# EECS 370 - Lecture 11

## Single & Multi-Cycle Data Path

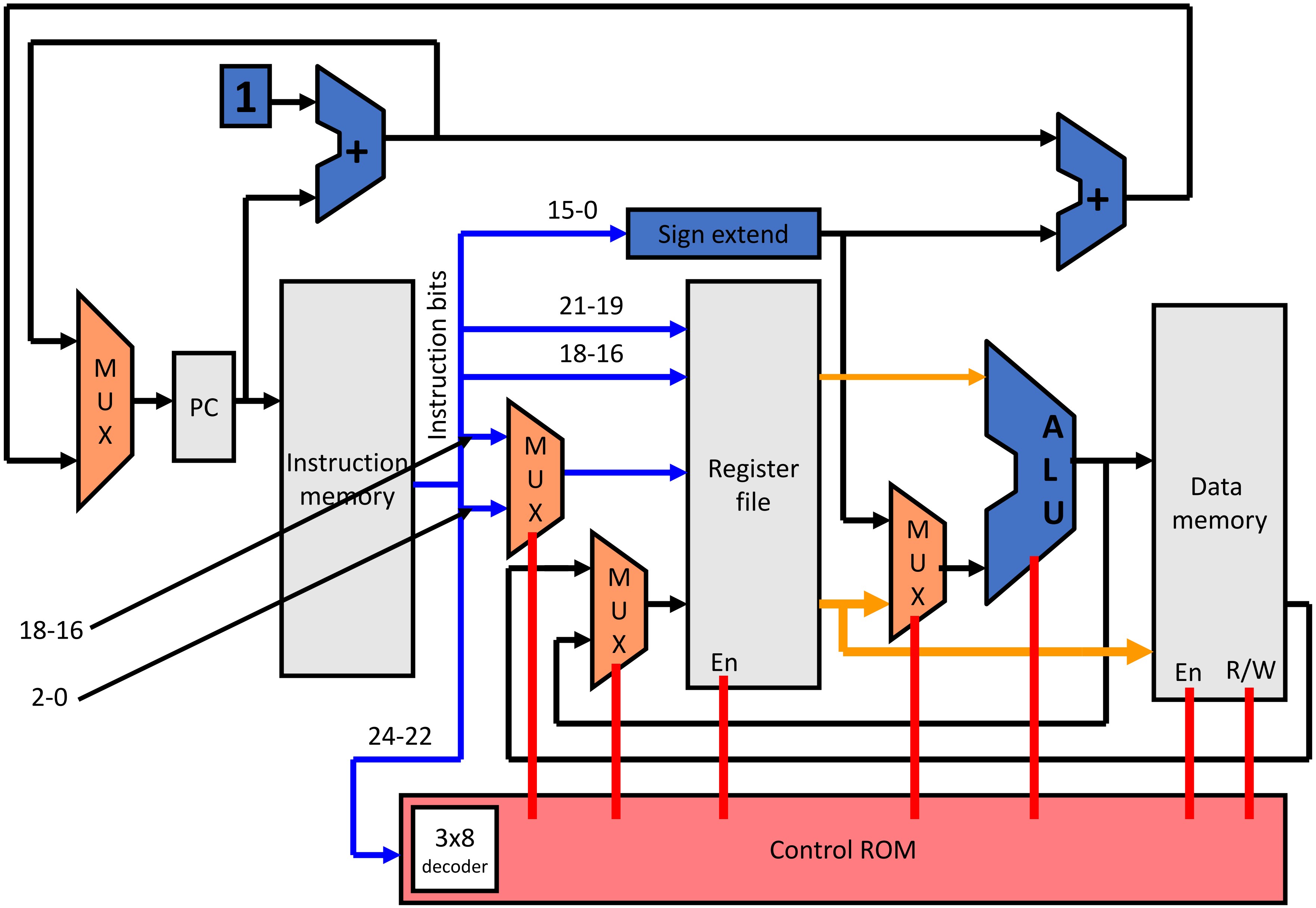
# Announcements

- P2
  - Two parts: part a is due **next Thursday**
- HW 2
  - Posted on website, due next **Fri**
- Lab due Wed @ 11:55 pm
- Midterm exam after break **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

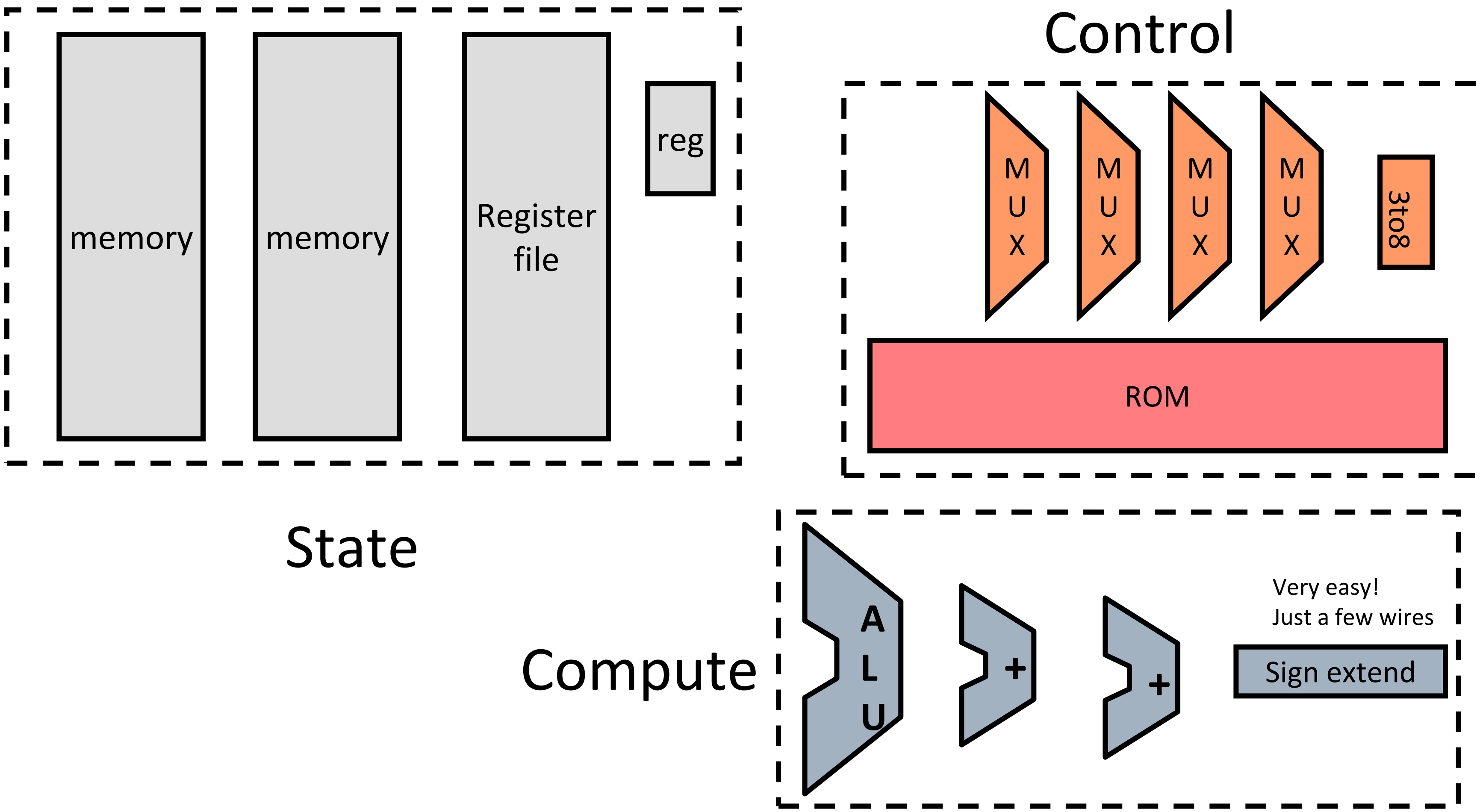
# Single-Cycle Processor Design

- 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

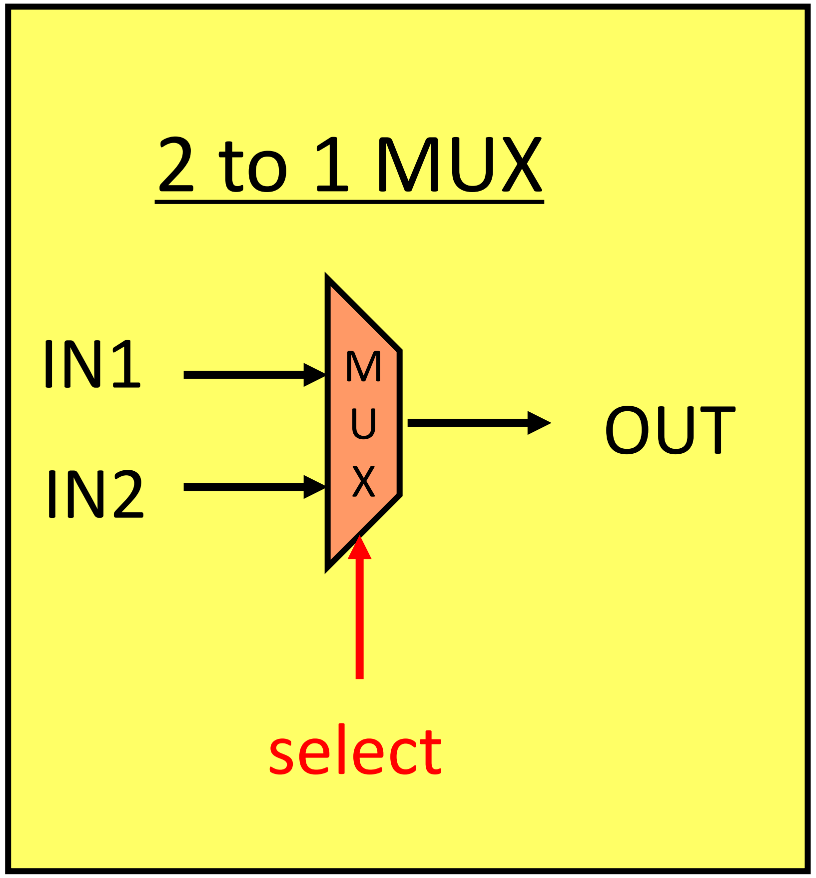


# Building Blocks for the LC2K




Here are the pieces, go build yourself a processor!

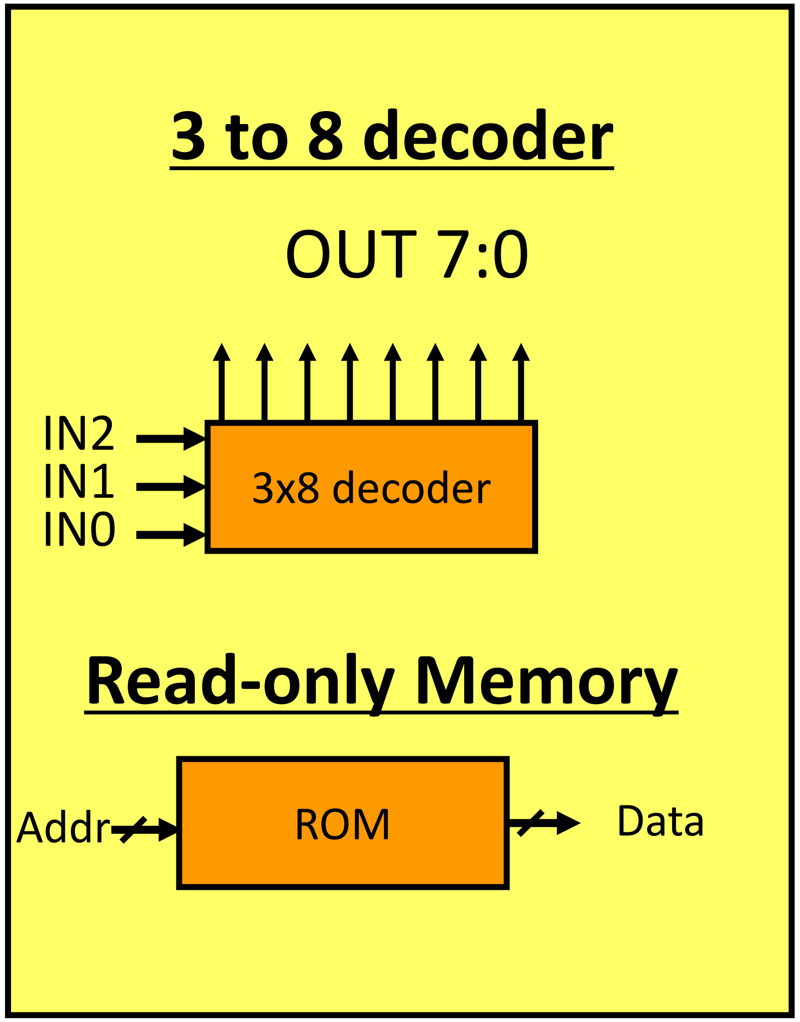
# Control Building Blocks (1)




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

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

# Control Building Blocks (2)



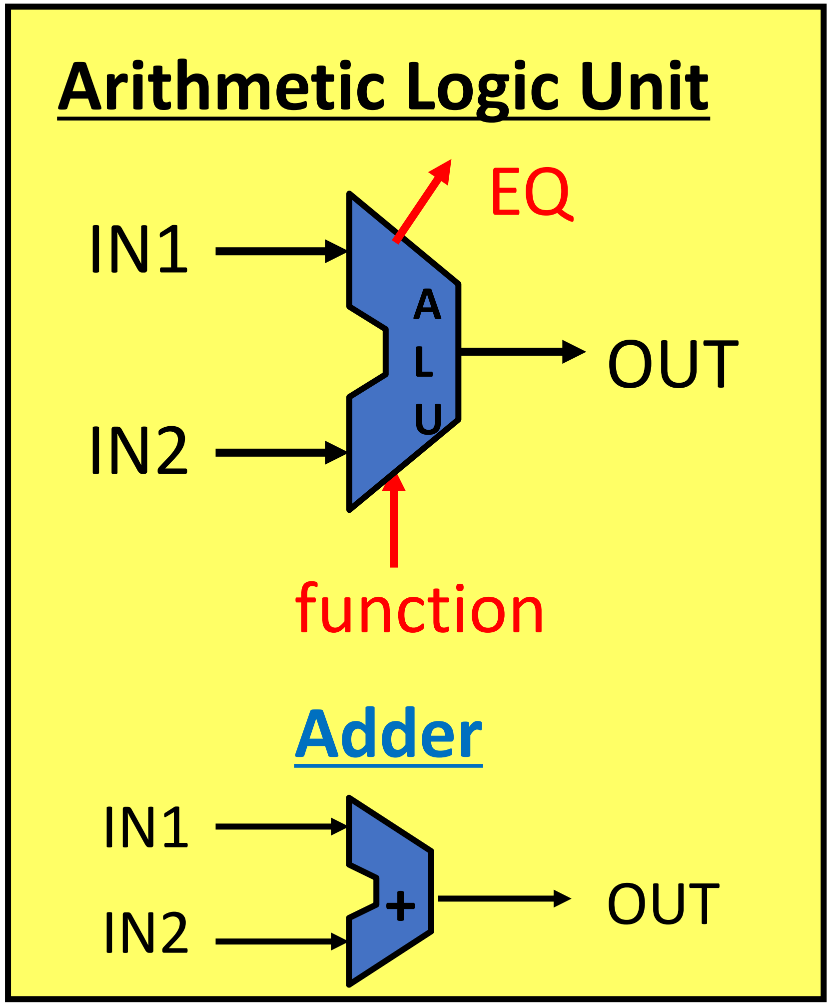

Decoder activates one of the output lines based on the input

| IN 210 | OUT 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:

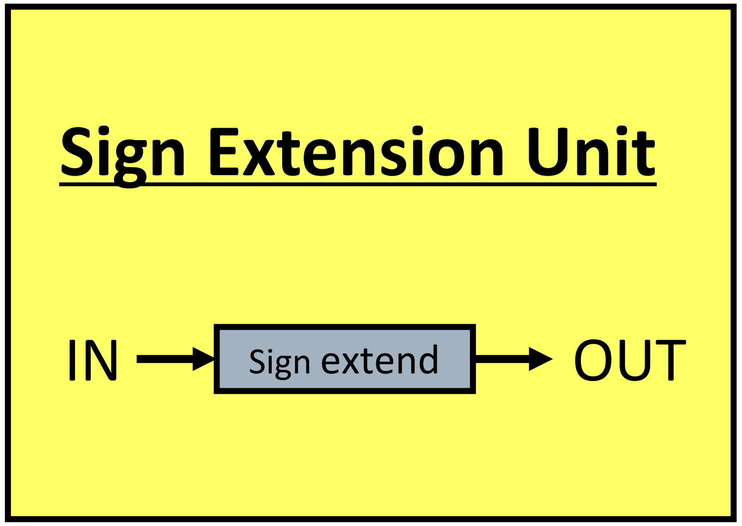
f=0 is add

f=1 is nor

For other processors, there are many more functions.

Just adds

# Compute Building Blocks (2)



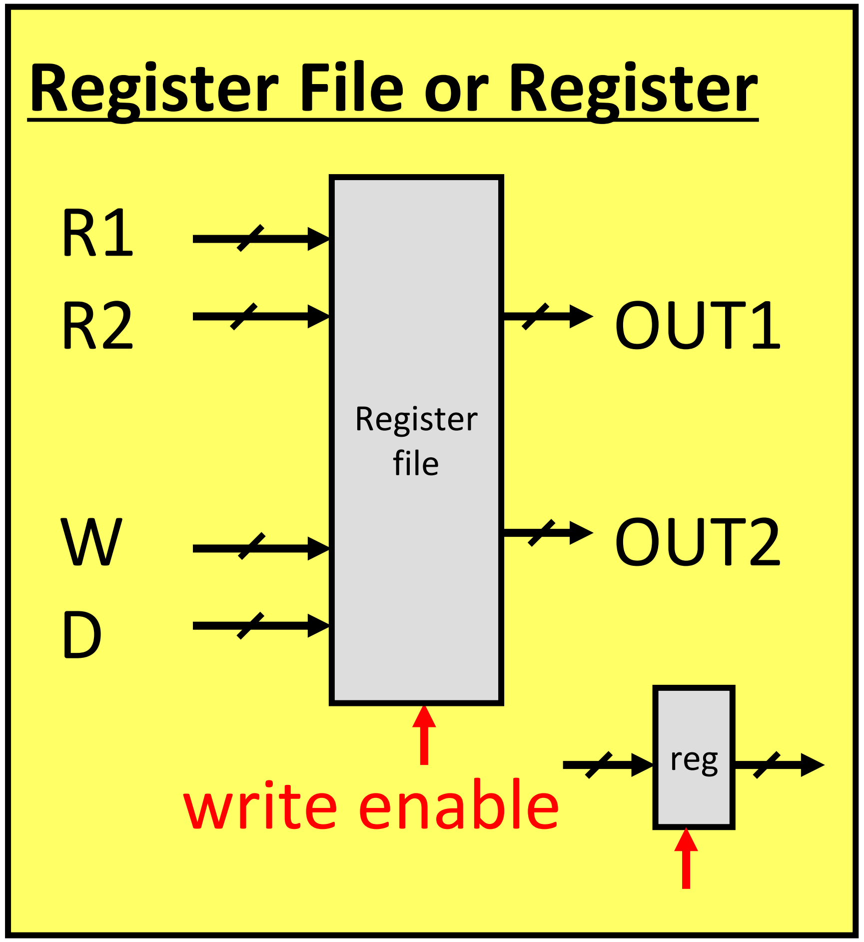

Sign extend (SE) input by replicating the MSB to width of output

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)

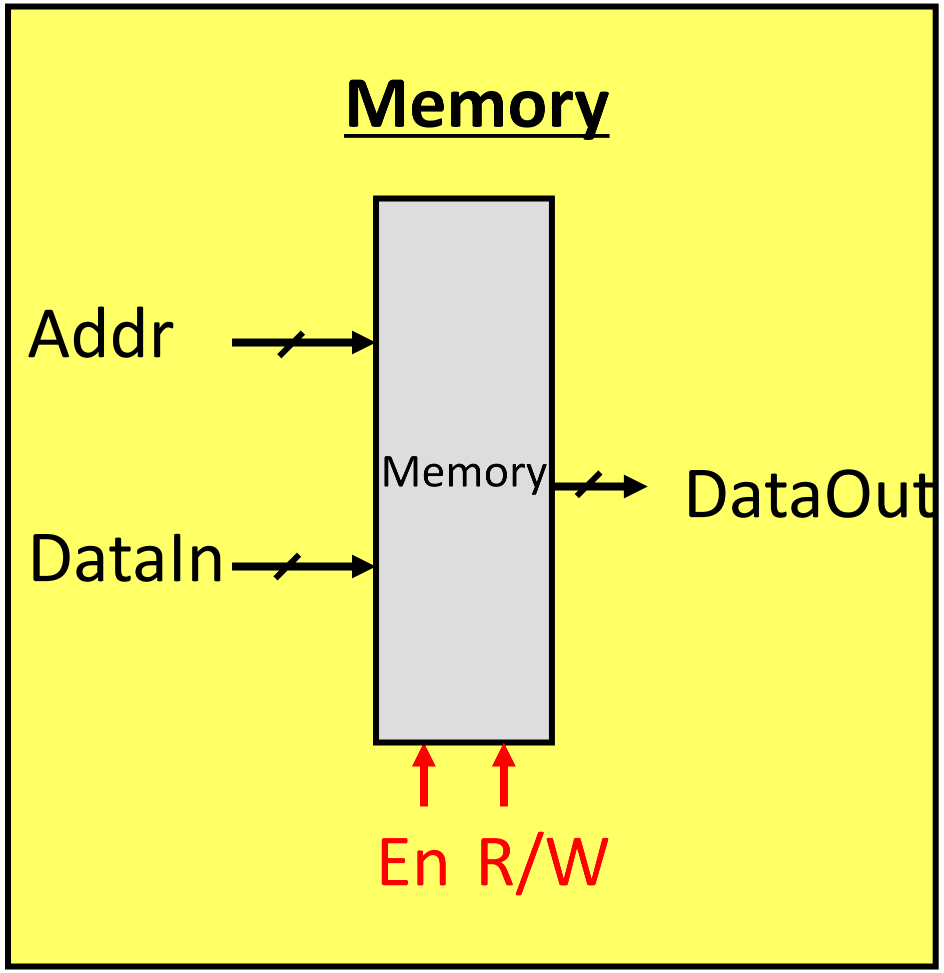
* Ri specifies register number to read

* W specifies register number to write

* D specifies data to write

**<u>Poll:</u> 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

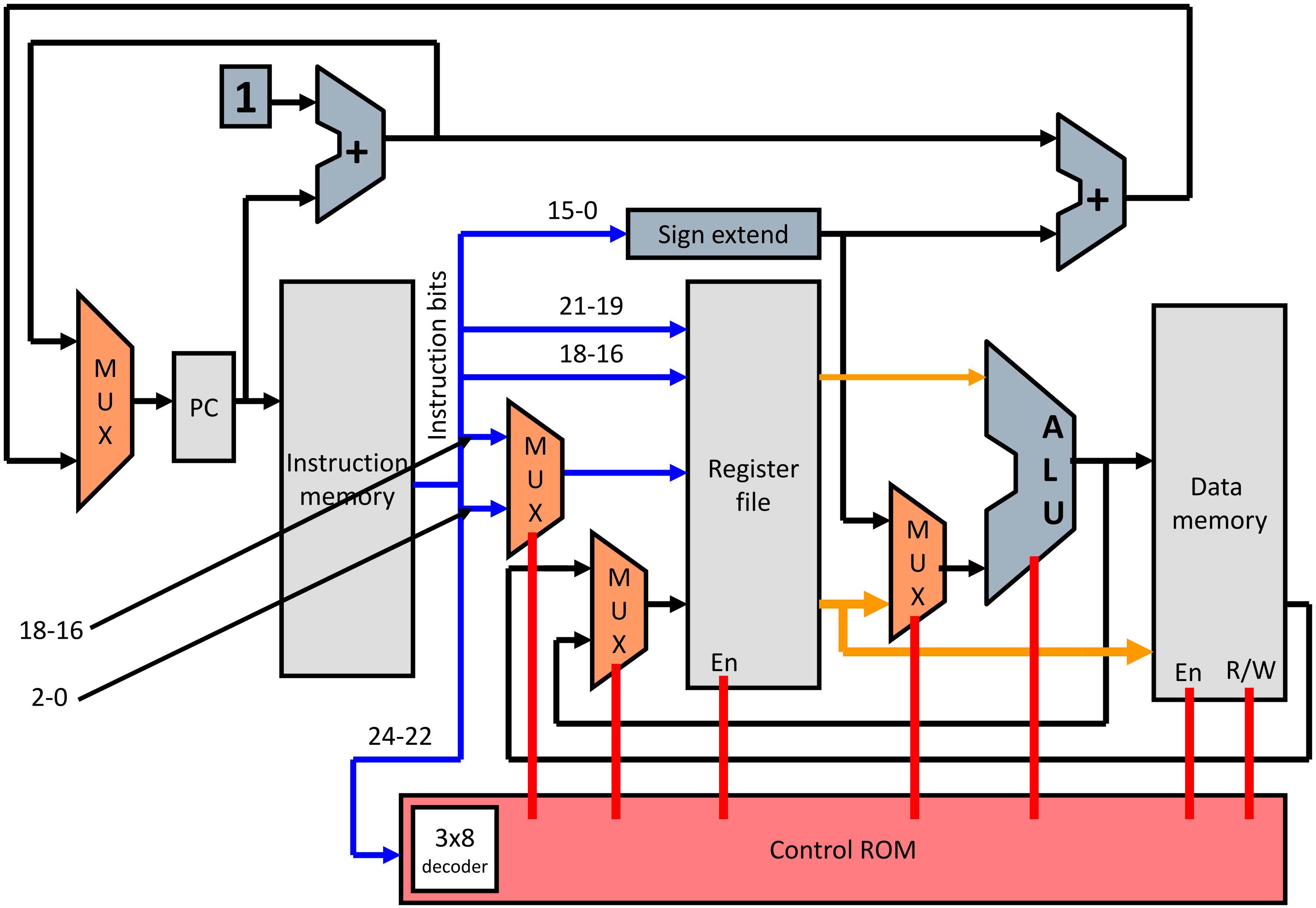
- R type instructions (add ‘000’, nor ‘001’)

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

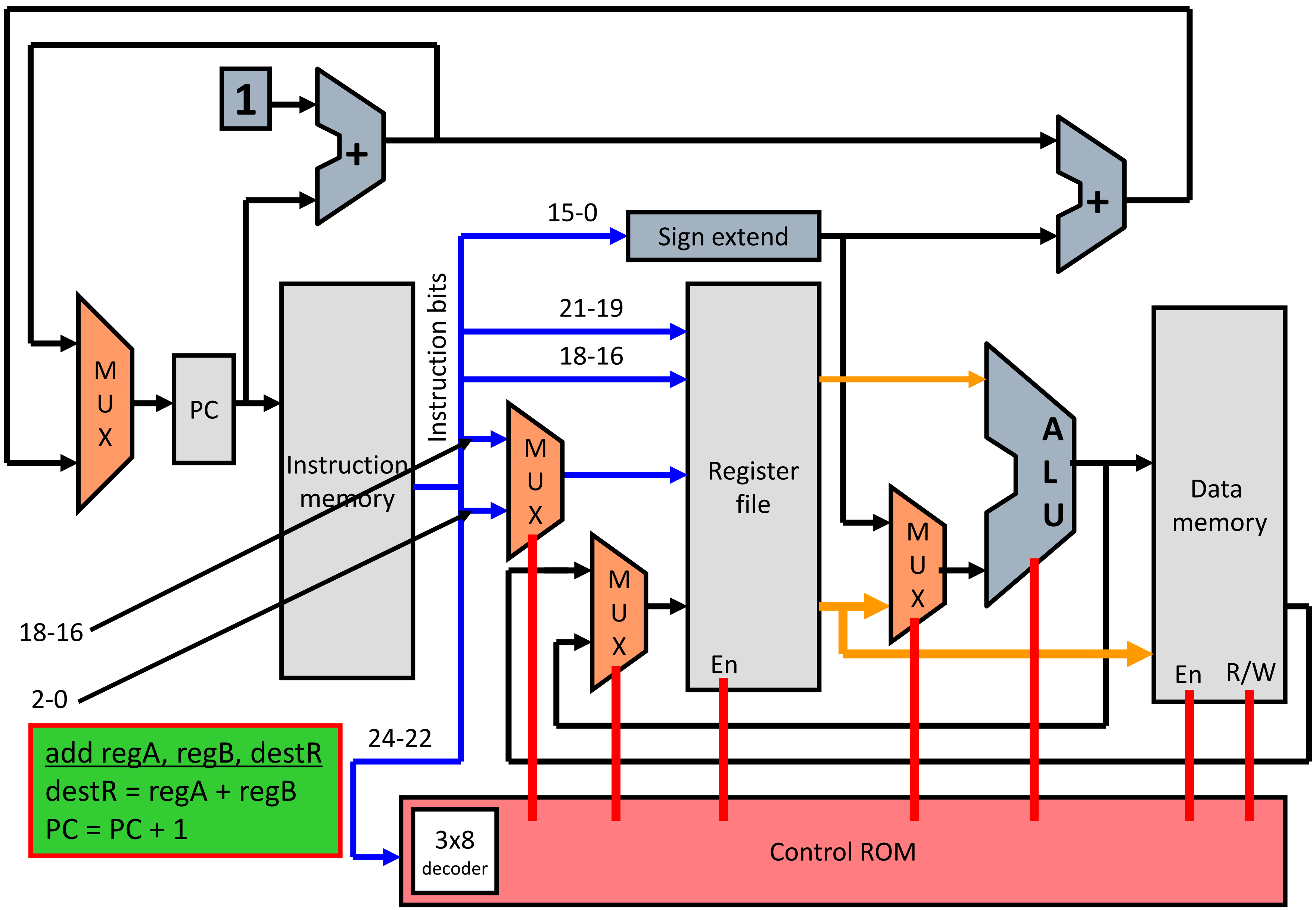
- I type instructions (lw ‘010’, sw ‘011’, beq ‘100’)

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

# LC2K Single-Cycle Datapath Implementation

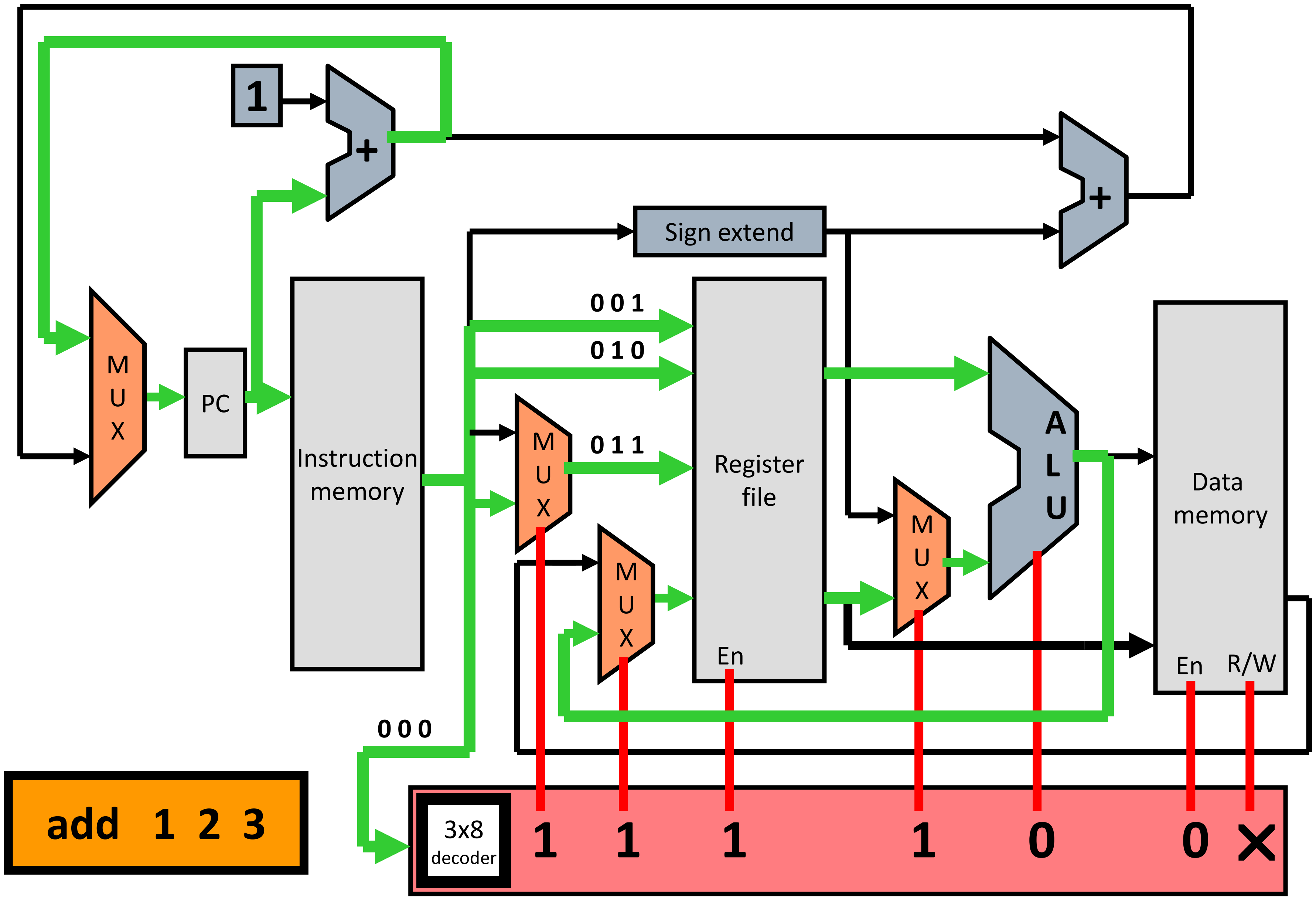


# Executing an ADD Instruction on LC2K Datapath



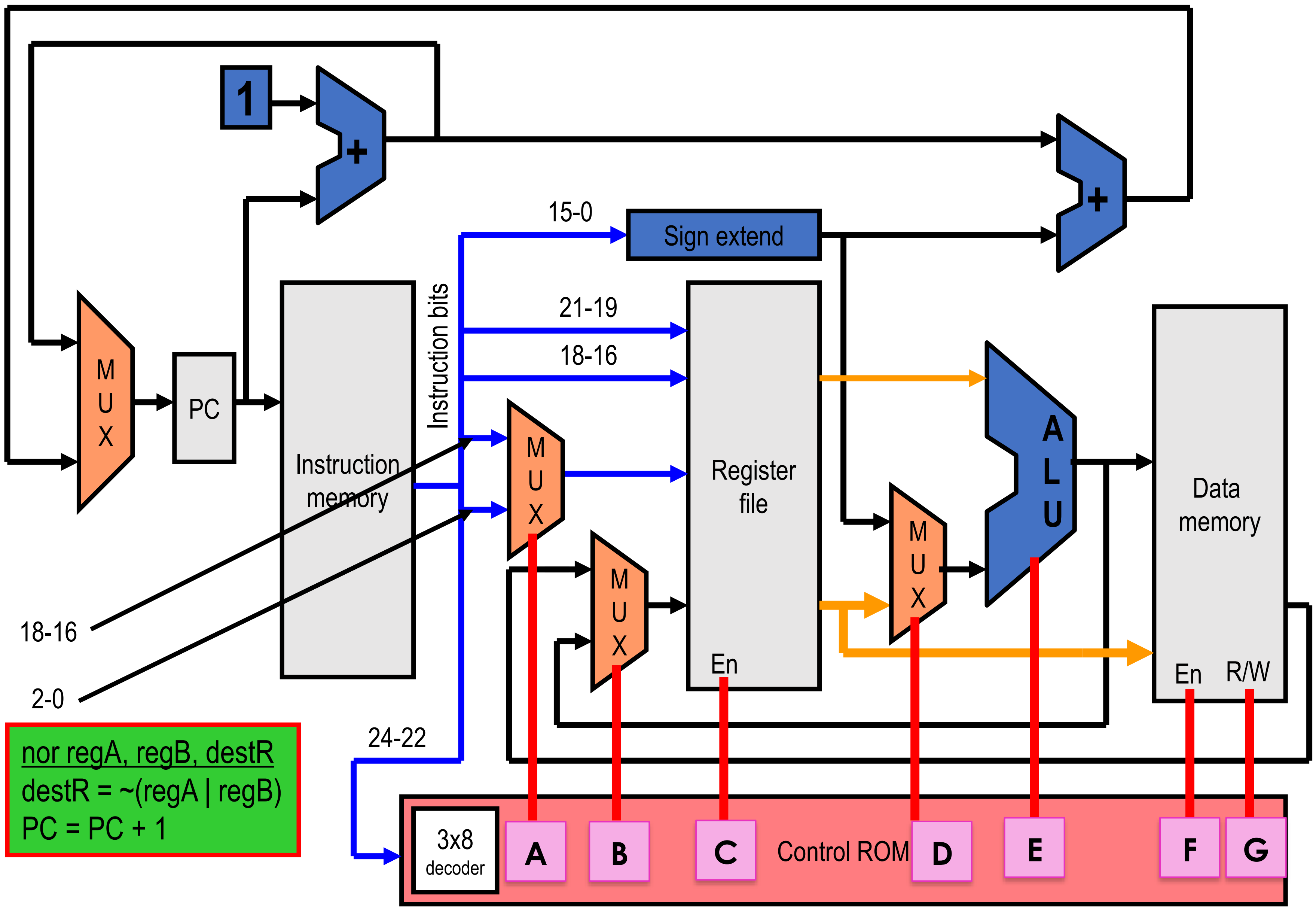

<u>add regA, regB, destR</u>
destR = regA + regB
PC = PC + 1

# Executing an ADD Instruction on LC2K Datapath



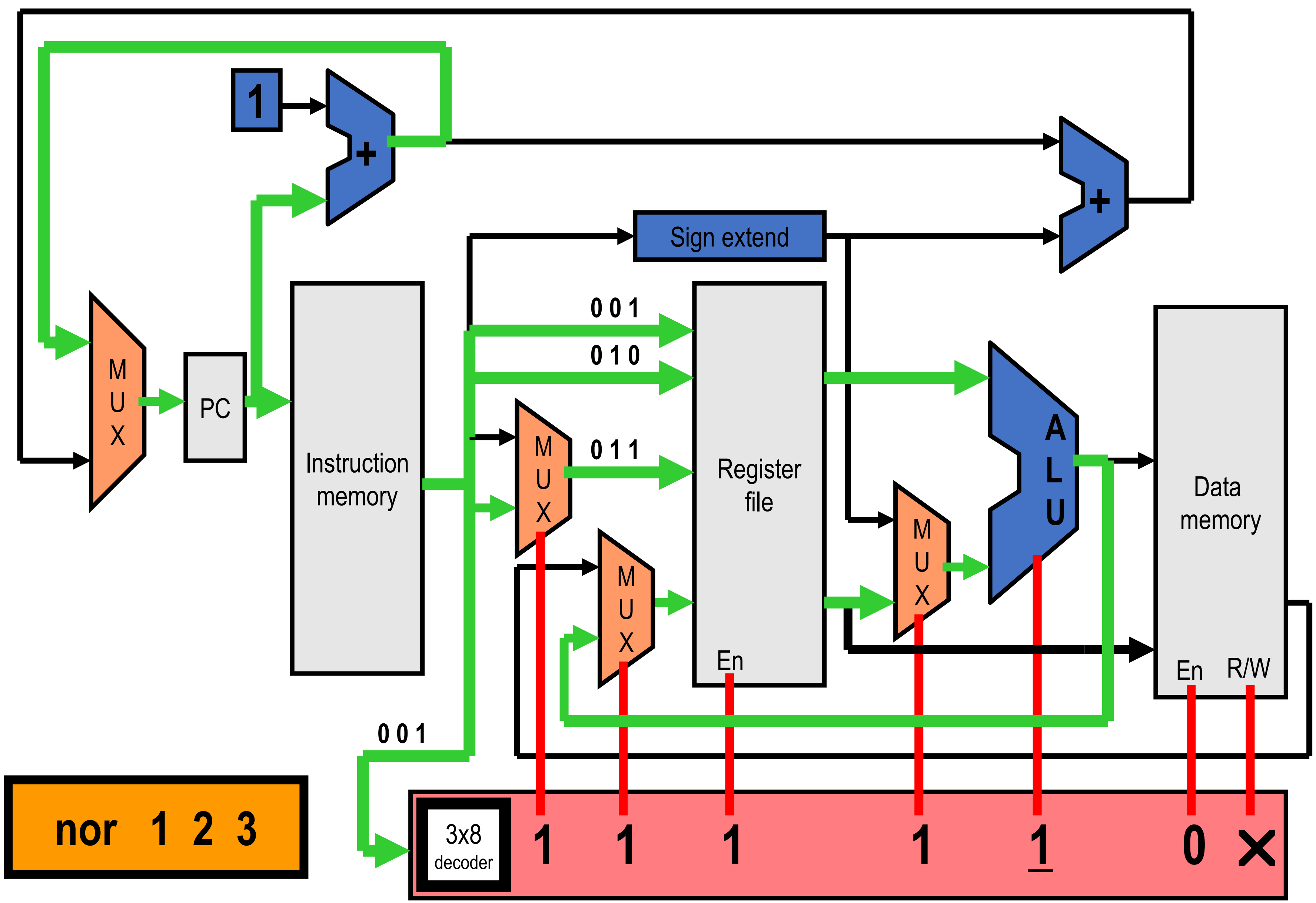
**Poll:** Which control bits need to be different from ADD?

# Executing a NOR Instruction




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

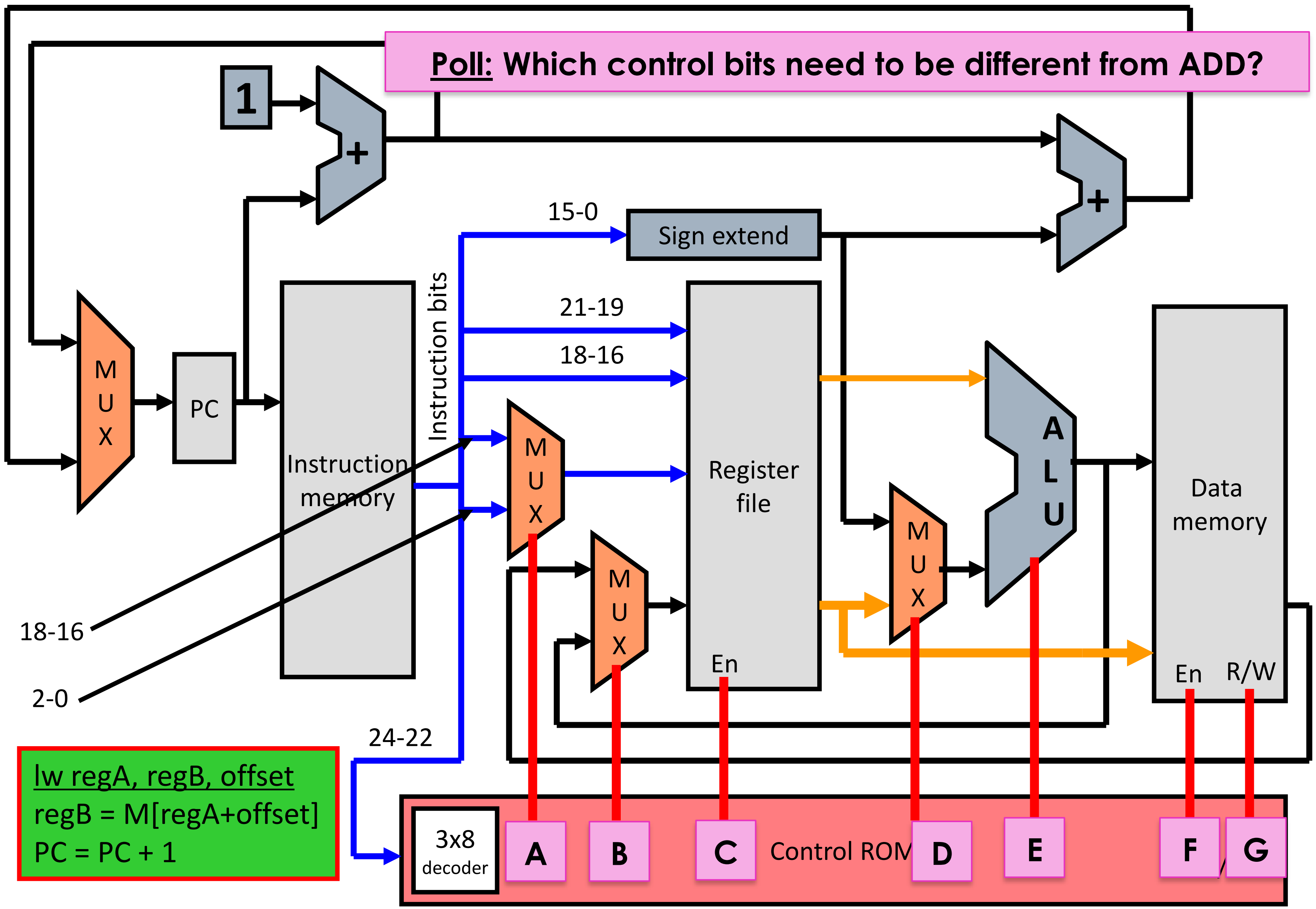
# Executing a NOR Instruction



# Executing a LW Instruction

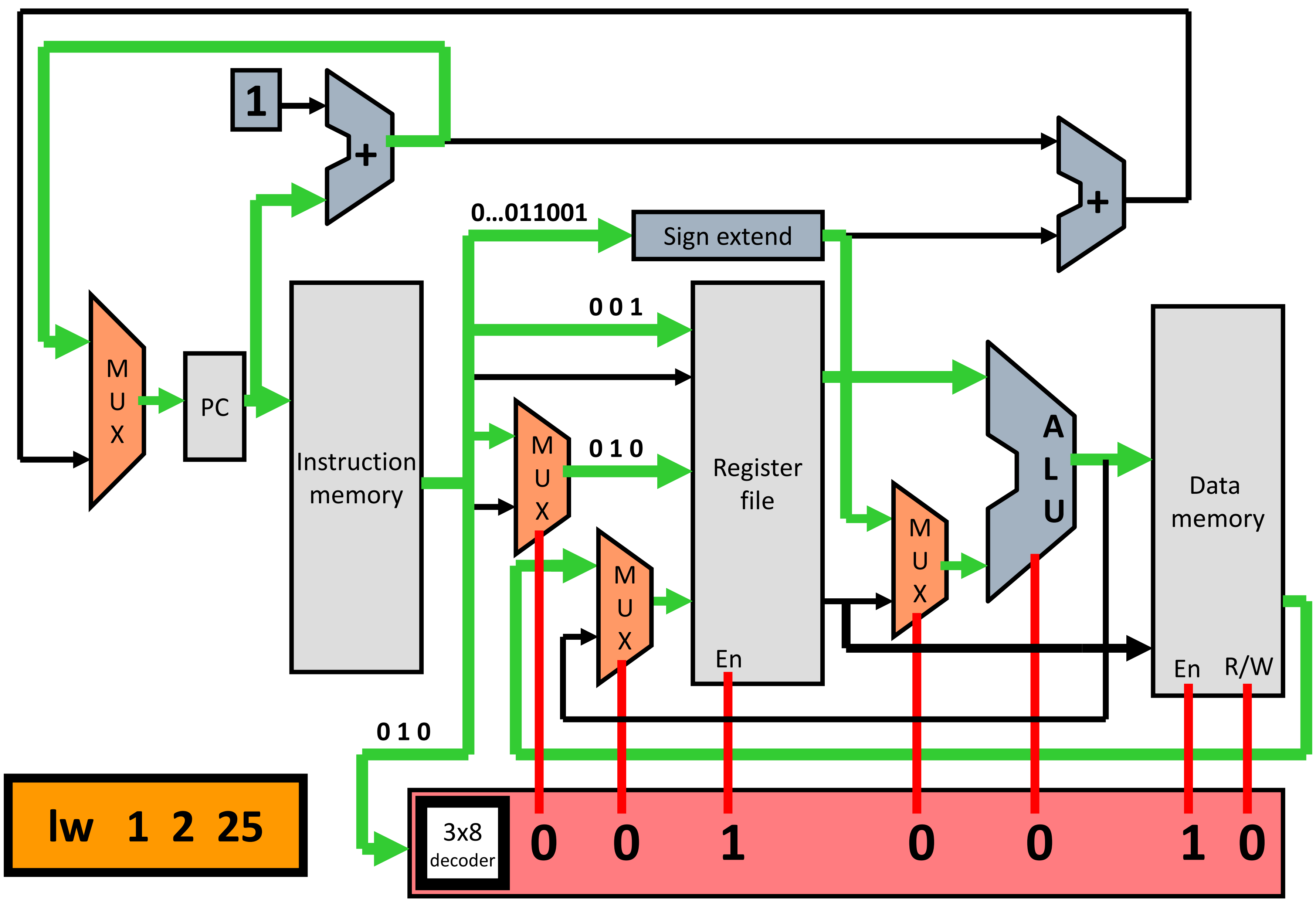
| 31-25 | 24-22 | 21-19 | 18-16 | 15-0 |
|---|---|---|---|---|
| *unused* | **opcode** | **regA** | **regB** | **offset** |

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



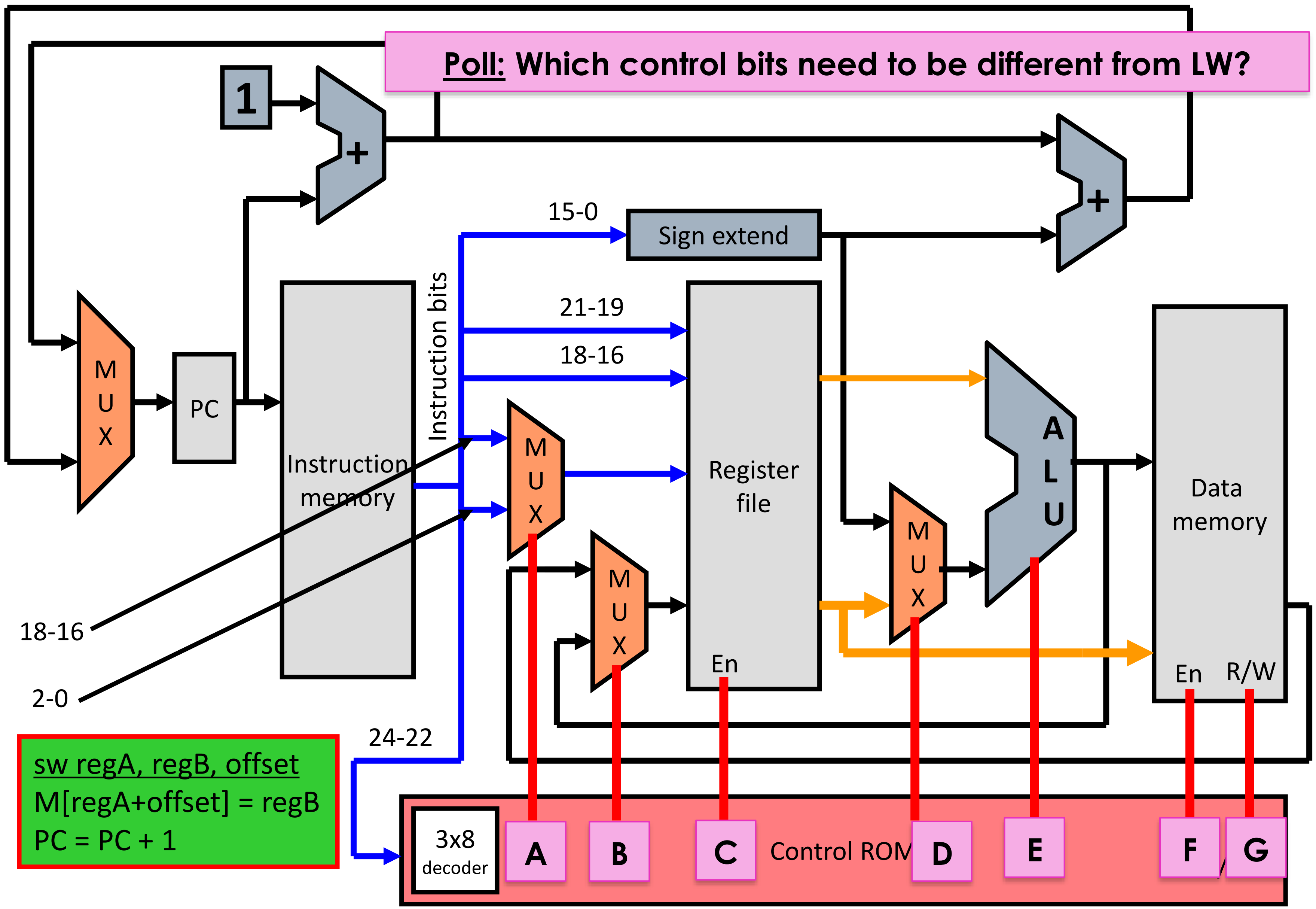

lw regA, regB, offset
regB = M[regA+offset]
PC = PC + 1

# Executing a LW Instruction



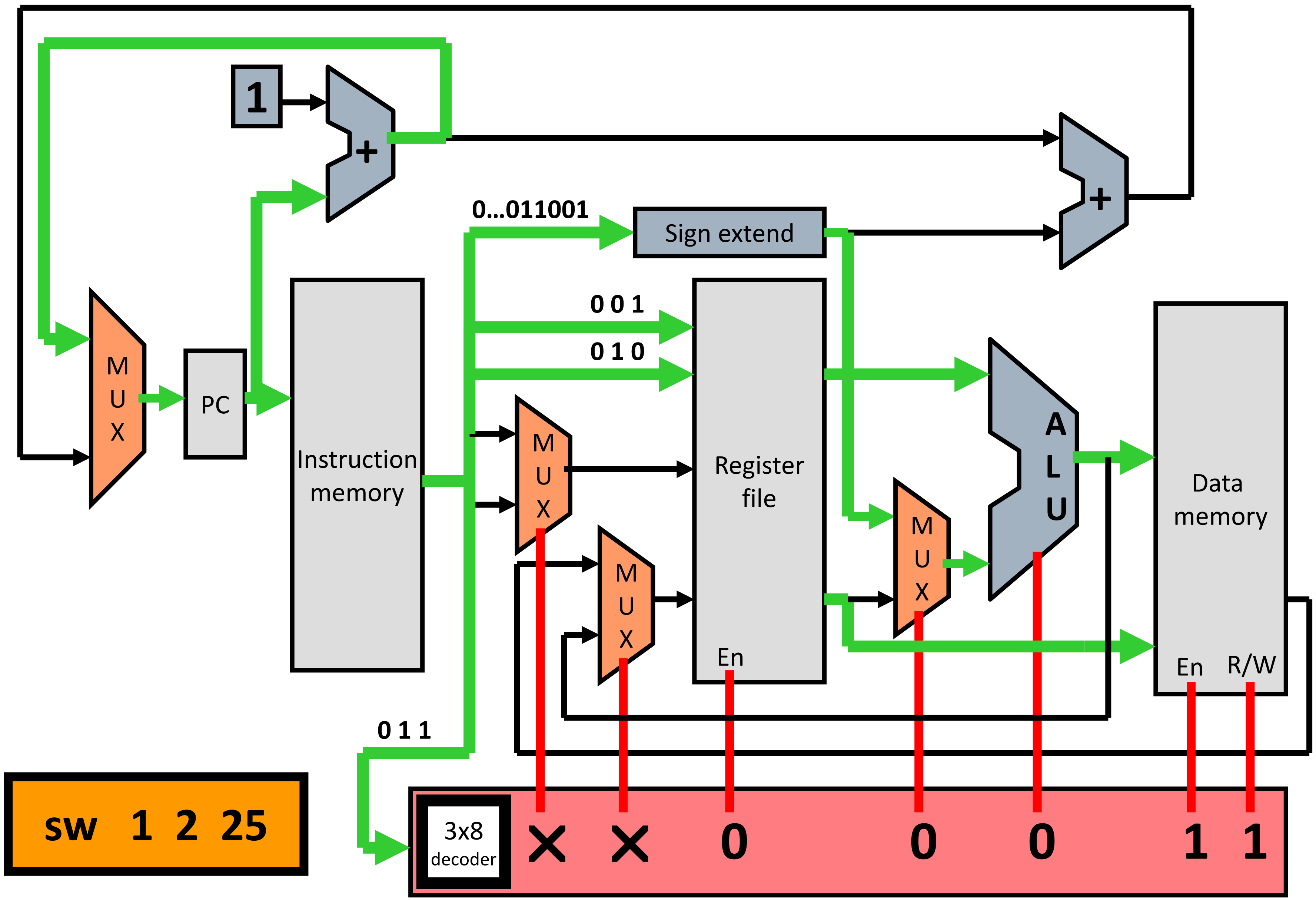
# Executing a SW Instruction

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

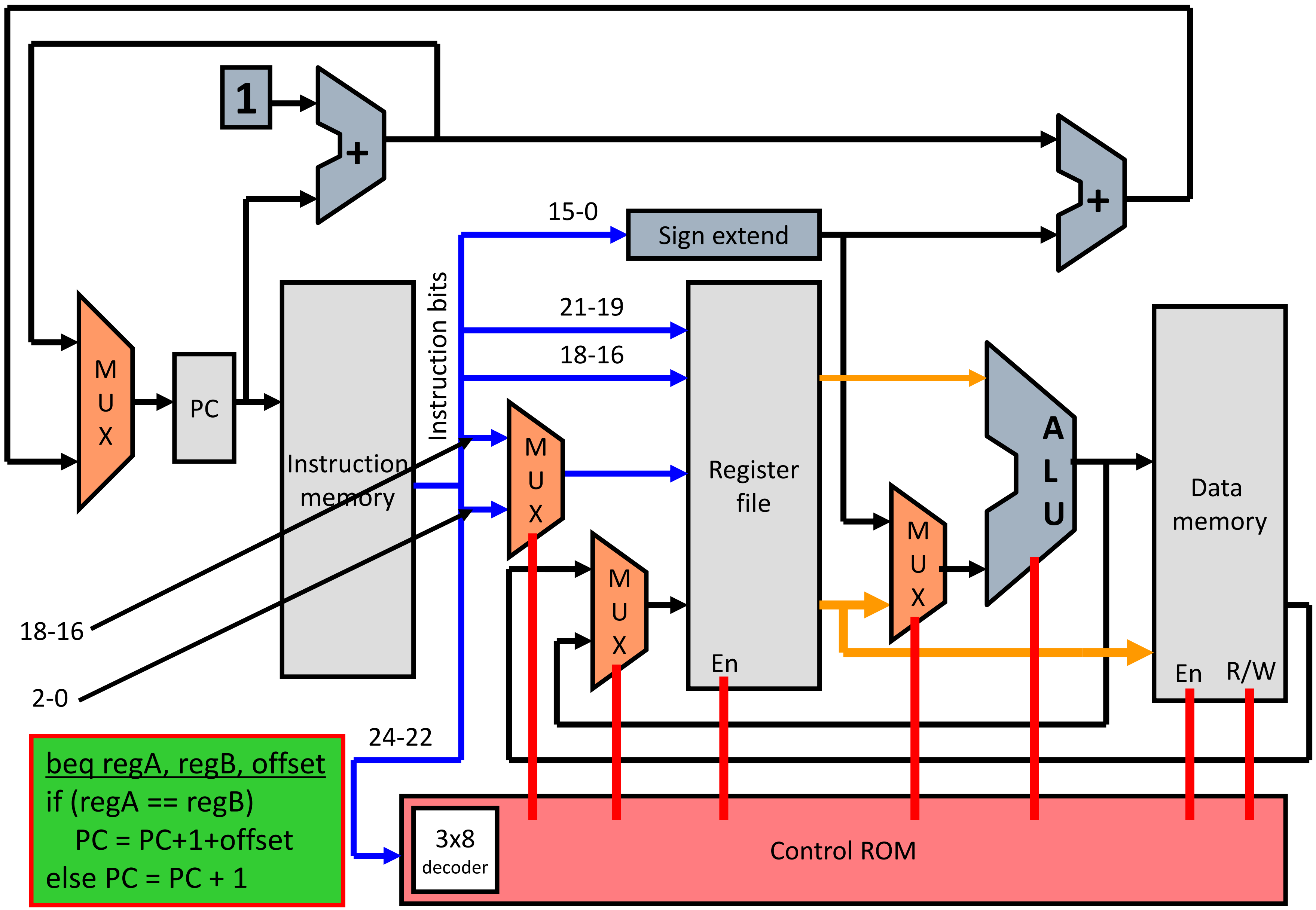



<u>sw regA, regB, offset</u>
M[regA+offset] = regB
PC = PC + 1

# Executing a SW Instruction

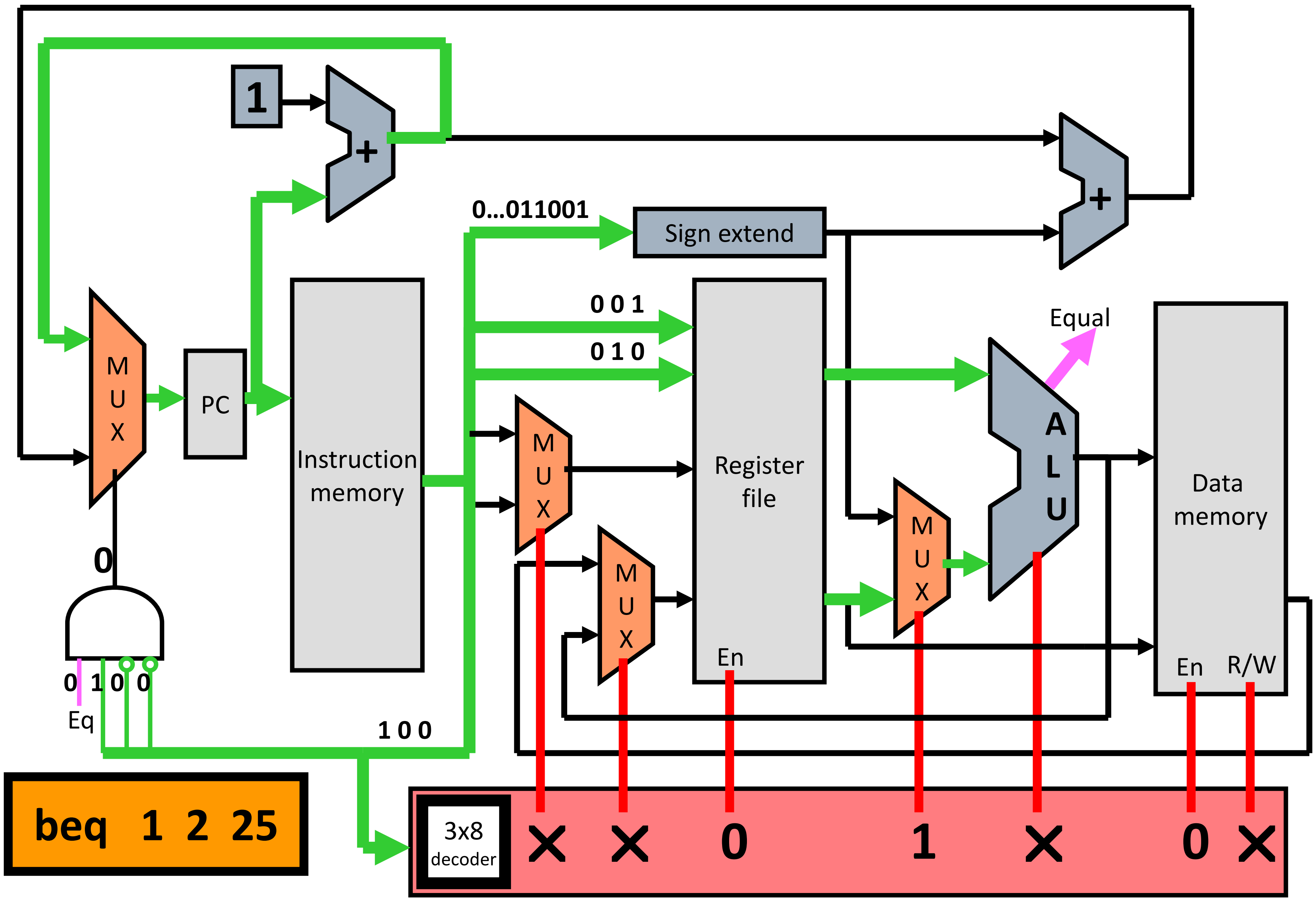


# Executing a BEQ Instruction



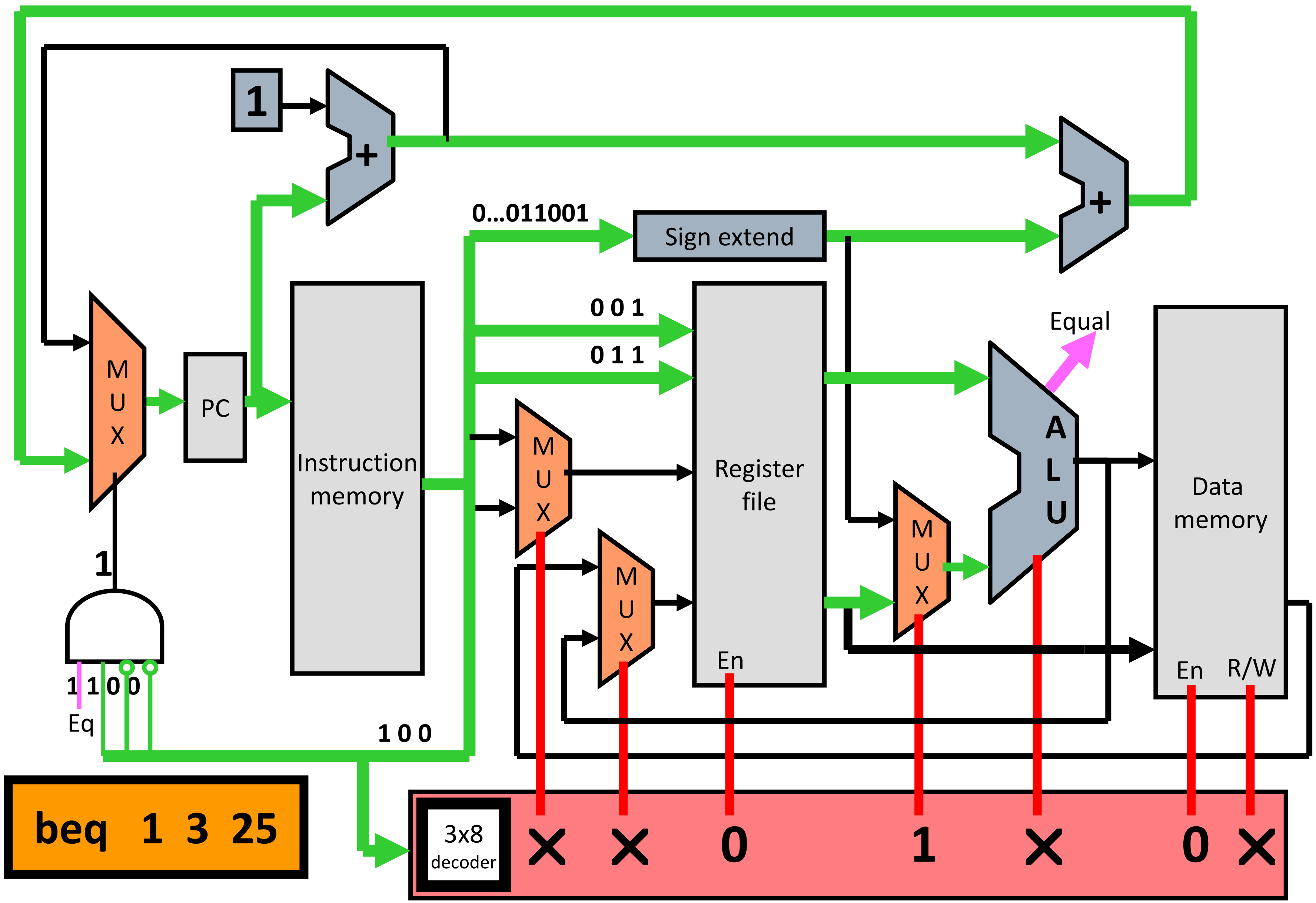

beq regA, regB, offset
if (regA == regB)
PC = PC+1+offset
else PC = PC + 1

# Executing "not taken" BEQ Instruction on LC2K Datapath




beq 1 2 25

# Executing a “taken” BEQ Instruction on LC2K Datapath



# So Far, So Good

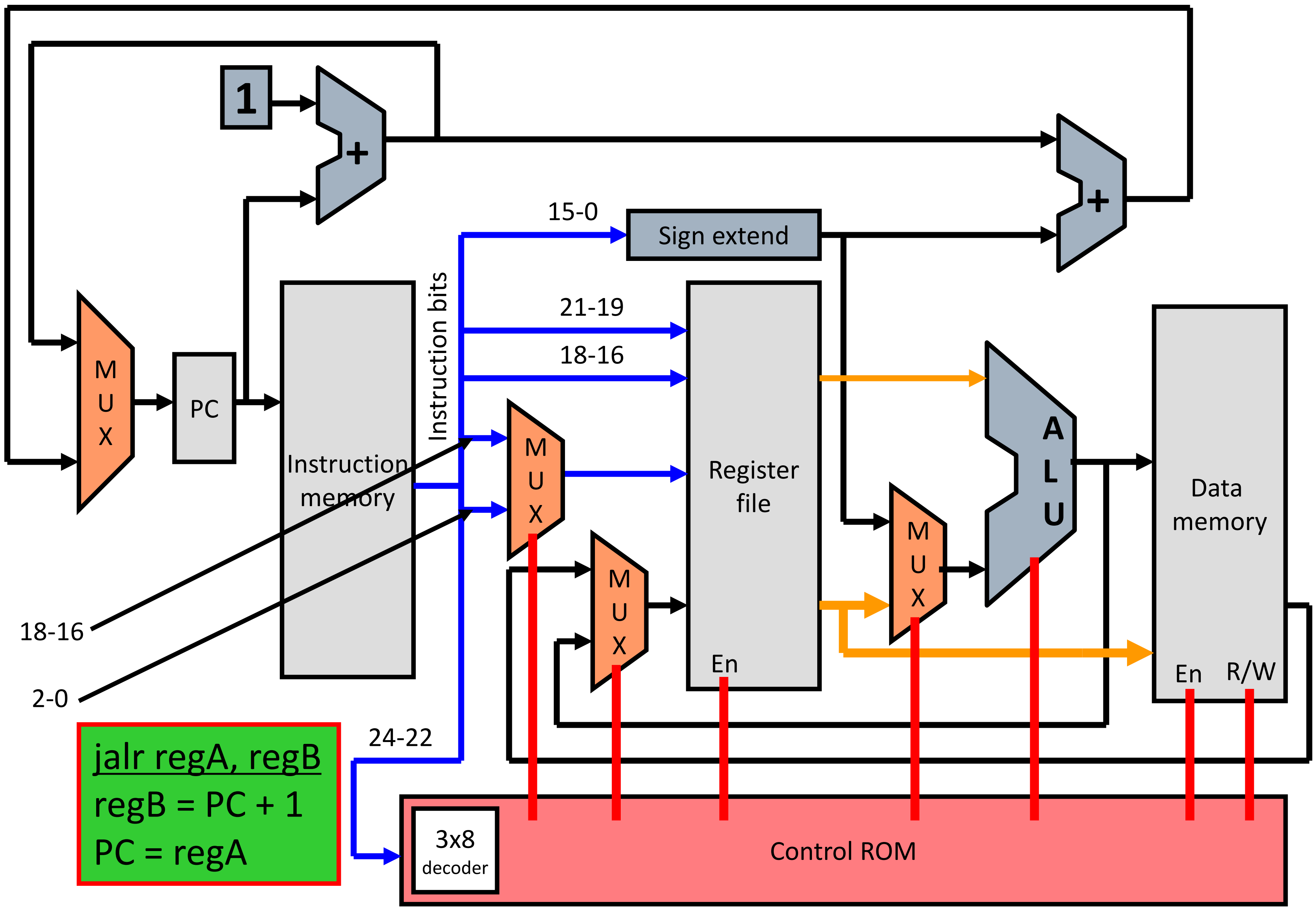
- Every architecture seems to have at least one "ugly" instruction
  - Something that doesn't elegantly fit in with the hardware we've already included

- 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

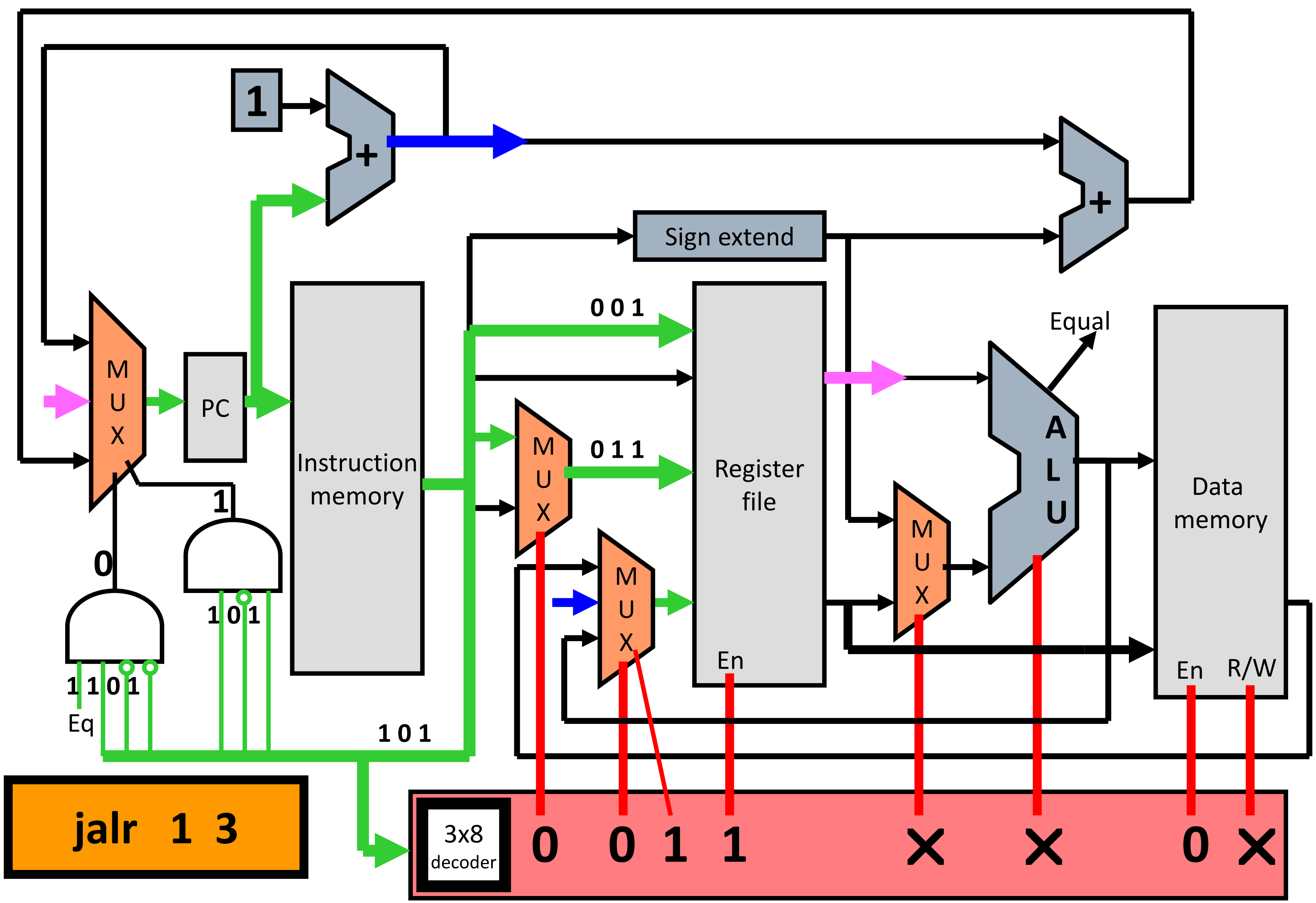



jalr regA, regB

regB = PC + 1

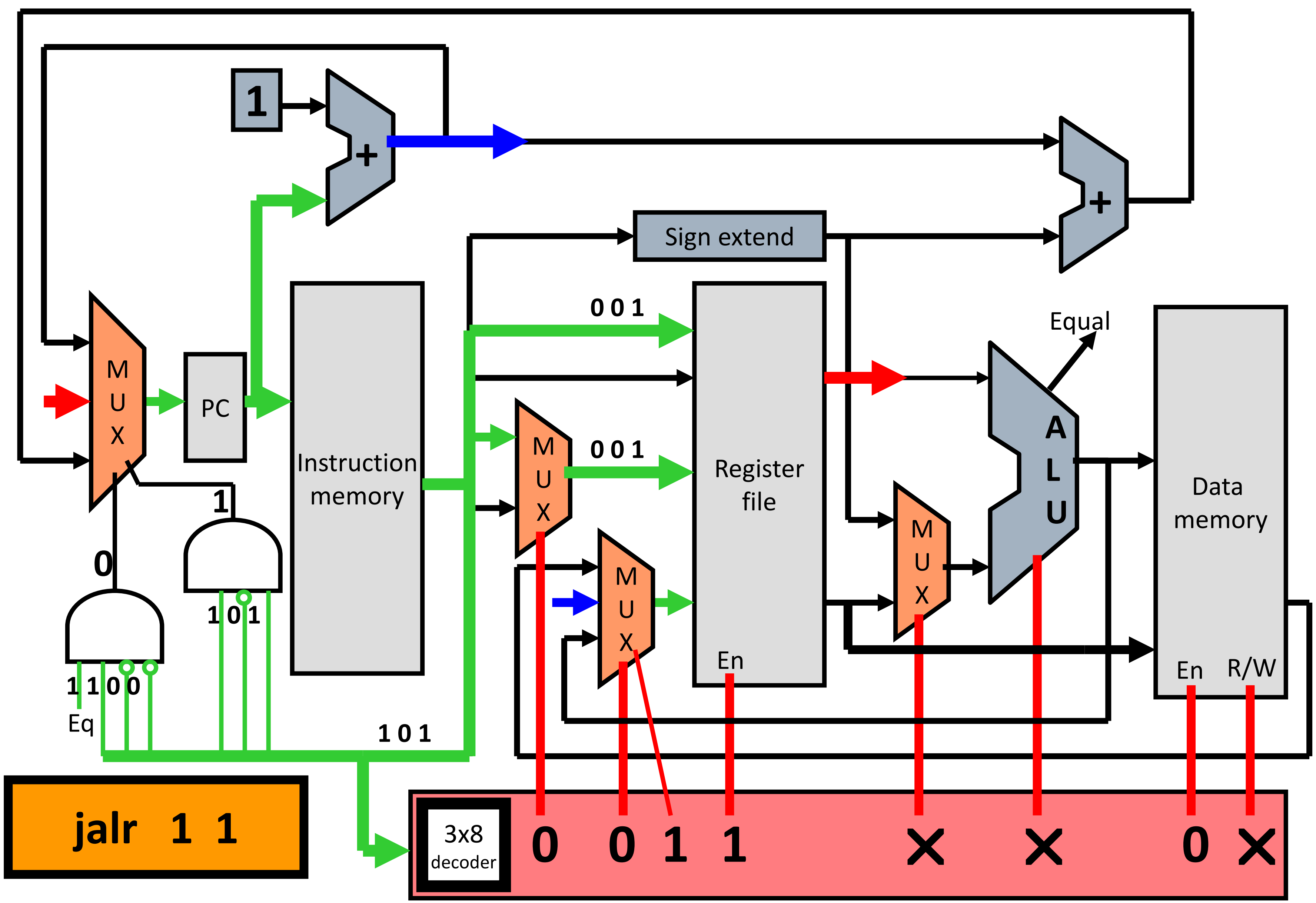
PC = regA

# Executing a JALR Instruction

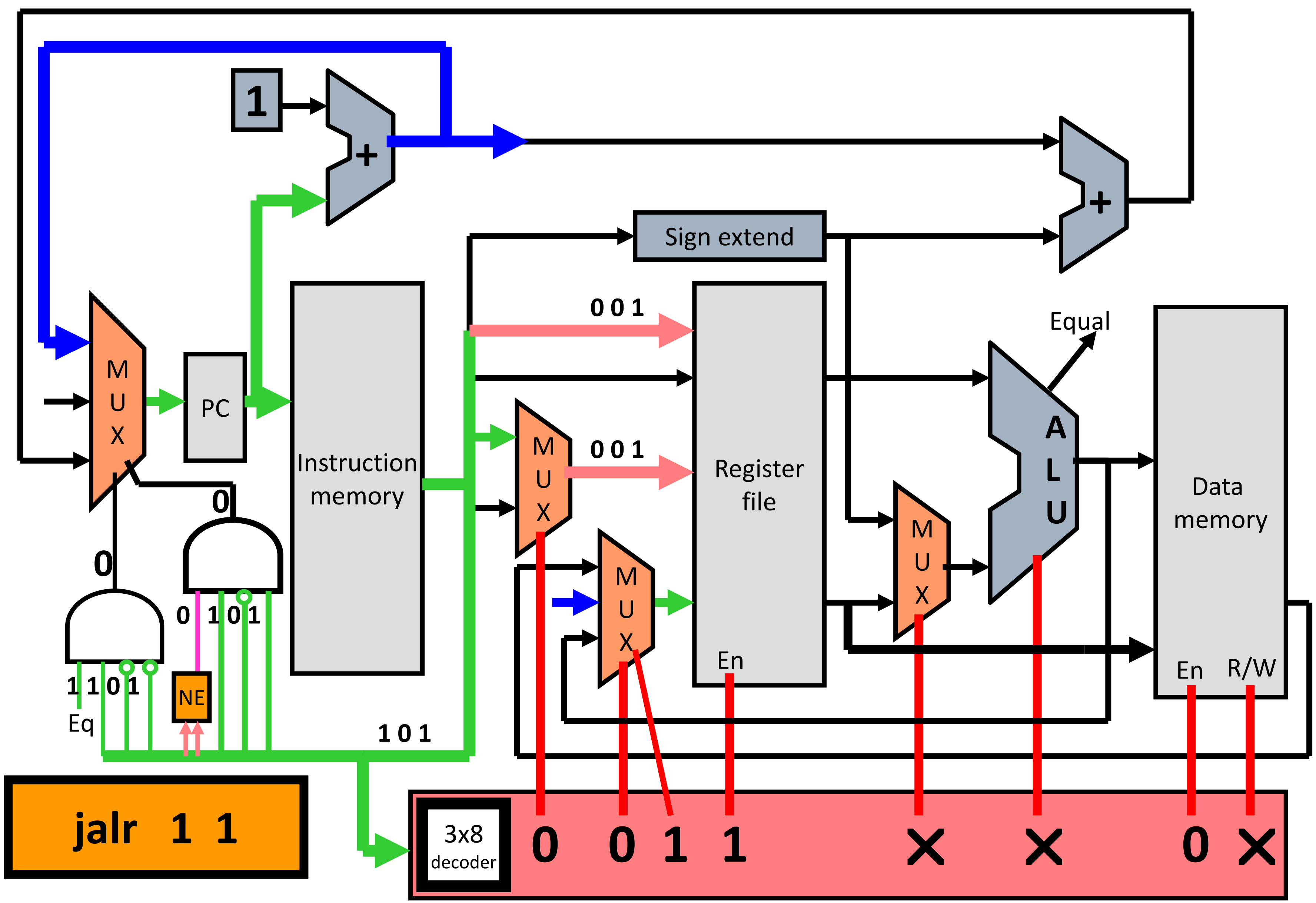



jalr 1 3

# 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:

??

**Poll:** 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

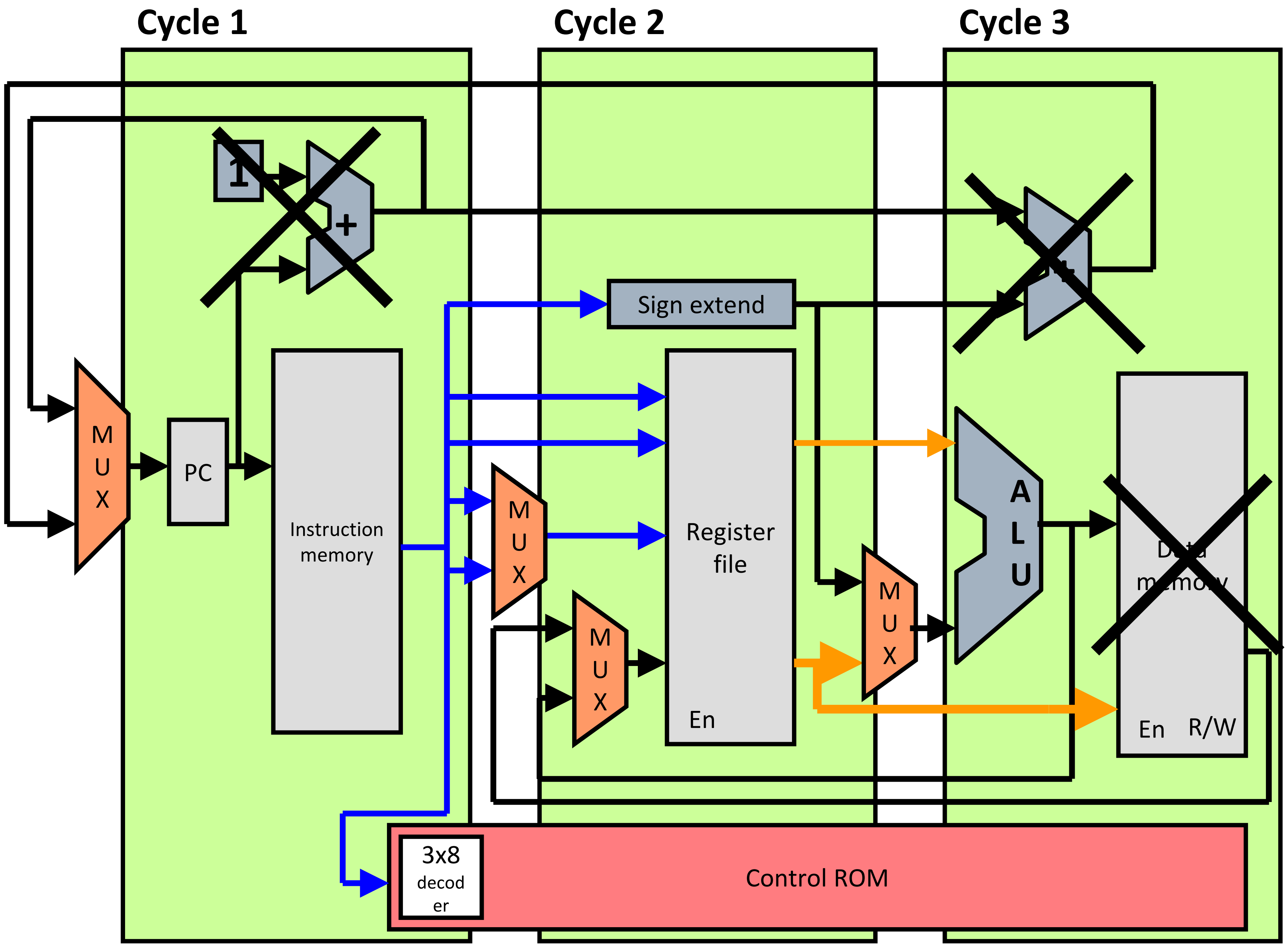
Optimal execution:

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

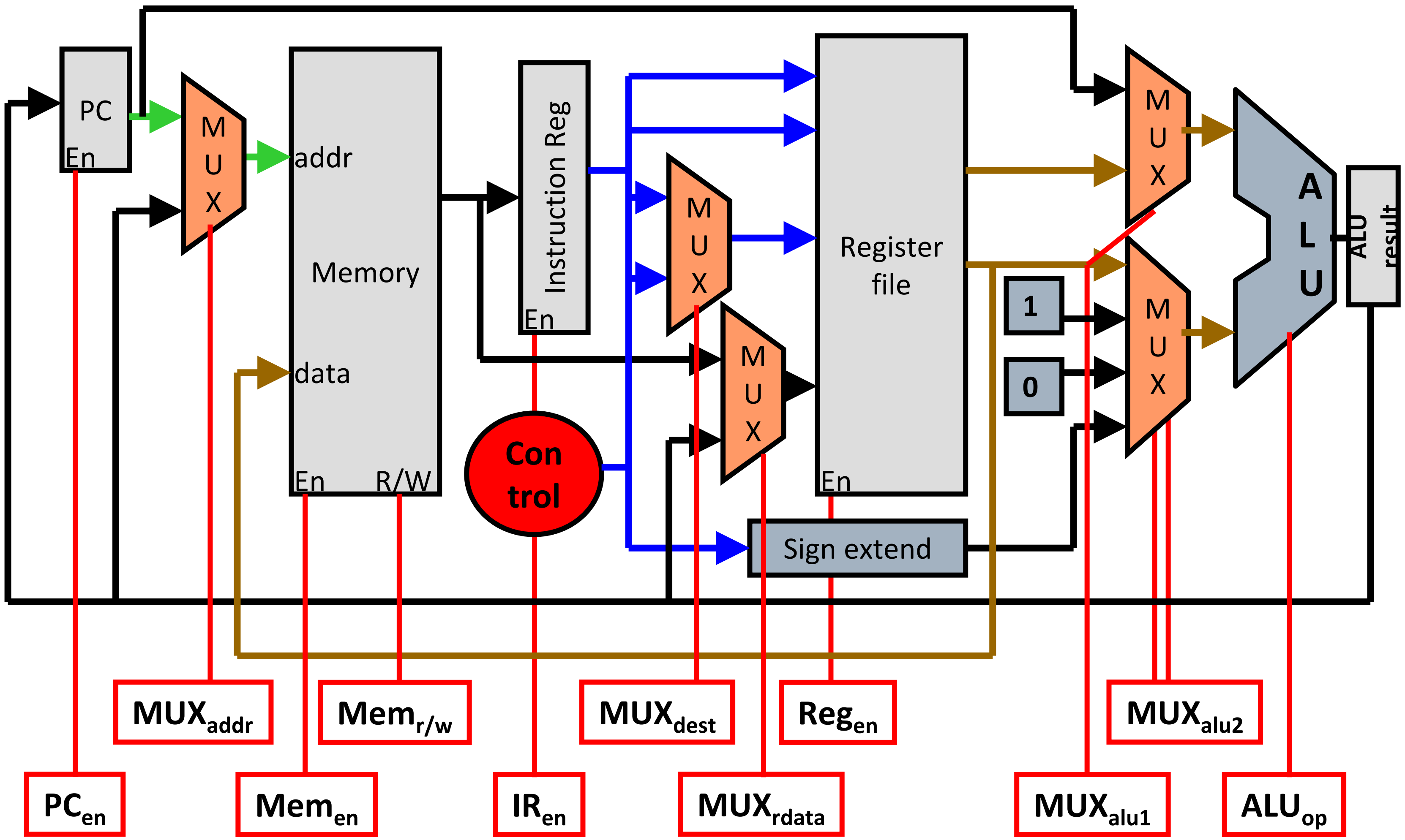
# Multiple-Cycle Execution

- 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

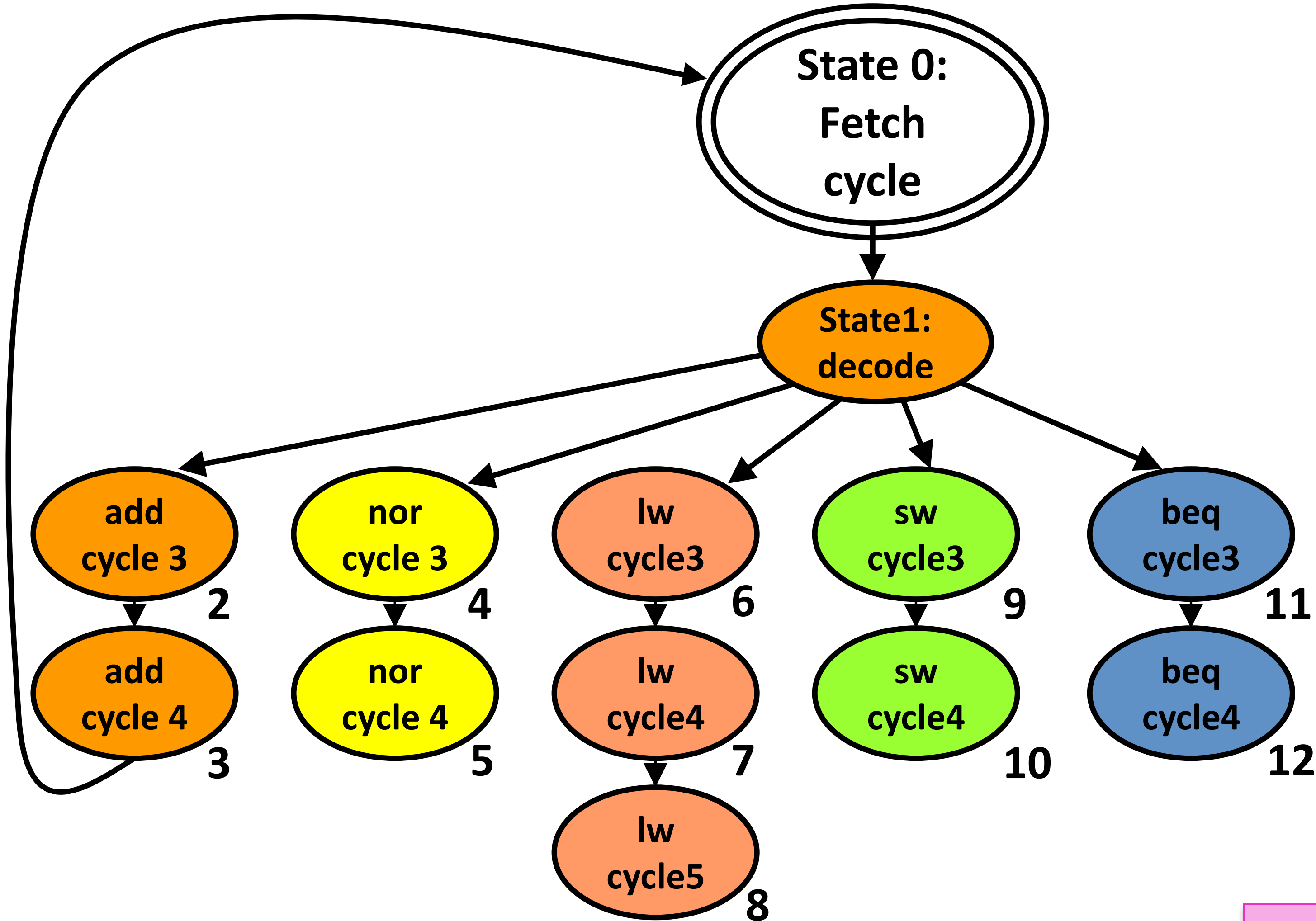


# Multi-cycle LC2 Datapath




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

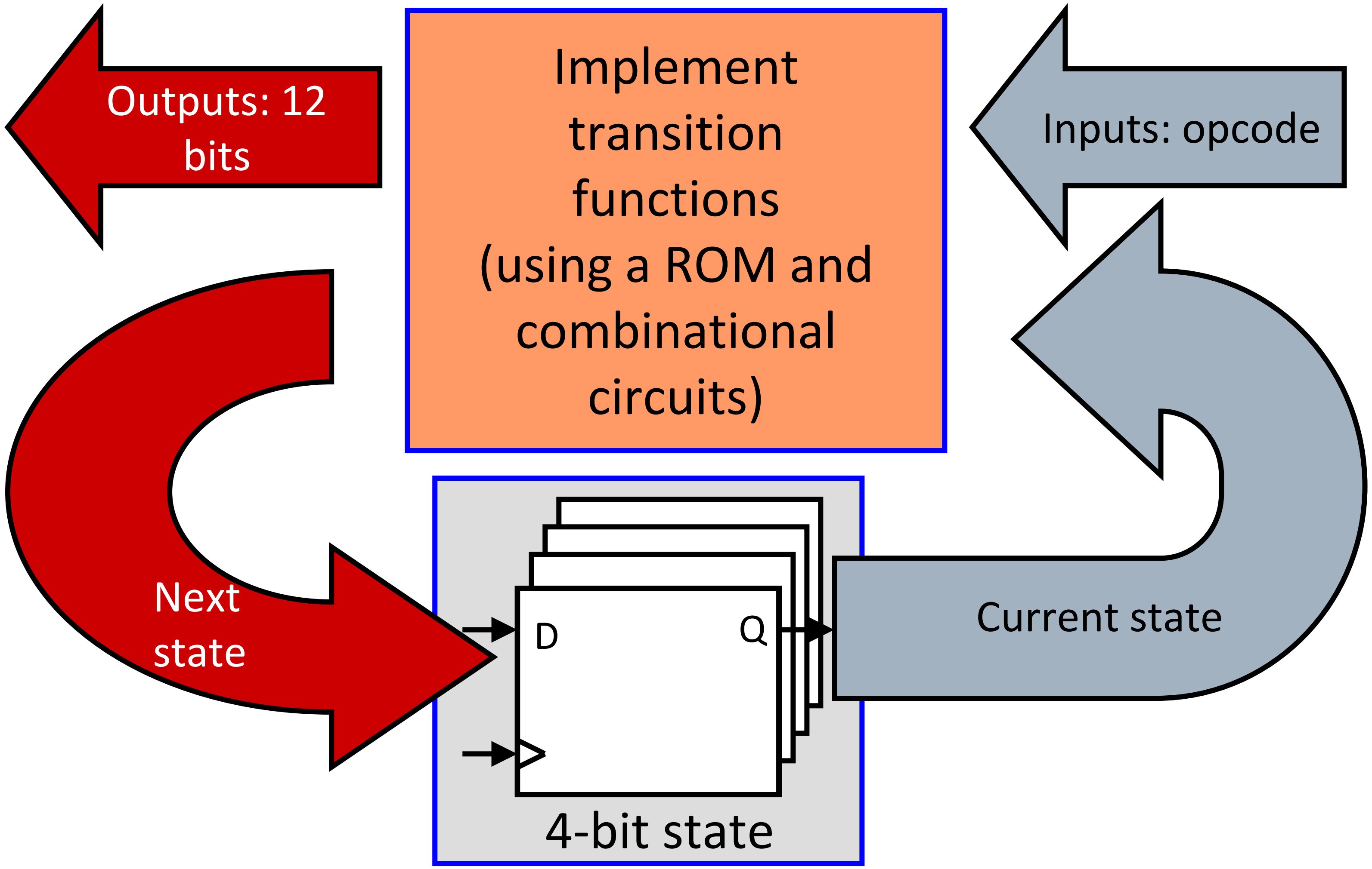
# State machine for multi-cycle control signals (transition functions)



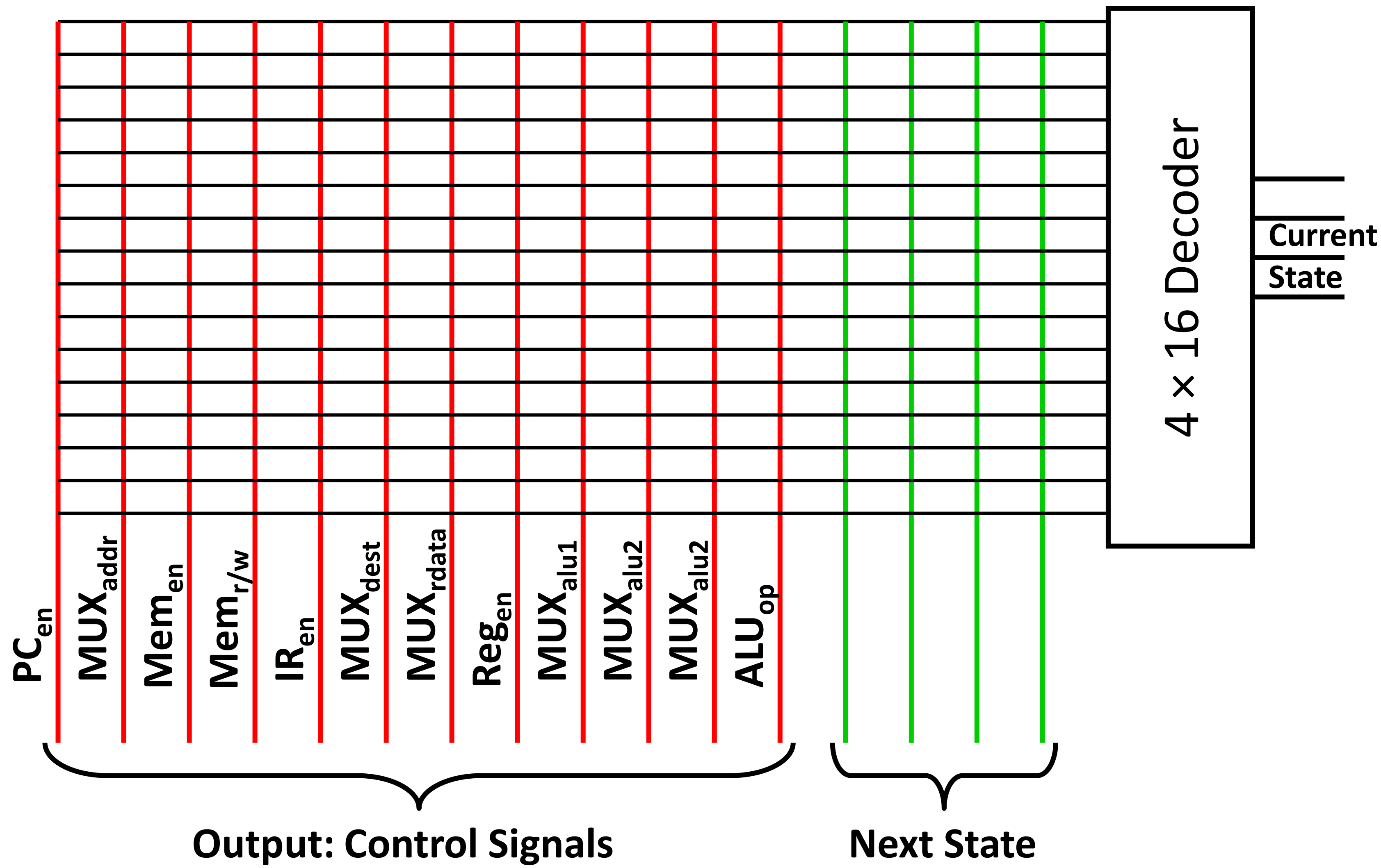

Note: we aren't worrying about JALR instruction in hardware going forward

__Poll:__ How many bits of storage are needed to store the state?

# Implementing FSM



# Building the Control ROM

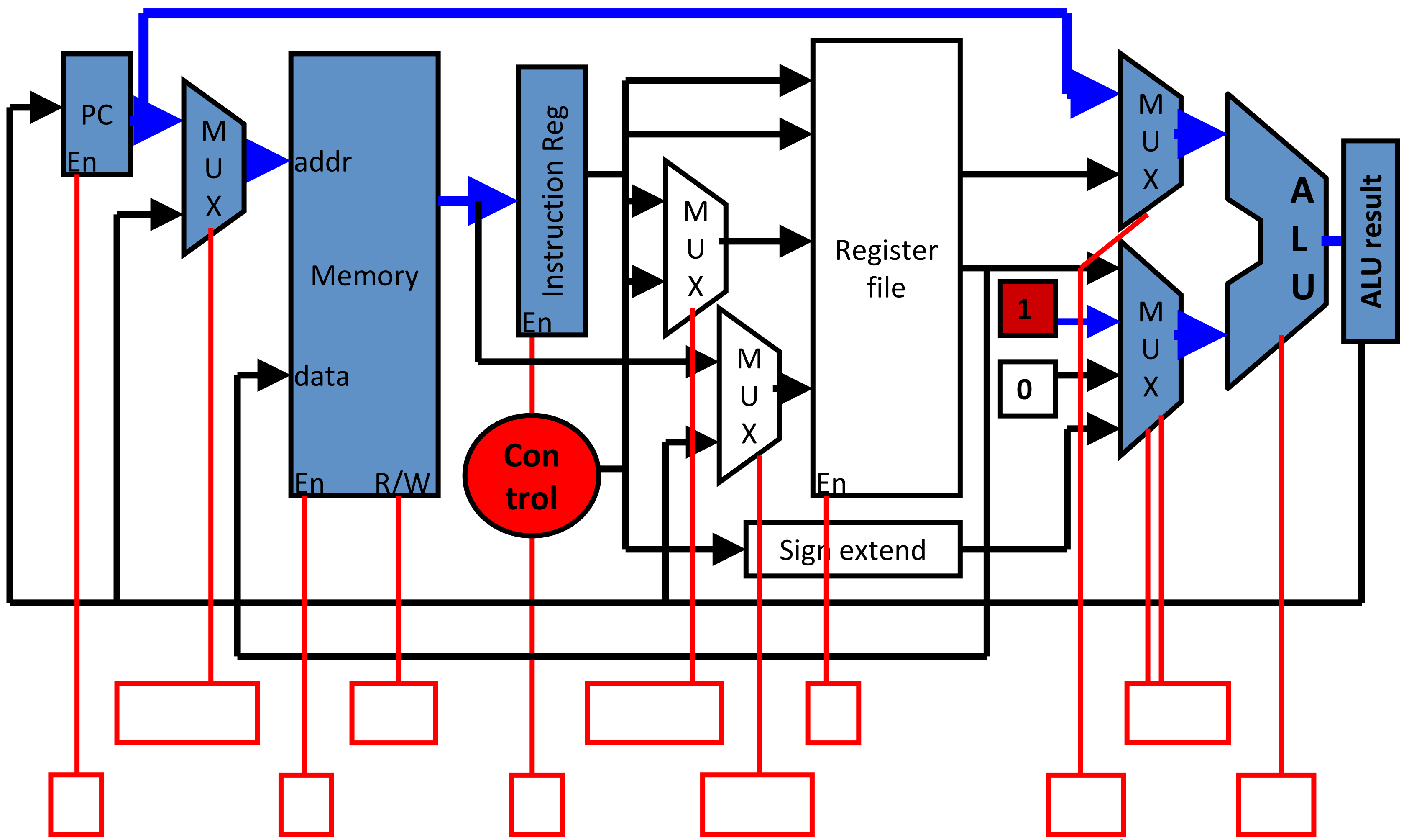


# First Cycle (State 0) Fetch Instr

- 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

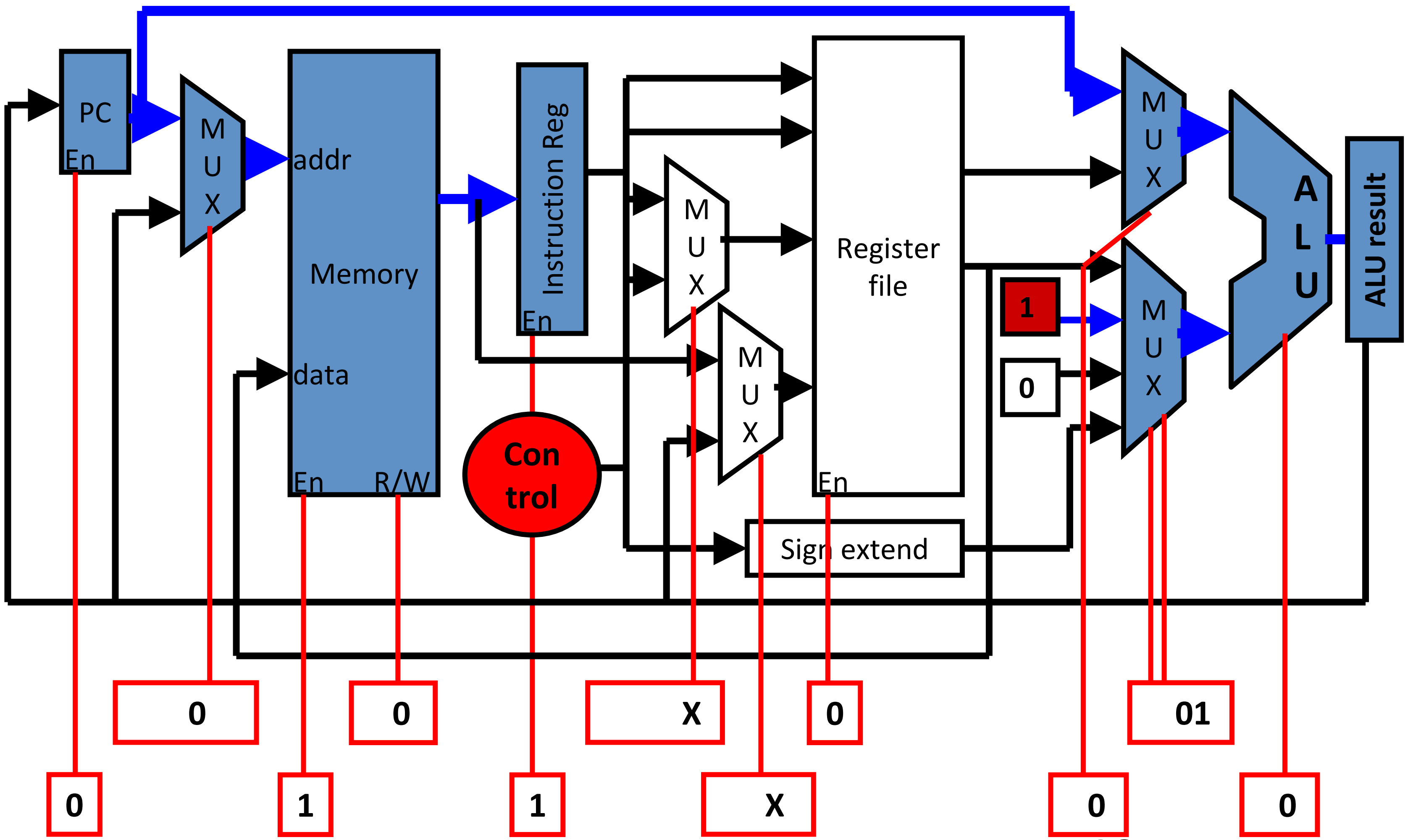
# First Cycle (State 0) Fetch Instr

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

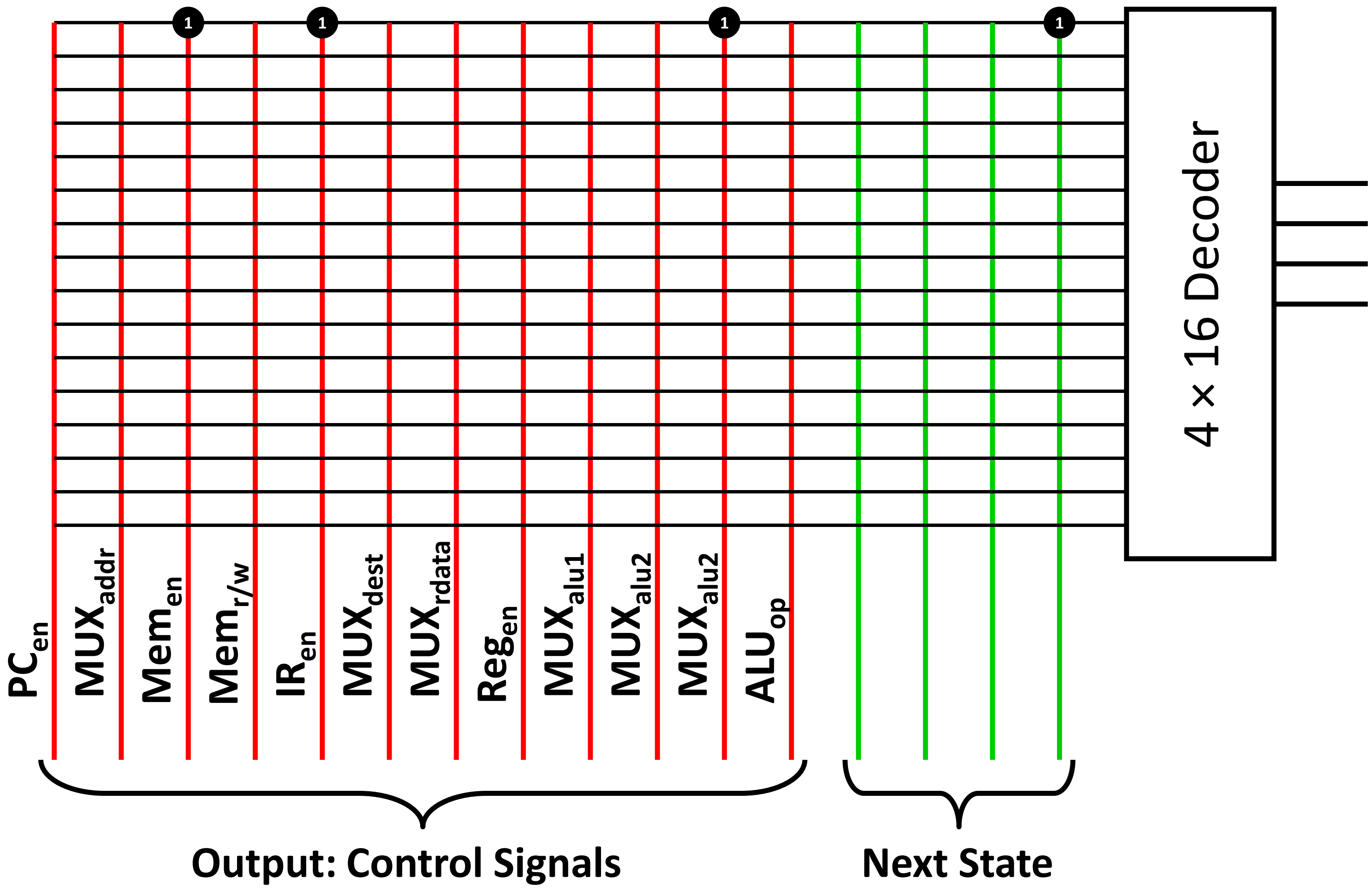


# First Cycle (State 0) Fetch Instr

**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