Information Processing Report

Ignacio Bricchi, Dominic Clough, Leonardo Garofalo, Nicholas Pfaff, Igor Silin, Bradley Stanley-Clamp

March 2021

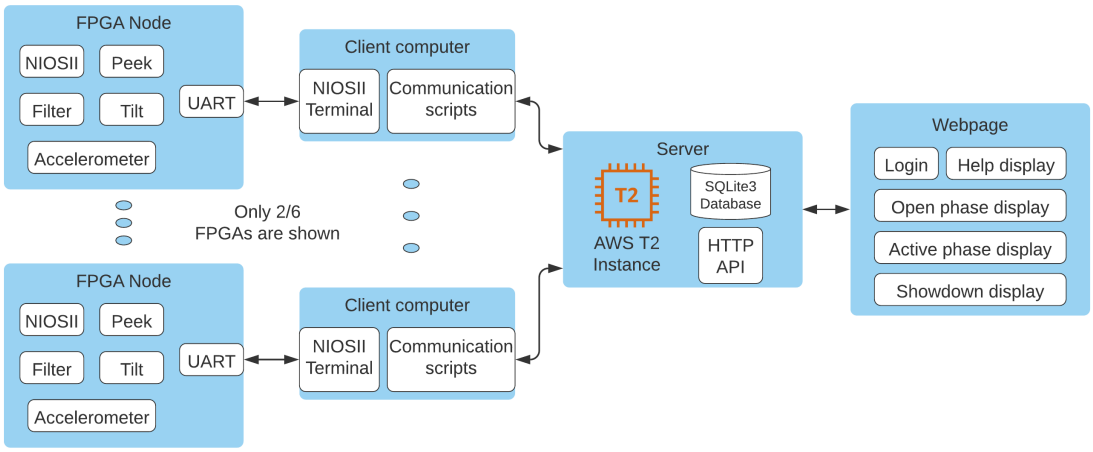
Contents

1 Introduction

The system's purpose is to play an online poker game using FPGAs as controllers. Poker is fun and suitable for multiple players. An online game is something that can be done together during the current COVID situation. In addition to standard Texas Holdem rules, the system supports additional rules such as the ability to peek at another player's cards. A player can tilt their FPGA to view their cards, similar to looking at one's cards in real life. Moreover, a player can select the amount to bet by tilting their FPGA and looking at the changing amount on the seven-segment display.

2 Overall Architecture

The diagram shown below represents the system's architecture. The hardware designed in the project is implemented on the six DE10-Lite FPGAs. An FPGA communicates with its client computer using a UART. The client computer sends data to and receives data from its FPGA using the NiosII terminal. The client computer is responsible for connecting the FPGA with the server. It does this by sending requests to the server's HTTP endpoints. To send data, the computer first needs to encode FPGA data into JSON format and decode the JSON data received from the server. The server hosts an SQLite3 database and an HTTP API. It is responsible for running the poker game logic. The server responds to HTTP "get" requests with the requested data and processes the data from HTTP "post" requests. The website also communicates with the server using HTTP requests. Its primary function is to display the current game state. The components shown in the diagram are explained in more detail in the following sections.



2.1 Server

It was decided to write the server in Golang due to it being both fast and relatively simple. The server is hosted on an AWS Free Tier T2 Micro instance. The server provides two programs: One for creating new users and one for running the HTTP API.

New users consist of a username and a password which are stored in an SQLite3 database. The passwords stored are hashed and salted for protection against potential data breaches. The API provides numerous HTTP "get" and "post" endpoints. An HTTP request is linked to a user from the database, using HTTP basic access authentication. The "get" endpoints are used to retrieve up-to-date game data, while the "post" endpoints are used to update the game state.

The API is responsible for running the Poker game logic and keeping track of the current game state. A poker game consists of an "open" phase, an "active" phase, and a "showdown" phase. The "showdown" phase is a sub-phase of the "active" phase. The "open" phase is started through the "/poker/openGame" endpoint by sending data for the initial player money and the small blind amount. During the "open" phase, players can join a game through the "/poker/joinGame" endpoint. Players must exist in the database to join a game. The "open" phase is ended, and the "active" phase started through the "/poker/startGame" endpoint. A new deck is shuffled, five cards are drawn for the community cards that will appear in the game, and two cards are allocated to each player. The dealer, small blind, and big blind are randomly allocated, and the players are sorted accordingly. Moreover, a relative card score is calculated for each player that reveals how good a player's hand is compared to the other player hands. The score takes the community cards into account. The FPGAs use this score to adjust the local peeking logic.

The webpage determines what to display by retrieving data from the "/poker/openGameStatus", the "/poker/activeGameStatus", or the "/poker/activeGameStatus/showdown" endpoint. The FPGAs can retrieve data about the current game state and send data to the server using the "/poker/fpgaData" endpoint. FPGA data send to the server is split into two categories: "Active" and "passive" data. "Active" data is sent due to a player making a move and can only be sent once during a player's turn. "Passive" data is sent continuously by every FPGA and includes information such as whether to show a player's cards.

A player can bet, call, check, fold, or raise, dependent on the previous player's move. During a player's turn, other players can attempt to peek at the current player's cards. Peeking succeeds if the current player tilted their FPGA too much while looking at their cards and another player attempted a peek. The player's cards are shown to players that successfully peeked for the remainder of the game. A failed peek results in a player's failed peek attempts increasing. More failed attempts affect the data processing on the FPGA and make it easier for other players to peek at that player's cards. The game moves to the next round once the last player that raised would be the next player, ensuring that every remaining player bet the same amount. The four rounds follow the standard Texas Holdem structure. A player wins by either being the last player remaining, with all other players having folded or due to having the best cards after the final round (the showdown round).

The response of the showdown endpoint comprises the complete showdown data, including all player hands to be revealed in the showdown phase. All other endpoint responses only include masked data, according to the player associated with the request by authentication. Masked data ensures that players cannot cheat by obtaining more information than would be available to them in a real game by looking at the HTTP responses. The winning players are computed for the showdown data with a reason for why they won (e.g., straight).

A new game with the same players can be started using the "/poker/startNewGameSamePlayers" endpoint. The last game's pot money is distributed across the winners, new cards are drawn from a new deck, and the dealer, small blind, and big blind move by one player. The "/poker/terminateGame" endpoint can be used to terminate the game. It is the only endpoint that does not perform error checking and can be called at any time. It completely resets the server's state. A new game could be "opened" after termination.

2.2 Webpage

The webpage is the game's visual interface. It displays the game state and provides an interface for selected game control.

The page has four display states. The login state is the landing page of the website. Players must log in to be associated with the page's HTTP requests. After verifying the credentials, the page switches to the lobby state. A form for entering the initial player money and the small blind amount is displayed. The webpage uses the data provided to send a request to the server for opening a game. From here, any logged-in player can join the game. After joining, a player can see a lobby of other joined players. The game can be started through a button once at least two players have joined.

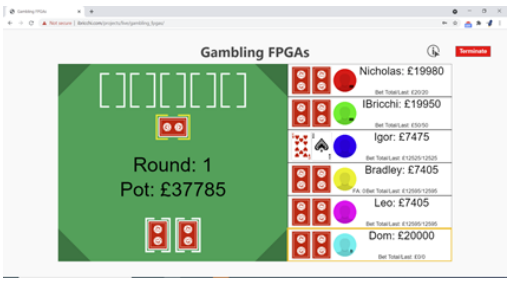


Figure 1: Active state of web page for game.

After starting a game, the page enters the active state, displaying the current game state as shown in fig. 1. Here, the webpage provides a single button for terminating the game. All other actions are done through the FPGAs. The showdown state starts at the end of the current game: The winning players are shown with their amounts won and a winning reason. A continue button enables starting a new game with the same players.

The webpage fetches data from the server every $100ms$, making the website very responsive to any changes in the server state. The server's response takes $50 - 300ms$. Hence, anything faster than $100ms$ slowed down the responsiveness as most of the requests would not complete before a new request was made.

It was decided to host the webpage separately from the server. Hosting the website on the server would increase the processing required by the single-core AWS T2 Micro instance. Further, separate hosting makes the server's data more secure from potential website compromises.

2.3 Client Computer (Communication)

The communication between the FPGA and the server passes through the client computer. A script asks the server for data, processes it, and passes it to the FPGA. It then sends the FPGA's response back to the server. The script interacts with the FPGA using the NiosII terminal, passing input into stdin and reading output from

stdout. The script sends HTTP requests using curl. The data processing concerns encoding and decoding JSON data: The server works with JSON data while the FPGA expects a lightweight custom data format.

2.4 DE10-Lite FPGA Nodes (Hardware)

The main Hardware components needed to realize our design can be divided into standard and custom components. Standard components are NiosII processor, parallel I/O peripherals, off-chip SDRAM, PLL, accelerometer, and JTAG UART. Custom IP components are FIR filter and Peek function.

2.4.1 Standard components

The NIOS II processor communicates with the hex display, switches and the buttons on the FPGA board using the parallel I/O peripherals. An SDRAM,connected to the processor with a PLL, is needed to allow the use of c libraries that take up more space than what the on-chip memory allows. The accelerometer data is used by the NIOS II processor to provide functionality. The JTAG UART allows connection between the computer and the NIOS II processor

2.4.2 Custom components

The approach taken to implement the Peek and Bet functions, and the FIR filter was to

- first implement their functionality in software in order to
  - provide a first working prototype of our project and test the communication and server sides
  - have a reference point to test our hardware against
- then progressively implement them in hardware using custom IP blocks in order to
  - optimize speed and performance of the already working prototype

FIR filter Processes the accelerometer data, to free up the NIOSII to communicate with the host PC.

A sequence of registers form a pipeline for the input accelerometer data, to store previous values. The output of each register is multiplied by a rounded, scaled tap coefficient. It is rounded to avoid floating point arithmetic, and scaled to keep precision. The larger the scale factor, the less information is lost by rounding. The final output is the sum of the scaled register outputs, so is scaled itself. This scale factor is absorbed into the constants used by the rest of the circuit (threshold values for Tilt and Bet), so the output does not need to be scaled down. The tap coefficients were chosen with the help of MATLAB Filter Designer to achieve the desired cutoff frequency (350Hz) and number of taps. Testing with MATLAB showed that 350Hz was a good corner frequency to remove high frequency noise. We chose 20 taps for the FIR as it provided an adequate frequency response without using excessive hardware.

Peek function A user is able to see his cards by tilting his FPGA board up as if it were a set of cards in real life. If the FPGA is tilted too much though, then the users cards will become visible to other users who have attempted to peek.

The purpose of the Peek function is to communicate to the server when a user is able to see his own cards and when other users are able to see his cards.

The Peek custom block receives 3 inputs

- the "tilt angle" which is the angle of the y axis of the accelerometer
- a relative card score value computed by the server
- the value of the "lock" switch

The user with the best set of cards will be assigned the highest relative card score. Similarly, the relative card score assigned to other users will be proportional to their chances of winning the game.

The relative card score value is used to compute the "user-peek angle", which is the angle at which the cards of the user become visible to the user themselves, and the "all-peek angle", which is the angle at which the cards of the user become visible to other users. The tilt-angle is then compared against the user-peek angle and the all peek angle. Depending on the output of the comparison different data is sent to the server, in particular

- if tilt angle $<$ user-peek angle then no one is able to see the user's cards
- if user-peek angle $<$ tilt angle $<$ all-peek angle then only the user is able to see his cards
- and if tilt angle $<$ user-peek angle $<$ all-peek angle then all the users in the game will be able to see the user's cards

If the lock switch is asserted then no one is able to see the cards of the user regardless of tilt angle. If a user tries to peek other users cards too many times than his all-peek angle will decrease.

3 Performance Metrics

Poker does not require ultra-low latency. However, some of the system's dynamic, such as tilting the FPGA to look at once's cards, feel smoother with lower latency.

The data path from tilting the FPGA to seeing one's cards on the screen has several delays. Sending requests to the server takes approximately $100ms$, dependent on physical location and internet speed. The server processes requests in around $2-3ms$. The website takes a negligible amount of time to display the fetched data: It runs at an average frame rate of $30-40fps$, dependent on the computer accessing the website.

The initial bottleneck delay used to be communicating between the computer and FPGA. The initial method used opened a new instance of the nios2-terminal for every read-write pair. This opening and closing resulted in the time taken for a read-write pair being at best $0.9s$ and on average $1.5s$.

The time taken from tilting the FPGA to seeing the cards appear on the screen is the sum of the time taken to get data from the server for the FPGA, to write to and read from the FPGA, to send the FPGA response to the server, to process data on the server, and to fetch data from the server for the website. Overall, the initial implementation resulted in this time being about $1800ms$. This made looking at one's cards laggy.

An optimisation made to the communication script was to open the nios2-terminal as a bash coprocess. This made it possible to keep the nios2-terminal open. This speed up reading/ writing from/ to the FPGA from an average of $1.5s$ to $500ms$, representing a $3x$ improvement.

The new bottleneck was the JSON decoding. An attempt to improve the decoding algorithm yielded little improvement. Hence, it was decided to move the decoding from the FPGA to the client computer. A simpler data format was designed for sending to the FPGA that could be processed more quickly. This provided a speedup from an average of $500ms$ to $300ms$. The same concept was tested for the JSON encoding. However, this led to a drop in performance, so it was reverted.

These optimisations improved the total latency from tilting to seeing one's cards from $2s$ to $600ms$, representing almost a $4x$ improvement.

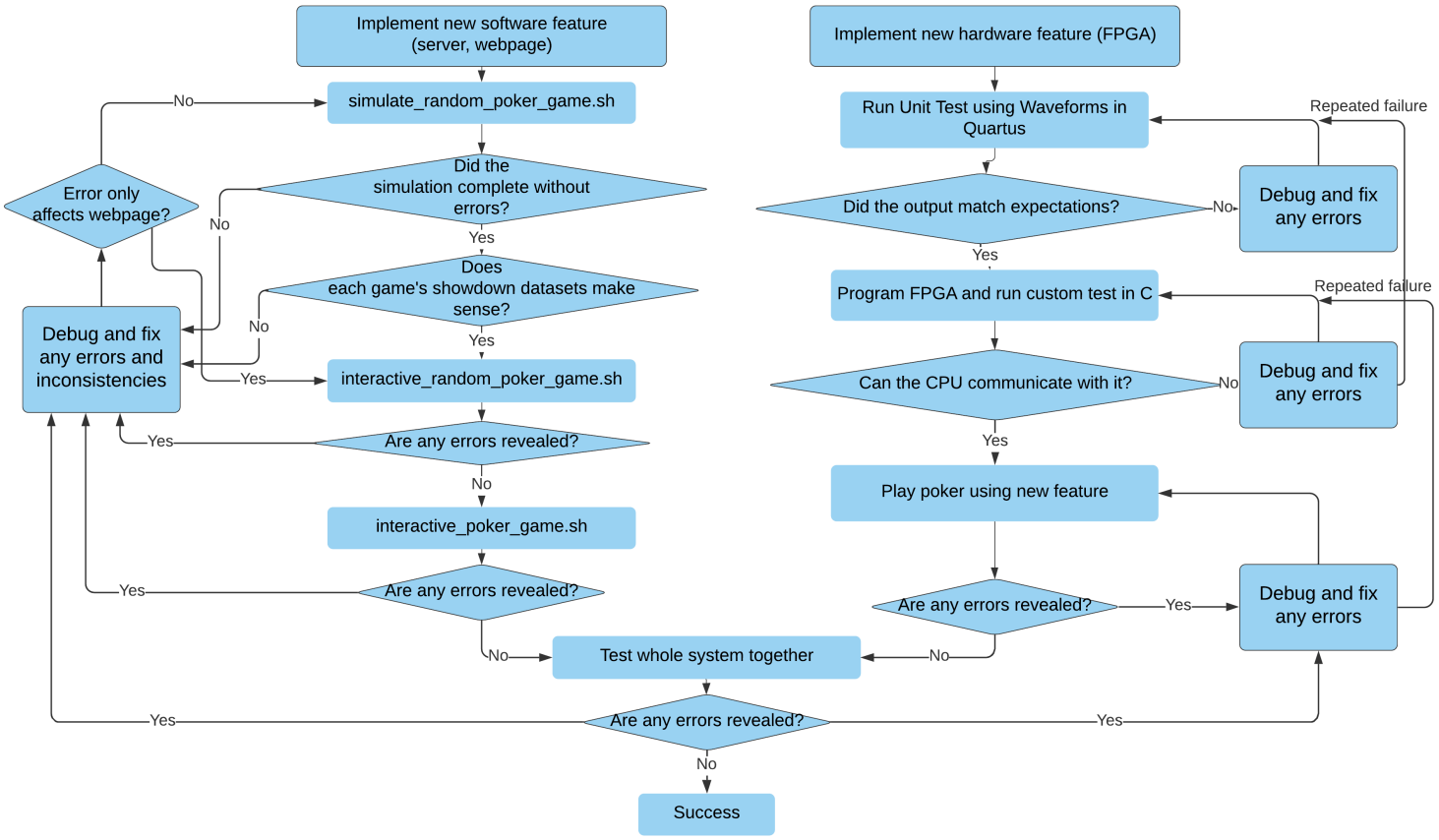
A final improvement was to replace the software filter, used for smoothing the accelerometer data, with a hardware version. The FPGA's NiosII processor used to be responsible for processing both the input/output data and running the filter. Moving the filter into hardware allows the processor to focus on the input/output data. This optimisation did not provide significant improvement in latency.

In the c code run by the NiosII processor, there are two loops, one for processing input and output, and a second which processes data. The second loop runs at a given interval rate by interrupting the main data processing loop. This interrupt has a significant performance cost. Future optimisation would begin by removing this way of running the two loops in exchange for a more efficient method.

4 Testing Approach

4.1 Server and Webpage

Three testing scripts were designed for testing the server and the webpage. "simulate_random_poker_game.sh" works by simulating numerous consecutive games within seconds. It prints a formatted version of each game's showdown data to stdout. This script neither relies on a functional webpage nor on functional FPGAs. Looking at the showdown data and adding up the numbers after several games is useful for revealing logical game errors. Moreover, this script does not require any user interaction and hence can be used for testing changes quickly. "interactive_random_poker_game.sh" can be used for testing both the server and the webpage interactively. It is useful for checking whether the game logic stays correct after each new action and whether the webpage displays the correct data. Both "simulate_random_poker_game.sh" and "interactive_random_poker_game.sh" simulate the FPGAs using a random number generator. The next bet amount is a random number in a range above the minimum next bet amount. Peeking and the next move are determined based on probabilities. This randomness allows for a huge variety of game flows that include scenarios that are unlikely to come up in real games. The third script, "interactive_poker_game.sh", is similar to "simulate_random_poker_game.sh" but prompts the user for each new action to be sent to the server. This makes testing specific functionalities easy. This script is the most time consuming to run, as it requires the most user input but is also the most precise in testing specific functionalities. These scripts enabled intensive and rapid testing of the server and webpage before the FPGAs were completed, allowing for more parallel software and hardware development. They helped to identify and address many bugs throughout the development process.



4.2 Hardware

Testing the custom hardware occurred in 3 separate stages:

1. Write the Verilog and perform a unit test using waveforms in Quartus. This ensures that the custom block behaves as expected on its own.

2. Add the custom IP Core to the system and program onto the FPGA, using a specific C file to test the component's connection with the CPU and that it still works as expected.

3. Modify the main C file to use the custom IP core instead of software functions, and simulate a poker game.

If an edge-case is found that fails, we modify the test (either waveform or C file) to focus on it and repeat the stage. This helps locate bugs and clarify how the system is failing. If we cannot debug it here, we revert to the previous stage.

5 DE10-Lite Resource Utilisation

Our final design utilised: 7475/49760 (15%) total logic elements, 4230 total registers, 187/360 (52%) total pins, 0 total virtual pins, 865280/1677312 (52%) total memory bits, 10/288(3%) embedded multiplier 9 elements, 1/4 (25%) total PLLs, 0/1 UFM blocks, 0/2 ADC blocks