ECE 385

Spring 2019

Final Project Report

Game: Dr. Boom

A Game of Bomberman

Tingfeng Yan; Zerui An

Section ABC / Tuesday 11:30 AM

TAs: Zhenhong Liu, Gene Shiue

Introduction

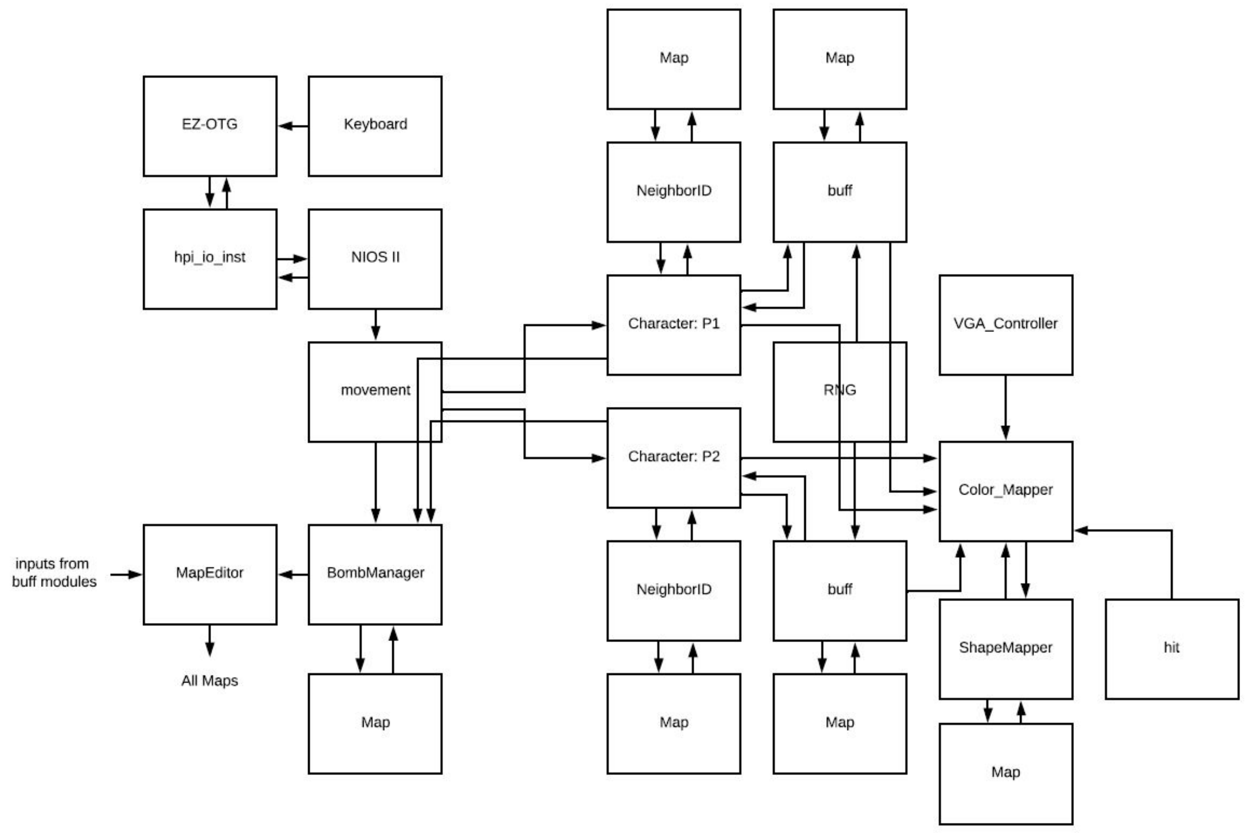
This is our final project for this course, so we decide not to make something easy like a digital filter to just pass the final. We put our eyes on games because a game is one of the best forms to present. Considering what we've learnt so far, there are not many game forms to select from. Our final decision is to create a game like bomberman, and we call it Dr. Boom.

Description of Circuit

The software part of our project is written in C code and runs on the NIOS II processor. It's only purpose is to collect key press information and pass it to the hardware.

The hardware part has several functions:
Keycode processor: interact with the software, decode the "purpose" of each key press.
Game control: modify the characters and the map based on key press and other internal logic.
Display: display the game board as well as scores and other parameters using VGA.

Block Diagram



Description of Modules

Module: VGA_controller.sv
Inputs: Clk, Reset, VGA_CLK
Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0]DrawX, [9:0]DrawY
Description: This module outputs data to the VGA port.
Purpose: Control the VGA port and anything connected to it.

Module: timer_64Hz.sv
Inputs: Clk, slow_clk
Outputs: [11:0]timer
Description: This module is a counter that increments on the rising edge of slow_clk.
Purpose: Other modules can use the output of this module to keep track of the time.

Module: ShapeMapper.sv
Inputs: [ADDR_WIDTH-1:0]raddr, clk
Outputs: [DATA_WIDTH - 1:0] q
Parameters: ADDR_WIDTH = 10, DATA_WIDTH = 9
Description: This module maps each ID on the spritesheet to characters, bombs and all other figures.
Purpose: Display correct shapes on the monitor.

Module: ScoreBoard.sv
Inputs: Clk, Reset, p1gain, p2gain
Outputs: [7:0]p1score, [7:0]p2score
Description: This module calculates the score of each player (in decimal).
Purpose: Display correct scores on the monitor.

Module: RNG.sv
Inputs: Clk, Reset, [15:0]seed
Outputs: [3:0]randhex
Description: This module takes a 16-bit seed and generates an integer between 0 and 15 every clock cycle. The algorithm used is linear feedback shift register.
Purpose: Generate a random 4-bit number for use.

Module: queue.sv
Inputs: [DATA_WIDTH-1:0]in, we, Clk, Reset, next
Outputs: [DATA_WIDTH-1:0]out, empty, full
Parameters: ADDR_WIDTH = 8, DATA_WIDTH = 8

Description: This module generates a FIFO data structure with suitable data width.
Purpose: This module is used to store bomb information which is waiting to be processed.

Module: movement.sv
Inputs: [15:0]keyA, [15:0]keyB, [15:0]keyC
Outputs: [7:0]p1move, [7:0]p2move, p1bomb, p2bomb
Description: This module analyses the 6 input keycodes, and set the outputs based on the last effective keys.
Purpose: Send keycode information to the correct modules.

Module: Map.sv
Inputs: [ADDR_WIDTH-1:0]waddr, [ADDR_WIDTH-1:0]raddr, [DATA_WIDTH-1:0]wdata, we, clk
Outputs: [DATA_WIDTH - 1:0]q
Parameters: ADDR_WIDTH = 8, DATA_WIDTH = 8
Description: Simple duo-Port RAM with different read/write addresses and single read/write clock and with a control for writing single bytes into the memory word.
Purpose: Initialize the map on the monitor.

Module: MapEditor
Inputs: WE0, WE1, WE2, WE3, WE4, WE5, Clk, Reset, ROM_Clk,
[7:0] addr0, addr1, addr2, addr3, addr4, addr5, data0, data1, data2, data3, data4, data5,
Outputs: WE, [7:0] MapWriteAddr,[7:0]MapWriteData
Description: This module contains a state machine, a multi-input queue, and a ROM that stores the initial map.
Purpose: This module is used to handle reset requests and all write-to-map requests.

Module: NeighborID
Inputs: [9:0]X_Pos, [9:0]Y_Pos, [9:0]Step, [7:0]waddr, [7:0]wdata, we, clk,
Outputs: up, down, left, right
Description: This module constantly read from the map to get the IDs of the items surrounding a player.
Purpose: Determines whether the player is allowed to move further in a direction.

Module: hpi_io_intf.sv
Inputs: Clk, Reset, [1:0]from_sw_address, [15:0]from_sw_data_out, from_sw_r, from_sw_w, from_sw_cs, from_sw_reset
Outputs: [15:0]from_sw_data_in, [1:0]OTG_ADDR, OTG_RD_N, OTG_WR_N, OTG_CS_N, OTG_RST_N

Inout: [15:0]OTG_DATA
Description: Write data from EZ-OTG chip to NIOS II if writing is enabled. Otherwise, set high Z.
Purpose: Interface between NIOS II and EZ-OTG chip.

Module: hit.sv
Inputs: Clk, Reset, p1gain, p2gain, [9:0]timer
Outputs: p1red, p2red
Description: p1red is set to high and kept for 0.5 second when p2gain is high. The same goes for p2red and p1gain.
Purpose: Turn the character figure red when it gets hit.

Module: HexDriver.sv
Inputs: [3:0]In0
Outputs: [6:0]Out0
Description: This module transform a 4-bit number to a 7-bit number whose bits correspond to the hexadecimal representation of the original number.
Purpose: Correctly displaying patterns corresponding to their values using the 7-segment display blocks on board.

Module: explosion.sv
Inputs: clk, ready, player, [9:0]timer, [1:0]length_in,[7:0]Data_IN, [7:0]coordinate, [3:0]randhex
Outputs: WE_O, t,[7:0] Data_O, [7:0]Address
Description: This is a state machine designed to edit the map as needed when a bomb is supposed to explode.
Purpose: Create and clear the explosion effect, and place treasure chests based on the input random number.

Module: ColorMapper.sv
Inputs: is_Char1, is_Char2, ROM_Clk, dir1, dir2, p1red, p2red, [9:0]Char1_step, [9:0]Char2_step, [1:0]bomb1_size, [1:0]bomb2_size, [9:0]delay1, [9:0]delay2, [9:0]cooldown1, [9:0]cooldown2, [9:0]DrawX, [9:0]DrawY, [9:0]Char1_X_Pos, [9:0]Char1_Y_Pos, [9:0]Char2_X_Pos, [9:0]Char2_Y_Pos, [7:0]p1score, [7:0]p2score, [23:0]defaultRGB
Outputs: [7:0]VGA_R, [7:0]VGA_G, [7:0]VGA_B
Description: This module sets colors for all the items, characters, and maps.
Purpose: Display items, characters, and maps with the correct color on the monitor.

Module: Character.sv

Inputs: Clk, Reset, frame_clk, [9:0]DrawX, [9:0]DrawY, [9:0]X_Reset, [9:0]Y_Reset, [9:0]Step, [7:0] move, upID, downID, leftID, rightID
Outputs: is_Char, [9:0]Char_X_Pos, [9:0]Char_Y_Pos, dir
Description: Updates character's position and movement every time the screen refreshes.
Purpose: Control character movement based on keypress ([7:0]move). Outputs is_Char and dir are used to help Color mapper display the character correctly. Char_X_Pos and Char_Y_Pos are used in other modules.

Module: buff.sv
Inputs: Clk, Reset, player, [7:0]id, [3:0]randhex
Outputs: WE, [9:0]step, [1:0]size, [9:0]delay, [9:0]cooldown
Description: This module generates buffs or debuffs for both characters.
Purpose: Add more elements to the game and make it more enjoyable.

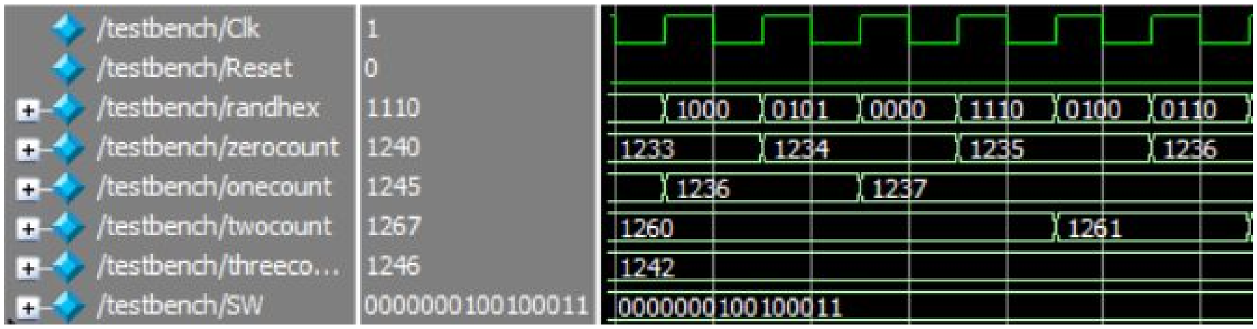
Module: BombManager.sv
Inputs: Clk, Reset, p1bomb, p2bomb, [9:0]timer, [9:0]cooldown1, [9:0]cooldown2, [9:0]delay1, [9:0]delay2, [9:0]Char1_X_Pos, [9:0]Char1_Y_Pos, [9:0]Char2_X_Pos, [9:0]Char2_Y_Pos, [7:0]id1, [7:0]id2, [1:0]p1size, [1:0]p2size, [3:0]randhex,
Outputs: WE0, WE1, WE2, WE3, p1gain, p2gain, [7:0]addr0, [7:0]addr1, [7:0]addr2, [7:0]addr3, [7:0]data0, [7:0]data1, [7:0]data2, [7:0]data3
Description: This module is made of a state machine and other helper modules including queue and explosion. The state machine is used to handle all "placing bomb" requests, store bomb informations and provide necessary inputs for the explosion modules to function correctly.
Purpose: Handles all bomb-related tasks.

Module: BomberMan_top.sv
Inputs: CLOCK_50, [3:0]KEY, [15:0]SW, OTG_INT
Outputs: [6:0]HEX0, [6:0]HEX1, [6:0]HEX2, [6:0]HEX3, [6:0]HEX4, [6:0]HEX5, [6:0]HEX6, [6:0]HEX7, [7:0]VGA_R, [7:0]VGA_G, [7:0]VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [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_CKE, DRAM_WE_N, DRAM_CS_N, DRAM_CLK
Inout: [15:0]OTG_DATA
Description: This is the top level of the whole project.
Purpose: Connect all modules together and make them work as expected.

Problems Encountered

How to generate (pseudo) random numbers ?

The first solution we thought of is to use the RNG functions in C. We would set up an Avalon MM slave, and modify our C code such that every time the software reads the key press, it also generates a random number and write it in the memory mapped register.
However, the C code is slow, so we do not want to give it too much work to do. Besides, we do not need a super good RNG since we are only using it for some add-ons in our game. Therefore we decided to write a hardware RNG. After doing some research on popular RNG algorithms, we chose to use linear feedback shift registers, which are easy to write and very widely used.



Note: the four "counts" are used to keep track of the bit pattern of the last 2 bits fo output. It can be seen from the simulation that the four bit patterns (00, 01, 10, 11) appeared roughly the same times.

How to handle multiple "write to map" requests?

Our map (aka game board) is stored in on-chip memory, so it can only handle one write request per clock cycle. However, we have multiple elements that need to edit the map. A simple solution is to discard all other write requests except the one with highest priority. This approach would mostly work fine since humans are much slower than the system running at 50 MHz, so it is unlikely for multiple write requests to occur at the same clock cycle.
Rather than assuming the problem would never occur, we would like to actually solve the problem. Starting from a FIFO data structure, we implemented a multi-input queue. Every clock cycle, it puts all the inputs (at most 6) into a queue and pass them out one by one. We made the queue long enough so that it is unlikely to overflow.

How do we know if a player scores and how to keep track of the scores?

It is stated in our rules that a player gets one point when the opponent is inside the affected area when a bomb explodes. Therefore we cannot simply check if a player is on a block with "explosion visual effect ID" and edit scores every clock cycle, as that may result in players getting multiple points from one explosion. What we ended up doing was to check players' positions when a bomb explodes and writes the visual effect to the map. Since this action only happens once per explosion, we can be sure that players scores increment by at most one each time.

How and when do we add power-ups to the map?

Simple answer: add them randomly. But the hard problem is what the trigger for a "random test" should be. We came up with several possible solutions:

(1) Every clock cycle or frame_clock cycle, check the RNG to see if an item should be generated. If so, use another RNG to determine the location of the generated item.
The problem of this approach is that both clocks are too fast, so that we need a very low success rate when running the first "random trial" to avoid flooding the screen with items. Such a low success rate is only possible with a RNG that has long output width. Besides, the algorithm we used to generate random numbers may not be stable enough to handle such a heavy task perfectly.

(2) Everytime a breakable wall is destroyed, generate a random item on that location.
This feature is not very hard to implement, but the total number of items would be limited, and the game would not be so fun once all the items are used up. Therefore we were not quite satisfied with this solution either.

(3) Items would randomly come out from bombs.
This solution may not make much sense at first sight, but actually solves both problems that idea(1) and (2) have. We finally decided to use this approach. Furthermore, to increase playability, we added the rule that the items from one player's bomb can only be used by the other player.

How do we design the scale of every grid?

Considering the fact that we have a 640*480 screen to display our work, we need to divide the space properly to satisfy the following two requirements: every element in a grid should be clear enough for people to tell what they are; every character should have enough space to move around. Ideally, the dimensions should be equal to or slightly less that some integer power of 2 to make writing and reading easier. After consideration, we decide to make each grid 32*32, which makes the whole screen 20*15 grids. The left 15*15 is for the game map while the rest 5*15 is used to display useful information for players.

Design Resources

LUT	6,857
DSP	0

BRAM	2,322,432 bits
Register	3,516
Frequency	57.01 MHz
Static Power	105.90 mW
Dynamic Power	67.37 mW
Total Power	262.11 mW

Conclusion

Final project is more helpful than any of our previous labs. We actually learn a lot and combine our knowledge to create such a work that we are proud with. It’s like none of the previous labs where we just follow the procedure and things will be done correctly. This project has more design components than adding together all those included in labs. Every graph, every number, every move requires consideration. The question is not about how to make it functional but how to make it at least playable. Due to the time constraint and pressure from other courses, we were not able to finish all we think of. But after all, this is an unprecedented practice for us and the result is good enough. We know the experience of this project will help us a lot in the future.