ECE 385

Fall 2022

Final Project

Touhou on FPGA: A Bullet Hell Arcade Game Implemented Using the NIOS II

Zayaan Ali, Ali Albaghdadi

TA: Nick Lu

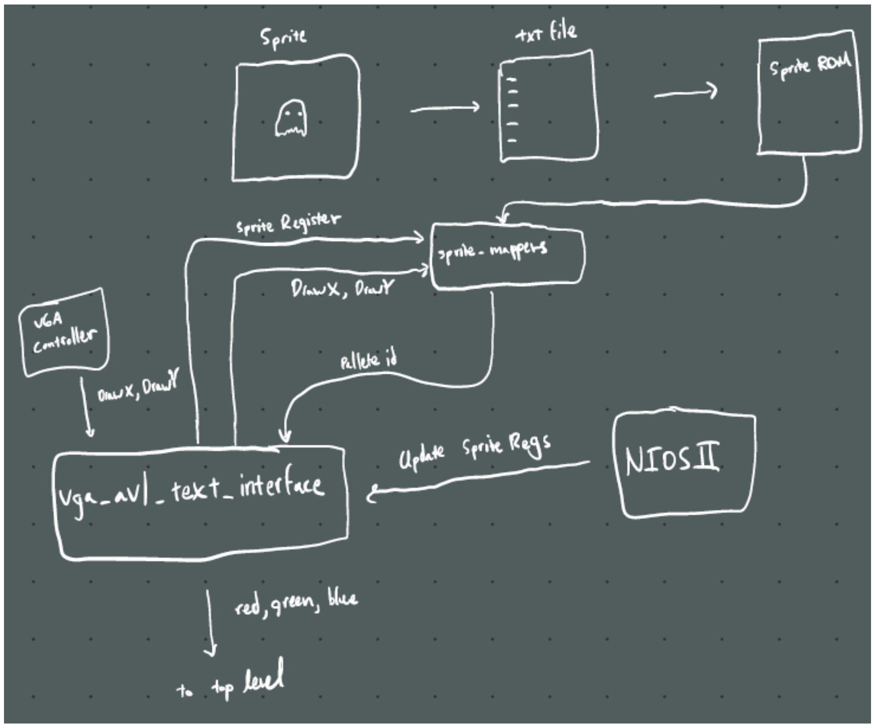
12/12/22

Introduction

For our final project, we created an arcade space-themed shooter taking inspiration from the Japanese 90s game Touhou Project. The game consists of controlling a playable character to avoid waves of enemies, each of which can shoot bullets at the player. The player loses a life each time he hits a projectile, and when three lives are lost, the game ends. The player can avoid enemies and projectiles by using the arrow keys to move the character around the playable space. A laser gun can be used to shoot enemies, and upon contact with its beam they will disappear. If needed, the player can activate a spell card up to 3 times per life to immediately despawn all visible enemies and projectiles. The player score, as well as the life and spell counters can be found to the right of the playable area and are updated each time the player kills an enemy, loses a life, and uses a spell card respectively.

Documentation and Design

Overview of Hardware Design



The high-level structure of the hardware consists of several On-Chip-Memory blocks storing important data for multiple different sprites, which is then accessed by a number of color mappers depending on the current pixel being drawn. The color mappers feed RGB data for each pixel to the vga_avl_text_interface module, which prioritizes which sprite should be drawn, and outputs the RGB signal to the top-level module. The software side consists of a NIOS II/e processor implemented on the same die and connected to the OCM blocks via the Avalon bus to allow it to read and write to them freely.

Storing the Sprites

To store each of the 17 sprites used in the game, we used the provided sprite_to_txt tool to generate a txt file with palettized data for each pixel in the sprite. Since our game only used 15 distinct colors, we were able to make the color encoding just 4 bits wide. Each of the sprite txt files were loaded into 17 different ROMs, stored on the On-Chip-Memory. All of the sprites used in our program were 32 pixels wide by 48 pixels high, which in turn resulted in a size of about 768 bytes to store each sprite (4 bits per pixel). This meant that the total space used to store all 17 of the sprites used in our game amounted to 13.056 KB, well within the size of the On-Chip-Memory on the FPGA.

Controlling Sprites

The control of the sprites is done entirely in software, through the use of the NIOS II. This was possible through the use of 60 sprite registers, each 20 bits wide. The mapping of each register can be found below:

Register Number	Use
0	Player Register
1-4	Enemy Winged Eye Registers
5-8	Enemy Fairy Registers
9-12	Enemy Ghost Registers
13-50	Bullet Registers

51-55	Unused
56	Scoreboard 4 digits (each digit as 4 bits)
57	[7:4] Life Counter [3:0] Spell Counter
58	[1:0] Player Sprite Direction
59	VSync signal (from VGA_controller.sv)

The mapping for each individual sprite register can be found below:

Enable	Sprite x-position	Sprite y-position
[19]	[18:9]	[8:0]

The enable bit controls whether the sprite is rendered or not – if the enable bit is 0 the sprite will not be drawn at all and the background color will be picked instead. If the enable is 1, the sprite x- and y-position bits dictate where on the screen the sprite will be rendered. Each of these values is directly set in software by the NIOS II, resulting in sprite rendering and position being fully controllable by software. This structure allowed us to greatly simplify the process of developing the game. In addition, it made compilation far faster on our modest hardware and would make expanding the scope of the game significantly easier than if it had been in hardware.

Accessing Sprite ROMs

To be able to access the color data from each sprite ROM, we created individual color mappers for each sprite. Each color mapper takes in the current X and Y positions being drawn by the VGA controller, as well as the sprite position and enable (from the sprite register). This information is processed, and the ROM is accessed to give the palettized pixel data for pixel currently being drawn. The logic used to access the ROM can be found below:

$$Xsprite = DrawX - XCoord, \qquad Ysprite = DrawY - YCoord$$

$$pixel\_access = (32 * Ysprite) + Xsprite$$

Xcoord and Ycoord both refer to the position of top left corner of the sprite. Xsprite and Ysprite each refer to the positon of the pixel being drawn inside the sprite itself. Then, pixel_access is used to access the ROM and get the palettized data for the exact pixel being drawn. Each sprite is hard-bound to a register in OCM and has its own color mapper (which itself instantiates its own ROMs to eliminate the need to make a memory access state machine), allowing for the easy layer prioritization of sprites during the render process. In order for the player sprite to change when moving direction, the player color mapper has a slightly modified structure. In addition to position and enable data, it also takes the current direction as a 2-bit input, and therefore changes the ROM picked depending on those 2 bits. The NIOS II sets those two bits in Register 58 based on the direction that the character is moving: $2'b10$ if the character was moving left, $2'b01$ if right, and $2'b00$ if the character's x-velocity is 0.

Rendering Everything

All of the render logic takes place in vga_text_avl_interface module. This module instantiates each of sprite color mappers, which provide palettized color data for each pixel to be rendered on the screen. Through additional logic, we were able to limit certain sprites to only render on the playable area of the screen, as well as implement priority for different sprites over others and different parts of the screen over others.

Software Functions

The software project is contained within "software/usb-kb," and the main bulk of the game engine code written by us resides within "main.c" and "main_helper.h". The project also makes use of the USB keyboard reader module given to us in Lab 6.2 of the class, the critical files of which are located in the "usb_kb" folder.

Main_helper.h defines a few critical helpers for main.c. For starters, it implements a struct to box the 60 registers of the Avalon OCM block into their respective uses so they can be accessed by name rather than by magic numbers, and it also instantiates this struct at the position of the

Avalon block on the bus. It also implements three useful helper methods: playerVsBullet, enemyVsLaser, and bulletRangefinder.

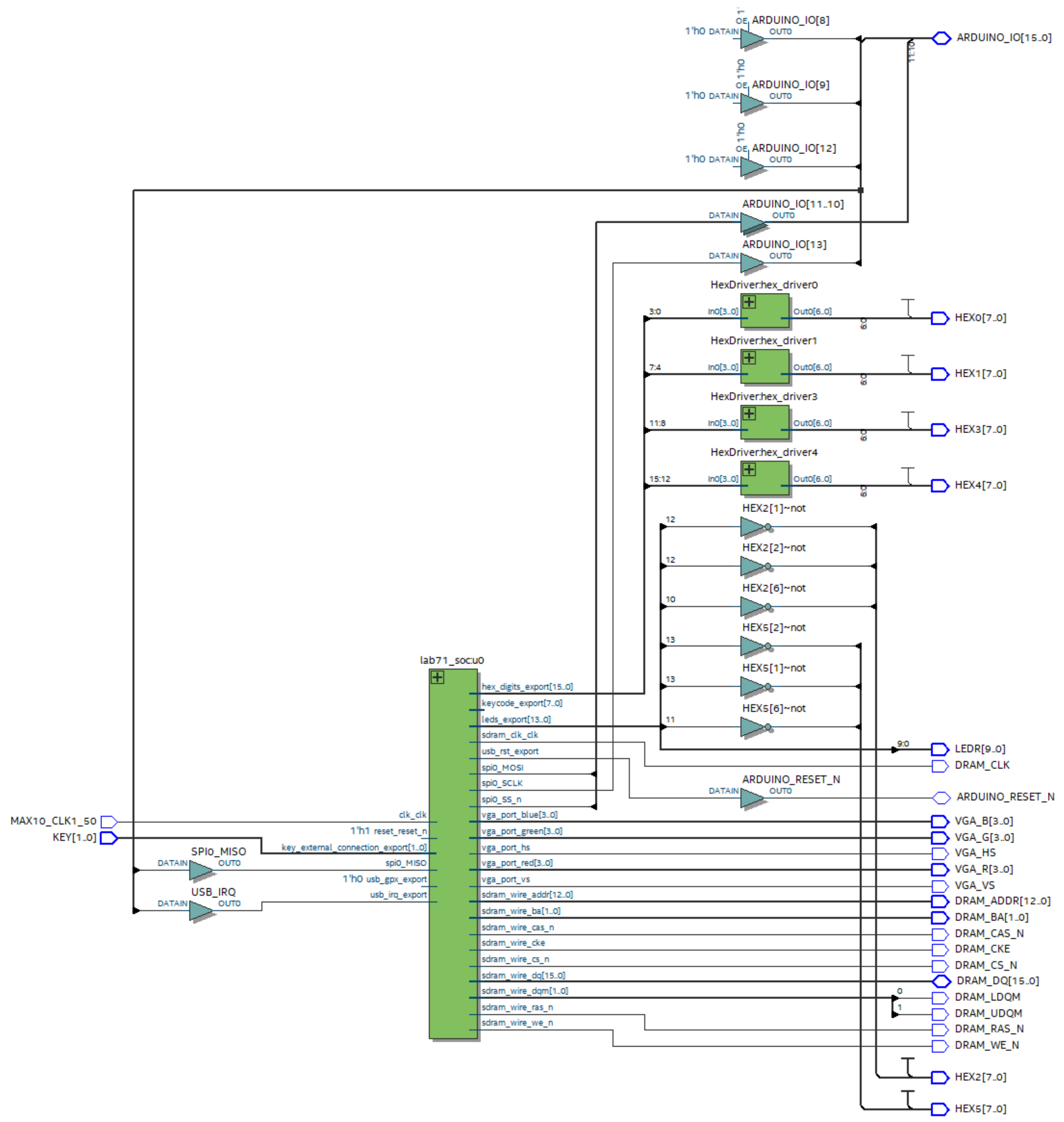
playerVsBullet calculates whether a player has made contact with a bullet using the positions of the top-left corners of their respective sprites. This method was created because the sprite of the bullet is mostly empty space, and we wanted to make sure the player box actually collided with the white bullet while also keeping the clutter of the math out of the body of main.c. enemyVsLaser serves a similar purpose, but instead to determine whether an enemy sprite has made contact with the thin laser shot by the player. bulletRangefinder was originally intended to shoot bullets with x- and y-velocities aimed at the center of the player sprite, but could not be made to work accurately without using floats, which would be extremely slow due to the NIOS II/e's lack of floating-point acceleration. We opted to simply shoot the bullets downwards with a constant velocity, and bulletRangefinder returned the x- and y-velocities in a format that was easy to use with the existing bullet registers' data layout.

The game logic timing operates in a very simple manner: to ensure that everything is updated in line with the framerate of the screen and while nothing is currently being drawn on the screen, the processor repeatedly polls Register 59 (which is connected to the VGA controller's VSync signal) until it becomes 0. This way, the processor ideally does all its work in the frame blanking time.

The random number generator is seeded by creating a stack variable "time_t t" and not initializing it, causing it to be set to a random value. Srand is seeded using this random value. The player is made to start in the bottom middle of the playable area, the player is given 2 lives and three spellcards, and all enemies and bullets from the previous game are cleared. The USB keyboard is polled with the game logic to reduce undue stress on the CPU, and all necessary flags, like player sprite direction and x- and y-velocity, shooting laser status, and spellcarding status are set by looping over the 6-key array of keycodes returned by it. A short segment of code after that latches the X key so that holding it down only uses one spellcard per keypress instead of one per frame. The player sprite's bounds checking is done in software shortly before writing it to the respective register.

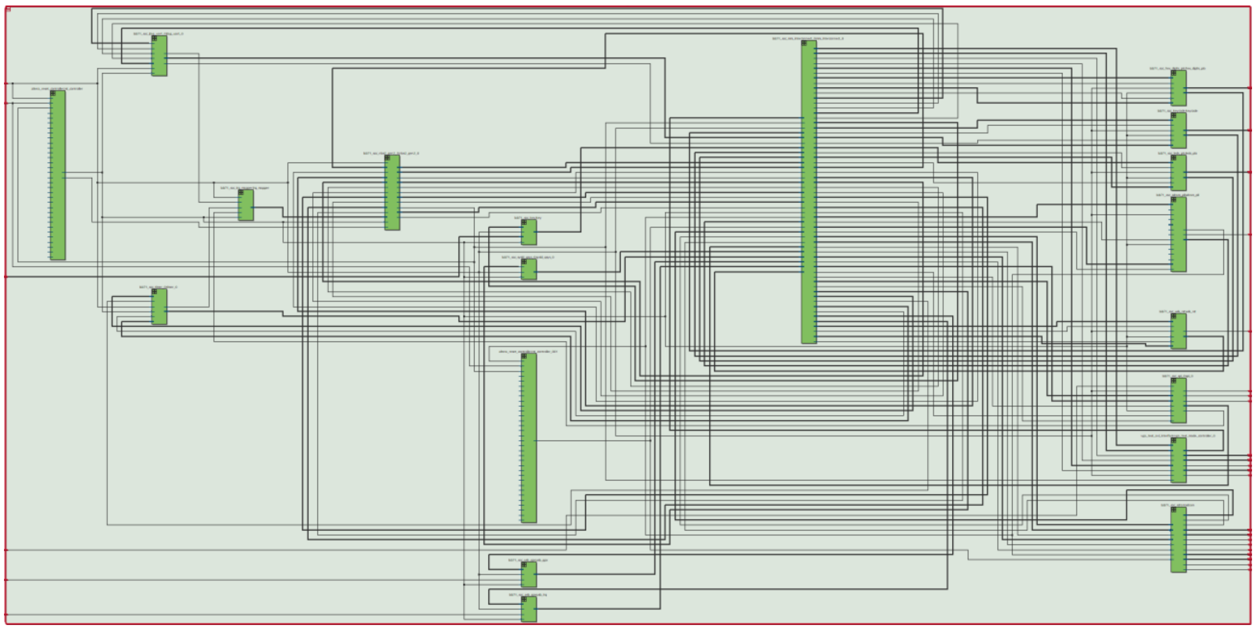
Then, all the enemy sprites are looped over and checked to be in contact with where the laser should be and if the player is shooting, and a random number is generated to see if this enemy should shoot a bullet as well. Random numbers are also generated to spawn new enemies if the max number of 12 enemies is not already on the screen, as well as their spawn position. Finally, all bullets are looped over and their positions updated using bulletRangefinder, the player is checked for contact with a bullet, the screen is cleared if the X key is currently being pressed, and score, lives and spellcards are retallied and written to the registers. The processor then polls the USB and waits for another VSync event.

Top level

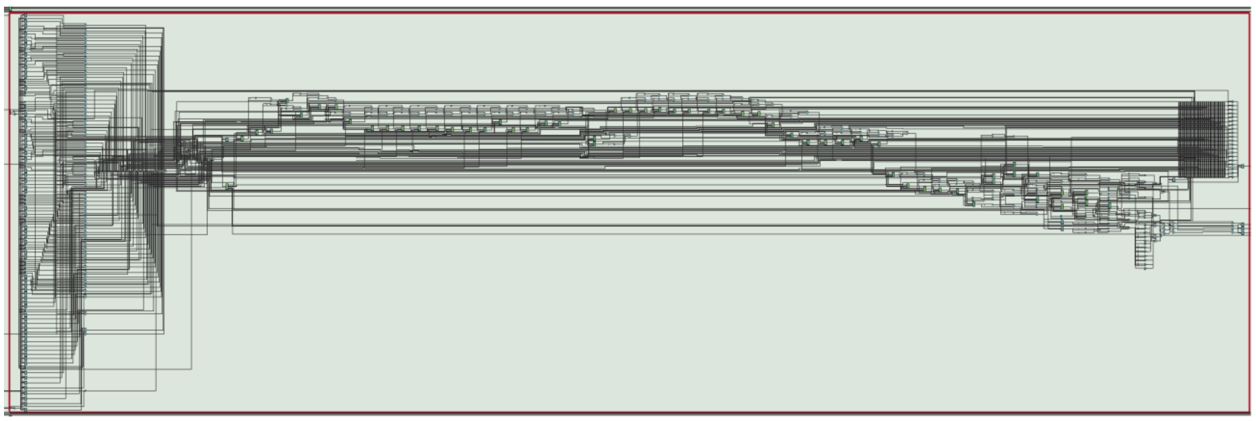


ARDUINO_IO[8]
ARDUINO_IO[9]
ARDUINO_IO[12]
ARDUINO_IO[11..10]
ARDUINO_IO[13]
ARDUINO_IO[15..0]
HexDriver:hex_driver0
HexDriver:hex_driver1
HexDriver:hex_driver3
HexDriver:hex_driver4
HEX0[7..0]
HEX1[7..0]
HEX3[7..0]
HEX4[7..0]
HEX2[1]~not
HEX2[2]~not
HEX2[6]~not
HEX5[2]~not
HEX5[1]~not
HEX5[6]~not
lab71_soc:u0
hex_digits_export[15..0]
keycode_export[7..0]
leds_export[13..0]
sdram_clk_clk
usb_rst_export
spi0_MOSI
spi0_SCLK
spi0_SS_n
vga_port_blue[3..0]
vga_port_green[3..0]
vga_port_hs
vga_port_red[3..0]
vga_port_vs
sdram_wire_addr[12..0]
sdram_wire_ba[1..0]
sdram_wire_cas_n
sdram_wire_cke
sdram_wire_cs_n
sdram_wire_dq[15..0]
sdram_wire_dqm[1..0]
sdram_wire_ras_n
sdram_wire_we_n
clk_clk
reset_reset_n
key_external_connection_export[1..0]
spi0_MISO
usb_gpx_export
usb_irq_export
MAX10_CLK1_50
KEY[1..0]
SPI0_MISO
USB_IRQ
ARDUINO_RESET_N
LEDR[9..0]
DRAM_CLK
VGA_B[3..0]
VGA_G[3..0]
VGA_HS
VGA_R[3..0]
VGA_VS
DRAM_ADDR[12..0]
DRAM_BA[1..0]
DRAM_CAS_N
DRAM_CKE
DRAM_CS_N
DRAM_DQ[15..0]
DRAM_LDQM
DRAM_UDQM
DRAM_RAS_N
DRAM_WE_N
HEX2[7..0]
HEX5[7..0]

Soc



vga_text_avl_interface



Module Descriptions

clk_0:

Clock source that feeds into the other hardware components.

nios2_gen2:

32-bit modified Harvard RISC embedded processor capable of completing simple operations written in C.

sdram:

512MBit external memory used to store the NIOS II processor.

sdram_pll:

creates two clock signals, both at 50MHz, with one delayed by 1ns in relation to the other. This is due to the SDRAM requiring a delay such that the memory is accessed during the correct window for a read/write.

Sysid_qsys:

Gives the hardware a unique ID that changes every time the hardware changes. Prevents trying to load new code on old hardware or old code on new hardware by ensuring everything is recompiled and refreshed.

Jtag_uart_0:

Allows communication between the console on the PC and the NIOS II through a serial bitstream.

keycode:

Parallel IO module used to read keys from the USB-to-SPI chip MAX3421E.

usb_irq:

Parallel IO that is used to interrupt the CPU as needed to read keycodes from the keyboard.

usb_gpx:

A General-Purpose Multiplexed Output signal from the MAX3421E to indicate the device's status.

Spi_0:

Serial peripheral interface which allows the FPGA to send and receive data from peripherals. In this lab, the peripheral device connected was the MAX3421E controller. The SPI transmits SS, MISO, MOSI, and SCK signals to the controller allowing it to receive and output keyboard input to the CPU.

VGA_text_mode_controller:

Custom IP that takes sprite data from the ROMs as well as DrawX and DrawY from the VGA controller, decides which sprite to render, and outputs the RGB values to be rendered on the screen.

Description of All Modules

Lab7.sv

Inputs: MAX10_CLK1_50, [1:0] KEY, [9:0] SW, [9:0] LEDR

Outputs: DRAM_CLK, DRAM_CKE, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, VGA_HS, VGA_VS, [9:0] LEDR, [12:0] DRAM_ADDR, [1:0] DRAM_BA

[3:0] VGA_R, VGA_B, VGA_G

[7:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5

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

Description: Top level module for the final project, instantiates lab71_soc.

Purpose: Takes in input from switches/buttons, instantiates SoC, and displays output to HEX display.

Vga_text_avl_interface.sv

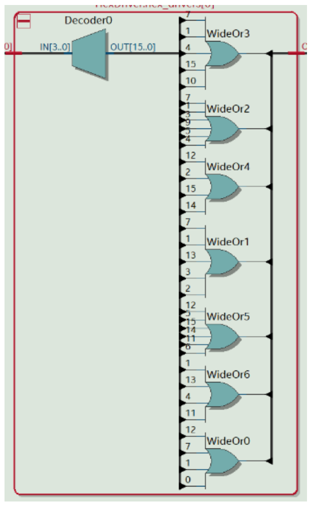
Input: CLK, RESET, [3:0] AVL_BYTE_EN, [11:0] AVL_ADDR, [31:0] AVL_WRITEDATA

Output: [31:0] AVL_READDATA, [3:0] red, [3:0] green, [3:0] blue, hs, vs

Description: Instantiates modules for each sprite that can be displayed on the screen, chooses the location that each sprite can be rendered in, as well as chooses which RGB value to printed depending on what pixel is currently being printed.

Purpose: Allows sprite registers to be written from software, as well as implements a render order in which each sprite is rendered.

HexDriver.sv



Inputs: [3:0] In0

Outputs: [6:0] Out0

Description: This unit has hardcoded values that convert 4-bit binary input into hexadecimal outputs to drive a 7-segment decimal display to show a single hexadecimal digit.

Purpose: Map the content of registers in hex values so they can be displayed on the devboard. Unused in the final project.

VGA_controller.sv

Inputs: CLK, Reset

Outputs: hs, vs, pixel_clk, blank, sync, [9:0] drawX, [9:0] drawY

Description: Outputs DrawX and DrawY signals to specify the current pixel to be drawn to the screen. Handles horizontal and vertical sync, as well as outputs signal to specify if is in the blanking interval.

Purpose: Specify the current pixel being drawn

sprite_mappers.sv

Inputs: [19:0] sprite_info, Clk, [9:0] DrawX, [9:0] DrawY

Outputs: [3:0] palette_out

Description: This file contains the color mappers for each individual sprite. Instantiates and accesses the sprite ROMs to find the color data of each pixel being drawn. Outputs the palettized pixel data depending on which pixel is current being drawn

Purpose: Find what color should be drawn for a given pixel for each sprite, so that can be processed by vga_text_avl_interface module

ROMs.sv

Inputs: [3:0] data_in, [10:0] write_address, [10:0] read_address, we, Clk

Outputs: [3:0] data_out

Description: this file includes the ROMs for each of the sprite types. Each ROM is preloaded with sprite encoding converted using the PNG to txt helper tool.

Purpose: Stores palettized pixel data that is used to render each sprite

Lab71_soc

Description: implements the hardware components added in the Platform Designer

Purpose: Creates the SoC and its components to allow the user to do low-performance tasks in C, such as interacting with VGA and USB peripherals.

Lab7.sdc

Description: Sets time delays for signals accessing DRAM as well as false paths for various other signals.

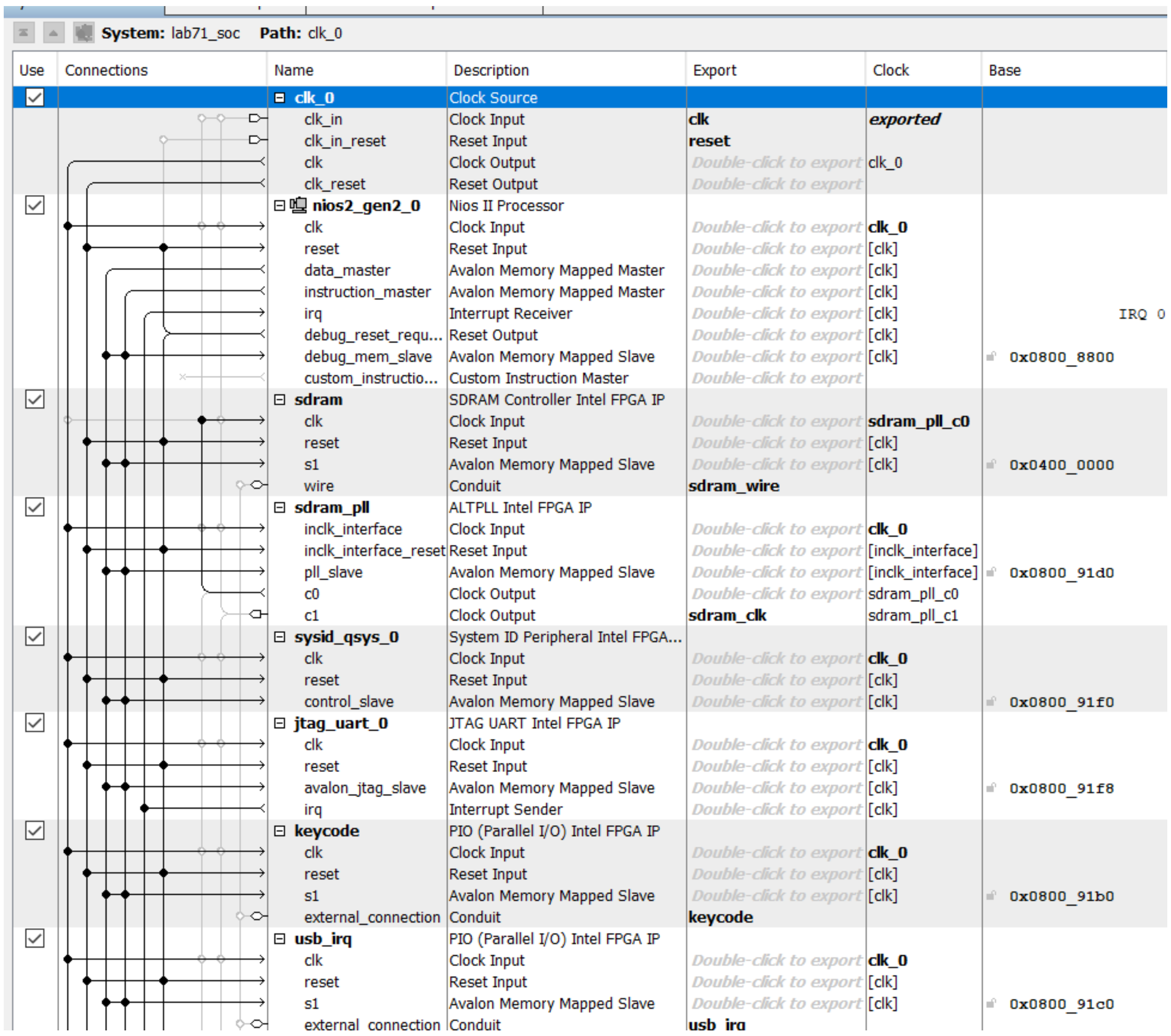
Purpose: Ensures setup and hold times are not violated and the design is not metastable.

software/usb-kb/main.c

Description: Contains the C code to be loaded onto the NIOS II CPU.

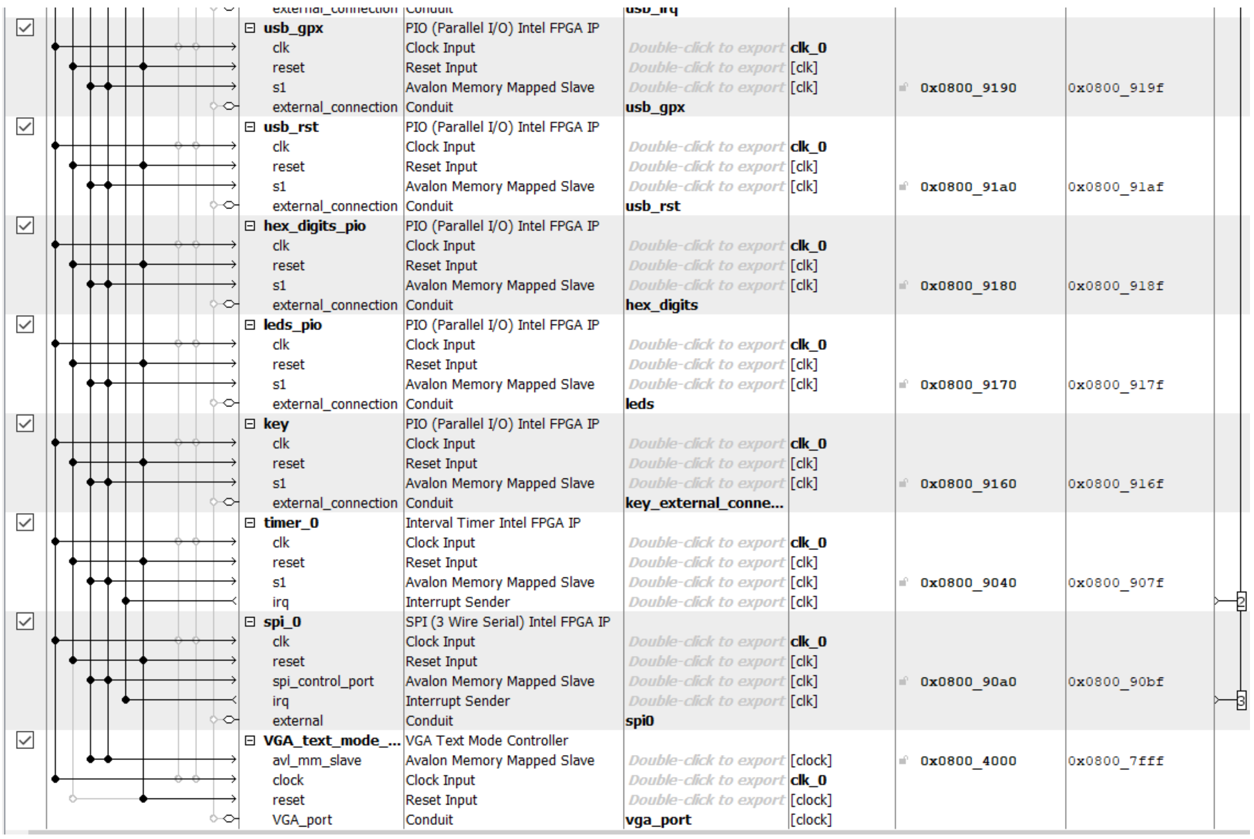
Purpose: Runs the game logic and communicates with the OCM drawer component on the Avalon bus.

System Level Block Diagram



System: lab71_soc **Path:** clk_0

Use	Connections	Name	Description	Export	Clock	Base
✓		**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
		debug_reset_requ...	Reset Output	*Double-click to export*	[clk]	
		debug_mem_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0800_8800
		custom_instructio...	Custom Instruction Master	*Double-click to export*		
✓		**sdram**	SDRAM Controller Intel FPGA IP			
		clk	Clock Input	*Double-click to export*	**sdram_pll_c0**	
		reset	Reset Input	*Double-click to export*	[clk]	
		s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0400_0000
		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_interface]	
		pll_slave	Avalon Memory Mapped Slave	*Double-click to export*	[inclk_interface]	0x0800_91d0
		c0	Clock Output	*Double-click to export*	sdram_pll_c0	
		c1	Clock Output	**sdram_clk**	sdram_pll_c1	
✓		**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]	0x0800_91f0
✓		**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]	0x0800_91f8
		irq	Interrupt Sender	*Double-click to export*	[clk]	
✓		**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]	0x0800_91b0
		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]	0x0800_91c0
		external_connection	Conduit	**usb_irq**		



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]	0x0800_9190	0x0800_919f	
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]	0x0800_91a0	0x0800_91af	
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]	0x0800_9180	0x0800_918f	
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]	0x0800_9170	0x0800_917f	
external_connection	Conduit	leds				
⊟ 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]	0x0800_9160	0x0800_916f	
external_connection	Conduit	key_external_conne...				
⊟ 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]	0x0800_9040	0x0800_907f	
irq	Interrupt Sender	Double-click to export	[clk]			2
⊟ 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]	0x0800_90a0	0x0800_90bf	
irq	Interrupt Sender	Double-click to export	[clk]			3
external	Conduit	spi0				
⊟ VGA_text_mode_...	VGA Text Mode Controller					
avl_mm_slave	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0800_4000	0x0800_7fff	
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
VGA_port	Conduit	vga_port	[clock]			

Design Statistics

LUT	9263
DSP	0
Memory (BRAM)	1087488
Flip-Flop	3762
Frequency	74.63 MHz
Static Power	97.20 mW
Dynamic Power	144.03 mW
Total Power	323.60 mW

Conclusion

We were able to get all the proposed features of the final project working, however we did have some issues in our development process. Initially, we had planned to use the SDRAM to store a frame buffer, which we would use to render to the screen. We were able to get the frame buffer working in a limited capacity, however the single port nature of it, as well as other issues with memory storage resulted in us backtracking, and we realized that it wasn't even necessary to use the SDRAM for the project. One of the main takeaways I had from the project was that planning is super important and that a poorly elaborated initial plan can easily cause a large setback. Since we didn't have a clear idea of what we wanted to do from the outset, once we had a clear structure for what needed to be done, we found that we had just over a week to complete the project with minimal code written. This ended up causing a decent amount of stress and resulting in the cutting of some other features that had wanted to implement. However, I think the implementation that we went with gave us the ability to quickly scale up on additional features, and given more time we would have been able to add more of the features that we had wanted to.