

ECE 385

Spring 2013

Final Project

“Dark Matter”

(Brick Breaker Clone)

Venkatasai Koppula, Raj Vinjamuri

ECE 385 - ABE (Tuesday, 3PM)

TA: Vishnu Priya Muthukrishnan; Amit Sangai

At a Glance:

- Gameplay:
 - When the ball hits the bottom, the player loses.
 - When all bricks have been destroyed, the player wins.
 - When a brick is broken, the score is incremented by +10 points (A – Hex)
 - When the ball hits the paddle, successfully staying in play, the score is incremented by +3.
 - When the player wins, by destroying all blocks, the score flashes around to celebrate!
- Visuals:
 - Moving gravity beam to reflect dark matter cluster
 - Dark matter impacts and spontaneously destroys blocks of orange-matter
 - Red LEDs: Seed Generation
 - Green LEDs: Status bits
 - (Win)(Brick Impact)(Paddle Impact)(Loss)(Reset)

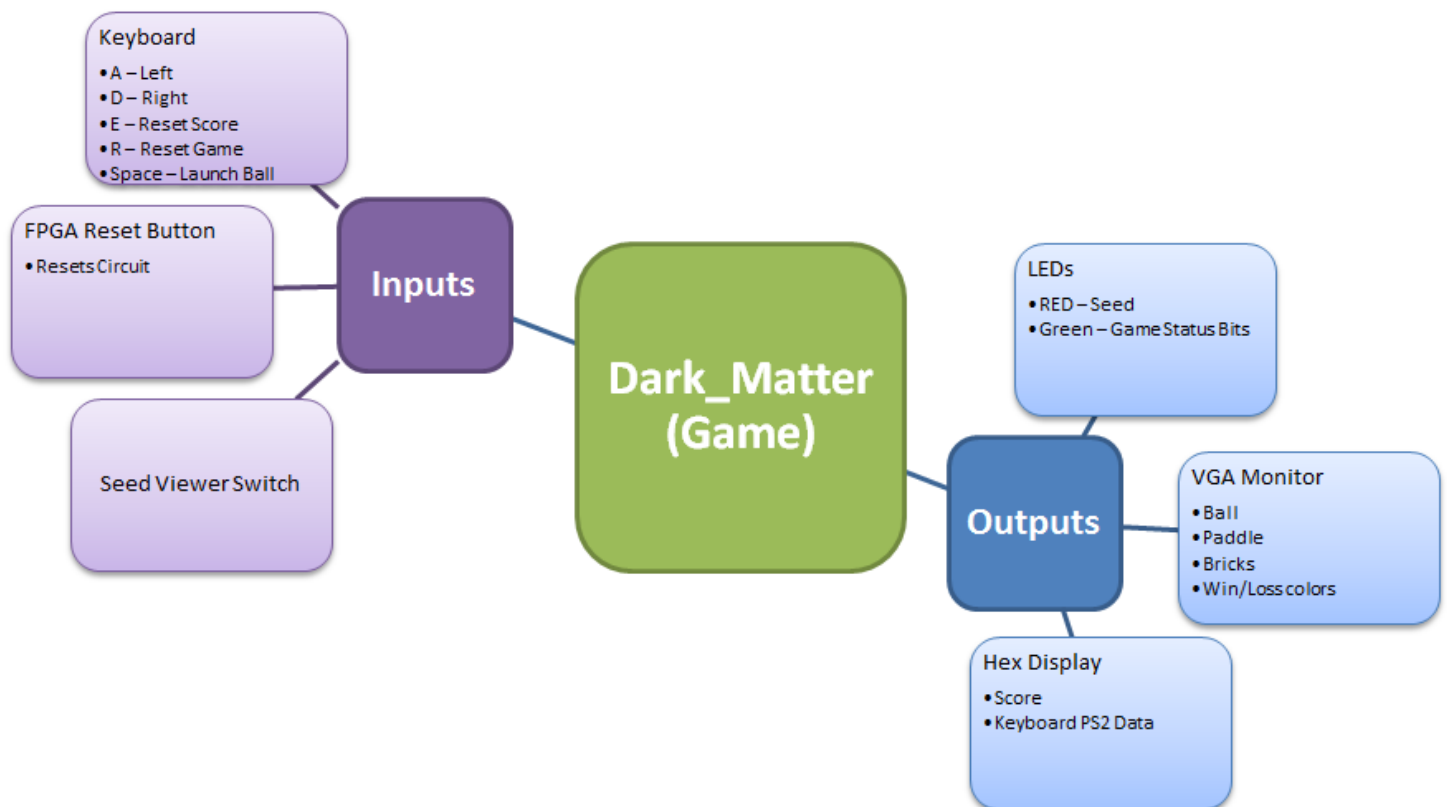


Figure 1. An overall view of our design

Introduction:

The ECE 385 final project provided an opportunity for us to demonstrate the skills, coding knowledge, and debugging methods we were taught throughout the semester. In order to do this, and also exercise creative freedom, we set out to make a creatively new rendition of brick breaker, which we called dark matter. The namesake is derived from plans to incorporate the sci-fi element of a moving particle of dark matter destroying other things in its path. Although the lore behind the idea was not really implemented, the product of our efforts is a feature filled “brick-breaker” game. Expanding upon the foundational understandings of our eighth lab, we added things including a paddle, bricks, scoring mechanism, and some random functionality. “Dark Matter” is a fun brick breaker game for all ages.

Note: it is suggested that you view the flow diagrams in the appendix for further introduction to our project.

Game/Circuit Operation:

This game is intended to be an intuitive user-experience, with the user only needing to interact with a PS2 keyboard and VGA-enabled screen. For robustness and back-up redundancy, the user can also manipulate certain things in the game via FPGA switches and buttons (although they must handle the paddle solely with the keyboard).

Similar to most computer games, movement is mapped in the W-A-S-D configuration. In this game, the paddle only moves laterally, so the ‘A’ key and ‘D’ control our paddle. It moves left when the ‘A’ key is pressed and right with the ‘D’ key is pressed. If the user loses a game, or simply wishes to play again, the user can reset the game and keep their score intact from previous sessions. A score-reset is mapped to the ‘E’ key and the game-reset mapped to a button on the FPGA board and the ‘R’ key on the PS2 keyboard. Due to latency in VGA monitors, the momentary buttons on the FPGA board provide more reliable feedback compared to PS2 input [thus the redundancy]. The games start with the ball resting on the paddle. To begin a session the user simply presses ‘space.’

The operation of our game is simple. The user has to reset the game using the appropriate keys and then moves the paddle to keep the ball from touching the ground. Ball interactions with the walls and paddle are randomized for added difficulty. This is achieved using a seed-generator of sorts. The seed generator provides alternating bits of various frequencies to choose from. The user can see what the seed is via on-board LEDs and can also turn off the LEDs by the flip of a switch. If the ball reaches the ground before all of the bricks are destroyed, the screen will turn red--indicating a loss--and the gameplay will stop. If the user successfully breaks all the bricks, the screen will turn green indicating a win. Score is indicated via two hex displays on the FPGA board. Overall, the above functionality allows for the user to easily enjoy a game of what is traditional “brick-breaker.”

Modular Operation:

Operation of our game consists of entities for each object the user interacts with (the ball, bricks, and paddle), the screen used (VGA controller, color mapper, hex driver), the keyboard input (D-ff, registers, keyboard processor, edge detector, and logic unit), and other incorporated entities (body mapper, counter, game handler, and random gen). For direct information, please refer to the code to be emailed online. The individual functionality and explanation of these functions is described below:

1. Ball

The ball class was a beefy class that had to update the position and motion of the ball as it interacted with the paddle and bricks. We had several large 'if'-statements that basically tracked where the ball was and if it ever intersected any part of the other components on the screen. If it did, the balls movement and position were updated to reflect a bounce in the appropriate behavior. Through this constant updating of the ball, we were able to achieve motion befitting of a ball in brick-breaker.

This class was also responsible for declaring when a loss was, in the event that the ball hits the bottom of the board. The game status signal is sent to the game handler from there. The randomization of any movement is coded in here as well.

2. Block

The block entity was mainly responsible for turning the brick off if the ball bounced it. It takes in the balls coordinates and the coordinates for that instance of the bricks. The brick entity is responsible for turning the brick off; all of the ball reaction to the brick is handled in the ball class. When a brick is hit, before disappearing it generates a status signal that it was hit. This is sent to the game_handler for LED display and score update.

3. Bouncing Ball

This entity had all of components together and tied all the necessary inputs together. It had 20 brick components that were all instances of block, as well as a ball, paddle, color mapper, game handler, and VGA controller. The entity connected all of these components together as needed to make the circuit function. There were many input signals including keyboard signals, generated seed, and reset buttons.

The win bit of the game_status signal was generated here. It checked if the brick vector indicated that all bricks were gone and generated a win bit accordingly. This bit was sent to the game_handler.

4. Color Mapper

The color mapper entity actually does the update of the colors on the VGA controller. Once it is told to run, it checks the passed pixel and paints it the appropriate color depending on

whether that pixel is at the location of the ball. It also paints the background, paddle, and all the bricks in the same manner.

5. Counter

This was a specially designed counter that output a single bit in order to generate a cut down clock frequency that the PS2 protocol could sync up with. It resulted in choosing the ninth bit of a counter from the original clock to form the `mod_clk` signal that drove our registers and keyboard input.

6. D FF

This entity was a standard Positive Edge Triggered D Flip-Flop. It specifically stored the value of the sign extension bit generated from the `add_sub9` unit. The D input was the most significant bit of the output of the `add_sub9` input and its load value was given to it from the control entity. The output of the D Flip Flop connected to the serial in value of Register A in the `register_unit` entity. Overall, the D Flip Flop handled the storing and passing of our sign extension bit.

7. Dark Matter

This was the top level entity that tied all the components and entities together to make the circuit work harmoniously. It was a block diagram schematic

8. Edge Detector

The edge detector entity's purpose and function was given to us in the lecture slides, we just had to actually design and write code that would execute it. It consisted of two D flip flop components in a master slave orientation that would help us find the rising clock edge and the falling clock edge of the `ps2clock`. We made the entity output two status bits that would indicate the status of the clock. The value we cared about was the "falling edge" which was when the status bits were set to "10". When this occurred, we allowed our keyboard processor to take in the data bits from the `ps2data` line. The other status bit values were representative of the following: clock is high ("11"), clock is low ("00"), and rising edge ("01").

9. Game Handler

The game handler entity updated the game status signal depending on the current state of the game. It helped stabilize the circuit when circuit win or lose condition occurred. It took in the current game status, assessed the state of the game, and output the status of the game to all of the entities that needed it, such as color mapper. The game handler also updated the score based on the status bits that were passed to it. The score is sent out to hex displays in standard-hexadecimal format.

Hex Driver

The HexDriver is an entity which translates the four bit input, which is a 4 bit unsigned binary value, into a 7 bit outputs, where each bit represents one segment in the seven segment displays on the FPGA board. The HexDriver just translates the values through one long select statement.

10. Keyboard Processor

The keyboard processor entity takes the code bit from the keyboard, as well as the keyboard clock status from the edge detector to determine the current codes in the keyboard. On the falling edge of the clock, the keyboard processor will pass the current bit from the key code to the register unit for storage. With the eleven consecutive ps2clk cycles, we will accomplish the task of storing the most recent key code in the SR register.

11. Logic Unit

The logic unit takes the data stored in the SR and PR registers and decides which way the ball moves. As long as PR doesn't have xF0, the circuit will look at SR and update the four directional bits. If the 'w' make code is in SR, then the UP signal will be on. If the 'a' make code is in SR, then the LEFT signal will be on. If the 's' make code is in SR, then the DOWN signal will be on. If the 'd' make code is in SR, then the RIGHT signal will be on. Any other code will result in the four output bits not being set. In addition, we also mapped signals for certain other key presses such as 'R', 'E' and the numbers from 1 to 9. This allowed us the flexibility of using more keys when we saw fit, such as when we used the 'R' key to reset the game.

12. Paddle

The paddle entity was responsible for updating the paddle's position and motion per the keyboard inputs. It took in signals from the logic unit, telling it to move left or right and then it reacted accordingly. It output two vectors indicating the paddle's x and y coordinates.

13. Reg 11

The Reg_11 entity simulates an eleven bit register. It takes in a Serial shift in value, a Load value, a Shift enable bit, a Clk input, and a reset bit. It implements a normally functioning eight bit register using a series of if and else statements. The functionality is that the register is cleared when reset is '1' and the register is set to another 11 bit D vector when load is high. When shift is high the register executes a logical shift right using the Shift_in bit as the logical shift in.

15. Reg Unit

The register unit consisted of two reg_11 components whose ports were mapped in such a way that they were wired in serial as one large continuous register. The least significant bit of Register SR was the serial in of Register PR so when they were shifted they were shifted together. The serial in of Register SR came from the Keyboard Processor entity. They shared shift enable signals but had independent load signals. The register unit just consolidated the two registers into one entity for better decomposition of the circuit. After each set of ps2clk cycles, the SR register would hold the most recent key code and the PR register would hold the previous key code.

16. VGA Controller

This entity was given to us and it drives the VGA output to the screen. It takes in the balls position and translates it to the pixels on screen. It updates the necessary values so that color mapper can actually decide what colors to refresh at the correct time. The VGA controller controls when the refreshing of the screen occurs.

17. Random_Gen

This component was the basis for our randomization concept. The idea was to have conditional statements that depended on one bit. To do this, we essentially took the clock and made a 34-bit counter and used bits in the 15 most significant bits to use for randomization. The bits were outputted for display to red LEDs on the board. The output of this circuit included a seventeen bit 'seed.' This was the signal sent around the circuit to choose something pseudo-random from.

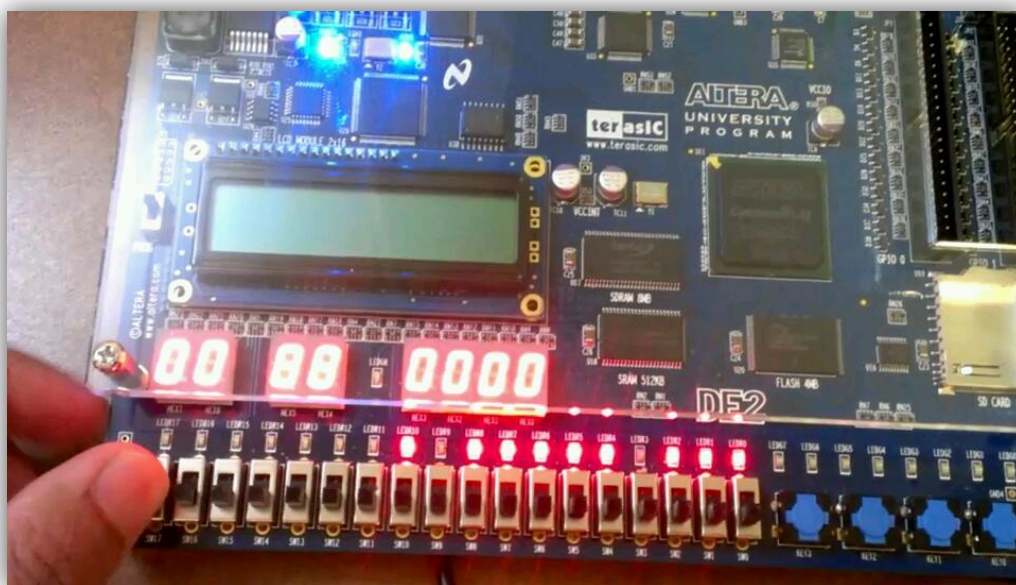


Figure 2. The random seed generation in action (bottom row of LEDs)

Design and Work-Method:

In this project, we were encouraged to work diligently, consistently, and feature-by-feature. The following is a broad account of this process:

Phase 1: Paddle and I/O

Since we wanted to implement a game with the VGA output and PS2 Keyboard input, we recycled and modified code from lab 8 to get basic functionality. However, it is noted that a vast majority of this code was modified and adapted to meet our needs, as we demanded much more than moving a simple ball around a screen.

Phase 1 consisted of us mapping more of the keys for future use. We then undertook the task of adding a paddle to the board. We started by just adding a block in the color mapper to represent the paddle. Once we got the paddle block to appear at a certain paddleX coordinate and paddleY coordinate, we designed the paddle class. The paddle class would output the paddle's coordinates to the color mapper and the ball. By changing the paddle coordinates constantly in the paddle class, we were able to have a moving paddle whose location on the screen would be reflected by the color mapper. We also passed the paddle coordinates to the ball entity which allowed us to hard code interactions between the paddle and the ball. We made the ball "bounce" off the paddle by changing directions of motion every time the ball's coordinates were within the paddle's bounds. The paddle itself had to respond to keyboard inputs and that was handled in the paddle class. Probably the largest and most frustrating this to handle with the paddle was weird edge cases that would result in the paddle disappearing off the screen or clipping. There was even a time when the paddle would disappear at a certain point on the right half of the screen. These were all fixed with rigid if statements that accounted for these off edge cases and an expansion of the vector size for the DrawX and DrawY signals that were signed and thus limited the output of the display.

Finally, we took this time to code in our first game end condition which was loss. If the ball missed the paddle, and hit the bottom of the screen, it would not bounce but rather turn the screen red informing the user that they have lost. We also implemented reset functionality with the 'R' key on the PS2 keyboard to make it easier for the user to reset. This was what we took care of in phase 1 of our project.

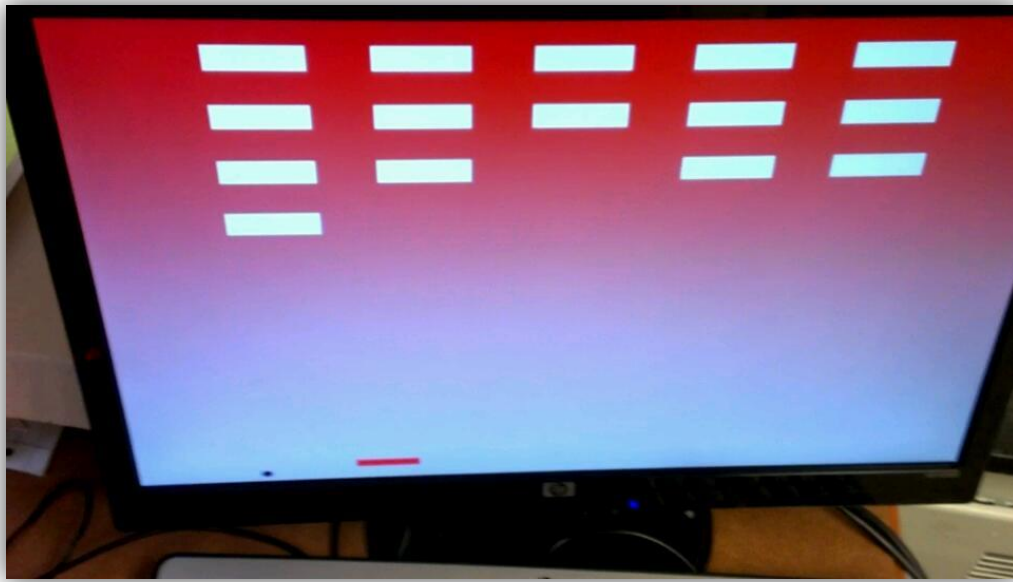


Figure 3a. Player lost

Phase 2: Single Brick

Phase two was undertaken with utmost caution. We had to now implement bricks into our circuit and we wanted to make it as easy as possible to add bricks. We designed our brick class to toggle the brick on or off. This was done by changing a brick-on signal that the block entity was outputting. The entity also took in the ball coordinates as well as the coordinates for that brick. The bouncing off the brick was added to ball entity. This was a design choice that we favored because it kept all things changing the motion of the ball within the ball class, thus preventing multiple entities from driving the same signal. We then passed the brick coordinates to the color mapper to make the brick appear on the screen. We took great care to make sure that the circuit worked properly for one brick. The process of adding multiple bricks was just a matter of creating more instances of the block entity and making a large vector to hold all of the signals needed.

Phase 3: Multiple Bricks

Phase three expanded from one brick to twenty bricks. We added nineteen more instances of the block class in the bouncing ball entity to allow for more bricks for the user to hit. We also increased the bus size from 11 bits for each coordinate to 220 bits. We made this design choice so we weren't having 20 different signals for the x coordinates of all of the bricks. Thus we had a 220 bit bus for our x coordinates, a 220 bit bus for our y coordinate, and a 20 bit bus for our brick on signals. We passed the appropriate parts of the bus to each bus and our circuit worked very well for each brick interaction. We also had to edit the ball entity by adding numerous if statements to make the ball bounce off each brick. This phase concluded when we were certain we could manipulate the vectors at will, removing and resetting bricks as appropriate.

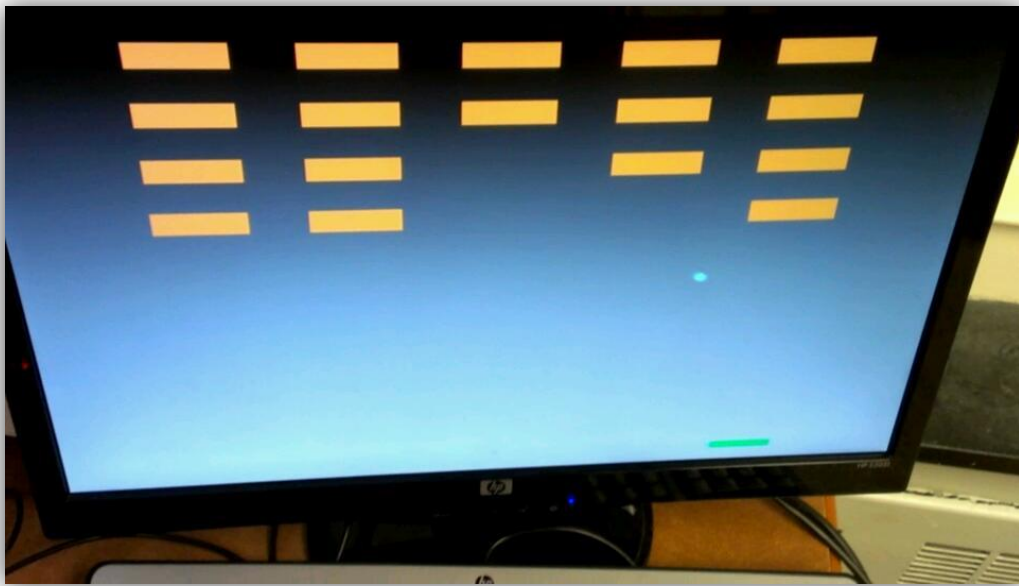


Figure 3b. The game in action

Phase 4: Proper interactions

At the start of this phase, we had a fully functional, although basic, version of our game. However, at this level and with the time we had, we wanted to do more.

To add the notion of anything random to our circuit, we designed a random seed generator. We wanted to be able to select a seed-bit and check if it was a logical high or low (and thus make conditional statements based on this for the purpose of randomization). To do this, we essentially made a sophisticated counter. We divided the clock into 36 bits and only used the most significant half for our seed. This vector was mapped to the on-board red LEDs. As these are a passive visual, we added a switch that would turn on and off the LED display, yet keep the generation going. The only thing mapped to reset the seed is the master reset on the board itself.

Once the random seed was able to be generated, it was circulated in the design. It was used to randomize the initial velocities the ball would have upon being launched with the spacebar. Therefore, the user would not know if the ball would start left or right. Another randomization was done with ball movement, with regard to the general interactions it had with other non-brick surfaces. The response would be random, thus making trajectories impossible to completely predict.

To further add user interface with the game and increase complexity, we added the functionality that the user could control which way the ball rebounded off the paddle based on the movement direction it was in. If the user wanted to move the ball right, then the paddle would be moved right at the same time.

At this stage, we were setting the infrastructure for score, timing, and other state-functions. This process included setting up LEDs that responded to status bits that funnels into

the game-handler. These status bits include winning indication, brick hits, paddle hits, and a loss. These are displayed in the green LEDs on the board.

Finally, we wanted the game to be more dynamic with the user in complete control of new games. With this, we added that the spacebar would initiate a start. Upon hitting reset, the ball is placed on the paddle, waiting to be launched. Once the spacebar is hit, it randomly starts its course.

Phase 5: End conditions (win) and scores

Here lies our last phase of work. Using the status bits previously set up, we implemented a score function. As seen at the beginning of the paper, points are awarded for staying in the game. The point values are also explained at the beginning. Finally, once all bricks are gone, the user receives immediate feedback of a happy green screen indicating a win, accompanied by a flashing/cycling score display. Note that the score is not displayed in decimal but in hexadecimal.

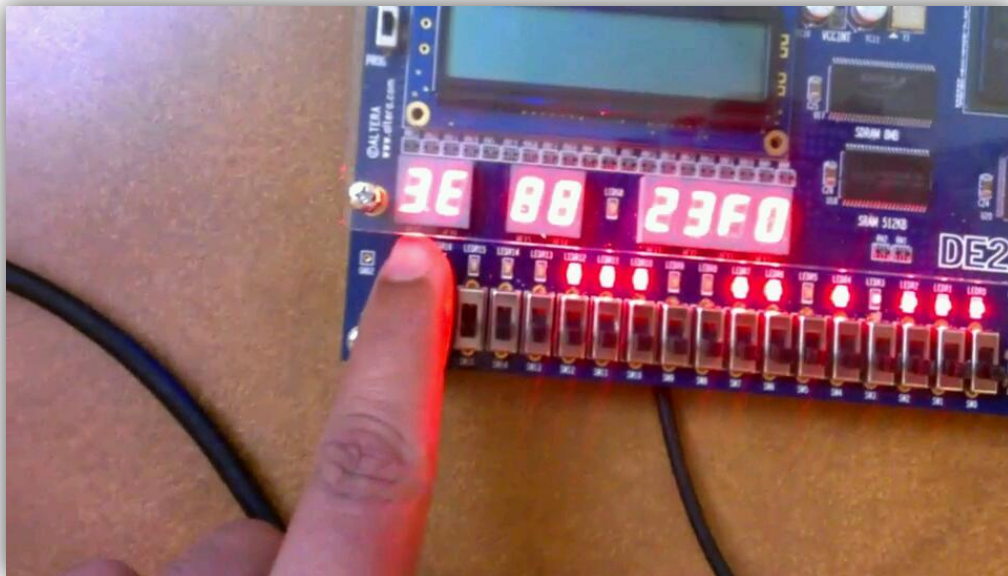


Figure 4. Pointing out the score hex display

Block Diagram:

Appended are several diagrams annotating the design and implementation of our game. Please see appendix.

State Diagram and Simulation:

Not applicable. Our circuit was largely something to be demonstrated, rather than data manipulation. There were many cases where numbers came out correctly, yet balls misbehaved, thus the lack of usefulness waveforms are. This is acceptable and expected depending on the nature of projects, as previous announcements have iterated.

Timing and Synthesis:

We used a total of 1,161 logic elements and 185 registers. This was more than any previous circuit coded in VHDL in the structured portion of the course. The maximum frequency of our circuit was 87.30 MHz.

Increasing Frequency:

Some ways we could have made this a faster process would be to optimize our signals within our entities. This would demand fewer registers and a shorter average data path. Also, we could have implemented bricks using an algorithm rather than hardcoding the regions of occupation individually. This algorithm would reference a vector that would give the information to populate the board with enough bricks, yet not call on multiple lines of code at the same time.

Reducing Hardware Usage:

To simplify our circuit we could have implemented previously said modifications. Furthermore, we may have been able to optimize overall functionality, instead of being so modular, as a single, large state machine. This is not a thoroughly investigated idea, so it is less certain.

Note: Zipped files sent on schedule

Conclusion:

This was a fun, rewarding, time consuming, and frustrating project. We fulfilled lab 8 when it occurred, and we also were well equipped with knowledge on how to utilize keyboard input and VGA displays. With these tools we set out to make a functional game, a goal we achieved.

We would have wanted to make a more complex game, however. We set up our work on a tiered system of goals, moving from one tier to the next [more complex] tier as time passed. As we worked, we had the mindset that we wanted to lay the foundation for future functionality—thus we coded accordingly. For instance, we designed the bricks to be handled largely on a set of vector signals sent throughout the circuit. Had we had the opportunity to, we would have taken the brick information hard coded in some of the entities and enclosed it in a level handler entity. We would have used the number keys which were numbered and ready to go to set what level to launch.

Other things that were set up for future functionality include other mapped keys such as “-“ and “+” which were intended to decrease/increase difficulty within any level. This functionality was just short of being implemented. Also, the four bit game status vector was used on a bit-by-bit basis for now. This four bit capacity could be used to relay other status factors such as time thresholds and power-ups. The following elaborates on further intentions and goals past the “lower tier” of already accomplished goals:

Medium Tier:

- The ball is currently respected as a rectangle in brick/paddle to ball interactions. This causes some reactions/responses to be rough and/or unexpected.
- Timer
- New levels

Upper Tier:

- Dynamic game interactions
 - Power-ups
 - Timer driven brick movement
 - Random color mode
 - Add some vertical movement to paddle
 - Collision animation
 - Start/end splash screens via utilization of sprites

While we did not implement all we wanted to, we did have a working – medium complexity – game. We did face some challenges along the way also.

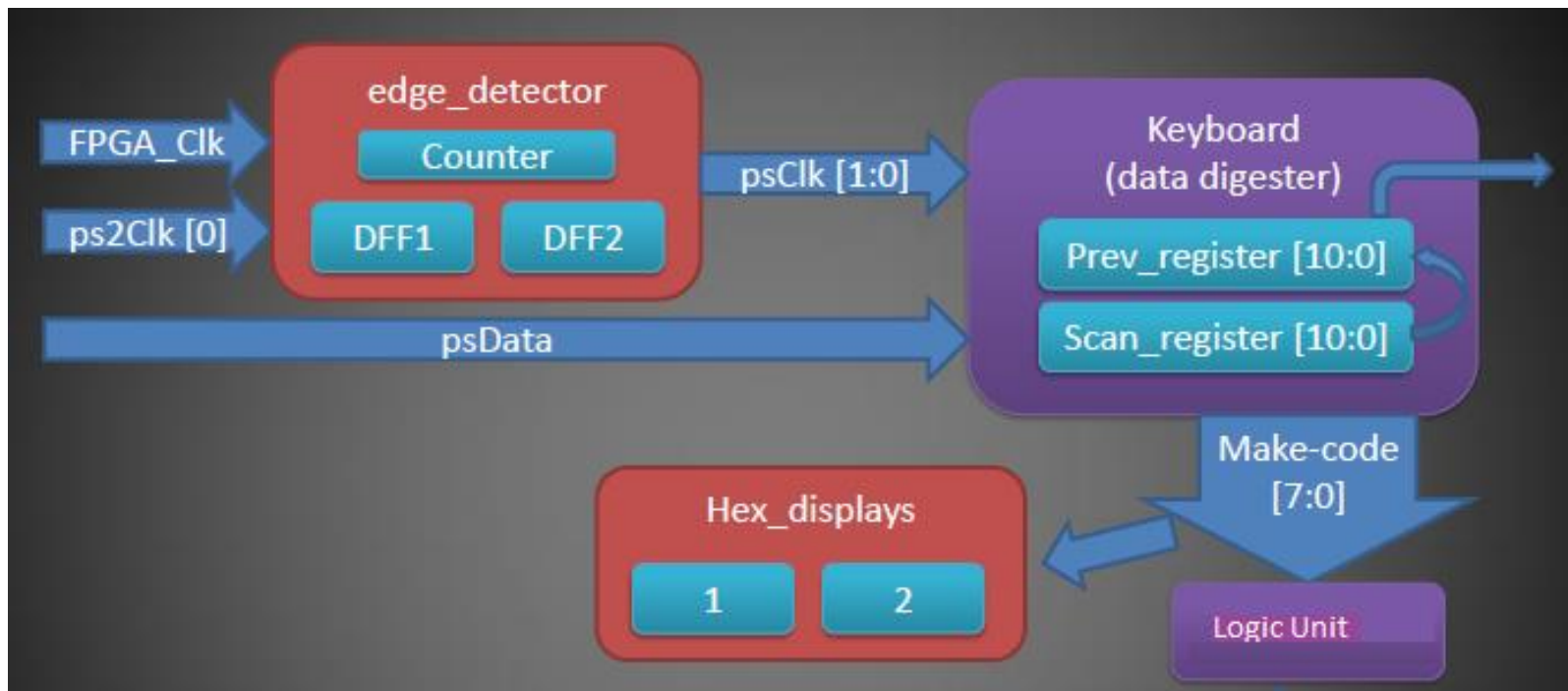
Sometimes we felt like we encountered what is expected to be hardware bugs of sorts. Not in the sense that the board is bad, but there are signals with high latency that sometimes

cause problems. We also experienced behavior that was not consistent without changes in code. Sometimes the break-code for our PS2 data was F8 instead of F0. This was unpredictable and we were only able to react to it when it occurred. Also, the circuit would refresh the screen on non-restart key presses sometimes. We spent several hours looking for the code we changed that caused the display to not work properly and never solved it. However, it solved itself somehow and began working again. We credit such behavior to timing issues that we have not pinned down in order to know how to fix.

Overall, our circuit demonstrated full functionality and we gained a better understanding of how to implement certain tasks using the FPGA board.

Appendix:





















































The attached pages include the flow charts, block-diagram, etc.



Keyboard interface

1		blank	Output	PIN_D6	3	B3_N1	3.3-V LVTTTL (default)
2		Blue[9]	Output	PIN_B12	3	B3_N0	3.3-V LVTTTL (default)
3		Blue[8]	Output	PIN_C12	3	B3_N0	3.3-V LVTTTL (default)
4		Blue[7]	Output	PIN_B11	3	B3_N0	3.3-V LVTTTL (default)
5		Blue[6]	Output	PIN_C11	3	B3_N0	3.3-V LVTTTL (default)
6		Blue[5]	Output	PIN_J11	3	B3_N0	3.3-V LVTTTL (default)
7		Blue[4]	Output	PIN_J10	3	B3_N0	3.3-V LVTTTL (default)
8		Blue[3]	Output	PIN_G12	3	B3_N0	3.3-V LVTTTL (default)
9		Blue[2]	Output	PIN_F12	3	B3_N0	3.3-V LVTTTL (default)
10		Blue[1]	Output	PIN_J14	3	B3_N0	3.3-V LVTTTL (default)
11		Blue[0]	Output	PIN_J13	3	B3_N0	3.3-V LVTTTL (default)
12		Clk	Input	PIN_N2	2	B2_N1	3.3-V LVTTTL (default)
13		game_status[3]	Output	PIN_U18	7	B7_N0	3.3-V LVTTTL (default)
14		game_status[2]	Output	PIN_V18	7	B7_N0	3.3-V LVTTTL (default)
15		game_status[1]	Output	PIN_W19	7	B7_N0	3.3-V LVTTTL (default)
16		game_status[0]	Output	PIN_AF22	7	B7_N0	3.3-V LVTTTL (default)
17		Green[9]	Output	PIN_D12	3	B3_N0	3.3-V LVTTTL (default)
18		Green[8]	Output	PIN_E12	3	B3_N0	3.3-V LVTTTL (default)
19		Green[7]	Output	PIN_D11	3	B3_N0	3.3-V LVTTTL (default)
20		Green[6]	Output	PIN_G11	3	B3_N0	3.3-V LVTTTL (default)
21		Green[5]	Output	PIN_A10	3	B3_N0	3.3-V LVTTTL (default)
22		Green[4]	Output	PIN_B10	3	B3_N0	3.3-V LVTTTL (default)
23		Green[3]	Output	PIN_D10	3	B3_N0	3.3-V LVTTTL (default)
24		Green[2]	Output	PIN_C10	3	B3_N0	3.3-V LVTTTL (default)
25		Green[1]	Output	PIN_A9	3	B3_N0	3.3-V LVTTTL (default)
26		Green[0]	Output	PIN_B9	3	B3_N0	3.3-V LVTTTL (default)
27		hs	Output	PIN_A7	3	B3_N1	3.3-V LVTTTL (default)
28		LedG0	Output	PIN_AE22	7	B7_N0	3.3-V LVTTTL (default)
29		PR_hexL[6]	Output	PIN_V13	8	B8_N0	3.3-V LVTTTL (default)
30		PR_hexL[5]	Output	PIN_V14	8	B8_N0	3.3-V LVTTTL (default)
31		PR_hexL[4]	Output	PIN_AE11	8	B8_N0	3.3-V LVTTTL (default)
32		PR_hexL[3]	Output	PIN_AD11	8	B8_N0	3.3-V LVTTTL (default)
33		PR_hexL[2]	Output	PIN_AC12	8	B8_N0	3.3-V LVTTTL (default)
34		PR_hexL[1]	Output	PIN_AB12	8	B8_N0	3.3-V LVTTTL (default)
35		PR_hexL[0]	Output	PIN_AD10	8	B8_N0	3.3-V LVTTTL (default)
36		PR_hexU[6]	Output	PIN_AB24	6	B6_N1	3.3-V LVTTTL (default)
37		PR_hexU[5]	Output	PIN_AA23	6	B6_N1	3.3-V LVTTTL (default)
38		PR_hexU[4]	Output	PIN_AA24	6	B6_N1	3.3-V LVTTTL (default)
39		PR_hexU[3]	Output	PIN_Y22	6	B6_N1	3.3-V LVTTTL (default)
40		PR_hexU[2]	Output	PIN_W21	6	B6_N1	3.3-V LVTTTL (default)
41		PR_hexU[1]	Output	PIN_V21	6	B6_N1	3.3-V LVTTTL (default)
42		PR_hexU[0]	Output	PIN_V20	6	B6_N1	3.3-V LVTTTL (default)
43		ps2clk	Input	PIN_D26	5	B5_N0	3.3-V LVTTTL (default)
44		ps2data	Input	PIN_C24	5	B5_N0	3.3-V LVTTTL (default)
45		Red[9]	Output	PIN_E10	3	B3_N0	3.3-V LVTTTL (default)
46		Red[8]	Output	PIN_F11	3	B3_N0	3.3-V LVTTTL (default)
47		Red[7]	Output	PIN_H12	3	B3_N0	3.3-V LVTTTL (default)
48		Red[6]	Output	PIN_H11	3	B3_N0	3.3-V LVTTTL (default)
49		Red[5]	Output	PIN_A8	3	B3_N0	3.3-V LVTTTL (default)
50		Red[4]	Output	PIN_C9	3	B3_N1	3.3-V LVTTTL (default)
51		Red[3]	Output	PIN_D9	3	B3_N1	3.3-V LVTTTL (default)
52		Red[2]	Output	PIN_G10	3	B3_N1	3.3-V LVTTTL (default)
53		Red[1]	Output	PIN_F10	3	B3_N1	3.3-V LVTTTL (default)

Pin Assignments (1 of 2)

54		Red[0]	Output	PIN_C8	3	B3_N1	3.3-V LVTTTL (default)
55		reset	Input	PIN_G26	5	B5_N0	3.3-V LVTTTL (default)
56		scoreH[6]	Output	PIN_N9	2	B2_N1	3.3-V LVTTTL (default)
57		scoreH[5]	Output	PIN_P9	2	B2_N1	3.3-V LVTTTL (default)
58		scoreH[4]	Output	PIN_L7	2	B2_N1	3.3-V LVTTTL (default)
59		scoreH[3]	Output	PIN_L6	2	B2_N1	3.3-V LVTTTL (default)
60		scoreH[2]	Output	PIN_L9	2	B2_N1	3.3-V LVTTTL (default)
61		scoreH[1]	Output	PIN_L2	2	B2_N1	3.3-V LVTTTL (default)
62		scoreH[0]	Output	PIN_L3	2	B2_N1	3.3-V LVTTTL (default)
63		scoreL[6]	Output	PIN_M4	2	B2_N1	3.3-V LVTTTL (default)
64		scoreL[5]	Output	PIN_M5	2	B2_N1	3.3-V LVTTTL (default)
65		scoreL[4]	Output	PIN_M3	2	B2_N1	3.3-V LVTTTL (default)
66		scoreL[3]	Output	PIN_M2	2	B2_N1	3.3-V LVTTTL (default)
67		scoreL[2]	Output	PIN_P3	1	B1_N0	3.3-V LVTTTL (default)
68		scoreL[1]	Output	PIN_P4	1	B1_N0	3.3-V LVTTTL (default)
69		scoreL[0]	Output	PIN_R2	1	B1_N0	3.3-V LVTTTL (default)
70		Seed[17]	Output	PIN_AD12	8	B8_N0	3.3-V LVTTTL (default)
71		Seed[16]	Output	PIN_AE12	8	B8_N0	3.3-V LVTTTL (default)
72		Seed[15]	Output	PIN_AE13	8	B8_N0	3.3-V LVTTTL (default)
73		Seed[14]	Output	PIN_AF13	8	B8_N0	3.3-V LVTTTL (default)
74		Seed[13]	Output	PIN_AE15	7	B7_N1	3.3-V LVTTTL (default)
75		Seed[12]	Output	PIN_AD15	7	B7_N1	3.3-V LVTTTL (default)
76		Seed[11]	Output	PIN_AC14	7	B7_N1	3.3-V LVTTTL (default)
77		Seed[10]	Output	PIN_AA13	7	B7_N1	3.3-V LVTTTL (default)
78		Seed[9]	Output	PIN_Y13	7	B7_N1	3.3-V LVTTTL (default)
79		Seed[8]	Output	PIN_AA14	7	B7_N1	3.3-V LVTTTL (default)
80		Seed[7]	Output	PIN_AC21	7	B7_N0	3.3-V LVTTTL (default)
81		Seed[6]	Output	PIN_AD21	7	B7_N0	3.3-V LVTTTL (default)
82		Seed[5]	Output	PIN_AD23	7	B7_N0	3.3-V LVTTTL (default)
83		Seed[4]	Output	PIN_AD22	7	B7_N0	3.3-V LVTTTL (default)
84		Seed[3]	Output	PIN_AC22	7	B7_N0	3.3-V LVTTTL (default)
85		Seed[2]	Output	PIN_AB21	7	B7_N0	3.3-V LVTTTL (default)
86		Seed[1]	Output	PIN_AF23	7	B7_N0	3.3-V LVTTTL (default)
87		Seed[0]	Output	PIN_AE23	7	B7_N0	3.3-V LVTTTL (default)
88		SR_hexL[6]	Output	PIN_Y24	6	B6_N1	3.3-V LVTTTL (default)
89		SR_hexL[5]	Output	PIN_AB25	6	B6_N1	3.3-V LVTTTL (default)
90		SR_hexL[4]	Output	PIN_AB26	6	B6_N1	3.3-V LVTTTL (default)
91		SR_hexL[3]	Output	PIN_AC26	6	B6_N1	3.3-V LVTTTL (default)
92		SR_hexL[2]	Output	PIN_AC25	6	B6_N1	3.3-V LVTTTL (default)
93		SR_hexL[1]	Output	PIN_V22	6	B6_N1	3.3-V LVTTTL (default)
94		SR_hexL[0]	Output	PIN_AB23	6	B6_N1	3.3-V LVTTTL (default)
95		SR_hexU[6]	Output	PIN_W24	6	B6_N1	3.3-V LVTTTL (default)
96		SR_hexU[5]	Output	PIN_U22	6	B6_N1	3.3-V LVTTTL (default)
97		SR_hexU[4]	Output	PIN_Y25	6	B6_N1	3.3-V LVTTTL (default)
98		SR_hexU[3]	Output	PIN_Y26	6	B6_N1	3.3-V LVTTTL (default)
99		SR_hexU[2]	Output	PIN_AA26	6	B6_N1	3.3-V LVTTTL (default)
100		SR_hexU[1]	Output	PIN_AA25	6	B6_N1	3.3-V LVTTTL (default)
101		SR_hexU[0]	Output	PIN_Y23	6	B6_N1	3.3-V LVTTTL (default)
102		sync	Output	PIN_B7	3	B3_N1	3.3-V LVTTTL (default)
103		Toggle_SW17	Input	PIN_V2	1	B1_N0	3.3-V LVTTTL (default)
104		VGA_clk	Output	PIN_B8	3	B3_N0	3.3-V LVTTTL (default)
105		vs	Output	PIN_D8	3	B3_N1	3.3-V LVTTTL (default)
106		<new node>>					

Pin assignments (2 of 2)

