

**ECE 385**  
Spring 2020

Final Project

## **Brick Breaker with USB and VGA Interface in SystemVerilog**

Ishaan Datta, Lukas Dumasius  
Lab ABE/Tuesday 3pm  
Jiaxuan Liu

### **Idea and Overview:**

The ECE 385 course has given us an opportunity to work with FPGA's, USB keyboards, VGA devices and more. During the Lab 8 exercise, we expanded our abilities with the DE2 board with

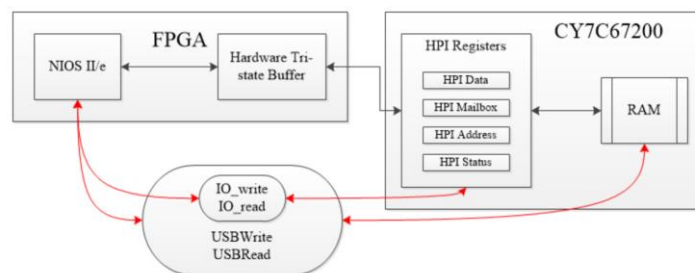
VGA and USB interaction by taking advantage of the NIOS System on Chip (SOC). We learned how to interface a USB keyboard with the DE2 FPGA board using the NIOS II processor and send keyboard signals to the FPGA. Similarly, we connected a VGA monitor to the FPGA using a USB Blaster connector to display the output. In the Lab 8 “Bouncing Ball” experiment, we added all of the components together to implement a bouncing ball which followed basic physics and bounced off of walls that we could display via VGA interface. **In our Final Project, we have extended this concept to implement a small game – a Brick Breaker.** Essentially this game involves using a paddle to bounce a ball and break bricks that are at the top of the screen. This project gave us an opportunity to get a thorough understanding of clocks, timings, gates vs latches – and especially parallel vs sequential execution that we commonly see manifest in an FPGA implementation in the form “always ff” and “always comb” logic.

### Summary of Operation:

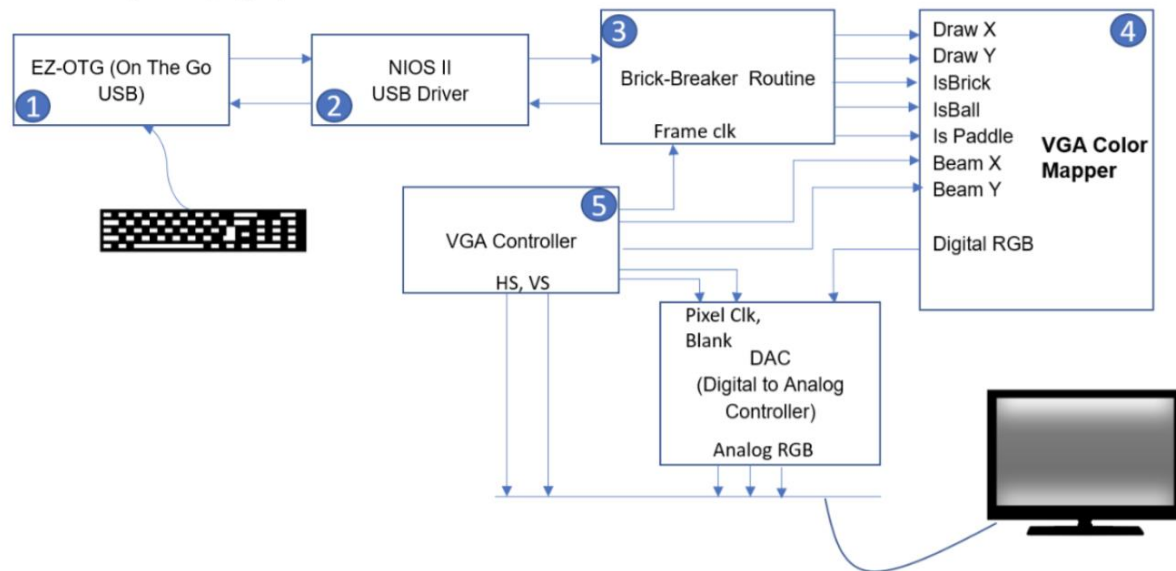
We have taken the opportunity to leverage our skills gained from lab 8 in the “Bouncing Ball” project to create an extension with a few additional components: The Brick Breaker game. Therefore, besides the bouncing ball, we have implemented a paddle that is moved left or right in response to the keys ‘A’ for left and ‘D’ for right, and when the ball hits a brick, the brick is destroyed. There are 12 bricks implemented in this project in a matrix comprised of 3 rows and 4 columns. The number and dimension of bricks is easily controlled via global variables. The ball can hit any brick from top, bottom or edge, it should destroy the brick and bounce in the opposite direction.

The hallmark of this project is that we have implemented the logic using structs, arrays and for-loops, all implemented in SystemVerilog using Quartus Prime software. Usually, all such projects that we observed were implemented using hardcoded variables for placement of each brick. However the tediousness of this approach led us to explore a better alternative to implementing the Brick Breaker.

With the NIOS II SOC we use a memory mapped implementation to connect to the peripherals with specific memory addresses for read/write operations. All of the balls movements are displayed via a Visual Graphics Array (VGA) connection which is organized as a matrix of 480x640 pixels. Drawing on our experience in lab 8, the ball.sv module contains logic for the white circular bouncing ball that moves across the screen and reflects off of the borders with appropriately changing X and Y movement based on keyboard presses.



**Block Diagram (Fig 1)**



## Description of components and code

As given in the diagram above, the system has the following components:

1. The EZ OTG (hpi\_io\_intf.sv) interface for the keyboard. This accepts USB protocol signals from the keyboard and further feeds it to the NIOS-II processor.
2. The NIOS-II USB driver to convert the USB signals to keyboard signals and sends them to the brick breaker routine.
3. The Ball-breaker routine (Ball.sv) is where we have primarily added on to the Bouncing Ball project. This is the main module which implements our game logic. It accepts keyboard inputs and moves the paddle around. It also implements the bouncing ball and the bricks.
4. The Color Mapper (Color\_Mapper.sv) is essentially used for accepting the current X & Y pixel location in the 640x480 screen and giving it a RGB colour depending on whether the current location is a ball, paddle, brick or background. We have modified the code in the color-mapper (given in Lab 8) to deal with the bricks and the paddle. Essentially we

have added extra inputs for the Is\_Paddle and Is\_Bricks and assigned different colors to them.

5. The VGA Controller (VGA\_controller) drive the beam, Frame Clock, Pixel Clock Horizontal/ Vertical shift signals.
6. The Overall controlling SV file is the already provided lab8.sv. We have modified this file to cater to the extra Paddle and Brick interfaces between ball.sv and color\_mapper.sv.

## Module Descriptions

### - Module: ball.sv

Inputs: Clk, Reset, frame\_clk, [9:0] DrawX, DrawY, [7:0] key\_code

Outputs: is\_ball

Description & Purpose: This module essentially serves to handle the movement of the ball on the VGA display. The module most importantly keeps track of the ball and the screen edges/boundary conditions defined within this file. The ball is redirected if it hits the screen edge, and its direction is changed in accordance with the user input through keyboard presses W, A, S, D.

### - Module: Color Mapper.sv

Inputs: is\_ball, [9:0] DrawX, DrawY

Outputs: [7:0] VGA\_R, [7:0] VGA\_G, [7:0] VGA\_B

Description & Purpose: This module serves to solely handle the colors on the background of the screen and the ball itself so we can clearly see its movements. The background is set to a purple-ish gradient (the background ends up looking just like you would have used a gradient/paint bucket tool in Adobe PS) and the ball is kept white as it bounces around.

### - Module: hpi\_io\_intf.sv

Inputs: Clk, Reset, from\_sw\_r, from\_sw\_w, from\_sw\_cs, from\_sw\_reset,  
[1:0] from\_sw\_address, [15:0] from\_sw\_data\_out

Outputs: OTG\_RD\_N, OTG\_WR\_N, OTG\_CS\_N, OTG\_RST\_N,  
[15:0] from\_sw\_data\_in, OTG\_ADDR

Inout wire: [15:0] OTG\_DATA

Description & Purpose: This module serves as the HPI I/O interface, giving us a way to access and read/write data to the memory-mapped setup with the USB EZ-OTG as further detailed in a short Description of CY7/USB Protocol using HPI\_ADDR and HPI\_DATA in “**Summary of Operation**”.

### - 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 & Purpose: Using the 25 MHz VGA\_CLK as described above, this module handles our VGA display preparation, in a literal sense (in terms of the visual graphics array) using

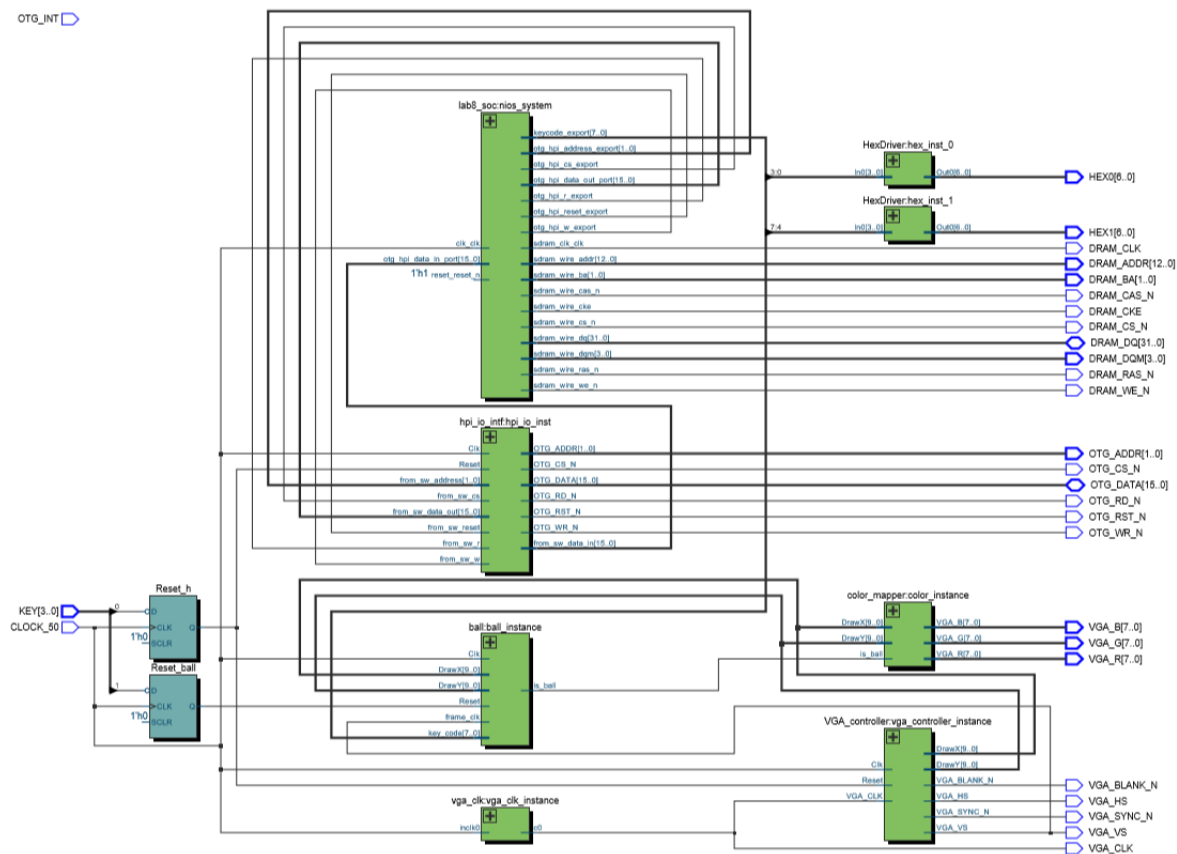
VGA\_HS and VS for horizontal/vertical sync pulse. Each pixel is handled as one element of the matrix, based on the display arguments received.

### - Module: nios\_system

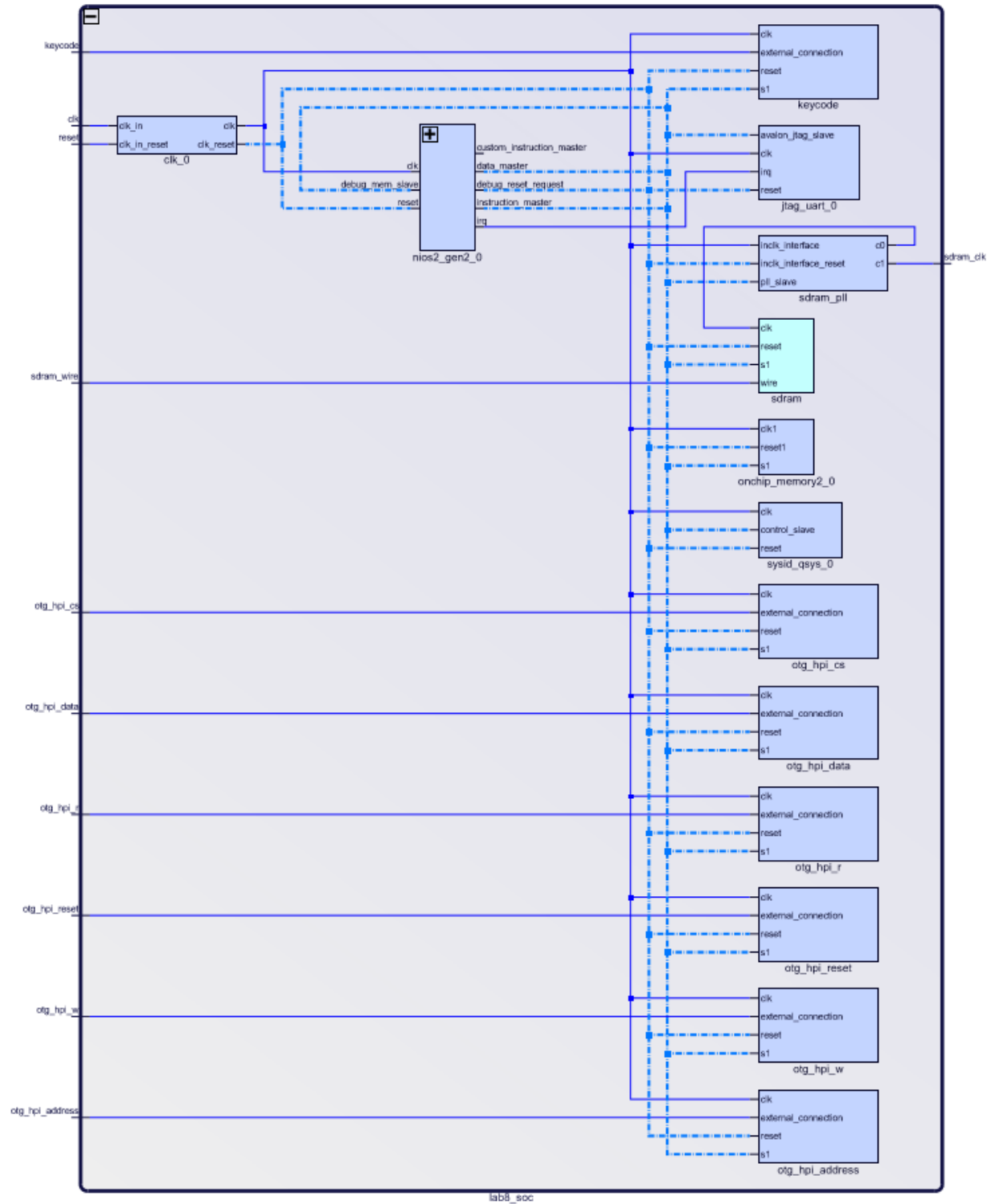
Inputs/Outputs: clk\_clk(Clk),reset\_reset\_n, sdram\_wire\_addr, sdram\_wire\_ba,  
sdram\_wire\_cas\_n, sdram\_wire\_cke, sdram\_wire\_cs\_n,  
sdram\_wire\_dq,sdram\_wire\_dqm, sdram\_wire\_ras\_n, sdram\_wire\_we\_n, sdram\_clk\_clk,  
keycode\_export, otg\_hpi\_address\_export, otg\_hpi\_data\_in\_port, otg\_hpi\_data\_out\_port,  
otg\_hpi\_cs\_export, otg\_hpi\_r\_export, otg\_hpi\_w\_export, otg\_hpi\_reset\_export

Description & Purpose: This is the Platform Designer/Qsys generated file to link to the rest of our SystemVerilog & top level. We connect in/out wires, components, and other glue logic signals to the top-level to make basic/simple hardware.

### Top Level Block Diagram



### System Level Block Diagram (Qsys/platform designer schematic)



**Complexity encountered in the project**


1. The primary issues that we encountered in the project was the implementation in code of the logic to implement bricks. These were:
  - a. A Logic element (e.g. Register) created/ initialised in one block (say the “Always ff” block) cannot be modified in another block (say the “Always Comb” block) as it causes “Multiple Constant Driver” error. Similarly, the vice versa is true.
  - b. Because of the above issue the implementation logic for every element, the ball, bricks and paddle become extremely complex wherein we have to preserve states of each logic element in a block and re-read them on the next clock.
  - c. Similarly we faced numerous issue in the If-The-Else logic (within “for loops”) as System Verilog tends to synthesize an element as an un-clocked “Latch” if the value of a logic element is not set in every branch of a loop. This is a highly undesirable problem as latches are not aligned to the clock and causes system instability.

### **Points of Interest and Innovation**

2. A unique aspect of this project is that unlike any other implementations on the Internet of Brick Breaker projects using Verilog, we have used standards C-like “Structs” and double dimension arrays successfully to implements the bricks.
3. The tricky part was to maintaining state between clock cycles. In order to avoid the issue of Multiple Constant Driver errors, we followed the approach taken to retain state of the ball between clock cycles. In this approach, two different 10 bit registers (logic elements) are used in “Always ff” and “Always comb” to maintain state of the ball, paddle and each brick, and the two registers are used to set each other’s state in the other block.
4. In every other such project, the location of bricks has been hard coded within the System Verilog code. We considered this an extremely in-elegant way to implement any project.
5. Therefore, we can easily modify the number of bricks and change their location or dimensions with minimal effort. All of this has been accomplished without synthesizing any element as a Latch.
6. Additionally, we had to be very careful while implementing logic in the “Always ff” block as we realised that all instructions within this block are executed concurrently and in parallel when the “posedge” (Positive Edge) of the clock signal is received. However, it was interesting to note that instructions within the “Always comb” logic are executed in sequence akin to a standard C program.
7. It was also interesting to note the rationale for the Frame Clock at 60 Hertz being used in the “Always comb” block while the 50MegaHertz standard clock was used to trigger the “Always ff” block. This was a key aspect in the project to make the ball as well as the paddle move at a reasonable speed – within the “Always comb” block. However, to keep refreshing the screen at a fast speed while preserving the location of the ball, paddle and bricks, the 50MHz clock was used to trigger the “Always ff” block.

### Design Resources and Statistics

LUT	2863
DSP	(image below)
Memory (BRAM)	55,296 / 3,981,312 ( 1 % )
Flip-Flop	2191
Frequency (MHz)	141.4
Static Power (mW)	105.15
Dynamic Power (mW)	0.84
Total Power (mW)	174.57

Analysis & Synthesis DSP Block Usage Summary		
 <<Filter>>		
	Statistic	Number Used
1	Simple Multipliers (9-bit)	0
2	Simple Multipliers (18-bit)	2
3	Embedded Multiplier Blocks	--
4	Embedded Multiplier 9-bit elements	4
5	Signed Embedded Multipliers	2
6	Unsigned Embedded Multipliers	0
7	Mixed Sign Embedded Multipliers	0
8	Variable Sign Embedded Multipliers	0
9	Dedicated Input Shift Register Chains	0



## Conclusions

One of the primary challenges we faced in this project was unit testing and debugging the separate components of the game that we were adding as we went. We tried adding multiple parts of the game such as bricks, a score counter, and the paddle itself and had a hard time reading through the signals and fixing bugs. We added onto our lab 8 test bench and after the ball movement from lab 8 was fixed, we simulated logic for the paddle and bricks separately to fix problems such as the paddle not responding to key presses and the invisible bricks. We would also recommend to future students to carefully step through their setup and settings in Quartus and Qsys if they aren't sure if they are correct before continuing to implement changes in code as we had a timing issue that caused the paddle to change positions on the screen unexpectedly which in hindsight we could have possibly spent less time debugging. Overall we feel satisfied choosing this project, as we wanted to implement something with the VGA display and this was challenging enough to be a good learning experience but also at our skill level. Further additions could be made with bricks, levels, and counters.