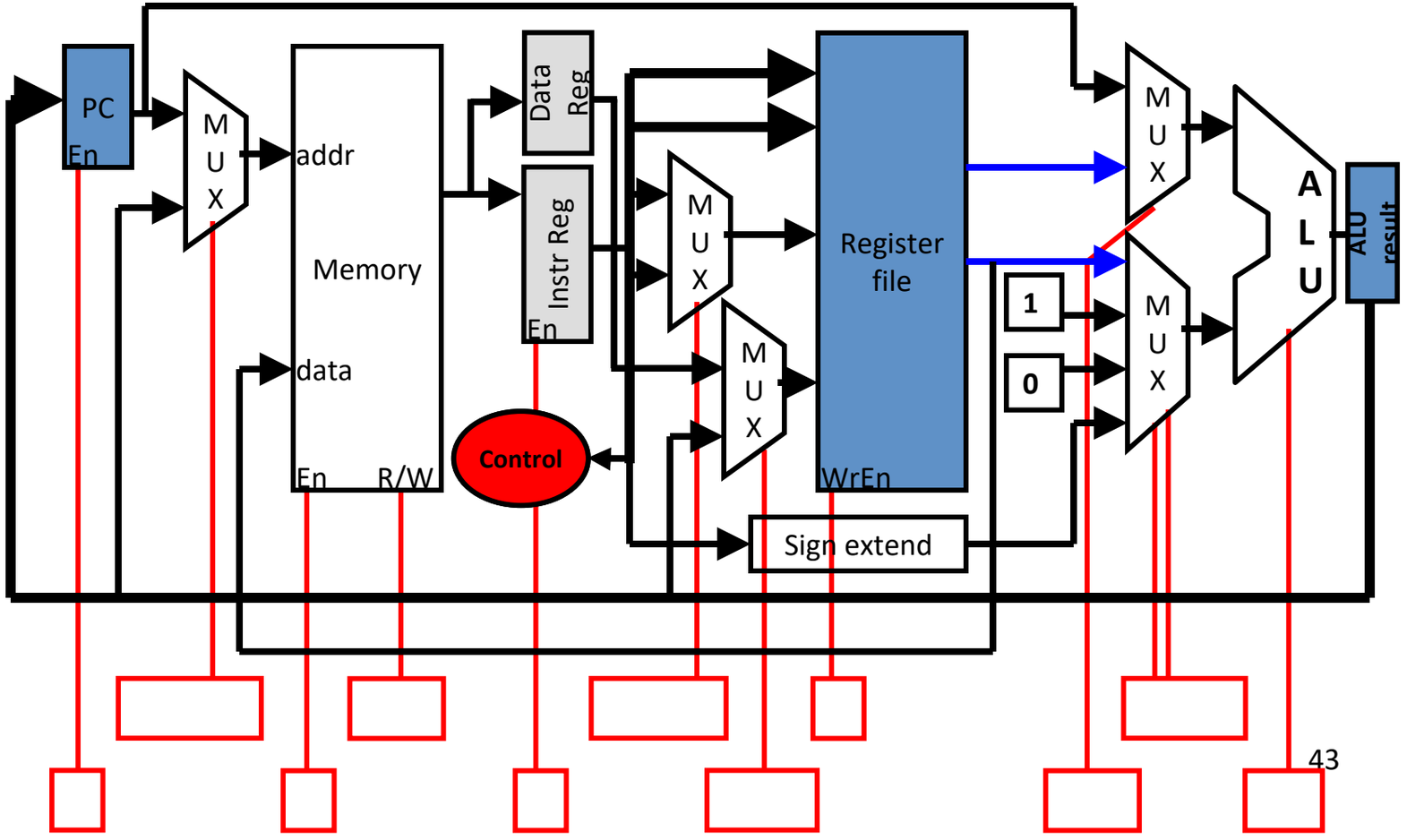
When `state == b0001` choose secondary ROM for next state logic

State 2: Add cycle 3



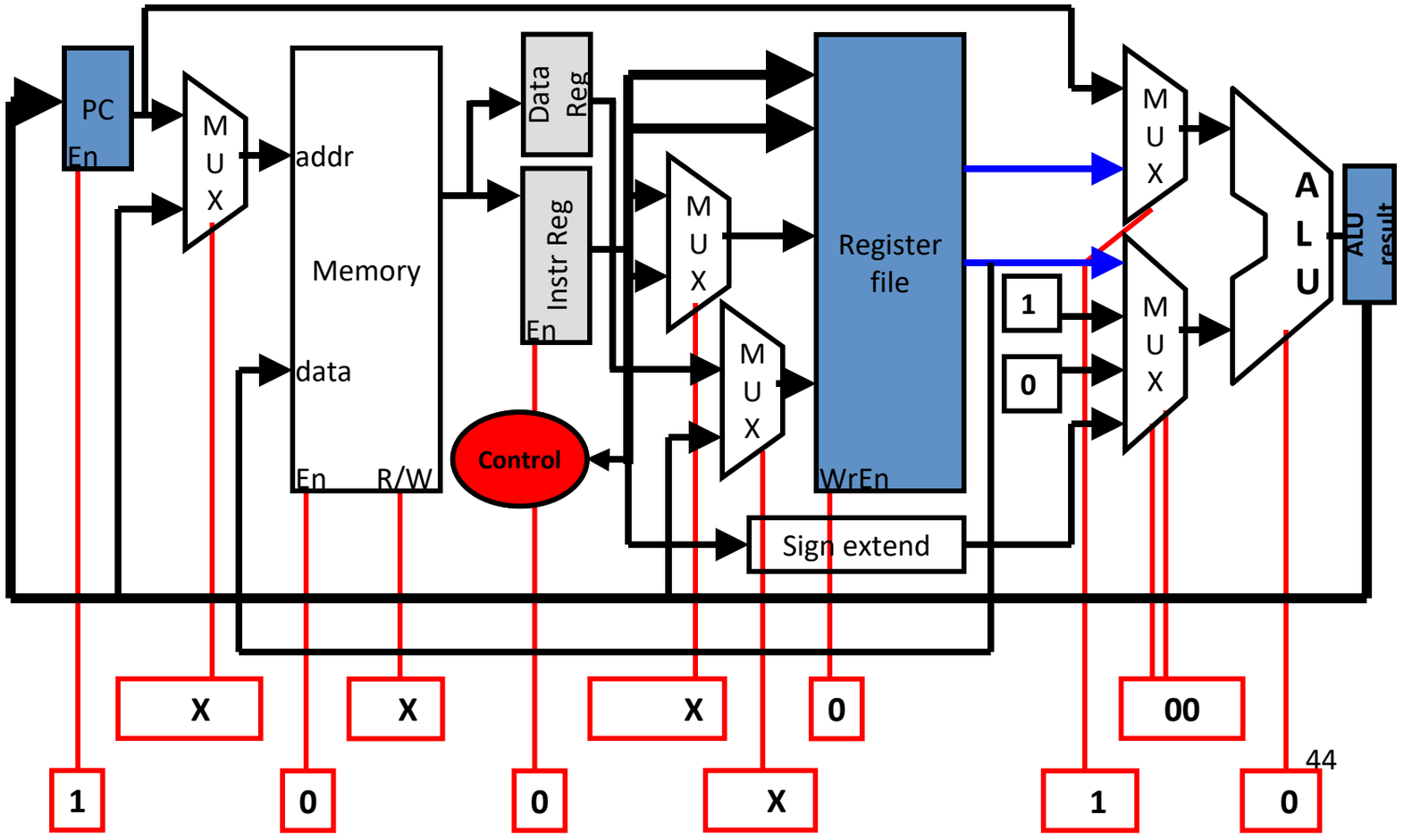
State 2: Add Cycle 3 Operation

Send control signals to MUX to select values of regA and regB and control signal to ALU to add

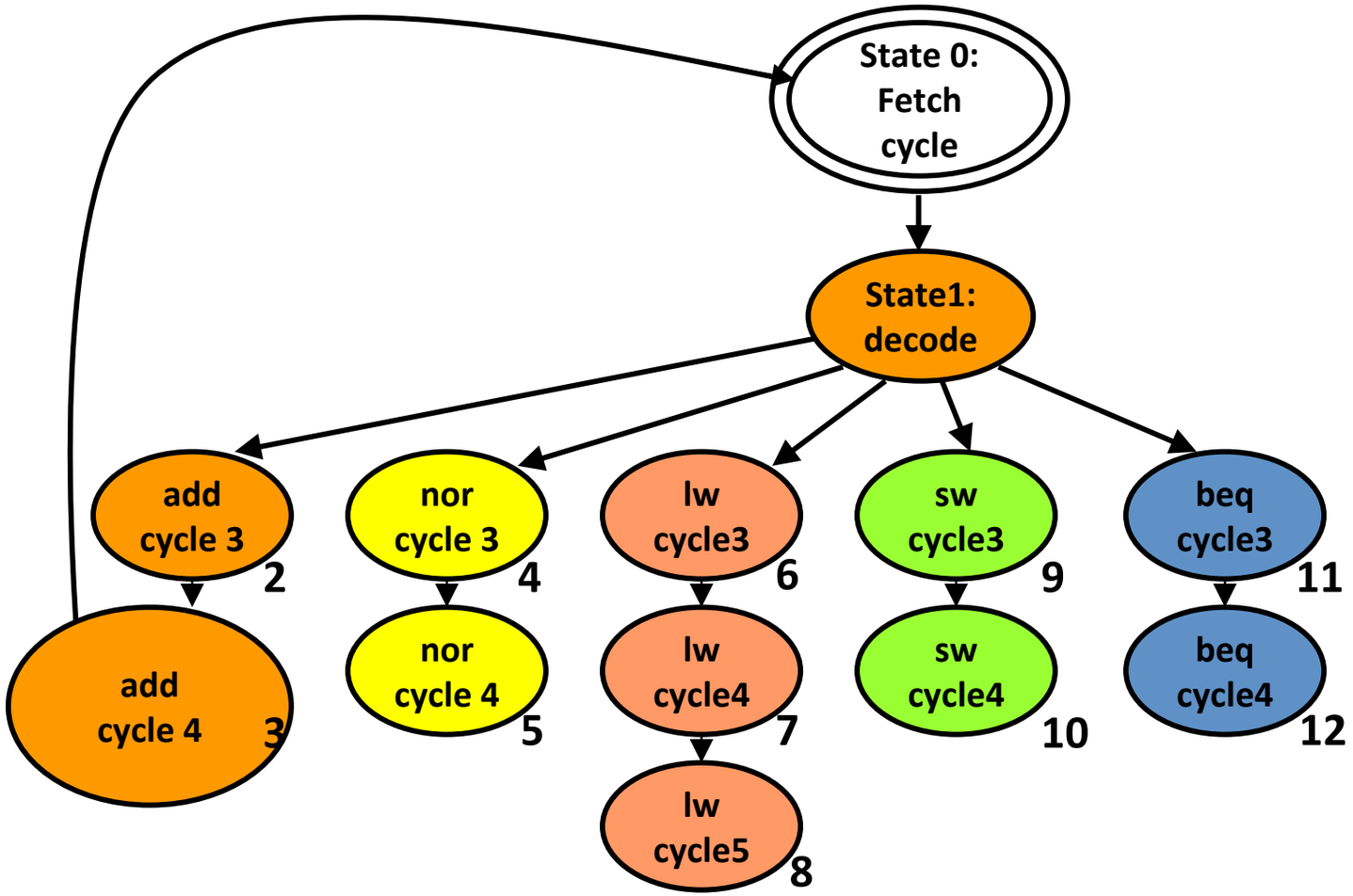


State 2: Add Cycle 3 Operation

Send control signals to MUX to select values of regA and regB and control signal to ALU to add

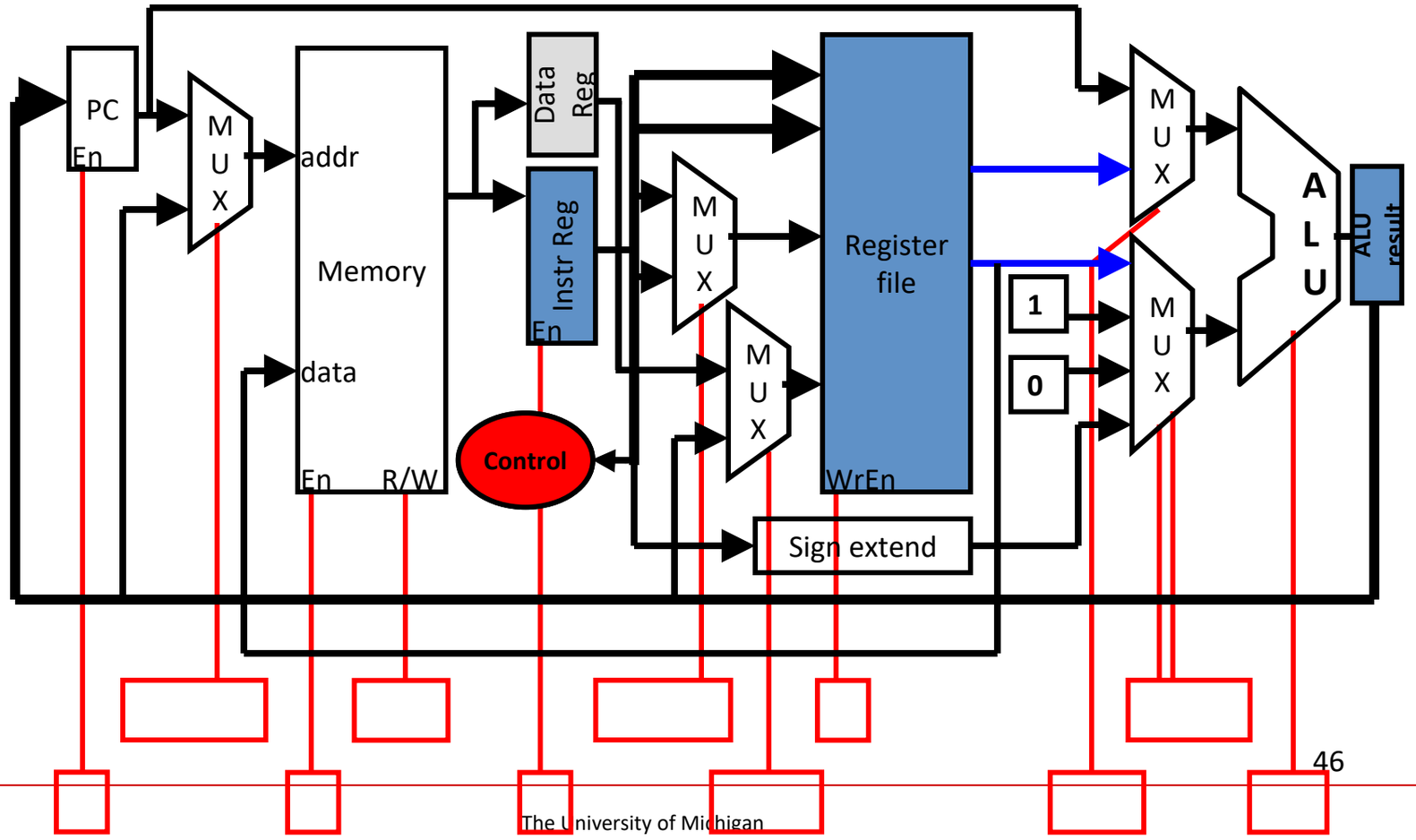


State 3: Add cycle 4



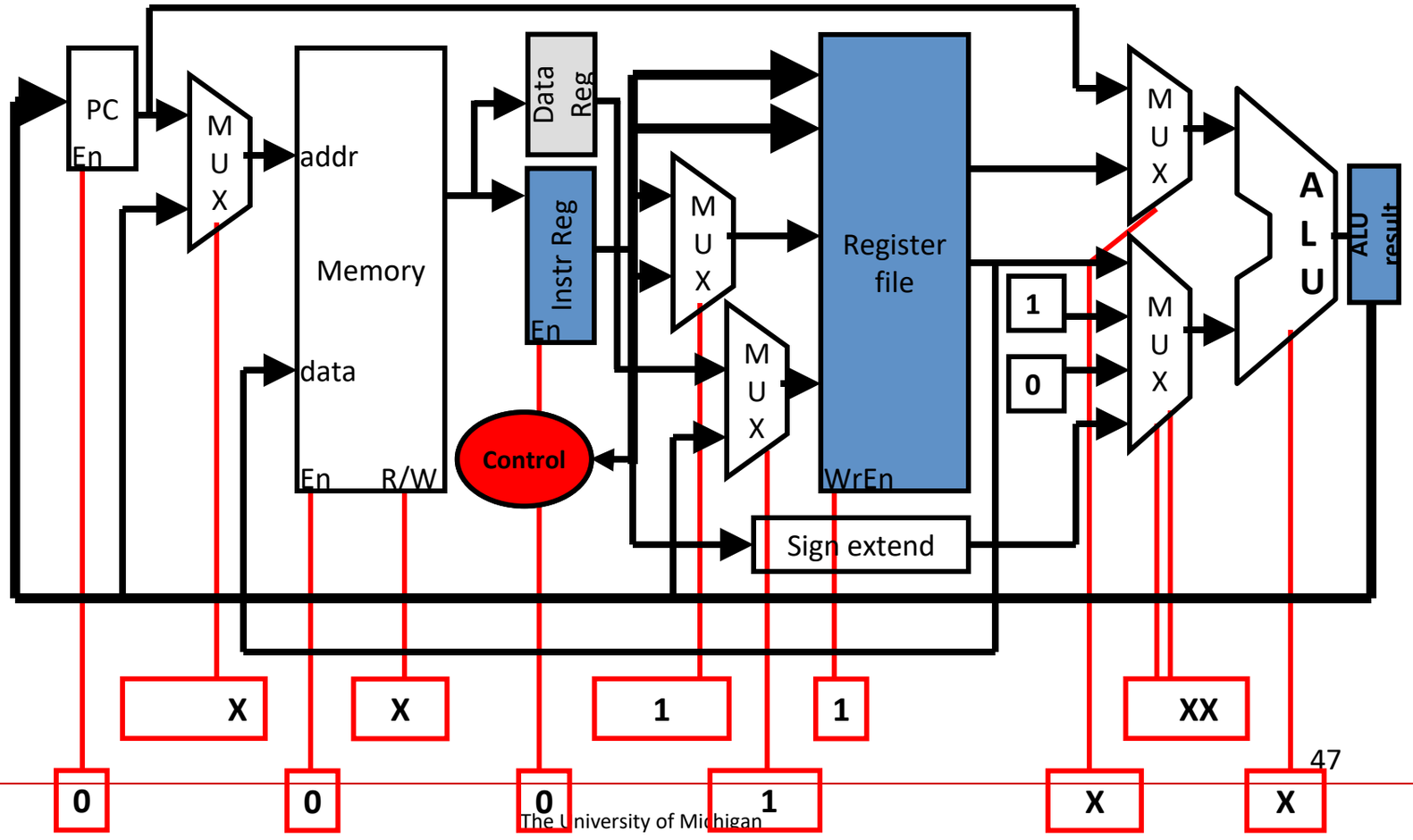
Add Cycle 4 (State 3) Operation

Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.

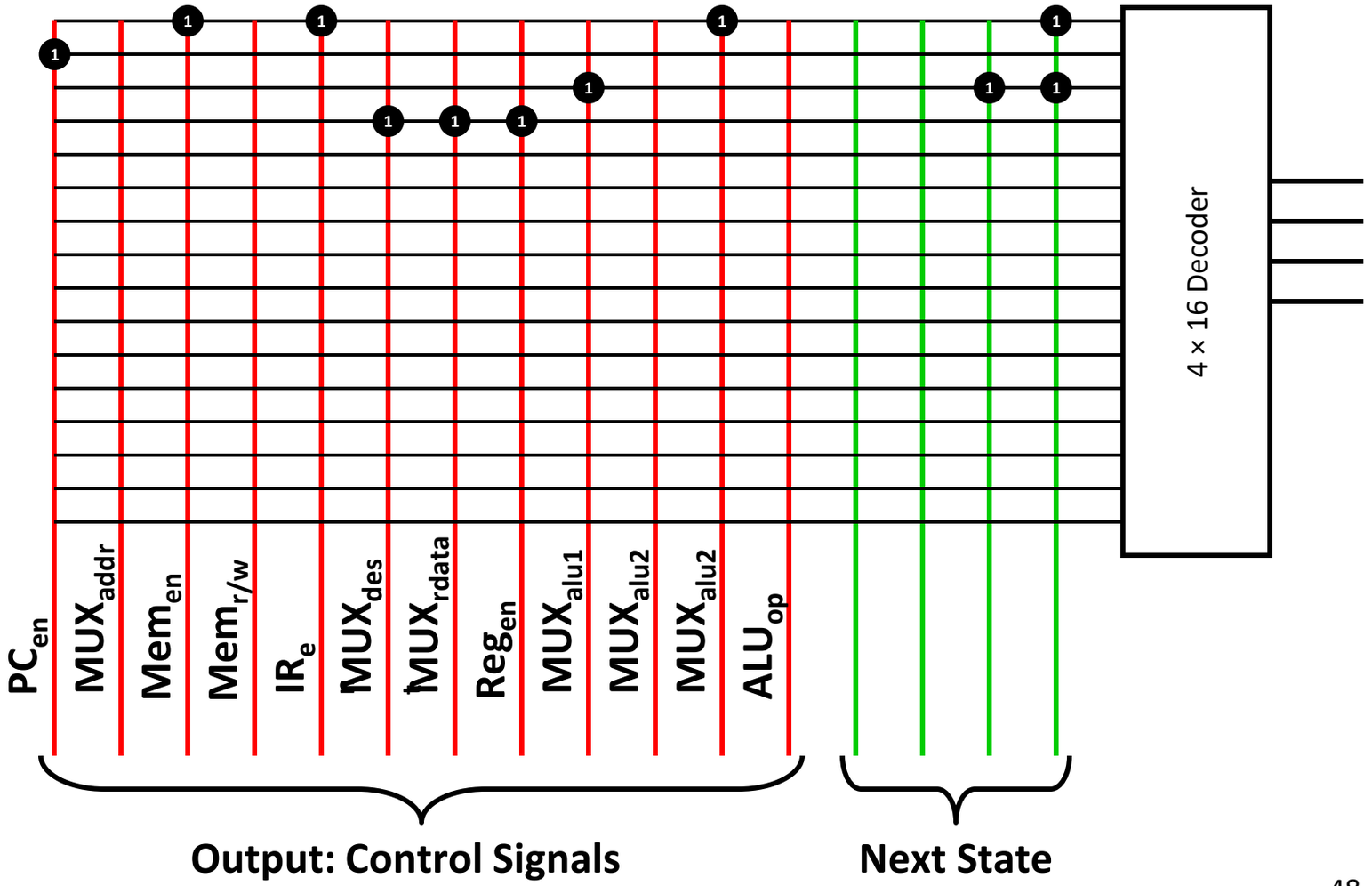


Add Cycle 4 (State 3) Operation

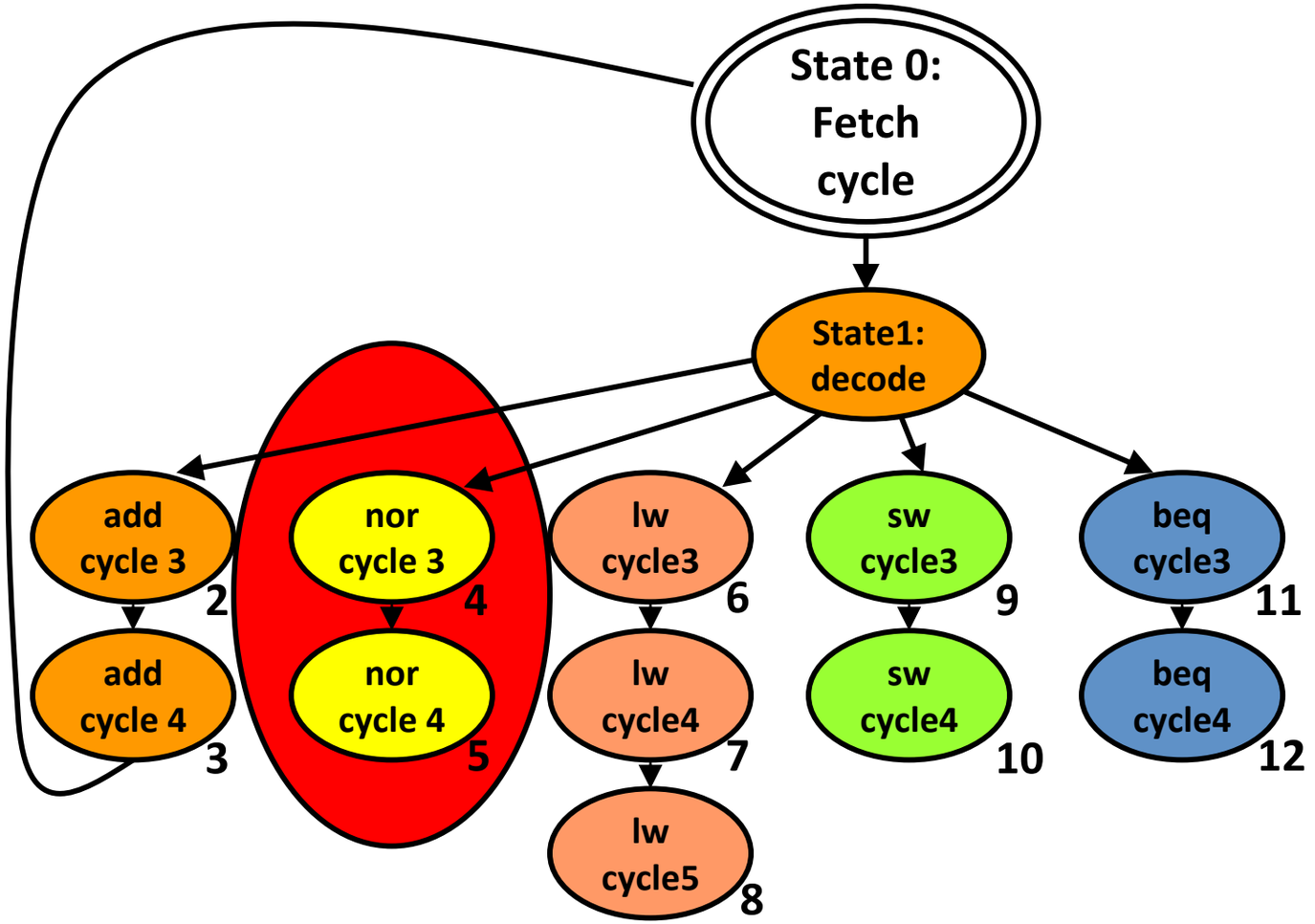
Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



Building the Control Rom

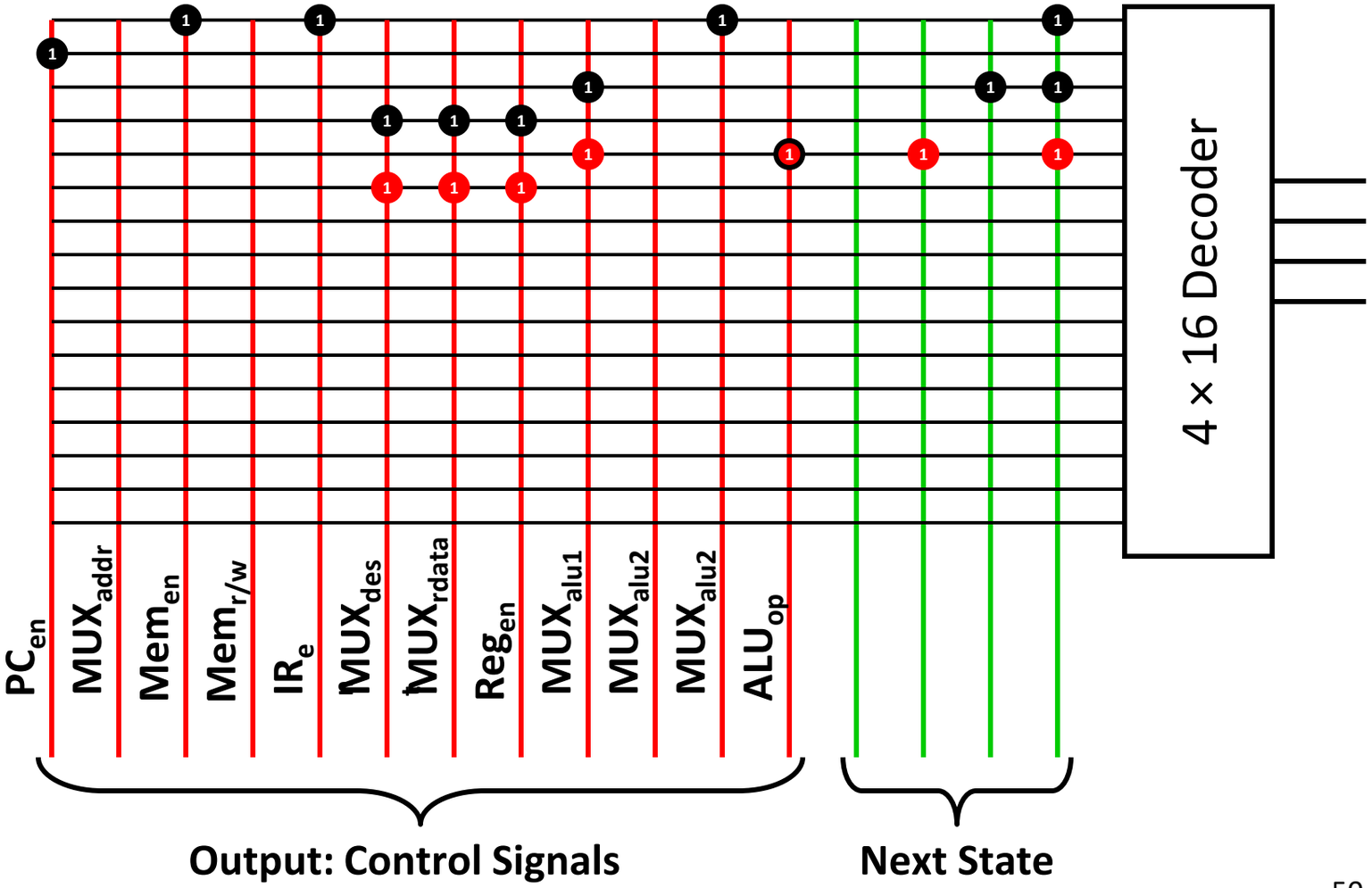


Return to State 0: Fetch cycle to execute the next instruction

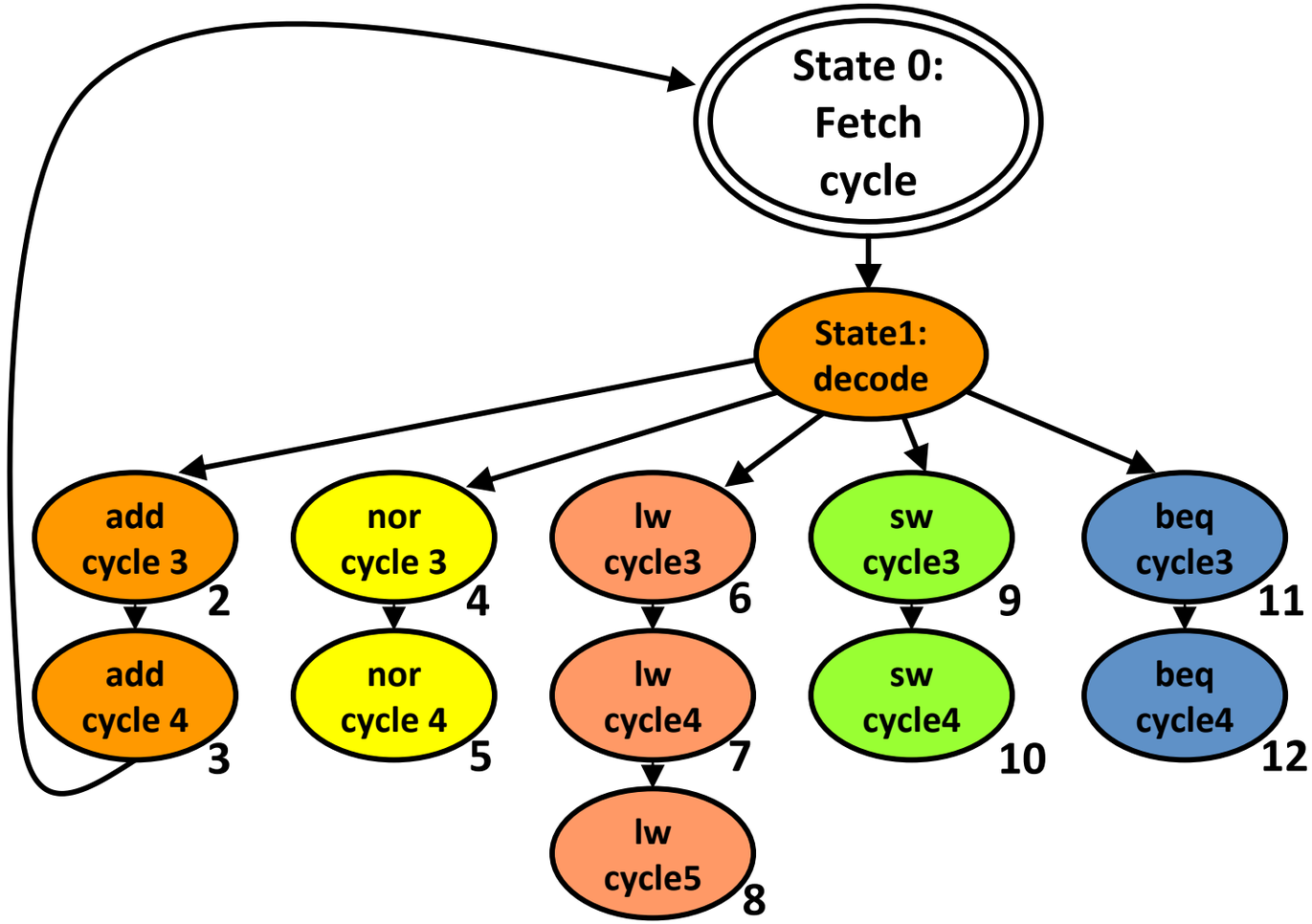


Control Rom for nor (4 and 5)

Same output as add except ALU_{op} and Next State

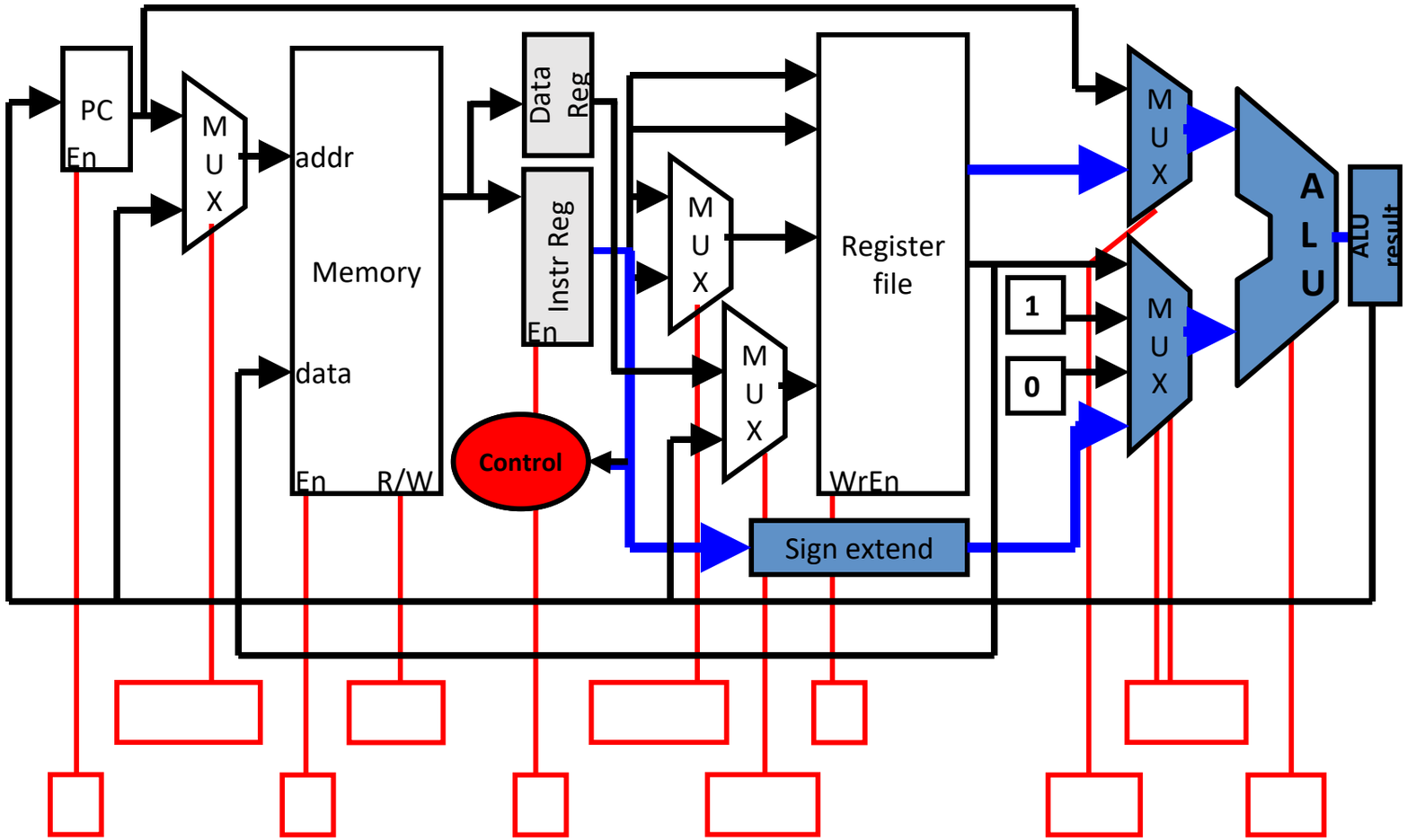


Return to State 0: Fetch cycle to execute the next instruction



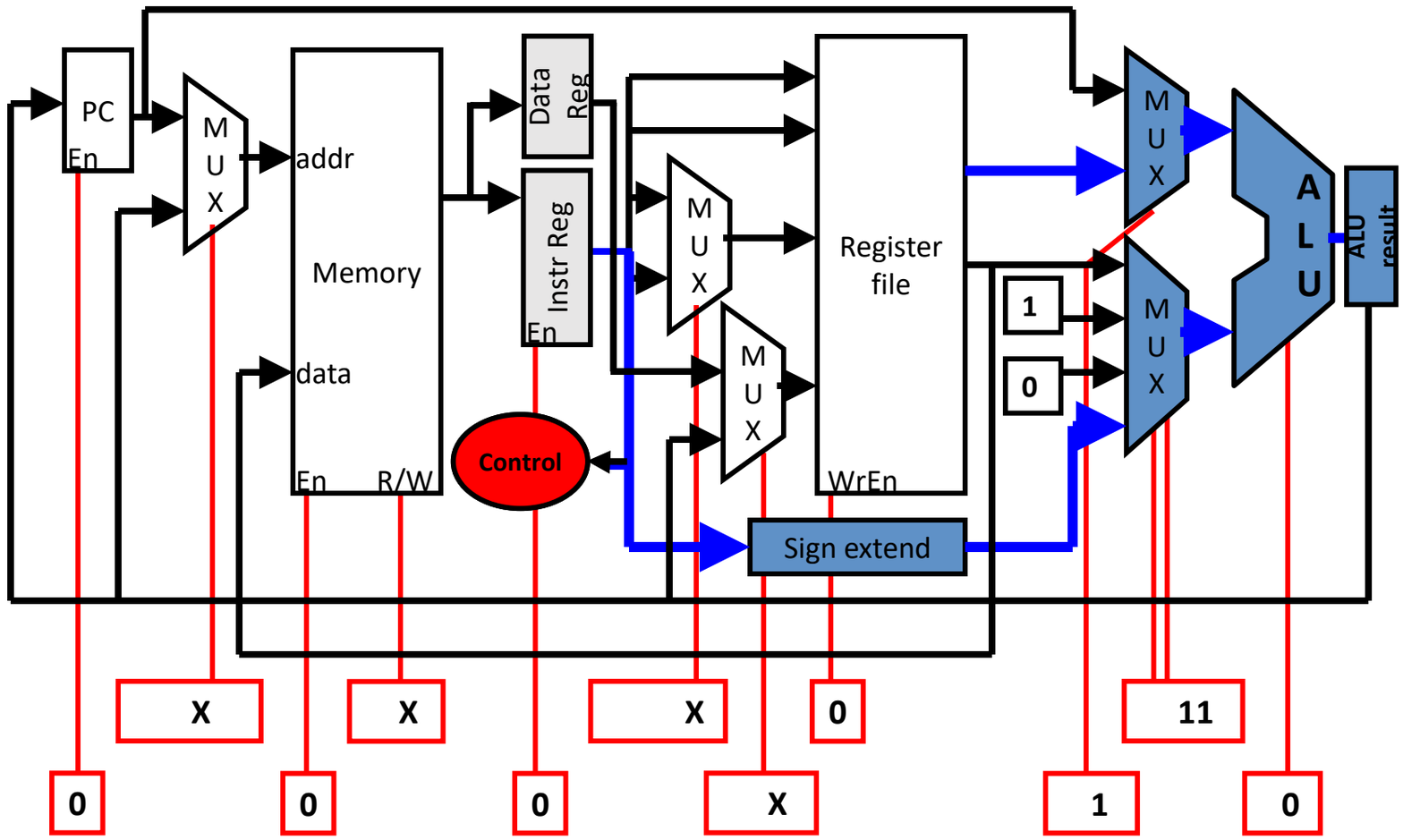
State 6: LW cycle 3

Calculate address for memory reference

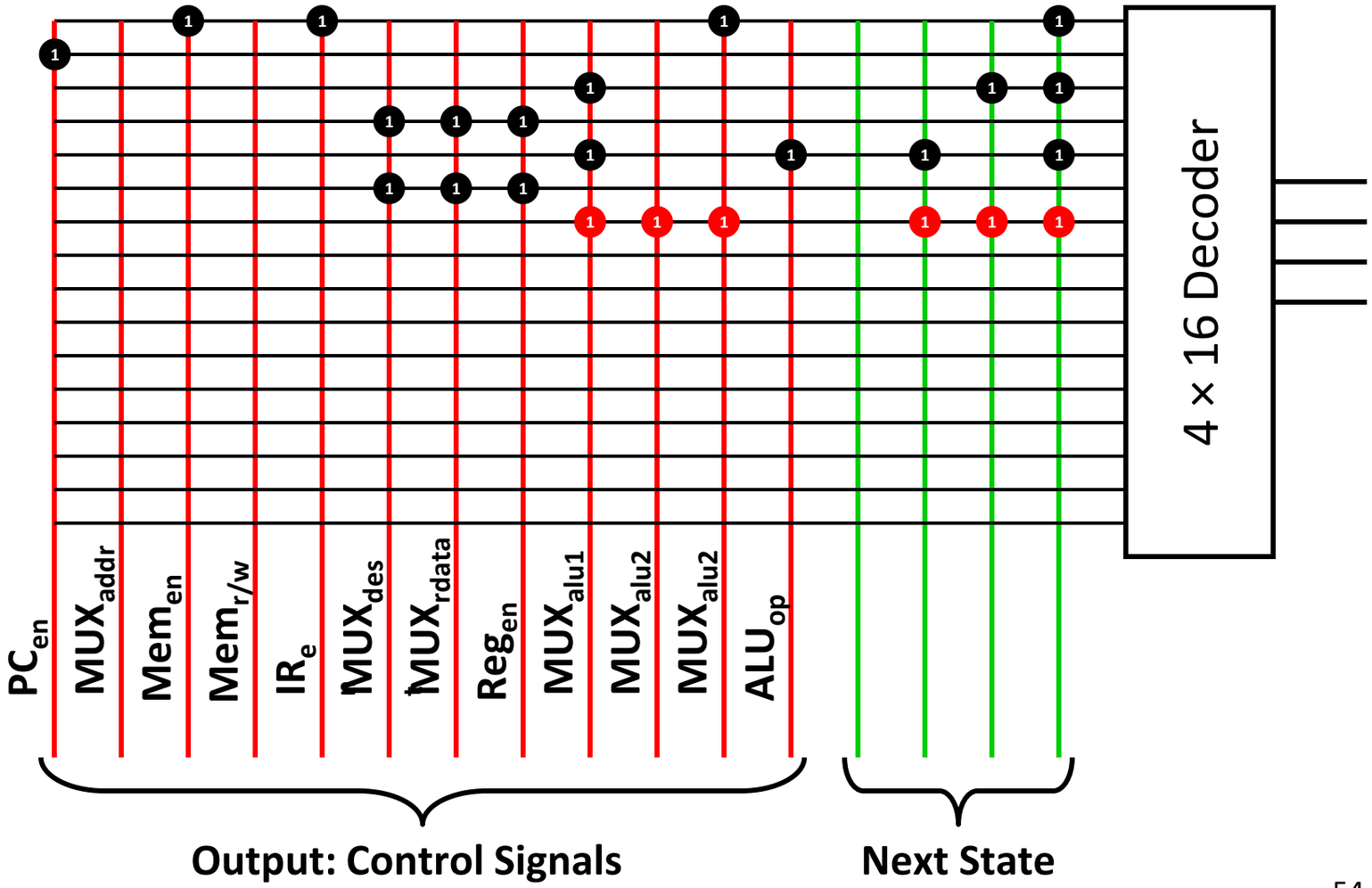


State 6: LW cycle 3

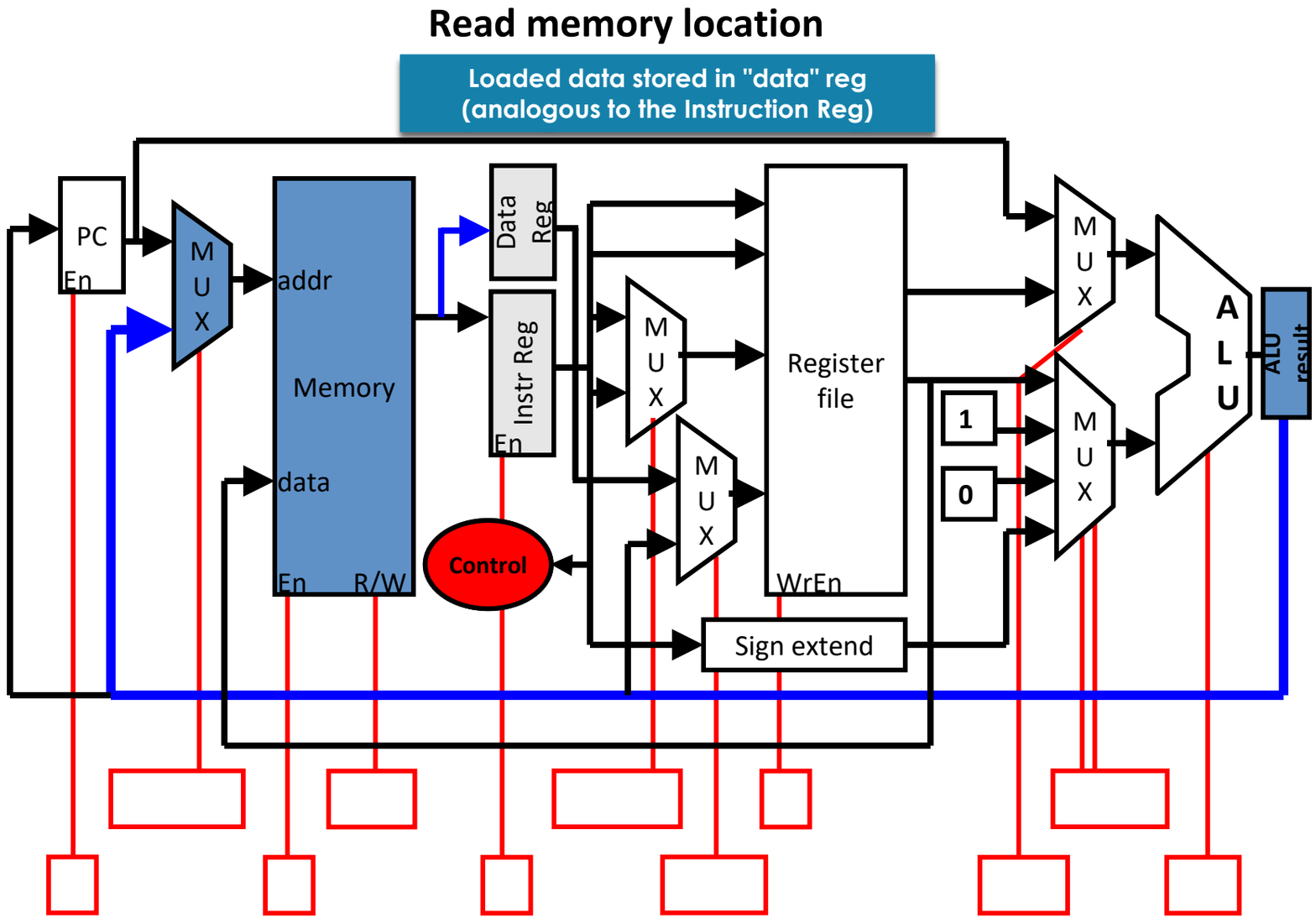
Calculate address for memory reference



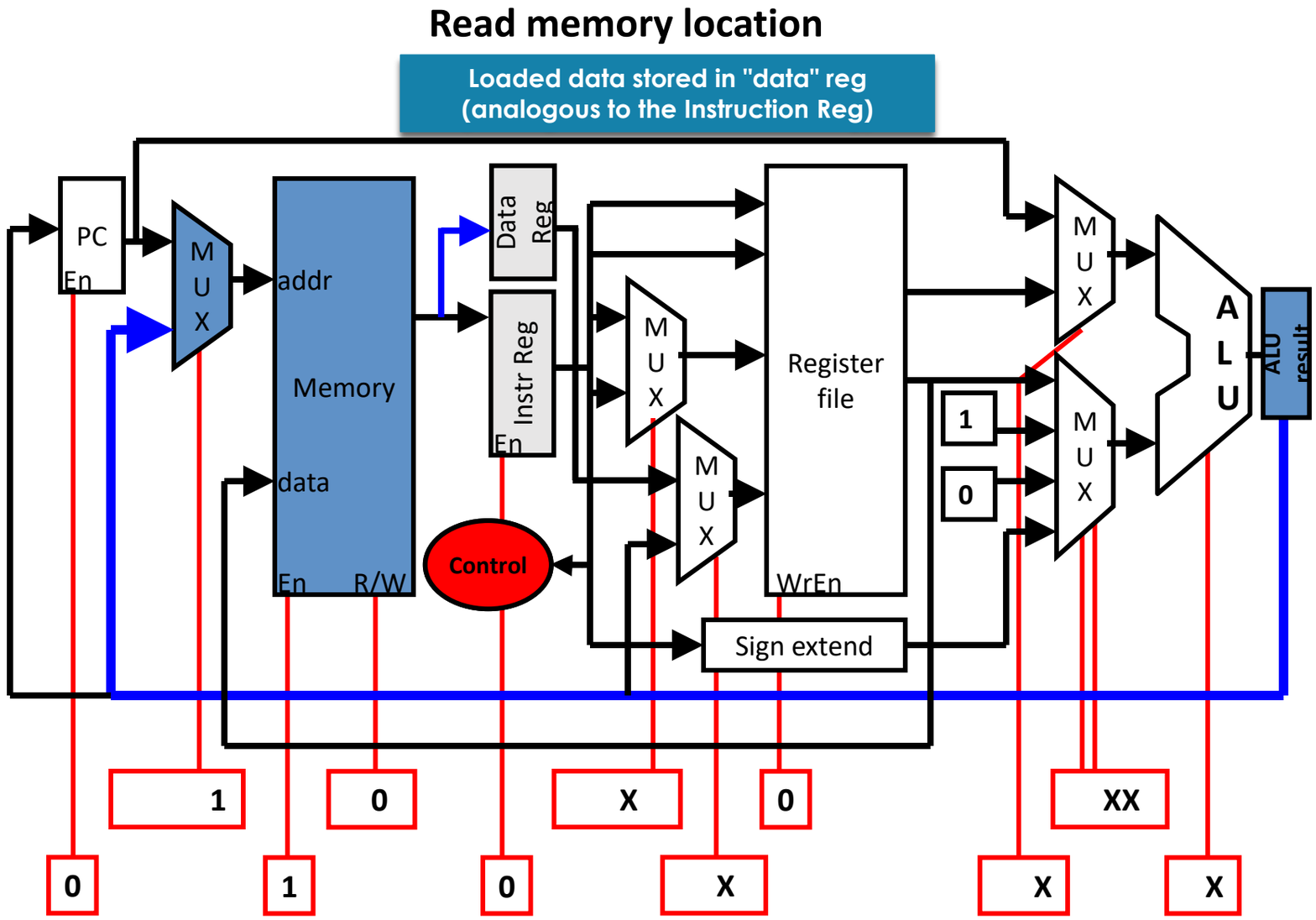
Control Rom (lw cycle 3)



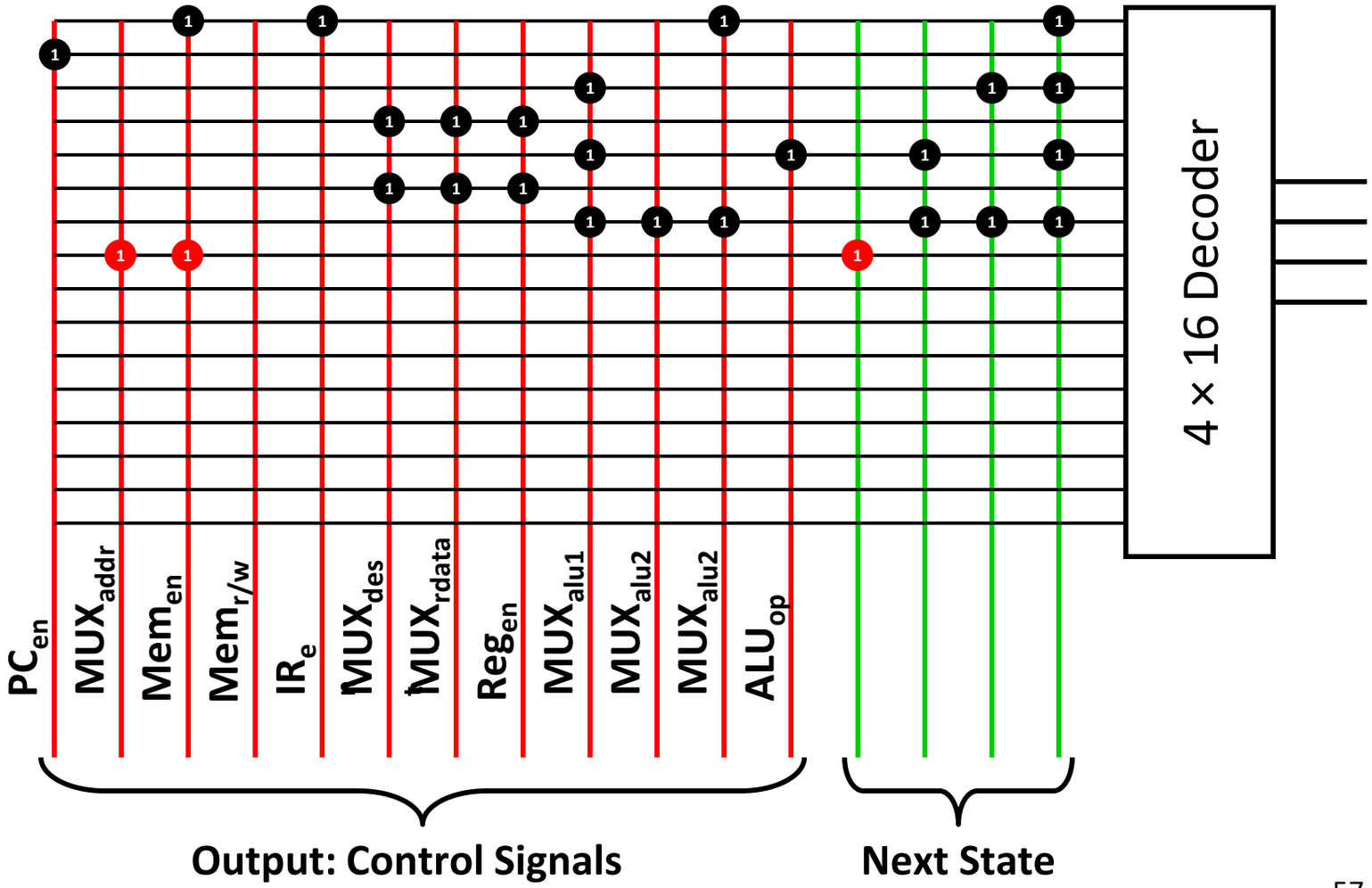
State 7: LW cycle 4



State 7: LW cycle 4

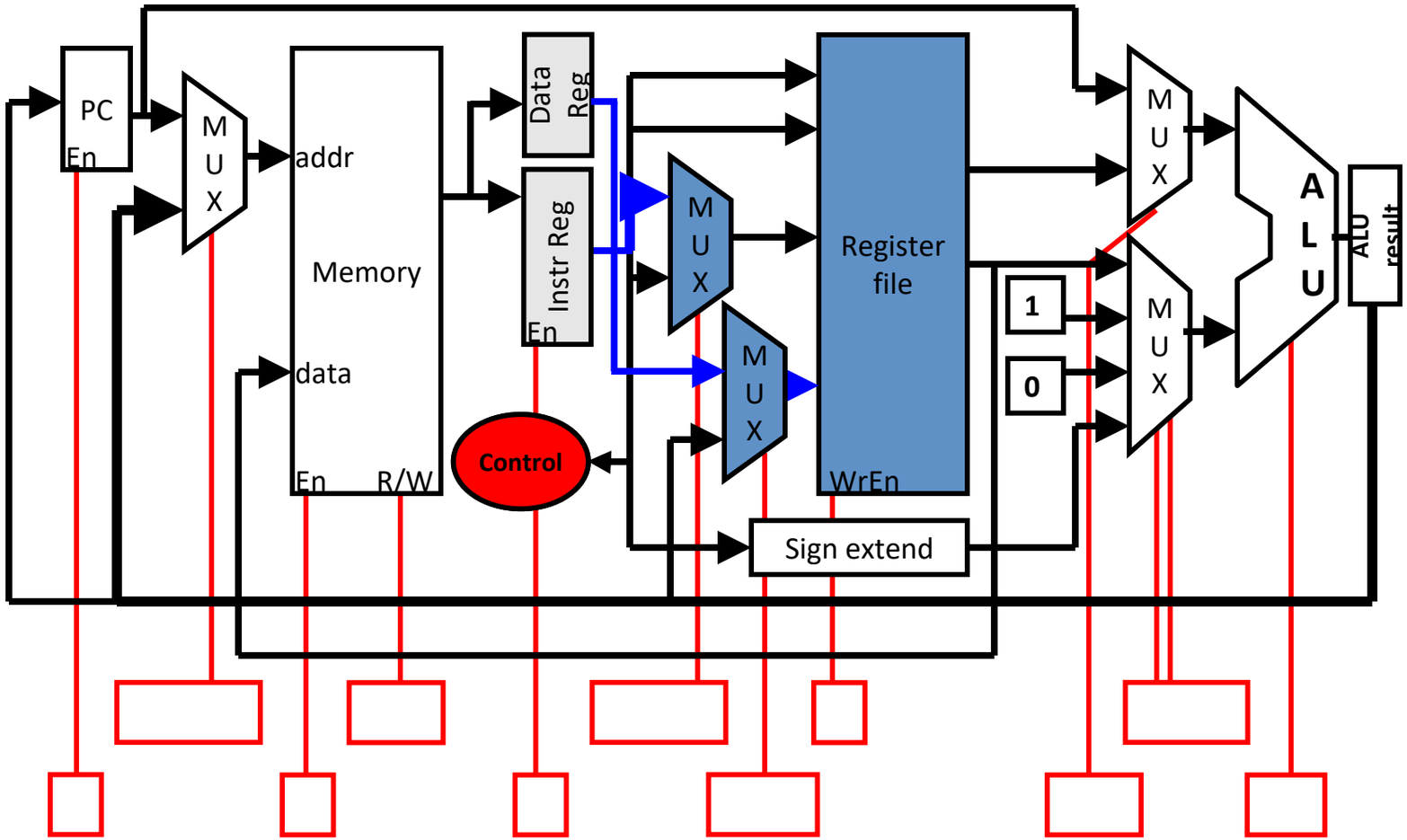


Control Rom (lw cycle 4)



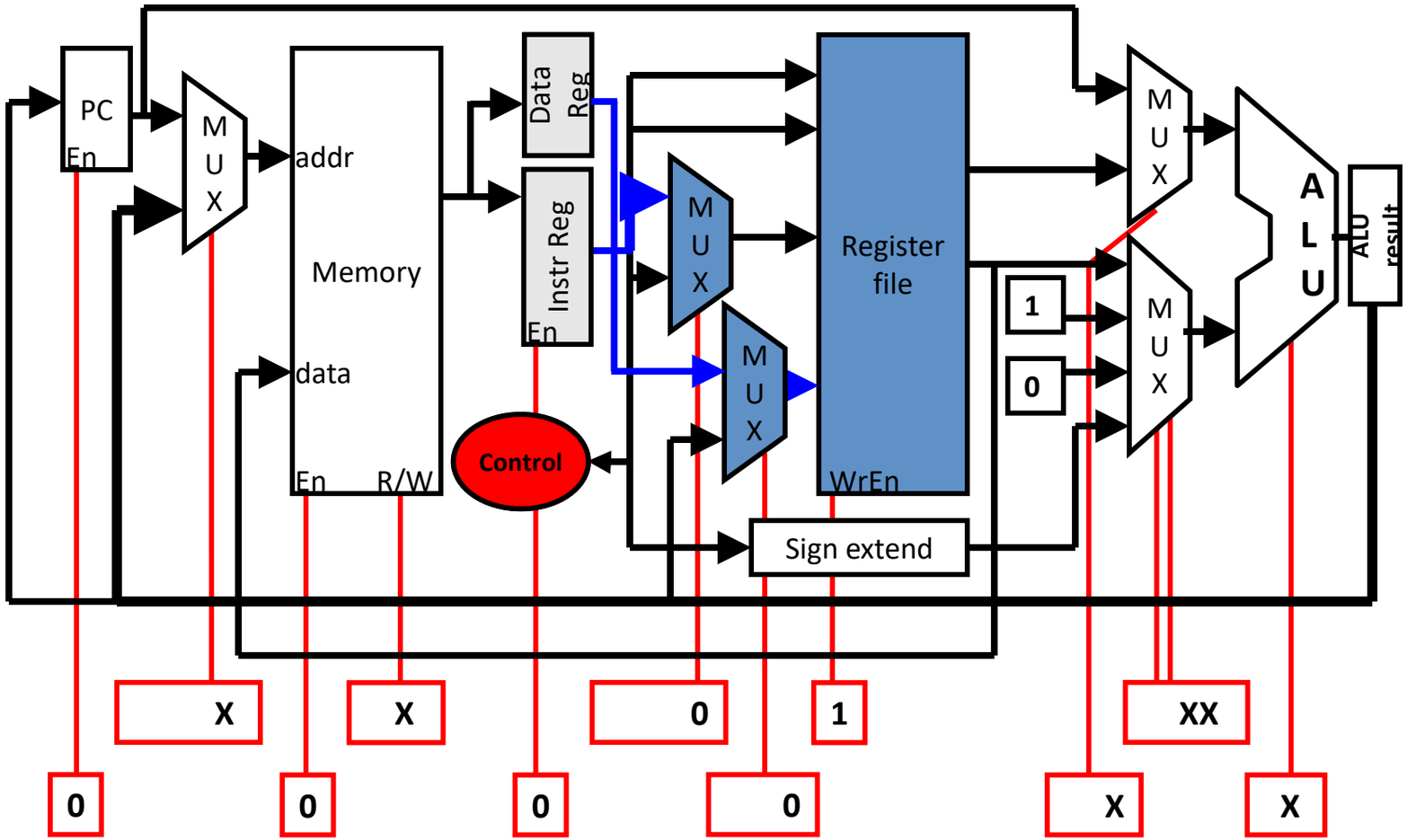
State 8: LW cycle 5

Write memory value to register file

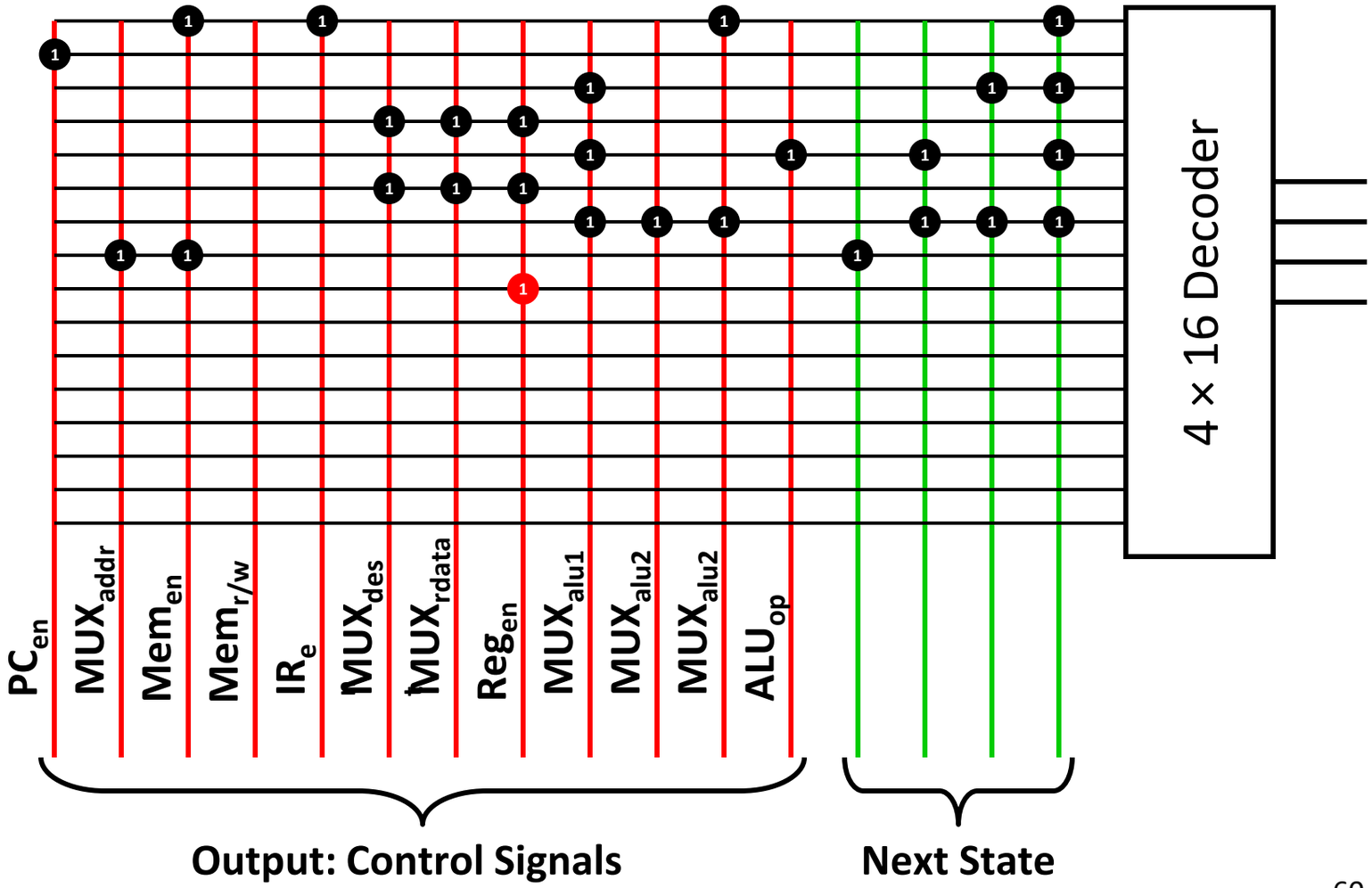


State 8: LW cycle 5

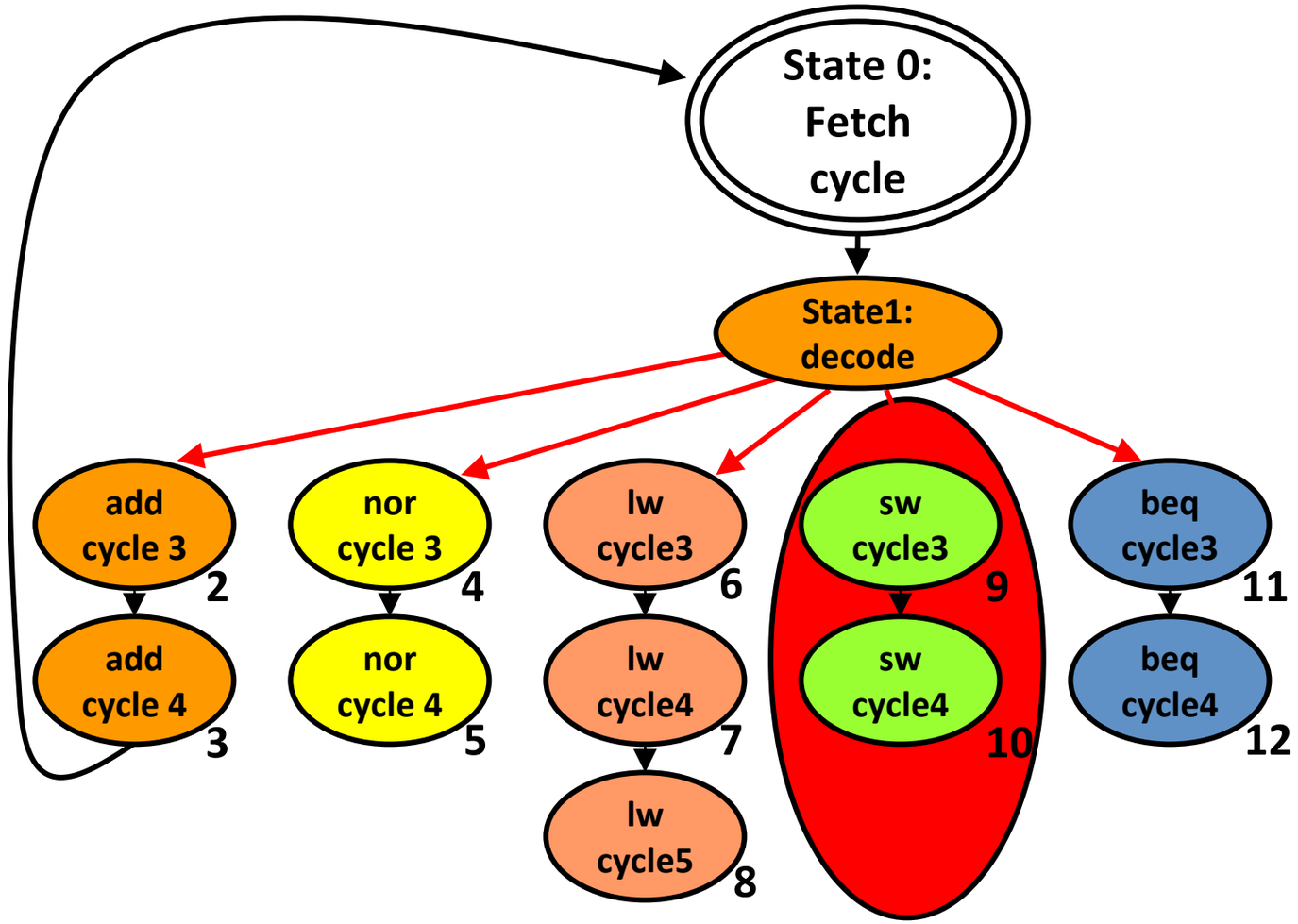
Write memory value to register file



Control Rom (lw cycle 5)

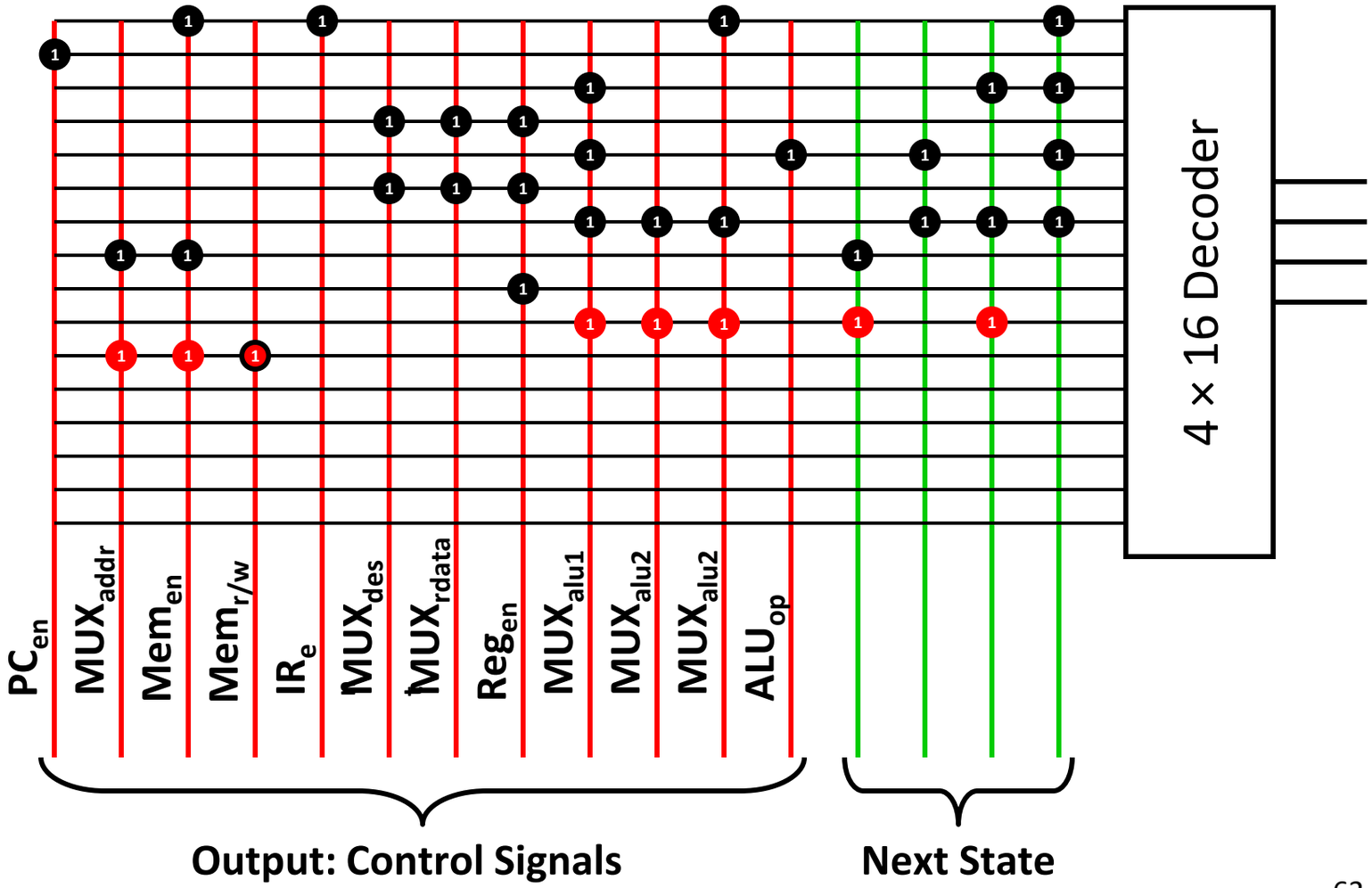


Return to State 0: Fetch cycle to execute the next instruction

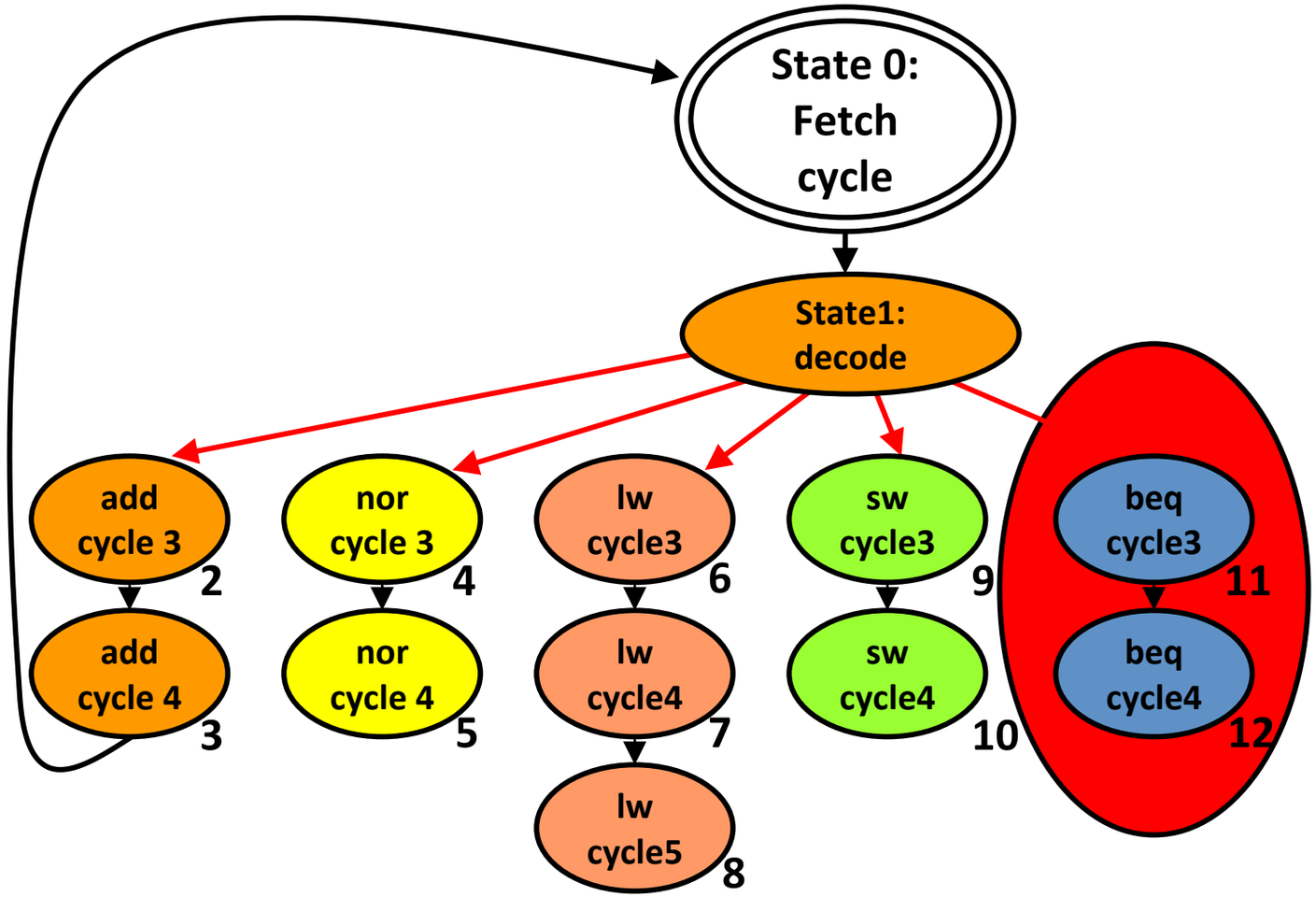


Control Rom (sw cycles 3 and 4)

Same as lw, except $Mem_{r/w}$ and Next State



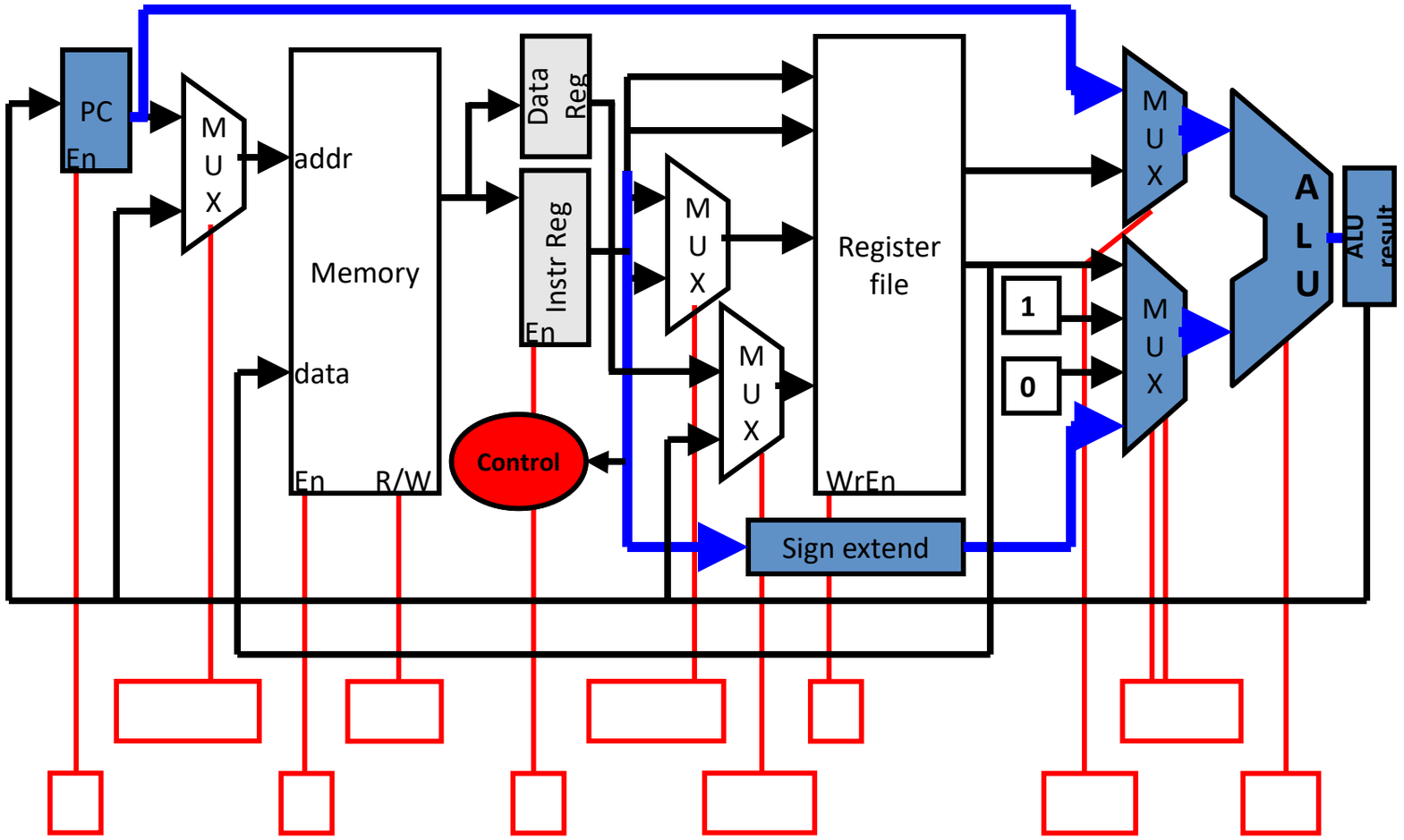
Return to State 0: Fetch cycle to execute the next instruction



State 11: beq cycle 3

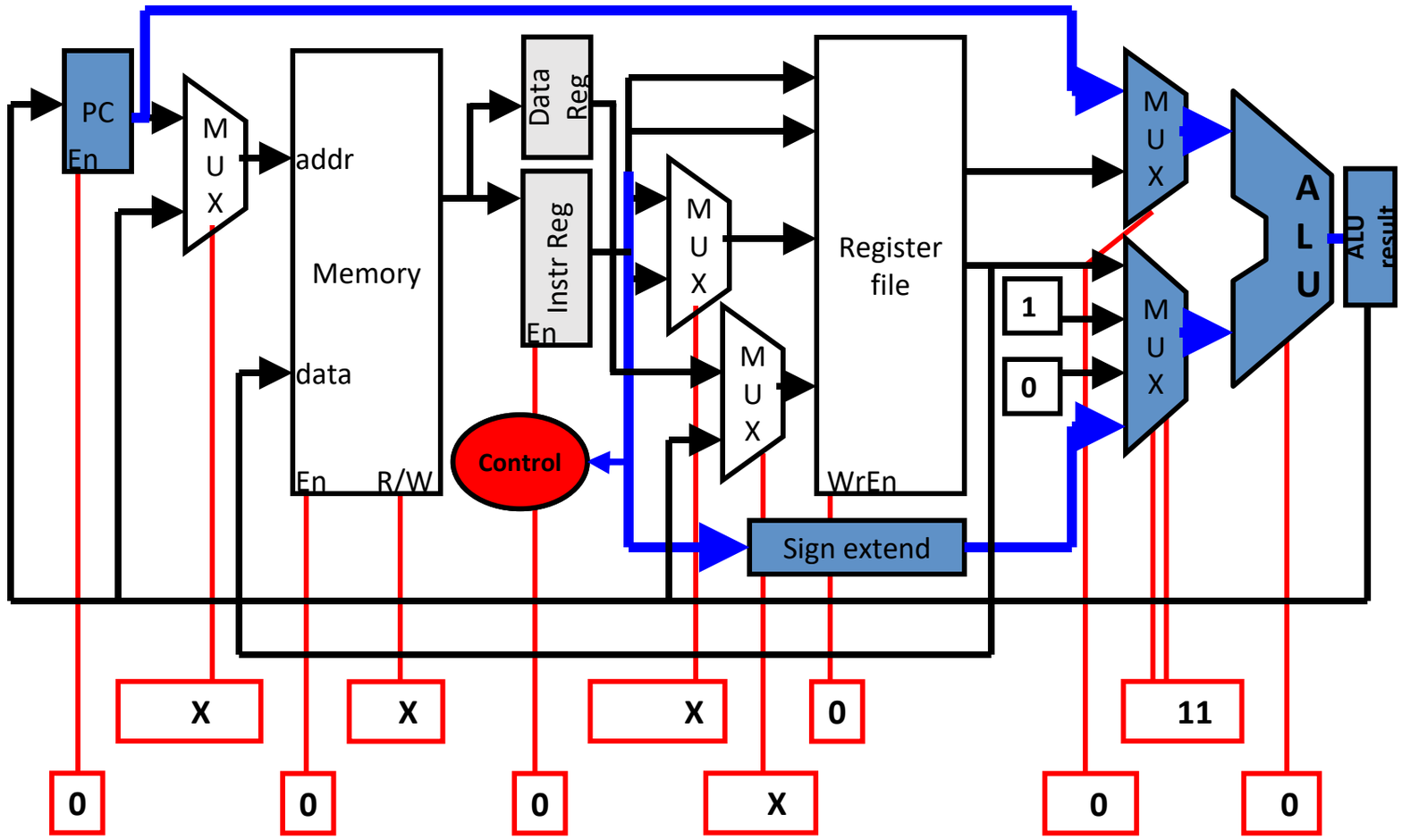
Poll: What will the control bits be?

Calculate target address for branch

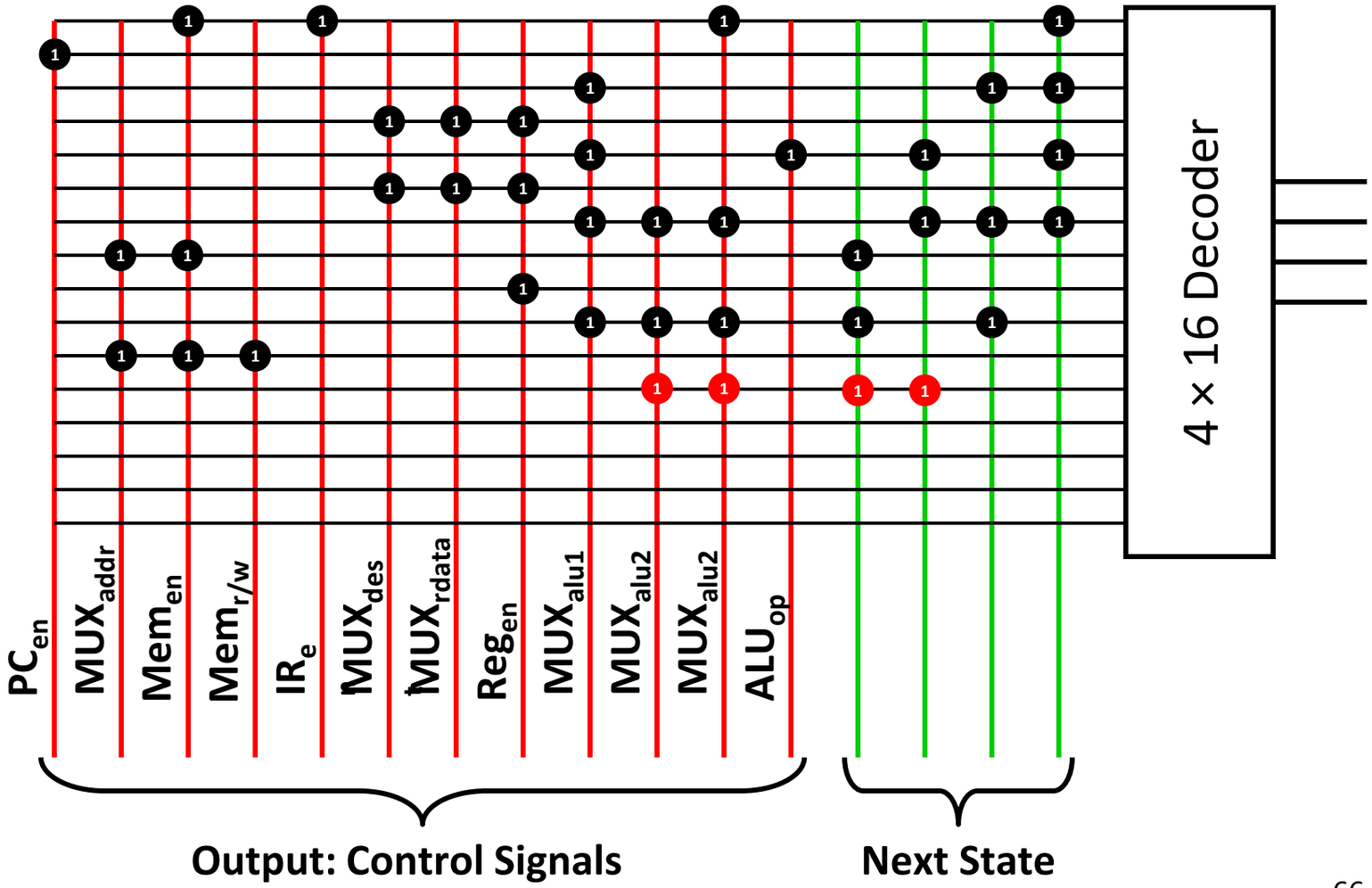


State 11: beq cycle 3

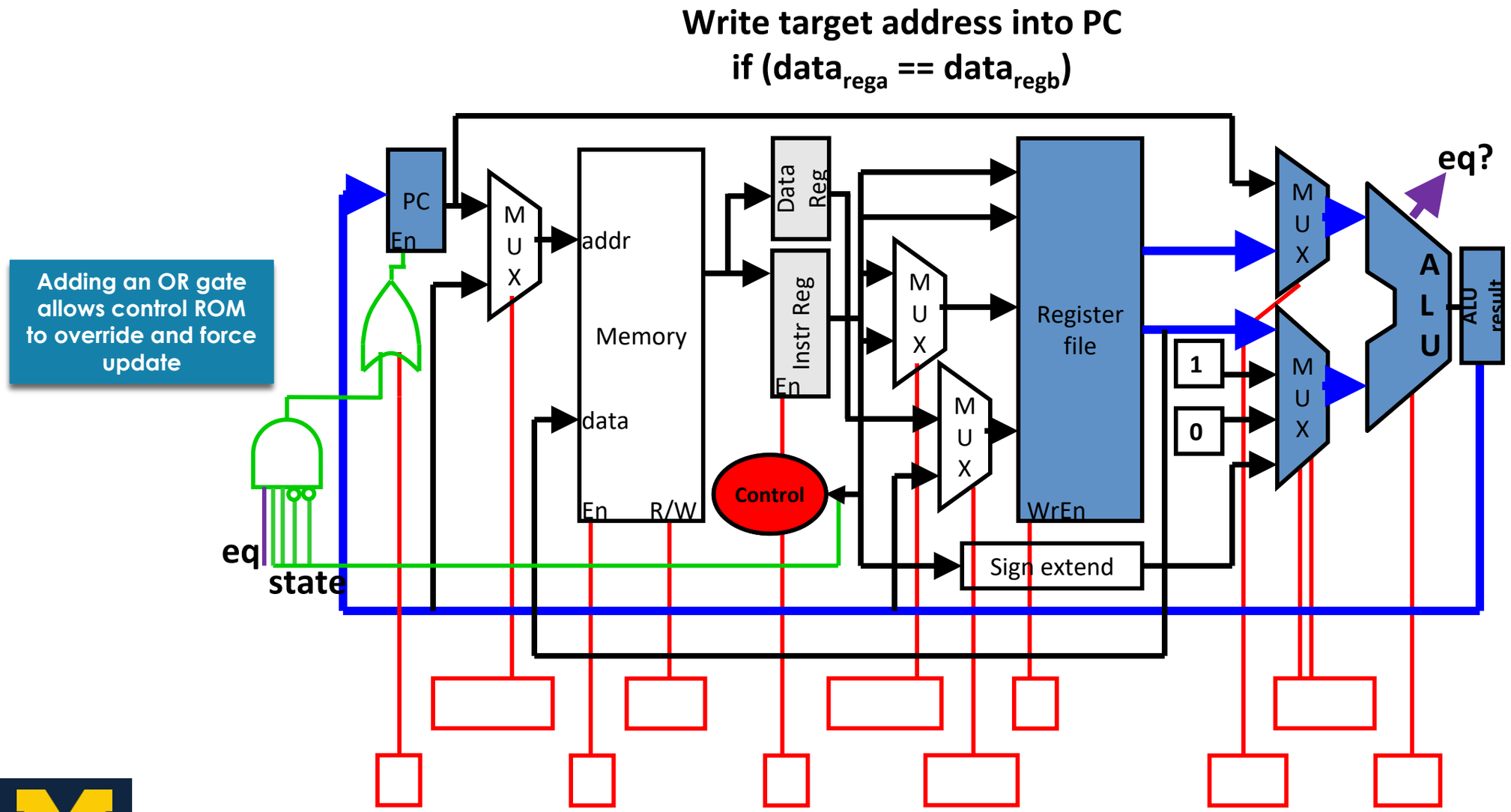
Calculate target address for branch



Control Rom (beq cycle 3)

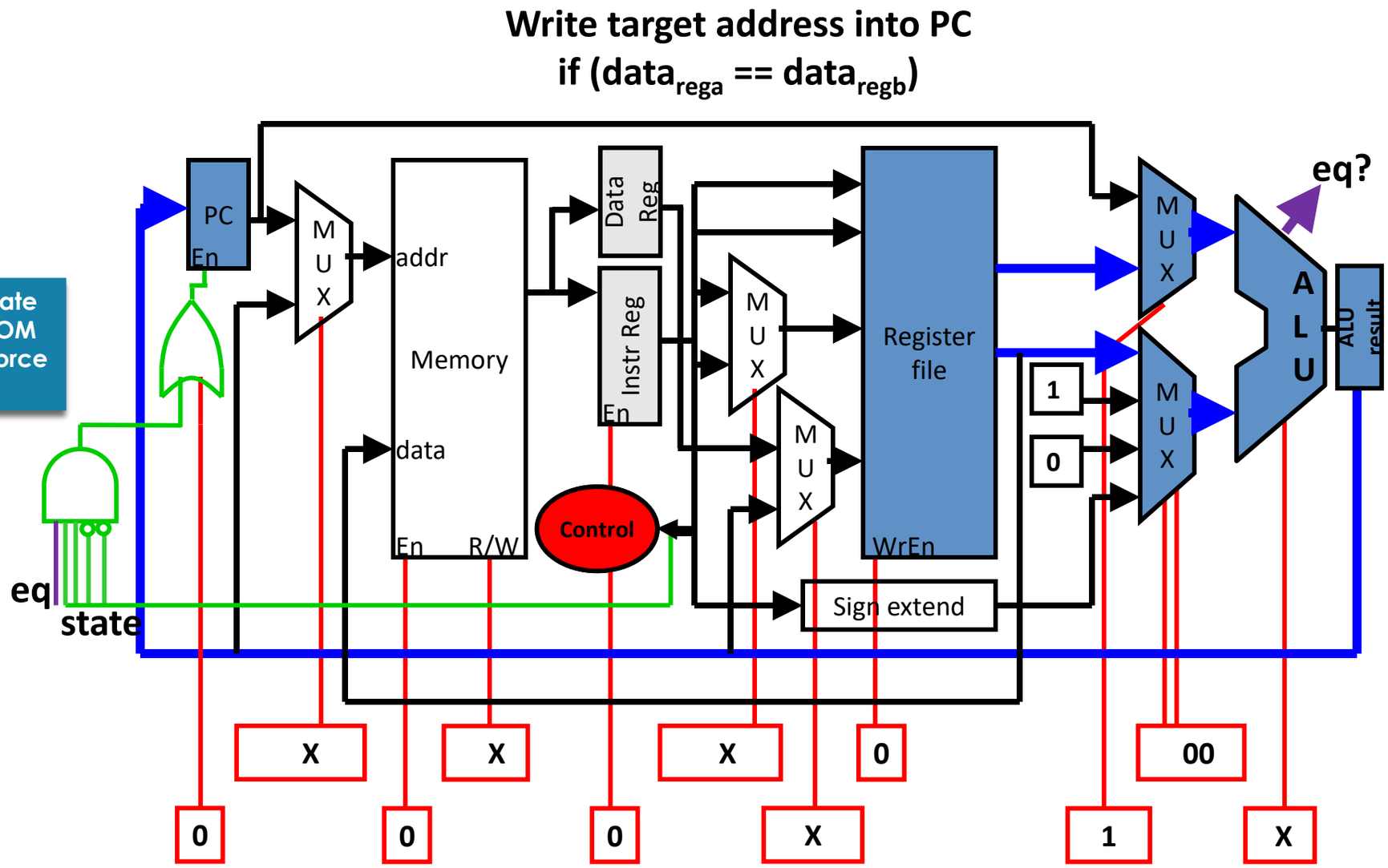


State 12: beq cycle 4

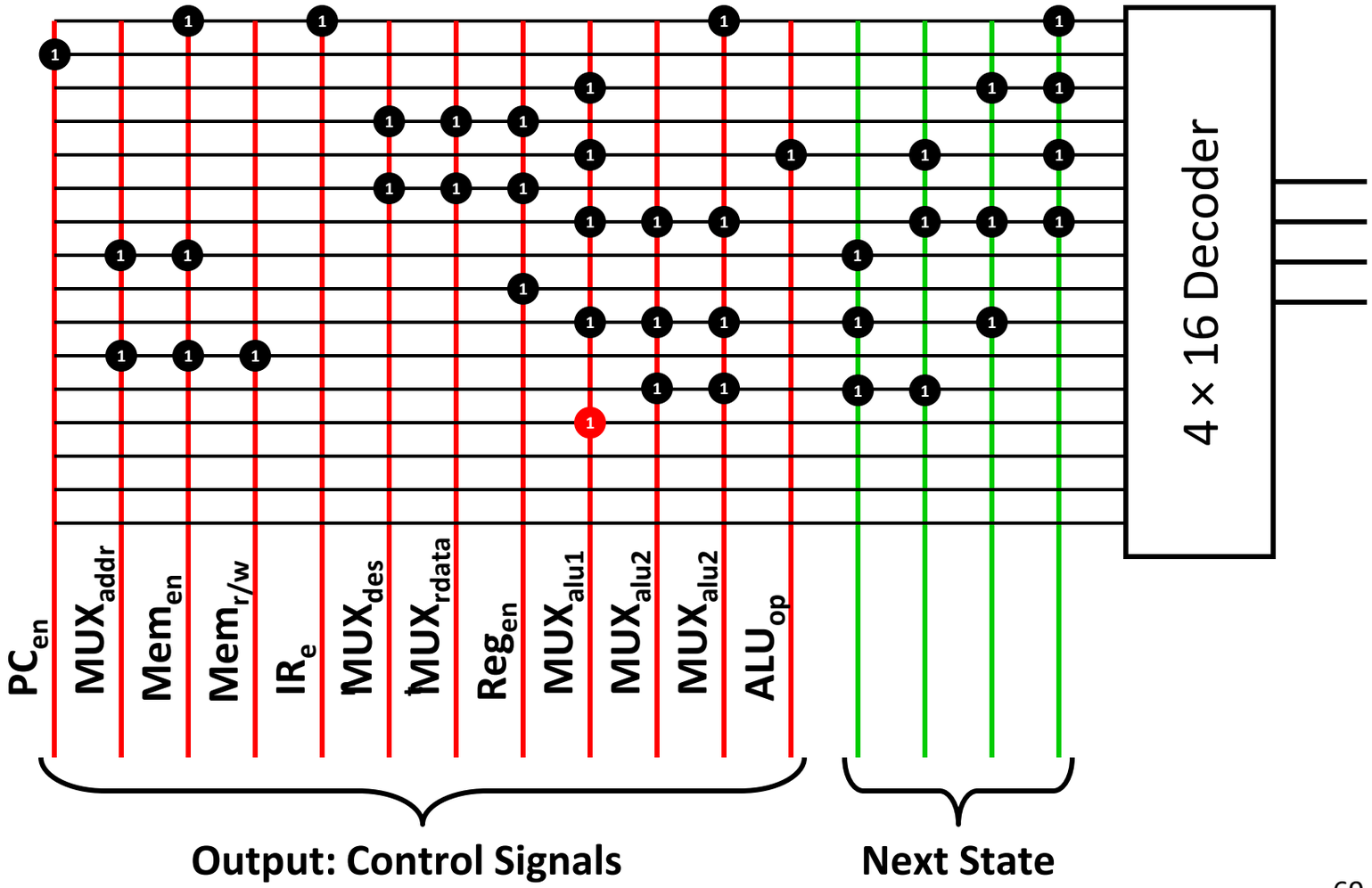


State 12: beq cycle 4

Adding an OR gate allows control ROM to override and force update



Control Rom (beq cycle 4)



Single vs Multi-cycle Performance

1 ns – Register File read/write time

2 ns – ALU/adder

2 ns – memory access

0 ns – MUX, PC access, sign extend, ROM

Poll: How many ns does SC take? MC?

1. Assuming the above delays, what is the best cycle time that the LC2k multi-cycle datapath could achieve? Single cycle?

2. Assuming the above delays, for a program consisting of 25 LW, 10 SW, 45 ADD, and 20 BEQ, which is faster?

Single vs Multi-cycle Performance

1 ns – Register File read/write time

2 ns – ALU/adder

2 ns – memory access

0 ns – MUX, PC access, sign extend, ROM

1. Assuming the above delays, what is the best cycle time that the LC2k multi-cycle datapath could achieve? Single cycle?

   MC: MAX(2, 1, 2, 2, 1) = 2ns

   SC: 2 + 1 + 2 + 2 + 1 = 8 ns

2. Assuming the above delays, for a program consisting of 25 LW, 10 SW, 45 ADD, and 20 BEQ, which is faster?

   SC: 100 cycles * 8 ns = 800 ns

   MC: (25*5 + 10*4 + 45*4 + 20*4)cycles * 2ns = 850 ns

Single and Multi-cycle performance

- Wait, multi-cycle is worse??
- For our ISA, most instructions take about the same time
- Multi-cycle shines when some instructions take much longer
- E.g. if we add a long latency instruction like multiply:
  - Let's say operation takes 10 ns, but could be split into 5 stages of 2 ns
  - SC: clock period = 16 ns, performance is 1600 ns
  - MC: clock period = 2 ns, performance is 850 ns

Performance Metrics – Execution time

- What we really care about in a program is **execution time**
  - **Execution time** = total instructions executed X CPI x clock period
  - The "Iron Law" of performance
- CPI = **average** number of clock **cycles per instruction** *for an application*
- To calculate multi-cycle CPI we need:
  - Cycles necessary for each type of instruction
  - Mix of instructions executed in the application (dynamic instruction execution profile)

Poll: What are the units of (instructions executed x CPI x clock period)?

Datapath Summary

- Single-cycle processor
  - CPI = 1 (by definition)
  - clock period = ~10 ns
- Multi-cycle processor
  - CPI = ~4.25
  - clock period = ~2 ns
- Better design:
  - CPI = 1
  - clock period = ~2ns
- How??
  - Work on multiple instructions at the same time