Crappy Car Simulator

1st Teonezcayotl GutieRuiz
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
tmgutier@mit.edu

2nd Jose H. Cerritos Arevalo
Department of Physics
Massachusetts Institute of Technology
Cambridge, MA, USA
joseca@mit.edu

***Abstract*—This paper presents the design and implementation of a driving simulator utilizing a Field-Programmable Gate Array (FPGA) platform. The simulator integrates an Inertial Measurement Unit (IMU) for steering input, a pedal system with an Analog-to-Digital Converter (ADC) for velocity control, and outputs the simulated environment through an HDMI interface. The FPGA-based approach allows for real-time processing and control, providing an immersive and responsive driving experience.**

***Index Terms*—FPGA, Driving Simulator, IMU, ADC, HDMI**

I. Introduction

Motivated by the increasing demand for realistic virtual experiences in the field of simulation and training, driving simulators have gained popularity in both entertainment and professional applications. Traditional driving simulators often rely on complex software simulations running on powerful computing systems. In this paper, we propose an innovative approach by implementing a driving simulator on an Field-Programmable Gate Array (FPGA), leveraging its parallel processing capabilities to achieve low-latency and high-performance real-time control.

II. Project Outline

Ideally, the user is able to interact and play the game through the use of a steering wheel, pedal, and button inputs. The user uses the steering wheel to control where the car is, the pedal to control the speed, and the button to control the difficulty. Obstacles are procedurally generated and move down the screen. The rate at which these are generated and the speed at which they move are controlled by the difficulty, or button presses. Additionally, the car will occasionally break down at higher difficulties, and this will require a quick time event to 'fix' the car to return to driving. The quick time event involves a sequence of button inputs on the FPGA board to complete.

III. Physical Construction

The driving simulator system is comprised of four main physical components: the AMD-based Spartan 7 Urbana FPGA Board [1], the MPU 9250 [2] Inertial Measurement Unit (IMU), a Pedal [3], the AD7705 [4] Analog-to-Digital Converter (ADC), and an HDMI connection from the FPGA Board to a monitor.

A. IMU

The MPU 9250 is an IMU that produces 3-axis gyroscope, 3-axis accelerometer, and a 3-axis magnetometer data, all of which are 16 bit and conveyed MSB first, that can then be conveyed using either the Inter-Integrated Circuit (I2C) bus interface connection protocol or the Serial Peripheral Interface (SPI) protocol. For the purposes of this project, we decided to utilize the SPI protocol for easier implementation and the fact that the ADC also utilizes SPI.

In order to interface the IMU with SPI, communication begins with the main device (the FPGA board) asserting the Chip Select line with an active low, signaling the IMU to listen; the FPGA then sends a 1MHz clock signal down the Serial Clock (SCLK) line, and data is transferred simultaneously across the Main Output, Secondary Input (MOSI) and Main Input, Secondary Output (MISO) lines. The FPGA sends a command or register address on the MOSI line, and the IMU responds with the requested data on the MISO line. The MPU 9250 has a set of registers that control its various functionalities and store sensor data, such as I2C_IF_DIS (bit 4 of the USER_CTRL register 106) which disables the I2C module and puts the serial interface in SPI mode only.

For the purposes of this project, we only need to calculate the angle of the IMU to act as a steering wheel, so the FPGA gathers the High and Low Bytes of the gyroscope x and y directions and then calculates the angle based on a simple integration approach. This data is then converted into a 2-bit "right, left, or still" signal based on angle thresholds to determine whether the car is moving to the right, left, or staying still.

B. Pedal and ADC

The Pedal itself is a Hall-effect device that collects pedal angle position data without contact; it is powered with 5V and outputs 1V to 4V based on the 0 to 30 degree position angle change from compression. Since this outputs an analog voltage, we decided to utilize the AD7705 ADC to convey the voltage value digitally to the FPGA utilizing the SPI protocol in a similar manner to the IMU. Since both peripherals utilize SPI, we are able to share the MOSI, MISO, and SCLK lines, while only having separate Chip Select lines going to each apparatus, thus removing the need for additional wires. In the project implementation, the FPGA oscillates between gathering pedal and IMU data.

The AD7705 itself is a 16-bit Sigma-Delta ADC which involves oversampling the analog input and feeding the result through a feedback loop. The ADC also provides a status bit that indicates when the conversion is complete, so the FPGA monitors this bit to determine when to read the converted data. The FPGA then takes in this 16-bit voltage data and converts it into 16-bit velocity data which corresponds to the speed at which the obstacles come at the player.

IV. Graphics

The aesthetic of the game should ideally be retro/4 bit to fit the style of gameplay. To display the graphics onto the screen, we are using Block RAMs (BRAMs) and an SD card. The SD card will be used for the start and end screens since those are not required to be updated as quickly as the sprites during the gameplay. The gameplay will use BRAMs for the obstacles and character as well as the side images to give an effect of moving forward.

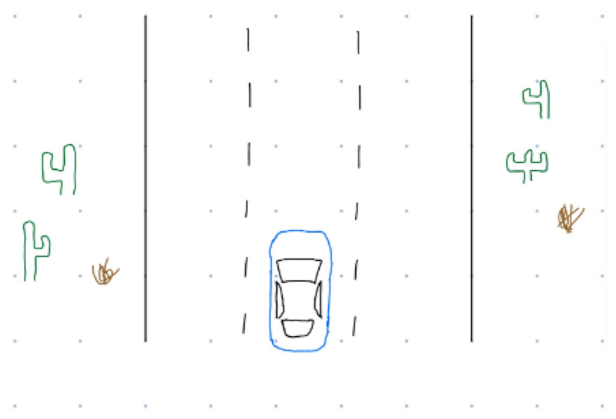

Fig. 1. Model of what Gameplay would look like

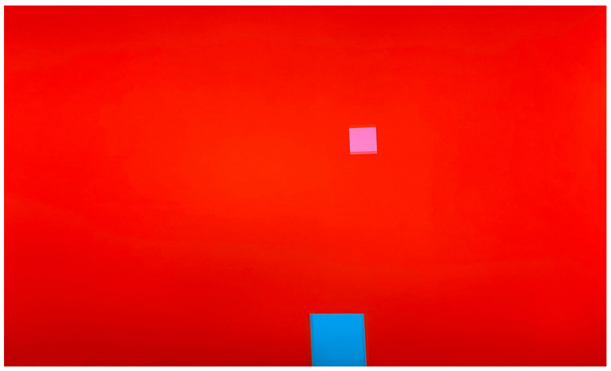

Fig. 2. Sample Gameplay

In Figure 2, the blue square represents the player car, and the pink square represents the obstacle. The blue square is able to go up, down, left, or right, and the pink square location is randomly generated at the top of the screen and then moved vertically towards the bottom.

A. Sprite, Background, and Track Storage

Due to the limited on-chip resources of the FPGA, the Urbana board comes with built-in BRAMs that can be used to store and access sprite data efficiently. The following structured approach was utilized to allocate memory efficiently:

- Color Palette: the color palette incorporates 256 colors, represented by 24 bits per color (8 bits for each RGB value), so the BRAM would be of size 24x256 which results in approximately 6.2 kbits of storage.
- 8x8 Blocks: Since storing and writing to each pixel on the screen would utilize too many memory resources, Image Sprites and the Background would be composed of 8x8 groups of pixels, aptly named “blocks,” that can be stored on a single BRAM; in other words, Image Sprites, the Background, and Track would be formed by smaller, modular image sprites in order to conserve memory on the FPGA. There would be, at most, 64 unique blocks, so the BRAM width would be 8 bits wide for the palette address and (64x8x8) bits deep, resulting in about 32.8 kbits of storage.
- Image Sprites: As of the writing of this paper, we have 6 unique image sprites in mind: a tree, shrub, three possible choices for the player car, and one enemy car. These sprites would be composed of 16 blocks, so each sprite would be stored in a BRAM that is 12 bits wide for the block address and (6*16) bits deep, resulting in roughly 1.2 kbits of space.
- Background and Track: The screen resolution is 720x1280 pixels, so it can therefore be composed of 14,400 blocks, meaning the Background and Track would be a BRAM of size 12x14,400 bits, utilizing about 172.8 kbits of storage.

This particular scheme utilizes roughly 213 kbits of the total 1 Gbit of memory on the FPGA, which allows for efficient storage and retrieval of graphical data within the constraints of available FPGA resources, ensuring optimal performance for the driving simulator and allowing for further expansion in either layer of the scheme.

B. Graphics Module

The Graphics Module of the project incorporates several key features to enhance the visual experience; dedicated BRAMs are assigned for the Palette, 8x8 Blocks, and the composite of Background, Track, and Sprites. Most notably, the BRAMs for Background, Track, and Sprites employ addresses that reference 8x8 blocks, which, in turn, point to the color palette. The design of the sprites, including obstacles and the car, follows a retro style aesthetic. Moreover, a unique aspect of the module is the option for players to switch between 3rd person (top-down view) and 1st person perspectives, adding an extra layer of complexity to the gaming experience; however, the first-person view has not yet been implemented. Additionally, a pause function can be implemented for extended user control during gameplay; in addition, the inclusion of a Quicktime function pauses the screen, allowing for user inputs to continue on in the game. These design elements collectively contribute to a dynamic and customizable graphical interface for the game, enhancing both aesthetics and user interaction.

V. Game Logic

The game uses a Finite State Machine (FSM) to keep track of the difficulty. The initial state has no obstacles being generated so as to give the user a chance to get a feel for the game. The later stages have more obstacle generation and random instances of the car breaking down triggering quick time events.

To represent this in Verilog we have a register routed to 3 bits to encode varying difficulties of game states. This register is able to be incremented to the next difficulty by pressing the button one on the FPGA board. This register is fed into a gameplay module which handles sprite generation for obstacles, and movement for the car. The module procedurally generates the obstacles by taking in the game state to decide how frequent obstacles should be generated and how fast they should move. The obstacles are generated at different parts of the track at random by using a Linear Feedback Shift Register (LFSR) module as an input. The car is controlled by switches on the FPGA as of the writing of this paper, but later will be controlled by the steering wheel and pedal when they are interfaced.

We had the idea of encoding the game state to show the start screen and end screen. Encoding the game state this way allows for effortless switching between screens as well as prevent the gameplay module from doing unnecessary calculations while on these screens. This will also help when we implement the quick time events and pause the game, where the FPGA would just have to switch to another screen.

The gameplay module will also handle collision checking since it updates the sprites movements every new frame in. If there is any overlap between the player sprite and obstacle sprite, the game state will change to trigger the game to stop playing and display a new screen: game over. To achieve this, we have an if-else chain with priorities. The first priority is the rst_in, then it would be the start screen game state, pause game state, or end screen game state to prevent unnecessary calculations from happening. After these are checked, then it would be the various gameplay states, with different difficulties.

To make it clearer, the game state will be enumerated as follows:

TABLE I
Game State

Bits	State
000	Start Screen - No gameplay
001	Gameplay
010	Gameplay with obstacle generation
011	Gameplay with faster obstacle generation
100	Gameplay with obstacle generation and quicktime events
101	Quicktime event screen
110	Quicktime event screen
111	End Screen

Quicktime events were created by making a timer and having another module check for completion of the event. The quicktime events are triggered by the lfsr_in input to simulate a random jump to a quicktime event. We had to adjust the if statements that checked the input such that they were not triggering every second by checking the lfsr_in for more of its bits. We created a quicktime module that takes in signals to check if it was active, and FPGA inputs to check if the action was completed. The quicktime event module also has a select signal which can be expanded on in future iterations, but, at this point, the module only has two options that require different FPGA inputs to create some sort of variety. From there, we had to make a timer module to see if the quicktime event was completed within the allotted time. To do this, we made a nested counter and adjusted to see what would be a good amount of time. The quicktime and timer modules are both activated when the lfsr_in inputs are triggered when in the game state bits value 100.

Moreover, we incorporated a scoring function into the game dynamics. The player's score is visibly displayed on the seven-segment display and undergoes updates every few frames. The scoring mechanism is tied to the current game state, introducing variability in scoring based on gameplay conditions. Specifically, at the start of the game and during the initial game difficulty, the score remains fixed at zero. At the next edifficulty level that introduces slow, random obstacle generation, the score increases at a certain rate. Subsequently, as the game difficulty level increases, the score update increases in rate as the obstacle generation increases, intensifying the competitive aspect of the game. During testing, we observed that this scoring system added an extra layer of challenge and engagement to the gaming experience.

The implementation of the gameplay module presented some challenges, particularly in handling button inputs to facilitate changes in the game state. In the beginning, we considered making the game state an input to the gameplay module, intending to complete the logic for incrementing the game state and holding it at a specific value at the top level. However, this approach proved less intuitive and introduced complications, particularly in having multiple writes to the game state in the same clock cycle, leaving us to abandon the idea. Subsequently, we explored an alternative option by checking for the btn_pulse in every game state and incrementing the game state from there. During testing, though, we encountered an issue where despite confirming the functionality of the button pulse, we observed no visible changes. Upon investigation, we pinpointed the problem to the unsynchronized nature of btn_pulse and nf_in operating on different clocks. To address this, we adjusted the priority in the game, utilizing a signal to retain the value of the btn_pulse and checking this held value to determine whether the game state should be incremented. Despite concerns about potential screen glitches, thorough testing revealed no adverse effects on the video output. Beyond these challenges, the gameplay module acts as a finite state machine, with the game state dictating the displayed content on the screen and influencing various gameplay mechanics. Notably, we incorporated a timer module activated during the end game state, looping back to the start screen game state and enabling the finite state machine to operate in a cyclical, and therefore recurrent, manner. This design choice enhances the overall coherence and replayability of the game.

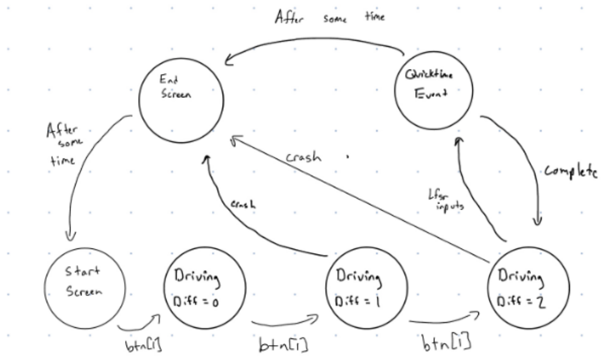

Fig. 3. Gameplay Mechanics Finite State Machine

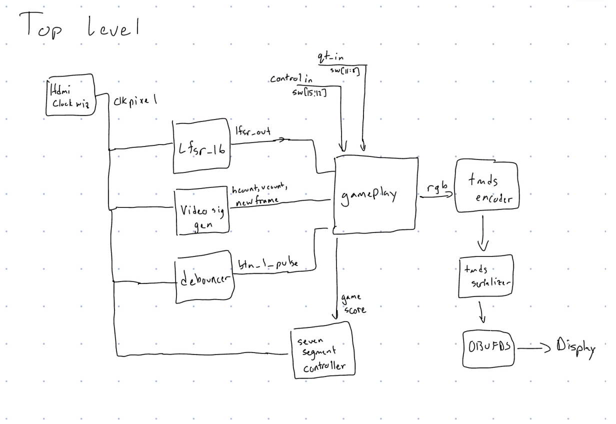

Fig. 4. Top-level Block Diagram

VI. Evaluation

A. *Project Commitments and Goals*

At the start, our main goal was to set up a basic game state and make the gameplay work. We wanted to use a top-down view and include physical devices as controllers for some parts of the game; although we managed to implement the former, we found some difficulty in the latter. Despite our efforts, getting the physical peripherals working didn't quite pan out as we hoped.

B. *Physical Peripheral Reflection*

If we were able to redo the physical peripheral implementation to our project, we would have focused more on asking for help in the synthesis and execution of the communication between the peripherals and FPGA, rather than spending hours to no avail on the same problem.

C. *Graphics Reflection*

If we had the chance to redo our graphics, we would have preferred to start off with an SD card to store Start and Game Over images. This choice stemmed from the challenges we faced when setting up the BRAMs, as pipelining proved to be quite resource-intensive, and we encountered limitations in the early stages of the project in terms of memory usage.

D. *Game Logic Reflection*

If we were able to redo game logic, the preferred method would have been to start by modularizing the code from the beginning. Instead of creating a lengthy module, which turned out to be susceptible to easy bugs and errors throughout the pipeline, breaking down the code into more manageable and independent modules would have been a more effective strategy. This would have enhanced the overall robustness of the game logic and minimized the likelihood of encountering issues in the development process.

E. *Future Implementations*

Other than completing our project commitments, we wanted to add a function to turn off the sprites in the game so that we could add more obstacles at higher difficulties. Right now, in order to hide the sprites, we send them to the front and back porches of the raster screen. We also wanted to add multiple lives that would be shown on screen and use the filter idea from previous labs to indicate when there was a collision. We also wanted to make the quicktime events more engaging with more variety, and different inputs needed besides the switches on the FPGA. We wanted to also experiment more with the gameplay to see what parameters provide the best gameplay for the user. Another possible idea was to perhaps add a drifting mechanic given the steering wheel input, which would heighten the experience and improve the overall enjoyability of the simulator.

VII. Code Base

The code base can be found here.

VIII. Credits

Teonezcayotl focused on the Graphics and Game State Module. Jose initially focused on the physical peripheral implementation, but then pivoted to helping with the Graphics and Game State after he could not get it to work in the interest of time. Teonezcayotl also focused on quantitative evaluation and producing images, while Jose focused on writing the reports and creating figures. Group 32 gives special thanks to Darren for all the help over the semester, and Joe Steinmeyer for being such a great Instructor!

References

[1] "Urbana board," RealDigital, https://www.realdigital.org/hardware/urbana (accessed Nov. 21, 2023).

[2] "MPU-9250," TDK InvenSense, https://invensense.tdk.com/products/motion-tracking/9-axis/mpu-9250/ (accessed Nov. 21, 2023).

[3] Amazon.com: MgcSTEM Variable Speed Pedal Electric pedal foot switch ..., https://www.amazon.com/MgcSTEM-Variable-Accessory-Replacement-Connectors/dp/B0BFQNSVR6 (accessed Nov. 21, 2023).

[4] "AD7705," AD7705 Datasheet and Product Info | Analog Devices, https://www.analog.com/en/products/ad7705.html#product-overview (accessed Nov. 21, 2023).