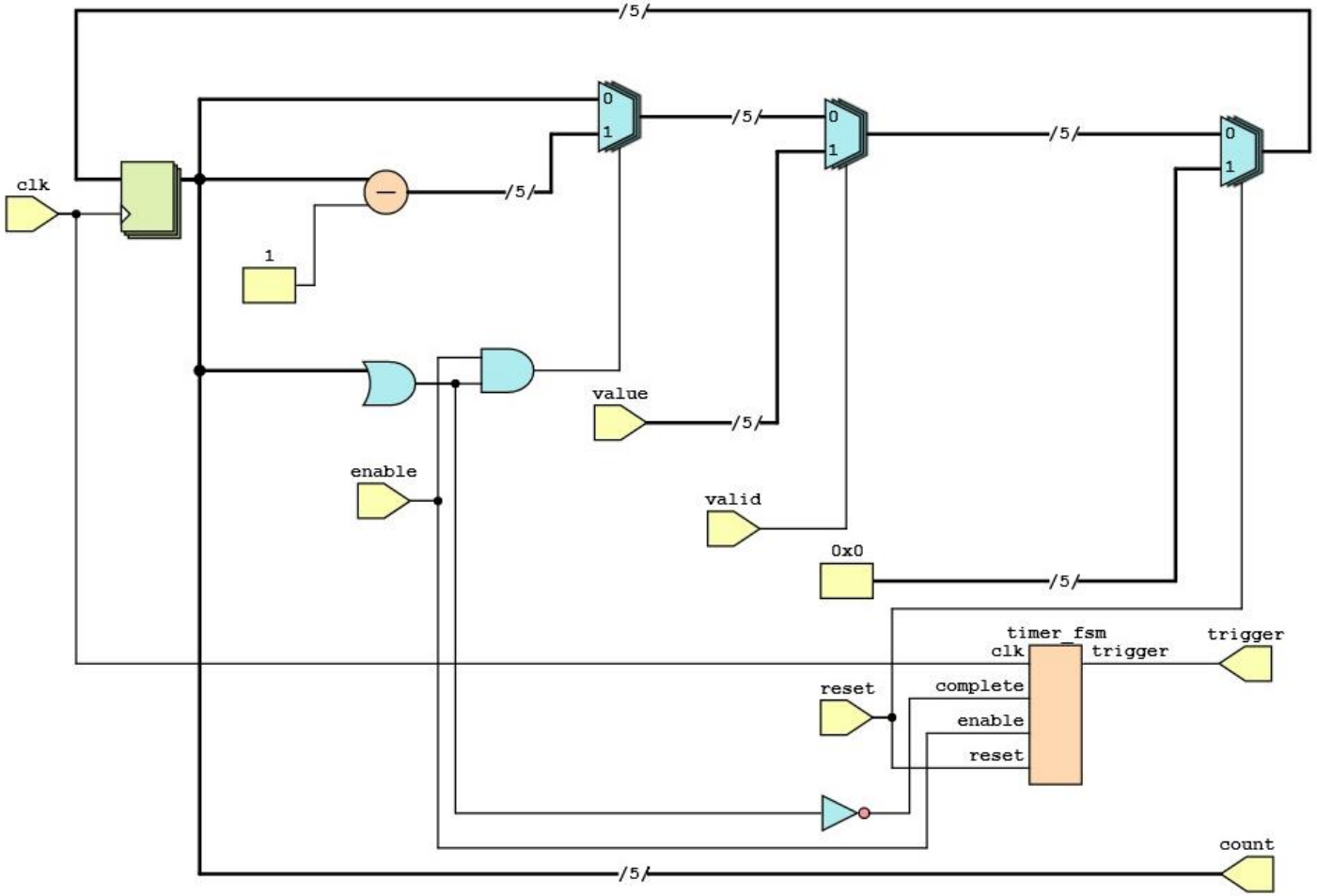
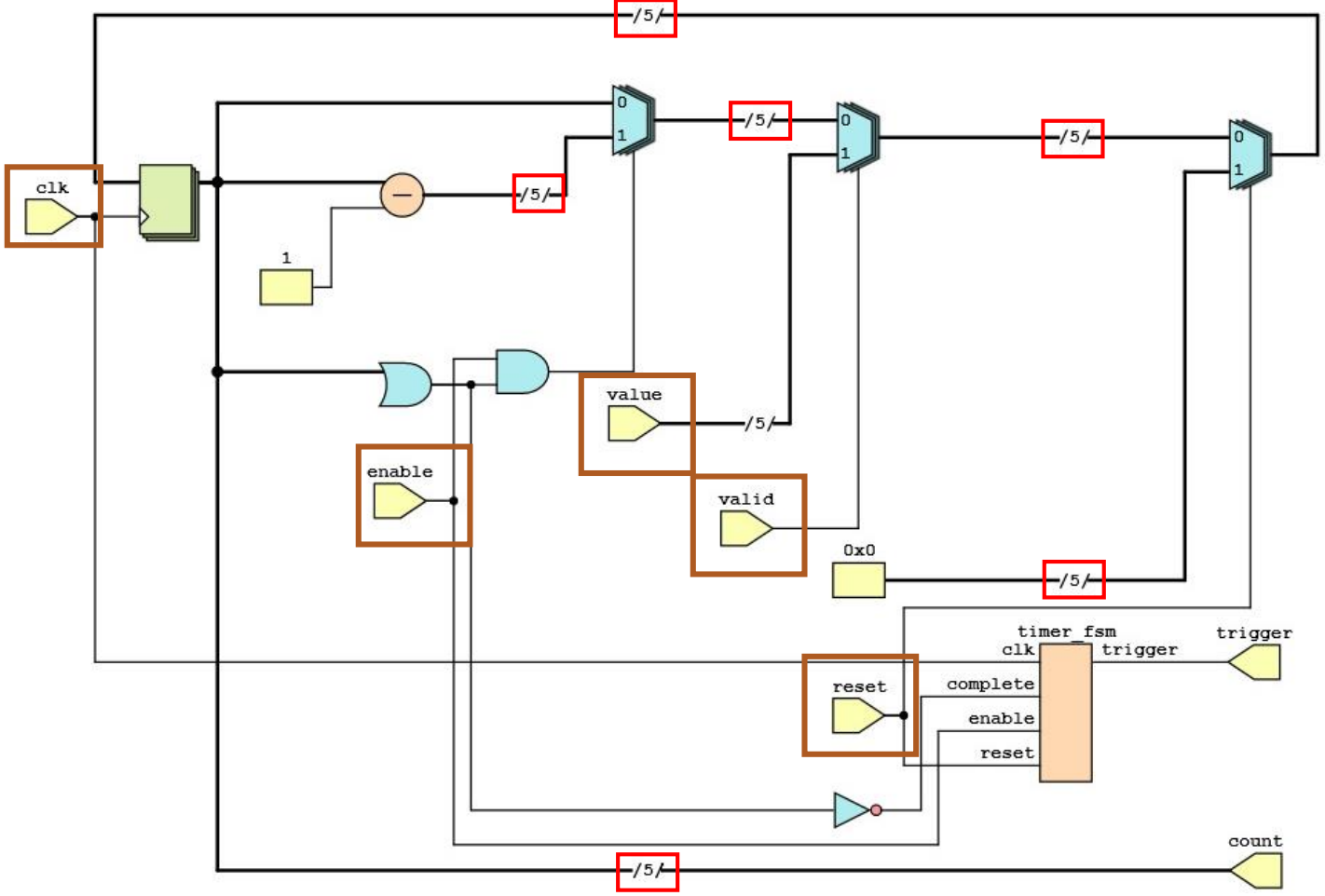


Analyzing a schematic for Verilog implementation.

Start identifying the inputs and outputs!



Analyzing a schematic for Verilog implementation.
Start identifying the inputs and outputs!

Inputs come in from the left, as we read from left to right:

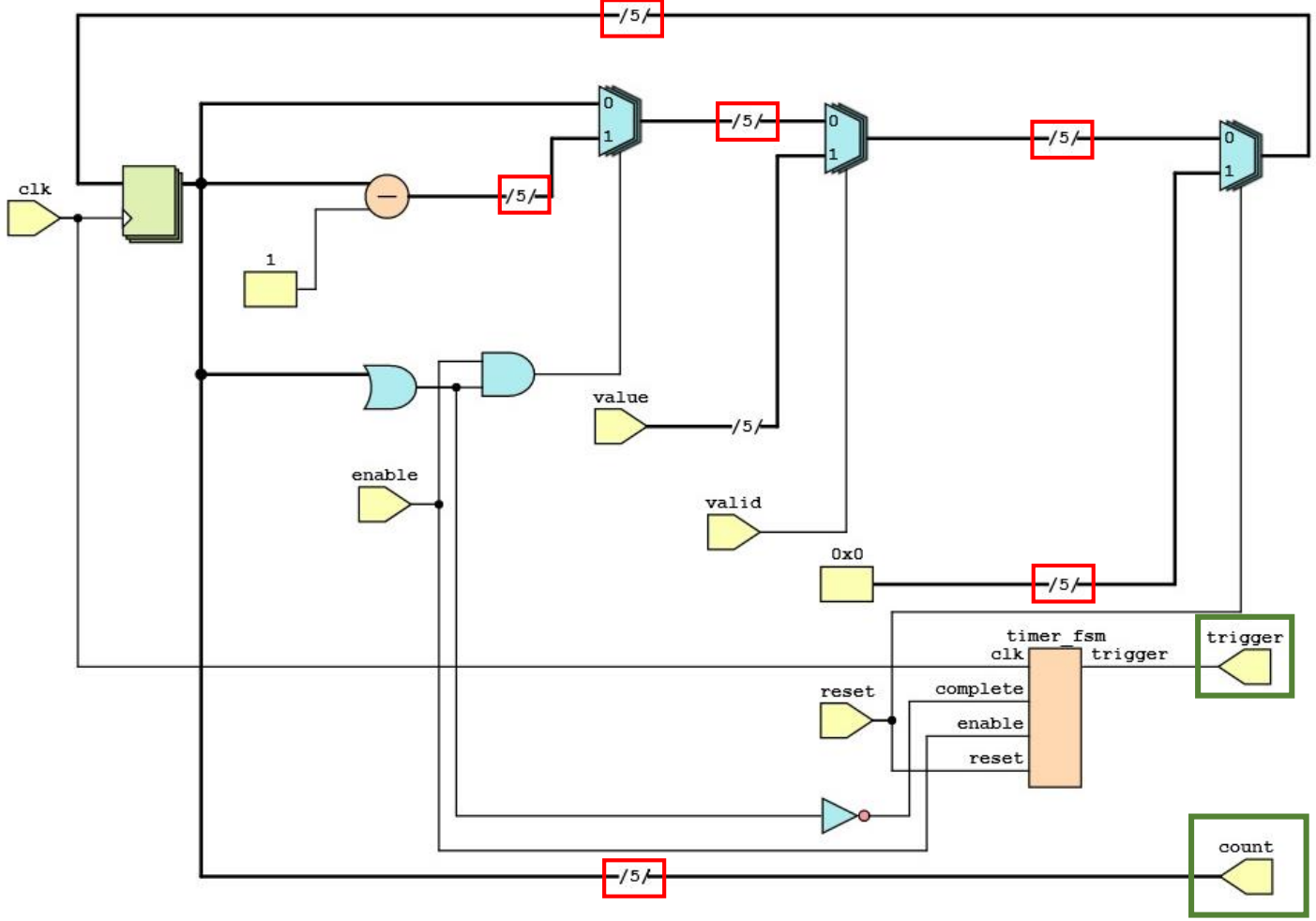
- **clk**
- **enable**
- **value** -> *5 bits wide!*
- **valid**
- **reset**

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
10 );
33 endmodule
34
```

Pay attention to the bit width of signals!

Always assume a signals is 1 bit wide, unless otherwise indicated with /X/ notation on the signal!

e.g: The **value** input is 5 bits wide -> specified in Verilog as [4:0]



Analyzing a schematic for Verilog implementation.
Start identifying the inputs and outputs!

Outputs go out towards the right, opposing the inputs!

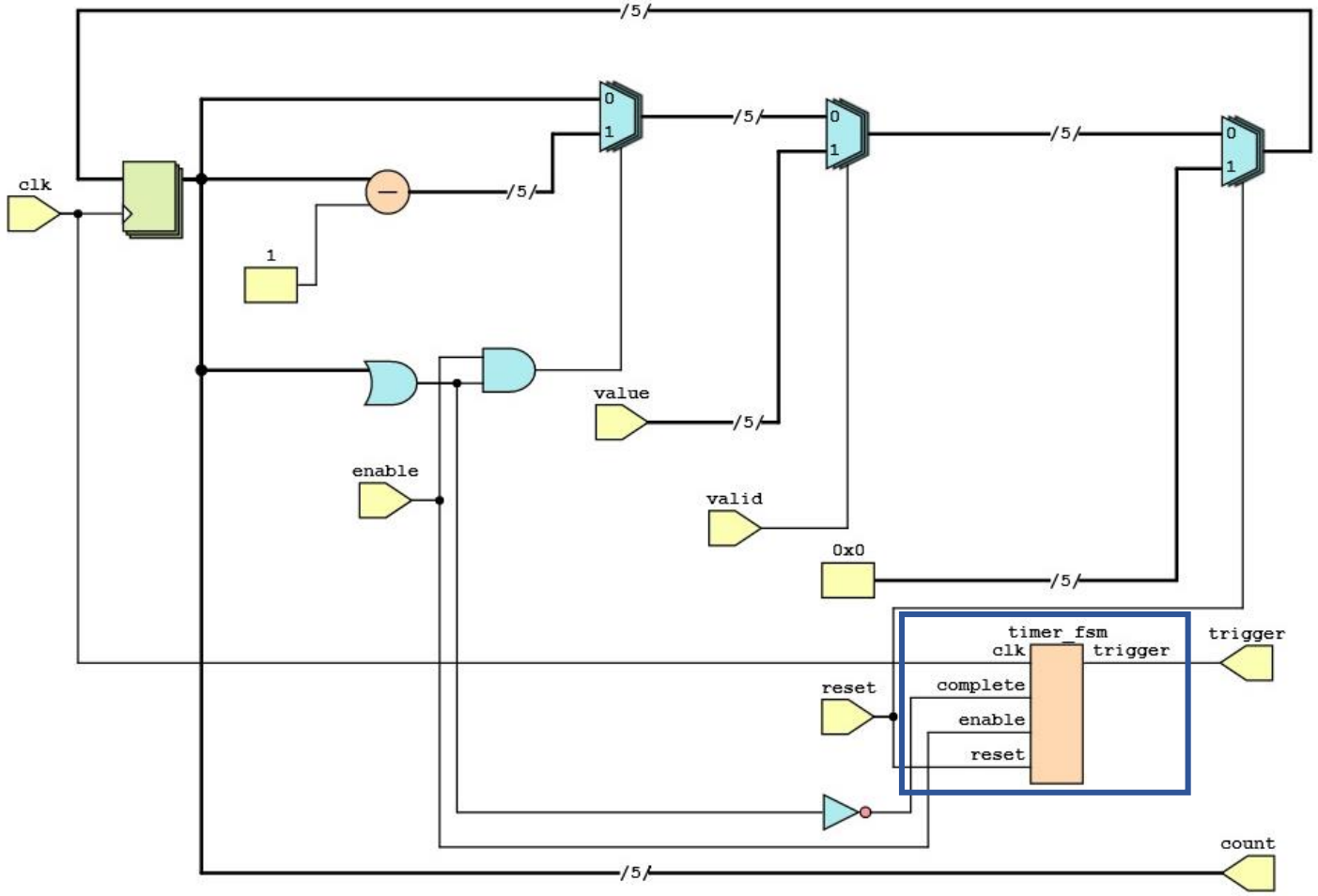
- **trigger**
- **count** -> *5 bits wide!*

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
33 endmodule
34
```

Pay attention to the bit width of signals!

Always assume a signals is 1 bit wide, unless otherwise indicated with /X/ notation on the signal!

e.g: The **count** output is 5 bits wide -> specified in Verilog as [4:0]

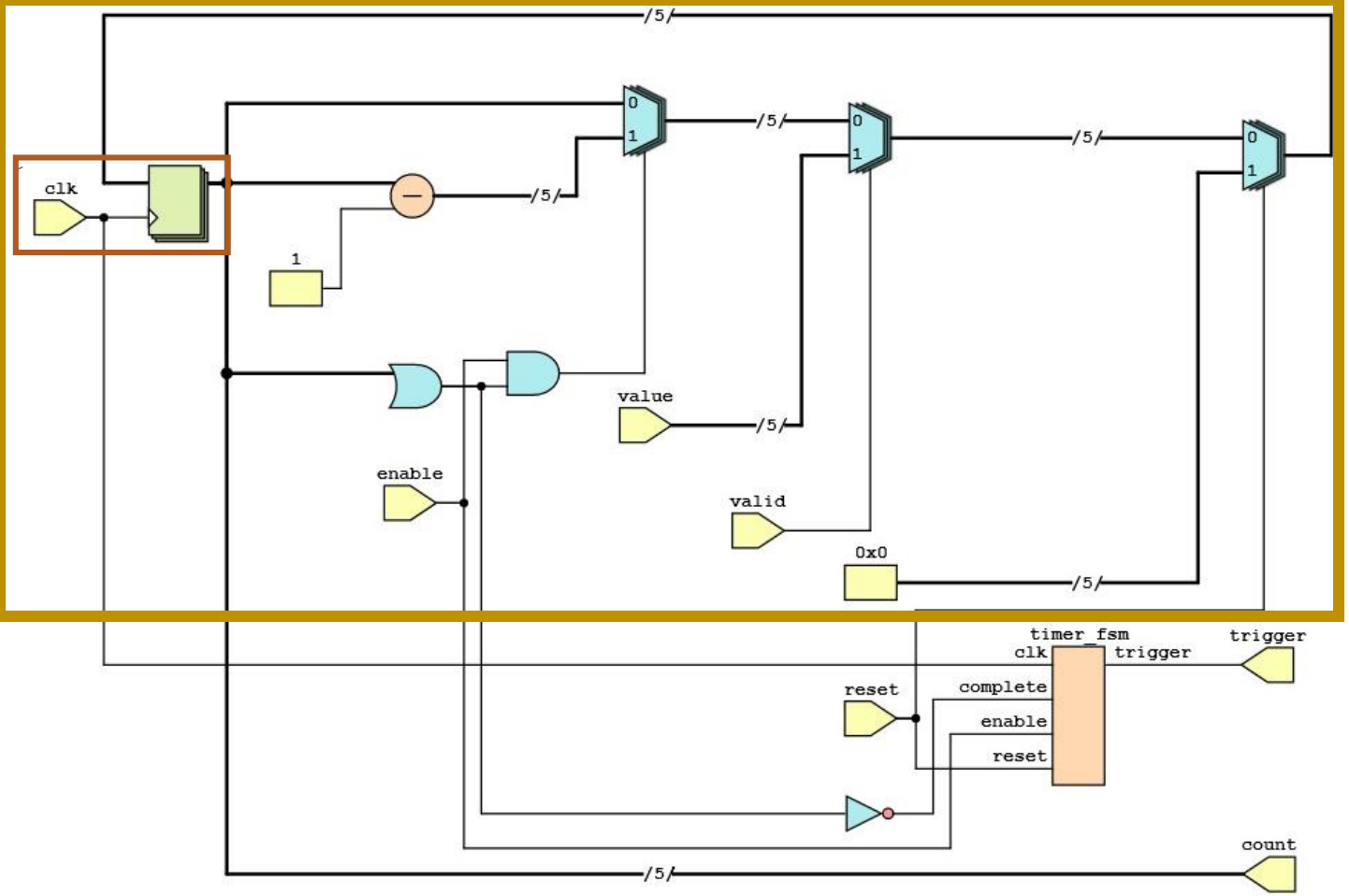


We aren't finished with the complete wire yet. What it is connected to is looked at in the next slides.

Analyzing a schematic for Verilog implementation.
Move on by including and connecting submodules!

- **timer_fsm**
- **Inputs: (On the left)**
  - **clk**
  - **complete**
  - **enable**
  - **Reset**
- **Outputs: (On the right)**
  - **trigger**

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
33 endmodule
34
```



Analyzing a schematic for Verilog implementation.
Now for the difficult part: Making sense of the LOGIC!

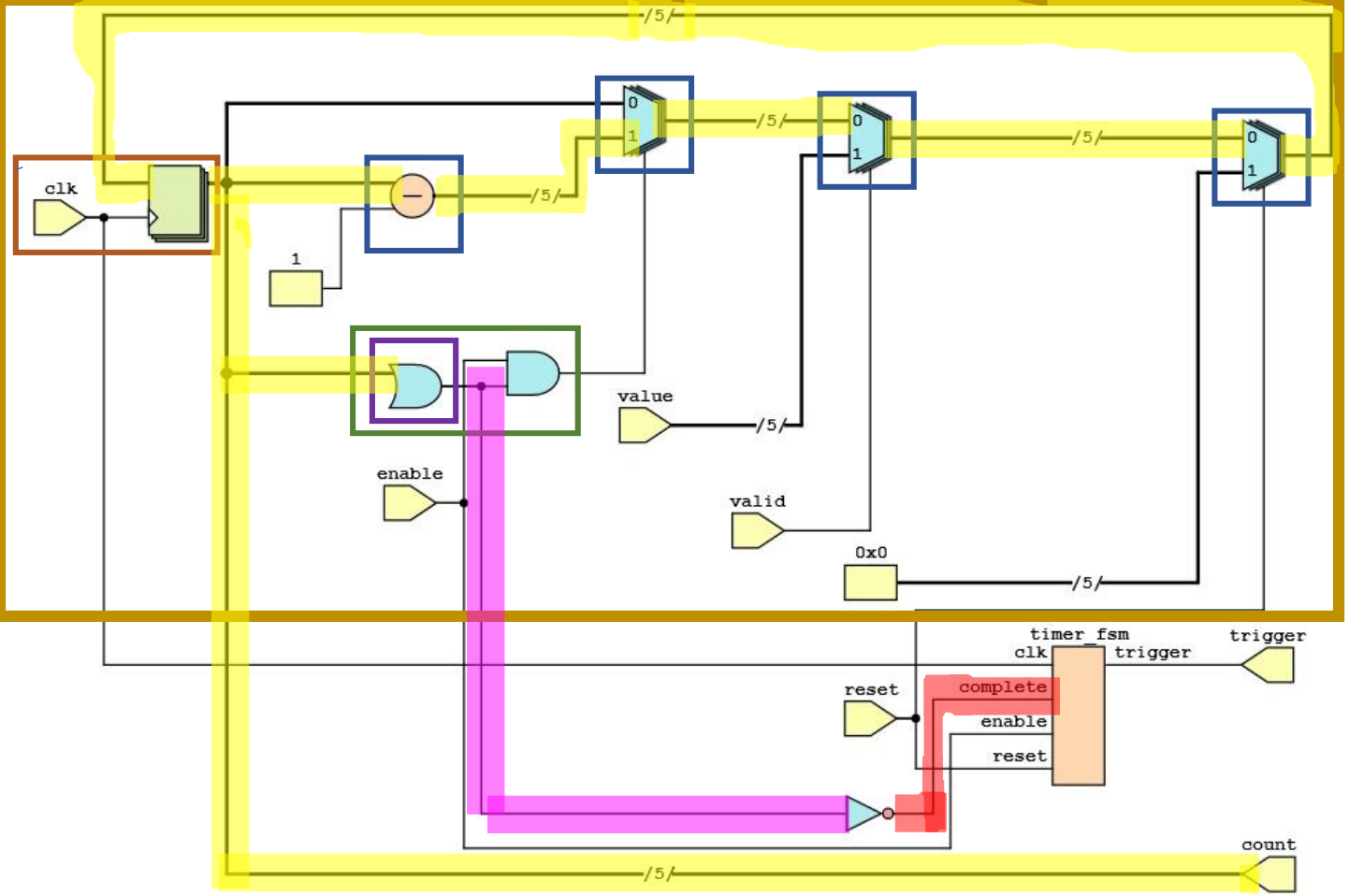
But is it **combinational** or **sequential** logic?

Sequential!
The logic circuit starts and ends in **clock** synchronized latches.

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
22 always @(posedge clk) begin
32 end
33 endmodule
34
```

The difference between logic being **combinational** or **sequential** is of the utmost importance!

Use the **clk** signal to identify **clock** synchronized part of the logic!



Analyzing a schematic for Verilog implementation.
Now for the difficult part: Making sense of the LOGIC!

We see some **logic gates** but what are these **other symbols**?
There is one **subtractor** and three **chained multiplexors** (MUX's)

The **subtractor** tells us this module must be counting down the **count** from the latches!

The **multiplexors** determine what value of **count** propagates

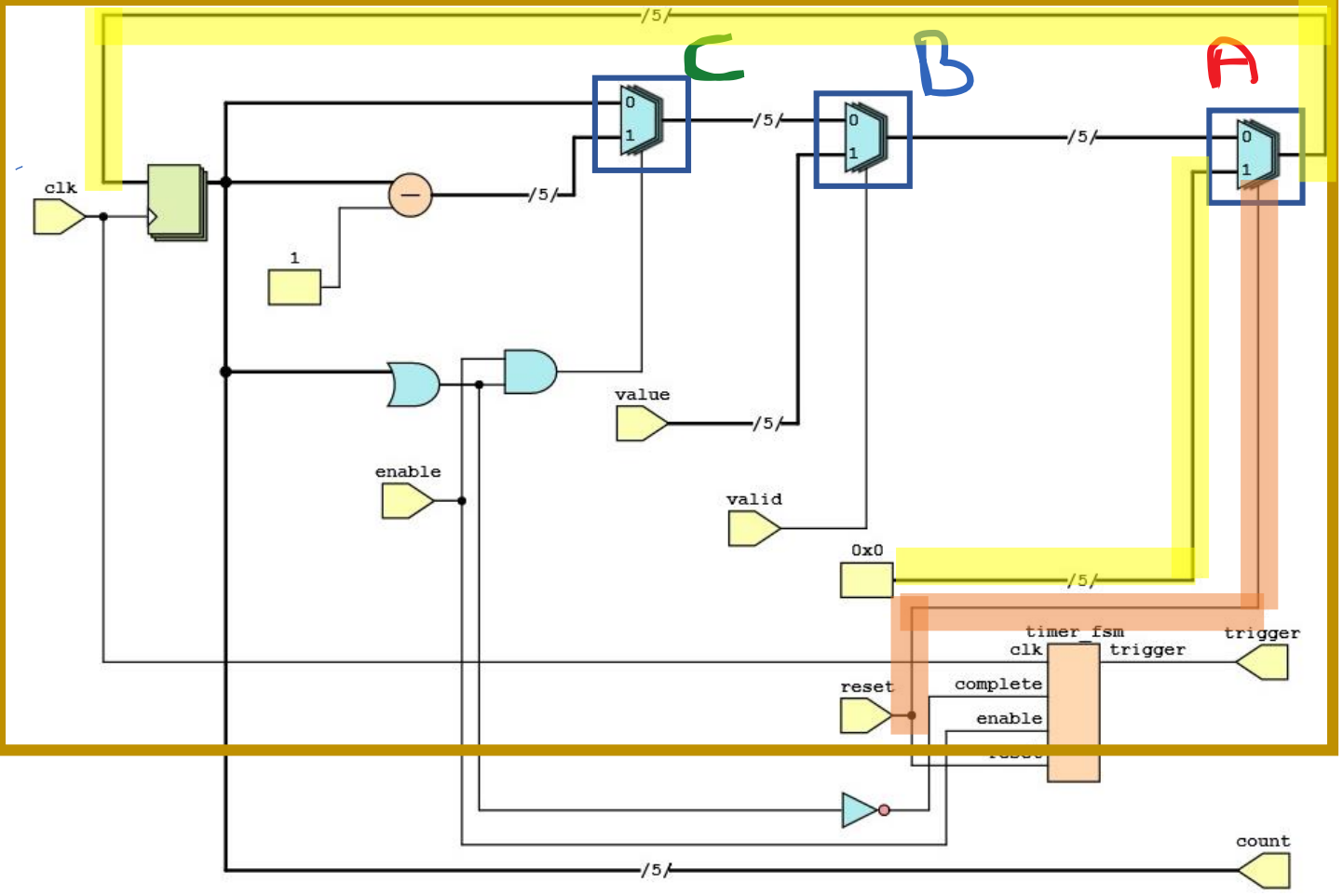
Forgot your logic gates? Check the last slide!

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete = count == 5'b0;
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
22 always @(posedge clk) begin
32 end
33 endmodule
34
```

Circuit designers are lazy and draw only that what is necessary. The **OR-gate** might be confusing, it only has one input, right?
It actually has 5! The single draw input wire, **count**, is 5 bits wide. The **OR-gate** is or'ing all of the bits of **count**.

If the count is anything but 00000, the **OR-gate** will be 1. This **OR-gate** checks if the count is zero! The **complete** wire is the inverse (NOT-gated) of the **OR-gate** output.

If **count** is 0, the **OR-gate** is 0 and **complete** is 1!



Multiplexors:
- Have inputs on the ***left***
- A single output on the ***right***
- MUX (decide) based on the *Top or Bottom inputs.*

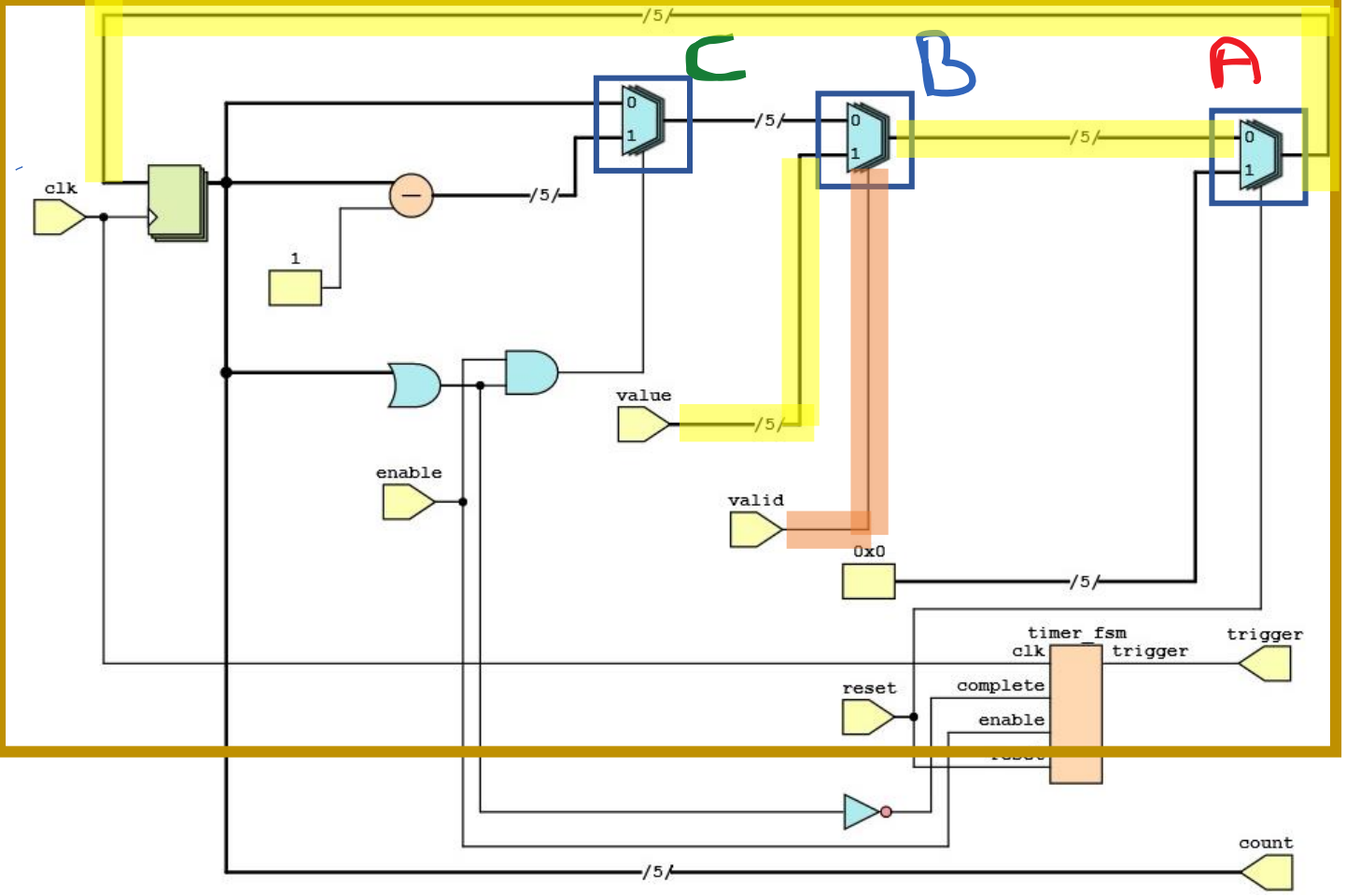
Analyzing a schematic for Verilog implementation.
Now for the difficult part: Making sense of the LOGIC!

The **multiplexor chain** is making decisions based on some values -> Its just an **IF-Else statement**!

Look at the chain from back to front.
Multiplexor A being last, can break the *whole decision chain*

It is the first IF statement! Simply resetting the count on reset.

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete = count == 5'b0;
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
22 always @(posedge clk) begin
23     if(reset) begin
24         count <= 5'b0;
31     end
32 end
33 endmodule
34
```



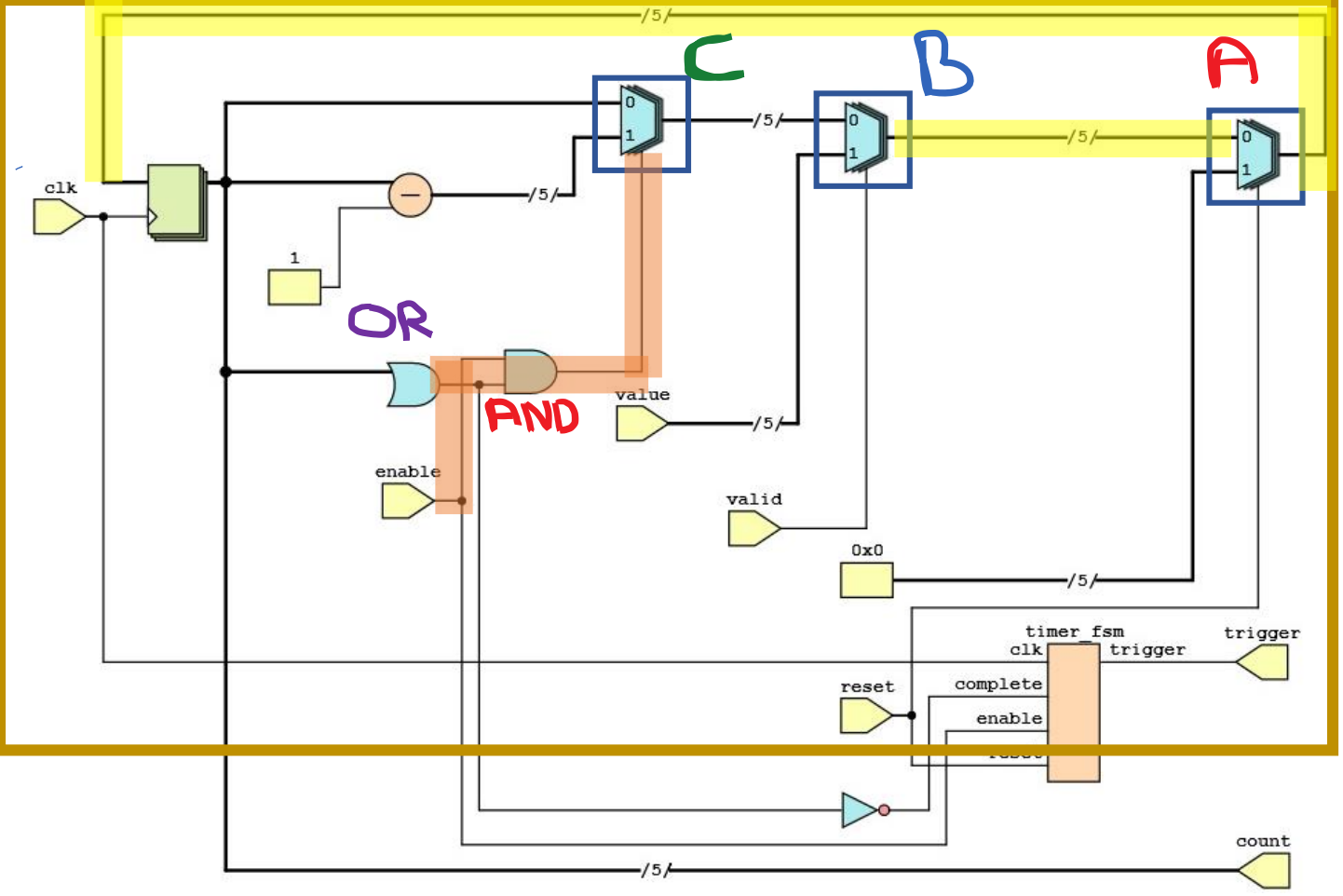
Multiplexors:
- Have inputs on the ***left***
- A single output on the ***right***
- MUX (decide) based on the *Top or Bottom inputs.*

Analyzing a schematic for Verilog implementation.
Now for the difficult part: Making sense of the LOGIC!

The **multiplexor chain** is making decisions based on some values -> Its just an **IF-Else statement**!

Look at the chain from back to front.
Multiplexor B being next:
Sets an external value into the count based on another external!

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete = count == 5'b0;
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
22 always @(posedge clk) begin
23     if(reset) begin
24         count <= 5'b0;
25     end else if (valid) begin
26         count <= value;
31     end
32 end
33 endmodule
34
```



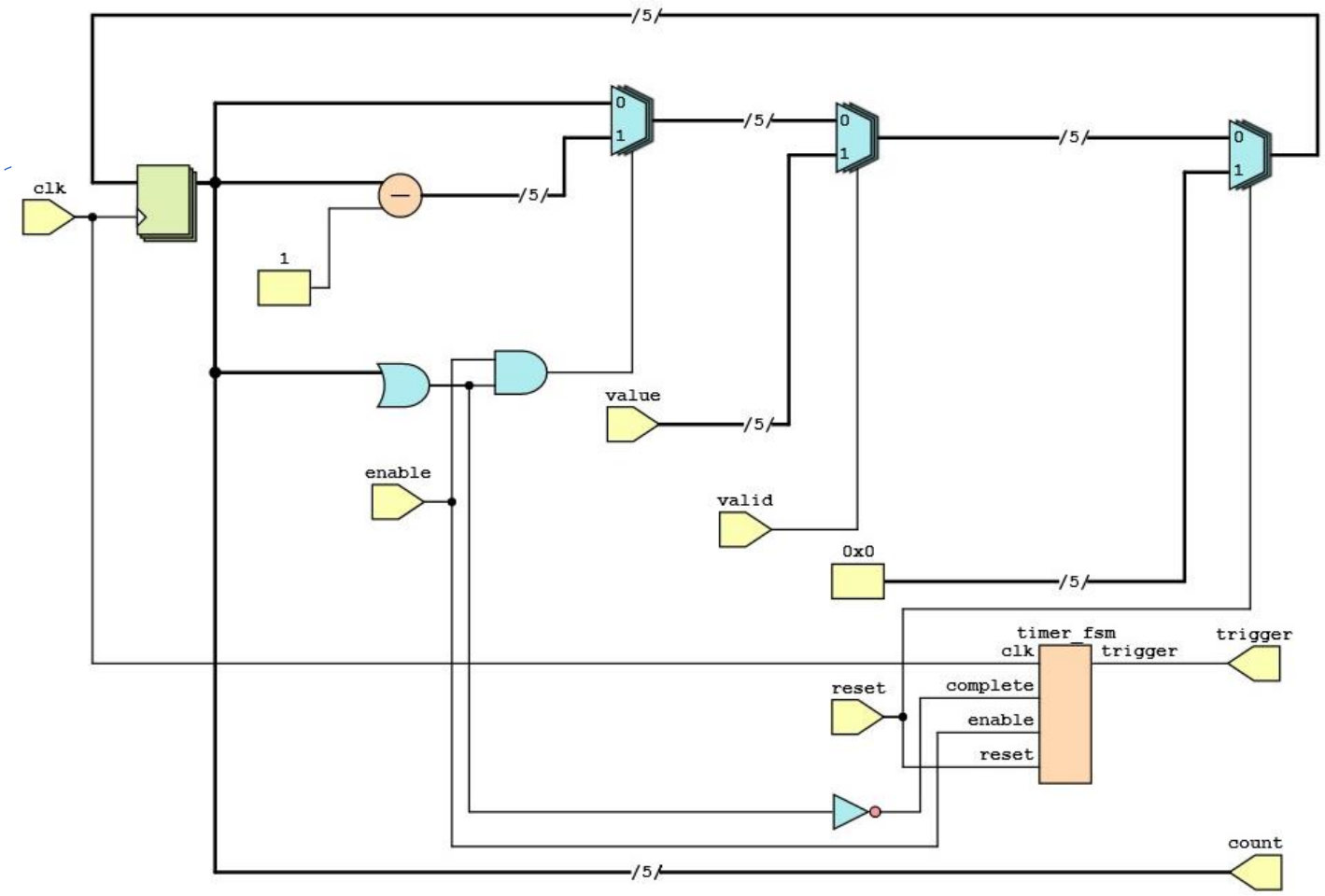
Multiplexors:
- Have inputs on the ***left***
- A single output on the ***right***
- MUX (decide) based on the *Top or Bottom inputs.*

Analyzing a schematic for Verilog implementation.
Now for the difficult part: Making sense of the LOGIC!

The **multiplexor chain** is making decisions based on some values -> Its just an **IF-Else statement**!

Look at the chain from back to front.
Multiplexor C being last:
*Counts down the count if: **enabled AND count is above zero**!*
Else the count stays the same.

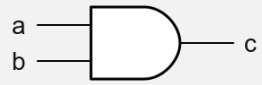
```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete = count == 5'b0;
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
22 always @(posedge clk) begin
23     if(reset) begin
24         count <= 5'b0;
25     end else if (valid) begin
26         count <= value;
27     end else if(count > 0 && enable) begin
28         count <= count - 1;
29     end else begin
30         count <= count;
31     end
32 end
33 endmodule
34
```



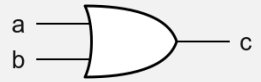
Analyzing a schematic for Verilog implementation.
With that you have turned a schematic into Verilog!

```
1  module timer(
2      input clk,
3      input reset,
4      input enable,
5      input [4:0] value,
6      input valid,
7      output trigger,
8      output reg [4:0] count
9
10 );
11
12 wire complete = count == 5'b0;
13
14 timer_fsm fsm1 (
15  .clk(clk),
16  .reset(reset),
17  .enable(enable),
18  .complete(complete),
19  .trigger(trigger)
20 );
21
22 always @(posedge clk) begin
23     if(reset) begin
24         count <= 5'b0;
25     end else if (valid) begin
26         count <= value;
27     end else if(count > 0 && enable) begin
28         count <= count - 1;
29     end else begin
30         count <= count;
31     end
32 end
33 endmodule
34
```

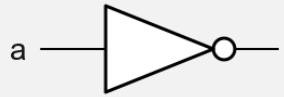
Logic gates:



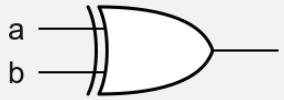
AND: c = a & b;



OR: c = a | b;



NOT: b = ~a;



XOR: c = a ^ b;