

ECE 551

System on Chip Design

Sequential Logic and SystemVerilog Testbenches

Garrett S. Rose
Fall 2018

Review: The Basic D Flip-Flop

- Use **always_ff** to model edge-triggered state of D Flip-Flop
- The **always_ff** block in SystemVerilog *infers* a D flip-flop
- Example detects rising edge of clock port ck with **posedge** keyword
- Falling edge of active low reset_1 port detected with **negedge** keyword
- Non-blocking <= assignments used in **always_ff** blocks

```
module dFF
    (output logic q,
     input  logic d, ck, reset_1);

    always_ff @(posedge ck, negedge reset_1)
        if(~reset_1)           // detect falling edge
            q <= 0;
        else q <= d;
endmodule: dFF
```

Testing a D Flip-Flop

- SystemVerilog testbench for D flip-flop:

```
module testDff;
    logic q, d, ck, reset_l;

    // instantiate the design under test (DUT)
    // .* -- connect ports to nets of same name
    dFF q0(.);

    initial begin
        $monitor($time, " d=%b,q=%b,ck=%b,reset_l=%b",
                d,q,ck,reset_l);
        ck = 0;
        reset_l = 0;
        d = 0;
        #1 reset_l = 1; d = 1;
        #1 ck = 1;
        #1 reset_l = 0;
        #1 ck = 0;
        #1 ck = 1;
        #1 reset_l = 1;
        #1 ck = 0;
        #1 ck = 1;
        #1 d = 0;
        #1 ck = 0;
        #1 ck = 1;
    end
endmodule: testDff
```

```
VSIM 4> run -all
```

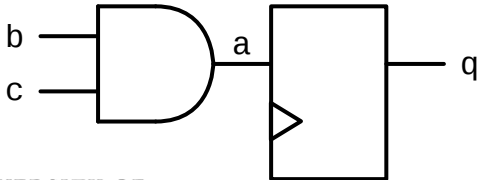
```
# 0 d=0,q=0,ck=0,reset_l=0
# 1 d=1,q=0,ck=0,reset_l=1
# 2 d=1,q=1,ck=1,reset_l=1
# 3 d=1,q=0,ck=1,reset_l=0
# 4 d=1,q=0,ck=0,reset_l=0
# 5 d=1,q=0,ck=1,reset_l=0
# 6 d=1,q=0,ck=1,reset_l=1
# 7 d=1,q=0,ck=0,reset_l=1
# 8 d=1,q=1,ck=1,reset_l=1
# 9 d=0,q=1,ck=1,reset_l=1
# 10 d=0,q=1,ck=0,reset_l=1
# 11 d=0,q=0,ck=1,reset_l=1
```

Non-Blocking Assignments Revisited

- Non-blocking assignments are *concurrent* assignments
 - Variable updated when procedural block done executing
 - All non-blocking assignments made at the same time – *concurrently*
- Blocking assignments are more like variables in a programming language

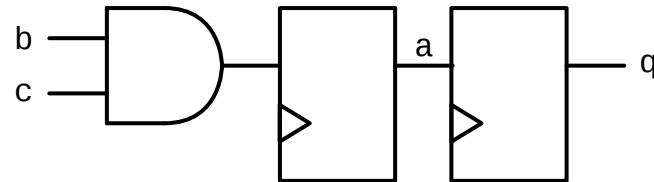
```
module ff_a
  (output logic q,
   input logic b, c, ck);
  logic a

  always_ff @(posedge ck) begin
    a = b & c; // blocking
    q <= a;
  end
endmodule: ff_a
```



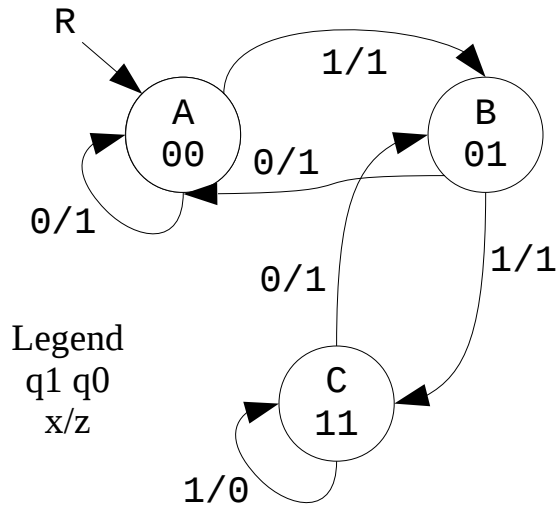
```
module ff_b
  (output logic q,
   input logic b, c, ck);
  logic a

  always_ff @(posedge ck) begin
    a <= b & c; // non-blocking
    q <= a;
  end
endmodule: ff_b
```



A Basic FSM Example

- Consider the state transition diagram below
- With simple K-maps, can determine the expressions for next state function (F) and output function (G)



		q0q1			
		00	01	11	10
x	0	0	0	0	D
	1	0	1	1	D

$$d1 = q0 \& x$$

		q0q1			
		00	01	11	10
x	0	0	0	1	D
	1	1	1	1	D

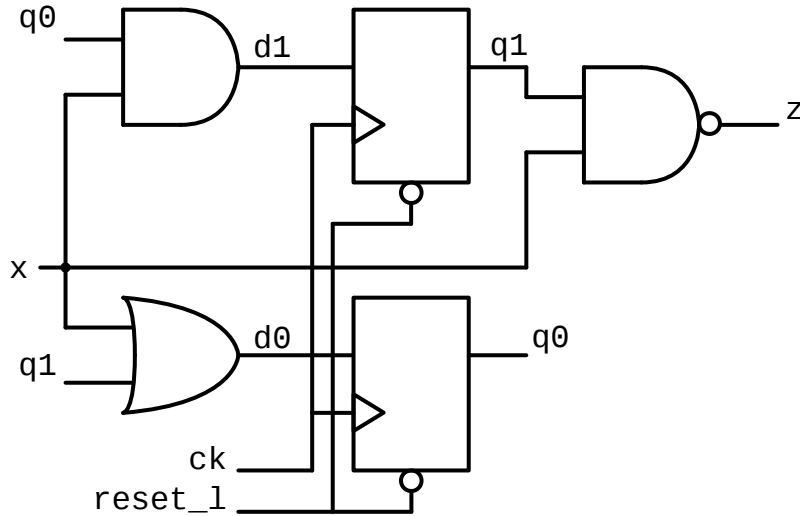
$$d0 = q1 \mid x$$

		q0q1			
		00	01	11	10
x	0	1	1	1	D
	1	1	1	0	D

$$z = \sim(q1 \& x)$$

Basic FSM Implementation – Explicit Logic

- For the basic FSM described earlier, we can derive the following circuit
- This logic circuit can be implemented directly in SystemVerilog code



```
module FSM
    (input logic x, ck, reset_l,
     output logic z);
    logic q0, q1;

    // state memory inferred; next state explicit
    always_ff @(posedge ck, negedge reset_l) begin
        if(~reset_l)
            {q1, q0} <= 2'b00;
        else begin
            q0 <= q1 | x;    // correct! - non-blocking
            q1 <= q0 & x;    // correct! - non-blocking
        end
    end

    // output logic
    assign z = ~(x & q1);
endmodule: FSM
```

Behavioral FSM Implementation

Combining State Logic with State Memory

- Use `enum` to enumerate state values

```
module FSMbehavior
  (input  logic x, ck, r_l,
   output logic z);
  enum {A, B, C} state;      // state variable

  // state memory AND next state logic in one always block
  always_ff @(posedge ck, negedge r_l) begin
    if(~r_l)                  // active low reset
      state <= A;
    else
      case (state)
        A: state <= (x) ? B : A;
        B: state <= (x) ? C : A;
        C: state <= (x) ? C : B;
        default: state <= A; // always include default!
      endcase
    end

  // output logic with always_comb
  always_comb begin
    z = 1'b1;                // z starts with 1 - may override
    if (state == C) z = ~x;   // OK - z is updated no matter what!
  end
endmodule: FSMbehavior
```

Behavioral FSM Implementation

Separate Blocks for F, G, and State Memory

- Use `enum` to enumerate state values

```
module FSMalternate
  (input  logic x, ck, r_l,
   output logic z);
  enum {A, B, C} state;      // state variable

  // 1- state memory only
  always_ff @(posedge ck, negedge r_l)
    if(~r_l) state <= A;
    else    state <= nextState;

  // 2- next state logic (F) only
  always_comb
    case (state)
      A: nextState <= (x) ? B : A;
      B: nextState <= (x) ? C : A;
      C: nextState <= (x) ? C : B;
      default: nextState <= A;
    endcase
end

// 3- output logic (G) with assign
z = (state == C) ? ~x : 1'b1;
endmodule: FSMalternate
```


Behavioral FSM Implementation

Separate Blocks for F, G, and State Memory

- Use `enum` to enumerate state values

```
module FSMalternate
  (input  logic x, cl, r_l,
   output logic z);
  enum {A, B, C} state;      // state variable

  // 1- state memory only
  always_ff @(posedge ck, negedge r_l)
    if(~r_l) state <= A;
    else    state <= nextState;

  // 2- next state logic (F) only
  always_comb
    case (state)
      A: nextState <= (x) ? B : A;
      B: nextState <= (x) ? C : A;
      C: nextState <= (x) ? C : B;
      default: nextState <= A;
    endcase
end

// 3- output logic (G) with assign
z = (state == C) ? ~x : 1'b1;
endmodule: FSMalternate
```

- Resulting synthesized logic likely similar for all styles
- Simulation likely slower for style with separated blocks
- Separating blocks simplifies design and debugging

SystemVerilog Testbenches for FSMs

- Large system verification – use the **program** construct (more to come)
- Smaller digital systems and state machines – use **module** construct
- Need to test:
 - State transitions (as close to exhaustive as possible)
 - Next state combinational logic
 - Output combinational logic

Testbench Clock Stimulation – forever

- Need to exercise the clock – you've seen some examples
- A **forever** statement – a loop that continuously executes, forever

```
logic clock;                                // internal clock

initial begin
    clock = 0;                               // must initialize clock to 0
    forever #5 clock = ~clock; // otherwise, clock always X
end
```

Testbench Clock Stimulation – repeat

- Perhaps you want the same statement to run many times, just not forever
- A **repeat** statement – a loop that executes for a set number of iterations
 - The iterations are set in the () after the **repeat** keyword

```
logic clock;                                // internal clock

initial begin
    clock = 0;                                // initialize clock to 0
    repeat (1000) #5 clock = ~clock; // repeats 1000 times
end
```

- Repeating ~clock (not clock) 1000 times leads to 500 full state changes

Stimulating Clock and Reset Together

- Need to exercise the clock – you've seen some examples
- A **forever** statement – a loop that continuously executes, forever

```
logic clock, reset_l;           // internal clock, active low reset

initial begin
    clock = 0;                   // initialize clock to 0
    reset_l = 0;                 // initialize reset_l to 0 - active
    #1 reset_l = 1;              // reset_l deactivated after 1 cycles
    #4 forever #5 clock = ~clock; // begin forever clock _after_ 4 cycles
end
```

Stimulating Clock and Reset Together

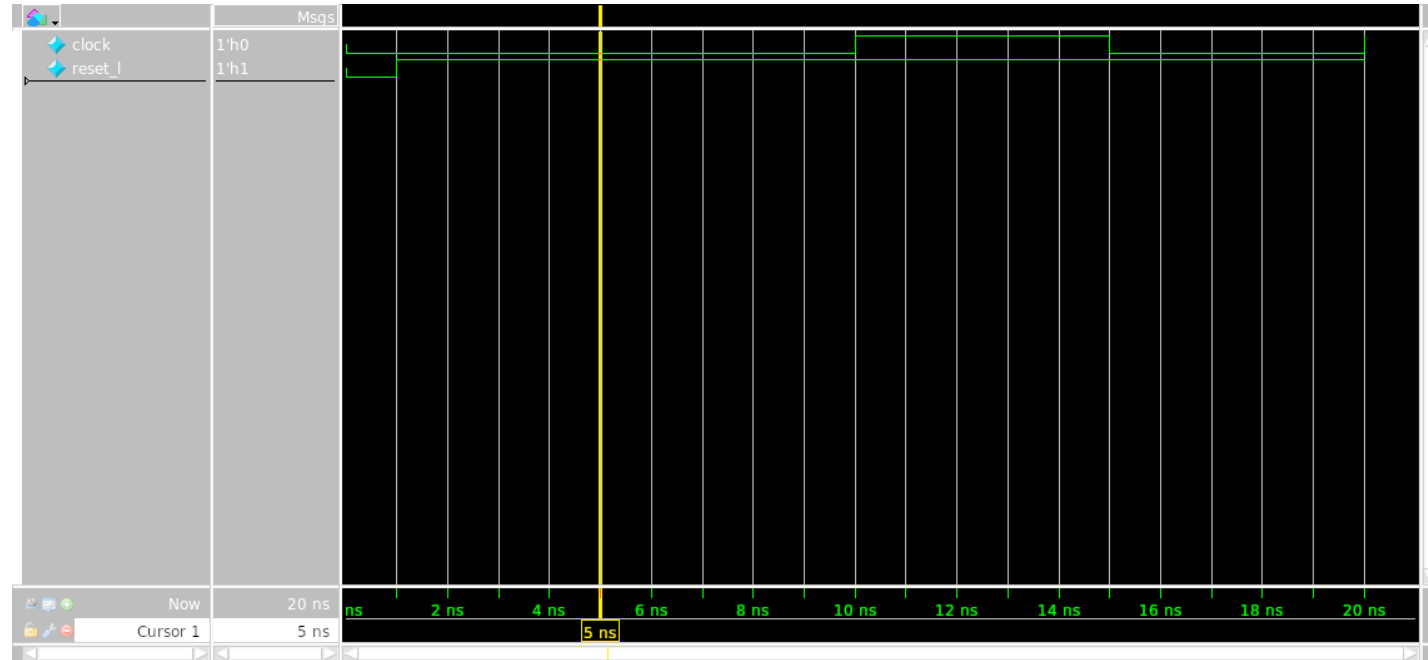
- Need to exercise the clock – you've seen some examples
- A **forever** statement – a loop that continuously executes, forever

```
logic clock, reset_l;           // internal clock, active low reset

initial begin
    clock = 0;                   // initialize clock to 0
    reset_l = 0;                 // initialize reset_l to 0 - active
    #1 reset_l = 1;             // reset_l deactivated after 1 cycles
    #4 forever #5 clock = ~clock; // begin forever clock _after_ 4 cycles
end
```

- Note: inversion of clock is delayed by 5 time steps/iterations
- reset_l deactivated at step #1, delay the forever 4 steps for first ~clock

Stimulating Clock and Reset Together



- Note: inversion of `clock` is delayed by 5 time steps/iterations
- `reset_1` deactivated at step #1, delay the forever 4 steps for first `~clock`

Using Non-Blocking Assignment

- Suppose we want to deactivate reset_l after 1 time step, regardless of clock
- In this case, reset_l would be set to 1 concurrently using non-blocking assignment – at end of time step 1, reset goes to 1, regardless of everything else

```
logic clock, reset_l;           // internal clock, active low reset

initial begin
    clock = 0;                  // initialize clock to 0
    reset_l = 0;                // initialize reset_l to 0 - active
    reset_l <= #1 1;            // reset_l deactivated at end of #1
    forever #5 clock = ~clock;  // forever clock begins immediately
end
```


Using Non-Blocking Assignment

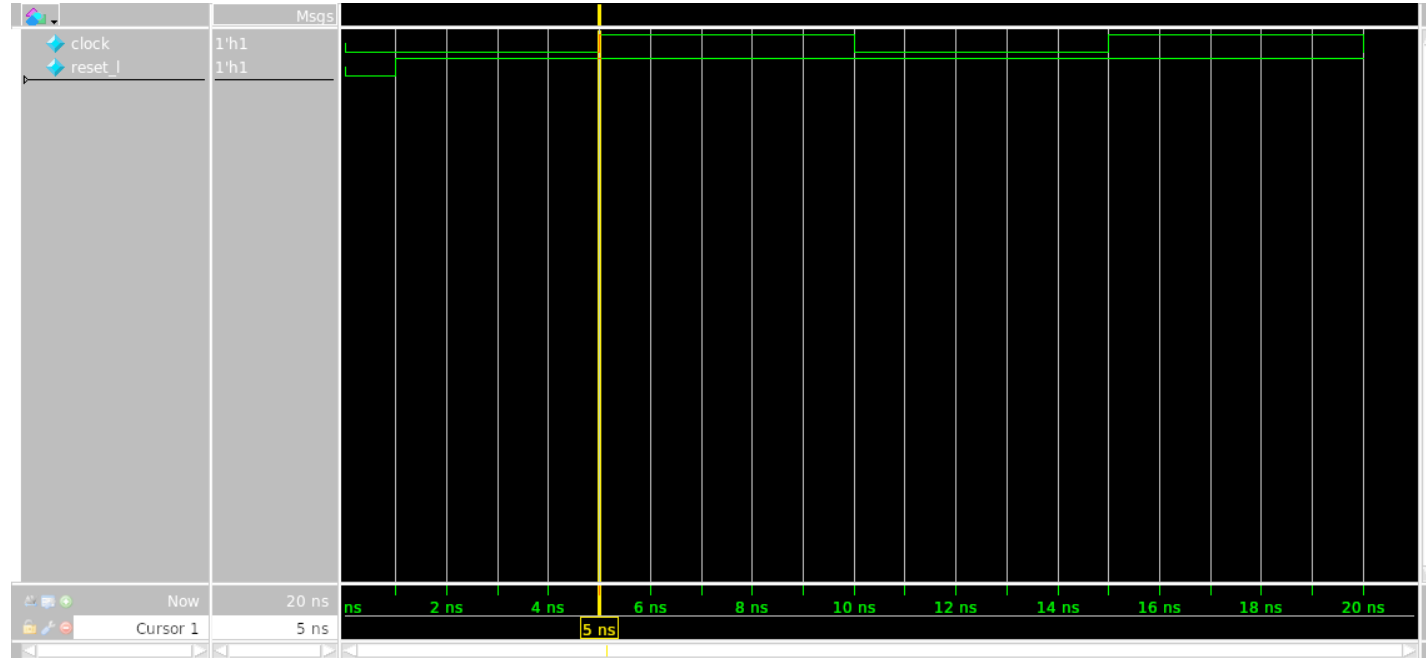
- Suppose we want to deactivate reset_l after 1 time step, regardless of clock
- In this case, reset_l would be set to 1 concurrently using non-blocking assignment – at end of time step 1, reset goes to 1, regardless of everything else

```
logic clock, reset_l;           // internal clock, active low reset

initial begin
    clock = 0;                  // initialize clock to 0
    reset_l = 0;                // initialize reset_l to 0 - active
    reset_l <= #1 1;            // reset_l deactivated at end of #1
    forever #5 clock = ~clock;  // forever clock begins immediately
end
```

- The forever clock assignment starts immediately, first transition after 5 steps

Using Non-Blocking Assignment



- The forever clock assignment starts immediately, first transition after 5 steps

Heirarchical Naming and \$monitor

- For an FSM, we will want to monitor state transitions, not just output state
- Assuming enum used to define state values, monitor can display state names using “.” to descend into the heirarchy of design under test

```
module test_bench;
    logic port_in, port_out;        // net names match port names of dut

    design dut(. *);                // .* maps nets and ports of same names

    initial begin
        $monitor($time, " Current State = %s", dut.state.name);
    end
endmodule: test_bench

module design
    (input  logic port_in,
     output logic port_out);        // port names same as net names in tb
    enum logic [2:0] {Init, A, B, C} state; // state is internal

    ...
endmodule: design
```

The Implicit State Machine

- Implicit state machine simply refers to stepping sequentially through state changes – useful for exciting and testing state transitions in an FSM
- The variable `x` in this example is assigned as state output

```
initial begin: I                // 'I' is just a name, of clock stim initial
    ck = 0;
    forever #5 ck = ~ck;        // 10 step (ns) clock - transition every 5 steps
end

initial begin: J                // 'J' is just a name, of our implicit FSM
    x <= 1;                    // non-blocking assignment of x - initial
    @(posedge ck);             // resume at time 5, no change to x
    @(posedge ck);             // resume at time 15, no change to x
    @(posedge ck);             // resume at time 25, x changes
    x <= 0;
    @(posedge ck);             // resume at time 35
    #1 $finish;
end
```

...

The Implicit State Machine - Example

- Test state transitions of basic FSM from earlier

```
module fsmTest;
  logic ck, x, z, r_l;
  FSMbehavior dut(.*);

  initial begin: I
    $monitor($time, " Current State = %s", dut.state.name);
    ck = 0; r_l = 0;
    r_l <= #1 1;
    forever #5 ck = ~ck;
  end

  initial begin: J
    x <= 1;
    @(posedge ck);
    @(posedge ck);
    @(posedge ck);
    x <= 0;
    @(posedge ck);
    #1 $finish;
  end
endmodule: fsmTest
```

The Implicit State Maching – Example

Reminder: FSMbehavior

```
module FSMbehavior
  (input  logic x, ck, r_l,
   output logic z);
  enum {A, B, C} state;      // state variable

  // state memory AND next state logic in one always block
  always_ff @(posedge ck, negedge r_l) begin
    if(~r_l)                  // active low reset
      state <= A;
    else
      case (state)
        A: state <= (x) ? B : A;
        B: state <= (x) ? C : A;
        C: state <= (x) ? C : B;
        default: state <= A; // always include default!
      endcase
    end

  // output logic with always_comb
  always_comb begin
    z = 1'b1;                // z starts with 1 - may override
    if (state == C) z = ~x;   // OK - z is updated no matter what!
  end
endmodule: FSMbehavior
```

The Implicit State Machine – Example Results

- Simulation marches through the 3 states of FSMbehavior

```
VSIM 26> run
```

```
#          0 Current State = A
#          5 Current State = B
#         15 Current State = C
#         35 Current State = B
```

