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)
Incomplete IF statement – causes Latches

```verilog
module mux2(
    input a,
    input b,
    input sel,
    output reg y
);

    always @ (a, b, sel)
    begin
        if (sel) // incomplete IF statement
            y = a;
        end // no ELSE part
endmodule
```

**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

```verilog
module dtype(
    input clk,
    input d,
    output reg q // reg keyword required for q
);

// process (clk)
begin
    // if clk'event and clk = '1' then
    // q <= d;
    always @(posedge clk)
    q <= d;
endmodule
```
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

```verilog
module counter2(
  input clk,
  input clr, // async clear (reset)
  output reg [3:0] q // reg keyword required for q
);

always @ (posedge clk, posedge clr)
  if (clr)
    q <= 0;
  else if (q == 13) // can read output
    q <= 4'h0;
  else
    q <= q + 1'b1; // to stop truncation warning
endmodule
```
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)

```
module counter_en(
  input  clk,
  input  en,
  input  reset,
  output reg [3:0] q,
  output  tc
);

always @(posedge clk, posedge reset)
begin
  if (reset) // async reset
    q <= 4'b0;
  else if (en) // sync en
    q <= q + 1'b1;
end
assign tc = (q == 4'hf);
endmodule
```
Synthesis (4 flip-flops)
Simulation and Implementation

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Check</th>
<th>Worst Case</th>
<th>Best Case</th>
<th>Timing</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slack</td>
<td>Achievable</td>
<td>Errors</td>
<td>Score</td>
<td></td>
</tr>
<tr>
<td>NET &quot;clk_BUFGP/IBUFG&quot; PERIOD = 20 ns HIGH</td>
<td>SETUP</td>
<td>18.626ns</td>
<td>1.374ns</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50%</td>
<td>HOLD</td>
<td>0.444ns</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>MINPERIOD</td>
<td>18.270ns</td>
<td>1.730ns</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

All constraints were met.
Creating slower clocks

```verilog
module one_MHZ(
    input  clk_fpga,  // assume 100MHz
    input  reset,
    output reg [3:0] cnta
);

reg [6:0] count;  // count from 0 to 99

wire clk_en;

always @ (posedge clk_fpga, posedge reset)
    if (reset)
        count <= 0;
    else if (count == 99)
        count <= 0;
    else
        count <= count + 1'b1;

assign clk_en = (count == 99);

always @ (posedge clk_fpga, posedge reset)
    if (reset)
        cnta <= 4'b0;
    else if (clk_en)  // clock this counter at 1MHz
        cnta <= cnta + 1'b1;

endmodule
```
Shift Register

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

```vhdl
module shift_reg(
  input  clk,
  input  load,
  input  left_right_n,
  input  [3:0] d,
  output reg [3:0] q
);

always @ (posedge clk) // all operations synchronous
  if (load)
    q <= d;       // sync load
  else if (left_right_n)
    q <= { q[2:0], 1'b0 }; // shift left
  else
    q <= q >> 1;   // shift right
endmodule
```
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

```
module lfsr(
    input clk,
    input reset,
    output q
);

    reg [5:0] shift;
    wire xor_sum;

    assign xor_sum = shift[1] ^ shift[4]; // feedback taps

    always @(posedge clk)
        if (reset)
            shift <= 6'b111111; // initialize LFSR
        else
            shift <= { xor_sum, shift[5:1] }; // shift right

    assign q = shift[0]; // output of LFSR
endmodule
```
LFSR - Simulation
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

```
Inputs                      Next State Logic
                                      State Memory
                                      Current State
                  Clock                      Outputs
                                      Reset
                                      Outputs
```

Jim Duckworth, WPI
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

- **S0**: C=0, SW=1
- **S1**: C=0, SW=0
- **S2**: C=0, SW=1
- **S3**: C=1, SW=0

- **RESET=1**

The diagram shows a state transition for a simple state machine with inputs SW and C.
State Machine Coding Style

module sml(
  input clk,
  input sw,
  input reset,
  output reg c
);

  parameter [1:0] s0 = 0, s1 = 2'd01, s2 = 2, s3 = 3;

  reg [1:0] current_state, next_state;

  // state-machine flip-flops (sequential logic - use non-blocking)

  // next state logic (combinational - use blocking)

  // output logic (could be part of next state logic)

endmodule
State Memory (flip-flops)

```verilog
define [1:0] s0 = 0, s1 = 2'b01, s2 = 2, s3 = 3;

reg [1:0] current_state, next_state;

// state-machine flip-flops (sequential logic - use non-blocking)
always @(posedge clk, posedge reset)
  if (reset)
    current_state <= s0;
  else
    current_state <= next_state;
```

Next State (combinational logic)

```vhdl
-- next state logic (combinational - use blocking)
always @(current_state, sw)
case (current_state)
  s0: begin
    c = 1'b0;
    if (sw)
      next_state = s1;
    else
      next_state = s0;
  end
  s1: begin
    c = 1'b0;
    if (sw)
      next_state = s2;
    else
      next_state = s1;
  end
  s2: begin
    c = 1'b0;
    if (sw)
      next_state = s3;
    else
      next_state = s2;
  end
  s3: begin
    c = 1'b1;
    if (sw)
      next_state = s0;
    else
      next_state = s3;
  end
endcase
```
SM1 – Verilog Code

```verilog
module sm1(
    input clk,
    input sw,
    input reset,
    output c
);

parameter [1:0] s0 = 0, s1 = 2'b01, s2 = 2, s3 = 3;

reg [1:0] current_state, next_state;

// state-machine flip-flops (sequential logic - use non-blocking)
always @(posedge clk, posedge reset)
    if (reset)
        current_state <= a0;
    else
        current_state <= next_state;

// next state logic (combinational - use blocking)
always @(current_state, sw)
    case (current_state)
        s0: if (sw)
            next_state = s1;
        else
            next_state = s0;
        s1: if (sw)
            next_state = s2;
        else
            next_state = s1;
        s2: if (sw)
            next_state = s3;
        else
            next_state = s2;
        s3: if (sw)
            next_state = s0;
        else
            next_state = s3;
    endcase

// output logic (could be part of next state logic)
assign c = current_state == s3;
endmodule
```
Synthesis Report

Synthesizing Unit <sm1>.
Related source file is "C:\ece3829\sm1\sm1.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 |

Inferred 1 Finite State Machine(s).

---

Analyzing FSM <MFsm> for best encoding. Optimizing FSM <FSM_0> on signal <current_state[1:2]> with gray encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>
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

![HDL Options for FSM Encoding](image-url)
Same Verilog with One-Hot Encoding

Optimizing FSM <FSM_0> on signal <current_state[1:4]> with One-Hot encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0001</td>
</tr>
<tr>
<td>01</td>
<td>0010</td>
</tr>
<tr>
<td>10</td>
<td>0100</td>
</tr>
<tr>
<td>11</td>
<td>1000</td>
</tr>
</tbody>
</table>