Yi Wang and Stephen Pueblo

6.111 Final Project

December 13, 2006

Abstract

This paper details the recreation of the classic arcade game Donkey Kong. The arcade game was created by two students using the Hardware Description Language, Verilog. Our goal was to implement three levels of the game. We were successful in accomplishing our goal. In the time frame of several weeks, we designed and built both the Game and Display Logic, which are the two main components of this project. The player controls Mario, and the objective of the game is to reach the princess. Overall, the experience was valuable and enjoyable.

Table of Contents

List of Tables

List of Figures

Overview

Donkey Kong was created by the legendary game designer from Nintendo, Shigeru Miyamoto. He and his development team created the game in the year 1982. It was an instant hit in arcades around the country, and Nintendo sold thousands upon thousands of systems per month. With this in mind, we decided to implement a recreation of this game on the FPGA lab kit.

There are three levels in the game. The game consists of three main characters: Donkey Kong, the princess, and Mario (originally known as "Jumpman"). The princess is situated on the very top platform and Donkey Kong is situated on the platform below her on the far left. Mario starts off the game at the lowest platform on the far left side. The player controls Mario, and there is only one objective to our game: to reach the princess and "save" her from the evil monster, Donkey Kong. There are ladders connecting each platform. While the player tries to climb up these ladders to reach the princess, Donkey Kong tries to throw barrels in order to impede Mario's progression. If Mario gets hit by a barrel, he loses one life. If he loses all of his lives, the game is over. However, if he reaches the princess, the player proceeds to the next level.

In the second level, there is a timer. The objective is the same as in the first level, but now the player needs to reach the princess in thirty seconds or less. If the player does not reach the princess in that time frame, he or she loses a life. On the other hand, if the player reaches the princess in the second level, he or she proceeds to the third level, which is the final level. The final level has one last twist to it. Not only is Donkey Kong throwing barrels to stop the player from reaching the princess, but there are also "electrocution zones" where if Mario touches them he dies. If the player manages to overcome these obstacles and reach the princess for the third and final time, the player wins the game, and the game is over.

Module Descriptions and Implementations

Description

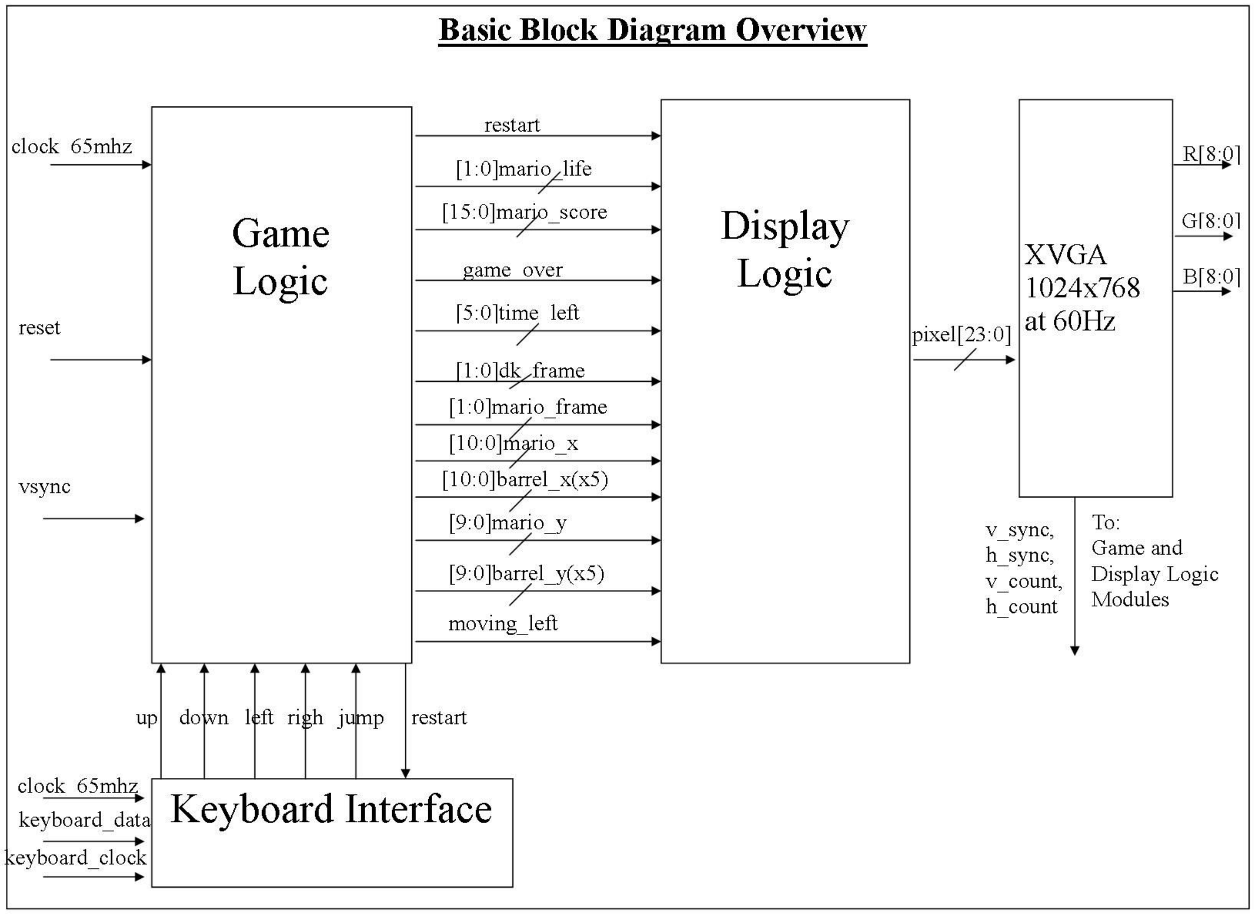


Figure 1. Basic Block Diagram Overview

Game Logic

The Game Logic controls the visuals onscreen. The sole purpose of the Game Logic is to make sure the user (who controls Mario) and the objects in the game adhere to the strict rules designed for them. In order to achieve its goal, the game logic has to be designed to calculate and know numerous details. The details it needs to know and calculate range from calculating the current positions of Mario and the barrels onscreen to knowing when to end the game or advance to the next level.

The Game Logic consists of seven main modules: Mario Logic, Barrel Logic, Collision Detector, Game Finite State Machine, Donkey Kong Logic, Clock Divider, and Game Timer. The main purpose of the Mario Logic is to take the control signals from the Keyboard Interface (the up, down, right, left, jump signals) and to use those signals to update Mario's position and frame of animation. The Barrel Logic controls the barrels on the screen. Like the name implies, the Collision Detector detects when Mario collides with a (detects also when Mario jumps over a barrel) barrel, the princess, or Donkey Kong. The Game FSM is the master controller of the entire Game Logic. It controls which level the game should be in at the moment. The Game FSM also keeps track of the score and the amount of lives Mario has left, and it tells the other modules when re-initialize themselves. Donkey Kong Logic uses the Clock Divider to determine when to tell the Barrel Logic to create the new barrel onscreen. Donkey Kong Logic also determines the specific frame of animation Donkey Kong should be in at that particular moment. The Clock Divider is the simplest module of them all. Its sole purpose is to take the 65 MHz video clock and convert it into a 1MHz clock, which Donkey Kong Logic and Game Timer will use. The Game Timer is used by the Game FSM for the second level (in the second level the user has to reach the princess in thirty seconds). It also has to determine when to transition to the next state. All seven of these modules work in unison to create the Game Logic. Figure 2 details the schematics of the Game Logic.

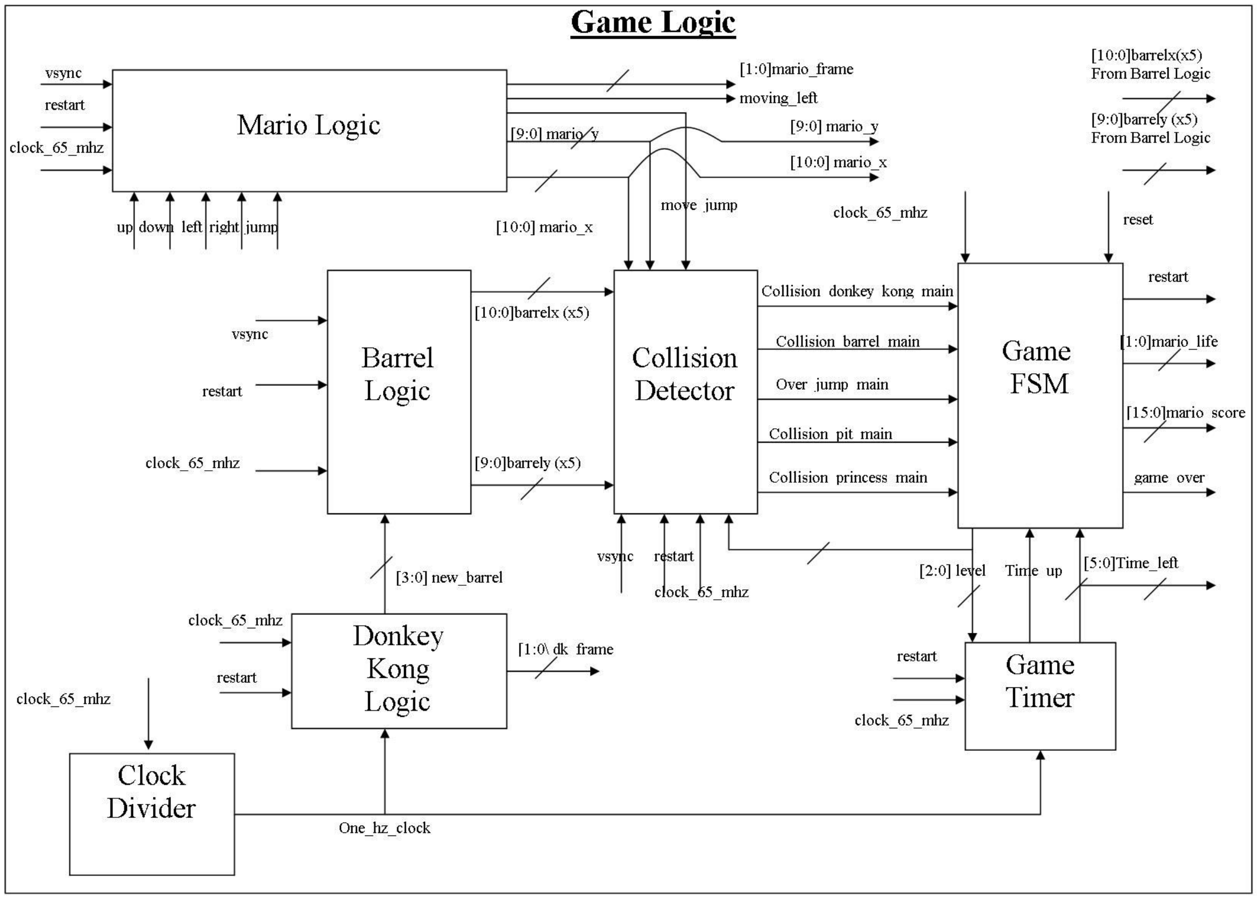


Figure 2. Block Diagram of Game Logic.

Clock Divider

The purpose of the Divider is to take the 65 MHz video clock and to create an output that is a 1MHz enabled clock. A counter, with the use of a register to store its value, is created within the Clock Divider module. It increments itself by one each time the video clock goes high. When the counter reaches the number 64,999,999, the Clock Divider goes to one. Therefore, by outputting a high value pulse on the video clock's 65,000,000th cycle, a 1MHz clock is created. There is a reason why a 1MHz clock is specifically needed. With a 1 MHz clock, a system can count in seconds instead of nanoseconds. Once the Clock Divider outputs its high pulse on the 65,000,000th cycle, the counter resets to zero, and the entire cycle is repeated again. In addition, when the restart signal is asserted by the Game FSM, the counter resets to zero.

Collision Detector

The Collision Detector is designed to achieve several goals. It needs to send a signal when Mario hits (or jumps over a barrel) a barrel, the "danger zone", Donkey Kong, or the Princess. To make the goals realizable, the Collision Detector is not designed to be a single module. Instead the module known as the Collision Detector consists of several sub-modules (seven in total), which when wired together create the Collision Detector system. Figure 3 shows the diagram of the Collision Detector.

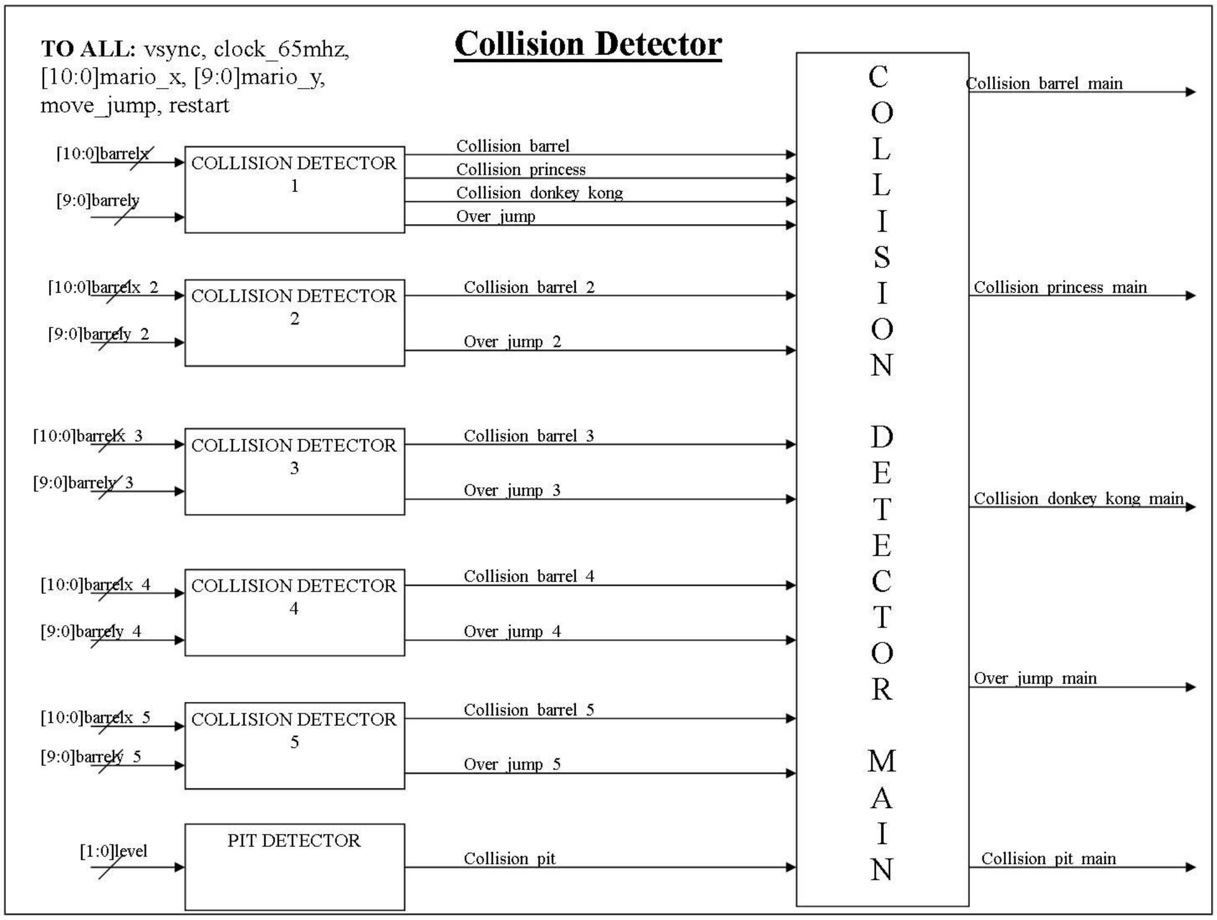


Figure 3. Block Diagram of the Collision Detector

There are a maximum of five unique barrels, which can roll onscreen. From here on, the five unique barrels will be called Barrel #1, Barrel #2, Barrel #3, Barrel #4, and Barrel #5. During the last level there are three “danger zones,” which will cause Mario to die if he touches them. Collision Detector 1’s purpose is to detect if Mario either hits or jumps over Barrel #1, collides with Donkey Kong, or reaches the princess. In order to detect these four different scenarios, Collision Detector 1 takes in as inputs the X and Y coordinates of Barrel #1 and the X and Y coordinates of Mario. During every frame, Collision Detector 1 tries to determine whether one of those four previously mentioned scenarios has occurred. As a side note, like the majority of the modules in the Game Logic, Collision Detector 1 goes through its calculations every time the frame changes. If one of the scenarios does occur, Collision Detector 1 will output a signal to the Game FSM.

In order to determine whether Mario collided or jumped over anything, a series of exhaustive tests were made. Taking the coordinates of Mario and the barrel (the Collision Detector already knows the coordinates of the princess and Donkey Kong), Collision Detector 1 uses a series of conditional statements, which contain inequalities and ranges, to determine when Mario has jumped over a barrel or collided with an object in the game. In order to determine when Mario is jumping in the air and possibly jumping over a barrel, Collision Detector 1 uses the move_jump signal from Mario Logic. The pixels of the platforms in the game are already known by the Collision Detector, and are use in the series of conditional statements. The ranges are carefully thought out, since being off by even one pixel could introduce glitches and annoying bugs. In order to detect whether Mario has collided with a barrel, Donkey Kong, or the princess, the Collision Detector compares the coordinates of the three objects onscreen with Mario's current pixel coordinate. Using these coordinates, the Collision Detector can determine if Mario is touching an object that he is not suppose to touch. After all of the calculations are done, Collision Detector 1 outputs four values: *collision_barrel, collision_princess, collision_donkey_kong,* and *over_barrel*. *Collision_barrel, collision_princess* and *collision_donkey_kong* are asserted when Mario collides with any of the three objects. *Over_barrel* is asserted when Mario jumps over a barrel. Not unlike the Clock Divider, when the restart signal is asserted by the Game FSM all the collision signals revert by to zero.

Collision Detector 2, 3, 4, and 5 are nearly identical to Collision Detector 1. The only difference between these four detectors and Collision Detector 1 is that the other four do not determine whether Mario collides with the princess and Donkey Kong, since Collision Detector 1 is already in the mist of testing those conditions. Collision Detector 2, 3, 4, and 5 are identical, and they follow the same logic as Collision Detector 1. The only difference between Collision Detectors 2, 3, 4, and 5 is that each individual collision detector determines whether Mario collides with its own specific barrel (i.e., Collision Detector 2 only determines when Mario collides or jumps over Barrel #2).

The Pit Detector is essentially the same as the other detectors, but it takes in an additional input. Because it is only used in the last level, it takes in the *level* signal to determine when to calculate whether Mario has run into a "danger zone" or not. The *level* signal tells the Pit Detector which level the user is currently playing. There is no need to take in any other coordinates for inputs except for Mario's coordinates, since the "danger zones" are stationary. During every frame, the Pit Detector outputs the collision_pit signal, which tells the Game FSM whether Mario has ran into a "danger zone".

Because there is a need to condense the outputs of the six sub-modules into cohesive output signals, another module is needed. This module is called the Collision Detector Main. It takes in all of the outputs of the six collision modules, and condenses them into only five signals.

Several OR gates are used. For example, all the *collision_barrel* signals from the five different Collision Detectors are OR'd together to form one signal which is called *collision_barrel_main*. The same logic is applied to the other signals. If there are no similar signals for a particular output, the signal is still routed through the Collision Detector Main to keep the wires logically organized.

Donkey Kong Logic

Donkey Kong Logic is a fairly simple module in design. It receives the 65 MHz video clock and the 1 MHz clock from the Clock Divider. The module is triggered on the video clock, and whenever the 1 MHz clock is high it increments its clock counter, which initially starts off at zero. There are three frames of animation. The three frames consist of Donkey Kong picking up a barrel, throwing a barrel, and staying inactive. When the clock counter is one, two and three, the Donkey Kong module outputs a different frame of animation. When the counter is at four, the Logic tells a specific Barrel Module to create a barrel and the frame of Donkey Kong throwing a barrel is drawn onto the screen. This gives the player the illusion that Donkey Kong is actually rolling a barrel. Moreover, when the clock counter reaches four, it is rolled back to zero, and the same process is repeated continuously.

Game Timer

The Game Timer has several purposes. Its purposes are to keep track of how much time is left in the second level and to output the remaining time to the Visual Logic and the Game FSM. When on the second level, the Game Timer also outputs a signal to the Game FSM telling the Game FSM when the time is up for that level. In order to perform correctly, the Game Timer is designed in a similar vein as the Donkey Kong Logic. It takes in as inputs the video clock and 1MHz clock from the clock divider. The timer starts off at thirty seconds. Whenever the 1MHz clock is high, it decrements one from the remaining time left. When the remaining time is equal to zero and user is on the second level, the *time_up* signal is asserted, and the timer is set back to thirty. If the game is on any other level, the timer just resets back to thirty. Because a 1 MHz clock is used, each time the clock goes high one second has elapsed, so in reality the timer is timing when thirty seconds have passed.

Barrel Logic

The Barrel Logic consists of five smaller, individual modules. Each of the five modules is responsible for keeping track of the coordinates for its own particular barrel. Subsequently, because there are five individual barrel sub-modules, there can only be a maximum of five barrels onscreen at a time. As inputs, all five modules receive the video clock, the *vsync* signal, the *restart* signal, and the *new_barrel* signal. Since the logic for each barrel sub-module is identical, only one would be discuss in detail.

The *new_barrel* signal comes from the Donkey Kong Logic. The number from the *new_barrel* signal determines which barrel needs to be initialized on the screen. Once the barrel is initialized and in its proper place, the barrel starts rolling on the platforms. The barrels roll by falling off of each platform and rolling onto the next platform. When the barrels reach the last platform and rolls toward the left of the screen, they disappear until they are reinitialized again by the *new_barrel* signal from the Donkey Kong Logic. The coordinates of the specific barrel were updated every frame by using an edge trigger on *vsync*, in a similar manner as was done with the Collision Detector. Using the coordinates of the platforms and barrels in its calculations to determine the barrels' new coordinates, the barrels roll properly down the platform.

Directional vectors are used to control the movements of the barrels. There are two vector signals used in the Barrel Logic: right and down. If right equals one, then the barrel should be going right. If right equals zero, then the barrel should be moving left. The same logic can be applied to the down signal. When a barrel's coordinates matching a specific condition in the Barrel Logic, a directional vector could potentially be changed. If so, the Barrel Logic will move the barrel a certain speed in the direction the directional vector signal specifies. Each barrel sub-module moves its barrel at different speeds than the other ones, therefore making the game more challenging and more difficult.

The outputs of each barrel module are the X and Y coordinates of each barrel. So, technically, there are five barrel X values and five barrel Y values being outputted by the Barrel Logic, each corresponding to a different barrel's coordinates. These coordinates are given to the Visual Logic and are also routed into the Collision Detector. Figure (INSERT NUMBER HERE) gives a block diagram of the Barrel Logic that was used in the Donkey Kong game.

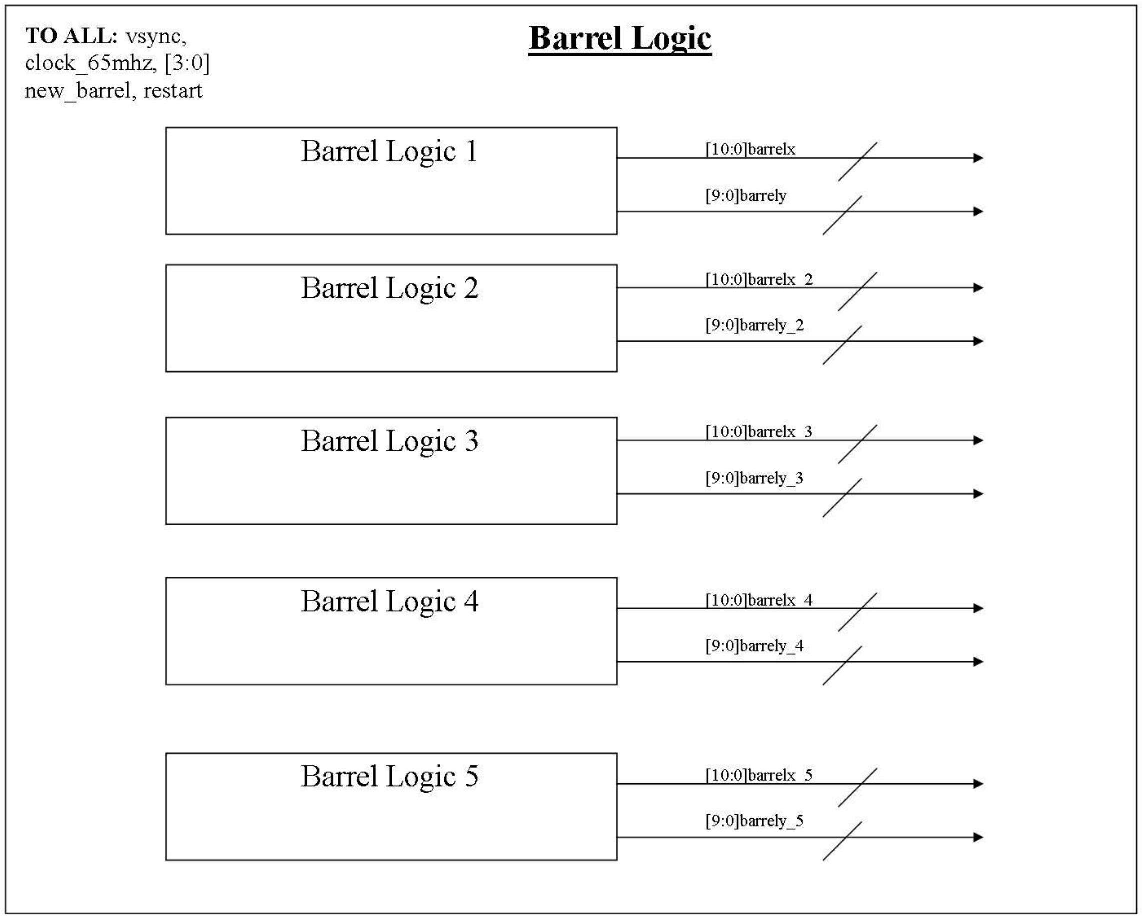


Figure 4. Block Diagram for the Barrel Logic

Mario Logic

The most difficult model to design and implement, the Mario Logic module's purpose is to give the Visual Logic the current coordinates of Mario, which is the character the user controls in the game. As inputs, the module takes in the control signals from the keyboard (*up, down, left, right, and jump*), the *vsync* signal, the video clock, and the *restart* signal from the Game FSM. Mario Logic outputs the X and Y coordinates of the character Mario. Mario Logic goes through its calculations of Mario's position during the transition of every frame. During the start of each game, Mario starts off at the far left of the lowest platform. The calculations are done using the coordinates of the various objects onscreen.

If the user wants Mario to jump and Mario is not about to go off the screen, the logic will make Mario jump (depending on the user's input) either in a north, northeast, or northwest direction. In order to make Mario "jump," a counter was needed. Once the Mario module receives the jump signal, it will create a signal to tell itself and the Collision Detector that Mario is in the midst of jumping. It will also change the coordinates of Mario to put Mario in the air. When in "jump mode," Mario Logic will increment a counter, which starts off at zero. During every clock cycle the counter increments itself by one. Every multiple of two that the counter increments to, the logic will move Mario down one pixel back from the air towards the platform. When the counter reaches the value of sixty, it will reset itself and the jumping signal back to zero. However, if Mario is suppose to jump left or jump right, every time the counter increments ten times (starting from five), the logic will move Mario left or right three pixels while also moving Mario down.

If Mario is off a platform, then he should fall down. The falling block of the Mario Logic calculates how Mario should move down. When the logic realizes that Mario should be falling, two signals are asserted high to tell itself that Mario should be falling and not doing anything else. These signals are the *falling* and the *no_move* signals. Once Mario Logic knows that Mario needs to be falling down, the logic moves Mario down until he is about to hit a platform. When the logic knows Mario is about to hit a platform during the next frame, it reasserts the *falling* and *no_move* signals back to zero.

When the user tells the Mario Logic to move Mario left or right, the logic moves Mario left or right by two pixels each frame until the user decides to not move Mario anymore. There are constraints added to the logic to make sure Mario will not move off of the screen while walking or jumping. While walking either left or right, the user can decide to go up a ladder when he or she reaches one.

If the user is in the proper range of the ladder and when he or she presses the up button, Mario will move up the ladder by two pixels per frame. Not unlike what the logic does when Mario is falling, when Mario is climbing the ladder the same *no_move* signal is asserted high in order to make sure Mario can only move vertically and to make sure Mario is not able to move in any other direction. While Mario is on the ladder, the user can also make Mario move down. If Mario should be going down, the logic moves Mario down two pixels per frame until the user decides not to move Mario down anymore. There are numerous conditional statements checking to see whether Mario is about to reach another platform. If Mario is about to reach another platform, the logic reasserts the *no_move* signal back down to zero, and afterwards Mario will then be able to walk along that newly reached platform.

The Mario Logic also tells which frame of animation of Mario to display on screen. When Mario is walking right, he alternates between two walking frames of animation every twenty frames. The same occurs when Mario is walking to the right; however, there is one caveat. The "moving_left" signal is asserted high to tell the Visual Logic that Mario is moving left. Knowing that Mario is moving left, the Visual Logic while reverse Mario's image. Also, when Mario is on the ladder, his frame changes to a frame where Mario's back is showing. Using these five frames of animation (if you include the reverse image of Mario when he is walking left), gives the user the illusion that Mario is actually walking on platforms and climbing up ladders.

Keep in mind that when Mario Logic calculates Mario's new X and Y coordinates, logical calculations are made. Whether it is checking to see if Mario falls off of a platform or checking to see if Mario is walking along a platform, a majority of the logical conditions deal with specific coordinates. This is the only way to check to see if Mario meets certain conditions, since Mario's own X and Y coordinates are coordinates of the "game screen." The X coordinates range from 0 to 1023, and the Y coordinates range from 0 to 767.

Game Finite State Machine

A FSM was created to serve as the "central hub" of all the other modules in the Game Logic. It takes in the collision signals from the Collision Detector; the amount of time left in a level and the *time_up* signal from the Game Timer; the 65 MHz video clock; and the *reset* signal from the user. Using these signals as inputs, the Game FSM will determine when to tell the other modules in Game Logic to reset; determine how many lives Mario has left; determine the current score; and determine what level the user is currently playing.

Whenever the FSM receives the *reset* signal from either the user or the initial power up sequence, it will reset and do the following actions: reset the score back to zero, re-initialize the amount of lives Mario has back to three, and revert back to the Level 1 state. All of the calculations and actions done by the FSM are triggered on the video clock. The diagram below details the state transitions of the Game Finite State Machine.

Game Finite State Machine

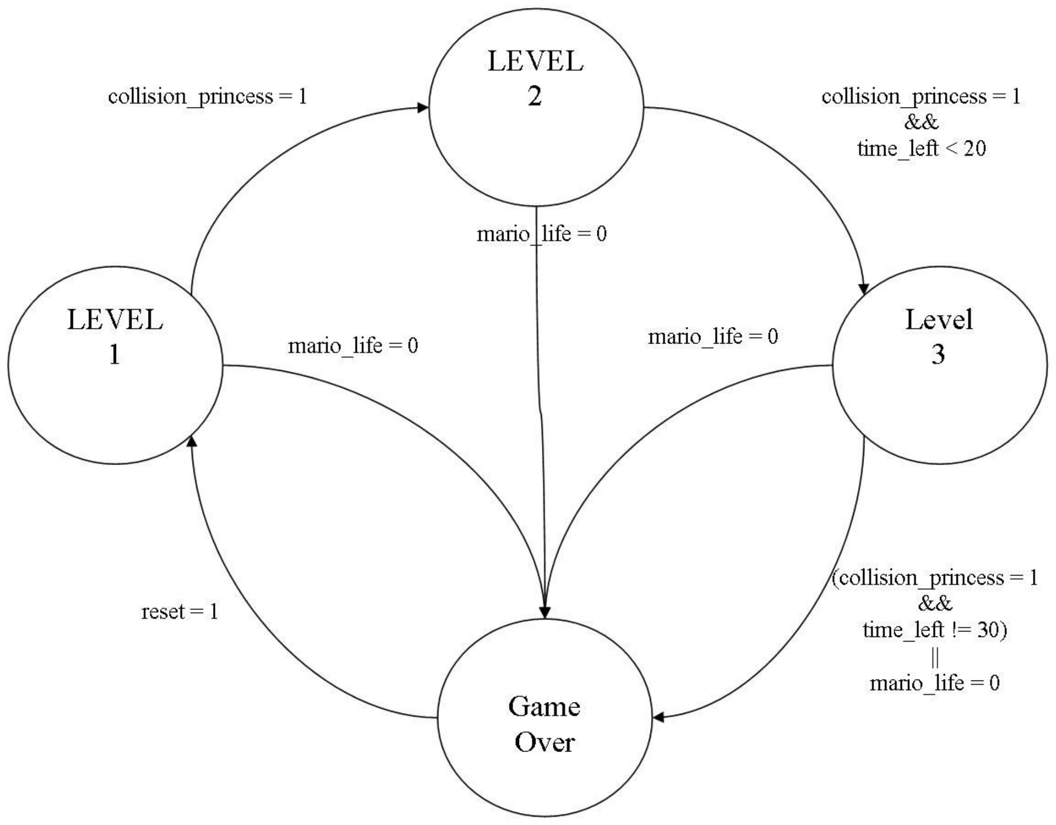


Figure 5. The Game Finite State Machine's state transition diagram

The FSM starts off in the Level 1 state. When the player reaches the princess, the FSM will transition to the next state, which is level two. There is one extremely subtle nuance that needs to be properly explained. The FSM only transitions from Level 1 to Level 2 and from Level 2 to Level 3 when the *collision_princess* signal is high. The *collision_princess* signal is high for more than one clock cycle; therefore, other constraints had to be added to make sure the FSM would not transition through multiple states at a time. The other constraints are to make sure the *time_left* signal is not thirty, or to make sure the *time_left* signal is less than twenty. Since the player cannot conceivably reach the princess in less than ten seconds, these constraints make sure that the FSM does not erroneously transition to another state. When the user reaches the princess in the Level 2 state, the FSM transitions to the Level 3 state, and when the user reaches the princess in the Level 3 state, the FSM transitions to the Game Over state. The FSM will stay in the Game Over state until the user manually resets the game. Also, if the user loses all of Mario's lives in any of the states, the FSM transitions to the Game Over state.

If the FSM receives the *over_barrel* signal from the Collision Detector in any state other than the Game Over state, it will increment the score by 100 points. If, on the other hand, the FSM receives any of the other collision signals from the Collision Detector (i.e., collision_barrel), it will decrement a life from the remaining lives Mario has left. In addition, when the FSM is in the Level 2 state and receives the *time_up* signal from the Game Timer, Mario's loses one of his lives

In any of the states, the FSM will output to the Visual Logic the score, the number of lives Mario has left, and the current level the user is playing. It will also output the *restart* signal to the other Game Logic modules when the user resets the game or when the game is in the Game Over state. When in the Game Over state, the FSM outputs a high *game_over_screen* signal to the Visual Logic in order to tell the Visual Logic to display the Game Over screen.

DISPLAY LOGIC OVERVIEW

The overall system was broken down into smaller modules, each with a specific function. Dividing the task into simpler pieces made the design as well as the debugging process much easier to grasp. The project was implemented with eight main modules. Table 1 below summarizes the function of each.

Module Name	Module Function
vga_ladder	Generates the pixels for the ladders to be displayed on the monitor
vga_moving	Generates the pixels for Mario and the 5 Barrels (moving objects) to be displayed on the monitor
vga_platform	Generates the platform pixels to be displayed on the monitor
vga_stationary	Generates pixels for the stationary objects (Donkey Kong, Peach, Life count) to be displayed on the monitor
vga_title	Generates the pixels for the Start and Game Over screens
screenbuffer	Takes as input moving_pixel, bram_pixel, platform_pixel, ladder_pixel, gameover_pixel, and start_pixel from the above 5 modules and determines which pixels are outputted onto the screen
vga_score	Generates a 5 digit score to be displayed on monitor
vga_timer	Generates the pixels for the Timer Countdown to be displayed on the monitor.

Table 1. Description of the 8 main display logic modules.

This first phase involves designing the functionality of each module. Figure 7 shows a block diagram of the overall display logic.

As can be seen in the block diagram, each module was chosen to assemble graphics of similar properties. For example, all the barrels and Mario frames are dealt with in the *vga_moving* module. These seemingly simple separations made life a lot easier when it came to developing the logic for each module because it was easier to specify instructions such as displaying moving objects in front of ladders if the two pixel positions were to overlap. Now, I will describe in detail the methods and logic behind the modules. Because these ideas recur in all my modules, I will present them here only once, and not repeat them during each module presentation.

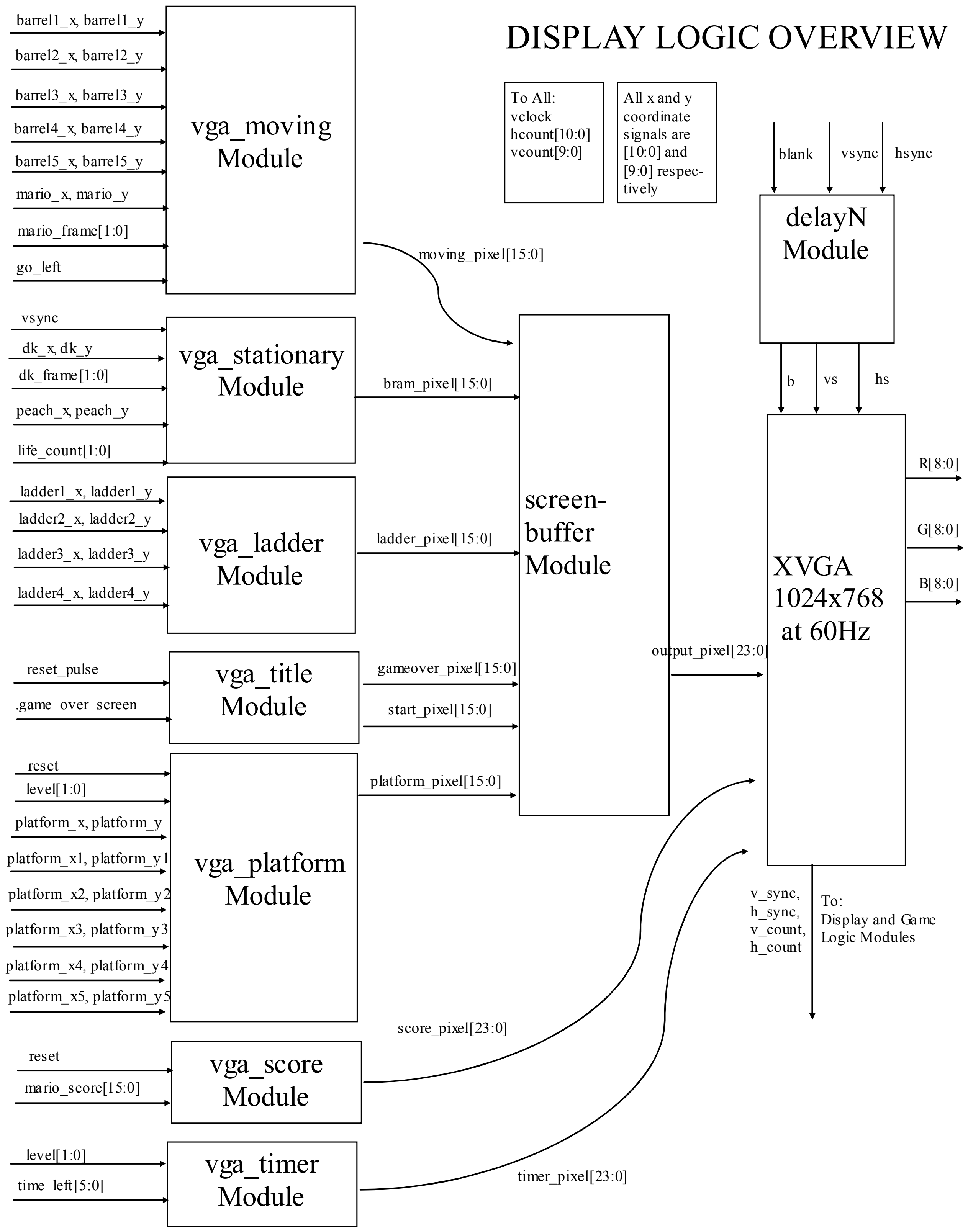


Figure 7. Display Logic Overview Block Diagram

Graphics Generation Method

I will begin by describing the process used to generate the game graphics. Each of the letters for the "DONKEY KONG" and "GAME OVER" screen were designed by Stephen and me using Adobe Photoshop. The animation frames for Mario, DK, and Peach were screenshots taken of the game online[1]. The screenshots obtained online were modified and redrawn in some parts to fit size and color constraints. The platforms, ladders, and barrels were replicated from the ones I saw in the actual arcade game.

In order to render the jpgs into a format suitable for the labkit, we used Paint to first convert the files into 16-COLOR BITMAP. I then used a MATLAB BMP2COE conversion tool[2] to convert the bitmaps into the COE format which is accepted by the Xilinx FPGA memory generator. Please refer to the attached Appendix for the detailed code. After successfully generating the COE file, the FPGA's block rams (BRAMs) were used to build the memory to store these pictures. When generating the BRAMs, I named each by their size. Thus, Mario32x32 means the Mario graphic is 32 pixels by 32 pixels. In order to call the stored pixels from memory, the appropriate read-addresses need to be given to the BRAM.

Additional, location zero of every BRAM in our project was initialized to zero. I did this by using notepad to modify the COE file. This way, when the object is not within the region in which it is supposed to be display, it is easy to assign the BRAM read-address to zero, and have the black pixel be outputted. Black is represented by 0's. This property is useful because anything OR'ed with black will not be affected. Also, our background is black, so it blends in well.

Three-Stage Pipeline Implementation

To synchronize the timing of the pixel arrivals onto the monitor, a three-stage pipeline was used throughout the code. To make the point clear, please find below an example of how the process is used to generate the fifth barrel's pixel values:

```
barrel5_delay <= (vcount-barrel5_y);
barrel5_delay2 <= hcount-barrel5_x;
raddr5 <= {barrel5_delay, barrel5_delay2};

barrel5_new_delay <= 1;
barrel5_new_delay2 <= barrel5_new_delay;
barrel5_new <= barrel5_new_delay2;
```

The computation of each read-address into the BRAM takes two clock cycles. The additional registers were added to give the logic enough time to propagate through within one clock cycle without causing timing violations. An additional clock cycle is also needed to account for the BRAM access time. Because of these three cycles of delays, the pixel outputs of the BRAMs are

[1] www.fetchfido.co.uk/games/donkey_kong/donkey_kong.htm

[2] http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=12437&objectType=file

offset by three cycles if displayed directly onto the monitor. Pipelined signals such as *title_display*, *plat_displaynew*, *barrel5_new*, and *mario_new* are used to account for this timing delay. Finally, the *vsync, hsync, and blank* signals all needed to be delayed by 3 clock cycles too in order to ensure that all the signals arrive at the monitor at the correct locations at the right times.

Generating the BRAM Read Address

All my modules use very similar methods for generating the BRAM read addresses. I will run through an example of how to calculate Donkey Kong's read address from BRAM. The pipelined calculation consists of first calculating (vcount-dk_y). dk_y is the y coordinate of the top-left hand side of the Donkey Kong graphic. (vcount-dk_y) represents how deep we are into the picture vertically. The next calculation is (hcount-dk_x), where dk_x is the x coordinate of the top-left hand side of the Donkey Kong graphic. This value represents how deep we are into the picture horizontally. Each location in the BRAM represents a pixel on the screen. Thus, given that DK is 64 by 64 pixels, location 0 of the BRAM represents the top-most left hand corner of the picture, while location (64*64-1) represents the bottom-most right hand corner. In other words, the size of a WIDTH by HEIGHT rectangular object's BRAM read address will be [WIDTH*HEIGHT-1 : 0]. Location 4 in the BRAM would be the fifth pixel in the first row. In order to systematically calculate the read-address, the following process was used:

1. Multiply the vertical depth of the current pixel location by the width of the picture (given the picture is rectangular)
2. Add to the result from step 1 the value of (hcount-dk_x).

Because multiplying can take a long time, I designed most of my graphics to be multiples of 2. This way, I can shift instead of multiplying, and save valuable computation time. The final read-address calculation can be summarized below:

```
// pipelined BRAM read address calculation
DK_addr1 <= (vcount-dk_y);   // how far into the picture
                             // vcount is vertically with respect to the top-left
                             // hand side of DK
DK_hcount <= hcount-dk_x; // how far into the picture
                             // hcount is width-wise with respect to the top-left
                             // hand side of the DK to be displayed
raddr3 <= {DK_addr1, DK_hcount}; // DK_hcount is 6 bits so the
                                    // shift is equivalent to multiplication by 64
                                    // saves significant computation time
```

The following paragraphs will go into detail about how the different modules work together to form the game graphics you saw in lab.

vga_title Module

The *vga_title* module is responsible for generating the pixels to be displayed onto the monitor when the game first starts, as well as when it ends. Please refer to Figure 8 for its block diagram.

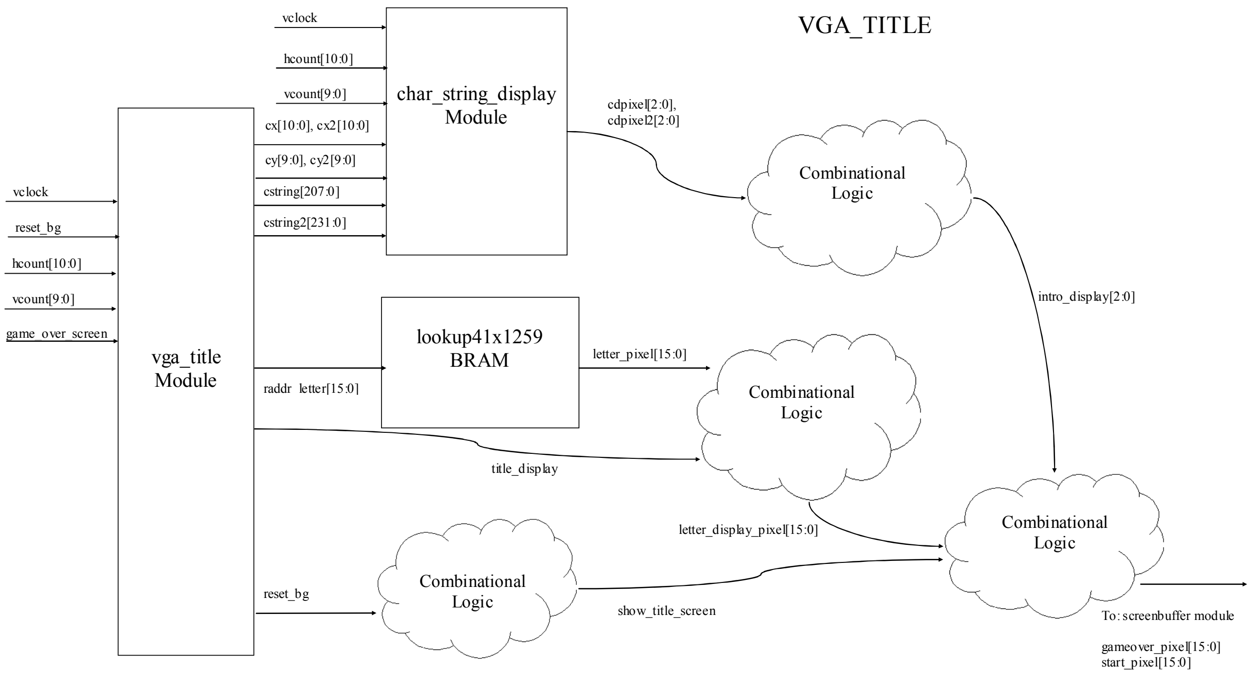


Figure 8. *vga-title* Module Block Diagram

The module takes as inputs *vclock*, *reset_bg*, *hcount[10:0], vcount[9:0],* and *game_over_screen*. *vclock* is a 65Mhz clock on which the video display runs. *hcount* and *vcount* specify the current horizontal and vertical indexes of the pixel's position on the monitor. The *game_over_screen* is high once Mario has lost all his lives, or once Mario passes all three levels and saves the Princess. *reset_bg* is a pulse which is high for one vclock cycle after the ENTER button on the labkit has been pressed. I used combinational logic to keep track of changes in *reset_bg*. Every time it is changed, *show_title_screen* will toggle between 1 and 0. Thus, when the user presses the ÉNTER button on the labkit once, the Start Screen will show up. When the user presses it again, *show_title_screen* will go low, and the Start Screen will not be displayed.

Start Screen

Figure 9 below shows the start screen: It consists of the letters "DONKEY KONG" centered across the monitor, as well as, "Yi Wang and Stephen Pueblo", and "6.111 Fall 2006 Final Project".

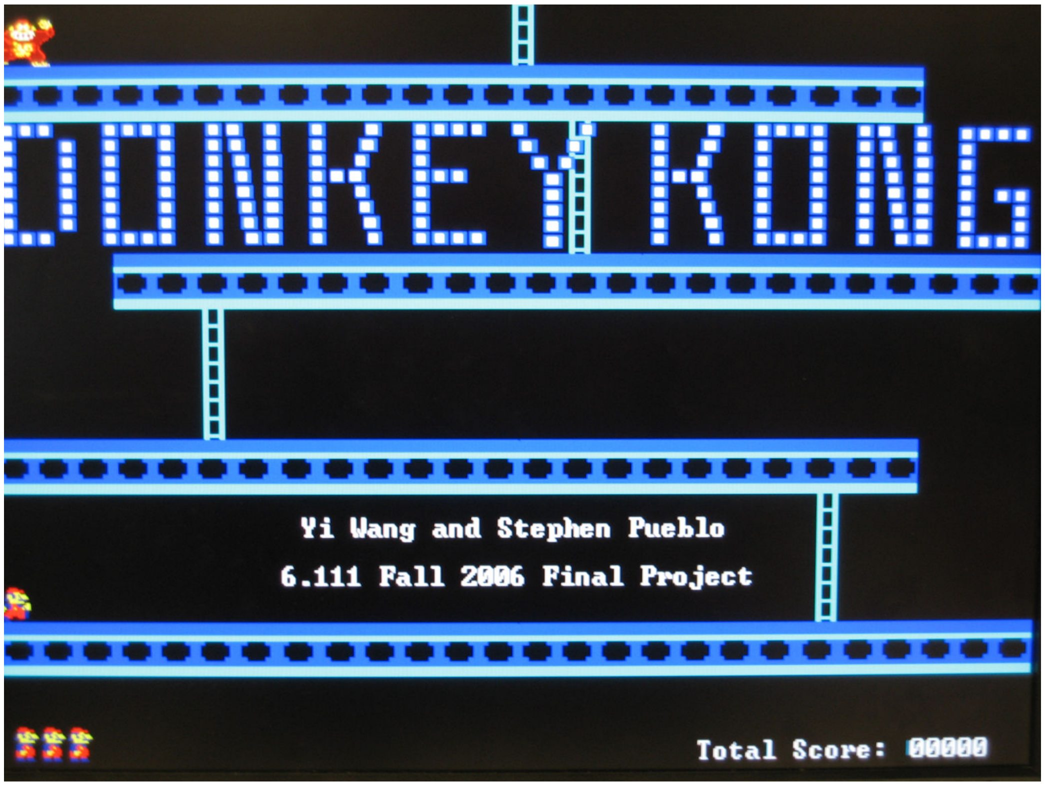


Figure 9. Start Screen Photograph

In thevga-title block diagram, *cstring* represents “Yi Wang and Stephen Pueblo”, while *cstring2* represents “6.111 Fall 2006 Final Project”. (*cx[10:0]*, *cy[9:0]),* and (*cy2[9:0], cx2[10:0]*) are the x and y coordinates of the top-left hand side of *cstring* and *cstring2* respectively. The *char_string_display* module was obtained from the “Fall 2005 – 6.111” course website. It looks up each character of *cstring* in a BRAM lookup table, and outputs the corresponding pixel. In the block diagram, these are *cdpixel[2:0]* and *cdpixel2[2:0]*. The two signals are put through an OR gate to produce *intro_display[2:0]*.

The title screen “DONKEY KONG” graphics were designed by Stephen and me. Each letter was placed under the previous one in a “graphic lookup table”. The letters were all the same size to ensure easy read-access from the BRAM. The picture stored into the BRAM is half the size of the picture which is eventually displayed onto the screen. This was due to the limiting number of BRAMS available for use: In order to save space, I shrunk down the picture horizontally by a factor of ½. When being read out from the BRAM, each pixel is repeated twice. The repetitive nature of our graphic made this visual effect easy to implement. The read-addresses are calculated as described above with the added caveat of repeating each pixel in the BRAM twice on the monitor.

The output of the lookup41x1259 BRAM *letter_pixel[15:0]* is used with *title_display* to create a properly pipelined time-delayed *letter_display_pixel[15:0]*. Depending on the value of the *show_title_screen* signal previously discussed, combinational logic decides whether *letter_display_pixel[15:0]* or black will be outputted.

Game Over Screen

Figure 10 below shows the game over screen: It consists of the letters "GAME OVER" centered across the monitor. The same logic used for the Start Screen was used here. Depending on the state the game is in, *vga_title* will output either *start_pixel[15:0]*, the Start Screen; *gameover_pixel[15:0]*, the Game Over Screen; or black, when the user is in the middle of a game. This is implemented through combinational logic.

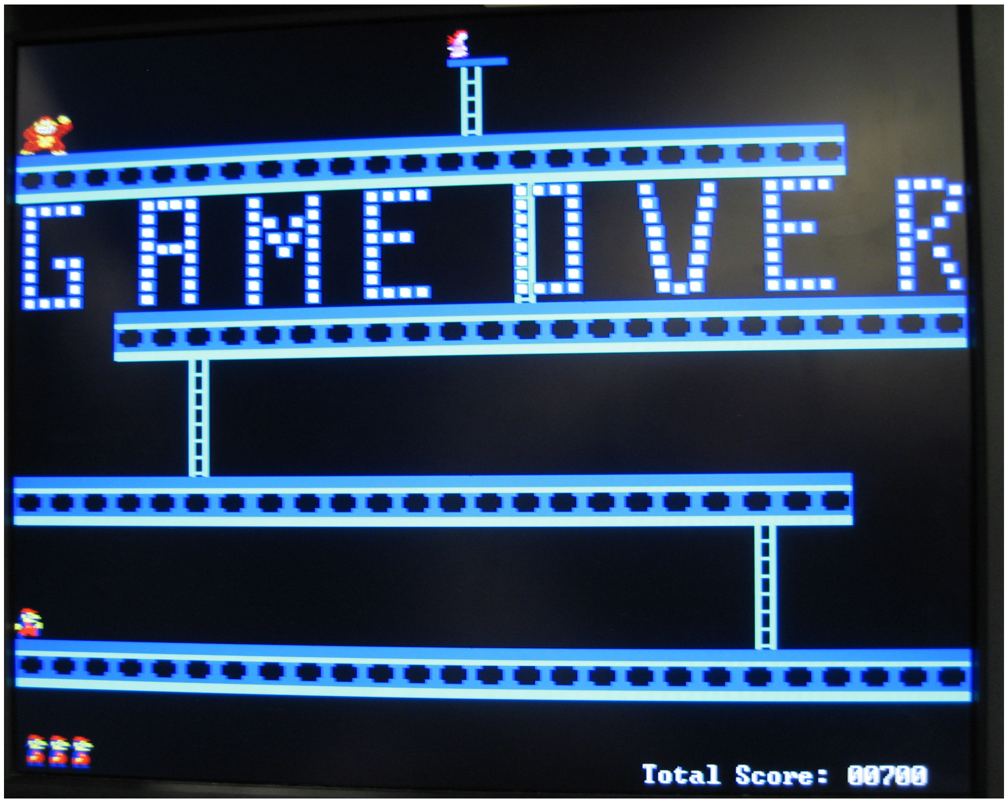


Figure 10. Game Over Screen Photograph

vga_ladder Module

The *vga_ladder* module generates the pixels for the ladders to be displayed onto the monitor. It takes as inputs the x and y locations of the top-left corners of the four ladders, and outputs the appropriate pixels to the *screenbuffer* module through the *ladder_pixel[15:0]* signal. The BRAM read-address computation process is exactly the same as described above. Please find below in Figure 11 the block diagram implementation of the module. Figure 12 is a copy of the picture that was loaded into the BRAM.

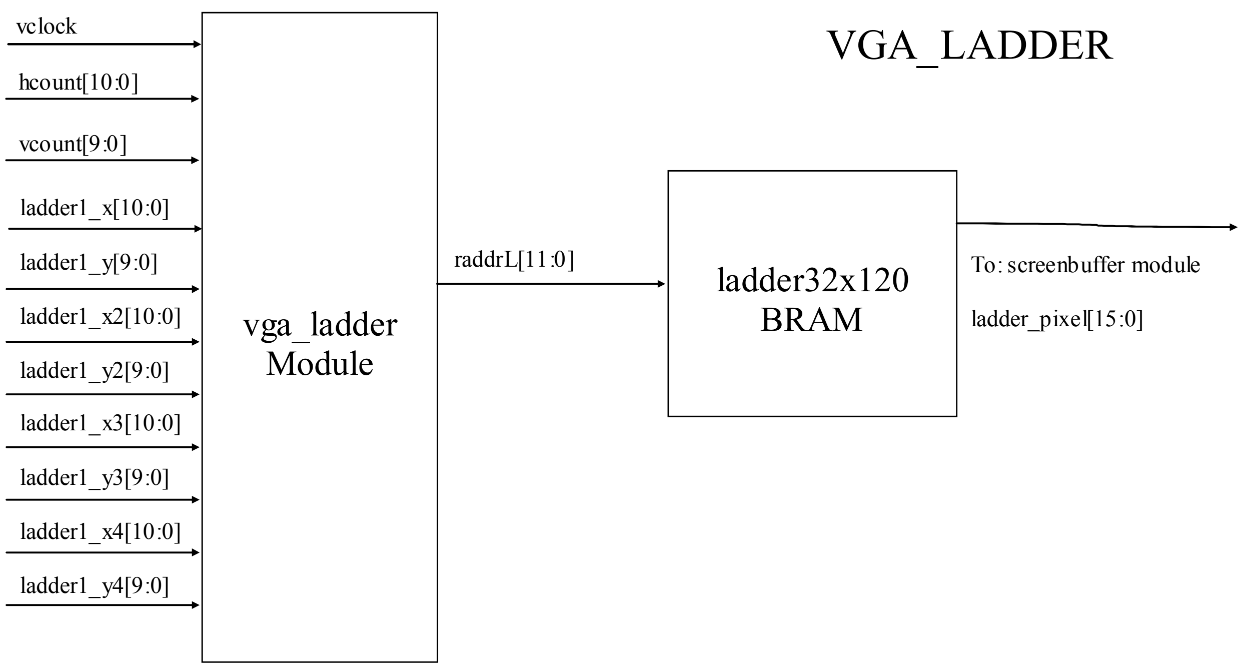


Figure 11. *vga_ladder* Module Block Diagram

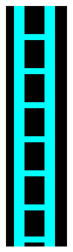


Figure 12. Ladder BRAM

vga_platform Module

This module is slightly more complex than the *vga_ladder* module because the platforms are different depending on the level the player has reached. Please refer to Figure 13 for the block diagram.

For all levels, the pixels are read directly from the BRAM. Please refer to Figure 14 below for a picture of what was loaded into the BRAM. The different color assignments of the platforms for each Game Level occur in the *screenbuffer* module. The three top platforms which do not cover the span of the monitor width are 904 pixels wide. In order to save BRAM space, I decided to use repeated units which are 512 pixels wide.

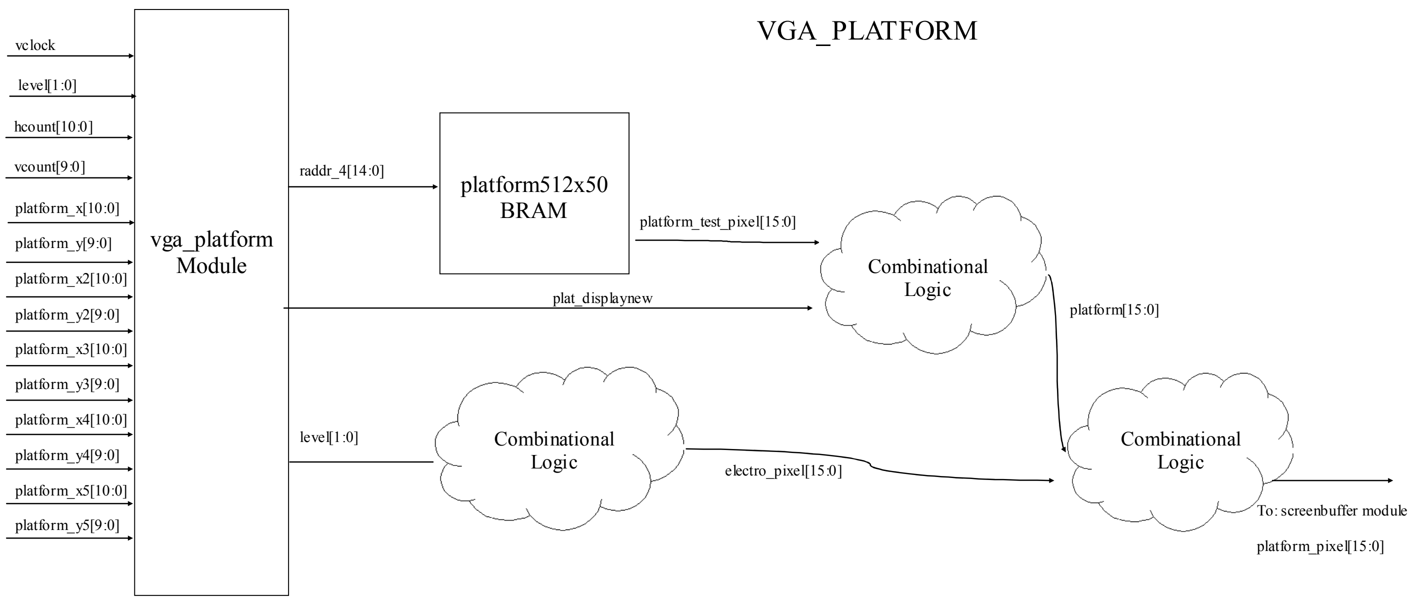


Figure 13. *vga_platform* Module Block Diagram

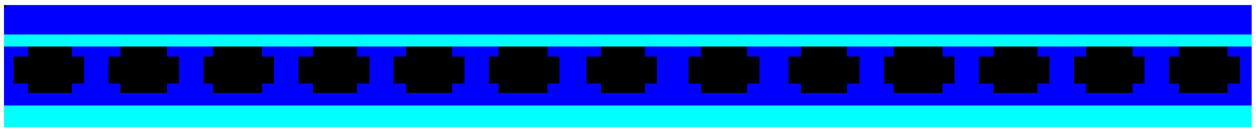


Figure 14. Platform BRAM

For the third level, we implemented three “electrocution” regions. They are represented on the monitor by a thin line of red that is 10 pixels wide. As previously described, when Mario touches these regions, he instantaneously loses a life. These electrocution regions were implemented with simple if-statements. Within a certain range specified by the locations of the electrocution regions, *vga_platform* module outputs a dark red pixel. This *electro_pixel[15:0]* is fed into an OR gate which outputs *platform_pixel[15:0]*.

vga_moving Module

The *vga_moving* module generates the pixels for Mario and the 5 barrels to be outputted onto the monitor. Please refer to Figure 15 for pictures of what was loaded into memory. The top row shows the different colored barrels. Each different color represents a barrel of different speeds. The bottom row shows Mario’s different animation frames. Going from left to right, we have Frame 0 - Mario_walk; Frame 1 – Mario_stand; Frame 2 - Mario_jump; and Frame 3 – Mario_climb.

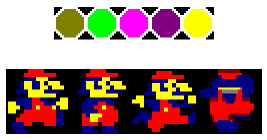


Figure 15. Barrels and Mario BRAMS

In addition to the already introduced inputs *vclock*, *hcount[10:0],* and *vcount[9:0],* this module takes in the (x,y) coordinates of the top-left corner of each of the 5 barrels, and Mario. It also receives from the Game Logic *mario_frame[1:0]*, which specifies the particular Mario animation frame to be displayed, as well as *go_left*, which is high when Mario is moving towards the left. Please find below the detailed block diagram for this module in Figure 16.

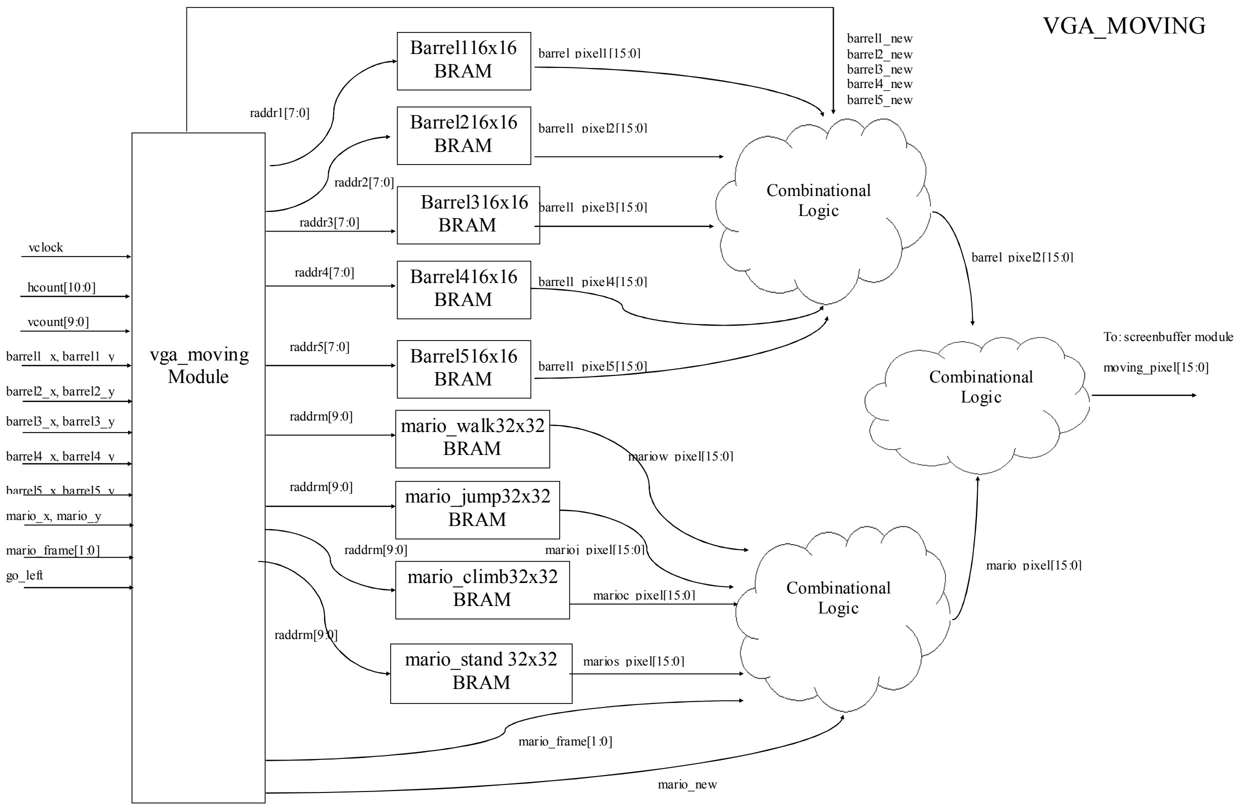


Figure 16. *vga_moving* Module Block Diagram

Timing becomes crucial in this module more so than the rest because of the constant location changes of both Mario and the barrels. A detailed discussion on the timing issues will be provided in the Testing and Debugging section.

Although this module's block diagram looks more complicated, it is in fact very similar to the two previously discussed. A read address is calculated with the (x,y) location of each pixel. A pixel output is obtained from the BRAM, which is then sent to a block of combinational logic which uses the signals *barrel1_new, barrel2_new, barrel3_new, barrel4_new, barrel5_new*, and *mario_new* to determine the appropriate time to output *moving_pixel[15:0]* to the *screenbuffer* module. Please refer to the above discussion on "Three-Stage Pipeline Implementation" for a detailed discussion on the logic behind the timing delays.

vga_stationary Module

The *vga_stationary* module generates the pixels for Donkey Kong, Peach, and Mario's life counter. This module takes as inputs *vclock, vsync, hcount[10:0], vcount[9:0], dk_x[10:0], dk_y[9:0],dk_frame[1:0], peach_x[10:0], peach_y[9:0],* and *life_count[1:0]*. The block diagram is found in Figure 17.

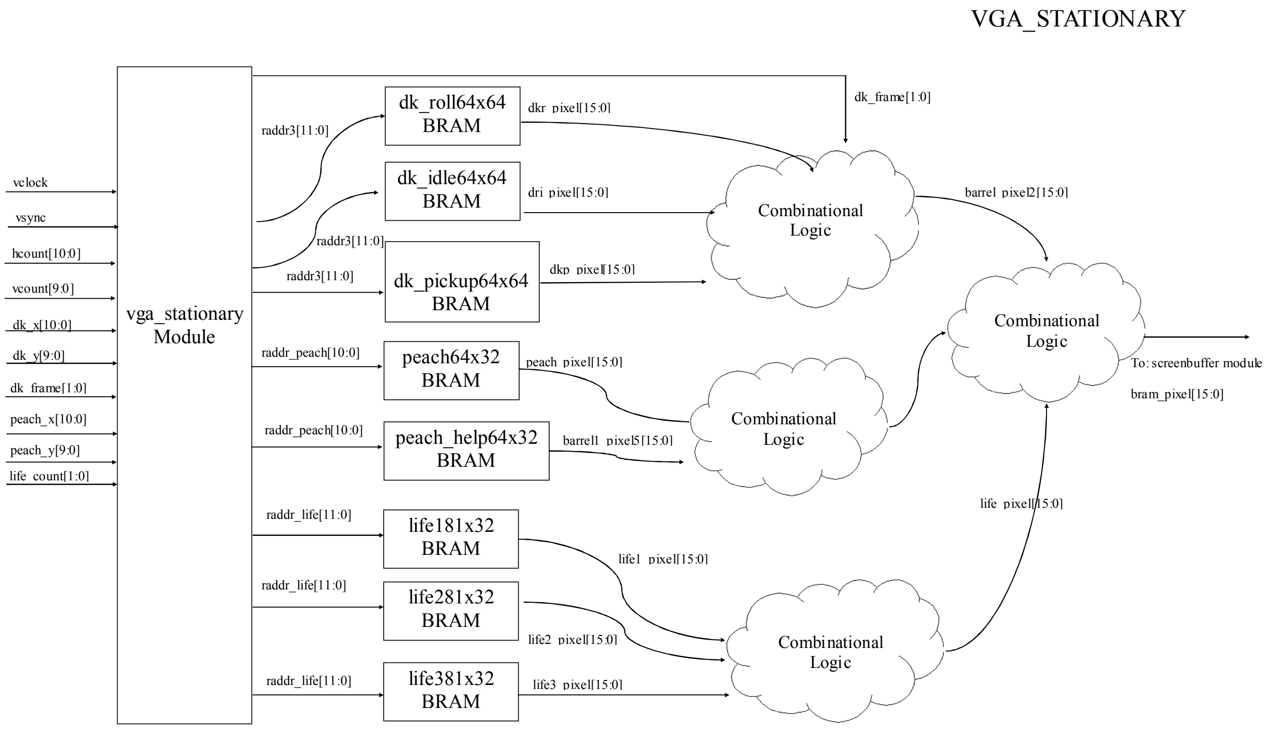


Figure 17. *vga_stationary* Module Block Diagram

For Donkey Kong's animation, similar to the previous modules, the frame outputted onto the screen depends on inputs from the Game Logic. In other words, *dk_frame[1:0]* determines whether Donkey is seen rolling a barrel, standing idle, or picking up a barrel. Going from left to right in Figure 18: Frame 0 - DK_idle; Frame 1 – DK_pickup; and Frame 2 - DK_roll.

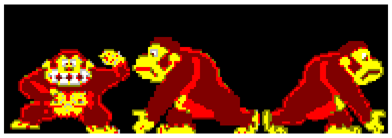


Figure 18. The three Donkey Kong animation screens stored into BRAMs.

For Peach's animation, I used a 6-bit register to count off approximately half a second. Approximately every half a second, the two Peach frames seen in Figure 19 are interchanged. The implementation is as follows: The 6-bit counter is incremented at every negedge of vsync. Because there are 60 vsyncs in a second, the counter can be used as a second counter.

Additionally, before the counter is 6bits wide, once you get to 63, 63+1 makes you loop back to the beginning again. A mux can thus be used to select whether to display *peach_pixel[15:0]* or *peachhelp_pixel[15:0]* depending on the value of the 6bit counter.

Figure 19. The two Peach animation screens stored into BRAMs

Finally, Mario's life counter shows Mario's Health status by using three mini-Marios located on the left bottom corner of the screen. Please see Figure 20. A mux selector is used to output the appropriate life pixel. For example, when *life_count[1:0]* has a value of 2, then *life2_pixel[15:0]* is selected to be outputted towards the *screenbuffer* module.

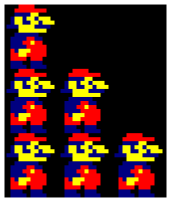


Figure 20. The three Mario Lives graphics stored into BRAMS.

***screenbuffer* Module**

The *screenbuffer* module is the "hub" of the display logic. It collects all the pixel outputs from the modules discussed above, and determines which ones to display pass onto the XVGA at any given time. Please refer to Figure 21 for its block diagram.

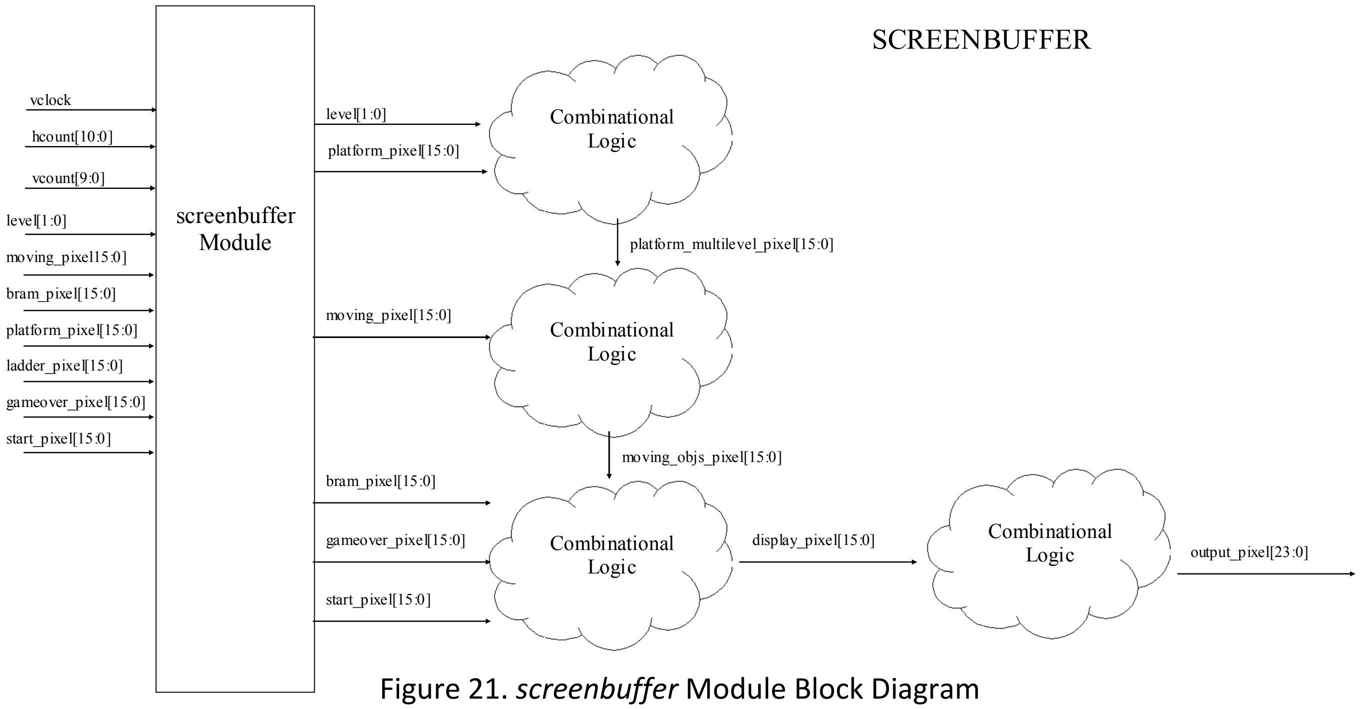


Figure 21. *screenbuffer* Module Block Diagram

In addition to the global *vclock, hcount[10:0], and vcount[9:0]* signals, the *screenbuffer* module requires the inputs *level[1:0]; moving_pixel[15:0],* an output of *vga_moving*; *bram_pixel[15:0]*, an output of *vga_stationary*; *platform_pixel[15:0]*, an output of *vga_platform*; *ladder_pixel[15:0]*, an output of *vga_ladder*; and *gameover_pixel[15:0]* and *start_pixel[15:0]*, outputs of *vga_title*.

As can be seen in the block diagram in Figure 21, the module first determines which platforms to display depending on the current level. Thus, given a different *level[1:0]* signal, the platforms will be different colors. The signal p*latform_multilevel_pixel[15:0]* represents the platform corresponding to the current level. Then, I needed to account for the possible overlapping of the barrels and Mario with the ladders and platforms. I decided to let Mario and the barrels take precedence. Thus, whenever *moving_pixel[15:0]* has a value other than 0, *moving_objs_pixel[15:0]*, the signal being passed to the XVGA, is assigned the value of *moving_pixel[15:0]*. Otherwise, *moving_objs_pixel[15:0]* is assigned the value of *platform_multilevel_pixel[15:0]* instead.

If game is over, *gameover_pixel* should be displayed. Otherwise, if *start_pixel* (DONKEY KONG title screen) has a value other than zero, it means it should be displayed. Otherwise, the regular game screen (platforms, ladders, Mario, DK, Peach, and barrels) should be displayed. The output of this last set of combinational logic is *display_pixel[15:0]*.

The final step is to convert this 16-bit color into the 24-bits necessary for display onto the monitor. Table 2 shows the conversion chart I used. Adobe Photoshop was the main tool used for this section because its color palettes gave the hex-code for the colors. After the bit-size conversion, *output_pixel[23:0]* is finally outputted to the XVGA.

16 Bit Color	24 Bit Color
Overall Game Colors	
16'h0000	24'h000000;
16'h0001	24'h800000
16'h0003	24'h808000
16'h0004	24'h000080
16'h0005	24'hE948FC
16'h0007	24'h808000
16'h0008	24'h808080
16'h0009	24'hFF0000
16'h000A	24'h3DFF33
16'h000B	24'hFFFF00
16'h000C	24'h0000FF
16'h000D	24'hFF00FF
16'h000E	24'h00FFFF
16'h000F	24'hFFFFFF
Level 2 Platform Colors	
16'h0060	24'hEB0652
16'h00F8	24'hEB81F8
Level 3 Platform Colors	
16'hFFC0	24'hFFC000
16'hCFCA	24'hCFCACA
Level 3 Electrocution Zone Color	
16'hE118	24'hE11809

Table 2. 16-bit to 24-bit Color Conversion

vga_timer Module

The *vga_timer* module generates a 30 seconds countdown timer display for Level 2 of our game. Please refer to Figure 22 for its block diagram.

The module takes as input *reset, vclock, hcount[10:0], vcount[9:0], level[1:0]* and *time_left[5:0]*. In order to display the time onto the monitor, I needed to find a way to convert the binary *time_left* into decimals since the *char_string_display module* downloaded from the course website only took in strings of characters and decimals as inputs. After some searching, and asking around the lab, I decided to use a *binary-to-bcd* module found on the OPENCORE.ORG site. [3]

[3] http://www.opencores.org/projects.cgi/web/binary_to_bcd/overview

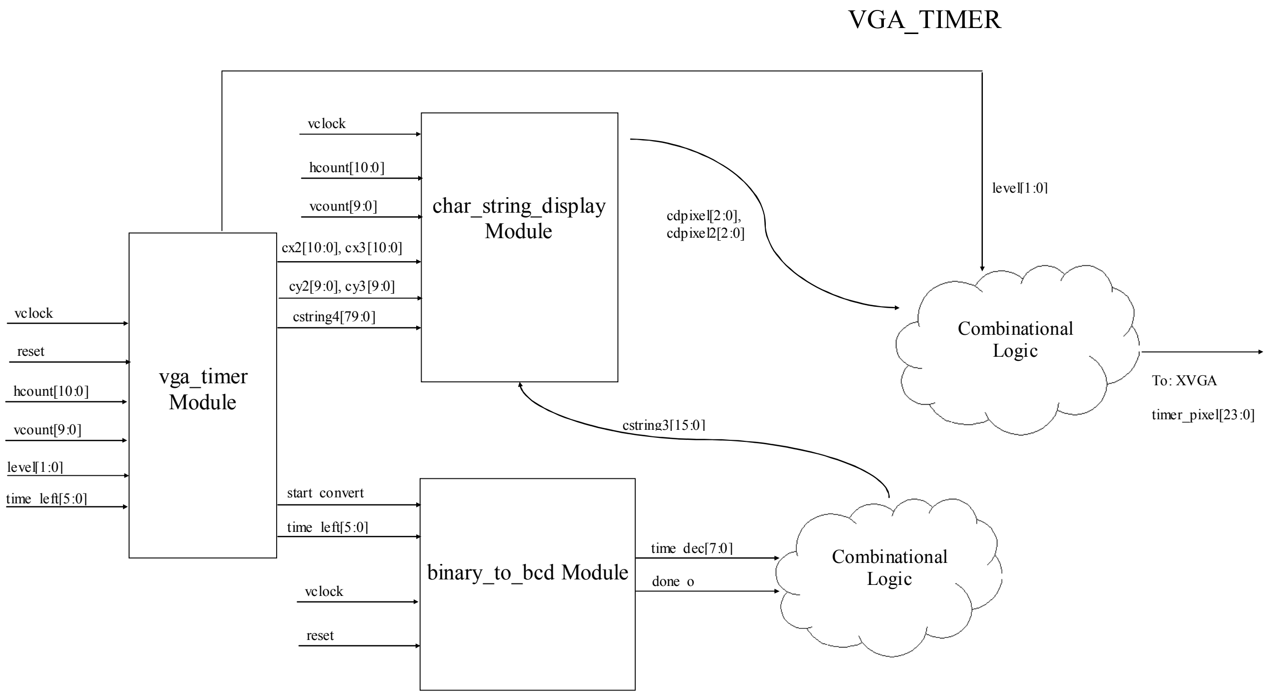


Figure 22. *vga_timer* Module Block Diagram

After every frame load, the module generates a *start_convert* pulse which starts the binary-to-decimal conversion process. When the conversion is done, the *binary-to-bcd* module outputs a signal *done_o* which signals that the output *time_dec[7:0]* is now stable to use. Each *time_dec[7:0]* output represents one digit. Since our countdown of 30 seconds require two digits, two *time_dec* signals are concatenated together to form *cstring3[15:0]*, which is then fed into the *char_string_display* module. This *char_string_display* module then converts the sequence of two digits containing the countdown seconds into pixels ready to be outputted onto the screen. This output is called *cdpixel2[2:0]* in the block diagram in Figure 22.

Also in Figure 22, *cstring4[79:0]* is the string representation for “Time Left:”. Its pixel representation is called *cdpixel[2:0]* in the block diagram. If the current user is currently on level 2 of our game, then the *cdpixel[2:0]* and *cdpixel2[2:0]* signals are passed through an OR gate, concatenated into 24-bits, and outputted to the XVGA.

vga_score Module

The *vga_score* module generates a pixel representation of Mario’s game score to be displayed on the monitor. Please refer to Figure 23 for its block diagram.

Its structure is very similar to that of *vga_timer*. Whereas in that module, the desired output was “Time Left: 00”, here, the result was “Total Score: 00000”. The change from 2 numerical digits in the former to 5 in the latter added great propagation delays to the *binary_to_bcd* module. As a result, the timing was affected, and the last digit of the score was missing 2 columns of pixels, while the first digit of the score had 2 columns too many in the front. An attempt was made to pipeline the paths with registers to allow for more computation time, but the result was unsuccessful. The final solution implemented outputs 6 digits to the monitor, and disguises the glitch by covering the last digit and the first two columns of the first digit with black. The player cannot in any way tell a difference.

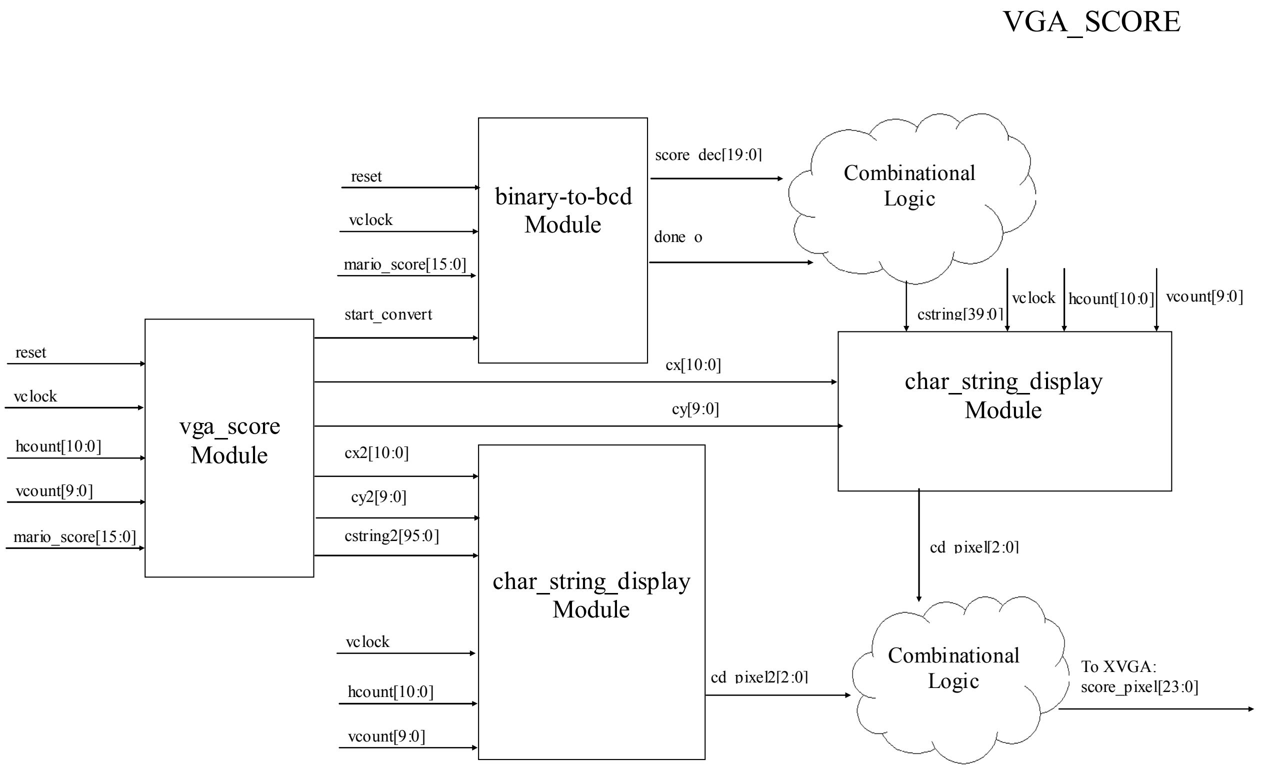


Figure 23. *vga_score* Module Block Diagram

Keyboard Interface

The Keyboard Interface is the interface the user will use to communicate with the game. The interface consists of two modules: the PS2/Keyboard Driver and ASCII Converter module and the ASCII-to-Control Signal Converter module. The user has a keyboard, which is connected to the FPGA lab kit. When the user presses certain keys on the keyboard, it will move Mario in the appropriate direction. Figure (INSERT HERE) shows the block diagram of the Keyboard Interface.

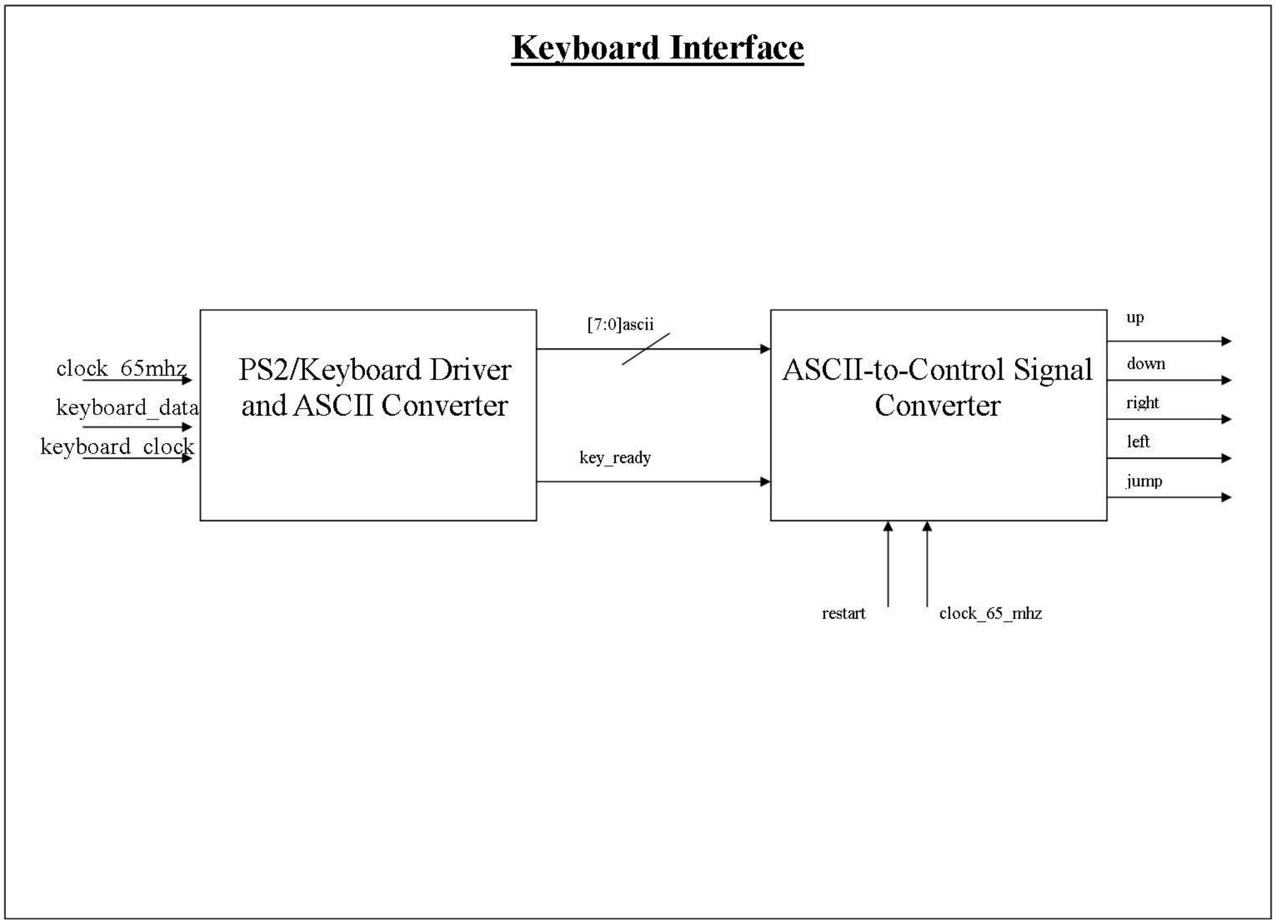


Figure 6. Block Diagram of Keyboard Interface

PS2/Keyboard Driver

There are two modules which compose this module, which was originally created by Professor Terman and I. Chuang. The keyboard sends several signals to the FIFO buffer keyboard driver of this module. Using the keyboard clock from the lab kit, and keyboard sends specific codes to the driver. The buffer extracts those codes from the keyboard and places the relevant data into the FIFO buffer. When the FIFO buffer is empty, the *fifo_empty* signal is transmitted to the PS2 module. The FIFO buffer is continuously read by the higher level PS2 module. The PS2 module takes in the keycode data from the buffer and converts it to ASCII code that the FPGA can understand. The PS2 module outputs the converted ASCII code. If the FIFO buffer is empty, the previous ASCII value is continuously outputted until a new keystroke gets registered into the buffer. Because there was no desire to have Mario continuously jump up and down when the user has already let go of the button, the PS2/Keyboard Driver was slightly hacked and modified. Not only does the converted ASCII code get outputted, but also the *key_ready* signal. The *key_ready* signal is the inverse of the *fifo_empty* signal. It was discovered that when the FIFO buffer has data in it, the driver does not continuously output a steady *key_ready* signal as intuition would have it. In reality, when the buffer continuously has data in it, the driver outputs pulses, not a steady high voltage level step function.

ASCII-to-Control Signal Converter

The ASCII code, the *key_ready signal,* and the video clock are all inputs into the ASCII-to-Control Signal Converter module. This module will take those two signals and ultimately output the appropriate signals that Mario Logic can understand. An implementation was desired that would make Mario jump only once, when the user pressed the jump button.

Without any hacking around the *key*_ready signal and having just a pure look-up-table converter, Mario would continuously execute the action that it was assigned to do, even though the user stopped pressing a key. The module basically consists of several conditional statements that convert the ASCII code to actual signals the Mario Logic will understand. Having Mario walk in the same direction was acceptable, but it was decided that it was annoying that Mario would continuously jump after the user told Mario to do so. In order to fix this nuisance, a design was conceived that centered on the *key_ready* signal.

It was discovered that the *key_ready* signal are pulses that are separated about 100ms apart when the buffer of the driver has something always in it. Observing this oddity of the system, it was decided that having a key counter in the module would solve the problem. The video clock would be use to count up to 100ms by using the key counter. If the key counter is greater than the value that equates to the interval between the pulses of the *key_ready* signal when the driver's buffer has something in it, then the logic knows that the buffer is empty. When it knows that the buffer is empty, it tells itself to only output the jump signal once, and then not move again until you get another keystroke. In order to make Mario move again, the key counter is always initialized back to zero when the *key_ready* signal is high. In summary, if the *key_ready* signal does not come in a certain amount of time, the logic knows that it is empty and will output the signals accordingly.

<u>Testing and Debugging</u>

<u>*Game Logic Testing and Debugging*</u>

Precision is important

Debugging and testing the Game Logic was extensive and exhaustive. The most significant problem that I ran into is getting the conditional statements exactly correct. Because I was using the entire screen as my coordinate plane for the various objects in the game, getting the logical conditional statements exactly right was extremely difficult. At initial glance, it seemed logical easy to just have a set of statements checking to see if the object meets a certain condition by comparing the pixel coordinates, and if so, execute that specific action. However, I soon learned that being off by even a single pixel proved to be disastrous. For instance, one time I had a statement to check to see if Mario's coordinates were at the edge of a platform. Instead of comparing to see if his coordinates were greater than the platform's edge pixel coordinate, I checked to see if it was greater than or equal to it. This would cause Mario to freeze while at the edge of a platform, instead or falling. Other bugs included Mario not moving or Mario going through platforms. These bugs were extremely frustrating because usually I had to recompile the code and check manually onscreen, by controlling Mario. The majority of these "coordinate bugs" were very subtle and, at times, very frustrating to fix. I had to methodically check every conditional statement in several modules to make sure I did not actually typed in, for instance, a less than sign instead of the greater than or equal sign, or to make sure that I did not compare Mario's coordinates to the coordinates of (800,900) instead of (801,900). Debugging these kinds of glitches and bugs was time consuming and tedious.

Forget what you learned in grade school

Another, surprising mistake that I would make numerous times is forgetting that the Y coordinates onscreen increases from top to bottom. After having beaten into my head since grade school that Y coordinates positively increase the further you go up, it was difficult to remember that for the coordinates onscreen, the opposite is true. Sometimes hours would be wasted because I did not realize that the Y values increased while going down. Becoming confused, I would compare if a Y coordinate value is greater than another value instead of less than another value. The majority of the time I would just continuously compile the code onto the FPGA and see what would happen onscreen and go from there.

The Logic Analyzer is your friend

By having all of these modules communicating with each other, timing is crucial. Initially, when I started this project timing was not a worry of mine. However when I started to connect my modules together, it immediately became a significant problem. Data from one module could be delayed by one cycle, or data from another module could be high for more than one cycle. The lists of problems that could occur with regards to timing did occur.

I was initially hesitant to use the logic analyzer because of the fear that it would be too complicated and inefficient to output the various signals to it. I was wrong. Without using the logic analyzer I would most likely not have been able to complete the project. The logic analyzer solved two of my major problems I had with regards to timing.

When testing the Game FSM, I noticed that it would transition from Level 1 to the Game Over state when Mario reached the princess, instead of transitioning to Level 2. I had an idea what was going on (I thought the *collision_princess* signal was high for too many clock cycles), but I was forced to use the logic analyzer to determine when and how long the *collision_princess* was high for and to determine how the state transitioned giving this signal. My hypothesis proved to be correct, and the logic analyzer specifically showed me where the problem was. After finding the bug, I immediately added another condition for the state machine and the bug was fixed.

Yet another major problem I had with timing occurred when I was trying to interface with the keyboard to make Mario only jump when the user told him to jump. I spent several days trying to figure out why the keyboard was not working properly with my Game Logic. Finally, it was suggested to me that I output all the various keyboard interface signals to the logic analyzer. After outputting the signals to the logic analyzer, the problem was quickly found. The keyboard driver outputted its keycode data one clock cycle after it told me that it had data in its FIFO buffer. Knowing how the driver actually worked, the bug was easily fixed by making my module wait one clock cycle later after it received the *key_ready* signal.

The bugs and problems mentioned in this section are the main ones that I encountered, but there were other minor bugs that were annoying to find (i.e., making sure which part of the logic circuit needed to take precedence of others), but they were easily fixable. When connecting both the Game Logic and Video Logic together the easiest and most efficient way to find and correct bugs was to recompile the game, and play it until something went wrong.

Display Logic Testing and Debugging

The most challenging and time-consuming process of the project was the testing and debugging process. It was also the most enriching part. The main take-away is the importance of really knowing the ins and outs of all the timing diagrams of all the modules. For example, if only one platform_pixel glitches, even though the rest of the modules met the timing specs, the whole screen can still be affected. In this section, I will discuss some of the problem I encountered, and how I went about solving them.

Early Lessons

I learned early on the patience and attention to detail that is required to successfully test and debug subtle errors in code which can have devastating consequences. My first goal was to read from a BRAM. The task seemed simple enough. It's just like Lab 4, only pictures instead of sound, right? Not so simple. During my first couple of attempts, my barrel did not show up on the monitor. To debug, I used the top-level file from the Pong game. I experimented with the switches to make sure that at least *something* was displaying. Instead of reading the whole BRAM, I tried assigning the whole screen to one particular location of the BRAM. The color happened to be white. Still nothing showed up. I then tried to assign the whole screen to be white directly. Again, nothing showed up. Several LAs and Gim puzzled over the problem with me for 4 straight nights in lab. Finally, it was discovered that pblank was not synchronized with vsync and hsync.

This experience taught me to be more flexible and imaginative when it came to the debugging process. When stuck on a problem for several days, it is important to step back and think about how relevant a blank screen not showing up is to my project. Indeed, the bug had no effect on my initial problem which actually stemmed from a completely unrelated issue. One week wasted! Although there is satisfaction derived from *finally* figuring out what was going wrong, something needs to be said about efficiency and the reality of the amount of time and resources available. Throughout the rest of the weeks, whenever I encountered strange, tedious bugs, I tried to ask myself questions like, "Is what I'm looking at right now actually helpful? Am I going on a tangent? Is the answer I'm chasing after really relevant to my current problem of finishing the final project in time?" That first week definitely taught me a lot.

Integration with Game Logic

Up until Thanksgiving, I created a screenbuffer through the ZBT interface to load my graphics onto the XVGA. Because of the read-write cycle delays for data transfer both to and from the ZBT, I experimented with many tricky timing issues. When Thanksgiving came along, I finally got all the graphics to display on screen without glitching edges. I wrote a test code to make Mario, and each of the barrels move across the monitor to test the timing of my system. I thought I was done with the display, and would be able to start working with the video interface for

camera-controlled motion. Oh, the inexperienced thoughts of someone 3 weeks into the final project and still naïve about the design and implementation process of a digital system…
After fixing the initial integration errors of mismatched variable names, multi-sourcing signals, and wire declarations, I found my display logic glitch like I've never seen before. Prior to integration, all my glitches were only lines of pixels misplaced due to timing errors. This time, the whole screen was off. Seemingly random pixels were displaced, and the timing delays seemed random to me. While talking to the staff, a similar theme began to emerge. They all asked me, "Why don't you just output the pixels directly onto the screen instead of loading it up into the ZBT?" At first I was reluctant because I had spent the past 3 and a half weeks getting the frame buffer running. So I spent the rest of the week trying to fix the ZBT timing issues. Finally, I decided to let the ZBT rest, and changed everything to display through combinational logic. With the first compilation, the number of glitches went down drastically. My display was once again back to being off by only a pixel or two. I learned my second important lesson of the term: when it comes to being stubborn versus being efficient, I should give efficiency more weight than I have in the past.

Timing Diagrams, Timing Diagrams, Timing Diagrams

After the switch to combinational logic, I soon learned that subtle mistake and glitches are much harder to fix than simple ones. Now came two weeks of timing diagrams. All the LAs and TAs were very helpful and patient with my questions. The first huge error I had in logic was uneven pipelining. It's been drilled into our heads many times: When you pipeline a path, you must add registers to every signal the line crosses. However, when it comes to actual implementation, it seems easy to forget. All my graphics were off by 1 pixel. For example, my barrels were only closed on one side; the 1-pixel thick right-side boundary was not displayed. The problem was I had pipelined the calculation of the read-address into BRAM, but had failed to pipeline the pixel output of the BRAM. Because the pixel outputs were delayed by three cycles (two cycle delays for read-address calculation, and one cycle delay for BRAM access), I was matching up the pixels with the wrong locations on screen. Thus, I spent most of one week drawing both block and timing diagrams in an attempt to figure out answers to 1. How many cycles of delay there are, and 2. Which paths need to be included in the pipeline?

A Xilinx tool I found particularly useful in this process was the RLT schematics tool. Figure 24 shows a screenshot of a part of my *vga_timer* circuit. These schematics are very useful because they serve as great guides to figuring out what each signal was connected to, where all the clocked registers are, which paths are crossed by the same pipeline, etc.

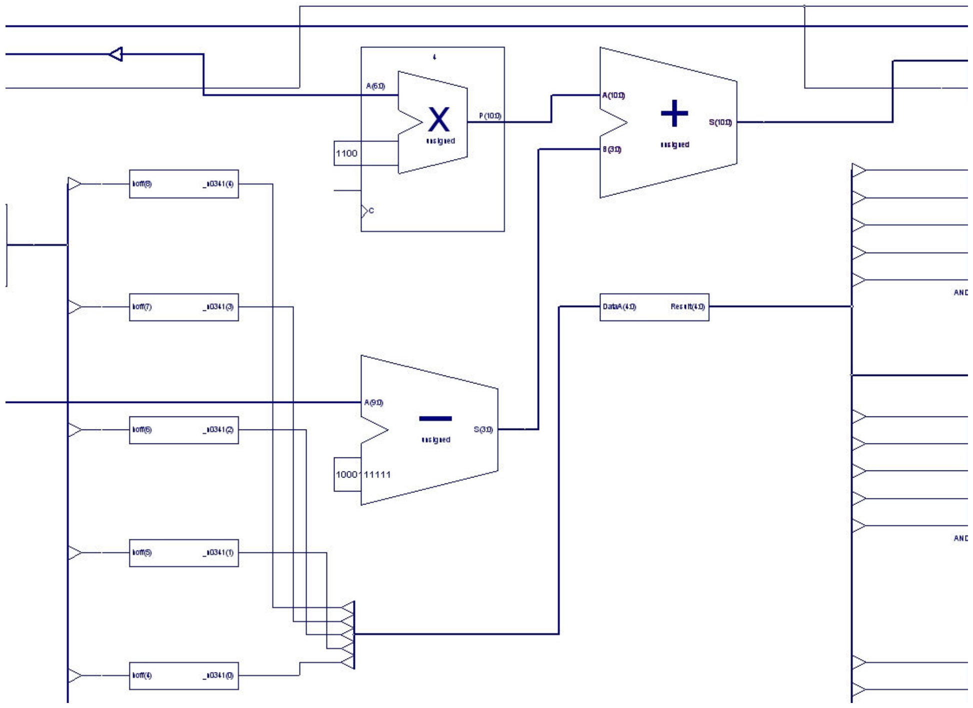
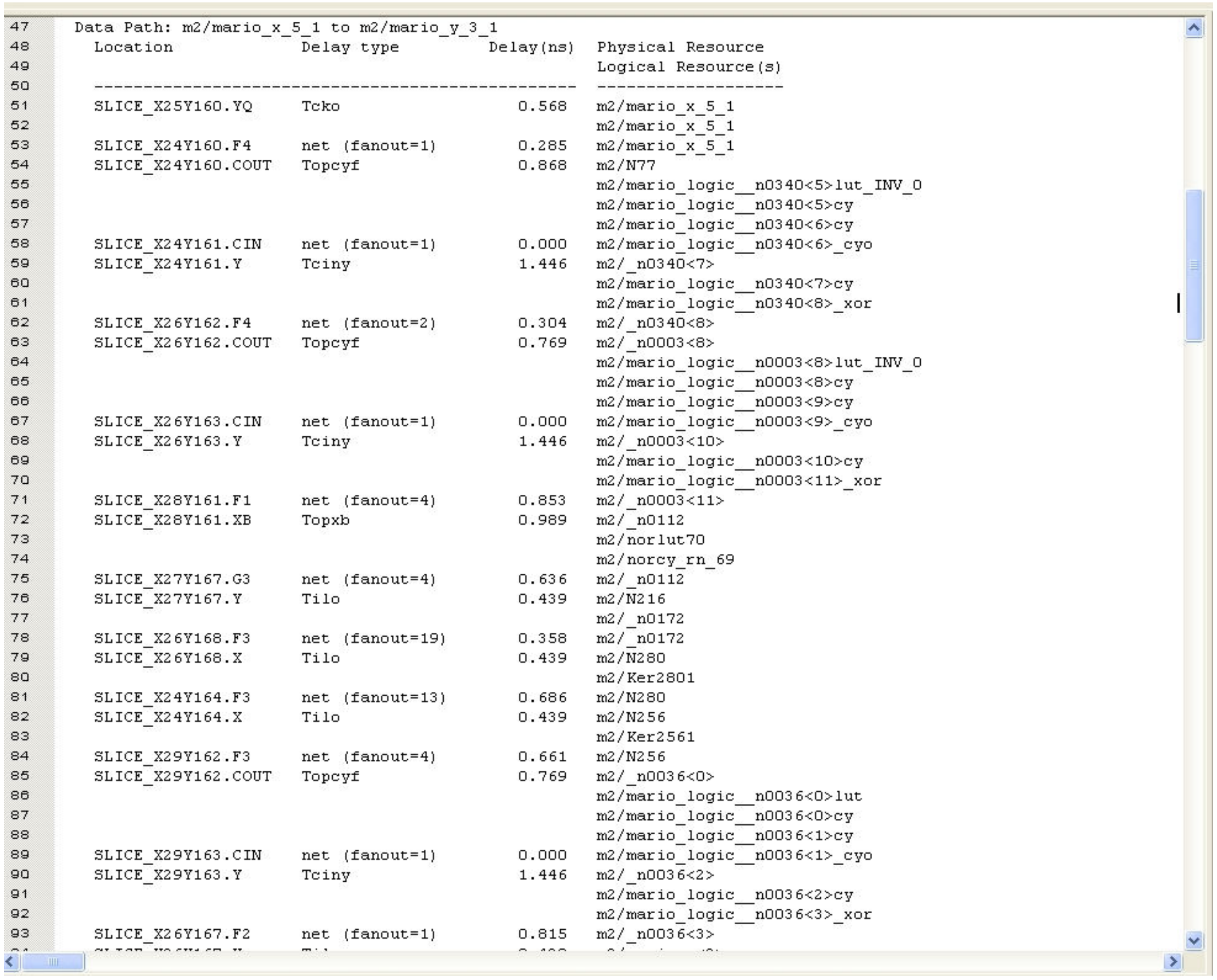


Figure 24. RTL Schematic Screenshot

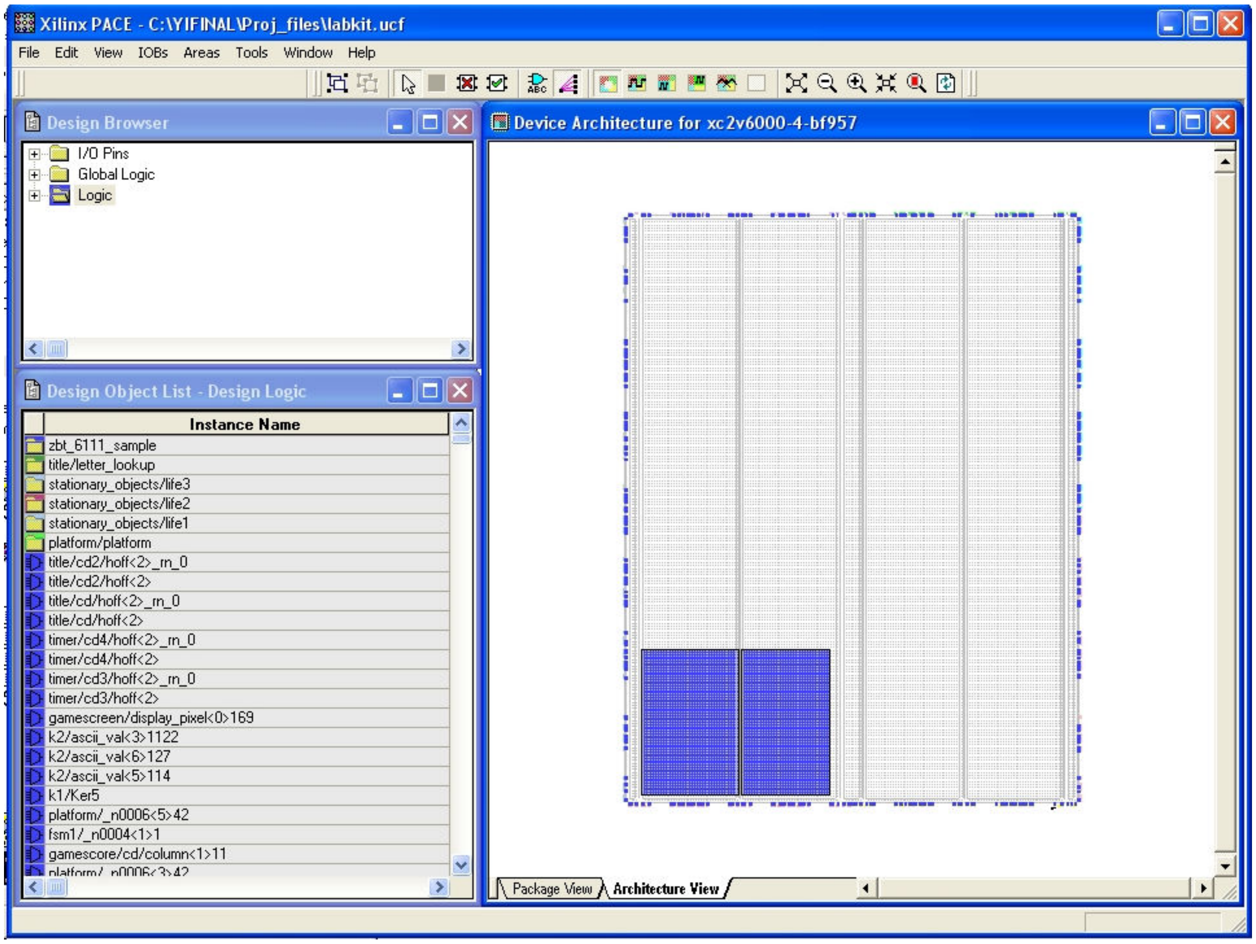
After making what I thought were the appropriate changes, I still had several glitches with the character string display. A tool I found very useful in finally solving my timing issues was the post place and route static timing report. Figure 25 shows a screenshot of this tool. This report lists the three worst-case timing violations of the system. Lo and behold, my problem was found. My character display circuit for *vga_timer* took over 18ns. This is well over the clock cycle of around 15ns. The breakup of how many nanoseconds each segment of the circuit takes helped me to clearly identify where to pipeline. The issue was determined to be the direct assignment of hoff and voff in the *char_string_display* module. I modified to code to assign them on the positive edge of vclock instead, and that solved my timing issues. Please refer to the appendix for the detailed Verilog code.

Finally, the last problem I encountered was the probabilistic nature of my DONKEY KONG start screen actually showing up. If I got lucky during compilations, it would show up. If not, then I was out of luck. Because the post place and route static timing report did not show any timing errors, the problem was determined to be one of routing. In order to solve the problem, Cassie and Daniel, another student in the class, helped me create area constraints on the labkit. Please see figure 26 for a screenshot of the process. Constraining the area should minimize the routing times and solve the random occurrences of my DONKEY KONG start screen. Indeed it did, and my last big bug was found.



```
Data Path: m2/mario_x_5_1 to m2/mario_y_3_1
  Location             Delay type         Delay(ns)  Physical Resource
                                                     Logical Resource(s)
  -------------------------------------------------  -------------------
  SLICE_X25Y160.YQ     Tcko                   0.568  m2/mario_x_5_1
                                                     m2/mario_x_5_1
  SLICE_X24Y160.F4     net (fanout=1)         0.285  m2/mario_x_5_1
  SLICE_X24Y160.COUT   Topcyf                 0.868  m2/N77
                                                     m2/mario_logic__n0340<5>lut_INV_0
                                                     m2/mario_logic__n0340<5>cy
                                                     m2/mario_logic__n0340<6>cy
  SLICE_X24Y161.CIN    net (fanout=1)         0.000  m2/mario_logic__n0340<6>_cyo
  SLICE_X24Y161.Y      Tciny                  1.446  m2/_n0340<7>
                                                     m2/mario_logic__n0340<7>cy
                                                     m2/mario_logic__n0340<8>_xor
  SLICE_X26Y162.F4     net (fanout=2)         0.304  m2/_n0340<8>
  SLICE_X26Y162.COUT   Topcyf                 0.769  m2/_n0003<8>
                                                     m2/mario_logic__n0003<8>lut_INV_0
                                                     m2/mario_logic__n0003<8>cy
                                                     m2/mario_logic__n0003<9>cy
  SLICE_X26Y163.CIN    net (fanout=1)         0.000  m2/mario_logic__n0003<9>_cyo
  SLICE_X26Y163.Y      Tciny                  1.446  m2/_n0003<10>
                                                     m2/mario_logic__n0003<10>cy
                                                     m2/mario_logic__n0003<11>_xor
  SLICE_X28Y161.F1     net (fanout=4)         0.853  m2/_n0003<11>
  SLICE_X28Y161.XB     Topxb                  0.989  m2/_n0112
                                                     m2/norlut70
                                                     m2/norcy_rn_69
  SLICE_X27Y167.G3     net (fanout=4)         0.636  m2/_n0112
  SLICE_X27Y167.Y      Tilo                   0.439  m2/N216
                                                     m2/_n0172
  SLICE_X26Y168.F3     net (fanout=19)        0.358  m2/_n0172
  SLICE_X26Y168.X      Tilo                   0.439  m2/N280
                                                     m2/Ker2801
  SLICE_X24Y164.F3     net (fanout=13)        0.686  m2/N280
  SLICE_X24Y164.X      Tilo                   0.439  m2/N256
                                                     m2/Ker2561
  SLICE_X29Y162.F3     net (fanout=4)         0.661  m2/N256
  SLICE_X29Y162.COUT   Topcyf                 0.769  m2/_n0036<0>
                                                     m2/mario_logic__n0036<0>lut
                                                     m2/mario_logic__n0036<0>cy
                                                     m2/mario_logic__n0036<1>cy
  SLICE_X29Y163.CIN    net (fanout=1)         0.000  m2/mario_logic__n0036<1>_cyo
  SLICE_X29Y163.Y      Tciny                  1.446  m2/_n0036<2>
                                                     m2/mario_logic__n0036<2>cy
                                                     m2/mario_logic__n0036<3>_xor
  SLICE_X26Y167.F2     net (fanout=1)         0.815  m2/_n0036<3>
```

Figure 25. Post Place and Route Static Timing Report Screenshot


Xilinx PACE - C:\YIFINAL\Proj_files\labkit.ucf
File Edit View IOBs Areas Tools Window Help
Design Browser
I/O Pins
Global Logic
Logic
Design Object List - Design Logic
Device Architecture for xc2v6000-4-bf957
Package View
Architecture View

Instance Name
zbt_6111_sample
title/letter_lookup
stationary_objects/life3
stationary_objects/life2
stationary_objects/life1
platform/platform
title/cd2/hoff<2>_rn_0
title/cd2/hoff<2>
title/cd/hoff<2>_rn_0
title/cd/hoff<2>
timer/cd4/hoff<2>_rn_0
timer/cd4/hoff<2>
timer/cd3/hoff<2>_rn_0
timer/cd3/hoff<2>
gamescreen/display_pixel<0>169
k2/ascii_val<3>1122
k2/ascii_val<6>127
k2/ascii_val<5>114
k1/Ker5
platform/_n0006<5>42
fsm1/_n0004<1>1
gamescore/cd/column<1>11
platform/_n0006<3>42

Figure 26. Device Architecture Screenshot

<u>Conclusion</u>

Stephen's Conclusion

Overall, the project was a resounding success, and both of us completed the majority of what we wanted to do. I learned a significant amount in the timeframe of creating our project, especially with respects to debugging. I initially did not know much about where the bugs were coming from in my project during the first week or so. Several weeks later, I generally made correct hypotheses about what was causing the various bugs in my modules. I also learned that it is foolish to try to connect everything at one time and expect it to work. It was much easier to take small chucks of the project and isolate the specific bug, instead of trying to debug an entire system at once. Another significant lesson learned is that the logic analyzer is one of the best tools to use to debug one's circuit. The logic analyzer allowed me to successfully complete my part of the project on time.

I learned that time management is important. Spreading the workload across several weeks instead of the last week of the project allowed us to complete the game more efficiently and with less stress. In addition, I learned that planning and designing a complex digital system is more crucial then the implementation part of it. I initially was skeptical when Professor Terman said that planning would take a significant amount of time, but he was right. I also learned that communication is important. Several times during the weeks we worked on the project, we each misinterpreted how our modules would interact with each other, therefore causing our game not to work properly. However, through clear communication we were able to solve these problems. In spite of all the success we had, like in life, there are also regrets.

I regret using the entire screen as my coordinate plane. Using this coordinate plane system made it extremely difficult to debug, and it made some of my code inefficient because of the all conditions I had to check by comparing coordinate values. Using this particular coordinate system also took up a significant portion of my time. I also regret not being able to get to the motion sensor part of the projects. We were finished with our project several days before it was due, and instead of working on the motion detection, we decided it would be more beneficial to add more levels to the game. If given several more days, I am sure we would have been able to implement the motion detection. It would have been interesting to move Mario with one's hand movements instead of using the keyboard. I would have liked to add more interesting barrel logic to make the barrels not only roll off the platforms, but also roll down the ladders.

Regardless of my small regrets, I am satisfied with the final outcome. We successfully created a version of the game Donkey Kong. Very few people can attest to making the game Donkey Kong from the ground up. Completing such a game in the timeframe given to us is something that both of us should be proud about.

Yi's Conclusion

This Donkey Kong final project provided me with an eye-opening experience into the design methodology that goes into building digital systems. The planning, coding, implementing, integrating, and especially the debugging process should prove to be very useful in future projects.

The first part of the implementation process consists of laying out a detailed design plan taking into account the interactions between each module of the greater system. This includes timing specs as well as how the various inputs and outputs are linked between modules. Breaking down the system into smaller, more easily manageable subparts was very effective in understanding the role each subsystem must take. The testing and debugging process became clearer and more easily defined as a result.

For this project, it was especially important to understand the relationships between the Game and Display logic. This includes details such as whether a particular signal is level signal, a pulse, etc. Otherwise, complications will surely arise during integration.

The project definitely drilled into my mind the importance of timing. Most of my glitches were results of asynchronous signals which I assumed to have been synchronous, calculations which overextended the 15ns clock cycle, and mismatched pipelines. I learned a lot by working with the staff to figure out exactly how each signal is propagated through the circuit. The staff also helped me familiarize myself with many useful Xilinx tools. Overall, I learned a lot about software-based design and debugging tools.

One of the greatest takeaways from this project for me was the importance of fully understanding the timing specs of each input and output down to the single cycle of the clock. Digital design isn't a discipline where one can fudge the understanding, and hope to be lucky and come out with a working design. The minute details really make or break the big picture.

On a more personal level, as I've previously alluded to, the past six weeks have taught me a lot about problem solving in general. For example, the afore mentioned first week of debugging the pblank taught me the importance of being open-minded to new solutions and different approaches; the switch from ZBT to combinational logic taught me the value of cutting my loses while I still had the chance to even if it meant letting go of something I really wanted to see work. More importantly, although this may sound cheesy, these past weeks have really taught me the meaning of cooperation. I was surprised to see so many students want to take the time to talk to me about my bugs and offer me their help, even when they were under pressure to get their work done too. For example, at the very beginning, the Python BMP-to-COE converter provided on the course website did not meet the specifications I needed for my graphics. It was Yunjie who told me about the Matlab code she had found online. When my converted COE files wouldn't load correctly into the BRAMs, it was Kevin who taught me the trick of opening it first in Notepad to change two lines. When I couldn't figure out how to convert my *mario_score* from binary to decimal, it was Daniel who helped point me to the OPENCORE.org website. In retrospect, I don’t think the experience would have been the same had it not been with this group of people.

Appendices

MATLAB BMPtoCOE

```
function BMPtoCOE(image_name)
%Converts a 16 color bitmap image to a Xilinx .COE file
%Was written so students could use a FPGA to display images on a VGA
%monitor

%read bmp data in and display it to the screen
[imdata,immap]=imread(image_name);
image(imdata);
colormap(immap);
numpixels=numel(imdata);

%create .COE file
COE_file=image_name;
COE_file(end-2:end)='coe';
fid=fopen(COE_file,'w');

%write header information
fprintf(fid,';*****************************************************************\n')
;
fprintf(fid,';****              BMP file in .COE Format              *****\n');
fprintf(fid,';*****************************************************************\n')
;
fprintf(fid,'; This .COE file specifies initialization values for a\n');
fprintf(fid,'; block memory of depth= %d, and width=4. In this case,\n',numpixels);
fprintf(fid,'; values are specified in hexadecimal format.\n');

%start writing data to the file
fprintf(fid,'memory_initialization_radix=16;\n');
fprintf(fid,'memory_initialization_vector=\n');
%convert image data to row major
newimdata=transpose(double(imdata));
%write image data to file
for j=1:(numpixels-1)
   fprintf(fid,'%s,\n',dec2hex(newimdata(j)));
end
%last data value supposed to have a semicolon instead of a comma
fprintf(fid,'%s;\n',dec2hex(newimdata(numpixels)));
%clean shutdown
fclose(fid)
```