
# ISTANBUL TECHNICAL UNIVERSITY

## Digital System Design & Applications

## *Verilog - Procedural Statements 1*

RES. ASST. FIRAT KULA

# So far…

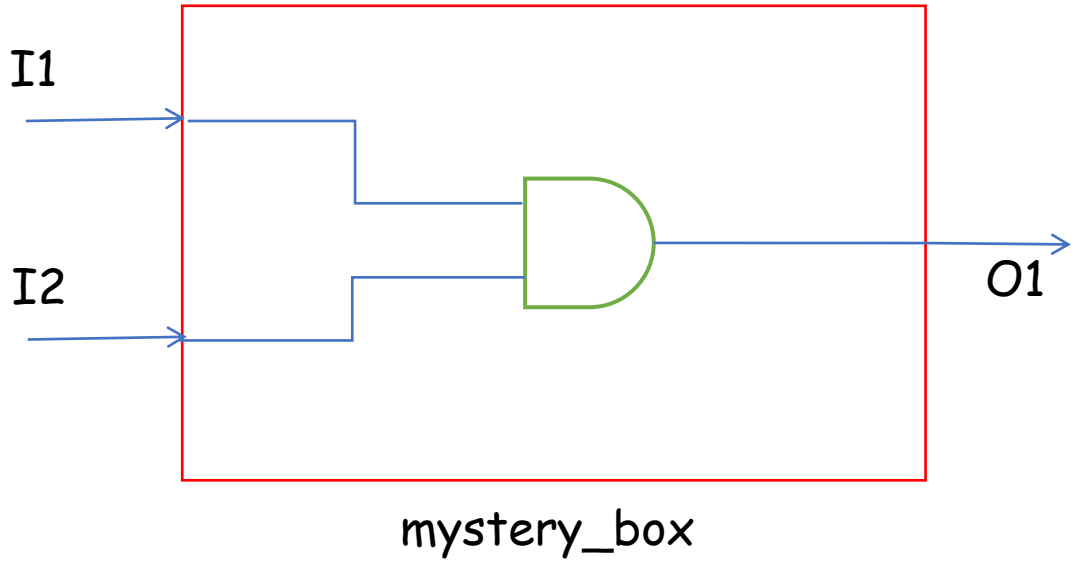
- Summary for designing purely combinational circuits with Verilog

Our arsenal:

- **Dataflow modeling:**
  - Continuous assignment
  - Operators
  - Procedural blocks (initial,always) with blocking assignments
- **Structural modeling:**
  - Module instances (submodules)
  - Primitives
- **Behavioral modeling:**
  - Procedural blocks (initial,always) with blocking assignments
  - Procedural statements (if,else,case…)

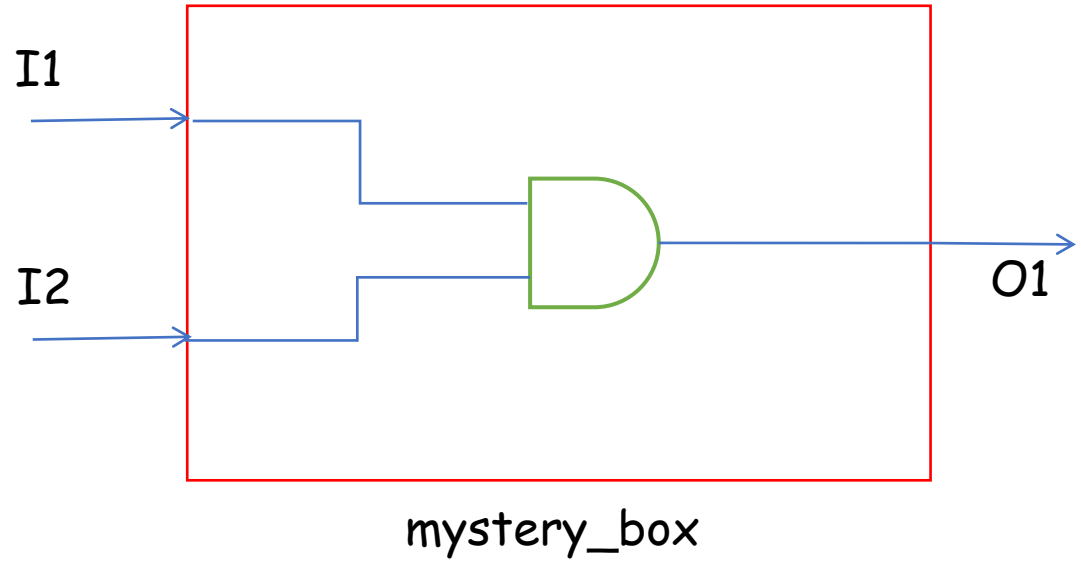
# So far…

```
module mystery_box(
    input I1,
    input I2,
    output reg O1
);

    initial I1=0;
    initial I2=0;

    always @(I1,I2)
        O1 = I1 & I2;

    // Functionally same as before!

endmodule
```



# So far…

```
module mystery_box(
    input I1,
    input I2,
    output reg O1
);

    initial I1=0;
    initial I2=0;

    always @(*)
    begin
        if(I1==1 && I2==1)
            O1 = 1;
        else
            O1 = 0;
    end

    // Functionally same as before!

endmodule
```




- We modeled the same circuit, but in a different way

# Behavioral Modeling

- Why bother modeling combinational circuits like this?
  - Using procedural blocks allows the usage of procedural statements (or behavioral statements), this approach is called behavioural modeling.
  - Procedural statements allow the designers to express a circuit in a **higher-level language/abstraction**

These statements are;

- If-else-else if blocks
- Case statement
- Loops (not all are sytnhesizable)

This week

# Behavioral Modeling – if/else blocks

```
if( /*conditonal_expression*/ )
begin
    /*statements*/
end

else if(/*conditonal_expression*/)
begin
    /*statements*/
end

//  ...

else
begin
    /*statements*/
end
```

- Used to make a decision whether the statements within the block should be excecuted or not
- if conditional expression is true, evaluate all statements within particular block
- If false ('0','x','z'), particular block won't be evaluated
- If there is an 'optional' **else** statement, it will be evaluated in case of conditional expression being false

Can only be used within a procedural block ('always' or 'initial')

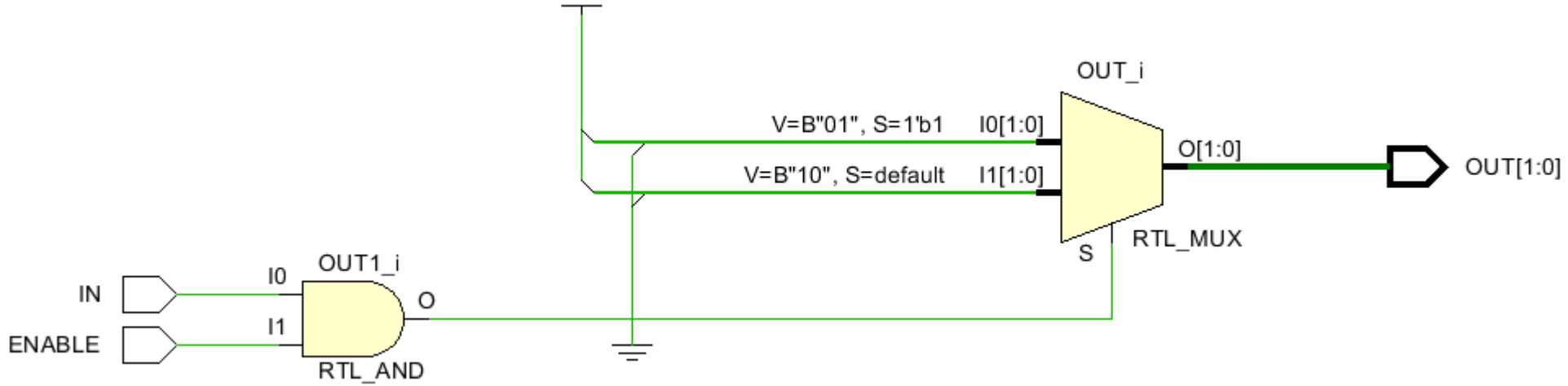
For non-procedural coding styles, we use **?** **:** operator

## ❑ Example 1: If-else block

```
always @ (*)
begin

    if(IN & ENABLE)
    begin
        OUT = 2'b01;
    end
    else
        OUT = 'b10;    // If only one statement is written,
                       // begin-end encapsulation might be discarded
end
```

```
assign OUT = (ENABLE & IN) ? 2'b01 : 2'b10;
    //        condition    if_true     if_false
```

Non-procedural equivalent




**An if-else pair corresponds to a MUX structure**

# Behavioral Modeling – if/else blocks

❑ **Example 2**

| I3 | I2 | I1 | I0 | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | I1 and I0 |
| 0 | 0 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 0 | I1 or I0 |
| 0 | 1 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 0 | I1 exor I0 |
| 1 | 0 | 0 | 1 | 1 | |
| 1 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 0 | |
| 1 | 1 | x | x | 1 | |

❑ Let inputs I3 and I2 be used as conditions:

- When they are both zero, we see that the function works as I0 and I1
- When I3=0 and I2=1, we see that the function works as I0 or I1
- When I3=1 and I2=0, we see that the function works as I0 xor I1
- Otherwise, F = 1.

# Behavioral Modeling – Chained if/else if blocks

- Example 2: If/else-if chain

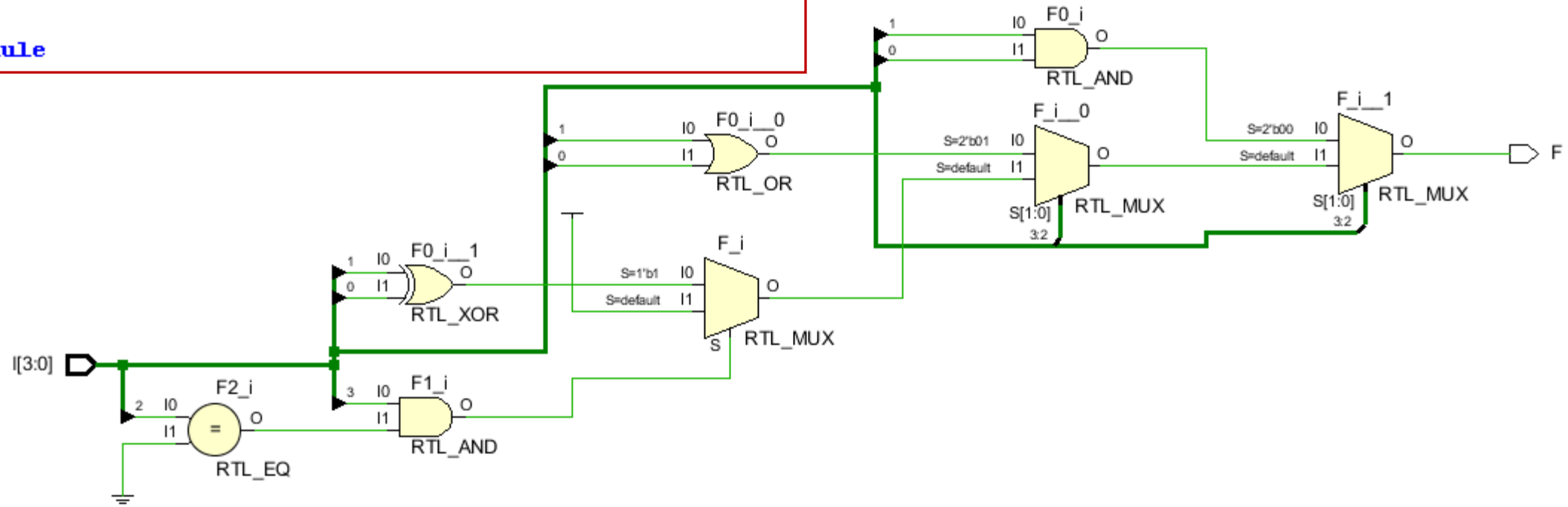
```
module function_F(I,F);
        input [3:0] I;
        output reg F;

    always @ (I)
    begin
        if( {I[3],I[2]} == 2'd0 )
            F = I[1] & I[0];
        else if( {I[3],I[2]} == 2'd1 )
            F = I[1] | I[0];
        else if( I[3]==1 && I[2]==1'b0 ) // Alternative!
            F = I[1] ^ I[0];
        else
            F = 1'b1;
    end

endmodule
```

Equivalent Behavioral Code

# ❑ Example 2: If/else-if chain

```
module function_F(I,F);
        input [3:0] I;
        output reg F;

    always @ (I)
    begin
        if( {I[3],I[2]} == 2'd0 )
            F = I[1] & I[0];
        else if( {I[3],I[2]} == 2'd1 )
            F = I[1] | I[0];
        else if( I[3]==1 && I[2]==1'b0 ) // Alternative!
            F = I[1] ^ I[0];
        else
            F = 1'b1;
    end

endmodule
```

if/else-if chains correspond to priority mux structure



```
`timescale 1ns / 1ps
module conditional_test;
wire O;
reg [3:0] I;

initial
    $monitor($time, "I=%b: O=%b",I,O);


conditional_if c1(.I(I),.F(O));

initial
begin

    I=4'b0000;  #20;
    I=4'b0001;  #20;
    I=4'b0010;  #20;
    I=4'b0011;  #20;
    I=4'b0100;  #20;
    I=4'b0101;  #20;
    I=4'b0110;  #20;
    I=4'b0111;  #20;
    I=4'b1000;  #20;
    I=4'b1001;  #20;
    I=4'b1010;  #20;
    I=4'b1011;  #20;
    I=4'b1100;  #20;
    I=4'b1101;  #20;
    I=4'b1110;  #20;
    I=4'b1111;  #20;
end

endmodule
```

```
# run 1000ns
                   0I=0000: O=0
                  20I=0001: O=0
                  40I=0010: O=0
                  60I=0011: O=1
                  80I=0100: O=0
                 100I=0101: O=1
                 120I=0110: O=1
                 140I=0111: O=1
                 160I=1000: O=0
                 180I=1001: O=1
                 200I=1010: O=1
                 220I=1011: O=0
                 240I=1100: O=1
                 260I=1101: O=1
                 280I=1110: O=1
                 300I=1111: O=1
```

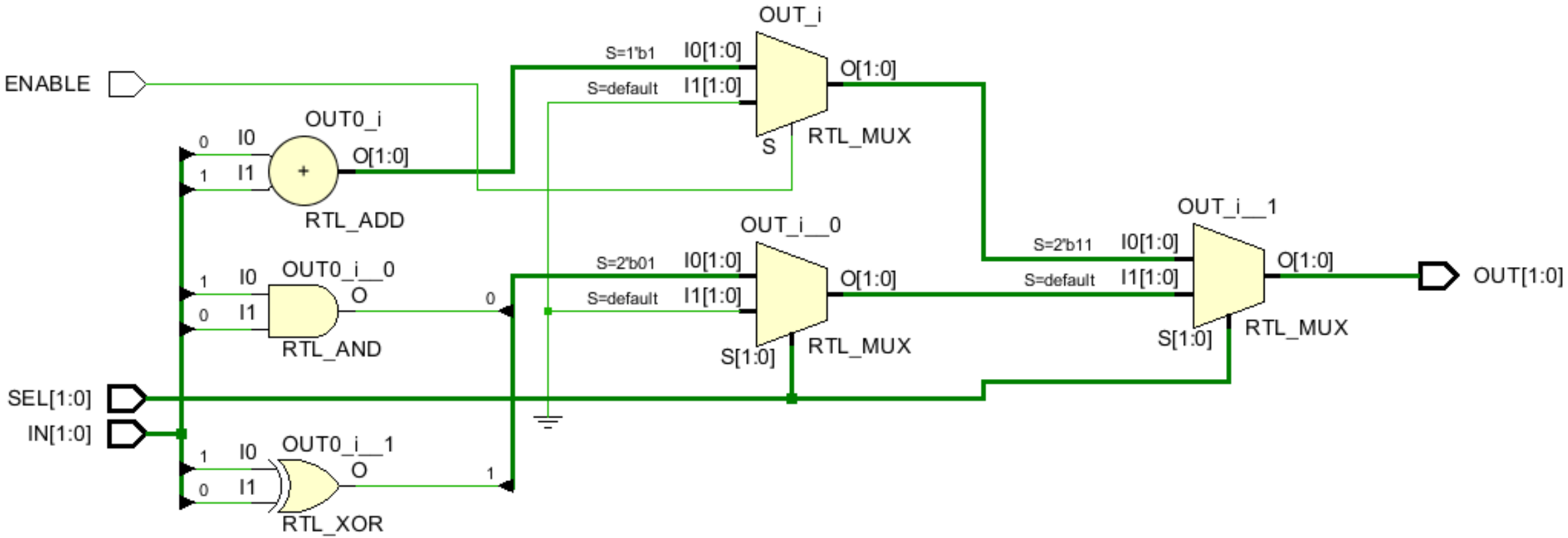
| I3 | I2 | I1 | I0 | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | I1 and I0 |
| 0 | 0 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 0 | I1 or I0 |
| 0 | 1 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 0 | I1 exor I0 |
| 1 | 0 | 0 | 1 | 1 | |
| 1 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 0 | |
| 1 | 1 | x | x | 1 | |

## ❑ Example 3: Nested if/else

```
module example1
(
    input [1:0] SEL,
    input [1:0] IN,
    input ENABLE,
    output reg [1:0] OUT
);

    always @ (*)
    begin

        if(SEL==2'b11)
        begin
            if(ENABLE) OUT = IN[0] + IN[1]; //Nested if-else
            else OUT = 0;
        end
        else if(SEL==2'b01)
        begin
            OUT[0]= IN[1] & IN[0];
            OUT[1]= IN[1] ^ IN[0];
        end
        else
            OUT = 0;


    end

endmodule
```

# Behavioral Modeling – Nested if/else blocks

## ❑ Example 3: Nested if/else




Resulting RTL

## ❑ Example 4: If without else

❑While coding combinational logic, if statements without elses will produce latches. This should be avoided.

❑Latches are mostly unintentional and caused by incomplete conditional assignments (forgetting to define outputs for certain possible input cases)

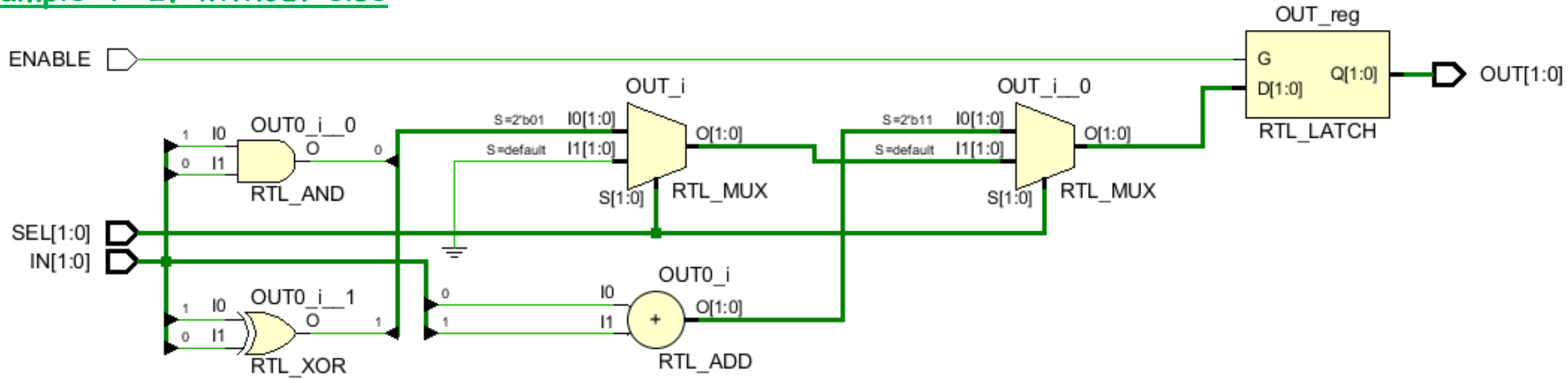
❑Latches cause timing and routing complications in FPGAs. Design tools usually throw a warning if they detect an inferred latch.

❑Note that latches only appear while coding combinational logic with procedural statements.

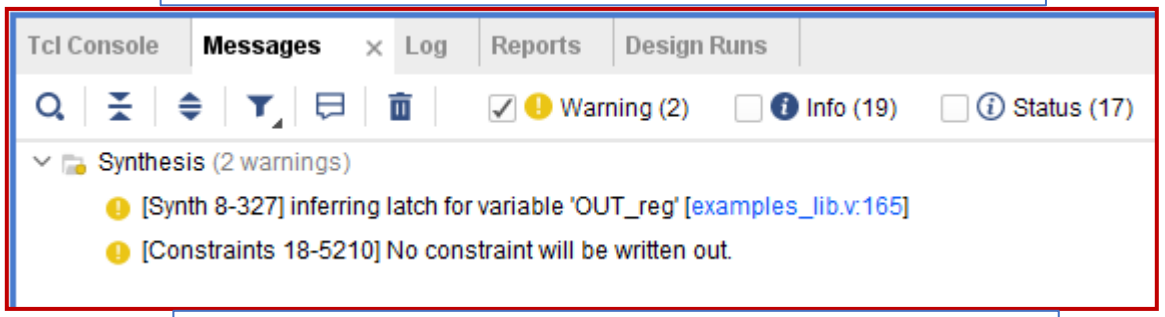
```
module example2
(
    input [1:0] SEL,
    input [1:0] IN,
    input ENABLE,
    output reg [1:0] OUT
);

    always @ (*)
    begin

        if(ENABLE)
        begin

          if(SEL==2'b11)
          begin
             OUT = IN[0] + IN[1];
          end

          else if(SEL==2'b01)
          begin
              OUT[0]= IN[1] & IN[0];
              OUT[1]= IN[1] ^ IN[0];
          end

          else
              OUT = 0;
        end

    end


endmodule
```

# Behavioral Modeling – if without else

## Example 4: If without else




Resulting RTL schematic




Synthesis tool threw a warning!

# Behavioral Modeling – Case statement

```
case(/*ExpressionToBeChecked*/)

    /*value1>*/:
    begin
        /*statements*/
    end

    /*value1>*/:
    begin
        /*statements*/
    end

    //   ...


    default:
    begin
        /*statements*/
    end

endcase
```

- When input combinations are various and many in number, coding with **case blocks** will be easier than coding with if-else blocks.
- "default" statement is optional, and corresponds to the case where none of the listed items match the check expression value
- The value of the check expression and each case item will be compared one by one, in the exact order specified, until a match is found. If there is no match, default expression will be executed.
- Check expressions and case items can be computed at runtime. They do not need to be a constant.

# Behavioral Modeling – Case statement

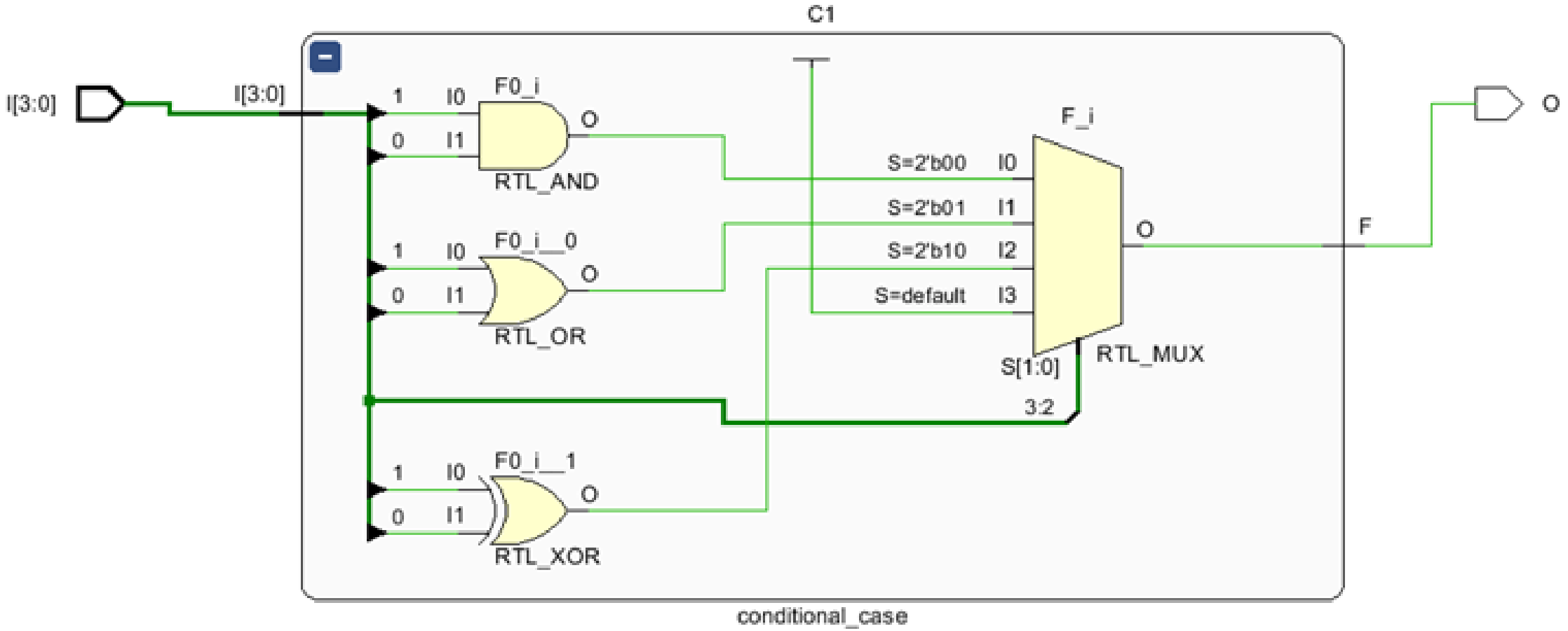
- **Example 5: Example 2 reworked with case statement**

| I3 | I2 | I1 | I0 | F | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | I1 and I0 |
| 0 | 0 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 0 | 0 | |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 0 | I1 or I0 |
| 0 | 1 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 0 | I1 exor I0 |
| 1 | 0 | 0 | 1 | 1 | |
| 1 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 0 | |
| 1 | 1 | x | x | 1 | |

```
module conditional_case(I,F);
input [3:0] I;
output reg F;

always @(I)
    case({I[3],I[2]})
        2'd0:F=I[1]&I[0];
        2'd1:F=I[1]|I[0];
        2'd2:F=I[1]^I[0];
        default:F=1'b1;
    endcase

endmodule
```

# Behavioral Modeling – Case statement

- **Example 5: Example 2 reworked with case statement**




Resulting RTL

# Behavioral Modeling – Case statement

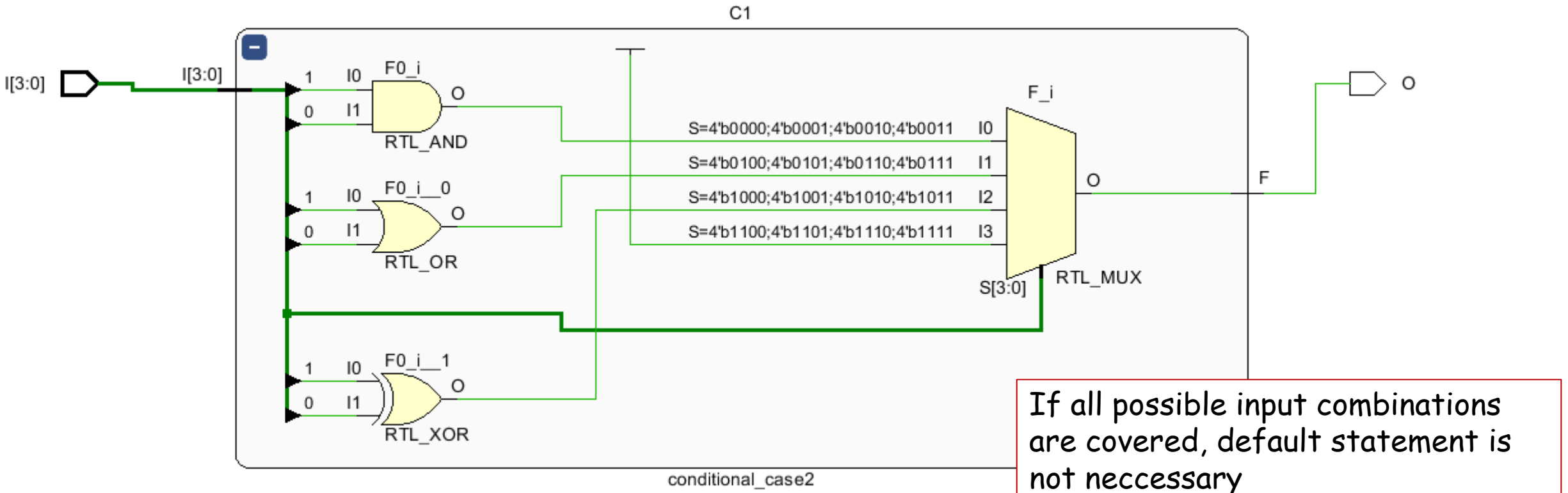
❑ **Example 6: Common cases**

```
module conditional_case2(I,F);
input [3:0] I;
output reg F;

always @(I)
    case(I)
        4'b0000,4'b0001,4'b0010,4'b0011:F=I[1]&I[0];
        4'b0100,4'b0101,4'b0110,4'b0111:F=I[1]|I[0];
        4'b1000,4'b1001,4'b1010,4'b1011:F=I[1]^I[0];
        4'b1100,4'b1101,4'b1110,4'b1111:F=1'b1;
    endcase
endmodule
```

❑ Case items which correspond to the same expression can be combined using commas

# Behavioral Modeling – Case statement

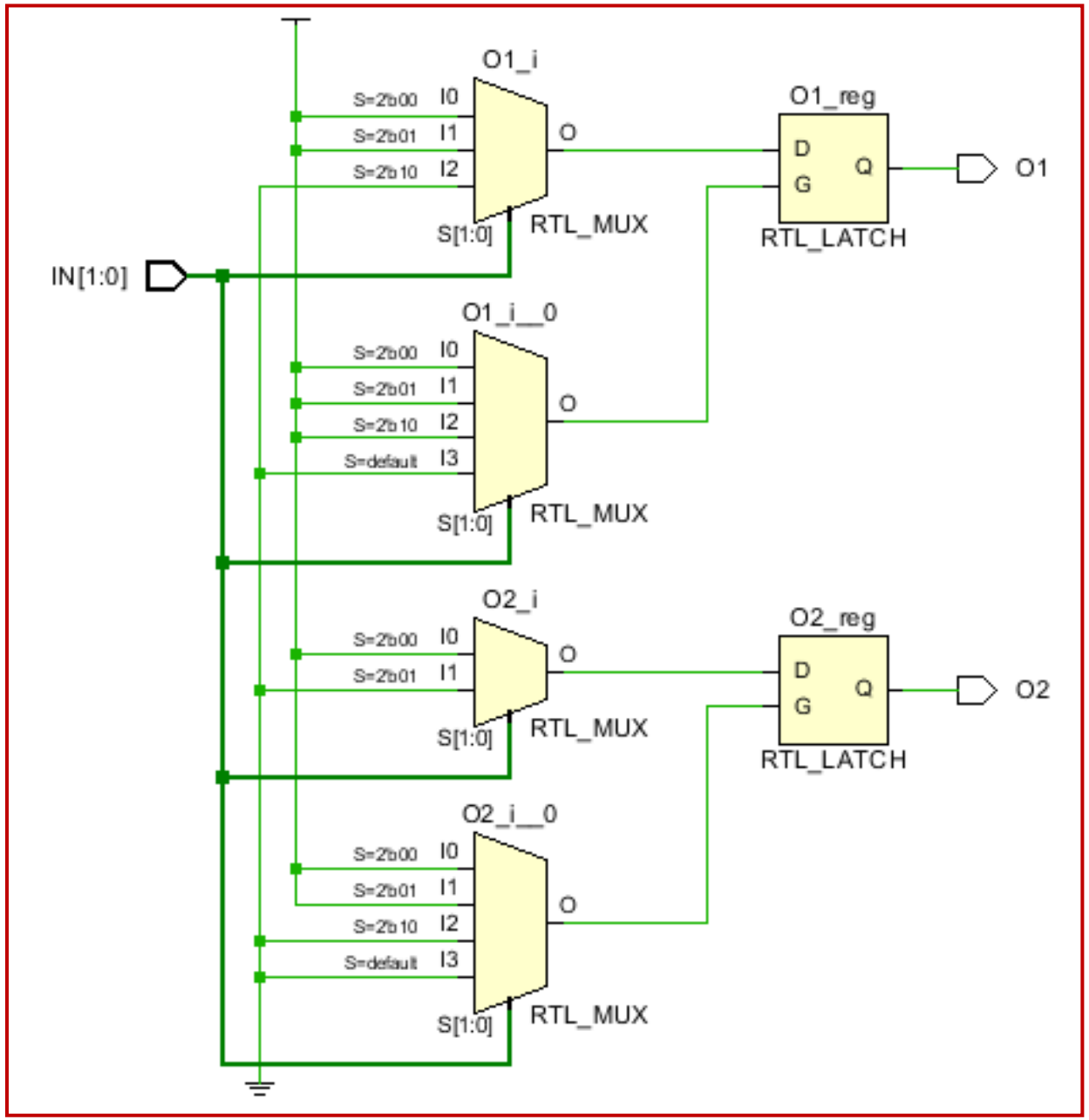
## ❑ Example 6: Common cases




If all possible input combinations are covered, default statement is not neccessary

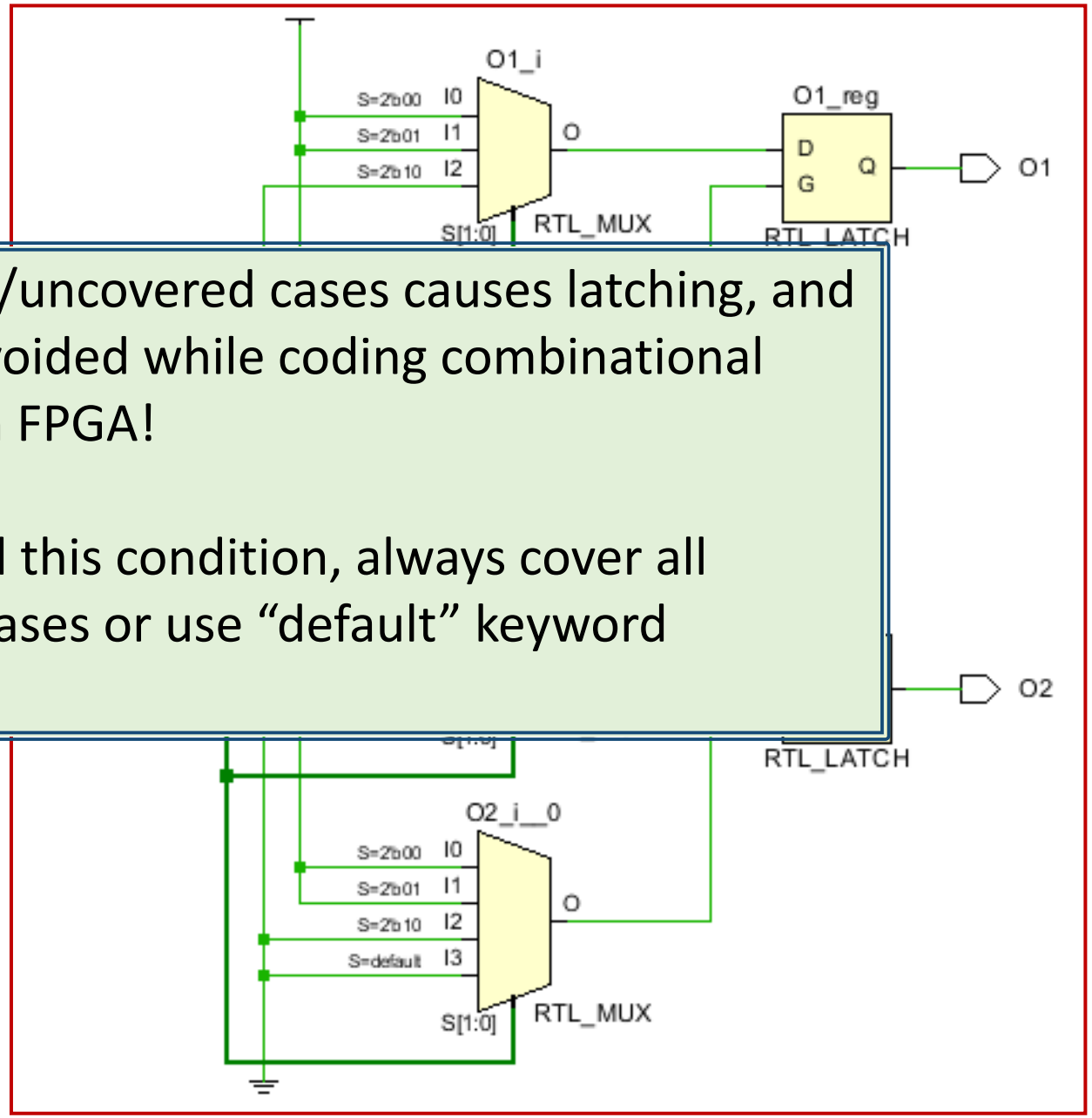
Resulting RTL

## ❑ Example 7: Incomplete case block




```
module example5
(
    input [1:0] IN,
    output reg O1,O2
);

    always @ (*)
    begin

        case(IN)
            0:
            begin
                O1 = 1;
                O2 = 1;
            end
            1:
            begin
                O1 = 1; O2 = 0;
            end
            2:
            begin
                O1 = 0;
            end

            // Missing case 3:
            // Missing default

        endcase
    end

endmodule
```

## ❑ Example 7: Incomplete case block

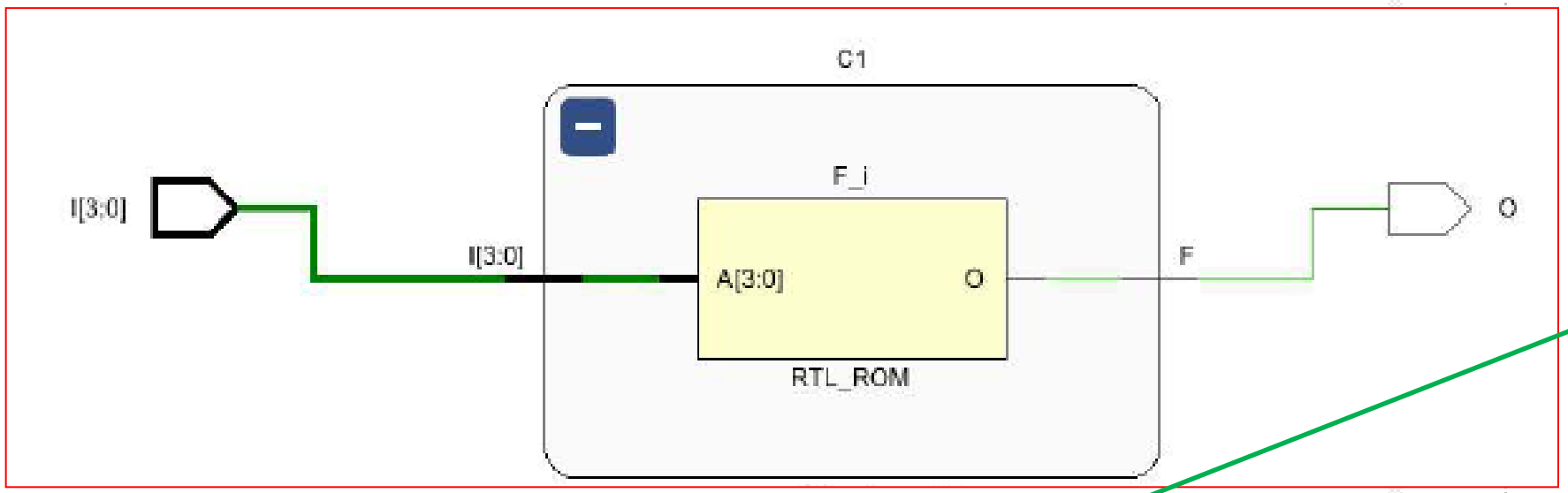



❑Missing/uncovered cases causes latching, and best be avoided while coding combinational circuits on FPGA!

❑To avoid this condition, always cover all possible cases or use “default” keyword

```
module example5
(
    input [1:0] IN,
    output reg O1,O2
);

    always @ (*)
    begin

        case(IN)
            0:
            begin
                O1 = 1;
                O2 = 1;
            end
            1:
            begin
                O1 = 1; O2 = 0;
            end
            2:
            begin
                O1 = 0;
            end

            // Missing case 3:
            // Missing default

        endcase
    end

endmodule
```

## ❑ Example 8: Fully pre-defined case statement




```
module conditional_case2(I,F);
input [3:0] I;
output reg F;

always @(I)
    case(I)
        4'b0000:F=I[1]&I[0];
        4'b0001:F=I[1]&I[0];
        4'b0010:F=I[1]&I[0];
        4'b0011:F=I[1]&I[0];
        4'b0100:F=I[1]|I[0];
        4'b0101:F=I[1]|I[0];
        4'b0110:F=I[1]|I[0];
        4'b0111:F=I[1]|I[0];
        4'b1000:F=I[1]^I[0];
        4'b1001:F=I[1]^I[0];
        4'b1010:F=I[1]^I[0];
        4'b1011:F=I[1]^I[0];
        4'b1100:F=1'b1;
        4'b1101:F=1'b1;
        4'b1110:F=1'b1;
        4'b1111:F=1'b1;
    endcase
endmodule
```

How come these expressions are inferred as constants??

# Behavioral Modeling – "x" and "z" in case statements

```
case (select[1:2])
    2'b00: result = 0;
    2'b01: result = flaga;
    2'b0x,
    2'b0z: result = flaga ? 'bx : 0;
    2'b10: result = flagb;
    2'bx0,
    2'bz0: result = flagb ? 'bx : 0;
    default result = 'bx;
endcase
```

- Expression in case paranthesis and case item expressions are compared bit by bit; thus, case statement gives definitve results when "x" and "z" values are involved in either of these expressions

- For example, consider the values below for this code;

  Select[1] == 0,
  flaga == 0,

  and when select[2] is "x" or "z", then **the result will be 0**, which is resolved at third case

# Behavioral Modeling – "casex" and "casez"

- There are two special types of case statement to handle don't care situations: "casex" and "casez"

- "casez" treats the value "z" as "don't care"
- "casex" treats both "x" and "z" as "don't care"

- Using "?" character instead of "z" is also allowed for a more convenient don't care specification
- The syntax is the same as traditional case statement, except they begin with casex and casez keywords, respectivelly.

# Behavioral Modeling – "casex" and "casez"

## Example 9: casez example

```
module example6
(
    input [2:0] in,
    output reg out1,out2,out0
);
    always @(in)
    begin
        {out2,out1,out0} = 3'b000;
        casez (in)
            3'b1?? : out2 = 1'b1;
            3'b?1? : out1 = 1'b1;
            3'b??1 : out0 = 1'b1;
            default: {out2,out1,out0} = 3'b000;
        endcase
    end
endmodule
```

- These are "wildcard" style of coding case statements, and prone to mistakes. Take extra care while using these

- **For example, if in[0] is left unconnected (value Z), then the first case branch will still be true**

# Behavioral Modeling – "casex" and "casez"

## Example 10: casex example

```
module casex_example(
    input [7:0]sig,
    output reg [7:0]result
);

    reg [7:0] mask;

    always@(*)
    begin
        mask = 8'bx0x0x0x0;
        casex (sig ^ mask)
            8'b00110Oxx: result=0;
            8'b1100xxOO: result=1;
            8'bOOxxOO11: result=2;
            8'bxxO10100: result=3;
            default: result='bz;
        endcase
    end

endmodule
```

- Let's consider the case when "sig == 8'b0110_0110"

"sig ^ mask" expression would be

0110_0110

x0x0_x0x0

---

x1x0_x1x0

*In that case, second case item will be a match, result will be 1*

# Behavioral Modeling – "casex" and "casez"

## Example 11: Functions with don't care values using casex

```
module case_dontcare
(
    input [2:0] in,
    output reg out
);
    always @(*)
    begin
        casex (in)
            3'b000: out=1;
            3'b001: out=1'bx;
            3'b010: out=1;
            3'b011: out=1;
            3'b100: out=0;
            3'b101: out=0;
            3'b110: out=0;
            3'b111: out=1'bx;
        endcase
    end
endmodule
```

| In[1]In[0] \ In[2] | 0 | 1 |
|---|---|---|
| 00 | 1 | 0 |
| 01 | x | 0 |
| 11 | 1 | x |
| 10 | 1 | 0 |

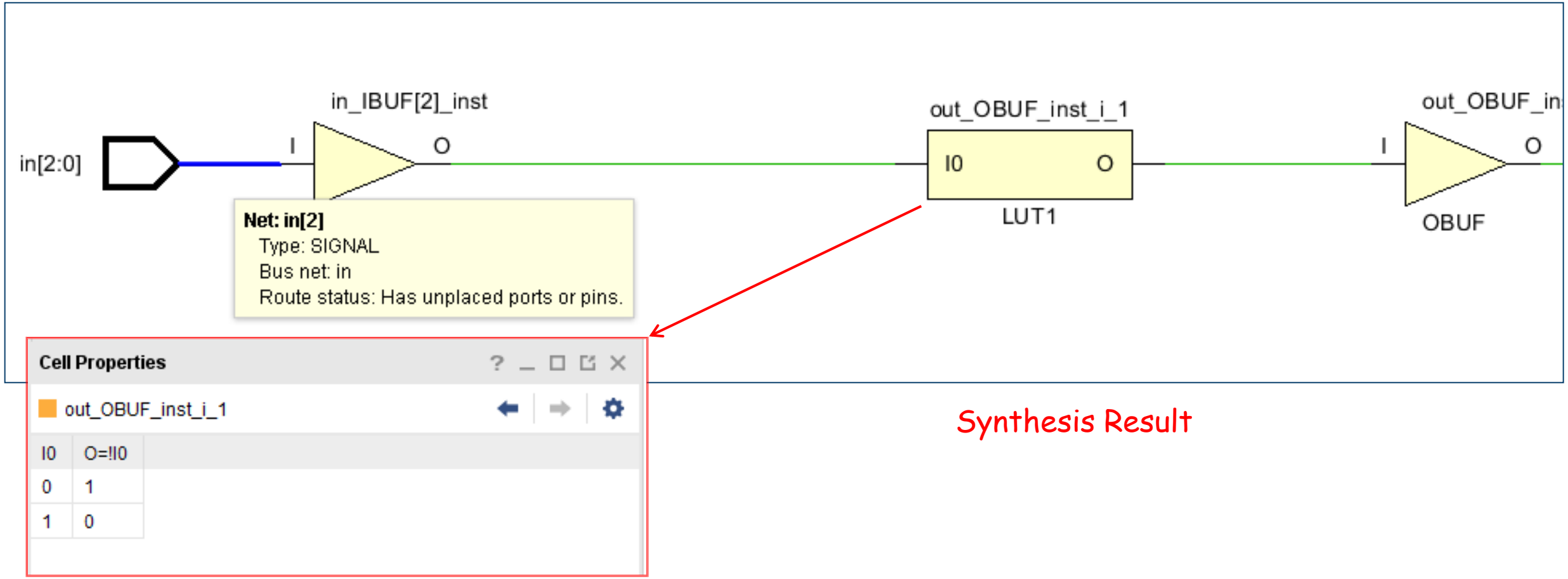
- We expect this to be simplified as:

  out = ~ in[2]

  ...according to Karnaugh Map simplification

# Behavioral Modeling – "casex" and "casez"

## Example 11: Functions with don't care values using casex




Synthesis Result

## ❑ Example 12: Comparing case behaviours

```
module conditional_case3(I,F);
input I;
output reg [1:0] F;

always @(I)
    case(I)
        1'b0:F=
        1'b1:F=
        1'bx:F=
        1'bz:F=
    endcase
endmodule
```

```
module conditional_casex(I,F);
input I;
output reg [1:0] F;

always @(I)
    casex(I)
        'b00;
        'b01;
        'b10;
        'b11;
```

```
`timescale 1ns / 1ps
module conditional_test2;
wire [1:0] O;
reg  I;

initial
    $monitor($time, "I=%b: O=%b",I,O);


conditional_casex c1(.I(I),.F(O));

initial
begin

    I=1'b0;  #20;
    I=1'b1;  #20;
    I=1'bx;  #20;
    I=1'bz;  #20;

end

endmodule
```

## ❑ Example 12: Comparing case behaviours

**What about "casez" ?**

```
module conditional_case3(I,F);
input I;
output reg [1:0] F;

always @(I)
    case(I)
        1'b0:F=2'b00;
        1'b1:F=2'b01;
        1'bx:F=2'b10;
        1'bz:F=2'b11;

    endcase
endmodule
```

```
module conditional_casex(I,F);
input I;
output reg [1:0] F;

always @(I)
    casex(I)
        1'b0:F=2'b00;
        1'b1:F=2'b01;
        1'bx:F=2'b10;
        1'bz:F=2'b11;

    endcase
endmodule
```

```
I=1'b0;   #20;
I=1'b1;   #20;
I=1'bx;   #20;
I=1'bz;   #20;
```

*Stimulus*

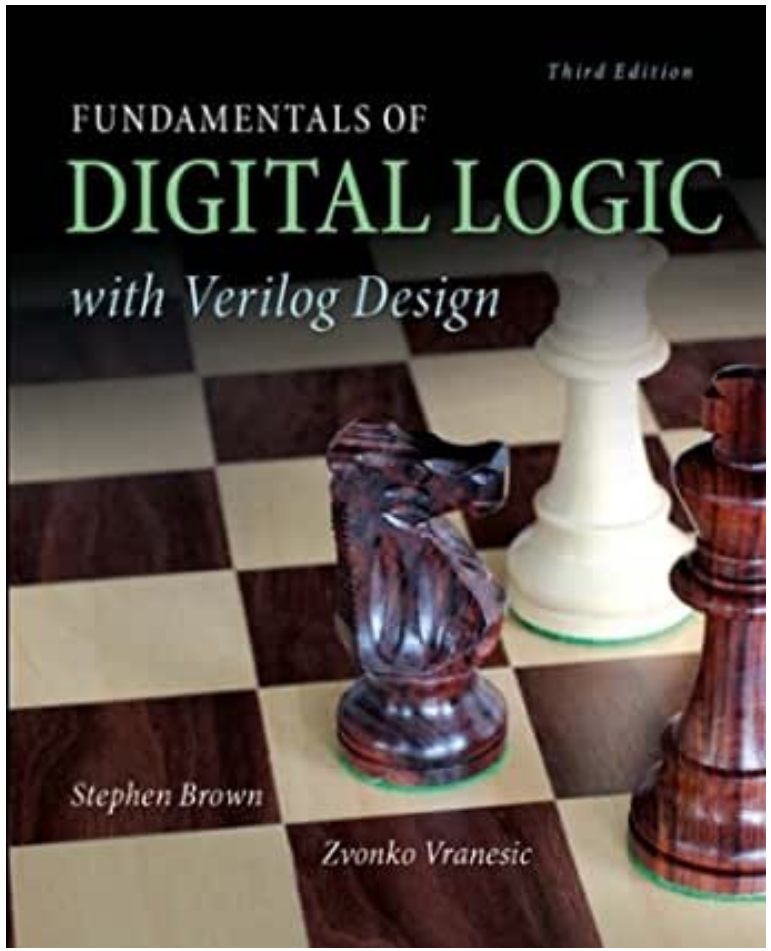
```
# run 1000ns
                    0I=0: O=00
                   20I=1: O=01
                   40I=x: O=10
                   60I=z: O=11
```

*"case" result*

```
# run 1000ns
                    0I=0: O=00
                   20I=1: O=01
                   40I=x: O=00
                   60I=z: O=00
```

*"casex" result*

# References




- IEEE Std **1364-2005**
- Brown & Vrasenic – "Fundamentals of Digital Design with Verilog"
- https://www.chipverify.com/verilog/verilog-tutorial
- http://www.asic-world.com/verilog/veritut.html

**Thank You!**