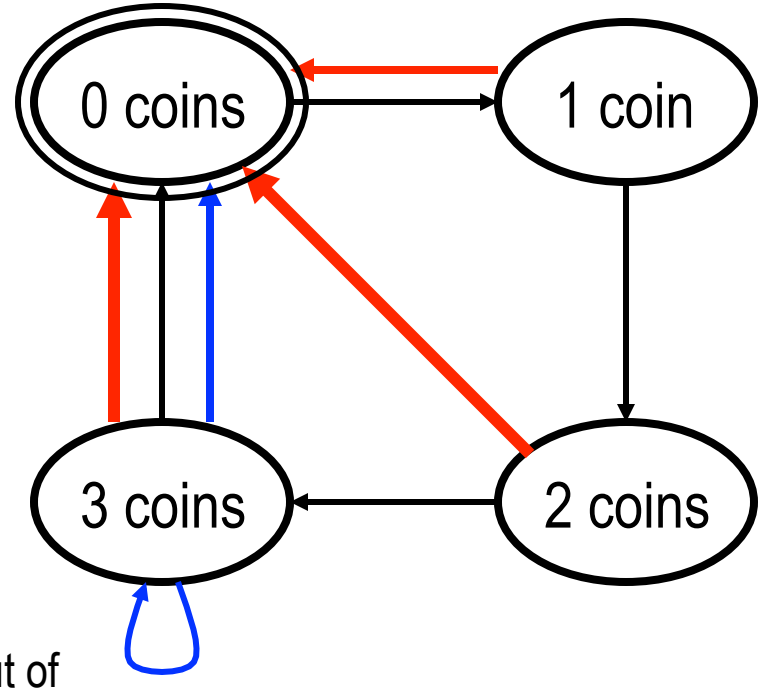
# Recap: FSM for Vending Machine



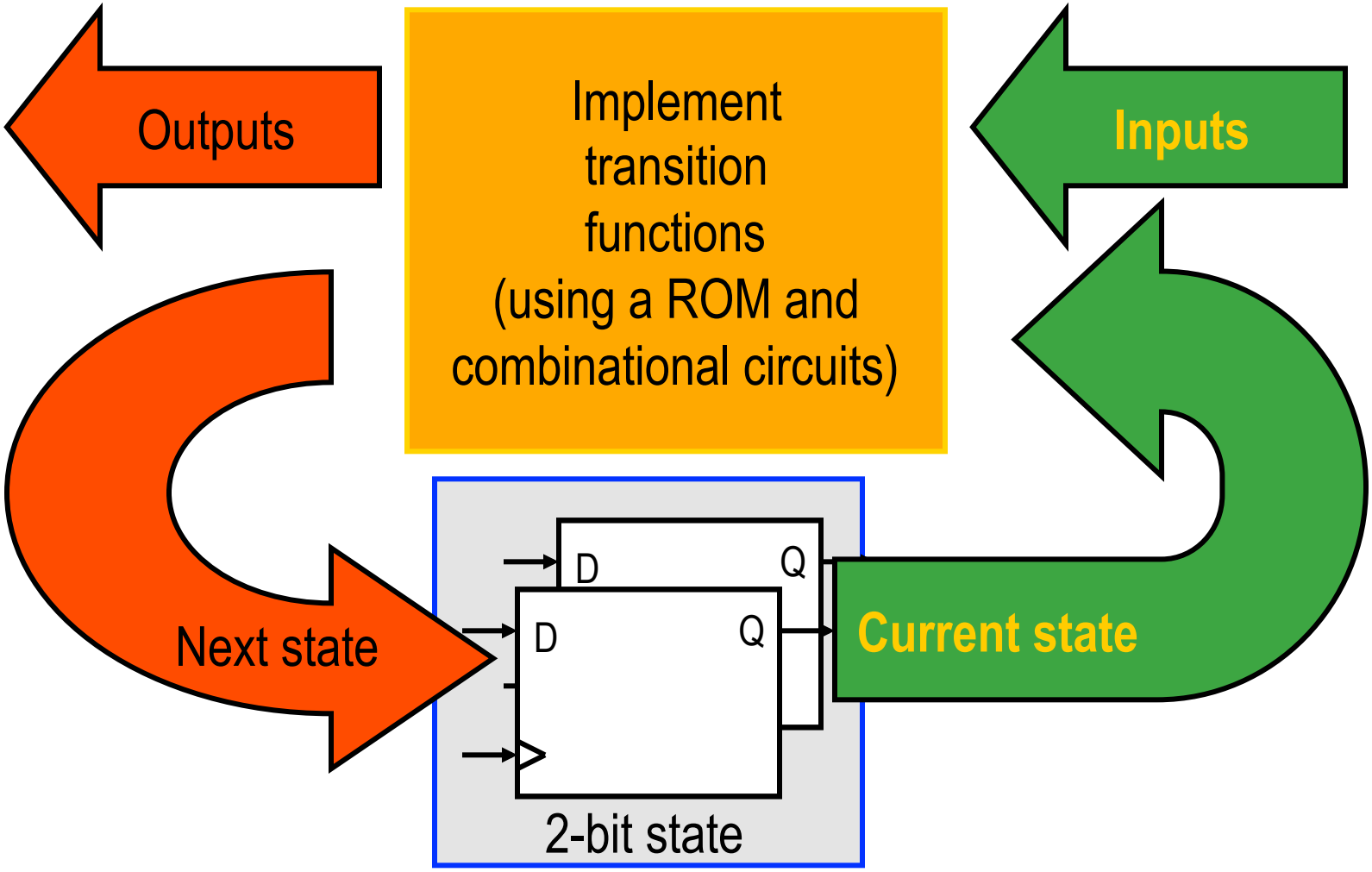

Ran out of specific drink selection

Coin trigger

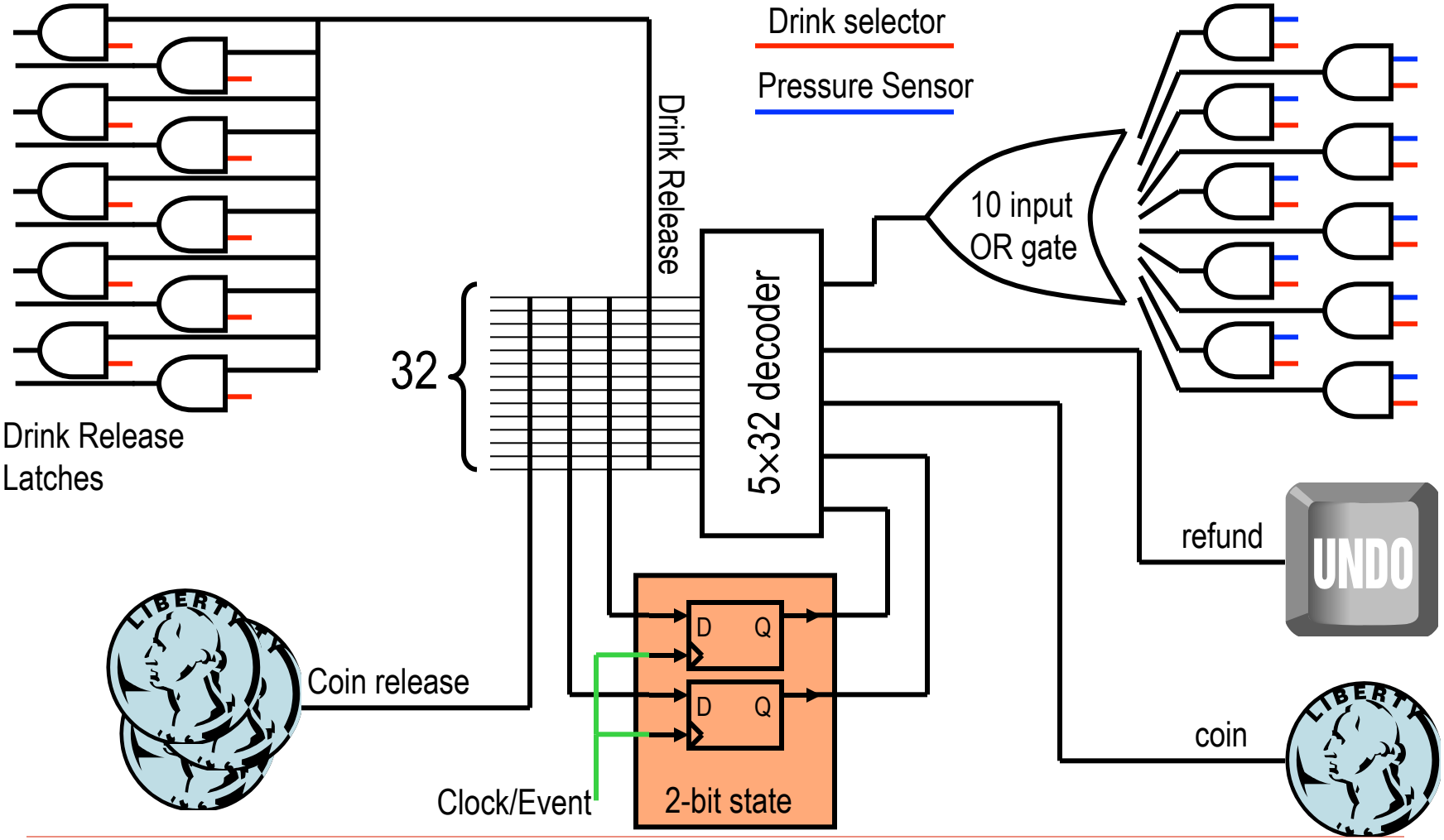
Refund button

Drink Select

# Recap: Implementing a FSM



# Putting it all together



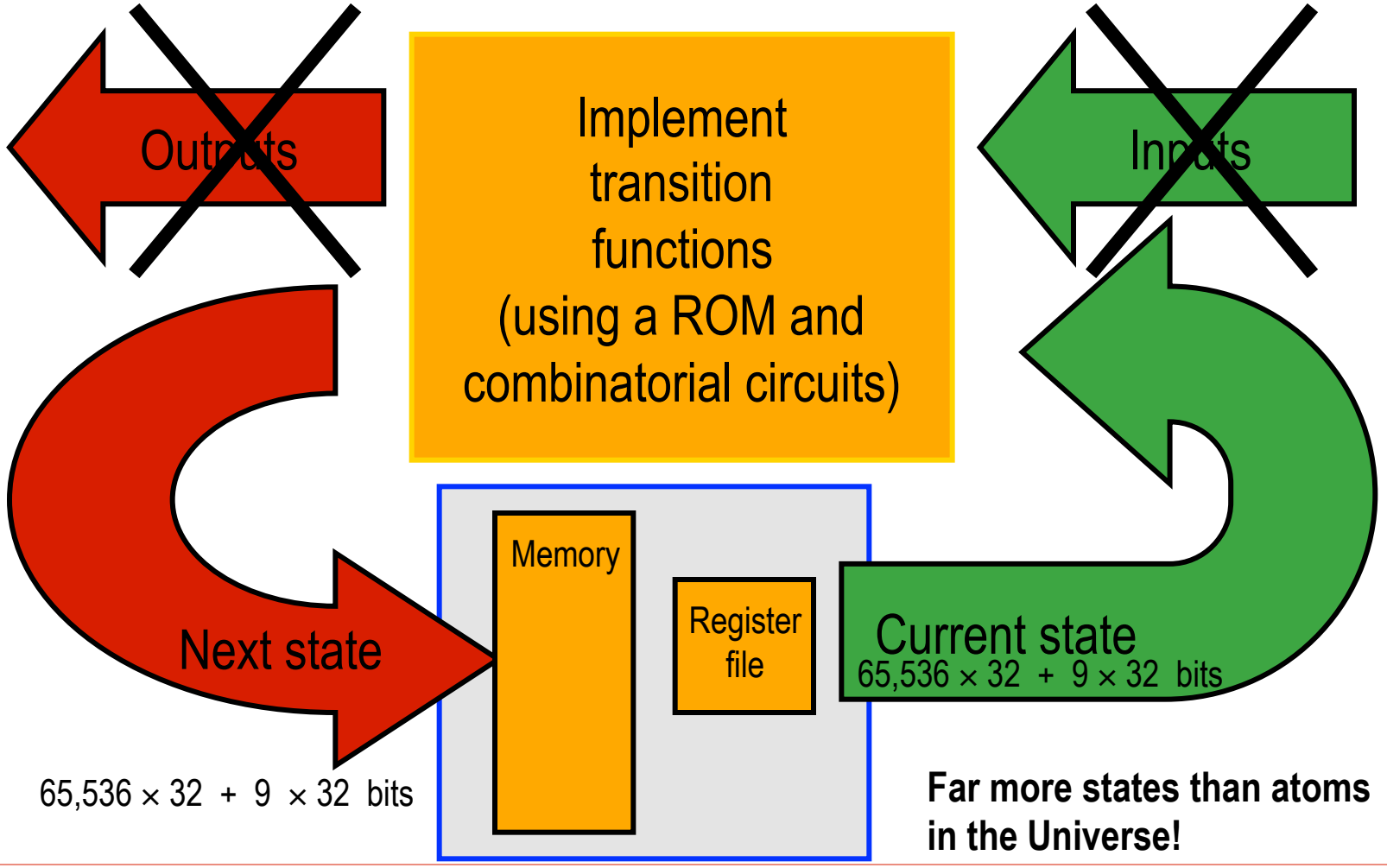
# Limitations of the controller

- 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

# Single-Cycle Processor Design

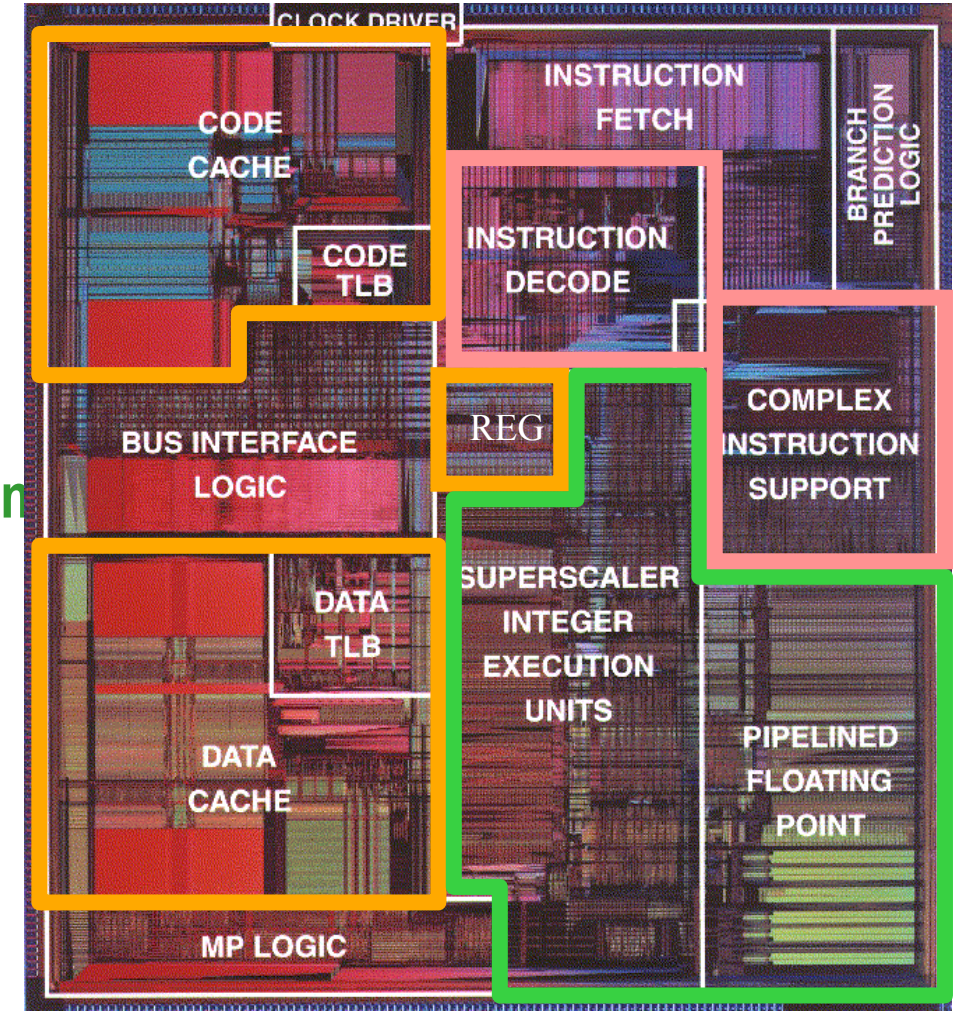
- General-Purpose Processor Design
  - Fetch Instructions
  - Decode Instructions
    - Instructions are input to control ROM
  - ROM data controls movement of data
    - Incrementing PC, reading registers, ALU control
  - Clock drives it all
  - Single-cycle datapath:  Each instruction completes in one clock cycle

# LC2Kx Processor as FSM

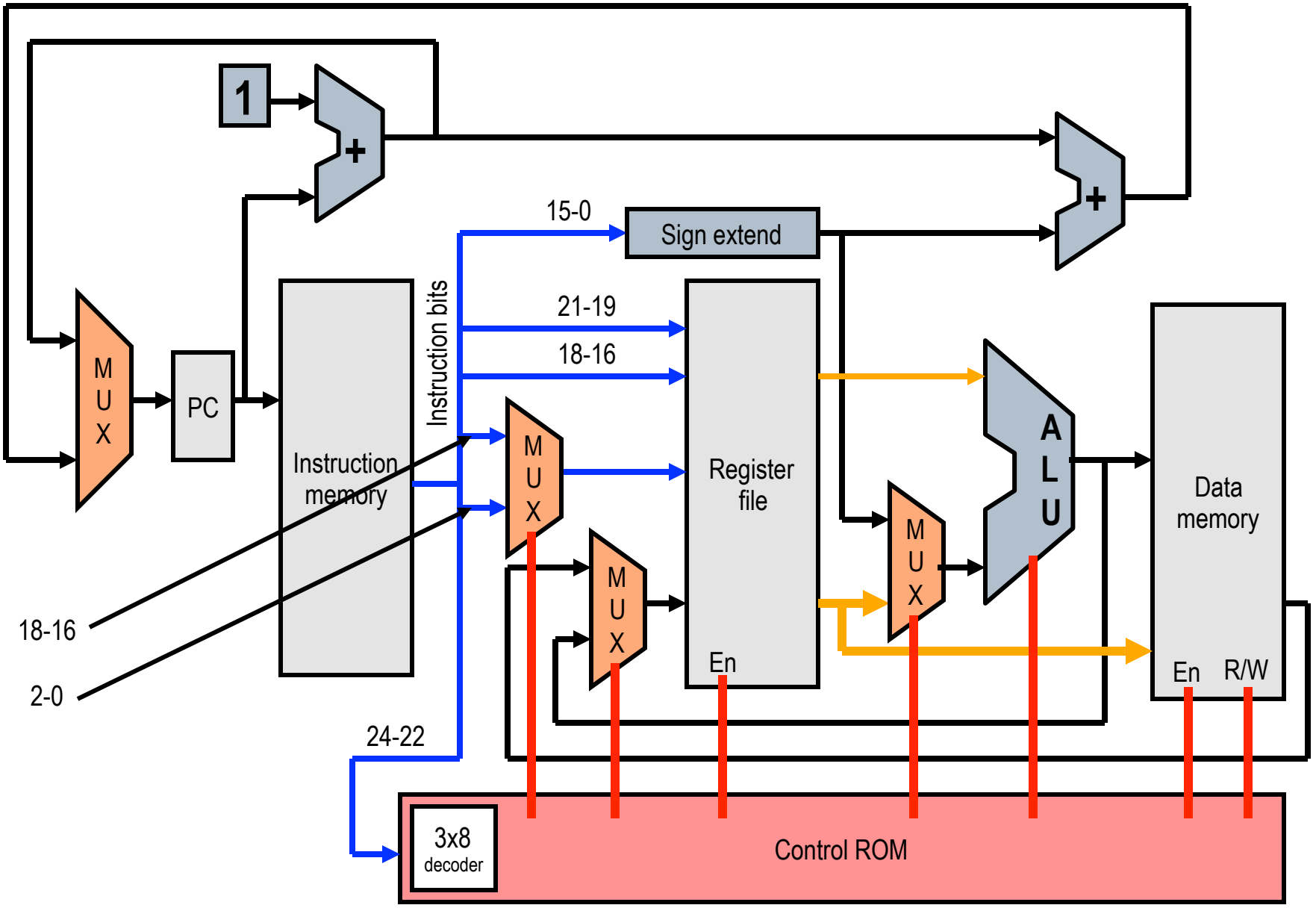


# Pentium Processor Die

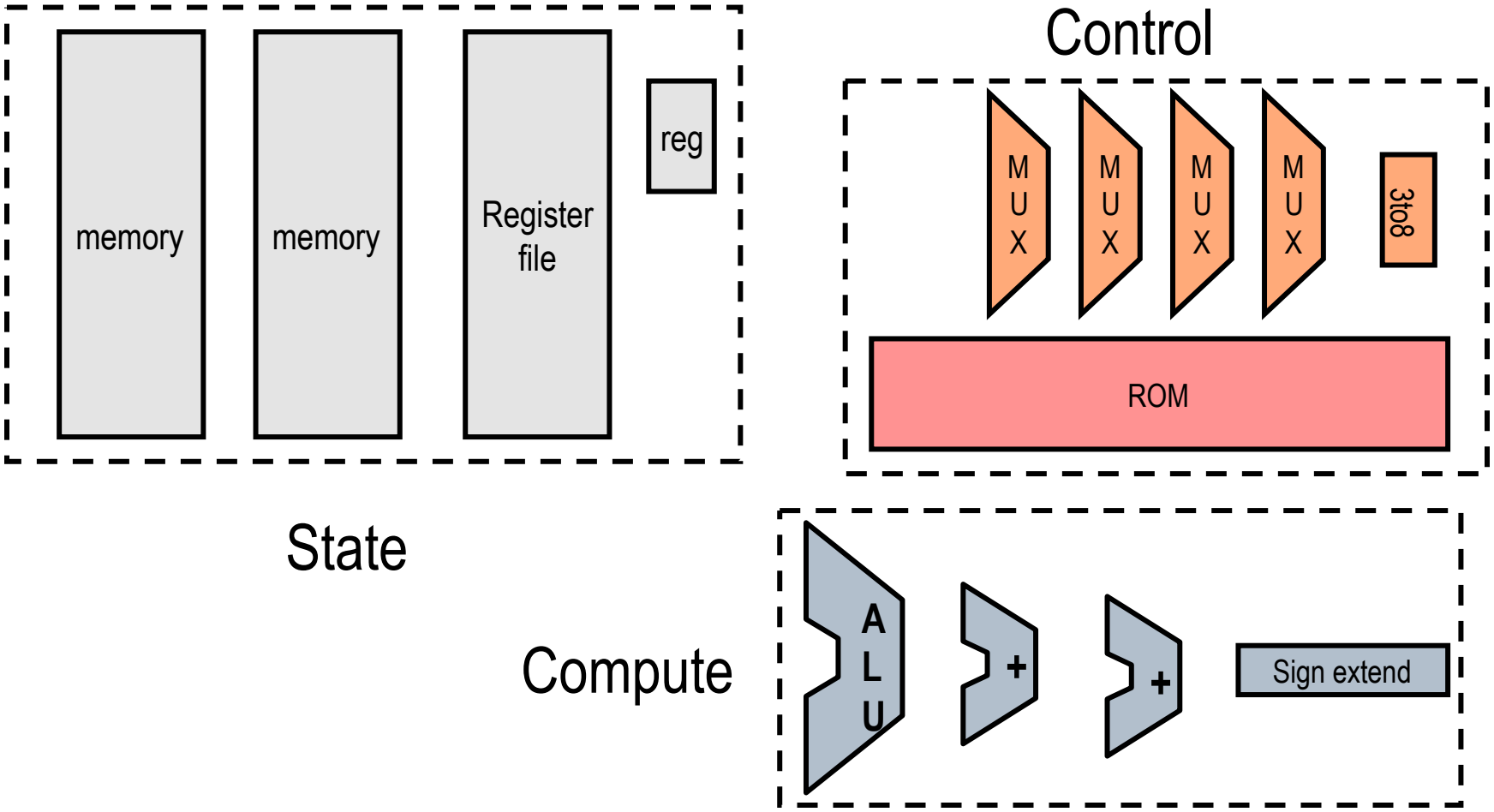
- State
  - Registers
  - Memory
- Control ROM
- Combinational logic (Con



# LC2Kx Datapath Implementation

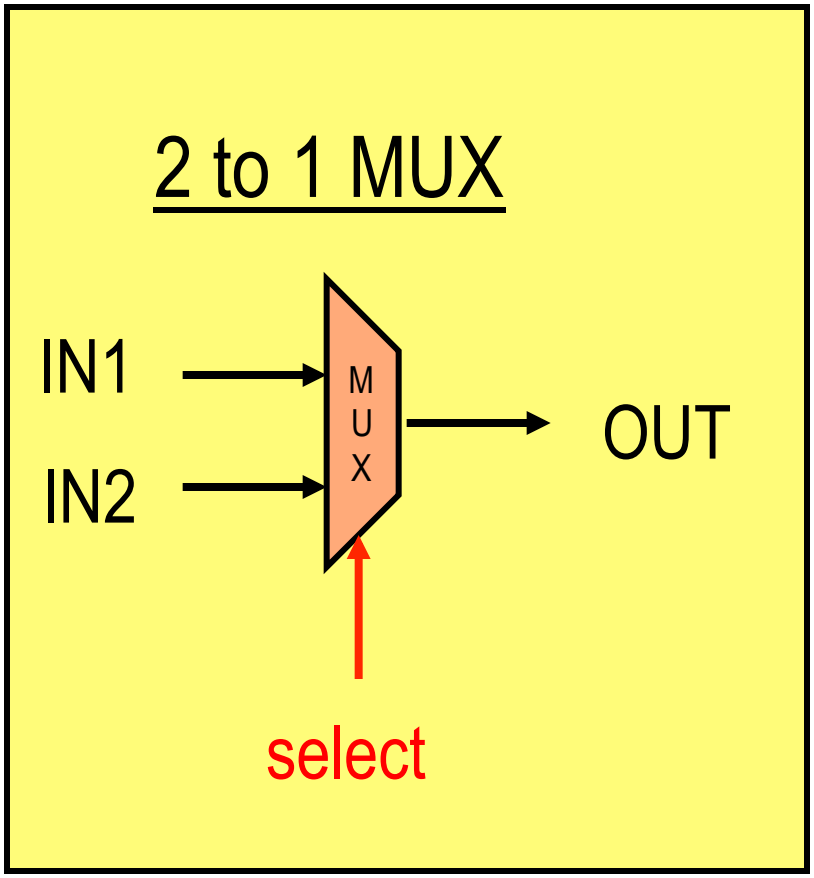


# Building Blocks for the LC2




Here are the pieces, go build yourself a processor!

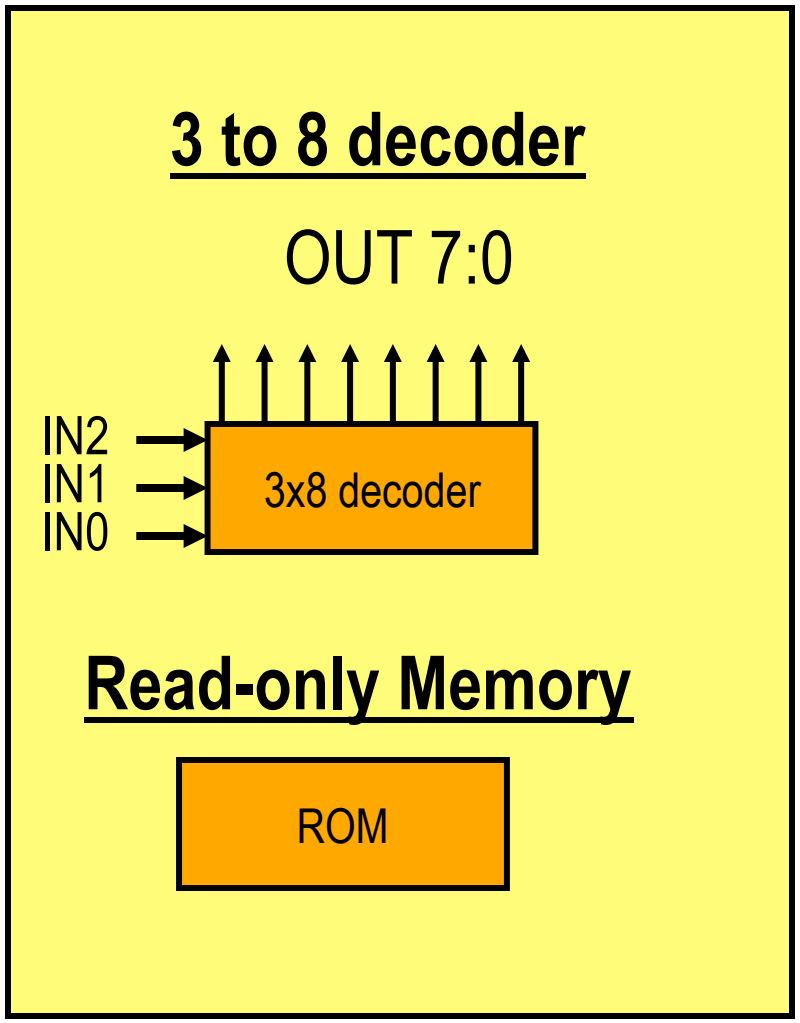
# Control Building Blocks (1)




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

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

# Control Building Blocks (2)

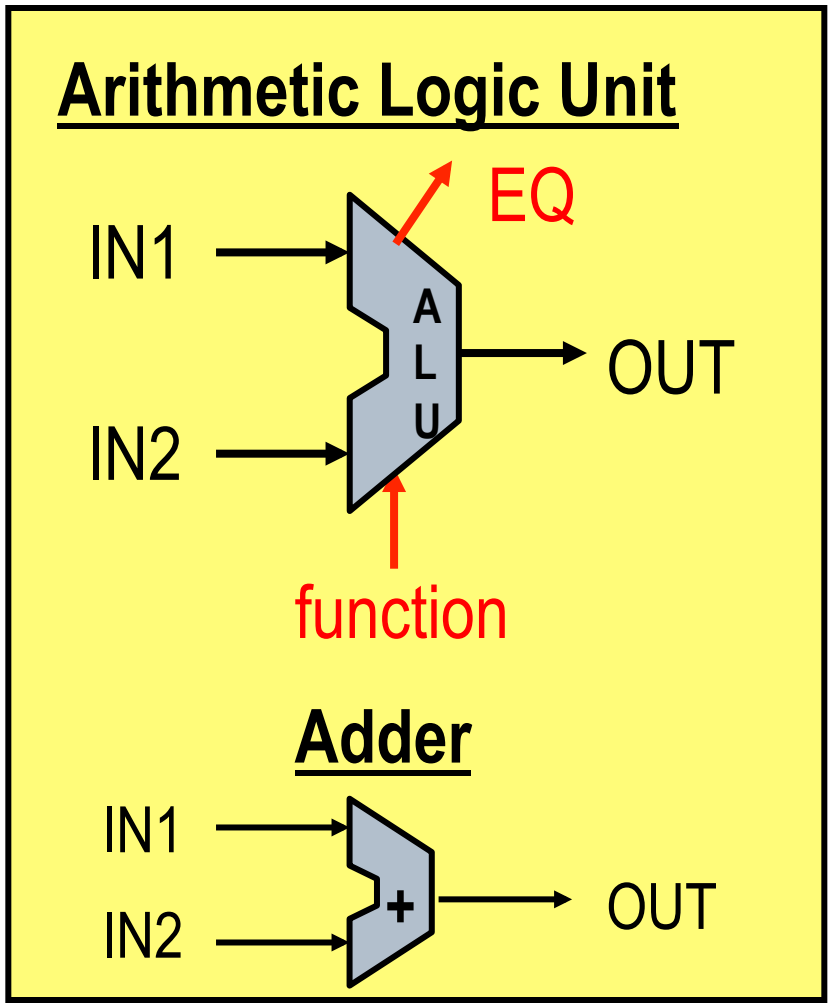



Decoder activates one of the output lines based on the input

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

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

# Compute Building Blocks (1)



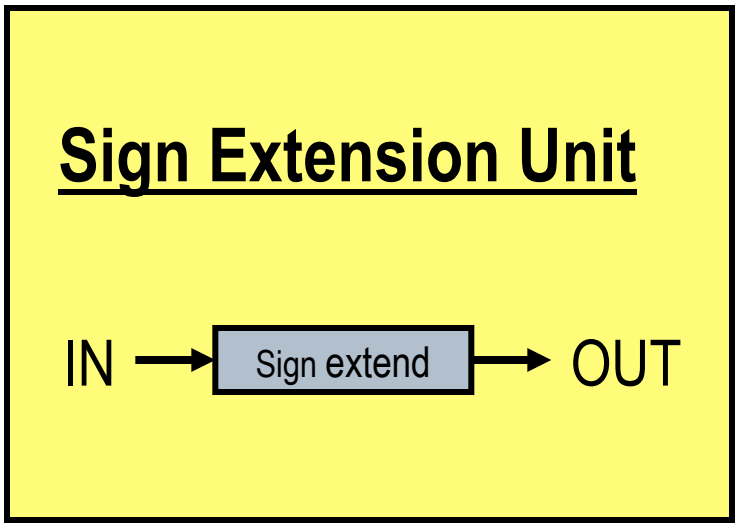

Perform basic arithmetic functions

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

For LC2, f = add, nand.
For other processors, there are many more functions.

Simple ALU – Does only adds

# Compute Building Blocks (2)



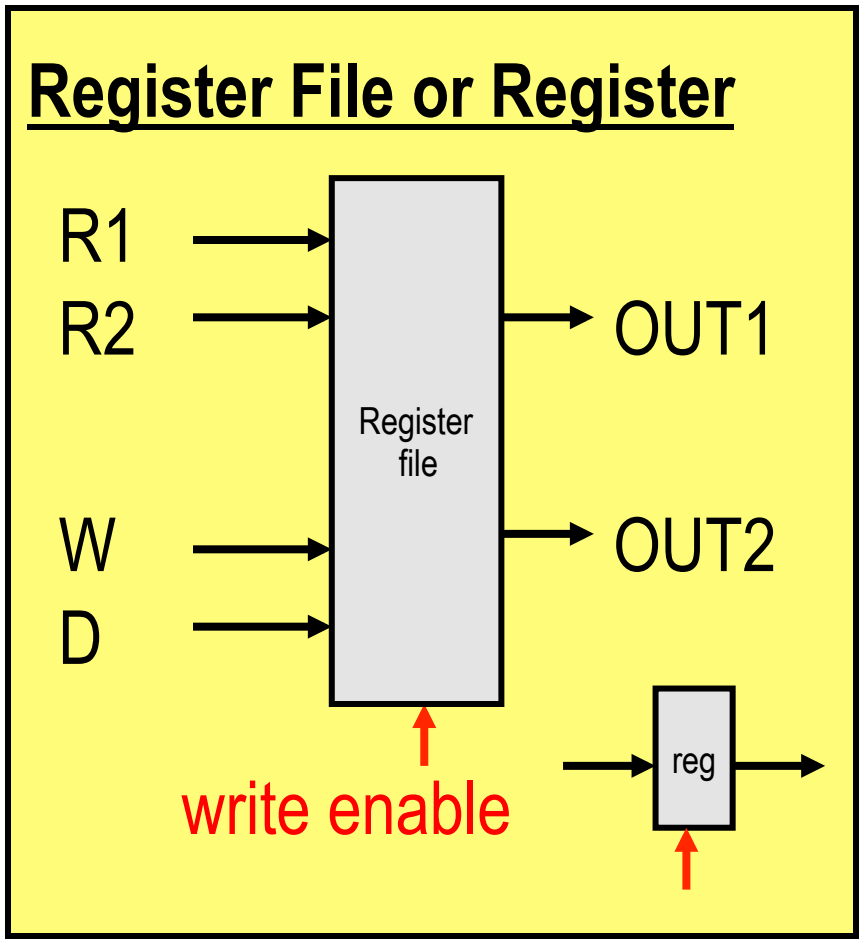

Sign extend input by replicating the MSB to width of output

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)



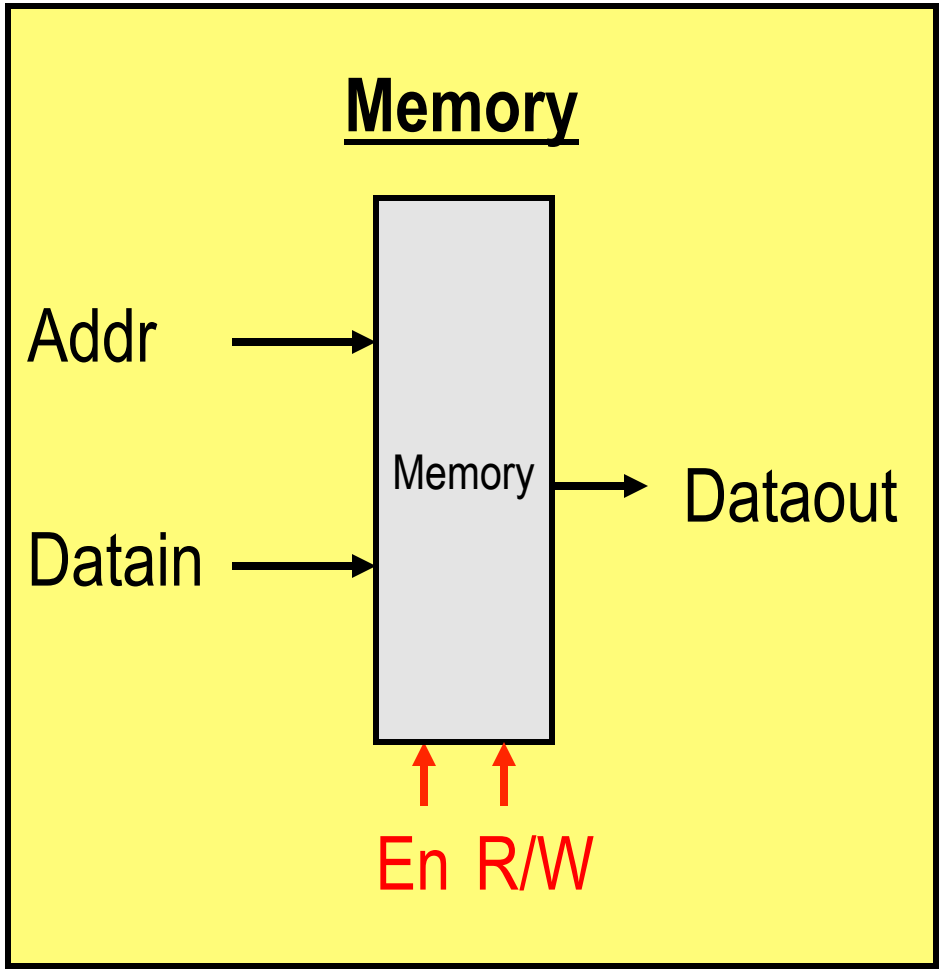

Small/fast memory to store temporary values

- n entries (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

How many bits are Ri and Wi in LC2?

# State Building Blocks (2)

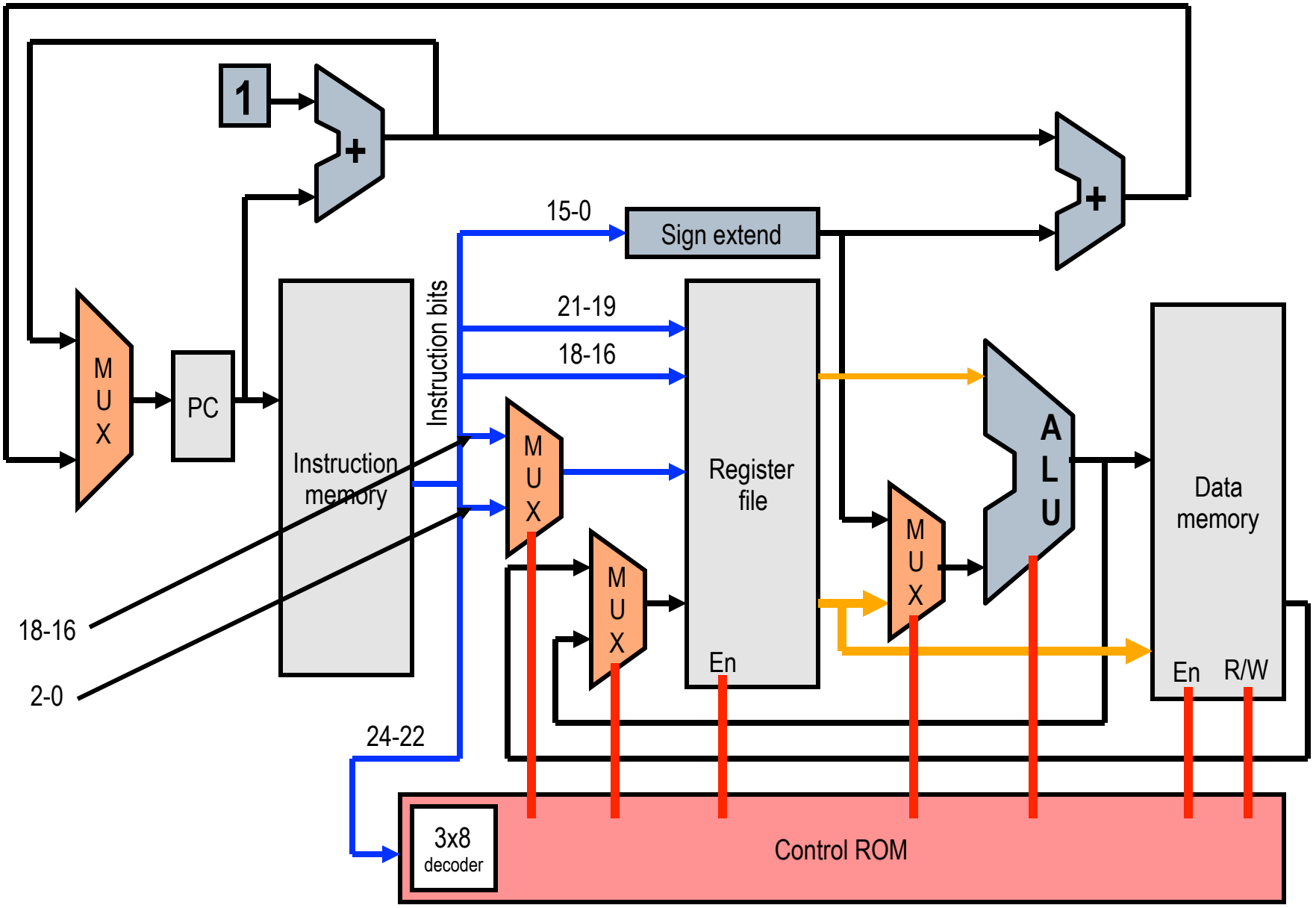



Slower storage structure to hold large amounts of stuff.

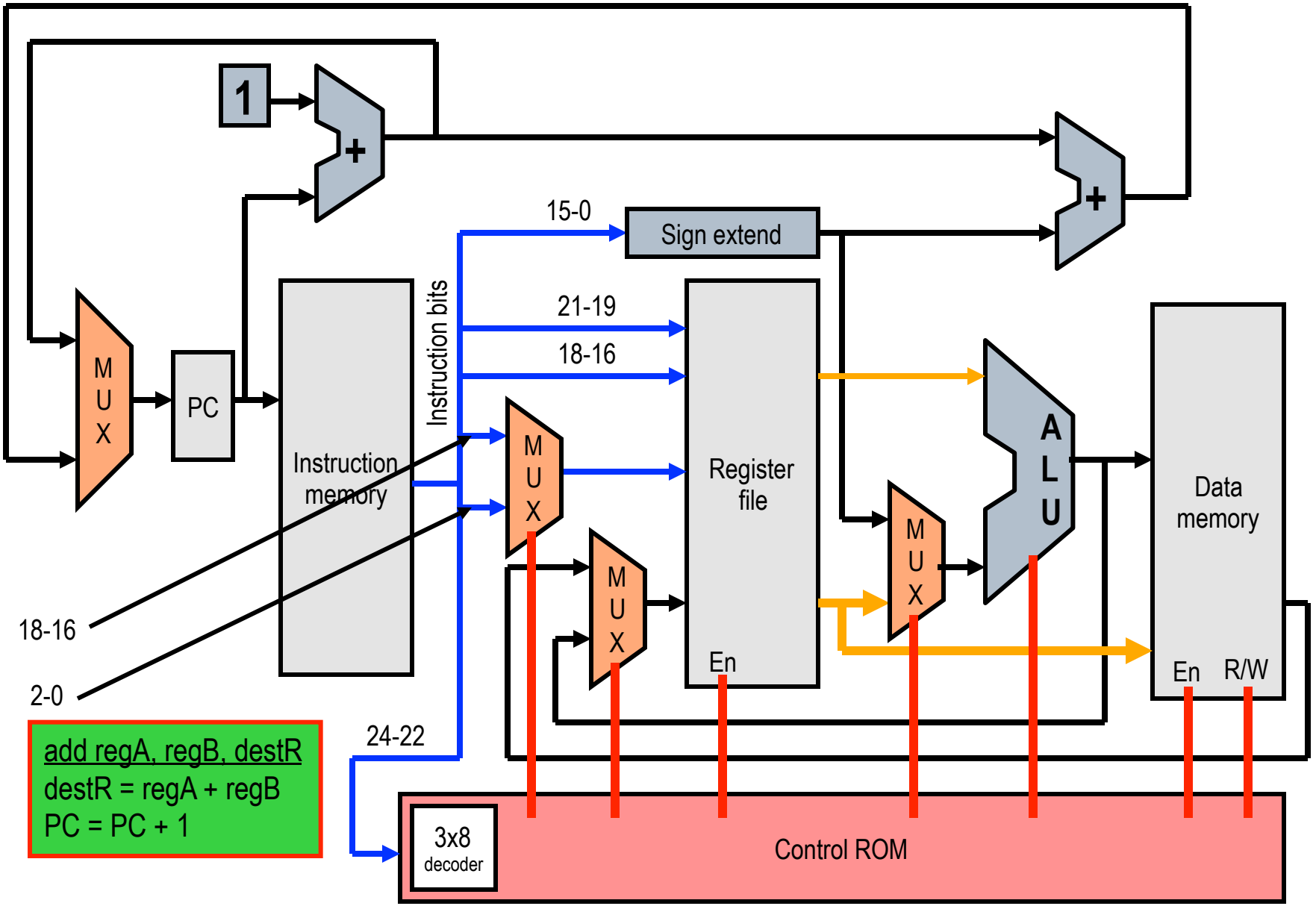
Use 2 memories for LC2

* Instructions
* Data
* 65,536 total words

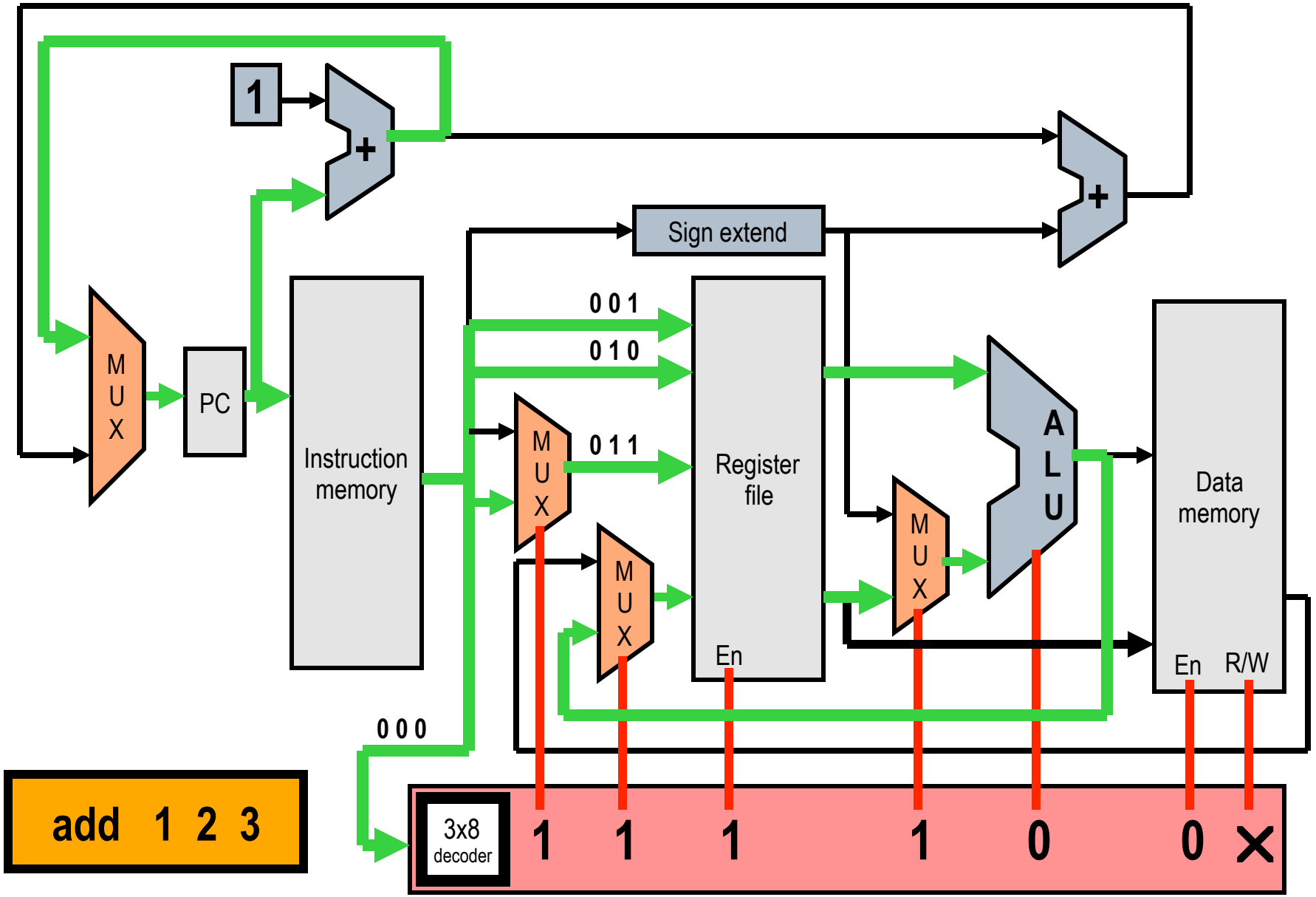
# LC2Kx Datapath Implementation



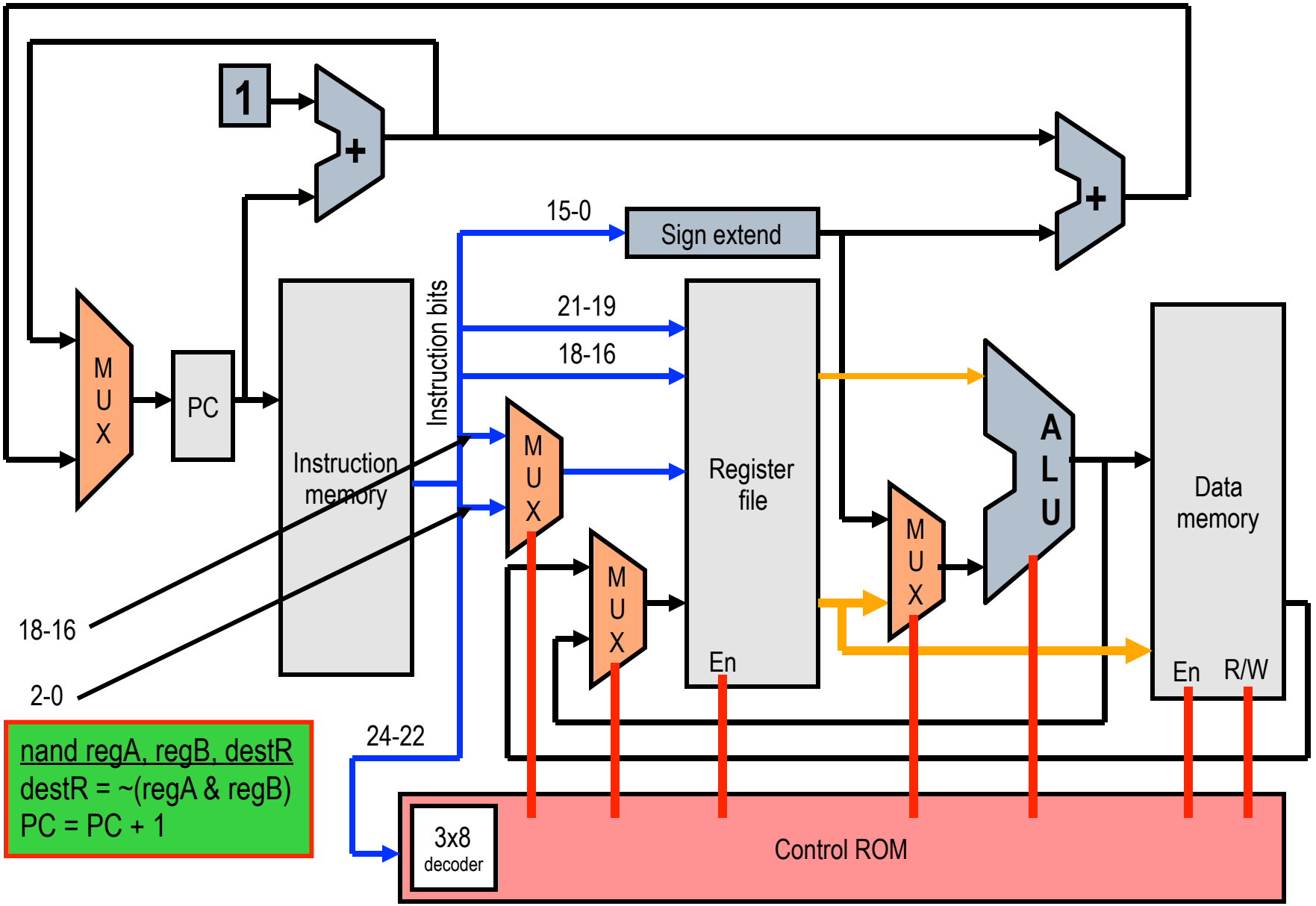
# Executing an ADD Instruction



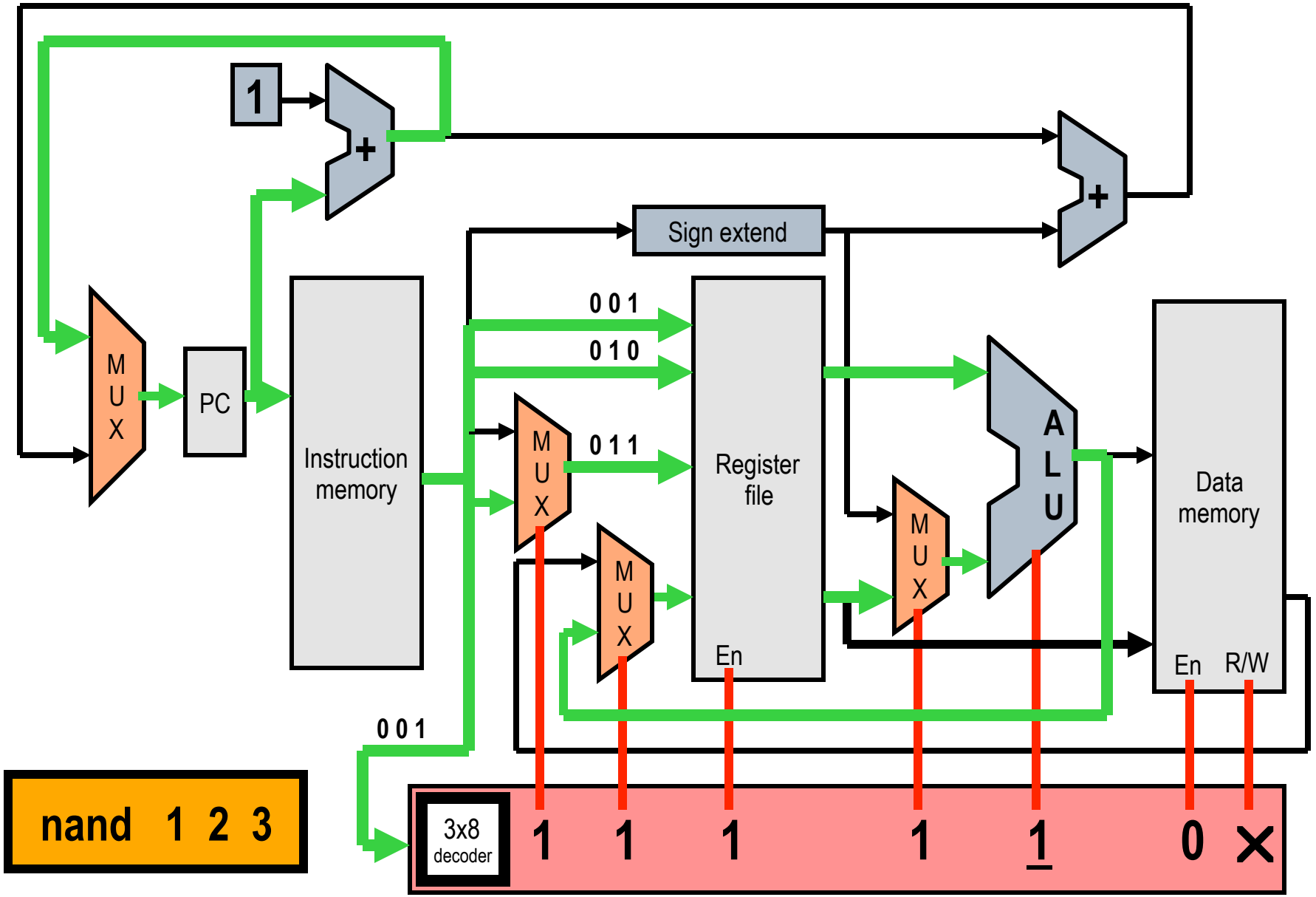
# Executing an ADD Instruction on LC2Kx Datapath



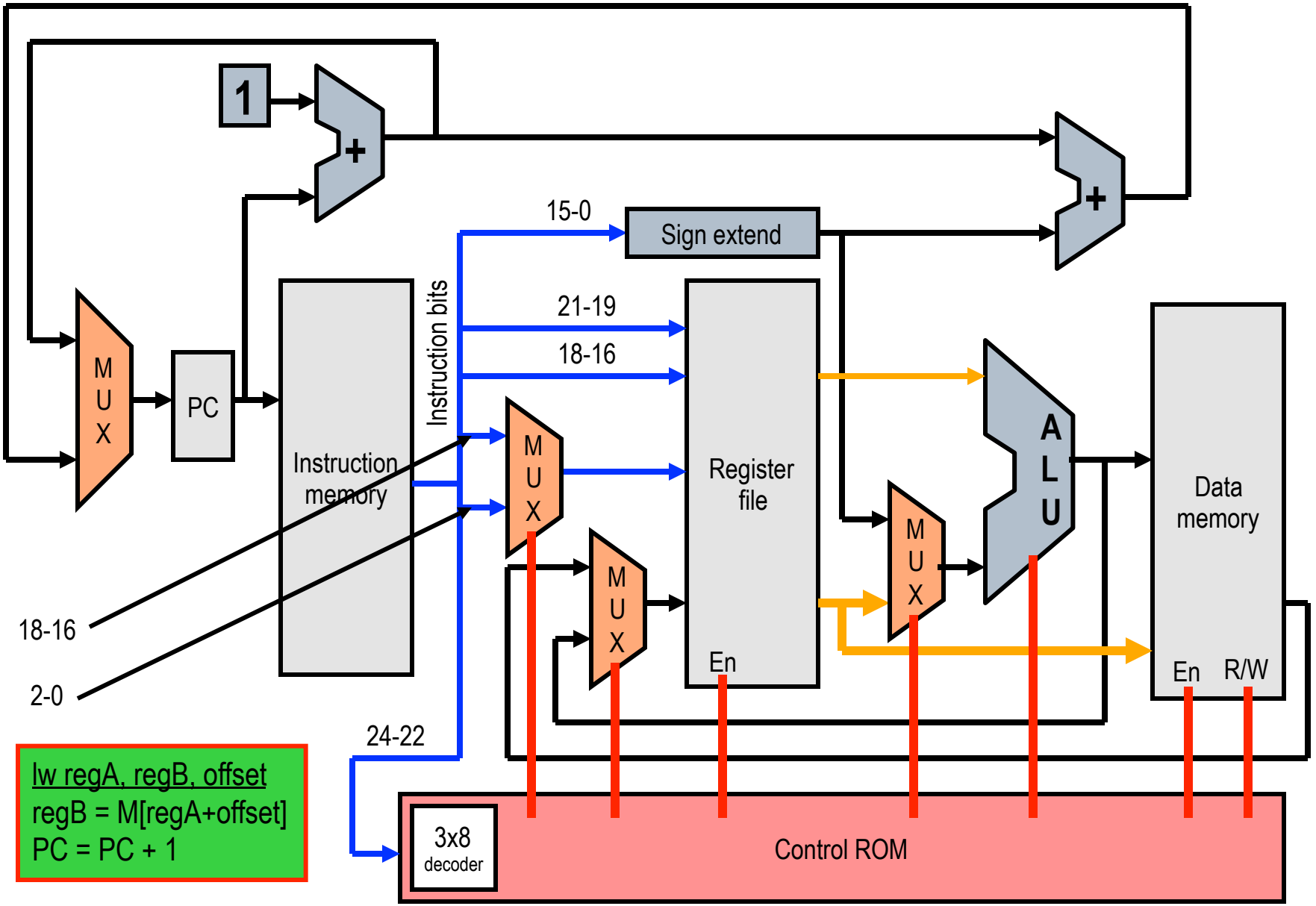
# Executing a NAND Instruction



# Executing NAND Instruction on LC2Kx Datapath

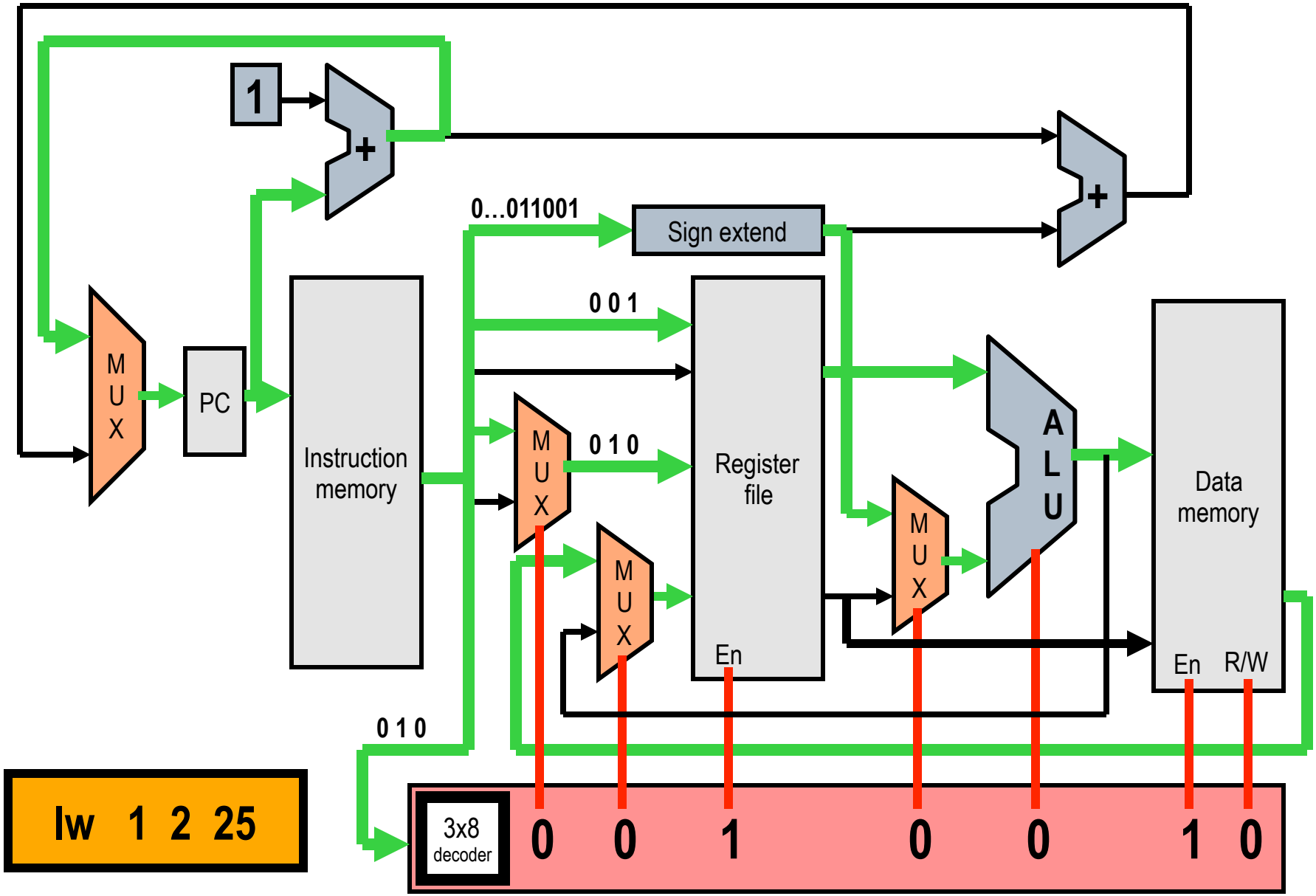


# Executing a LW Instruction

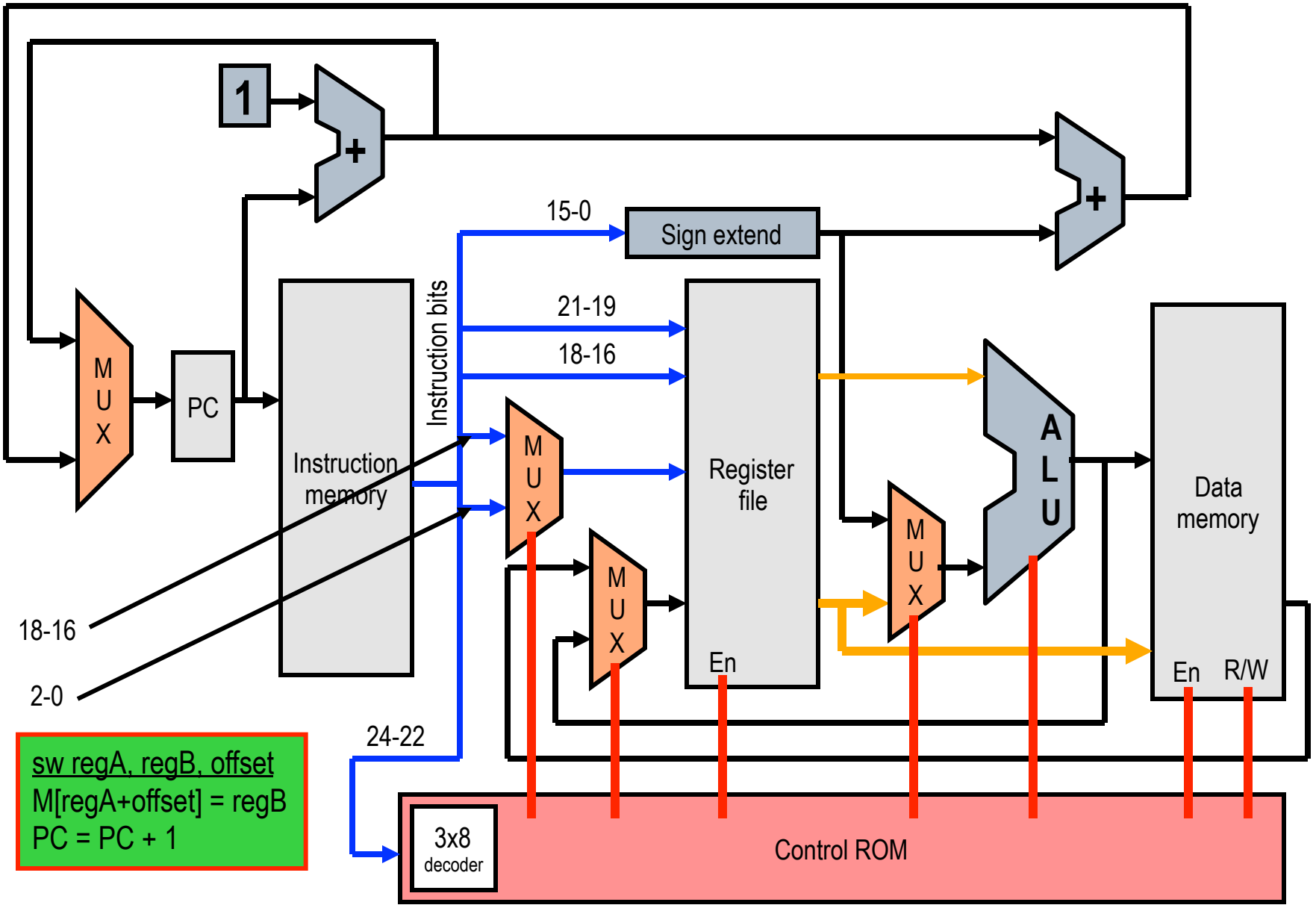



lw regA, regB, offset
regB = M[regA+offset]
PC = PC + 1

# Executing a LW Instruction on LC2Kx Datapath

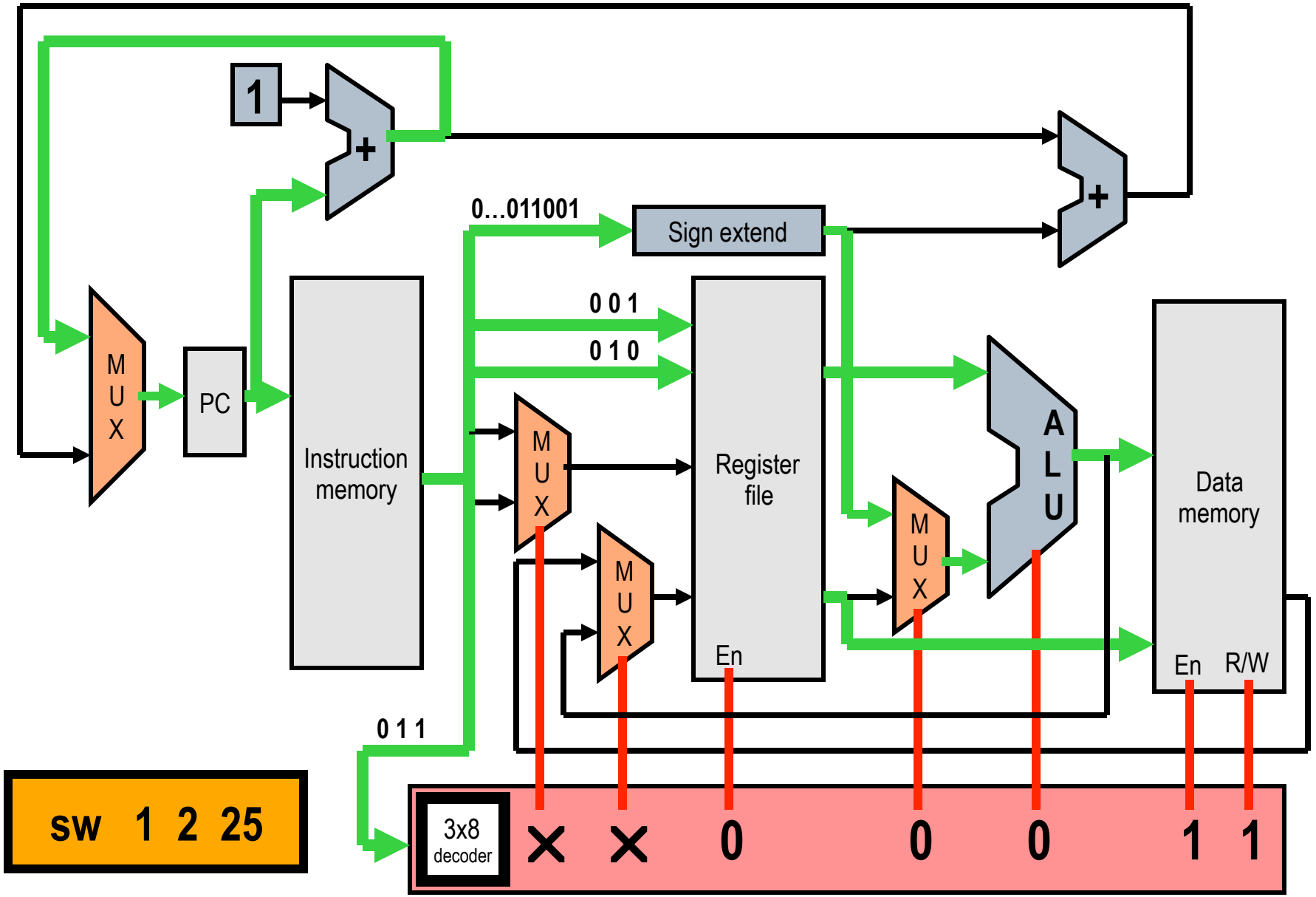


# Executing a SW Instruction

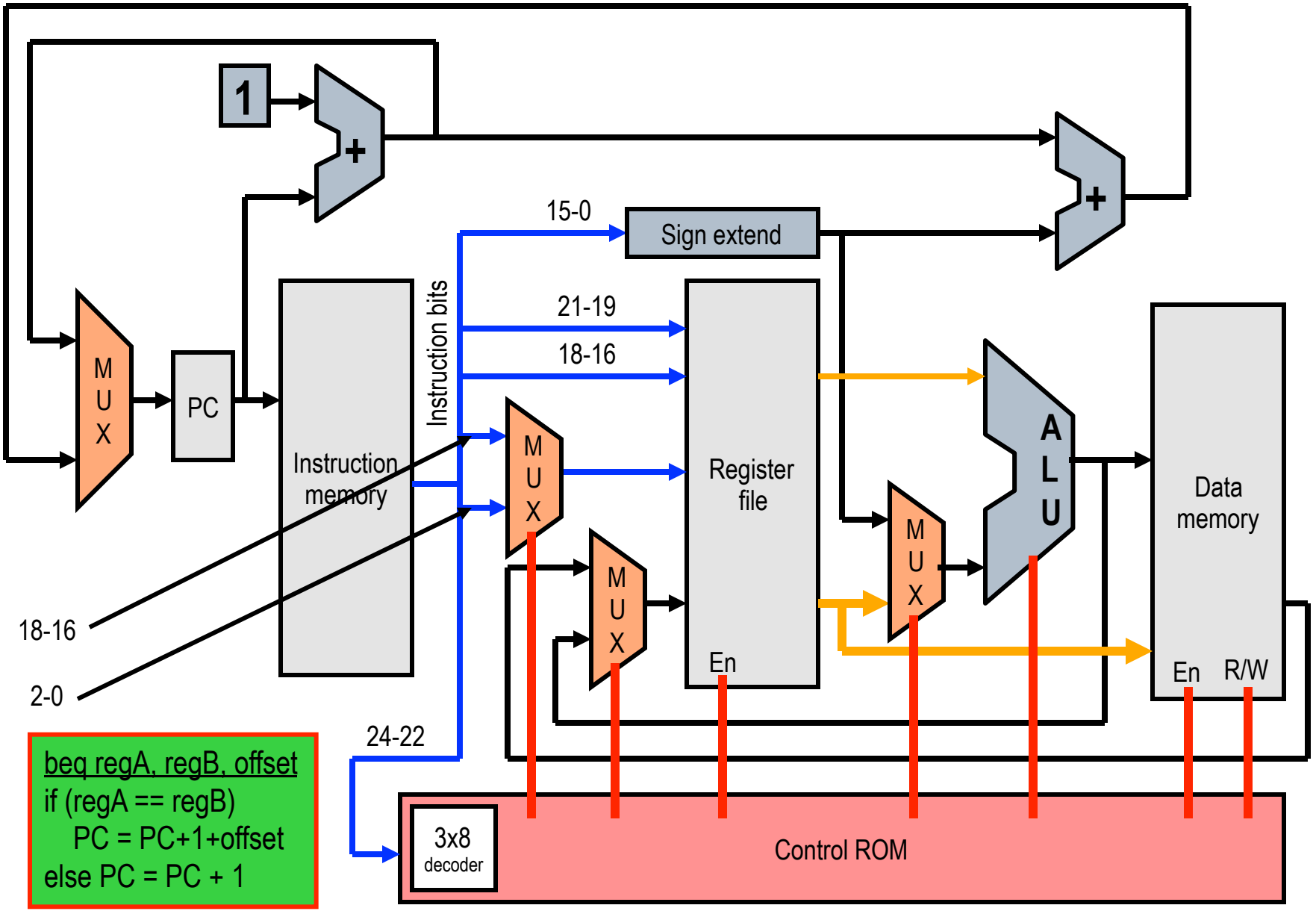



sw regA, regB, offset
M[regA+offset] = regB
PC = PC + 1

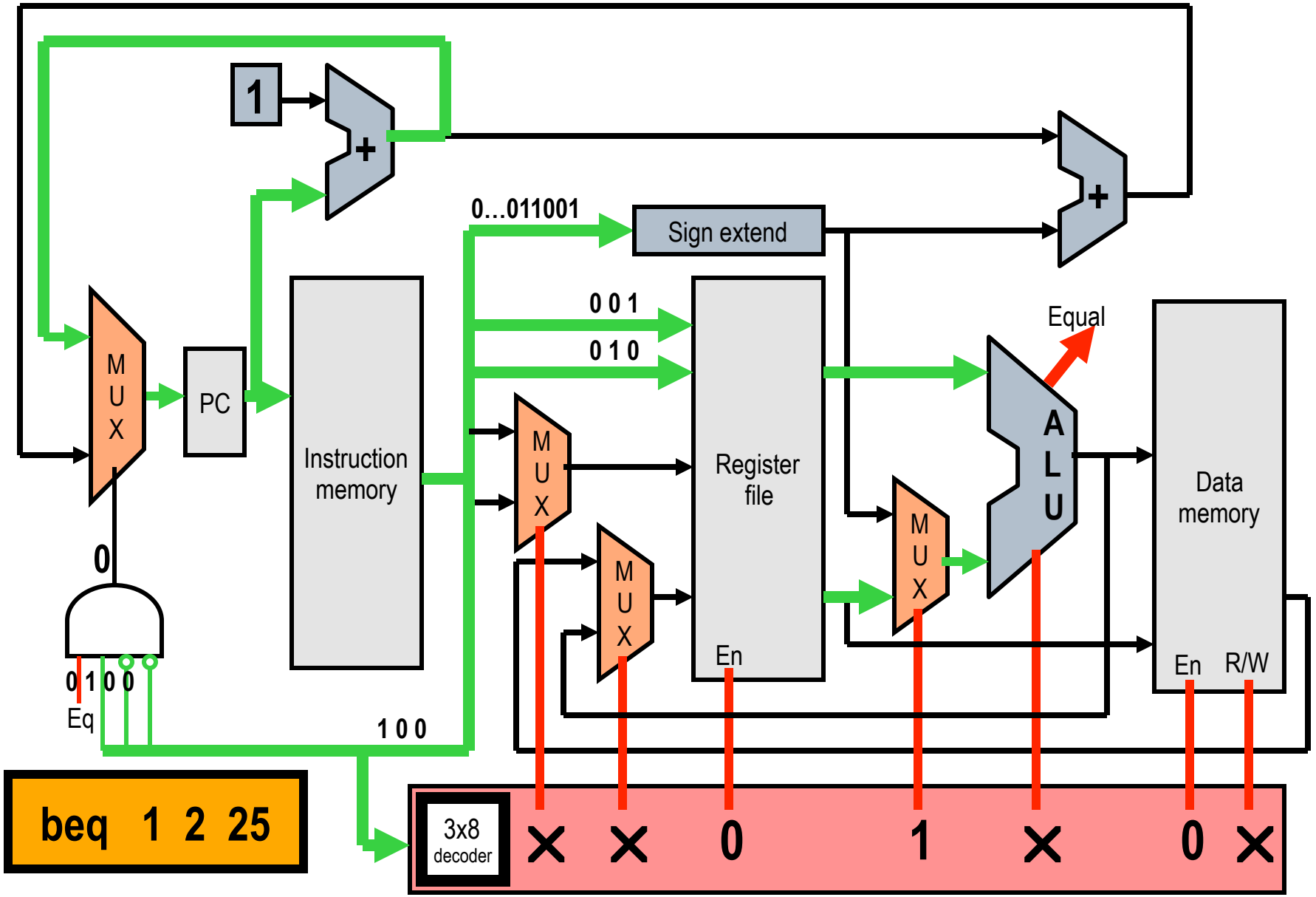
# Executing a SW Instruction on LC2Kx Datapath



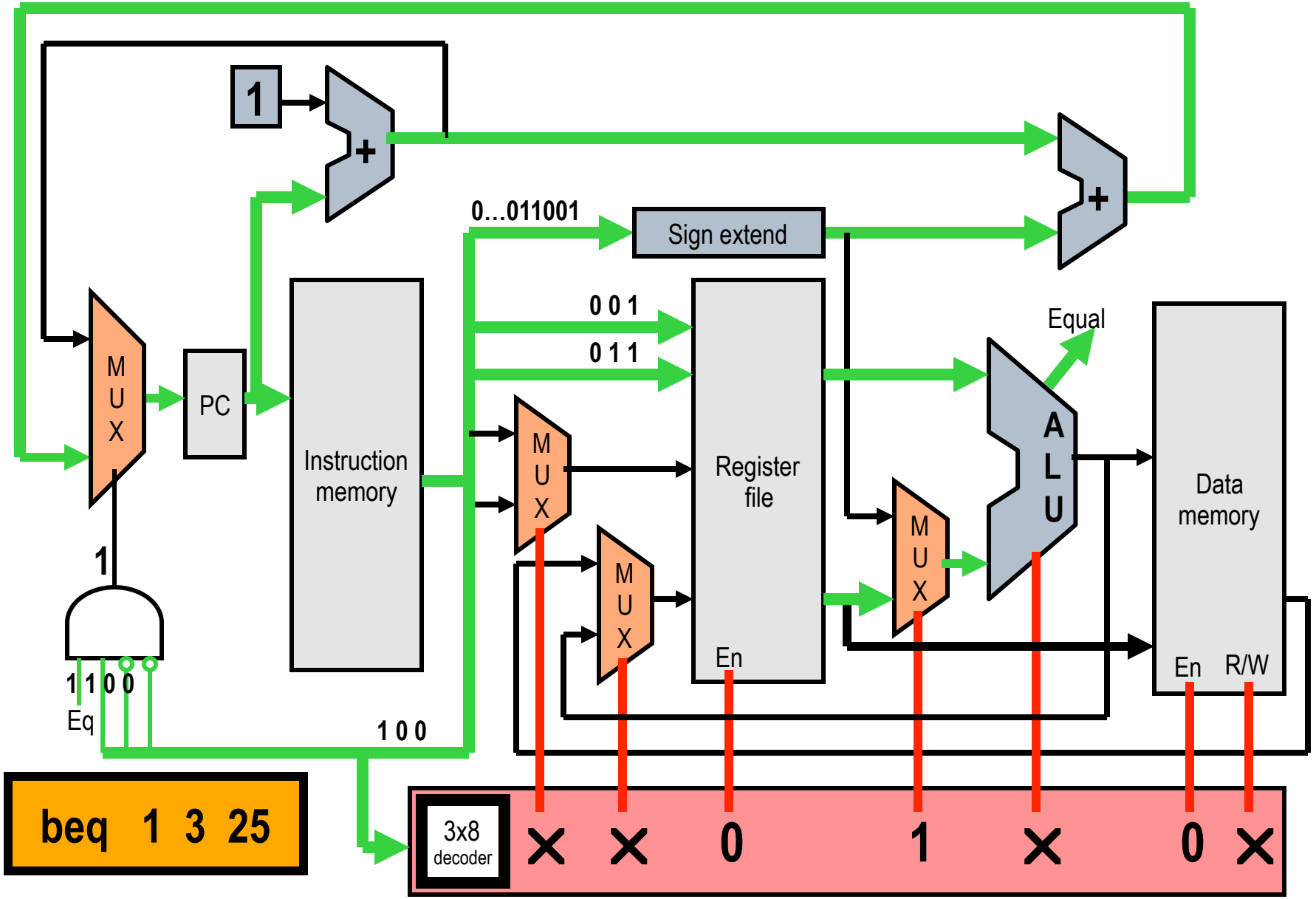
# Executing a BEQ Instruction



# Executing a "not taken" BEQ Instruction on LC2K Datapath



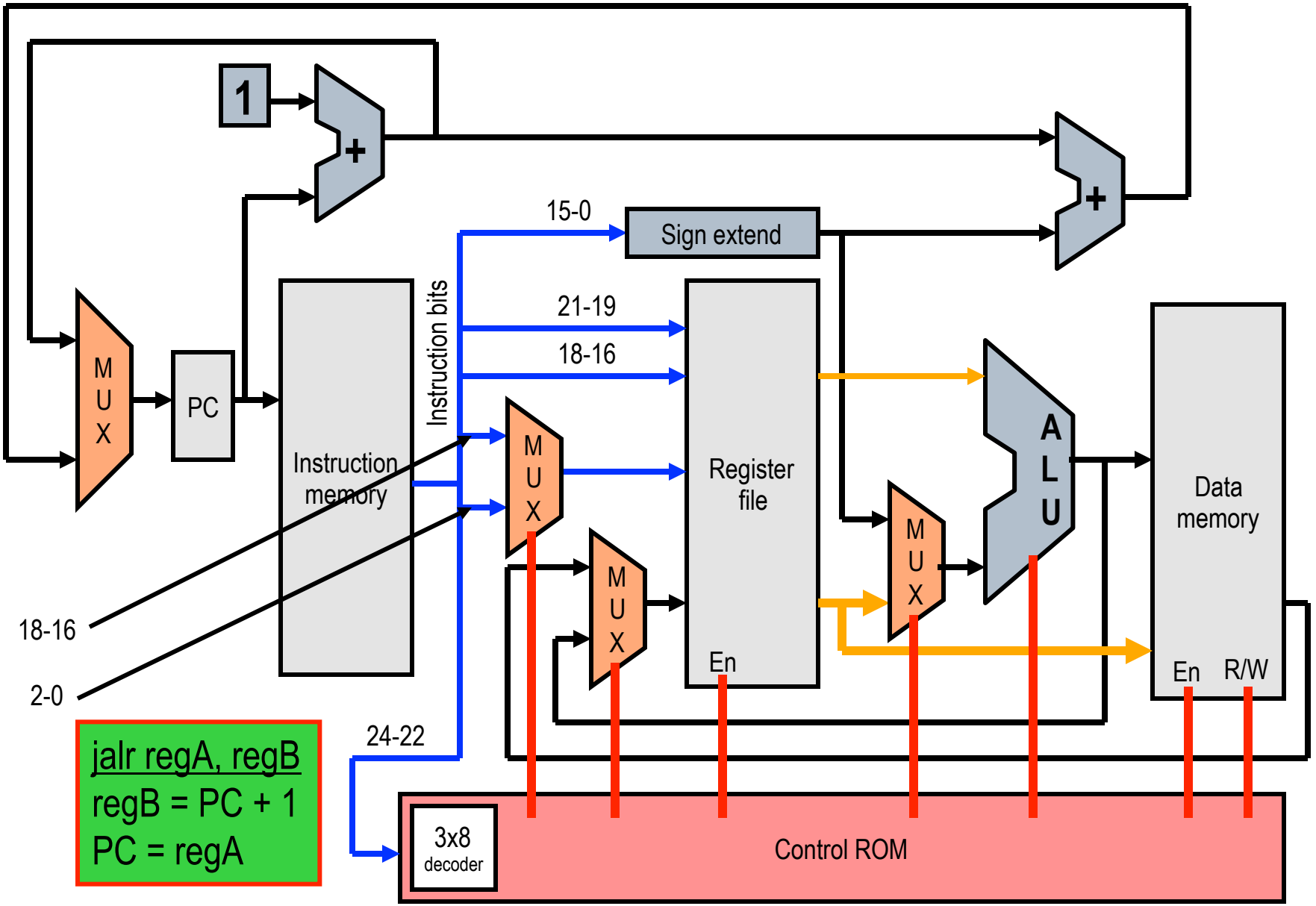
# Executing a "taken" BEQ Instruction on LC2K Datapath



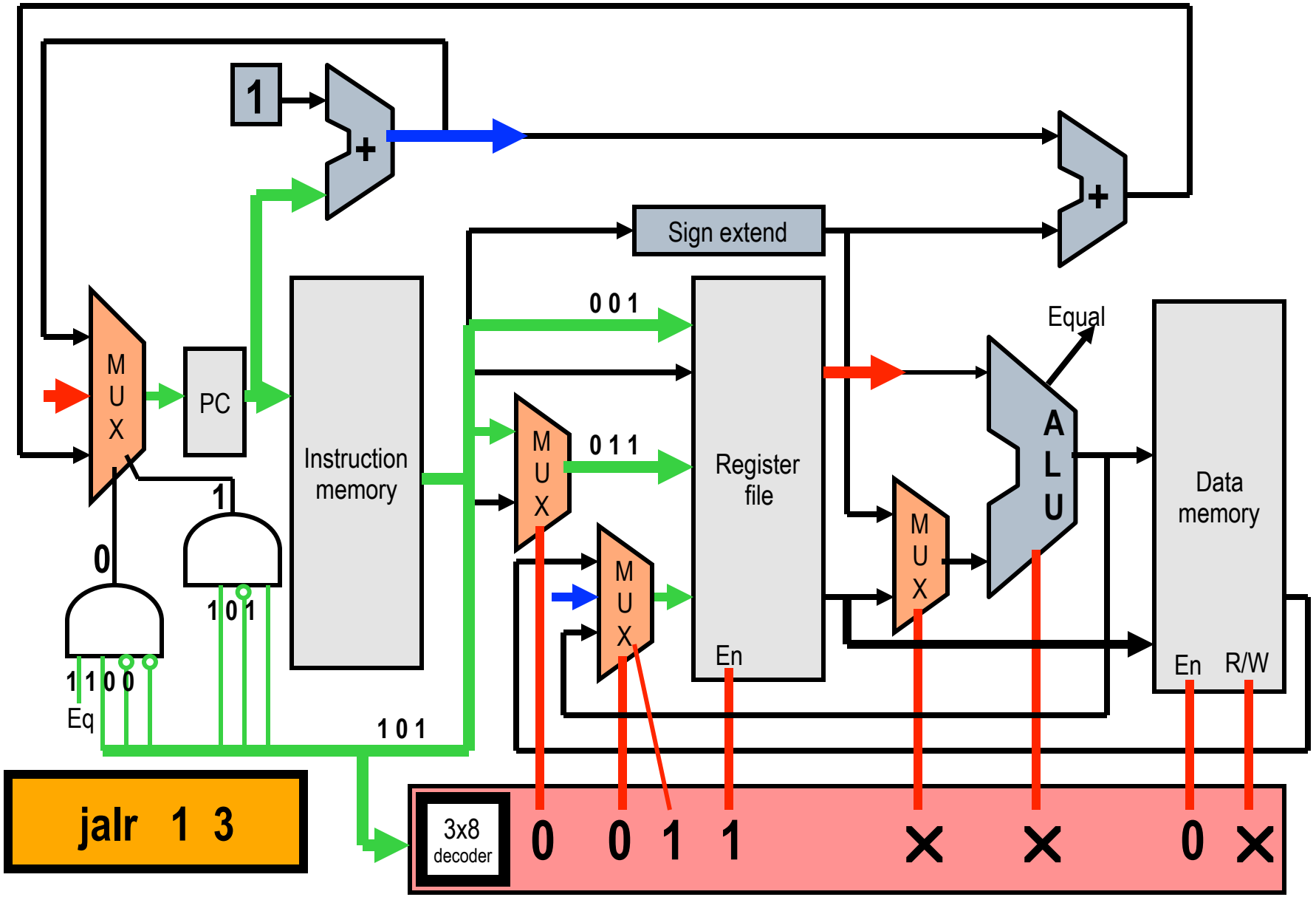
# So Far, So Good

- 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

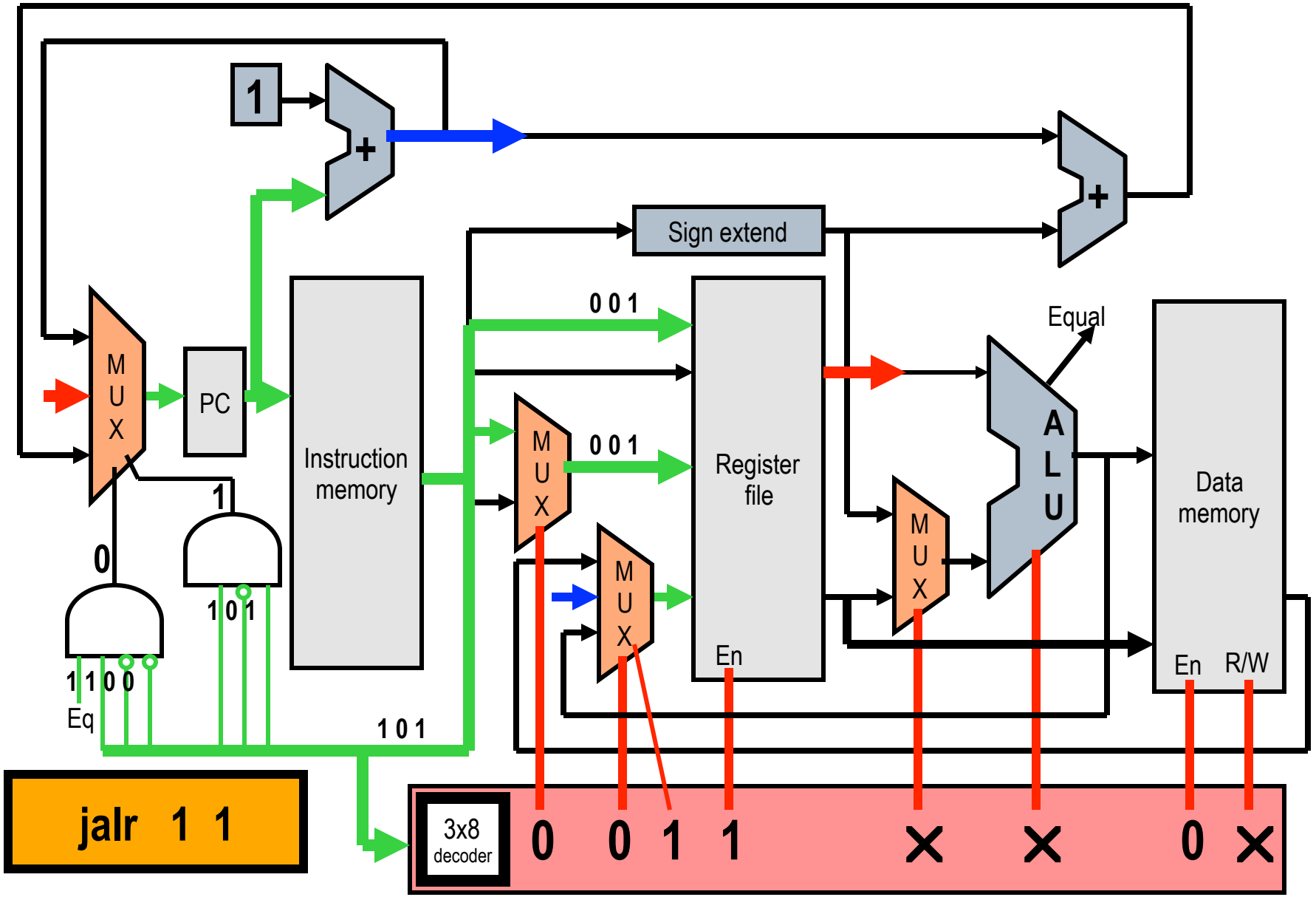
# Executing a JALR Instruction



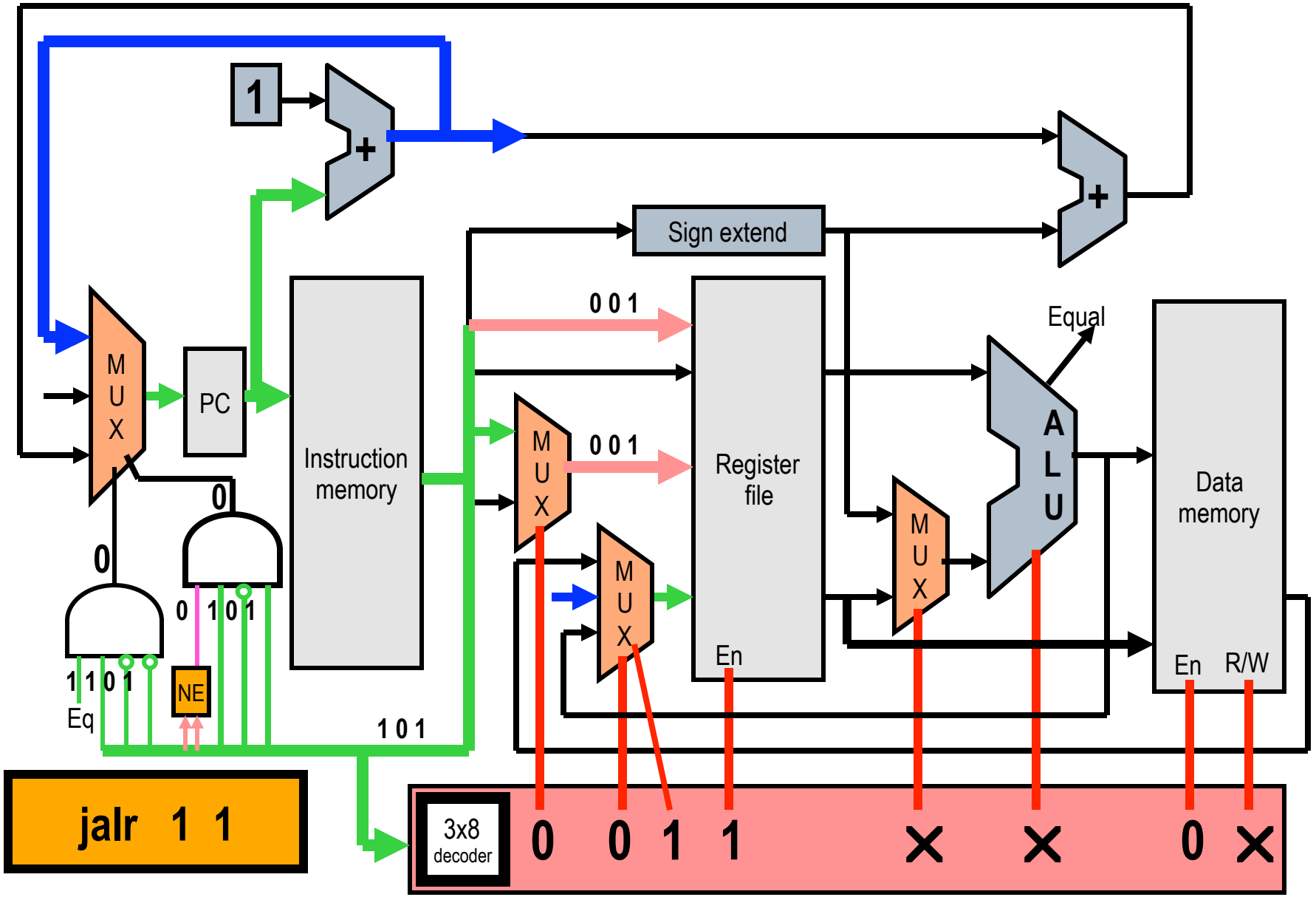
# Executing a JALR Instruction on LC2Kx Datapath



# What If regA = regB for a JALR ?



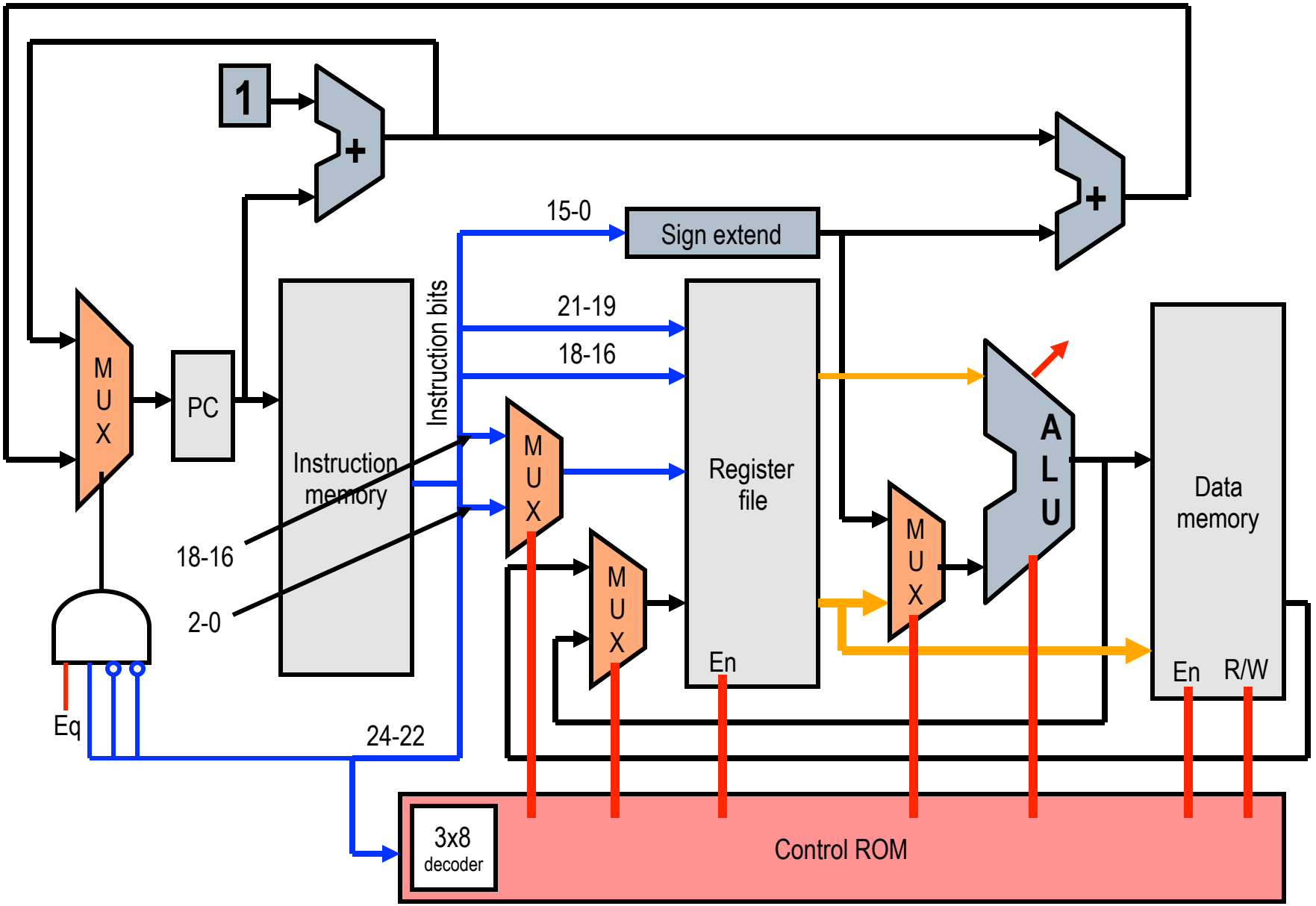
# Changes for a JALR 1 1 Instruction



# 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)



# 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 | | | + | 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:

  = ??

- Optimal execution:

  = ??

**END OF EXAM1 MATERIAL**