SV Notes #5: FSM Basics

New Keywords and Operators

1. **`always_ff @( ... events ... )`**

   Generates a **synchronous** logic block. This block is **sensitive** to signal conditions specified in `@( ... events ... )`. The block may contain procedural Verilog constructs (e.g. **`if (...)`**) and is bounded by **`begin ... end`** keywords.

   (a) **`posedge`** - Example: **`always_ff @(posedge clock)`**

   Denotes sensitivity to a **positive** (rising) edge (`0`→`1`) in a signal event list.

   (b) **`negedge`** - Example: **`always_ff @(negedge reset_L)`**

   Denotes sensitivity to a **negative** (falling) edge (`1`→`0`) in a signal event list.

2. **`<=`** - Example: `q <= d;`

   The **concurrent assignment** or **non-blocking assignment** operator. Assigns the value of the right-hand operand into the left-hand operand *synchronously* with all other concurrent assignments triggered by the same event (e.g. positive clock edges).

   - The concurrent assignment operator should always be used when updating state variables in **`always_ff`** blocks!
   - This operator may also be used in testbenches to *schedule* events concurrently at the *current simulation timestep*!

3. **`forever`** - Example: **`forever #5 clock = ~clock;`**

   Loops over procedural commands bounded by **`begin ... end`** keywords. Handy for generating clock signals.

Code Listings

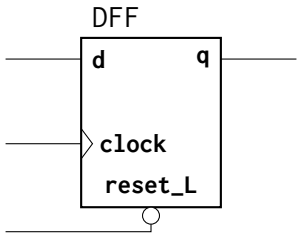
D Flip-Flop with Asynchronous Reset

```
module DFF
  (input  logic d, clock, reset_L,
   output logic q);

  // rising clock or falling reset_L events trigger this block
  always_ff @(posedge clock, negedge reset_L) begin
    if (~reset_L)      // reset signal is active-low
      q <= 0;
    else
      q <= d;          // state changes use concurrent assignment operator!
  end

endmodule: DFF
```

When DFF is instantiated, it synthesizes to the following equivalent:



Clocking a DFF

This testbench generates a clock module and signals to test a D flip-flop.

```
module DFF_test;
  logic clock, d, reset_L, q;  // declare logic signals

  DFF dut(.d, .q, .reset_L, .clock);  // instantiate the DFF

  initial begin       // generate clock signal with period = 20 time units
    clock = 0;
    forever #10 clock = ~clock; // posedges at time = 10, 30, 50, 70, 90, etc.
  end

  initial begin       // monitor
    $monitor($time,,"clock=%b, reset_L=%b, d=%b, q=%b",
                        clock, reset_L, d, q);
  end

  initial begin         // generate data (d) and reset_L waveforms to test
        reset_L <= 0;   // time = 0, schedule reset of the flip flop,
    #5        d <= 1;   // time = 5, schedule d = 1
    #10 reset_L <= 1;   // time = 15, schedule de-assertion of reset
    #20       d <= 0;   // time = 35, schedule d = 0
    #5        d <= 1;   // time = 40, schedule d = 1
    #15       d <= 0;   // time = 55, schedule d = 0
    #10  reset_L = 0;   // time = 65, reset the flip flop,
        reset_L <= 1;   //       and schedule immediate de-assertion of reset,
              d <= 1;   //       and schedule d = 1
    #35       $finish;  // time = 100, stop simulation
  end
endmodule : DFF_test
```

The following waveforms will result from this testbench: