ECE 385

Fall 2022

Final Project

Contra

Nitish Bhupathi Raju, Sean Ramirez
December 14, 2022
Hongshuo Zhang

Introduction:

Serving as the culmination of the concepts rehearsed within ECE 385, the final project emulates the 1989 NES port of the video game Contra. Employing On-Chip Memory as the primary domain wherein the program elements exist, the project possesses player-enemy control logic, finite state machines governing animations and game state logic, font rom-generated tile placement, and imported sprites reflecting the elements present within the original NES game. SOC design elements from Labs 6 and 7 were also present within the project, with the VGA and USB_KB peripherals being used for the purposes of displaying the sprite elements and enabling game control, respectively. Relating to the visual elements of the project, each visual had been modified, such that the number of bits used to represent the palette of the image were limited to essential colors, and compressed, such that the original size of the image imported to the project was reduced to limit the space occupied within the On-Chip Memory. Each converted image is accompanied by three additional .sv files necessary to generate the visual element via the VGA: a color palette .sv file, a rom .sv file, and a color mapper .sv file. These three generated files govern the visual characteristics of all images present within the project. In addition to the .sv files compiled within the project, the .qsys peripherals comprise all past platform designer elements from Lab 6 and Lab 7; there are no peripherals that were created specifically for the final project. The product of the labor spent on the project is a single level representative of the first level within the original NES game, two enemy types derived from the same game, player animations, a point and life system, and an easter egg enabled upon satisfying certain conditions upon the completion of the level.

Block Diagram:

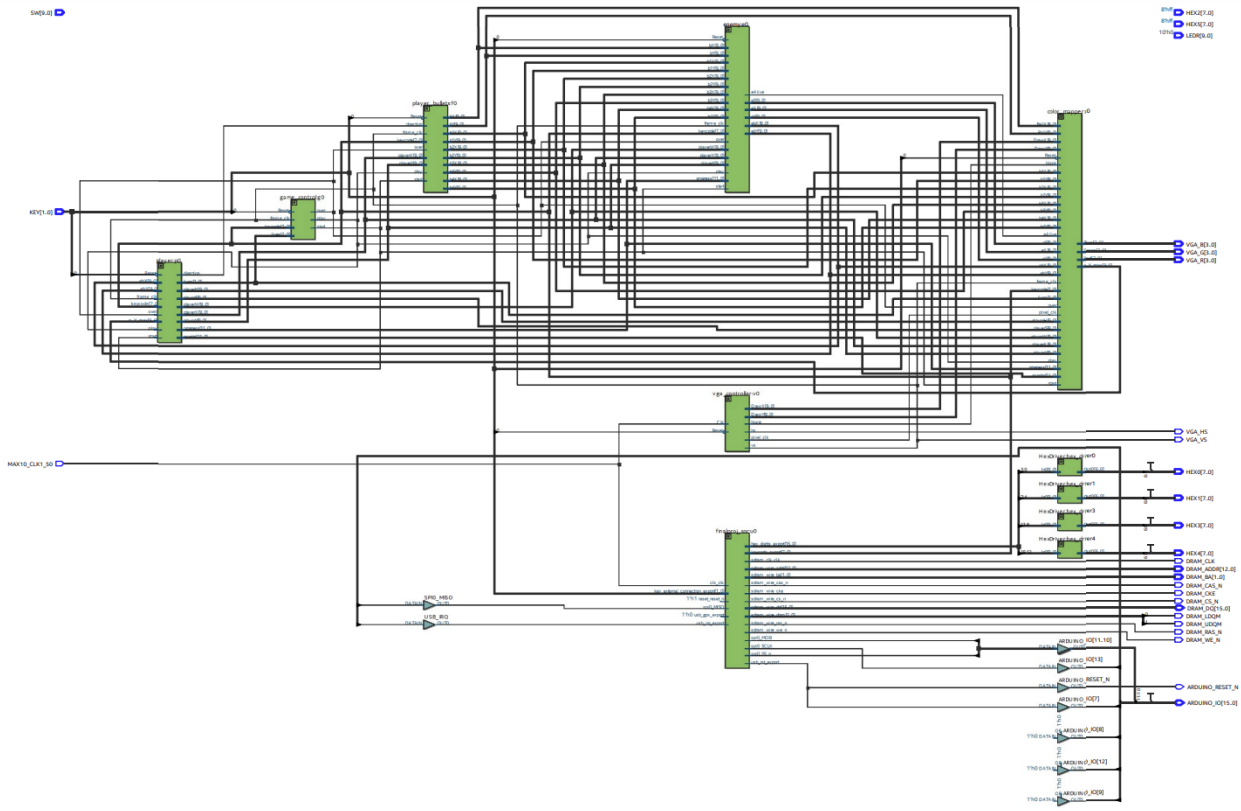


Figure 1: RTL Block Diagram - Overview

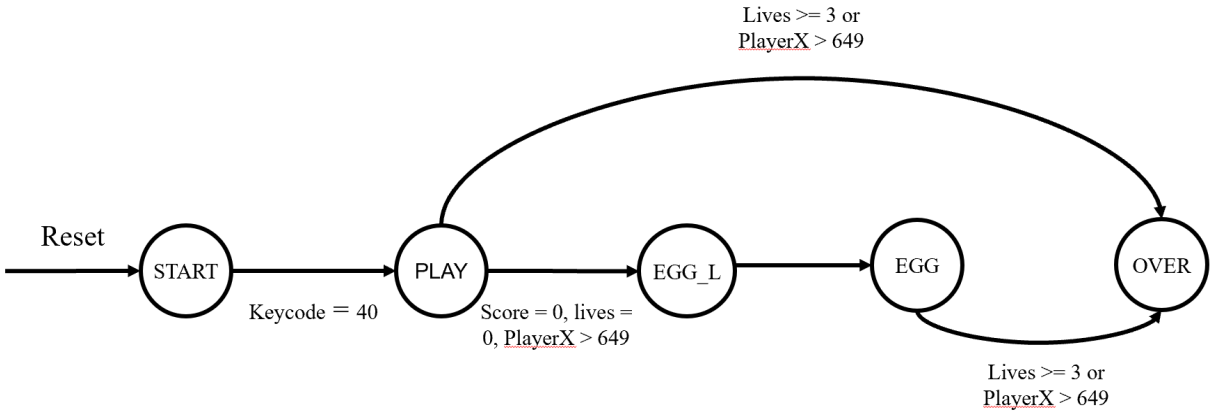


Figure 2: Game Control FSM Logic

Note: Inputs to the FSM are: Reset, Keycode, PlayerX, Lives, and Score. The input combinations indicated on the arc are the only ones that determine the transition; all other inputs are don't cares. The transition to START due to reset can happen from any state.

	play	start	over	egg
START	0	1	0	0
PLAY	1	0	0	0
OVER	0	0	1	0
EGG_L	0	0	0	1
EGG	0	0	0	1

Figure 3: FSM outputs

Written Description of Final Project:

Overview of project:

Pertaining to the aggregation of all elements that comprise the final project, the program functions via referencing the peripherals within the Platform Designer and the generated .sv files included within the project. Relating to the SystemVerilog implementation for the final project, the implementation employs the .mif files that accompany the three .sv file generation when image conversion occurs: in this instance, each sprite within the project retains a respective rom, color_mapper, and palette .sv file, with reference to the rom being facilitated via the respective .mif file for a sprite. The number of total sprites employed within the project totals to 51, with 6 sprites relating to tile generation for the map, 12 employed for the use of representing player animation, 16 employed for the use of conveying enemy animation, 2 employed for use in the "START" and "OVER" states of the game flow, 11 employed for use in displaying the point total for the player, 1 employed for representing the life count for the player, 2 employed for displaying both player and enemy bullets, and 1 employed for use in the easter egg state. Animation for the player is governed via an FSM implemented within "player_animation.sv", whilst the animation for the Running Enemy type is implemented within "enemy_run_animation.sv". Each file retains a sprite array output that is assigned a value representative of the sprite selected via the FSM. This is subsequently allotted to an instantiation of the animations .sv files for their respective player and enemy logic implementations. The "color_mapper.sv" governs all other sprite generation via the VGA, whilst the "final_proj.sv"

contains ancillary FPGA elements, such as the clock, along with object instantiations for all auxiliary files. All .sv files relating to sprites were created via the “Ian’s Tools” utility authored by the ECE385 UA of the same name. Relating to the Platform Designer, all peripherals present within the .qsys file are compiled from prior laboratory assignments.

Background Map:

Our map of Contra has a resolution of 3840x480. This map is quite large to create a sprite for as the data for the map exceeds the maximum amount of On-Chip Memory available. However, we realized that the map for Contra uses a certain number of tiles repeatedly - six, to be exact. The tiles we used are grass tile, boulder tile, water tile, sky tile, wood tile, and tree tile, So, using a tile size of 96x96, we created a 1D array that's 40x6 long (row-major order). The array contains numbers from 0 to 5 to indicate the corresponding tile to draw in that location. Furthermore, to implement scrolling, we would increment a variable when the player is at the center of the screen and the right keycode is pressed, and that variable was used to read the appropriate tiles from the 1D array. The Color Mapper Module (further explained in a later section) would draw the appropriate tiles on the screen based on the sprite data.

Player:

Our player character was initially represented by a 66x40 box on the screen. In the Color Mapper Module, we mapped the appropriate sprite to that box to represent the player on screen. We had multiple sprites for the player, which allowed us to create a running animation. One aspect of the project that we initially struggled with was implementing collisions with platforms for the player (to facilitate jumping and dropping between platforms). Once we created the background rom array, we used the current position of the player to determine the lowest tile beneath the player that was a grass tile, and using that information from background rom, we used a local maximum Y position variable that prevented the player from going past that position while jumping or standing. This allowed us to implement collisions quite smoothly. Regarding jumping, we had a Y motion variable that was set to a certain negative value when the jump keycode was pressed. This motion variable, that essentially served as vertical velocity, was incremented every frame, which allowed us to simulate gravity in the game. The vertical velocity variable kept incrementing until the player reached the maximum Y position (which could change based on the current position of the player). Once the player reached the maximum position, their vertical velocity was set to zero. This allowed us to implement gravity and platform collisions quite effectively. Furthermore, we made a decision to only give the player five bullets that they code shoot repeatedly (since every additional bullet for the player needed to be instantiated).

Figure 4: Player Sprite

Enemy:

For the first level, we decided to implement two enemies. One enemy would walk back and forth while shooting every two seconds, while the other enemy would stand still and shoot in the direction of the player. To implement interaction between enemy and player bullets, the position of the enemy bullets served as inputs to the player module (and vice versa). If the bullets overlapped with the position of the character, that would be registered as a hit. We implemented some local flags to ensure that each bullet would register only as one hit. Furthermore, to avoid multiple instantiates of the same enemy type, we simply teleported the enemy to a further point in the map once they were dead. This meant we only needed one instantiation for each enemy in our top-level.

Figure 5: Standing Enemy Sprite

Figure 6: Running Enemy Sprite

Game Logic:

If the player lost three lives (one life is lost per hit or everytime the player falls through the bottom), the game would end. The game would also end if the player surpassed the end of the map. Meanwhile, every enemy the player hits causes an internal score counter to increment by 10. This score is displayed on the Game Over screen, which is displayed once the game ends.

Easter Egg:

To have some fun with the project, we decided to implement an Easter Egg in the game. If the player finishes the game with all lives remaining and a score of zero (which means no enemies were killed), the player is transported to a bonus level. In this bonus level, the background consists of an image of our professor. The player can run around in this level until they reach the end, at which point the Game Over screen is displayed.

Description for SV Modules:

Module: finalproj.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW, inout [15:0] DRAM_DQ, inout [15:0] ARDUINO_IO, inout ARDUINO_RESET_N
Outputs: [7:0] HEX0, HEX1, HEX2, HEX3, HEX4, 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
Description: The module governs the wiring for each of the employed objects present within the final project. The logic encapsulates instantiations for the "vga_controller.sv", "player.sv", "color_mapper.sv", "enemy.sv", "enemy_run.sv", "player_bullets.sv", and "game_control.sv" files. Moreover, the logic variables for each of these instantiations are also initialized within this module: the variables include 'eY', 'eW', 'eH', 'eX', 'ebX', 'ebY', 'e1sprite', 'eRY', 'eRW', 'eRH', 'eRX', 'eRbX', 'eRbY', 'eR1sprite', 'eAlive', 'eRAlive', 'start', 'play', 'over', 'lives', 'score', 'scroll', 'h1', 'h2', and 'egg'.
Purpose: The purpose of this module is to facilitate interactions between essential game elements, such as sprite generation, player-enemy logic, animations, and game state control.

Module: game_control.sv
Inputs: Reset, frame_clk, h1, h2, [2:0] lives, [7:0] keycode, [9:0] playerX
Outputs: start, play, over, egg, [7:0] score
Description: This module contains the FSM that controls the current game state of the program. The states, initialized within a packed enumerated logic array, consist of 'START', 'PLAY', 'OVER', 'EGG', and 'EGG_L'. Upon program execution, the current state of the program is set to 'START', with local logic variables 'score' and 'change' being set to '0'. During program execution, if any one of the the two enemy types is struck, defined by 'h1' and 'h2', then the 'score' variable is incremented by '10'. Moreover, if the player's current x-axis position, defined by 'playerX', exceeds 10'd649, which is 10 pixels over the drawing limit for the VGA, then the 'change' variable is set to 1'b1 to signify completion of the level. These logic variables also define the FSM logic: if the keycode array is equal to 8'd40, equivalent to pressing the Enter key on the connected keyboard, then the 'START' state ventures into the 'PLAY' state; if lives have been lost, or if the score has been incremented, then the current state will venture to 'OVER' if the player either loses all lives or completes the level; if the current score is '0', no lives have been lost, and level completion is true, then the current state will progress to the gap state 'EGG_L'; if the current state is 'EGG_L', then the subsequent states will be 'EGG', regardless of input; if the current state is 'EGG', then the easter egg will be active, and the typical conditions for "loss" and "win" are applicable; if the current state is 'OVER', then the point totals for the player will be displayed. The program will remain at the 'OVER' state once reached; there is no

further game control logic to be executed. Complementing the state logic, there exist local logic variables ‘start’, ‘play’, ‘over’, and ‘egg’ that dictate VGA drawing within “color_mapper.sv”.
Purpose: The purpose of this module is to govern game state logic–this comprising the title screen, gameplay, and point totalling portions of the game. Moreover, the easter egg state is reached upon fulfilling certain conditions during gameplay.

Module: background_rom.sv
Inputs: [9:0] DrawX, [9:0] DrawY, [9:0] playerX, [9:0] playerY, [11:0] progress
Outputs: [9:0] p_Y_max, [2:0] tile
Description: This module possesses the tile placement logic that is subsequently drawn by “color_mapper.sv”. The map generated during the ‘PLAY’ state is represented by a 1D array of 200 elements titled “ROM”. Each of the array elements represent a 96x96 tile to be placed within the background during gameplay: ‘0’ represents “Sky” tiles; ‘1’ represents “Water” tiles’ ‘2’ represents “Platform” tiles; ‘3’ represents “Rock” tiles; ‘4’ represents “Tree” tiles’ and ‘5’ represents “Tree Top” tiles. Moreover, local logic variables ‘x_address’, ‘y_address’, ‘rom_address’, ‘p_x’, and ‘p_y’ are employed to track the x-position of the current pixel drawn, y-position of the current pixel drawn, the current tile drawn, and the x- and y-positions of the player character, respectively. The tile to be drawn is selected via indexing the “ROM” array with the ‘rom_address’ variable following assignment via $x_{address} + y_{address} * 40$. Additionally, the platform logic for the player character is present within this module: if the current player y-position, defined in 96 pixel intervals, is less than the top-most tile level on the map and corresponds, in tandem with the player’s x-position, to the same ‘rom_address’ as a platform tile, then the player will “stand” on the platform. The player begins to “fall” if their current position does not correspond to a platform tile.
Purpose: The purpose of this module is to govern the tile placement for the level upon which the game elements exist. Sprite generation for the tiles is subsequently achieved via “color_mapper.sv”.

Module: ball.sv
Inputs: Reset, frame_clk, [7:0] keycode, [9:0] playerX, [9:0] playerY, f, direction, scroll
Outputs: [9:0] BallX, [9:0] BallY, bfiring
Description: This module governs the motion of bullets on-screen. The impetus for firing is governed by the “enemy.sv”, “enemy_run.sv” and “player.sv” modules. To aid in scrolling bullets, the module initializes the local logic variables ‘Ball_X_Pos’, ‘Ball_X_motion’, ‘Ball_Y_Pos’, ‘Ball_Y_Motion’, ‘Ball_Size’, ‘firing’, ‘subtract’ ‘ball_step’, and ‘Ball_Size’. Upon program execution, the x- and y- motion of all bullets is set to 10’d0. All bullets present on-screen are first generated at positions relative to either the player’s or enemy’s x-position and placed outside of the drawing interval (10’d485) for the VGA. Upon receiving either an ‘f’ variable–set to logic level ‘1’–or the corresponding keycode indicating a registered press of the spacebar, the player’s bullet is generated at the player’s current x- and y-position on the screen:

the bullet motion is executed via incrementation of the 'ball_step' variable relative to originating position on-screen. The respective ball motion variables simply denote if the ball is currently moving somewhere on-screen. Upon reaching either a boundary position, such as the edge of the drawn screen or an enemy, then the bullet will cease movement and reorient itself to the off-screen position. The scroll variable is used to modify the motion of the bullet relative to the background so that the ball's speed doesn't seem like it's changing arbitrarily.
Purpose: The purpose of this module is to govern the logic corresponding to bullets on-screen. This logic is applicable to both enemy and player bullets.

Module: color_mapper.sv
Inputs: pixel_clk, blank, frame_clk, Reset, start, play, over, egg, [7:0] keycode, [9:0] playerX, [9:0] playerY, [9:0] playerS, [9:0] playerW, [9:0] playerH, [11:0] progress, [2:0] lives, [11:0] psprite, [9:0] DrawX, [9:0] DrawY, [9:0] BallX, [9:0] BallY, [9:0] b1X, [9:0] b1Y, [9:0] b2X, [9:0] b2Y, [9:0] b3X, [9:0] b3Y, [9:0] b4X, [9:0] b4Y, [9:0] eY, [9:0] eW, [9:0] eH, [12:0] eX, eAlive, [9:0] ebX, [9:0] ebY, e1sprite, [9:0] eRY, [9:0] eRW, [9:0] eRH, [12:0] eRX, eRAlive, [9:0] eRbX, [9:0] eRbY, eR1sprite, [11:0] esprite, [7:0] score
Outputs: [9:0] p_Y_max, [7:0] Red, [7:0] Green, [7:0] Blue
Description: This module governs the rendering of sprites using the RGB values, rom file, and rom address associated with each image. The rendered sprites include all sprites previously described within the 'Written Description' section of the report. To aid in properly displaying the imported sprites, each image is accompanied by a series of local variable initializations: 4-bit packed arrays corresponding to the Red, Green, and Blue values present within the associated color palette for the sprite, 4-bit packed arrays representing the rom addresses assigned with the corresponding on-screen address (a packed array of varying length for various sprites and determined via the present x- and y-positions of the sprite on-screen), and at least 3 4-bit packed arrays representing the Red, Blue, and Green values as determined by the corresponding sprite color_palette. Each sprite within the project also retains two object instantiation within this module–these being objects for the respective "...color palette.sv" and "...rom.sv" files. Each rom object is assigned the "pixel_clk" variable for the '.clock' port, its corresponding on-screen address for the '.address' port, and outputs the appropriate rom address ('.q' output port) to its associated rom address array variable; the color palette object is assigned the now populated rom address array and all RGB arrays relating to the object's sprite. Likewise, all "sprite appearance flags", such as "life_on" and "player_on", are assigned based upon conditional statements relating to their appearance, such as the absence of bullet collision for enemies or appearance thereof. Moreover, the "Red", "Green", and "Blue" signals for the VGA display, corresponding to the sprites that should appear on-screen, are assigned depending upon the sprite palette outputs for each RGB array: certain hues are omitted from VGA drawing, as all sprites retain "neon-green" borders surrounding the sprite itself. Generally, if the "red" color array value for a sprite is 4'h0, or if the "green" color array value is less than 4'h7, then the sprite is not rendered during the pixel clock interval.

Purpose: The purpose of this module is to properly render the sprites appearing on-screen via the VGA RGB signals and the associated .sv files for each imported sprite.

Module: enemy.sv
Inputs: Reset, frame_clk, start, play, over, egg, [7:0] keycode, [11:0] progress, scroll, [9:0] b1X, [9:0] b1Y, [9:0] b2X, [9:0] b2Y, [9:0] b3X, [9:0] b3Y, [9:0] b4X, [9:0] b4Y, [9:0] playerX, [9:0] playerY, [9:0] playerW
Outputs: [9:0] eY, [9:0] eW, [9:0] eH, [12:0] eX, eAlive, [9:0] ebX, [9:0] ebY, e1sprite, h1
Description: This module comprises the player-tracking logic, hit detection, firing logic, pivoting logic, and death-alive status of the static, pivoting enemy within Contra. The local logic variables ‘enemyX’, ‘enemyY’, ‘bound_check’, ‘hit’, ‘fire’, ‘d’, ‘dead_check’, and ‘k’ are employed to aid in determining enemy behavior; moreover, the integer variables ‘colx1’, ‘coly1’, ‘colx2’, ‘coly2’, ‘colx3’, ‘coly3’, ‘colx4’, ‘coly4’, ‘colx5’, ‘coly5’, ‘enemyW’, ‘enemyH’, ‘pde’, ‘counter’, and ‘advance’ also aid in determining enemy behavior. Upon program execution, the enemy position is set to the first predetermined location on the map: as the player progresses throughout the level, the ‘boundary_check’ variable is employed to track whether the player has traversed a full 640 pixels along the horizontal direction. If this is true, then the enemy will “respawn” at a new predetermined location along the map. This “respawning” action is conducted a total of four times for the pivoting enemy. To ensure that the enemy-to-player distance is conserved with player movement, the ‘progress’ variable is employed to track the current player x-position relative to the enemy’s x-position. This player-to-enemy distance is similarly employed when determining the ‘pde’ value: if the player’s x-position is greater than the enemy’s x-position, then the ‘pde’ value is greater than ‘0’, and the enemy orients itself to the right to face the player. The converse of this is applicable to left-wise orientation of the enemy. Firing bullets is governed by a ‘counter’ variable that, upon incrementing to 30 from 0, enables ‘fire’: this is subsequently placed into a “ball.sv” object every clock cycle. If the ‘hit’ variable is set to ‘1’ or boundary conditions place the enemy below the threshold to be drawn on-screen, then the ‘e1sprite’, governing whether the enemy should be drawn, is set to ‘1’; likewise, the ‘eAlive’ variable is set to ‘0’, and the ‘dead_check’ variable is set to ‘0’. The ‘col’ series of variables governs whether any one of the five bullets that may be generated by the player collide with the enemy via aligning with its current coordinates: the ‘hit’ variable is set to ‘1’ if a collision is registered.
Purpose: The purpose of this module is to govern the gameplay logic for the pivoting enemy.

Module: enemy_run.sv
Inputs: Reset, frame_clk, start, play, over, egg, [7:0] keycode, [11:0] progress, scroll, [9:0] b1X, [9:0] b1Y, [9:0] b2X, [9:0] b2Y, [9:0] b3X, [9:0] b3Y, [9:0] b4X, [9:0] b4Y, [9:0] playerX, [9:0] playerY, [9:0] playerW
Outputs: [9:0] eRY, [9:0] eRW, [9:0] eRH, [12:0] eRX, eRAlive, [9:0] eRbX, [9:0] eRbY, eR1sprite, h2, [11:0] esprite

Description: This module comprises the logic elements necessary to properly emulate the behavior of the running enemy from Contra. The local logic variables initialized to aid in this process are ‘enemyRX’, ‘enemyRY’, ‘bound_check’, ‘hit’, ‘fire’, ‘d’, ‘dead_check’, ‘switch_dir’, ‘still’, ‘move’, and ‘k’. Moreover, the integer variables ‘colx1’, ‘coly1’, ‘colx2’, ‘coly2’, ‘colx3’, ‘coly3’, ‘colx4’, ‘coly4’, ‘colx5’, ‘coly5’, ‘enemyRW’, ‘enemyRH’, ‘pde’, ‘counter’, ‘frame_coutner’, ‘advance’, and ‘still_c’ also aid in determining enemy behavior. The module bears similarities to the “enemy.sv” file: the collision detection system for determining ‘hit’, ‘dead_check’ conditional statements, and end-of-file assignments of positions are syntactically the same as those found within “enemy.sv”. Relating to the differences, the initial enemy positions undergo displacement via conditions placed upon the ‘move’ variable: if the ‘frame_counter’ variable is incremented to 166, then the enemy switches directions via the ‘switch_dir’ flag; else, then the current flag value for ‘switch_dir’ remains unchanged. In both instances, the ‘move’ variable is temporarily set to 13’d0, reflecting no movement. If the ‘switch_dir’ variable is set to ‘1’, then the enemy is oriented to move leftwards; else, then the enemy is oriented to move rightwards. Whilst the ‘switch_dir’ variable is set to either ‘1’ or ‘0’, the ‘frame_counter’ is incremented or decremented, respectively. Pertaining to enemy movement, the ‘progress’ variable is employed in the same manner as found within “enemy.sv” with the addition of the ‘move’ variable displacement. An object instantiation of “enemy_run_animation.sv” is found within this module.
Purpose: The purpose of this module is to govern the gameplay logic for the running enemy.

Module: enemy_run_animation.sv
Inputs: Reset, frame_clk, direction, [7:0] keycode
Outputs: [11:0] esprite
Description: This module contains the FSM corresponding to the running enemy animation from Contra. There are 14 states: L1, L2, L3, L4, L5, L6, R1, R2, R3, R4, R5, R6, SL, SR. “L” corresponds to the left running states, “R” corresponds to the right running states, and “S”' corresponds to the shooting states. If enemy_run indicates that the enemy is shooting, then based on the direction, we transition to the shooting state. Otherwise, we transition to the appropriate running state. The current state of the FSM determines which sprite needs to be drawn through esprite.
Purpose: The purpose of this module is to select the sprite employed in representing running enemy animation.

Enemy Run Animation FSM:

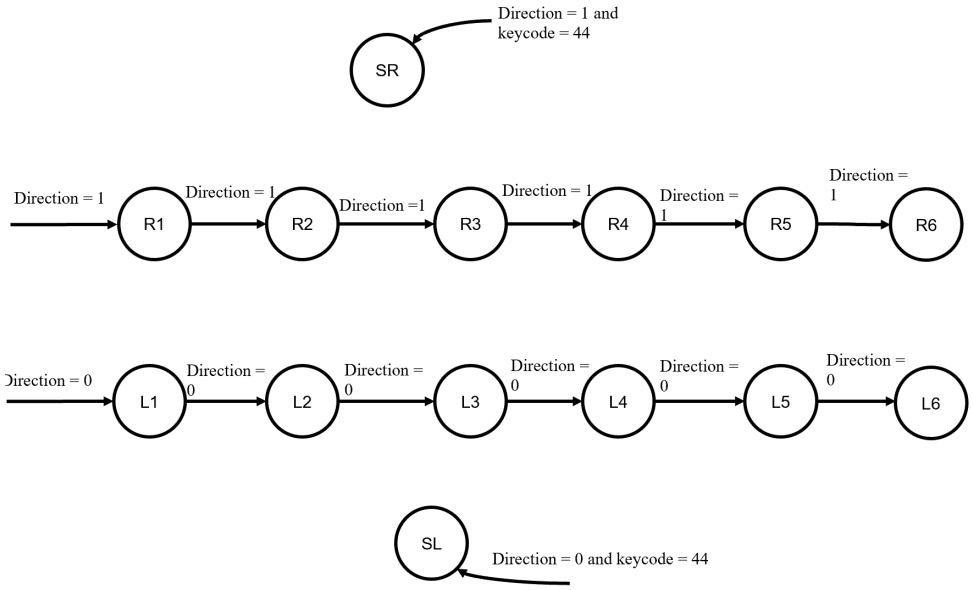


Figure 7: Enemy Run FSM

Note: The inputs to this FSM are direction and keycode. The transition to rR1. rL1, SR and SL can happen from any state.

State	esprite
L1	1
L2	2
L3	4
L4	8
L5	16
L6	32
R1	64
R2	128
R3	256
R4	512
R5	1024

R6	2048
SL	2000
SR	2030

Figure 8: Enemy Run FSM outputs

Module: player.sv
Inputs: Reset, frame_clk, start, play, over, egg, [7:0] keycode, [9:0] p_Y_max, [9:0] ebX, [9:0] ebY, [9:0] eRbX, [9:0] eRbY
Outputs: [9:0] playerX, [9:0] playerY, [9:0] playerS, [9:0] playerW, [9:0] playerH, [11:0] progress, [2:0] lives, direction, scroll, [11:0] psprite
Description: This module contains all player-input logic meant to control the player character actions within the program. In addition to the input variables, the local logic variable initializations include "p_X_Pos", "p_X_Motion", "p_Y_Pos", "p_Y_Motion", "p_Size", "pheight", "pwidth", "p", "jumping", "l", "hit", "c", "dir", and "egg_check". Moreover, the accompanying int and parameter variables are "p_Y_step", "min", "minD", "p_S", "p_W", and "p_H". Upon program execution or reset, the player's motion and jumping flags are set to '0', whilst the current player x-position and y-position are fixed to (20, 100). Likewise, the player character begins facing toward the right, denoted by the "dir" variable being set to '1', and the "egg_check" flag is set to '1'. If either the "play" or "egg" flags are set to '1', then player control is enabled. Player movement is determined via the keycode currently assigned to its respective input port: 'W' will cause the player to jump; 'A' will cause the player to venture leftwards; 'D' will cause the player to venture rightwards; and 'Spacebar' will incur the firing of up to five bullets. No other keys govern player character control. The life system is also coordinated within this module: up to three instances of receiving enemy fire will be permitted, with the "l" variable incrementing once per hit sustained. The "playerX", "playerY", "playerS", "playerW", "playerW", "playerH", "progress", "lives", and "direction" variables will be assigned by the "p_X_Pos", "p_Y_Pos", "p_Size", "pwidth", "pheight", "p", "l", and "dir" variables, respectively. There exists an object instantiation of "player_animation.sv" at the end of the module, into which the current "keycode", "dir", and "jumping" variables are assigned.
Purpose: This module serves to represent the player character in the game.

Module: player_animation.sv
Inputs: Reset, frame_clk, [7:0] keycode, direction, jumping
Outputs: [11:0] psprite
Description: This module is a finite state machine that determines the sprite to be displayed. There are 12 states, represented by the wR, wL, rR1, rR2, rR3, rR4, rR5, rL1, rL2, rL3, rL4, and rL5 variables. Each variable corresponds to a certain sprite. For example, 'w' corresponds to the

wait sprites, while ‘r’ corresponds to the running sprites. “R” corresponds to the right, while “L” corresponds to the left. The transitions between states are determined by the values of the keycode, direction, and jumping signals. For example, if we are in the right running state, it transitions to the next right running state if the right keycode is pressed. If instead the left keycode is pressed, it transitions to the rL1 state. If no keycode is pressed and the player is not jumping, the FSM transitions to the appropriate wait state based on direction. If the player character is currently jumping, the appropriate R1 animation is chosen based on the direction. The pssprite value corresponds to the current state of the FSM.
Purpose: This module serves to create an animation for the player character.

Player Animation FSM:

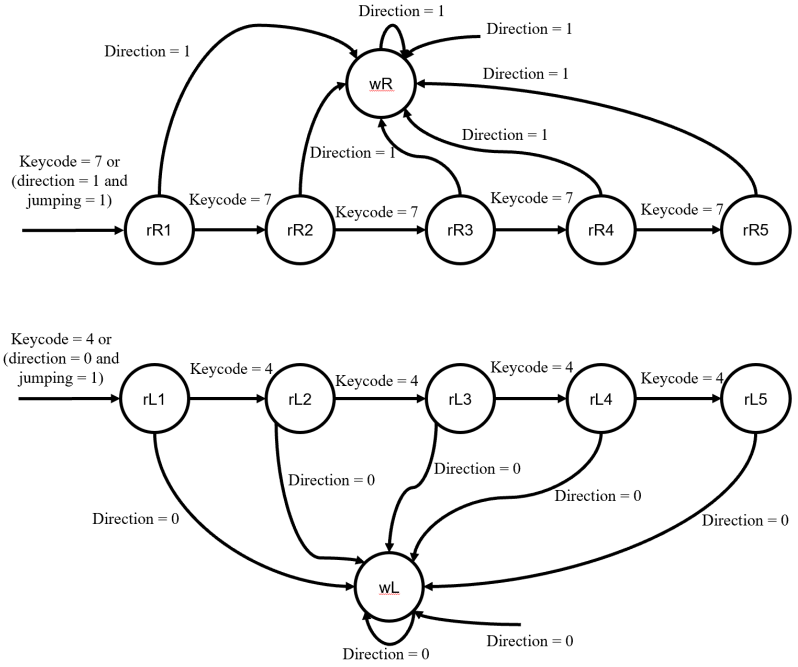


Figure 9: Player Animation FSM

Note: The inputs that determine the transition are indicated on the FSM. All other inputs are don't cares. For example, the transition form rL1 to rL2 only occurs if keycode is 4. Otherwise, the FSM transition to wL if the direction is 0. If the direction changes to 1, it transitions to wR. If keycode = 7, it transitions to rR1. If keycode is 7 once more, it transitions to rR2. Also, these transitions only occur every nine frames.

State	psprite

wR	1
wL	2
rR1	4
rR2	8
rR3	16
rR4	32
rR5	64
rL1	128
rL2	256
rL3	512
rL4	1024
rL5	2048

Figure 10: Player Animation FSM Outputs

Module: player_bullets
Inputs: frame_clk, Reset, [7:0] keycode, start, play, over, direction, scroll, [9:0] playerX, [9:0] playerY
Outputs: [9:0] bX, [9:0] bY, [9:0] b1X, [9:0] b1Y, [9:0] b2X, [9:0] b2Y, [9:0] b3X, [9:0] b3Y, [9:0] b4X, [9:0] b4Y
Description: This module uses internal counters to determine which bullets will be fired and the interval between the bullets. This ensures that upon one press of the space button, all five bullets aren't fired at once. This module contains five instantiations of the ball module since the player only has five bullets they can shoot.
Purpose: The purpose of this module is to organize all the logic for player shooting.

Module: HexDriver
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the hex driver unit. This particular module receives a 4-bit input and subsequently converts the value to a 8-bit format. Conversions are accommodated for values

ranging from 0 to F, in addition to a default High Impedance value case. This module is unchanged from the previous Labs.
Purpose: This module serves as the hex driver unit. The eventual summations of each of the 3 Adders present within this laboratory assignment will be displayed on the seven-segment hex displays present on the FPGA. Conversion from a simple 16-bit binary value to a duo hexadecimal representation is facilitated by this module.

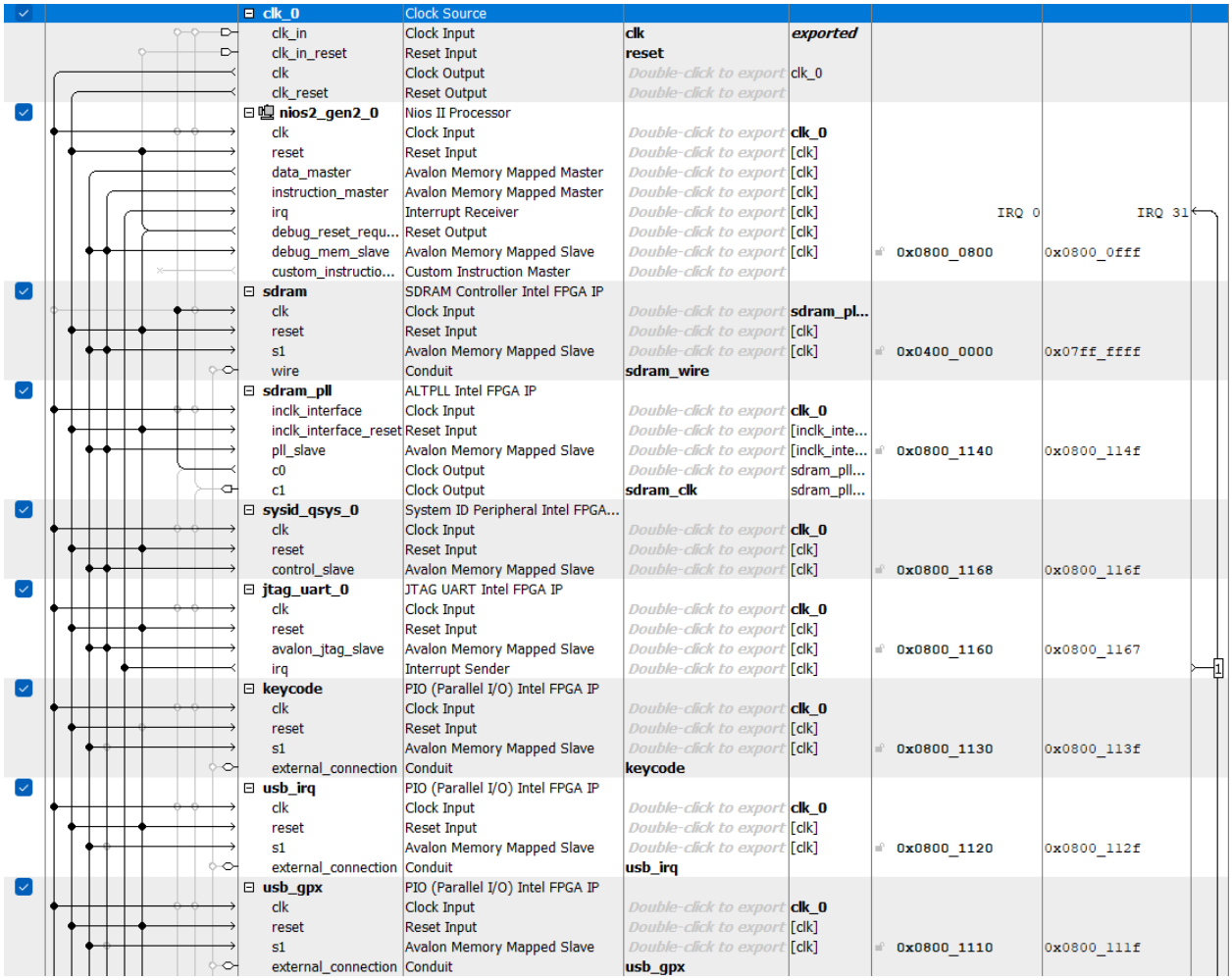
Module: vga_controller
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY
Description: This module governs the horizontal and vertical scrolling exhibited by typical raster order population of pixels on a defined resolution, namely 640x480–with the inclusion of the blanking intervals meant to permit an "electron gun" to recalibrate to the initial frame position, thus totalling to a resolution of 800x525. This behavior is achieved via the use of various locally declared logic and parameter variables: [9:0] hpixels (10'b1100011111), [9:0] vlines (10'b1000001100), [9:0] hc, [9:0] vc, clkdiv, display, and sync (1'b0). DrawX and DrawY are assigned hc and vc, respectively. The module encapsulates four sequential logic blocks whose triggers lie upon the positive edges of the imported Clk or Reset; there also exists a single combinational block appended. Relating to the first sequential block, it comprises a single if-else case statement: if the Reset signal is triggered, then the clkdiv signal, whose purpose is to divide the 50 MHz clock imported by half to achieve a 25 MHz pixel clock, is allotted the value (1'b0). Else, it will be allotted the negative value of itself (~clkdiv). Pertaining to the second sequential block, the hc and vc signal, which represent a horizontal and vertical counter, respectively, are reset to logic level '0' upon the triggering of the Reset signal. Else, the hc and vc counters are merely reset upon reaching their limits, as defined within hpixels and vlines, and continue to increment otherwise. Relating to the remaining sequential blocks, the logic governing the horizontal and vertical sync signals is present within these encapsulations: if the Reset signal is incurred in either case, the hs and vs signals will revert to logic level '0'. Else, if the horizontal and vertical sync signals have yet to reach their respective limit positions, defined as the end of a row for hs and end of the bottom-rightmost non-blanking pixel for vs, then the hs and vs signals remain at logic level '1'. Else, they will revert to logic level '0'. The remaining combinational block facilitates the display of pixels between horizontal plane 0-639 and vertical plane 0-479; if the horizontal and vertical counters reach their defined limits, then display is subsequently disabled until reorientation on a permitted pixel. This module is unchanged from Lab 7.
Purpose: This module facilitates the display of the pixels comprising the total 800x525 resolution, inclusive of the blanking intervals, present within VGA signals processed within Quartus and subsequently printed upon the connected monitor.

Module: sprite_rom.sv
Inputs: clock, [ADDR_WIDTH:0] address

Outputs: [DATA_WIDTH:0] q
Description: For each sprite in our project, a corresponding rom module was created with appropriate ADDR_WIDTH and DATA_WIDTH (which depended on the size of the sprite and number of colors in the sprite). This allowed us to store each sprite on on-chip memory, where each memory location corresponded to the one of the colors in the sprite.
Purpose: All the rom files in our project served to store the data of the sprite in OCM.

Module: sprite_palette.sv
Inputs: [DATA_WIDTH:0] q
Outputs: [3:0] red, [3:0] green, [3:0] blue
Description: Each sprite in our project has a corresponding palette. Depending on the number of colors in the sprite, the palette module has the corresponding RGB output for each palette. The input to this module determines which combination of RGB will be the output.
Purpose: Combined with the corresponding rom modules, the appropriate color can be retrieved from the palette in the Color Mapper Module.

Description of Platform Designer Components:



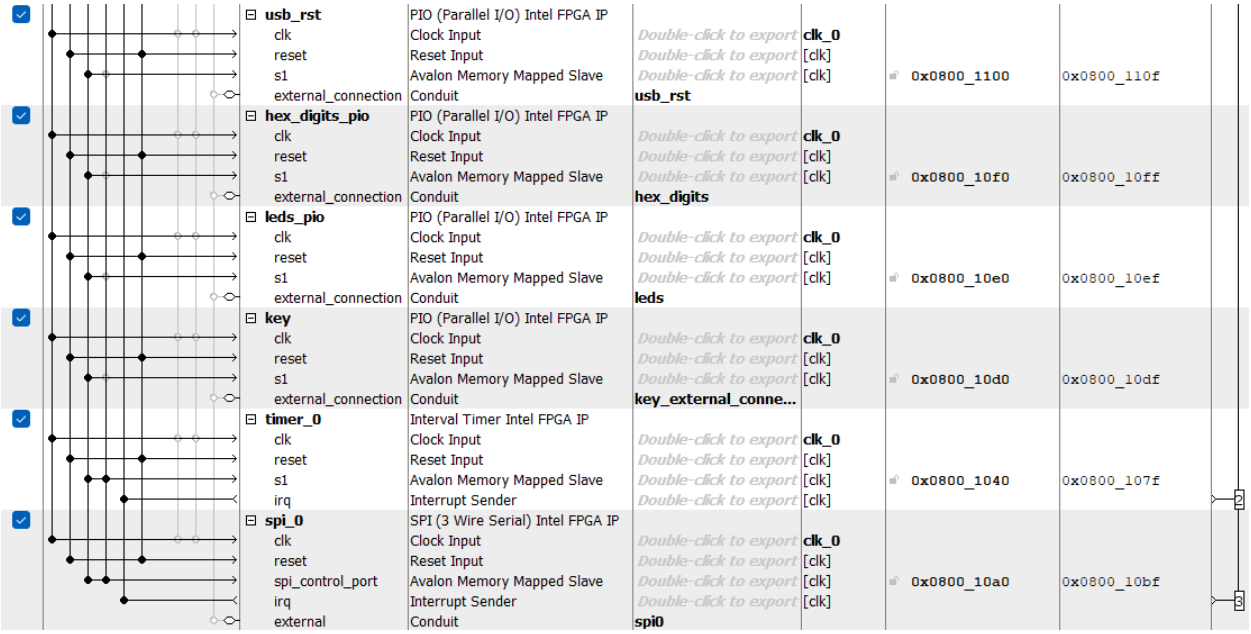


Figure 11: Platform Designer - Peripheral Block Overview

Processor: nios2_gen2_0
Input Ports: clk, reset, irq, debug_mem_slave
Output Ports: data_master, instruction_master, debug_reset_request
Description: This processor consists of the input ports clk, reset, irq, and debug_mem_slave, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, all slave irq ports, and its own data master and instruction master buses. The output ports consist of data_master, instruction_master, and debug_reset_request, whose ports are connected to the all slave peripherals that require data, instructions, and reset request processing, such as sdram, sdram_pll, sysid_qsys_0, jtag_uart_0, keycode, usb_irq, usb_gpx, usb_rst, hex_digits_pio, leds_pio, key, timer_0, spi_0.
Purpose: This processor facilitates the manipulation of data and instructions, in addition to delivery and receiving of data, relating to the NIOS II platform and its associated members, such as the C programs created within Eclipse. This is the central peripheral for the final project.

Peripheral: clk_0
Input Ports: clk_in (clk), clk_in_reset (reset)
Output Ports: clk, clk_reset
Description: This peripheral governs the clk inputs for each of the synchronous peripherals present within the Platform Designer. This is achieved with the imported wires “clk” and “reset” to the ports clk_in and clk_in_reset. The output ports clk and clk_reset govern the transmission of the clock signals for all other synchronous components.

Purpose: This peripheral governs the primary internal clock hailing from the SystemVerilog implementation files, in addition to the asynchronous reset signal triggered by the user.

Memory Interface: sdram
Input Ports: clk, reset, s1
Output Ports: wire (sdram_wire)
Description: This interface functions as an intermediary for the memory interactions with the SDRAM. It consists of the input ports clk, reset, and s1, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 “data_master” and “instruction_master”. The exported connection “sdram_wire” is employed within the top-level entity for the purposes of facilitating proper connections to the SDRAM.
Purpose: This interface encapsulates the operations relating to the data transfers to and fro the instantiated memory or SDRAM. This is essential for the storage of values to be employed within later intervals of one’s program.

Peripheral: sdram_pll
Input Ports: inclk_interface, inclk_interface_reset, pll_slave
Output Ports: c0, c1 (sdram_clk)
Description: This peripheral governs the input ports inclk_interface, inclk_interface_reset, and pll_slave, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 “data_master” and “instruction_master”, respectively. The output port c0 is connected to the sdram “clk”, and the external connection “c1” is exported as “sdram_clk”.
Purpose: This peripheral facilitates phase shifts necessary to correcting discordances in the exchange of data to and fro the SDRAM. The phase shifts extend the time interval in which data transfers should occur.

Peripheral: sysid_qsys_0
Input Ports: clk, reset, control_slave
Output Ports: N/A
Description: This peripheral governs the input ports clk, reset, and control slave, whose connections are between the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 “data_master” and “instruction_master”. There exist no output ports within this peripheral.
Purpose: This peripheral is a system ID checker that ensures the compatibility between the software and the hardware.

Peripheral: jtag_uart_0
Input Ports: clk, reset, avalon_jtag_slave
Output Ports: irq
Description: This peripheral governs the port connections between the NIOS II and its associated program running within Eclipse. It consists of input ports clk, reset, and

avalon_jtag_slave, which correspond to port type Clock Input, Reset Input, and Avalon Memory-Mapped Slave. Likewise, there exists an output port denoted as irq, which is representative of an Interrupt Request Output. The use of the irq port is to prohibit blocking by the CPU when transmitting data to the terminal via the printf commands. The “clk” port is connected directly to the clk_0 peripheral Clock output; this is, likewise, the case with the reset port, whose connection is tethered to both the clk_0 and NIOS II reset buses. The “avalon_jtag_slave” port is connected to both the instruction and data master buses of the NIOS II peripheral, whilst the “irq” port is connected to its NIOS II peripheral counterpart.
Purpose: This peripheral facilitates the use of iostream operations, such as printf, within the Eclipse IDE. Employed for the purposes of maintaining communication between the host computer presently running Eclipse and the NIOS II, in addition to debugging and error handling.

Peripheral: keycode
Input Ports: clk, reset, s1
Output Ports: external_connection (keycode)
Description: This peripheral governs the port connections between the NIOS II and the connected USB keyboard. With the use of the associated .sv files, the input ports, clk, reset, and s1, are connected to the “clk_0” Clock Output, “clk_reset” output, and NIOS II data master bus, respectively. The external connection, keycode, enables use of the peripheral within the .sv file implementations.
Purpose: This peripheral facilitates the use of inputs hailing from a Keyboard employed within Eclipse C programs. Employed within the final project to control player movement and game control logic.

Peripheral: usb_irq
Input Ports: clk, reset, s1
Output Ports: external_connection (usb_irq)
Description: This peripheral governs the use of interrupt requests from the USB Peripheral–this being the keyboard. Its input ports, clk, reset, and s1, are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 data master. Its external connection, denoted by usb_irq, permits use of the peripheral outside of the Platform Designer.
Purpose: This peripheral facilitates the use of interrupt requests to allow the transmission of data from the usb peripheral, this being the connected keyboard, without interference from the CPU, NIOS II, during transmission.

Peripheral: usb_gpx
Input Ports: clk, reset, s1
Output Ports: external_connection (usb_gpx)
Description: This peripheral consists of the input ports clk, reset, and s1, whose ports are

connected to the clk_0 “clk”, clk_0 “clk_reset”, and NIOS II data master, respectively. The external connection is for the purposes of manipulation outside of the Platform Designer.
Purpose: This peripheral facilitates the MAX3421E operation where the output signal selects between 4 possible signals such as operate (which indicates that MAX3421E is ready to operate).

Peripheral: usb_rst
Input Ports: clk, reset, s1
Output Ports: external_connection (usb_rst)
Description: This peripheral consists of the input ports clk, reset, and s1, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and NIOS II data master, respectively. The external connection is for the purposes of manipulation outside of the Platform Designer.
Purpose: This peripheral facilitates the use of resets from the USB peripheral, this being the keyboard employed within the final project, to the NIOS II.

Peripheral: hex_digits_pio
Input Ports: clk, reset, s1
Output Ports: external_connection (hex_digits)
Description: This peripheral consists of the input ports clk, reset, and s1, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 data master, respectively. The external connection is for the purposes of manipulation outside of the Platform Designer.
Purpose: Facilitates access to the onboard HEX display on the FPGA.

Peripheral: leds_pio
Input Ports: clk, reset, s1
Output Ports: external_connection (leds)
Description: This peripheral consists of the input ports clk, reset, and s1, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 data master, respectively. The external connection is for the purposes of manipulation outside of the Platform Designer.
Purpose: Facilitates access to the onboard FPGA LEDs adjacent to the [9:0] SW input switches.

Peripheral: key
Input Ports: clk, reset, s1,
Output Ports: irq
Description: This peripheral consists of the input ports clk, reset, and s1, whose ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 data master, respectively. The output port irq is connected to the nios2_gen2_0 port “irq”.
Purpose: This peripheral facilitates the use of the KEY0 and KEY1 buttons present on the FPGA hardware.

Peripheral: timer_0

Input Ports: clk, reset, s1
Output Ports: irq
Description: This peripheral consists of the input ports clk, reset, and s1. These ports are connected to the clk_0 “clk”, clk_0 “clk_reset”, and nios2_gen2_0 data master. The output port irq is connected to the nios2_gen2_0 “irq” port.
Purpose: This peripheral facilitates the use of timeouts when transmitting information within the NIOS II and USB pairing. If the transmission of data takes far too long, defined as 1 ms, within the final project, then data transmission is ceased.

Peripheral: spi_0
Input Ports: clk, reset, spi_control_port
Output Ports: irq, external (spi0)
Description: This peripheral facilitates the communication of the USB peripheral and the NIOS II. Its input ports, clk, reset, and spi_control_port, are connected to the clk_0 “clk” bus, “reset” bus, NIOS II data master and instruction master, and NIOS II “irq” port. The irq port is connected to its NIOS II counterpart. The external connection, denoted by spi0, is employed within the program’s associated .sv files.
Purpose: This peripheral governs the transmission and handling of the MISO, MOSI, CLK, and SS signals employed when transmitting instructions from the master device, NIOS II, to the corresponding slave devices connected via the instruction and data buses. This is employed within the final project via enabling correspondence between the USB peripheral and the NIOS II, thus permitting data transmission.

Design and Resources Statistics:

LUT	9250
DSP	70
BRAM	608256
Flip-Flop	3047
Frequency	73.66 MHz
Static Power	96.96 mW
Dynamic Power	156.73 mW
Total Power	253.69 mW

Conclusion:

Despite some difficulty in selecting an avenue through which to proceed with the project, such as deciding on necessary tiles, sprites, and game elements to include within the program so as to avoid burdening the On-Chip Memory, the project proceeded smoothly from its initial proposal to its eventual completion during Week 5 of the timeline. The produced project successfully emulates a truncated and playable version of the NES game Contra. Much of the project's success is dependent upon the collaboration held between both project members, in addition to the aid received from the course UAs–especially the eponymous individual who created the "Ian's Tools" sprite utility.

Ultimately, there is little that requires change with respect to the manner in which the project is conducted within the curriculum. From the assistance granted by the course staff during the project period to the feedback received on the submitted proposal, each aspect served to better craft a program that would correctly reflect the strengths of the group members. The standards to which the group members were expected to adhere were met, and each contributed to the project in roughly equal portions, with some components being completed solely by one member and others by both: in most instances, such as the "font_rom" creation for the map, both contributed significantly to the final implementation of a given .sv file. Constant communication between members also aided in the completion of the project, and the periodic assistance from the assigned TA and UA pair similarly propelled the project in the correct direction.