Gareth Peters (260678626)
Peter Quinn (260689207)

November 24, 2017
ECSE 323

Lab 4: rules, dealer, lab4_testbed

Gareth Peters (260678626)
Peter Quinn (260689207)

Gareth Peters (260678626)
Peter Quinn (260689207)

November 24, 2017
ECSE 323

Introduction

The purpose of this lab was to begin creating the circuits necessary to play the game blackjack (also known as 21). A circuit to control the rules of the game was built, as well as a circuit capable of dealing a card from a stack randomly. A testbed was also designed to test the dealer circuit, and integrate it with the testbed developed in lab 3. The circuits designed in this lab are g07_rules, g07_dealer_FSM and g07_lab4_testbed.

Gareth Peters (260678626)
Peter Quinn (260689207)

November 24, 2017
ECSE 323

g07_rules

Design Method

This circuit is designed to check if a player busts in a game of 21. A player busts in 21 if their hand sums up to a number greater than 21. A further explanation of the rules for 21 can be seen in the document for Lab 4 [1].

This circuit takes two inputs. The first is card_play, a 6-bit input that represents the new card most recently deal to the player. The second input, play_pile_top_card, is a 6-bit input that represents the player's current hand. The MSB represents whether the player was holding an ace that was counted as 11 (1 bit is sufficient for this since no more than 1 ace can be counted as 11). The bottom five bits represent the sum of the player's hand so far.

The circuit has two outputs, legal_play and sum. legal_play is 1 bit representing whether or not the play was legal (i.e. the play was successful as the player's hand did not exceed 21.). The sum output is 6 bits and represent the new total of the cards in the hand (bottom 5 bits), along with whether or not the hand contains an ace being counted as 11 (highest bit/MSB).

The modulo 13 of the card is taken to strip the suit from the card, leaving only the value. This is accomplished using the g07_Modulo_13 circuit designed in lab 1. The output of the modulo circuit and the bottom 5 bits of play_pile_top card are converted to integers to simplify the VHDL code.

The value of the new is then set to the correct value for the game using the following logic.

1. Add 1 to the value
2. Check if the value is greater than 10
    a. If >10 then set the value to 10
        i. Sets the face cards to 10
3. Check if the value is 1
    a. If = then set the value to 11
        i. Having a 1 means we have an ace, and we want to start with it assuming its higher value

After the proper value has been set for the new card, the following logic is implemented to determine whether the player's hand would exceed 21 with the addition of the new card. The new sum is computed and the number of aces that are being counted as 11 are also determined according to the flow chart.

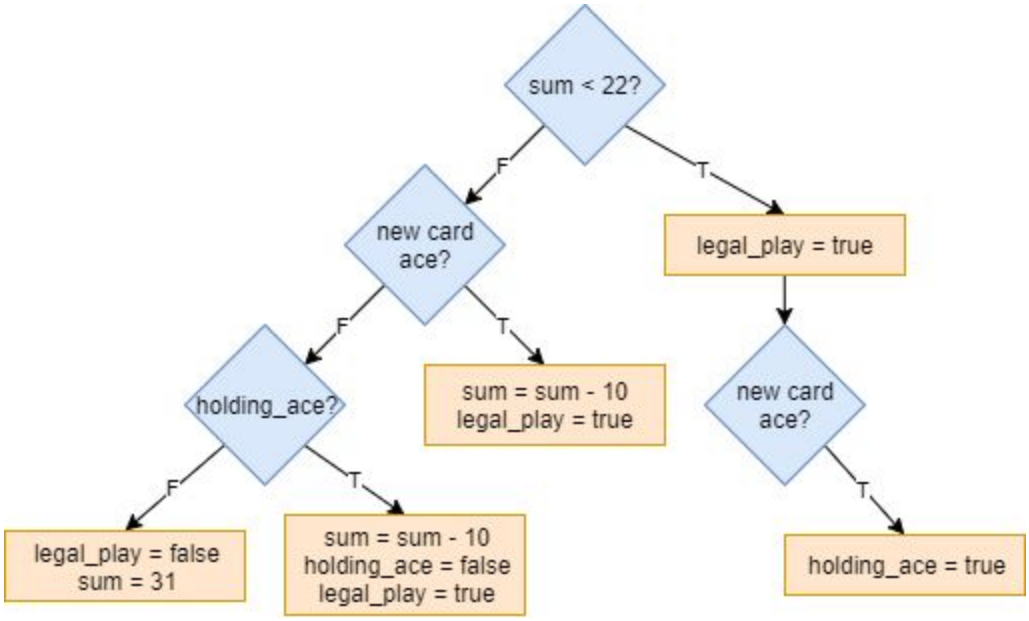

Fig. 1: Flowchart describing logic used in g07_rules

Once the new sum is computed, it is encoded along with whether or not the hand contains an ace that is being counted as 11. If the sum exceeds 21, it is set to 31 arbitrarily . The sum and ace are encoded to match the format expected by the play_pile_top_card input. Note that this circuit automatically decides how to count aces (as 11 or 1). The circuit makes the choice that will prevent the hand from exceeding 21 whenever possible. .

Schematic

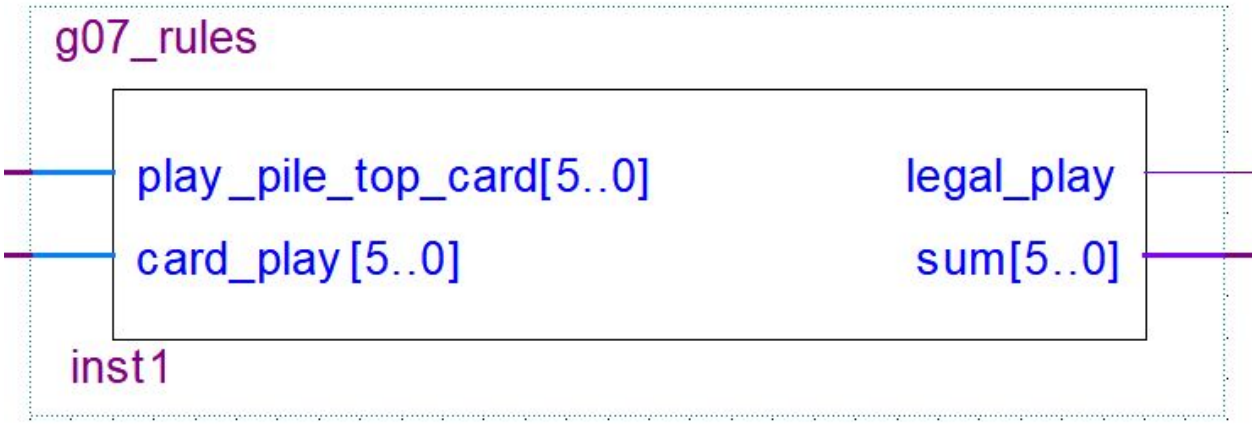

Fig. 2: Symbol file of g07_rules

This circuit was built by programming the logic with VHDL. As such, no gate level diagram is available. The VHDL code is provided here.

Simulation/Hardware Synthesis Procedure

This circuit was tested using several voltage waveforms to show that it behaved properly in typical situations it would encounter.

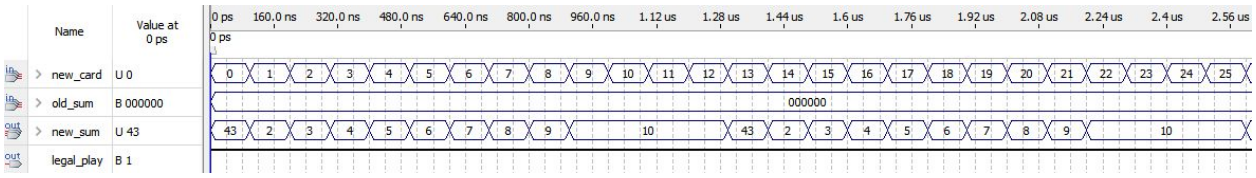

Fig. 3: The circuit is able to correctly parse new card inputs to determine the value of the card, and assign it the correct value for the game. Note that 43 corresponds to 32+11, which means that we have a 1 in the 2^5=32 place indicating that we are holding an ace being counted as 11.

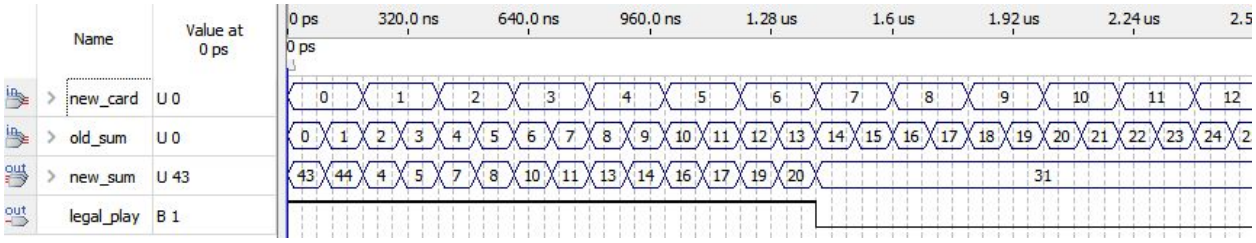

Fig. 4: The circuit correctly adds cards and old sums to determine whether or not the play is legal

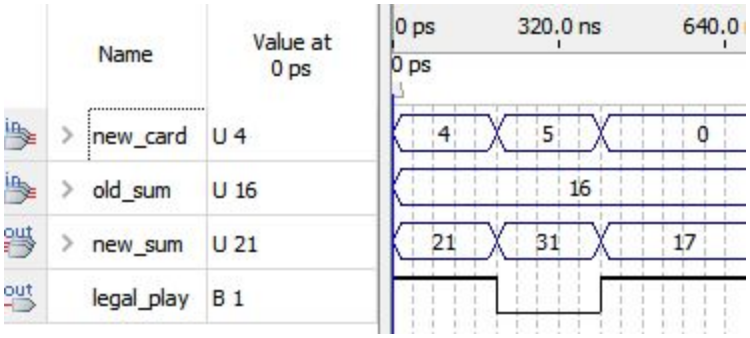

Fig. 5: When new_card is 4, a value of 5 is added to the old sum 16 for a total of 21, a legal play. When new_card is 5, a value of 6 is added to 16 for a total of 22, an illegal play. When new_card is 0 (an ace), the value 1 is selected and added to 16 for a total of 17, a legal play.

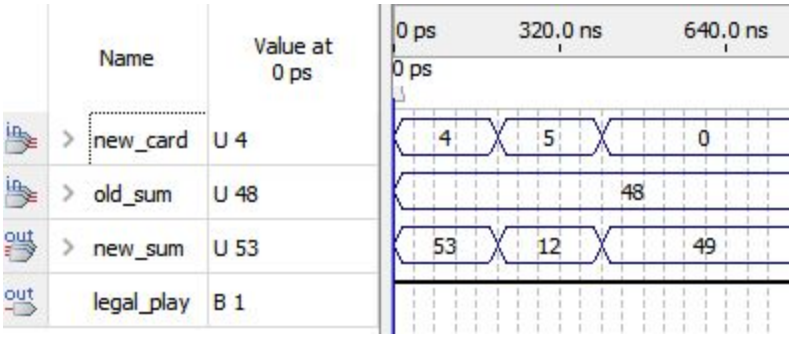

Fig. 6: When new_card is 4, a value of 5 is added to the old sum 16 (48-32) for a total of 21, a legal play. The ace held in the old hand is still being counted as 11. When new_card is 5, a value of 6 is added to 16 for a total of 22, an illegal play, but since there was an ace counted in the old hand, it can be counted as 1, giving the total 12 which is a legal play. The old ace is then removed from the hand. When new_card is 0 (an ace), the value 1 is selected for the new ace and added to 16 for a total of 17, a legal play with one ace still being counted as 11 (32+17=49)

Results and Discussion

The circuit performed correctly in all test scenarios. It is able to determine whether or not a hand will exceed 21 with the addition of a new card, taking into account aces in the past hand.

An advantage of this circuit is that is able to quickly compute the outputs and does not rely on the system clock, so there is minimal delay.

This circuit has some limitations. The circuit requires that both inputs be formatted correctly. The circuit requires external memory to hold the previous value of hand.

g07_dealer_FSM

Design Method

The purpose of the dealer FSM is to provide the enable for the stack to remove a card at a random address. This is done with Moore style FSM logic. There are 3 inputs for the FSM: request_deal, reset and clock. The only output is the enable signal for the stack to pop. It was decided to do the random generation of the addresses different to the way explained in Lab 4 [1].

Instead of continuously generating a random number until a number is obtained that is between NUM-1 and 0, it was found that a more efficient way to do this would be to instead use an lpm_divide circuit to perform RANDUmod(NUM) where NUM is the number of elements in the stack. This always provides a random address that is never greater than NUM. This is explained more in the g07_lab4_testbed section. This then removes the requirement of a rand_enable output that would enable the flip flop of the stored random number. It also removes the need for the RAND_LT_NUM be inputted to the g07_dealer_FSM.

This FSM has 3 states: Wait for Low, Wait for High, and Enable Stack. The state diagram used is what is in Fig. 7. The Wait for Low transitions to Wait for High when request_deal input goes low, and the Wait for High state goes to Enable Stack when request_deal is high. Otherwise, Wait for Low and Wait for High stay keep their states. After one clock cycle, Enable stack goes back to waiting for low no matter what the request_deal input is. The reset input bit resets the FSM back to Wait for Low.

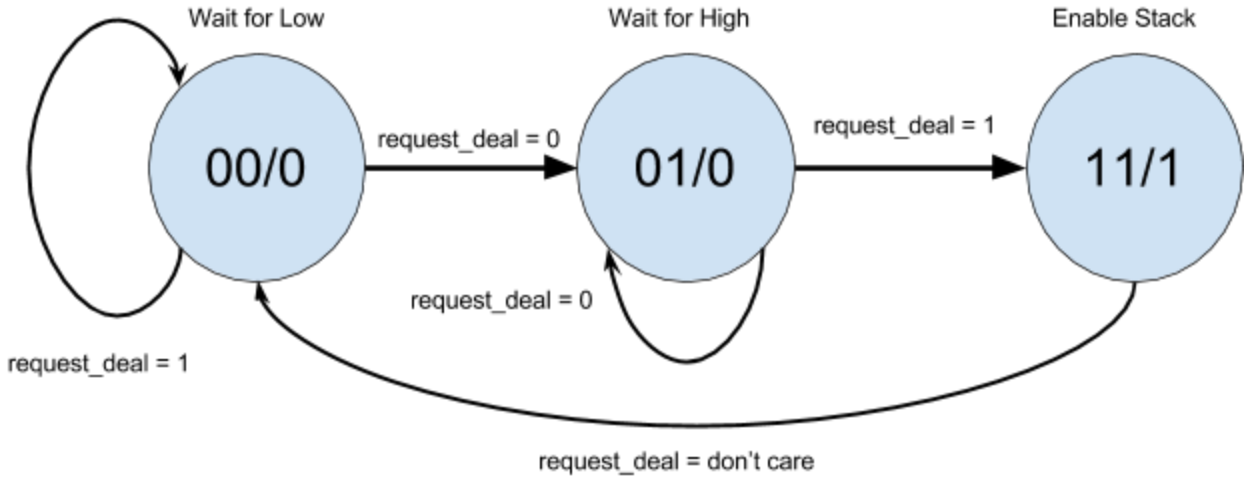

Fig. 7: State transition diagram of the Moore-style g07_dealer_FSM

Schematic and VHDL Description

This circuit was built by programming the logic with VHDL. As such, no gate level diagram is available. The VHDL code is provided here.

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   library lpm;
4   use lpm.lpm_components.all;
5
6   entity g07_dealer_FSM is
7
8   port (request_deal: in std_logic;
9         reset:in std_logic;
10        clock:in std_logic;
11        stack_enable:out std_logic);
12
13  end g07_dealer_FSM;
14
15  architecture g07_dealer_FSM_arch of g07_dealer_FSM is
16     TYPE state_signal IS (WAIT_LOW,WAIT_HIGH,ENABLE_STACK);
17     signal state: state_signal;
18  begin
19     update: process(clock,reset)
20     begin
21        if reset = '1' then
22           state<= WAIT_LOW;
23        elsif clock'EVENT and clock = '1' then
24           case state is
25           when WAIT_LOW =>
26              if request_deal = '0' then state<= WAIT_HIGH; end if;
27           when WAIT_HIGH =>
28              if request_deal = '1' then state<= ENABLE_STACK; end if;
29           when ENABLE_STACK => state <= WAIT_LOW;
30           end case;
31        end if;
32     end process;
33
34     output_logic: process(state)
35     begin
36        case state is
37           when WAIT_LOW => stack_enable <= '0';
38           when WAIT_HIGH=> stack_enable <= '0';
39           when ENABLE_STACK=> stack_enable <= '1';
40        end case;
41     end process;
42  end architecture;
```

Fig. 8: VHDL description of the g07_dealer_FSM

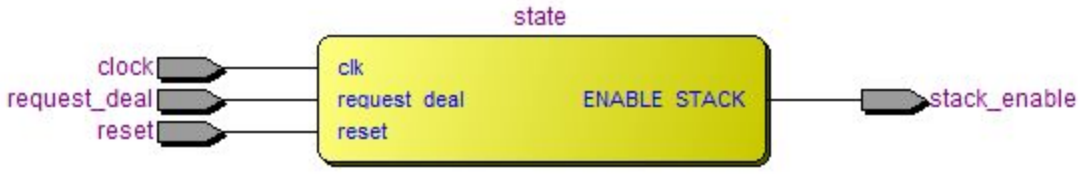

Fig. 9: Schematic diagram of the g07_dealer_FSM

Simulation/Hardware Synthesis Procedure

Timing simulation was performed with a 50MHz clock signal, which is what the clock for the stack will be in the g07_lab4_testbed. The request_deal input was first set to high to show that nothing actually occurs during this state because the circuit first waits for a low input. Request_deal is then set to zero, which changes the state from Wait for Low to Wait for High. In normal circumstances, once request_deal goes back to high, the stack_enable output will be set to high. However, in the simulation the reset bit is set to high to show that the reset bit resets the FSM to be in Wait for Low state. The reset bit is then set back to low in the simulation. Request_deal goes from high to low again, and then from low to high to show that the stack_enable output gets set to high as it should. Notice how the stack_enable output lasts for a single clock cycle and overlaps with the next rising edge so that the stack can pop on the next clock cycle.

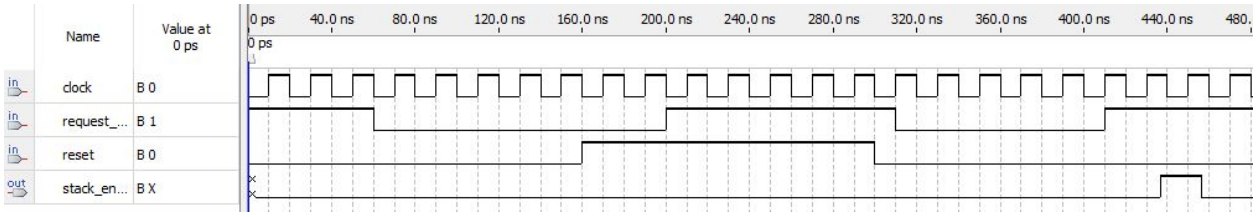

Fig. 10: Timing simulation of the g07_dealer_FSM circuit

Results and Discussion

The circuit enables the stack after waiting for low and then high as it should.

The main advantage to this circuit is that it does not have to waste clock cycles waiting for a proper random number to be generated from g07_RANDU. This is especially convenient for cases where NUM is small. In our design, g07_RANDU always has a number ready that is between 0 and NUM-1 because of the lpm_divide circuit.

However, by removing the need to compare the random number in the g07_dealer_FSM circuit, it has a very close functionality to that of the g07_debouncer circuit developed in Lab 3. The difference between g07_dealer_FSM and g07_debouncer is that the g07_debouncer uses only one latch while the dealer FSM uses 2 since it has 3 states. This is caused by the constraint of

having the FSM be a Moore model. Another difference is the fact the g07_debouncer has a flexible amount of time to wait caused by the lpm_counter, and the g07_dealer_FSM circuit will produce a signal at every 3rd clock cycle no matter what.

g07_lab4_testbed

Design Method

This circuit implements all of the same functionality as g07_lab3_testbed, with the addition of some components. Only the new components and functionality will be discussed here. For the description of the rest of the circuit, see Lab 3 report.

The additional components are as follows:

1. Random number generator
2. Dealer FSM
3. Number of Cards in Stack Display

Random Number Generator

The random number generator is used to create random addresses for the dealer circuit. 32 bit random numbers are generated using the g07_RANDU circuit. The random values are stored in a 32 bit LPM_FF [2]. This flip flop also provides the initial seed value of the RANDU circuit. After a random value is produced, it is fed back into g07_RANDU to produce the next random number. The initial value of the flip flop is set to 42 when the stack is initialized (both mode bits are 1 and the trigger button has been pressed). An lpm_divide module is used to take the modulus of the 6 most significant (most random) bits of the 32 bit random number with respect to the number cards in the stack. This ensures that we produce a 6 bit address that is valid (i.e. within the range 0 to NUM-1, which are the addresses that have cards in the stack).

A new random number is generated every clock cycle. This forces the random numbers to be a function of time whose output changes rapidly. This ensures that different values occur each time the game is played, as long as cards are not dealt the exact same time each play. Since the clock period is 20 ns and the board relies on mechanical components and human input, the numbers produced will be essentially random.

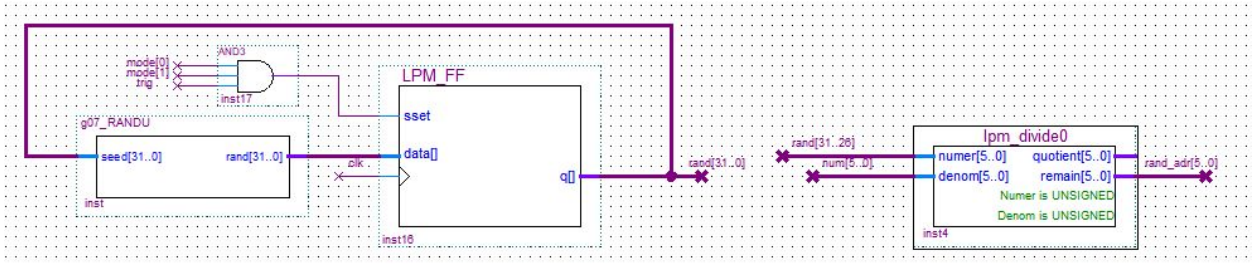

Fig. 11: Schematic of the components involved in the generation of random numbers in g07_lab4_testbed

Dealer Finite State Machine

The g07_dealer_FSM circuit is implemented into the testbed circuit. It's request_deal input is connected to a button on the board (following the g07_debouncer circuit) and its reset input is connected to the reset input that is connected to all other resets. The stack_enable output of the g07_dealer_FSM is connected to the enable of two separate 2x1 multiplexers.

The first multiplexer has its enable=1 input connected to a lpm_constant circuit that contains the POP operation. The second input is attached to the dip switches on the board so that any other operation can be performed (especially INIT). This multiplexer has its output connected to the mode input of the stack. The second multiplexer works similarly to the first but instead controls the addresses to be seen by the stack. When enable=1, the output is given the random address generated by g07_RANDU. If enable=0, then the address to be seen the address bits controlled by dip switches on the board so that each FF can be still viewed in the stack.

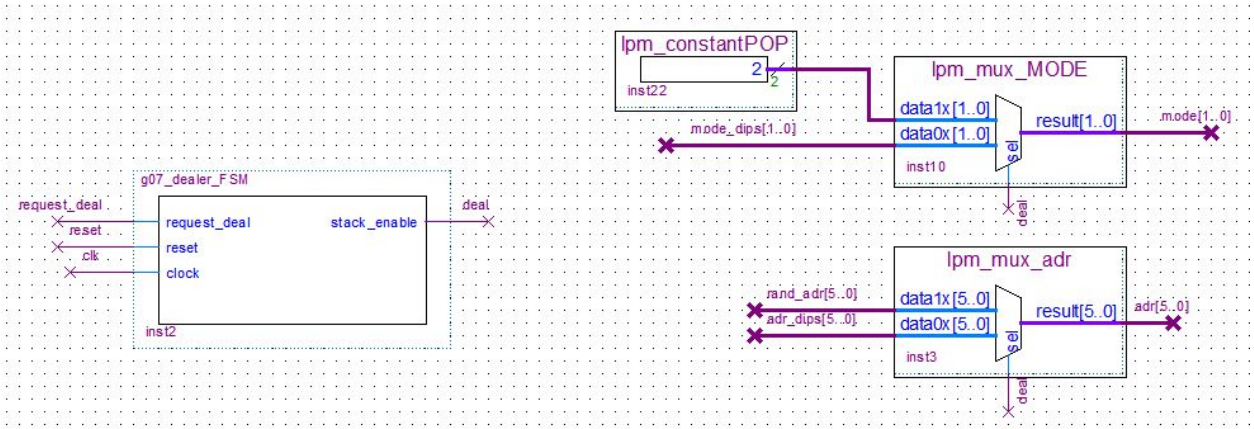

Fig. 12: The Dealer Finite State Machine section of the testbed circuit

Number of Cards in Stack Display

Circuitry to display the number of cards in the stack using LED displays 3 and 4 was implemented. The 6 bit number from the stack was converted to base 10 using an lpm_divide component to divide by 10 to find the quotient and the remainder. The quotient (10s digit) was passed to a g07_7_segment_decoder connected to LED display 4. The remainder (1s digit) was passed to another g07_7_segment_decoder connected to LED display 3.

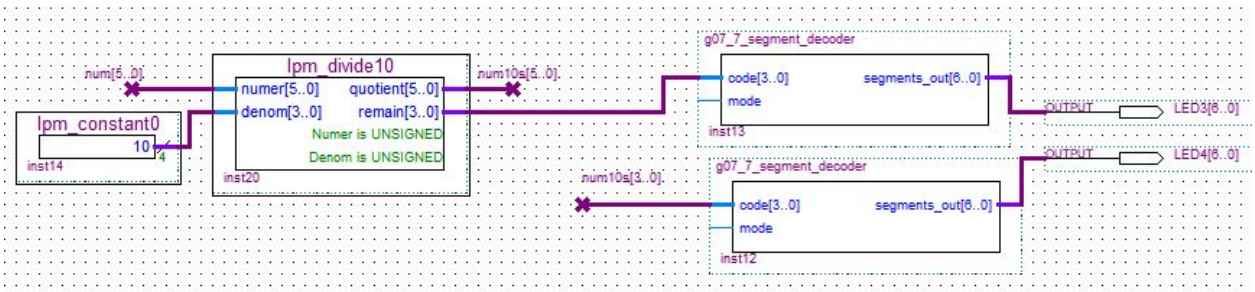

Fig. 13: Circuitry to display the number of cards in the stack using LED displays 3 and 4

Simulation/Hardware Synthesis Procedure

The circuit was tested by uploading it to the board. It was found that the circuit behaved as expected. When the deal card button was pushed, a random card was removed from the stack. The stack was reset several times, and cards dealt, to ensure that the cards were really dealt in a random order.

A timing analysis of the testbed circuit was done and it was found that the longest propagation delay in the circuit 18.798 ns.

Total logic elements	476 / 18,752 (3 %)
Total combinational functions	465 / 18,752 (2 %)
Dedicated logic registers	220 / 18,752 (1 %)
Total registers	220
Total pins	42 / 315 (13 %)
Total virtual pins	0
Total memory bits	3,328 / 239,616 (1 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

Table 1: Flow Summary of g07_lab4_testbed

Results and Discussion

The testbed circuit functioned correctly. The popped cards were always at a random address, and were not predictable values.

As discussed in the g07_dealer_FSM, the testbed has the advantage that g07_dealer_FSM will not be waiting on a correct random number to be generated. In this testbed design, a correct random number is guaranteed and so no time is wasted. Furthermore, the random generator is much more random due to the fact the FF that contains the random number is continuously overwritten every clock cycle by a new random number from g07_RANDU. This provides a more random card dealing for the user as a true card deal should be.

A limitation of our circuit would be that it uses several lpm_divide circuits. Even though these circuits are convenient for reducing the amount of time of the g07_dealer_FSM, they are also expensive in terms of circuit components. This is because division circuits are generally expensive for doing the division of a single number, where lpm_divide can divide by any number.

References

[1] J. Clark, Lab #4 – VHDL for Sequential Circuit Design. Montreal: McGill, 2017.

[2] "LPM Quick Reference Guide", Altera.com, 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/catalogs/lpm.pdf. [Accessed: 24- Nov- 2017].