ECE 385

Fall 2022

Final Project

Super Mario

Zhuoyuan Li, Shihua Cheng

Tianhao Yu

Introduction

For the final project, we built a modified version of the classic game “Super Mario” on the DE-10 board. We designed a list of features for this game, realized these features using SystemVerilog and C code, and added a hidden element to the original game. The goal of the game is to control the main character Mario using “W, A, S, D” keys and to reach the flag at the end of the map. A score counter is present on top of the screen. Mario can collect coins or hit the “?” box in the map for score. The game can be restarted by pressing the key on the DE-10 board.

Written Description

Our game is designed to run at 640 * 480 resolution, 60 frames per second. We decided to use a frame buffer since we wanted to add objects to the background, and the background should be visible under transparent pixels of the Mario sprites. In order to fit the frame buffer into the on-chip memory, a color palette method is used. We used 16 colors for the game, which can be represented by 4 bits. In this way, we would need a total of 640*480*4 = 1,228,800 on-chip memory bits, which fits perfectly into the 1,677,312 bits of the on-chip memory. We used the provided png_to_palette_resizer.py script to convert sprites into .txt files in hexadecimal format, which can be loaded into the on-chip memory at compilation.

Following is a list of features of our project:

Multiple keyboard inputs: The USB keyboard input reading logic is adapted from Lab 6.2. We made some modifications to it so that two inputs can be read simultaneously, otherwise the character cannot jump while moving and vice versa. The details will be introduced later.

Score counter: A score counter is placed at the top of the screen to keep track of the score that the player has. Score can be obtained by either hitting the “?” block or collecting coins. The provided font_rom.sv is used for the number fonts.

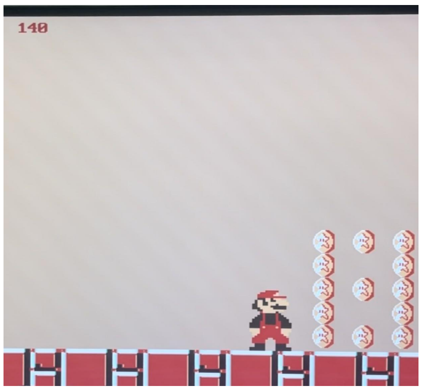


Figure 1. Score Counter (top-left corner)

Complicated background: We added clouds and trees to the game to make the game more visually appealing. The ground blocks are also drawn from sprites for more texture. When

Mario stacks with a background object (cloud or tree), the transparent pixels or the Mario sprite will appear as the color of the background object. This is achieved by using a frame buffer and makes the game look more realistic. Our game map is designed to be 1920 pixels long, so we used a tiled approach of drawing the background object sprites. The modulo operator is used when calculating the drawing position of each object, so that the objects are drawn evenly across the game map. For example, a tree is drawn every 256 pixels.

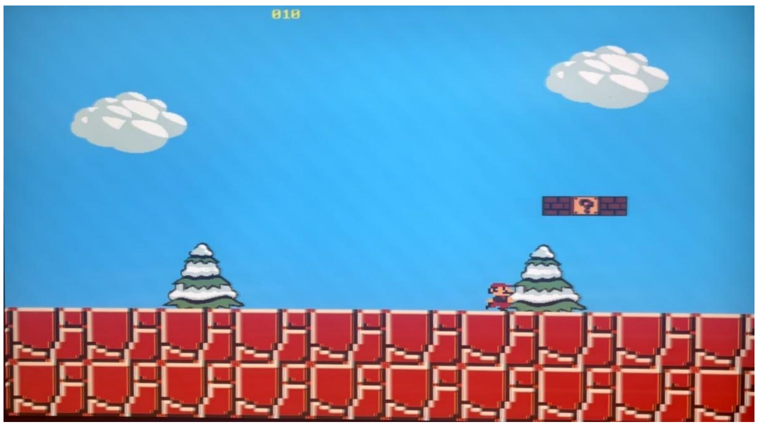


Figure 2. Background Sprites

Animated character movement: When the player moves Mario using the keys “W, A, D”, there will be an animated effect on Mario. First, the direction of Mario will be determined by the last direction key pressed. If it is “D”, then the direction will be to the right; if it is “A”, it will be to the left. For each of the small and big Mario states, five sprites are used. Following are the sprites for small Mario:

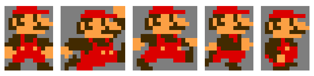


Figure 3. Small Mario Sprites

The sprites facing to the left are not used because they will take up more memory. If the direction is to the left, the horizontal indexing will be reversed (from right to left) to achieve the desired effect. The first sprite is “idle”, second is “jump”, and the others are “walk”. If the character is in the air, the “jump” sprite will be used. If no key is pressed, the “idle” sprite will be used. For “walk”, a 4-bit counter is used. While a direction key (A or D) is pressed, the walk counter keeps increasing. If no direction key is pressed, the counter will be set to zero. If the character is not in the air, the walking animation is achieved by using different sprites at different counter values.

Counter value	0	1-5	6-10	11-15
Sprite	idle	walk1	walk2	walk3

The values are chosen so that the walking animation is 12 frames per second, which is neither too fast nor too slow. Also, the 4-bit walk counter automatically resets to zero when it is larger than 15, so no additional logic is needed.

Dynamic sprites: Several dynamic sprites are implemented in the game. When Mario hits the "?" block, there will be a flashing animation for 1 second, and then Mario will become bigger. Also, the "?" block will turn into a normal block as soon as Mario hits it. This is achieved using a state machine, which is introduced later. Following is a comparison between two frames before and after Mario hits the "?" block.

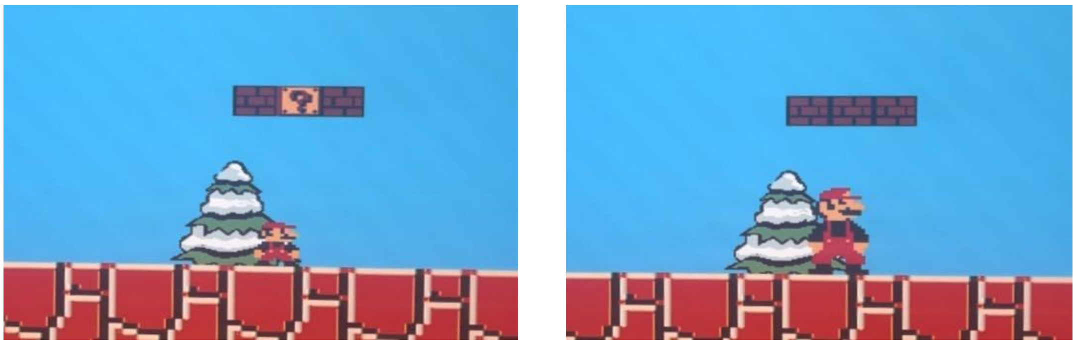


Figure 4. Dynamic Sprites

Also, the coins will disappear once Mario reaches them.

Gravity: We implemented gravity for the game so that Mario will slow down in the air after he jumps, and finally fall down and hit the ground. We used a parameter, *g*, as the acceleration constant to simulate gravity. When Mario jumps, its initial vertical velocity is set to a value smaller than zero. Mario's vertical position is added by <vertical velocity> each clock cycle, and its vertical velocity is added by *g* each clock cycle. As long as Mario falls back to the ground, the vertical velocity is fixed to zero unless the jump key is pressed.

Scrolling game world: The game world will scroll as Mario moves toward each end of the map. We used two variables, *mario_x* and *screen_x*, to keep track of the position. *mario_x* is the character position on the screen, and *screen_x* is the position of the screen with respect to the game world. Taking moving forward as an example, when Mario reaches around ¼ of the screen, *mario_x* will no longer be updated if Mario keeps moving forward. Instead, *screen_x* will be incremented. The background sprites are drawn with respect to *screen_x*, so the background sprites will look as if they are moving backward if *screen_x* is increasing. The game world will scroll using this logic. The same logic applies to moving backward.

Multiple levels: There are two levels for this game. At the end of the first level, there is a tube on which Mario can jump. When Mario is standing on the tube and the "S" key is pressed, the game will move to the second level. The background color and the objects are different in the second level.

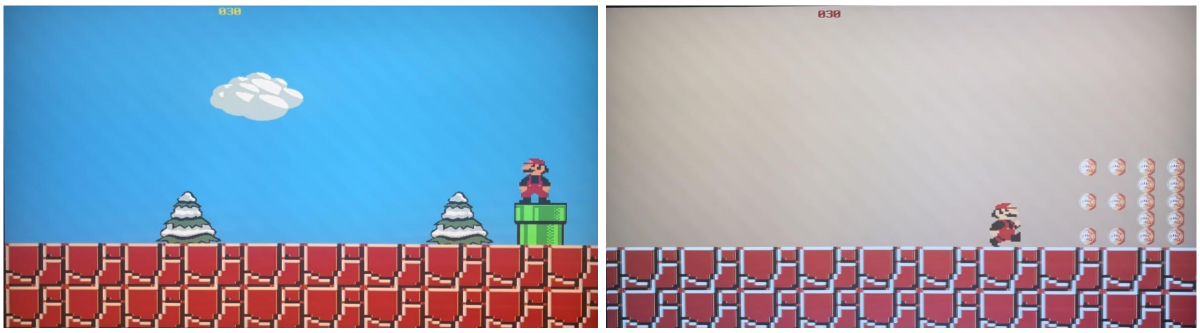


Figure 5. Multiple levels

End screen: In the second level, when Mario reaches the flag, the game will be over. The screen will be immediately frozen and the game will be unresponsive to keyboard inputs. A game over message will be displayed in the center of the screen.

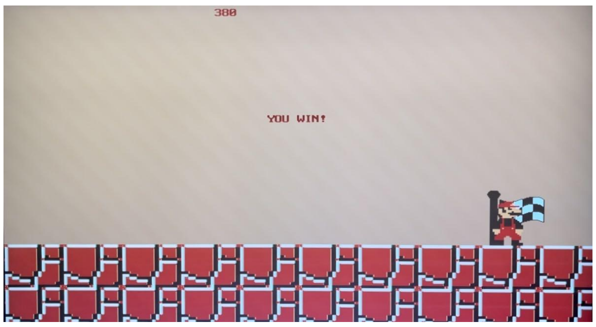


Figure 6. End Screen

An Easter egg: There are a total of 36 coins in the game, one in the first level and 35 in the second level. The coins are arranged in the shape of "385" in the second level. If all 36 coins are collected by Mario, Professor Cheng will appear and say congrats!

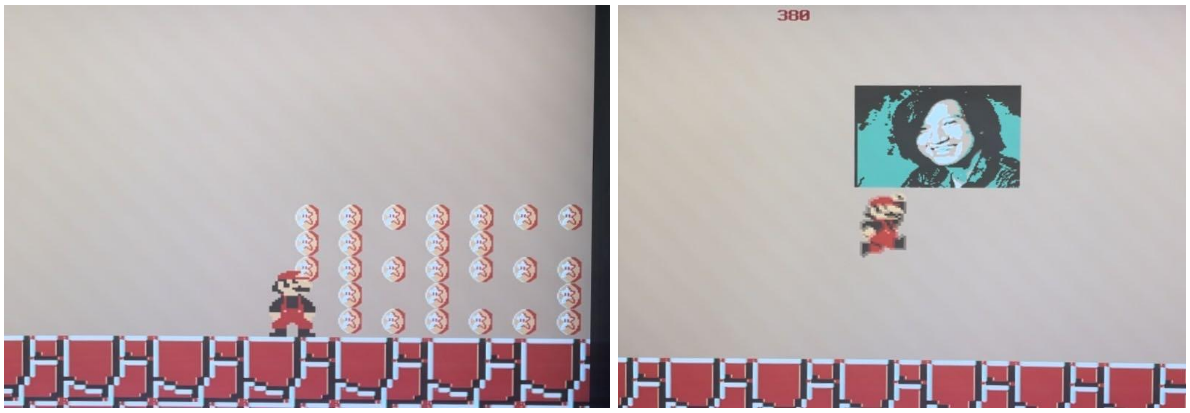


Figure 7. Easter Egg!

Block Diagrams

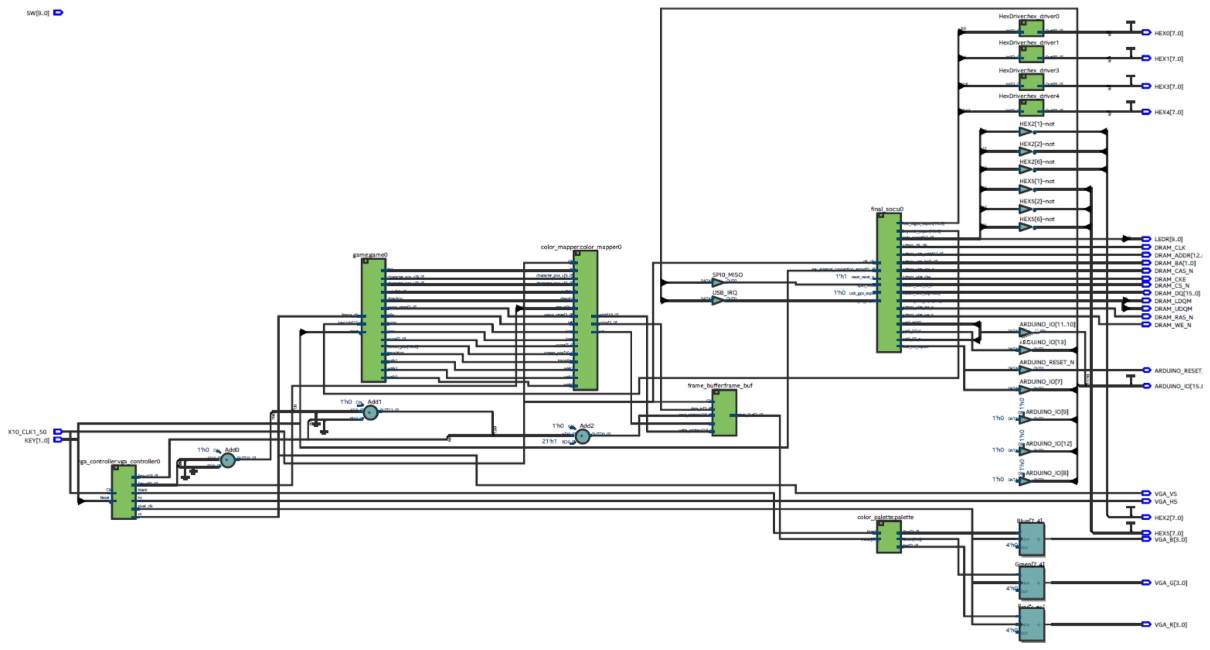


Figure 8. Top-level Block Diagram

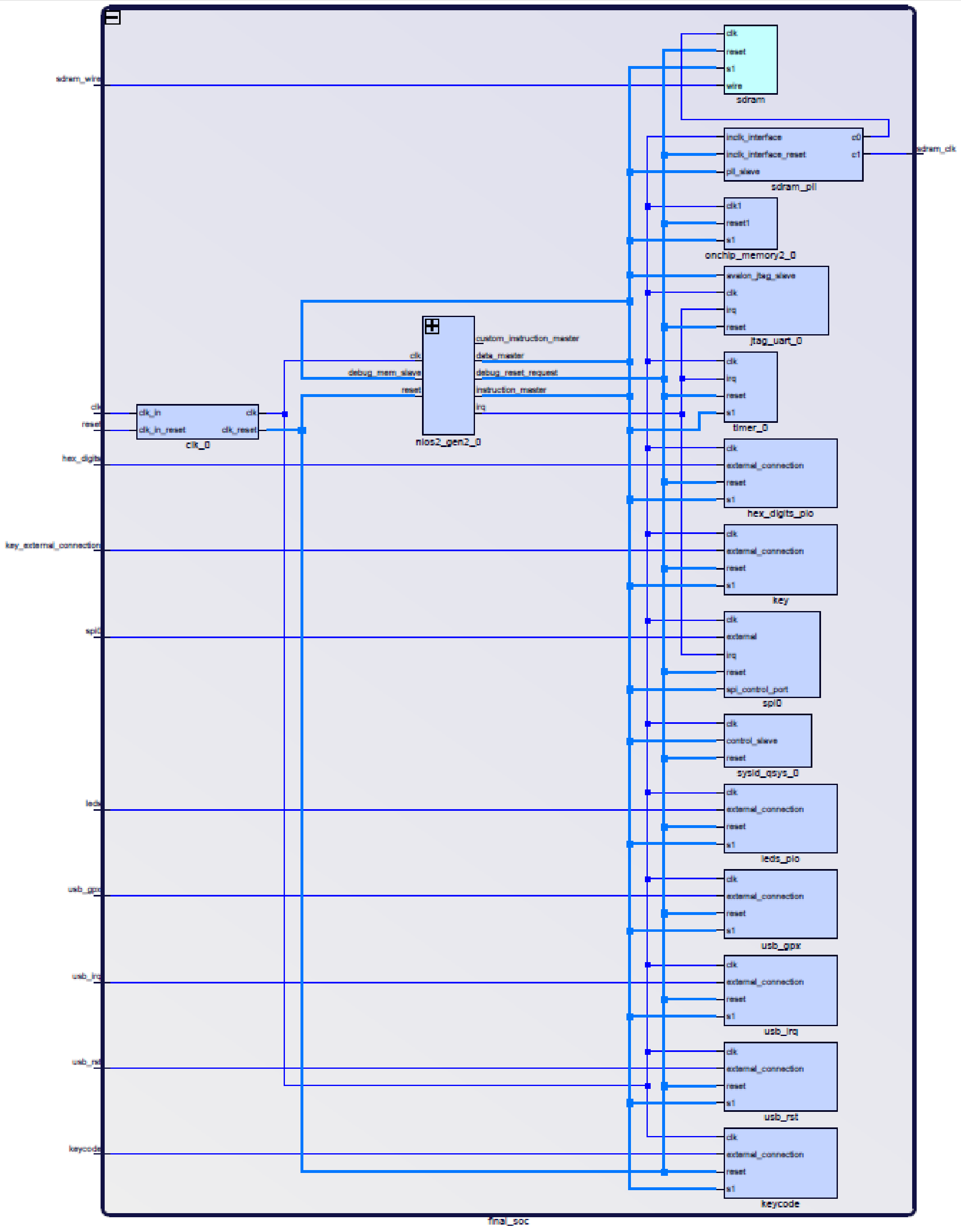


Figure 9. Platform Designer Block Diagram

State Machine Description & Diagrams

We used two simple state machines for this project. If the reset button is clicked, both state machines will return to the first state.

The first state machine is for game states. There are a total of 4 game states:

state1: This state is the initial state of the game. It is the first level of the game. When in this state, the color mapper module will draw the background of level 1. The objects unique to this level, such as the blocks and the tube, will interact with Mario in this state. For example, if the "?" block is at (400, 300) and Mario hits it, Mario will become bigger in this state, but Mario will not become bigger in the next level if he hits (400, 300) again. The next state *trans1* will be reached if the character is standing on the tube and the "S" key is pressed.

trans1: This is the transition state between level 1 and level 2. It will reach the next state after one clock cycle. In this state, the game variables such as the position of Mario and the position of the screen will be reset and initialized for level 2.

state2: This state is the second state of the game. This is level 2 (also the last level), and the color mapper module will draw the background for level 2 which is different from level 1. There will be many coins in this level and there is also a hidden element. A flag is present at the end of the level and the player will win if Mario reaches the flag.

over: This state is reached when the flag is reached by Mario. At this state, a "You win!" message will be printed to the center of the screen, and the screen will freeze (not responsive to keyboard inputs).

Following is the state diagram for the game state machine:

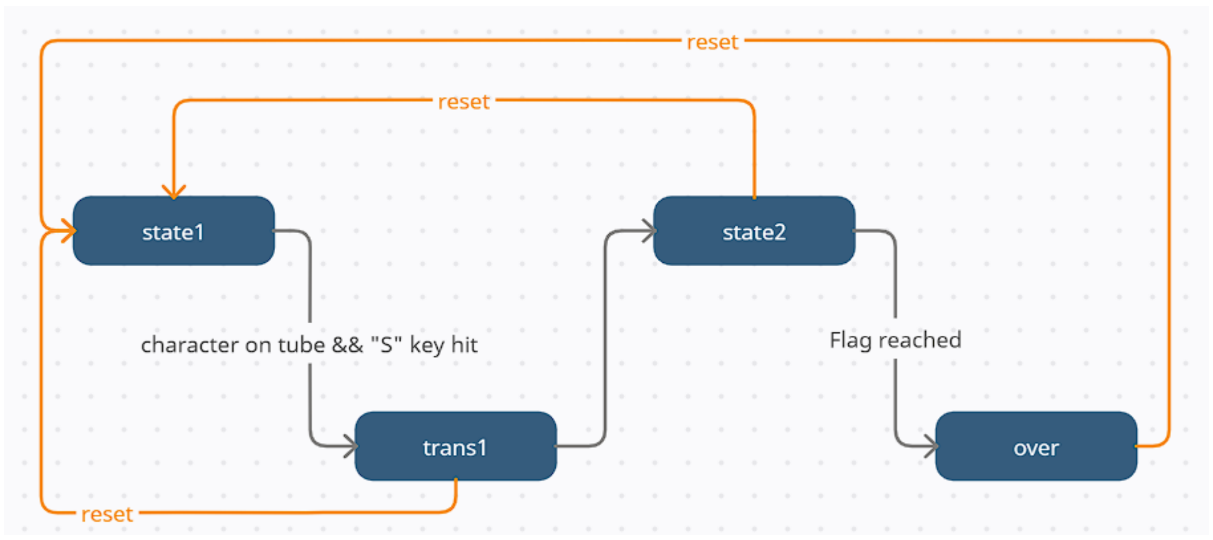


Figure 10. Game State Diagram

The second state machine is for dynamic Mario sprite. There are 3 states:

Small: This is the initial state of the machine and is the default state for the game. In this state, 32*32 small Mario sprites will be used. If the “?” box is hit by Mario, the next state *Transition* will be reached.

Transition: This is the transition state when Mario hits the “?” box. This state will last for exactly 64 clock cycles. This is because an animation is present when Mario becomes bigger. It will flash between small and big for 8 times in about 1 second. A counter is used in this state to calculate whether a big or small sprite should be drawn. For example, the big sprite is used between counter value 0-7 and the small sprite is used between counter value 8-15. Also, the “?” box will be drawn as a normal block once this state is reached. The next state *Big* will be reached when the counter value reaches 64.

Big: In this state, 32*64 big Mario sprites will be used. Mario will remain big for the rest of the game unless the reset button is pressed.

Following is the state diagram for the Mario state machine:

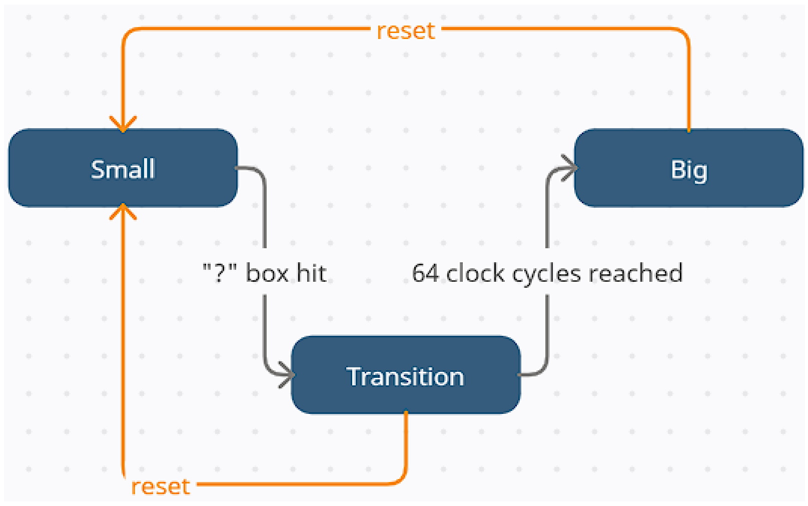


Figure 11. Mario Sprite State Diagram

Simulation Waveforms

The following simulations test the basic moving functionality (W, A, D), the scrolling world feature, and the dynamic sprite feature when Mario hits the “?” block.

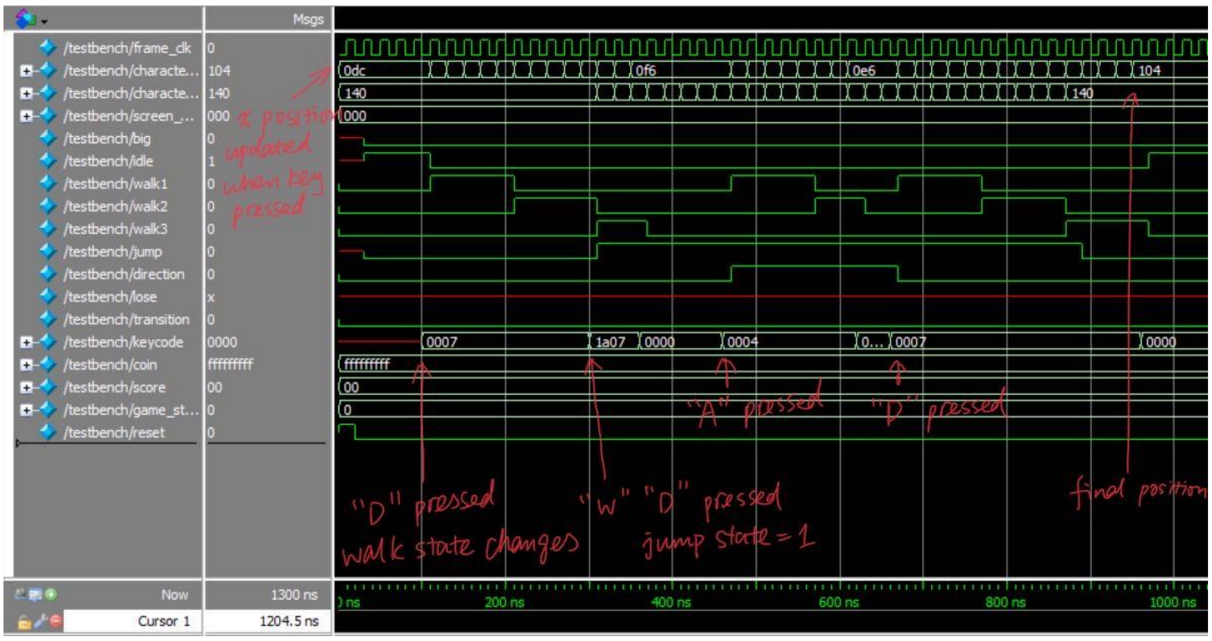


Figure 12. Basic Movement Simulation

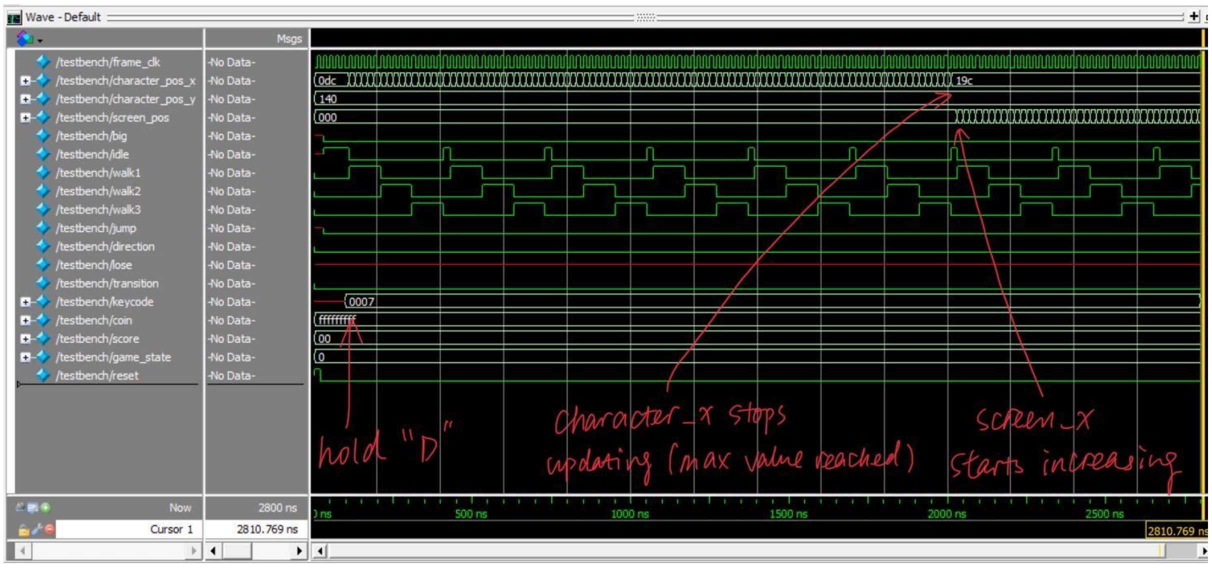


Figure 13. Scrolling World Simulation

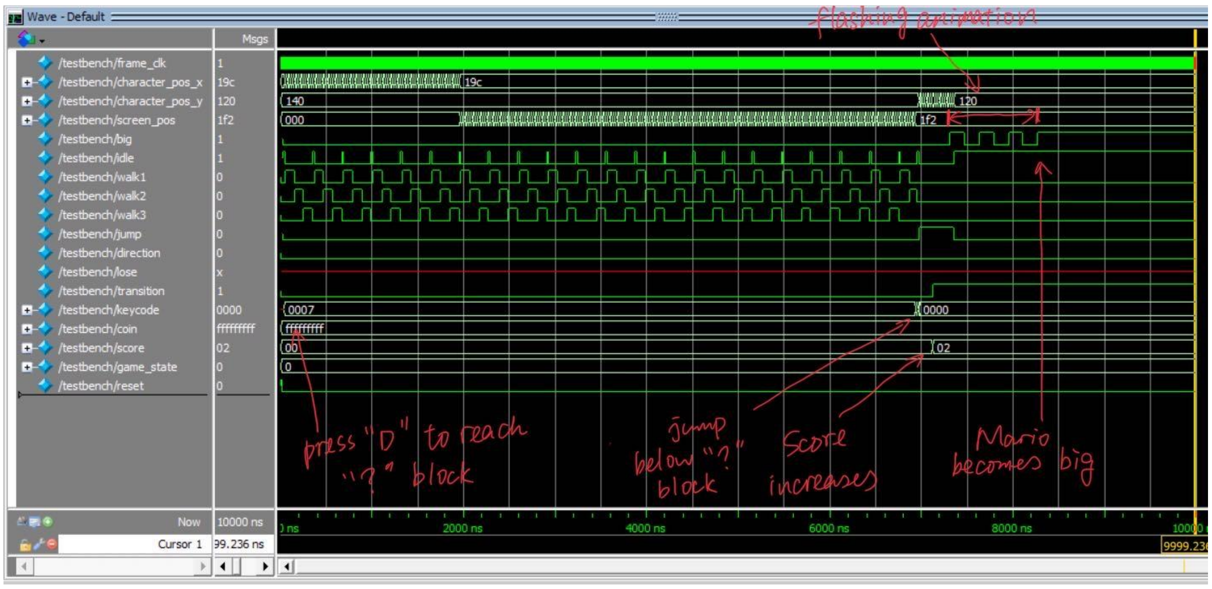


Figure 14. Dynamic Sprite Simulation

SV Module Description

color_palette.sv

input: blank, [3:0]index,

output: [7:0] Red, Green, Blue

description: Using the input index, this module saves the color with 16 indexes. If input is blank, then RGB output will be 8'h00; otherwise, the RGB will take the 23:16, 15:8, 7:0 bits of the 24 bits saved color.

purpose: This serves as a palette table, and the color could be chosen using 4 bits, so it saves memory to store the color.

HexDriver.sv

input: [3:0] In0

output: [6:0] Out0

description: The dexdriver is used to convert a 4-bit unsigned integer (0-F) to hexadecimal form.

purpose: Convert a 4-bit unsigned integer (0-F) to hexadecimal form.

frame_buffer.sv

input: [3:0] data_In, [18:0] write_address, [18:0] read_address, we, Clk,

output: [3:0] data_Out

description: The frame buffer serves as a RAM which store 640 * 480 words, or the entire screen which will be printed by the VGA. If we, the data_In, which is the index to the color palette, will be written to the given write_address within the frame buffer. data_out is the data within that read_address.

purpose: The frame_buffer produces a smooth motion and enables screen scrolling.

vga_controller.sv

input: Clk, Reset,

output: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY,

description: The same VGA_controller as the one in lab7.2. This is the module that controls the VGA signal output on the FPGA board, so that contents can be displayed on an external monitor. It has asynchronous reset, and it uses the 50MHz Clk input to generate the 25MHz pixel_clk signal. This 25MHz pixel clock provides 25MHz / 800 (horizontal) / 524 (vertical) = 59.6Hz refresh rate on the external monitor, which is the output vs. HS is the horizontal sync signal, it is equal to 25MHz/800 = 31.25KHz. Blank = 0 when the electron beam is in the blanking interval and the output color should be RGB=000. Sync is not. DrawX is a number from 0-639, DrawY is a number from 0-479, they represent the coordinate of the current pixel being drawn.

purpose: This module is used to control the output VGA signal on the FPGA board so that contents can be displayed on an external monitor.

game.sv

input: frame_clk, reset, [15:0] keycode,

output: [9:0] character_pos_x, [8:0] character_pos_y, [10:0] screen_pos, big, idle, walk1, walk2, walk3, jump, direction, lose, transition, [35:0] coin, [3:0] game_state, [5:0] score,

description: It reads the keycode, wasd on the keyboard, and does the following operations: 1. it outputs a character_pos_x and character_pos_y, which is the character's top right pixel's position on the screen, which will be affected by the keycode, character's position on the screen. The screen_pos is the position of the current screen that relates to the entire map, and it's related to the scrolling of the screen when the character proceeds. idle, walk1, walk2, walk3, jump, lose, are several states which will be affected by the keycode the character is in, it will be outputted to the color_mapper to show the corresponding script for the character mario. Coins record the current amount of coins that the mario has got. Score records the current character has got. Game_state records the state of the game it's in. It could be state1, trans1, state2, or over. In state 1, the character hasn't gotten into the tube, and block collision detection, coin calculation, and tube platform calculation are performed to update the output coin, the character_pos_x, character_pos_y, and the score. Transition state is the process of changing the map. state 2 is the map under the tube. It still updates the coin and score, but it also determines the game reaches over state when the character reaches the position where the flag is at. The big state is the bigger Mario after Mario hit the question block. Transition is the state when mario's shift from small to big.

purpose: It updates and outputs the position of character, the position of screen, the current state of mario's motion, the state of the game, the output of coins number and score. It also determines all interaction characters may have with the map including hitting a normal block, hitting a question block, standing on a tube, getting a coin, reaching the end and how these interactions may result in difference in its outputs.

font_rom.sv

input: [10:0] addr

output: [7:0] data

description: A ROM used to store the data of the characters that shall be drawn. There are 127 characters in total. All characters inside are stored in 16 rows, each containing 8 bits. Every time the output is one row of the data stored.

purpose: Used to store the data of all characters.

color_mapper.sv

input: CLK, [8:0]drawY, [9:0] character_pos_x, [8:0] character_pos_y, [10:0] screen_pos, idle, walk1, walk2, walk3, jump, direction, big, lose, transition, [35:0] coin, [3:0] game_state, [5:0] score

output: [3:0] color, [18:0] addr, we

description: This module contains a state machine which controls the state of updating the current state of what should be drawn. waiting, drawBG, drawBG2, drawMario. The VGA is aimed to draw the background in drawBG, then perform drawMario, and then enter the waiting state. The drawBG2 will replace the state of drawBG when the game_state input is no longer 4'h0, which indicates that it’s the second map. Different objects' ROMs such as the rom of cloud, coins, flag, tube, tree, blocks, ground, mario and font are declared to be drawn. The read_address of each Rom where the data about the next pixel that will be drawn is calculated and updated per cycle. Within each state, it will determine at which location the object will be drawn according to the preset location and the screen_pos within the input. In drawMario, mario’s facing, or the second that the pixel is read out from the mario rom, is determined by the direction of the input. The inputs are generally the output of game.sv, and the output addr is the write_addr, which is equal to count, the counter of each state. write_ena for drawMario state will be calculated using the character’s position. we is write_ena, which will be 1’b1 as default and will change to 0’b0 at the waiting state. The color is the color output of the object ROMs.

purpose: A FSM which controls what to update to the buffer, it calculates the position of different objects in different states. It also calculates the read_address of the ROM and gets the color out for the addr.

final_top.sv

input: MAX10_CLK1_50, [1: 0] KEY, [9: 0] SW,

output: [9: 0] LEDR, [7: 0] HEX0, [7: 0] HEX1, [7: 0] HEX2, [7: 0] HEX3, [7: 0] HEX4, [7: 0] HEX5, DRAM_CLK, DRAM_CKE, [12: 0] DRAM_ADDR, [1: 0] DRAM_BA, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [3: 0] VGA_R, [3: 0] VGA_G, [3: 0] VGA_B,

inout: [15: 0] ARDUINO_IO, ARDUINO_RESET_N, [15: 0] DRAM_DQ

description: This serves as the top level. It is generally used to connect the components including the SDRAM, the VGA, the Adruino, the clocks, the sw, the Led and the Hex and external components such as key.

purpose: Use to connect all the hardware components.

Besides these modules, there’s the ROM module for tree, tube, block. Mario, etc. Their inputs are the reading address and CLK, and outputs are [3:0] data_Out which is the index that input to the color_palette.sv.

Platform Designer Description

All the components in Platform Designer are the same as Lab 6.2 except the keycode PIO. The keycode PIO is modified to support reading two key presses simultaneously. The platform designer view can be found in the block diagram section above. The description of the following components is from our Lab 6.2 report:

nios2_gen2_0: This is the Nios II/e processor used to execute C programs. The inputs Clk and reset are from the FPGA. The data_master is connected to the SDRAM to write and read data needed for the program. The instruction_master is connected to the SDRAM to retrieve instructions.

onchip_memory2_0: This is the 16-bit on-chip memory. It is not used in our final project since the on-chip memory is reserved for the FPGA as the frame buffer. The program on Nios II is run on the SDRAM.

sdram: This is the Synchronous Dynamic RAM used by the Nios II processor to store data and instructions to execute the program. It has 512 Mbits of size.

sdram_pll: This is the sdram controller. It adds a 1ns phase to the input clock from FPGA and feeds the output clock that lags the FPGA clock to the SDRAM. This is needed to compensate for clock skew in board layout. The 1ns delay guarantees that all synchronous signals are stabilized at the rising clock of the SDRAM clock.

sysid_qsys_0: This module produces a serial number which will be checked by the software loader before software execution. It prevents loading software onto an FPGA with incompatible Nios II configuration.

usb_irq, usb_gpx, usb_rst: These are 1-bit parallel I/O blocks that are necessary for the USB keyboard to function correctly.

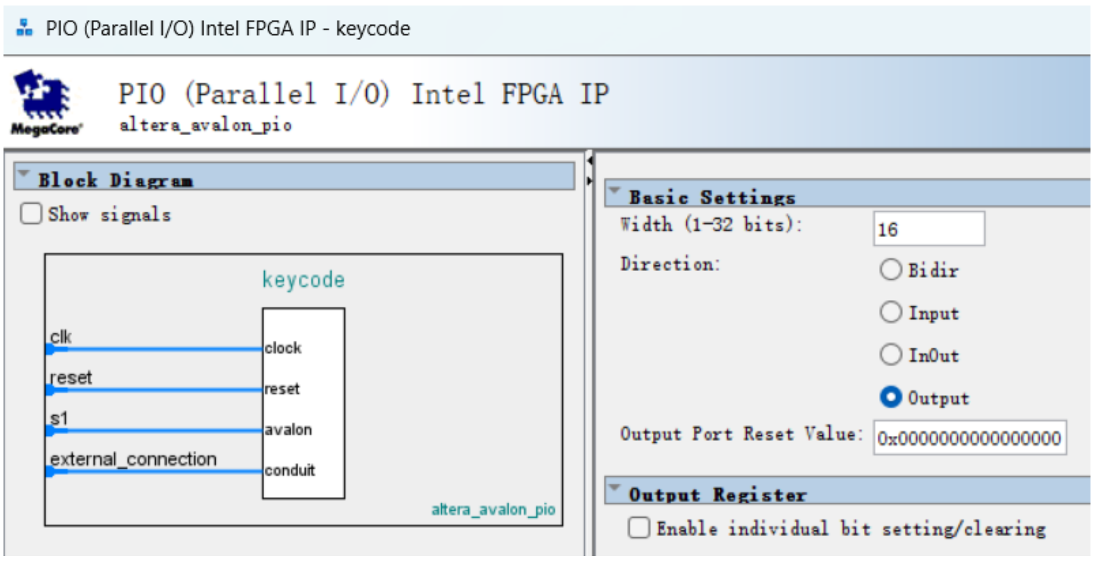
hex_digits_pio: This is a 16-bit parallel I/O block. It is assigned a base address of 0x0170 to 0x017f. Each 4 bit in the 16-bit value represents a hexadecimal number from 0-F. This block is externally exported to the hexdrivers in the FPGA top module so that the current keycode can be displayed on the on-board hex displays.

leds_pio: Used to produce on-board LED output.

timer_0: This is a timer that sends out a signal per millisecond. This is used by the Nios II processor for USB timeout.

spi0: This is an SPI block. It is used for the data transfer in master mode. The Nios II (master) can send data to the MAX3421E (slave) through MOSI, and it can read data from the slave through MISO. SS (slave select) is active low. This block can be controlled by the Nios II using the alt_avalon_spi_command() function.

The modified keycode PIO:


PIO (Parallel I/O) Intel FPGA IP - keycode

PIO (Parallel I/O) Intel FPGA IP
altera_avalon_pio

Block Diagram
Show signals

Basic Settings
Width (1-32 bits): 16
Direction: Bidir / Input / InOut / [✓ Output]
Output Port Reset Value: 0x0000000000000000

Output Register
Enable individual bit setting/clearing

Figure 15. Modified Keycode PIO

keycode: This is a 16-bit parallel I/O block. It is assigned a base address of 0x0180 to 0x018f. It is the input of the first two keys read from the USB keyboard represented in 8 bits * 2. The top-level module in SystemVerilog can then divide the 16-bit value into two separate 8-bit keycodes so that two inputs can be processed in the same cycle. Using this modification, the character can jump and move at the same time.

Software Description

The Nios II system is used only for reading USB keyboard inputs. The software used is the same as in Lab 6.2. There is only one modification to the program, in the main.c file:

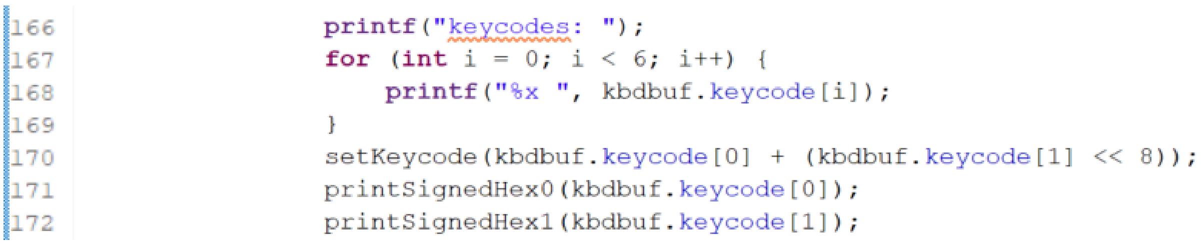

```
166                 printf("keycodes: ");
167                 for (int i = 0; i < 6; i++) {
168                     printf("%x ", kbdbuf.keycode[i]);
169                 }
170                 setKeycode(kbdbuf.keycode[0] + (kbdbuf.keycode[1] << 8));
171                 printSignedHex0(kbdbuf.keycode[0]);
172                 printSignedHex1(kbdbuf.keycode[1]);
```

Figure 16. Software Modification

The modification is on line 170. The argument of the setKeycode function used to be kbdbuf.keycode[0], which is a single keycode value. Since the setKeycode function supports an input of 16 bits, we assigned the highest 8 bits to be the second keycode input and the lowest 8 bits to be the first keycode input. With this modification, the FPGA modules can read two keyboard inputs from the PIO at the same time.

Design Timeline

11/7-11/13: Finding resources for Mario and various objects.

11/14-11/20 (mid-checkpoint): implement Mario character, its motion (controlled by the USB keyboard), gravity, the animation of Mario when walking and jumping.

11/21-11/27: Implement a moving frame according to mario, complete the design of the first game world and its background, add basic interaction with Mario, implement winning conditions.

11/28-12/4: addition to the background, mario's transition from small to big

12/5-12/8: add a second map, add the tube, coins, scores, and professor's picture, final test and debug

Design Resources & Statistics

LUT	8.709
DSP	0
BRAM	1,438,940 bits
Flip-Flop	2,786
Max Frequency	19.18 MHz
Static Power	97.26 mW
Dynamic Power	217.78 mW
Total Power	337.24 mW

Conclusion

In this project, we successfully realized the basic features of the classic game "Super Mario", such as the scrolling game world, gravity, animation, coins, score counter, level switching, and dynamic Mario sprite. We have thought of some improvements that can be achieved if there is more time. First is to add some moving monsters to the map. The basic idea would be the same as collecting coins, the monster sprite gets changed when vertical contact is detected. The demanding part would be adding a lose animation and state. Second, audio can be added to make the game more immersive. The classic Mario background music can be added as well as some audio effects when jumping or collecting coins. Also, an SD card can be used to store more sprites since we used almost all on-chip memory bits in our project. Overall, the result of this project is satisfactory and the designing process was rewarding.