Developing an IoT System

ELEC50009 Information Processing

I.Mohamed, T.Moores, A.Rehman, H.Solomon

Table of Contents

Aim

The aim of this project was to recreate the popular arcade game Pong through which users could connect to the game via their DE10 FPGA boards. The data from the Accelerometer SPI interface of each FPGA would be processed locally and used to move the player's paddle through communication with the cloud-based Game server, which would handle updates to the game state.

Design

Functional & Non-Functional Requirements

For the system to perform as stated in the in line with system requirement, the following functional and non-functional requirements need to be met:

- Functional
  - Establish connection between nodes (user FPGAs) and the cloud based server.
  - Have a method of sourcing data to alter players' paddle position via the Accelerometer SPI Interface.
  - Authentically recreate the game 'Pong'.
  - Have means for at least 2 players to join the gain simultaneously.
- Non-functional
  - Have a pleasant user inteface.
  - Display the score of the game via the 7-segment display
  - Have minimal Latency between when the user moves their FPGA board and the paddle moves on screen.
  - Have a low-maintenance and stable server application.

System Design & Architecture

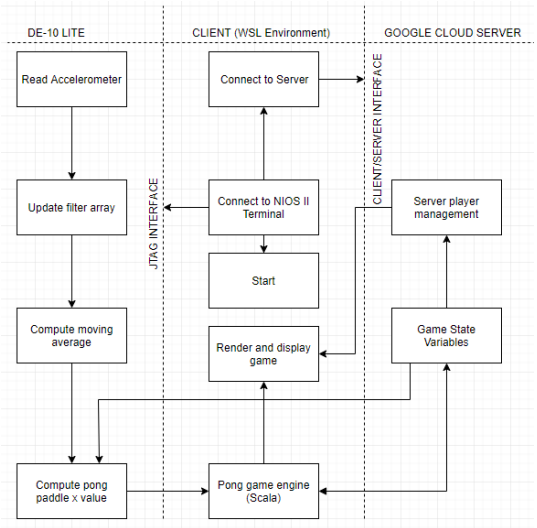

Figure 1. System Architecture

Server

Erlang, a highly specialised programming language designed for building and monitoring multi-nodal networks and distributed applications, was chosen to implement the server to take full advantage of the Open Telecom Platform, which offers a stable connection between nodes of distributed systems with powerful message passing and monitoring features built in [1], as well as the crash tolerance and supervision system which would allow our application to automatically come back online following a crash. Whilst writing the game itself in a native language such as Rust or C++ and calling it from Erlang using 'Native Implemented Functions' would have provided a slight performance gain, these performance gains did not offset the risk of crashing the entire system such that it had to be manually restarted in the event of the game crashing.

As the server was set up as an OTP node, the clients would also need to be set up as OTP nodes to make full use of the benefits of the platform. While Erlang could have also been used to implement the client, the IO modules are not well suited for a console based game and the immutability of objects means making minor changes to the buffer would have used a larger amount of computing power than was necessary. As the Java Virtual Machine's console handling is well suited to our needs while also being an OS and system architecture independent platform such that we could run the client on both Windows and Linux, the JVM implementation of Scala was chosen to gain access to its powerful "Process" library to control nios2-terminal [2] and its pattern matching abilities to more easily process the messages received from the server.

FPGA

WE NEED SOMETHING HERE

The design of the system in Quartus was kept relatively simple, the only notable exception when compared to the system from Lab 3 being that off chip memory was allocated to allow for a larger C program to be stored and run on the board. This is reflected in the resource usage, which was slightly decreased from that in lab 4.

	Resource	Usage
1	Estimated Total logic elements	5,088
2		
3	Total combinational functions	4100
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	2320
2	-- 3 input functions	1098
3	-- <=2 input functions	682
5		
6	Logic elements by mode	
1	-- normal mode	3849
2	-- arithmetic mode	251
7		
8	Total registers	3016
1	-- Dedicated logic registers	3016
2	-- I/O registers	0
9		
10	I/O pins	47
11	Total memory bits	64896
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	iCLK_50~input
16	Maximum fan-out	3015
17	Total fan-out	29044
18	Average fan-out	3.88

Figure 2. Lab 4 Resource Usage

	Resource	Usage
1	Estimated Total logic elements	5,416
2		
3	Total combinational functions	4368
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	2462
2	-- 3 input functions	1183
3	-- <=2 input functions	723
5		
6	Logic elements by mode	
1	-- normal mode	4117
2	-- arithmetic mode	251
7		
8	Total registers	3200
1	-- Dedicated logic registers	3200
2	-- I/O registers	0
9		
10	I/O pins	185
11	Total memory bits	64896
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	MAX10_...~input
16	Maximum fan-out	3199
17	Total fan-out	30801
18	Average fan-out	3.73

Figure 3. System Resource Usage

Implementation

Approach

The following approach was taken to implement the design:

1. Create a Google Cloud Platform server which could handle connections between clients.
    - Test this by having a simple UI-less program which could be interacted with by multiple users
2. Create a UI on the client program which would aesthetically display the data recieved from the server
    - Test this by capturing key-presses with an AWT Frame & sending this to the server as the paddle movements
3. Create the desired filter to smooth the movement of the paddle in C.
4. Alter the client to accept data from the FPGA rather than the keyboard
5. Test the final design

User Interface & Cloud Server Implementation

User Interface

As stated in the design brief, all IoT systems via the FPGA will have to receive and process accelerometer data, hence to sufficiently test and optimise the game interface whilst this code was being developed, we created a temporary method to obtain input data in the form of an AWT frame with a key monitor. Pressing the up key would move the paddle up by a given amount & the down key would move the paddle down by the same amount. After the new position of the paddle was calculated, a message was sent to the client application relaying the change in position.

On receiving an output from the FPGA, the client would send a message to the server in the form of an Erlang tuple containing the node name of the client, which allowed the client to be uniquely identified, and the data received from the FPGA. On receiving a new game state from the server, it would update the buffer before clearing the console and printing it to the screen. This gave the illusion of the contents of the console changing. The other messages received from the server were communicated straight to the board once they had been translated into the appropriate format: a control code followed by the score or new speed of the ball.

Server

The game's server is comprised of a standard Erlang application [3]. The supervision tree comprises of 2 modules: 'pong game' and 'pong server'.

- 'Pong server' deals with network communication with players as well as controlling the tick rate of the game, which implemented with a timer which is created on load. It receives messages from the timer and clients and acts accordingly depending in the form of the message; each message will be processed by performing an asynchronous cast to the game or, in the case of a tick from the timer, with a synchronous call to the game to get the game state which is then sent to the player, followed by a cast to update the game state. To reduce the potential for malicious attacks on the server, the origin of a tick is confirmed to be the timer as a unique reference is sent along with the tick message, and the node name of each player is known only to that player and the server.
- 'Pong game' is implemented with the 'Generic Server' behaviour, keeps track and updates track of the game state. It updates the game state by moving the ball and players then checking for

goals and bounces following asynchronous casts from the Server module and sends the game state to the server following synchronous calls. When it updates the game state, it applies safeguards intended improve playability, such as not allowing the user to move out of the game buffer. The Game module also contains functions to rotate the game state such that each player is sent the game state as though they are on the left. This makes it easier to implement the client and board systems as they can always assume that any local systems only modify the state of the player on the left wall.

Creating the desired filter for filtering accelerometer data in C.

The best way to control the paddle was to use the angle of the accelerometer to calculate the rate of change of the paddle: a greater angle would result in faster movement up or down. This would then be filtered and smoothed to give the impression of real movement of a paddle. The filter was a size 128 equally weighted moving average filter with weights $W_n = \frac{1}{128} \forall\ n \in [0, 127]$. After further review it was clear that this could be heavily simplified, by keeping a running average where the last value in the array is subtracted and the new value is added to an accumulator, resulting in only 2 additions per calculation rather than 128.

Altering the Server/ UI to accommodate input data from the FPGA

The outputs to the server from the AWT and the JTAG Interface were identical, so changing the values was relatively easy. A `nios2-terminal` process was created and IO captured; the newlines were removed from the output & the resulting data was then formatted & sent to the server.

References

- [1] Fred Hebert, "The Hitchhiker's guide to concurrency", in *Learn You Some Erlang For Great Good!* [Online]. Available at: https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#concepts-of-concurrency. [Accessed 29 March 2021].
- [2] École Polytechnique Fédérale, Lausanne, Switzerland. *Scala Standard Library - scala.sys.Process*, École Polytechnique Fédérale [Online]. Available at: https://www.scala-lang.org/api/current/scala/sys/process/index.html. [Accessed 29 March 2021].
- [3] Ericsson AB, Stockholm, Sweden. Erlang — Applications" in *OTP Design Principles: User's Guide, Version 11.2*. [Online]. Available at: https://erlang.org/doc/design_principles/applications.html. [Accessed 29 March 2021].