EECS151/251A Discussion 3

Christopher Yarp

Feb. 8, 2019

Latches

Latches

- There are several different flip-flop and latch types discussed in literature and the nomenclature can disagree at times.
- In this class:
  - a “flip-flop” is assumed to be a d flip-flop (edge triggered)
  - a “latch” is assumed to be a device that passes a signal through when enabled (transparent) and holds the most recent value of its output when disabled (opaque)
- They key difference between the flip-flop and latch is that the flip-flop only updates it’s output on a clock edge while the latch updates its output whenever its enable line is high and holds it otherwise

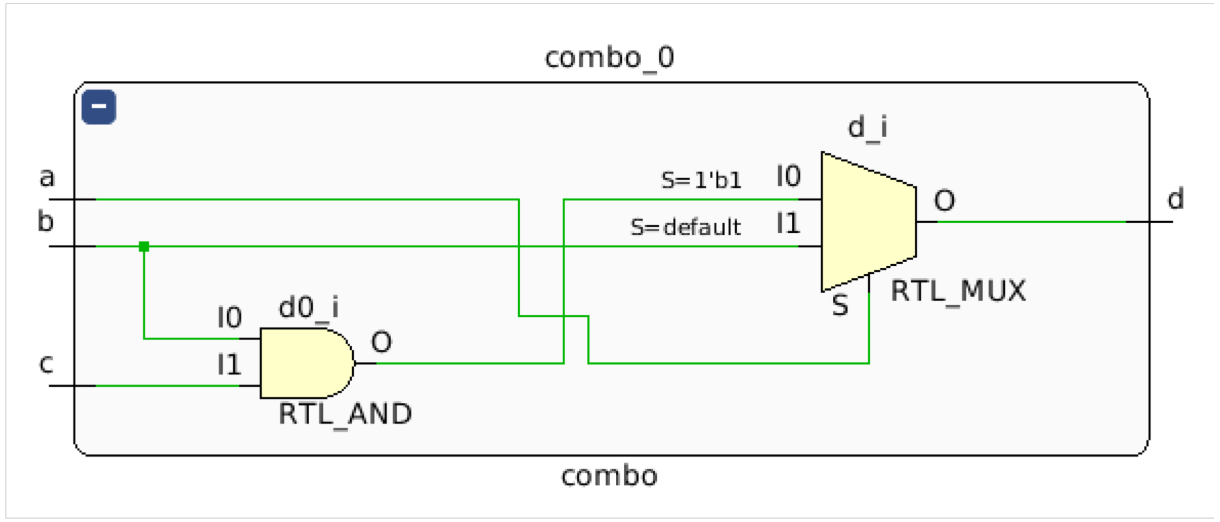
Inferring Latches in Verilog

- Latches can be inferred in Verilog by not assigning a reg type in all possible cases
- In the case when the reg type was not assigned, Verilog assumes it’s value should remain unchanged
  - This requires the previous value to be held …
  - … which requires a latch

Inferring Latches in Verilog

Combinational

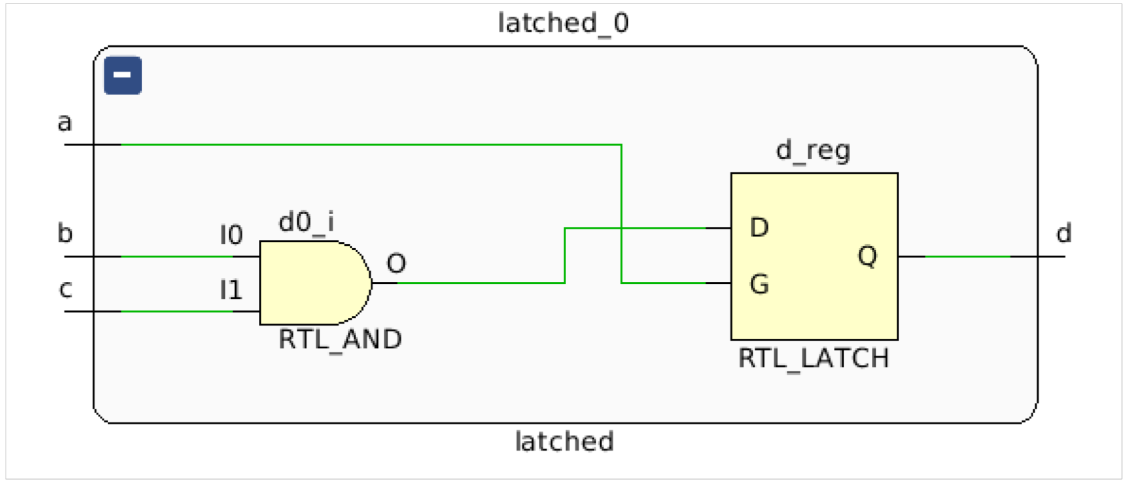
```
module combo(input a, input b, input c, output reg d);
always @(*) begin
    if(a) begin
        d = b & c;
    end else begin
        d = b;
    end
end
endmodule
```



Latched (Sequential)

```
module latched(input a, input b, input c, output reg d);
always @(*) begin
    if(a) begin
        d = b & c;
    end
end
endmodule
```

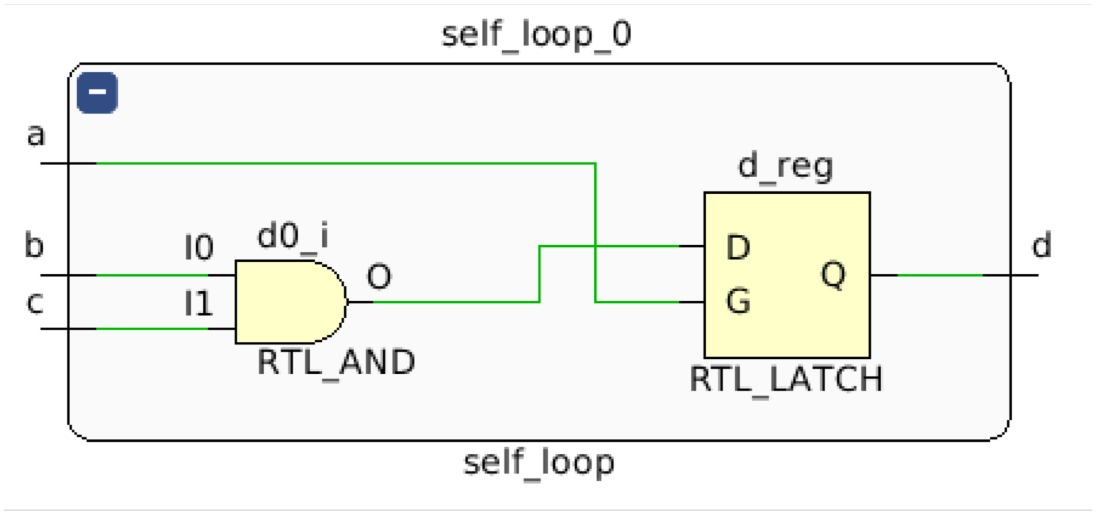
WARNING: [Synth 8-327] inferring latch for variable 'd_reg'



Inferring Latches in Verilog

- Assigning a reg type to itself in an always @(*) block also infers a latch

```
module self_loop(input a, input
b, input c, output reg d);
always @(*) begin
    if(a) begin
        d = b & c;
    end else begin
        d = d;
    end
end
endmodule
```



Unintentional Latch Inference

- Unintentional latch inference is one of the more common bugs you will probably come across
- Especially when writing a big case statement or if/else tree (for example in an FSM), it can be easy to overlook assigning a reg type in all cases
- Many synthesis tools will emit a warning if they infer a latch, look for these warnings and make sure any you see are intentional
  - Vivado Synthesis Log: WARNING: [Synth 8-327] inferring latch for variable 'd_reg'

Updating our understanding of always @ blocks

always @(*)

- Describes a combinational circuit if reg types are assigned in all possible cases
- Describes a sequential circuit (with latches) if reg types are not assigned in all cases
    - Or are assigned to themselves
- **Describes a sequential circuit (with latches) if not all signals are in the sensitivity list
    - The synthesis tools may assume you meant to include the signal in the sensitivity list -> Vivado does
    - Your simulator may not make the same assumption!

Always @(posedge clk)

- Describes sequential circuits (with flip-flops)

Blocking vs. Non-Blocking Assignment Conventions

… and why you should follow them

Sequential Logic

Non-Blocking

```
always @(posedge clk)
begin
    c <= a & b;
    e <= c | d;
end
```

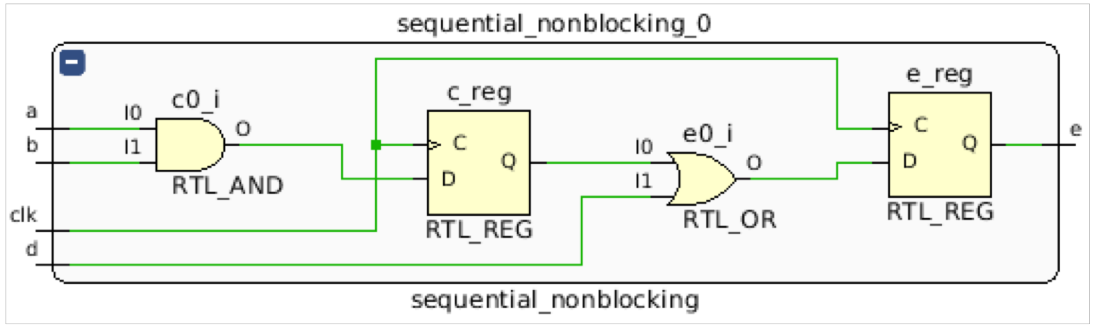
Blocking

```
always @(posedge clk)
begin
    c = a & b;
    e = c | d;
end
```

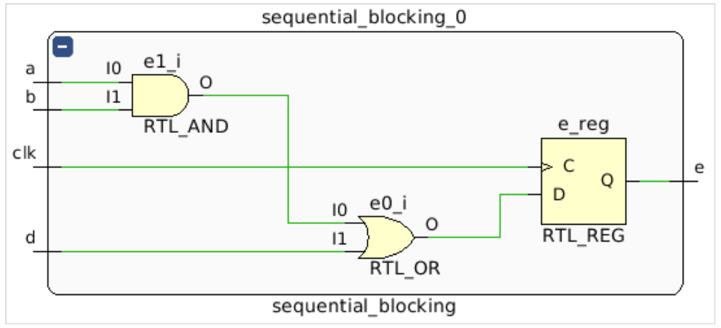
Are these the same module?

Sequential Logic

Non-Blocking



Blocking



Cycle	A	B	D	Non-Blocking	Blocking
0	0	1	0	x	x
1	1	1	0	x	0
2	1	1	0	0	1
3	1	1	0	1	1
4	1	1	0	1	1

Are these the same module? – No

Combinational Logic

Blocking

```
always @(a, b, d)
begin
   c = a & b;
   e = c | d;
end
```

Non-Blocking
(Input Sensitivity)

```
always @(a, b, d)
begin
   c <= a & b;
   e <= c | d;
end
```

Non-Blocking
(Intermediate Sensitivity)

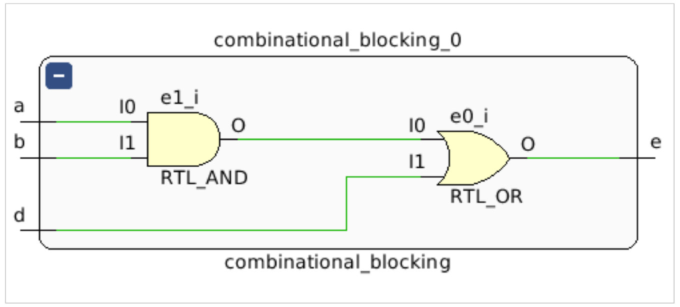
```
always @(a, b, d, c)
begin
   c <= a & b;
   e <= c | d;
end
```

Are these the same module?

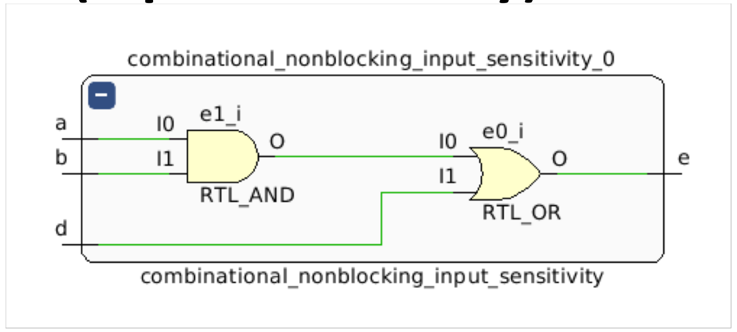
Combinational Logic

WARNING: [Synth 8-567] referenced signal 'c' should be on the sensitivity list [/home/cyarp/test_blocking/combinational_nonblocking_input_only.v:33]

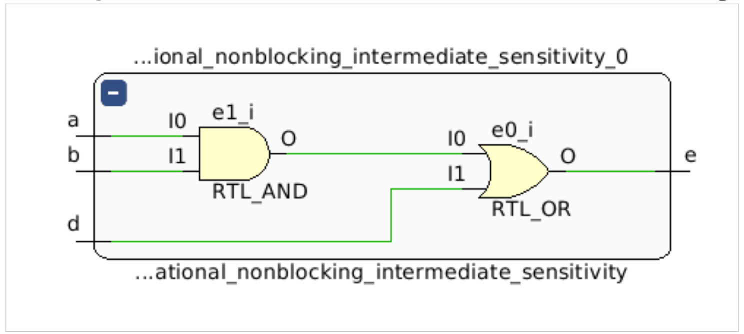
Blocking



Non-Blocking (Input Sensitivity)


combinational_nonblocking_input_sensitivity_0
a
b
d
I0
I1
e1_i
O
RTL_AND
I0
I1
e0_i
O
RTL_OR
e
combinational_nonblocking_input_sensitivity

Non-Blocking (Intermediate Sensitivity)


...ional_nonblocking_intermediate_sensitivity_0
a
b
d
I0
I1
e1_i
O
RTL_AND
I0
I1
e0_i
O
RTL_OR
e
...ational_nonblocking_intermediate_sensitivity

Cycle	A	B	D	Blocking	Non-Blocking (Input Sensitivity)	Non-Blocking (Intermediate Sensitivity)
0	0	1	0	0	x	0
1	1	1	0	1	0	1
2	1	1	0	1	0	1
3	1	1	0	1	0	1

Looks like a latch.
Where is it??

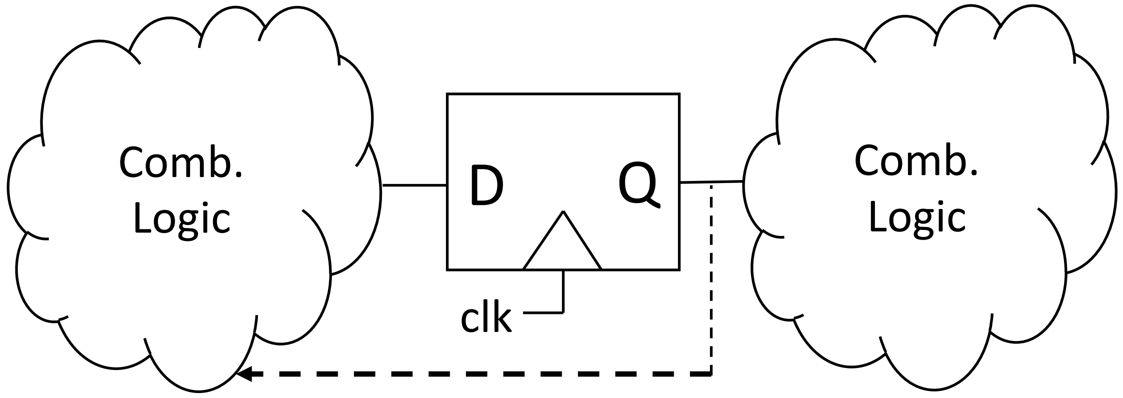
Are these the same module? – ??

Keep to the Rules of Thumb

- Sequential Logic: Use non-blocking assignments
- Combinational Logic: Use blocking assignments
- You can always break up your sequential logic into combinational and sequential components
    - Allows you to cleanly have intermediates in your combinational logic

```
always @(*)
begin
      c = a & b;
      next_state = c | state;
end

always @(posedge clk)
begin
      state <= next_state;
end
```



Implementing Logic Functions with LUTs

LUT

- LUT is short for “Look Up Table”
- The number of rows in the table is 2^N where N = number of input bits
- There is 1 row for every possible input combination
  - If you view the inputs as a single multiple bit wide wire, you can think of it as specifying an *address* in the LUT
- The designer determines what the output will be for each row of the table

Implementing Functions with LUTs

- You can view the entries of an N-input LUT as being entries in a truth table for an N-input combinational logic block
- Since the LUT contains a row for every possible combination of inputs, we can implement any combination function by specifying the output values for each row in the table

What function is this?

- This LUT only outputs 1 when A, B, and C are all true.
- This LUT is implementing A&B&C

{C,B,A}	C	B	A	Out
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

What function is this?

- This outputs 1 only when exactly 1 of A, B, and C are true
- A&(~B)&(~C) | (~A)&B&(~C) | (~A)&(~B)&C

{C,B,A}	C	B	A	Out
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
7	1	1	0	0
7	1	1	1	0

Implementing Functions with LUTs

- If you can write a truth table for it (which you can with any combinational block), you **can** implement it with a single LUT!
- This does **not** necessarily mean you **should** implement all combinational functions with a single LUT:
  - A combinational block with 64 inputs would require a LUT of $2^{64} \cong 1.84\text{x}10^{19}$ entries! Just storing all of the output bits would require 2305843 TB of data!
    - When would you want a 64 input combinational block? How about a 32 bit adder (32 bits for each input operand)
  - There is likely a more efficient way of implementing a 64 input combinational block

Building Larger LUTs

… with smaller LUTs

Building Larger LUTs

- Let’s say we have 3 input LUTs, is there a way we could create a 4 input LUT?
- Yes, let’s look at the truth table for a 4 input LUT
- The bottom half of the table looks like a repeat of the top half of the table except …
  - The top half of the table is when d is 0, the bottom half is when d is 1
  - The top and bottom halves of the table have different outputs

d	c	b	a	out
0	0	0	0	o_1
0	0	0	1	o_2
0	0	1	0	o_3
0	0	1	1	o_4
0	1	0	0	o_5
0	1	0	1	o_6
0	1	1	0	o_7
0	1	1	1	o_8
1	0	0	0	o_9
1	0	0	1	o_{10}
1	0	1	0	o_{11}
1	0	1	1	o_{12}
1	1	0	0	o_{13}
1	1	0	1	o_{14}
1	1	1	0	o_{15}
1	1	1	1	o_{16}

Let’s split the table

d	c	b	a	out
0	0	0	0	o_1
0	0	0	1	o_2
0	0	1	0	o_3
0	0	1	1	o_4
0	1	0	0	o_5
0	1	0	1	o_6
0	1	1	0	o_7
0	1	1	1	o_8
1	0	0	0	o_9
1	0	0	1	o_{10}
1	0	1	0	o_{11}
1	0	1	1	o_{12}
1	1	0	0	o_{13}
1	1	0	1	o_{14}
1	1	1	0	o_{15}
1	1	1	1	o_{16}

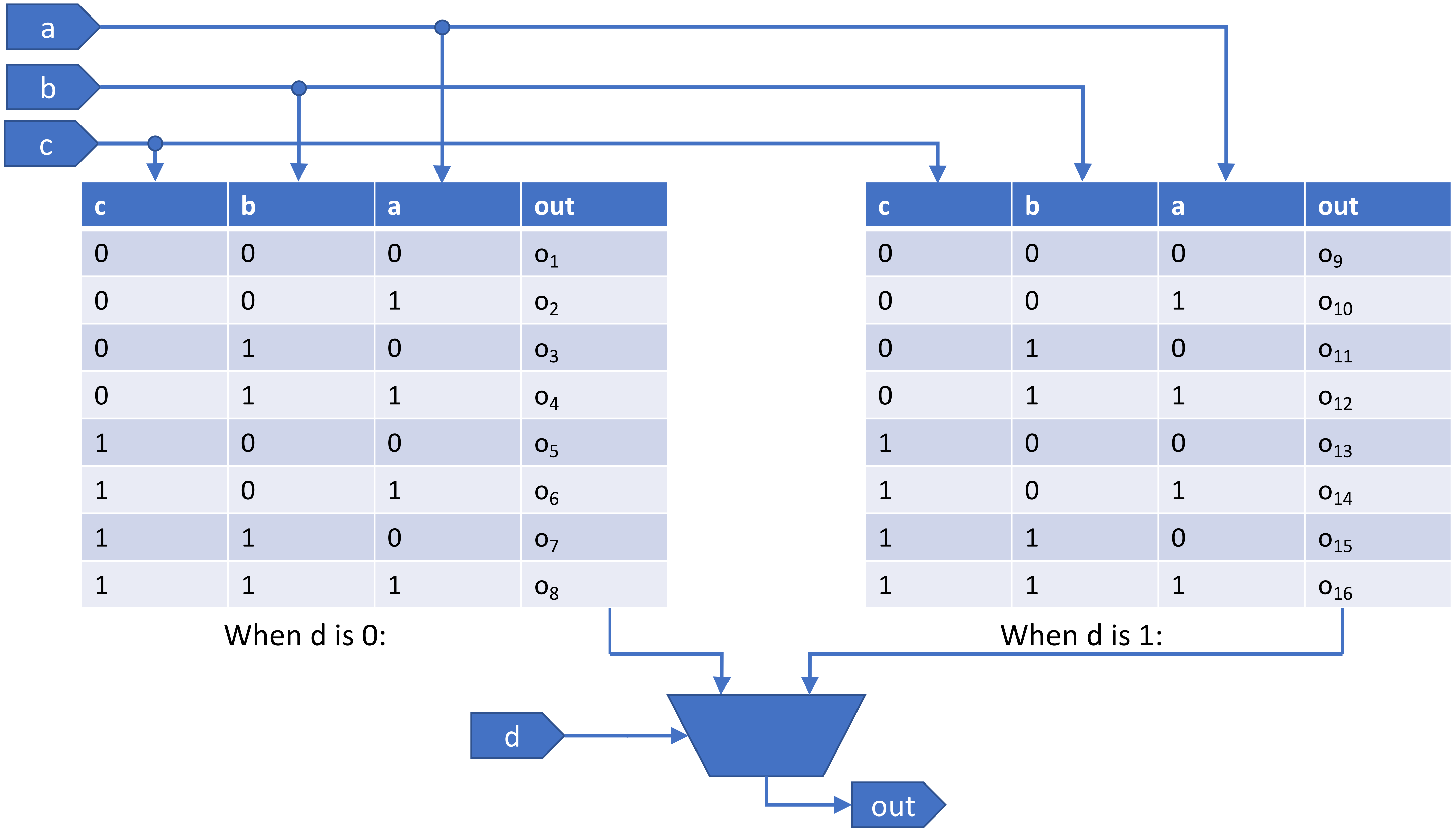
When d is 0:

c	b	a	out
0	0	0	o_1
0	0	1	o_2
0	1	0	o_3
0	1	1	o_4
1	0	0	o_5
1	0	1	o_6
1	1	0	o_7
1	1	1	o_8

When d is 1:

c	b	a	out
0	0	0	o_9
0	0	1	o_{10}
0	1	0	o_{11}
0	1	1	o_{12}
1	0	0	o_{13}
1	0	1	o_{14}
1	1	0	o_{15}
1	1	1	o_{16}

Select which table to use



c	b	a	out
0	0	0	o_1
0	0	1	o_2
0	1	0	o_3
0	1	1	o_4
1	0	0	o_5
1	0	1	o_6
1	1	0	o_7
1	1	1	o_8

When d is 0:

c	b	a	out
0	0	0	o_9
0	0	1	o_{10}
0	1	0	o_{11}
0	1	1	o_{12}
1	0	0	o_{13}
1	0	1	o_{14}
1	1	0	o_{15}
1	1	1	o_{16}

When d is 1:

Simulation

Initial blocks

- Initial blocks are executed at the start of the simulation
  - In general, initial blocks are not synthesized
- You can have multiple initial blocks in the same module
  - Each will run in parallel starting at the beginning of the simulation

```
initial begin
   $dumpfile("dump.vcd");
   $dumpvars;

   a = 1;
   #4
   a = 0;
   #10
   $finish();
 end
```

```
initial begin
   #1
   forever begin
     $strobe("time: %4d, a: %b, b: %b, c:
%b", $time, a, b, c);
     #2;
   end
 end
```

Forever blocks

- Will execute a sequence of Verilog statements repeatedly forever (until the simulation ends)

```
initial begin
   #1; //Wait 1 Cycle
   forever begin //Print out values every 2 cycles
     $strobe("time: %4d, a: %b, b: %b", $time, a, b);
     #2;
   end
end
```

Waiting for Events

- So far, we have used the delay statement, #
- If we want to wait for a particular even to happen, we can use another type of statement: @(posedge signal) or @(negedge signal)

```
initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
    a = 1;
    @(negedge clk) //Continue simulation until the next negative edge of clk
    a = 0;
    @(negedge clk) //Continue simulation until the next negative edge of clk
    #4 //Continue for 4 periods
    $finish();
  end
```

Repeat blocks

- You may want to repeat a block of Verilog a fixed number of times
- Repeat blocks let you do that

```
repeat (10) begin //Repeat this 10 times
    signal = ~glitchy_signal;
    @(negedge clk);
end
```

And even more …

- There are other loop statements in Verilog
  - For loops
  - While loops
  - *Note that these are different from generate loops*
- You can also fork multiple parallel threads of execution in an initial block, wait for all of them to finish, then continue executing Verilog expressions
  - These are called fork-join blocks
  - Having multiple initial blocks is also a way to create multiple parallel threads of execution that all start when the simulation starts