University of San Carlos | Department of
COMPUTER ENGINEERING

CpE 3101L – Introduction to HDL

Unit 6: Behavioral Modeling of Sequential Circuits

Lecture 1

Outline

- Behavioral Modeling of Sequential Circuits
- *Always* Block and Sensitivity List
- Latches, Flip-flops, Registers, and Counters
- Non-blocking vs. Blocking Assignments
- Synchronous vs. Asynchronous Resets
- Combinational vs. Sequential Logic
- Guidelines for Good Synthesis of Sequential Circuits
- Simulation for Sequential Circuits

Behavioral Modeling of Sequential Circuits

- **Behavioral modeling** represents digital circuits at a functional and algorithmic level.
  - Involves describing at a higher level of abstraction
- Mostly used to describe sequential logic.
- **Behavioral HDL models** describe sequential circuits by their *behavior (functionality).*

Always Block and Sensitivity List

- Behavioral descriptions uses procedural assignment statements with keyword ***always***
  - ***always*** can also be used in simulations (together with ***initial***)
  - ***always*** is synthesizable (unlike ***initial***)

- Consists of a sensitivity list *(an event control expression)*

- Sensitivity list is a list of signals that is monitored for changes
  - Whenever one of the signals in the sensitivity list changes its value, the statements inside the ***always*** block are ***evaluated sequentially***.

Always Block and Sensitivity List

- All signals assigned inside the *always* block must be declared as *reg*

- Contrary to a *wire* (a *net* data type), a *reg* retains its value until a new value is assigned (*variable* data type)

```
// declare reg here;

always @( [sensitivity_list] )
begin
   // local variable declarations here; (optional)

   // body: composed of procedural statements;
end
```

- Note: Use of **begin-end** are required when the contents of the *always* block have *more than one statement*

Latches, Flip-flops, and Registers

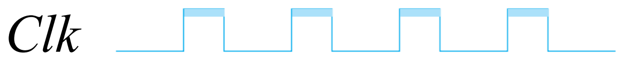
- Use of same behavioral construct (*always* block), but change in sensitivity list.

- All sequential circuits must have *clock signal*.
  - Latches and flip-flops differ in the clock sensitivity.
  - Latches are level-sensitive *(positive or negative level)*
  - Flip-flops are edge-sensitive *(positive/rising or negative/falling edge)*

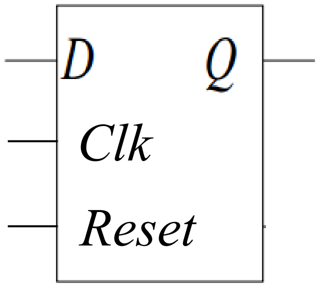
- If D and Q are more than 1-bit wide, they are usually called **registers** and are edge-sensitive

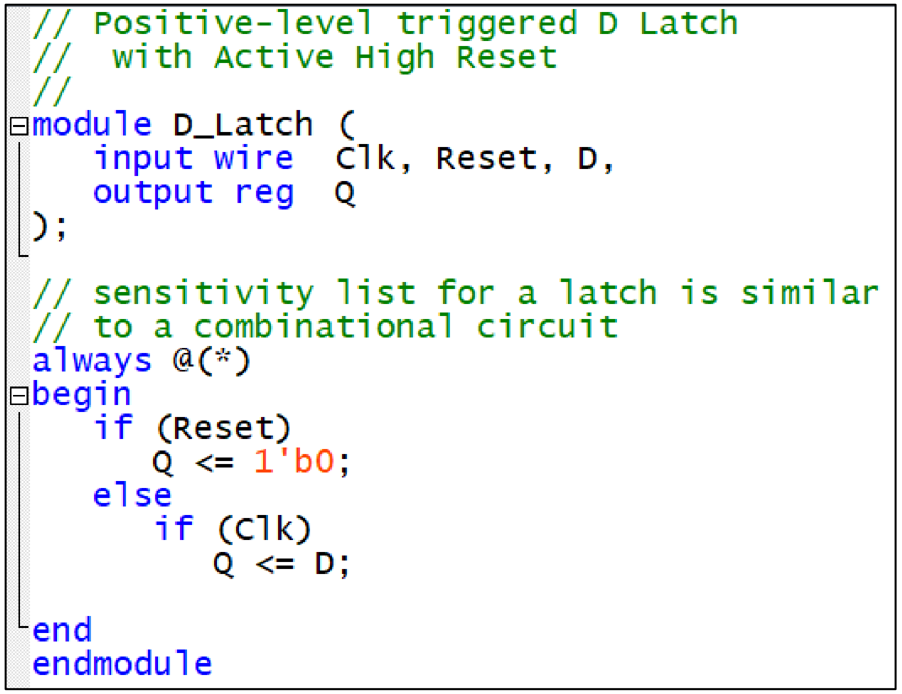
Example 1: Positive-level D Latch

- Verilog behavioral description of a positive-level D latch:



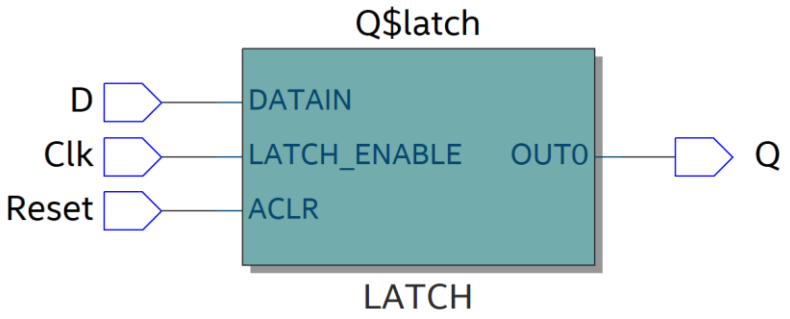
Response to positive level



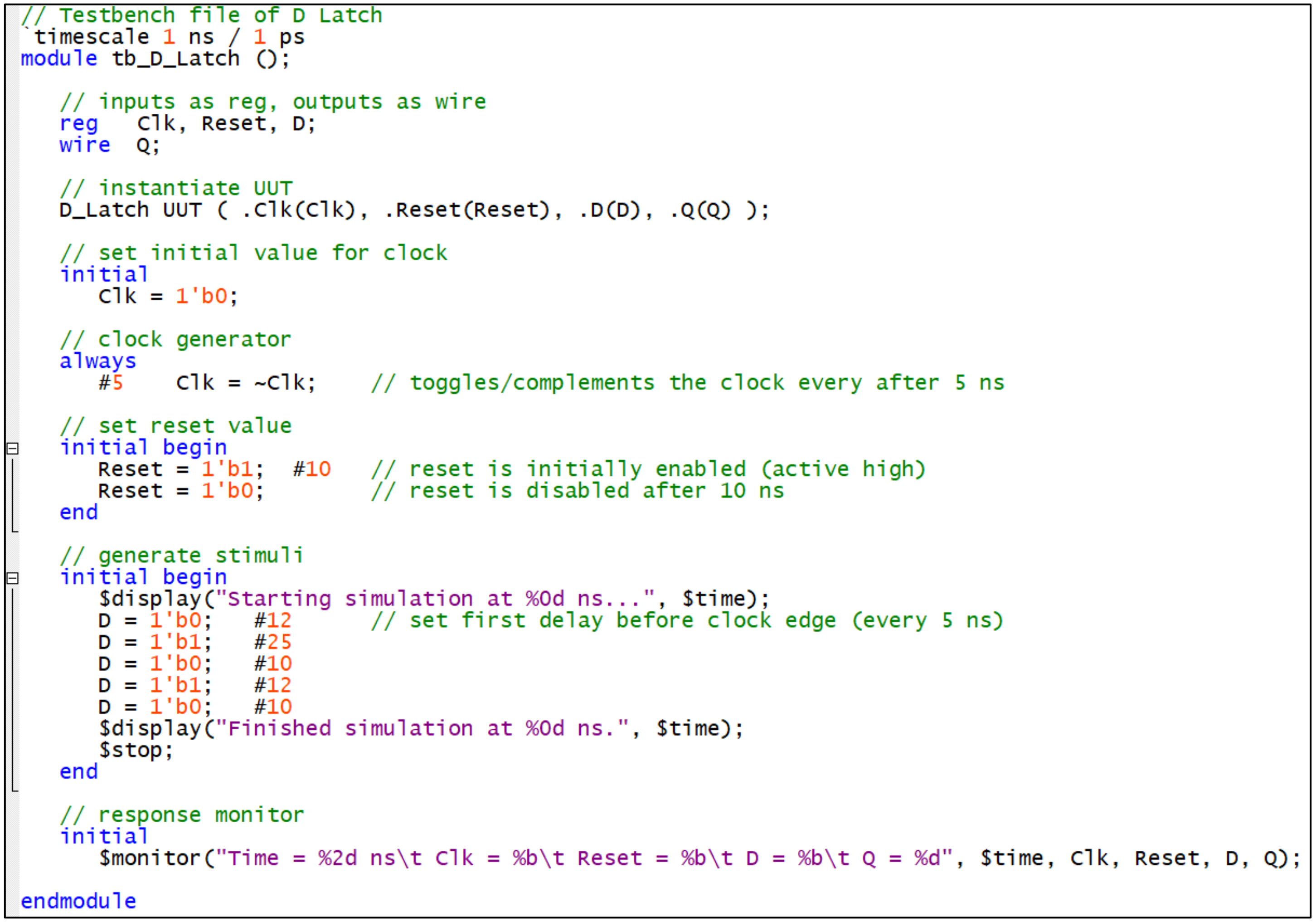


```
// Positive-level triggered D Latch
//  with Active High Reset
//
module D_Latch (
    input wire  Clk, Reset, D,
    output reg  Q
);

// sensitivity list for a latch is similar
// to a combinational circuit
always @(*)
begin
    if (Reset)
        Q <= 1'b0;
    else
        if (Clk)
            Q <= D;

end
endmodule
```

- RTL Schematic View:



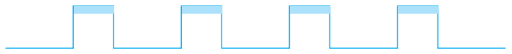
Example 1: Testbench File (D Latch)



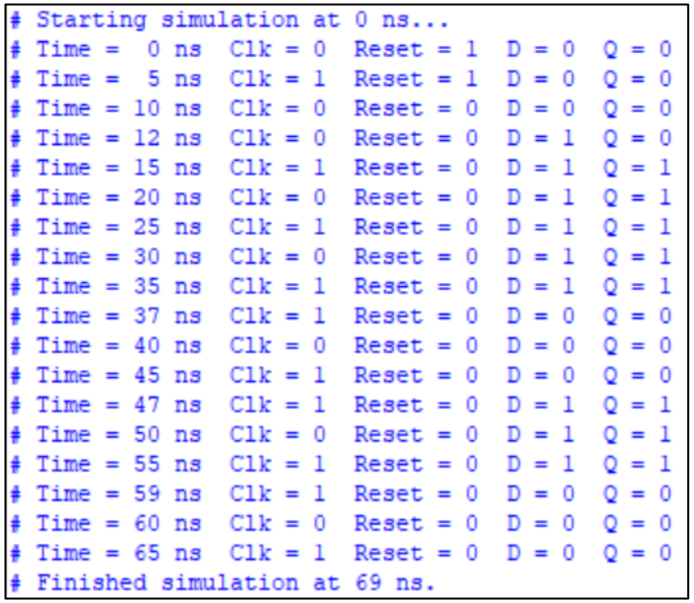
```verilog
// Testbench file of D Latch
`timescale 1 ns / 1 ps
module tb_D_Latch ();

   // inputs as reg, outputs as wire
   reg   Clk, Reset, D;
   wire  Q;

   // instantiate UUT
   D_Latch UUT ( .Clk(Clk), .Reset(Reset), .D(D), .Q(Q) );

   // set initial value for clock
   initial
      Clk = 1'b0;

   // clock generator
   always
      #5     Clk = ~Clk;     // toggles/complements the clock every after 5 ns

   // set reset value
   initial begin
      Reset = 1'b1;  #10     // reset is initially enabled (active high)
      Reset = 1'b0;          // reset is disabled after 10 ns
   end

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);
      D = 1'b0;   #12        // set first delay before clock edge (every 5 ns)
      D = 1'b1;   #25
      D = 1'b0;   #10
      D = 1'b1;   #12
      D = 1'b0;   #10
      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial
      $monitor("Time = %2d ns\t Clk = %b\t Reset = %b\t D = %b\t Q = %d", $time, Clk, Reset, D, Q);

endmodule
```

Example 1: Simulation (D Latch)

- Standard Output:

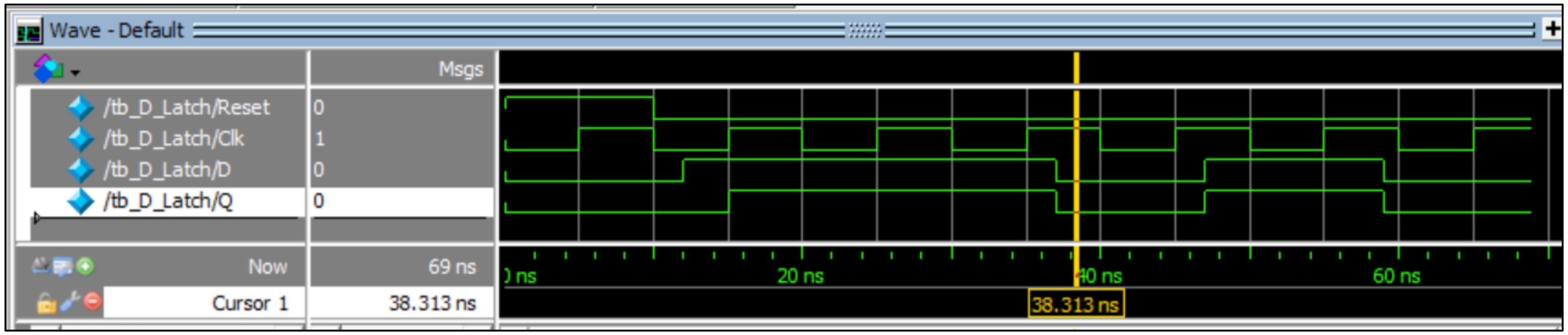


Response to positive level



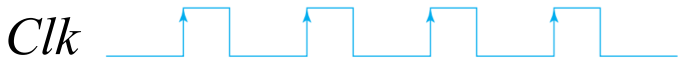
```
# Starting simulation at 0 ns...
# Time =  0 ns  Clk = 0  Reset = 1  D = 0  Q = 0
# Time =  5 ns  Clk = 1  Reset = 1  D = 0  Q = 0
# Time = 10 ns  Clk = 0  Reset = 0  D = 0  Q = 0
# Time = 12 ns  Clk = 0  Reset = 0  D = 1  Q = 0
# Time = 15 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 20 ns  Clk = 0  Reset = 0  D = 1  Q = 1
# Time = 25 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 30 ns  Clk = 0  Reset = 0  D = 1  Q = 1
# Time = 35 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 37 ns  Clk = 1  Reset = 0  D = 0  Q = 0
# Time = 40 ns  Clk = 0  Reset = 0  D = 0  Q = 0
# Time = 45 ns  Clk = 1  Reset = 0  D = 0  Q = 0
# Time = 47 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 50 ns  Clk = 0  Reset = 0  D = 1  Q = 1
# Time = 55 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 59 ns  Clk = 1  Reset = 0  D = 0  Q = 0
# Time = 60 ns  Clk = 0  Reset = 0  D = 0  Q = 0
# Time = 65 ns  Clk = 1  Reset = 0  D = 0  Q = 0
# Finished simulation at 69 ns.
```

- Waveform:

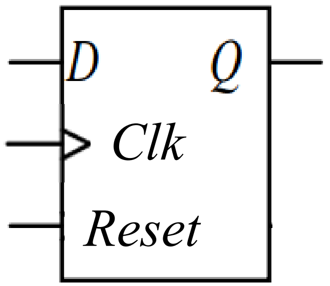


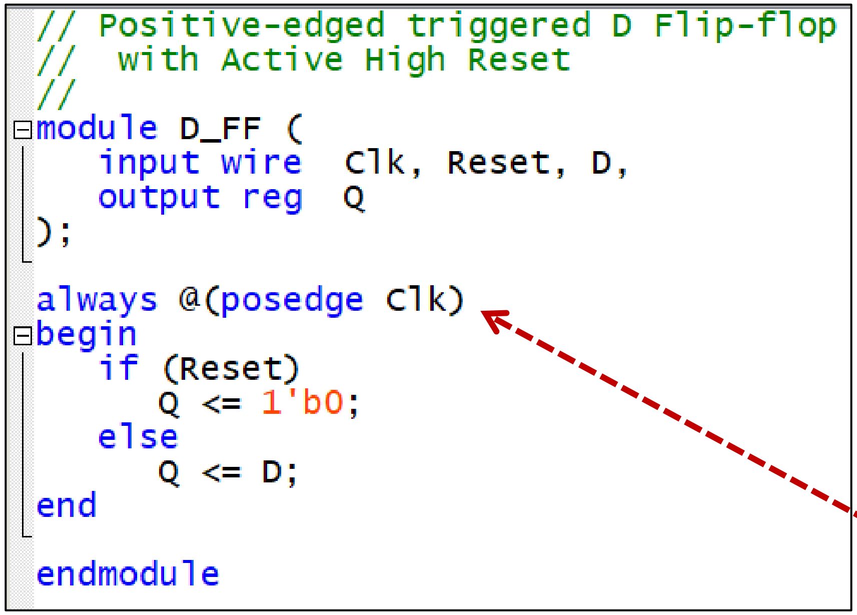
Example 2: Positive-edged D Flip-flop

- Verilog behavioral description of a positive-edged D flip-flop:

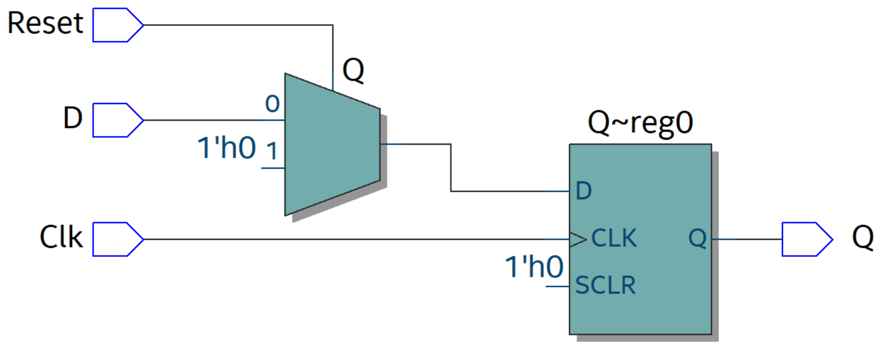


Positive-edge response





```
// Positive-edged triggered D Flip-flop
//  with Active High Reset
//
module D_FF (
    input wire  Clk, Reset, D,
    output reg  Q
);

always @(posedge Clk)
begin
    if (Reset)
        Q <= 1'b0;
    else
        Q <= D;
end

endmodule
```

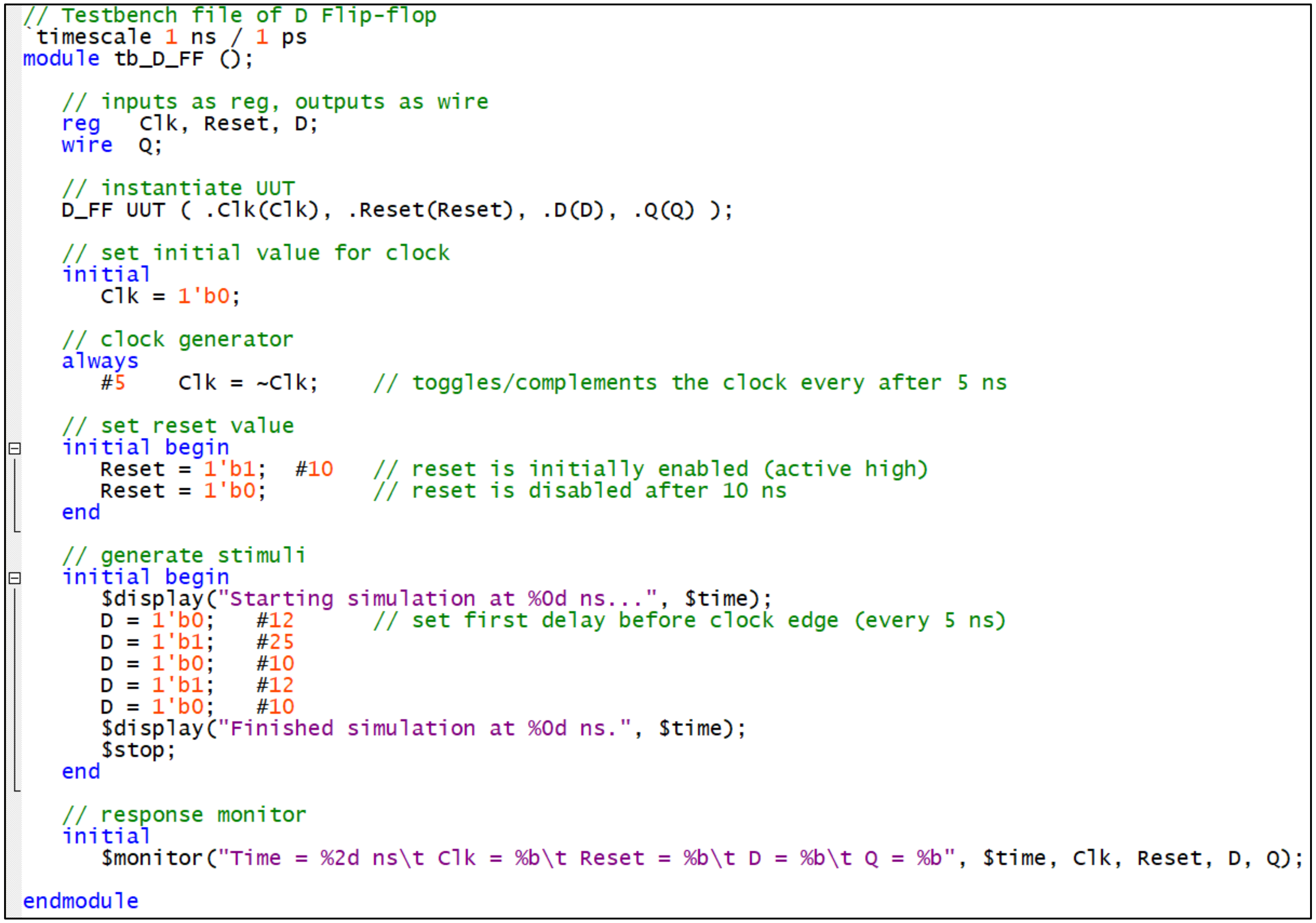


See the difference in the sensitivity list

If negative-edged triggered:

```
always @(negedge Clk)
```

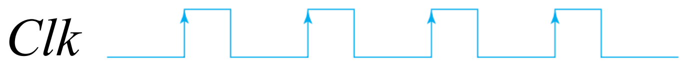
Example 2: Testbench File (D F/F)



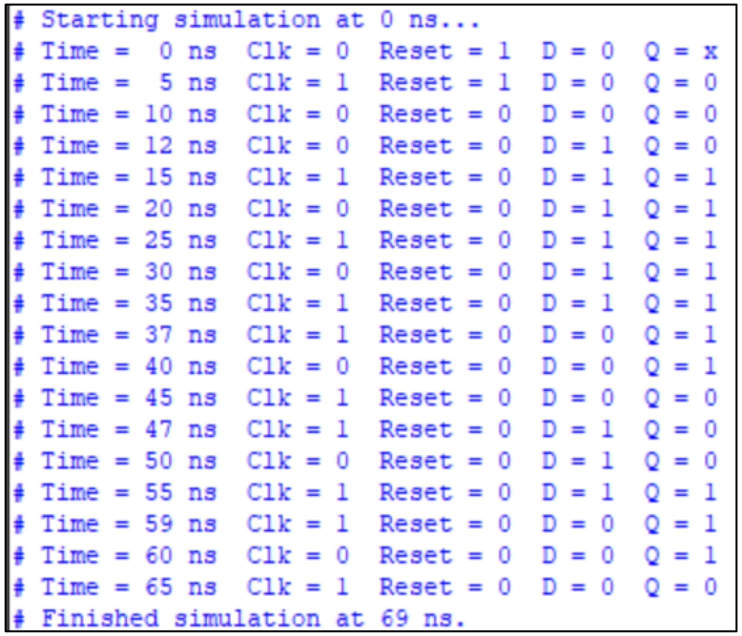
```verilog
// Testbench file of D Flip-flop
`timescale 1 ns / 1 ps
module tb_D_FF ();

   // inputs as reg, outputs as wire
   reg   Clk, Reset, D;
   wire  Q;

   // instantiate UUT
   D_FF UUT ( .Clk(Clk), .Reset(Reset), .D(D), .Q(Q) );

   // set initial value for clock
   initial
      Clk = 1'b0;

   // clock generator
   always
      #5    Clk = ~Clk;    // toggles/complements the clock every after 5 ns

   // set reset value
   initial begin
      Reset = 1'b1;  #10   // reset is initially enabled (active high)
      Reset = 1'b0;        // reset is disabled after 10 ns
   end

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);
      D = 1'b0;   #12      // set first delay before clock edge (every 5 ns)
      D = 1'b1;   #25
      D = 1'b0;   #10
      D = 1'b1;   #12
      D = 1'b0;   #10
      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial
      $monitor("Time = %2d ns\t Clk = %b\t Reset = %b\t D = %b\t Q = %b", $time, Clk, Reset, D, Q);

endmodule
```

Example 2: Simulation (D F/F)

- Standard Output:

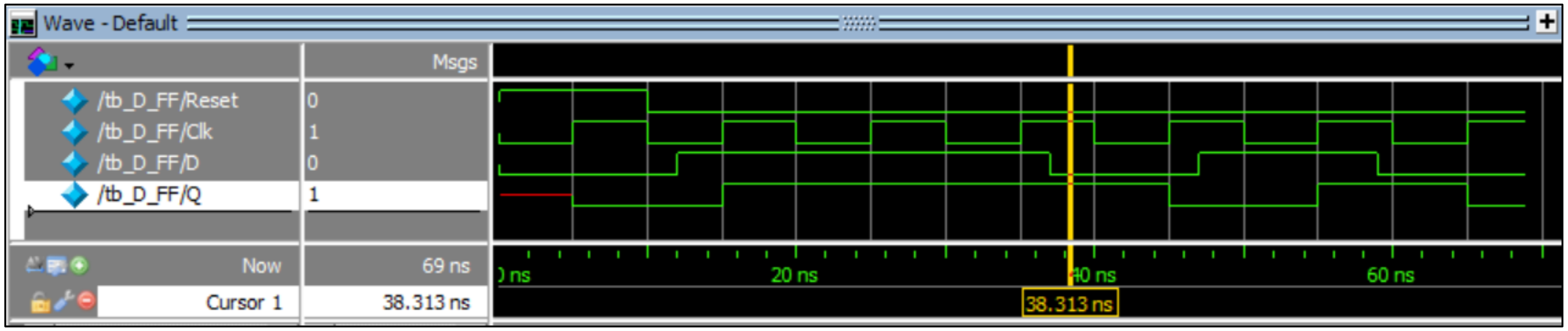


Positive-edge response



```
# Starting simulation at 0 ns...
# Time =  0 ns  Clk = 0  Reset = 1  D = 0  Q = x
# Time =  5 ns  Clk = 1  Reset = 1  D = 0  Q = 0
# Time = 10 ns  Clk = 0  Reset = 0  D = 0  Q = 0
# Time = 12 ns  Clk = 0  Reset = 0  D = 1  Q = 0
# Time = 15 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 20 ns  Clk = 0  Reset = 0  D = 1  Q = 1
# Time = 25 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 30 ns  Clk = 0  Reset = 0  D = 1  Q = 1
# Time = 35 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 37 ns  Clk = 1  Reset = 0  D = 0  Q = 1
# Time = 40 ns  Clk = 0  Reset = 0  D = 0  Q = 1
# Time = 45 ns  Clk = 1  Reset = 0  D = 0  Q = 0
# Time = 47 ns  Clk = 1  Reset = 0  D = 1  Q = 0
# Time = 50 ns  Clk = 0  Reset = 0  D = 1  Q = 0
# Time = 55 ns  Clk = 1  Reset = 0  D = 1  Q = 1
# Time = 59 ns  Clk = 1  Reset = 0  D = 0  Q = 1
# Time = 60 ns  Clk = 0  Reset = 0  D = 0  Q = 1
# Time = 65 ns  Clk = 1  Reset = 0  D = 0  Q = 0
# Finished simulation at 69 ns.
```

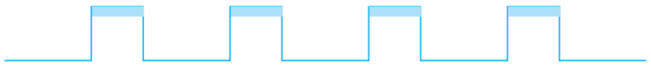
- Waveform:

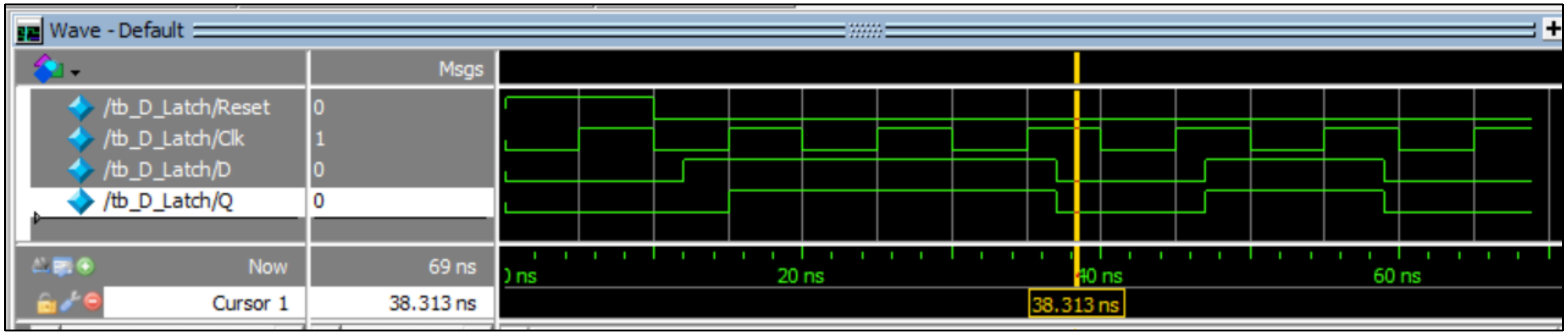


	Msgs
/tb_D_FF/Reset	0
/tb_D_FF/Clk	1
/tb_D_FF/D	0
/tb_D_FF/Q	1
Now	69 ns
Cursor 1	38.313 ns

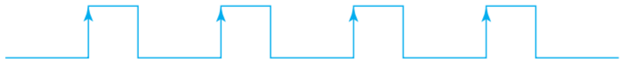
Compare Waveforms

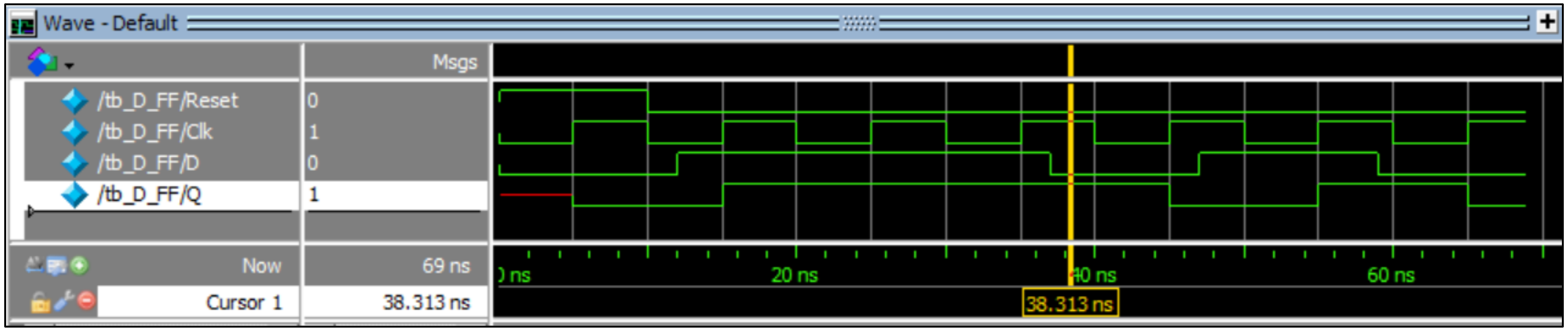
Positive-level triggered D Latch





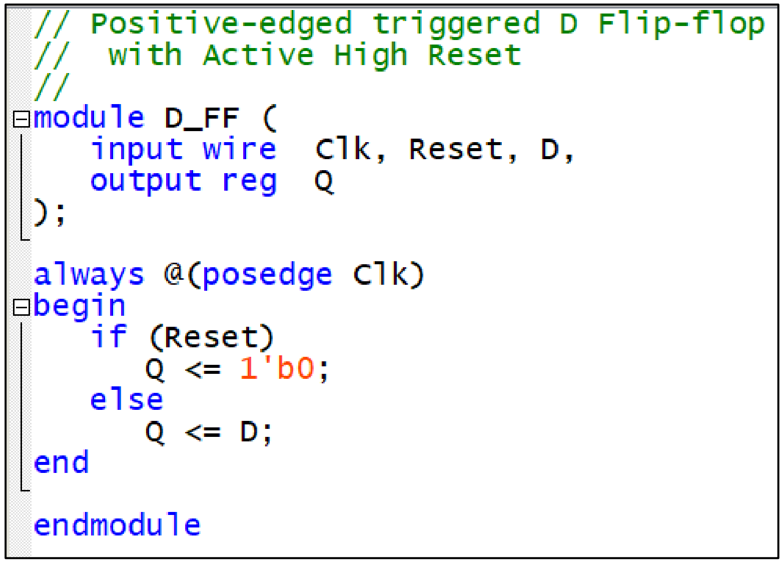
Positive-edged triggered D Flip-flop





Non-blocking Assignment

- Uses **<= operator** in assigning values in a *reg (inside an always block)*
- Executes all statements in the *always* block concurrently
- Evaluated and assigned in two steps:
  - The right-hand side is evaluated immediately.
  - The assignment to the left-hand side is postponed until other evaluations in the current time step are completed
- **Non-blocking assignment** should be used in sequential logic



```
// Positive-edged triggered D Flip-flop
//  with Active High Reset
//
module D_FF (
    input wire  Clk, Reset, D,
    output reg  Q
);

always @(posedge Clk)
begin
    if (Reset)
        Q <= 1'b0;
    else
        Q <= D;
end

endmodule
```

Blocking vs. Non-blocking

- Blocking Assignment (=)
  - Used in combinational logic

```
// using blocking statements
reg x, y;
always @(*)
begin
    x = a & b;
    y = x | c;
end
```

Execution:

1. x is evaluated
2. y is evaluated using value from (1)

- Non-blocking Assignment (<=)
  - Used in sequential logic

```
// using non-blocking statements
reg x, y;
always @(*)
begin
    x <= a & b;
    y <= x | c;
end
```

This example is for illustration and discussion purposes only.
Combinational logic should always use blocking assignments (=).

Execution:

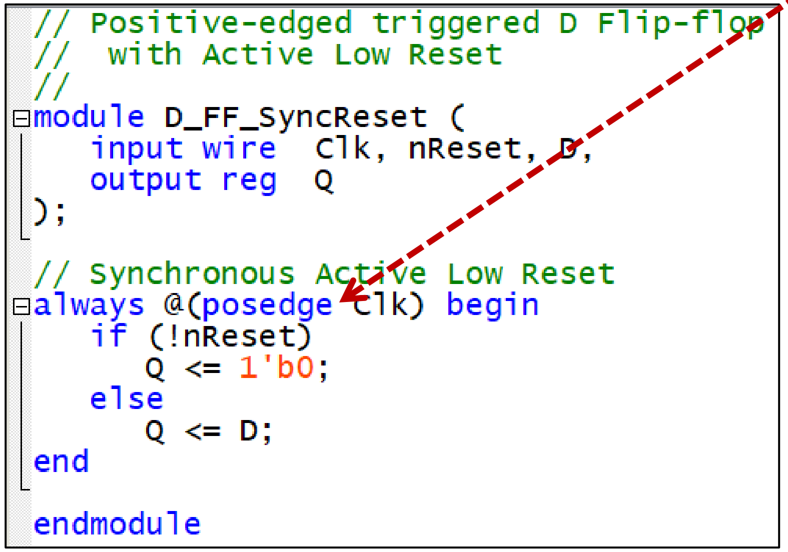
1. (a & b) is evaluated
2. (x | c) is evaluated
3. (1) is assigned to x, (2) is assigned to y *(concurrent assignments)*

Reset

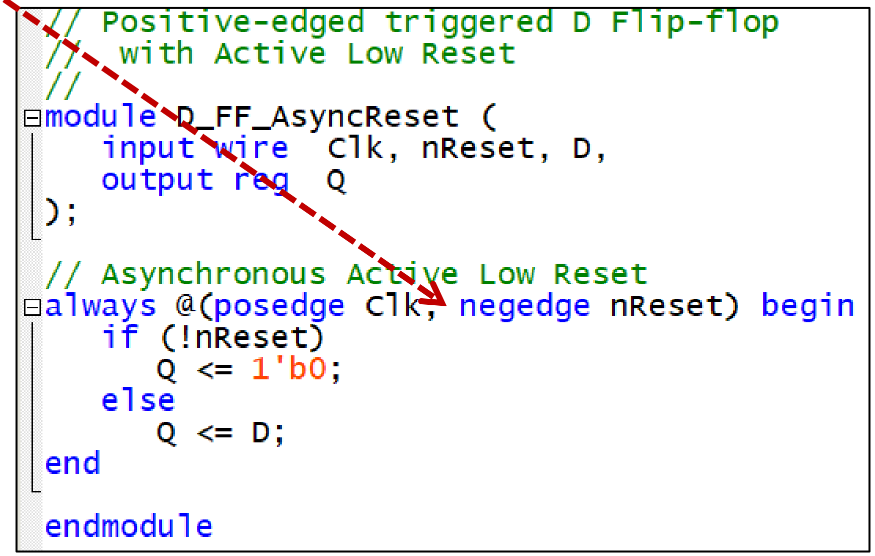
- It is common practice to have a **reset** signal for easy initialization.
- Reset signal is usually low-asserted (active-low) by convention.
- Synthesis tools usually have strict format flip-flops.
- Two types: synchronous and asynchronous resets

Synchronous vs. Asynchronous Reset

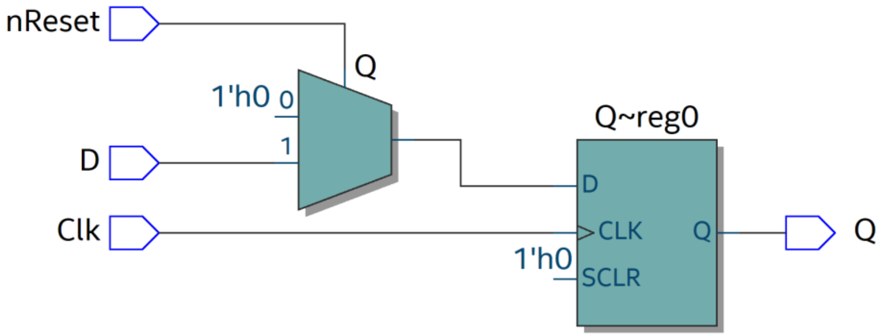
See the difference in the sensitivity lists

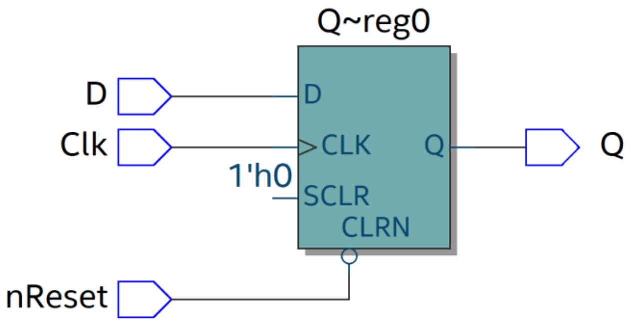


```
// Positive-edged triggered D Flip-flop
//  with Active Low Reset
//
module D_FF_SyncReset (
    input wire  Clk, nReset, D,
    output reg  Q
);

// Synchronous Active Low Reset
always @(posedge Clk) begin
    if (!nReset)
        Q <= 1'b0;
    else
        Q <= D;
end

endmodule
```



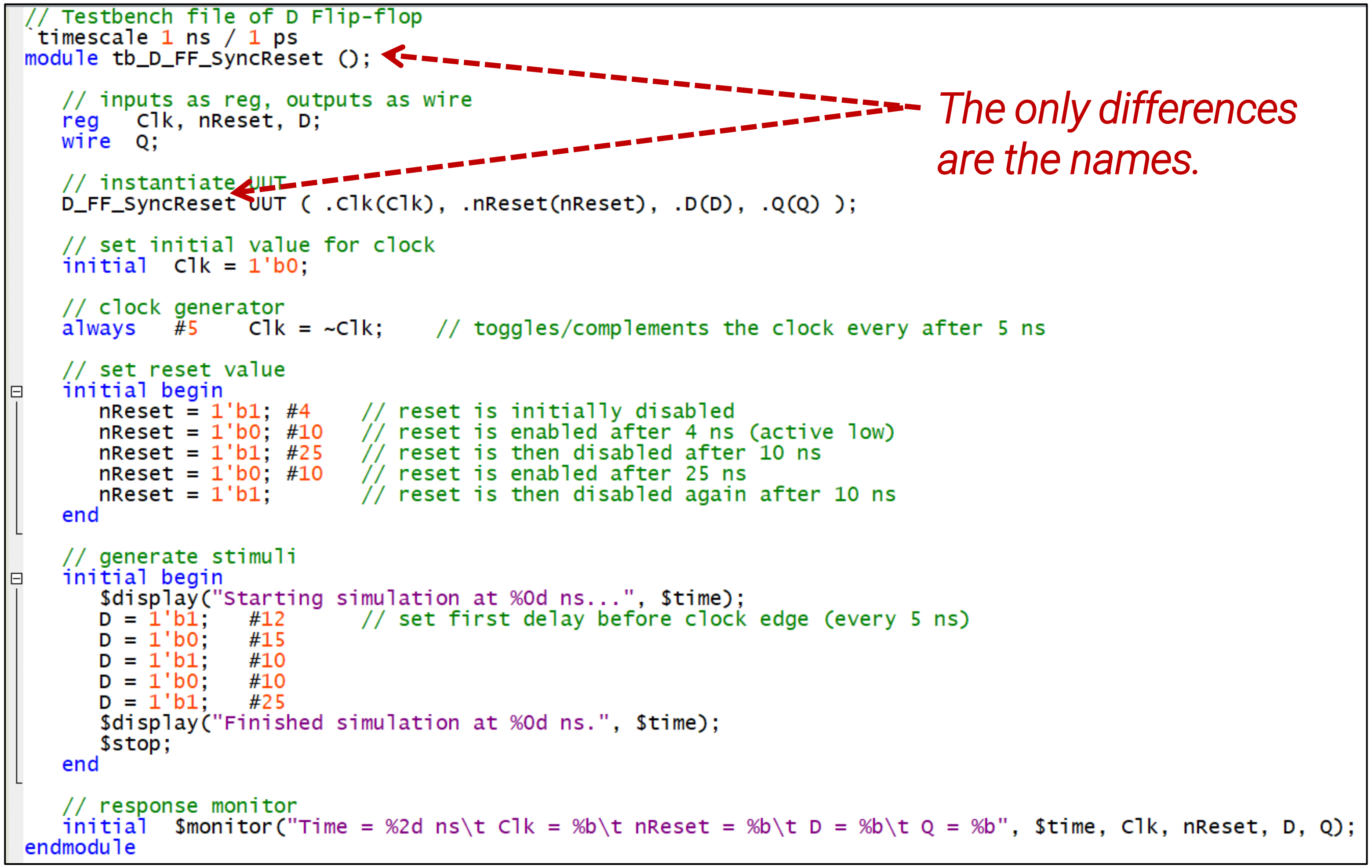
```
// Positive-edged triggered D Flip-flop
//  with Active Low Reset
//
module D_FF_AsyncReset (
    input wire  Clk, nReset, D,
    output reg  Q
);

// Asynchronous Active Low Reset
always @(posedge Clk, negedge nReset) begin
    if (!nReset)
        Q <= 1'b0;
    else
        Q <= D;
end

endmodule
```





If active high Reset: `always @(…, posedge Reset)`

Similar Testbench Files

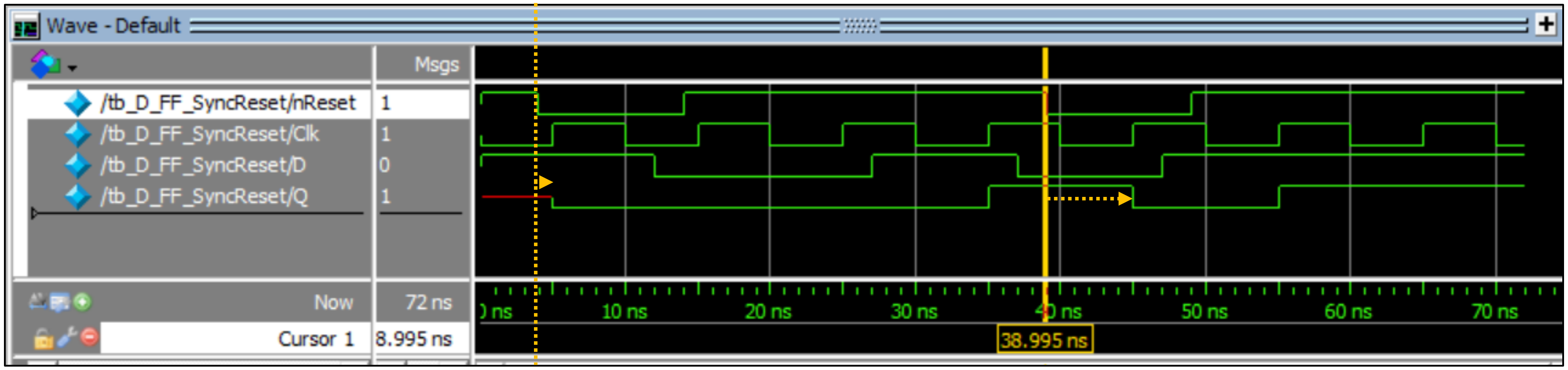


```
// Testbench file of D Flip-flop
`timescale 1 ns / 1 ps
module tb_D_FF_SyncReset ();

   // inputs as reg, outputs as wire
   reg   Clk, nReset, D;
   wire  Q;

   // instantiate UUT
   D_FF_SyncReset UUT ( .Clk(Clk), .nReset(nReset), .D(D), .Q(Q) );

   // set initial value for clock
   initial  Clk = 1'b0;

   // clock generator
   always   #5    Clk = ~Clk;    // toggles/complements the clock every after 5 ns

   // set reset value
   initial begin
      nReset = 1'b1; #4     // reset is initially disabled
      nReset = 1'b0; #10    // reset is enabled after 4 ns (active low)
      nReset = 1'b1; #25    // reset is then disabled after 10 ns
      nReset = 1'b0; #10    // reset is enabled after 25 ns
      nReset = 1'b1;        // reset is then disabled again after 10 ns
   end

   // generate stimuli
   initial begin
      $display("Starting simulation at %0d ns...", $time);
      D = 1'b1;    #12      // set first delay before clock edge (every 5 ns)
      D = 1'b0;    #15
      D = 1'b1;    #10
      D = 1'b0;    #10
      D = 1'b1;    #25
      $display("Finished simulation at %0d ns.", $time);
      $stop;
   end

   // response monitor
   initial  $monitor("Time = %2d ns\t Clk = %b\t nReset = %b\t D = %b\t Q = %b", $time, Clk, nReset, D, Q);
endmodule
```

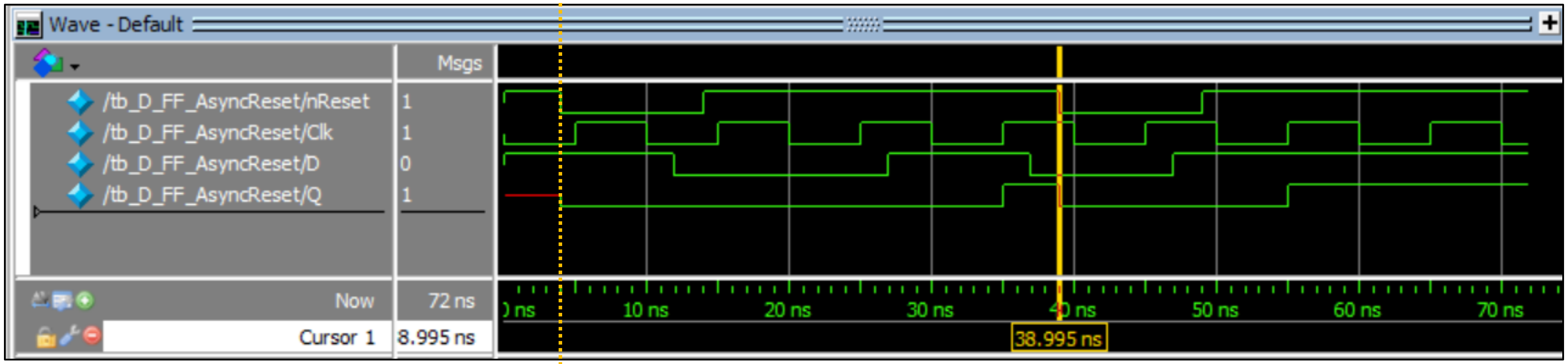
The only differences are the names.

Compare Waveforms

Synchronous (Active Low) Reset



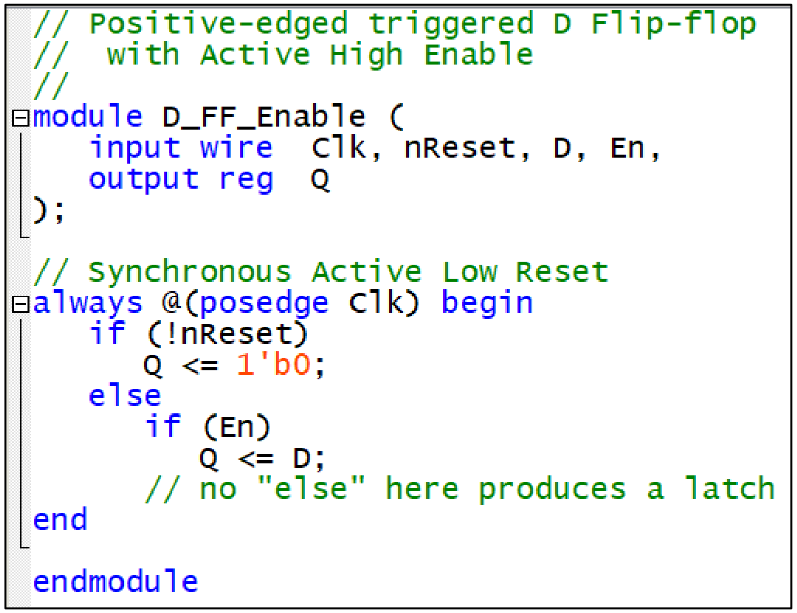
Asynchronous (Active Low) Reset



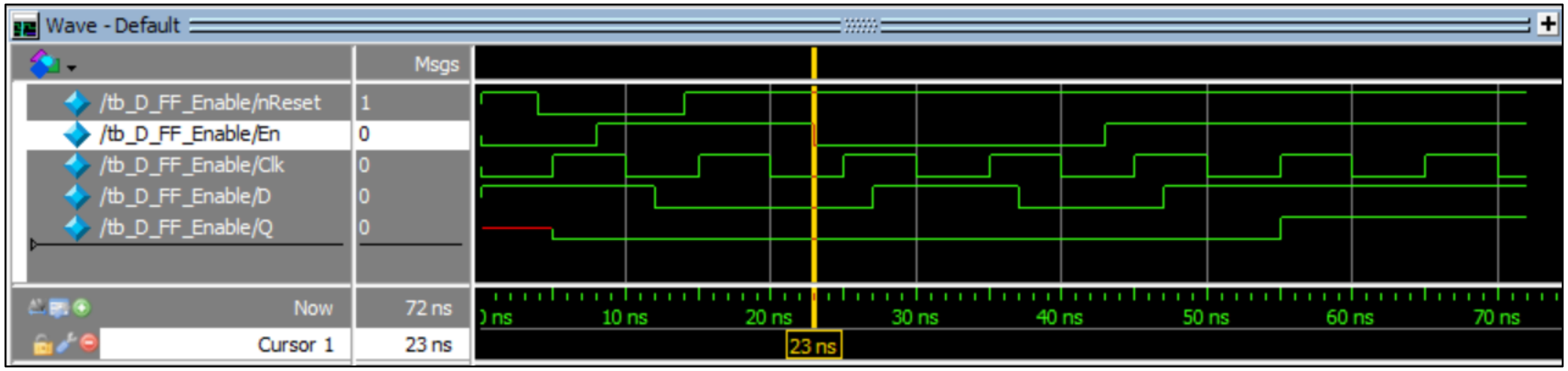
Enable

- Many circuits require a flip-flop or register to hold a value for more than a clock cycle, controlled by an enable signal.

- In Verilog, an incompletely-specified *if-else* and *case* statements produce a latch.



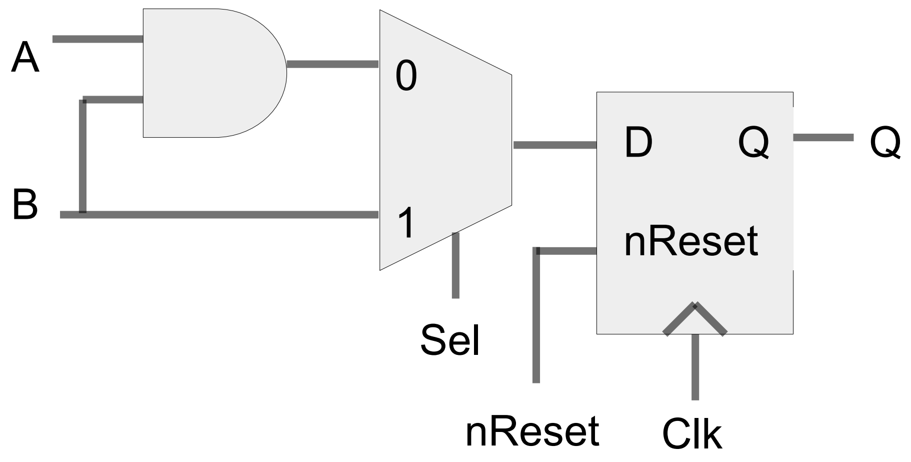
```
// Positive-edged triggered D Flip-flop
//  with Active High Enable
//
module D_FF_Enable (
   input wire  Clk, nReset, D, En,
   output reg  Q
);

// Synchronous Active Low Reset
always @(posedge Clk) begin
   if (!nReset)
      Q <= 1'b0;
   else
      if (En)
         Q <= D;
      // no "else" here produces a latch
end

endmodule
```



	Msgs
/tb_D_FF_Enable/nReset	1
/tb_D_FF_Enable/En	0
/tb_D_FF_Enable/Clk	0
/tb_D_FF_Enable/D	0
/tb_D_FF_Enable/Q	0
Now	72 ns
Cursor 1	23 ns

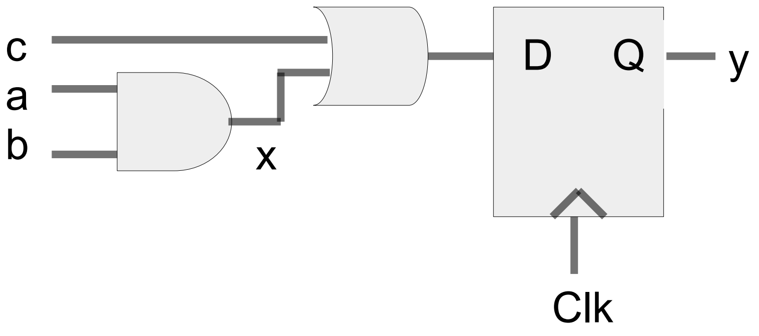
Incorporating Logic

```
// Synchronous Active Low Reset
…
always @(posedge Clk)
begin
   if (!nReset)
        Q <= 1’b0;
   else
        if (Sel)
               Q <= B;
        else
               Q <= A & B;
end
…
```

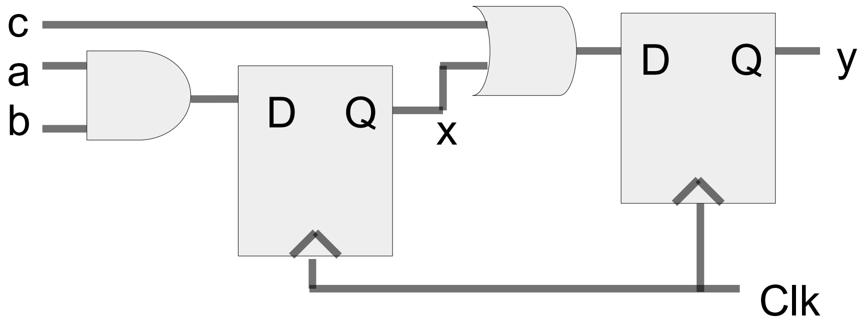


Blocking vs. Non-blocking Assignments

```
// blocking assignments
// positive-edge triggered Clk
reg x, y;
always @(posedge Clk) begin
    x = a & b;
    y = x | c;
end
```

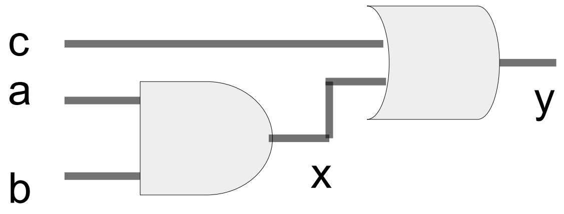


```
// non-blocking assignments
// positive-edge triggered Clk
reg y;
always @(posedge Clk) begin
    x <= a & b;
    y <= x | c;
end
```

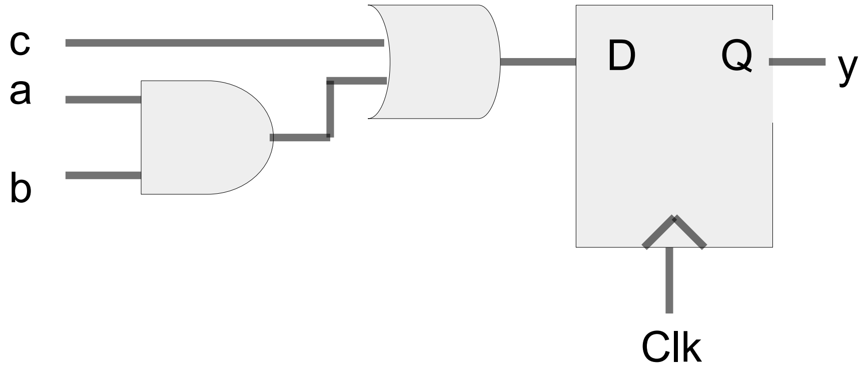


Combinational vs. Sequential Logic

```
// combinational logic uses
// blocking assignments
reg x, y;
always @(*) begin
   x = a & b;
   y = x | c;
end
```



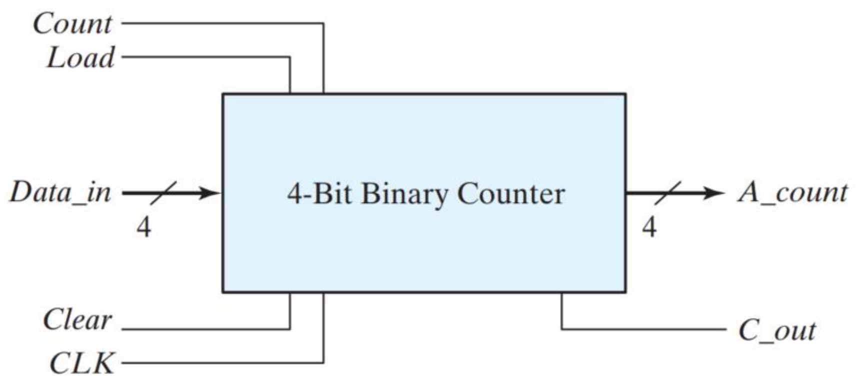
```
// sequential logic uses
// non-blocking assignments
reg y;
always @(posedge Clk) begin
   y <= (a & b) | c;
end
```



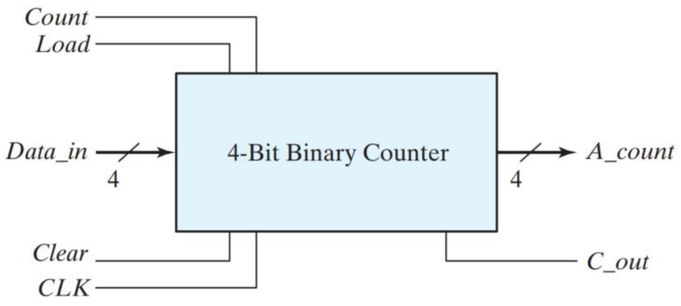
Example 3: 4-bit Binary Counter

- Verilog behavioral description of a 4-bit binary counter with parallel load:

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change



Example 3: 4-bit Binary Counter



Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	Count next binary state
1	↑	0	0	No change

```
module Binary_Counter_4_Par_Load (
 output reg [3: 0]              A_count,      // Data output
 output                         C_out,        // Output carry
 input [3: 0]                   Data_in,      // Data input
 input                          Count,        // Active high to count
                                Load,         // Active high to load
                                CLK,          // Positive-edge sensitive
                                Clear_b       // Active low
);
assign C_out = Count && (~Load) && (A_count == 4'b1111);
always @ (posedge CLK, negedge Clear_b)
  if (~Clear_b)                 A_count <= 4'b0000;
  else if (Load)                A_count <= Data_in;
  else if (Count)               A_count <= A_count + 1'b1;
  else                          A_count <= A_count;  // redundant statement
endmodule
```

Synthesis of Sequential Elements

- For most synthesis tools, an ***always*** block with a **posedge** or **negedge** in the sensitivity list will create a **flip-flop** *(clock automatically set by **posedge** or **negedge**).*

Guidelines for Good Synthesis

- Use **non-blocking assignments (<=)** to model sequential circuits.
  - It may be fine to use blocking statements, but it is NOT recommended if you do not know what you are doing.
  - The hardware generated by non-blocking assignments is **independent** of the ordering of the assignments.

- **Do not mix blocking** and **non-blocking assignments** in the same *always* block.

- Do not make assignments to the **same variable from more than one** *always* block.
  - It is a Verilog race condition, even when using non-blocking assignments.

End of Unit 6