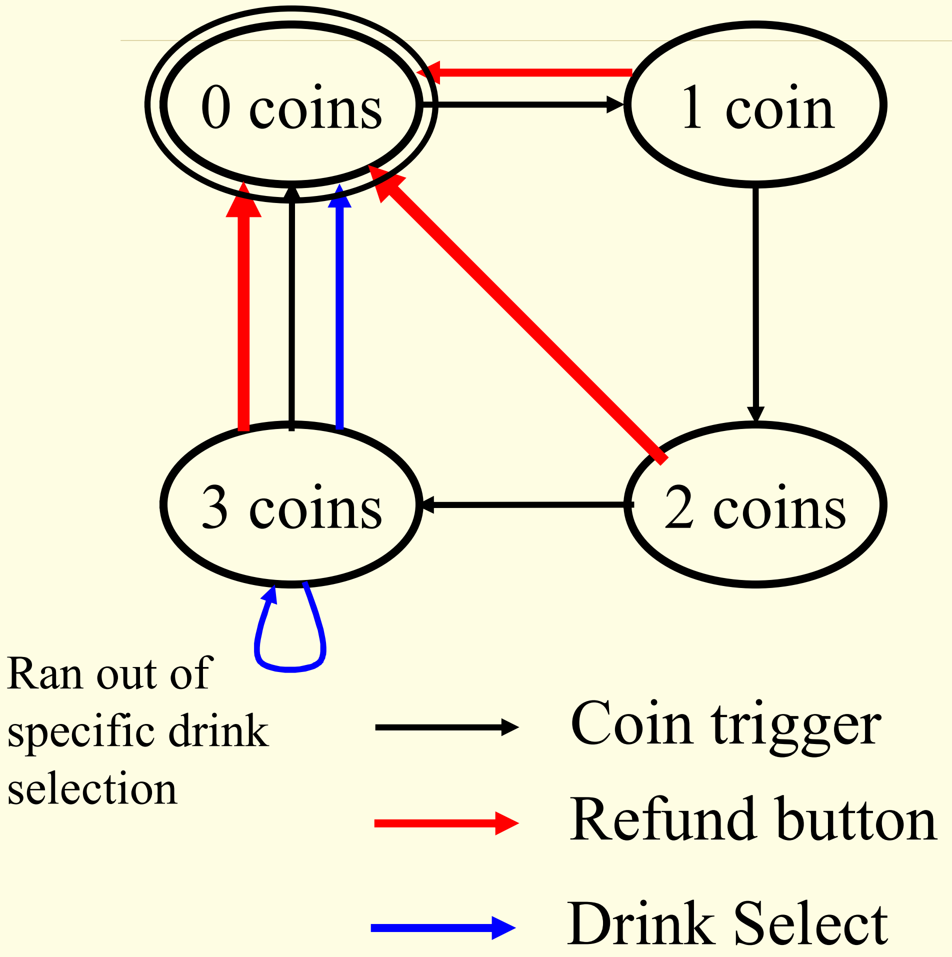
330: Computer Architecture

Basic Processor Design: Datapaths

Slides prepared by Dr. Gary Tyson.

Recap (FSM for Vending Machine)



0 coins

1 coin

2 coins

3 coins

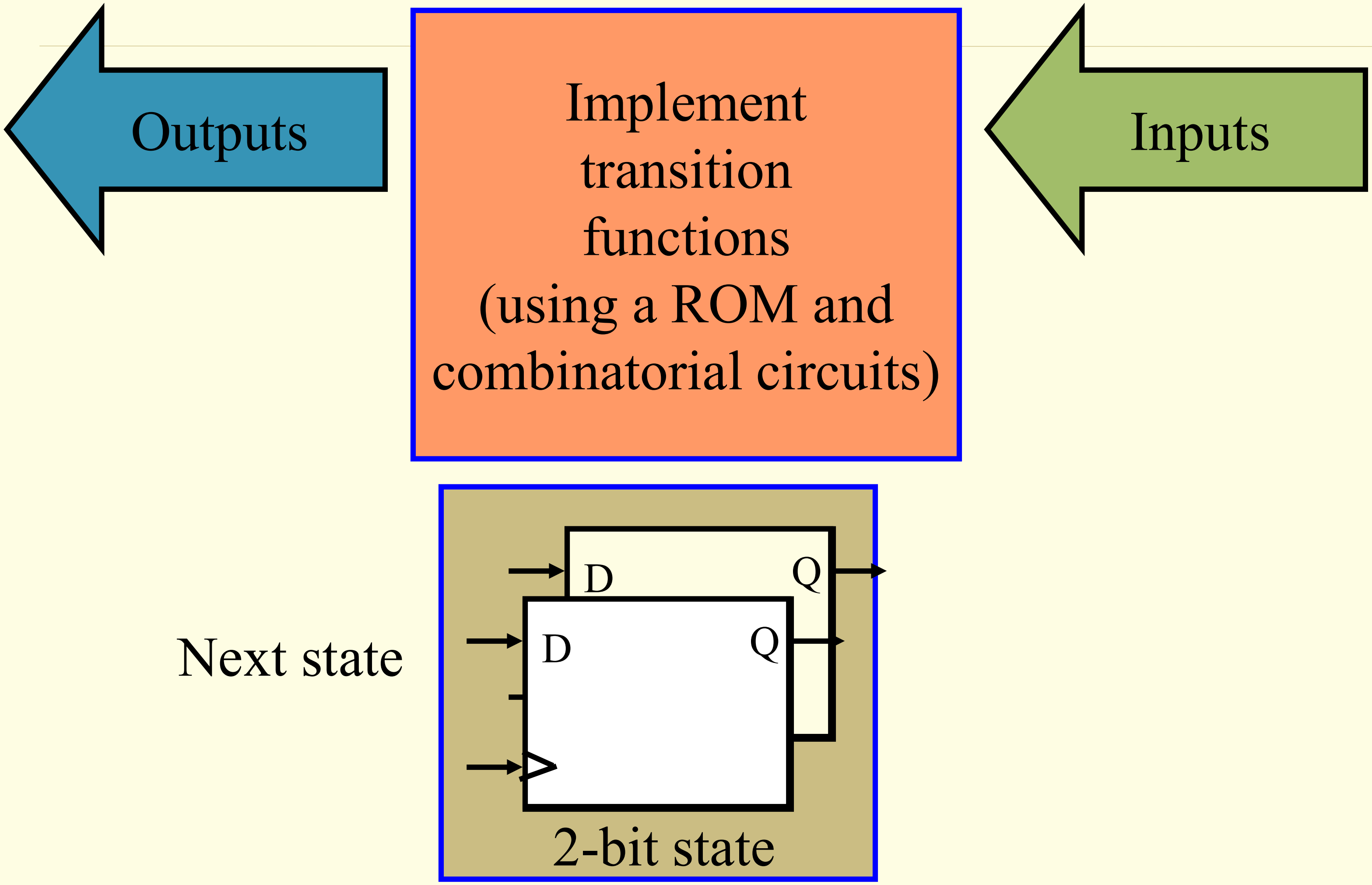
Ran out of specific drink selection

Coin trigger

Refund button

Drink Select

Recap (Implementing FSM)



Outputs

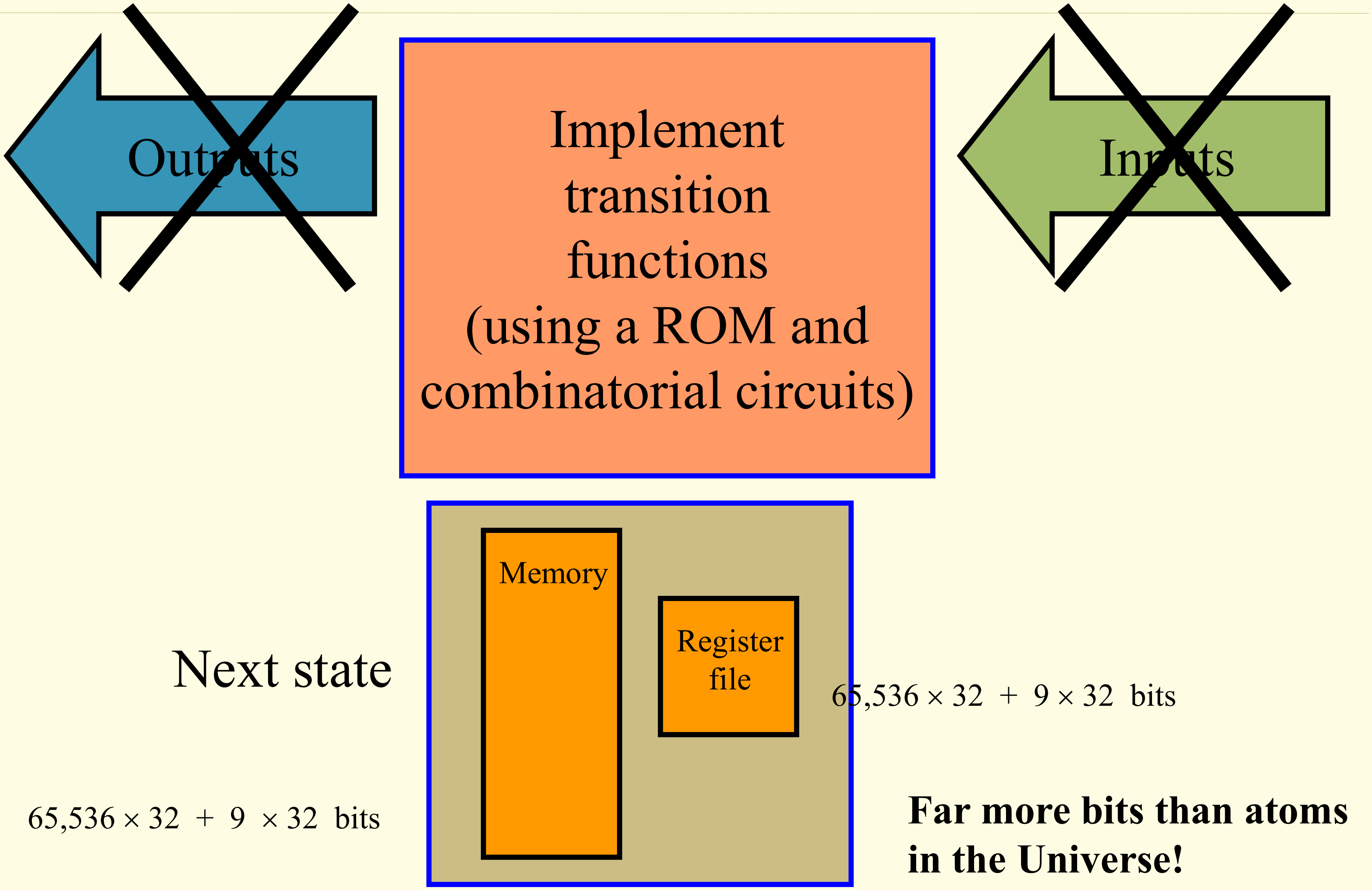
Implement transition functions (using a ROM and combinatorial circuits)

Inputs

Next state

2-bit state

A MIPS-32 Processor as FSM



~~Outputs~~

Implement transition functions (using a ROM and combinatorial circuits)

~~Inputs~~

Next state

Memory

Register file

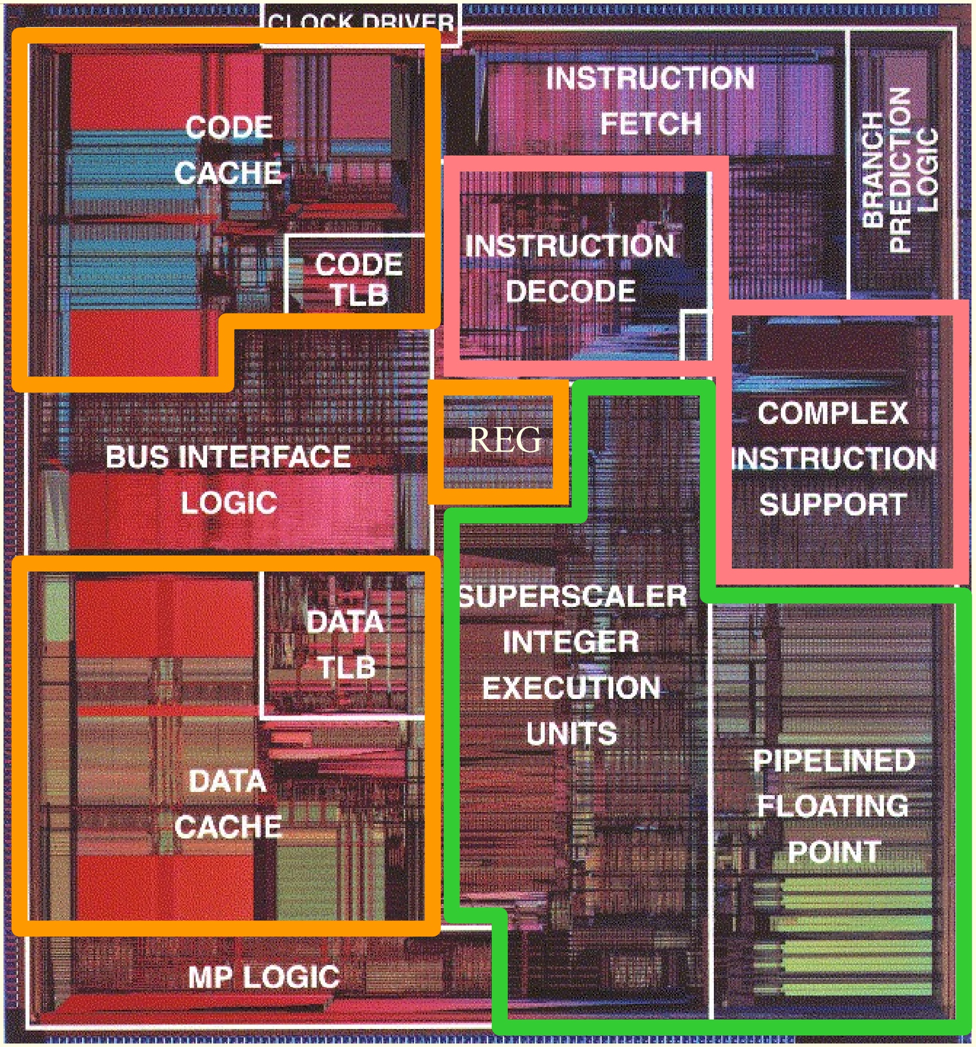
65,536 × 32 + 9 × 32 bits

65,536 × 32 + 9 × 32 bits

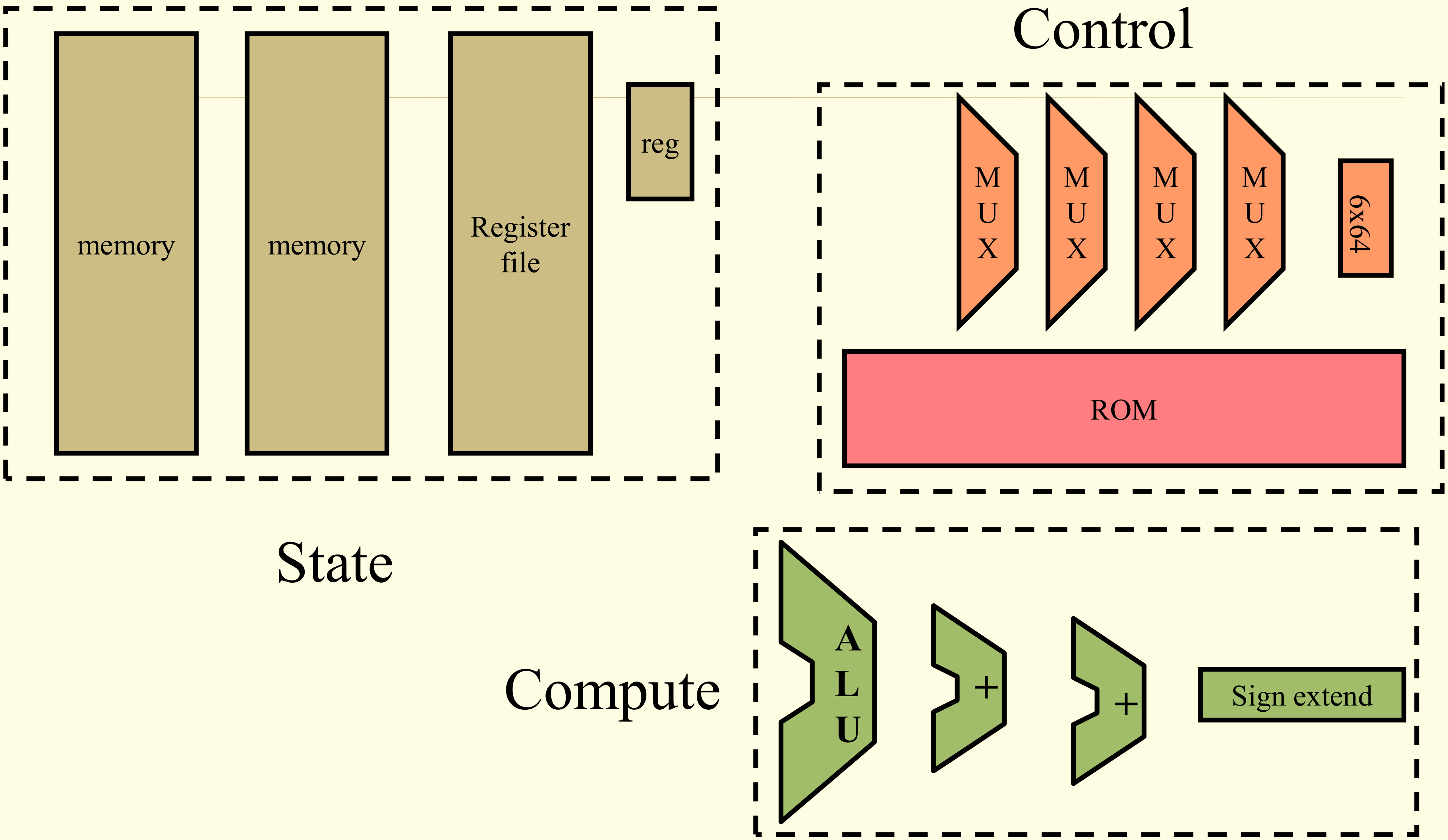
Far more bits than atoms in the Universe!

Pentium Processor Die

- State
  - Registers
  - Memory
- Control ROM
- Combinational logic (Compute)



Building Blocks for the MIPS-32



State

memory

memory

Register file

reg

Control

MUX

MUX

MUX

MUX

6x64

ROM

Compute

ALU

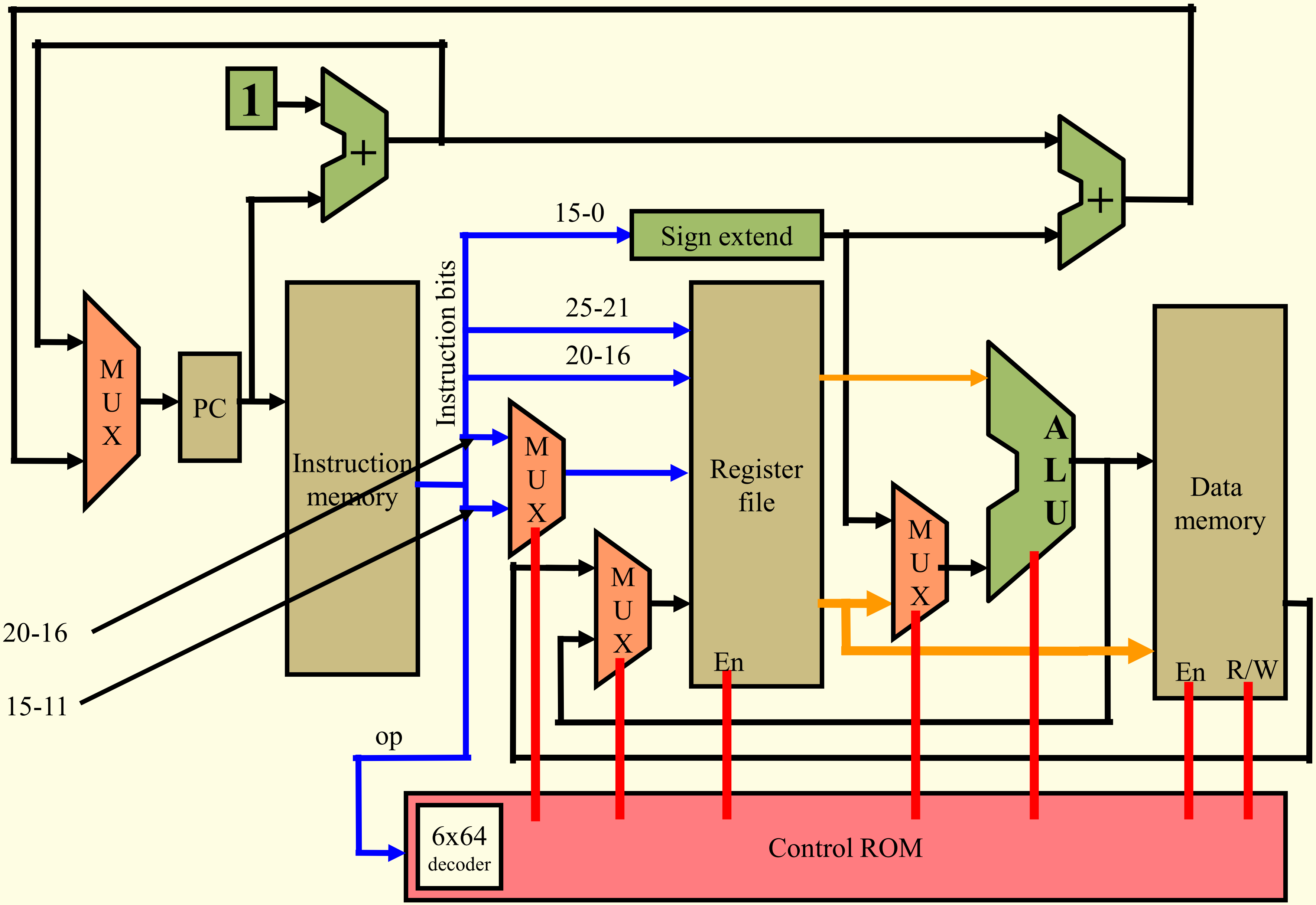
+

+

Sign extend

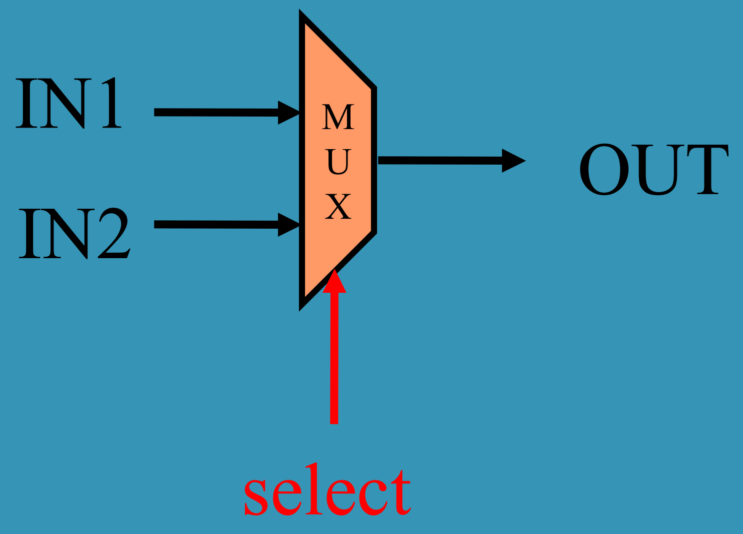
Here are the pieces, go build yourself a processor!

A MIPS-32 Datapath Implementation



Control Building Blocks (1)

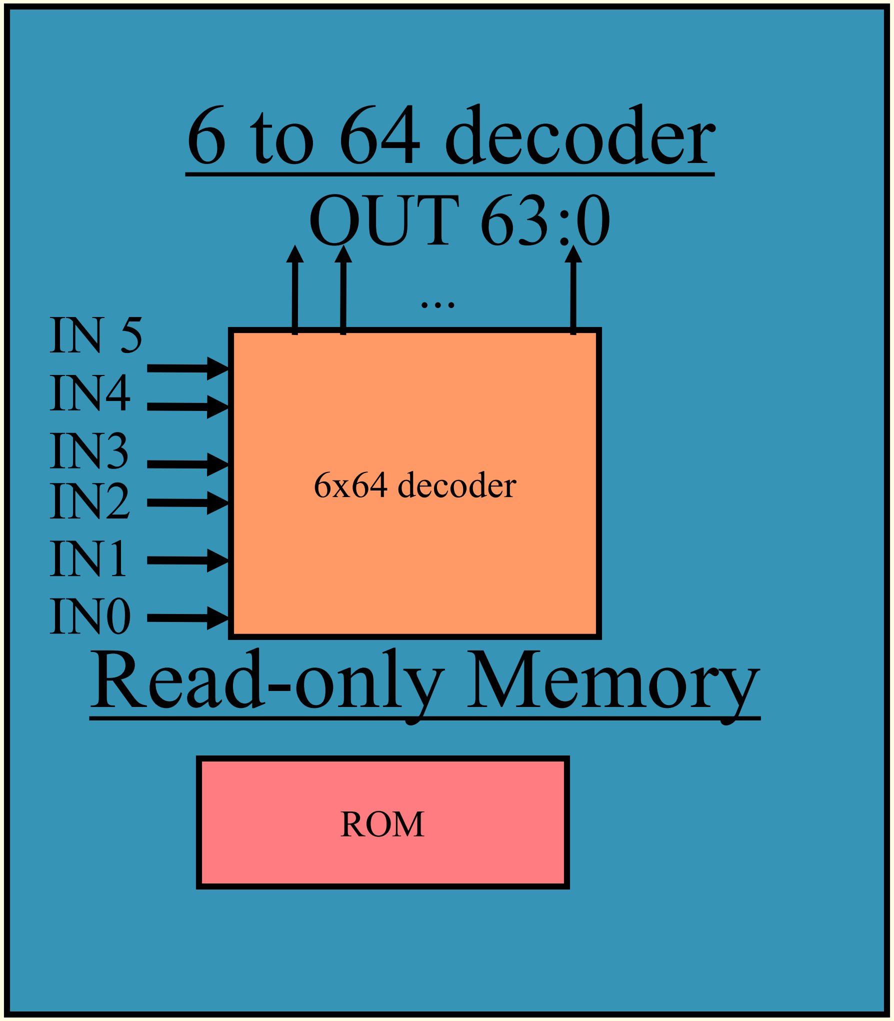
2 to 1 MUX



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

```
If (select == 0)
    OUT = IN1
Else
    OUT = IN2
```

Control Building Blocks (2)



6 to 64 decoder

Read-only Memory

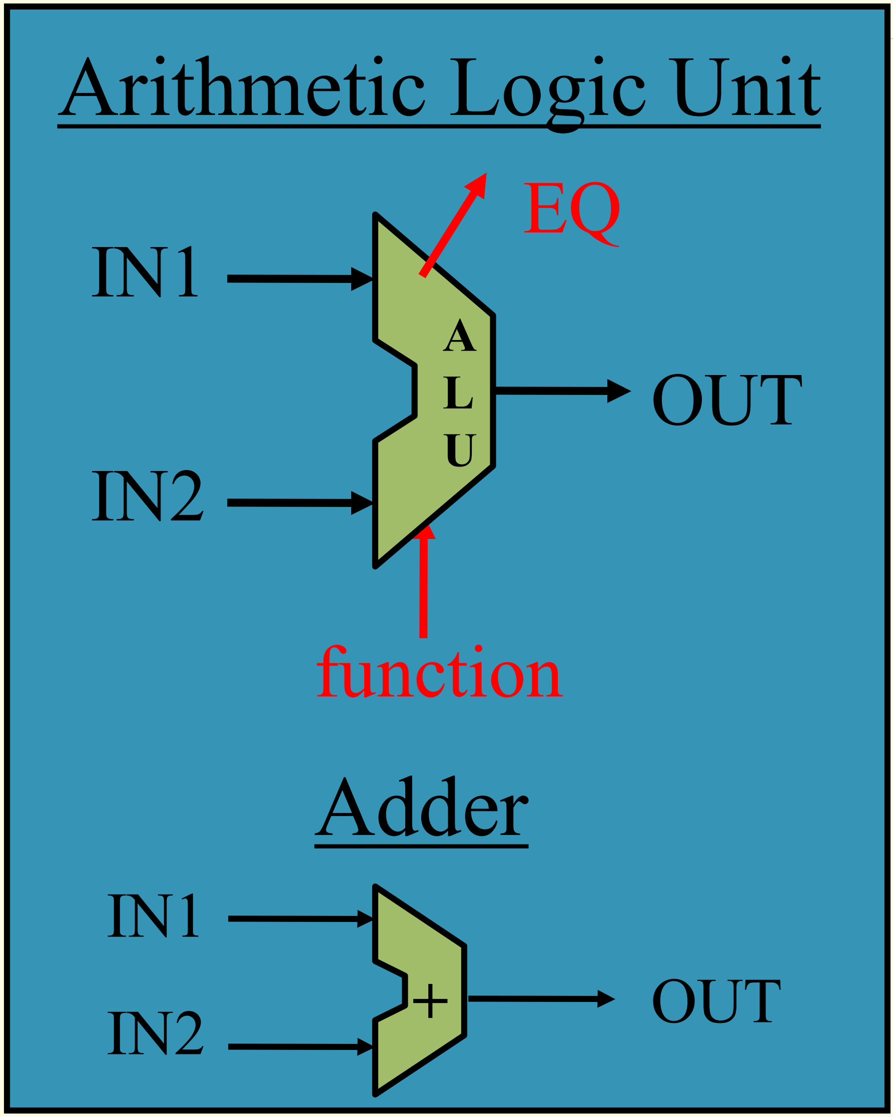
Decoder activates one of the output lines based on the input

IN	OUT
6543210	6543210
0000000	0000000
0000001	0000001
0000010	0000010
0000011	0000100

etc.

ROM just stores preset data in each location. Give address to access data

Compute Building Blocks (1)



Perform basic arithmetic functions

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

For MIPS32, f = add, nor, eq for other processors, there are many more functions

Simple ALU – Does only adds

inA	inB	outC

Compute Building Blocks (2)

Sign Extension Unit

IN

Sign extend

OUT

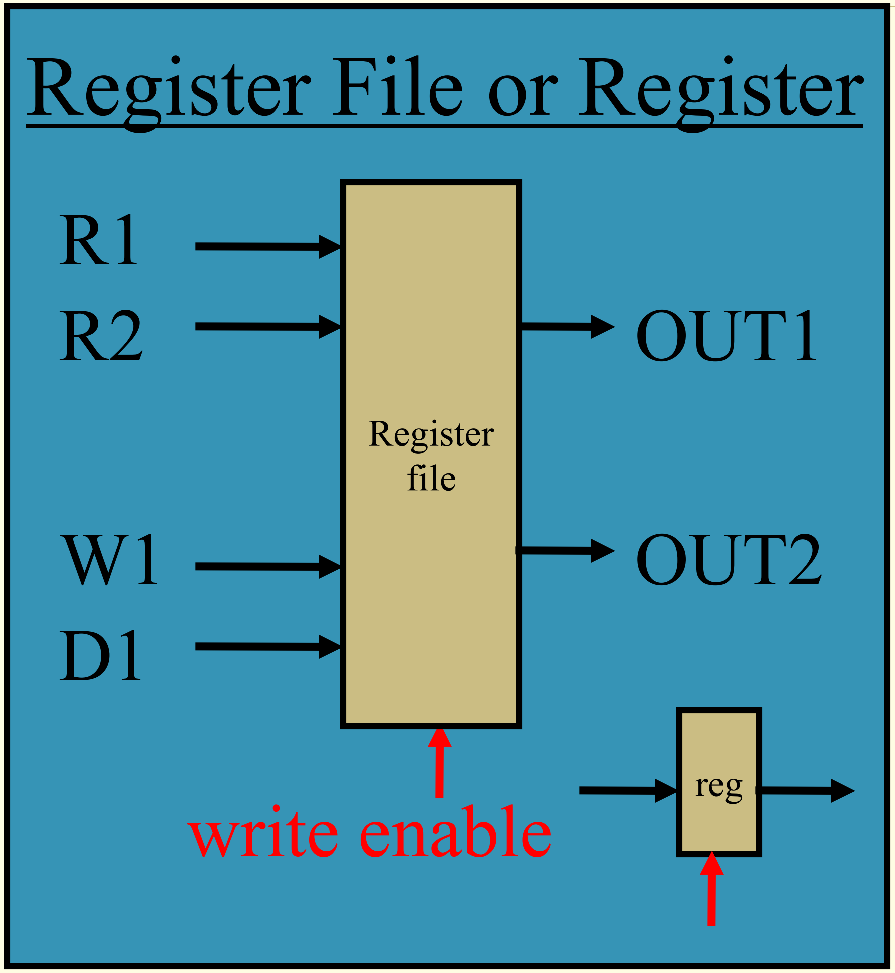
Sign extend input by replicating the MSB to width of output

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

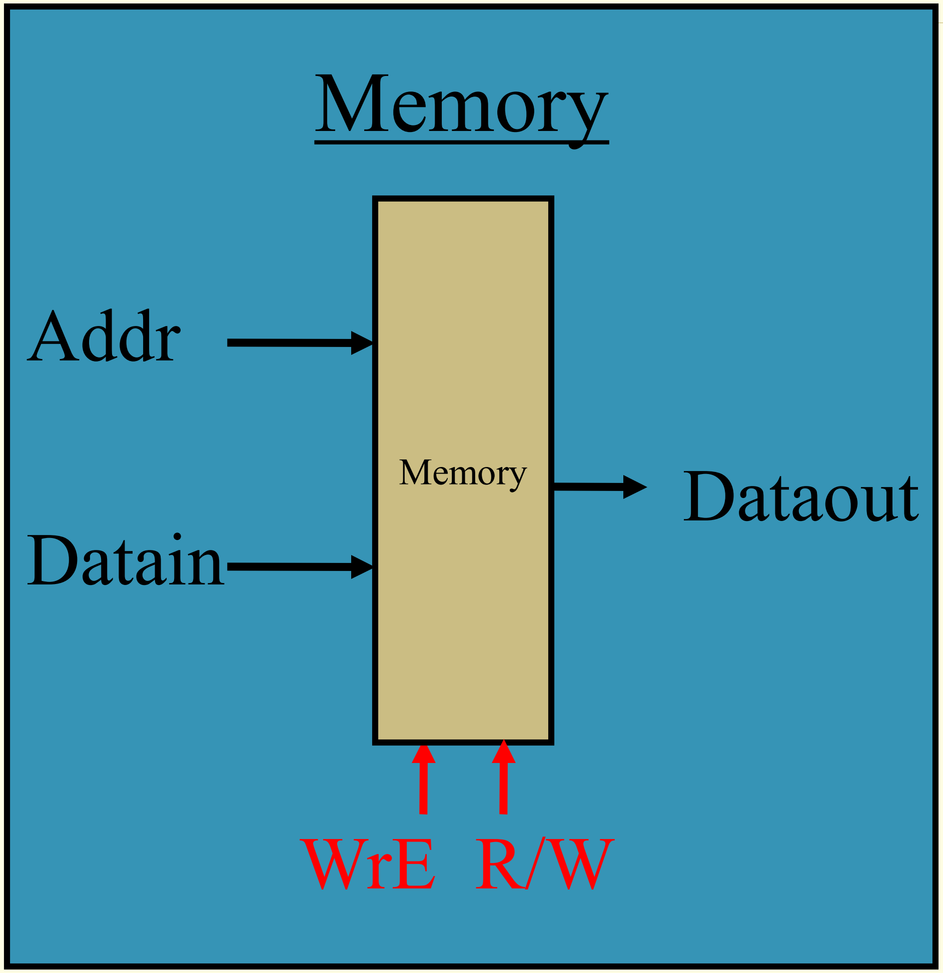
- n entries (MIPS32 = 32)
- r read ports (MIPS32 = 2)
- w write ports (MIPS32 = 1)

*Num r/w ports varies

* Ri specifies register number to read
* Wi specifies register number to write
* Di specifies data to write

How many bits are Ri and Wi in MIPS32?

State Building Blocks (2)



Slower storage structure to hold large amounts of stuff

Use 2 memories for MIPS32

- * Instructions
- * Data
- * 65536 total words

MIPS-32 processor

- Microprocessor without Interlocked Pipeline Stages
  - For programming projects
  - Simple!

MIPS-32 processor

R-type instructions

opcode	rs	rt	rd	sa	func
31-26	25-21	20-16	15 – 11	10-6	5-0

Name	Exec	Func
add	rd, rs, rt	100000
nor	rd, rs, rt	100111

Opcode is always 00000!
https://www.d.umn.edu/~gshute/mips/rtype.xhtml

MIPS-32 R-Type

add	rd, rs, rt	100000
addu	rd, rs, rt	100001
and	rd, rs, rt	100100
break		001101
div	rs, rt	011010
divu	rs, rt	011011
jalr	rd, rs	001001
jr	rs	001000
mfhi	rd	010000
mflo	rd	010010
mthi	rs	010001
mtlo	rs	010011
mult	rs, rt	011000
multu	rs, rt	011001
nor	rd, rs, rt	100111
or	rd, rs, rt	100101
sll	rd, rt, sa	000000
sllv	rd, rt, rs	000100
slt	rd, rs, rt	101010
sltu	rd, rs, rt	101011
sra	rd, rt, sa	000011
srav	rd, rt, rs	000111
srl	rd, rt, sa	000010
srlv	rd, rt, rs	000110
sub	rd, rs, rt	100010
subu	rd, rs, rt	100011
syscall		001100
xor	rd, rs, rt	100110

MIPS-32 processor

I-type instructions

opcode	rs	rt	offset
31-26	25-21	20-16	15 - 0

Name	Exec	opcode	
lw	rt, imm(rs)	100011	
sw	rt, imm(rs)	101011	–
beq	rs, rt, label	000100	

https://www.d.umn.edu/~gshute/mips/itype.xhtml

MIPS-32 processor

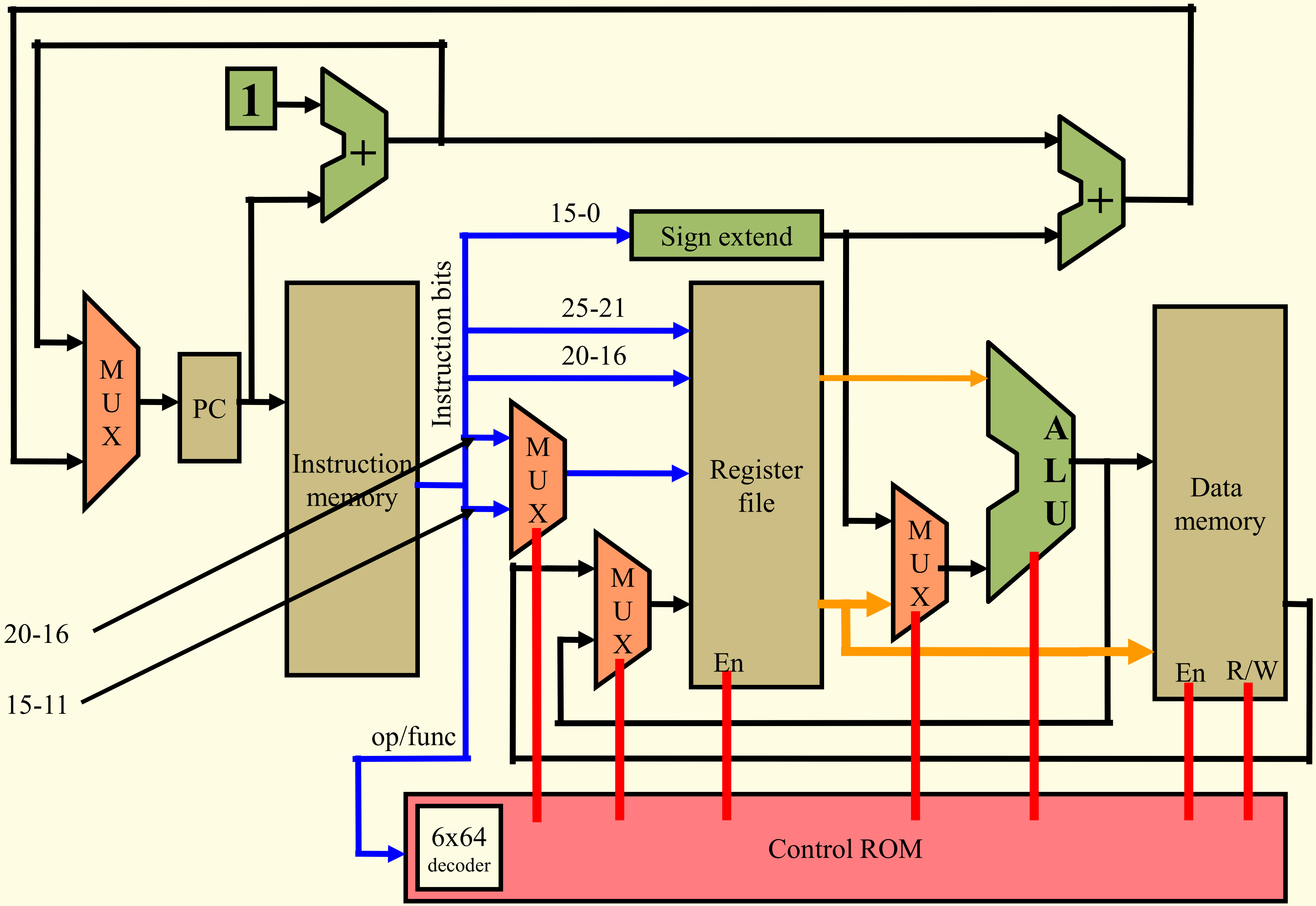
j-type instructions

opcode	offset
31-26	25 - 0

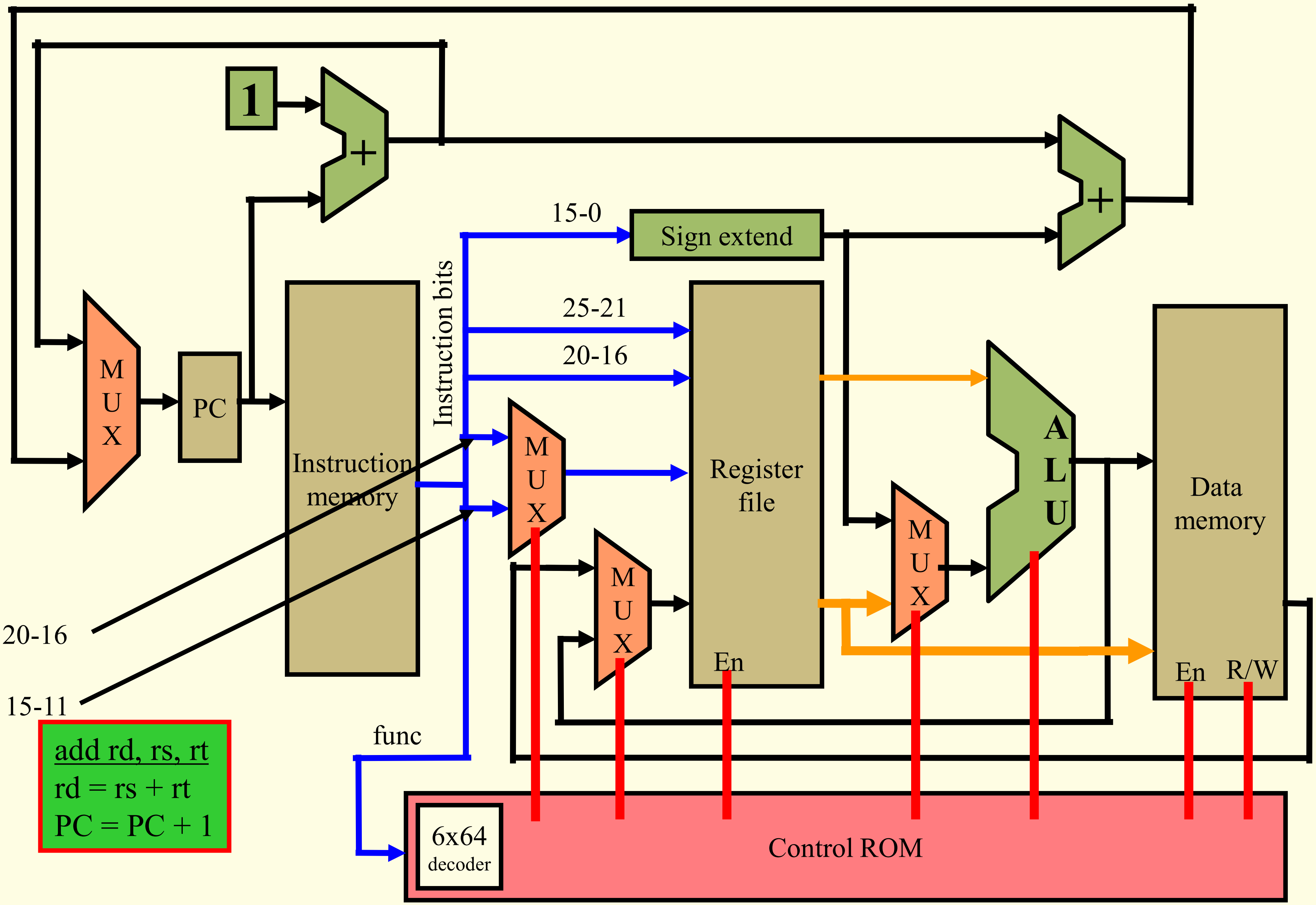
Name	Exec	opcode	
j	label	000010	coded address of label

https://www.d.umn.edu/~gshute/mips/jtype.xhtml

MIPS-32 Datapath Implementation

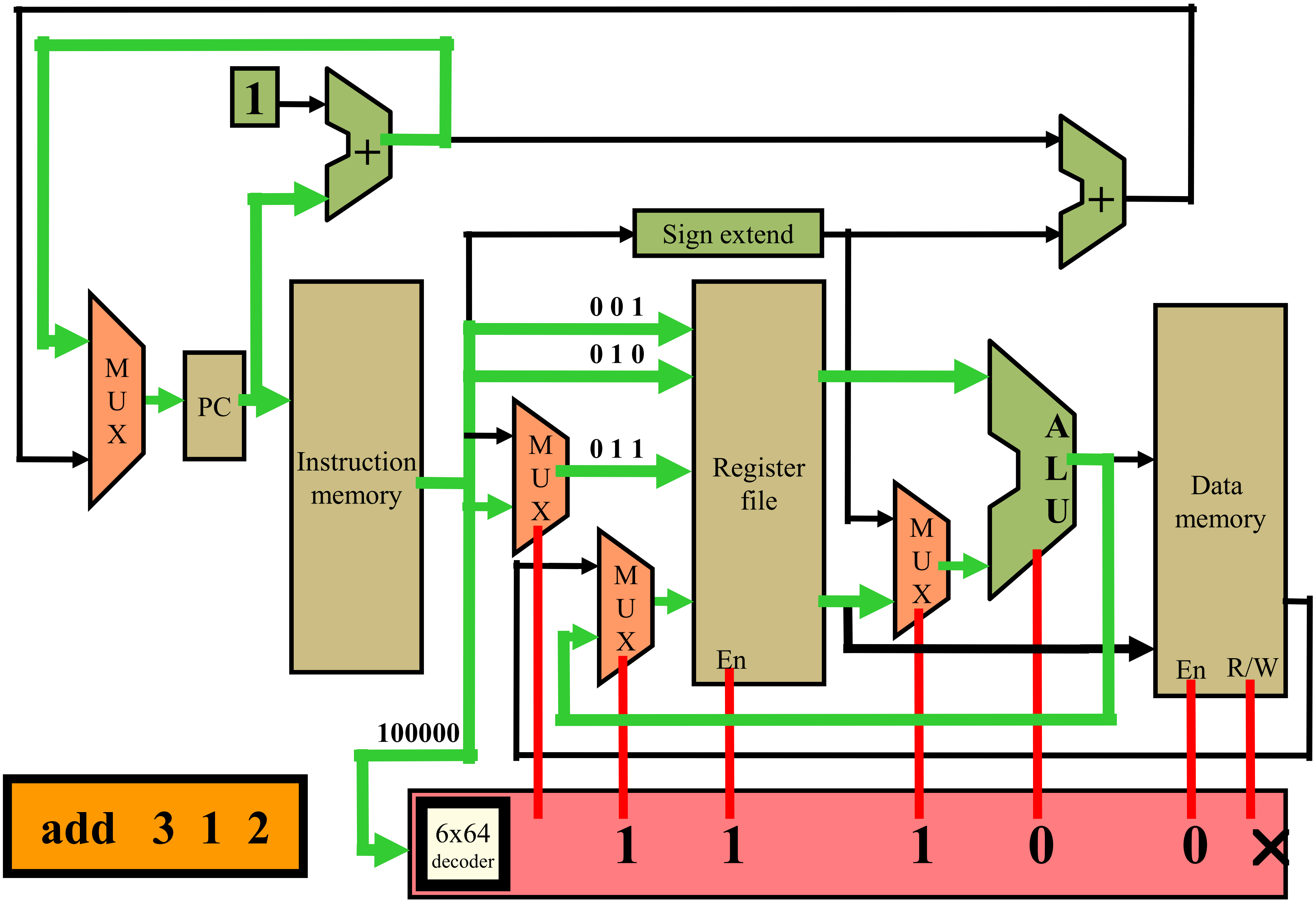


Executing an ADD Instruction



<u>add rd, rs, rt</u>
rd = rs + rt
PC = PC + 1

Executing an ADD instruction on this MIPS32 datapath

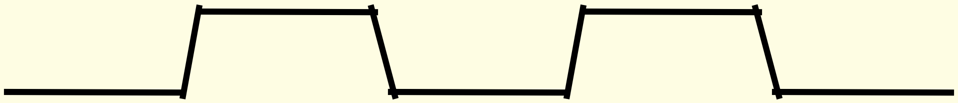


Adding a clock to the mix

- We can design more interesting circuits if we have a clock signal.
- The use of a clock enables a sequential circuit to ***predictably*** change state (and store information).
- A clock signal alternates between 0 and 1 states at a fixed frequency (e.g. 100MHz)
- What should the clock frequency be?

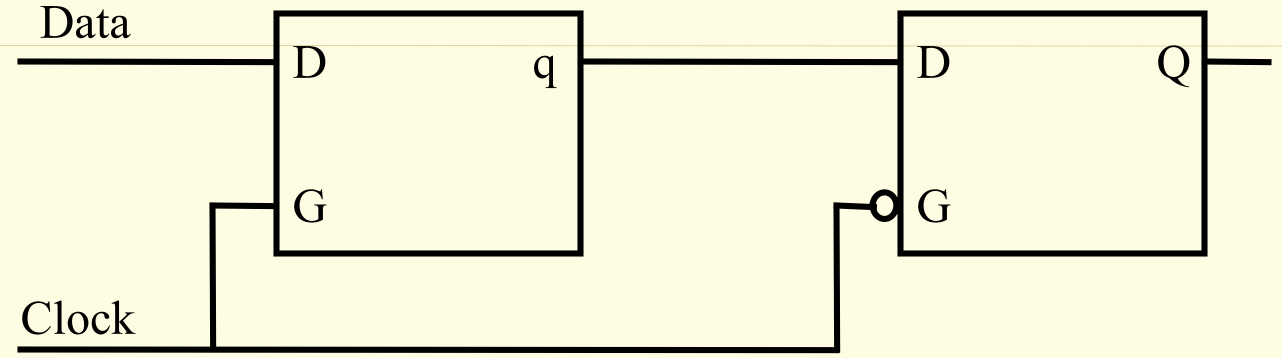
Clocks

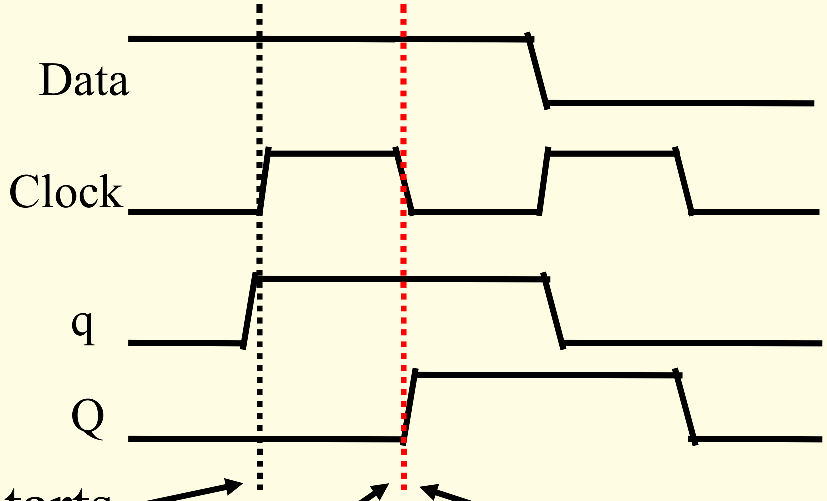
- Clock signal
  - Periodic pulse
  - Generated using oscillating crystal
  - Distributed throughout chip using clock distribution net

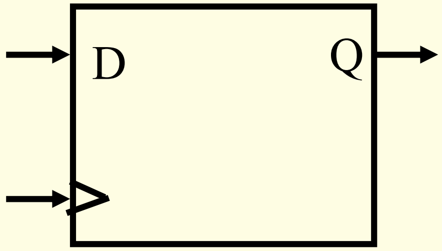


- With clock signals we can create a new class of circuits called ***sequential***
  - Output determined by inputs & **previous** state

Edge Triggered D Flip-flop







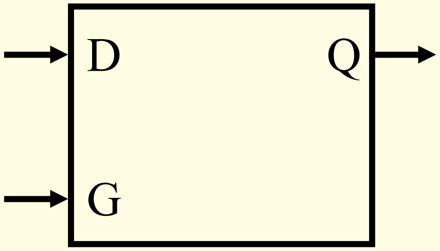
Latching starts

Latching edge

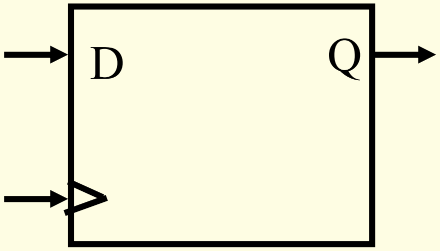
Data “sealed” inside flip-flop

Data should remain valid and stable during latching!

Why edge-triggered flip-flops?



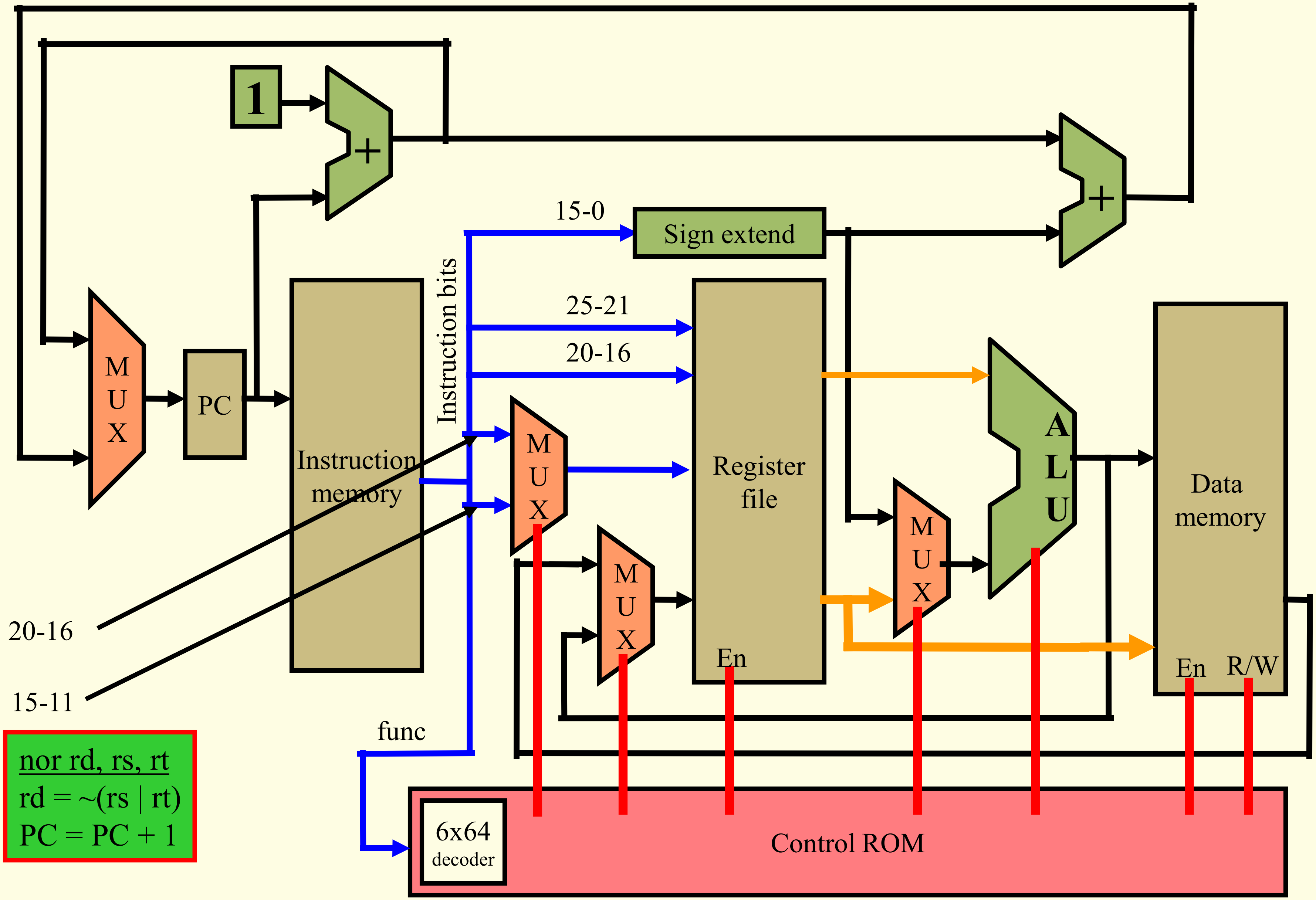
IS LIKE A TOLL-GATE



IS LIKE AN AIR-LOCK

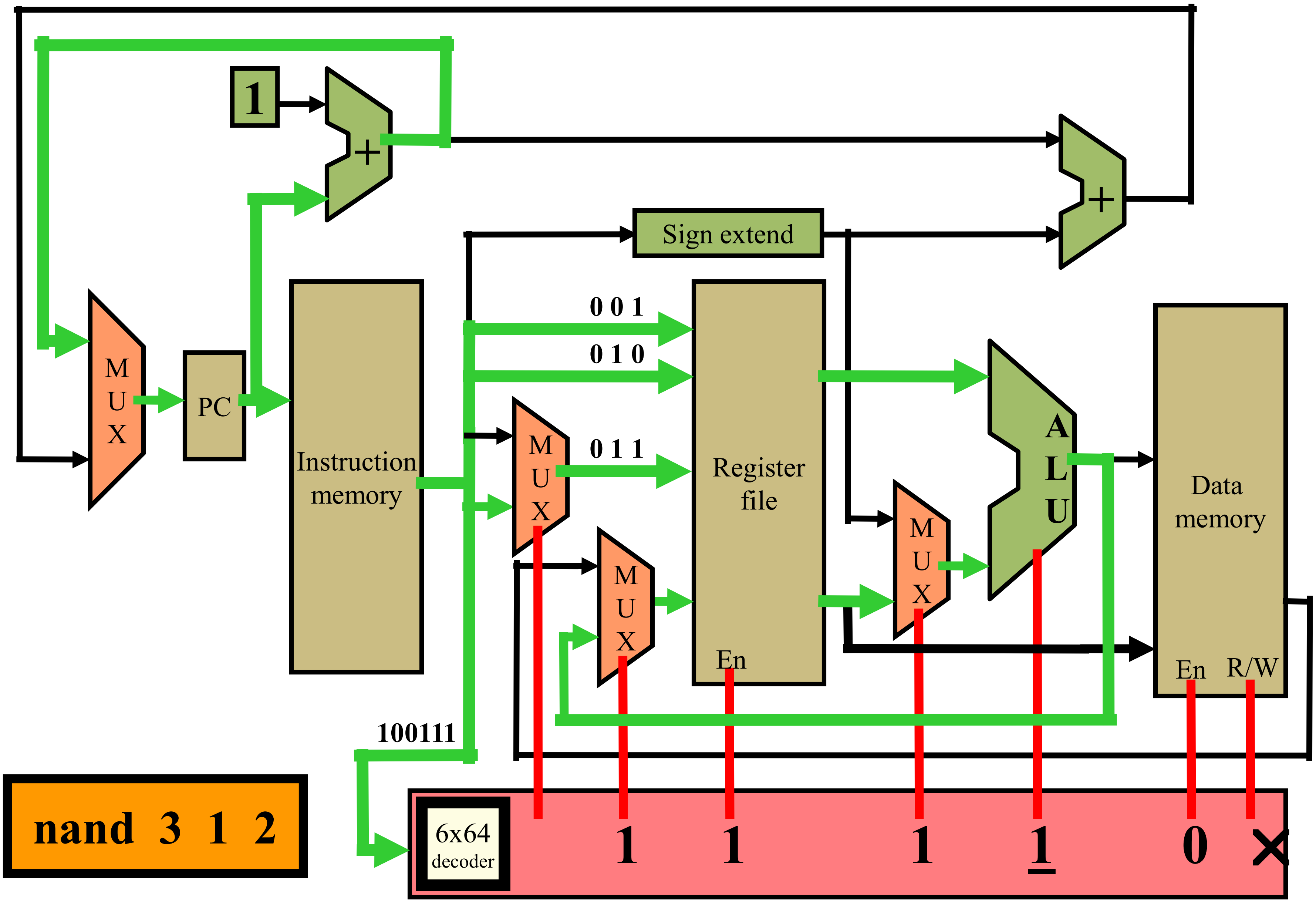
In edge-triggered flip-flops, the latching edge provides convenient abstraction of “instantaneous” change of state.

Executing an NOR Instruction

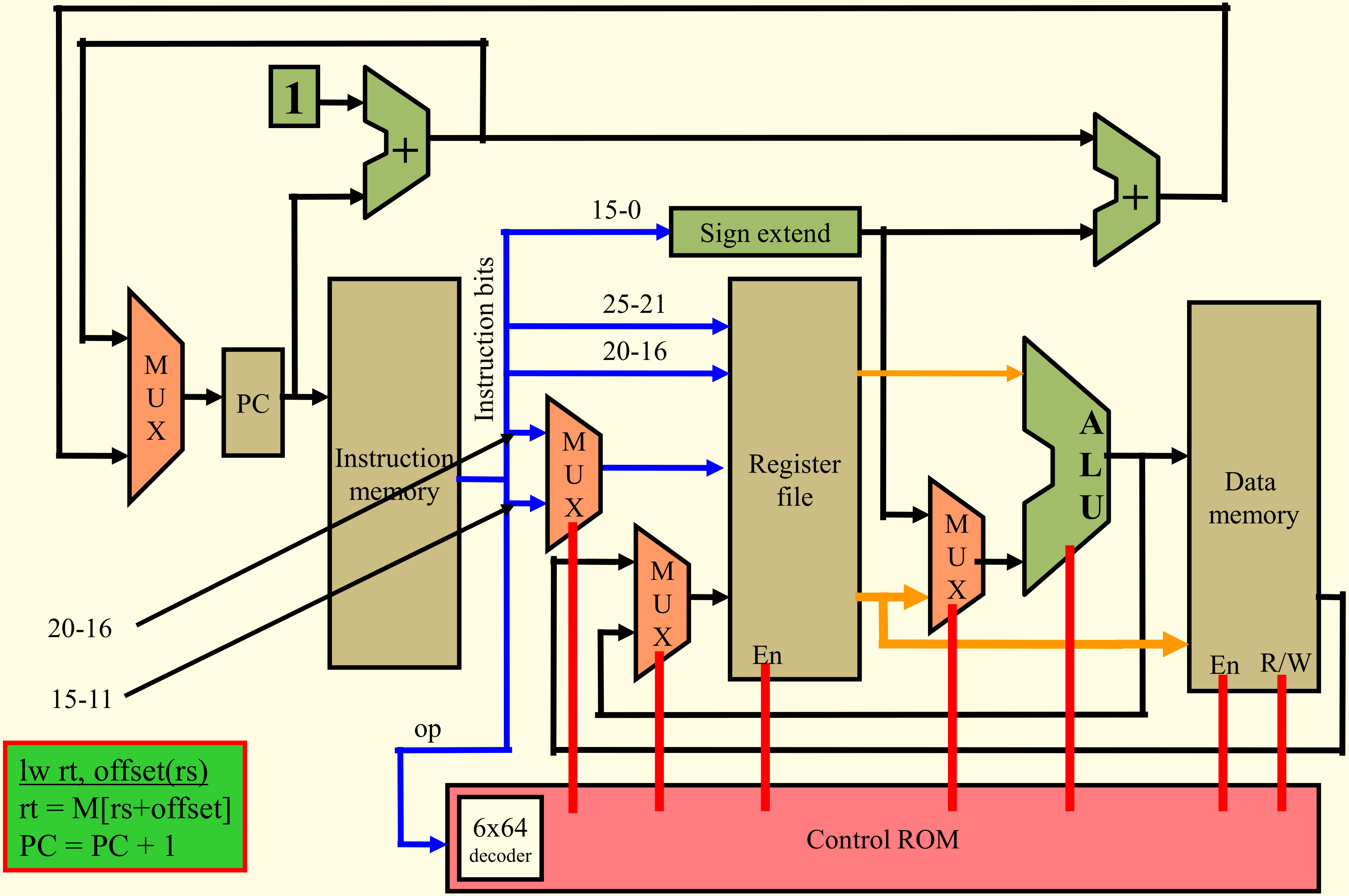


nor rd, rs, rt
rd = ~(rs | rt)
PC = PC + 1

Executing a NOR instruction on this MIPS-32 datapath

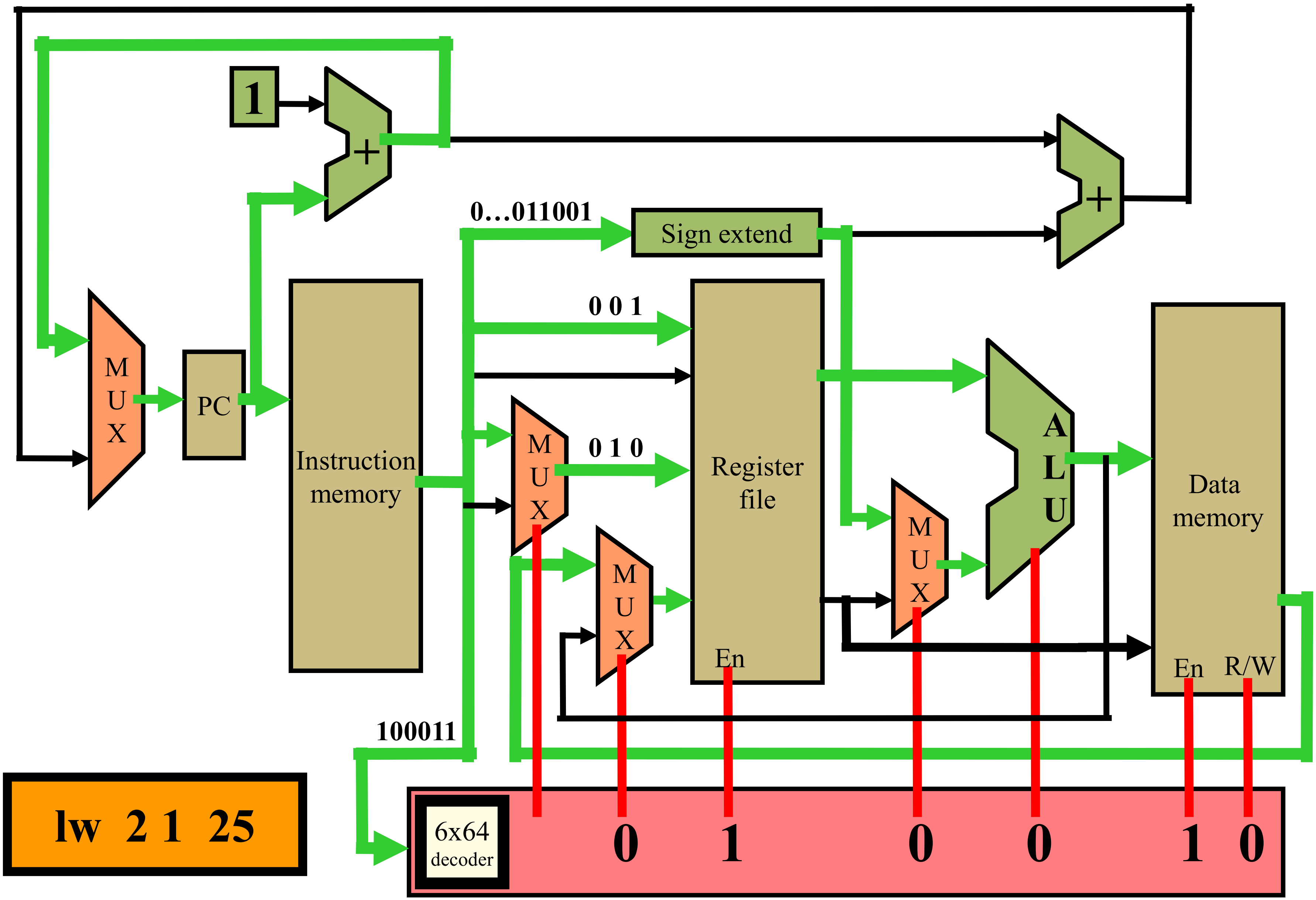


Executing an LW Instruction

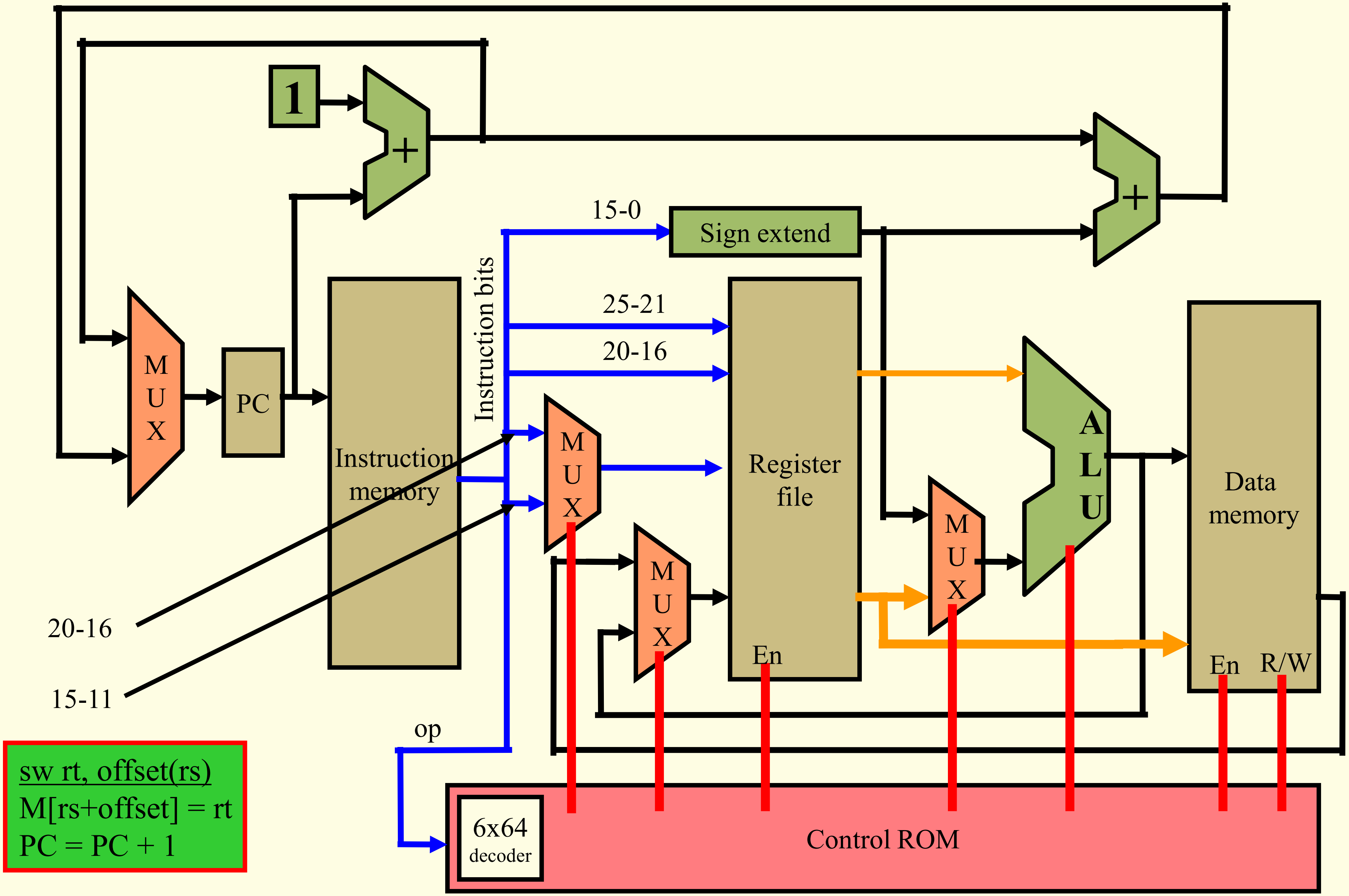


lw rt, offset(rs)
rt = M[rs+offset]
PC = PC + 1

Executing a LW instruction on this MIPS-32 datapath

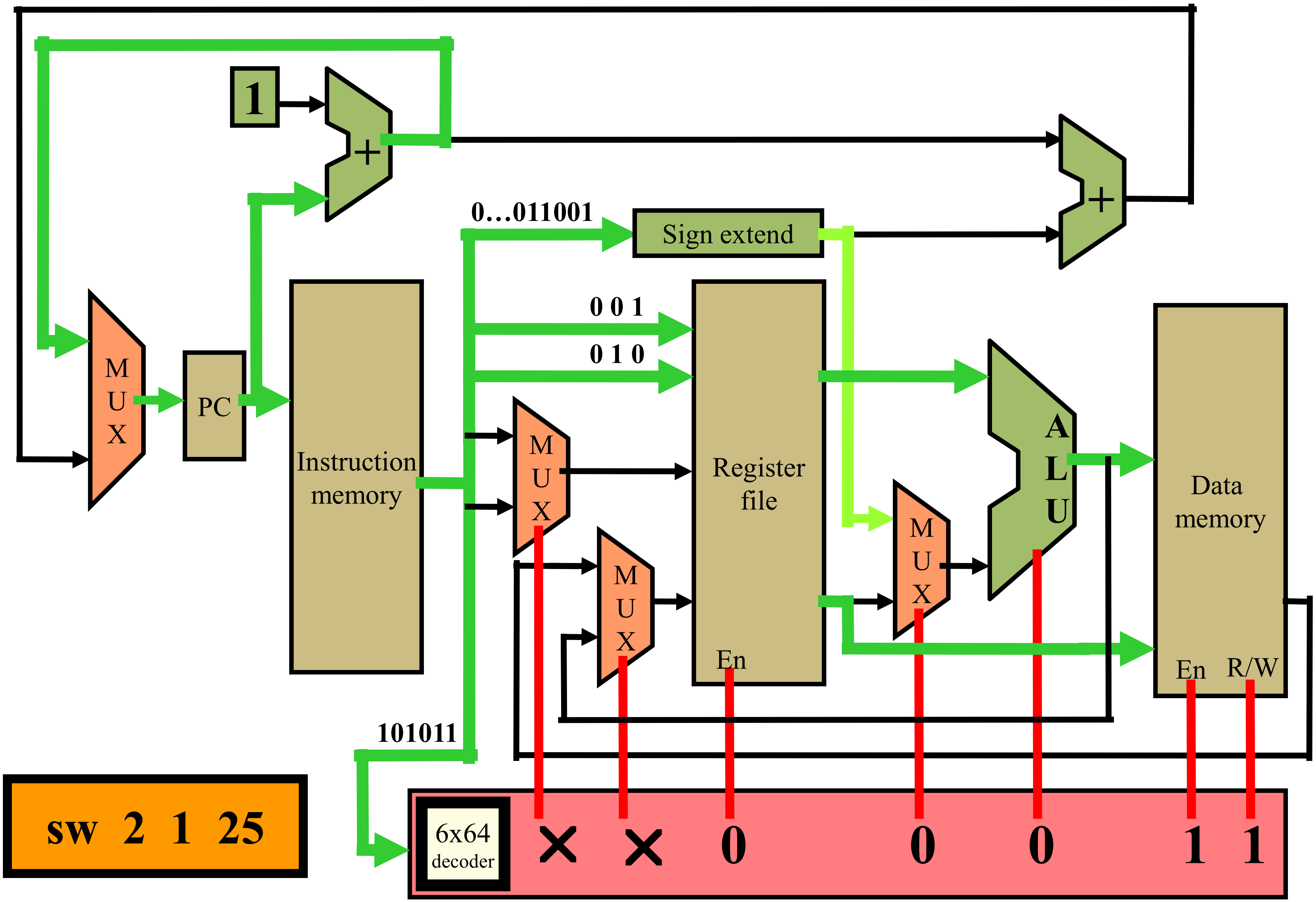


Executing an SW Instruction

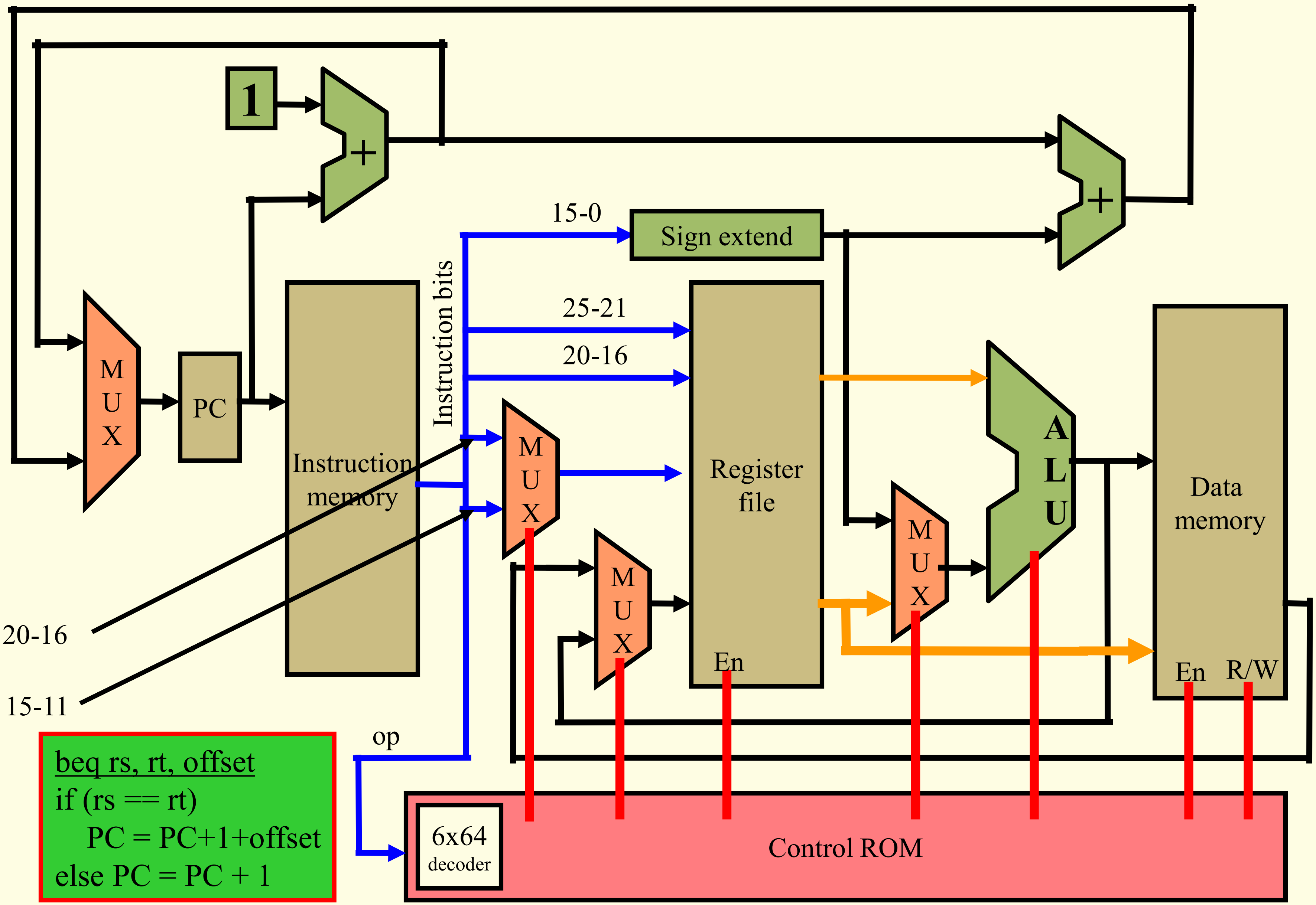


sw rt, offset(rs)
M[rs+offset] = rt
PC = PC + 1

Executing a SW instruction on this MIPS-32 datapath

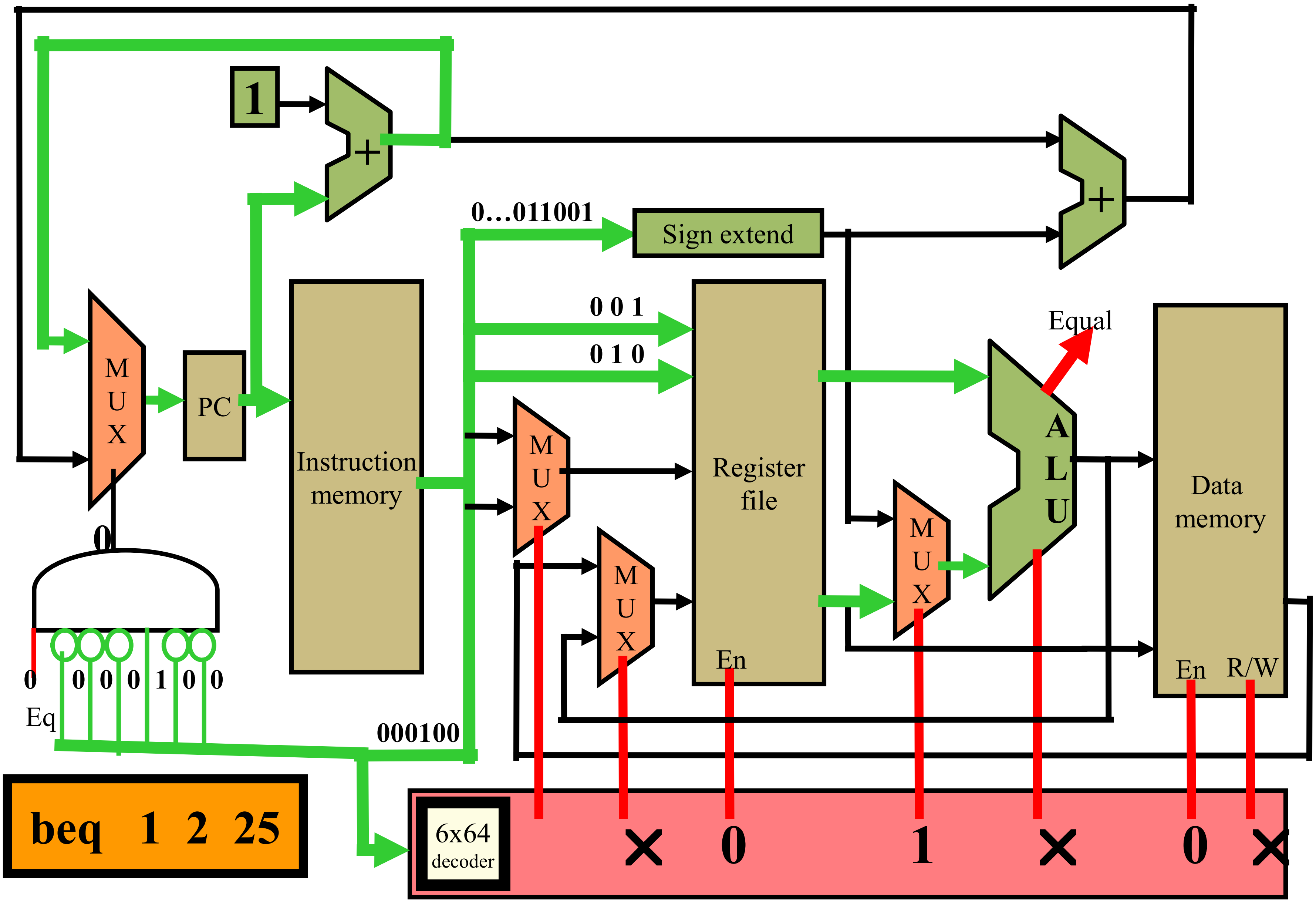


Executing an BEQ Instruction



beq rs, rt, offset
if (rs == rt)
    PC = PC+1+offset
else PC = PC + 1

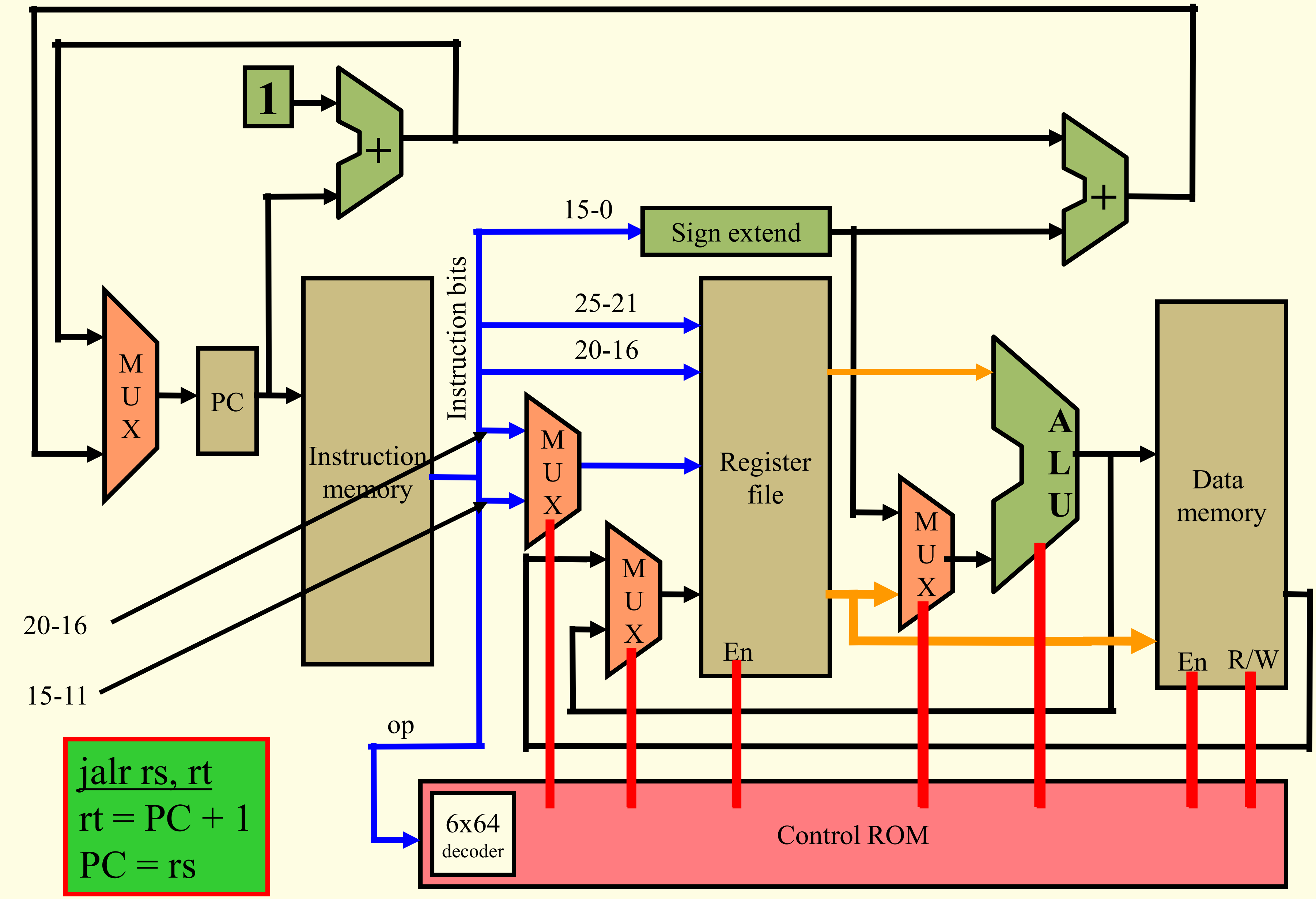
Executing a not taken BEQ instruction on this MIPS-32 datapath



So Far, So Good

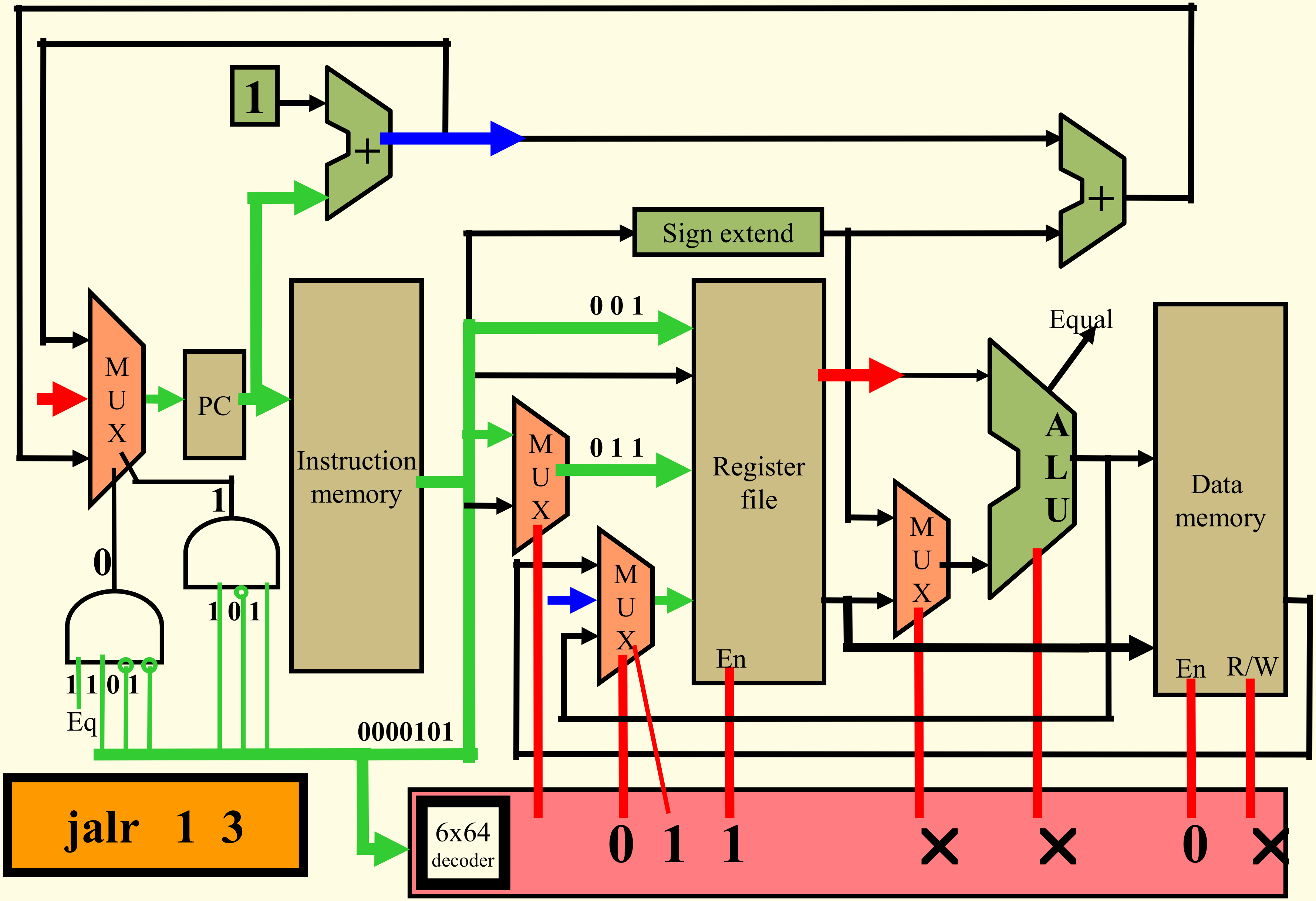
- Let try to add a more complex instruction.
- Jump And Link Register (JALR)
  - To implement JALR we need to
    - Write PC+1 into regB
    - Move regA into PC
  - JALR doesn’t fit into our nice clean datapath
    - Right now there is:
    - No path to write PC+1 into a register
    - No path to write a register to the PC

Executing an JALR Instruction

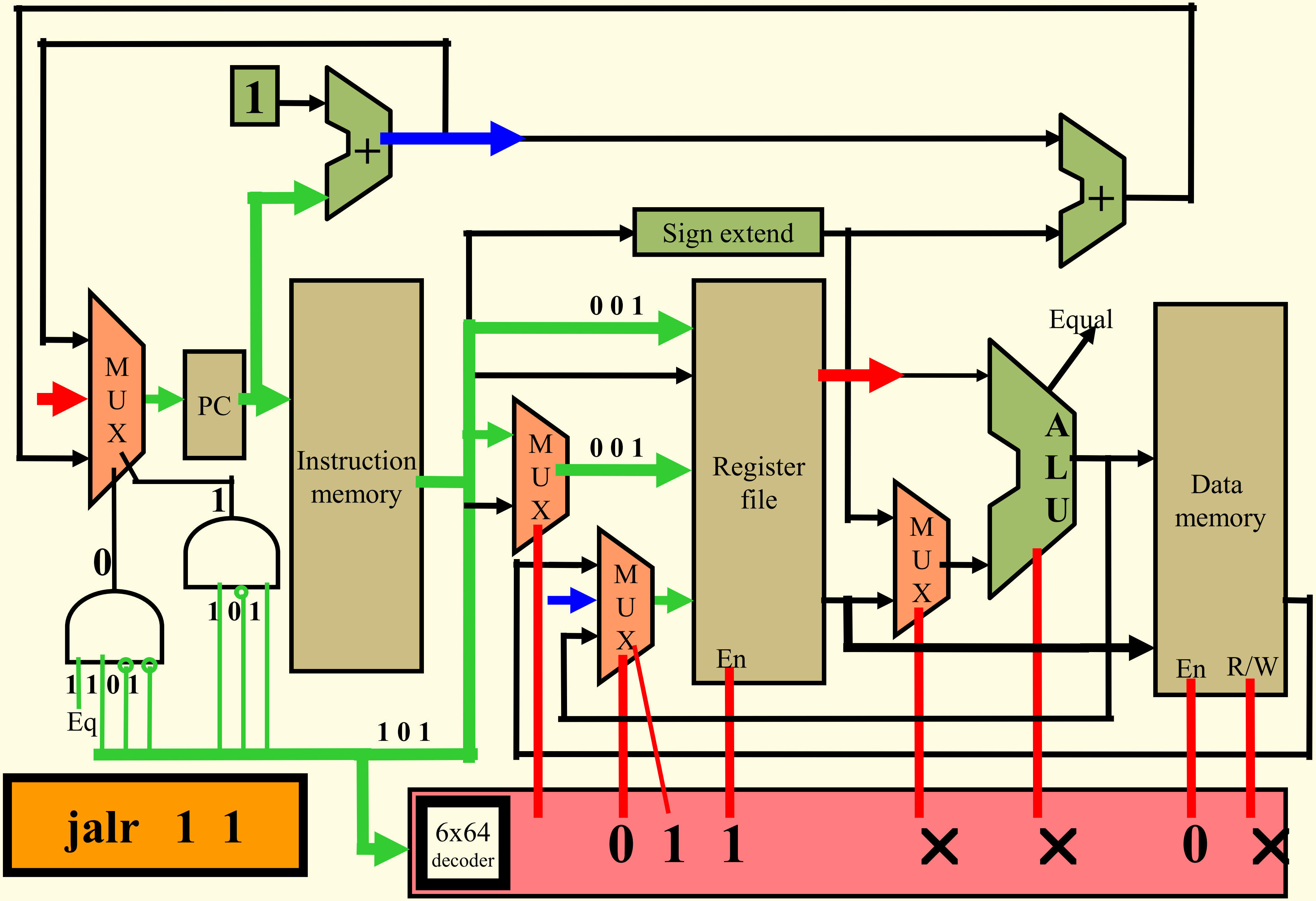


jalr rs, rt
rt = PC + 1
PC = rs

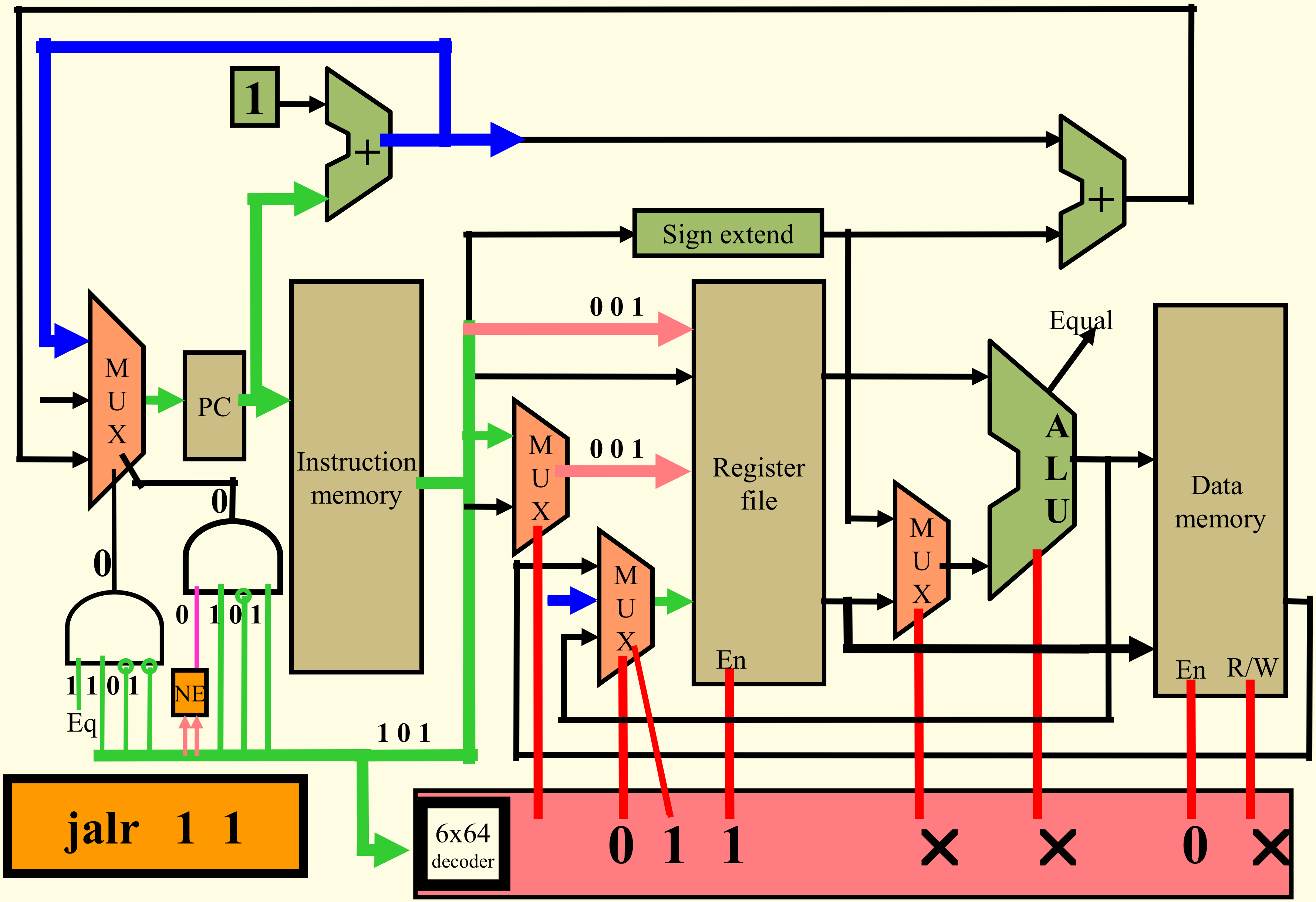
Executing a JALR Instruction



What If regA = regB for a JALR?



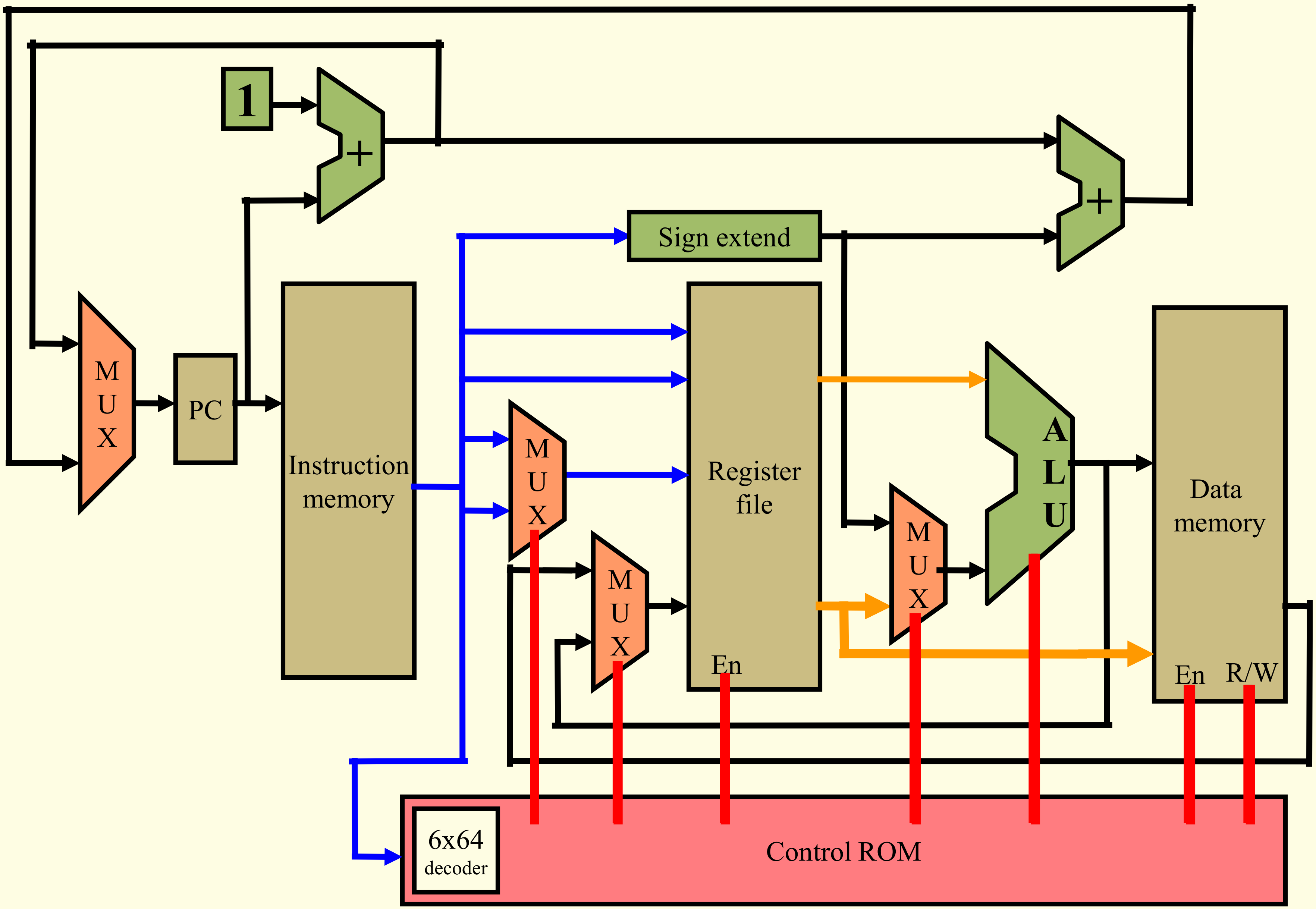
Changes for a JALR 1 1 Instruction



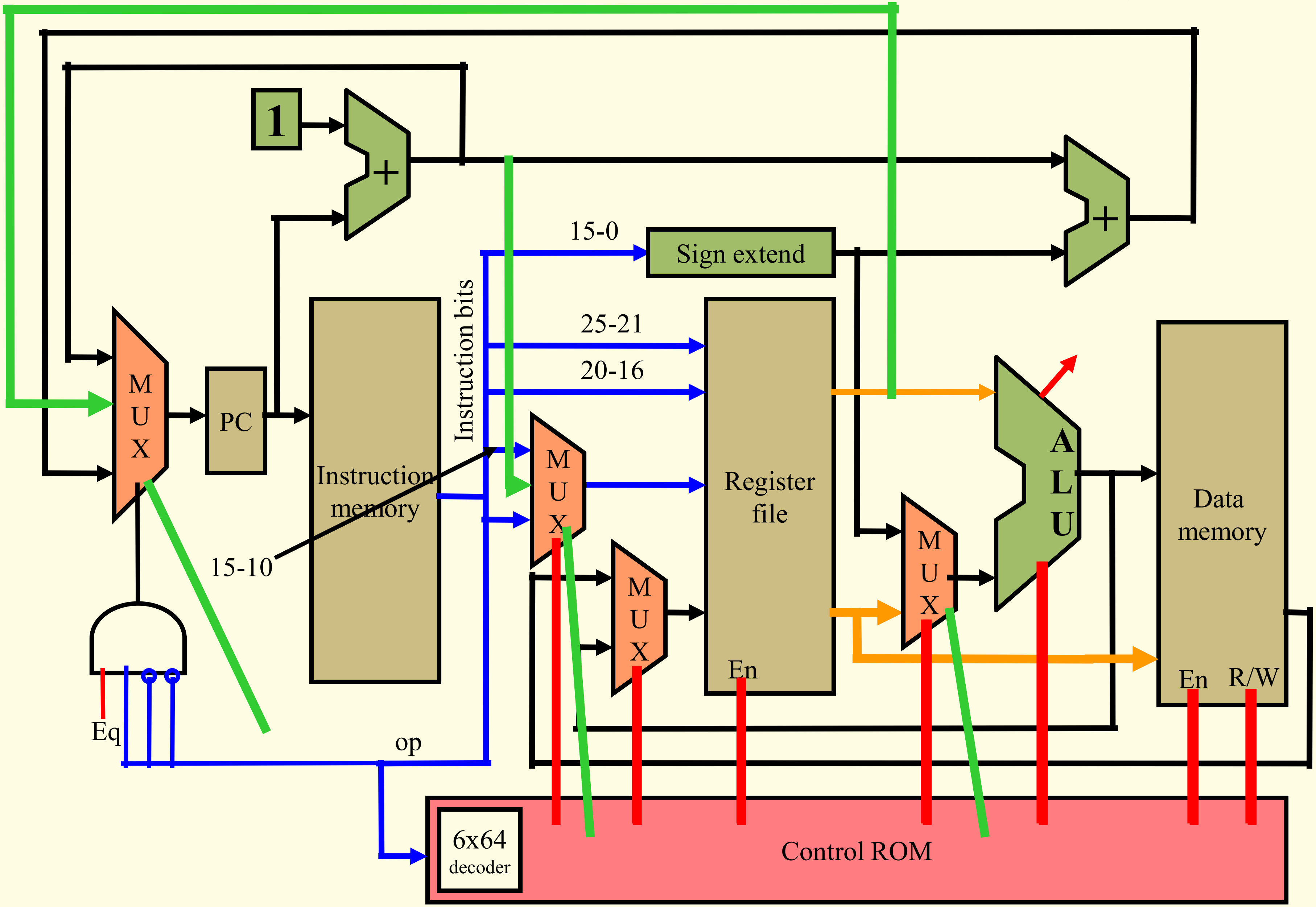
Class Problem 1

- Extend the single cycle datapath to perform the following operation
  - cmov rd, rs, rt
    - rd = rs (if rt != 0)
    - PC = PC + 1

Executing a CMOV Instruction

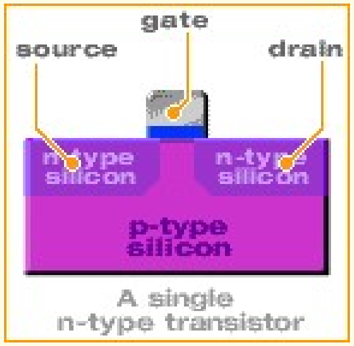


Class Problem 1 (continued)



Calculating cycle time

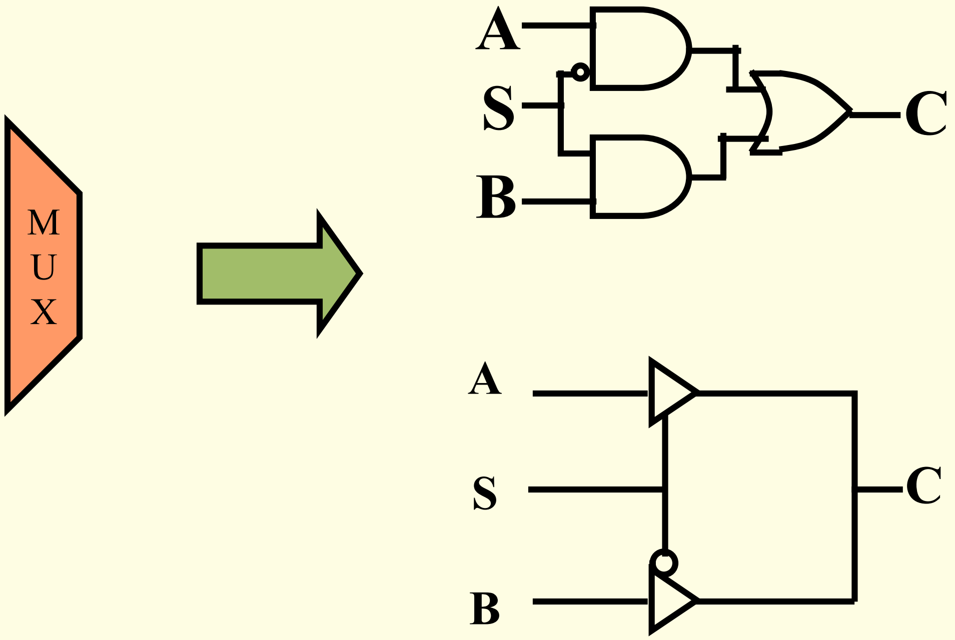
- Switching delay
  - Propagation time through a transistor



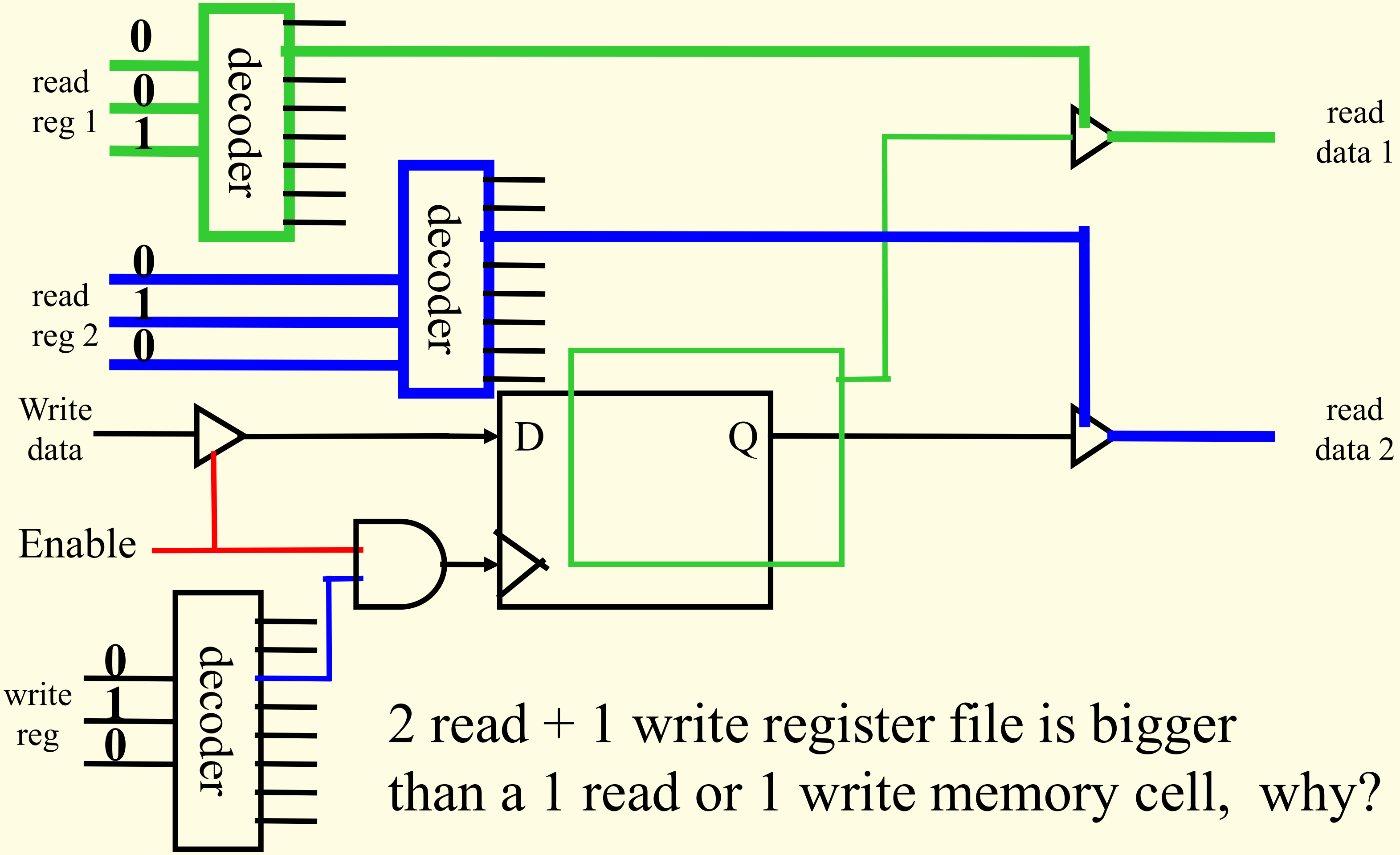
- Signal propagation
  - Electon flow over the wires
  - Set by the speed of light

Calculating cycle time

- Time through a MUX

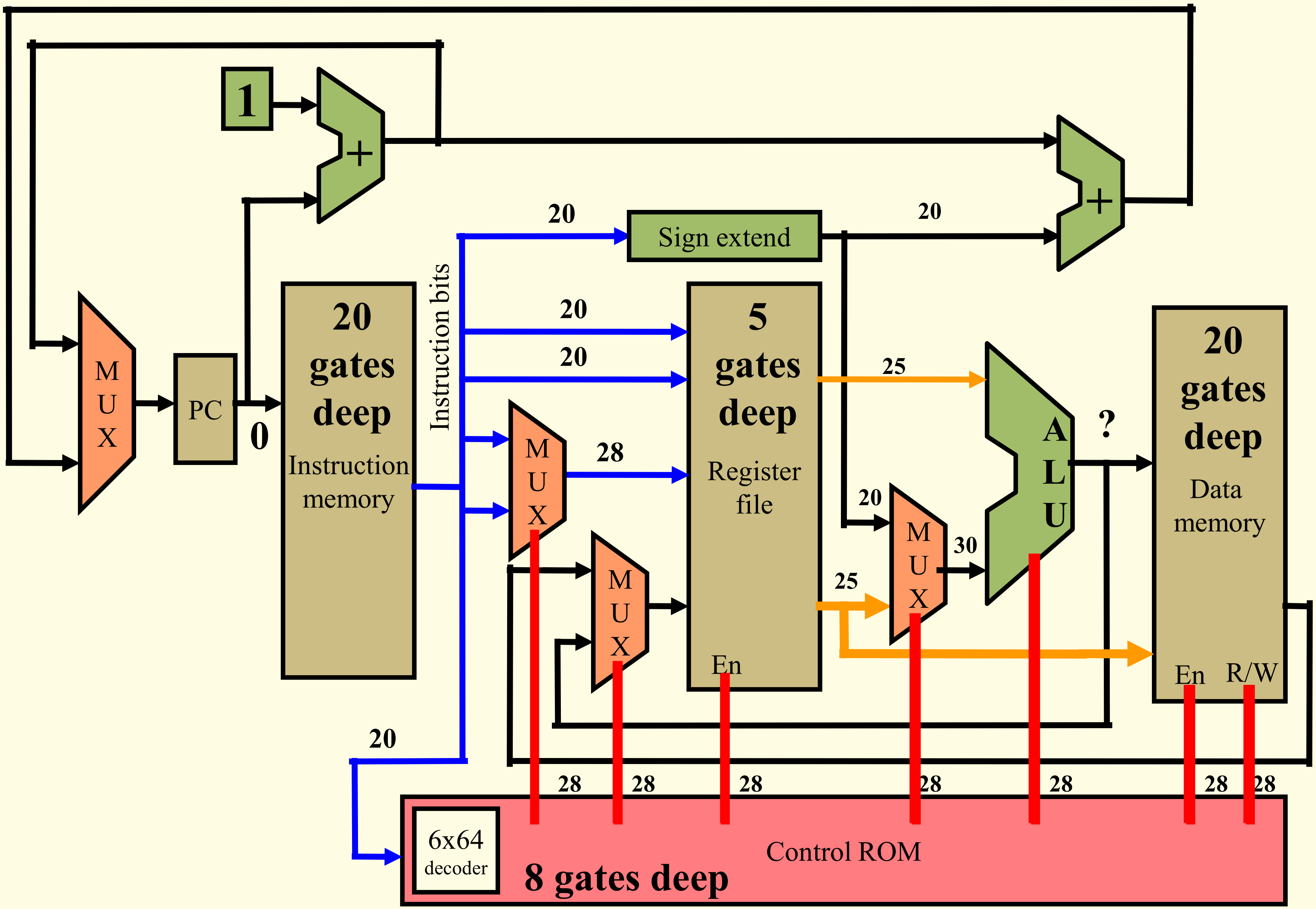


Time through a register file

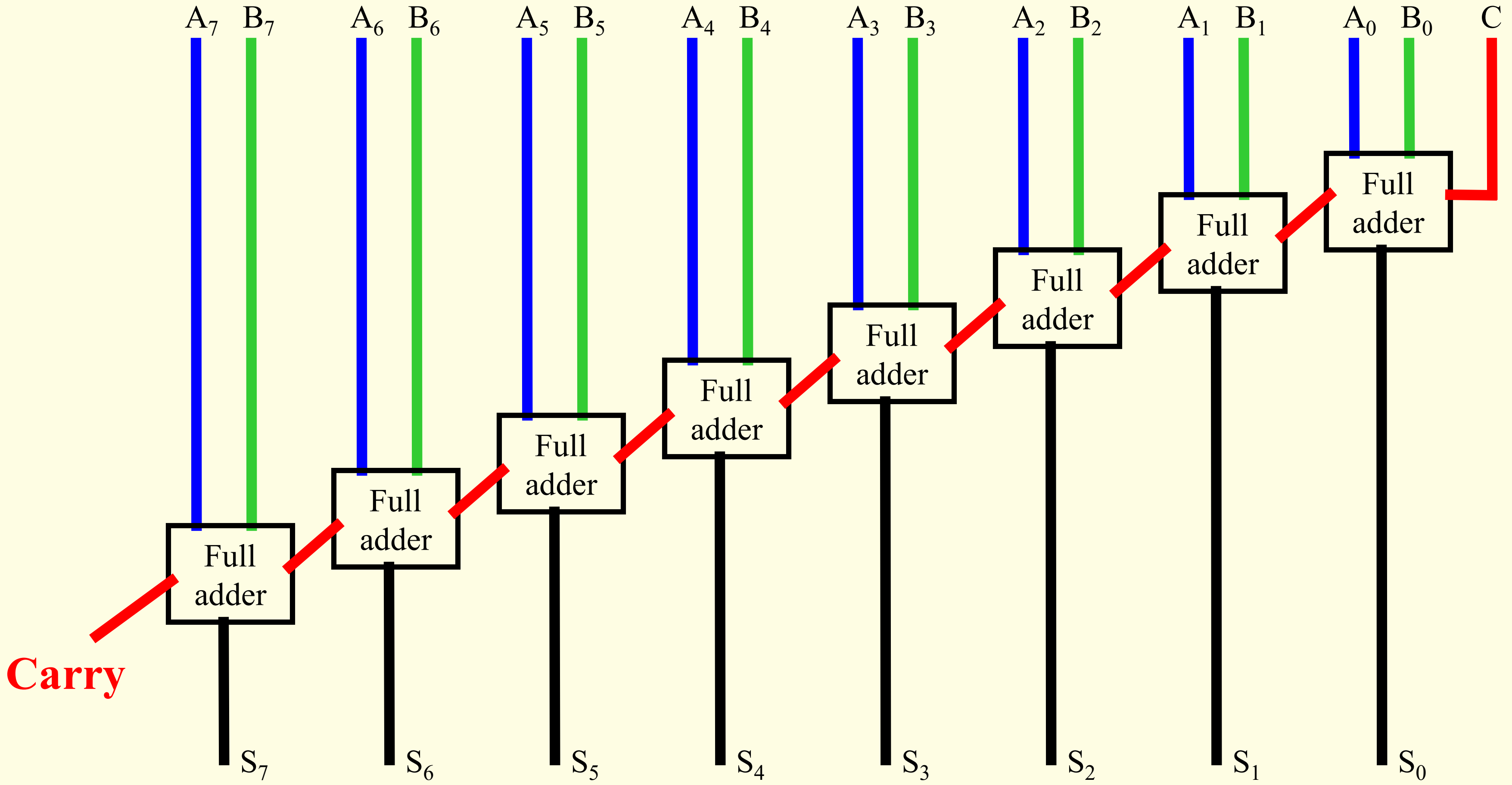


2 read + 1 write register file is bigger than a 1 read or 1 write memory cell, why?

Cycle time for Instructions

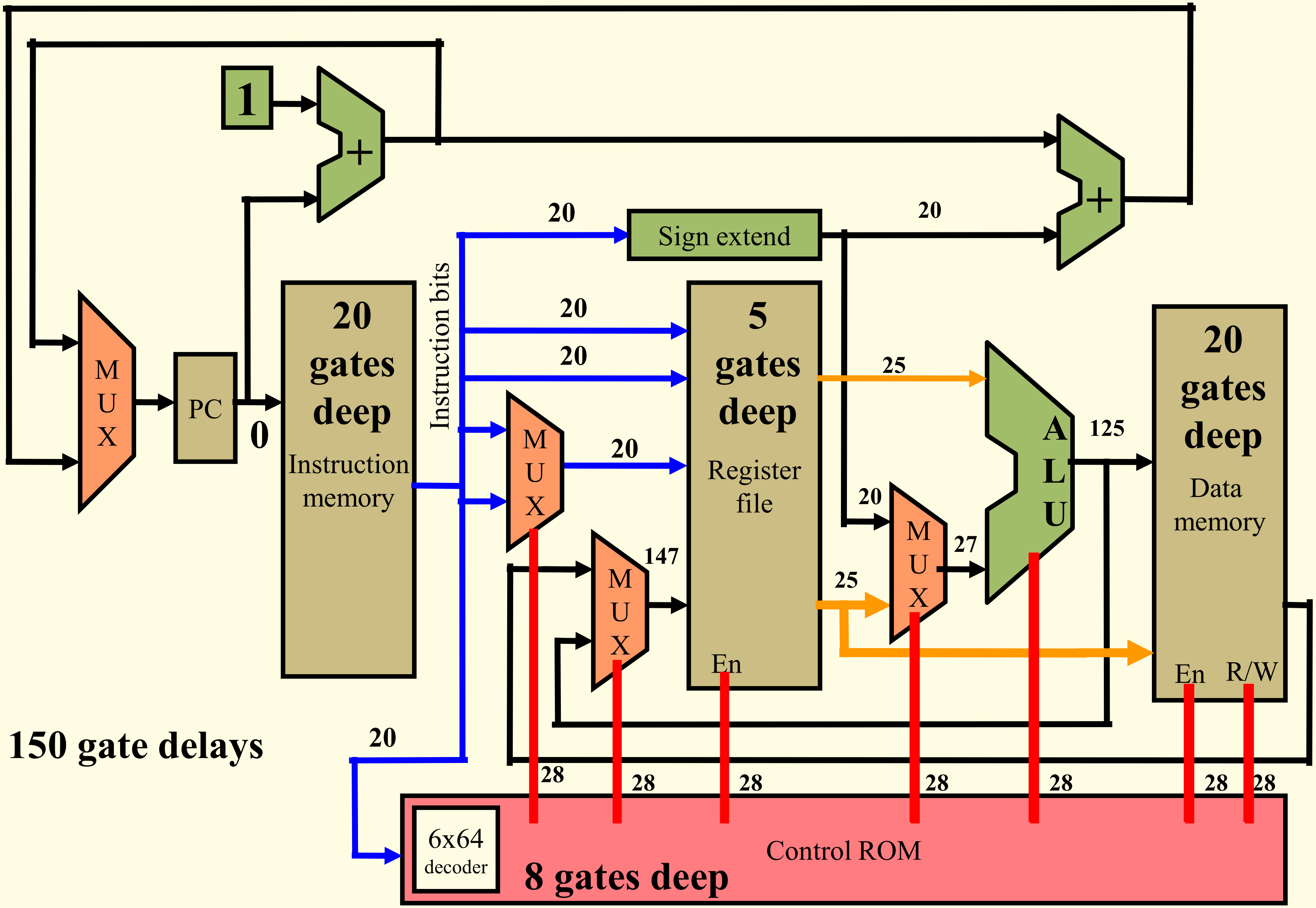


8-bit Ripple Carry Adder



Unfortunately this has a very large propagation time for 32 bit adds (~96)

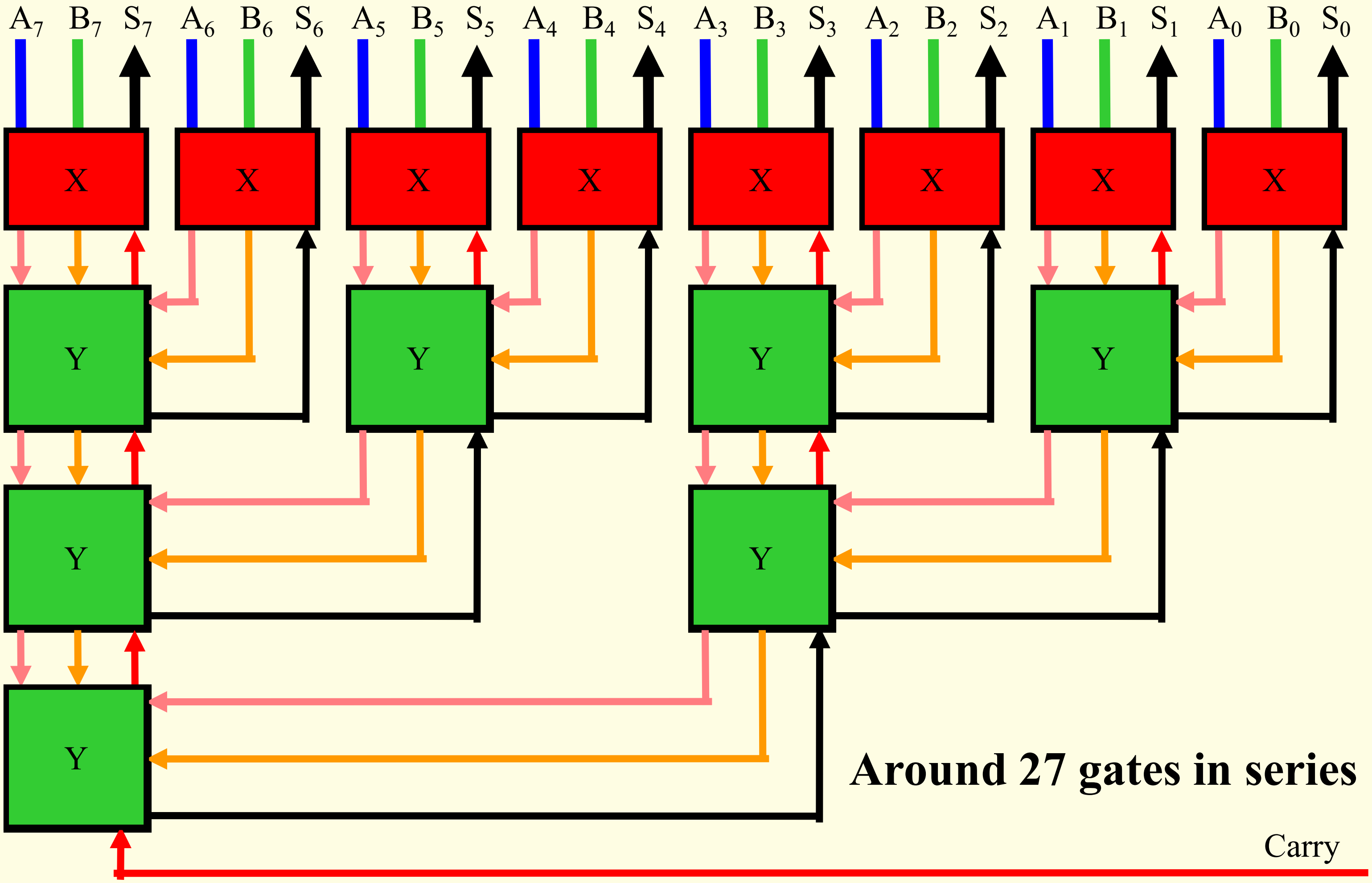
Cycle time for Instructions



out 150 gate delays

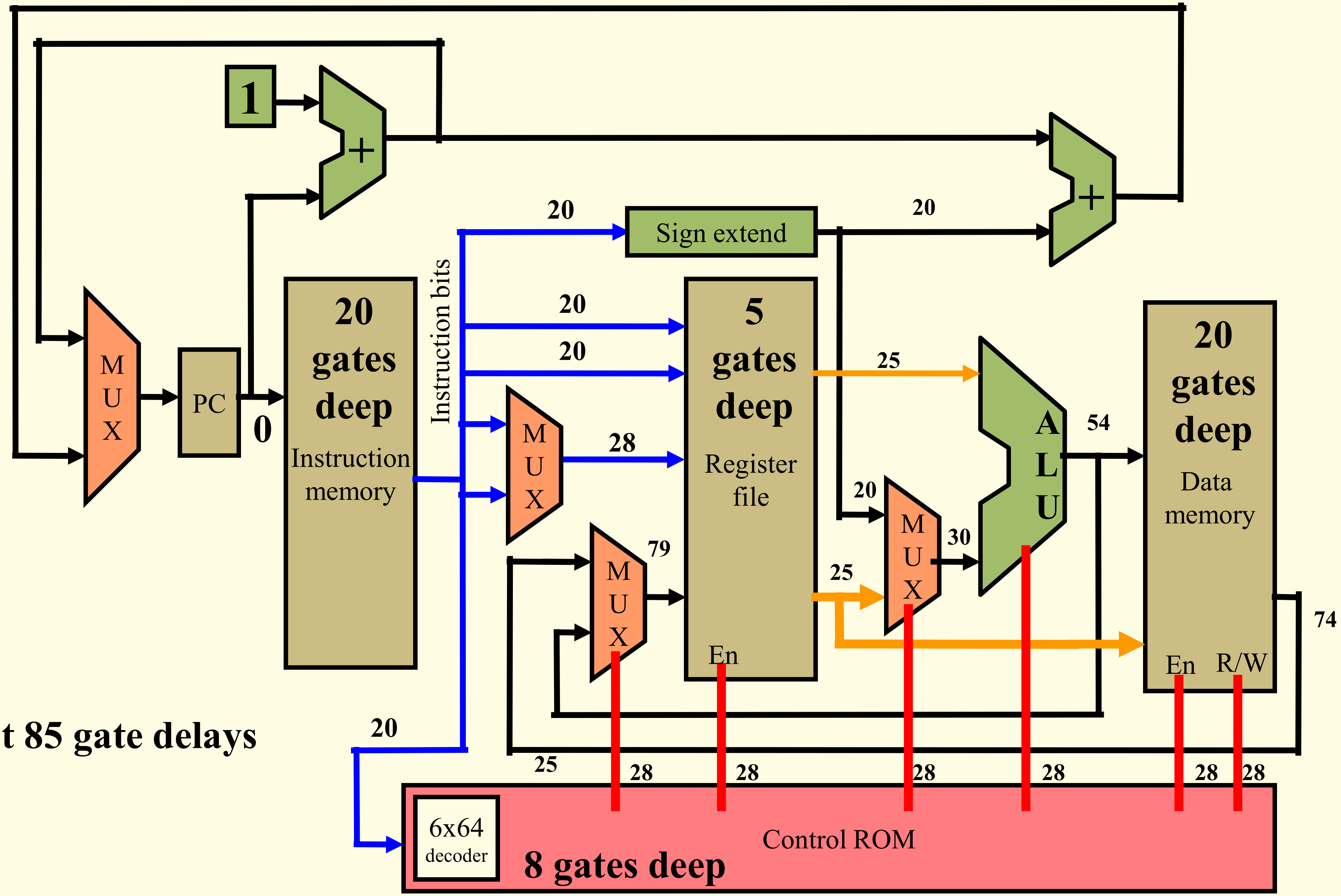
8-bit Carry Look-ahead Adder

A much faster (and more complex) adder for 32 or 64 bit adds


A_7 B_7 S_7 A_6 B_6 S_6 A_5 B_5 S_5 A_4 B_4 S_4 A_3 B_3 S_3 A_2 B_2 S_2 A_1 B_1 S_1 A_0 B_0 S_0
X
X
X
X
X
X
X
X
Y
Y
Y
Y
Y
Y
Y
Carry

Around 27 gates in series

Cycle time for Instructions



bout 85 gate delays

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 profit 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 operation**	**mem**	**write reg**	
add:	2ns	+ 1ns	+ 2ns		+ 1ns	= 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:

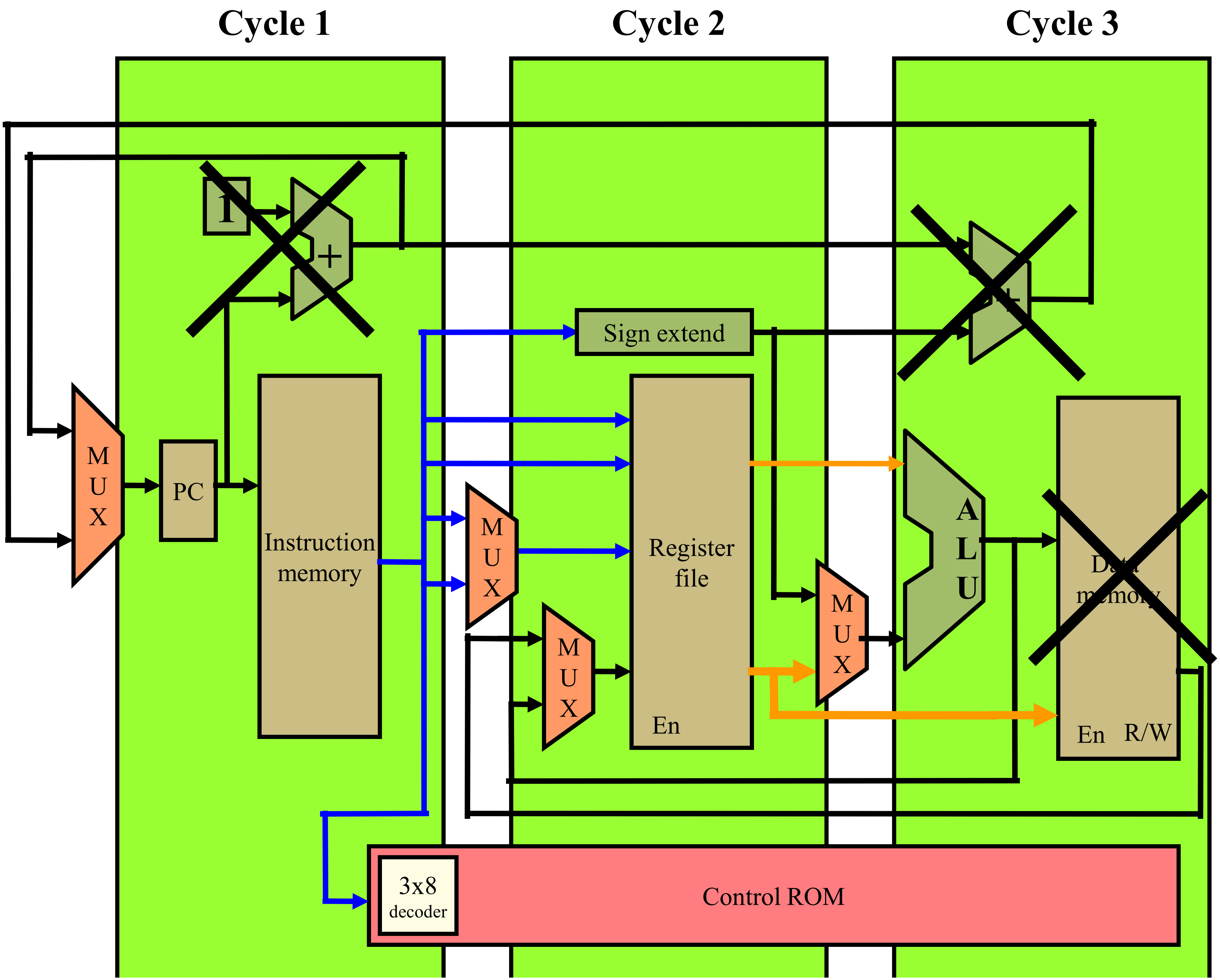
100 * 8ns = **800** ns

Optimal execution:

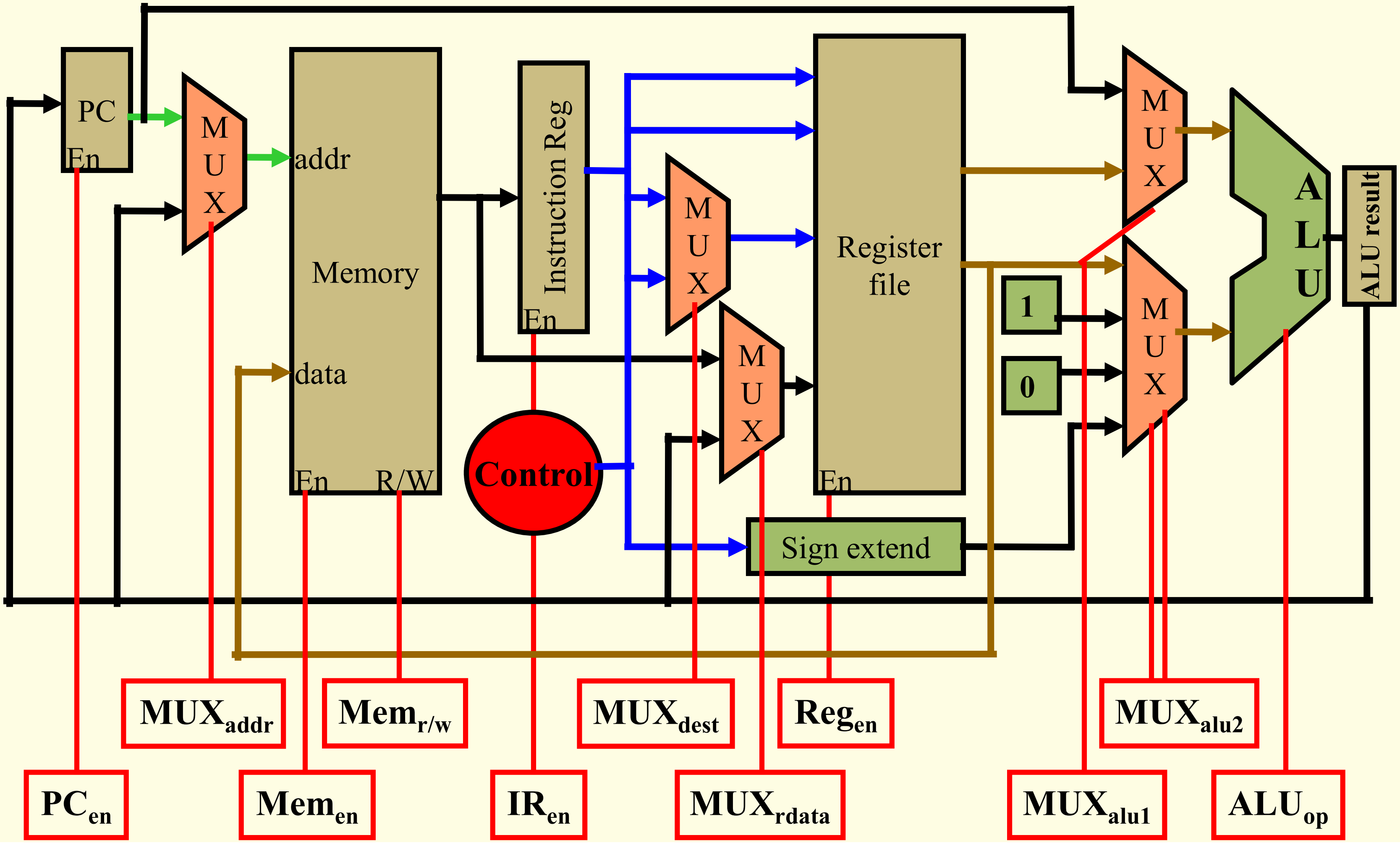
25*8ns + 10*7ns + 45*6ns + 20*5ns = **640** ns

Multiple-cycle Execution

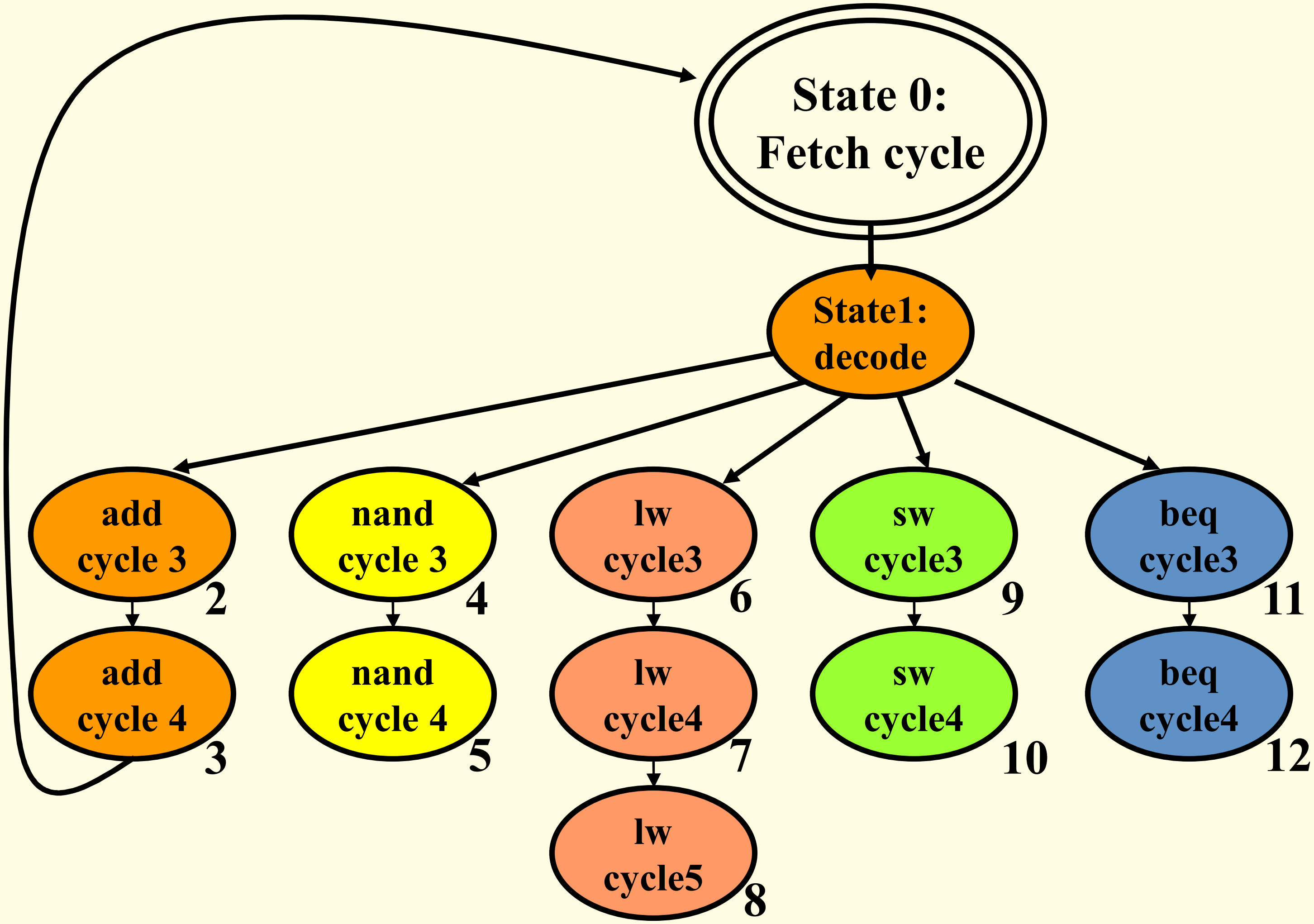
- Each instruction takes multiple cycles to execute
  - Cycle time is reduced!
  - Slower instructions take more cycles
  - Can reuse datapath elements each cycle (smaller circuit)
- What is needed to make this work?
  - Since you are re-using elements for different purposes, you need more and/or wider 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
+
+
Sign extend
M U X
PC
Instruction memory
M U X
M U X
Register file
En
M U X
A L U
Data memory
En
R/W
3x8 decoder
Control ROM

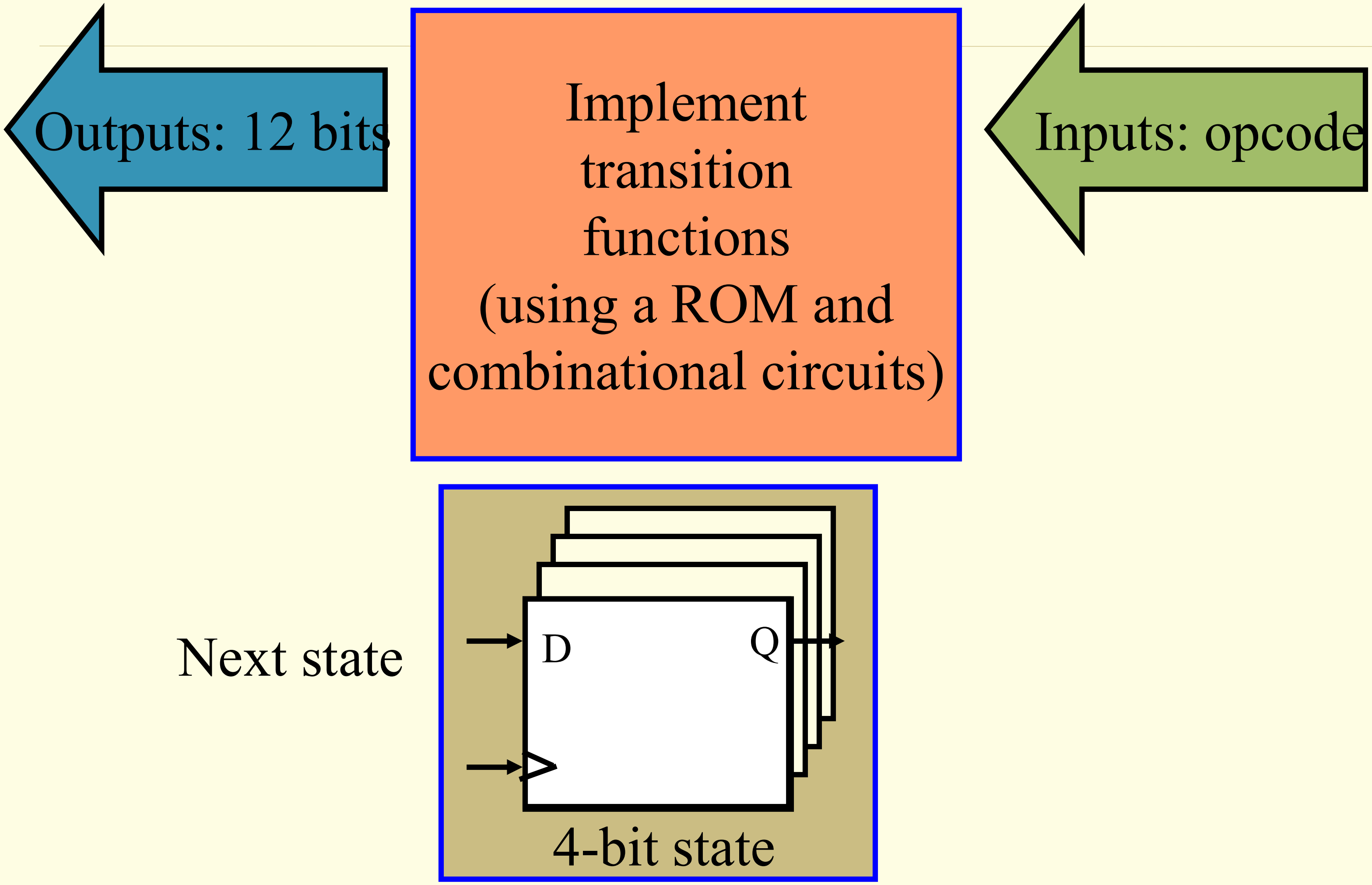
Multi-cycle MIPS-32 Datapath



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



Implementing FSM



Outputs: 12 bits

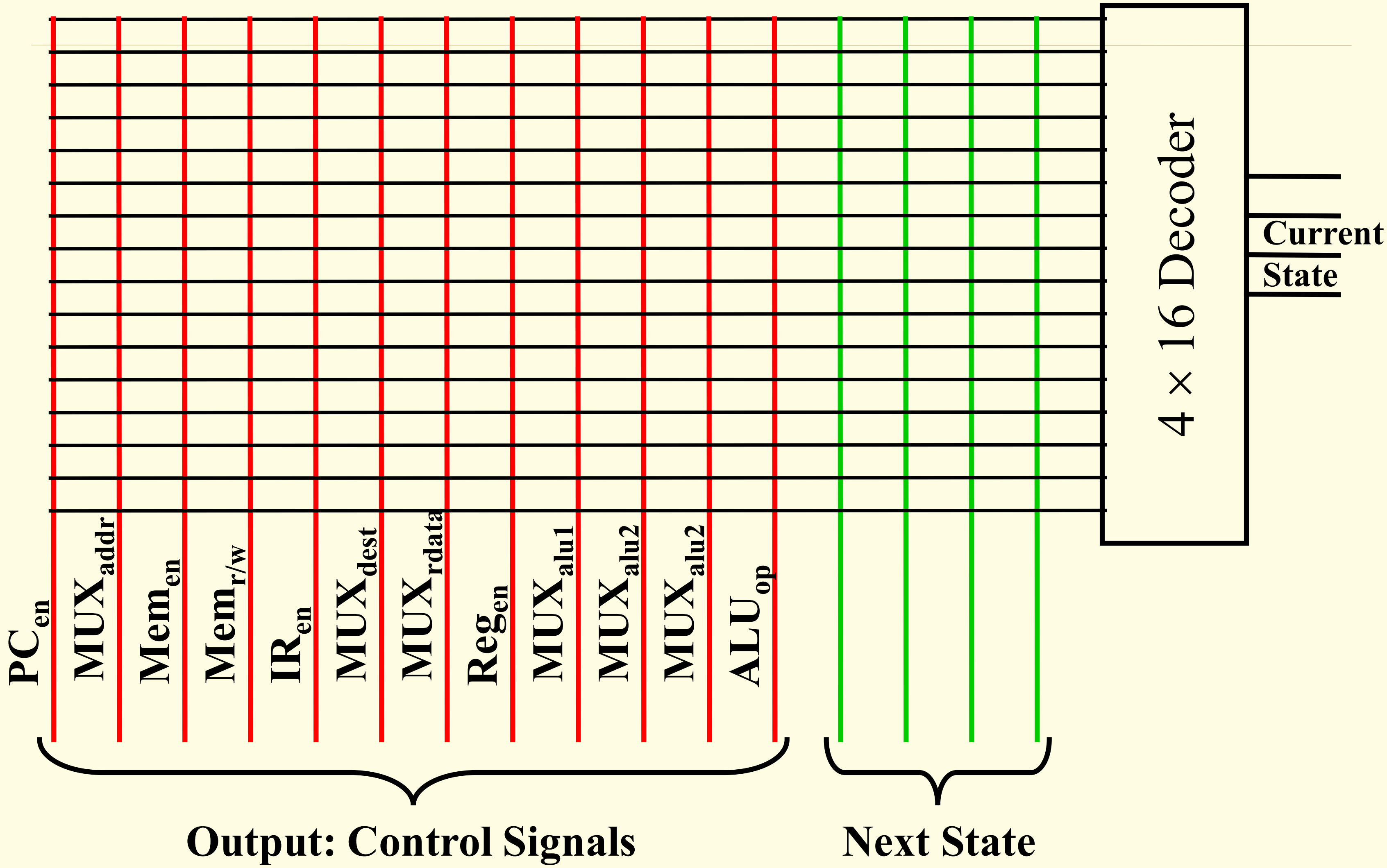
Implement transition functions (using a ROM and combinational circuits)

Inputs: opcode

Next state

4-bit state

Building the Control Rom



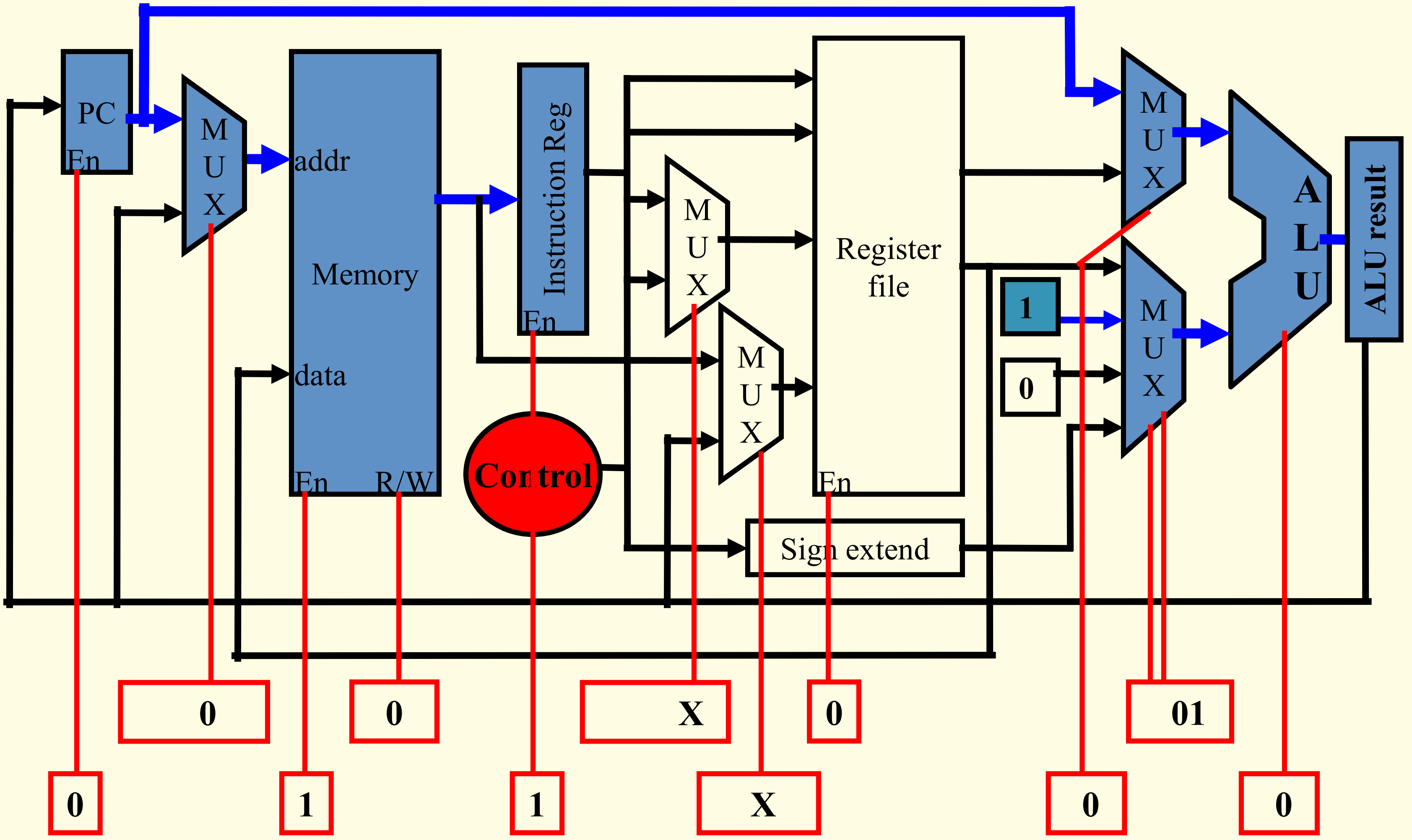
First Cycle (State 0) Fetch Instr

- 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

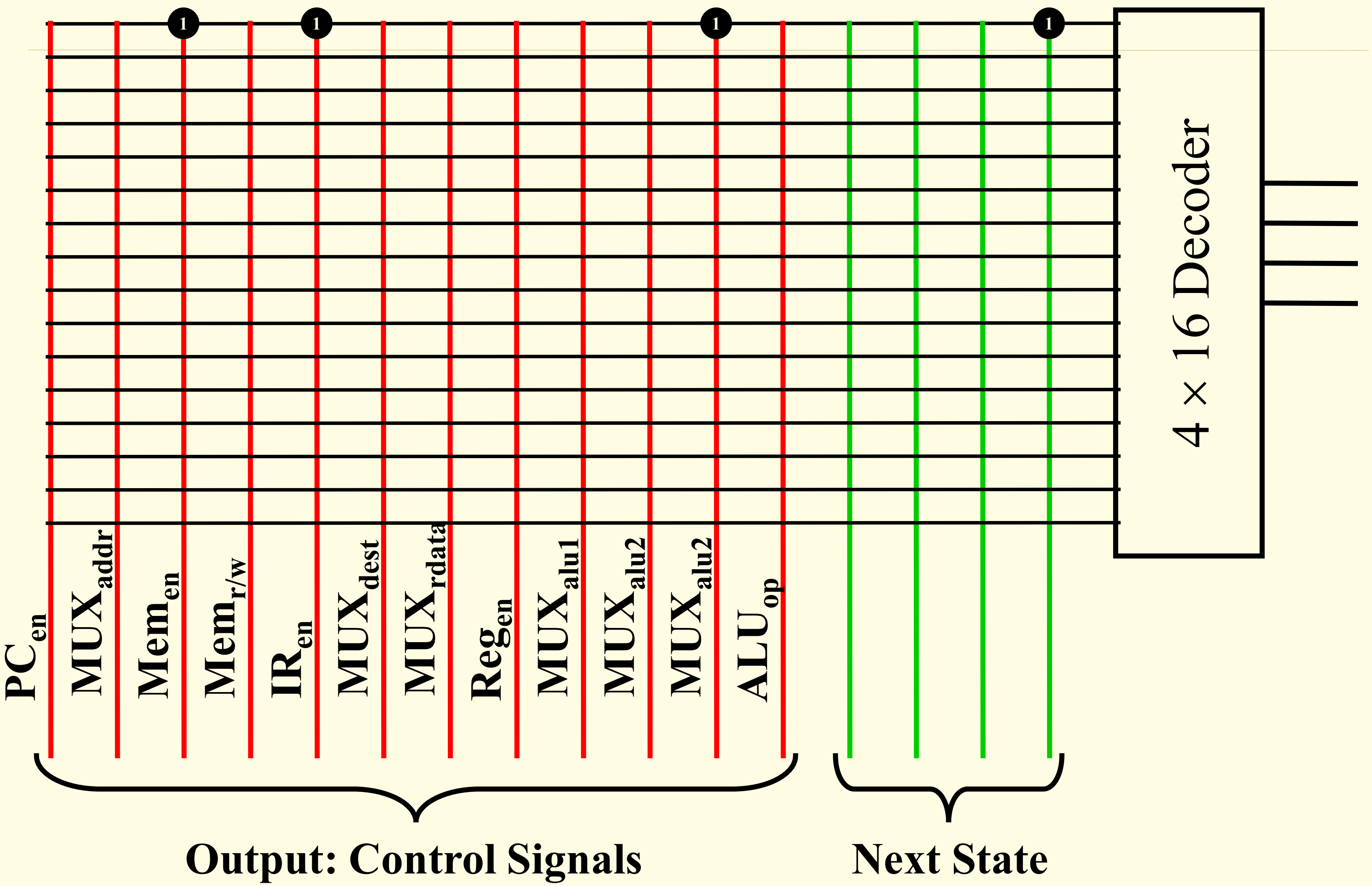
Cycle 1 Operation

This is the same for all instructions

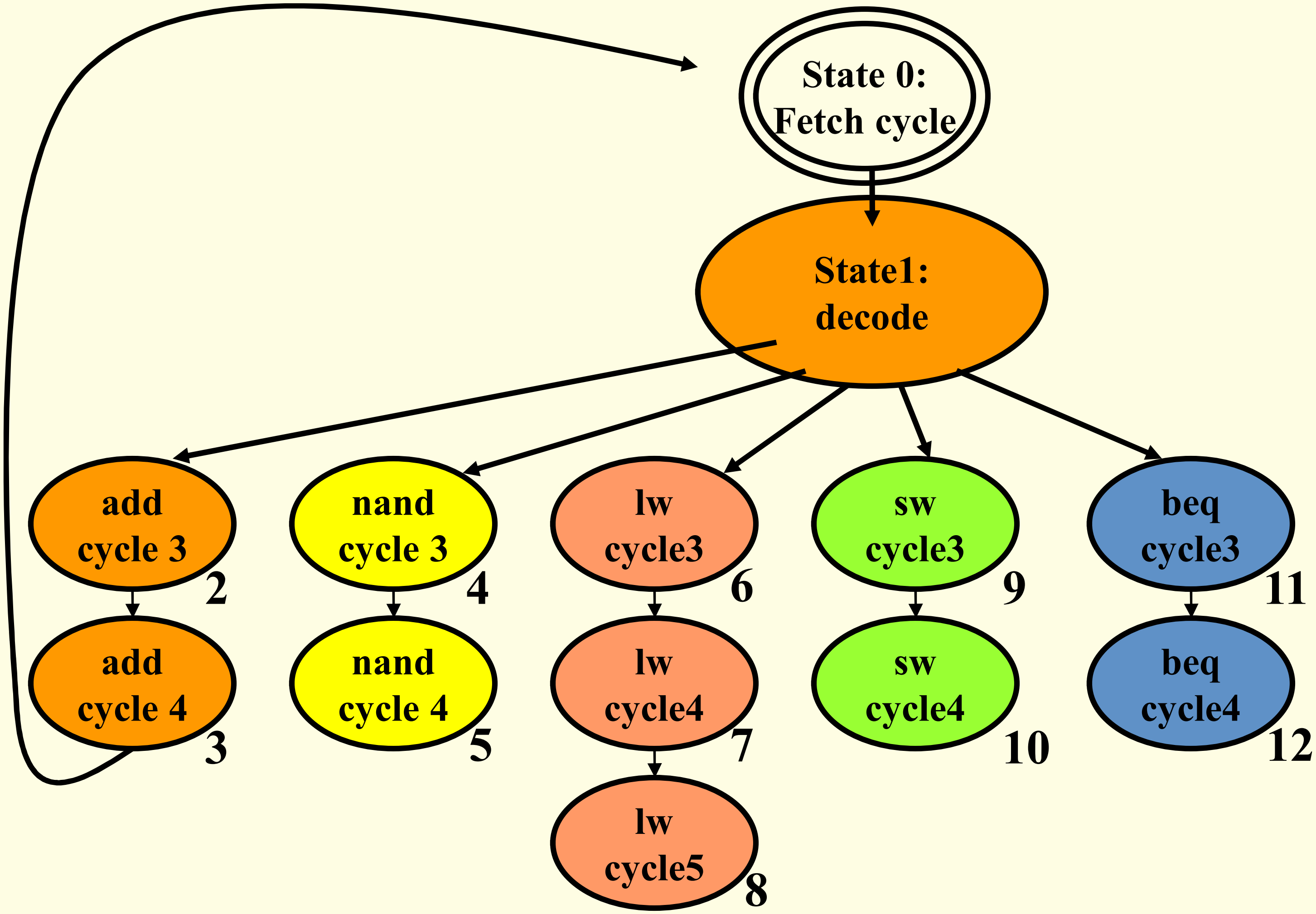
(since we don’t know the instruction yet!)



Building the Control Rom

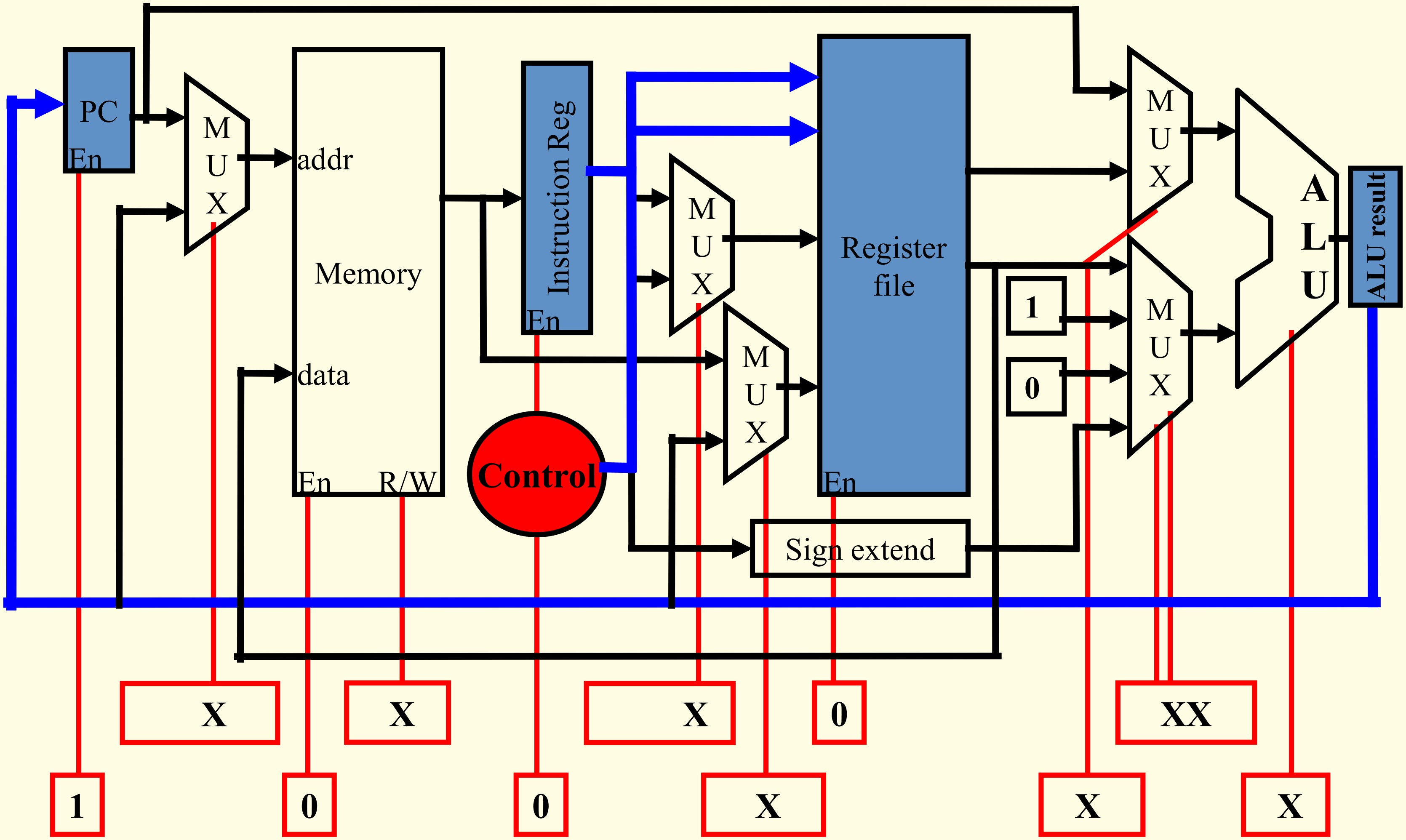


State 1: instruction decode

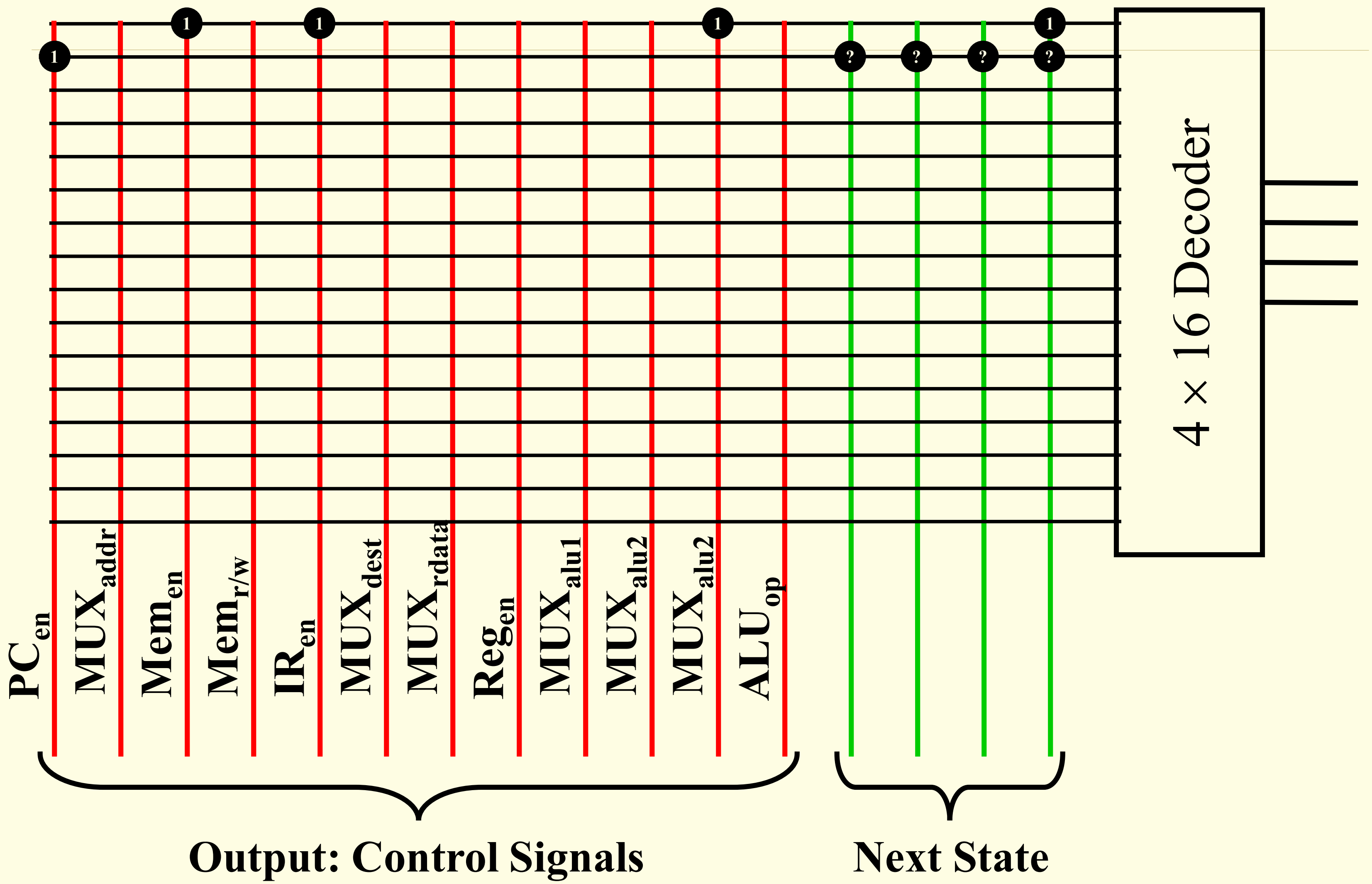


State 1: output function

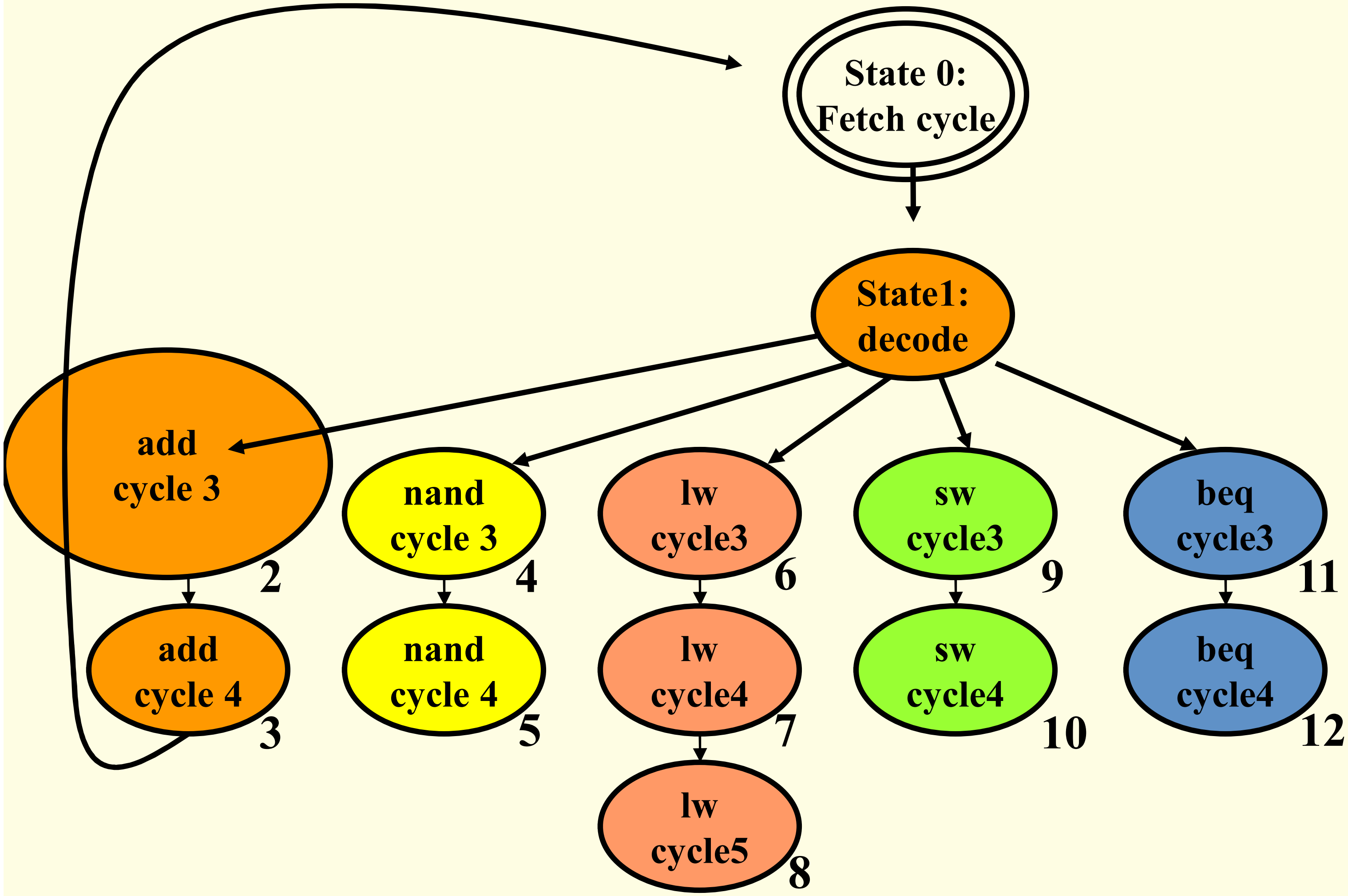
**Update PC; read registers (RegA and regB);
use opcode to determine next state**



Building the Control Rom

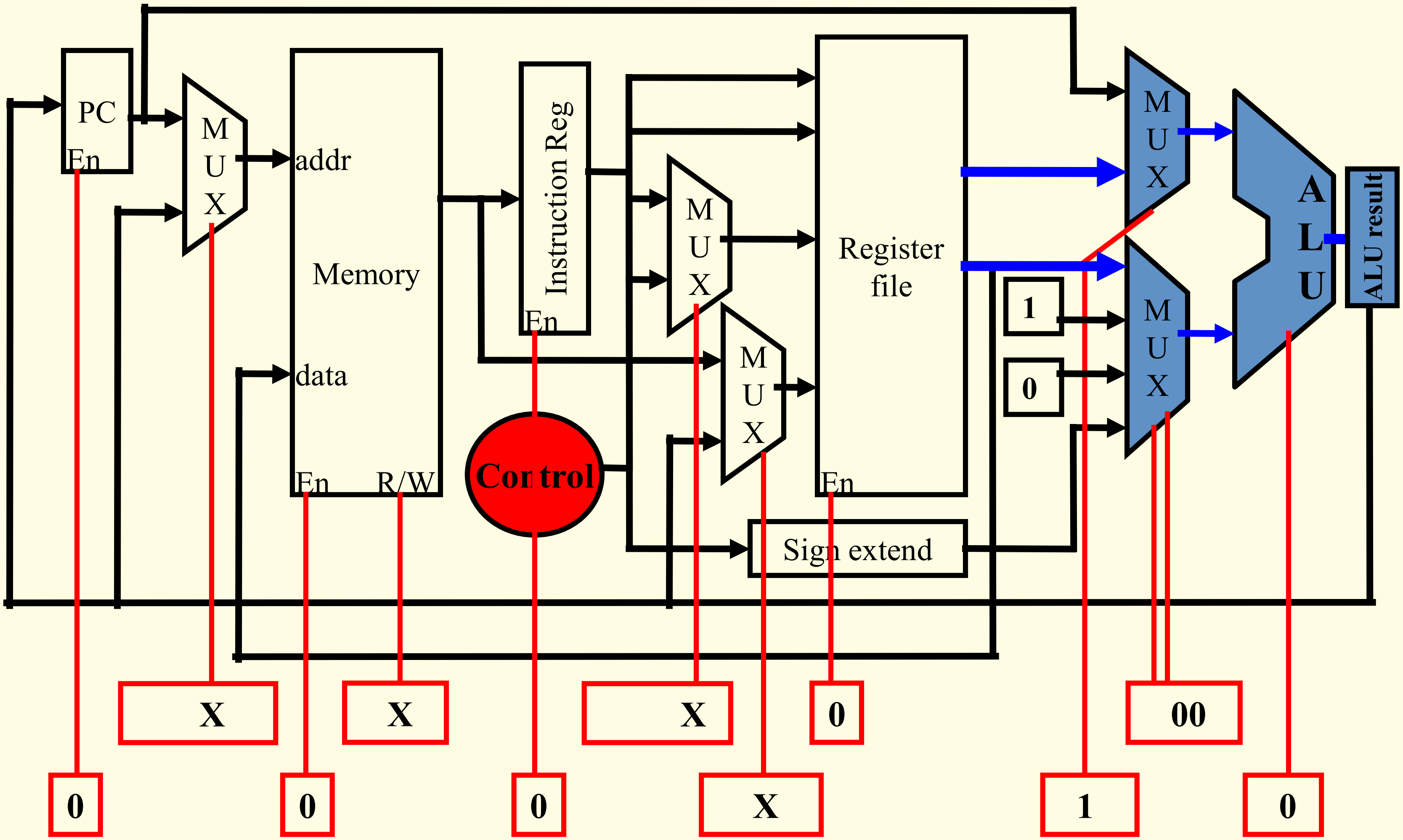


State 2: Add cycle 3

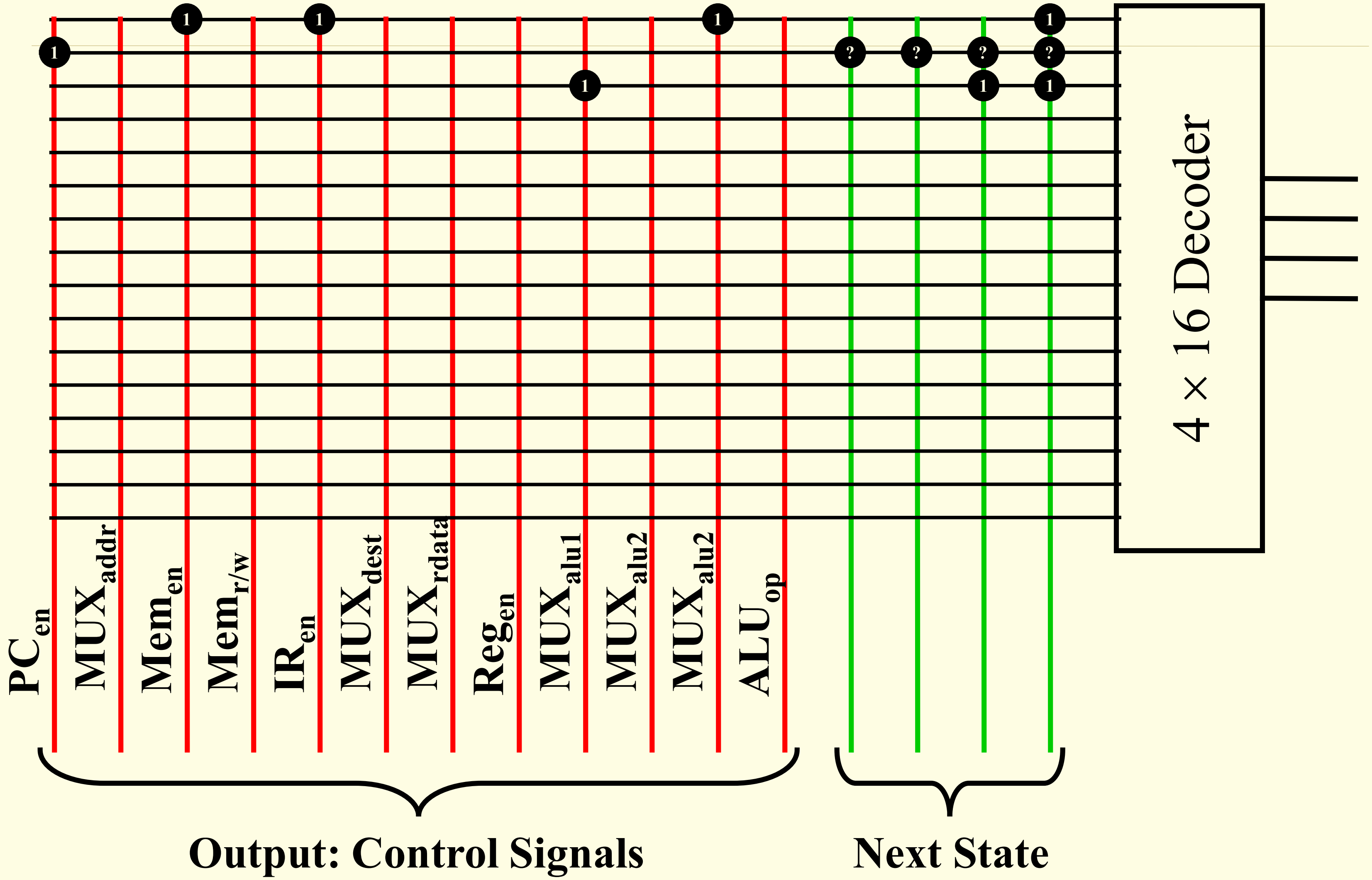


State 2: Add Cycle 3 Operation

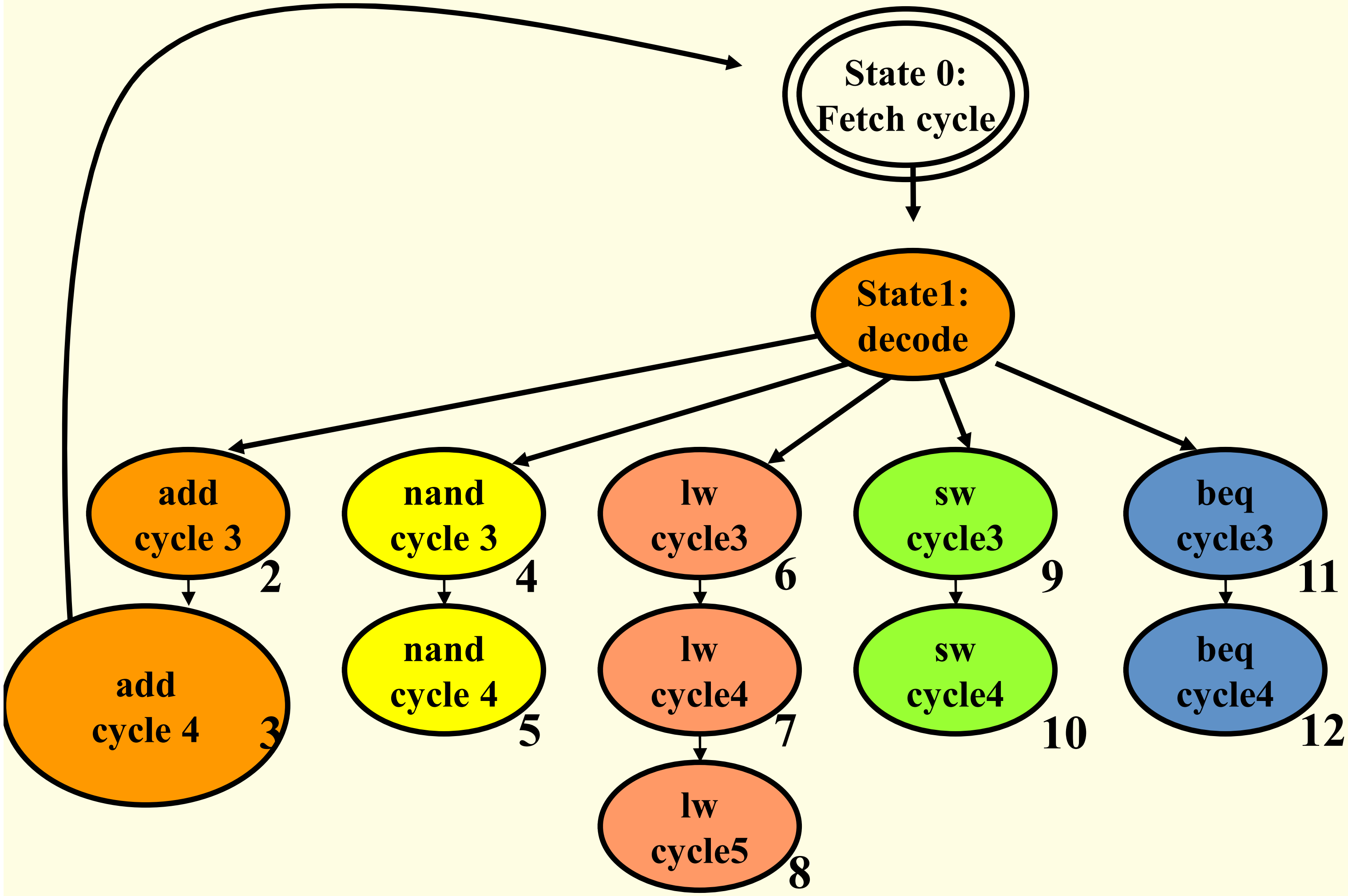
Send control signals to MUX to select values of regA and regB and control signal to ALU to add



Building the Control Rom

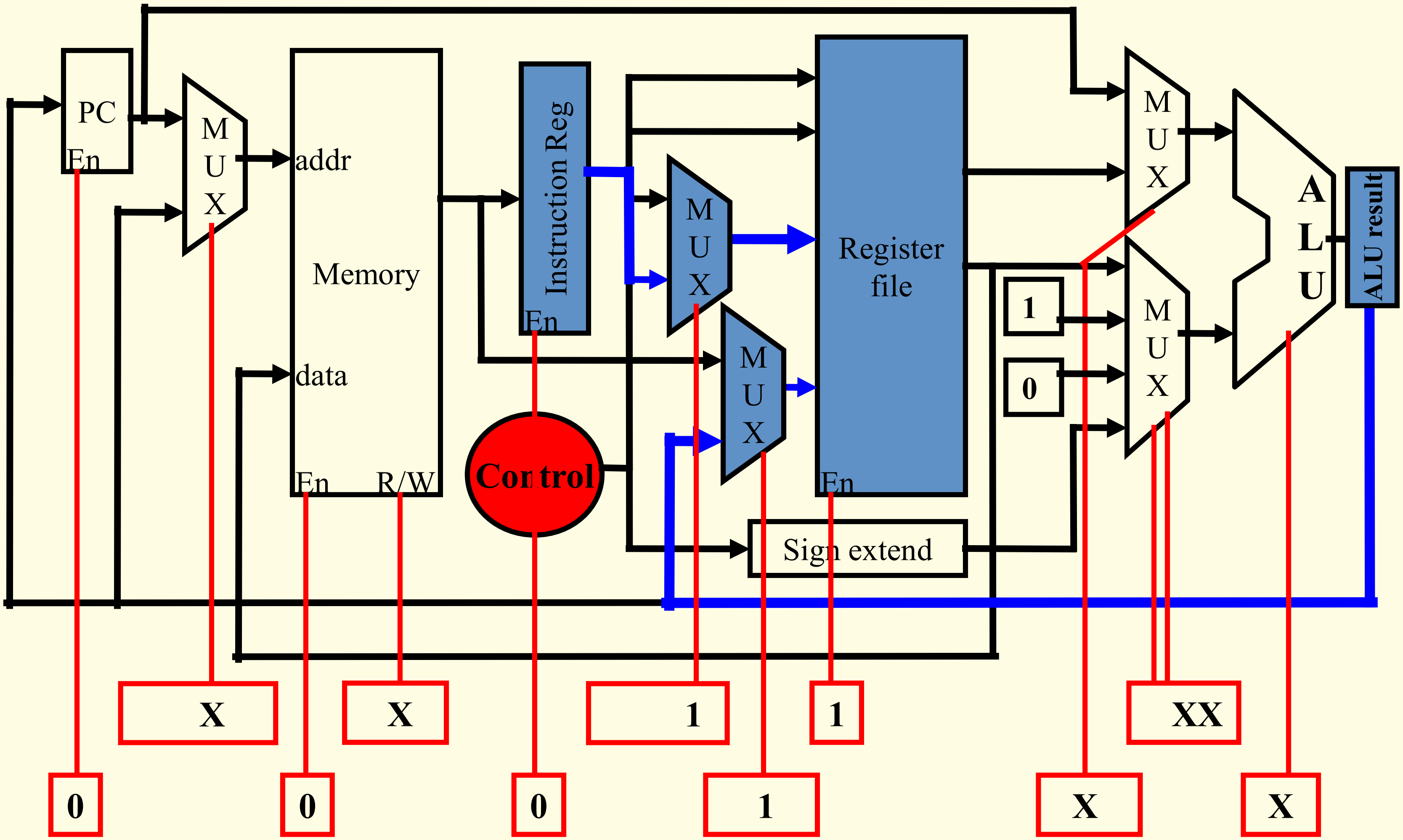


State 2: Add cycle 3

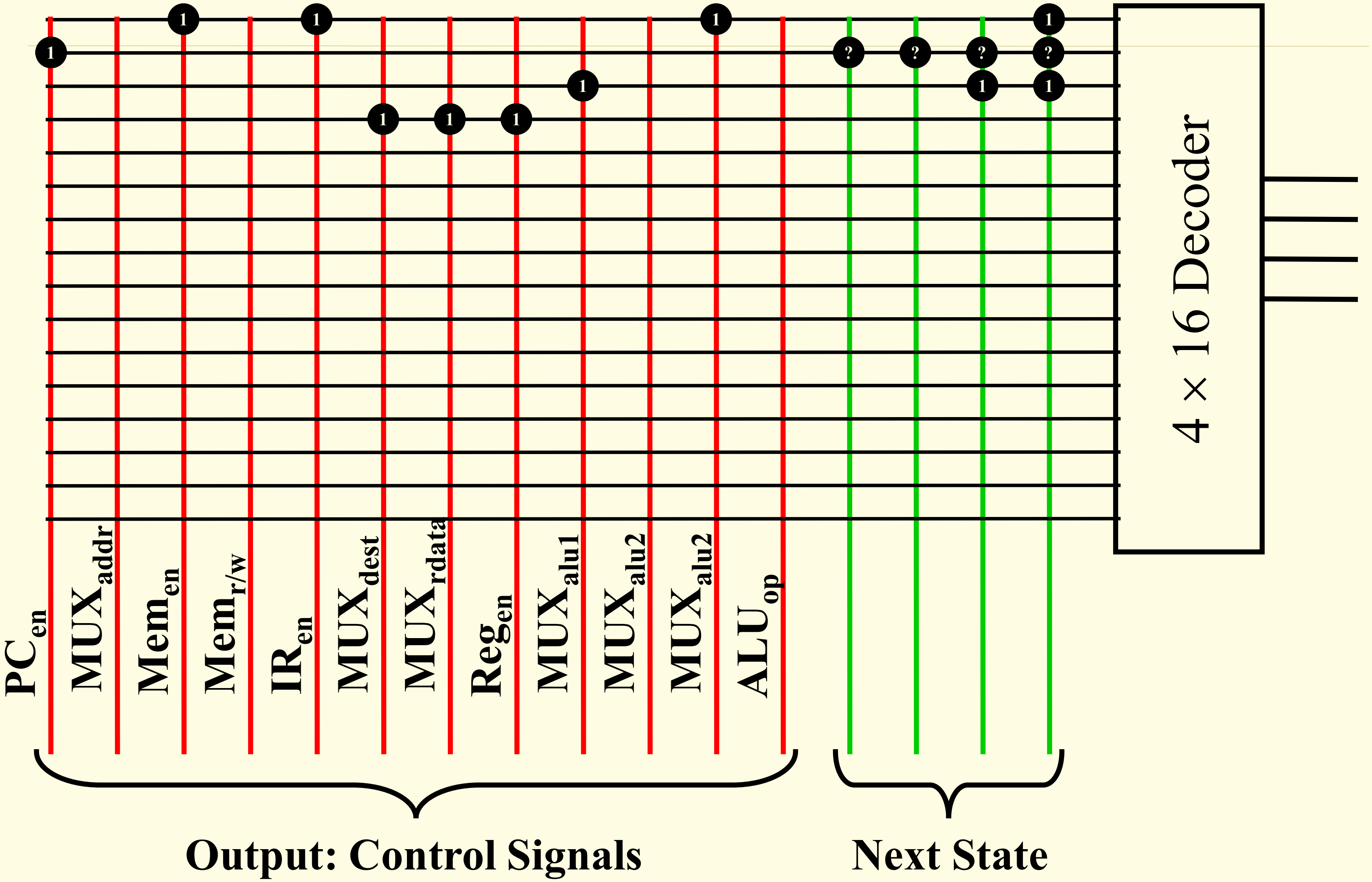


Add Cycle 4 Operation

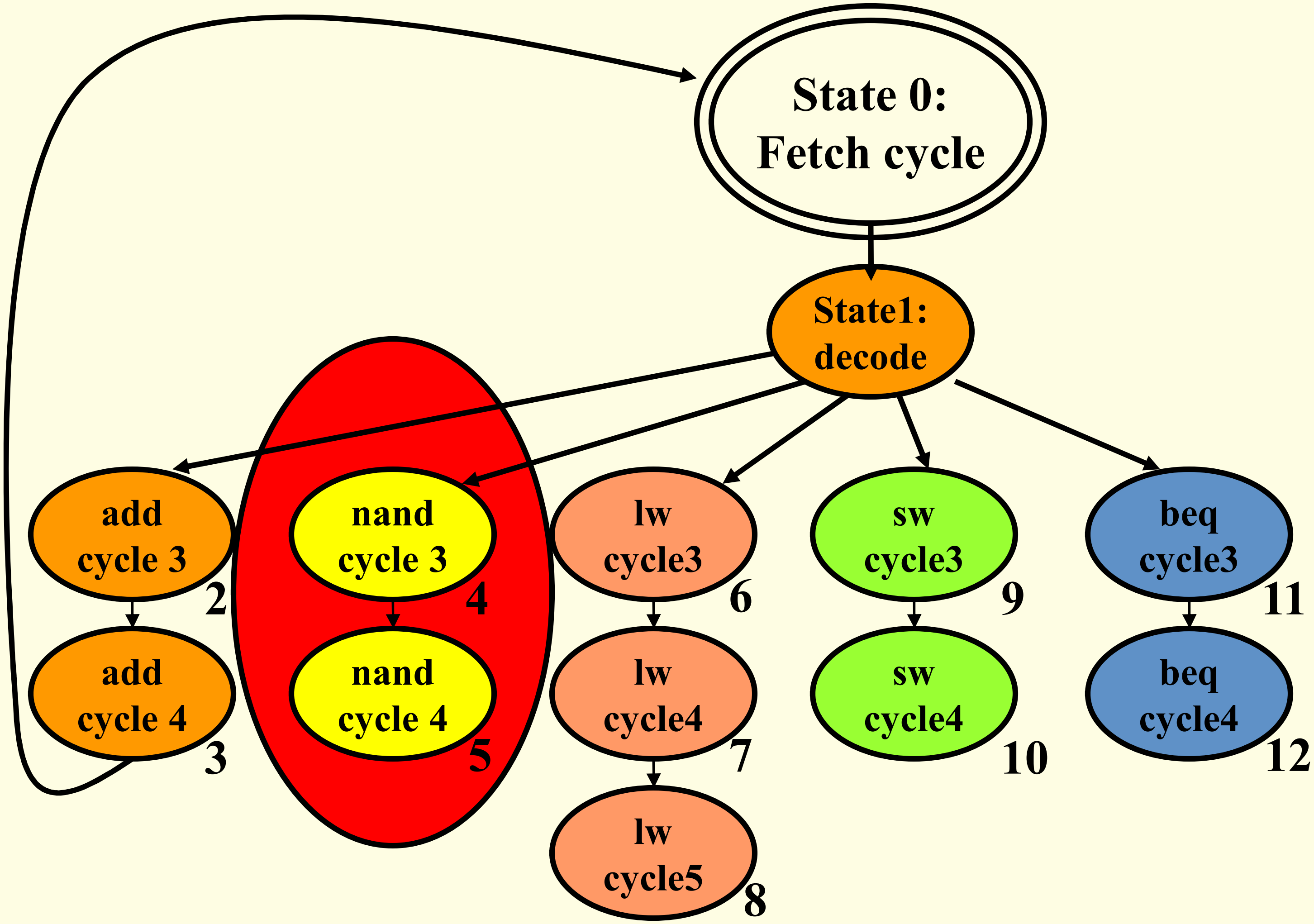
Send control signal to address MUX to select dest and to data MUX to select ALU output, then send write enable to register file.



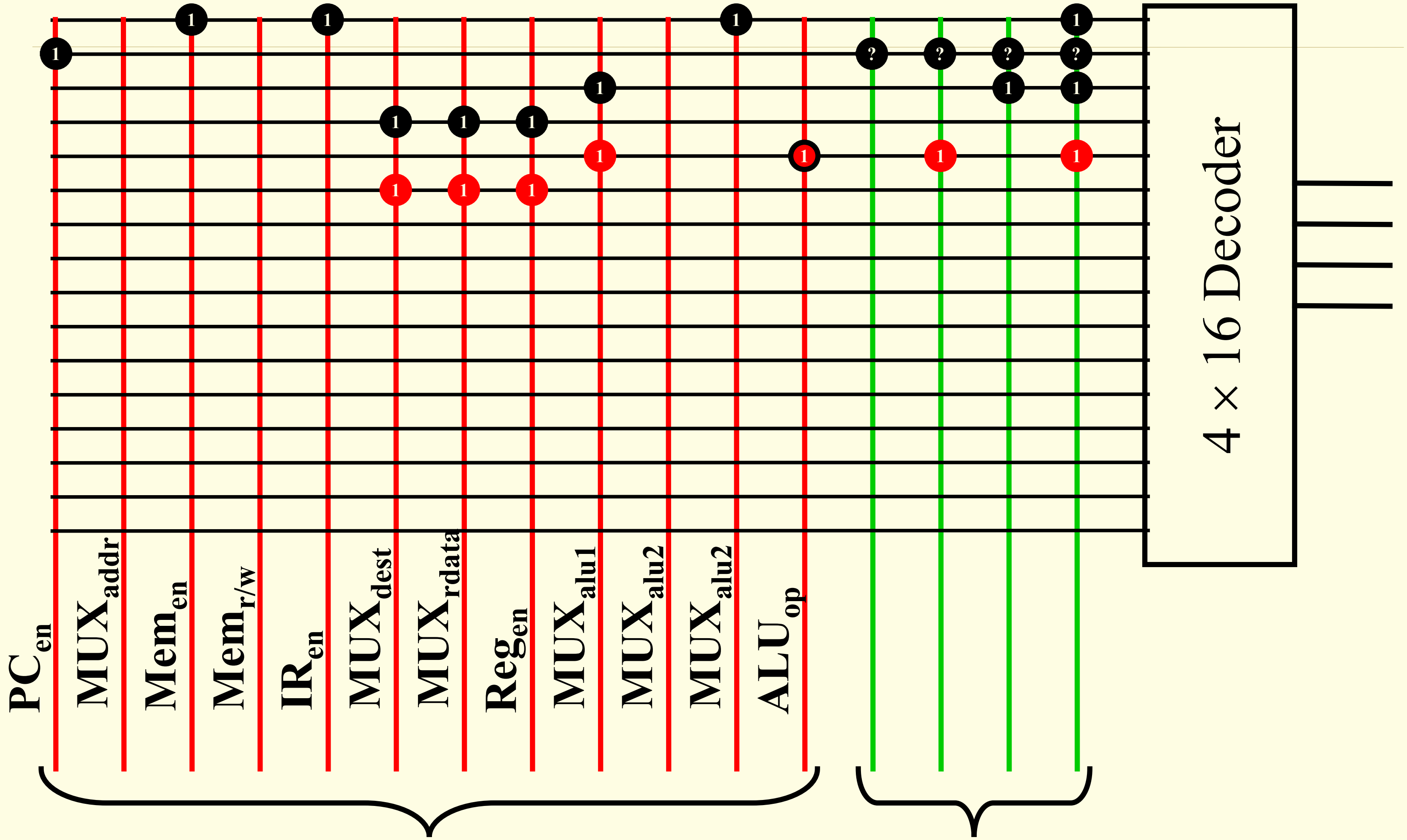
Building the Control Rom



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



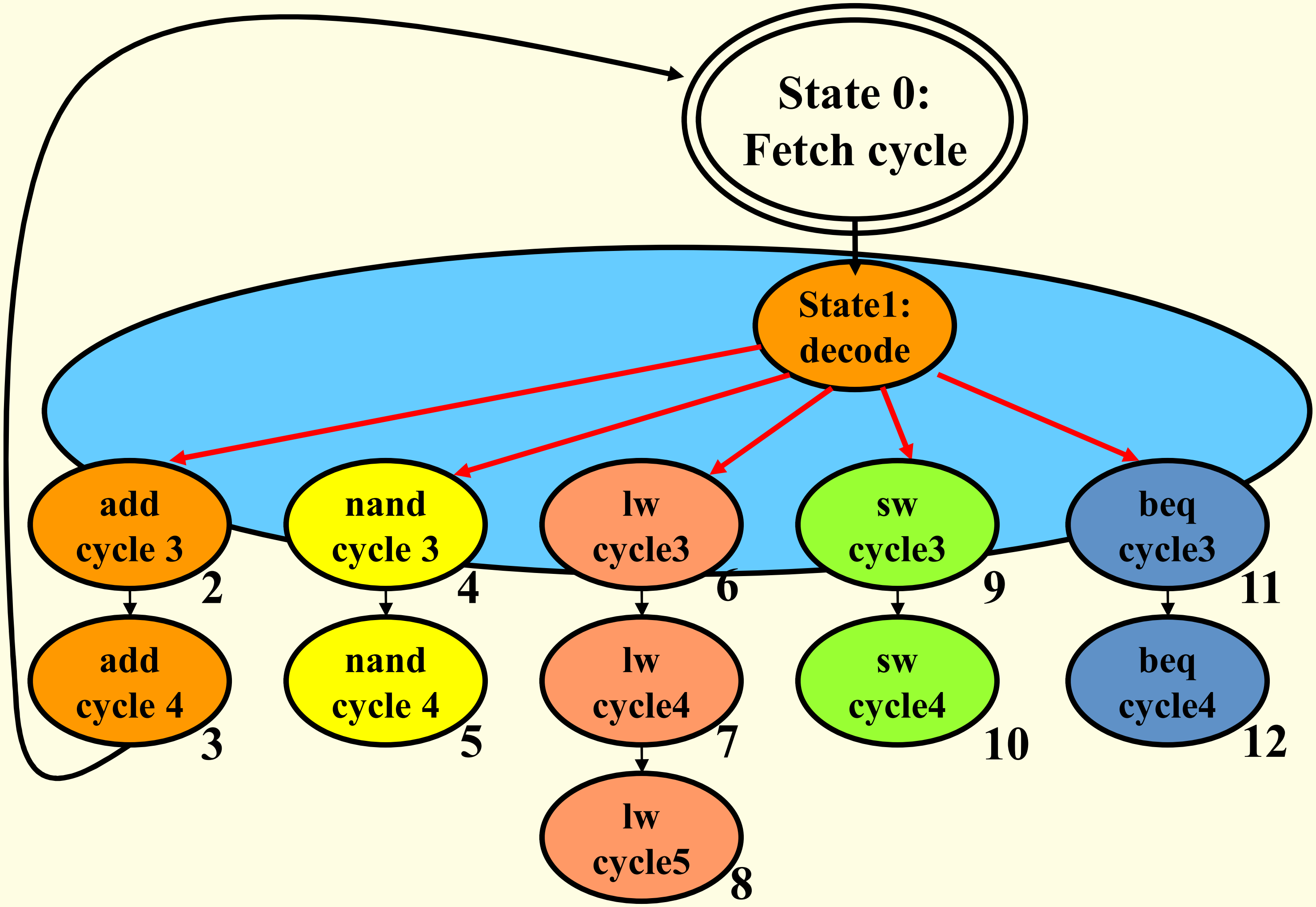
Control Rom for nand (4 and 5)



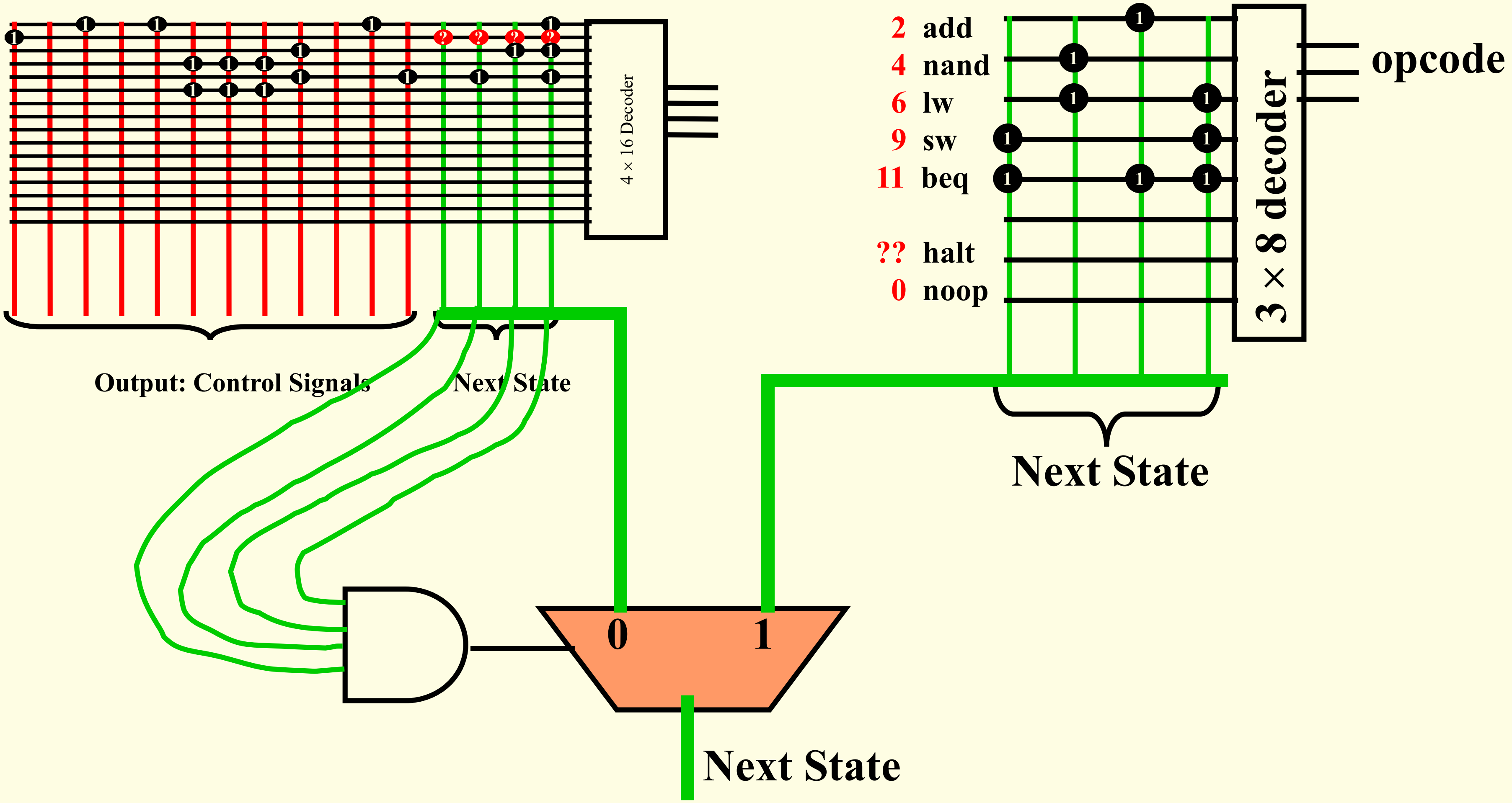
Output: Control Signals

Next State

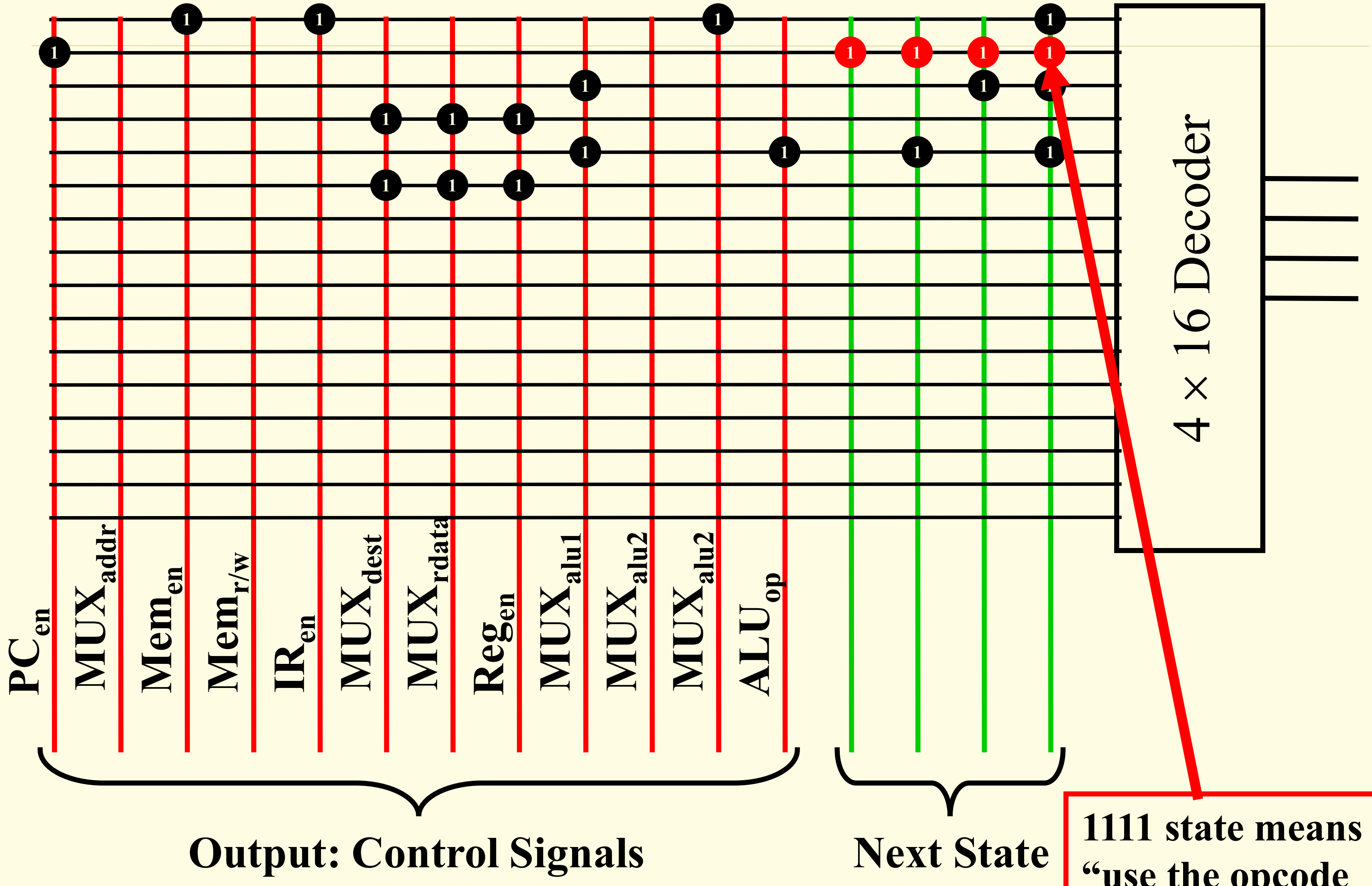
What about the transition from state 1?



Complete transition function circuit

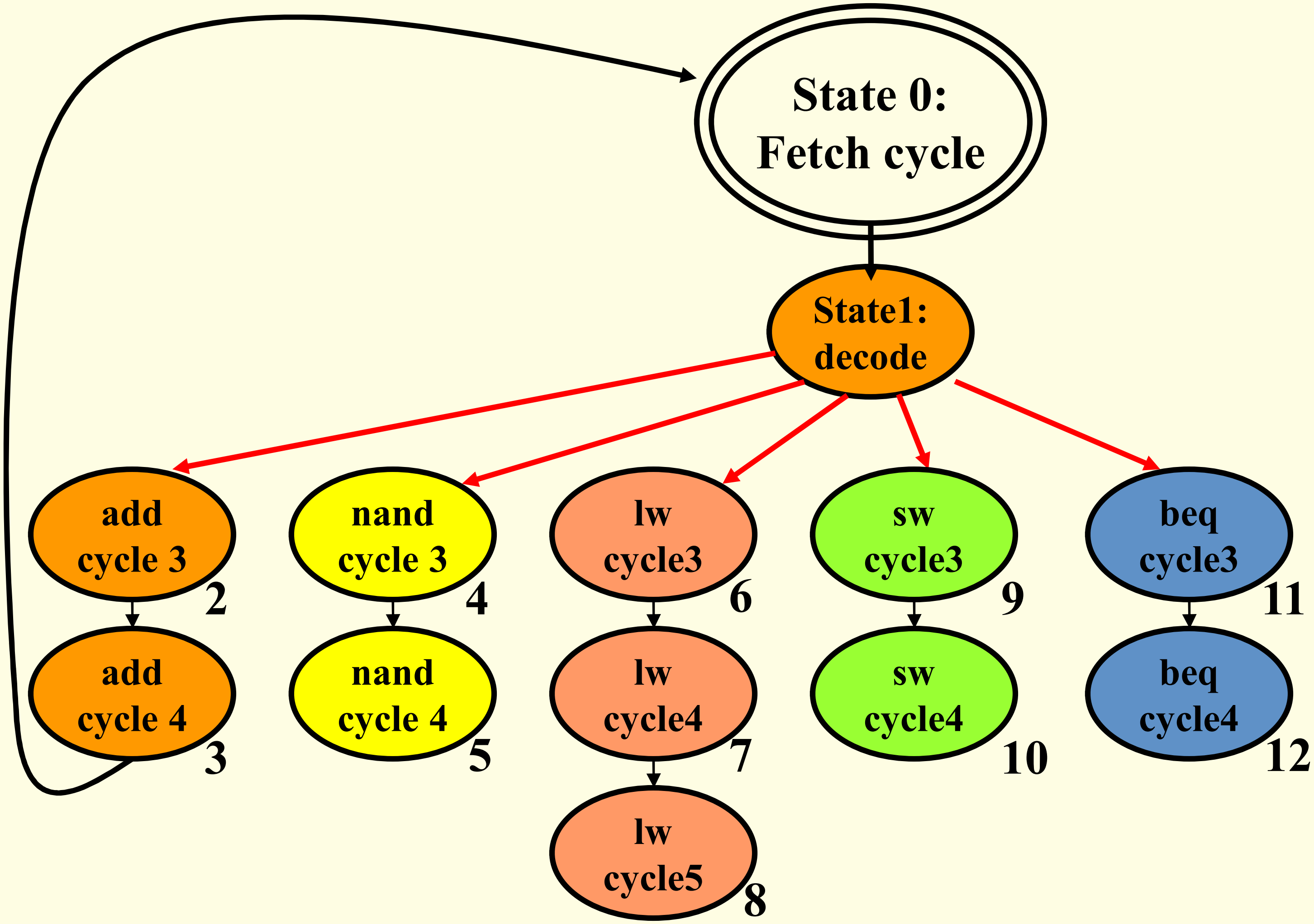


Control Rom (use of 1111 state)



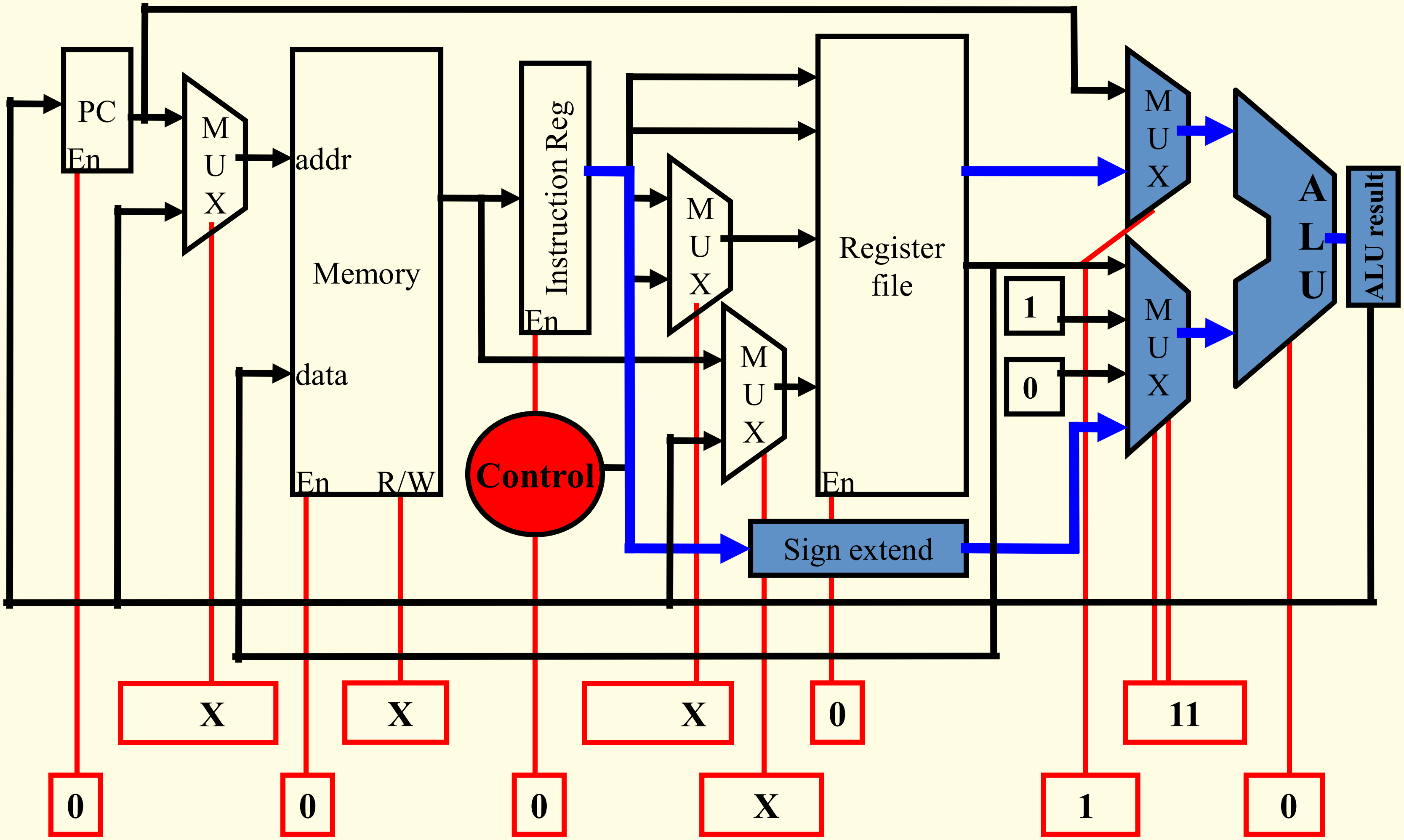
**1111 state means
“use the opcode
to decide next state”**

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

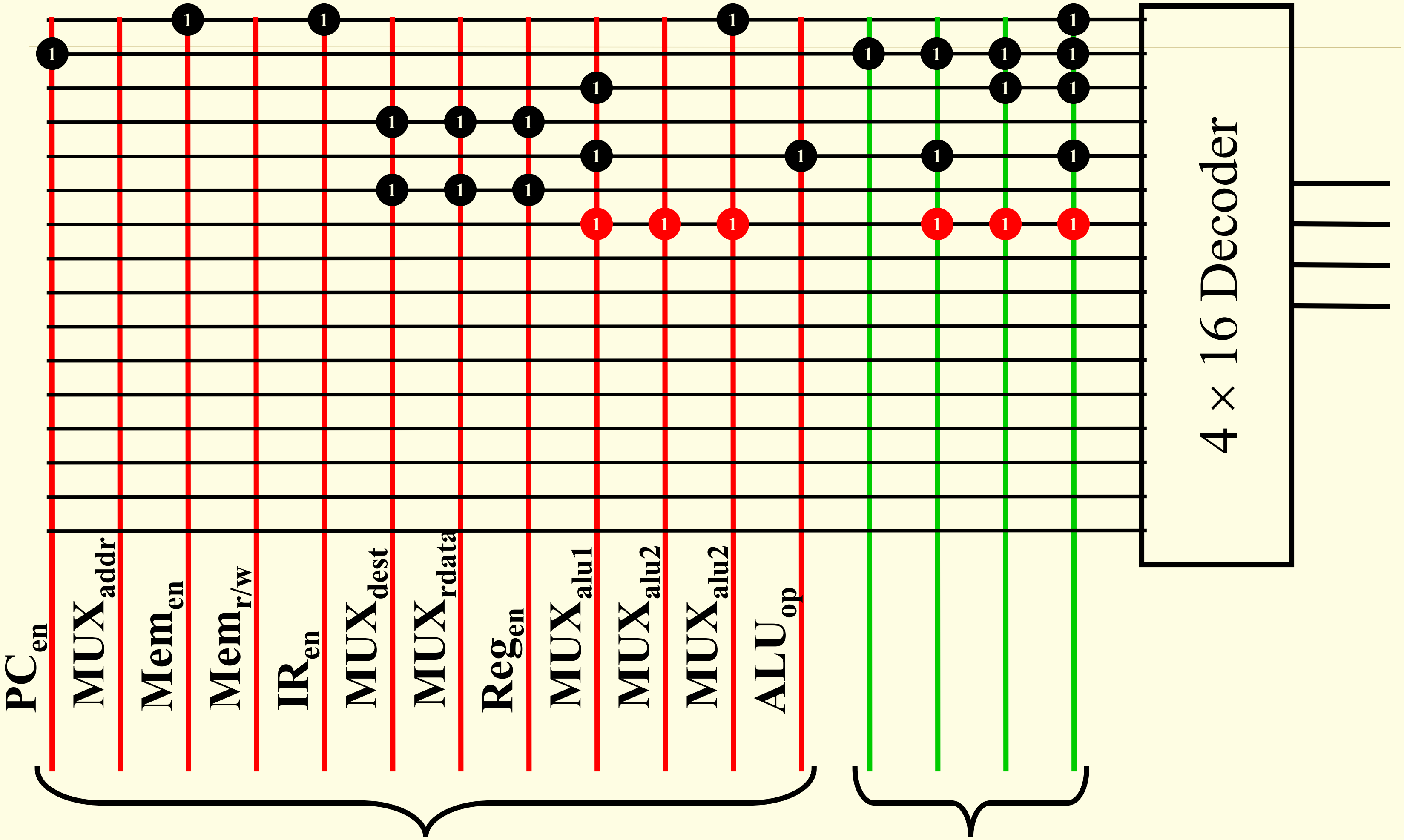


State 6: LW cycle 3

Calculate address for memory reference



Control Rom (lw cycle 3)

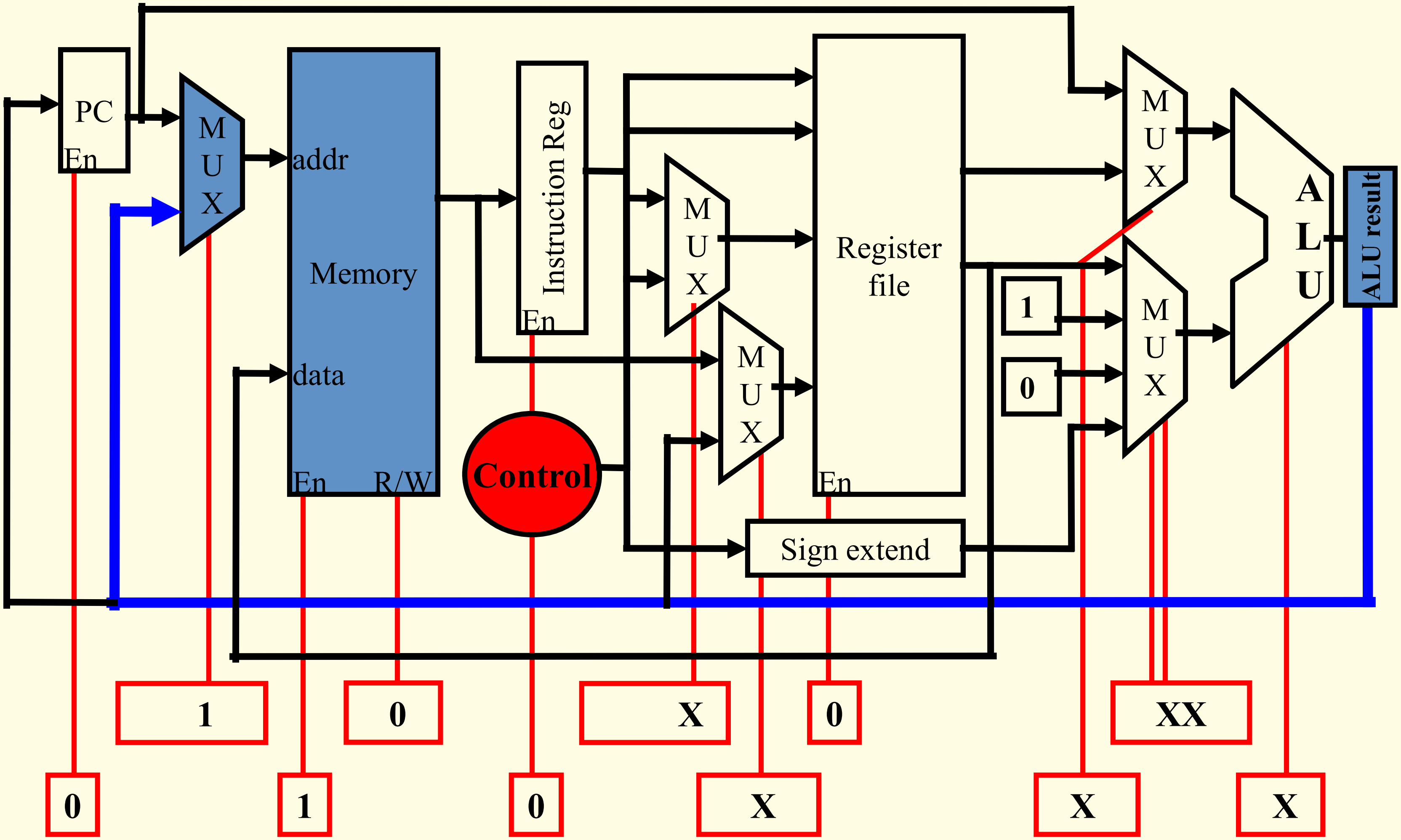


Output: Control Signals

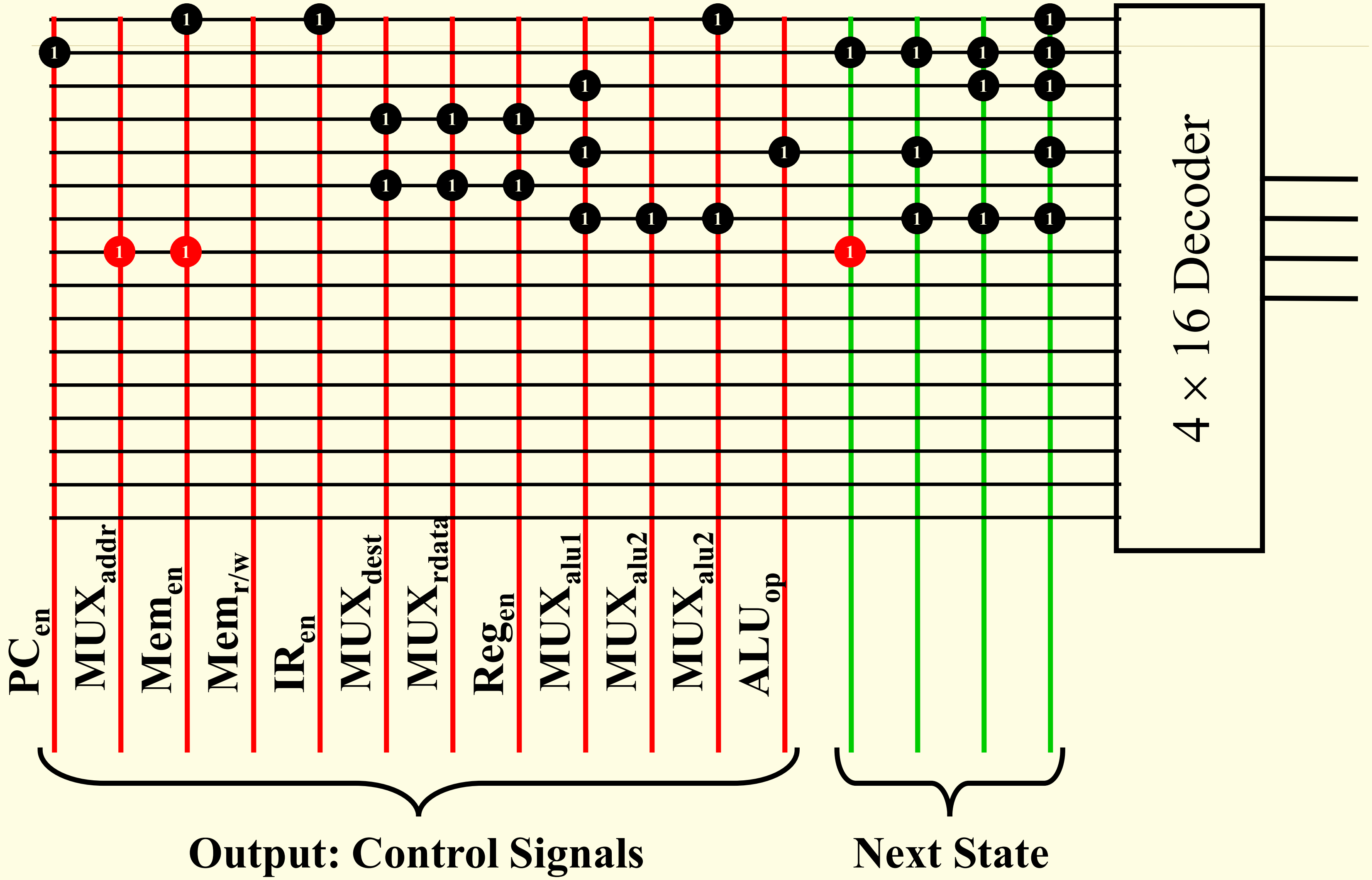
Next State

State 7: LW cycle 4

Read memory location



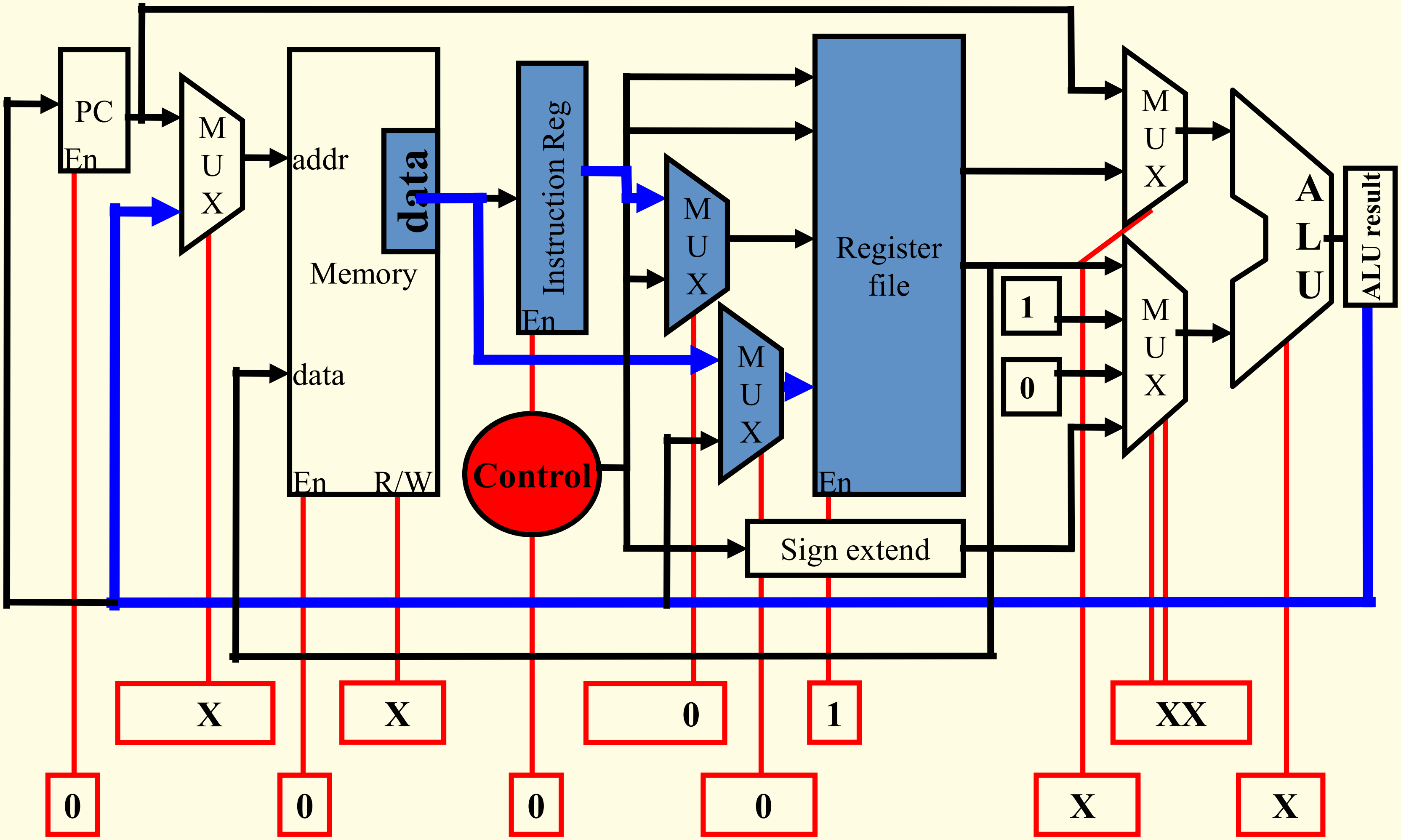
Control Rom (lw cycle 4)



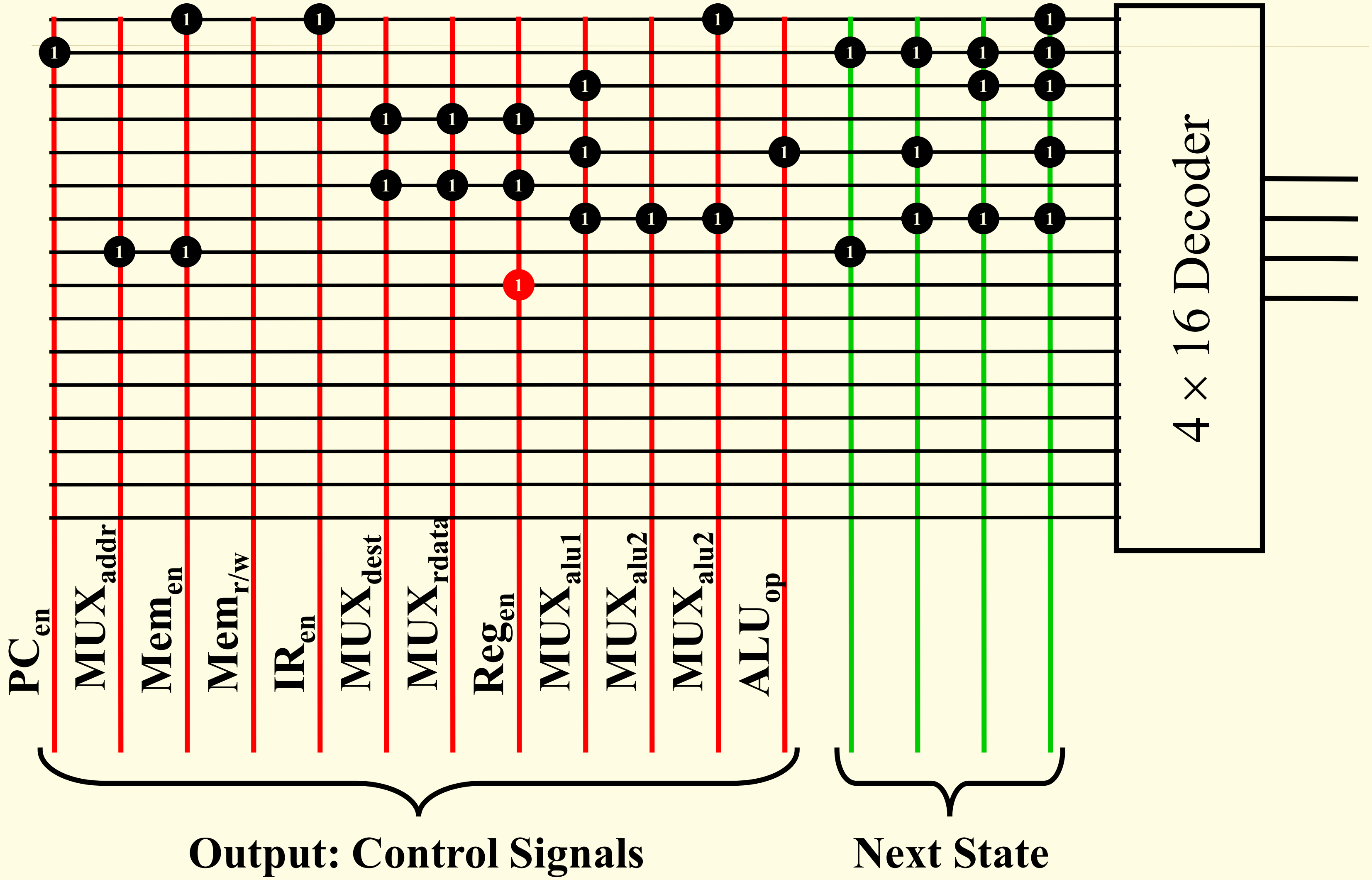
PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_{en}	MUX_{dest}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			

State 8: LW cycle 5

Write memory value to register file

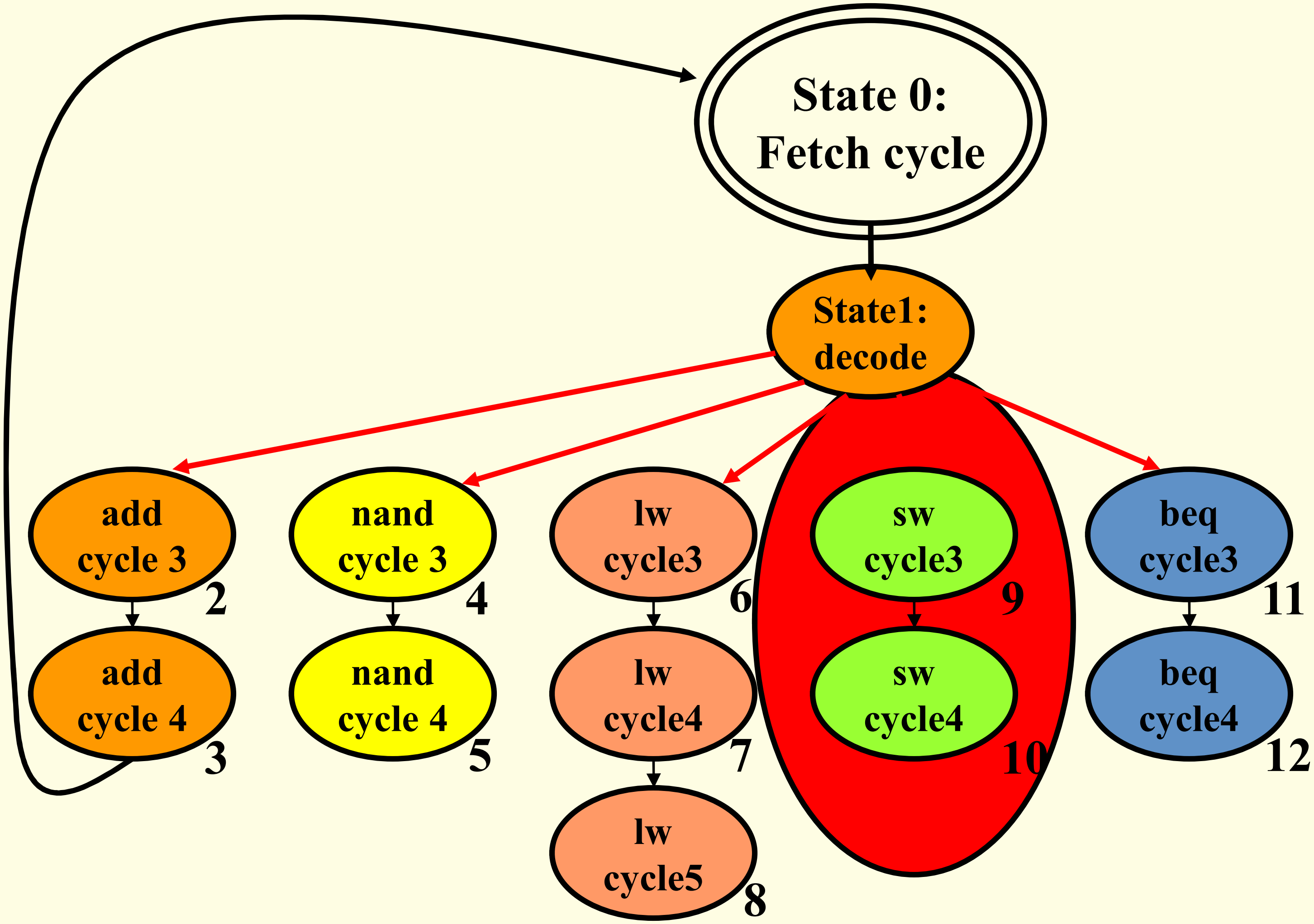


Control Rom (lw cycle 5)

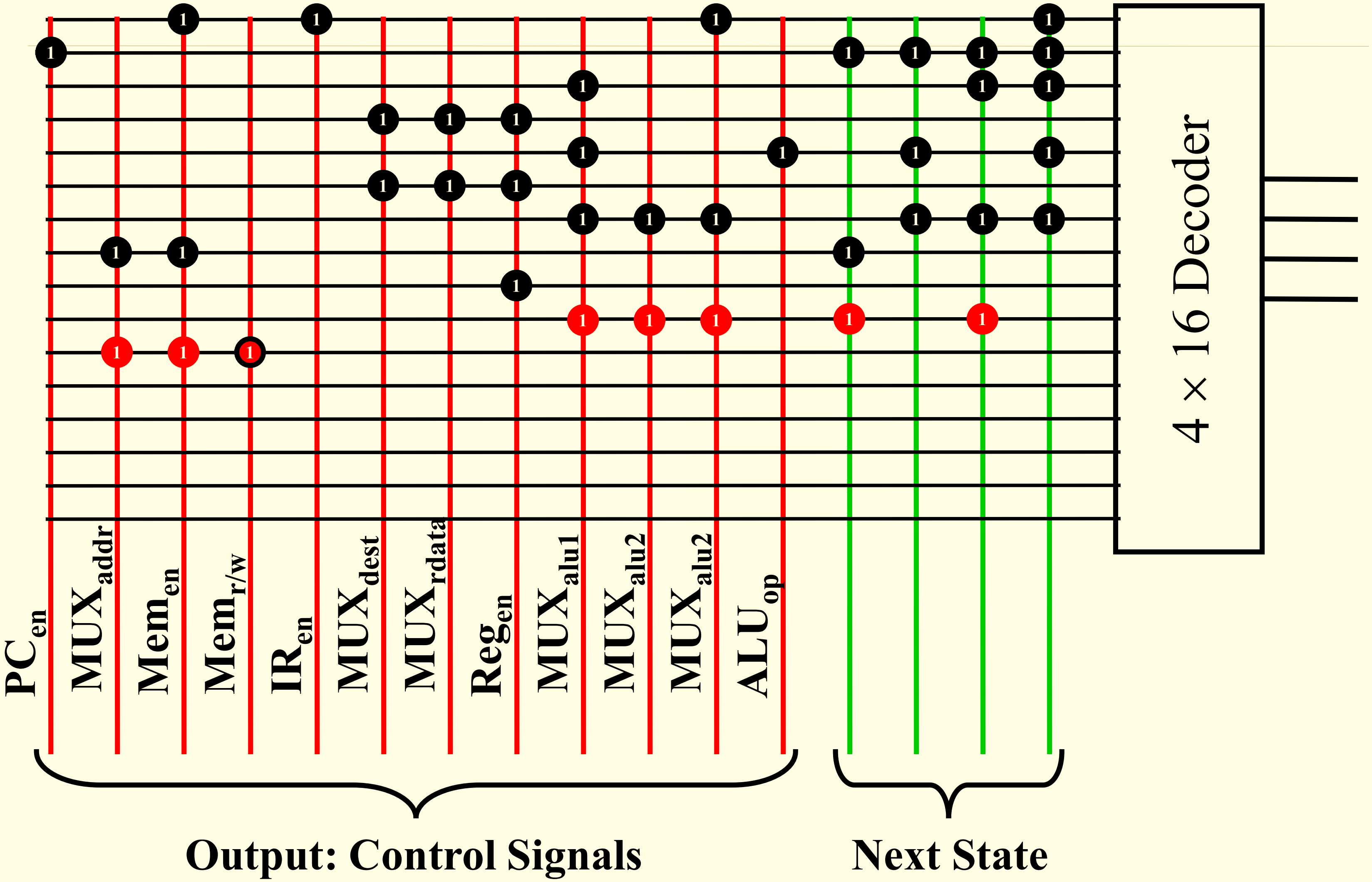


PC_{en}	MUX_{addr}	Mem_{en}	$Mem_{r/w}$	IR_{en}	MUX_{dest}	MUX_{rdata}	Reg_{en}	MUX_{alu1}	MUX_{alu2}	MUX_{alu2}	ALU_{op}				
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			
							1								

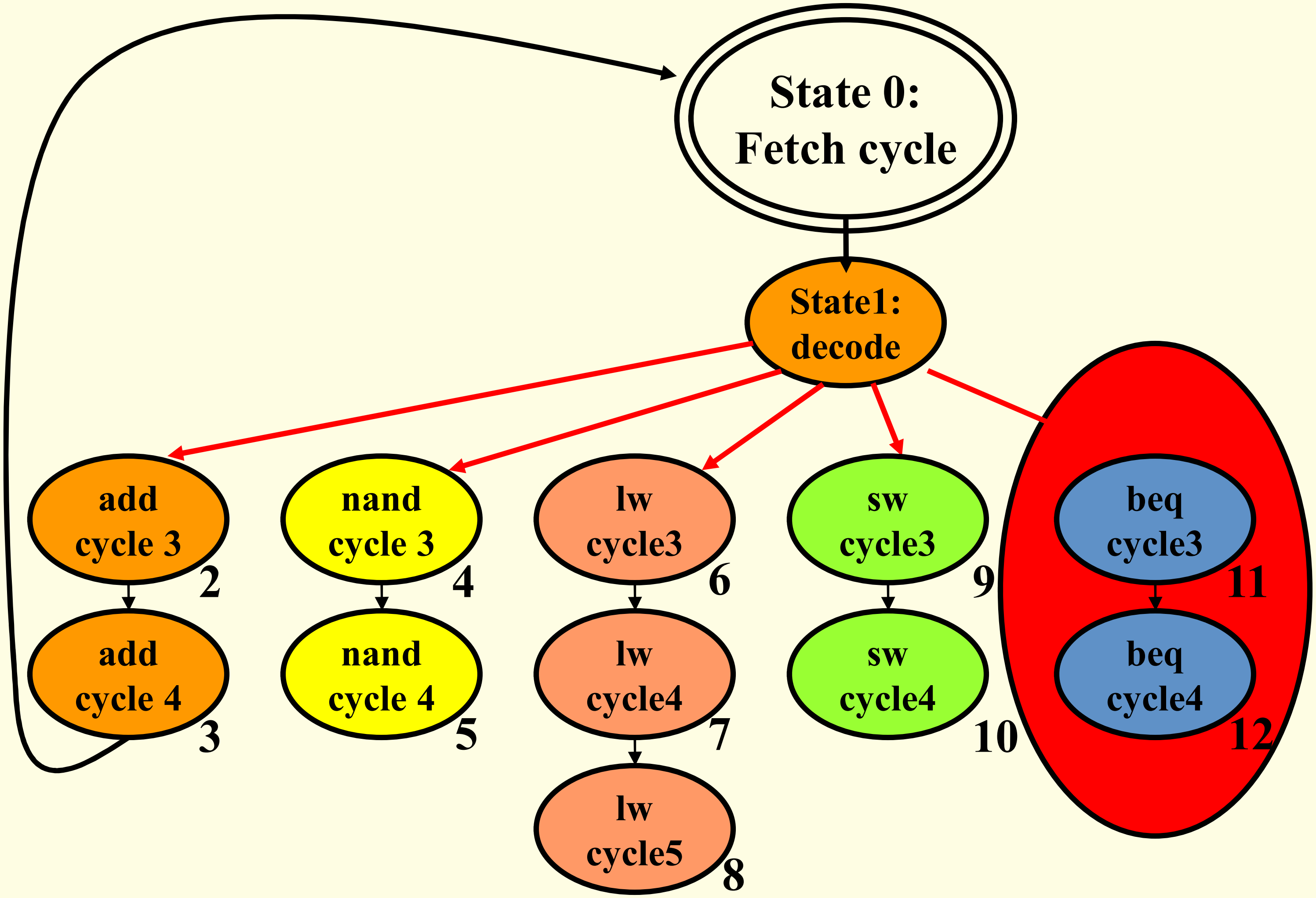
Return to State 0: Fetch cycle to execute the next instruction



Control Rom (sw cycles 3 and 4)

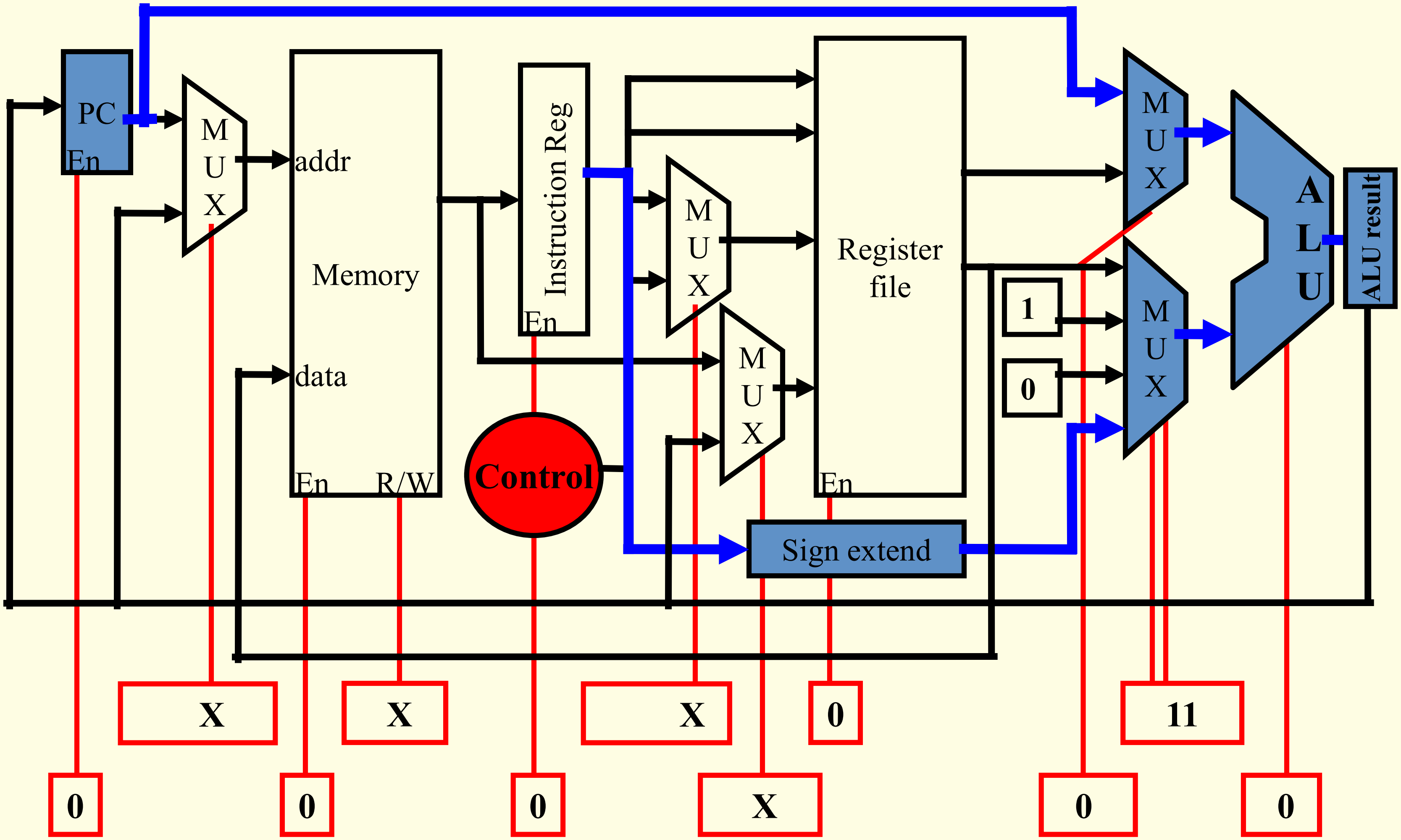


Return to State 0: Fetch cycle to execute the next instruction

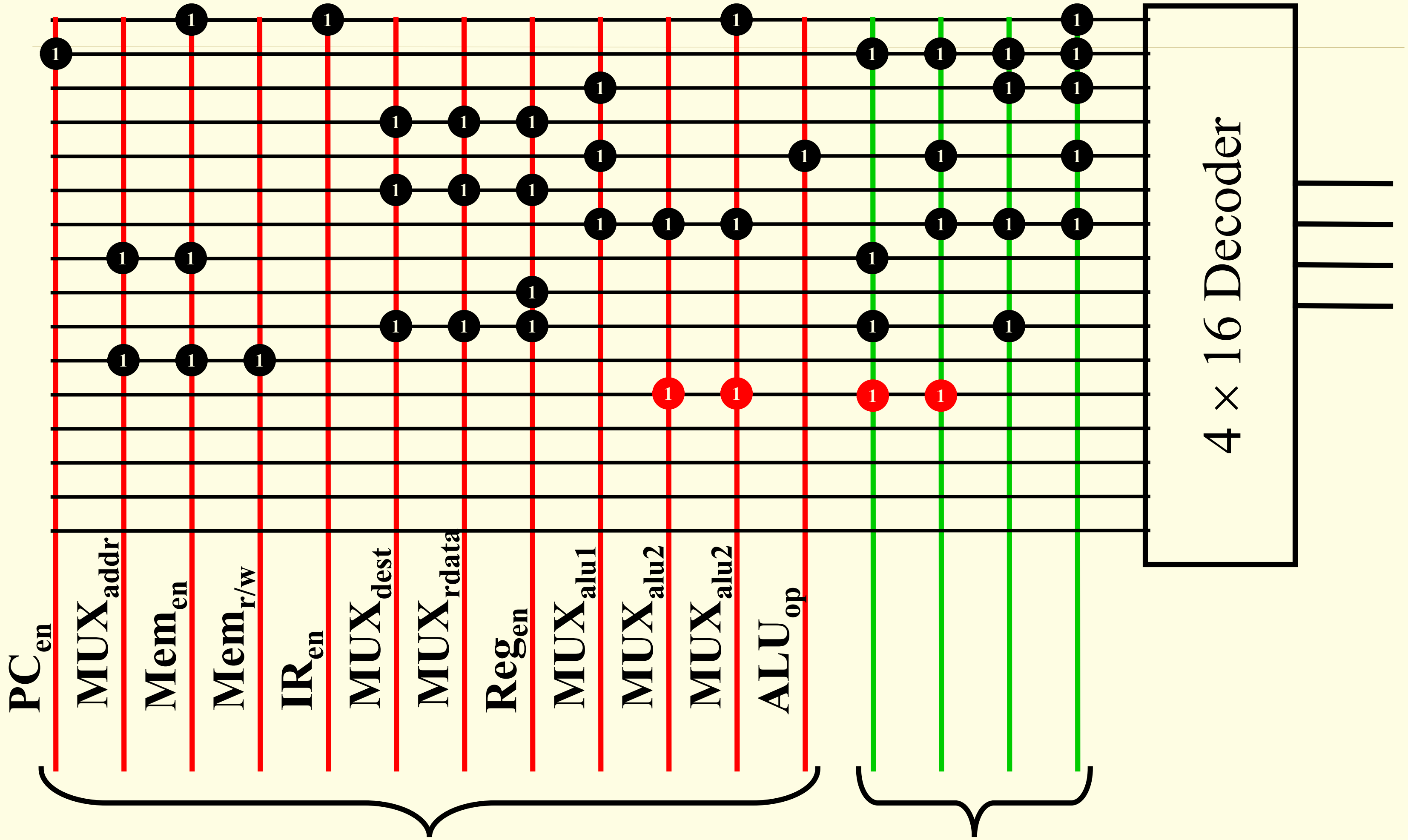


State 6: beq cycle 3

Calculate target address for branch



Control Rom (beq cycle 3)

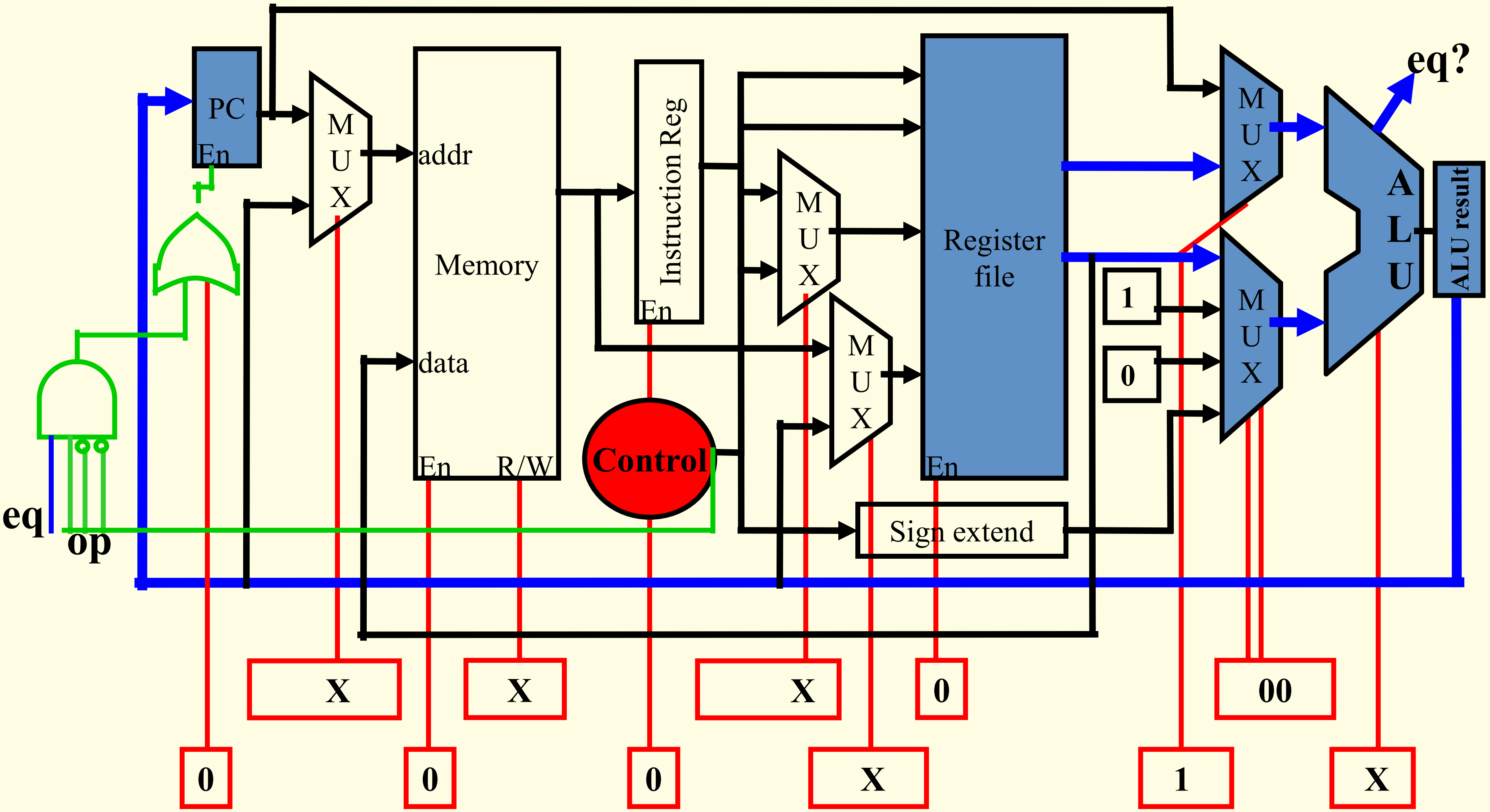


Output: Control Signals

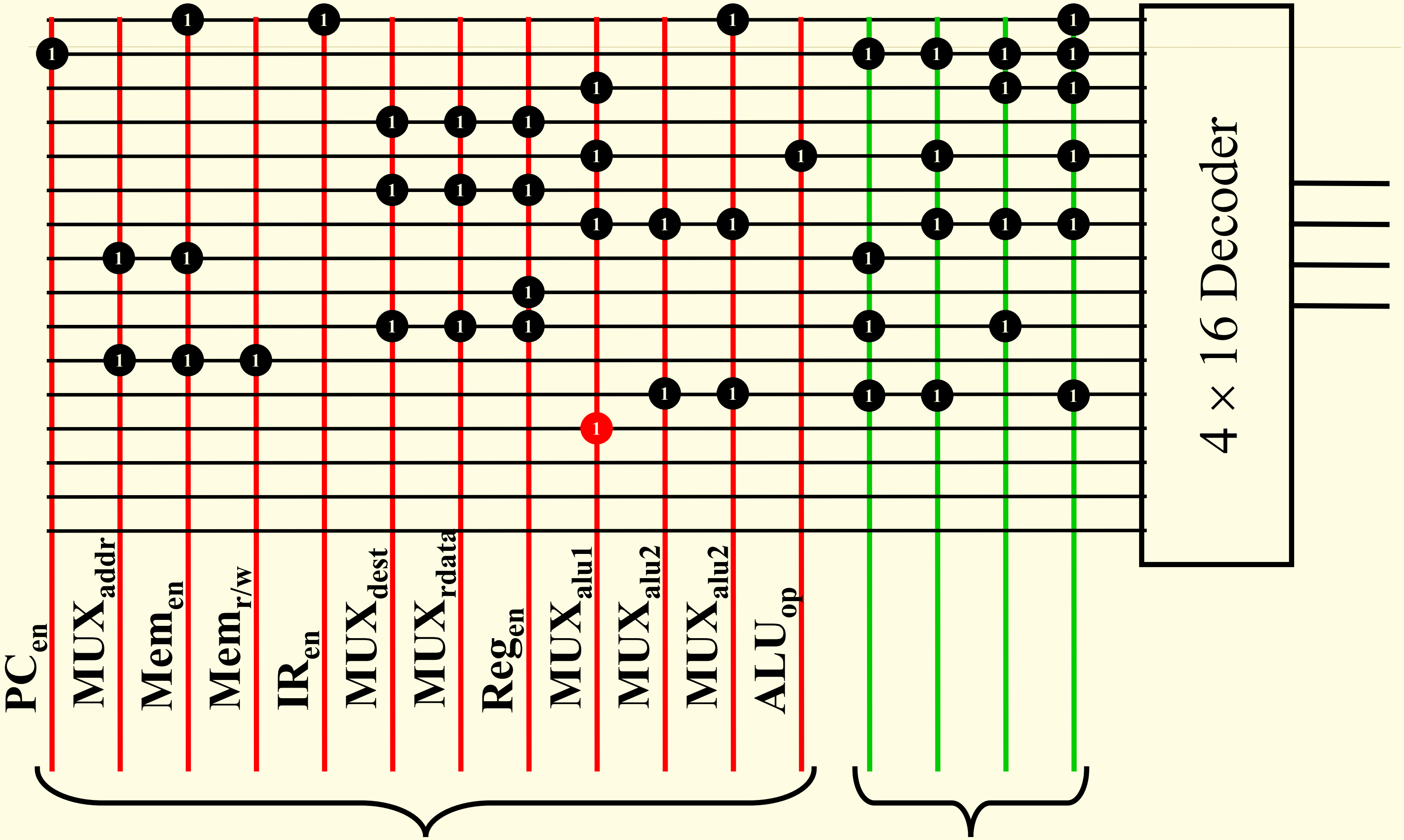
Next State

State 13: beq cycle 4 (incorrect)

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



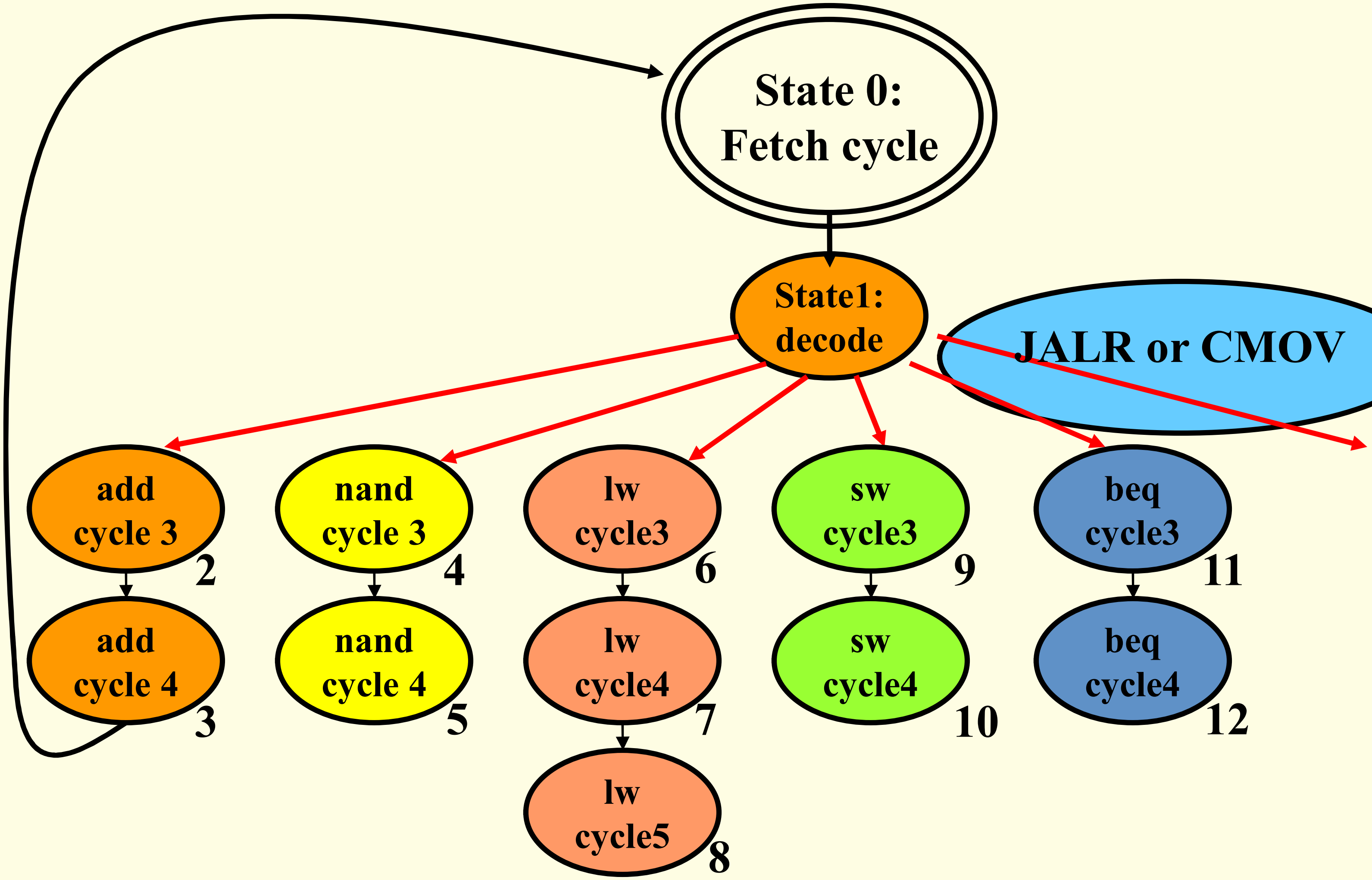
Control Rom (beq cycle 4)



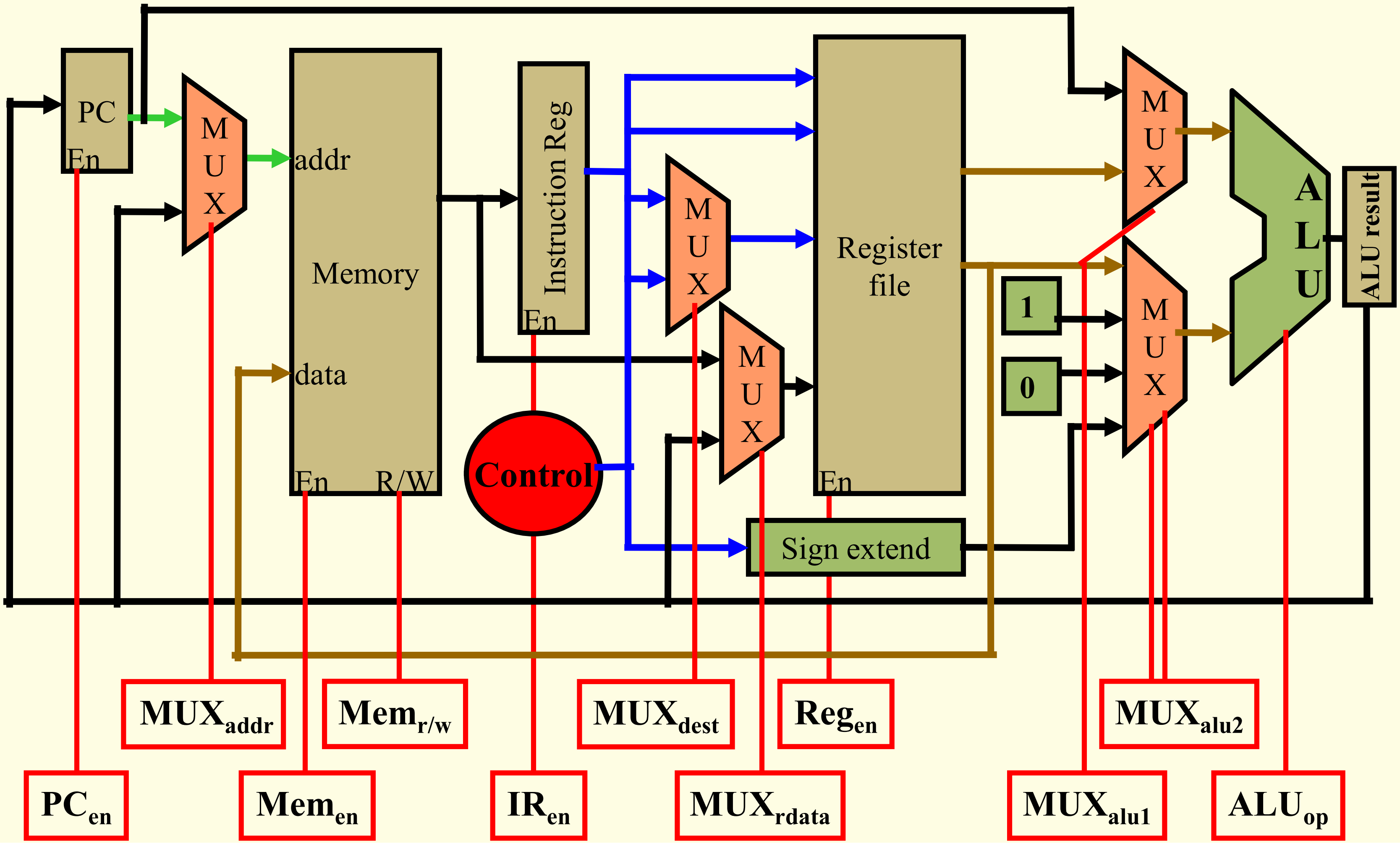
PC$_{en}$	MUX$_{addr}$	Mem$_{en}$	Mem$_{r/w}$	IR$_{en}$	MUX$_{dest}$	MUX$_{rdata}$	Reg$_{en}$	MUX$_{alu1}$	MUX$_{alu2}$	MUX$_{alu2}$	ALU$_{op}$	Next State			
		1		1						1					1
1												1	1	1	1
								1						1	1
					1	1	1								
								1			1		1		1
					1	1	1								
								1	1	1			1	1	1
	1	1										1			
							1								
					1	1	1					1		1	
	1	1	1												
									1	1		1	1		1
								1							

Output: Control Signals **Next State**

OK, what about the JALR or CMOV instructions?



Multi-cycle MIPS-32 Datapath


PC
En
M U X
addr
Memory
data
En
R/W
Instruction Reg
En
Control
M U X
M U X
Register file
En
Sign extend
M U X
1
0
M U X
A L U
ALU result
MUX$_{addr}$
Mem$_{r/w}$
MUX$_{dest}$
Reg$_{en}$
MUX$_{alu2}$
PC$_{en}$
Mem$_{en}$
IR$_{en}$
MUX$_{rdata}$
MUX$_{alu1}$
ALU$_{op}$