REAL-TIME DIGITAL SYSTEMS DESIGN AND VERIFICATION WITH FPGAS
ECE 387 – LECTURE 3

PROF. DAVID ZARETSKY

DAVID.ZARETSKY@NORTHWESTERN.EDU

AGENDA

- Introduction to SystemVerilog
  - Processes, Functions & Tasks
  - Finite State Machines

PROCESSES REVIEW

- An always block may be used to implement sequential logic which has memory elements like flip flops that can hold values.
- SystemVerilog introduces 2 new process types:
  - always – default process for combinational logic
  - always_comb – does not require a sensitivity list
  - always_ff – for clocked signals

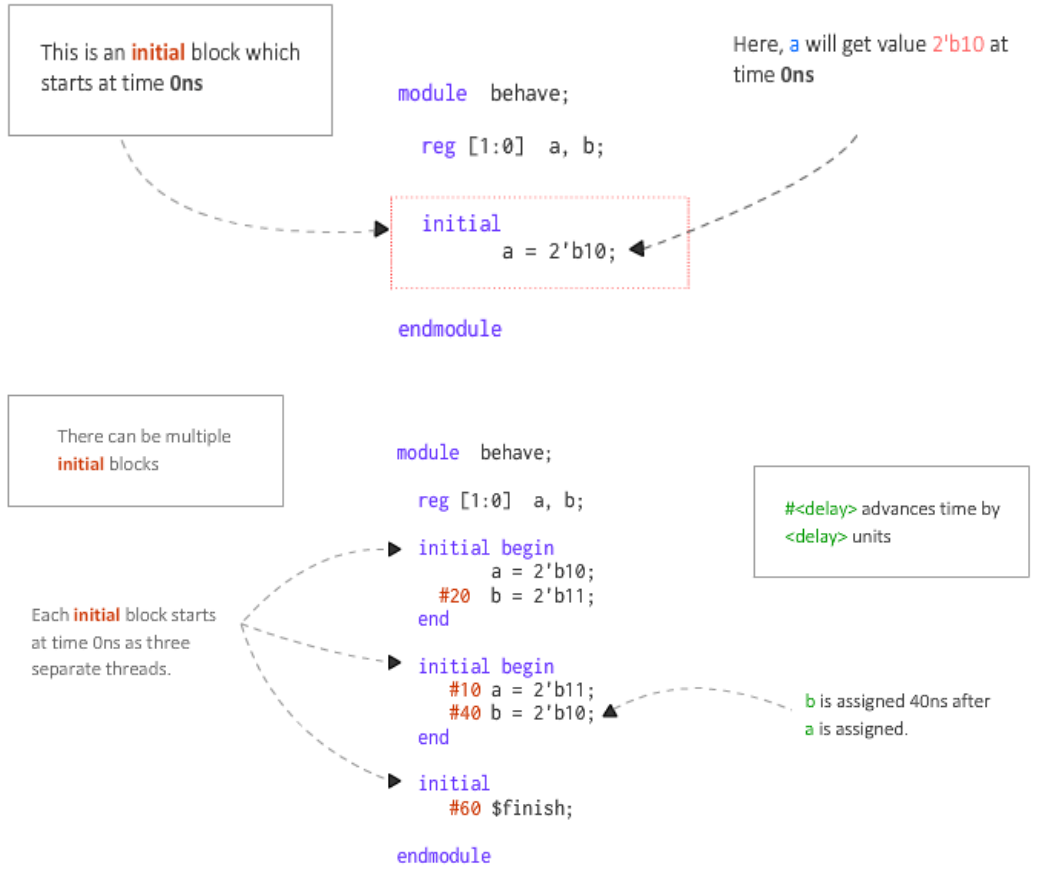
```
always @ (a or b or cin)
begin
   {cout, sum} = a + b + cin;
end
```

```
always_comb
begin
   out = 2'd0;  // default value
   if (in1 == 1)
      out = 2'd1;
   else if (in2 == 1)
      out = 2'd2;
   end
end
```

```
always_ff @(posedge clk or posedge rst)
begin
     if (rst == 1'b1) q <= '0;
     else q <= d1;
end
```

INITIAL STATEMENTS

- Initial blocks are used for initializing variables and arrays
- You can have any number of initial blocks
- Initial blocks are NOT synthesizable
- Mostly used for simulation purposes



FUNCTIONS

- Properties of functions
  - Inlined expressions evaluated at compile time
  - Instantaneous combinational logic
  - May not contain time-controlled statements
  - Can return only one value
- Static functions share the same storage space for all function calls.
- Automatic functions allocate unique, stacked storage for each call, and are re-entrant (e.g. recursive)
- Arguments may be passed
  - by value
  - by reference
  - as constant
  - with default value

```
// function to add two integer numbers.
function int sum(input const int a=0,b=0);
   sum = a + b;
endfunction

function int sum1(const ref int x,y);
  x = x + y;
  return x;
endfunction

initial begin
  x = sum(10,5);
  $display("Value of x = %0d",x);
end
```

```
// re-entrant (automatic) function
function automatic
  int factorial (input [31:0] x);
begin
  if (x >= 2) factorial = factorial(x - 1) * x;
  else factorial = 1;
endfunction: factorial
```

TASKS

- Static tasks share the same storage space for all task calls.
- Automatic tasks allocate unique, stacked storage for each task call, and are re-entrant (e.g. recursive)
- Tasks can calculate multiple results over multiple cycles
- Task may contain
  - parameters
  - input / output / in-out arguments
  - registers
  - events,
  - zero or more behavioral statements

```
// task to add two integer numbers.
task sum(input int a, b, output int c);
  begin
     c = a + b;
  end
endtask

initial begin
  sum(10,5,x);
  $display("Value of x = %0d",x);
end
```

```
// automatic task initializes i=0 each call
task automatic display();
  begin
     integer i = 0;
     i = i + 1;
     $display("i=%0d", i);
  end
endtask
```

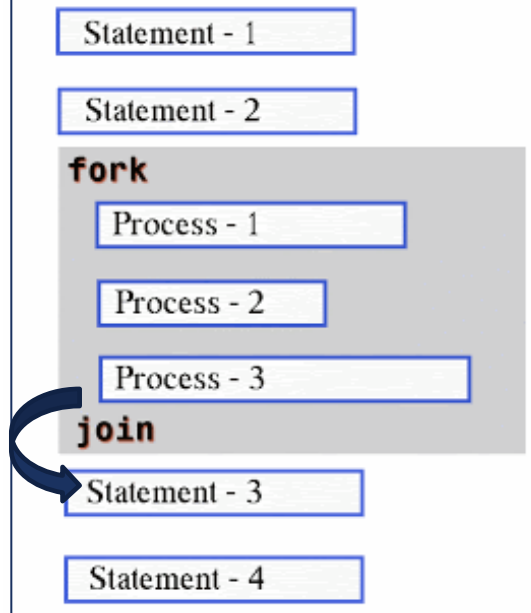
FORK-JOIN

- Fork-Join starts all processes inside in parallel then waits for all processes to complete.
- join is a blocking process

```
initial begin
  $display($time, "Starting processes");
  fork
    // Process-1
    begin
      $display($time,"Process-1 Started");
      #10 $display($time,"Process-1 Finished");
    end

    // Process-2
    begin
      $display($time,"Process-2 Started");
      #5 $display($time,"Process-2 Finished");
    end

    // Process-3
    begin
      $display($time,"Process-3 Started");
      #20 $display($time,"Process-3 Finished");
    end
  join
  $display($time, "All processes completed");
end
```

Waits for all processes to complete

```
0 Starting processes
0 Process-1 Started
0 Process-2 Started
0 Process-3 Started
5 Process-2 Finished
10 Process-1 Finished
20 Process-3 Finished
20 All processes completed
```



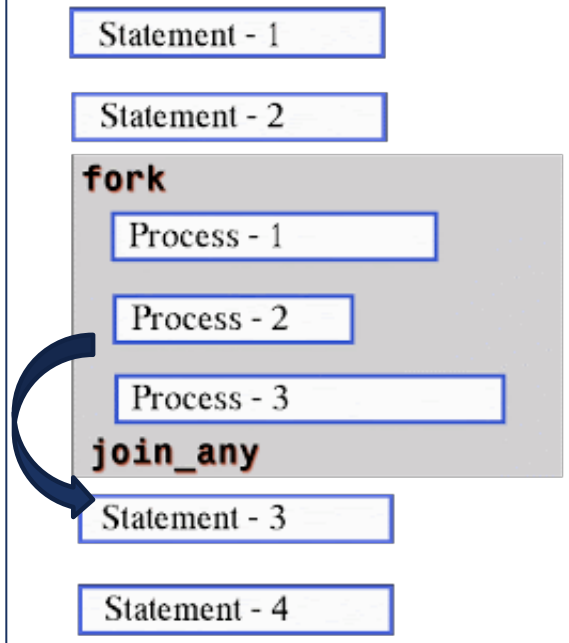
FORK-JOIN-ANY

- Fork-Join-Any starts all processes inside in parallel then waits for any process to complete.
- join_any is a blocking process

```
initial begin
  $display($time, "Starting processes");
  fork
    // Process-1
    begin
      $display($time,"Process-1 Started");
      #10 $display($time,"Process-1 Finished");
    end

    // Process-2
    begin
      $display($time,"Process-2 Started");
      #5 $display($time,"Process-2 Finished");
    end

    // Process-3
    begin
      $display($time,"Process-3 Started");
      #20 $display($time,"Process-3 Finished");
    end
  join_any
  $display($time, "All processes completed");
end
```

Waits for any process to complete

```
0 Starting processes
0 Process-1 Started
0 Process-2 Started
0 Process-3 Started
5 Process-2 Finished
5 All processes completed
10 Process-1 Finished
20 Process-3 Finished
```



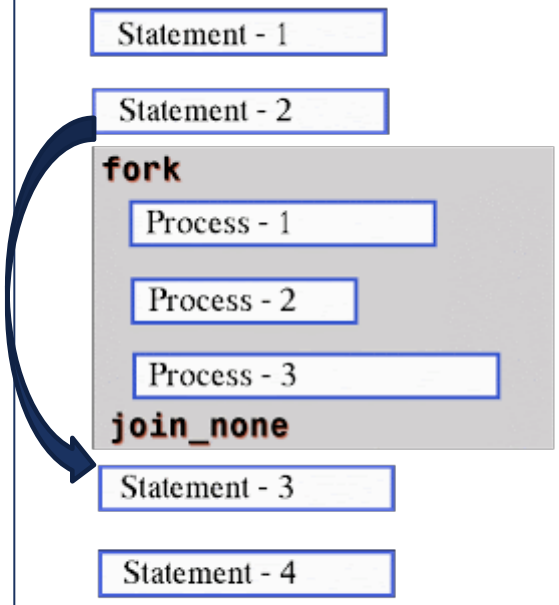
FORK-JOIN-NONE

- Fork-Join-None starts all processes inside in parallel but does not wait for all processes to complete.
- join_none is non-blocking process

Does not wait for processes to complete

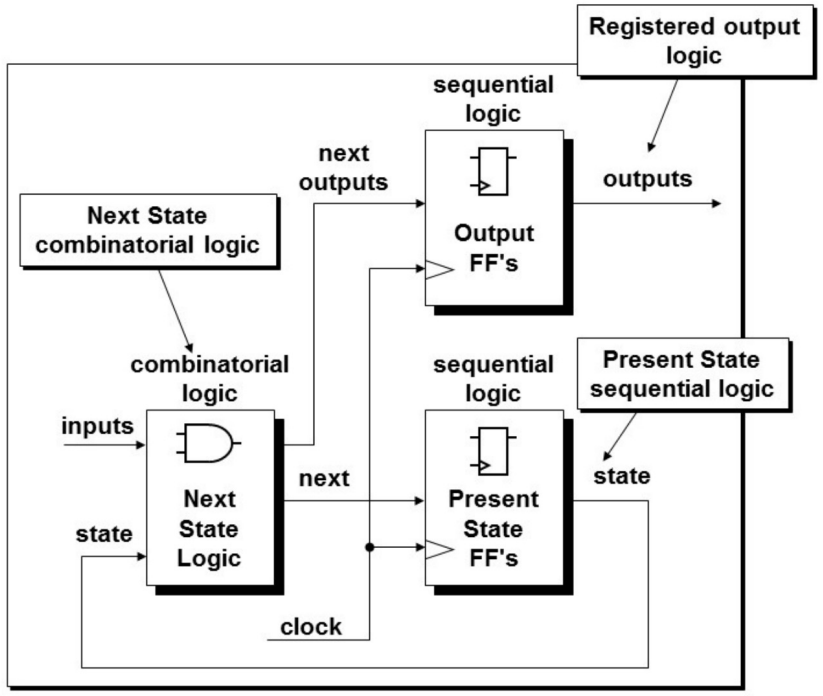
```
0 Starting processes
0 Process-1 Started
0 Process-2 Started
0 Process-3 Started
0 All processes completed
5 Process-2 Finished
10 Process-1 Finished
20 Process-3 Finished
```

```
initial begin
  $display($time, "Starting processes");
  fork
    // Process-1
    begin
      $display($time,"Process-1 Started");
      #10 $display($time,"Process-1 Finished");
    end

    // Process-2
    begin
      $display($time,"Process-2 Started");
      #5 $display($time,"Process-2 Finished");
    end

    // Process-3
    begin
      $display($time,"Process-3 Started");
      #20 $display($time,"Process-3 Finished");
    end
  join_none
  $display($time, "All processes completed");
end
```



FINITE STATE MACHINES – SINGLE PROCESS FSM

- In a single-process FSM, the case statement and all signals are registered under a clocked process (always_ff)




```
always_ff @(posedge clk, negedge rst_n)
  if (!rst_n) begin
    state      <= IDLE;
    <outputs> <= '0;
  end
  else begin
    state <= XXX;
    <outputs> <= '0;
    case (state)
      IDLE : if (go) begin
                rd <= '1;

             end
             else
      READ : begin
                rd <= '1;

             end
      ...
    endcase
  end
```

Present state being registered

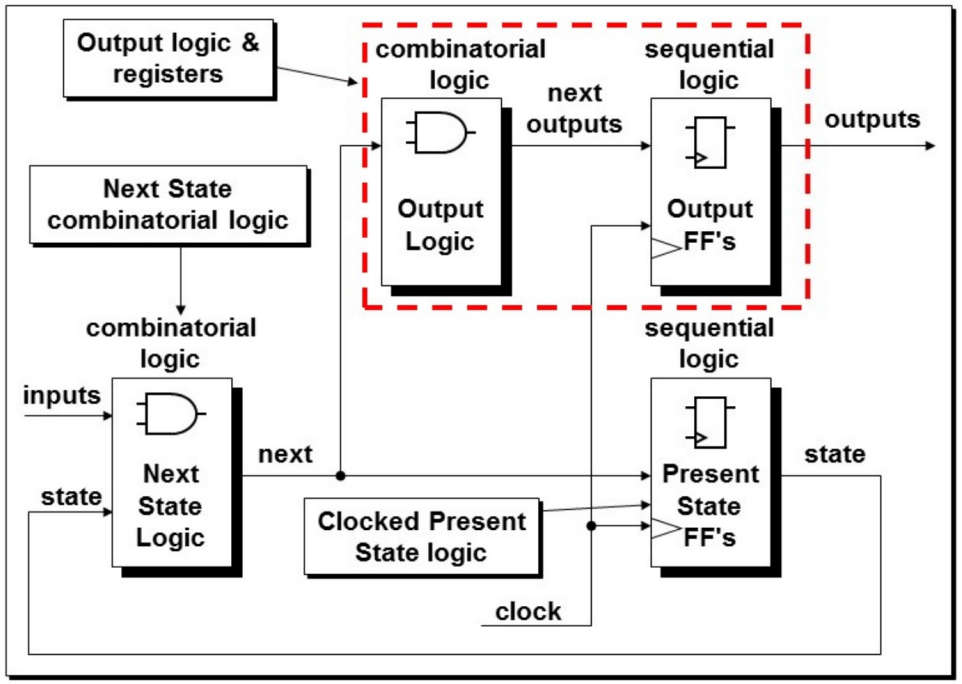
Next outputs being registered

```
state <= READ;

state <= IDLE;


state <= DLY;
```

Next states being registered

FINITE STATE MACHINES – TWO PROCESS FSM

- In a two-process FSM, the combinational and register logic are separated into 2 process (always_comb and always_ff)



Clocked present state logic

```
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) state <= IDLE;
  else        state <= next;
```

always_comb next logic

```
always_comb begin
  next = XXX;
  case (state)
    IDLE : if (go)   next = READ;
           else      next = IDLE;
    READ :           next = DLY;
    ...
  endcase
end
```

always_ff *next*-output logic & registers

```
always_ff @(posedge c
  if (!rst_n) begin
    rd <= '0;
    ds <= '0;
  end
  else begin
    rd <= '0;
    ds <= '0;
    case (next)
      READ : rd <= '1;
      ...
    endcase
  end
```

When `next` is `READ`, output will be `rd<=1`

FSM BEST PRACTICES

- Use case-statement instead of if-then-else
- Assign next_state and output in every state under every conditions, otherwise latches result.
- Specify default values before the FSM in the always_comb process
- Specify default/reset values or don't cares ('X') in default state
  - "default" clause is typically not implemented by the FSM extractor. It's used for "reachability" analysis.
  - It's up to you to generate reset logic which will reset the machine and place it in a known state.

EXAMPLE: SEQUENCE DETECTOR

```
module seq_detector (
input logic clock,
input logic reset,
input logic sequence_in,
output logic detector_out );

// local signals
enum logic [3:0] {S0, S1, S2, S3} state, next_state;
logic detector_c;

always_ff @(posedge clock, posedge reset) begin
  if( reset == 1'b1 ) begin
    state <= S0;
    detector_out <= 1'b0;
  end else begin
    state <= next_state;
    detector_out <= detector_c;
  end
end
```

Combinational process

Declared states & signals

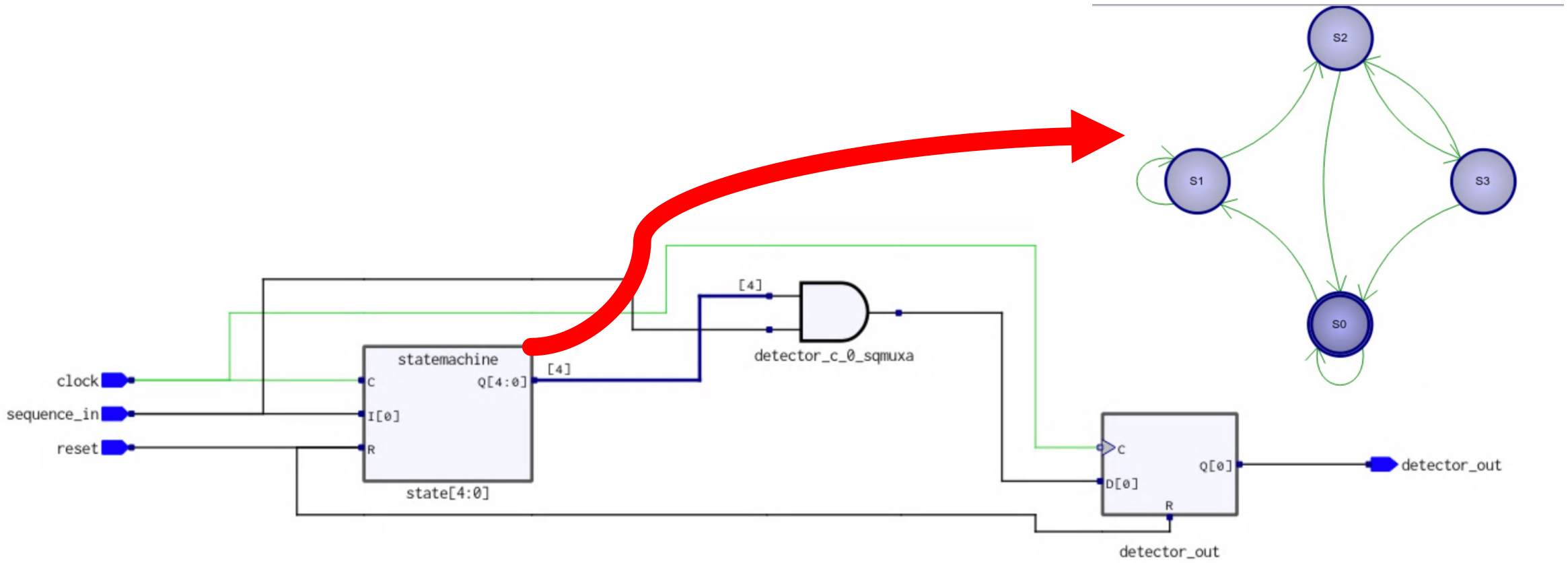
Clocked process & signals

```
always_comb begin
  next_state = state;
  detector_c = 1'b0;
  case ( state )
    S0: // 0000
      if( sequence_in == 1'b1 ) next_state = S1;
    S1: // 0001
      if ( sequence_in == 1'b0 ) next_state = S2;
    S2: // 0010
      if ( sequence_in == 1'b1 ) next_state = S3;
      else next_state = S0;
    S3: // 0101
      if ( sequence_in == 1'b1 ) begin
         next_state = S0;
         detector_c = 1'b1;
      end
      else next_state = S2;
    default:
          next_state = S0;
          detector_c = 1'bX;
  endcase
end

endmodule
```

Default signal assignments

Default case

SEQUENCE DETECTOR SYNTHESIS



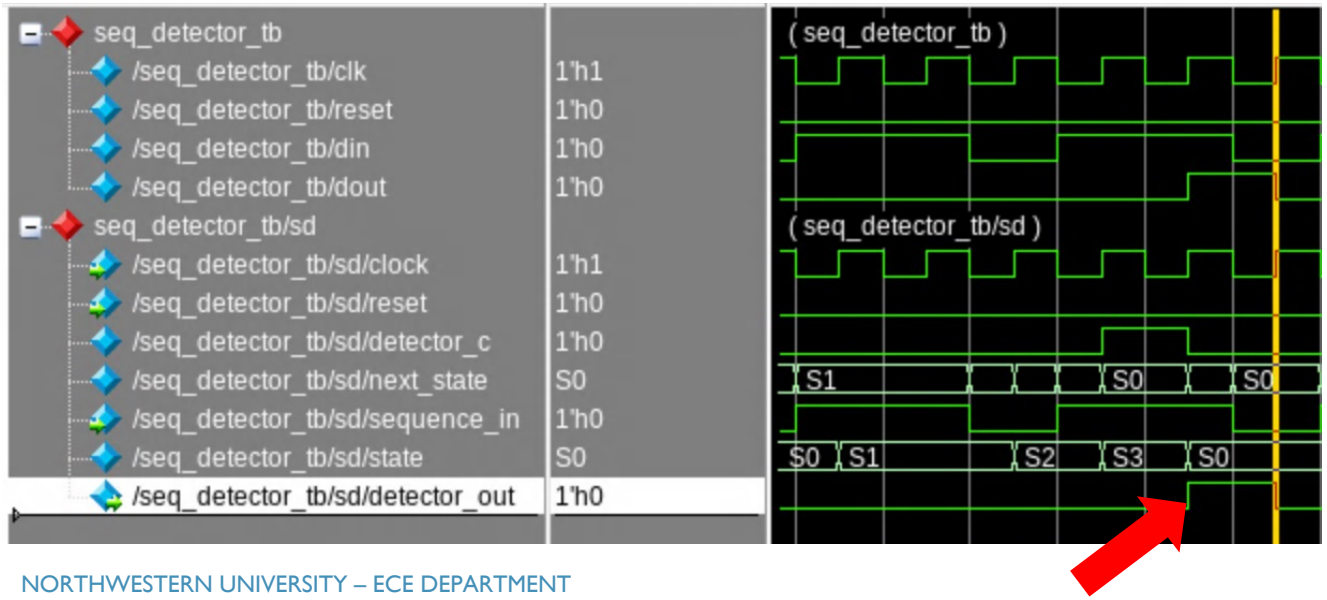
SEQUENCE DETECTOR TESTBENCH

```
`timescale 1ns/1ns

module seq_detector_tb;

// local signals
logic clk = 1'b0;
logic reset = 1'b0;
logic din = 1'h0;
logic dout;

// instantiate sequence detector
seq_detector sd(clk, reset, din, dout);

// clock generator (10 ns)
always begin
  #5 clk = 1'b1;
  #5 clk = 1'b0;
end
```

```
initial begin
   // reset
   #0 reset = 0;
   #10 reset = 1;
   #10 reset = 0;

   // generate random bits every clock cycle (10ns)
   for (int i=0; i < 100; i++ ) begin
       #10 din = $random;
   end

   // instantiate 1011 sequence at the end
  #10 din = 1'b1;
  #10 din = 1'b0;
  #10 din = 1'b1;
  #10 din = 1'b1;
  #10 din = 1'b0;
  #50;

  $stop;
end

endmodule
```

SEQUENCE DETECTOR SIMULATION

- Create a seq_detector_sim.do file.
- To simulate, run:
  - `vsim –do seq_detector_sim.do`




```
setenv LMC_TIMEUNIT -9
vlib work
vmap work work

# compile
vlog -work work "../sv/seq_detector.sv"
vlog -work work "../sv/seq_detector_tb.sv"

# run simulation
vsim -classdebug -voptargs=+acc +notimingchecks -L work
work.seq_detector_tb -wlf seq_detector.wlf

# wave
add wave -noupdate -group seq_detector_tb
add wave -noupdate -group seq_detector_tb -radix hexadecimal
/seq_detector_tb/*
add wave -noupdate -group seq_detector_tb/sd
add wave -noupdate -group seq_detector_tb/sd -radix hexadecimal
/seq_detector_tb/sd/*

run -all
```

FSM STATE ENCODINGS

- Binary (Sequential) –
  - Encoded as Consecutive Binary Numbers
  - Small number of used flip-flops
  - Potentially complex transition functions leading to slow implementations
- One-Hot
  - Only one bit is active per state
  - Number of flip-flops = number of states
  - Simple and fast transition functions
  - Preferable coding technique in FPGAs
  - Faster and Low Power
- Gray Encoding
  - One bit switches between consecutive states.
  - Minimizes hazards and glitches

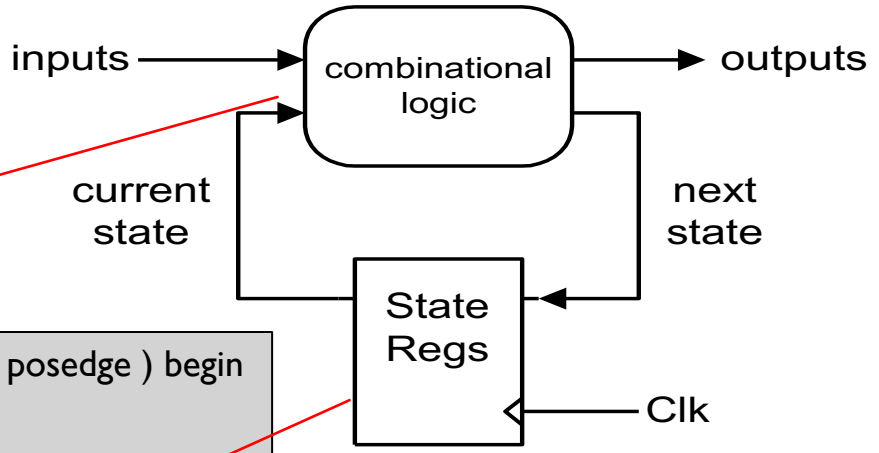
State	Binary	One-Hot	Gray
S0	000	10000000	000
S1	001	01000000	001
S2	010	00100000	011
S3	011	00010000	010
S4	100	00001000	110
S5	101	00000100	111
S6	110	00000010	101
S7	111	00000001	100

```
typedef enum logic [2:0] {S0=3'b000, S1=3'b001, S2=3'b011, S3=3'b010, S4=3'b110, … } gray_states
```

HIERARCHICAL DESIGN SYNCHRONIZATION

- Separates the FSM into combinational and clocked processes
- Use **start** and **done** signals for synchronization
- Parent module initiates start / waits on done

```
always_comb begin
  next_state = state;
  z_c = z;
  done_c = done;
  case ( state ) begin
    S0 :
      if ( start = 1'b1 ) begin
          done_c = 1'b0;
          next_state = S1;
       end else begin
          next_state = S0;
       end
    S1:
       z_c = x + y + z;
       done_c = 1'b1;
       next_state = S0;
     :
    default:
       done_c <= X;
       next_state <= S0;
  end
end
```




```
always_ff( posedge clock, posedge ) begin
  if ( reset = 1'b1 ) then
     state <= S0;
     done <= '0';
     z <= 0;
  else begin
     state <= next_state;
     done <= done_c;
      z <= z_c;
  end
end
```

NEXT…

- Assignment 1: Fibonacci
- Behavioral Designs
- Testbench
- Simulation