

---

# Verilog – Sequential Logic

Verilog for Synthesis – Rev C (module 3 and 4)

# Latches and Flip-Flops

---

- Implemented by using signals in always statements with edge-triggered clk
- Necessary flip-flops are inferred by the synthesis tool.
- Edge-triggered flip-flop
- Reset
  - Asynchronous
  - Synchronous
- Counters
- Shift Registers
- Finite State Machines

# Concurrent statements

---

- Verilog
  - `always` statement
  - Continuous assignment - `assign`

# Sequential Statements

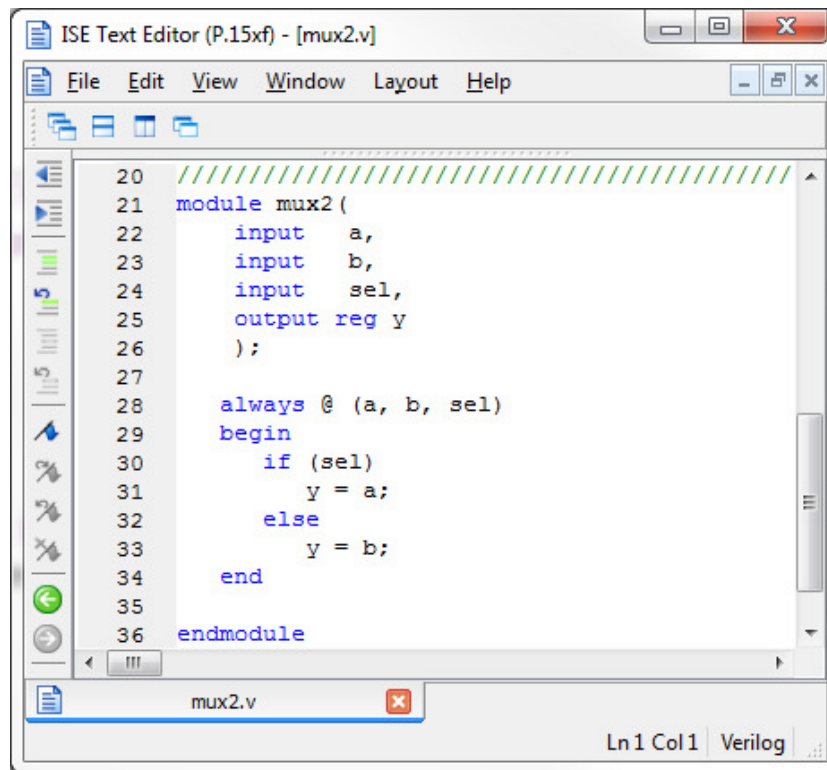
---

- Verilog
  - reside in an `always` statement
  - `if` statements (no `endif`)
  - `case` statements (`endcase`)
  - `for`, `repeat while` loop statements
  - Note: use `begin` and `end` to block sequential statements
  - Sequential statements can be used in an `always` statement to describe both sequential and combinational logic
    - use non-blocking (`<=`) assignment for sequential logic
    - use blocking (`=`) assignment for combinational logic

# Combinational Logic - review

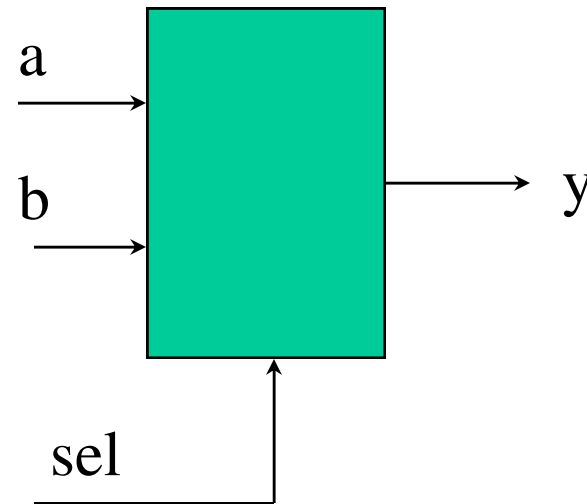
---

- Using always statement
  - All input signals specified in sensitivity list
- All conditions evaluated (LUTs used, no-flip-flops)

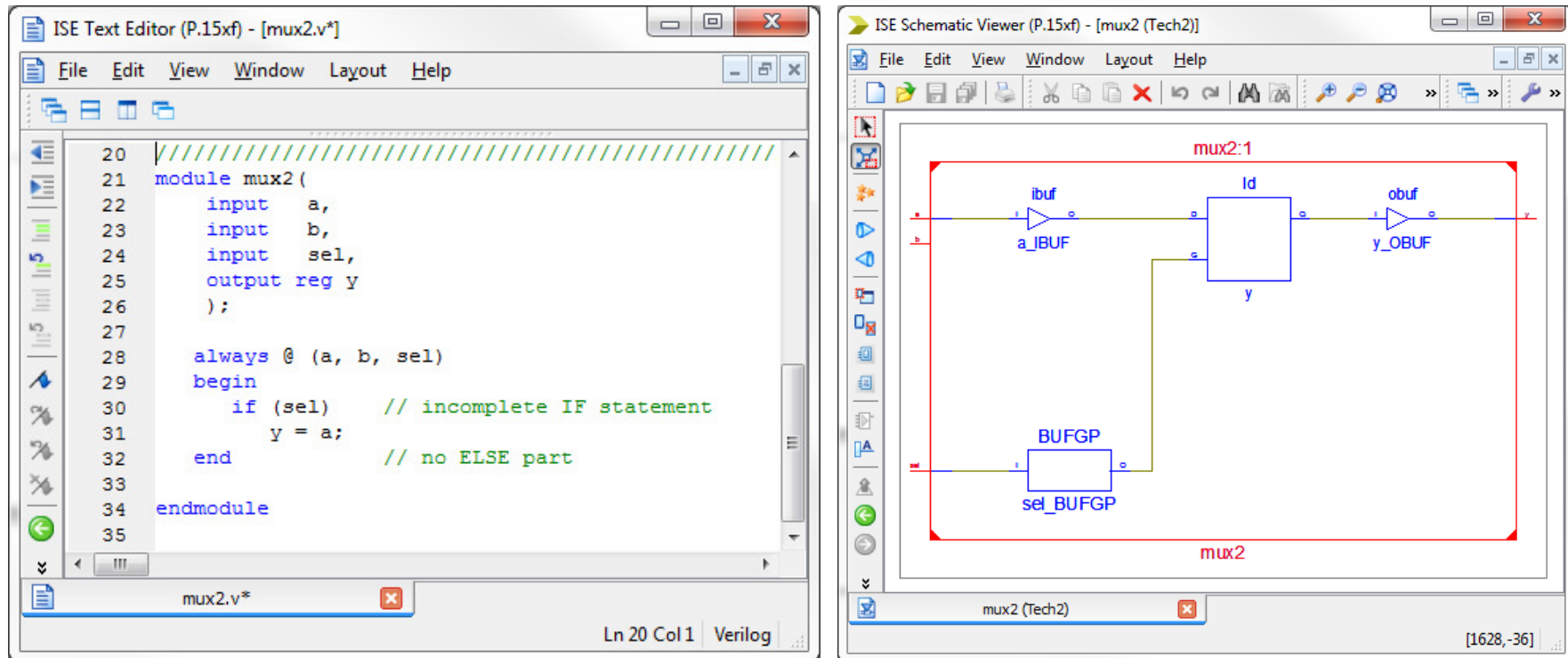


```
20 //////////////////////////////////////////////////
21 module mux2 (
22     input  a,
23     input  b,
24     input  sel,
25     output reg y
26 );
27
28 always @ (a, b, sel)
29 begin
30     if (sel)
31         y = a;
32     else
33         y = b;
34 end
35
36 endmodule
```

The screenshot shows a Verilog module named `mux2` in the ISE Text Editor. The code defines a 2-to-1 multiplexer with inputs `a`, `b`, and `sel`, and an output `y`. The output is a registered signal (`reg y`). The logic is implemented using an `always` statement with a sensitivity list containing `a`, `b`, and `sel`. Inside the `always` block, an `if` statement selects between `a` and `b` based on the value of `sel`. The status bar at the bottom indicates the file is `mux2.v` and the cursor is at line 1, column 1 in Verilog mode.



# Incomplete IF statement – causes Latches

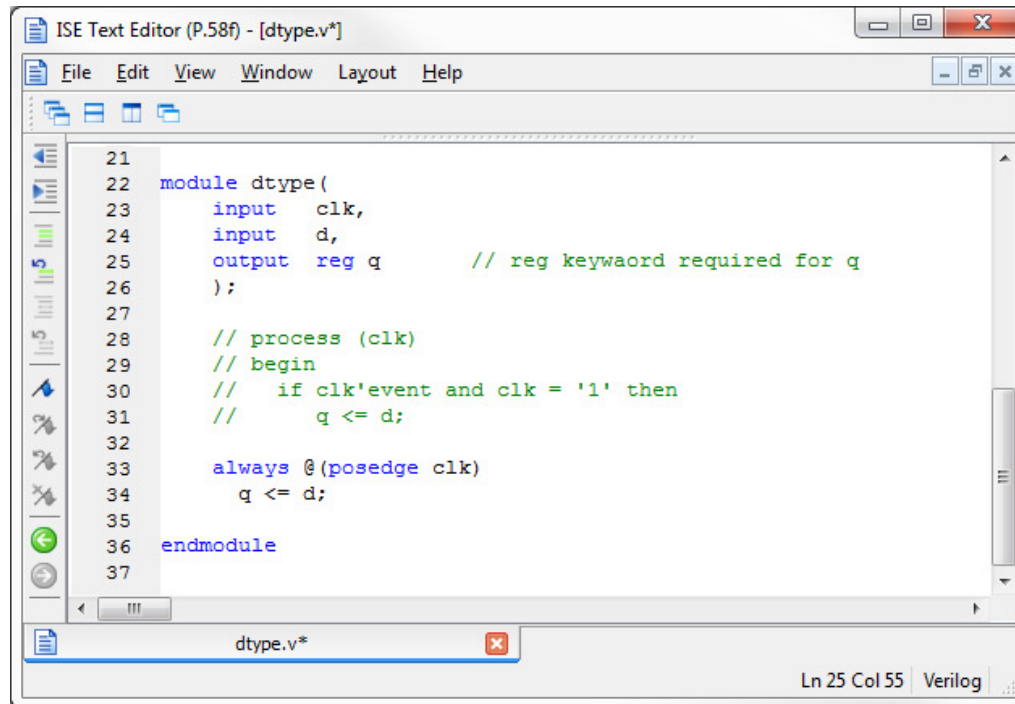


**WARNING:**Xst:647 - Input <b> is never used. This port will be preserved and left unconnected if it belongs to a top-level block or it belongs to a sub-block and the hierarchy of this sub-block is preserved.

**WARNING:**Xst:737 - Found 1-bit latch for signal <y>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.

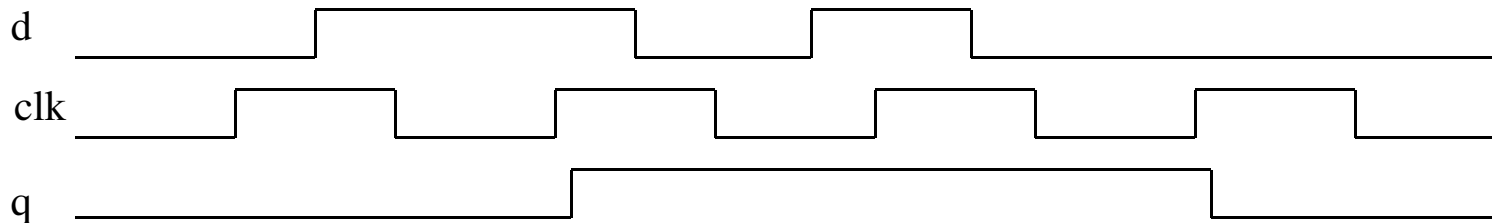
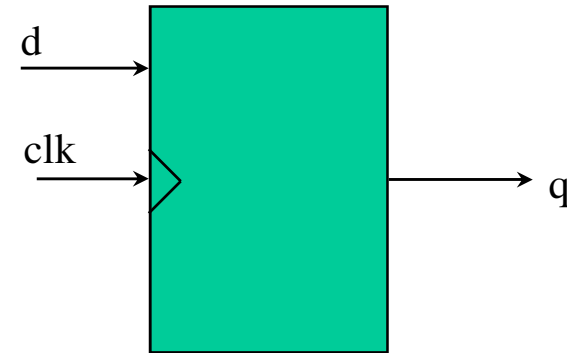
# Flip-flops in Verilog

- Always inferred using edge-triggered `always` statement



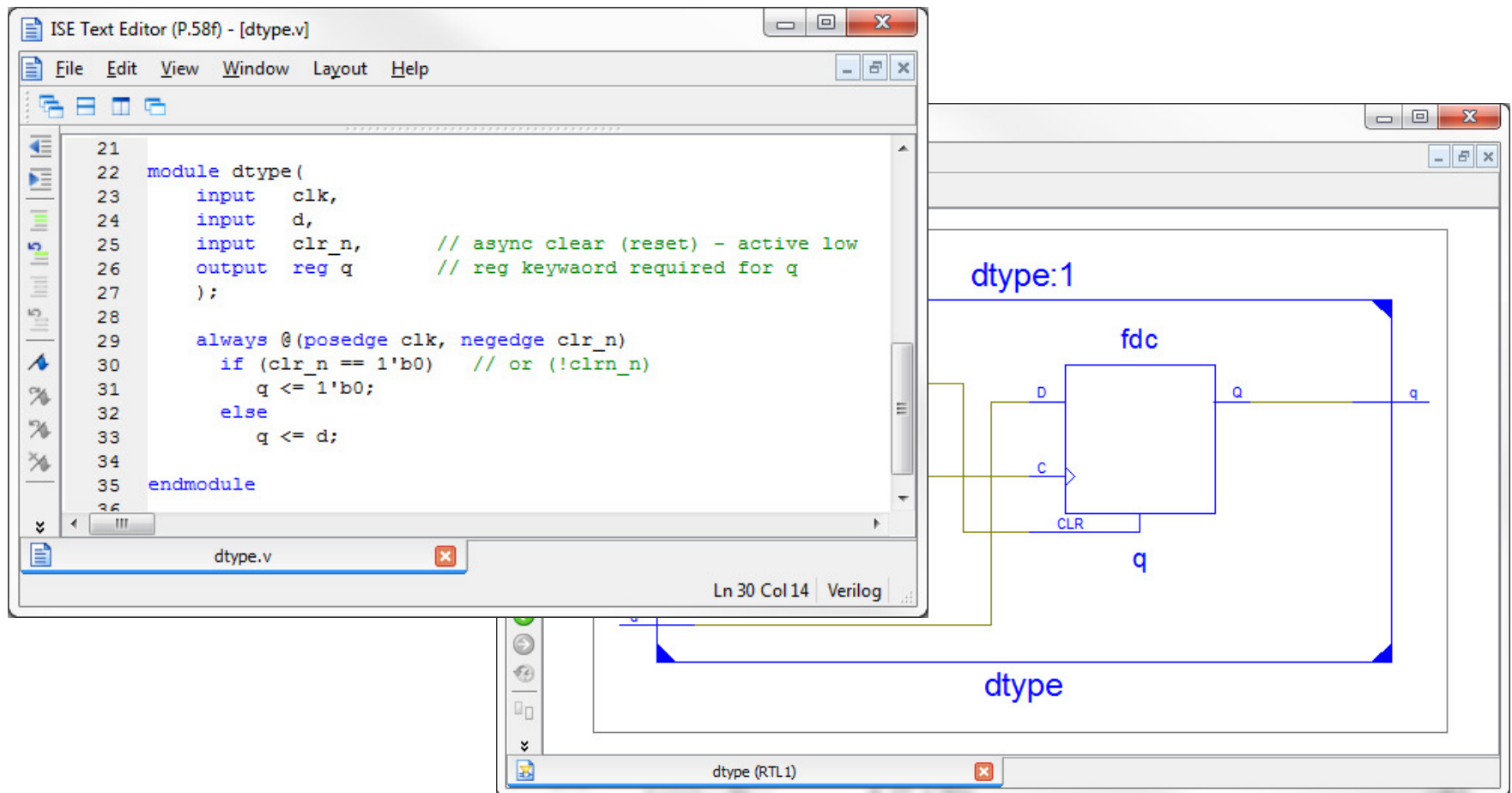
```
21
22 module dtype(
23     input  clk,
24     input  d,
25     output reg q    // reg keyword required for q
26 );
27
28 // process (clk)
29 // begin
30 //   if clk'event and clk = '1' then
31 //     q <= d;
32
33 always @(posedge clk)
34     q <= d;
35
36 endmodule
37
```

Ln 25 Col 55 | Verilog



# Flip-flops in Verilog (with async)

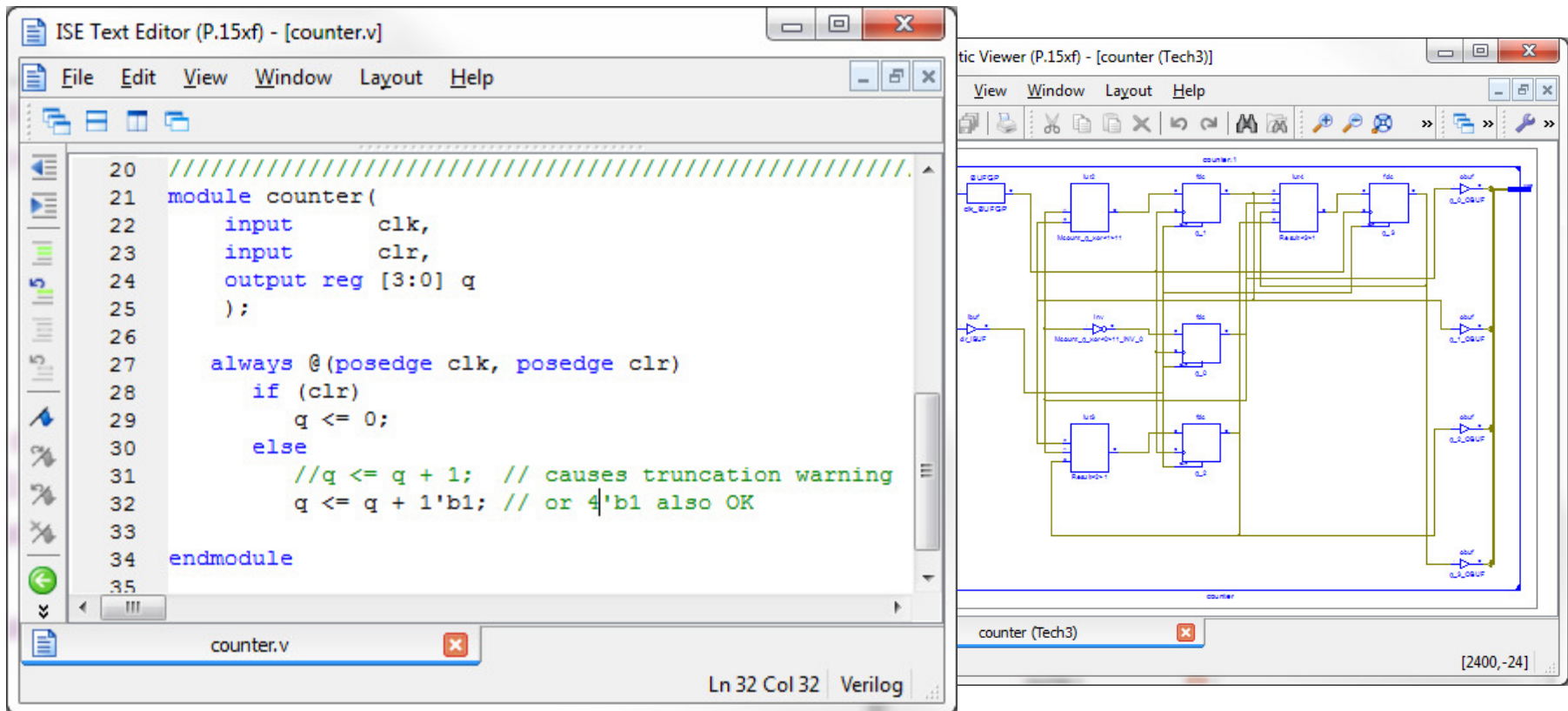
- Add async signals to sensitivity list





# Counters in Verilog

- Just extension of D type
- This example has async, active-high clear (reset)



# Counter - synthesis

---

Elaborating module <counter>.

**WARNING:**HDLCompiler:413 - "C:\ece3829\counter\counter.v" Line 31: Result of 5-bit expression is truncated to fit in 4-bit target.

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <counter>.
  Related source file is "C:\ece3829\counter\counter.v".
  Found 4-bit register for signal <q>.
  Found 4-bit adder for signal <q[3]_GND_1_o_add_0_OUT> created at line 31.
  Summary:
    inferred   1 Adder/Subtractor(s).
    inferred   4 D-type flip-flop(s).
Unit <counter> synthesized.

Timing Summary: (Note - before PAR)
-----
Speed Grade: -3

Minimum period: 2.048ns (Maximum Frequency: 488.317MHz)
Minimum input arrival time before clock: 2.335ns
Maximum output required time after clock: 3.732ns
Maximum combinational path delay: No path found

=====
```

# Counters in Verilog (cont'd)

- With terminal count

The screenshot displays the Xilinx ISE environment. The ISE Text Editor (P.58f) - [dtype.v] window shows the following Verilog code:

```
21
22 module counter2(
23     input          clk,
24     input          clr, // async clear (reset)
25     output reg [3:0] q // reg keyword required for q
26 );
27
28 always @(posedge clk, posedge clr)
29     if (clr)
30         q <= 0;
31     else if (q == 13) // can read output
32         q <= 4'h0;
33     else
34         q <= q + 1'b1; // to stop truncation warning
35
36 endmodule
```

The background window shows a logic diagram of the counter2 module. It includes a 4-bit register (q[3:0]), a 4-bit output (q[3:0]), and a 4-bit input (q[3:0]). The output is connected to a 4-bit bus (q[3:0]). The input is connected to a 4-bit bus (q[3:0]). The output is connected to a 4-bit bus (q[3:0]).

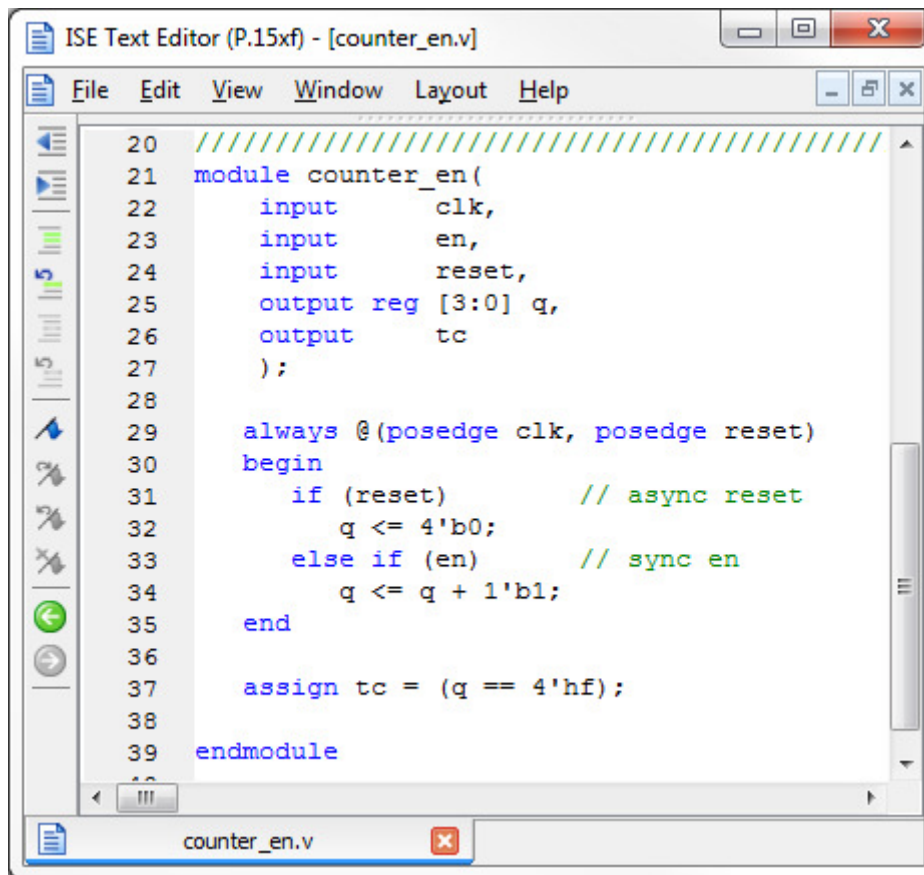
The timing diagram at the bottom shows the clock signal (clk) and the output signal (q[3:0]). The clock signal is a square wave with a period of 1 ns. The output signal is a 4-bit bus that increments by 1 on each clock edge. The output signal is shown for the first 1444 ps, with the cursor at 1426 ps.

# Blocking and non-blocking assignment

---

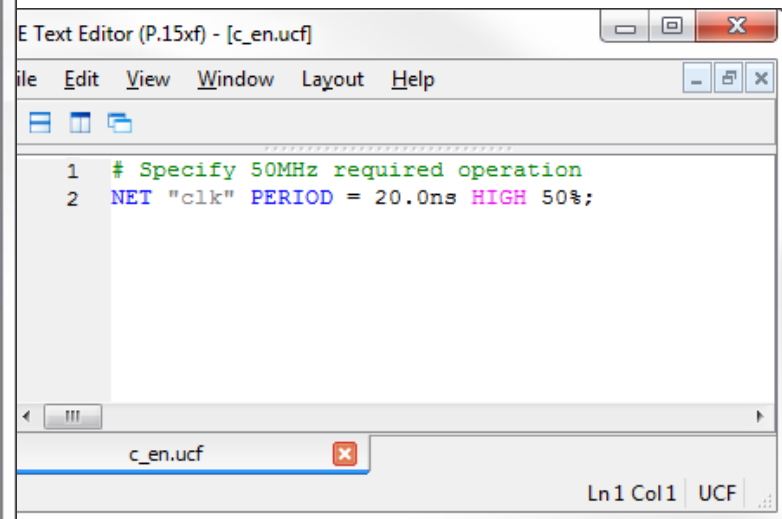
- To ensure correct synthesis and simulation results:
- Combinational logic
  - Use blocking assignment `=` statements in `always` block
- Sequential logic
  - Use non blocking assignment `<=` statements in `always` block
  - Can only be used on `reg` types
    - Can only be used in an `initial` or `always` procedural blocks
  - Can not be used in continuous assignments

# Counter with Clock Enable (and TC)



The screenshot shows the ISE Text Editor window titled "ISE Text Editor (P.15xf) - [counter\_en.v]". The code is as follows:

```
20 //////////////////////////////////////////////////
21 module counter_en(
22     input    clk,
23     input    en,
24     input    reset,
25     output reg [3:0] q,
26     output    tc
27 );
28
29 always @(posedge clk, posedge reset)
30 begin
31     if (reset)          // async reset
32         q <= 4'b0;
33     else if (en)        // sync en
34         q <= q + 1'b1;
35 end
36
37 assign tc = (q == 4'hf);
38
39 endmodule
```

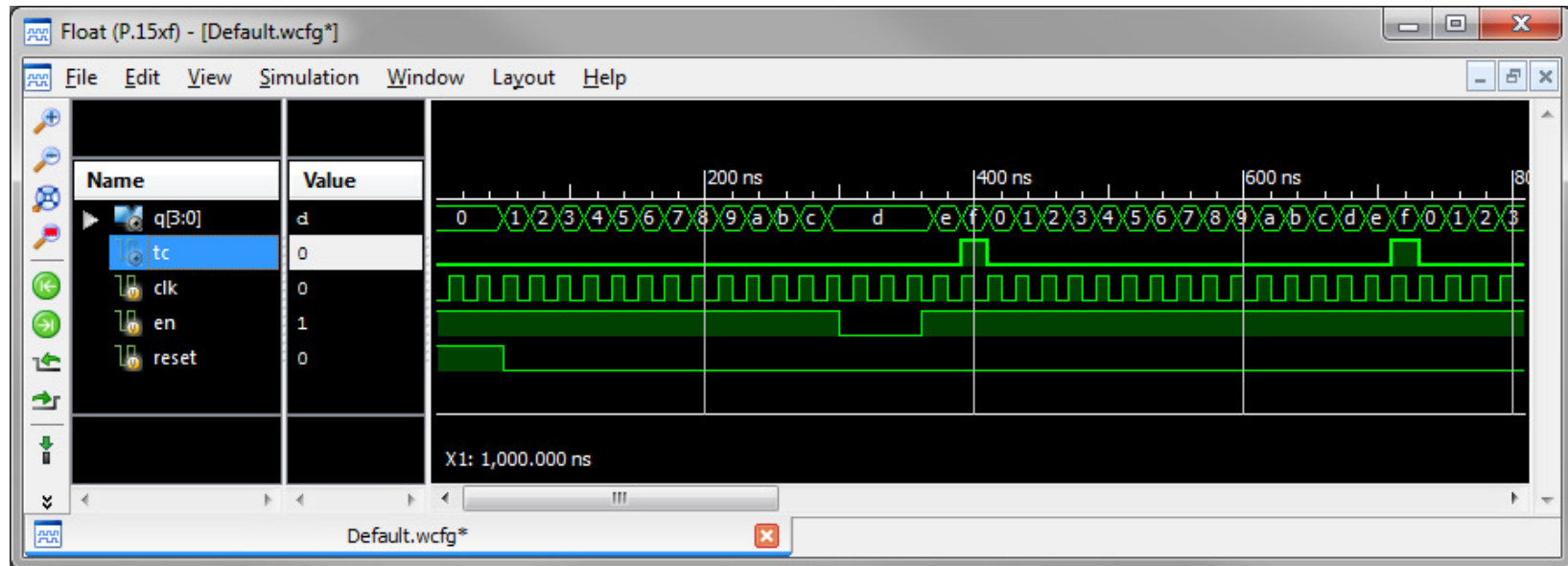


The screenshot shows the ISE Text Editor window titled "E Text Editor (P.15xf) - [c\_en.ucf]". The configuration is as follows:

```
1 # Specify 50MHz required operation
2 NET "clk" PERIOD = 20.0ns HIGH 50%;
```



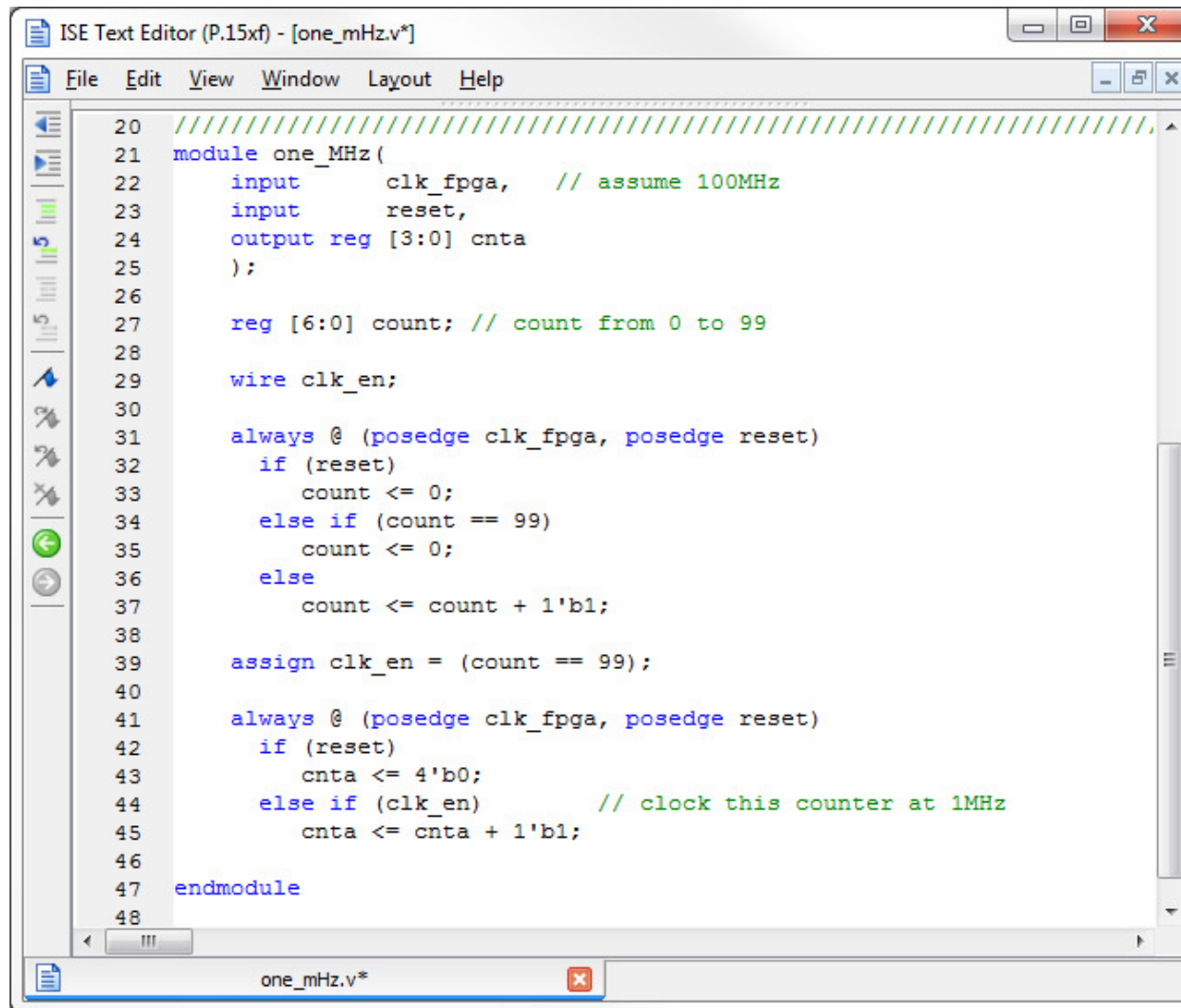
# Simulation and Implementation



Constraint	Check	Worst Case	Best Case	Timing	Timing
		Slack	Achievable	Errors	Score
NET "clk_BUFGP/IBUFG" PERIOD = 20 ns HIGH 50%	SETUP	18.626ns	1.374ns	0	0
	HOLD	0.444ns		0	0
	MINPERIOD	18.270ns	1.730ns	0	0

All constraints were met.

# Creating slower clocks

The image shows a screenshot of the ISE Text Editor window. The title bar reads "ISE Text Editor (P.15xf) - [one\_mHz.v\*]". The menu bar includes "File", "Edit", "View", "Window", "Layout", and "Help". The code is written in Verilog and is as follows:

```
20 //////////////////////////////////////////////////
21 module one_MHz(
22     input    clk_fpga,    // assume 100MHz
23     input    reset,
24     output reg [3:0] cnta
25 );
26
27     reg [6:0] count; // count from 0 to 99
28
29     wire clk_en;
30
31     always @ (posedge clk_fpga, posedge reset)
32         if (reset)
33             count <= 0;
34         else if (count == 99)
35             count <= 0;
36         else
37             count <= count + 1'b1;
38
39     assign clk_en = (count == 99);
40
41     always @ (posedge clk_fpga, posedge reset)
42         if (reset)
43             cnta <= 4'b0;
44         else if (clk_en) // clock this counter at 1MHz
45             cnta <= cnta + 1'b1;
46
47 endmodule
48
```

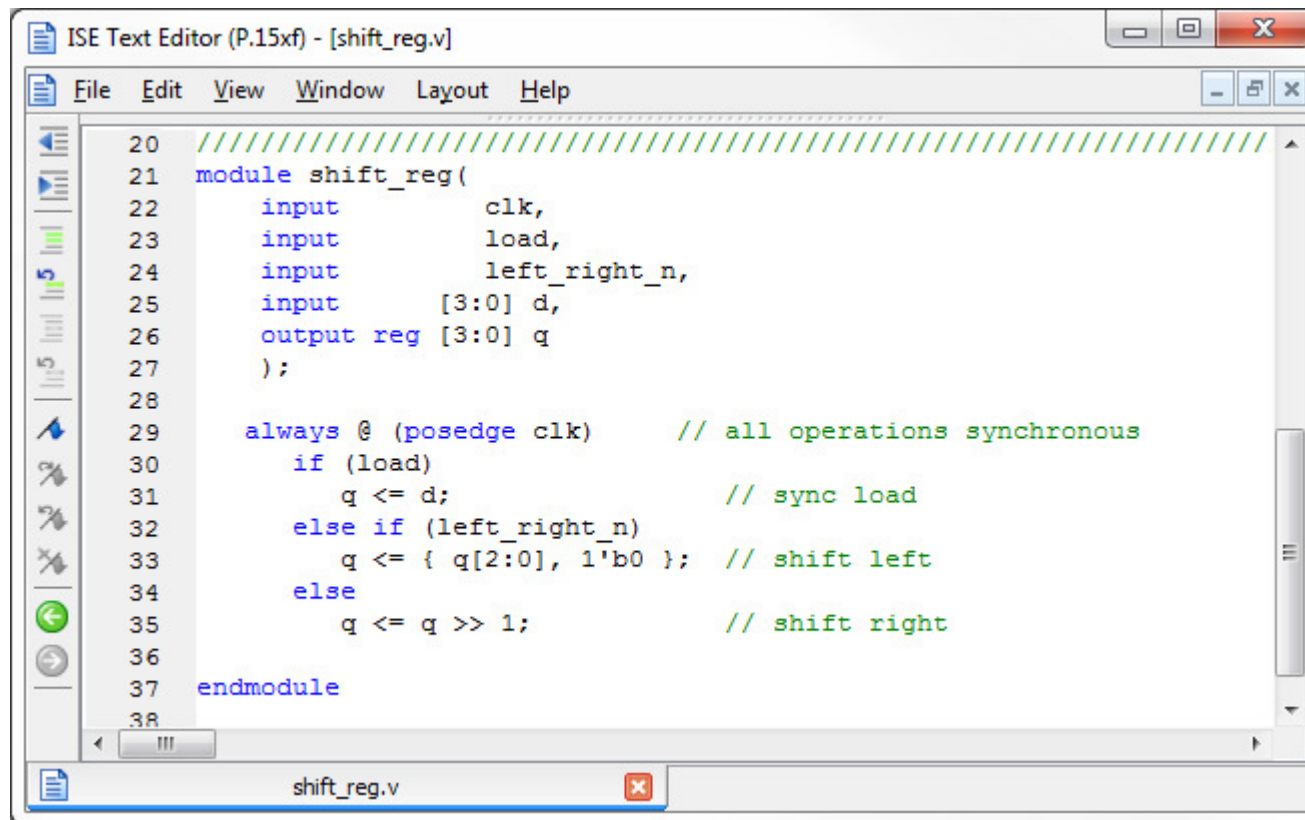
The status bar at the bottom shows the file name "one\_mHz.v\*" and a red 'x' icon.



# Shift Register

---

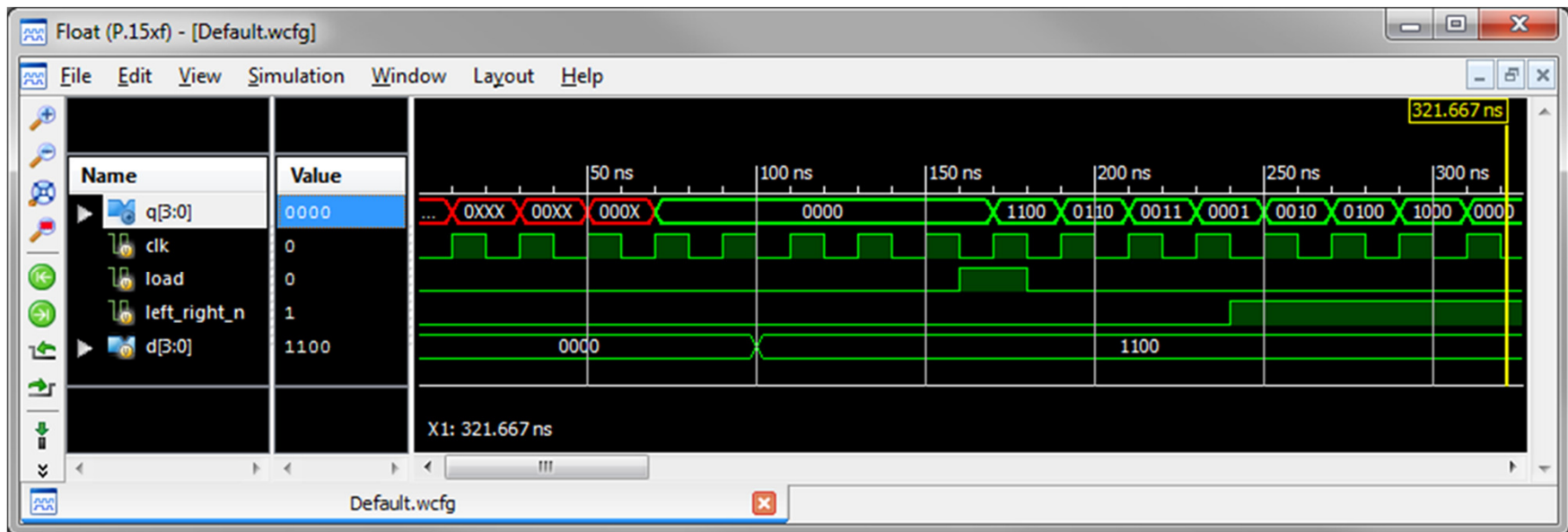
- 4-bit with Parallel Load
- Shift left and shift right



```
20 //////////////////////////////////////////////////
21 module shift_reg(
22     input      clk,
23     input      load,
24     input      left_right_n,
25     input [3:0] d,
26     output reg [3:0] q
27 );
28
29     always @ (posedge clk)    // all operations synchronous
30     if (load)
31         q <= d;                // sync load
32     else if (left_right_n)
33         q <= { q[2:0], 1'b0 }; // shift left
34     else
35         q <= q >> 1;           // shift right
36
37 endmodule
38
```

# Shift Register - Simulation

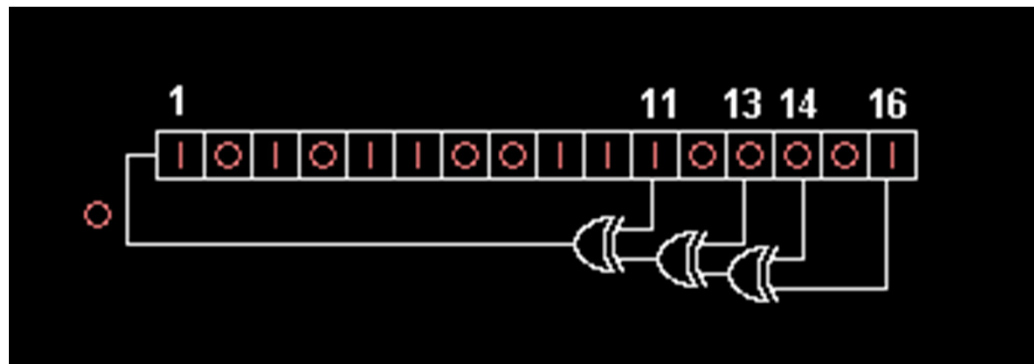
- Note: 'X' denotes unknown
- All operations synchronous
  - change on positive edge of clock



# LFSR

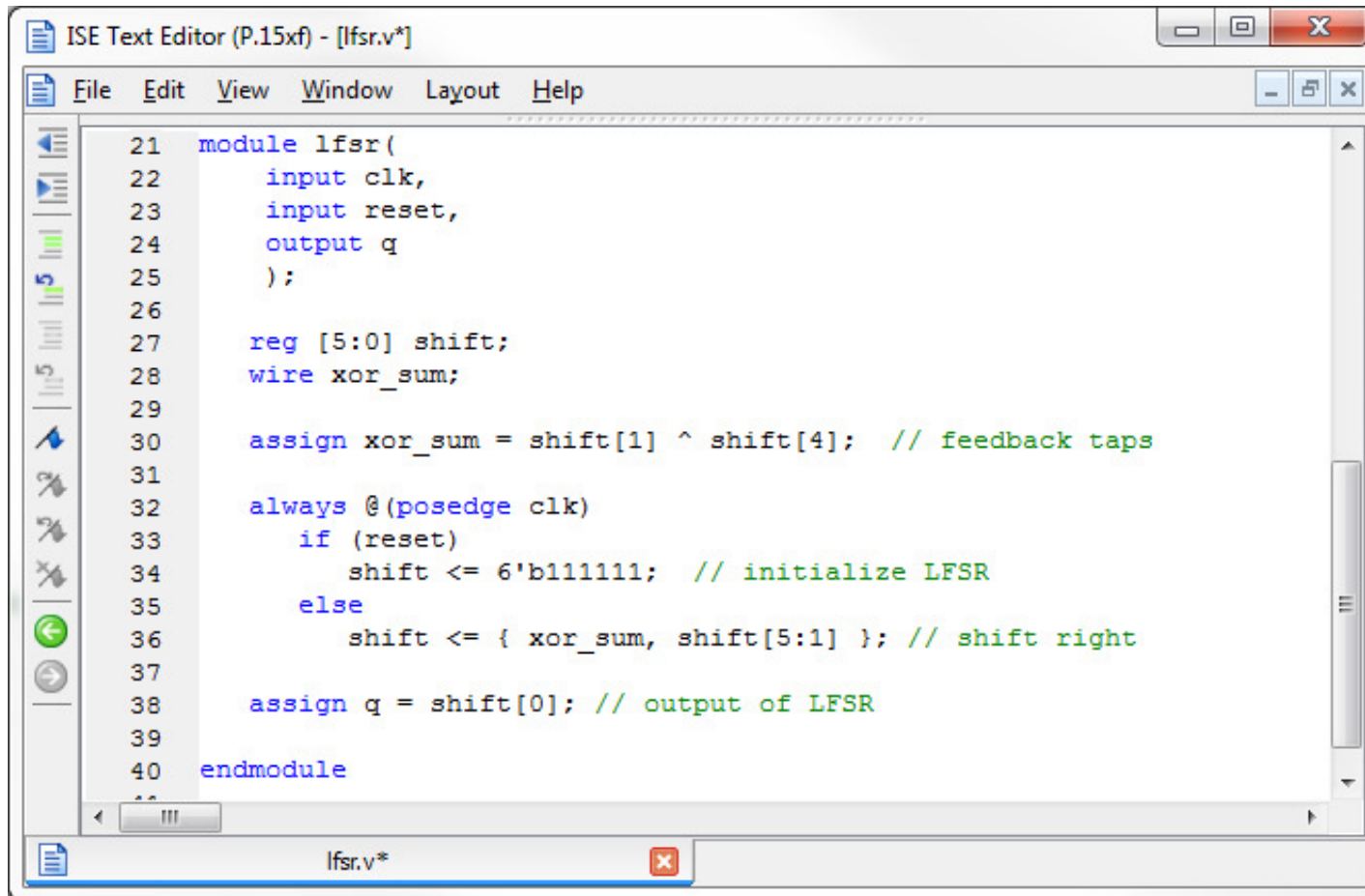
---

- Input bit driven by XOR of some bits (feedback taps) of shift reg value
- Initial value called seed
- Eventually enters repeating cycle
- n-bit LSR has  $2^n - 1$  states (0000 missing state)
- Sequence can appear random – generate PRN



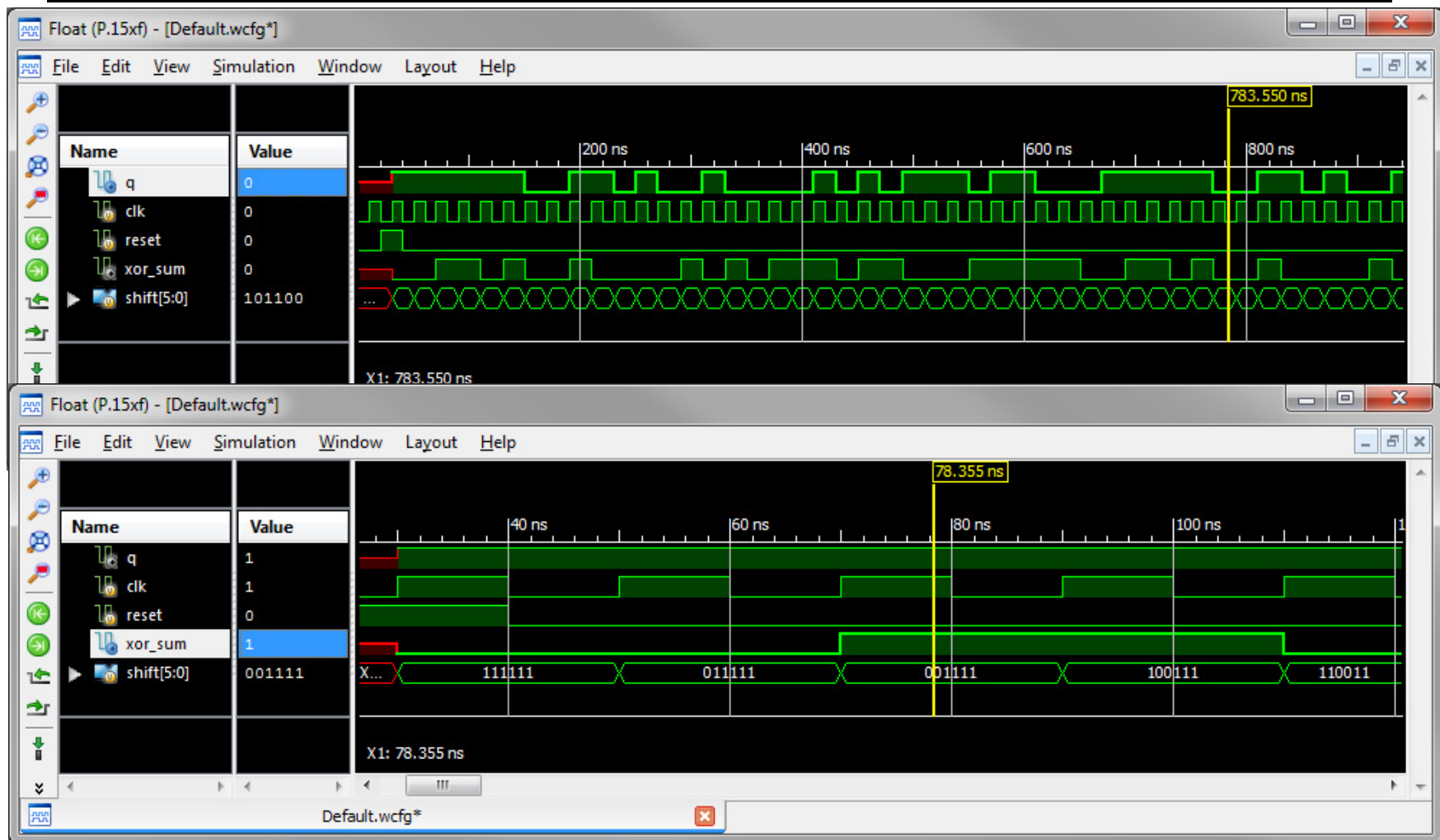
# Example 6-bit LFSR

---

A screenshot of the ISE Text Editor window titled "ISE Text Editor (P.15xf) - [lfsr.v\*]". The window displays Verilog code for a 6-bit LFSR module. The code includes inputs for clock (clk) and reset, and an output q. It defines a 6-bit shift register and an XOR sum. The feedback taps are at positions 1 and 4. The LFSR is initialized to 6'b111111. The shift register is updated on the positive edge of the clock, shifting right and inserting the XOR sum at the most significant bit. The output q is assigned the value of the least significant bit of the shift register.

```
21 module lfsr(  
22     input clk,  
23     input reset,  
24     output q  
25 );  
26  
27 reg [5:0] shift;  
28 wire xor_sum;  
29  
30 assign xor_sum = shift[1] ^ shift[4]; // feedback taps  
31  
32 always @(posedge clk)  
33     if (reset)  
34         shift <= 6'b111111; // initialize LFSR  
35     else  
36         shift <= { xor_sum, shift[5:1] }; // shift right  
37  
38 assign q = shift[0]; // output of LFSR  
39  
40 endmodule
```

# LFSR - Simulation



# Design Review

---

- Combinational Logic (Outputs only dependent on inputs)
  - Implemented with LUTs
  - Can be described using Assign statement or Always Statement
    - (Use Assign for simpler logic)
    - Always statement rules
      - Sensitivity list must include all inputs
      - Only use blocking assignment '='
- Sequential Logic (Outputs depend on inputs and past events – has clock)
  - Implemented with LUTs and Flip-Flops
  - Will always have a clock edge
  - Can only be described using an Always Statement
    - Sensitivity list must only include clock edge and any async signals
    - Only use non-blocking assignment '<='
    - All signals used in always statement will result in flip-flops
- Note: LHS signals in always statements must be of type 'reg'

# State Machines

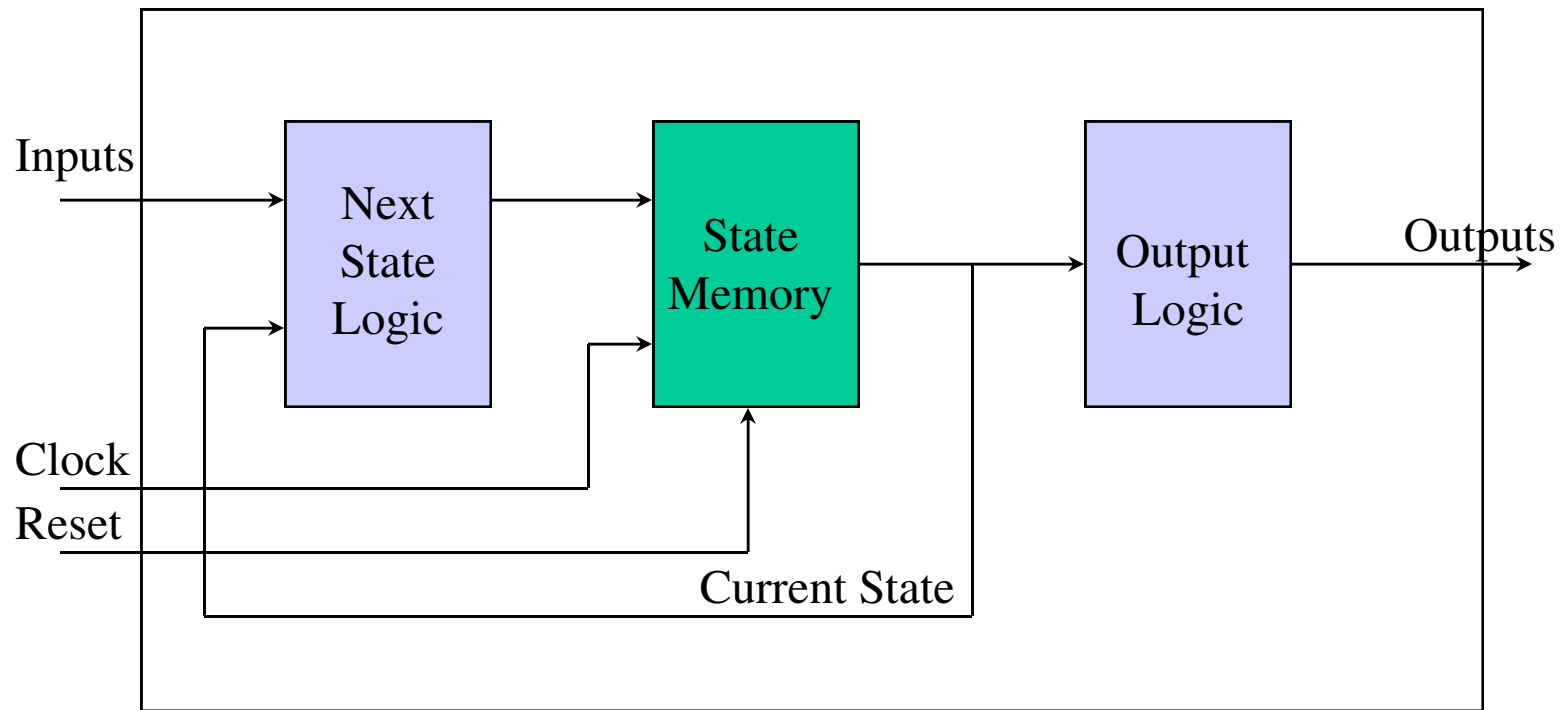
---

- A common form of sequential logic circuit
  - relatively easy to design and analyze
- “Clocked Synchronous State Machine”
  - clocked - storage elements (flip-flops) have clock input
  - synchronous - flip-flops have common clock signal
  - only changes state on clock transition (edge)
- Use a standard coding convention
  - enumerated type for states
  - Always statement (state memory and next state logic)
- Need to understand state encoding (optimal design)
  - one-hot
  - binary
  - other

# State Machines

---

- Block Diagram - Moore Machine
  - Outputs determined by current state





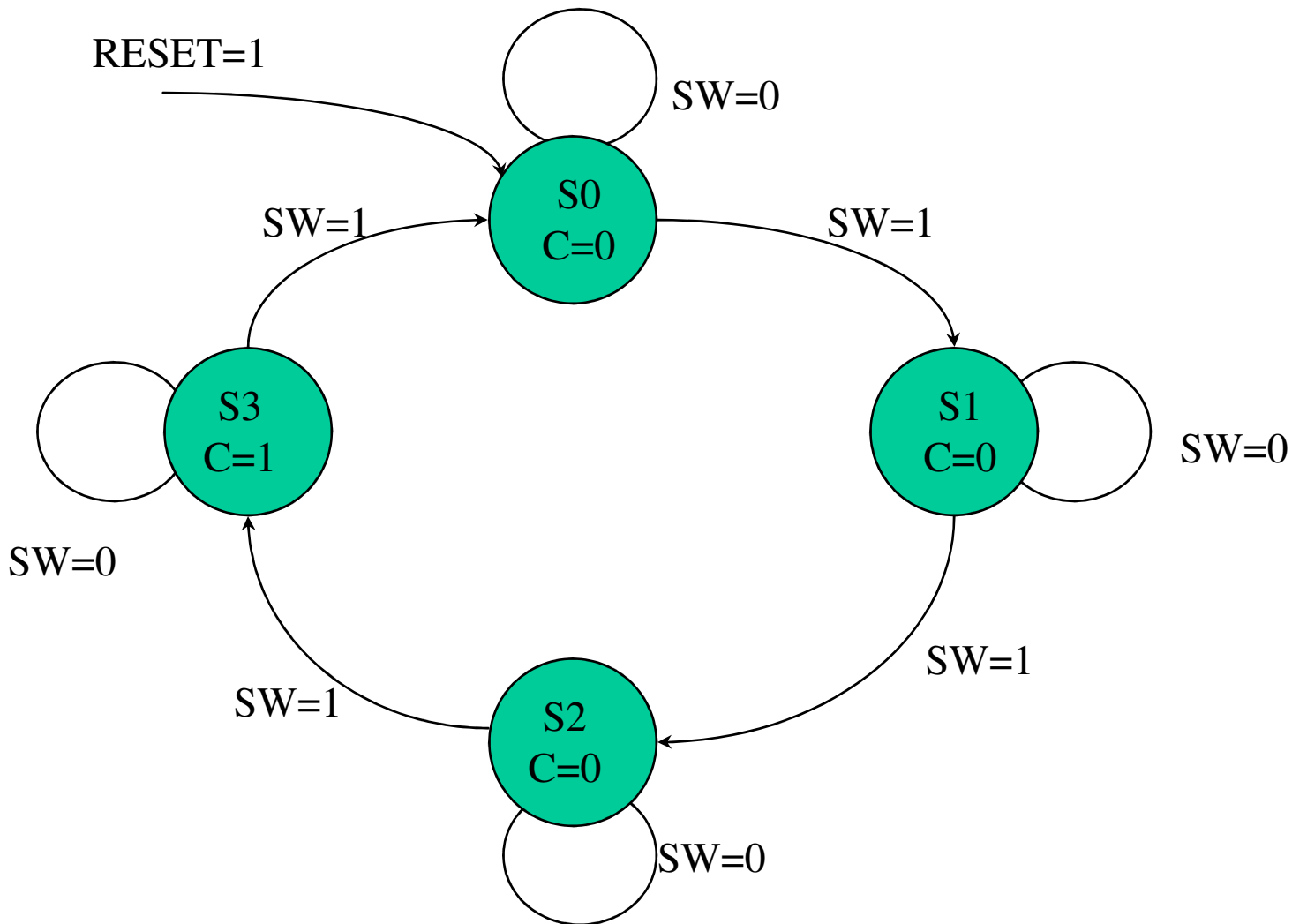
## State Machine (cont'd)

---

- Current state determined by state memory (flip-flops)
- Outputs are decoded from the value of the current state
  - combinational logic
- Next state is determined by current state and inputs at time of next triggering clock edge.

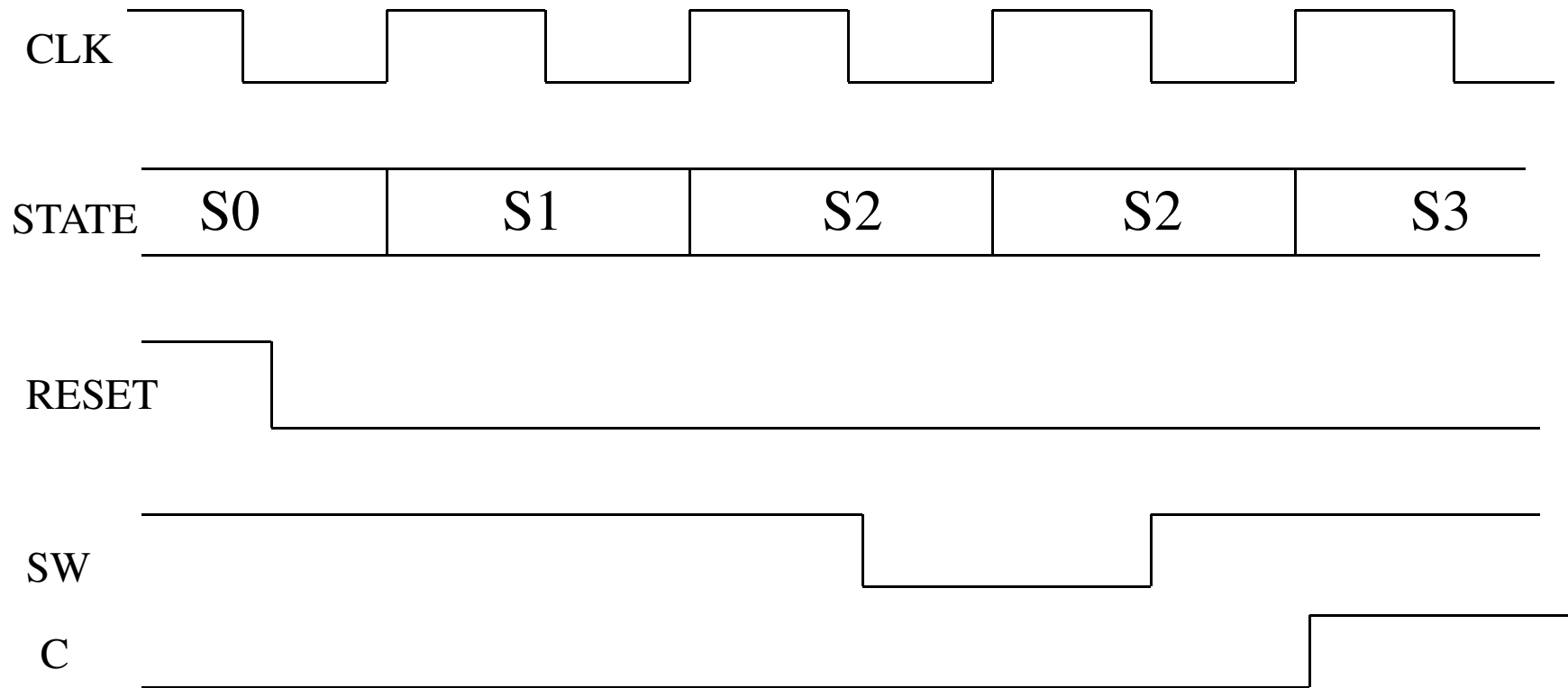
# Simple State Machine Example

---



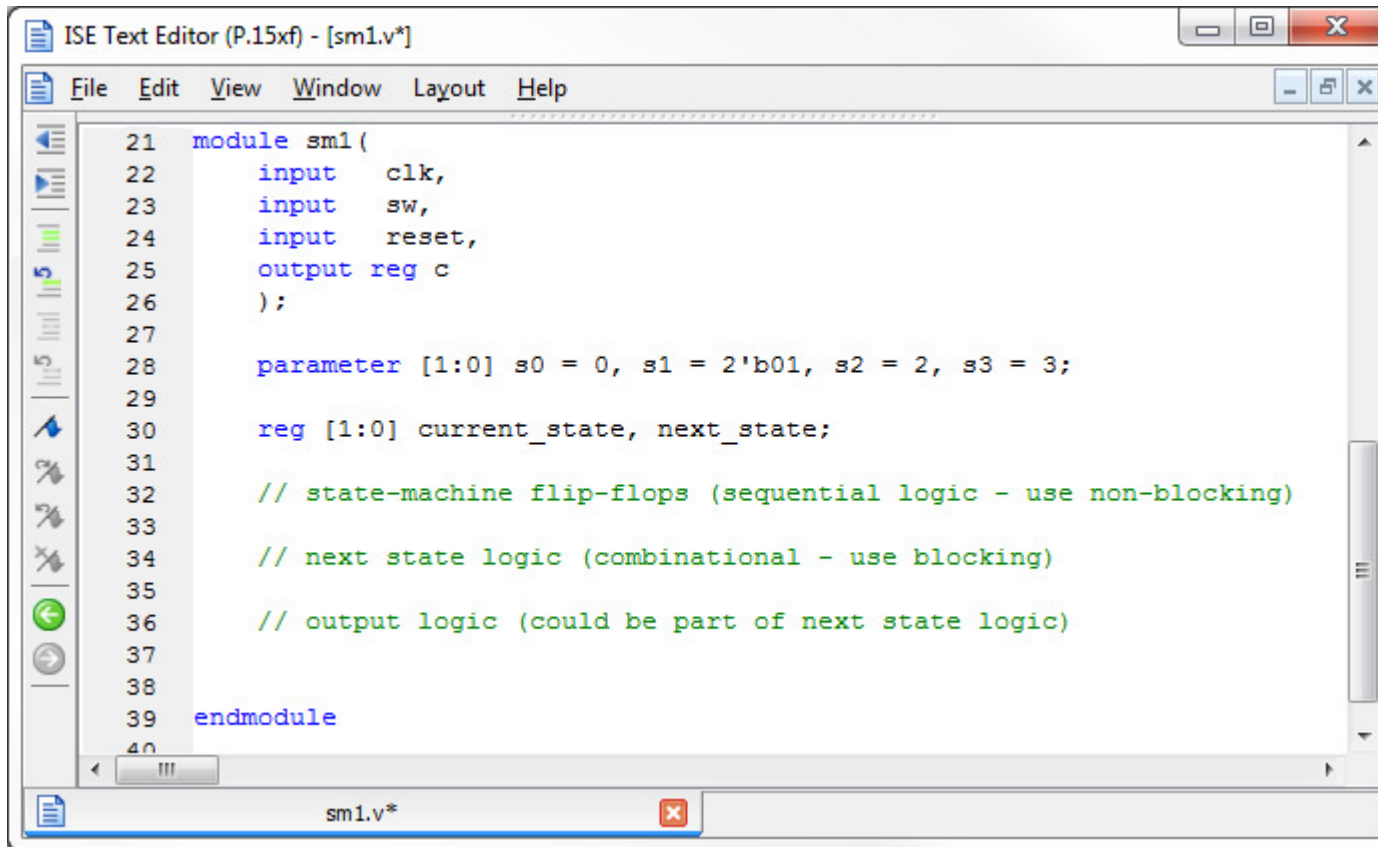
# Timing Diagram

---



# State Machine Coding Style

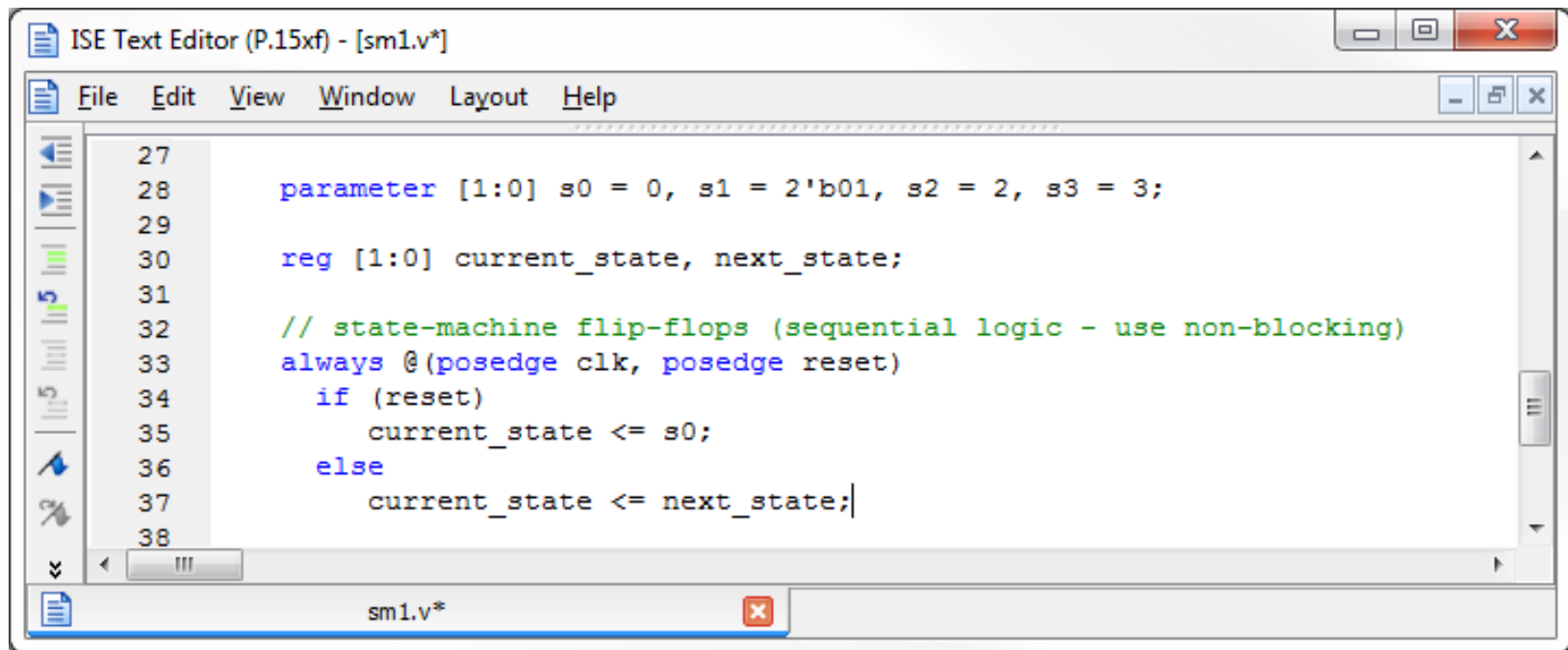
---



```
ISE Text Editor (P.15xf) - [sm1.v*]  
File Edit View Window Layout Help  
21 module sm1 (  
22     input  clk,  
23     input  sw,  
24     input  reset,  
25     output reg c  
26 );  
27  
28     parameter [1:0] s0 = 0, s1 = 2'b01, s2 = 2, s3 = 3;  
29  
30     reg [1:0] current_state, next_state;  
31  
32     // state-machine flip-flops (sequential logic - use non-blocking)  
33  
34     // next state logic (combinational - use blocking)  
35  
36     // output logic (could be part of next state logic)  
37  
38  
39 endmodule  
40  
sm1.v*
```

# State Memory (flip-flops)

---

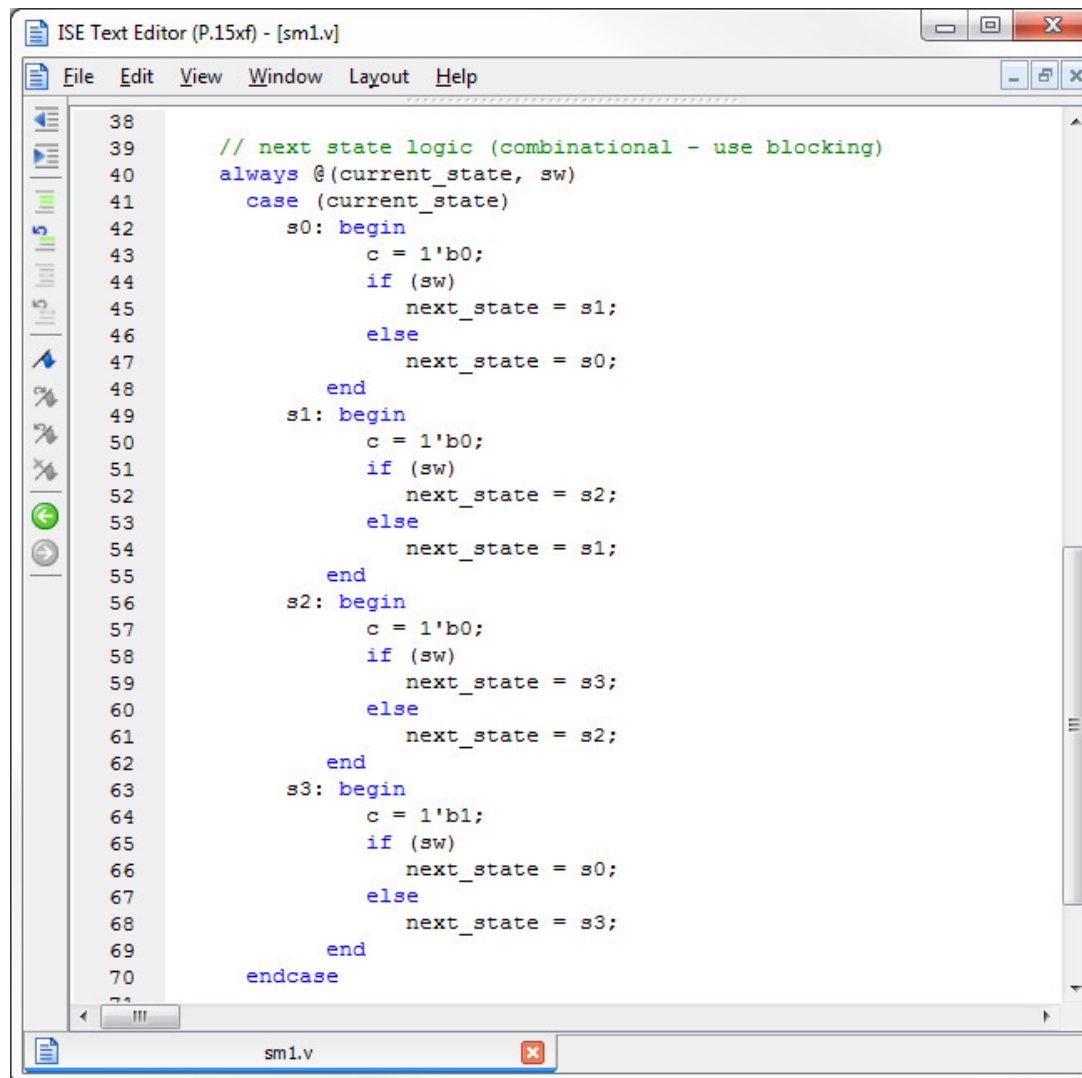


The screenshot shows the ISE Text Editor window titled "ISE Text Editor (P.15xf) - [sm1.v\*]". The window contains a menu bar with "File", "Edit", "View", "Window", "Layout", and "Help". On the left is a vertical toolbar with icons for file operations and editing. The main text area displays Verilog code for state memory, with line numbers 27 through 38 on the left margin. The code defines parameters, registers, and a state-machine flip-flop logic block.

```
27
28     parameter [1:0] s0 = 0, s1 = 2'b01, s2 = 2, s3 = 3;
29
30     reg [1:0] current_state, next_state;
31
32     // state-machine flip-flops (sequential logic - use non-blocking)
33     always @(posedge clk, posedge reset)
34         if (reset)
35             current_state <= s0;
36         else
37             current_state <= next_state;
38
```

The status bar at the bottom shows the file name "sm1.v\*" and a close button.

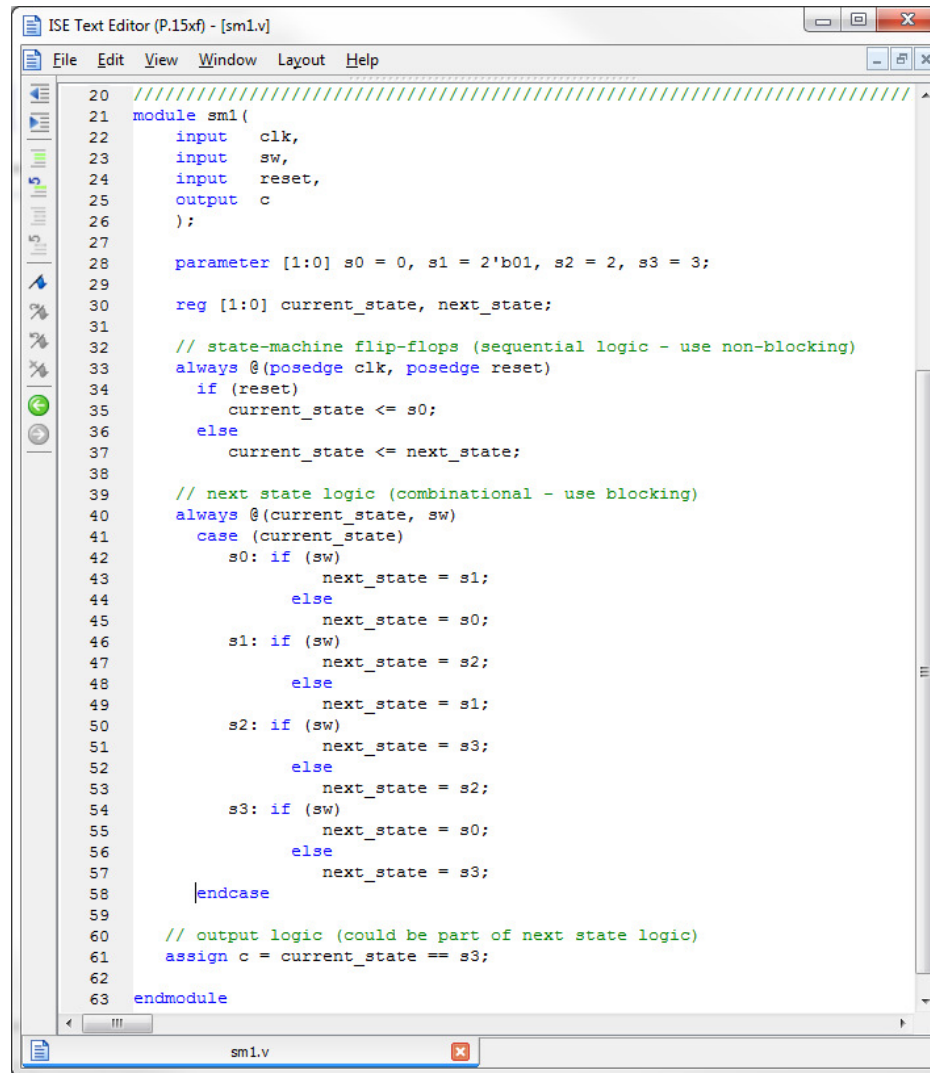
# Next State (combinational logic)



The screenshot shows the ISE Text Editor window titled "ISE Text Editor (P.15xf) - [sm1.v]". The code is written in Verilog and implements the next state logic for a finite state machine. It uses a blocking assignment and a case statement to determine the next state based on the current state and a switch variable 'sw'.

```
38
39 // next state logic (combinational - use blocking)
40 always @(current_state, sw)
41     case (current_state)
42         s0: begin
43             c = 1'b0;
44             if (sw)
45                 next_state = s1;
46             else
47                 next_state = s0;
48         end
49         s1: begin
50             c = 1'b0;
51             if (sw)
52                 next_state = s2;
53             else
54                 next_state = s1;
55         end
56         s2: begin
57             c = 1'b0;
58             if (sw)
59                 next_state = s3;
60             else
61                 next_state = s2;
62         end
63         s3: begin
64             c = 1'b1;
65             if (sw)
66                 next_state = s0;
67             else
68                 next_state = s3;
69         end
70     endcase
71
```

# SM1 – Verilog Code



```
20 //////////////////////////////////////////////////
21 module sm1(
22     input  clk,
23     input  sw,
24     input  reset,
25     output c
26 );
27
28     parameter [1:0] s0 = 0, s1 = 2'b01, s2 = 2, s3 = 3;
29
30     reg [1:0] current_state, next_state;
31
32     // state-machine flip-flops (sequential logic - use non-blocking)
33     always @(posedge clk, posedge reset)
34         if (reset)
35             current_state <= s0;
36         else
37             current_state <= next_state;
38
39     // next state logic (combinational - use blocking)
40     always @(current_state, sw)
41         case (current_state)
42             s0: if (sw)
43                 next_state = s1;
44                 else
45                     next_state = s0;
46             s1: if (sw)
47                 next_state = s2;
48                 else
49                     next_state = s1;
50             s2: if (sw)
51                 next_state = s3;
52                 else
53                     next_state = s2;
54             s3: if (sw)
55                 next_state = s0;
56                 else
57                     next_state = s3;
58         endcase
59
60     // output logic (could be part of next state logic)
61     assign c = current_state == s3;
62
63 endmodule
```

# Synthesis Report

---

```
Synthesizing Unit <sml>.
  Related source file is "C:\ece3829\sml\sml.v".
    s0 = 2'b00
    s1 = 2'b01
    s2 = 2'b10
    s3 = 2'b11
  Found 2-bit register for signal <current_state>.
  Found finite state machine <FSM_0> for signal <current_state>.
-----
| States           | 4 |
| Transitions      | 8 |
| Inputs           | 1 |
| Outputs          | 1 |
| Clock            | clk (rising_edge) |
| Reset            | reset (positive)  |
| Reset type       | asynchronous       |
| Reset State      | 00                 |
| Encoding         | auto               |
| Implementation   | LUT                 |
-----

Summary:
  inferred   1 Finite State Machine(s).

=====
*                               Advanced HDL Synthesis                               *
=====

Advanced HDL Synthesis Report
Macro Statistics
# FSMs                               : 1
=====

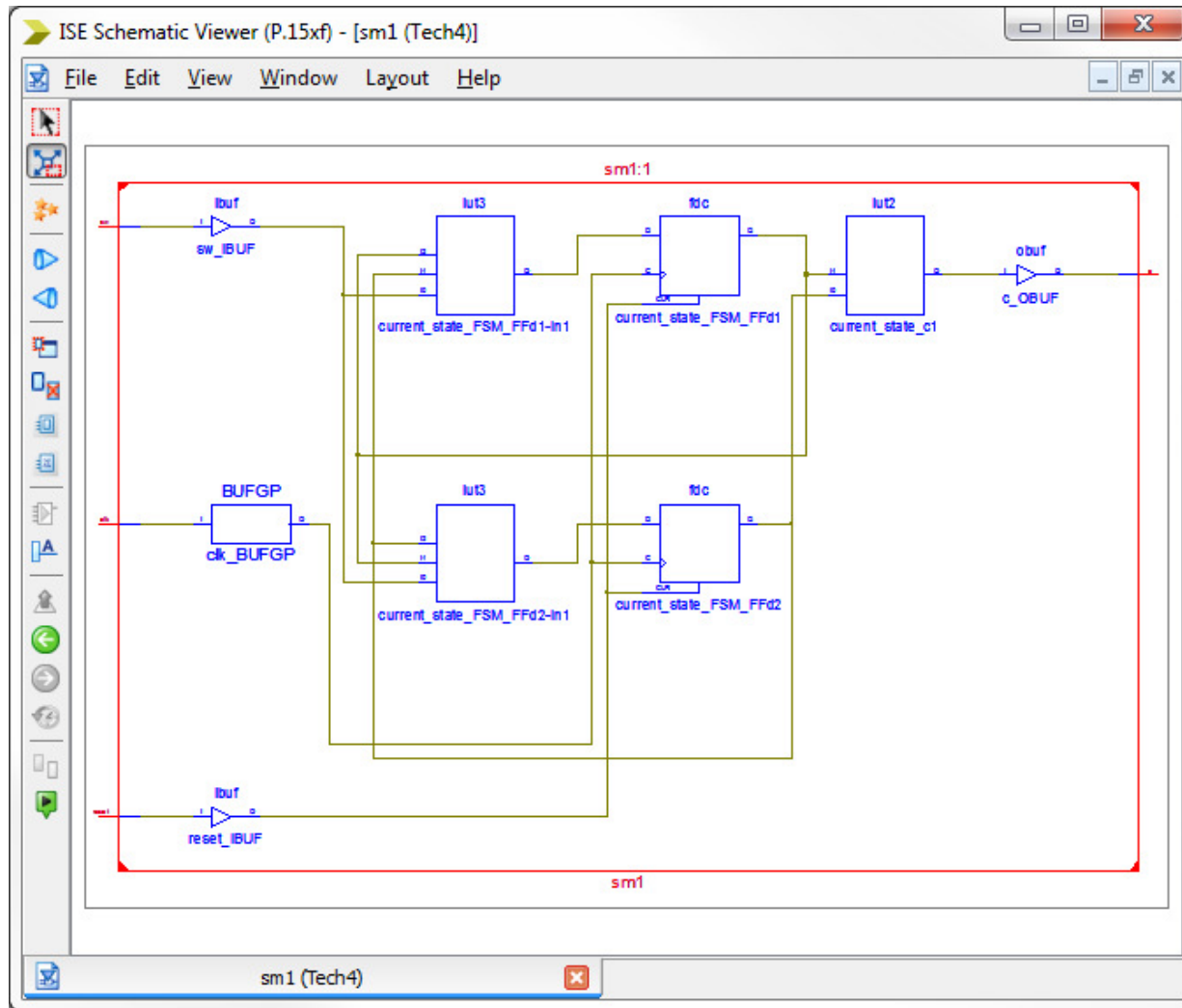
*                               Low Level Synthesis                               *
=====

Analyzing FSM <MFsm> for best encoding.
Optimizing FSM <FSM_0> on signal <current_state[1:2]> with gray encoding.
-----
State | Encoding
-----
00    | 00
01    | 01
10    | 11
11    | 10
```

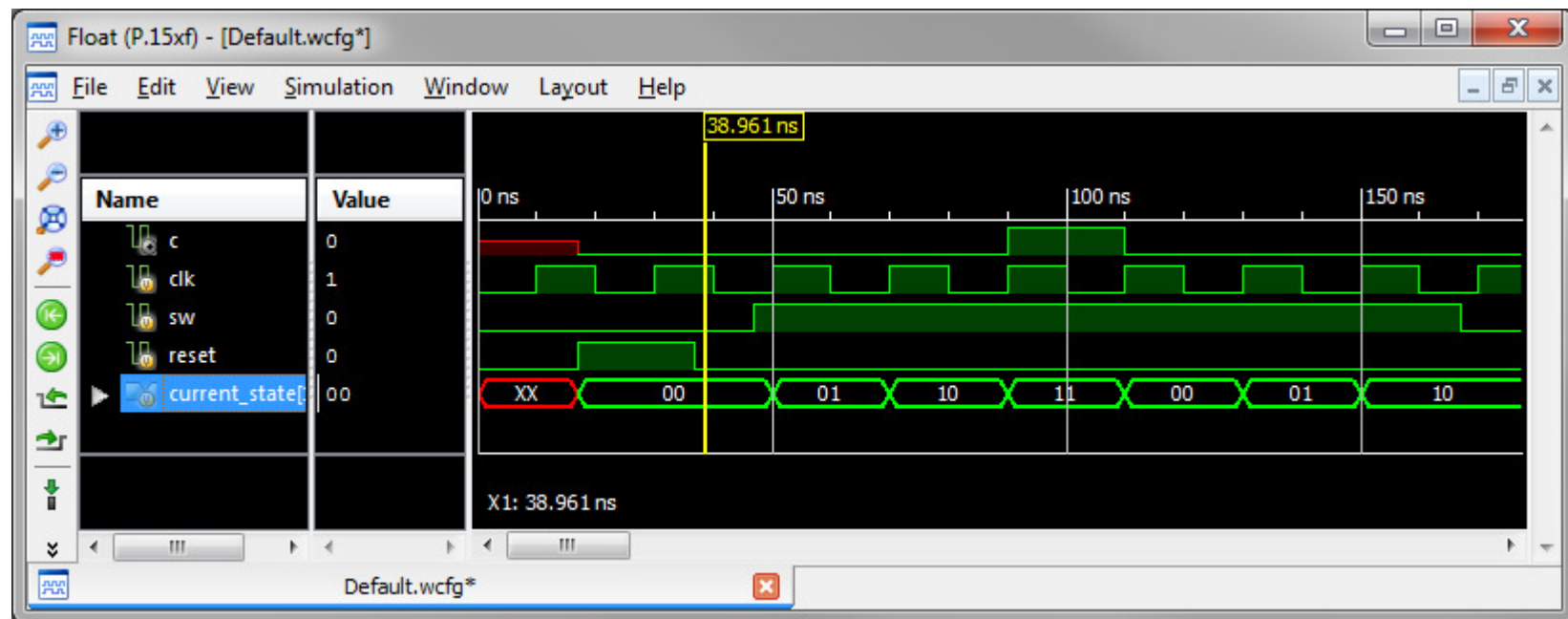
---



# SM1 – Schematic (2 flip-flops)



# Behavioral Simulation

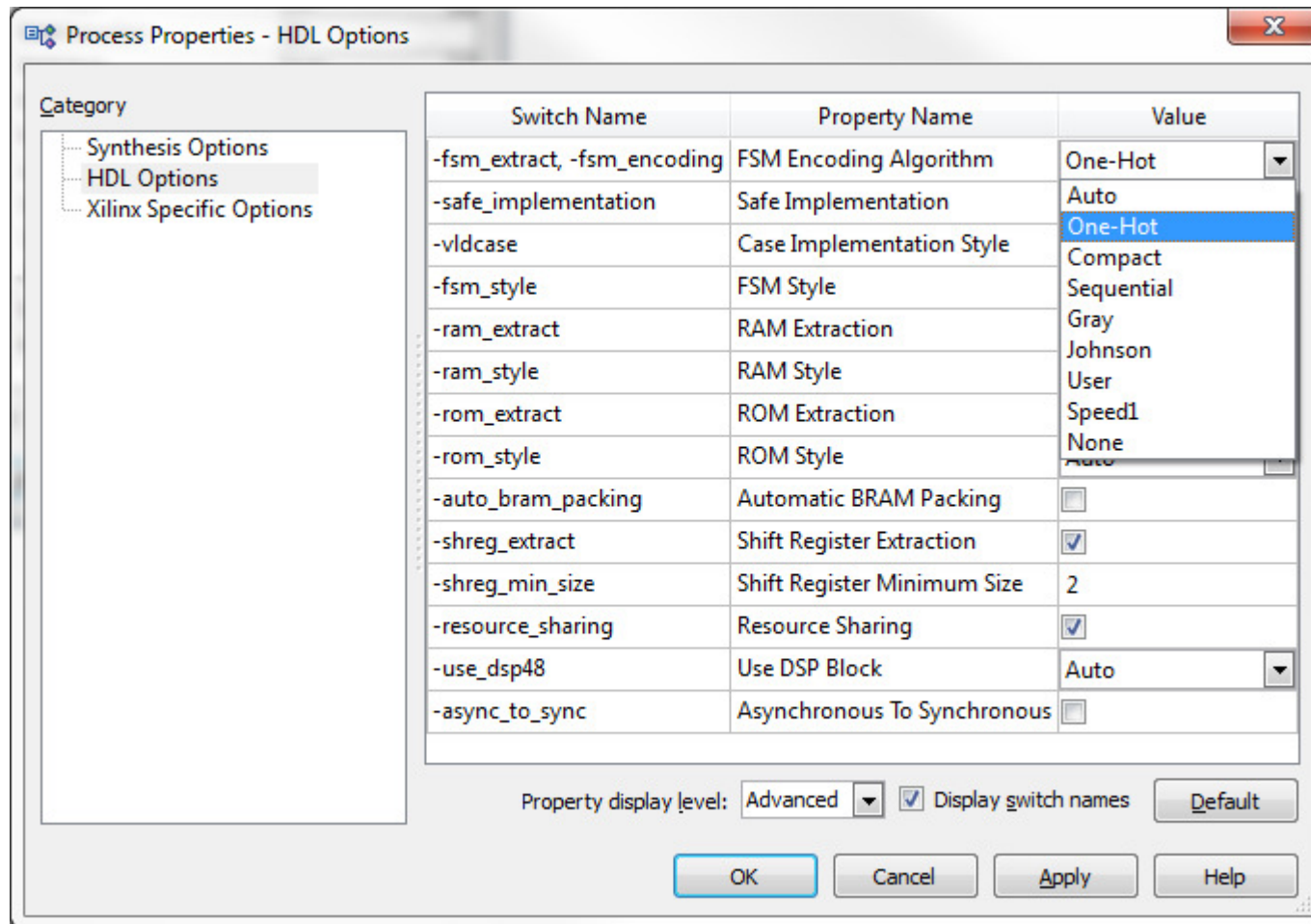


# Encoding Style

---

- One-hot
  - use one flip-flop per state
  - only one flip-flop active (hot)
  - best for flip-flop rich technology
    - use more flip-flops but simpler next state logic (faster)
  - e.g. Xilinx FPGAs (Spartan 3, Virtex, etc)
- Sequential (binary) encoding
  - generates sequential values for enumerated states
    - 00, 01, 10, 11
  - less flip-flops but more complex next-state logic
  - e.g. Altera CPLDs

# HDL Options for FSM Encoding



# Same Verilog with One-Hot Encoding

```
=====
*                               Low Level Synthesis                               *
=====
Optimizing FSM <FSM_0> on signal <current_state[1:4]> with One-Hot encoding.
-----
State | Encoding
-----
00    | 0001
01    | 0010
10    | 0100
11    | 1000
-----
```

