Verilog Game: Unicorn Explosion

A Simple Game that Illustrates Complex Programming Concepts in Hardware Design Languages

Changcan (Benjamin) Hu
Department of Electrical and Computer Engineering
California Polytechnic Univerity, Pomona
Pomona, California
changcanhu@cpp.edu

Andrew Swanson
Department of Electrical and Computer Engineering
California Polytechnic Univerity, Pomona
Pomona, California
akswanson@cpp.edu

Macade Walker
Department of Electrical and Computer Engineering
California Polytechnic Univerity, Pomona
Pomona, California
mjwalker@cpp.edu

Burhan al-Estwani
Department of Electrical and Computer Engineering
California Polytechnic Univerity, Pomona
Pomona, California
bualestwani@cpp.edu

Yashika Malik
Department of Electrical and Computer Engineering
California Polytechnic Univerity, Pomona
Pomona, California
ymalik@cpp.edu

***Abstract*—A semester project that covers theoretical concepts from Verilog lecture and practical knowledge from the lab to build a circuit that implements sequential and combinational features.**

I. Introduction

For our class project, we decided to take a simple, already existing video game and reimplement it in verilog. We picked the unnamed dinosaur game that is displayed in Google Chrome when the user doesn't have an internet connection. Any implementation of this game would require using all of the primary features of verilog, including both sequential and combinational logic. The desire to stylistically emulate both the dinosaur game and arcade games informed many design decisions for the project. We titled our game by picking the first two words that came to mind.

While we used the concepts and features of an existing game, we didn't have access to the codebase, requiring us to design the program from the ground up. In our initial meeting, we decided that the game would require six basic modules. The game needed a map generator, a physics engine, a display decoder, a score recorder, an audio engine, and a top module to tie everything together. The modularity of Verilog made it easy to split up the work, allowing all of us to work on modules separately and combine them at the end. These six modules were all planned from the beginning to use certain inputs and outputs in ways that would make the compatible. The five modules instantiated by the top modules are presented here.

II. Map Generator

A. Purpose

For the game to work, we needed some function that creates the various obstacles for the player. One thing we could have done was just make a large binary number and have that represent our map of obstacles, but that would not only be extremely tedious to actually type into the computer, it would also keep outputting the exact same map if they played multiple times, which could become boring for the player.

B. Function

We use a 16 bit register as a counter randomly choose which output will be on the map. We decided that we wanted 3 different types of outputs for each section of the map: empty, bottom block, and top block. This means we need 2 bits to represent each spot on the map. There are 8 segmented displays on the FPGA, so we need 16 bits to represent the map.

C. Implementation

To break down the Always block that starts on line 94, every time the positive edge of the divided clock occurs, we shift the map to the left and add the next section to the right 2 bits of the map. The majority of the block is spent figuring out what that next map section is supposed to be. Basically, the next part of the map is whichever part of the preload we have not put into the map yet, and when there is not any preload left, the portion of the seed the program is currently on is what determines what the next preload is. (EECS)

D. Problems

One of the problems we initially had with this design was that it had to be fair. For example, if there were a bottom block followed by a top block on the next display, then the player would lose whether or not they jumped. This drove us to make it so the seed decides between a number of different modules that all include spaces and a block, which ensures there are enough spaces between blocks so the game does not feel too unfair. We got the idea from a game called Spelunky, which is a game where the caves explored are randomly generated in a similar way, where each portion of the map is chosen between several different possible blocks which guarantees the game is fair and no impossible levels are created. In addition, there was another challenge occurred during loops, where no jumping button blocks has created by the seed. In event of the problem, player has no reason to jump, and the jump input is the only way the seed would change. To fix this. Therefore, we had the program check to see if a bottom block on the map before it has determined a preload. If there is not a bottom block, the preload becomes a bottom block, which guarantees the player will jump and the seed will change.

III. Physics Engine

A. Introduction

This is a synchronous sequential circuit that uses combinational logic and multiple flip flops, where the outputs depend on the past behavior of the circuit, as well as on the present values of inputs, where clock signal is used to control the operation. In this case. This module is a type of Finite State Machine (FSM)

B. Purpose

The physics engine is responbile for taking the jump button input and converting that to a semi-realistic jump for the player. It compares this value with the incoming map tiles to determine if the player successfully jumped over the block. If the jump was successful, an output is asserted to the score module to increase the player's score, and if not, the player is recorded as having died, prompting the game to enter its end state.

C. Logic

For jumping, in order to prevent the user from continuously jumping in the air to avoid any low blocks easily, we control the duration of jump. Pressing jump only works while the user is not already jumping, and once the player jumps, they stay in the air for two clock cycles before returning to the ground.

For collision detection, we simply compare the player's current position with the value of the leftmost block on the map. If the physics engine detects a collision, i.e. that the player has not jumped while a low block passes, or that the player has jumped while a high block passes, it notifies the rest of the modules that the player has died.

IV. Display Engine

A. Graphics

Obviously, one cannot have a video game without video, so graphics were high on our priorities list. It is possible to connect our FPGA board to an external graphics output, but we decided that it would be simpler to utilize the onboard 7 segment display for graphics output. We found that it was easy to convey all of the required information, including a welcome message, gameplay graphics, and the player's score.

B. Limitations of the 7 Seg Display

The main problem to solve with regard to displaying outputs is that the eight seperate 7 segment displays only have eight inputs and eight outputs. For each segment, every anode is connected together, as is typical for 7 segment displays. However, all of the segments in each position--for example, all of the lines at the top--are connected across the entire display. At first examination, it would seem impossible to show different things on different displays, as there is no way to individually show different characters on different displays at the same time. However, while there is a limitation on what can be shown simultaneously, there is a solution that bypasses this by merely appearing simultaneous

C. Scanning Display Controller Circuit

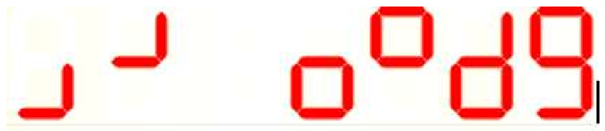
The recommended solution is to alternate rapidly between showing individual characters on each display. [2] By alternating fast enough, rather than seeing individual characters shown in sequence, the viewer will instead see the entire display showing a cohesive image. The biological principle behind this optical illusion is persistence of vision, in which the eye, having a limited refresh rate, which groups stimuli that happen in a very short time period together. Since only 1/8th of the displays are on at any given time, the display appears dimmer, but since the designers anticipated this kind of pulse width modulation, the displays are very bright to begin with

D. Artwork

The artwork for the game was limited, due to the use of the onboard display, but we found that we were able to convey all the needed information. For the welcome screen, this meant that all of the letters had to be in a particular case, leaving the welcome message reading "PrESS StArt" but in practice, this didn't impair readability.

PrESS StArt

For designing our character, we quickly realized the additional limitation of needing artwork that would fit in either the top half or the bottom half, so that we could convey the appearance of the character jumping. The obstacles were rendered as boxes. The final artwork used in the game appears as this, with the character on the ground, the character jumping, the lower box, the upper box, and the character jumping over the lower box, then running under the higher box



Finally, the score screen simply uses the standard versions of the 7-segment display artwork for numbers, along with some hexadecimal digits.

V. Score Engine

A. Purpose

The score engine is meant to count the output from the Physic engine as player starts playing the game. At the end of the game, score_engine will pass the register score array to the display engine in order to display the final score of the player for the round.

B. Logic

The score engine shares the same clock as the rest of the system, so that the score will react appropriately to any in-game events. It operates base on an output of the physics engine, which is called score_in. In addition, the Difficulty array is inputted to evaluate the score differently for different difficulties. The higher difficulties award a higher score to the player. To ensure that the score is properly reset, it also takes the start input from the top module.

C. Behavior

The main logic runs on the positive edge of the clock. The score engine awards one point for an empty tile successfully navigated, and ten points for sucessfully passing a block. These scores are multiplied by the difficulty level, increasing the point values.

D. Problems

The score module was relatively straightforward to develop, with the only design misstep being that the score originally kept incrementing after the player died, awarding them for the non-existent progress they were making. The other hurdle to overcome was converting number to a decimal format, which was eventually accomplished in the display module.

VI. Audio Engine

A. Purpose

A game's immersiveness increases as more and more media is added to it, and we decided that audio would be an interesting addition. Within the scope of this project, we decided that we would aim for basic sound effects, rather than trying to implement a full soundtrack. Rather than implementing a sample playback system, which has been successfully implemented on FPGA, we decided instead to aim for more of the CPU driven audio style typical of the 1980's video game. We felt that this would both be an interesting programming challenge and a stylistically appropriate choice.

B. History of Computer Audio

CPU driven audio is a term retroactively applied to the earliest forms of computer audio, dating back before the use of dedicated audio hardware. [3] The earliest home computers would be equipped with simple speakers, but since every audio element had to be driven from the CPU's main (and only) thread, the sounds they generated were limited to beeps and clicks. Later machines began implementing FM synthesis and sample playback to enhance audio quality, but we decided to stick to the basics.

C. Digital Audio and the Square Wave

The fundamental idea behind any digital logic circuit, particularly the FPGA, is that everything in a computer is binary. Audio, at least, the kind the people listen to, is fundamentally analog, so without some analog circuitry to convert ones and zeroes generated by the FPGA into continuous-amplitude waveforms, it seems impossible to generate an analog signal. However, the workaround used by early computer interface designers, and the method we adopted, was to simply pass the digital output through an amplifier and interpret the digital signal directly as audio. By switching back and forth between 1 and 0 on the output port, we found that we could generate a square wave and, with come calculations, make sure that it sat within the audible range. Then, by connecting a speaker of suitably low power requirements, we could directly output simple audio. As the Nexys 4 DDR board used for the project has a headphone jack soldered onto it, it appeared that this portion of the project would be relatively straightforward.

D. Sound Design

We decided that there would be two important sound effects in the game, one of which being the sound when the player character jumped, and the other when the character died. Taking some inspiration from the Super Mario games, some of the pioneers in block jumping, we decided that the audio should be played as a rising tone to imply a jump, and a falling tone to imply failure. This could be implemented behaviorally in Verilog with little difficulty. We simply calculated out the number of clock cycles for the audio to be the right frequency at the start of the sound and stored this in a register. The system tracked time in a separate counter and waited for the counter to match the predetermined value. When a match occurred, the timer was reset, the audio was flipped, and the register to hold the period of the audio tone was modified to change the tone for the next cycle. By trying out

E. Hardware Limitations

Upon synthesizing our initial implementation, we were initially met with only a DC pop in response to our input, rather than the square waves we had programmed. After some research, we found that the audio system implemented on the Nexys 4 wasn't as simple as we had expected. Rather than simply running the FPGA audio through an isolator and into an amplifier, the board uses a system to transmit analog voltages via Pulse Width Modulation, or PWM. Through use of an analog filter, it converts the duty cycle of the waveform to analog voltage. This seems like a great workaround for the problem of outputting analog audio from a digital circuit, but the rest of the work is useless without some kind of workaround.

F. Implementation

Our modulator took the generated audio stream as an input and used the FPGA's 100 MHz clock as a timing reference. Once every 16 clock cycles, the circuit pulls the audio stream down to 0 for one clock cycle, passed the rest of the audio straight through for the next 14 cycles, then on the 16th cycle, it pulls the stream up to high. The resulting bitstream is 0000_0000_0000_0001 when the audio stream is at zero, and 0111_1111_1111_1111 when the audio stream is high. In this way, the square wave passes through nearly unchanged, but still goes through the amplification system allowing us to hear the audio without any external circuitry.

VII. Conclusions

The project, though it had its hurdles, was ultimately a success. We were able to implement an existing game, doing almost all of the code from scratch. The new game was not only functional, but, in our opinion, entertaining. The project was a great learning experience, both in terms of Verilog programming and in working with a project team.

References

[1] Eecs.umich.edu. (2018). *debounce*. [online] Available: https://www.eecs.umich.edu/courses/eecs270/270lab/270_docs/debounce.html [Accessed 4 Dec. 2018].

[2] Digilent Inc, *Nexys 4 DDR Reference Manual.* Digilent Inc, 2018. [Accessed November 23, 2018]. Available:

[3] https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

[4] D Murray, *How Oldschool Sound/Music Worked.* [Video]. The 8-Bit Guy. Youtube.com, *5 October 2015,* Available: https://www.youtube.com/watch?v=q_3d1x2VPxk [Accessed October 2018]