11. Foundations of Processor Design: Multi-cycle Processors

EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

What’s on the schedule?

- P2a due Thursday 2/16
  - If you didn’t get full points on P1a, you’ll need help. See Piazza post @1247
- P2l due Thursday 2/23
  - This is the harder one!
- HW3 due 2/20

Last time…

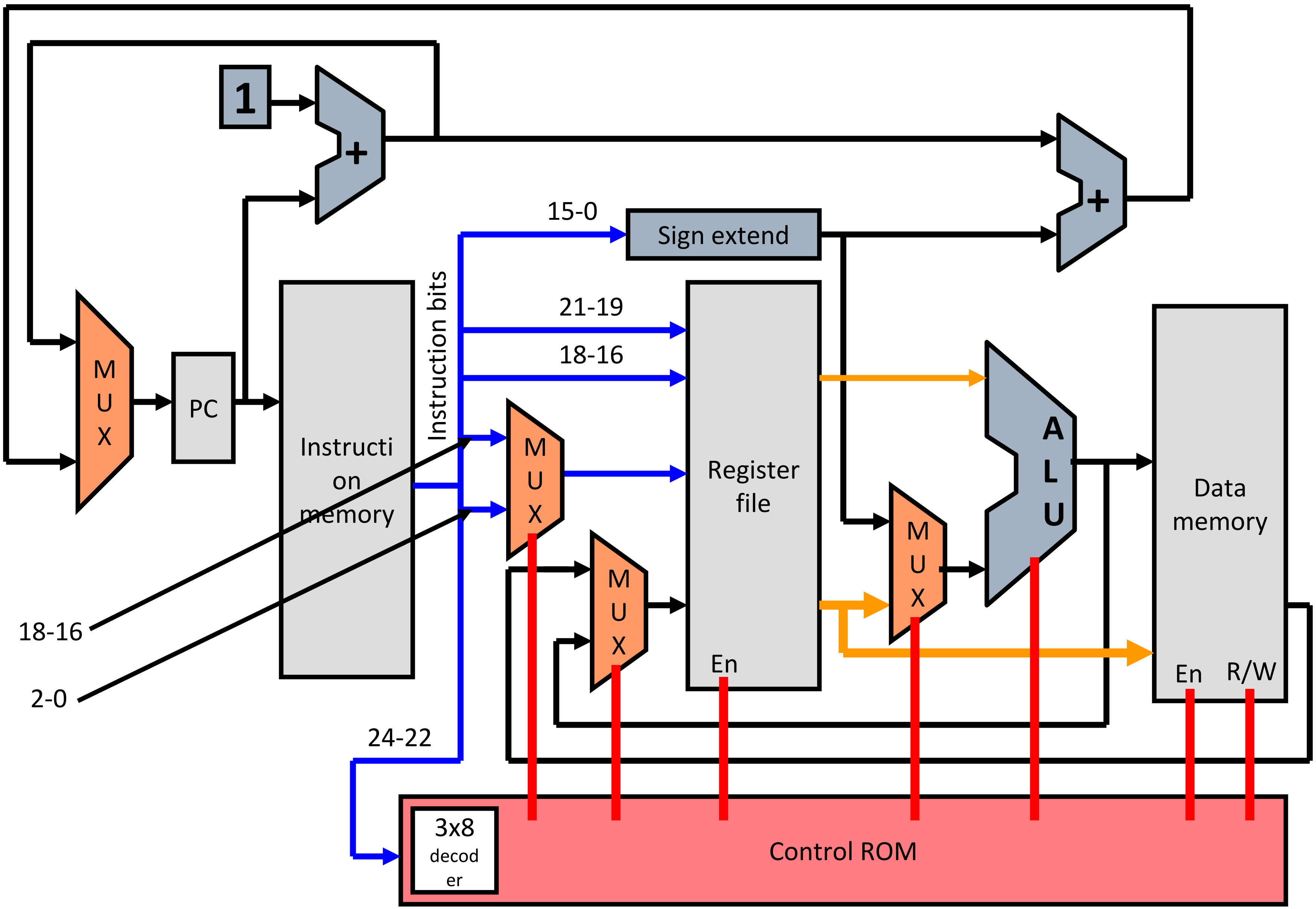
- We discussed the single-cycle processor.

- We walked through the various elements of the datapath
  - MUXes, adders, ALUs, memories, register files, and more.

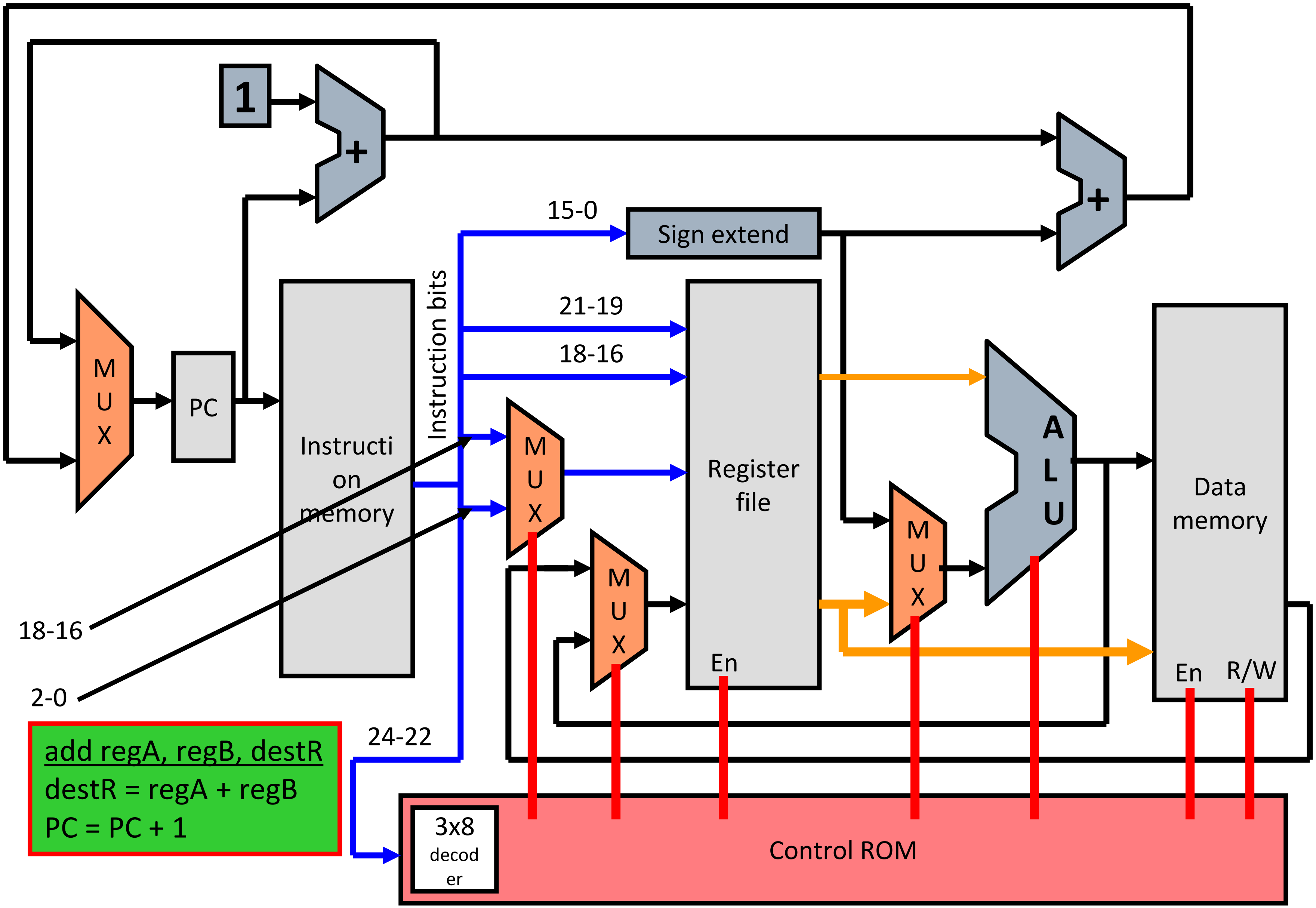
- We looked at the way we'd implement the add, nor, beq, lw, and sw.

- Today we'll review the single-cycle processor and start on an improved version: the multi-cycle processor.
  - First, let's look back at the single-cycle datapath and the add instruction.

LC2K Datapath Implementation

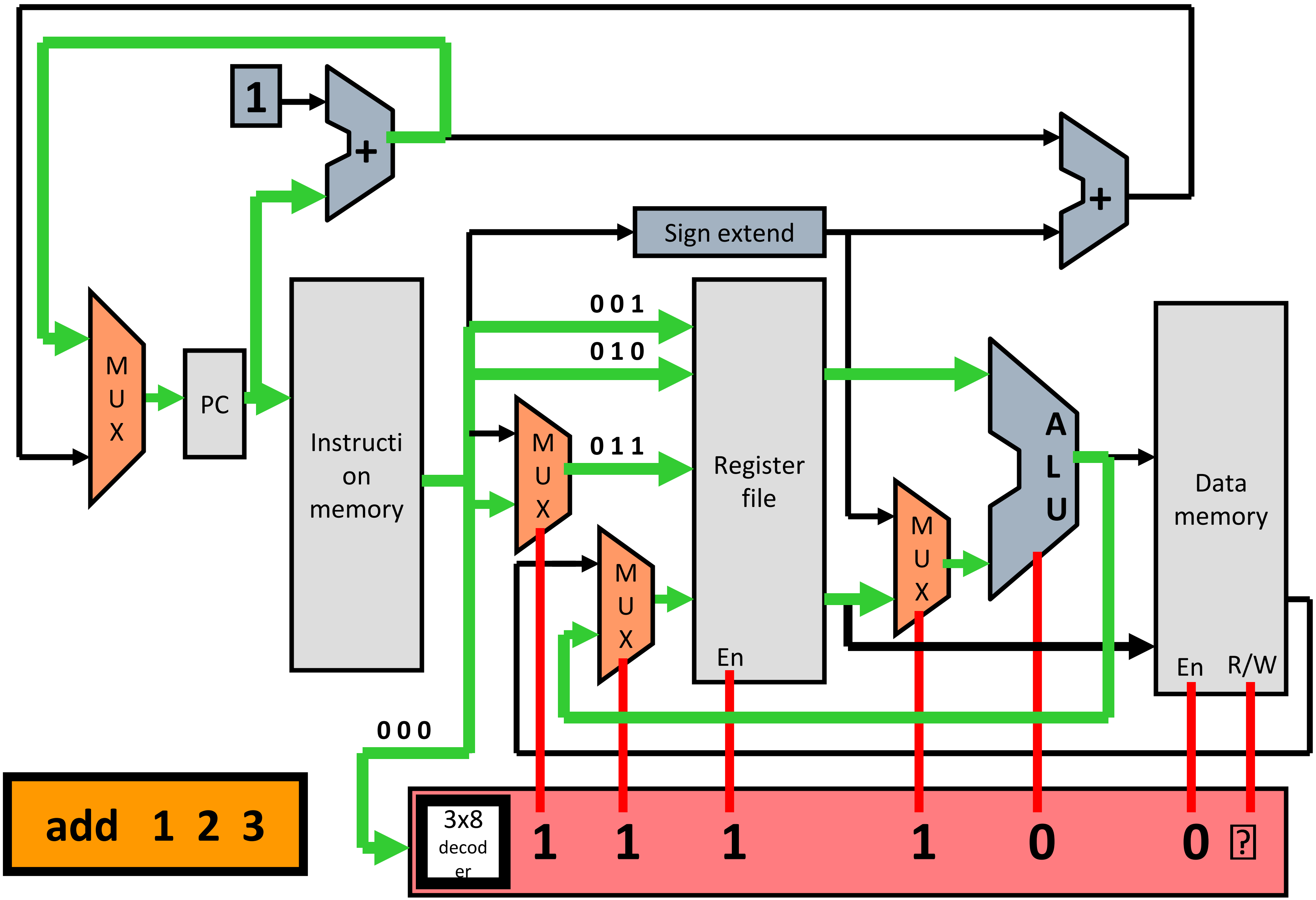


Executing an ADD Instruction



add regA, regB, destR
destR = regA + regB
PC = PC + 1

Executing an ADD Instruction on LC2Kx Datapath



add 1 2 3

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

	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:

??

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:

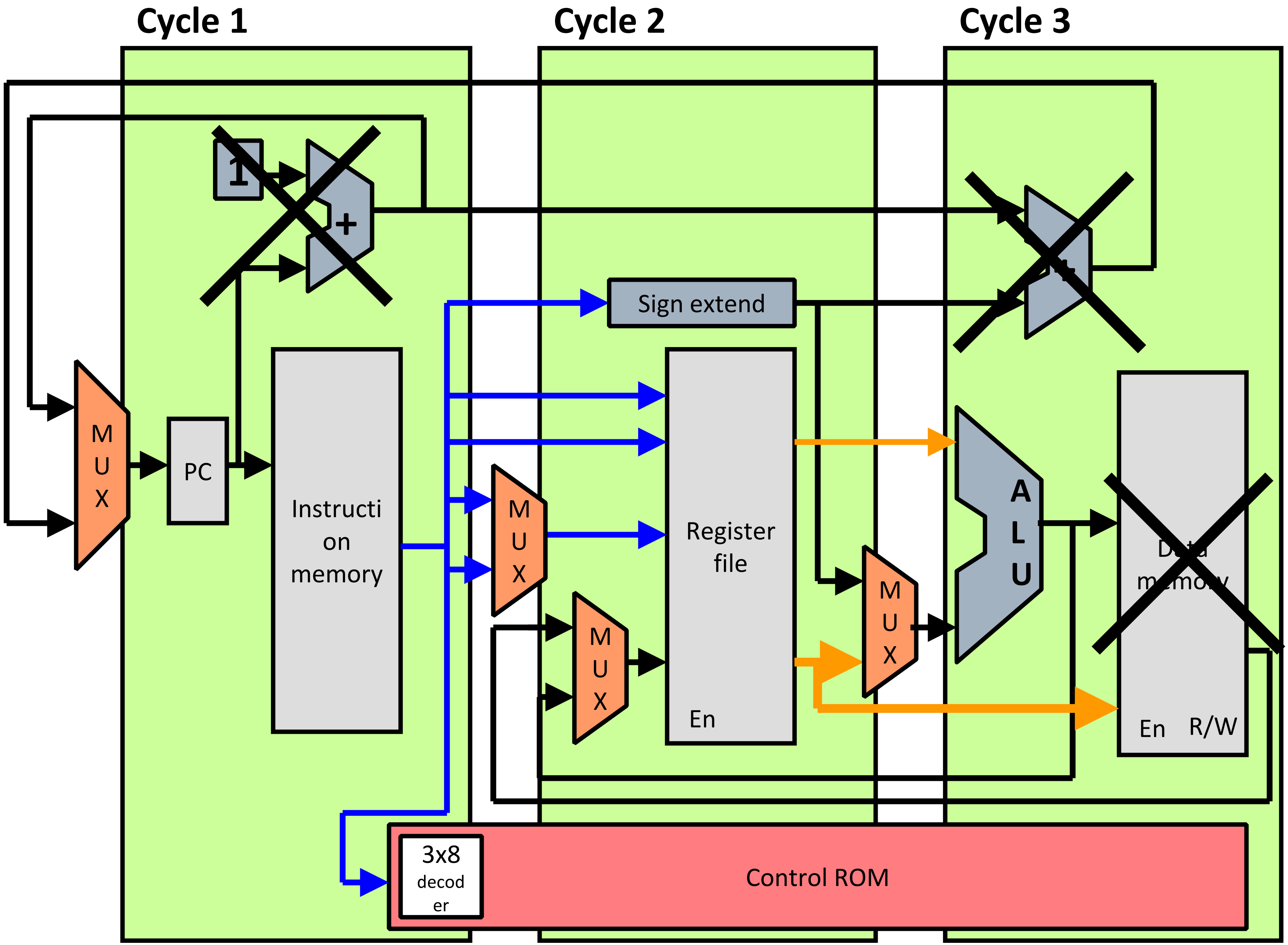
100 * 8ns = **800** ns

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
  - 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.



Cycle 1

Cycle 2

Cycle 3

1

+

MUX

PC

Instruction memory

Sign extend

Register file

En

MUX

MUX

+

ALU

MUX

Data memory

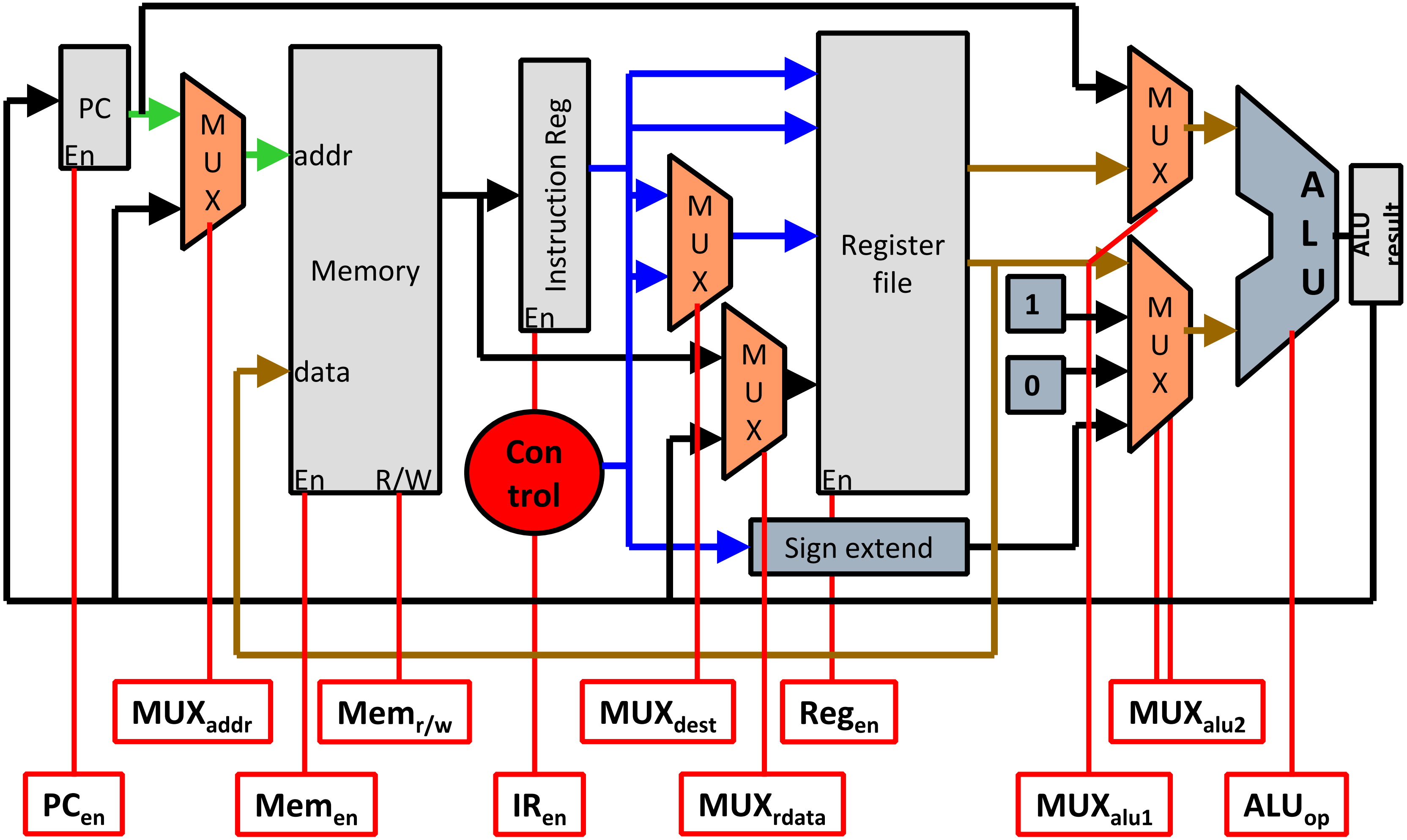
En

R/W

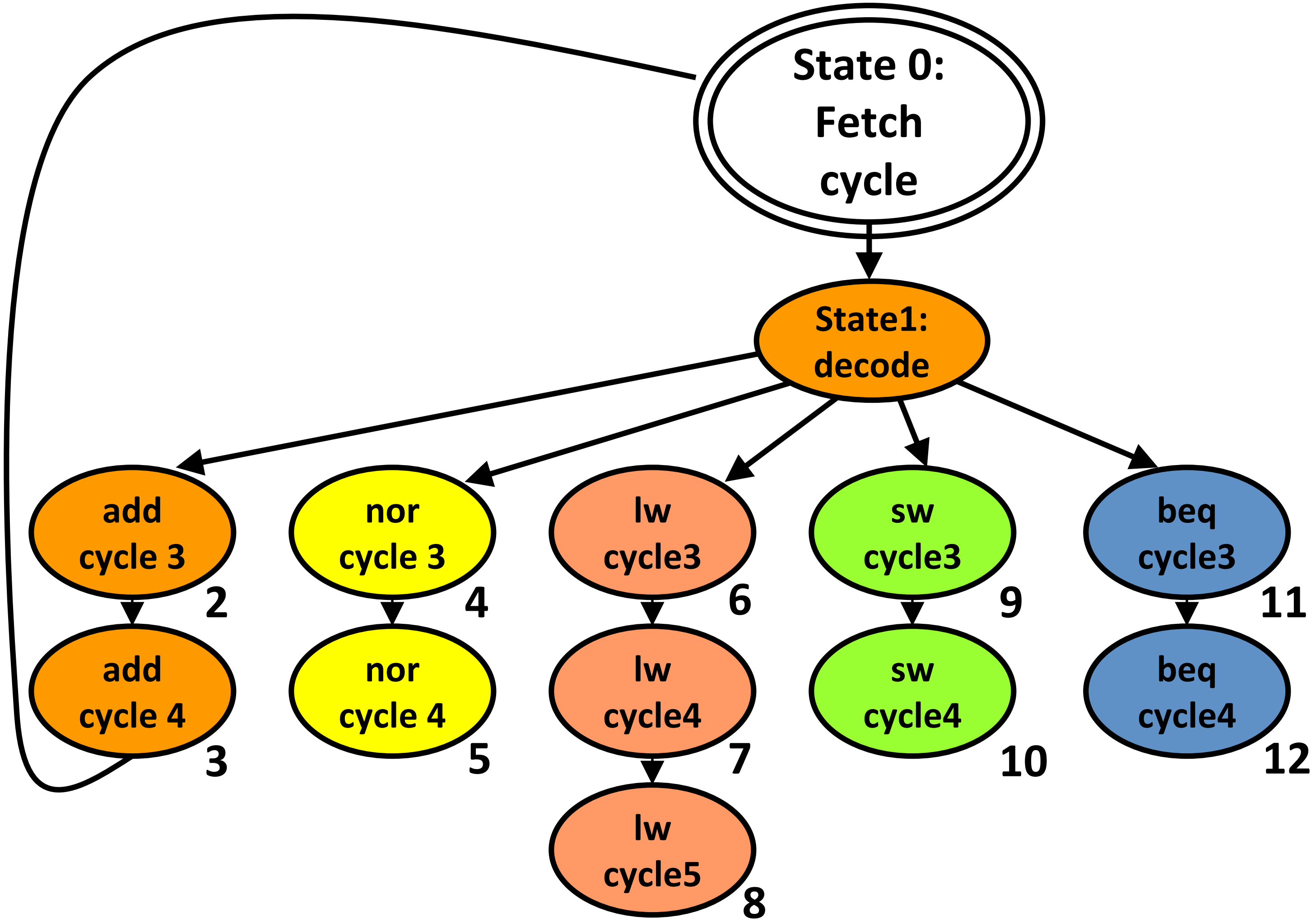
3x8 decoder

Control ROM

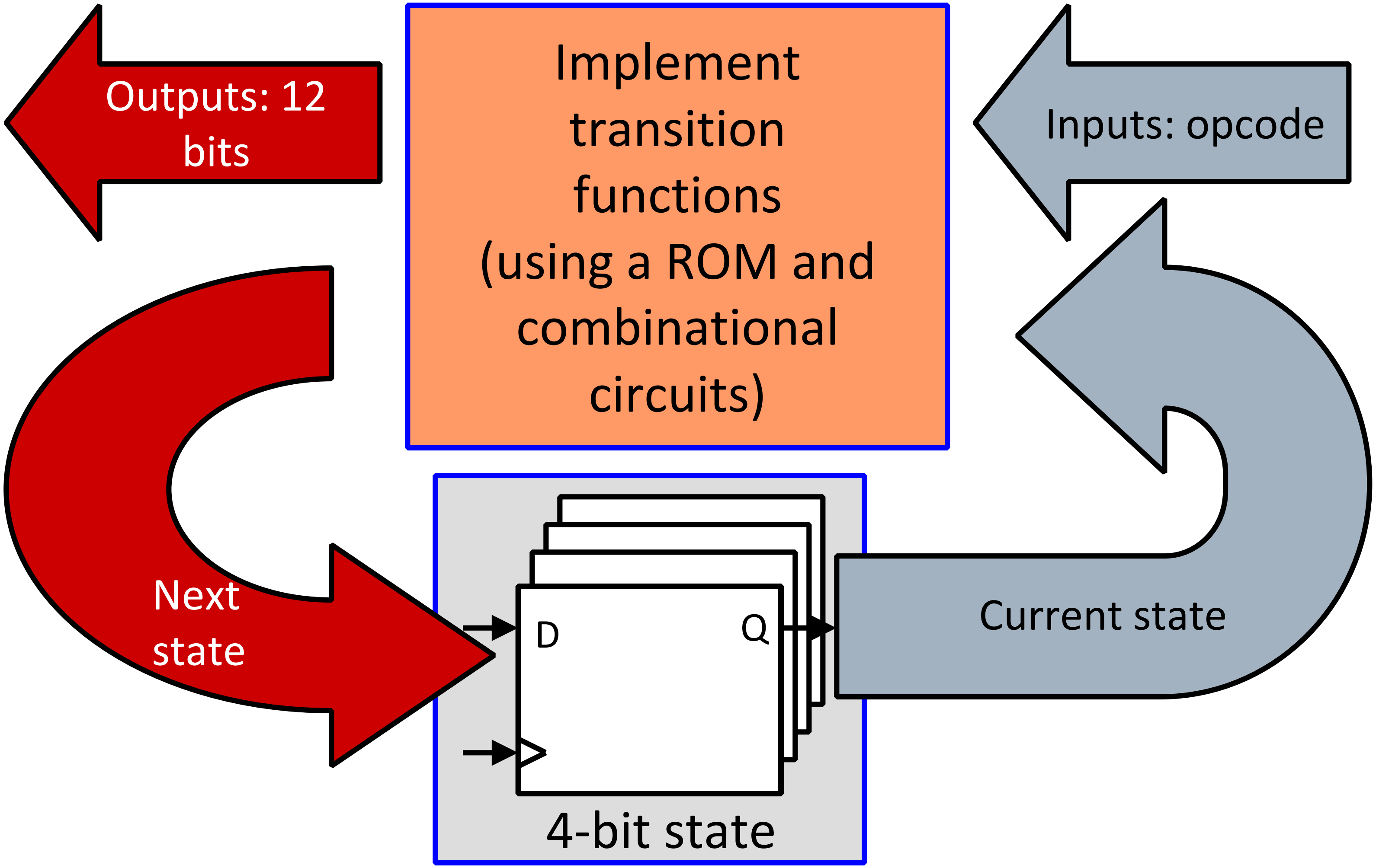
Multicycle LC2K Datapath



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

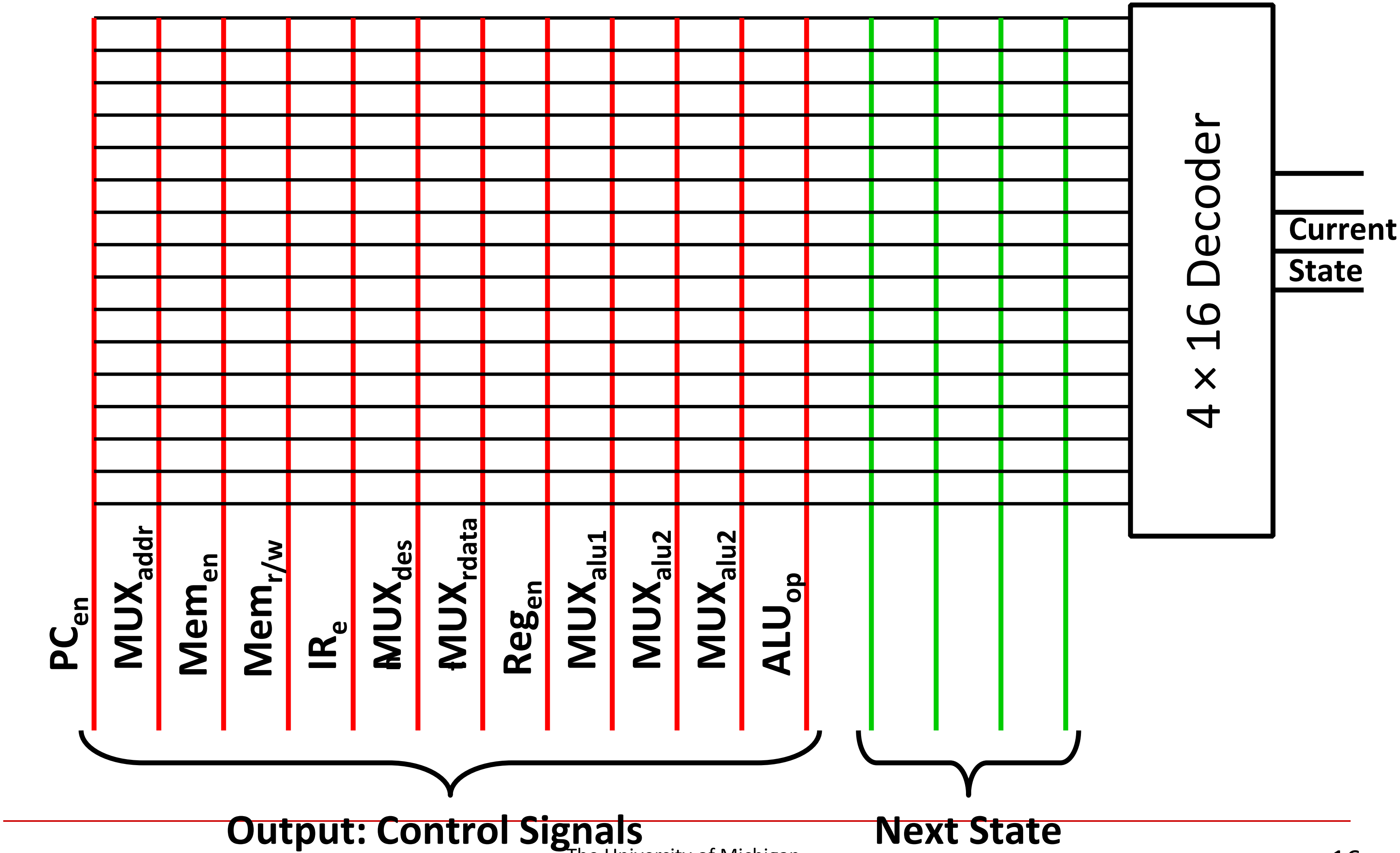


Implementing the FSM



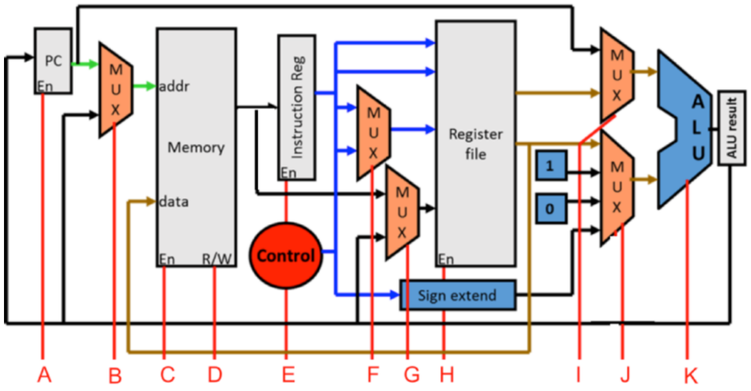
Implement transition functions (using a ROM and combinational circuits)

Building the Control ROM



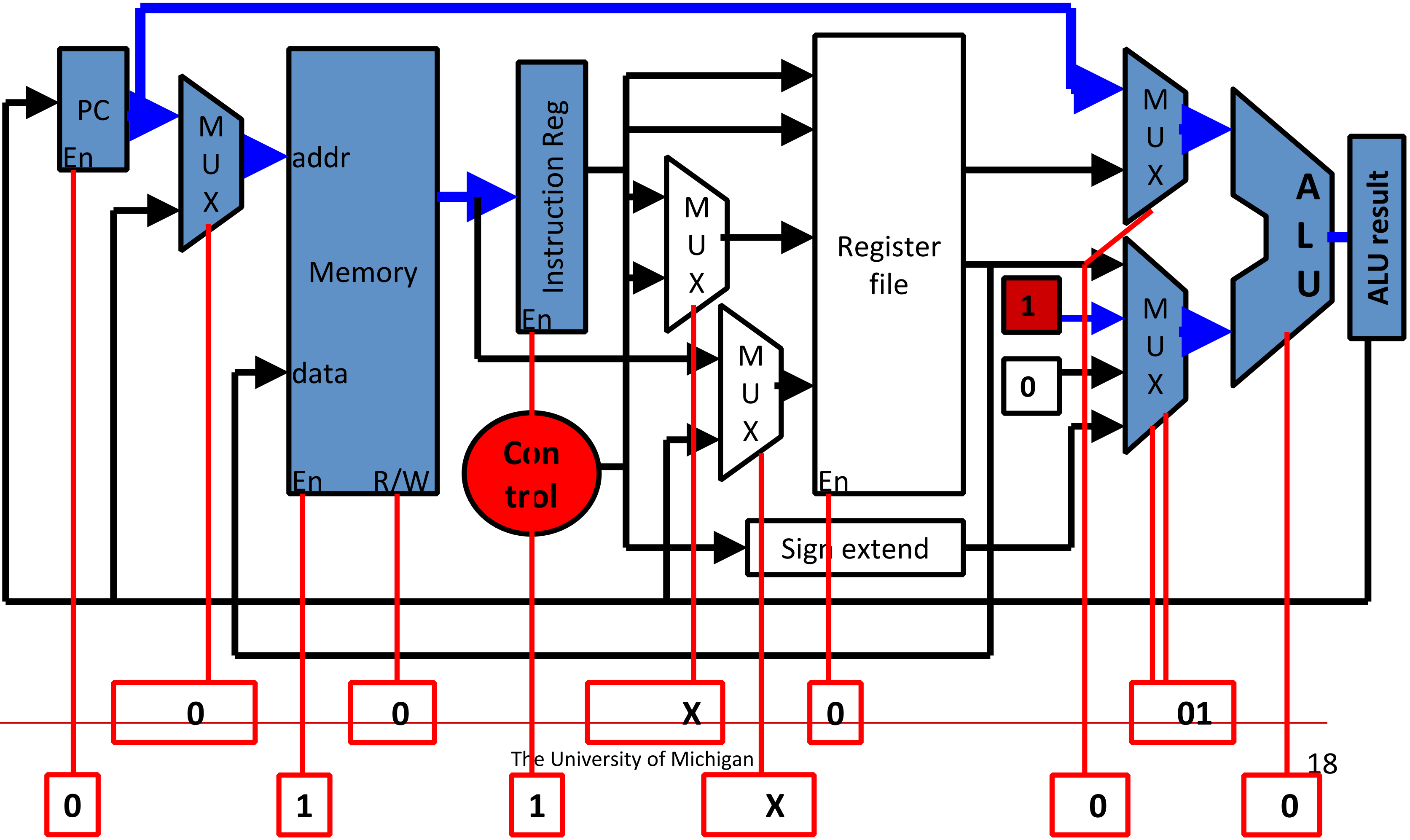
First Cycle (State 0) Fetch Instr

- 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 ($\text{MUX}_{\text{addr}}= 0$)
    - Enable memory operation ($\text{Mem}_{\text{en}}= 1$)
    - R/W should be (read) ($\text{Mem}_{\text{r/w}}= 0$)
    - Enable Instruction Register write ($\text{IR}_{\text{en}}= 1$)
  - Calculate PC + 1
    - Send PC to ALU ($\text{MUX}_{\text{alu1}} = 0$)
    - Send 1 to ALU ($\text{MUX}_{\text{alu2}} = 01$)
    - Select ALU add operation ($\text{ALU}_{\text{op}} = 0$)
  - $\text{PC}_{\text{en}} = 0$; $\text{Reg}_{\text{en}} = 0$; MUX_{dest} and $\text{MUX}_{\text{rdata}}= X$
- Next State: Decode Instruction

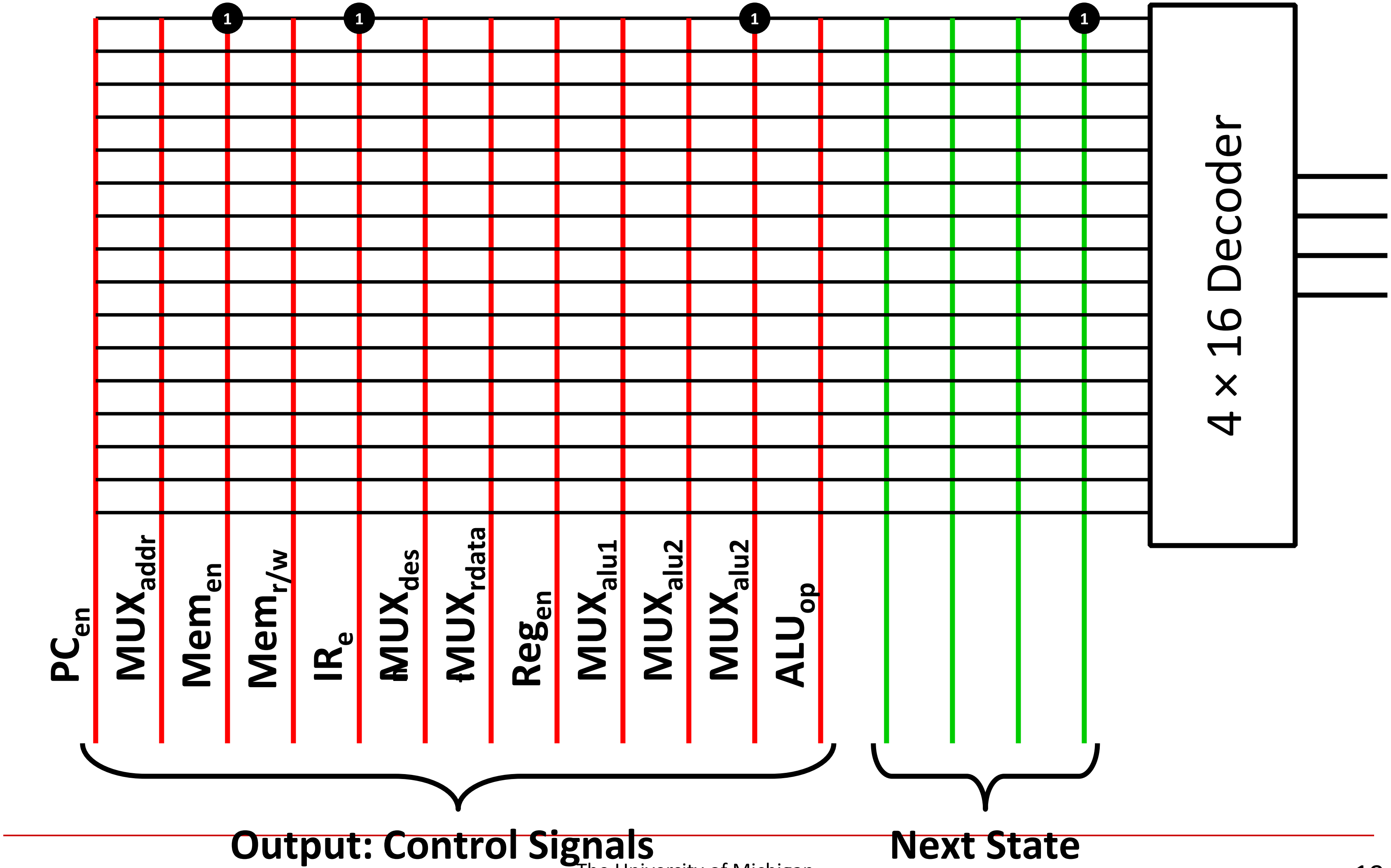


First Cycle (State 0) Operation

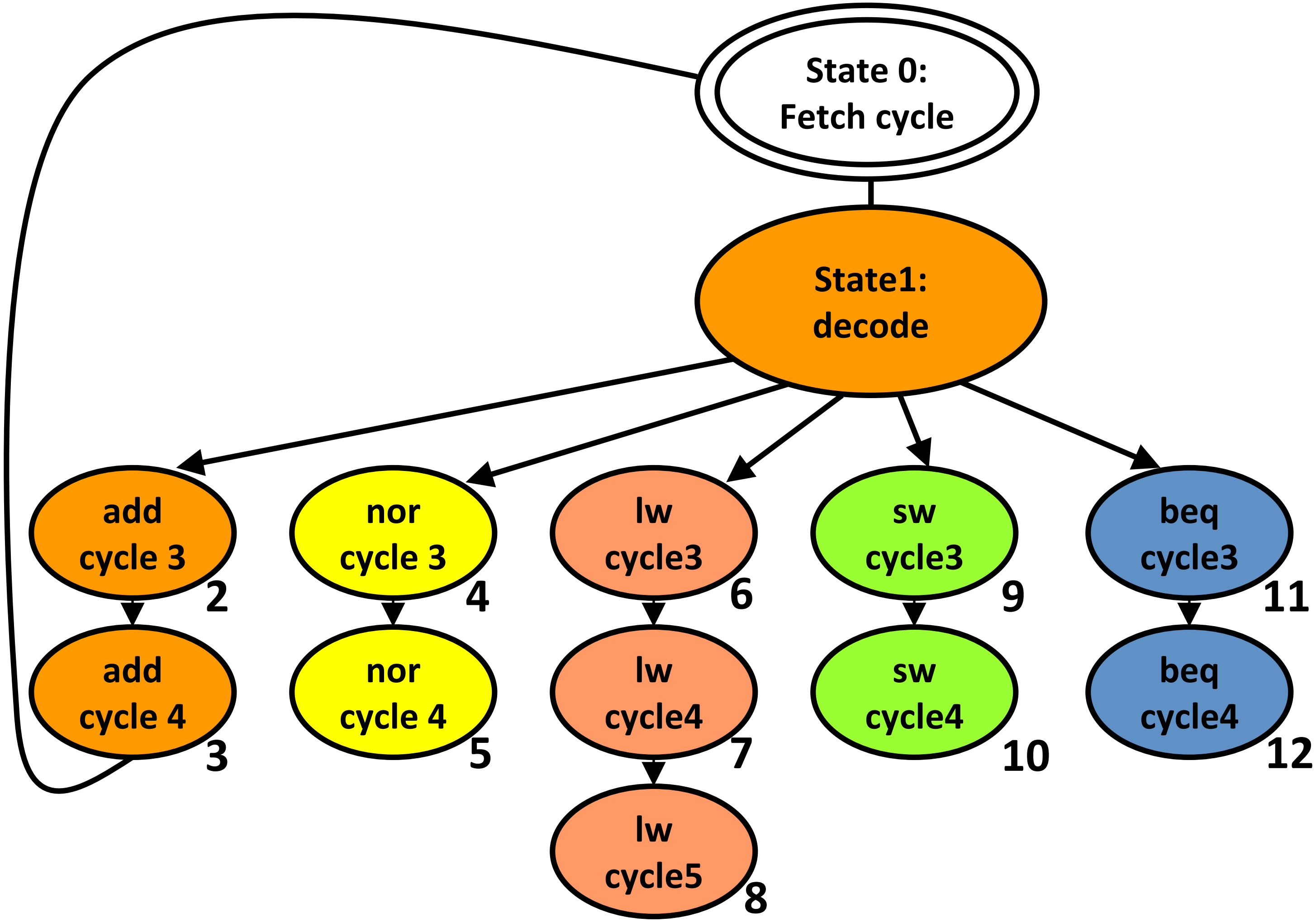
This is the same for all instructions
(since we don't know the instruction yet!)



Building the Control Rom



State 1: instruction decode



State 0: Fetch cycle

State1: decode

add cycle 3

2

add cycle 4

3

nor cycle 3

4

nor cycle 4

5

lw cycle3

6

lw cycle4

7

lw cycle5

8

sw cycle3

9

sw cycle4

10

beq cycle3

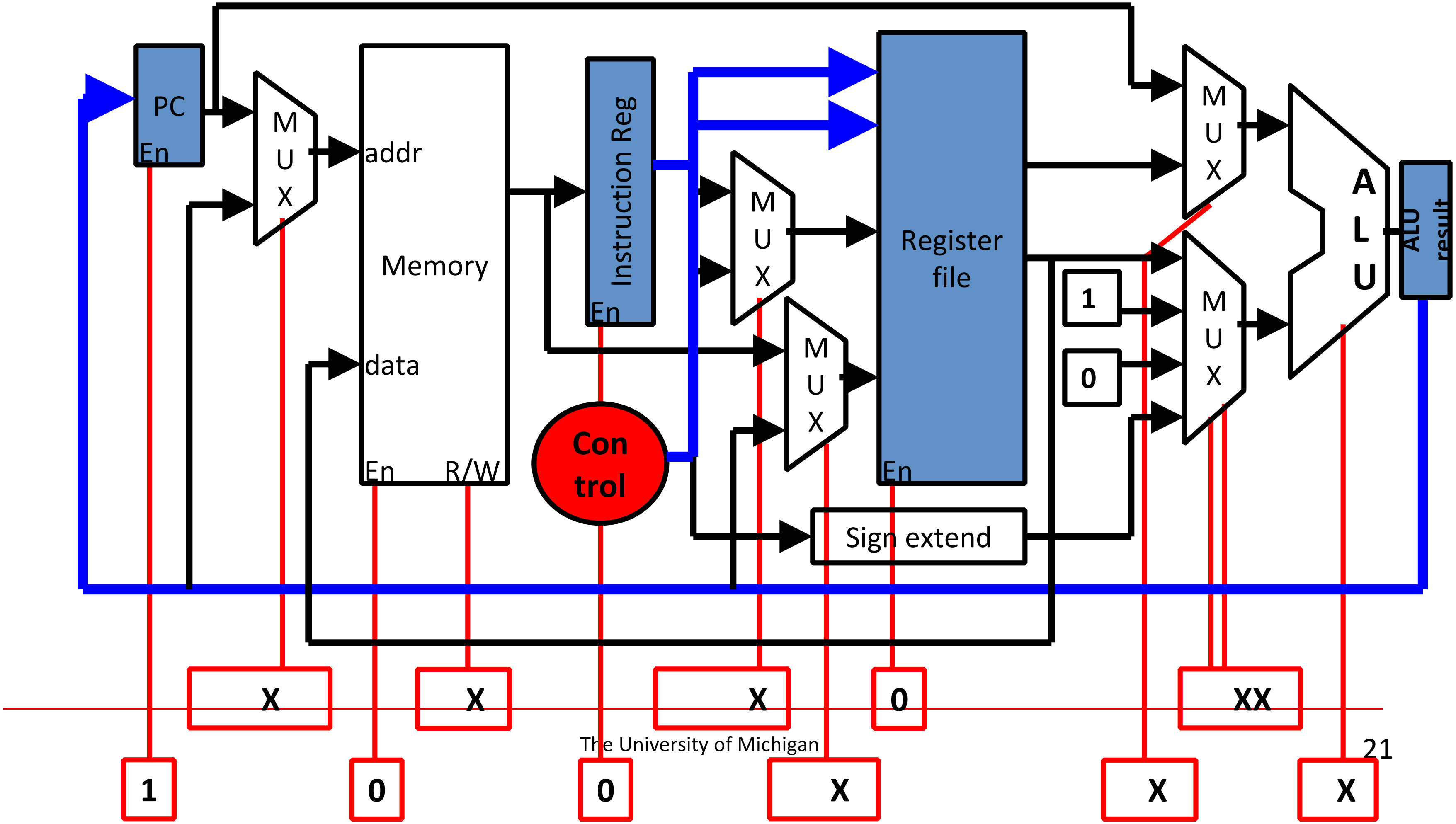
11

beq cycle4

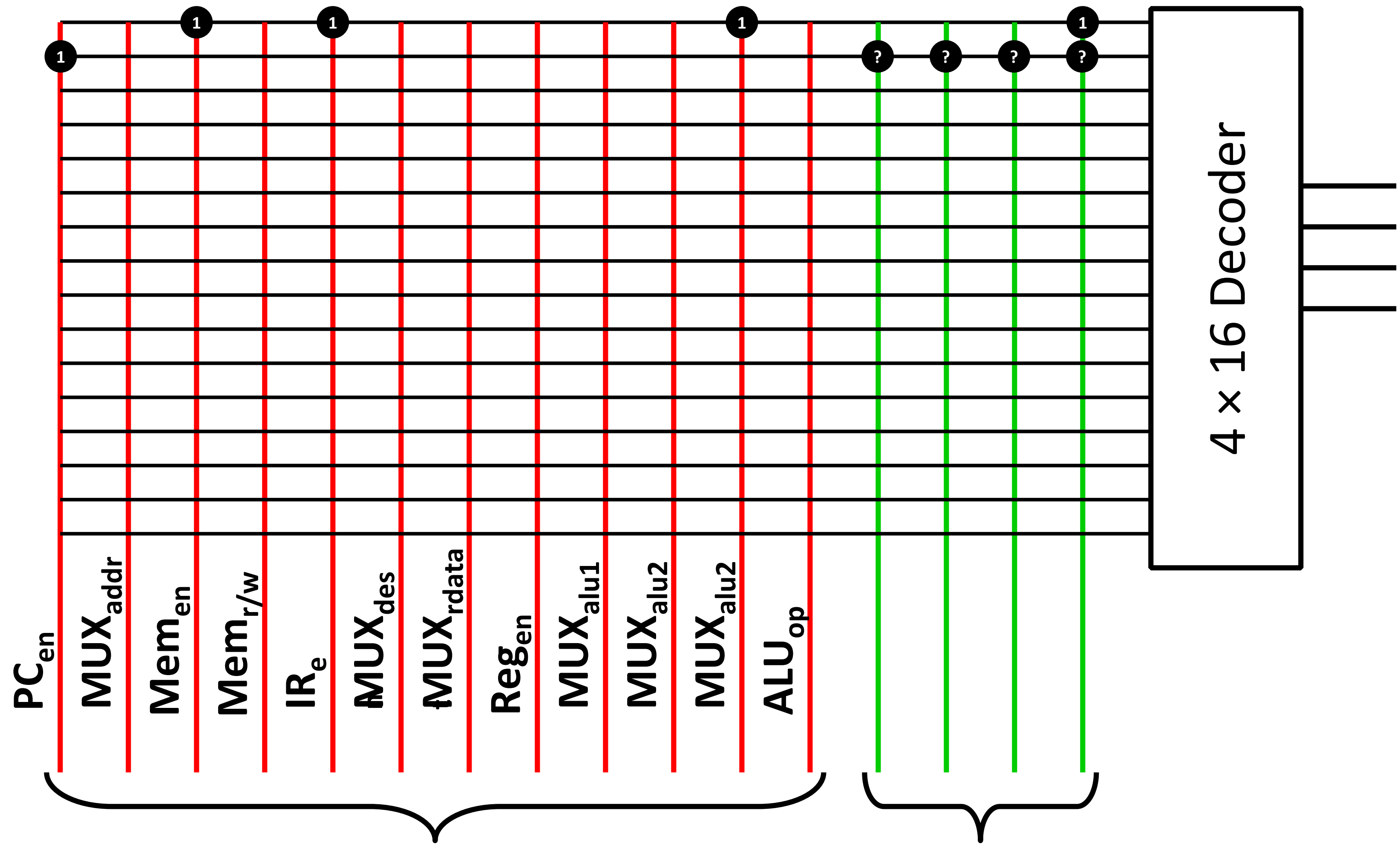
12

State 1: output function

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



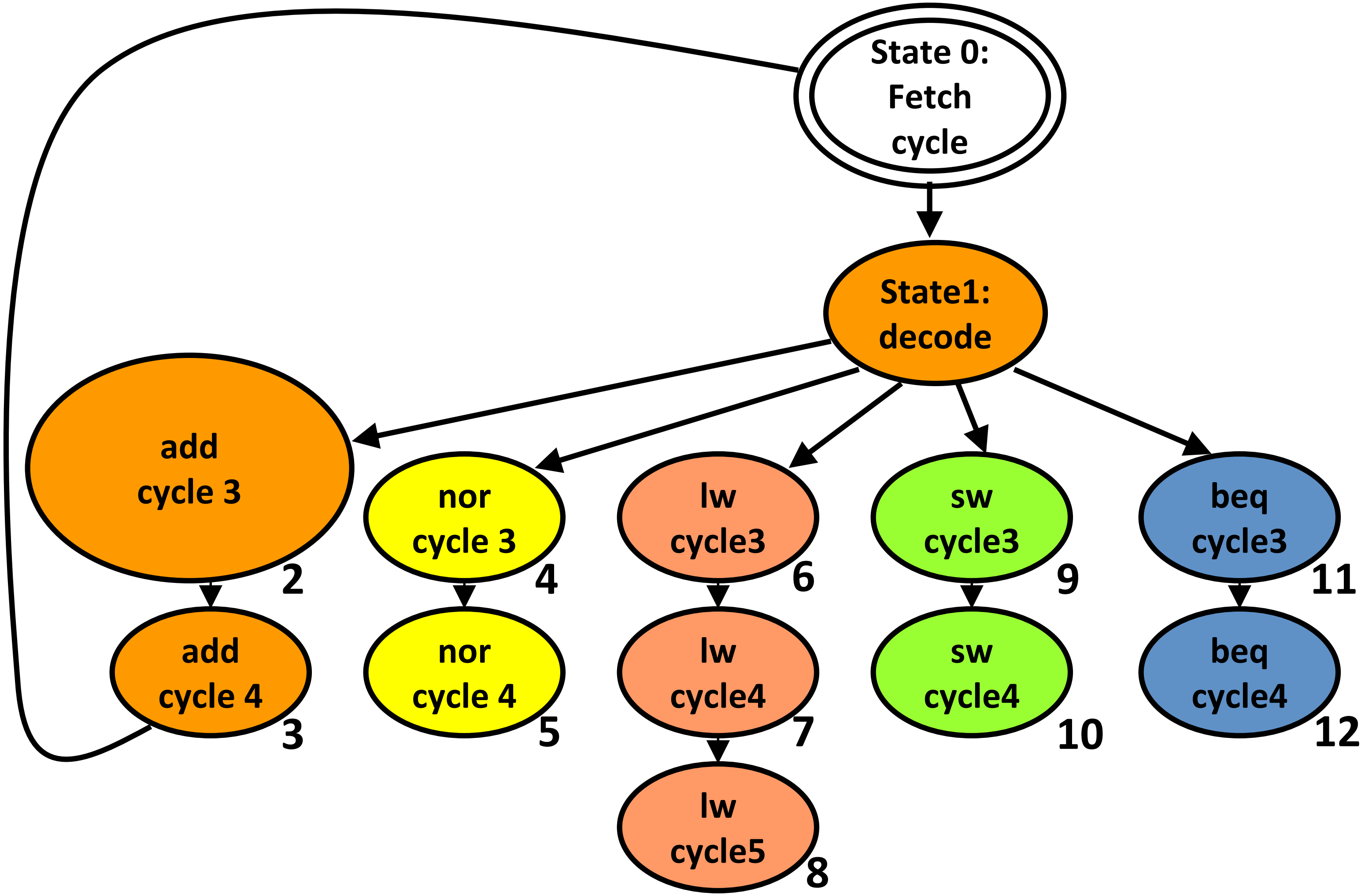
Building the Control Rom



Output: Control Signals

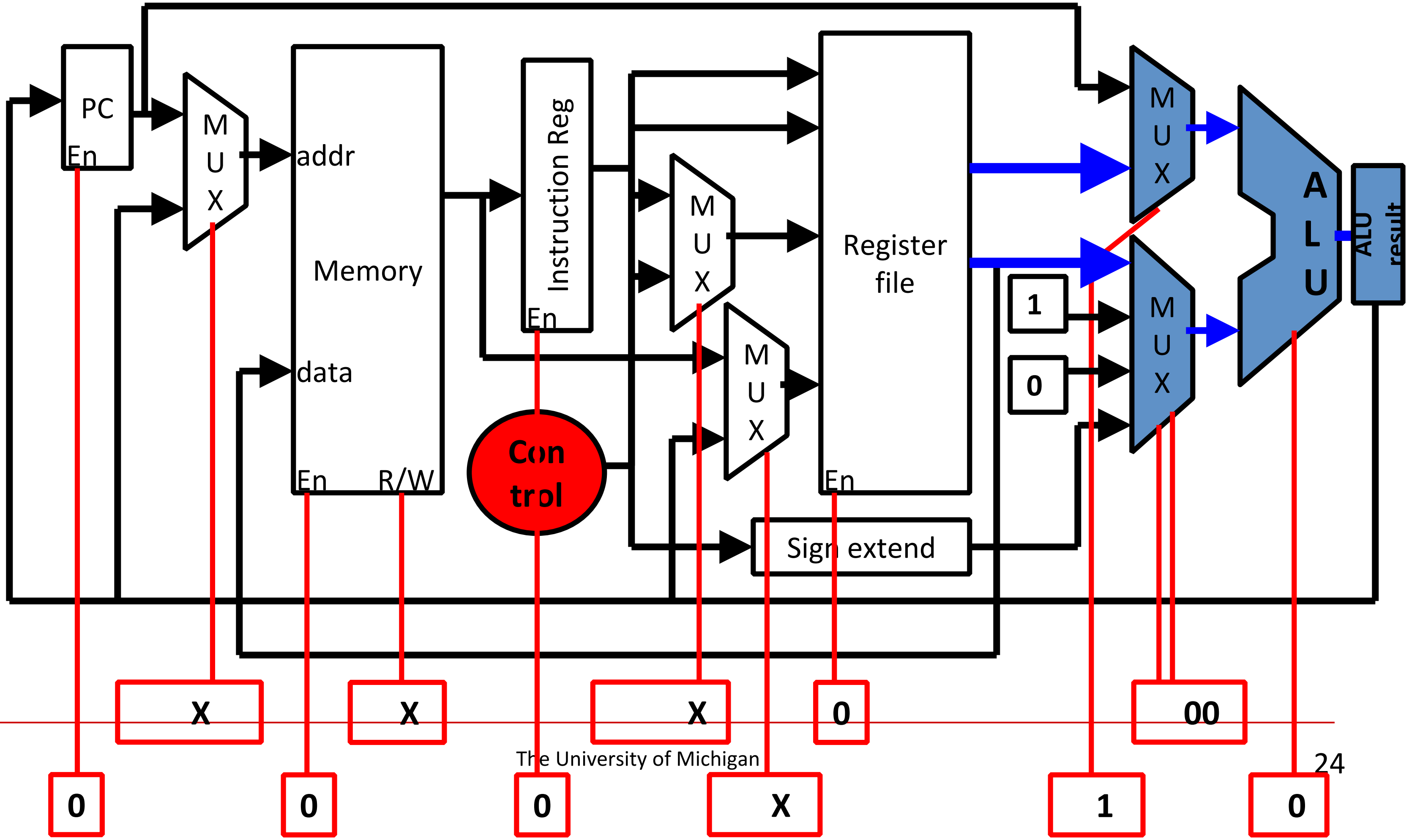
Next State

State 2: Add Cycle 3

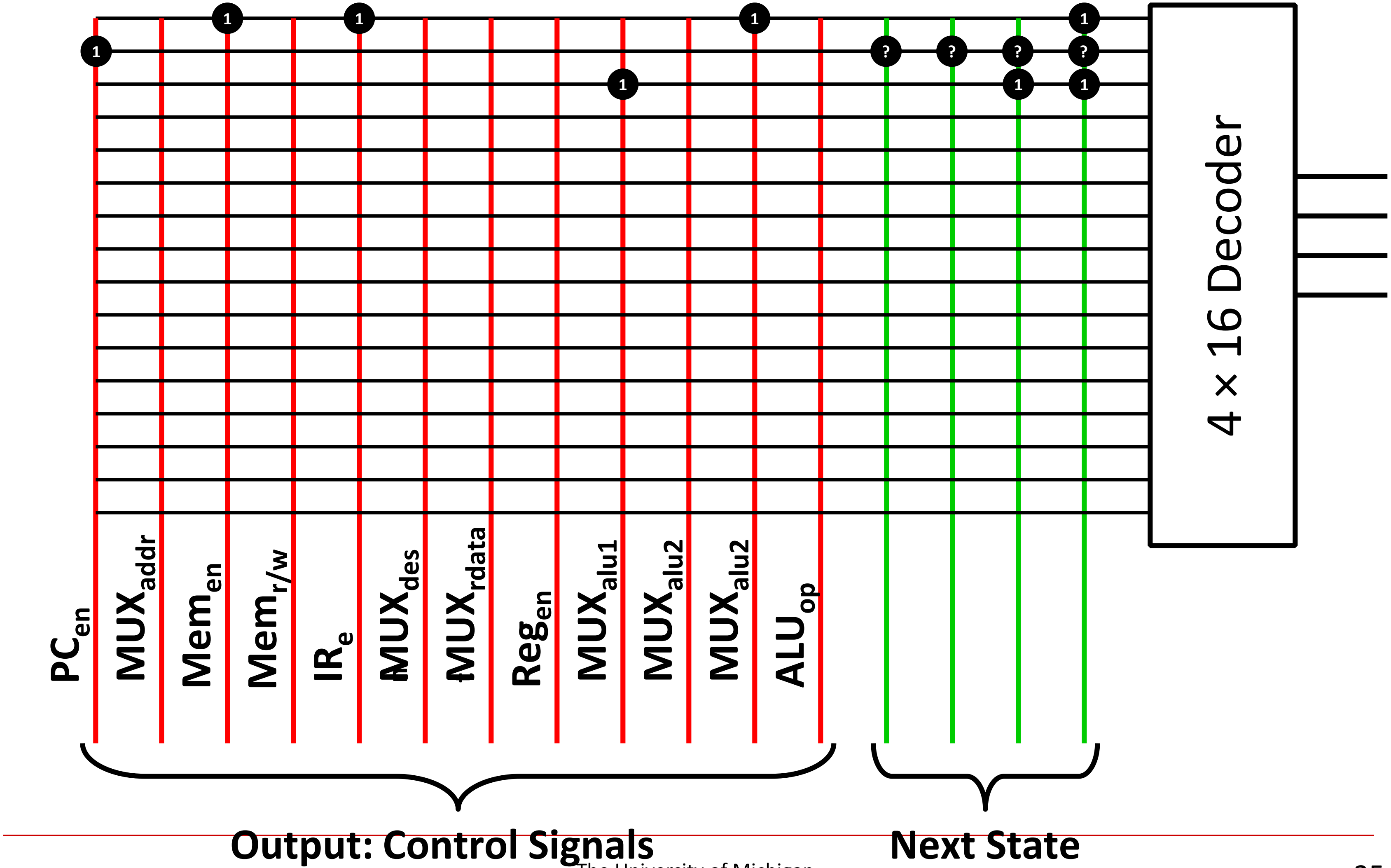


State 2: Add Cycle 3 Operation

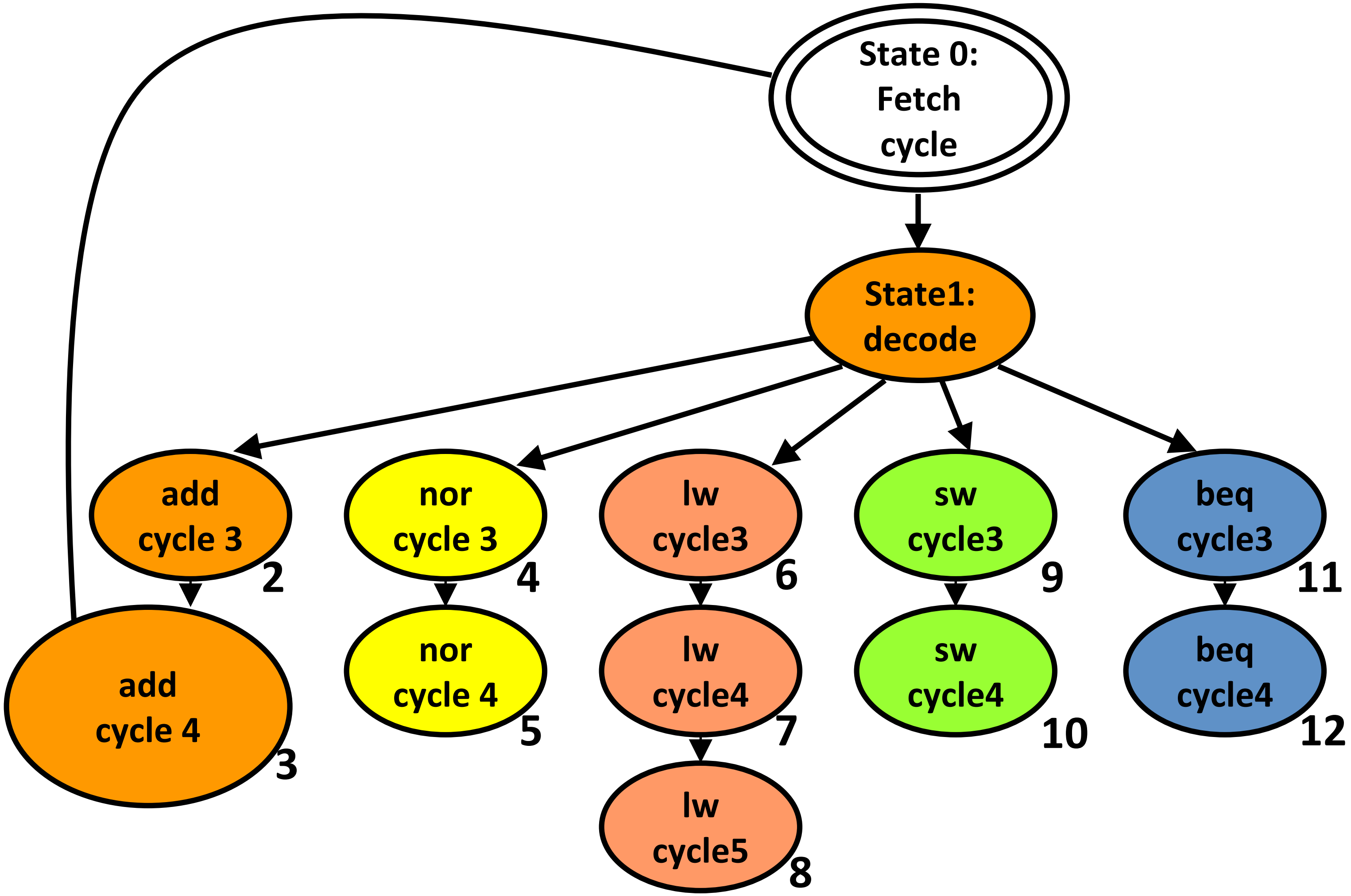
Send control signals to MUX to select values of regA and regB and control signal to ALU to add



Building the Control Rom

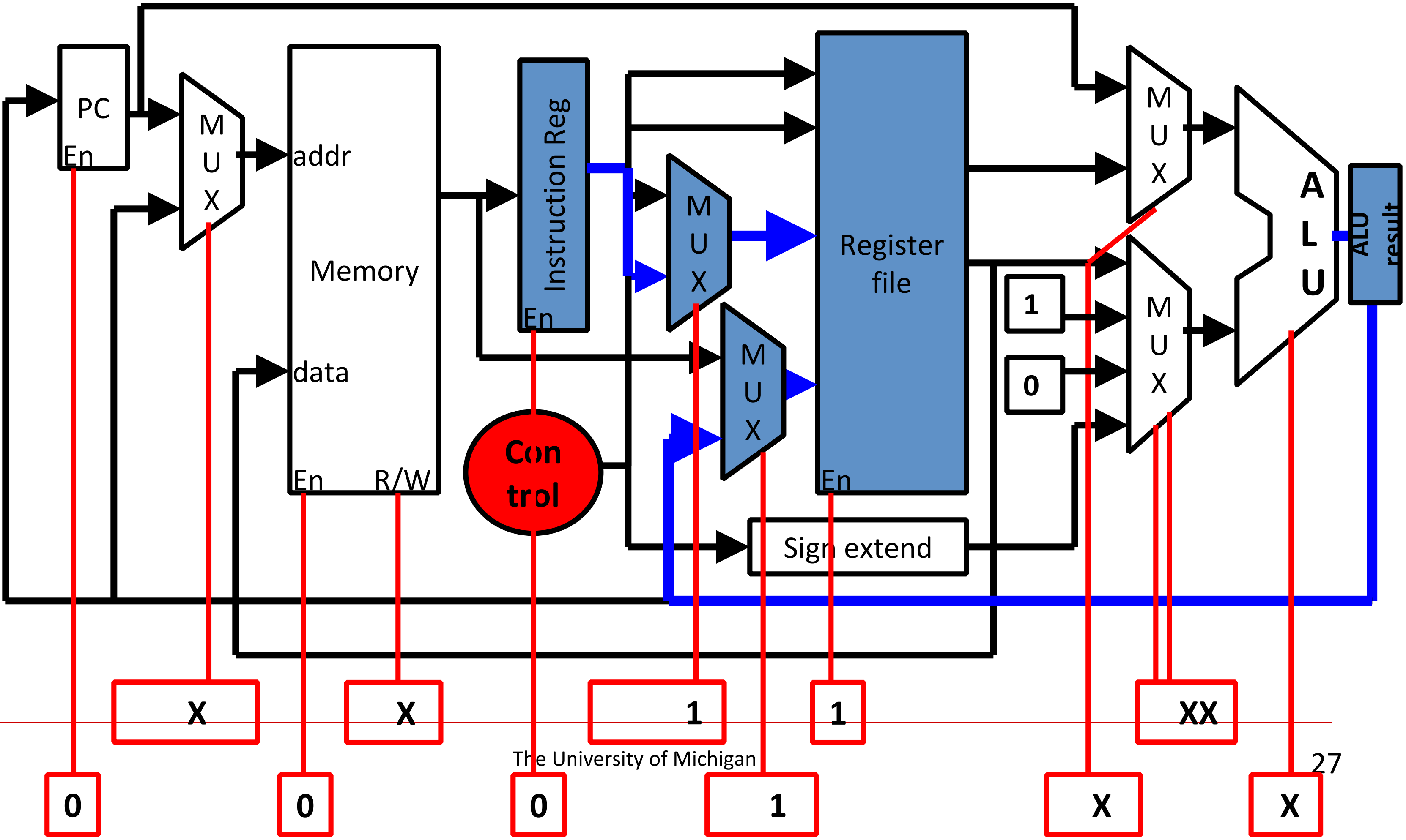


State 3: Add cycle 4

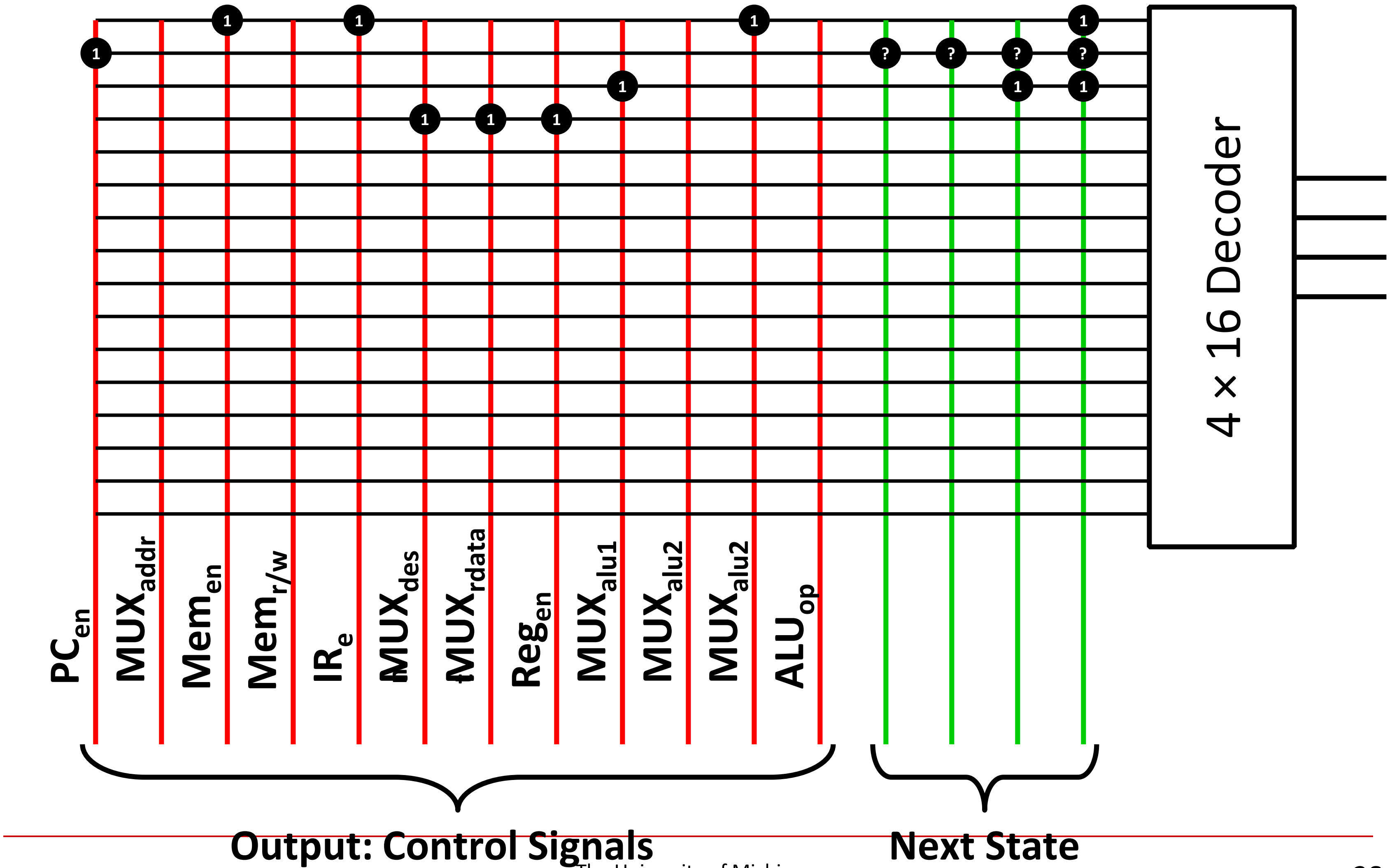


Add Cycle 4 (State 3) Operation

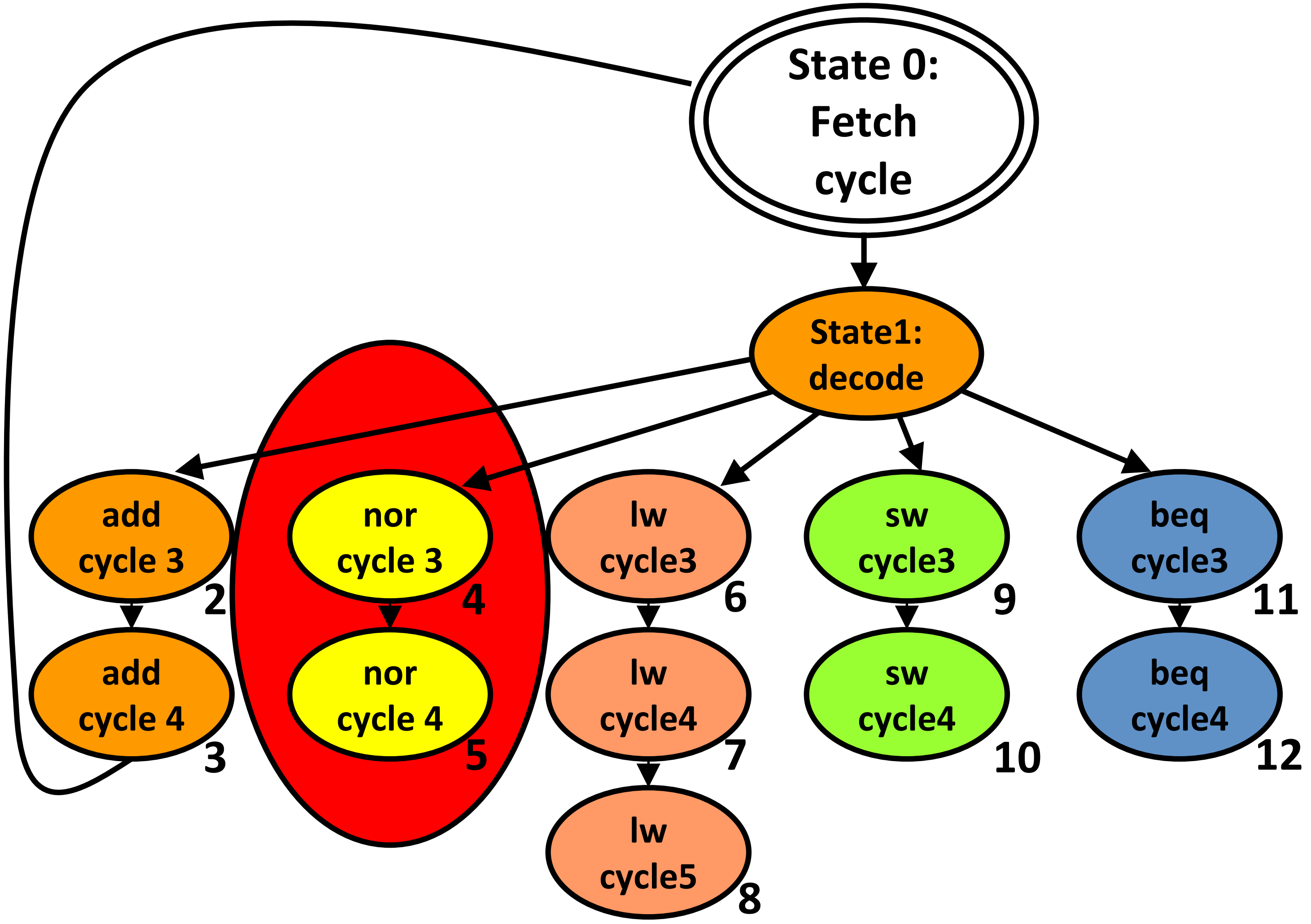
Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



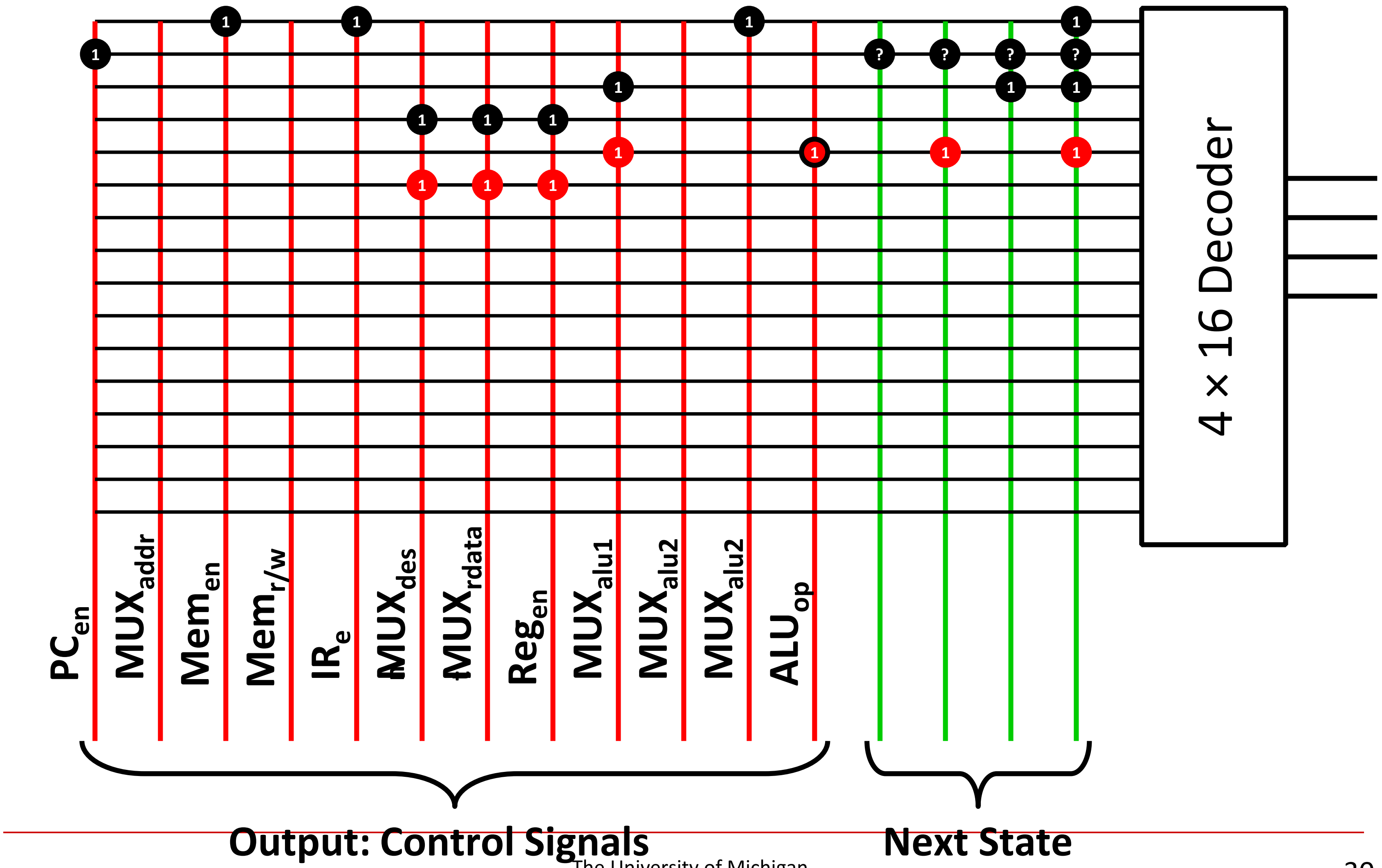
Building the Control Rom



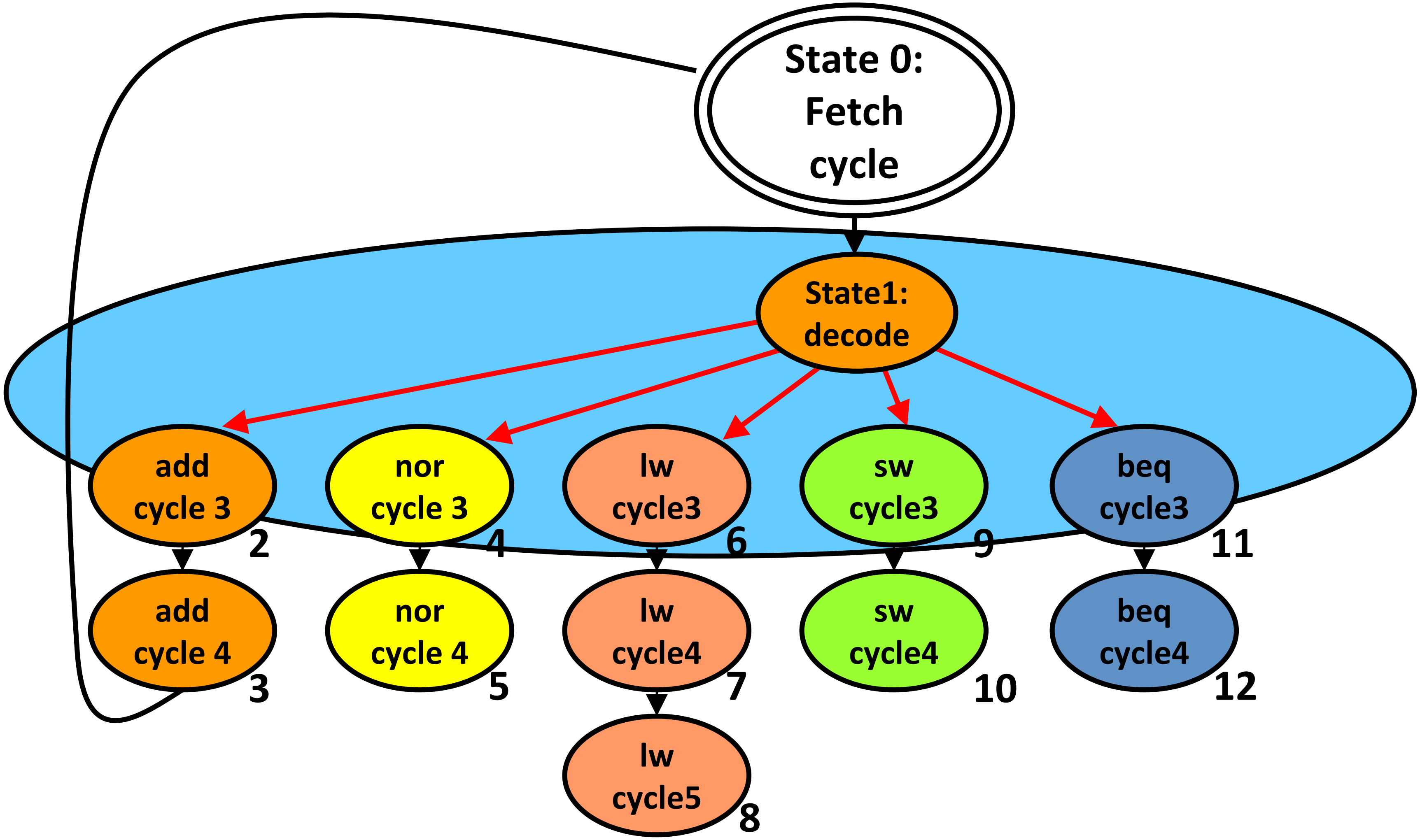
Return to State 0:
Fetch cycle to execute the next instruction



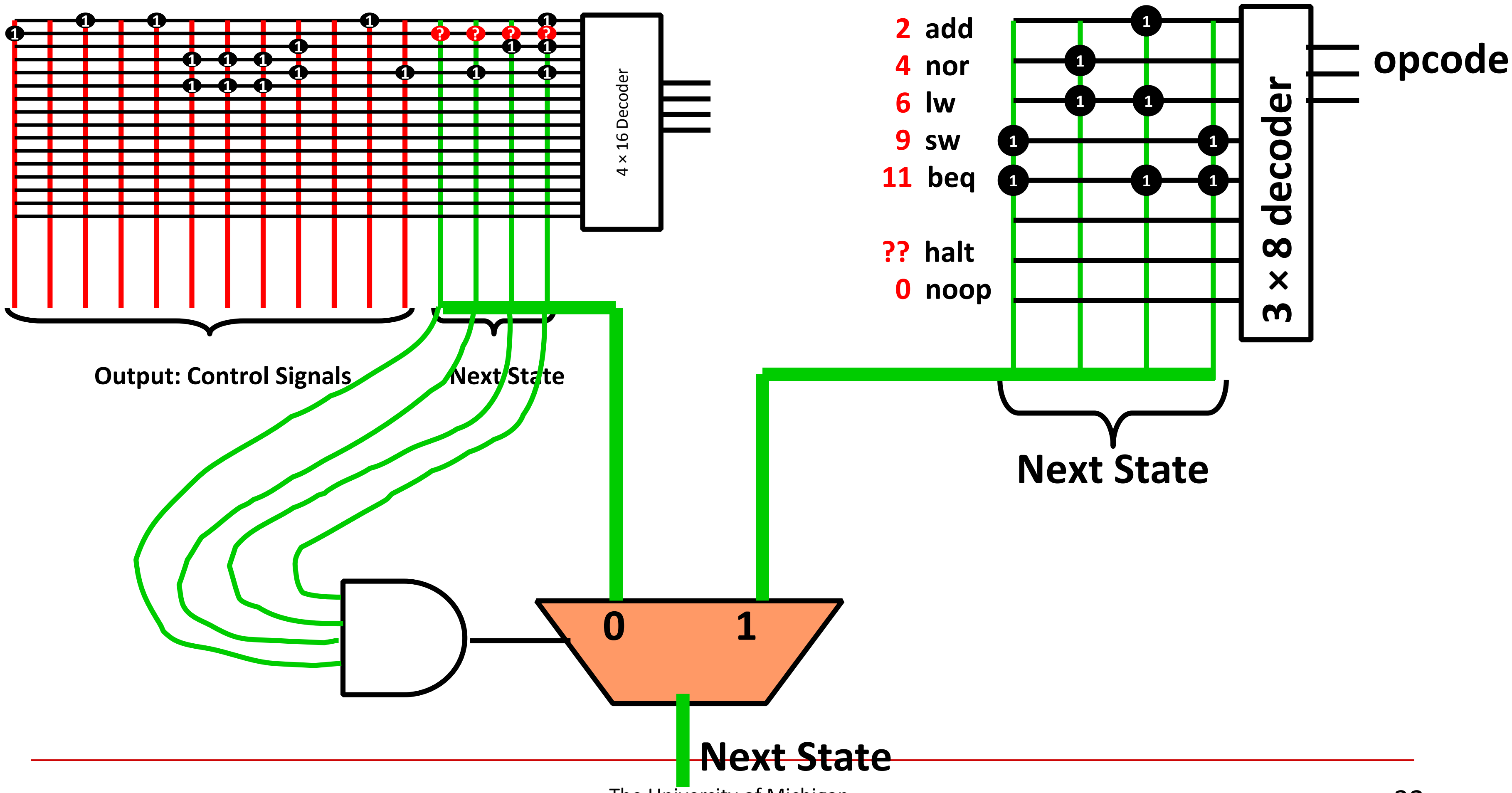
Control Rom for nor (4 and 5)



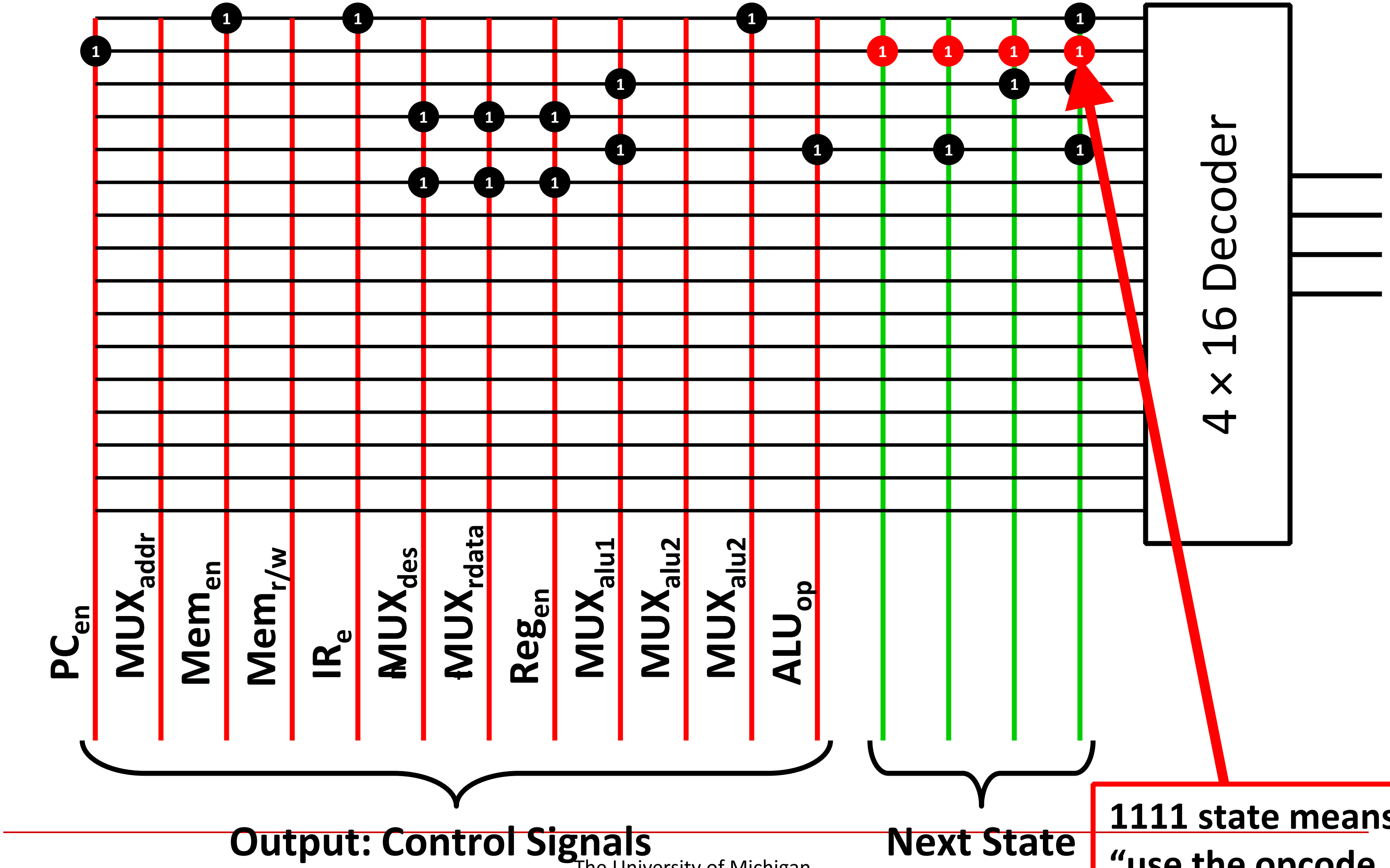
What about the transition from state 1?



Complete transition function circuit

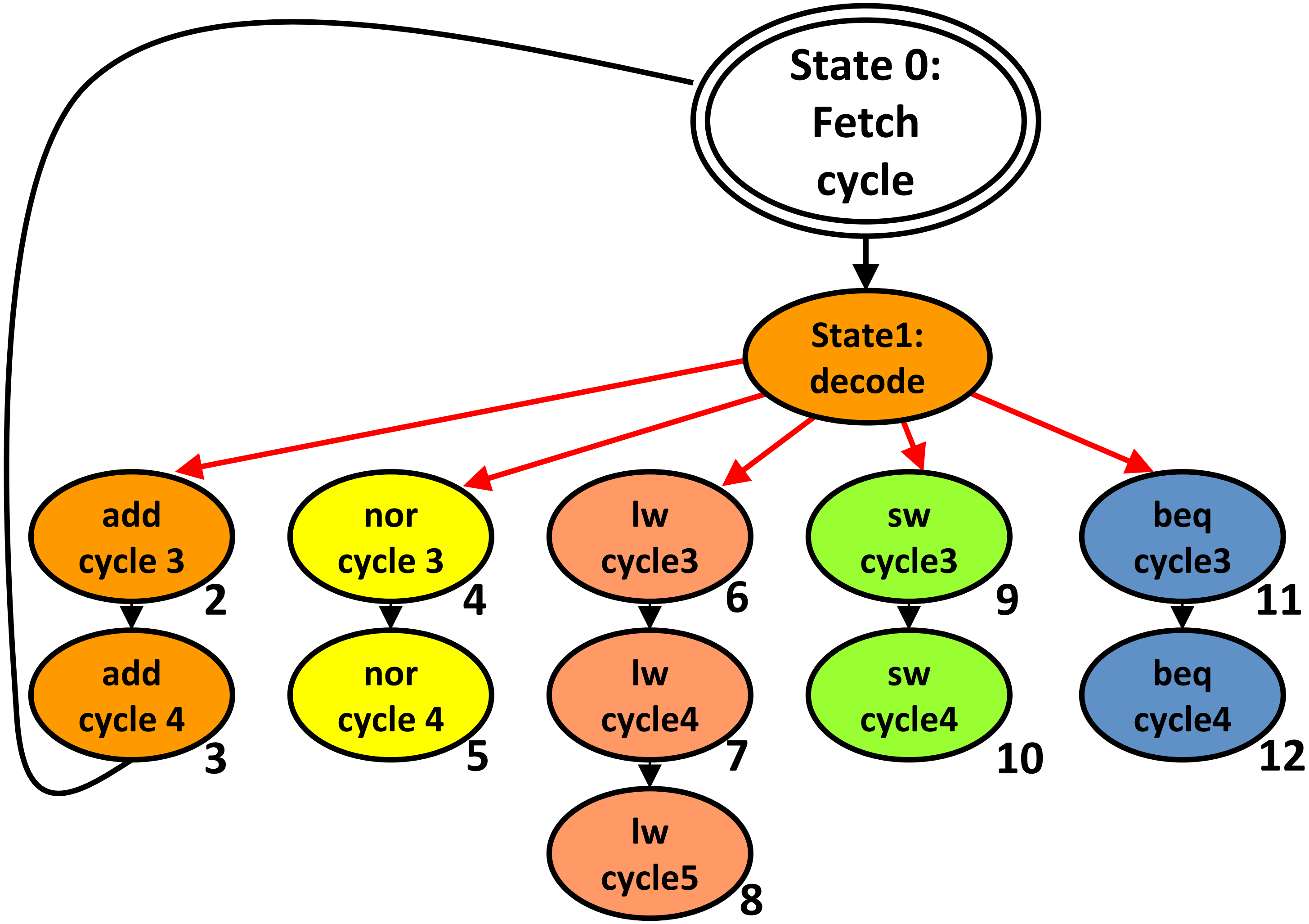


Control ROM (use of 1111 state)



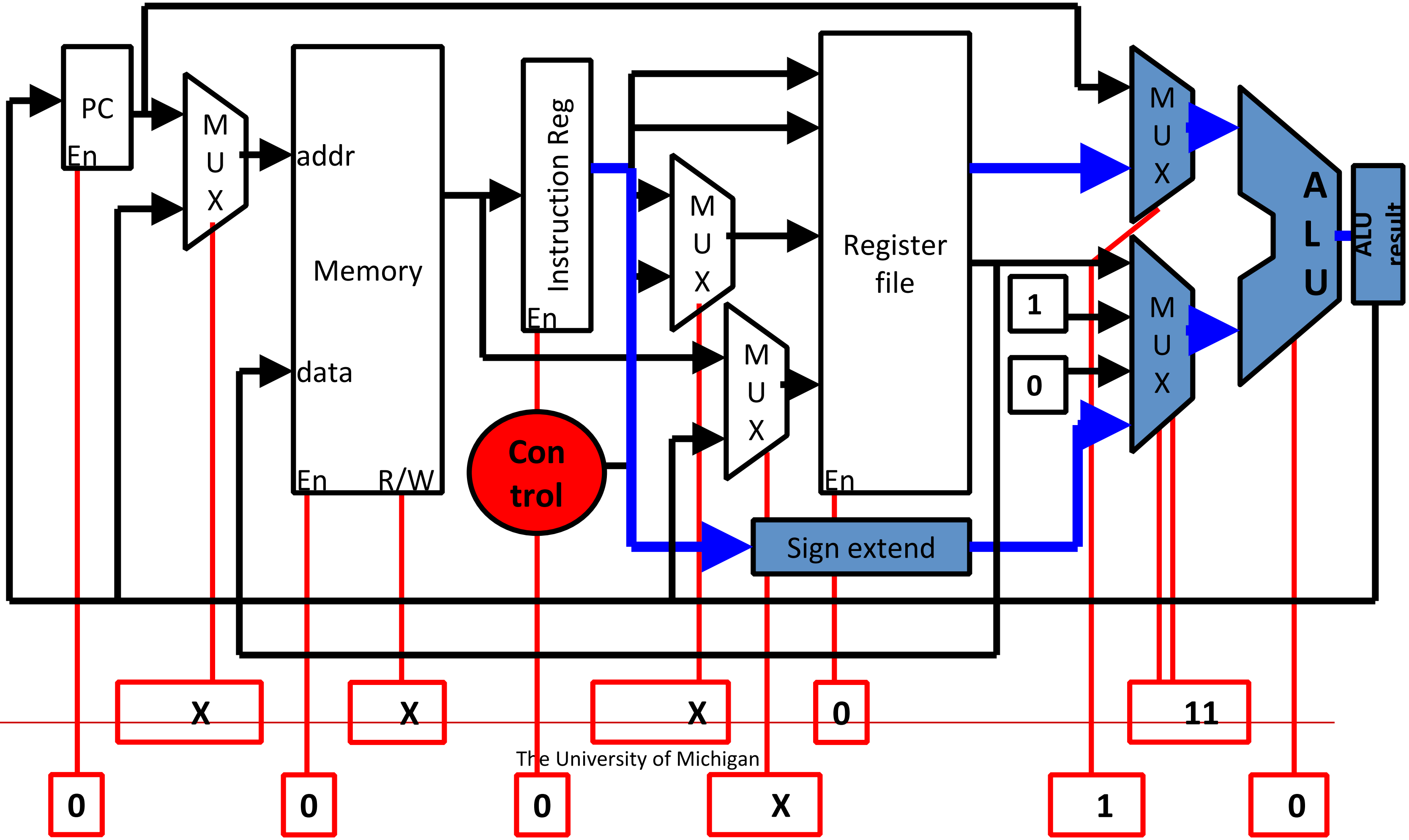
1111 state means
“use the opcode
to decide next state”

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

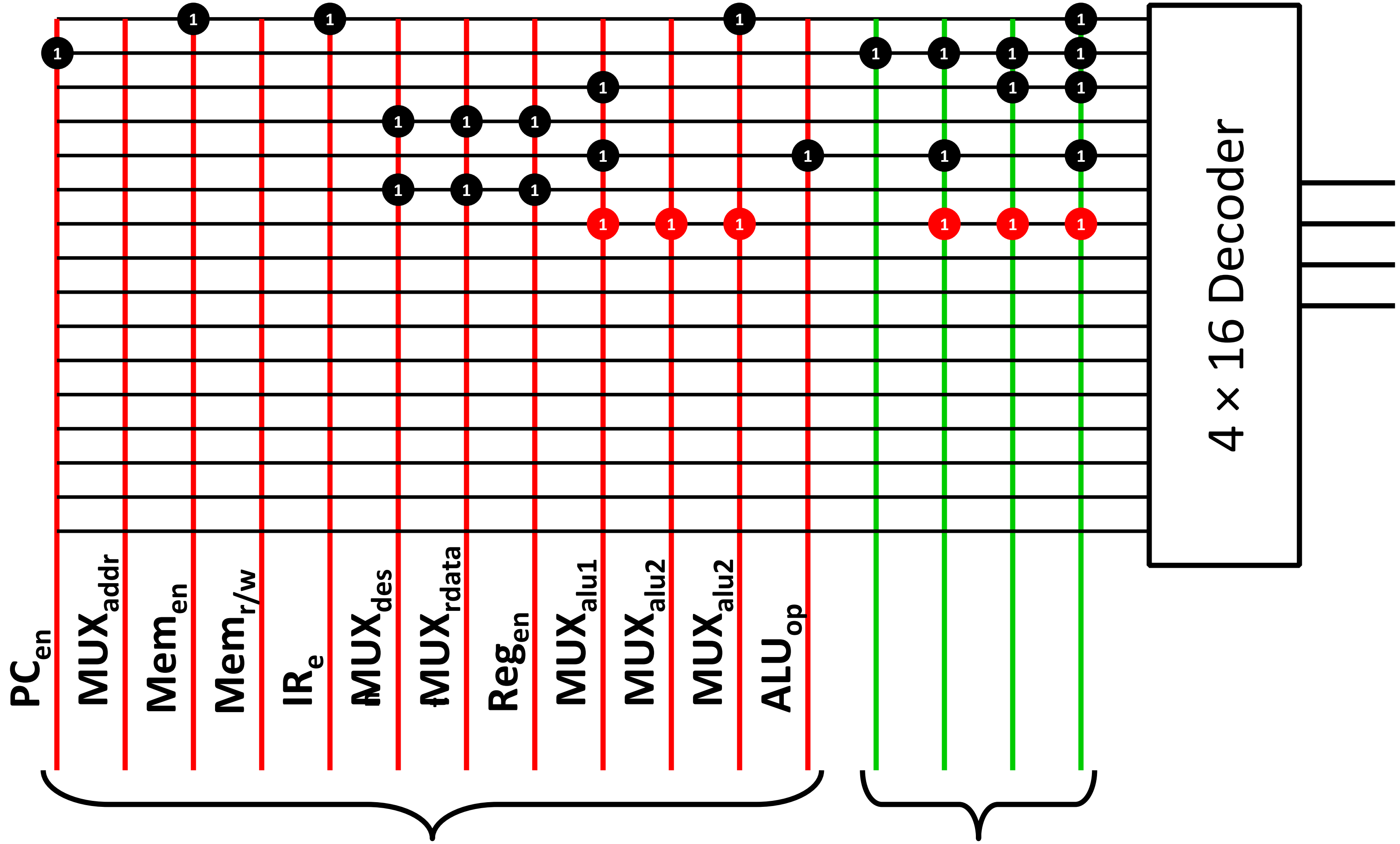


State 6: LW cycle 3

Calculate address for memory reference



Control Rom (lw cycle 3)



PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_e	MUX_{des}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1

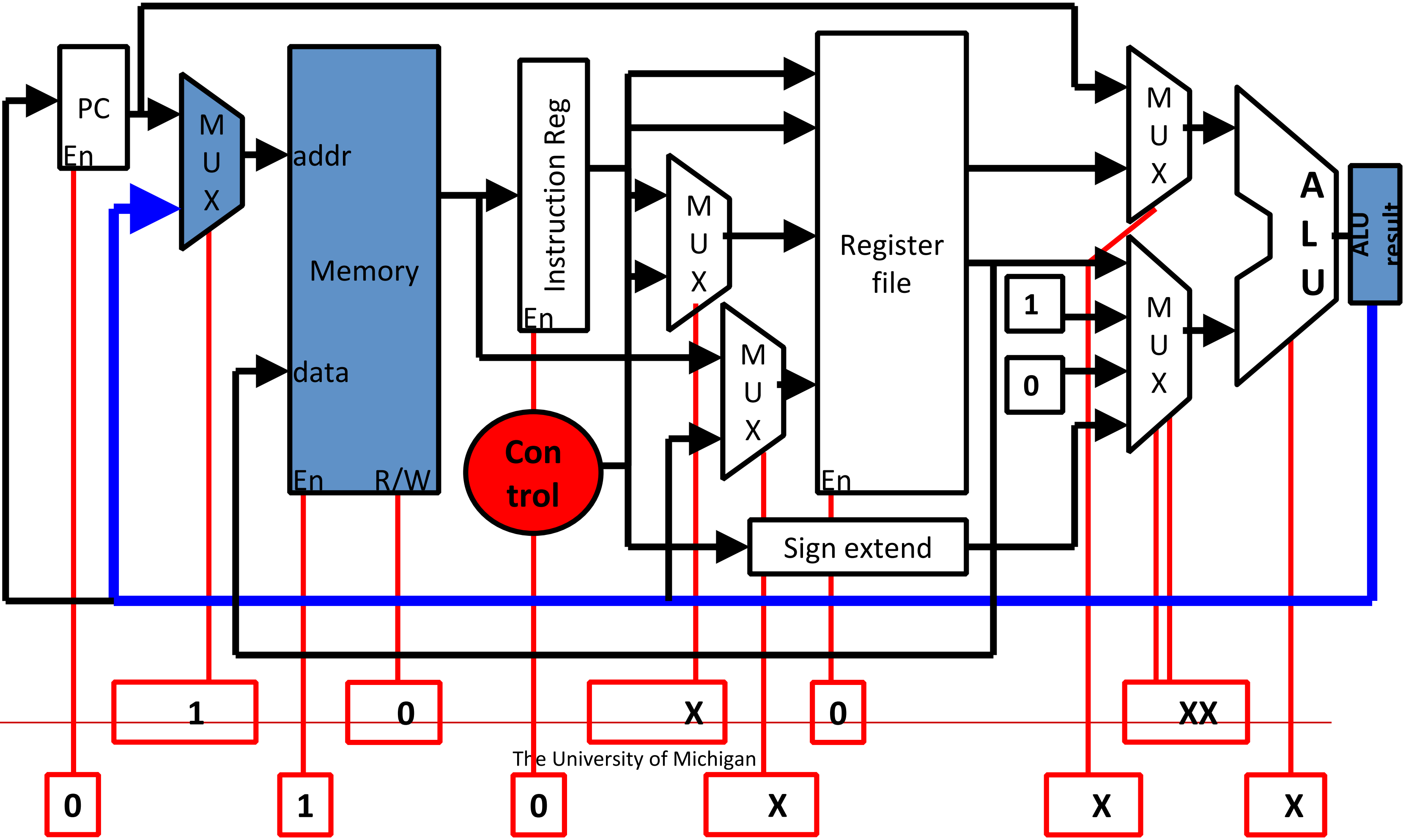
4 × 16 Decoder

Output: Control Signals

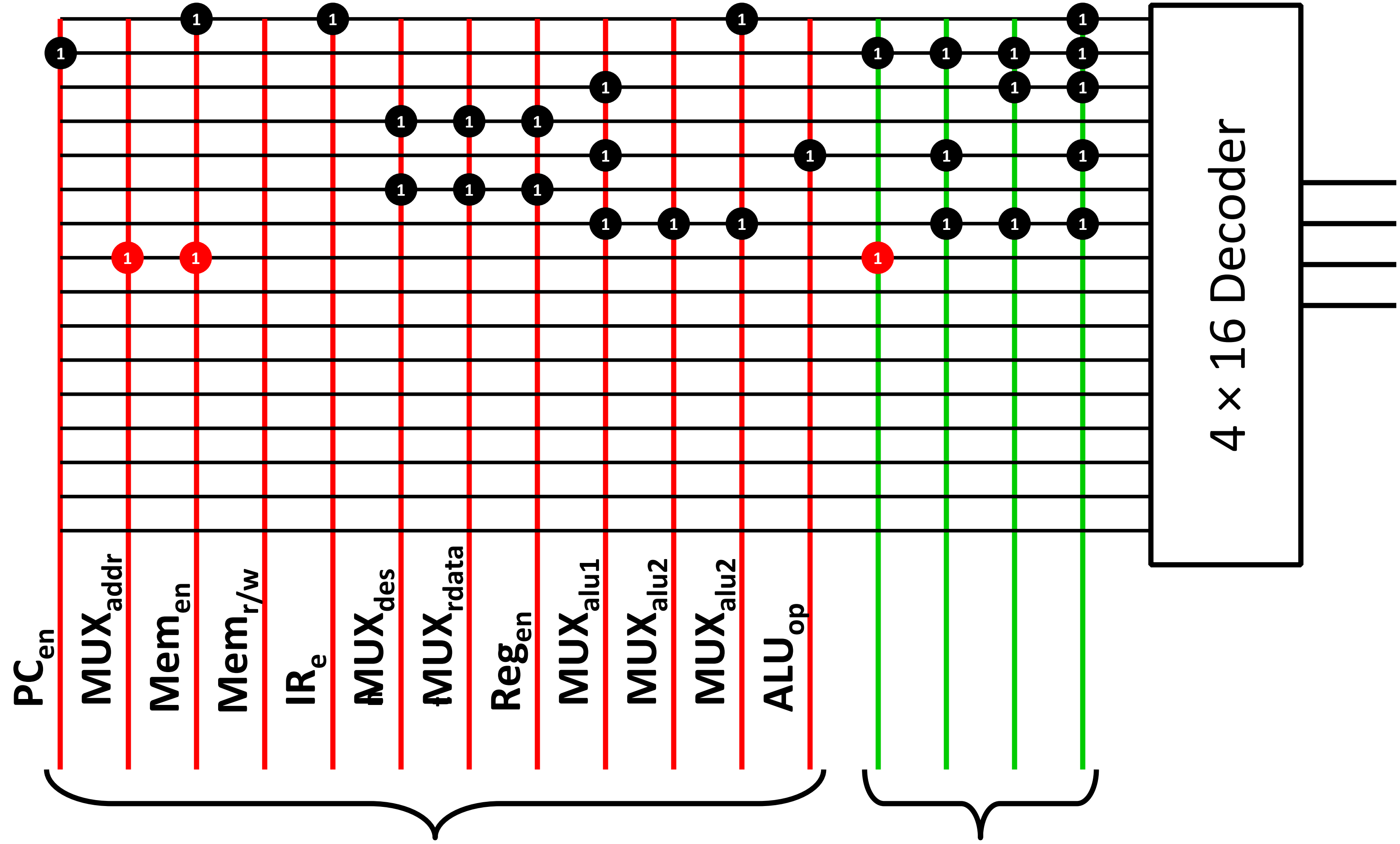
Next State

State 7: LW cycle 4

Read memory location



Control Rom (lw cycle 4)



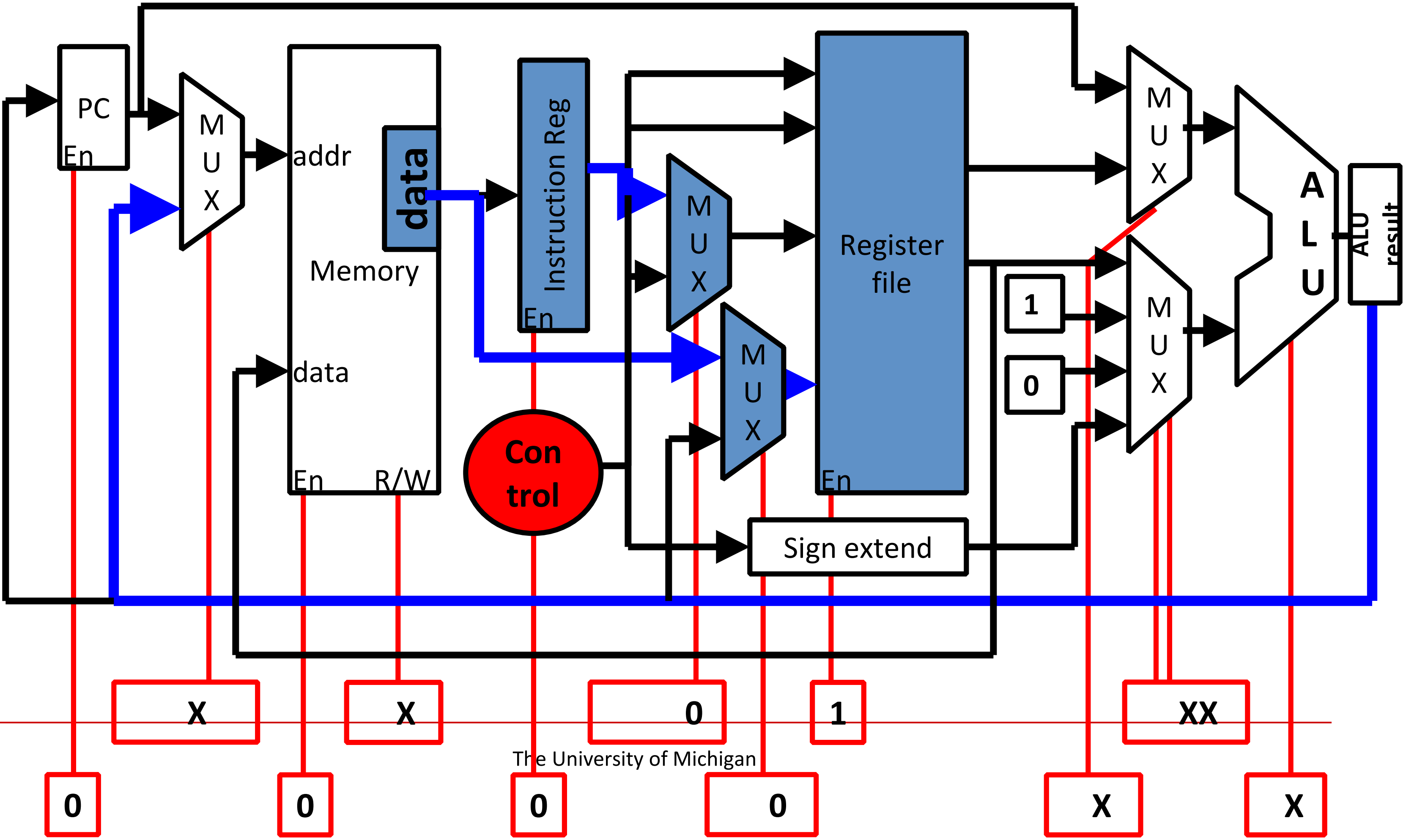
PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_e	MUX_{des}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			

Output: Control Signals

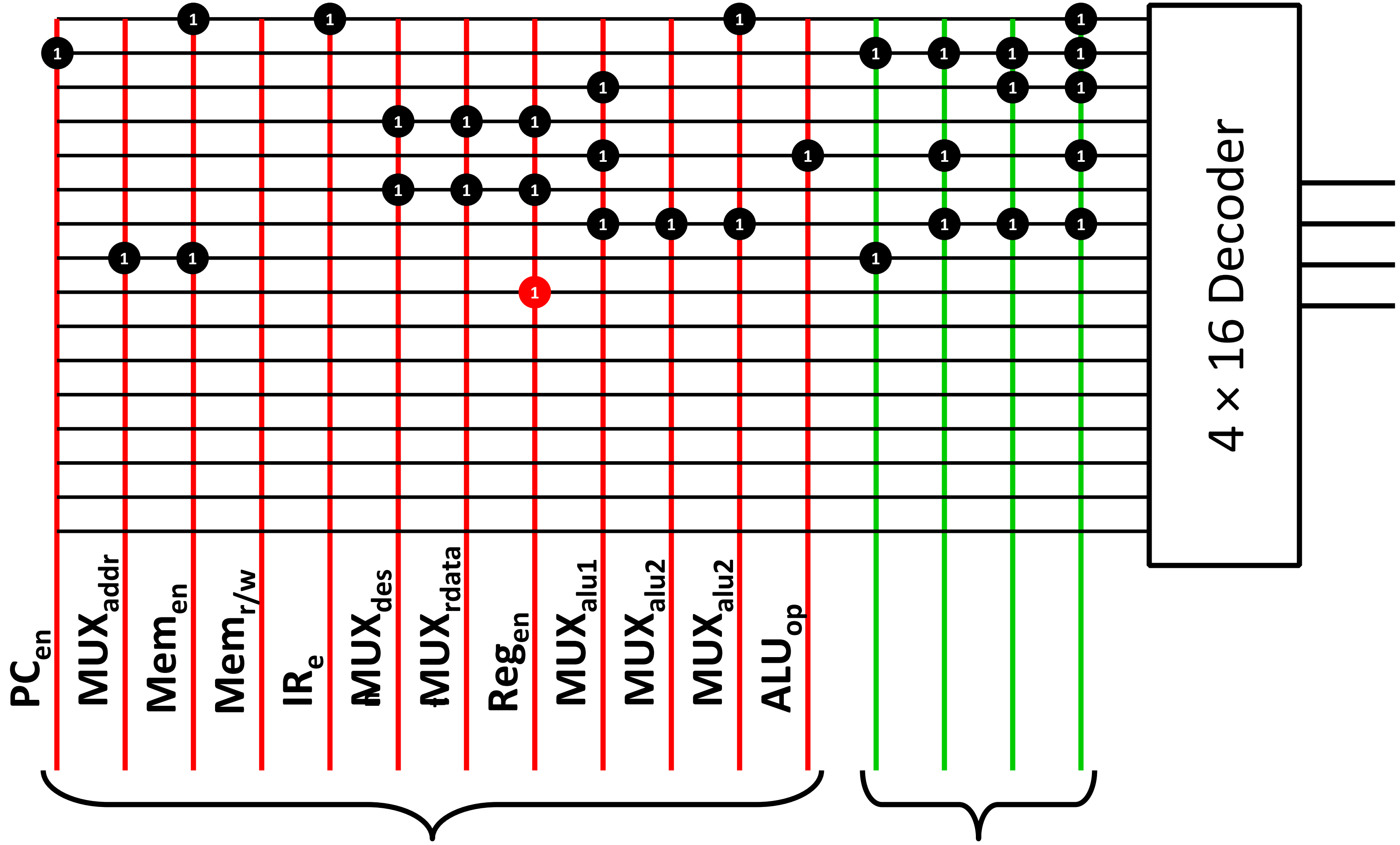
Next State

State 8: LW cycle 5

Write memory value to register file

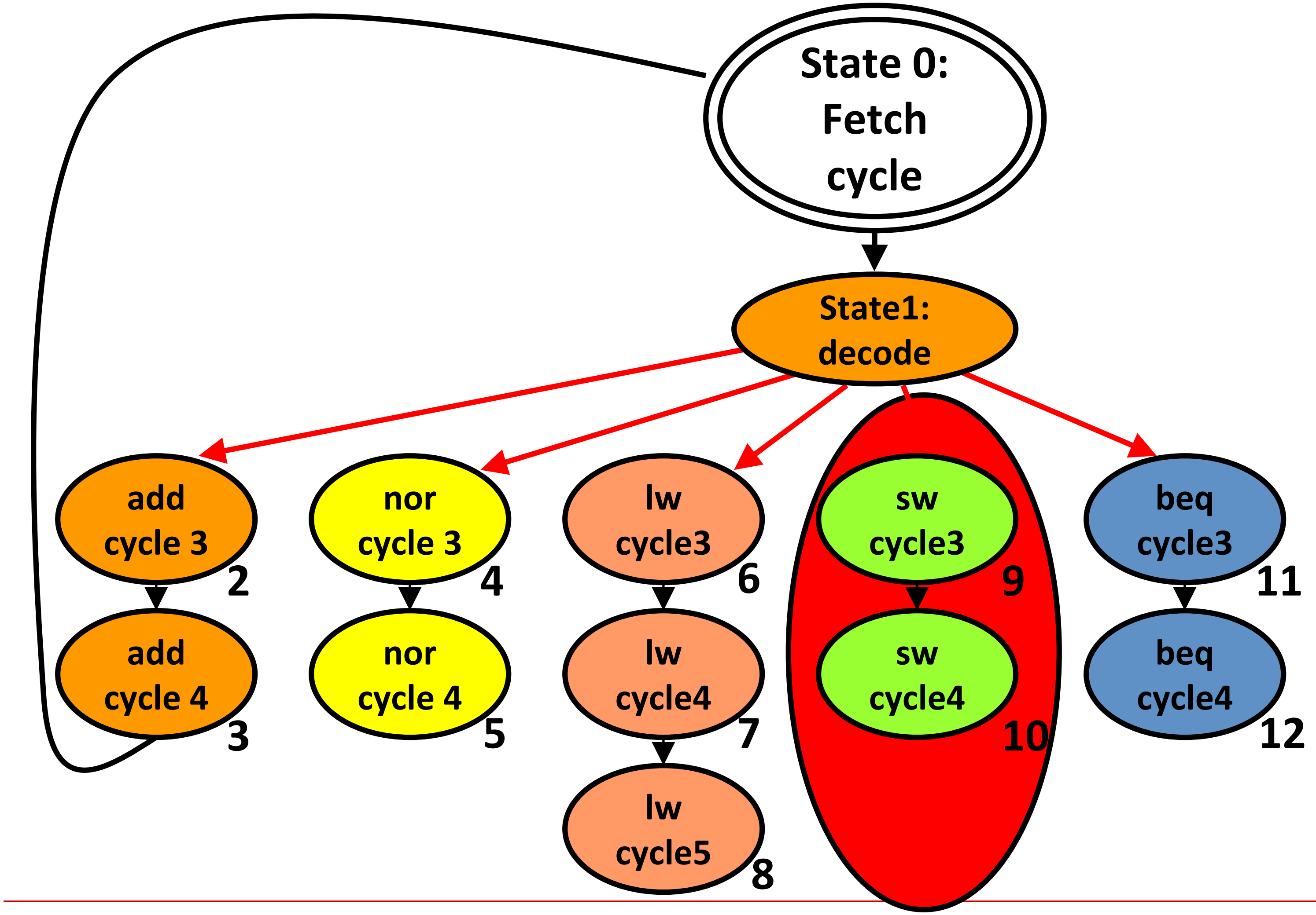


Control Rom (lw cycle 5)

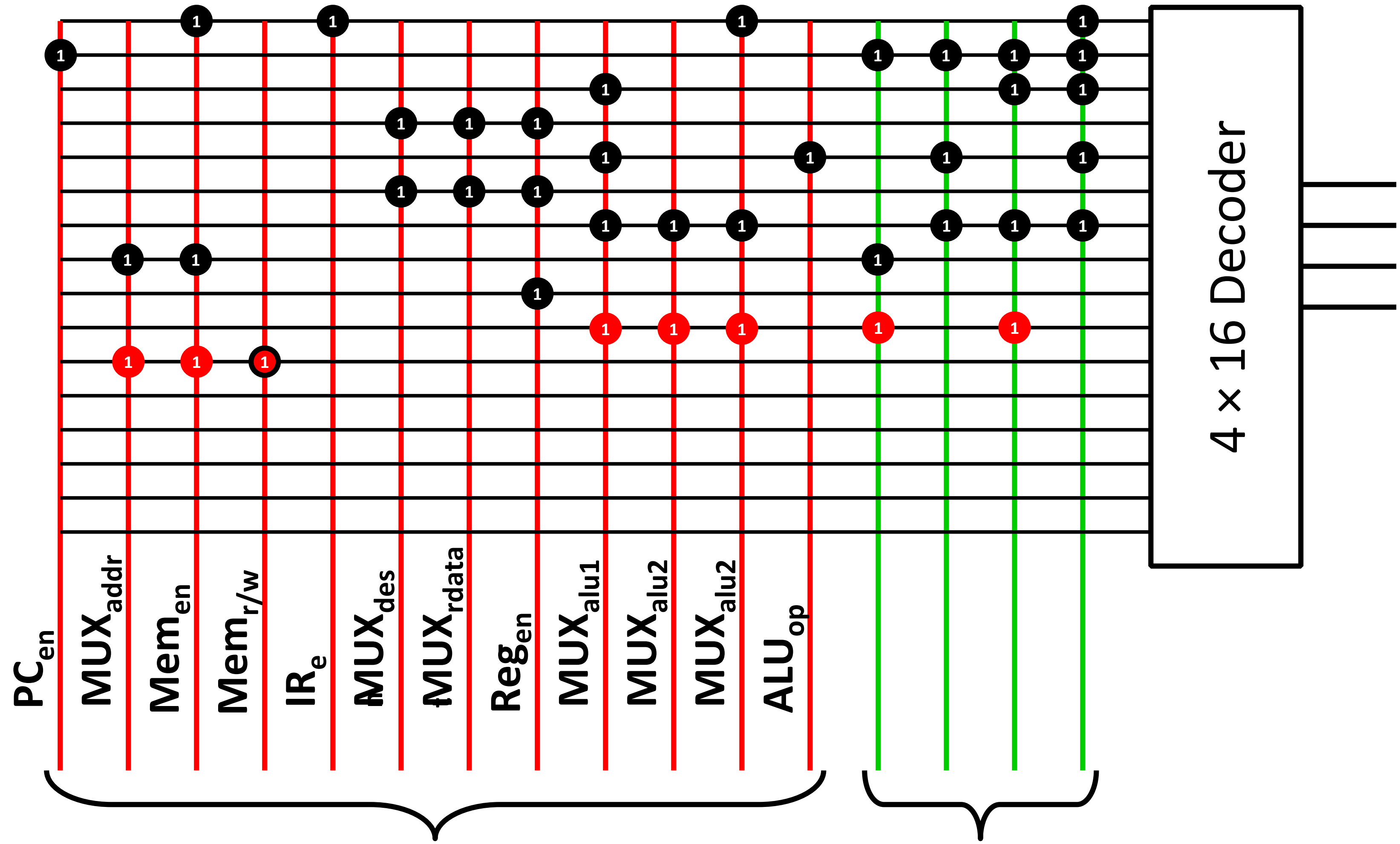


PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_e	MUX_{des}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			
							1								

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



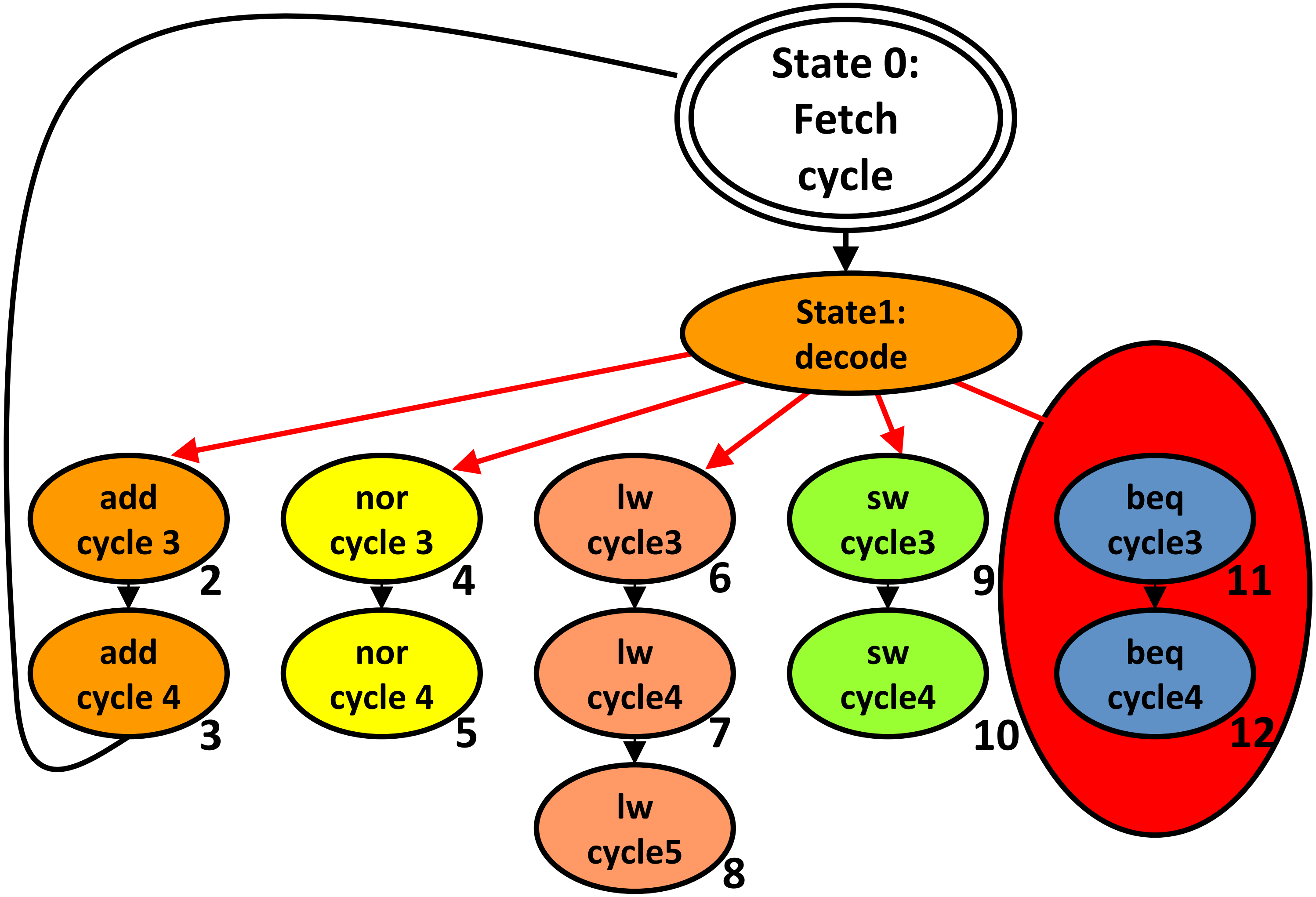
Control ROM (sw cycles 3 and 4)



PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_e	MUX_{des}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			
							1								
								1	1	1		1		1	
	1	1	1												

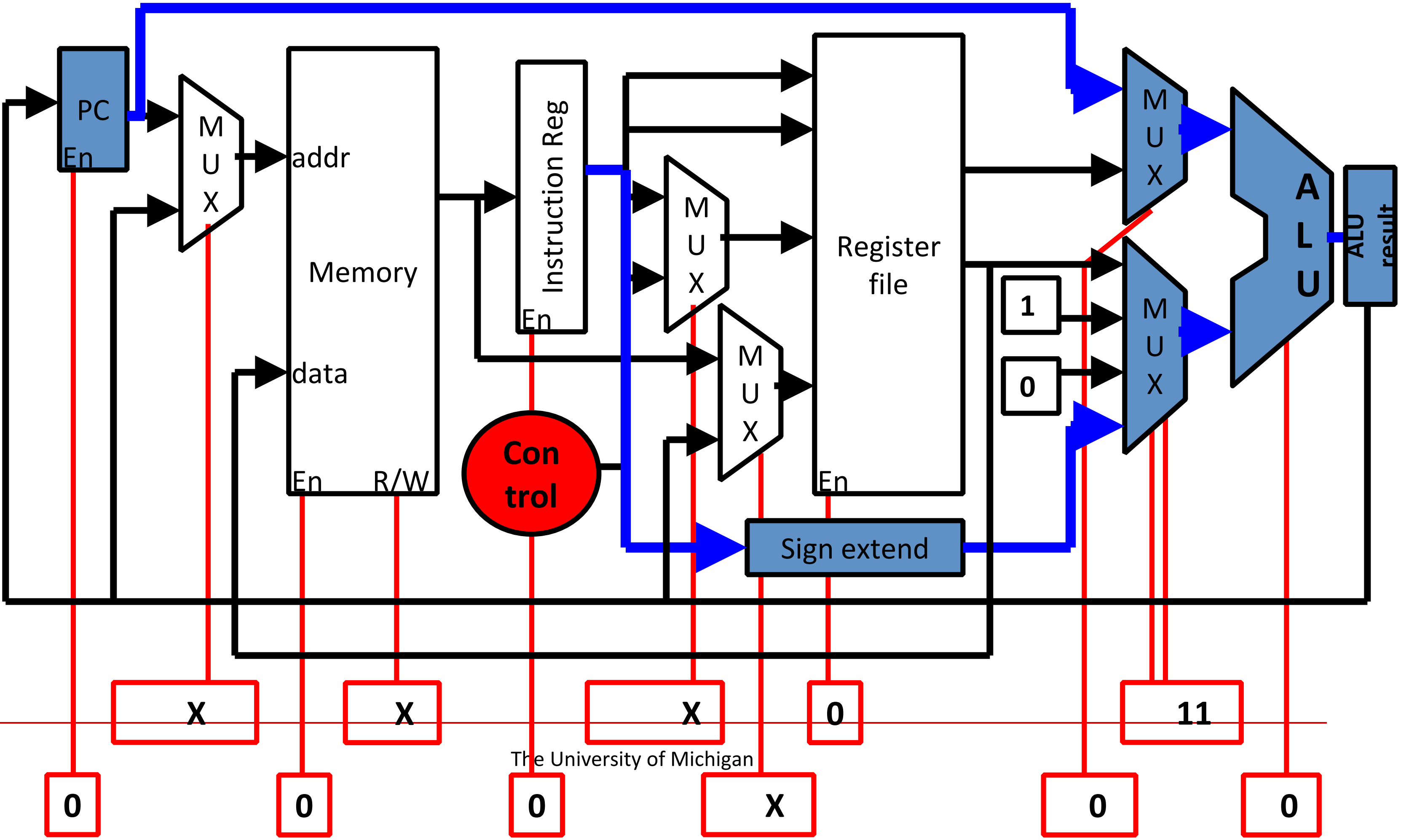
Output: Control Signals **Next State**

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

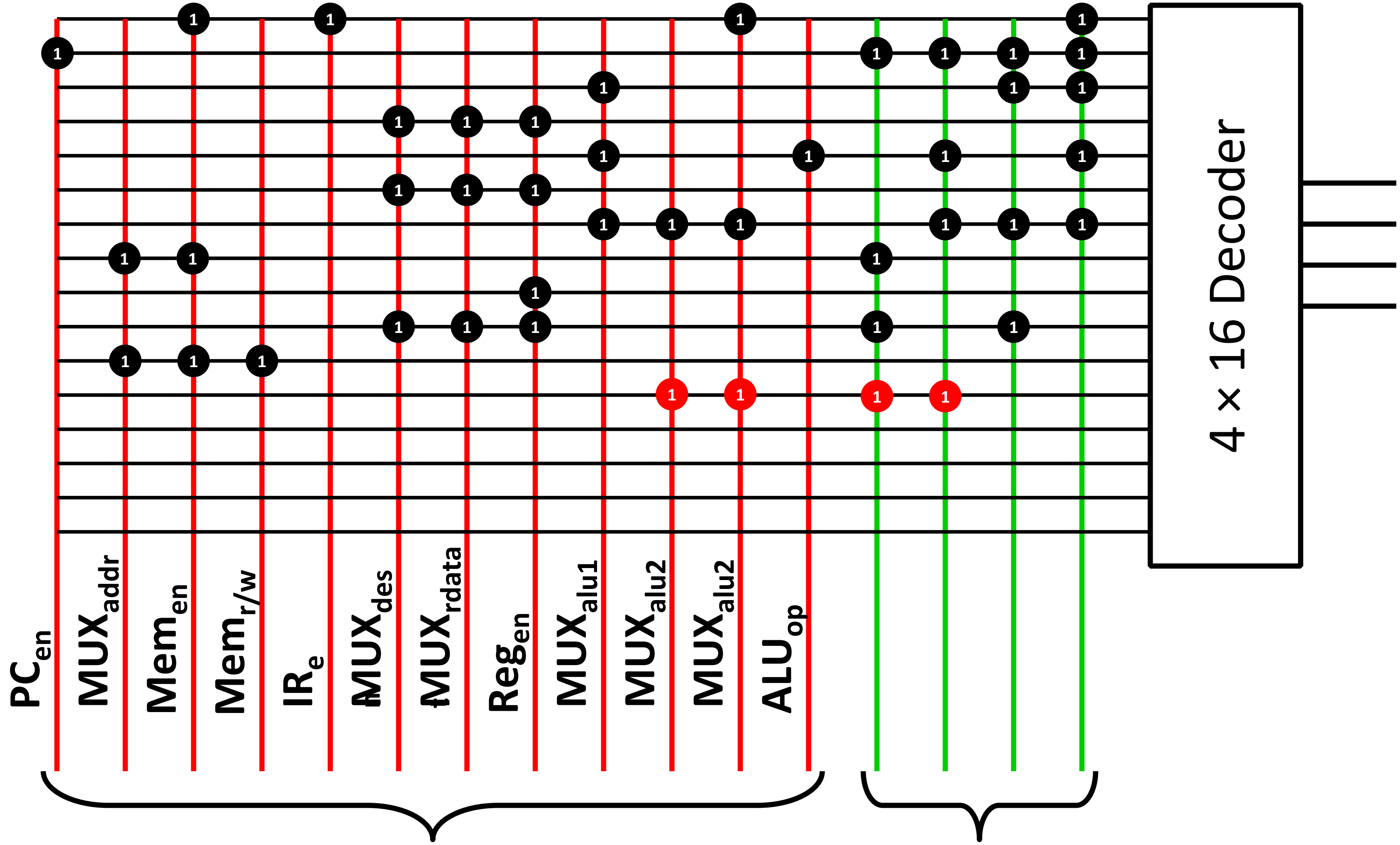


State 11: beq cycle 3

Calculate target address for branch



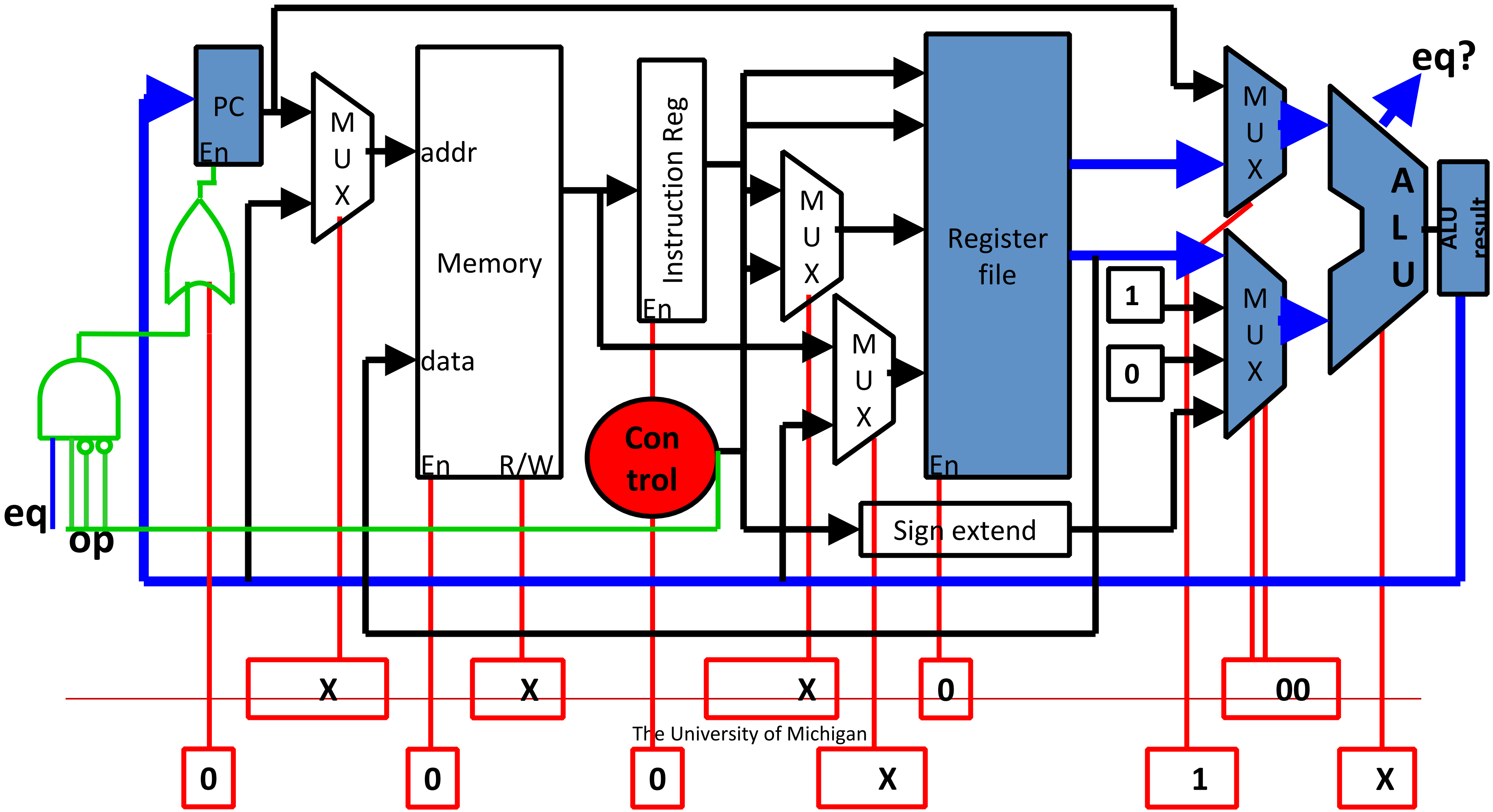
Control ROM (beq cycle 3)



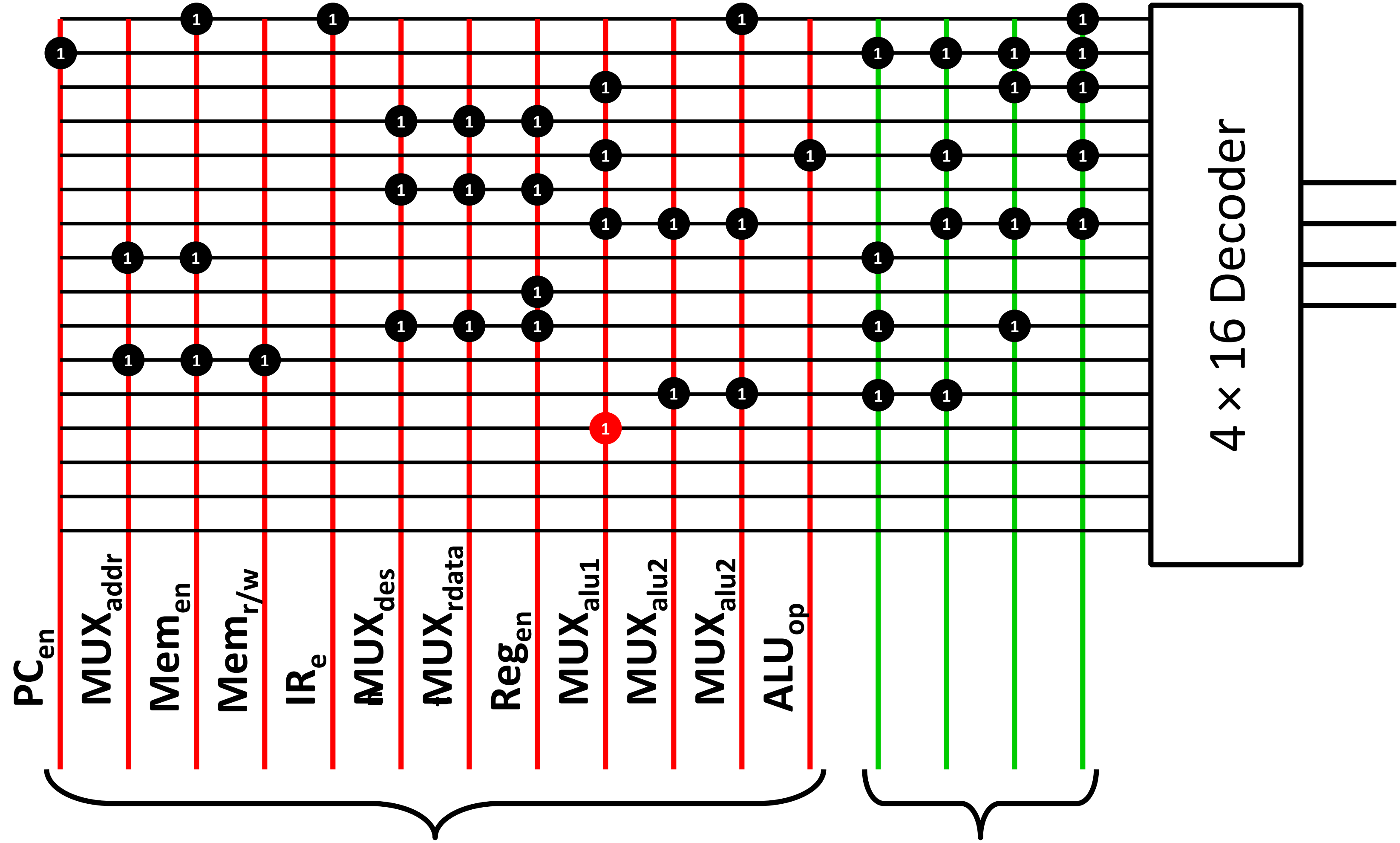
PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_e	MUX_{des}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			
							1								
					1	1	1					1		1	
	1	1	1												
									1	1		1	1		

State 12: beq cycle 4

Write target address into PC
if ($\text{data}_{\text{rega}}$ == $\text{data}_{\text{regb}}$)



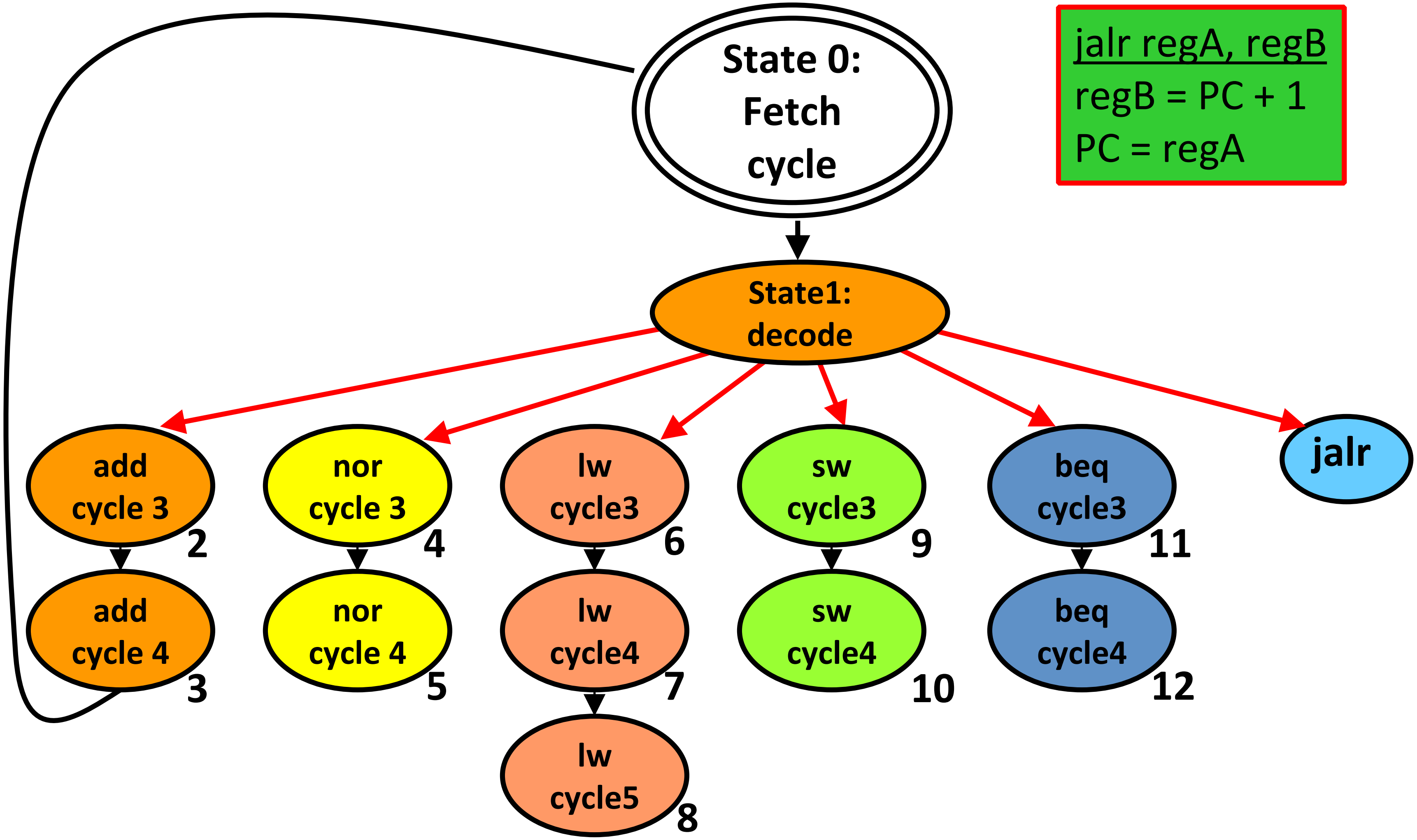
Control ROM (beq cycle 4)



Output: Control Signals

Next State

OK, what about the JALR instruction?



jalr regA, regB

regB = PC + 1

PC = regA

Single and Multicycle 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 multicycle datapath could achieve?

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

Single and Multicycle Performance

1 ns – Register 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 multicycle datapath could achieve?

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