Final Project Report

“Mario Dodgeball”

Vincent Crabtree and Daniel Kobold

1. Project Plan:   
   The goal of this project is to implement an interactive game of “dodgeball” using sprites (implemented in SDK) and using the Zedboard buttons (hardware) to control the player sprite. The player will be represented by a sprite of Mario, and the objects to be avoided will be Green Turtle Shells (“Green Shells”), all from Nintendo’s “Mario Bros.” games. The game will involve pushing the directional buttons on the Zedboard to help Mario avoid collision with Green Shells. The Green Shells will be independent sprites that move around the screen, bouncing off of the walls and off of other shells. If they collide with Mario, the game stops and the Game Over message appears. Throughout the game, a score appears in the lower right hand corner of the screen and increases throughout the game. As the score gets higher, the speed of the Green Shells increases, making the game more difficult.  
   The basis of the entire project was Lab 7 from ECE 42100, but many elements of the lab were changed to complete the project. Those will be detailed below.  
   The motions of Mario will be mapped to the four direction buttons on the Zedboard hardware.  
   The sprites will be generated by finding simple images of Mario and a Green Shell, and will be converted to C header files which contain the color information for each pixel of the sprite. The information was originally in Hexadecimal, which we converted to integer values 0-7, using the RGB color system. This limited the colors, but the character and items are still easily recognizable. Each of the eight Green Shells operate like the bouncing block from the ECE 42100 labs, but also bounce off of each other. Mario can move anywhere on the screen, but the instant his block collides with a Green Shell, the game ends. The score in the bottom right hand corner is completed using the “SomeSimpleBitMapFont.c” file from the course files. The number is incremented by cycle and goes up very rapidly. The program is capable of filling up the entire bottom row of the screen with score numbers, but the Green Shells speed up fast enough that using the entire space will probably not be necessary. By looping through the positions of the shells and Mario and making sure they do not overlap, we were able to make sure that collisions are handled correctly.  
   1. Work Statement:  
      System Specifications: Xilinx Vivado 2018.2 and Zedboard  
      Tasks:
      1. Sprites including Mario and Green Turtle Shells
      2. Bouncing behavior of Green Turtle Shells
      3. Control over Mario using the Push Buttons on the Zedboard
      4. Timer/Score-keeping Mechanism
      5. Game Over Screen with Score display
      6. Hardware Design and Integration
      7. ~~(Optional) Start Screen with Difficulty Levels~~
      8. (Optional) Number of Shells
      9. ~~(Optional) Bowser as an Obstacle following the Player~~
      10. ~~(Optional) Computer Player with Difficulty Levels~~

Additional Tasks:

* + 1. Testing
       1. Test one element/feature at a time as they are added
    2. Documentation
       1. Comment in files during programming and document project after development concludes
  1. Project Schedule:

|  |  |  |
| --- | --- | --- |
| **Milestone** | **Daniel Kobold** | **Vincent Crabtree** |
| Graphics  Deadline: 11/16/18 | Mario and Green Shell Sprites  *Completed 11/16/18* | |
| Gameplay  Deadline: 11/23/18 | Bouncing Shells  *Completed 11/19/18* | User Input Movement  *Completed 11/20/18* |
| Mechanics  Deadline: 11/30/18 | Timer / Score  *Completed 11/20/18* | Game Over Screen  *Completed 11/20/18* |
| Documentation  Deadline: 12/5/18 | Documentation  *Completed 12/03/18* | |

Table Project Schedule

* 1. Resource Requirements:  
     Group Members: Two  
     Additional Hardware: Computer Monitor, USB Cables
  2. Previous Work in Area:  
     ECE282 Daniel – Multiplayer Snake  
     ECE282 Vincent – Multiplayer Pong  
     ECE 362 Both – Microprocessor User Input

1. Work Accomplished:
   1. Daniel:
      1. Mario and Green Shell Sprites
         1. Initially, the hardware was kept largely the same as Lab 7 to make sure the sprites were displaying properly. We selected basic images for Mario and the Green Turtle Shell and saved them onto the computer. These were found on Google Images. All Mario characters and objects are property of Nintendo and no copyright infringement is intended. The original images can be seen below.  
             (cited in Appendix)

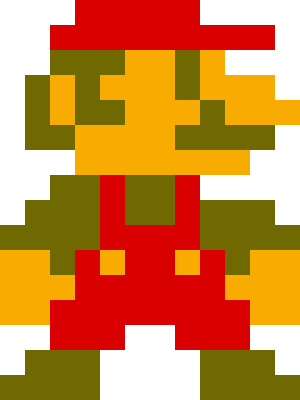
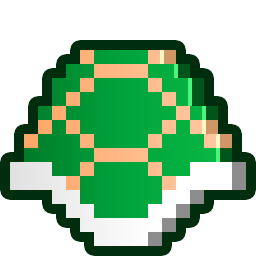
 

Figure Original Mario Image Figure Original Green Turtle Shell Image

* + - 1. Next, we plugged the two files into multiple website tools, in an effort to convert the image to a header file. The tool we ultimately used is linked below. The header array that was given had to be modified because the images included more colors than the basic RRRGGGBB colors we had available, so each color code was converted by hand and the nearest color was selected. Green replaced black because the background of the game is black and would make the shapes harder to understand. Having replaced all of the pixels (Control-F Replace All was very useful) with an integer value 0-7, the arrays were then added to the header files in software. The converted file can be seen in Figure 4. We tried the MATLAB tools made available on the course site, but found the below tool to be the best for conversion.  
         <http://www.digole.com/tools/PicturetoC_Hex_converter.php>

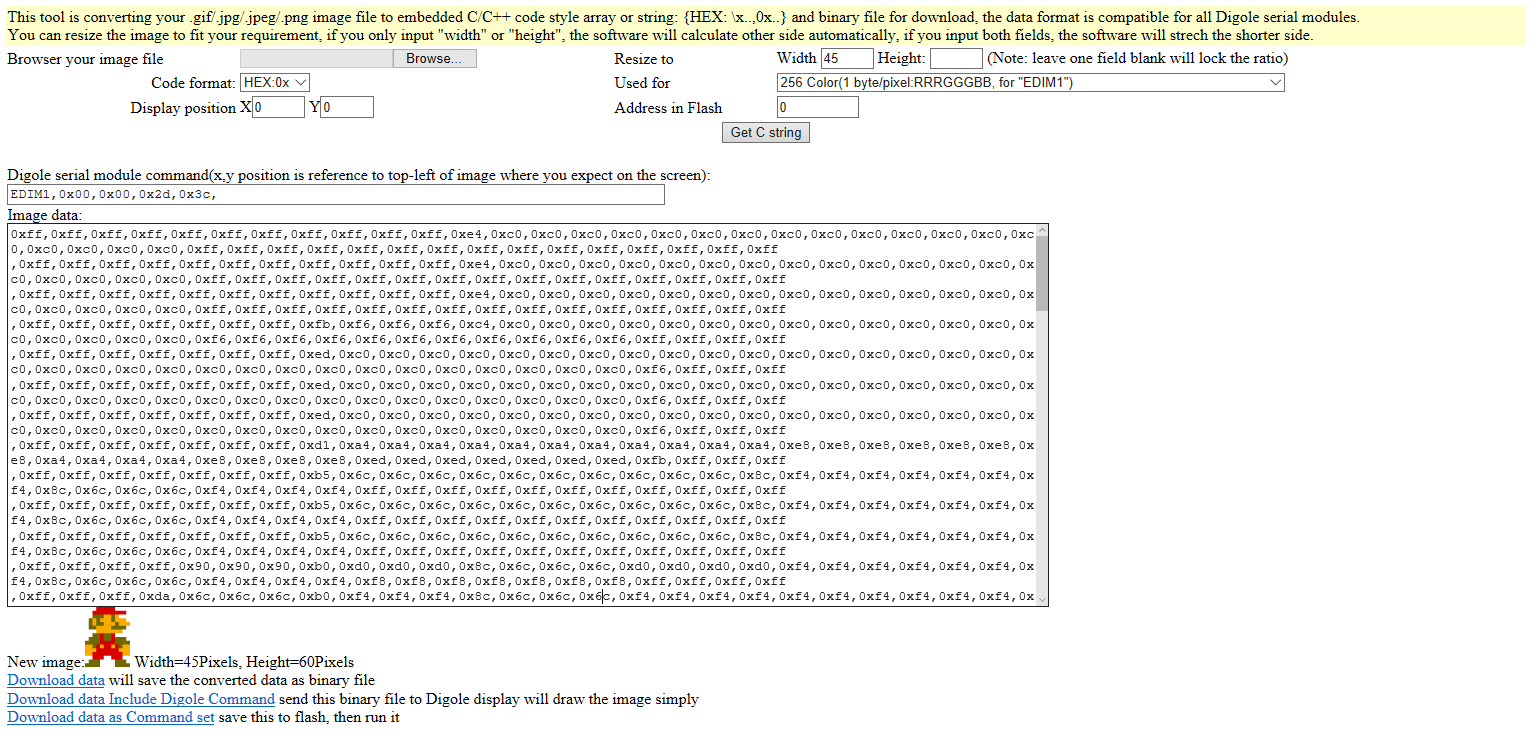
When using this tool, type in the width and height and change the “Used for” field to “256 Color” (as shown below in Figure 3)  


Figure Conversion of Image to C Array

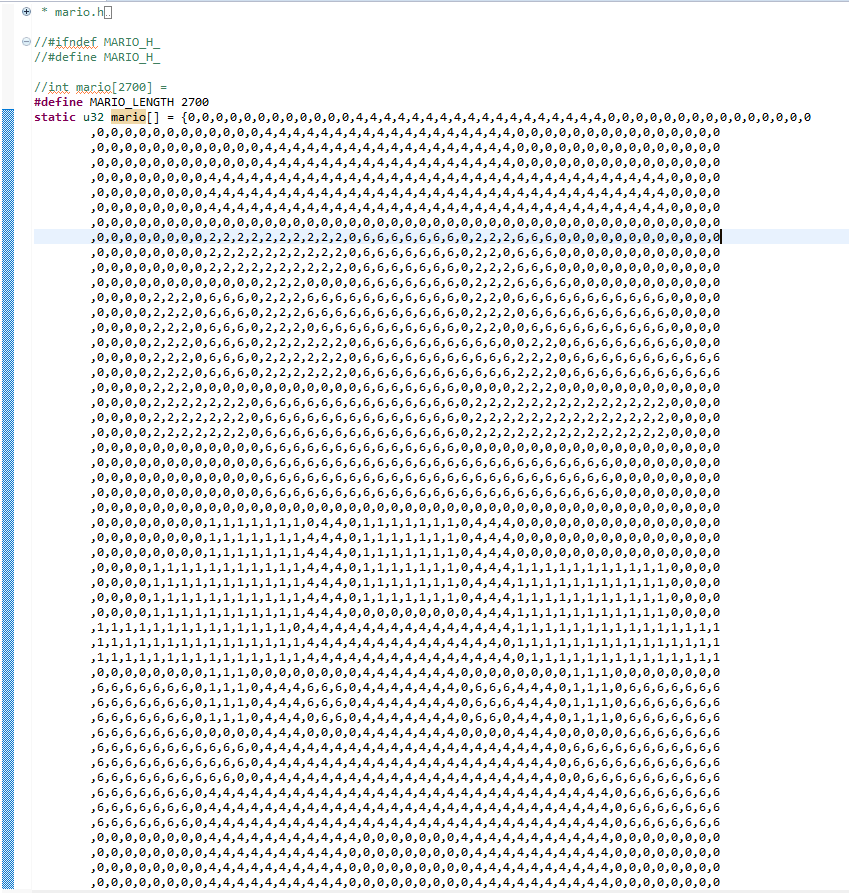


Figure Header File with Hex Values Replaced by Integer Values: If you look closely, you can see the shape of Mario

* + - 1. The next task was making Mario appear on the computer screen by using “Debug Configurations” in SDK. The first step was making a block of the correct dimensions appear on the screen. By adjusting the Lab 7 code, the correct-sized block appeared, but flashed random colors. When the block approached the top of the screen, the images of Mario and the Green Shell appeared, but as a full row of the image, repeated one after another.
      2. In order to make the sprites appear in the correct location on the screen, the code for selecting the color was adjusted. The basic form of set\_pix from Lab 7 was used, but the color was calculated by taking the pixel being printed and subtracting the X and Y positions of the Mario image, so the sprite would appear completely, and in the correct place (See Figure 5). Finally, set\_pix from Lab 7 was used to set the individual pixel on the screen to the appropriate color. A nested loop is used to make sure the entire image prints onto the screen.

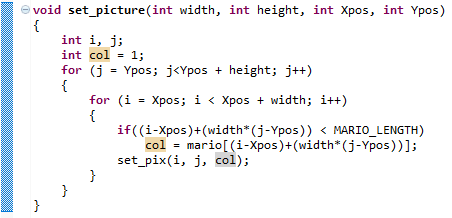


Figure Set Picture

* + - 1. This process (Steps 2-4) was repeated for the Green Shells, with a new function called set\_shell which is basically the version of set\_picture made for the Green Shell image (See Figure 6).

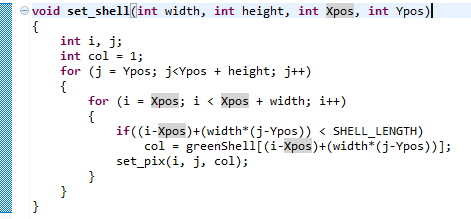


Figure Set Shell (Set Picture for Green Shells)

* + - 1. At this point, a Mario sprite was bouncing around the screen. The Mario sprite can be seen in Figure 7. The Green Shell sprite can be seen in Figure 8.

Figure Mario Sprite Figure Green Shell Sprite

* + - 1. Next, stationary shells were added to the screen. The existence of a shell was represented by an array known as “s”. If s[n] = 1, then that shell existed and needed to be added to the screen. This would allow easier addition and removal of shells from the screen. In Figure 9 below, the initial location of the shell is set, or if the shell doesn’t exist, is set off of the boundaries of the screen. The else part was added later when the shells were colliding with the locations of the shells that did not exist. The other variable arrays are explained in the table below.

|  |  |
| --- | --- |
| Array | Explanation |
| s[ ] | Shell existence, if 1 then the shell should appear |
| sx[ ] | X position of the shell |
| sy[ ] | Y position of the shell |
| sup[ ] | Shell movement up/down (up=1, down=0) |
| slf[ ] | Shell movement left/right (left=1, right=0) |
| supn[ ] | Shell movement up/down, next |
| slfn[ ] | Shell movement left/right, next |

Table Array Explanations

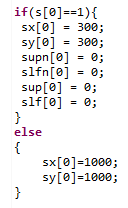


Figure Example of Setting Shell Position

* + - 1. Next, shell movement was added (without bouncing off of each other). This was completed similar to Mario’s current bouncing movement, which was based off of the Lab 7 block movement code. At this point, Mario and at least one Green Shell should be visible, bouncing around the screen but not interacting with each other when they reach the same location. Also, make sure to add a loop to move all of the shells (in the main loop), as in Figure 11.



Figure Code for Moving the Shells

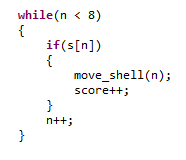


Figure Code for Calling Move Shell

* + - 1. At this point, there should be a (currently infinite) loop in the main of your code. One way to prepare this for later steps is to make a variable called “game” which is initialized to 1. Then, make a loop with “while(game)”, and game is not set to 0 anywhere yet. This will allow the movement of the sprites to keep moving.
    1. Bouncing Shells
       1. Next, the bouncing mechanic of the shells was added. The way this is completed is by making a function that checks if two specified shells are colliding with each other. This can be seen in Figure 12. This function takes two shell numbers and checks the positions of the shells, which are specified in the global arrays in Table 2. The function checks for any of the possible collisions between two shells.

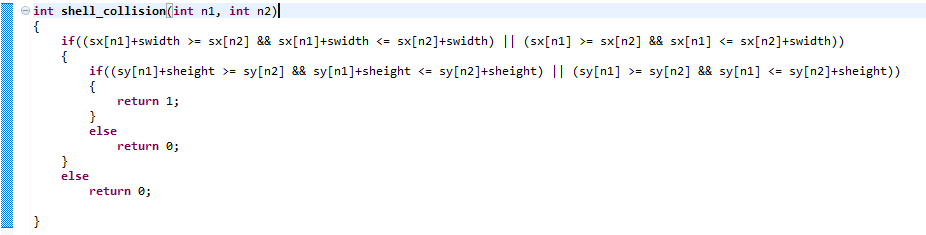


Figure Shell Collision Checker

* + - 1. In the main, there needs to be a loop that loops through all the shells to check if any of the shells collides with any of the other shells. Figure 13 shows this section of the main. The Mario collision will be explained in a future step.

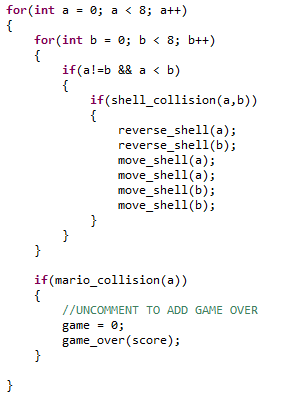


Figure Loop for Checking Shell Collision

* + - 1. Next, Mario’s location is checked with the shell locations. The Mario Collision function only takes one integer, because each shell only needs to be checked once to see if Mario collided with it. Mario Collision is almost identical to Shell Collision, but is reproduced below in Figure 14.

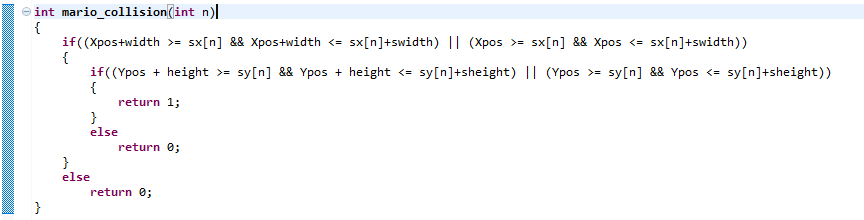


Figure Mario Collision

* + - 1. The best way to test at this point is to set up Mario and a Green Shell to move directly towards each other, hopefully resulting in a collision where the game variable is set to 0, as seen in Figure 13. Once the push buttons are added to the project, the shells can be setup in any locations, and the game can be played without a score.
    1. Timer / Score
       1. This part of the program draws heavily from the “SomeSimpleBitMapFont.c” file found in the course files. Using this, numbers were placed on the screen to make sure the files were integrated into our program correctly, and then the score mechanism was added. The idea is that for each shell, every movement is one point to the player. So eight shells (the default number for the game) makes the score have a multiplier of eight.
       2. To make the numbers appear as requested, the entire bottom row of the screen is allocated for the score, but will likely not be required because the score will not get very high. Every time a new number is to be displayed, the previous number in that location must be erased to make the numbers not overlap on the screen. The Print Score function in Figure 15 shows both the final implementation, but some of the tests used, as commented out code. Set Block is used to erase the old number, and a function known as Print Position is used to print the digit calculated in Print Score. Print Position can be seen in Figure 16. Also, note that the number is printed on the screen only if it is after a nonzero number, or is not zero itself.

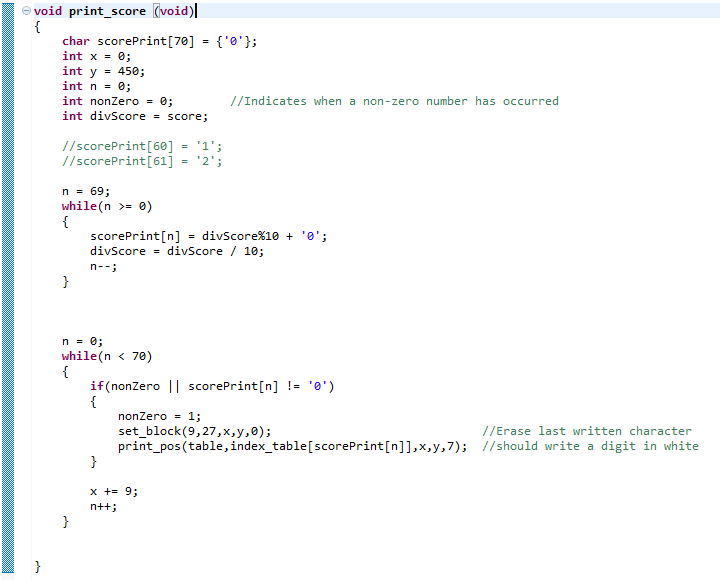


Figure Print Score

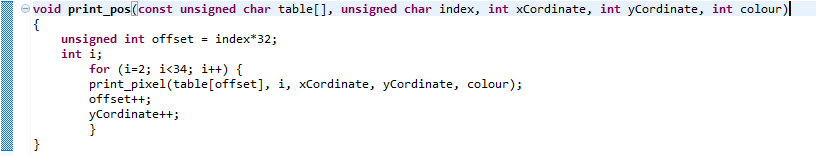


Figure Print Position

* + - 1. As an optional game mechanic, the speed of the shells can be adjusted depending on the player’s score by changing the step size.
      2. Combined with the steps below for the buttons and hardware, the game will be functional and will look like the screen shown in Figure 17.

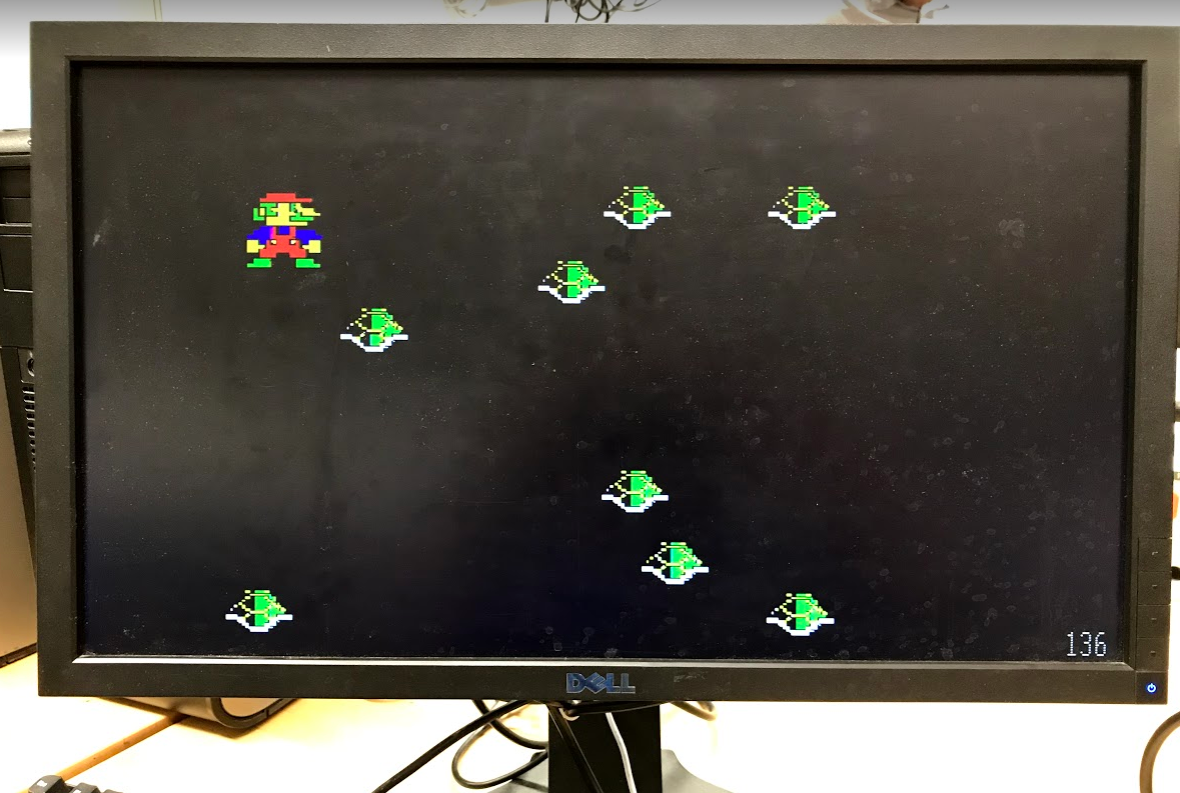
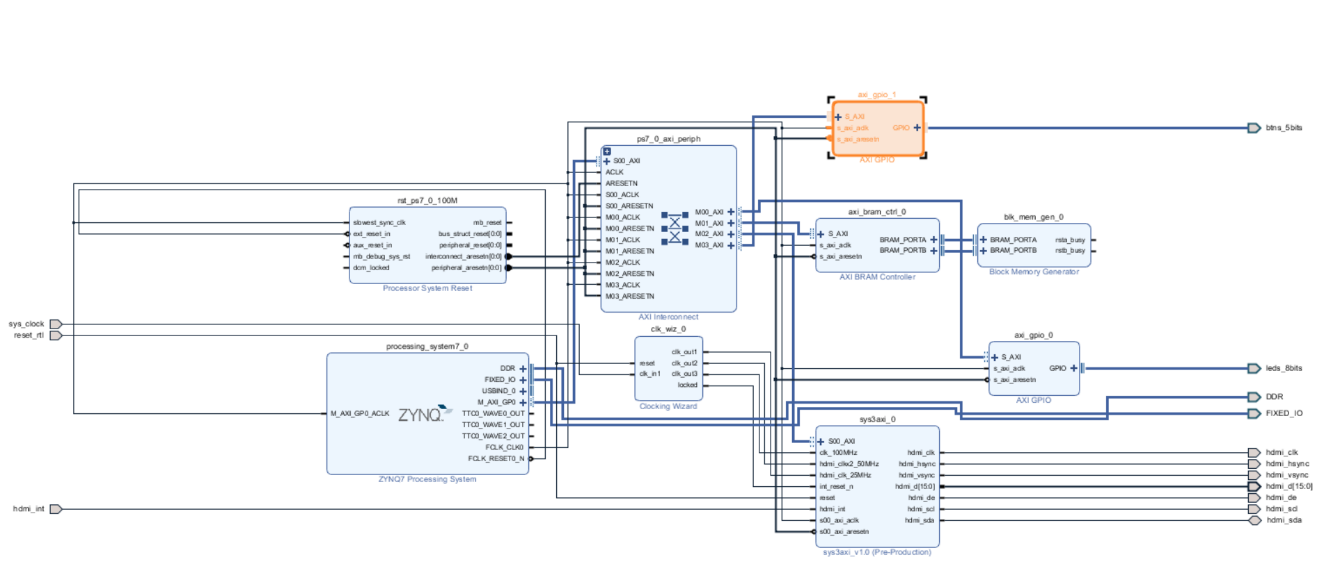
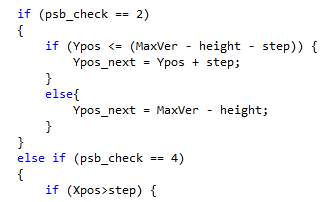


Figure Gameplay Screen

* + 1. Documentation
       1. The documentation phase mostly consisted on working on this report.
  1. Vincent:
     1. GPIO Push Button Programmable Logic
        1. The hardware from lab 7 was used as a baseline, and a GPIO block was added and set to btns 5 bits.

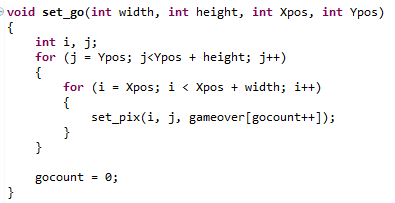
**Figure 18: New System Block Diagram**

* + - 1. The “reset\_n” signal was set to one of the five buttons that comprise the button GPIO block, so it was changed and set to switch 0 in the constraints file.
      2. The design was validated, output products were generated along with a new HDL wrapper and finally bitstream was generated.
    1. User-Controlled Mario
       1. After the push buttons were added on the hardware level, they were tested using the “SupplementButtonsSwitchesLEDsVivado2014.2.docx” file from class files in Canvas.
       2. Once this test was successfully running for the push buttons the moving block code was modified to move the block manually according to which button was pressed. This was accomplished by removing the “up”, “left”, “up\_next” “left\_next” from the moving block code and replacing them with the push button polling function (psb\_check = XGpio\_DiscreteRead(&push, 1);). This way the block moves only when a button is pressed and only in the direction of that button. The code that calculated the next move for the block was modified to calculated the next move according to the value returned by the button polling function.



**Figure 19: Example of updated moving block code.**

* + - 1. Next, the same idea was added to the code that moves the Mario sprite, so the user could control him using the push buttons.
    1. Game Over Screen
       1. The game over screen was implemented in basically the same way as the sprites. First, a usable game over image was found online.
       2. Then the image converter (<http://www.digole.com/tools/PicturetoC_Hex_converter.php>) was used to convert the image to a C array which was placed in a header file to reference from the main.
       3. All of the values in the array were then changed to one of the 8 values possible in 3-bit color (0-7).
       4. After this, the set\_block function was modified to draw the game over screen by referencing the game over array of pixels in the header file.



**Figure 20: Function that draws the game over screen.**

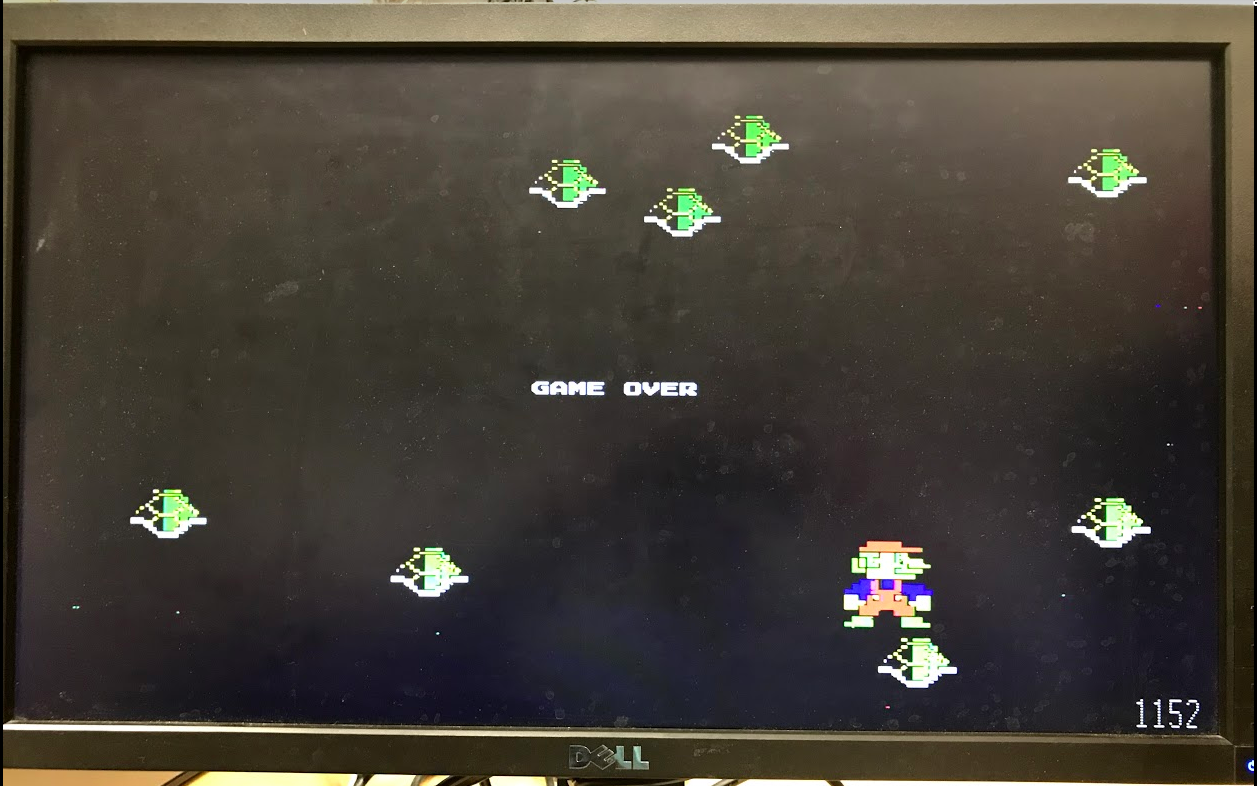


Figure 21 Game Over Screen

1. Results

After the steps above were completed, the resulting project was a resounding success. Not only did sprites appear on the screen, but could be controlled using the Zedboard and could interact with each other. One performance metric that was noted was the fact that the game slowed down depending on the number of shells present in the game. This makes sense because each additional shell adds a number of comparisons equal to the number of shells minus one. The resulting screen can be found in Figure 17 above, and the game functioned as intended. Part of the Project Summary can be found below.

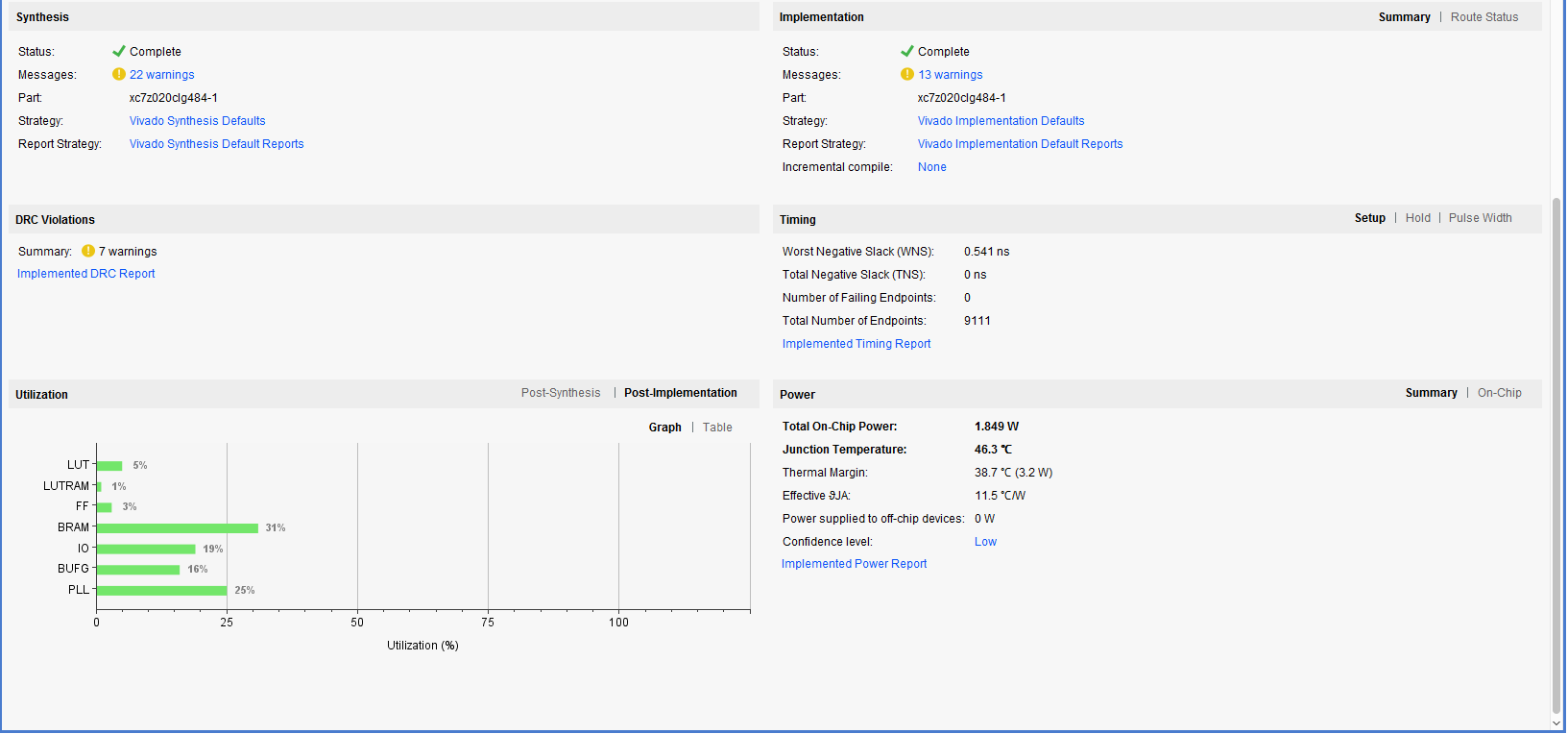


Figure 22 Snippet of Project Summary

1. Recommendations for Future Work

Future work could include some of the additional features that were not added to this project due to time constraints. Some of these include:

* + 1. Start Screen with Difficulty Levels
    2. Bowser as an Obstacle following the Player
    3. Computer Player with Difficulty Levels
    4. Multiple players at once

Appendix A: Works Cited

Nikkatsa. “Space Invaders FPGA Game.” *FPGAWORLD*, 11 Feb. 2017, fpgaw0rld.wordpress.com/2016/05/20/space-invaders-fpga-game/.

Nintendo. “8 Bit Mario.png.” *Mario Wiki*, Wikia, 2 June 2012, mario.wikia.com/wiki/File:8\_Bit\_Mario.png.

Nintendo. “Koopa Shell Icon.” *Icon Easy*, Icon Easy, www.iconeasy.com/icon/koopa-shell-icon/.

“SomeSimpleBItMapFont.c.”