

10. Foundations of Processor Design - Single Cycle Data Path

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

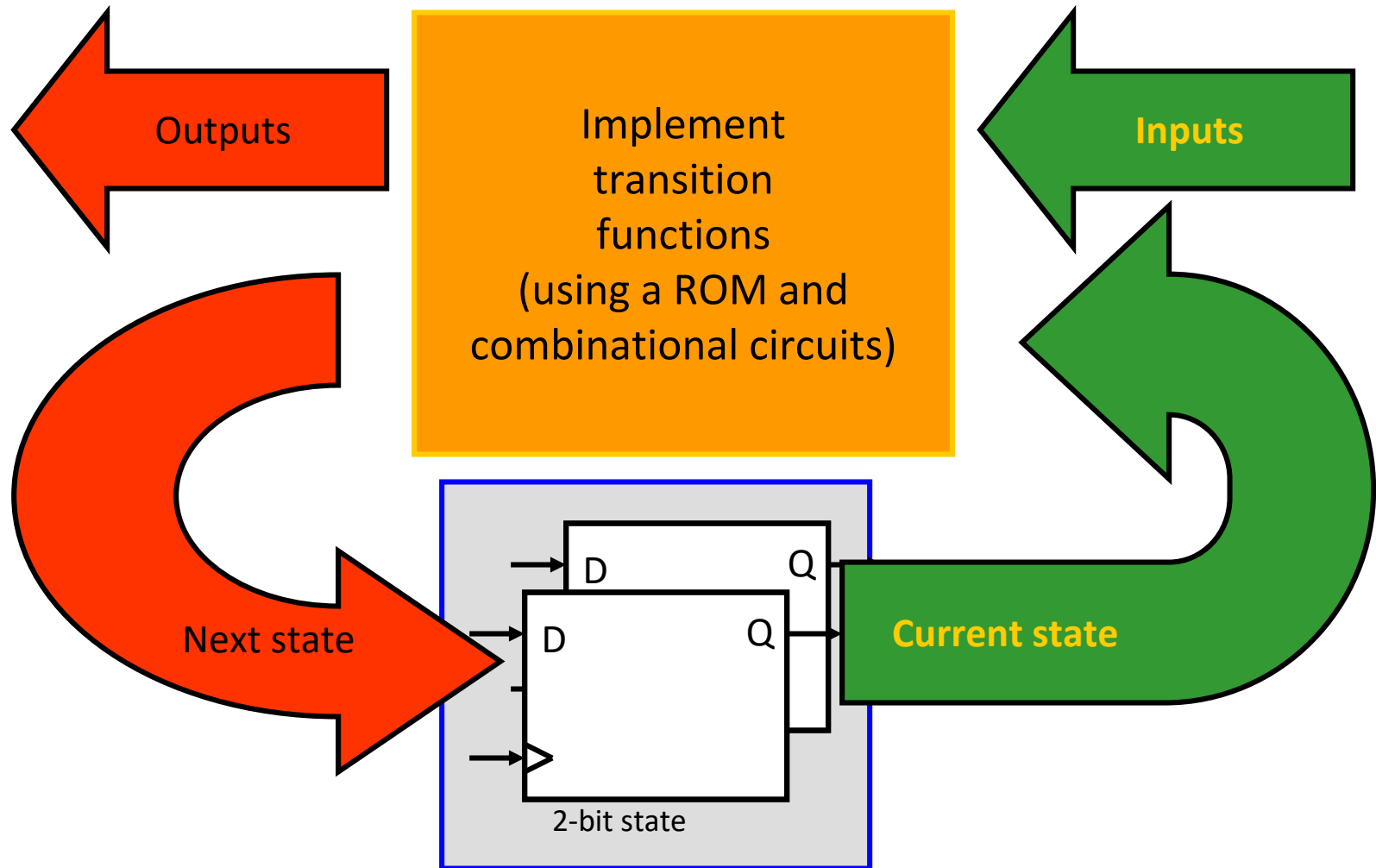
- If you didn't get full points on P1a, you'll need help.
See Piazza post @1247

❑ P2I due Thursday 2/23

- This is the harder one!

❑ HW3 due 2/20

Implementing a FSM



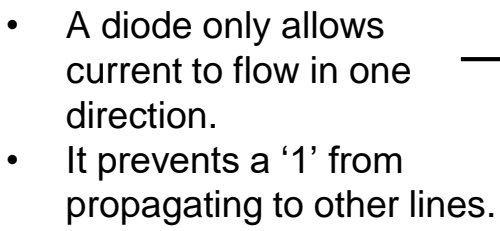
ROMs and PROMs

- ❑ Read Only Memory
 - Array of memory values that are constant
 - Non-volatile

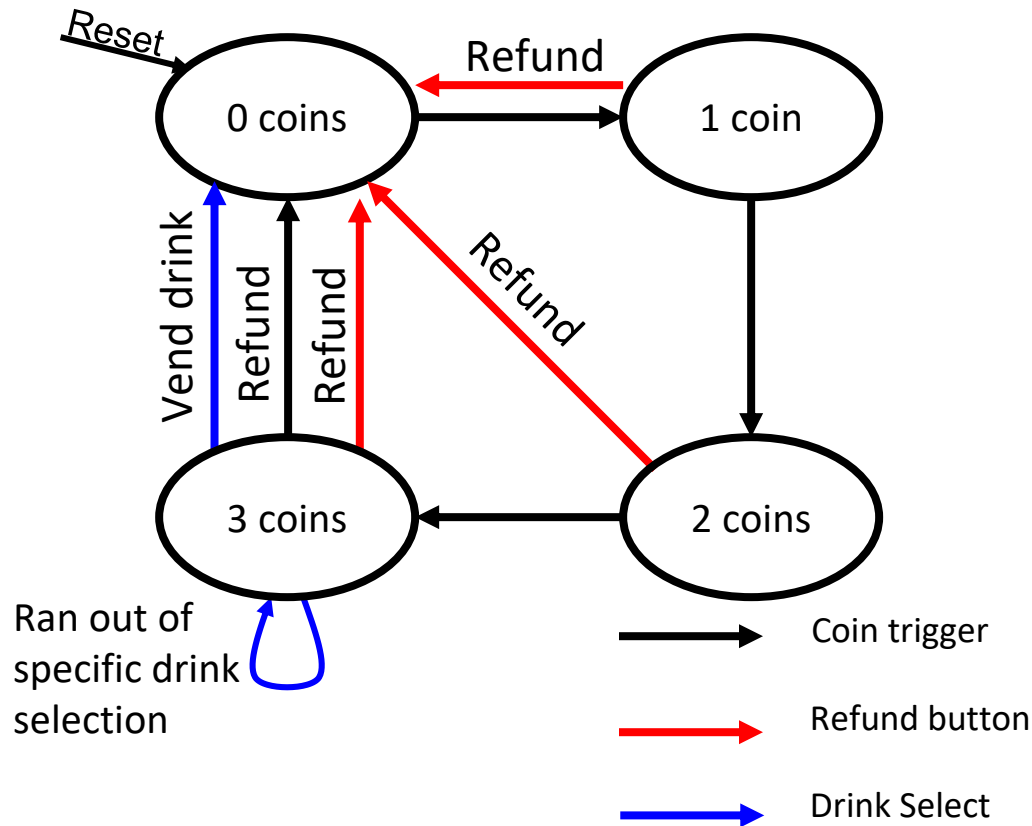
- ❑ Programmable Read Only Memory
 - Array of memory values that can be written exactly once (destructive writes)

- ❑ You can use ROMs to implement FSM transition functions
 - ROM inputs (i.e., ROM address): current state, primary inputs
 - ROM outputs (i.e., ROM data): next state, primary outputs

Review



FSM for Vending Machine

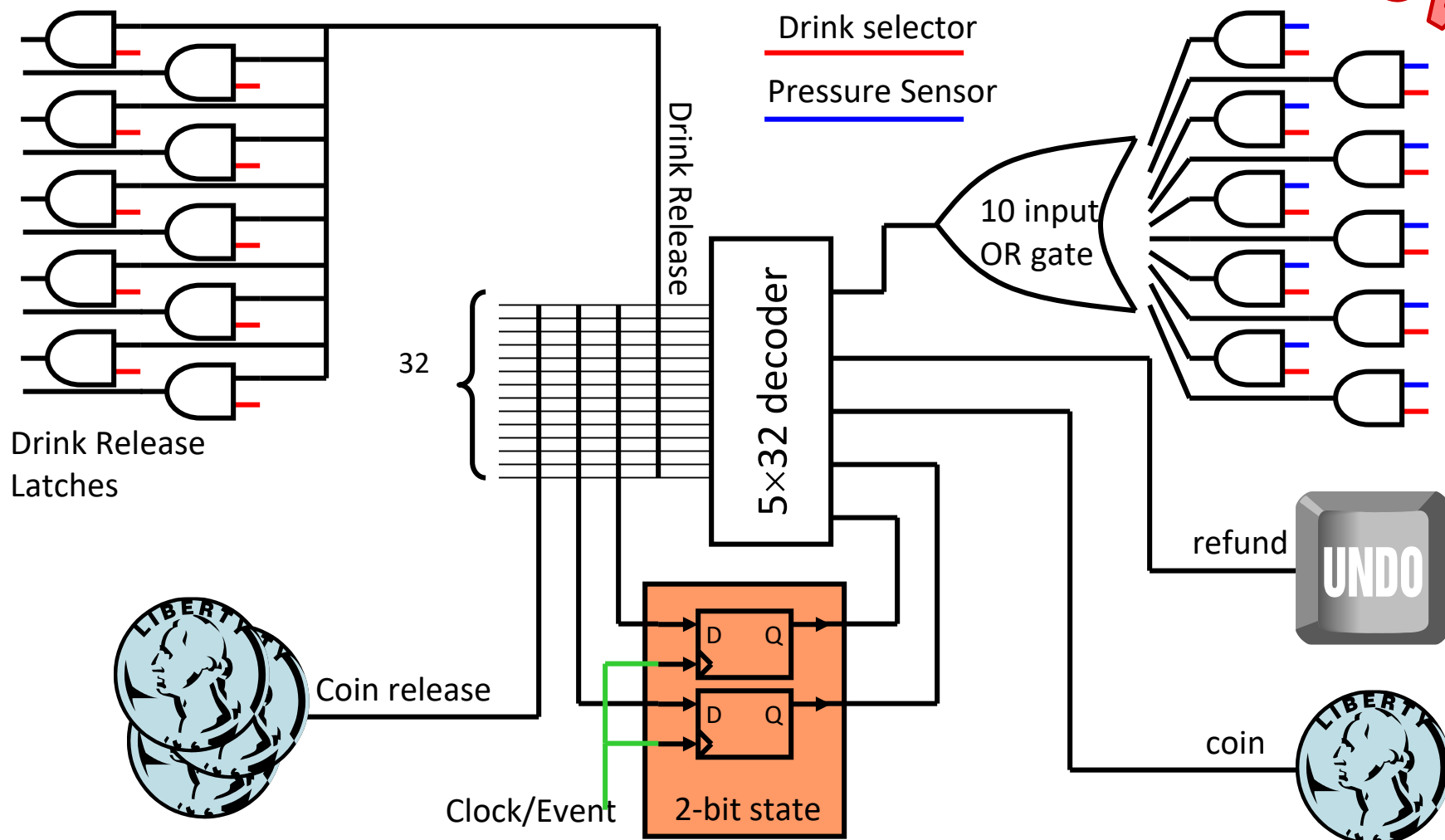


Is this a Mealy or Moore Machine?

Mealy ~ output is based on current state *AND* input

Review

Putting it all together

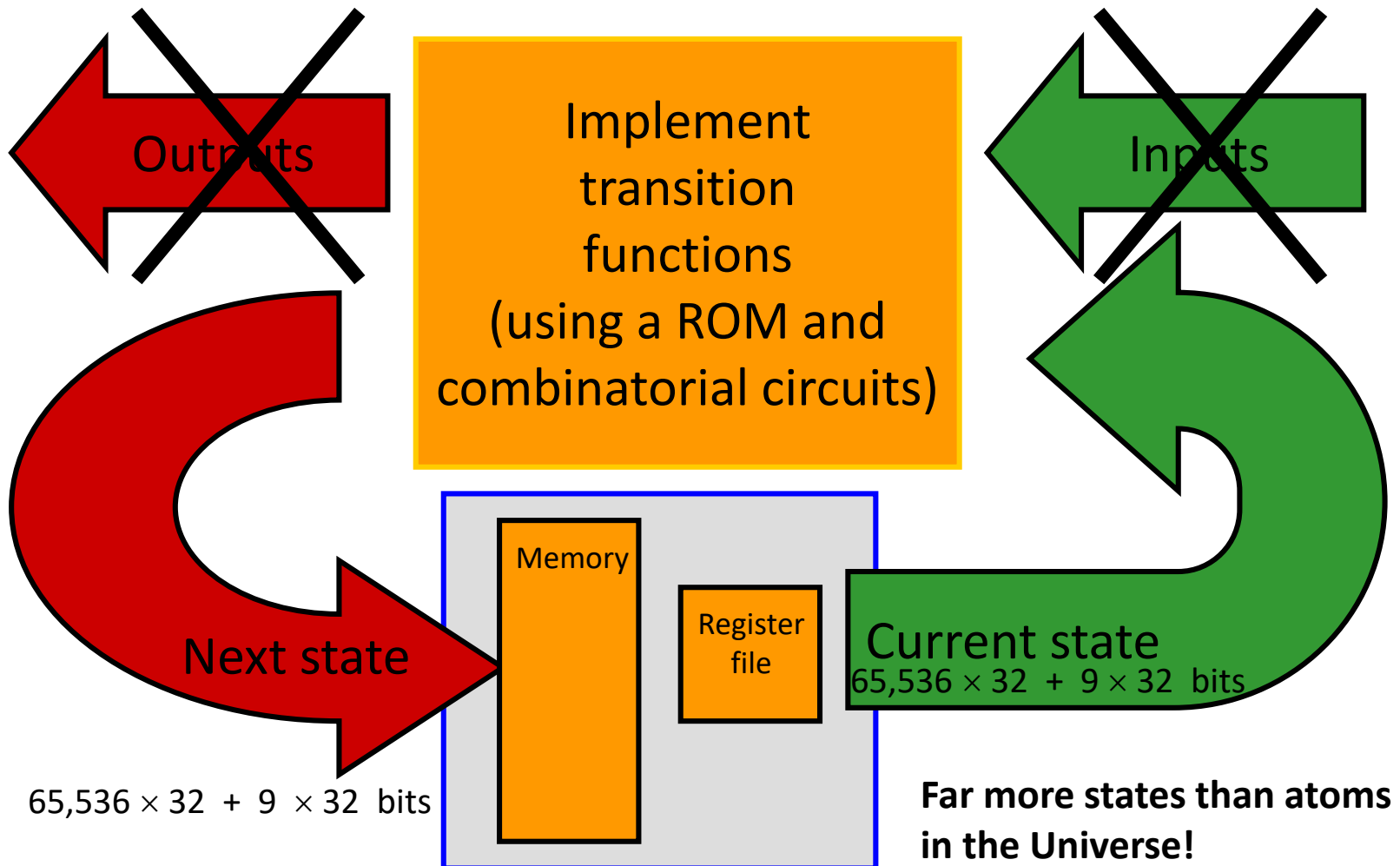


Limitations of the controller

- ❑ What happens if we make the price \$1.00?, or what if we want to accept nickels, dimes and quarters?
 - Must redesign the controller (more state, different transitions)
 - A programmable processor only needs a software upgrade.
 - If you had written really good software anticipating a variable price, perhaps no change is even needed

- ❑ **Next Topic - Our first processor!**

LC2Kx Processor as FSM



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

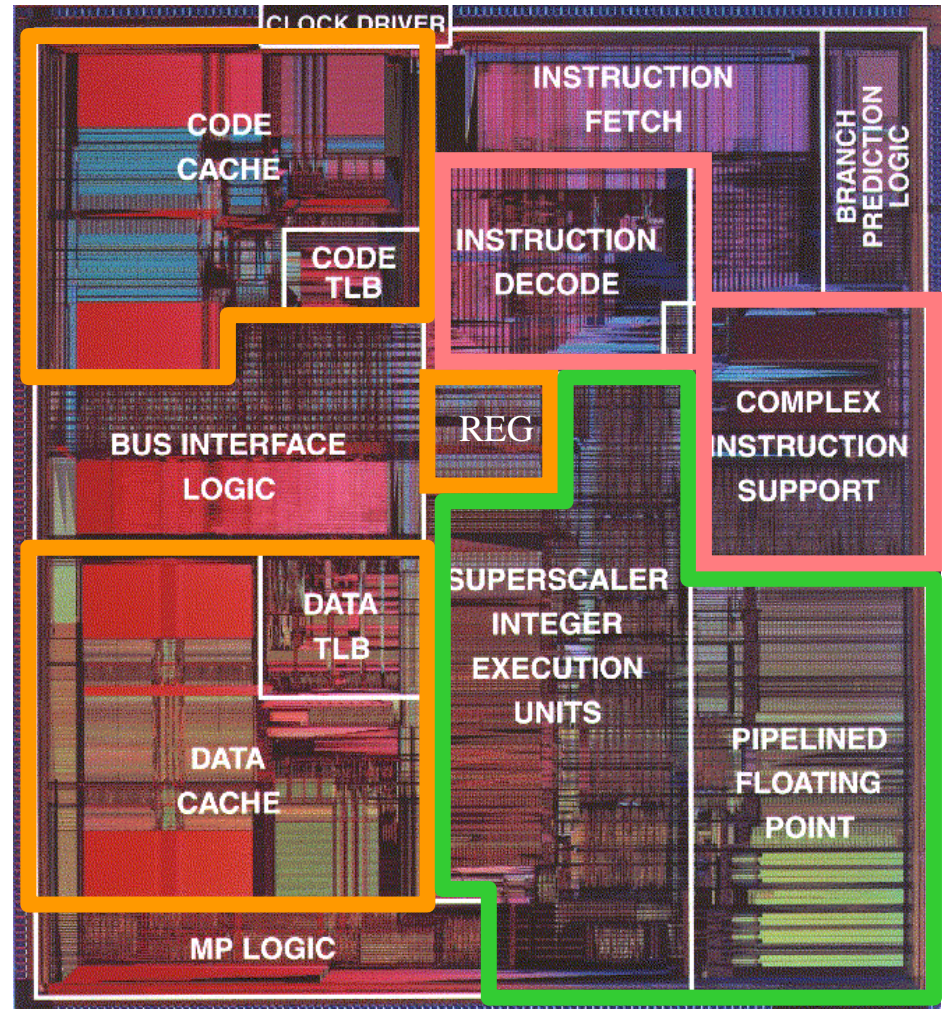
Pentium Processor Die

State

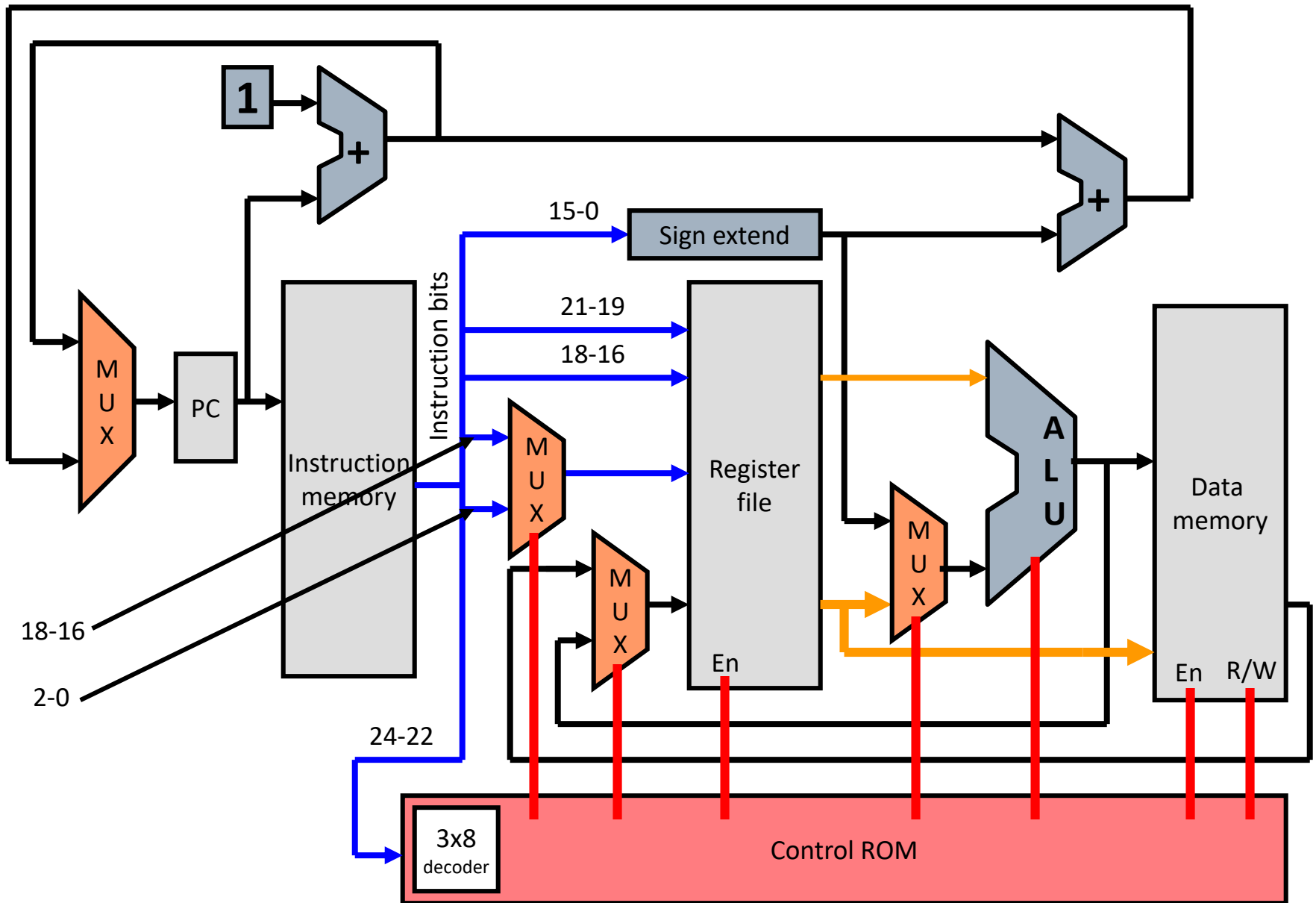
- Registers
- Memory

Control ROM

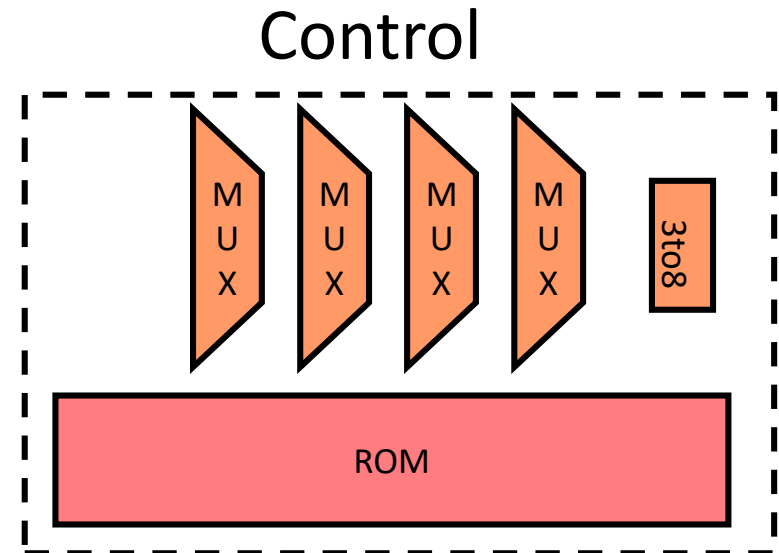
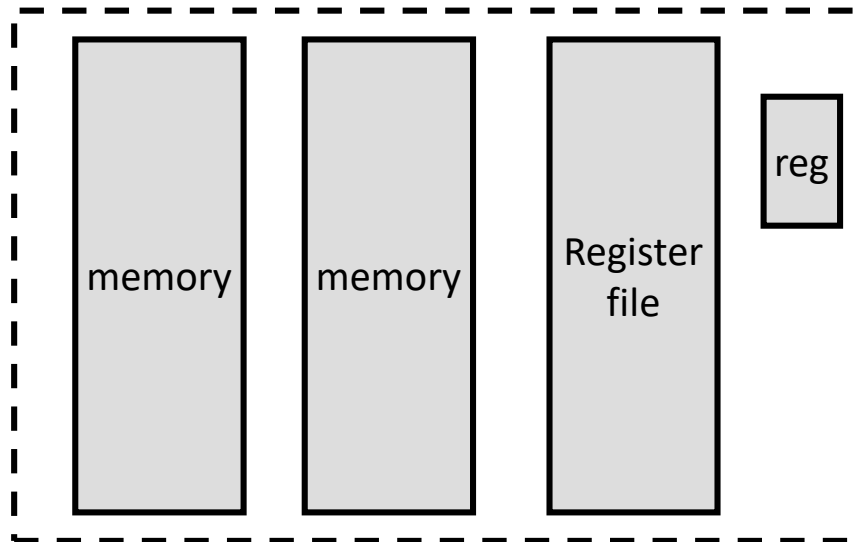
Combinational logic (Compute)



LC2Kx Datapath Implementation

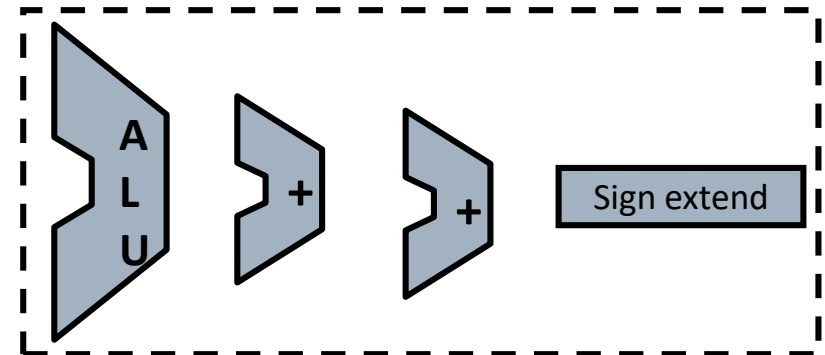


Building Blocks for the LC2



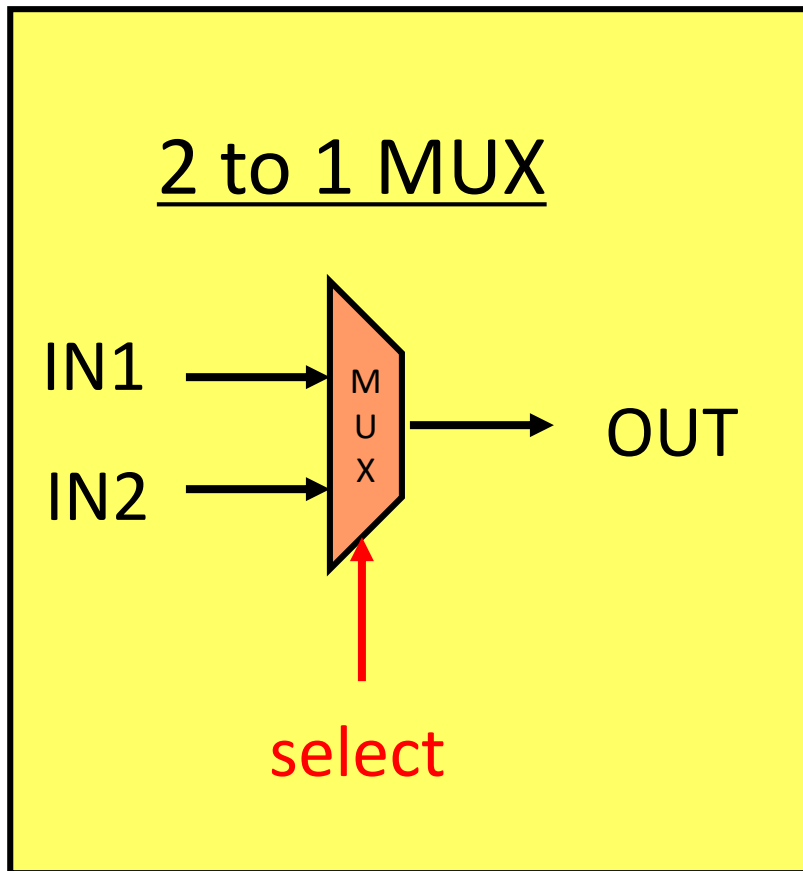
State

Compute



Here are the pieces, go build yourself a processor!

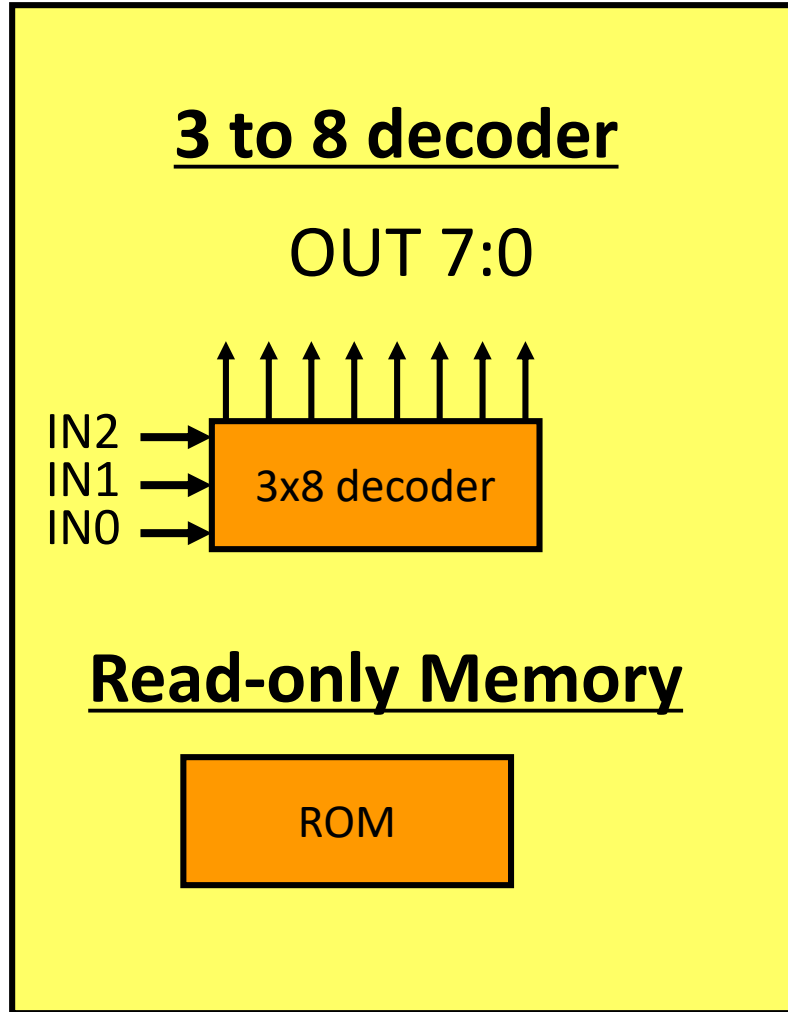
Control Building Blocks (1)



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

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

Control Building Blocks (2)



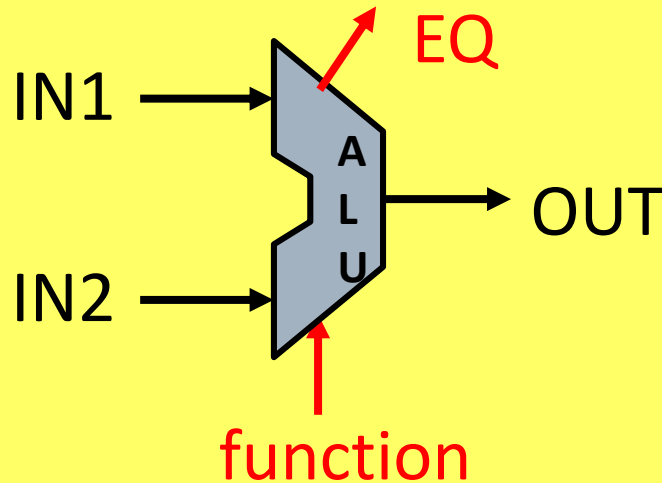
Decoder activates one of the output lines based on the input

| IN | OUT |
|------------|-----------------|
| <u>210</u> | <u>76543210</u> |
| 000 | 00000001 |
| 001 | 00000010 |
| 010 | 00000100 |
| 011 | 00001000 |
| etc. | |

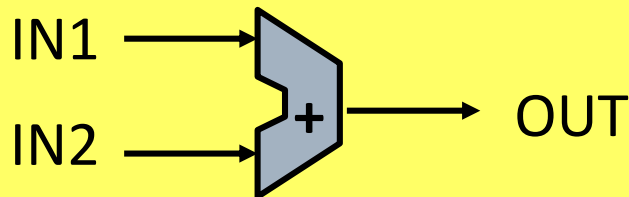
ROM stores preset data in each location.
Give address to access data.

Compute Building Blocks (1)

Arithmetic Logic Unit



Adder



Perform basic arithmetic functions

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

$$EQ = (IN1 == IN2)$$

For LC2K:

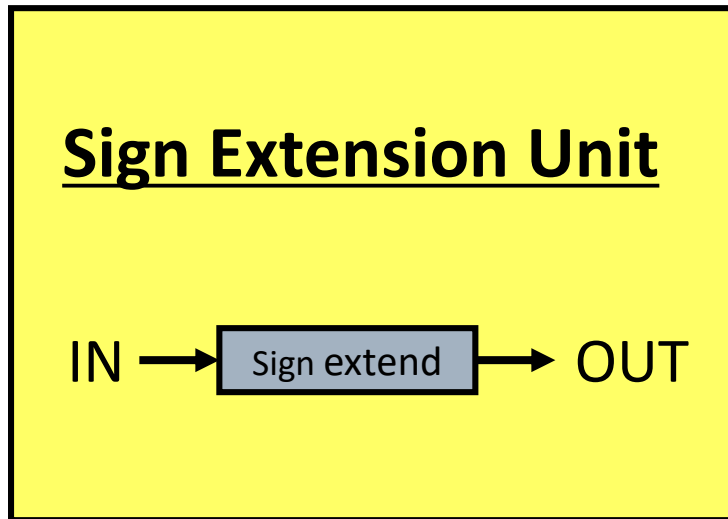
f=0 is add

f=1 is nor

For other processors, there are many more functions.

Just adds

Compute Building Blocks (2)



Sign extend input by replicating the MSB to width of output

$$\text{OUT}(31:0) = \text{SE}(\text{IN}(15:0))$$

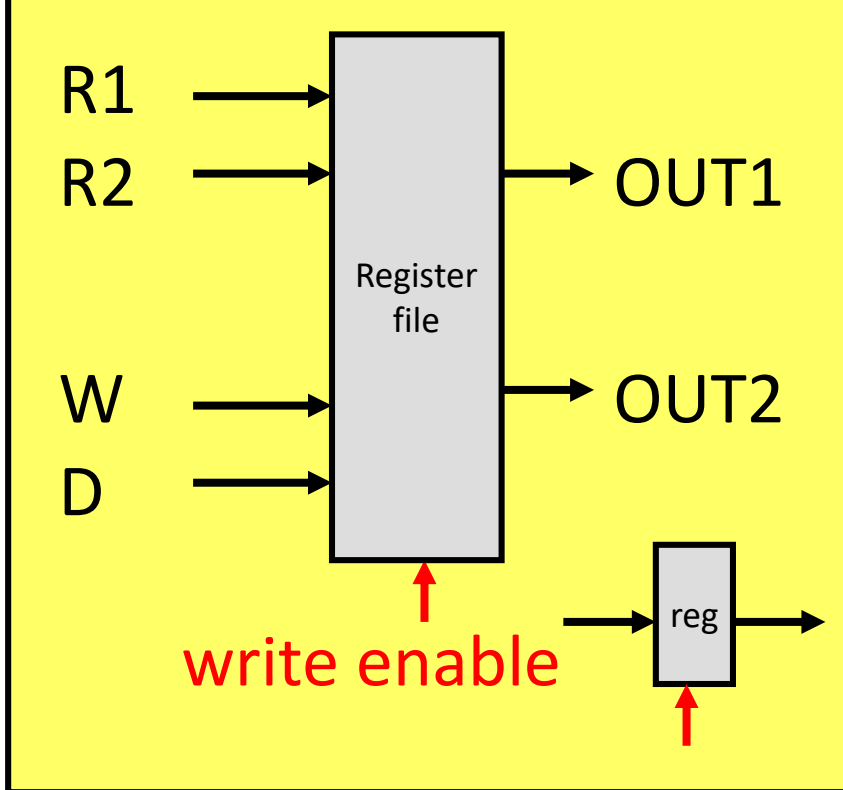
$$\text{OUT}(31:16) = \text{IN}(15)$$

$$\text{OUT}(15:0) = \text{IN}(15:0)$$

Useful when compute unit is wider than data

State Building Blocks (1)

Register File or Register



Small/fast memory to store temporary values

n entries (LC2 = 8)

r read ports (LC2 = 2)

w write ports (LC2 = 1)

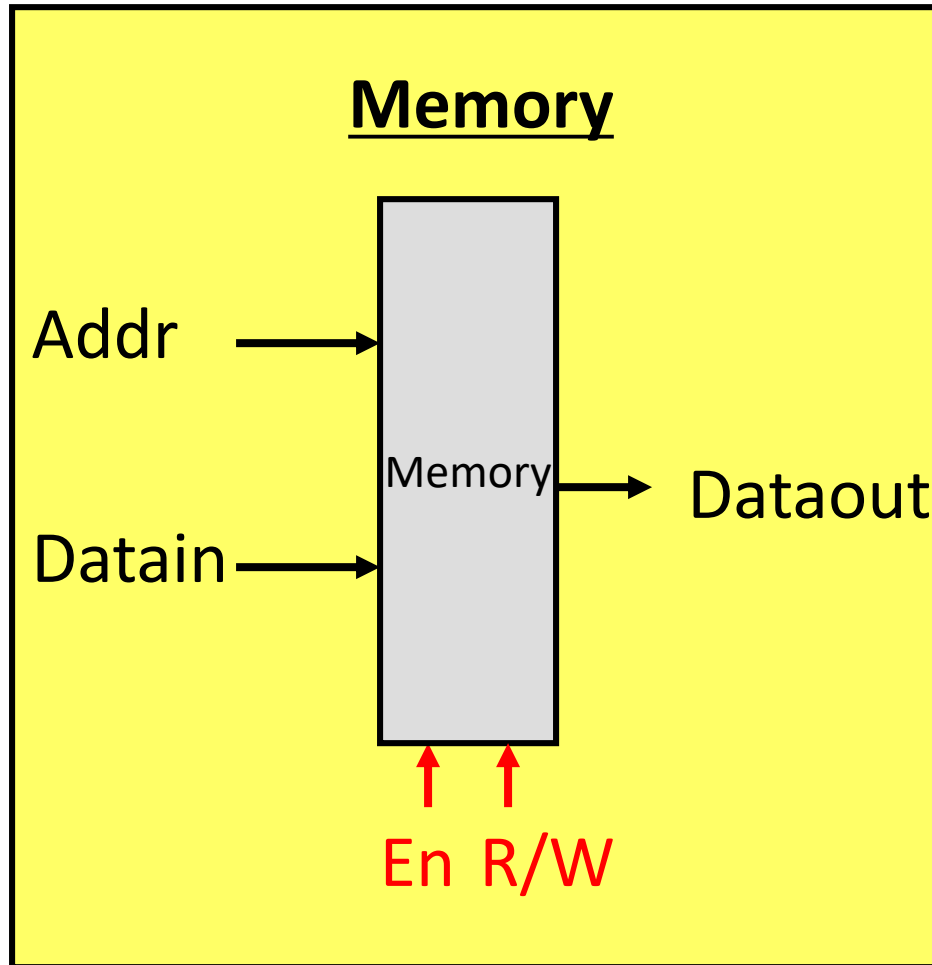
* R_i specifies register number to read

* W specifies register number to write

* D specifies data to write

How many bits are R_i and W_i in LC2?

State Building Blocks (2)



Slower storage structure to hold large amounts of stuff.

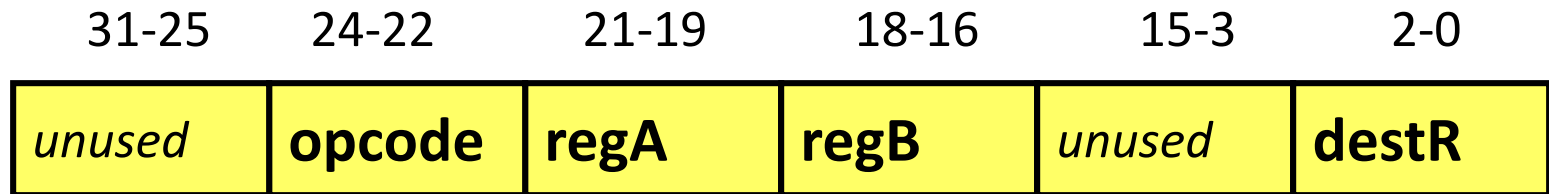
Use 2 memories for LC2

- * Instructions
- * Data
- * 65,536 total words

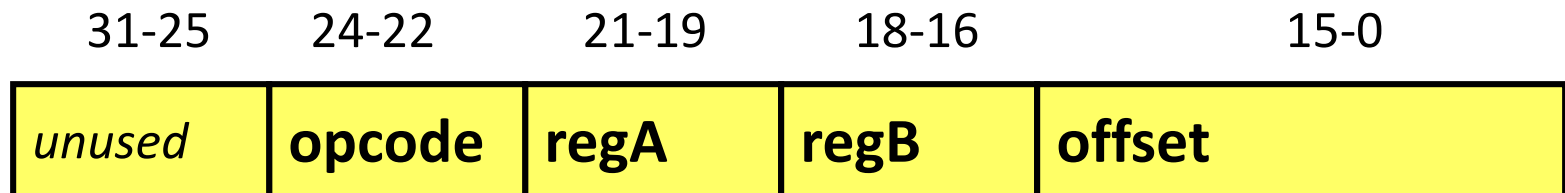
Recap: LC2K Instruction Formats

- ❑ Tells you which bit positions mean what

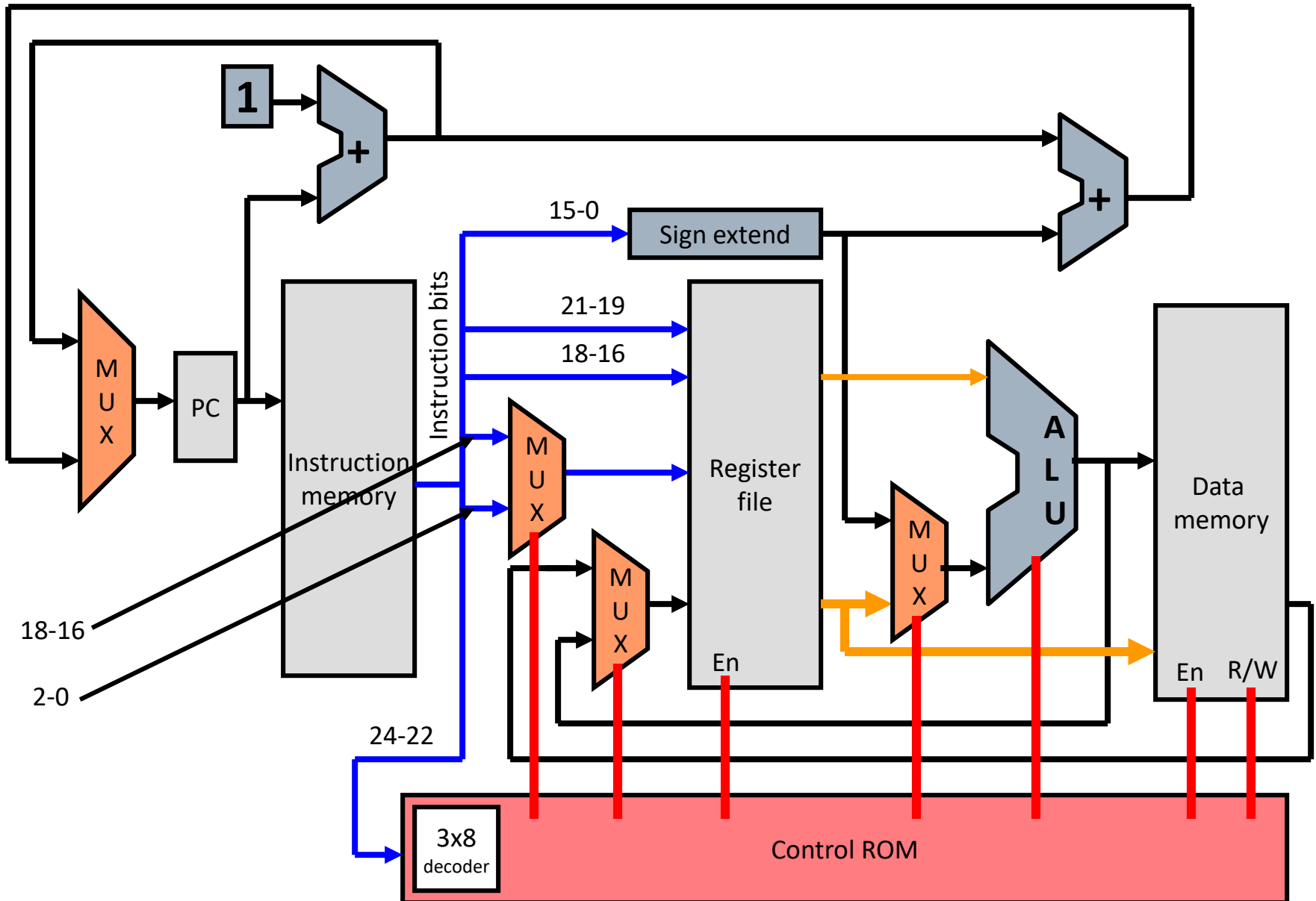
- ❑ R type instructions (add '000', nor '001')



- ❑ I type instructions (lw '010', sw '011', beq '100')



LC2Kx Datapath Implementation

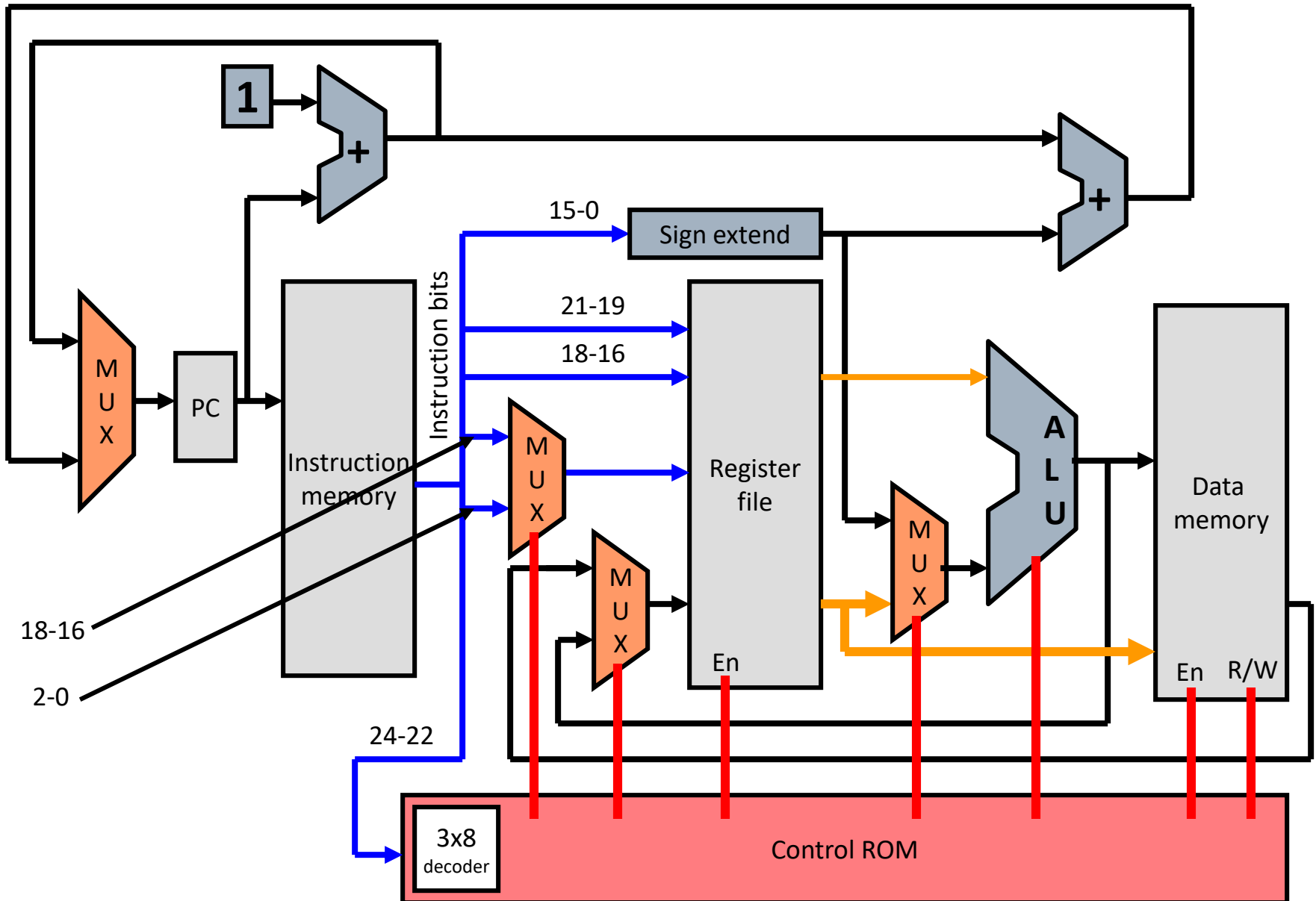


Single-Cycle Processor Design

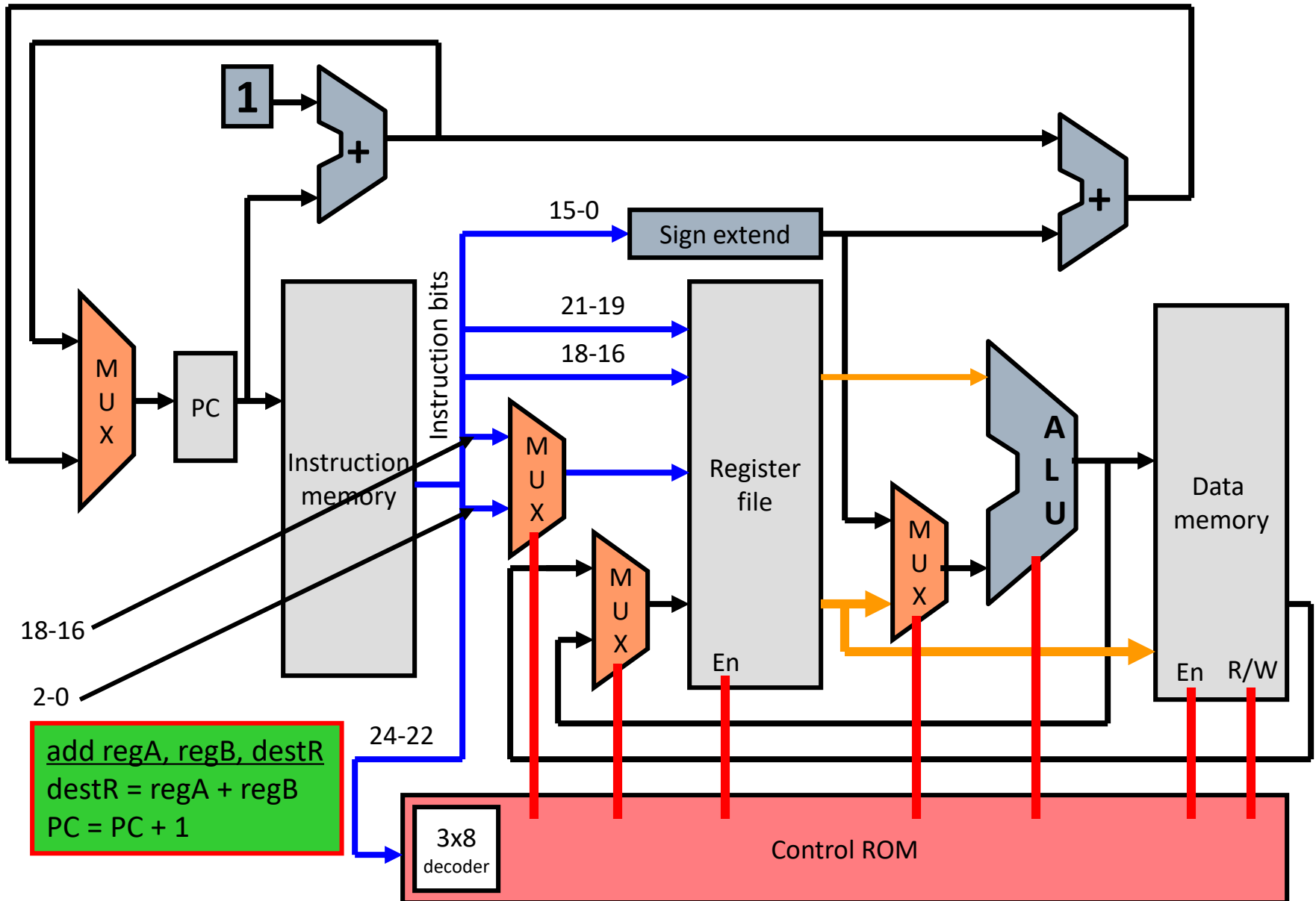
❑ General-Purpose Processor Design

- 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

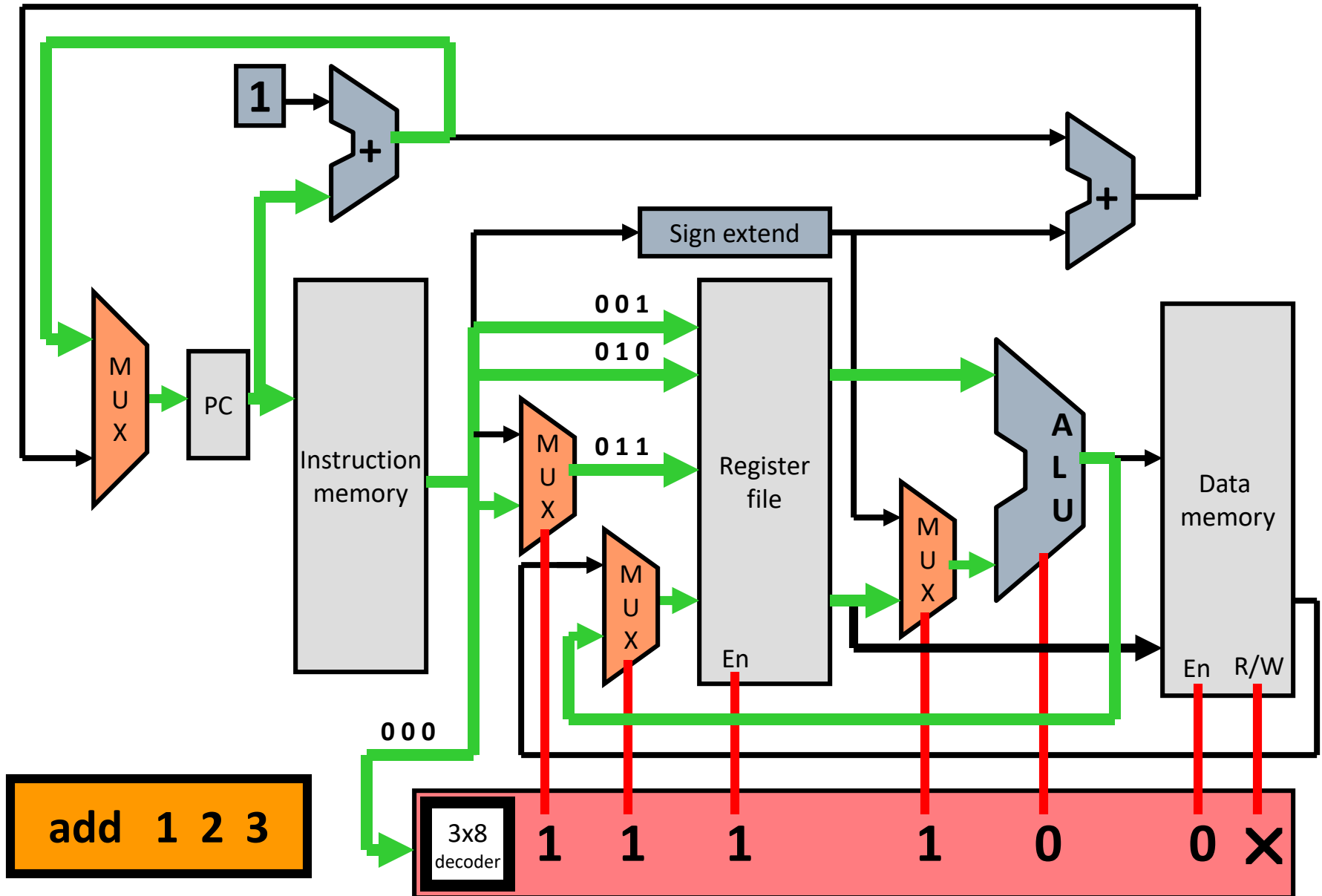
LC2Kx Datapath Implementation



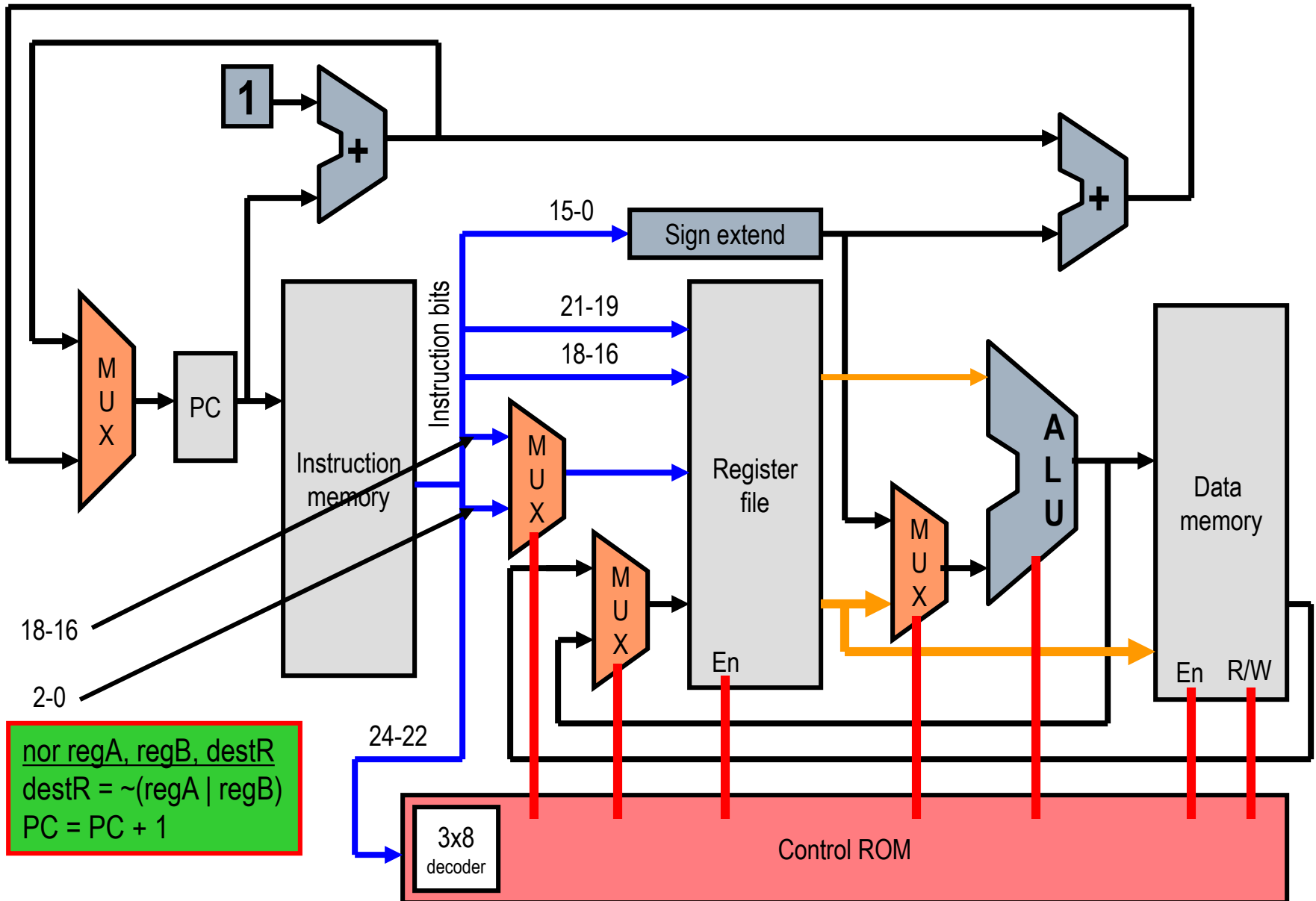
Executing an **ADD** Instruction



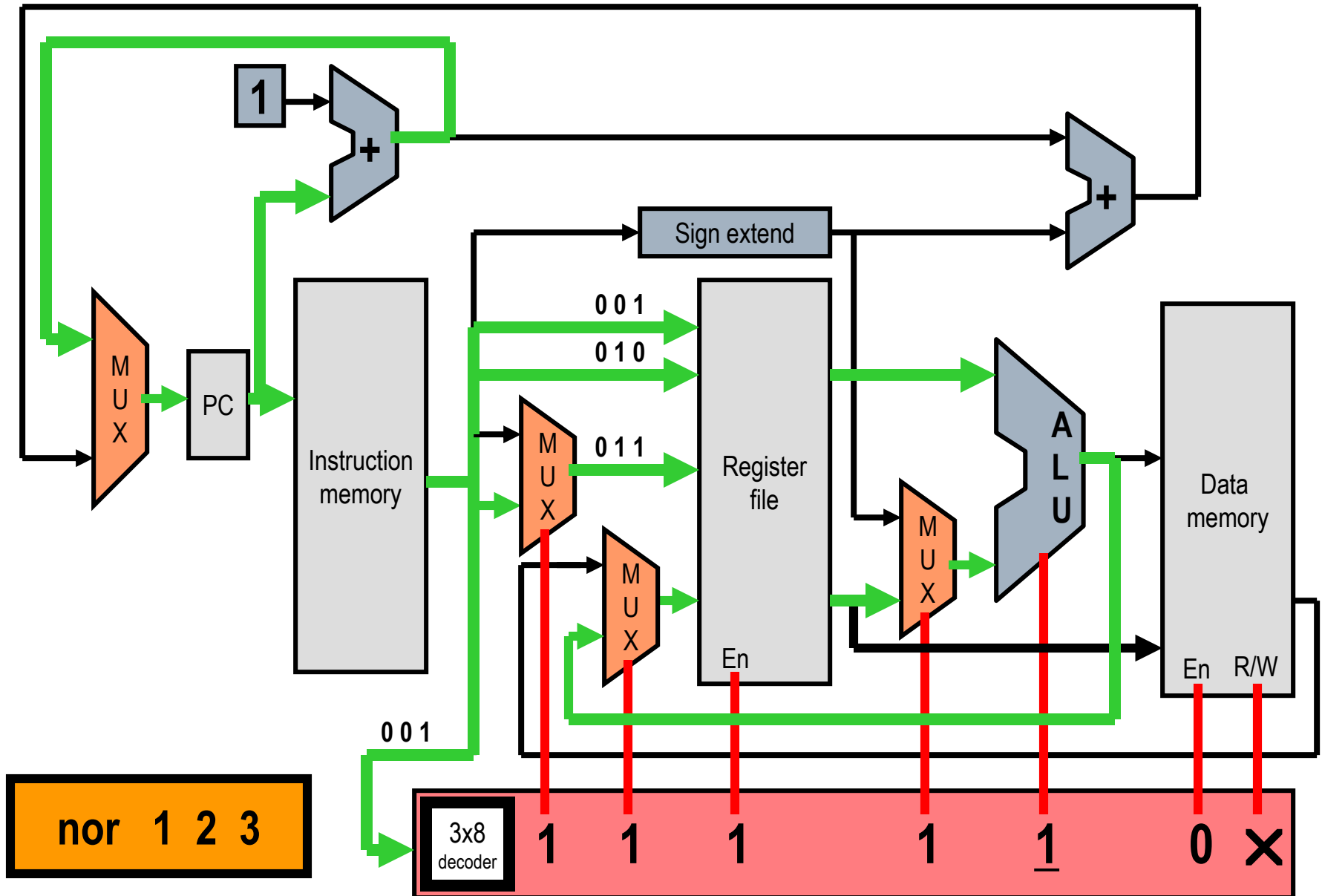
Executing an **ADD** Instruction on LC2Kx Datapath



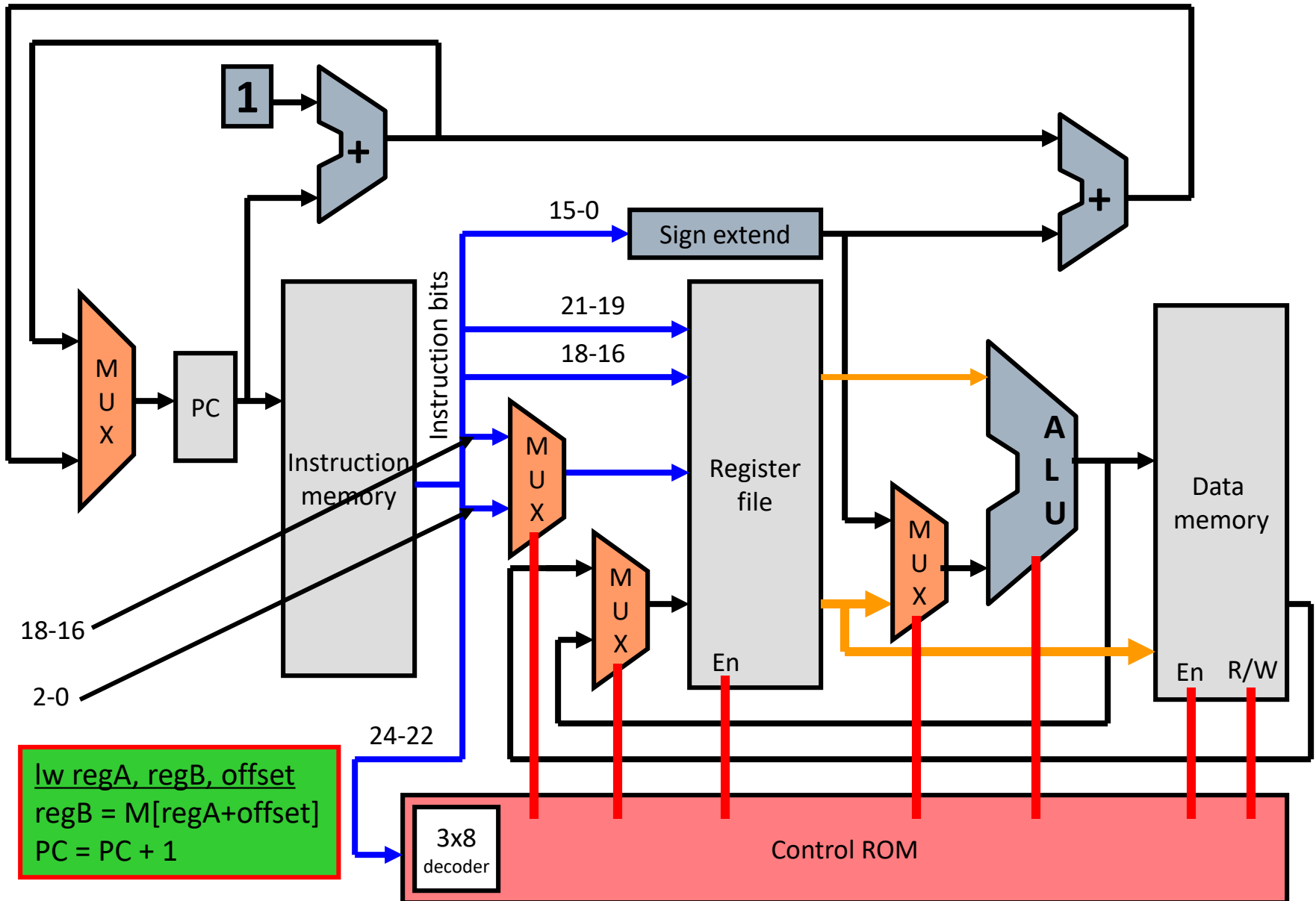
Executing a NOR Instruction



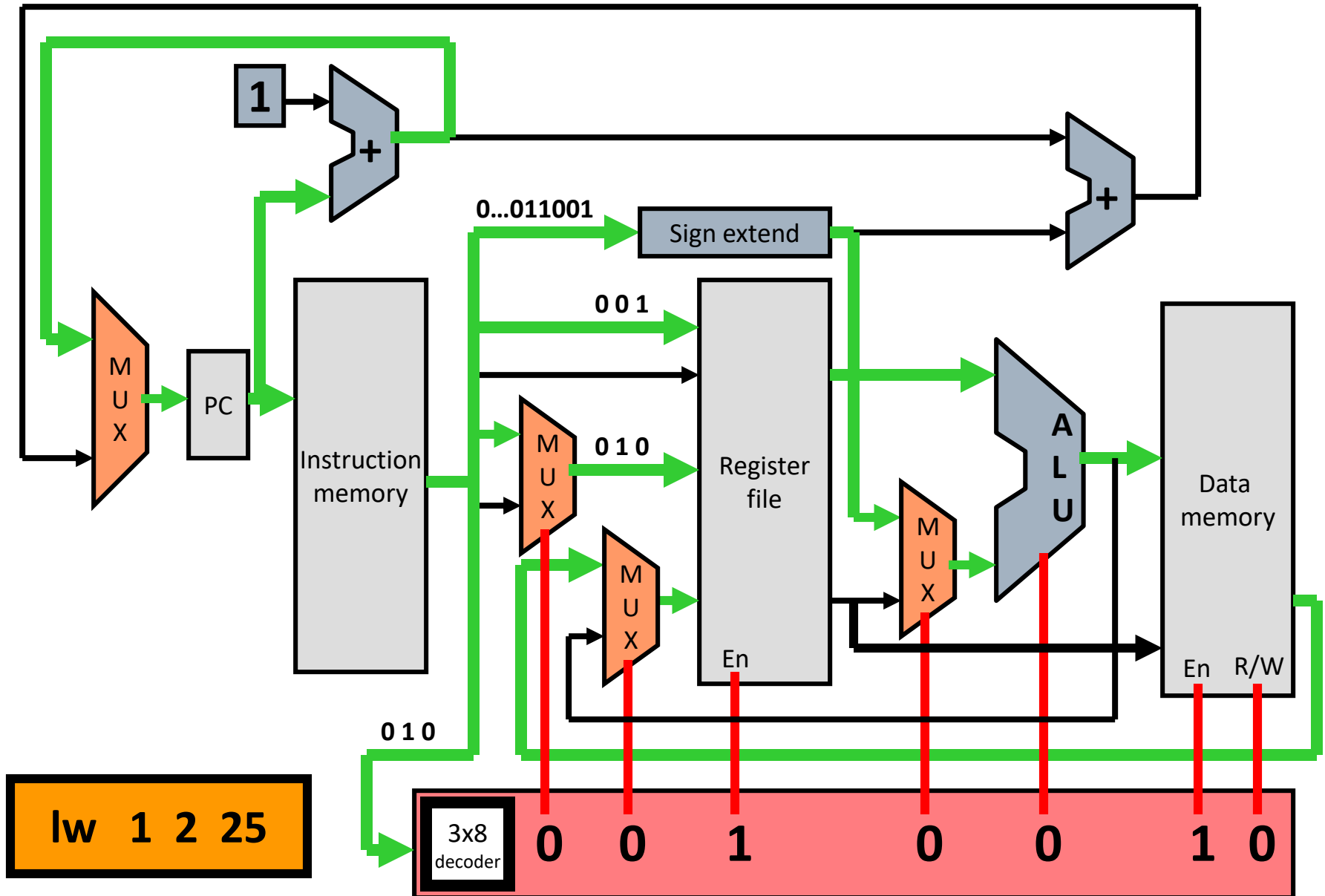
Executing **NOR** Instruction on LC2K



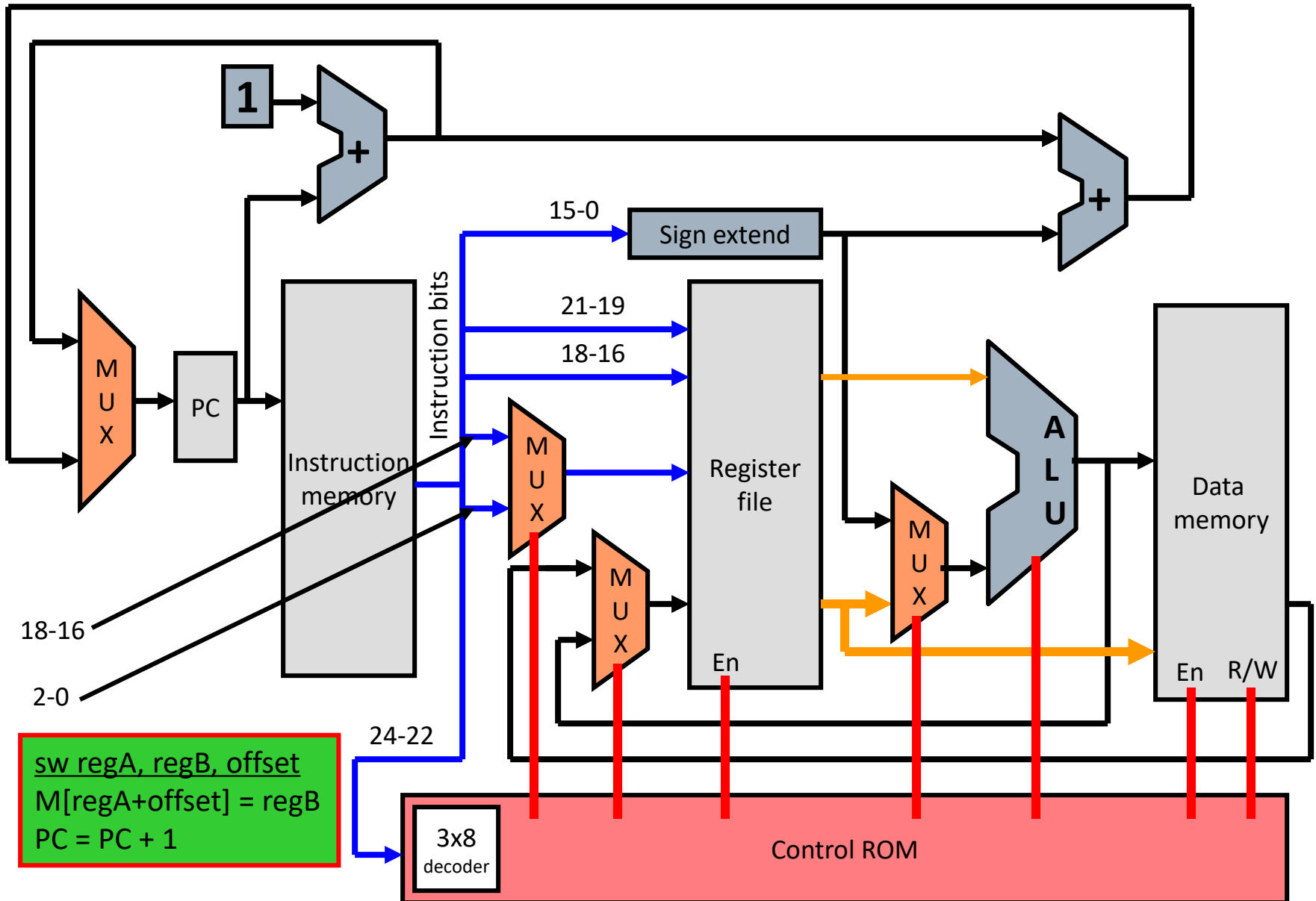
Executing a **LW** Instruction



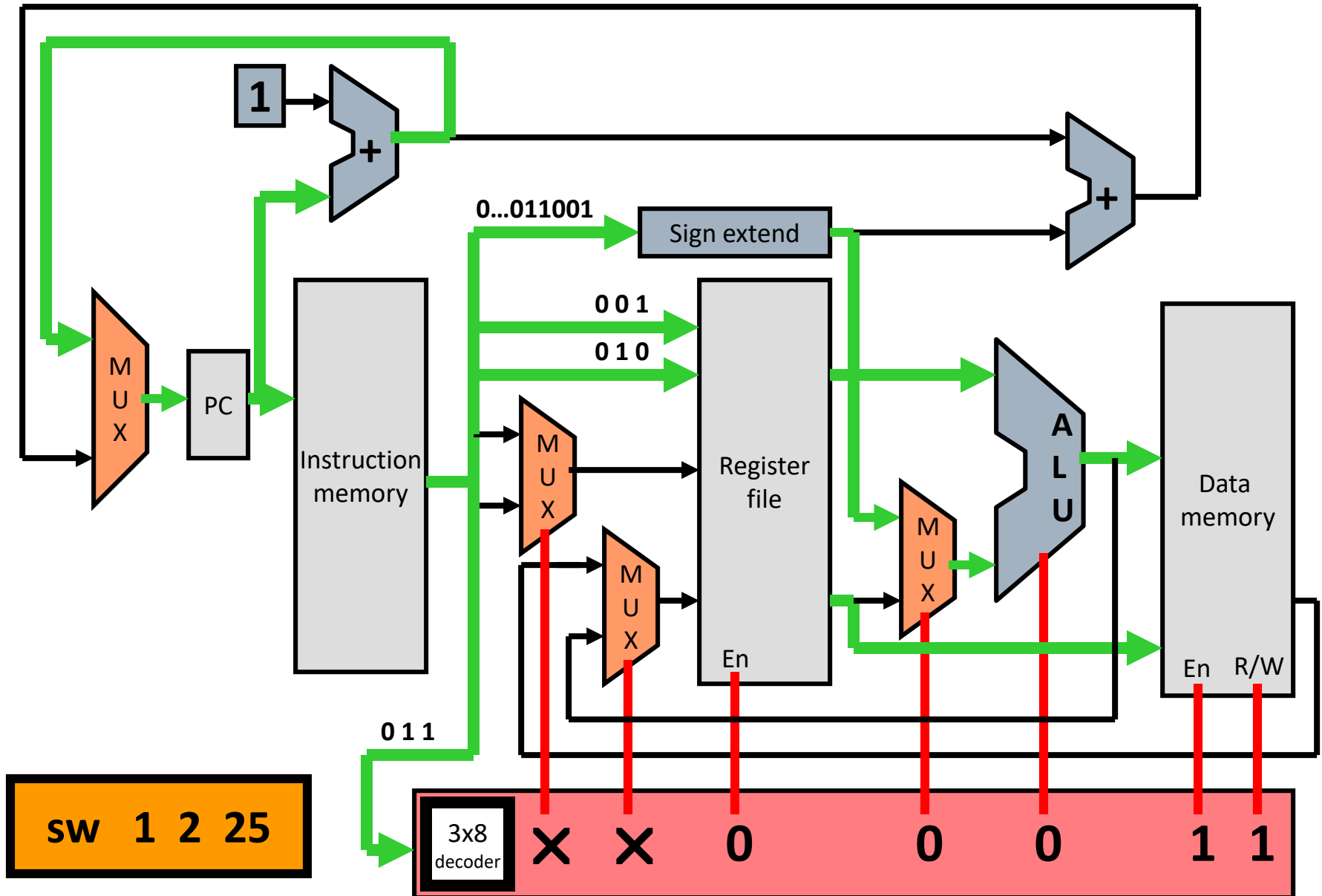
Executing a **LW** Instruction on LC2Kx Datapath



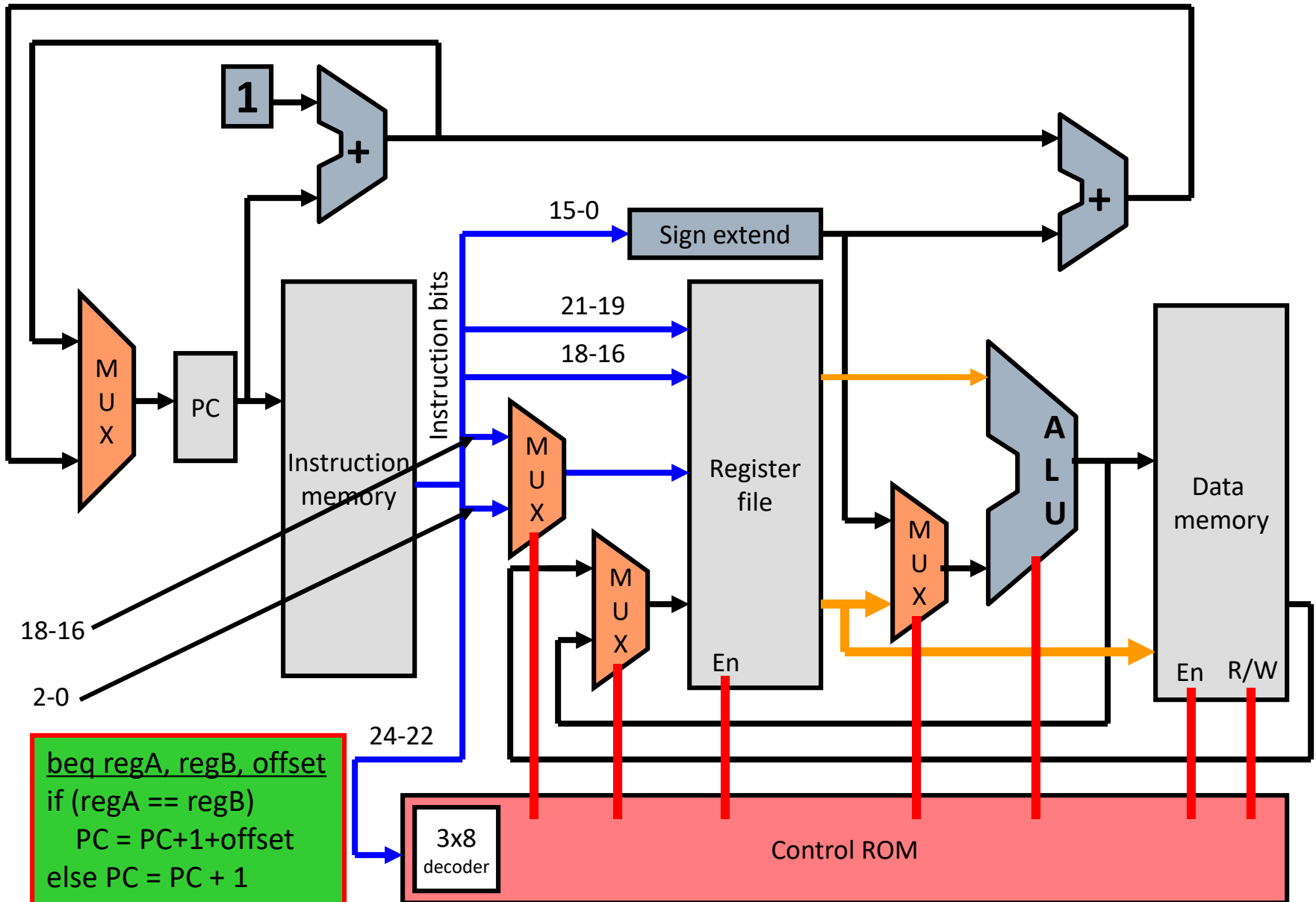
Executing a **SW** Instruction



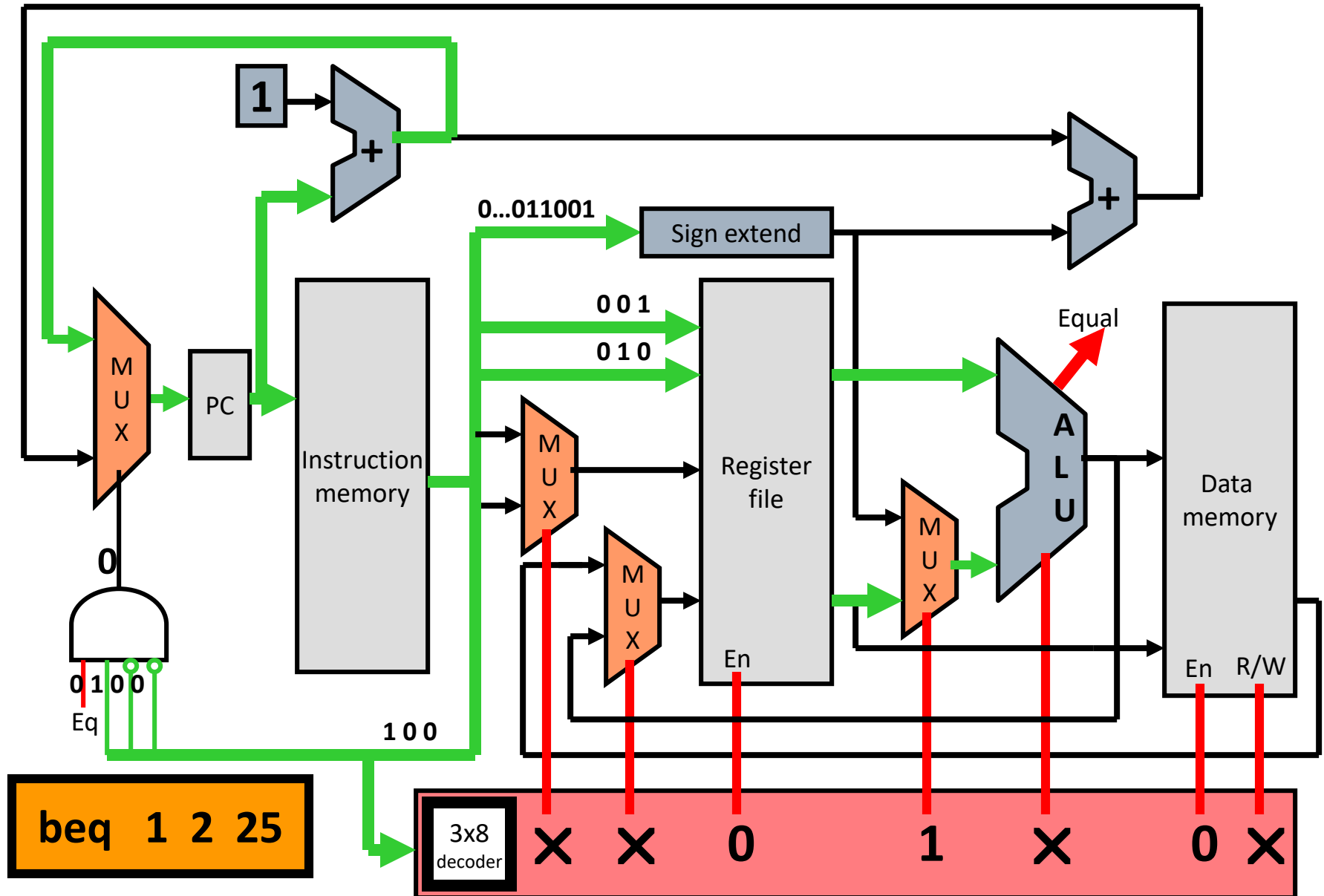
Executing a **SW** Instruction on LC2Kx Datapath



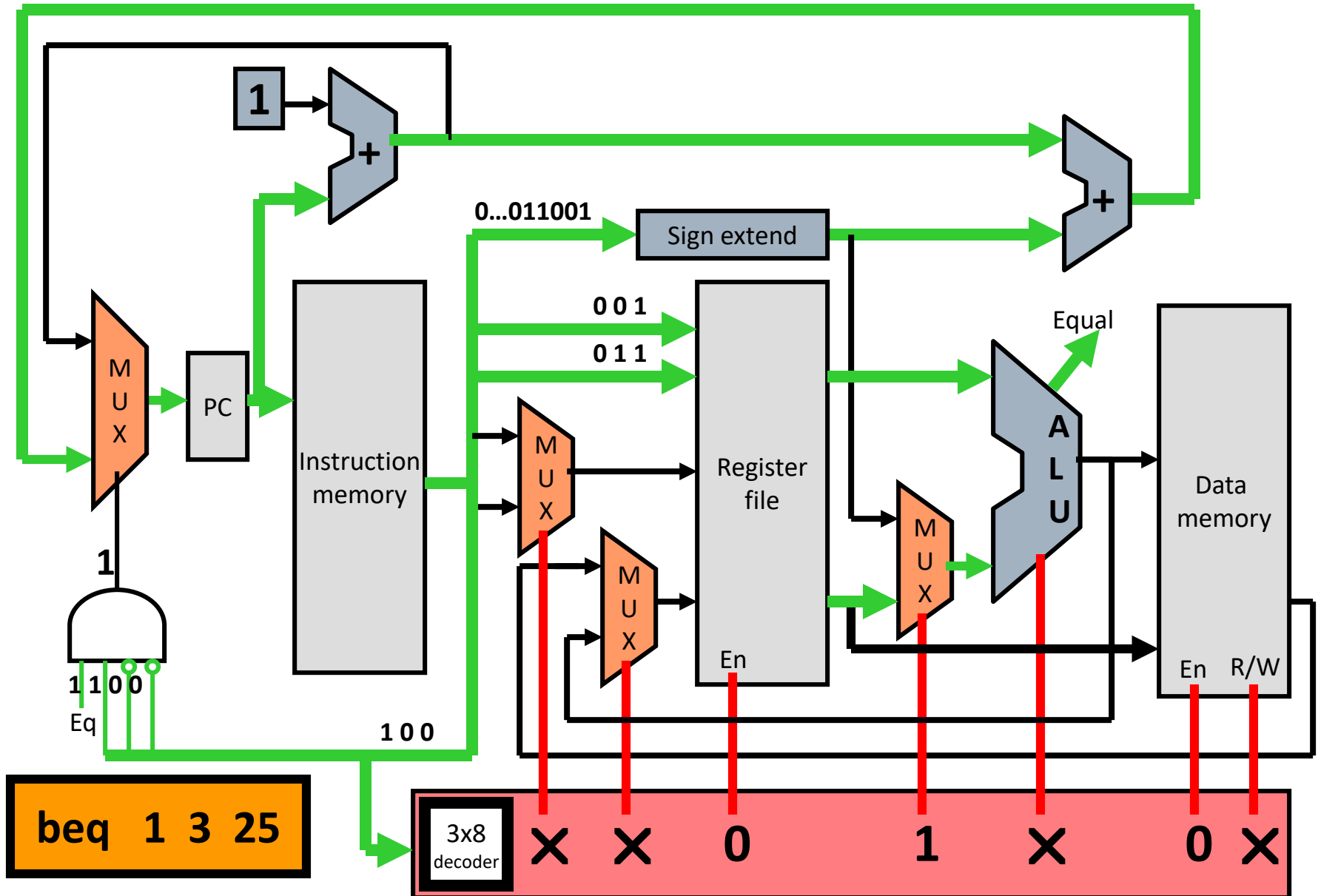
Executing a **BEQ** Instruction



Executing “not taken” **BEQ** Instruction on LC2K Datapath



Executing a “taken” **BEQ** Instruction on LC2K Datapath

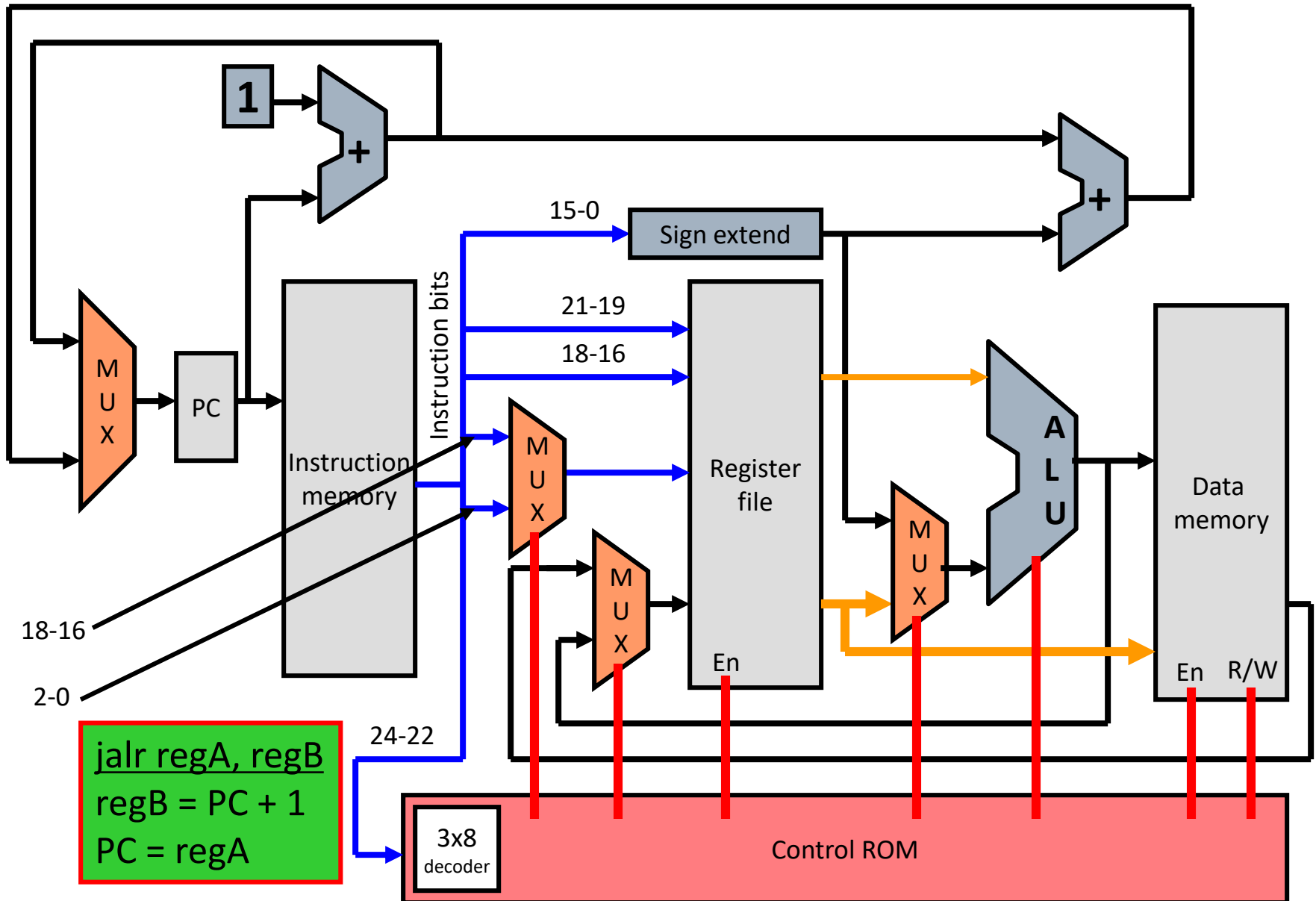


So Far, So Good

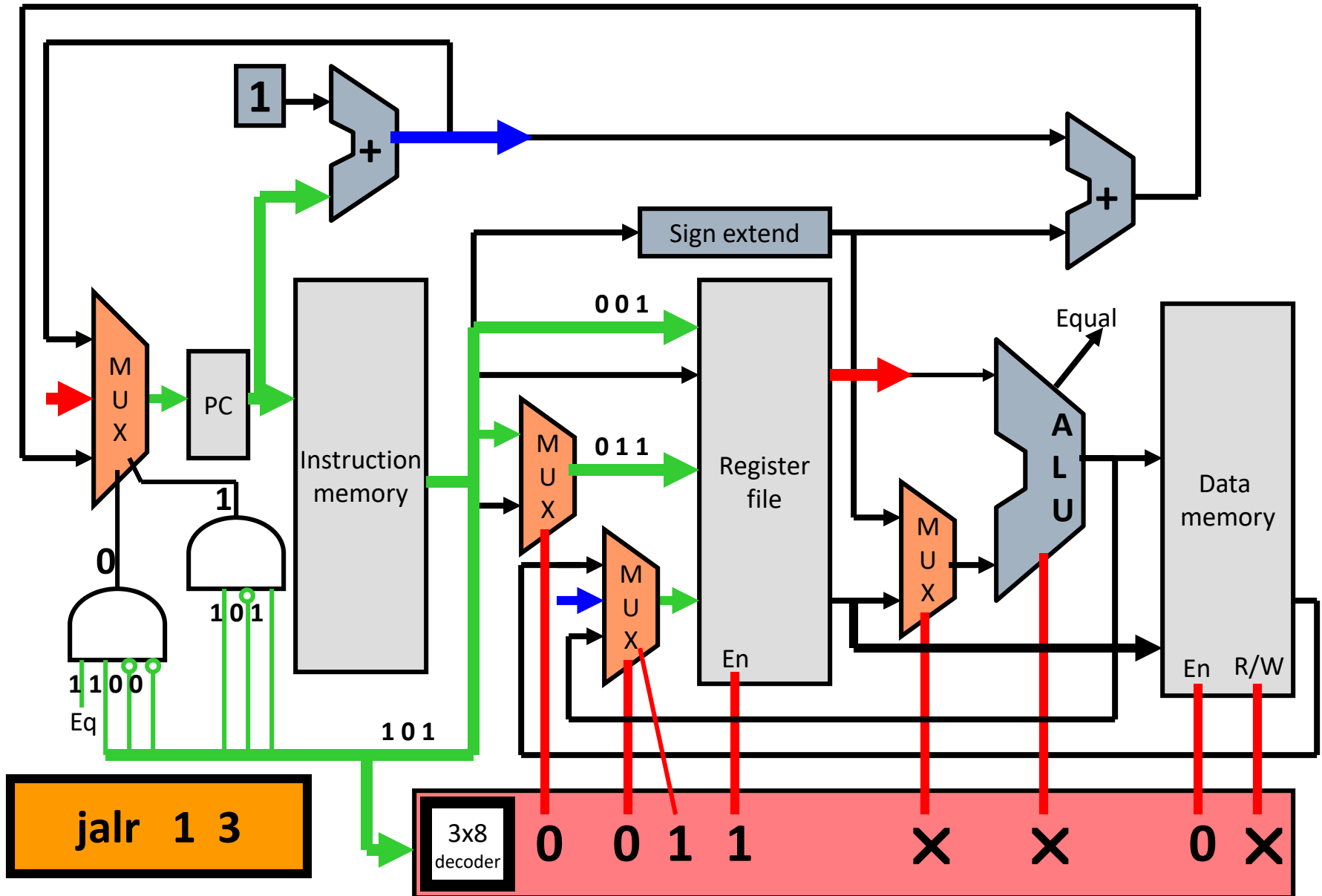
❑ Every architecture seems to have at least one ugly instruction.

- JALR doesn't fit 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

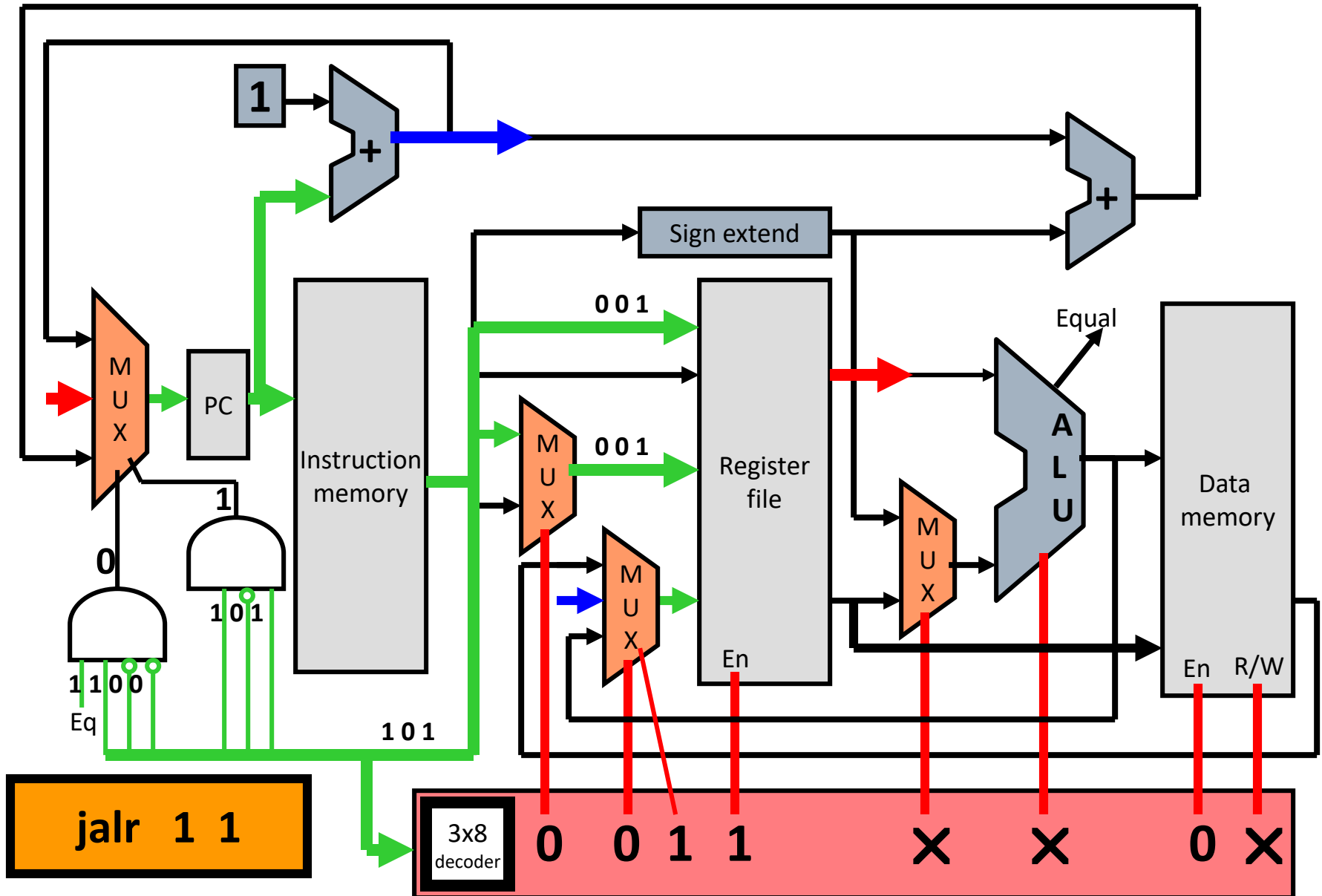
Executing a JALR Instruction



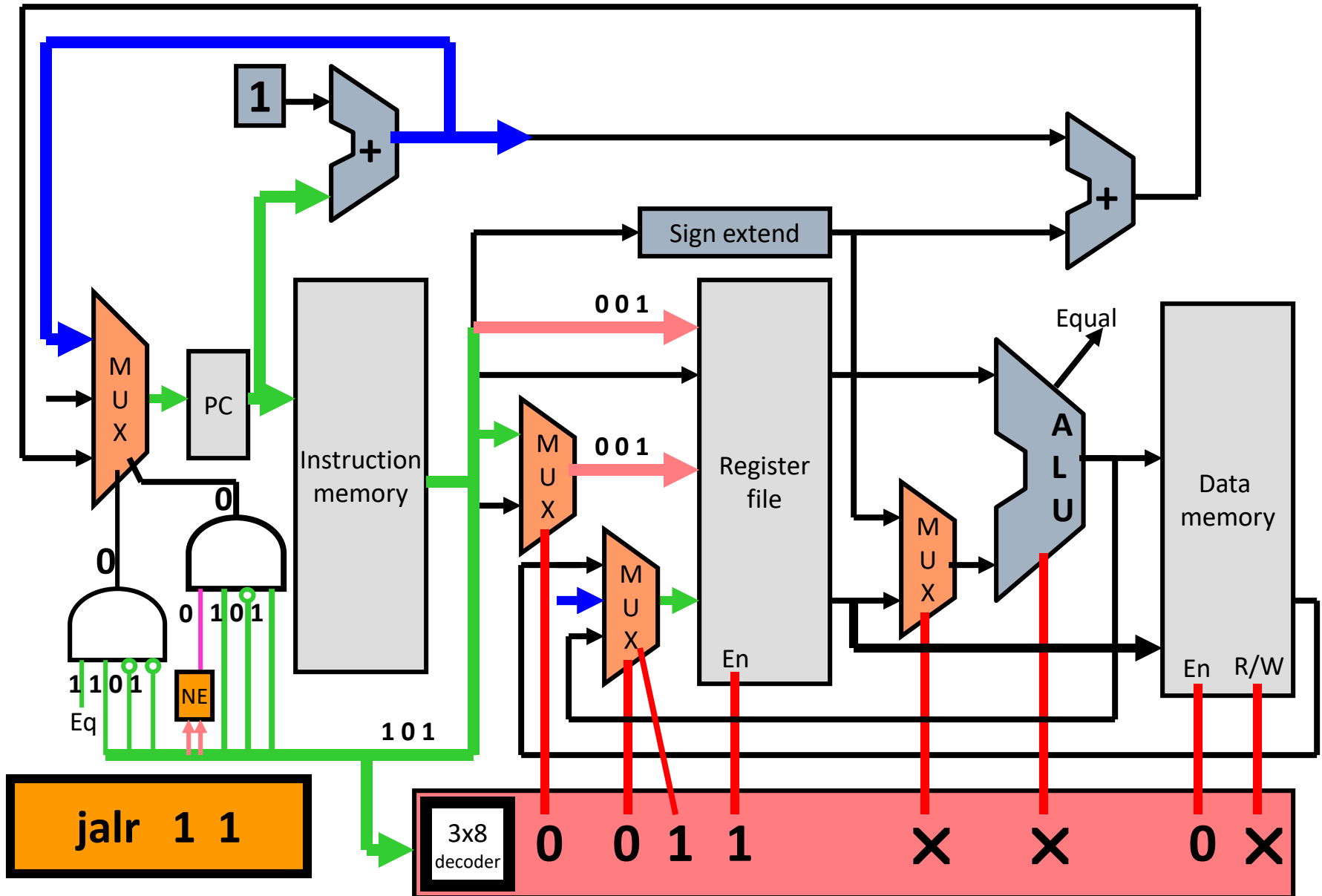
Executing a **JALR** Instruction on LC2Kx Datapath



What If regA = regB for a **JALR** ?



Changes for a **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

| | 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:

$$100 * 8\text{ns} = \underline{\mathbf{800}} \text{ ns}$$

Optimal execution:

$$25*8\text{ns} + 10*7\text{ns} + 45*6\text{ns} + 20*5\text{ns} = \underline{\mathbf{640}} \text{ ns}$$

What about other instructions?

- ❑ We could imagine replacing an instruction (noop) with some other instruction in our ISA.
 - We'd have to support that in our microarchitecture.

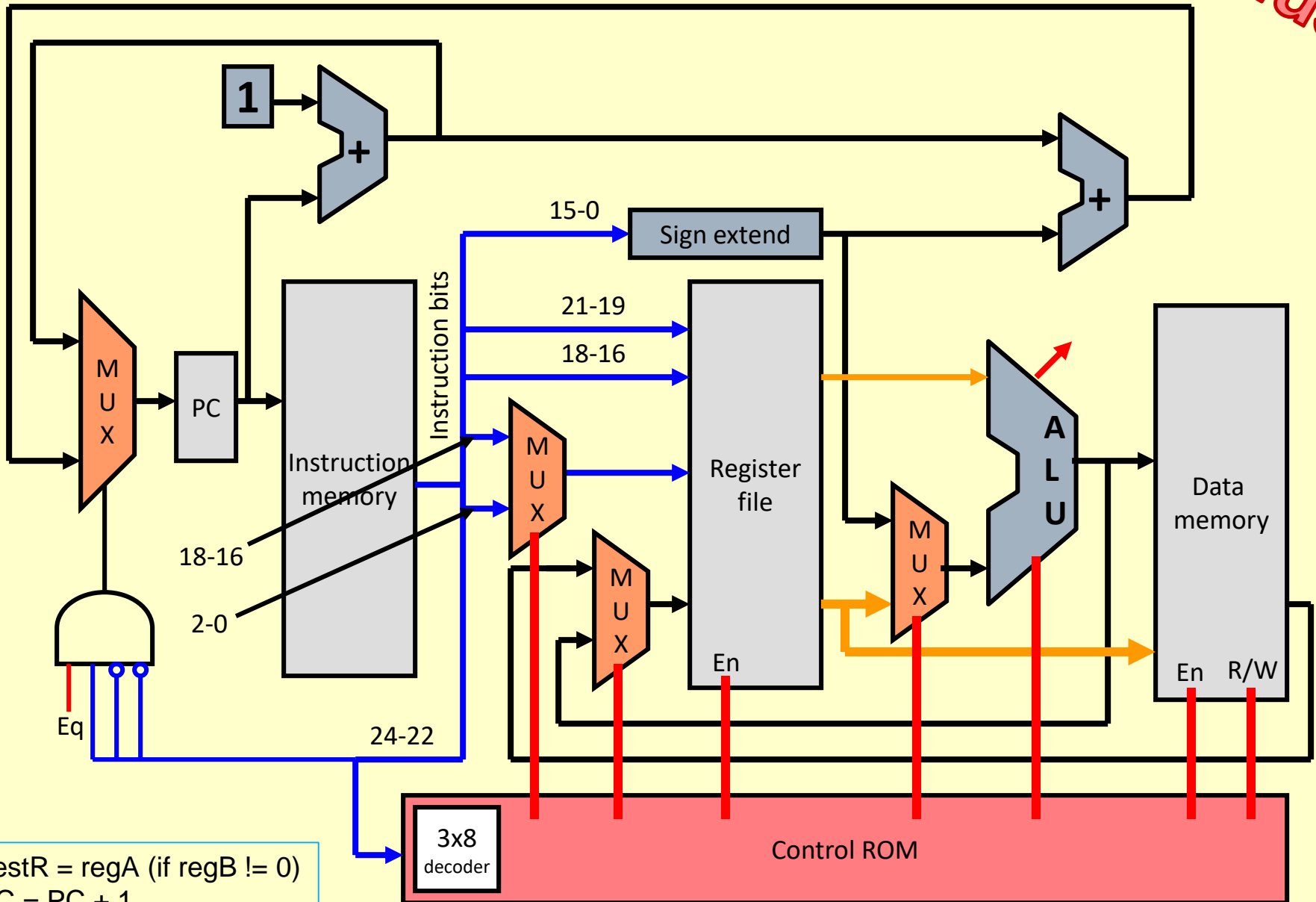
- ❑ The following slides discuss on possible instruction.
 - We probably won't have time to do this in class, but it's a useful exercise so it's included here as "bonus" material.

Class Problem

- ❑ Extend the single cycle datapath to perform the following operation
 - `cmov regA, regB, destR`
 - `destR = regA` (if `regB != 0`)
 - `PC = PC + 1`

Class Problem (continued)

Bonus



Class Problem (continued)

Bonus

