ISTANBUL TECHNICAL UNIVERSITY

Digital System Design & Applications

If/else & case

RES. ASST. FIRAT KULA & RES. ASST. BILGI GÖRKEM YAZGAÇ

if-else Statement

```
if(<conditional_expression1>)
    begin
        <procedural_statement1>;
    end
else if(<conditional_expression2>)
        <single_statement2>;
...

else
    begin
        <procedural_statement3>;
    end
```

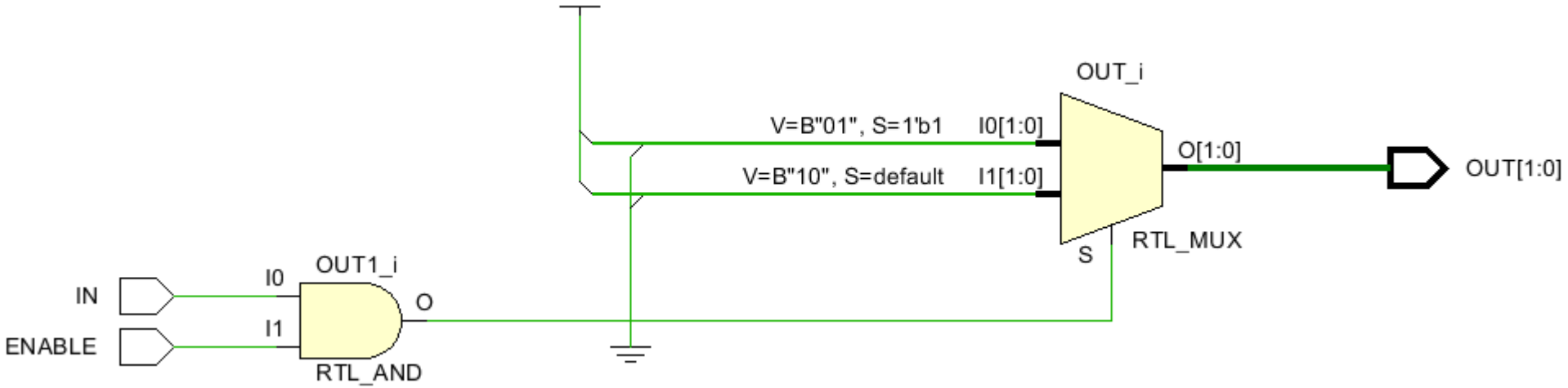
- Used to make a decision whether the statements within the block should be excecuted or not
- if conditional expression is true, excecute all statements within particular block
- İf false (‘0,’x’,’z’),particular block won’t be excecuted
- İf there is ‘optional’ **else** statement, it will be excecuted in case of false conditional expression
- İf-else structure corresponds to MUX

Can only be used within a procedural statement (always or initial blocks)
For non-procedural coding styles, we use **? :** operator

```
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

if-else 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	

- For given truth table
  - 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.

if-else 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 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 Statement

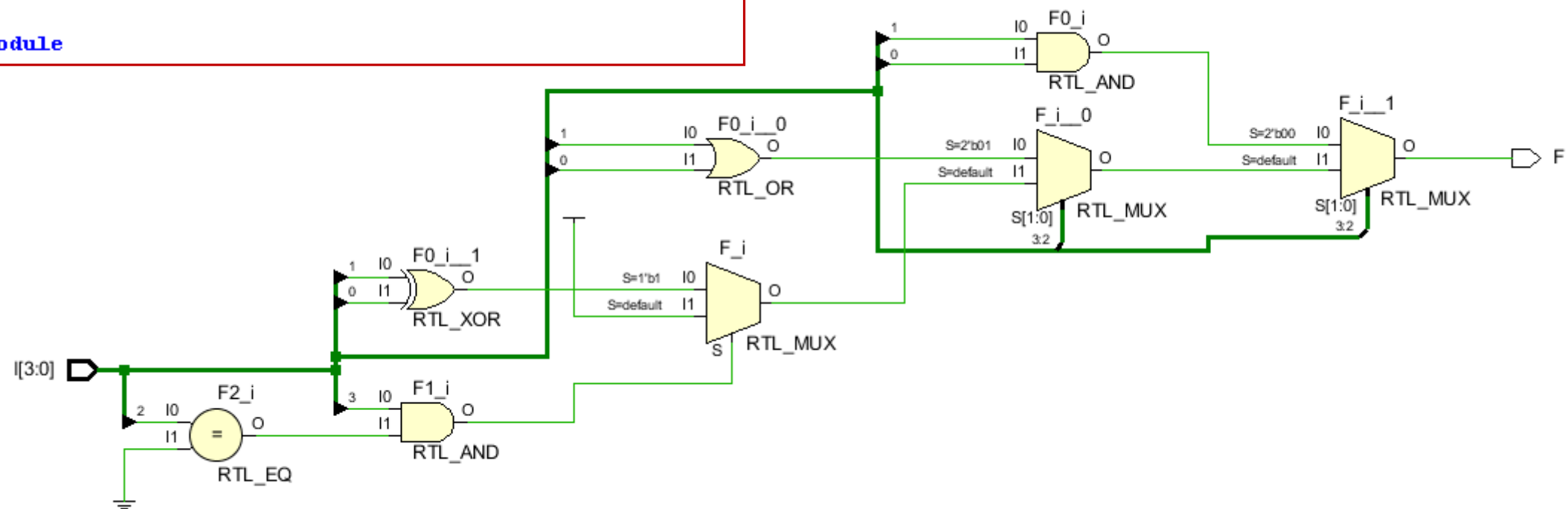
I3	I2	I1	I0	F	
0	0	0	0	0	I1 and I0
0	0	0	1	0	
0	0	1	0	0	
0	0	1	1	1	
0	1	0	0	0	I1 or I0
0	1	0	1	1	
0	1	1	0	1	
0	1	1	1	1	
1	0	0	0	0	I1 exor I0
1	0	0	1	1	
1	0	1	0	1	
1	0	1	1	0	
1	1	x	x	1	

```
`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
```

```
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
```

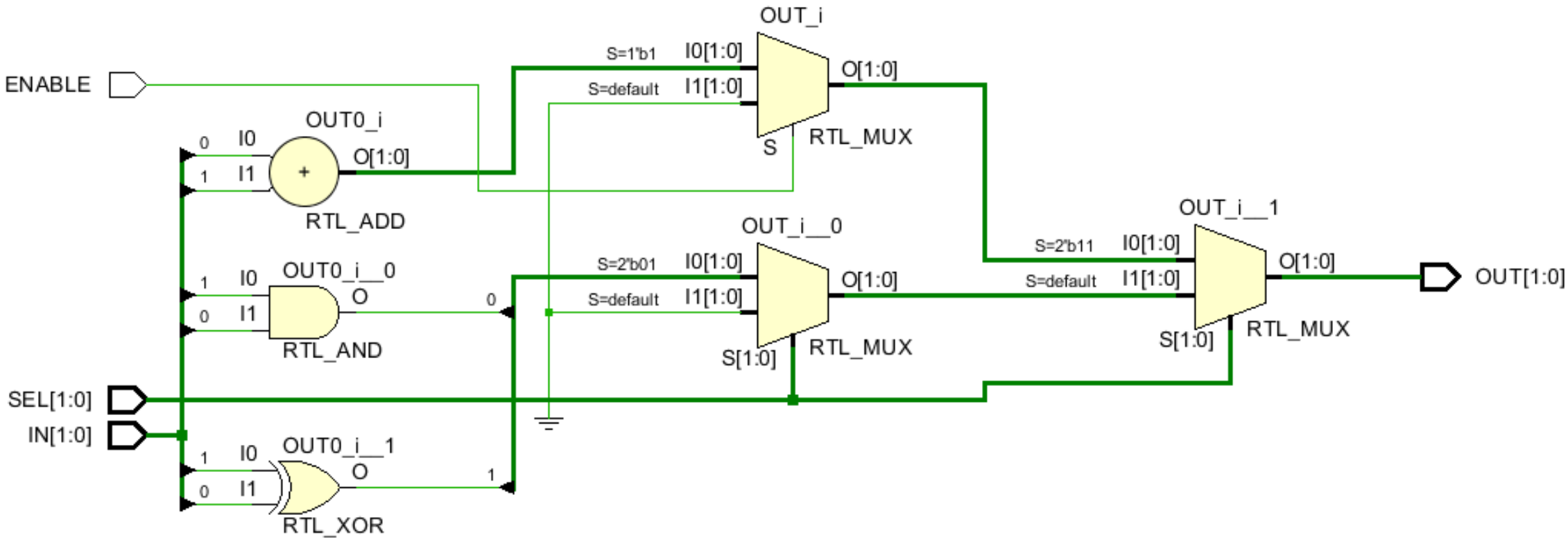
if-else if-else chains correspond to priority mux structure



Nested if-else Example

```
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
```

Nested if-else Example

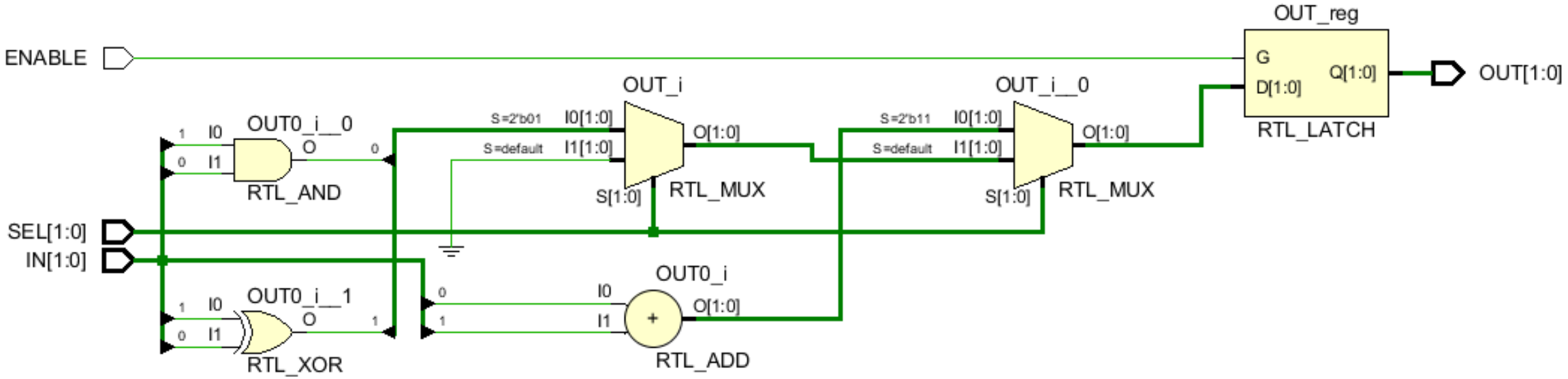


“if” without “else”

- 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
```

“if” without “else”



Resulting RTL schematic…

Tcl Console | **Messages** | Log | Reports | Design Runs

Warning (2) | Info (19) | Status (17)

Synthesis (2 warnings)

[Synth 8-327] inferring latch for variable 'OUT_reg' [examples_lib.v:165]

[Constraints 18-5210] No constraint will be written out.

Tool throwing a warning…

case Statement

```
case(<expression>)
        alternative1: <single_statement1>;
        alternative2:
            begin
                <procedural_statement2>;
            end
        ....
        default: <single_statement>

endcase
```

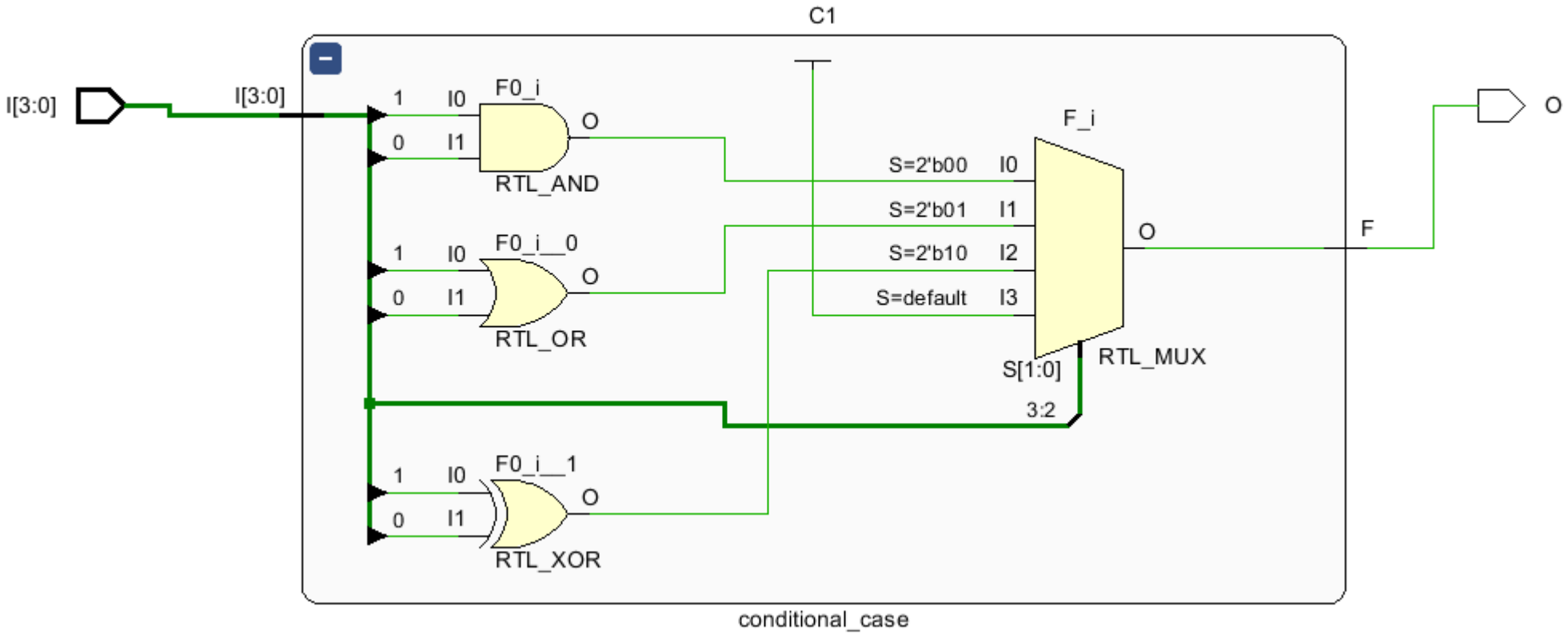
- When there are many possible alternatives case is more suitable for reading
- The value of the expression and each alternative are compared bit by bit
- When there are more possible variations than number of alternatives ‘optional’ **default** statement should be used.

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	

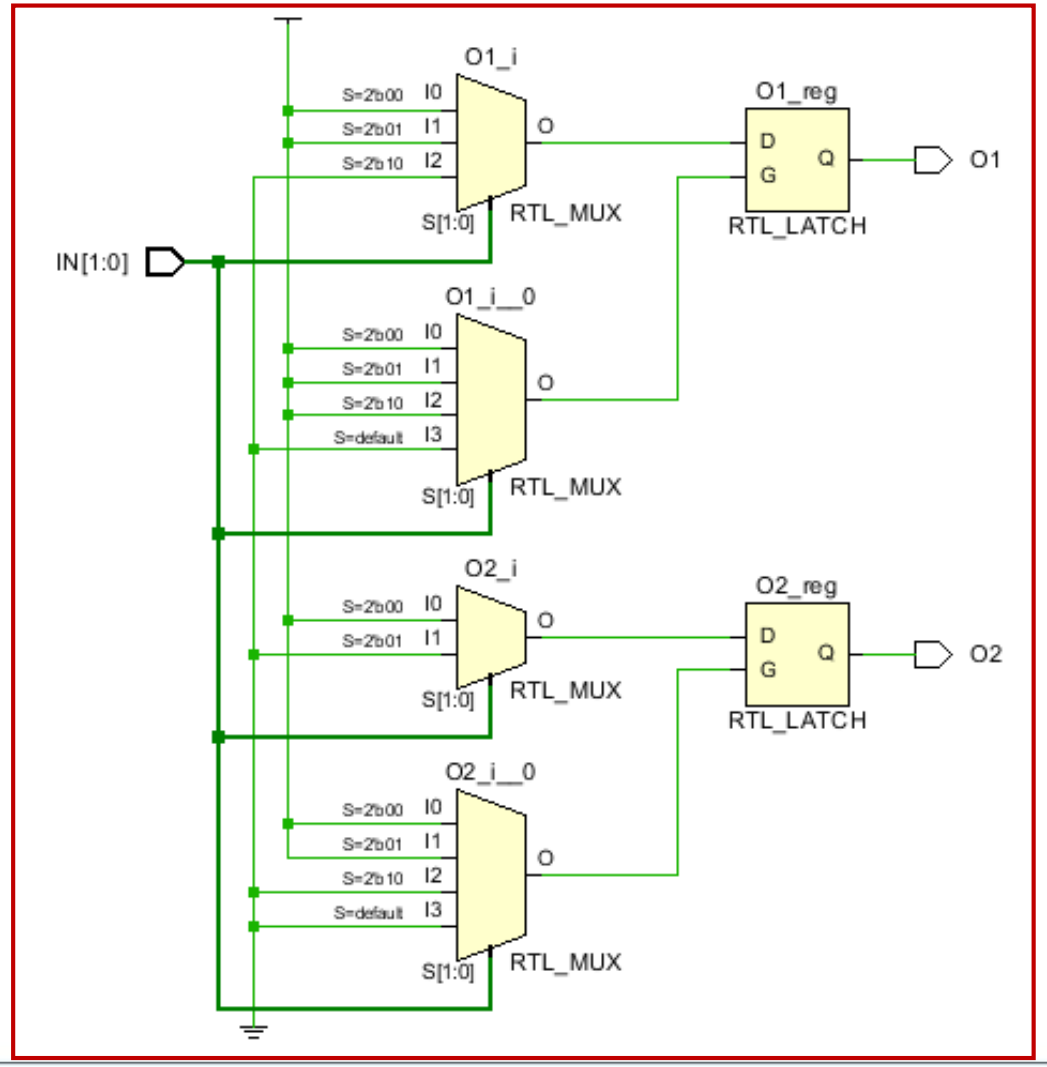
```
module conditional_case(I,F);
input [3:0] I;
output reg F;

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
```

case Statement



case Statement – Missing cases

```
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
```



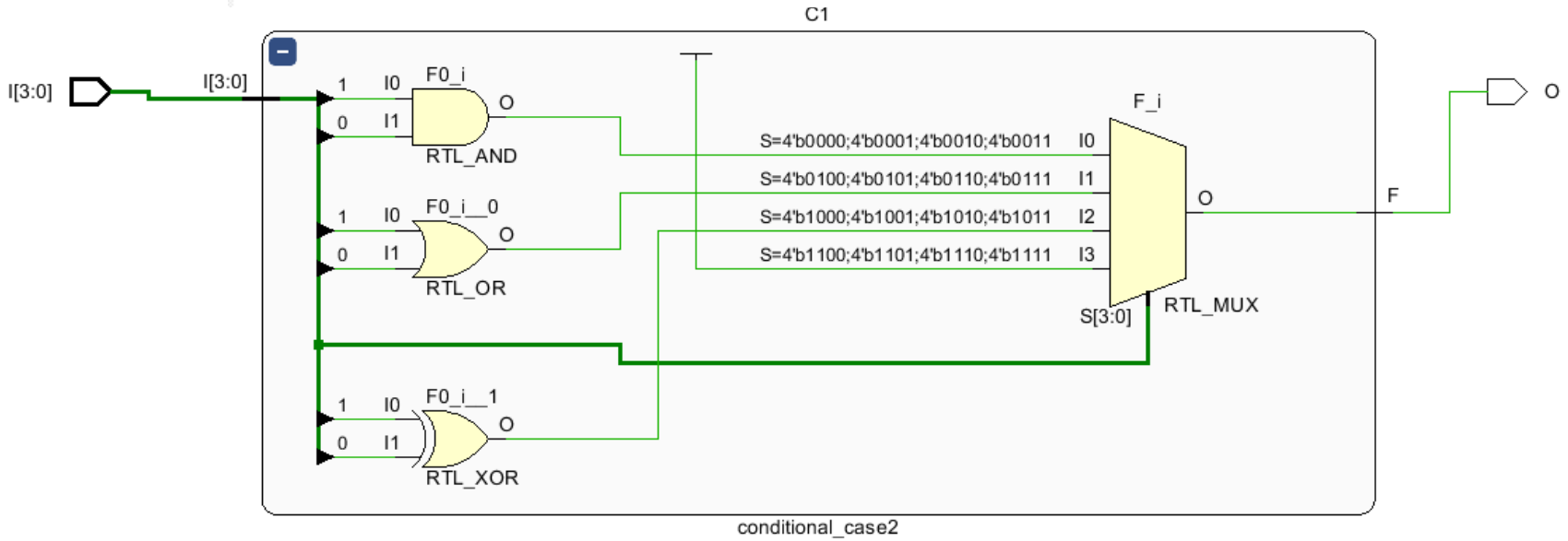
Missing/uncovered cases causes latching, and should be avoided while coding combinational circuits!

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

case Statement – Combined 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
```

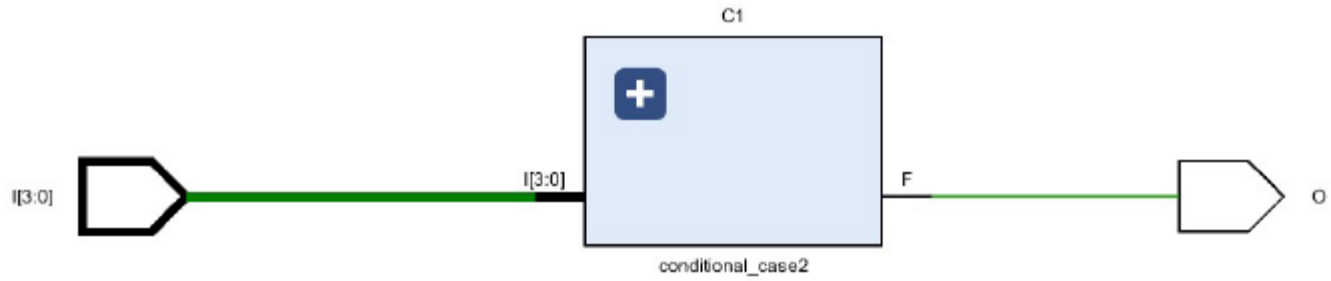
If all possible input combinations are covered, default statement is not neccessary



case Statement – case with full pre-defined values

- How would the schematic look like if the design is like below?

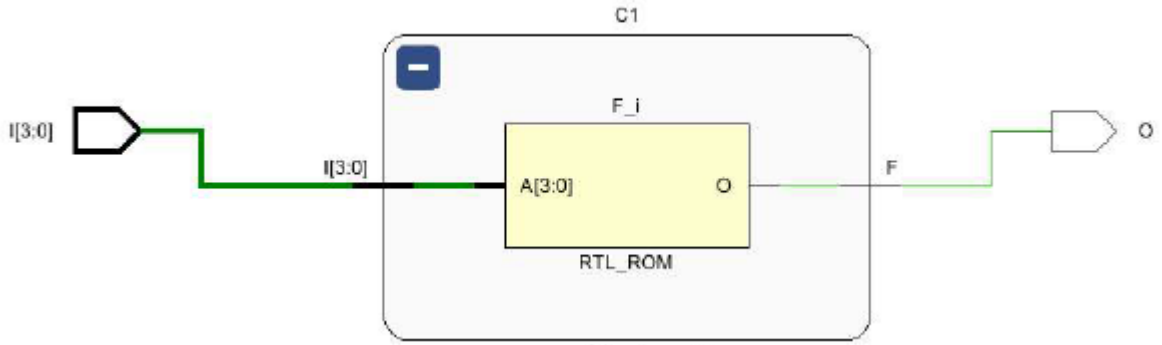
```
module conditional_case2(I,F);
input [3:0] I;
output reg F;

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
```



case Statement - case with full pre-defined values

- How would the schematic look like if the design is like below?

```
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
```



Case Statement – “casez” and “casex”

- casez:
  - Allows ‘z’ and ‘?’ in the alternatives of expression to be seen as don’t care values
- casex:
  - Allows ‘x’, ‘?’ and ‘z’ as don’t care characters
- If more than one match occurs, firstly listed one will be taken.
- They are “wildcard” styles of coding cases, but prone to bugs. For example if in[0] is left unconnected (value Z), then the first case branch will still be true

```
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
```

Case Statement

- Example run

```
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
```

```
`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
```

```
# 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

Thank you! Questions…

References

- Brown&Vrasenic, "Fundamentals of Digital Logic with Verilog Design", McGraw-Hill
- Use google ☺
  - http://www.asic-world.com/verilog
  - Etc.