Imperial College London

Information Processing

Coursework Report

Authors
Brandon Cann
Vajiravidh Pongpaew
David Cormier
Petra Ratkai
Razvan Rusu
Thomas Loureiro van Issum

Contents

1 System Purpose

The IoT system we have designed is a points-based racing game. It currently supports 6 players; however, this could readily be scaled up to facilitate more players. The FPGA is primarily used as a steering wheel for the game, however the score of each player is also displayed on the FPGA's 7-segment display. The game itself is hosted on an HTTP server that each player can connect to using a web browser. Each player controls their vehicle by moving their FPGA and at the same time they can see their vehicle's movements once connected to the server. Players can see a map of the course with various terrain, as well as their relative position compared to other vehicles and scores. Each player has a finite number of "lives" after which they will no longer be able to gain any points and their vehicle will disappear. The course itself consists of several different terrain types, represented by different coloured tiles. The FPGA receives the type of terrain its vehicle is on and processes the accelerometer data, in order to represent the differing responsiveness of driving on different types of terrain in real life.

2 System Architecture

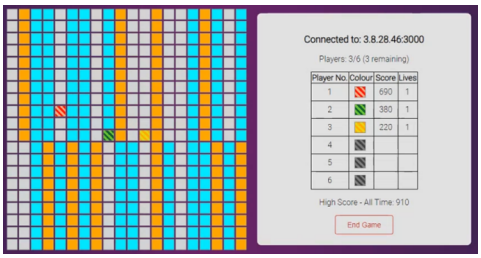
Our system consists of four nodes. The first and most important is the FPGA, which serves as the primary method of user interaction with the game. The players can move the FPGA which uses an accelerometer to quantify the movement data. Subsequently, the FPGA uses an FIR filter to make this data usable to control the vehicles. This data is then sent to the next node, the host PC via a JTAG UART connection. At the same time, the FPGA receives data from the host PC regarding the relevant vehicle's current score as well as the type of terrain the vehicle is currently driving over. The FPGA displays its current score on its 7-segment display and uses the terrain type to select what type of filtering it uses. The filtering is intended to represent the difficulty and lack of responsiveness of driving over certain types of terrain e.g., ice or mud as compared to a racetrack.

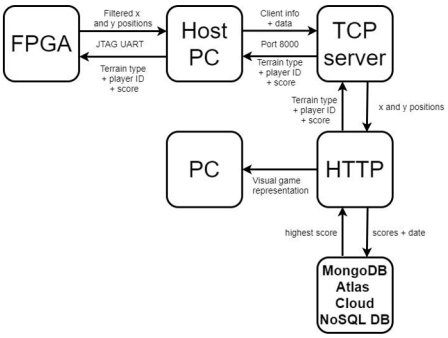
The two intermediary nodes are the host PC and the TCP server. The host PC's primary purpose is to transfer data to and from the FPGA as well as the TCP server. It receives data from the FPGA and transfers data to the FPGA using C code that utilises the JTAG UART connection. Similarly, the host PC transfers data to and from the TCP server using a python script that takes the data from the C code. The TCP server serves a similar role to the host PC as an intermediary, transferring data between the host PC and the HTTP server (which is the final node). Specifically, the TCP server has an array of connections that continuously listen to the incoming data at the port. Whenever a new client connects it is added to this vector and the TCP server subsequently forwards the data to the HTTP server. The TCP and HTTP server were both implemented using Nodejs, because its flexible, lightweight, and fast which are all desirable characteristics for the servers.

A HTTP server was implemented as it can easily serve a webpage to a browser, whilst a TCP cannot. The HTTP server hosts the game and serves a html page to a browser. This is subsequently decoded by the browser to visually represent the game. The server receives the x and y position generated by the accelerometer and filtering of the FPGA, which it subsequently uses to identify which tile each vehicle will

move to in the next server tick. Simultaneously, the server identifies the terrain and score of each vehicle, which it transfers to the TCP server so that it can be transferred back to the FPGA to direct filtering selection. The HTTP server also keeps track of the current number of lives of each player to identify when they have "died" and can no longer accumulate score. Finally, the server tracks the all-time high score via use of an external database MongoDB. MongoDB is a NOSQL database that can be hosted in the cloud. It was chosen because it frees up system resources on AWS whilst also being fast and performant. Finally, as a NOSQL database, it stores entries as JSON 'documents' allowing them to be easily manipulated by the HTTP Server. This database can be readily expanded to store other data in future.

Finally, whilst not technically a node of the system, it is possible for anybody to independently connect to the HTTP server, to see the visual representation of the game. This can be either a player or a spectator. Beyond the map itself, people connected to the HTTP server can also see the player scores and the all-time high score as can be seen in the following image. The diagram of the architecture can also be seen below.





3 Performance Metrics

There are a wide variety of performance metrics that can be used given the multi- faceted nature of the project. Several key metrics were identified: Tick rate of server, ping from host PC to server, connection time from host PC to FPGA, the processing time of the FPGA itself and finally the accuracy of the driving of the vehicle. These criteria are together capable of providing a comprehensive measurement of the system.

The processing time of the FPGA itself is evidently a critical measurement as we are reliant on the FPGA to fundamentally produce the data that the entire system is reliant on. If this were to take too long, then the system would be rendered pointless. There is a myriad of methods to implement the processing that all have a variety of trade-offs. Some methods considered were using a dedicated hardware FIR filter as opposed to a filter implemented in a softcore processor. Our project instead chose to focus on using a softcore processor filter, but within this there are still multiple implementations. Specifically, either using a floating-point implementation or a fixed-point implementation. The timing data for this can be seen below, ultimately the fixed-point was significantly faster. This implementation did have the trade-off of producing less precise data, however this was not a problem as the precision of the data only required to

differentiate between which tile out of the 9 tiles the vehicle will move to. For this reason, it was decided to use a fixed-point implementation as it had a superior processing time, and the drawbacks were not significant. Finally, these FIR filters are designed using a sampling frequency of 10Hz, this is acceptable as it is significantly higher than the tick rate of the game server, whilst limiting the amount of processing done by the FPGA.

Trial Number	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)	6 (ms)	7 (ms)	8 (ms)	9 (ms)	10 (ms)	Average (ms)
Floating point	94	184	183	183	92	187	91	184	93	153	144.4
Fixed point	21	32	31	32	31	32	32	31	32	32	30.6

Whilst the connection time from the FPGA to the host PC is potentially not as critical as the processing time itself, it still can potentially serve as a large delay, and thus impact the quality of the game itself. The testing data for this can be seen below, but it was observed that changing the type or amount of the data did not significantly impact the time taken to receive a response. Therefore, the project transferred relatively more data to improve the overall quality of the game without negatively impact the lag in playing the game.

Trial Number	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)	6 (ms)	7 (ms)	8 (ms)	9 (ms)	10 (ms)	Average (ms)
Char	811	789	793	819	819	814	824	811	799	815	809.4
Int	822	799	809	796	815	814	793	803	798	803	805.2
Long	798	816	808	823	813	806	798	811	809	812	809.4

The ping from the host PC to the server is relatively hard to influence directly, but the quality of the transferred data itself must also be measured and maintained. This metric is largely dependent on the location of the host PC's and the server, and the internet speed the host PCs are using. Therefore, the server was hosted in London, a central location relative to most host PCs. This served to minimise the ping from the host PC to the server, the data for which can be seen below.

Trial Number	1 (ms)	2 (ms)	3 (ms)	4 (ms)	5 (ms)	6 (ms)	7 (ms)	8 (ms)	9 (ms)	10 (ms)	Average(ms)
London	43.599	87.765	43.267	36.772	38.945	34.621	32.550	59.627	56.822	37.469	47.1437
N. Virginia	98.423	105.335	150.432	105.643	101.324	90.543	130.532	102.332	110.523	105.322	110.0409

The tick rate of the server is one of the most integral metrics to the game itself, it dictates how accurately the game can utilise the data it receives as well as how the user experiences the game. Whilst there is a theoretical maximum tick rate, the data for which can be seen below, this was largely irrelevant in deciding the tick rate. If the server's tick rate was increased too much then it would be unable to handle all 6 players connecting simultaneously, furthermore due to the difference in the time between data from the FPGA being received and the players being able to see the game on the HTTP server, if the tick rate was increased too much it could become significantly harder to accurately predict the upcoming terrain and change the "driving" to accommodate. Conversely if the tick rate were lower than 1Hz the game would be relatively unresponsive and slow-paced drastically reducing the player's enjoyment.

Finally, the accuracy of driving the vehicle is critical to the player experience of the game, despite this however there is not a quantifiable metric that can be directly measured. Instead, this was tested by using a variety of different players and filters to ensure that the responsiveness of the vehicle was suitable for meaningfully playing the game.

Time taken from server to pc	Time taken from pc to server	Preparation Time	Overall time taken for Response	Best Case Server Tick Rate
25ms	25ms	12ms	62ms	16Hz

4 Design Decisions

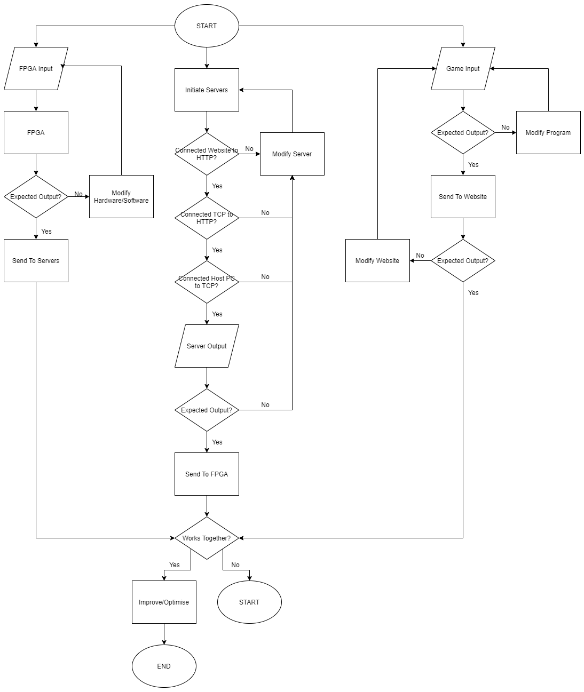
There were two main design decisions in the project. The first was the visual aspect of the game, which was critical to the project. This must be clear and understandable at first glance, as well as be able to convey a lot of information such as where the player's vehicle is, what the players score is, how many lives the player has, and the actual terrain around the player's vehicle. Without this information a player would be unable to meaningfully play this game and so ensuring that it can be easily understood is incredibly important. Furthermore, this visual representation would need to be supported on a variety of monitors and web browsers. All these considerations had to be carefully balanced against the functional capabilities of the server. This led to a design that had large distinctly coloured tiles to represent the terrain of the course as well as clear colours to represent each vehicle. These colours were specifically selected to ensure red-green colour blindness compatibility. Furthermore, there is a clear scoreboard to the right of the course that indicates each player's score as well as their remaining number of lives.

The second design decision was how sensitive to make the controller and server. The sensitivity had to be such that tiny movements would not impact the overall movement of each player's vehicle but also that players would not need to make huge movements to see a meaningful response. This was done in two main ways, the first is the actual FIR filters implemented FPGA board, whilst the data is filtered differently according to the terrain, each filter endeavours to minimise the impact of small movements. Furthermore, the server uses the direction of each movement and does not significantly consider the magnitude of each movement, thus ensuring that controlled movements are sufficient to control the vehicle.

5 Testing

Initial testing was done by testing each component of the IoT system individually. This ensured that any errors or optimisations that could be performed on any individual component were all completed before it encountered any other components. This has the obvious benefits of being able to identify specific issues early in the design process and ensure that debugging them is simpler as there is no confusion as to which component was causing it. Similarly, it meant that each component could be optimised independently of other components and therefore that future trade-offs in terms of processing location could be tracked more easily. After this each connection between components was tested individually and debugged once again making it apparent where any errors were occurring. Finally, once the entire project was connected, it was directly play tested by 6 players, ensuring that the final level of functionality desired was fully implemented.

Testing Flowchart



6 Resources Utilised for DE10-Lite

The resources utilised by DE10-Lite can be seen below as synthesised by Quartus.

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Memory Bits	UFM Blocks	DSP Elements	DSP 9x9	DSP 18x18	Pins	Virtual Pins	ADC blocks
1	|hello_world	4276 (0)	3125 (0)	64896	0	0	0	0	95	0	0
1	|cpu:inst5|	3985 (0)	2962 (0)	64896	0	0	0	0	0	0	0
1	|altera_reset_controller:rst_controller|	6 (5)	16 (10)	0	0	0	0	0	0	0	0
2	|cpu_accelerometer_spi:accelerometer_spi|	129 (45)	108 (37)	0	0	0	0	0	0	0	0
3	|cpu_cpu:cpu|	2178 (33)	1589 (39)	63872	0	0	0	0	0	0	0
4	|cpu_hex0:hex0|	10 (10)	7 (7)	0	0	0	0	0	0	0	0
5	|cpu_hex0:hex1|	8 (8)	7 (7)	0	0	0	0	0	0	0	0
6	|cpu_hex0:hex2|	8 (8)	7 (7)	0	0	0	0	0	0	0	0
7	|cpu_hex0:hex3|	8 (8)	7 (7)	0	0	0	0	0	0	0	0
8	|cpu_hex0:hex4|	8 (8)	7 (7)	0	0	0	0	0	0	0	0
9	|cpu_hex0:hex5|	10 (10)	7 (7)	0	0	0	0	0	0	0	0
10	|cpu_jtag_uart:jtag_uart|	139 (36)	113 (13)	1024	0	0	0	0	0	0	0
11	|cpu_mm_interconnect_0:mm_interconnect_0|	958 (0)	616 (0)	0	0	0	0	0	0	0	0
12	|cpu_sdram:sdram|	265 (213)	238 (152)	0	0	0	0	0	0	0	0
13	|cpu_sys_clk_timer:sys_clk_timer|	126 (126)	120 (120)	0	0	0	0	0	0	0	0
14	|cpu_sys_clk_timer:timer_0|	132 (132)	120 (120)	0	0	0	0	0	0	0	0
2	|pzdyqx:nabboc|	122 (0)	72 (0)	0	0	0	0	0	0	0	0
3	|sld_hub:auto_hub|	169 (1)	91 (0)	0	0	0	0	0	0	0	0

	Resource	Usage
1	Total registers	3125
1	-- I/O registers	0
2	-- Dedicated logic registers	3125
2	Total memory bits	64896
3	Total fan-out	30113
4	Total combinational functions	4276
5	Maximum fan-out node	iCLK_50~input
6	Maximum fan-out	3124
7	Logic elements by mode	
1	-- normal mode	4025
2	-- arithmetic mode	251
8	Logic element usage by number of LUT inputs	
1	-- 3 input functions	1139
2	-- 4 input functions	2428
3	-- <=2 input functions	709
9	I/O pins	95
10	Estimated Total logic elements	5,297
11	Embedded Multiplier 9-bit elements	0

7 Appendix

Entry 1: How to run the game

To play the game, one person must start a server on AWS and run the http_server.js file which starts the HTTP server and then start the TCP server by running the TCP_server.js file, again in the same AWS instance (commands: node http_server.js and node TCP_server.js)

(Note: to run the JavaScript files, the following JavaScript modules must be installed: mongodb, express, cors and path by using the npm install module_name command)

After that the players can open a browser and connect to port 3000 of the AWS server (so IP:3000 where IP is the address of the AWS server) and then in the browser they must click Connect (without changing the default IP address), then on the next page click Start and the game will start displaying.

The players must connect their FPGAs to the host PCs via JTAG UART, blast the Quartus project 'hello_world' onto the FPGAs, build the C project 'x_final_jtag_uart_accel_fir_sevseg' and download the .elf file to the boards. Then they must run the host.py script (command: python3 host.py AWS_IP) in a terminal. After running the python script, the cars show up in the browser and the players can already start steering their cars.

Once the game is running the players need to try to avoid hitting the orange fields. If one hits an orange field, they are out of the game. The game can be restarted at any time by clicking on the End Game and then the New Game buttons.