Grab the Ninjas!

Final Project Report

Giovanni Reveles and Chuan Zhang

December 13, 2006

Grab the Ninjas! is an interactive video game in which players attempt to grab ninjas and drop them, thereby killing them and also avoid ninja stars thrown by the ninjas. A camera will detect player's finger motions and transfer these onto the display. Ninjas will randomly appear on screen and move around. If a player makes a grabbing motion and the fingers are close to the ninja, the ninja will be grabbed and will be moved around at will. Whenever ninja stars hit the player fingers, the player will lose health, shown by a health bar. The game ends when the player loses all health. A two player version will also be implemented.

Table of Contents

I. Overview 3

II. Video Modules (Chuan) 4

A. Video Overall Design 4

B. Color Video 6

C. Color Filtering Algorithm 6

D. Finger Detection Algorithm 8

E. Contact and Squeeze Detection Algorithm 9

F. Video Overlay 9

G. Video Overlay: Timing Issues 10

III Game Logic Modules 11

A. Game Logic Module Overview 11

B. Enables 13

C. Ninja Control 13

1. vsync 13

2. Moving 14

3. Grabbing/getting grabbed 14

4. Outputs 15

5. Jumping/Falling with gravity 16

C. Ninja Display 19

E. Star Display 20

F. Health Display 22

G. Ninja AI 23

IV. Testing and Debugging 24

A. Video Modules (Chuan) 24

B. Game Logic Modules (Giovanni) 24

IV. Conclusion 27

A. Video Modules Conclusion 27

B. Game Logic Modules Conclusion 29

Table of Figures

Figure 1 Video Module Block Diagram 5
Figure 2 Camera with Colored Fingers 7
Figure 3 Finger Detection Algorithm 8
Figure 4 Video Overlay 10
Figure 5 Game Logic Module Block Diagram 12
Figure 6 GRAB Internal FSM 15
Figure 7 JUMP Logic internal FSM 17
Figure 8 ModelSim Wave Forms for Jump 17
Figure 9 ModelSim Wave Form for Jump, (jump_DONE asserted High) 18
Figure 10 jump_DONE close up 18
Figure 11 Editing the Ninja Frames 19
Figure 12 Star Internal FSM 21
Figure 13 Star Display Wave Form 22
Figure 14 Star Wave Form 2 22
Figure 15 Ninja AI Waveform 24
Figure 16 Game Screen w/out grabbing 27
Figure 17 Game Screen w/ grabbing 28

I. Overview

Our game, Grab the Ninjas is a simple concept designed to test skills we have learned from previous labs in 6.111. The game consists of a person trying to grab ninjas and drop them, thereby killing them. A camera shines on the fingers of the player and translates a finger image onto the screen. Game figures such as the ninja are overlaid on top of this image for interaction. The ninjas have artificial intelligence and can shoot at the fingers. The game will be implemented as both a two player game, where the other player uses the buttons on the labkit to control a ninja, and a one player game, where all ninjas are controlled by AI.

The project was divided into two main parts. Chuan Zhang was responsible for all video portions of the project. This involved retrieving color video from an NTSC Video camera, storing the video to a ZBT SRAM, retrieving the video from a ZBT SRAM, filtering the video and running a detection algorithm to detect fingers, and finally overlaying pixels from the Game Logic portion onto the camera image.

Giovanni Reveles was responsible for all game logic portions of the project which involved storing and displaying animated ninjas on the screen using ROMs. Having the ninjas animate based on different commands asserted as well as throwing of a projectile star on command. The game module also has states that control the ninja differently whether he has been grabbed or is jumping etc. As well as programming the AI module and have a simple autonomous ninja as well as a ninja controlled by a second player and a health display module controlling the health.

II. Video Modules (Chuan)

A. Video Overall Design

The overall design of the Video portion was a stream of data. The block diagram is shown below. Digital data is stored in the ZBT and then retrieved from it. Filtering translates the YCrCb pixel to an RGB pixel. Other graphics from the game module are layered on top of these pixels until a final pixel value is given to the SVGA. Finger Detection modules run in parallel with the color filtering to detect the finger location.

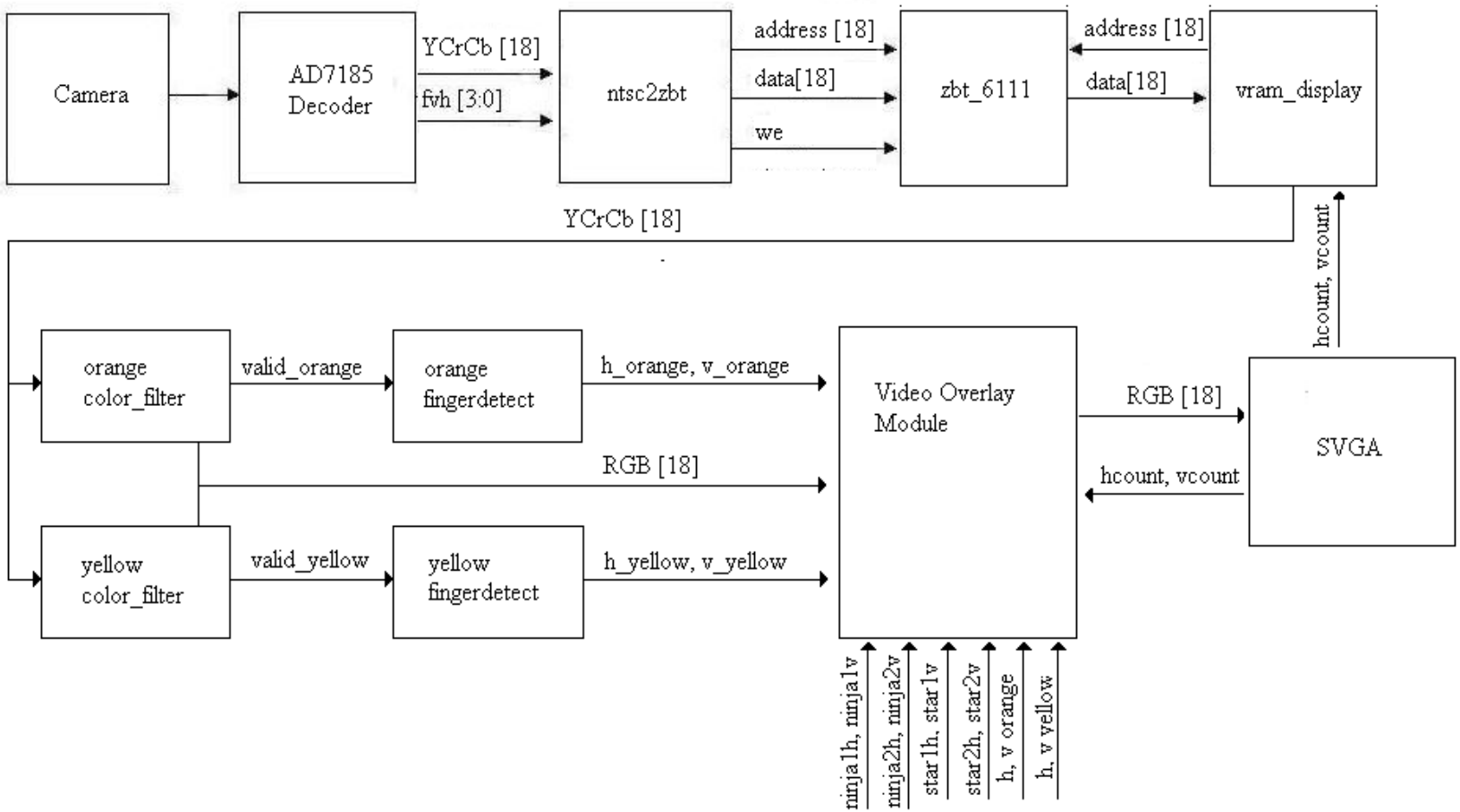

Figure 1 Video Module Block Diagram

B. Color Video

An NTSC video camera was used to capture the fingers. NTSC stands for National Television System Committee and refers to a particular analog encoding of the data. An ADV7185 Decoder chip converts the analog data into 30 bit digital data. 10 bits are allotted for Y, or luminance, Cr, or red chrominance, and Cb, blue chrominance. The most significant 6 bits of Y,Cr, and Cb are saved and the least significant two bits are thrown away. The encoding scheme was borrowed from Rebecca Arvanites and Cristina Domnisoru.

The display used is an SVGA with a 800 x 600 resolution. Ideally, pixel data coming in from the decoder could be fed to the SVGA directly. Unfortunately, this is not possible because of timing issues. The SVGA runs on a 40 Mhz clock whereas the camera and decoder run on a 27 Mhz clock. Camera inputs could not keep up with SVGA requests for information.

We utilized the labkit's built in ZBT SRAM's to act as a buffer. Data from the camera would be written to the RAM. Data from the ZBT would be read out to the SVGA. Because the camera takes longer to write one full frame to the ZBT than the SVGA takes to read one full frame, the SVGA would simply read one frame more than once. Each ZBT on the labkit contains 512,000 words 36 bits wide. With our 18 bit encoding, we can fit 2 pixels per word. Because 800*600 is only 480,000, we can fit two frames at any one time in the ZBT RAM. This encoding works because the SVGA can read one frame while the video data is being written to another frame. After a frame is written, the two frames can switch reading and writing.

Previous code interfacing the video to the ZBT and the SVGA to the ZBT was created by Isaac Chung. Three main modules facilitate the use of the ZBT as a buffer. The ntsc2zbt module takes in as input the video data from the decoder. It also takes in three signals: f (even or odd row), v (vsync) and h (hsync). The ntsc2zbt module calculates an address for the video data based on which frame is used, and f, v, and h. Logically, pixel data would come into the ZBT from the Decoder in order left to right and then top to bottom. However, the camera instead outputs every even row first and then every odd row. The ntsc2zbt waits until data for two pixels have come in before changing the word in accordance with our encoding scheme. A write enable signal is also outputted from the ntsc2zbt module. The ntsc2zbt module also synchronizes the data coming in, latching it twice to reduce the probability of metastability.

From the ntsc2zbt module, the address, write enable, and the data are hooked to the zbt_6111 module. The zbt_6111 module is a layer of abstraction which interfaces with the physical lines of the ZBT. The zbt_6111 module takes in read and write data, read and write address, and write enable. It also outputs data read.

The vram_display module is another pre-written module which interfaces the SVGA with the zbt_6111 module. The vram_display module takes in hcount and vcount and calculates an address to send to zbt_6111. The module also takes in the word read from the ZBT and outputs it to the SVGA over two clock cycles because each word contains two pixels.

The previous code was developed for black and white video. Therefore, each word of the ZBT stored 4 8 bit luminance values. To modify the code for color, small changes were made to the addressing of ntsc2zbt and vram_display.

C. Color Filtering Algorithm

To distinguish the fingers from the surrounding environment, two colors were chosen for each finger to detect. A piece of an orange glove was cut off and worn on the index finger. A piece of a yellow glove was cut off and worn on the thumb. These colors were chosen

because they represent extremes in Chrominance space, thus distinguishing them from the surrounding environment. Yellow has a very low Cb value and Orange has a very high Cr value.

Figure 2 Camera with Colored Fingers

The color filtering algorithm processes pixels as hcount and vcount from the SVGA retrieves them. For orange, the orange color_filter module would determine whether the 6 bit Cr value is below a certain threshold. If valid, the module would output a blue pixel into the video stream. If not valid, the module would output a white pixel. Similarly, for yellow, the yellow color_filter module determines whether the 6 bit Cb value is above a certain threshold. If valid, the module would output a yellow pixel into the video stream. If not valid, the module would output white. Each module also asserts a valid_pixel pulse corresponding to the color. For example, the orange color_filter module asserts the valid_orange pulse and the yellow color_filter module asserts the valid_yellow pulse. These pulses allow the fingerdetect module to detect concentrations of yellow or orange representing fingers.

Because the output of the color_filter module feeds into the fingerdetect algorithm, clean data is needed to ensure successful finger detection. A variety of physical methods were used to reduce noise where noise is defined as both false positives, detecting orange or yellow that is not from the fingers, or a false negative, not detecting a finger region as orange or yellow. First, the focus on the camera was turned as low as possible. This creates a blurring effect, ignoring small amounts of noise. Secondly, bright light was shown onto the fingers, thus increasing their luminance with respect to other data. Ideally, all luminance data is contained in the 6 bit Y data. However, if luminance is low, then Cr and Cb data will also be affected. An example is that no Cr and Cb data can be obtained if the room is totally dark. Thus, by illuminating the fingers, the probability of a false negative is reduced. Finally, thresholds are chosen which minimize false positives and false negatives. Because this threshold can change dramatically with lighting or time of day, the labkit buttons were used to program the thresholds without recompiling.

D. Finger Detection Algorithm

Each color_filter module is connected to an instance of the fingerdetect module. After the color_filter filters out all pixels except for those of the desired color, the fingerdetect module detects fingers in real time. The fingerdetect module receives valid_orange or valid_yellow pulses from the respective color_filter module. A high valid_pixel signal means the pixel corresponding to the current hcount and vcount passes the color threshold. Instead of doing center of mass calculations, a model real-time finger detection algorithm was developed. A detailed diagram followed by a description is shown below.

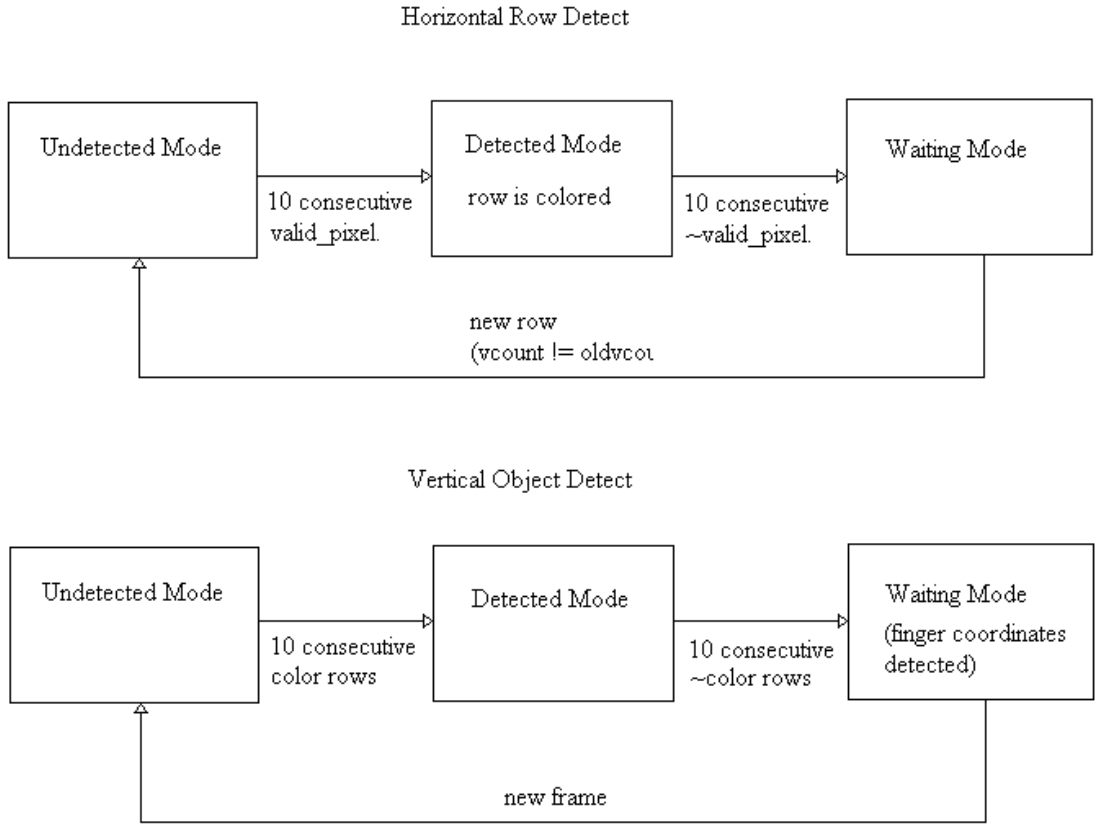

Figure 3 Finger Detection Algorithm

Pixels arrive from the ZBT from left to right and then top to bottom. Given the sequential arrival of pixels, we choose to use two parallel finite state machines. The first finite state machine is responsible for detecting whether a particular row has a high concentration of valid pixels. This corresponds to a finger begin on that row. To implement this, at every hcount and vcount, the finite state machine examines the valid_pixel signal. If it recognizes a string of 10 consecutive valid_pixels, the finite state machine believes it is on a row where the finger is and stores the horizontal position temporarily and a f1detect signal is raised high. At the end of every row, the finite state machine is set to the original searching mode again. If the finite state machine fails to find 10 consecutive valid_pixels, then the f1detect signal stays low and the horizontal position is not overwritten.

A parallel finite state machine attempts to detect 10 consecutive valid rows. 10 consecutive valid rows would correspond to a region of high density both in width and in height, which would most likely be a finger. The implementation is slightly different than the first finite

state machine. Everytime the vcount changes, the second finite state machine examines the f1detect signal. If it sees 10 consecutive signals, then it believes it has reached a high density area and enters a second state. In this second state, the finite state machine attempts to detect 10 consecutive low f1detect signals. This corresponds to the end of the high density region heightwise. If 10 consecutive low f1detect signals are found, the vertical location is stored as the finger's vertical location. The horizontal position stored from the last valid row is stored as the finger's horizontal location.

The approach used has advantages in that it does not use a complicated divider. Also, it naturally detects the bottom tips of the fingers regardless of their size. This would not be possible using a center of mass approach. The disadvantages of this approach are that the detection algorithm is sensitive to noise. Noise can both increase the possibility of false positives, where valid signals are detected at locations other than a finger, or false negatives, when no string of 10 consecutive valid signals can be found near an actual finger. The previous section addresses methods used to reduce noise.

E. Contact and Squeeze Detection Algorithm

After the finger_detect module detects the location of the fingers, the contactdetect and squeezedetect modules determine whether a squeeze is being asserted. The contact module takes in the horizontal locations of the orange finger and the yellow finger. If the two locations are within a certain threshold, the contact signal is asserted. Else, the contact signal is not asserted. The squeeze signal detects a change in the contact signal. A rising edge of the contact signal signals the squeeze signal to go high for ¼ of a second. The ¼ of a second was created by making a timer which was counted down from 5 million each clock cycle. The squeeze and contact signals are fed to the ninja modules to determine grabbing behavior.

F. Video Overlay

Our game contained many sources of images which needed to be integrated into a final image. The system we chose for video overlay was a simple parallel overlay scheme depicted in the diagram below.

Firstly, the color_filter module processes the camera data. If the pixel passes the threshold test, the module outputs a certain color, (orange pixels – blue; yellow pixels – yellow). The two signals from the two color_filter modules are concatenated to make a combined image. Because the fingers can never intersect in space and distinctly different colors are used for each color_filter module, the outputted pixel will either be blue, yellow, or white.

Additional modules are layered on top of the color_filter module in series. The addhand module draws blocks around the location of the fingertips based on their detected location. This module takes in hcount, vcount, the hand location, and the pixel from color_filter. It checks to see if hcount and vcount are within a certain range of the hand location. If true, it will replace the incoming pixel with a blue pixel. If false, it will allow the incoming pixel to pass through.

Next, an adddot module adds dots representing the ninja stars. It takes in the location of the ninja stars and the current hcount and vcount. If the hcount and vcount are within a certain range, the pixel outputted is a specific color, blue. Otherwise, the incoming pixel is allowed to pass through.

A parallel stream processes data from the ninja modules. The addninja module receives an 8 bit pixel input from the ninja_display module and hcount and vcount. If this data is not

black or 0, an 18 bit version is outputted. If the data is black, the incoming pixel is allowed to pass through. Pixel data is transferred from the ROM's which store the ninja animation. Each separate ninja is overlayed on top of each other. The health display and background are then overlayed on top of the ninja pixels. At the end, an if statement displays the second stream if the data is not white. If the data is

white, the pixel sent to the SVGA is the data from the first stream.

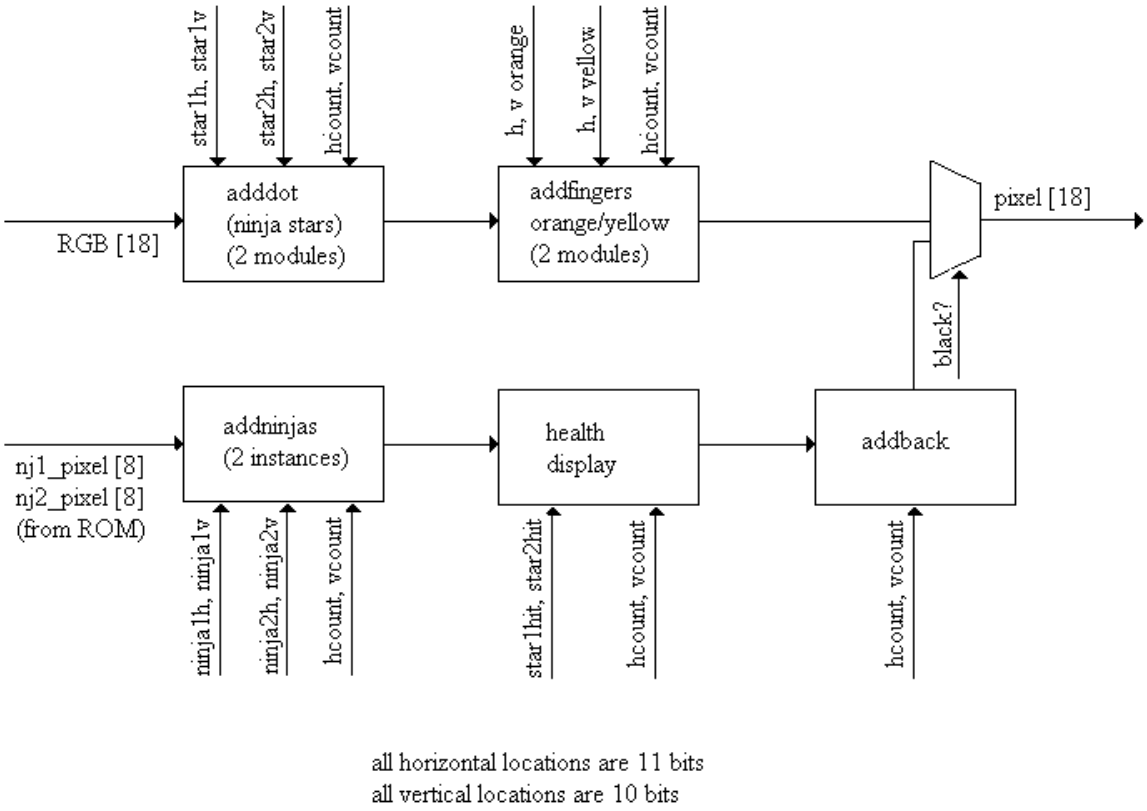

Figure 4 Video Overlay

G. Video Overlay: Timing Issues

An original video overlay scheme showed all items being overlaid on top of each other in series. However, timing issues arise because not all computation can be done within one clock cycle. The propagation delay, or the longest path of all the data was too large. Glitches appeared on the display. The traditional remedy for such a problem is to pipeline the video stream. The video stream is divided into two stages. Registers divide these two stages and store data between one stage and the next. Each individual stage now takes one smaller clock cycle.

Because the timing issues were discovered very late, a simpler method described above was used. This method traded an abundant resource, computational space for a scarce resource, time. An analogy can be taken in an assembly line. Our previous method was to have one assembly line that would be very long. It is too long to produce one unit of good per clock cycle. Pipelining allows us to do only half the work per clock cycle but on twice the number of goods. In our solution, we use two parallel assembly lines of half length each to produce the good in the desired time.

III. Game Logic Modules

A. Game Logic Module Overview

The Game Logic module is in charge of everything involving the display and control of the ninja. This involves having a module control the x and y position of the ninja on screen based on commands and states for all the commands. Actually displaying the ninja correctly and have it be animated to react to appropriate commands is in another module. Also, the ninja was to by autonomous by implementing simple AI that allowed the first player user to grab it and have its behaviors change. The ninja can throw projectiles, so it's necessary for the projectile to appear and be thrown accordingly as well as the display and decrements of the player health. It is also to determine whether the ninja is currently being grabbed and handles the game state accordingly. A block diagram in the following page shows the overall flow of the game logic module and all sub-modules that were worked on and implemented.

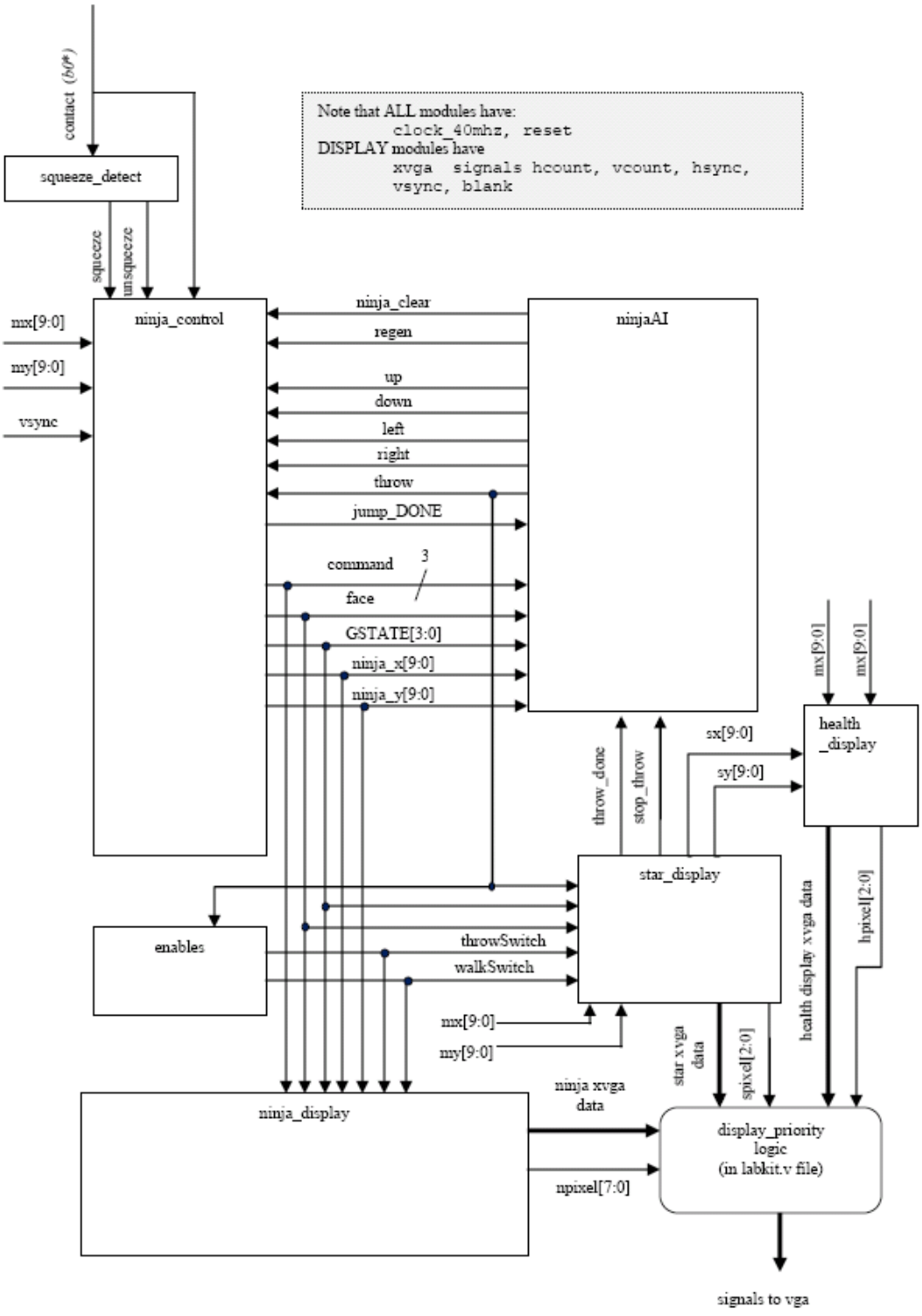

Figure 5 Game Logic Module Block Diagram

B. Enables

Enables was in charge of outputting pulses after certain amounts of time. It was in charge of the throwSwitch pulse and the walkSwitch pulse which would indicate when the animation frame of a ninja should change. These were then used by the ninja display as indication to change the current frame and output frames based on these timings which effectively displayed the animation. Since the throwing animation FSM used both throwSwitch and walkSwitch to time its animation both were fed into the star display module as well. It also output a one hertz signal which was going to be used to implement a timer for the game as a whole.

The module had three internal counters that handled the three different timings of walk throw and one hertz. Although walk and hertz started and continued indefinitely, the throw counter didn't. The module also took as input the throw command so that the throw counter would not start counting until a throw command had been detected. This made sure that the throw animation lasted just as long and looked the same at every throw command.

C. Ninja Control

The ninja control module was in charge of changing the ninja's position by outputting the 10-bit-wide new or default x and y values and abilities based on control inputs. These coordinates point to the upper-left corner of the ninja 'square'. As explained in the ninja_display module, each ninja 'frame' was 104 x 106. The controls were up, down, left, right, and throw and were inputs. Both left and right moved the ninja in those directions respectively. An up command would make the ninja jump, and if left and right were asserted also, it would jump and move in those respective directions. Another control involved whether the ninja was grabbed or not, and this was asserted internally in the module using inputs contact (input to squeezedetect module), squeeze and unsqueeze (generated by the squeezedetect module). The module also kept track of internal states making sure that if the ninja was grabbed, it couldn't throw ninja stars. The jump and grabbing internal states were sub FSMs that would not allow the ninja to do anything else until the sub-process was completed.

1. vsync

First and foremost, the ninja control module needs to know how frequently we are changing frames. The control is in charge of changing the ninja position, and it needs to be changed after a frame on the screen has been outputted. The position can't be changed at every clock cycle for obvious reasons, the human eye would be unable to see anything, and it would greatly affect the display of the ninja on the actual screen. Our design involves displaying at a 800 x 600 pixel resolution which requires a 40 MHZ clock an xvga module modified accordingly. The then appropriate xvga vsync signal (asserted LOW) indicates when the screen has been scanned through (at 60 Hz). When the vsync signal is asserted, it remains asserted for several clock cycles before being unasserted and the next frame begins to be scanned through. A simple FSM was used to keep track of whether we had just had the vsync signal asserted and if it was the same one from the cycle before or a few cycles. It then outputs a pulse just before the next frame is scanned and all of the changes made to the ninjas position are not made until this signal is detected. This was done by detecting the state at every clock cycle, and if it was not the state where a new vsync signal was detected, the x and y values that were sent out

remained the same, otherwise it would move the ninja in the appropriate direction. In this way, the ninja's position would remain the same until it was the correct time to change positions (at the start of a new frame).

2. Moving

Moving left and right are fairly self explanatory, move the ninja left or right at a constant speed, that is change the x value by the same number every frame as long as the command was asserted. Initially jumping was handled in a similar fashion, change the coordinates by the same value at every frame up and then after a certain height threshold down, again handled by an FSM where once the coordinates reached the floor (or close enough), the next y value would be the default y value and it would be able to continue. The jumps could be asserted using a small pulse since after asserting an up, the internal jump state would continue to be in this state regardless of whether up is still asserted and the control would then continue to change x and y accordingly to the jump command. This was incidentally the ninja control module version we used in the final project demonstration. The result was then a constant velocity trajectory which was slightly unrealistic. A newer version of ninja control implemented a more realistic in the air motion displaying and changing the vertical y coordinate with negative gravity acceleration. The approach to this will be talked about shortly.

3. Grabbing/getting grabbed

The way that grabbing worked was through the squeezedetect module. In testing, 'contact' was assigned to b0, and the hand x and y coordinates were the coordinates of the mouse cursor. In the project, 'contact' was asserted by the two blobs on the finger tips coming closer than a certain threshold value. The squeezedetect module made sure that the squeeze assertion was made within 1/4 of a second so that the potential bug of asserting squeeze well away from the ninja and simply sweeping in such that the ninja would attach itself to the hand is avoided. After that stage, the grabbed logic makes sure that the hand x and y are within the range of the box before asserting grabbed signal. Once a grabbed signal is asserted, GSTATE (the internal grab state) goes in to a JUST_GRABBED state where as long as the grabbed signal remains high, the state will remain the same. The control then keeps track of the hand x and y were at the previous clock cycle (mx_old, my_old) using registers and computes a differential signed value that will be used to add to the current value so that the ninja x and y change by the same amount as the hand x and y changed. The result is the ninja gets 'grabbed' and is forced to be moved by the hand as long as grabbed remains true. Once grabbed goes LOW, the state changes and the ninja should fall to the ground if it is not there already. Again, movement in the vertical axis was at constant speed initially for an earlier ninja control module. When assigning x and y values, the module gave priority to the state cases based on grabbed, so a ninja will always be grabbed even if say, right is asserted while this happens. It isn't until GSTATE returns to its initial state that the ninja is allowed to move again (or do anything else for that matter). The module doesn't allow the ninja_x and ninja_y values to handle other commands if the grab state isn't one where the ninja isn't being grabbed.

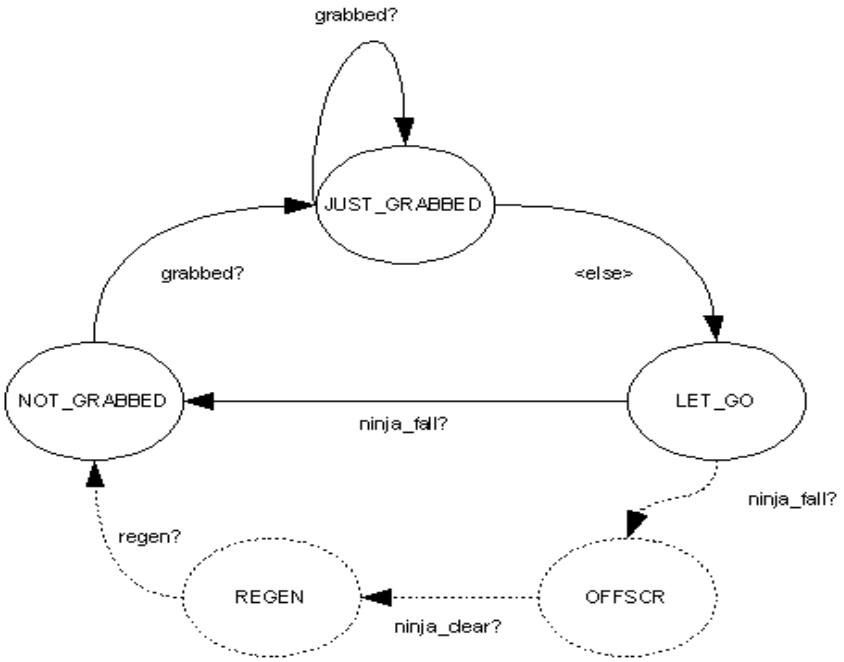

Figure 6 GRAB Internal FSM
The dotted lines indicate the states that would've been implemented with a fully fledged regenerating AI module.

4. Outputs

Outputs to the ninja display as already mentioned are the x and y values for the ninja animation, as well as command, face, and GSTATE. Command is output to the ninja display module to let it know what command is currently being executed. In the ninja control module, the command registers (there are 6 commands so command is 3-bits) are assigned in an always block, meaning they'll be valid at the next clock cycle. The internal FSMs which handle jumping and grabbing for the ninja change based on what value command seems to be taking. This keeps both the display and the control on the same page regarding which command is currently being processed. This way the ninja displays only appropriate animations based on commands that are being processed in the control. Face keeps tabs on which way the ninja is facing. When checking which command to process the module keeps track of which of left or right commands have been processed by assigning face a value at each case. Face is output to the star display module and to the ninja display module so they can handle the animations accordingly. Star display only shoots the star in the direction that the ninja is facing and likewise, the displayed frame should be facing whichever of left or right were processed last. The GSTATE which was mentioned before is the state register bus for the grab states that are processed in the module. Originally this was not meant to be sent out, but for AI and regeneration purposes, it was convenient to have those modules behave according to the current state grab had and give it the same level of priority in case the ninja happened to be in a grabbed state. Additionally, the frame that the ninja outputs when it is grabbed should remain until the final state of Grab.

5. Jumping/Falling with gravity

For simplicity, initially falling or jumping in the air was made at a constant rate. When the ninja was let go after being grabbed its y value would fall at the constant rate of 6 pixels per frame (since the bottom of the screen is the higher value, 6 was added to the current y). Jumping would have the ninja rise (by subtracting) at 6 pixels/frame and after it's y value surpassed a certain threshold, it would begin it's return to the floor (by adding) at the same 6 pixels/frame rate. To model gravity, the rate at which the ninja fell needed to change at a constant rate. This can be achieved by having a counter and adding this value in along with the value of 6 pixels at every frame. That method turns out to be incredibly high since frames refresh at 60 Hz on the screen. The way it was implemented for jump was using a start_index bit which goes high whenever in jumping mode, that is based on the internal jump state, start_index was asserted high whenever the jump state changed to indicate there was a jump and remained high until the final state of the jumping process. A counter which was called jump_index was then made to start counting when start index was HIGH, and set to zero in any other case. A seperate bit jump_v_inc checked if jump index reached a certain value (which turned out to be about a quarter of a second in clock cycles) and if it was, it would go HIGH for one clock cycle. This would then set the jump_index counter back to zero and would make a separate counter jumpCount go up by one. The end result was the jumpCount went up by one every 1/4 of a second. When handling jumps, jumpCount was then added in addition to the constant pixel value and the states were changed to just check whether the y value had returned to (or close enough) to the floor. The figures in the following page show the waveforms for the jump logic in the ninja control. They were wired to the outputs when implementing and debugging the system and demonstrate how the jumping scheme works. In the simulation, a vsync wave form was generated to help visualize the transitions. Mathematically, the infinitesimal length of time van be viewed as 25ns (1/60Hz) since that is as fast as we can change the velocity to meet the display constraints. The constant value that we change y by can then be viewed as the dy/dt, since we make this change every at every frame refresh. The jumpCount is the resultant change dy/dt will have at every frame so it can represent acceleration. When tested, the ninja demonstrated a quadratic trajectory when jumping. A similar method was used for the grabbing stage when letting go and when tested, the ninja also accelerated to the floor.

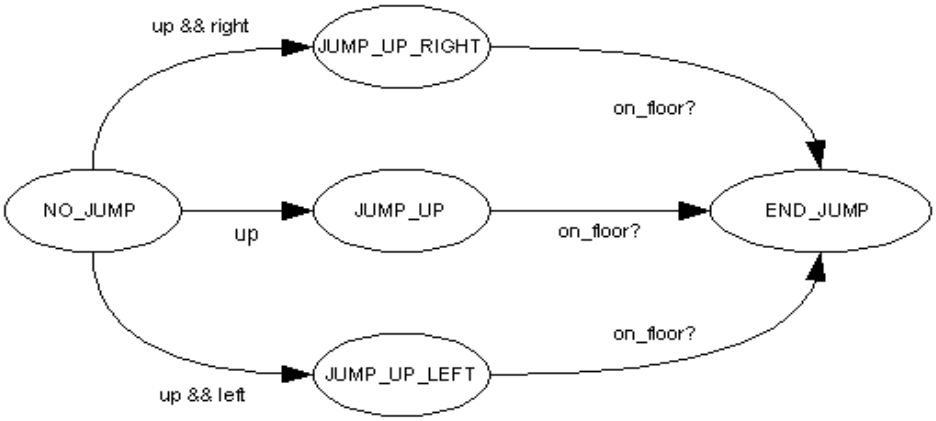

Figure 7 JUMP Logic internal FSM

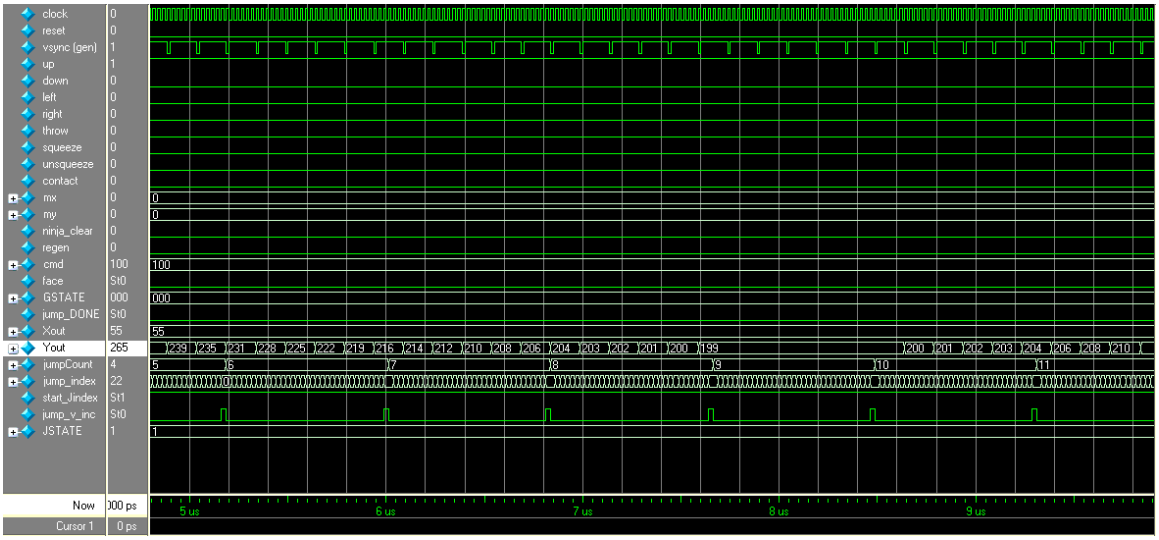

Figure 8 ModelSim Wave Forms for Jump

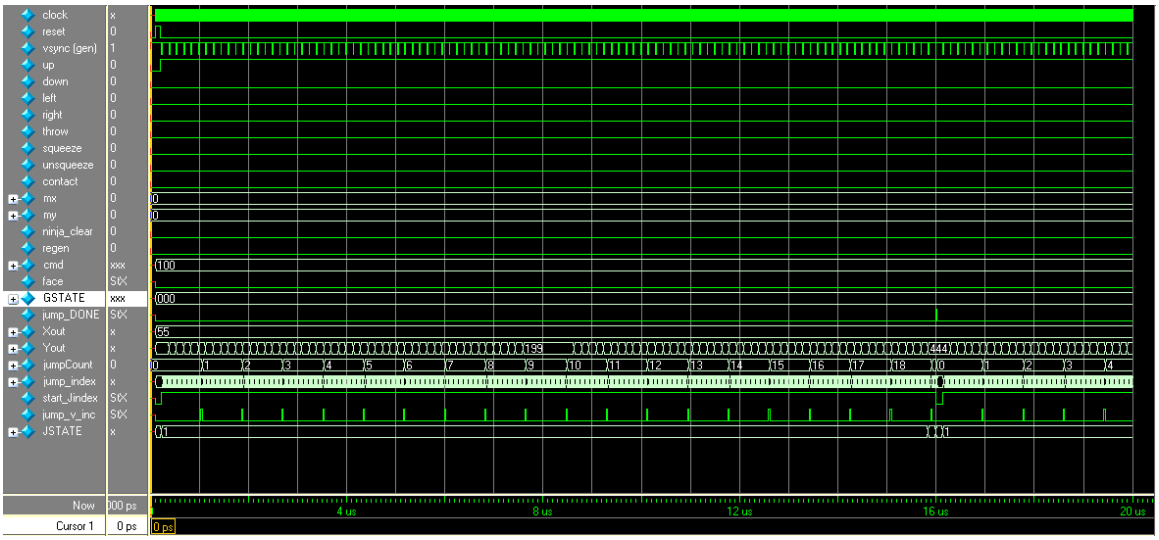

Figure 9 ModelSim Wave Form for Jump, (jump_DONE asserted High)

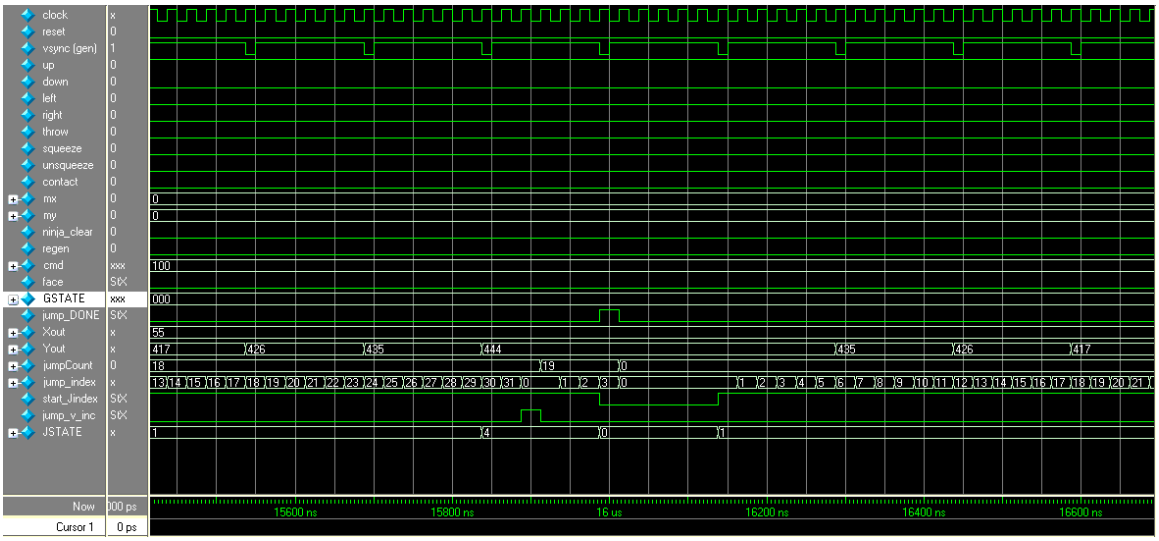

Figure 10 jump_DONE close up

C. Ninja Display

The Ninja Display was in charge of outputting the ninja pixel based on control commands and face. The position of the ninja was determined by the control and also fed into the ninja display. The throwSwitch and walkSwitch signals from the enables module were also inputs to handle when animation frames should change. The grab state from ninja control was also an input to handle the grabbing display and make sure the modules were on the same page regarding grabbing.

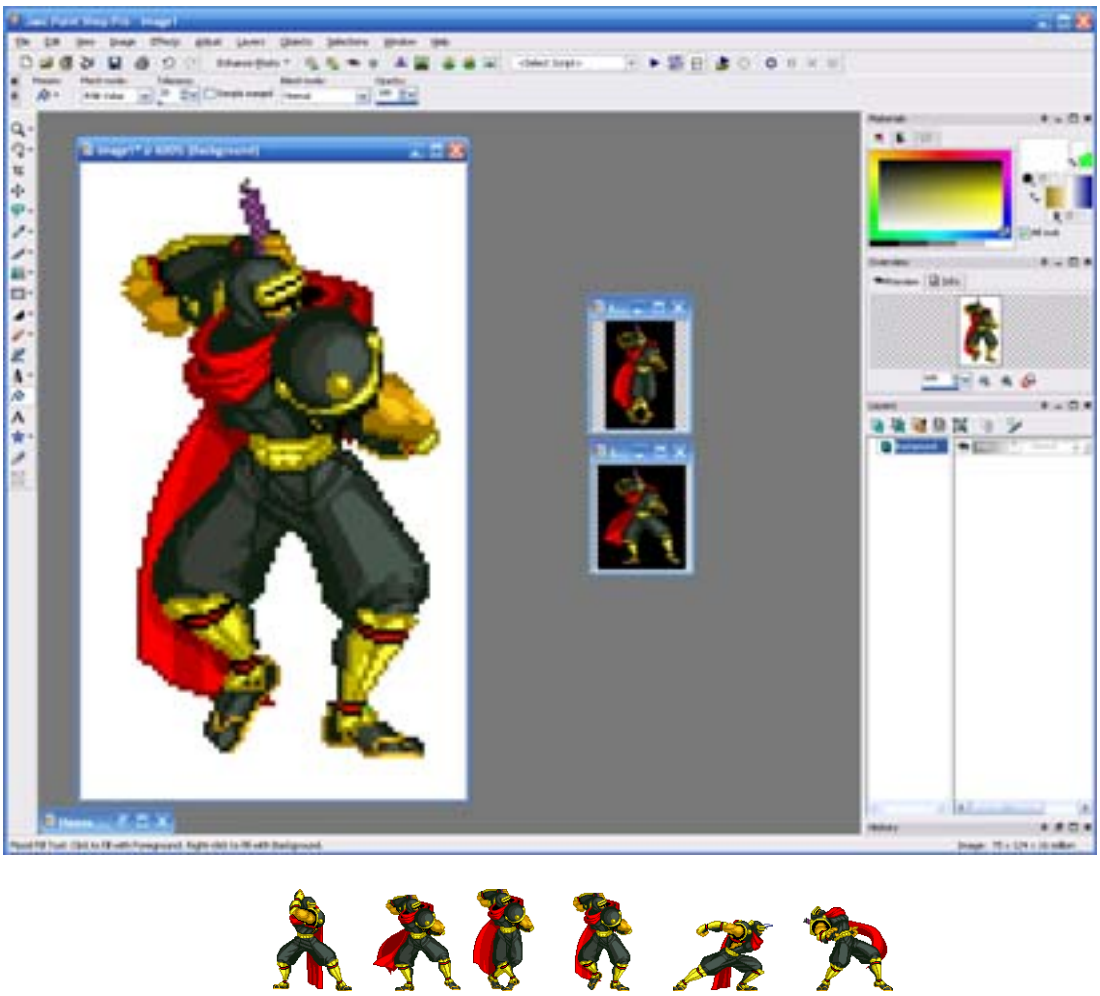

Figure 11 Editing the Ninja Frames

There were 6 ninja display frames that were mapped from PGM (portable grey map) files to COE files using the pgm2coe.py script. The COE files were used as init files for ROMs using the Xilinx Core Generator for single port Block RAMs. Initially the size of the ninjas were 155 x 142, and as we found, there wasn't enough BRAM to map more than 3 ninja frames at this size since the COE file stores an 8-bit gray value. They were resized to 104 x 106 and they fit in the BRAM in this new size. There were three frames for walk, two frames for throw and one frame for grabbed as shown above. The frame for grabbed turned out to be smaller than the other frames but not by a lot. They [the ROMs] were all instantiated in the module and were constantly being addressed so that there was always a valid pixel value for each animation state.

The logic in the display determined which was the correct pixel to display based on the command received and the grab state from the control module.

The way the ROMs were addressed depended on the face. The same ROM was used for both left and right faces of the same frame, there were just differences in the way the ROM was being addressed. Two different addresses were then computed at each clock cycle, addr, and addrREV and two different instantiations were made of each frame, one reading it with addr, and one with addrREV so that a valid pixel value for each frame was always ready. For addr (right face), the addressing worked as follows: 1. Reset to zero if hcount and vcount are both 0, 2. If hcount and vcount are at a location where a ninja pixel should be displayed, increment the address by one. 3. Then if the address exceeds the maximum address value (104 x 106 = 11024), set it back to zero. This addressing scheme worked and kept the code simple so this was the addressing scheme that was used. A formula depending on the x, y, hcount, and vcount values could be used to determine the address for right face. Using, $addr = (v_C - y)(W) + (h_C - x)$, where v_C, and h_C are vcount and hcount, x and y the ninja coordinates and W the width of the frame (104). This way, the address is only increased when it is in the valid range of the ninja and won't increment otherwise. For the left face, such a formula was used since trying to implement it another way proved to be more difficult and complex. The formula used was $addrREV = (v_C - y)W + ((W-1) - (h_C - x))$. For the reverse address, we basically needed the last pixel in the current horizontal line first and count backwards, then at the next line, increment the starting point and count backwards again. The same valid range was used even though there would be technically a valid address everywhere, so the logic which determines which pixel to output also checks for the valid ninja range before outputting a ninja pixel.

The ninja frames were designed to not have any white pixels inside of where there was a ninja and to have white pixels outside the ninja but inside the 104 x 106 block. In order to display just the ninja pixels, the display logic could simply filter out any white pixels detected when determining whether to output a ninja pixel or a pixel of another module.

As mentioned before, the walkSwitch and throwSwitch timing pulses fed from the enables modules indicated when to switch the animation frames for respective ninja commands. Internal FSMs were used to determine which animation stage we were in for both the walk and the throw commands and they switched states based on the pulses. Logic was also used with the states and the face to determine which face to output for each command, and finally, at the end using the command input, face and whether hcount and vcount were in the valid range, the correct frame pixel was outputted.

E. Star Display

The star display was in charge of outputting a small star projectile at the location of the ninjas hand and moving as if the ninja had thrown it based on the throw command. It took as inputs the same timing pulses from the enables module as well as the face and location of the ninja to time the animation and appearance of the star correctly. It also takes in as input the GSTATE from the control in order to disable the star from displaying if the ninja happens to be grabbed.

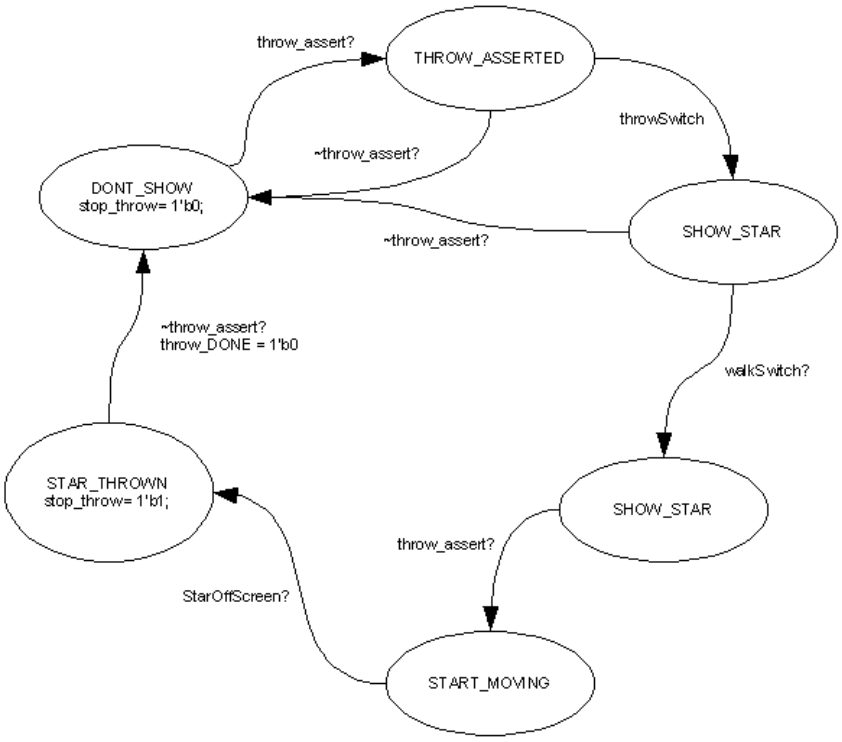

Figure 12 Star Internal FSM

The internal FSM was designed so that a second throw could not happen unless the throw command was de-asserted. This way a new throw command needed to be made every time a new star was to be displayed. The throw command could also be a small pulse such that once the star had been thrown it would remain on the screen. When the star wasn't to be displayed its location was simply put off the screen. Parameters were used to determine where to initially place the star. The coordinates were taken from the ninja picture just under where the hand displays. This location was different based on whether the ninja was facing right or left. A target bit was also taken as an input to determine whether the star would be sent out flying strictly horizontally or at an arbitrary angle. The module then would display, and change the stars x and y coordinates accordingly. The star x and y values are inputs to the health display module (as well as the hands x and y) to determine whether there has been a collision and would result in making the health go down.

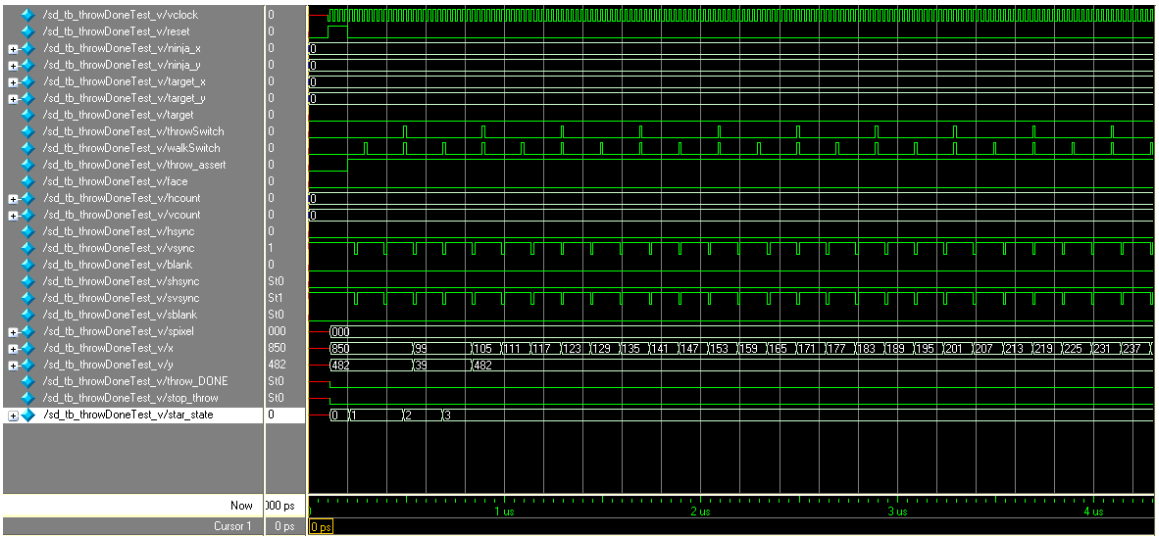

Figure 13 Star Display Wave Form

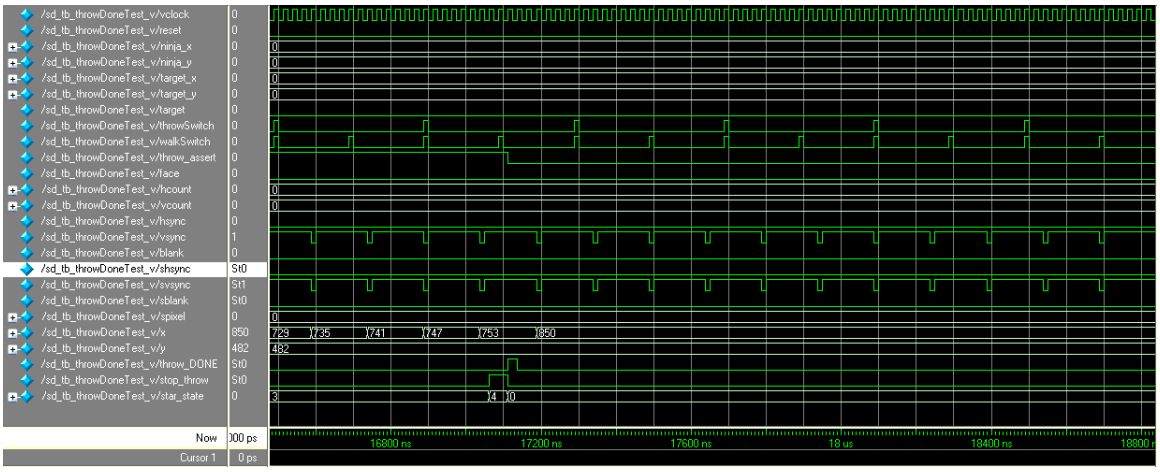

Figure 14 Star Wave Form 2

F. Health Display

The health display module is basically a blob that has an internal health value that is decreased based on whether there has been a collision between a star that has been thrown and a certain range from the hands x and y values. It is a bar of height determined by a parameter internally and outputs red/green pixels based on whether hcount is greater, less than or equal to the current health value. Its width is also determined by an internal parameter.

G. Ninja AI

Before adding AI, the ninja on screen was controlled entirely by the user using the labkit buttons. The AI module was modeled after a user asserting such commands based on timing and states of the system. Since the ninja control handled the jumping states, a jump_DONE bit was output from the ninja control to the AI module to indicate when a jump had finished and another command could be asserted. Likewise for the throw, since the star display handled the stars position, throw_DONE and stop_throw outputs from the star display were fed into the AI module to indicate when the star had gone off the screen (indicating that the AI should stop outputting a throw command) and when the throw process had finished. These new signals were used to trigger the changing of states if the respective command had been issued. That is for example once the AI was in the state where a jump is asserted, it then waits for the jump_DONE signal before moving on to the next state.

Regardless of any behavior the AI module is output, priority needs to be given to whether or not the ninja is being grabbed. The GSTATE from the ninja control is also set as an input to the AI module so that the module can determine if the ninja is being grabbed, and output no commands. When assigning different states for distinct behaviors, the AI module first checks whether the ninja is being grabbed, and assign appropriate states for being grabbed for the different behaviors. The ninja's were to be removed from the screen and set to be regenerated after a certain time. The counts were triggered based on whether that state had been reached after being grabbed.

The first AI that was implemented was the one that I called "GOOMBA". Like in the Mario game, the goomba walks back and forth until it bumps into something, then turns around. Likewise the ninja behavior walks in one direction until it gets out of range, then walks in the other direction. This was very simple behavior and was implemented first to show that the AI module works and effectively it did. The GOOMBA states also handled whether the ninja was being grabbed and output no commands when those states were reached. Each command up, down, left, right, and throw were assigned separately and output to the ninja control. This behavior was displayed in the demo.

A similar strategy was used when trying to implement more complicated AI, although the results were still buggy and unstable. Debugging issues and possible reasons for more complex AI not working in their entirety will be talked about in the debugging section. As long as grabbing states were given the appropriate priority, all commands sent out to ninja control were tested and confirmed to work. There were problems between state transitions to different commands when synthesizing, but some during simulation (which assumed all transition signals would be valid and asserted at correct timing since they were stimulated in the test bench) displayed correct state transitions based on the signal assumptions. So more complicated AI states were attempted and although some of the transitions between states were successful, there were many bugs in bounding the ninja on the screen, asserting the next command, etc.

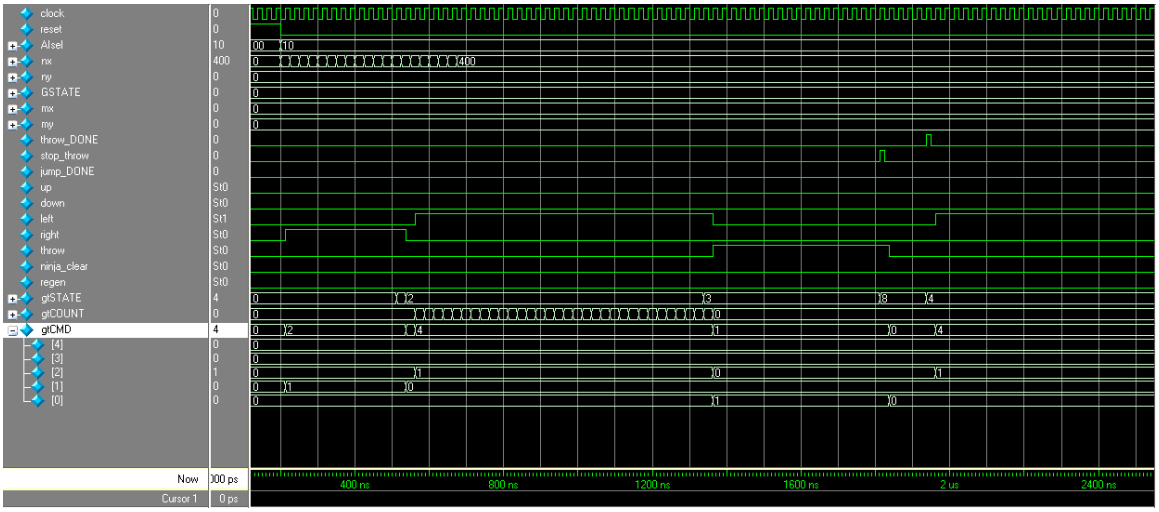

Figure 15 Ninja AI Waveform

IV. Testing and Debugging

A. Video Modules (Chuan)

By far most of the time on this project was spent on testing and debugging. The general approach was to begin with Isaac Chung's pre-built code for black and white video and to add effects incrementally. This method worked in general and allowed me to succeed in adding small pieces efficiently and relatively error free. I achieved color video and filtering capability relatively early.

The main problems I ran into had to do with my finger detection algorithm not working. The finite state machines I used originally were highly complex and small errors would break it. Instead of making use of simulation, I would recompile the code and see if images appeared on the screen. This cost me lots of time. In the end, I achieved success by drawing out a detailed state transition diagram. I then recoded the file over and over until it worked.

Another piece which required lots of testing and debugging was the interaction between the fingerdetect algorithm and the physical factors. The algorithm developed above is very sensitive to noise. Thus, a lot of time was spent tweaking the thresholds of the detection algorithm. An experimental optimum was reached after trying many different glove colors and surface types, trying many different lighting conditions and camera focus settings. The difficulty also originated from the interconnectivity of the physical factors and the software factors. The performances of both were tied to each other and an individual optimum did not necessarily lead to the overall optimum. At first, these thresholds were hardwired into the file and each testing and debugging took many minutes.

B. Game Logic Modules (Giovanni)

We had decided to use an 800 x 600 display in order to maximize the camera display and not have the ninjas display on screen too small. We needed a 40 MHz clock and a modified

xvga module that asserted the hcount, vcount, hsync, vsync, and blank values at the correct bound parameters.

Since the hand grabbing mechanism involving the camera might not be ready in time, I used the ps2_mouse file available on the fall 2005 website to use the mouse through the ps2 port, display a cursor for it and use that as the inputs for the hands. Although I changed the timing parameters to match the 40 MHz clock that we were using for our project, on the ps2 interface modules, the mouse still only worked about half of the time. I spent a day trying to figure out why, and then later decided that it was not important to spend time trying to make the mouse work. When it would work, I would use it to provide with movable hand x and y coordinates and simulate grabbing with the click of a button. In this way I was able to test the grabbing states and ninja display in action.

The First thing that was implemented was the ninja display module. The ninja pictures were edited, transferred to COE and to ROMs. So the first bugs were with the addressing scheme. It was easiest (although time consuming) to test by synthesis and actually displaying the ninja frames. It was very easy to see if there were addressing errors since the ninja would not look ok. The problems that were run into were problems with timing, or wrong computation of address meaning that the arithmetic for computing the address was wrong. The same address scheme was used for all ROMs so this only needed to be debugged and fixed once, then we knew that the others would behave in the same way with the exception of the grabbed frame which was a different size but it only differed in address bounds on height and width and not on how the address was computed.

Next to test and implemented was animation. The enables module was easy to write since we did a similar one in lab3 with the one hertz enable. Again, simulation cannot read from a ROM so the best way to test whether the animations worked were to look at them. It proved to be somewhat time consuming trying to fix state transitions in regards to faces and addressing and having to re-synthesize every time to make sure that everything worked, but the debugging process was fairly straight-forward. After synthesis and viewing of the behavior of the animations, I could identify where the problems were and look in likely places that caused those problems. A simple control module controlled behaviors for left, right and standing still first to make sure that the ninja changed face states and kept them, as well as made animated motions only when left or right requests were asserted. The animation for throw was also implemented in this same manner and got it working when synthesized.

The star display module came next. Again, face and position of the ninja were critical to making the star appear in the correct location. I had the star change states according to how the display changed animations using the same throwSwitch and walkSwitch signals. The star display could be simulated in modelsim and that's how it was debugged as shown in some of the diagrams. I hooked the internal states to the outputs and observed the waveforms. There were a few transition issues (as well as assumptions that throwSwitch and walkSwitch were asserted correctly, but since they are the same signals that change the animations, if the animations work, so will the star state transitions) but they were identifiable and after enough time was invested, the bugs were fixed and phased out. A case I initially ignored was the grabbing case. It wasn't handled correctly and we discovered that after being grabbed, the star would still generate a star being thrown on the screen, the decision to include the GSTATE as an input to the star display module was then made and we could keep track on when the ninja had been grabbed and limit the appearance of the star accordingly.

The ninja control module involved extensive debugging, and reevaluation of case priority. The first case was making sure that no changes occurred unless we had the new vsync pulse that was talked about in the ninja control module. Anything faster than that would be too

fast and would not output the desired results correctly. Certain cases were not easy to see where the problem was other than it wasn't applying the desired changes. In one case, on the transition between the end of a grab and the default grab state, the logic that assigned x, y, and face would only check if the grab state was back to the original before checking for movement signals so the ninja would not move at all since initially it's at the original grabbing state, so everything remained still. Jumping was also difficult to implement. One major bug that was evident was when after asserting a jump, in order to indicate that the ninja has reached the floor, the logic checks if it surpass the default y boundary on the next transition. Instead of simply assigning the next y as default, it proceeded to the next state, but assigned the value of y that surpassed the default y and as a result the ninja was displayed only from his ankles and above.

After a certain period of time when synthesizing was getting very long, I decided to start a new project and use blobs that were the size of the ninja instead. By this time I had confirmed that all the animations worked to satisfactory levels so moving to blobs to try to decrease synthesis time seemed like a good way to go.

The ninja AI I would say was the hardest thing to implement. Although the very early simple goomba AI worked fairly soon after trying to implement it, trying to implement AI that was slightly more complex proved to be very difficult. The state transitions depend on many other inputs that although the AI was testable using tesbenches in Modelsim, doesn't work exactly the same when synthesized. The reason is when using testbenches, all inputs are generated by the user and assumed to be correct. If there are slight variations on the inputs by the other modules during synthesis, it could cause the AI FSM to just hang there waiting for the signal. A huge problem I ran into was keeping the ninja on the screen. Using a similar mechanism of bounce back with the goomba, I tried to have it output different commands, but gave priority to the x going out of bounds and having the ninja move to another state so that it would move in the opposite direction and found that there were cases where it would still go out of bounds. Another strategy I was trying to implement was a "chicken" strategy where the ninja moved away from the hand x position at all times. Again, the bounding problem persisted and ended up bounding the ninja outside of the screen instead of inside. I think if given another week or so of debug time, some pretty cool AI could've been implemented.

Aside from modelsim on the ninjaAI, star_display and ninja_control modules and hooking up internal states and wires to outputs to see what was going on, the labkits led displays were also useful. In particular the hex display (using the display hex v file found online in the fall 2005 site) proved useful to hook up the AI state transitions to know what the ninja was "thinking" and where it was transitioning. I used the LEDs to assert which command the AI was outputting and this helped me figure out what was going on and where the ninja was getting stuck, or try to determine when and where it went to certain states.

IV. Conclusion

A. Video Modules Conclusion

In conclusion, I had achieved all of the goals I stated on my checkoff list. I had reliable finger detection. I could detect a squeeze and unsqueeze easily, and I could grab objects and pick them up.

Figure 16 Game Screen w/out grabbing

Figure 17 Game Screen w/ grabbing

Overall the experience was difficult and rewarding. I am glad that we chose to do something which involved video effects. This made the end result much more palpable. If there was more time, I would have experimented with ways to make the finger detection even more robust. The finger detection fades out around the edges because of uneven lighting.

If I had could do this project over, I would have picked my direction more carefully. Many important design choices were made halfway through the project, in effect doubling the amount of work performed. An example was deciding whether to filter fingers by RGB data converted from YCrCb or filtering YCrCb data directly. Secondly, the initial plan was to use two gloves which were of the same color. This required a complicated algorithm to distinguish between two fingers of the same color. Also, when the fingers were close together, no detection is possible. Danny Vo, a fellow 6.111 student gave me the idea of using different colors for different fingers. Following this scheme from the beginning would have saved me a lot of time because the fingerdetect module was the most difficult to make.

I would have started integration much earlier than Saturday before the due date. Many problems cannot be discovered until the system as a whole is put together. An example is the video timing issues which we luckily corrected.

Most importantly, I would have done more simulation in lieu of actual testing in this project. Compile time increased from 1 minute in the beginning to over 4 minutes at the end. Particularly at the beginning, I wasted a lot of time making small changes and then recompiling. Towards the end, I began using switches and buttons to find optimal values for necessary parameters without recompiling.

B. Game Logic Modules Conclusion

I feel that all of the basic functionalities were implemented and shown to have worked well. Transitioning from blob to ninja wasn't buggy or difficult and the animations the ninja displayed remained the same. Moving from a mouse grab to a hand grab worked quite well also. And the interface between the two worked very well. User controls versus AI worked, although AI FSM behaviors had some problems, I think they could have worked if I had tried to implement them sooner or had more time to debug. I would have definitely worked on AI with blobs initially, even though I spent a large amount of time fixing addressing issues, I think some complex AI would have been awesome to show. There was a lot of figuring how to do things on my own instead of asking for help when I was working with the modules. Although I like figuring things out on my own, asking for help on say addressing would have may cut time spent on addressing the ninja and more time spent on AI. Lots of people display pictures in their projects so there was no need to reinvent the wheel. I later had found that color pictures could be mapped to COEs using MATLAB. Having colored ninjas would've made the project a little more color friendly.

```
//////////////////////////////////////////////////////////////////////////////
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
// Modified: Chuan Zhang
//
//////////////////////////////////////////////////////////////////////////////

// Prepare data and address values to fill ZBT memory with NTSC data

module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we);
   input clk;  // system clock
   input vclk; // video clock from camera
   input [2:0] fvh;
   input dv;
   input [17:0]        din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output ntsc_we;     // write enable for NTSC data

   parameter COL_START = 10'd20;
   parameter ROW_START = 10'd20;

   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 x 768 XGA display

   reg [9:0]    col = 0;
   reg [9:0]    row = 0;
   reg [17:0]   vdata = 0;
   reg          vwe;
   reg          old_dv;
   reg          old_frame;    // frames are even / odd interlaced
   reg          even_odd;     // decode interlaced frame to this wire

   wire         frame = fvh[2];
   wire         frame_edge = frame & ~old_frame;

   always @ (posedge vclk) //LLC1 is reference
     begin
        old_dv <= dv;
        vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
        old_frame <= frame;
        even_odd = frame_edge ? ~even_odd : even_odd;

        if (!fvh[2])
          begin
             col <= fvh[0] ? COL_START :
                    (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
             row <= fvh[1] ? ROW_START :
                    (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
             vdata <= (dv && !fvh[2]) ? din : vdata;
          end
     end

   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
   reg [17:0] data[1:0];
   reg       we[1:0];
   reg       eo[1:0];

   always @(posedge clk)
     begin
        {x[1],x[0]} <= {x[0],col};
        {y[1],y[0]} <= {y[0],row};
        {data[1],data[0]} <= {data[0],vdata};
        {we[1],we[0]} <= {we[0],vwe};
        {eo[1],eo[0]} <= {eo[0],even_odd};
     end

   // edge detection on write enable signal

   reg old_we;
   wire we_edge = we[1] & ~old_we;
   always @(posedge clk) old_we <= we[1];
```

```
   // shift each set of four bytes into a large register for the ZBT

   reg [35:0] mydata;
   always @(posedge clk)
     if (we_edge)
       mydata <= {mydata[17:0], data[1]};

   // compute address to store data in

   wire [18:0] myaddr = {y[1][8:0], 1'b0, x[1][9:1]};

   // update the output address and data only when two bytes ready

   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = (we_edge & (x[1][0]==1'b0));

   always @(posedge clk)
     if ( ntsc_we )
       begin
          ntsc_addr <= myaddr; // normal and expanded modes
          ntsc_data <= mydata;
       end

endmodule // ntsc_to_zbt

//////////////////////////////////////////////////////////////////////////////////
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//

//////////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.
//////////////////////////////////////////////////////////////////////////////////

module zbt_6111(clk, cen, we, addr, write_data, read_data,
                ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

   input clk;                 // system clock
   input cen;                 // clock enable for gating ZBT cycles
   input we;                  // write enable (active HIGH)
   input [18:0] addr;         // memory address
   input [35:0] write_data;   // data to write
   output [35:0] read_data;   // data read from memory
   output       ram_clk;      // physical line to ram clock
   output       ram_we_b;     // physical line to ram we_b
   output [18:0] ram_address; // physical line to ram address
   inout [35:0]  ram_data;    // physical line to ram data
   output       ram_cen_b;    // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire         ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.

   reg [1:0]   we_delay;

   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;
```

```
   // create two-stage pipeline for write data

   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};

   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA's
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
   assign      ram_address = addr;

   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;

endmodule // zbt_6111

//////////////////////////////////////////////////////////////////////////////////
// File:   vram_display.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
// Modified: Chuan Zhang
//
//////////////////////////////////////////////////////////////////////////////////

module vram_display(reset,clk,hcount,vcount,vr_pixel,
           vram_addr,vram_read_data);

  input reset, clk;
  input [10:0] hcount;
  input [9:0]  vcount;

  output [17:0] vr_pixel;
  output [18:0] vram_addr;
  input [35:0]  vram_read_data;

  wire [18:0]   vram_addr = {vcount[9:1],1'b0,hcount[9:1]};

  wire   hc2 = hcount[0];
  reg [17:0]    vr_pixel;
  reg [35:0]    vr_data_latched;
  reg [35:0]    last_vr_data;

  always @(posedge clk) begin
    last_vr_data <= (hc2==1'b0) ? vr_data_latched : last_vr_data;
    vr_data_latched <= (hc2==1'b1) ? vram_read_data : vr_data_latched;

end

  always @(*)          // each 36-bit word from RAM is decoded to 4 bytes
    case (hc2)
      1'b1: vr_pixel = last_vr_data[17:0];
      1'b0: vr_pixel = last_vr_data[35:18];
    endcase

endmodule // vram_display


//////////////////////////////////////////////////////////////////////
//  FingerDetect Module
//  Author: Chuan Zhang
//  This module takes in valid_pixel, hcount, vcount and returns
//  the h and v of the finger
//////////////////////////////////////////////////////////////////////

module fingerdetect(reset, clk, valid_pixel, hcount, vcount, h, v);
        input reset, clk, valid_pixel;
        input [10:0] hcount;
        input [9:0] vcount;

        output [10:0] h;
```

```
output [9:0] v;
reg [10:0] h, htemp;
reg [9:0] v, vtemp;

// detection logic
reg [1:0] detectmode;
reg [11:0] hcounter;

reg [9:0] oldvcount;
reg rowdetect;
reg [1:0] vdetectmode;
reg [5:0] vcounter;



always @(posedge clk) begin

        oldvcount <= vcount;

        if (reset) begin
                h <= 0;
                v <= 0;
                detectmode <= 2'b00;
                hcounter <= 10;
                oldvcount <= 0;

        end
        if (~reset) begin
                if (vcount == oldvcount) begin
                        if (detectmode == 2'b00) begin
                                if (valid_pixel) begin
                                        if (hcounter == 0) begin
                                                detectmode <= 2'b01;
                                                hcounter <= 10;
                                                htemp <= hcount;
                                                rowdetect<= 1;

                                        end
                                        if (hcounter > 0) begin
                                                hcounter <= hcounter - 1;
                                        end
                                end
                                if (~valid_pixel) begin
                                        hcounter <= 10;
                                end

                        end

                        if (detectmode == 2'b01) begin
                                if (valid_pixel) begin
                                        hcounter <= 10;
                                end
                                if (~valid_pixel) begin
                                        if (hcounter == 0) begin
                                                detectmode <= 2'b10;
                                                hcounter <= 10;
                                        end
                                        if (hcounter > 0) begin
                                                hcounter <= hcounter - 1;
                                        end

                                end

                        end
                        if (detectmode == 2'b10) begin

                        end

                end


                if (vcount != oldvcount) begin
                        hcounter <= 10;
                        rowdetect<= 0;
                        detectmode <= 2'b00;
```

```
                                if (vcount !=580) begin

                                        if (vdetectmode == 2'b00) begin
                                                if (rowdetect) begin
                                                        if (vcounter == 0) begin
                                                                vdetectmode <= 2'b01;
                                                                vcounter <= 10;
                                                        end
                                                        if (vcounter != 0) begin
                                                                vcounter <= vcounter - 1;
                                                        end
                                                end
                                                if (~rowdetect) begin
                                                        vcounter <= 10;
                                                end

                                        end
                                        if (vdetectmode == 2'b01) begin
                                                if (rowdetect) begin
                                                        vcounter <= 10;
                                                end
                                                if (~rowdetect) begin
                                                        if (vcounter == 0) begin
                                                                vdetectmode <= 2'b10;
                                                                h <= htemp;
                                                                v <= vcount;


                                                        end
                                                        if (vcounter != 0) begin
                                                                vcounter <= vcounter - 1;
                                                        end
                                                end
                                        end
                                        if (vdetectmode == 2'b10) begin

                                        end
                                        end
                                        if (vcount == 580) begin
                                                vdetectmode <= 2'b00;
                                                vcounter <= 10;

                                        end



                        end
                end

        end


endmodule


////////////////////////////////////////////////////////////////////////////////
// Yellow Color_Filter Module
// Author: Chuan Zhang
// Translates YCrCb pixels into filtered RGB pixels
////////////////////////////////////////////////////////////////////////////////

module yellow(reset, clk, pixel_in, pixel_out, contrast_input, selector, valid_yellow);
        input reset, clk;
        input [17:0] pixel_in;

        // determines the threshold value
        input [17:0] contrast_input;
        reg [6:0] yhighthreshold;
        reg [6:0] ylowthreshold;
        reg [6:0] cbthreshold;

        // selector to switch mode
        input [1:0] selector;

        assign crselect = selector[1];
        assign cbselect = selector[0];
```

```
        output [17:0] pixel_out;
        reg [17:0] pixel_out_all;
        reg [17:0] pixel_out_y;
        reg [17:0] pixel_out_cr;
        reg [17:0] pixel_out_cb;

        output valid_yellow;
        reg valid_yellow;


        assign yhigh = (pixel_in[17:12] < 63);
        assign ylow = (pixel_in[17:12] > 2);
        assign cb = (pixel_in[5:0] < cbthreshold + 13);

        assign pixel_out = crselect? pixel_out_cr : cbselect? pixel_out_cb :
pixel_out_all;

        always @(posedge clk) begin

                        yhighthreshold <= {0,contrast_input[17:12]};
                        ylowthreshold <= {0,contrast_input[11:6]};
                        cbthreshold <= {0,contrast_input[5:0]};

                        valid_yellow <= (yhigh && ylow && cb);

                        if (yhigh&&ylow) begin
                                pixel_out_y <= {pixel_in[17:12],pixel_in[17:12],pixel_in
[17:12]};

                        end
                        if (~(yhigh && ylow)) begin

                                pixel_out_y <= 18'b0;
                        end

                        pixel_out_cr <= {pixel_in[11:6],pixel_in[11:6],pixel_in[11:6]};

                        if (cb) begin
                                pixel_out_cb <= {pixel_in[5:0],pixel_in[5:0],pixel_in[5:0]};
                        end
                        if (~cb) begin
                                pixel_out_cb <= 18'b0;
                        end
                        if (yhigh && ylow && cb) begin
                                pixel_out_all <= {1'b1, 5'b0, 1'b1,11'b0};
                        end
                        if (~(yhigh && ylow && cb)) begin
                                pixel_out_all <= 18'b0;
                        end


        end
endmodule

////////////////////////////////////////////////////////////////////////////////
// Orange Color_Filter Module
// Author: Chuan Zhang
// Translates YCrCb pixels into filtered RGB pixels
////////////////////////////////////////////////////////////////////////////////

module orange(reset, clk, pixel_in, pixel_out, contrast_input, selector, valid_orange);
        input reset, clk;
        input [17:0] pixel_in;

        // determines the threshold value
        input [17:0] contrast_input;
        reg [6:0] ythreshold;
        reg [6:0] crthreshold;
        reg [6:0] cbthreshold;

        // selector to switch mode
        input [1:0] selector;

        assign crselect = selector[1];
```

```
        assign cbselect = selector[0];


        output [17:0] pixel_out;
        reg [17:0] pixel_out_all;
        reg [17:0] pixel_out_y;
        reg [17:0] pixel_out_cr;
        reg [17:0] pixel_out_cb;

        output valid_orange;
        reg valid_orange;


        assign y = (pixel_in[17:12] < 63);
        assign cr = (pixel_in[11:6] > crthreshold);
        assign cb = (pixel_in[5:0] < 60);

        assign pixel_out = crselect? pixel_out_cr : cbselect? pixel_out_cb :
pixel_out_all;

        always @(posedge clk) begin

                        ythreshold <= {0,contrast_input[17:12]};
                        crthreshold <= {0,contrast_input[11:6]};
                        cbthreshold <= {0,contrast_input[5:0]};

                        valid_orange <= (y && cr && cb);

                        if (y) begin
                                pixel_out_y <= {pixel_in[17:12],pixel_in[17:12],pixel_in
[17:12]};

                        end
                        if (~y) begin

                                pixel_out_y <= 18'b0;
                        end
                        if (cr) begin
                                pixel_out_cr <= {pixel_in[11:6],pixel_in[11:6],pixel_in
[11:6]};

                        end
                        if (~cr) begin

                                pixel_out_cr <= 18'b0;
                        end



                        if (cb) begin
                                pixel_out_cb <= {pixel_in[5:0],pixel_in[5:0],pixel_in[5:0]};
                        end
                        if (~cb) begin
                                pixel_out_cb <= 18'b0;
                        end
                        if (y && cr && cb) begin
                                pixel_out_all <= {12'b0, 1'b1,5'b0};
                        end
                        if (~(y && cr && cb)) begin
                                pixel_out_all <= 18'b0;
                        end


        end
endmodule


//////////////////////////////////////////////////
// ContactDetect Module
// Author: Chuan Zhang
// The contact detect module takes in the two horizontal
// positions of the fingers and returns whether
// they are within a certain threshold distance
// away from each other.
//////////////////////////////////////////////////
```

```
module contactdetect (reset, clk, f1h, f2h, contact);
	input reset, clk;
	input [10:0] f1h, f2h;

	output contact;
	reg contact;

	assign range = (f2h[10:4] <= 10 + f1h[10:4]);

	always @(posedge clk) begin
		if (reset) begin
			contact <= 0;
		end
		if (~reset) begin
			if (range) begin
				contact <= 1;
			end

			if (~range) begin
				contact <= 0;

			end
		end
	end
endmodule

////////////////////////////////////////////////
// SqueezeDetect Module
// Author: Chuan Zhang
// The Squeezedetect module takes in contact from
// the contact detect module.  If contact makes
// a rising edge, then squeeze is asserted for 1/4
// of a second.  If contact makes a falling edge
// then unsqueeze is asserted for 1/4 of a second
////////////////////////////////////////////////


module squeezedetect (reset, clk, contact, squeeze, unsqueeze);
	input reset, clk, contact;

	output squeeze, unsqueeze;
	reg squeeze, unsqueeze;

	reg oldcontact;

	reg [25:0] counter;

	assign enter_contact = ((contact != oldcontact) && (contact == 1));
	assign exit_contact = ((contact != oldcontact) && (contact == 0));


	always @(posedge clk) begin
		if (reset) begin
			counter <= 5000000;
			squeeze <= 0;
			unsqueeze <= 0;
			oldcontact <= 0;
		end

		if (~reset) begin
			oldcontact <= contact;

			if (counter == 0) begin

				if (enter_contact) begin
					squeeze <= 1;
					unsqueeze <= 0;
					counter <= 5000000;
				end
				if (exit_contact) begin
					squeeze <= 0;
					unsqueeze <= 1;
					counter <= 5000000;
				end
				if (~enter_contact && ~exit_contact) begin
					squeeze <= 0;
```

```
                                        unsqueeze <= 0;
                                end
                        end

                        if (counter > 0) begin
                                counter <= counter - 1;
                        end

                end
        end


endmodule
```

```
module ninja_control(clk, reset, vsync, up, down, left, right,
                           throw, squeeze, unsqueeze, contact, mx, my,
                     xout, yout,
                           command, face, GSTATE, ninja_clear, regen,);
    input clk;
    input reset;

    input vsync;
    input up;
    input down;
    input left;
    input right;
         input throw;
         input squeeze, unsqueeze;
         input contact;

         input [11:0] mx, my;
         input ninja_clear, regen;                  // signals from AI for ninja regeneration after grab
         output jump_DONE;

         output[9:0] jumpCount;
         output start_index, jump_vinc;
         output[20:0] jump_index;
         output[2:0] JSTATE;


         parameter DEFAULT_Y = 600-106-50;    // Ninja at bottom-most
         parameter DEFAULT_X = 55;
         parameter MAX_WIDTH = 800-104;
         parameter MAX_HEIGHT = 600-100;
         parameter LEFT_EDGE = 50;
         parameter RIGHT_EDGE = 750;
         parameter TOP_EDGE = 50;
         parameter BOTTOM_EDGE = 550;
         parameter OFFSCR_X = 801;

        //--vsync states -------
         parameter NOT_VSYNC = 0;
         parameter VSNEW = 1;
         parameter VSOLD = 2;
        //-----------------------

         //--command parms---------
         parameter RIGHT = 0;
         parameter LEFT = 1;
         parameter UP = 2;
         parameter DOWN = 3;
         parameter NOCMD = 4;
         parameter THROW = 5;
         parameter GRABBED = 6;

         //------------------------

    output[11:0] xout;
    output[11:0] yout;
         output[2:0] command;
         output face;
         output[2:0] GSTATE;

         reg [11:0] xout, yout;
         reg[2:0] VSTATE, command;
         reg face, jump_DONE;

         always @(posedge clk)
               if (reset)
                      VSTATE <= NOT_VSYNC;
               else case(VSTATE)
                      NOT_VSYNC      : VSTATE <= vsync ? NOT_VSYNC : VSNEW;
                      VSNEW          : VSTATE <= vsync ? NOT_VSYNC : VSOLD;
                      VSOLD                  : VSTATE <= vsync ? NOT_VSYNC : VSOLD;
               endcase


//--- GRAB STATES------------------
               parameter NOT_GRABBED = 0;
               parameter JUST_GRABBED = 1;
               parameter LET_GO = 2;
               parameter ON_FLOOR = 3;
```

```
              parameter OFFSCR = 4;
              parameter REGEN = 5;
//-----------------------------

              reg[2:0] GSTATE;
              wire ninja_fall = (yout >= DEFAULT_Y);
              reg grabbed;

        always @(posedge clk)
              if (reset)
                     GSTATE <= NOT_GRABBED;
              else if (vsync || (VSTATE == VSOLD))
                     GSTATE <= GSTATE;
              else case (GSTATE)
                     NOT_GRABBED           : GSTATE <= grabbed ? JUST_GRABBED : NOT_GRABBED;
                     JUST_GRABBED   : GSTATE <= grabbed ? JUST_GRABBED : LET_GO;
                     LET_GO                : GSTATE <= ninja_fall ? ON_FLOOR : LET_GO;
                     ON_FLOOR                     : GSTATE <= ninja_clear ? OFFSCR : ON_FLOOR;
                     OFFSCR                : GSTATE <= regen ? REGEN : OFFSCR;
                     REGEN                        : GSTATE <= NOT_GRABBED;
              endcase

         // ---- JUMP STATES ------------
              parameter NO_JUMP = 0;
              parameter JUMP_UP = 1;
              parameter JUMP_UP_RIGHT = 2;
              parameter JUMP_UP_LEFT= 3;
              parameter END_JUMP = 4;
         // ----------------------------

              reg[9:0] jumpCount = 0;                     //to change velocity
              reg start_index, jump_vinc;   //start index count, jump_vinc is pulse that goes high when v
should change
              reg[20:0] jump_index;               // count up to 2^20 clock cycles
              reg[2:0] JSTATE;                                    // 4 states

              always @(posedge clk)
                     if (reset) begin
                            JSTATE <= NO_JUMP;
                            start_index <= 1'b0;
                            end
                     else if (vsync || (VSTATE == VSOLD)) begin
                            JSTATE <= JSTATE;
                            start_index <= start_index;
                            end
                     else case (JSTATE)
                            NO_JUMP              :
                                   if (up && right) begin
                                          JSTATE <= JUMP_UP_RIGHT;
                                          start_index <= 1'b1;
                                   end
                                   else if (up && left) begin
                                          JSTATE <= JUMP_UP_LEFT;
                                          start_index <= 1'b1;
                                   end
                                   else if (up) begin
                                          JSTATE <= JUMP_UP;
                                          start_index <= 1'b1;
                                          end
                                   else begin
                                          JSTATE <= NO_JUMP;
                                          start_index <= 1'b0;
                                   end
                            JUMP_UP              :
                                   if ((yout-6+jumpCount) >= DEFAULT_Y)
                                          JSTATE <= END_JUMP;
                                   else JSTATE <= JUMP_UP;

                            JUMP_UP_RIGHT  : if ((yout-6+jumpCount) >= DEFAULT_Y)
                                                                              JSTATE <= END_JUMP;
                                                                     else JSTATE <= JUMP_UP_RIGHT;
                            JUMP_UP_LEFT   :if ((yout-6+jumpCount) >= DEFAULT_Y) begin
                                                                              JSTATE <= END_JUMP;
                                                                              end
                                                                     else JSTATE <= JUMP_UP_LEFT;
                            END_JUMP                     : if (yout == DEFAULT_Y) begin
                                                                     JSTATE <= NO_JUMP;
                                                                     start_index <= 1'b0;
                                                                     end
```

```
                                                                        else JSTATE <= END_JUMP;
                        default begin
                         JSTATE <= NO_JUMP;
                         start_index <= 1'b0;
                         end
                endcase

        always @(posedge clk)
                if (reset)
                        jump_DONE = 1'b0;
                else if (JSTATE != END_JUMP)
                        jump_DONE = 1'b0;
                else if (vsync || (VSTATE == VSOLD))
                        jump_DONE = 1'b0;
                else if (JSTATE == END_JUMP)
                        jump_DONE = 1'b1;

        always @(posedge clk)
                if (reset)
                        jump_index <= 21'd0;
                else if (start_index == 1'b0)
                        jump_index <= 21'd0;
                else if (jump_vinc == 1'b1)
                        jump_index <= 21'd0;
                else if (start_index == 1'b1)
                        jump_index <= jump_index+1;

        always @(posedge clk)
                if (reset)
                        jump_vinc <= 1'b0;
                else if (jump_index == 5'b111111)
                        jump_vinc <= 1'b1;
                else jump_vinc <= 1'b0;


        always @(posedge clk)
                if (reset)
                        jumpCount <= 0;
                else if (JSTATE == NO_JUMP)
                        jumpCount <= 0;
                else if (jump_vinc == 1'b1)
                        jumpCount <= jumpCount+1;
                else jumpCount <= jumpCount;



 always @(posedge clk)
        if (reset)
                command = NOCMD;
        else if (grabbed)
                command = GRABBED;
        else if (right)
                command = RIGHT;
        else if (left)
                command = LEFT;
        else if (throw)
                command = THROW;
        else command = NOCMD;

reg [11:0] my_old, my_new, mx_old, mx_new;
reg signed [11:0] dmy, dmx;

always @(posedge clk)
        if (reset) begin
                my_old <= 0;
                my_new <= my;
                mx_old <= 0;
                mx_new <= mx;
                dmy <= 0;
                dmx <= 0;
        end else if (vsync || (VSTATE == VSOLD)) begin
                mx_old <= mx_old;
                mx_new <= mx_new;
                my_old <= my_old;
                my_new <= my_new;
                dmy <= dmy;
                dmx <= dmx;
        end else begin
                my_old <= my_new;
```

```
                my_new <= my;
                mx_old <= mx_new;
                mx_new <= mx;
                dmy <= my_new - my_old;
                dmx <= mx_new - mx_old;
        end



reg falling;
always @(posedge clk)
        if (GSTATE == LET_GO)
                falling = 1'b1;
        else falling = 1'b0;

reg fall_vc;
reg[20:0] fall_index;
always @(posedge clk)
        if (reset) fall_index <= 21'd0;
        else if (fall_vc == 1'b1)
                fall_index <= 21'd0;
        else if (falling)
                fall_index <= fall_index+1;
        else fall_index <= 21'd0;


always @(posedge clk)
        if (reset) fall_vc = 1'b0;
        else if (fall_index == 5'b11111)
                fall_vc = 1'b1;
        else fall_vc = 1'b0;

reg[9:0] fallCount;
always @(posedge clk)
        if (reset) fallCount <= 10'd0;
        else if (fall_vc == 1'b1)
                fallCount <= fallCount+1;
        else fallCount <= fallCount;



 always @(posedge clk)
        if (reset)
                begin
                        xout <= DEFAULT_X;
                        yout <= DEFAULT_Y;
                        face <= RIGHT;
                end
        else if (vsync || (VSTATE == VSOLD))
                begin
                        xout <= xout;
                        yout <= yout;
                        face <= face;
                end
        else if (GSTATE == REGEN)
                begin
                        xout <= DEFAULT_X;
                        yout <= DEFAULT_Y;
                        face <= RIGHT;
                end
        else if (GSTATE == ON_FLOOR)
                begin
                        xout <= xout;
                        yout <= yout;
                        face <= face;
                end
        else if (GSTATE == OFFSCR)
                begin
                        xout <= 10'd805;
                        yout <= DEFAULT_Y;
                        face <= RIGHT;
                end
        else if ((GSTATE == NOT_GRABBED) && (grabbed))
                begin
                        xout <= xout;
                        yout <= yout;
                        face <= face;
                end
        else if (GSTATE == JUST_GRABBED)
```

```
                begin
                        xout <= xout+dmx;
                        yout <= yout+dmy;
                        face <= face;
                end
        else if (GSTATE == LET_GO)
                begin
                        xout <= xout;
                        yout <= yout+6+fallCount;
                        face <= face;
                end
        else if (JSTATE == JUMP_UP) begin
                xout <= xout;
                if ((yout-9+jumpCount) > DEFAULT_Y)
                                yout <= DEFAULT_Y;
                                else yout <= yout-9+jumpCount;
                face <= face;
                end
        else if (JSTATE == JUMP_UP_RIGHT) begin
                xout <= xout+2;
                if ((yout-9+jumpCount) > DEFAULT_Y)
                                yout <= DEFAULT_Y;
                                else yout <= yout-9+jumpCount;
                face <= RIGHT;
                end
        else if (JSTATE == JUMP_UP_LEFT)
                begin
                xout <= xout-2;
                if ((yout-9+jumpCount) > DEFAULT_Y)
                                yout <= DEFAULT_Y;
                                else yout <= yout-9+jumpCount;
                face <= LEFT;
                end
        else if (JSTATE == END_JUMP)
                begin
                        xout <= xout;
                        yout <= DEFAULT_Y;
                        face <= face;
                end
        else if (yout != DEFAULT_Y)
                begin
                        xout <= xout;
                        yout <= DEFAULT_Y;
                        face <= face;
                end
        else if (right)
                begin
                xout <= xout+4;
                yout <= yout;
                face <= RIGHT;
                end
        else if (left)
                begin
                xout <= xout-4;
                yout <= yout;
                face <= LEFT;
                end
        else
                begin
                        xout <= xout;
                        yout <= yout;
                        face <= face;
                end


// GRAB LOGIC ----------------------------------------------------------
        parameter BOXWIDTH = 104;
        parameter BOXHEIGHT = 106;
        assign valid = ((mx[11:6]<=xout[11:6]+BOXWIDTH)&&(mx[11:6]>=xout[11:6])&&(my[11:6]<=yout[11:6]+BOXHEIGHT)
&& (my[11:6]>=yout[11:6]));

        always @(posedge clk) begin
                if (reset) begin
                        grabbed <= 0;
                end

                if (~reset) begin
                        if (grabbed) begin
                                if (~contact) begin
```

```
                                        grabbed <= 0;
                                end
                        end
                        if (~grabbed) begin
                                if (valid && squeeze) begin
                                        grabbed <= 1;
                                end
                        end
                end
        end




endmodule


module star_display(vclock, reset,
                                                        ninja_x, ninja_y,

                                                        target_x, target_y, target,
                                                        throwSwitch, walkSwitch, throw_assert, face,
                                                        GSTATE,
                                                        hcount, vcount, hsync, vsync, blank,
                                                        shsync, svsync, sblank, spixel,
                                                        x,y, throw_DONE, stop_throw, star_state);

    input vclock;
    input reset;
    input throwSwitch;
    input walkSwitch;
    input throw_assert;
    input face;                                         //0 for right, 1 for left
    input[10:0] hcount;
    input[9:0] vcount;
    input hsync;
    input vsync;
    input blank;
        input[11:0] ninja_x;
        input[11:0] ninja_y;
        input[9:0] target_x, target_y;
        input target;                                   // if throwing at target HIGH
        input[2:0] GSTATE;

    output shsync;
    output svsync;
    output sblank;
    output[2:0] spixel;
        output[11:0] x,y;
        output throw_DONE;
        output stop_throw;
        output[2:0] star_state;



        assign shsync = hsync;
        assign svsync = vsync;
        assign sblank = blank;

        //---FACES--------------
        parameter RIGHT = 0;
        parameter LEFT = 1;
        //-----------------------

        //---STAR STATES ------
        parameter DONT_SHOW = 0;
        parameter THROW_ASSERTED = 1;
        parameter SHOW_STAR = 2;
        parameter STAR_MOVING = 3;
        parameter STAR_THROWN = 4;
        //------------------------

               //---GSTATEs---------
        parameter NOT_GRABBED = 0;
        parameter JUST_GRABBED = 1;
        parameter LET_GO = 2;
        parameter ON_FLOOR = 3;
        parameter OFFSCR = 4;
```

```
//-------------------------

 //--vsync states -------
 parameter NOT_VSYNC = 0;
 parameter VSNEW = 1;
 parameter VSOLD = 2;
//-----------------------

//---test states---
parameter no_throw_TEST = 0;
parameter throw_TEST = 1;
//-------------------
reg[2:0] star_state;
reg[11:0] x,y;
reg[2:0] VSTATE;
reg throw_DONE, stop_throw;

always @(posedge vclock)
        if (reset)
                VSTATE <= NOT_VSYNC;
        else case(VSTATE)
                NOT_VSYNC       : VSTATE <= vsync ? NOT_VSYNC : VSNEW;
                VSNEW           : VSTATE <= vsync ? NOT_VSYNC : VSOLD;
                VSOLD                   : VSTATE <= vsync ? NOT_VSYNC : VSOLD;
        endcase

parameter DEFAULT_X = 850;
parameter X_SPEED = 6;
parameter X_HAND_RIGHT = 99;

always@(posedge vclock)
        if (reset) x <= DEFAULT_X;
        else if (vsync || (VSTATE == VSOLD))
                x <= x;
        else case (star_state)
                        DONT_SHOW               : x <= DEFAULT_X;
                        THROW_ASSERTED : x <= DEFAULT_X;
                        SHOW_STAR               : x <= (face == RIGHT) ? ninja_x+X_HAND_RIGHT : ninja_x;
                        STAR_MOVING             : x <= (face == RIGHT) ? x+X_SPEED : x-X_SPEED;
                        STAR_THROWN             : x <= DEFAULT_X;
                endcase

 parameter WIDTH = 750;
 parameter HEIGHT = 550;

 parameter DEFAULT_Y = 549-67;



 wire xOFF = x > WIDTH;
 wire yOFF = y > HEIGHT;
 wire starOffScreen = xOFF || yOFF;


 always @(posedge vclock)
        if (reset) begin
                star_state <= DONT_SHOW;
                throw_DONE <= 1'b0;
                stop_throw <= 1'b0;
                end
        else case (star_state)
                DONT_SHOW               : begin
                                                star_state <=  throw_assert ? THROW_ASSERTED : DONT_SHOW;
                                                throw_DONE <= 1'b0;
                                                        end
                THROW_ASSERTED :
                                if (~throw_assert)
                                        star_state <= DONT_SHOW;
                                else if (GSTATE != NOT_GRABBED)
                                        star_state <= DONT_SHOW;
                                else if (throwSwitch)
                                        star_state <= SHOW_STAR;
                                else
                                        star_state <= THROW_ASSERTED;
                SHOW_STAR               :
                                if (~throw_assert)
                                        star_state <= DONT_SHOW;
                                else if (walkSwitch)
                                        star_state <= STAR_MOVING;
```

```
                                else star_state <= SHOW_STAR;
                        STAR_MOVING             :
                                if (starOffScreen) begin
                                        star_state <= STAR_THROWN;
                                        stop_throw <= 1'b1;
                                        end

                                else star_state <= STAR_MOVING;
                        STAR_THROWN     :
                                if (~throw_assert) begin
                                        star_state <= DONT_SHOW;
                                        stop_throw <= 1'b0;
                                        throw_DONE <= 1'b1;
                                        end
                                else star_state <= STAR_THROWN;
                        default star_state <= DONT_SHOW;
                endcase

        wire ninja_on_right = ((ninja_x+3'd5) <= target_x);
        wire ninja_lower = (ninja_y >= target_y);


//      wire [9:0] Rx, Ry;
//      assign Ry = ninja_lower ? (DEFAULT_Y - ninja_y) : (ninja_y - DEFAULT_Y);
//      assign Rx = ninja_on_right ? ((ninja_x+3'd5) - target_x) : (target_x - (ninja_x + 7'd98));
//      wire [12:0] numerator = Ry*10;
//      wire [9:0] Ratio_y, remainder;
//      wire rfd;       //ready for data
//      divider yx (numerator, Rx, Ratio_y, remainder, vclock, rfd, 1'b0, reset, 1'b1);

        reg[9:0] dy;

        always @(posedge vclock)
                if (reset)
                        y <= DEFAULT_Y;
                else if (vsync || (VSTATE == VSOLD))
                        y <= y;
                else case (star_state)
                        DONT_SHOW             : y <= DEFAULT_Y;
                        THROW_ASSERTED : y <= DEFAULT_Y;
                        SHOW_STAR             : y <= ninja_y+39;
                        STAR_MOVING           : y <= target ? y-5 : DEFAULT_Y;
                        STAR_THROWN           : y <= DEFAULT_Y;
                endcase

         wire[2:0] starblob_pix;
         blob starblob (vclock, x, y, hcount, vcount, starblob_pix);
         defparam starblob.HEIGHT = 5;
         defparam starblob.WIDTH = 5;
         defparam starblob.COLOR = 3'b111;

        assign spixel = starblob_pix;

endmodule



module ninjaAI(vclock, reset, AIsel, up, down, left, right, throw, ninja_x, ninja_y,
                                GSTATE, mx, my, ninja_clear, regen, throw_DONE, stop_throw, jump_DONE,
                                gtSTATE,       gtCOUNT, gtCMD);
    input vclock;
    input reset;
         input[1:0] AIsel;
    output up;
    output down;
    output left;
    output right;
    output throw;
    input[11:0] ninja_x;
    input[11:0] ninja_y;
         input[11:0] mx, my;
         input[2:0] GSTATE;
         input throw_DONE, stop_throw, jump_DONE;
         output ninja_clear, regen;
         /*output[3:0] throwerSTATE;*/
         /*output[24:0] stopTurnAround;*/

/*       output[1:0] chickenSTATE;
         output[4:0] chicken_cmd;
```

```
*/
	output[20:0] gtCOUNT;
	output[4:0] gtCMD;
	output[3:0] gtSTATE;
	//---GSTATEs---------
	 parameter NOT_GRABBED = 0;
	 parameter JUST_GRABBED = 1;
	 parameter LET_GO = 2;
	 parameter ON_FLOOR = 3;
	 parameter OFFSCR = 4;
	//------------------------

	parameter NINJA_FLOOR_TIME = 40000000;
	parameter NINJA_OFFSCREEN_TIME = 40000000;
	reg[25:0] time_floor_count, time_offscreen_count;
	reg ninja_clear, regen;

	always @(posedge vclock)
		if (reset) begin
			time_floor_count <= 0;
			ninja_clear <= 1'b0;
		end else if (time_floor_count == NINJA_FLOOR_TIME) begin
			ninja_clear <= 1'b1;
			time_floor_count <= 0;
		end else if (GSTATE == ON_FLOOR) begin
			ninja_clear <= 1'b0;
			time_floor_count <= time_floor_count+1;
		end else begin
			ninja_clear <= 1'b0;
			time_floor_count <= 0;
		end

	always @(posedge vclock)
		if (reset) begin
			time_offscreen_count <= 0;
			regen <= 1'b0;
		end else if (time_offscreen_count == NINJA_OFFSCREEN_TIME) begin
			time_offscreen_count <= 0;
			regen <= 1'b1;
		end else if (GSTATE == OFFSCR) begin
			time_offscreen_count <= time_offscreen_count+1;
			regen <= 1'b0;
		end else begin
			time_offscreen_count <= 0;
			regen <= 1'b0;
		end



	 parameter RIGHT_EDGE = 644;
	 parameter LEFT_EDGE = 50;
	 wire nx_right_bound = (ninja_x > RIGHT_EDGE);
	 wire nx_left_bound   = (ninja_x < LEFT_EDGE);

	 // "goomba" AI
	 parameter GOOMBA_RIGHT = 0;
	 parameter GOOMBA_LEFT = 1;
	 parameter GOOMBA_GRABBED = 2;
	 parameter GOOMBA_FALLING = 3;
	 parameter GOOMBA_FLOOR = 4;
	 parameter GOOMBA_OFFSCREEN = 5;

	 reg[2:0] goombaSTATE;
	 reg[4:0] goomba_cmd;

	 always @(posedge vclock)
		if (reset)
				goombaSTATE = GOOMBA_RIGHT;
		else if (GSTATE == JUST_GRABBED)
				goombaSTATE = GOOMBA_GRABBED;
		else case(goombaSTATE)
			GOOMBA_RIGHT          : goombaSTATE = nx_right_bound ? GOOMBA_LEFT : GOOMBA_RIGHT;
			GOOMBA_LEFT                  : goombaSTATE = nx_left_bound ? GOOMBA_RIGHT : GOOMBA_LEFT;
			GOOMBA_GRABBED        : goombaSTATE = GOOMBA_FALLING;
			GOOMBA_FALLING        : goombaSTATE = (GSTATE == ON_FLOOR) ? GOOMBA_FLOOR :
GOOMBA_FALLING;
			GOOMBA_FLOOR          : goombaSTATE = ninja_clear ? GOOMBA_OFFSCREEN : GOOMBA_FLOOR;
			GOOMBA_OFFSCREEN  : goombaSTATE = regen ? GOOMBA_RIGHT : GOOMBA_OFFSCREEN;
	endcase
```

```
 always @(posedge vclock)
        if (reset)
                goomba_cmd = 5'b00000;
        else case (goombaSTATE)
                GOOMBA_RIGHT   :       goomba_cmd = 5'b00010;
                GOOMBA_LEFT            :       goomba_cmd = 5'b00100;
                GOOMBA_GRABBED :  goomba_cmd = 5'b00000;
                GOOMBA_FALLING :       goomba_cmd = 5'b00000;
                GOOMBA_FLOOR   :  goomba_cmd = 5'b00000;
                GOOMBA_OFFSCREEN : goomba_cmd = 5'b00000;
                default goomba_cmd = GOOMBA_RIGHT;
        endcase

parameter GT_RIGHT = 0;
parameter GT_REDGE = 1;
parameter GT_REDGEL = 2;
parameter GT_RTHROW = 3;
parameter GT_LEFT = 4;
parameter GT_LEDGE = 5;
parameter GT_LEDGER = 6;
parameter GT_LTHROW = 7;
parameter GT_STOPTHROW1 = 8;
parameter GT_STOPTHROW2 = 9;

reg[3:0] gtSTATE;
reg[4:0] gtCMD;
reg[19:0] gtCOUNT;

always @(posedge vclock)
        if (reset)
                gtCOUNT <= 20'd0;
        else if (gtSTATE == GT_REDGEL)
                gtCOUNT <= gtCOUNT+1;
        else if (gtSTATE == GT_LEDGER)
                gtCOUNT <= gtCOUNT+1;
        else gtCOUNT <= 20'd0;

always @(posedge vclock)
        if (reset)
                gtSTATE <= GT_RIGHT;
        else case (gtSTATE)
                GT_RIGHT               :       gtSTATE <= nx_right_bound ? GT_REDGE : GT_RIGHT;
                GT_REDGE               :       gtSTATE <= GT_REDGEL;
                GT_REDGEL      :       gtSTATE <= (gtCOUNT == 5'b11111) ? GT_RTHROW : GT_REDGEL;
                GT_RTHROW      :       gtSTATE <= stop_throw ? GT_STOPTHROW1 : GT_RTHROW;
                GT_STOPTHROW1 : gtSTATE <= throw_DONE ? GT_LEFT : GT_STOPTHROW1;
                GT_LEFT        : gtSTATE <= nx_left_bound ? GT_LEDGE : GT_LEFT;
                GT_LEDGE               : gtSTATE <= GT_LEDGER;
                GT_LEDGER      : gtSTATE <= (gtCOUNT == 5'b11111) ? GT_LTHROW : GT_LEDGER;
                GT_LTHROW      : gtSTATE <= stop_throw ? GT_STOPTHROW2 : GT_LTHROW ;
                GT_STOPTHROW2 : gtSTATE <= throw_DONE ? GT_RIGHT : GT_STOPTHROW2;
        default gtSTATE <= GT_RIGHT;
        endcase

always @(posedge vclock)
        if (reset)
                gtCMD = 5'b00000;
        else case (gtSTATE)
                GT_RIGHT               :       gtCMD = 5'b00010;
                GT_REDGE               :       gtCMD   = 5'b00000;
                GT_REDGEL              :       gtCMD   = 5'b00100;
                GT_RTHROW              :       gtCMD   = 5'b00001;
                GT_STOPTHROW1  :       gtCMD = 5'b00000;
                GT_LEFT                :       gtCMD = 5'b00100;
                GT_LEDGE                       :       gtCMD   = 5'b00000;
                GT_LEDGER              :       gtCMD   = 5'b00010;
                GT_LTHROW              :       gtCMD   = 5'b00001;
                GT_STOPTHROW2  :       gtCMD   = 5'b00000;
        default gtSTATE = 5'b00010;
        endcase

//      //thrower AI
//
//      parameter THROWER_JUMP_IN = 0;
//      parameter THROWER_THROW_1 = 1;
//      parameter THROWER_WALK_RIGHT = 2;
//      parameter THROWER_TURN_RL = 3;
//      parameter THROWER_AT_RIGHT = 4;
```

```
//      parameter THROWER_JUMP_LEFT = 5;
//      parameter THROWER_THROW_2 = 6;
//      parameter THROWER_WALK_LEFT = 7;
//      parameter THROWER_TURN_LR = 8;
//      parameter THROWER_THROW_3 = 9;
//      parameter THROWER_GRABBED = 10;
//      parameter THROWER_FALLING = 11;
//      parameter THROWER_FLOOR = 12;
//      parameter THROWER_OFFSCREEN = 13;
//
//
//      reg[3:0] throwerSTATE;
//      reg[24:0] stopTurnAround;
//      reg sTA;
//      parameter STOPTA = 20000000; //change this to 20000000
//
//      always @(posedge vclock)
//              if (reset) begin
//                      stopTurnAround <= 24'd0;
//                      sTA <= 1'b0;
//                      end
//              else if (stopTurnAround ==    STOPTA) begin
//                      stopTurnAround <= 24'd0;
//                      sTA <= 1'b1;
//                      end
//              else if (throwerSTATE == THROWER_TURN_RL) begin
//                      stopTurnAround <= stopTurnAround+1;
//                      sTA <= 1'b0;
//                      end
//              else if (throwerSTATE == THROWER_AT_RIGHT) begin
//                      stopTurnAround <= stopTurnAround+1;
//                      sTA <= 1'b0;
//                      end
//              else if (throwerSTATE == THROWER_TURN_LR) begin
//                      stopTurnAround <= stopTurnAround+1;
//                      sTA <= 1'b0;
//                      end
//              else begin stopTurnAround <= stopTurnAround;
//                                      sTA <= 1'b0; end
//
//      always @(posedge vclock)
//              if (reset)
//                      throwerSTATE = THROWER_JUMP_IN;
//              else if (GSTATE == JUST_GRABBED)
//                      throwerSTATE = THROWER_GRABBED;
//              else if (nx_right_bound)
//                      throwerSTATE = THROWER_TURN_RL;
//              else if (nx_left_bound)
//                      throwerSTATE = THROWER_TURN_LR;
//              else case (throwerSTATE)
//                      THROWER_JUMP_IN              : throwerSTATE = jump_DONE ? THROWER_THROW_1 :
THROWER_JUMP_IN;
//                      THROWER_THROW_1              : throwerSTATE = throw_DONE ? THROWER_WALK_RIGHT :
THROWER_THROW_1;
//                      THROWER_WALK_RIGHT     : throwerSTATE = nx_right_bound ? THROWER_TURN_RL :
THROWER_WALK_RIGHT;
//                      THROWER_TURN_RL              : throwerSTATE = sTA  ? THROWER_AT_RIGHT : THROWER_TURN_RL;
//                      THROWER_AT_RIGHT             : throwerSTATE = throw_DONE ? THROWER_JUMP_LEFT :
THROWER_AT_RIGHT;
//                      THROWER_JUMP_LEFT            : throwerSTATE = jump_DONE ? THROWER_THROW_2 :
THROWER_JUMP_LEFT;
//                      THROWER_THROW_2              : throwerSTATE = throw_DONE ? THROWER_WALK_LEFT :
THROWER_THROW_2;
//                      THROWER_WALK_LEFT            : throwerSTATE = nx_left_bound ? THROWER_TURN_LR :
THROWER_WALK_LEFT;
//                      THROWER_TURN_LR              : throwerSTATE = sTA ? THROWER_THROW_3 : THROWER_TURN_LR;
//                      THROWER_THROW_3              : throwerSTATE = throw_DONE ? THROWER_JUMP_IN :
THROWER_THROW_3;
//                      THROWER_GRABBED              : throwerSTATE = THROWER_FALLING;
//                      THROWER_FALLING              : throwerSTATE = (GSTATE == ON_FLOOR) ? THROWER_FLOOR :
THROWER_FALLING;
//                      THROWER_FLOOR                : throwerSTATE = ninja_clear ? THROWER_OFFSCREEN :
THROWER_FLOOR;
//                      THROWER_OFFSCREEN            : throwerSTATE = regen ? THROWER_JUMP_IN :
THROWER_OFFSCREEN;
//              default throwerSTATE = THROWER_JUMP_IN;
//              endcase
//
//      reg[4:0] thrower_cmd;
```

```
//      always @(posedge vclock)
//              if (reset)
//                      thrower_cmd = 5'b00000;
//              else if (stop_throw)
//                      thrower_cmd = 5'b00000;
//              else case (throwerSTATE)
//                      THROWER_JUMP_IN              : thrower_cmd = 5'b10010;
//                      THROWER_THROW_1              : thrower_cmd = 5'b00001;
//                      THROWER_WALK_RIGHT     : thrower_cmd = 5'b00010;
//                      THROWER_TURN_RL              : thrower_cmd = 5'b00100;
//                      THROWER_AT_RIGHT             : thrower_cmd = 5'b00001;
//                      THROWER_JUMP_LEFT            : thrower_cmd = 5'b10100;
//                      THROWER_THROW_2              : thrower_cmd = 5'b00001;
//                      THROWER_WALK_LEFT            : thrower_cmd = 5'b00100;
//                      THROWER_TURN_LR              : thrower_cmd = 5'b00010;
//                      THROWER_THROW_3              : thrower_cmd = 5'b00001;
//                      THROWER_GRABBED              : thrower_cmd = 5'b00000;
//                      THROWER_FALLING              : thrower_cmd = 5'b00000;
//                      THROWER_FLOOR                : thrower_cmd = 5'b00000;
//                      THROWER_OFFSCREEN            : thrower_cmd = 5'b00000;
//              default thrower_cmd = 5'b00000;
//              endcase




        parameter CHICKEN_LEFT = 0;
        parameter CHICKEN_RIGHT = 1;
        parameter CHICKEN_JUMP = 2;
        parameter CHICKEN_WAIT= 3;

        parameter THRESHHOLD_X_LEFT = 250;
        parameter THRESHHOLD_X_RIGHT = 500;

        reg[1:0] chickenSTATE;
        reg[4:0] chicken_cmd;

        always @(posedge vclock)
                if (reset)
                        chickenSTATE <= CHICKEN_RIGHT;
                else if (mx <= 6'd50)
                        chickenSTATE <= CHICKEN_RIGHT;
                else if (mx >= 10'd740)
                        chickenSTATE <= CHICKEN_LEFT;
                else if ((chickenSTATE == CHICKEN_RIGHT) && (nx_right_bound))
                        chickenSTATE <= CHICKEN_WAIT;
                else if ((chickenSTATE == CHICKEN_LEFT) && (nx_left_bound))
                        chickenSTATE <= CHICKEN_WAIT;
                else case (chickenSTATE)
                        CHICKEN_LEFT   :
                                if ((ninja_x < THRESHHOLD_X_LEFT) && (mx < THRESHHOLD_X_LEFT))
                                        chickenSTATE <= CHICKEN_JUMP;
                                else if ((ninja_x > THRESHHOLD_X_RIGHT) && (mx > THRESHHOLD_X_RIGHT))
                                        chickenSTATE <= CHICKEN_JUMP;
                                else if (nx_left_bound || (ninja_x == 6'd50))
                                        chickenSTATE <= CHICKEN_WAIT;
                                else if (nx_right_bound || (ninja_x == 10'd644))
                                        chickenSTATE <= CHICKEN_WAIT;
                                else if (mx <= ninja_x)
                                        chickenSTATE <= CHICKEN_RIGHT;
                                else if (mx >= ninja_x)
                                        chickenSTATE <= CHICKEN_LEFT;
                                else chickenSTATE <= CHICKEN_LEFT;
                        CHICKEN_RIGHT :
                                if ((ninja_x > THRESHHOLD_X_RIGHT) && (mx > THRESHHOLD_X_RIGHT))
                                        chickenSTATE <= CHICKEN_JUMP;
                                else if ((ninja_x < THRESHHOLD_X_RIGHT) && (mx < THRESHHOLD_X_RIGHT))
                                        chickenSTATE <= CHICKEN_JUMP;
                                else if (nx_right_bound || (ninja_x == 10'd644))
                                        chickenSTATE <= CHICKEN_WAIT;
                                else if (nx_left_bound || (ninja_x == 6'd50))
                                        chickenSTATE <= CHICKEN_WAIT;
                                else if (mx > ninja_x)
                                        chickenSTATE <= CHICKEN_LEFT;
                                else chickenSTATE <= CHICKEN_RIGHT;
                        CHICKEN_JUMP :
                                if (jump_DONE == 1'b0)
                                        chickenSTATE <= CHICKEN_JUMP;
```

```
                        else if (mx >= THRESHHOLD_X_RIGHT)
                               chickenSTATE <= CHICKEN_LEFT;
                        else if (mx < THRESHHOLD_X_LEFT)
                               chickenSTATE <= CHICKEN_RIGHT;

                CHICKEN_WAIT :
                        if ((chickenSTATE == CHICKEN_RIGHT) && (nx_right_bound))
                                       chickenSTATE <= CHICKEN_LEFT;
                        else if ((chickenSTATE == CHICKEN_LEFT) && (nx_left_bound))
                                       chickenSTATE <= CHICKEN_RIGHT;
                        else if (mx > THRESHHOLD_X_RIGHT)
                               chickenSTATE <= CHICKEN_LEFT;
                        else if (mx < THRESHHOLD_X_LEFT)
                               chickenSTATE <= CHICKEN_RIGHT;
                        else chickenSTATE <= CHICKEN_WAIT;

        endcase

    always @(posedge vclock)
        if (reset)
                chicken_cmd = 5'b00000;
        else case (chickenSTATE)
                CHICKEN_LEFT   : chicken_cmd = 5'b00100;
                CHICKEN_RIGHT  : chicken_cmd = 5'b00010;
                CHICKEN_JUMP   : chicken_cmd = 5'b10000;
                CHICKEN_WAIT   : chicken_cmd = 5'b00000;
                default chicken_cmd = 5'b00000;
        endcase




    assign up             = (AIsel == 2'b00) ? goomba_cmd[4] /* (AIsel == 2'b01) ? chicken_cmd[4] */:
    gtCMD[4];
    assign down    = (AIsel == 2'b00) ? goomba_cmd[3] /* (AIsel == 2'b01) ? chicken_cmd[3]    */:    gtCMD
[3];
    assign left    = (AIsel == 2'b00) ? goomba_cmd[2]  /*(AIsel == 2'b01) ?  chicken_cmd[2]*/ :      gtCMD
[2];
    assign right   = (AIsel == 2'b00) ? goomba_cmd[1]  /*(AIsel == 2'b01) ?  chicken_cmd[1]*/ :      gtCMD
[1];
    assign throw   = (AIsel == 2'b00) ? goomba_cmd[0]  /*(AIsel == 2'b01) ?  chicken_cmd[0]*/ :      gtCMD
[0];



endmodule



module health_display(vclock, reset, hcount, vcount, hsync, vsync, blank, dec, inc, hRGB,
                                                        hsync_health, vsync_health, blank_health);
    input vclock;
    input reset;
    input[10:0] hcount;
    input[9:0] vcount;
        input hsync, vsync, blank;
        input dec;
        input inc;
    output[2:0] hRGB;
        output hsync_health, vsync_health, blank_health;

        assign hsync_health = hsync;
        assign vsync_health = vsync;
        assign blank_health = blank;

        parameter DEFAULT_HEALTH = 300;
        parameter HEALTH_BAR_TOP = 25;
        parameter  HEALTH_BAR_BOTTOM = 40;
        reg[9:0] health;
        reg hRGB;

        always @(posedge vclock)
              if(reset)
                      health <= 8'd300;
```

```
                else if (dec)
                        health <= health - 25;
                else if (inc)
                        health <= health + 25;
                else health <= health;

                wire xRange    = (hcount >= 25) && (hcount < DEFAULT_HEALTH);
                wire yRange = (vcount >= HEALTH_BAR_TOP) && (vcount < HEALTH_BAR_BOTTOM);
                wire validRange = (xRange && yRange);

        always @(posedge vclock)
                if (validRange && (hcount <= health))
                        hRGB = 3'b010;
                else if (validRange)
                        hRGB = 3'b100;
                else hRGB = 3'b000;



endmodule




module Ninja_display(vclock, reset,
                                                x, y,
                                               walkSwitch, throwSwitch, command, face,
                                               hcount, vcount, hsync, vsync, blank,
                                               nhsync, nvsync, nblank, npixel);

    input vclock;                                       // 40 mhz clock
    input reset;                                        // to help initialize module
    input[10:0] hcount;                                // xvga horizontal count (0...799)
    input[9:0] vcount;                          // xvga vertical count  (0...599)
    input hsync;                                        // XVGA output Horizontal Sync Signal (active LOW)
    input vsync;                                        // XVGA output Vertical Sync Signal (active LOW)
    input blank;                                        // XVGA output blank signal (1 means output Black pixel)

        input [11:0] x;                                         // x coordinate of where ninja should be displayed
        input [11:0] y;                                         // y coordinate of where ninja should be displayed
        input walkSwitch;                                               // goes HIGH when walk frame should be
switched
        input throwSwitch;                                              //goes HIGH when throwFrame should be
switched
        input face;                                                             // RIGHT of LEFT direction of ninja
        input[2:0] command;

       // ninjas vga signals
    output nhsync;
    output nvsync;
    output nblank;
    output[7:0] npixel;

        assign nhsync = hsync;
        assign nvsync = vsync;
        assign nblank = blank;

        // Width and Height dimensions for different animation frames                      49 x 23
        parameter WIDTH = 104;
        parameter HEIGHT = 106;
        parameter GRABBED_W = 92;
        parameter GRABBED_H = 88;

        //----Walk Frames---------
        parameter W1 = 0;
        parameter W2 = 1;
        parameter W3 = 2;
        //-----------------------

        //---Throw Frames------
        parameter T1 = 0;
        parameter T2 = 1;
        parameter T3 = 2;
        parameter T4 = 5;
        parameter THROWN = 3;
        parameter NOT_THROWN = 4;
        //---------------------
```

```
        //---Command parms-------
        parameter RIGHT = 0;
        parameter LEFT = 1;
        parameter UP = 2;
        parameter DOWN = 3;
        parameter NOCMD = 4;
        parameter THROW = 5;
        parameter GRABBED = 6;
        //----------------------

        //---face states ----------
        parameter RIGHT_FACE = 0;
        parameter LEFT_FACE = 1;
        //------------------------

        reg facestate;
        reg[2:0] walkState, throwState;
        reg[13:0] addr, addrREV;
        reg[12:0] addr_grabbed, addr_grabbedREV;
        wire[2:0] starBlob;

        wire xRange          = (hcount >= x) && (hcount < (x+WIDTH));
        wire xRange_grab = (hcount >= x) && (hcount < (x+GRABBED_W));
        wire yRange          = (vcount >= y) && (vcount < (y+HEIGHT));
        wire yRange_grab = (vcount >= y) && (vcount < (y+GRABBED_H));
        wire validRange      = (xRange && yRange);
        wire validRange_grab  = (xRange_grab && yRange_grab);


       //"Reverse" address function
       //assigns the address to red from pixel to
       // invert image from left to right

       //walk addresses
       always @(posedge vclock)
              if (hcount ==0 && vcount ==0)
                     begin
                            addrREV                             <= WIDTH-1;
                            addr_grabbedREV       <= GRABBED_W-1;
                     end
              else if (x < 0)
                     begin
                            addrREV                             <= ((vcount-y)*((hcount-x)+WIDTH))+((hcount-
x)+WIDTH);
                            addr_grabbedREV       <= ((vcount-y)*((hcount-x)+GRABBED_W))+((hcount-x)
+GRABBED_W);
                     end
              else begin
                     addrREV                       <= (validRange) ? ((vcount-y)*WIDTH)+((WIDTH-1)-(hcount-x))
: addrREV;
                     addr_grabbedREV <= (validRange_grab) ? ((vcount-y)*GRABBED_W)+(GRABBED_W-(hcount-x)) :
addr_grabbedREV;
              end

       //regular address blocks
        always @(posedge vclock)
              if (hcount == 0 && vcount == 0)
                     addr <= 0;
              else if (validRange)
                     addr <= addr+1;
              else if (addr > 14'd11025)
                     addr <= 0;
              else
                     addr <= addr;


        always @(posedge vclock)
              if (hcount == 0 && vcount == 0)
                     addr_grabbed <= 0;
              else if (validRange_grab)
                     addr_grabbed <= addr_grabbed+1;
              else if (addr_grabbed > 13'd8096)
                     addr_grabbed <= 0;
              else
                     addr_grabbed <= addr_grabbed;



       wire[7:0] walk1, walk2, walk3,
```

```
                              walk_rev1, walk_rev2, walk_rev3,
                              walk_left, walk_right, walking,

                              grabbed, grabbed_rev,

                              throw1, throw2, slash,
                              throw_rev1, throw_rev2, slash_rev,
                              throw_left, throw_right, throwing;

        wire STAND_TRUE = (command == NOCMD);
        wire WALK_TRUE = ((command == LEFT) || (command == RIGHT));  //assert if there is a walk cmd
        wire THROW_TRUE = (command == THROW);                                                    //assert if
there is a throw cmd
        wire GRAB_TRUE = (command == GRABBED);

        always @(posedge vclock)
                if (walkSwitch)
                        begin
                                case (walkState)
                                        W1 : walkState = W2;
                                        W2 : walkState = W3;
                                        W3 : walkState = W1;
                                endcase
                        end
                else walkState = walkState;

        always @(posedge vclock)
                if (reset)
                        throwState = THROWN;
                else case (throwState)
                        T1                      : throwState = throwSwitch ? T2 :
                                                                                        (command == THROW) ?
T1 : NOT_THROWN;
                        T2                      : throwState = walkSwitch  ? T3 :
                                                                                        (command == THROW) ?
T2 : NOT_THROWN;
                        T3                              : throwState = throwSwitch ? THROWN :
                                                                                        (command == THROW) ?
T3 : NOT_THROWN;
                        THROWN          : throwState = (command == THROW) ? THROWN : NOT_THROWN;
                        NOT_THROWN  : throwState = (command == THROW) ?  T1 : NOT_THROWN;
                        default throwState = NOT_THROWN;
                endcase

    //--- Walk Frame Logic ----------------------------------
        assign walk_right = (walkState == W1) ? walk1 :
                                                        (walkState == W2) ? walk2 : walk3;
        assign walk_left = (walkState == W1) ? walk_rev1 :
                                                        (walkState == W2) ? walk_rev2 : walk_rev3;
        assign standing = (face == LEFT) ? walk_rev3 : walk3;
        assign walking = (command == LEFT) ? walk_left : walk_right;
        //-------------------------------------------------------------

        //----  Throw frame logic ---------------------------------------

        wire showThrow = (throwState != NOT_THROWN);        //should throw be animated?

        assign throw_right = (throwState == T1) ? throw1+starBlob :
                             (throwState == T2) ? throw2+starBlob :
                                                        (throwState == T3) ? throw2+starBlob :
                                                        (validRange) ? walk3+starBlob : 8'b11111111;

        assign throw_left = (throwState == T1) ? throw_rev1+starBlob :
                                                (throwState == T2) ? throw_rev2+starBlob :
                                                    (throwState == T3) ? throw_rev2+starBlob :
                                                    (validRange) ? walk_rev3+starBlob : 8'b11111111;
        //-------------------------------------------------------------



        assign npixel = (GRAB_TRUE && (validRange_grab) && (face == RIGHT)) ? grabbed :
                                        (GRAB_TRUE && (validRange_grab) && (face == LEFT)) ? grabbed_rev :
                                        (WALK_TRUE && (validRange) && (face == RIGHT)) ? walk_right :
                                        (WALK_TRUE && (validRange) && (face == LEFT)) ? walk_left :
                                        (THROW_TRUE && (validRange) && (face == RIGHT)) ? (throw_right) :
                                        (THROW_TRUE && (validRange) && (face == LEFT)) ? (throw_left) :
                                        (STAND_TRUE && (validRange) && (face == RIGHT)) ? walk3:
                                        (STAND_TRUE && (validRange) && (face == LEFT)) ? walk_rev3 :
```

```
                                        8'b11111111;


        // ninja pixel reads from ROMS
        // FACING RIGHT
        ninja_walk_1 walk_f1(addr, vclock, walk1);  //not walking frame
        ninja_walk_2 walk_f2(addr, vclock, walk2);
        ninja_walk_3 walk_f3(addr, vclock, walk3);

        ninja_grabbed ng (addr_grabbed, vclock, grabbed);

        ninja_throw_1 nt1 (addr, vclock, throw1);
        ninja_throw_2 nt2 (addr, vclock, throw2);

        // FACING LEFT
        ninja_walk_1 walk_l1(addrREV, vclock, walk_rev1);
        ninja_walk_2 walk_l2(addrREV, vclock, walk_rev2);
        ninja_walk_3 walk_l3(addrREV, vclock, walk_rev3);

        ninja_grabbed ngr (addr_grabbedREV, vclock, grabbed_rev);

        ninja_throw_1 nt1r (addrREV, vclock, throw_rev1);
        ninja_throw_2 nt2r (addrREV, vclock, throw_rev2);


endmodule
```