

ENGR 210 / CSCI B441
“Digital Design”

UART III

Andrew Lukefahr

Always specify
defaults for
always_comb!

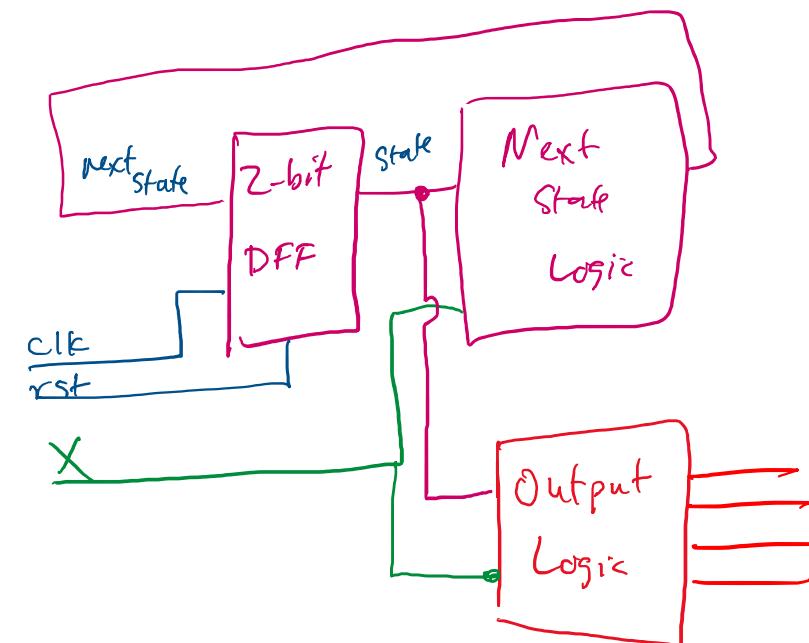
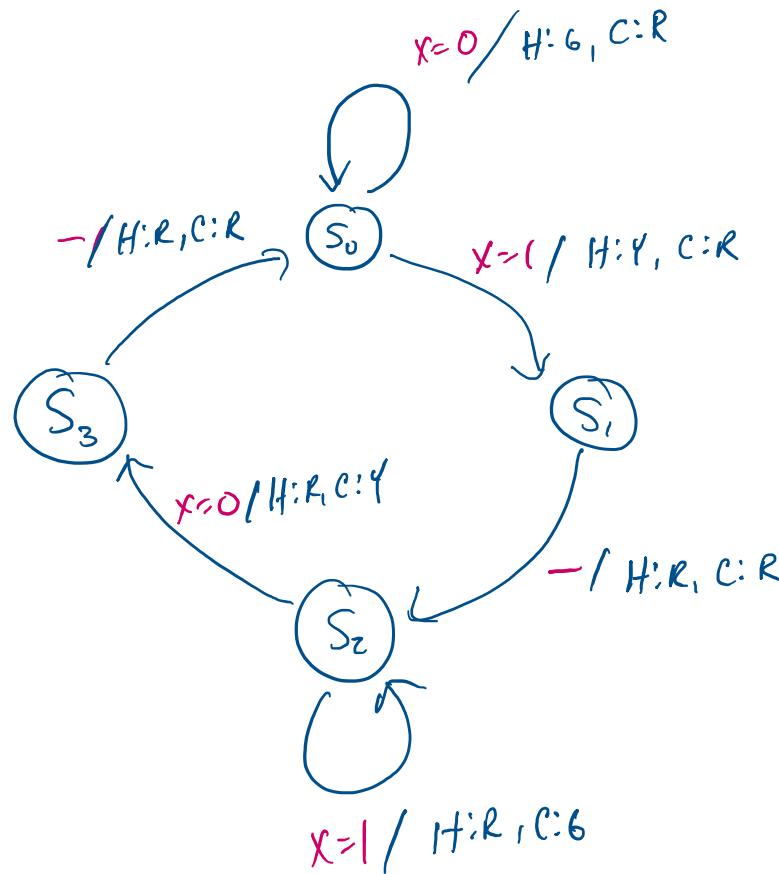
BLOCKING (=) FOR

always_comb

NON-BLOCKING (<=) for

always_ff

State Machine to Logic



```

module traffic(
    input clk,
    input rst,
    input x,
    output logic [2:0] Hryg, //red-yellow-green
    output logic [2:0] Cryg //red-yellow-green
);

enum { ST_0, ST_1, ST_2, ST_3 } state, nextState;

always_ff @(posedge clk) begin
    if (rst) state <= ST_0;
    else      state <= nextState;
end

always_comb begin
    nextState = state; //default
    Hryg = 3'b001; Cryg = 3'b100;

    case (state)

        ST_0: begin
            if (x) begin
                nextState = ST_1;
                Hryg = 3'b010;
                Cryg = 3'b100; //opt
            end else begin //opt
                nextState = ST_0; //opt
                Hryg = 3'b001; //opt
                Cryg = 3'b100; //opt
            end
        end
    end
end

```

```

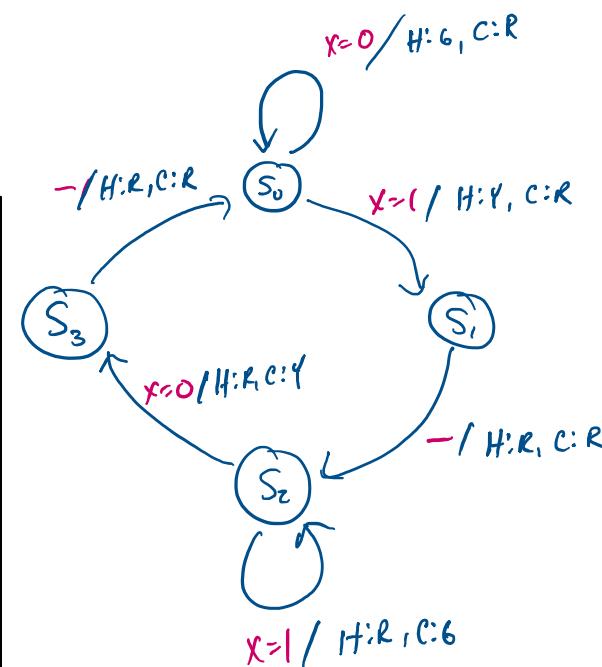
ST_1: begin
    nextState = ST_2;
    Hryg = 3'b100;
    Cryg = 3'b100; //opt
end

ST_2: begin
    if (x) begin
        nextState = ST_2;
        Hryg = 3'b100;
        Cryg = 3'b001;
    end else begin
        nextState = ST_3;
        Hryg = 3'b100;
        Cryg = 3'b010;
    end
end

ST_3: begin
    nextState = ST_0;
    Hryg = 3'b100;
    Cryg = 3'b100; //opt
end

endcase
end
endmodule

```



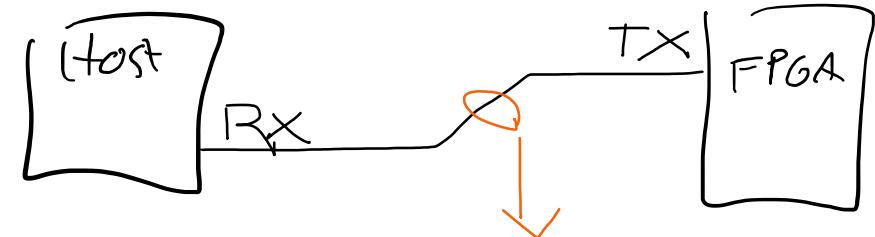
UART RX/TX LEDs on Basys3

- 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

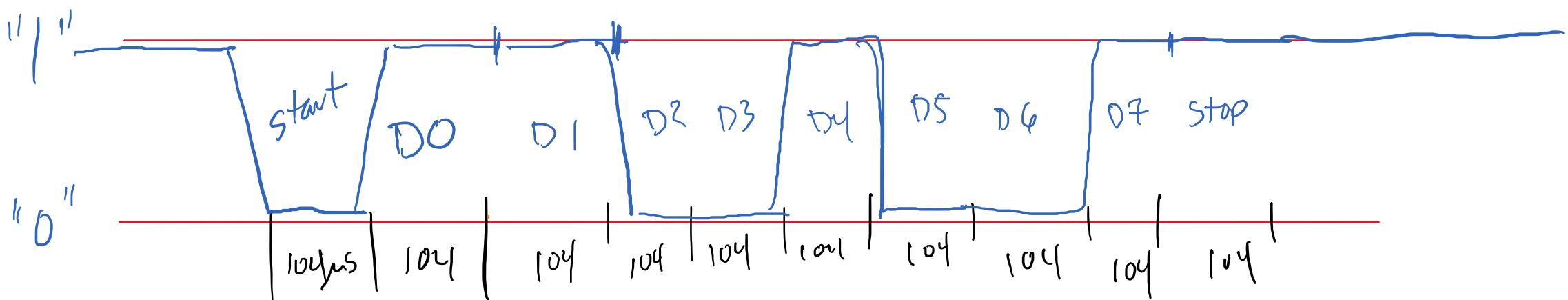
- How to transmit 8'b10010011?

MSB
↓
↑ ↑ T ↑
D7 D4 D1 D0
LSB



- Draw the packet!

- Hint: UART is transmitted LSB -> MSB



UART: TX

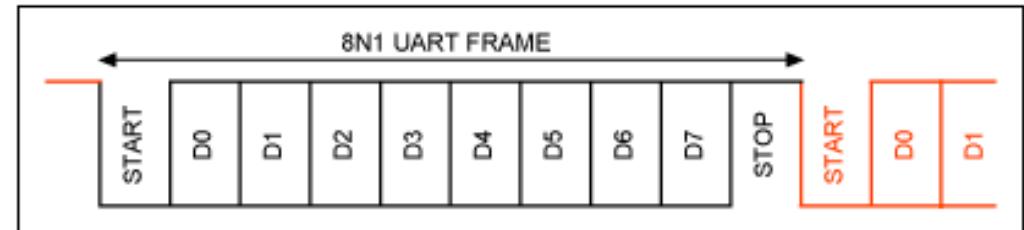
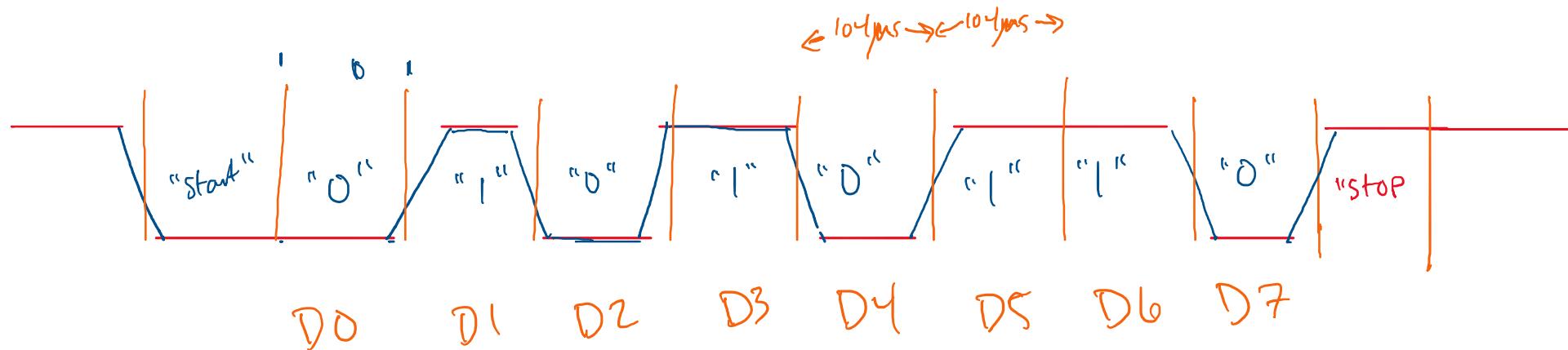


Figure 1. A typical UART data frame.

- How to transmit 8'b01101010?
- Draw the packet!
 - Hint: UART is transmitted LSB -> MSB



UART RX Frame Timing

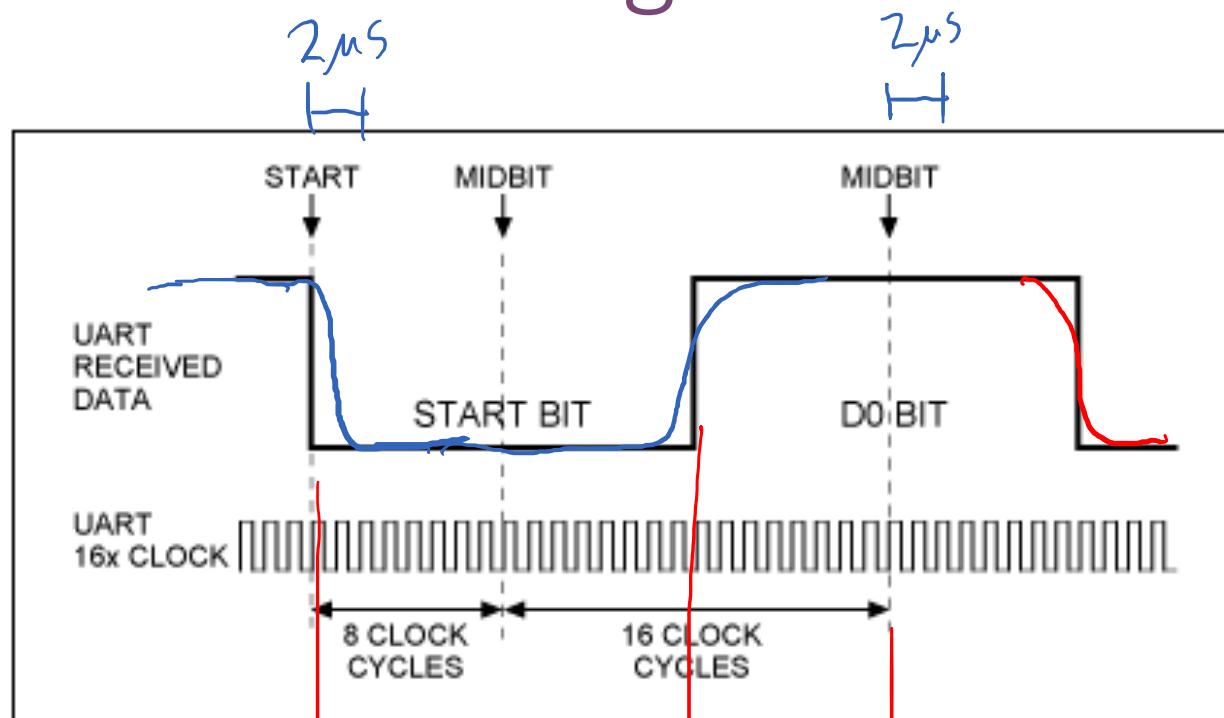


Figure 2. UART receive frame synchronization and data sampling points.

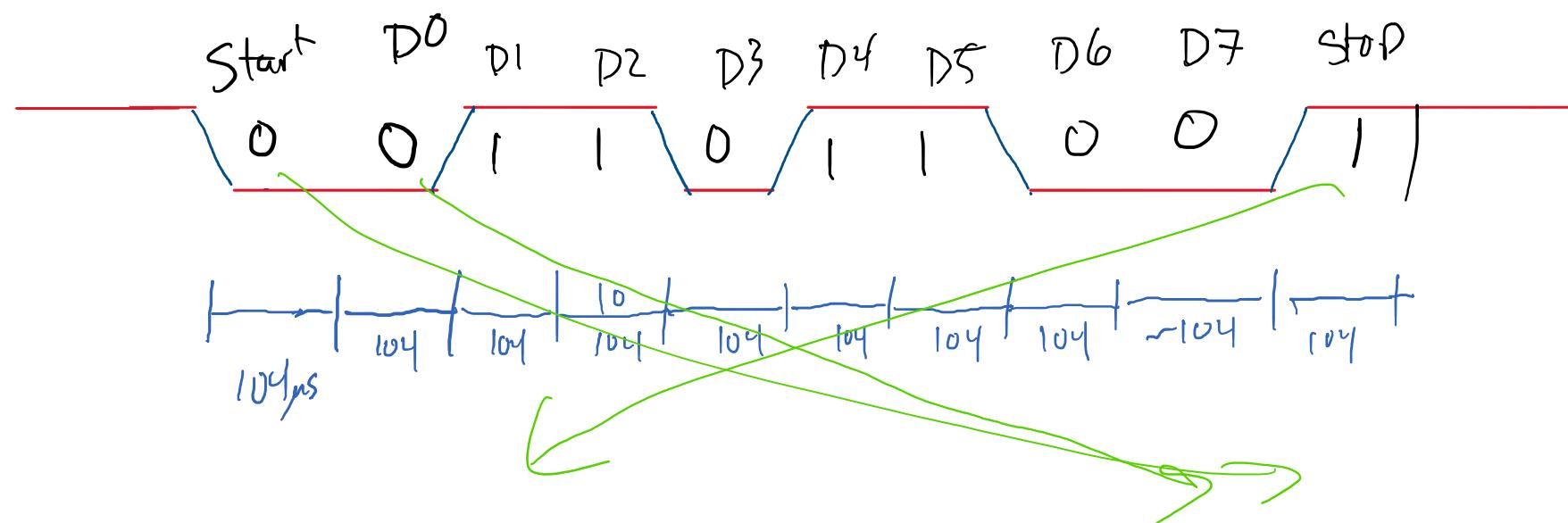
$104\mu s$ | $57\mu s$

$150\mu s$

UART RX

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

- What data is this?
 - Recall: LSB first



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

UART RX Frame Timing

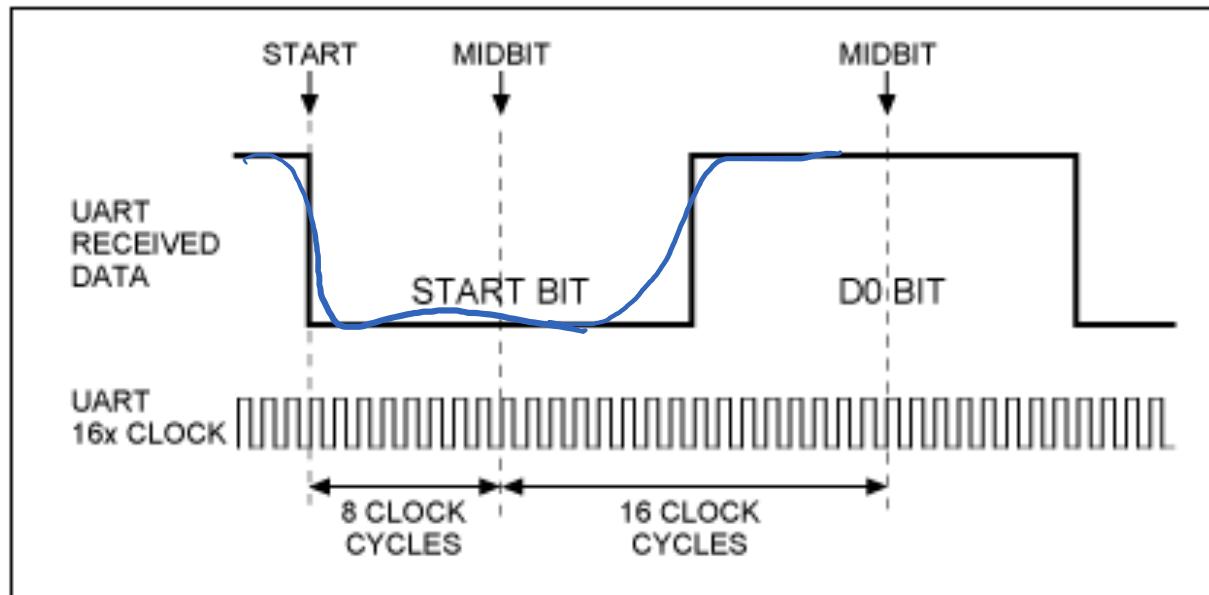


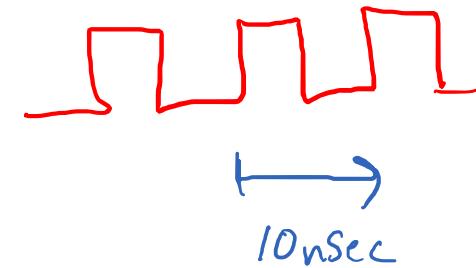
Figure 2. *UART receive frame synchronization and data sampling points.*

$$\frac{1}{2}(10^4) + 104\mu s$$

UART TX: Timing

~~CLK100MHz~~

- Basys3 ~~CLK100MHz~~ = 100MHz clock
- How do we create a 104μS delay?



$$f_c \approx 100 \text{ MHz} = 100 \cdot 10^6 \text{ cycles/sec}$$

$$P = \frac{1}{f} = \frac{1}{100 \cdot 10^6} = \frac{1 \cdot 10^{-6}}{100} = 1 \cdot 10^{-8} \text{ sec} \approx 10 \cdot 10^{-9} \text{ sec} = 10 \text{nSec}$$

$$10 \text{nSec} \cdot X = 104 \mu\text{Sec} \Rightarrow 10 \text{nSec} \cdot X = 104000 \text{nSec}$$

$$10 \cdot 10^{-9} \cdot X = 104 \cdot 10^{-3}$$

$$X = \frac{104000}{10} = 10400 \text{ cycles}$$

$$\text{Basys3} \rightarrow f = 100 \text{ MHz} \quad P = \frac{1}{100 \cdot 10^6} = \frac{1}{100} \cdot 10^{-6}$$

= $1 \cdot 10^{-8}$

$= 10 \text{ nsecs}$

$$104,16 \mu\text{sec} = \frac{104,16 \cdot 10^{-6} \text{ sec}}{10 \text{ nsec}} = 10 \cdot 10^{-9} \text{ sec}$$

$$10 \text{ nsec} = 10 \cdot 10^{-9} \text{ sec}$$

$$10 \text{ nsec} = \frac{104,160 \cdot 10^{-9} \text{ sec}}{10 \cdot 10^{-9} \text{ sec}}$$

$$= 10416 \text{ cycles}$$

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
```

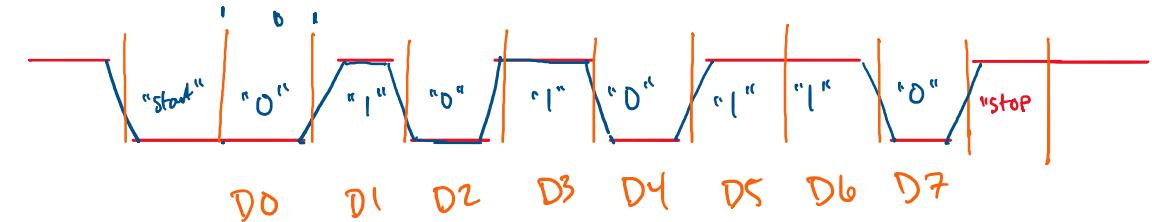
Fun Extra: Parameterizable Modules

```
timer #(  
    .TMR_BITS(16) //16-bit  
) tim0 (  
    .clk(clk),  
    .load(load),  
    .data(data),  
    .trigger(trigger)  
) ;
```

```
module timer #(  
    parameter TMR_BITS = 32  
) (  
    input clk,  
    input load,  
    input [TMR_BITS-1:0] data,      // <- 32-bit timer  
    output trigger  
) ;  
  
logic [TMR_BITS-1:0] count;  
  
always_ff @ (posedge clk) begin  
    if (load)      count <= data;  
    else if (count != 0)  
        count <= count - 'h1;  
end  
  
assign trigger = (count == 0);  
  
endmodule
```

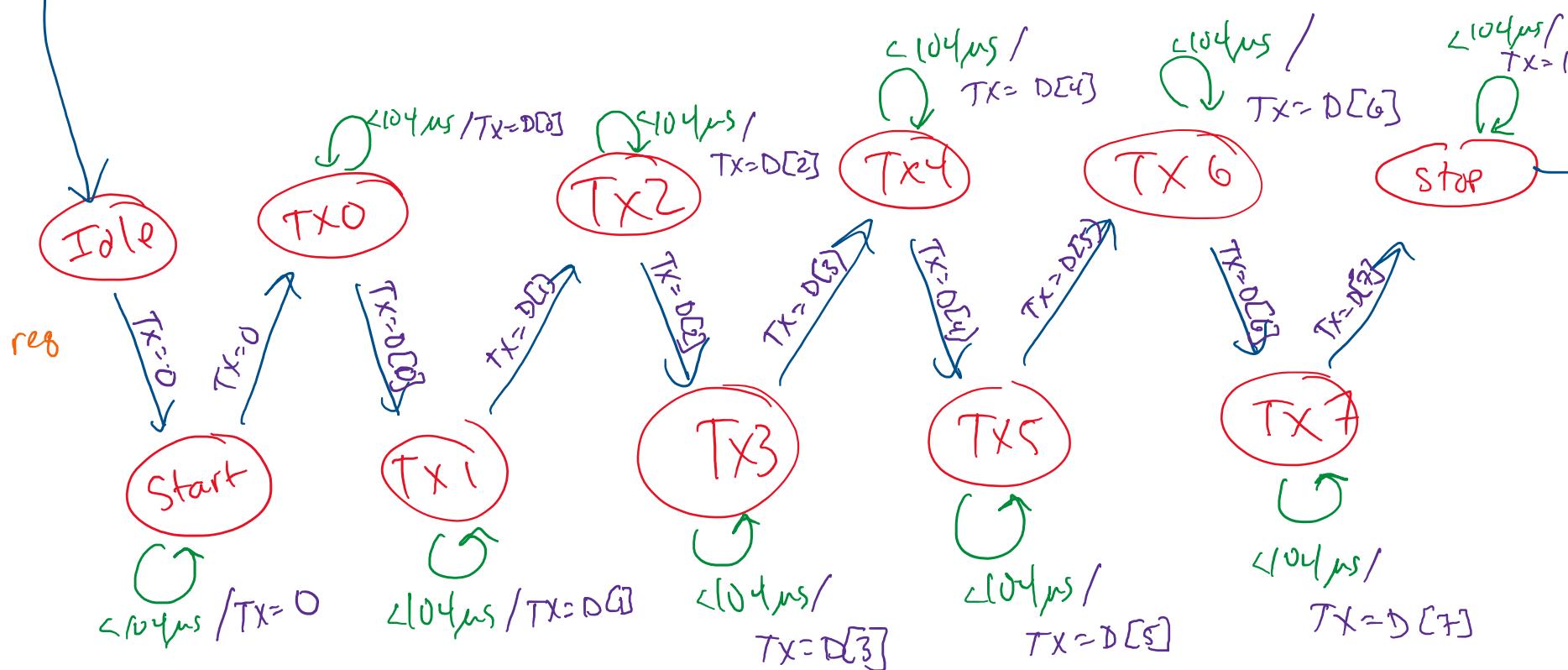
UART TX State Machine

- How to transmit 8'b01101010?

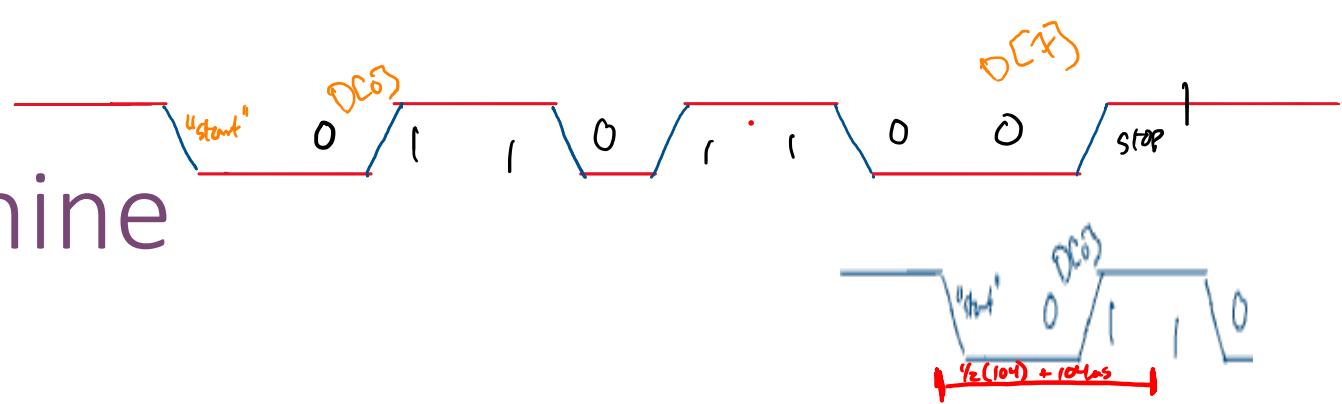


UART: Transmit

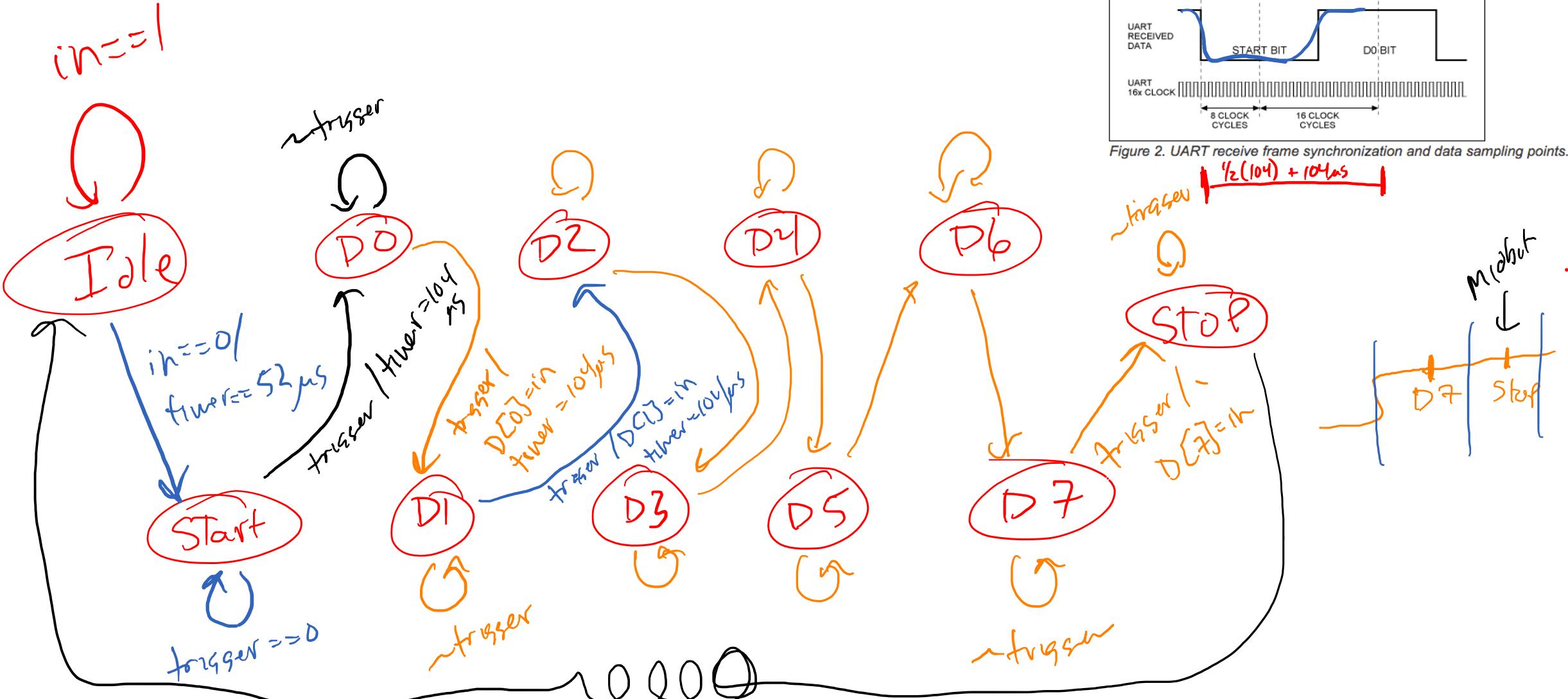
- How to transmit 8'b01101010?



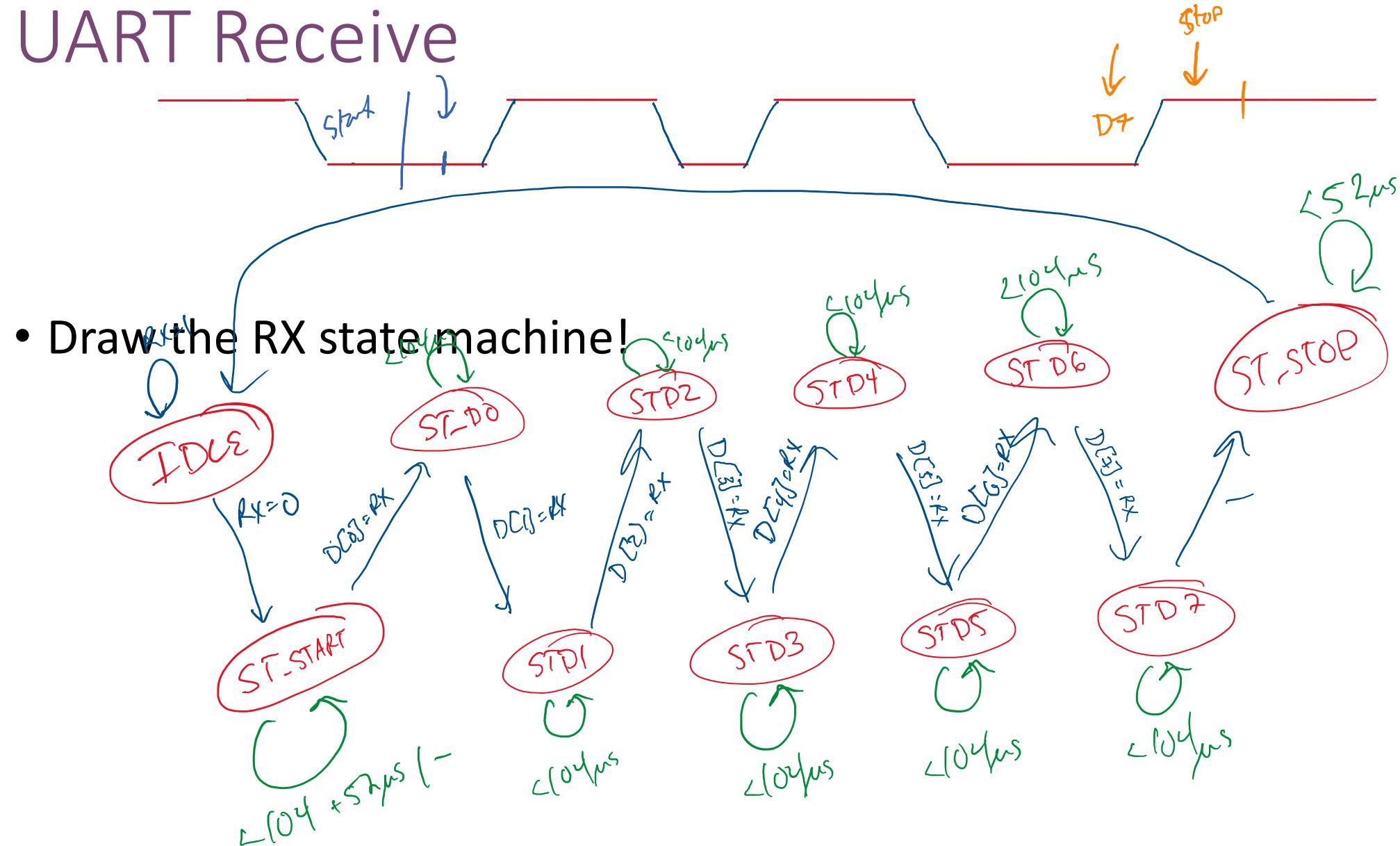
UART RX: State Machine



UART RX: State Machine



UART Receive



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
```

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

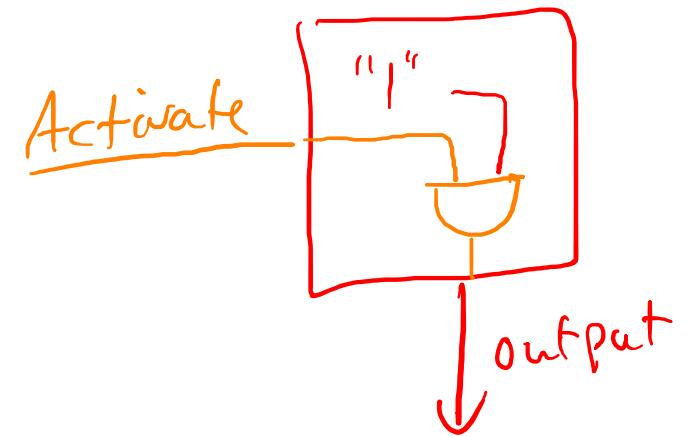
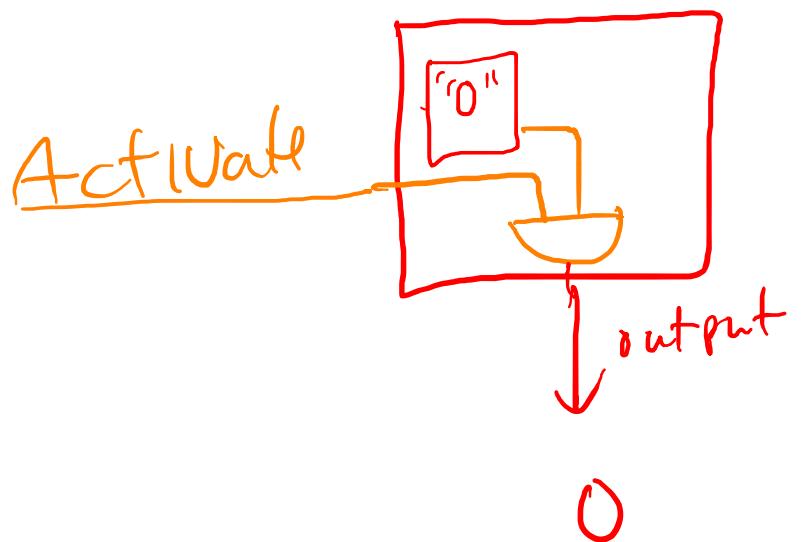
- You get to build a UART interface
- P5: just echo RX back over TX
- Connect your FPGA to your PC
- Allows you to “talk” to your FPGA with keyboard
- P6: we add python UART interface

Memory

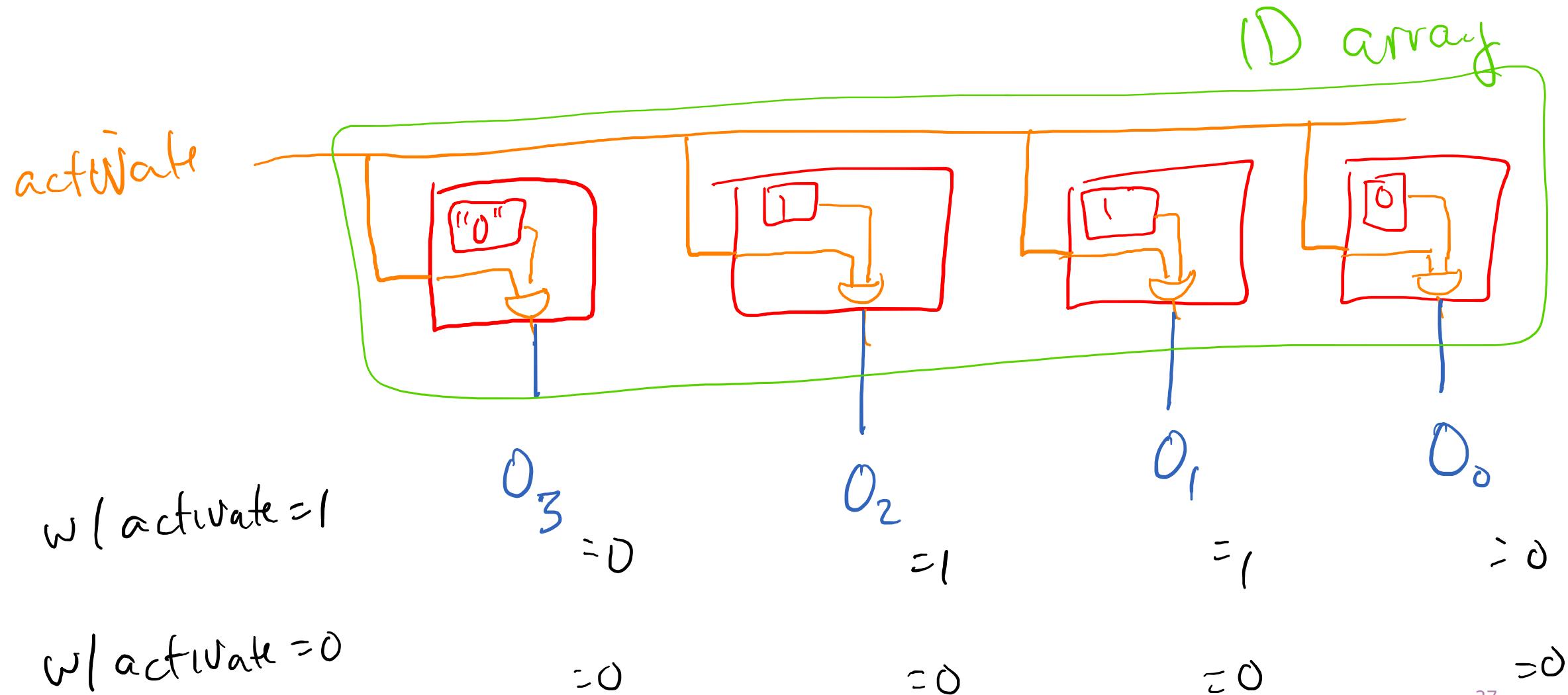
ROM vs RAM

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

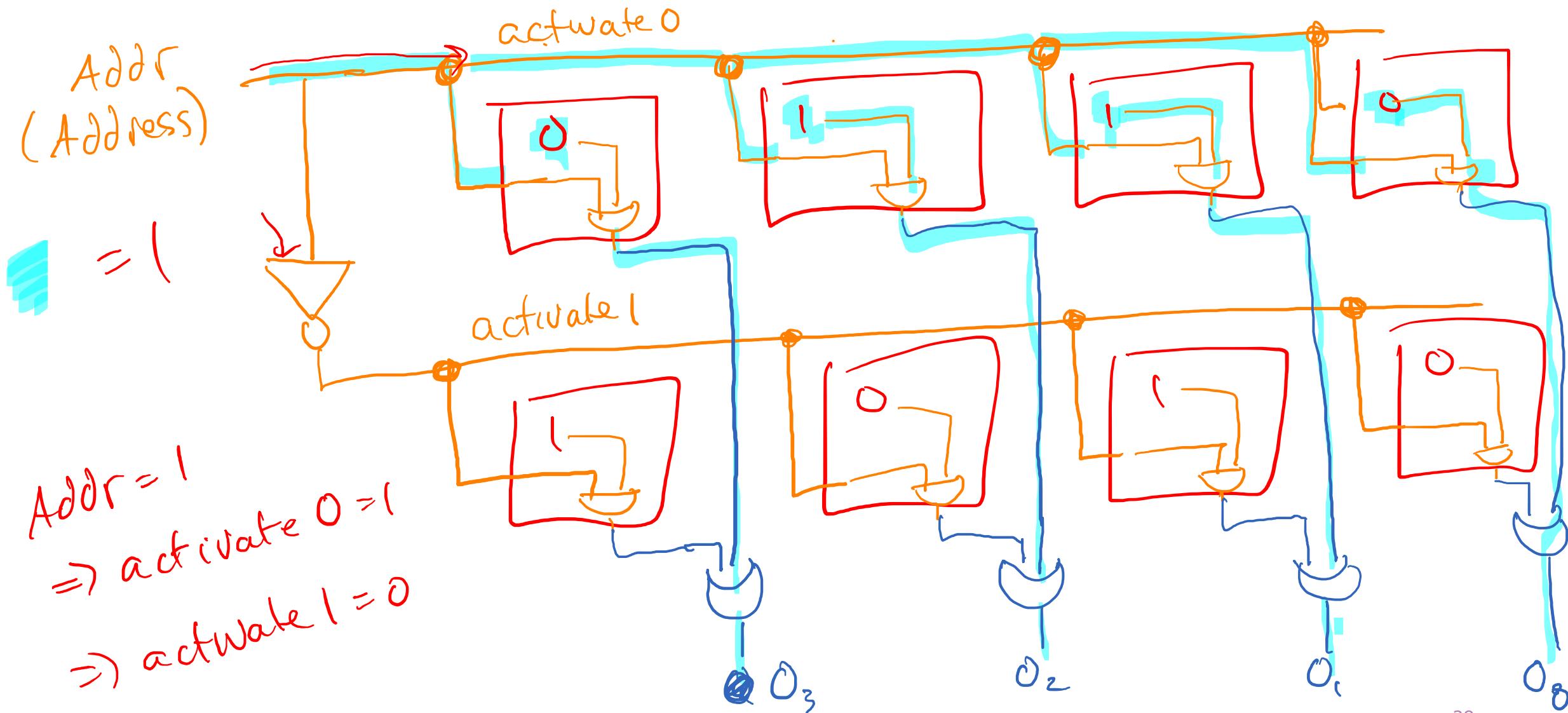
Rom Cell



Multiple ROM Cells

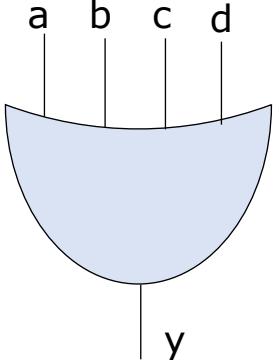


Array of ROM Cells

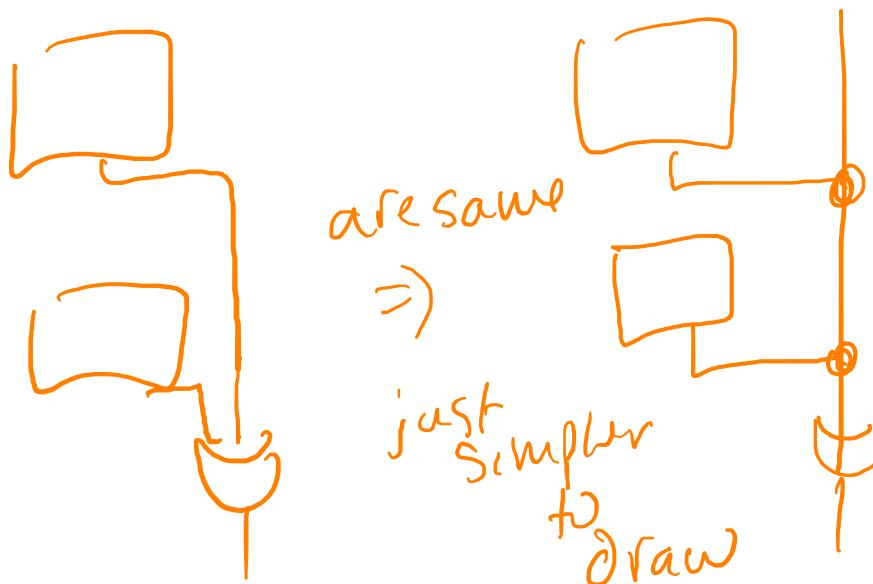
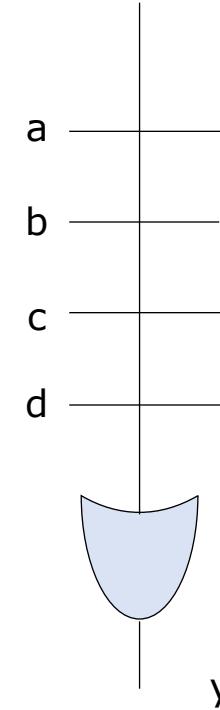


Notation for the gates

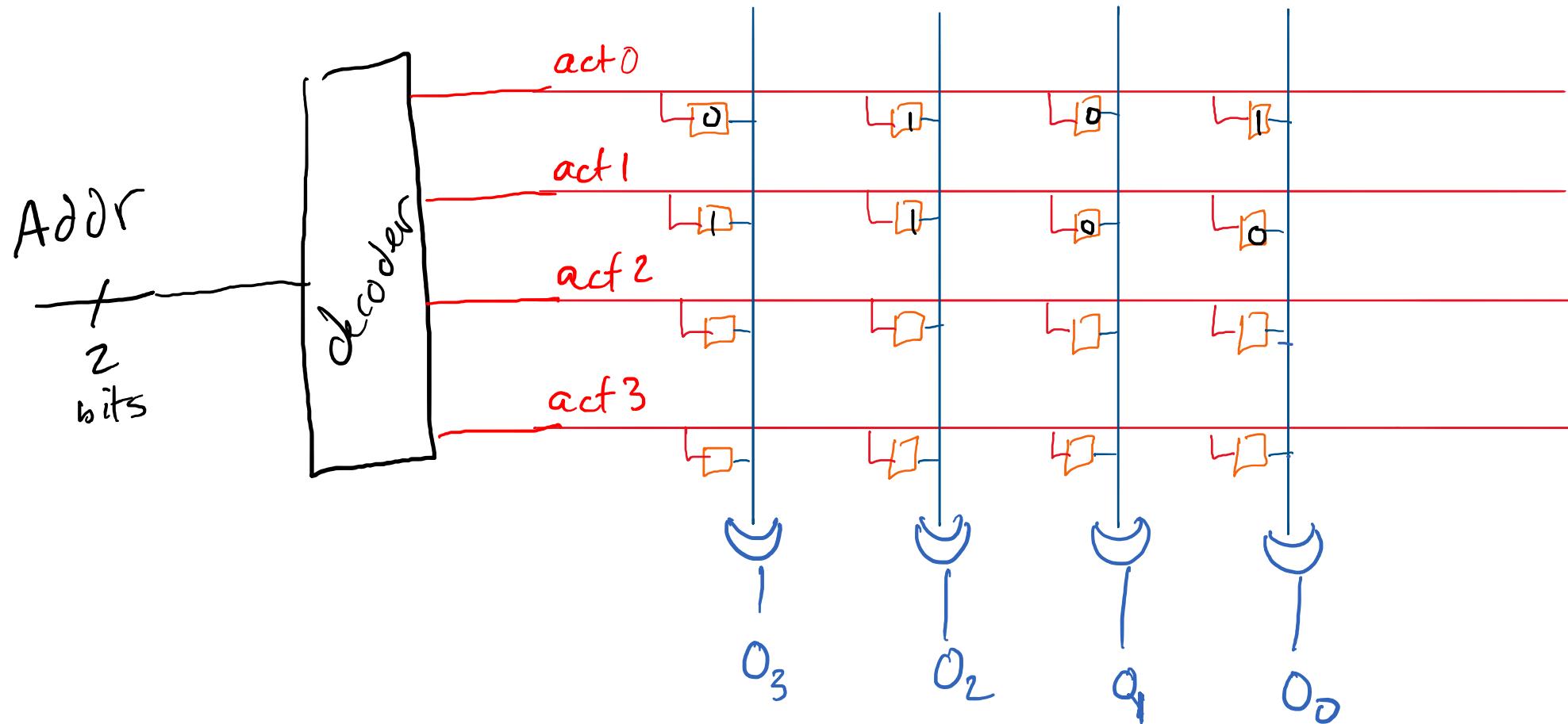
Instead of:



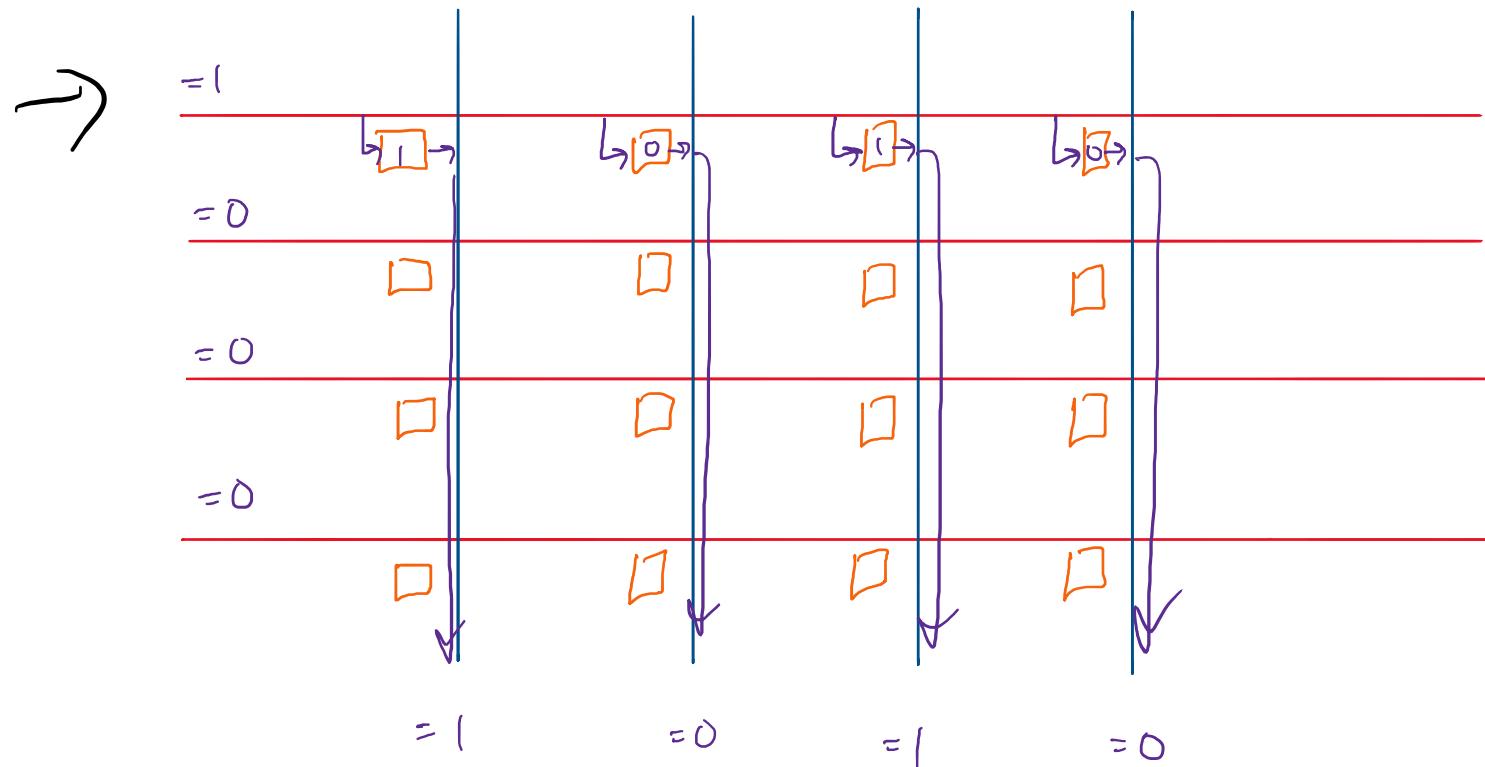
We often use
notation:



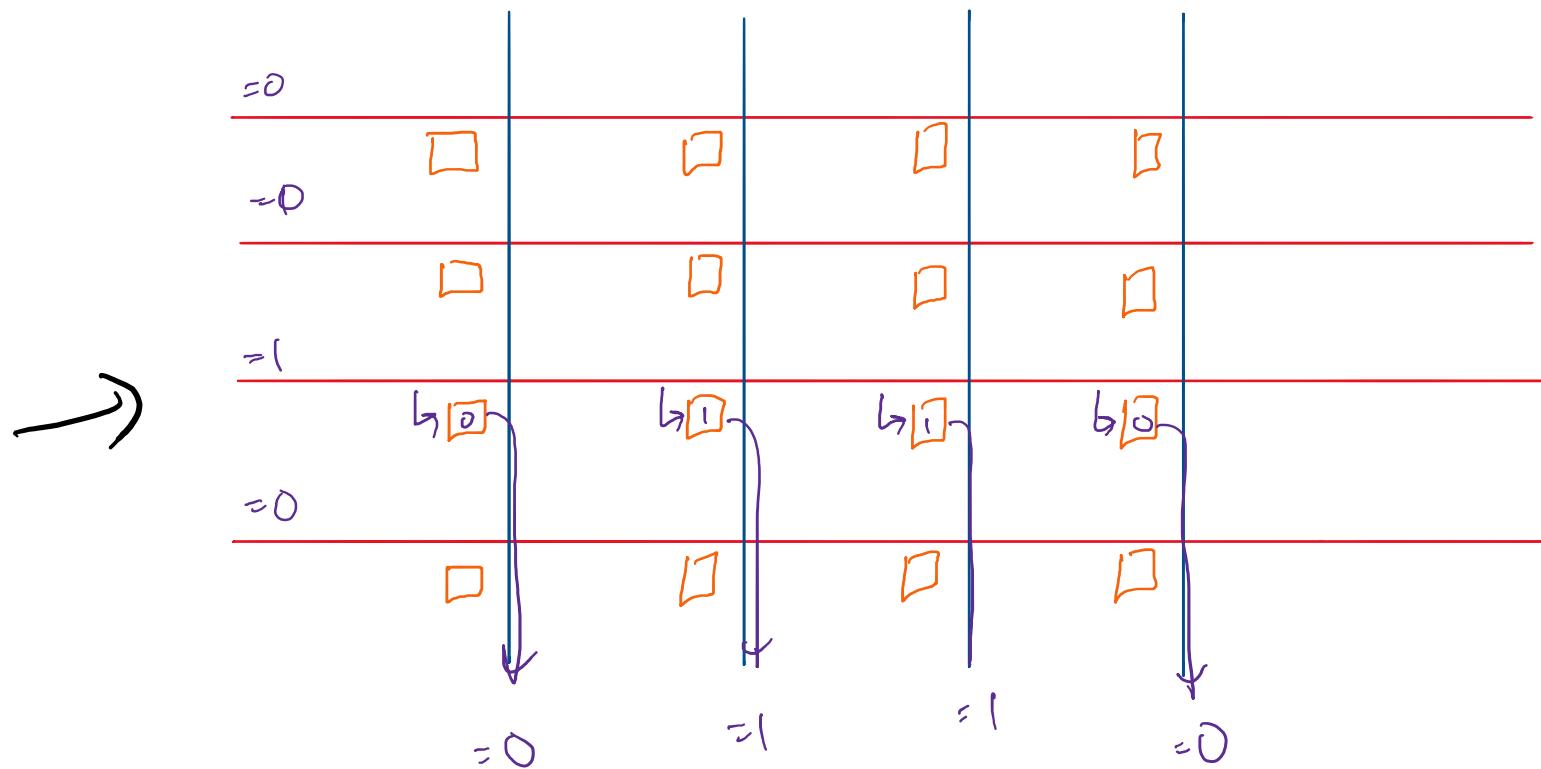
2-bit ROM



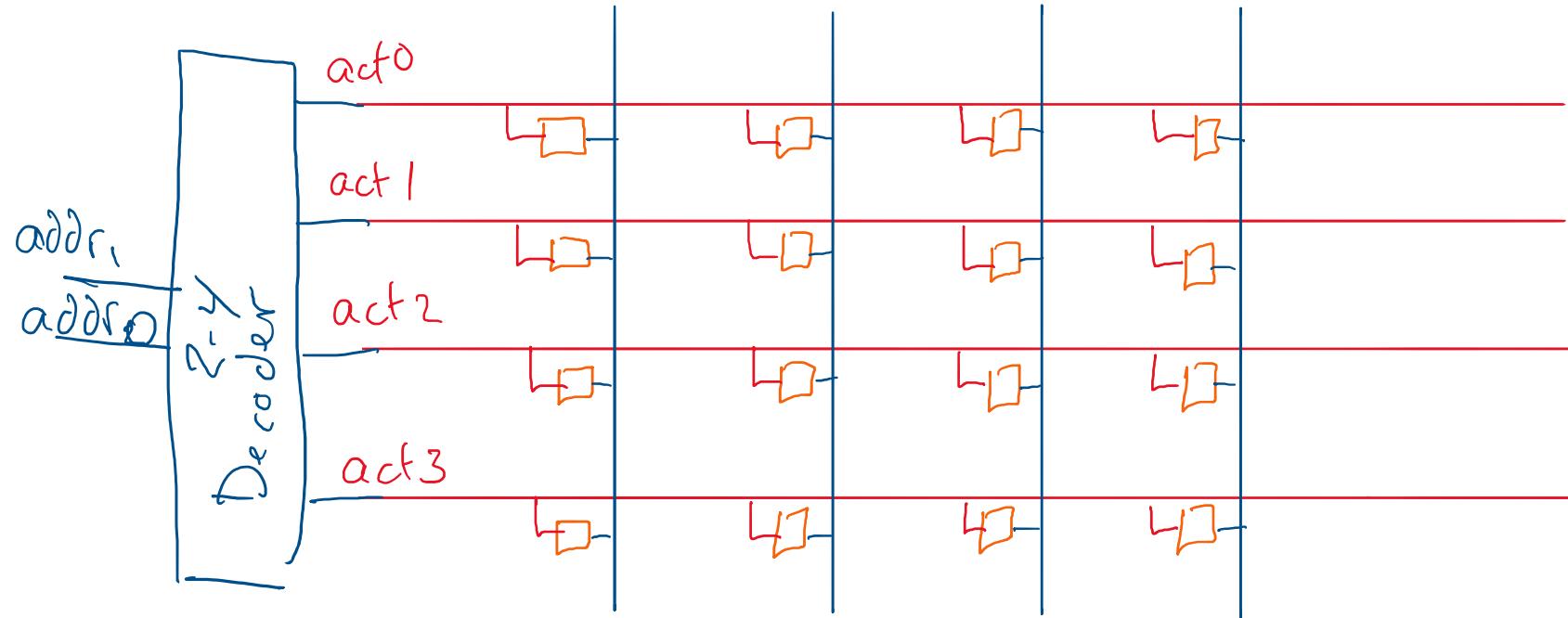
2-bit ROM



2-bit ROM



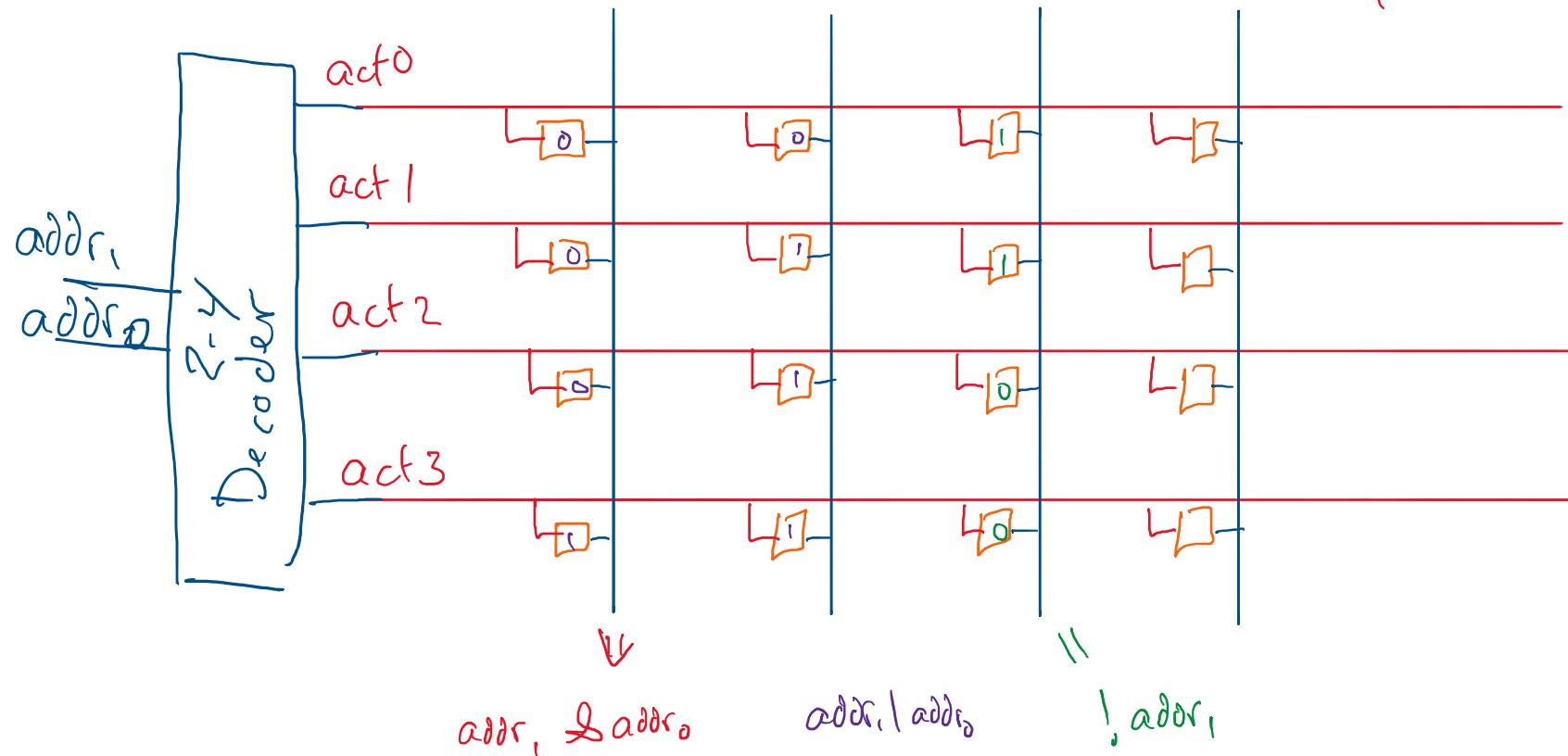
2-bit ROM of AND + OR



2-bit ROM of AND + OR

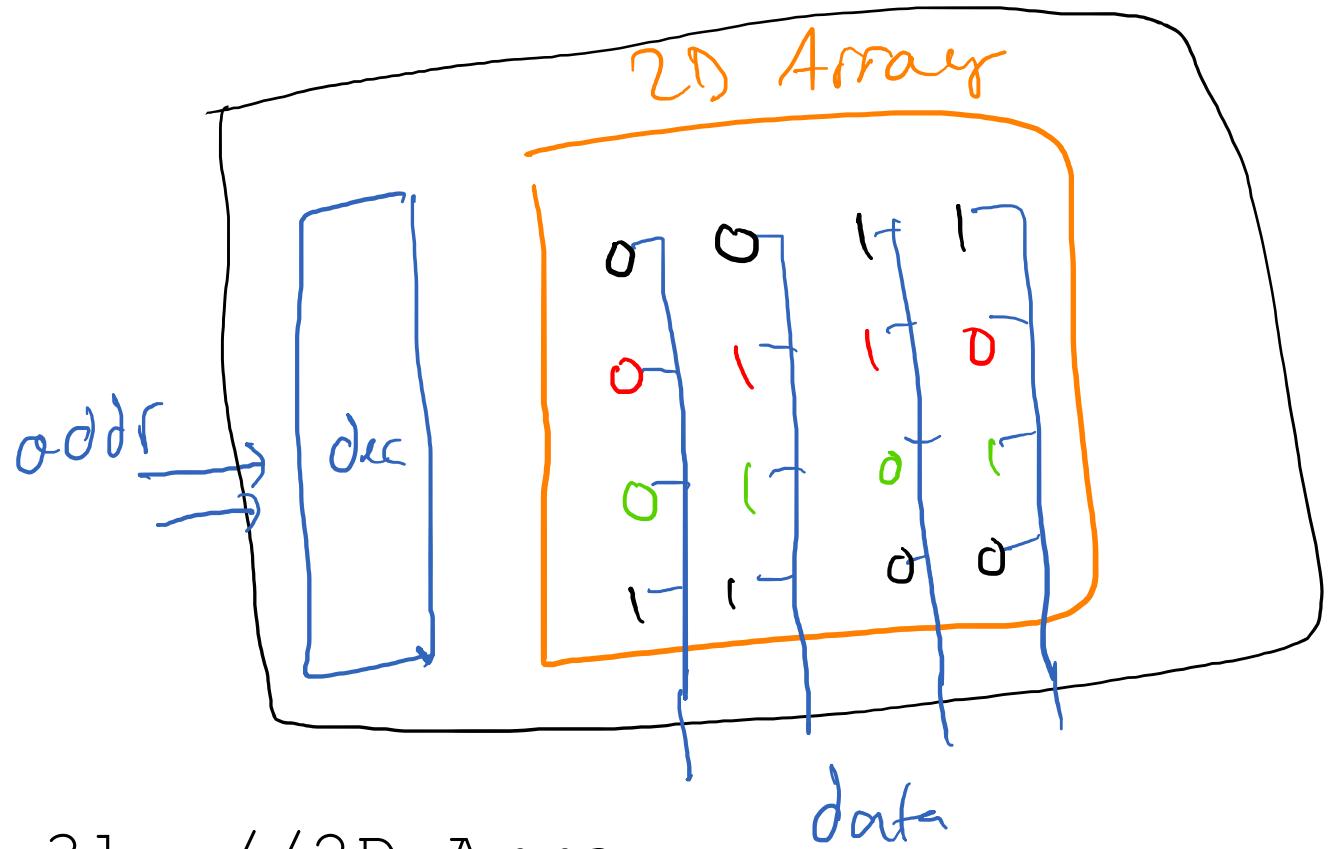
AND / OR

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
```

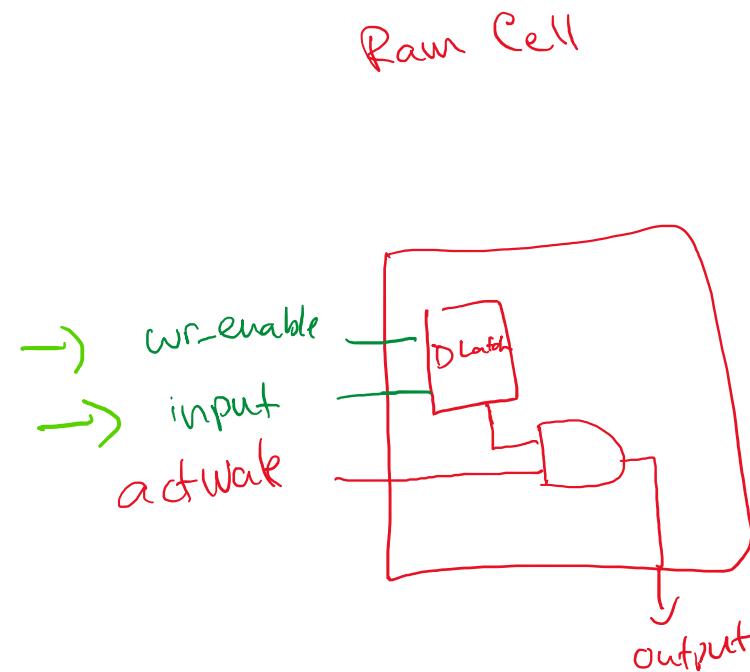
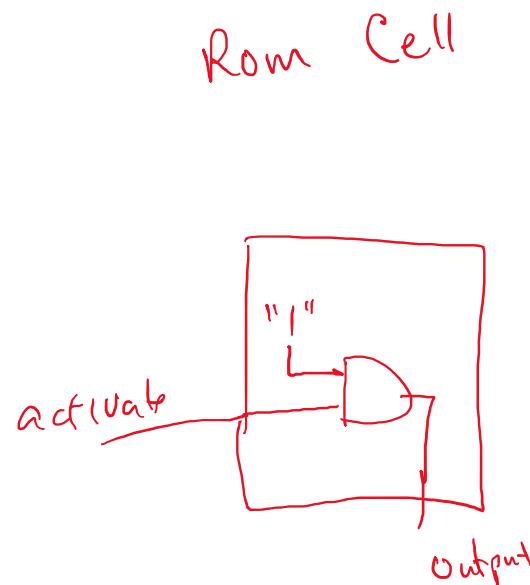


RAM

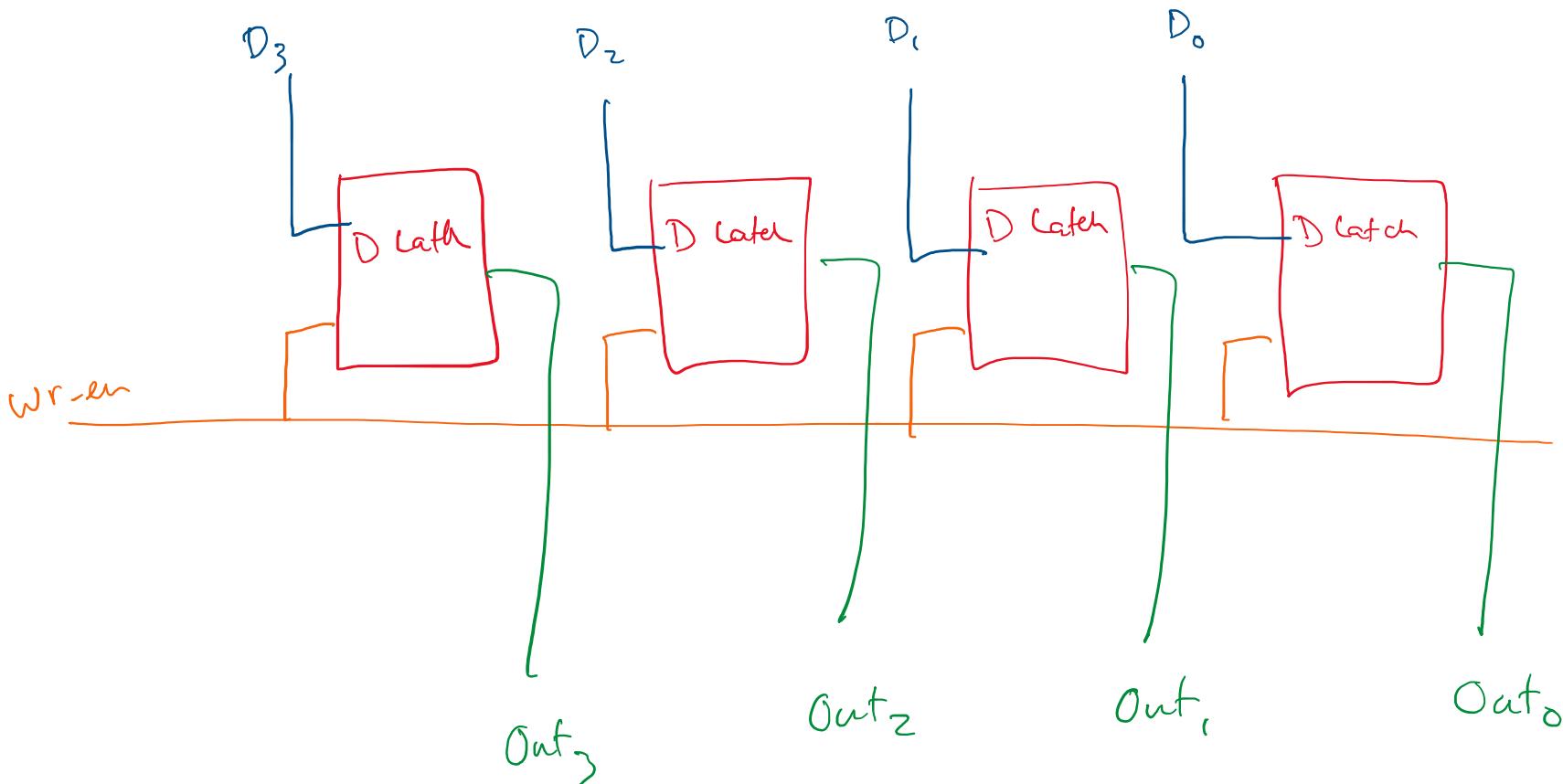
- Similar to ROM
- BUT WRITABLE!

RAM

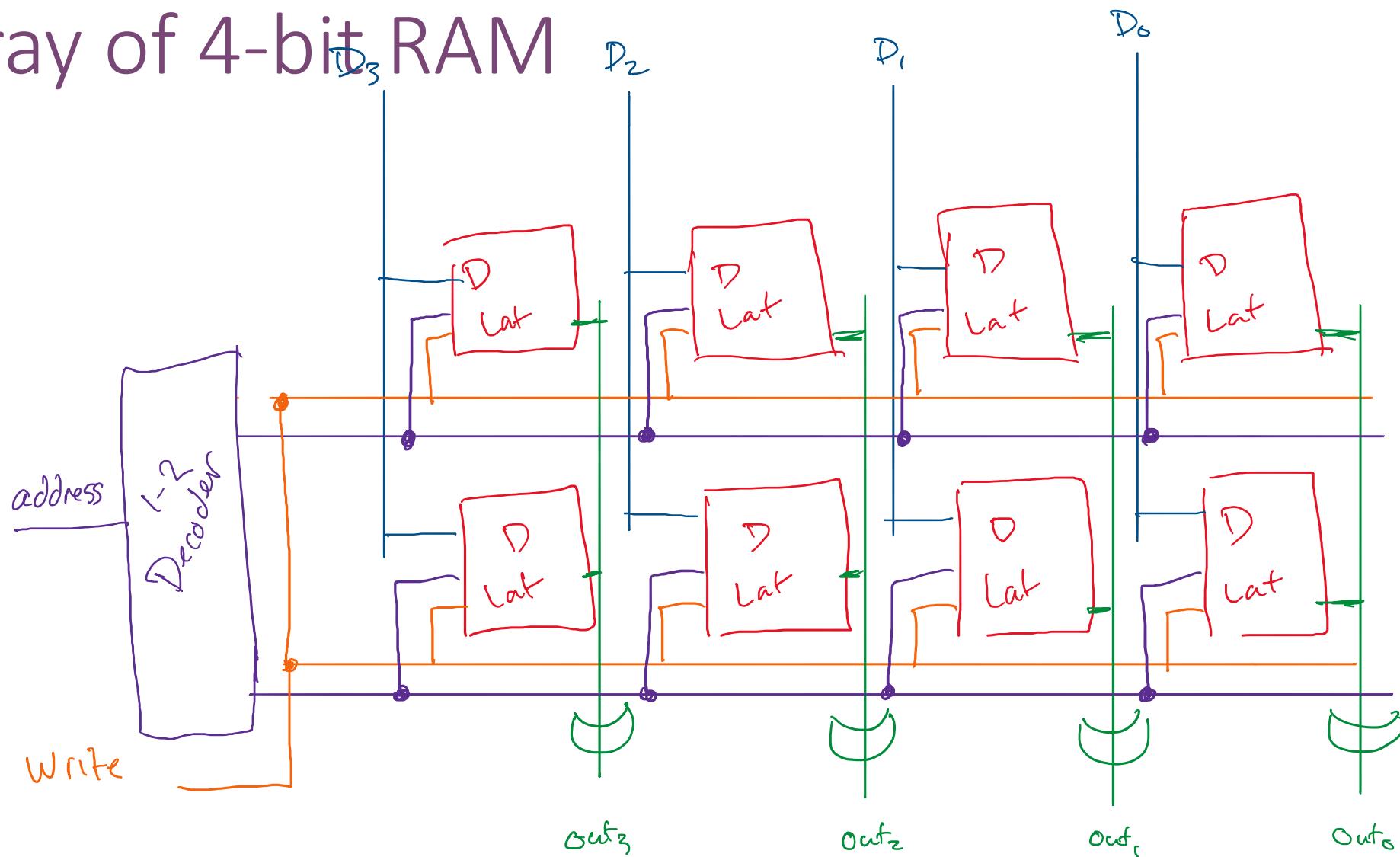
- Similar to ROM
- BUT WRITABLE!



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!

Next Time

- Data Structures with RAMs