EE 271 Final Project - Towers of Hanoi

Siyu Jian, Michael Molina, Kendrick Tang

December 6, 2012

Abstract

In this project, we designed and implemented the game of tower of Hanoi on the DE1 Board. The lab specification asked for a three ring game, but we went beyond that and gave the user the ability to select the number of rings they played with. Our implementation of the design was successful: it utilizes several databusses to move the right ring to the right tower if user selects a valid pair of towers, and waits for a valid pair otherwise. One difficulty we came across was implementing the GAL chip. The display unit had more input and output ports than the GAL chip could implement, so we were limited to using the GAL chip as a module to count the number of seconds it took for the user to finish the puzzle. Unfortunately, we were unable to finish this extra feature due to time constraints. Regardless, our game still runs very effectively.

We all worked together. We checked each other's work, helpe debug, code, design, and test our product, but we couldn't have done it without Kendrick's amazing design and debugging skills. By signing below, we all agree to this statement to some degree.

Signature _____

Signature _____

Signature Kendrick Tang (digitally signed December 6, 2012)

Contents

1 Introduction

The purpose of this lab was to reflect on the knowledge accumulated from the rest of the quarter and to design from start to finish our own project. We were able to experience firsthand all phases of a project from design, debugging, redesigning, debugging, implementing, integrating and testing, and debugging. Furthermore, due to the unrestricted nature of the guidelines, our design had no template to help us get started and was limited only by our imagination and creativity. In other words, the design was completely up to us. How robust our game was, was also completely up to us. We were in charge of determining test vectors, boundary cases, user errors, backup systems and error prevention all on our own, which tested our ability to find and balance all possible cases of error with practicality. We implemented our design of the Towers of Hanoi game on the DE1 board. We used almost every single output port from the breadboard, but also all the switches, buttons and LEDs for user input and debugging. With access to the 18 LEDs on the board itself, we were able to observe and verify the condition of our state machines, as well as values of different input and output vectors simultaneously.

2 Design Specification

In general, this lab only specified us to design a functional game on an FPGA board. We chose to create the Towers of Hanoi game. This game starts with a number of rings stacked on one peg, where each ring is smaller than the one below it. From there, rings are moved between three pegs, where a ring can only be placed on a ring that is larger than it. The winning condition is when the user has successfully transferred all the rings on the first tower to the third tower. Nobody can play the game without being able to see the state of the rings and towers, so a visualizer or display unit was essential to this design. Additional specifications were to use a databus to transfer information between registers, and implement a GAL chip as an external module. We decided to implement a game timer on the GAL chip as an additional feature to the game.

3 Design Procedure

3.1 On the FPGA

The design boiled down to capturing a way to encode the tower information, and a way to manipulate the information. We needed to not only encode how many rings were in each tower, but also the size of each ring in the tower. Furthermore, we needed to use this information to restrict the possible moves the user could do. We decided to use a seven bit register to represent each tower, where the least significant bit represented whether the largest ring was present, and the most significant bit represented whether the smallest ring was present. This way, a pair of choices is valid only if the value held in the register that represents the first choice is greater than the value held in the register that represents the

second choice. To manipulate the information, we used combinational logic to encode the value of a tower register in a one-hot binary value where the active bit represented the topmost ring. From here, all that needed to be done was to subtract the one-hot value from the first choice, and add it to the second choice. We utilized two modules to implement this design:

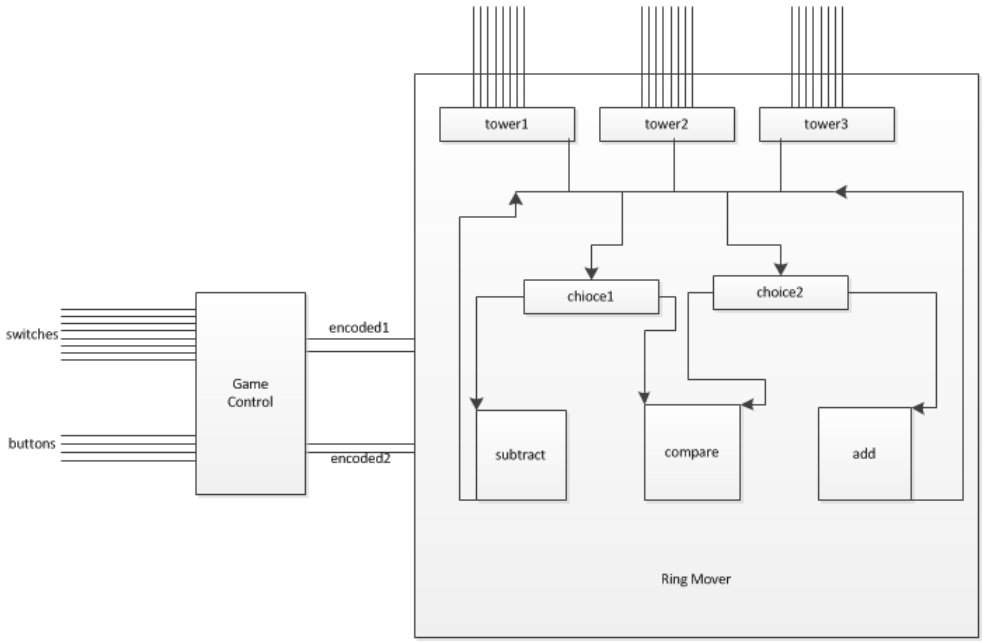

Figure 3.1: High level design for the entire system.

Interacting with the user is the game control module. It takes in all user inputs: switches for selecting the number of rings in the game, and buttons for selecting towers. The game control is responsible for initializing the system based on the number of rings selected by the user. If the user selects multiple initial rings, then the machine goes into an error state and waits for a reset. It also encoded the two choices the user makes each round as a four bit binary number, which is then inputted into the next module. Below is the state table and diagram for the game control:

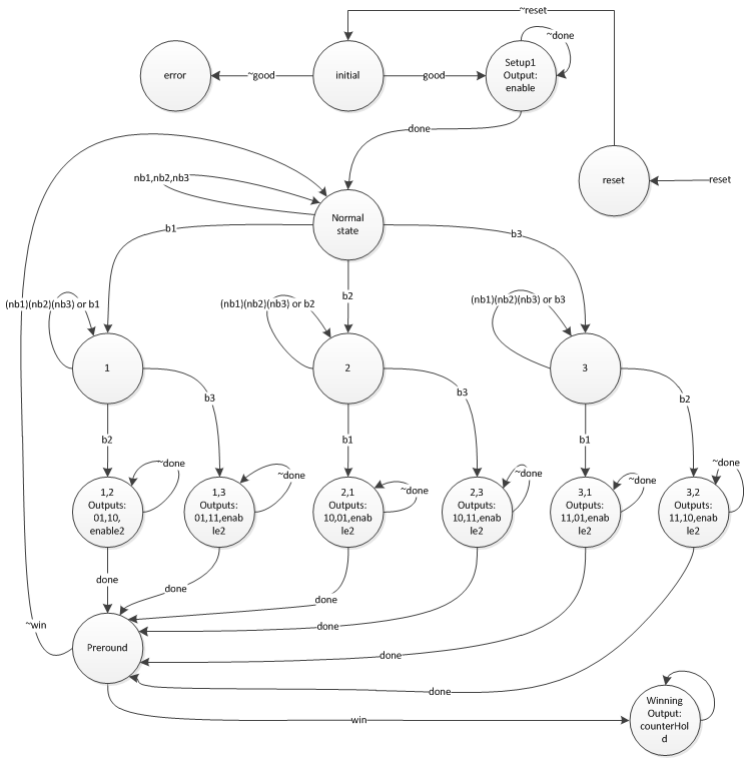

Figure 3.2: State diagram for game control.

State Name	state #	d3,d2,d1,d0 binary assignment	(1,x,x,x,x,x,x,x)	(0,x,x,x,x,x,x,x)	(0,good,x,x,x,x,x,x)	(0,ngood,x,x,x,x,x,x)
reset	0	0000	reset	initial	x	x
initial	1	0001	reset	x	setup 1	error
error	2	0010	reset	error	error	error
setup 1	3	0011	reset	x	x	x
normal state	4	0100	reset	x	x	x
1	5	0101	reset	x	x	x
2	6	0110	reset	x	x	x
3	7	0111	reset	x	x	x
1,2	8	1000	reset	x	x	x
1,3	9	1001	reset	x	x	x
2,1	10	1010	reset	x	x	x
2,3	11	1011	reset	x	x	x
3,1	12	1100	reset	x	x	x
3,2	13	1101	reset	x	x	x
preround state	14	1110	reset	x	x	x
winner state	15	1111	reset	winner state	winner state	winner state

Figure 3.3: State table for game control.

NEXT STATE (reset, good, b1, b2, b3, b4, done, win)						
(0,x,x,x,x,x,1,x)	(0,x,x,x,x,x,0,x)	(0,x,1,x,x,x,x,x)	(0,x,x,1,x,x,x,x)	(0,x,x,x,1,x,x,x)	(0,x,x,x,x,1,x,x)	(0,x,0,0,0,x,x,x)
x	x	x	x	x	x	x
x	x	x	x	x	x	x
error	error	error	error	error	error	error
normal state	setup1	x	x	x	x	x
x	x	1	2	3	x	normal state
x	x	1	1,2	1,3	normal state	1
x	x	2,1	2	2,3	normal state	2
x	x	3,1	3,2	3	normal state	3
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
winner state	winner state	winner state	winner state	winner state	winner state	winner state

Figure 3.4: State table for game control.

(0,x,x,x,x,x,1,x)	(0,x,x,x,x,x,0,x)	(0,x,x,x,x,x,x,1)	(0,x,x,x,x,x,x,0)	Outputs: (enable, ch1_1, ch2_1, ch1_2, ch2_2, counterHold)
x	x	x	x	(x,x,x,x,x,x)
x	x	x	x	(x,x,x,x,x,x)
error	error	error	error	(x,x,x,x,x,x)
x	x	x	x	(1,x,x,x,x,x)
x	x	x	x	(x,x,x,x,x,x)
x	x	x	x	(x,x,x,x,x,x)
x	x	x	x	(x,x,x,x,x,x)
x	x	x	x	(x,x,x,x,x,x)
preround state	1,2	x	x	(1,0,1,1,0,x)
preround state	1,3	x	x	(1,0,1,1,1,x)
preround state	2,1	x	x	(1,1,0,0,1,x)
preround state	2,3	x	x	(1,1,0,1,1,x)
preround state	3,1	x	x	(1,1,1,0,1,x)
preround state	3,2	x	x	(1,1,1,1,0,x)
x	x	winner state	normal state	(x,x,x,x,x,x)
winner state	winner state	winner state	winner state	(x,x,x,x,x,1)

Figure 3.5: State table for game control.

Responsible for recording and changing the state of the towers and rings is the ring mover module. It decodes the four bit input from the game control to determine which towers to compare and update. In the initialization phase, it just loads the first tower register with the right number of rings, and empties the other two tower registers. From there it compares the first and second tower choices made by the user to check if the move is valid. If the move is invalid then the user has to make another pair of choices. If the move is valid then the module moves the rings and updates the tower registers. Below is the state table and diagram for the ring mover:

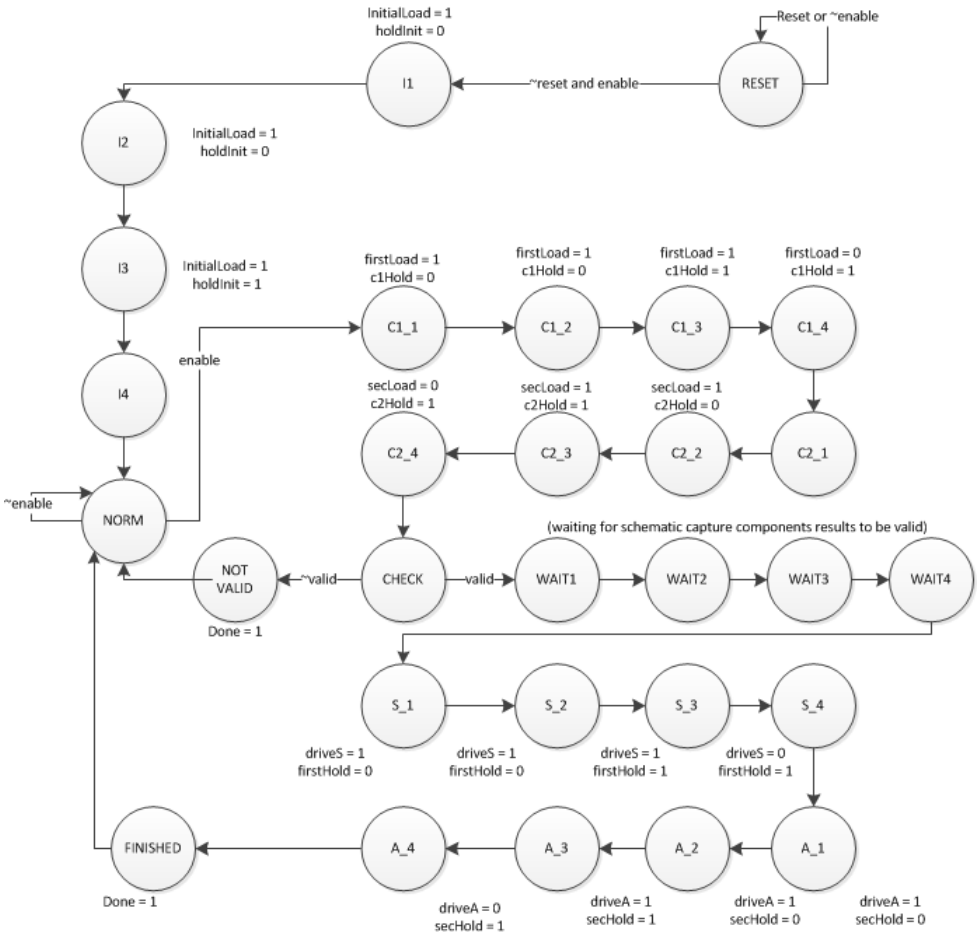

Figure 3.6: Top view of Quartus Pin Planner.

State Name	state #	binary assignment	NEXT STATE (reset, enable,valid) (1,x)	(0,1,x)	(0,0,x)	(0,x,1)	(0,x,0)	Outputs: (done)
reset	0	00000	reset	initial1	initial1	initial1	initial1	0
initial1	1	00001	reset	initial2	initial2	initial2	initial2	0
initial2	2	00010	reset	initial3	initial3	initial3	initial3	0
initial3	3	00011	reset	initial4	initial4	initial4	initial4	1
initial4	4	00100	reset	normal	normal	normal	normal	1
normal	5	00101	reset	loadc1 1	normal	x	x	0
loadc1 1	6	00110	reset	loadc1 2	loadc1 3	loadc1 4	loadc1 5	0
loadc1 2	7	00111	reset	load c1 3	load c1 3	load c1 3	load c1 3	0
loadc1 3	8	01000	reset	load c1 4	load c1 4	load c1 4	load c1 4	0
loadc1 4	9	01001	reset	load c2 1	load c2 1	load c2 1	load c2 1	0
loadc2 1	10	01010	reset	load c2 2	load c2 2	load c2 2	load c2 2	0
loadc2 2	11	01011	reset	load c2 3	load c2 3	load c2 3	load c2 3	0
loadc2 3	12	01100	reset	load c2 4	load c2 4	load c2 4	load c2 4	0
loadc2 4	13	01101	reset	x	x	load a2 s2 1	sendDone	0
wait1	14	01110	reset	wait2	wait2	wait2	wait2	0
wait2	15	01111	reset	wait3	wait3	wait3	wait3	0
wait3	16	10000	reset	wait4	wait4	wait4	wait4	0
wait4	17	10001	reset	load new1 1	load new1 1	load new1 1	load new1 1	0
load new1 1	18	10010	reset	load new1 2	load new1 2	load new1 2	load new1 2	0
load new1 2	19	10011	reset	load new1 3	load new1 3	load new1 3	load new1 3	0
load new1 3	20	10100	reset	load new1 4	load new1 4	load new1 4	load new1 4	0
load new1 4	21	10101	reset	load new2 1	load new2 1	load new2 1	load new2 1	0
load new2 1	22	10110	reset	load new2 2	load new2 2	load new2 2	load new2 2	0
load new2 2	23	10111	reset	load new2 3	load new2 3	load new2 3	load new2 3	0
load new2 3	24	11000	reset	load new2 4	load new2 4	load new2 4	load new2 4	0
load new2 4	25	11001	reset	finished	finished	finished	finished	0
finished	26	11010	reset	normal	normal	normal	normal	1
sendDone	27	11011	reset	normal	normal	normal	normal	0

Figure 3.7: State table for game control.

There were potential problems related to the clock running these state machines. If the clock was too fast, the user inputs would not be readable due to bouncing and clocking speeds. To correct this, we slowed down the clock to a point where the user wasnt hindered to wait for the state machine, yet able to easily input their tower choices without any issue. To achieve this we used a 92 Hz clock, derived from the DE1 boards 24 MHz internal clock.

3.2 On the GAL Chip

We attempted to implement a game counter on the GAL chip. The top level design looked like this:

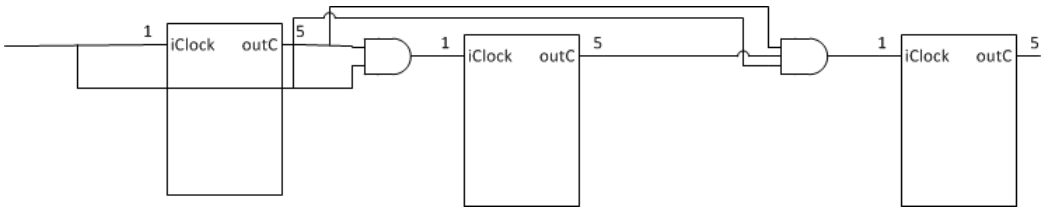

Figure 3.8: High level design for timer module.

The idea behind the design was to simplify a general counting problem to just counting to ten. If we had a state machine that counted to ten, then that could represent the ones digit. If the machine output a binary 1 in the ninth state, then at the next clock cycle the tens digit would be clocked. Similarly, the hundreds digit would clock after both the ones and tens digit was in the ninth state. We were unable to implement this due to time constraints.

4 Hardware Implementation

Our virtual Towers of Hanoi game was implemented using one GAL 22V10 chip, the DE1 board , and three 10-segment bar graph arrays with 21 $1k\Omega$-resistors. The GAL chip contained a simple counter, which was decoded and outputted to the HEX displays on the DE1 board. This acted as a timer so people could compare different completion times. The GAL chip ran off of a clock, and had a reset from the DE1 board, and output four bits which encoded how much time had passed, modulo 10. Below is the chip diagram:

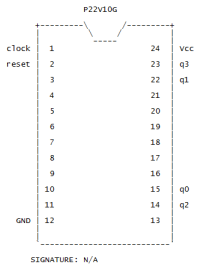

Figure 4.1: This is the chip report for the GAL chip.

The DE1 board holds all of our combinational and sequential logic. It acts as a controller taking user inputs to determine the next state of the system. We used 16 inputs:

- A switch as a reset control
- Seven other switched behaved as selectors used to determine how many rings the game would be initialized with
- Three active low buttons were used to select which towers to change
- One clock to run the system
- Four bits of data from the GAL chip.

As a result of several state machines, combinational logic and schematic capture components from Quartus, our module produced 53 outputs, some as external displays, others back into the DE1 board as displays and some controls for the GAL chip:

- Five LEDs to represent the state of the ring mover module
- Four LEDs to represent the state of the game controller
- Three sets of seven bit outputs for each LED display to represent the state of the towers
- Three sets of seven bit outputs for each HEX display to represent the time
- A slowed clock to run the timer
- A reset for the timer

Below is a diagram of the pin assignment below:

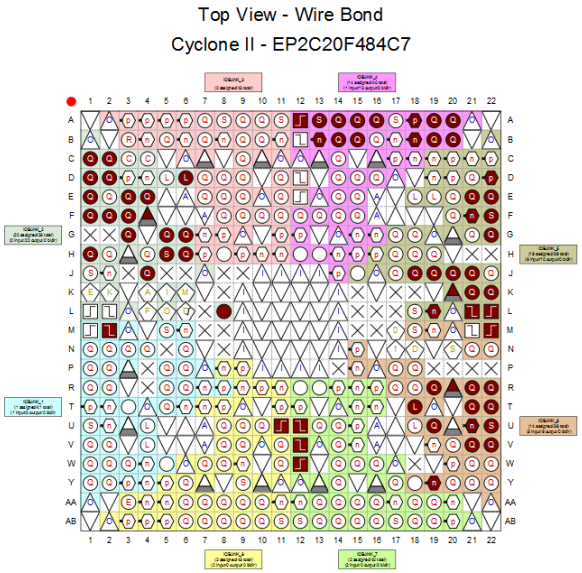

Figure 4.2: Top view of Quartus Pin Planner.

b1	Input	PIN_T21	6	B6_N0	PIN_T21	3.3-V LV...default)	24mA (default)
b2	Input	PIN_T22	6	B6_N0	PIN_T22	3.3-V LV...default)	24mA (default)
b3	Input	PIN_R21	6	B6_N0	PIN_R21	3.3-V LV...default)	24mA (default)
b4	Input	PIN_R22	6	B6_N0	PIN_R22	3.3-V LV...default)	24mA (default)
clock	Input	PIN_A12	4	B4_N1	PIN_A12	3.3-V LV...default)	24mA (default)
dispOut1[6]	Output	PIN_A13	4	B4_N1	PIN_A13	3.3-V LV...default)	24mA (default)
dispOut1[5]	Output	PIN_B13	4	B4_N1	PIN_B13	3.3-V LV...default)	24mA (default)
dispOut1[4]	Output	PIN_A14	4	B4_N1	PIN_A14	3.3-V LV...default)	24mA (default)
dispOut1[3]	Output	PIN_B14	4	B4_N1	PIN_B14	3.3-V LV...default)	24mA (default)
dispOut1[2]	Output	PIN_A15	4	B4_N1	PIN_A15	3.3-V LV...default)	24mA (default)
dispOut1[1]	Output	PIN_B15	4	B4_N1	PIN_B15	3.3-V LV...default)	24mA (default)
dispOut1[0]	Output	PIN_A16	4	B4_N1	PIN_A16	3.3-V LV...default)	24mA (default)
dispOut2[6]	Output	PIN_A18	4	B4_N0	PIN_A18	3.3-V LV...default)	24mA (default)
dispOut2[5]	Output	PIN_B18	4	B4_N0	PIN_B18	3.3-V LV...default)	24mA (default)
dispOut2[4]	Output	PIN_A19	4	B4_N0	PIN_A19	3.3-V LV...default)	24mA (default)
dispOut2[3]	Output	PIN_B19	4	B4_N0	PIN_B19	3.3-V LV...default)	24mA (default)
dispOut2[2]	Output	PIN_A20	4	B4_N0	PIN_A20	3.3-V LV...default)	24mA (default)
dispOut2[1]	Output	PIN_B20	4	B4_N0	PIN_B20	3.3-V LV...default)	24mA (default)
dispOut2[0]	Output	PIN_E21	5	B5_N0	PIN_E21	3.3-V LV...default)	24mA (default)
dispOut3[6]	Output	PIN_K21	5	B5_N1	PIN_K21	3.3-V LV...default)	24mA (default)
dispOut3[5]	Output	PIN_K22	5	B5_N1	PIN_K22	3.3-V LV...default)	24mA (default)
dispOut3[4]	Output	PIN_J19	5	B5_N1	PIN_J19	3.3-V LV...default)	24mA (default)
dispOut3[3]	Output	PIN_J20	5	B5_N1	PIN_J20	3.3-V LV...default)	24mA (default)
dispOut3[2]	Output	PIN_J18	5	B5_N1	PIN_J18	3.3-V LV...default)	24mA (default)
dispOut3[1]	Output	PIN_K20	5	B5_N1	PIN_K20	3.3-V LV...default)	24mA (default)
dispOut3[0]	Output	PIN_L19	5	B5_N1	PIN_L19	3.3-V LV...default)	24mA (default)
gTClock	Output	PIN_D22	5	B5_N0	PIN_D22	3.3-V LV...default)	24mA (default)
q0	Output	PIN_R20	6	B6_N0	PIN_R20	3.3-V LV...default)	24mA (default)
q0A	Output	PIN_U22	6	B6_N1	PIN_U22	3.3-V LV...default)	24mA (default)
q0_chip	Input	PIN_J21	5	B5_N1	PIN_J21	3.3-V LV...default)	24mA (default)
q1	Output	PIN_R19	6	B6_N0	PIN_R19	3.3-V LV...default)	24mA (default)
q1A	Output	PIN_U21	6	B6_N1	PIN_U21	3.3-V LV...default)	24mA (default)
q1_chip	Input	PIN_G22	5	B5_N1	PIN_G22	3.3-V LV...default)	24mA (default)
q2	Output	PIN_U19	6	B6_N1	PIN_U19	3.3-V LV...default)	24mA (default)
q2A	Output	PIN_V22	6	B6_N1	PIN_V22	3.3-V LV...default)	24mA (default)
q2_chip	Input	PIN_F22	5	B5_N0	PIN_F22	3.3-V LV...default)	24mA (default)
q3	Output	PIN_Y19	6	B6_N1	PIN_Y19	3.3-V LV...default)	24mA (default)
q3A	Output	PIN_V21	6	B6_N1	PIN_V21	3.3-V LV...default)	24mA (default)
q3_chip	Input	PIN_F21	5	B5_N0	PIN_F21	3.3-V LV...default)	24mA (default)
q4	Output	PIN_T18	6	B6_N1	PIN_T18	3.3-V LV...default)	24mA (default)
r1	Input	PIN_L21	5	B5_N1	PIN_L21	3.3-V LV...default)	24mA (default)
r2	Input	PIN_M22	6	B6_N0	PIN_M22	3.3-V LV...default)	24mA (default)
r3	Input	PIN_V12	7	B7_N1	PIN_V12	3.3-V LV...default)	24mA (default)
r4	Input	PIN_W12	7	B7_N1	PIN_W12	3.3-V LV...default)	24mA (default)
r5	Input	PIN_U12	8	B8_N0	PIN_U12	3.3-V LV...default)	24mA (default)
r6	Input	PIN_U11	8	B8_N0	PIN_U11	3.3-V LV...default)	24mA (default)
r7	Input	PIN_M2	1	B1_N0	PIN_M2	3.3-V LV...default)	24mA (default)
reset	Input	PIN_L22	5	B5_N1	PIN_L22	3.3-V LV...default)	24mA (default)
resetOut	Output	PIN_E22	5	B5_N0	PIN_E22	3.3-V LV...default)	24mA (default)

Figure 4.3: List of pin assignments

The three 10-Segement Bar Graph Arrays were used to visually represent the seven bit tower registers. Each anode pin was connected serially to a 1kΩ resistor, and each cathode pin was grounded. The diagram of the component is below:

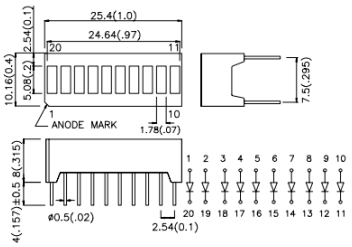

Figure 4.4: Top, side and front view of 10-segment LED array with diodes and pin assignments labeled.

5 Test Plan

5.1 Game Controller Module

Before we implement anything onto the DE1 board, we need to ensure that our codes first works in Bughunter, and then in the Quartus environment. However, we found out that we could not verify the schematics modules from the Quartus so our test plan is to first test the game control and ring mover modules.

In the game control module, we need to test the following conditions:

- If the user selects multiple choices for the number of rings they want, then the state machine goes to the error state.
- If user selects only one ring switch, then the state machine moves to the next state.
- At a waiting state, the module waits for a done signal and only moves on to the next state when the done signal arrives.
- If the user presses one button then the machine should stay in the same state if the user continues to press the same button. This is to prevent bouncing errors.
- If the user presses one button, and then presses a different one, then the machine should output enable and wait for a done signal.

5.2 Ring Mover Module

The ring mover called instances of Quartus built in comparator, adder and subtracter, which restricted our ability to model the device in Bughunter. However, we were still able to model the initialization of the rings and the databus. We tested all different possible initializations, which at this point are actually limited to only one active input since the game control module wont begin initialization if there are multiple active inputs. We also exhaustively tested moving information from the tower registers into the choice registers via the databus. Our databus test was only one directional because of the symmetry of the databus.

Beyond this point, testing the ring mover in Bughunter is impossible because it wont compile Quartus schematic capture components. To test it, we used the 18 LEDs available on the FPGA board. To check the state machine, we assigned the outputs of all the flip flops to the LEDs, slowed down the clock and observed which states were happening. Similarly, we assigned outputs of the comparator, adder and subtracter to the LEDs to check those values as well. Obviously there arent enough LEDs to test these all at the same time so wed have to different sets of assignments at a time.

The comparator, adder and subtracter were tested in exactly the same way. Here we will explain the test procedure for the comparator because values are compared before they are added and subtracted so without knowing the functionality of the comparator, we cant efficiently test the other two parts. The procedures for the other two parts can be obtained by simply replacing comparator with adder or subtracter. To test the comparator we only needed to check the simplest case, due to the symmetry of our design. We designed our game to work with any given number of rings, and because of this our design had to be symmetric. Also, at this point we have already verified that the databus is working properly because that was tested in Bughunter. This way we knew if we could get the comparator to work once, it would always work. We used the LEDs to display the two seven bit inputs, and the one seven bit output of the comparator.

The comparator inputs just followed the selected towers, which is an easy visual check. The output of the comparator is a one-hot binary representation of the smallest ring on the tower. For example, if the smallest ring was the third smallest possible ring, the output would be 001000. Essentially, it should zero out all terms except the smallest present ring.

The adder inputs just follow the second selected tower and the comparator. The output should be those two added together, and since the comparator output was essentially just one ring, the result should be the second selected tower, with one new ring.

Similarly, the subtracter inputs follow the first selected tower and the comparator. The output should be the first selected tower minus the result from the comparator. The output should just be the first selected tower, with the first ring bit zeroed out.

6 Results

6.1 Results of Game Controller

Below is an example of one set of tests completed and passed by our game controller module. The test was performed in Bughunter:

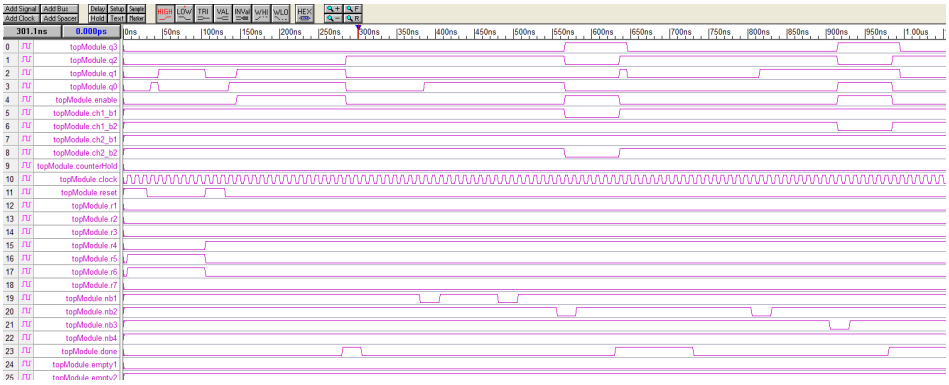

Figure 6.1: Timing diagram of tests on Game Controller Module.

First we tested the error state. We input into the system multiple active ring switches. In this case switch five and six were both active. As you can see, the system goes and stays in the error state until reset is hit. From here, we only activate switch four, and as we expect, the machine runs until it activates enable and waits for a done signal. From here, we test the effects of bouncing. We press button one multiple times and observe that the state remains unchanged. We then press a different button and observe that it waits for a done signal and returns to the normal state. From here, we can pick another pair of towers to modify and check that the four bit code outputted by the game control is correct. In this case, we pressed buttons two and three, so we expect the output to be 1011. We exhaustively checked that the other initializations worked, and that the four bit codes were always what we expected.

6.2 Results of Ring Mover

Below is the timing diagram testing the first half of the ring mover module because some parts of ring mover use schematic capture Verilog code from Quartus which does not compile in Bughunter.

First we tested staying in the reset state. We activated switch four and sent reset equal to zero. As you can see in the timing diagram, the system stays in the same state as we expected. Then the test activates enable to move on to the next state where we start loading the binary value 1 for the first four least significant bits of the tower1s register from the Bus giving a value of 0F in hexadecimal. Then we go back to normal state and turn on switch three only

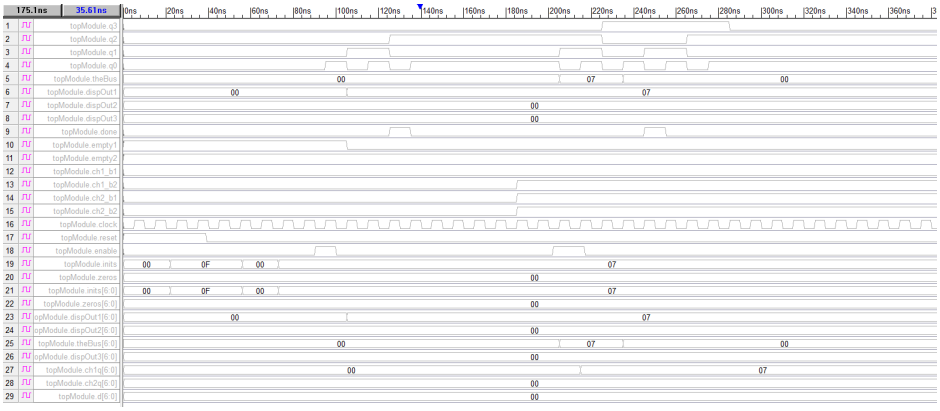

Figure 6.2: Timing diagram of tests on Ring Mover Module.

to test what happens when the choice bit is 0111 is inputted. As seen in the diagram the bus value gets the initial values which is 07 in hexadecimal giving the three less significant bits in the first tower a binary value of 1 which is what we expected. Testing different choices of towers and initial rings in tower 1 we find that the test results are what we expected. However, this only confirms that the data bus is working. To test the comparator, adder and subtracter we had to implement the design through Quartus.

We first tested the comparator because it outputs to the adder and subtracter. The comparator inputs should follow the tower registers that were picked, and by assigning those values to LEDs, we could compare them to the tower's LED display and confirm that it wasn't working. After some debugging and fixing typos the comparator was following the towers, and was outputting the one-hot code.

Then we tested the adder and subtractor, in the same exact way we tested the comparator. The adder should produce what the second choice should look like after the ring has been added, and the subtractor shoud produce what the first choice should look like after the ring has been removed. After a long series of debugging and fixing, we got the adder and subtractor working.

6.3 Results of Tower of Hanoi

After testing each part individually, we integrated the system together using the Quartus environment and just played the game exhaustively.

7 Error Analysis

One error in this project is our extra feature. The game counter does not count the time in order. For example, in first ten seconds, it counts as $0, 1, 2, 3, 4, 7, 9$

instead of supposedly $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ and in the following, it counts as $11, 12, 13, 14, 17, 18, 29$, which is not in order. We think maybe the problem has to do with some simple mistakes, like the logic equations for the D-flip flops are wrong, or a design problem. We think maybe for the tenth digit and hundredth digit on the hex display, we need to clock in with an AND gate of the combination of input clock and the output from the unit digit module. For the hundredth digit, we need to use AND gate for the input clock, tenth clock, and hundredth clock. Since we do not have these and gates to control the clock for each module, maybe the time counter will be out of order.

8 Reasons for Incomplete Project

We did not manage to finish the extra feature, game counter due to the time constraints. We tried to fix it by negating the data from the GAL chip since first we observed the negated hex display. However, after doing so, we noticed that it skips some digit, like 5 and 9. Then, we tried to add the AND gate to each clock with the data from the GAL chip. However, it still does not help. We ended up running out of time and were unable to complete the game counter. However, we still incorporated the GAL chip in our design.

9 Summary

Out of all the game projects, we chose to design and implement towers of Hanoi. We planned to design a seven ring game, meaning that user could choose the number of rings from one to seven. Also, we decided to use LED display to visualize the rings to the users. As an extra feature, we put GAL chip onto the DE1 board and make it a counter. Under our test plan, it splits into both the software and hardware sides. On the software side, we need to test the game control and ring mover on the bughunter pro. Then, we test all of the codes under the Quartus. Next, we assign the pins in a way that we could determine which states both the game control and ring mover are in. We managed to catch several hundred errors when implementing our test plan. However, we did not have time to fix the game counter as an extra feature, implemented by the GAL chip.

10 Conclusion

For this lab we designed and implemented a virtual game of Tower of Hanoi on the DE1 board using essentially two modules containing both sequential and combinational logic. We also attempted to implement a counter on the GAL chip in order to display a game timer on the hexadecimal display. Using Verilog we tested the code separately and then put it all together in one big top module and programmed it into the DE1 board with Quartus. After exhaustively testing different cases for our game functions we found out our system does exactly what it expected to. One error that we encountered was on the timer because

it would not count correctly and due to time constraints we could not debug and correct Verilog code. This did not affect the functionality of our game just inconvenient for serious competitive gamers.

11 Appendix

```
module topModule(resetOut,gTClock,q3A,q2A,q1A,q0A,q3,q2,q1,q0,dispOut1, dispOut2, dispOut3, q3_chip, q2_chip, q1_chip, q0_chip,r1,r2,r3,r4,r5,r6,r7,b1,b2,b3,b4,clock,

  output [6:0] dispOut1, dispOut2, dispOut3;
  output q3A,q2A,q1A,q0A,q4,q3,q2,q1,q0,resetOut, gTClock;

  input clock;
  input q3_chip, q2_chip, q1_chip, q0_chip, r1,r3,r4,r5,r6,r7,b1,b2,r2,b3,b4,reset;

  buf buf1(resetOut, reset);
  //slows down the 24 MHz clock for the game counter
  gameTimerClock gTimerClock(gTClock, clock,reset);

  playingClock playingClock(pClock, clock,reset);

  //controls the game
  gameControl gameControl(q3A,q2A,q1A,q0A,enable,ch1_b1,ch1_b2,ch2_b1,ch2_b2,counterHold,pClock,reset,r1,r2,r3,r4,r5,r6,r7,b1,b2,b3,b4,done,empty1,empty2);

  //contains ring/tower information and moves rings around;
  ringMover theRinger(outputAdd,cOut, ch1q, ch2q,outputSub,theBus,q4,q3,q2,q1,q0,dispOut1,dispOut2,dispOut3 ,done,empty1,empty2, r1,r2,r3,r4,r5,r6,r7, ch1_b1,ch1_b2, ch2_

   //displays time
   timeDisplay timeDisplay(hex0_0,hex0_1,hex0_2,hex0_3,hex0_4,hex0_5,hex0_6,
                           hex1_0,hex1_1,hex1_2,hex1_3,hex1_4,hex1_5,hex1_6,
                          hex2_0,hex2_1,hex2_2,hex2_3,hex2_4,hex2_5,hex2_6,
                          q3_chip, q2_chip, q1_chip, q0_chip, reset);
endmodule
```

Figure 11.1: Verilog code for Top Module.

```
odule gameControl(G_0,G_5,G_4,G_3,G_2,O_0,O_1,O_2,O_3,O_4,O_5,A_0,A_1,A_2,A_4,A_5,A_6,L_5,L_4,L_3,q3,q2,q1,q0,enable,ch1_b1,ch1_b2,ch2_b1,ch2_b2,counterHold,clock,reset,r1,r2,r3,r4,r5,r6,r7,

    output q3,q2,q1,q0,enable,ch1_b1,ch1_b2,ch2_b1,ch2_b2,counterHold;
    input clock,reset,r1,r2,r3,r4,r5,r6,r7,nb1,nb2,nb3,nb4,done,empty1,empty2;

    output G_0,G_5,G_4,G_3,G_2,O_0,O_1,O_2,O_3,O_4,O_5,A_0,A_1,A_2,A_4,A_5,A_6,L_5,L_4,L_3;

    and and9999(win, empty1,empty2);

'/Sequential logic

    //DFF_awesome
    DFF_awesome df0(q0, nq0, d0, clock, reset),
                   df1(q1, nq1, d1, clock, reset),
                   df2(q2, nq2, d2, clock, reset),
                   df3(q3, nq3, d3, clock, reset);

    //d0
    and and12(d0_0,nq3,nq2,nq1,nq0,nReset),
        and13(d0_1,nq3,nq2,nq1,q0,nReset,good),
        and14(d0_2,nq3,nq2,q1,q0,nReset,nDone),
      and15(d0_3,nq3,q2,nq1,nq0,nReset,b1),
      and16(d0_4,nq3,q2,nq1,nq0,nReset,b3),
      and17(d0_5,nq3,q2,nq1,q0,nReset,b1),
      and18(d0_6,nq3,q2,nq1,q0,nReset,nb1,nb2,nb3),
        and19(d0_7,nq3,q2,q1,q0,nReset,b3),
        and20(d0_8,nq3,q2,q1,q0,nReset,nb1,nb2,nb3),
      and21(d0_9,nq3,q2,nq1,q0,nReset,b3),
        and22(d0_10,nq3,q2,q1,nq0,nReset,b3),//m
        and23(d0_11,nq3,q2,q1,q0,nReset,b2),
        and24(d0_12,q3,nq2,nq1,q0,nReset,nDone),
        and25(d0_13,q3,nq2,q1,q0,nReset,nDone),
        and26(d0_14,q3,q2,nq1,q0,nReset,nDone),
        and27(d0_15,q3,q2,q1,nq0,nReset,win),
        and28(d0_16,q3,q2,q1,q0,nReset);

    or or3(d0,d0_0,d0_1,d0_2,d0_3,d0_4,d0_5,d0_6,d0_7,d0_8,d0_9,d0_10,d0_11,d0_12,d0_13,d0_14,d0_15,d0_16);


    //d1
    and and29(d1_0,nq3,nq2,nq1,q0,nReset,ngood),
        and30(d1_1,nq3,nq2,q1,nq0,nReset),
        and31(d1_2,nq3,nq2,nq1,q0,nReset,good),
        and32(d1_3,nq3,nq2,q1,q0,nReset,nDone),
        and33(d1_4,nq3,q2,nq1,nq0,nReset,b2),
        and34(d1_5,nq3,q2,q1,nq0,nReset,b2), //stay
        and35(d1_6,nq3,q2,q1,nq0,nReset,nb1,nb2,nb3),//stay
      and36(d1_7,nq3,q2,nq1,nq0,nReset,b3),
        and37(d1_8,nq3,q2,q1,q0,nReset,b3),  //stay
        and38(d1_9,nq3,q2,q1,q0,nReset,nb1,nb2,nb3),  //stay
        and39(d1_10,nq3,q2,q1,nq0,nReset,b1),
```

Figure 11.2: Verilog code for Game Control pt. 1

```
    and74(d2_23,q3,nq2,q1,nq0,nReset,done),
    and75(d2_24,q3,nq2,q1,q0,nReset,done),
    and76(d2_25,q3,q2,nq1,nq0,nReset,done),
    and77(d2_26,q3,q2,nq1,q0,nReset,done);

or or5(d2,d2_0,d2_1,d2_2,d2_3,d2_4,d2_5,d2_6,d2_7,d2_8,d2_9,d2_10,d2_11,d2_12,d2_13,d2_14,d2_15,d2_16,d2_17,d2_18,d2_19,d2_20,d2_21,d2_22,d2_23,d2_24,d2_25,d2_26);


//d3
and and78(d3_0,nq3,q2,nq1,q0,nReset,b2),
    and79(d3_1,q3,nq2,nq1,nq0,nReset,nDone),
    and80(d3_2,nq3,q2,nq1,q0,nReset,b3),
    and81(d3_3,q3,nq2,nq1,q0,nReset,nDone),
    and82(d3_4,nq3,q2,q1,nq0,nReset,b1),
    and83(d3_5,q3,nq2,q1,nq0,nReset,nDone),
    and84(d3_6,nq3,q2,q1,nq0,nReset,b3),
    and85(d3_7,q3,nq2,q1,q0,nReset,nDone),
    and86(d3_8,nq3,q2,q1,q0,nReset,b1),
    and87(d3_9,q3,q2,nq1,nq0,nReset,nDone),
    and88(d3_10,nq3,q2,q1,q0,nReset,b2),
    and89(d3_11,q3,q2,nq1,q0,nReset,nDone),
    and90(d3_12,q3,nq2,nq1,nq0,nReset,done),
    and91(d3_13,q3,nq2,nq1,q0,nReset,done),
    and92(d3_14,q3,nq2,q1,nq0,nReset,done),
    and93(d3_15,q3,nq2,q1,q0,nReset,done),
    and94(d3_16,q3,q2,nq1,nq0,nReset,done),
    and95(d3_17,q3,q2,nq1,q0,nReset,done),
    and96(d3_18,q3,q2,q1,nq0,nReset,win),
    and97(d3_19,q3,q2,q1,q0,nReset);
or or6(d3, d3_0,d3_1,d3_2,d3_3,d3_4,d3_5,d3_6,d3_7,d3_8,d3_9,d3_10,d3_11,d3_12,d3_13,d3_14,d3_15,d3_16,d3_17,d3_18,d3_19);


mbinational logic and outputs
not not0(nr1,r1),
    not1(nr2,r2),
    not2(nr3,r3),
    not3(nr4,r4),
    not4(nr5,r5),
    not5(nr6,r6),
    not6(nr7,r7),
    not7(nReset,reset),
    not11(nDone,done),
    not12(nWin,win),
  not13(ngood, good),
    not8(b1,nb1),
    not9(b2,nb2),
    not10(b3,nb3),
    notk11(b4,nb4);


choice 1 and 2 bits for ringMovingModule
```

Figure 11.3: Verilog code for Game Control pt. 2

```
module ringMover(outputAdd,oOut, ch1q, ch2q,output3ub,theBus,q4,q3,q2,q1,q0,dispOut1,dispOut2,dispOut3 ,done,empty1,empty2, sw1,sw2,sw3,sw4,sw5,sw6,sw7, ch1_b1,ch1_b2, ch2_b1,ch2_b2, clock, r

  output done, empty1, empty2,q4,q3,q2,q1,q0;
  output [6:0] dispOut1, dispOut2, dispOut3,output3ub, oOut, ch1q, ch2q,outputAdd;
  input sw1,sw2,sw3,sw4,sw5,sw6,sw7, ch1_b1,ch1_b2, ch2_b1,ch2_b2, clock, reset, enable;

  wire [6:0] spec, inits, zeros;
  output [6:0] theBus;

  //STATE MACHINE LOGIC!
  //d0
  and andk75(d0_0, nq4,nq3,nq2,nq1,nq0,enable,~reset),
      andk76(d0_1, nq4,nq3,nq2,q1,nq0,~reset),
      andk77(d0_2, nq4,nq3,q2,nq1,nq0,~reset),
      andk78(d0_3, nq4,nq3,q2,q1,nq0,~reset),
      andk79(d0_4, nq4,q3,nq2,nq1,nq0,~reset),
      andk80(d0_5, nq4,q3,nq2,q1,nq0,~reset),
      andk81(d0_6, nq4,q3,q2,nq1,nq0,~reset),
      andFIX2(d0_14, nq4,nq3,q2,nq1,q0,~enable,~reset),
      andmimo0(d0_7,nq4,q3,q2,q1,nq0,~reset),
      andmimo1(d0_8,q4,nq3,nq2,nq1,nq0,~reset),
      andmimo2(d0_9,q4,nq3,nq2,q1,nq0,~reset),
      andmimo3(d0_10,q4,nq3,q2,nq1,nq0,~reset),
      andmimo4(d0_11,q4,nq3,q2,q1,nq0,~reset),
      andmimo5(d0_12,q4,q3,nq2,nq1,nq0,~reset),
      andmimo6(d0_15,q4,q3,nq2,q1,nq0,~reset);
          and fixedDon123e(d0_13, nq4,q3,q2,nq1,q0,~valid,~reset);
            and fixedDone0(d0_16, q4,q3,nq2,q1,q0,~reset);


  or or001(d0, d0_0,d0_1,d0_2,d0_3,d0_4,d0_5,d0_6,d0_7,d0_8,d0_9,d0_10,d0_11,d0_12,d0_13,d0_14,d0_15,d0_16);


  //d1
  and ande75(d1_0, nq4,nq3,nq2,nq1,q0,~reset),
      ande76(d1_1, nq4,nq3,nq2,q1,nq0,~reset),
      ande77(d1_2, nq4,nq3,q2,nq1,q0,~reset,enable),
      ande78(d1_3, nq4,nq3,q2,q1,nq0,~reset),
      ande79(d1_4, nq4,q3,nq2,nq1,q0,~reset),
      ande80(d1_5, nq4,q3,nq2,q1,nq0,~reset),
      andmimo7(d1_6, nq4,q3,q2,nq1,q0,valid,~reset),
      andmimo8(d1_7, nq4,q3,q2,q1,nq0,~reset),
      andmimo9(d1_8, q4,nq3,nq2,nq1,q0,~reset),
      andmimo10(d1_9, q4,nq3,nq2,q1,nq0,~reset),
      andmimo11(d1_10, q4,nq3,q2,nq1,q0,~reset),
      andmimo12(d1_11, q4,nq3,q2,q1,nq0,~reset),
      andmimo13(d1_12, q4,q3,nq2,nq1,q0,~reset);
          and fixedDodne(d1_13, nq4,q3,q2,nq1,q0,~valid,~reset);

  or or101(d1, d1_0,d1_1,d1_2,d1_3,d1_4,d1_5,d1_6,d1_7,d1_8,d1_9,d1_10,d1_11,d1_12,d1_13);
```

Figure 11.4: Verilog code for Ring Mover pt. 1.

```
//d2
and andn75(d2_0, nq4,nq3,nq2,q1,q0,~reset),
    andn76(d2_1, nq4,nq3,q2,nq1,nq0,~reset),
    andn77(d2_2, nq4,nq3,q2,nq1,q0,~reset,enable),
    andn78(d2_3, nq4,nq3,q2,q1,nq0,~reset),
    andn79(d2_4, nq4,q3,nq2,q1,q0,~reset),
    andn80(d2_5, nq4,q3,q2,nq1,nq0,~reset),
    andnFIX2(d2_14, nq4,nq3,q2,nq1,q0,~enable,~reset),
    andmimo14(d2_7, nq4,q3,q2,nq1,q0,valid,~reset),
    andmimo15(d2_8, nq4,q3,q2,q1,nq0,~reset),
    andmimo16(d2_9, q4,nq3,nq2,q1,q0,~reset),
    andmimo17(d2_10, q4,nq3,q2,nq1,nq0,~reset),
    andmimo18(d2_11, q4,nq3,q2,nq1,q0,~reset),
    andmimo19(d2_12, q4,nq3,q2,q1,nq0,~reset),
    andmimo20(d2_13, q4,q3,nq2,q1,nq0,~reset);
      and fixedDonejk0(d2_15, q4,q3,nq2,q1,q0,~reset);

or or201(d2, d2_0,d2_1,d2_2,d2_3,d2_4,d2_5,d2_7,d2_8,d2_9,d2_10,d2_11,d2_12,d2_13,d2_14,d2_15);



//d3
and andd75(d3_0, nq4,nq3,q2,q1,q0,~reset),
    andd76(d3_1, nq4,q3,nq2,nq1,nq0,~reset),
    andd77(d3_2, nq4,q3,nq2,nq1,q0,~reset),
    andd78(d3_3, nq4,q3,nq2,q1,nq0,~reset),
    andd79(d3_4, nq4,q3,nq2,q1,q0,~reset),
    andd80(d3_5, nq4,q3,q2,nq1,nq0,~reset),
    andmimo21(d3_6, nq4,q3,q2,nq1,q0,valid),
    andmimo22(d3_7, nq4,q3,q2,q1,nq0,~reset),
    andmimo23(d3_8, q4,nq3,q2,q1,q0,~reset),
    andmimo24(d3_9, q4,q3,nq2,nq1,nq0,~reset),
    andmimo25(d3_10, q4,q3,nq2,nq1,q0,~reset);
      and fixedDone(d3_11, nq4,q3,q2,nq1,q0,~valid,~reset);

or or301(d3, d3_0,d3_1,d3_2,d3_3,d3_4,d3_5,d3_6,d3_7,d3_8,d3_9,d3_10,d3_11);

//d4
and    andmimo26(d4_0, nq4,q3,q2,q1,q0,~reset),
    andmimo27(d4_1, q4,nq3,nq2,nq1,nq0,~reset),
    andmimo28(d4_2, q4,nq3,nq2,nq1,q0,~reset),
    andmimo29(d4_3, q4,nq3,nq2,q1,nq0,~reset),
    andmimo30(d4_4, q4,nq3,nq2,q1,q0,~reset),
    andmimo31(d4_5, q4,nq3,q2,nq1,nq0,~reset),
    andmimo32(d4_6, q4,nq3,q2,nq1,q0,~reset),
    andmimo33(d4_7, q4,nq3,q2,q1,nq0,~reset),
    andmimo34(d4_8, q4,nq3,q2,q1,q0,~reset),
    andmimo35(d4_9, q4,q3,nq2,nq1,nq0,~reset),
    andmimo36(d4_10, q4,q3,nq2,nq1,q0,~reset);
      and fixedaDone(d4_11, nq4,q3,q2,nq1,q0,~valid,~reset);
```

Figure 11.5: Verilog code for Ring Mover pt. 2.

```
        //flippy floppies
        DFF_awesome df0(q0, nq0, d0, clock, reset),
                    df1(q1, nq1, d1, clock, reset),
                    df2(q2, nq2, d2, clock, reset),
                    df3(q3, nq3, d3, clock, reset),
                    df4(q4, nq4, d4, clock, reset);









        //control logic
        //done
        and and94(done_0, nq4,nq3,q2,nq1,nq0),
            and95(done_1, q4,q3,nq2,q1,q0),
            and96(done_2, q4, q3,nq2,q1,nq0);
        or  or7  (done, done_0,done_1,done_2);

        //game towers
        pl0reg tower1(empty1,dispOut1,  inits,theBus,reset,clock,~nhold1,loadInit,drive1),
               tower2(empty2,dispOut2,  zeros,theBus,reset,clock,~nhold2,loadInit,drive2),
              tower3(empty3,dispOut3,  zeros,theBus,reset,clock,~nhold3,loadInit,drive3);


        choiceReg ch1(ch1q, theBus,reset,clock, ~nholdC1),
                  ch2(ch2q, theBus,reset,clock, ~nholdC2);

        comparatorTop compy(valid, cOut, ch1q,ch2q);
        mySub         submarine(ch1q,cOut,outputSub);
        myAdd         adderall(ch2q,cOut,outputAdd);

        driveReg resultSub(theBus,reset,clock, outputSub, driveS),
                 resultAdd(theBus,reset,clock, outputAdd, driveA);









        //determines how many rings are initalized
        or or0(inits[0], sw1,sw2,sw3,sw4,sw5,sw6,sw7),
           or1(inits[1],    sw2,sw3,sw4,sw5,sw6,sw7),
           or2(inits[2],       sw3,sw4,sw5,sw6,sw7),
           or3(inits[3],          sw4,sw5,sw6,sw7),
           or4(inits[4],             sw5,sw6,sw7),
           or5(inits[5],                sw6,sw7),
           or6(inits[6],                   sw7);
```

Figure 11.6: Verilog code for Ring Mover pt. 3.

```
and andZERO0(zeros[0], reset, ~reset);
and andZERO1(zeros[1], reset, ~reset);
and andZERO2(zeros[2], reset, ~reset);
and andZERO3(zeros[3], reset, ~reset);
and andZERO4(zeros[4], reset, ~reset);
and andZERO5(zeros[5], reset, ~reset);
and andZERO6(zeros[6], reset, ~reset);

/special register to drive bus when no driver
driveReg specialReg(theBus,reset,clock, spec, specialDrive);
and specialAnd0(spec[0], reset,~reset),
    specialAnd1(spec[1], reset,~reset),
    specialAnd2(spec[2], reset,~reset),
    specialAnd3(spec[3], reset,~reset),
    specialAnd4(spec[4], reset,~reset),
    specialAnd5(spec[5], reset,~reset),
    specialAnd6(spec[6], reset,~reset),
    specialAnd10(specialDrive, ~firstDrive,~secDrive,~driveS,~driveA);




//initialization
//initial tower load
and and9960(initialLoad_0,  nq4,nq3,nq2,nq1,q0),
    and97(initialLoad_1,  nq4,nq3,nq2,q1,nq0),
    and98(initialLoad_2,  nq4,nq3,nq2,q1,q0);
or  or8   (loadInit, initialLoad_0,initialLoad_1,initialLoad_2); //GOOD!!!!

//initial tower hold
and and99(ninitialHold_0,   nq4,nq3,nq2,nq1,q0),
    and100(ninitialHold_1,  nq4,nq3,nq2,q1,nq0);
or  orKEN   (initialHold, ninitialHold_0,ninitialHold_1); //GOOD!!!!

//loading choices
//drive first choice on bus
and and111(firstDrive_0,  nq4,nq3,q2,q1,nq0),
    and112(firstDrive_1,  nq4,nq3,q2,q1,q0),
    and113(firstDrive_2,  nq4,q3,nq2,nq1,nq0);
or  or14  (firstDrive, firstDrive_0,firstDrive_1,firstDrive_2);

  //holding choice1 from bus
  and and101(nholdC1_0, nq4,nq3,q2,q1,nq0),
      and102(nholdC1_1, nq4,nq3,q2,q1,q0);
  or  orMM0(nholdC1, nholdC1_0,nholdC1_1);

//drive second choice on bus
and and114(secDrive_0,  nq4,q3,nq2,q1,nq0),
    and115(secDrive_1,  nq4,q3,nq2,q1,q0),
```

Figure 11.7: Verilog code for Ring Mover pt. 4.

```
    and116(secDrive_2,  nq4,q3,q2,nq1,nq0);
or  or15  (secDrive, secDrive_0,secDrive_1,secDrive_2);

  //holding choice 2 from bus
  and and103(nholdC2_0, nq4,q3,nq2,q1,nq0),
      and104(nholdC2_1, nq4,q3,nq2,q1,q0);
  or  orMM1(nholdC2, nholdC2_0,nholdC2_1);




//updating towers
//drive subtractor onto bus
and and132(driveS_0, q4,nq3,nq2,q1,nq0),
    and133(driveS_1, q4,nq3,nq2,q1,q0),
    and134(driveS_2, q4,q3,nq2,nq1,nq0);
or  or17  (driveS, driveS_0,driveS_1,driveS_2);

/holding information in towers
nd andkenfack2(updateHold1_0, q4,nq3,nq2,q1,nq0),
   andkenfack3(updateHold1_1, q4,nq3,nq2,q1,q0);
or  orKENKENEN1(updateHold1, updateHold1_0,updateHold1_1);




//drive adder onto bus
and anda132(driveA_0, q4,nq3,q2,q1,nq0),
    anda133(driveA_1, q4,nq3,q2,q1,q0),
    anda134(driveA_2, q4,q3,nq2,nq1,nq0);
or  orKEENENEKEN0(driveA, driveA_0,driveA_1,driveA_2);

/holding information in towers
nd andkenfack4(updateHold2_0, q4,nq3,q2,q1,nq0),
   andkenfack5(updateHold2_1, q4,nq3,q2,q1,q0);
or  orKENKENEN3(updateHold2, updateHold2_0,updateHold2_1);



//combination logic for loading and holding towers
and and117(hold1_0, updateHold1,~ch1_b2, ch1_b1),
    and118(hold2_0, updateHold1, ch1_b2,~ch1_b1),
    and119(hold3_0, updateHold1, ch1_b2, ch1_b1),

    and120(hold1_1, updateHold2,~ch2_b2, ch2_b1),
    and121(hold2_1, updateHold2, ch2_b2,~ch2_b1),
    and122(hold3_1, updateHold2, ch2_b2, ch2_b1);


//combining holds and drives
```

Figure 11.8: Verilog code for Ring Mover pt. 5.

```
  //combining holds and drives
  and and123(drive1_0, firstDrive,~ch1_b2, ch1_b1),
      and124(drive2_0, firstDrive, ch1_b2,~ch1_b1),
      and125(drive3_0, firstDrive, ch1_b2, ch1_b1),

      and126(drive1_1, secDrive,~ch2_b2, ch2_b1),
      and127(drive2_1, secDrive, ch2_b2,~ch2_b1),
      and128(drive3_1, secDrive, ch2_b2, ch2_b1);


  or  orKEN0(nhold1, hold1_0, hold1_1,initialHold),
      orKEN1(nhold2, hold2_0, hold2_1,initialHold),
      orKEN2(nhold3, hold3_0, hold3_1,initialHold),

      orKEN3(drive1, drive1_0,drive1_1),
      orKEN4(drive2, drive2_0,drive2_1),
      orKEN5(drive3, drive3_0,drive3_1);
endmodule
```

Figure 11.9: Verilog code for Ring Mover pt. 6.

```
module comparatorTop(valid, cOut, c1,c2);

  input [6:0] c1, c2;
  output valid;
  output [6:0] cOut;

  wire [6:0] seven, six, five, four, three, two, one;

  or orSev(seven[6], c1[1], ~c1[1]);
  or orSix(six[5], c1[1], ~c1[1]);
  or orFiv(five[4], c1[1], ~c1[1]);

  or orFou(four[3], c1[1], ~c1[1]);
  or orThr(three[2], c1[1], ~c1[1]);
  or orTwo(two[1], c1[1], ~c1[1]);
  or orOne(one[0], c1[1], ~c1[1]);

  and andSev1(seven[5],c1[1],~c1[1]);
  and andSev2(seven[4],c1[1],~c1[1]);
  and andSev3(seven[3],c1[1],~c1[1]);
  and andSev4(seven[2],c1[1],~c1[1]);
  and andSev5(seven[1],c1[1],~c1[1]);
  and andSev6(seven[0],c1[1],~c1[1]);

  and andSix1(six[6],c1[1],~c1[1]);
  and andSix2(six[4],c1[1],~c1[1]);
  and andSix3(six[3],c1[1],~c1[1]);
  and andSix4(six[2],c1[1],~c1[1]);
  and andSix5(six[1],c1[1],~c1[1]);
  and andSix6(six[0],c1[1],~c1[1]);

  and andFiv1(five[6],c1[1],~c1[1]);
  and andFiv2(five[5],c1[1],~c1[1]);
  and andFiv3(five[3],c1[1],~c1[1]);
  and andFiv4(five[2],c1[1],~c1[1]);
  and andFiv5(five[1],c1[1],~c1[1]);
  and andFiv6(five[0],c1[1],~c1[1]);


  and andFou1(four[6],c1[1],~c1[1]);
  and andFou2(four[5],c1[1],~c1[1]);
  and andFou3(four[4],c1[1],~c1[1]);
  and andFou4(four[2],c1[1],~c1[1]);
  and andFou5(four[1],c1[1],~c1[1]);
  and andFou6(four[0],c1[1],~c1[1]);

  and andThr1(three[6],c1[1],~c1[1]);
  and andThr2(three[5],c1[1],~c1[1]);
  and andThr3(three[4],c1[1],~c1[1]);
  and andThr4(three[3],c1[1],~c1[1]);
  and andThr5(three[1],c1[1],~c1[1]);
  and andThr6(three[0],c1[1],~c1[1]);
```

Figure 11.10: Verilog code for Comparator pt. 1.

```
  and andTwo1(two[6],c1[1],~c1[1]);
  and andTwo2(two[5],c1[1],~c1[1]);
  and andTwo3(two[4],c1[1],~c1[1]);
  and andTwo4(two[3],c1[1],~c1[1]);
  and andTwo5(two[2],c1[1],~c1[1]);
  and andTwo6(two[0],c1[1],~c1[1]);

  and andOne1(one[6],c1[1],~c1[1]);
  and andOne2(one[5],c1[1],~c1[1]);
  and andOne3(one[4],c1[1],~c1[1]);
  and andOne4(one[3],c1[1],~c1[1]);
  and andOne5(one[2],c1[1],~c1[1]);
  and andOne6(one[1],c1[1],~c1[1]);



  //compares the first tower picked with the second tower picked
  myCompare compararer(c1,c2,valid);

    //top ring on first choice
  //loc = 0010000 means 5th smallest ring is on top!
  compareSeven cs1(c1,seven,GEseven1);
  compareSeven cs10(c1,six,GEsix1);
  compareSeven cs2(c1,five,GEfive1);
  compareSeven cs3(c1,four,GEfour1);
  compareSeven cs4(c1,three,GEthree1);
  compareSeven cs5(c1,two,GEtwo1);
  compareSeven cs6(c1,one,GEone1);

  and andken7(cOut[6], GEseven1,GEsix1,GEfive1,GEfour1,GEthree1,GEtwo1,GEone1);
  and andken6(cOut[5], ~GEseven1,GEsix1,GEfive1,GEfour1,GEthree1,GEtwo1,GEone1);
  and andken5(cOut[4], ~GEseven1,~GEsix1,GEfive1,GEfour1,GEthree1,GEtwo1,GEone1);
  and andken4(cOut[3], ~GEseven1,~GEsix1,~GEfive1,GEfour1,GEthree1,GEtwo1,GEone1);
  and andken3(cOut[2], ~GEseven1,~GEsix1,~GEfive1,~GEfour1,GEthree1,GEtwo1,GEone1);
  and andken2(cOut[1], ~GEseven1,~GEsix1,~GEfive1,~GEfour1,~GEthree1,GEtwo1,GEone1);
  and andken1(cOut[0], ~GEseven1,~GEsix1,~GEfive1,~GEfour1,~GEthree1,~GEtwo1,GEone1);

endmodule
```

Figure 11.11: Verilog code for Comparator pt. 2.

```
module myAdd(
    data0x,
    data1x,
    result
);


input wire  [6:0] data0x;
input wire  [6:0] data1x;
output wire [6:0] result;




parallel_add0   b2v_inst(
    .data0x(data0x),
    .data1x(data1x),
    .result(result));


endmodule



// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module parallel_add0 (
    data0x,
    data1x,
    result);

    input   [6:0]  data0x;
    input   [6:0]  data1x;
    output  [6:0]  result;

    wire [6:0] sub_wire0;
    wire [6:0] sub_wire3 = data1x[6:0];
    wire [6:0] result = sub_wire0[6:0];
    wire [6:0] sub_wire1 = data0x[6:0];
    wire [13:0] sub_wire2 = {sub_wire3, sub_wire1};

    parallel_add    parallel_add_component (
                .data (sub_wire2),
                .result (sub_wire0)
                // synopsys translate_off
                ,
                .aclr (),
                .clken (),
                .clock ()
```

Figure 11.12: Verilog code for adder pt. 1.

```
                defparam
                    parallel_add_component.msw_subtract = "NO",
                    parallel_add_component.pipeline = 0,
                    parallel_add_component.representation = "UNSIGNED",
                    parallel_add_component.result_alignment = "LSB",
                    parallel_add_component.shift = 0,
                    parallel_add_component.size = 2,
                    parallel_add_component.width = 7,
                    parallel_add_component.widthr = 7;


                endmodule
```

Figure 11.13: Verilog code for adder pt. 2.

```
module pIOreg(empty,dispOuts, inits,theBUS,reset,clock, hold,loadInit,drive);

  output empty;
  output [6:0] dispOuts;

  input [6:0] inits;
  input hold;
  input loadInit;
  input reset,clock;
  input drive;

  inout [6:0] theBUS;

  not not0(q1, nq1);
  not not1(q2, nq2);
  not not2(q3, nq3);
  not not3(q4, nq4);
  not not4(q5, nq5);
  not not5(q6, nq6);
  not not6(q0, nq0);

  //for empty:
  and andE(empty, nq0, nq1,nq2,nq3,nq4,nq5,nq6);

  //for DFF0
  and and1(hold0, dispOuts[0], hold),
      and2(parallel0, inits[0],~hold,loadInit),
      and3(parallel00, theBUS[0],~hold,-loadInit);
  or or1(d0, hold0, parallel0, parallel00);


  //for DFF1
  and and4(hold1, dispOuts[1], hold),
      and5(parallel1, inits[1],~hold,loadInit),
      and6(parallel11, theBUS[1],~hold,-loadInit);
  or or2(d1, hold1, parallel1, parallel11);

  //for DFF2
  and and7(hold2, dispOuts[2], hold),
      and8(parallel2, inits[2],~hold,loadInit),
      and9(parallel22, theBUS[2],~hold,-loadInit);
  or or3(d2, hold2, parallel2, parallel22);

  //for DFF3
  and and10(hold3, dispOuts[3], hold),
      and11(parallel3, inits[3],~hold,loadInit),
```

Figure 11.14: Verilog code for parallel register pt. 1.

```
  //for DFF3
  and and10(hold3, dispOuts[3], hold),
      and11(parallel3, inits[3],~hold,loadInit),
      and90(parallel33, theBUS[3],~hold,-loadInit);
  or or4(d3, hold3, parallel3, parallel33);

  //for DFF4
  and and71(hold4, dispOuts[4], hold),
      and81(parallel4, inits[4],~hold,loadInit),
      and91(parallel44, theBUS[4],~hold,-loadInit);
  or or5(d4, hold4, parallel4, parallel44);

  //for DFF5
  and and72(hold5, dispOuts[5], hold),
      and82(parallel5, inits[5],~hold,loadInit),
      and92(parallel55, theBUS[5],~hold,-loadInit);
  or or6(d5, hold5, parallel5, parallel55);

  //for DFF6
  and and73(hold6, dispOuts[6], hold),
      and83(parallel6, inits[6],~hold,loadInit),
      and93(parallel66, theBUS[6],~hold,-loadInit);
  or or7(d6, hold6, parallel6, parallel66);


  //DFF(q, qBar, D, clk, rst);
  DFF_awesome   DFF1(dispOuts[0], nq0, d0, clock, reset),
                     DFF2(dispOuts[1], nq1, d1, clock, reset),
                    DFF3(dispOuts[2], nq2, d2, clock, reset),
                    DFF4(dispOuts[3], nq3, d3, clock, reset),
                    DFF5(dispOuts[4], nq4, d4, clock, reset),
                    DFF6(dispOuts[5], nq5, d5, clock, reset),
                    DFF7(dispOuts[6], nq6, d6, clock, reset);



  bufif1 bffff0(theBUS[0], dispOuts[0],drive),
         bffff1(theBUS[1], dispOuts[1],drive),
         bffff2(theBUS[2], dispOuts[2],drive),
         bffff3(theBUS[3], dispOuts[3],drive),
         bffff4(theBUS[4], dispOuts[4],drive),
         bffff5(theBUS[5], dispOuts[5],drive),
         bffff6(theBUS[6], dispOuts[6],drive);


endmodule
```

Figure 11.15: Verilog code for parallel register pt. 2.

```
module choiceReg(q, theBUS,reset,clock, hold);

  output [6:0] q;
  input hold;
  input reset,clock;

  input [6:0] theBUS;

  //for DFF0
  and and1(hold0, q[0], hold),
      and2(parallel00, theBUS[0],~hold);
  or or1(d0, hold0, parallel00);


  //for DFF1
  and and3(hold1, q[1], hold),
      and4(parallel11, theBUS[1],~hold);
  or or2(d1, hold1, parallel11);

  //for DFF2
  and and5(hold2, q[2], hold),
      and6(parallel22, theBUS[2],~hold);
  or or3(d2, hold2, parallel22);

  //for DFF3
  and and7(hold3, q[3], hold),
      and8(parallel33, theBUS[3],~hold);
  or or4(d3, hold3, parallel33);

  //for DFF4
  and and9(hold4, q[4], hold),
      and10(parallel44, theBUS[4],~hold);
  or or5(d4, hold4, parallel44);

  //for DFF5
  and and11(hold5, q[5], hold),
      and12(parallel55, theBUS[5],~hold);
  or or6(d5, hold5, parallel55);

  //for DFF6
  and and13(hold6, q[6], hold),
      and14(parallel66, theBUS[6],~hold);
  or or7(d6, hold6, parallel66);


  //DFF(q, qBar, D, clk, rst);
  DFF_awesome   DFF1(q[0], nq0, d0, clock, reset),
                DFF2(q[1], nq1, d1, clock, reset),
                DFF3(q[2], nq2, d2, clock, reset),
                DFF4(q[3], nq3, d3, clock, reset),
                DFF5(q[4], nq4, d4, clock, reset),
                DFF6(q[5], nq5, d5, clock, reset),
                DFF7(q[6], nq6, d6, clock, reset);
endmodule
```

Figure 11.16: Verilog code for choice register

```
module gameTimerClock(gTClock, iClock,reset);

  output gTClock; //a PERFECT 1 Hz clock
  input iClock,reset; //24MHz internal clock

  reg [25:0]  tBase;                    // system timebase

  reg gTClock;

  always@(posedge iClock) begin
    if (reset) begin
      tBase <= 1'b0;
    end else begin
      tBase <= tBase + 1'b1;
    end

    gTClock <= tBase[22];

  end


endmodule
```

Figure 11.17: Timing diagram of tests on timer clock.

```
module playingClock(pClock, iClock,reset);

  output pClock; //a PERFECT 1 Hz clock
  input iClock,reset; //24MHz internal clock

  reg [25:0]  tBase;                    // system timebase

  reg pClock;

  always@(posedge iClock) begin
    if (reset) begin
      tBase <= 1'b0;
    end else begin
      tBase <= tBase + 1'b1;
    end

    pClock <= tBase[18];

  end


endmodule
```

Figure 11.18: Timing diagram of tests on game clock.

```
module DFF_awesome(q, qBar, D, clk, rst);
  input D, clk, rst;
  output q, qBar;

  reg q;

  not n1 (qBar, q);

  always@ (posedge rst or posedge clk)
  begin
    if(rst)
      q = 0;

    else
      q = D;

  end
```

Figure 11.19: Verilog code for flip flop.