Abbas Yadollahi – 260680343
He Qian Wang – 260688073

December 7, 2017
Team A6

Lab 5 Report: gA6 Blackjack

Introduction 2
Part 1: Computer Player FSM 3
Design Method 3
Hardware 4
Simulation 4
Results 4
Part 2: Datapath FSM 6
Design Method 6
Hardware 7
Simulation 9
Results 9
Part 3: Winner 10
Design Method 10
Hardware 10
Simulation 11
Results 11
Part 4: Blackjack 12
Design Method 12
Hardware 12
Pin Planner 16
Analysis 17
Chip Planner 17
Compilation Analysis 17
Simulations 18
Waveform 18
SignalTap 18
Results 20
References 21

INTRODUCTION

In lab 5, we are tasked with creating a complete Blackjack game by implementing all the modules we have previously made in lab 1 to 4. The game consists of dealing cards to both the player and the dealer in the hopes of reaching a total sum of 21, or the closest to it without going over. To win, your hand must be greater than the dealer's. If either party's hand total 22 or more, they have bust and automatically lose that round (the dealer has priority if both bust). If the total sum of both the player and dealer is the same, then the game is a push and neither one wins.

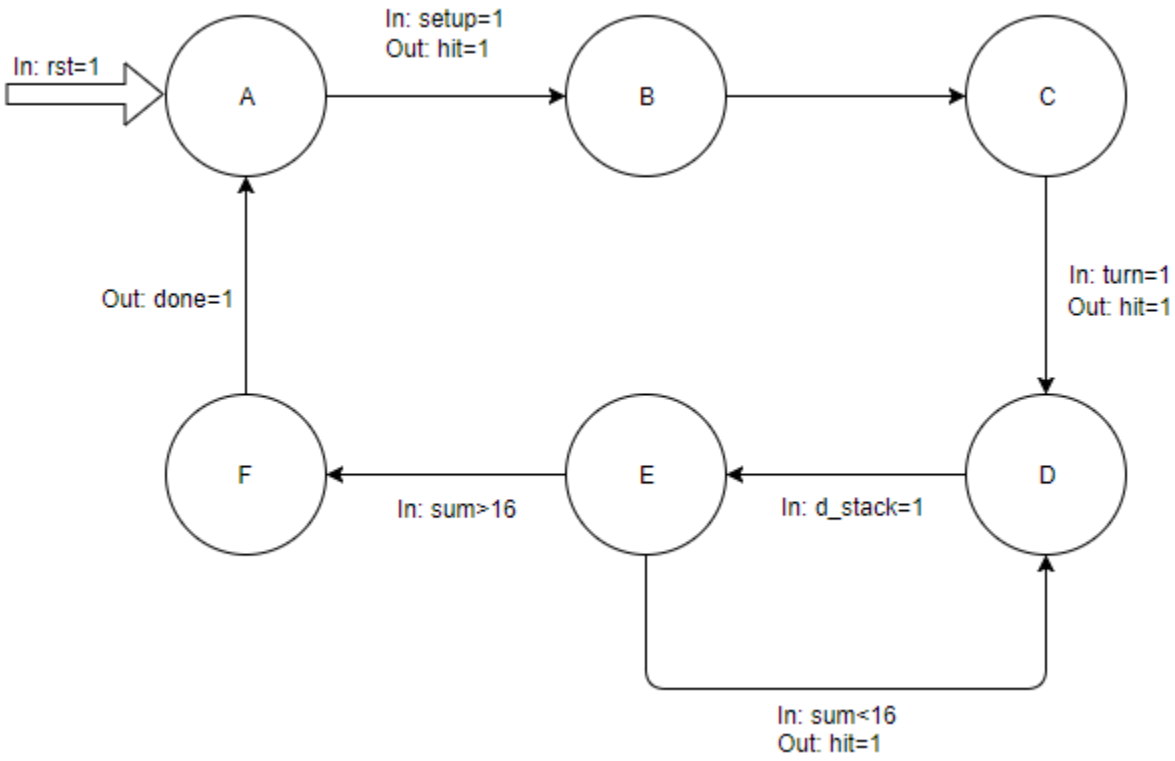
The game will begin when a human player presses the **New Game** button, after which the setup is automatically processed. It will deal two cards to the player, which can be seen by everyone, and two cards to the dealer, although only the first card is visible. Once all cards are dealt, the player has the option to add a card to their hand to increase the total sum (**Hit**) or continue the game if they are satisfied with the sum of their hand (**Stay**). If the player stays or hits to the point they bust, it becomes the dealers turn. The dealer automatically hits until its hand reaches a minimum of 17, after which it will stay.

Once a round is finished, the player needs to press the **New Game** button to start the next round. The first party to reach three wins takes over the game of Blackjack. To restart another game, the player needs to press the **Reset** button.

PART 1: COMPUTER PLAYER FSM

Design Method

The first part of the lab requires us to build a finite state machine for the computer player. In blackjack, when the player ends his turn, it becomes the dealer's turn. The dealer's turn does not require the input of the player, its actions and decisions are completed by following a set of rules. The dealer will receive his cards and it must add cards to its pile until the value of his hand reaches a minimum of 17. The computer player FSM is only responsible for adding cards to the hand of the dealer; the FSM is not responsible to indicate the winner of the round or to count any wins.

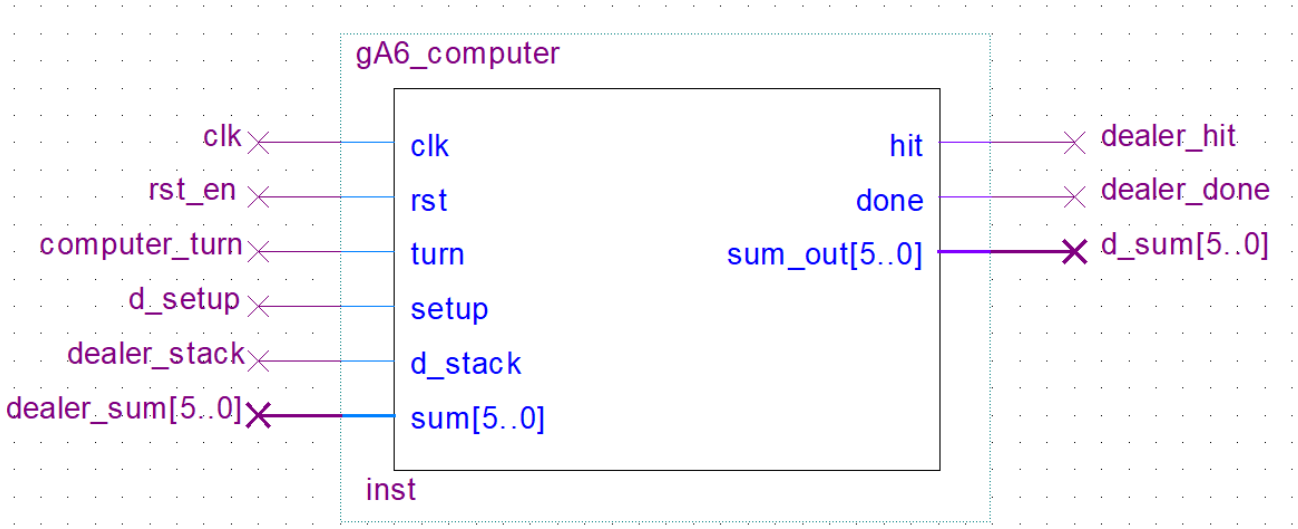


The FSM has been designed with the following state diagram. The FSM is synchronized with the rest of the circuit with the same clock input (*clk*) and shares the same reset input (*rst*) as the rest of the circuit. Our hardware is based on the following design. We decided to use 6 inputs and 3 outputs for our **Mealy** finite state machine. On this state diagram, unless specified, it is assumed that the input and outputs are 0. With the exception of the clk input and the rst input, all of them are assumed to be 0 unless specified. When the input rst is 1, it will bring the FSM back to state A, no matter what the other inputs are set to. When the clk is low, it will force the module to stay in the present state.

The computer player FSM will only be dealt one card at the start of a new game since the player can only see the top card of the dealer's hand. After the player ends his turn, the computer FSM will deal itself cards until the sum of his cards reaches a value above 16. In the next state, it stops dealing itself cards and the output dealer_done of the computer FSM becomes high to allow the rest of the program to continue with the game.

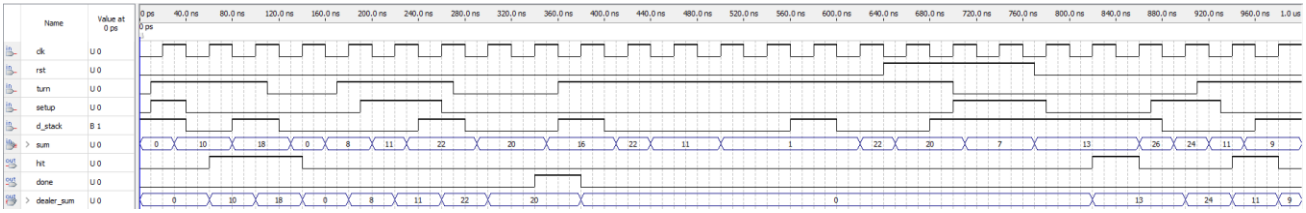
Hardware

The computer FSM module is built using VHDL. Our entity has five 1-bit inputs, one 6-bit input, two 1-bit outputs and one 6-bit output. The clock (*clk*) and reset (*rst*) inputs are synchronized with the rest of the modules. The other inputs allow the finite state machine to change its states.



The computer FSM starts at state A; it waits for the input setup=1 to move to the next state. At the same time, a card is dealt to the dealer. At state B, the dealer FSM is waiting for the card to be properly added to the dealer pile. It takes more than one clock cycle for the card to be added. The FSM moves to state C at the next clock cycle. In state C, we are waiting for the player's turn to be finished. When the player is done his turn, the *turn* input will be high which means that the dealer can move forward with dealing another card to the computer pile. When the *d_stack* input is enabled, we will move to state E to avoid race conditions. At state E, the FSM will check whether the value of the dealer's hand is larger than 16. If it is smaller than 16, the dealer goes back to state D and deals itself another card. Otherwise, we will move to state F; we set the output *done* to 1 so that the rest of the modules know that the dealer is done his turn and that the program may move on. At the next clock cycle, the module moves back to state A and waits for the *setup* input to be 1 again.

Simulation



The functionality of the FSM is tested with the use of a simulation waveform. The *clk* has a period of 40ns and we manually set the inputs to be certain specific values. A lot of emphasis has been placed on testing the race conditions. For example, popping a card from the stack may take more than one clock cycle. We must account for the clock cycles that the operation takes before changing states. We also simulated the design to validate the state change when the sum of the hand revolves around 15 to 18.

Results

Overall, the module functions as expected. The errors were located with the help of the waveform and allowed us to make sure that all edge cases were well tested. From the waveform, we can see that our circuit requires a few clock cycles before a card can be popped and the finite state machine must wait for the pop operation

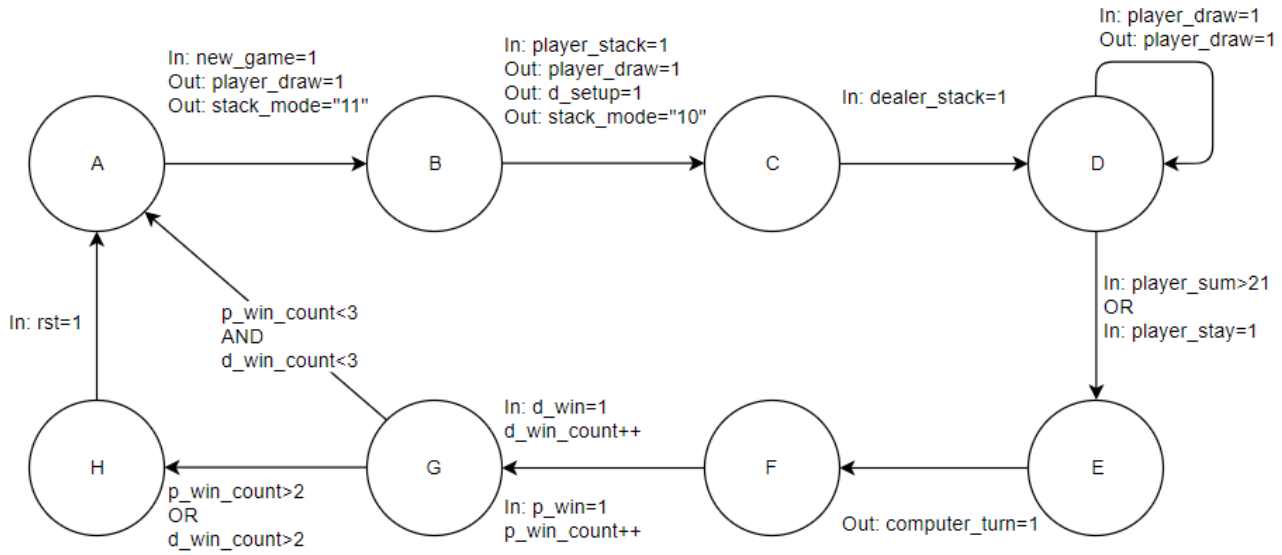
to be finished before the state may change. The limitation is that our circuit may not be completing everything in a simultaneous fashion which means we might be wasting a few clock cycles. Our operating frequency may be a bit higher if we manage to shave off a few clock cycles. However, the advantage is that we are using less memory and fewer states to run the module. Also, requiring an input to confirm the completion of the pop operation assures that there are no mistakes during state changes. The advantage in this case of using a Mealy machine is that we are saving up on the number of states and clocking our device slightly faster.

PART 2: DATAPATH FSM

Design Method

The objective of the datapath FSM is to control the flow of data in the blackjack game. It controls the player's turn and it dictates the flow of the game. It is the controller for the entire system including the player's turns, the number of games played, the score count, starting new games and the rest of the machine's states. It is responsible for deciding when to deal the cards and taking care of the player's inputs. The controller will also decide when the computer enters its turn (by giving the control over to the computer FSM) and when to start the next game after the current round is completed.

The design is a synchronous module that changes at the common clock cycle (*clk*) and it shares the same reset (*rst*) as the rest of the design. This finite state machine is a Mealy type as well since we are heavily reliant on the use of inputs to set outputs. The advantage, once again, of using a Mealy machine is that we are saving up on the number of states and clocking our device slightly faster.



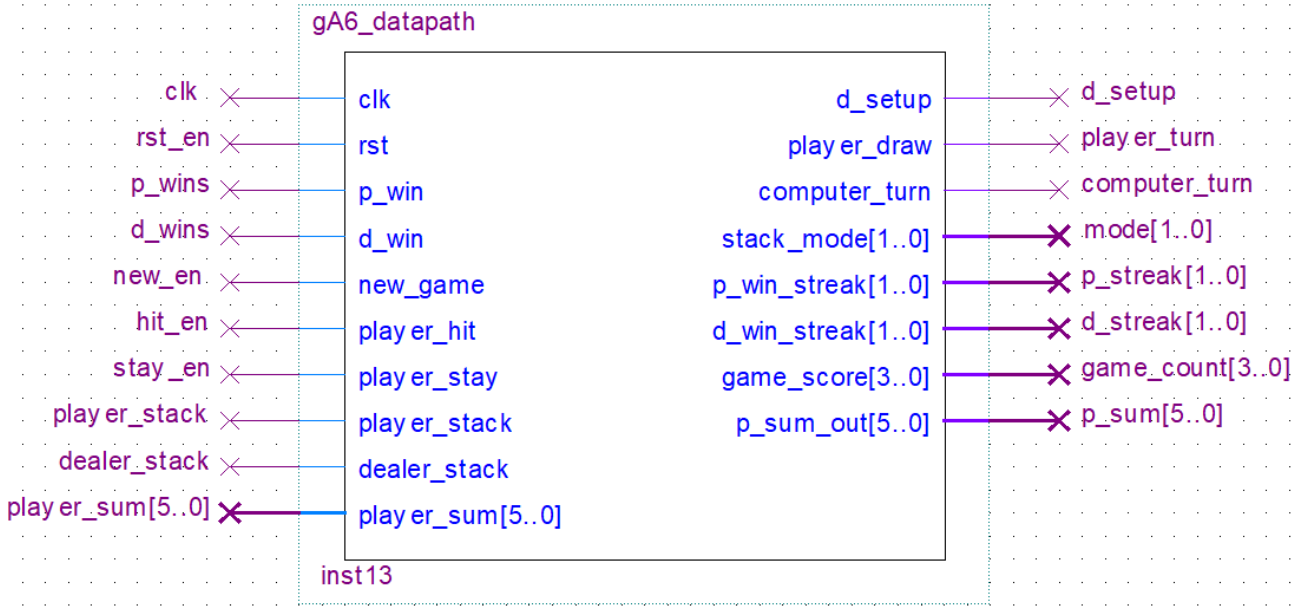
Our device is built according to the above **Mealy** state diagram. The role of the datapath FSM is to control the turn of the player, initialize the turn of the computer and at the same time, keep track of the winners and losers. On this state diagram, unless specified, the output *stack_mode* has the value of "10", which tells our stack52 module that it is in POP mode operation. Only in the transition from state A to state B does our FSM set the output *stack_mode* to "11". In this situation, we are initializing the stack of cards before starting the new round.

In this module, we will assume that the inputs and outputs are 0. With the exception of the *clk* input and the *rst* input, all of them are assumed to be 0 unless specified. The *rst* input being 1 will bring the FSM back to state A without regards to other inputs or outputs. When the *clk* is low, it will force the module to stay in its current state. We start with an input that tells us that the user wishes to start a game; the deck is initialized once *new_game* is high. The player is dealt two cards and the dealer is dealt one card. Our reasoning is that the player can only see the top card of the dealer's pile, as a result, we only need to deal a single card. The system then waits for the player to draw cards until he either busts (*player_sum*>21) or he decides to stay (*player_stay*=1) with his current hand. Once either of the latter conditions have been met, the output *computer_turn* will be high which will give the computer player FSM control over the program. It then initializes and goes through the dealer's turn before finding out who the winner of the round is and relinquishing its control. Depending on which party won, the Datapath module will increment the number of

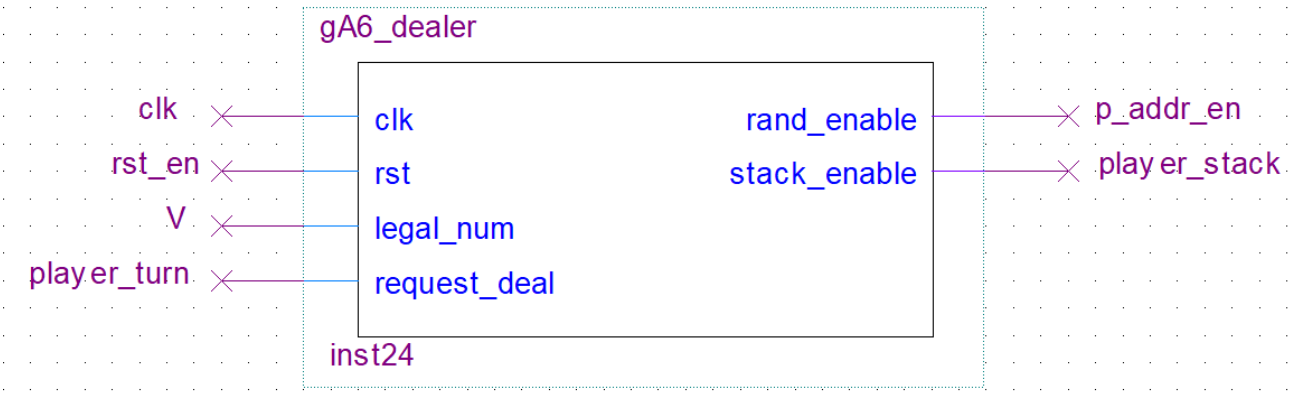
wins for the player or the dealer. Since the blackjack game consists of a first to three wins, if one of the two participants of the game manages to win three games, the datapath FSM will indicate the score and who won. It should now be in the final state H, where it will wait for the *rst* signal to be high before being able to return to its original state A.

Hardware

The datapath FSM has been designed using VHDL. Our entity is composed of nine 1-bit inputs, one 6-bit input, three 1-bit outputs, three 2-bit outputs, one 4-bit output and one 6-bit output. The clock (*clk*) and reset (*rst*) inputs are shared with the rest of the circuit. The architecture of the program is designed using a process with a sensitivity list including all the inputs. We employ several variables that allow us to keep track of the states and the different counters of the program in memory. The circuit states are set on the clock's rising edge, except for the *rst* input which doesn't require the circuit to be active to change states. When the reset button is pressed, it will bring the state back to A and clear all values in memory.



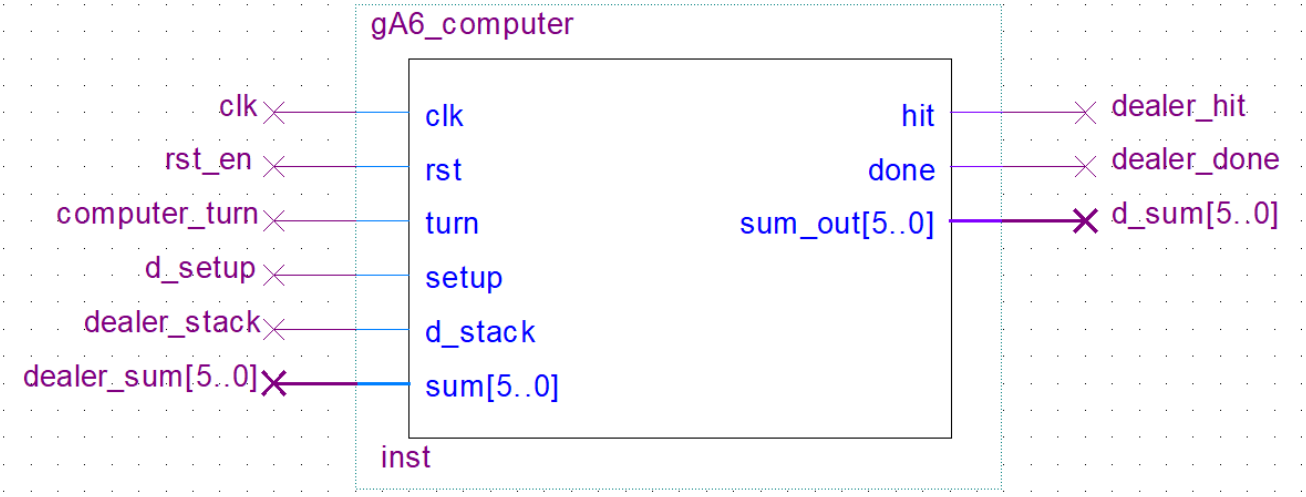
We decided to set the output values at every state to avoid unwanted signals from repeating from previous states. Within the process, our module has 8 different states of operation. The flow of operation is set with the usage of conditional if and else statements. At state A, the FSM will wait for the user's input, which is the *new_game* button. When that signal is active, we will set the *player_draw* output to be 1 and we move to the next state B. To avoid race conditions, we decided to add an enable signal for this state before it changes to



state C; the enable signal is the *player_stack* signal. The *player_draw* output is connected to the dealer FSM, which itself takes about four clock cycles before it may enable a stack operation at a valid address, after which it will enable the next operation for the datapath FSM. This state is simply present to allow us to wait out the four clock cycles so that we do not enable two stacks signals at the same time and only pop one card. When the player has drawn all his cards, the dealer's setup signal is active and we move onto state C.

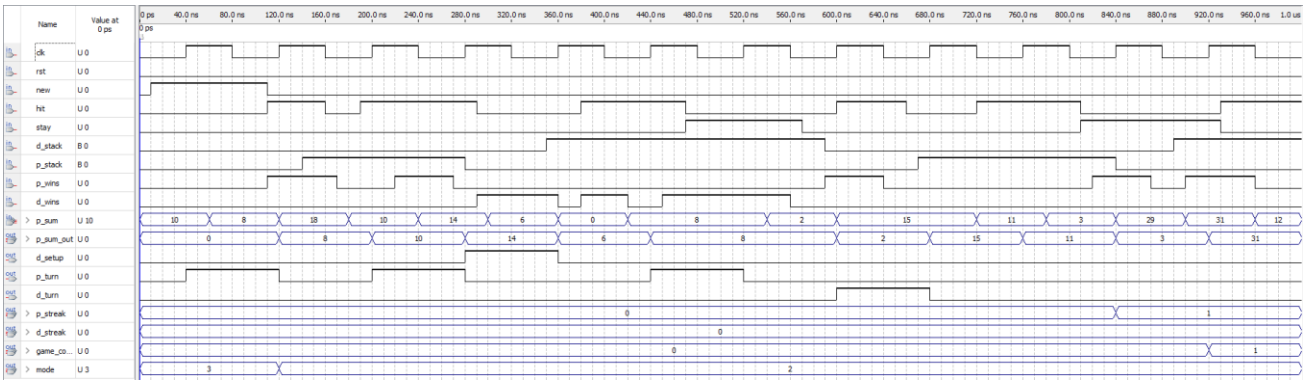
Concerning the computer's turn, it will deal a card to the computer player using by going through computer FSM states. Once again, we wait for an enable signal *dealer_stack* to avoid race conditions. The output will be connected to a dealer FSM which will signal when to deal a card by enabling the POP operation on the stack52 module at a valid address. Once the stack has been enabled, the datapath module will be allowed to move on to the next state.

At state D, it's the player's turn; we wait for the signals from the human player to either draw (*hit*=1) or stay (*stay*=1) on his hand. If the player decides to draw, we loop back to state D and wait for another *hit* or *stay* signal, but only if we didn't bust by going above 21. If the player's hand busts or if the player decided to stay with his current hand, we will move to state E. In state E, it is the computer's turn; the output signal is *computer_turn* is high and is connected to the computer FSM's *turn* input. This FSM will go over the dealer's turn and, once finished dealing its cards, will give the signal to the datapath FSM to change to state F.



In state F, we are looking for the winner of the round. We use the winner module to figure out which party won the current round. The winner module will be explained in more details further in the report. If we have a winner for the round, we increment the internal counter (a variable in the process) of the winner by 1. Afterwards, we move to state G; in state G, we simply check whether the player or the dealer has reached a win count of 3. If none of them won three times, we will go back to state A and keep playing. If a player won 3 times, we will go to state H and announce the winner on the LED display. At that point, the FSM will wait for a reset signal before changing back to state A for a fresh new start.

Simulation



The datapath module is simulated using a waveform. Like the previous FSM, computer player, we checked most of the edge cases and we resolved most of the race conditions using the clock cycles from the FSM. We check most of the edge cases such as what happens when the two players have the same hand value. The circuit module functions as expected. To track the current states, we created an extra output *state_out* on our FSM which helped in debugging. From there, we managed to follow the status of the current state of the waveform graph. We also used the waveforms to track how many clock cycles for the delay between the change of states.

Results

The limitations of the circuit are that instead of having everything happen at the same time, we use a few clock cycles. Another limitation of this module is that we need external modules to compute the results of our outputs. By doing this, we put more reliance on the rest of the circuit to perform most of the work.

On the other side, for the advantages, we have only 8 states for this machine. Using a Mealy state machine considerably reduces the numbers of states in our case and our operating frequency should be quite high. The amount of memory used by our circuit should also be lower than if we chose to use a Moore state machine. Most of the outputs are heavily reliant on inputs. The reliance on inputs is what allows us to avoid race conditions. By using Mealy, we reduce considerably the number of states and at the same, we reduce the amount of delay between the states.

PART 3: WINNER

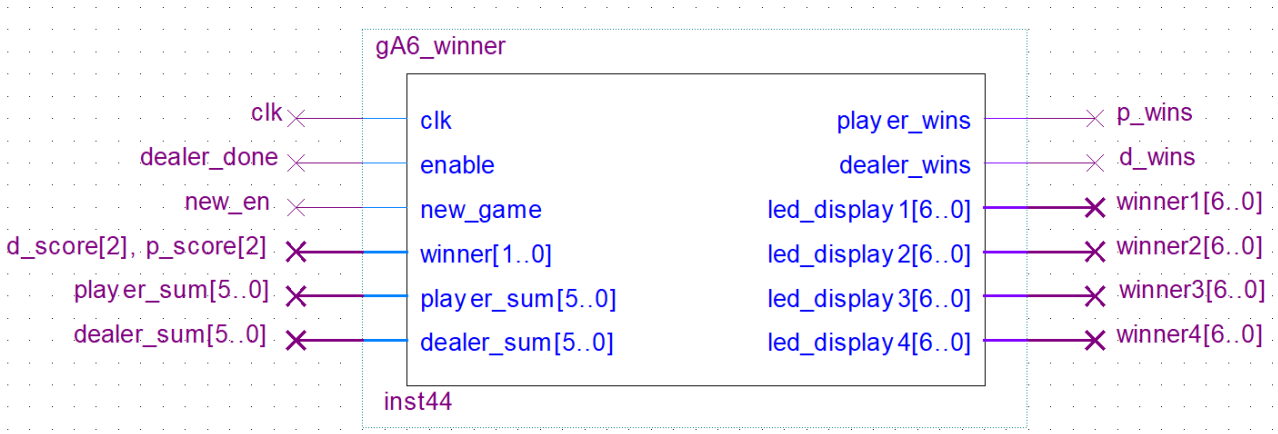
Design Method

A crucial part of the blackjack game is to determine who is the winner of a given round. Seeing as it requires quite a few operations, we decided to create a module to take care of the finding the winner. Once again, we created the digital circuit with the help of process blocks in VHDL seeing as it would be the most effective and simple way.

The principle behind the winner module is fairly simple. It takes as input the total sum of the hands of both the player and the dealer and compares them to find the better hand. It follows the same rules as previously mentioned, where the party with the highest hand, while still totaling 21 or less, wins the round. The first step is to check if either player's hands bust, starting with the human player (since the dealer has priority on winning when both bust). If a party busts, the opponent automatically wins the round. If neither players hand sum to over 21, then we look for the highest hand. The party with the highest total wins the round. In the situation where both hands equal the same, we have a tie (push), and neither player wins.

Once we sorted out the hands, we output a high signal for the winning party and an encoded signal for the 7-segment LED display that will either spell out "WIN" (if the player wins), "LOSE" (if the player loses) or "PUSH" (if it's a tie).

Hardware

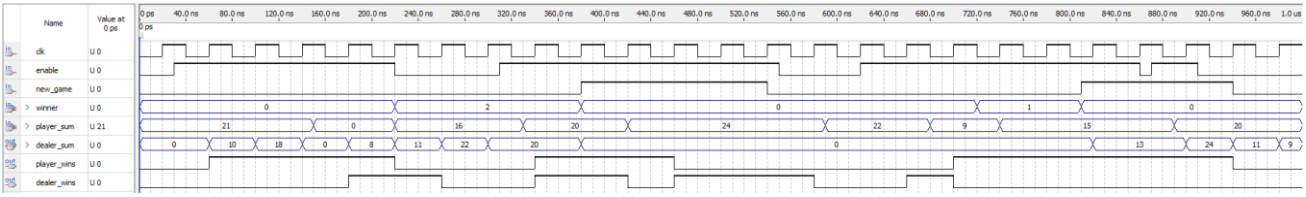


The winner module has three 1-bit inputs, one 2-bit input, two 6-bit inputs, two 1-bit outputs and four 7-bit outputs. The module is clocked with the *clk* input since it needs to be synchronized with the outputs of other components in the blackjack circuit. The inputs *clk* and *enable* are used to determine when the module should be active as it is only required a single time per round.

If *enable* is high and we're on a rising edge, the module will become active and will determine the winner by comparing the hands of the player and the dealer. The total of both party's hands is given through the 6-bit inputs *player_sum* and *dealer_sum*. Once it has found the winner of the current round, it outputs a high signal for the winning party, being *player_wins* if the player wins and *dealer_wins* if the dealer wins, and a low signal for the other. If it's a tie, both outputs will be high. To help visualize the result of the round, we added four outputs to display if the player won, lost or tied. Each *led_display* output represents one 7_segment led slot.

If enable is low, but the *clk* is on a rising edge, the winner outputs will both be low and the led display outputs will give the encoded signal for a dash since the round hasn't finished yet. At the end of a round, the display will only reset back to dashes once the player has given a signal to start a new game.

Simulation



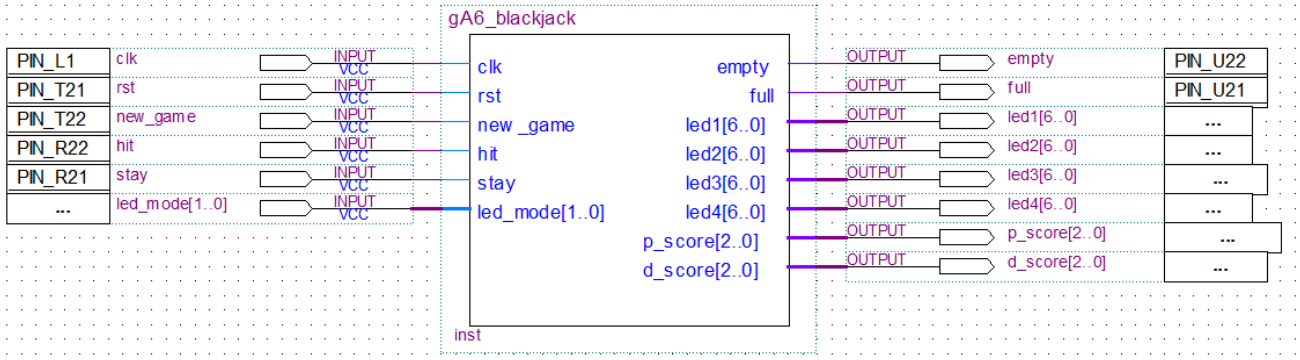
We test the winner module using functional simulations and waveforms. We made sure to test each scenario (win, lose, push) at least once to confirm that the module can properly determine which party wins the round. The concerning edge cases were when both the player and the dealer bust or had the same total sum. Although, after testing our module we were able to validate and confirm that our circuit functions as expected.

Results

This module did not present any major limitations as we only created it to facilitate how we determine the winner of a round. Although by doing so, we added an extra synchronous step that our blackjack game must go through before completing an entire round. As a result, we added a couple clock cycles of delay to the speed of the program.

PART 4: BLACKJACK

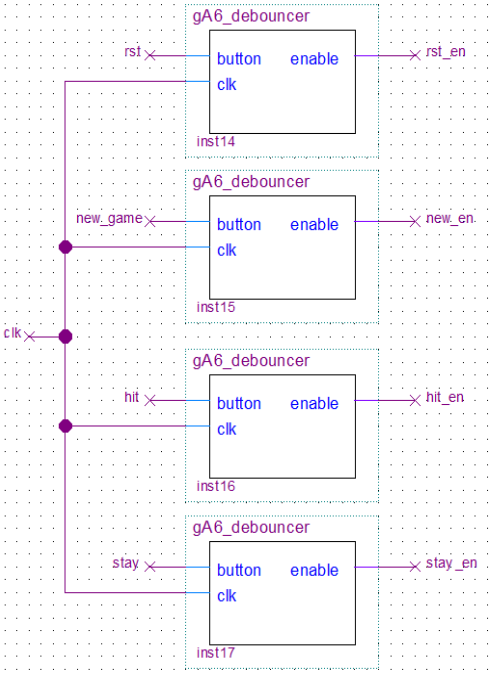
Design Method

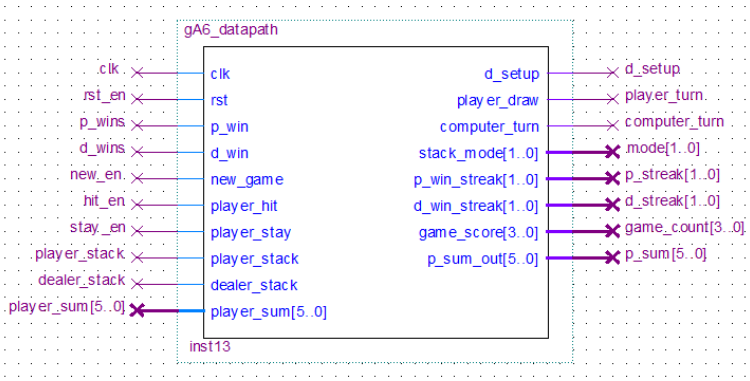


For the overall digital system that implements blackjack, we are using all other modules previously created, including the two FSMs *dealer* and *rules* and circuits such as stack52, modulo_13, 7-segment decoder and debouncer. We decided to use a different type of random number generator for our address in this lab to increase the chances that the sequence of cards popped differs from game to game. Both FSMs from the previous parts of the lab, being *datapath* and *computer*, are connected along with the rest of the modules from previous labs. The only new module that we created for this part is the winner module. It is written in VHDL and its job is to simply check who won the round based on the total sums of the player and dealer and outputs the winning party.

Since the many of the modules will depend on the outputs of other modules within the circuit, we decided to clock many of the elements and add enable signals to most critical modules to assure a smooth flow through the game.

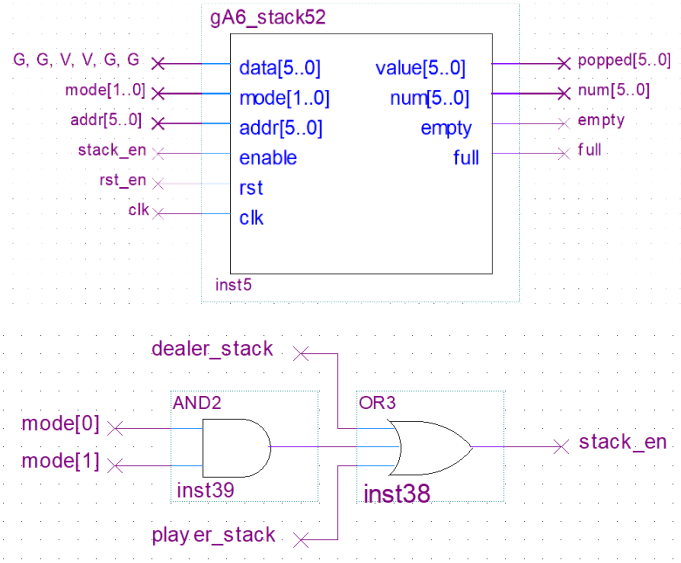
Hardware





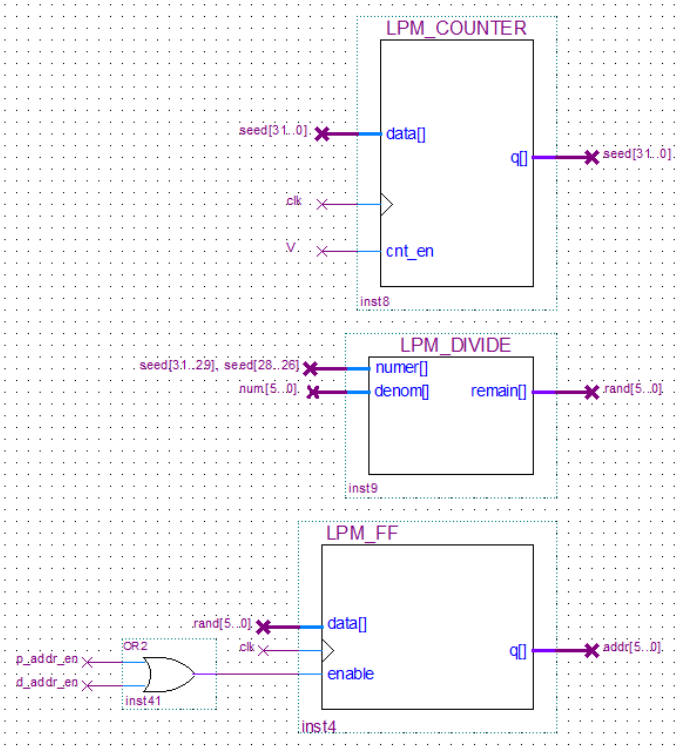
The circuit uses four signal pulse generators. We have four input buttons so four generators are necessary to cancel out the noise. The *rst* (*rst_en*) signal will be connected to every module and when high, all FSMs will reset to their initial state and the program resets. The *new_game* (*new_en*) signal is used only to initialize a new round of blackjack and the *hit* (*hit_en*) and *stay* (*stay_en*) inputs are only used during the human player's turn.

Everything is clocked using *clk* in this synchronous machine. The *new_en*, *hit_en,* and *stay_en* outputs will be connected to the datapath FSM, whereas the *rst_en* output is connected to almost all modules that implement some type of internal memory. The datapath FSM will go through each state depending on the inputs it receives and its current state. As discussed earlier in Part 2: Datapath FSM, the module will setup both parties and give access to them to play their turn through the *d_setup*, *player_draw* and *computer_turn* outputs. At the same time, it will keep track of all the score and game counters. All human player inputs will go through the datapath FSM which will take care of the given operation.

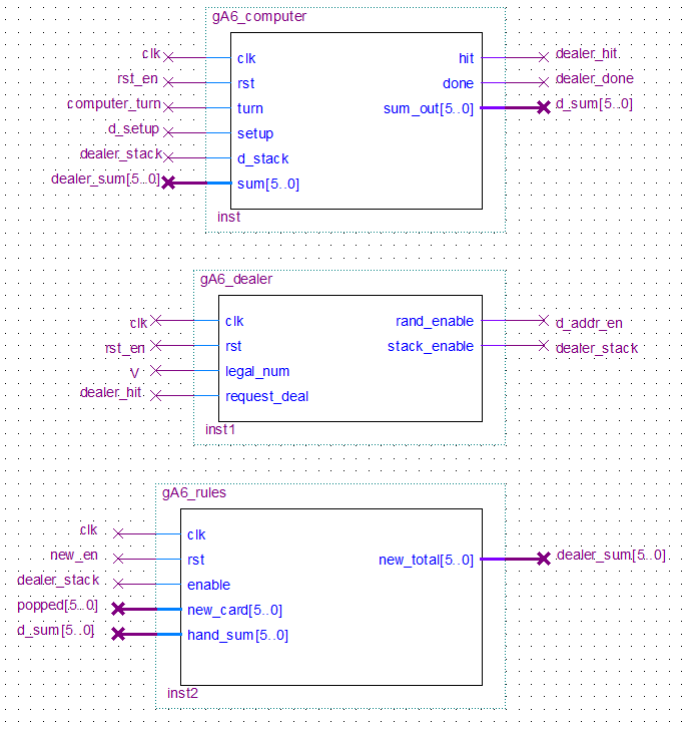


The stack52 module is the heart of our blackjack game seeing as it is the holder of all cards and performs all the dealing operations, although its tasks are simple. It will only operate in two modes, dictated by the *mode* input, being INIT or POP. If the stack is to operate, the inputs *enable* or *rst* will need to be high; we started a new game (INIT), the player or dealer is hitting (POP) or the player signaled for a reset. After a POP operation, we send the card to the *value* output.

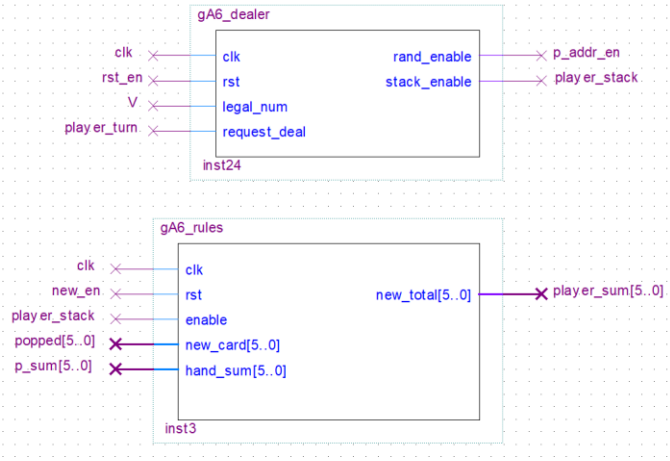
The *enable* input of the stack52 uses the *stack_en* signal which is decided by three other signals: *mode*, *player_stack* and *dealer_stack*. If mode is in INIT mode ("11") or either of the two other signals are high, then *stack_en* will also be high.



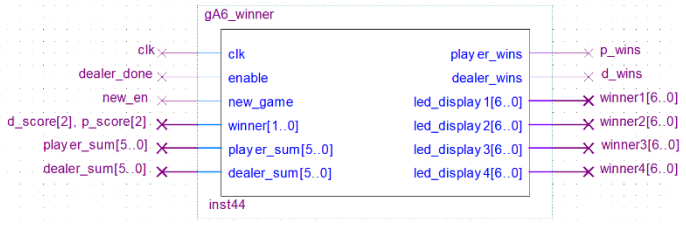
For the blackjack circuit, we decided to use a different random number generator for the address since the RANDU circuit we previously created had a few issues and was problematic on the programs start up. In the figure to the left, we simply use an LPM counter, and LPM divide and an LPM Flipflop. The process is simple, we use the clock as an input for the counter, hence at every rising edge, we increment by 1. We use the first and last three bits of the counter's output *seed* as the *num* input for our divide module. The *denom* input will be the number of cards currently present in the stack. We then take the remainder of the division, which will always be lower than the number of cards left, and use it for the *addr* input of the stack. Before doing the latter, we store it inside the flipflop and only release the value when the signals *p_addr_en* or *d_addr_en*, which represent when the stack is about to pop, are high. The output *addr* of the flipflop is then fed into the stack52 module.



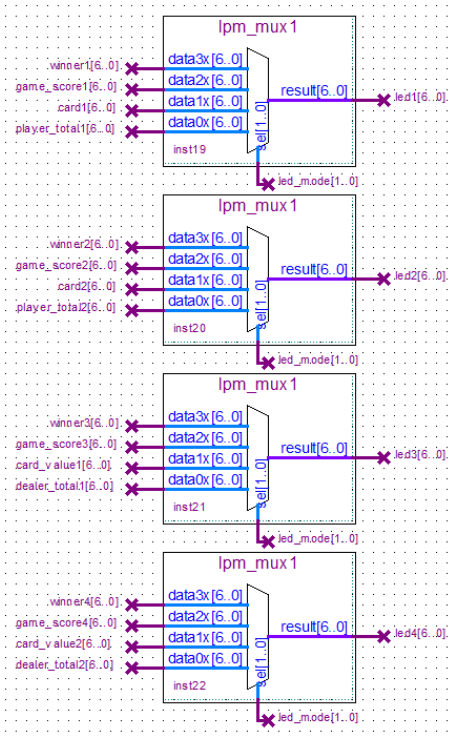
All operations related to the dealer are taken care of using the three modules shown in the figure to the left, including a computer player FSM, a dealer FSM and rules. The first step is to either receive a high *setup* signal from the datapath FSM which will begin the setup or to receive a high *turn* signal that will allow the dealer to hit until he finishes his turn. In either case, the computer module will go through its different states until it outputs a high signal for *dealer_hit* which will activate the *request_deal* input of the dealer FSM and begin the popping process. Once the dealer module receives a high signal for the *legal_num* input (as discussed above, the address will always be valid thanks to the modulo), it will give a signal to pop a random card from the stack52 module. The popped card will then be used as an input for *new_card* on the rules circuit with the sum of the dealer's current hand as the input for *hand_sum*. Finally, the rules module will output the dealer's new total sum.

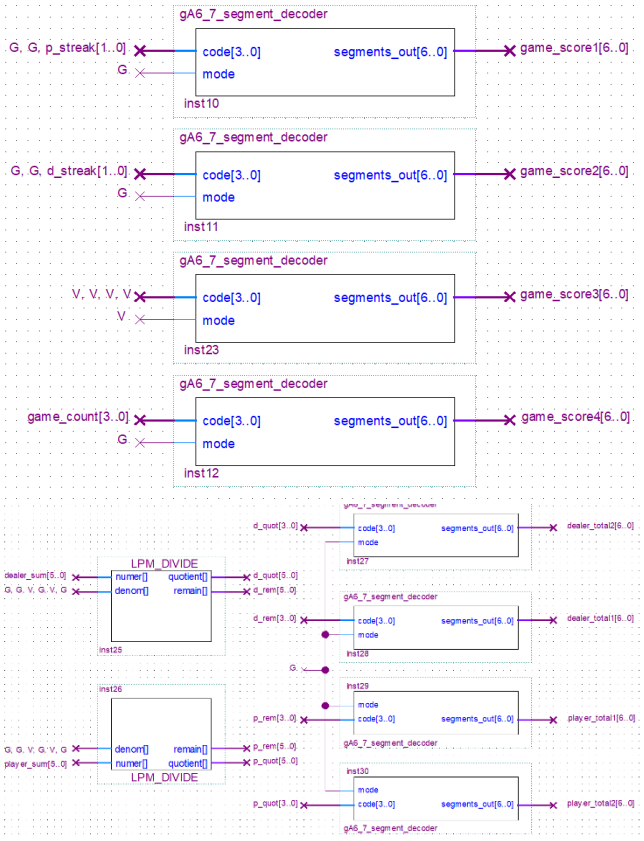


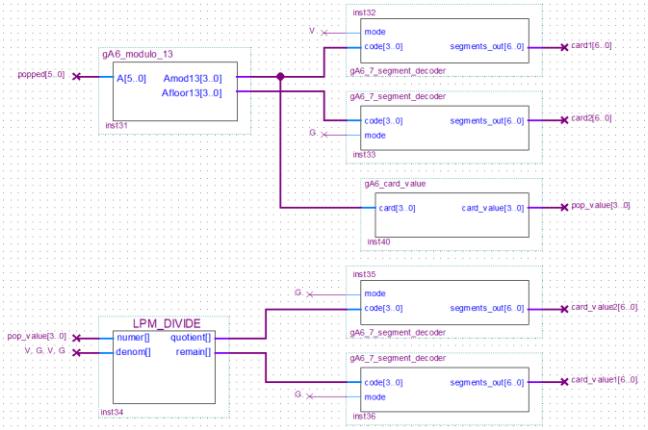
All operations related to the player are taken care of using the two modules shown in the figure to the left, including a dealer FSM and rules. The first step is to receive a high *request_deal* signal from the datapath FSM. In the same manner as the dealer's structure, it will output a high signal for *stack_enable* after going through the states and receiving the proper inputs. After popping a card from the stack, it will find the new total sum using the rules module, where the *new_card* and *hand_sum* inputs are the popped card and the total sum of the player's current hand, respectively. Finally, it will output the player's new total sum.



Once both parties have finished playing, we send the total of their hands to the winner module which will determine the winner of the round. The outputs *player_wins* or *dealer_wins* will be high if the player wins or if the dealer wins, respectively, or both in case of a tie. The outputs *led_display* will show the result as an encoded signal for the 7-segment LED displays.

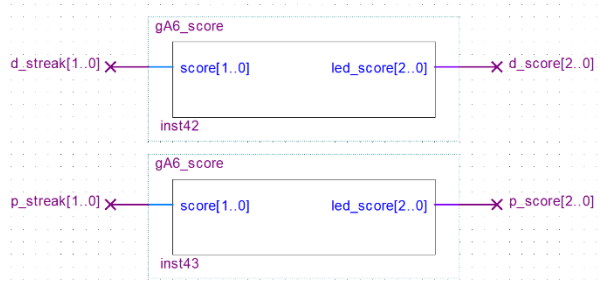






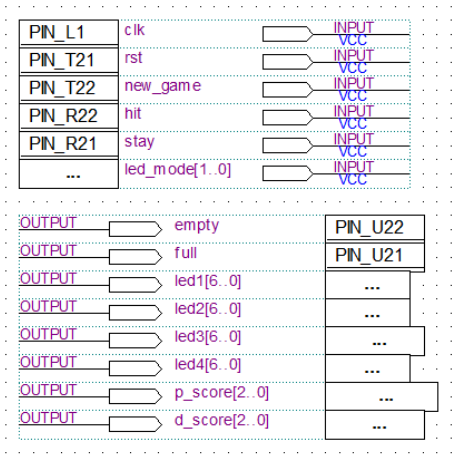
We decided to display all the required information on the LED displays, and switch between them using two switches (giving us four modes) linked to the *led_mode* input. To choose which information is being displayed, we used a 4-1 LPM multiplexer, where the first option shows the total sum of the player and the dealer, the second shows the last card to have been popped and its face value, the third shows the number of games played and the score of the player and dealer and the fourth shows if the result of the round played.

To gather all the required information, we employ a card value circuit (outputs the face value of a given card), many instances of LPM divide, modulo_13, 7-segment decoder circuits and one winner module.



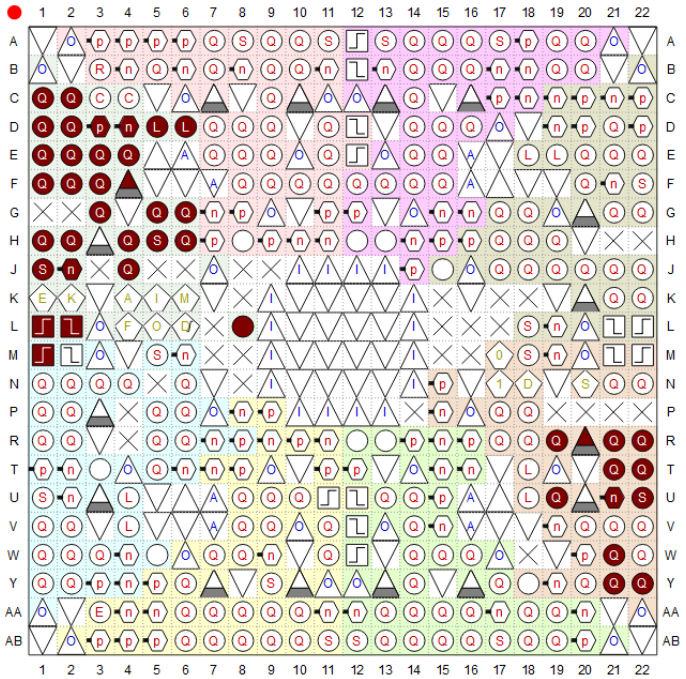
Finally, the last section of the blackjack circuit is two score modules, which simply takes the player's and dealer's score and turn it into a 3-bit output which we can use to light up the LEDs on the board. Each win represents one lit LED.

Pin Planner



We used the pin planner to map all the inputs and outputs of our blackjack circuit to the FPGA board's buttons, switches and LEDs.

Button: new_game, rst, hit, stay
Switch: led_mode
Red LED: d_score
Green LED: p_score, empty, full
Display: led1, led2, led3, led4



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			
clk	led_mode[1..0]		
0	0	1	
L1	M1	L2	
new_game	rst	hit	stay
0	0	0	0
T22	T21	R22	R21

Outputs	
empty	full
0	0
U22	U21

led1[6..0]						
0	1	2	3	4	5	6
J2	J1	H2	H1	F2	F1	E2

led2[6..0]						
0	1	2	3	4	5	6
E1	H6	H5	H4	G3	D2	D1

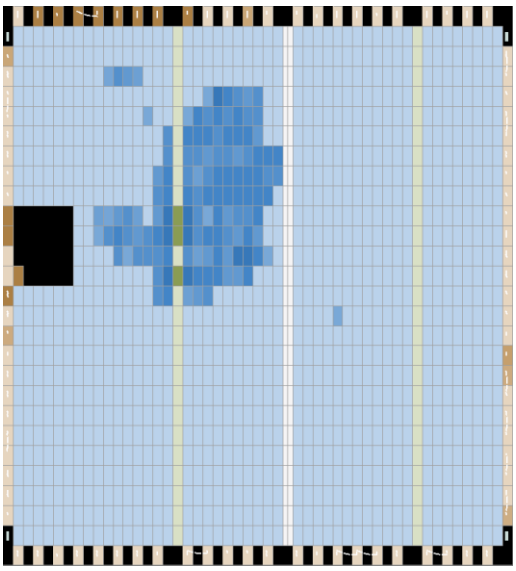
led3[6..0]						
0	1	2	3	4	5	6
F4	D5	D6	J4	L8	F3	D4

led4[6..0]						
0	1	2	3	4	5	6
G5	G6	C2	C1	E3	E4	D3

p_score[2..0]			d_score[2..0]		
0	1	2	0	1	2
Y21	Y22	W21	R20	R19	U19

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 just under 3000 elements in our circuit. We managed to use approximately 16% of the board's blocks.

Compilation Analysis

	Fmax	Restricted Fmax	Clock Name	Note
1	46.89 MHz	46.89 MHz	clk	

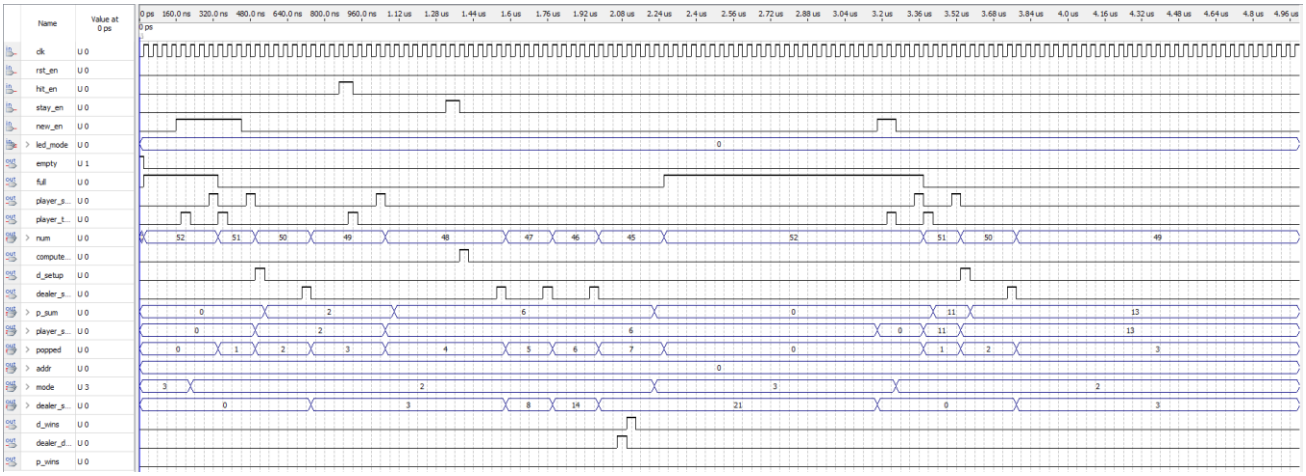
Flow Status	Successful - Mon Dec 04 18:08:59 2017
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	gA6_lab5
Top-level Entity Name	gA6_lab5
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	2,963 / 18,752 (16 %)
Total combinational functions	2,097 / 18,752 (11 %)
Dedicated logic registers	2,013 / 18,752 (11 %)
Total registers	2013
Total pins	43 / 315 (14 %)
Total virtual pins	0
Total memory bits	14,976 / 239,616 (6 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

The frequency of our digital circuit if $F_{max} = 46.89\ MHz$, which is quick enough considering that our circuit runs at 50 Mhz. After comparing our circuit to those of other colleagues, we notice that our maximum frequency is quite high.

The total propagation delay of our circuit can be calculated using the inverse of the maximum frequency. Thus, we find that $t_p = \frac{1}{F_{max}} = 21.33\ ns$.

Simulations

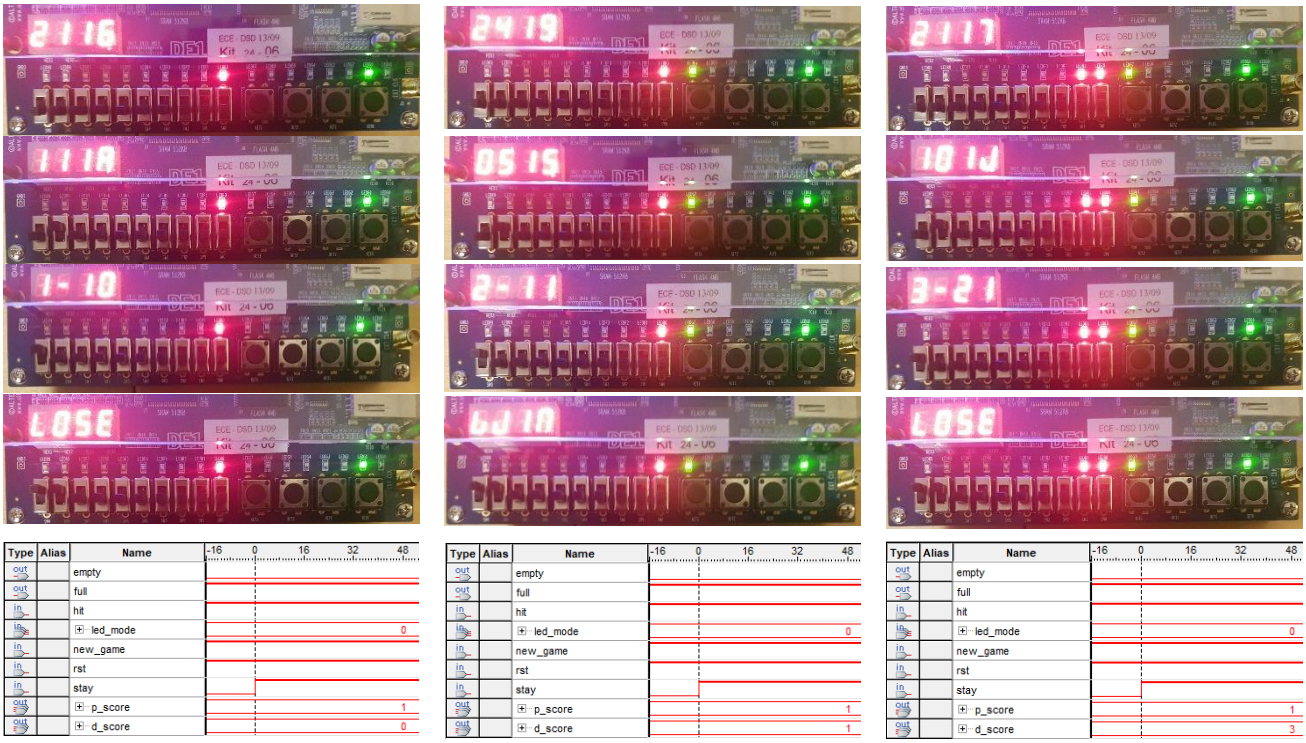
Waveform



We test the winner module using functional simulations and waveforms. We made sure to test each scenario at least once to confirm that the module can properly determine which party wins the round. We tried to play as many different games as possible to cover as many different pathway possibilities. Of course, this is barely scratching the surface as there are so many different ways of playing the same game, although, after testing our module we were able to validate and confirm that the blackjack circuit was functioning as expected.

SignalTap

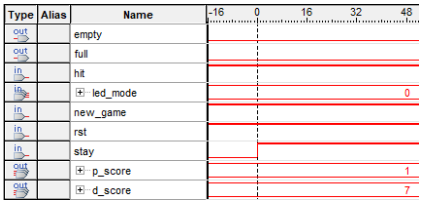
It's impossible to test every single situation using waveforms, so we also tested the circuit using SignalTap. The following images demonstrate the main tests we went through.

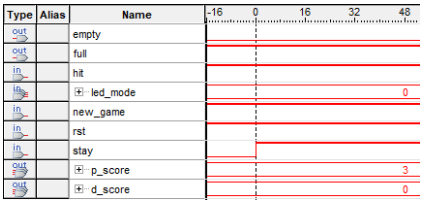


The first set of images shows the result of the round after losing. The player got a hand of 16 whereas the dealer got a hand of 21. Modes 1 through 4 are displayed in the pictures.
The score is 1-0.

The second set of images shows the result of the round after winning. The player got a hand of 19 whereas the dealer got a hand of 24, meaning he bust. Modes 1 through 4 are displayed in the pictures.
The score is 1-1.

The third set of images shows the result of the round after losing. The player got a hand of 17 whereas the dealer got a hand of 21. Modes 1 through 4 are displayed in the pictures.
The score is 2-1.





The fourth set of images shows the result of the round after losing. The player got a hand of 23, meaning he bust, whereas the dealer got a hand of 25, meaning he bust. Since the player bust (even if bost parties bust), the dealer wins. Modes 1 through 4 are displayed in the pictures.
The score is 3-2.

The fifth set of images shows the result of the round after tieing. The player got a hand of 18 and the dealer also got a hand of 18. Since both parties have the same total, nobody wins the round. Modes 1 through 4 are displayed in the pictures.
The score is 0-2, even though 3 games have been played.

Results

Concerning the limitation of the blackjack module, there are indeed many. Firstly, there is the lack of betting which overall makes the blackjack game not as interesting. With the FPGA board at hand, it is hard to implement such a design, especially with the lack of switches and buttons. Overall, it is very difficult to implement a good user interface for the player on an FPGA board since there is lacking so many inputs and ways to display information.

Secondly, the way we designed our module is to maximize the efficiency and to reduce the amount of human input. For example, we decided to automate the computer player's turn completely. It happens extremely fast and automatically. We are unable to see on the board if the computer hits or stays on his current hand, or even how many times he hits. This makes it hard for the player to track the cards that have already been played and makes calculating the probabilities of obtaining certain cards almost impossible.

Lastly, our blackjack design only features one player against the computer. This is very limiting in terms of odds and it advantages the dealer. Usually, with more people, the player is able to count the odds of getting each card and without the ability to see any other cards in play, it basically becomes a game of pure odds void of any strategy.

REFERENCES

Altera. *LPM Quick Reference Guide*, December 1996
Altera. *Development and Education Board*, 2012
Clark, J. *Lab #5 – System Integration for the Card Game*, Fall 2017, McGill University