ECE 385

Spring 2023

Final Project

Air Fighter Video Game Report

Kevin Huang (kuanwei2), Steven Chang (sychang5)
TY 10:15 A.M.
Tianhao Yu

Introduction

For our final project of the course, we designed and implemented a fighter jet video game based on the popular 1945 Air Force: Airplane game. The game involves the user controlling a fighter jet using an external keyboard and the W, A, S, D keys. The user-controlled jet is able to move in all two-dimensional directions: forward, backward, left, right, and diagonal, within the bounds of the screen. The user-control jet is also able to fire one bullet at a time using the "space" key. The user-controlled jet will have to avoid getting hit by shots fired by enemy fighter jets and try to shoot down as many enemy jets as possible, racking up points. The game involves three levels, with variations in difficulty. The first level involves the enemy fighter jet moving down to a target position and staying there, firing one shot at a time continuously. The second level involves the enemy fighter jet moving down to a target y-position(s) then shifting left and right on the screen/monitor, bouncing off the edges. The final level involves a larger enemy fighter jet with a simple algorithm for firing missiles toward the user-controlled jet. The key features of our game include the ability of the user-controlled jet to move in all two-dimensional directions (and stay within the bounds of the screen without bouncing off) as well as all jets' ability to fire and get hit (either take damage or get destroyed). We included additional features like a health regeneration kit and the simple missile tracking feature of the enemy as mentioned above. To implement our design, we will utilize both hardware and software code. The software aspect supports the external USB keyboard and user input, while the hardware aspect covers the positions of the jets and various visual elements of the game as well as the movement on-screen and state machine.

Written Description of Final Project System

I. Hardware Components/Description of System (System Level Block Diagram)

The hardware aspects of our design for the final project are pretty similar and based on that of week 2 of lab 6. It includes the NIOS-II processor, SDRAM controller, SDRAM PLL, system ID checker, on-chip memory block, as well as a number of PIO (parallel input/output) peripherals, the JTAG UART peripheral, and an SPI protocol. The NIOS-II processor is the core component that communicates with the peripherals through the Avalon Bus. A significant portion of the on-chip memory is used to store the register locations of graphics and the palette colors of images that we convert from PNG/JPEG to RAM. The SDRAM PLL helps compensate for the clock skew and ensures that the SDRAM runs properly with the SDRAM clock connected to the output of the SDRAM PLL. The system ID checker helps ensure compatibility between the software and hardware aspects of our design. To support movement in all two-dimensional directions and reset the game, we incorporated the key and keycode PIOs. These peripherals interface with the FPGA board and keyboard inputs. The JTAG UART peripheral, on the other hand, enables us to use the terminal of the computer running the program to communicate with the NIOS-II to print statements in the console for debugging purposes. The SPI protocol then helps interface the MAX3421 USB transceiver with the NIOS-II processor and FPGA board in order for data to be transferred from the external keyboard, which is critical in transferring user input. The most important change from lab 6 for our final project is that we added two additional keycode peripherals to allow up to 3 simultaneous key inputs instead of one to reach the requirement of our video game, which is to fire the ammo and move diagonally at the same time. This also helps support potential switching of backgrounds while moving the jet or firing ammos. This ensures that our game runs as smoothly as possible for the user.

II. VGA Operations (Collision, Healthbar, Explotion Animation)

Our final project design consists of several modules tied together through the top-level .sv file, with the color_mapper module ultimately deciding the RGB colors at each specific x and y position on the screen. In addition, the project includes several sv files for controlling the motion and position of each elements, such as the user-controlled jet, enemy jets, user ammo, and enemy ammo. Below contains descriptions for each of these (or groups of these) modules and how they interact with each other in our design implementation.

Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY
Description: This module contains a positive-edge triggered flip flop that will set hc and vc, which are the current drawing pixels to 0 whenever Reset is high. Otherwise, it will then increase hc by 1. If hc reaches the max for horizontal pixel, then it will be set to 0, and vc is incremented by 1. It also sets pixel_clk to the current clkdiv. Not to mention, the module will modify the hs and vs, which are the sync pulses for the display, depending on the current drawing pixel. Last, the display signal will be high only when the hc is between 0-639 and vc is between 0-479.
Purpose: This module determines the sync pulses and pixel_clk for VGA display. It will also run through each pixel and calculate the DrawX and DrawY and blank signals for the display.

Module: Color_Mapper.sv
Inputs: [9:0] BallX, BallY, AmmoX, AmmoY, DrawX, DrawY, Ball_W, Ball_H, Ammo_W, Ammo_H, [9:0] Enemy11X, Enemy11Y, Enemy21X, Enemy21Y, Enem_W, Enem_H, e11_AmmoX, e11_AmmoY, e21_AmmoX, e21_AmmoY, e_Ammo_W, e_Ammo_H, [9:0] Enemy12X, Enemy12Y, Enemy13X, Enemy13Y, e12_AmmoX, e12_AmmoY, e13_AmmoX, e13_AmmoY, [9:0] Enemy22X, Enemy22Y, Enemy23X, Enemy23Y, e22_AmmoX, e22_AmmoY, e23_AmmoX, e23_AmmoY, [9:0] bossX, bossY, boss_AmmoY, boss_AmmoX, boss_AmmoY2, boss_AmmoX2, boss_AmmoY3, boss_AmmoX3, fixX, fixY, flag, e11_flag, e12_flag, e13_flag, e21_flag, e22_flag, e23_flag, boss_flag, boss_flag2, boss_flag3, flagfix, [1:0] phase, [1:0] p11, p12, p13, p21, p22, p23, vga_clk, blank, frame_clk, reset, [7:0] keycode, keycode1, keycode2
Outputs: [3:0] Red, Green, Blue, [5:0] current_level, hit_11, hit_12, hit_13, hit_21, hit_22, hit_23, hit_boss, hit_boss2, hit_boss3, jhit
Description: This module takes in the x and y coordinates of each element in the video games, such as the jet, enemy jets, ammos, health bar, and background in order to determine the color to draw. In addition, the module contains a state machine to determine which level the game should be at and either to draw the start screen, end screen, or the game background. It also takes the input signal keycode to determine which level to stay at and how to move the jet and ammo.

Whenever the ammo and jet meet each other, it will then draw the explosion effect and increase the score.
Purpose: This module is used to draw all the visual elements of the video game and determine the logic in the game, such as when to reduce health or increase score.

Module: ball.sv
Inputs: [7:0] keycode, [7:0] keycode1, [7:0] keycode2, Reset, frame_clk, [5:0] level
Outputs: [9:0] BallX, [9:0] BallY, [9:0] Ball_W, [9:0] Ball_H, [1:0] phase
Description: This is a positive-edge triggered module that will set the jet's position to the center and motion to 0 whenever the Reset signal is high. The module will set the motion of the jet according to the input signal keycode with key A being the motion of moving left, key D being the motion of moving right, key W being the motion of moving up, and key S being the motion of moving down. Whenever the jet's position plus the motion exceeds the boundary, the motion in that specific direction will be set to 0. More importantly, there will be an output signal phase determined by the x motion of the jet to indicate which sprite (phase) of the jet to be drawn.
Purpose: This module controls the position, motion, and phase of the jet depending on the reset and keycode signals.

Module: enemy.sv
Inputs: Reset, frame_clk, [5:0] level, [5:0] jetlevel, [9:0] Enemy_X_Original, [9:0] Enemy_Y_Original, [9:0] target_Y
Outputs: [9:0] EnemyX, [9:0] EnemyY, [1:0] phase
Description: This is a positive-edge triggered module that will set the enemy jet's position to the specific location and motion to 0 whenever the Reset signal is high. The module will control the motion of the enemy jet. If the current game level matches the jetlevel of the enemy jet, then the module will first set the enemy jet's y motion to 3. Once the enemy jet is at the target y, the y motion will be set to 0. Then, depending on the level of the enemy jet, the module will then control its x motion. Whenever the jet's position plus the motion exceeds the boundary, the motion in that specific direction will be set to the opposite direction. More importantly, there will be an output signal phase determined by the x motion of the enemy jet to indicate which sprite (phase) of the enemy jet to be drawn.
Purpose: This module controls the position, motion, and phase of the enemy jet depending on the reset and current game level.

Module: boss.sv
Inputs: Reset, frame_clk, [5:0] level, [9:0] Enemy_X_Original, [9:0] Enemy_Y_Original
Outputs: [9:0] EnemyX, [9:0] EnemyY
Description: This is a positive-edge triggered module that will set the boss' position to the specific location and motion to 0 whenever the Reset signal is high. The module will control the motion of the boss. If the current game level is at the boss level, then the module will first set the boss's y motion to 3. Once the boss is at the target y, the y motion will be set to 0. Then, the module will then control its x motion. Whenever the boss' position plus the motion exceeds the boundary, the motion in that specific direction will be set to the opposite direction.
Purpose: This module is used to control the position and motion of the boss depending on the reset and current game level.

Module: ammo.sv
Inputs: [7:0] keycode, [7:0] keycode1, [7:0] keycode2, Reset, frame_clk, jhit
Outputs: [9:0] AmmoX, [9:0] AmmoY, [9:0] Ammo_W, [9:0] Ammo_H, flag
Description: This is a positive-edge triggered module that will set the ammo's position to the calculated position and motion to 0 whenever the Reset signal is high. The module will determine whether the ammo is fired or not based on the key code. If the input key is "space" , then the ammo will be fired with y motion being -5 and the flag will be high. Whenever the ammo's position plus the motion exceeds the boundary or jhit signal is high, meaning the ammo hits an object, the motion in that specific direction will be set to 0 and will be returned back to the original position waiting for another space key to be pressed.
Purpose: This module is used to control the position, motion, and whether to draw the ammo or not based on the keycodes and reset signal.

Module: enemy_ammo.sv
Inputs: Reset, frame_clk, [5:0] level, hit, [5:0] ammolevel, [9:0] BallX, [9:0] BallY, [9:0] Ball_W
Outputs: [9:0] AmmoX, [9:0] AmmoY, flag
Description: This is a positive-edge triggered module that will set the enemy ammo's position according to the given enemy position and motion to 0 whenever the Reset signal is high. The module will determine whether to fire the enemy ammo or not based on the ammo level and the current game level. If they are the same, then the enemy ammo will be fired with the y motion being 8 and the flag being high. If the ammo exceeds the boundary or the hit signal is high, meaning that it hits the jet, then it will be set to the current enemy jet position and fired again.
Purpose: This module controls the position, motion, and whether to draw the enemy ammo or not based on the current level and reset signal.

Module: boss_ammo.sv
Inputs: Reset, frame_clk, [5:0] level, hit, [5:0] ammolevel, [9:0] BallX, [9:0] BallY, [9:0] Ball_W, Ammo_X_Start, Ammo_Y_Start, jetX, jetY
Outputs: [9:0] AmmoX, [9:0] AmmoY, flag
Description: This is a positive-edge triggered module that will set the boss ammo's position according to the given boss position and motion to 0 whenever the Reset signal is high. The module will determine whether to fire the boss's ammo or not based on the current game level. If it is at the boss level, then the boss's ammo will be fired with the x motion being the displacement between the Ammo_X_Start and jetX and the y motion being the displacement between the Ammo_X_Start and jetX in order to fire the ammo towards the jet, and the flag will be high. If the ammo exceeds the boundary or the hit signal is high, meaning that it hits the jet, then it will be set to the current boss position and fired again.
Purpose: This module is used to control the position, motion, and whether to draw the boss ammo or not based on the current level and reset signal.

Module: fix.sv
Inputs: [5:0] level, Reset, frame_clk
Outputs: [9:0] fixX, fixY, flagfix

Description: This is a positive-edge triggered module that will set the position of our health regeneration kit at the left of the screen once the current level is at level 2. Once the player enters level 2, then the flag will be on and starts moving toward the right of the screen. If it exceeds the boundary, then we set the flag to low to disable the display.
Purpose: This module is used to control the position, motion, and whether to draw the health regeneration kit or not based on the current level and reset signal.

One key aspect of our design is a state machine to control the stages/levels in our game. We opted to write our state machine inside the color mapper module stead of the top-level module or in a separate sv file as many critical flags like damage to the user-controlled jet and keycode inputs are also used in the color-mapper module for our coloring algorithm. In total, our design incorporates 6 stages, the start screen, level 1 enemies, level 2 enemies, level 3 boss enemy, the gameover screen, as well as an intermediate stage between the gameover and start screen where the user can opt to restart the game, with the transition back to the start screen on the release of the ‘enter key’. Transitions between the 3 in-game stages/levels as well as to the gameover screen depends on damage to the user-controlled jet and hit flags on the enemy jets, which are triggered in collision logic further described below.

In order to support multiple backgrounds in our game, we incorporated an additional always_ff block in order to set a single background R, G, B color set that we can assign to the overall R, G, B output of the color mapper module in the coloring block, which significantly reduces complexity from the already complex coloring block. The background R, G, B is set based on the background flag and keycode inputs (for the hidden background), as well as the level flag determined by the state machine for the start and gameover screens. Another important note is that we set start and end flags of our game in this block along with the background since level is used as the determining parameter.

Another critical part of our game design is our health bar. One of the main goals of our game is for the user-controlled jet to have multiple ‘lives’. To support this, we included a damage flag for our user-controlled jet that serves more as a counter. Once the counter for damage on the user-controlled jet hits 3, the game ends. To support this, we utilized an always_ff block. The logic itself is quite simple and straightforward. On reset, touchfix (meaning we have made contact with the ‘fix’ special power module, and level == start screen, our damage is reset to 0. Else, when our draw user controlled jet flag is high (meaning we are supposed to draw the user-controlled jet at this particular location, we nest additional conditions within to check and see if we are also meant to draw the enemy jet or enemy ammo at this location. If so, it means that our user-controlled jet should take damage (increment the damage counter). One issue with this implementation that we will describe below is that without additional flags, our damage counter will increment significantly more than we anticipate as the ammo or enemy will be in contact with our user-controlled jet for a significantly longer period of time, leading to one shot kill situations on our user-controlled jet. To overcome this issue, we added additional flags to reset the position of our enemy ammo whenever they come into contact with the user-controlled jet. This in effect is the same as moving the enemy ammo back to its jet once it reaches the boundaries of the monitor/screen. To support decrements in the health counter, we simply used the damage flag/counter to determine which stages/phase of our healthbar to draw, as we have all 3 health stages on one single RAM palette.

We applied a similar version of damage/collision logic for our user-controlled jet onto the enemy jets. On reset and when start flag is low (meaning game has not started yet), we reset all of the hit flags of the enemy to low. We then checked for the condition when we are supposed to draw the user-fired ammo in each specific location, then further include statements nested within to check and see if we are also meant to draw the enemy at that particular location. If so, it means that we set the hit flag for that specific enemy to high. During this process, we also increment our score counter that helps determine what final score to display at the end of the game on the gameover screen. Again, we use the score flag/counter to determine the phase and which specific score to draw, with all possible scores on a single palette/image.

With our collision logic setting our hit flag for each individual enemy, we then worked on the explosion animation. For ease of tracking, we opted to have an explosion animation prepared for each individual enemy jet, instead of a single explosion animation toggled onto whichever enemy jet is destroyed. This also helps ensure no glitches occur if the user were to somehow eliminate two jets consecutively in a short time period (without the animation having sufficient time to run entirely). The animation is set to be the same size as the enemy jet itself, which means that our 'explosion_on_proc' logic can be reused. This 'on_proc' flag is then combined with background subtraction and the explode flag of the enemy (different from the hit flag) to give a final flag indicating an explosion on the enemy aircraft. To implement the animation of the explotion, we utilized an always_ff block that includes a counter and an additional flag to determine the phase/which stage of the animation to draw onto the screen. On reset, the counter is reset to 0. When the hit flag of the enemy is not yet triggered (at a low), the startexplosion flag and the explosion flag of the enemy remains low. When the hit flag of the enemy is high and the start explosion flag is still low (meaning the enemy is hit but we have yet to start the explosion animation), we set the explode flag to high and the animation flag to 0 (stage 1 of the explosion animation), as well as the start animation flag to high. The counter is reset to 0 here as well. The else condition of the block involves the counter incrementing. Once the counter hits 3 and the animation flag is still low, we reset the counter and set the animation flag to high (to start stage 2 of the explosion animation). The counter will then increment until we hit 3 again, but this time with the animation flag as high. This will trigger the explosion flag to drop from high to low, thus ending the explosion animation. This entire block of code is written for each individual jet in order to support multiple simultaneous explosions.

III. On-Chip Memory-Based VRAM & Graphics RAM

As our final project requires the color information of each background and sprite of each element, we utilized the Python program provided on the ECE 385 website to convert the JPEG/PNG of our sprites and background into ROM and color palette modules. The color palette module stores 12 bits of color information for each color used for the specific sprite consisting of 4 bits for red, 4 bits for green, and 4 bits for blue. On the other hand, the ROM module stores the address of the color palette for each pixel of the drawing sprite. To get the ROM address for the color palette, we first detect if the current DrawX and DrawY are in the range of the certain sprite. If they are in the range, then we compute the address with the generalized equation $DrawX - spriteX + (DrawY - spriteY) * Sprite\ width.$ Thus, the ROM module will return the palette index for the current drawing pixel of the specific sprite. Then, the index will be given to the color palette module to get the RGB values of the current pixel as the outputs, which will later be displayed on the screen.

IV. Drawing Algorithm

Regarding the drawing algorithm, for each of the sprites we have, we created flags for them to indicate whether they should be drawn or not, and we created a simple state machine as shown below in Figure 1.

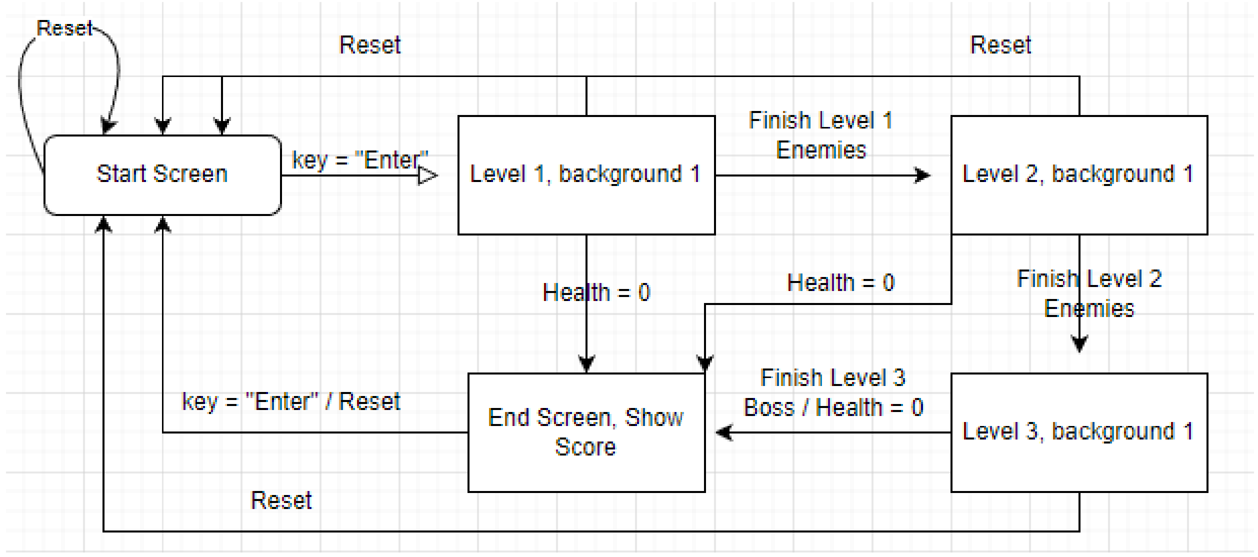


Figure 1. Final Project State Machine Diagram

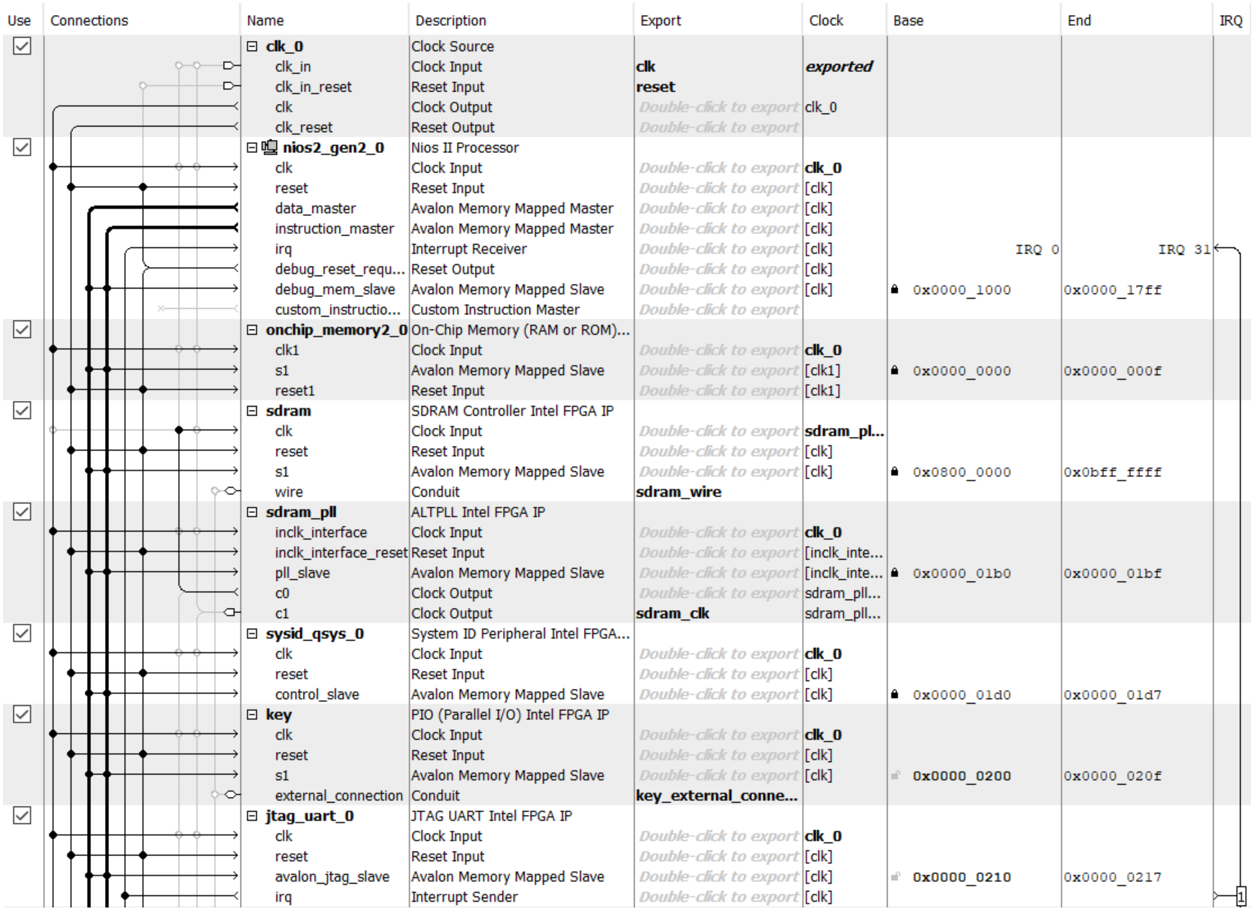
Starting from the start screen, the flags for all sprites are set to low to prevent the display of elements, such as jets and ammos, at the start screen. Later, if the "Enter" key is pressed, then we moved to level 1, and the background 1 will be displayed with the flags of all level 1 enemies and ammo set to 1. Once all the enemies are destroyed, then we move to level 2 and set all the flags of level 2 elements to high, and so on til we finished the final boss. Then, we moved to the end screen and display the score. In addition, during each state, whenever the reset button is pressed, we return to the start screen, and whenever the health is equal to 0 during any level, then we also move to the end screen to display the score. More importantly, we have listed a priority-ranking list for drawing the sprites so we can determine which sprites should be drawn on top of which sprites. For example, we set the health bar sprite as our top priority so elements such as bullets and enemies won't cover the health bar. To do so, we put the health bar flag in our top if statement to allow the system to draw its color first. Then, we simply put the second most important sprite into the next else if statement, and so on, with the least significant element in the last else if statement.

V. Platform Designer Modifications From Lab 6.2

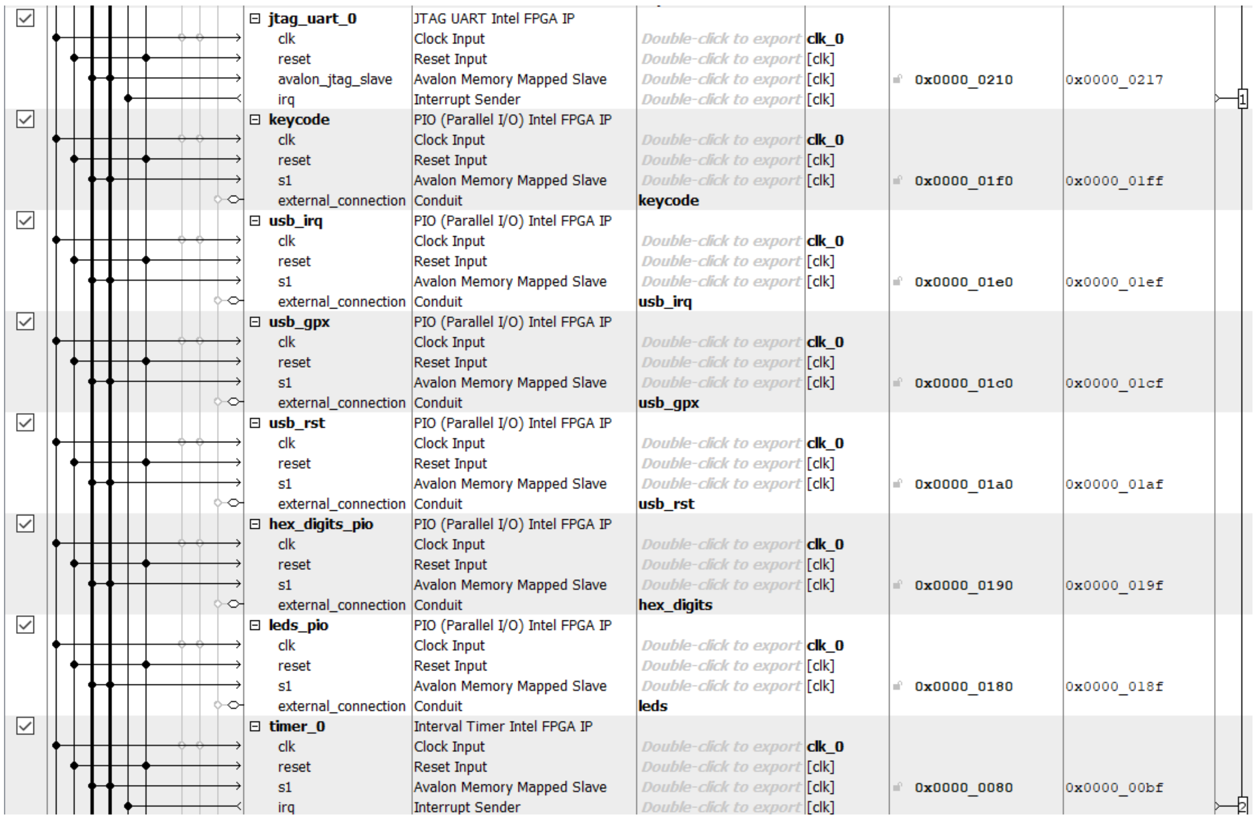
In terms of modifications to the platform designer, not much was needed to transition from Lab 6.2 to our final project. As mentioned above, the major change we had for our final project is the addition of two more keycodes in our platform designer so we can read a maximum of three keys from the keyboard simultaneously. This is needed for our game design, as our user-controlled jet neds to be able to move in all two-dimensional directions while firing ammos. In addition, this helps account for potential switching of the background when moving the jet or when firing.

VI. System Level Block Diagram

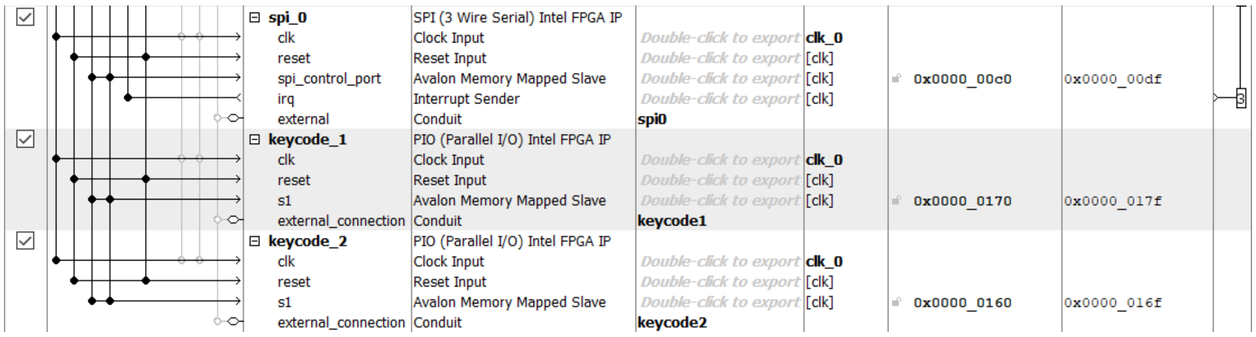
Final Project Design Platform Designer



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		**clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	*Double-click to export*	clk_0			
		clk_reset	Reset Output	*Double-click to export*				
✓		**nios2_gen2_0**	Nios II Processor					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		data_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]			
		instruction_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]			
		irq	Interrupt Receiver	*Double-click to export*	[clk]	IRQ 0	IRQ 31	
		debug_reset_requ...	Reset Output	*Double-click to export*	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_1000	0x0000_17ff	
		custom_instructio...	Custom Instruction Master	*Double-click to export*				
✓		**onchip_memory2_0**	On-Chip Memory (RAM or ROM)...					
		clk1	Clock Input	*Double-click to export*	**clk_0**			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	0x0000_0000	0x0000_000f	
		reset1	Reset Input	*Double-click to export*	[clk1]			
✓		**sdram**	SDRAM Controller Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**sdram_pl...**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_0000	0x0bff_ffff	
		wire	Conduit	**sdram_wire**				
✓		**sdram_pll**	ALTPLL Intel FPGA IP					
		inclk_interface	Clock Input	*Double-click to export*	**clk_0**			
		inclk_interface_reset	Reset Input	*Double-click to export*	[inclk_inte...			
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_inte...	0x0000_01b0	0x0000_01bf	
		c0	Clock Output	*Double-click to export*	sdram_pll...			
		c1	Clock Output	**sdram_clk**	sdram_pll...			
✓		**sysid_qsys_0**	System ID Peripheral Intel FPGA...					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0000_01d0	0x0000_01d7	
✓		**key**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	**0x0000_0200**	0x0000_020f	
		external_connection	Conduit	**key_external_conne...**				
✓		**jtag_uart_0**	JTAG UART Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	**0x0000_0210**	0x0000_0217	
		irq	Interrupt Sender	*Double-click to export*	[clk]			1

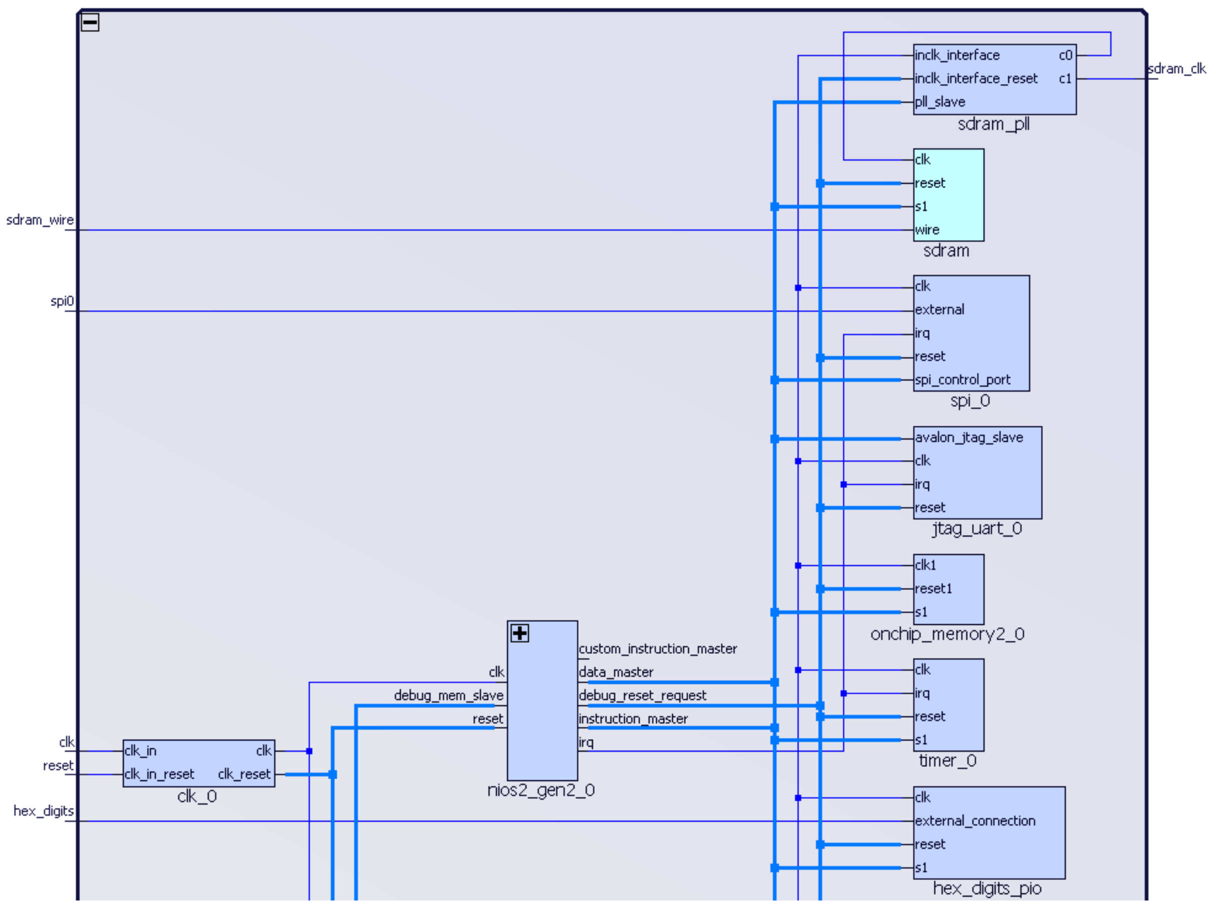


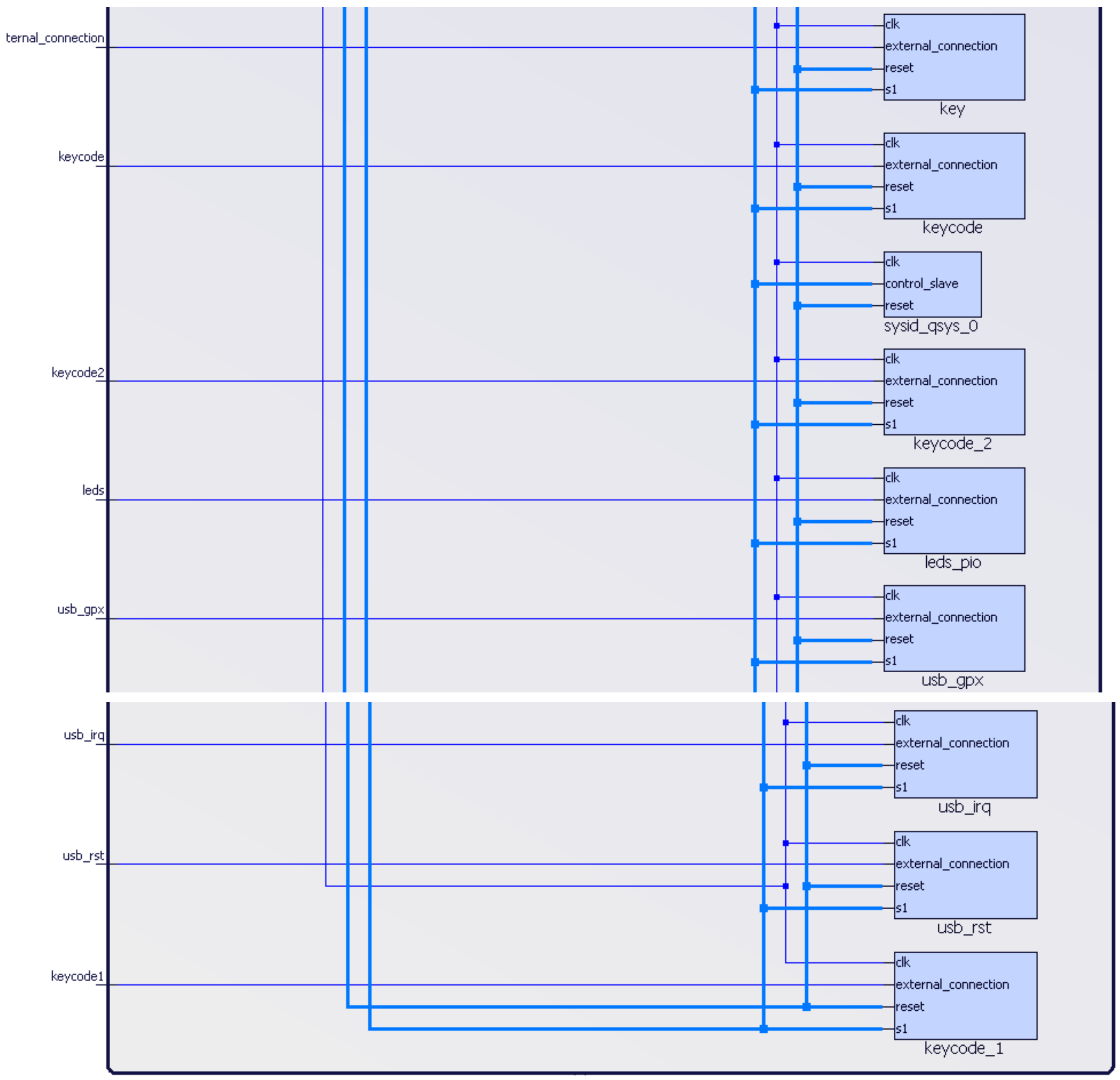
	Name	Description	Export	Clock	Base	End	IRQ
☑	⊟ **jtag_uart_0**	JTAG UART Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0210	0x0000_0217	
	irq	Interrupt Sender	Double-click to export	[clk]			1
☑	⊟ **keycode**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_01f0	0x0000_01ff	
	external_connection	Conduit	**keycode**				
☑	⊟ **usb_irq**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_01e0	0x0000_01ef	
	external_connection	Conduit	**usb_irq**				
☑	⊟ **usb_gpx**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_01c0	0x0000_01cf	
	external_connection	Conduit	**usb_gpx**				
☑	⊟ **usb_rst**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_01a0	0x0000_01af	
	external_connection	Conduit	**usb_rst**				
☑	⊟ **hex_digits_pio**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0190	0x0000_019f	
	external_connection	Conduit	**hex_digits**				
☑	⊟ **leds_pio**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0180	0x0000_018f	
	external_connection	Conduit	**leds**				
☑	⊟ **timer_0**	Interval Timer Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0080	0x0000_00bf	
	irq	Interrupt Sender	Double-click to export	[clk]			2



	Name	Description	Export	Clock	Base	End	IRQ
☑	⊟ **spi_0**	SPI (3 Wire Serial) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	spi_control_port	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_00c0	0x0000_00df	
	irq	Interrupt Sender	Double-click to export	[clk]			3
	external	Conduit	**spi0**				
☑	⊟ **keycode_1**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0170	0x0000_017f	
	external_connection	Conduit	**keycode1**				
☑	⊟ **keycode_2**	PIO (Parallel I/O) Intel FPGA IP					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0000_0160	0x0000_016f	
	external_connection	Conduit	**keycode2**				

Final Project Design System Level Block Diagram





clk_0 - System clock with 50 MHz from the FPGA that is used to clock and reset other modules.

nios2_gen2_0 - 32-bit CPU that allows the developers to code and debug in C for the processor and send out designated control signals to modules.

sdram - the sdram is the off-chip memory block that is used to store code and data. It is interfaced with the NIOS-II processor through the memory controller that helps manage data transfers. It effectively serves as the bridge between data transferred between the processor and the SDRAM.

sdram_pll - Provides clock signal to compensate for the propagation delay that may arise between the input and output clocks, making them synchronized.

sysid_qsys_0 - This is used to ensure the hardware and software are compatible and updated to avoid any error in the connections between them.

jtag_uart__0 - The JTAG UART provides a communication interface between the FPGA and other devices. In this lab, it allows the user to use “printf” on the console for debugging.

Onchip_memory2_0 - On-chip memory used to store R, G, B colors of graphic designs of images and backgrounds for our game. Stores color palettes as well as registers that correctly outputs the correct color palette register based on rom address input.

key - This is used as a PIO input to determine the signal for accumulate button and reset button.

keycode - Allows the processor to retrieve the specific code corresponding to the key that is pressed on the keyboard. Used in our design to control the movement of the jet (supports all two dimensional movements) as well as background switches and start control.

keycode_1, keycode_2 - Additional keycodes to support multiple actions at the same time (i.e. movement of the jet diagonally (combination of two keycode direction movements) as well as background switch)

usb_irq - This is used as the interrupt signal by the USB device to notify the computer about the key that is being pressed.

usb_gpx - The general purpose input/output peripheral is used to control the device functionality, including data transfer, etc.

usb_rst - The hardware reset pin resets the USB controller. Active low, it will reset the chips internal registers.

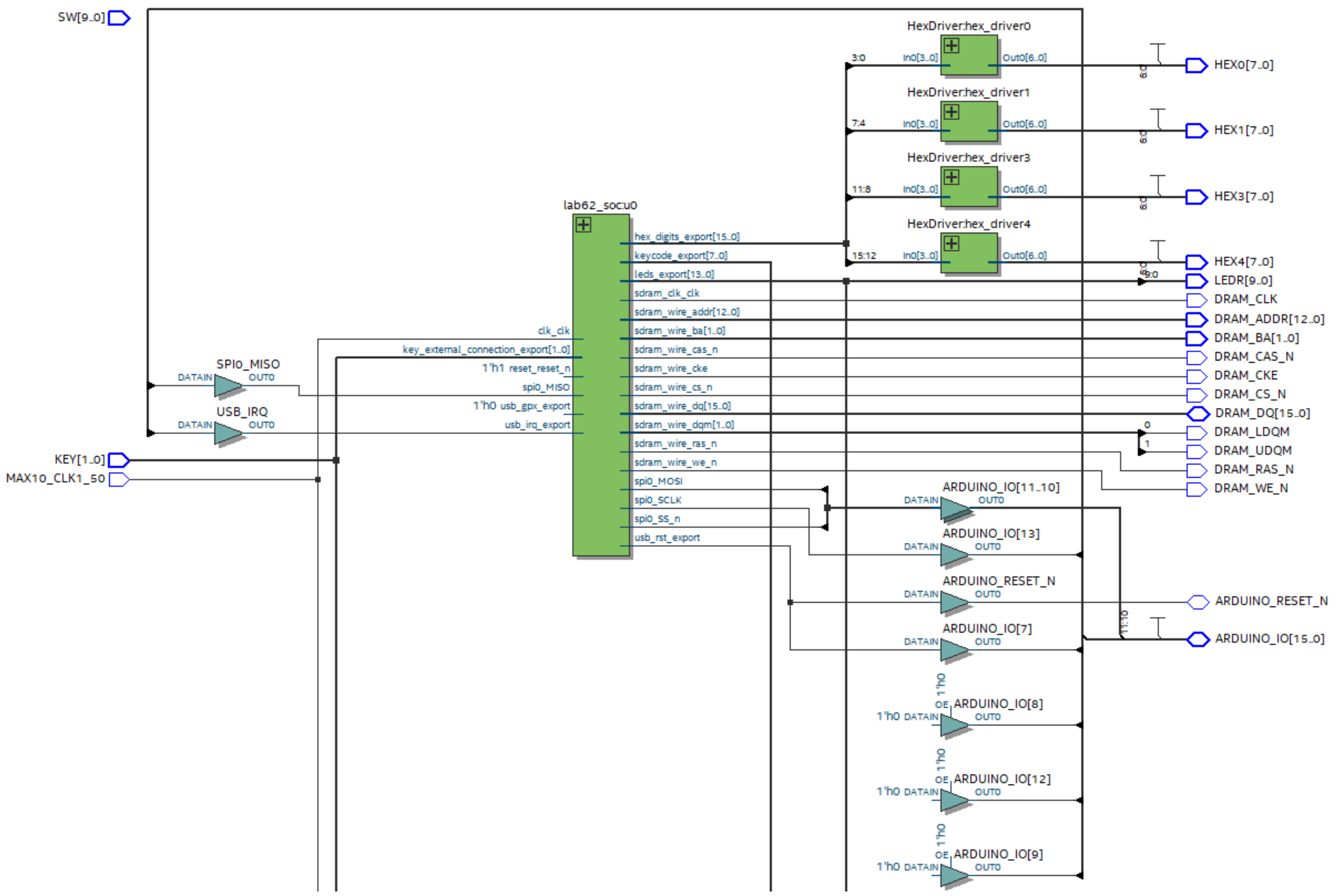
hex_digits_pio - This is used to display the keycode of the currently pressed key on the two hex displays.

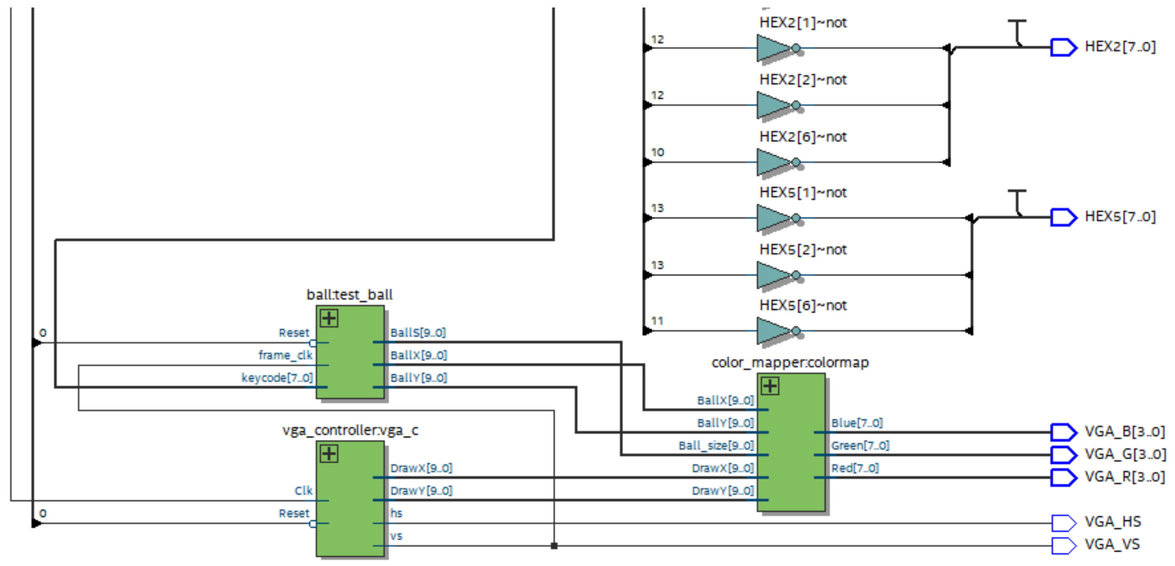
timer_0 - This is used to keep track of the time passed for Nios II and USB.

spi_0 - SPI_0 provides communication between a master device and one or more slave devices using four communication signals, SS (Slave Select), MOSI (Master Output Slave Input), Clock, and MISO (Master Input Slave Output). In this lab, Nios II is the master device while the MAX 3421E is the slave device. The SPI allows the master to request to read or write data from the USB device.

Top Level Block Diagram

Final Project Design Top Level Block Diagram





HEX2[1]~not
HEX2[2]~not
HEX2[6]~not
HEX5[1]~not
HEX5[2]~not
HEX5[6]~not
12
12
10
13
13
11
HEX2[7..0]
HEX5[7..0]
ball:test_ball
Reset
frame_clk
keycode[7..0]
BallS[9..0]
BallX[9..0]
BallY[9..0]
vga_controller:vga_c
Clk
Reset
DrawX[9..0]
DrawY[9..0]
hs
vs
color_mapper:colormap
BallX[9..0]
BallY[9..0]
Ball_size[9..0]
DrawX[9..0]
DrawY[9..0]
Blue[7..0]
Green[7..0]
Red[7..0]
0
0
VGA_B[3..0]
VGA_G[3..0]
VGA_R[3..0]
VGA_HS
VGA_VS

SV Modules Summary

Module: lab62.sv
Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW
Outputs: [9:0] LEDR, [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_HS, [3:0] VGA_R, VGA_G, VGA_B
Inouts: [15:0] DRAM_DQ, ARDUINO_IO, ARDUINO_RESET_N
Description: This module instantiates module lab7soc, which was generated by Platform Designer, and all the connections on FPGA, such as the clock, I/O, and SDRAM signals. Also, this module instantiates all other modules, such as the color mapper and enemies modules for the video game.
Purpose: This module is used as a top-level to connect the Nios II system to the rest of the hardware and instantiate all the objects in the video game.

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: This is the Hex driver that takes in 4-bit value In0 from the register to determine which segment of the 7-segment LED has to be light up and store the value into 7-bit output Out0 with 0 meaning turning light on and 1 meaning turning light off.
Purpose: This module is used to show the value of the key code pressed on the FPGA board by having 7-bit output indicating which segments of the LED have to be turned on.

Module: Graphics_rom.sv (This is a general file description for multiple sprites ROM modules)
Inputs: clock, [9:0] address
Outputs: [3:0] q
Description: This is a positive-edge triggered module that stores the color palette address for the current drawing pixel of the sprite.
Purpose: This module is used to return the color palette address of the current pixel.

Module: Graphics_palette.sv (This is a general file description for multiple sprites palette modules)
Inputs: [3:0] index
Outputs: [3:0] red, green, blue
Description: This module simply returns the RGB values stored in the specific index according to the input signal index.
Purpose: This module is used to get the RGB values of the current drawing pixel.

Post-Lab Questions

1. Fill in the table shown in IQT.30-32 with your design's resource and statistics.

Final Project Design's Resource and Statistics

LUTs	12,845 / 49,760 (26 %)
DSP	0
Memory (BRAM)	1,390,112 / 1,677,312 (83 %)
Flip-Flop	3285
Max Frequency	76.17 MHz
Static Power	96.83 mW
Dynamic Power	126.51 mW
Total Power	245.52 mW

Document any problems you encountered and your solutions to them.

In the final project building process, we encountered numerous problems. First of all, in the beginning, we were unable to display the graphics, such as the background and jet, on our screen. After debugging, we found out that the issue is due to the errors in how we called the RAM modules generated through the Python program, particularly how we incorporated the register module and the color palette module. Furthermore, we encountered a minor error in ammo glitching. What happens is that after a few shots, the ammo will glitch out and cycle through from the bottom of the screen to the top, which we later figured out that it has to do with the boundary issue of our screen. Thus, we lower the upper boundary to prevent the ammo from going over and cycling through again and again. Last but not least, the major issue we had is about collision detection. When we originally implemented our collision logic, we did not account for how we wanted to decrement health. Therefore, when the enemy ammo hit the user-controlled jet, instead of deducting one unit of health, it led to multiple deductions, which ended the game abruptly. To fix the bug, we added additional flags in the collision detection algorithm. Therefore, once the ammo hits the jet, we move away the ammo back to the enemy jet from which it was fired from to prevent it from causing more damage, while also resetting the ammo to be ready to fire again.

Conclusions

For our final project for the course, we designed a two-dimensional fighter jet video game on the intel FPGA. Our baseline for the project design was week two of lab 6, with the moving ball on screen with bouncing at the edges. For the purposes of our design, we got rid of the bouncing at the edges and made it so our ‘ball’ would only move at the press of a keycode(s). We applied the knowledge we gained through the course, particularly lab 5, to implement a simple state machine to control the levels of our game. We also incorporated graphic designs into our video game through the PNG/JPEG to RAM resource, utilizing appropriate scaling tricks in order to account for memory constraints.

a. Discuss functionality of your design. If parts of your design did not work, discuss what could be done to fix it.

While we encountered issues and bugs throughout the design and testing process, our final demo went smoothly with a design that was functional and complete. If we were to making improvements to our game, we could explore adding sound effects and a more advanced missile tracking algorithm/simple AI to make the game more challenging. There could also be more graphic design improvements made.

b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester? You can also specify what we did right, so it does not get changed.

No, we did not find anything particularly ambiguous or incorrect in the provided materials for our final project.