ISTANBUL TECHNICAL UNIVERSITY

Digital System Design & Applications

Verilog – Sequential Circuits 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…)

Sequential Circuits?

button

When the button is pushed:
1) Turn on the light if it is off
2) Turn off the light if it is on

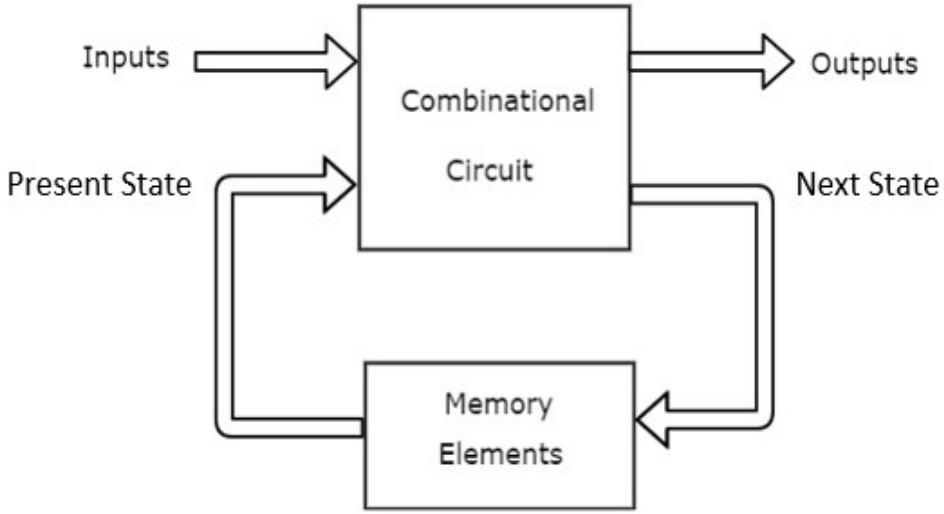
The light should change state within a second of the button press

light

- State
- The output was changed by an input " event" (pushing a button) rather than an input "value"

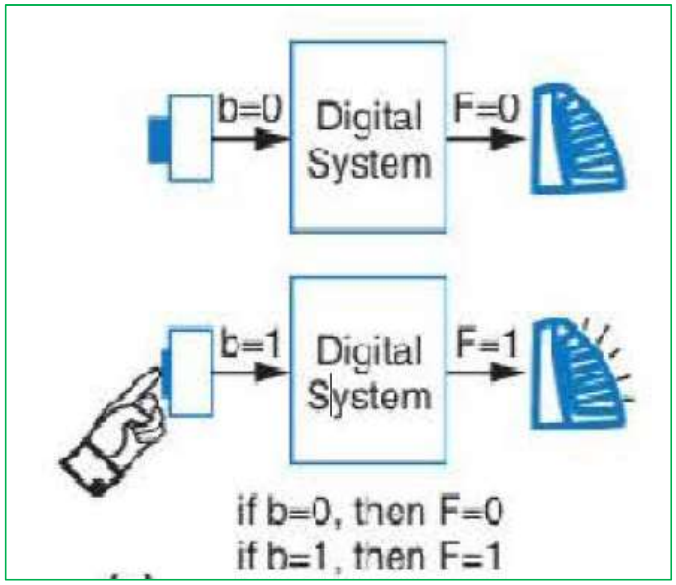
Sequential Circuits?

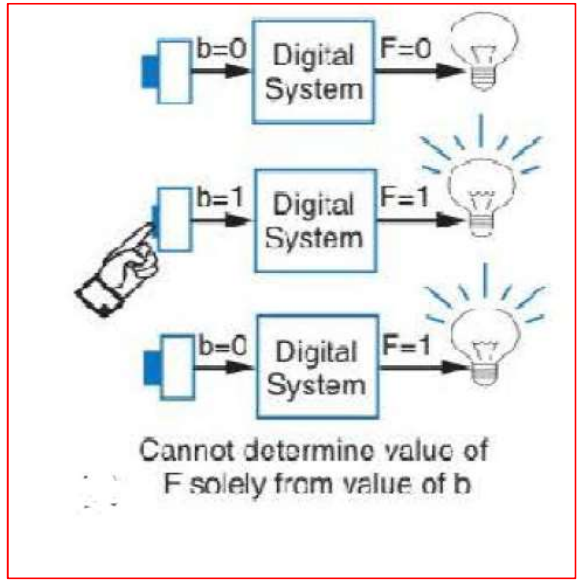
- A sequential circuit is an integration of combinational circuits and storage elements.
- The output depends on the present value of the input signal as well as the sequence of past inputs.



Sequential Circuits?

- A digital circuit whose output value depends solely on the present combination of the circuit inputs' values is called a combinational circuit. For example,doorbell system.
- On the other hand, in a sequantial circuit the output value depends on the present and past input values,such as the toggle lamp.





Combinational Sequential

Two simple examples

Sequential Circuits in Verilog

- **Always blocks**

  We've seen that always blocks are used when we want to allow conventional programming language structures (if,else,case etc.) to describe circuit behavior

  This is specifically useful when describing sequential circuits in Verilog

- **Recall:**

```
always @( /*Sensitivity List*/)
begin
    // statements
end
```

- While describing combinational circuits with always blocks, we included all affecting signals in sensitivity list, and used blocking assignments for coding the statements

Sequential Circuits in Verilog

- **Blocking (=) versus Non-blocking (<=) assignments**

Blocking

```
always @(*)
begin
    w1 = in1 & w2;
    w2 = in2 ^ in3;
    w3 = w1 | w2;
end
```

Non-blocking

```
always @(*)
begin
    w1 <= in1 & w2;
    w2 <= in2 ^ in3;
    w3 <= w1 | w2;
end
```

Sequential Circuits in Verilog

- **Blocking (=) versus Non-blocking (<=) assignments**

Blocking

- When the procedural block is triggered, statements are evaluated sequentially
- Executions happen at the end of the related statement
- Best practice for this assignment is to describe pure combinational logic within always blocks

Non-blocking

- When the procedural block is triggered, statements are scheduled
- Executions happen at the end of always block, parallelly.
- Best practice for this assignment is to describe sequential logic within always blocks

Sequential Circuits in Verilog

- **Blocking (=) versus Non-blocking (<=) assignments**

❖**Example 1: Single layer logic**

```
• always @ (a or b or c)
begin
        x = a | b;       %%1. Evaluate a | b, assign result to x
        y = a ^ b ^ c;  %%2. Evaluate a^b^c, assign result to y
        z = b & ~c;     %%3. Evaluate b&(~c), assign result to z
end
```

```
• always @ (a or b or c)
begin
        x <= a | b;       %%1. Evaluate a | b but defer assignment of x
        y <= a ^ b ^ c;  %%2. Evaluate a^b^c but defer assignment of y
        z <= b & ~c;      %%3. Evaluate b&(~c) but defer assignment of z
end                       %% 4. Assign x, y, and z with their new values
```

Since this is a single layer logic, two codes will behave the same. But what about multi layer logic?

Sequential Circuits in Verilog

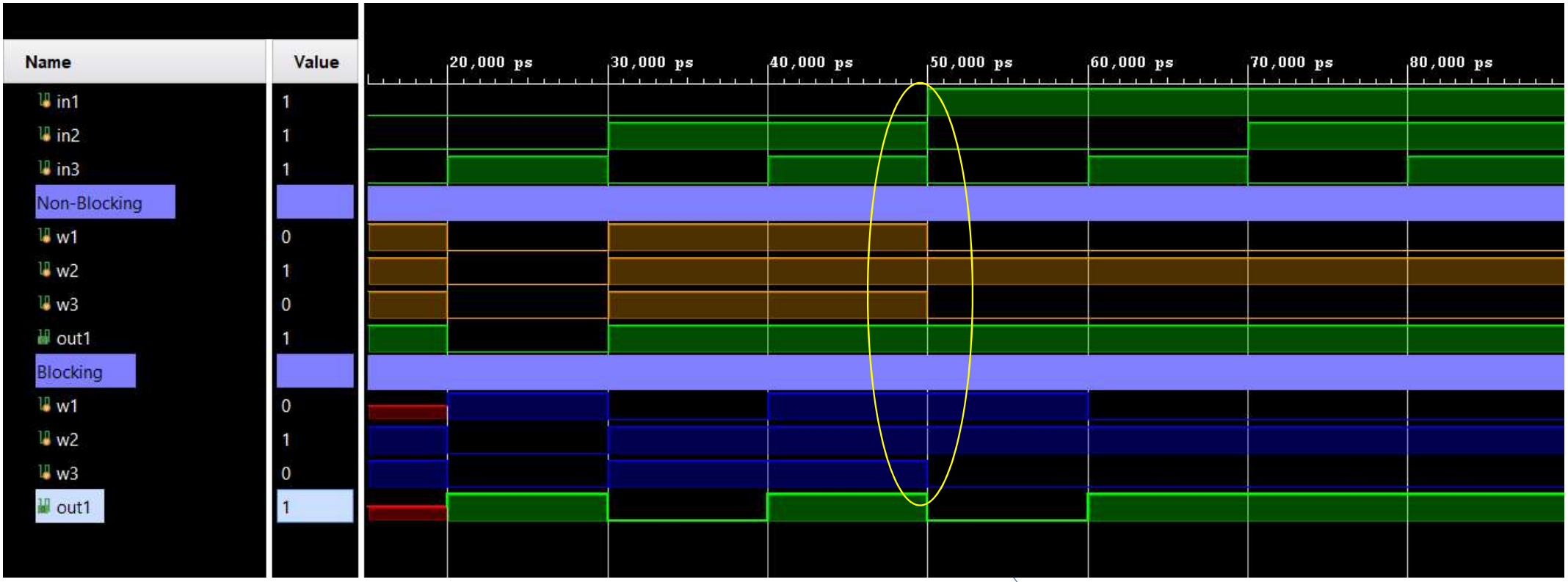
- **Blocking (=) versus Non-blocking (<=) assignments**

❖**Example 2: Multi layer logic**

```
module assignments_test1(
    input in1,in2,in3,
    output out1
);

    reg w1,w2,w3;

    always @(*)
    begin
        w1 = w3 & w2;
        w2 = in1 | in2 | ~in3;
        w3 = in1 ^ w2;
    end

    assign out1 = w1 + w2 + w3;

endmodule
```

```
module assignments_test2(
    input in1,in2,in3,
    output out1
);

    reg w1,w2,w3;

    always @(*)
    begin
        w1 <= w3 & w2;
        w2 <= in1 | in2 | ~in3;
        w3 <= in1 ^ w2;
    end

    assign out1 = w1 + w2 + w3;

endmodule
```

❖Example 2: Multi level logic

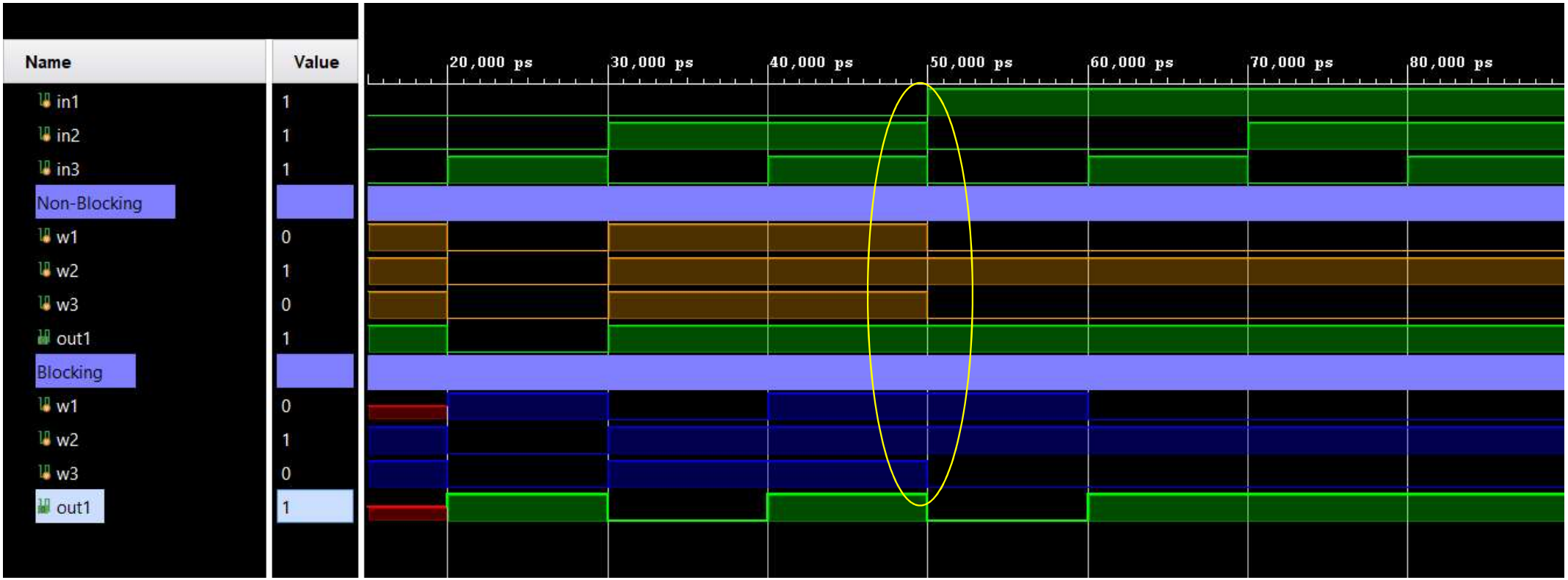


For both models
w values at the time of triggering event are:
W1 = 1, w2 = 1, w3=1;

And the value of inputs that triggered the event are
In1=1, in2=0, in3=0

Let's examine this transition…

❖Example 2: Multi level logic

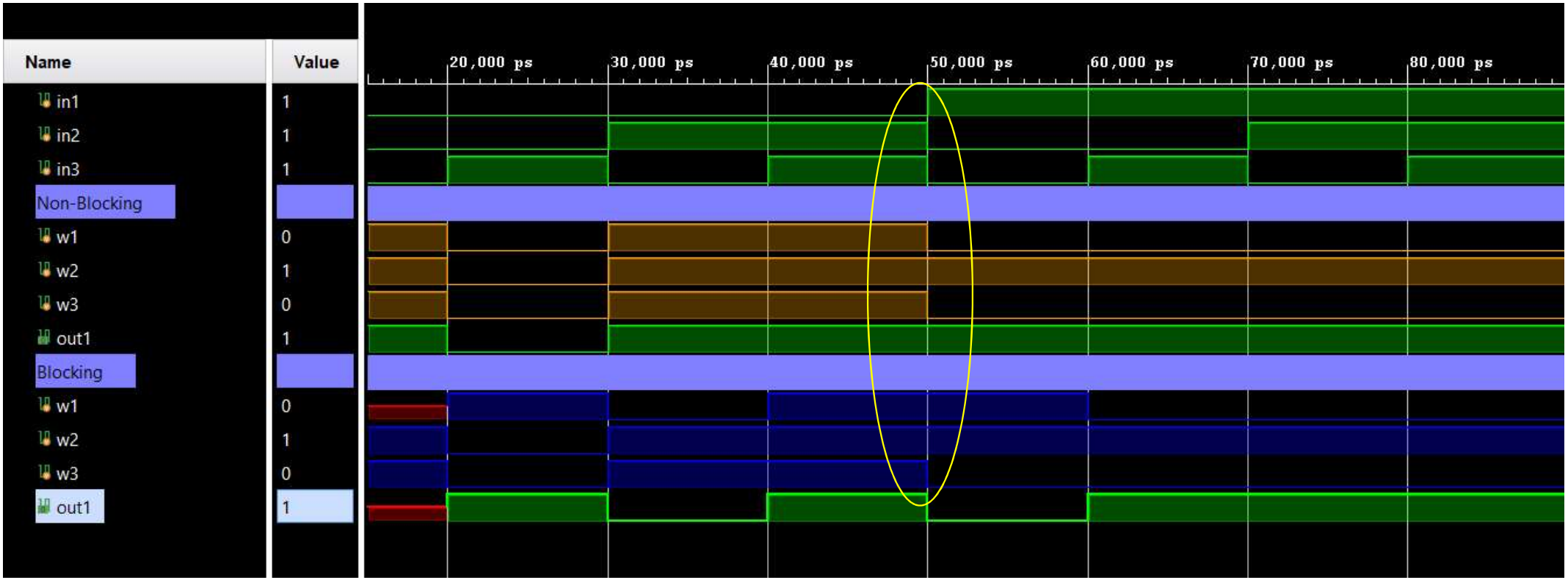


Initially: w1 = 1, w2 = 1, w3=1; In1=1, in2=0, in3=0

```
always @(*)
begin                                                    // Initially,   -- w1 = 1  w2=1   w3=1
    w1 = w3 & w2;            // w3&w2 = 1&1 = 1 is assigned to w1.       -- w1 = 1  w2=1   w3=1
    w2 = in1 | in2 | ~in3;   // in1 | in2 | ~in3 =  1  is assigned to w2. -- w1 = 1  w2=1   w3=1
    w3 = in1 ^ w2;           // in xor w2 = 1 xor 1 = 0 is assigned to w3. -- w1 = 1  w2=1   w3=0, out=0
end
```

```
assign out1 = w1 + w2 + w3;
```

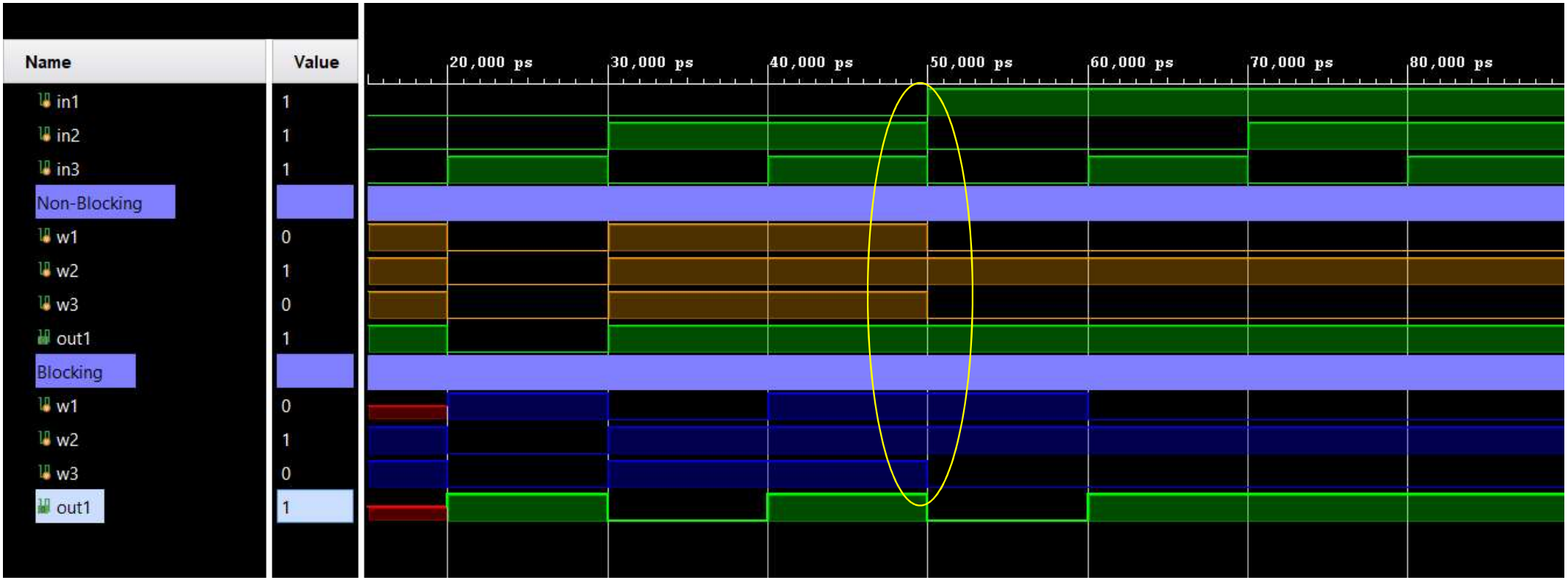
❖Example 2: Multi level logic



Initially: w1 = 1, w2 = 1, w3=1; In1=1, in2=0, in3=0

```
always @(*)
begin                          // Initially                                          --w1=1, w2=1, w3=1;  i1=1 i2=0 i3=0
    w1 <= w3 & w2;             //           w1 = w3&w2 = 1&1 = 1 is scheduled        --w1=1, w2=1, w3=1;
    w2 <= in1 | in2 | ~in3;    //          w2 = in1|in2|~in3 = 1 is scheduled        --w1=1, w2=1, w3=1;
    w3 <= in1 ^ w2;            //  w3 = in1 xor w2 = 1 xor 1 = 0 is scheduled        --w1=1, w2=1, w3=1;
end
                               // assignments happen when always block ends,  --w1=1, w2=1, w3=0;
                               // But change in w values triggers always block again at the same time!
```

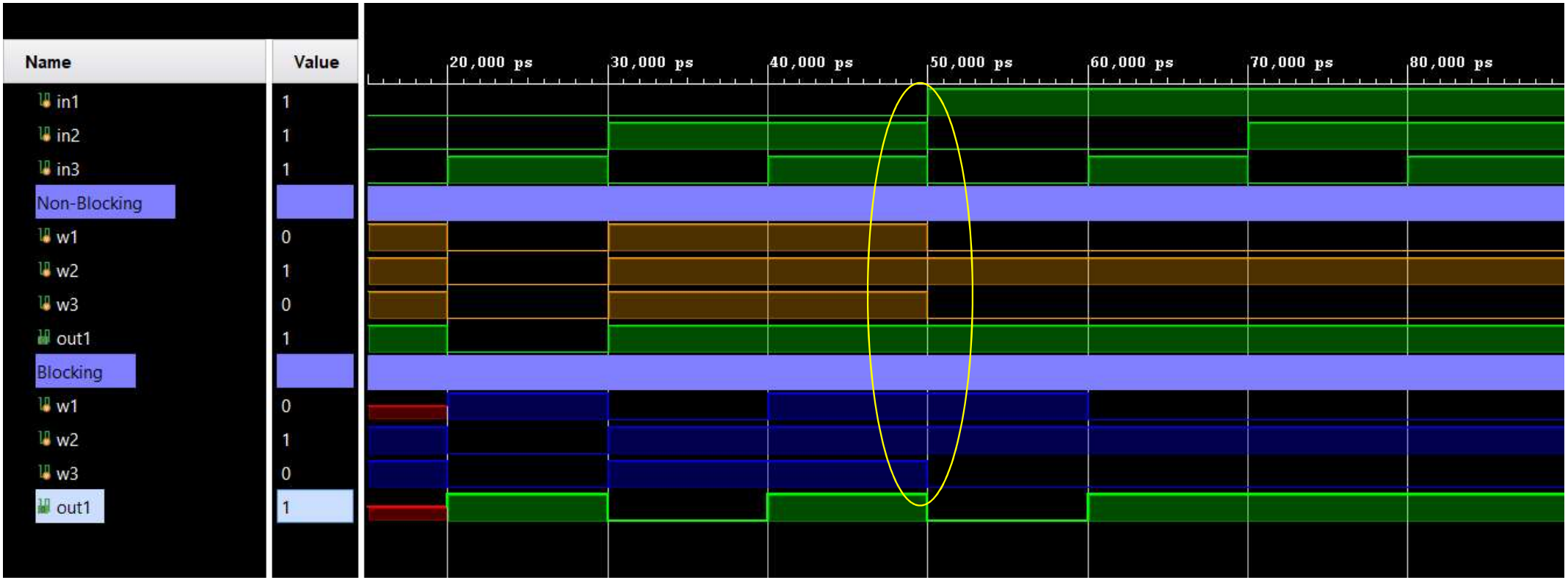
❖Example 2: Multi level logic



Initially: w1 = 1, w2 = 1, w3=0; In1=1, in2=0, in3=0

```
always @(*)
begin                          // Initially                                          --w1=1, w2=1, w3=0;  i1=1 i2=0 i3=0
    w1 <= w3 & w2;             //          w1 = w3&w2 = 0&1 = 0 is scheduled         --w1=1, w2=1, w3=0;
    w2 <= in1 | in2 | ~in3;    //          w2 = in1|in2|~in3 = 1 is scheduled        --w1=1, w2=1, w3=0;
    w3 <= in1 ^ w2;            //  w3 = in1 xor w2 = 1 xor 1 = 0 is scheduled        --w1=1, w2=1, w3=0;
end
                               // assignments happen when always block ends,  --w1=0, w2=1, w3=0;
                               // Change in w1 will trigger the block again!
```

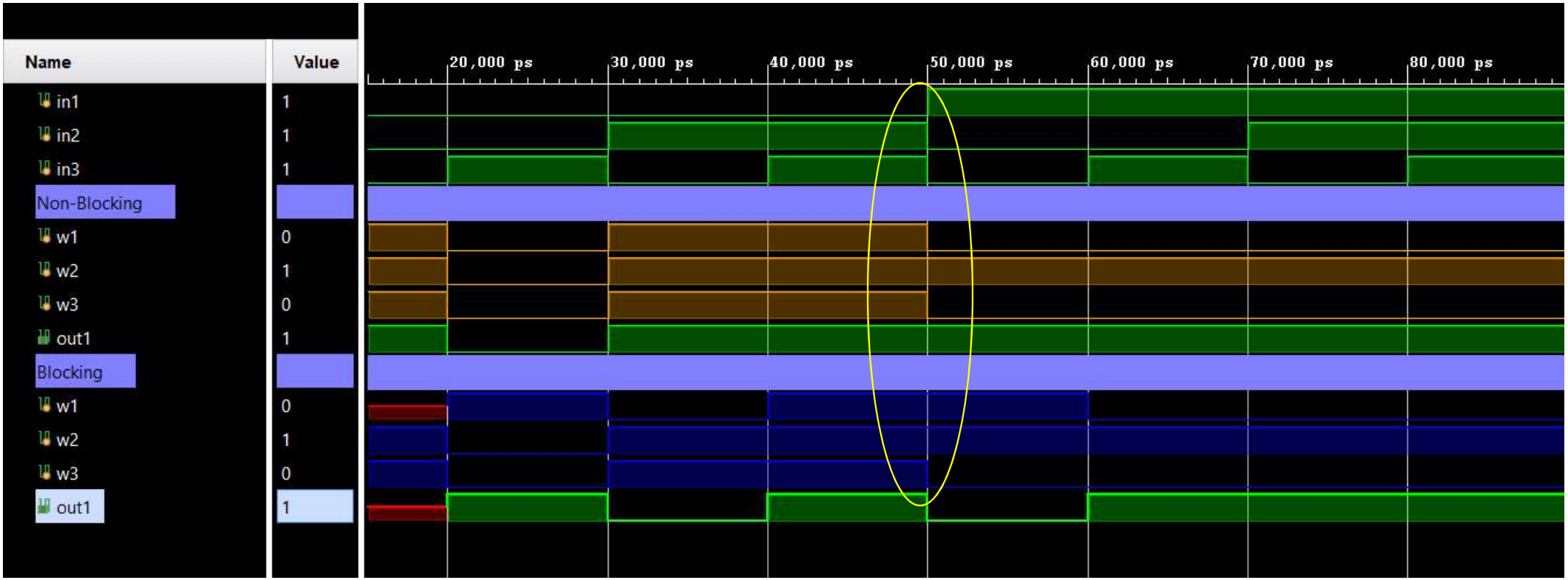
❖Example 2: Multi level logic



Initially: w1 = 1, w2 = 1, w3=0; In1=1, in2=0, in3=0

```
always @(*)
begin                          // Initially                                          --w1=0, w2=1, w3=0;  i1=1 i2=0 i3=0
    w1 <= w3 & w2;             //           w1 = w3&w2 = 0&1 = 0 is scheduled        --w1=0, w2=1, w3=0;
    w2 <= in1 | in2 | ~in3;    //          w2 = in1|in2|~in3 = 1 is scheduled        --w1=0, w2=1, w3=0;
    w3 <= in1 ^ w2;            //  w3 = in1 xor w2 = 1 xor 1 = 0 is scheduled        --w1=0, w2=1, w3=0;
end
                               // assignments happen when always block ends,  --w1=0, w2=1, w3=0;
                               // No more changes happened to trigger the block again, values will stay like this.
```

❖Example 2: Multi level logic



Initially: w1 = 1, w2 = 1, w3=0; In1=1, in2=0, in3=0

```
always @(*)
begin                          // Initially                                          --w1=1, w2=1, w3=1;  i1=1 i2=0 i3=0
    w1 <= w3 & w2;             //           w1 = w3&w2 = 1&1 = 1 is scheduled        --w1=1, w2=1, w3=1;
    w2 <= in1 | in2 | ~in3;    //          w2 = in1|in2|~in3 = 1 is scheduled        --w1=1, w2=1, w3=1;
    w3 <= in1 ^ w2;            //  w3 = in1 xor w2 = 1 xor 1 = 0 is scheduled        --w1=1, w2=1, w3=1;
end
                               // assignments happen when always block ends,  --w1=1, w2=1, w3=0;
                               // But change in w values triggers always block again at the same time!
```

Sequential Circuits in Verilog

❖**To summarize it up:**

Non-blocking assignments will happen in parallel, but the changes in internal values may wrongly trigger the always block again. That's the reason why it is not recommended to use non-blocking assignments while coding pure combinational. Blocking assignments make assignments in sequence, without getting affected from other Verilog statements/constructs.

Blocking

- ❑When the procedural block is triggered, statements are evaluated sequentially
- ❑Executions happen at the end of the related statement
- ❑Best practice for this assignment is to describe pure combinational logic within always blocks (Assignments will happen sequentially)

Non-blocking

- ❑When the procedural block is triggered, statements are scheduled
- ❑Executions happen at the end of always block, parallelly.
- ❑Best practice for this assignment is to describe sequential logic within always blocks (Why?)

Sequential Circuits in Verilog

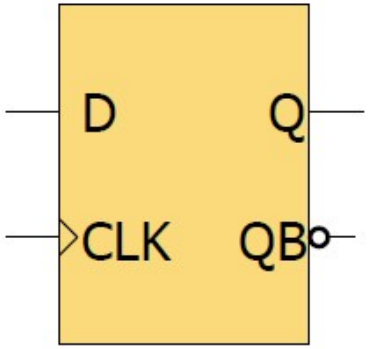
- **Example 3: A simple clock triggered storage element**

```
module an_FF(
    input D,clk,
    output reg Q,QB
);

    always @(posedge clk)
    begin
        Q <= D;
        QB <= !D;
    end

endmodule
```

Style-1

```
module an_FF_2(
    input D,clk,
    output Q,QB
);

    reg r_Q, r_QB;

    always @(posedge clk)
    begin
        r_Q <= D;
        r_QB <= !D;
    end


    assign Q = r_Q;
    assign QB = r_QB;

endmodule
```

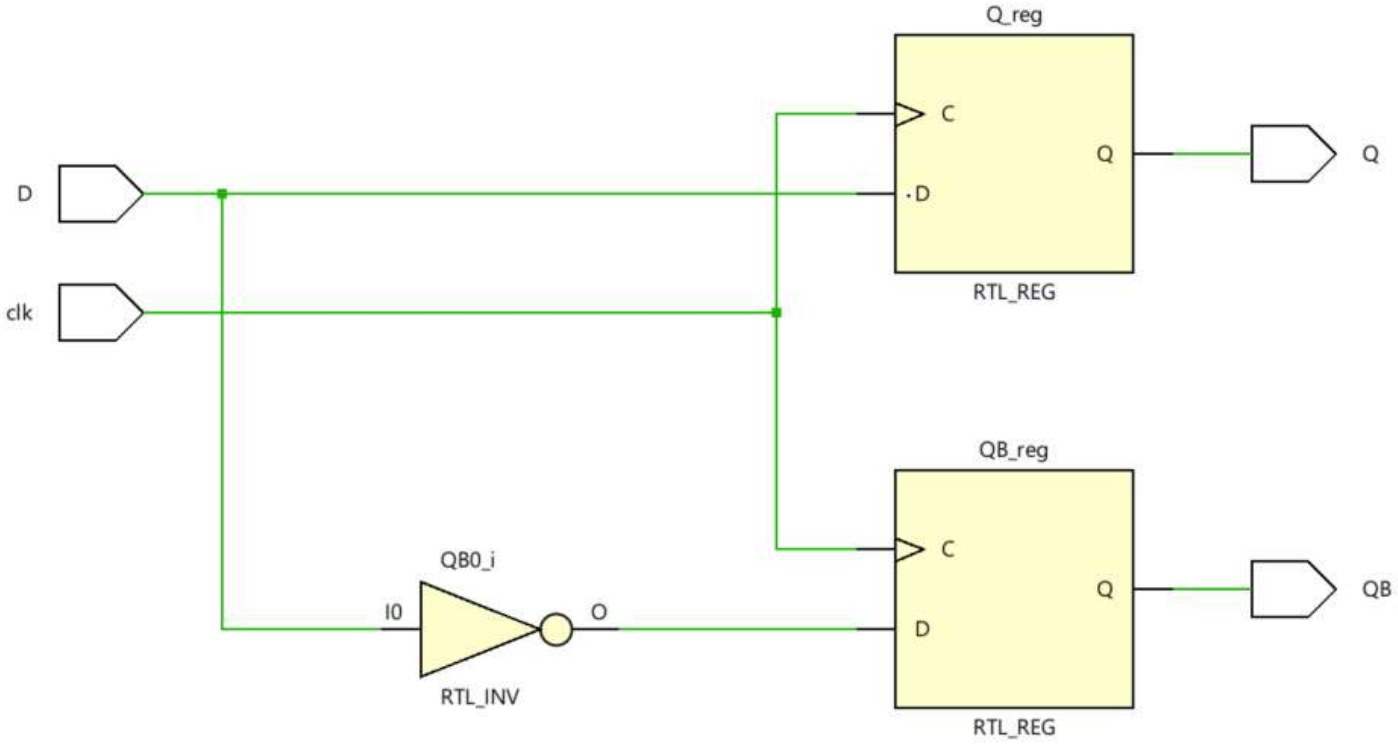
Style-2



"**posedge**" represents rising edge triggering, "**negedge**" represents falling edge triggering

Sequential Circuits in Verilog

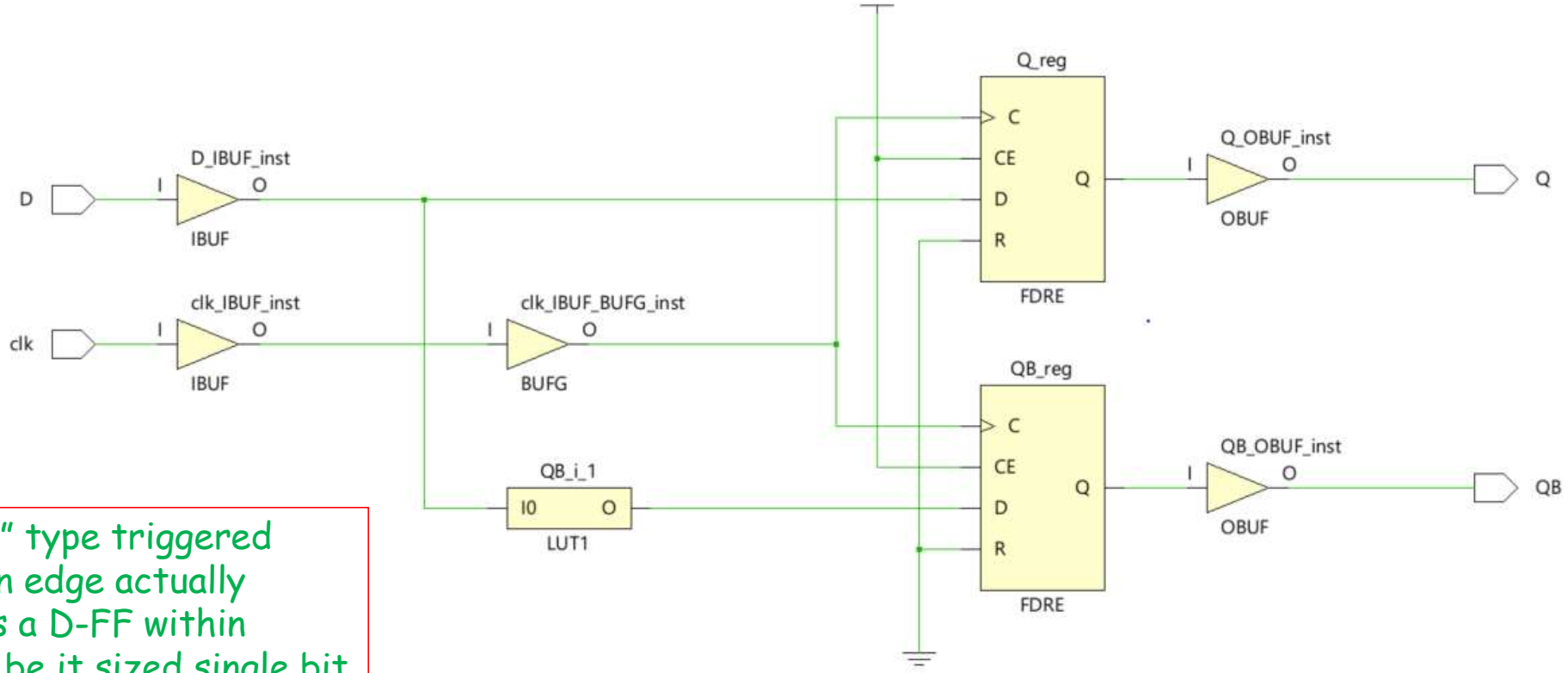
- **Example 3: A simple clock triggered storage element**



Resulting RTL

Sequential Circuits in Verilog

- **Example 3: A simple clock triggered storage element**



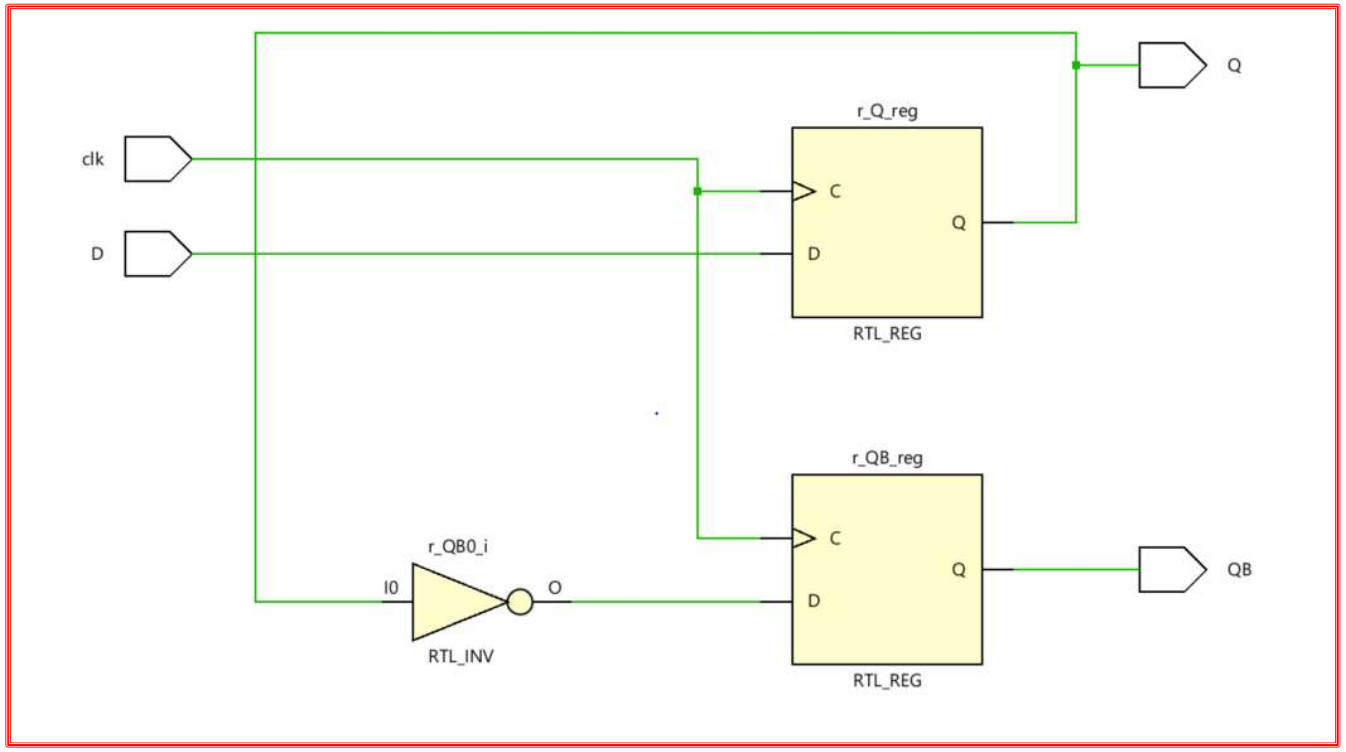
A "reg" type triggered with an edge actually models a D-FF within FPGA, be it sized single bit or multi-bit .

Synthesis Result

Sequential Circuits in Verilog

- **A common design error**

```
module an_FF(
    input D,clk,
    output reg Q,QB
);

    always @(posedge clk)
    begin
        Q <= D;
        QB <= !Q;
    end

endmodule
```



QB will get the inverted Q value of previous cycle. Desired QB result will appear one clock cycle late

Sequential Circuits in Verilog

- **Example 4: Shift Register**

```
module shiftreg1(
    input D,clk,
    output Q
);

    reg [2:0] int_regs;

    always @(posedge clk)
    begin
        int_regs[0] <= D;
        int_regs[1] <= int_regs[0];
        int_regs[2] <= int_regs[1];
    end

    assign Q = int_regs[2];

endmodule
```

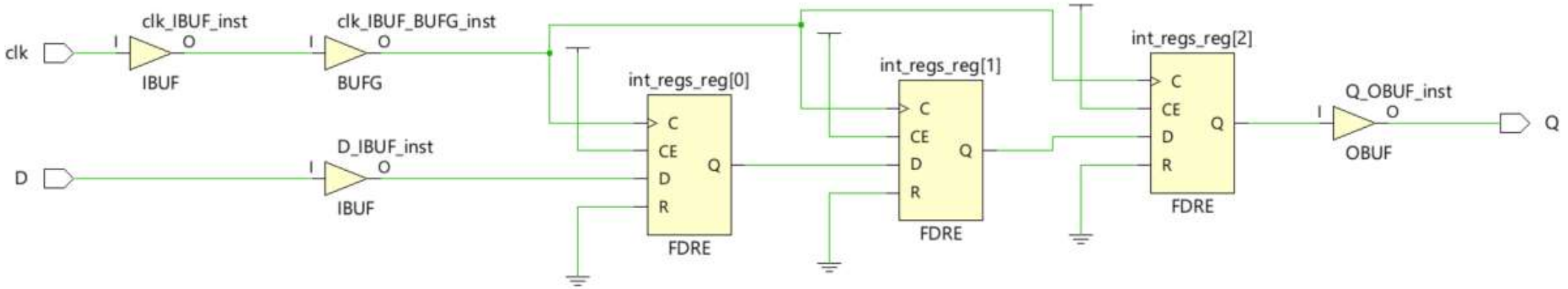
Blocking (=) or Non-blocking (<=) for sequential?

```
module shiftreg2(
    input D,clk,
    output Q
);

    reg [2:0] int_regs;

    always @(posedge clk)
    begin
        int_regs[0] = D;
        int_regs[1] = int_regs[0];
        int_regs[2] = int_regs[1];
    end

    assign Q = int_regs[2];

endmodule
```

Sequential Circuits in Verilog

- **Example 4: Shift Register**

Blocking (=) or Non-blocking (<=) for sequential?



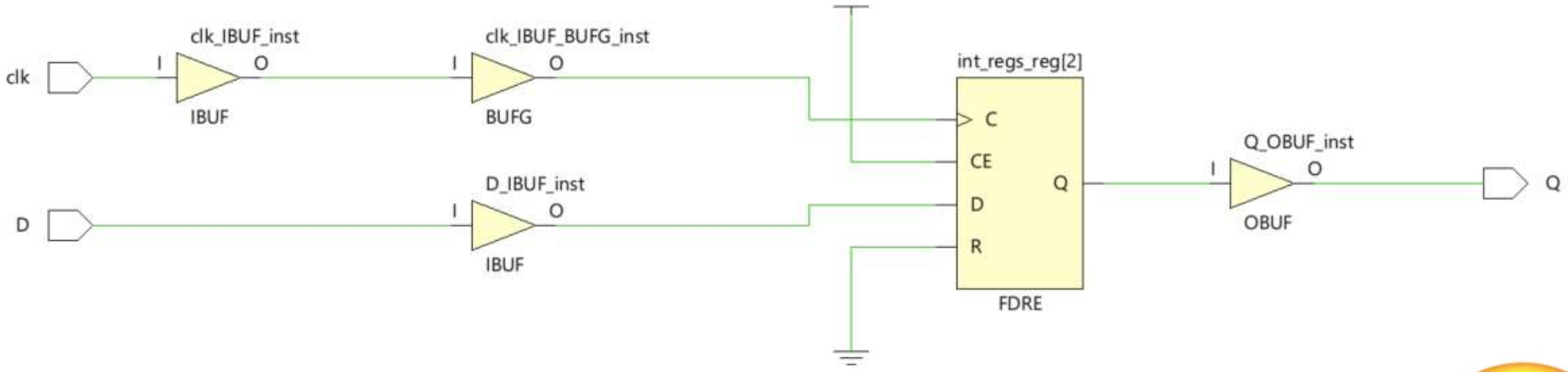
Synthesis Result for Non-blocking

We see the expected circuit

Sequential Circuits in Verilog

- **Example 4: Shift Register**

Blocking (=) or Non-blocking (<=) for sequential?



Synthesis Result for Blocking

Isn't that supposed to be a 3-bit shift register?

Sequential Circuits in Verilog

Synchronous and Asynchronous Inputs

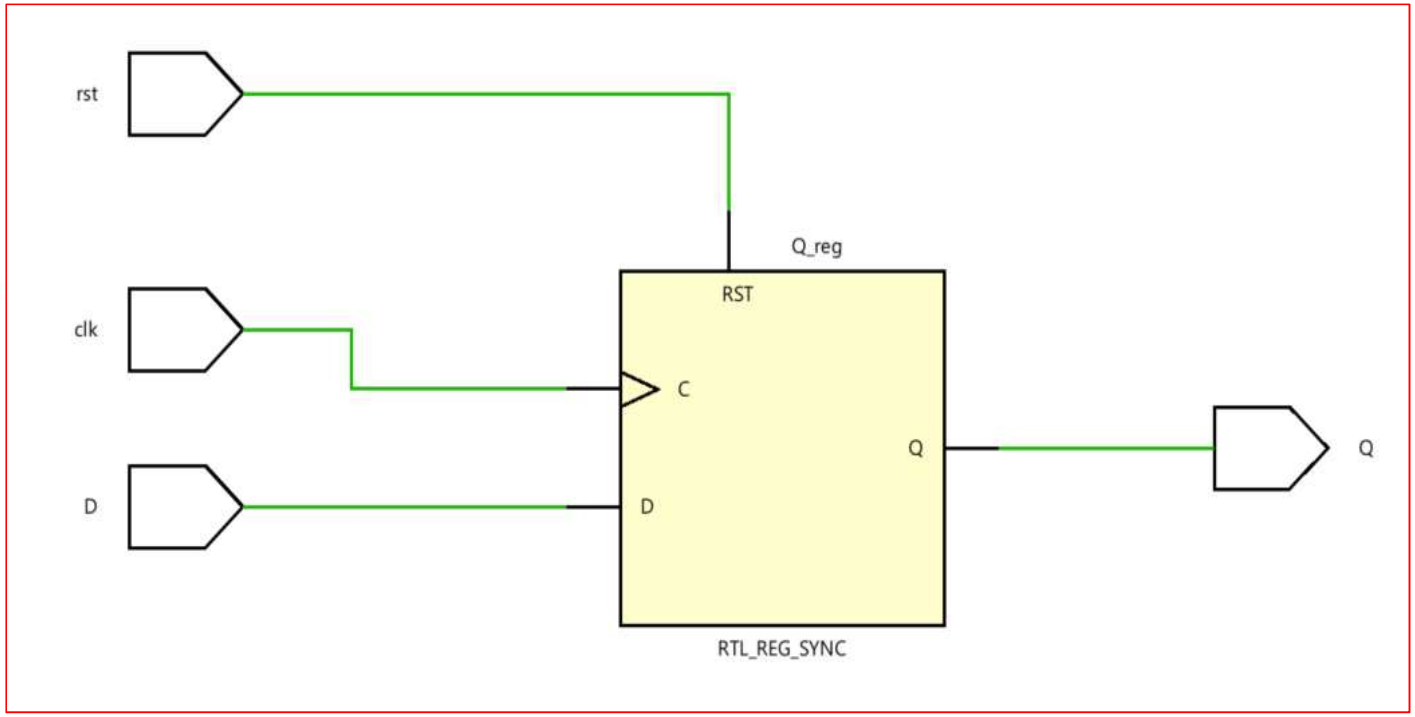
- Synchronous inputs are synchronized to clock.
- Asynchronous inputs are not, and cause immediate change
- Asynchronous inputs normally have precedence over synchronous inputs.

```
always@( /*posedge/negedge clock*/ or /*posedge/negedge async_signal*/)
begin
    if(/* async_signal value check */)
    begin
        // asynchronous assignments
    end
    else if (/* async_signal value check */)
    begin
        // asynchronous assignments
    end

    ...

    else
    begin
        // sycnhronous assignments
    end


end
```

Sequential Circuits in Verilog

- **Example 5: Flip_flop with synchronous active-1 reset**

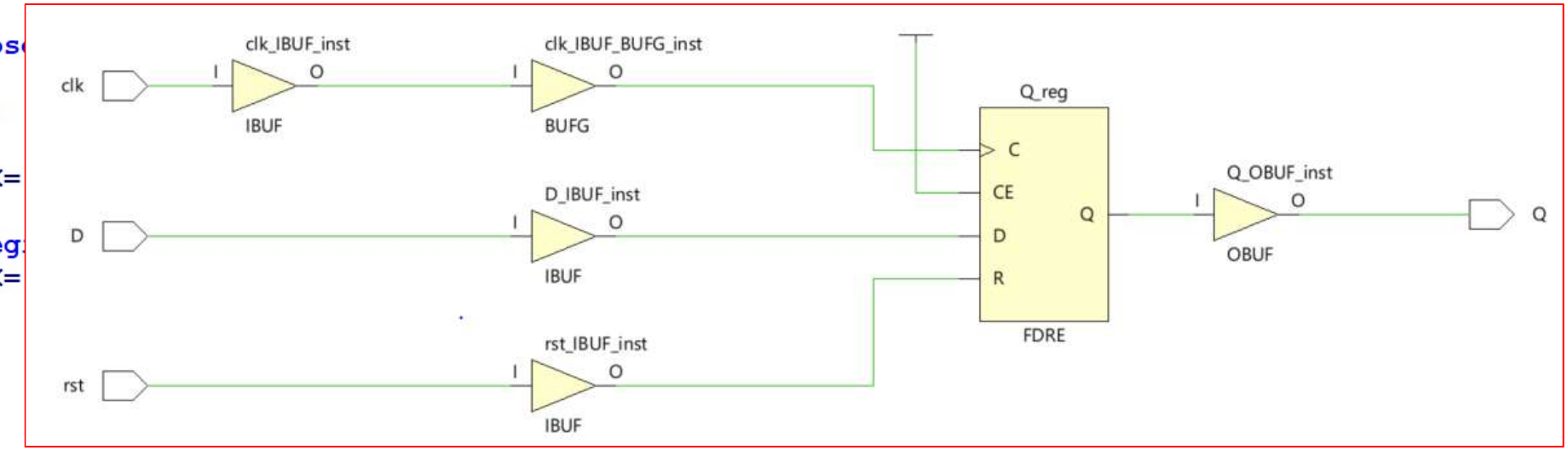
```
module an_FF(
    input D,clk,rst
    output reg Q
);

    always @(posedge clk)
    begin
        if(rst)    // Synchronous
        begin
            Q <= 0;
        end
        else begin
            Q <= D;
        end
    end

endmodule
```



Sequential Circuits in Verilog

- **Example 5: Flip_flop with synchronous active-1 reset**

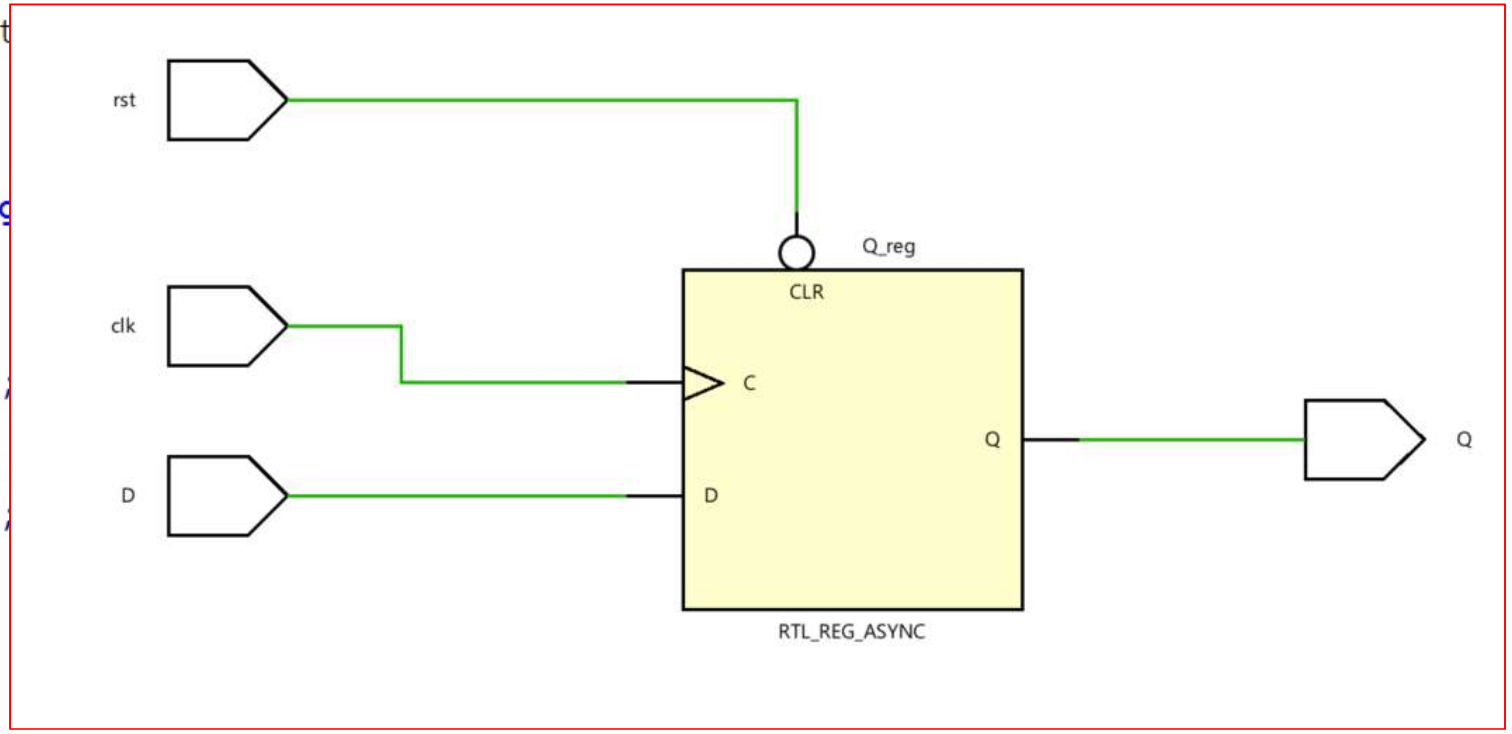
```
module an_FF(
    input D,clk,rst
    output reg Q
);

    always @(pos
    begin
        if(rst)
        begin
            Q <=
        end
        else beg
            Q <=
        end
    end

endmodule
```



Sequential Circuits in Verilog

- **Example 6: Flip_flop with asynchronous active-0 reset**

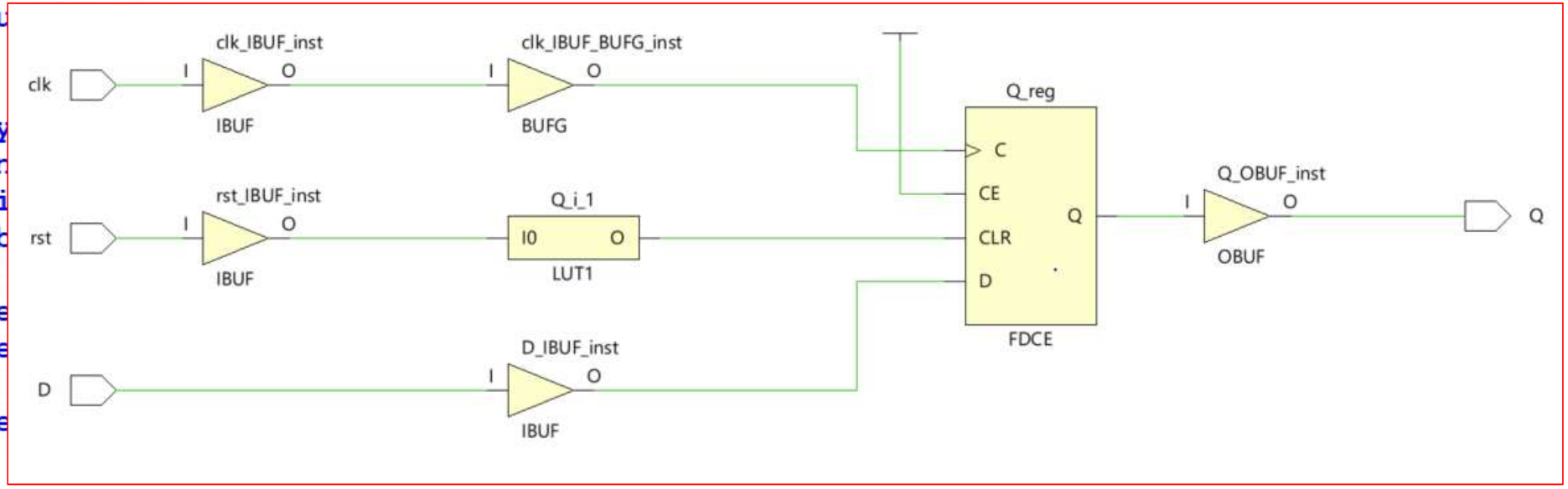
```
module an_FF(
    input D,clk,rst
    output reg Q
);

    always @(posedg
    begin
        if(!rst)
        begin
            Q <= 0;
        end
        else begin
            Q <= D;
        end
    end

endmodule
```



Sequential Circuits in Verilog

- **Example 6: Flip_flop with asynchronous active-0 reset**

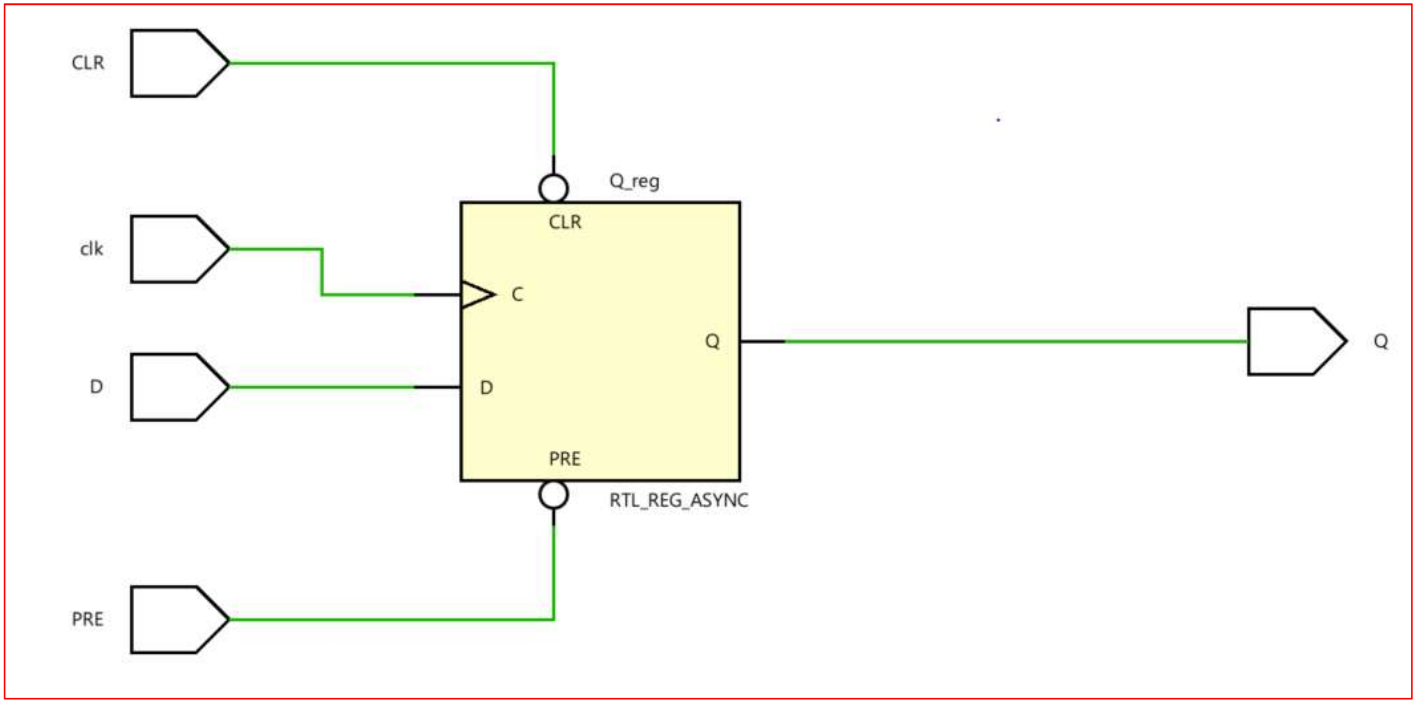
```
module an_FF(
    input D,clk,rst
    outpu
);

    alway
    begin
        i
        b

        e
        e

        e
    end

endmodule
```



Sequential Circuits in Verilog

- **Example 7: Flip_flop with asynchronous Preset and Clear**

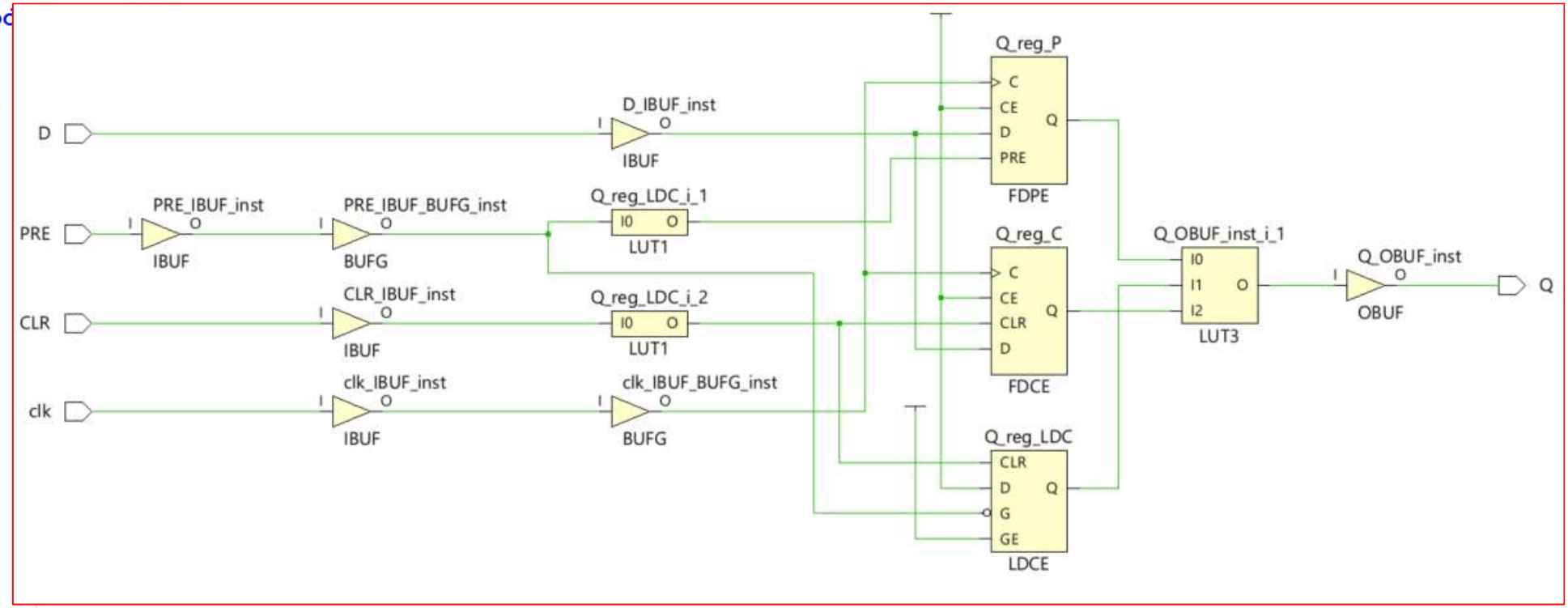
```
module an_FF(
    input D,clk,PRE,CLR,
    output reg Q
);

    always @(posedge clk
    begin
        if(!CLR)
        begin
            Q <= 0;
        end
        else if(!PRE)
        begin
            Q <= 1;
        end
        else begin
            Q <= D;
        end
    end

endmodule
```



Sequential Circuits in Verilog

- **Example 7: Flip_flop with asynchronous Preset and Clear**

```
mod
);
endmodule
```



Sequential Circuits in Verilog

- **Example 7: Flip_flop with asynchronous Preset and Clear**

```
module an_FF(
    input D,clk,PRE,CLR,
    output reg Q
);

    always @(posedge clk or negedge PRE or negedge CLR)
    begin
        if(!CLR)
        begin
            Q <= 0;
        end
        else if(!PRE)
        begin
            Q <= 1;
        end
        else begin
            Q <= D;
        end
    end

endmodule
```

What happens when both PRE and CLR are 0 ?

Sequential Circuits in Verilog

Tips for better coding

- Use blocking assignment for coding pure combinational, use non-blocking assignment for sequential.
- Always include some reset procedure into your sequential design. Be sure to reset all regs.
- Multiple always blocks can exist within the same module, BUT do not attempt to change the same reg within different always blocks, this will cause multi-driven net issue.
- It is good practice to split your code's combinational and sequential parts. That will make codes more neat and readable.
- For better maintenance and readability, try to divide the work into multiple always blocks. Specify what that block does with comments. Do not force to fit all the work within the same always block.
- Remember that you can safely use operators/dataflow modeling within always blocks & non-blocking statements

Sequential Circuits in Verilog

Generating Clocks in Nexys Boards

- The board has an internal oscillator
- Related constraints in board master files

```
## Clock signal
#set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
```

- For different frequencies, either a frequency divider circuit should be implemented or Xilinx Clock Wizard IP should be used. This IP creates clocks from another clock source, using built-in PLL circuits.

Sequential Circuits in Verilog

Generating Clocks in Nexys Boards

```
## Clock signal
#set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }];
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
```

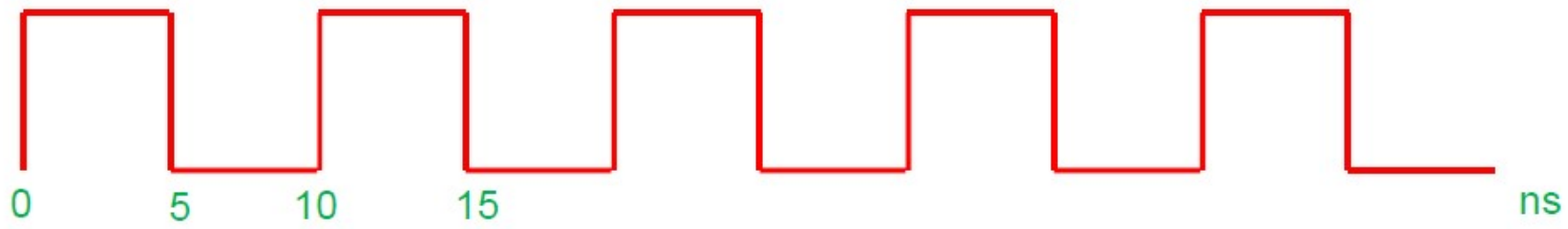
- The package pin E3 contains the output of internal oscillator that generates a 100MHz clock signal.
- Your design's clock pin should be connected to the place that is indicated by " CLK100MHZ"
- Note that the command "create_clock" solely creates a virtual clock for timing analysis. It does not set the internal oscillator frequency!

  - -name <sys_clk_pin> // sys_clk_pin is the arbitrary name you give to the virtual clock object. This virtual clock is used by timing analyzer as source
  - -period <period_value> // In nanoseconds
  - -waveform [<val1> <val2>] // Within a period, clock rising edge is seen at time "val1", falling edge is seen at time <val2>

Sequential Circuits in Verilog

Generating Clocks in Nexys Boards

```
## Clock signal
#set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }];
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];
```



Resulting clock waveform

Sequential Circuits in Verilog

Generating Clocks in Nexys Boards

- In real world, clocks will not be that ideal. Synthesis tools allow the virtual clock waveform to include non-idealities such as skew and latency. These can be specified using constraints, and is useful for timing critical designs.