UW Student
EE/CSE 371
January 4, 2021
Sample Lab Report

The lab report should contain the sections listed in the lab handout.

Design Procedure

In this lab we were tasked with building a two-player game of tug-of-war game. Player 1 (right) uses `KEY[0]` and the Player 2 (left) uses `KEY[3]` to pull a light using the `LEDR` display, much like the flag tied to a rope. When the light goes past the farthest left or right side of the `LEDR` display, the winning player is shown on the `HEX` display. `SW[9]` is used as the reset signal.

Task #1

A good way to divide each section is into subsections by task.

Key presses are asynchronous inputs that usually last longer than one clock cycle. To handle these logical issues, we created a module that passes these inputs through a two-DFF synchronizer (not shown) to the finite state machine (FSM) in Figure 1 that pulses high for a single clock cycle when they key is initially pressed. Note that DE1-SoC `KEY` signals are **active-low** so a value of 0 means pressed.

Included images should be titled with figure labels.

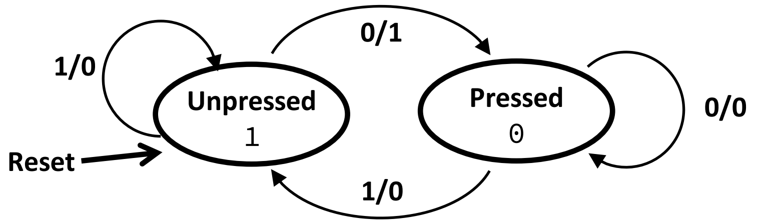
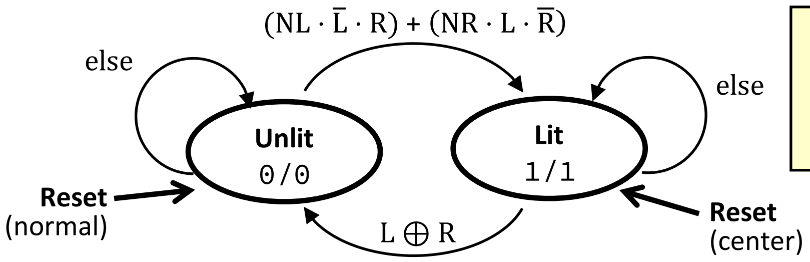


Figure 1: The Mealy FSM for a portion of the `singlePress` module. The active-low `KEY` input comes from a synchronizer and the output represents a single pulse for each time the key has been pressed (and held).

Task #2

To control the state of the LEDs, we created two very similar modules (both represented in Figure 2): one for the center/starting LED and one for the normal/other LEDs that only differ in reset behavior.


$(NL \cdot \overline{L} \cdot R) + (NR \cdot L \cdot \overline{R})$
else
else
Unlit
0/0
Lit
1/1
Reset
(normal)
$L \oplus R$
Reset
(center)

If signal names are not immediately obvious, they should be explained.

Figure 2: The Moore FSM for the `normalLight` and `centerLight` modules, which only differ in reset state. L and R are the left and right key presses. NL and NR represent if the left and right neighboring lights are on.

The FSM in Figure 2 relies on the following behavior:

- If unlit, the LED will only turn on if one of its neighbors is on and there is a tug in its direction.
- If lit, the LED will turn off if there is a tug in either direction.
- A tug is only registered if one player tugs and the other does not (*i.e.*, $L \oplus R$).
- Only one LED can be on at a time so we ignore the `NL` and `NR` inputs (*i.e.*, assume them to be 0) in the Lit state.

Task #3

The final task was to display which player had won the game on the 7-segment/HEX displays. We created a module to determine a winner when one of the outermost LEDs (FL = far left, FR = far right) is lit and there is tug off of the play field. To ensure that the game ends/freezes once there is a winner, we added state as shown in Figure 3.

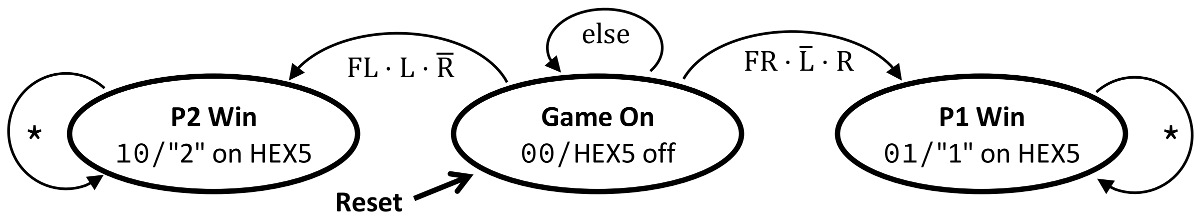
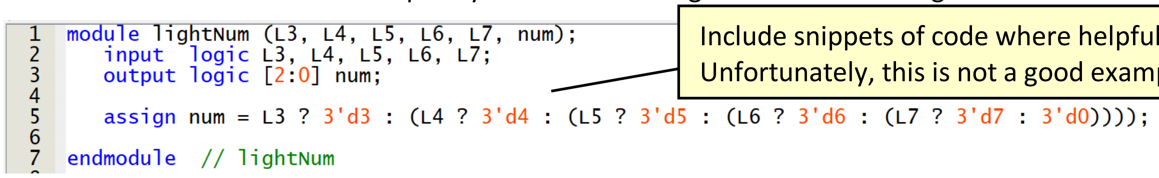


Figure 3: The Moore FSM for the `Winner` module. HEX5 is used for the output. The "" transition means regardless of inputs.*

Tester Module

Although not explicitly asked for in the lab specification, we decided to create an additional module for testing. This `lightCounter` module takes the current value of every LED in the playfield and outputs the number of the lit LED, which will allow us to replace the individual `LEDR` signals and simplify the ModelSim wave view. This module is purely combinational logic with the following code:



```
module lightNum (L3, L4, L5, L6, L7, num);
   input  logic L3, L4, L5, L6, L7;
   output logic [2:0] num;

   assign num = L3 ? 3'd3 : (L4 ? 3'd4 : (L5 ? 3'd5 : (L6 ? 3'd6 : (L7 ? 3'd7 : 3'd0))));

endmodule  // lightNum
```

Include snippets of code where helpful. Unfortunately, this is not a good example.

Figure 4: Code for the `lightNum` module used for testing.

Overall System

Make sure that signals are labeled and port directions are indicated.

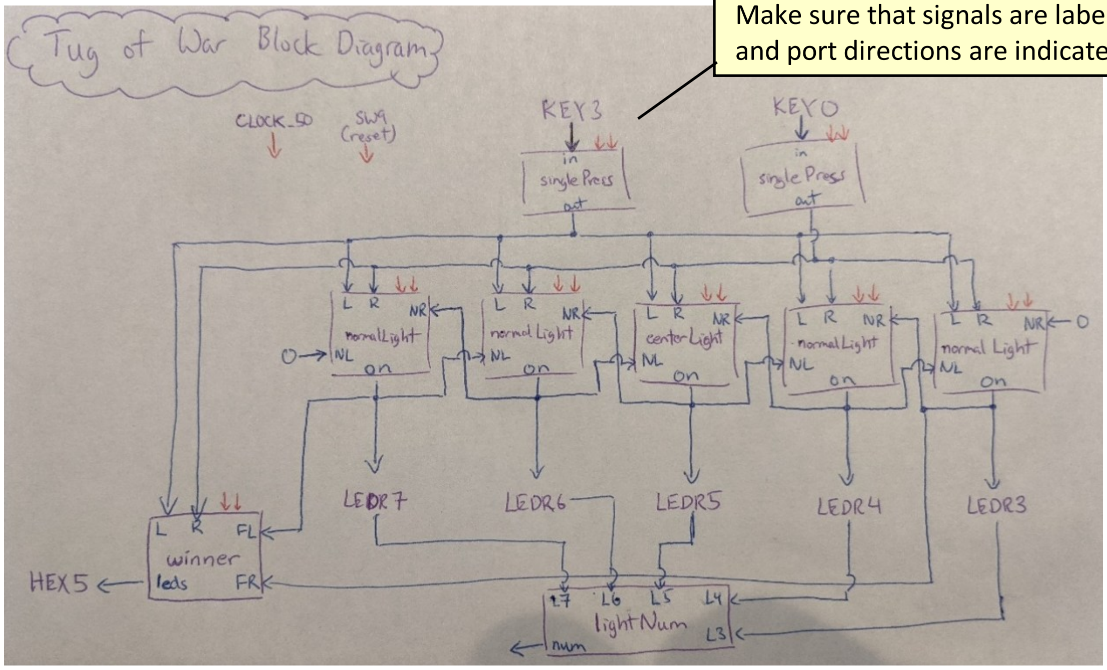


Figure 5: Top-level block diagram of the Tug of War system.

The block diagram of the system as a whole can be seen in Figure 5, which comprises nine total modules of five different types. The output of `lightNum` is left unconnected as it is only used for testing. None of the modules instantiate internal submodules. The disconnected red arrows are assumed to be connected to the clock (`CLOCK_50`) and reset (`SW9`) inputs and apply to all sequential modules.

Results

The overview briefly describes what the final system is designed to do (often paraphrased from the spec) and the unspecified design decisions that you made.

Our system is a 2-player tug of war game with a lit LED representing the 'flag' that starts in the middle of the 'rope.' Players pull the flag towards their side by pressing and releasing pushbuttons. When the flag is pulled off of the side of the playfield, the corresponding player wins and their player number (1 or 2) is shown on a hex display. We decided to make the system freeze (*i.e.*, pulling inputs are ignored) after a win until a reset signal is seen.

singlePress

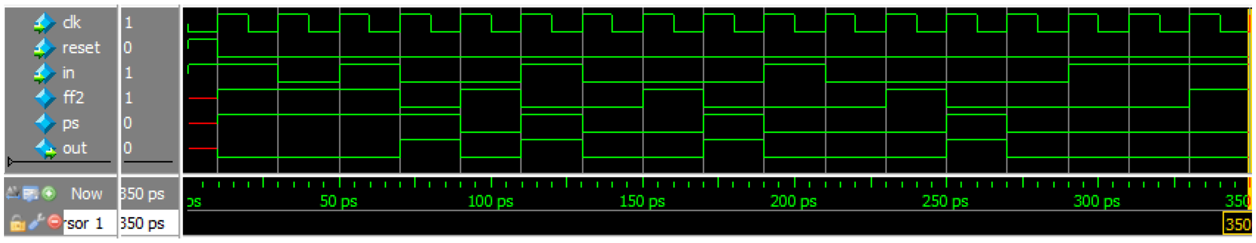


Figure 6: The ModelSim waveform of the `singlePress` module.

Here, `in` is the key value (active-low) and ff2 is the output of the second DFF of our synchronizer, which is the input to our FSM (Figure 1). In this testbench, we show `in` presses of duration 1 (starting at 30 ps), 2 (70 ps), 3 (130 ps), and 4 (210 ps) clock cycles. These each produce a one clock cycle pulse in `out` that is delayed by two clock cycles (starting at 70, 110, 1 70, and 250 ps).

winner

Notice how all signal names are clearly visible and large enough to be read, and that each signal's value at the cursor is displayed.

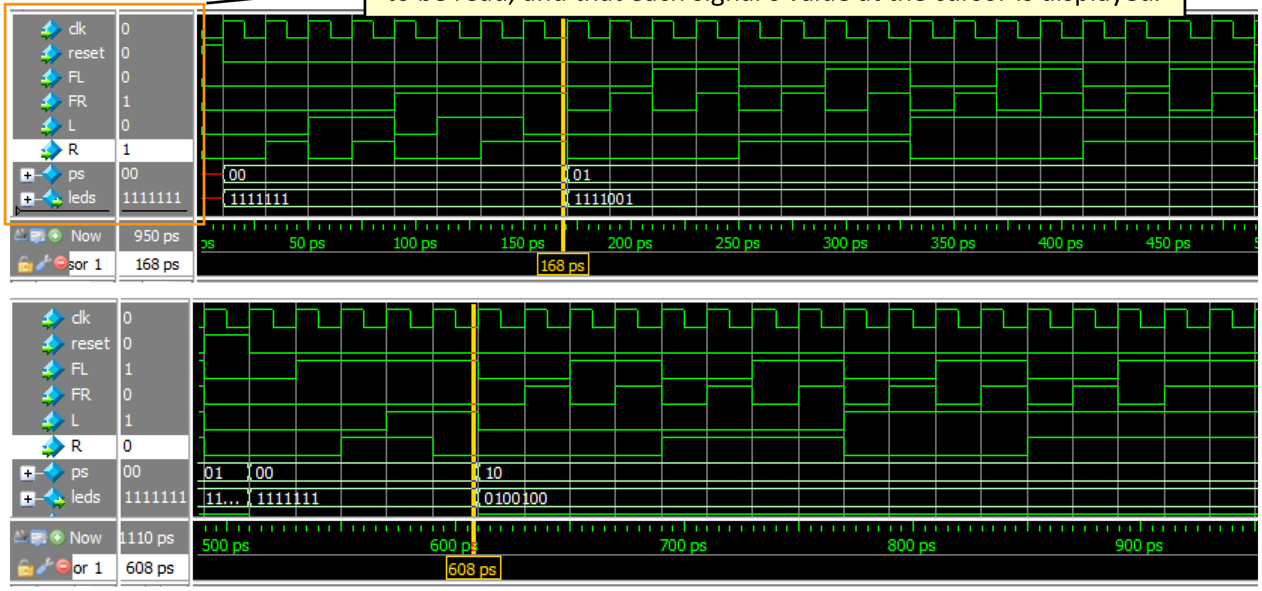


Figure 7: The ModelSim waveforms (split into two images for clarity) of the `winner` module.

In the upper image, we start in the "Game On" state (`ps == 0`) and try all input combinations with $\overline{\mathrm{FL}}$, ordered such that only the last combination is the only one that actually causes a win ($\mathrm{FR} \cdot \overline{\mathrm{L}} \cdot \mathrm{R}$ at the cursor at 168 ps). We then see the transition to the "P1 Win" state (`ps == 1`) and verify that we are outputting a "1" to the 7seg (recall that the segments are active-low). We then try all 16 input combinations to show that we remain stuck in the "P1 Win" state.

After resetting (lower image), we then try all new input combinations with FL, with the last one causing a win ($\mathrm{FL} \cdot \mathrm{L} \cdot \overline{\mathrm{R}}$ at the cursor at 608 ps). We then see the transition to the "P2 Win" state (`ps == 2`) and verify that we are outputting a "2" to the 7seg. We then try all 16 input combinations to show that we remain stuck in the "P2 Win" state.

normalLight/centerLight

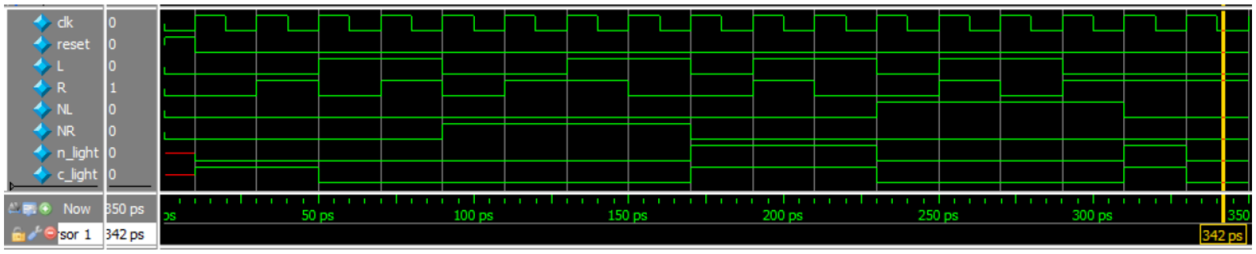


Figure 8: The ModelSim waveform for both the `normalLight` (n_light) and `centerLight` (c_light) modules.

The testbench was designed from the perspective of the `normalLight` module:

- After the reset, starting at 10 ps, we try all combinations of L and R when $\overline{\mathrm{NL}} \cdot \overline{\mathrm{NR}}$, which should result in no change to `n_light`.
- Then, starting at 90 ps, we try all combinations of L and R when $\overline{\mathrm{NL}} \cdot \mathrm{NR}$, ordered such that the last one turns on `n_light` ($\mathrm{NR} \cdot \mathrm{L} \cdot \overline{\mathrm{R}}$).
- With `n_light` on (recall: we assume we *must* have $\overline{\mathrm{NL}} \cdot \overline{\mathrm{NR}}$), we keep try the two combinations that keep the light on, then shift off to the left.
- With `n_light` back off, starting at 230 ps, we try all combinations of L and R when $\mathrm{NL} \cdot \overline{\mathrm{NR}}$, ordered such that the last one turns on `n_light` ($\mathrm{NL} \cdot \overline{\mathrm{L}} \cdot \mathrm{R}$).
- We then shift off to the right, completing the check of all valid input combinations when the light is on (combined with the range 170 – 230 ps).

Note that `c_light` correctly starts on after a reset and then turns off at 50 ps during the no-change tests for `n_light`. After this, it correctly follows the behavior of `n_light` the rest of the way.

DE1_SoC

The goal of the top-level simulation is to verify the *interconnections* between your modules – you've already shown each module working properly independently!

The top-level waveform is shown below in Figure 9. Note that the use of the `num` output from the `lightNum` module here means that we don't have to show `LEDR7`–`LEDR3`. Also recall that the use of the `singlePress` module means that we have to introduce a pulsing behavior (press and then depress) in our `KEY` inputs and that the effects of those presses are delayed by two clock cycles.

After a reset (top image), we start with only the `LEDR5` lit and `HEX5` off. We then go right by two to the right end of the play field (`LEDR3`). Then we go left all the way across the play field until a Player 2 win (past `LEDR7`, just past the 315 ps cursor). We reset again (middle image) and then go left by two to the left end of the play field (`LEDR7`). Now, on the way to a Player 1 win (past `LEDR3`), we include both players tugging (*e.g.*, the 520 ps cursor). This means that, altogether, we've tested a left tug, right tug, and both tug with every possible light state and correctly seen wins recorded (via HEX5 changes) only when expected.

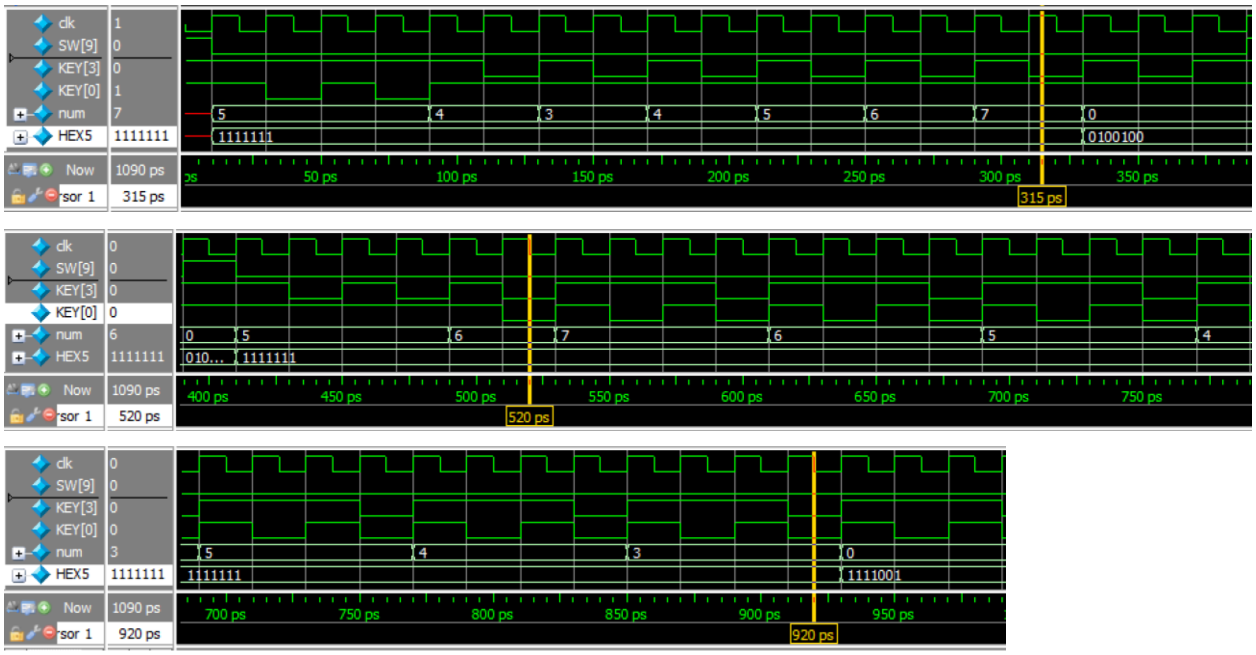
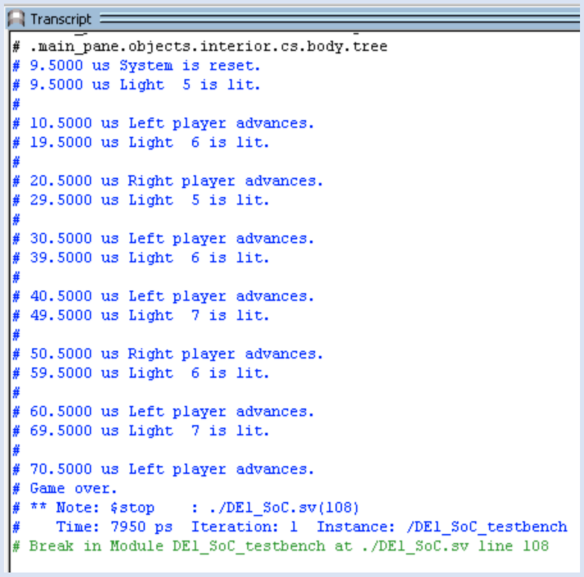


Figure 9: The ModelSim waveform for the `DE1_SoC` top-level module, split into three images for clarity.

As you can see, ModelSim waveforms can sometimes get a bit tedious to document and explain and may not be a good fit for all applications. We will learn some other SystemVerilog simulation tricks in this course to give you alternative ways of “proving” the correctness of your modules, such as the `$display` function, which could provide output in the transcript window like what’s shown below:

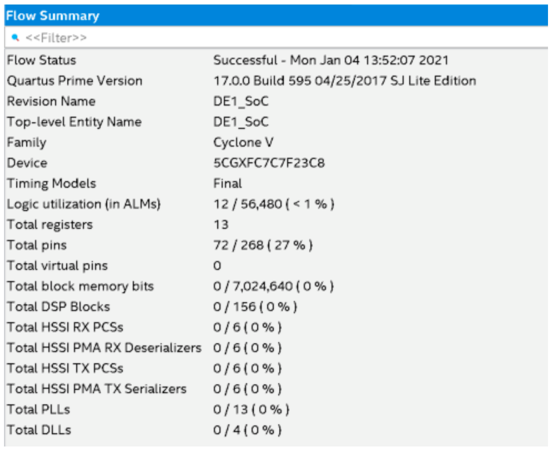


Transcript

```
# .main_pane.objects.interior.cs.body.tree
# 9.5000 us System is reset.
# 9.5000 us Light  5 is lit.
#
# 10.5000 us Left player advances.
# 19.5000 us Light  6 is lit.
#
# 20.5000 us Right player advances.
# 29.5000 us Light  5 is lit.
#
# 30.5000 us Left player advances.
# 39.5000 us Light  6 is lit.
#
# 40.5000 us Left player advances.
# 49.5000 us Light  7 is lit.
#
# 50.5000 us Right player advances.
# 59.5000 us Light  6 is lit.
#
# 60.5000 us Left player advances.
# 69.5000 us Light  7 is lit.
#
# 70.5000 us Left player advances.
# Game over.
# ** Note: $stop    : ./DE1_SoC.sv(108)
#    Time: 7950 ps  Iteration: 1  Instance: /DE1_SoC_testbench
# Break in Module DE1_SoC_testbench at ./DE1_SoC.sv line 108
```

You should decide which method (or combination of methods) to use to best illustrate your verification to the reader.

Flow Summary



Flow Status	Successful - Mon Jan 04 13:52:07 2021
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CGXFC7C7F23C8
Timing Models	Final
Logic utilization (in ALMs)	12 / 56,480 (< 1 %)
Total registers	13
Total pins	72 / 268 (27 %)
Total virtual pins	0
Total block memory bits	0 / 7,024,640 (0 %)
Total DSP Blocks	0 / 156 (0 %)
Total HSSI RX PCSs	0 / 6 (0 %)
Total HSSI PMA RX Deserializers	0 / 6 (0 %)
Total HSSI TX PCSs	0 / 6 (0 %)
Total HSSI PMA TX Serializers	0 / 6 (0 %)
Total PLLs	0 / 13 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 10: The ModelSim Flow Summary of the compilation of the `DE1_SoC` module.

Experience Report

I found this lab to be rather straightforward in what was expected and felt that the lab specification did a decent job of guiding me through the lab. It took us a while to figure out that I would need to treat the center LED and other LEDs differently. In addition, we mixed up which signals were active-low and active-high in multiple places, particularly with the `KEY` vs. `SW` signals and the `HEX` display.

One trick that I found to be particularly useful was to start by implementing a smaller system and then expanding it to what we were expected to accomplish. In practice, this mean that I first created a tug-of-war system that only used 3 LEDs – a center LED and then one on each side. Once I had this working, it was very easy and fast to add additional LEDs as all I had to do was instantiate more of the `normalLight` submodule.

This lab took me approximately 10 hours, broken down as follows:

- Reading – 30 minutes
- Planning – 30 minutes
- Design – 2 hours
- Coding – 5 hours
- Testing – 1 hour
- Debugging – 1 hour

Further Guidance for Lab Reports

- FSM diagrams, ASM/D charts, and block diagrams may be done on paper and scanned into your report; however, they must be legible.
- Photographs of your computer screen will not be accepted in place of a screenshot.
- Although you are only expected to include ModelSim screenshots of important modules in your lab report, all modules must have testbenches that accurately test their expected functionality.
- All screenshots of ModelSim waveforms should be able to be replicated by the Instruction Team without augmenting the module in any way.