ECE 385

Spring 2024

Final Project

Game: Big Bet, a Modified Temple Run Game on FPGA

By Wendi Wang (wendiw2)

1. **Introduction**

   a) **Basic Functionality of the "Big Bet" Game on FPGA**

   The modified version of Temple Run game, named "Big Bet", on FPGA using SystemVerilog involves creating a System-on-Chip (SoC) on the Urbana Board. The project aims to develop key components such as a scenery generator, color mapper, and VGA controller, alongside essential hardware components for audio storage, and keyboard inputs. The game will feature an "infinite" style with continuous obstacle updates and coins appearing for gaining scores that require user interaction through keyboard inputs. Background music will play throughout the game, and the character's speed will be set to two gears (one faster and the other one slower) for users to choose over time. The end of the game will show the player's score based on the number of coins collected. Also, we will record the highest score for users automatically.

   b) **The Operation of the "Big Bet" Game**

   The operation of the "Big Bet" game, the modified Temple Run game, on FPGA involves an integration of various modules, including those for controlling visual and audio outputs, handling user input, and designing game mechanics such as character movement and obstacle avoidance. The game makes use of a series of logic circuits and modules for generating a random sequence of obstacles as well as coins, controlling the character's motion based on keyboard inputs, and handling score tracking. The game operates in a continuous loop, updating the position of the character and obstacles, and managing interactions based on user input such as keyboard presses. Collision detection and game state management are key components of the operation, allowing the game to provide real-time feedback such as scores and game-over notifications, and coordinating the display of graphical content through HDMI.

   In the game "Big Bet," players begin by pressing "Enter" to start. They control the orange box's motion using the "A" key to move left and "D" key to move right at a slower speed, or "K" key to move left and "J" key to move right at a slower speed. Points are earned by hitting the center of the yellow "coins," while avoiding hitting the boundary or the center of the blue "obstacles" to prevent a "game over" scenario. If the game ends, players can restart by pressing the "Space" key. Both the highest score and the current score an user earning will be recorded automatically.

2. **Written Description and Diagrams of Microblaze System**

   a. **Module Descriptions (Mainly Following Previous Labs' Modules, especially Lab 6 and Lab 7)**

      a) **With postfix .sv files:**

      Module: mb_usb_hdmi_top.sv

      Inputs: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd

Outputs: gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0] hdmi_tmds_data_n, [2:0] hdmi_tmds_data_p, [7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB
Description: This file contains a top-level module designed for integrating various peripherals including USB, UART, HDMI, and VGA. It uses an integrated IP to generate HDMI signals. It can also serve as a bridge between ball motion and USB keycode control. This module interfaces with multiple blocks including USB, UART, HDMI text controller, hex displays, a ball module, and a color mapper.
Purpose: Integrating USB, UART, HDMI, VGA, and additional functionalities within a single top-level file (module).

Module: hex_driver.sv
Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [7:0] hex_grid
Description: This file contains a module of visualizing outputs. To be specific, it contains a module of demonstrating output values on LEDs using 7-segment method. An LED digit on the Urbana Board is divided into 7 segments, with different segment combinations we can express numbers from 0 to F in hexadecimal.
Purpose: To translate hex numbers (from 0 to F) into figures appearing on LEDs.

Module: VGA_controller.sv
Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY
Description: This file contains a module of generating VGA video signals, which requires synchronizing the horizontal and vertical timing of the display as well as managing the drawing coordinates within the screen resolution. It finally outputs several necessary signals for activating the display pixels including x and y pixel coordinates, horizontal and vertical drawing frequencies, etc. It contains internal counters for horizontal and vertical positions to provide proper timing for horizontal and vertical sync pulses. Besides, it also takes reset conditions and display range into consideration.
Purpose: Generating VGA related signals and pixel coordinates for further drawing as well as displaying on the screen after converting to HDMI signals.

Module: ball.sv
Inputs: Reset, frame_clk, [7:0] keycode, [9:0] StoneX, [9:0] StoneY, [9:0]

Stone_size

Outputs: game_over, [9:0] BallX, [9:0] BallY, [9:0] BallS

Description: This module controls the movement of the “ball” (actually drawn as a square box) in the game, responding to user inputs for controlling its motion and handling collisions with stones. It computes the ball’s next position and updates the motion accordingly, ensuring the ball stays within specified boundaries.

Purpose: The module is designed to manage the ball’s position and motion in response to user keycode inputs, ensuring the ball avoids obstacles and the game is played according to the rules.

Module: stone.sv

Inputs: Reset, frame_clk, [7:0] keycode, [9:0] x, [9:0] y, [9:0] size, game_over

Outputs: int rand_num, game_begin, [9:0] BallX, [9:0] BallY, [9:0] BallS

Description: This module handles the motion of the stones in the game, adjusting their positions based on game states and generating random numbers for stone movements. It also determines when the game should begin.

Purpose: The module controls the positions and movements of stones in the game, interacting with other game components to manage the game state and determine the start of a new game.

Module: coin.sv

Inputs: Reset, frame_clk, keycode, [9:0] x, [9:0] y, [9:0] size, game_over

Outputs: int rand_num, int score, [9:0] BallX, [9:0] BallY, [9:0] BallS

Description: This module manages the movement of coins in the game, updating positions based on user interactions and computing scores when the ball and coin collide.

Purpose: The module controls the coins in the game, calculating scores and updating positions based on game states and interactions with the ball.

Module: Color_mapper.sv

Inputs: [9:0] BallX, [9:0] BallY, [9:0] Ball_size, [9:0] StoneX, [9:0] StoneY, [9:0] Stone_size, [9:0] CoinX, [9:0] CoinY, [9:0] Coin_size, [9:0] drawX, [9:0] drawY, [31:0] regs[8], [31:0] q, no_draw, game_begin, game_over, restart

Outputs: [3:0] Red, [3:0] Green, [3:0] Blue

Description: This file contains of module of determining the color to display at each pixel on the screen based on the relative position of the ball and the current drawing coordinates. It also uses a circular boundary check to determine if the pixel falls within the ball’s area or falls within the background area to determine

what RGB values should be drawn. The color_mapper module is responsible for determining the color outputs (Red, Green, and Blue) based on the positions and sizes of the ball, stones, and coins. The module uses the input parameters to set the appropriate colors for the different elements in the game.
Purpose: This module's primary purpose is to manage the game's visual representation by mapping game elements to specific colors. By controlling the color outputs, the module ensures that the game's visuals are displayed correctly on the screen.

Module: font_rom
Inputs: [10:0] addr
Outputs: [7:0] data
Description: This module serves as a read-only memory (ROM) containing font data. It has a parameterized width for address and data, defined by ADDR_WIDTH and DATA_WIDTH respectively. The ROM holds font data for characters encoded in 8-bit patterns. Each character occupies a block of 8 bits within the ROM, allowing for quick access based on the address input. The font data represents characters in a 16x8 pixel matrix.
Purpose: To provide font data for display purposes, enabling the generation of characters on a screen or display device based on input addresses.

Module: hdmi_text_controller_v1_0_AXI
Inputs: 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: S_AXI_AWREADY, S_AXI_WREADY, S_AXI_BRESP, S_AXI_BVALID, S_AXI_ARREADY, S_AXI_RDATA, S_AXI_RRESP, S_AXI_RVALID, regs
Description: This module includes logic for handling AXI4-Lite transactions, memory mapped register selects and write, as well as read operations. The module interfaces with a global clock signal, reset signal, write, and read address/data channels, and various control signals for AXI transactions.
Purpose: To provide an AXI4-Lite interface for accessing registers and handling transactions.

Module: ram_32x8
Inputs: [31:0] d, [11:0] write_address, read_address, we, clk, S_AXI_ARESETN,

slv_reg_wren, C_S_AXI_DATA_WIDTH, register, [3:0] S_AXI_WSTRB
Outputs: [31:0] q, data_out, [31:0] palette[8]
Description: This module implements a 32x8 RAM with dual-port access to complete on-chip memory design. It stores data in a memory array "mem" and a palette array "palette" based on write and read addresses. The module handles write and read operations, considering byte-level enable signals and reset conditions, and outputs the appropriate data through signals "q" and "data_out". The palette is utilized for addresses exceeding 2048 using registers.
Purpose: To provide memory storage and retrieval functionality with dual-port access for data and palette values in a 32x8 RAM configuration to complete on-chip memory design.

Module: hdmi_text_controller_v1_0
Inputs: [7:0] keycode0_gpio, restart, axi_aclk, axi_aresetn, [C_AXI_ADDR_WIDTH-1 : 0] axi_awaddr, [2:0] axi_awprot, axi_awvalid, axi_wdata, axi_wstrb, axi_wvalid, [C_AXI_ADDR_WIDTH-1 : 0] axi_araddr, [2:0] axi_arprot, axi_arvalid, axi_rready
Outputs: [31:0] q, data_out, [31:0] palette[8], [31:0] score_record, game_over
Description: The hdmi_text_controller_v1_0 module is a top-level module that integrates the AXI4 protocol with HDMI control and various other game modules such as ball, stone, and coin. The module also interfaces with VGA signals and clock signals, allowing for synchronization of video and audio outputs. It handles user inputs through keycodes and facilitates the game's graphical output.
Purpose: The purpose of this module is to provide comprehensive control and coordination for the game's visual and audio components using HDMI and AXI4 protocols. It enables the game to interface with external devices such as displays and monitors for seamless game operation.

b) Components inside the Block Design

mb_block Module:
This module is likely the MicroBlaze block, responsible for implementing the MicroBlaze processor in the FPGA design.
Interface signals:
clk_100MHz: Clock signal at 100 MHz provided to the MicroBlaze block.
gpio_usb_int_tri_i: Tri-state input signal for USB interrupt from the USB controller.
gpio_usb_keycode_0_tri_o, gpio_usb_keycode_1_tri_o: Tri-state output signals for keyboard scan codes received from the USB controller.
gpio_usb_rst_tri_o: Tri-state output signal for USB reset control.

reset_rtl_0: Active low reset signal for the MicroBlaze block.
uart_rtl_0_rxd, uart_rtl_0_txd: UART receive and transmit signals.
usb_spi_miso, usb_spi_mosi, usb_spi_sclk, usb_spi_ss: SPI interface signals for communication with the USB controller.

clk_wiz_0 Module:
This module is likely a clock wizard or clock generator block, responsible for generating clock signals with specific frequencies required by the HDMI interface.
Interface signals:
clk_out1: Output clock at 25 MHz, used for VGA timing.
clk_out2: Output clock at 125 MHz, used for HDMI timing.
reset: Active high reset signal for the clock wizard.
locked: Output signal indicating whether the generated clocks are locked and stable.
clk_in1: Input clock signal provided as the reference clock input to the clock wizard.

hdmi_tx_0 Module:
This module is likely the HDMI transmitter block, responsible for converting VGA signals to HDMI-compatible signals for display on an HDMI monitor.
Interface signals:
Various clocking and reset signals for synchronization.
Color and synchronization signals (red, green, blue, hsync, vsync, vde) for HDMI output.
Differential outputs (TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N) for HDMI TMDS signaling.

AXI GPIO:
It is a fundamental interface commonly found in microcontrollers and FPGAs, allowing them to interact with the external world by controlling and monitoring digital signals. In this context, GPIO signals are often used to interface with various peripherals, such as buttons, switches, LEDs, and other devices.

AXI UART:
It is a commonly used serial communication interface found in microcontrollers, FPGAs, and other electronic devices. It enables asynchronous serial communication between devices, allowing them to exchange data over serial links.

AXI TIMER:

The AXI Timer module represents a programmable timer component designed to generate periodic or one-shot timing events in FPGA designs. AXI Timer modules are commonly used for tasks such as scheduling tasks, measuring time intervals, and triggering events at specific time intervals.

AXI Quad SPI:

The AXI Quad SPI module represents the Advanced eXtensible Interface (AXI) Quad Serial Peripheral Interface (SPI) controller, commonly used in FPGA designs to interface with SPI flash memories, sensors, and other SPI-compatible devices. The AXI Quad SPI controller provides a high-speed serial communication interface between the FPGA and external SPI devices, supporting quad data lines for increased data transfer rates.

AXI INTC:

The AXI INTC serves as the central hub for managing interrupts within the MicroBlaze system, facilitating communication between the processor and peripheral devices. It handles interrupt requests generated by various peripherals connected to the MicroBlaze processor, prioritizing and dispatching them to the processor for handling.

AXI Peripheral Interface:

This block serves as the bridge between the MicroBlaze processor and the AXI interconnect, facilitating communication between the processor and various peripheral devices in the system. The AXI Peripheral Interface block typically supports multiple AXI interfaces, allowing it to connect to multiple peripherals simultaneously.

b. I/O Workflow (Controlling the start/end of the game as well as (highest) score keeping)

The I/O workflow for controlling the start and end of the game, as well as score keeping, involves handling user inputs, managing game state, and updating score records. The game begins with user input via keycode0_gpio to start or restart the game when specific keys are pressed. The hdmi_text_controller_v1_0 module coordinates with other modules like ball, stone, and coin to control game mechanics and track the highest score. It updates the score and other game states such as game_over based on collisions and user actions. The workflow ensures that the game operates seamlessly by managing the display and sound outputs, as well as updating the score record (score_record) for the player’s highest score throughout the game.

After the generation of scores as well as the game_over signal, we used two GPIOs (General Purpose I/O) to communicate information between the hardware and the software. When scores and the game_over signal are delivered to the software part, the software code will convert logic values into decimals and finally display differences on the screen (these decimals and signals control the color of a specific region on the 640x480 drawing screen, and color information and changes are delivered using AXI bus and color palette registers). All GPIOs are single-ported and one GPIO is "all inputs" with width 32 (supporting 32 bits score recording), and the other "all inputs" GPIO is with width 1 (game_over signal contains only one bit).

The storage of highest score earned is implemented purely in software by creating a global variable "highest_score" to save the highest score in time and accurately. After updating "highest_score", it will also appear on the screen.

c. Interaction of Microblaze with USB and Ball Motion

The MicroBlaze communicates with the MAX3421E USB chip using the SPI (Serial Peripheral Interface) protocol. SPI allows for serial communication between devices and is commonly used for interfacing with peripherals like the MAX3421E. The MicroBlaze sends commands to the MAX3421E to control its operation, such as initializing USB communication, sending, and receiving data packets, and handling USB device enumeration. The MAX3421E, in turn, communicates with the USB keyboard, constantly polling for key press information. When a key press is detected, the MAX3421E sends the corresponding scan code to the MicroBlaze. The MicroBlaze receives the scan code from the MAX3421E and processes it to determine the direction key pressed by the user (A, D, K, or J).

The MicroBlaze controls the motion of the ball displayed on the HDMI monitor screen. It receives input from the USB keyboard via the MAX3421E to determine the direction in which the ball should move. Initially, when the program starts, the MicroBlaze ensures that a stationary ball is displayed in the center of the screen. It continuously monitors for keyboard input from the user. Upon detecting a direction key press (A, D, K, or J), the MicroBlaze updates the motion parameters of the ball accordingly.

The MicroBlaze also ensures that the ball follows the specified behavior, such as bouncing off the screen edges when it reaches them and changing direction immediately upon receiving a new key press without returning to the center of the screen.

d. VGA Operation and Corresponding Interaction

The operation of VGA involves generating horizontal and vertical synchronization signals along with pixel color information to produce a video signal that can be displayed on a VGA monitor or compatible display device. The VGA controller module generates timing signals for horizontal sync (hs), vertical sync (vs), and active/negative blanking (vde). These signals define the timing and structure of the video signal.

The pixel_clk input provides the pixel clock signal, typically running at a frequency of 25 MHz, which synchronizes the timing of pixel data transmission.

The reset input resets the VGA controller.

The drawX and drawY outputs provide the current pixel position within the active display area. Then the HDMI converter module takes the VGA signal generated by the VGA controller and converts it into an HDMI-compatible signal. It receives the pixel clock (pix_clk), horizontal sync (hsync), vertical sync (vsync), and active/negative blanking (vde) signals from the VGA controller. It also receives the red, green, and blue color signals (red, green, blue) representing the pixel colors. These signals are then processed and converted into TMDS (Transition Minimized Differential Signaling) signals (TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N) suitable for transmission over an HDMI cable.

e. Description of the Collision Detection in the Game

This game involves detecting collisions between the ball and the stone or coin, using the squared distance between the ball's position and the stone or coin's position. If the squared distance is less than or equal to the square of the combined radii (or sizes) of the ball and the stone or coin, a collision is detected.

When a collision between the ball and the stone occurs, the ball's motion is halted, and the game_over flag is set to indicate the end of the game. If a collision occurs between the ball and the coin, the player's score is increased, and the ball's movement may be altered according to the new score.

The ball's motion can be controlled by user input via keycodes, which dictate the ball's speed and direction of movement. If the user inputs specific keycodes during gameplay, the ball's movement will change accordingly.

The code also includes boundaries for the ball's movement, ensuring the ball stays within a defined playing area. If the ball's motion takes it beyond these boundaries, its movement is adjusted to keep it within the playable space.

Finally, the game has reset mechanics where, upon a game-over condition, the ball's position and motion are reset to their initial states, and the score is reset to zero. This allows for a fresh start each time the game is restarted.

f. Description of the Random Generator in the Game

The code for the ball, stone, and coin game includes a random generator used to influence game mechanics such as object movement and collision response. In the stone module, the random number (random) is generated when a specific keycode (8'h28) is detected, triggering a new random number using $random and beginning the game (game_start = 1'b1). This random number influences the movement of the stone, which changes its direction and speed based on the value of random and the position of the ball. For instance, when the ball reaches the maximum Y position (Ball_Y_Max), the stone's X and Y positions and movements are adjusted according to the value of random, leading to different patterns of stone behavior.

In the coin module, the random generator affects the coin's behavior similarly. The game keeps track of the score (keep_score) as the ball collects coins, and the random number (random) determines how the ball and coin interact. When a collision occurs, the score is incremented, and based on the value of keep_score and its remainder when divided by three, the coin's position and the ball's movement are adjusted. For example, if keep_score % 3 == 0, the ball and coin's X and Y positions, as well as the ball's motion, are changed according to the current value of random.

Overall, the random generator adds variability and unpredictability to the game's mechanics, affecting the movement and positioning of the stone and coin based on the ball's interactions with them. This introduces an element of challenge and excitement for the player as the game dynamically responds to collisions and the player's inputs.

g. Description of the Audio Generation in the Game

The code provided outlines the audio generation system in a game that uses specific frequencies of notes generated as square waves. The module sets up parameters for different frequencies of musical notes (e.g., 196, 220, 262, 294, 330, and 600 Hz), which are derived from the clock frequency (CLK_FREQ of 100 MHz). These frequencies represent different musical notes and are used to calculate counter values (COUNTER_MAX_n) for each note. The code uses multiple counters (counter1 through counter6) to generate square waves at specific frequencies.

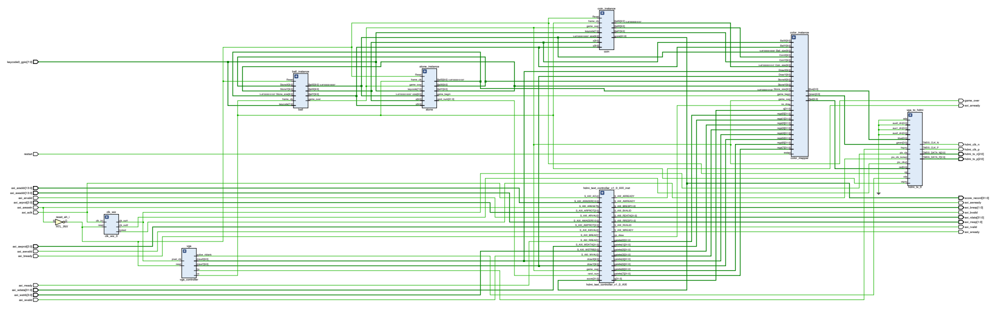
Each counter is responsible for toggling a corresponding wave (wave1 through wave6) whenever it reaches its maximum value, creating a square wave at a given frequency. When a counter reaches its maximum count (COUNTER_MAX_n), the corresponding wave is toggled (~wave_n), and the counter is reset. This behavior generates square waves of different frequencies for the respective notes.

The change counter increments over time and is used to determine which musical note (or frequency) should be played at a given moment. The code selects one of the waves based on the change counter and the current keycode input, producing a combined square wave output.

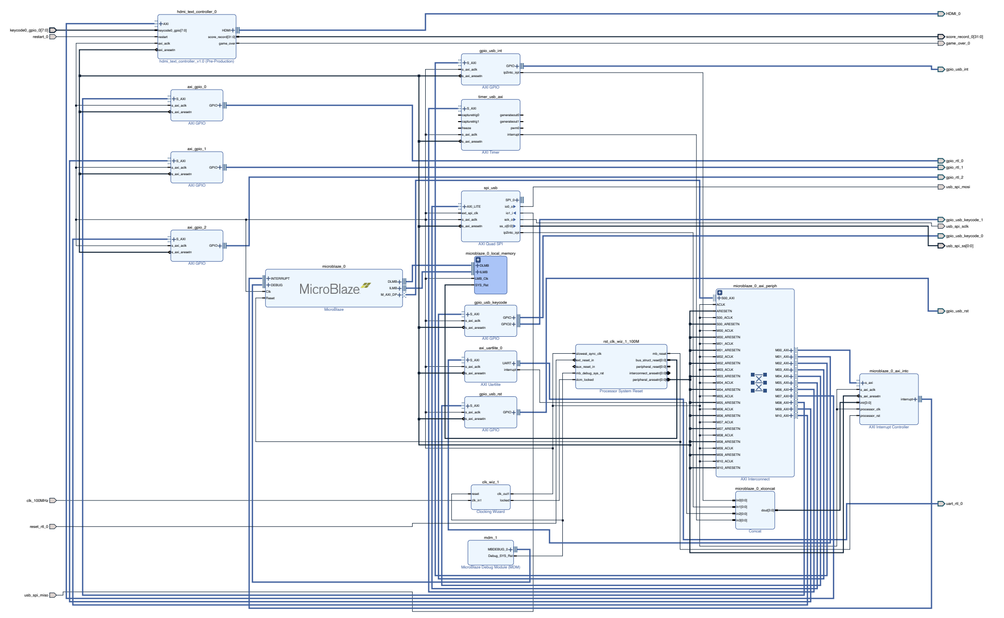
When a key is pressed (e.g., keycode 8'h28 or 8'h2c), the module changes the output wave, combining specific waves to create different sound effects depending on the game state or user input. The implementation provides a systematic way to control audio generation, enabling different sound effects by combining multiple square waves according to specified frequencies.

3. Top Level Block Diagram

a) RTL Schematic

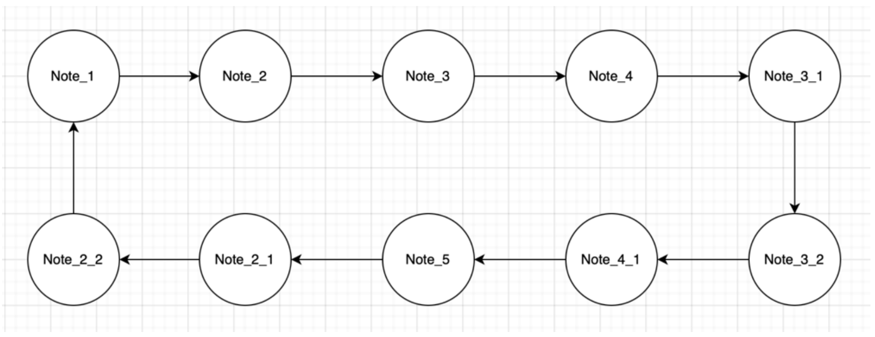


b) The Microblaze Block Diagram



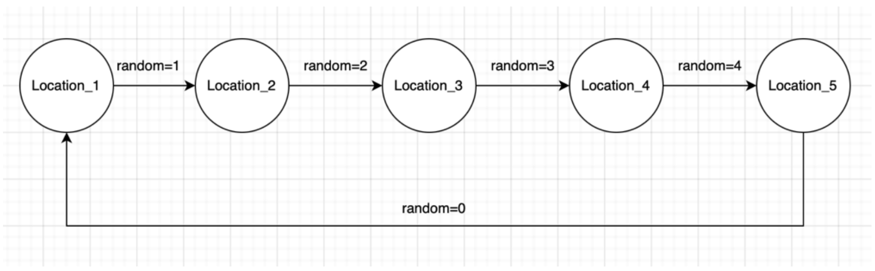
4. State Diagrams

a) Audio Generation



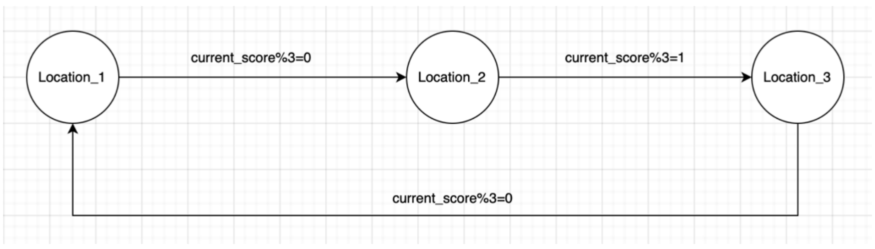
Note: Note_i_j represents the j-th appearance of Note_i of a specific frequency. And each state occupies exactly 0.5 s.

b) Random Generator for Obstacles



Note: Each location represents a starting x and y coordinates as well as velocities in the x and y directions of a new obstacle. “random” signal is created manually according to the user’s current score and position instead of using the random generator provided in SystemVerilog.

c) **Random Generator for Coins**



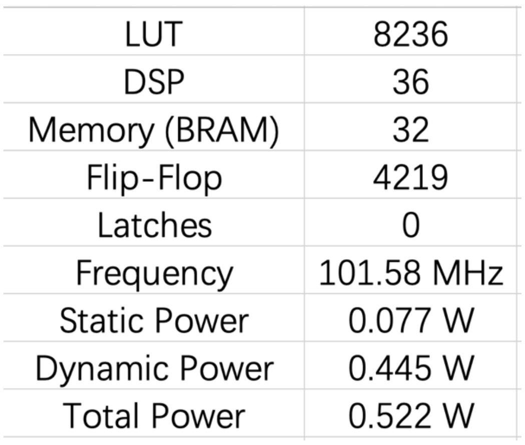
Note: Each location represents a starting x and y coordinates as well as velocities in the x and y directions of a new coin.

5. **Description of Software Implementations in the C Code**

The program initializes the USB subsystem using MAX3421E_init() and USB_init(), and then enters an infinite loop that performs USB tasks through MAX3421E_Task() and USB_Task(). It checks the state of the USB connection using GetUsbTaskState(), taking appropriate actions such as initializing or reinitializing the USB controller based on the state (e.g., USB_STATE_RUNNING or USB_STATE_ERROR). When connected USB devices are detected, the program uses GetDriverandReport() to identify the device classes and query device-specific information such as the rate and protocol. For keyboard devices, the program polls for keycodes using kbdPoll, while for mouse devices, it retrieves displacement and button states using mousePoll.

Additionally, the program monitors keyboard inputs to detect key presses such as the "Enter" key (keycode[0] == 40), and sets flags or performs specific actions based on the input. The program tracks the current score and highest score, updating the highest score if the current score surpasses it. This score information is then utilized to provide graphical output on the HDMI text controller, which is initialized with a predefined color palette. The program outputs various text and graphical elements on the HDMI display, such as scores, messages, and patterns, depending on game states like game over or game restart. This graphical output includes displaying messages like "GAME OVER" and "HIGH SCORE" at predefined positions on the screen, as well as rendering specific patterns based on the game's state. In summary, the code integrates USB handling, user input processing, and graphical display control, orchestrating the interactions between different components to achieve the desired functionality.

6. Documentation of Design Sources and Statistics



LUT	8236
DSP	36
Memory (BRAM)	32
Flip-Flop	4219
Latches	0
Frequency	101.58 MHz
Static Power	0.077 W
Dynamic Power	0.445 W
Total Power	0.522 W

7. Conclusion

a. Functionality of This Design

The design of the "Big Bet" game on FPGA demonstrates the successful implementation of a modified Temple Run-style game with various key features. It showcases the integration of an FPGA-based System-on-Chip (SoC) with a scenery generator, color mapper, and VGA controller, along with essential hardware components for audio storage and user input handling. The game includes an "infinite" style, where obstacles and coins are continuously updated, providing an engaging gameplay experience for the player. User interaction is achieved through keyboard inputs, enabling control over the character's motion at different speeds. The design effectively manages game mechanics such as character movement and obstacle avoidance, while collision detection ensures real-time feedback such as scores and game-over notifications. Additionally, the game tracks and records both current and highest scores, allowing for an element of competition and replayability.

b. Summary

In summary, the "Big Bet" game on FPGA is a well-executed project that brings together various elements of hardware and software design, including audio and visual outputs, user input handling, and game state management. The design leverages FPGA capabilities to create a smooth, interactive gaming experience with real-time processing and control. Through its integration of different modules and components, the game delivers a comprehensive gameplay experience, complete with obstacle and coin generation, collision detection, and scoring systems. The game also highlights the efficient use of hardware resources and demonstrates the potential of FPGA platforms for creating engaging and immersive gaming applications. Overall, "Big Bet" is a successful final project that showcases the skills and knowledge acquired in ECE 385 and provides a strong foundation for future FPGA-based designs..