Section 3

Test Benches
Administtrivia

- **Lab 3:** Report due next Wednesday (4/17) @ 2:30 pm, demo by last OH on Friday (4/19), but expected during your assigned slot.
- **Lab 4:** Report due 4/24, demo by last OH on 4/26
New SystemVerilog Commands

- **always_comb** – higher-level description of more complex combinational behavior.
  - Used to combine multiple assignment statements or express more situational assignments.

- **case/endcase** – describe desired behavior situationally (based on value of expression)
  - Like a switch statement in other languages, but no fall-through and no `break`.
  - Use `default` to cover remaining cases.

- Use `begin` and `end` to group multiple statements together.
  - Like `{ and }` in other languages.
  - *e.g.*, to put multiple statements in one `always_comb` or for one case
Test Benches
Writing a Test Bench

1) Start with the module skeleton (`module / endmodule`).
   a) Please use the naming convention of `<module_name>_tb`.

2) Create signals for all ports of the module you’re going to test.
   a) Suggested to copy-and-paste from module definition, but remove port types (i.e., `input`, `output`).

3) Instantiate device under test (`dut` as instance name).
   a) Port connections: `.<port>(<signal>)` but names match can do `.<port>` as shorthand, or `.*` if all signal names match port names.

4) Define test vectors in an `initial` block.
   a) Needs to end with `$stop`; system task for ModelSim to pause.
Test Vectors for Combinational Logic

- Output of combinational logic is determined by current value of inputs.
  - Need to run through all possible input combinations in simulation to thoroughly test.
Test Vectors for Combinational Logic

- Output of combinational logic is determined by current value of inputs.
  - Need to run through all possible input combinations in simulation to thoroughly test.

- In order to have output values be visible in simulation, need to add arbitrary time delays `<num>`; (e.g., `#10`) in-between input changes.
  - Note that our ModelSim setup has all combinational logic delays set to 0.
Test Vectors for Combinational Logic

- Output of combinational logic is determined by current value of inputs.
  - Need to run through all possible input combinations in simulation to thoroughly test.

- In order to have output values be visible in simulation, need to add arbitrary time delays `#<num>; (e.g., #10;) in-between input changes.
  - Note that our ModelSim setup has all combinational logic delays set to 0.

- Can use “for-loop” to run through all input combinations:
  ```
  for (int i; i < 8; i++) begin
    // set inputs based on i, then time delay
  end
  ```
  - No sequential execution just condenses your code.
Exercise 1

- Write a test bench for the provided `seg7` module from Lecture 3.
  - Be thorough, including all 16 input combinations!

```verilog
module seg7 (bcd, leds);
  input logic [3:0] bcd;
  output logic [6:0] leds;

  always_comb
    case (bcd)
      // Light: 6543210
      4'b0000: leds = 7'b0111111; // 0
      ... // implementation
      4'b1001: leds = 7'b1101111; // 9
      default: leds = 7'bX;
    endcase
endmodule // seg7
```
Exercise 1 (Solution)

- First we declare our simulated port connections.
  - Can copy-and-paste port declarations and remove input/output.

```verilog
module seg7_tb ();
  logic [3:0] bcd;
  logic [6:0] leds;
endmodule  // seg7_tb
```
Exercise 1 (Solution)

- Instantiate device under test.

```verilog
module seg7_tb ();
  logic [3:0] bcd;
  logic [6:0] leds;

  seg7 dut (.bcd(bcd), .leds(leds));
endmodule // seg7_tb
```
Exercise 1 (Solution)

- Instantiate device under test.
  - Alternatively, can use .* since our signals match the port names.

```verilog
module seg7_tb ();
    logic [3:0] bcd;
    logic [6:0] leds;

    seg7 dut (.*);

dendmodule  // seg7_tb
```
Exercise 1 (Solution)

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

```verilog
module seg7_tb ();
    logic [3:0] bcd;
    logic [6:0] leds;

    seg7 dut (.*);

    int i;
    initial begin

        $stop;
    end
endmodule  // seg7_tb
```
Exercise 1 (Solution)

- Test all possible combinations of inputs.

```verilog
module seg7_tb();
    ...
    // signal declarations & dut instantiation

    int i;
    initial begin
        for (i = 0; i < 16; i++) begin
            bcd = i; #20;
        end
    $stop;
    end

endmodule  // seg7_tb
```
Exercise 2

- Write a test bench for the `guessing_game` module from Section 2.
  - Be thorough: how many input combinations are there?

```verilog
// Game to check user's 3-bit input guess against a hard-coded secret #
// - SW[2:0] is the guess, KEY[0] is check
// - LEDR[0] is <, LEDR[1] is ==, LEDR[2] is >
module guessing_game (  
    output logic [9:0] LEDR,  
    input logic [3:0] KEY,  
    input logic [9:0] SW  
);

... // implementation

endmodule // guessing_game
Exercise 2 (Solution)

- First we declare our simulated port connections.
  - Can copy-and-paste port declarations and remove input/output.

```verbatim
module guessing_game_tb ();
  logic [9:0] LEDR;
  logic [3:0] KEY;
  logic [9:0] SW;
endmodule  // guessing_game_tb
```
Exercise 2 (Solution)

- Instantiate device under test.

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

  guessing_game dut (  
    .LEDR(LEDR),  
    .KEY(KEY),  
    .SW(SW)  
  );

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

- Instantiate device under test.
  - Alternatively, can use .\* since our signals match the port names

```verilog
dmodule guessing_game_tb ();
  logic [9:0] LEDR;
  logic [3:0] KEY;
  logic [9:0] SW;

  guessing_game dut (.\*);
endmodule // guessing_game_tb
```
Exercise 2 (Solution)

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

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

    guessing_game dut (.);

    initial begin
        $stop;
    end
endmodule  // guessing_game_tb
```
Exercise 2 (Solution)

- Test all possible combinations of inputs.

```verilog
module guessing_game_tb ();
    ...
    // signal declarations & dut instantiation

    initial begin
        KEY[0] = 1'b1; #10;                          // KEYs are active low
        for (int i = 0; i < 8; i++) begin
            SW[2:0] = i; KEY[0] = 1'b0; #10;          // LEDs should light up
            KEY[0] = 1'b1; #10;                       // LEDs should be disabled
        end
        $stop;
    end
endmodule // guessing_game_tb
```
ModelSim Tips & Tricks
Simulation Workflow (Review)

- Double-click `Launch_ModelSim.bat` in the project directory.

- In a text editor, modify `runlab.do` for your project:
  - Add files to compile (modules + test benches).
  - Change which test bench you wish to simulate.
  - Change the waveform script file (`*_wave.do`) – this won’t exist at first.

- Execute `do runlab.do` in the Transcript pane.
  - Use waveforms to verify/debug logical behavior of your module(s).

- Update waveform script file as desired.
  - Click on different modules in the `sim` pane to access different signals.
  - Drag signals from the `Objects` pane into the `Wave` pane.
  - With the `Wave` pane selected, `Ctrl+S` to overwrite your waveform script file.
**Zoom Tools**

- Zoom tools allow you to adjust the amount of the simulation you can view at once as well as the visibility of the signal values.
  - Critical for generating understandable screenshots for your lab reports!
Zoom Tools

● Zoom in/Zoom out
  ○ Allows you to change the amount of information shown at once (e.g., 200 ps at a time, 1000 ps at a time).

● Zoom Full
  ○ Automatically zooms to show the whole simulation at once.
  ○ Good for short simulations, not great for longer simulations.

● Zoom In on Active Cursor
  ○ Zooms in based on the location of the yellow cursor.
Signal Radix

- Right-click a signal in the Wave pane and use the “Radix” menu to change the display of a signal’s value
  - This does NOT change the actual bits, just how we interpret them!!!
  - Common choices: Binary (default), Unsigned, Decimal (i.e., signed integer), Hexadecimal
Exercise 3

- Run your `guessing_game` simulation in ModelSim and use it to identify a few input combinations that produce the wrong outputs for *signed* integer interpretation.
  - Tools we just covered: zoom tools and signal radix.
Exercise 3 (Solution)

- By dragging the relevant signals from the Objects window to our Wave window, we get the following waveform:
Let’s make the waveform more interpretable!

- Since we are interpreting the switch signals as *signed* numbers, we should change the radix to be *Decimal*.
Internal Signals

- ModelSim lets you add internal signals from any instantiated module to your simulation!
  - Incredibly useful to trace buggy or unexpected values to their source.
  - Click [+] next to an instance name to reveal submodules (by instance name).
  - Click the instance name to access different internal signals in the Objects pane:
Exercise 3 (Debugging)

- Our secret num is 1 but our system reports that -4 is greater...?
  - LEDR[2:0] = { is_gt&~KEY[0], is_eq&~KEY[0], is_lt&~KEY[0] };

- Let’s investigate further:
  - We know that is_gt is an output of the comparator module (instantiated within dut as number_comparator).
  - Can add sub signal from within the comparator module to the waveform!
Exercise 3 (Debugging)

- Let’s investigate further:
  - We know that `is_greater_than` is an output of the comparator module (instantiated within `dut`).
  - Can add `sub` signal from within the comparator module to the waveform!

- $-4-1$ is returning `3'b011` (–5 can’t be represented in 3 bits), so the outputs produce unexpected values!
Lab Reports
Simulations for Lab Reports

- You are using simulations to communicate something to the reader.
  - Usually, “proving” correct behavior of your circuit/system.
  - Difficult to interpret on their own, so accompanying explanation is critical.
  - Useful both to the grader and to you looking back on this work in the future.
Simulations for Lab Reports

- You are using simulations to communicate something to the reader.
  - Usually, “proving” correct behavior of your circuit/system.
  - Difficult to interpret on their own, so accompanying explanation is critical.
  - Useful both to the grader and to you looking back on this work in the future.

- Goals and Tips:
  - All of simulation is included – can be split across multiple images, if needed.
    - Helpful to design test bench to be as concise as possible.
  - Labeling – time (horizontal) axis and all signal names are clearly visible.
    - Can undock the Wave pane to change window size or can drag vertical divider of Transcript pane up to get time axis label closer to signals.
    - Toggling to leaf names shortens signal names.
  - All signal values are visible throughout the simulation.
    - Changing radix can help condense but should make sense in context.
Simulations for Lab Reports (BAD Example)

- What are we looking at here???
Simulations for Lab Reports (GOOD Example)

- Split across two images to make values of `leds` legible.
- Changed `bcd` radix to hexadecimal: easier to read and matches use case.
  - Decimal would work here, too.
- Can refer to specific times in simulation in explanation now.