ECE 385

Spring 2021

Final Project

Pac-Man Game

Wenhao Tan & Jiacheng Huang
ABH
Jiaxuan Liu

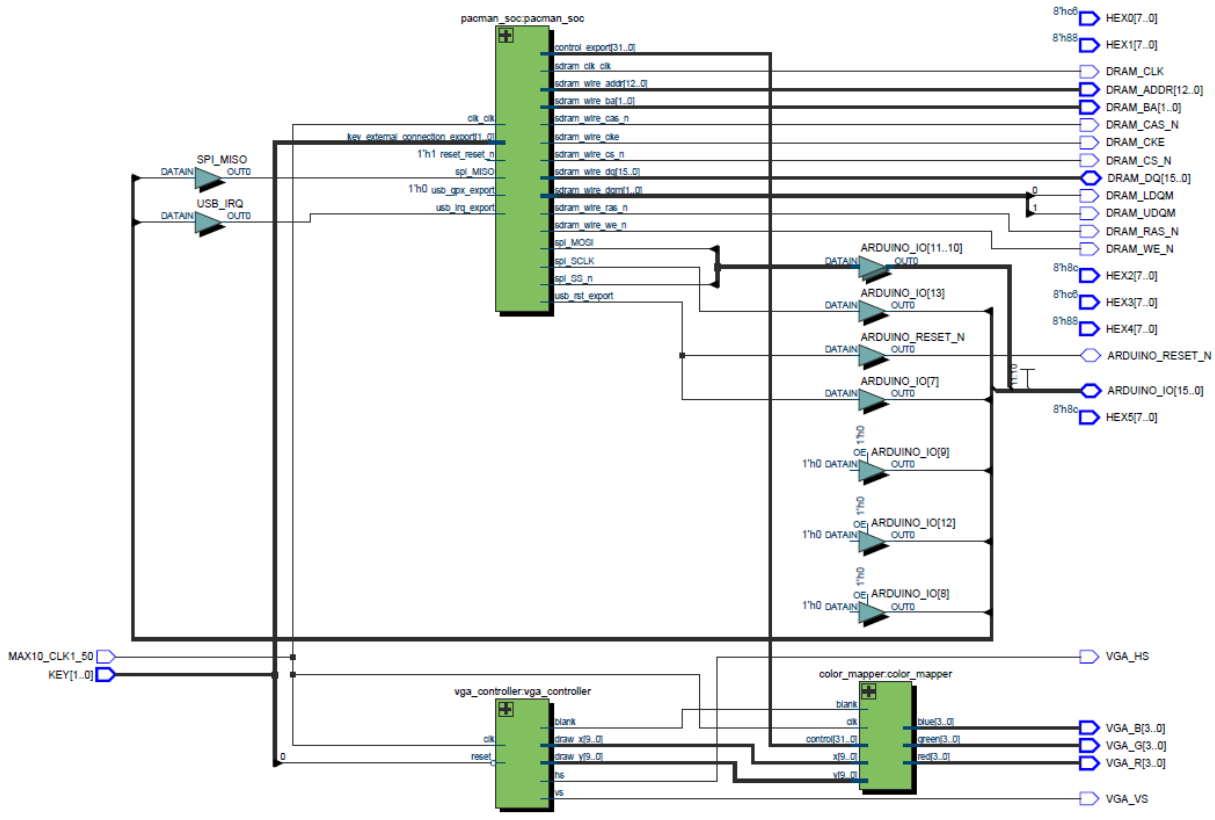
Introduction

Our final project is an implementation of the famous Pac-Man game on the DE-10 Lite FPGA board with both a single-player mode and a multiplayer mode, which includes both hardware and software parts, coding respectively in SystemVerilog and C programming language. The game will be displayed on a VGA monitor and can be played using a keyboard.

Written Description of Hardware Component

The hardware part is made up with an SoC, VGA controller, and a color mapper. The color mapper is the primary part that we were working on, in which the map is divided into 40 * 30 sprites. Each sprite is made up of 16 by 16 pixels, and each pixel chooses a color from the palette that we preconfigure. During each compilation, the color mapper reads the “pallete.txt” and “sprite_sheet.txt” files and encode the colors and sprite indices that we use. During execution, the color mapper receives a 32-bit control signal from the SoC to set a location to be a specific sprite on the map. In this way, whenever the VGA controller tells the color mapper a specific pixel location, the color mapper knows the sprite by its corresponding, and is able to return a specific rgb color to the VGA port.

Top Level RTL Diagram



SV Modules Descriptions

Module: `pacman.sv`

Inputs: `MAX10_CLK1_50,[1:0]KEY`

Outputs: `[7:0]HEX0-5,VGA_HS,VGA_VS,[3:0]VGA_R,[3:0]VGA_G,[3:0]VGA_B`

Descriptions: HEX displays are hard-coded to "PACPAC" when the FPGA is programmed. VGA_HS and VGA_VS are connected with the horizontal and vertical sync from the vga controller. VGA_R, VGA_G, and VGA_B are connected with the red, green, and blue signals in the color mapper.

Purpose: The top-level entity module of the project. Initializing all the required modules and connecting SOC with other components so that signals from the software can be sent to the corresponding hardware.

Module: `color_mapper.sv`

Inputs: `clk,blank,[9:0]x,[9:0]y,[31:0]control`

Outputs: `[3:0]red,[3:0]green,[3:0]blue`

Descriptions: The `x` and `y` inputs are read from the `draw_x` and `draw_y` signals in the vga controller. The `control` input is the instruction we read from NIOS II. We also hard coded a pallete.txt, which contains the 16 colors we have in the game, and sprite_sheet.txt, which contains the color information of all the sprites. We use the control instruction to determine the next scene to be outputted on the VGA monitor and output the `red`, `green`, and `blue` signals. If the `blank` signal is low, we have to output pure black.

Purpose: This module takes in the control instruction from the NIOS II and determines what needs to be displayed on the VGA monitor for the next frame.

Module: `vga_controller.sv`

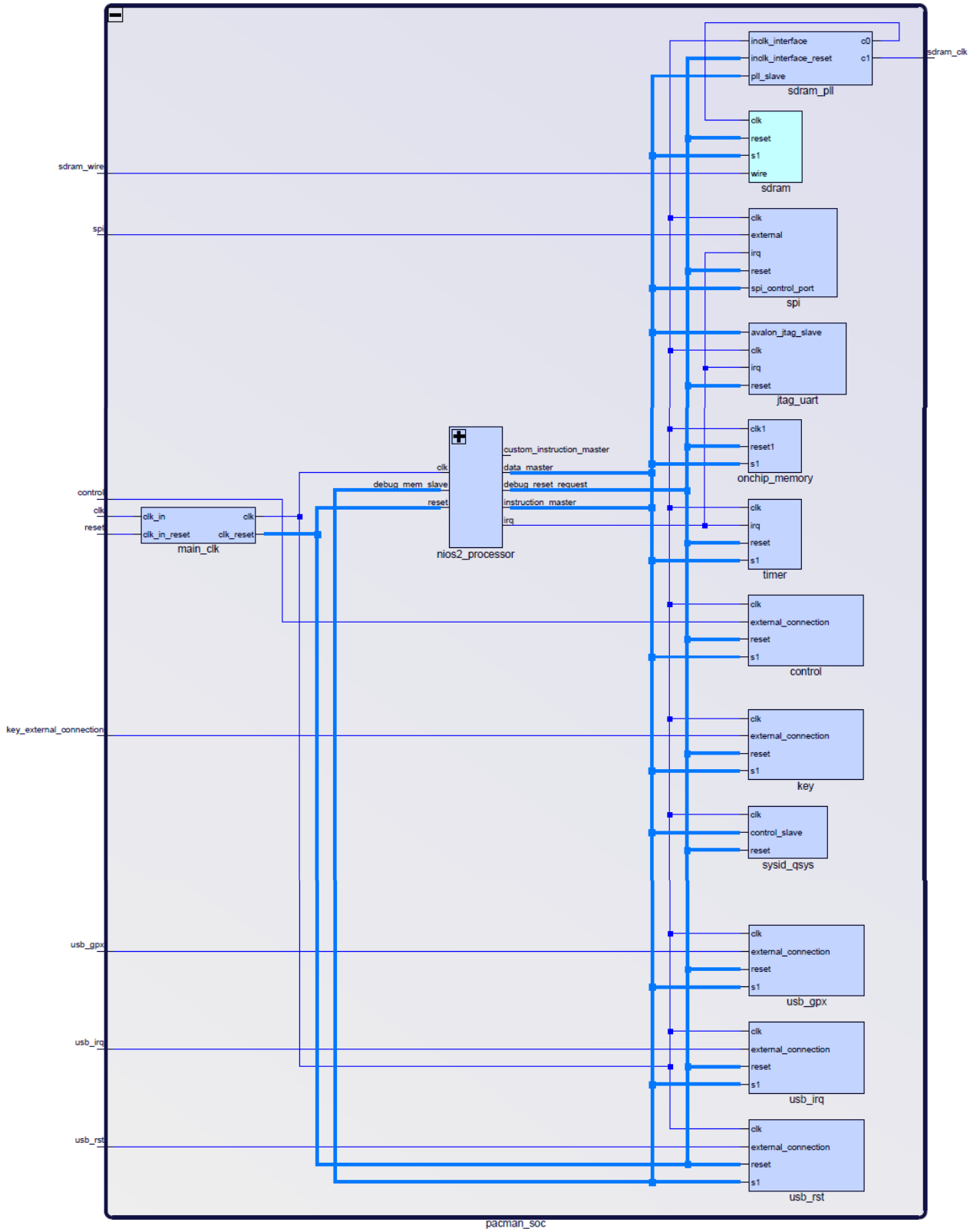
Inputs: `clk,reset`

Outputs: `hs,vs,pixel_clk,blank,sync,[9:0]draw_x,[9:0]draw_y`

Description: The VGA controller module produces the timing signals Horizontal Sync (`hs`) and Vertical Sync (`vs`) required by any VGA monitor. `pixel_clk` is half of the input clk, so it is 25MHz. `blank` is the blanking interval, and `sync` is the Composite Sync signal. `draw_x` and `draw_y` are the X and Y coordinates on the VGA monitor.

Purpose: This module is used to produce the signals we need to have a proper VGA display on the monitor.

System Level Block Diagram



Written Description of Software Component

The software part is divided into the hardware API part and the game development part. We have written an easy-to-use SPU (Sprite Processing Unit) API in order to select the sprite we want to display in the corresponding 16 by 16 grid. We have implemented the game logic on top of our SPU API to synthesize game maps and menus. We also modified the provided keyboard API in lab 6.2 to control the characters in-game. We have implemented a naive AI algorithm for the enemies in the game as well.

C Files Descriptions

File: `main.c`

`spu_control` sends the 32-bits instruction from NIOS II to the control address in the FPGA.
`random_change_direction` randomly changes the direction of the sprite.
`ghost_go` determines the ghosts' next step movements. We implemented a naive algorithm for the ghosts to move towards the location of pac-man. By using a random number generator, the ghosts have 1/5 chance of chasing the pac-man, 1/10 chance of randomly changing direction, and remaining probability to continue its previous direction.
`pacman_task` takes in the keyboard input and determines the pac-man's next step movement. We used the arrow keys for pac-man movements.
`two_player_task` is used when playing multiplayer mode. It takes in two keyboard inputs simultaneously and determines the pac-man and ghost's next step movements. We used the arrow keys for pac-man movements, and WASD for ghost's movement.
`init_game` initializes the map layout for the single player mode.
`init_2game` initializes the map layout for the multiplayer mode.
`main` initializes the keyboard API and starts to read in keyboard inputs. We initialize the game in a start menu, and use the keyboard to select which game mode to play.

File: `pacman.c`

`spawn_all_sprites` displays all the sprites encoded in FPGA for debugging purposes.
`sprite_index` returns the index of a sprite.
`sprite_type` returns the type of a sprite.
`sprite_direction` returns the direction of a sprite.
`sprite_property` returns the property of a sprite.
`get_pacman` gets the index of a pacman.
`get_ghost` gets the index of a ghost.
`get_food` returns the index of a food.
`get_sprite` returns the index of a sprite.
`map_get_sprite` gets the sprite on a specific location on map.
`map_set_sprite` sets the sprite on a specific location on map.
`next_pacman` gets the next sprite of pacman in animation.

`next_ghost` gets the next sprite of ghost in animation.
`next_sprite` gets the next sprite in animation.
`animate_map` animates the map to their corresponding next sprites.
`main_map` generates the single player map.
`game_over` generates the “Game Over” scene.
`you_win` generates the “You Win” scene.
`start_game` generates the “Start Game” scene.
`can_walk` returns if a sprite can walk in a specific direction.
`show_score` shows the score on the top of the scene.

File: `spu.c`
`spu_set_sprite` sends the sprite to the hardware.
`spu_set_map` sends the map to the hardware.

Design Resources and Statistics

LUT	17482
DSP	0
Memory (BRAM)	1225728
Flip-Flop	12050
Frequency	128.01 MHz
Static Power	96.18 mW
Dynamic Power	0.73 mW
Total Power	106.25 mW

Problems
At first, we wanted to store rgb data for each pixel (640 * 480) of the VGA in hardware, but we soon find out that we do not have enough resources on DE-10 Lite to do that. We have tried more than three design patterns, such as storing 4 pixels as 1 megapixel, using MUXes instead, using pure software to send data and then render, but none of them worked, either because of the lack of memory or logic gates, or because of the slow frequency caused by software. Finally, we came out of a solution to divide the 640 * 480 VGA pixels into 40 * 30 sprites wit each sprite containing 16 * 16 pixels. Instead of storing rgb data in each pixel, we store the sprite indices at

each location, and a palette for the few colors we use. In this way, we are able to find the corresponding color at a specific pixel.

Conclusions

During this lab, we got more familiar with how to use Platform Designer to connect hardware (.v in Quartus) and software (.c in Nios II), because this time we are making this game from ground up, and we have to write a simple driver for a hardware component to be used in the software side. Therefore, we have to take care of each detail involved in this game development. More importantly, we learned how to divide a project into multiple layers.