L10 – FSM Implementation and Single Cycle

EECS 370 – Introduction to Computer Organization – Winter 2022

Austin, Biernacki, Das, Sample, Tang

Admin

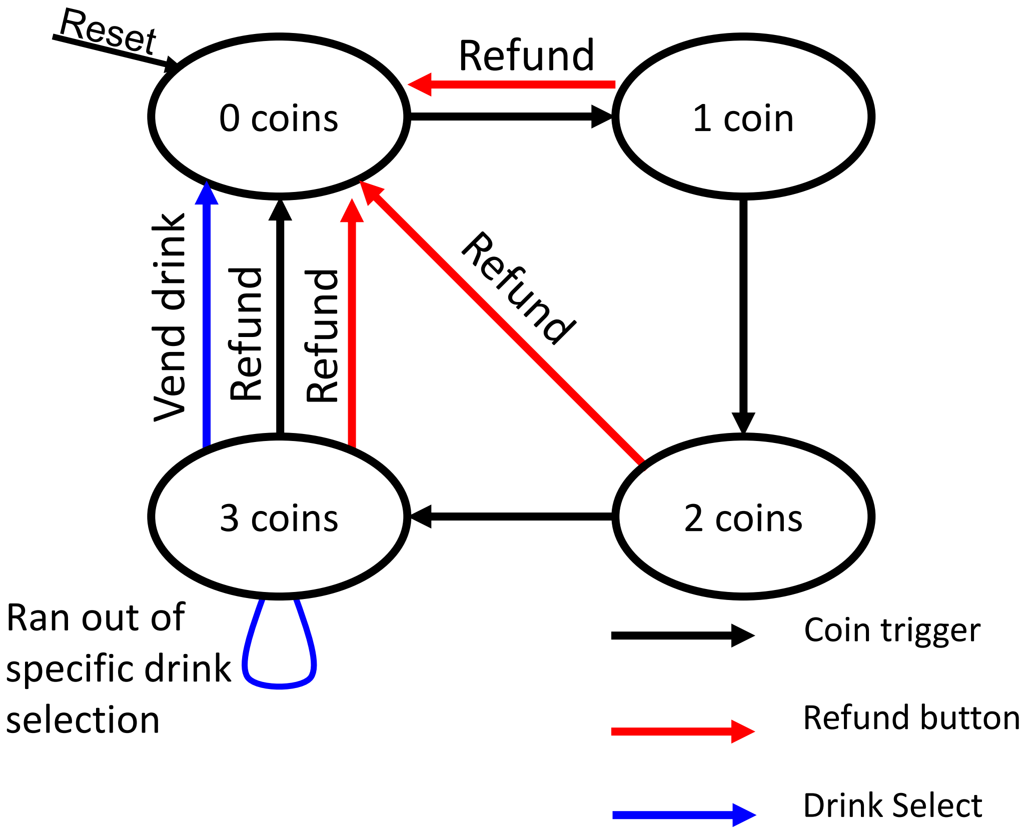
- Project 2
  - 2A due *this* Thursday (2/10)
  - 2L due *next* Thursday (2/17)

Learning Objectives

- Be able to identify the components and trade-offs relevant to a finite state machine.
- Identify the course-granularity operation of the implementation for an FSM.

FSM Review

FSM for Vending Machine

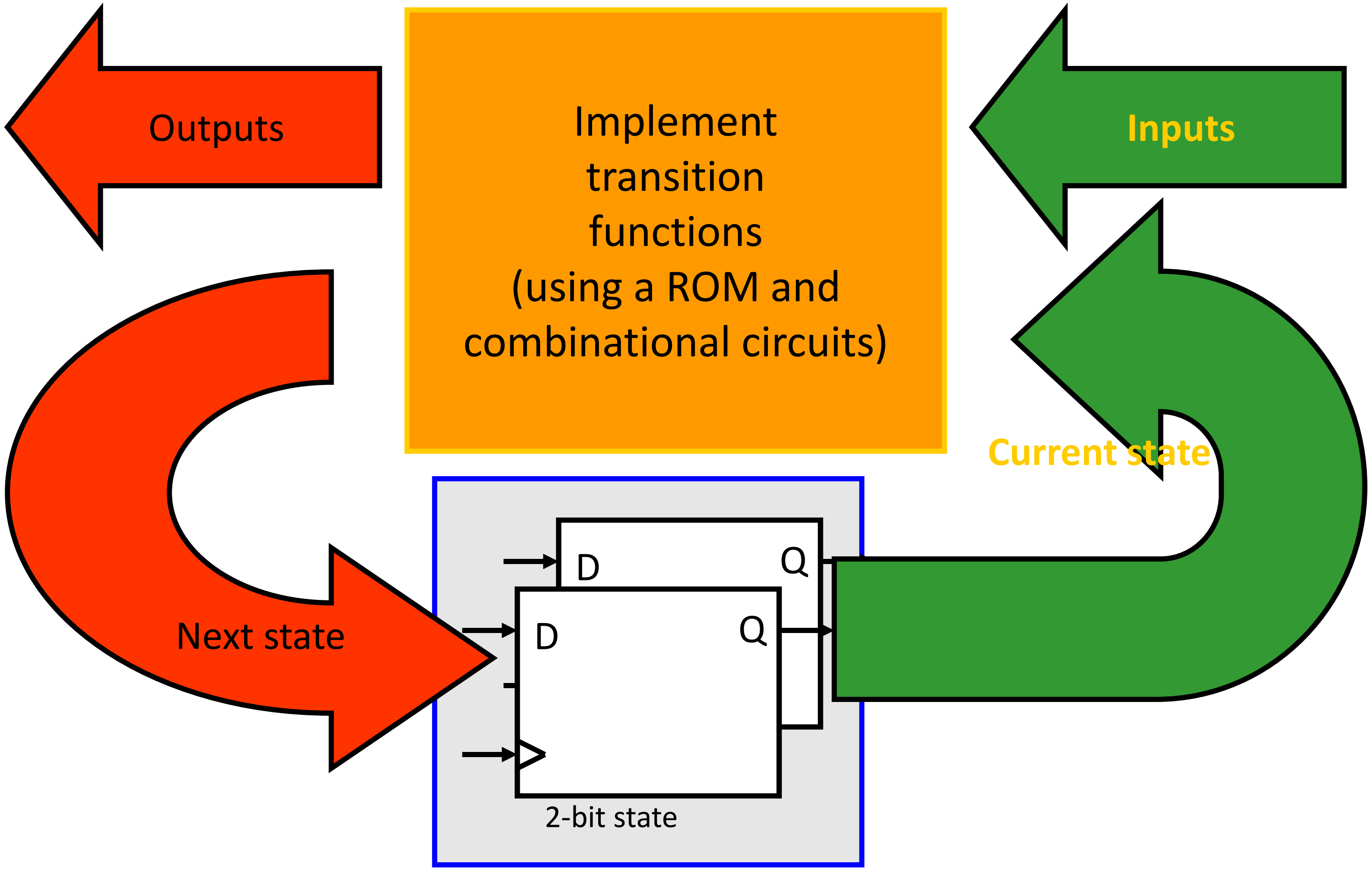


Is this a Mealy or Moore Machine?

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

Implementing a FSM

FSM Review



Implement transition functions (using a ROM and combinational circuits)

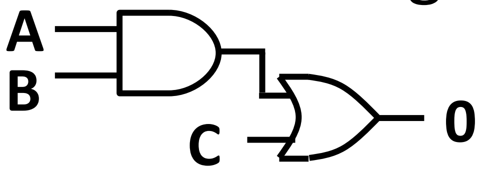
FSM

Implementing Combinational Logic (1)

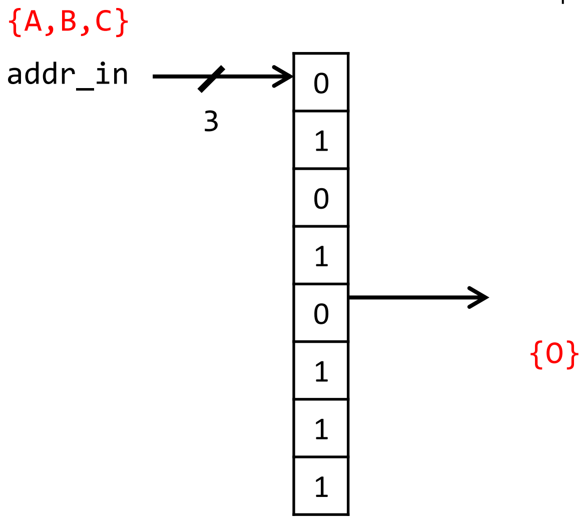
- If I have a truth table:

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

- I can either implement this using combinational logic:



- ...or I could literally just store the entire truth table in a memory and just "address" it using the input!



FSM

Implementing Combinational Logic (2)

- 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!)

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

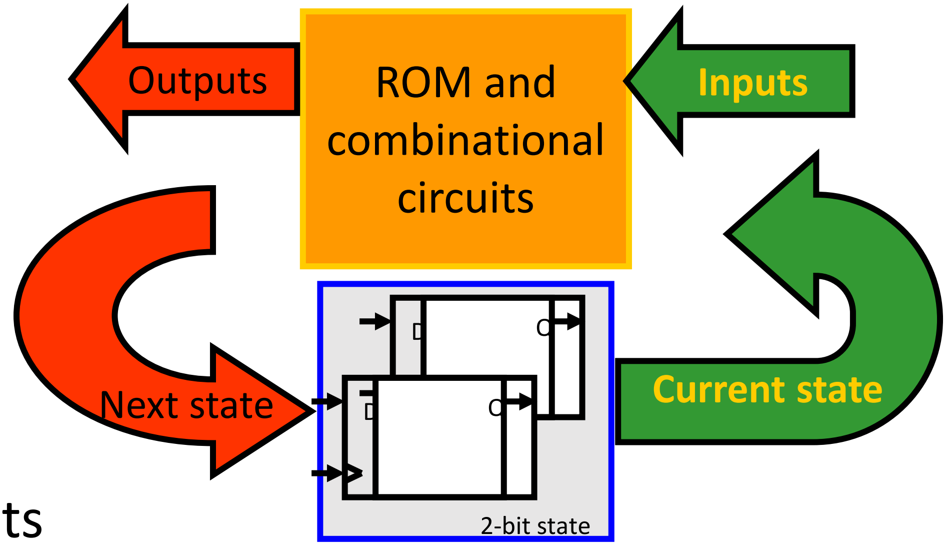
Add one more input…

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

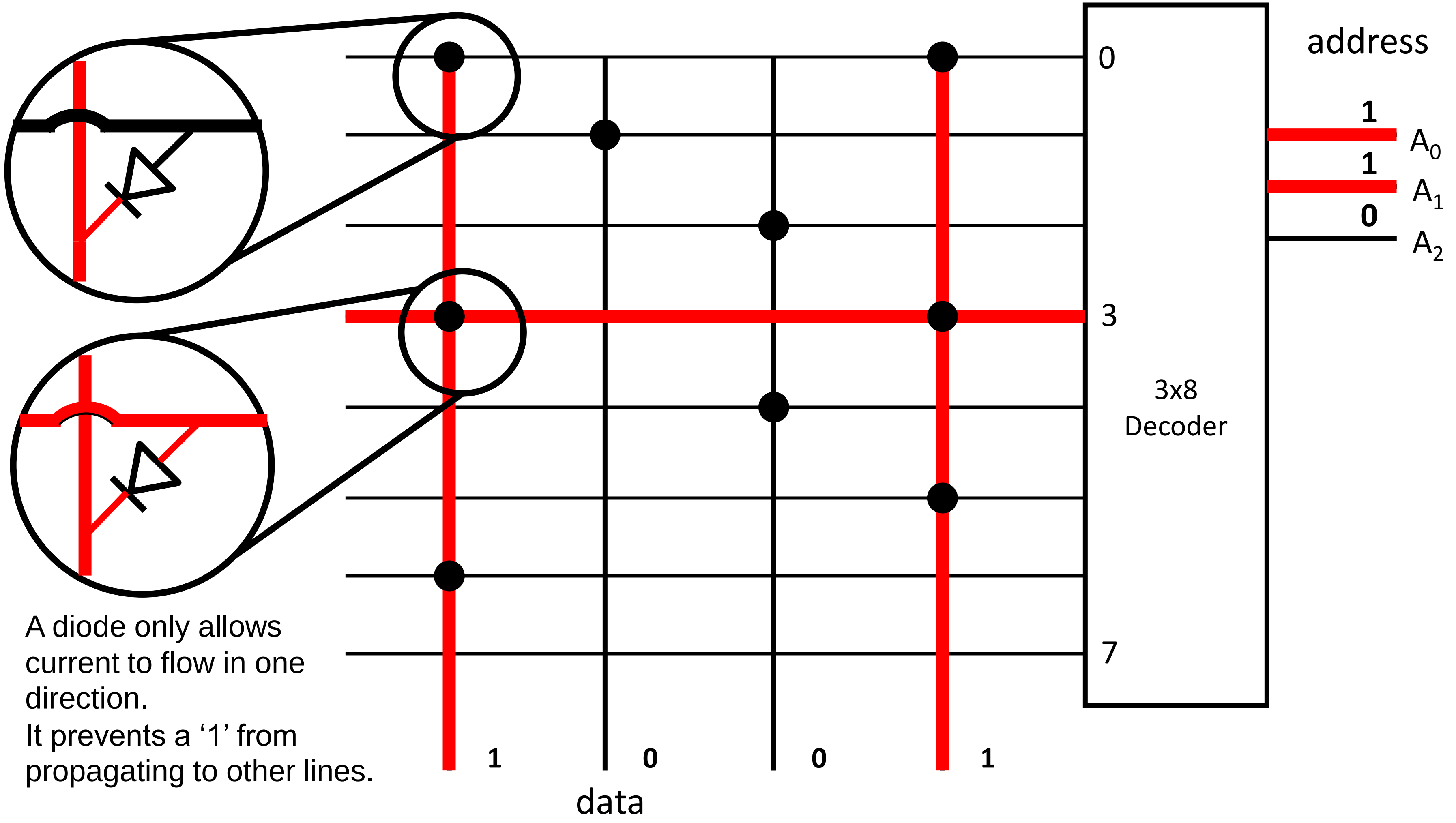
FSM

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



8-entry 4-bit ROM



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

FSM

ROM for Vending Machine Controller

- Use current state and inputs as address
  - 2 state bits + 22 inputs = 24 bits (address)
  - Coin, refund, 10 drink selectors, 10 sensors

- Read next state and outputs from ROM
  - 2 state bits + 11 outputs = 13 bit (memory)
  - Refund release, 10 drink latches

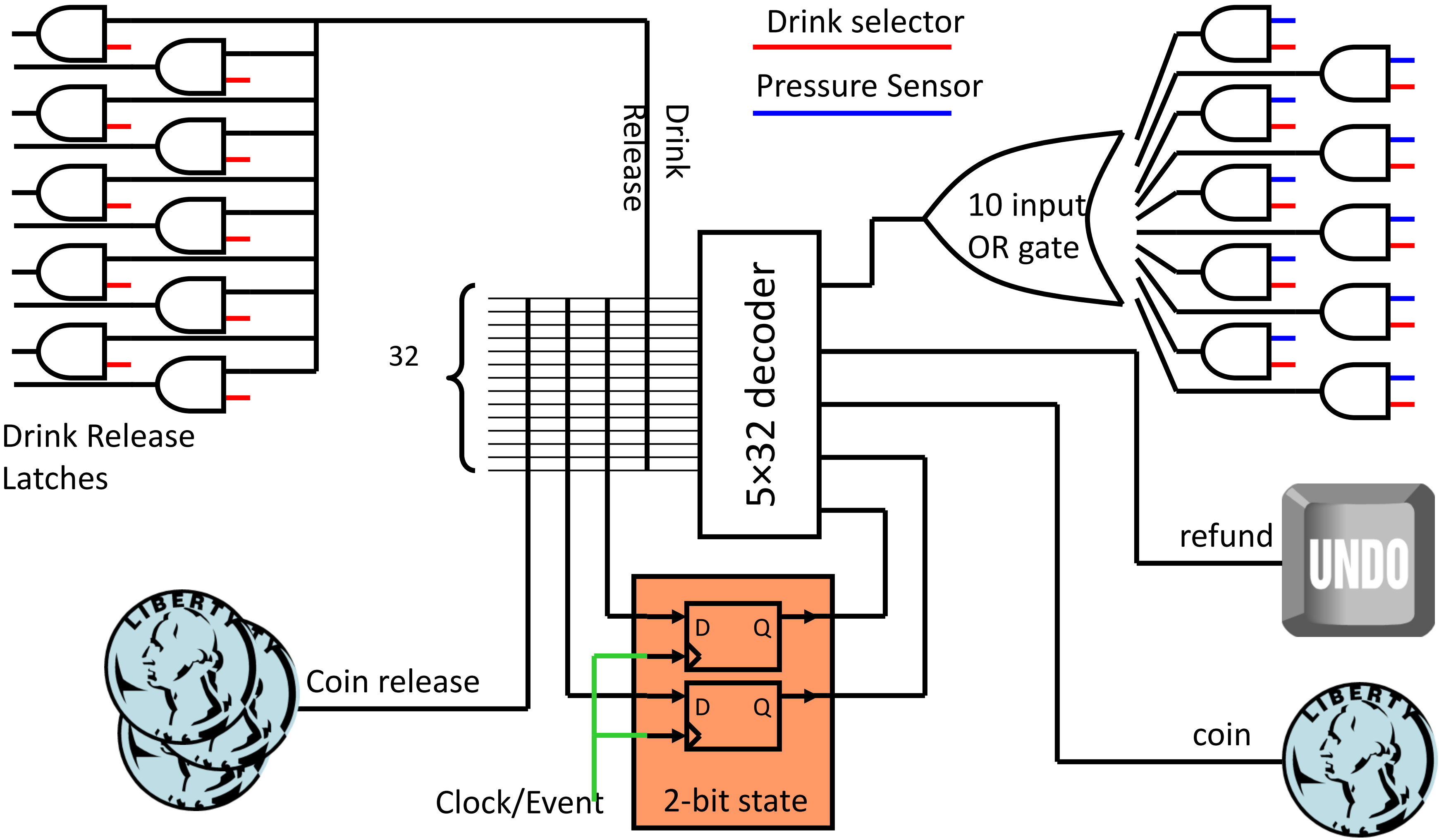
- We need 2^{24} entry, 13 bit ROM memories
  - 218,103,808 bits of ROM seems excessive for our cheap controller

Reducing the ROM Needed

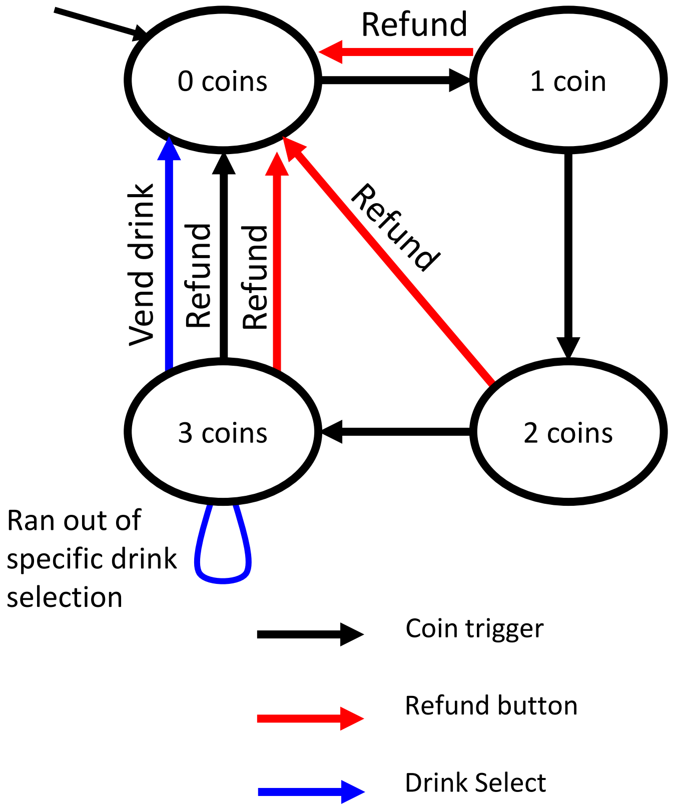
- Idea: let's do a hybrid between combinational logic and a lookup table
  - Use basic hardware (AND / OR) gates where we can, and a ROM for everything more complicated
- Replace 10 selector inputs and 10 pressure inputs with a single bit input (drink selected)
  - Use drink selection input to specify 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)
- Now:
  - 2 current state bits + 3 input bits (5 bit ROM address)
  - 2 next state bits + 2 control trigger bits (4 bit memory)
  - 32 × 4 = 128 bit ROM (good!)

Putting It All Together

FSM



Some of the ROM Contents



Current state		Coin trigger	Drink select	Refund button
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	0	1	0
1	1	1	0	0
… 24 more entries				

ROM address (current state, inputs)

Next state		Coin release	Drink release
0	0	0	0
0	0	0	0
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	0	1
0	0	1	0
… 24 more entries			

ROM contents (next state, outputs)

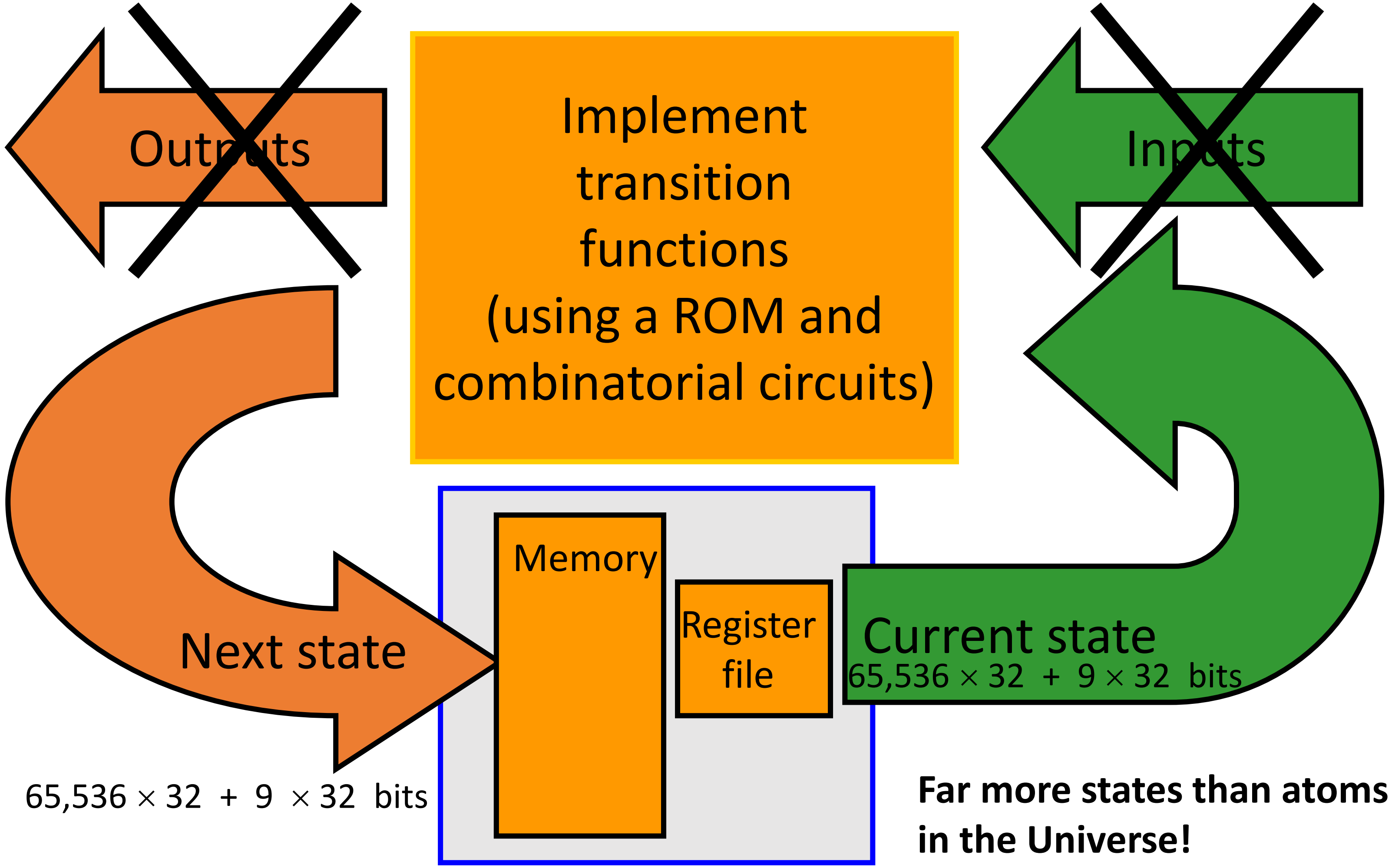
FSM

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 software anticipating a variable price, perhaps no change is even needed
- Next Topic - Our first processor!

LC2Kx Processor as FSM

FSM



Outputs
Implement transition functions (using a ROM and combinatorial circuits)
Inputs
Memory
Register file
Next state
Current state
65,536 × 32 + 9 × 32 bits
65,536 × 32 + 9 × 32 bits

Far more states than atoms in the Universe!

L10_2 Single-Cycle-Processor

EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- To identify the components used to implement a processor for LC-2K and understand the mapping from these components to LC-2K instructions.

Single-Cycle Processor Design

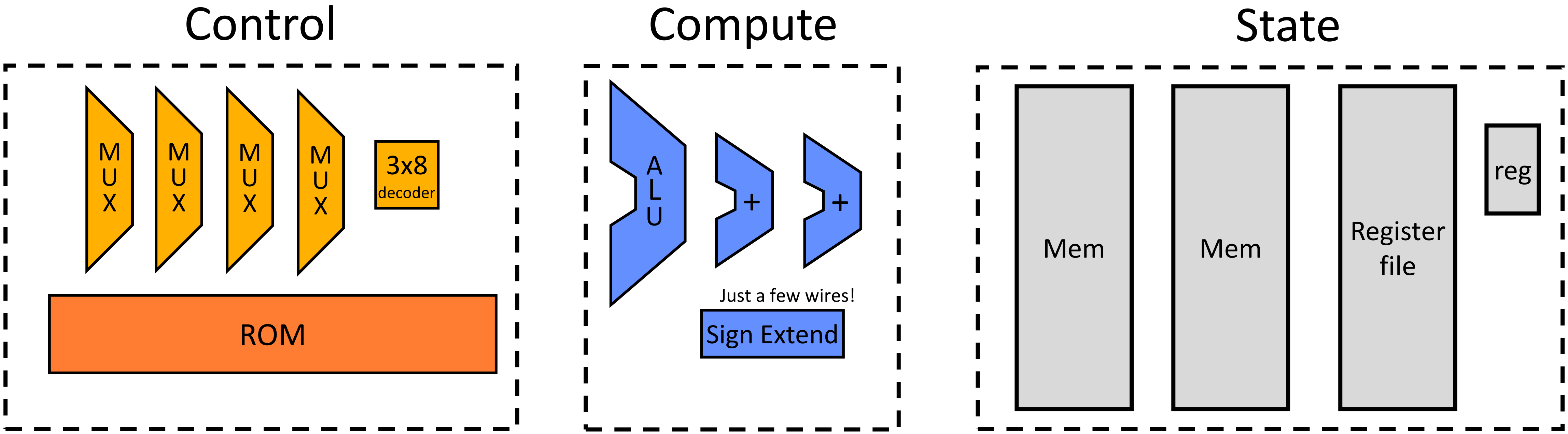
Single-Cycle

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

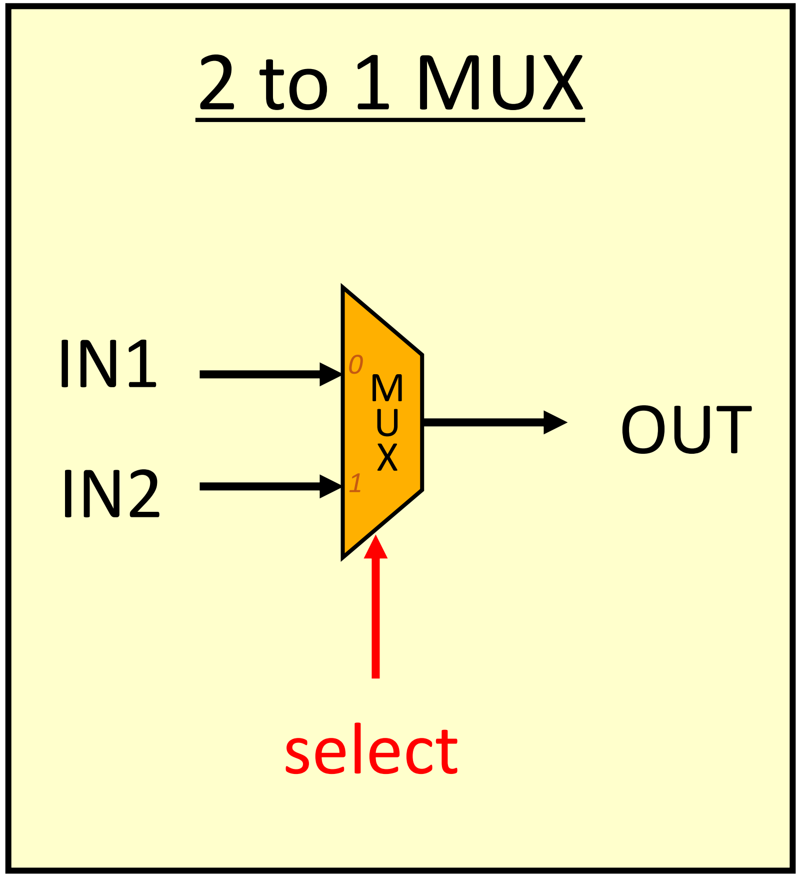
Building Blocks for the LC2K

Single-Cycle



Building Blocks for the LC2K: **Control**

Single-Cycle



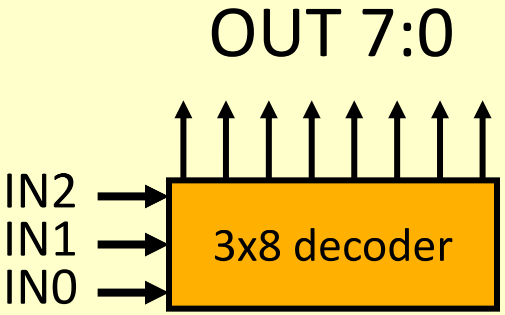
Connect one of the inputs to OUT based on the value of select

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

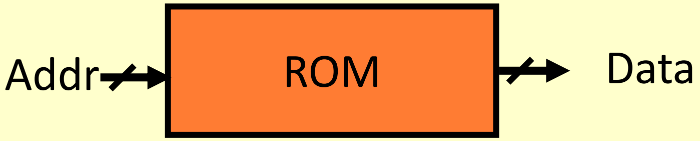
Single-Cycle

Building Blocks for the LC2K: **Control**

3 to 8 decoder



Read-only Memory



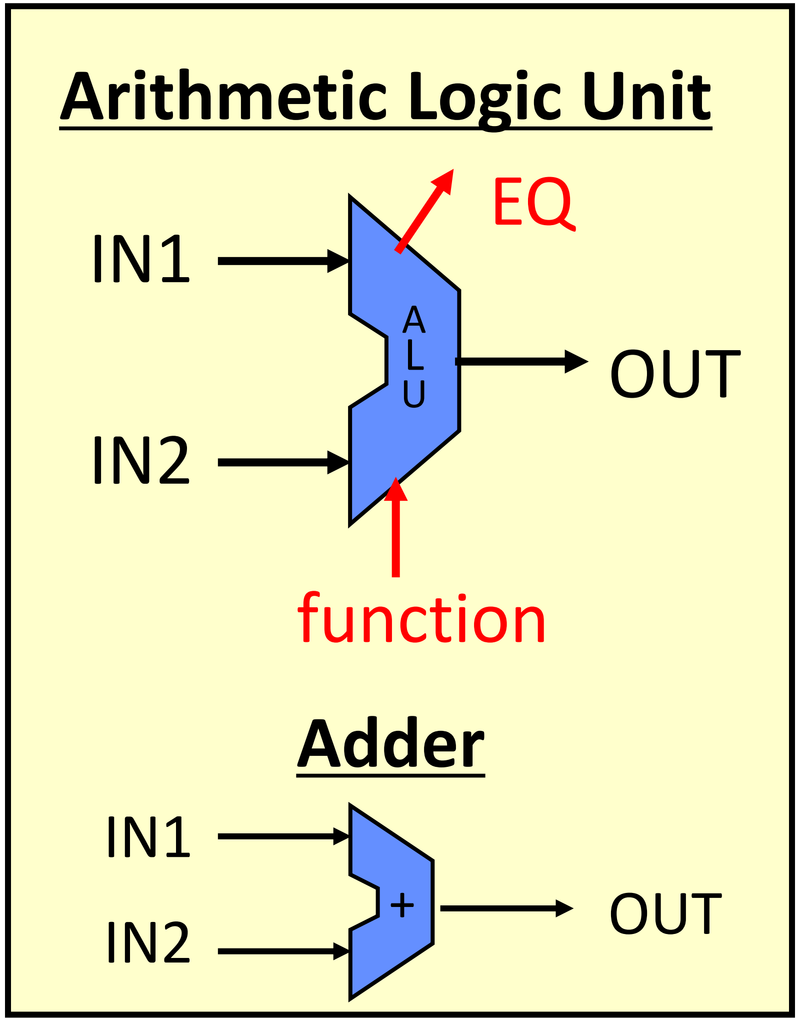
Decoder activates one of the output lines based on the input

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

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

Building Blocks for the LC2K: **Compute**

Single-Cycle



Perform basic arithmetic functions

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

For LC2K:

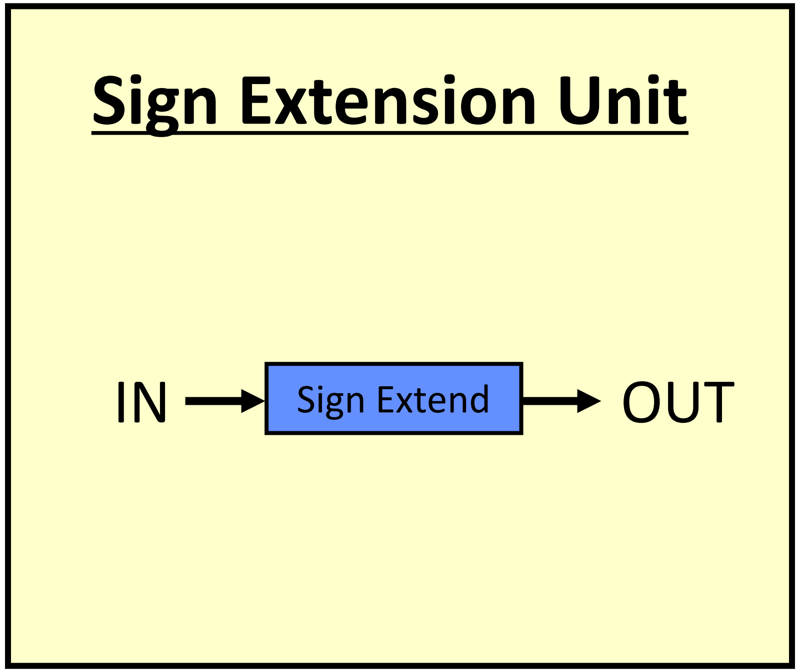
```
        f=0 is add
        f=1 is nor
```

For other processors, there are many more functions.

Just adds

Building Blocks for the LC2K: **Compute**

Single-Cycle



Sign Extension Unit

IN → Sign Extend → OUT

Sign extend input by replicating the MSB to width of output

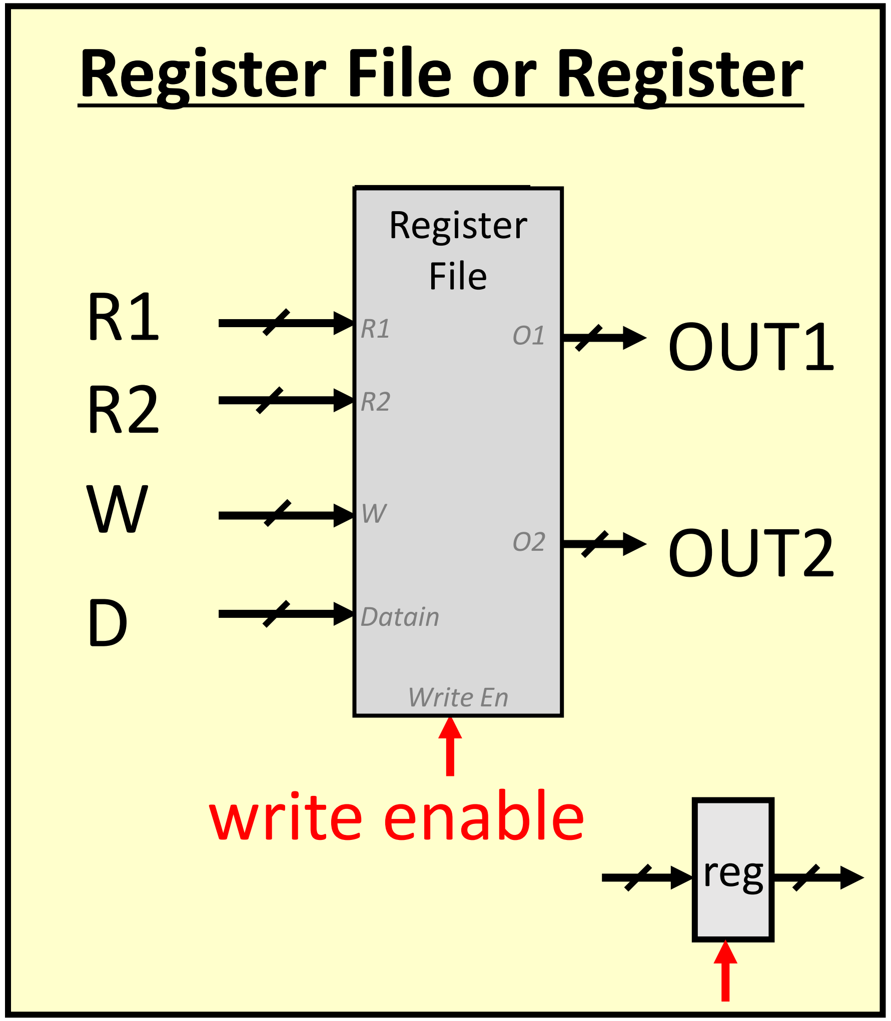
```
OUT(31:0) = SE(IN(15:0))

OUT(31:16) = IN(15)
OUT(15:0) = IN(15:0)
```

Useful when compute unit is wider than data

Building Blocks for the LC2K: **State**

Single-Cycle



Small/fast memory to store temporary values

- **n** entries (LC2 = 8)
- **r** read ports (LC2 = 2)
- **w** write ports (LC2 = 1)

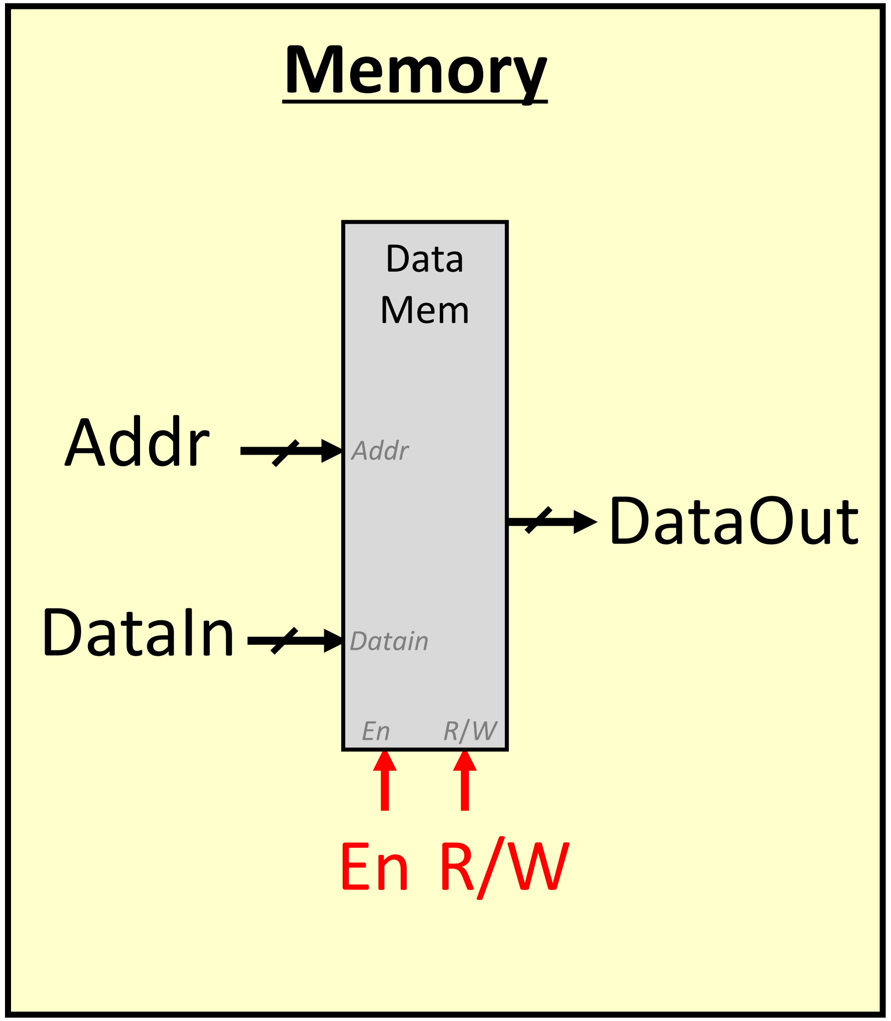
* Ri specifies register number to read

* W specifies register number to write

* D specifies data to write

Building Blocks for the LC2K: **State**

Single-Cycle



Slower storage structure to hold large amounts of stuff.

Use 2 memories for LC2

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

Review: LC2K Instruction Formats

- Tells you which bit positions mean what
- R-type instructions (opcodes `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 (opcodes `lw 010, sw 011, beq 100`)

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

L10_3 LC2K-Datapath

EECS 370 – Introduction to Computer Organization – Fall 2020

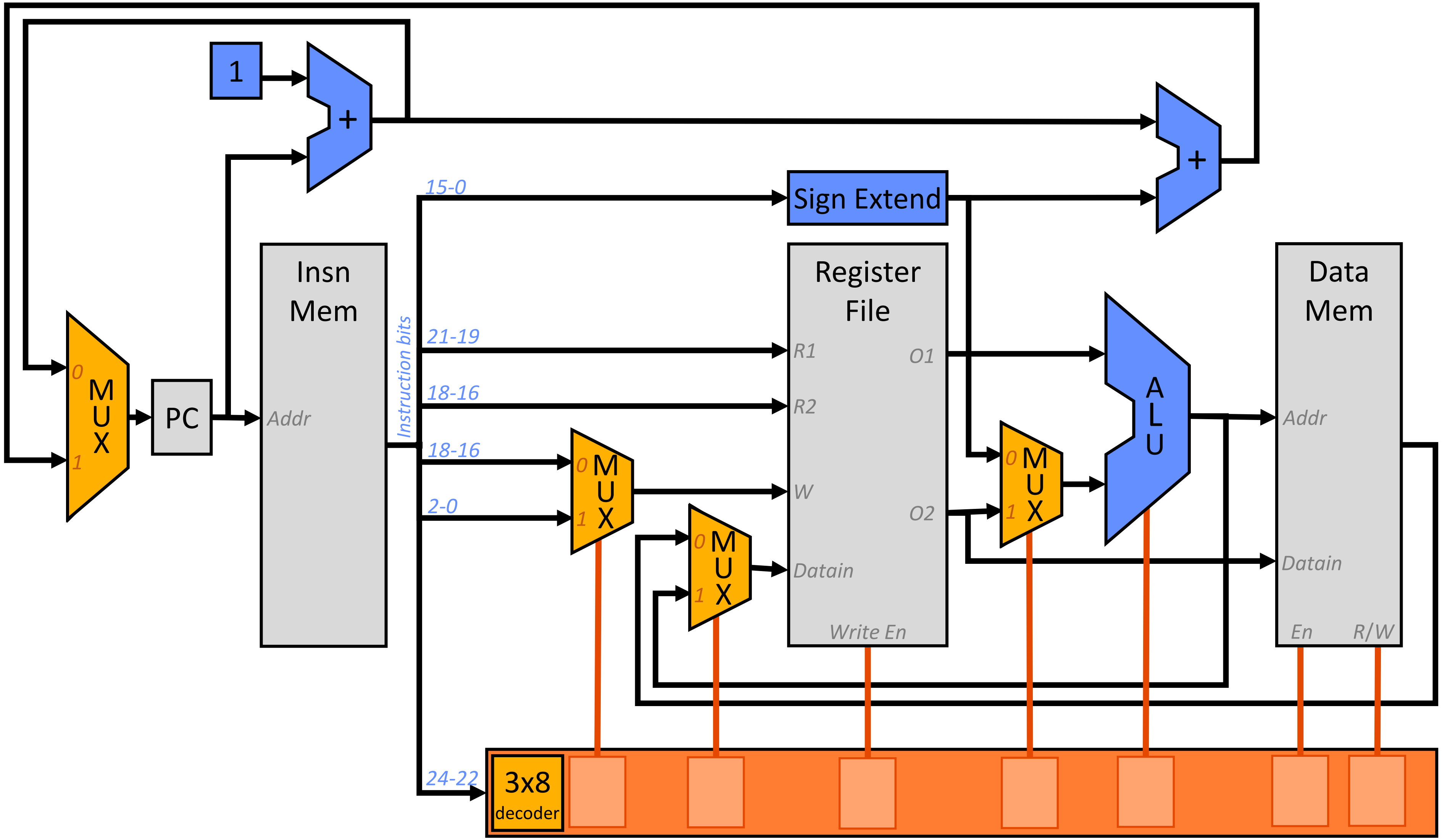
Single-Cycle Processor Design

General-Purpose Processor Design

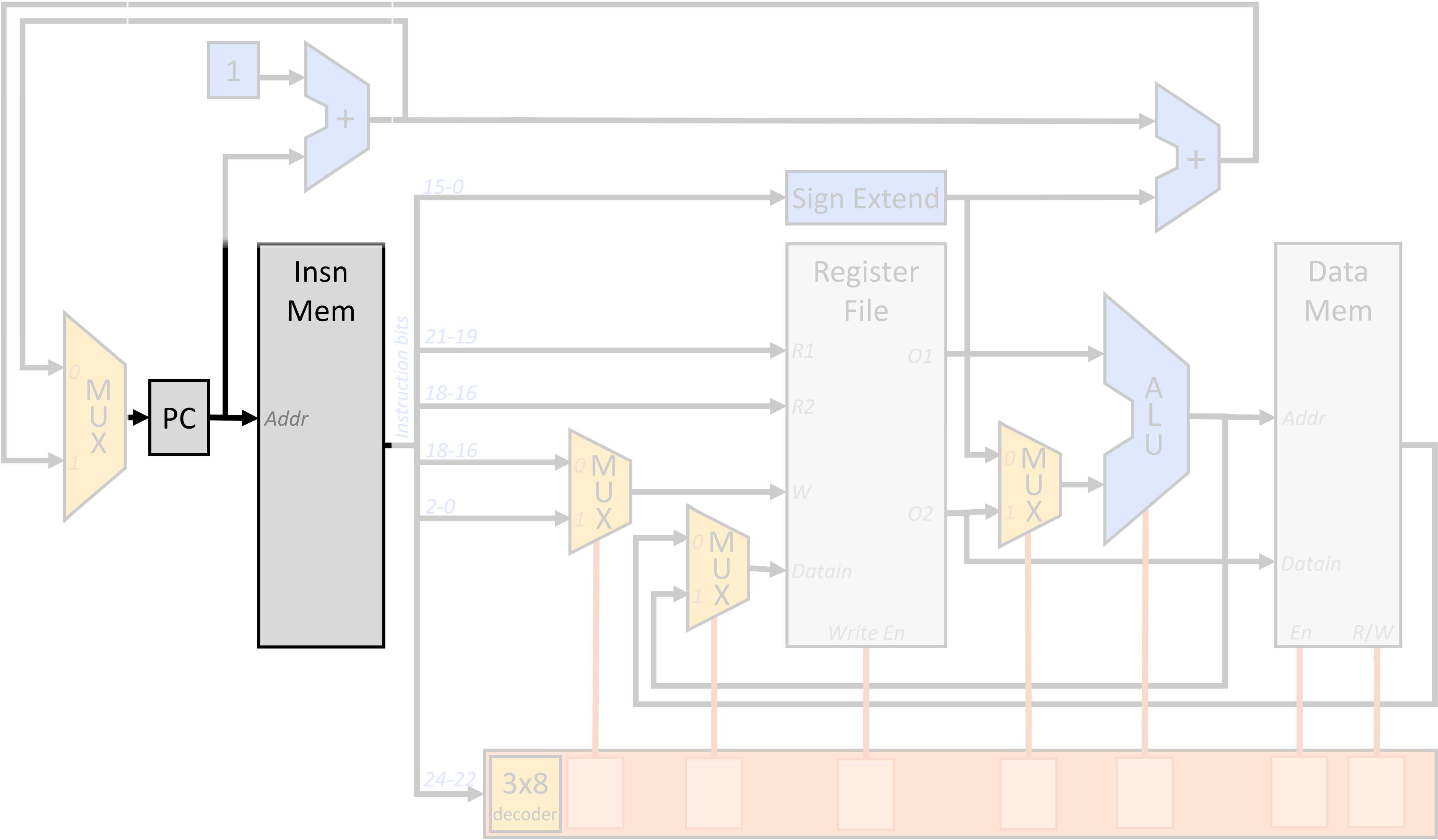
- Fetch Instructions
- Decode Instructions
    - Instructions are input to control ROM
- ROM data controls movement of data
    - Incrementing PC, reading registers, ALU control
- Clock drives it all

Single-cycle datapath:
Each instruction completes in one clock cycle

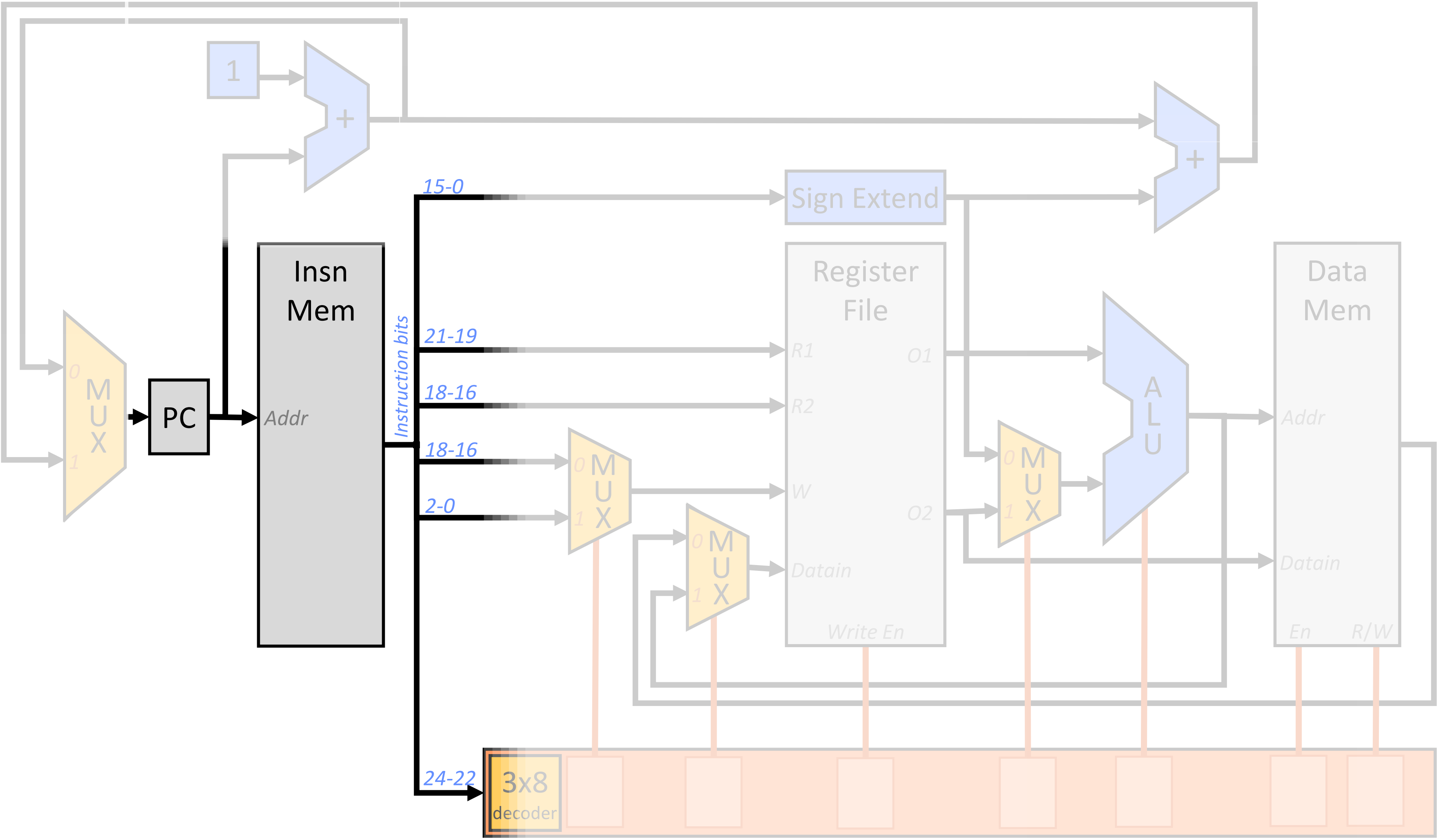
LC2K Single-Cycle Processor



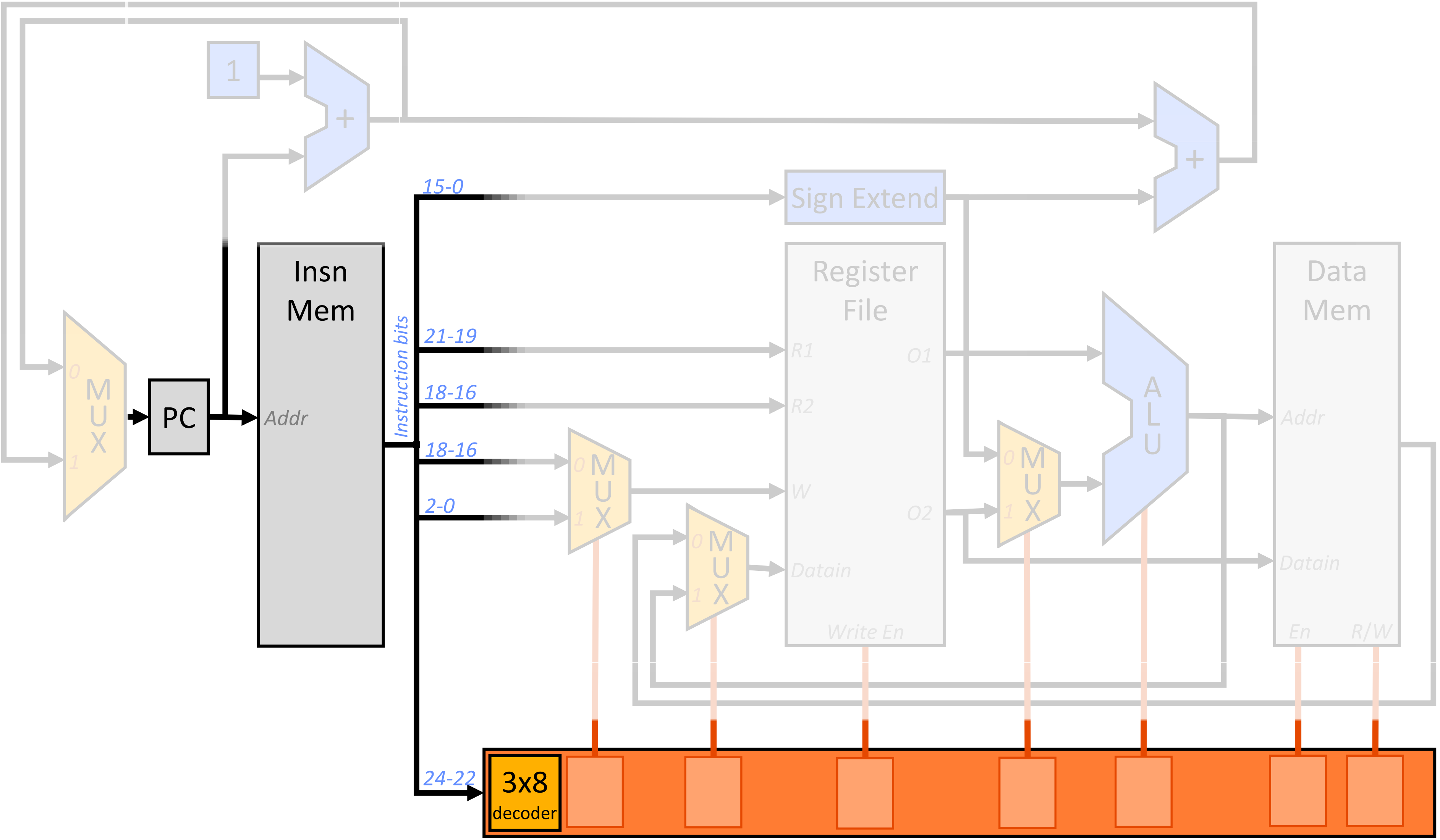
LC2K Single-Cycle Processor



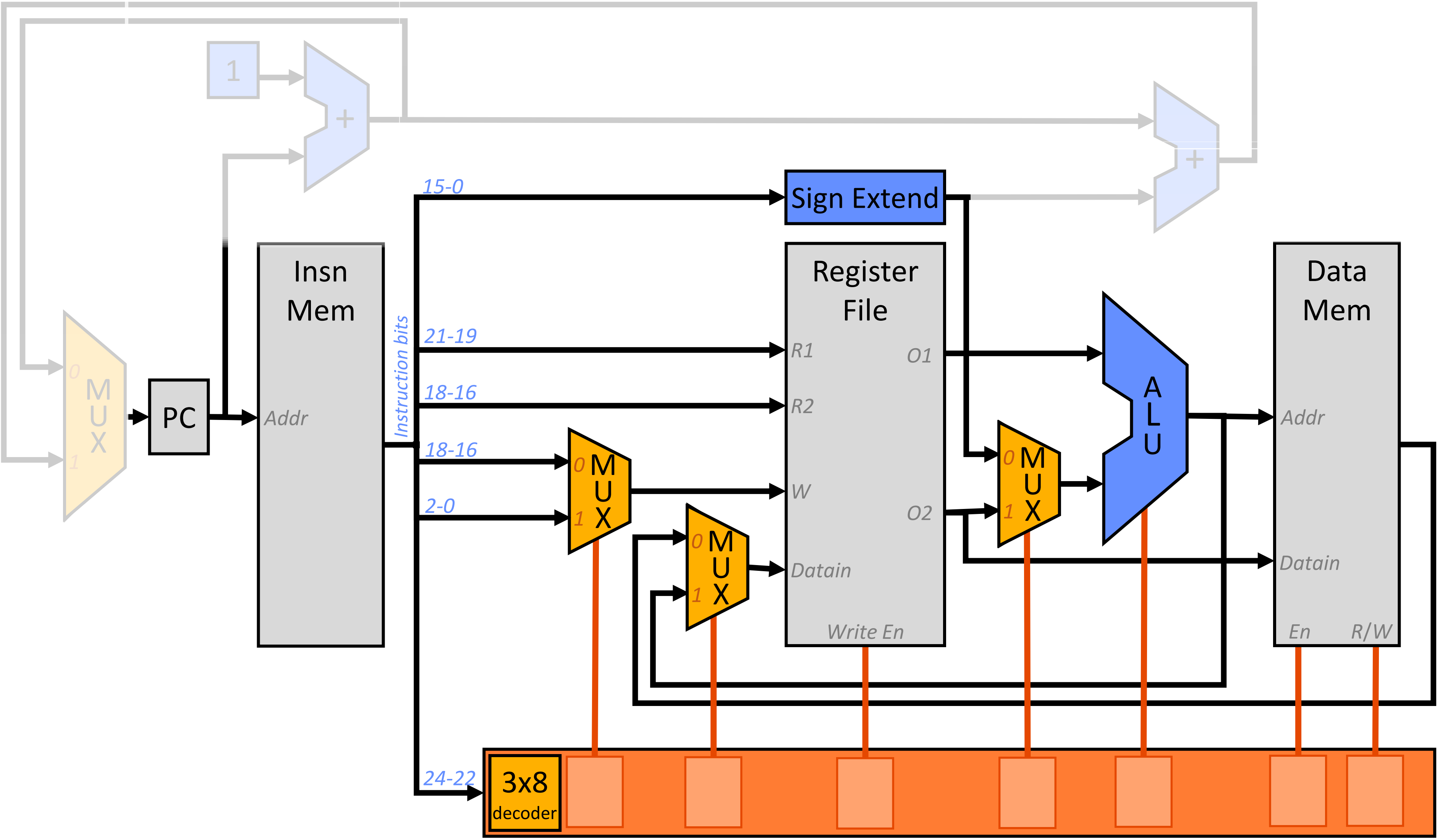
LC2K Single-Cycle Processor



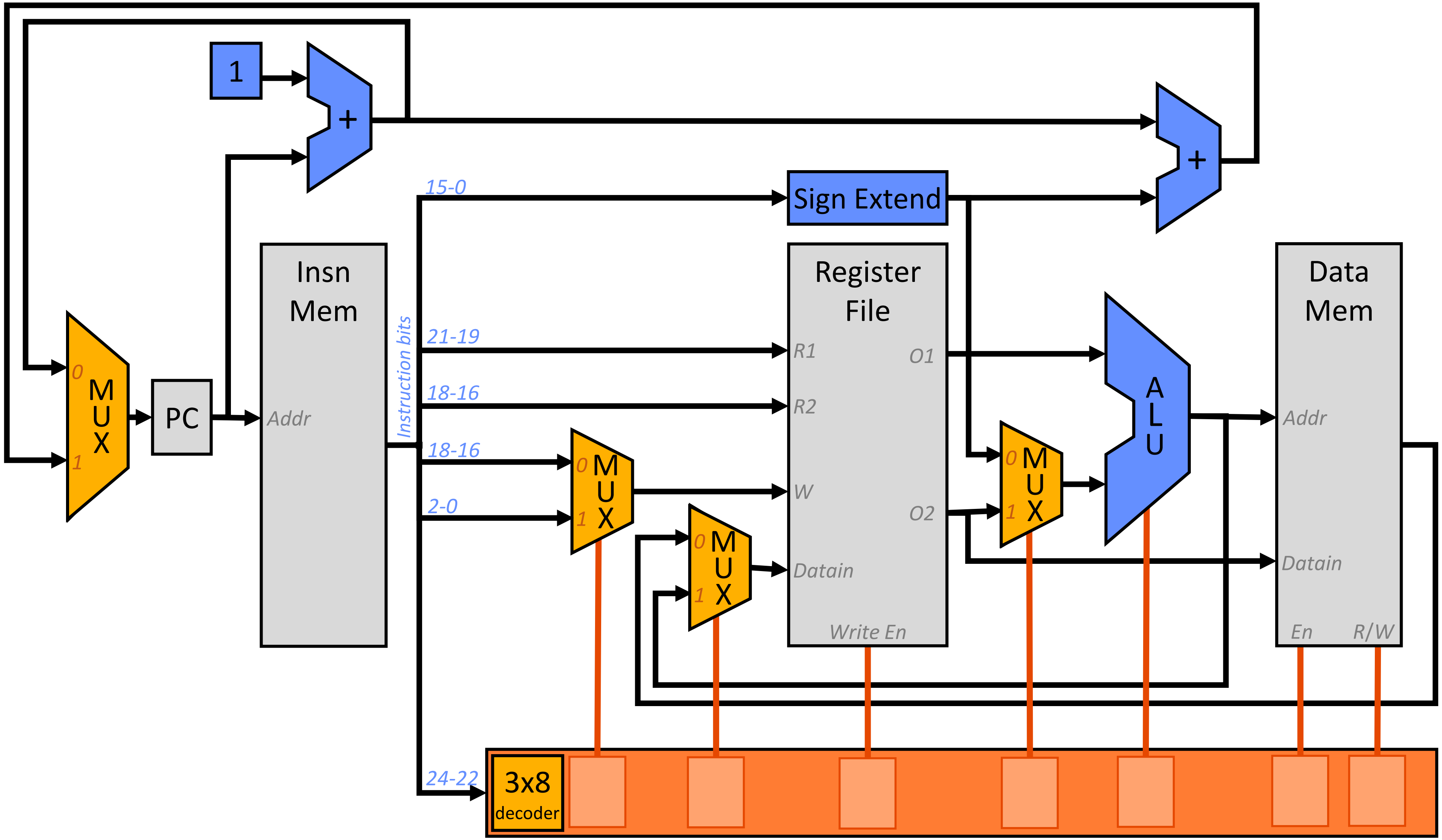
LC2K Single-Cycle Processor



LC2K Single-Cycle Processor



LC2K Single-Cycle Processor

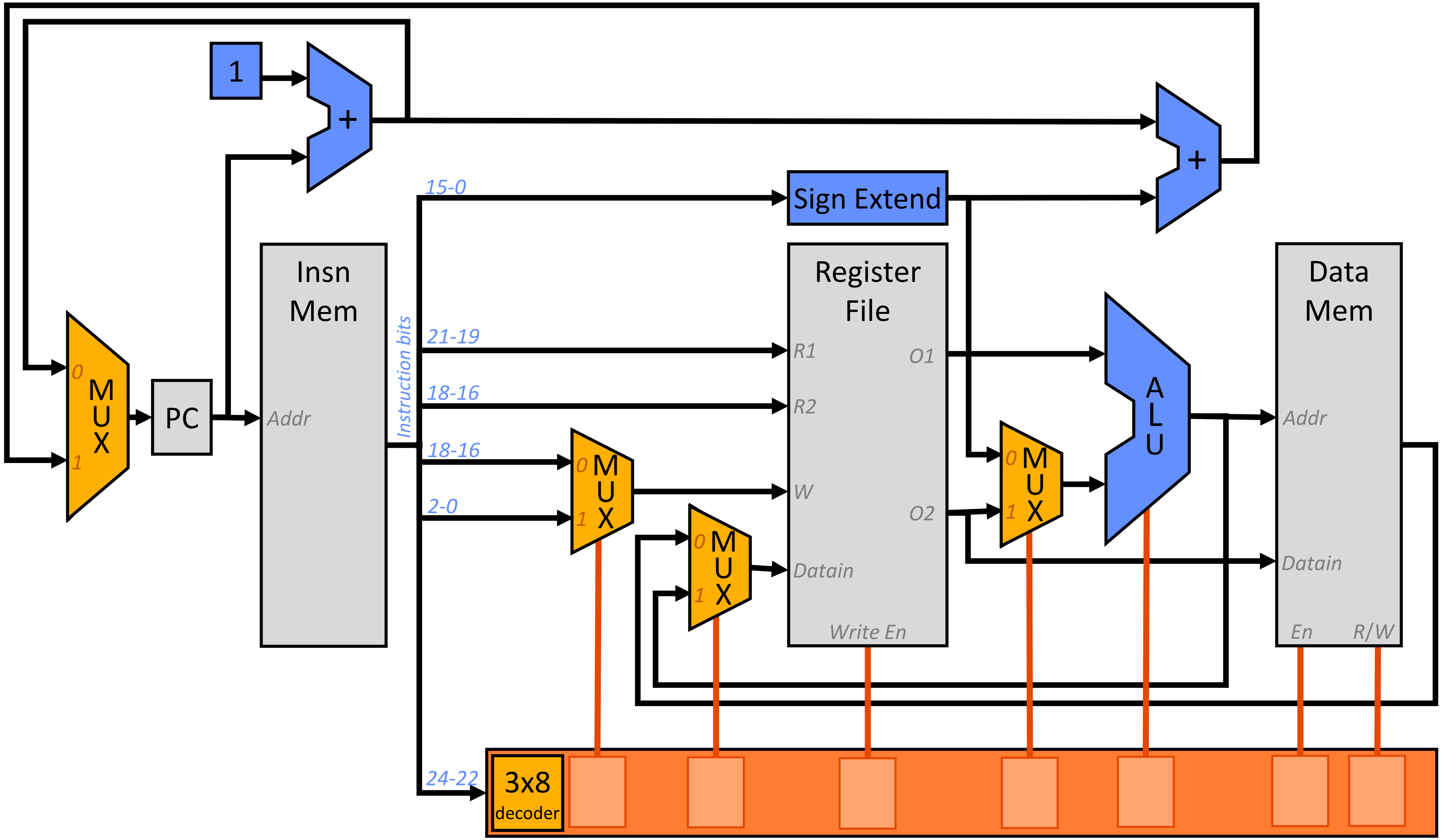


ADD

Instruction	`add regA, regB, destR`
Functionality	`destR = regA + regB;      PC = PC+1`
R-Type	

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

LC2K Single-Cycle Processor

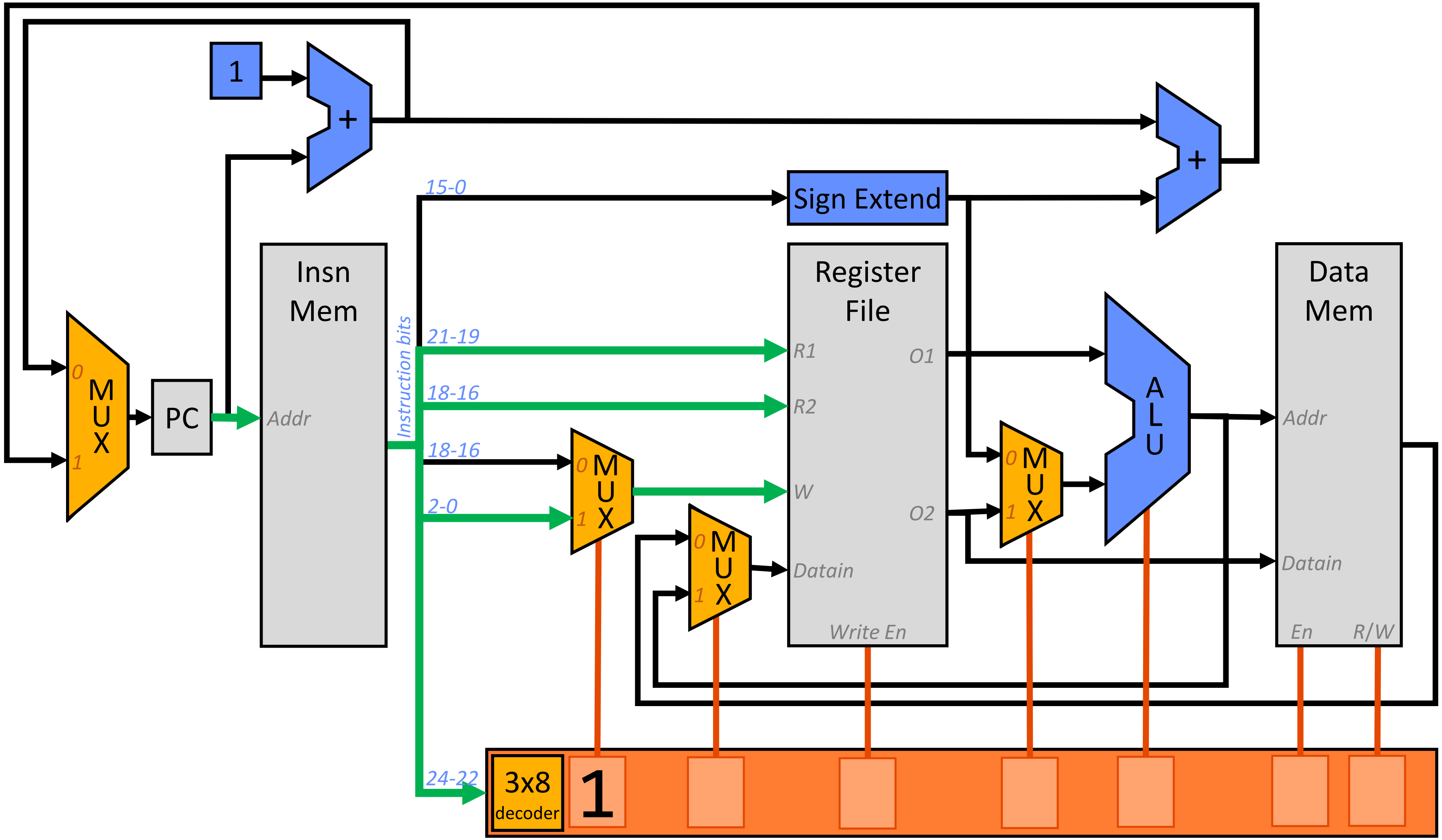


ADD

LC2K Single-Cycle Processor

Instruction	`add regA, regB, destR`
Functionality	`destR = regA + regB;     PC = PC+1`

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

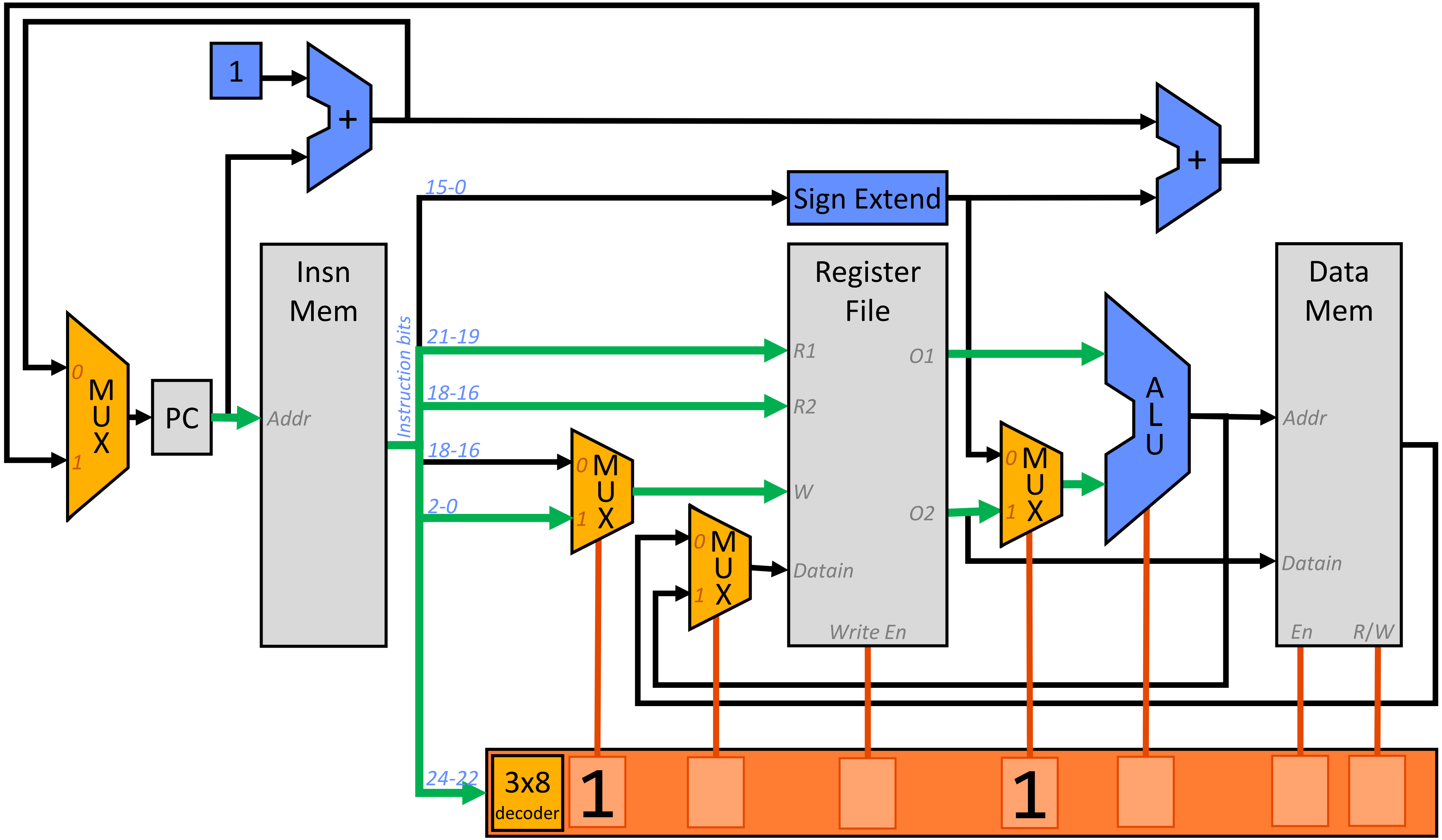


ADD

Instruction	add regA, regB, destR
Functionality	destR = regA + regB; PC = PC+1
R-Type	see bit fields below

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

LC2K Single-Cycle Processor



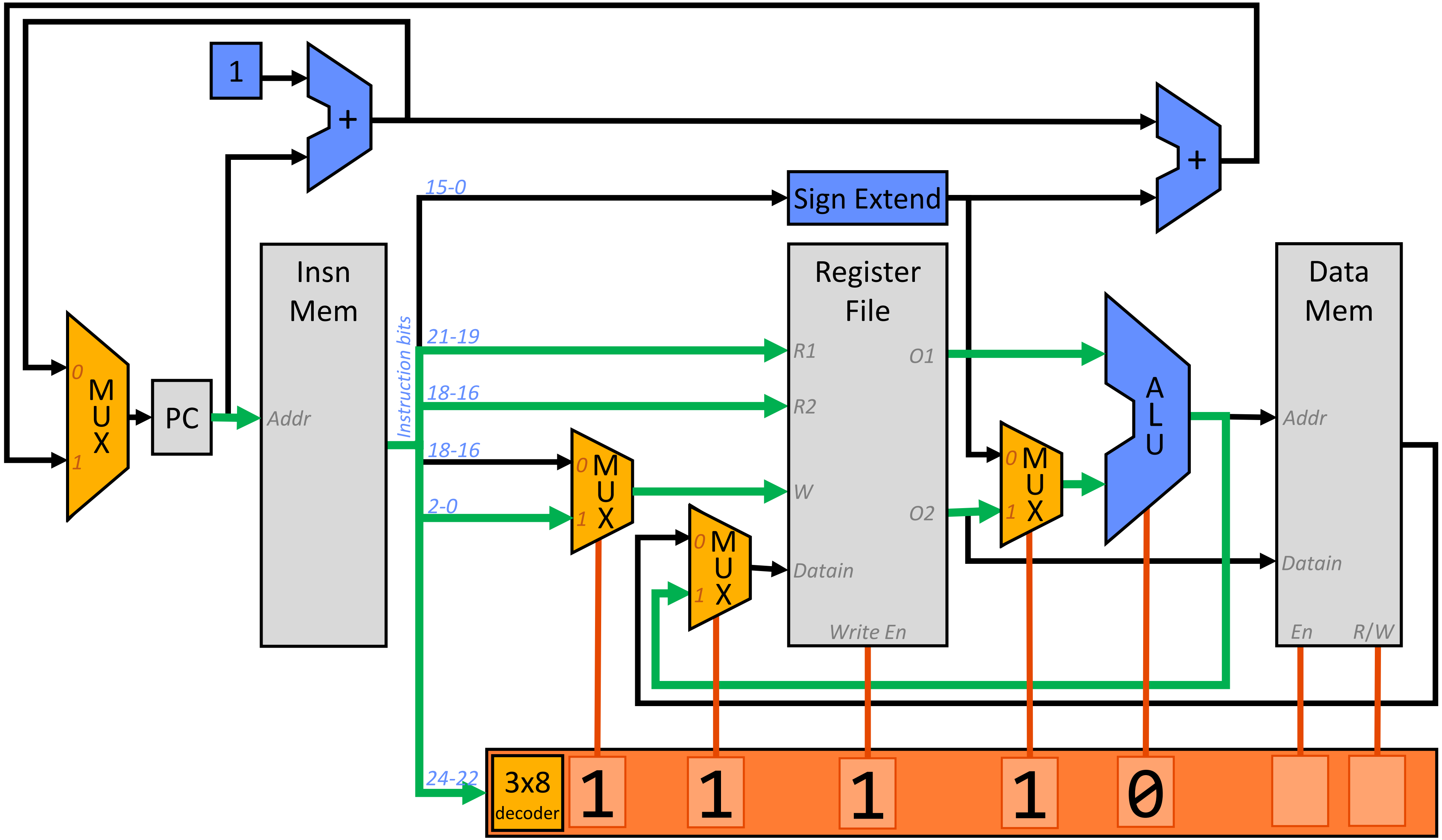
LC2K Single-Cycle Processor

ADD

Instruction	`add regA, regB, destR`
Functionality	`destR = regA + regB;        PC = PC+1`

R-Type

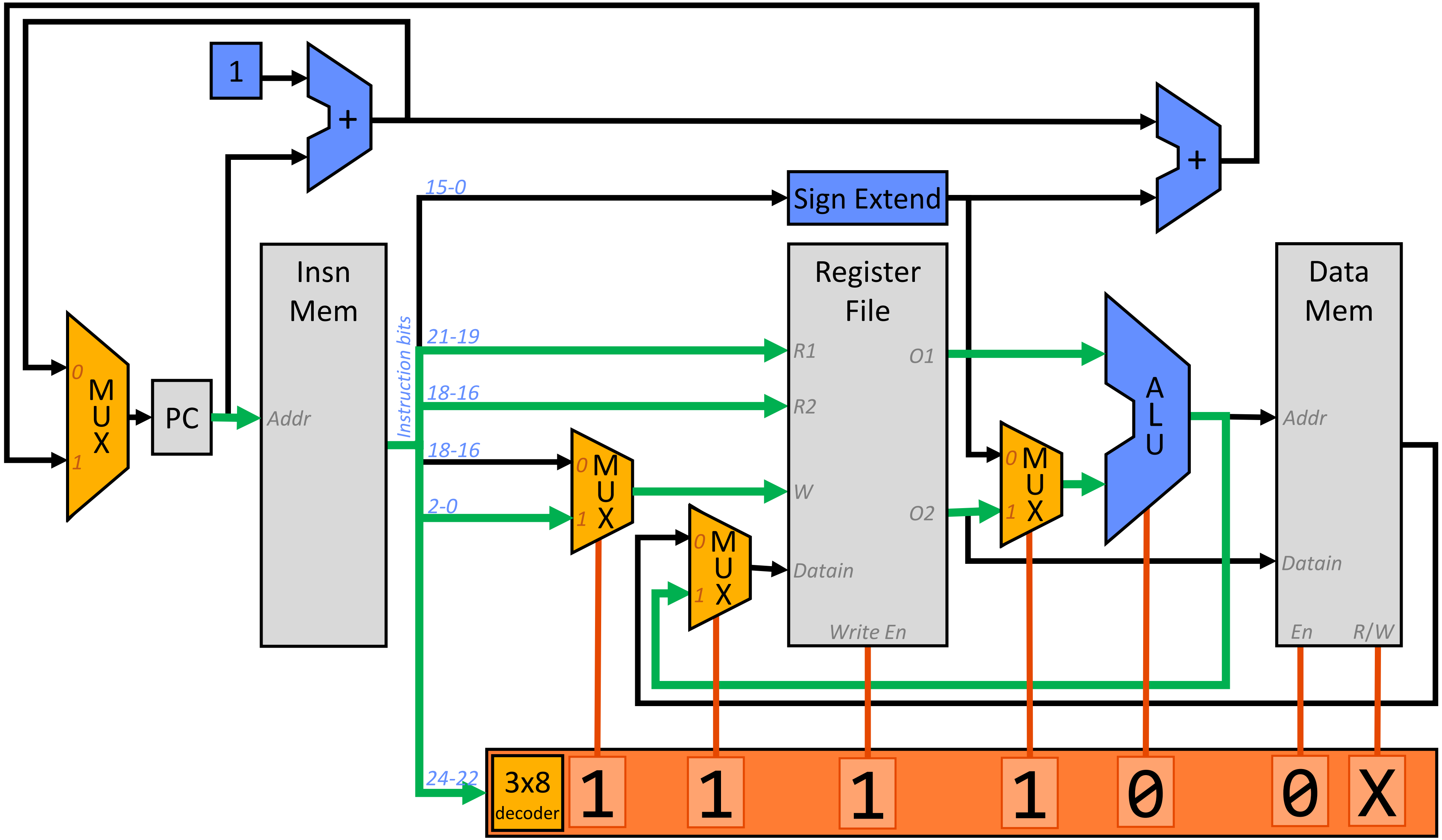
31-25	24-22	21-19	18-16	15-3	2-0
	opcode	regA	regB		destR



ADD	Instruction	add regA, regB, destR
	Functionality	destR = regA + regB; PC = PC+1
	R-Type	

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

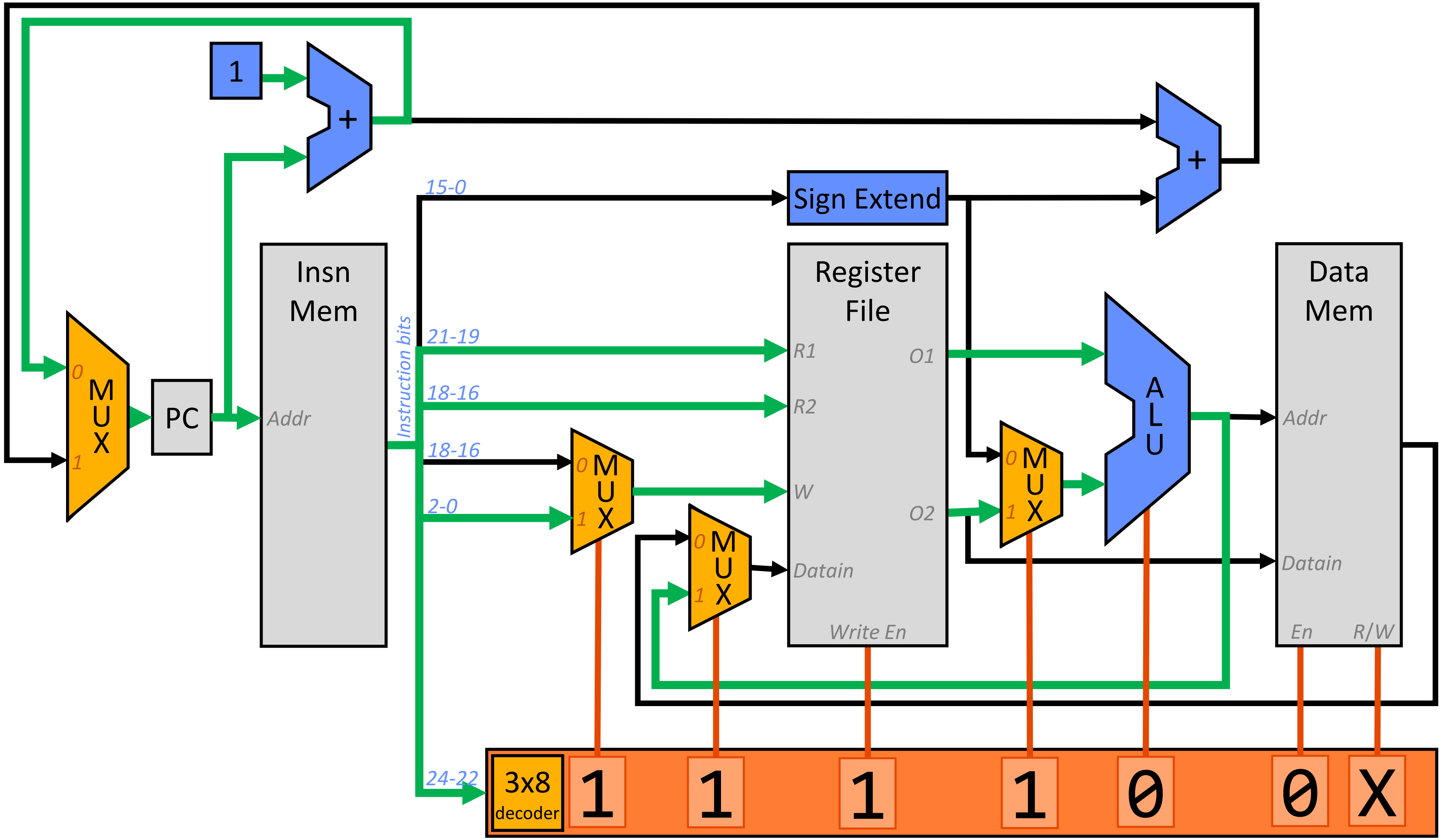
LC2K Single-Cycle Processor



ADD	Instruction	add regA, regB, destR
	Functionality	destR = regA + regB; PC = PC+1

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

LC2K Single-Cycle Processor



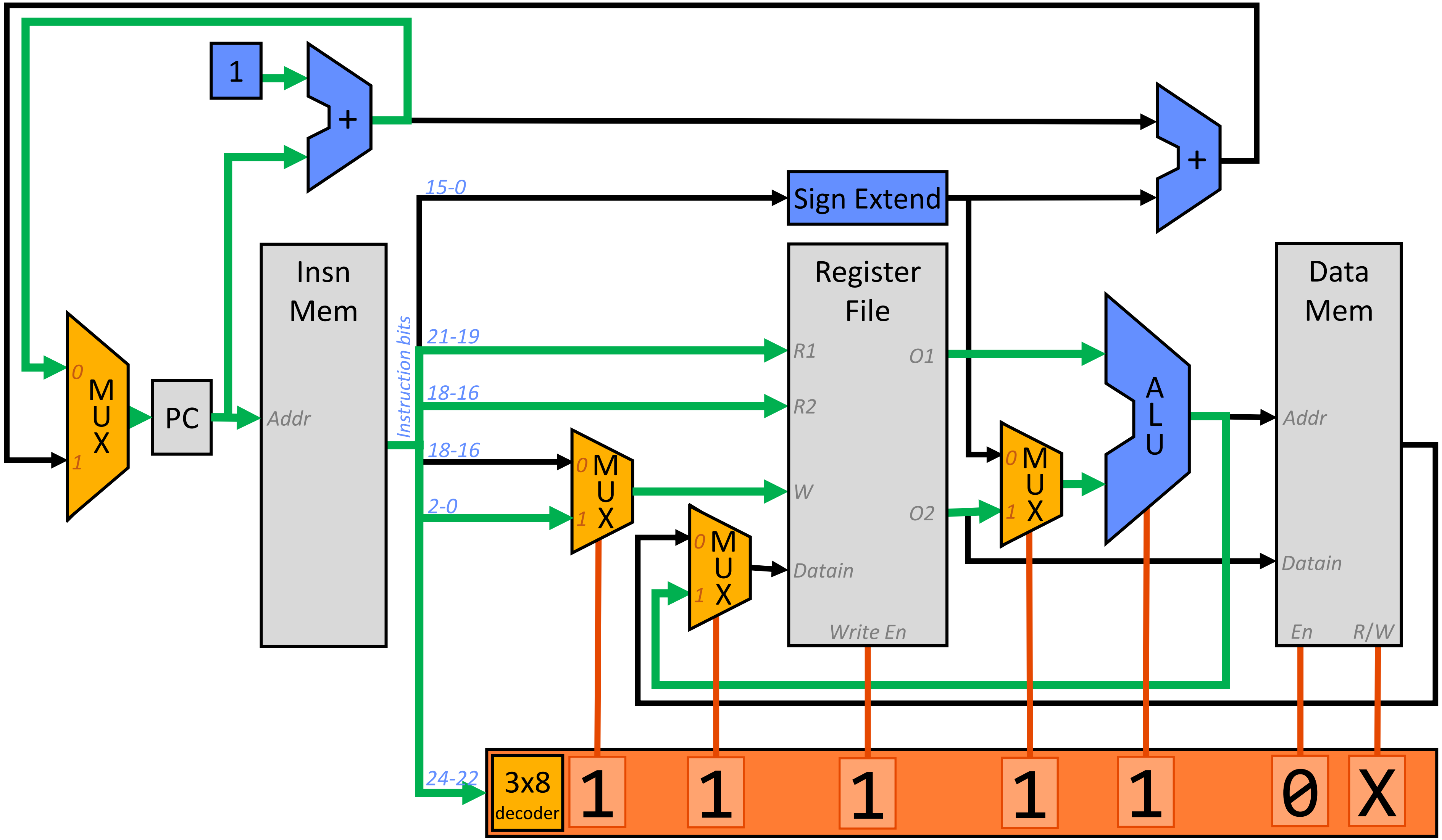
NOR

Instruction: `nor regA, regB, destR`

Functionality: `destR = ~(regA|regB);     PC = PC+1`

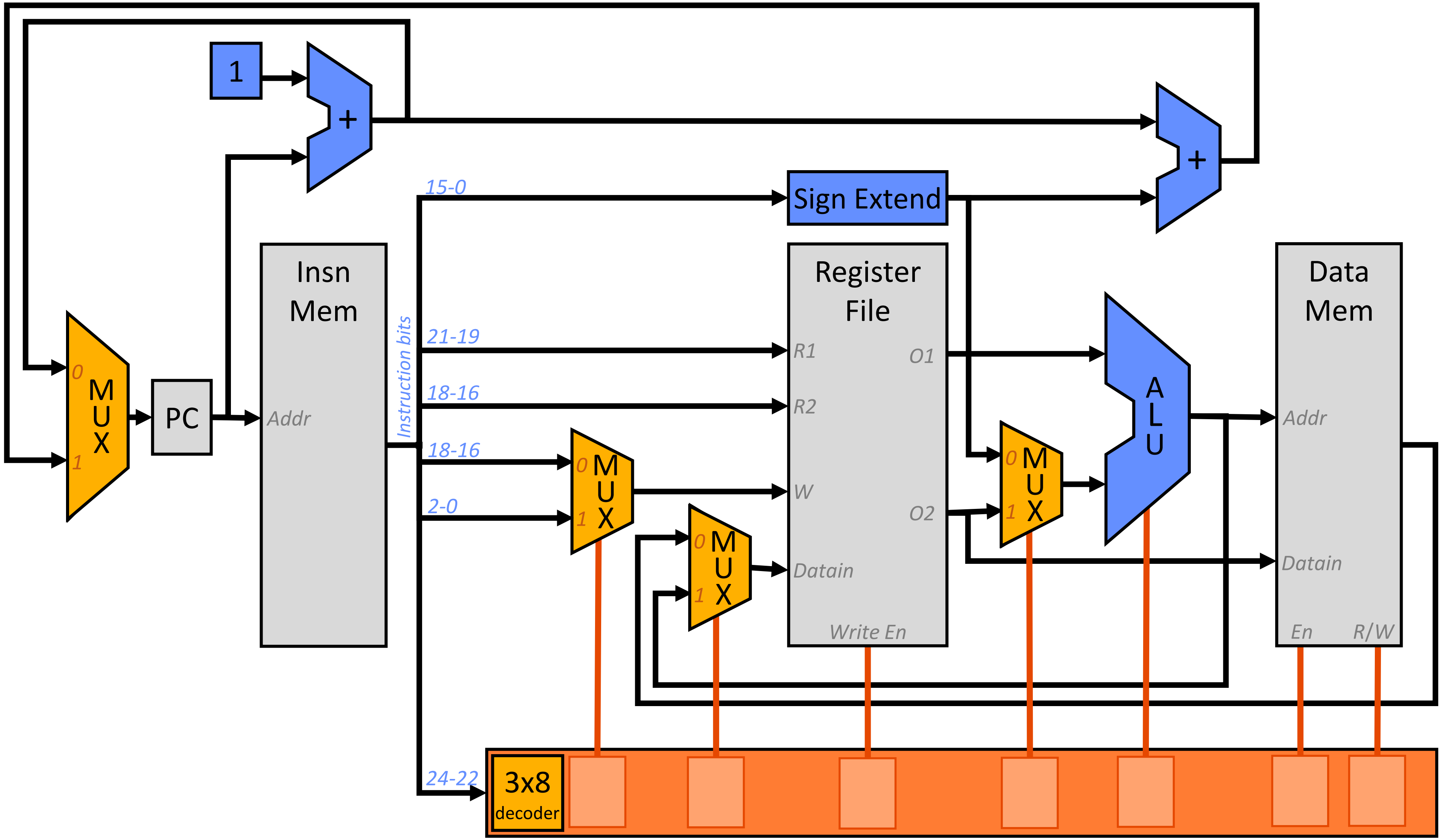
R-Type

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



LW	Instruction	lw regA, regB, offset				
	Functionality	regB = M[regA + offset]; PC = PC+1				
		31-25	24-22	21-19	18-16	15-0
	I-Type		opcode	regA	regB	destR

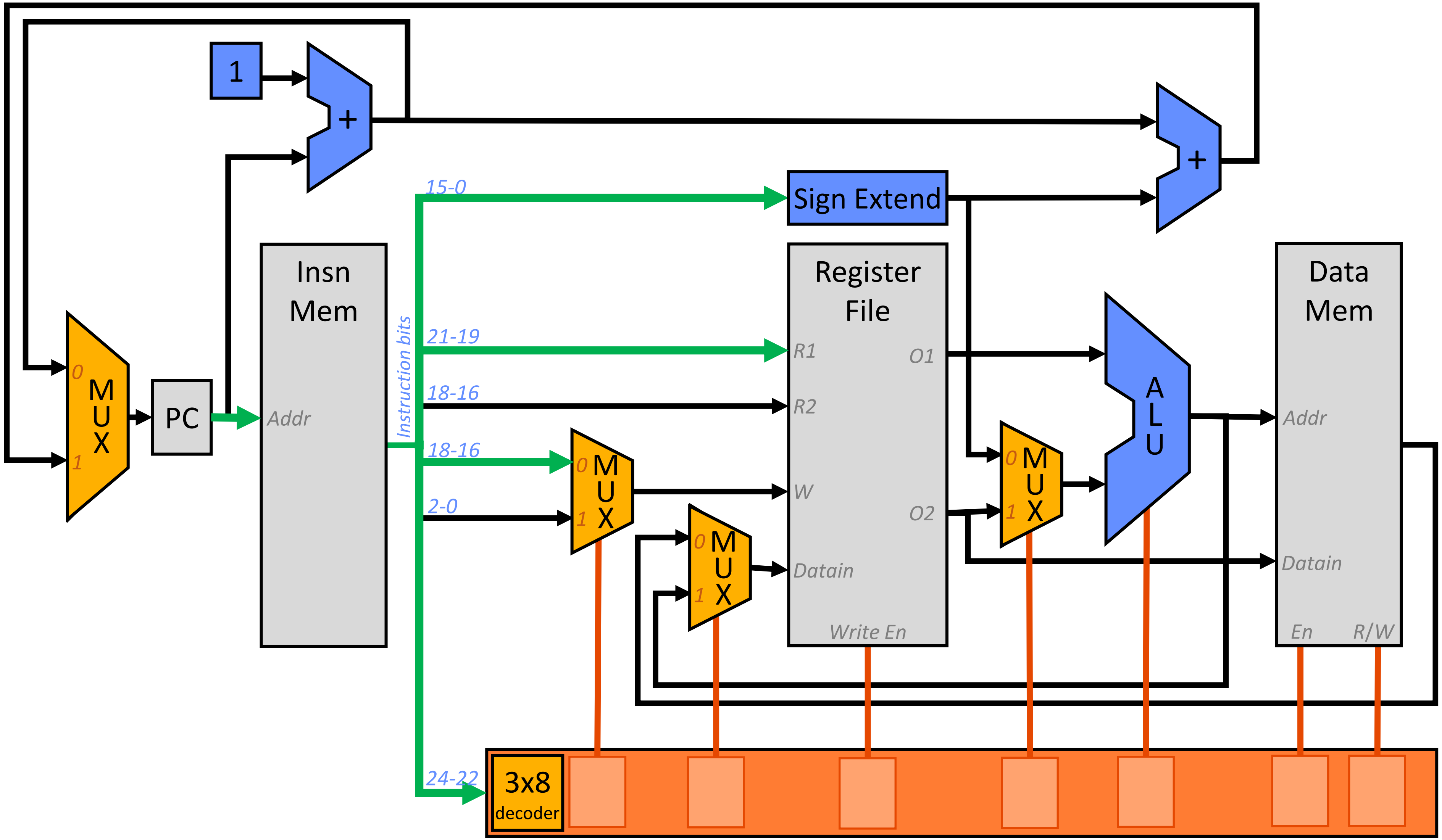
LC2K Single-Cycle Processor



LW		
	Instruction	lw regA, regB, offset
	Functionality	regB = M[regA + offset]; PC = PC+1

I-Type	31-25	24-22	21-19	18-16	15-0
		opcode	regA	regB	destR

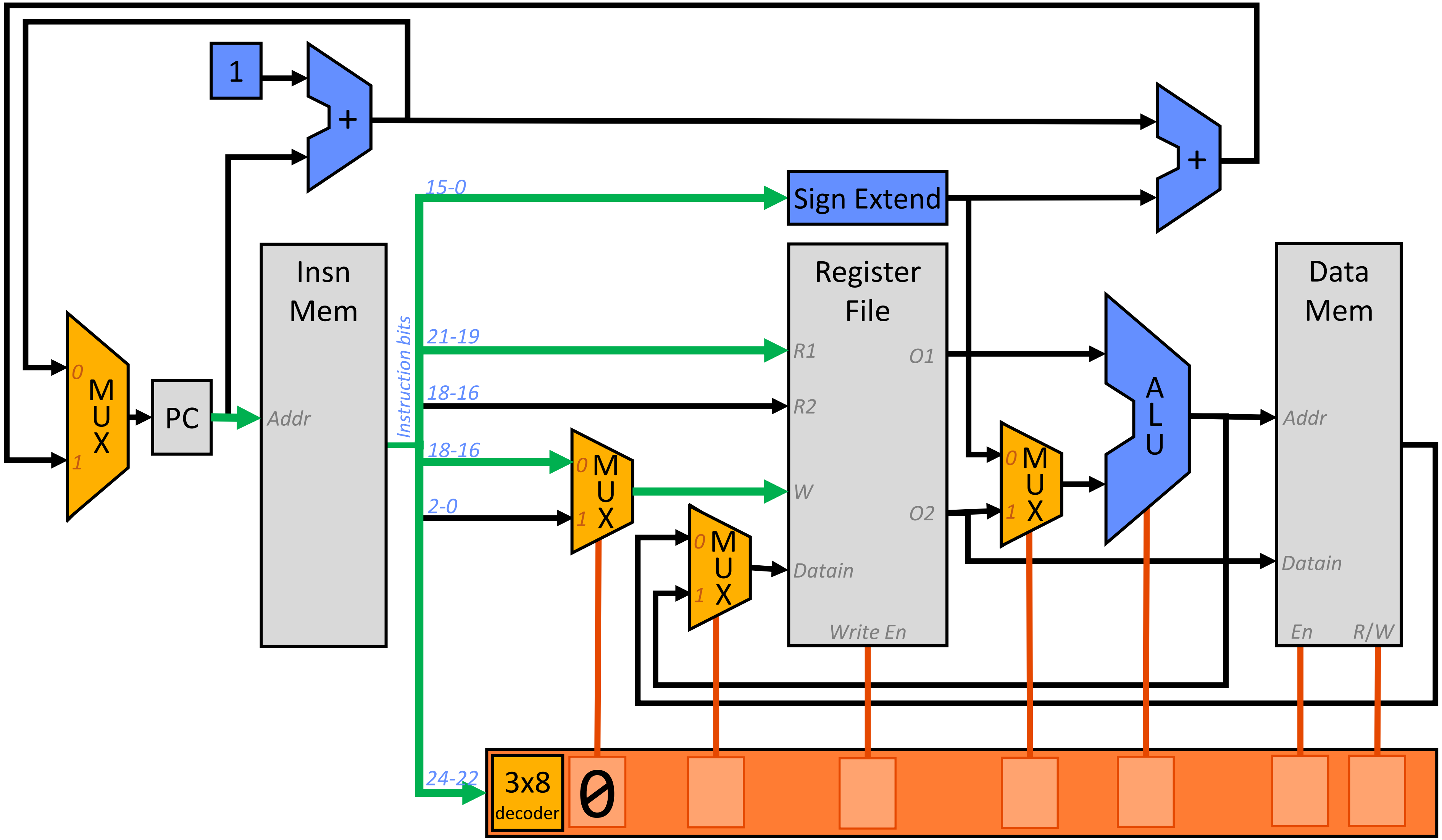
LC2K Single-Cycle Processor



LW	Instruction	`lw regA, regB, offset`
	Functionality	`regB = M[regA + offset];   PC = PC+1`
	I-Type	

31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR

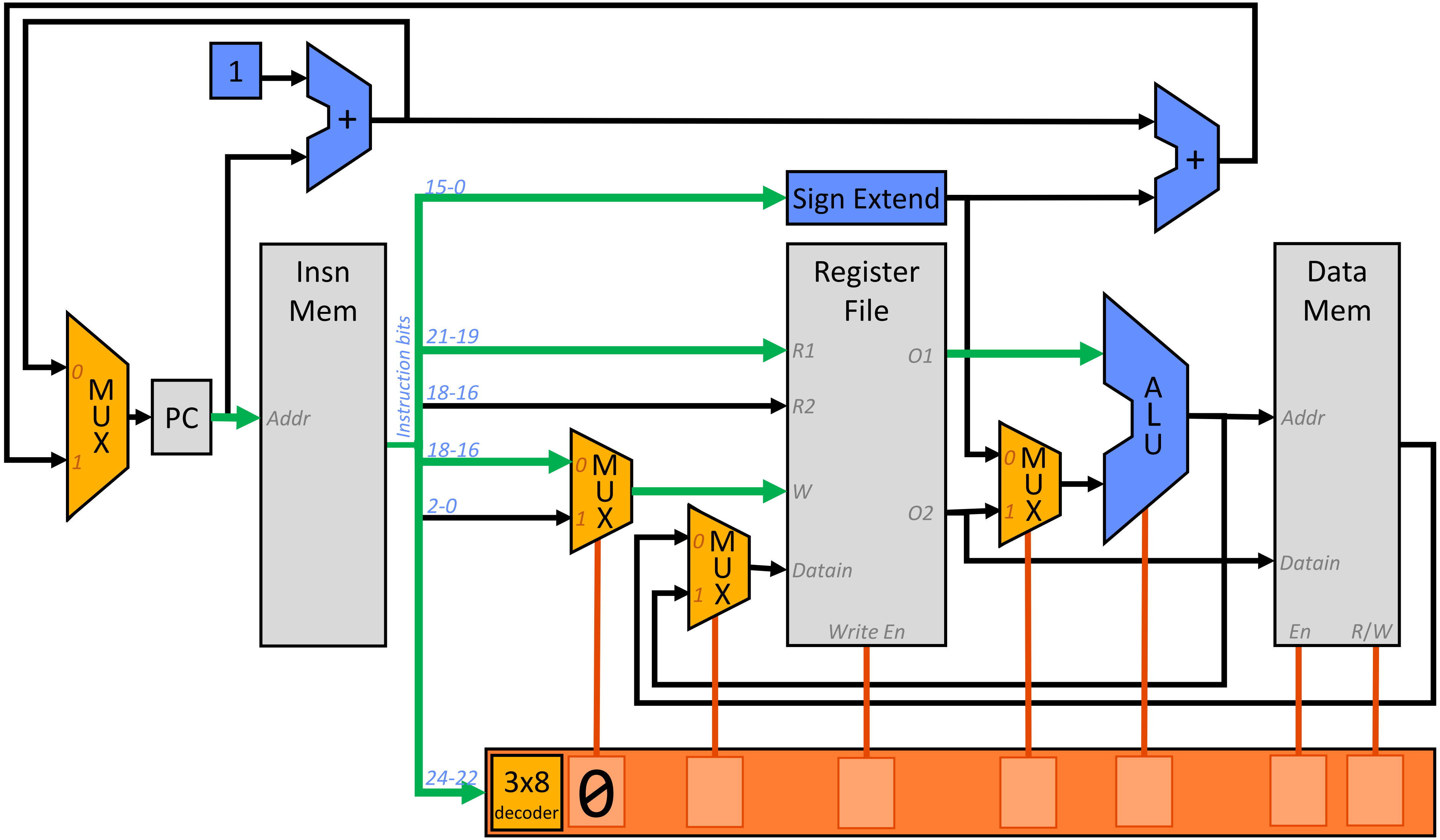
LC2K Single-Cycle Processor



LW

LC2K Single-Cycle Processor

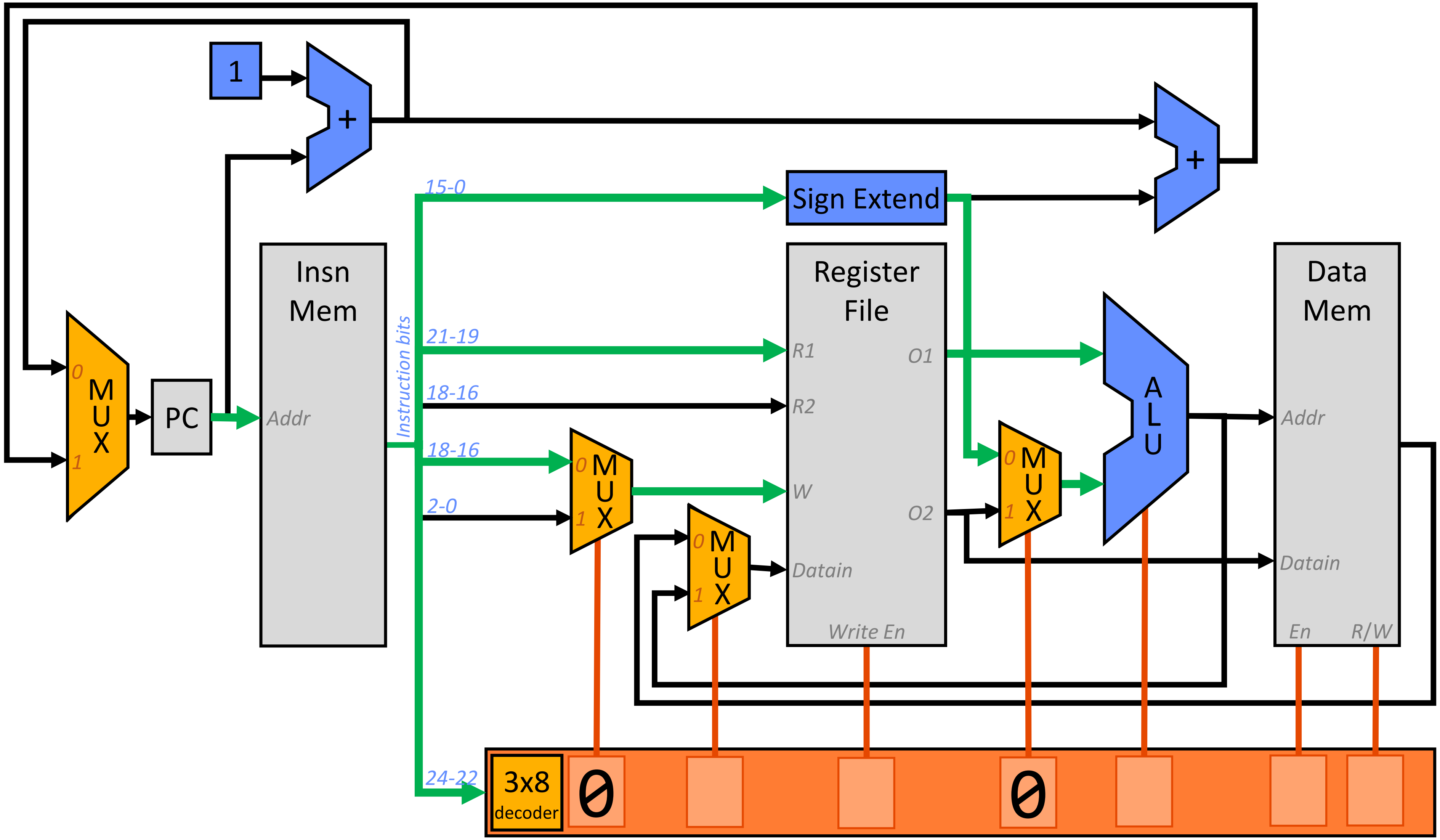
Instruction	`lw regA, regB, offset`				
Functionality	`regB = M[regA + offset];   PC = PC+1`				
	31-25	24-22	21-19	18-16	15-0
I-Type		opcode	regA	regB	destR



LW

Instruction	lw regA, regB, offset				
Functionality	regB = M[regA + offset]; PC = PC+1				
I-Type	31-25	24-22	21-19	18-16	15-0
		opcode	regA	regB	destR

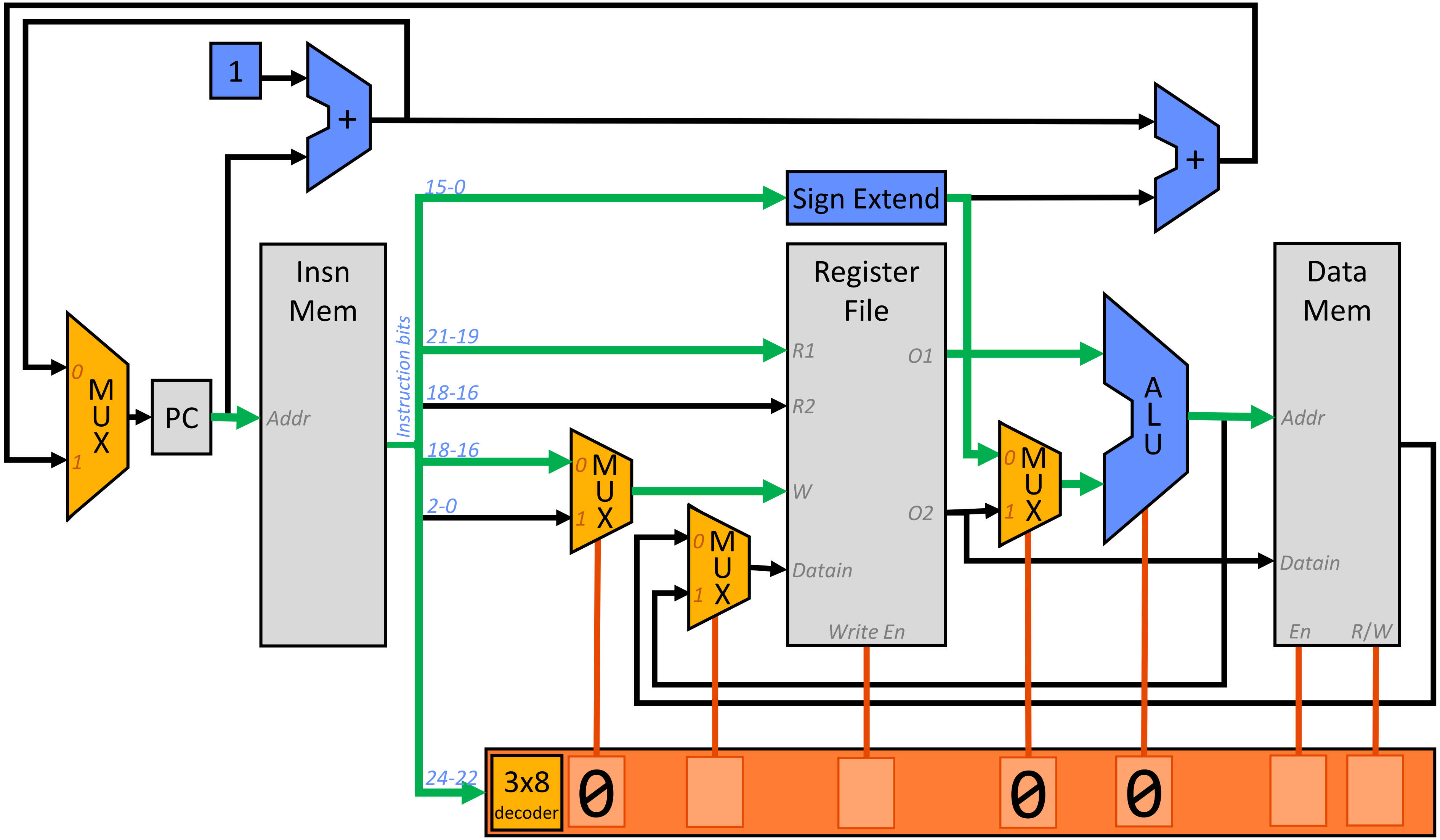
LC2K Single-Cycle Processor



LW

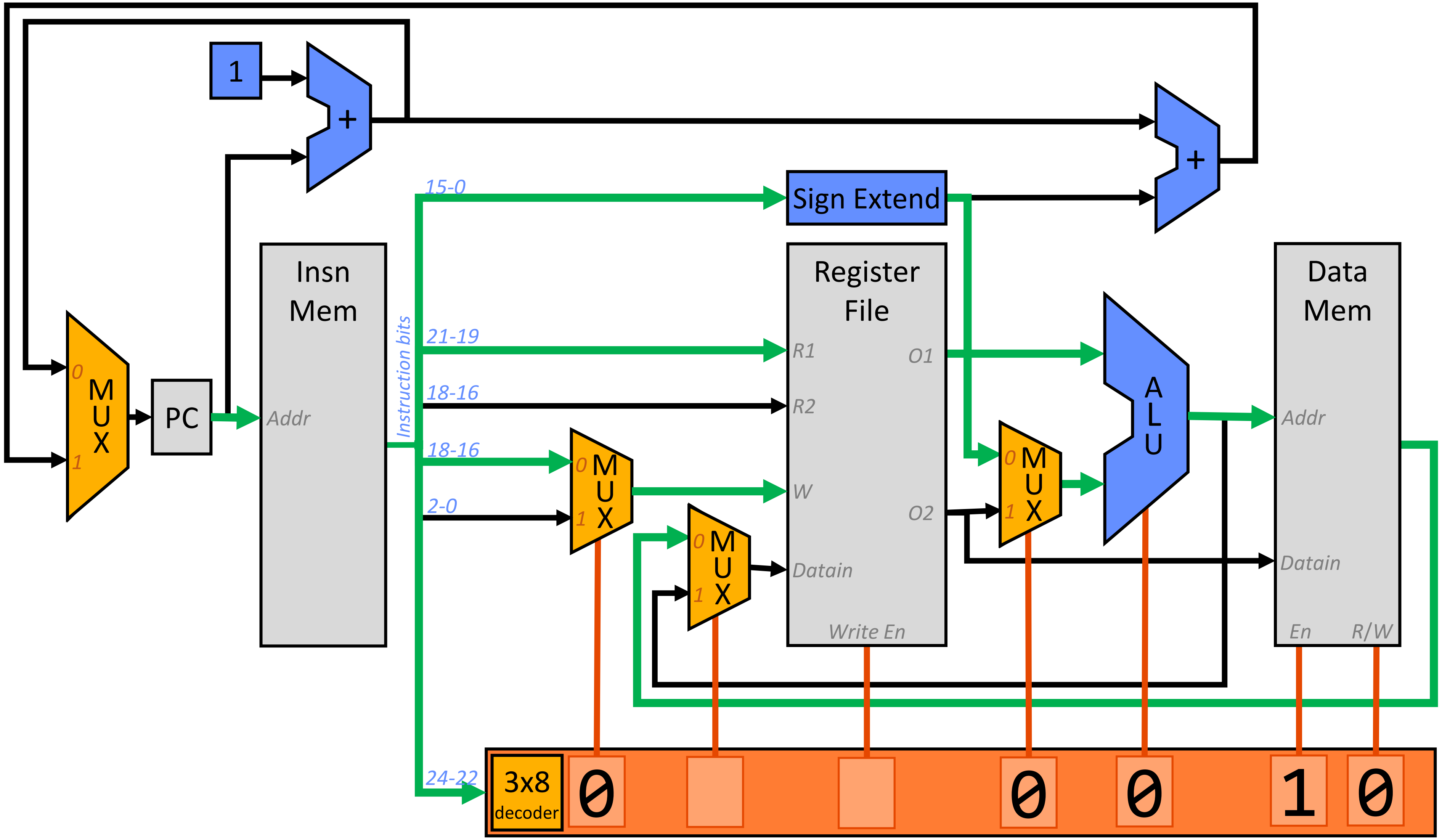
Instruction	lw regA, regB, offset					
Functionality	regB = M[regA + offset]; PC = PC+1					
	31-25	24-22	21-19	18-16	15-0	
I-Type		opcode	regA	regB	destR	

LC2K Single-Cycle Processor



LW	Instruction	lw regA, regB, offset				
	Functionality	regB = M[regA + offset]; PC = PC+1				
		31-25	24-22	21-19	18-16	15-0
	I-Type		opcode	regA	regB	destR

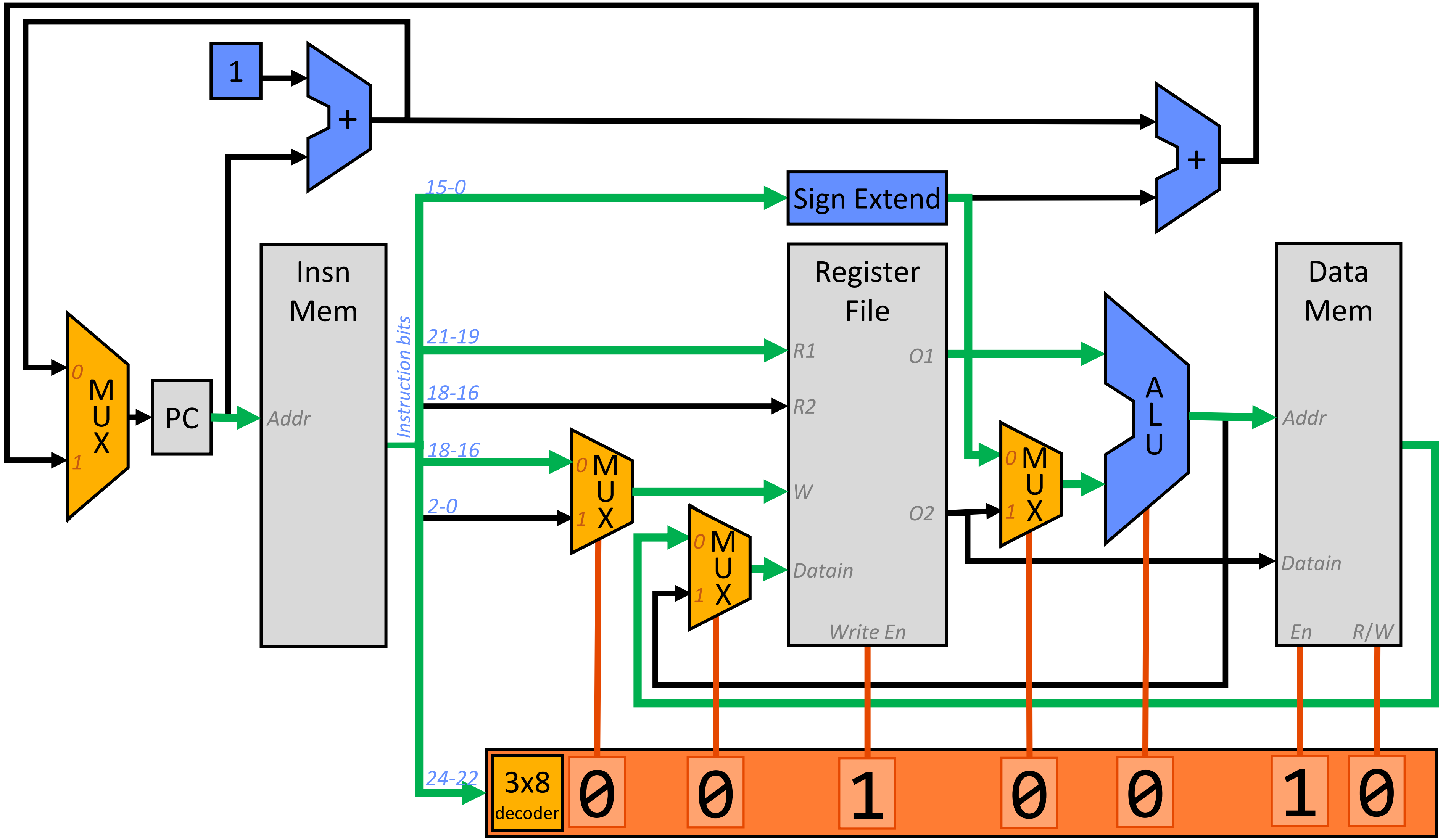
LC2K Single-Cycle Processor



LW	Instruction	`lw regA, regB, offset`
	Functionality	`regB = M[regA + offset];   PC = PC+1`
	I-Type	

31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR

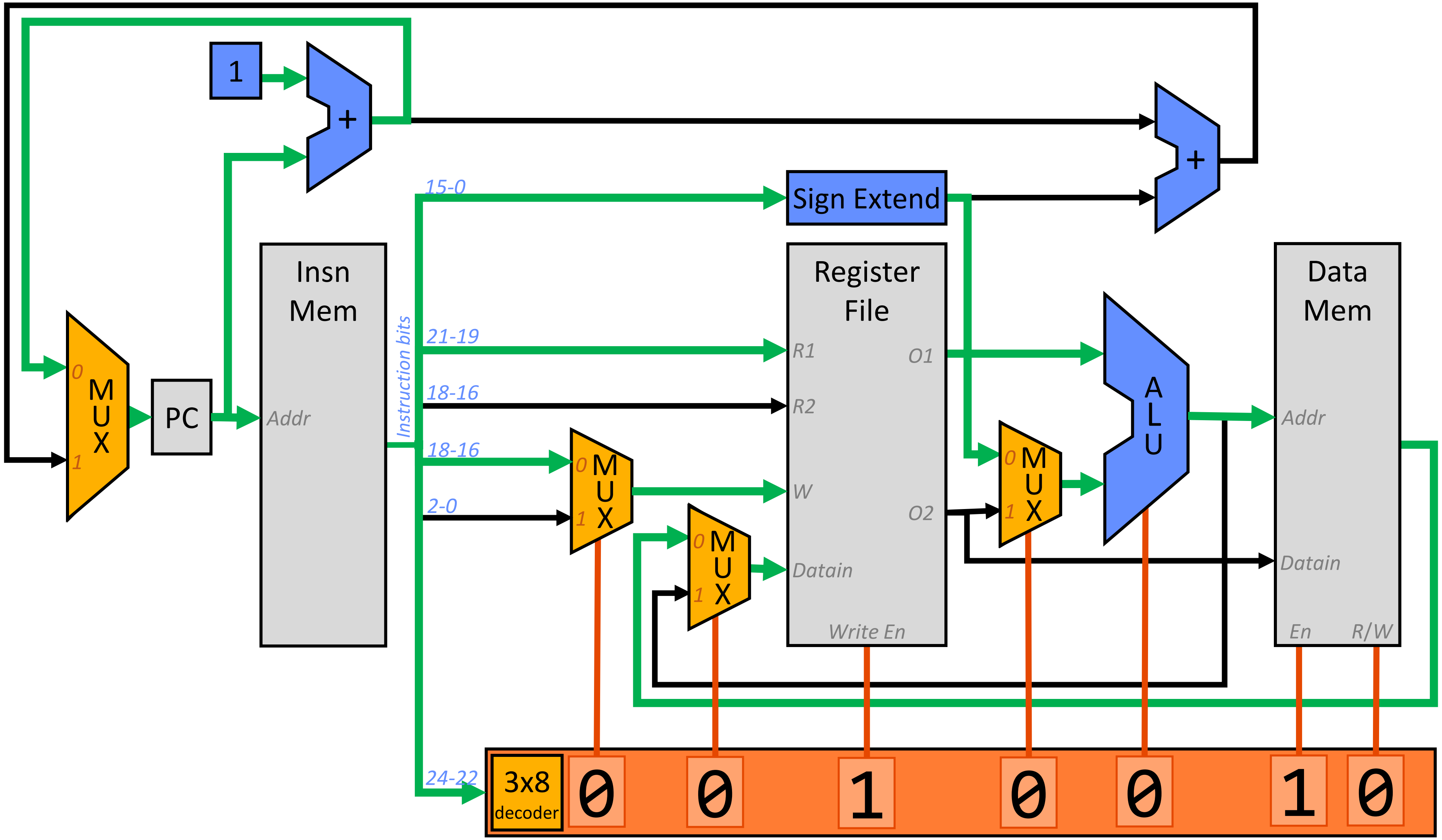
LC2K Single-Cycle Processor



LC2K Single-Cycle Processor

LW		
	Instruction	`lw regA, regB, offset`
	Functionality	`regB = M[regA + offset];   PC = PC+1`
	I-Type	

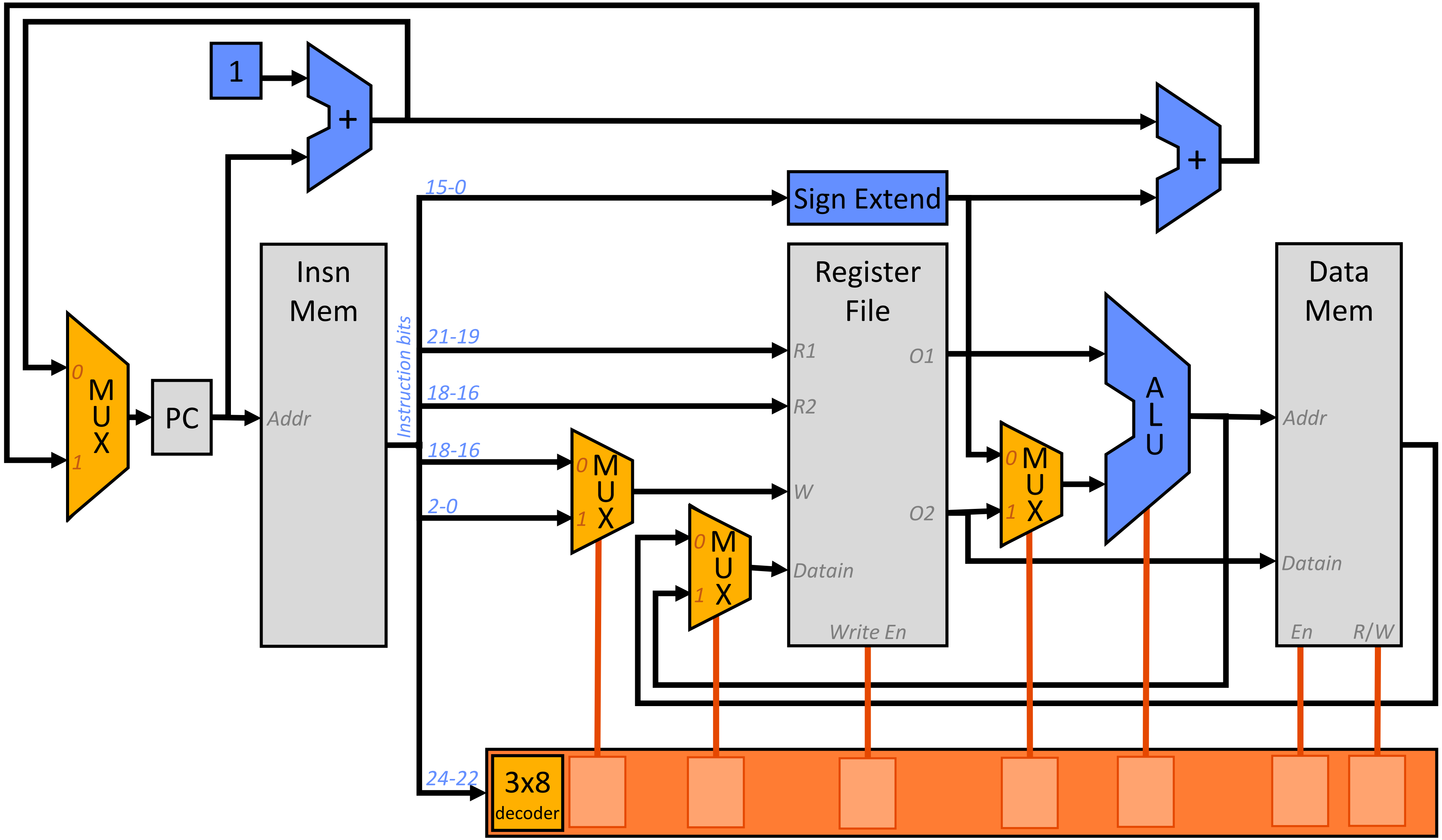
31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR



SW	Instruction	sw regA, regB, offset
	Functionality	M[regA + offset] = regB; PC = PC+1
	I-Type	

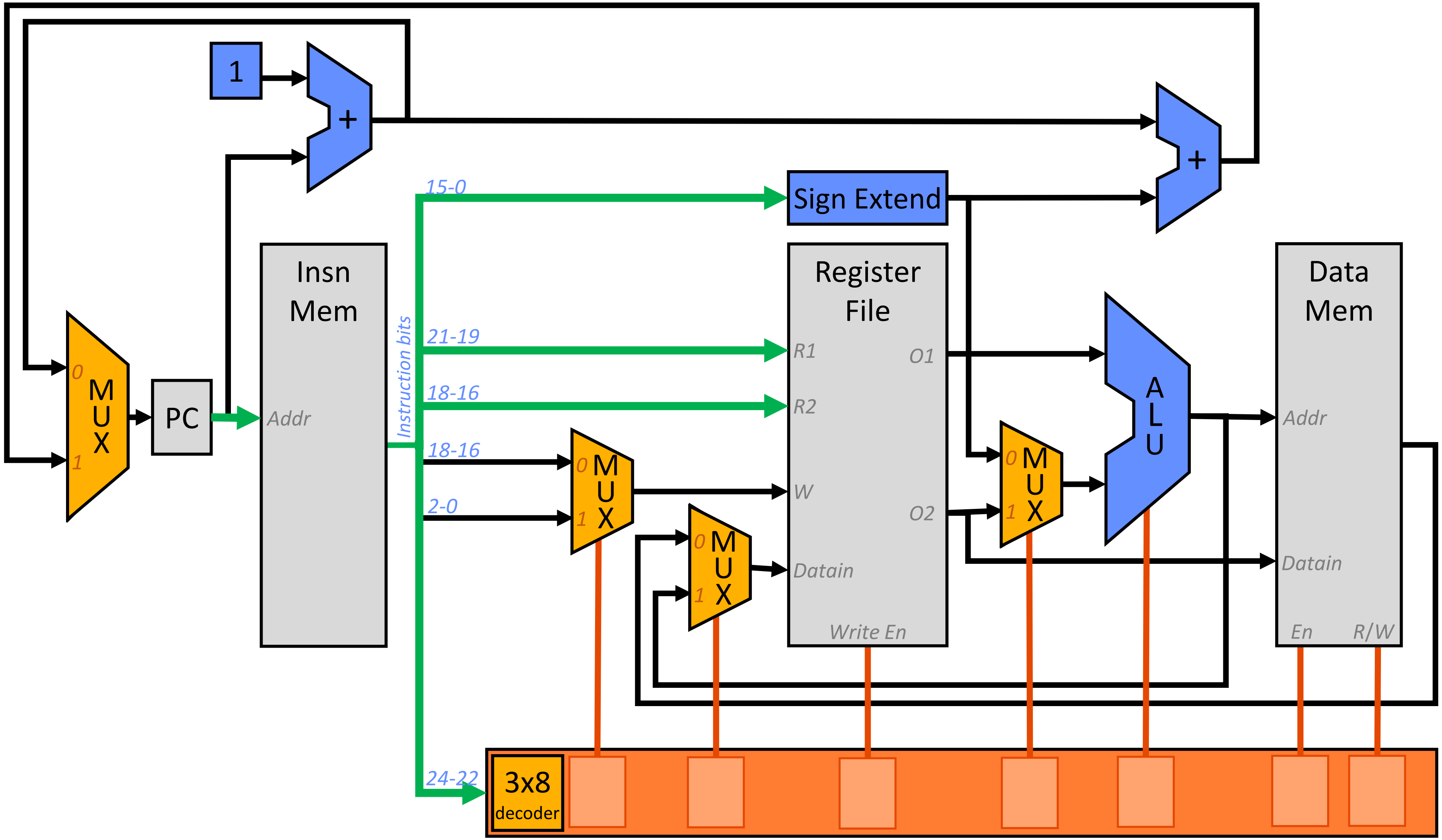
31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR

LC2K Single-Cycle Processor



SW	Instruction	sw regA, regB, offset				
	Functionality	M[regA + offset] = regB; PC = PC+1				
	I-Type	31-25	24-22	21-19	18-16	15-0
			opcode	regA	regB	destR

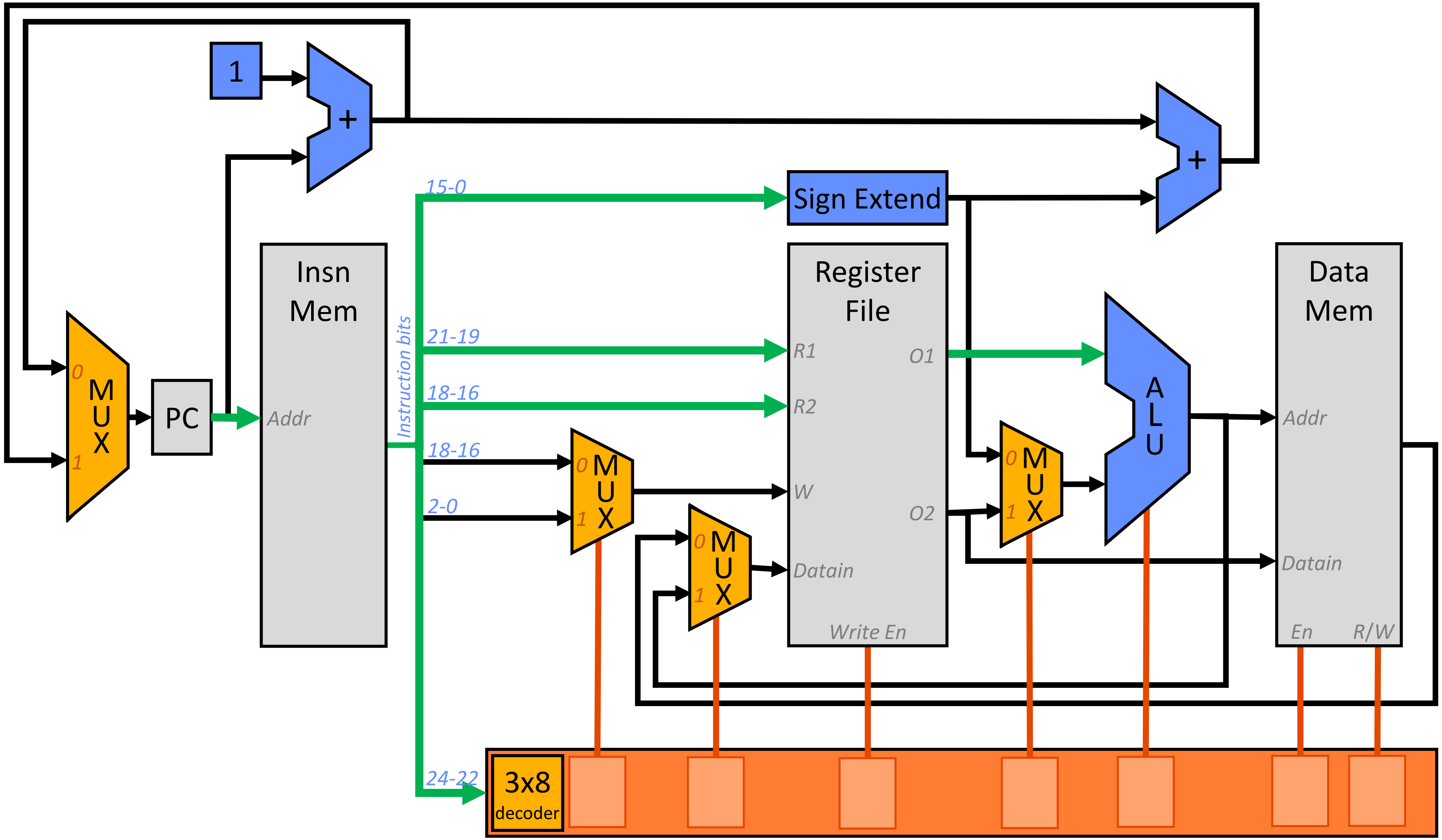
LC2K Single-Cycle Processor



LC2K Single-Cycle Processor

SW		
	Instruction	sw regA, regB, offset
	Functionality	M[regA + offset] = regB; PC = PC+1
	I-Type	

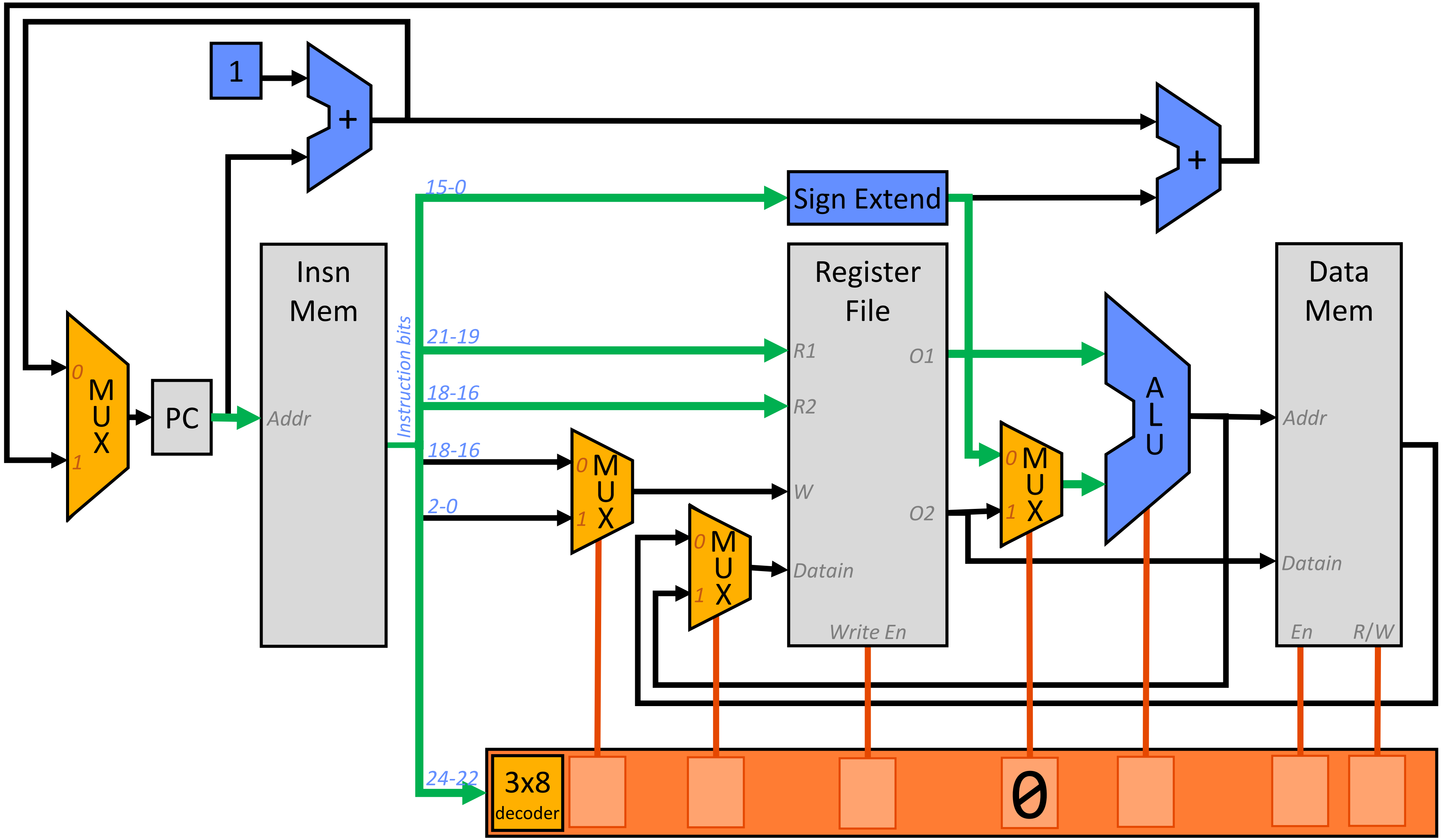
31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR



LC2K Single-Cycle Processor

SW	Instruction	`sw regA, regB, offset`
	Functionality	`M[regA + offset] = regB;   PC = PC+1`
	I-Type	

31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR



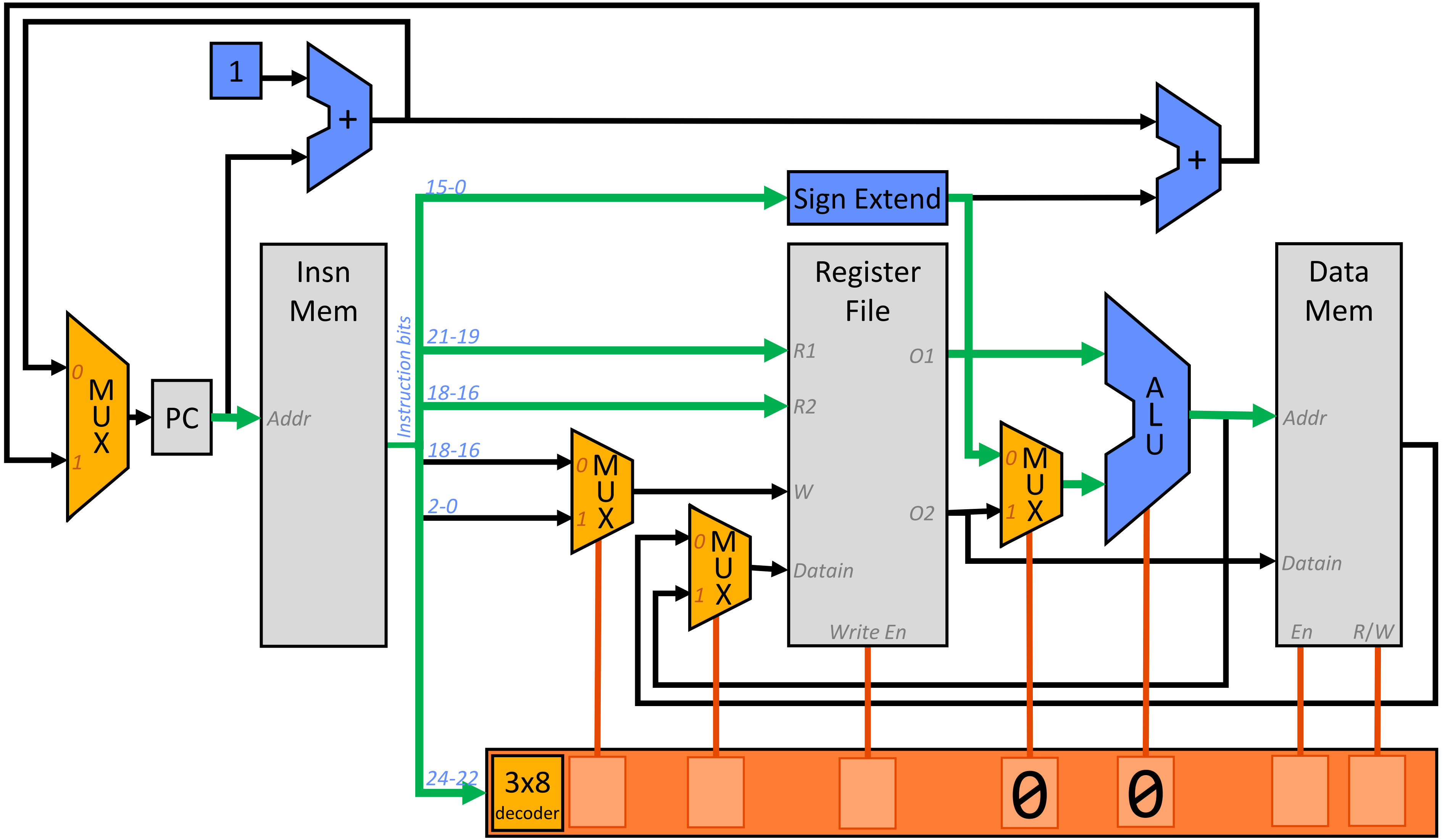
SW

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

I-Type

31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR

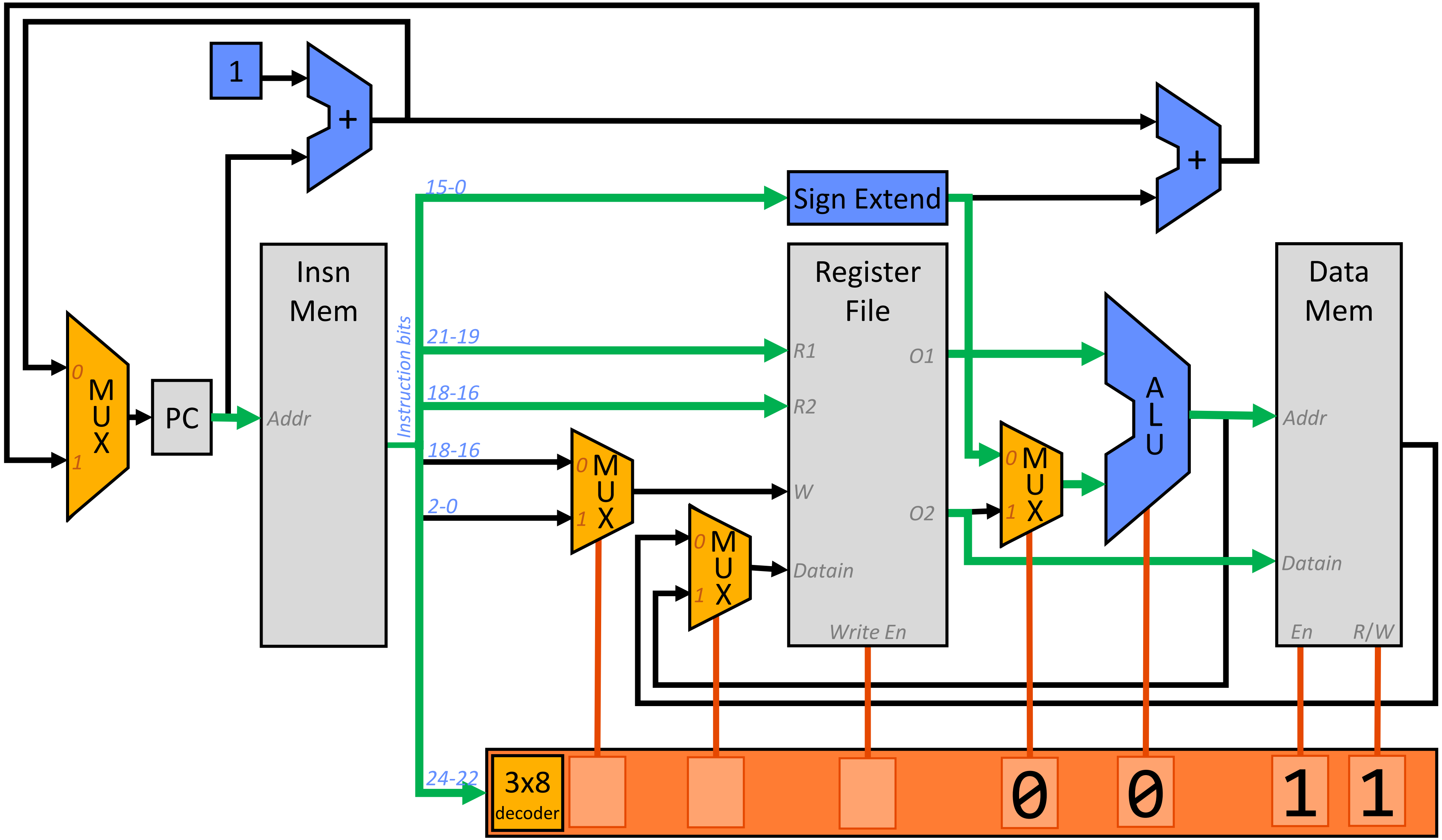
LC2K Single-Cycle Processor



SW

Instruction	sw regA, regB, offset				
Functionality	M[regA + offset] = regB; PC = PC+1				
	31-25	24-22	21-19	18-16	15-0
I-Type		opcode	regA	regB	destR

LC2K Single-Cycle Processor

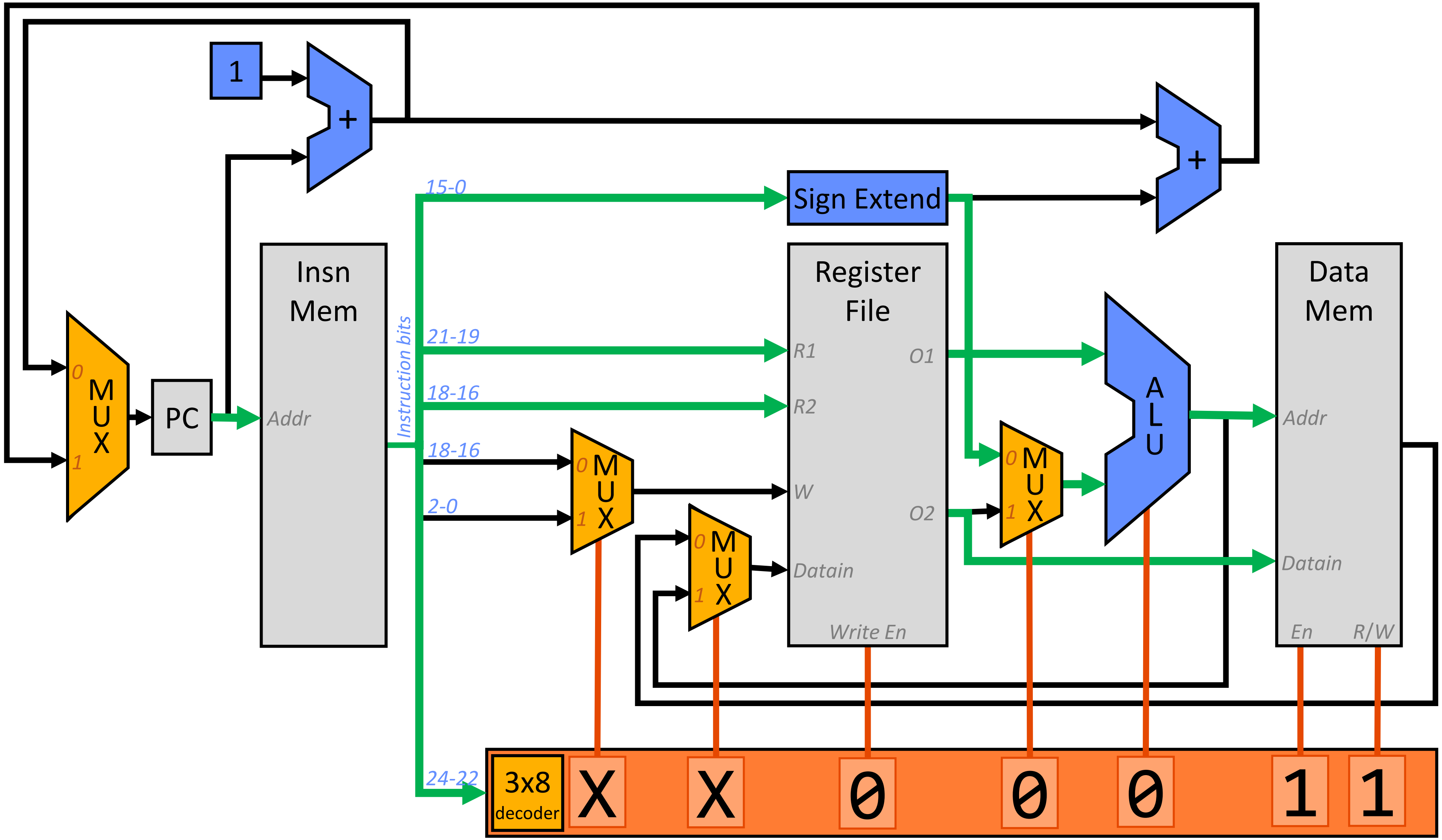


SW

LC2K Single-Cycle Processor

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

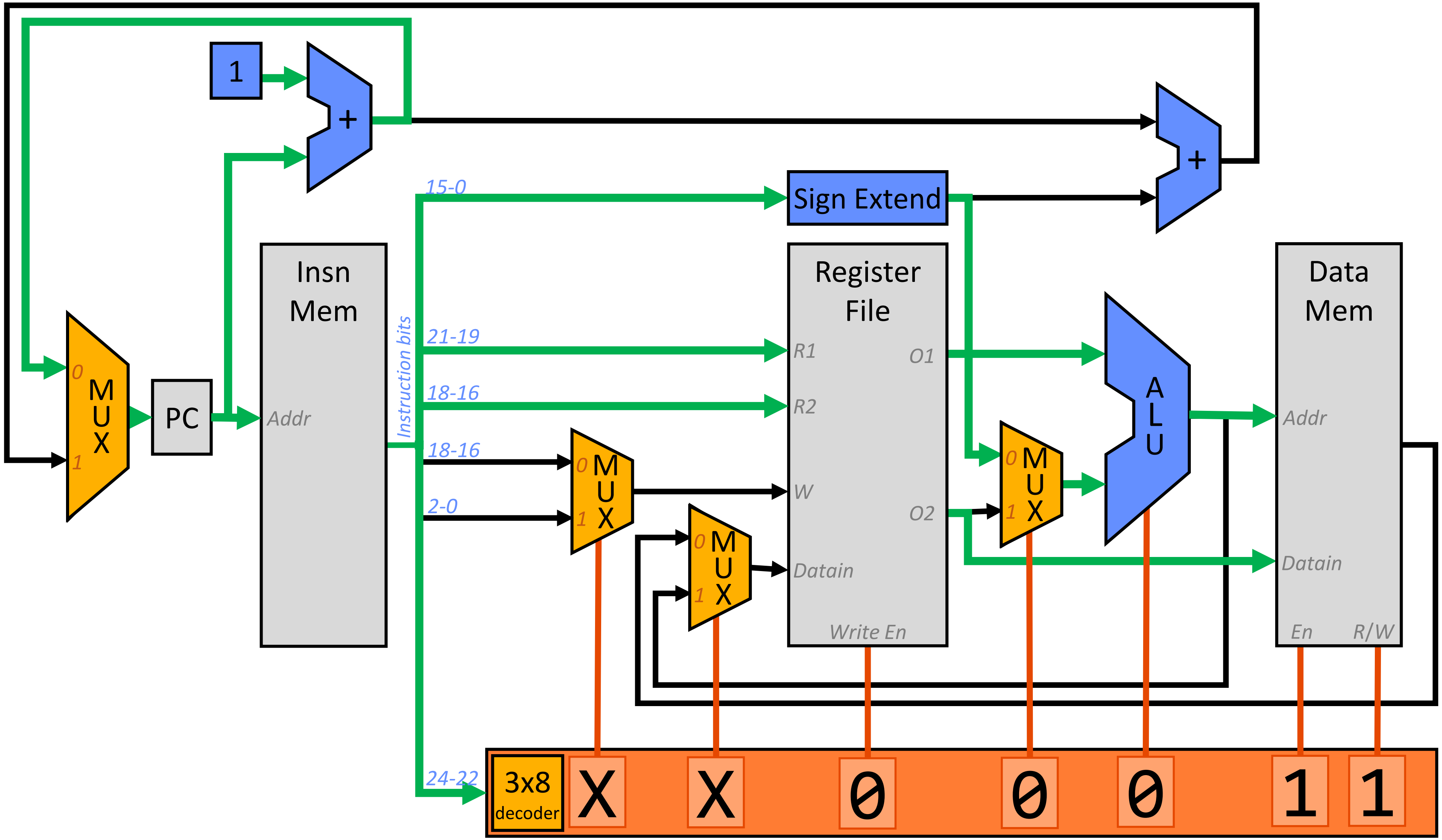
	31-25	24-22	21-19	18-16	15-0
I-Type		opcode	regA	regB	destR



SW	Instruction	sw regA, regB, offset
	Functionality	M[regA + offset] = regB; PC = PC+1
	I-Type	

31-25	24-22	21-19	18-16	15-0
	opcode	regA	regB	destR

LC2K Single-Cycle Processor



More instructions to come…

Next lecture!

Logistics

- There are 3 videos for lecture 10
  - L10_1 – Finite-State-Machines_Implementation
  - L10_2 – Single-Cycle-Processor
  - L10_3 – LC2K-Datapath
- There is one worksheet for lecture 10
  1. Finite state machine