Abbas Yadollahi – 260680343
He Qian Wang – 260688073

November 24, 2017
Team A6

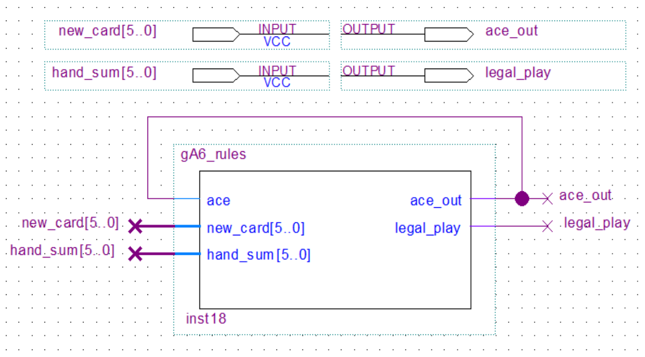
Lab 4 Report: gA6 Dealer Rules

PART 1: RULES

Design Method

The first part of the lab required us to build a rules module for a game of Blackjack. The purpose of this circuit is to tell us whether the player's hand will bust or not. Basically, we are taking the value of the new card that is drawn and adding it to the total value of the cards in the player's hand. If the new sum is above 21, then the player bust and he is out of the game, otherwise, he has a legal play. There are a few exceptional cases that we need to consider such as when there is already an ace in the player's hand or if the drawn card is an ace. In Blackjack, an ace can either adopt the value of 1 or 11. The rules module accounts for the different cases with aces and updates the value of the hand accordingly.

Hardware



To build the rules module, we used VHDL. We have two 6-bit inputs *new_card* and *hand_sum*, a 1-bit input *ace* and two 1-bit outputs *legal_play* and *ace_out*. The inputs are respectively the card that is drawn, the total value in hand and if there's currently an ace in hand. The outputs are respectively to indicate whether the player has a legal play or bust and if there's an ace left in hand. The output *ace_out* is fed back into the input *ace* the loop using a simple wire. This is to track the presence of an ace in our hand prior to drawing.

$$face\_value = V \; modulo \; 13 + 1$$

In this game, the suit of the card is irrelevant since we are only looking for the card's face value. The value V of the cards ranges in between 0 to 51 which represent the 52 cards in a deck. When we receive a card, we use a modulo 13 on its value to obtain the card's face value and we add one 1 to the result since the cards start from 0 and not 1.

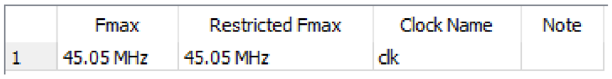
To have access to arithmetic operations, we imported the *ieee.numeric_std.all* library. Instead of comparing binary numbers bit by bit, we can simply use decimal numbers and unsigned versions of the face_value when performing our operations. The entire structure of the VHDL design is within a process block with *new_card* and *hand_sum* for the sensitivity list. From there, we define a few variables to keep track of the aces, the value of the new card and the new total value of the cards in hand.

The first step is to we obtain the face value of the card that is drawn by using the modulo 13 and add 1 formula. When we detect that the new card is an ace, we by default start by setting its face value to 11, although its value may change depending on the new sum in hand, and we set one ace in hand. If we detect that the new card is a Jack, Queen or King, we set its value down to 10 sinces in Blackjack, face cards are only worth 10.

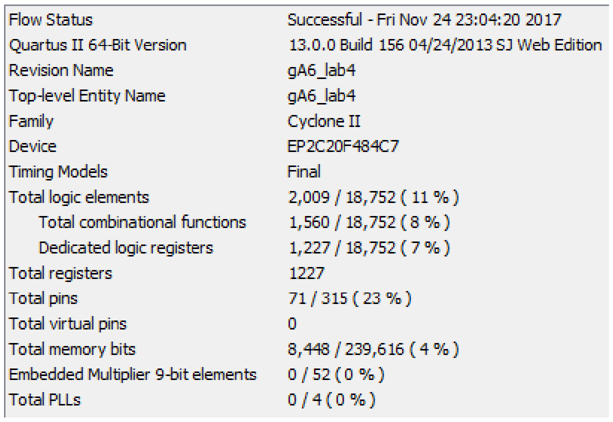
In the next step, we take the sum of the new card and the total in hand and check whether this tentative sum is bigger than 21. If the sum is bigger than 21, we will have to check whether we have an ace in hand. If we have an ace present, we set its value from 11 to 1. If not, we check if the card drawn is an ace and if so, we set its value from 11 to 1. Afterwards, we check if the updated sum is smaller than 22. If so, we set the output *legal_play* to high and we set *legal_play* to low otherwise. At the end, if we have an ace left in hand, we set *ace_out* to high.

Analysis

Compilation Analysis



	Fmax	Restricted Fmax	Clock Name	Note
1	45.05 MHz	45.05 MHz	clk	

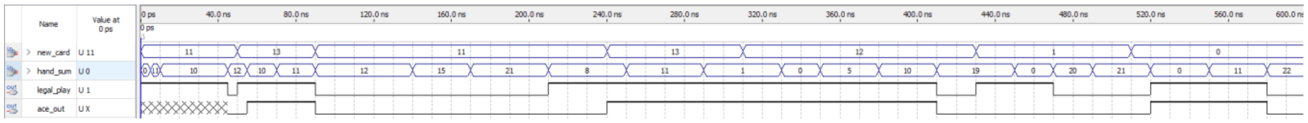


Flow Status	Successful - Fri Nov 24 23:04:20 2017
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	gA6_lab4
Top-level Entity Name	gA6_lab4
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	2,009 / 18,752 (11 %)
Total combinational functions	1,560 / 18,752 (8 %)
Dedicated logic registers	1,227 / 18,752 (7 %)
Total registers	1227
Total pins	71 / 315 (23 %)
Total virtual pins	0
Total memory bits	8,448 / 239,616 (4 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

The restricted maximum frequency of our rules module is $F_{max} = 45.05\ MHz$, which is quick enough considering that our circuit runs at 50 Mhz. Although, after comparing our circuit to those of other colleagues, we notice that our maximum frequency is quite average.

The total propagation delay of our circuit can be calculated using the inverse of the maximum frequency. Thus, we find that $t_p = \frac{1}{F_{max}} = 22.20\ ns$.

Simulation



The simulation is done using waveform simulations. We are unable to check all possible values but we paid specific attention to edge cases and to situations that need to handle aces. As we can see from the waveform above, whenever the sum of the *new_card* and *hand_sum* is over 21, we have *legal_play* set to low. Although, when the *new_card* is an ace or there is an ace currently in the player's hand, there is a possibility for the total sum to be under 22 and have *legal_play* set to high.

Remarks

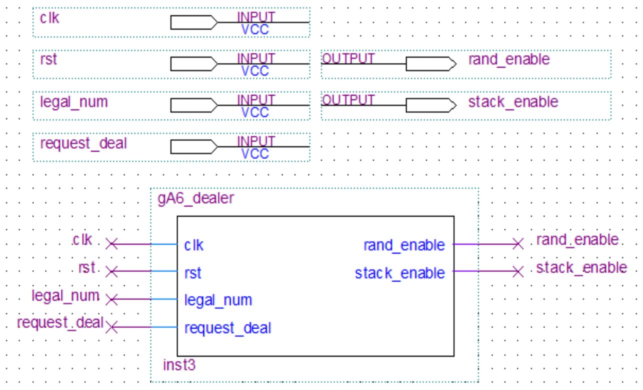
Most of the limitations of the modules concern the presence of aces in the hand of the player and it is difficult to keep track of them. For example, the cases where we have more than 1 ace are troublesome. Another issue that could be error prone with this module is the initially undefined value of the input *ace*. Since it uses a feedback from the output, it is initially undeclared, which can be seen in the waveform. These placed a few limitations on our code that we had to account for.

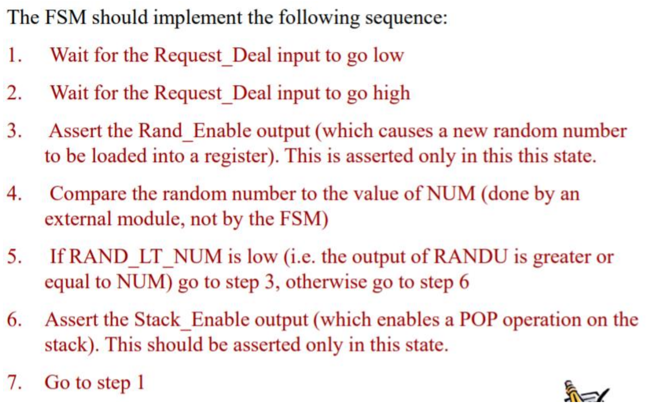
Part 2: Dealer FSM

Design Method

The objective of the dealer module is to randomize the cards that are played from the deck. It is used as a replacement for shuffling a deck; instead of shuffling, we simply generate a random number with the use of our RANDU circuit. We take the 6 most significant bits from the 32-bit random number output and use it as the address input for the stack52 circuit when we want to pop from the stack. Instead of randomizing the stack, this is an easier way to retrieve a card in a pseudo-random manner.

Hardware




The FSM should implement the following sequence:

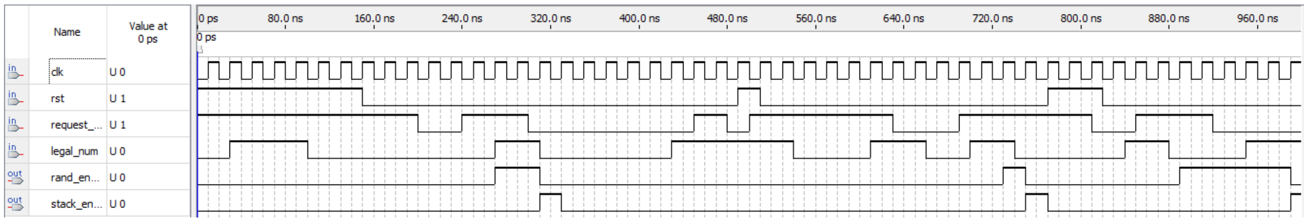
1. Wait for the Request_Deal input to go low
2. Wait for the Request_Deal input to go high
3. Assert the Rand_Enable output (which causes a new random number to be loaded into a register). This is asserted only in this this state.
4. Compare the random number to the value of NUM (done by an external module, not by the FSM)
5. If RAND_LT_NUM is low (i.e. the output of RANDU is greater or equal to NUM) go to step 3, otherwise go to step 6
6. Assert the Stack_Enable output (which enables a POP operation on the stack). This should be asserted only in this state.
7. Go to step 1

The overall functionality is represented as a Moore finite state machine with the outputs defined in the states. The module's designs are represented using a state-diagram and an excitation table. The dealer module has 4 different states with four 1-bit inputs and two 1-bit outputs. The inputs are *clk*, *rst*, *legal_num* and r*equest_deal*. The clock synchronizes everything all elements of the circuit, and the reset is an asynchronous signal to clear everything and bring the current state back to the initial state A/00 regardless whether the clock is high or low. All other operations require the clock to be high to function.

The rest of the states follow the following sequence. We have four different states starting with A/00 where both outputs are 0. We wait for the input *request_deal* to become 0 before moving onto state B/01. At state B, both outputs are 0 and we wait for *request_deal* to become 1 which represents the button being pressed. Then we move onto state C/10 where the output of *stack_enable* is 0 and *rand_enable* is 1. The purpose of *rand_enable* is to go fetch a random value from our RANDU circuit. We then compare the 6 most significant bits of the random RANDU output to the output *num* of stack52 circuit to see if the address is smaller than the number of cards left in the deck. If it is indeed smaller, we set the input *legal_num* to 1 which triggers our dealer module to change states. It will move from C/10 to state D/11. In state D, the output of *stack_enable* is 1 and the output of *rand_enable* is 0. From this point, the 6-bit number that we obtained from the previous state will be used as the address at which we will pop the card. After that operation, no matter the inputs that are given to the dealer module, the state will go back to state A/00 at the next clock cycle.

The dealer FSM has been completed using VHDL code. We defined four *std_logic* input ports and two *std_logic* output ports. Concerning the behavior of our dealer circuit, we defined the logic inside a process block in the dealer entity's architecture. We use two variables to store our state and the inputs of *request_deal* and *legal_num*. First, we check if *rst* is active since it trumps all other operations and if so, we must set the state back to A/00 and the outputs to 0. Otherwise, during the rising edge of the clock, we proceed by checking for the different conditions of the four states. The way we chose to represent the operations of each state is by using a switch case statement. We compare the two inputs and continue operating once we find a matching case, otherwise, we restart the whole process.

Simulation



The module is simulated using waveform simulation to properly identify whether the circuit functions as required. The state output is especially useful for testing purposes; it allows us to track the active states of the machines. As we can see, once *rst* goes high, the state is set back to 0, requiring the module to repeat the whole process again and all outputs are set to 0. When we have the following sequence: *request_deal* low, *request_deal* high, the output *rand_enable* is set to 1. When the previous sequence is followed by *legal_num* high, the output *stack_enable* is also set to 1 as required.

Remarks

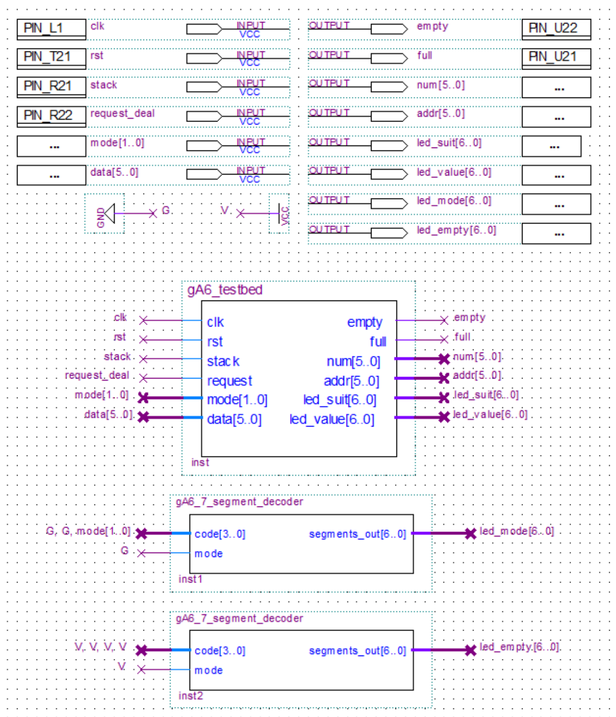
Concerning the limitations of the dealer FSM module, one issue is that our starting case must wait for *request_deal* to go low first. Technically, the module is almost always in the state B/01, although the Quartus software requires the first state to be low input, however, it makes more sense for the first state to be waiting for the *request_deal* input to go high instead of having to wait for a low input followed by a high input.

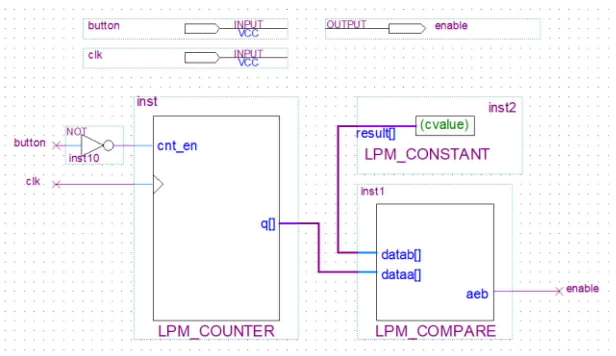
Part 3: Testbed

Design Method

The testbed circuit takes all components previously created, including those from other labs and integrates them together in a program that may be uploaded onto the FPGA boards. It allows us to properly simulate the Dealer FSM on the board. The inputs for switches and the buttons are linked together using the pin planner.

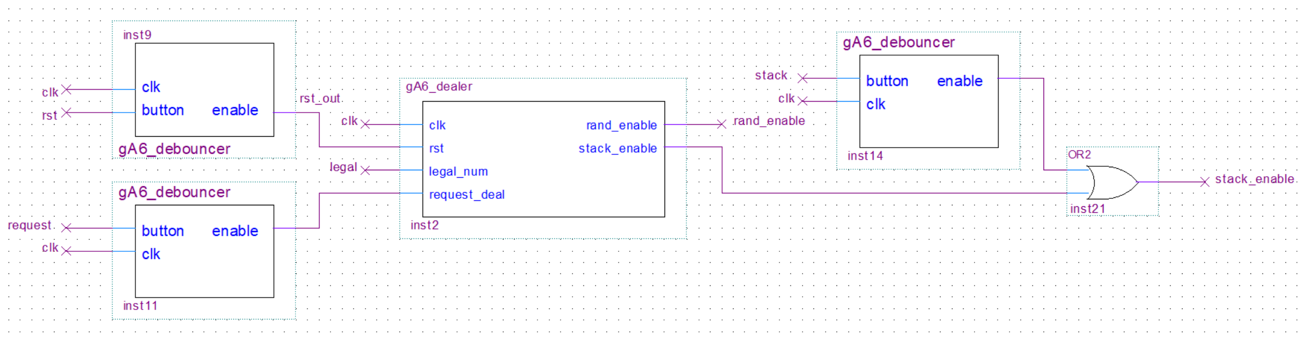
Hardware

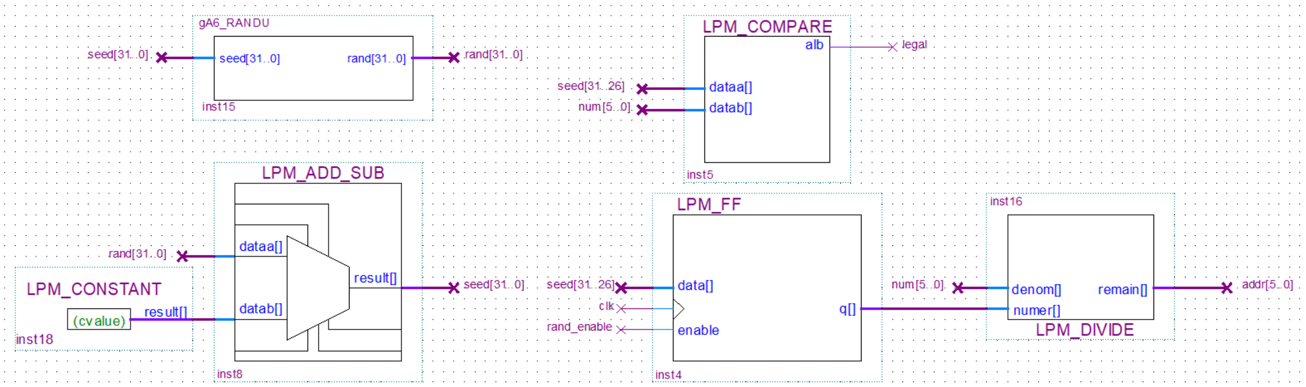




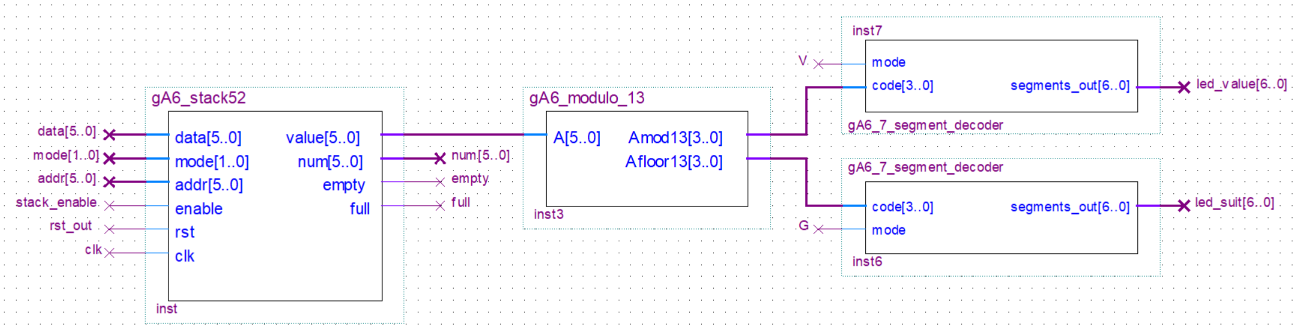
We are using three pulse generators which transform the wavy signal from the push buttons into a single enable signal for our circuit. We use one for the stack52 enable, one for the reset enable and another for the dealer request. We use the stack52 enable to override the dealer component in situations where we need to initiate the deck for example. After triggering the dealer request and going through all four states, the output stack_enable is set to high and connected to the stack52 circuit. The address of the random card is determined by our RANDU module. Finally, the popped card will be displayed on the FPGA board using the 7-segment decoder circuit.

On a step by step basis, here is how the circuit was completed. Firstly, we have the inputs from the reset, the dealer request and stack enable buttons being fed as inputs to the debouncer module which represents the single pulse generator. The outputs of the debouncers of rst and request_deal will drive the inputs of the dealer FSM. The debouncer used for the input stack will feed its output into an OR gate which can be used to override the dealer module. With a request_deal input of 1, dealer FSM will change its state. The output rand_enable will be high and the functionality of the output will be explained later in the report.



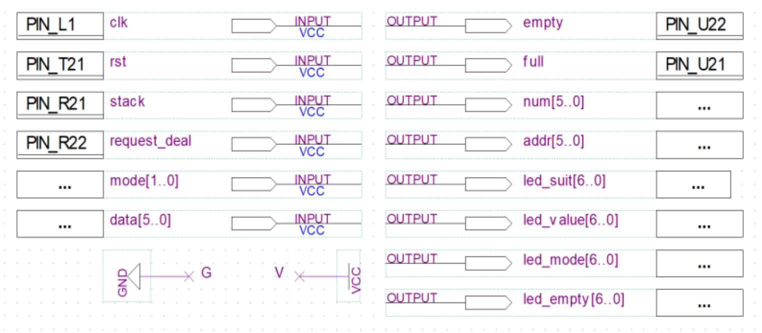


On the next part of the design, the output of *rand_enable* will allow us to fetch a random value from our RANDU module. We take the output *rand* and feed it into an LPM adder to add 1. The reason for this is to prevent the situation where *rand* equals 0, which will then create an infinite loop of 0 outputs given the formula the RANDU circuit uses. Afterwards, we use the 6 most significant bits of the output *seed* and compare it with the numbers of cards left in the stack *num* to make sure we point to a non-empty location. When the value of the input *dataa* is smaller than *datab*, the output *legal* will be high and fed into the dealer module where it will then continue to the next state. In the final state D/11, the output *stack_enable* will be high and allow the stack52 circuit to pop a card at the location *addr*.



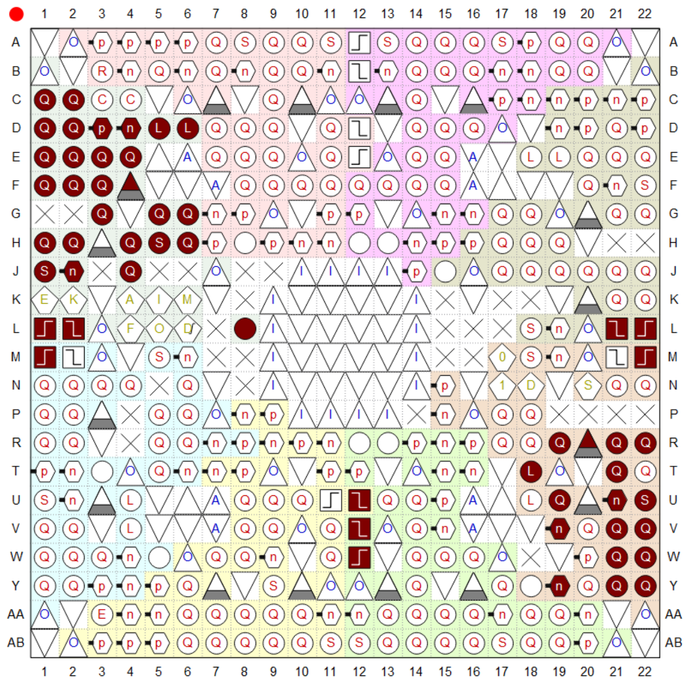
The output *value* of the stack52 circuit will then be connected to the modulo_13 circuit where it will output the value of $A\ modulo\ 13$ and $floor\left(\frac{A}{13}\right)$. The outputs will then each be connected to a 7_segment decoder for the LED display. The floor will represent the suit and the modulo will represent the face value of the card that has just been popped.

Pin Planner



PIN_L1	clk	INPUT VCC	OUTPUT	empty	PIN_U22
PIN_T21	rst	INPUT VCC	OUTPUT	full	PIN_U21
PIN_R21	stack	INPUT VCC	OUTPUT	num[5..0]	...
PIN_R22	request_deal	INPUT VCC	OUTPUT	addr[5..0]	...
...	mode[1..0]	INPUT VCC	OUTPUT	led_suit[6..0]	...
...	data[5..0]	INPUT VCC	OUTPUT	led_value[6..0]	...
			OUTPUT	led_mode[6..0]	...
			OUTPUT	led_empty[6..0]	...

We used the pin planner to map all of the inputs and outputs of our testbed circuit to the FPGA board's buttons, switches, and LEDs.
Button: rst, stack, request_deal
Switch: mode, data
Red LED: addr
Green LED: empty, full, num
Display: led_value, led_suit, led_mode, led_empty



The table on the right indicates the pin placement of every bit that is mapped to the FPGA board. All the inputs and outputs can be controlled or shown using the board.

Inputs

stack	rst	clk
0	0	0
R21	T21	L1

request_deal	mode[1..0]	
0	0	1
R22	M1	L2

data[5..0]					
0	1	2	3	4	5
L22	L21	M22	V12	W12	U12

Outputs

empty	full
0	0
U22	U21

led_value[6..0]						
0	1	2	3	4	5	6
J2	J1	H2	H1	F2	F1	E2

led_suit[6..0]						
0	1	2	3	4	5	6
E1	H6	H5	H4	G3	D2	D1

led_mode[6..0]						
0	1	2	3	4	5	6
F4	D5	D6	J4	L8	F3	D4

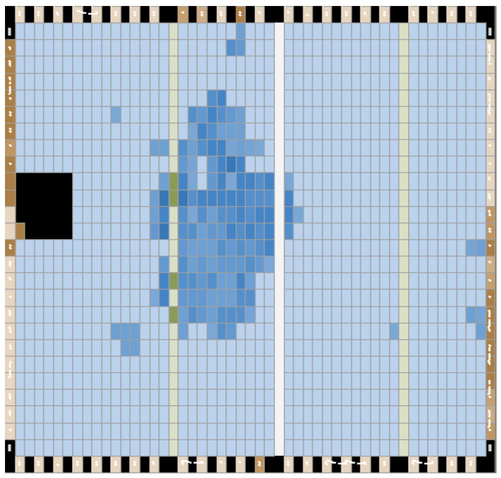
led_empty[6..0]						
0	1	2	3	4	5	6
G5	G6	C2	C1	E3	E4	D3

addr[5..0]					
0	1	2	3	4	5
R20	R19	U19	Y19	T18	V19

num[5..0]					
0	1	2	3	4	5
V22	V21	W22	W21	Y22	Y21

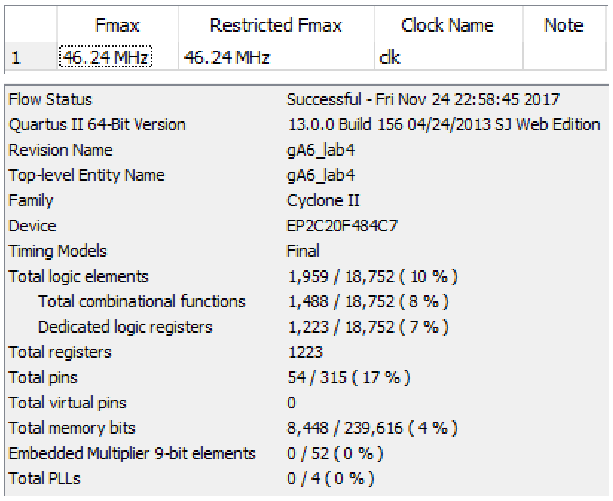
Analysis

Chip Planner



As we can see on the floorplan, the mapping of the circuit is compactly fitted onto the FPGA board. The amount of spaced used (dark blue) is quite low compared to the capacity of the board. The block utilization is very low, even though we are using over 2000 elements in our circuit. We managed to use approximately 11% of the board's blocks.

Compilation Analysis



	Fmax	Restricted Fmax	Clock Name	Note
1	46.24 MHz	46.24 MHz	clk	

Flow Status	Successful - Fri Nov 24 22:58:45 2017
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	gA6_lab4
Top-level Entity Name	gA6_lab4
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	1,959 / 18,752 (10 %)
Total combinational functions	1,488 / 18,752 (8 %)
Dedicated logic registers	1,223 / 18,752 (7 %)
Total registers	1223
Total pins	54 / 315 (17 %)
Total virtual pins	0
Total memory bits	8,448 / 239,616 (4 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

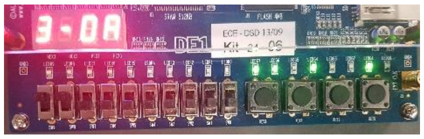
The restricted maximum frequency of our testbed circuit is $F_{max} = 46.24\ MHz$, which is quick enough considering that our circuit runs at 50 Mhz. Although, after comparing our circuit to those of other colleagues, we notice that our maximum frequency is quite average.

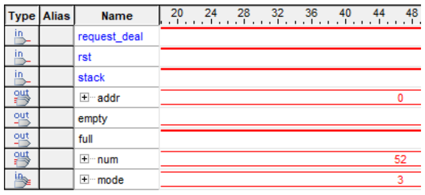
The total propagation delay of our circuit can be calculated using the inverse of the maximum frequency. Thus, we find that $t_p = \frac{1}{F_{max}} = 21.62\ ns$.

Simulation

SignalTap

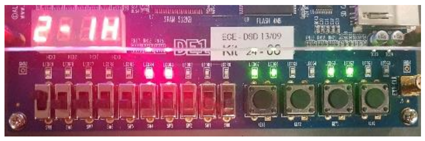
It's impossible to test every single situation using waveforms, so we also tested the circuit using SignalTap. The following images demonstrate the main tests we went through.

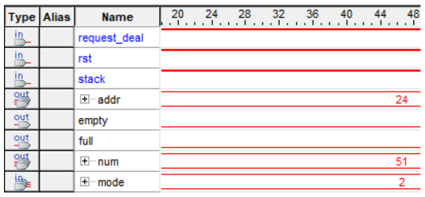




Type	Alias	Name	20 24 28 32 36 40 44 48
in		request_deal	
in		rst	
in		stack	
out		⊞ addr	0
out		empty	
out		full	
out		⊞ num	52
in		⊞ mode	3

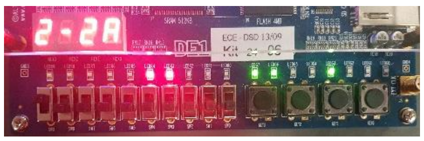
The first set of images shows the program right after initializing the stack. As we can see, the stack is full.

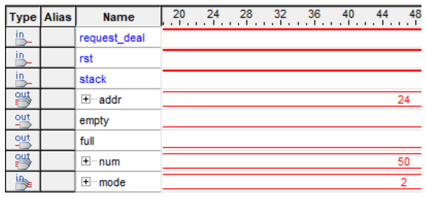




Type	Alias	Name	20 24 28 32 36 40 44 48
in		request_deal	
in		rst	
in		stack	
out		⊞ addr	24
out		empty	
out		full	
out		⊞ num	51
in		⊞ mode	2

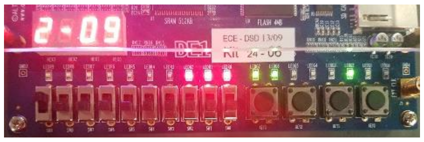
The second set shows the testbed after pressing the request button in POP mode. The card at the random address 24 was popped.

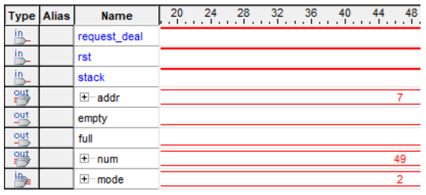




Type	Alias	Name	20 24 28 32 36 40 44 48
in		request_deal	
in		rst	
in		stack	
out		⊞ addr	24
out		empty	
out		full	
out		⊞ num	50
in		⊞ mode	2

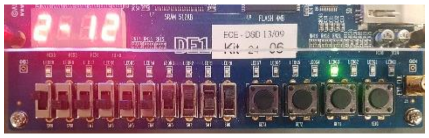
The third set shows the testbed after pressing the request button in POP mode. The card at the random address 24 was popped.

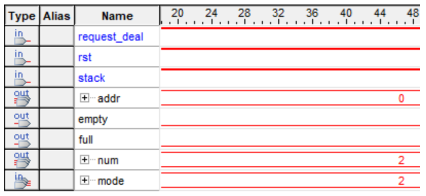




Type	Alias	Name	20 24 28 32 36 40 44 48
in		request_deal	
in		rst	
in		stack	
out		⊞ addr	7
out		empty	
out		full	
out		⊞ num	49
in		⊞ mode	2

In this set, we popped a random card at the address 7. Notice how the address is always lower than num.





Type	Alias	Name	20 24 28 32 36 40 44 48
in		request_deal	
in		rst	
in		stack	
out		⊞ addr	0
out		empty	
out		full	
out		⊞ num	2
in		⊞ mode	2

In this set, we popped a random card at the address 0. Notice how the address is always lower than num.

Remarks

The main limitation associated with our program concerns the FPGA board. Our digital circuit has many inputs and outputs, although the board only has one LED display and very limited other ways to display or control information on the board. As a result, we must neglect a lot of the valuable information.
Another issue was that we couldn't directly control the stack52 circuit. This became a problem when we needed to initialize the stack. It was impossible to do using request deal button since we'd have to go through the dealer FSM, as a result, we created a button to directly enable to stack.

References

Altera. *LPM Quick Reference Guide*, December 1996
Altera. *Development and Education Board*, 2012
Clark, J. *Lab #4 – Sequential Circuit Design*, Fall 2017, McGill University