Galaxian Revolution:
Interactive Arcade Shooting Game

6.111 Fall 2006
Final Project Report

Team #27
Danh (Danny) Vo and Weijie (Jeff) Yuan

Mentoring TA:
Javier Castro

Abstract:
Galaxian Revolution represents a coalescence of the classic arcade game Galaxian and a novel hand motion based controller scheme. The project involves the design and implementation of three different module blocks: video processor to detect hand motion and translate into game commands, the game logic itself which governs the flow of gameplay, and finally the display output to the monitor. The successful integration of these independently designed blocks is also a crucial component of the project. Through five weeks of planning, designing, implantation, and integration, the project is fully functional and can be enjoyed by adults and children of all ages.

Table of Contents

Table of Figures

1. Overview

Galaxian is a classic arcade game released by Namco in 1979. The game features a horde of aliens creatures which attempts to destroy a spaceship controlled by the player. The player can shoot bullets at the aliens in an attempt to exterminate them. A novel feature of the game is that periodically, aliens swoop down from their formation and make kamikaze attacks at the player's ship. Galaxian was a huge success for Namco and spawned a large number of sequels, including Galaga (1981), Gaplus (1984), and Galaga '88 (1987).

Figure 1: Original logo of the Galaxian game[1]

For our project, we aimed to implement a version of Galaxian on the FPGA board. While keeping the spirit of the game in mind, we did not intend to copy the design of the game exactly. In addition, we wanted to combine this classic game with an innovative controller scheme where the ship can be controlled by the player's hand motion. This allows for a more interactive and exciting gaming experience.

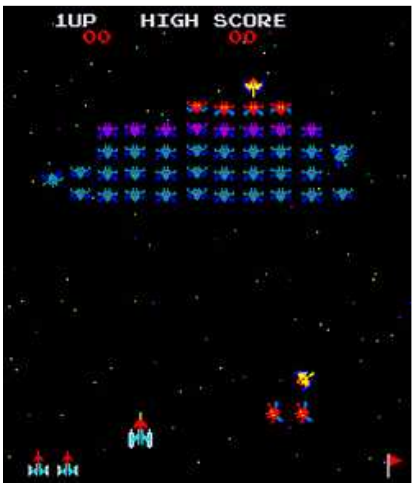

Figure 2: Screenshot of Galaxian arcade game[2]

To reach the goals stated above, we designed our project to include three main blocks. The video processing block handles the signal from the camera, which detects the player's hand motion and converts it into commands such as "move left", "move right" and "fire bullet". The game logic is responsible for creating the game itself, including the graphics, the simple physics in the game, and the progression of the game itself. Finally, the video output block takes video signal generated from the game logic and outputs it to the screen.

The following sections in this document detail the design of the system, the process of implementation and debugging, and the end result of this project. Possible improvements to the design process and the system are also discussed at the end of the document.

2. System Overview

The system is divided up into three main blocks (see figure 3). The video processor block contains code which interfaces with the video camera to get the relevant pixels. It also uses various algorithms to detect the desired object, and track its velocity and position. The video processor is designed and implemented by Danny Vo. The game logic contains all the images, rules, and interactions which make the game operate. Jeff Yuan is responsible for the design and implementation of the game logic. Finally, the

[1] Image © Namco, 1979
[2] Image courtesy of Wikipedia (http://en.wikipedia.org/wiki/Galaxian)

video output module takes the image information from the game logic and efficiently outputs to the display. This block is implemented by Danny Vo.

Between the video processor and the game logic is a selector which allows the game to be controlled by either the video processor input or labkit buttons. This allows for multiple ways of controlling the game. It also was helpful in testing and debugging the connections between the different modules.

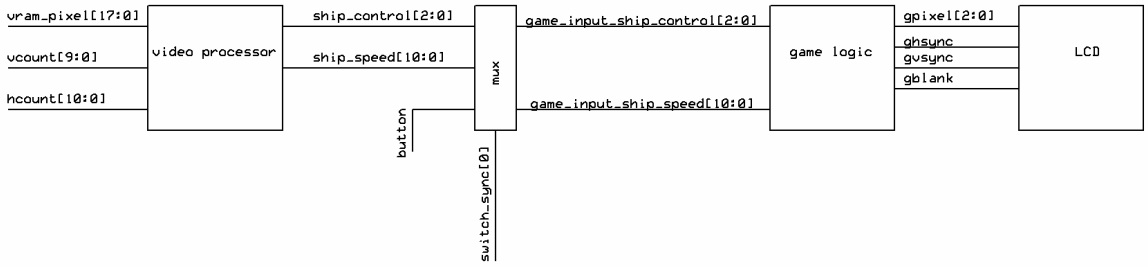

Figure 3: Overview of system logic blocks

3. Video Processor

The main purpose of the video processor is to provide the control input for the game logic block. The video processor interprets the user input in the following setting:

> The user holds an orange ball in his or her hand with the ball in the view of the video camera. As the hand is moved from left to right or vice versa, the video processor calculates the velocity and the direction of the ball. From this information, the video processor provides appropriate control inputs for the game logic block.

In order to process the ball's movement, the video processor must be provided with streams of pixels from the video camera. The code for most of the camera interfacing modules is based on those written by the 6.111 staff. In particular, the *adv7185init*[3] module is used as the driver for the video decoder. The *ntsc_decode*[4] module takes in a stream of LLC data from the *adv7185init* and generates the corresponding pixels in YCrCb format. These pixels are then written into the ZBT ram by the *ntsc_to_zbt* module and read out by the *vram_display* module. The video processor then takes in these pixel values from the *vram_display* and processes them.

Some minor modifications have been made to the modules mentioned above. In particular, the *ntsc_to_zbt* module is changed to extract an 18-bit value from the 30-bit YCrCb value provided by the *ntsc_decode*. Then the *ntsc_to_zbt* writes two 18-bit YCrCb values which correspond to two adjacent pixels into each row of the ZBT ram. In

[3] The adv7185init module is written by Nathan Ickes
[4] The ntsc_decode module is written by Javier Castro

addition, the *vram_display* module was changed to take into account of the fact that two adjacent pixels are stored in the ZBT ram.

The video processor block consists of three modules: *object_detector*, *object_pos_history* and the *game_input*. The *object_detector* module takes in the pixel value from the *vram_display* and provides the current pixel position of the orange ball to the *object_pos_history*. The *object_pos_history* module keeps a record of the pixel position of the orange ball. From these records, *object_pos_history* calculates the velocity and direction of the ball. It then computes the appropriate control inputs and passes that to the *game_input* module. The *game_input* module makes the final decision before passing the control inputs to the game logic block. The interconnection between these three modules can be seen in figure 4.

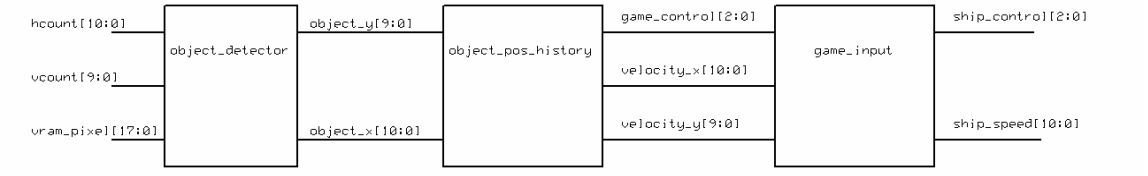

Figure 4: Block diagram for video processor

3.1 Object Detector

The main idea in detecting an object is to look for pixels which belong to the object and calculate the center of mass from these pixels. This detection algorithm examines the incoming pixel values and sums up the vertical and horizontal pixel positions that satisfy a threshold test. At the end of the frame, *object_detector* triggers a divider to take the average of these vertical and horizontal sums to calculate the center of mass. In the case of detecting an orange ball, the *object_detector* module looks for pixels that characterize the ball's color based on its threshold test. This threshold test is a comparison to see if a particular pixel falls within a range of allowed color values. If this is the case, this pixel is recognized as being part of the ball. This range of values depends on which color space the threshold test is operating on.

Originally, the *object_detector* module was implemented to detect green LEDs in the RGB color space. The detection algorithm looked for pixels which have the green value greater than a threshold value of 251. However, this particular scheme performed poorly due to two main reasons: high sensitivity to noise and low robustness to changes in the environment. In order for the detection to work, the camera must face downward to avoid the ceiling lights or any surface reflection because they also have green value greater than the threshold value. Furthermore, the camera must be blurred to reduce the amount of noise produced not only by the surround environment but also by the LEDs themselves. The sensitivity to noise is mostly resolved by implementing a median filter (discussed below). The low robustness to changes in the environment has its roots in the RGB color space, which does not separate brightness and color. In other words, as one increases or decreases the green value, the brightness is simultaneously increased or

decreased. Thus to the detection algorithm, a green LED is no different than a ceiling light. This problem is overcome by doing detection in the YCrCb color space.

To reduce the sensitivity to noise, a median filter is used. Specifically, as the *object_detector* receives a pixel which passes its threshold test, it requires that the four previous pixels also satisfy the test. If this condition is met, the *object_detector* adds the vertical and horizontal pixel positions of the previous second and third pixels (hence the median). This allows the *object_detector* to avoid noises which generally have less than four consecutive pixel values that can pass the threshold test. Figure 5 illustrates this point. In this example, the median filter recognizes that on row 15, five consecutive pixels pass the threshold test. It then adds the vertical and horizontal pixel position of pixel 1 and pixel 2. Row 17 contains two consecutive pixels which pass the threshold test. These are rejected as noise by the median filter.

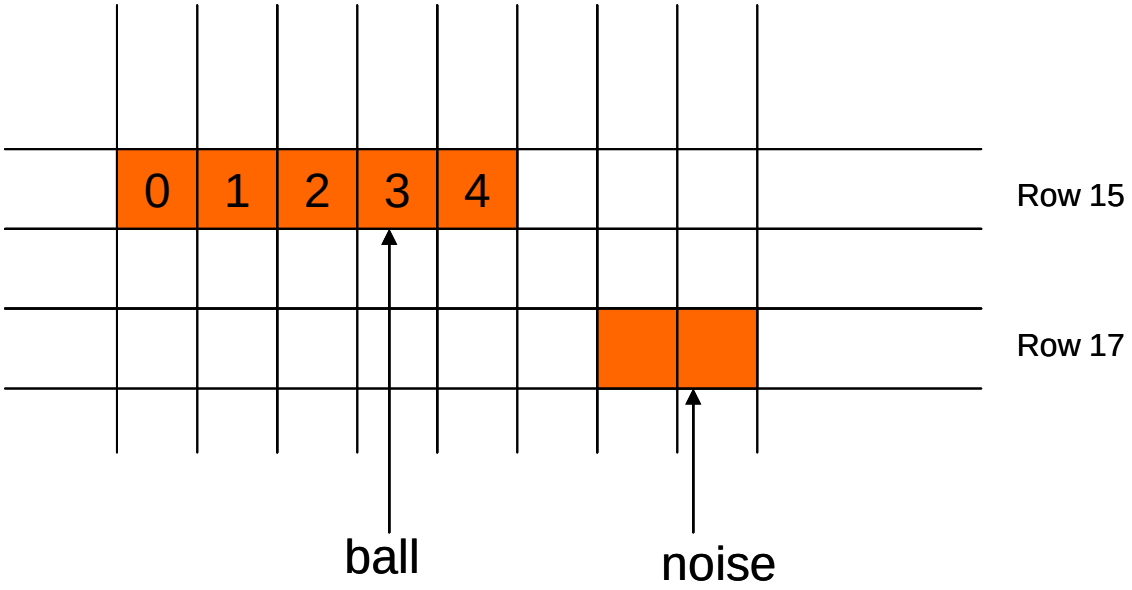

Figure 5: A median filter is used to reduce noise in object detection. Pixels in row 15 pass through the median filter since there are more than four consecutive pixels which pass the threshold test. On the other hand, the two pixels in row 17 are rejected by the median filter.

The low robustness issue is solved by switching to YCrCb color space and a better threshold test. YCrCb color space separates brightness and color. Specifically, Y value contains all information about the brightness of a particular pixel. The new threshold test is composed of testing all three color value: Y, Cr, and Cb: the *object_detector* has three ranges of value for Y, Cr, and Cb. A pixel is recognized as belonging to the ball only when its Y, Cr, and Cb values fall within the corresponding threshold range.

3.2 Object Position History

The *object_pos_history* keeps tracks of the pixel position of the orange ball and calculates its velocity every four frames. This module does not use signed number. It outputs to the game logic block the velocity of the ball, along with a game control signal

which denotes which action the user is currently performing: left, right or shoot. The shoot action is triggered when an upward velocity above a certain threshold is detected.

3.3 Game Input

The *game_input* processes the output of the *object_pos_history* module. It restricts the horizontal velocity to be within a certain range. For instance, if the user gives a velocity that is too fast for moving a ship, then the *game_input* module acknowledges the user's command but it reduces the velocity before input that into the game logic block. So this module effectively imposes an upper bound on the various output signals from the video processor to the game logic.

3.4 Testing and Debugging

Since the system is driven at 65 MHz, the timing constraint is particularly tight such that a long signal path could cause the propagation delay to exceed 15ns, the period of the clock. This would cause the pixel data from the camera to not be written correctly to the ZBT. Such a routing problem was discovered during the integration phase of the project. It was made obvious by the fact that the video camera image was correctly displayed on some compilations and not others, while the code has changed very little.

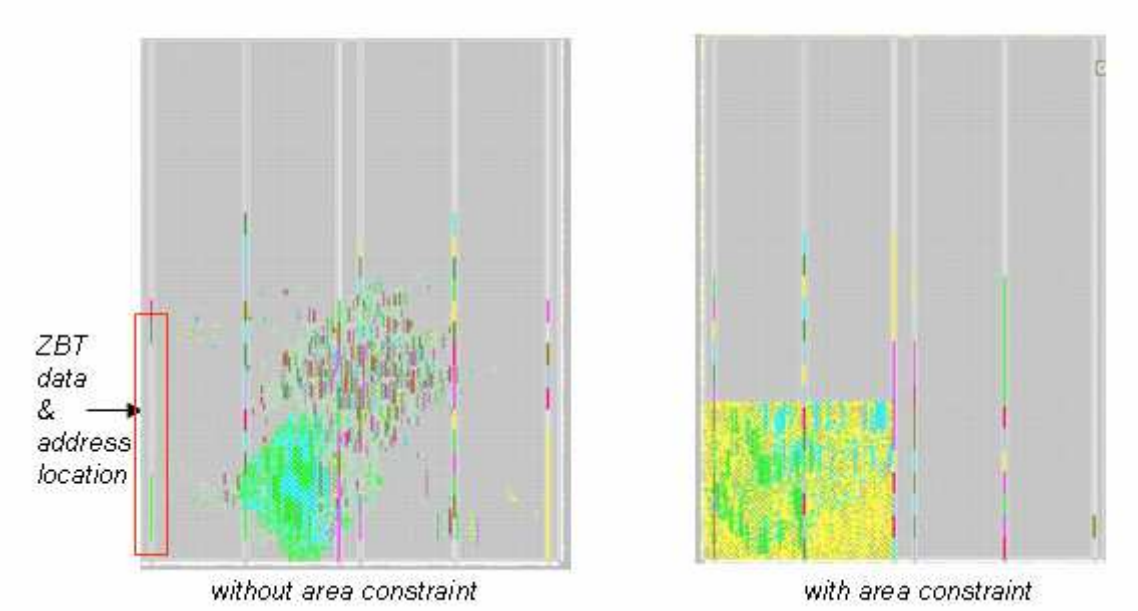

Figure 6: Comparison of project component placement on the board, with and without an area constraint.

After combing through various Xilinx manuals, a method was discovered that would allow the mapping of components on the board to be constrained within a certain area. This allowed for a more concentrated placement of components to be near the ZBT, thus eliminating the problem of having long path with high propagation delays. After applying this area constraint, the system consistently obtained good results on every compilation. To illustrate this point, figure 6 shows the difference between a layout obtained without using a constraint and one which used a constraint. As one can see,

without using an area constraint, Xilinx ISE just randomly “optimizes” the placement of components all over the FPGA chip. After applying the error constraint, the placement is limited to the bottom left hand corner of the chip close to the location of the ZBT.

4. Game Logic

The game logic block is responsible for creating the actual game play. It instantiates all the sprites seen on screen, and interprets user control into actions that are reflected on the screen. The game logic block (shown in figure 7) is composed of about ten modules which perform a variety of tasks. The main module (game_logic.v) exists as a foundation from which all other modules are instantiated and connected together. Then there are the game objects modules (ship.v, bullet.v, alien.v, and alien_formation.v) which represent the objects that appear directly the game. In addition, these modules perform calculation for the location of the objects in each frame. All game object modules are connected to the collision detector (collision.v), which reports any collisions to the game logic. A game FSM module (game_fsm.v) coordinates the sequence of actions in the game, including when certain automatic actions of aliens are triggered. Finally, there are modules (startscreen.v, endscreen.v and background.v) which are responsible for loading stored image from ROMs onto the screen.

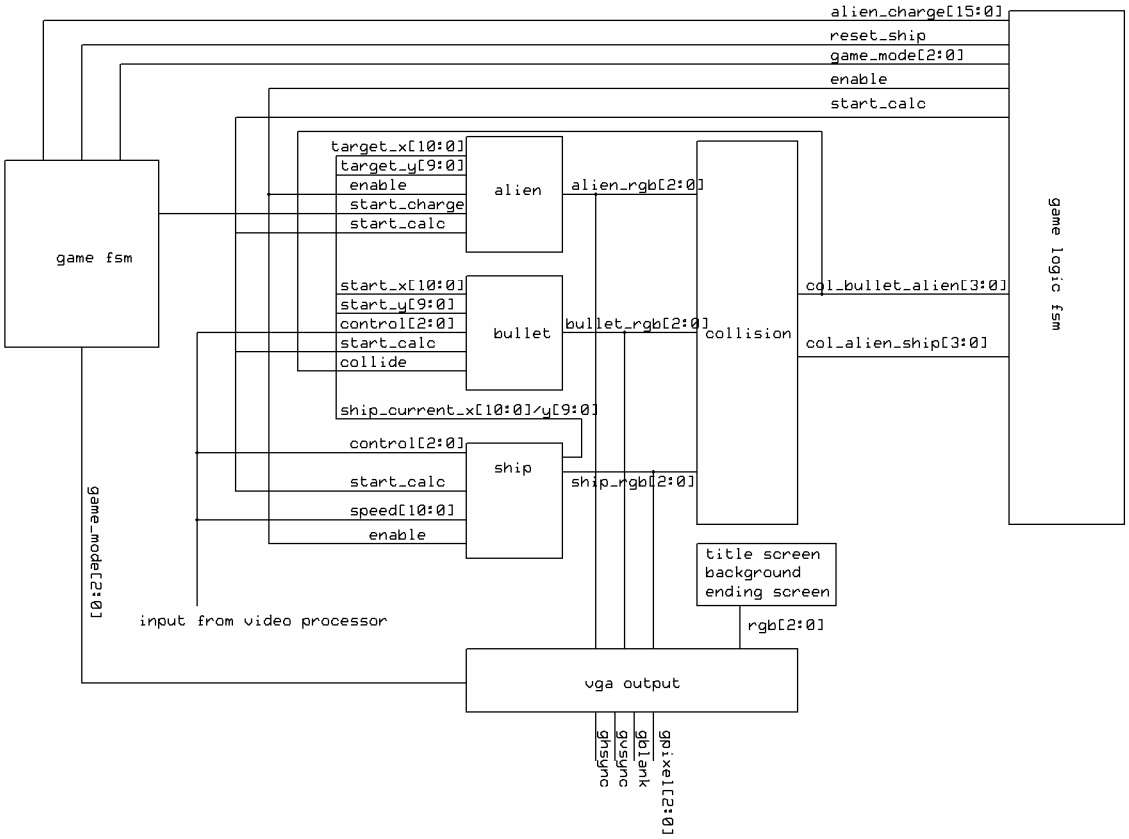

Figure 7: Block diagram for game logic

4.1 Main Game Logic

The main game logic module (game_logic.v) is responsible for instantiating all other modules of the game logic and facilitating the signal connections between them. Specifically, the ship, bullet, and alien (16 instances of it) modules are instantiated. They are connected together with the collision detector, which is also instantiated in this module. The game logic module also gathers together the sprite output (3-big RGB signal) from each game object, and combines them together into a single video output signal (a more in depth discussion of this is present in section 5).

Contained within the main module is a minor FSM which directs individual modules to perform their respective calculations on specific clock cycles. The FSM makes one pass through the following states (in order) during the vertical blanking period in each frame:

- S_NONE: latches in the value from the user input control (either labkit button or video processor).
- S_UPDATE_SHIP: sends a signal which triggers the ship module to begin calculation of its position in the next frame. Also disables the ship if it is found to be colliding with another object.
- S_UPDATE_BULLET: sends a signal which triggers the bullet module to begin calculating its position in the next frame.
- S_UPDATE_ALIEN: sends a signal to all the alien modules, which triggers their calculation of position in the next frame. Also disables any aliens which are found to be colliding with another object.
- S_DELAY: self loops in this state for 3 cycles, to give alien modules sufficient time to finish its calculations.
- S_END: reset back to the first state, and terminates all calculation for the current frame.

4.2 Collision Detection

Game objects often collide into one another in the game, and these collisions must be accurately detected. There are two types of collisions that are considered in the game, a bullet (fired by the ship) colliding with an alien, and an alien (while in kamikaze attack mode) colliding with the ship. This module performs pair-wise comparisons between bullets and all aliens, along with ship and all aliens in order to detect when both of these objects are trying to write to the same pixel. For each pair of game objects, their 3-bit RGB values are added, to produce a 4-bit sum. If the most significant bit of this sum is 1, then a collision is considered to have happened. Otherwise, no collision is detected. The reason this scheme works is because all game objects have RGB values such that when two try to occupy the same pixel, their sum would produce a “1” in the most significant bit.

An alternative scheme that was considered involved taking the RGB values of all pairs of objects and passing them through an AND gate. If both of these objects are trying to write to the same pixel, the result of the AND operation would be a “1.” The reason this scheme was not used was due to the need to optimize. In the method above, only the highest bit of each addition is inspected. But in this implementation, each bit of the result

must be implicitly checked. The different in performance is probably not too great, but the simpler and more optimized solution was chosen to minimize the chance that a timing issue would occur.

4.3 Timer

Because many of the modules in the game logic trigger actions after a certain delay, it is desirable to have a timer module which can count up to some number. The timer, when enabled, counts up from 0 to the value set as the parameter TIMEMAX. Upon reaching this value, the timer asserts the *expired* signal and resets its counter. Because the system clock runs at 65Mhz, it would be unreasonable to trigger the count on the rising edge of the system clock. Instead, the count is incremented on every new frame. This allows a much smaller register to be made in order to keep the count. This utility module is used by the alien module, game_fsm module, and others. While each of these modules can create its own custom timer, it is much more efficient to refactor this code into a single module.

4.4 Ship Object

The ship object (ship.v) is responsible for outputting the sprite of the ship in 3-bit RGB. The module reads the color information for the ship sprite from a ROM which contains the color information. Because the ship has only one frame (no animation), only a single ROM is needed.

The ship module also calculates the position of a ship in any given frame. It does so by taking in the input from the controller signal, and moving left or right accordingly. The speed of movement (in pixels per frame) is specified by the speed signal. This value is a constant if the controller being used is the labkit buttons. However, if the controller used is the video processor, the speed can actually vary according to the speed with which the player moves his hand.

4.5 Bullet Object

The bullet object is responsible for outputting a sprite for the bullet in 3-bit RGB. Because the bullet object looks like a rectangle in the original game, it is simply rendered in a similar fashion as the blob object from a previous lab. The main idea is that a pixel of a specific color is only displayed when hcount and vcount reach a specific bounded range. This allows a sprite in the form of a square or rectangle to be formed on the screen. The bullet also takes in the *control* signal, and enables itself when the “fire” command is detected from the signal. Whenever that happens, the bullet moves upward from the location it is fired, and either exits the top of the screen, or collides with an alien.

4.6 Alien Object

Among the three game objects, the alien is probably the most complex. Like the *ship* and the bullet, the alien module (alien.v) outputs a 3-bit RGB signal for each hcount and

vcount for displaying its sprite. However, alien has a three frame animation, so the module must cycle between 3 ROM images, and read data from each in turn. A timer is created which counts from 0 to 2, and the three time values are mapped to the three different frames of animation. At each time transition of timer, the output of a different ROM is set to the RGB output of the module.

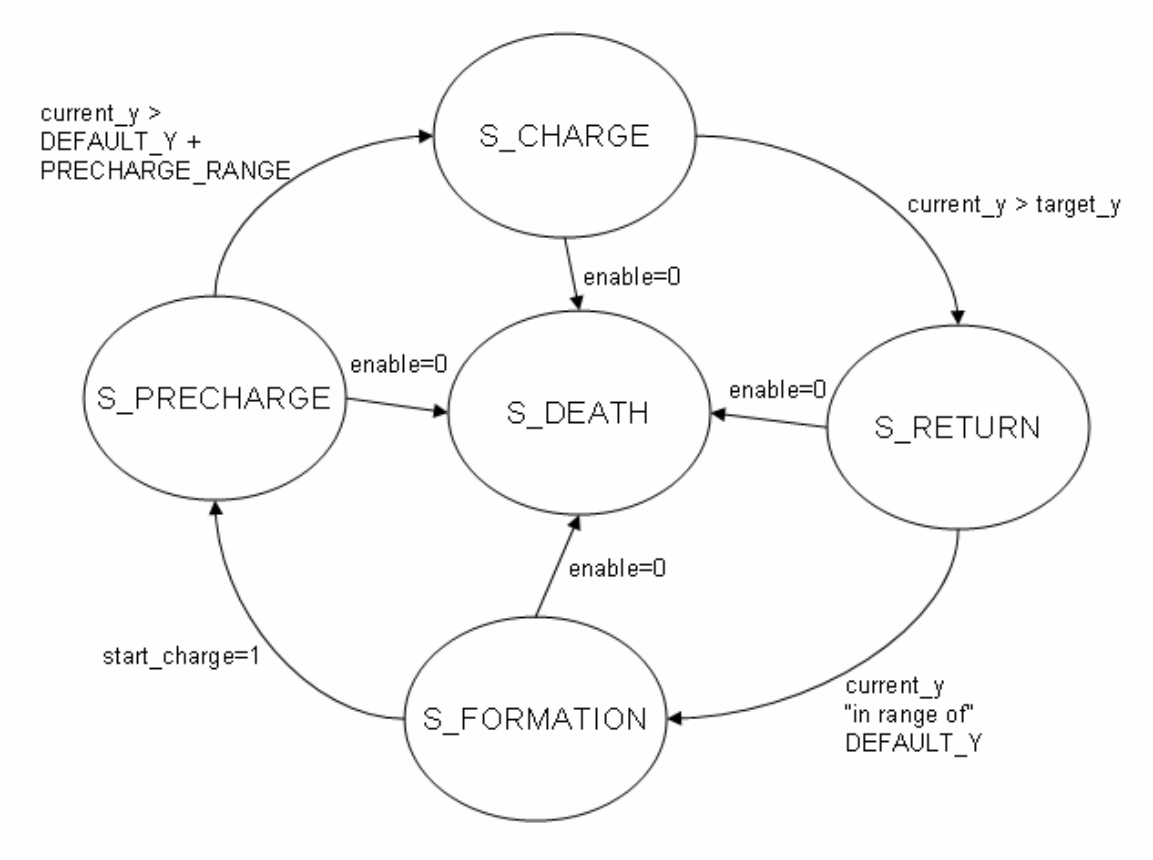

Figure 8: State transition diagram for alien module (alien.v)

The alien module must also calculate the position of the alien object in each subsequent frame. A simple FSM is used to keep track of which behavior pattern the alien is currently engaged in. As can be seen from the state transition diagram (figure 8), the alien FSM operates in a cyclical fashion, with one sink state (S_DEATH):

- S_FORMATION: This is the starting state, where the alien moves back and forth (to the left and then to the right) near the top of the screen. All the aliens move in a formation of sorts with all other aliens. The exact coordinate of the alien is calculated by a helper module (alien_formation.v). This module takes in as parameter the speed of the alien while in formation and the range of movement (how many pixels to the left and right). It calculates the position of the alien from these pieces of information. It is important to note that the calculation made by the alien_formation module is persistent, meaning that calculation is made even when the alien is not in the S_FORMATION state. A transition is made to the S_PRECHARGE state when the signal *start_charge* goes high.
- S_PRECHARGE: This is an intermediary state between S_FORMATION and S_CHARGE. The alien moves downward for some number of pixels

(parameterized in the module) before transitioning to the S_CHARGE state. The reason for this state is to prevent two aliens from running into each other.

- S_CHARGE: In this state, the alien moves downward toward the player's ship. It tracks the player's ship's position, and attempts to collide into it. The vertical speed of the alien is fixed, and the horizontal speed is limited to a maximum value. This gives the player a reasonable chance to avoid the alien. Upon reaching the y-coordinate of the ship, a transition is made to the S_RETURN state (assuming the alien does not collide with the ship).
- S_RETURN: This state indicates that the alien has gotten past the position of the ship in the y-direction, and is exiting the bottom of the screen to reappear in formation at the top. The desired coordinate for the alien in formation is calculated by the alien_formation module. When the alien gets close enough to its formation position, it transitions back to the S_FORMATION state.
- S_DEATH: This state is reached when the *enable* signal of the alien goes low. This indicates that the alien is killed. This state can be arrived at through any of the other states. However, the only way to transition out of this state is through a system reset.

4.7 Game FSM

This module contains the major Finite State Machine in the game logic block. It determines which state of the game is currently active, and also controls the exact behavior of the artificial intelligence within the game mode. This module only interacts with the game logic module, and only passes signals to that module. It is split off for modularity and clarity. The FSM in the game_logic module is a minor FSM, which controls the updating of sprites on every frame. In this module, the FSM controls the overall state of the game (refer to figure 9 for a complete state transition diagram):

- S_TITLE: This is the default state, and in this state, the title screen is displayed. When the user presses buttons 1, 2, 3, or 4 on the labkit, it transitions into the S_ALIENFORMATION state.
- S_ALIENFORMATION: This is the state that is active for the majority of time when the game is active. In this state, the aliens do not charge the ship and remain at the top of the screen. However, aliens that have already started the charge can return to formation in this state. It transitions to the S_ALIENCHARGE state after a specified amount of time (a timer module is used here to do this).
- S_ALIENCHARGE: When a transition is made to this state, one column of aliens is ordered to charge the ship. This is done by sending on the appropriate *alien_charge* signal. After this is done, it transitions back to the S_ALIENFORMATION state.
- S_SHIPDEATH: Whenever a collision occurs to the ship, a transition is made to death state. Here, a comparison is made to see if there are any more lives left for the player. If there are no more lives, a transition is made to the S_GAMEOVER state. Otherwise, it goes to the S_SHIPINVINSIBLE state.
- S_SHIPINVINSIBLE: This state is mainly a delay mechanism used to wait a few seconds before reinstating the ship on screen. The timer module is used to create

this delay effect. The intention here is to give the player some amount of time to be ready before the ship again appears on the screen. A transition is made back to the S_ALIENFORMATION state after a few seconds of delay.

- S_GAMEOVER: In this state, the end game screen is displayed.

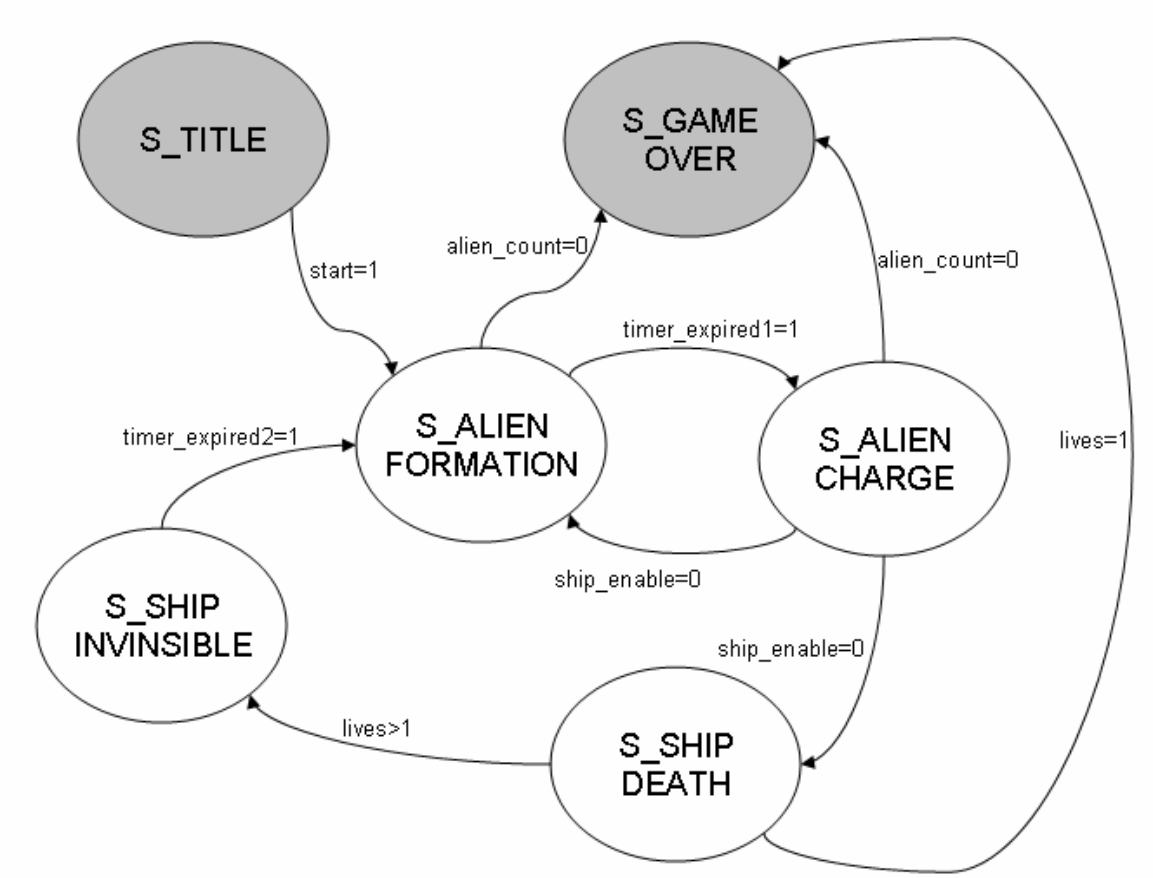

Figure 9: State transition diagram for game FSM (game_fsm.v)

4.8 Titlescreen and Endscreen:

The titlescreen and endscreen appear at the beginning and end of the game, respectively. These modules are identical, except that they read from different ROMs. The image data is read from ROMs which contain information for the 128 by 192 pixel display. Each address in the ROM is mapped to 16 pixels on the screen. Since the game resolution is 512 by 768, each hcount and vcount is right shifted by 2 to obtain map into the correct address on the ROM.

4.9 Background Image:

This module is similar to the titlescreen and endscreen in that it reads image information from a ROM with 128 by 192 pixels. However, some more work is necessary to create the scrolling effect in the background. This is done by maintaining a register with an offset value. This voffset value is incremented at a regular interval (for example every 60

frames) and added to vcount. This makes the background image appear to shift, and create the intended scrolling effect.

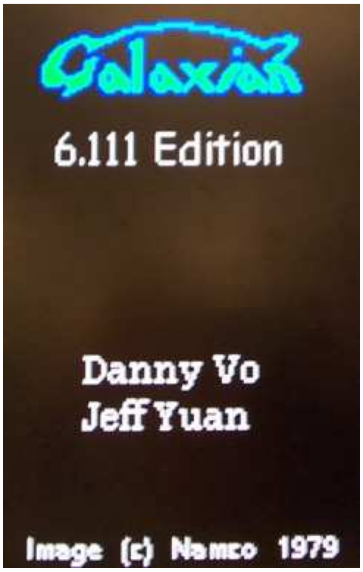

Figure 10: Title screen, as displayed on the LCD

Figure 11: End screen, as displayed on the LCD

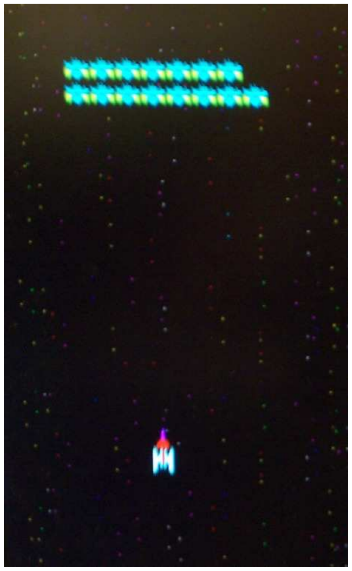
Figure 12: In game screenshot, as displayed on LCD display

4.10 Score Display

This module is creates the text for the score count and lives remaining. The only major component for this module is the binary-coded decimal module[5]. Binary-coded decimal encodes a digit by using four-bit binary number. For example, the decimal number 127 would be represent as 0001 0010 0111. The *char_string_display* module is also used along with a font ROM to retrieve the correct characters to display. These are all obtained from the 6.111 website.

4.11 Testing and Debugging:

The game logic portion of the project was debugged mainly by generating the code and outputting it onto the screen. Because most of the modules in this block have a visual component, it was quite straightforward to check the correctness of the verilog code by inspecting the pixels on the monitor. The 16 character alpha-numeric display on the labkit was also used to verify the values of certain signals.

One serious problem discovered during testing was that images displayed onto the screen appeared to have some pixels shifted. After some investigation, the cause of the glitch was discovered to be a delay in the readout of pixel information from the ROM. The calculation necessary to compute the address for reading from a ROM took one whole cycle. In addition, it took another cycle to read from the BRAM. In total, there was a two cycle delay from when an output for a particular pixel is needed, to when the data

[5] This module is taken from Binary to BCD conversions project on http://www.opencores.org.

can be read out. This caused certain pixels to be shifted from the end of one pixel line to the beginning of the next. While it was possible to implement a prefetching mechanism to calculate the address of pixels two cycles ahead, it was overly complex and proved to be unnecessary. The "quick and dirty" solution was to simply modify the image stored in the ROM so that at the left and right end matches the color of the background. This way, when a reading offset occurred, it was not noticeable.

Another problem encountered in the design process was in the collision detection module. At first, it appears that collisions that happened on the screen were never detected. However, after some investigation using a logic analyzer, it was apparent that the collision detector did not "detect" on the right clock cycles. The collision detector was only active during the vertical blanking period while collisions happened at values of hcount and vcount before the blanking period. After this discovery, the solution to the problem proved to be relatively easy.

5. Video Output

The video output block is responsible for outputting a single 3-bit RGB signal which is sent to the LCD display. All the code for this block actually resides in the game_logic module. This is due to the fact that the video output block must takes in RGB signals from many different modules in the game_logic. Thus, its operation is directly coupled with the output of many game logic modules. The discussion of its operation is separated into a separate section for clarity.

One of the major problems with this is the strict timing constraint. The system is driven at 65 MHz which restricts all the computations to be completed within 15ns. Additionally, there are 21 different modules output different 3-bit RGB signals. Thus, if these signals are naively connected to a giant "OR" gate and the video output is taken from the resulting signal, glitches would definitely occur on the LCD display.

To eliminate these glitches and to create a smooth image, a more sophisticated technique must be used. The original plan was to use double frame buffer architecture: create two BRAM, one is used for writing and one is used for displaying to the LCD. Double frame buffer architecture allows the system to writes pixels which are displayed in the next frame to one of the BRAM while the displaying is reading from the other one. This avoids the issue of changing the data while displaying. The original plan as stated above was to create two BRAM with 3-bit width and 393216 (512*768) bit depth BRAM. However, Xilinx logic core generator did not allow this, even though in theory there are enough BRAM: the XCV2V6000 contains 144 BRAM and 2952K bits total while the above architecture requires 2359K bits total. Another option was to use the ZBT ram as a frame buffer. However, due to the difficulty in timing with the ZBT, and the fact that one ZBT is already being used for video processing, this idea was also rejected.

In the end, a pipelining solution was chosen. Pipelining increases latency in exchange for high throughput. In other words, it allows the computation, which originally has a propagation delay greater than one clock period, to be divided into smaller computation

over several clock cycles. Since glitches in the display are caused by incomplete computation within one clock cycle, pipelining would effectively eliminate these glitches.

In the video output block, a 7-stage pipeline is implemented for each of red, green and blue channel. Figure 13 illustrates the green channel pipeline. The red and blue channel pipeline architecture is analogous to the green one. Furthermore, vsync, hsync, and blank signal are also pipelined so that when a pixel comes out of the pipeline stage, they still match with their original vsync, hsync and blank signal. In addition, the pipeline stages also implement priority in pixel overlay. There are two different layers: the background and the game components layer. The game components layer takes precedence over the backgrounds. This function is implemented by the multiplexers as can be seen in figure 9 below.

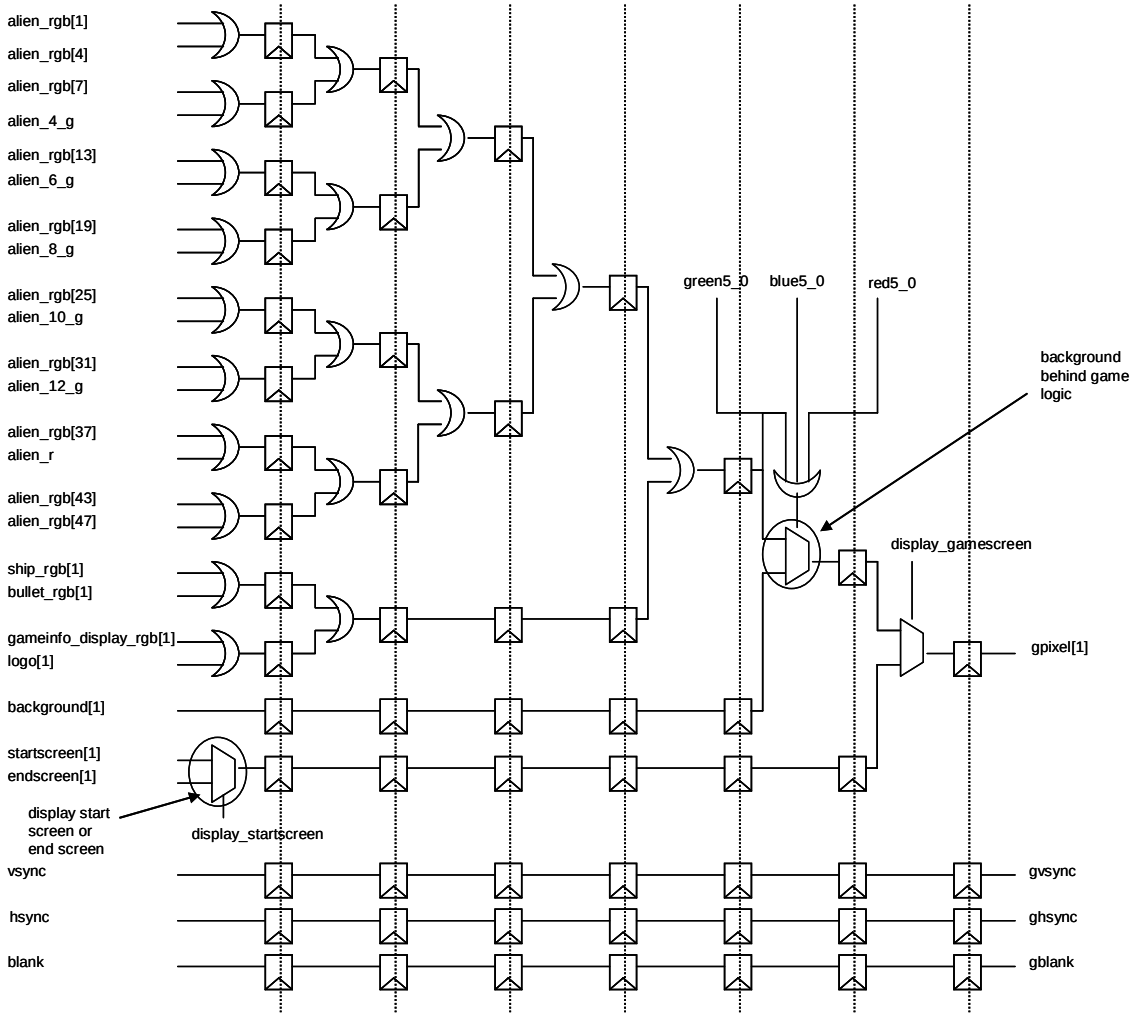

Figure 13: Diagram for the 7-stage pipeline used in video output block

5.1 Testing and Debugging

Due to the presence of the pipeline in the game logic, it runs on different vsync, hsync and blank then the video processor. Thus a selection of which vsync, hsync and blank to output to the LCD is required to correctly display the game logic and the video processor.

Originally, the system vsync, hsync and blank took one the value of the ZBT hsync, vsync and blank: the ZBT hsync, vsync and blank were delayed by three clock cycles to synchronize with the ZBT. However, due to the requirement for selecting between the ZBT and the game logic hsync, vsync, and blank, the ZBT hsync, vsync and blank were delayed by two clock cycles instead. At the third clock cycle, depending whether the system were displaying the game logic or the ZBT, the system hsync, vsync and blank would take on the corresponding hsync, vsync and blank value.

6. Conclusion

Our final project attempted to integrate the classic video game Galaxian with motion detection based controller scheme to create a new gaming experience. This system involved the design and implementation of two main components and the integration of these parts to form a functional system.

Given an opportunity to repeat this project experience, significantly more time would be devoted to planning out the modules before starting implementation. This would have reduced the amount of time used to scrap existing implementation and create something new from scratch. Time management could also have been improved to space out the work more evenly through the five week period. Although the objectives for this project were successfully implemented, the final week felt quite rushed.

While noting the possible process improvements above, the project as a whole was a tremendous success. This project allowed us to experience the whole design cycle of idea generation, preliminary planning, coding, debugging, and integration. The valuable lessons we learned here will undoubtedly prove invaluable to us in our future careers.

7. Acknowledgements

We would like to acknowledge the following people for their ideas and contributions to this project:

- Professor Terman – Numerous inspirations on game logic implementation, help in debugging video processing code
- Javier Castro – Ideas on implementing pipelining, help in debugging
- Cassie Huang – Inspiration on project idea
- Gim Hom – Answering numerous technical questions, providing ample supply of coffee
- Kevin Miu – Constructive criticism of game art, ideas on collision detection