Section 4
Sequential Logic
**Administrivia**

- **Lab 4:** Report due next Wednesday (4/24) @ 2:30 pm, demo by last OH on Friday (4/26), but expected during your assigned slot.
- **Lab 5:** Report due 5/1, demo by last OH on 5/3.
  -⚠️ This lab is a LOT harder than previous labs⚠️
- **Quiz 1:** Tuesday (4/23) at end of Lecture.
  - Very formulaic: gates, DeMorgan’s, K-map, waveforms, test benches.
  - Study from past quizzes on course website!
Parameters
New SystemVerilog Commands

- `parameter` – create a symbolic constant for a value that can be referenced in scope.
  - Like `#define` in C/C++.
  - Useful for things like timing constants, state names, module widths.
New SystemVerilog Commands

- **parameter** – create a symbolic constant for a value that can be referenced in scope.
  - Like `#define` in C/C++.
  - Useful for things like timing constants, state names, module widths.

- **Parameterized modules:**
  - Definition: `module <name> #(<param list>) (<port list>);`
    - `<param list>` is comma-separated and can include default values (e.g., `#(M, N=4)`).
  - Instantiation: `<name> #(<params>) <inst_name> (<ports>);`
    - Notice that parameter definitions are to the *left* of the instance name!
    - Generates different *versions* of the same module definition (like templates in C++).
Exercise 1

● (1) Parameterize the comparator module for bit-width \( N \):
  ○ Hint: you will need to use a reduction operator (e.g., \(~\&A\)) which will reduce all the bits of a vector into a single value using the specified Boolean operator.

```verbatim
// Implements an N-bit comparator circuit
module comparator (A, B, is_lt, is_eq, is_gt);
```

● (2) Parameterize the guessing_game module for bit-width \( N \) and secret number \( S \):

```verbatim
// Game to check user's N-bit input guess against a secret #
module guessing_game (LEDR, KEY, SW);
```
Exercise 1 (Solution)

- Changes **underlined** and shown in **red**:

```verilog
module comparator #(N = 3)
    (input logic [N-1:0] A, B,
    output logic is_lt, is_gt, is_eq);

    // subtraction result (intermediate)
    logic [N-1:0] sub;
    assign sub = A - B;

    assign is_eq = ~|sub;
    assign is_lt = sub[N-1];
    assign is_gt = ~is_eq & ~is_lt;

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

- Changes **underlined** and shown in **red**:

```verilog
text
module guessing_game #(N=3, S=3'd1)
   (output logic [9:0] LEDR,
    input logic [3:0] KEY, input logic [9:0] SW);

   logic is_lt, is_eq, is_gt;

   comparator #(.N(N)) number_comparator (   
     .A(SW[N-1:0]), .B(S), .is_lt, .is_eq, .is_gt 
   );

   ... // LEDR assignments (unchanged)
endmodule  // guessing_game
text
```
Sequential Logic
New SystemVerilog Commands

- **always_ff** – higher-level description of behavior that includes sequential logic.
  - Requires an explicit sensitivity/trigger list (e.g., @(posedge clk)) that dictates when the code block will take effect.

- Non-blocking statements (<=) should be used with `always_ff`, blocking statements (=) should be used with `assign` and `always_comb`. 
Flip-Flops and Registers (Review)

- A **flip-flop** samples d on triggers and transfers its value to q.

```verilog
define basic_D_FF (output logic q, input logic d, clk);
  always_ff @(posedge clk)
    q <= d;
endmodule  // basic_D_FF
```

- A **register** is a collection of N flip-flops together.

```verilog
define basic_reg #(N) (output logic [N-1:0] Q,
  input logic [N-1:0] D,
  input logic clk);
  always_ff @(posedge clk)
    Q <= D;
endmodule  // basic_reg
Reset Functionality (Review)

- A sequential element often has a reset signal that will drive its output to a known value.
  - Useful in hardware to substitute for “initialization.”
  - Two options, synchronous (left) or asynchronous (right):

```verilog
module D_FF1 (output logic q, input logic d, reset, clk);
    always_ff @(posedge clk)
        if (reset)
            q <= 0;
        else
            q <= d;
endmodule // D_FF1

module D_FF2 (output logic q, input logic d, reset, clk);
    always_ff @(posedge clk or posedge reset)
        if (reset)
            q <= 0;
        else
            q <= d;
endmodule // D_FF2
```
Clock in Hardware

- We will use the DE1-SoC’s built-in 50 MHz clock called `CLOCK_50`.
  - Accessed by adding `CLOCK_50` as an input logic to your top-level module.
- Because 50 MHz (i.e., clock period = 20 ns) may be too fast for humans, can use provided `clock_divider` module to slow things down.
  - Recommendation: assign extra signal `clk` to `divided_clocks[#]`.
  - Make sure to comment out `clock_divider` for simulation!

```
logic [31:0] divided_clocks;
logic clk;
clock_divider cdiv (.clock(CLOCK_50), .divided_clocks);
assign clk = divided_clocks[23];  // replace with = CLOCK_50 for simulation
// Instantiating a module that is using clock 23
<module_name> <instance_name> (.clk, .reset, ...);
```
Exercise 2

- Write a module called `string_lights` that implements the system shown below (a string of 10 flip-flops/1-bit registers tied to the LEDRs) for the DE1-SoC.
  - Use `SW[9]` as the reset, `SW[0]` as In, and `~KEY[0]` as clk.
    - Since we are using a KEY for the clock, no need for `clock Divider`.
  - **Hint:** flip-flops can be module instances or inferred from an `always_ff` block.
**Exercise 2 (Solution)**

- **Version 1:** module instances
  - Connections made via ports.

```
module string_lights (output logic [9:0] LEDR,
                      input logic [3:0] KEY,
                      input logic [9:0] SW);

logic clk, reset, in;
assign clk   = ~KEY[0];
assign reset = SW[9];
assign in    = SW[0];

D_FF1 ff9 (.q(LED[9]), .d(in), .reset, .clk);
D_FF1 ff8 (.q(LED[8]), .d(LED[9]), .reset, .clk);
...
D_FF1 ff1 (.q(LED[1]), .d(LED[2]), .reset, .clk);
D_FF1 ff0 (.q(LED[0]), .d(LED[1]), .reset, .clk);

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

- **Version 2:** `always_ff`
  - Connections made via non-blocking assignments.

```verilog
module string_lights (output logic [9:0] LEDR,
                      input  logic [3:0] KEY,
                      input  logic [9:0] SW);

  logic clk, reset, in;
  assign clk   = ~KEY[0];
  assign reset = SW[9];
  assign in    = SW[0];

  always_ff @(posedge clk)
  begin
    if (reset)
      LEDR <= 10'd0;
    else
      LEDR <= {in, LEDR[9:1]};
  end

endmodule  // string_lights
```
Exercise 2 Demo (If Time)

- Compile and run `string_lights` on a DE1-SoC.
  - Normally, you should ALWAYS run simulations first.
Sequential Logic Test Benches
Clock Generation (Review)

- In simulation, need to create a clock signal yourself (steady square wave).
  - Just pick your favorite form and copy-and-paste into your future test benches.
  - Exact period doesn’t really matter since it’s all arbitrary time units.

```vhdl
parameter T = 100;  // period
initial
    clk = 0;
always begin
    #(T/2) clk <= 1;
    #(T/2) clk <= 0;
end
```

Explicit Edges:

Toggle:

```vhdl
parameter T = 100;  // period
initial
    clk = 0;
always
    #(T/2) clk <= ~clk;
```
Edge-Sensitive Delays

- Delays until specified transition on signal: `@(<pos/negedge> signal);`
  - Allows us to wait for the next clock trigger in our simulation since that's when sequential elements will update.

- Example test bench block:

```verilog
initial begin
  d <= 1'b1; reset <= 1'b1; @(posedge clk); // reset
  reset <= 1'b0; @(posedge clk); // store 1
  @(posedge clk); // hold 1
  d <= 1'b0;
  @(posedge clk); // store 0
  @(posedge clk); // hold 0
  $stop();
end
```
Sequential Test Bench Notes

- Need to manually track the expected state for sequential elements.
- Always define ALL of your inputs at $t=0$, even if you’re resetting, to eliminate unnecessary red lines in simulation.
- Whitespace in initial block doesn’t matter but we recommend being consistent (i.e., line up your delays on right or left side of each line).
- All logic delays set to 0 in our ModelSim setup, so be careful with interpreting signal changes.
- Include an extra delay at the end to see the effects of your last input changes.
Exercise 3

- Create a test bench for `string_lights` and simulate it in ModelSim.
  - Do we need this test bench to be thorough? What would be enough to convince you that it is working properly?
  - What do you think the best combination of signals (and radices) are to use for the reader of your simulation?
    - *e.g.*, do you want to show the top-level `SW[9]` signal or an internal `reset` signal?
Exercise 3 (Solution)

● Create Module, create ports, instantiate dut

```verilog
module string_lights_tb ();
    logic [9:0] LEDR;
    logic [3:0] KEY;
    logic [9:0] SW;

    string_lights dut (.*);

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

- Setup clock – since KEY[0] is active-low, need to start with 1 instead of 0.

```verilog
testbench
module string_lights_tb ();
...

parameter T = 100;
initial
  KEY[0] = 1'b1;
always begin
  #(T/2) KEY[0] <= 1'b0;
  #(T/2) KEY[0] <= 1'b1;
end

endmodule
```
Exercise 3 (Solution)

- Define `initial` block and add `$stop` system task.
  - Make sure to initialize all inputs at $t = 0$!

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

    initial begin
        SW[0] <= 1'b0; SW[9] <= 1'b1; @(negedge KEY[0]); // reset

        $stop;
        end

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

- We can now start simulating some possible behaviors of our design!
  - e.g., let's try the input sequence 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1

```verilog
module string_lights_tb ();

  // signal declarations and dut instantiation
  ...

  // clock generation
  initial begin
    SW[0] <= 1'b0; SW[9] <= 1'b1; @(negedge KEY[0]); // reset
    SW[0] <= 1'b0; SW[9] <= 1'b0; @(negedge KEY[0]); // 0
    SW[0] <= 1'b1; @(negedge KEY[0]); // 1
    ...
    // finish desired pattern
    @(negedge KEY[0]); // final delay
    $stop;
  end
endmodule // string_lights_tb
```
Exercise 3 (Solution)

- Simulation results verify (1) reset works, (2) inputs travel across entire string, and (3) a variety of combinations of inputs.
  - Using internal signal names for readability.
Exercise 3 (Solution)

- Simulation results verify (1) reset works, (2) inputs travel across entire string, and (3) a variety of combinations of inputs.
  - Using internal signal names for readability.

- Many other behaviors are possible and should be tested!
  - The idea here is not necessarily to test out all possibilities like in combinational logic but enough relevant scenarios to give you confidence that it is working properly.