ECE385

Fall 2023

Final project

Crossing Guard Joe

Haoyu Huang Eitan Tse
Lab section: ZY 10/Day & Time: 12.13
Your TA's Name: Yang Zhou

1. Overview

We rebuild a game that go viral in Steam, called “Crossing Guard Joe”. The user will act as Joe, who will guarantee the safety of children crossing a busy street. You can control the movement of Joe. Joe will wave his hand or raise a stop sign to tell the children to stop or go. They will line up behind a crossing sign The objective of the game is to get as many children as possible to cross the street safely, while cars drive down the street at random speeds. Once a certain amount of children are hit, the game ends.

We have design score tracking system, death signalling, and sound effects for the game.
The game is built on Urbana board combining C code and SystemVerilog modules. The user will use HDMI cable to connect to an external monitor, and an external keyboard to control the game.

Instructions: Run the .elf file on the board to initialize the game. The start screen will show onscreen immediately, displaying the words “CROSSING GUARD JOE”. Users press any key to start. Joe starts at the center of the screen and children will walk from the right. Cars start from a point roughly 2/3rds of the way up on the screen, and move down the screen at random speeds. They can hit children. Once 5 children are hit, the game will stop immediately and will transition to an end-game screen.

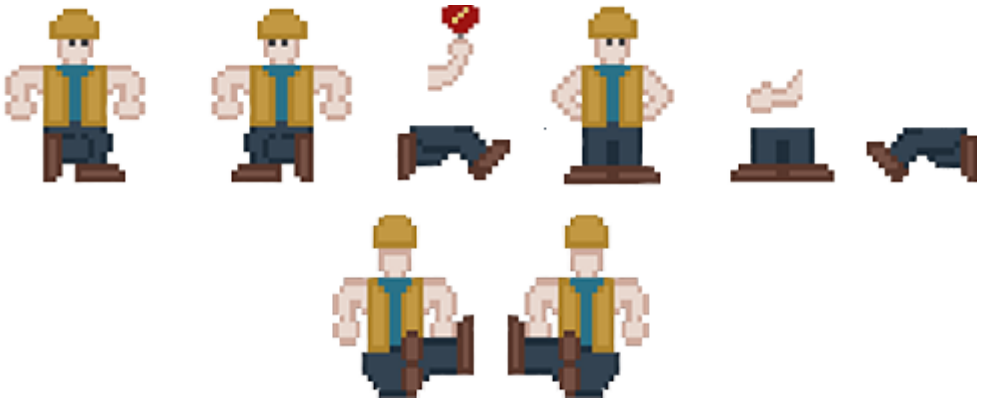
2. List of features

1.Background

We incorporate three backgrounds into the game, a start screen, the game background itself, and an end screen. These three backgrounds are stored in BRAM, with their palettes stored in a ROM. Pressing any key will exit the start background and enter the game background. Once five children have been hit by cars, the game switches to the end background.

2.Joe

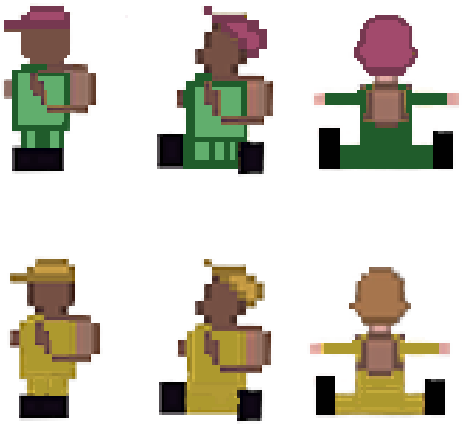
Description:
Joe is our main character and directly controlled by user. We can use A and D to control its movement. In addition, we can use J and K to control his hand. For his animations, we first divide joe into 3 different parts include body, hand, and foot. In this way we can control his hand while he is in moving frame. We use its sprite as its hit box. For walking, we have 2 frames for moving left or right. For hand, we have two different frames for left hand and right hand. For foot, we have 5 different states including standing still. We can save more memory space as we don’t need frame that is 180 degrees reflected. We can use logic to reflect the frame. Apart from moving, we design animation for hitting joe. We will introduce his FSM in detail in FSM part.
Joe can’t move outside the screen.



3.Children

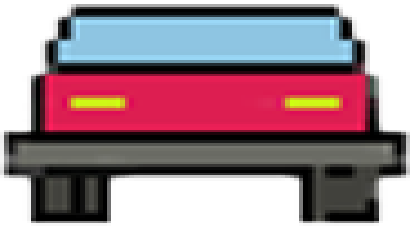
We have designed two set of children sprite. We need two different memories since we want to get access to memory at the same time. In total, we have four children. Each of them will move toward left without joe ask them to stop. They have two frames for moving. If they are hit by car. Once they get hit, they will move with car until they reach the corner of the screen. Car. Children will move outside the screen and appear at the other side of the screen.

Children also have special waiting system. At first, they will appear at the right side of the screen and move toward the stop sign at the right side of crossing sign. Four children will wait in line at the right side of road at first. In addition, if the previous child stop, the latter children can't walk through it but wait behind him.



3.Car:

The car has one sprite. It will have random speed. A new speed is acquired when the car goes off the screen. It can only move downwards and will reset to the horizon line after it goes a certain distance offscreen.



4.Score counter / death counter

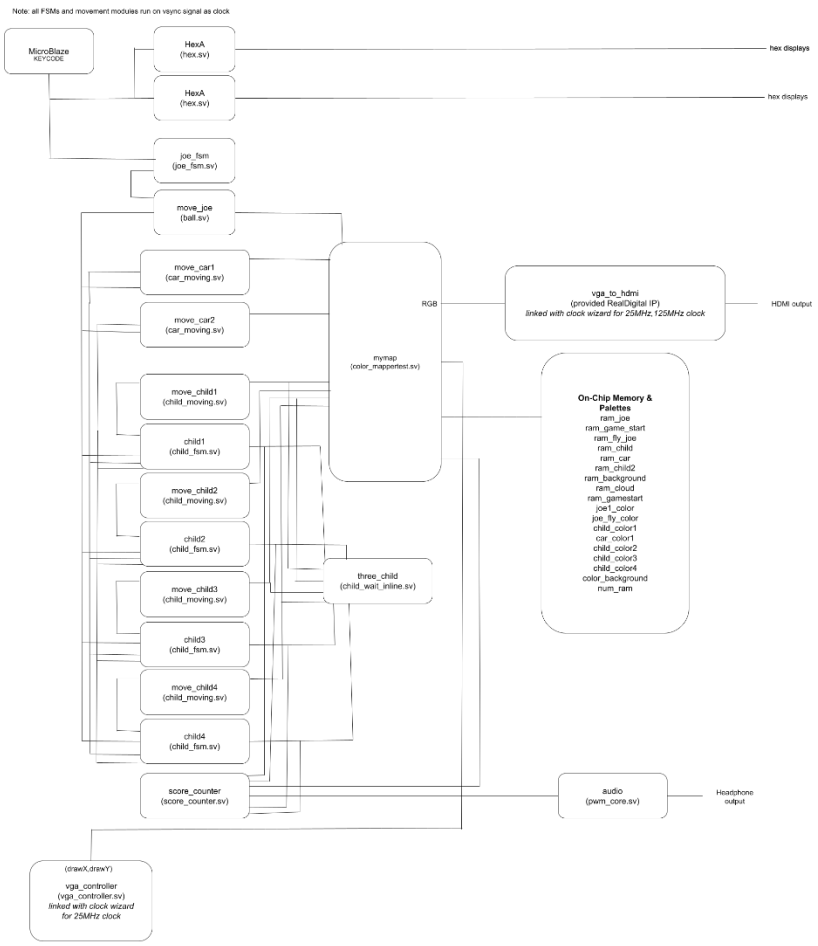
We have one score counter to count the number of children that have passed the crossing road. On the right corner of the screen, we count the number of children that is hitting. If one child is hit, one red circle will appear on the right corner. When 5 children are dead, it will end the game.

5.Audio

We use Urbana board to generate sound effect. We use PWM method to get analog sound wave. When children get hit, it will generate sound. When children walk in from the right side of the screen, it will generate sound. B13 and B14 are left speaker and right speaker. We set pulse width, pulse period and counter to control the sound, setting its frequency.

3. Block design

We used the same block design as we did for lab 6.2. This subsystem sets up a way for the MicroBlaze to communicate with the MAX3421E and run the appropriate drivers, so the written C code enables keycode reading from a keyboard. The keycode outputs are tied to a GPIO, which the game can interface with.



4. General description of how circuit work

The circuit reads keycodes from the GPIO of the MicroBlaze subsystem, and uses it to control the motion of the player character. Multiple FSMs are implemented to control animations and detect collisions, while movement modules store the current locations of the sprites onscreen and update them based on movement. Separate logic controls score counting and when to play tones. Block RAMs and register banks storing sprites and color palettes respectively are hooked up to a color_mapper, which reads the current coordinates of each sprite and draws the sprites onscreen. To handle screen controls, there is a provided VGA controller module and a provided VGA-to-HDMI IP that we use.

5. Module description

Hardware description:

Module: mb_usb_hdmi_top (mb_usb_hdmi_top.sv)
Input: Clk, rtl_reset_0, gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
Output: 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, hdmi_tmds_data_n, hdmi_tmds_data_p, hex_segA, hex_gridA, hex_segB, hex_gridB, leftsound, rightsound
Description: All hardware submodules for the game are implemented in this module. The module itself provides an interface between board I/O and the various modules provided and designed.
Purpose: This module provides the top-level interface between board I/O and the inbuilt game hardware.

Module: color_mappertest (color_mappertest.sv)
Input: x,y, ,centerx,centery, CLK, joe_run_left, joe_run_right, right_hand_warm, stand, left_hand_go, hit_joe, centerx_child1, centery_child1, run_child, hit, centerx_child2, centery_child2, run_child2, hit2, centerx_child3, centery_child3, run_child3, hit3, centerx_child4, centery_child4, run_child4, hit4, centerx_car, centery_car, centerx_car2, centery_car2, score1, score2, start_game, end_game, hit_counter
output: red, green, blue,

Description: We have input x,y which is location of current pixel. We also have input for centerx and centery for different object such as joe, children, score board and background. We also have signal like joe_run_left, right_hand_warm, hit_joe, run_child ……. We will discuss these signals in the FSM module for children and joe. These signals indicate which frame we want to display. We have different frame to achieve animation effect. The module consists of a large always_comb block to decide the order to draw sprite. We will draw joe first, then children and car, finally background. We also have module to decide whether to draw sprite in specific range. If display signal is high, we compute the address to get access to memory block. We store red channel value in memory block. Once we get back red channel value, we use color mapper to get the 24-bit rgb value and assign them to output ports.

Purpose: The module is used to assign (x,y) pixel with different RGB value. The module can help to display on object.

Module: car_address.sv, joe_address, child_address
These modules have same logic but different parameter since we have different size of sprite.
Input: show, x, y, centerx, centery,
Output: addra_car
Description: <Show> signal indicate whether we have the object display on screen. X and Y

is the location the current pixel. Centerx and Century is the location of the object. It can be joe, car or children. We use one always block inside the module. With respect to different sprite, we get the center point of the picture first.

Sprite_X= x - (centerx_object) + centerx_sprite
Sprite_Y = y - (century_object) + century_sprite

The specific address can be computed by address = sprite_X + sprite_Y * width (width of sprite).
Purpose: we use these series of module to compute the address in memory block. We will then assert address to block memory to get the color pixel.

Module:car_color.sv,joe_color.sv,child_color.sv,child2_color.sv,backgroundcolor_map.sv
These modules have the same logic. We assert red channel value and get back rgb value.
car_color map car's color. Joe_color map joe's color. Child_color and child2_color map two different child color. Backgroundcolor_map map background RGB value.

Input: red
Output: rgb

Description: The input red is the output from memory block. The ouput rgb will be assigned to red, green, blue channel in color_mapper. We use python to convert picture into RGB value. We use unique case block to link red channel value with RGB value.

Purpose: The module is used to decrease the size of memory block. We just need to store red channel value in OCM rather than 24-bit RGB value instead we store them in color module. However, it will increase the synthesis and implement time.

Module: is_child.sv, is_car.sv
The two modules have the same logic and purpose to determine whether pixel is in range of child or car.
Input: x,y, centerx,centery,
Output: show

Description: The module uses an always_comb block to compare centerx, centery with current pixel since every sprite have different size.
Purpose: We use the module to decide whether current pixel is in the sprite range.

Module: is_joe.sv, is_joe_fly.sv (check_location.sv)
The two modules have same logic. The previous one is used to determine the range of walking frame of joe. The other is used to determine the hit frame of joe.
Input: x,y, centerx,centery, hit_joe

Output: show_joe

Description: The module uses an always_comb block to compare centerx, centery with current pixel since every sprite have different size. The reason that we divide them to two modules is that two sprites have different size. When hit_joe is low and x, y is inside box of joe, we set the show signal high. If hit_joe is high and x, y is inside box of fly_joe, we set the show signal high.

Purpose: We use the module to decide whether current pixel is in the sprite range.

Module: joe_address.sv

Input: show_joe, x,y, centerx,centery, joe_run_left, joe_run_right, right_hand_warm, stand, left_hand_go
Output: addra

Description: The module has the input which indicate which frame should be display. As our sprite is well design, we will use expression:

$$addra = spiritex + spiritey*512 + offset$$

In our sprite, centery of different frame is the same. We need to adjust offsetx to get access to different frame. The module uses many always_comb to determine the animation. We have divided animation into body animation, hand animation and foot animation. We will control the animation.
Purpose: The module is used to compute the address that store in memory block.

Module: joe_fly_address.sv

Input: show_joe, x,y, centerx,centery, joe_run_left, joe_run_right, right_hand_warm, stand, left_hand_go
Output: addra

Description: The module is very similar to joe_address.sv. The only different is that we only have hit_left_joe sprite. To compute frame that is 180 degree rotate. We use expression: addra= width – x + y * width.

Purpose: The module is used to compute the address that store in memory block.

Module: car_address.sv

Input: x,y, centerx,centery, show_car
Output: addra_car

Description: We use x,y and centerx, centery to get pixel location in sprite. The module use expression addra_car = spiritex + spiritey * 92 to compute address

Purpose: The module is used to compute the address that store in memory block.

Module: child_address.sv

Input: x,y, centerx,centery, show_child, run_child, hit
Output: addra_child

Description: The logic is like joe_address. We have input to indicate which frame to display. We use different expression to compute address.

Purpose: The module is used to compute the address that store in memory block.

Module: check_circle.sv
Input: x,y, centerx,centery
Output: in

Description: The module uses radius expression to determine circle range. If it is in range, output <in> set high.

Purpose: The module is used determine whether pixel is inside the circle. We designed to draw circle on the right corner of the screen if children hit by car.

Module: check_back.sv, check_cloud, check_game_start
These modules have the same logic with different parametera

Input: left,upper, x,y,
Output: in, addra

Description: The <left> and <upper> signal indicate where to paste the sprite. We use similar logic as check_joe to determine whether pixel is in range of sprite. If the pixel is in range, we use sprite width to get the address inside block memory. These 3 modules will detect object include clouds, game start cover and game over page.

Purpose: The module is used to whether pixel is in range and get the address at the same

time.

Module: score_counter.sv
Input: vsync, reset_ah, child_X, child2_X, child3_X, child4_X
Output: score1, score2

Description: Vsync is frame clock. Reset_ah is reset signal. child_X, child2_X, child3_X, child4_X are centerx of children location. If child location is at the left of the screen, counter will add 1. We design a decimal adder to store two score bits.
Purpose: The module is used to count the number of children that have passed the road.

Module: score_board.sv
Input: left,upper, x,y, score1, score2, CLK
Output: show_black

Description: The module is adapted from lab7. In lab7 we use sprite to display text. In order to save memory space, we use 0 and 1 to store number 0-9. Input left and upper is the location of the number displaying on screen. Score1 and score2 are the current score. Since the sprite is mirrored shaped, we need to adjust the x coordinate. The module instantiate memory block to get data.

Purpose: The module is used to display number on screen.

Module: pwm_core_top.sv
Input: clk, reset_rtl_0, en
Output: leftsound, rightsound

Description: The module instantiates a pwm core to generate sound signal. When input en is high, counter will start to determine the time of sound.
Purpose: The module is the top_level of pwm core which will determine when to stop generate sound.

Module: pwm_core.sv
Input: rst_n, clk, en, period, pulse_width,
Output: pwm

Description: The module is adapted from pwm ip. The module uses pwm method to get analogy sound. We use two always_ff block to generate pwm signal. When counter is equal to period, counter get updated. If counter is smaller than pulse width, we set pwm high.

Purpose: The module implements pwm modulation to generate pwm signal.

Module: joe_fsm.sv
Input: clk, reset, joe_X, joe_Y, car_X, car_Y, car2_X, car2_Y, keycode
Output: raise_right, left_hand_go, stand, run_left, run_right, move, hit_joe

Description: The clock input is tied to the vsync signal of the VGA_controller module. This controls Joe's animations, allowing him to cycle through his three-frame run animation in either direction, raise/lower the stop sign, or put him into a flying animation if he's hit by a car based on the side of the car he's on. The animation frames will change every third of a second, using an internal counter.

Purpose: Controls Joe's animations.

Module: Child_fsm.sv
Input: clk, reset, child_X,child_Y, go, stop, joe_X, joe_Y, car_X,ccar_Y, car2_X, car2_Y, update_hit, child_wait_inline, start_game, run_child
output: hit_child, move.

Description: The clock input is tied to the vsync signal of the VGA_controller module. This controls children's animations, allowing them to cycle through two-frame run animation or hit animation if he's hit by a car. The animation frames will change every third of a second, using an internal counter. Besides, children will wait in line if previous stop and wait on right side of road. If children run or hit out of screen, we will update its state.

Purpose: Controls children's animations.

Module: lfsr.sv
Input: rst, start, seed, clk, spd

Description: Internally, a 64-bit linear feedback shift register is implemented. This length was chosen to introduce a significant amount of randomness, as a previous implementation with 3 bits was insufficiently random The seed is loaded as a default value when start_game XOR inverted reset. Otherwise, it internally shifts itself by one bit over. It is implemented with an XNOR signal at the taps; thus it will lock-up when its value is all 1s. This means its minimum value is 0. That is not desired behavior, so I add 1 to ensure the minimum speed is 1. I also modulo the output value by 5 to limit its speed. Thus, the output value of the module is: (LFSR value) % 5 + 1.

Purpose: Used for pseudo-random speed generation for the cars onscreen.

IP block:
Block Design:

MicroBlaze:
MicroBlaze is a soft-IP based 32-bit CPU which can be programmed using a high-level language which has 32-bit modified Harvard RISC architecture. In this lab, Microblaze is the system controller and handles tasks which do not need to be high performance. It can be configured with different instruction and data cache sizes, as well as various peripherals to suit the specific application requirements.

Clocking Wizard:
The Clocking Wizard is a clock generating tool that helps in generating clocking structures for the FPGA design. It is used to create clock signals with specific frequencies and phase relationships, which are essential for synchronizing various components in the design.

Processor System Reset:
The Processor System Reset IP block is used to control the reset signals for the MicroBlaze processor and other components in the design. It provides a way to control and synchronize the reset process to ensure the system starts up in a defined state. It is an active low reset.

MicroBlaze Local Memory:
Local memory refers to the on-chip memory resources available to the MicroBlaze processor. The MicroBlaze local memory is an integral part of the processor and is used for storing program code, data, and stack memory.

AXI Interconnect:
The AXI (Advanced extensible Interface) Interconnect is used to connect various IP blocks within an FPGA design. It provides a high-speed and flexible communication infrastructure. The AXI Interconnect can be configured to support different data widths, address ranges, and arbitration schemes to facilitate communication between different components like the MicroBlaze, memory, and peripherals.
The AXI Interconnect follows a bus-based architecture. Various masters (IP cores that initiate transactions) and slaves (IP cores that respond to transactions) are connected through the AXI Interconnect. This architecture allows for multiple masters and slaves to communicate simultaneously.

AXI Interrupt Controller:
The AXI Interrupt Controller controls interrupt signals within the design. It's responsible for handling and distributing interrupt requests from various peripherals to the MicroBlaze processor. It can allow the processor to respond to external inputs.

AXI GPIO:
AXI General Purpose Input/Output (GPIO) is an IP block used for interfacing with external digital I/O devices. It provides a way to read inputs and control outputs from the FPGA, allowing the system to interact with the external world. AXI GPIO can be configured to work with different numbers of pins and can be used for various purposes, such as controlling LEDs, buttons, or other digital devices.

AXI Uartlite:
The UART (Universal Asynchronous Receiver/Transmitter) communication module is designed to work with AXI (Advanced eXtensible Interface) bus architecture. UART communication is a standard method for serial data transfer, often used to establish communication between an FPGA or SoC system and external devices, such as microcontrollers, sensors, or other computers. In this lab we use UART to print some messages in serial terminal.

Concat:
The "Concat" IP core is a simple but important component used in FPGA designs. Its primary purpose is to concatenate or combine multiple input signals into a single output signal. This is particularly useful when merging multiple data or control signals into a single bus or register.

AXI Quad SPI:
The AXI Quad SPI provides an interface between an AXI-4 bus to SPI devices. The SPI device in this case is the MAX3421E, which interfaces with USBs directly and is the slave. This allows the Microblaze to enumerate and read keycodes and use it to control other hardware.

AXI Timer:
The AXI Timer is an IP core designed to provide accurate timing and counting functions within an FPGA or SoC system. AXI Timers can be configured for various counting and timing functions, making them useful for generating time intervals, timeouts, or clocking signals. In this lab, since USB has many timeouts that require timekeeping in milliseconds, the timer allows the MicroBlaze core to keep track of when 10 milliseconds have passed.

vga_to_hdmi
The VGA-HDMI IP shares similarities by rendering everything with RGB signals, horizontal and vertical sync, much like VGA does. The VGA-HDMI IP uses complex logic to generate its outputs, creating control signals for the red and green channel and using the blue channel to encode the hsync and vsync signals, generating specific hardware to encode that. There are also multiple serializers to convert the data to appropriate TMDS_DATA and TMDS_CLK outputs, which have their own output buffers.

Software design:
We reused the same C code as lab 6.2 without any changes, as we just need the keyboard to be read into the GPIO.

6. Design procedure
Week1: Write proposal and begin to gather some sprite for the game. We learn to convert photo to sprite and load as .coe file in OCM.

Week2: Design incomplete FSM for coe. Successfully load animation for joe. We can control

its foot and hand separately without changing his center location.

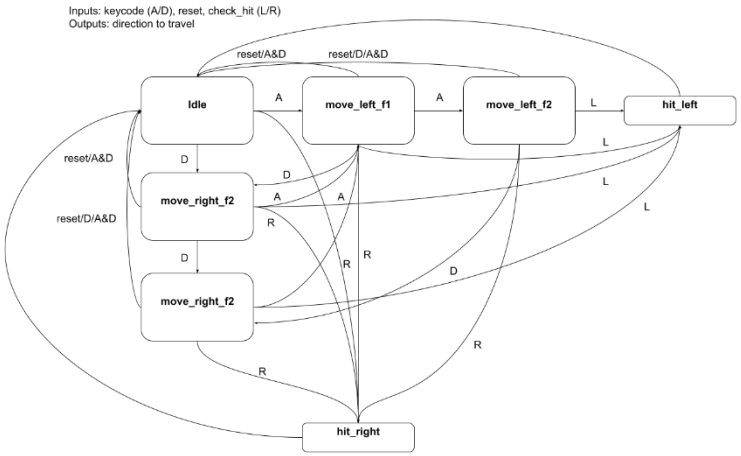
Week3: Design joe_moving module to update joe's location and connected with joe's FSM. We can move joe and control his hands. We add children sprite and display them on screen. We design child_moving module and make children keep walking. Besides, we design child' FSM. We add car sprite and make it move.

Week4: We update more detail for joe and children. Joe can only control his closest children. We connect child_moving module with its FSM to get animation for hitting children.
Children will stop if the previous child stop and will wait in line. We also add more detail in child_moving module to have infinity children loop. Besides, we must update state of children if they get hit out of screen.

Week 5: We add more detail include background, start cover and game over sign. We load a background image the screen with some clouds and road. We add a counter to count the number of passed children and red circle to mark death number. Then we add audio model to generate sound.

7. Fsm

Joe_FSM



States:

idle: represents the idle animation frame

move_left_f1, move_left_f2, move_right_f1, move_right_f2: the two animation frames for Joe running left and right, respectively.

hit_right, hit_left: Joe has been hit by a car and will change animations to face a particular direction, based on which side of the car he is on.

Inputs:

keycodes: read from the GPIO of the keyboard-reading subsystem

reset: tied to the physical button reset on the board itself

check_hit: detects if Joe has been hit by a car and signals if he is on the left side (and thus should fly left) or on the right side (should fly right).

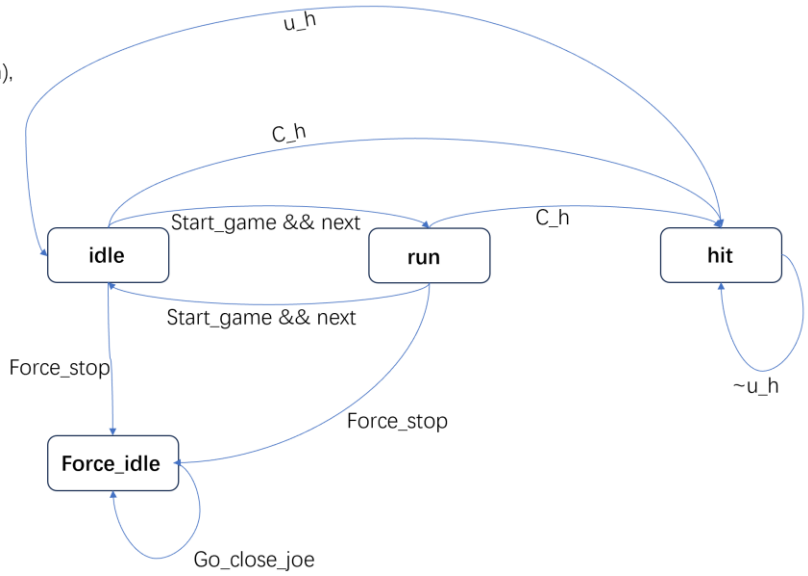
Output:

direction: tied to the 8-bit “move” signal in the SystemVerilog module itself. Outputs an appropriate value to move Joe in the correct directions.

Children_fsm

Input: check_hit (c_h), next, force_stop(f_s), update_hit(u_h), go_close_joe, start_game

Ouput: direction to travel



States:

Idle and run are two frame for running. While force_idle is children keep still. Hit state is children hit by car.

Input:

Force_stop: the signal is kind of complex. Children will stop in 3 conditions. 1. Joe force them to stop. 2. The previous children stop. 3. They reach the right side of road.

Check_hit: the module has input of car location and children location.

Next: the signal is control by counter to update frame for running

Update_hit: when children hit out of screen, we need to update state.

Go_close_joe: Since joe can only control his closest children, the signal will change the state of children.

Start_game: it indicates game have started. (if user press any keycode in start page, the game gets started)

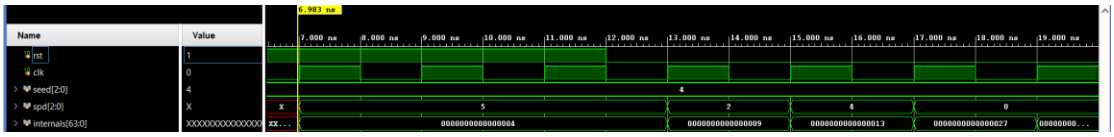
8. Data Analysis

Resource Usage

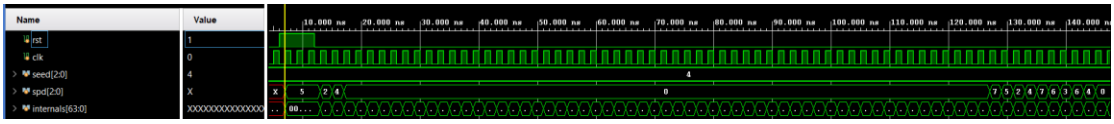
LUT	5782
DSP	30
Memory (BRAM)	74
Flip-Flop	3669
Latches	0
Frequency	76.342MHz
Static Power	0.079W
Dynamic Power	0.427W
Total Power	0.505W

9. Simulation

LFSR Simulation Waveform

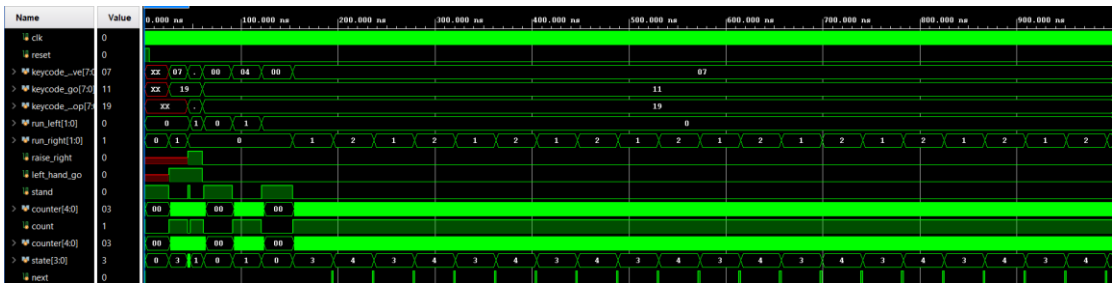


Here is the LFSR simulation waveform to verify functionality. This just shows the beginning of the LFSR starting up, but it exhibits correct behavior.



A zoomed-out screencap showing the changing speed values calculated by moduloing the LFSR output by 5 and adding 1. The LFSR can be seen changing values every clock cycle on the bottom.

Joe_FSM Simulation Waveform



Here is a barebones functionality simulation of Joe's FSM. We can see the counter functioning, the states transitioning, and the various output control signals changing according to the keycodes.

10. Bugs

Introducing randomized speeds to the cars caused the children to not be hit correctly by them. This was resolved by increasing the y-value range in which the children would consider themselves as being hit by the car- expanding the car's hitboxes vertically, to be accurate.

When implement the score counter, I met a problem score will update at fast speed. I found a set the clock wrong. As we have made our animation updated every third of a second, using an internal counter. We should change clock accordingly.

When implement the hit logic, we met a problem that children disappear when they hit by car. We found that we ignore the logic that their state should be changed to walk after hit by car. We also changed the logic in children_moving module to update its location.

11. Conclusion

This game was a very good extension and application of the material we covered in this course. Lab 6.2 formed the backbone of the game, as it allowed player interfacing via keyboard to control the game. However, learning how to draw sprites from Lab 7 was the most useful. We learned how to draw sprites and set their priorities, which would determine what sprite is drawn if two of them overlap at the same time in making this game.