ECSE 323 - Digital System Design
After Action Report - Lab 3

Harley Wiltzer (260690006)
Spiros-Daniel Mavroidakos (260689391)

February 23, 2017

Contents

Part I

A Modest Overture

"My son, ask for thyself another kingdom; for that which I leave is too small for thee."

- Iron Maiden - Alexander the Great, Somewhere In Time(1986)

SINCE the dawn of time, mankind has sought to make things smaller. Not so long ago, an average computer would take up an entire room, and yet by the early 2000's Ultrium tapes were capable of storing 100GB of data in a cartridge the size of a deck of cards. Some computers today, such as the Raspberry Pi, are approximately the same size as a credit card. In fact, scientists are now even capable of genetically modifying pigs to make them smaller.

The gorgeous part of this past laboratory experiment is that a complex machine was created using hardware alone. The design was carried out without the use of any software. No sticky, hot, messy software; no software at all. What a beautiful idea! Of course, the evident consequence of this is that the system was created with very little overhead, and operates very quickly.

To illustrate this, consider the micro-pig. These pigs exhibit the exact same properties as their

Figure 1: Genetically-modified miniature pig [1]

regular-pig counterparts, but they were genetically modified to be smaller. They are all the pig one can ask for, but in a smaller package. Engineers designed these pigs with similar ideas to those that designed the Raspberry Pi, and this is not surprising: if something can be designed to be smaller without sacrificing performance, it will be done for the mere convenience of it.
This phenomenon is what was exhibited, more than anything else, in this laboratory experiment. Fifty years ago, if two individuals had desired to play a simple game of cards, they would have to use a deck of cards. The designs that were developed over the past two weeks will eventually allow two individuals to play a game of cards using an FPGA that is smaller than a deck of cards.

Of course, such a feat was only possible due to the magic offered by Altera's Quartus II and Modelsim software. Without it, the schematics and VHDL code that make up the design of this system would never make it from the computer to the FPGA.

The process of designing what will effectively be the card deck for a digital crazy-eights game will be documented thoroughly in the following pages of this report. Details concerning the testing of the system will be provided, and insight concerning the design tactics will be given. Please enjoy reading the remainder of this *exposé* as much as the authors did writing it.

Part II

The `g07_stack` Circuit

Introduction

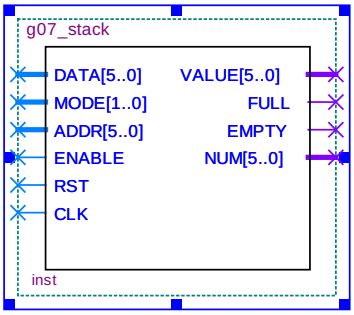
The majority of this report will be concerned with the design of the `g07_stack` circuit. Note that the `g07_stack52` circuit is merely a circuit composed solely of the `g07_stack` block, which was used as a convenience for testing and integration purposes.

The goal of the `g07_stack` circuit was to implement a stack-like data structure for the Cyclone II FPGA. This structure will later be used to store a deck of cards for a crazy-eights game. A stack is a canonical data structure used in computing for dynamic memory allocation. A traditional stack follows a last-in, first-out paradigm, meaning data is added to the *top* of the structure and also removed at the top. However, such a paradigm is not flexible enough for the purposes of the `g07_stack`. Not only should this circuit be capable of pushing data to the top of the stack and *popping* data back off, but it should also be able to access and remove data from an arbitrary location within the structure. However, according to D.A. Lowther, author of "Computer Engineering", "data can only be retrieved from the top of a stack"[2].

Therefore, it is clear that the use of "stack" in the context of the `g07_stack` is a misnomer. Rather, the `g07_stack` is more akin to a Bi-directional Jenga® Tower (BJT), which allows objects to be pushed to the top safely, and can ideally have objects removed from within without causing pandemonium. Unfortunately, due to constraints on the project, the `g07_stack` must keep its current name. However, it is recommended that the reader think of the `g07_stack` as a BJT rather than a stack for its functionality to make more intuitive sense.

Circuit Description

Figure 2: Pin-out diagram of `g07_stack` circuit



Above is the pin-out diagram corresponding to the `g07_stack` circuit. The input and output ports are described below.

DATA[5..0]

The `DATA[5..0]` port takes in a 6-bit bus which describes data that should be pushed to the BJT when a push operation is specified.

MODE[1..0]

The `MODE[1..0]` port takes in a 2-bit bus which represents the operation that is to be done on the BJT. Below are the configurations of `MODE` and their meanings:

- 00: NOP - No operation is done
- 01: PUSH - The value stored in `DATA` is pushed to the top of the structure (if it isn't already full). The value output at `NUM` is updated accordingly.
- 10: INIT - Values 0-51 are stored in the structure such that the top memory cell stores 0 and the bottom one stores 51. The value at `NUM` is set to 52, and `FULL` is set to high.
- 11: POP - The value stored in the cell determined by the `ADDR` port is removed from the stack(if it isn't empty), and data below this address are shifted upward to remove the gap. The value output at `NUM` is updated accordingly.

ADDR[5..0]

The ADDR port is a 6-bit bus that stores the address of the memory cell to be popped or displayed.

ENABLE

The ENABLE bit controls when the structure is enabled. When ENABLE is low, the structure remains unaffected by the other inputs.

RST

The RST bit, when high, asynchronously resets all memory blocks to store 0, and thus resets NUM to 0, and sets EMPTY high.

VALUE[5..0] - Output

The VALUE output port outputs the value of the data stored in the memory cell designated by the ADDR input.

FULL - Output

The FULL output port is set high when the value of NUM is 52, and low otherwise.

EMPTY - Output

The EMPTY output port is set high when the value of NUM is 0, and low otherwise.

NUM[5..0] - Output

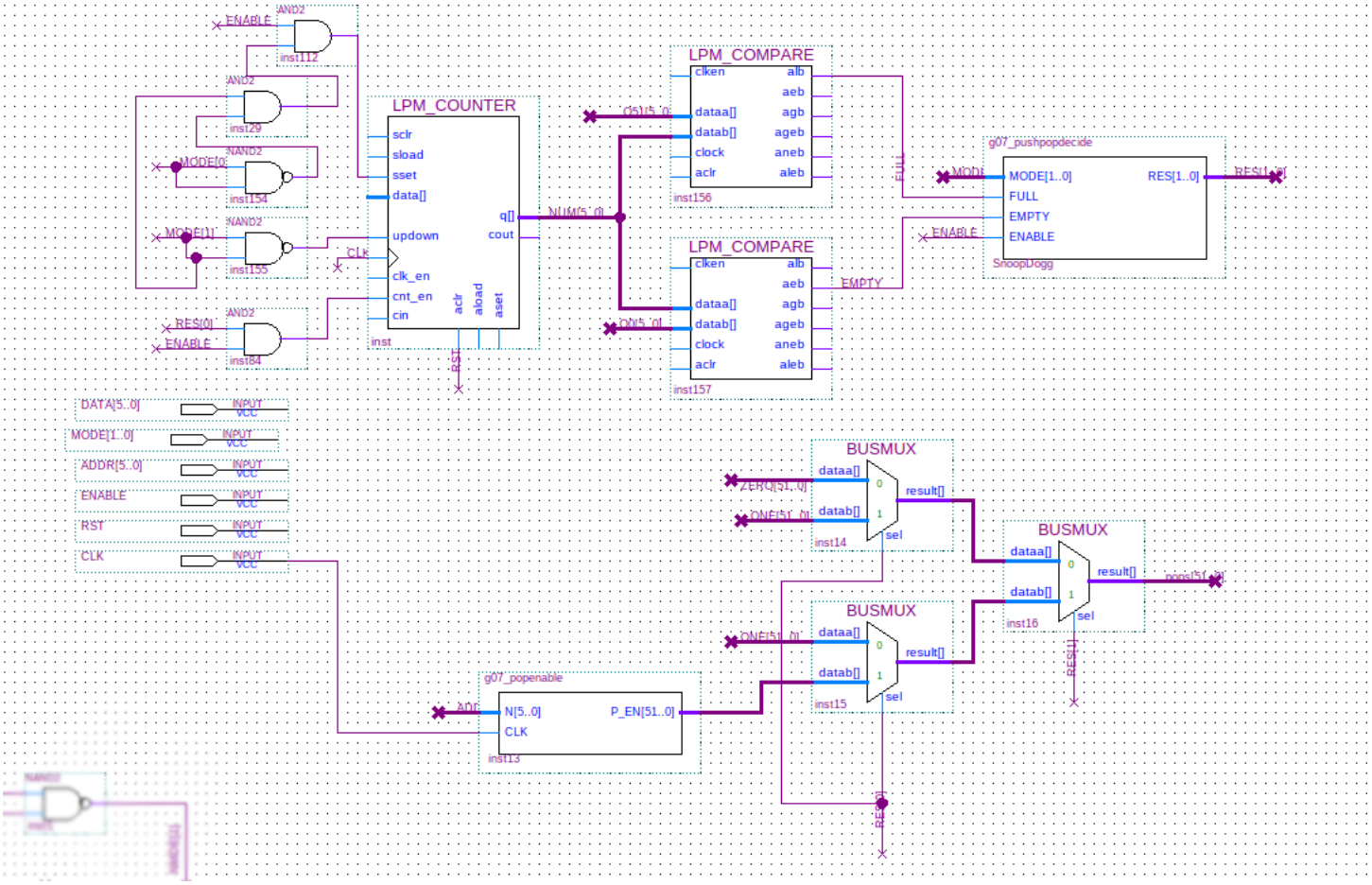
The NUM output bus outputs a 6-bit vector representing the amount of cells being used by the BJT. Initially, when empty, the NUM outputs 0. When data is pushed, NUM increments, and when data is poppsed, NUM decrements.

Gate-Level Schematic

Due to the sheer size of the `g07_stack` circuit, it is much more convenient to show the gate-level schematic broken down into multiple parts. Each part will be described individually.

Enable Section

Figure 3: Enable section of the circuit - controls the enable bits of the individual flip flops.

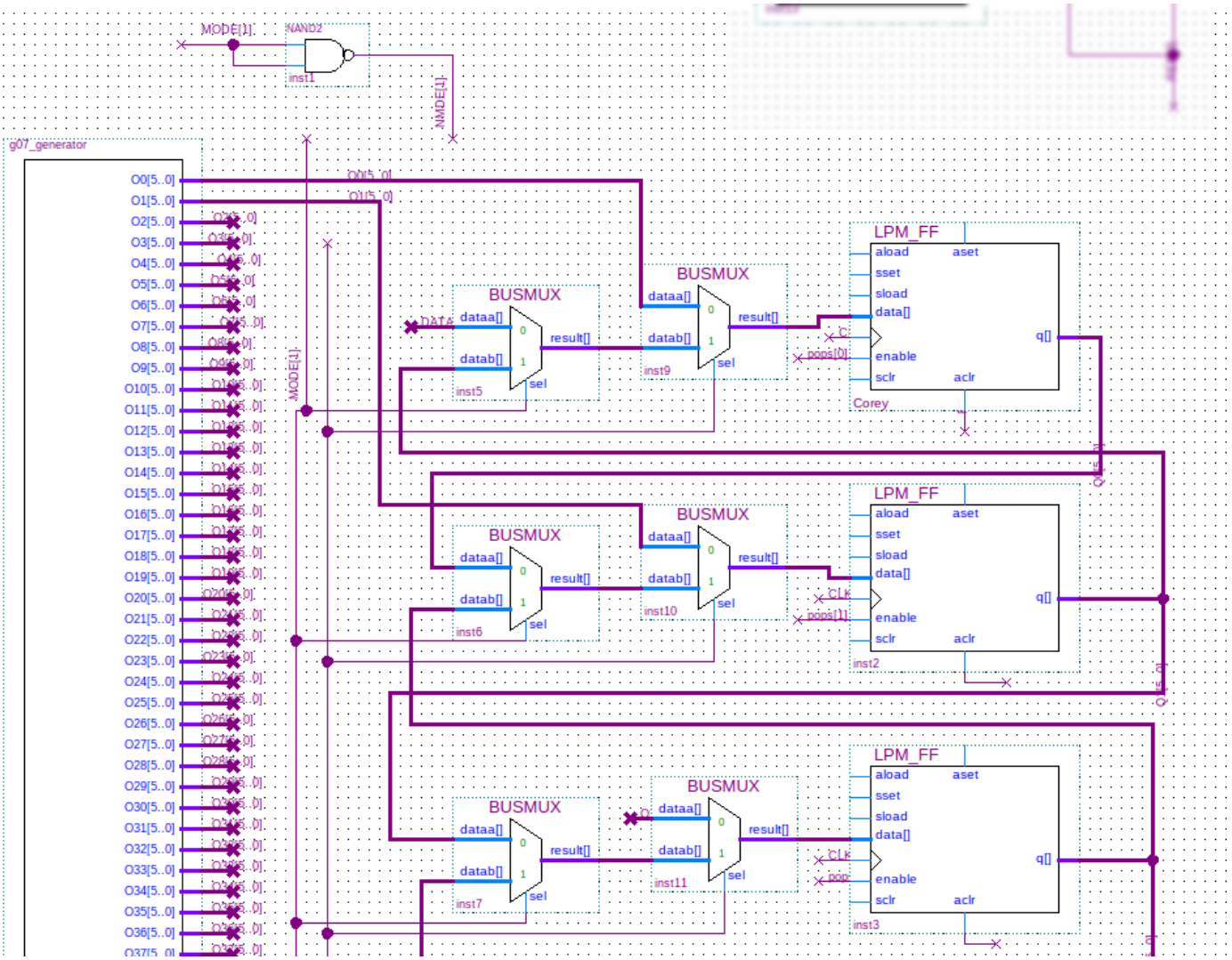


The enable section of the `g07_stack` circuit is responsible for managing the enable bits on the individual flip flops in the memory array. Flip flops must be enabled for their data to change, however, depending on the operation requested, the flip flops that modify data may change. The upper

half of this section includes the counter used to specify the `NUM, FULL, EMPTY` outputs. These outputs, as well as the `ENABLE, MODE` inputs are passed to the `g07_pushpopdecide` circuit, which sends a modified `MODE` signal called `RES` to the lower circuit. The `g07_pushpopdecide` circuit transforms a push instruction to a NOP if the BJT is full, and transforms a POP instruction to a NOP if the BJT is empty. Next, the `RES` signal is passed through a 4-1 multiplexer to choose the array of enable bits to send. When PUSH or init are specified, the MUX outputs an array of 1's, one NOP is specified the MUX outputs an array of 0's, and when POP is specified the MUX relays the output of the pop-enable ROM that was designed several weeks ago. The output bits of the MUX are then passed to the `pops` signal.

Memory Array Section (Top)

Figure 4: Top of the memory array section of the circuit - manages data to be stored in the BJT



The above diagram shows the top of the memory array. At the left end of the diagram is the `g07_generator` subcircuit, which was designed to store the constants 0-51 as 6-bit vectors.

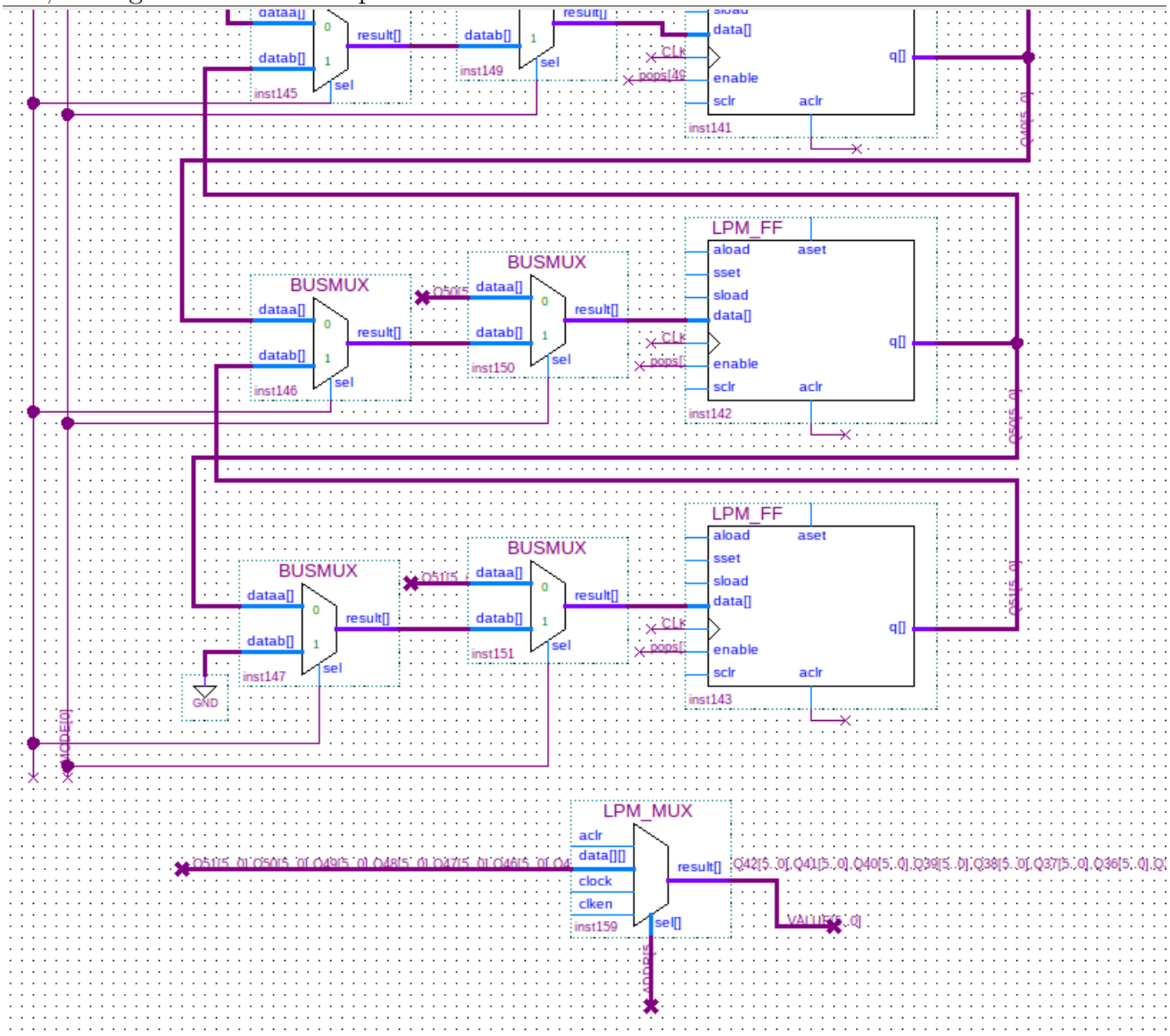
For convenience, a 52-bit array of 1's and a 52-bit array of 0's was also included. This circuit was designed with VHDL code that was generated with Haskell code, which can be perused here. Between the `g07_generator` and the flip flops lie two columns of MUX's. Each flip flop is associated to two MUX's which choose which data to pass to its `data` input. The leftmost MUX's take in the data from the flip flop above it and the data from the flip flop below it, so as to accomodate data shifting in both directions (which is why it's called a *bidirectional* Jenga® tower). This MUX takes the most-significant bit of the `MODE` input as its select line. Then, the rightmost MUX takes in the output of the leftmost MUX and the corresponding constant from the `g07_generator` as its data inputs, and takes the least-significant bit of the `MODE` signal as its select line. This MUX chooses whether to load data from another memory cell (push or pop) or load a constant (init) to its corresponding flip flop.

The BJT continues on in this fashion all the way to the bottom, and is not shown in its entirety since no additional logic is available. However, the bottom portion of the BJT will be displayed, which shows some extra circuitry.

Memory Array Section (Bottom)

Here, the last three memory cells of the BJT are displayed. Note that the bottom memory cell has a grounded input at its leftmost MUX, which effectively loads 0 to the bottom element when a pop operation is carried out. Finally, below the bottom memory cell lies the logic that manages the `VALUE` output. It takes the form of another MUX, which takes every flip flop value as data inputs, and uses the `ADDR` input as its select line.

Figure 5: Bottom of the memory array section of the circuit - manages data to be stored in the BJT, manages the VALUE output.



Part III

On the Testing of the `g07_stack` Circuit

Overview

Naturally, with a circuit as complex as the g07_stack, it is imperative that testing is done thoroughly. Of course, the g07_stack supports four commands, which, when considering branches in logic depending on the current state of the BJT, may support an unwieldly amount of input combinations. Thus, in the effort to test the g07_stack circuit, the goal was simply to test each functionality and its edge cases rather than each possible input scenario.
To do this, the g07_testbed was created. This circuit is composed of the g07_stack, as well as the simple mod13 circuit and some g07_7_segment_decoder circuits. The testbed was created so that the g07_stack could be tested on the Altera DE1 board and produce visual, comprehensive output. Please consult this appendix for the testbed's VHDL code.

However, the elements that have been described are not good enough on their own for the practical testing of the g07_stack on the DE1 board, as they do not account for the all-feared button bounce problem. For the interested reader, a description of this problem and how it was solved is included in the interlude below.

The g07_debounder Circuit: A Brief Interlude

When dealing with physical buttons, all hell may very well break loose. Unless the button is ideal (and it isn't), pressing the button once will not actually look like one button press to the electronics that the button is connected to.

This is what is known as the button bouncing problem, and it must be solved immediately to carry out the desired tests. It was decided that a *debouncing circuit* should be created, which led to the conception of the g07_debounder[1]. The purpose of this circuit was to remove the "bouncing" of the button output. But what *is* bouncing? Is bouncing the process of moving along in a lively manner while repeatedly striking the surface below and rebounding? Or is bouncing the act of removing a vicious intoxicated person from Sir Winston Churchill's? For the sake of the g07_debounder circuit, bouncing will be modeled as a phenomenon similar to the former.

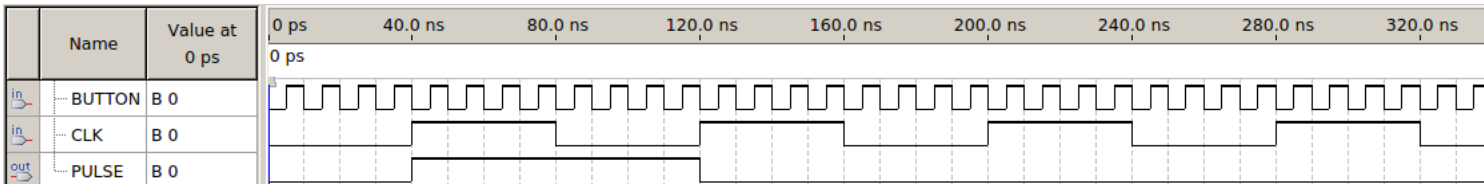
Button presses tend to cause the inner metal contacts to *bounce*, which causes the circuit to open and close repetitively in an unpredictable manner. Therefore, without the g07_debounder circuit, each button press would actually cause multiple pulses to be sent to the g07_stack, which is a severe issue. The g07_debounder circuit was designed to only register one clock pulse of

[1]Notice the simple typo in the name of the g07_debounder. This typo was committed by Harley Wiltzer when rushing through the design of the circuit, which was done during a time when he had "other work to do". Later on, when the success of the circuit put Wiltzer and Mavroidakos in a state of enlightenment, they realized that the typo was present. However, they left the typo there for sentimental purposes.

high output within a 2^{23} clock-pulse period. This sets a limit on how frequently button presses can be registered, but it also removes the possibility of registering multiple pulses from one button press.

To achieve this functionality, a Moore-type finite state machine was designed. This machine includes two SR Flip Flops to store the current state of the machine, as well as a 23-bit counter to count clock pulses since the button was first pressed. The counter output is fed to a comparator which keeps the machine in a certain state until the count goes back to 0. The circuit may be observed here.

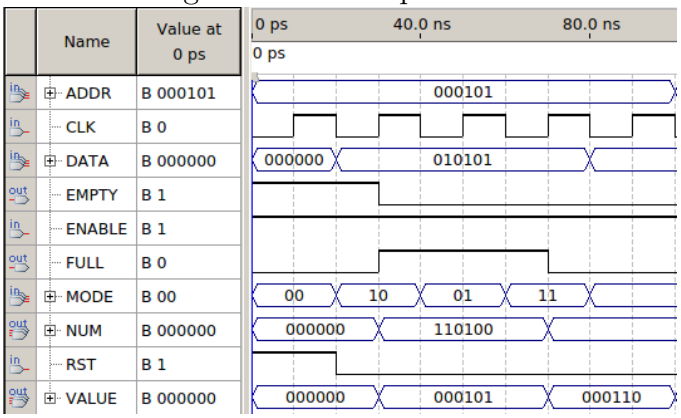
In case the reader is still skeptical of the functionality of the `g07_debounder`, the functional analysis results are given below. Note that both inputs, the button and the clock, are modeled as clock signals. The button has a higher frequency to represent the signal *bouncing* within a clock cycle. Notice that although the button oscillates from high to low several times, the output of the circuit is high for only one clock pulse.



Testing the `g07_stack` Circuit

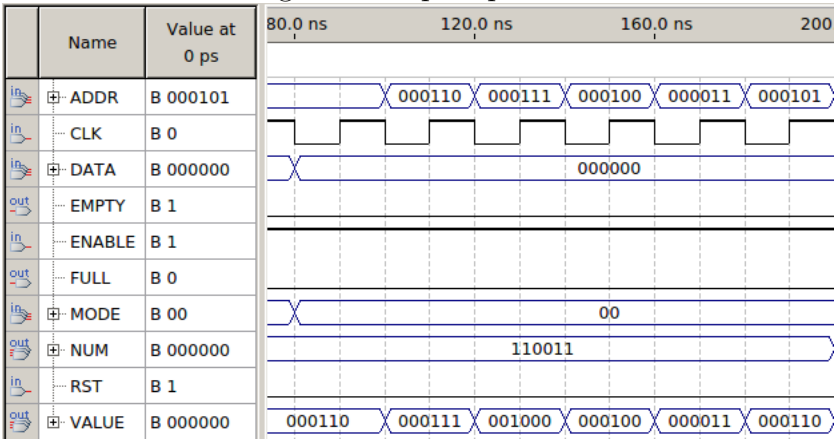
Before testing the testbed on the DE1 board, it was decided to do a thorough test of the `g07_stack` circuit's core functionality on its own. This made drawing test case waveforms for all of the BJT operations simpler. Sample output waveforms are given below. This sequence shows that the BJT

Figure 6: Initial operations



initializes in an empty state, and after calling init (mode = 10), `FULL` goes high and `EMPTY` goes low. Next, a push operation (mode 01) is attempted, but no data changes because `FULL` is high. Then, after pop (mode 11) is called, `FULL` goes low and the value stored in address 000101 goes from 000101 to 000110, showing that the element underneath it was shifted upward after the pop, as expected. It can be seen from the next image that after switching the address to 000110, the

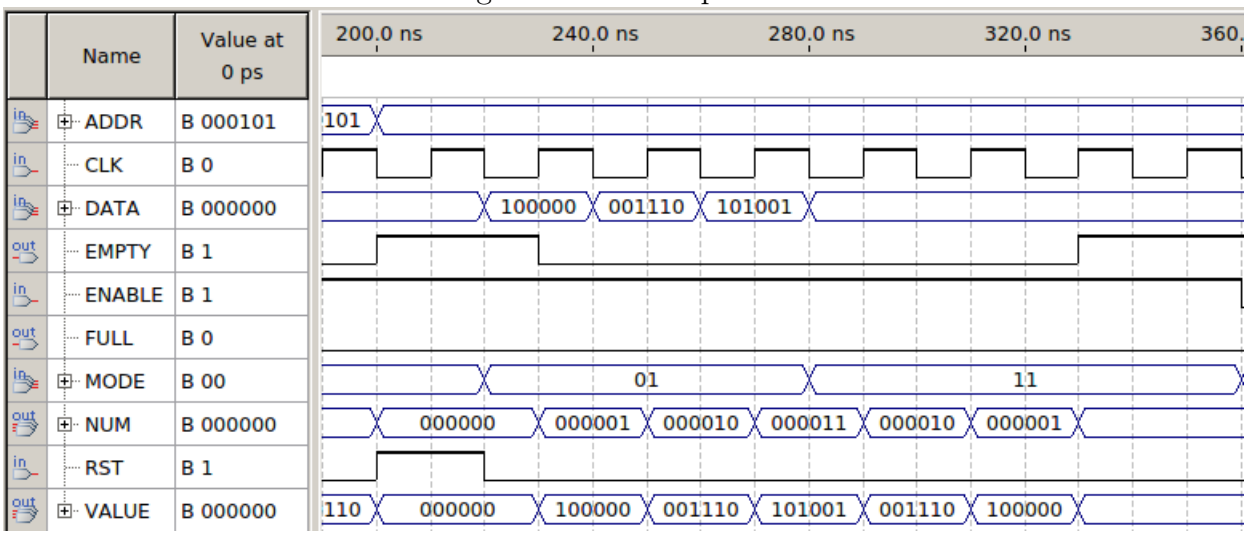
Figure 7: Pop inspection



VALUE is 000111, showing that an element below the pop source received shifted data as well, which is expected. Then, address 000111 is expected showing again that shifting below the pop source is functional. Next, address 000100 was observed to be storing 000100, showing that data above the source of the pop was not affected, which is also expected. This trend was consistent at address 000011 as well. Note that the value of NUM is 110011, representing 51 in decimal, because one element was popped.

At this stage, RST was set high for a clock pulse, and consequently NUM, as well as the data

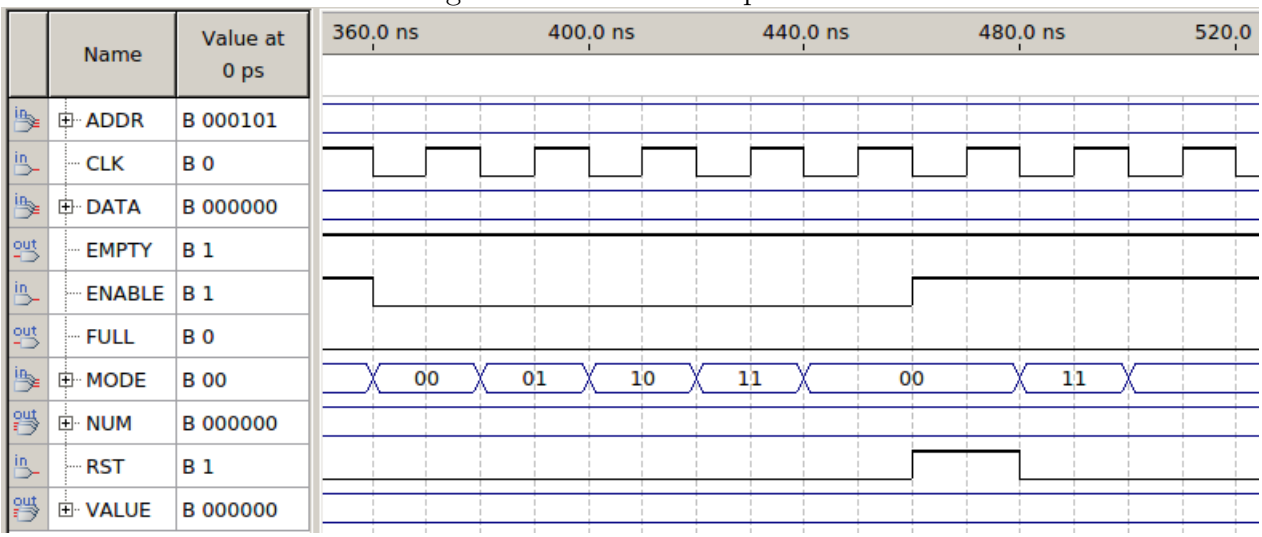
Figure 8: Push inspection



stored in the flip flops, are set to 0. Following the reset, the numbers 100000, 001110, and 101001 are pushed to the stack, and are observed sequentially at address 0. Furthermore, NUM increments after each push.

Then, ENABLE was set low while each operation was attempted. It can be seen that in the interval

Figure 9: Resistence inspection



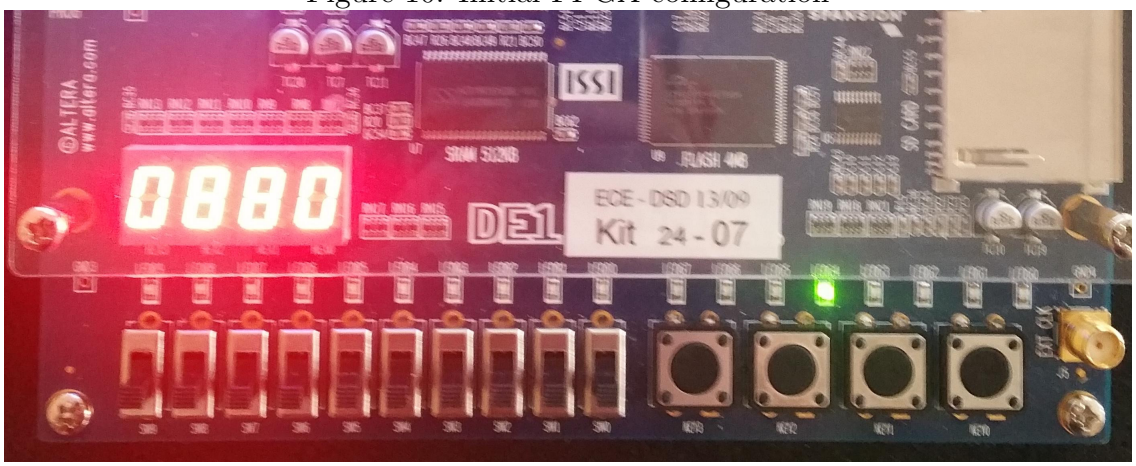
where ENABLE is low, no other outputs change, which is expected. Finally, RST was set high once again, bringing NUM to 0 and EMPTY high. A pop operation was then attempted (mode 11), but it can be seen that the value of NUM does not change, and no other outputs change, which is expected.

The series of tests shown above test all intended functionalities and conditions of the BJT, and all were shown to be successful.

Testing the Testbed on the Hardware

Now, the testbed is ready to be tested on the Altera DE1 board. Rather than showing functional simulations and output waveforms for the testbed (which would be hard because the outputs are 7 segment display codes), a series of photographs was taken to give evidence of the testbed's functionality. Here is how the FPGA board looks when the testbed is first loaded. The buttons,

Figure 10: Initial FPGA configuration

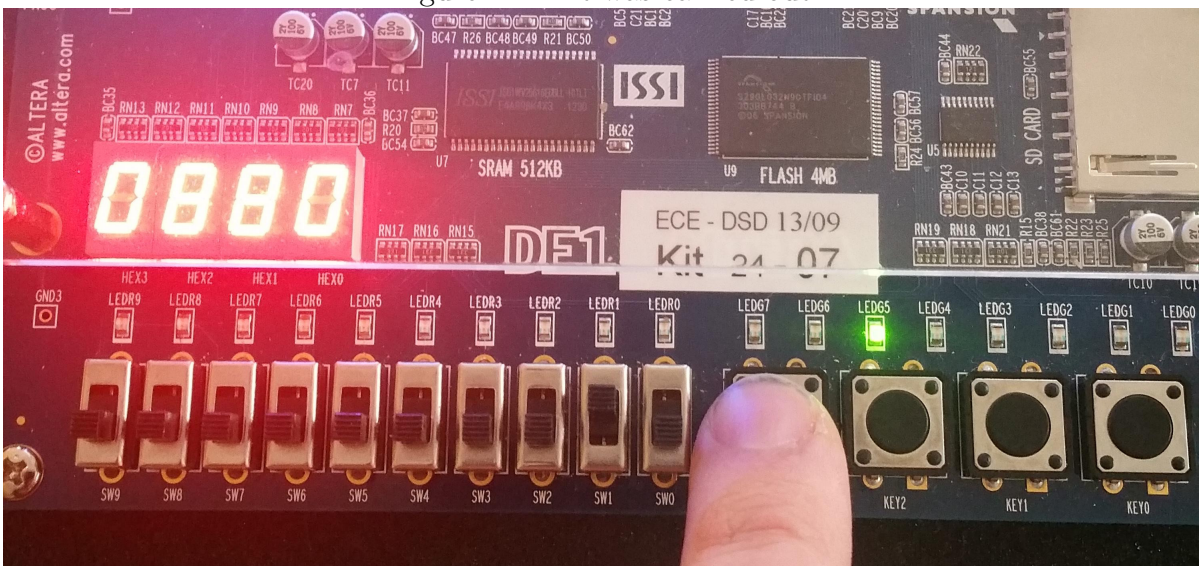


switches, and LED's of interested are listed below:

- The leftmost 7 segment display represents the `VALUE` output of the stack at a given `ADDR`, modulo 13.
- The rightmost 7 segment display represents the floor of the `VALUE` output at `ADDR` divided by 13.
- The six leftmost switches determine the `ADDR` to get values from in the BJT.
- The two rightmost switches determine the `MODE` input of the BJT.
- The leftmost button controls the `ENABLE` input of the BJT.
- The rightmost button controls the `RST` input of the BJT.
- The rightmost LED above the second pushbutton from the left (the LED that is illuminated in the first picture) is illuminated when the `EMPTY` output of the `g07_stack` is high.
- The LED to the left of the `EMPTY` LED is illuminated when the `FULL` output of the `g07_stack` is high.

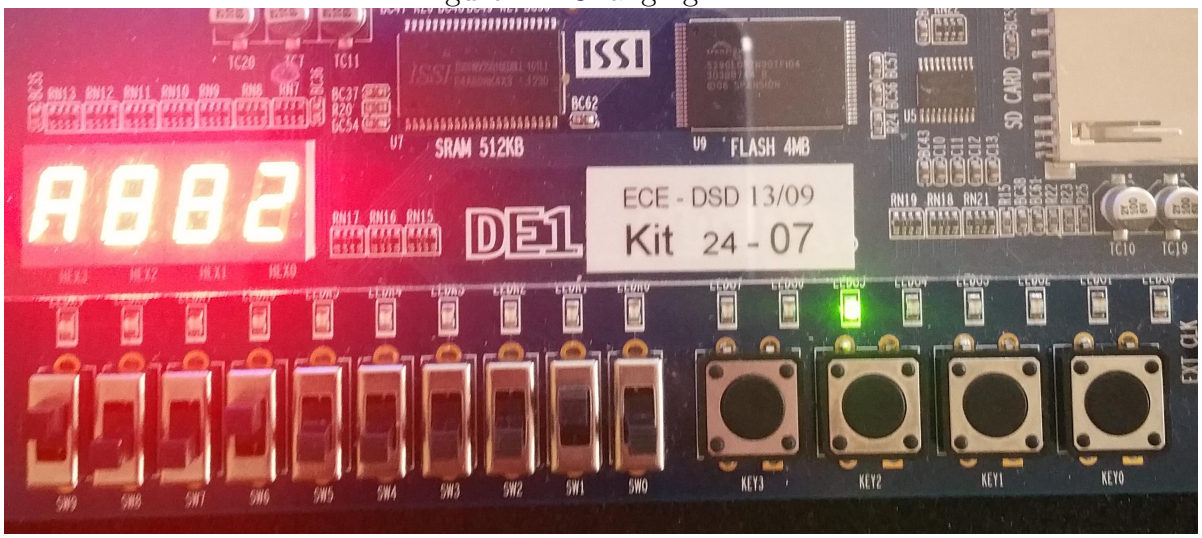
In the initial configuration, the stack is empty, so surely both 7 segment displays show zero, and the EMPTY button is illuminated.

Figure 11: Init was carried out



Next, the MODE switches were set to 10 to indicate an init operation, and the enable buttonw was pressed. Since ADDR is still 0, the 7 segment displays show zero, however it is important to note that the EMPTY LED is now off, and the FULL LED is on.

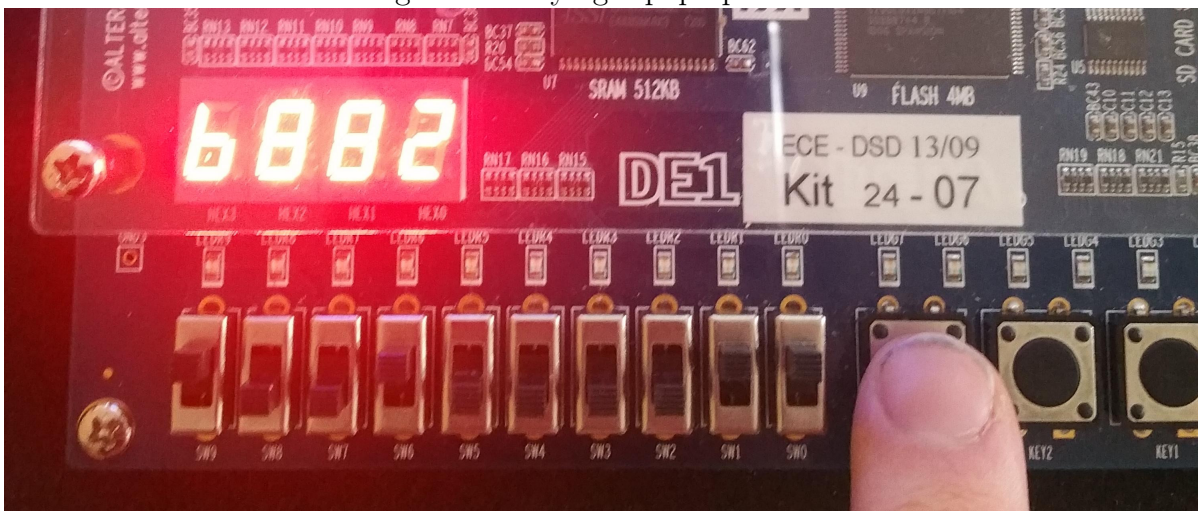
Figure 12: Changing ADDR



At this stage, ADDR was changed to a value of 100100. The 7 segment displays show A2, which represents $2 * 13 + A = 26 + 10 = 36_{10} = 100100_2$. Thus, the data is correct.

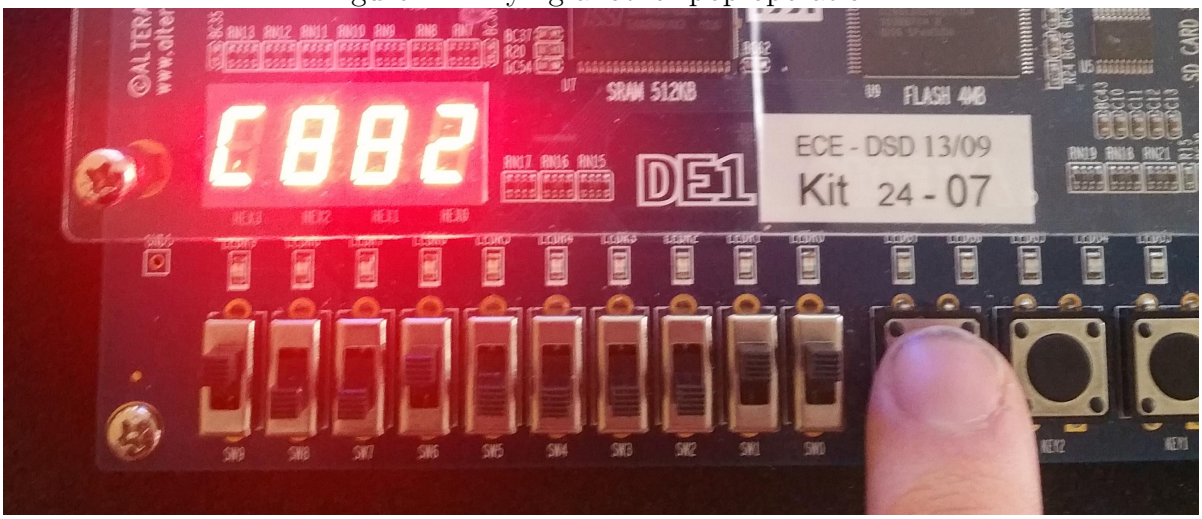
Next, the mode buttons were set to 11, indicating a pop operation. The enable button was pushed once and the 7 segment displays show B2, which is one larger than A2. Therefore, the data from the cell below 100100 was shifted into 100100, implying that *one* pop operation has taken

Figure 13: Trying a pop operation



place. This confirms the functionality of the `g07_debounder`. Furthermore, it is clear now that both `EMPTY` and `FULL` LED's are off, because now the stack is neither full or empty.

Figure 14: Trying another pop operation

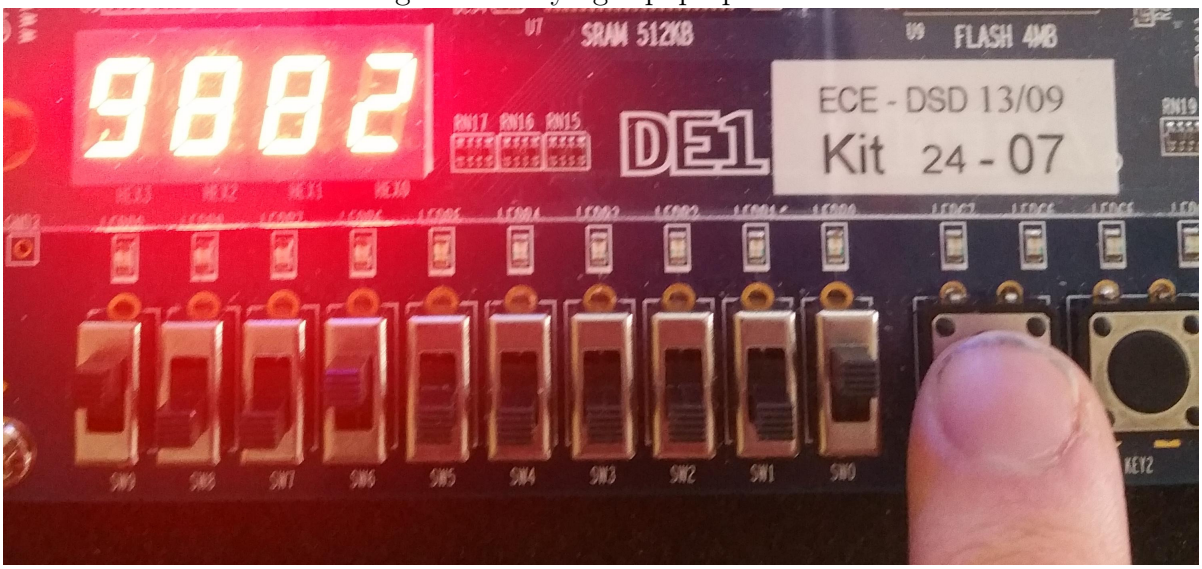


Then, the enable button was pushed again, allowing for another pop operation. This time, the 7 segment display shows C2, which is one greater than B2, so another pop operation was shown to be successful.

Finally, the mode buttons were set to 01, indicating a push operation, and the enable button was pressed. The 7 segment displays show 92. Since the values A2 and B2 were popped off the stack, the next value on the BJT above the current address must be 92 (one less than A2), which shows that the BJT's push and pop are operating properly, and once again shows that the `g07_debounder` is working.

It is important to note that the purpose of these tests on the DE1 board was not to test the functionality of the BJT implementation itself, but rather the testbed. It was already shown

Figure 15: Trying a pop operation



that the `g07_stack` is functional, so all that was left to be shown was the functionality of the `g07_debounder`, the proper codes being sent to the 7 segment displays, and the proper pin assignments to the DE1. All of these functionalities were shown to be successful in these tests.

Part IV

The Results

The Plight of the `g07_stack` and its Companions

The following data were taken from Quartus II's divine analysis of the testbed circuit. Note that the testbed circuit contains a `g07_stack` as well as the 7 segment decoders, the `mod13` circuit, and the almighty `g07_debounder`.

The Flow Summary

Figure 16: Tabulated results from Quartus II's Flow Summary

Total Logic Elements	4%
Total Combinational Functions	4%
Dedicated Logic Registers	2%
Total Registers	344
Total Pins	9%
Total Virtual Pins	0
Total Memory Bits	1%
Embedded Multiplier 9-bit Elements	0
Total PLL's	0

The table above shows statistics regarding how much of the Cyclone II's hardware is being used by the testbed. These results are more-than-reasonable considering the magnitude of the circuit being implemented.

The Timing Analysis

To determine the feasibility of the testbed with regard to speed, Quartus II's TimeQuest Timing Analyzer was used. This tool was used to calculate the longest path of data within the testbed, and reported that this path required a maximum clock frequency of 158.93 MHz. Given the 50 MHz clock frequency maximum on the DE1 board, this restriction is not very restricting, and is perfectly acceptable.

Appendices

Appendix A

Haskell Code for `g07_generator`

```
1  import System.IO
2
3  generatePorts :: Integer -> String -> String
4  generatePorts (-2) s = s ++ "\t\tONES: OUT STD_LOGIC_VECTOR(51 downto 0)"
5  generatePorts (-1) s =
6      generatePorts (-2) $ s ++ "\t\tZERO: OUT STD_LOGIC_VECTOR(51 downto 0);\n"
7  generatePorts n s =
8      generatePorts (n-1) $ s ++ "\t\tO" ++ (show n) ++ ": OUT STD_LOGIC_VECTOR(5
           downto 0);\n"
9
10 generateLogic :: Integer -> String -> String
11 generateLogic (-2) s = s ++ "\t\tONES <= X\"FFFFFFFFFFFF\";\n"
12 generateLogic (-1) s =
13     generateLogic (-2) $ s ++ "\t\tZERO <= X\"0000000000000\";\n"
14 generateLogic n s =
15     generateLogic (n-1) $ s++"\t\tO"++(show n)++" <= STD_LOGIC_VECTOR (to_unsigned(
           "++(show n)++",6));\n"
16
17 main = do
18     fh <- openFile "generator.vhdl" WriteMode
19     hPutStrLn fh $ "library ieee;\nuse ieee.std_logic_1164.all;\nuse ieee.
           numeric_std.all;\n"
20     hPutStrLn fh $ "entity g07_generator is"
21     hPutStrLn fh $ "\tport ("
22     hPutStrLn fh $ (generatePorts 51 "") ++ ");";
23     hPutStrLn fh $ "end g07_generator;"
24     hPutStrLn fh $ "\narchitecture behavior of g07_generator is"
25     hPutStrLn fh $ "begin"
26     hPutStrLn fh $ generateLogic 51 ""
27     hPutStrLn fh $ "end behavior;"
28     hClose fh
```

Appendix B

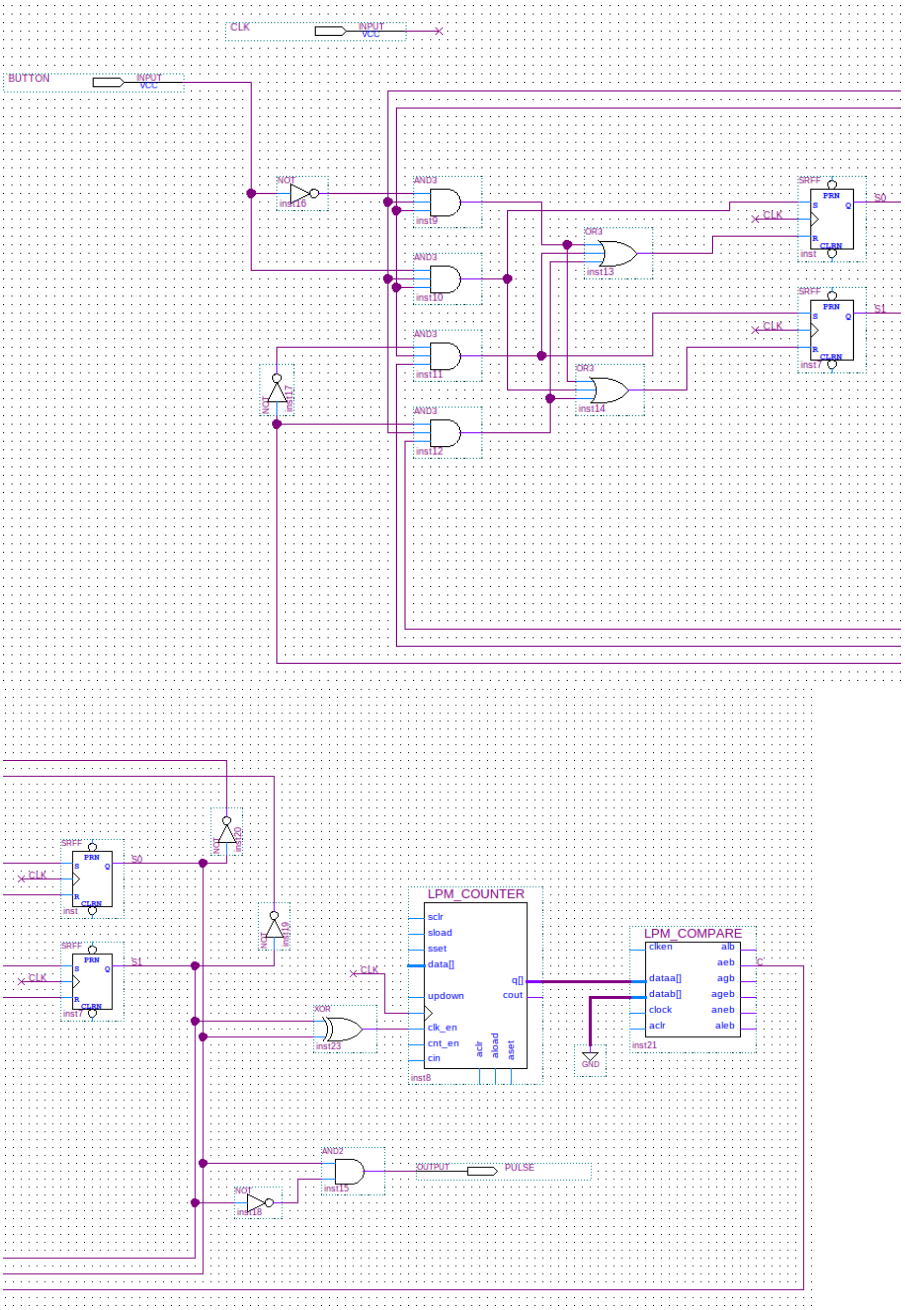
VHDL Code tor `g07_testbed`

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity g07_testbed is
5          port (
6                  b: in std_logic;
7                  clk: in std_logic;
8                  mode: in std_logic_vector(1 downto 0);
9                  addr: in std_logic_vector(5 downto 0);
10                 rst: in std_logic;
11                 led1, led2: out std_logic_vector(6 downto 0)
12         );
13 end g07_testbed;
14
15 architecture behavior of g07_testbed is
16         component g07_debounder
17                 port (
18                         button: in std_logic;
19                         clk: in std_logic;
20                         pulse: out std_logic
21                 );
22         end component;
23         component g07_stack
24                 port (
25                         mode: in std_logic_vector(1 downto 0);
26                         data: in std_logic_vector(5 downto 0);
27                         enable: in std_logic;
28                         rst: in std_logic;
29                         addr: in std_logic_vector(5 downto 0);
30                         clk: in std_logic;
31                         value: out std_logic_vector(5 downto 0);
32                         full: out std_logic;
33                         empty: out std_logic;
34                         num: out std_logic_vector(5 downto 0)
35                 );
36         end component;
37         component mod13
38                 port (
39                         x: in std_logic_vector(5 downto 0);
40                         modulo: out std_logic_vector(5 downto 0);
41                         floor: out std_logic_vector(5 downto 0)
```

```
42                 );
43         end component;
44         component g07_7_segment_decoder
45                 port (
46                         code: in std_logic_vector(3 downto 0);
47                         mode: in std_logic;
48                         segments_out: out std_logic_vector(6 downto 0)
49                 );
50         end component;
51
52         SIGNAL button: std_logic;
53         SIGNAL pulse: std_logic;
54         SIGNAL value: std_logic_vector(5 downto 0);
55         SIGNAL modulo, floor: std_logic_vector(5 downto 0);
56
57 begin
58         button <= not b;
59         gen: g07_debounder port map (button => button, clk => clk, pulse => pulse);
60         stack: g07_stack
61                 port map (
62                         mode => mode,
63                         enable => pulse,
64                         data => "000000",
65                         rst => rst,
66                         clk => clk,
67                         addr => addr,
68                         value => value);
69         modder: mod13 port map (x => value, floor => floor, modulo => modulo);
70         d1: g07_7_segment_decoder port map (code => modulo(3 downto 0), mode => '0'
               , segments_out => led1);
71         d2: g07_7_segment_decoder port map (code => floor(3 downto 0), mode => '0',
                segments_out => led2);
72 end behavior;
```

Appendix C

Schematic of the `g07_debounder`



Part V

Works Cited

Bibliography

[1] J. O'Connor, "This little piggy's on the market!," *timesunion*, 11 2011. http://blog.timesunion.com/highschool/this-little-piggys-on-the-market/25046.

[2] D. A. Lowther, *Computer Engineering*. McGill University, 2016.