YORK
UNIVERSITÉ
UNIVERSITY

EECS 3201: Digital Logic Design Lecture 13

Ihab Amer, PhD, SMIEEE, P.Eng.

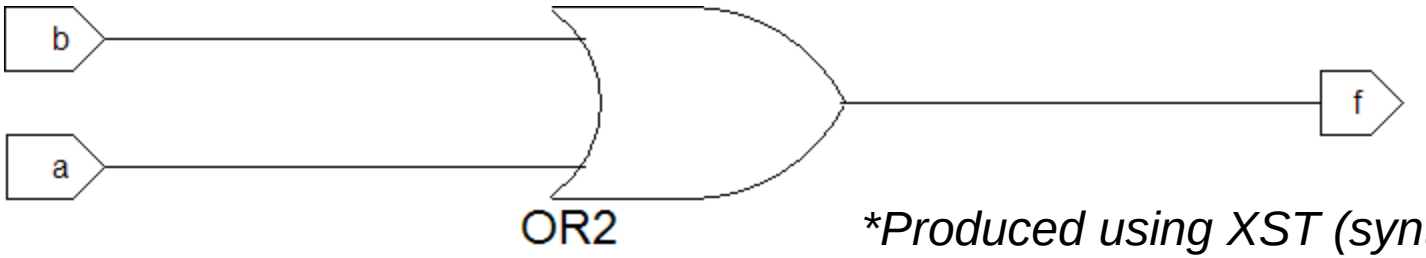
Verilog HDL for Synthesis

- Verilog HDL was originally designed as a logic circuit *description* and *simulation* language, and was later adopted to *synthesis*
- Thus, the language has several features and constructs that cannot be synthesized
- Even if synthesizable code is used, it should be taken care that the style of the written code can have a big impact on the quality of synthesized circuits

Using **always** Blocks for Combinational Logic

```
always @(a or b) begin
    if (a== 1'b0 && b==1'b0)
      f =1'b0;
    else
      f =1'b1;
  end
```

*Synthesis**



**Produced using XST (synthesis tool of Xilinx ISE 10.1). Target Device: XC3S500E-4FG320. Same applies for rest of examples.*

Rules to Infer Combinational Logic

- All inputs to the circuit should be included in the sensitivity list
- No other signals should be included in the sensitivity list
- None of the statements within the **always** block can be sensitive to rising or falling edges of any signals

Examples' Template

```
// DEFINITIONS
`define VEC_SIZE 2

// MODULE DECLARATION
module seq1 (vec1, c, d, e, f, g, clk, a, b);

  output [`VEC_SIZE-1:0] vec1;          // bit vector output
  output c, d, e, f, g;            // single bit outputs output
  input clk;                                // clock signal
  input a, b;                               // data inputs

  // SIGNAL DECLARATIONS
  reg c, d, e, f, g;               // connects to the outputs
  reg [`VEC_SIZE-1:0] vec1;   // bit vector output
  reg [`VEC_SIZE-1:0] vec2;   // other bit vectors
  wire [`VEC_SIZE-1:0] vec3;

 // ** MODULE UNDER SYNTHESIS TEST GOES HERE

endmodule
```

Example 1

```
always @(a or b) begin
   if (b)
     c = a;
  end
```

Synthesis

Inferred latch. Typically, unintended

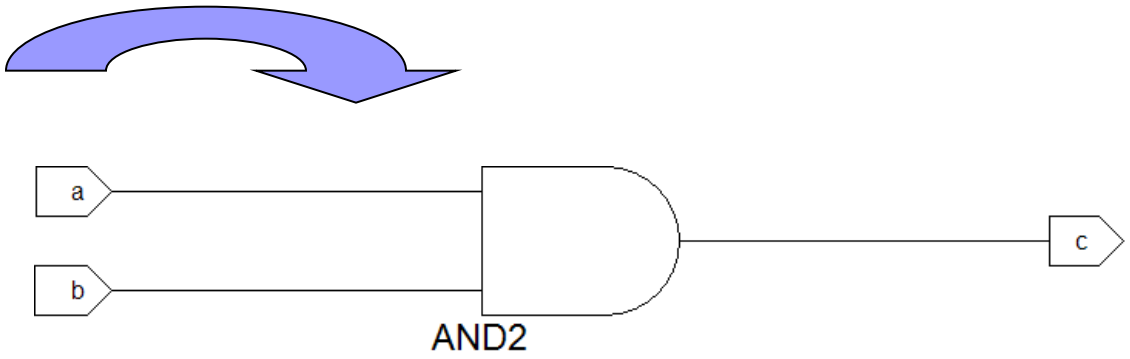
Workaround

```
always @(a or b) begin
   if (b)
      c = a;
  else
      c = 1'bx;
end
```

OR

```
always @(a or b) begin
     c = 1'bx;
     if (b)
        c = a;
 end
```

Synthesis



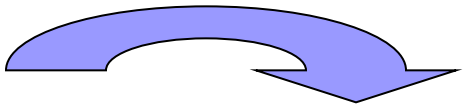
Pure combinational logic.
Inferred latch disappeared!

Synthesis

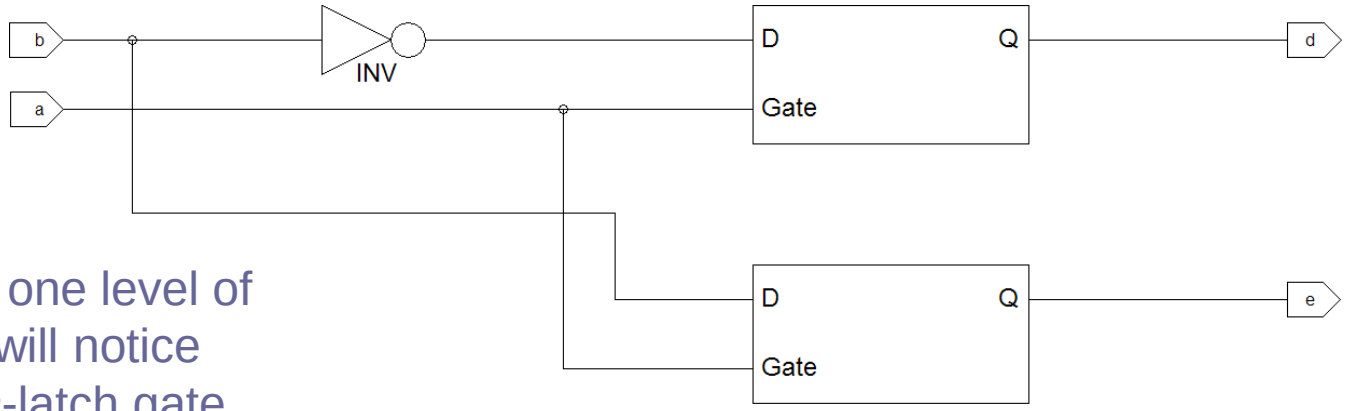
Example 2

```
always @(a or b) begin
    if (a)
        d = ~b;
    else
        e = b;
end
```

Synthesis



Two inferred latches!



If you go down one level of hierarchy, you will notice that the *bottom*-latch gate port is inverted. Why?

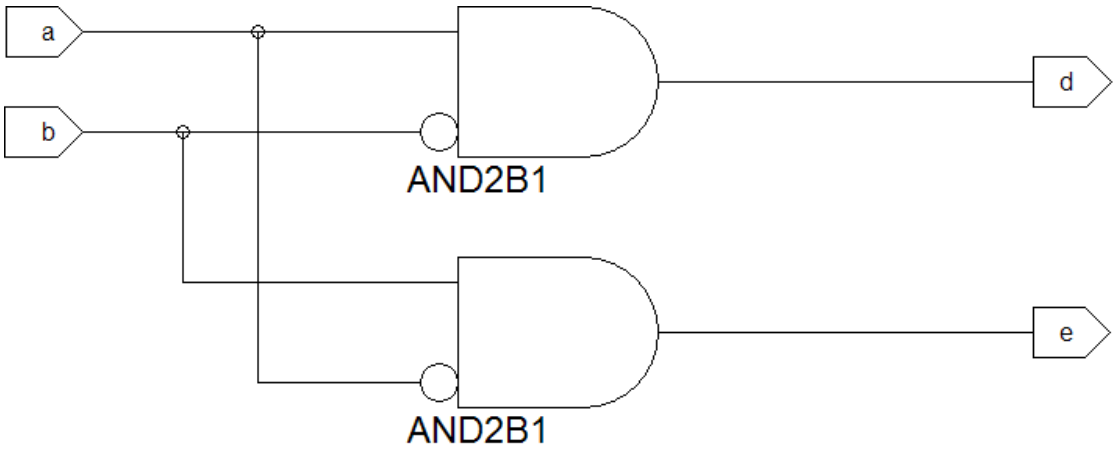
Workaround

```
always @(a or b) begin
   if (a)
      begin
         d = ~b;
         e = 1'bx;
      end
   else
      begin
         e = b;
         d = 1'bx;
      end
end
```

OR

```
always @(a or b) begin
   d = 1'bx;
   e = 1'bx;
   if (a)
         d = ~b;
   else
         e = b;
end
```

Synthesis



Synthesis

Pure combinational logic.
Inferred latches disappeared!

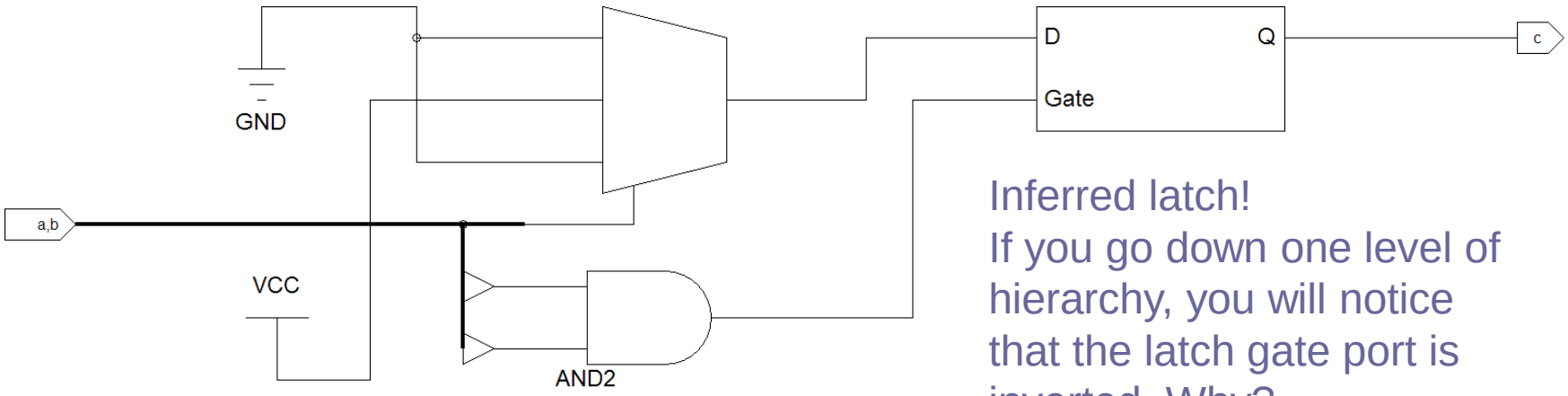
Moral

- Latches are created by **if-else** statements with no **else** clause. Such latches are typically unintended
- Workaround – Two possible solutions:
  - Every **if** statement should have an **else** clause. Every variable that is assigned a value in one **if** or **else** clause, should also be assigned a value in every other clause (even if we do not care about this value)
  - Unconditionally assign default values to variables at the beginning of the **always** block

Example 3

```
always @(a or b)
case ({a,b})
      2'b00: c = 1'b0;
      2'b01: c = 1'b1;
      2'b10: c = 1'b0;
endcase
```

Synthesis



Inferred latch!
If you go down one level of hierarchy, you will notice that the latch gate port is inverted. Why?

Workaround

```
always @(a or b)
case ({a,b})
      2'b00: c = 1'b0;
      2'b01: c = 1'b1;
      2'b10: c = 1'b0;
      2'b11: c = 1'bx;
endcase
```

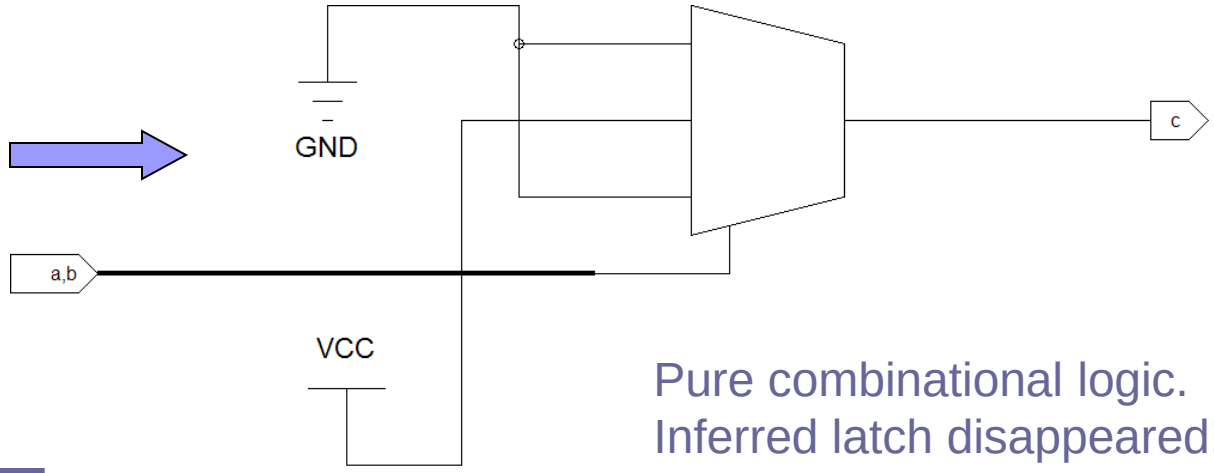
OR

```
always @(a or b)
case ({a,b})
      2'b00: c = 1'b0;
      2'b01: c = 1'b1;
      2'b10: c = 1'b0;
      default: c = 1'bx;
endcase
```

OR

```
always @(a or b) begin
c = 1'bx;
case ({a,b})
      2'b00: c = 1'b0;
      2'b01: c = 1'b1;
      2'b10: c = 1'b0;
endcase
end
```

Synthesis



Synthesis

Pure combinational logic.
Inferred latch disappeared!

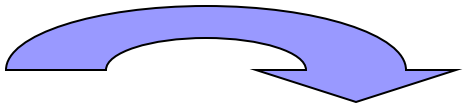
Moral

- Latches are created by **case** statements that miss the code for an outcome of some input combinations. Such latches are typically unintended
- Workaround – Three possible solutions:
  - Every **case** statement should be a "*full*" ***case*** (covering all input combinations). Every variable that is assigned a value in one **case**, should also be assigned a value in every other **case** (even if we do not care about this value)
  - Unconditionally assign default values to variables. This can be done at the beginning of the **always** block or using the **default** statement
  - Using default values is a good coding practice, even though the **case** choices are *all-inclusive*. It accounts for 'x' and 'z' values at the inputs (especially for simulation)

Example 4

```
always @(a) begin
    if (~a)
        vec2 = ~vec3;
end
```

Synthesis



Nothing!
Why?!

The output vec2 is an internal signal value that is not used to generate any output signal – thus, this section of code is redundant, and is optimized out by the synthesis tools

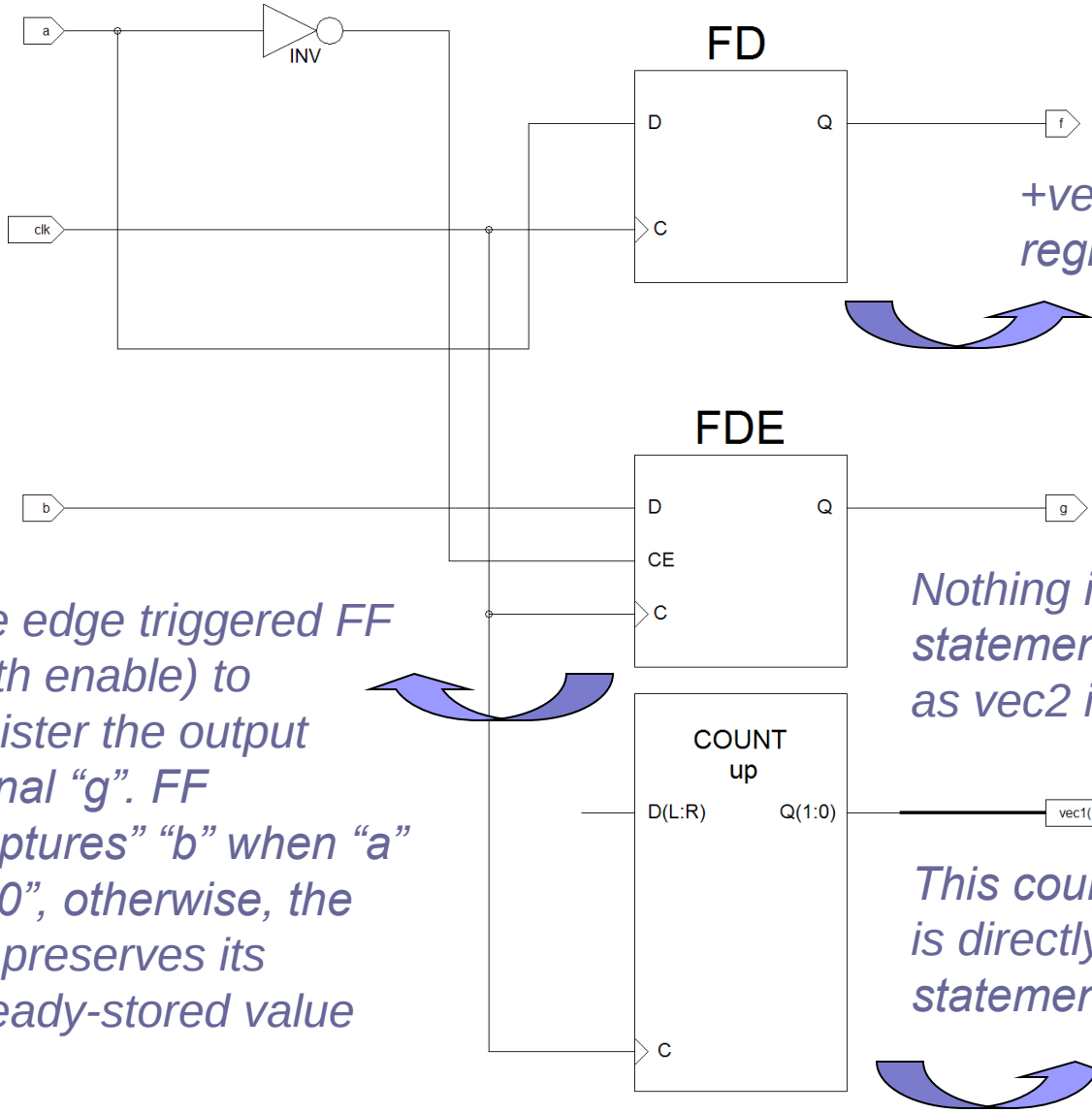
Moral

- Any logic that is deemed to be redundant may be optimized out by the synthesis tool
- Even a structural Verilog description may not result in the same structural connections once it is synthesized
- If the designer would like to retain a specific structure, then synthesis options must be used to disable “flattening” of the hierarchy. Submodules must be created (in order to create an artificial hierarchy), and “Don’t Touch” options must be checked for those submodules that must not be optimized

Example 5

```
always @(posedge clk) begin
    f <= a;                          // using nonblocking assignment
    vec1 <= vec1 + 1;                // counter
    vec2 <= vec2 + 1;                // This part will be optimized out
    case (a)
        1'b0: g <= b;
        //default: g <= 1'bx; // What happens if this is uncommented?
    endcase                          // note that case 1'b1 is not covered
end
```

Example 5 – Synthesis Results



+ve edge triggered FF to register the output signal "f"

+ve edge triggered FF (with enable) to register the output signal "g". FF "captures" "b" when "a" is "0", otherwise, the FF preserves its already-stored value

Nothing is synthesized for the statement: "vec2 <= vec2 + 1"; as vec2 is a redundant internal signal

This counter (2 FF's and logic gates) is directly synthesized by this statement: "vec1 <= vec1 + 1;"

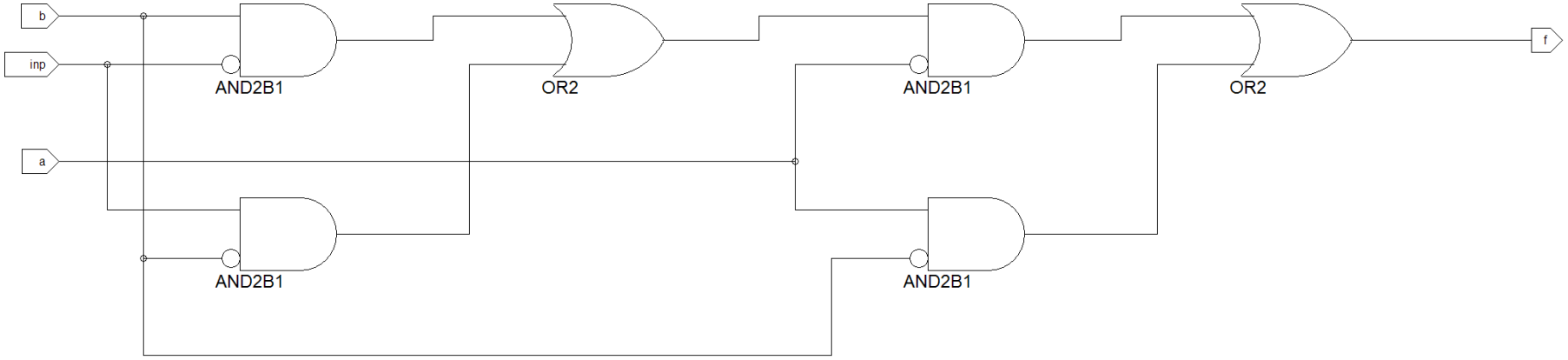
Example 6*

*Multi-level nested **if-else** statement*

```
always @(a or b or inp) begin
        if (a==1'b0)
                if (b==1'b0)
                        if (inp==1'b0)
                                f = 1'b0;
                        else
                                f = 1'b1;
                else
                        if (inp==1'b0)
                                f = 1'b1;
                        else
                                f = 1'b0;
        else
                if (b==1'b0)
                        if (inp==1'b0)
                                f = 1'b1;
                        else
                                f = 1'b1;
                else
                        if (inp==1'b0)
                                f = 1'b0;
                        else
                                f = 1'b0;
end
```

**An input port (inp) is added to the module template for this specific example*

Example 6 – Synthesis Results



-Completely Serial Circuit

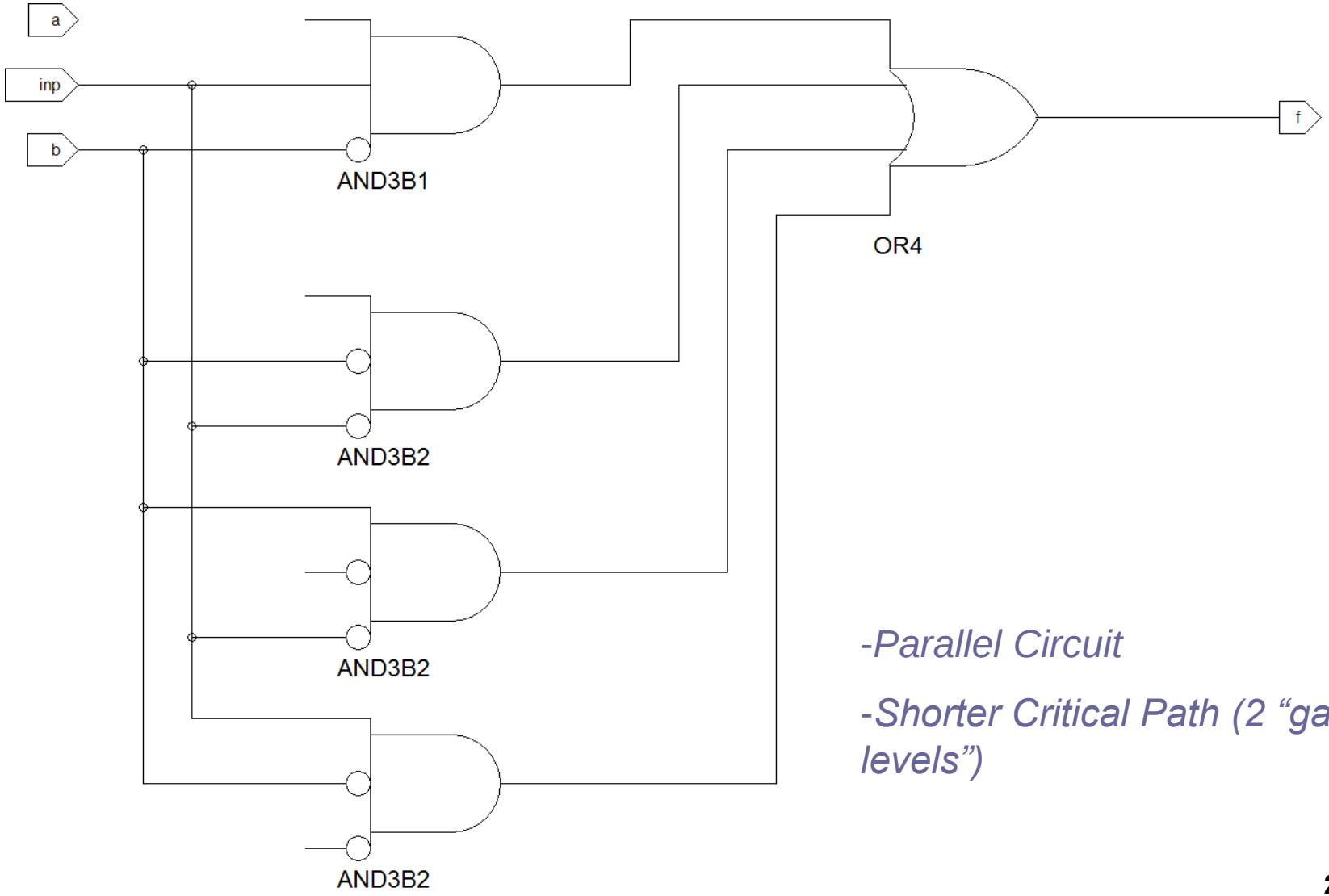
-Long Critical Path (4 “gate-levels”)

Example 6 – Workaround I

*Parallel **if-else** statement*

```
always @(a or b or inp) begin
        if (a==1'b0 && b==1'b0 && inp==1'b0)
                f = 1'b0;
        else if (a==1'b0 && b==1'b0 && inp==1'b1)
                f = 1'b1;
        else if (a==1'b0 && b==1'b1 && inp==1'b0)
                f = 1'b1;
        else if (a==1'b0 && b==1'b1 && inp==1'b1)
                f = 1'b0;
        else if (a==1'b1 && b==1'b0 && inp==1'b0)
                f = 1'b1;
        else if (a==1'b1 && b==1'b0 && inp==1'b1)
                f = 1'b1;
        else if (a==1'b1 && b==1'b1 && inp==1'b0)
                f = 1'b0;
        else
                f = 1'b0;
end
```

Workaround I – Synthesis Results



-Parallel Circuit

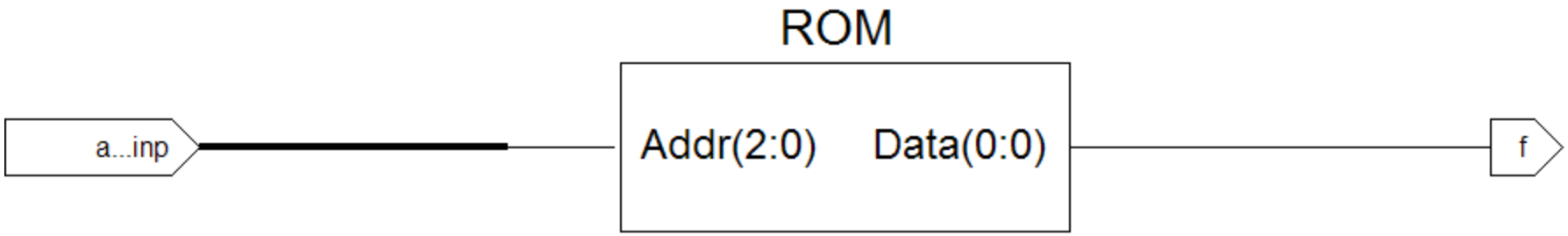
-Shorter Critical Path (2 "gate-levels")

Example 6 – Workaround II

*Parallel **case** statement*

```
always @(a or b or inp)
        case ({a,b,inp})
                3'b000: f = 1'b0;
                3'b001: f = 1'b1;
                3'b010: f = 1'b1;
                3'b011: f = 1'b0;
                3'b100: f = 1'b1;
                3'b101: f = 1'b1;
                3'b110: f = 1'b0;
                default: f = 1'b0;
        endcase
```

Workaround II – Synthesis Results



-Optimized ROM (or MUX) circuit

-2 "gate-levels"

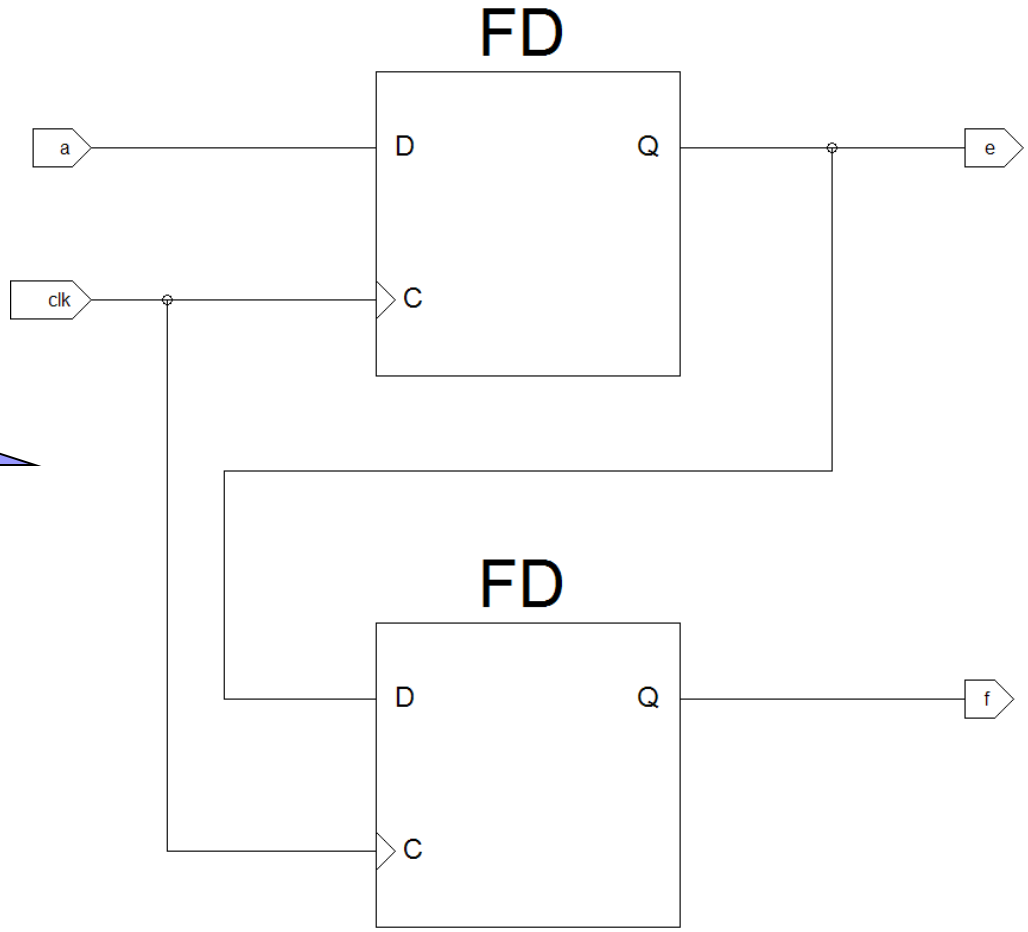
Moral

- "Serial" control structures (like nested **if-else** statements) can result in corresponding serial chain of logic gates to test conditions
- It is often better to use a **case** statement, especially if the conditions are *mutually exclusive* (so can be evaluated in parallel). This is strongly valid for situations with two or more cases

Non-blocking Assignment

```
always @(posedge clk) begin
        e <= a;
        f <= e;
end
```

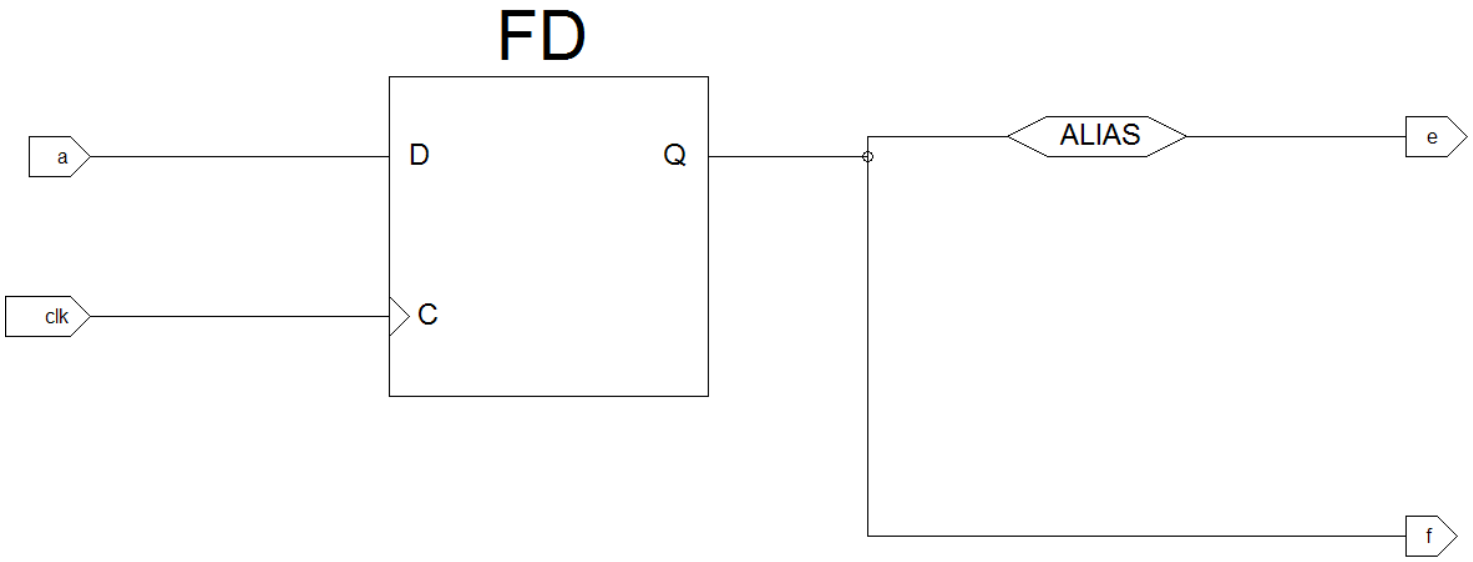
Synthesis



Blocking Assignment

```
always @(posedge clk) begin
        e = a;
        f = e;
end
```

Synthesis



Examples of Non-synthesizable Verilog HDL Features

- The “delay” statements “#”
- The “**initial**” blocks
- Usage of sensitivity lists that contain data inputs beside the clock-edge (other than the RST)
- **Always** blocks that are sensitive to both the *rising* and the *falling* edges of the same signal
- Such statements still can be useful in:
  - Testbenches (as they are not synthesized; they are generally used to test the behavior of the circuit)
  - Early-stage models of the circuit to perform early verification of its behavior

Concept of Simulation

- Pretends to simulate concurrent statements using event lists and sensitivity matrix, based on all the individual sensitivity lists
- Each concurrent statement gives rise to at least one SW process in the simulator
- At every *simulation cycle*, the “concurrent” processes are executed one after the other
- After a simulation cycle completes, the event list is scanned for the signal(s) that change at the earliest time on the list, and the simulation time advances to this time
- All the processes that are sensitive to a signal that just changed are scheduled for execution in the next simulation cycle, which now begins

Further Recommended Coding Practices

- Look at the synthesis options, and choose the ones that suits your needs
- Inspect the synthesized circuit whenever possible
- Loops generally create multiple copies of combinational logic. If you prefer re-using one copy of combinational logic, then use an FSM-controlled sequential circuit
- Avoid using **always** blocks with empty sensitivity list

Practice Assignment

- Redo all examples of this lecture using Quartus II RTL Viewer
- Analyze the difference(s) if any, and consult your professor if you fail to explain it

References

- Digital Design, 4th Edition, John Wakerly
- Advanced Digital Logic Design, Sunggu Lee
- www.ece.cmu.edu/~thomas/VSLIDES.pdf