Modeling Synchronous Systems With Verilog

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

NOR gate RS Latch

```
module rs_latch(output logic q, qb, input logic r, s );
     nor reset_g( q, qb, r);
     nor set_g( qb, q, s);
endmodule
```

- Structural verilog
- nor_rs.v - Module and testbench
- r=0 s=1 q=1 qb=0
- r=1 s=0 q=0 qb=1
- r=0 s=0 q=0 qb=1 (keep)
- r=1 s=1 q=0 qb=0 “illegal” state

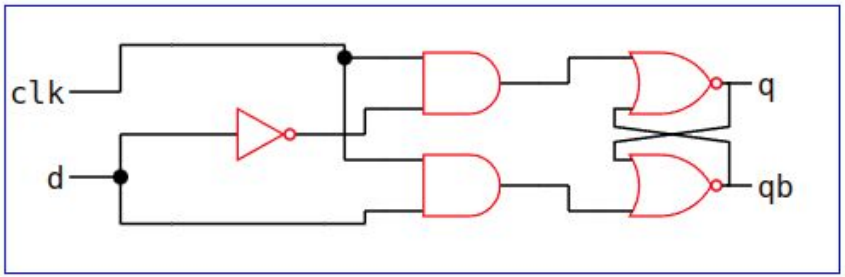
NAND gate RS Latch

```
module rs_latch(output logic q, qb, input logic r, s );
     nand reset_g( q, qb, r);
     nand set_g( qb, q, s);
endmodule
```

- nand_rs.v - Module and testbench
- S = 1, R = 1 - keep value;
- S = 0, R = 0 - “illegal” state

Modeling a D Latch

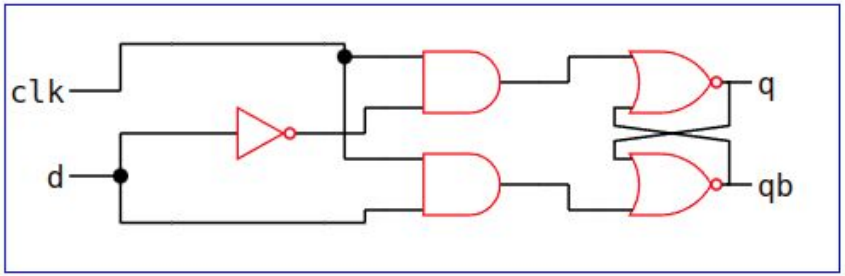
- Avoids the “illegal” inputs
- Input directly affects the “q” output when the clk is 1
- “q” value is retained when clk is 0
- While clk is 1 the “d” latch is said to be transparent
    - “d” changes/affects “q” during if clock is 1



Modeling a D Latch

- Verilog uses the “always_latch” statement to model this behaviour

```
module d_latch( output logic q, input logic d, clk);
     always_latch begin
          if ( clk ) q <= #delay d;
     end
endmodule
```



Modeling a D Latch

- d_latch.v, d_latch_tb.v
- The $monitor task will display the variables, and the current simulation time whenever one of the variables changes
- waveform, d_latch.vcd

Modeling a D Flip-flop

```
module dff( output logic q, input logic d, clk);
     always_ff @(posedge clk) begin // Positive edge
          q <= #delay d;
     end
endmodule
```

- Positive edge
- q is output and the memory (register) saved
- Not transparent
  - does not change while clock is high, only during the positive edge
- dff.v, dff_tb.v, waveform, dff.vcd

Register (n-bit DFF) with reset

- N-bit bus input/output
- Reset
- nbit_dff.v, waveform, nbit_dff.vcd

Register (n-bit DFF) with reset

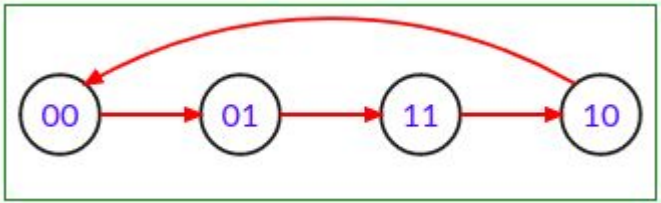
```
module nbit_dff #(parameter N=8) (output logic [N-1:0] q, // bus of N bits
                                  input  logic [N-1:0] d, // bus of N bits
                                  input logic clk, reset );
     always_ff @(posedge clk) begin
              if ( reset ) begin
                       q <= 0;
              end
              else begin
                       q <= d;
              end
     end
endmodule
```

Synchronous Circuits Design rules - Lec 13

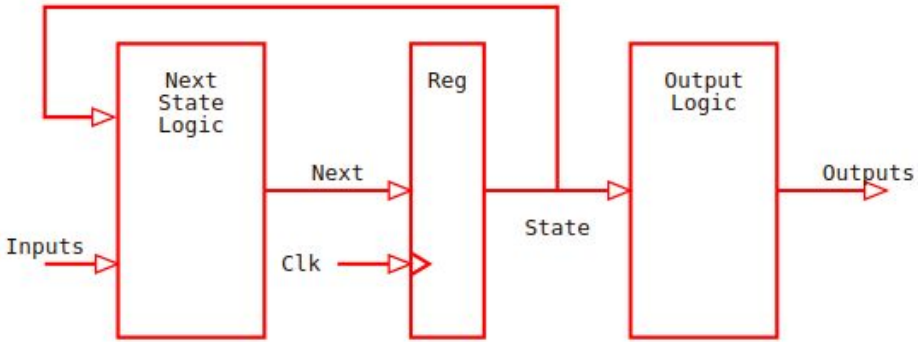
- Every circuit element is either a register (group of D flop-flops) or a combinational circuit.
- At least one circuit is a register.
- All registers receive the same clock signal.
- Every cyclic path contains at least one register.

Sequence Generator (FSM Implementation)

Consider the following state graph.

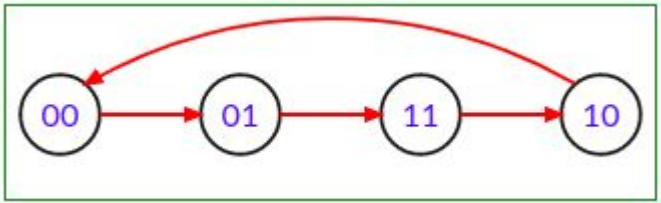


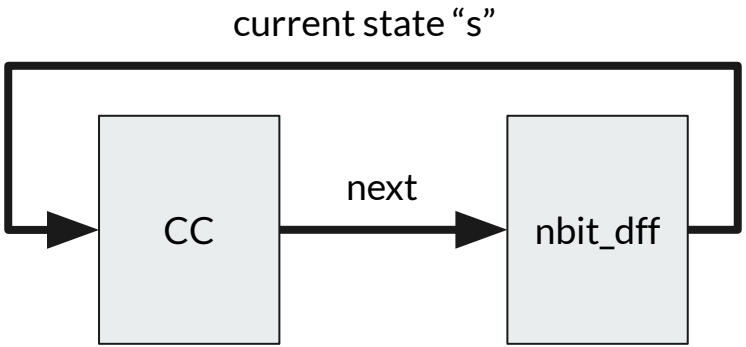
- next state function
  - Implemented with a combinational logic block
- a two-bit register
  - hold the state



Sequence Generator With A Next State Function

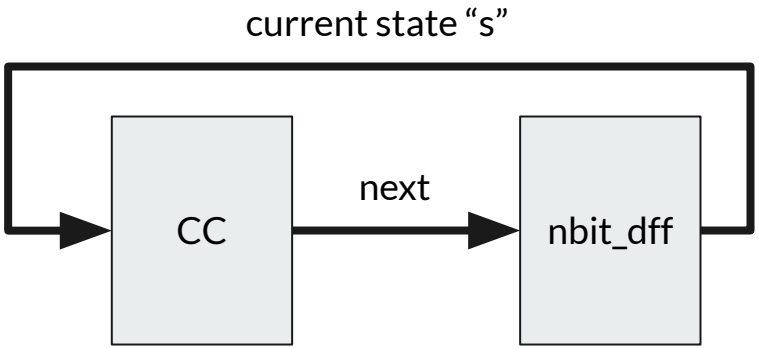
- create next state function module
  - Implemented with a combinational logic block
- a two-bit register
  - hold the state
- example: seq4_v1.v





Sequence Generator With A Next State Function

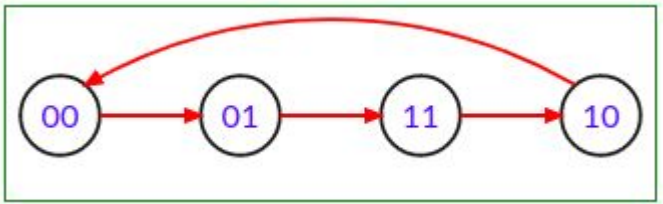
- nbit_dff module is used to create the two bit register
- seq4_v1.v
  - uses: nbit_dff.v




```
module next_state (output logic [1:0]nx,
input logic [1:0] st);
     always_comb begin

          case(st)

               2'b00: nx = 2'b01;

               2'b01: nx = 2'b11;

               2'b11: nx = 2'b10;

               2'b10: nx = 2'b00;

          endcase

     end
endmodule

nbit_dff #(2) st( st, nx, clk, reset);
```

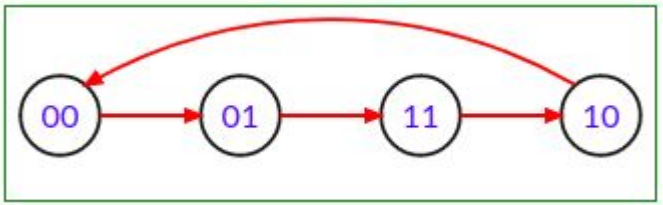
Sequence Generator (FSM Implementation)

- implementation using always_ff and case
- seq4_func.v



Sequence Generator (FSM Implementation)

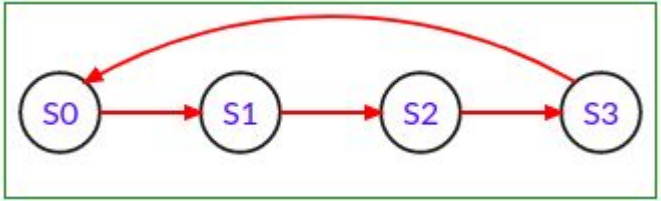
- implementation using always_ff and case
- 00-sandbox/seq4_func.v




```
always_ff @(posedge clk) begin

      if ( reset )

            s <= 0; // register

      else

            case(s)

                  2'b00: s <= 2'b01; // state: next state

                  2'b01: s <= 2'b11;

                  2'b11: s <= 2'b10;

                  2'b10: s <= 2'b00;

            endcase

      end
```

Sequence Generator With Enum

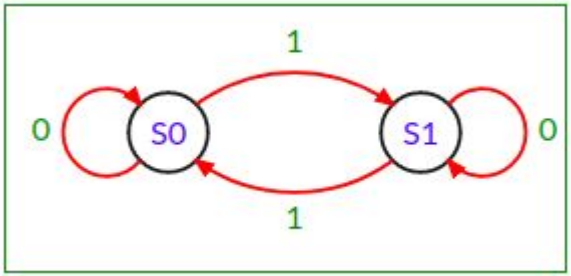
- label the state with symbolic names
- seq4_v2.v




```
typedef enum logic [1:0] { S0=0, S1=1, S2=3, S3=2 } seq_t;

case(s)
     S0: s <= S1;
     S1: s <= S2;
     S2: s <= S3;
     S3: s <= S0;
endcase
```

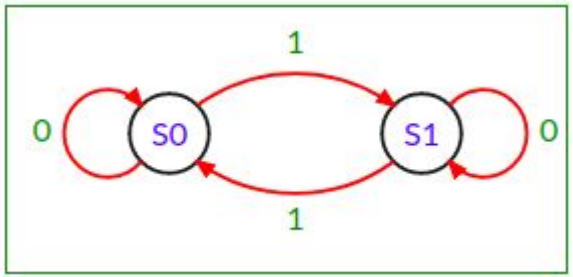
Odd/Even FSM

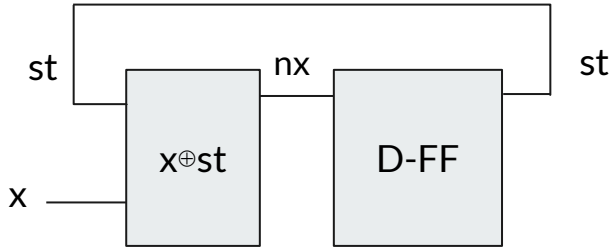
- Odd/even number of 1s
- Let use a structural design
  - only using gates
- odd_even_v1.v



Odd/Even FSM

- Odd even number of 1s
- Let use a structural design
  - only using gates
- odd_even_v1.v

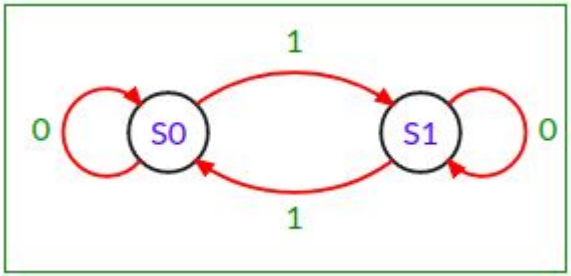




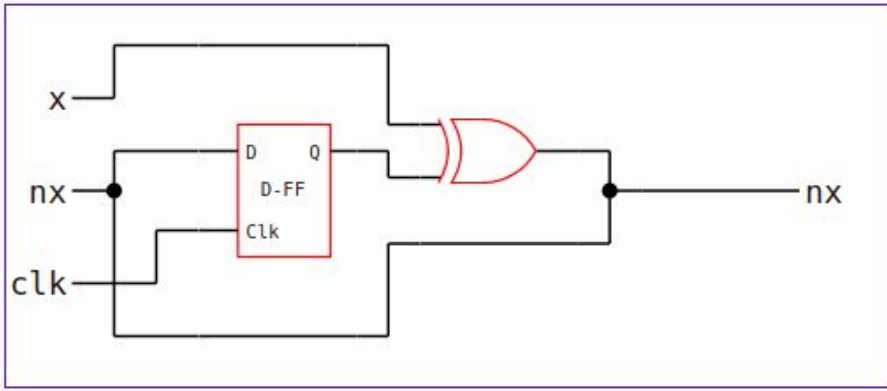

```
module odd_even( output logic nx, input logic x, clk, reset );
        logic st;
        xor next( nx, st, x);
        dff state( st, nx,  clk, reset );
endmodule
```

Behavioural Odd/Even FSM (odd_even_behave.v)

```
module behave_odd_even( output logic nx,
                        input logic x, clk, reset );
      logic st; // store the state
      assign nx = st;
      always_ff @(posedge clk) begin
                  if ( reset ) st <= 0;
                  else st <= x ^ st; // register
      end
endmodule
```

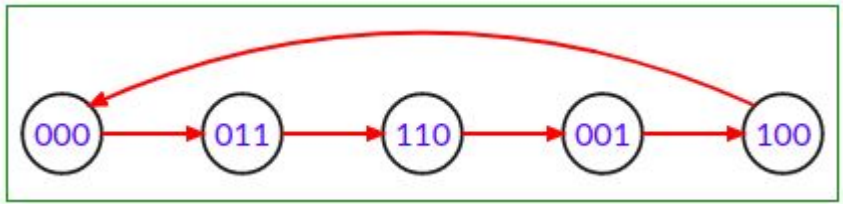


- Synthesises a register for “st”
- Mealy machine example
  - input “x” affects the output



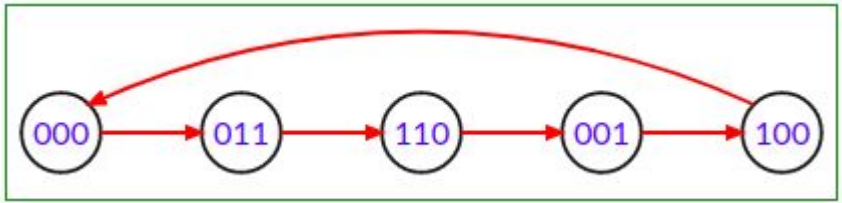
Counter FSM In Structural Verilog

- modules
  - “nbit_dff” sequential ff
  - “next” combinational next state (state + 3)
  - “counter” everything together
  - counter.v and counter_func.v



Counter FSM In Structural Verilog

- modules
  - “nbit_dff” sequential ff
  - “next” combinational next state (state + 3)
  - “counter” everything together
  - counter.v
- Example in the notes



s2	s1	s0	n2	n1	n0
0	0	0	0	1	1
0	0	1	1	0	0
0	1	0	X	X	X
0	1	1	1	1	0
1	0	0	0	0	0
1	0	1	X	X	X
1	1	0	0	0	1
1	1	1	X	X	X

Counters (counter_fun.v)

- Addition of the count output/register
- Instead of updating the next state

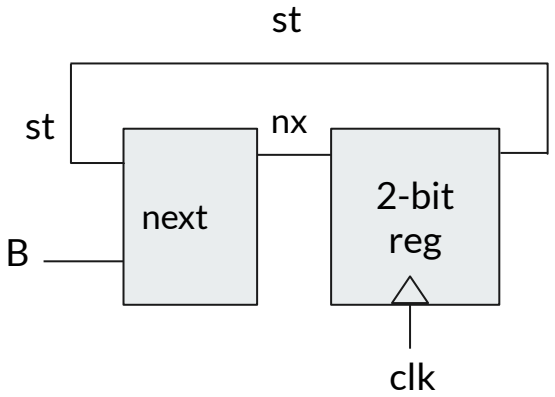
```
module counter #(parameter N=8) ( output logic [N-1:0] count,
                                                  input logic clk, reset );
      always_ff @(posedge clk) begin
            if ( reset )
                  count <= #2 0;
            else
                  count <= #2 (count + 1);
      end
endmodule
```

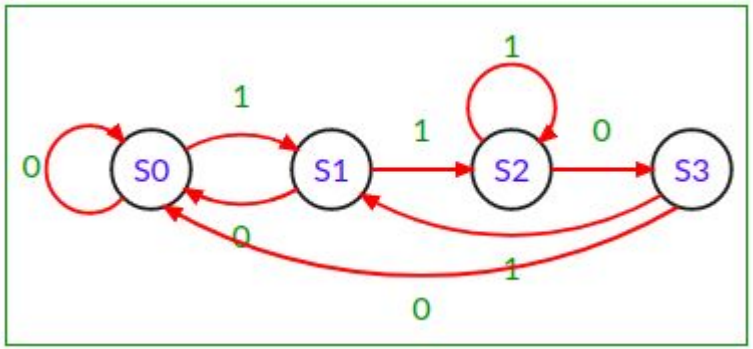
Counter With Adders And Registers

- modules
  - “ha” half-adder
  - “fa” full-adder
  - “adder” adder together
  - “counter” everything together
  - Example in the repo but will be covered in Unit 4
- Example in the notes

A Recognizer FSM (recognize.v)

- When the FSM is in S3, the input sequence has been accepted
- Input: 0101, States: S0 S1 S0 S1
- Input: 1110, States: S1 S2 S2 S3
- Input: 1110110, States: S1 S2 S2 S3 S1 S2 S3
- shift_reg_tb.v






```
typedef enum logic [1:0] {S0, S1, S2, S3} state_t;

always_ff @(posedge clk) begin
```

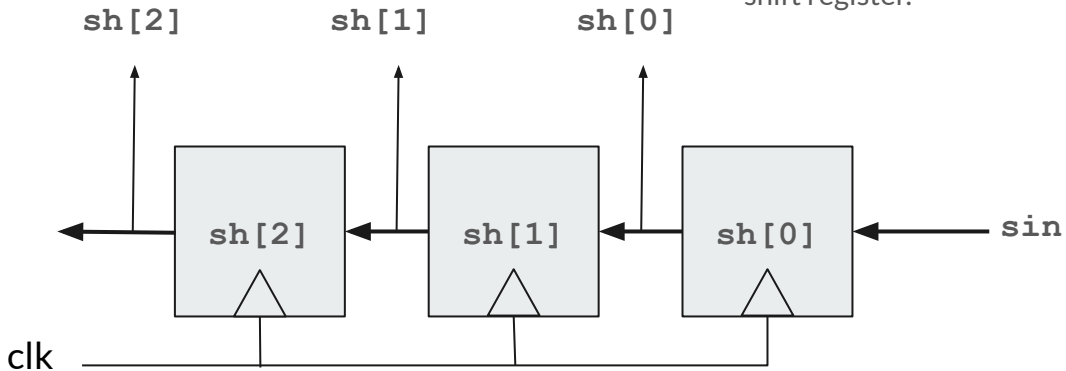
Three Bit Shift Register

- positive edge
- value of sh[0] is updated by sin
- sh[0] current value is copied to sh[1]
- The same is true for sh[1] and sh[2]
- sh[2] current value is lost, since it is at the end of the shift register.

Three Bit Shift Register (shift_reg_tb.v)

```
always_ff @(posedge clk) begin

        sh[0] <= sin;

        sh[1] <= sh[0];

        sh[2] <= sh[1];

    end
```

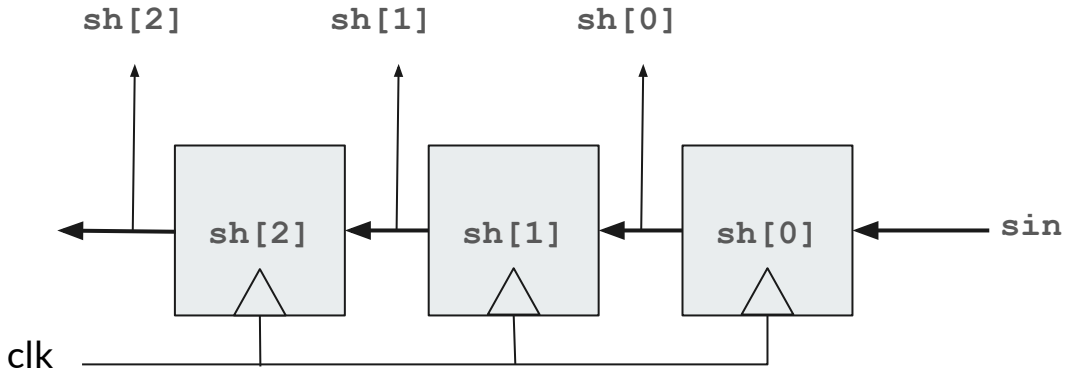
- all the non-blocking assignments (<=) are executed at the same instant, load value at that instant.
  - value of sh[0] is updated by sin
  - sh[0] current value is copied to sh[1]
  - The same is true for sh[1] and sh[2]
  - sh[2] current value is lost, since it is at the end of the shift register.



Three Bit Shift Register

```
dff bit0( sh[0], sin, clk);

dff bit1( sh[1], sh[0], clk);

dff bit2( sh[2], sh[1], clk);
```

- Same behaviour
- Using dff
- Example in the notes



Blocking vs Nonblocking Assignment

Programming languages:

```
1  LED_on = 0;
2  count = count + 1;
3  LED_on = 1;
```

Blocking vs Nonblocking Assignment

Blocking assignments:
The value from r_Test_1 is copied to r_Test_3 in the same clock

```
1  always @(posedge i_clock)
2  begin
3    r_Test_1 = 1'b1;
4    r_Test_2 = r_Test_1;
5    r_Test_3 = r_Test_2;
6  end
```

Blocking vs Non-blocking Assignment

Non-Blocking assignments:

- Previous value before the clock is kept for the next assignments.
- The value from r_Test_1 **needs 3 clocks** to be copied to r_Test_3.

```
1  always @(posedge i_clock)
2  begin
3    r_Test_1 <= 1'b1;
4    r_Test_2 <= r_Test_1;
5    r_Test_3 <= r_Test_2;
6  end
```

Blocking vs Nonblocking Assignment

- <= Nonblocking Assignment
- = Blocking Assignment
- Rule of thumb:

  In Verilog, if you want to create sequential logic use a clocked always block with Nonblocking assignments. If you want to create combinational logic use an always block with Blocking assignments. Try not to mix the two in the same always block.
- https://nandland.com/blocking-vs-nonblocking-in-verilog/
- The Compiler creates registers for the register (or "reg") variables in all non-blocking and some blocking Procedural Assignments of any Always Construct that is sensitive to a rising or falling edge (i.e. posedge or negedge) of a clock.
- https://www.intel.com/content/www/us/en/programmable/quartushelp/14.1/mergedProjects/hdl/vlog/vlog_pro_registers.htm

Questions?

- Next: Unit 4
  - Arithmetic circuits
  - Memory arrays
  - RTL (Register Transfer logic)