Final Project

CPRE 281

## Drew Hirsh & Tim Kuehn

## December 9th, 2022

## Contents

[Introduction 3](#_Toc8776)

[Design 4](#_Toc8777)

[Random Number Generator 7](#_Toc8778)

[Score Reset Function 11](#_Toc8779)

[Card Register Inputs 11](#_Toc8780)

[Register Files 13](#_Toc8781)

[Adders 15](#_Toc8782)

[Comparator 16](#_Toc8783)

[Output 18](#_Toc8784)

[Finite State Machine (FSM) 22](#_Toc8785)

[Design Process 22](#_Toc8786)

[Troubleshooting 23](#_Toc8787)

[Conclusion 24](#_Toc8788)

# Introduction

For our final project, we decided to design and program a simple blackjack game. In rudimentary terms, the game is played between up to nine players and one “dealer”, wherein the objective is for the players to beat the dealer. Using a standard deck or multiple decks of playing cards, if any player achieves a hand value greater than that of the dealer’s, he/she beats the dealer. Both the players and the dealer may lose the game if the opposite party achieves a higher score than theirs, or if their hand value exceeds 21 and 17, respectively; the latter condition is known colloquially as a “bust”.

The player is dealt an initial two cards, each corresponding to a number value and compounded to form a hand value. During his/her turn, the player is given the option to “hit” or “stand”. If the player hits, he/she is dealt another card from the deck; if the player stands, the player yields his/her turn to the next player or to the dealer keeping his/her current hand value.

Obviously, to implement a comprehensive blackjack game using playing cards, chips and all standard rules would far exceed the scope of our curriculum and even the technological capabilities of the FPGA. Therefore, we decided to modify the game rules to better accommodate these limitations.

The abridged game rules are as follows:

* The game is played against two players.
* Each player’s initial score is the random number between 4 and 22.
* Each player has the option to hit or stand.
* Hitting adds a random number between 1 and 10 to a player’s score.
* If either player’s score exceeds 21, he/she busts and loses the game; his/her score is counted as zero.
* After both players have completed their turns, the player with the higher score value not exceeding 21 is awarded one point.
* The game is over once one player achieves ten points.

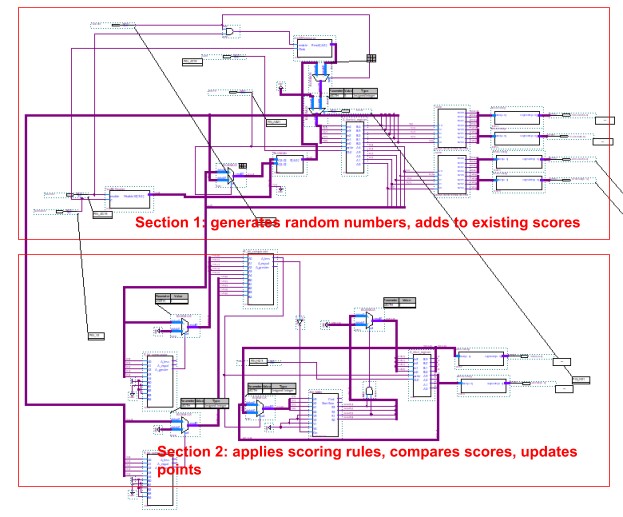
# Design

To design our project, we would need to implement two basic logic functions: one to administer random numbers to the players and one to compare both scores to determine the winning player. Furthermore, each of us designed and programmed one of these components. The scores of each player would be represented using five binary digits, which store values between 0 and 31. The points of each player would be represented using four binary digits, which store values between 0 and 15.

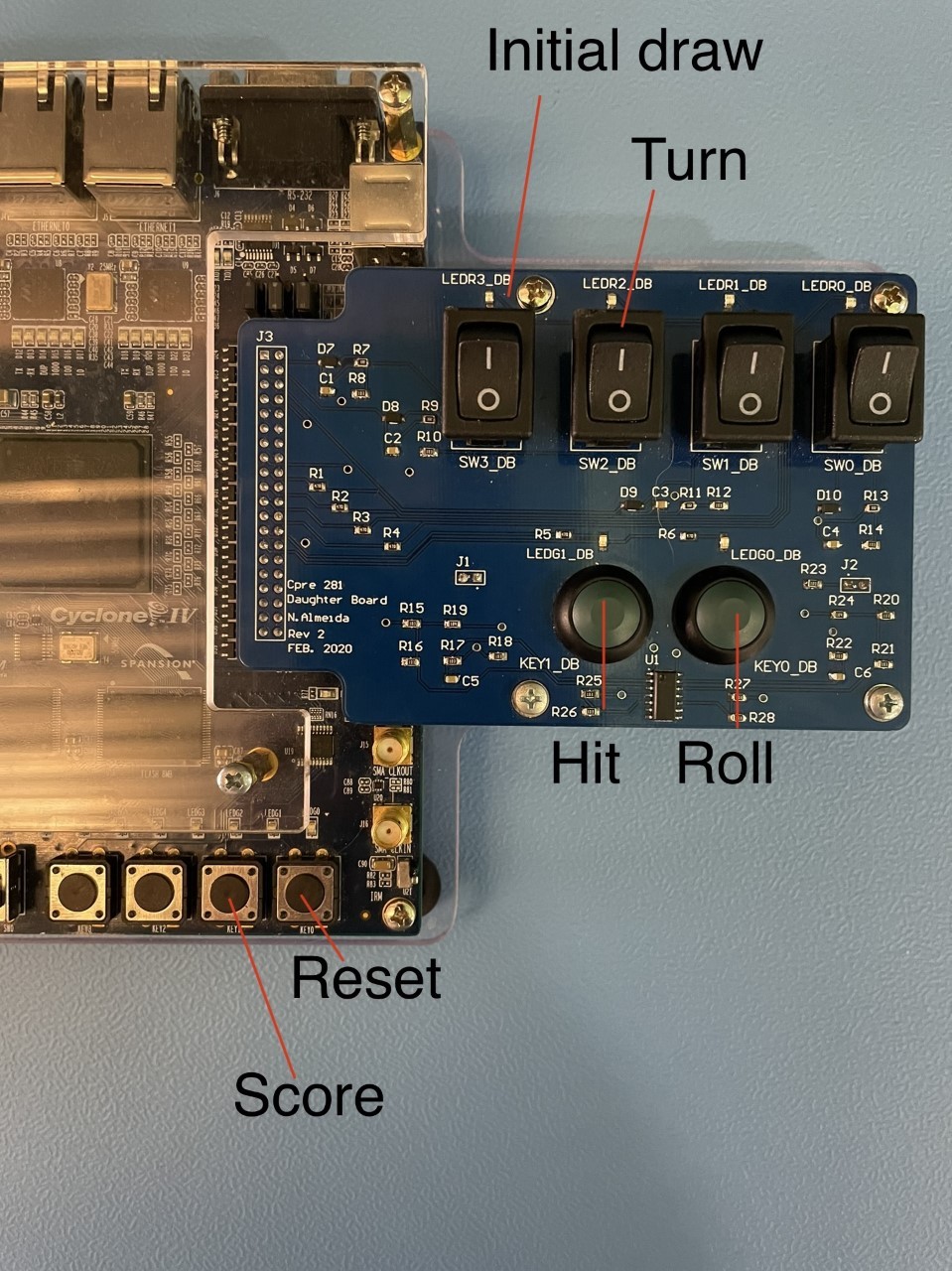
Many of the components of our project are mediated by human input, but structurally the circuit begins by passing a five-bit random number into a five-bit adder, which is finally passed into a register file. Another random number may be added to the existing number read from the register file, which is consequently passed back into the register file. All preceding functions are all performed for the next player after the first player completes his/her turn.

A separate scoring function checks first if either player’s score exceeds 21 (if so the score is set to 0), then passes the resulting value into a five-bit comparator; the comparator outputs a one-bit value that corresponds to one of two five-bit registers each containing the points of the first and second player, respectively. In the same vein, the register file input is passed in as the existing number of points incremented by 1.

Below is a top level diagram of our final project, divided into the aforementioned two logic components:



The controls programmed to the FPGA are the following:



(Note: the “Hit” and “Roll” buttons must be pressed in tandem in order to accurately generate random numbers. Similarly, the reset button resets the current player’s score when the Hit button is pressed. The functionality of each button will be explained in later sections).

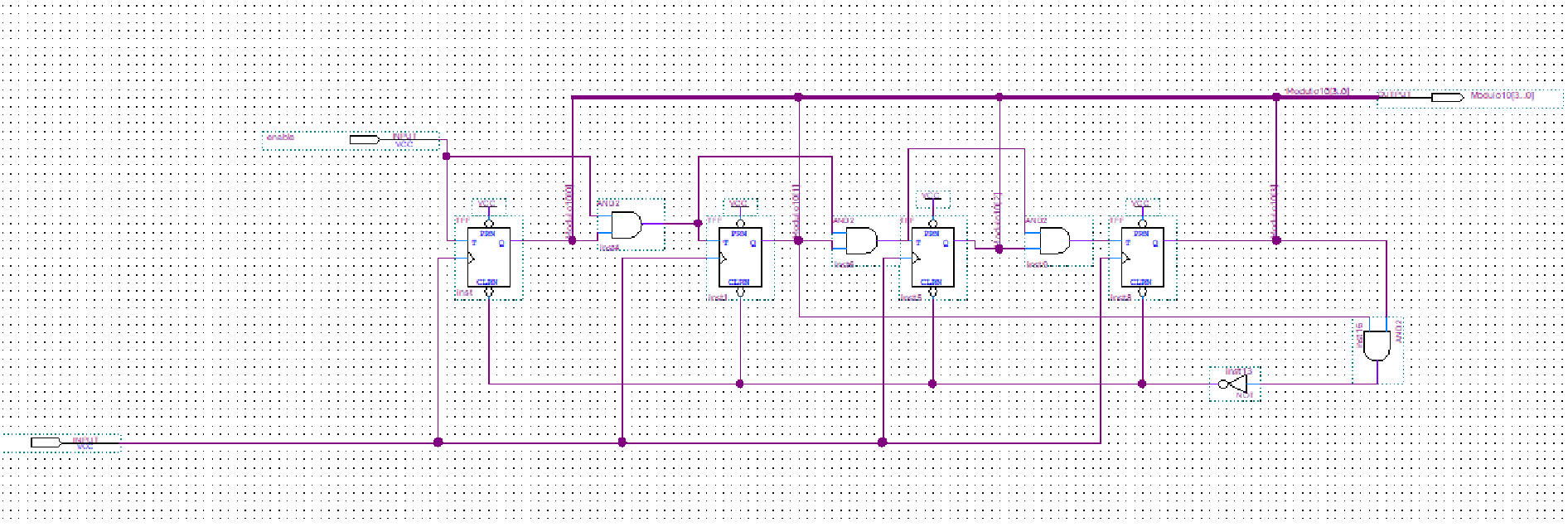
# Random Number Generator

In computers, random number generation requires the application of some external metric in order to produce “random” numbers. For simplicity’s sake, we decided to create a pseudo-random number generator mediated by human input.

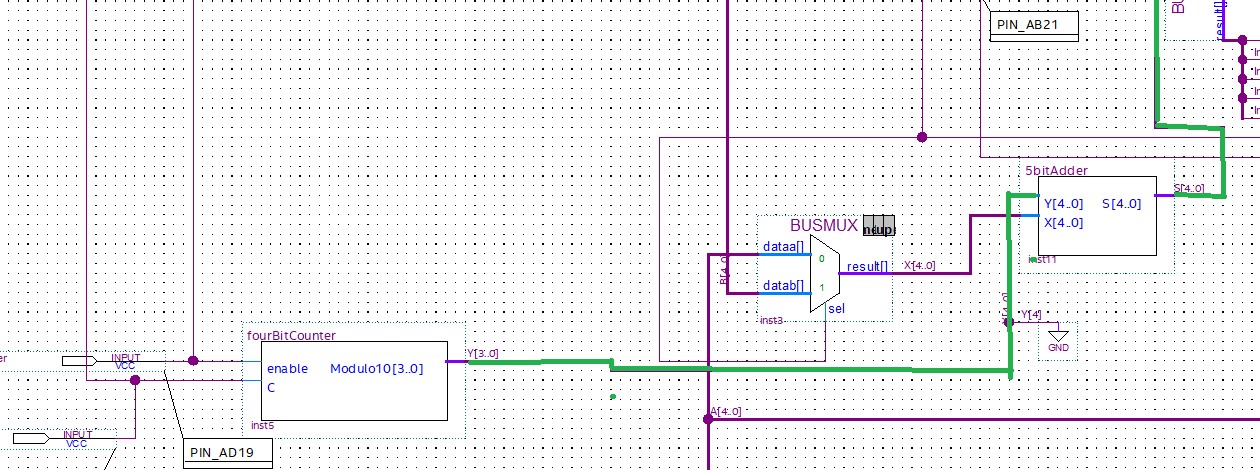
### Regular Hit Function (1 number from 1-10)

More specifically, the random number generator runs a four-bit counter (from 1-10) at the speed of the FPGA’s internal clock (50 MHz) while the Roll (KEY0\_DB) button is being pressed. When the player activates the Hit (KEY1\_DB) button, the current value of the counter is passed to the adder where it is then processed to eventually be stored in the register file. If the player is not activating Hit (KEY1\_DB) then a multiplexor passes in a value of GND to the adder and the register file does not change values. This means that although a number is generated every tick of the clock, the value is only passed on when the Hit button is pressed, allowing for an almost random number output.

Below is a schematic of the four-bit counter:



Here is how the counter and clock are implemented into the circuit (the output of the counter is traced in green):

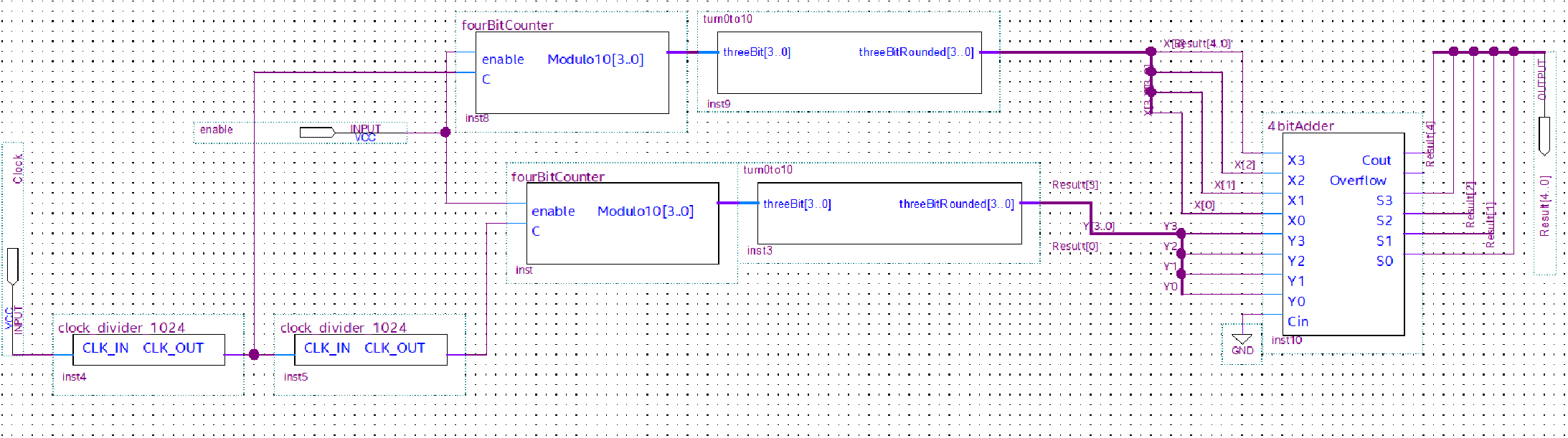


### Initial Draw Generation (1 number 4-22)

Another use for our random number generator was in the “initial draw” ; this is where instead of creating a number value between 1 and 10 it adds two number values from 2-11 and stores them automatically. The end result is a pseudo-randomly generated number from 4-22 that is the “initial flop” and overwrites whatever is stored in the player’s register. To activate this, the player hits the Initial Draw Switch, (SW3\_DB) to the ON position and holds the Roll button (KEY0\_DB) then presses Hit (KEY1\_DB). This then results in a newly generated number being stored in the register file regardless of what was previously stored. This is used at the beginning of a round.

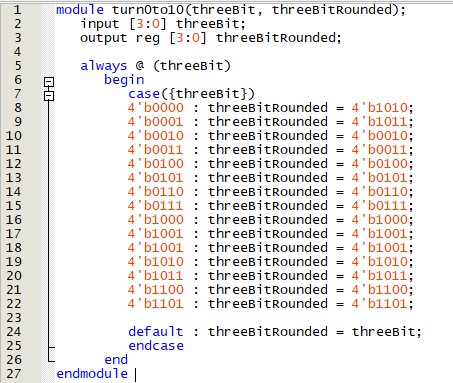
Below is an explanation of how the numbers are generated.

*4to22generator* looks like this:



The way that the *4to22generator* works is an internal clock (50MHz) is passed in to a clock divider, the speed of this clock divider is not actually import for the number generation as we do not really care what the speed is as long as it is faster than a human can react. So, in our case we used 2 1024 clock dividers, these slowed the clock down. However, to make it so the speed that is passed into the 2-11 generation is different, only one of the clocks is passed through both, and the other is just passed through one. In hindsight we did not need both of these and could have just used one, where only one of the values is passed through it and the other is at 50 MHz. But, the result of these dividers is two “arbitrary” (used loosely here) clock speeds that can be then passed into a 4 bit counter, which is enabled by SW3\_DB and KEY0\_DB. The counter is the same as the one used in the 1-10 counter so reference the above photo. Once a number 1-10 has been generated, this value is then passed into a symbol that essentially turns 0 to 10 and 1 to 11. This was done so that a player could get a “blackjack” or a 21 off of the initial draw. The Verilog code for this is below. Then, the two different values from 2-11 are passed into a 4 bit adder, which results in a 5 bit output ranging from 4-22.

Verilog code for turn 0 to 10:



The output of the 4 to 22 is then passed into a multiplexer which only overwrites the stored register if the SW3\_DB (initial draw switch) is set to on and the hit button is pressed.

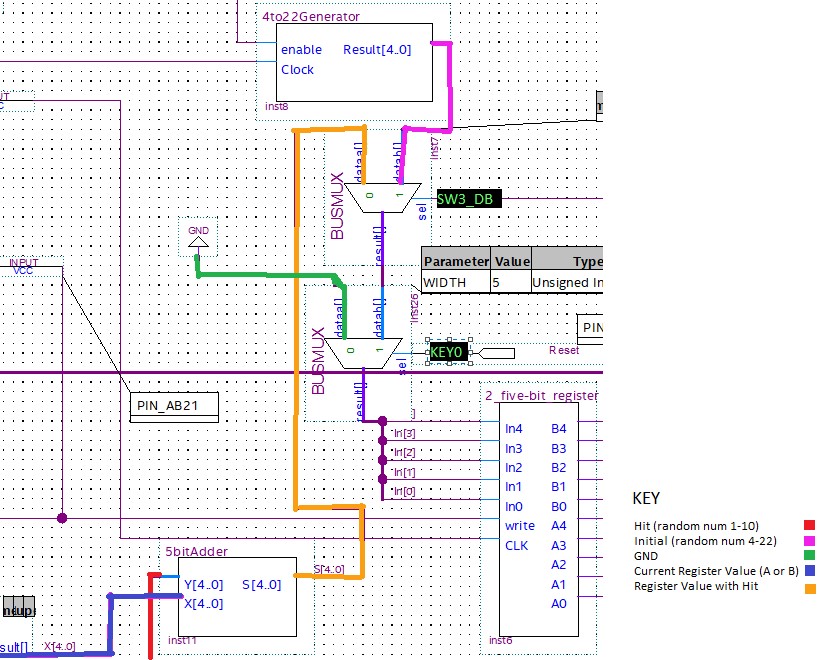
# Score Reset Function

The scores in blackjack must be set back to 0 after every hand. So, to accomplish this, we passed the “updated value” that is heading to be stored in the register file through a multiplexer that checks whether the reset button KEY0 is depressed. If it is, then the value passed into the register file is GND, resulting in all 5 bits of the register storing 0’s resetting the players cards to 0, this must be done with both players, meaning SW2\_DB must be at 0 to reset one player and at 1 to reset the other player. The end result of the different types of logic and inputs that get passed into the register is below.

# Card Register Inputs

There are 3 possible inputs for the register file, these options are outlined in the chart below:

|  |  |  |
| --- | --- | --- |
| SW3\_DB  State | KEY0  State | Result passed into register |
| 0 | 0 | Current value + random(1-10) |
| 0 | 1 | *00000* (resets player score to 0) |
| 1 | 0 | Initial hit (4-22) |
| 1 | 1 | *00000* (reset has precedence over any hits) |



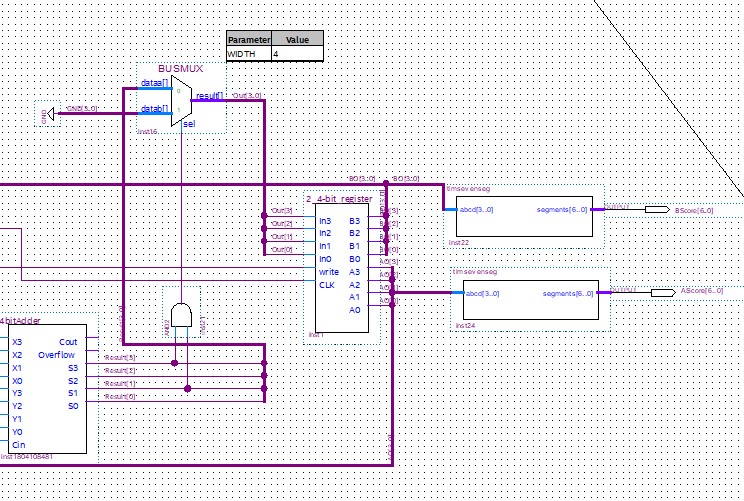
# Register File

Our project uses two different types of register files, each containing two registers controlled by a 2x1 MUX: one five-bit register to store the scores of each player and one four-bit register to store the points of each player.

### Four-bit register

The four-bit register uses a 2x1 MUX to reset to 0 once the counter has reached a value of 10; this effectively achieves the same result as two modulo 10 four-bit counters, but we decided to implement a register file to better understand its implementation and application.

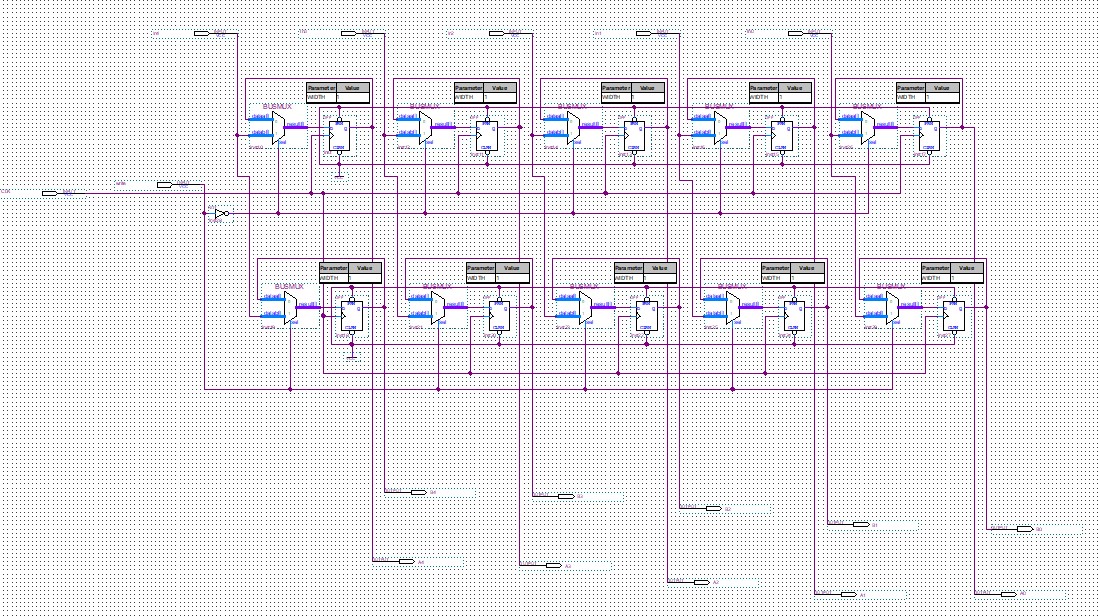
Below is a schematic of the four-bit register file as well as the associated logic components:



As shown, the output of the four-bit adder is passed into a MUX whose select value is configured according to an AND gate. The AND gate returns a value of 1 if the output is equal to ten (*1010*), thereby writing the value 0 (*0000)* into the register. Similar logic is implemented when comparing the final scores, which will be discussed in the next section.

### Five-bit register

Like the four-bit register file, the values are immediately passed to a HEX display. Below is a schematic of the five-bit register file, which alternates between two five-bit registers:

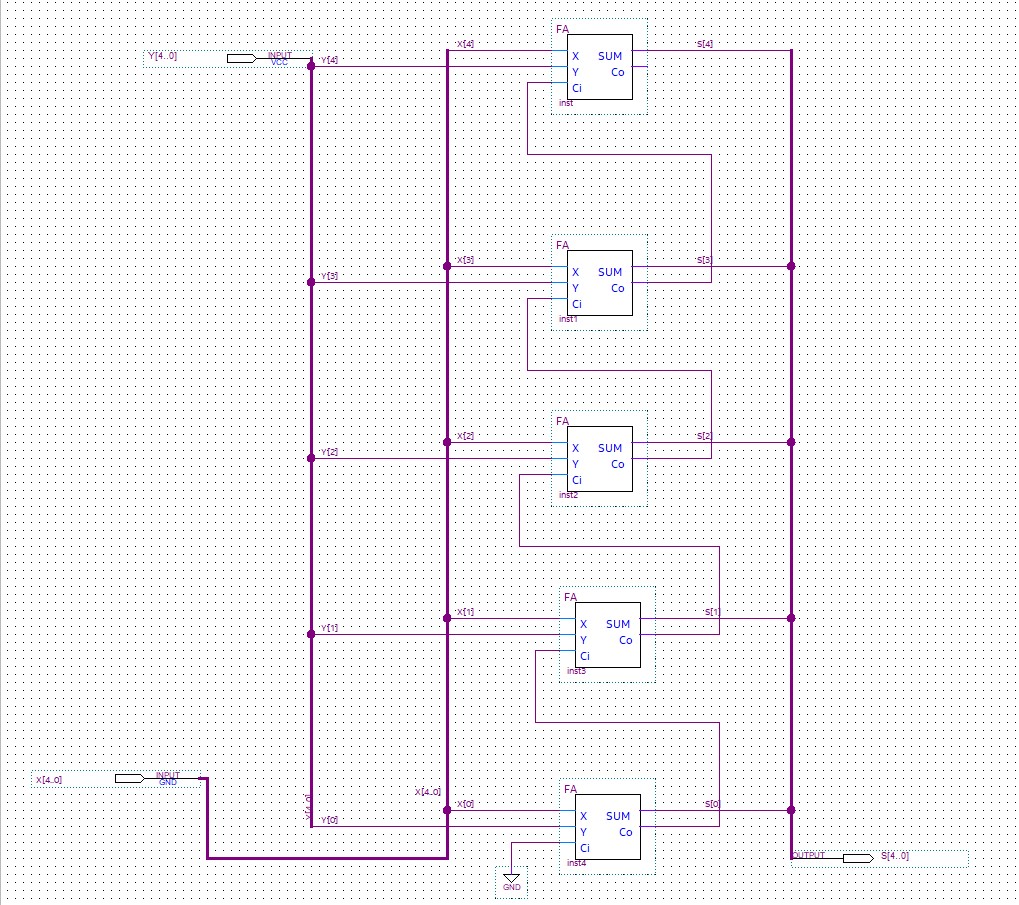


Because the register file is immediately passed to a HEX display, we viewed implementing a *READ* function as redundant. The four-bit register file is nearly identical, barring the fact that each register stores only four bits as opposed to five.

# Adders

We made use of two adders in this design, one 4 bit one and one 5 bit one. We used the 5 bit adder to add to the cards that the player had, which of course was a number ranging from 0-31, then the 4 bit adder was used to increment the stored player score by one when they “win” a hand. Both adders use full adders and carry the overflow bit up to the next full adder. We did not utilize an overflow bit as we had no function for it.

Below is a schematic of the five-bit adder:



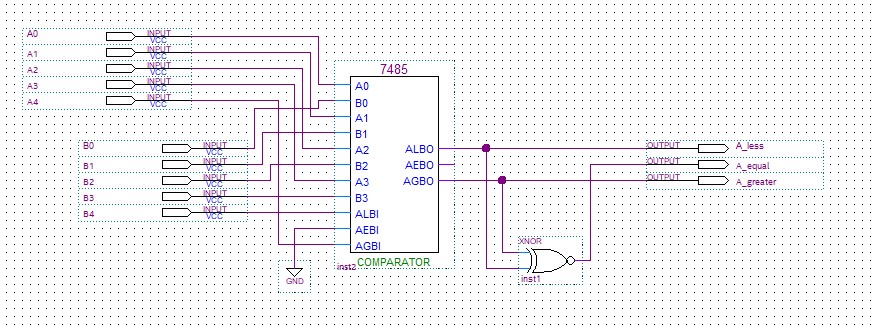
# Comparator

Once both players have completed their respective turns, the score values of each player are finally passed into a comparator function that returns a one-bit value indicating the winning player (a value of *0* means the first player wins, while a value of *1* means the second player wins). Additionally, if both scores are equal the four-bit adder increments a value of 0; this is to prevent the first player from earning a point.

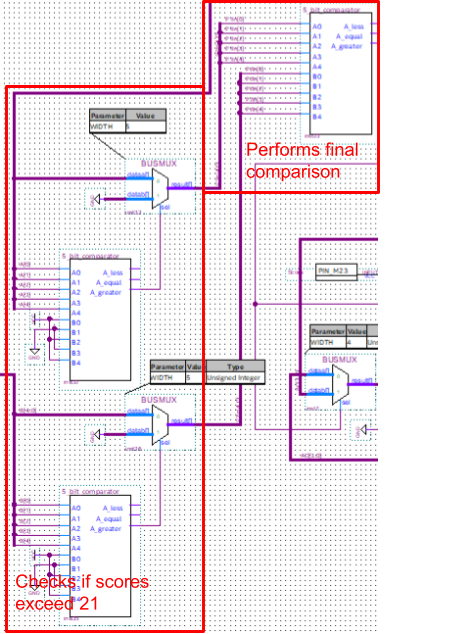
Before the final comparator function is executed, both scores are each passed through an additional five-bit comparator which checks to ensure the values are less than 21. If this is not the case, the value is counted as 0 (*00000*).

The five-bit comparator is an extension of the existing four-bit comparator provided with Quartus Prime’s software, called the *7485 COMPARATOR*. The four-bit comparator has three cascading input lines which, when group A is paired with *AGBI* and group B with *ALBI*, creates a comparator with two five-bit inputs. Because the *AEBI* input is discarded, we manually create an equal condition when neither *A\_less* or

*A\_greater* are true:



Below is a schematic of the five-bit comparators as well as their associated logic components:

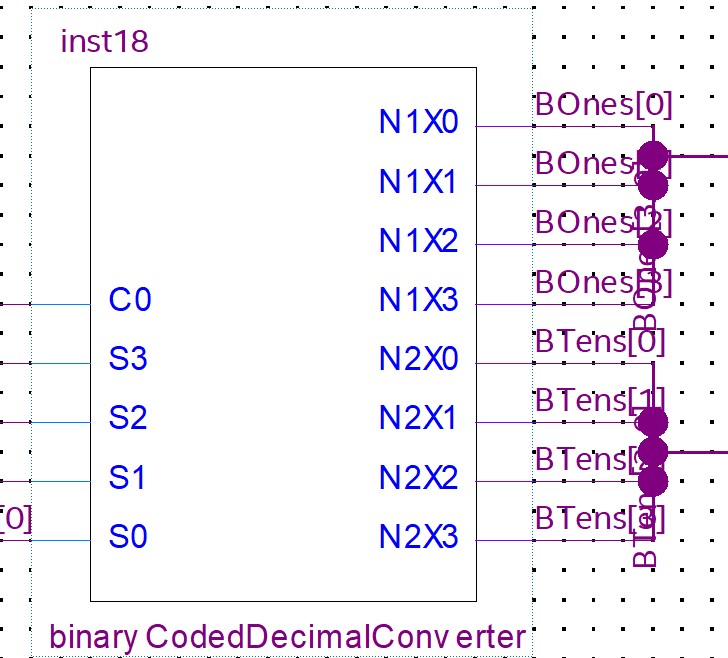


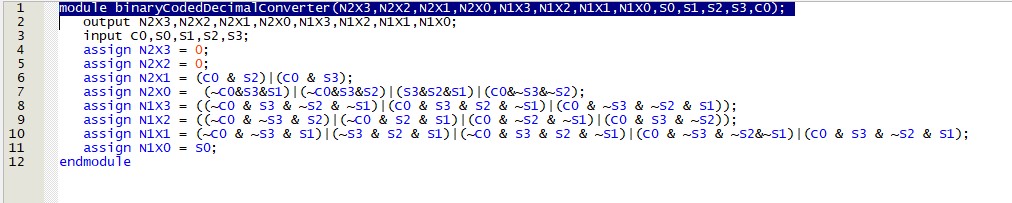
# Output

We had 6 things that needed to be outputted/communicated to the user: whose turn it was, what type of draw they were making, the card score of Player A and B and the game score of Player A and B. The turn is visible with the switch (SW2\_DB) that the player uses to select the turns, the type of draw is the same where it is visible on switch SW3\_DB. The player card score is output to two seven segment displays each for Player A and Player B. This was implemented using a binary decimal converter which passes into a seven segment display.

### Binary Decimal Converter

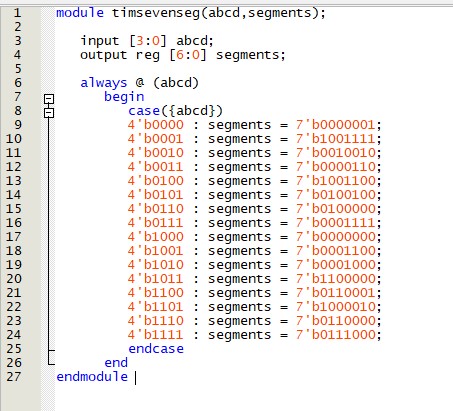
One of the issues we ran into was that the values we were storing ranged from 0-31, thus we needed a way to take a 5 bit number and convert that into two numbers from 0-9. For this we used a binary decimal converter (or BCD). The BCD only works from 0-31 because we only passed in 5 bits, S0-S3, and C0. This BCD was initially only designed in the lab to work as a 4-bit converter but we repurposed it for 5 bits. The output is 8 bits for each BCD, that has a decimal value (0-9) stored in 4 bits for both the ones place and the tens place. Below is the Verilog used to create the BCD, and below that is the implementation in our project of the BCD to output the players current cards.





### Seven Segment Decoder

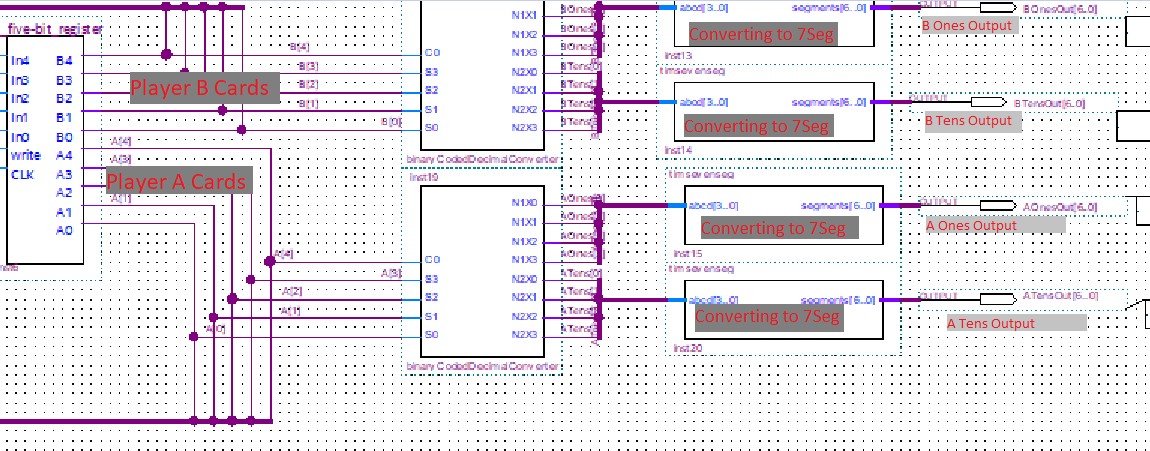
After converting from binary to decimal as described above, these values are passed into a seven-segment decoder. This decoder takes a value from 0-15 in binary (4 bits) and converts it into a seven-segment output. Despite being capable of doing values 0-F we only used 0-9 in our output to make the design more user friendly to those who do not understand hexadecimal. Below is the Verilog code that makes up our seven-segment display:



### Overall Card Output

When the player card output is put together it looks something like this. Where 1.

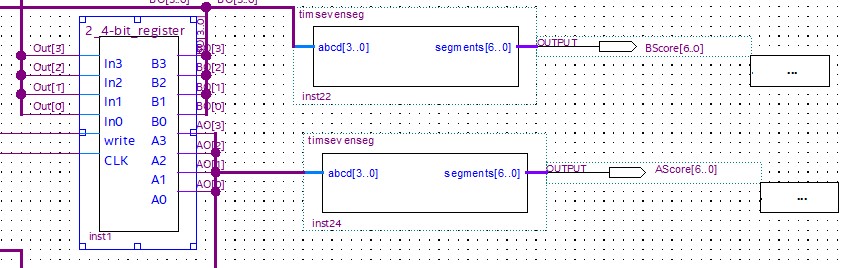
the card values are pulled from the register 2. The data is split so that it can be reused in the writing process earlier. 3. The five bit card values are passed into the BCD. 4. The four bit value from the BCD is passed into a seven segment decoder. 5. Finally the 7 bits from the seven segment decoder are passed to the output pins in HEX7 and HEX6 for Player A and HEX5 and HEX4 for Player B respectively.



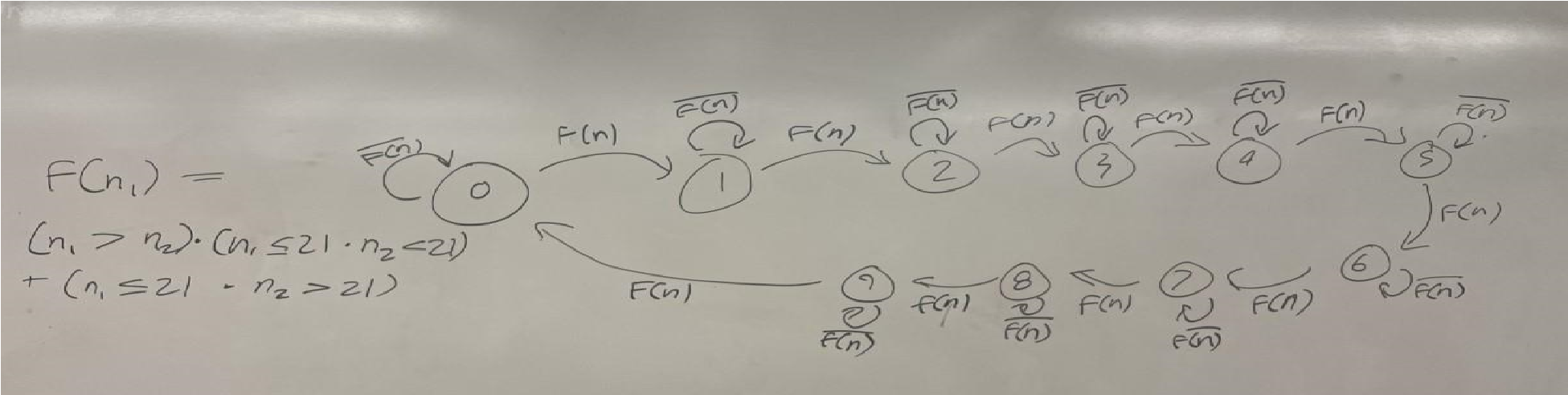
### Score Output

The score output is a much simpler task than the cards as the score is set to only range from 0-9 before restarting when the match is over. This means that there is not need for a binary coded decimal converter as we are only passing in 4 bits. Instead the player score value (stored in the 4 bit register) is taken and directly passed into the seven segment decoder, which then passes the 7 bit output to HEX3 for Player A and

HEX0 for Player B. The end circuit looks something like this:



# Finite State Machine (FSM)



Above is our finite state machine for the scoring cases. This should account for any game states, both when a player earns a point and when one does not. We wrote this as a function F(n1) where n1 and n2 are player scores (from 0-31). To boil the equation down to its bare minimum, when n1 and n2 are both below 21 (thus not “busted” and capable of scoring a point) then Player1’s score will increment if n1 > n2. If n2 is busted, and above 21 then as long as n1 ≤ 21 Then the score for Player1 would increase. If neither of these cases are true then Player1’s score will store the same value and not change. The result of this logic is only one player can score at a time and there is a chance neither player scores if they both bust, just like real blackjack. The expression n1 indicates an arbitrary player relative to n2 not a specific player, the same truth equations are true for any player n.

# Design Process

To design this project, we came into it with the goal of creating a game of blackjack. The details were not fully fleshed-out, as we were not entirely sure of our capabilities as well as the FPGA’s ability to create “random numbers.” One thing that was clear from our research is that we would have to create the numbers using a human input, and after consulting with Ahmed we realized that using the clock creates what is essentially a random number generator. After this first big hurdle we had to figure out how to manage the number adding to create the players “cards.” Most of this was just taking parts like the BCD and the random number generator and a 5 bit adder to combine into the player cards. Then we created a register file to store these cards. Once we had a way to generate two numbers with a hit function and an initial draw, we looked towards comparing the two numbers to score each “round.” The comparators took a long time to create but we discovered that Quartus comes with a default comparator that we combined to fit our needs. Then, it just came down to creating a register file to store the score and outputting all of the values to the seven segment displays.

# Troubleshooting

In general, many of our issues in development can be reduced to one of two fundamental errors: incorrect file directories and errors involving BUS wiring. More specifically, we ran into a prolonged issue in which the file directory housing our five-bit adder was somehow reconfigured. This caused the compiler to use an unfinished version of our five-bit adder, whereupon the circuit only functioned properly until we rewrote the adder completely. Wiring the BUS’s also became a headache quickly, as they demanded several specifications such as distinct names. We reached a point in our project where many issues were resolved by renaming or redrawing BUS lines, which we did not anticipate in the slightest. In our end product there are a few “unintended quirks” that came from a time-crunch and a limitation on the number of test cases we could run. One of such quirks was that if Player A busts and Player B has a small number (roughly less than 15) Player B may not get a point scored. We assume this is an error that has something to do with the bus lines into the comparators but were unable to spot the problem. Additionally, users may sometimes run into issues with the “hit” function overflowing, this was solved when we had our old adder in place but we lost this when fixing that error. These cases are rare and do not impact the functionality of our game in any major ways, however, are good to note and analyze to learn for the future.