Poll: How cheaply do you think we can build a vending machine controller?

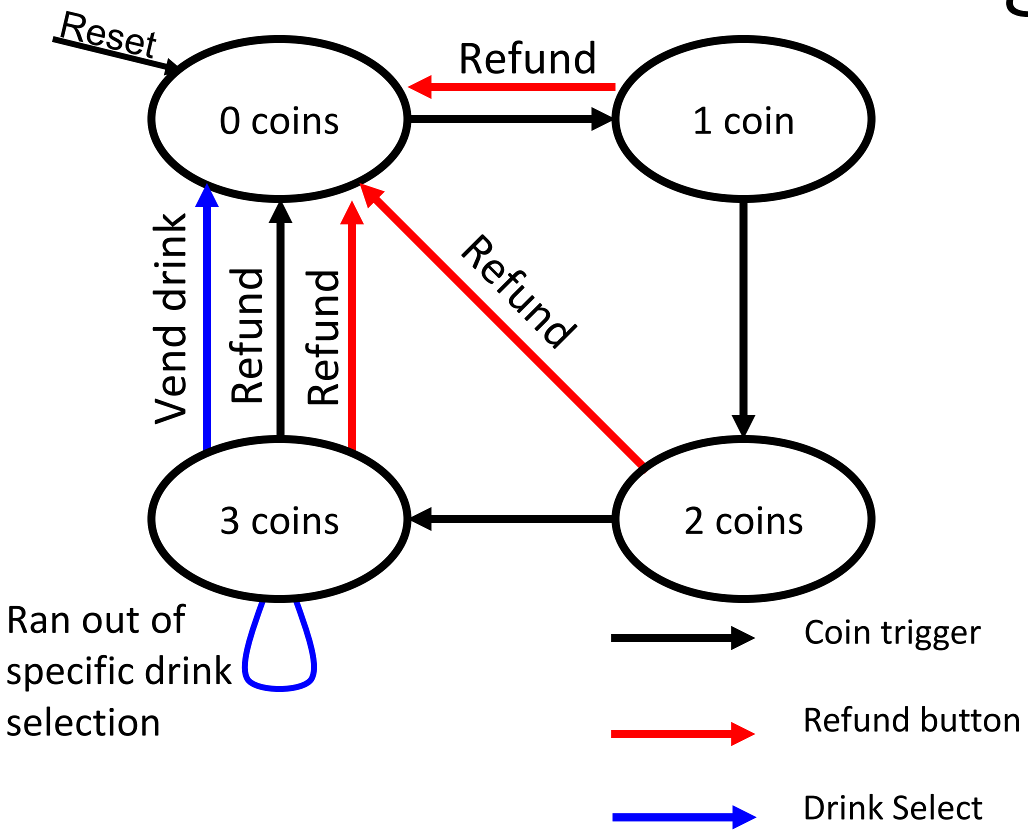
EECS 370 - Lecture 11

Single & 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

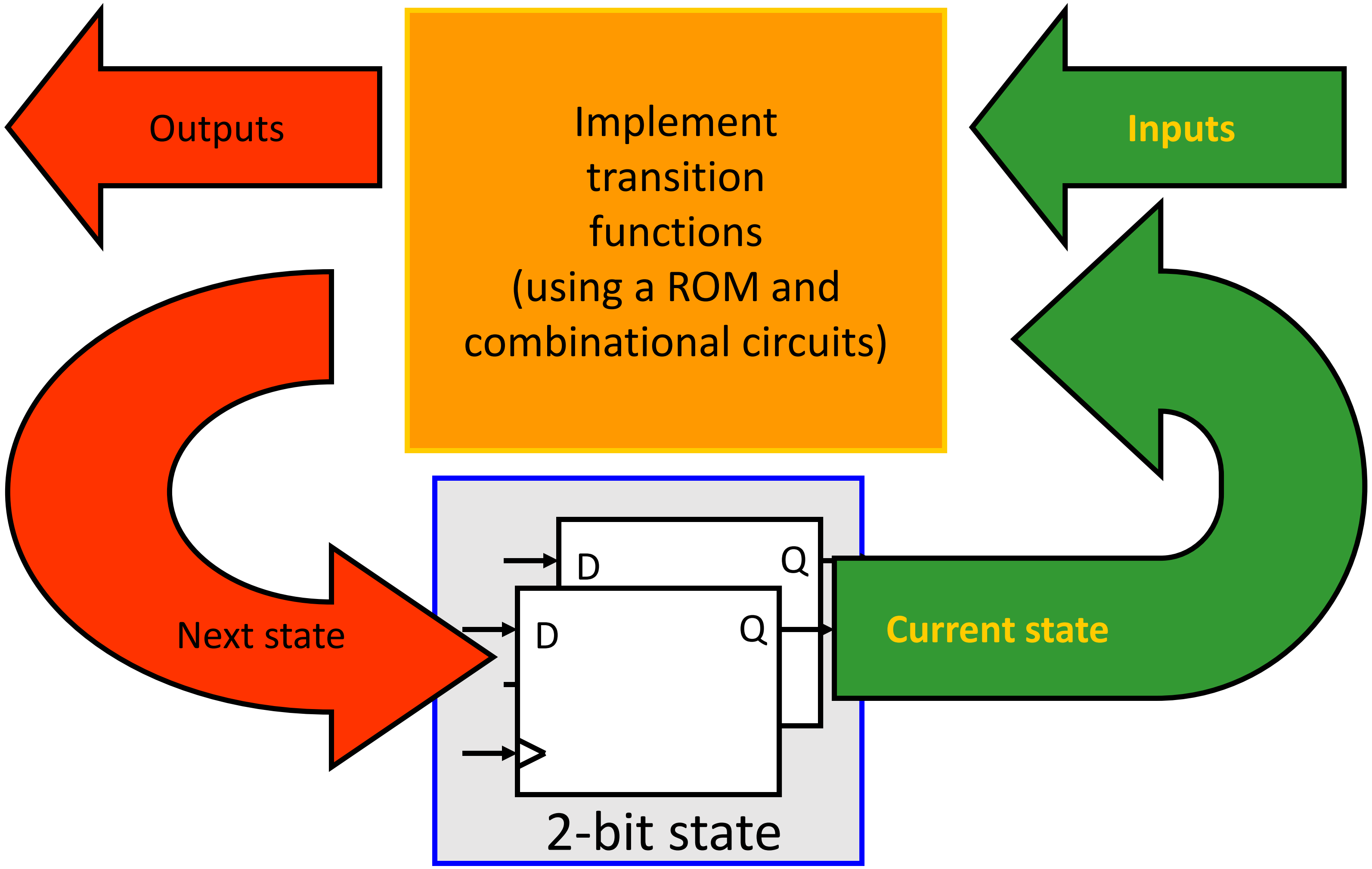
Reminder: FSM for Vending Machine



Is this a Mealy or Moore Machine?

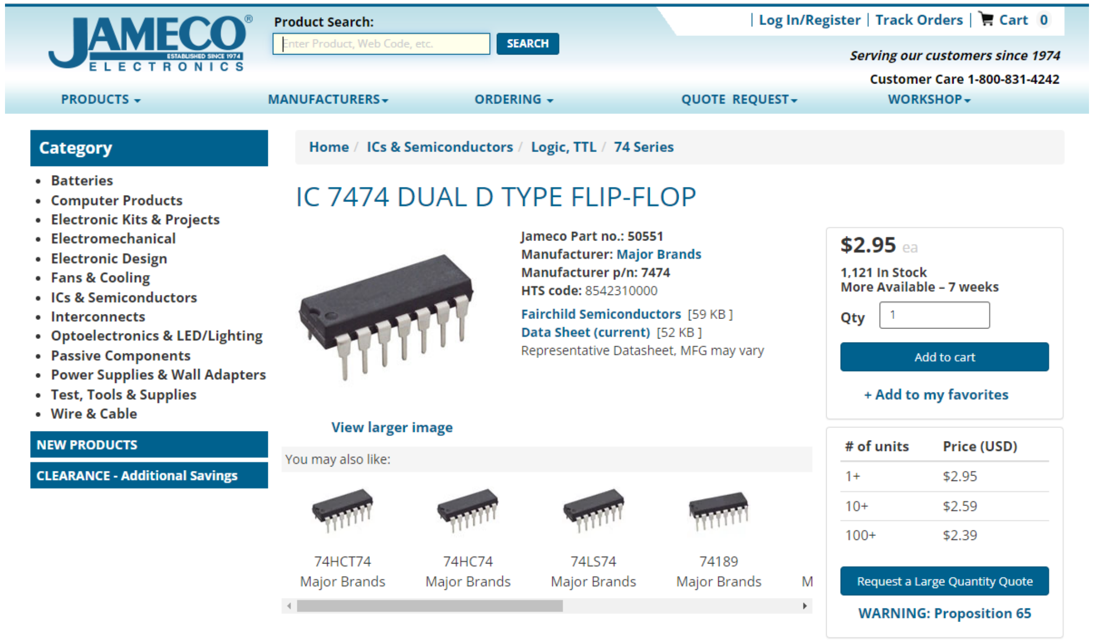
This is Mealy: Mealy output is based on current state *AND* input

Implementing an FSM



Implementing an FSM

- Let's see how cheap we can build this vending machine controller!
- [Jameco.com](http://jameco.com) sells electronic chips we can use
  - D-Flip-flops: $3, includes several in one package
- For custom combinational circuits, would need to design and send to a fabrication facility
  - Thousands or millions of dollars!!
  - Alternative?

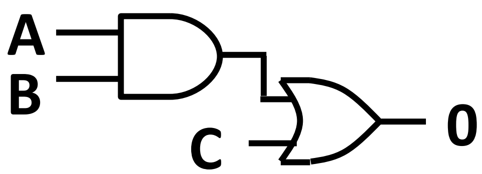


Implementing Combinational Logic

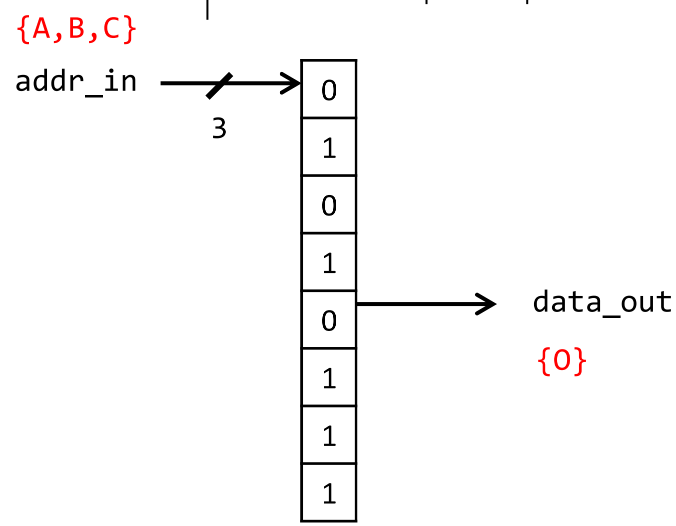
A	B	C	O
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

If I have a truth table:

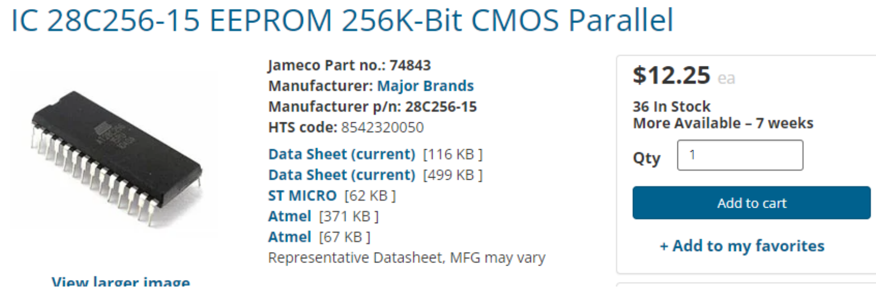
- I can either implement this using combinational logic:



- …or I could literally just store the entire truth table in a memory and just "index" it by treating the input as a number!
  - Can be implemented cheaply using "Read Only Memories", or "ROMS"



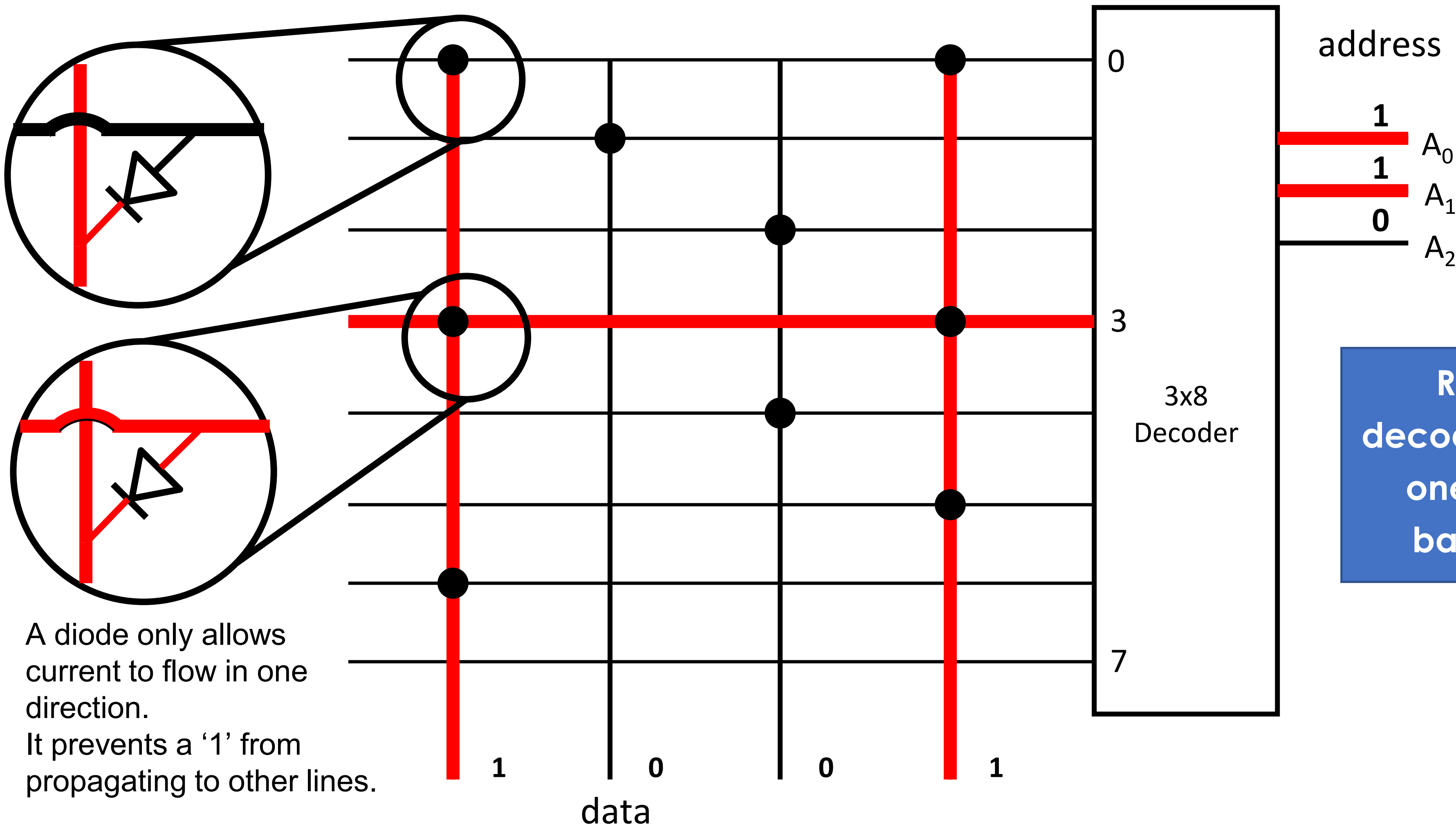
ROMs and PROMs



- Read Only Memory (ROM)
  - Array of memory values that are constant
  - Non-volatile (doesn’t need constant power to save values)
- Programmable Read Only Memory
  - Array of memory values that can be written exactly once
- Electronically Erasable PROM (EEPROM)
  - Can write to memory, deploy in field
  - Use special hardware to reset bits if need to update

- 256 KBs of EEPROM costs ~$10 on Jameco
  - Much better then spending thousands on design costs unless we're gonna make **tons** of these

8-entry 4-bit ROM



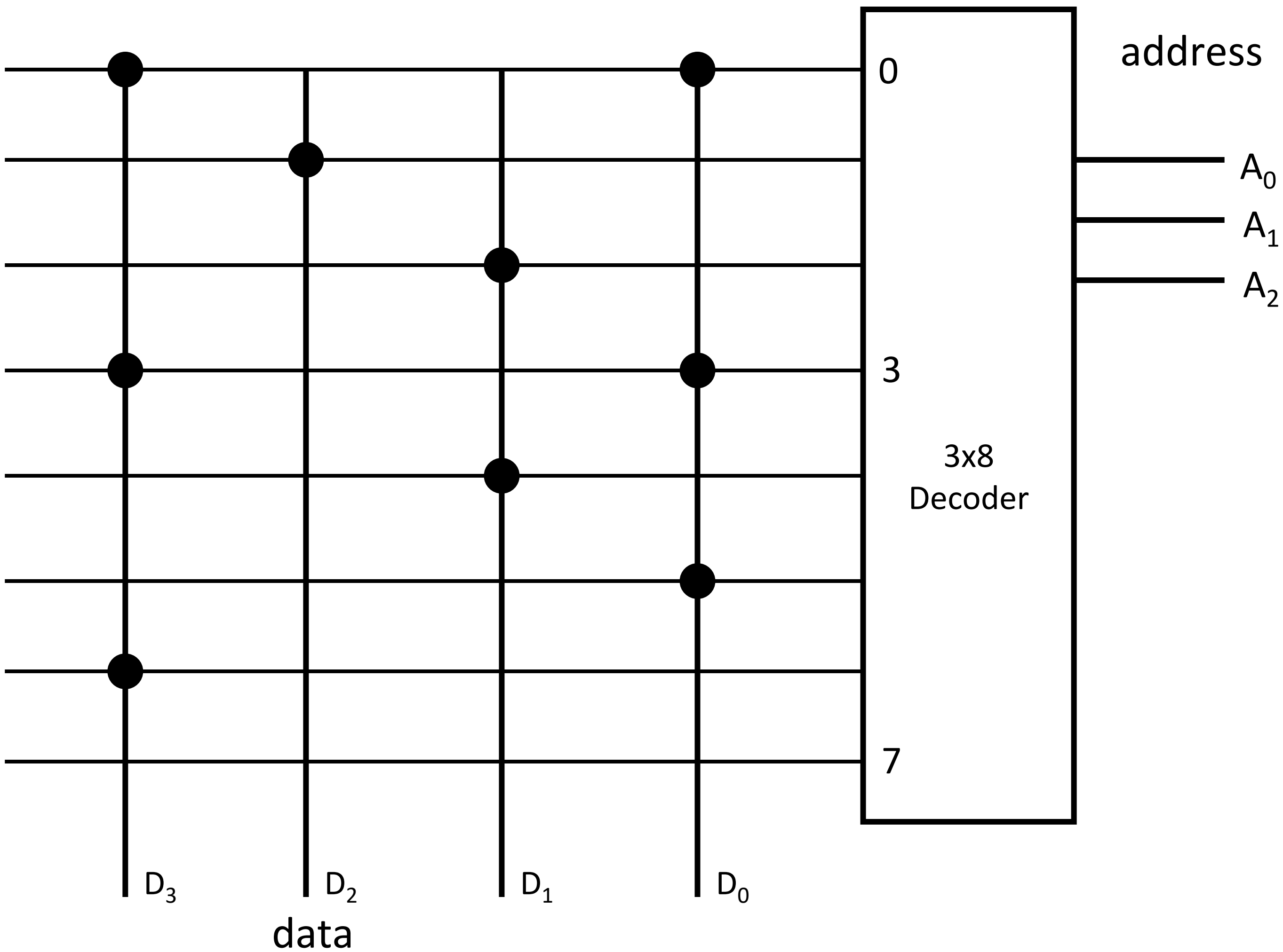
Reminder: A decoder sets exactly one output high based on input

- A diode only allows current to flow in one direction.
- It prevents a ‘1’ from propagating to other lines.

8-entry 4-bit ROM

Input	Output
000	
001	
010	
011	
100	
101	
110	
111	

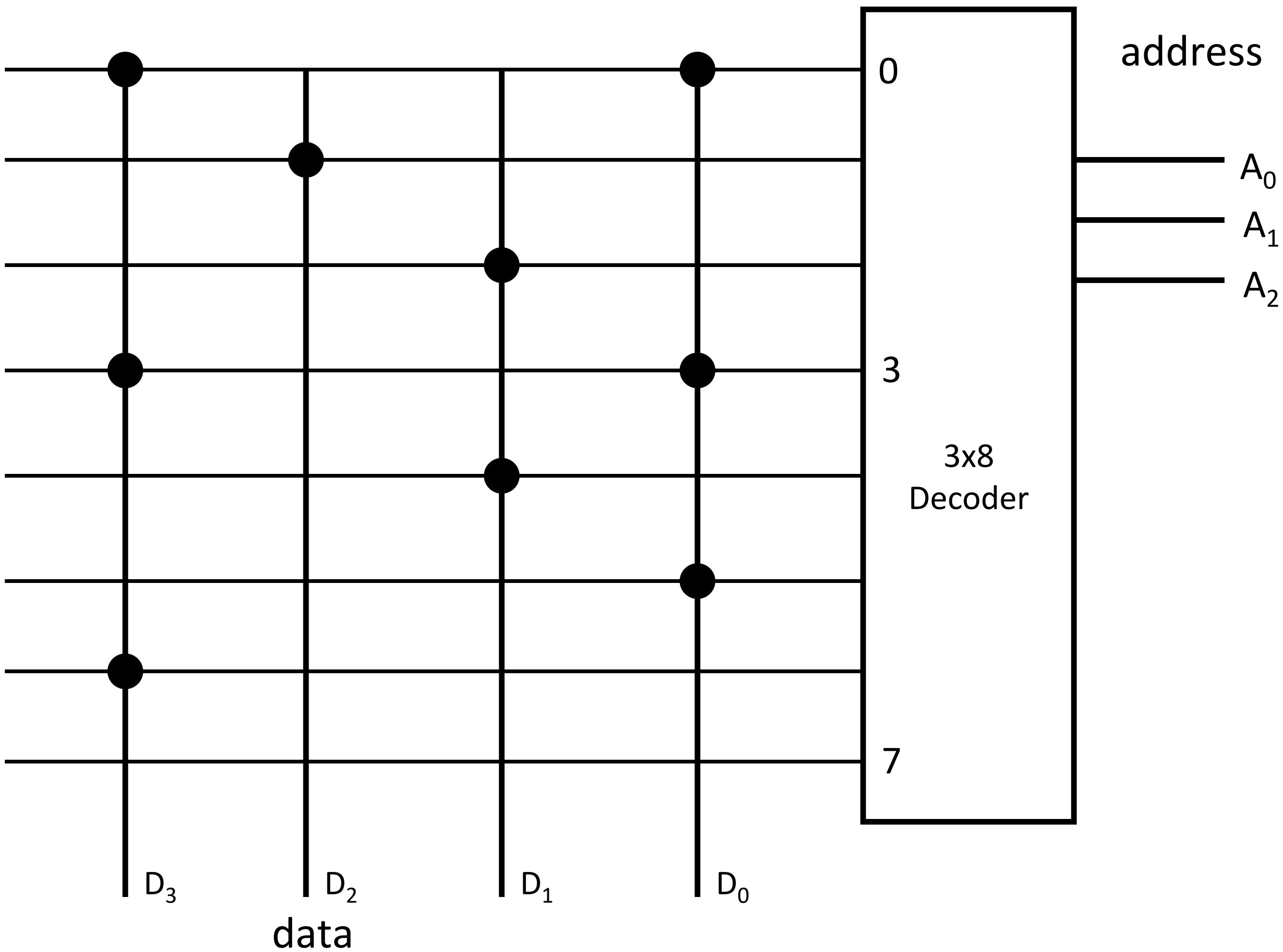
This ROM corresponds to this truth table



8-entry 4-bit ROM

Input	Output
000	1001
001	0100
010	0010
011	1001
100	0010
101	0001
110	1000
111	0000

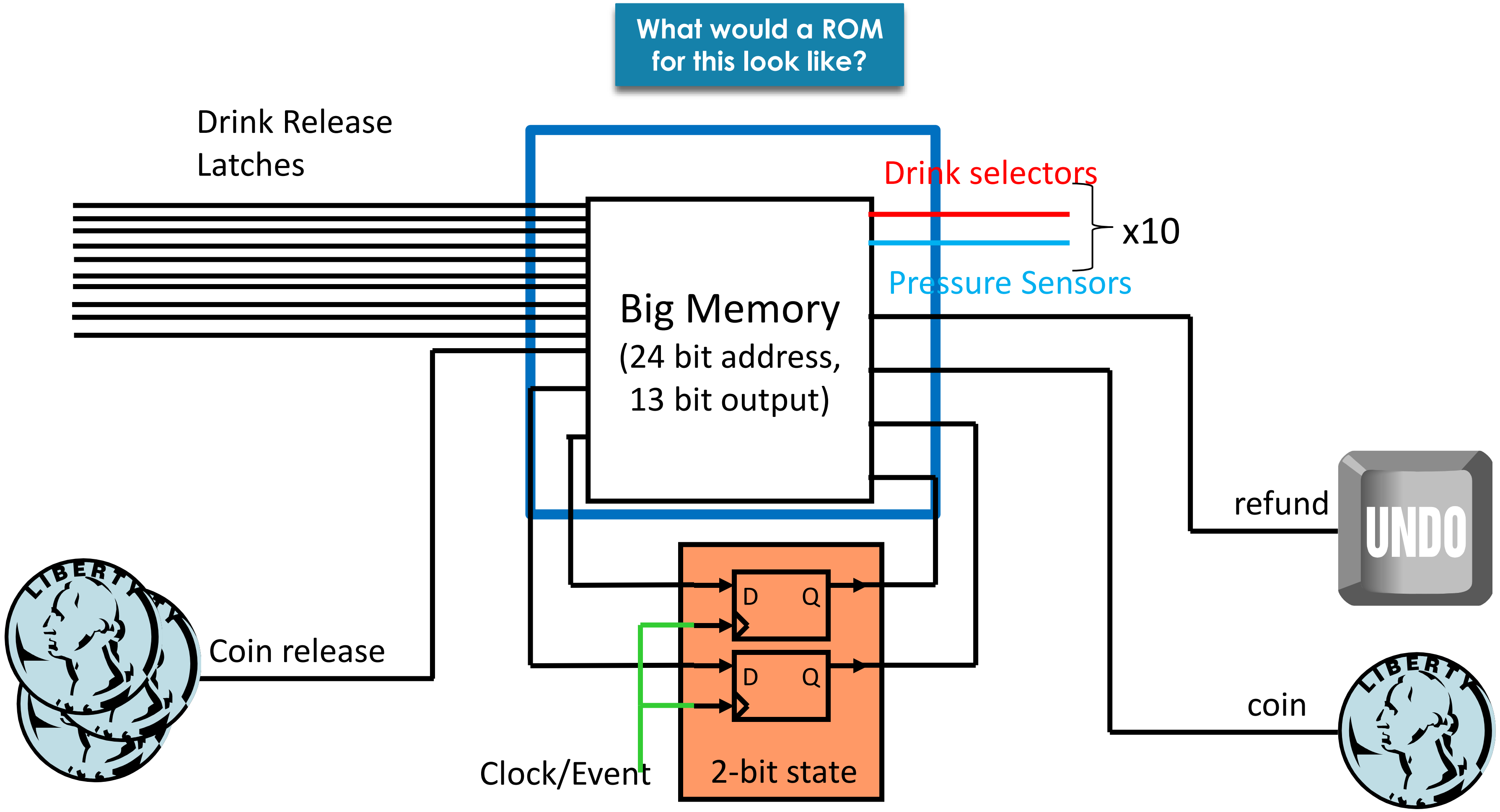
This ROM corresponds to this truth table



Implementing Combinational Logic

- Custom logic
  - Pros:
    - Can optimize the number of gates used
  - Cons:
    - Can be expensive / time consuming to make custom logic circuits
- Lookup table:
  - Pros:
    - Programmable ROMs (Read-Only Memories) are very cheap and can be programmed very quickly
  - Cons:
    - Size requirement grows exponentially with number of inputs (adding one just more bit **doubles** the storage requirements!)

Controller Design So far



ROM for Vending Machine

Size of ROM is (# of ROM entries * size of each entry)

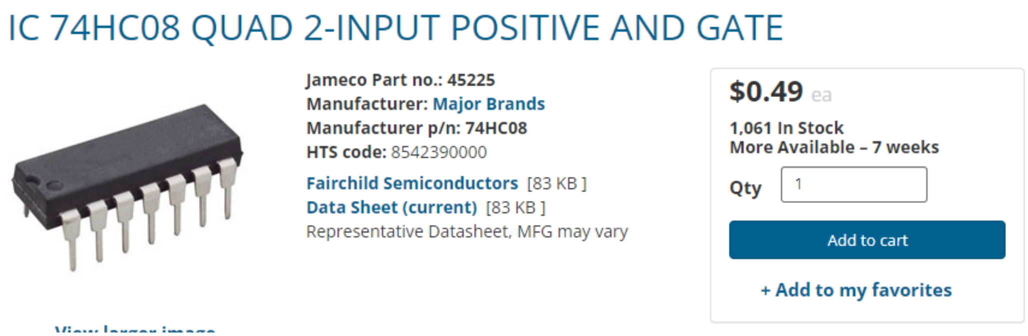
- # of ROM entries = $2^{\text{input\_size}} = 2^{24}$
- Size of each entry = output size = 13 bits

We need 2^{24} entry, 13 bit ROM memories

- **218,103,808 bits of ROM (26 MB)**
- Biggest ROM I could find on Jameco was 4 MB @ $6
  - Need 7 of these at $42??
- Let's see if we can do better

Reducing the ROM needed

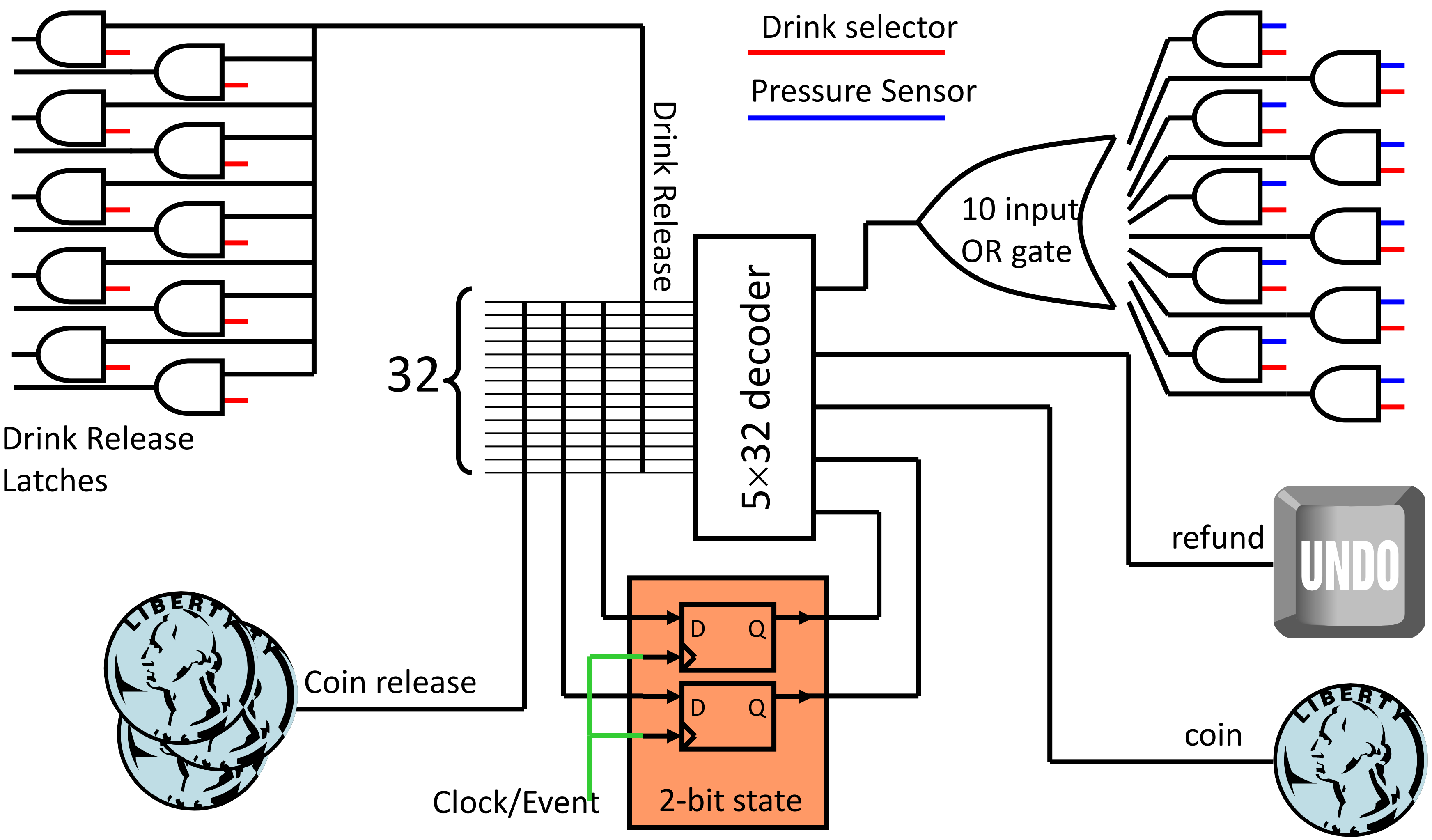
- Idea: let's do a hybrid between combinational logic and a lookup table
  - Use basic hardware (AND / OR) gates where we can, and a ROM for everything more complicated
  - AND / OR gates are mass producible & cheap!
    - ~$0.15 each on Jameco



Reducing the ROM needed

- Observation: overall logic doesn't really need to distinguish between **which** button was pressed
  - That's only relevant for choosing **which** latch is released, but overall logic is the same
- Replace 10 selector inputs and 10 pressure inputs with a **single** bit input (drink selected)
  - Use drink selection input to specify which drink release latch to activate
  - Only allow trigger if pressure sensor indicates that there is a bottle in that selection. (10 2-bit ANDs)

Putting it all together



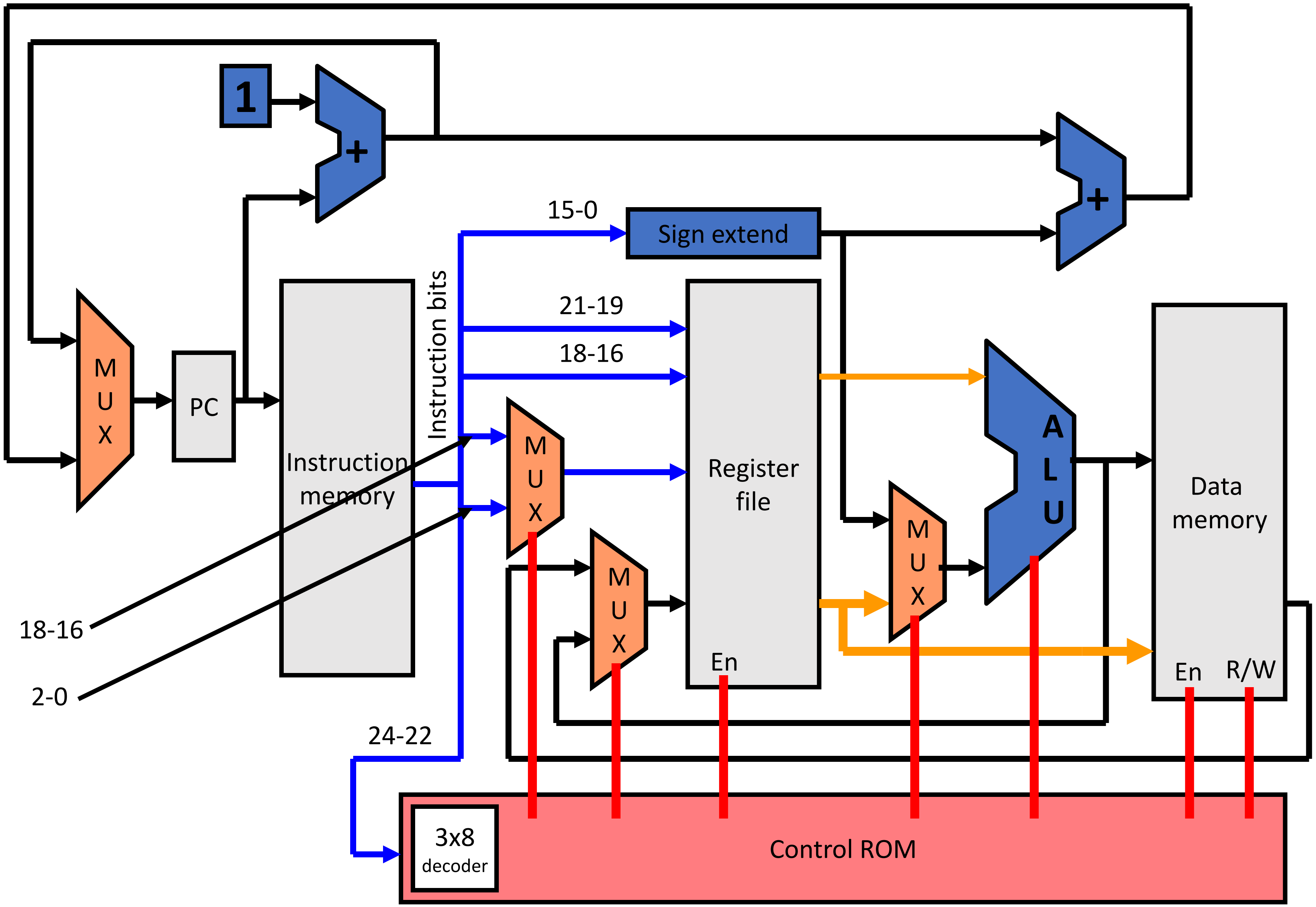
Total cost of our controller

- Now:
  - 2 current state bits + 3 input bits (5 bit ROM address)
  - 2 next state bits + 2 control trigger bits (4 bit memory)
  - $2^5 \times 4 = 128$ bit ROM
    - 1-millionth size of our 26 MB ROM 😬
- Total cost on Jameco:
  - Flip-flops to store state: $3
  - ROM to implement logic: $3
  - AND/OR gates: $5
  - **Total: $11**
- Could probably do a lot cheaper if we buy in bulk

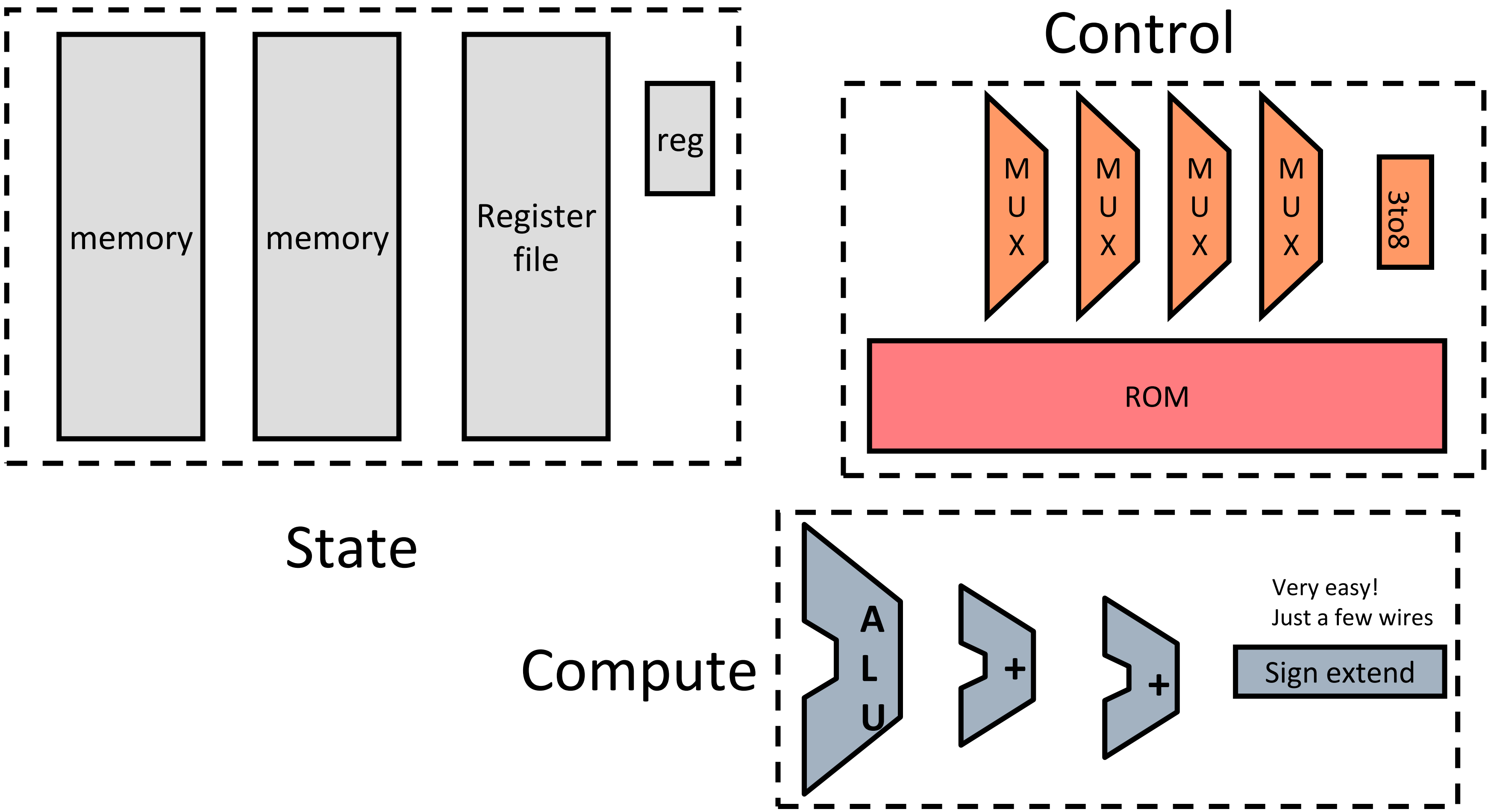
Single-Cycle Processor Design

- 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

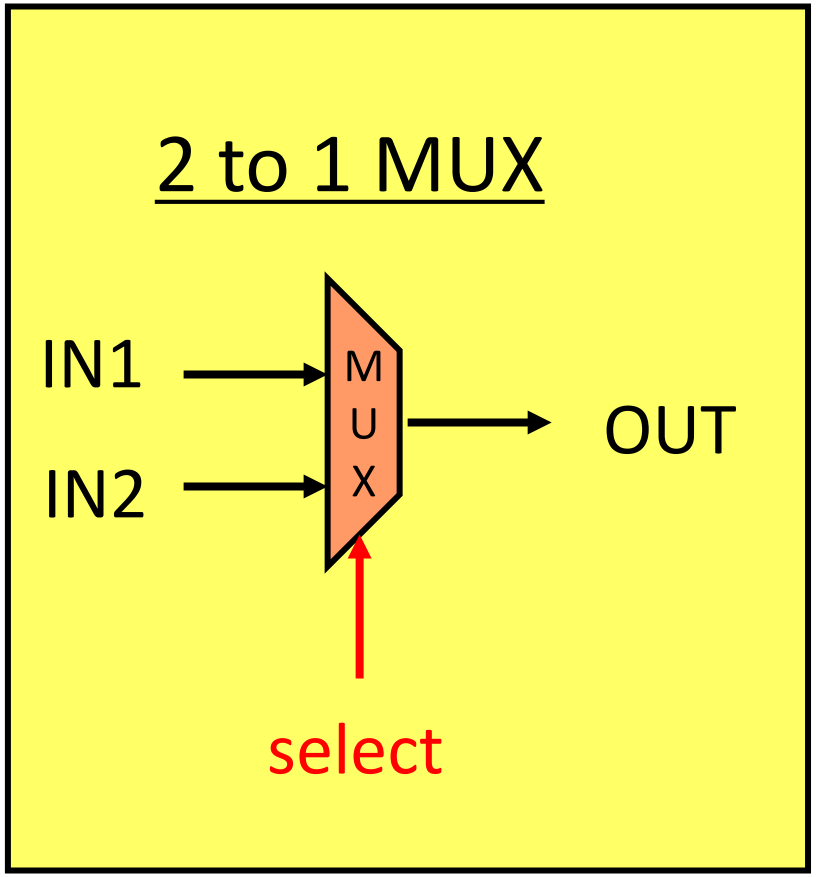


Building Blocks for the LC2K



Here are the pieces, go build yourself a processor!

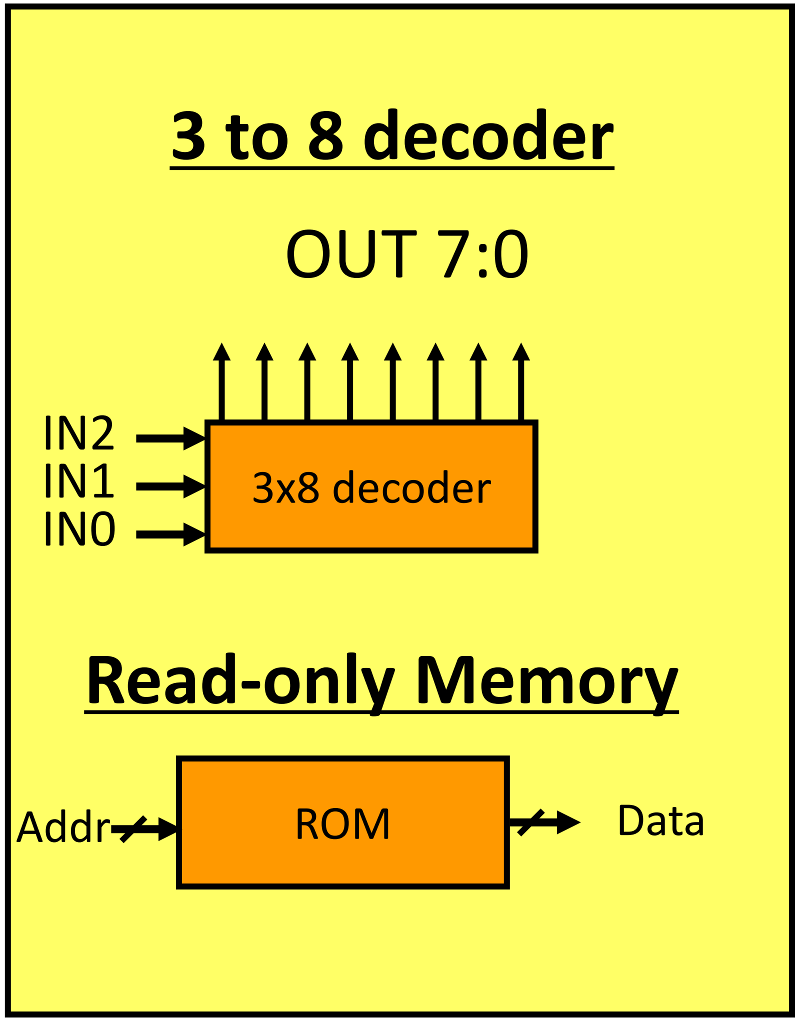
Control Building Blocks (1)



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

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

Control Building Blocks (2)



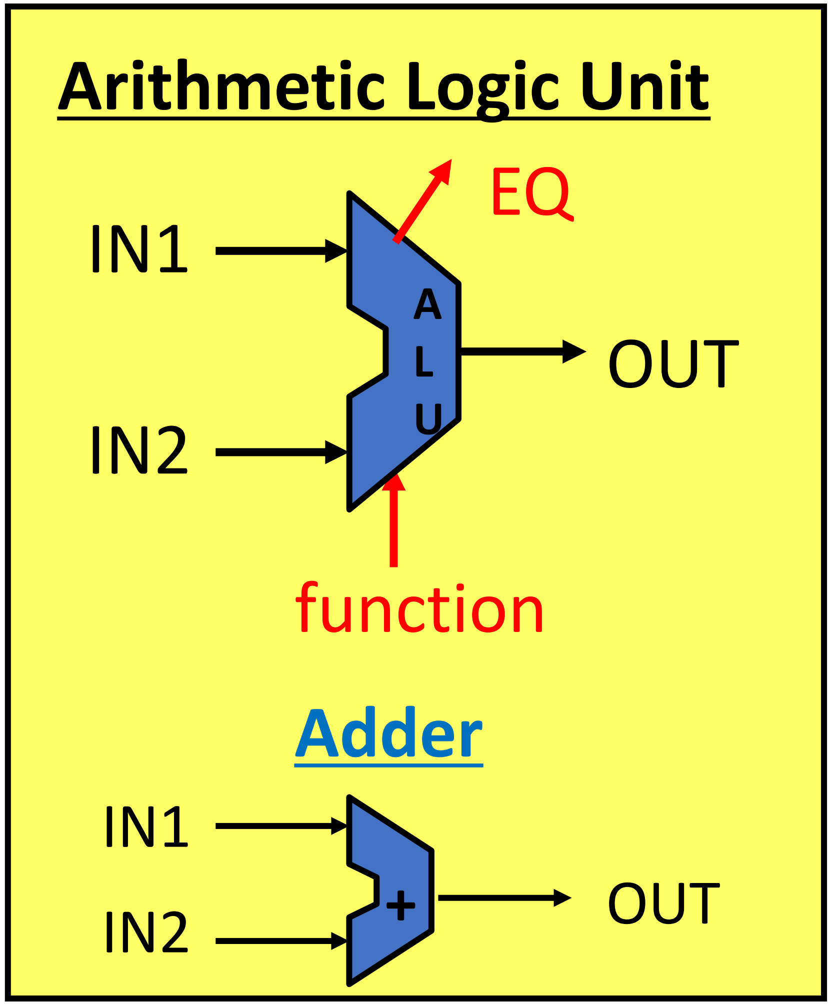
Decoder activates one of the output lines based on the input

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:

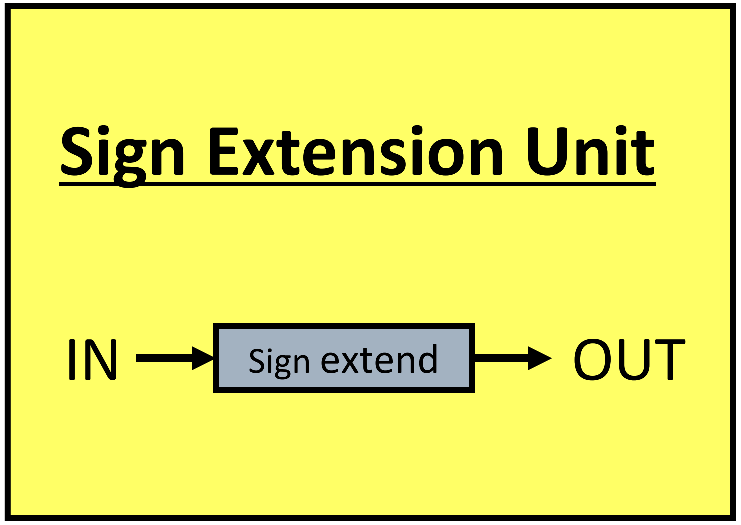
f=0 is add

f=1 is nor

For other processors, there are many more functions.

Just adds

Compute Building Blocks (2)



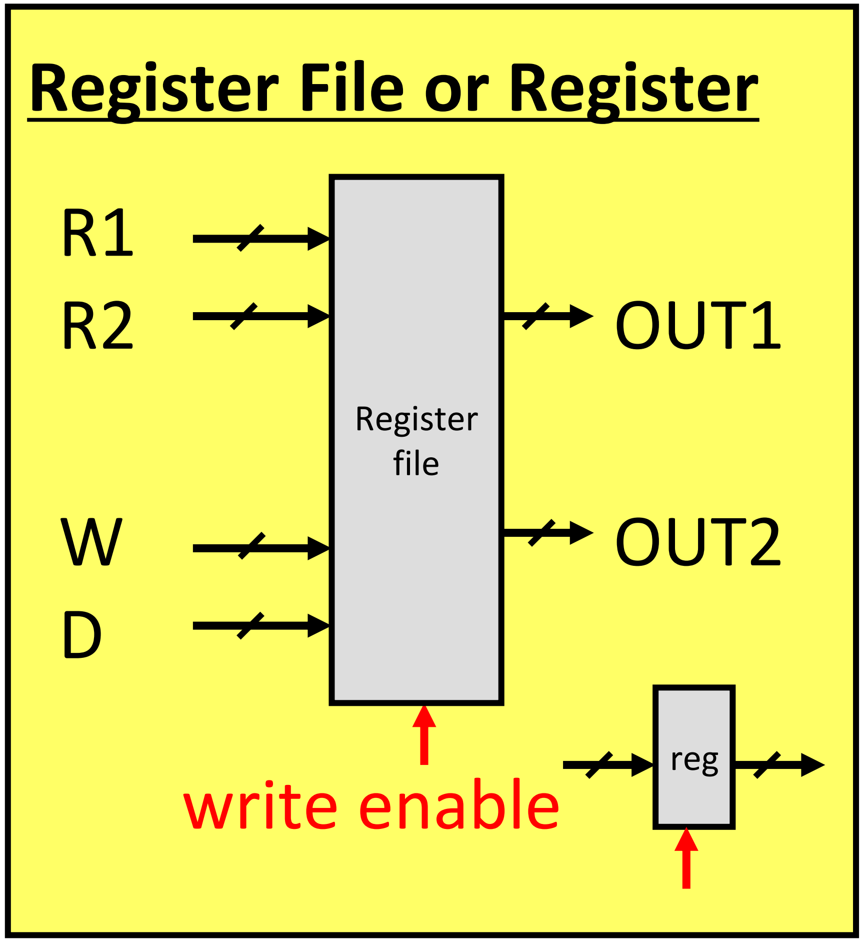
Sign extend (SE) input by replicating the MSB to width of output

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)

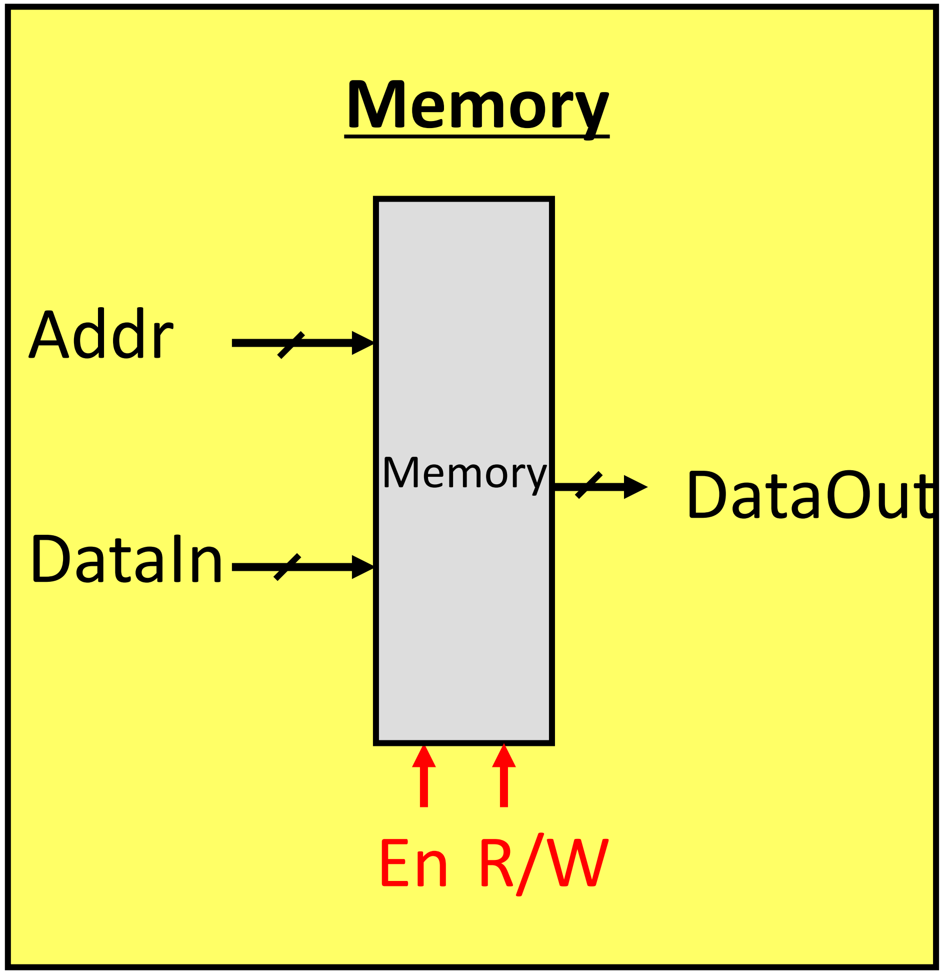


Small/fast memory to store temporary values

- **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

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

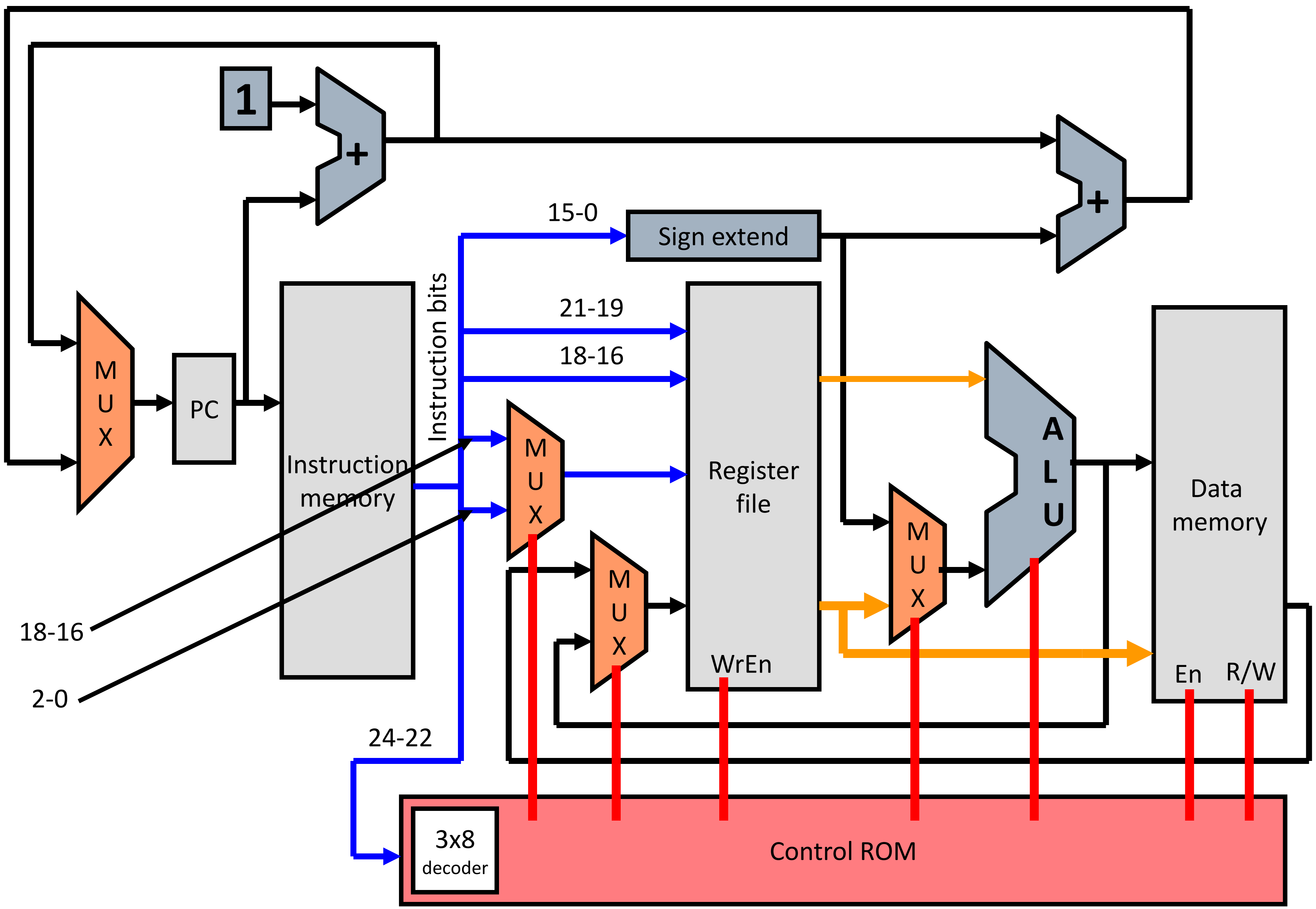
- R type instructions (add ‘000’, nor ‘001’)

31-25	24-22	21-19	18-16	15-3	2-0
unused	**opcode**	**regA**	**regB**	*unused*	**destR**

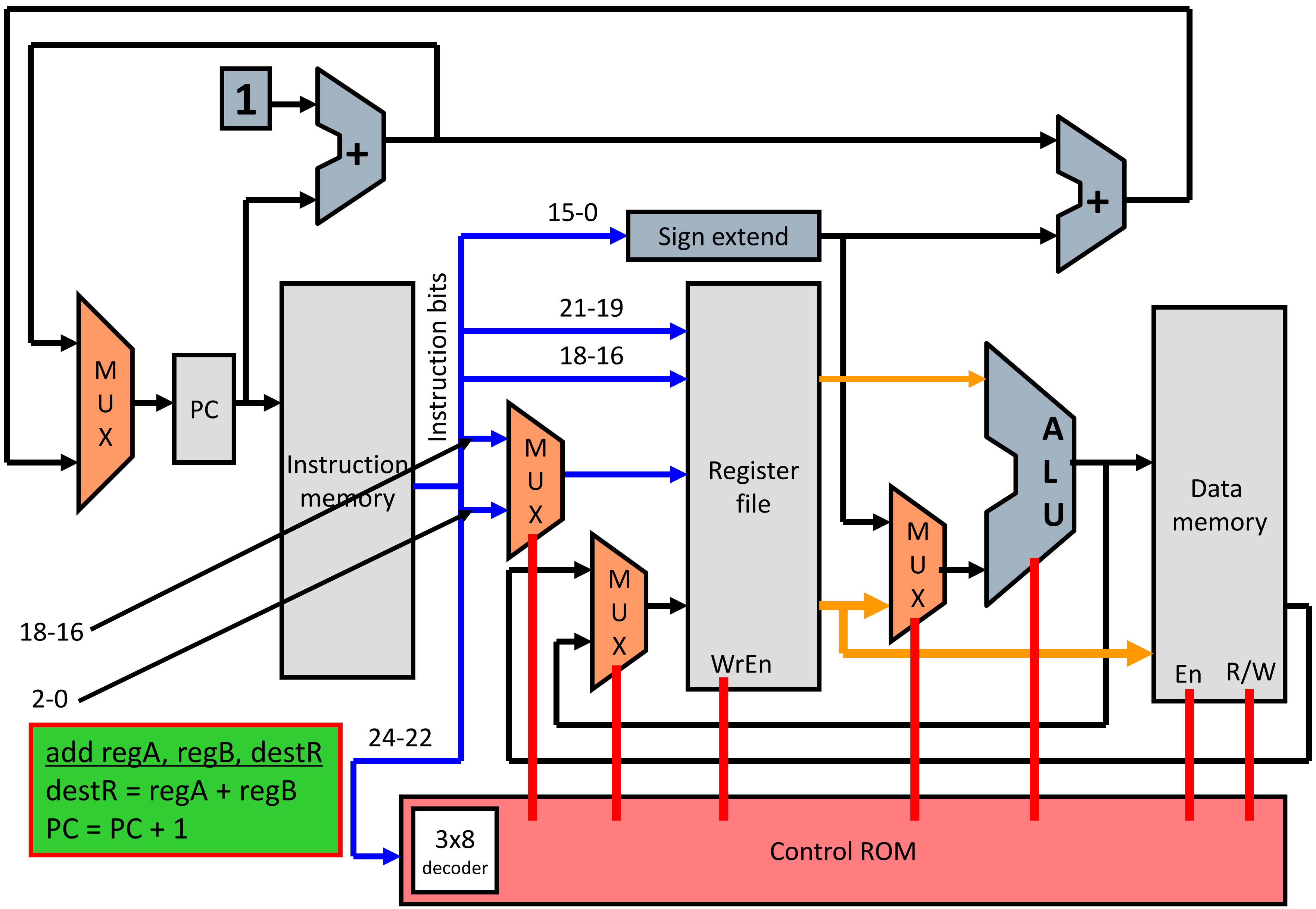
- I type instructions (lw ‘010’, sw ‘011’, beq ‘100’)

31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**

LC2K Single-Cycle Datapath Implementation

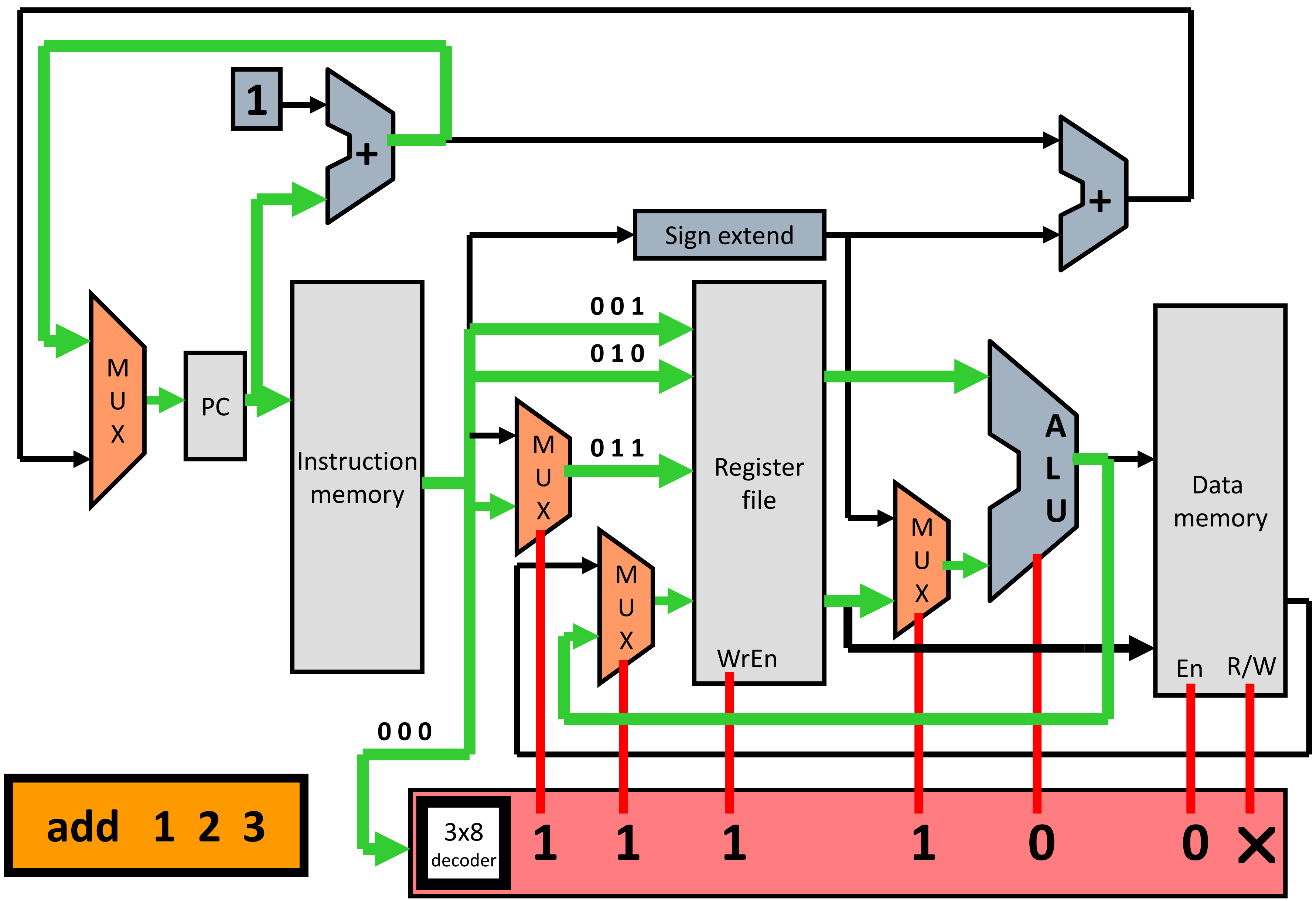


Executing an ADD Instruction on LC2K Datapath



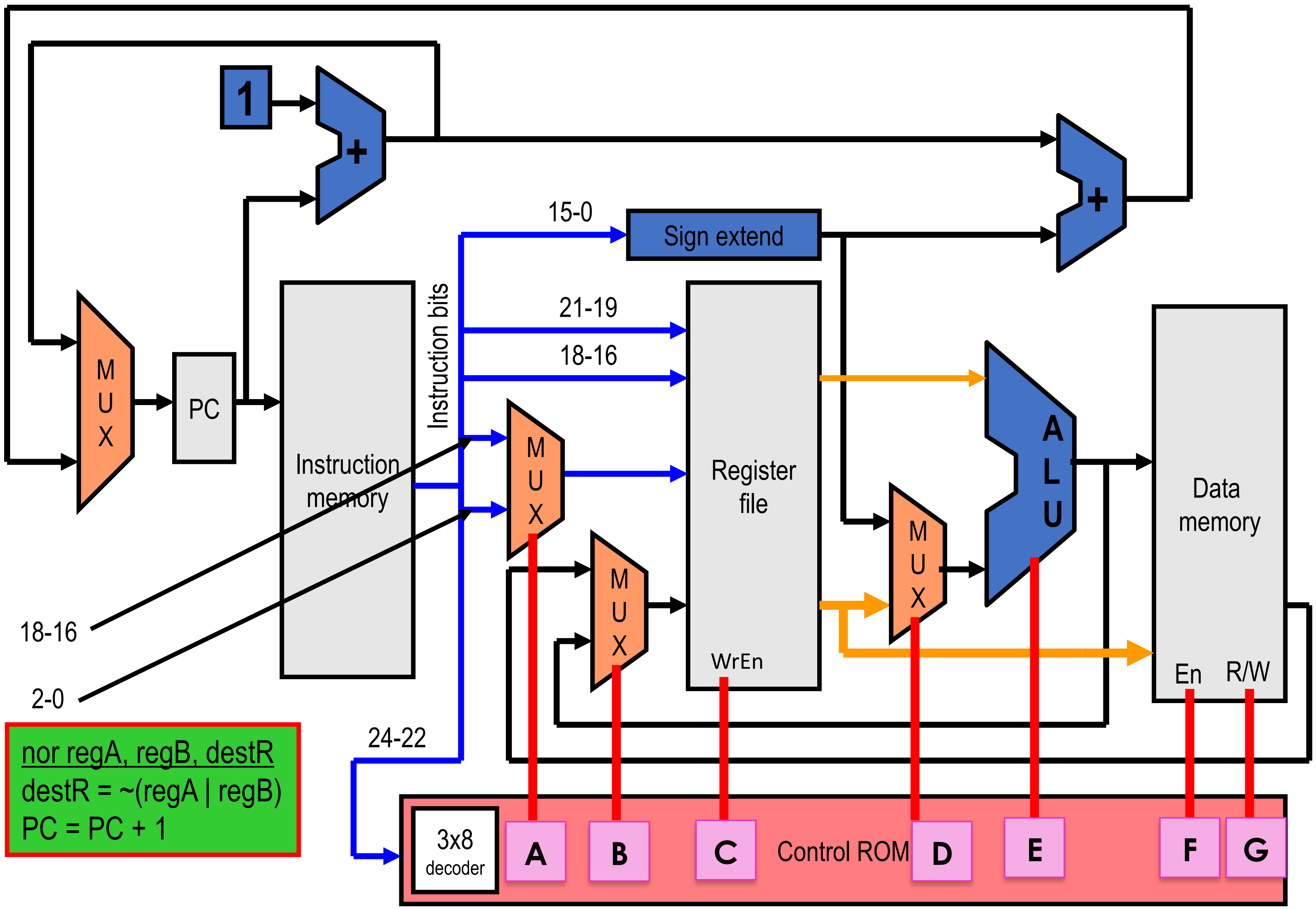
add regA, regB, destR
destR = regA + regB
PC = PC + 1

Executing an ADD Instruction on LC2K Datapath



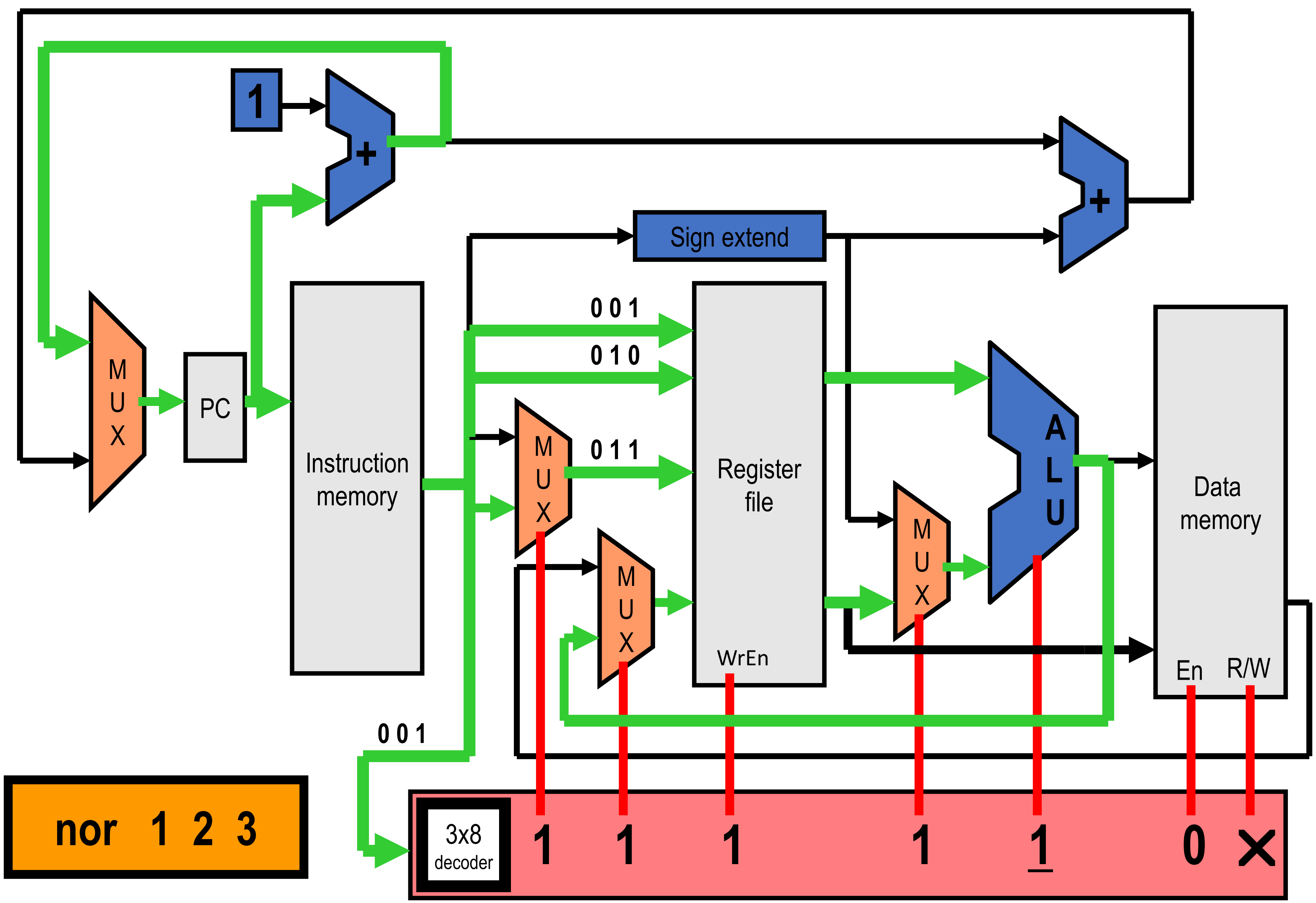
Poll: Which control bits need to be different from ADD?

Executing a NOR Instruction



nor regA, regB, destR
destR = ~(regA | regB)
PC = PC + 1

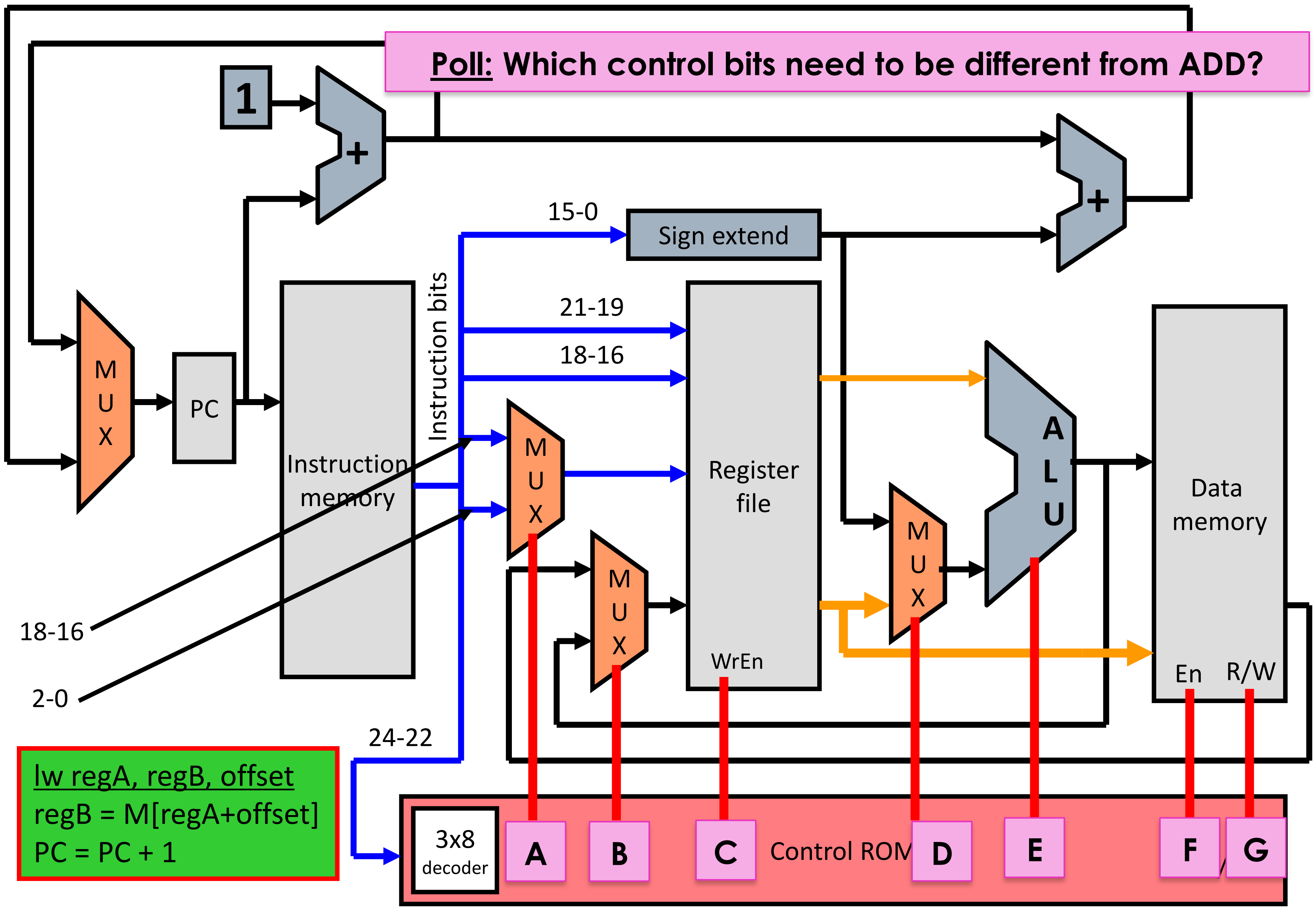
Executing a NOR Instruction



Executing a LW Instruction

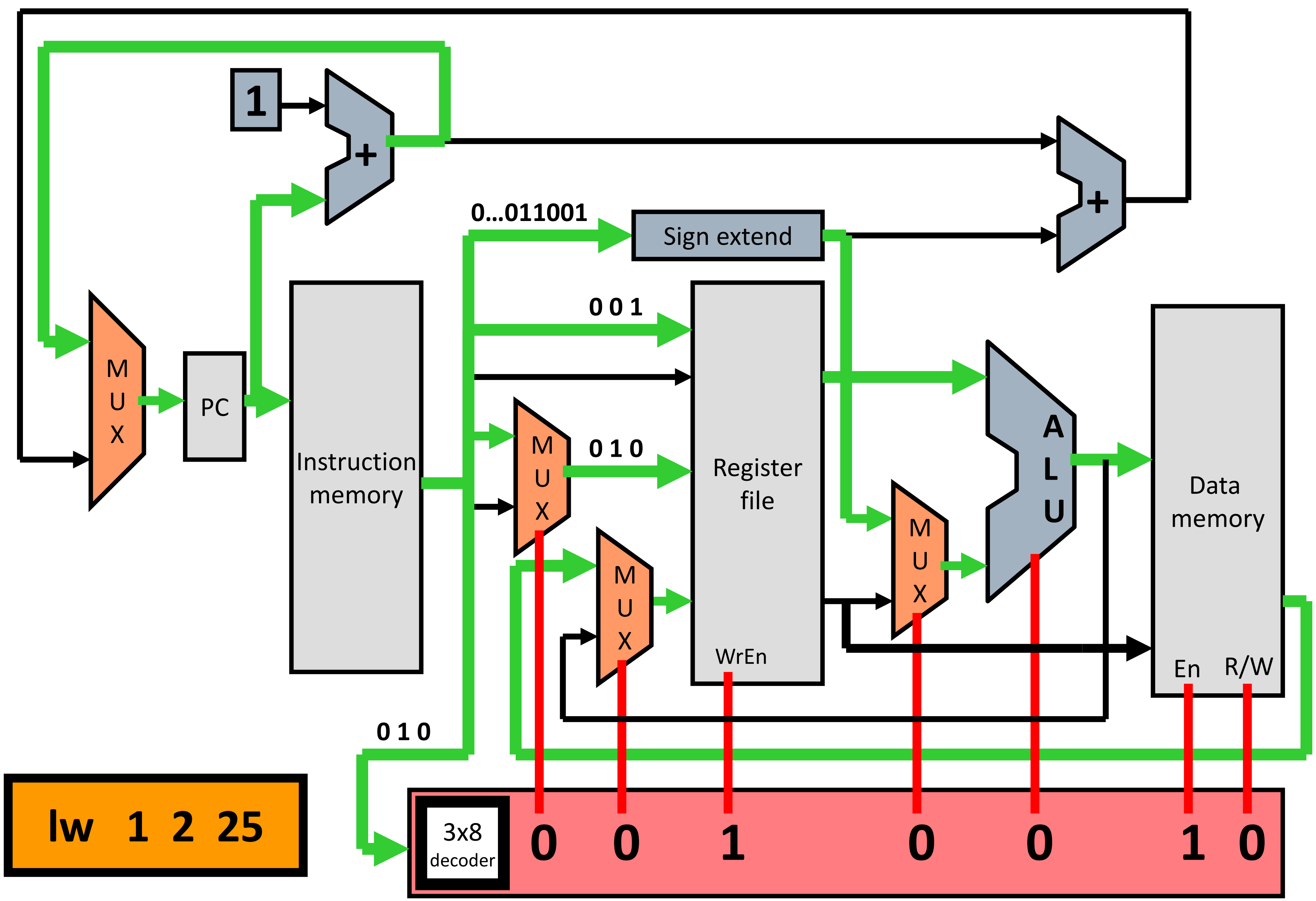
31-25	24-22	21-19	18-16	15-0
unused	**opcode**	**regA**	**regB**	**offset**

<u>Poll:</u> Which control bits need to be different from ADD?

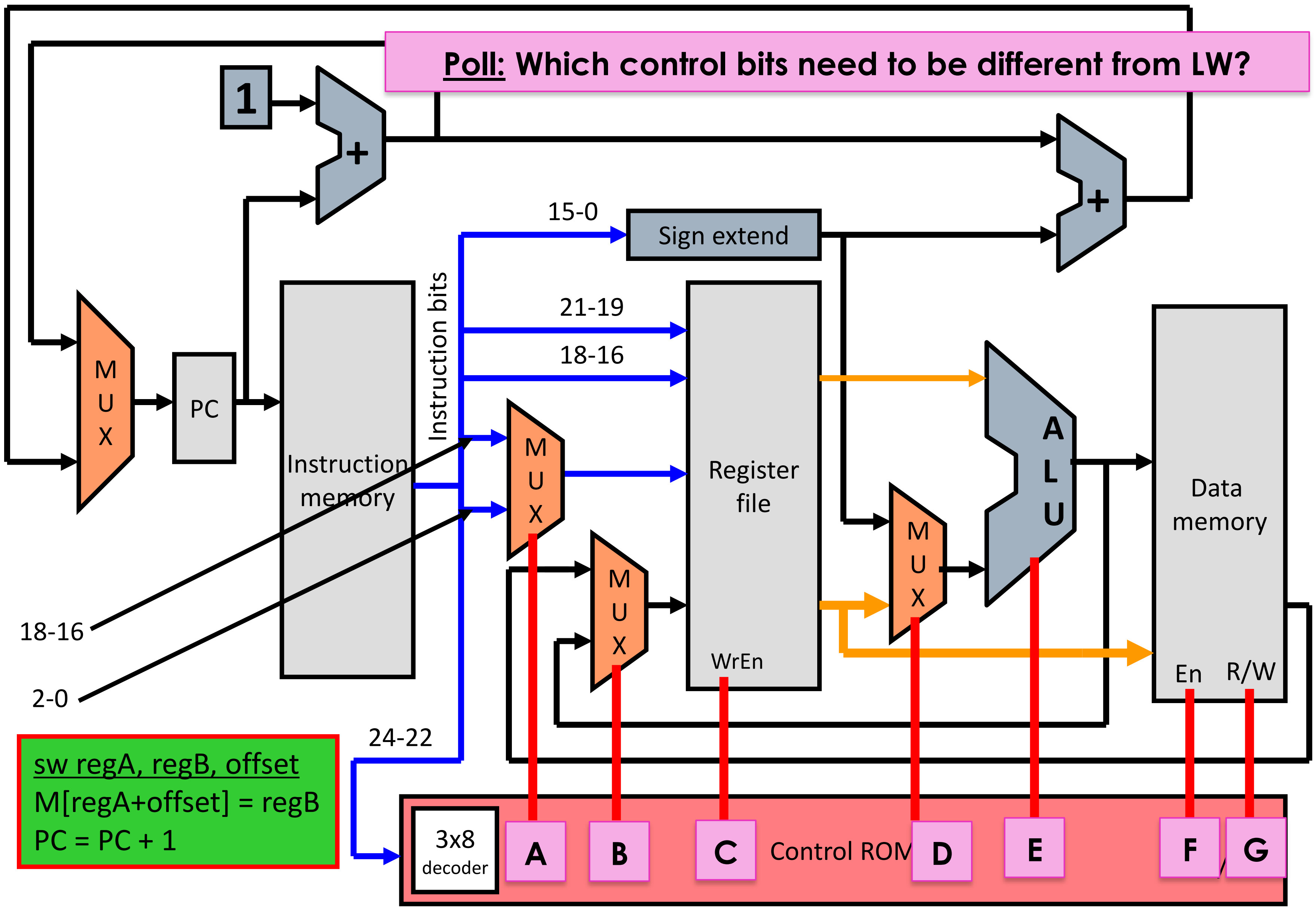


lw regA, regB, offset
regB = M[regA+offset]
PC = PC + 1

Executing a LW Instruction

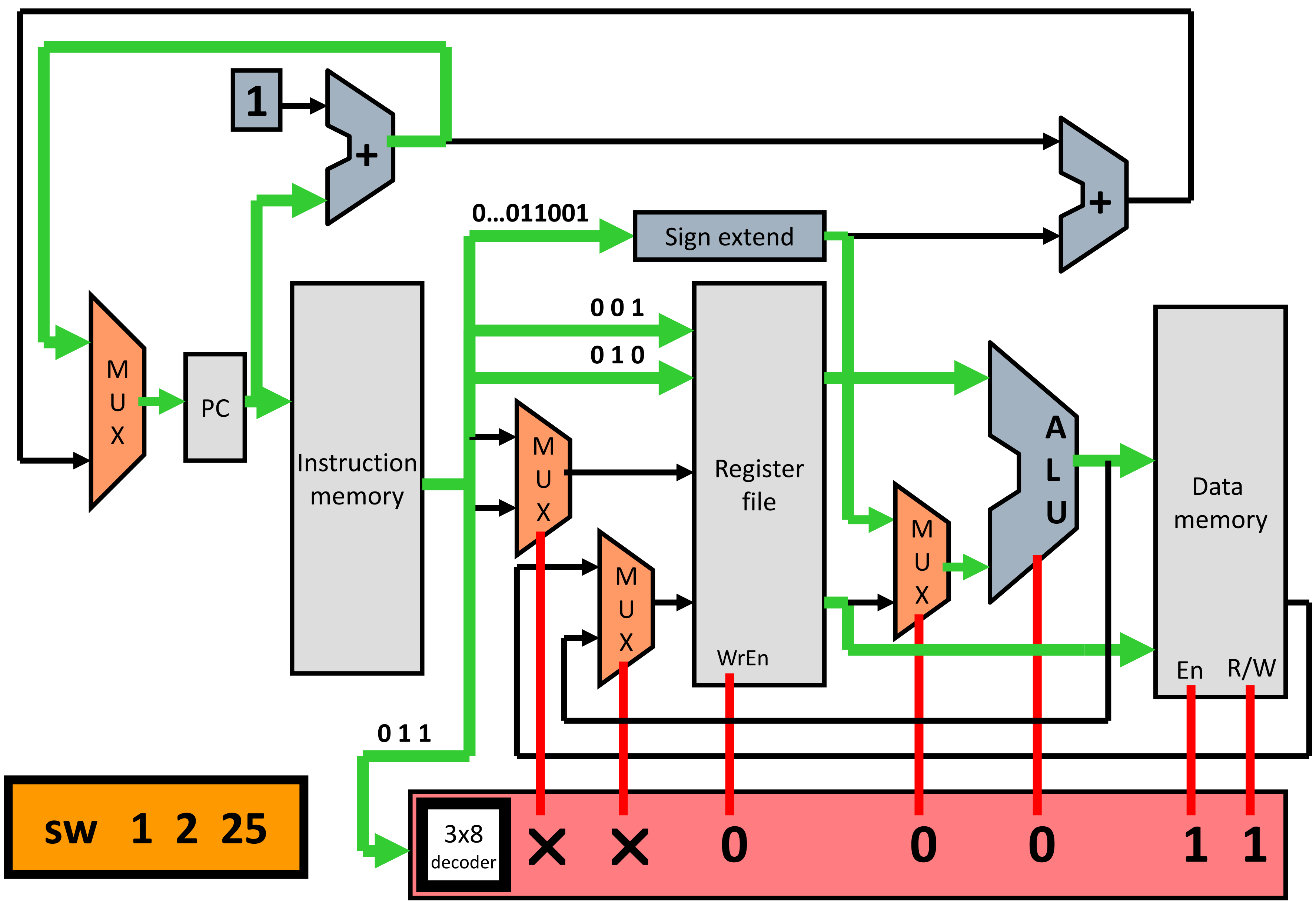


Executing a SW Instruction

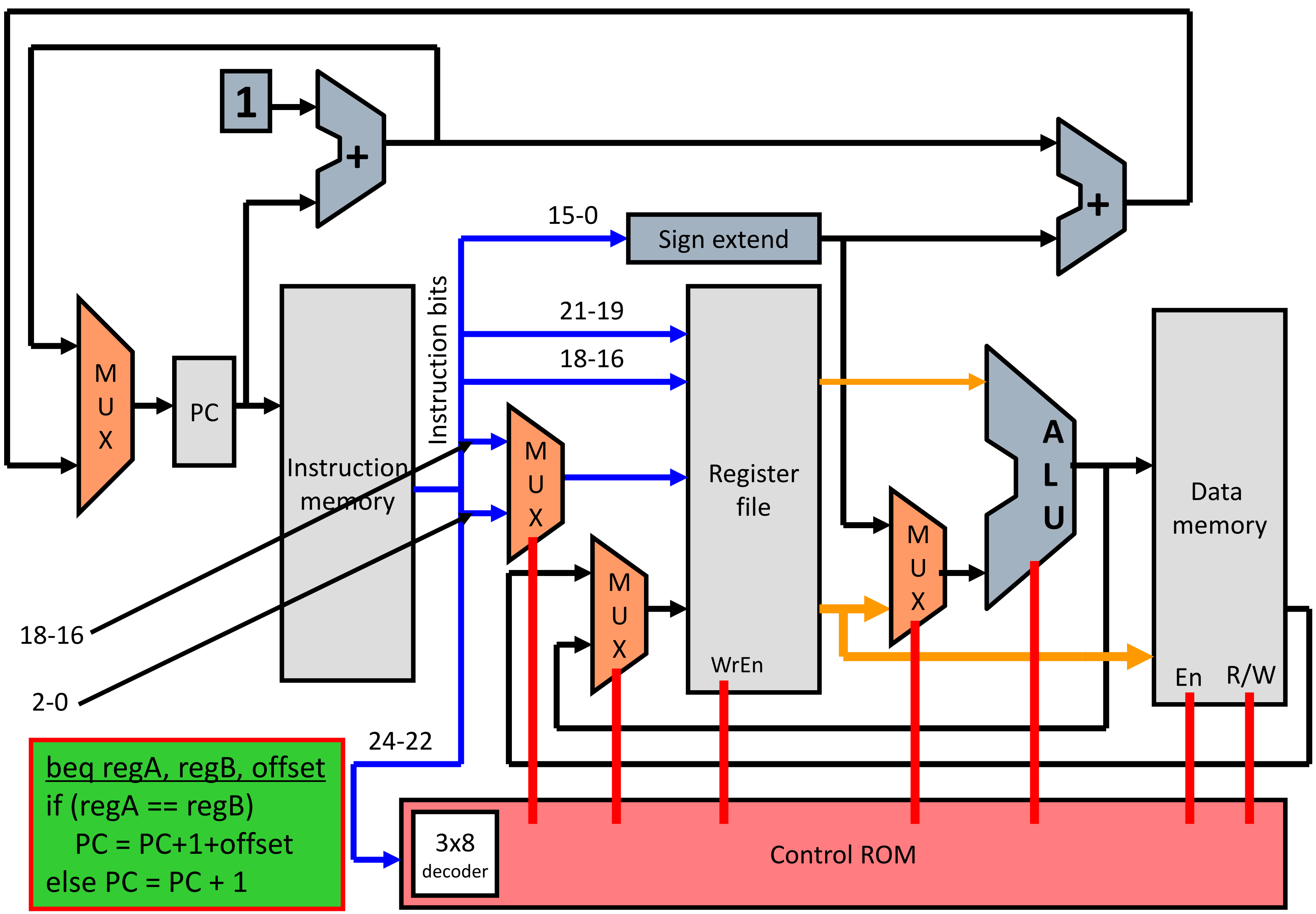


sw regA, regB, offset
M[regA+offset] = regB
PC = PC + 1

Executing a SW Instruction

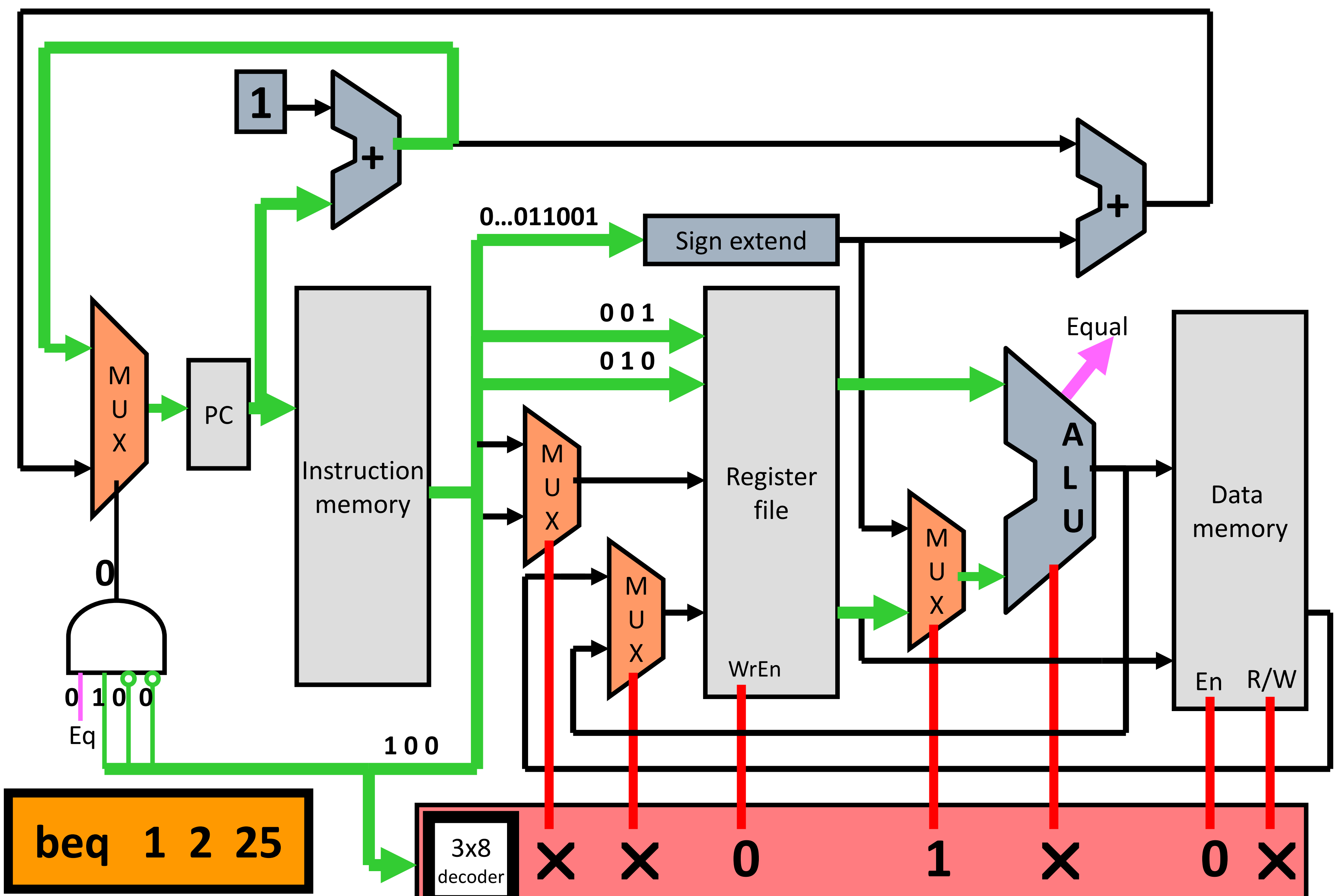


Executing a BEQ Instruction

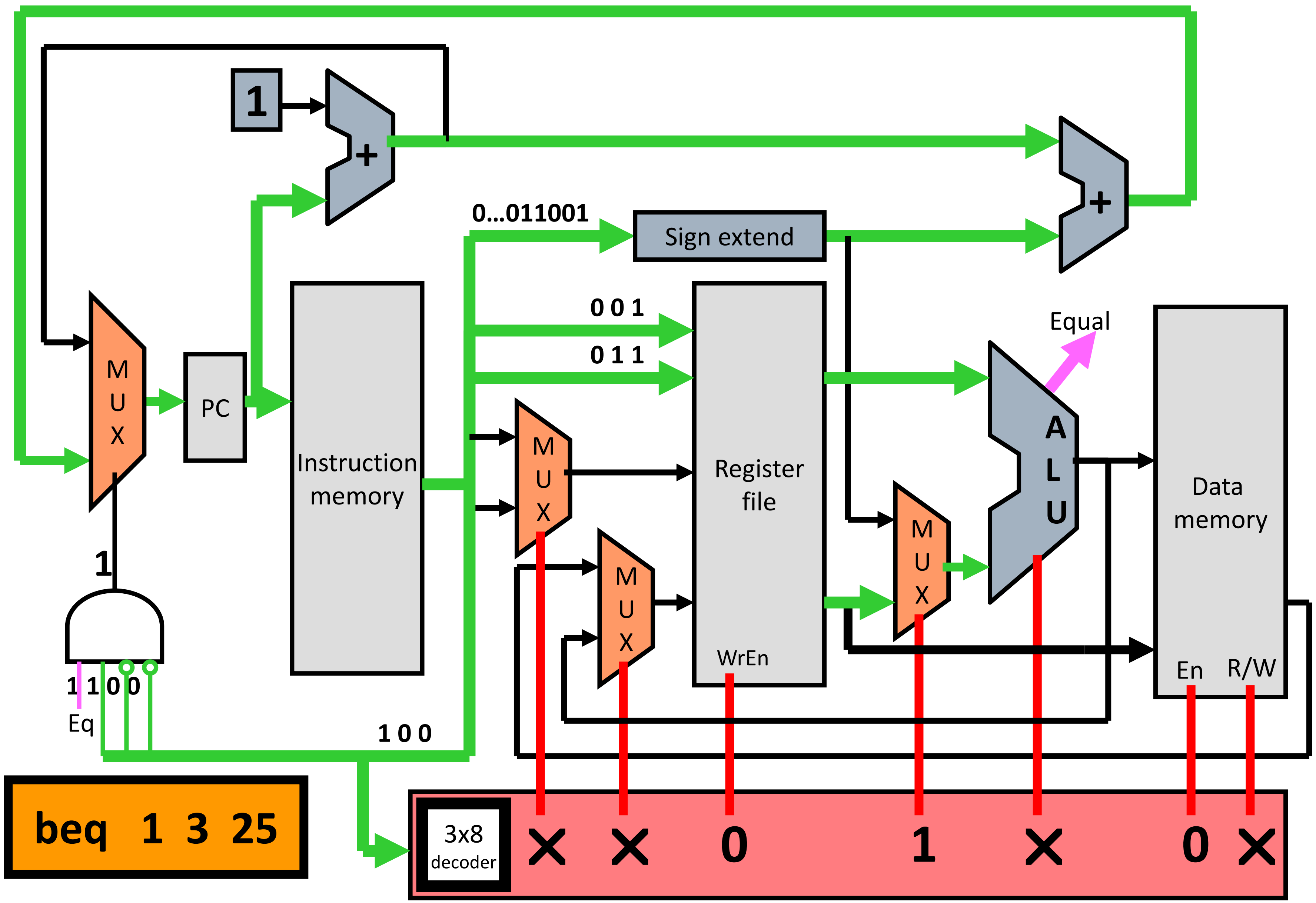


beq regA, regB, offset
if (regA == regB)
PC = PC+1+offset
else PC = PC + 1

Executing “not taken” BEQ Instruction on LC2K Datapath



Executing a “taken” BEQ Instruction on LC2K Datapath



So Far, So Good

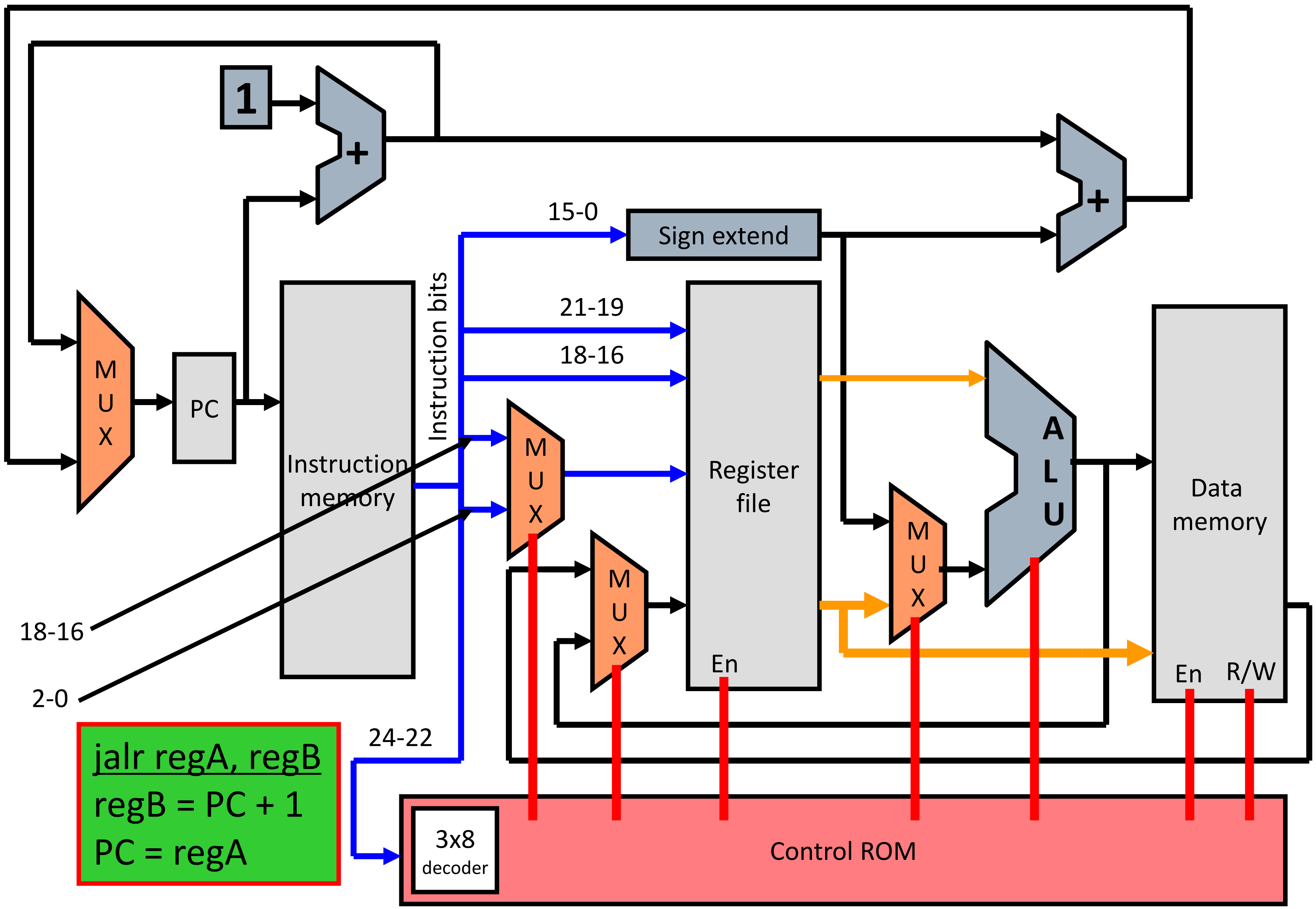
- Every architecture seems to have at least one "ugly" instruction
  - Something that doesn't elegantly fit in with the hardware we've already included

- 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

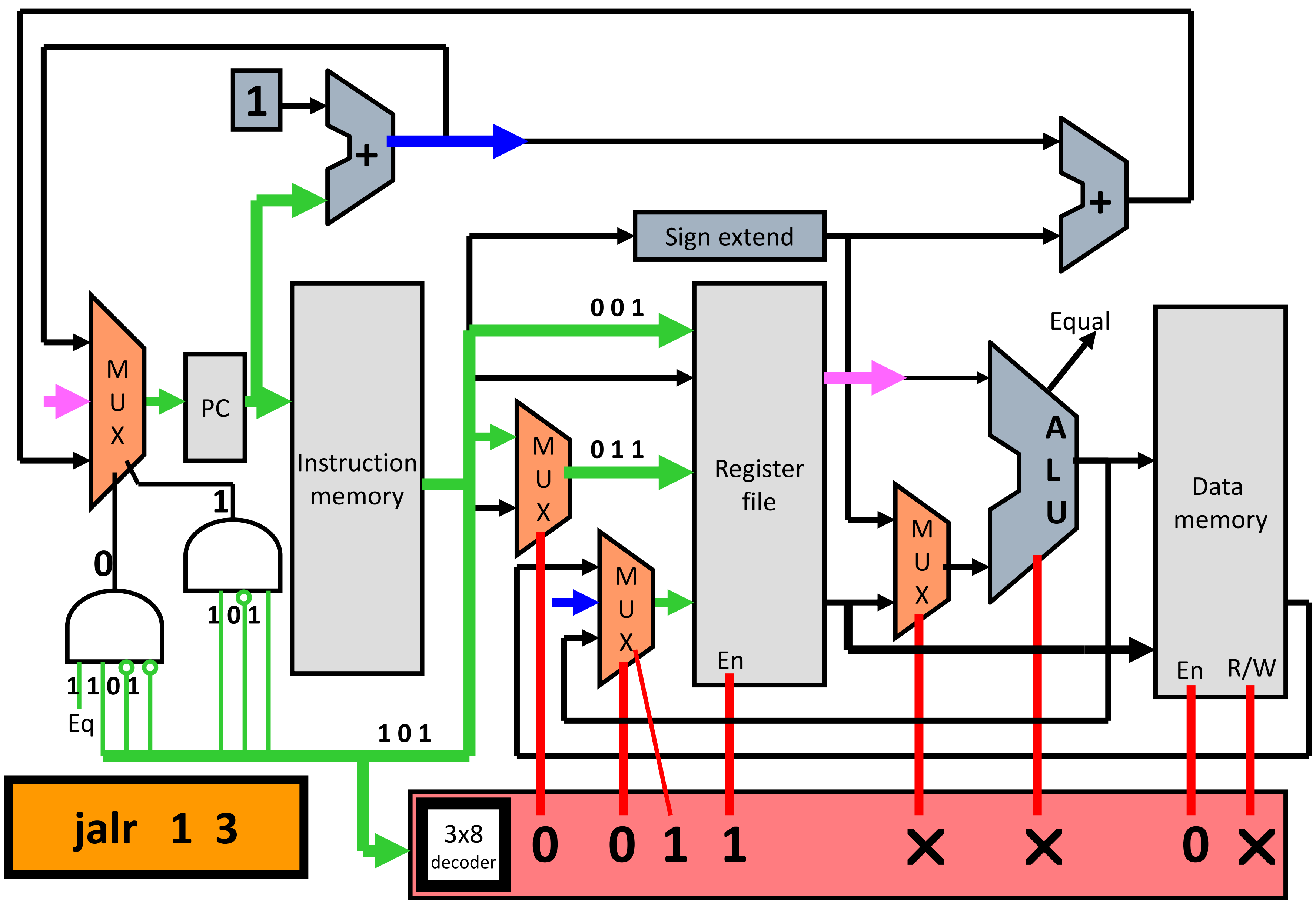
- Right now there is:
  - No path to write PC+1 into a register
  - No path to write a register to the PC

Executing a JALR Instruction

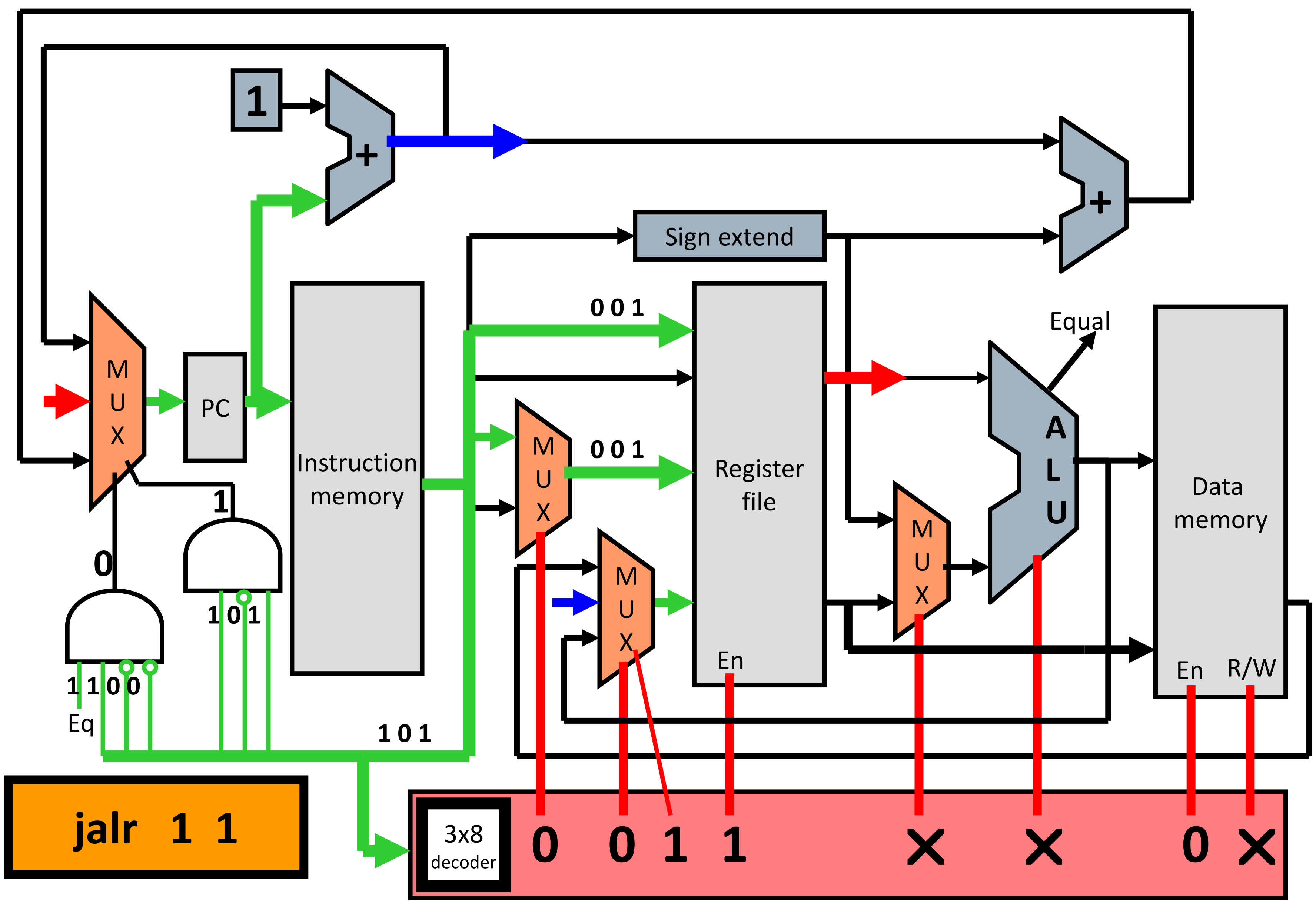


jalr regA, regB
regB = PC + 1
PC = regA

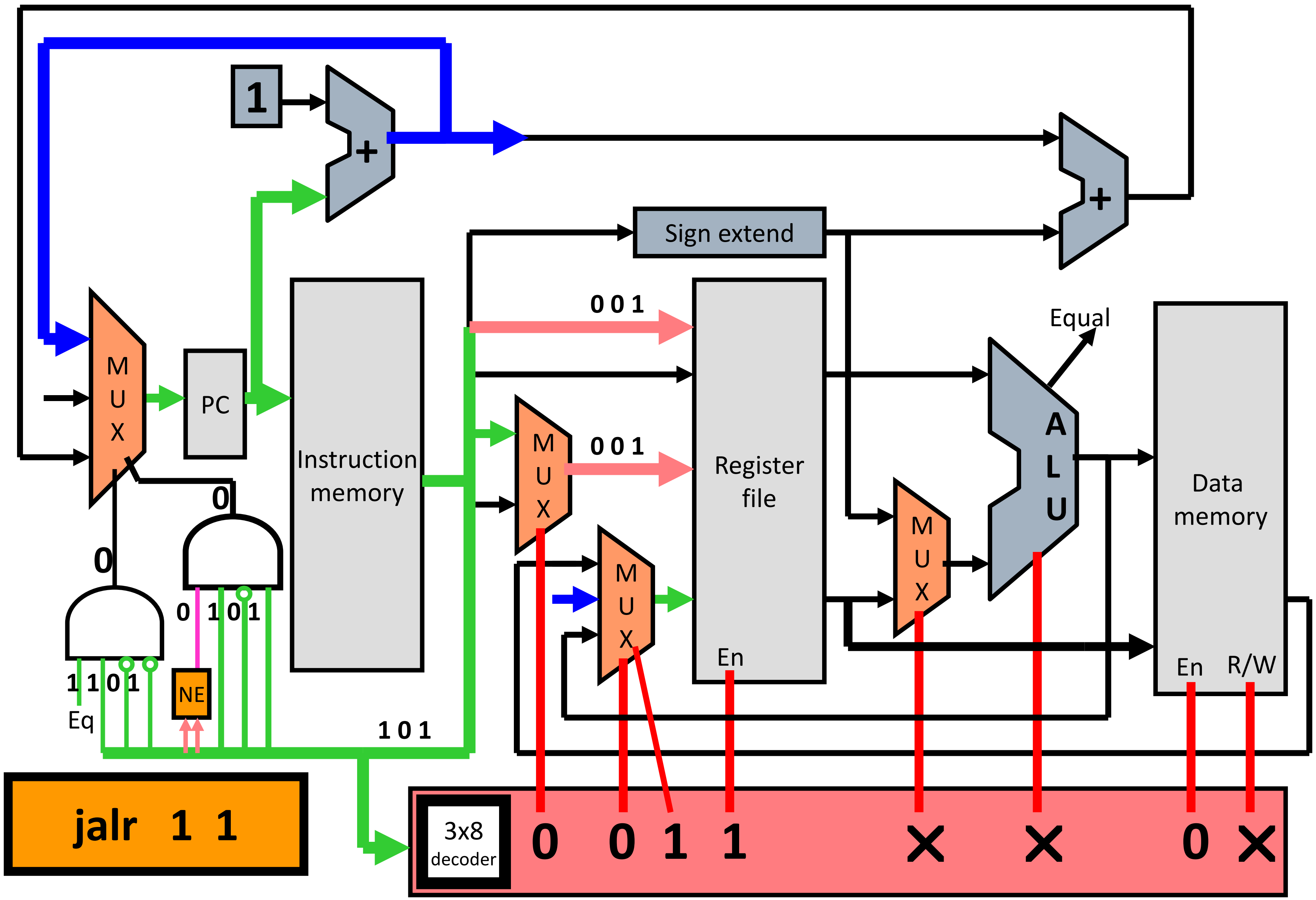
Executing a JALR Instruction



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

Poll: 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:

??

<u>Poll:</u> 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

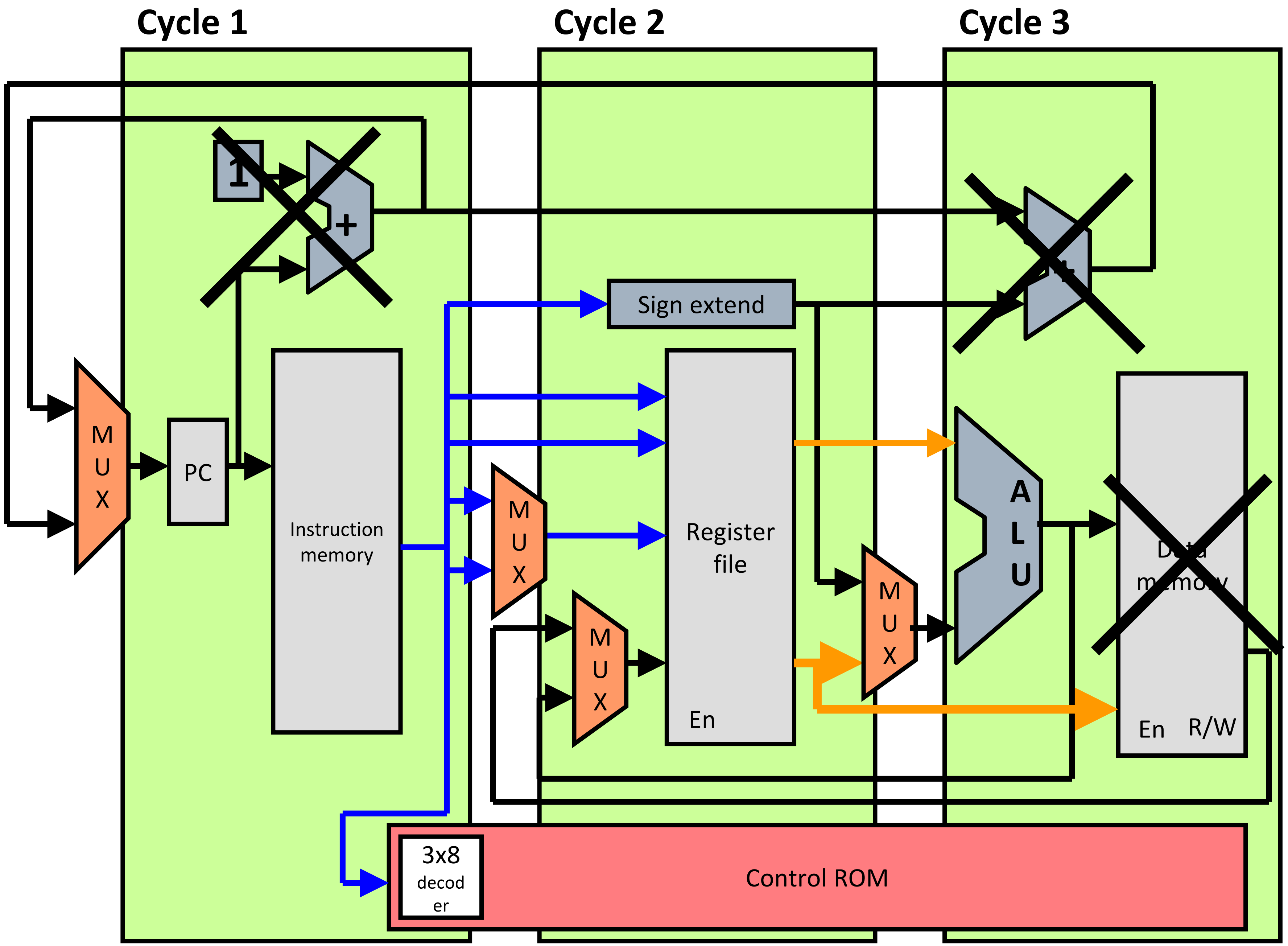
Optimal execution:

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

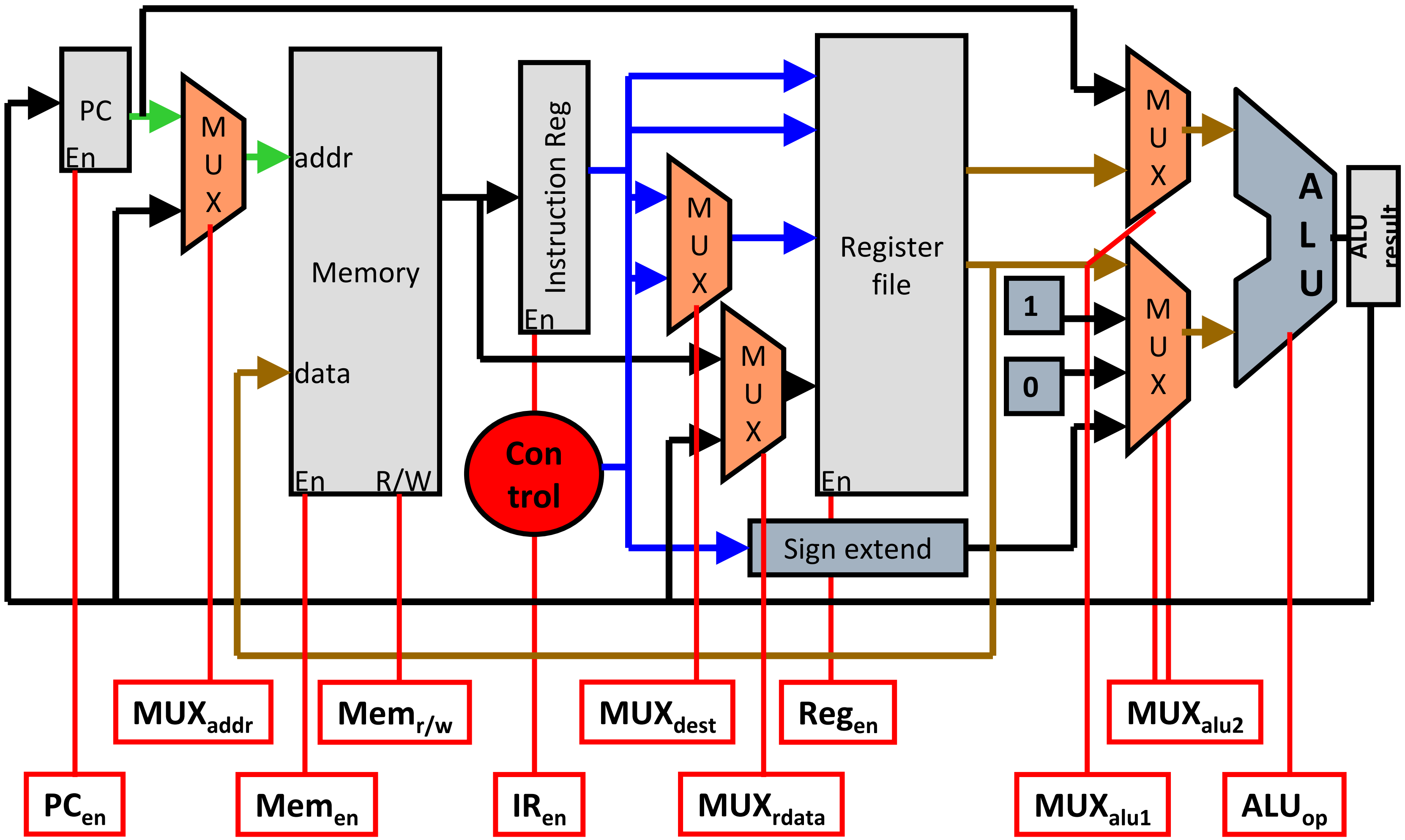
Multiple-Cycle Execution

- Each instruction takes multiple cycles to execute
  - Cycle time is reduced
  - Slower instructions take more cycles
  - Faster instruction take fewer cycles
    - <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.

LC2K Datapath – cycle groups

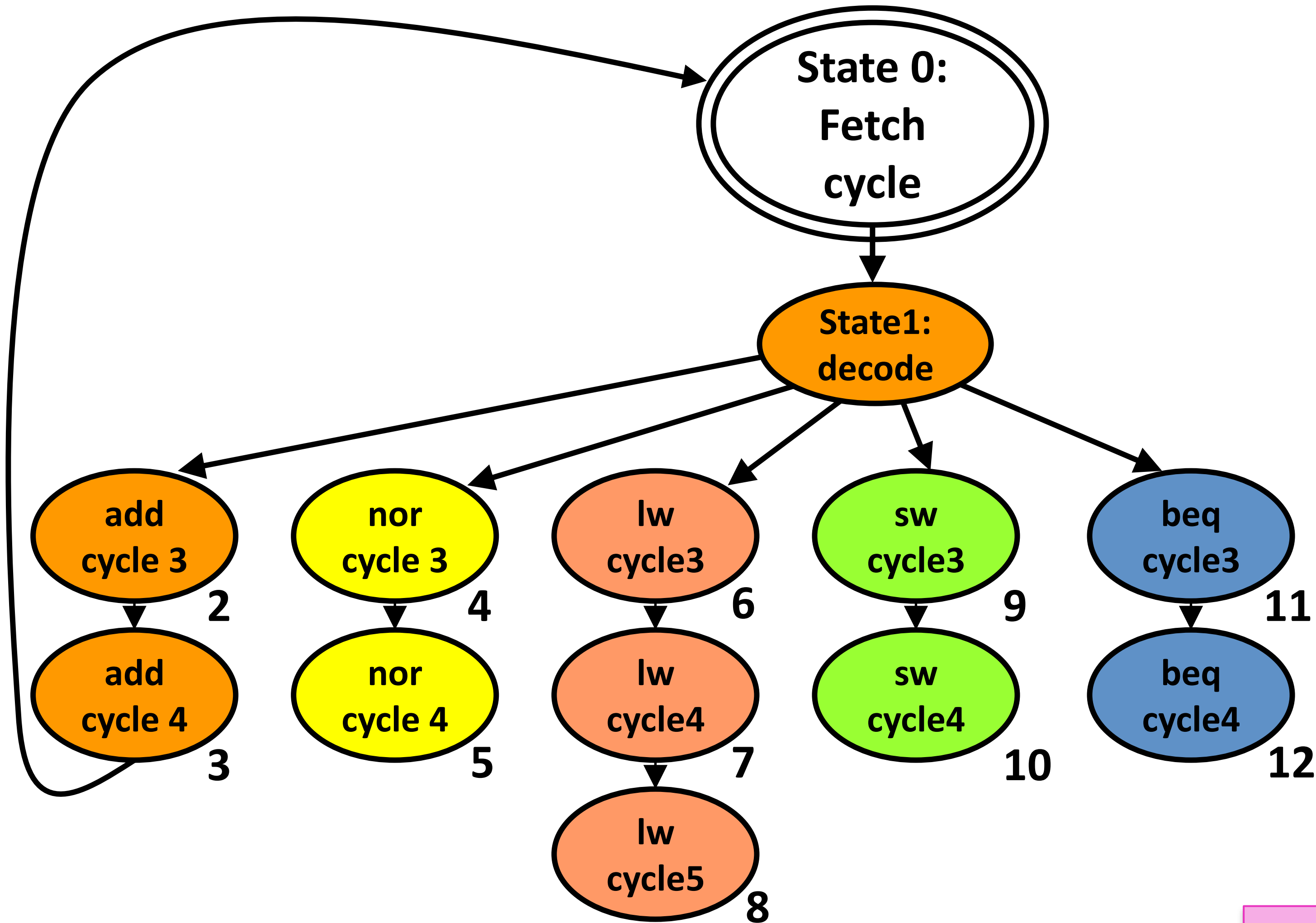


Multi-cycle LC2 Datapath



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

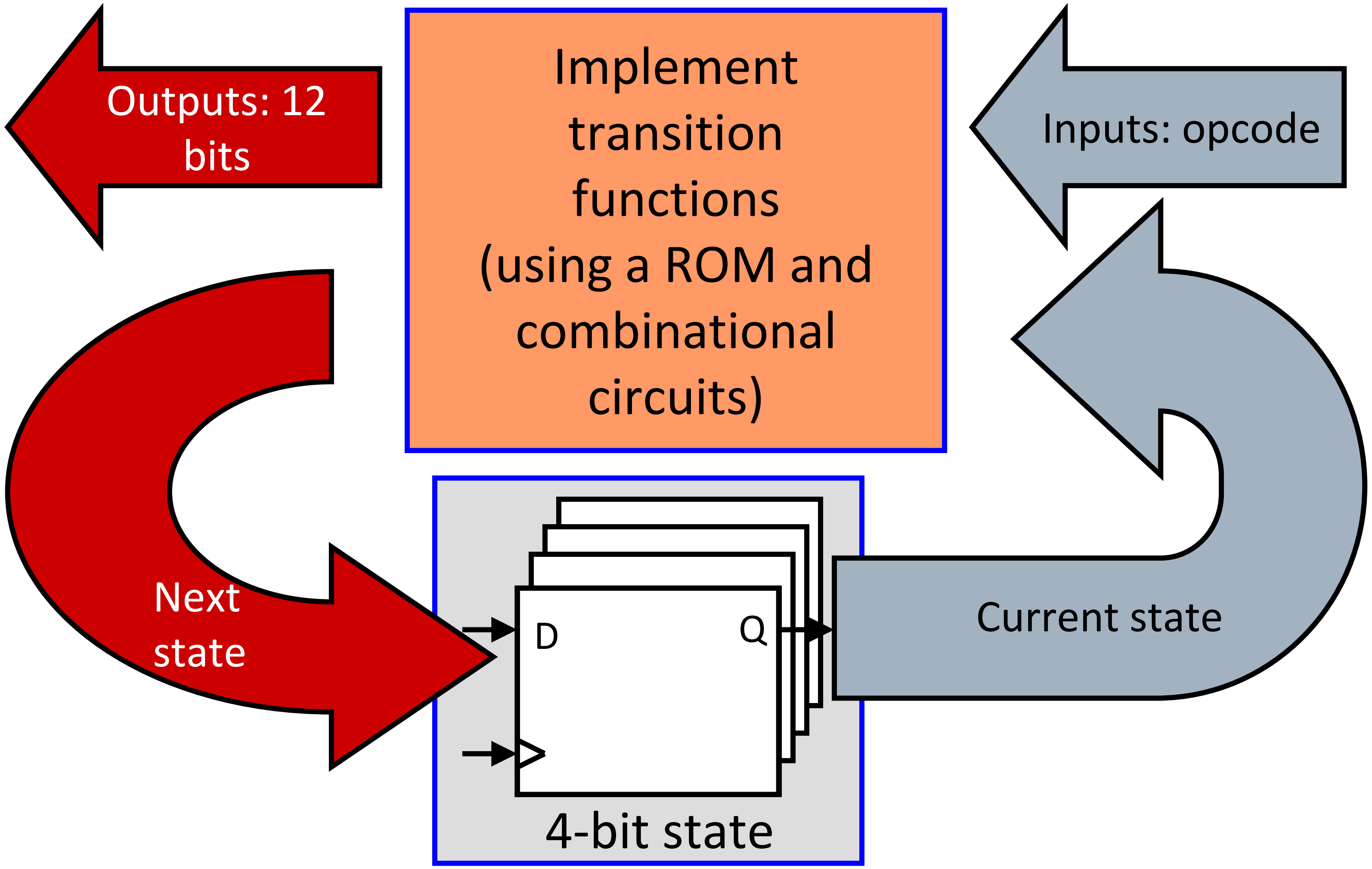
State machine for multi-cycle control signals (transition functions)



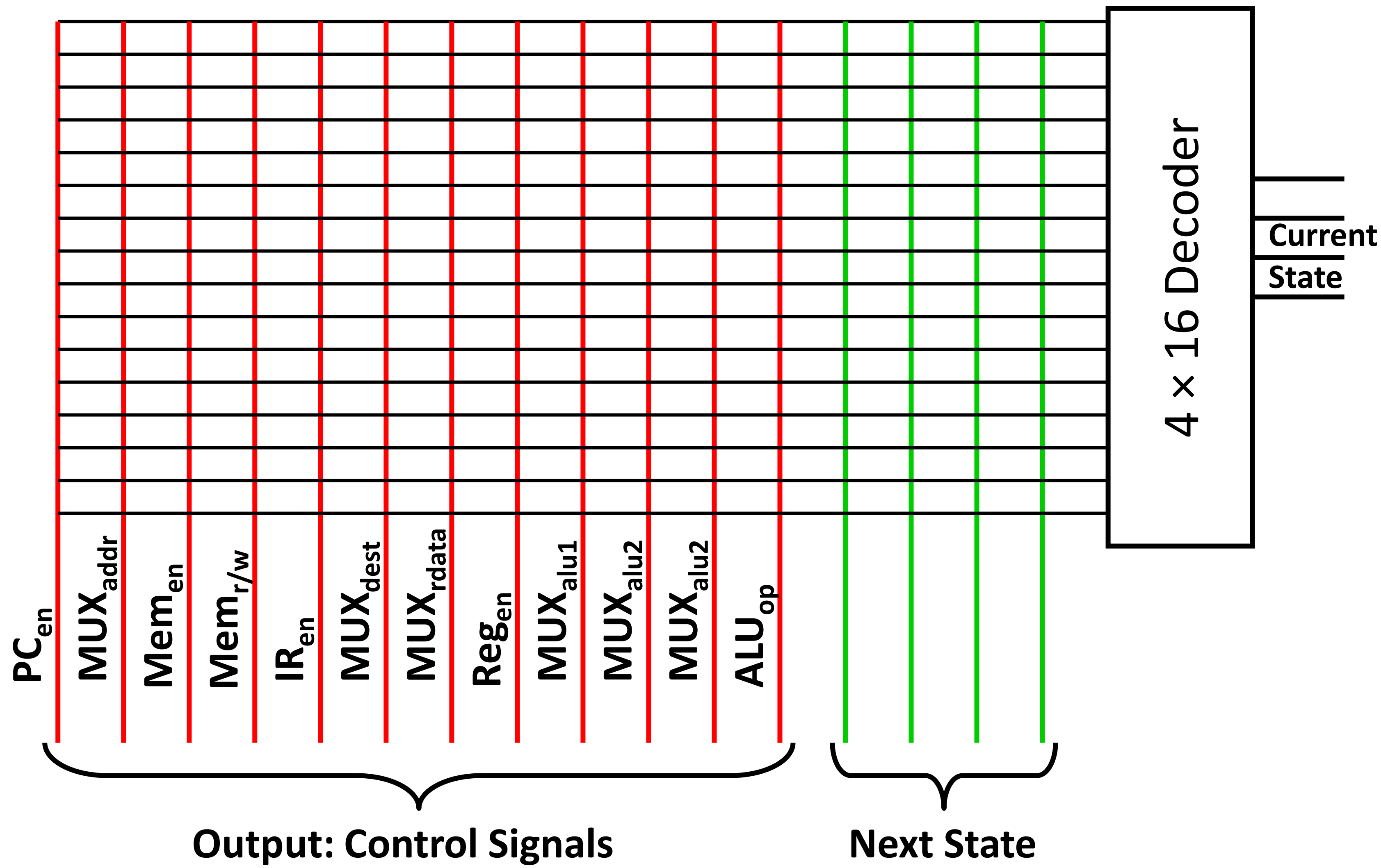
Note: we aren't worrying about JALR instruction in hardware going forward

<u>Poll:</u> How many bits of storage are needed to store the state?

Implementing FSM



Building the Control ROM

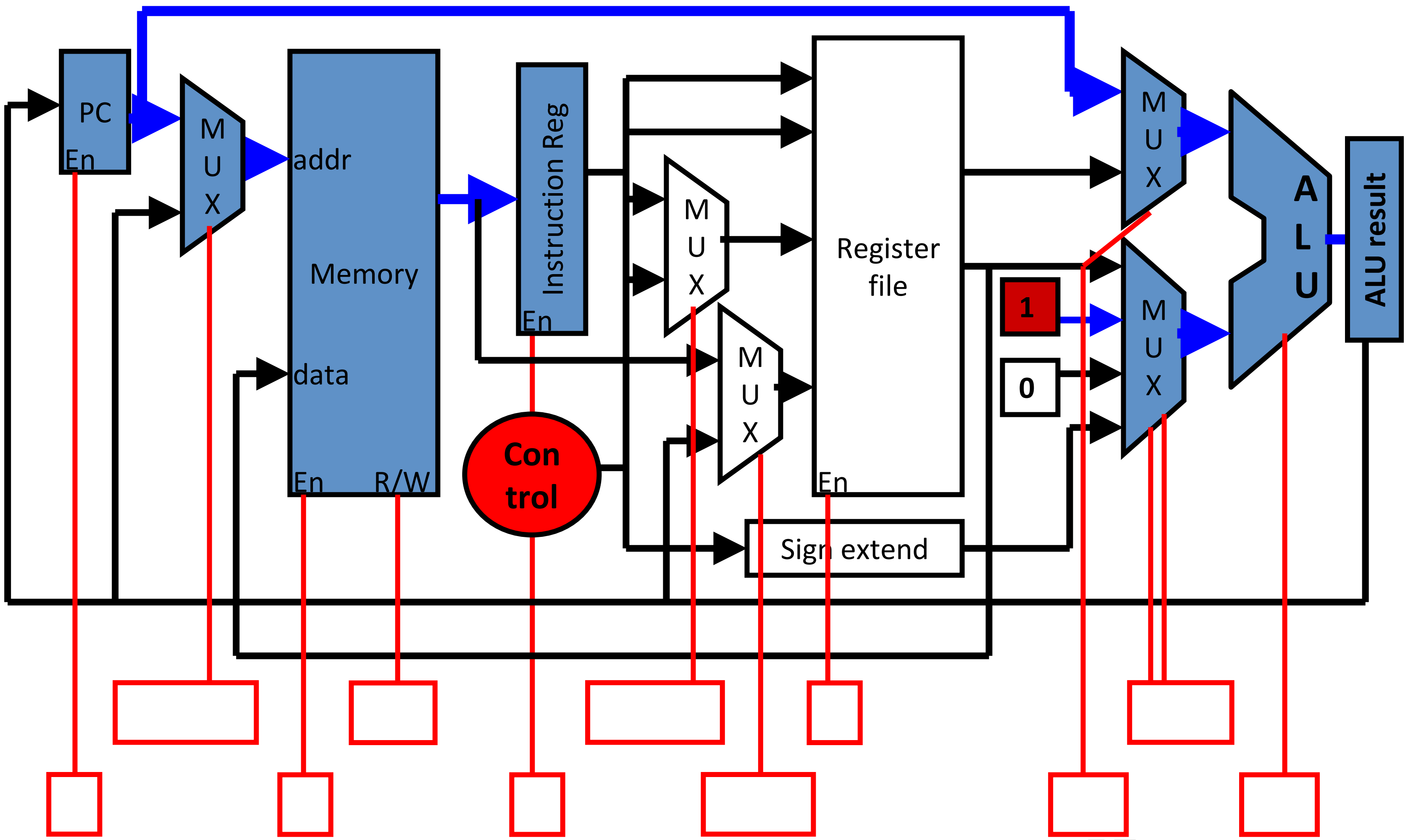


First Cycle (State 0) Fetch Instr

- 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

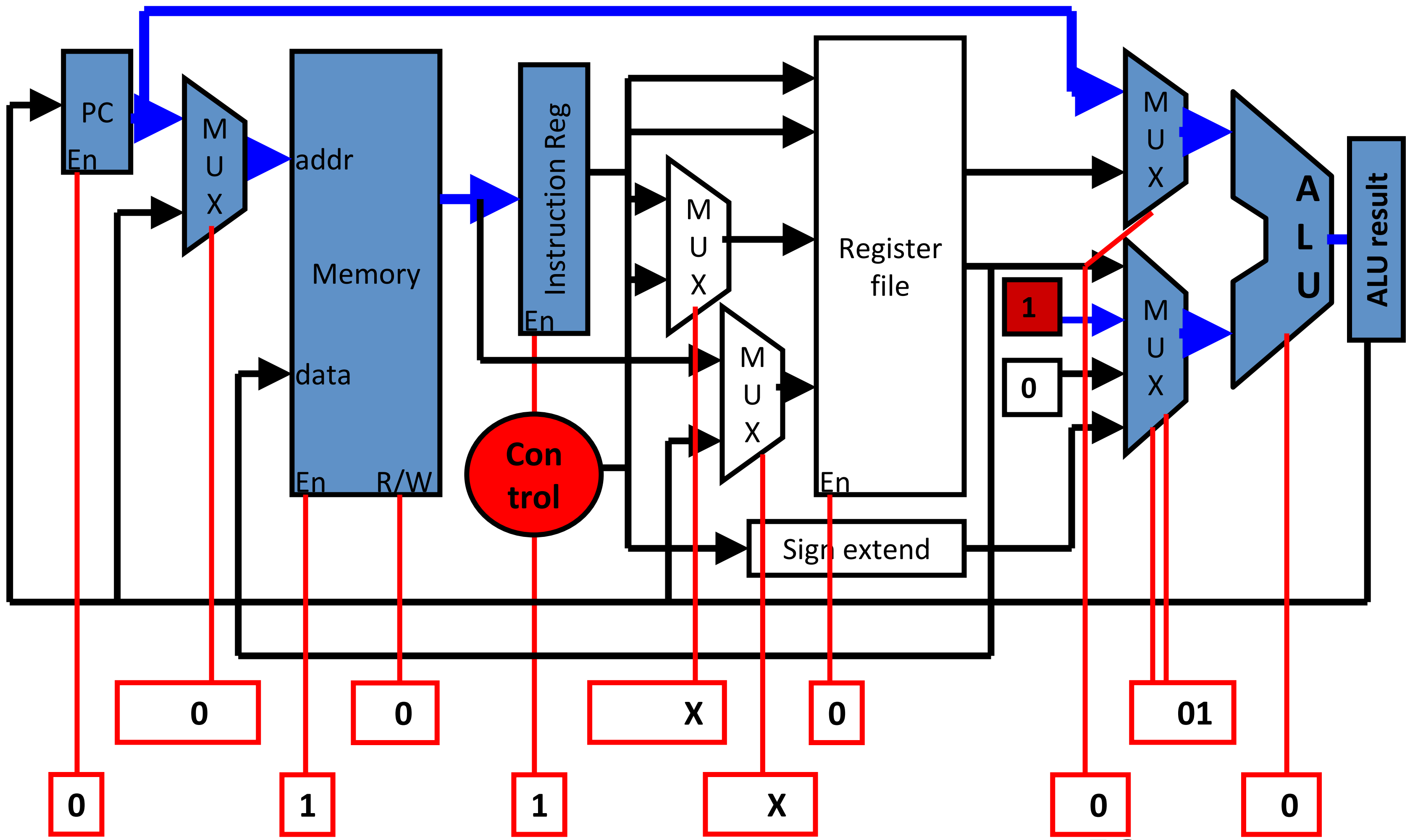
First Cycle (State 0) Fetch Instr

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

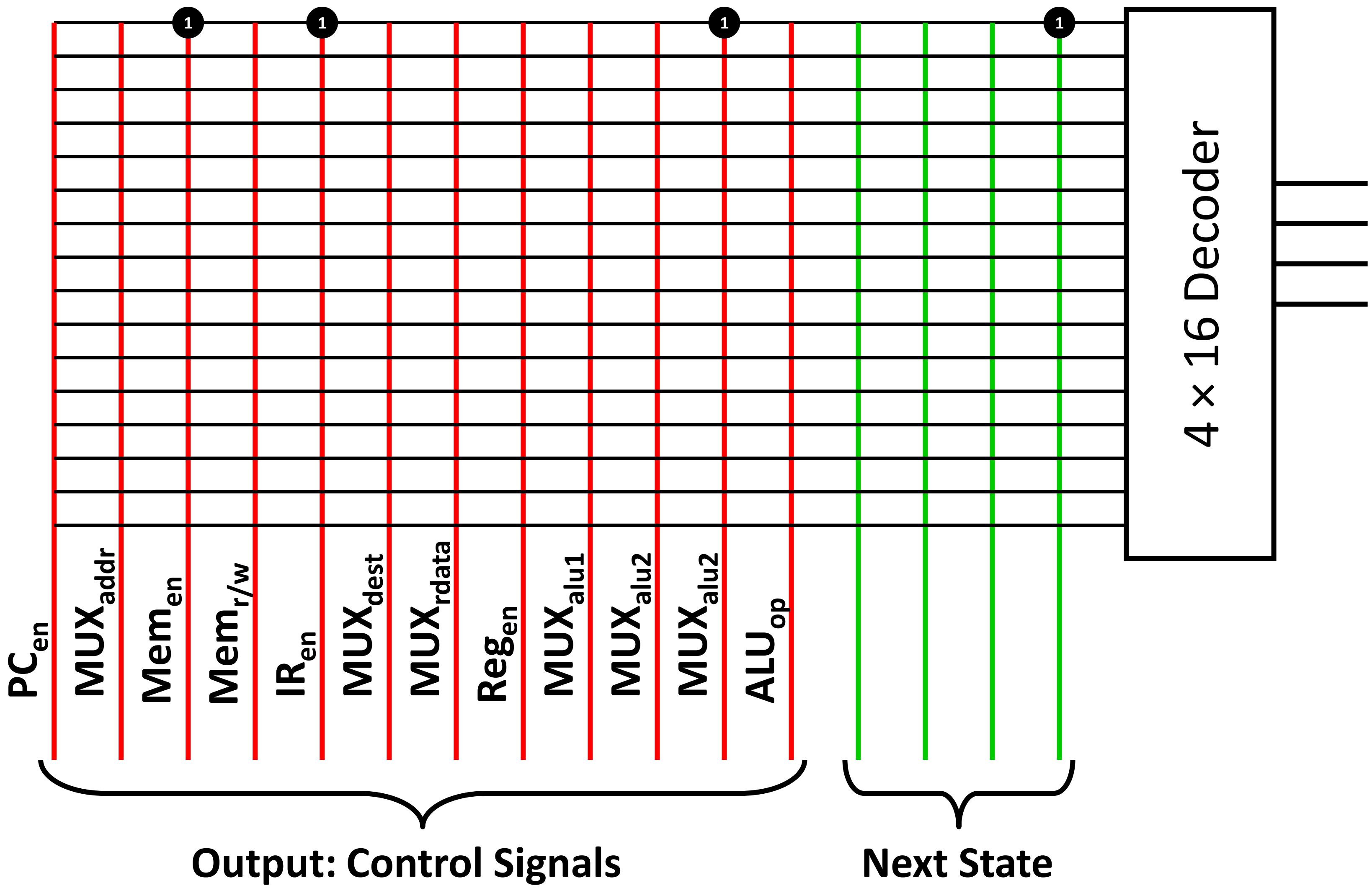


First Cycle (State 0) Fetch Instr

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



Building the Control ROM



Next time

- Finish up multi-cycle processors
- Introduce pipelining