

Memory

Andrew Lukefahr

Always specify
defaults for
always_comb!

BLOCKING (=) FOR

always_comb

NON-BLOCKING (<=) for

always_ff

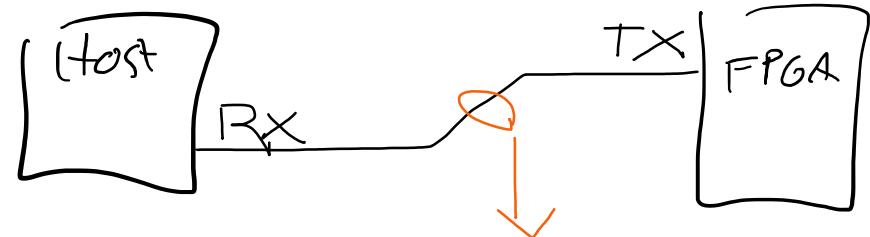
UART RX/TX LEDs on Basys3

- A word of **caution**:
- The Basys3's **RX + TX LEDs are backwards** from what you expect.
- They are the USB adaptor chip's RX+TX, not the FPGAs.

UART: TX

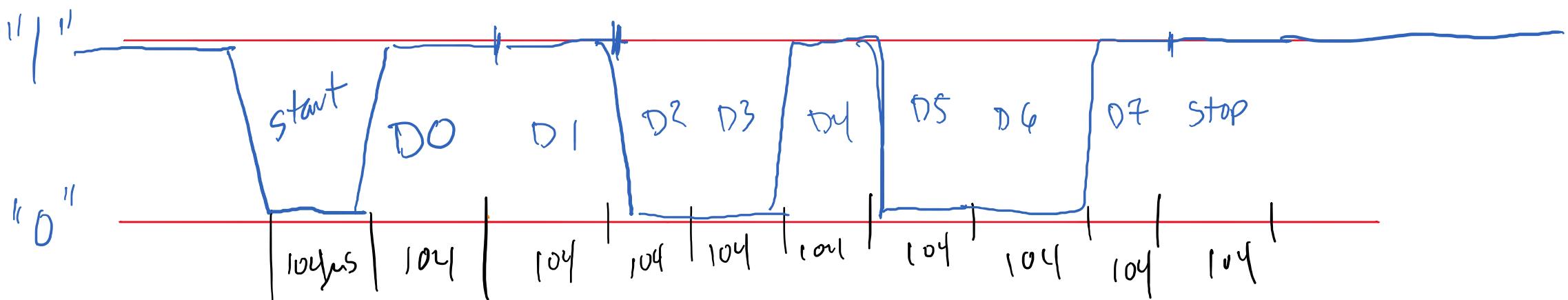
- How to transmit 8'b10010011?

MSB
↓
↑ ↑ T ↑
D7 D6 D1 D0
LSB



- Draw the packet!

- Hint: UART is transmitted LSB -> MSB



UART RX Frame Timing

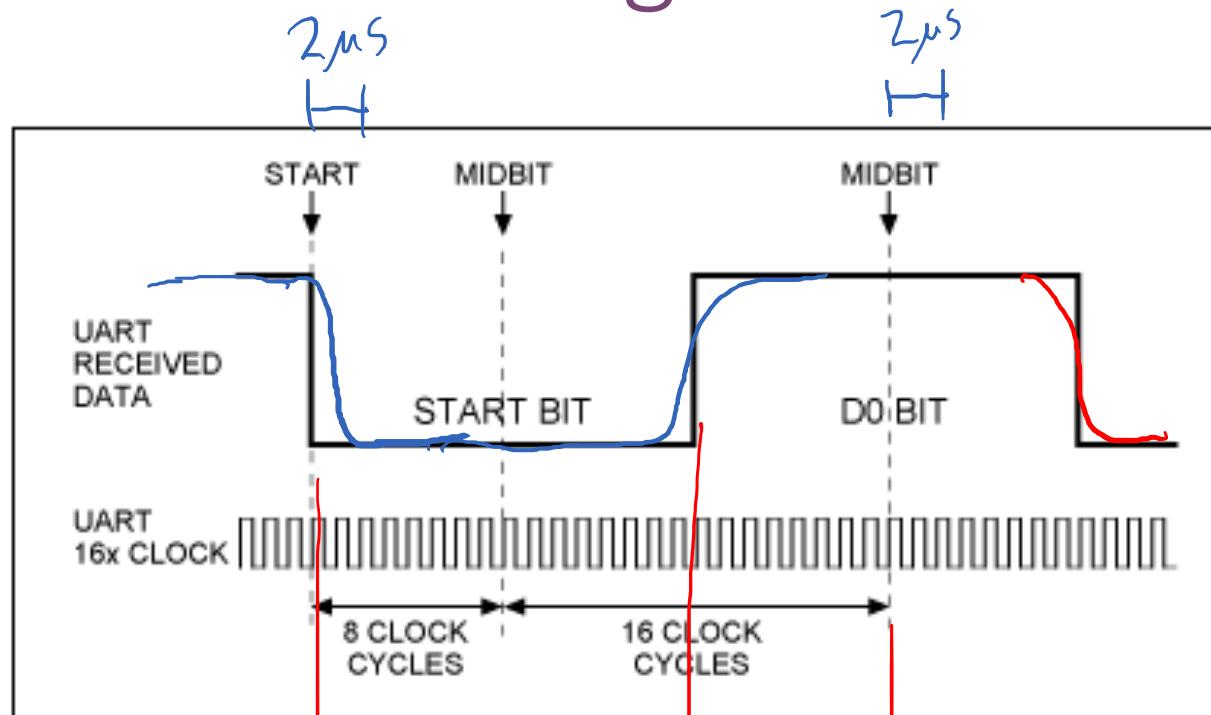


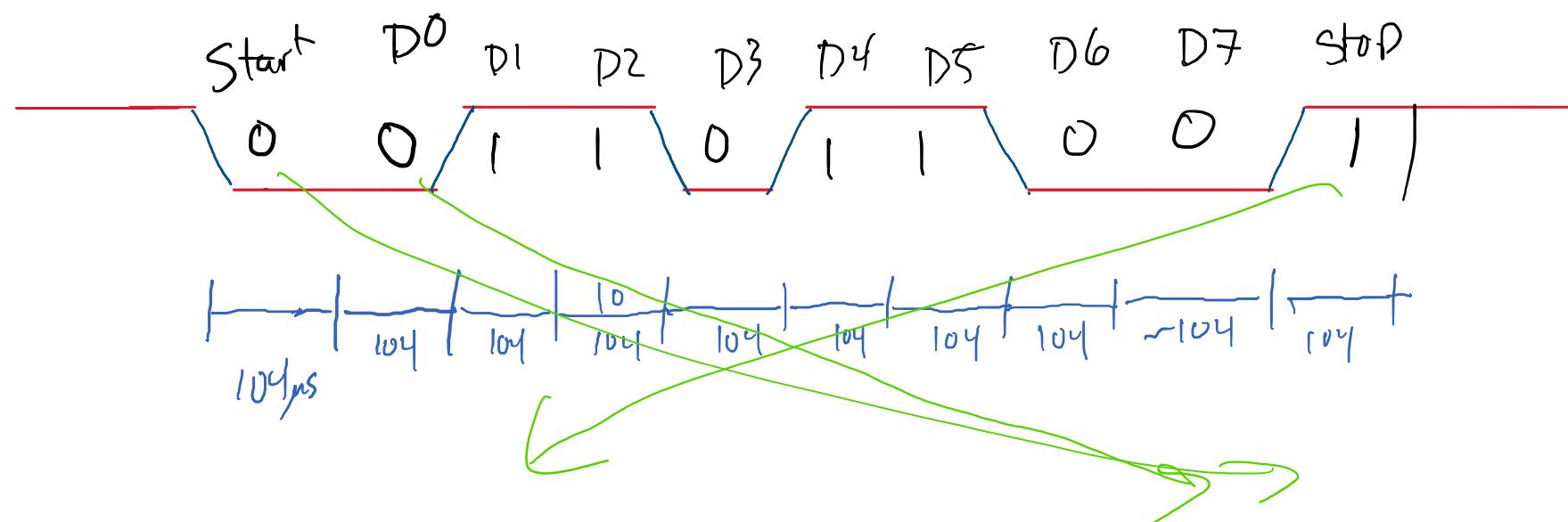
Figure 2. UART receive frame synchronization and data sampling points.

104 μ s | 57 μ s
150 μ s

UART RX

Scale $\frac{1}{104\mu s}$

- What data is this?
 - Recall: LSB first



$$8'b \ 0011\ 0110 = 8'h = 36$$

Simple Countdown Timer

```
timer tim0 (
    .clk(clk),
    .load(load),
    .data(data),
    .trigger(trigger)
);
```

```
module timer (
    input clk,
    input load,           // load-request
    input [31:0] data,    // <- 32-bit timer
    output trigger
);

logic [31:0] count;

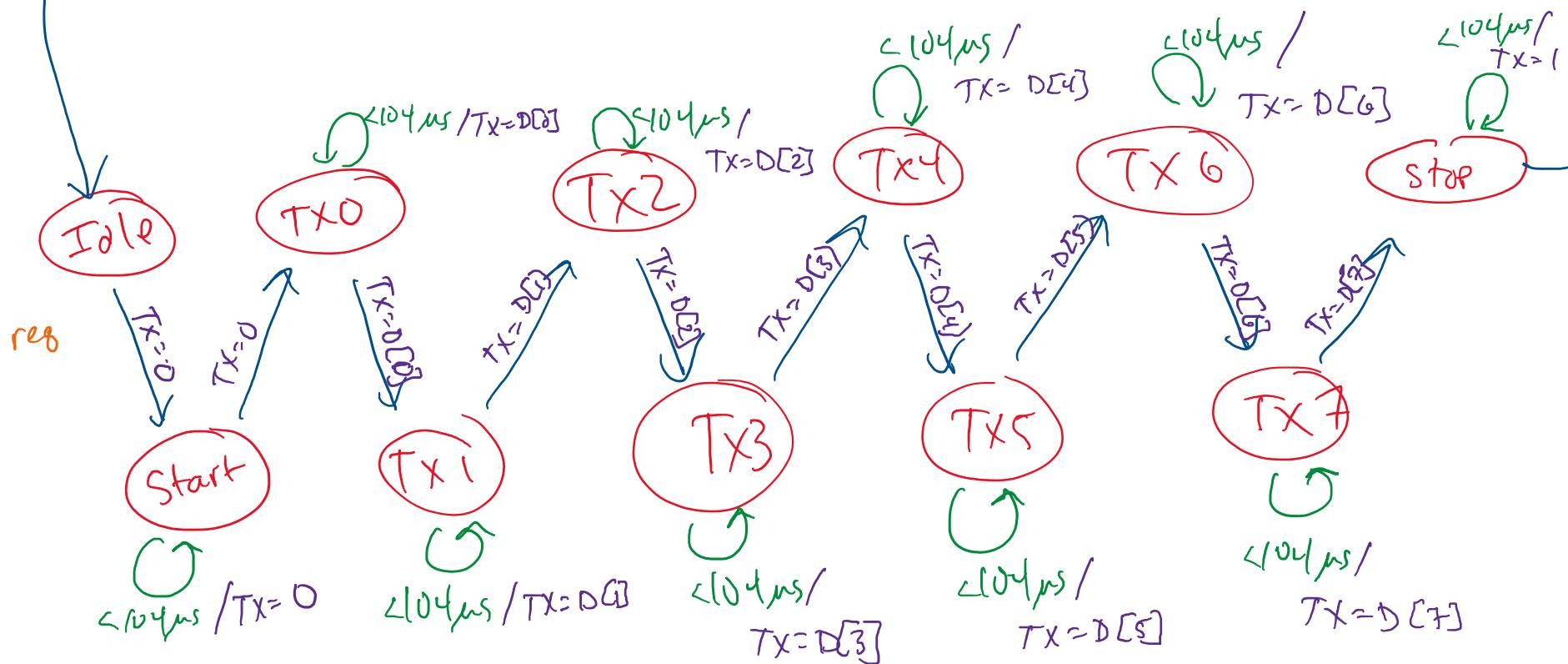
always_ff @ (posedge clk) begin
    if (load)      count <= data;
    else if (count != 0)
        count <= count - 32'h1;
end

assign trigger = (count == 0);

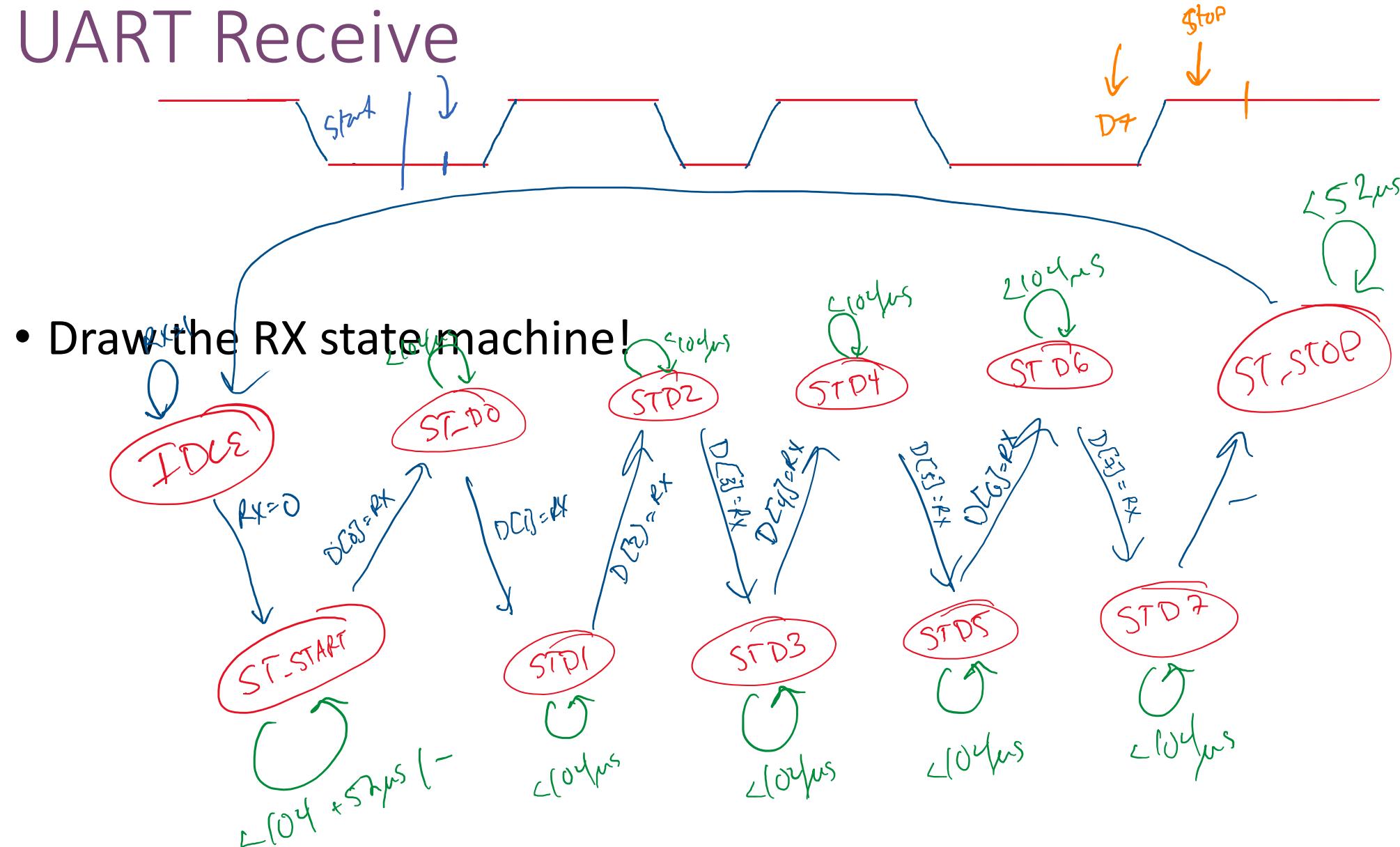
endmodule
```

UART: Transmit

- How to transmit 8'b01101010?



UART Receive



OPTIONAL: UART RX Shift Registers

- Rather than explicitly assign destination indexes, can also use a shift register

```
always_ff @(posedge clk) begin
    //other code here!
    if (rst)
        shift_reg <= 8'h0;
    else if (shift_in)
        shift_reg <= {in, shift_reg[7:1]};
    else //optional
        shift_reg <= shift_reg;
end
```

OPTIONAL: UART TX Shift Registers

- Rather than explicitly assign destination indexes, can also use a shift register

```
always_ff @ (posedge clk) begin
    //other code here!
    if (load)
        shift_reg <= load_value;
    else if (shift_out)
        shift_reg <= {1'h0, shift_reg[7:1]};
    else //optional
        shift_reg <= shift_reg;
end

assign out = shift_reg[0];
```

P5: UART

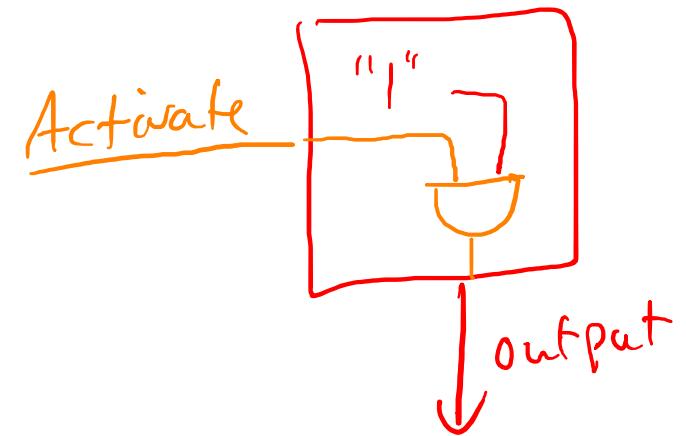
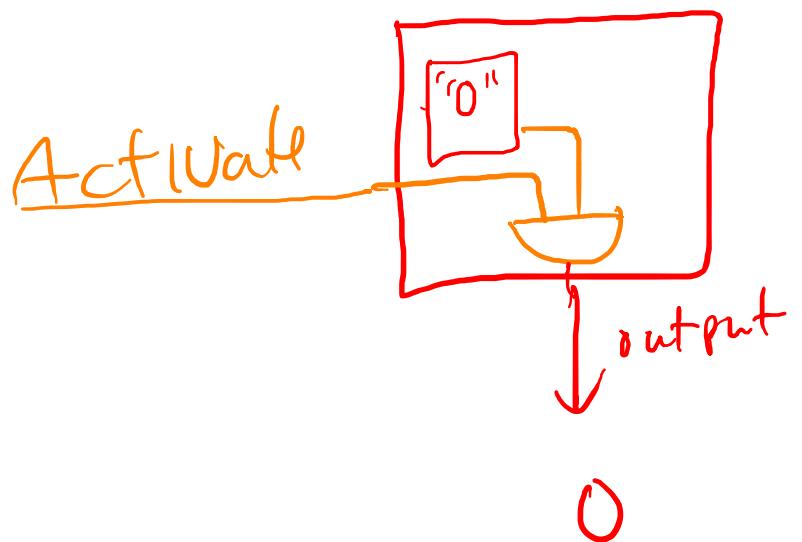
- You get to build a UART interface
- P5: just echo RX back over TX
- Connect your FPGA to your PC
- Allows you to “talk” to your FPGA with keyboard
- P6: we add python UART interface

Memory

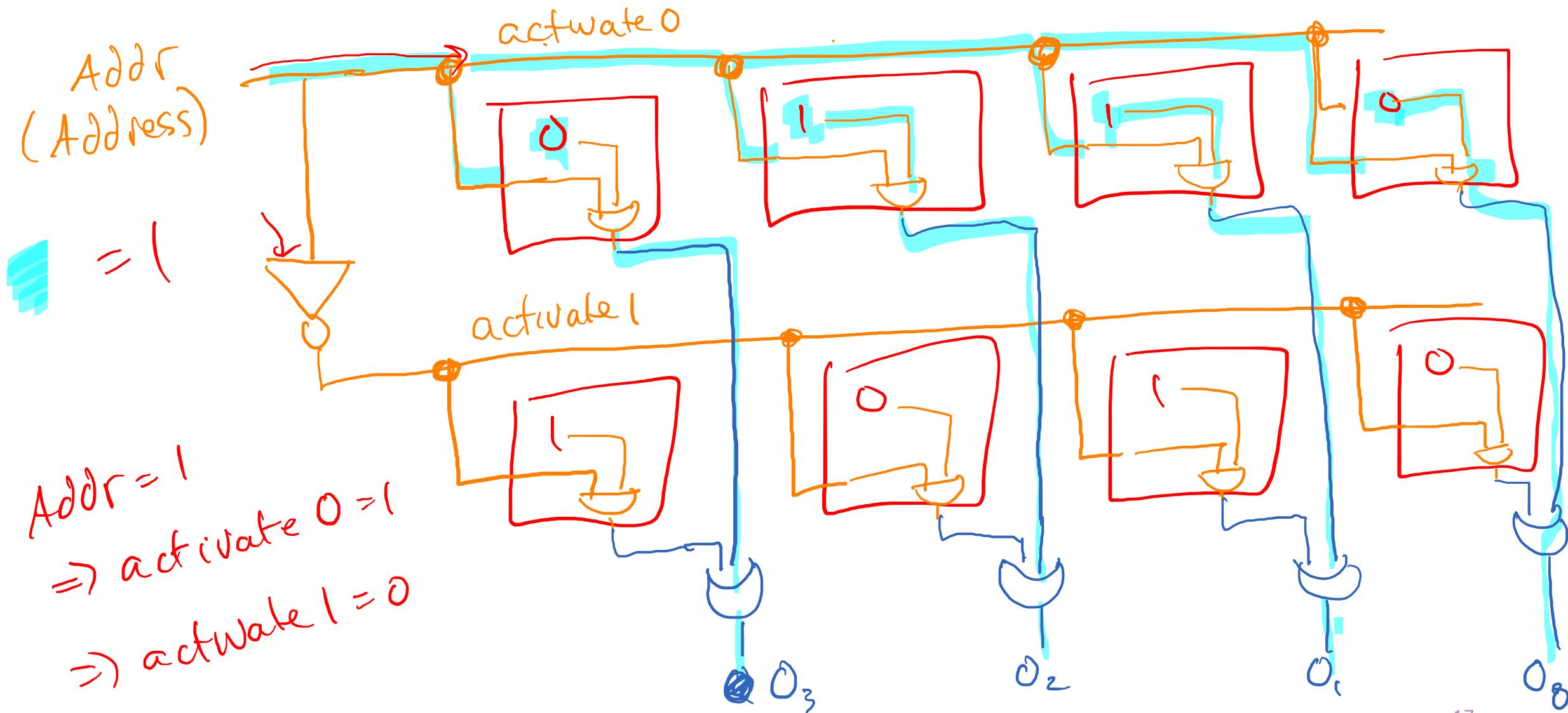
ROM vs RAM

- ROM – Read-Only Memory
 - Input: address
 - Output: fixed value
- RAM – Random-Access Memory
 - Read/Write version of a ROM

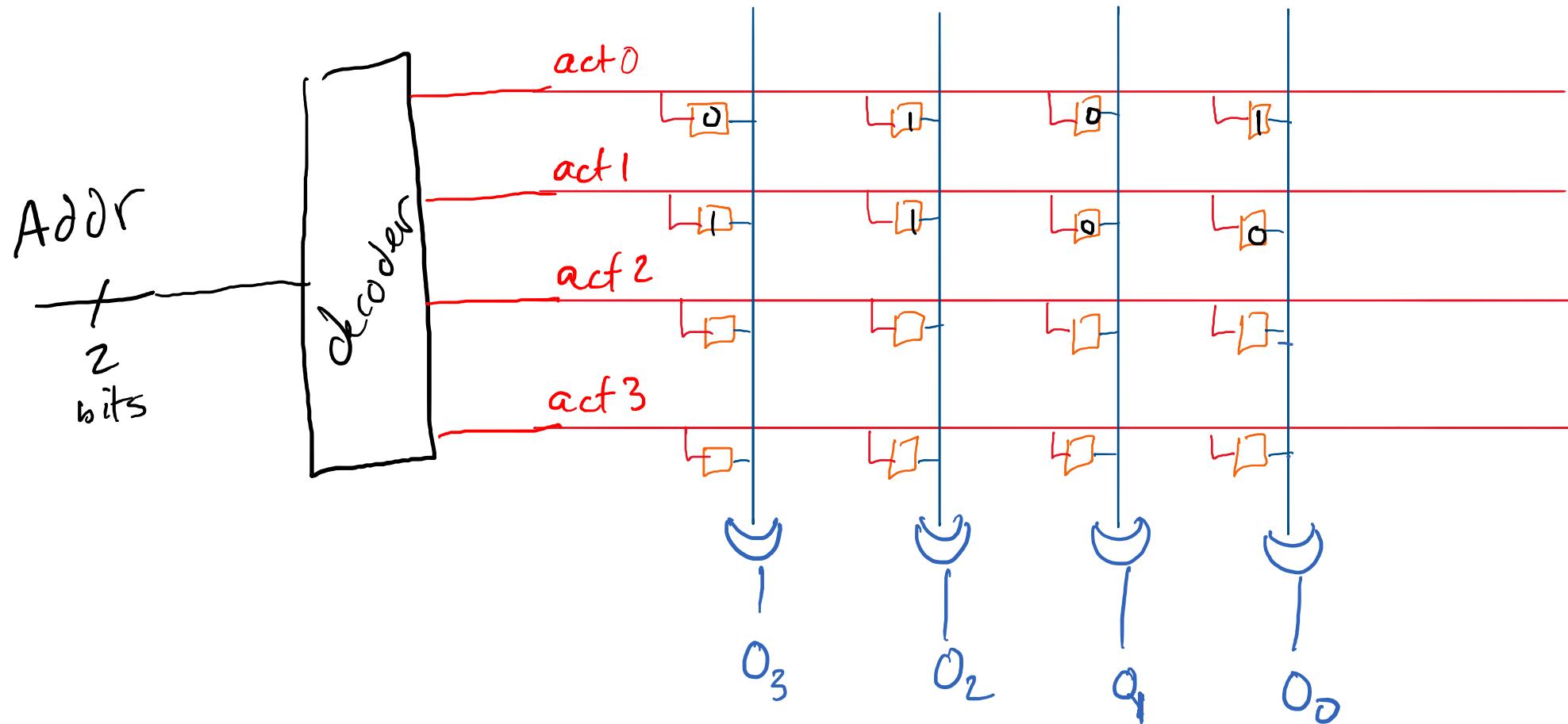
Rom Cell



Array of ROM Cells



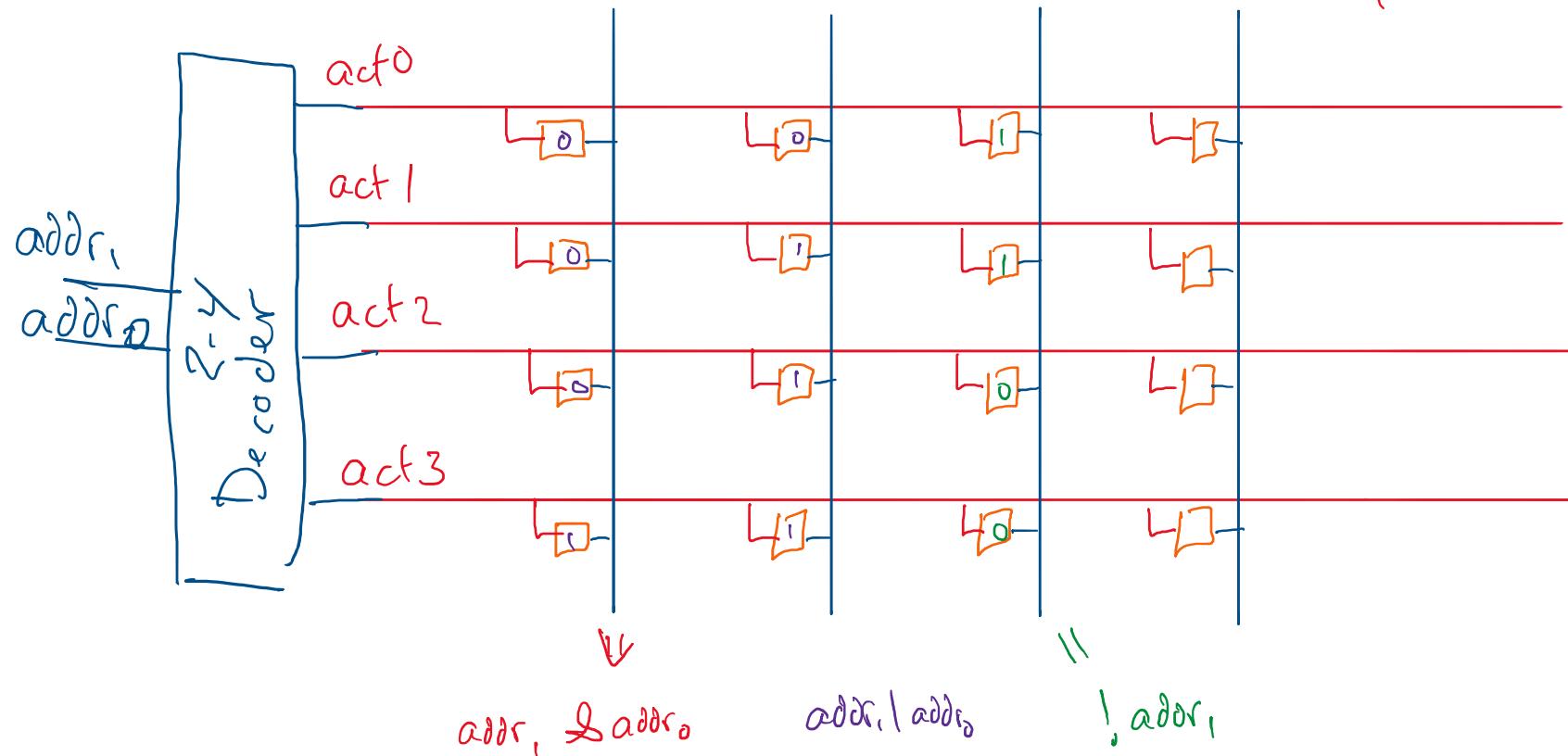
2-bit ROM



2-bit ROM of AND + OR

AND / OR

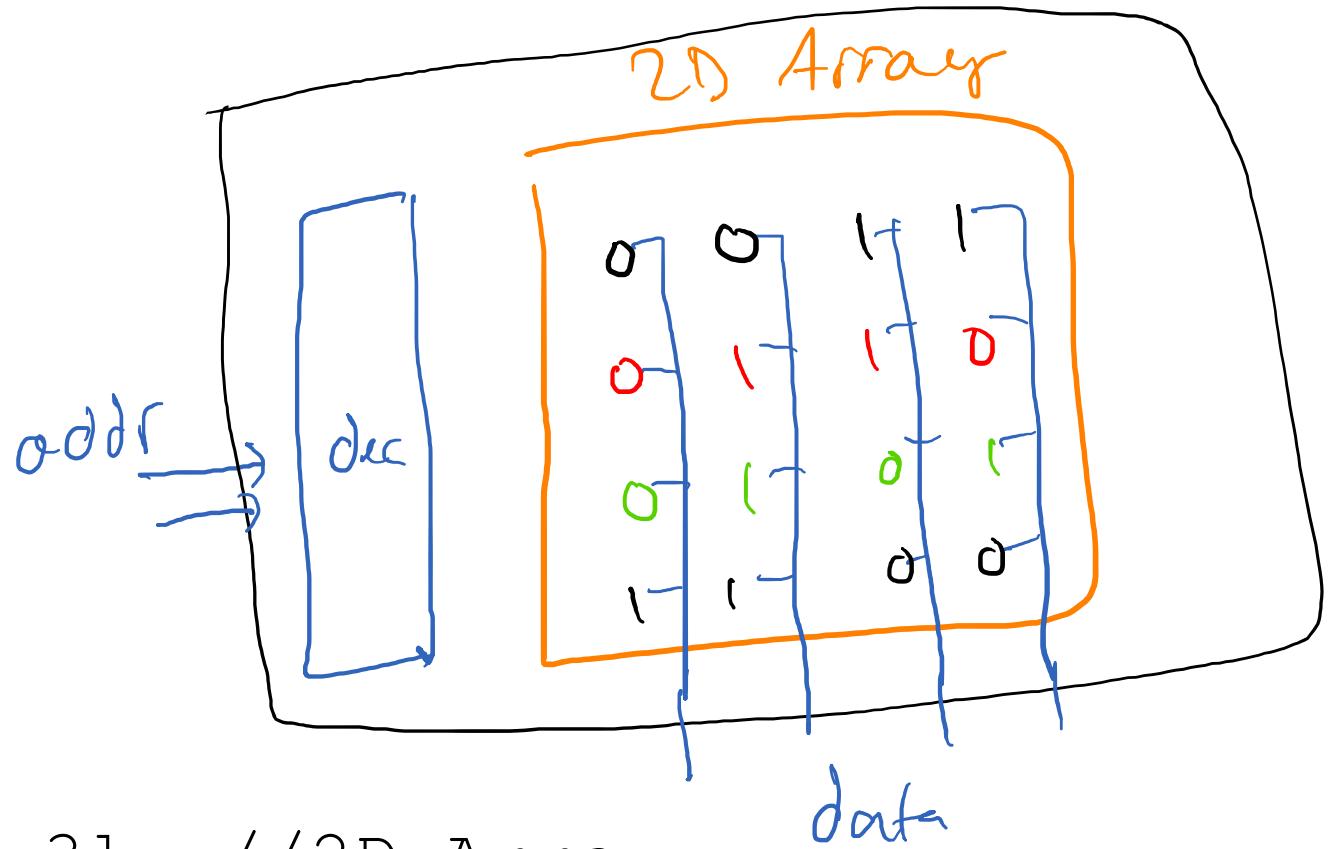
addr₁	addr₀	addr ₁	addr ₀	AND	OR
X	X			0	1
0	0	0	0	0	1
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	1



ROM in Verilog

```
module ROM (
    input [1:0] addr,
    output [3:0] data
)
```

```
logic [3:0] array [0:3]; //2D Array
assign array = { 4'b0011, 4'b0110, 4'b0101, 4'b1100 }
assign data = array[addr]; ← Select a row for output
endmodule
```

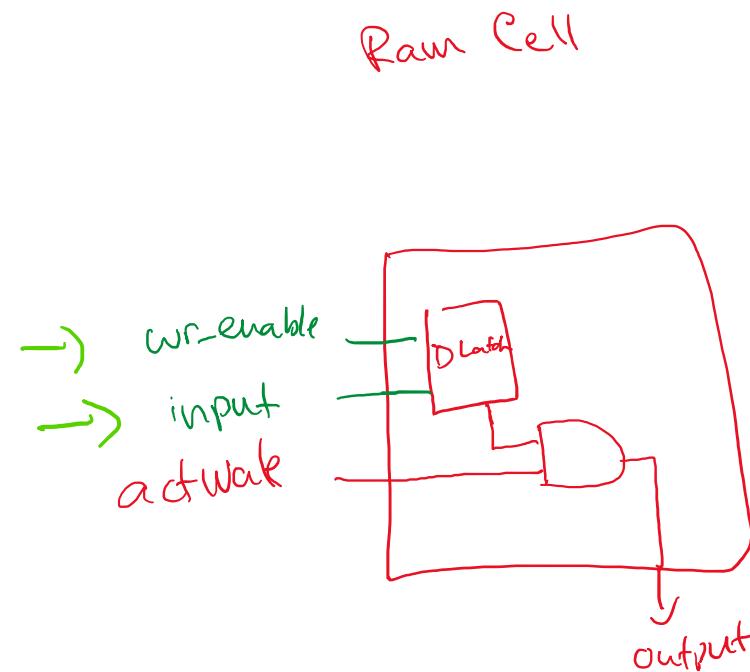
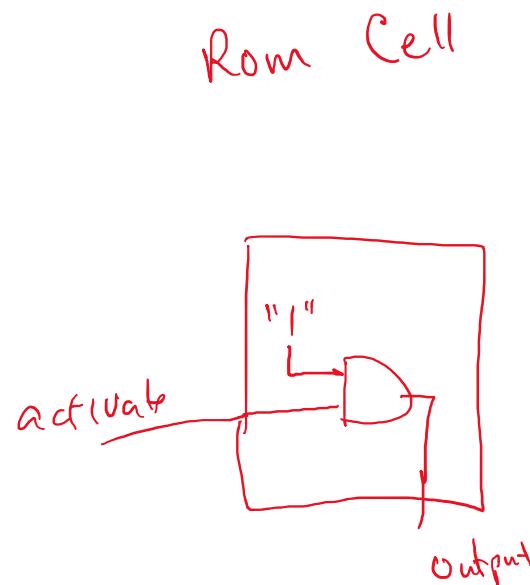


RAM

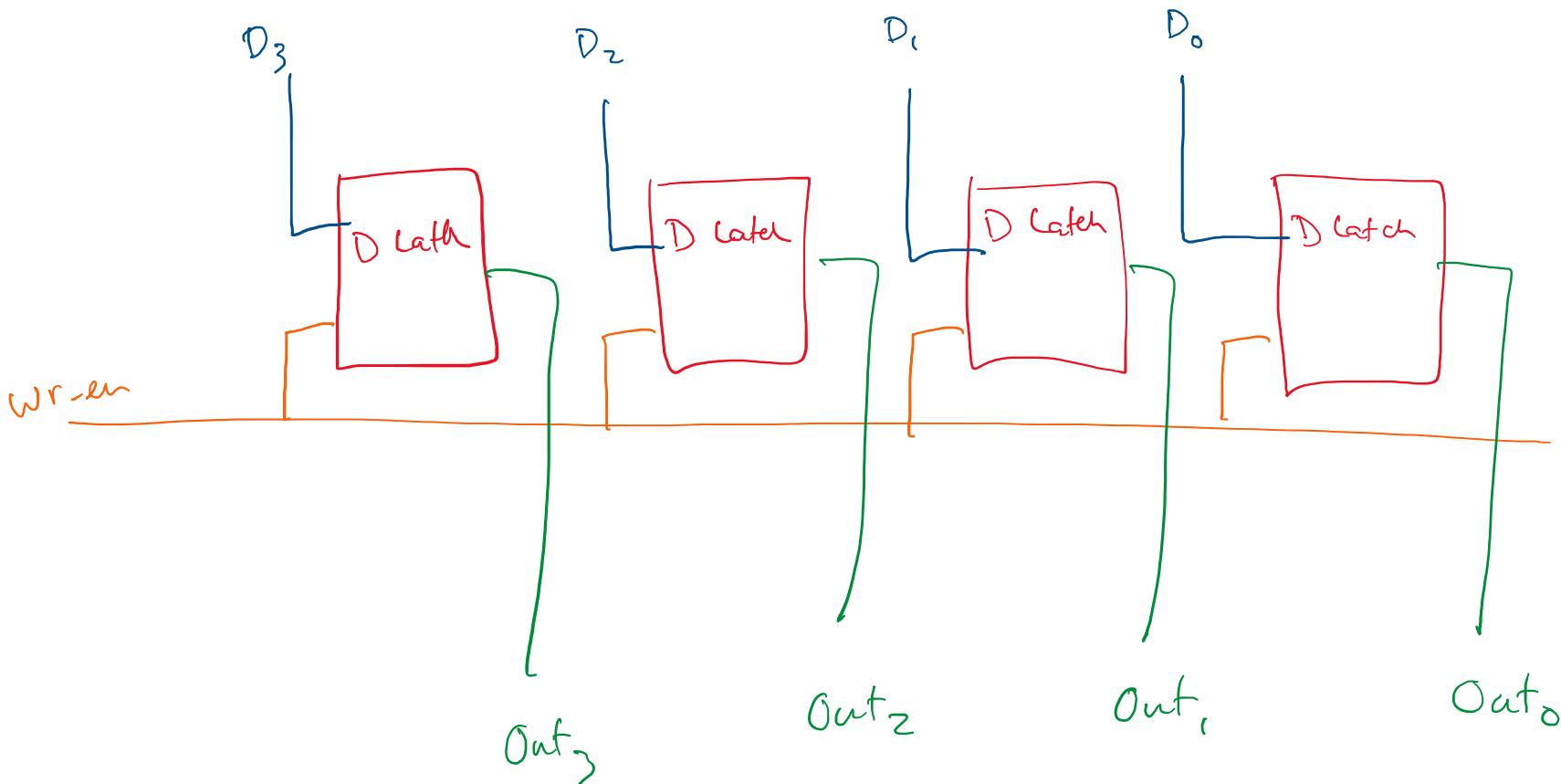
- Similar to ROM
- BUT WRITABLE!

RAM

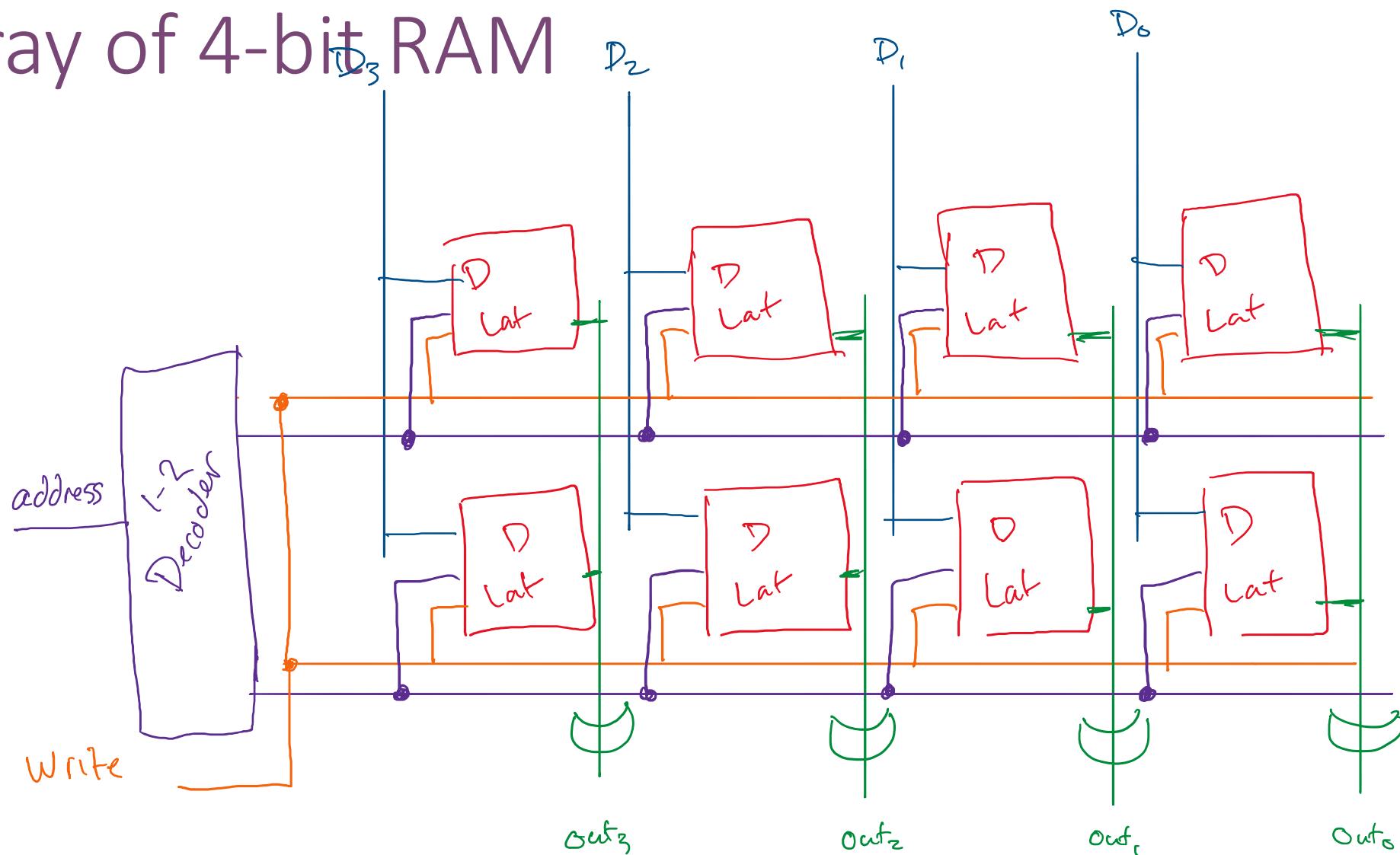
- Similar to ROM
- BUT WRITABLE!



4-bit RAM



Array of 4-bit RAM



Flip-Flop RAM in Verilog

```
module RAM (
    input      clk,
    input [1:0] addr,
    input      set,
    input [3:0] set_data,
    output [3:0] read_data
)
logic [3:0] array [0:3]; //2D Array

    assign read_data = array[addr];
endmodule
```

Flip-Flop RAM in Verilog

```
module RAM (
    input      clk,
    input [1:0] addr,
    input      set,
    input [3:0] set_data,
    output [3:0] read_data
)
    logic [3:0] array [0:3]; //2D Array
    always_ff @(posedge clk) begin
        if (set) array[addr] <= set_data;
    end
    assign read_data = array[addr];
endmodule
```

Aside: Latch RAM in Verilog

```
module RAM (                                ← does not need clk
    input [1:0] addr,
    input       set,
    input [3:0] set_data,
    output [3:0] read_data
)
    logic [3:0] array [0:3]; //2D Array
    always_latch begin //if you really want a latch
        if (set) array[addr] = set_data; ← not have default
    end
    assign read_data = array[addr];
endmodule
```

Any glitch on set will kill this.
Do not use in class!

Aside: SRAM vs DRAM

- SRAM:
 - Static RAM
 - What we've discussed so far
 - Uses full SR Latch to maintain value
- DRAM:
 - Dynamic RAM
 - Charges capacitor to store charge
 - Requires active “refresh” circuit

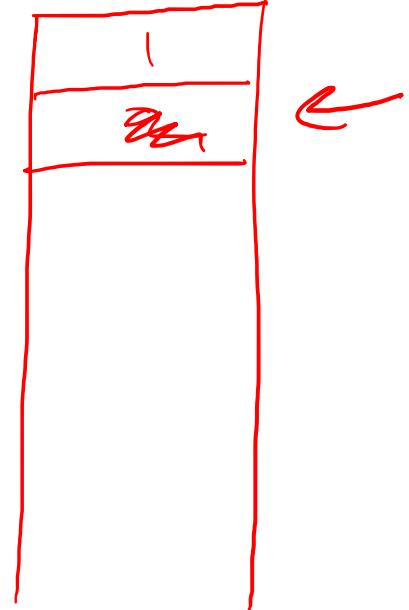
Aside: SRAM vs DRAM

- SRAM:
 - Uses full SR Latch to maintain value
 - + Easier
 - Bigger cells
 - Higher power
- DRAM:
 - Charges capacitor to store charge
 - + Smaller cells, higher density
 - Requires sophisticated read and “refresh” circuits

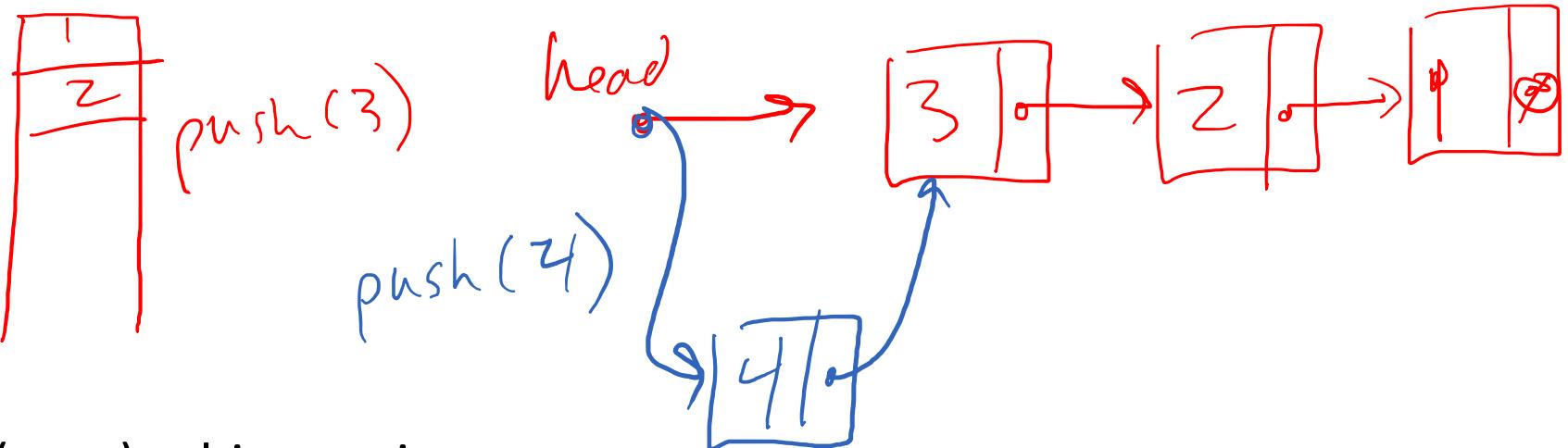
Stacks

- First-In-Last-Out data structure
- Defines two operations:
 - $\text{push}(x)$: adds an element to the end of the stack
 - $X = \text{pop}()$: returns most recently-added element from the stack

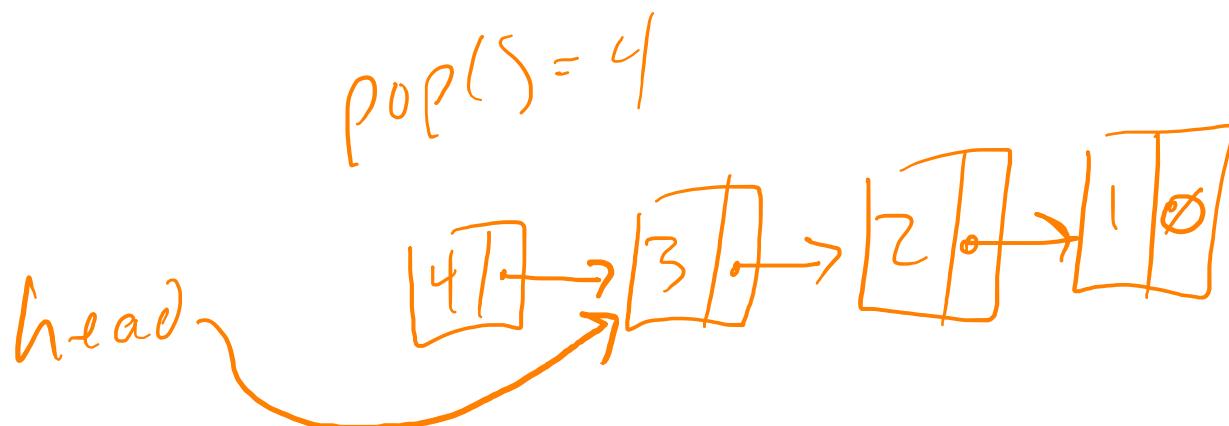
push(1)
push(2)
 $\text{pop}() \rightarrow 2$



Stacks

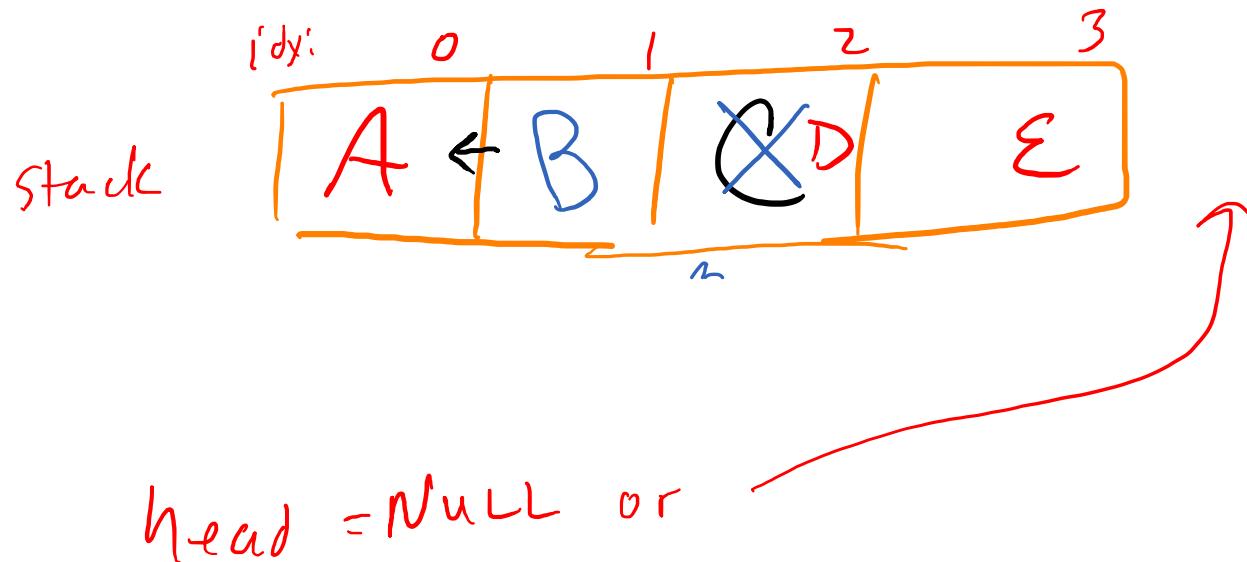


- In C/C++:
 - Can grow to (near) arbitrary sizes
 - Implemented with linked lists
 - `malloc()` allows more memory for bigger stacks
- In Hardware:
 - Don't have `malloc()`
 - Can't get "more gates"
 - Fixed size!



Fixed-Size Stacks

- Use an array as a fixed-size stack



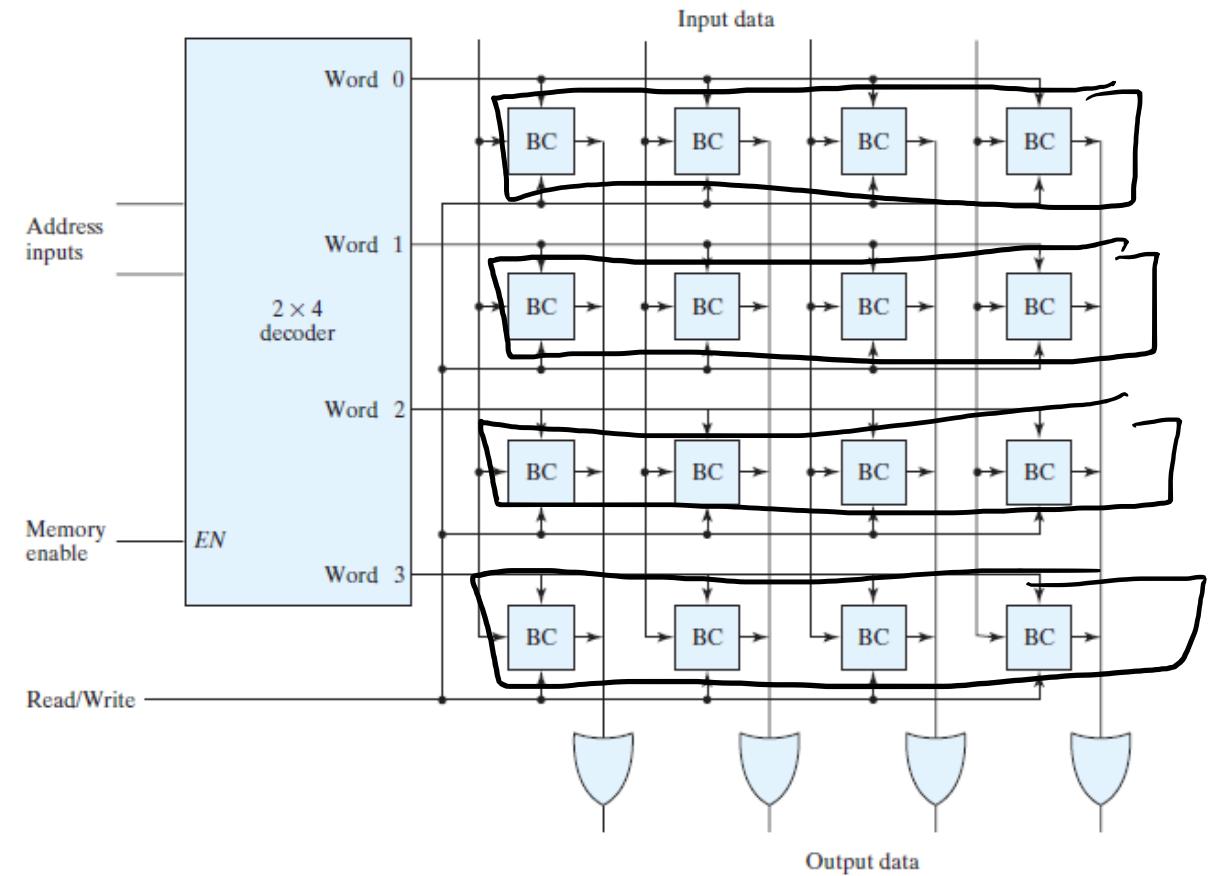
head = 2



push(A)
push(B)
push(C)
pop()
push(D)
push(E)

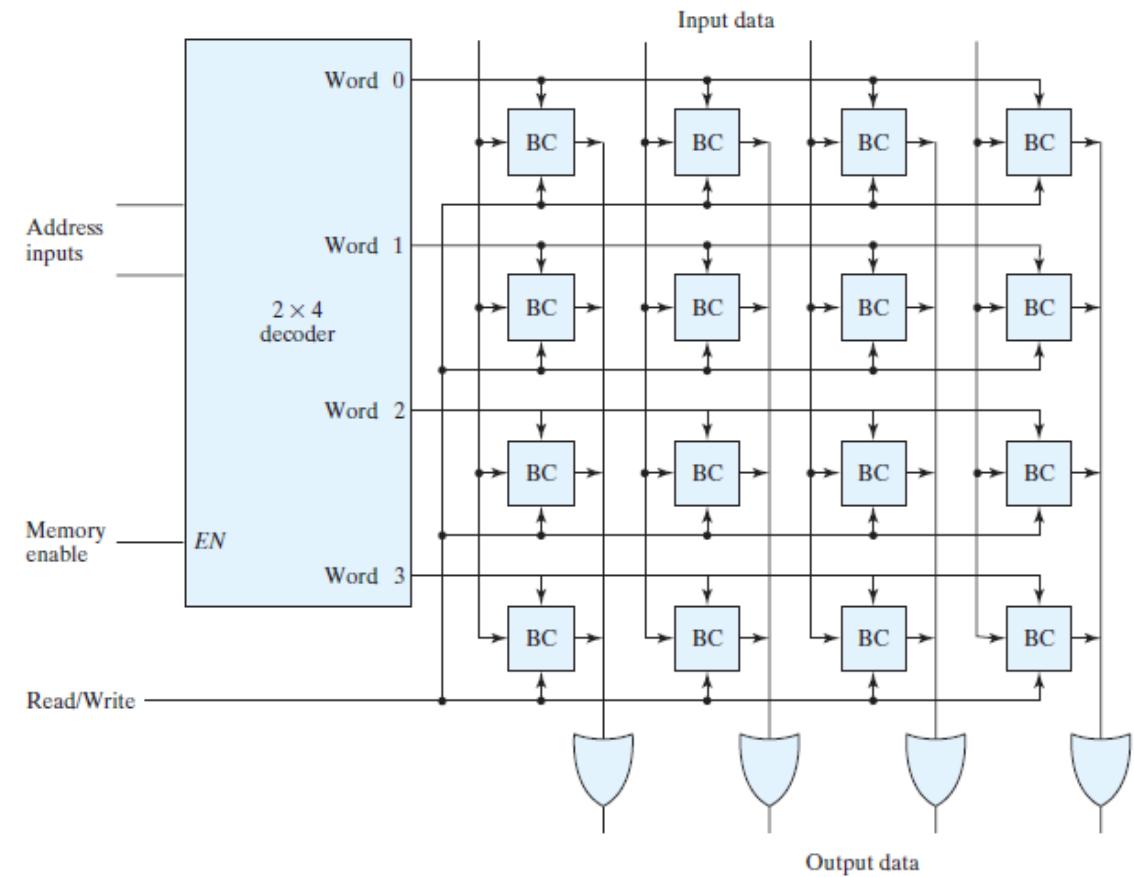
Fixed-Size Stack in Hardware

- We can use a RAM block as a stack
- Just need to add head index



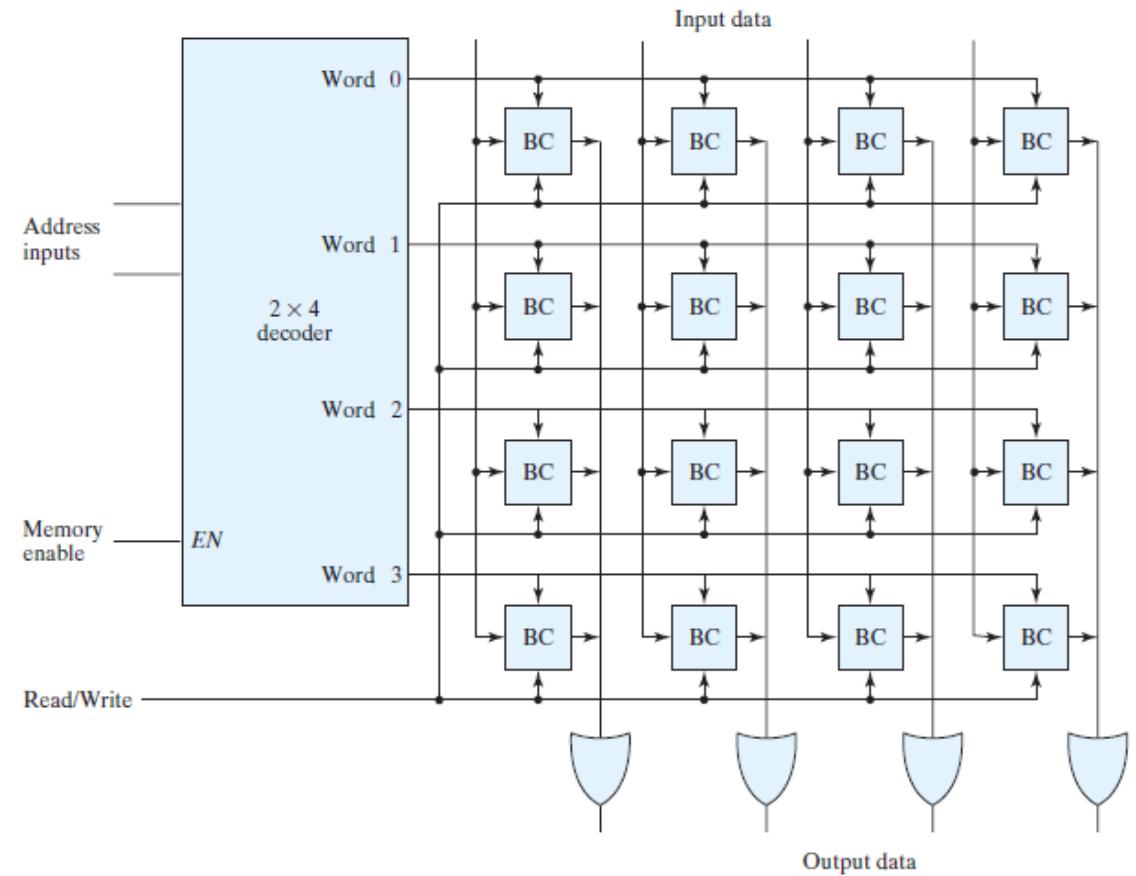
Stack with RAMs

- Given: RAM array (shown)
- Make: 4-element 4-bit **Stack**
 - Recall: First-In-Last-Out
- Tip: Use a state machine!



Stack with RAMs

- Two stack “functions”
- push:
 - Adds element to stack
 - $\text{push}(\ 4'b\ \text{XXXX})$
- pop:
 - Removes element from stack
 - $4'b\text{XXXX} = \text{pop}()$



Stack with RAMs

`push(4'b 0001)`

head = 00, input = 0001, RDWR = 0, memEn = 1

head <= head + 1

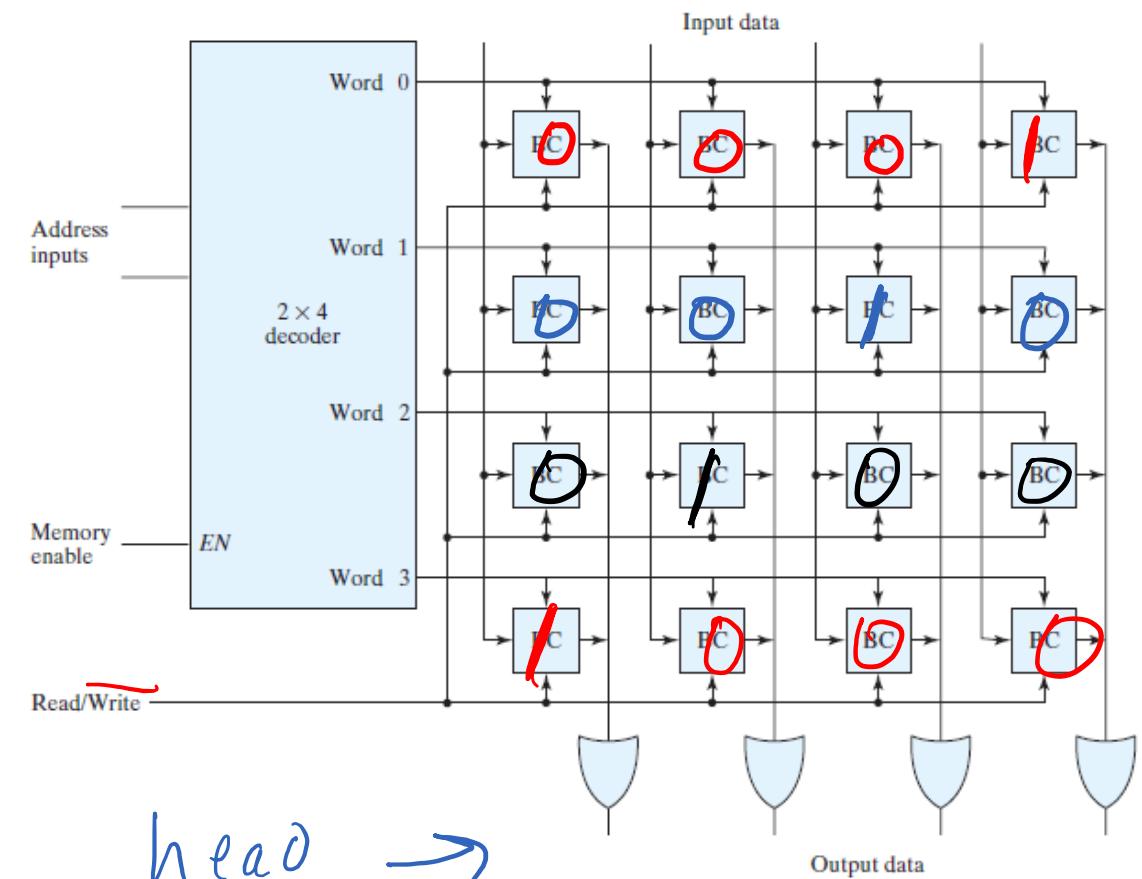
`push(4'b 0010)`

head = 01, input = 0010, RDWR = 0, memEn = 1

head <= head + 1

`push(4'b 0100)`

`push(4'b 1000)`



*head →
100*

push/pop with RAMs

push(4'b 0001) ✓

push(4'b 0010) ✓

push(4'b 0100) ✓

pop() \Rightarrow 0100

pop() \Rightarrow 0010

push(4'b 1000) ✓

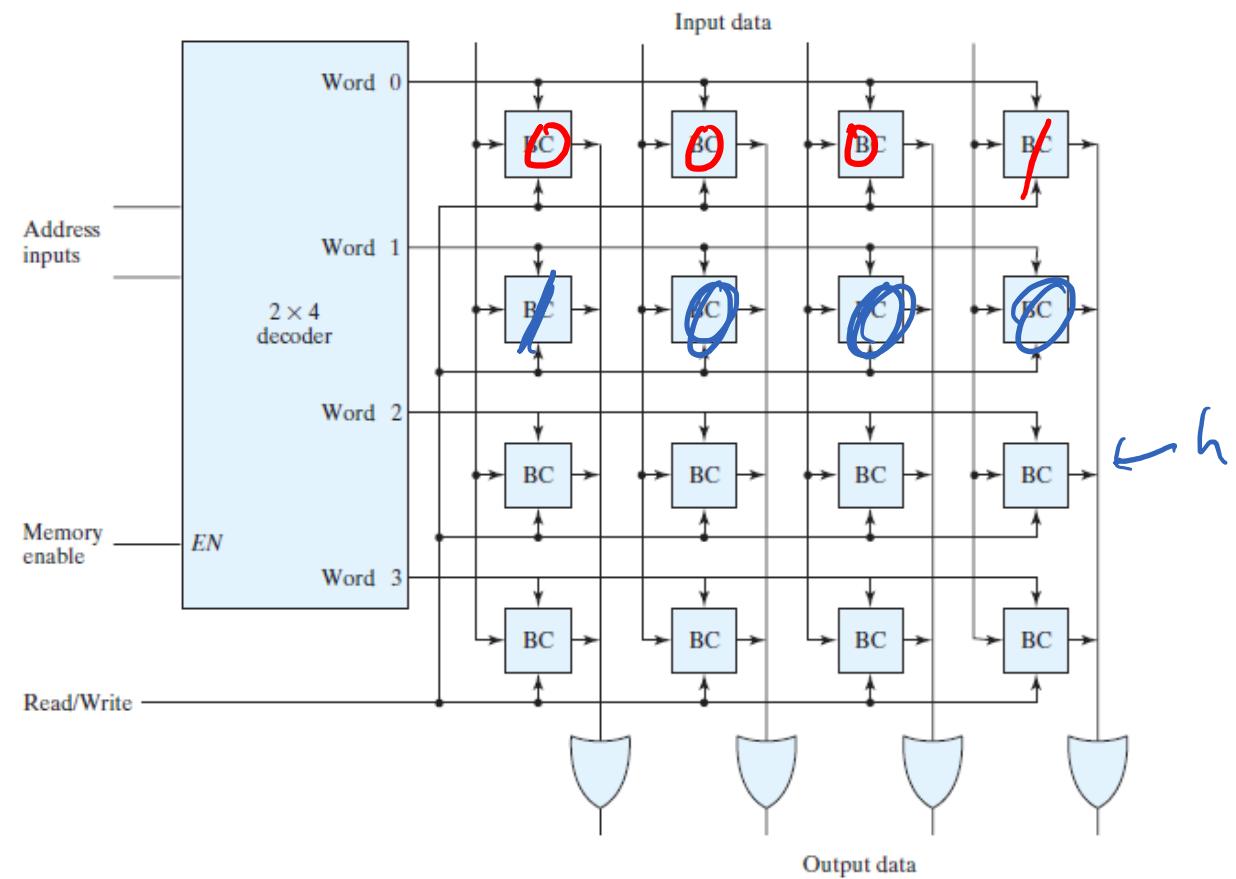
push(4'b 0011)

pop()

pop()

push(4'b 0110)

pop()



Stack Logic

- **Inputs:** push_req, [3:0] push_data
- **Inputs To RAM:** addr, set, [3:0] set_data

```
module RAM (
    input          clk,
    input [1:0]    addr,
    input         set,
    input [3:0]   set_data,
    output [3:0]  read_data
)
```

Push State Machine

```
assign pop-data =
```

```
always_ff (@posedge clk) begin  
    if (rst) ...  
    else begin
```

end

end

```
module RAM (  
    input          clk,  
    input [1:0]    addr,  
    input          set,  
    input [3:0]    set_data,  
    output [3:0]   read_data  
)
```

Pop Logic

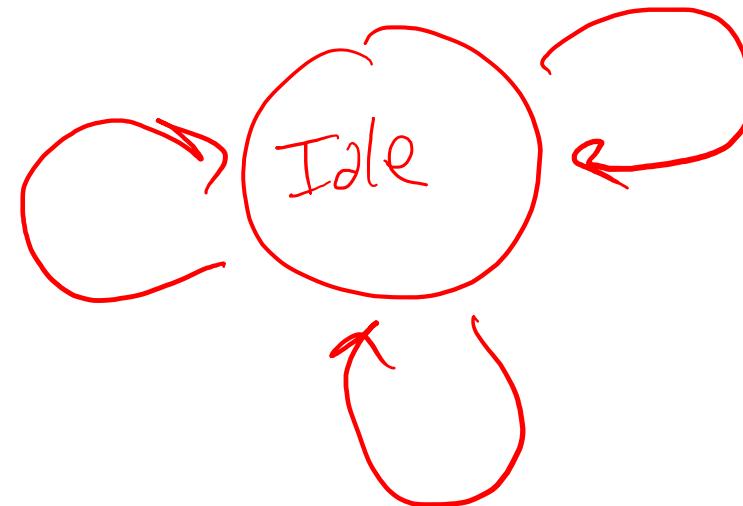
- **Inputs:** pop_req
- **Outputs:** [3:0] pop_data
- **Inputs To RAM:** addr, set
- **From RAM:** [3:0] read_data

```
module RAM (
    input          clk,
    input [1:0]    addr,
    input         set,
    input [3:0]   set_data,
    output [3:0]  read_data
)
```

Stack

~~Pop State~~ Machine

~~push & pop~~



~~push & pop~~

$\text{head} \leftarrow \text{head} + 1$
 $\text{output} = \text{mem}[\text{head}-1]$
 mem

~~push & '~~

$\text{head} \leftarrow \text{head}$

$\text{output} = \text{@mem}[\text{head}-1]$ (comb)

$\text{mem}[\text{head}-1] \leftarrow \text{input}$ (@clk)

~~(@clk)~~

Challenge: Push+Pop Together

- This needs to be a “replace” in the RAM.

0	0	0	1
0	0	1	0

pop() + push(0100)



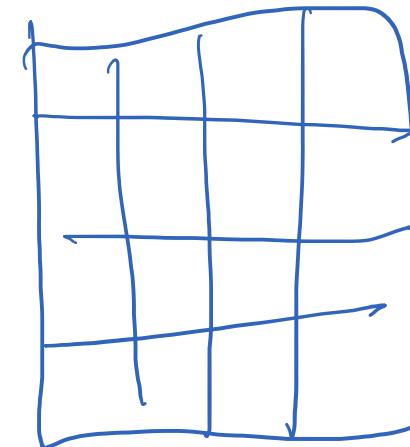
0	0	0	1
0	1	0	0

Challenge: Push+Pop Error Logic

- What happens if the RAM is empty? Or Full?

0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

push → fail
pop → succeed
push + pop → succeed



push → succeed
pop → fail
push + pop →
push-err = 0
pop-err = 1

Next Time

- FPGA Structures