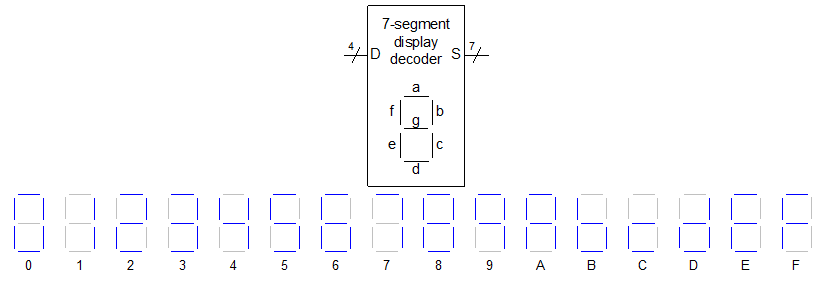
**Lab 2: Seven-Segment Display Decoder**

**Authors: C3C Lauren Humpherys and C3C Christopher Katz**

**Documentation: Captain Johnson indicated that we needed to map signal Sa to output zero and Sg to output 6, instead of the reverse order.**

**Purpose:** The purpose of this lab is to write, test and implement a seven-segment display decoder on the Basys3 development board. The decoder will take a four bit input value using SW3, SW2, SW1, and SW0. The right most display of the seven-segment displays will be activated by pressing BTNC. When activated, the display will show the hexadecimal value represented by the four switches.

**Prelab:** The first step in our design was to determine the Boolean logic needed to appropriately activate each of the seven segments. This was done according to the following diagram.

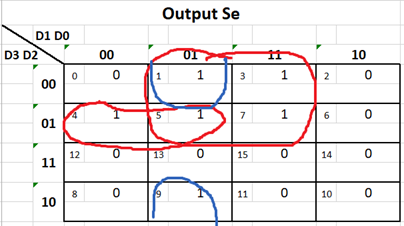


Each of the possible letters was evaluated to determine how the individual segments would be affected. The results were compiled in Table 1 below. The hex column is to make the test bench validation easier.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Hexadecimal Digit** | **Inputs** | | | | **Outputs** | | | | | | | **(in hex)** |
| **D3** | **D2** | **D1** | **D0** | **Sg** | **Sf** | **Se** | **Sd** | **Sc** | **Sb** | **Sa** |
| **0** | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40 |
| **1** | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0x79 |
| **2** | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0x24 |
| **3** | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0x30 |
| **4** | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0x19 |
| **5** | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12 |
| **6** | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0x02 |
| **7** | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0x78 |
| **8** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0x00 |
| **9** | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0x18 |
| **A** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0x08 |
| **B** | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0x03 |
| **C** | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0x27 |
| **D** | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21 |
| **E** | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0x06 |
| **F** | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0x0e |

Table 1 – Showing the mapping between possible inputs and desired outputs for each segment

Utilizing Table 1 above, we were able to derive the following boolean equations for segments Sa-Sg. If we had to implement the full SOP or POS equations in hardware, we would choose SOP equations. In order to make the implementation easier, K-maps were used to reduce the equations and can be found with the equations below



**Sa = BC'D' + ABC' + A'B'C'D + AB'CD**

**Sb = ABD' + A'BC'D + ACD + BCD'**

**Sc = ABD' + ABC + A'B'CD'**

**Sd = A'BC'D' + B'C'D + BCD + AB'CD'**

**Se = A'BC' + A'D + C'D**

**Sf = ABC' + A'B'D + A'B'C + A'CD**

**Sg = A'B'C' + A'BC’D**

**Design:**  In order to implement this lab our final design needed to match the design seen in Figure 1.

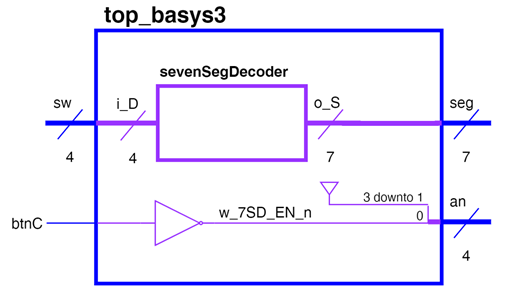


Figure 1. Top Level Schematic

Our first step in reaching this goal was to design and test the SevenSegDecoder component separately utilizing the equations Sa-Sg above and matching Figure 2.

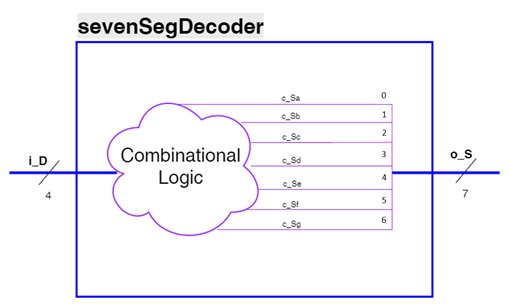


Figure 2. High Level Approach View of sevenSegDecoder

We set up our component file utilizing the provided class template and renamed it SevenSegmentDecoder. After renaming the template we defined the entity to match Figure 1 with a four-bit i\_D input and a seven-bit o\_S output. We then defined signals for Sa-Se to align with our prelab equations. After we implemented the Sa equation, we stopped and checked for syntax errors. The lab handout and the given top\_basys3.vhd file told us to map signal Sa to output 6, and so on while Sg maps to zero. Meanwhile, Captain Johnson corrected us that the order should be the other way around because we need to match the schematic in Figure 2. We implemented the next segment, Sf and Sg, utilizing a lookup table (LUT) instead of the equation. We found that we prefer implementing the LUT approach because it is simpler to create. It is also easier to find mistakes because we can quickly note whether or not lights are on when they should be. We repeated this process with each of the segments, checking syntax as we went.

Our next step was to design the test bench to ensure our component worked correctly. We modified the class testbench template and renamed it sevenSegmentDecoder\_tb.vhd. We declared our sevenSegmentDecoder and instantiated a version of it. We mapped the component to our artificially created input i\_D and output o\_S. We then created a test process to run through all possible inputs. Since the input is 4-bits we were able to use hex notation and ran through test cases from 0x0 to 0xF. The primary issue we ran into as we developed the testbench was figuring out the proper syntax. We found it difficult to keep straight when to use “<=” vs “=>,” semicolons vs commas, and whether or not the last line in a section should be appended by punctuation at all. Once the file was free of syntax errors we ran the simulation. The simulation waveform can be seen in Figure 3.

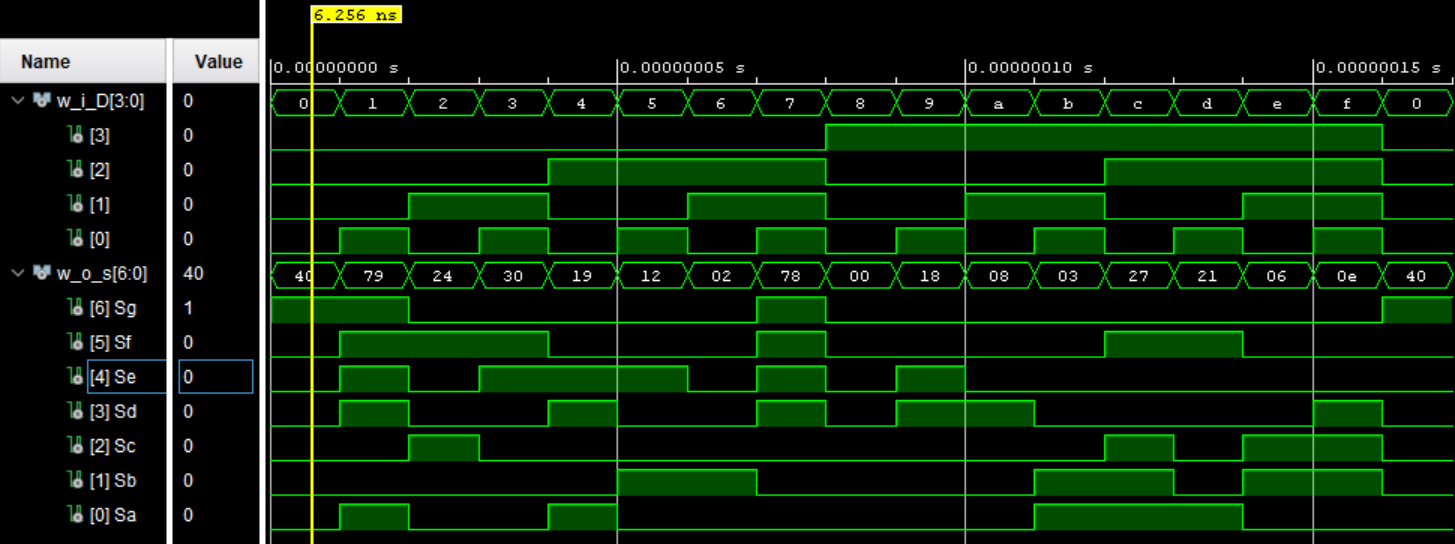


Figure 3. Simulation Waveform

We examined the waveform and confirmed that test cases 0x0 – 0xF were all run. For each test case, we checked the output vector’s hexadecimal values against the values we determined in Table 1. After verifying that they matched, we were confident that our component was operating normally.

The next design step was to create the top level design file to match Figure 1. We started with the provided top\_basys3.vhd file provided for the lab. The entity was already defined for us, so we just needed to complete the architecture. We declared our sevenSegmentDecoder and created a wire w\_7SD\_EN\_n for our button based enable. We then instantiated the component and connected i\_D to sw and o\_S to seg. The primary syntax errors that we ran into involved commas and semicolons. At first it was difficult to figure out whether a semicolon or comma should be used, with a decent chance of being completely wrong when the code requires no punctuation. We then connected our w\_7SD\_EN\_n to not btnC (since the an is 0 enabled) and connected an(0) to w\_7SD\_EN\_n. The remaining three bits of an were connected to ‘1’. The most debugging we had to do involved the relation between o\_S and seg. Most of the instructions we were given told us to map o\_S(0) to seg(6) and on down the line. Even once it was corrected, only the waveform reflected the change and the board did not change at all. After going in for EI and trying the same exact thing on Captain Johnson’s system, interestingly enough it worked.

**Final Results:** Once we finished creating our design in VHDL we looked at the RTL schematic for both the top\_basys3 design and the sevenSegmentDecoder design. Figure 4 shows top\_basys3.

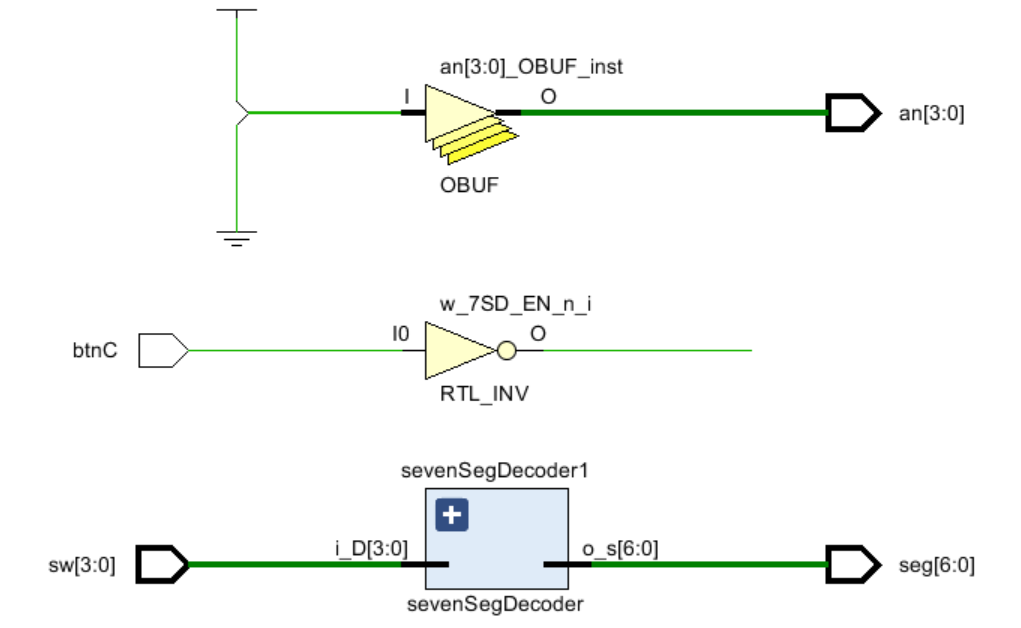


Figure 4. Top\_Basys3 RTL Schematic

The design shows the 4 anode inputs, the center button with its inverter, and the seven segment decoder with its four inputs and seven outputs. The reason the wires aren’t connected is because btnC’s default value is “1,” which in this case is off. When the button is depressed, the wires will be connected.

Figure 5 shows sevenSegmentDecoder.

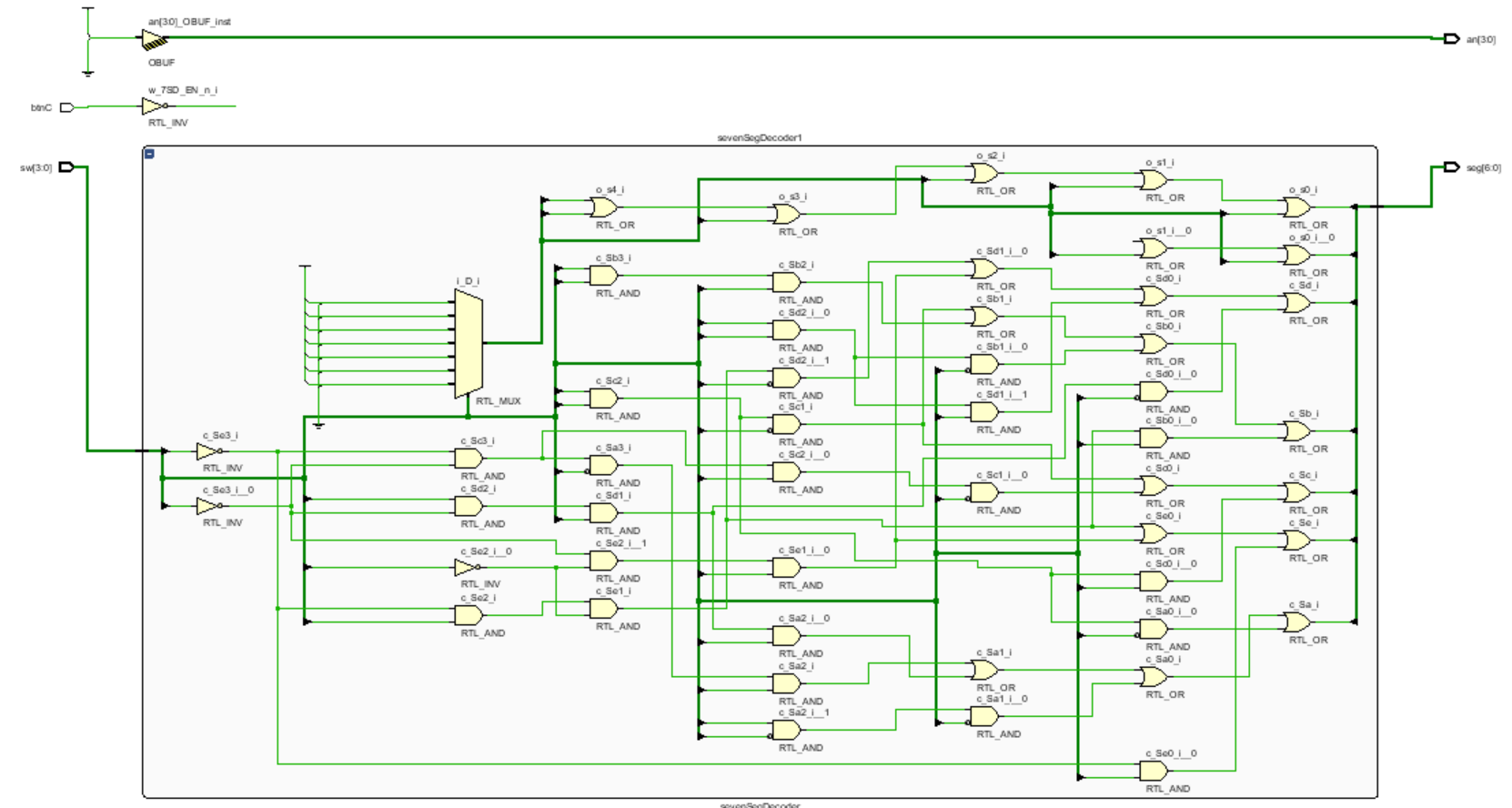


Figure 5. SevenSegDecoder RTL Schematic

The design for the decoder is composed of several lots AND gates, OR gates, three inverters, and a 7:1 multiplexor.

Our final step was to generate the bitstream for our project. Because the top\_basys3 utilized standard notation for sw and seg, we only had to uncomment the respective lines in the Basys3\_Master.xdc file. We initially failed to uncomment the lines that enabled the anodes for the board, but figured out that we needed to uncomment them when the board was not lighting up. We pushed the bit stream to our board and successfully demonstrated all 16 possible inputs to Captain Johnson.

**Conclusions:** Throughout this lab we learned how important it is to focus on the details, especially syntax. Consequently, we learned a lot about VHDL syntax through much practice and analysis of previous projects. Second, we learned how important communication is concerning instructions. Only one partner had heard that our original instructions were faulty, and the other partner was charged with that section of the coding. The most fascinating part of this lab was that changing a major portion of the code—the mapping of o\_S(0) to seg(6)—had absolutely no initial impact on the board. Even after closing the program and regenerating the bitstream the board stayed the same. It was not until the following day, when Captain Johnson typed the exact same thing we typed, that the board finally corrected to match the waveform.

**Reflection:**

* **Number of hours spent on Lab2 (Combined):** ****
* What portion of the lab was the most difficult for you? How did you overcome it?
  + Debugging syntax errors was the most difficult challenge for us, especially considering that the Vivado console is generally not very specific in its error messages. Therefore, much trial and error had to be done in order to overcome them.
* What lessons, previous assignments, or activities did you find helpful is completing this lab?
  + ICE2 was the most helpful reference for Vivado file setup and coding formats. Code from Lab 1 was also useful in building the entities and architecture of our circuit.
* What suggestions do you have for improving Lab2 in future years? Be specific. Ex: “The instructions were confusing” does not help. What parts of the instructions were confusing?
  + Before doing this lab, it may be helpful to have a lab just on debugging. Here, the professor can hand us a premade program with errors, and have students practice deducing errors and fixing them.