Synthesizable Verilog

Dr. Paul D. Franzon

Outline
1. Combinational Logic Examples.
2. Sequential Logic
3. Finite State Machines
4. Datapath Design

References
1. Smith and Franzon, Chapters 5, 8, Appendix A (on this web site)
2. M. Smith, Chapter 10, 11
3. D. Smith, Chapters 4, 6, 8.
D. Smith, Example 9.4 (modified): Shift Register:

```verilog
module shift_reg(clock, reset, load, sel, inload, shftreg);
  input clock, reset, load;
  input [4:0] sel;
  input [3:0] inload;
  output [3:0] shftreg;

  reg [3:0] shftreg;
  reg shftleft, shftright;
  wire zero, ten, both;

  always@(posedge clock)
    if (reset) shftreg <= 0;
    else if (load) shftreg <= inload;
    else if (shftleft) shftreg <= shftreg << 1;
    else if (shftright) shftreg <= shftreg >> 1;

  always@(sel)
    begin
      shftleft = 1'b0;  shftright = 1'b0;
      casex (sel) //synopsys full_case
        5'b000x1 : shftleft = 1'b1;
        5'b0001x : shftright = 1'b1;
        default : shftleft = 1'b0;
      endcase
    end
  assign zero = (shftreg == 4'd0);
  assign ten  = (shftreg == 4'd10);
  assign both = zero & ten;
endmodule
```
... Shift Register

1. Sketch circuit being described.

2. Consider the `always@(posedge clock)` procedural block
   - Is the reset synchronous (synchronized with the clock) or asynchronous?
   - Every variable assigned to in this procedural block becomes a ......

3. Consider the `always@(sel)` procedural block
   - What is a `casex()` statement?
   - Why are no latches inferred?

The `//synopsys full_case` tells Synopsys that all alternatives have been specified, thus preventing latches. It is an example of a Synopsys Directive. The ‘default’ and pre-assignment works to prevent unintentional latches also.
Revision - Design Steps

1. Understand the specification fully
   - Preferably have a C or Verilog code that captures the overall behavior

2. Design the hardware
   - The closer it looks like the final netlist, the better your synthesis results generally are.
   - At least explicitly identify ALL OF THE FOLLOWING:
     - Registers
     - Combinational logic blocks and their functions
     - Interconnecting wires
   - Think carefully about the following:
     - What happens on power up?
       - On power-up all signals take random values (Verilog does not simulate this
         … on power up, all signals are initialized as ‘x’ for unknown)
       - Is the behavior safe or do you need a reset signal?
     - What happens before and after each clock edge
...Design Steps

3. Write your Verilog:
   - All register outputs are assigned within `always@(posedge clock)` blocks
     - Also place register input logic within these blocks
     - **EVERY VARIABLE ASSIGNED WITHIN SUCH A BLOCK BECOMES THE OUTPUT OF A FLIP-FLOP**
   - Combinational logic can be designed:
     - As input logic to the registers, or,
     - as its own Procedural block, or,
     - using continuous assignment, or,
     - structurally (see later)

4. Write your verification test fixture
   - Make sure your verification is as complete as possible (see later section)

5. Synthesize your design
   - Fix all Errors, understand or fix (as appropriate) all Warnings
   - Make sure timing is met (‘slack (met)’)

6. Conduct post-synthesis verification
   - Especially, timing and function
Combinational Logic Approaches

The following types of examples are given:

- Logic built in procedural blocks
  - Encoders, decoders, selectors (multiplexors)
  - Some other useful constructs
- Logic built using continuous assignment
  - Gate-level logic, multiplexors
- Logic specified directly in the target library
Priority Encoder

Truth Table:

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Y1</th>
<th>Y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Capture in Verilog using if-then-else or a casex statement:

```verilog
input [2:0] A;
input       F;
reg   [1:0] Y;
always@(A or F)
    begin
        if (A[2]) Y = 2’b11;
        else if (A[1]) Y = {1’b0,F};
        else if (A[0]) Y = {F,1’b0};
        else Y = 2’b00;
    end
```
...Priority Encoder

begin

  casex (A) //synopsys full_case
  3'b1xx : Y = 2'b11;
  3'b01x : Y = {1'b0,F};
  3'b001 : Y = {F,1'b0};
  default : Y = 2'b00;

endcase

end

case and casex statements are much like the C ‘switch’ statement
  Only one alternative is selected though
casex statement allows some of the alternatives to contain ‘don’t cares’
Non-Priority Encoder

Truth Table:

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Y1</th>
<th>Y0</th>
</tr>
</thead>
<tbody>
<tr>
<td>all others</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Capture in a Verilog case statement:

```verilog
input [2:0] A;
input F;
reg [1:0] Y;
always@(A or F)
    case (A)
        3'b100 : Y = 2'b11;
        3'b010 : Y = {1'b0, F};
        3'b001 : Y = {F, 1'b0};
        default : Y = 2'b00;
    endcase
```
...Procedural Blocks

//synopsys parallel_case tells Synopsys we only ever expect one of the case alternatives to be true, leading to non-priority logic which is usually smaller.

Q. Why do the examples on the previous page describe priority logic while the example on this page describes non-priority logic?

Q. How are latches prevented in these cases?
Exercise

Implement the following logic:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>x</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>xxx11</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>xxx01</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
Other Procedural Block Examples

Loops can be used (with care) to specify logic:

```vhdl
integer i;
reg [7:0] A;

always@(A)
begin
  OddParity = 1'b0;
  for (i=0; i<=7; i=i+1)
    if (A[i]) OddParity = ~OddParity;
end
```

Is the following code fragment synthesizable?

```vhdl
integer i, N;
parameter N=7;
reg [N:0] A;
always@(A)
begin
  OddParity = 1'b0;
  for (i=0; i<=N; i=i+1)
    if (A[i]) OddParity = ~OddParity;
end
```
"…Other Procedural Block Examples"

The loop constructs (for, repeat, while) can sometimes be used to specify combinational logic

- For example, to traverse an array of bits
- The loop determination must be a constant
  - If it’s a variable, the size of the loop can not be determined statically and thus the loop can not be synthesized

Generally, the following statements are rarely used in synthesizable designs:
wait, forever, disable, [procedural] assign/deassign, force, release

- forever is unsynthesizable in Synopsys under all uses.
Logic Built Using Continuous Assignment

Example 1. Multiplexor built via indexing:

```verbatim
input [2:0] bc;
input A, B, C;
input [3:0] D, E;
wire [7:0] codes;
wire final;

assign codes = {1'b1, {2{C}}, ~&D, A^B, !A, A&B, D>E};
assign final = codes[bc[2:0]];
```

Draw this logic:

Notes:

- Multiplexor-based design is often smaller and faster than logicgate based design in CMOS. Often structural code has to be used to force this implementation style.
- Explicit coding of width and msb of bc in `final = codes[bc[1:0]]`;
Continuous Assignment Logic

Examples:

1.

    wire          foo;
    wire   [3:0]  bar;

    assign foo = select ? &bar : ^bar;

2.

    wire   [7:0] mush;
    tri    [3:0] bus;

    assign bus = enable ? mush[4:1] : 4’bz;

• A continuous assignment logic style must be used in Synthesis for tri-state logic.
Registers

Some Flip Flop Types:

```plaintext
reg Q0, Q1, Q2, Q3, Q4;

// D Flip Flop
always@(posedge clock)
  Q0 <= D;

// D Flip Flop with asynchronous reset
always@(posedge clock or negedge reset)
  if (!reset) Q1 <= 0;
  else Q1 <= D;

// D Flip Flop with synchronous reset
always@(posedge clock)
  if (!reset) Q2 <= 0;
  else Q2 <= D;

// D Flip Flop with enable
always@(posedge clock)
  if (enable) Q3 <= D;

// D Flip Flop with synchronous clear and preset
always@(posedge clock)
  if (!clear) Q4 <= 0;
  else if (!preset) Q4 <= 1;
  else Q4 <= D;
```

Note:

Registers with asynchronous reset are smaller than those with synchronous reset
+ don’t need clock to reset
BUT it is a good idea to synchronize reset at the block level to reduce impact of noise.
Finite State Machines can be classified by the following attributes:

- **Moore or Mealy type outputs**

**Moore Outputs**
Outputs depend solely on state vector (generally, a Moore FSM is the simplest to design)

**Mealy Outputs**
Outputs depend on inputs and state vector (only use if it is significantly smaller or faster)
… FSM Types

- **State Vector Encoding**
  - Minimal encoding
    - Minimum number of bits
  - Minimum, sequential encoding
    - Minimum number of bits and states in sequence
      - Does not necessarily optimize ‘next state logic’ size
  - Gray encoding
    - State bit changes by only one bit between sequential states
      - Minimizes switching activity in state vector register
  - One-hot encoding
    - One bit per state
      - Usually gives fastest ‘next state’ logic

**Example:** 7-state FSM, states S0 … S7:
… FSM Types

• Resets:
  ● Reset usually occurs only on power-up and when someone hits the ‘reset’ button
  ● *Asynchronous* reset:
    ◦ FSM goes to reset state whenever reset occurs
  ● *Synchronous* reset:
    ◦ FSM goes to reset state on the next clock edge after reset occurs
  ● Asynchronous reset leads to smaller flip-flops while synchronous reset is ‘safer’
    (noise on the reset line is less likely to accidently cause a reset).

• *Fail-Safe Behavior*:
  ● If the FSM enters an ‘illegal’ state due to noise is it guaranteed to then enter a
    legal state?
    ◦ ‘Yes’ is generally desirable
… FSM Types

- **Sequential Next state or output logic**
  - Usually, these blocks are combinational logic only
  - However, can place sequential logic (e.g. a counter, or a toggle-flip-flop) in these blocks if it is advantageous
  - **AVOID DOING THIS AS MUCH AS YOU CAN UNLESS YOU ARE REALLY SURE ABOUT WHAT YOU ARE DOING**
    - Sequential next state or output logic can get very confusing to design and debug
- **Registered or Unregistered Outputs**
  - Do not register the outputs unless you need to ‘deglitch’ the outputs (for example, for asynchronous handshaking - combinational logic has to be assumed to be glitchy) or are pipelining the control
  - e.g.

![Diagram](attachment:image.png)
Example - Drag Racing Lights

At the start of a new race (‘car’), go through the Red-Yellow-Green sequence:

**Moore Machine:**

Nomenclature: inputs
- car?

On states: red yellow green

**Mealy Machine:**

Nomenclature: inputs / outputs
- car? / red yellow green
Drag Light Controller ... Verilog

module traffic_light_controller (clock, reset, car, red, yellow, green);

input clock;
input reset;
input car;
output red, yellow, green;

parameter [1:0] // synopsys enum states
S0 = 2'b00,
S1 = 2'b01,
S2 = 2'b10,
S3 = 2'b11;

reg [1:0] /* synopsys enum states */ current_state, next_state;// synopsys state_vector current_state
reg red, yellow, green;

//-- Sequential Logic ----*/
always@(posedge clock or negedge reset)
  if (!reset) #1 current_state <= S0;
  else #1 current_state <= next_state;

/* next state logic and output logic */
always@(current_state or car)
begin
  case (current_state) // synopsys full_case parallel_case
    S0: begin
      red = 1;
      if (car) next_state = S1
      else next_state = S0;
    end
    S1: begin
      yellow = 1;
      next_state = S2;
    end
    S2 : begin
      green = 1;
      next_state = S0;
    end
    default : next_state = S0;
  endcase
end
endmodule
**FSM Verilog Notes**

1. Code each FSM by itself in one module.
2. Separate Sequential and Combinational Logic
3. Is this reset Synchronous or Asynchronous?
   - Asynchronous usually results in less logic (reset is actually synchronized when it enters the chip).
4. Note use of Synthesis directives:
   - //synopsys enum states and //synopsys state_vector current_state tell Synopsys what the state vector is.
   - Why can we state //synopsys full_case parallel_case for FSMs?
5. How to we prevent accidently inferring latches?
FSM Synthesis...Script Extract...

replace_synthetic /* map onto generic logic − could use compile */
unigroup −all −flatten /* create one single level */

/* optimize the FSM */
set_fsm_state_vector {current_state_reg[0] current_state_reg[1]}
set_fsm_encoding {}
set_fsm_encoding_style auto /*auto=encode state vector for min. area*/
group −fsm −design_name traffic_fsm
current_design = traffic_fsm
report_fsm
extract
report_fsm
set_fsm_minimize true
compile

/* optimize the design at the top level */
current_design = traffic_light_controller
compile
FSM State Encoding Options

Can either do `by hand' in Verilog source code or by reassigning states in Synopsys:

- Binary or Sequential (minimal) encoding:
  
  State 0 = 000
  State 1 = 001, etc.

- Gray encoding gives the minimum change in the state vector between states:
  
  State 0 = 000
  State 1 = 001
  State 2 = 011, etc

  ➤ Reduces state transition errors caused by asynchronous inputs changing during flip-flop set-up times.
  ➤ Minimizes power consumed in state vector flip-flops
  
  Synopsys: set_fsm_encoding_style gray //+ See manual

- One-hot encoding assigns one flip-flop per state:
  
  State 0 = 0001
  State 1 = 0010
  State 2 = 0100, etc

  ➤ Fastest but largest design
  
  Synopsys: set_fsm_encoding_style one_hot

- Custom: Assign states by hand in Verilog of Synopsys
Registering FSM Outputs

Sometimes useful to **register** the outputs of FSMs:

- Necessary when these outputs are interfacing asynchronously with other modules or off-th-chip
  - e.g. RAS and CAS outputs for a memory interface
- Useful if delays in output combinational logic are making it hard to meet timing requirements in the module they are connected to.
  - assumes flip-flop t_cp_Q is faster (might not be - look in library sheets)

**Example**

```verbatim
always@(posedge clock)
begin
  red = int_red;
  yellow = int_yellow;
  green = int_green;
end

...case (current_state)
  S0: begin  int_red=1;

  Note: changes now delayed one clock when compared with previous version
```
Final Exercise

Linear Feedback Shift Register

- Used to generate a pseudo-random sequence of numbers
  - ‘pseudo-random’ because the number sequence cycles
- Next number obtained by ‘hashing’ the previous number with a bunch of XOR gates
  - With the right design, no number is ever repeated (up to \(2^N\), where \(N = \# \text{ bits}\))

Applications

- Random number generators
- Built In Self Test (BIST)
  - Generates test vector sequence for production test of a block of logic
- Used in data encryption, checksum generation, and data compression techniques
Final Exercise

What is the following Verilog building? (D. Smith, Example 7.10)

```verilog
module LFSR_8BIT (Clock, Reset, Y);

input Clock, Reset;
output [7:0] Y;

integer N [1:7];
parameter [7:0] Taps = 8'b10001110;
reg Bits0_6_Zero, Feedback;
reg [7:0] LFSR_Reg, Next_LFSR_Reg;

always@(posedge clock or negedge Reset)
  begin: LFSR_Reg
    if (!Reset)      LFSR_Reg <= 8'b0;    else      LFSR_Reg <= Next_LFSR_Reg;
  end
```
always@(LFSR_Reg)
begin: LFSR_Feedback
    Bits0_6_Zero = ~ | LFSR_Reg[6:0];
    Feedback = LFSR_Reg[7] ^ Bits0_6_Zero;
    for (N=7; N>=1; N=N-1)
        if (Taps[N-1] == 1)
            Next_LFSR_Reg[N] = LFSR_Reg[N-1] ^ Feedback;
        else
            Next_LFSR_Reg[N] = LFSR_Reg[N-1];
    Next_LFSR_Reg[0] = Feedback;
end

assign Y = LFSR_Reg;
endmodule
Features of Design

- Parameterized Design:
  - Details of design can be changed by changing Taps (see example)

- Employing Functions and Tasks
  - See examples provided

- `always@(LFSR_Reg) block`
  - Note ‘chain’ of logic in this block
    - Intermediate results used on road to final result
Summary

- Combinational Logic implied by `always@(x ...)` or `assign` statements
  - Try to build unprioritized logic in a case statement when possible
  - Use structural Verilog (assign) to build muxes for smallest fastest design

- Continuous Assignment Logic
  - Often leads to most efficient implementation
  - Only approach to building tri-state buses

- Finite State Machines
  - Describe and optimize separately from datapath and random logic
  - Consider style of implementation for best design