Lurking in the Dark Game

ECE 385 | Spring 2020 | Final Project

Sahil Patel and Zach Deardorff

Section ABB (Friday 11am - 1:50pm)
TA: Yucheng Liang

INTRODUCTION

For our final project we emulated a game that was created during a game jam in 2019 called *Lurking in the Dark*. We chose to do a game because it was the most fun idea we came up with, and we chose the game Lurking in the Dark because it struck us as such an interesting game idea when we first came across it. For our implementation of the game we created a software/hardware interface so that most of the game logic could be done in software, and the graphics would be handled by the hardware. The game logic in from the software determines what will be printed to the screen, and the hardware is what actually does the printing and acceleration of the graphics. We made use of the system on chip design of the NIOS II to run our software and the USB interface we created in lab 8 to handle user inputs for our game.

GAME DESCRIPTION

The idea of Lurking in the Dark is that you have a candle that you can turn on and off to see only one tile ahead of you. With only being able to see only one tile ahead of you, you must get to the end of the room avoiding spikes and monsters. The difficulty appears due to the fact that the monsters follow you when you have your light on and you can only kill the monsters by leading them over spikes. This is how the game transforms into a puzzle game where the player must create a mental map of the layout of the floor so that they can create ways to kill the monsters by knowing how to move without your light on so that when you turn the light back on the monsters will take a path through spikes to get to you. In our implementation we also included a high score table for how fast people completed our five levels. The levels we created were also a mixture of ones from the original game and some we came up with ourselves.

SYSTEM DESCRIPTION

Our system revolves around a central NIOS processor. The processor is connected to three types of memory: OCM for the spritesheet, SRAM for the frame buffers, and SDRAM for instructions and the C stack. The NIOS has two main peripherals: the EZ-OTG chip which allows for USB-keyboard input and a graphics accelerator which handles software requests to write to the next-frame buffer (NFB).

The graphics accelerator operates through an Avalon bus interface with 7 registers. The

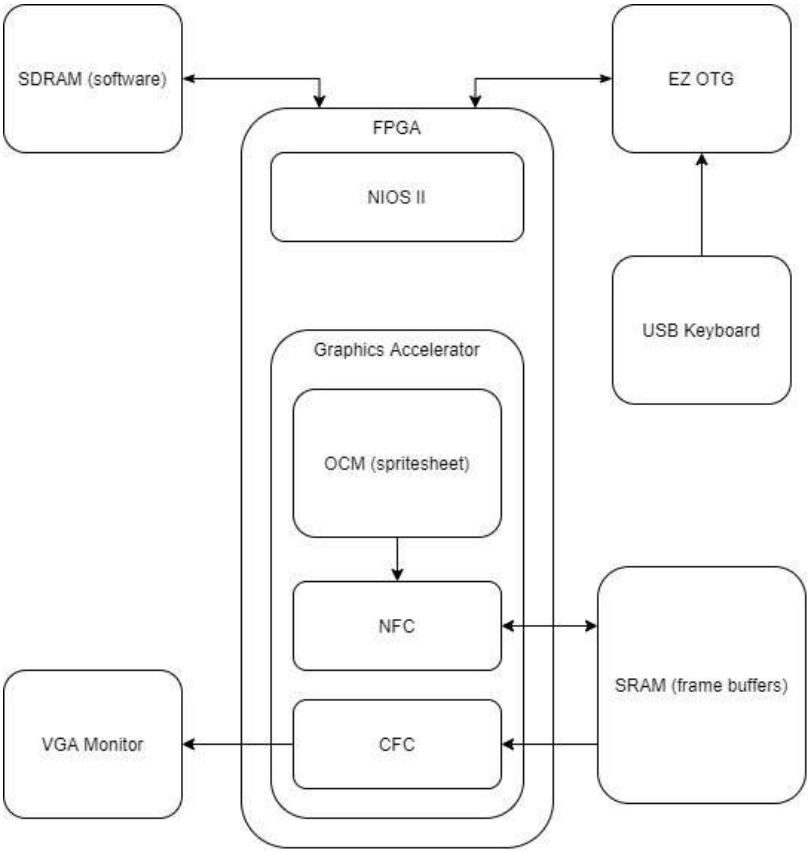
writeable registers (R0-R5) allow the software to ask the accelerator to start a task. Register 6 is read-only and is where the accelerator can tell the software that the requested operation has completed.

The accelerator has two subcomponents: the current-frame controller (CFC) and the next-frame controller (NFC). The CFC reads the upcoming row of the current-frame buffer (CFB) into a row buffer in OCM at the start of horizontal blanking. Then, during the next row period, it outputs the corresponding color to the VGA interface. The NFC, on the other hand, handles software requests. It has two possible functions: clearing and drawing. Drawing requests require the coordinates on the sprite sheet and screen to draw while clearing requests require no other input.

Since both the CFC and NFC interface with the SRAM, the graphics accelerator must switch between each one's operations. To do this, both the CFC and NFC have enable inputs and step_done outputs. They each have control of the SRAM while enabled and will continue to be enabled until they raise step_done. Since the CFC must have control over the SRAM during horizontal blanking, it only raises step_done for a short window after its read operation is done. This prevents the NFC's draw request from delaying the CFC's read.

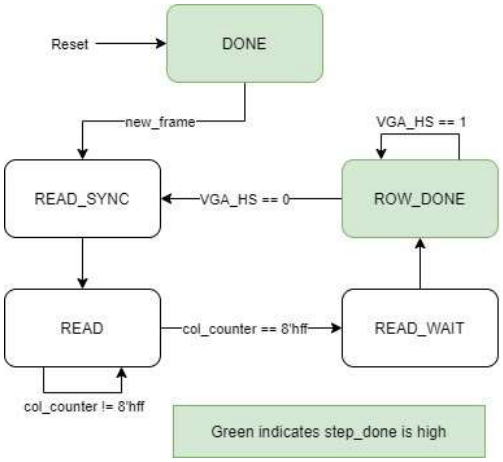
We also wanted to avoid copying data from the NFC to the CFC in order to start displaying it on screen. To solve this problem we used a signal called even_frame, so named because it initially swapped values every frame. This signal acted as the 2nd MSB of the SRAM address of the CFB (while ~even_frame was the 2nd MSB of the NFB address). In the current iteration, even_frame doesn't change via the frame clock but instead when a clear is requested. Essentially, a clear request swaps the frame buffers and clears the new NFB so that it is ready to be drawn on from scratch.

BLOCK DIAGRAM

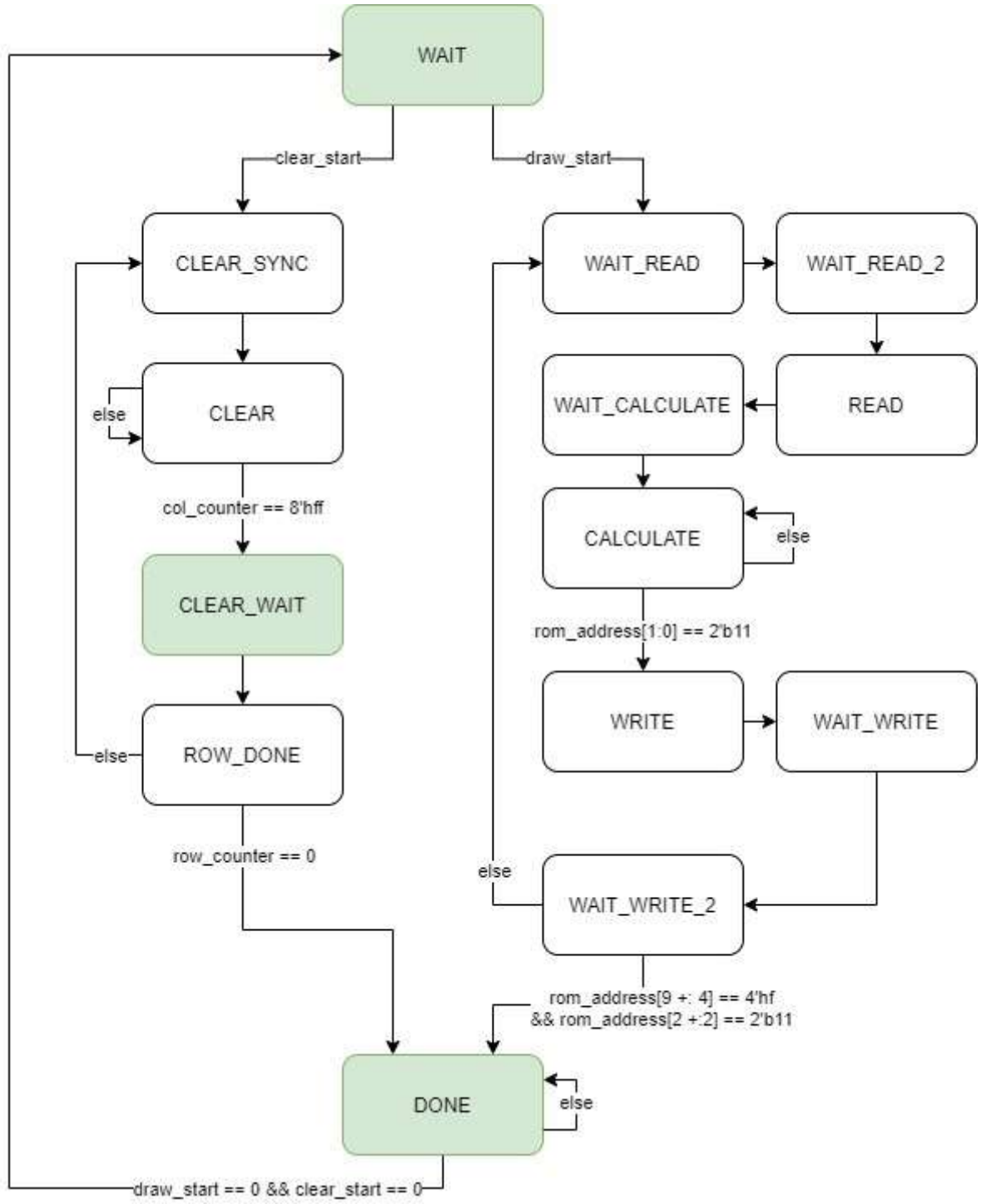


STATE DIAGRAMS

Current Frame Controller State Diagram



Next Frame Controller State Diagram



MODULE DESCRIPTIONS

Module: VGA_contoller

Inputs: Clk, Reset, and VGA_CLK

Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX and DrawY

Description: Outputs the correct horizontal and vertical signals required to know where the VGA is at.

Purpose: The VGA needs to know where on the grid of pixels it is at so it knows what it should paint based on the position it's at.

Module: hpi_io_intf

Inputs: Clk, Reset, from_ sw_r, from_sw_w, from_sw_reset, frome_sw_cs, [1:0] from_sw_address, [15:0] from_sw_data_out and OTG_DATA,

Outputs: OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N, [1:0] OTG_ADDR, [15:0] from_sw_data_in

Description: This module is the interface between the NIOS II and the EZ-OTG chip.

Purpose: The NIOS II and the EZ-OTG chip need to be connected in some way so they can interface with each other this module provides that connection.

Module: graphics_accelerator

Inputs: Clk, Reset, [4:0] spritesheetX, [4:0] sprtiesheetY, [9:0] imgX and imgY, draw_start, clear_start, [15:0] SRAM_DQ

Outputs: done, frame_clk, [7:0] VGA_R, VGA_G, and VGA_ B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N VGA_VS, VGA_HS, [15:0] SRAM_DQ, SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, [19:0] SRAM_ADDRESS

Description: The graphics accelerator switches off between the current frame controller and the next frame controller and also tells the software that the requested operation

was completed.

Purpose: Tells the software when the operation has been completed and controls the switching between the current frame controller and the next frame controller.

Module: rising_edge_detector (edge_detectors.sv)

Inputs: signal, Clk

Outputs: rising_edge

Description: detects rising edge of the clock

Purpose: Used for the frame clock (hardly used since we use the drawX and drawY to draw it is used on reset to reset the frame clock).

Module: top_level

Inputs: CLOCK_50, [3:0] KEY, [15:0] OTG_DATA, OTG_INT, [31:0] DRAM_DQ, [15:0] SRAM_DQ

Outputs: [6:0] HEX0 and HEX1, [7:0] VGA_R, VGA_B, and VGA_G, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [15:0] OTG_DATA, [1:0] OTG_ADDR, OTG_CS_N, OTG_RD_N, OTG_WR_N, OTG_RST_N, [12:0] DRAM_ADDR, [31:0] DRAM_DQ, [1:0] DRAM_BA, [3:0] DRAM_DQM, DRAM_RAS_N, DRAM_CAS_N, DRAM_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK, SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, [19:0] SRAM_ADDRESS

Description: This is the main module of lab 8 it specifies the connections between all submodules.

Purpose: To connect all submodules so the NIOS II could interact with the VGA and the USB correctly. For our final project we also added SRAM to have the NIOS II interface with the SRAM.

Module: spritesheetROM (ram.sv)

Inputs: Clk, address

Outputs: data

Description: ROM memory initialization for the spritesheet

Purpose: Stores the sprite sheet

Module: rowRAM

Inputs: Clk, write_address, read_address, we, data_in

Outputs: data_out

Description: ROM memory for the row that is currently being drawn.

Purpose: Store the current row.

Module: palette

Inputs: [3:0] colorIdx

Outputs: [7:0] VGA_R, VGA_G, VGA_B

Description: Has the hex values for the colors we used

Purpose: Outputs the correct VGA values for the color we want based on the colorID given.

Module: next_frame_controller

Inputs: [4:0] spritesheetX and spritesheetY, [9:0] imgX and imgY, draw_start, clear_start, [15:0] Data_from_SRAM

Outputs: done. Step_done, even_frame, [15:0] Data_to_SRAM, SRAM_WE_N, SRAM_OE_N, [19:0] SRAM_ADDRESS

Description: Contains a state machine for the next frame controller.

Purpose: The purpose of the next frame controller is to handle software requests. It has two possible functions: clearing and drawing. Drawing requests require the coordinates on the sprite sheet and screen to draw while clearing requests require no other input.

Module: curr_frame_conroller

Inputs: Clk, Reset, EN, even_frame, [15:0] Data_from_SRAM

Outputs: step_done, [7:0] VGA_R, VGA_G, VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, frame_clk, [15:0] Data_to_SRAM, SRAM_WE_N, SRAM_OE_N, [19:0] SRAM_ADDRESS

Description: Contains a state machine for the current frame controller.

Purpose: Reads the upcoming row of the current-frame buffer (CFB) into a row buffer in OCM at the start of horizontal blanking. Then, during the next row period, it outputs the corresponding color to the VGA interface.

Module: avalon_graphics_interface

Inputs: Clk, RESET, AVL_READ, AVL_WRITE, AVL_CS, [3:0] AVL_BYTE_EN, [2:0] AVL_ADDR, [31:0] AVL_WRITEDATA, [15:0] SRAM_DQ

Outputs: [31:0] AVL_READDATA, [7:0] VGA_R, VGA_G, and VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, [19:0] SRAM ADDRESS

Description: It is the interface between software and hardware and contains 7 registers to handle input from software.

Purpose: The writeable registers (R0-R5) allow the software to ask the accelerator to start a task. Register 6 is read-only and is where the accelerator can tell the software that the requested operation has completed.

Module: nios_system

Inputs: clk_clk, [15:0] otg_hpi_data_in_port, reset_reset_n, [31:0] sdram_wire_dq, [31:0] sram_data_sram_dq

Outputs: [7:0] keycode_export, [1:0] otg_hpi_address_export, otg_hpi_cs_export, [15:0] otg_hpi_data_out_port, otg_hpi_r_export, otg_hpi_reset_export, otg_hpi_w_export, reset_reset_n, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [31:0] sdram_wire_dq, [3:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, sram_data_ub_n,

sram_data_lb_n, sram_data_ce_n, sram_data_oe_n, sram_data_we_n

Description: The NIOS IIe processor. This processor interfaces with memory separately for instructions and data. For this lab we added a few new PIOs that included keycode, and all of the otg_hpi connections to ensure that our NIOS II could interact with the USB. We also added an SRAM interface to include SRAM.

Purpose: This is the main processing unit of the SoC. It will execute the software and interact with the VGA, USB, and SRAM.

Some modules not listed here were only used in previous iterations of our design but were preserved in code in case we wanted to re-use them.

DESIGN RESOURCES AND STATISTICS

LUT	4,343
DSP	0
Memory (BRAM)	1,064,064
Flip-Flops	2,513
Frequency (MHz)	113.12
Static Power (mW)	105.34
Dynamic Power (mW)	0.95
Total Power (mW)	106.29

SOFTWARE DESCRIPTION

The software has three important parts: the USB code, the graphics functions, the Game class. The USB code was not edited much from our original working version in Lab 8. We did remove some sleep commands or lower their duration to improve the frame rate of our game.

The graphics functions interface with our graphics accelerator. We had functions to draw a single 16x16 sprite, a 32x32 sprite, or a string of characters (using the alphanumeric sprites). We also had a function called swapFrameBuffers which would request a clear

from the NFC, resetting the canvas.

The Game class has two important public functions: update and draw (which should be called in that order). The update function takes the current key inputs and updates the state of the game accordingly (this is the function which imposes the rules of the game). The draw function translates the state of the game into visuals for the screen. It calls the graphics functions first to draw sprites to screen and then to swap the NFB.

REFERENCES

ECE 385 Helper Tools

Conforming an image to a color palette (see answer by user Trang Oul)

Game concept by Asher Aryam

USB keycode header file

SIMULATION

We haven't included any simulations here because we did not use simulation for the current iteration of our modules. We did have simulations for early versions, and they seemed to work well, but we found that there were many issues that only became visible once the hardware was connected to the software, which isn't something we could simulate.

In our debugging we used the SignalTap software, but we do not have the memory capacity to be able to show the signals on a timescale useful for a "simulated" demonstration. For that reason, we hope that the absence of simulations is not counted against us, as the visual nature of the final project is the truest representation of our work.

CONCLUSIONS

We learned a lot through the process of building this project, mostly because we had to redesign our hardware a few times before we found a system that worked.

At first, we had just one submodule inside the avalon_graphics_interface just like Lab 9 did. As we began to write the state machine for the module, we realized that we had a

problem: The current frame needed to be read from consistently at the beginning of a row in order for the VGA output to be correct. Meanwhile software requests also used the SRAM and so had to share a separate “substate” (i.e. they had to remember which address they were on). To solve this, we created separate modules for the CFC and NFC and split clock cycles between them after talking with Professor Cheng. After that we added another small modification which wouldn’t split the time half-and-half but would rather allot it in chunks, reducing the time wasted due to SRAM-signal synchronizers. This first redesign taught us the value of subdividing tasks to make it easier to understand what had to be done to solve each part of a problem.

The next redesign had to do with when and how we cleared the frame buffers to prepare for the next set of drawings. Initially, we had the CFC reading a row and then immediately clearing it. This system worked in the beginning, so we didn’t think twice about it for a while. However, once we added pathfinding code for the monsters, we saw that the screen would blink every time a monster moved. We realized that our hardware system was making a critical assumption that the software would be able to draw everything every frame. So when the pathfinding code took longer than a frame to complete, the CFB would already be cleared and would then draw a blank frame. To solve this we changed the clearing mechanism to happen on a software call. (The details of this current system are in the System Description section.)

There are a bunch of other bugs that we found over the course of development as well -- both in software and hardware -- that taught us lessons about design, about understanding code given to you, and more. But the biggest lesson we learned was that it is important to have a vision or a goal, because that is what motivated us to continue even when our current situation wasn’t so good. During the Week 2 checkpoint, we had nothing to show but basic simulations of the CFC and NFC (not even in their final forms). It wasn’t until a few days after that checkpoint that we were even able to draw something to the screen. And yet, in the next two weeks we managed to implement the game rules without simplification and on top of that add support for features like a leaderboard, a title screen, actually appealing sprites, and our own custom levels.