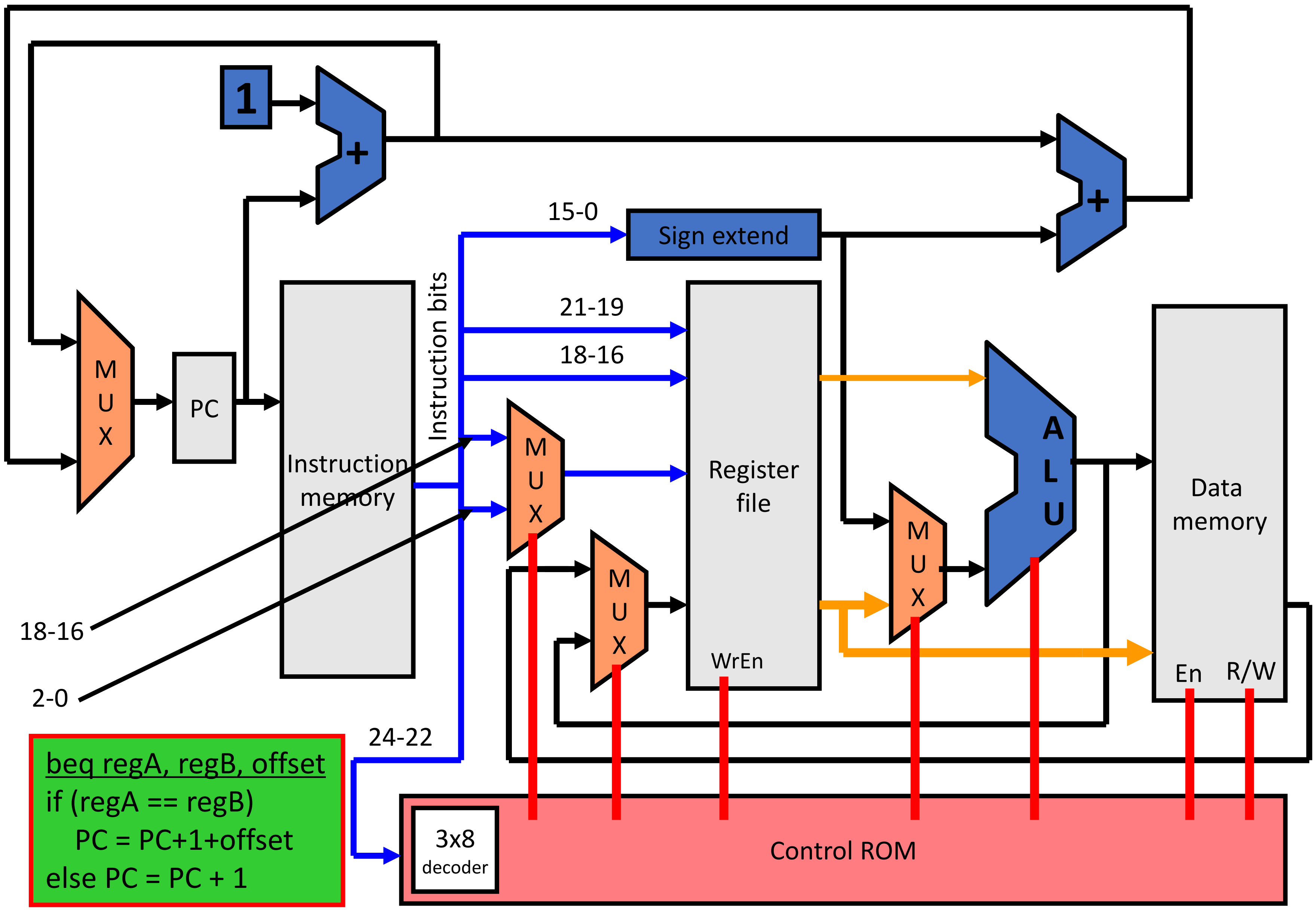
EECS 370 - Lecture 12

Multi-cycle + Introduction to Pipelining

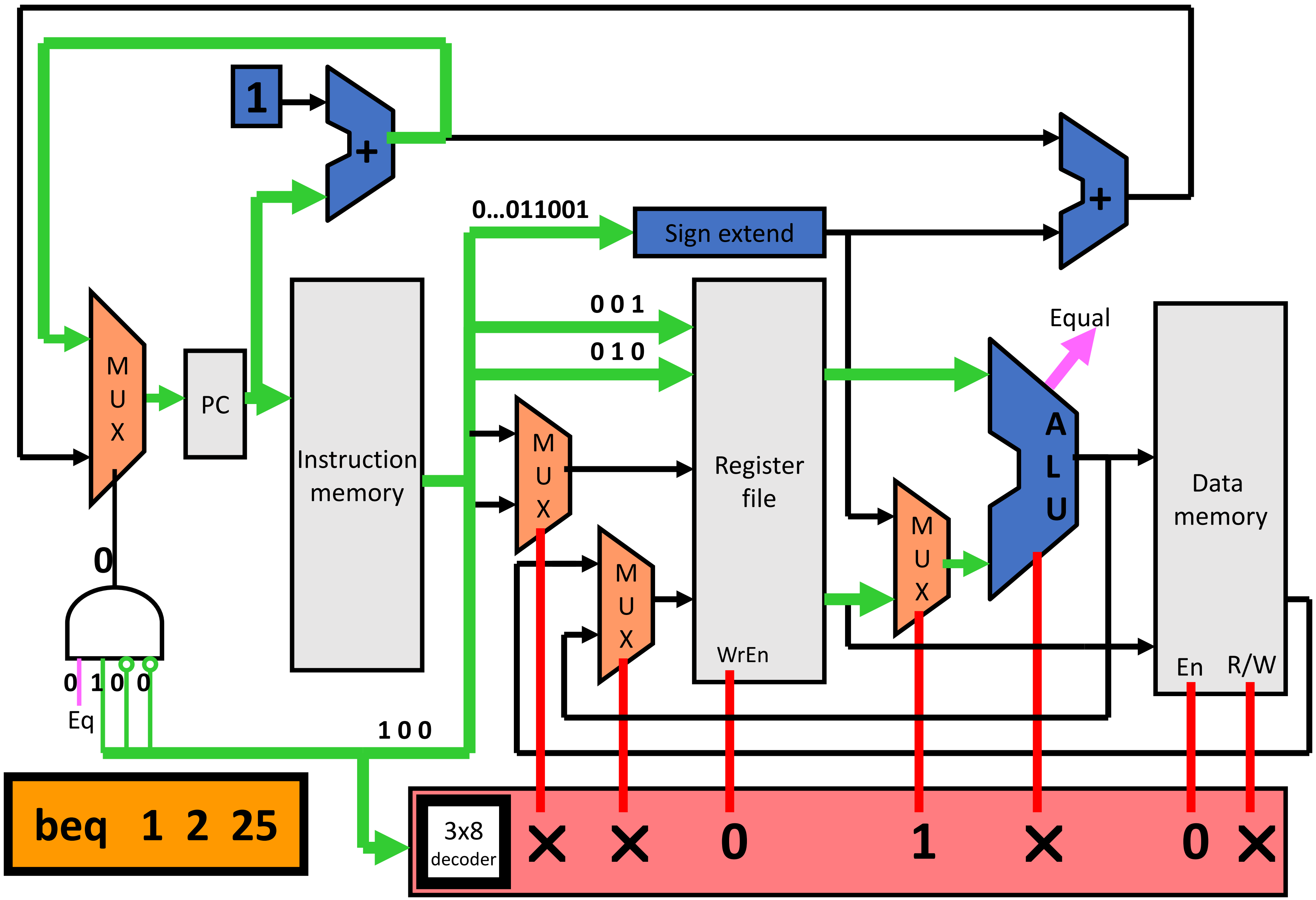
Announcements

- P2
  - Three parts: part a is due **today**
- HW 2
  - Posted on website, due next **Mon**
- Midterm exam **Thu 6-8 pm**
  - 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
- Staff led review session on Sunday
  - Will be recorded, see Ed post
- Lecture on Tuesday will also be review

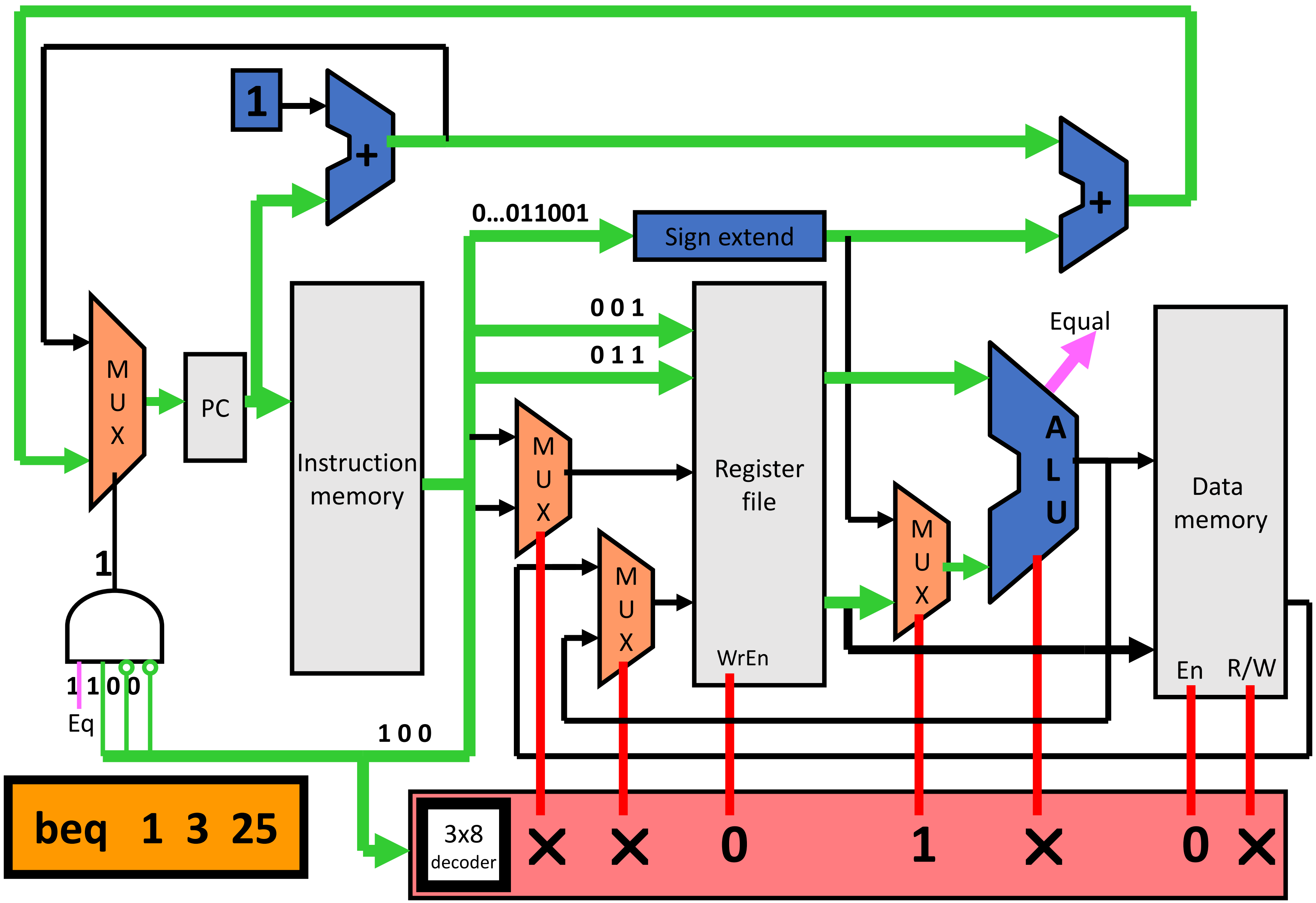
Executing a BEQ Instruction



Executing “not taken” BEQ Instruction on LC2K Datapath



Executing a “taken” BEQ Instruction on LC2K Datapath



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
- Won’t cover here (you don’t need to know it explicitly)
  - Studio recordings go over it if you are curious

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

Clock Period Example

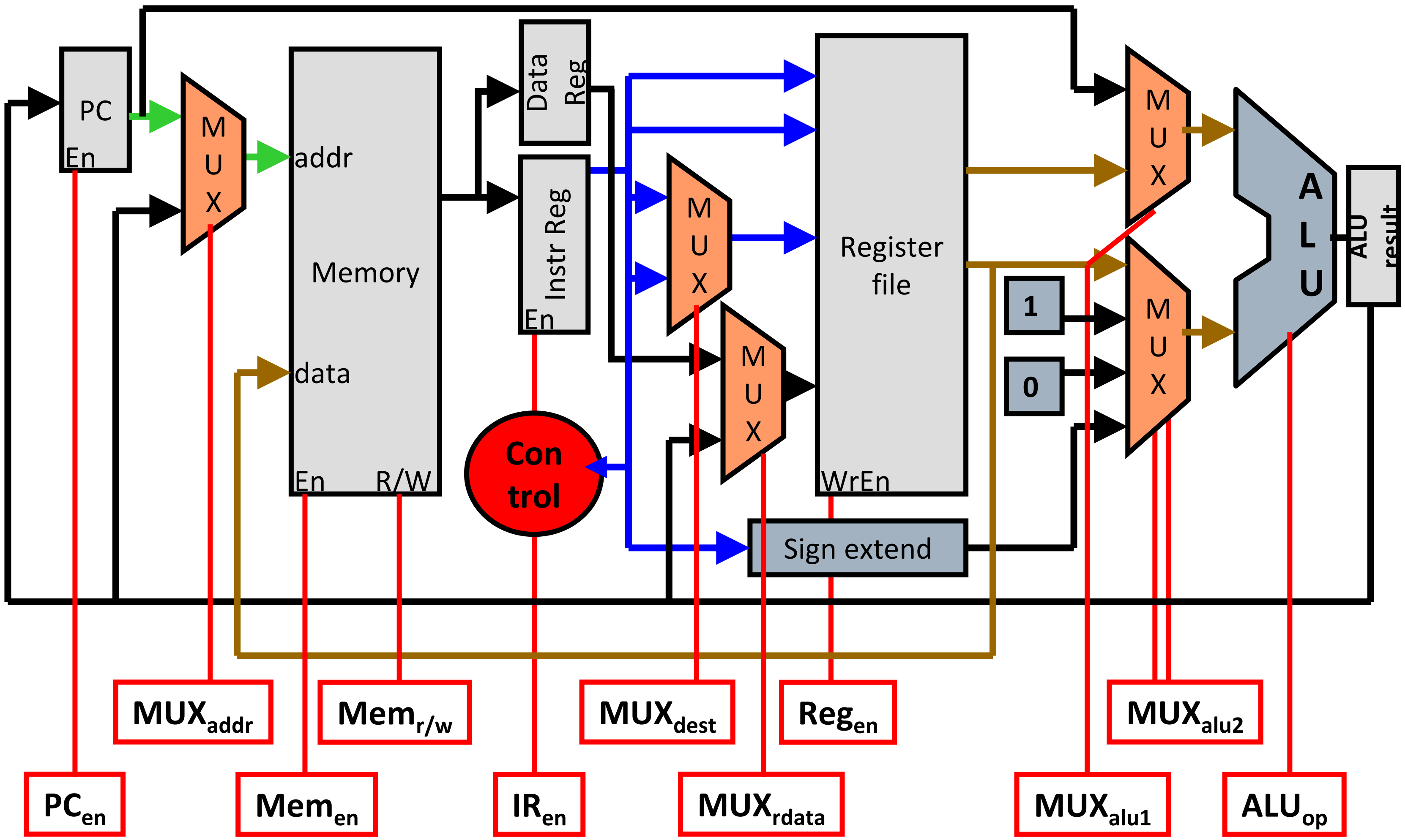
- 1 ns – Register read/write time
- 2 ns – ALU/adder
- 2 ns – memory access
- 0 ns – MUX, PC access, sign extend, ROM

	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

Multiple-Cycle Execution

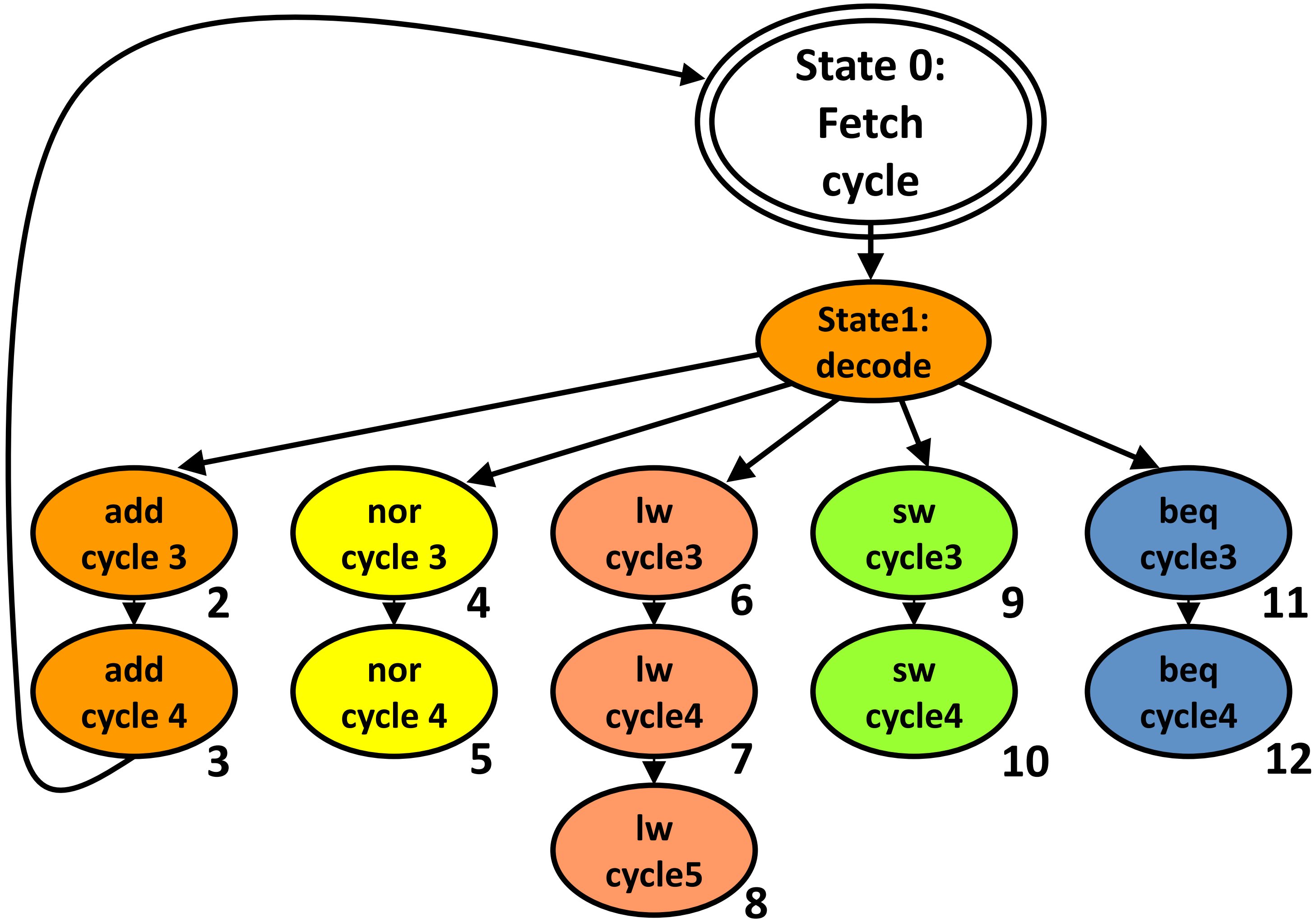
- Each instruction takes multiple cycles to execute
  - Cycle time is reduced
  - Slower instructions take more cycles
  - Faster instruction take fewer cycles
    - <u>We can start next instruction earlier, rather than just waiting</u>
  - Can reuse datapath elements each cycle
- What is needed to make 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.

Multi-cycle LC2 Datapath



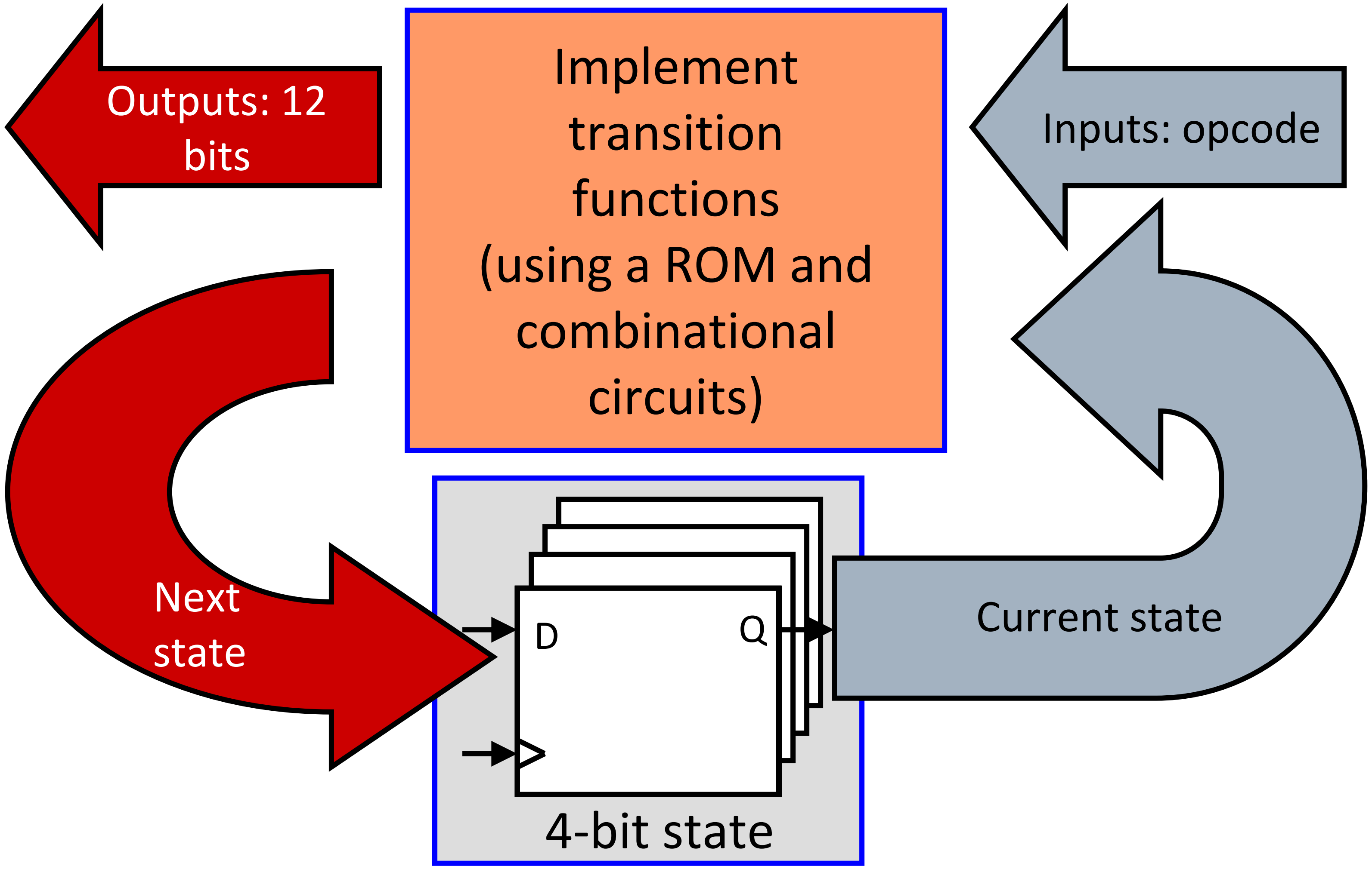
Each red signal comes from "Control" (implemented via ROM as before)

State machine for multi-cycle control signals (transition functions)

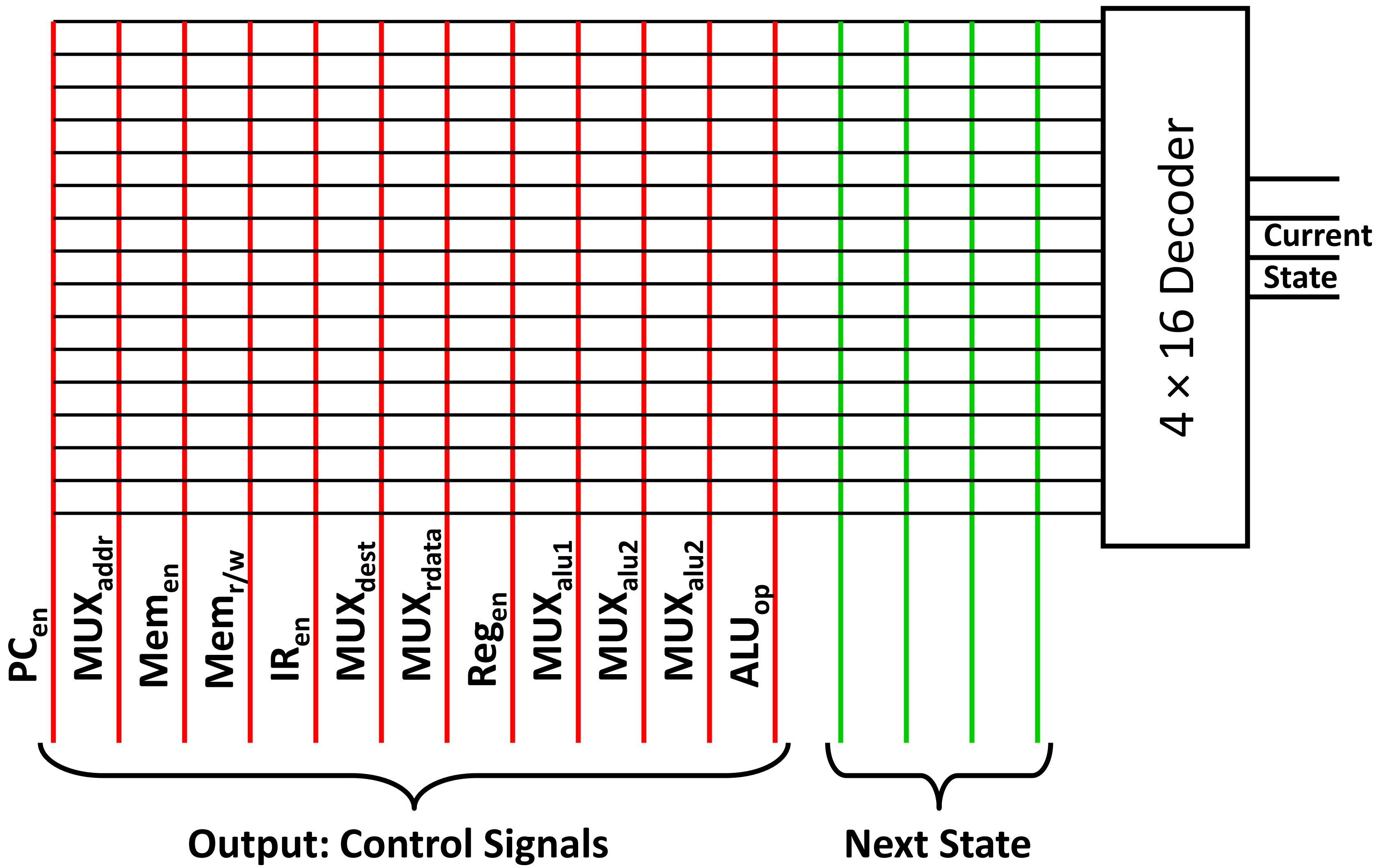


Note: we aren't worrying about JALR instruction in hardware going forward

Implementing FSM



Building the Control ROM

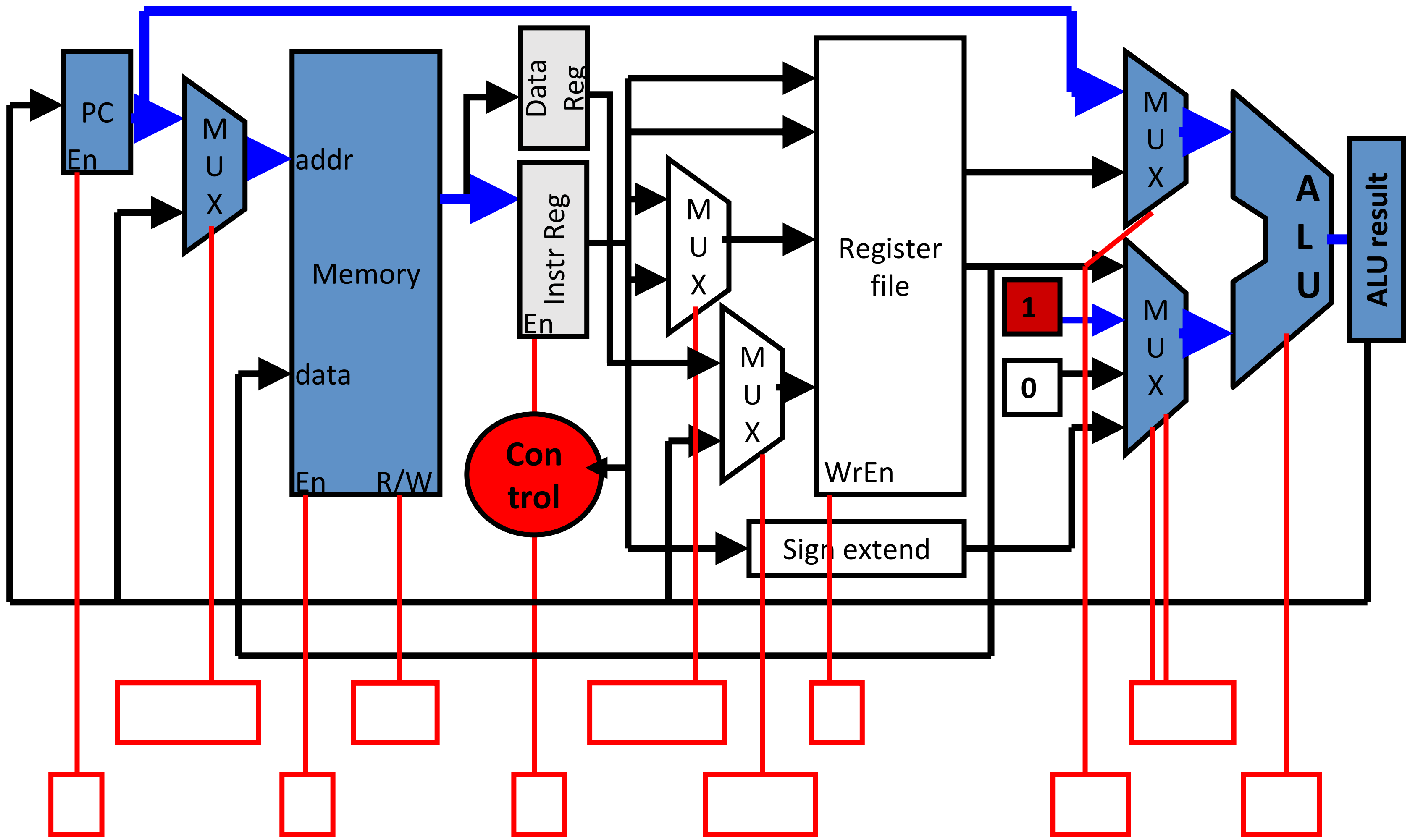


First Cycle (State 0) Fetch Instr

- What operations need to be done in the first cycle of executing any instruction?
  - Read memory[PC] and store into instruction register.
    - Must select PC in memory address MUX ($MUX_{addr} = 0$)
    - Enable memory operation ($Mem_{en} = 1$)
    - R/W should be (read) ($Mem_{r/w} = 0$)
    - Enable Instruction Register write ($IR_{en} = 1$)
  - Calculate PC + 1
    - Send PC to ALU ($MUX_{alu1} = 0$)
    - Send 1 to ALU ($MUX_{alu2} = 01$)
    - Select ALU add operation ($ALU_{op} = 0$)
  - $PC_{en} = 0$; $Reg_{en} = 0$; MUX_{dest} and $MUX_{rdata} = X$
- Next State: Decode Instruction

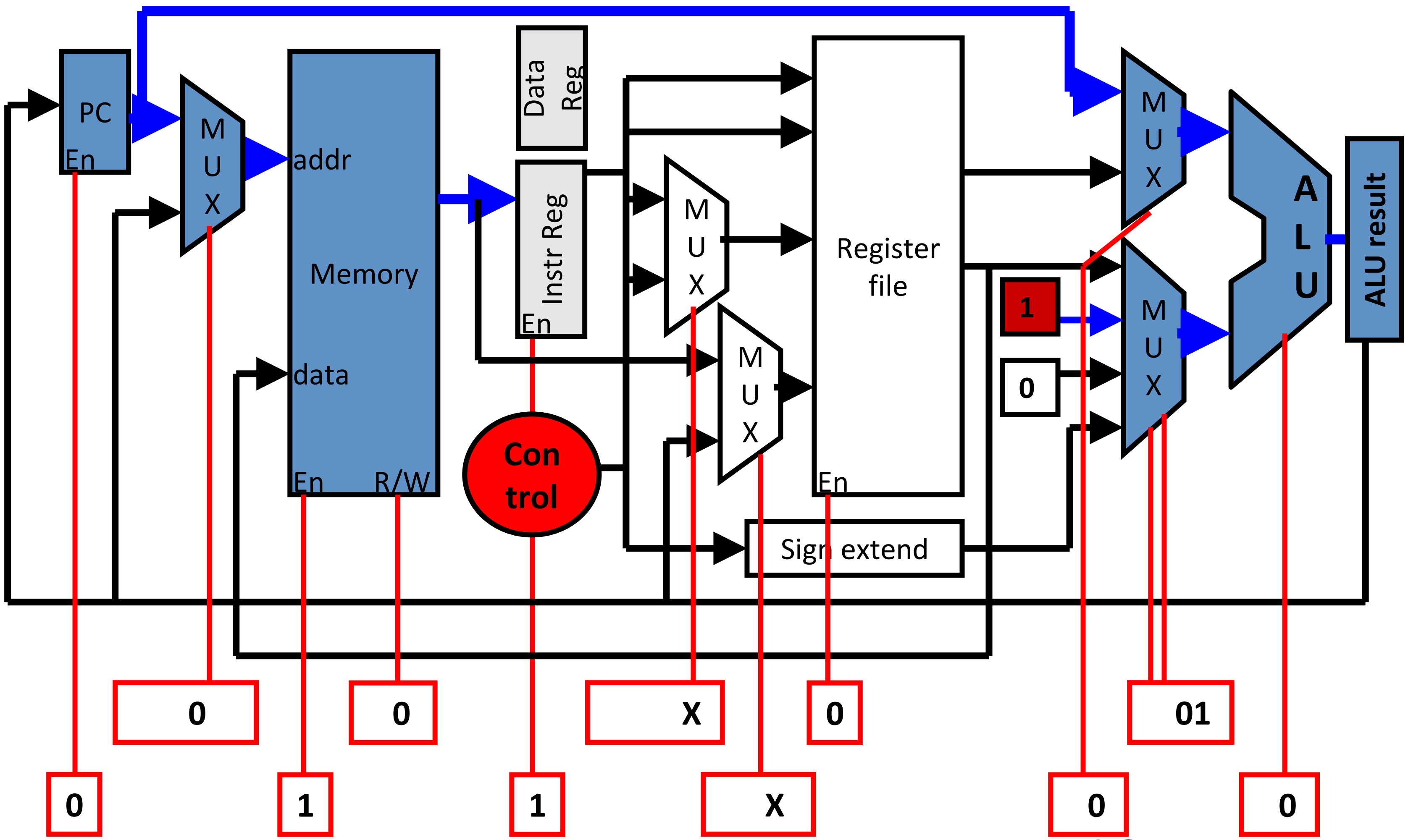
First Cycle (State 0) Fetch Instr

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

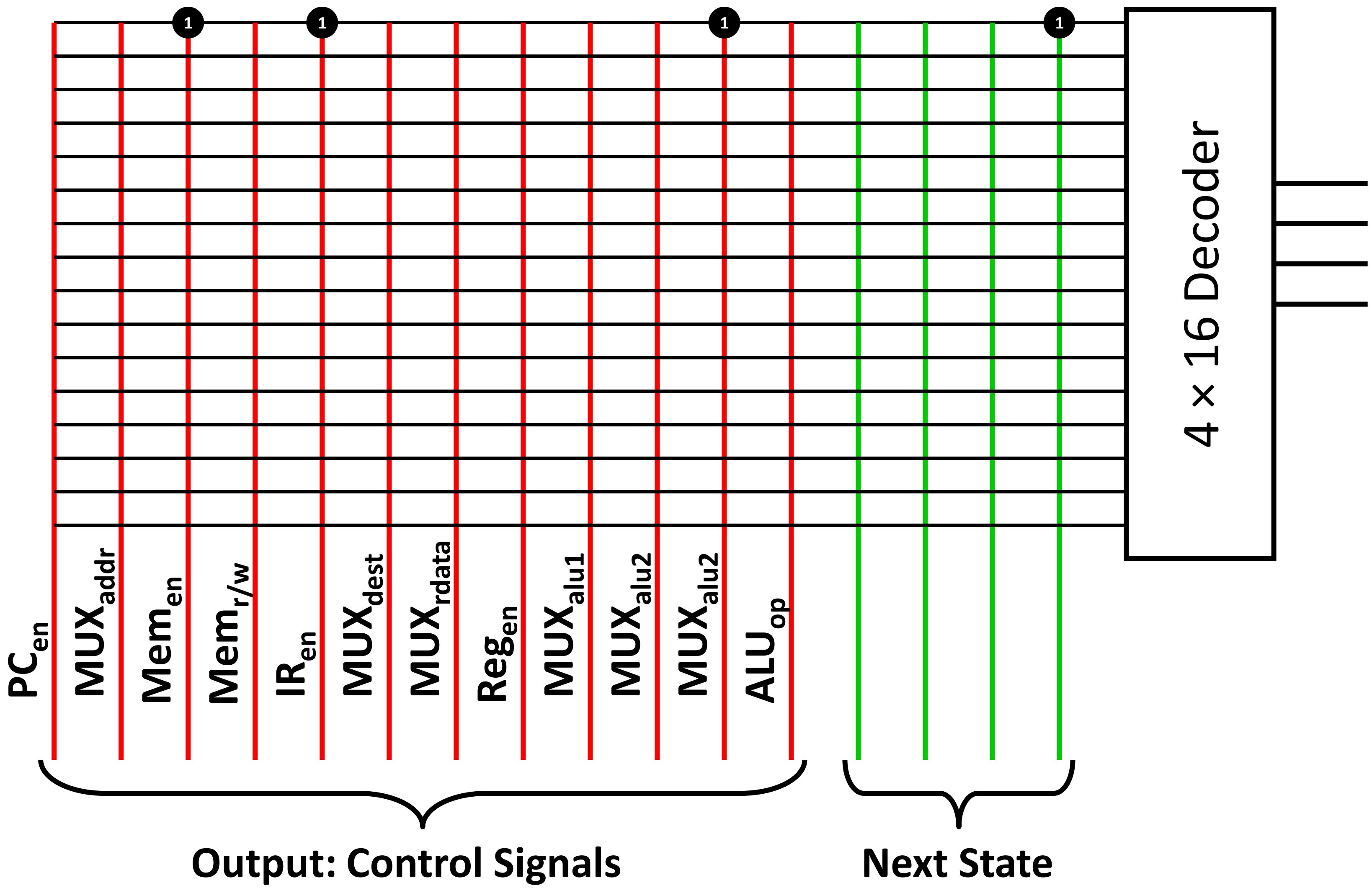


First Cycle (State 0) Fetch Instr

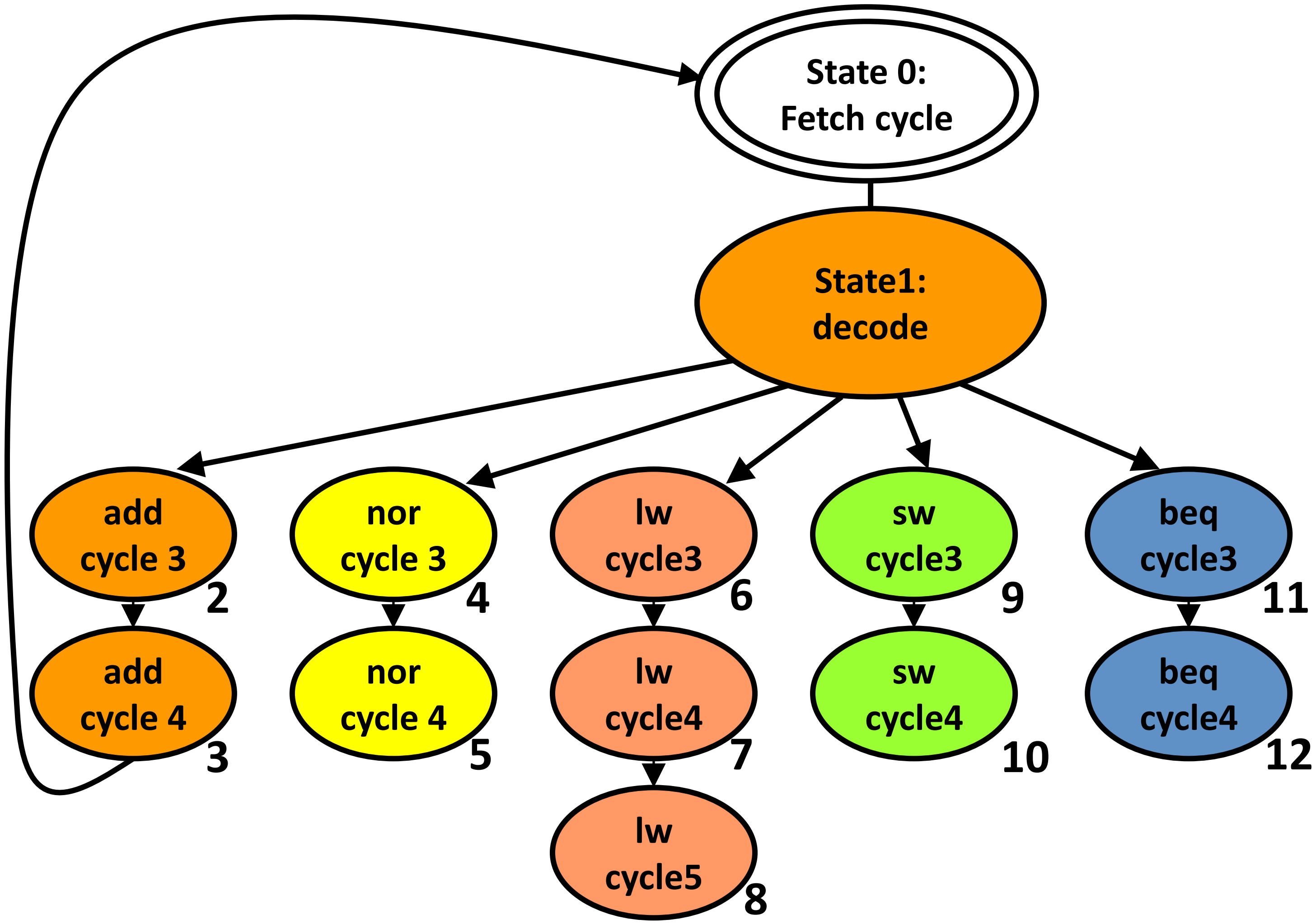
This is the same for all instructions
(since we don't know the instruction yet!)



Building the Control ROM



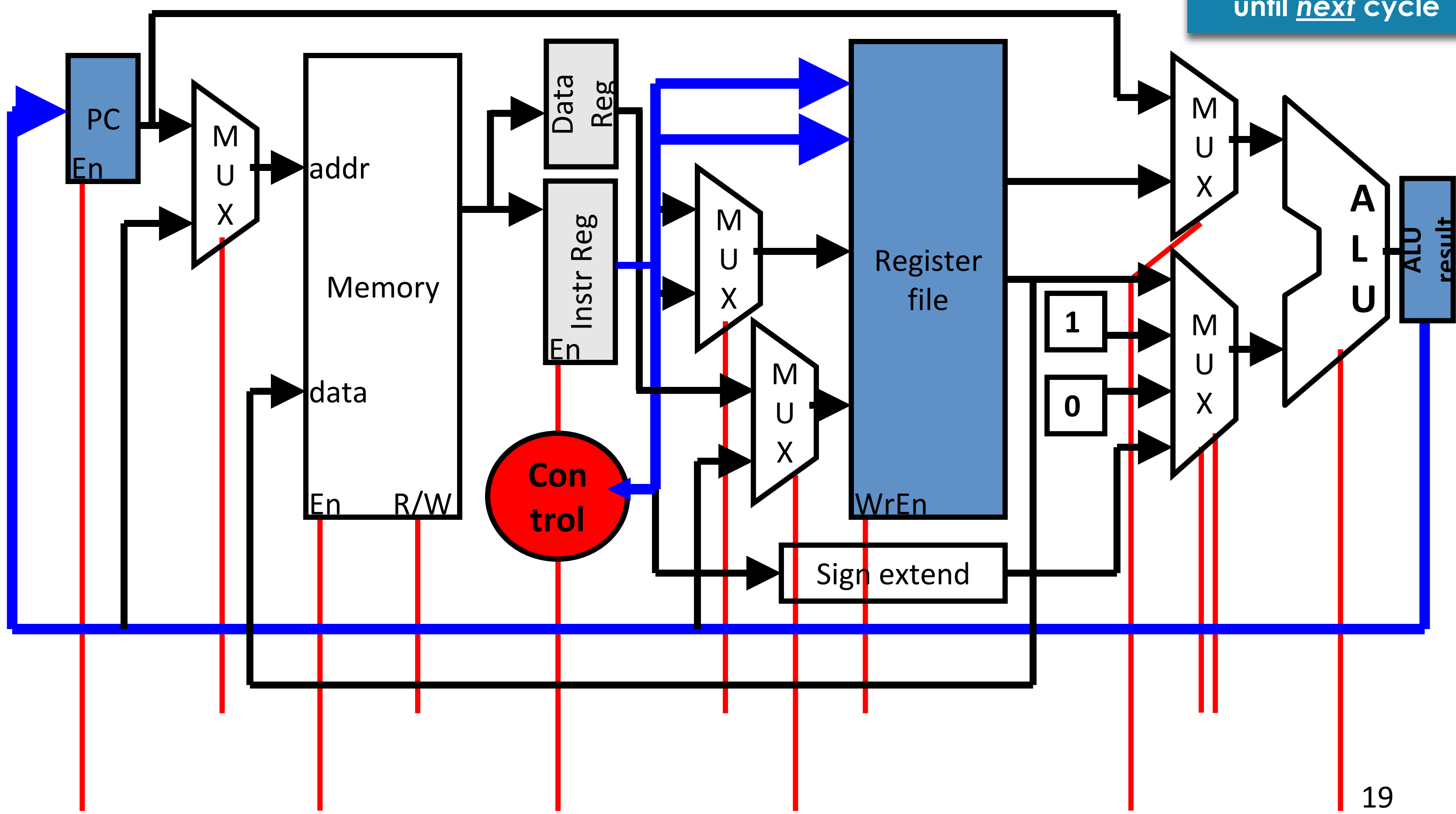
State 1: instruction decode



State 1: output function

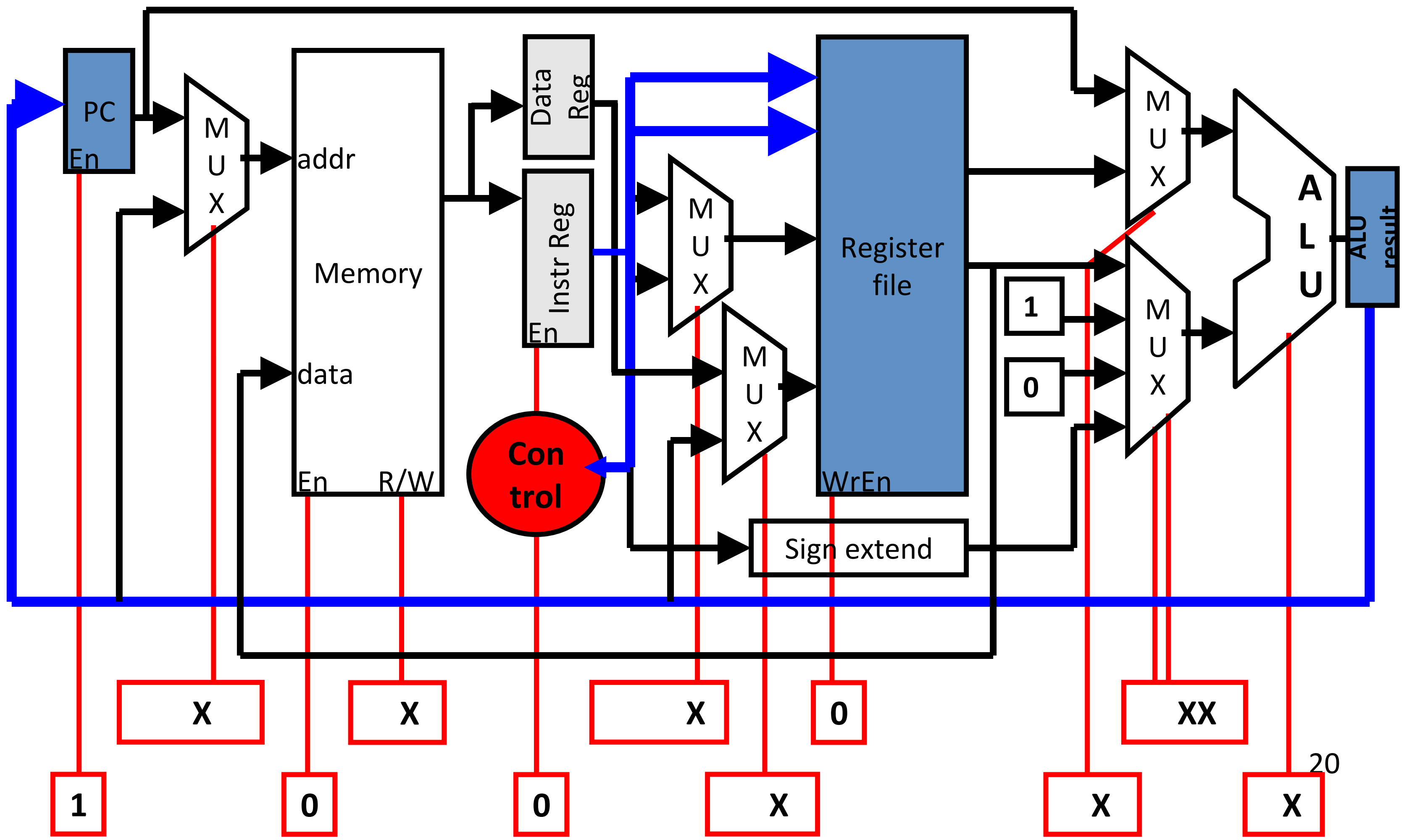
Update PC; read registers (regA and regB); use opcode to determine next state

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



State 1: output function

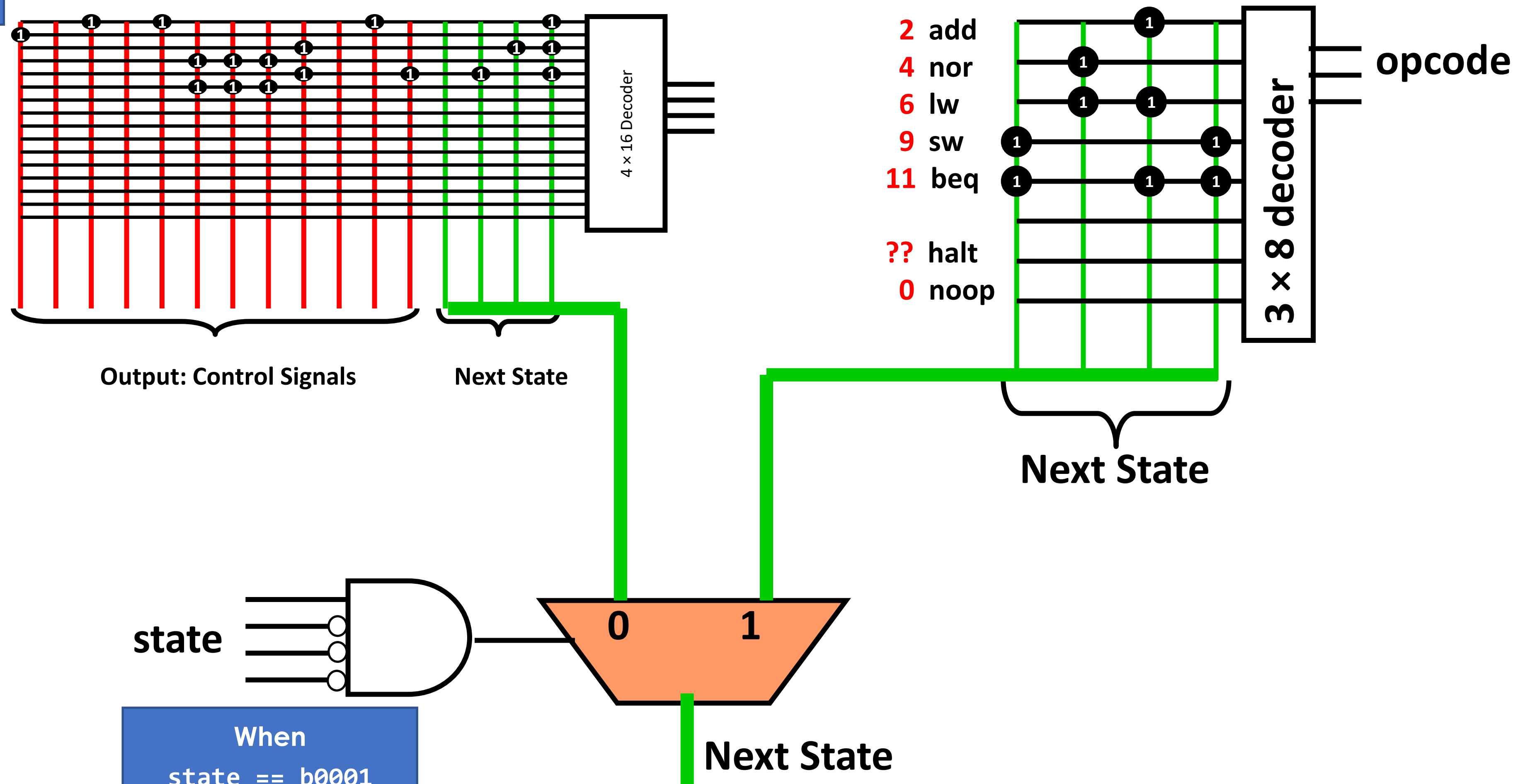
Update PC; read registers (regA and regB); use opcode to determine next state



Transitioning from Decode State

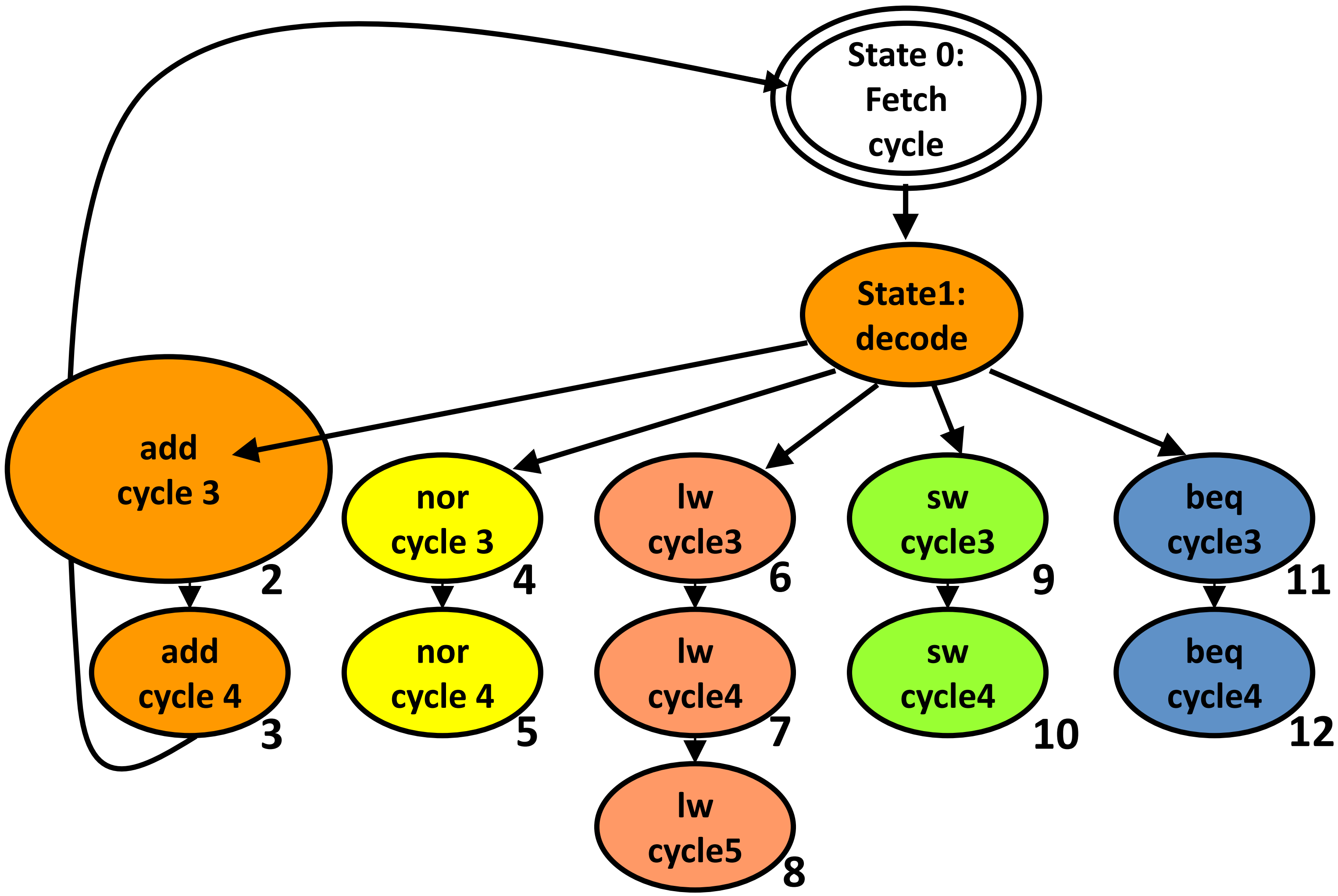
Which state we go to depends on opcode, can't just look at current state like before

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



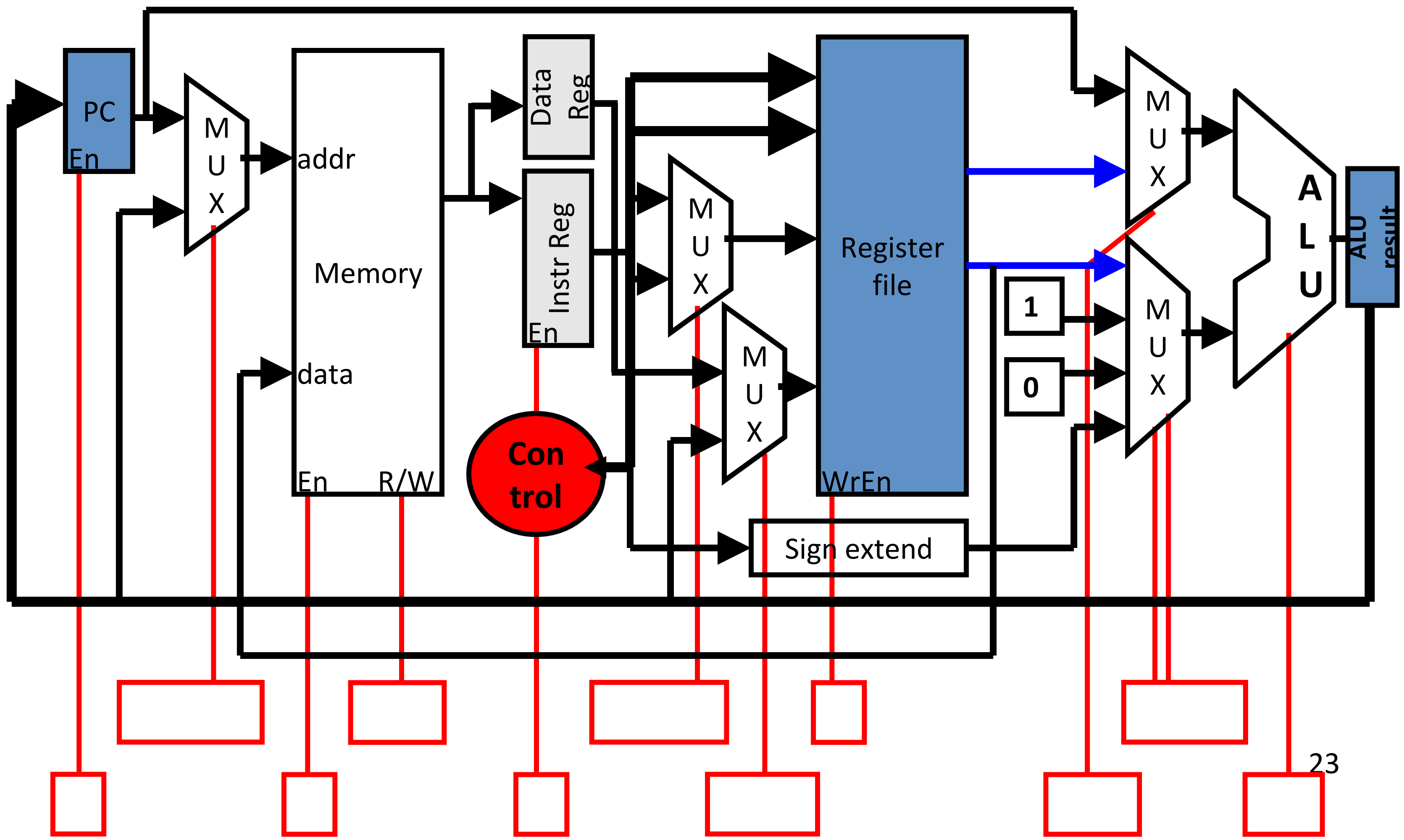
When `state == b0001` choose secondary ROM for next state logic

State 2: Add cycle 3



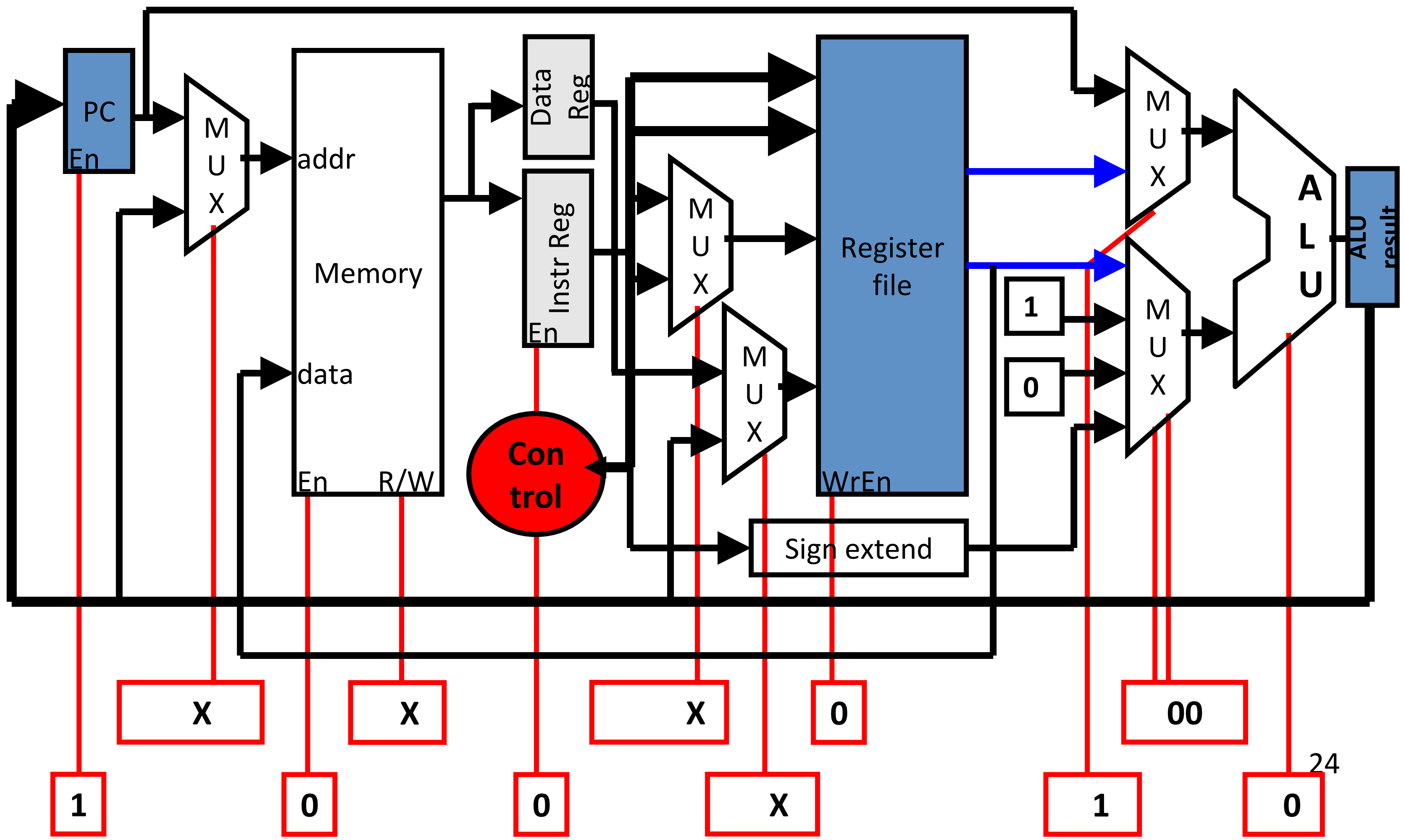
State 2: Add Cycle 3 Operation

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

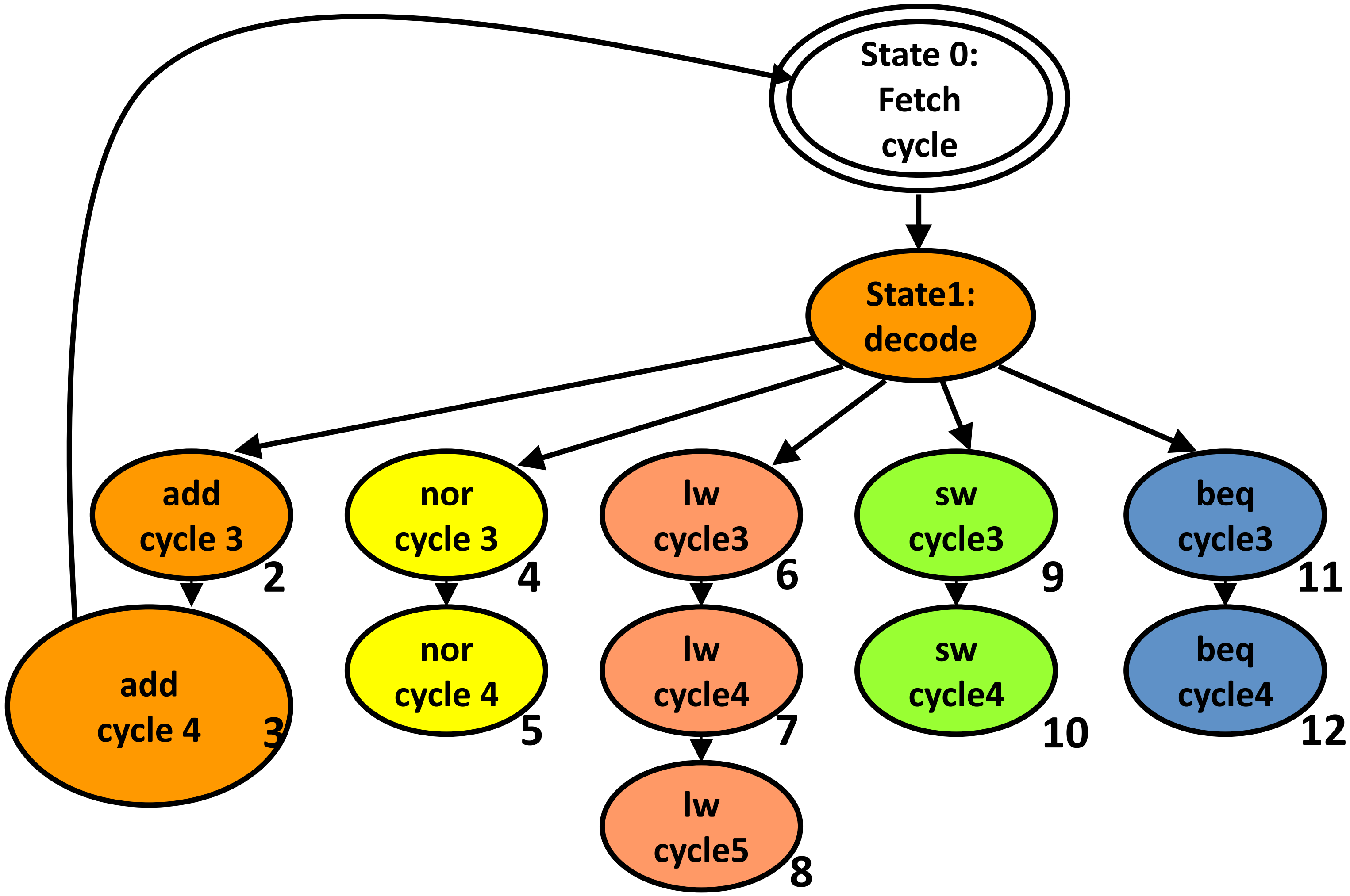


State 2: Add Cycle 3 Operation

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

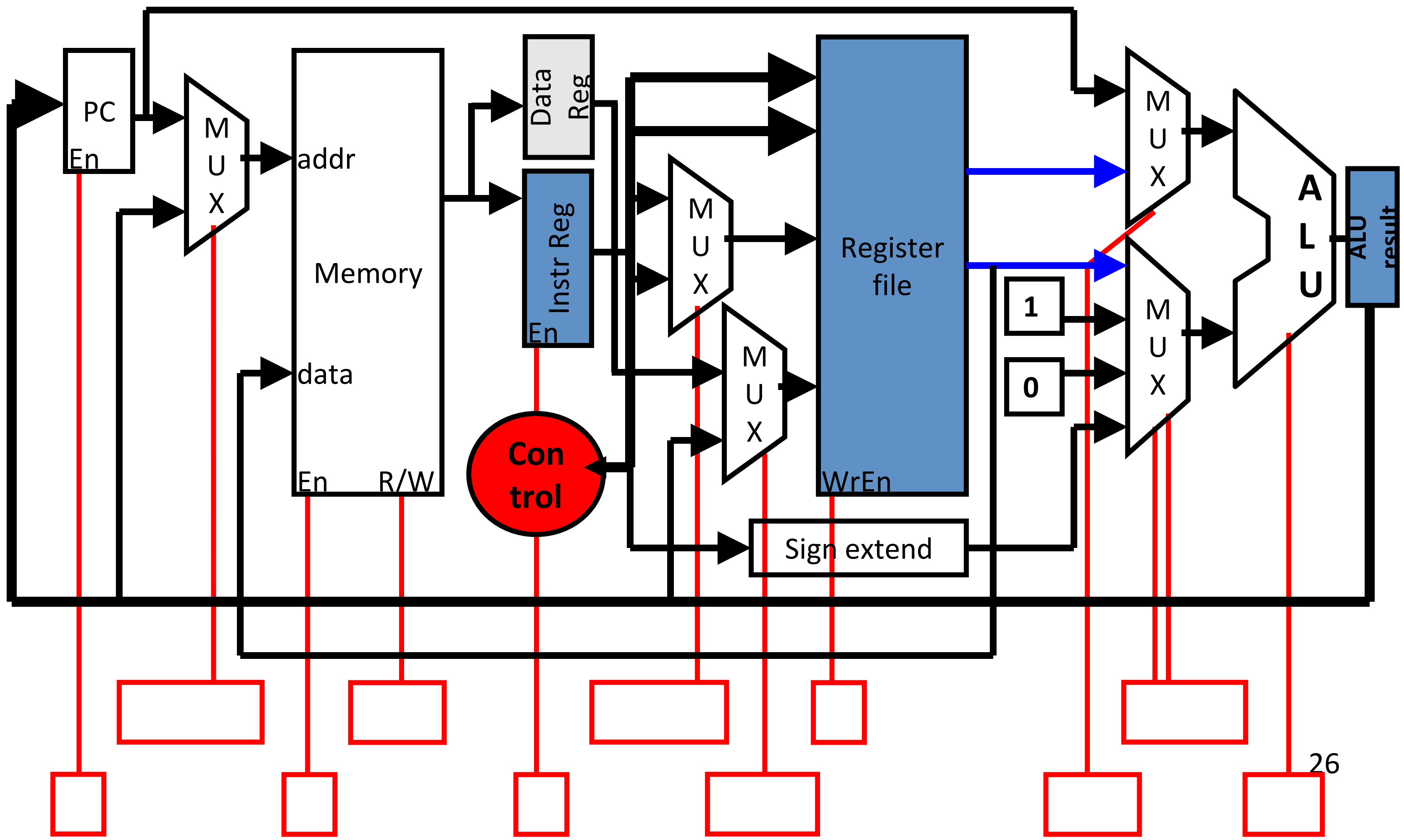


State 3: Add cycle 4



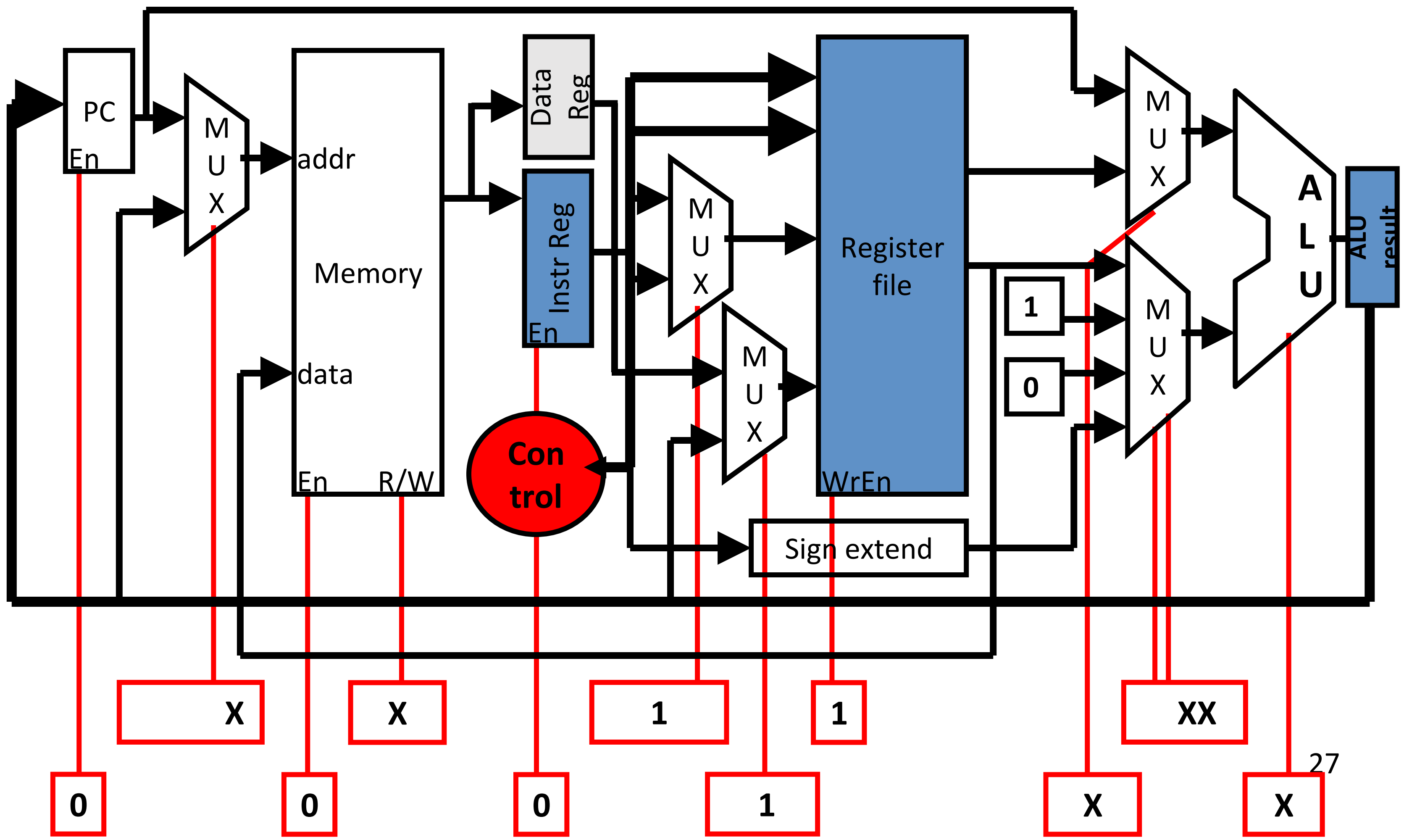
Add Cycle 4 (State 3) Operation

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

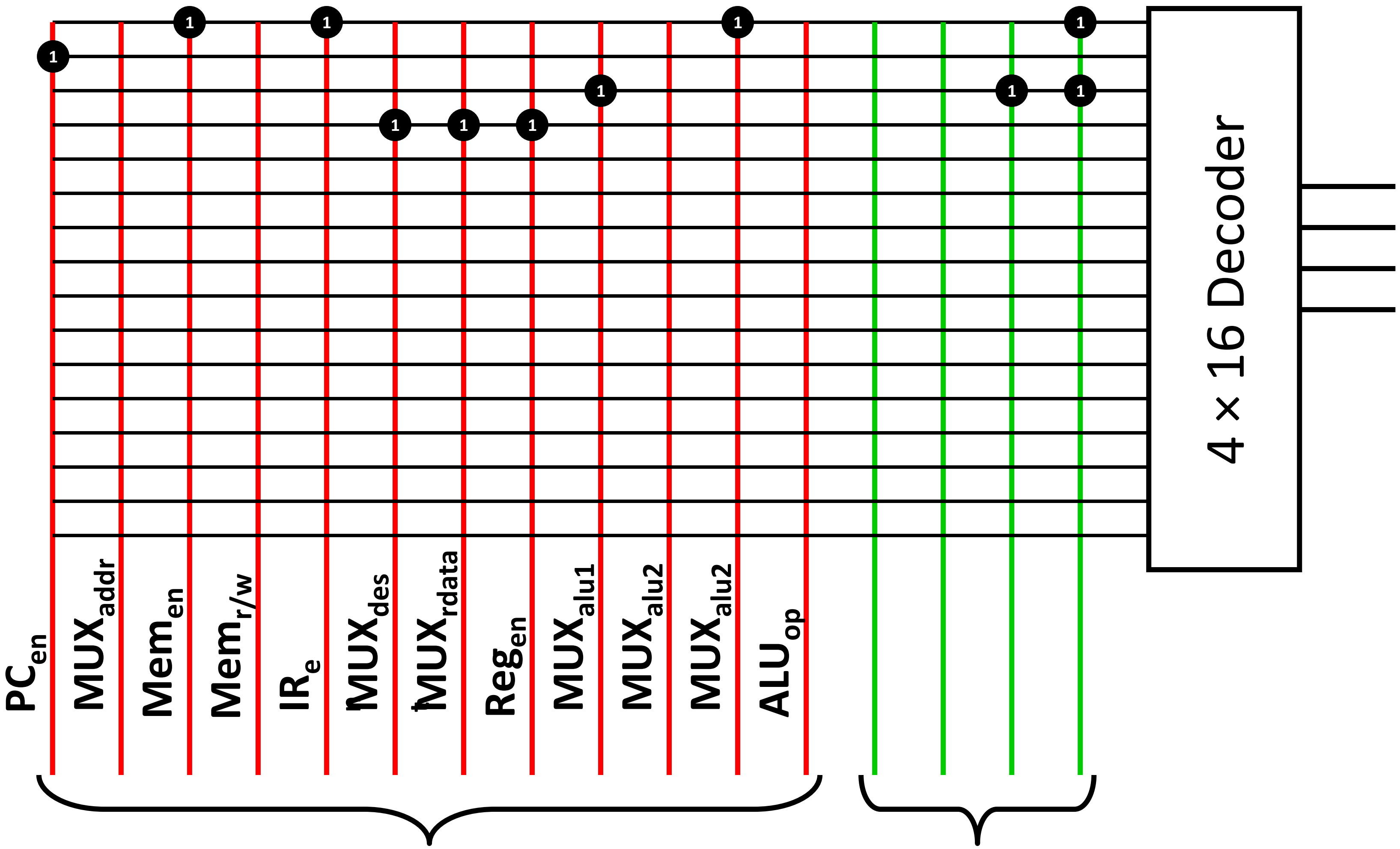


Add Cycle 4 (State 3) Operation

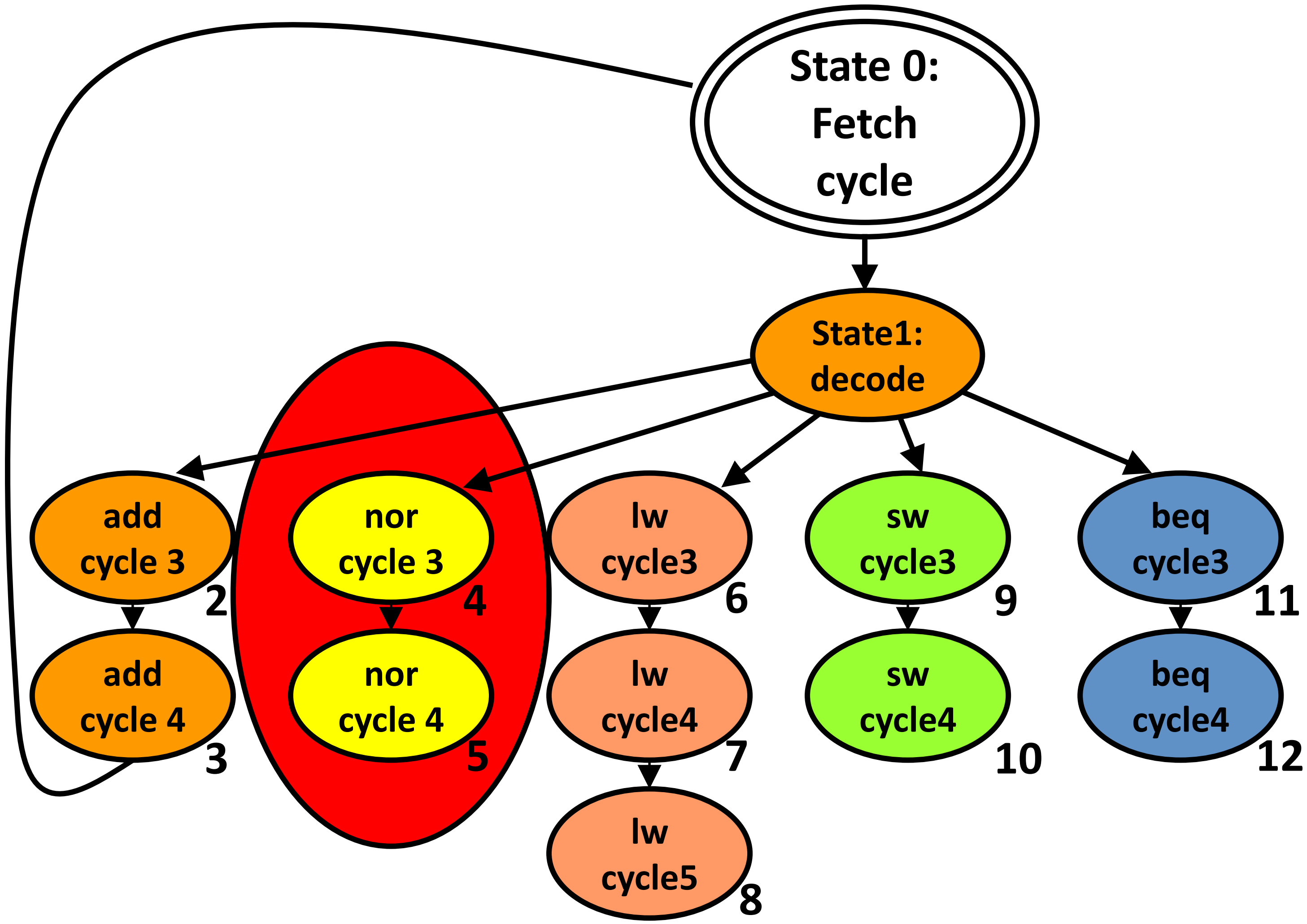
Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



Building the Control Rom

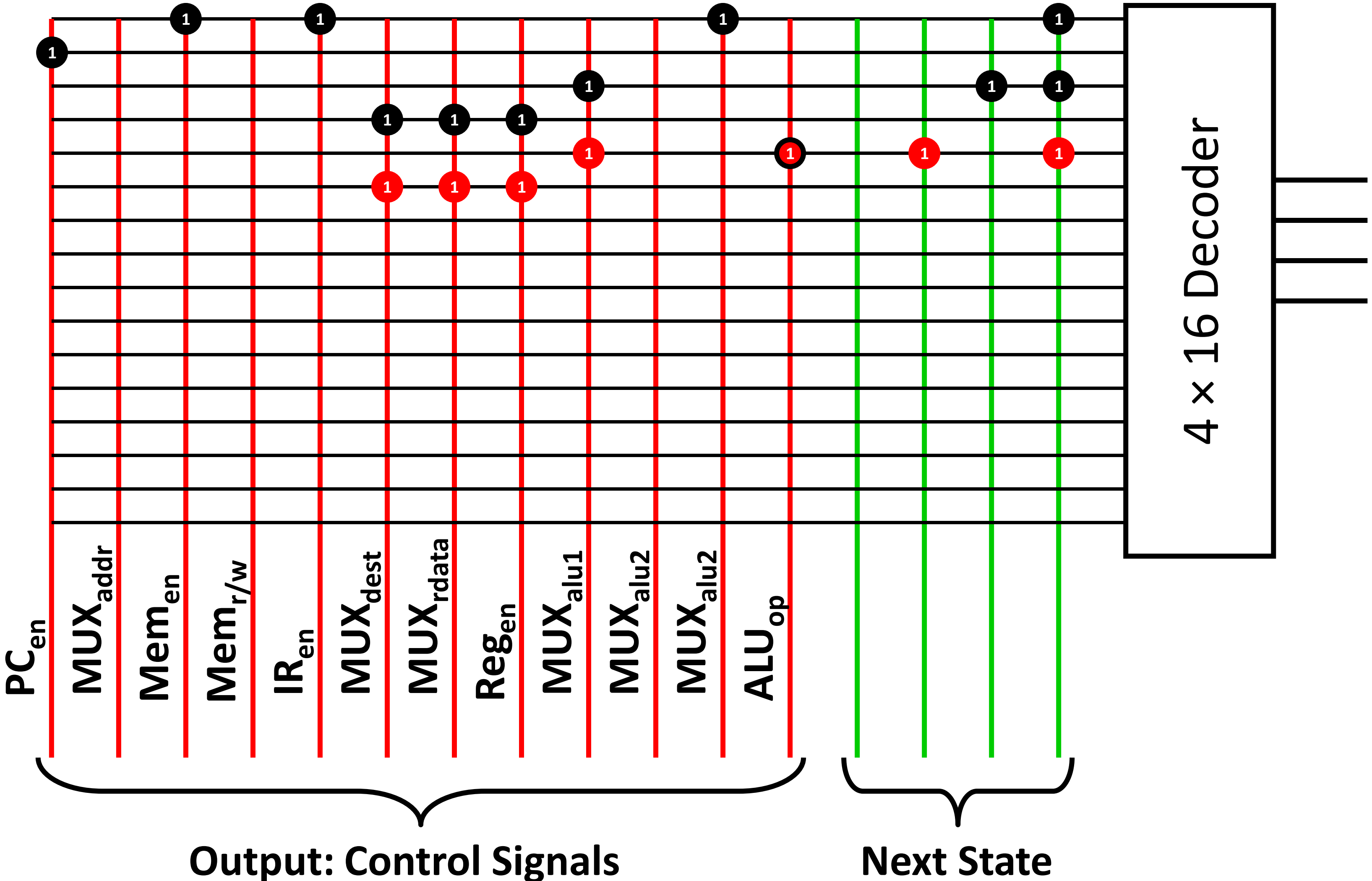


Return to State 0: Fetch cycle to execute the next instruction

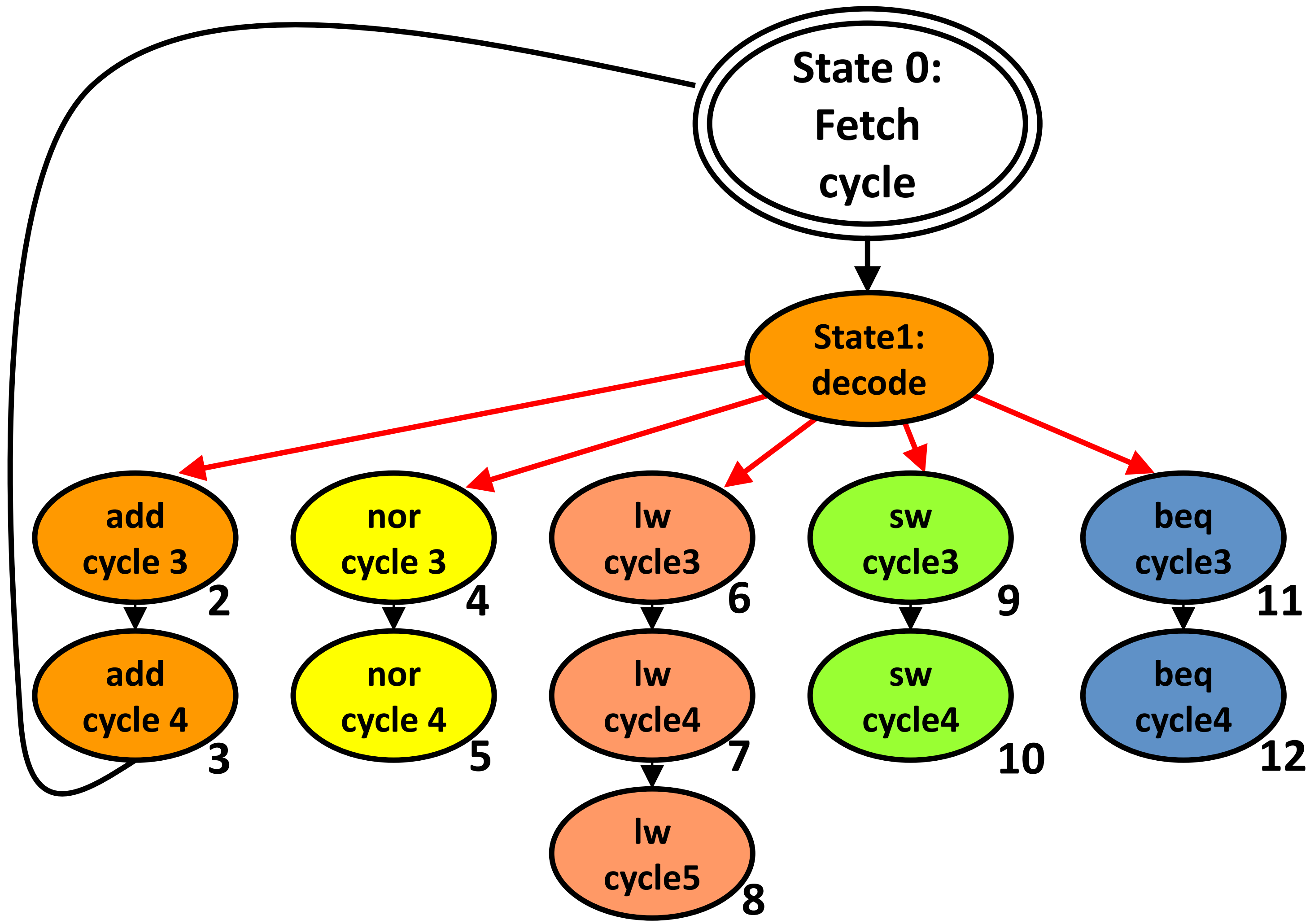


Control Rom for nor (4 and 5)

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

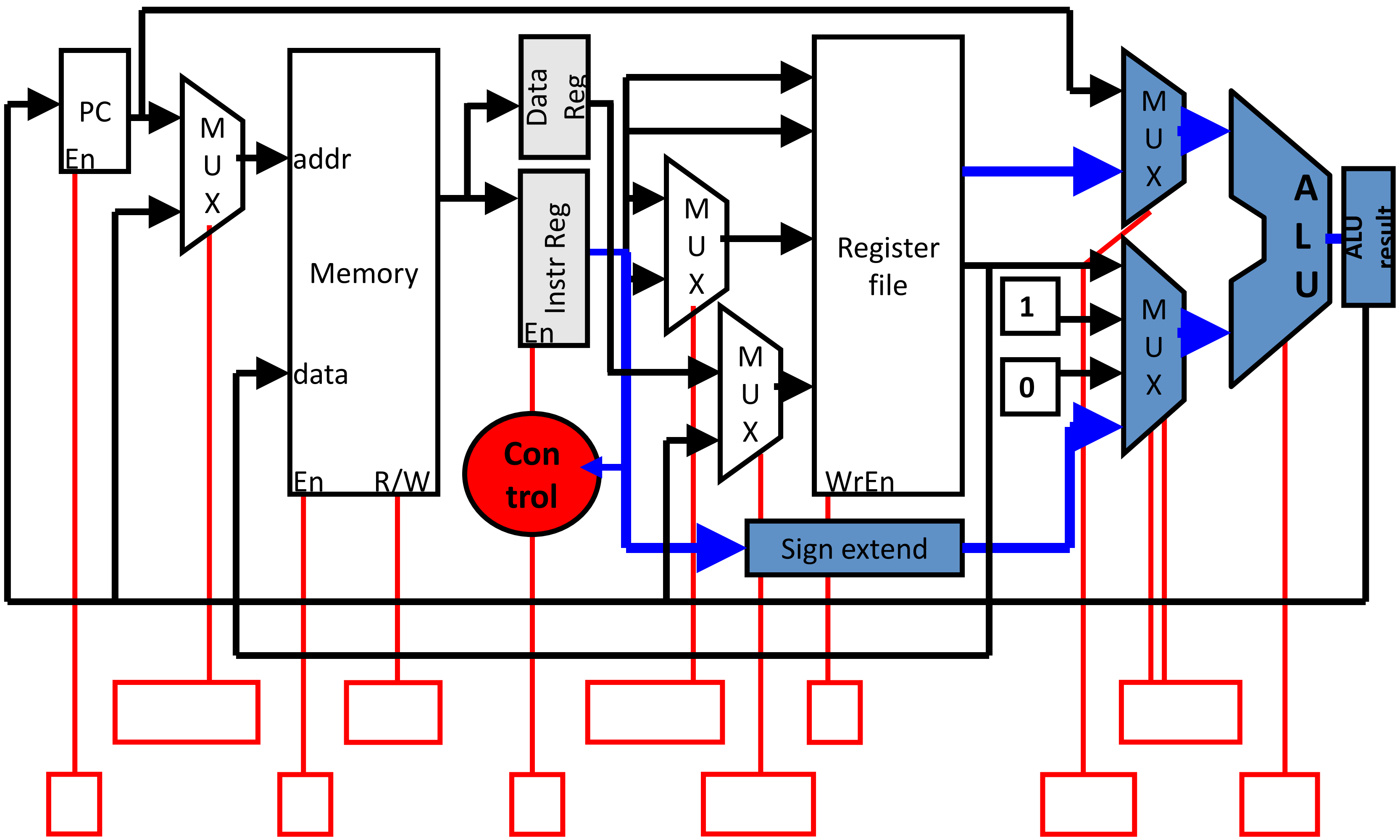


Return to State 0: Fetch cycle to execute the next instruction



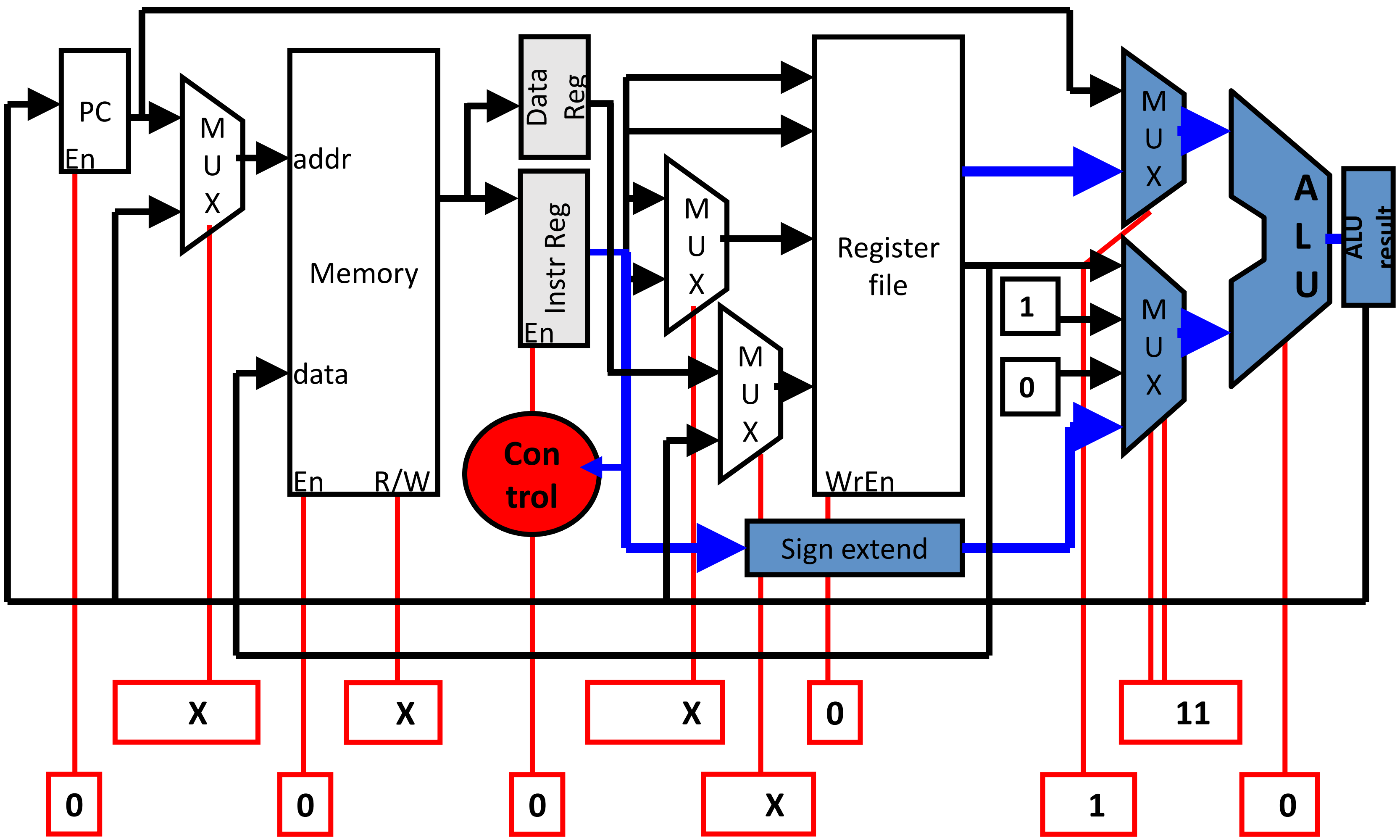
State 6: LW cycle 3

Calculate address for memory reference

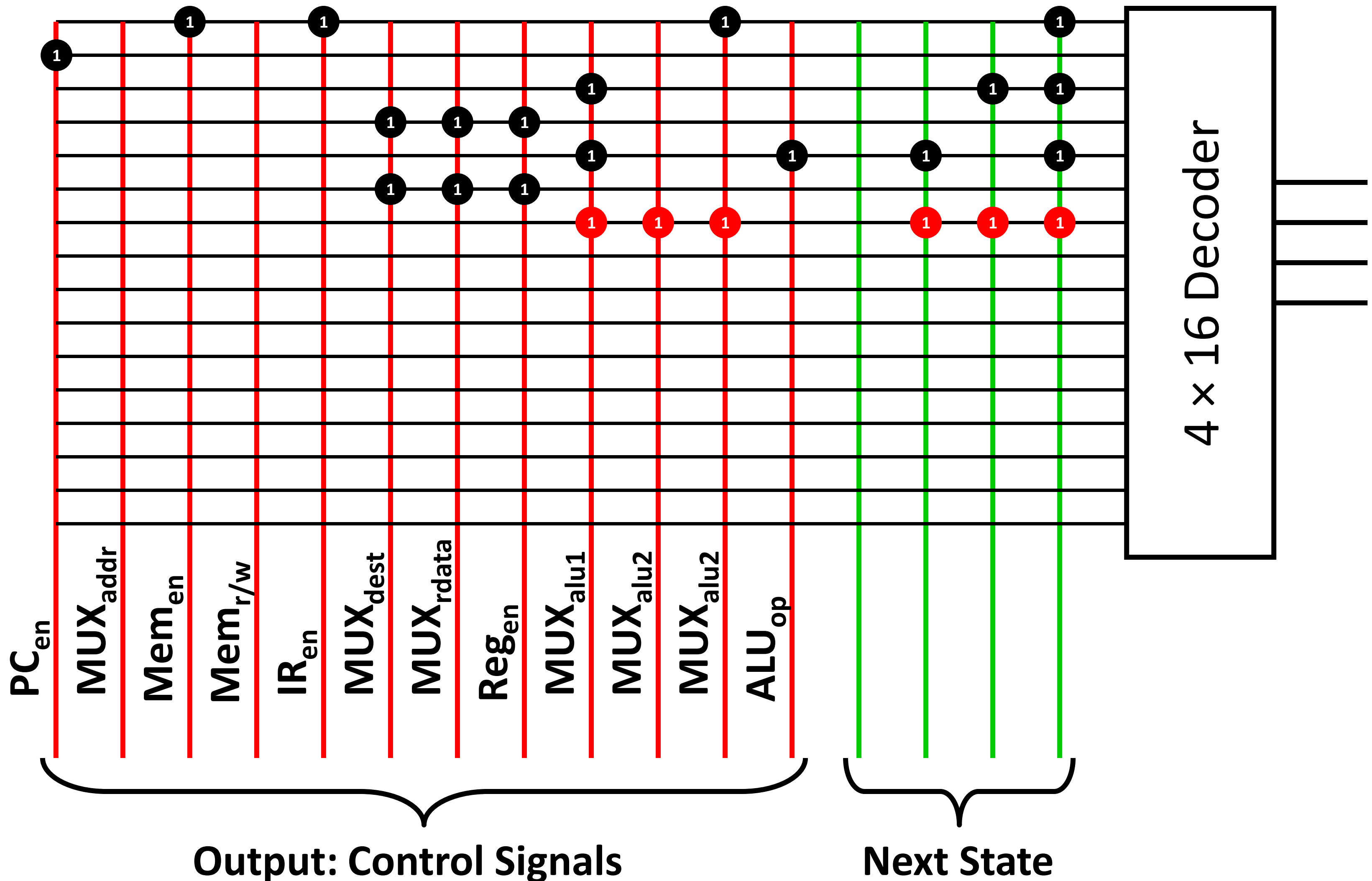


State 6: LW cycle 3

Calculate address for memory reference



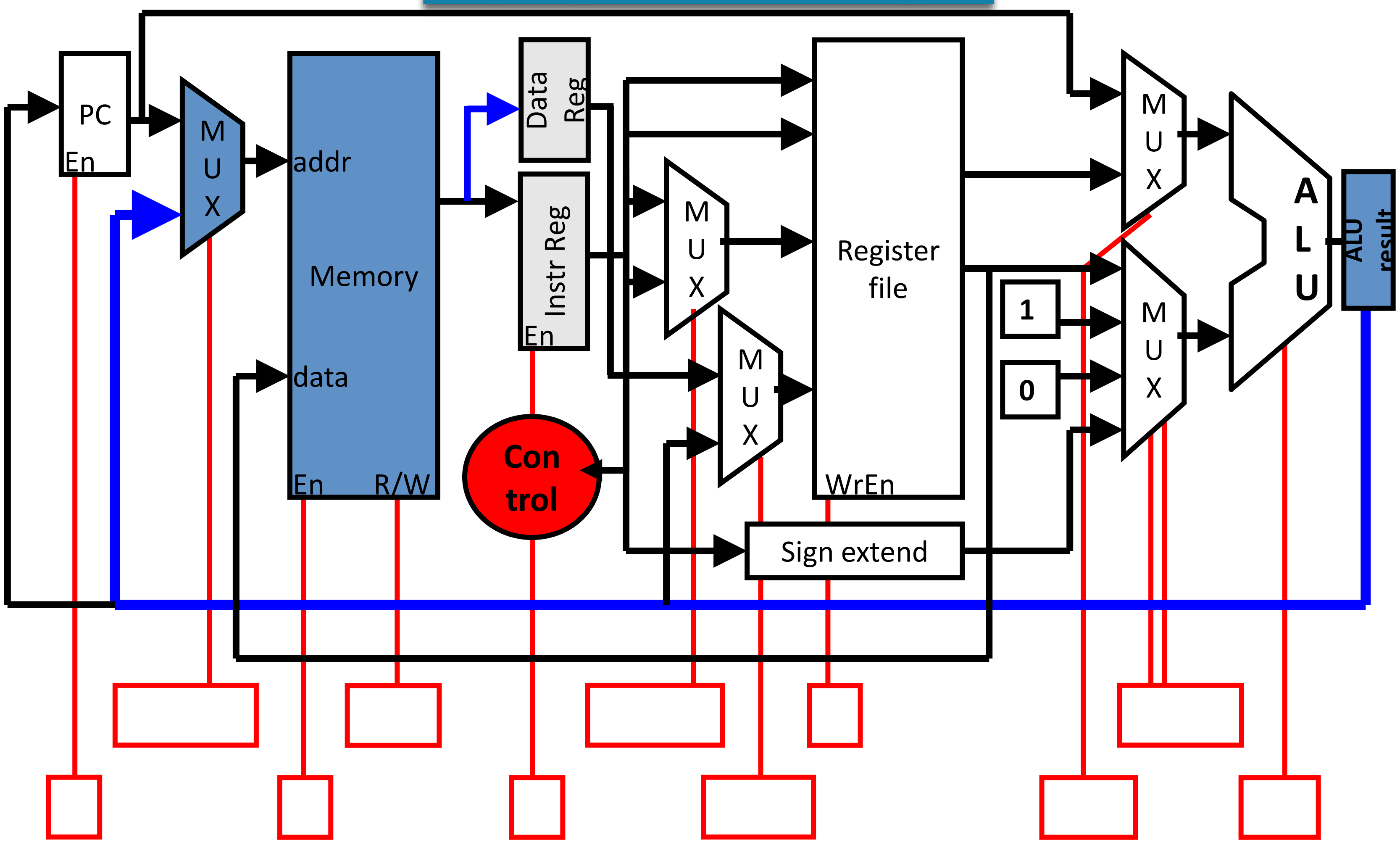
Control Rom (lw cycle 3)



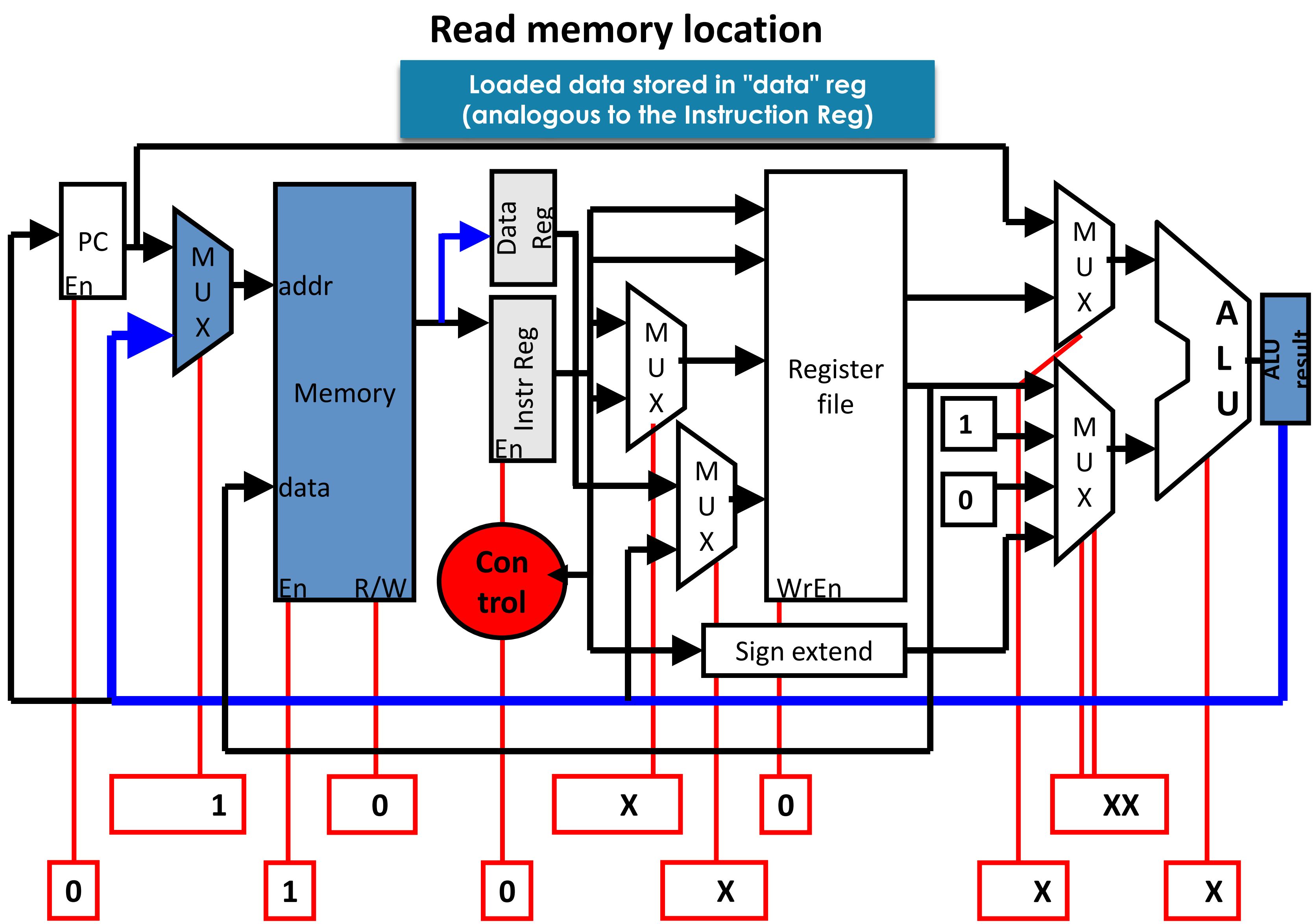
State 7: LW cycle 4

Read memory location

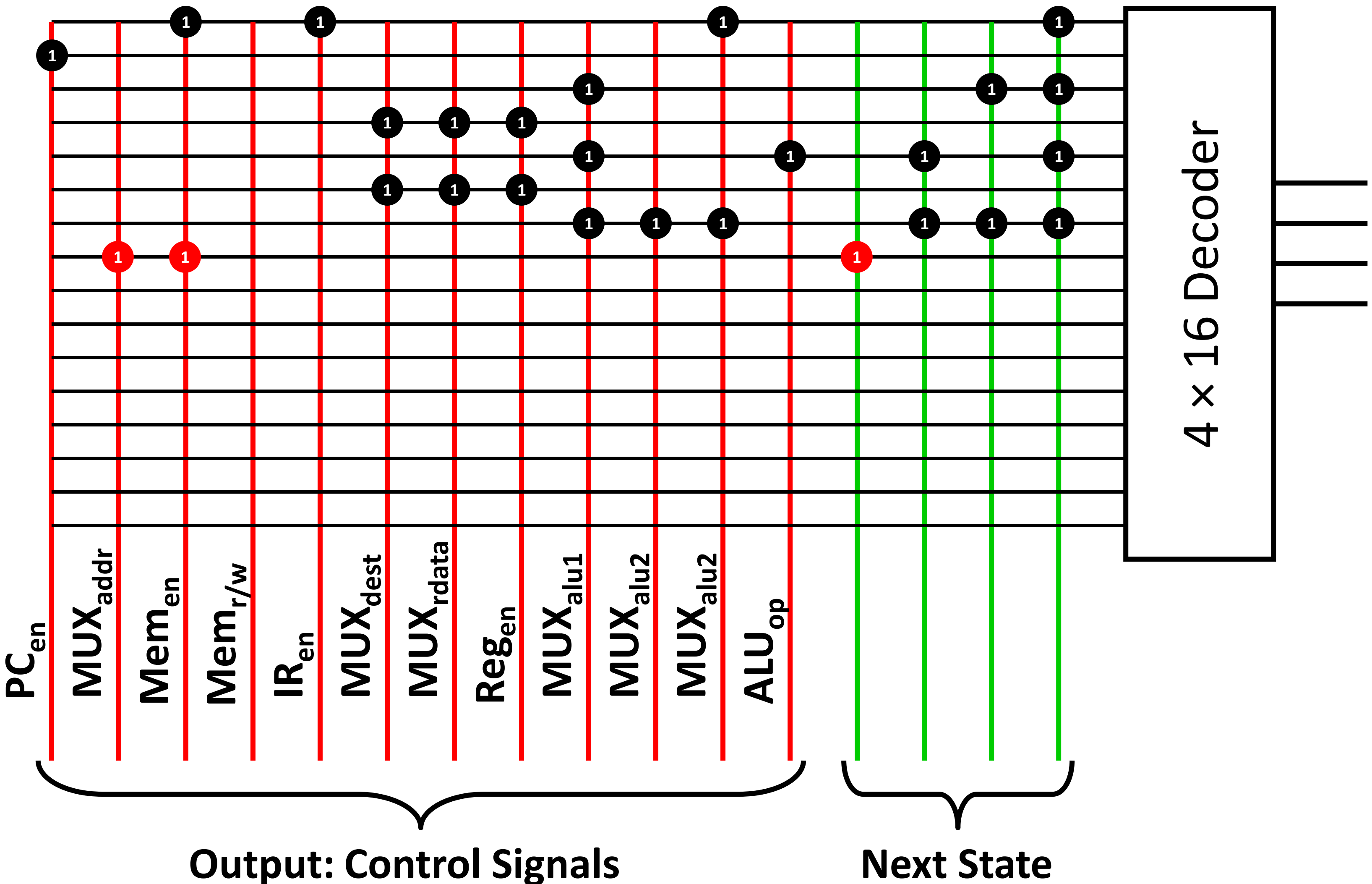
Loaded data stored in "data" reg
(analogous to the Instruction Reg)



State 7: LW cycle 4

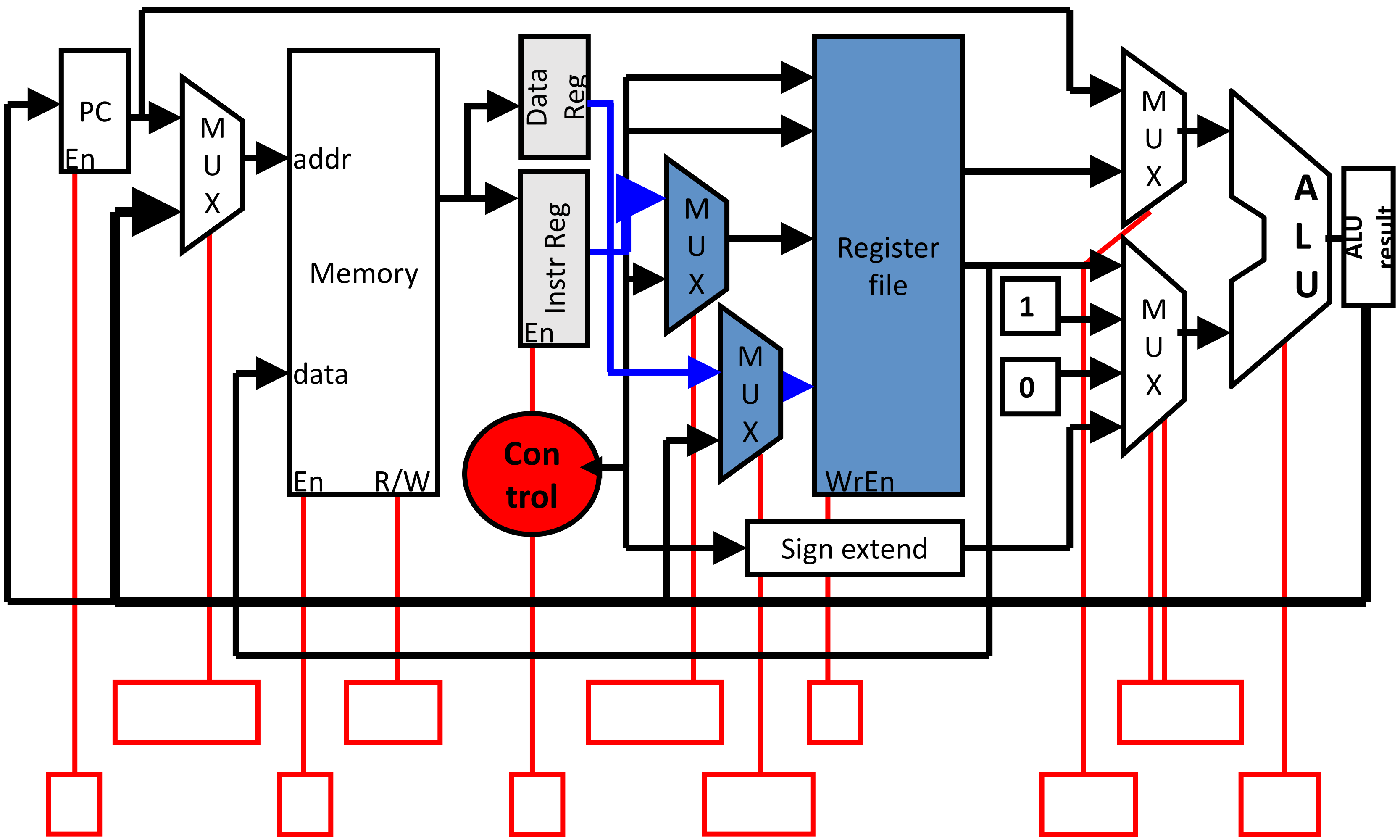


Control Rom (lw cycle 4)



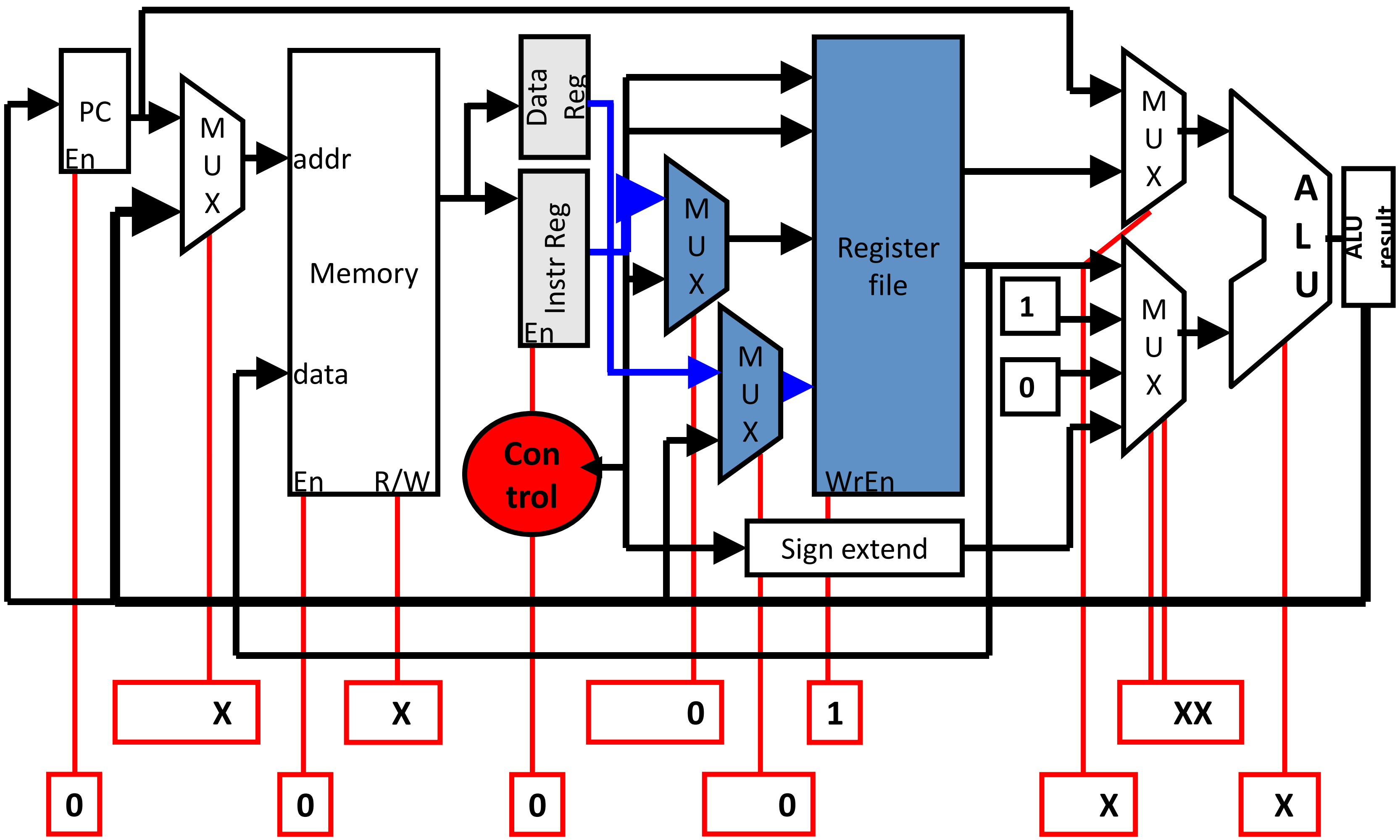
State 8: LW cycle 5

Write memory value to register file

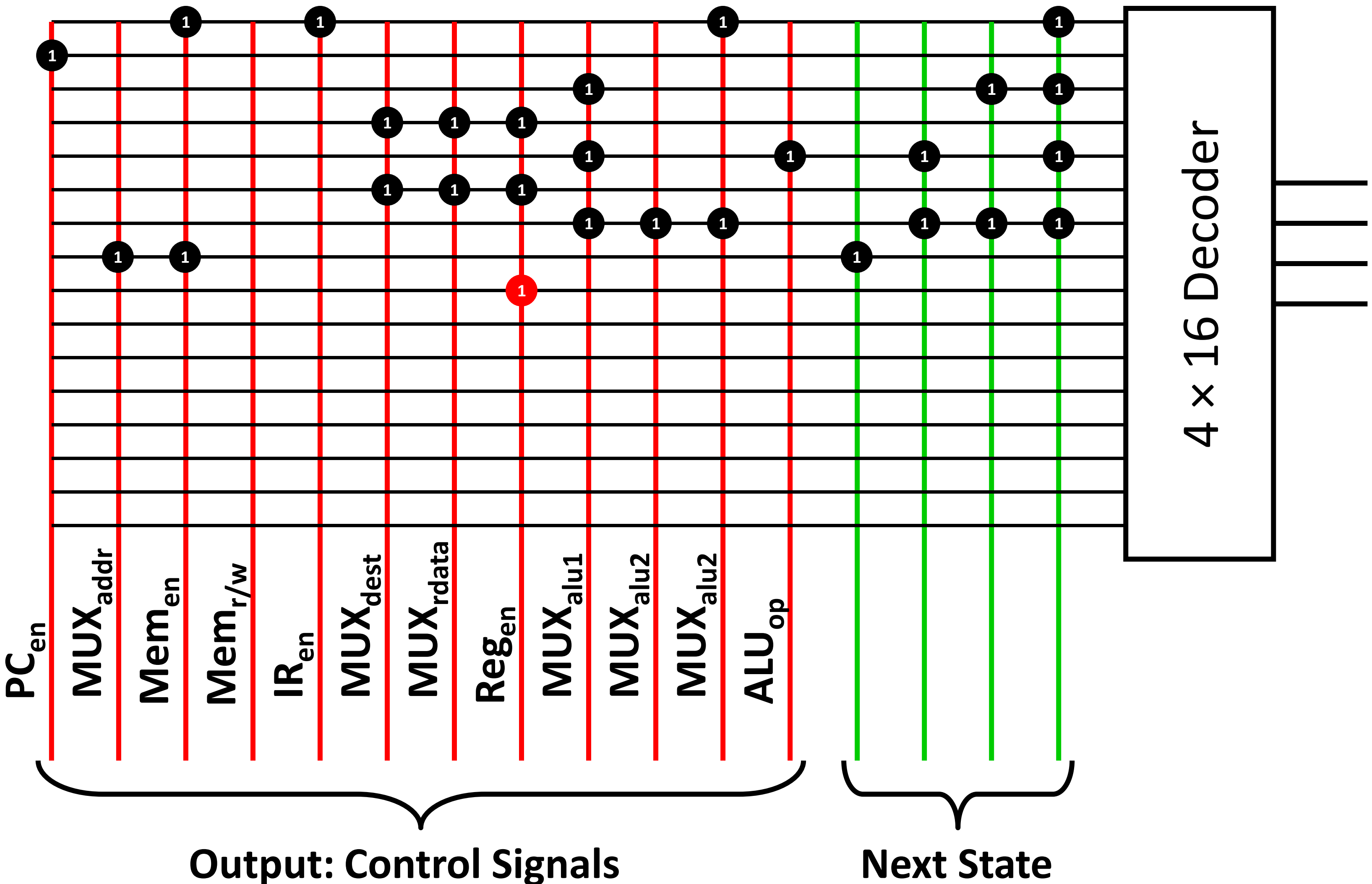


State 8: LW cycle 5

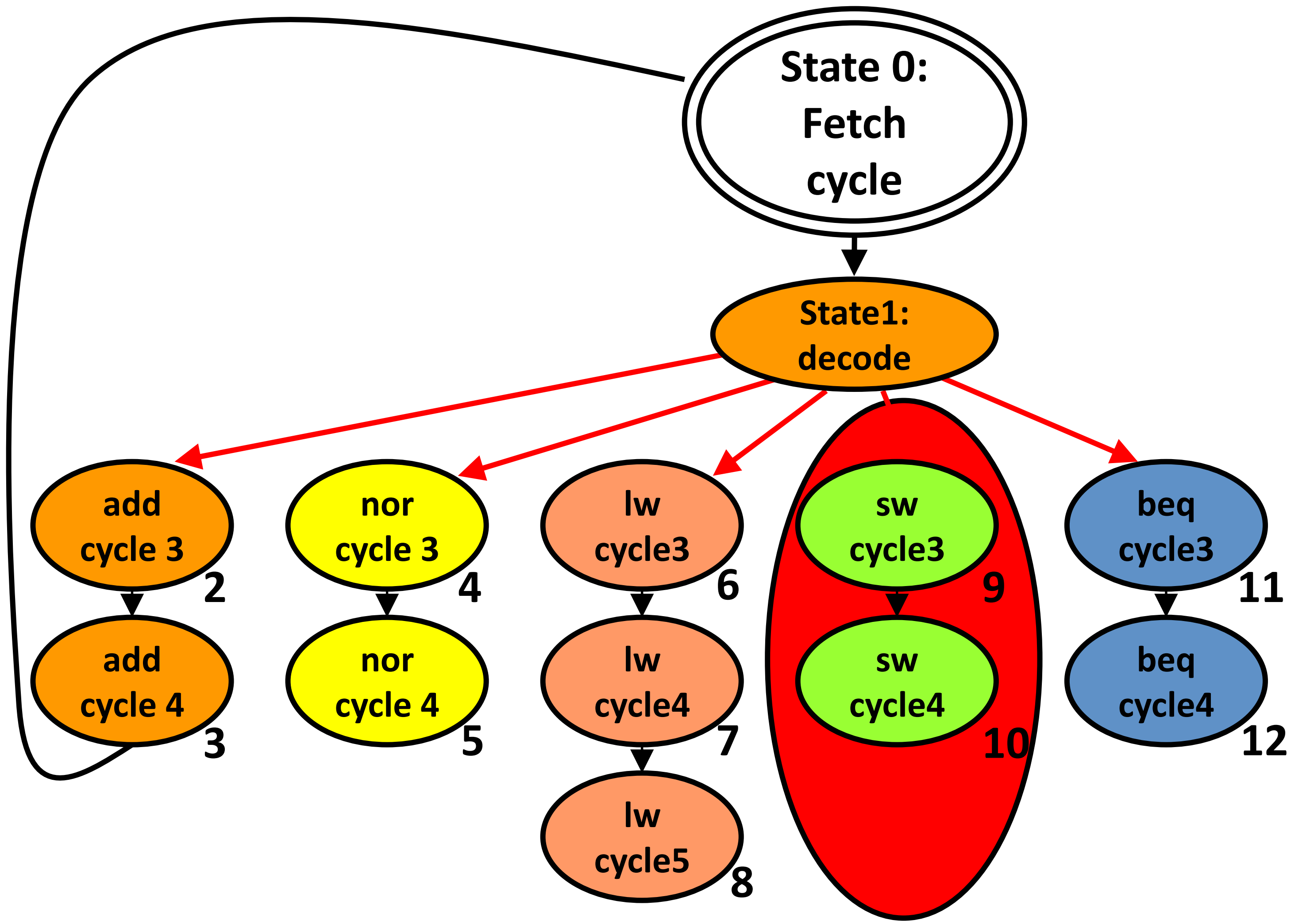
Write memory value to register file



Control Rom (lw cycle 5)

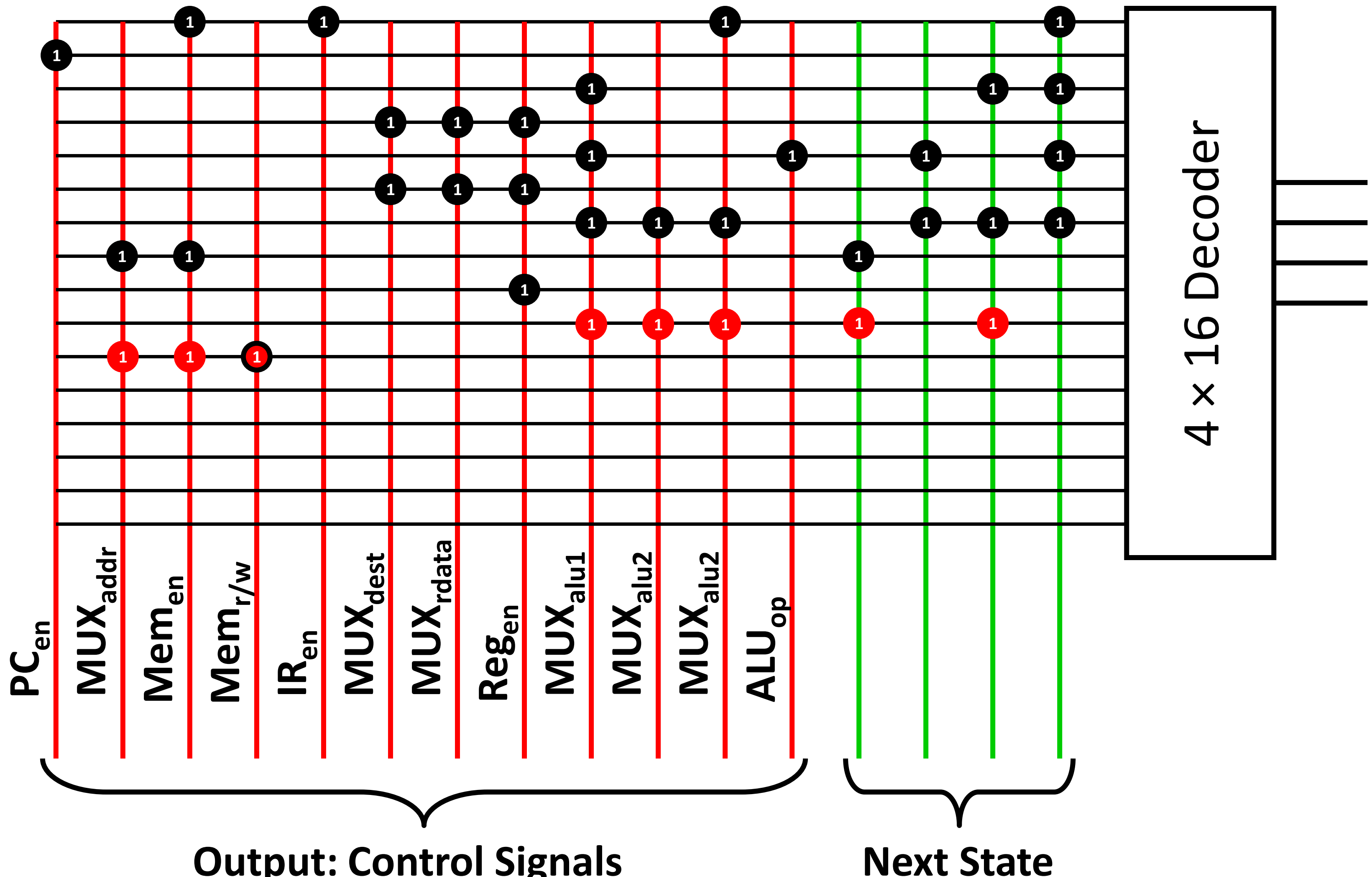


Return to State 0: Fetch cycle to execute the next instruction

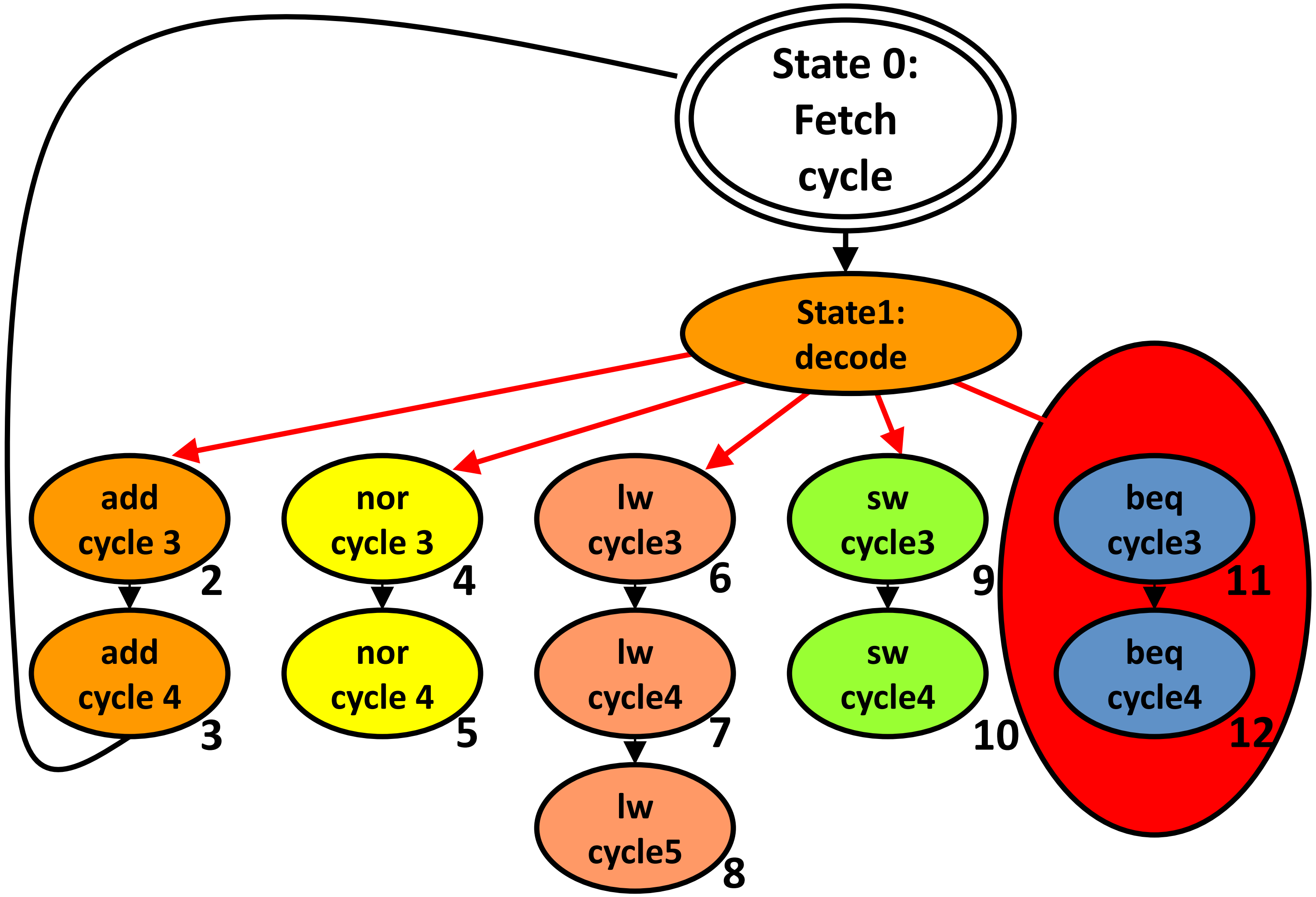


Control Rom (sw cycles 3 and 4)

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

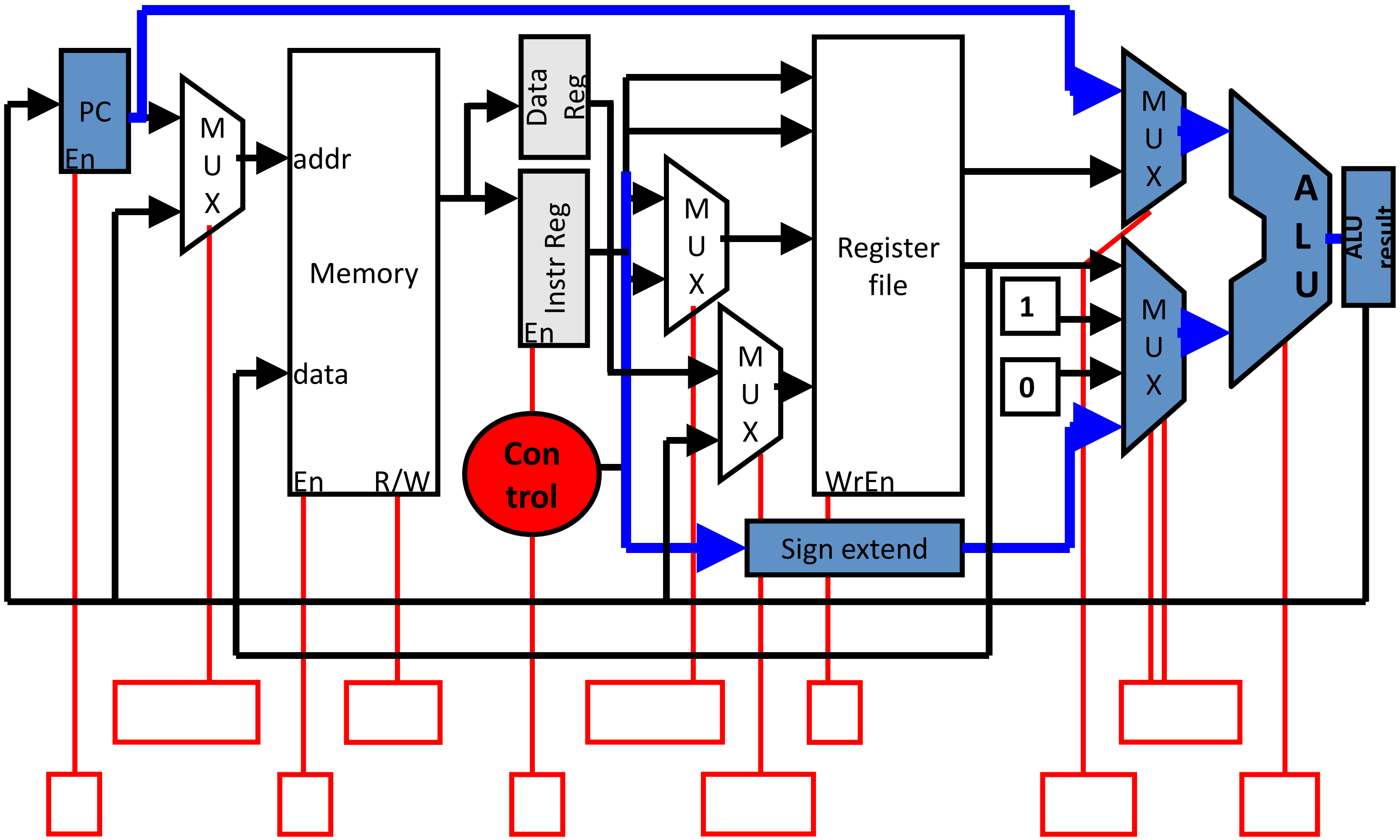


Return to State 0: Fetch cycle to execute the next instruction



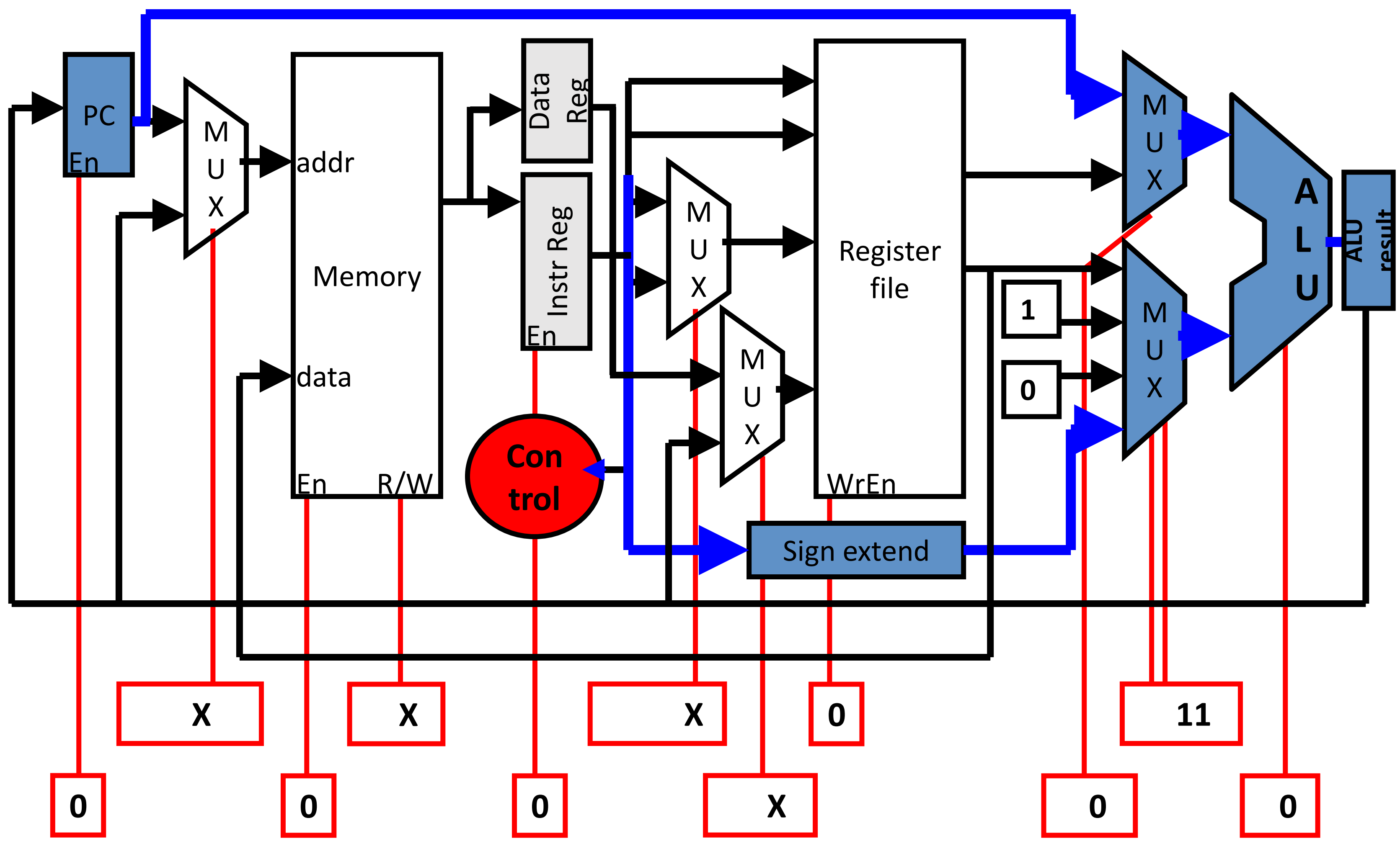
State 11: beq cycle 3

Calculate target address for branch

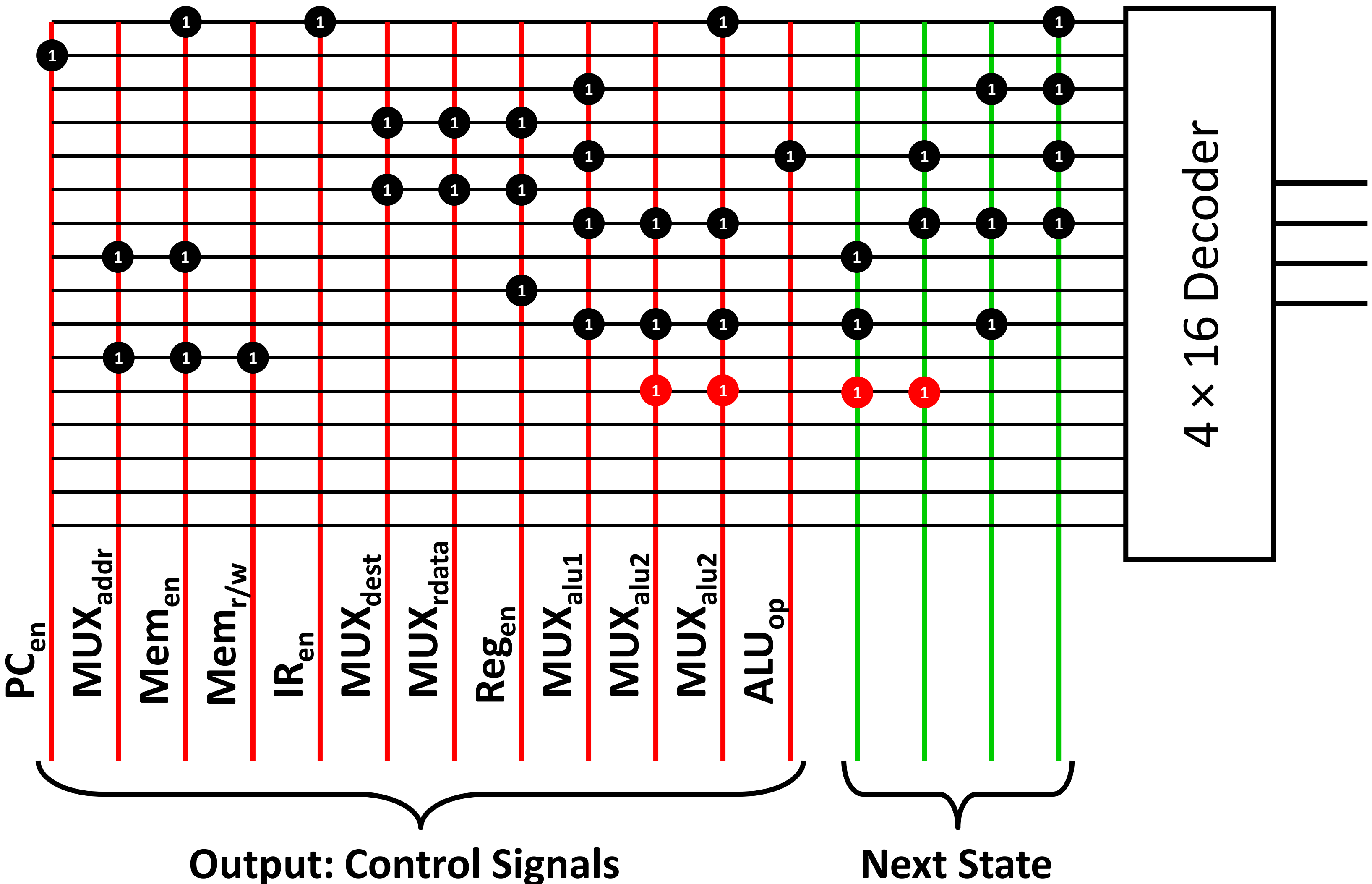


State 11: beq cycle 3

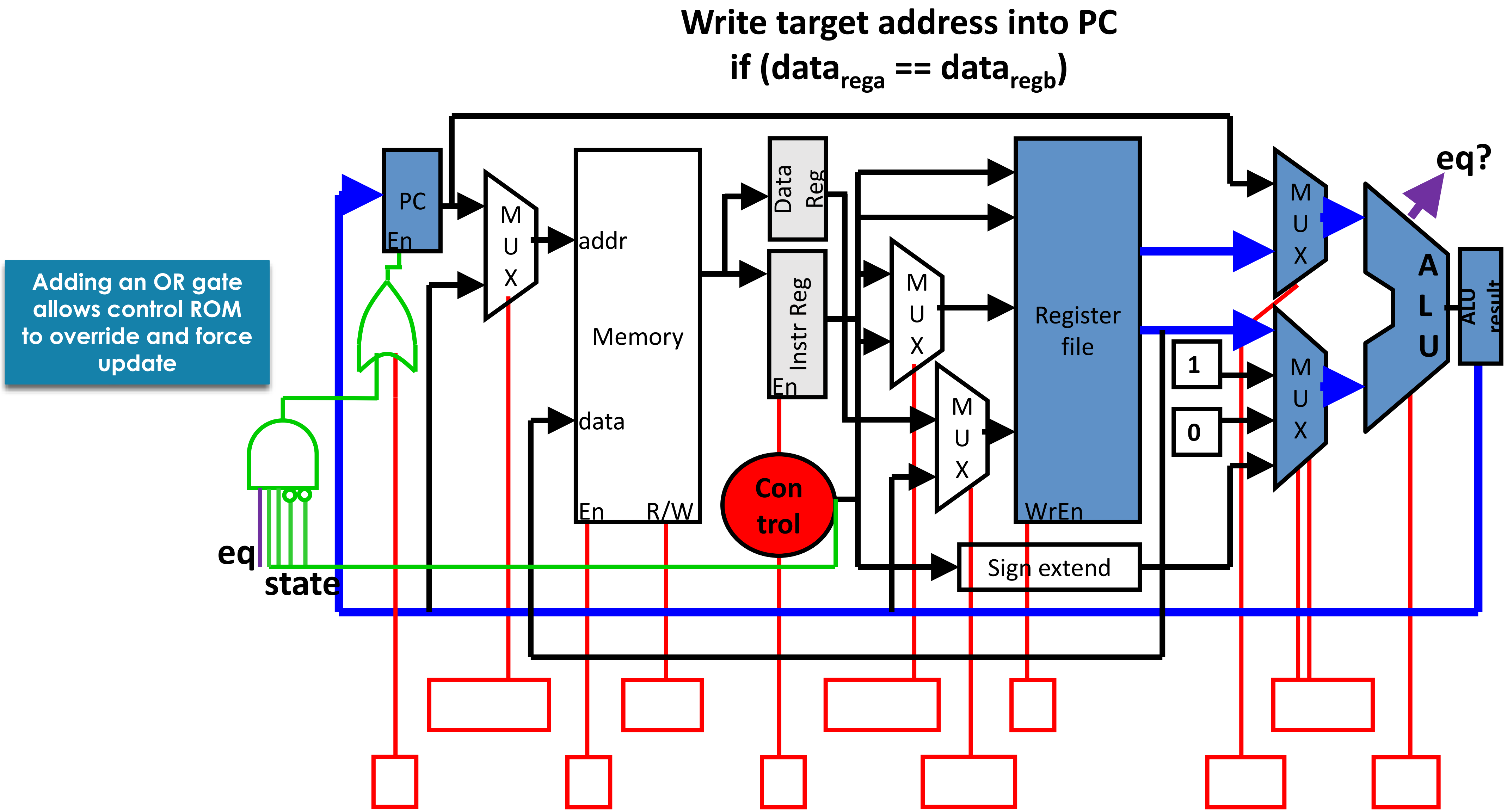
Calculate target address for branch



Control Rom (beq cycle 3)



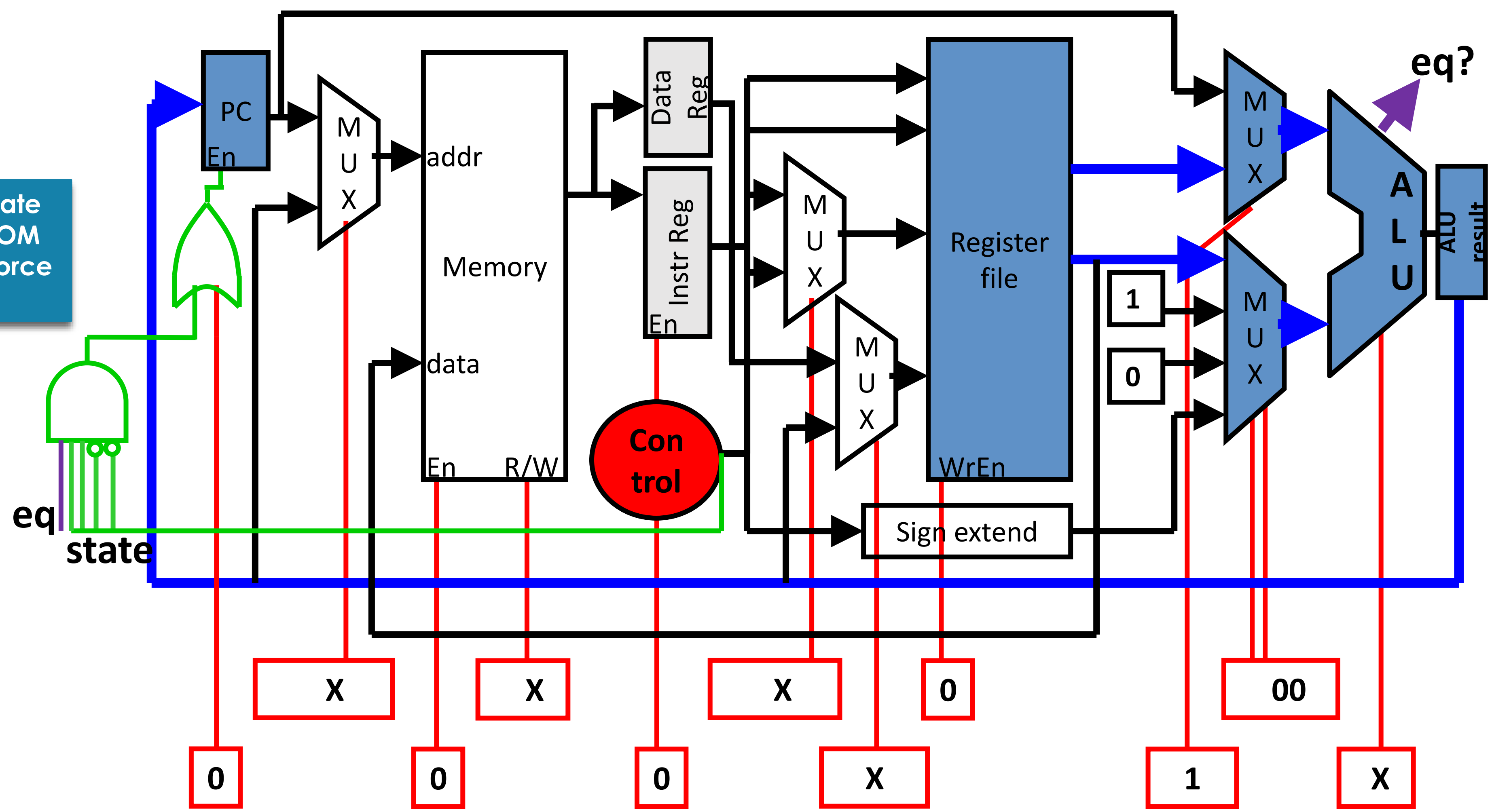
State 12: beq cycle 4



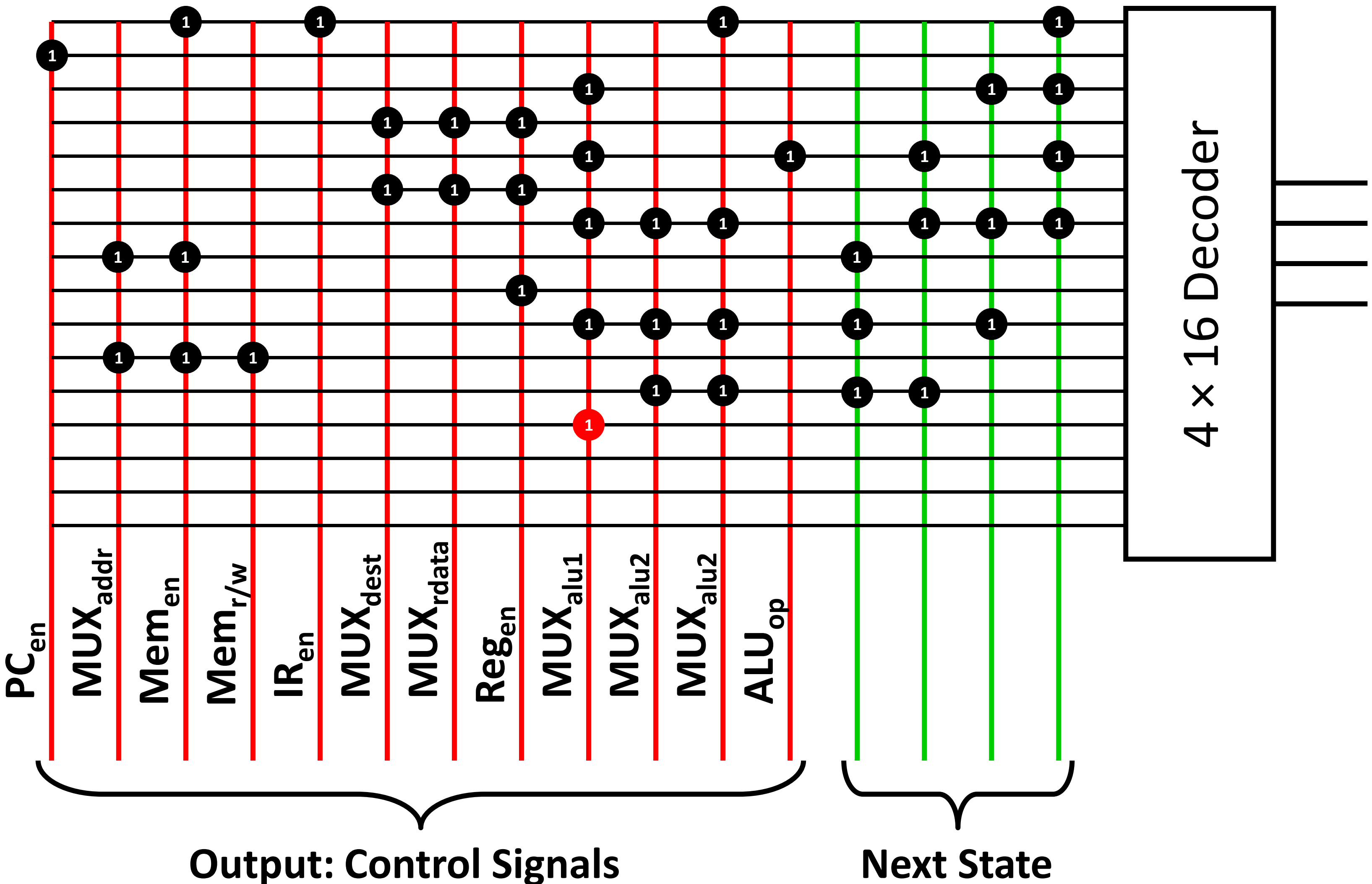
State 12: beq cycle 4

Write target address into PC
if ($data_{rega}$ == $data_{regb}$)

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

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)

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

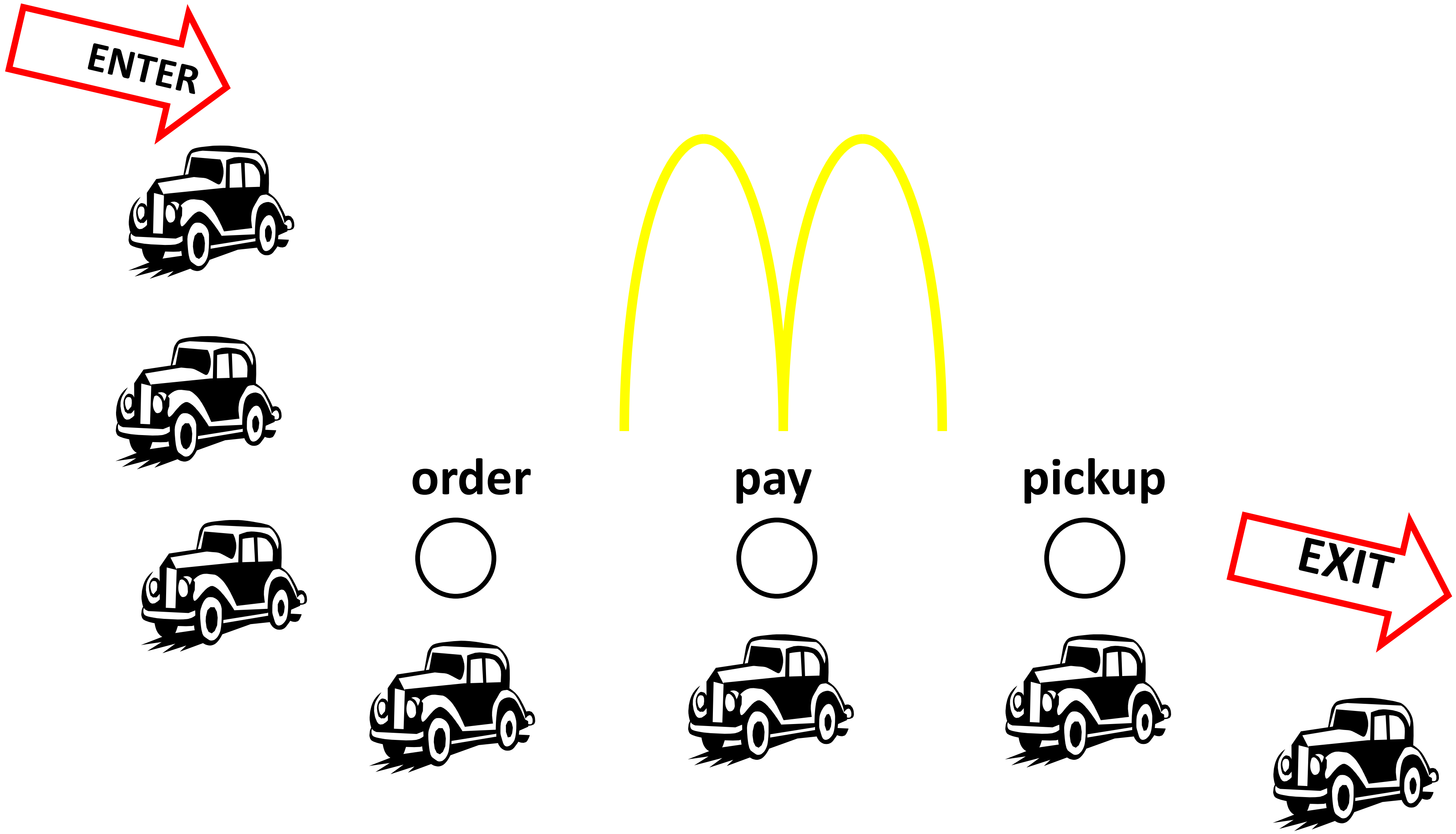
Datapath Summary

- 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

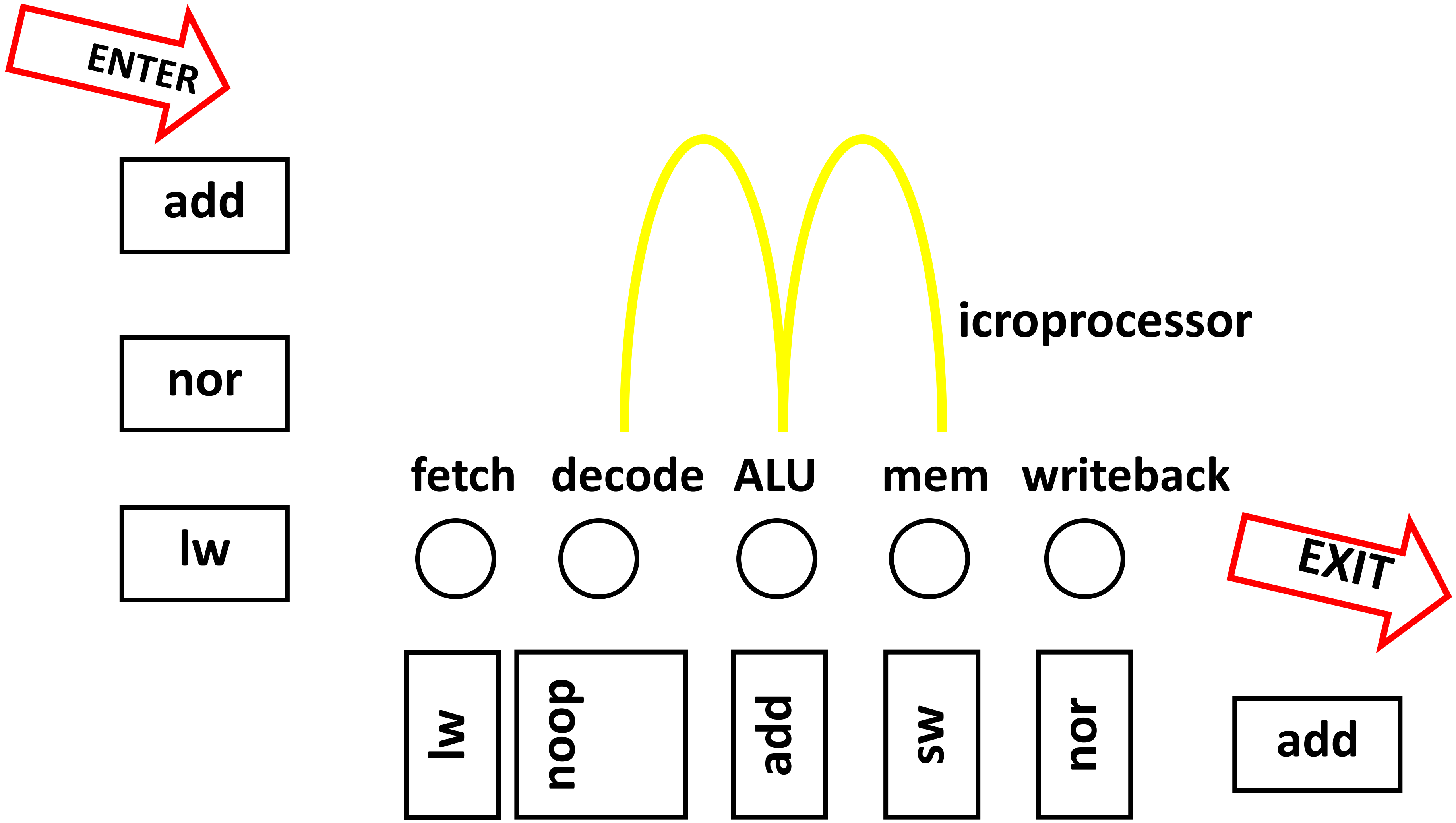
Pipelining

- Want to execute an instruction?
  - Build a processor (multi-cycle)
  - Find instructions
  - Line up instructions (1, 2, 3, …)
  - Overlap execution
    - Cycle #1: Fetch 1
    - Cycle #2: Decode 1 Fetch 2
    - Cycle #3: ALU 1 Decode 2 Fetch 3
    - . . . . . .
  - This is called pipelining instruction execution.
  - Used extensively for the first time on IBM 360 (1960s).
  - CPI approaches 1.

Pipelining



Pipelining


ENTER
add
nor
lw
icroprocessor
fetch
decode
ALU
mem
writeback
EXIT
lw
noop
add
sw
nor
add

Next time

- Exam Review