Section 5

Finite State Machines
Administrivia

- **Lab 5**: Report due next Wednesday (5/1) @ 2:30 pm, demo by last OH on Friday (5/3), but expected during your assigned slot.
  - ⚠️ This lab is harder than previous labs ⚠️

- **Lab 6**: Report due 5/8, demo by last OH on 5/10.
  - ⚠️ This lab is a LOT harder than Lab 5 ⚠️
New SystemVerilog Commands
New SystemVerilog Commands

- **enum** – create an enumerated type with a restricted set of named values.
  - Basic usage: `enum <original type> {<name_list>} <vars>;
  - `<original type>` must be wide enough to support the length of `<name_list>;` if omitted, defaults to `int` type.
  - By default, names in the `<name_list>` are assigned consecutive values starting from 0.
    - Can explicitly assign values using `name=<value>` syntax.
New SystemVerilog Commands

- **enum** – create an enumerated type with a restricted set of named values.
  - Basic usage: `enum <original type> {<name_list>} <vars>;
  - `<original type>` must be wide enough to support the length of `<name_list>`; if omitted, defaults to `int` type.
  - By default, names in the `<name_list>` are assigned consecutive values starting from 0.
    - Can explicitly assign values using `name=<value>` syntax.
- Example: `enum logic [1:0] {S0, S1, S11=2'b11} ps, ns;
  - S0 assigned 2'b00, S1 assigned 2'b01.
  - Two variables declared that can only take on the values S0, S1, and S11 (no 2'b10).
New SystemVerilog Commands

- Ternary operator – shorthand for an if-else statement using the syntax `<cond> ? <then> : <else>` (same syntax as C).
  - Same syntax as C/C++.
  - Never necessary to use, just results in more compact code.
  - Very useful in combinational logic for next state and output logic.
New SystemVerilog Commands

- Ternary operator – shorthand for an `if-else` statement using the syntax `<cond> ? <then> : <else>` (same syntax as C).
  - Same syntax as C/C++.
  - Never necessary to use, just results in more compact code.
  - Very useful in combinational logic for next state and output logic.

- Examples:
  - `case` (ps)
    - `S0:  ns = w ? S1  : S0;`
    - `S1:  ns = w ? S11 : S0;`
    - `S11: ns = w ? S11 : S0;`
    - `endcase`
  - `assign` `HEX0 = SW[0] ? leds : 7'b1111111;`
Finite State Machine Implementation
FSM Implementation Notes

- The **state diagram design** is *by far* the most important part! The SystemVerilog implementation process is fairly mechanical.
  - Best to implement from scratch rather than tweak a broken initial design.
The state diagram design is by far the most important part! The SystemVerilog implementation process is fairly mechanical.

- Best to implement from scratch rather than tweak a broken initial design.

Module design notes:
- Must have a clock input (e.g., clk, clock, CLOCK_50) for sequential elements.
- Should have a reset input (e.g., rst, reset) for "initialization."
- Must have a present state (ps); recommended to also have a next state (ns).
FSM Design Pattern

1) // State Encodings and Variables
   a) enum to define ps and ns

2) // Next State Logic (ns)
   a) always_comb or assign with blocking assignments (=)

3) // Output Logic
   a) assign or always_comb with blocking assignments (=)
   b) Mealy-type output example: assign out = (ps == S1) & in;

4) // State Update Logic (ps) – including reset
   a) always_ff with non-blocking assignments (<=)
Exercise 1

- The following FSM represents a Red Light, Green Light game, where a player is only allowed to move forward ($M=1$) when the light is green ($L=1$). Here, the player wins (output $W=1$) after successfully moving twice; moving when the light is red ($L=0$) results in returning to the start.

  - Implement this system in a module called light_game.
Exercise 1 (Solution)

- Module outline

```verilog
module light_game (input logic clk, reset, M, L, output logic W);
endmodule // light_game
```
Exercise 1 (Solution)

- State encodings and variables

```verilog
module light_game (input logic clk, reset, M, L, output logic W);

    enum logic [1:0] {Start, Mid, Win} ps, ns;

endmodule  // light_game
```
Exercise 1 (Solution)

- Next state logic

```verilog
module light_game (input logic clk, reset, M, L, output logic W);

    enum logic [1:0] {Start, Mid, Win} ps, ns;

    always_comb
        case (ps)
            Start: ns = (L & M) ? Mid : Start;
            Mid:   ns = (L & M) ? Win : (M ? Start : Mid);
            Win:   ns = M ? Start : Win;
        endcase

endmodule // light_game
```
Exercise 1 (Solution)

- Output logic

```verilog
module light_game (input logic clk, reset, M, L, output logic W);

  enum logic [1:0] {Start, Mid, Win} ps, ns;

  always_comb
    case (ps)
      Start: ns = (L & M) ? Mid : Start;
      Mid:   ns = (L & M) ? Win : (M ? Start : Mid);
      Win:   ns = M ? Start : Win;
    endcase

  assign W = (ns == Win);
  // alt: ((ps == Mid) & L & M) | ((ps == Win) & ~M)
endmodule // light_game
```
Exercise 1 (Solution)

- State update logic

```verilog
module light_game (input logic clk, reset, M, L, output logic W);

  enum logic [1:0] {Start, Mid, Win} ps, ns;

  ...
  // next state logic
  ...
  // output logic

  always_ff @(posedge clk)
  begin
    if (reset)
      ps <= Start;
    else
      ps <= ns;
  end

endmodule  // light_game
```
Exercise 2

- Below is an FSM for a modified vending machine with increased cost of 15¢ for gumballs that also accepting quarters (Q: 25¢); it still does not give change and can only take one coin at a time.

○ Implement this system in a module called **vend15**.
Exercise 2 (Solution)

- Module outline

```verilog
module vend15 (input logic clk, reset, N, D, Q, output logic Open);

endmodule // vend15
```
Exercise 2 (Solution)

- State encodings and variables

```verilog
module vend15 (input logic clk, reset, N, D, Q, output logic Open);
  enum logic [1:0] {Zero, Five=2'b10, Ten=2'b11} ps, ns;
endmodule  // vend15
```
Exercise 2 (Solution)

- Next state logic

```verilog
module vend15 (input logic clk, reset, N, D, Q, output logic Open);
    enum logic [1:0] {Zero, Five=2'b10, Ten=2'b11} ps, ns;

    always_comb
        case (ps)
            Zero: case ({N, D, Q})
                3'b000: ns = Zero;
                3'b100: ns = Five;
                3'b010: ns = Ten;
                3'b001: ns = Zero;
                default: ns = ps;
            endcase
        endcase
    ...
        // Five and Ten defined similarly
    endcase
endmodule // vend15
```
Exercise 2 (Solution)

- Output logic

```verilog
module vend15 (input logic clk, reset, N, D, Q, output logic Open);
    enum logic [1:0] {Zero, Five=2'b10, Ten=2'b11} ps, ns;

    // next state logic
    assign Open = Q | ((ps != Zero) & D) | ((ps == Ten) & N);

endmodule // vend15
```
Exercise 2 (Solution)

- State update logic

```verilog
module vend15 (input logic clk, reset, N, D, Q, output logic Open);
  enum logic [1:0] {Zero, Five=2'b10, Ten=2'b11} ps, ns;
...
  // next state logic

  assign Open = Q | ((ps != Zero) & D) | ((ps == Ten) & N);

  always_ff @(posedge clk)
    if (reset)
      ps <= Zero;
    else
      ps <= ns;

endmodule  // vend15
```
Finite State Machine Testing
FSM Test Bench Notes

- All notes about sequential test benches from last week still apply!
  - Generate a simulated clock (don’t use `clock_divider`), start with a reset and define all inputs at \( t=0 \), add extra delay at end to see the effects of your last input changes.

- To thoroughly test your FSM, need to take every transition that we care about (can omit/ignore don’t cares).

- Recommended test bench lines in `initial` block:
  ```
  <input changes> @\( \text{posedge} \text{ clk} \); // current state: ???
  ```

- In ModelSim, you should at least add `ps` to waveforms.
  - Could also include `ns` or other signals involved in `ps/ns` computations.
"FSM Test Bench Example"

```verilog
// generate test vectors
initial begin
    reset <= 1; w <= 0; @(posedge clk); // reset
    reset <= 0; @(posedge clk); // curr state: S0

    $stop; // pause the simulation
end
```
FSM Test Bench Example

```
// generate test vectors
initial begin
    reset <= 1; w <= 0; @(posedge clk); // reset
    reset <= 0;
    @(posedge clk); // curr state: S0
    w <= 1; @(posedge clk); // curr state: S0
    w <= 0; @(posedge clk); // curr state: S1
    w <= 1; @(posedge clk); // curr state: S0
    @(posedge clk); // curr state: S1
    @(posedge clk); // curr state: S11
    @(posedge clk); // curr state: S11
    w <= 0; @(posedge clk); // curr state: S11
    @(posedge clk); // curr state: S0 (extra cycle)

    $stop; // pause the simulation
end
```
Exercise 3

- Create a test bench for vend15 and simulate it in ModelSim.
  - What’s the minimum number of clock cycles required to thoroughly test it?
Exercise 3 (Solution)

- Create module, declare port connections, instantiate dut.

```verbatim
module vend15_tb ();
  logic clk, reset, N, D, Q, Open;

  vend15 dut (.*);

endmodule   // vend15_tb
```
Exercise 3 (Solution)

- Setup clock.

```verilog
module vend15_tb ();
    ...
    // signal declarations and dut instantiation

    parameter T = 100;
    initial
        clk = 1'b0;
    always begin
        #(T/2) clk <= 1'b0;
        #(T/2) clk <= 1'b1;
    end

endmodule  // vend15_tb
```
Exercise 3 (Solution)

- Define `initial` block and add `$stop` system task.

```verilog
module vend15_tb ();
  ...
  // signal declarations and dut instantiation
  ...
  // clock generation

  initial begin
  $stop;
  end

endmodule  // vend15_tb
```
Exercise 3 (Solution)

- Start with a reset and initialize all inputs.

```verilog
module vend15_tb ();
    ...
    // signal declarations and dut instantiation
    ...
    // clock generation

    initial begin
        {reset,N,D,Q} <= 4'b1000; @(posedge clk); // reset

        $stop;
        end

endmodule // vend15_tb
```
Exercise 3 (Solution)

- Map out a sequence of inputs that would allow us to test every transition.
Exercise 3 (Solution)

- Map out a sequence of inputs that would allow us to test every transition.
  - This is just one of many possibilities!
Exercise 3 (Solution)

- Add the transitions we mapped out.

```verilog
module vend15_tb ();
    // signal declarations, dut instantiation, clock generation
initial begin
    {reset,N,D,Q} <= 4'b1000; @(posedge clk); // reset
    {reset,N,D,Q} <= 4'b0000; @(posedge clk); // Zero (1)
    {N,D,Q} <= 3'b001; @(posedge clk); // Zero (2)
    {N,D,Q} <= 3'b100; @(posedge clk); // Zero (3)
    {N,D,Q} <= 3'b001; @(posedge clk); // Five (4)
    {N,D,Q} <= 3'b100; @(posedge clk); // Zero (5)
    {N,D,Q} <= 3'b000; @(posedge clk); // Five (6)
    {N,D,Q} <= 3'b010; @(posedge clk); // Five (7)
    {N,D,Q} <= 3'b100; @(posedge clk); // Zero (8)
    @(posedge clk); // Five (9)
    ... // continued on next slide
```
Exercise 3 (Solution)

- Add the transitions we mapped out.

```verilog
... // signal declarations, dut instantiation, clock generation
initial begin
  ... // previous clock cycles
  {N,D,Q} <= 3'b001; @(posedge clk);  // Ten (10)
  {N,D,Q} <= 3'b010; @(posedge clk);  // Zero (11)
  @(posedge clk);  // Ten (12)
  @(posedge clk);  // Zero (13)
  {N,D,Q} <= 3'b100; @(posedge clk);  // Ten (14)
  {N,D,Q} <= 3'b010; @(posedge clk);  // Zero (15)
  {N,D,Q} <= 3'b000; @(posedge clk);  // Ten (16)
  @(posedge clk);  // extra
  $stop;
end
dendmodule  // vend15_tb
```
Exercise 3 (Solution)

- Simulation results should verify that (1) reset works, (2) the transition between states as expected, and (3) our output matches what we expect.
Exercise 3 (Solution)

- Step 1 - Verify the reset behavior.

```verilog
module vend15 (...)

    ...

    always_ff @(posedge clk)
    if (reset)
        ps <= Zero;
    else
        ps <= ns;

endmodule // vend15
```

```verilog
module vend15_tb ();

    ...

    // signal declarations, dut instantiation, clock generation
    initial begin
        {reset,N,D,Q} <= 4'b1000; @(posedge clk); // reset
        {reset,N,D,Q} <= 4'b0000; @(posedge clk); // Zero (1)
        ...
    endmodule
```
Exercise 3 (Solution)

- Step 2 - Verifying every transition between states as expected.

```verilog
initial begin
    {reset,N,D,Q} <= 4'b1000; @(posedge clk); // reset
    {reset,N,D,Q} <= 4'b0000; @(posedge clk); // Zero (1)
    {N,D,Q} <= 3'b001; @(posedge clk); // Zero (2)
    {N,D,Q} <= 3'b100; @(posedge clk); // Zero (3)
    {N,D,Q} <= 3'b001; @(posedge clk); // Five (4)
    {N,D,Q} <= 3'b100; @(posedge clk); // Five (5)
    {N,D,Q} <= 3'b000; @(posedge clk); // Five (6)
    {N,D,Q} <= 3'b010; @(posedge clk); // Five (7)
    {N,D,Q} <= 3'b100; @(posedge clk); // Five (8)
    {N,D,Q} <= 3'b001; @(posedge clk); // Five (9)
    {N,D,Q} <= 3'b010; @(posedge clk); // Ten (10)
    {N,D,Q} <= 3'b001; @(posedge clk); // Ten (11)
    {N,D,Q} <= 3'b010; @(posedge clk); // Ten (12)
    {N,D,Q} <= 3'b100; @(posedge clk); // Ten (13)
    {N,D,Q} <= 3'b010; @(posedge clk); // Ten (14)
    {N,D,Q} <= 3'b000; @(posedge clk); // Ten (15)
    {N,D,Q} <= 3'b000; @(posedge clk); // extra

...
Exercise 3 (Solution)

- Step 3 - Verifying our output matches what we expect.

```
assign Open = Q | ((ps != Zero) & D) | ((ps == Ten) & N);
```