ECE385

Spring 2024

Final Project: Geometry Dash

Tony Liu (zikunl2)
Ansley Tsai (ansleyt2)
TA: Gene Lee (genel2)

Contents

1 Introduction

For our ECE 385 final project, we decided to recreate the popular game Geometry Dash using System Verilog, Vivado, Vitis, and the Urbana FPGA. Geometry is a music side-scrolling platforming game where the player controls a character through a map while avoiding deadly obstacles. The game features 24 original levels by the developer RobTop, along with millions of custom levels created by millions of players worldwide. For our implementation, we remade the first level of the game called Stereo Madness, which contains all the fundamental mechanics that the game utilizes. This includes a scrolling map, obstacle collision, jumping and flying physics, animations, and sound effects/music. Ultimately, we were able to implement most of these features in our project using the resources available on the Urbana Board and Vivado.

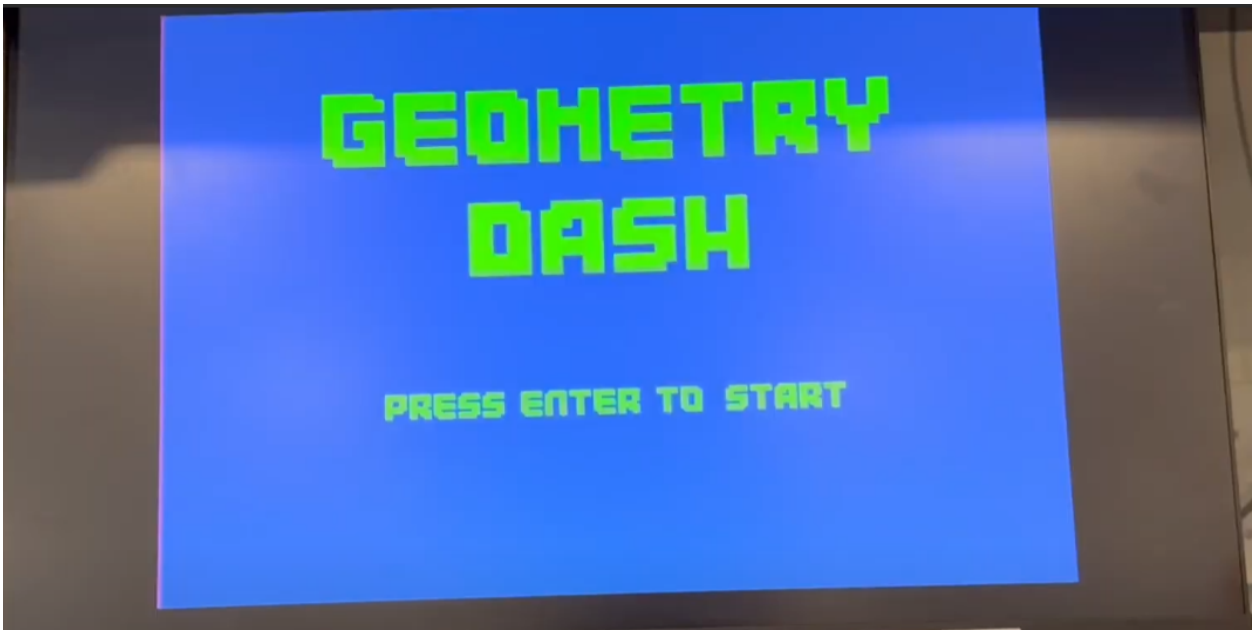

Figure 1: Title Screen of our Geometry Dash implementation.

2 Overview of the Geometry Dash Hardware System

Geometry Dash is built on fundamental topics that we learned from ECE 385, specifically from Lab 6 and 7. The game contains a MicroBlaze-based SoC to control user and data input, GPIO/USB for keyboard input, a PWM controller for sound, and an HDMI controller IP which houses the video display hardware along with all the game logic.

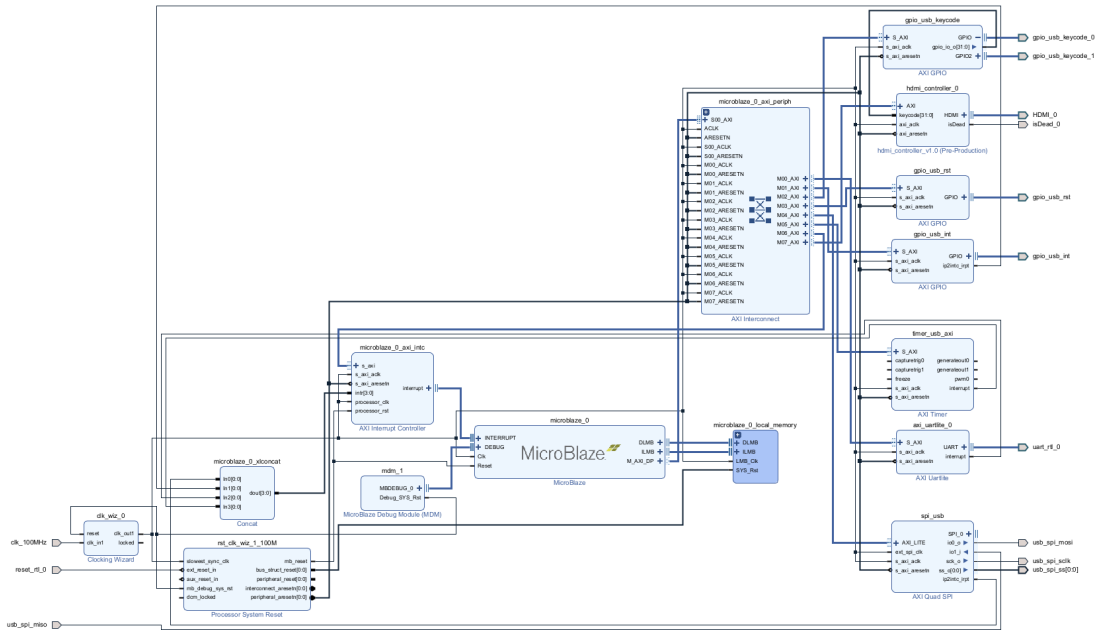

Figure 2: Top level Block Diagram of Geometry Dash Hardware System.

3 Description of Geometry Dash HDMI Controller

The HDMI controller in Geometry is an IP block that contains all the game logic and visual displaying hardware, based on the HDMI controller introduced in lab 7. However, the Geometry Dash HDMI controller features far more modules compared to lab 7's IP block, consisting of the following:

1. A full map in BRAM based on the first level of the game "Stereo Madness" that can be modified through the AXI bus.
2. A frame buffer in BRAM that stores the current frame of the game
3. A cube/ship module determining player physics.
4. Collision system between the player and map.
5. VGA controller and VGA to HDMI for video output.

6. Multiple ROM files to draw appropriate graphics/animations on the screen.

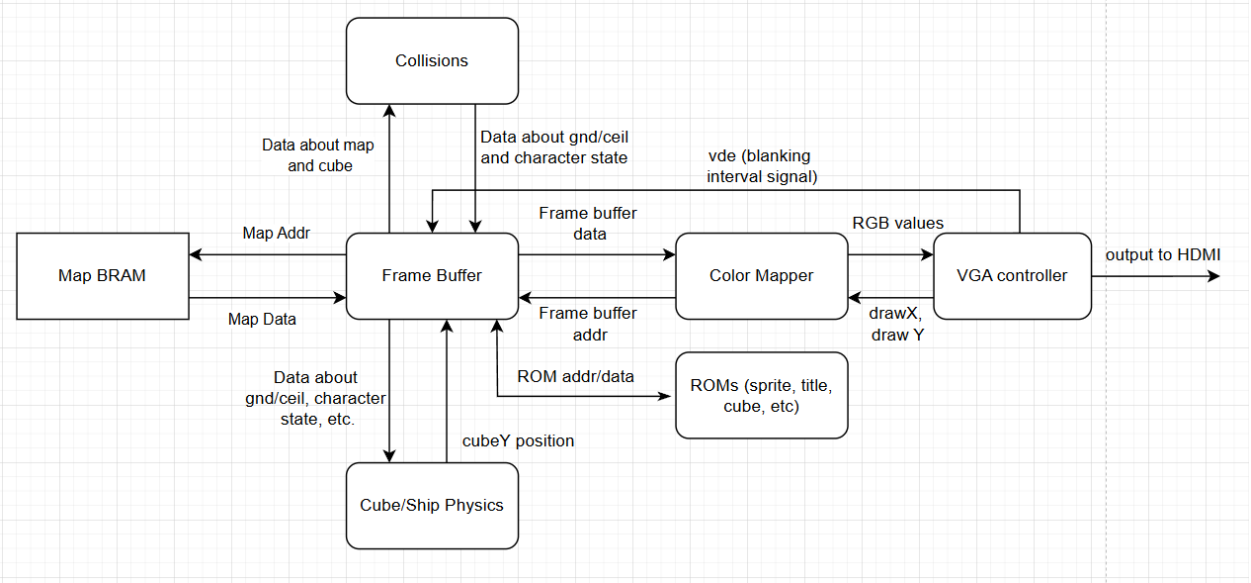

Figure 3: High-level block diagram of the Geometry Dash HDMI controller.

3.1 The Map

The level map in our Geometry Dash implementation is a 1024x12 grid in BRAM, where each grid consists of one of many sprites, such as a background, block, spikes, etc. Each grid is represented with 8 bits (1 byte) in BRAM, storing an index number referring to the desired sprite in a separate ROM file. Since the AXI bus width is 32 bytes, each memory location or "word" holds 32 bytes, or 4 grids. With 8 bits per grid, up to 256 distinct sprites can drawn on the map, although only five different sprites were used in our final iteration. For example, if the first grid in the map has value x00, it refers to the background, while a grid with a value of x0e refers to a spike in ROM.

To allow for easy modification of the map level and colors without needing to re-synthesize and generate bitstream, the map BRAM and color palette can be entirely written through the AXI bus in the exact manner as in lab 7. Reading and writing through the AXI bus is as follows:

READING:

1. The master initiates the transaction by providing a ARVALID address, indicating readiness to receive data as well as a RREADY representing readiness to receive data.
2. The slave acknowledges readiness to receive the address with ARREADY.

3. A successful handshake between the master and the slave occurs, indicating that the address has been successfully received by the slave. ARVALID and ARREADY will only last for one cycle.
4. In the Read channel, the slave responds by supplying the requested data and validating its presence on the data channel with RVALID.
5. The transaction completes with both parties confirming data transfer readiness, allowing for the de-assertion of RREADY and RVALID.

WRITING:

1. The master initiates the writing process by asserting AWVALID and WVALID indicating the address and data are ready to be sent out. BREADY is also asserted by the master, indicating it is ready to receive a response.
2. Then, waiting for the slave to output AWREADY and WREADY to acknowledge overwriting readiness.
3. Once both pairs of signals are present, the handshake process occurs.
4. Last, BVALID will send out a signal notifying the success of the transaction.

With AXI read/write added, instantiating the map data is done by simply creating a struct the points the start of the map BRAM, and writing C code in Vitis to change the value of the certain byte position. For example, to set the top leftmost grid in the map to the background, one asserts:

```
hdmi_ctrl->VRAM[1*1024 + 0] = 0x00;
```

Repeating this code format for all the other grids generates a complete map for a 90-second level in the game. This also applies to the color palette as well to change the color of certain blocks in the game. The final level design thus included over 4000 lines of AXI write code, which fortunately was mostly generated by a Python script. Our final implementation featured a color-changing background, which was achieved by setting the background color to an incrementing counter storing a 12-bit RGB value. Since keyboard input was done in the loop, the counter was simply included in the loop to be incremented periodically.

Figure 4: Example section of the map drawn in pixilart.com. Python code processed the exported PNG to help write the majority of the AXI write code.

3.2 Frame Buffer

While an entire level can now be stored in BRAM and modified through AXI, there still needs to be a way to scroll through the entire map and output a certain section of it every frame at a time. This is where the frame buffer comes in, which is essentially another BRAM that stores the current frame of the game.

In lab 7, colored text display was done entirely through the Color Mapper module, where for each pixel update, it accesses a certain point in memory and determines the correct RGB output. However, with the pixel clock at 25 Mhz, this approach would struggle to meet timing with a fast scrolling background and multiple levels of combination/sequential logic. Additionally, it becomes an increasingly more difficult task to manage as additional elements such as the character or text are added to the screen. Thus, the frame buffer seeks to address these issues by copying in all the appropriate pixels into its BRAM during the blanking intervals of the VGA controller. This way, there is a longer period of time in which logic can be executed.

For Geometry Dash, the frame buffer stores a byte for every pixel in a 256x192 resolution screen. With each grid being 16x16 in resolution, it means that the frame buffer displays a 16x12 grid section of the map every frame. Each byte refers to an index value in the color palette, which aforementioned can be modified through the AXI bus. When copying the pixels into the frame buffer, a separate horizontal and vertical counter, similar to the ones in the VGA controller, are used to keep track of which memory location in the BRAM to write into and are incremented in raster order every other clock cycle to account for latency when reading from map BRAM. This data is then sent to the Color Mapper, which determines the 12-bit RGB values for every pixel that the VGA/HDMI controller will process.

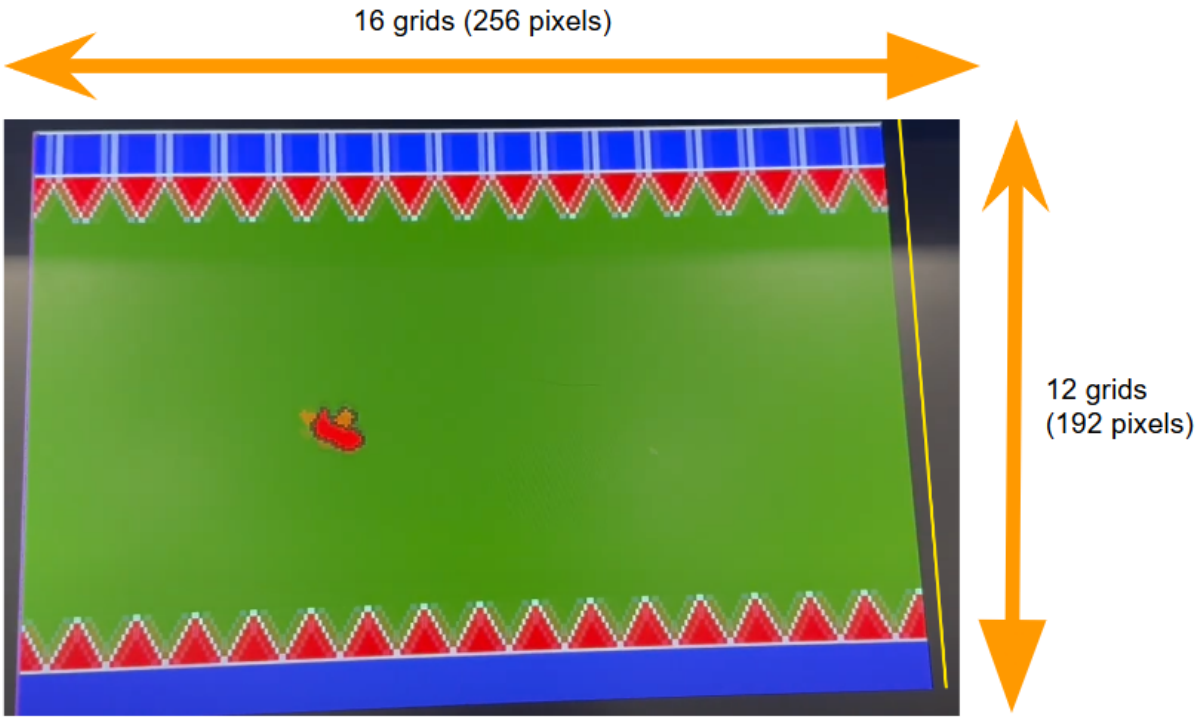

Figure 5: Frame of a ship flying in Geometry Dash. Data in this frame is stored in the frame buffer and outputted to HDMI.

To simulate background scrolling, there is a curX variable that stores the current X position of the character in the map. This variable is incremented by 3 pixels every frame and is used to access the correct memory location in the map. Suppose curX is pixel #256. This means that the frame buffer should start copying pixels from byte address 256/16 = 8 (word #2) in the map, and all the bytes 16 grids to the right and 12 grids down. With this logic, the game can now scroll through the entire map in around 90 seconds.

For other elements such as the attempt counter or the character, it is simply drawn by creating additional signals that check if the horizontal and vertical counter are within the location of those elements. Then, when copying the pixels into the frame buffer, if those signals are high, the frame buffer will prioritize writing those elements' data instead of the map.

3.3 Cube/Ship Physics

The first level of Geometry Dash features two types of control: cube (jumping) and ship (flying). While the map scrolls at a constant horizontal speed of 3 pixels/frame, a separate module is still needed to determine the Y position of the character to simulate jumping/flying

motion. It takes in keyboard input along with other signals about the current state of the map to determine whether or not to enable cube or ship controls. The cube jumps about 40 pixels when space is pressed, while the ship moves up and down up to a speed of 3 pixels/frame depending on if space is held down.

For this to occur, speedY and accelY signals were created to determine the next Y position of the character. This is then stored in an alwaysff block that updates every frame or when the player dies. Additional logic is also considered when the player hits the ground or the ceiling, in which the all vertical motion must seize.

3.4 Collision system

In Geometry Dash, the character's state varies based on which edge of the grid it collides with. For example, if the character collides with the left side of the block or hits a spike, it should die and the level should restart. However, if the character collides with the top side of a block (i.e. the ground) it should land on that ground and continue moving. Differentiating between these two edge cases for every non-background block in the level poses a challenging problem, and after a week of iterating and debugging, we arrived at this final approach: to store the highest ground and lowest ceiling of the horizontal position character is currently in.

To achieve this, data pertaining to the highest ground and lowest ceiling needs to be stored in the map BRAM. While the total map displayed on the screen is a 1024x12 grid, there is an extra hidden row in the BRAM that stores the necessary ceiling/ground data. Just like with the grids, 8 bits were used to store all this information for every horizontal grid position, along with the signal of whether the current ground/ceiling is a spike:

CEIL spike	CEIL[2]	CEIL[1]	CEIL[0]	GND spike	GND[2]	GND[1]	GND[0]

Figure 6: Bitwise organization of the ceiling/ground data. Note that only 3 bits are used for the ground and ceiling despite the map being 12 blocks high. This was decided upon as a memory-saving measure as well as realizing that the ground/ceiling will likely never become too high/low.

Another observation about the cube/ship is that it is always within the bounds of two grids. Thus, data from both grids must be read at the same time for every frame. Originally, we had one map BRAM to read from, so we first read from the left grid the cube is on and then the right grid. However, it resulted in multiple timing issues, so instead a second BRAM copy of the map was added so that both the left and right grid could be read simultaneously, while still allowing the map to be written through AXI. Then, determining which ground the cube should land on is done by simply taking the minimum of the two ground values,

and the same applies to the ceiling but with the minimum.

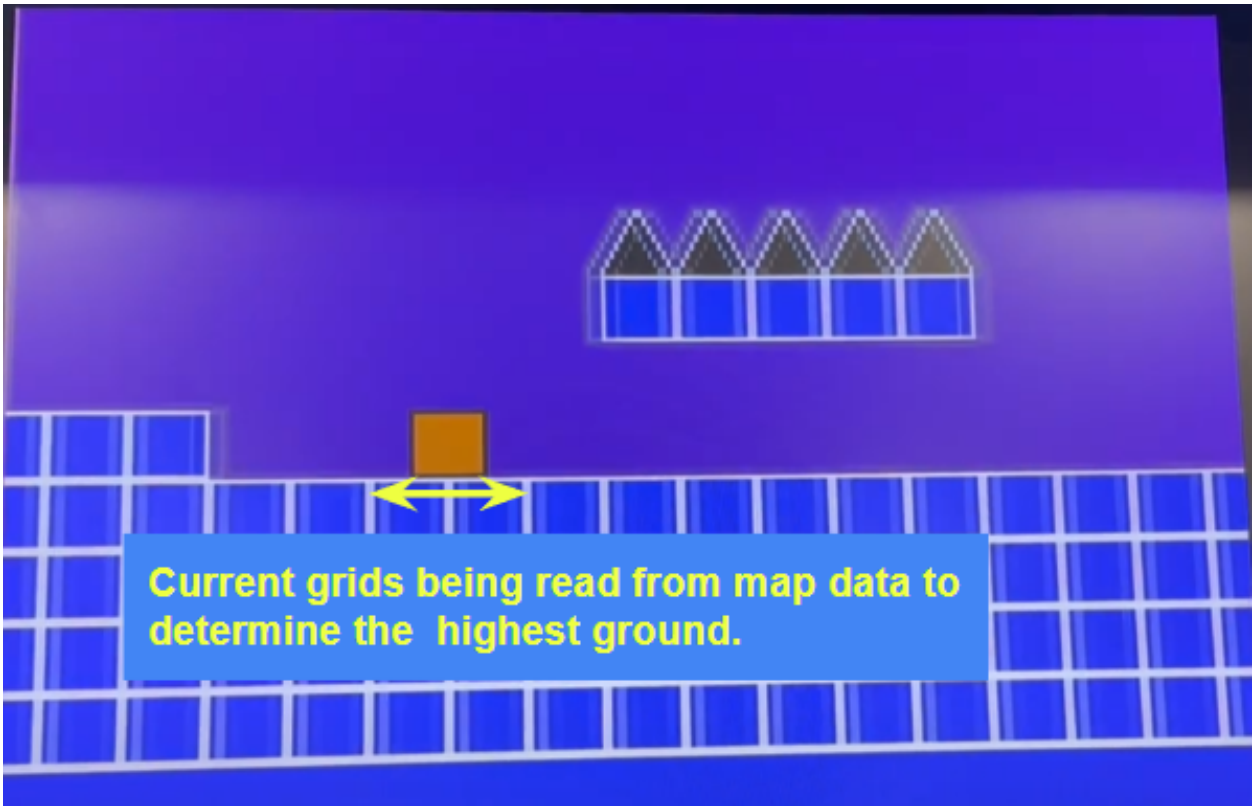

Figure 7: Annotated picture of how the highest ground is determined.

With data on both grounds/ceilings known, it can now be easily determined whether or not the character crashes into the side of the wall or lands on the ground. For crashing into a wall, the module checks if the character's Y position is below the right grid by 3 pixels; if so, it will die, and the level restarts. However, if the character is not dead and has fallen below the highest ground, it means it has hit the ground, so its vertical speed reduces to 0 and its Y position resets to that ground value.

For spikes, the leftmost bit for the ceiling and ground data determines whether or not the current ground/ceiling is a spike. Spike collisions are essentially a much stricter version of ground collisions: if the character falls below the ground and that ground happens to be a spike, the character will immediately die.

Throughout this entire explanation of collision, one major conclusion that is determined from the collision module is whether or not the character is dead or not. This information is in fact a signal that is outputted from the collision module and sent throughout all the

pertinent modules to trigger a level restart, cube motions reset, and a death sound effect to play. With collision logic added, Geometry Dash now becomes a playable but challenging game to experience.

3.5 VGA controller

VGA is a type of video display controller based on how a Cathode Ray Tube (CRT) Monitor operates. It outputs a maximum video resolution of 640x480 pixels. In a CRT, an electron beam shoots through a phosphorescent layer in raster order (left to right, top to bottom).

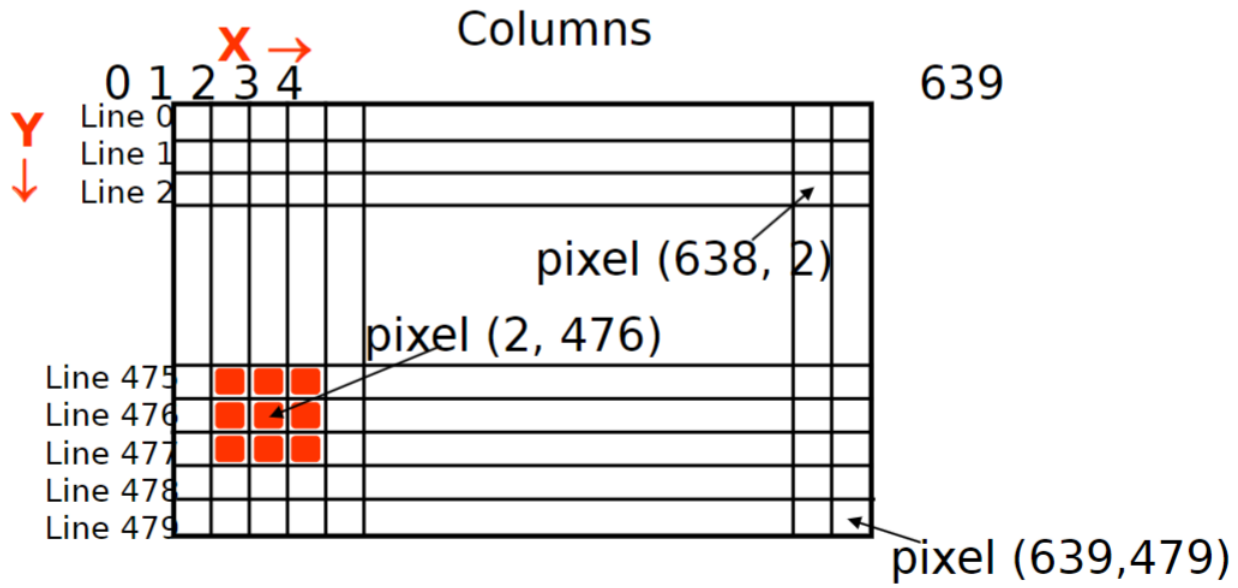

Figure 8: Row column map for VGA. When pixel drawing, the "electron beam" goes from left to right pixel by pixel, and top to bottom row by row.

To achieve this kind of operation in Vivado, a VGA controller module is created to keep track of the drawX and drawY coordinates of the pixel being drawn, similar to how an electron beam would aim at the screen. The controller also takes into a pixel clock of 25Mhz, meaning that an individual pixel would update in 1/25Mhz seconds.

The controller outputs two important signals, horizontal sync (hsync) and vertical sync (vsync). HSync signals when the controller has complete the drawing of another row (i.e.) when the drawX has reached the maximum X coordinate of 640) and resets DrawX back to 0. Vertical sync does the same for DrawY when an entire screen is done updating, or when DrawY reaches 480 pixels. With the pixel clock operating at 25MHz, it takes around 30 microseconds for each row to update, and multiplying by 480 rows gives a screen refresh time of 16.67ms, or around 60Hz. In practice, there are pauses (called front and back porch) between when the hsync and vsyncs pulse are set high, so a pixel counter up to 800 is used

for horizontal timing and up to 525 for vertical.

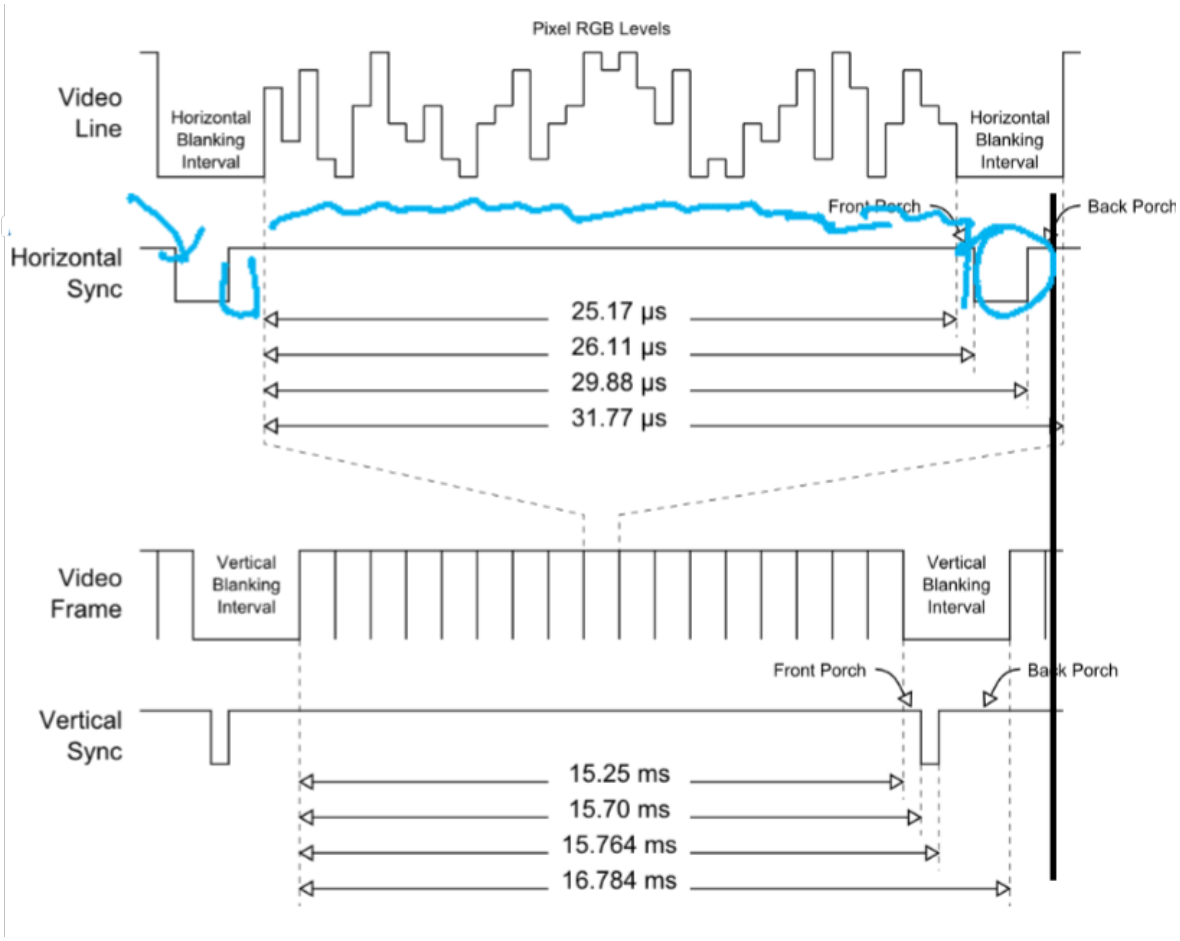

Figure 9: Timing diagram for VGA.

For Geometry Dash, the resolution of the game stored in the frame buffer is 256x192 pixels, but each pixel is copied over four times to create a final resolution of 512x384. The color mapper module then takes each value from the frame buffer based on drawX and drawY, indexes into the color palette, and then determines the final 12-bit RGB value for each pixel displayed on the monitor.

3.6 Graphics/Animations using ROM

To make player movement more fluid and complete, animations of the cube and ship were added. Cube and ship animations were done by creating individual frames of the cube and ship in an online pixel drawer, and them exporting it as a PNG. To turn the PNG into SystemVerilog ROM file (similar to fontROM from lab7), a Python script was written to convert specific RGB values from the image into a 4-bit number used to index into the color palette.

This allows for much easier creation of the ROM files used in the program. For the cube, if it is currently jumping, a counter storing the current frame index increments to simulate rotation, and for the ship, its current Y speed determines its tilt angle and the appropriate frame.

Figure 10: Frames used to animate cube rotation in the game, drawn on pixilart.com.

For the attempt counter, an 8-bit signal was added to store the number of deaths, and it increments during the positive edge of the isDead signal. This 8-bit signal is then converted to a 2-digit decimal number through combinational logic, and it is then copied into the frame buffer at the beginning of the level.

Using the online pixel drawer and Python script, the title screen and word "attempt" were easily generated and added as ROM files, and combined with the color palette greatly minimized the number of LUTs used on the FPGA.

Below is some example code on how the title_rom.sv file was generated from PNG input:

```
from PIL import Image
from collections import Counter

import numpy as np

outFile = open('title' + '.txt', 'w')

filename = "title"

im = Image.open(filename + ".png") #Can be many different formats.
im = im.convert("RGBA")

for y in range(im.size[1]):
    outFile.write("256'b")

    for x in range(im.size[0]):
        pixel = im.getpixel((x,y))
```

```
            r, g, b, a = im.getpixel((x,y))
            if (a==0):
                outFile.write('0')
            else:
                outFile.write('1')
        outFile.write(",\n")
im = im.close()
outFile.close()
```

4 Description of Audio

Playing out realistic audio from the hardware device is usually associated with digitized data that mimics the effect of the analog signal data. Pulse-width Modulation is a waveform with rectangle waves of various duty cycles. This data is transmitted through a low-path filter that generates an asymptotic voltage value as shown below in the figure.

In this section, we'd like to walk through the designs for playing out sound effects/music.

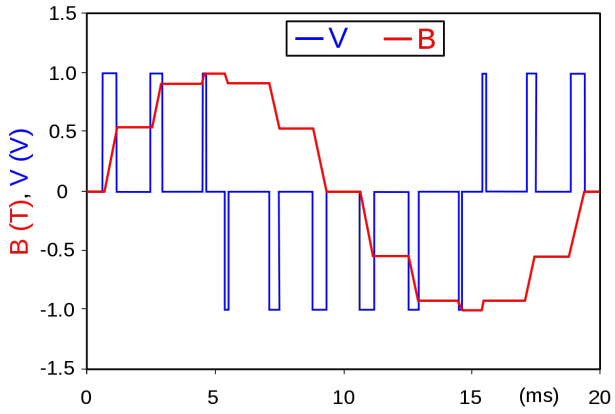

Figure 11: PWM Waveform

4.1 Memory Storage

First, we acquired the binary sample data file (.wav) using Audacity which supports exporting files with 8-bit data width in 8kHz sampling rate. Then, the binary data is modified through a third-party COE converter that gives out a COE file. Last but not least, a Python script was written to properly adding the prefix of the data so that it's in the correct syntax. We'll address more about how the binary data is stored as the OCM's size is too small compared to the sheer amount of data the music has.

- DDR3

  The pipeline for using a DDR3 to store the data and play out with the audio jack is slightly complicated. The binary data will be allocated within the .text data section of the DDR3 memory by manually setting it on Vitis. We have created an unsigned 8-bit integer array in the C file, in which we could send the data out per requested. This involves when and how the data's sent out. Instead of using the AXI-bus, we have come up with a solution with using GPIO and FIFO to control the data flow.

  A First-In-First-Out IP with an independent clock setup allows managing different read and write frequencies. As the figure below shows, there's an individual read and write clock, where in our design, we used $83.333mHz$ as the write clock because that's the frequency Microblaze's operating at, and a read clock at $8kHz$ that pertains to the speed we want to pull out data. Moreover, there are useful flag signals such as "almost_full" and "almost_empty" signals that inform the extent of occupation of the FIFO memory.

  With the aid of FIFO, we decided to control the data flow by the flag signals so that whenever almost all the data has been drained out, we'd send in more data through GPIO into the FIFO. Similar to the reading scenario.

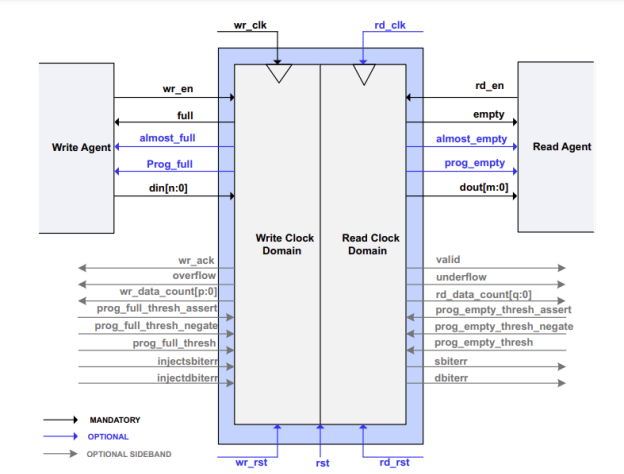

Figure 12: FIFO

Nevertheless, this approach did not perform well as we have yet to come up with a solution for the reading latency in DDR3 as suggested in the figure below. Moreover, the software also has instruction latency that requires more sophisticated algorithm to resolve.

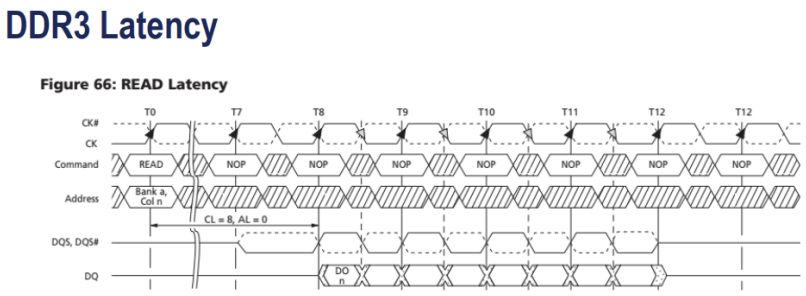

Figure 13: DDR3 Read Latency

- BRAM

  The other approach that performs much better is with the BRAM. We initialized the BRAM with the pre-configured COE file. The rate at which we're accessing the data within the BRAM is actually sent out from our PWM generator that operates at the sampling rate. This implementation has allowed us to playback the death sound effect with a minor high-frequency noise.

4.2 PWM Generator

The logistics for the PWM generator's operation is that every cycle, we have a sample width, where

$$DutyCycle = \frac{SampleWidth}{Period}.$$

Meanwhile, there's an internal counter that counts up to the period value, in which the period of one cycle is determined by $\frac{100mHz}{8kHz}$, the ratio between the system clock rate and the sampling rate of the data. While the counter is smaller than the pulse width, we'll output active high for the pwm signal, vice versa.

At the same time, we have also succeeded in generating a sampling rate clock signal by toggling the rising and falling edge with respect to the counter value.

5 Descriptions of .SV modules

Modules in Top Level:

Module: mb_gdash_top.sv
Inputs: Clk, reset_rtl_0, gpio_usb_int_tri_i, uart_rtl_0_rxd, usb_spi_miso
Outputs: uart_rtl_0_txd, hdmi_tmds_clk_n, gpio_usb_rst_tri_o, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, pwm

Description: This is the top level of the Geometry Dash MicroBlaze system. It contains all the submodules and IP blocks required to display the game on a monitor and play sound.
Purpose: To combine all the modules so that the MicroBlaze system can be exported as a bitstream file to be programmed into the FPGA along with some additional high-level software on Vitis.

Module: mb_gdash
Inputs: clk_100Mhz, reset_rtl_0, uart_rtl_0_rxd, gpio_usb_int_tri_i
Outputs: hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0]hdmi_tmds_data_n, [2:0]hdmi_tmds_data_p, uart_rtl_0_txd, isDead, usb_spi_ss,
Description: This is overall IP block for the MicroBlaze processor and HDMI controller. It contains all the modules required for the Microblaze, such as clocking wizard, AXI, local memory, etc, along with the HDMI controller.
Purpose: To instantiate the MicroBlaze and its I/O interfaces and connect it to the HDMI text controller for outputting VGA/HDMI to a monitor.

Module: MicroBlaze
Inputs/Outputs: clk, reset, interrupt, debug, DLMB, ILMB, AXI
Description: This is the MicroBlaze processor IP block that is included in Vivado.
Purpose: To handle low-performance tasks such as user interface, and data input/output to perform various functions based on how it is programmed. In this lab, it was used to send and receive signal between the hdmi text controller via the AXI bus to display text on the monitor.

Module: microblaze_0_local_memory
Inputs/Outputs: clk, reset, DLMB, ILMB
Description: This is a local memory for the MicroBlaze processor
Purpose: To store instructions and data in the MicroBlaze.

Module: clk_wiz (MicroBlaze)
Inputs: reset, clk_in1
Outputs: clk_out1, clk_out2
Description: This is a clocking wizard to takes in a 100MHz clock as input and outputs and same 100Mhz clock for use in the MicroBlaze. It is automatically added by Vivado when initializing the MicroBlaze.
Purpose: To manage the 100Mhz clock used in the MicroBlaze.

Module: rst_clk_wiz_1_100M (Processor System Reset)
Inputs: aux_reset_in, dcm_locked, ext_reset_in, mb_debug_sys_rst, slowest_sync_clk
Outputs: bus_struct_reset, _interconnect_aresetn, mb_reset, periphreal_aresetn, peripheral_reset
Description: This is an reset IP block automatically added by Vivado to handle the system reset of the MicroBlaze processor. It takes in the clock signal and the reset RTL signal from the FPGA and restarts the entire MicroBlaze program.
Purpose: To reset the MicroBlaze when the reset on the FPGA board is pressed. In this lab, it reset the screen on the monitors and the text displaying programs.

Module: microblaze_axi_periph (AXI Interconnect)
Inputs: S_AXI, uart_rx, s_axi_aclk, s_axi_aresetn
Outputs: M00_AXI, M01_AXI, M02_AXI
Description: This is the AXI Interconnect that contains all the master and slave AXI buses. It takes in the MicroBlaze as the master and has seven different ports to feed into various slave ports such as UART and the HDMI text controller.
Purpose: To connect the master (MicroBlaze) to all the slave devices to allow for communication between all the various devices.

Module: axi_uartlite_0
Inputs: S_AXI, uart_rx, s_axi_aclk, s_axi_aresetn
Outputs: S_AXI, uart_tx, interrupt
Description: This is the AXI Uartlite IP block. It runs at a baud rate of 115200 and transmits/receives data serially between the MicroBlaze and the FPGA using uart_rx (receiver) and uart_tx (transmitter).
Purpose: To handle UART communication to allow for print statements on the Vitis Serial Terminal.

Module: microblaze_axi_intc (AXI Interrupt Controller)
Inputs: S_AXI, s_axi_aclk, s_axi_aresetn, [0:0] intr, processor_clk, processor_rst, interrupt(bus)
Outputs: interrupt
Description: This is a main interrupt controller for the MicroBlaze. It uses the AXI bus to determine whether or not the MicroBlaze should be interrupted when it receives a signal.
Purpose: To handle interrupt logic and feed interrupt signals directly to the MicroBlaze to alert the processor of any new signal from the HDMI text controller.

Module: timer_usb_axi (AXI Timer)
Inputs: S_AXI, capturetrig0, capturetrig1, freeze, s_axi_clk, s_axi_aresetn

Outputs: S_AXI, generateout0, generateout1, pwm0, interrupt.
Description: This is an AXI timer IP block for the USB driver to keep track of the polling rates of different USB devices. It uses the AXI bus, the clk, reset, and interrupt to communicate with the MicroBlaze and SPI. In this lab, the generous, capture, and pwm0 signals were not utilized.
Purpose: To manage timeouts and polling rates for different USB devices so that inputs and readouts from the USB peripheral are handled consistently. For example, a mouse can have a polling rate of 10ms, so the timer keeps track of when 10 milliseconds have elapsed before sending a readout to the MicroBlaze.

Module: gpio_usb_int (GPIO Interrupt)
Inputs: S_AXI, uart_rx, s_axi_aclk, s_axi_aresetn, gpio_usb_int
Outputs: S_AXI
Description: This is an interrupt GPIO port that connects with the MAX3421E USB transceiver. When the MAX3421E sends a interrupt signal, it will send it through the AXI bus in order to interrupt the MicroBlaze to read input.
Purpose: To communicate with MAX3421E in sending interrupt signals to the MicroBlaze

Module: gpio_usb_keycode
Inputs: S_AXI, uart_rx, s_axi_aclk, s_axi_aresetn
Outputs: S_AXI,, gpio_usb_keycode_0, gpio_usb_keycode_1
Description: This is a GPIO keycode port that processes keyboard input into a keycode coming from the MAX3421 USB. It has two 32-bit keycode outputs to be able to read and output 8 different eight, although a standard USB keyboard only supports six simultaneous keypresses at once.
Purpose: To output keycodes from the SPI USB peripheral to be used to control the game in HDMI controller.

Module: spi_usb (AXI Quad SPI)
Inputs: ext_spi_clk, s_axi_aclk, AXI_LITE(bus), s_axi_aresetn, usb_spi_miso
Outputs: usb_si_mosi, usb_spi_sclk, ip2intc_irpt
Description: This is a QUAD SPI USB peripheral that handle data transfers between the MAX3421E and the MicroBlaze, using master-in-slave-out (MISO) and master-out-slave-in (MOSI) signals. It also outputs 25Mhz clock that required for the MAX3421E to function. More about how SPI works will be discussed later.
Purpose: To send data to and from USB devices and the MicroBlaze via SPI and the MAX3421E, so that input can be read from the keyboard.

Module: microblaze_0_xlconcat (Concat)
Inputs: [0:0] In0, In1, In2, In3
Outputs: [3:0] dout
Description: This is a concatenation IP block in the MicroBlaze block diagram. It takes in multiple interrupt signals from various AXI modules (GPIO, UART, SPI) and feeds them into the AXI Interrupt Controller.
Purpose: To combines all the interrupt signals from all the peripheral so that it can respond to user inputs from the USB keyboard or FPGA.

Module: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0]hex_seg [3:0]hex_grid
Description: This is a hex driver used to control the segmented displays on the FPGA. hex_seg refers to each segment in a digit, while hex_grid refers to the 4 hexadecimal digits used in the processor.
Purpose: To display the values of the keycodes when a key is pressed on the keyboard.

Module: pwm_generator.sv
Inputs: Clk, reset, en, [13:0] period, [13:0] pulse_width
Outputs: pwm
Description: This is a pwm generator served for the audio outputs. It takes in the pulse_width data and compare it with the internal counter to decide the voltage for the pwm signal. The pwm waveform that it creates drives an analog voltage output through the low-pass filter.
Purpose: To output an 8-bit pwm waveform so that the game has a death sound effect whenever collision's triggred.

Module: clock.sv
Inputs: Clk, reset, en, [13:0] period
Outputs: sample_clk
Description: This is a clock generator that sends out a 8kHz signal that is used in pulling data from the BRAM once every cycle.
Purpose: To assist with updating the pulse_width data of the pwm_generator accurately

Modules in HDMI Controller:

Module: hdmi_controller_v1_0.sv
Inputs: [31:0] keycode, axi_aclk, axi_aresetn, axi_awaddr, axi_awprot, axi_awvalid, axi_wdata, axi_wstrb, axi_wvalid, axi_bready, axi_araddr, axi_arprot, axi_arvalid, axi_rready
Outputs: isDead, hdmi_tmds_clk_n, hdmi_tmds_clk_p, hdmi_tmds_data_n, hdmi_tmds_data_p, axi_awready, axi_wready, axi_bresp, axi_bvalid, axi_arready, axi_rdata, axi_rresp, axi_rvalid, axi_rready
Description: This is a HDMI controller top module for Geoemtry Dash. It communicates as a slave with the Microblaze, and contains all the submodules requires for the game to function. This includes the AXI bus, clocking wizard, VGA controller, VGA to HDMI converter, and Color Mapper. Most of the signals are the same, except that the inputs and outputs from the submodules are different.
Purpose: To combine all the submodules together and allow for communication of signals between all of them.

Module: hdmi_controller_v1_0_AXI.sv
Inputs: [12:0] map_addr1, [12:0] map_addr2, S_AXI_ACLK, S_AXI_ARESETN, S_AXI_AWADDR, S_AXI_AWPROT, S_AXI_AWVALID, S_AXI_WDATA, S_AXI_WSTRB, S_AXI_WVALID, S_AXI_BREADY, S_AXI_ARADDR, S_AXI_ARPROT, S_AXI_ARVALID, S_AXI_RREADY
Outputs: map_data1, map_data2, palette[8], S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RVALID
Description: This is HDMI controller AXI module. It has two 13-bit dual port BRAM modules to hold the entire level and allow for reading from two grids simultaneously. It also has 8 palette registers that store 16 different colors that can be used in the game.
Purpose: To perform read and write operations via the AXI bus and store the data (color, characters) needed in Geometry Dash using BRAM.

Module: BRAM
Inputs: clka, rsta, ena, wea, [10:0] addra, [31:0] dina, clkb, rstb, enb, [3:0] web, [10:0] addrb, [31:0] dinb
Outputs: [31:0] douta, [31:0] doutb
Description: This is the BRAM memory block used in Geometry Dash. It is a true dual port memory block with a write depth of 8192 and data width of 32-bits. It stores four grids per address space, with 1024x12 grids in total. With two ports, it can read/write from both the AXI and the color mapper at the same time. There are two of these BRAMs for simultaneous reading from two grids.
Purpose: To store all grids in the level map and allow for read/writes between the Microblaze or the color mapper.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This is the VGA controller that determines which pixel is to be drawn at a certain clock cycle. It uses a 25Mhz clock, meaning that each pixel changes at a frequency of 25Mhz. With 640x480 pixels outputted, the overall screen refresh rate sums to 60Hz. Horizontal sync and vertical sync are to used to signal a row or screen change, while the current pixel being drawn is outputted to drawX/drawY.
Purpose: To control the monitor output based on VGA standard and draw the correct pixels in a timely manner.

Module: clk_wiz (VGA/HDMI)
Inputs: reset, clk_in1
Outputs: clk_out1, clk_out2
Description: This is a clocking wizard to takes in a 100MHz clock as input and outputs two different clocks of difference frequency. For this design, it outputs a 25Mhz for updating the pixels in the VGA controller and HDMI, and a 125MHz clock 5x TMDS (Transition-Minimized Differential Signaling) for transmitting high-speed serial data in HDMI.
Purpose: To create a pixel and clock and 5x TMDS clocks required for outputting an image in HDMI.

Module: vga_to_hdmi (hdmi_tx_0)
Inputs: pix_clk, pix_clkx5, pix_clk_locked, rst, [C_RED_WIDTH-1:0] red, [C_GREEN_WIDTH-1:0] green, [C_BLUE_WIDTH-1:0] blue, hsync, vsync, vde, [3:0] aux0_din,[3:0] aux1_din, [3:0] aux2_din, ade
Outputs: TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N
Description: This is an imported IP block that converts the VGA signal produced by the VGA controller into an HDMI signal. It outputs differential clocks and data channels that are contained in a HDMI signal.
Purpose: To convert from VGA to HDMI so that an HDMI cable can be utilized to connect the FPGA to the monitor.

Module: frame_buffer.sv
Inputs: reset, axi_clk, frame_clk, vde, keycode, map_data1, map_data2, color_addr
Outputs: map_addr1, map_addr2, color_data, isDead

Description: This is the framebuffer module that processes and stores the current frame of the game. It uses a BRAM to store 256x192 resolution frame, copying over the a specific section of the map. It also handles horizontal scrolling along with drawing of all sprites, characters, and text.
Purpose: To create the current frame of the game that is displayed to the monitor.

Module: Frame Buffer BRAM
Inputs: clka, rsta, ena, wea, [10:0] addra, [31:0] dina, clkb, rstb, enb, [3:0] web, [10:0] addrb, [31:0] dinb
Outputs: [31:0] douta, [31:0] doutb
Description: This is the BRAM memory block used in the frame buffer. It is a true dual port memory block with a write depth of 49152 and data width of 8-bits. Each address spaces stores a number used to index into the color palette in color mapper.
Purpose: To all the color palette index values for every pixel in the game.

Module: cube.sv
Inputs: reset, frame_clk, vde, keycode, title_state, cubeGND, cubeCEIL, cube_ship, spike, isDead
Outputs: cubeY, frame_num
Description: This is the cube module that handle keyboard input and the physics of the cube and ship. It takes in multiple signals about the current state of the game to determine the correct Y position of the character in the map.
Purpose: To process character physics and determine the Y positions of the character to be written into the frame buffer.

Module: collisions.sv
Inputs: reset, updateGND, cubeY, cur_grid, next_grid, cube_ship
Outputs: spike, isDead, cubeGND, cubeCEIL

Description: This is the collision module that processes character collisions in the game. Depending on the character's position, it determines whether of not the character lands on the ground, hits a spike, or crashes into the wall. If it is the latter two, it will set the isDead to high to signal a level restart.
Purpose: To determine whether or not the player lives or dies based on its it collides with certain blocks.

Module: Color_Mapper.sv

Inputs: [7:0] Rdata, [9:0] DrawX, DrawY, [31:0] palette[8] **Outputs**: [15:0] Raddr, [3:0] Red, Green, Blue
Description: The Color Mapper determines the 4-bit RGB values of a pixel based on drawX and drawY from the VGA controller. It reads from the framebuffer and the palette to display the correct frame on the monitor.
Purpose: To set the colors of each pixel so that the right colors and text are displayed on the monitor.

Module: title_rom.sv
Inputs: [6:0] addr
Outputs: [255:0] data
Description: This the title ROM (read-only memory) module that stores the title picture.
Purpose: To store the title picture so it can be written into the frame buffer when appropriate.

Module: cube_rom.sv
Inputs: [8:0] addr
Outputs: [127:0] data
Description: This the cube ROM (read-only memory) module that stores all the different frames for the cube and ship for animations.
Purpose: To store the character frames so they can be written into the frame buffer when appropriate.

Module: attempt_rom.sv
Inputs: [3:0] addr
Outputs: [127:0] data
Description: This the attempt ROM (read-only memory) module that stores the attempt picture.
Purpose: To store the attempt picture so it can be written into the frame buffer when appropriate.

Module: score_rom.sv
Inputs: [7:0] addr
Outputs: [15:0] data
Description: This the score ROM (read-only memory) module that stores numbers 0-9 to counts the number of attempts/deaths.
Purpose: To store the score picture so it can be written into the frame buffer when appro-

priate.

Module: sprite_rom.sv
Inputs: [63:0] addr
Outputs: [255:0] data
Description: This the sprite ROM (read-only memory) module that stores all the different types of blocks used in the map.
Purpose: To store all the different block sprites so it can be written into the frame buffer when appropriate.

6 Simulation Results

To most effectively test the audio data, we have created a test bench to pinpoint the signals. The simulation below is a snippet of our death sound effect playback simulation. As we can see, every $\frac{1}{sample\_rate}$ amount of time, the BRAM sends out a new sample width and the pwm generator outputs a corresponding rectangle wave with the correct duty cycle.

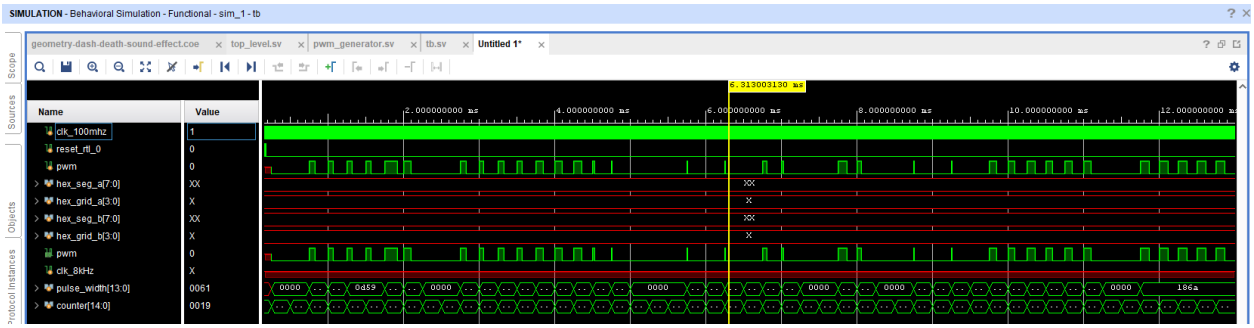

Figure 14: Audio Simulation

Because the game involves thousands of different frames in its graphics, it was unreasonable and unnecessary to spend up to 15 minutes simulating a single frame of the game. However, simulation was still useful in checking that all logic signals were instantiated properly when debugging.

7 Design Resources and Statistics

Usages	Values
LUT	4624
DSP	1
Memory (BRAM)	67
Flip-Flop	3366
Latches	0
Frequency (MHz)	86.4
Static Power (W)	0.078
Dynamic Power (W)	0.443
Total Power (W)	0.521

8 Conclusion

Recreating the popular game Geometry Dash on an FPGA platform using System Verilog, Vivado, Vitis, and the Urbana FPGA board provided a comprehensive learning experience in digital design and FPGA programming. Our successful implementation of the game, including key features such as a scrolling map/framebuffer, collision detection, and audio playback, demonstrates the practical application of all the concepts learned throughout the course.

One of the main challenges faced was the integration of various components into a cohesive system. The need to manage and synchronize between multiple modules such as the HDMI controller, frame buffer, and audio output, required careful planning and optimization. Additionally, the implementation of a robust collision detection algorithm was particularly challenging due to the necessity to balance accuracy and performance. Even the final iteration of the collision system was extremely sensitive to timing changes; one modification in some other part of the project could impact the reliability of the collision system. What's more, the biggest difficulty beneath the audio playback function was the memory storage, where we have tried numerous methods. Avoiding using the AXI-bus for interacting with the DDR3 might have seemed to be a clever and time-saving approach, but the cost is the additional complexity in synchronizing the data flow.

Through overcoming these challenges, the project not only enhanced our understanding of FPGA architecture and System Verilog but also improved our problem-solving and debugging skills. The experience gained from using industry-standard tools like Vivado and Vitis is invaluable and will undoubtedly aid in future electronic design projects.

References

[1] Pulse-width modulation (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/Pulse-width_modulation/media/File:PWM,_3-level.svg (Accessed: 08 May 2024).

[2] Project 7 digital audio player (no date) RealDigital. Available at: https://www.realdigital.org/doc/5d71051961d630e2dfc8026312db0570 (Accessed: 08 May 2024).

[3] RealDigitalOrg (no date) VivadoIP/axi_pwm_1.0/HDL/pwm_core.V at master · RealDigitalOrg/VivadoIP, GitHub. Available at: https://github.com/RealDigitalOrg/VivadoIP/blob/master/axi_pwm_1.0/hdl/pwm_core.v (Accessed: 08 May 2024).

[4] (No date) AMD Technical Information Portal. Available at: https://docs.amd.com/v/u/en-US/pg057-fifo-generator (Accessed: 08 May 2024).