Poke Kirby with a Laser
(Kirby Laser Attack)
(Kirby vs King Dedede)
(Super Ultra Laser Attack Game)

Daniel Gerber
Tynan Smith

Abstract

This project deals with gesture recognition of data produced by the coordinates of a laser on a screen. The setup will involve a video camera pointed at a white screen. Data from the video camera is analyzed to determine the coordinates of the laser which are then further processed to determine the overall motion of the laser. This is used as input to a simple multiplayer sprite-based combat video game. The game features Kirby and King Dedede as the main characters, with their movement controlled by keyboard input and their attacks controlled by gestures from a red and green laser pointer respectively.

Table of Contents

List of Figures

1. Overview

This project is a game of many names, Kirby Laser Attack, Poke Kirby with a Laser, Kirby vs King Dedede (super ultra laser attack game) and many more. However, none of them can quite convey the true awesomeness that is this game. This is a side-view two-dimensional fighting game similar in style to the Super Smash Brothers games. It is a two or four player game in which the players work by themselves or with a teammate to control one of two characters, Kirby or King Dedede. The movement of the characters (jumping, flying and walking) is controlled through keyboard inputs, but the attack and defense moves are controlled through laser gestures on a white screen. The game can be played by two people, but also works well with four if two people control movement and two perform actions with the lasers.

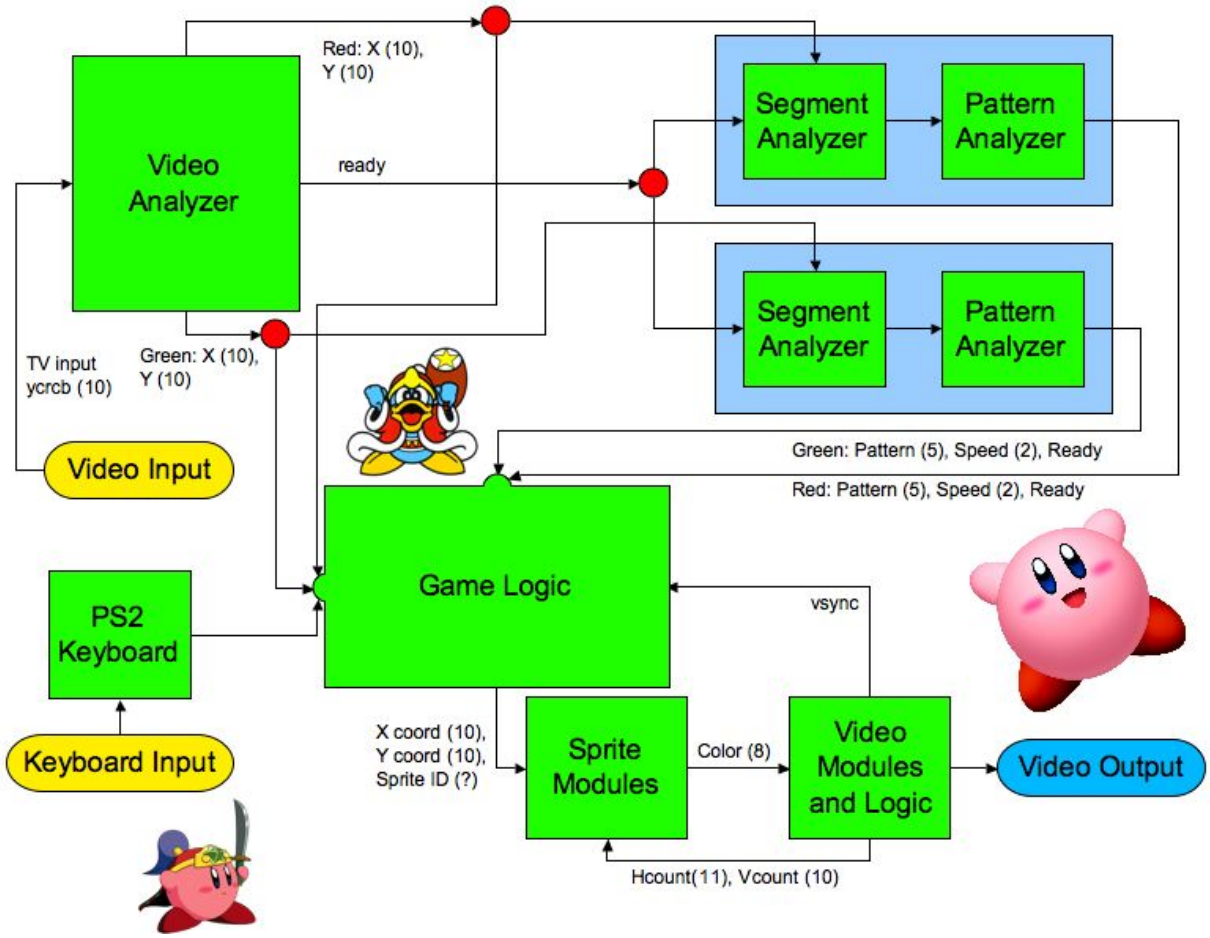

Figure 1: Overall Block Diagram

By using a green laser for one player and a red laser for another, the camera watching the screen is able to simultaneously track the position of each dot separately. The video module detects red and green laser pixels by looking at the red and green values of any pixels that have a luminosity above a specified threshold. The first 128 such dots that are found are averaged to produce the coordinates of each dot. These coordinates are fed to two gesture recognition modules which contain two sub modules. The first submodule analyzes the coordinates as they

are received from the video analyzer module and produces four bit codes indicating the current direction of motion of the dot. The second submodule contains a module for each possible gesture that can be detected. When the codes from the first submodule (called segments) are sent to the second submodule, the pattern analyzer, it gets fed into each of these gesture modules. The gesture modules have an array of registers storing the segments in the gesture they represent and as they receive segments from the first submodule, the segment analyzer, they compare them to the segment they expect to receive in their pattern. Each gesture module uses simple scoring heuristics to keep a running score of the gesture being received segment by segment from the segment analyzer. When the pattern analyzer receives a stop segment, then it knows the gesture is ended and it finds the highest scoring gesture. If the corresponding score is above a threshold it is counted as a valid gesture and sent to the game logic.

The keyboard input, pixel-based collision detection information from the sprite module, and gestures from the gesture recognition module are sent to the game logic. The game logic controls how the game works and is in charge of animating and moving all the sprites in response to the inputs it receives. The game logic has support for two melee and two ranged attacks per character as well as a defensive move. The characters can jump and fly around the level with moving platforms. Animated sequences and blood effects, including fatalities, add to the excitement of the game. The position, orientation and state of each sprite is then sent from the game logic to the sprite module, which sends back bits indicating when sprites collide and outputs the color of the current pixel to the monitor. The sprite module stores all the sprites in BRAMs and computes the color of each pixel based on the state of the sprites and the hcount and vcount sent from the video output module. Each sprite contains several different possible sub-sprites it can display, the one being displayed is controlled by input from the game logic. The orientation of the sprites is also controlled by the inputs, with simple logic being used to flip the sprite horizontally. Transparency is supported in sprites, allowing for a prettier game and better collision detection between sprites. All these modules work together to produce a truly fun to play game.

2. Video Input Module

2.1 Overview

The Video Input Module takes in data from the camera's output and outputs coordinate data that the game can use. Its primary purpose is to format the NTSC data, analyze it, and send the coordinates of the red and green laser dots to the gesture recognition modules. Its secondary purpose is to send video data to the VGA output that is displayed on the monitor. The reason why this is important is that it is useful to be able to know what the camera can see in order to center the camera on the white screen.

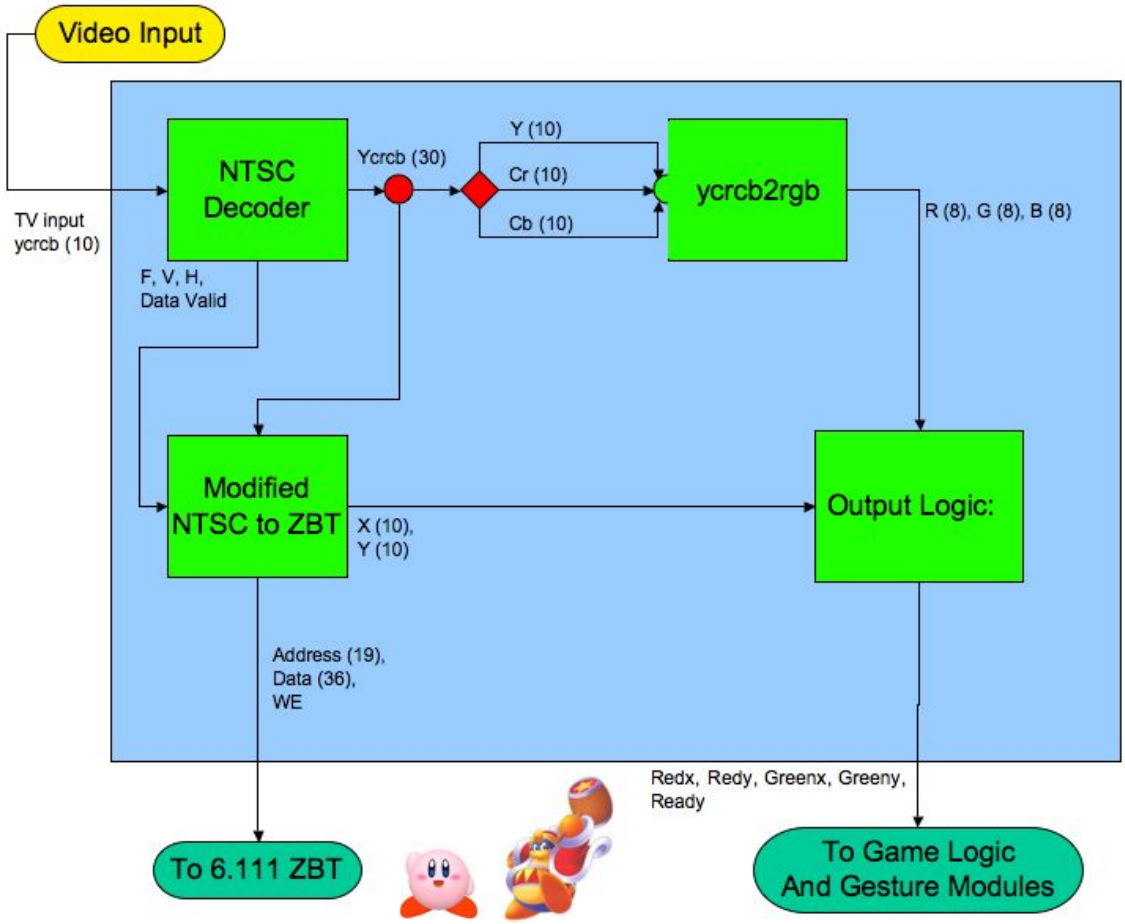

Figure 2: Video Analyzer Block Diagram

2.2 The Video Input Module's Inputs

The Video Input Module's inputs come directly from the lab kit. Its inputs are the Clock, the 27 MHz Clock, the TV Input Clock, the Reset signal, and the TV Input YCrCb Data. The Clock signal is hooked up to the 65 MHz clock of the lab kit. It is used for calculations that can be done very quickly. The output submodule (explained later) uses this clock to do its computations. The 27 MHz Clock and TV Input Clock both use the 27 MHz clock signal from the lab kit. Some sub-modules within the Video Module require a slightly slower clock to perform their calculations. The reason that there are two separate inputs hooked up to the same clock signal is that doing so provides the ability to further customize this module if necessary. The Reset signal is asserted when the lab kit is reprogrammed or when the user resets the lab kit manually. It is responsible for reinitializing the Video Input Module's registers. Finally, the TV Input YCrCb Data signal comes directly from the ADV7185 chip, which attaches to the camera.

2.3 The ADV7185 Initialization, NTSC Decoder, and YCrCb to RGB Sub-Modules

The Video Input Module contains a number of sub-modules that make it work. The first sub-module is the ADV7185 initialization module. This sub-module sends the signals required to initialize the ADV7185 video chip, thus allowing the camera to communicate with the lab kit. Its outputs are TV Input Reset B, TV Input I2C Clock, and TV Input I2C Data. They go straight through the lab kit to the ADV7185 chip. This sub-module was written by J. Castro.

The sub-module that actually takes in the NTSC serial data is the NTSC decoder, which takes the NTSC data line as input and outputs the YCrCb, F, V, H, and data valid signals. The NTSC decoder contains an FSM that tracks which part of the signal is being inputted, allowing the serial data to be processed. The YCrCb output is the color of a particular pixel in luminance and chrominance color space. The F signal represents the particular field that is being scanned when the signal is sent. The V and H signals are equivalent to vsync and hsync, which fire each time the scan reaches the bottom or edge of the screen respectively. Take note that the scan begins before the color data is valid and ends after the color data becomes invalid again. The NTSC Decoder was also written by J. Castro.

The YCrCb signal from the NTSC Decoder gets sent to a YCrCb to RGB Converter module. This module uses five constants and five multipliers to determine the 24-bit RGB signal output from the 30-bit YCrCb input. The actual operations that occur are simply a hardware recreation of the YCrCb to RGB conversion equations. Since floating point arithmetic is very costly, this sub-module uses a ten-bit number to store the constants for which the YCrCb values are to be multiplied by. This ten-bit number is simply a bit shifted binary version of the constants from the conversion equations. The ten-bit input is multiplied by this ten bit constant. It is then bit shifted back again before being sent as the 8-bit output for the red, green, and blue signals. This module was written by Xilinx.

2.4 The NTSC to ZBT Sub-Module

Meanwhile, the F, V, and H, Data Valid, and YCrCb signals are sent to a NTSC to ZBT converter sub-module. This sub-module turns these signals into outputs that the ZBT RAM can store. The outputs of this sub-module are the ZBT Address, the ZBT Data, and the ZBT Write Enable. These signals are then sent to the ZBT Manager module, which stores the value in the ZBT Data signal at the ZBT Address line of the ZBT RAM. The first part of this sub-module works by triggering a counter on the V and H signals. The counter increments on the TV Input Clock signal and keeps track of the current position of the input pixel's color data. After a little bit of pipelining, an X and Y coordinate are produced. These coordinates are associated with a YCrCb value for the given pixel data. The second part of this sub-module takes the YCrCb value and its associated X and Y coordinates, combines them, and turns them into the ZBT Data output signal. This part of the sub-module is run on the 65 MHz clock, so some synchronization is required for the X and Y data coming form the 27 MHz clock section. The ZBT address is determined by the X and Y coordinates. This works because the ZBT RAM only ever needs to hold one screen full of pixel data to display on the VGA.

The NTSC to ZBT module was modified slightly from its original state in order to output the X and Y coordinates as well. These coordinates are sent back into the Video Input Module

for processing with the output from the YCrCb to RGB converter. The X and Y coordinates need to be synchronize with the corresponding RGB data. Since the YCrCb to RGB converter has a three-stage pipeline on the 27 MHz clock, the X and Y coordinates need to be pipelined within the NTSC to ZBT module on the same clock. The unmodified version of the NTSC to ZBT module was written by I. Chuang.

2.5 The Video Input Module's Outputs

The actual output signals of the Video Input Module are the ZBT Address, the ZBT Data, the ZBT Write Enable, the X and Y coordinates for the red and green laser dots, and a Ready signal. The ZBT Address, Data, and Write Enable are sent to the ZBT memory, which will be discussed later. These signals are used for sending the processed video data to a ZBT RAM, which will then be sent to the VGA output. The X and Y coordinates for the red and green laser dots are determined by averaging and are sent to the Gesture Recognition module. The Ready signal is asserted when the averaging is complete in order to tell the Gesture Recognition module that it is time to record new coordinate values. This happens once on every camera screen refresh.

2.6 The Output Logic

In order to actually fulfill its role, the Video Input Module has an output sub-module that processes the signals from the Video Input Module's other sub-modules and compiles them into the expected output format. This sub-module does not exist in a module of its own, but is isolated to an always block in such a way that it is independent of the rest of the module. The function of this logic is to process the X, Y, and YCrCb signals that are outputted by various other sub-modules and turn it into the outputs of the Video Input Module. In order to find the X and Y coordinates of each dot color, the YCrCb signal for a given X and Y coordinate first needs to be evaluated to see whether it corresponds to the red laser, the green laser, or nothing. If the data is determined to correspond to a laser dot, the X and Y coordinates are stored. Every time the camera screen refreshes (in other words, on a cycle that the X and Y coordinates are equal to a given constant value), the stored X and Y coordinates are averaged and their average is outputted as the red and green X and Y dot coordinates. In addition, the ready signal is asserted to tell the Gesture Recognition module to input the X and Y data.

The actual mechanics of the output sub-module are slightly more complicated. In order to check whether the particular RGB value is red or green, the YCrCb value corresponding to the given RGB value is first checked. The highest ten bits of this signal represent the luminance, or Y. The luminance is higher for brighter input and lower for dim input. Since the laser dot on the screen is exceptionally bright input, this check is an easy way to find whether a laser dot is shining at a particular point. The ten-bit luminance threshold was set as high as hex 300, in order to block ambient light from triggering a result in the sub-module. However, it was not set to an extremely high value such as 3F0 because the light coming from the laser pointer has a distribution of intensity and at times, it will randomly fall short of the threshold. Even with a threshold of 300, there is a small but finite chance that the intensity detected at a given laser saturated pixel will be less than the threshold.

Once a pixel has been determined to be a laser dot, the actual color of the pixel must be determined. The simplest way of doing this is to compare the red and green values from the RGB signal (bits 16 to 23 and 8 to 15 respectively). The problem with this method is that the laser is generally so bright that it saturates the camera. When tested, the red and green values of the pixel data would both be hex FF, which is the maximum value. Though this could be passed off as white light, what made it tricky is that the RGB color of the pixel was sometimes incorrect. For example, there were times that a red dot would show up as a green dot because its red value was hex FE and its green value was hex FF. Such an error happened because the light coming reflecting off the screen was so bright that it saturated the camera and random disturbances would produce incorrect color values. One idea for compensating for these disturbances was to use a difference threshold, in which if the difference between the red and green values were not greater than the threshold value, it would be passed off as white light. However, an even better method was derived to correct these errors and it will be explained later.

In order to get the color detection to work, a lot of calibration had to be done. Specifically, a red and green laser pointer of the same power and intensity were chosen and dimmed such that they did not saturate the camera. However, even though they are both labeled as having the same intensity, the two lasers actually appear very different in intensity to the camera. This could either be because of the manufacturing differences in the laser pointers, or it could be because of the camera's transfer function favoring one color over the other. Nonetheless, the laser pointers were dimmed to the point that they no longer saturated the camera by attaching layers of tape (containing Tynan's dead skin) to the front of them. Since the tape diffuses the beam, the player is required to stand sufficiently far back from the screen such that the beam doesn't get too focused.

In its first state, the output logic did not use an averager, but rather simply took the first point it saw and sent that point as output. However, the laser dot appears much bigger to the camera than it does to the human eye, and therefore incorrect coordinate data ended up being outputted. Such a method also produced a lot of noisy data since the light coming from the laser scattered over the screen in interesting ways. Therefore, finding the center of mass by using an averager was absolutely necessary. There is an averager for each of the red and green x and y coordinates. These averagers use the standard discrete averaging equation, which is the sum of the values divided by the number of values.

In order to find the sum of the values for the red and green X and Y coordinates, a cumulative sum is taken. It is updated every time a green or red laser pixel is detected. In addition, a counter is updated indicating how many red or green laser pixels have been detected so far. Once 128 pixels of a color have been detected, the cumulative sum and counter will no longer increase their values. However, if a particular color counter did not reach 32, then the corresponding sum will be discarded and it will be assumed that the laser of that particular color is not pointing at the screen. This feature greatly improved the performance of the video module. Prior to its implementation, the averager would be used and its quotient would be outputted should a single pixel of the required luminance be either red or green. However, that method did not work well since each laser had a small chance of randomly appearing to the camera as the other color.

The division portion of averaging equation is implemented by feeding the sum and counter into the dividend and divisor inputs of a divider. The dividers are implemented by using

Coregen. They have an 18-bit dividend input, 7-bit divisor input, 18-bit quotient output, 7-bit remainder output, and a 1-bit ready for data signal. The ready for data signal and the remainder are both unused. The way that the module knows when to check the quotient value is that it times each divider with a separate counter. When 32 clock cycles have passed, the module checks the value on the quotient line. Technically only about 20 clock cycles are needed, but since the divider only does one divide every camera scan refresh, it doesn't really matter.

The dividers are connected to some logic that will determine whether or not the necessary dividers are finished. For example, if there is only a green dot, then only the green dividers have to finish. If the required dividers are finished computing, the logic triggers a pseudo-FSM. This pseudo-FSM is more like a script than an FSM because it is linear and functions only to assert the output signals when they need to be asserted. The red and green X and Y coordinate signals connect to the quotients of the dividers. The FSM also manages when the ready signal should be asserted and turns it off after a couple of clock cycles.

2.7 Testing and Debugging of the Video Input Module

The Video Input Module was initially tested using the hex display. Various outputs on the hex display included the green and red X and Y coordinates, various flags, and up to four 8-bit diagnostic outputs of the Video Input Module. These diagnostic outputs were set to various values such as the red and green color value at the point (100, 100). In addition, the color data for the pixel corresponding to all X = 100 or Y = 100 was set to black. This way, while the Video Input Module was set to camera to VGA mode, it would be much easier to point the laser at a specific location to measure its color values.

Finally, the testing module was improved such that it would output a red or green circle on the screen to where it thought the laser was pointing. This was very useful because on its first run, the circles it displayed jumped all over the place. The noisy data led to the development of the averagers, and such a modification drastically improved the module's performance.

3. Gesture Recognition Module

3.1 Gesture Recognition Module Overview

The gesture recognition module processes the coordinates of the laser dots given by the video analyzer module and determines what gesture is being done, which is then sent to the game logic. The gesture recognition module is composed of two sub-modules, a segment analyzer and a pattern analyzer. The segment analyzer processes coordinate data as it is received and looks for net movement over several points and encodes this as segments which are sent to the pattern analyzer The pattern analyzer is also streaming, with a module representing each pattern. These compare incoming segments to expected segments and keep a running score of the current gesture. When a special stop segment is sent by the segment analyzer (triggered by the laser not moving or leaving the field of view), the pattern analyzer selects the pattern with the highest score and, if it is higher than the threshold for a match, sends the id of the pattern to the game

logic module. There is one gesture recognition module, each containing its own segment analyzer and pattern analyzer modules, for each laser dot.

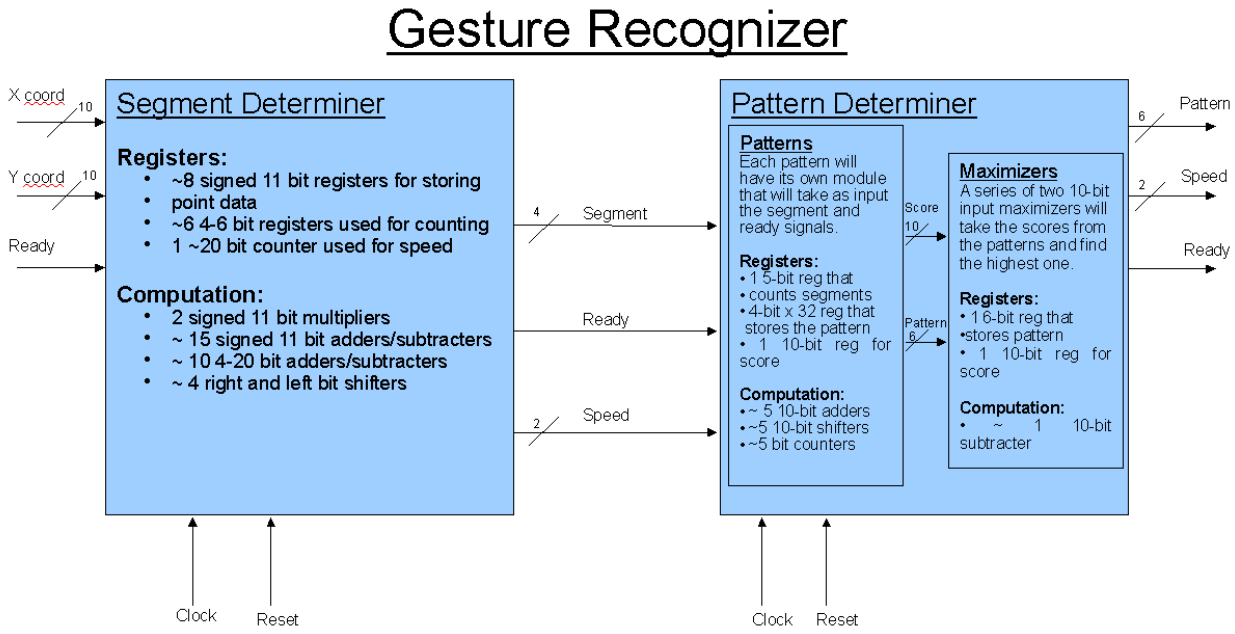

Figure 3: Gesture Recognition Module

3.2 Segment Analyzer

The segment analyzer takes as input a ready signal and coordinate pairs from the video analyzer module and outputs a ready signal, a segment and a speed to the pattern analyzer module. The segment analyzer module runs on the 27 mhz clock and takes as input the coordinates and ready signal from the video analyzer module. To accommodate the fact that the video analyzer module holds the ready signal high for only once clock cycle on the 65mhz clock it uses, the gesture recognition module also has a few registers used to create a new ready signal that is held high for one of the 27mhz clock cycles. The module is also hooked in to the universal reset signal which when sent resets all the registers to their default values and causes gesture recognition to start over. The segment analyzer takes in coordinates from the video analyzer module and outputs a series of segments to the pattern analyzer module. There are ten possible segments, eight represent the four cardinal directions and four main diagonals, one represents the start of a gesture and the last represents the end of the gesture. The segments are determined based on the net movement in the x and y directions of several points in a row. Segments are only sent to the pattern analyzer if they are detected multiple times in a row and the same segment is never sent twice in a row. This way the segment analyzer sends to the pattern analyzer the directional components of the gesture in the order they occur.

The process for converting the sequence of coordinates into segments is fairly simple. The module's behavior is modified by four main parameters: num_points, which determines

how many points of movement (points that are not too close together) are used to compute each segment; num_segs, which determines how many times a segment must be detected in a row before it is considered valid and sent to the pattern analyzer module; move_cutoff, which is the number of successive coordinates of non-movement that must be detected before the gesture is considered ended; and move_dist, which is the square of the distance above which a coordinate is considered to be a moved point. The module waits until it receives the first ready signal and valid coordinate pair (a y-value of 1023 in a coordinate pair indicates the laser is out of view and ends any currently active gesture). At this point the module looks at each set of incoming coordinates until it gets a pair that are above a minimum cut-off distance from the first set (given by move_dist). If none of the next move_cutoff inputs are outside of move_dist from the first input then the gesture is considered to have ended and a stop segment is sent. Once an input demonstrating sufficient movement is found, the process starts over with this new point as the reference and a counter increments keeping track of the number of so-called "movement points." Also upon detecting the first movement point, the module considers a gesture to have begun and sends a start segment to the pattern analyzer module.

When the number of movement points received reaches num_points, then the segment analyzer compares the coordinates of the first and last movement points to find the net x and y distance traversed. It compares the x movement to the y movement to determine which of the eight possible movement segments best matches the data. If the magnitude of the x-movement is more than twice the magnitude of the y movement, the segment is either horizontal left or right; if the magnitude of the y-movement is more than twice that of the x the segment is considered vertical, otherwise it is considered diagonal and the signs of the movements determine the direction. When the segment has been determined the segment analyzer compares the segment to the previous segment, if they are the same then a register called state_count is incremented, otherwise it is reset to zero. When enough of the same segment have been seen in a row (i.e. when state_count is equal to num_segs), then the segment is considered valid and sent to the pattern analyzer module. A segment is only sent to the pattern analyzer module when state_count is equal to num_segs, this way if the same segment continues to be detected, it is not sent more than once to the pattern analyzer. This process of counting up movement points to determine segments and counting up segments to send to the pattern analyzer continues until either movement stops, or the laser leaves the field of view, at which point a stop segment is sent to the pattern analyzer. The segment analyzer then starts over waiting for sufficient movement to indicate the start of the next gesture. Each time a new coordinate pair is received (regardless of whether or not it has moved), a counter indicating the total number of coordinates in the gesture is incremented. Since new coordinates are received every frame, we use this to determine the speed of the gesture as a whole by comparing the count to certain cutoffs. This speed is fed through the pattern analyzer to the game logic module where it can be used to affect actions depending on the corresponding gesture.

Because analyzing each set of coordinates requires a few multiplications (to determine distances), many comparisons and a few other computations, and because I was using the 27mhz clock, I made a simple FSM to spread the computation out over several clock cycles. I chose to do this rather than normal pipelining because it made things a little simpler and because I was only getting new coordinates once per frame, which was many more clock cycles than the computation needed. The FSM has eight states, each one containing a different chunk of the

computation necessary to analyze the incoming data (see Figure 4). The FSM sits idly in state 0, not performing any calculations until a ready signal and a new set of coordinates are received from the video analyzer module. A case statement on the state register compute_stage is run every clock cycle to carry out computation for the current state and control transitions to the next state based on computations in previous states (see verilog in Appendix VII).

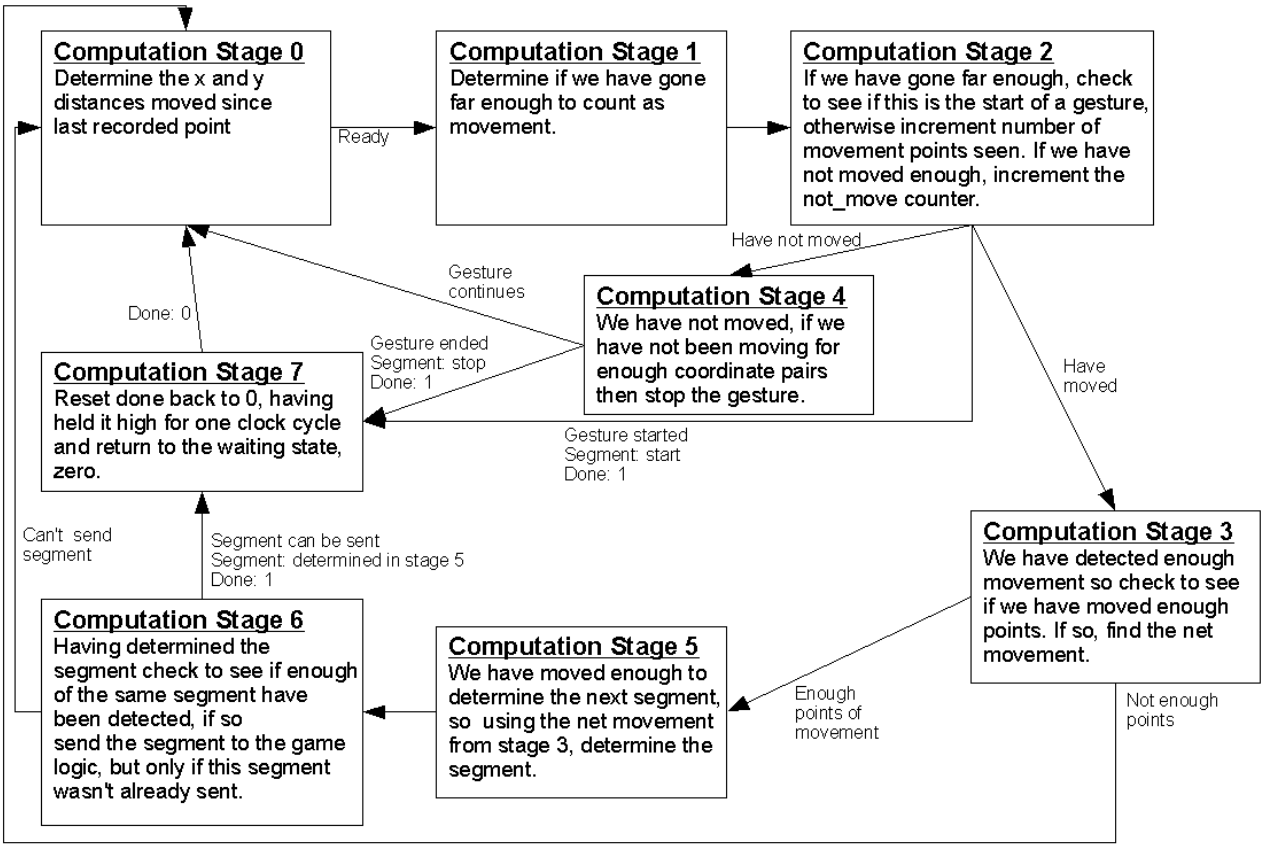

Figure 4: Segment Analyzer FSM

3.3 Pattern Analyzer

The pattern analyzer takes as input a ready signal, a speed and a segment from the segment analyzer and outputs a gesture id, a ready signal and a speed to the game logic module. The module also uses the universal reset signal to clear registers and the 27mhz clock. Segments are encoded as four bit numbers, 0010 and 0100 are used for stop and start respectively, and the codes for the eight directions were chosen such that the hamming distance between codes indicates the similarity of the segments (see figure 5). This property is used by the scoring heuristic within each pattern module to account for misinterpreted segment errors. The pattern module also does not store the incoming data but rather processes it as it is received. Each possible gesture is represented by its own pattern module that takes as input the segment and ready signal from the segment analyzer and outputs the gesture id for the module and a running score indicating how much the current gesture matches that particular module. The scores and

ids from each of the patterns are fed to a giant maximizer which outputs the highest scoring pattern id and the corresponding score. When a stop segment is received the pattern analyzer waits 2 clock cycles to guarantee all the scoring computations are finished then checks to see if the score of the highest scoring pattern is greater than the threshold indicating a valid pattern. If it is, the pattern id and ready signal are sent to the game logic module, otherwise nothing is sent.

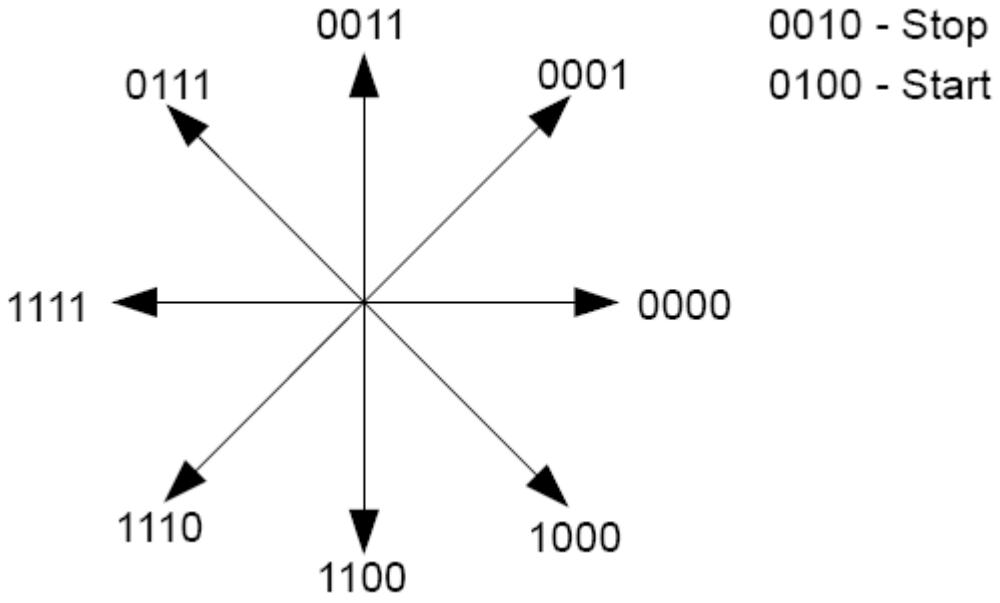

Figure 5: Segment encoding scheme

The scoring system contained within each pattern module is not overly complex. Each module has parameters indicating the length (in segments) of the gesture it represents as well as the segments that make up the gesture. An internal register keeps track of the number of segments received so far, indicating where in the pattern the gesture currently is. When a new segment is received, a few checks are made first. If the segment is a stop segment then we compare the number of segments received to the expected length of the gesture, and if the inputed gesture ended earlier or later than expected, the difference in length is subtracted from the score, otherwise ten is added to the score because the input gesture had the expected length. If a stop segment is not received, then we compare the segment count to the length anyway to see if we have gone longer than expected, if so we again subtract the difference in length.

If neither of these conditions hold then we compare the incoming segment to the expected segment (the segment of the stored pattern that corresponds to the segment count) and add a multiple 2 minus the hamming distance to the score. If the segment count is not the first or last segment of the pattern than the newly received segment is compared to the current, next and previous segments that the pattern module is expecting (based on the count of segments received). By including comparisons to next and previous segments we are able to accommodate inaccurate gestures resulting in dropped or added segments, provided we are not trying to distinguish between gestures that are very similar. The hamming distance between the input segment and each of the three segments current, previous and next is subtracted from two to give a temporary score for each. This comparison score results in -2 for segments that are opposite, -1

for segments that are 135 degrees apart, 0 for segments that are 90 degrees apart, 1 for segments that are 45 degrees apart and 2 for segments that are the same. The best match of the three segments (previous, current and next) to the inputed segment is used for scoring. If the best match was with the current segment then the hamming score is multiplied by eight and added to the score, otherwise it is multiplied by two and added in. The verilog in appendix VII describes this more clearly.

A combinational logic maximizer composed of a bunch of two-input maximizers takes the scores and id from all the pattern modules and outputs the highest score and corresponding id. Whenever a start segment is received by the pattern analyzer a reset signal is sent to all the pattern modules to zero the score and segment counting registers. When a stop segment is received the highest scoring pattern id is sent to the game logic module, provided its score is greater than the valid gesture cutoff score. In addition the speed of the gesture is fed through from the segment analyzer module.

3.4 Testing and Debugging

A lot of time was spent designing and planning out the gesture recognition module before it was actually implemented on the labkit. Before the current system was decided on, we experimented with a variety of similar systems in software, including one which used a large complex state machine. The current method was chosen because it was effective, but still fairly simple and did not require much computation. Originally we had tried to implement curved segments as well, but found it too hard to determine curvature using only a small number of points. Once the software version was working well it was used to generate test data to be used as input for a Modelsim implementation. After more tweaking and modifications to the Modelsim version the module was implemented on the labkit with the early version of the video analyzer module the existed at the time. Segments, gestures and scores were all output to the hex display. A few bugs had to be removed that did not come up in Modelsim because the Modelsim tests did not include multiple gestures in a row. Also the heuristics and segment analyzer parameters had to be tweaked to better suit the quality and rate of data received from the video module. When the entire game was hooked up and the final gestures for actions chosen more tweaking was performed to optimize the detection for the specific combination of gestures chosen, and to take advantage of the improved video module.

4. Keyboard Module

4.1 Overview of the Keyboard Module

The Keyboard Module's purpose is to input the keyboard PS/2 data, process it, and output signals for whether certain keys are depressed or not. The function of the Keyboard Module in the game is for moving the characters around. The Keyboard Module was programmed by Chris Terman and I. Chuang. It initially served to display ASCII characters on the hex display. It has been very slightly modified with the given key down outputs but in general, it preserves its original structure, function, inputs, and outputs.

The Keyboard Module's inputs consist of the 27MHz Clock, a Reset signal, the 65MHz

Clock, and the keyboard's PS/2 Data. The two clock signals are used for synchronous calculation and communication between the module and its sub-module. They are especially important in order for the key down outputs to function correctly. And the PS/2 Data comes through the lab kit and straight from the keyboard. The outputs to the Keyboard Module consist of the ASCII Code output, and ASCII Ready signal. The ASCII Code output is simply the ASCII representation of what key was most recently pressed or released. The ASCII Ready signal is asserted when all of the ASCII Code output is valid. This signal is important in avoiding glitchy data.

4.2 The Inner Mechanics of the Keyboard Module

The Keyboard Module contains a sub-module for parsing the input data that it receives from the PS/2 port. This sub-module takes in the PS/2 clock and data and outputs the equivalent scan code for the current PS/2 signal. The way it does so is by reading in the data and implementing a counter that counts with the clock. The data's 10 bits should have the first bit as 0, the next 8 bits as the scan code, the next bit as a parity bit, and the last bit as a 1. This sub-module checks the bits and synchronizes itself with the clock until it is outputting the correct scan code.

The scan code is then read by the Keyboard Module and through a case statement, is turned into ASCII code. However, my module does not even use the ASCII output. One useful feature of the original Keyboard module is that it records the last key pressed. This feature was very useful in determining when a key was released. Whenever a key is pressed, the scan code for that key goes through the PS/2 port. If that key is released, then the keyboard first passes a break code and then passes the scan code for the released key. So when checking the current scan code, it is also useful to know the last scan code to be able to tell whether a key has been pressed or released. With all of that on mind, it is easy to know when a particular key (such as the arrow keys) is being held down.

5. Game Logic Module

5.1 Game Logic Module Overview

The game logic module is the skeleton of Kirby Laser Attack that all the other organs attach to. It determine all aspects of how the game actually plays. The game itself is a multiplayer combat game featuring Kirby and his arch nemesis King Dedede. The game is in theory two-player, but works even better with four people because the movement (walking, jumping, flying etc.) of the two characters is controlled by two keyboards, and the attack and defense actions are performed by gestures on a screen a short distance away. The game is sprite-based and has several melee and ranged attacks, as well as a lot of animations and violence effects. The game logic module takes in the player input and causes the characters to move, attack and defend appropriately. It also controls the movement and animation of all other dynamic sprites in the game, including the moving platforms, the blood effects and projectiles. Furthermore, it handles dealing damage, gravity, the bounding box-based collisions for the

platforms and floor, and most importantly the fatality animations.

The game logic module has a lot of inputs: the eight keyboard buttons used to control the platforming of the two characters; the ready, pattern and pattern speed signals from both gesture recognition modules; the pixel-based collision data from the sprite module; an enable fatality signal hooked to a switch on the labkit; a pause signal controlled by the keyboard; the 27mhz clock; the universal reset signal; and vsync from the video output module. The output from the module is all the information used to control the position and appearance of all the sprites: the positions of the sprites; the left-right orientation of the sprites; a single bit for each sprite controlling whether it should be displayed or not; and the id for each sprite indicating which version should be displayed (e.g. which frame of the kirby walking animation to display). Lastly the module outputs the current health of Kirby and King Dedede, which is used by the video/sprite module to display the health meters.

The module uses the vsync signal to trigger calculation of the next frame so that sprite positions are updated while video isn't being actively output to the monitor, allowing us to avoid graphical glitches. The 27mhz clock is used to sync with the signals coming from the gesture analyzer and hold them in registers for a frame so that they can be used in the logic triggered off of vsync. If the game is still active (i.e. both characters have positive health and the pause input is 0) then the standard frame is computed. Every frame counters controlling all aspects of the game, from the length of attacks, to the flow of animations are decremented each frame. If any keyboard input is received the character movement state is changed, if gestures are received actions are started, and if any collisions are detected then the resulting animations and other effects are started. Also, each frame, any current animations are continued for the characters, the weapons and the blood. Any actions whose timers have expired are ended and the position of each sprite is updated based on that sprite's current velocity. If pause is enabled then no actions are performed, and if one of the character's health is below zero then the game ends and any final death animations are played. If the fatality switch is flipped then a fatality animation is performed whenever someone dies.

The game uses many parameterized constants to control all aspects of the game, including starting positions, the damage, length and hit-back of attacks, the velocities of projectiles, the length of sequences in sprite animations, weapon offsets from characters, character movement speeds, and much more. This allows us to easily and independently vary the behavior of both Kirby and King Dedede. See Appendix I for the verilog.

5.2 The Level and Movement

The level consists of a floor with health meters embedded in it and six platforms, two of which move up and down and two of which move left to right. The characters can walk, jump and float around. Jumping is achieved by pressing up when standing on the floor or a platform. Once in the air (through falling or jumping) a character and can jump again to float higher, up to three times before they must land. While in the air a character moves side-to-side more slowly. A character may fall more slowly by floating (jumping while in the air), but they can fall more quickly again by pressing down. Gravity is implemented by incrementing a counter frame, and when the counter reaches a certain value (controlled by the parameter g_frames) it is reset and the parameter gravity is subtracted from every sprite affected by gravity (unless a character is

floating, in which case half the parameter is subtracted).

Collision with the floor and platforms is done using bounding boxes that allow a character to move up through a platform, but prevent them from falling back through. Each character has several state registers to control and keep track of movement. One is for vertical movement which can take on the values of falling, floating and sitting still. This register, combined with y velocity affects the character sprite while in the air and also changes how they are affected by gravity. The value of this register is controlled by jumping, falling (pressing down while floating) and collisions with platforms and the floor. Another state register is for horizontal movement and can take on the values of walking left, walking right or sitting still. This is used to determine the left-right orientation of the character and weapon sprites as well as the direction of attacks. It is based only on the current state of the left and right movement keys of the character's keyboard. A third state register keeps track of whether an attack, a defense or no action is being performed, which is also used to control sprite animation as well as for processing collisions.

If the game logic module receives keyboard input then it changes the corresponding character's movement state registers as necessary. If left is pressed then the horizontal movement state changes to walking left, the character and melee weapon are oriented towards the left and the melee weapon is positioned on the left side. If the right button is pressed then symmetric changes happen. If a character is in a walking state and they are not performing any actions and on the ground then the walking animation is displayed. If the up key changes from not pressed to pressed then a jump is performed (or a float if the character is already in the air), unless the character has already jumped the maximum times before touching the ground again. A down input transitions a character in the air from floating to falling, allowing them to fall a little faster.

5.3 Performing Actions and Handling Collisions

Each character has two melee attacks (one forward and one upward), two ranged attacks (one straight and one lobbed) and one defensive action. Whenever a ready signal is received from either player's gesture analyzer, the ready signal, the incoming pattern and the incoming speed are all stored in registers to be processed when computing the next frame. Each gesture id can correspond to multiple gestures, allowing us to have similar gestures perform the same action, giving the players a little more flexibility. When a valid gesture is received by the game logic, first we check if it is an attack or defense gesture. The action corresponding to the gesture is only performed if the character is not currently being hit by another attack (which temporarily stuns and knocks back an opponent), and not in the process of performing an attack or defense.

When an action is performed, several things happen. The timer that keeps track of if the character is doing something is set to the length of the animation for the action. Any sprites involved in the action (e.g. projectiles or melee weapons) are positioned, oriented, displayed, given initial velocities and set on the first frame of their animation. Also, the state register indicating a characters action status is set to either attack or defend, and if an attack, the particular attack being performed is set in another state register (this allows us to determine which melee attack a character was hit by even though we use the same weapon sprite for both so they both trigger the same collision bit). If a projectile attack is performed the velocity of the projectile is a base velocity plus a bonus velocity based on the speed of the gesture. There are also cool-down timers for each ranged attack that prevent a character from using the attack to

quickly, so the ranged attack will not be performed if the corresponding cool down timer has not yet reached zero.

In addition to the bounding box collisions used for the platforms and floors in the level, there are pixel-based sprite collisions sent to the game logic from the sprite module. There are seven such collisions we use in the game logic. The two collisions for when Kirby is colliding with Dedede's melee weapon and vice versa are used to determine when a character is hit by a melee attack. Similarly there are four collisions between character's and the enemies two projectiles to indicate when a character is hit by a ranged attack. Lastly there is a collision for when Dedede and Kirby touch, which is only used to trigger a fatality if one or both characters has below ten health. When hit by a melee attack a character's hit counter is set to the hit time of the attack (which is determined using the attack state of the other character). This prevents a character from performing any actions or moving for a few frames. Also the hit character is knocked with initial velocity based on the parameters of the attack. Furthermore, damage is dealt to the hit chracter. Lastly a blood spurt animation is played at the character's location and globs of blood erupt from the location and fall off the screen. If the hit character is defending at the time then they take less damage, there hit timer is not set, they are not knocked back and there are fewer globs of blood. If a character is hit by a ranged attack the same things are done, except the velocity of the projectile determines how the character is hit and the velocity of the blood globs, also the projectile disappears. If a character is defending then they effectively dodge the projectile, allowing it to pass through them without getting hit. The character-character collision is only used if one or both characters have less than ten health. If so then the game ends with a gruesome death animation known as a fatality for the character with low health, or an eruption of blood if both characters have low health.

5.4 Animations, Blood and Fatalities

The game logic module also determines which version of each sprite is being displayed each frame. Sprites have multiple possible pictures to allow for simple animations by rapidly changing which picture the sprite is displaying. This is accomplished through a timer for each sprite that needs to be animated. Whenever an animation sequence is started, this timer is set to the length of the first frame of that sequence. It is decrement every frame and when it reaches zero the current frame is used to determine the next frame of the animation and the timer is updated with the length of the next segment of animation. The blood effects, projectiles, characters and melee weapons all have animations that are controlled this way. In the case of the characters, the state of the character as well as animation timers determine the sprite. If the character's hit timer is not zero (meaning he recently got hit by an attack) then the hurt sprite is displayed. If an action is being performed then the corresponding sprite or animation is displayed. If the character is in the air then the falling sprite is displayed if the y velocity is down the screen and the jumping sprite is shown if the y velocity is up the screen. If the character is not in the air but is moving right or left then the walking animation is displayed, otherwise the sitting still animation is displayed. Projectiles have two sprites that just alternate until the projectile's life counter reaches zero and it disappears. The blood animations are a few quick frames positioned where the character got hurt that then disappear. The melee animations are a

sequence of three frames that are displayed in conjunction with two frames of character animation.

There are eight chunks of blood that are affected by gravity and squirt from a character when they get hit. Normally only two chunks (for being hit by ranged or being meleed while defending) or four chunks (for normal melee hits) are displayed per character at a time. The initial velocity of the blood is determined by parameters and the orientation of the attack (andspeed of the projectile) and it falls in a parabola until it leaves the screen and is disabled. There are not timers on the blood, so if another attack lands while the blood is still falling it simply starts over from the new attack location. All eight chunks of blood are used when a fatality is triggered. A fatality is basically a gruesome, hard coded animation that, as mentioned previously, is triggered when the characters touch and one or more as very low health. The fatality uses the same counter based animation technique and simply moves and changes the sprites as necessary whenever the counter reaches zero. An extra large blood animation is also used when the losing character's corpse explodes in a shower of blood and flesh. All told, there is about as much verilog used to code up the blood and gore as there is in the gesture recognition modules. So either we had efficient gesture recognition, inefficient blood and gore, or simply misplaced priorities.

6. Sprite Module

6.1 Overview of the Sprite Module

The Sprite Module is the least necessary and most awesome module. This module stores all of the game's graphics and determines the color of every pixel. It stores each sprite in block memory (BRAM) and contains a number of sub-modules to simplify the logic of choosing the sprites. These sub-modules are a set of mutually exclusive sprites that are all of relatively the same size such that they can all be calculated and treated as a single sprite. In other words, no two sprites in an instance of a sprite sub-module will ever be displayed at the same time. The Sprite Module also contains a sprite-by-sprite priority list of which sprite to display should there be an overlap of sprites. In addition, this module is also responsible for the game's collisions, all of which are pixel-based collisions except the ground and platforms.

The Sprite Module's inputs, with a few exceptions, are the coordinates and ID for every sprite sub-module. The coordinates represent where the top left corner of the sprite should be placed and the ID represents which sprite the corresponding sprite sub-module should use. The highest order bit of the ID is usually used to determine which direction the sprite should be drawn in. Sometimes there are exceptions to this, as that the platform sub-modules, for example, use the ID to color the sprite in different ways. In addition to these inputs are the standard Clock and Reset inputs. The Clock input is used by the BRAMs to synchronize their value look-ups. Therefore, the sprite sub-modules also need to use the Clock signal in order to output the correct color data. The Reset input helps initialize the registers of the sprite sub-modules. The two final inputs are Hcount and Vcount, which are two signals that basically represent the X and Y coordinates of the VGA's current color output value. The Sprite Module's outputs are the 8-bit

color value for a given pixel specified by Hcount and Vcount and the eight 1-bit collision signals. These collision signals are on when two specific sprites have pixels that overlap and off otherwise.

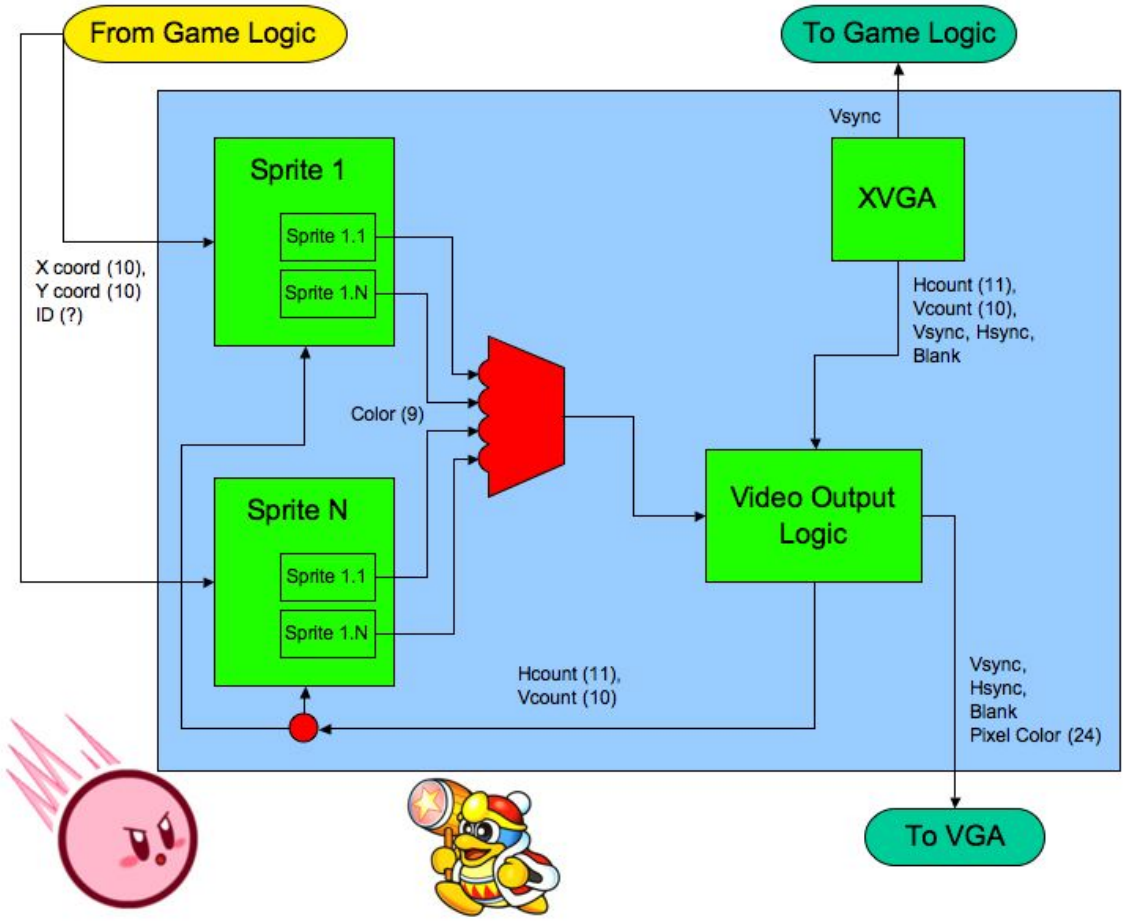

Figure 6: Sprite module block diagram

6.2 The Sprite Sub-Modules

The sprite sub-modules' purpose is to simplify the logic and handling of the sprites. Each sprite sub-module contains BRAMs for mutually exclusive sprites. For example, the Dedede sprite sub-module contains all of King Dedede's animations, but there will only ever be one King Dedede on the screen at a time. The general method for specifying a sprite from the sprite sub-module is to specify its X and Y coordinates and its ID. For most sprites, the ID determines which of the BRAMs within the sprite sub-module to read from (though there are exceptions).

Whenever the X, Y, Hcount, or Vcount signals to the sprite sub-modules change, a logic block is initialized to determine whether or not Hcount and Vcount are still within the sprite's boundaries. The sprite's boundaries are specified by X, X + WIDTH, Y, and Y + HEIGHT, where WIDTH and HEIGHT are the sprite's width and height respectively. If the logic

determines Hcount and Vcount to be within the range of the sprite's boundaries, it will then check the sprite ID's highest order bit for orientation. After that, the sprite sub-module specifies the address line of the BRAMs that should be read from based on the sprite's X and Y coordinates relative to the Hcount and Vcount signals. If the sprite should be read forward, the address is set to (Vcount - Y) * WIDTH + (Hcount - X), and if it should be read backwards, the address is (Vcount - Y + 1) * WIDTH - (Hcount - X). In addition, this logic block asserts a Ready signal to signify that the data outputs of BRAMs will soon be ready.

The rest of the logic is clocked and therefore sequential. If the hardware detects the Ready register to be on, it then sets a 9-bit temporary register called Precolor to the output value of the particular BRAM specified by the sprite ID. The reason that Precolor is 9 bits is that the highest order bit represents whether or not the pixel should be invisible. If the ready signal is not asserted, then the pixel will be invisible since the Hcount and Vcount signals are not within the sprite's boundaries. If the specified ID is not a valid value, then Precolor will be invisible. This allows the game logic to easily set the sprite to be invisible if necessary.

One final role of Precolor is to format the sprite. In order for the sprite not to have an ugly black box behind it, the sprite must be formatted such that the parts of the sprite that should not be part of the image are removed. The actual data for the image that was loaded into the BRAM specifies invisible pixels with the color green, or 8'b00011100 in 8-bit color. Therefore, whenever Precolor is equal to 8'b00011100, the final Color output of the sprite sub-module will be set to invisible.

There are several exceptions to the sprite sub-module template described above. The first exception is the sword and hammer sub-modules. The first three sprites in this sub-module have different heights and widths than the last three. However, the height of the first three is the width of the last three and the width of the last three is the height of the first three. Therefore, after a check of the ID to determine which sprite is being displayed, displaying the correct sprite is simply a matter of switching the HEIGHT and WIDTH parameters for the last three sprites. The second exception is the blood chunk and platform sub-modules. These sub-modules contain the capability to select the color of the sprite within the bits of the sprite ID. When the sprite was created as an image, it was colored in a certain way. In particular, the original platform was colored pure red (8'b11100000). The sprite sub-module can then check the color of the BRAM output, and if it is pure red, then the sprite sub-module will recolor it depending on the sprite ID. The third exception is the health meters. These sprites do not have X or Y inputs since they are not intended to move. Instead of an ID, they have a health input that represents Kirby or King Dedede's health. The health meters also display red or blue depending on how high the character's health is. These sub-modules do that by comparing whether Y + HEIGHT - Vcount is less than the health input, and recoloring the pixel red or blue if so.

6.3 Priority and Collisions

Once all of the sprite modules know what color they want a pixel to be, the Sprite Module must determine the color of the pixel to be outputted to the VGA. It does this based on invisibility and priority. If a sprite module says that a pixel should be invisible, then its output will be ignored for that pixel. However, an overlap occurs if two or more sprite sub-modules believe that a pixel should be some particular color. In the event of an overlap, a set of priority

logic determines which sprite sub-module's pixel color will be outputted to the screen. For example, since Kirby is the main character, Kirby's pixel color will have higher priority than Dedede's pixel color, and therefore if there is an overlap of Kirby and King Dedede, then Kirby's pixel color will be the one displayed. However, blood has the highest priority in this game.

Eight of the sprite module's outputs are collision signals. A collision can happen between particular sprite sub-modules, such as Kirby and the Hammer. When the Kirby sub-module and the Hammer sub-module both believe that the current pixel is not invisible, then a collision will occur. It is nice for the collision signal to remain valid for more than a single Hcount value, so additional sequential logic exists such that this signal is held for several screen refreshes. The module does this by firing a Vcountflag every time Vcount is zero, and updating a signal counter for each collision signal. The counter allows the collision signal to remain for several screen refreshes, and resets if another collision has been detected before the counter has overflowed and gone back to zero.

6.4 Creating the BRAMs

All of the sprites in the game were ripped from various games and posted on the Internet. These images were downloaded and saved as bitmaps. Bitmaps were chosen because they are uncompressed, and therefore there isn't any of the annoying interpolation that occurs when an image gets compressed. It is important that the colors are precise because of all the recoloring that occurs in the sprite sub-modules. Once downloaded, these images were processed using GIMP (like Photoshop, except it's free). The processing that had to be done includes basic scaling and rotation, along with sharpening, and even manually coloring pixels to tune the images up to shape. And of course, the parts of the sprite that are supposed to be invisible were colored pure green.

Once the images were fully processed, they were converted into .coe files using a Matlab M-File. When the imread() command is used, Matlab can read in an image and generate 8-bit matrix data for the image's red, green, and blue components. In order to convert this 24-bit color data into 8-bit color, the top three bits of the red, top three bits of the green, and top two bits of the blue colors must be used. The .coe file is simply a list of the different 8-bit color values, going across and then down.

The Xilinx Core Generator can import .coe files and turn them into BRAMs. The BRAMs are all read-only, so they have inputs of the read Address and the Clock and have an output of the Data. The BRAM dimensions vary from sprite to sprite. The small ones have 8-bit Address inputs and the large ones have 14-bit Address inputs. The output is always 8 bits because it is always the 8-bit color value.

Figure 7: Sprites

7. Video Output

7.1 Overview of the Video Output

The video output does not actually exist in its own module, but rather over several modules, including the lab kit interface. One of these modules is the XVGA module, which generates the basic VGA signals, as well as the Hcount and Vcount signals that the Sprite Module relies on. There is also additional logic to read from the ZBT RAM and send the ZBT Data to the VGA port. The only other logic switches the game between camera mode and sprite mode. The portion of the video logic that reads from the ZBT RAM was written by I. Chuang.

The inputs of these modules include basic signals like clock and reset. However, they mainly include the ZBT Address, ZBT Data, and ZBT Write Enable, in addition to the Spritefinal signal that comes from the sprite module. The outputs of the Video output are the VGA red, green, and blue signals as well as the Hsync, Vsync, and Blank signals that all get sent through the lab kit to the VGA port. The Hsync, Vsync, and Blank signals from the XVGA Module have been delayed by three clock cycles in order to synchronize with the ZBT RAM's Data.

Camera mode uses data from ZBT RAM and ultimately outputs a black and white version of what the camera sees. This mode is useful for aligning the camera to point at the center of the screen. Game mode converts the Spritefinal data from the Sprite Module from 8-bit color to 24-bit color and outputs the result on the red, green, and blue lines of the VGA output.

The XVGA module's purpose is to create Hsync, Vsync, Blank, Hcount, and Vcount signals based on the Video Clock input. The VGA signal generated is the VGA signal of a 1024x768 sized screen that refreshes with a rate of 60 Hz. Hsync, Vsync, and Blank are actually very important to the VGA signal. Hsync is asserted whenever the pixel has reached the side, Vsync is asserted when it has reached the bottom, and Blank is asserted when the pixels are not on the screen. From these signals, Hcount and Vcount can also be derived and can be used to determine the coordinates of the pixel data that is currently being sent through the VGA port.

8. Conclusion

This project demonstrated a laser based gesture recognition system in a combat game. The video input system required a video analyzer module to input video data and process it into coordinate data for the laser dots. This data was fed into the gesture recognition module, which used the coordinate data to form segment data to determine which pattern the user was trying to execute. The gesture recognition module and the keyboard module connect to the game logic module, which determines what sprites should be used and where they should be placed. With this information, the sprite module was able to decide what the color of the VGA's current output pixel should be.

Overall, our project was interesting to work on and interesting to play. We wished to experiment with a new and different interface for gaming. This interface would probably most closely resemble the Nintendo DS or any other type of touch screen based gaming console. However, we discovered that video recognition of laser dots would also work very well if fine-

tuned and calibrated correctly.

Though the project was sick nasty awesome, there was definitely room for change and improvement. The version that was used in the demo had inferior video and gesture recognition modules, and the gestures were therefore slightly difficult to execute. Even while writing this paper, we were able to come up with several ways to improve the performance of these modules. There are also plenty of balancing aspects of the game logic that would not necessarily improve its performance, but would simply make it a lot more fun. For example, there is currently no advantage of being in the air since there is no way to attack in a downward direction. A final improvement would probably be to slightly optimize the sprite module such that it did not have to create a new BRAM for repeated sprite sub-module instances.

Near the end, our project was plagued with hardware issues. At this point, our project was huge, and the sprites used up 98% of the BRAMs. As we pushed the FPGA to its limits, we started getting strange behavior from the compiled result. For example, the second keyboard would work during one compilation, and then fail to work on the next even though nothing related to the keyboard was changed. As our problems worsened, so did Xilinx's penchant for crashing or corrupting our project files. Overall, Our project was interesting to work on and interesting to play. We wished to experiment with a different interface for gaming. This interface would probably most closely resemble the Nintendo DS or any other type of touch screen based gaming console.

In conclusion, our game was wicked sick and blew Harley and Rajeev's project out of the water!

Appendix I – Labkit File

```
//
// File:   kirbylaser.v
// Date:   12/10/08
// Author: Daniel Gerber <dgerb@mit.edu>
// Author: Tynan Smith <tssmith@mit.edu>
//

//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
//////////////////////////////////////////////////////////////////////////////
```

```
module zbt_6111_sample(beep, audio_reset_b,
                       ac97_sdata_out, ac97_sdata_in, ac97_synch,
                ac97_bit_clock,

                vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
                vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
                vga_out_vsync,

                tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
                tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
                tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

                tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
                tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
                tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
                tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

                ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
                ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

                ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
                ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

                clock_feedback_out, clock_feedback_in,

                flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
                flash_reset_b, flash_sts, flash_byte_b,

                rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

                mouse_clock, mouse_data, keyboard_clock, keyboard_data,

                clock_27mhz, clock1, clock2,

                disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_in,

                button0, button1, button2, button3, button_enter, button_right,
                button_left, button_down, button_up,

                switch,

                led,

                user1, user2, user3, user4,

                daughtercard,

                systemace_data, systemace_address, systemace_ce_b,
                systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

                analyzer1_data, analyzer1_clock,
                analyzer2_data, analyzer2_clock,
                analyzer3_data, analyzer3_clock,
                analyzer4_data, analyzer4_clock);

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
          vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
          tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
          tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
```

```
    input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
           tv_in_hff, tv_in_aff;
    output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
           tv_in_reset_b, tv_in_clock;
    inout  tv_in_i2c_data;

    inout  [35:0] ram0_data;
    output [18:0] ram0_address;
    output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
    output [3:0] ram0_bwe_b;

    inout  [35:0] ram1_data;
    output [18:0] ram1_address;
    output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
    output [3:0] ram1_bwe_b;

    input  clock_feedback_in;
    output clock_feedback_out;

    inout  [15:0] flash_data;
    output [23:0] flash_address;
    output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
    input  flash_sts;

    output rs232_txd, rs232_rts;
    input  rs232_rxd, rs232_cts;

    input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

    input  clock_27mhz, clock1, clock2;

    output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
    input  disp_data_in;
    output  disp_data_out;

    input  button0, button1, button2, button3, button_enter, button_right,
           button_left, button_down, button_up;
    input  [7:0] switch;
    output [7:0] led;

    inout [31:0] user1, user2, user3, user4;

    inout [43:0] daughtercard;

    inout  [15:0] systemace_data;
    output [6:0]  systemace_address;
    output systemace_ce_b, systemace_we_b, systemace_oe_b;
    input  systemace_irq, systemace_mpbrdy;

    output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                  analyzer4_data;
    output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

    ////////////////////////////////////////////////////////////////////////////
    //
    // I/O Assignments
    //
    ////////////////////////////////////////////////////////////////////////////

    // Audio Input and Output
    assign beep= 1'b0;
    assign audio_reset_b = 1'b0;
    assign ac97_synch = 1'b0;
    assign ac97_sdata_out = 1'b0;
/*
*/
    // ac97_sdata_in is an input

    // Video Output
    assign tv_out_ycrcb = 10'h0;
```

```
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

/* change lines below to enable ZBT RAM bank0 */

/*
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;  // clock enable
*/

/* enable RAM pins */

   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_bwe_b = 4'h0;

/**********/

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs
```

```
   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // LED Displays
/*
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
*/
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

   ////////////////////////////////////////////////////////////////////////
   // Demonstration of ZBT RAM as video memory

   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   wire clk = clock_65mhz;

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
                   .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clk, ~button_enter, user_reset);
```

```
   assign reset = user_reset | power_on_reset;

   // display module for debugging

   reg [63:0] dispdata;
   display_16hex hexdisp1(reset, clk, dispdata,
                          disp_blank, disp_clock, disp_rs, disp_ce_b,
                          disp_reset_b, disp_data_out);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

   // wire up to ZBT ram
   wire [35:0] vram_write_data;
   wire [35:0] vram_read_data;
   wire [18:0] vram_addr;
   wire        vram_we;

   zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
                 vram_write_data, vram_read_data,
                 ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

   // generate pixel value from reading ZBT memory
   wire [7:0]  vr_pixel;
   wire [18:0]        vram_addr1;

   vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
                    vram_addr1,vram_read_data);


       //video analyzer module to turn camera input into green and red laser dot coordinates

       wire [9:0] redx, redy, greenx, greeny;

       wire reddone, greendone, ready;

       wire [18:0] ntsc_addr;
       wire [35:0] ntsc_data;
       wire        ntsc_we;

       wire [7:0] diag1, diag2, diag3, diag4;



       video_module video_analyzer(
              .clk(clk), .clock_27mhz(clock_27mhz), .tv_in_line_clock1(tv_in_line_clock1),

                     .reset(reset), .tv_in_ycrcb(tv_in_ycrcb[19:10]),
              .tv_in_reset_b(tv_in_reset_b), .tv_in_i2c_clock(tv_in_i2c_clock),

                     .tv_in_i2c_data(tv_in_i2c_data),
                     .ntsc_addr(ntsc_addr), .ntsc_data(ntsc_data), .ntsc_we(ntsc_we),
                     .redx(redx), .redy(redy), .greenx(greenx), .greeny(greeny),

                     .reddone(reddone), .greendone(greendone), .ready(ready),

                     .diag1(diag1), .diag2(diag2), .diag3(diag3), .diag4(diag4));




// gesture recognition


       wire green_gesture_done;
```

```
        wire [1:0] green_speed;

        wire [6:0] green_pattern;

        wire [3:0] green_segment;

        wire [9:0] temp;



   gesture_recognizer greenizer(.ready(ready), .reset(reset), .clock(clock_27mhz),
.fast_clock(clk),

                            .pos_x(greenx), .pos_y(greeny), .done(green_gesture_done),

                            .speed_out(green_speed), .pattern(green_pattern),
.state(green_segment), .max_score(temp));



        wire red_gesture_done;
        wire [1:0] red_speed;
        wire [6:0] red_pattern;
        wire [3:0] red_segment;

   gesture_recognizer redizer(.ready(ready), .reset(reset), .clock(clock_27mhz),
.fast_clock(clk),
                            .pos_x(redx), .pos_y(redy), .done(red_gesture_done),
                            .speed_out(red_speed), .pattern(red_pattern), .state(red_segment));




        // game logic

   //sprite params
        /*parameter KIRBY = 0;
   parameter DEDEDE = 1;
   parameter SWORD = 2;
   parameter HAMMER = 3;
   parameter PLATFORM1 = 4;
   parameter PLATFORM2 = 5;
   parameter PLATFORM3 = 6;
   parameter PLATFORM4 = 7;
   parameter PLATFORM5 = 8;
   parameter PLATFORM6 = 9;
   parameter HEALTH1 = 10;
   parameter HEALTH2 = 11;
   parameter KAMMO1 = 12;
   parameter KAMMO2 = 13;
   parameter DAMMO1 = 14;
   parameter DAMMO2 = 15;
   parameter BLOOD1 = 16;
   parameter BLOOD2 = 17;
   parameter BLOOD3 = 18;
   parameter BLOOD4 = 19;
   parameter BLOOD5 = 20;
   parameter BLOOD6 = 21;
   parameter BLOOD7 = 22;
   parameter BLOOD8 = 23;
        parameter HURT1 = 24;
        parameter HURT2 = 25;*/



        wire [7:0] k_ascii, d_ascii;

        wire k_ascii_ready, d_ascii_ready, k_up, k_down, k_left, k_right, k_enter, d_up, d_down,
```

```
d_left, d_right, d_enter;


	ps2_ascii_input key1(clock_27mhz, reset, keyboard_clock, keyboard_data, k_ascii,
k_ascii_ready, k_up, k_down, k_left, k_right, k_enter);

	ps2_ascii_input key2(clock_27mhz, reset, mouse_clock, mouse_data, d_ascii, d_ascii_ready,
d_up, d_down, d_left, d_right, d_enter);


	reg pause;


	always @(posedge clock_27mhz) begin

		if (reset)

			pause <= 0;

		else

			pause <= pause ? (k_ascii_ready & k_enter) | (d_ascii_ready & d_enter) ?
0 : 1 :

(k_ascii_ready & k_enter) |
(d_ascii_ready & d_enter) ? 1 : 0;

	end


	wire [25:0] sprite_display, sprite_orientation;

	wire [9:0] sprite_x [25:0], sprite_y [25:0];

	wire [3:0] sprite_pic [25:0];

	wire [7:0] k_health, d_health;

	wire [6:0] k_floor;

	wire k_attack, k_defend;


	wire [8:0] spritefinal;

	wire [7:0] collisions;


	assign spritefinal[8] = 1'b0;


   game_logic #(.SPRITES(26),.COLLISIONS(8),

.INIT_SPRITES(26'b00_0000_0000_0000_0011_1111_0011),

.INIT_ORIENT(26'b00_0000_0000_0000_0000_0000_0101),

.INIT_POS_X({160'd0,10'd512,10'd512,10'd684,10'd250,10'd522,10'd10,20'd0,10'd900,10'd100}),

.INIT_POS_Y({160'd0,10'd500,10'd200,10'd550,10'd150,10'd350,10'd350,20'd0,10'd100,10'd100}),

.INIT_DATA({4'd1,4'd1,4'b0101,4'b0101,4'b0101,4'b0101,4'b0101,4'b0101,4'b0101,4'b0101,4'd1,4'd1,4
```

```
'd1,4'd1,40'd0,4'd1,4'd1}),


.SPRITE_WIDTH({128'd0,8'd90,8'd90,8'd90,8'd90,8'd90,8'd90,8'd45,8'd45,8'd60,8'd60}),

.SPRITE_HEIGHT({128'd0,8'd30,8'd30,8'd30,8'd30,8'd30,8'd30,8'd90,8'd90,8'd60,8'd60}),

                              .KEYS(8),.ATTACKS(4))

               gl (.reset(reset), .clock(clock_27mhz), .green_gesture_ready(green_gesture_done),
.red_gesture_ready(red_gesture_done), .vsync(vsync),

                                                .keyboard({d_right || ~button_right, d_left ||
~button_left, d_down || ~button_down, d_up || ~button_up,

                                                          k_right || ~button2, k_left || ~button3,
k_down || ~button1, k_up || ~button0}), .enable_fatal(switch[7]),

                                                .green_gesture(switch[1]?green_pattern:
{4'b0000,switch[7:5]}), .red_gesture(switch[1]?red_pattern:{4'b0000,switch[7:5]}),
.green_speed(green_speed), .red_speed(red_speed),

                                                .collisions(collisions|
{3'b000,switch[4],1'b0,switch[3],1'b0}), .green_x(greenx), .green_y(greeny), .red_x(redx),
.red_y(redy),

                  .sprite_display(sprite_display), .sprite_orientation(sprite_orientation),

                                                .sprite_x({sprite_x[25], sprite_x[24], sprite_x[23],
sprite_x[22], sprite_x[21], sprite_x[20], sprite_x[19], sprite_x[18], sprite_x[17], sprite_x[16],
sprite_x[15], sprite_x[14], sprite_x[13], sprite_x[12], sprite_x[11], sprite_x[10], sprite_x[9],
sprite_x[8], sprite_x[7], sprite_x[6], sprite_x[5], sprite_x[4], sprite_x[3], sprite_x[2],
sprite_x[1], sprite_x[0]}),

                                                .sprite_y({sprite_y[25], sprite_y[24], sprite_y[23],
sprite_y[22], sprite_y[21], sprite_y[20], sprite_y[19], sprite_y[18], sprite_y[17], sprite_y[16],
sprite_y[15], sprite_y[14], sprite_y[13], sprite_y[12], sprite_y[11], sprite_y[10], sprite_y[9],
sprite_y[8], sprite_y[7], sprite_y[6], sprite_y[5], sprite_y[4], sprite_y[3], sprite_y[2],
sprite_y[1], sprite_y[0]}),

                  .sprite_data({sprite_pic[25], sprite_pic[24], sprite_pic[23], sprite_pic[22],
sprite_pic[21], sprite_pic[20], sprite_pic[19], sprite_pic[18], sprite_pic[17], sprite_pic[16],
sprite_pic[15], sprite_pic[14], sprite_pic[13], sprite_pic[12], sprite_pic[11], sprite_pic[10],
sprite_pic[9], sprite_pic[8], sprite_pic[7], sprite_pic[6], sprite_pic[5], sprite_pic[4],
sprite_pic[3], sprite_pic[2], sprite_pic[1], sprite_pic[0]}),

                                                .kirby_health(k_health),.dedede_health(d_health),.pau
se(pause));



   // collision params
   /*parameter KIRBY_HAMMER = 0;
   parameter DEDEDE_SWORD = 1;
   parameter KIRBY_DAMMO1 = 2;
   parameter KIRBY_DAMMO2 = 3;
   parameter DEDEDE_KAMMO1 = 4;
   parameter DEDEDE_KAMMO2 = 5;
        parameter HAMMER_SWORD = 6;
        parameter KIRBY_DEDEDE = 7;   */



       sprite_module s(.clk(clk), .reset(reset), .hcount(hcount), .vcount(vcount),
                              .platform1x(sprite_x[4]), .platform1y(sprite_y[4]),
.platform2x(sprite_x[5]), .platform2y(sprite_y[5]),

                              .platform3x(sprite_x[6]), .platform3y(sprite_y[6]),
.platform4x(sprite_x[7]), .platform4y(sprite_y[7]),
```

```
                              .platform5x(sprite_x[8]), .platform5y(sprite_y[8]),
.platform6x(sprite_x[9]), .platform6y(sprite_y[9]),
            .kirbyx(sprite_x[0]), .kirbyy(sprite_y[0]), .dededex(sprite_x[1]),
.dededey(sprite_y[1]),
            .smallbloodx(sprite_x[24]), .smallbloody(sprite_y[24]), .bigbloodx(sprite_x[25]),
.bigbloody(sprite_y[25]),
            .chunk1x(sprite_x[16]), .chunk1y(sprite_y[16]), .chunk2x(sprite_x[17]),
.chunk2y(sprite_y[17]),

                              .chunk3x(sprite_x[18]), .chunk3y(sprite_y[18]),
.chunk4x(sprite_x[19]), .chunk4y(sprite_y[19]),
            .chunk5x(sprite_x[20]), .chunk5y(sprite_y[20]), .chunk6x(sprite_x[21]),
.chunk6y(sprite_y[21]),

                              .chunk7x(sprite_x[22]), .chunk7y(sprite_y[22]),
.chunk8x(sprite_x[23]), .chunk8y(sprite_y[23]),
            .swordx(sprite_x[2]), .swordy(sprite_y[2]), .hammerx(sprite_x[3]),
.hammery(sprite_y[3]),
            .kproj1x(sprite_x[12]), .kproj1y(sprite_y[12]), .kproj2x(sprite_x[13]),
.kproj2y(sprite_y[13]),
            .dproj1x(sprite_x[14]), .dproj1y(sprite_y[14]), .dproj2x(sprite_x[15]),
.dproj2y(sprite_y[15]),
            .kirbyid(sprite_display[0]?{sprite_orientation[0],sprite_pic[0]}:5'b00000),
.dededeid(sprite_display[1]?{sprite_orientation[1],sprite_pic[1]}:5'b00000),

                              .swordid(sprite_display[1]?
{sprite_orientation[2],sprite_pic[2]}:5'b00000), .hammerid(sprite_display[1]?
{sprite_orientation[3],sprite_pic[3]}:5'b00000),
            .smallbloodid(sprite_display[24]?{sprite_orientation[24],sprite_pic[24]
[1:0]}:3'b000), .bigbloodid(sprite_display[25]?{sprite_orientation[25],sprite_pic[25]
[1:0]}:3'b000),

                              .kirbyhealth(k_health), .dededehealth(d_health),
            .chunk1id(sprite_display[16]?sprite_pic[16][3:0]:3'b000),
.chunk2id(sprite_display[17]?sprite_pic[17][3:0]:3'b000),

                              .chunk3id(sprite_display[18]?sprite_pic[18][3:0]:3'b000),
.chunk4id(sprite_display[19]?sprite_pic[19][3:0]:3'b000),

                              .chunk5id(sprite_display[20]?sprite_pic[20][3:0]:3'b000),
.chunk6id(sprite_display[21]?sprite_pic[21][3:0]:3'b000),

                              .chunk7id(sprite_display[22]?sprite_pic[22][3:0]:3'b000),
.chunk8id(sprite_display[23]?sprite_pic[23][3:0]:3'b000),
            .kproj1id(sprite_display[12]?{sprite_orientation[12],sprite_pic[12][1:0]}:3'b000),
.kproj2id(sprite_display[13]?{sprite_orientation[13],sprite_pic[13][1:0]}:3'b000),

                              .dproj1id(sprite_display[14]?{sprite_orientation[14],sprite_pic[14]
[1:0]}:3'b000), .dproj2id(sprite_display[15]?{sprite_orientation[15],sprite_pic[15]
[1:0]}:3'b000),
                         .color(spritefinal[7:0]),
            .kvsd(collisions[7]), .kvsh(collisions[0]), .dvss(collisions[1]),
.svsh(collisions[6]),

                              .kvsb(collisions[3]), .kvsg(collisions[2]),
.dvssh(collisions[5]), .dvsh(collisions[4]));


   // video output logic starts here:
   // code to write pattern to ZBT memory
   reg [31:0]  count;
   always @(posedge clk) count <= reset ? 0 : count + 1;

   wire [18:0]        vram_addr2 = count[0+18:0];
   wire [35:0]        vpat = ( switch[1] ? {4{count[3+3:3],4'b0}}
                       : {4{count[3+4:4],4'b0}} );

   // mux selecting read/write to memory based on which write-enable is chosen

   wire        sw_ntsc = ~switch[7];
```

```
   wire        my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
   wire [18:0]         write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
   wire [35:0]         write_data = sw_ntsc ? ntsc_data : vpat;

//   wire      write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign    vram_addr = write_enable ? write_addr : vram_addr1;
//   assign    vram_we = write_enable;

   assign      vram_addr = my_we ? write_addr : vram_addr1;
   assign      vram_we = my_we;
   assign      vram_write_data = write_data;

   // select output pixel data

   reg [7:0]   pixel;
   wire        b,hs,vs;

   delayN dn1(clk,hsync,hs);  // delay by 3 cycles to sync with ZBT read
   delayN dn2(clk,vsync,vs);
   delayN dn3(clk,blank,b);

   always @(posedge clk)
     begin
        pixel <= switch[0] ? vr_pixel : {hcount[8:6],5'b0};
     end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.

       // dgerb added ability to switch between game and camera


   assign vga_out_red = switch[0] ? pixel : {spritefinal[7:5],5'b00000};
   assign vga_out_green = switch[0] ? pixel : {spritefinal[4:2],5'b00000};
   assign vga_out_blue = switch[0] ? pixel : {spritefinal[1:0],6'b000000};


   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   // debugging

   assign led[7:0] = ~{6'b000000,reset,switch[0]};


       reg [23:0] happycounter;

       reg finished;



   always @(posedge clock_27mhz) begin

          if(!finished && ready) begin

                     dispdata[63:24] <= {3'b000,k_defend,3'b000,k_attack,3'b000, |k_floor,
8'b0000_0000, red_segment ,1'b0, red_pattern, 2'b00, red_speed, 3'b000, red_gesture_done};

          end

              if(reset) begin

                     happycounter <= 0;

                     finished <= 0;

              end
```

```
            if(happycounter == 0) begin

                    // dispdata <= {vram_read_data,9'b0,vram_addr};

                    //dispdata <= {3'b000, fvh[2], 3'b000, fvh[1],3'b000, fvh[0], 2'b00, ycrcb,
tv_in_ycrcb};

                    //dispdata <= {6'b000000, greenx, 6'b000000, greeny, 7'b0000000, greendone,
7'b0000000, reddone, 15'b000000000000000, ready};

                    //dispdata <= {6'b000000, greenx, 6'b000000, greeny, 6'b000000, redx,
7'b0000000, greendone, 3'b0000000, reddone, 3'b0000000, ready};

                    dispdata[23:0] <= {2'b00, redx[9:0], 2'b00, redy[9:0]};

                    //finished <= !(green_segment == 4) && finished;

            end

                    //dispdata <= {ntsc_data,9'b0,ntsc_addr};
                    //finished <= green_gesture_done;

                    happycounter <= happycounter + 1;

      end



endmodule

//////////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)

module xvga(vclock,hcount,vcount,hsync,vsync,blank);
   input vclock;
   output [10:0] hcount;
   output [9:0] vcount;
   output      vsync;
   output      hsync;
   output      blank;

   reg   hsync,vsync,hblank,vblank,blank;
   reg [10:0]   hcount;    // pixel number on current line
   reg [9:0] vcount;   // line number

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   wire      hsyncon,hsyncoff,hreset,hblankon;
   assign    hblankon = (hcount == 1023);
   assign    hsyncon = (hcount == 1047);
   assign    hsyncoff = (hcount == 1183);
   assign    hreset = (hcount == 1343);

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

   // sync and blanking
   wire      next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always @(posedge vclock) begin
      hcount <= hreset ? 0 : hcount + 1;
      hblank <= next_hblank;
```

```
      hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

      blank <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

//////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

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 [7:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;

   wire [18:0]         vram_addr = {1'b0, vcount, hcount[9:2]};

   wire [1:0]   hc4 = hcount[1:0];
   reg [7:0]    vr_pixel;
   reg [35:0]   vr_data_latched;
   reg [35:0]   last_vr_data;

   always @(posedge clk)
     last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;

   always @(posedge clk)
     vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;

   always @(*/*)*/)           // each 36-bit word from RAM is decoded to 4 bytes
         case (hc4)
       2'd3: vr_pixel = last_vr_data[7:0];
       2'd2: vr_pixel = last_vr_data[7+8:0+8];
       2'd1: vr_pixel = last_vr_data[7+16:0+16];
       2'd0: vr_pixel = last_vr_data[7+24:0+24];
     endcase

endmodule // vram_display

//////////////////////////////////////////////////////////////////////////////
// parameterized delay line

module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 3;

   reg [NDELAY-1:0] shiftreg;
   wire             out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN
```

Appendix II – Video Module

```
//////////////////////////////////////////////////////////////////////////////////
//
// File:   video_module.v
// Date:   12/9/08
// Author: Daniel Gerber
//
// Video Module for Kirby Laser Attack
//
//////////////////////////////////////////////////////////////////////////////////
module video_module(
        clk, clock_27mhz, tv_in_line_clock1, reset, tv_in_ycrcb,
        tv_in_reset_b, tv_in_i2c_clock, tv_in_i2c_data,
                ntsc_addr, ntsc_data, ntsc_we,
                redx, redy, greenx, greeny, reddone, greendone, ready,
                diag1, diag2, diag3, diag4);

        input clk, clock_27mhz, tv_in_line_clock1, reset; // clk is 65 mhz
    input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
        output tv_in_reset_b, tv_in_i2c_clock, tv_in_i2c_data; // signals to adv7185
    output [18:0] ntsc_addr; // address line to zbt
    output [35:0] ntsc_data; // data line to zbt
    output ntsc_we; // write enable for NTSC data
        output reg [9:0] redx, greenx, redy, greeny; // x and y coords of red and green dot
        output reg reddone, greendone, ready;
                // reddone and greendone are held until screen reset
                // ready is asserted at screen reset
        output reg [7:0] diag1, diag2, diag3, diag4; // four optional diagnostic outputs


    // ADV7185 NTSC decoder interface code
    // adv7185 initialization module
    adv7185init adv7185(
                .reset(reset), .clock_27mhz(clock_27mhz), .source(1'b0),
                .tv_in_reset_b(tv_in_reset_b), .tv_in_i2c_clock(tv_in_i2c_clock),
```

```
                .tv_in_i2c_data(tv_in_i2c_data));

wire [29:0] ycrcb; // video data (luminance, chrominance)
wire [2:0] fvh; // sync for field, vertical, horizontal
wire dv;    // data valid

    // code to decode NTSC data into ycrcb
    ntsc_decode decode(
            .clk(tv_in_line_clock1), .reset(reset), .tv_in_ycrcb(tv_in_ycrcb),
            .ycrcb(ycrcb), .f(fvh[2]), .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

    wire [9:0] xcoord, ycoord;

// code to write NTSC data to video memory
ntsc_to_zbt n2z_modified(
            .clk(clk), .vclk(tv_in_line_clock1), .fvh(fvh),
                    .dv(dv), .din(ycrcb[29:22]), .sw(1'b0),
            .ntsc_addr(ntsc_addr), .ntsc_data(ntsc_data), .ntsc_we(ntsc_we),
                    .xout(xcoord), .yout(ycoord));

    wire [7:0] reddata, greendata, bluedata;

    // ycrcb to rgb converter
    YCrCb2RGB converter(
            .Y(ycrcb[29:20]), .Cr(ycrcb[19:10]), .Cb(ycrcb[9:0]),
                    .clk(tv_in_line_clock1), .rst(reset),
            .R(reddata), .G(greendata), .B(bluedata));

    // pipelined inputs
    reg [9:0] xcoordin[2:0], ycoordin[2:0];
    reg [9:0] oldxcoord;
    reg [23:0] rgb[2:0];
    // sum and counter for averaging
    reg [17:0] greensumx, greensumy, redsumx, redsumy; // sum of coordinate data
    reg [6:0] greencounter, redcounter; // amount of data acquired
```

```
// dividend, divisor, and counter for divider modules
reg [17:0] greenxdividend, greenydividend, redxdividend, redydividend; // sum
reg [6:0] greenxdivisor, greenydivisor, redxdivisor, redydivisor; // counter
reg [4:0] redxdivcount, greenxdivcount, redydivcount, greenydivcount;
// outputs from dividers
wire [17:0] redxtemp, redytemp, greenxtemp, greenytemp; // quotient
wire [6:0] redxrem, greenxrem, redyrem, greenyrem; // dangles
wire redxrfd, greenxrfd, redyrfd, greenyrfd; // dangles
// for finite state machine to set video_module outputs
reg redxready, redyready, greenxready, greenyready; // asserted when divider's done
reg [2:0] state, nextstate; // state variables of fsm

// divider modules
average redaveragerx(.clk(clk), .dividend(redxdividend), .divisor(redxdivisor),
        .quotient(redxtemp), .remainder(redxrem), .rfd(redxrfd));
average redaveragery(.clk(clk), .dividend(redydividend), .divisor(redydivisor),
        .quotient(redytemp), .remainder(redyrem), .rfd(redyrfd));
average greenaveragerx(.clk(clk), .dividend(greenxdividend), .divisor(greenxdivisor),
        .quotient(greenxtemp), .remainder(greenxrem), .rfd(greenxrfd));
average greenaveragery(.clk(clk), .dividend(greenydividend), .divisor(greenydivisor),
        .quotient(greenytemp), .remainder(greenyrem), .rfd(greenyrfd));

// output submodule
always @(posedge clk) begin
        // initialize variables
        if(reset) begin
                {greencounter, redcounter} <= 0;
                {greensumx, greensumy, redsumx, redsumy} <= 0;
                {redxdivcount, greenxdivcount, redydivcount, greenydivcount} <= 0;
                {redxready, redyready, greenxready, greenyready} <= 0;
                {state, nextstate} <= 0;
                {greenxdividend, greenydividend, redxdividend, redydividend} <= 0;
                {greenxdivisor, greenydivisor, redxdivisor, redydivisor} <= 0;
                {diag1, diag2, diag3, diag4} <= 0;
        end
```

```
		// synchronize inputs from 27 mhz clock to 65 mhz clock
		{xcoordin[2], xcoordin[1], xcoordin[0]} <= {xcoordin[1], xcoordin[0], xcoord};
		{ycoordin[2], ycoordin[1], ycoordin[0]} <= {ycoordin[1], ycoordin[0], ycoord};
		{rgb[2], rgb[1], rgb[0]} <= {rgb[1], rgb[0], reddata, greendata, bluedata};

		// add data to sum and update counter if pixel has detected red or green laser dot
		if(ycrcb[29:20] > 10'h2C0) begin
			if(redcounter != 127 && oldxcoord != xcoordin[2] && rgb[2][23:16] > rgb[2]
[15:8]) begin
				redsumx <= redsumx + xcoordin[2];
				redsumy <= redsumy + ycoordin[2];
				redcounter <= redcounter + 1;
			end
			if(greencounter != 127 && oldxcoord != xcoordin[2] && rgb[2][23:16] <
rgb[2][15:8]) begin
				greensumx <= greensumx + xcoordin[2];
				greensumy <= greensumy + ycoordin[2];
				greencounter <= greencounter + 1;
			end
		end

		// this condition makes sure that only one data point is added per coordinate
		oldxcoord <= (oldxcoord != xcoordin[2]) ? xcoordin[2] : oldxcoord;



		// at top left corner of scan, input sum and counter to divider
		if(xcoordin[2] == 31 && ycoordin[2] == 31) begin
			if(redcounter > 31) begin
				redxdivcount <= 1;
				redxdividend <= redsumx;
				redxdivisor <= redcounter;
				redydivcount <= 1;
				redydividend <= redsumy;
				redydivisor <= redcounter;
```

```
                reddone <= 1;
        end
        else
                reddone <= 0;
        if(greencounter > 31) begin
                greenxdivcount <= 1;
                greenxdividend <= greensumx;
                greenxdivisor <= greencounter;
                greenydivcount <= 1;
                greenydividend <= greensumy;
                greenydivisor <= greencounter;
                greendone <= 1;
        end
        else
                greendone <= 0;
end
// at the next coordinate, clear sum and counter and start over again
else if(xcoordin[2] == 32 && ycoordin[2] == 31) begin
        redcounter <= 0;
        greencounter <= 0;
        redsumx <= 0;
        redsumy <= 0;
        greensumx <= 0;
        greensumy <= 0;
        // if no dots detected, enter fsm at state 3, don't update module output
        if(greendone == 0 && reddone == 0) begin
                ready <= 1;
                {redxready, redyready, greenxready, greenyready} <= 0;
                nextstate <= 3;
        end
end

// if divider's counter timer is done, signal ready
if(redxdivcount != 0)
        if(redxdivcount == 30) begin
```

```
                redxready <= 1;
                redxdivcount <= 0;
        end
        else redxdivcount <= redxdivcount + 1;
if(redydivcount != 0)
        if(redydivcount == 30) begin
                redyready <= 1;
                redydivcount <= 0;
        end
        else redydivcount <= redydivcount + 1;
if(greenxdivcount != 0)
        if(greenxdivcount == 30) begin
                greenxready <= 1;
                greenxdivcount <= 0;
        end
        else greenxdivcount <= greenxdivcount + 1;
if(greenydivcount != 0)
        if(greenydivcount == 30) begin
                greenyready <= 1;
                greenydivcount <= 0;
        end
        else greenydivcount <= greenydivcount + 1;

// if all dividers are done, enter the fsm at state 1
if(redxready && redyready && (greendone == 0) ||
                greenxready && greenyready && (reddone == 0) ||
                redxready && redyready && greenxready && greenyready) nextstate <=
1;

case(state)
        // state 1: filler state
        1: begin
                nextstate <= 2;
                {redxready, redyready, greenxready, greenyready} <= 0;
                end
```

```
                        // state 2: update output with divider's quotients, assert ready
                        2: begin
                                ready <= 1;
                                {redx, redy, greenx, greeny} <= {redxtemp[9:0], redytemp[9:0],
greenxtemp[9:0], greenytemp[9:0]};
                                nextstate <= 3;
                                end
                        // state 3: filler state
                        3: nextstate <= 4;
                        // state 4: reset divider and "done" values, unassert ready
                        4: begin
                                {reddone, greendone} <= 0;
                                ready <= 0;
                                {redxdivcount, greenxdivcount, redydivcount, greenydivcount} <= 0;
                                {greenxdividend, greenydividend, redxdividend, redydividend} <= 0;
                                {greenxdivisor, greenydivisor, redxdivisor, redydivisor} <= 0;
                                nextstate <= 0;
                                end
                        // state 0 or other: do nothing
                        default: state <= 0;
                endcase

                state <= nextstate;

        end

endmodule
```

Appendix III – Video Decoder

```
//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
//

//////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output      f;
   output      v;
   output      h;
   output      data_valid;
   // output [4:0] state;

   parameter   SYNC_1 = 0;
   parameter   SYNC_2 = 1;
   parameter   SAV_f1_cb0 = 3;
   parameter   SAV_f1_y0 = 4;
   parameter   SAV_f1_cr1 = 5;
   parameter   SAV_f1_y1 = 6;
   parameter   EAV_f1 = 7;
   parameter   SAV_VBI_f1 = 8;
   parameter   EAV_VBI_f1 = 9;
   parameter   SAV_f2_cb0 = 10;
   parameter   SAV_f2_y0 = 11;
   parameter   SAV_f2_cr1 = 12;
   parameter   SAV_f2_y1 = 13;
   parameter   EAV_f2 = 14;
   parameter   SAV_VBI_f2 = 15;
   parameter   EAV_VBI_f2 = 16;



   // In the start state, the module doesn't know where
   // in the sequence of pixels, it is looking.

   // Once we determine where to start, the FSM goes through a normal
```

```
// sequence of SAV process_YCrCb EAV... repeat

// The data stream looks as follows
// SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV sequence
// There are two things we need to do:
//   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
//   2. Decode the subsequent data

reg [4:0]   current_state = 5'h00;
reg [9:0]   y = 10'h000;  // luminance
reg [9:0]   cr = 10'h000; // chrominance
reg [9:0]   cb = 10'h000; // more chrominance

assign      state = current_state;

always @ (posedge clk)
  begin
    if (reset)
      begin

      end
    else
      begin
         // these states don't do much except allow us to know where we are in the stream.
         // whenever the synchronization code is seen, go back to the sync_state before
         // transitioning to the new state
         case (current_state)
           SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
           SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
           SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                                    (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                                    (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                                    (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
                                    (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                                    (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                                    (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                                    (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

           SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
           SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
           SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
           SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;

           SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
           SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
           SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
           SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

           // These states are here in the event that we want to cover these signals
           // in the future. For now, they just send the state machine back to SYNC_1
           EAV_f1: current_state <= SYNC_1;
           SAV_VBI_f1: current_state <= SYNC_1;
           EAV_VBI_f1: current_state <= SYNC_1;
           EAV_f2: current_state <= SYNC_1;
           SAV_VBI_f2: current_state <= SYNC_1;
           EAV_VBI_f2: current_state <= SYNC_1;

         endcase
      end
  end // always @ (posedge clk)

// implement our decoding mechanism

wire y_enable;
wire cr_enable;
wire cb_enable;

// if y is coming in, enable the register
// likewise for cr and cb
assign y_enable = (current_state == SAV_f1_y0) ||
```

```
                    (current_state == SAV_f1_y1) ||
                    (current_state == SAV_f2_y0) ||
                    (current_state == SAV_f2_y1);
   assign cr_enable = (current_state == SAV_f1_cr1) ||
                      (current_state == SAV_f2_cr1);
   assign cb_enable = (current_state == SAV_f1_cb0) ||
                      (current_state == SAV_f2_cb0);

   // f, v, and h only go high when active
   assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;

   // data is valid when we have all three values: y, cr, cb
   assign data_valid = y_enable;
   assign ycrcb = {y,cr,cb};

   reg   f = 0;

   always @ (posedge clk)
     begin
        y <= y_enable ? tv_in_ycrcb : y;
        cr <= cr_enable ? tv_in_ycrcb : cr;
        cb <= cb_enable ? tv_in_ycrcb : cb;
        f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule



//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
// Register 0
//////////////////////////////////////////////////////////////////////////////////

`define INPUT_SELECT                            4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                              4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

//////////////////////////////////////////////////////////////////////////////////
// Register 1
//////////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY                           2'h0
```

```
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE                    1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT                      1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING                     1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                                 1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE                1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM, `FOUR_TIMES_SAMPLING,
`DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}

////////////////////////////////////////////////////////////////////////////////
// Register 2
////////////////////////////////////////////////////////////////////////////////

`define Y_PEAKING_FILTER                        3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                                  2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

////////////////////////////////////////////////////////////////////////////////
// Register 3
////////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                        2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                           4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS                 1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                              1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT,
```

```
`INTERFACE_SELECT}

////////////////////////////////////////////////////////////////////////////
// Register 4
////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                       1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                              1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible

`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

////////////////////////////////////////////////////////////////////////////
// Register 5
////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                 4'b0000
`define GPO_0_1_ENABLE                          1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                          1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                     1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                            1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE,
`GPO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS}

////////////////////////////////////////////////////////////////////////////
// Register 7
////////////////////////////////////////////////////////////////////////////

`define FIFO_FLAG_MARGIN                        5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                              1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                    1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                     1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RESET,
`FIFO_FLAG_MARGIN}

////////////////////////////////////////////////////////////////////////////
// Register 8
////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                       8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register 9
////////////////////////////////////////////////////////////////////////////

`define INPUT_SATURATION_ADJUST                     8'h8C
```

```
`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register A
////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST                      8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register B
////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                             8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

////////////////////////////////////////////////////////////////////////////
// Register C
////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                   1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE         1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE                        6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE, `DEFAULT_VALUE_AUTOMATIC_ENABLE,
`DEFAULT_VALUE_ENABLE}

////////////////////////////////////////////////////////////////////////////
// Register D
////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                       4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                       4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

////////////////////////////////////////////////////////////////////////////
// Register E
////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE             1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL            2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE               4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_CONTROL,
`TEMPORAL_DECIMATION_ENABLE}

////////////////////////////////////////////////////////////////////////////
// Register F
////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                     2'h0
  // 0: Full operation
  // 1: CVBS only
```

```
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY              1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                    1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR                1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                         1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                        1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                              1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP,
`POWER_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY,
`POWER_SAVE_CONTROL}

////////////////////////////////////////////////////////////////////////////
// Register 33
////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                       1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES               1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                             3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                              1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE, `AVERAGE_BIRIGHTNESS_LINES,
`PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
```

```
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB

`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
                    tv_in_i2c_clock, tv_in_i2c_data);

   input reset;
   input clock_27mhz;
   output tv_in_reset_b; // Reset signal to ADV7185
   output tv_in_i2c_clock; // I2C clock output to ADV7185
   output tv_in_i2c_data; // I2C data line to ADV7185
   input source; // 0: composite, 1: s-video

   initial begin
      $display("ADV7185 Initialization values:");
      $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
      $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
      $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
      $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
      $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
      $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
      $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
      $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
      $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
      $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
      $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
      $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
      $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
      $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
      $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
      $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
   end

   //
   // Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
   //

   reg [7:0] clk_div_count, reset_count;
   reg clock_slow;
   wire reset_slow;

   initial
```

```
  begin
     clk_div_count <= 8'h00;
     // synthesis attribute init of clk_div_count is "00";
     clock_slow <= 1'b0;
     // synthesis attribute init of clock_slow is "0";
  end

always @(posedge clock_27mhz)
  if (clk_div_count == 26)
    begin
       clock_slow <= ~clock_slow;
       clk_div_count <= 0;
    end
  else
    clk_div_count <= clk_div_count+1;

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign reset_slow = reset_count != 0;

//
// I2C driver
//

reg load;
reg [7:0] data;
wire ack, idle;

i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
        .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
        .sda(tv_in_i2c_data));

//
// State machine
//

reg [7:0] state;
reg tv_in_reset_b;
reg old_source;

always @(posedge clock_slow)
   if (reset_slow)
    begin
       state <= 0;
       load <= 0;
       tv_in_reset_b <= 0;
       old_source <= 0;
    end
   else
    case (state)
      8'h00:
        begin
           // Assert reset
           load <= 1'b0;
           tv_in_reset_b <= 1'b0;
           if (!ack)
             state <= state+1;
        end
      8'h01:
        state <= state+1;
      8'h02:
        begin
           // Release reset
           tv_in_reset_b <= 1'b1;
           state <= state+1;
                end
```

```
8'h03:
  begin
     // Send ADV7185 address
     data <= 8'h8A;
     load <= 1'b1;
     if (ack)
      state <= state+1;
  end
8'h04:
  begin
     // Send subaddress of first register
     data <= 8'h00;
     if (ack)
      state <= state+1;
  end
8'h05:
  begin
     // Write to register 0
     data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
     if (ack)
      state <= state+1;
  end
8'h06:
  begin
     // Write to register 1
     data <= `ADV7185_REGISTER_1;
     if (ack)
      state <= state+1;
  end
8'h07:
  begin
     // Write to register 2
     data <= `ADV7185_REGISTER_2;
     if (ack)
      state <= state+1;
  end
8'h08:
  begin
     // Write to register 3
     data <= `ADV7185_REGISTER_3;
     if (ack)
      state <= state+1;
  end
8'h09:
  begin
     // Write to register 4
     data <= `ADV7185_REGISTER_4;
     if (ack)
      state <= state+1;
  end
8'h0A:
  begin
     // Write to register 5
     data <= `ADV7185_REGISTER_5;
     if (ack)
      state <= state+1;
  end
8'h0B:
  begin
     // Write to register 6
     data <= 8'h00; // Reserved register, write all zeros
     if (ack)
      state <= state+1;
  end
8'h0C:
  begin
     // Write to register 7
     data <= `ADV7185_REGISTER_7;
     if (ack)
      state <= state+1;
```

```
  end
8'h0D:
  begin
     // Write to register 8
     data <= `ADV7185_REGISTER_8;
     if (ack)
      state <= state+1;
  end
8'h0E:
  begin
     // Write to register 9
     data <= `ADV7185_REGISTER_9;
     if (ack)
      state <= state+1;
  end
8'h0F: begin
   // Write to register A
   data <= `ADV7185_REGISTER_A;
 if (ack)
   state <= state+1;
end
8'h10:
  begin
     // Write to register B
     data <= `ADV7185_REGISTER_B;
     if (ack)
      state <= state+1;
  end
8'h11:
  begin
     // Write to register C
     data <= `ADV7185_REGISTER_C;
     if (ack)
      state <= state+1;
  end
8'h12:
  begin
     // Write to register D
     data <= `ADV7185_REGISTER_D;
     if (ack)
      state <= state+1;
  end
8'h13:
  begin
     // Write to register E
     data <= `ADV7185_REGISTER_E;
     if (ack)
      state <= state+1;
  end
8'h14:
  begin
     // Write to register F
     data <= `ADV7185_REGISTER_F;
     if (ack)
      state <= state+1;
  end
8'h15:
  begin
     // Wait for I2C transmitter to finish
     load <= 1'b0;
     if (idle)
      state <= state+1;
  end
8'h16:
  begin
     // Write address
     data <= 8'h8A;
     load <= 1'b1;
     if (ack)
      state <= state+1;
```

```
  end
8'h17:
  begin
     data <= 8'h33;
     if (ack)
      state <= state+1;
  end
8'h18:
  begin
     data <= `ADV7185_REGISTER_33;
     if (ack)
      state <= state+1;
  end
8'h19:
  begin
     load <= 1'b0;
     if (idle)
      state <= state+1;
  end

8'h1A: begin
   data <= 8'h8A;
   load <= 1'b1;
   if (ack)
     state <= state+1;
end
8'h1B:
  begin
     data <= 8'h33;
     if (ack)
      state <= state+1;
  end
8'h1C:
  begin
     load <= 1'b0;
     if (idle)
      state <= state+1;
  end
8'h1D:
  begin
     load <= 1'b1;
     data <= 8'h8B;
     if (ack)
      state <= state+1;
  end
8'h1E:
  begin
     data <= 8'hFF;
     if (ack)
      state <= state+1;
  end
8'h1F:
  begin
     load <= 1'b0;
     if (idle)
      state <= state+1;
  end
8'h20:
  begin
     // Idle
     if (old_source != source) state <= state+1;
     old_source <= source;
  end
8'h21: begin
   // Send ADV7185 address
   data <= 8'h8A;
   load <= 1'b1;
   if (ack) state <= state+1;
end
8'h22: begin
```

```
            // Send subaddress of register 0
            data <= 8'h00;
            if (ack) state <= state+1;
         end
         8'h23: begin
            // Write to register 0
            data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
            if (ack) state <= state+1;
         end
         8'h24: begin
            // Wait for I2C transmitter to finish
            load <= 1'b0;
            if (idle) state <= 8'h20;
         end
       endcase

endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
          state <= 0;
          ack <= 0;
       end
     else
       case (state)
         8'h00: // idle
           begin
              scl <= 1'b1;
              sdai <= 1'b1;
              ack <= 1'b0;
              idle <= 1'b1;
              if (load)
                begin
                   ldata <= data;
                   ack <= 1'b1;
                   state <= state+1;
                end
           end
         8'h01: // Start
           begin
              ack <= 1'b0;
              idle <= 1'b0;
              sdai <= 1'b0;
              state <= state+1;
           end
         8'h02:
           begin
```

```
      scl <= 1'b0;
      state <= state+1;
    end
  8'h03: // Send bit 7
    begin
      ack <= 1'b0;
      sdai <= ldata[7];
      state <= state+1;
    end
  8'h04:
    begin
      scl <= 1'b1;
      state <= state+1;
    end
  8'h05:
    begin
      state <= state+1;
    end
  8'h06:
    begin
      scl <= 1'b0;
      state <= state+1;
    end
  8'h07:
    begin
      sdai <= ldata[6];
      state <= state+1;
    end
  8'h08:
    begin
      scl <= 1'b1;
      state <= state+1;
    end
  8'h09:
    begin
      state <= state+1;
    end
  8'h0A:
    begin
      scl <= 1'b0;
      state <= state+1;
    end
  8'h0B:
    begin
      sdai <= ldata[5];
      state <= state+1;
    end
  8'h0C:
    begin
      scl <= 1'b1;
      state <= state+1;
    end
  8'h0D:
    begin
      state <= state+1;
    end
  8'h0E:
    begin
      scl <= 1'b0;
      state <= state+1;
    end
  8'h0F:
    begin
      sdai <= ldata[4];
      state <= state+1;
    end
  8'h10:
    begin
      scl <= 1'b1;
      state <= state+1;
```

```
  end
8'h11:
  begin
     state <= state+1;
  end
8'h12:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h13:
  begin
     sdai <= ldata[3];
     state <= state+1;
  end
8'h14:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h15:
  begin
     state <= state+1;
  end
8'h16:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h17:
  begin
     sdai <= ldata[2];
     state <= state+1;
  end
8'h18:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h19:
  begin
     state <= state+1;
  end
8'h1A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1B:
  begin
     sdai <= ldata[1];
     state <= state+1;
  end
8'h1C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h1D:
  begin
     state <= state+1;
  end
8'h1E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1F:
  begin
     sdai <= ldata[0];
```

```
            state <= state+1;
          end
        8'h20:
          begin
            scl <= 1'b1;
            state <= state+1;
          end
        8'h21:
          begin
            state <= state+1;
          end
        8'h22:
          begin
            scl <= 1'b0;
            state <= state+1;
          end
        8'h23: // Acknowledge bit
          begin
            state <= state+1;
          end
        8'h24:
          begin
            scl <= 1'b1;
            state <= state+1;
          end
        8'h25:
          begin
            state <= state+1;
          end
        8'h26:
          begin
            scl <= 1'b0;
            if (load)
             begin
                ldata <= data;
                ack <= 1'b1;
                state <= 3;
             end
            else
             state <= state+1;
          end
        8'h27:
          begin
            sdai <= 1'b0;
            state <= state+1;
          end
        8'h28:
          begin
            scl <= 1'b1;
            state <= state+1;
          end
        8'h29:
          begin
            sdai <= 1'b1;
            state <= 0;
          end
      endcase

endmodule
```

Appendix IV – YCrCb to RGB Converter

```
/*********************************************************************************
 **
 ** Module: ycrcb2rgb
 **
 ** Generic Equations:
 *********************************************************************************/

module YCrCb2RGB ( R, G, B, clk, rst, Y, Cr, Cb );

output [7:0]  R, G, B;

input clk,rst;
input[9:0] Y, Cr, Cb;

wire [7:0] R,G,B;
reg [20:0] R_int,G_int,B_int,X_int,A_int,B1_int,B2_int,C_int;
reg [9:0] const1,const2,const3,const4,const5;
reg[9:0] Y_reg, Cr_reg, Cb_reg;

//registering constants
always @ (posedge clk)
begin
 const1 = 10'b 0100101010; //1.164 = 01.00101010
 const2 = 10'b 0110011000; //1.596 = 01.10011000
 const3 = 10'b 0011010000; //0.813 = 00.11010000
 const4 = 10'b 0001100100; //0.392 = 00.01100100
 const5 = 10'b 1000000100; //2.017 = 10.00000100
end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
      Y_reg <= 0; Cr_reg <= 0; Cb_reg <= 0;
      end
```

```
   else
      begin
         Y_reg <= Y; Cr_reg <= Cr; Cb_reg <= Cb;
      end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       A_int <= 0; B1_int <= 0; B2_int <= 0; C_int <= 0; X_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     A_int <= (const2 * (Cr_reg - 'd512));
     B1_int <= (const3 * (Cr_reg - 'd512));
     B2_int <= (const4 * (Cb_reg - 'd512));
     C_int <= (const5 * (Cb_reg - 'd512));
     end

always @ (posedge clk or posedge rst)
   if (rst)
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     R_int <= X_int + A_int;
     G_int <= X_int - B1_int - B2_int;
     B_int <= X_int + C_int;
     end




/*always @ (posedge clk or posedge rst)
   if (rst)
```

```
      begin
       R_int <= 0; G_int <= 0; B_int <= 0;
      end
   else
     begin
     X_int <= (const1 * (Y_reg - 'd64)) ;
     R_int <= X_int + (const2 * (Cr_reg - 'd512));
     G_int <= X_int - (const3 * (Cr_reg - 'd512)) - (const4 * (Cb_reg - 'd512));
     B_int <= X_int + (const5 * (Cb_reg - 'd512));
     end

*/
/* limit output to 0 - 4095, <0 equals o and >4095 equals 4095 */
assign R = (R_int[20]) ? 0 : (R_int[19:18] == 2'b0) ? R_int[17:10] : 8'b11111111;
assign G = (G_int[20]) ? 0 : (G_int[19:18] == 2'b0) ? G_int[17:10] : 8'b11111111;
assign B = (B_int[20]) ? 0 : (B_int[19:18] == 2'b0) ? B_int[17:10] : 8'b11111111;

endmodule
```

Appendix V – NTSC to ZBT Converter

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

// Modified by dgerb for Kirby Laser Attack project
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.

//////////////////////////////////////////////////////////////////////////
// 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, sw,

// dgerb modifications

       xout, yout

// end of modifications

       );

   input        clk;   // system clock
   input        vclk;  // video clock from camera
   input [2:0]          fvh;
   input        dv;
   input [7:0]          din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output       ntsc_we;      // write enable for NTSC data
   input        sw;           // switch which determines mode (for debugging)



// dgerb modifications

       output [9:0] xout, yout;
// end of modifications


   parameter    COL_START = 10'd30;
   parameter    ROW_START = 10'd30;

   // 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 [7:0]    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;


// dgerb modifications

       reg [9:0] xbout[1:0], ybout[1:0];
```

```
// end of modifications


   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



// dgerb modifications
        // pipeline xout and yout on the vclk for easier synchronization

        //      with the ycrcb converter (which also uses vclk)

        {xbout[1], xbout[0]} <= {xbout[0], col};

        {ybout[1], ybout[0]} <= {ybout[0], row};
// end of modifications


     end


// dgerb modifications
        assign xout = xbout[1];
        assign yout = ybout[1];
// end of modifications


   // synchronize with system clock

   reg [9:0] x[1:0],y[1:0];
   reg [7:0] 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};


//modified

        if(col == 100 || row == 100) data[0] <= 0;

               else data[0] <= vdata;

        data[1] <= data[0];

//end of mod



        //{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 [31:0] mydata;
    always @(posedge clk)
      if (we_edge)
        mydata <= { mydata[23:0], data[1] };

    // compute address to store data in

    wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};

    // alternate (256x192) image data and address
    wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
    wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};

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

    reg [18:0] ntsc_addr;
    reg [35:0] ntsc_data;
    wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][1:0]==2'b00));

    always @(posedge clk)
      if ( ntsc_we )
        begin
          ntsc_addr <= sw ? myaddr2 : myaddr;        // normal and expanded modes
          ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
        end

endmodule // ntsc_to_zbt
```

Appendix VI – Gesture Recognition Verilog

```
//grmodule
module gesture_recognizer(input wire ready, reset, clock, fast_clock,
                                        input wire [9:0] pos_x, pos_y,
                                        output wire done,
                                        output wire [1:0] speed_out,
                                        output wire [6:0] pattern,
                                                    output wire [3:0] state,
                                                    output wire signed [9:0] max_score);

       // params
       parameter down_sample = 1;

                  wire seg_done;
                  wire [1:0] speed;

        reg [1:0] clk_count;
        reg pt_ready, prev_clock;

        always @(posedge fast_clock) begin
              if (reset) begin
              clk_count <= 0;
              pt_ready <= 0;
              end else if (ready) begin
              pt_ready <= 1;
              clk_count <= 0;
              end else if (pt_ready && prev_clock != clock) begin
              clk_count <= clk_count + 1;
              if (clk_count == 2) begin
                     pt_ready <= 0;
                     clk_count <= 0;
              end
              end
              prev_clock <= clock;
        end

                  segment_analyzer
              sega(.ready(pt_ready),.reset(reset),.clock(clock),.pos_x(pos_x),.pos_y(pos_y),.don
              e(seg_done),.speed(speed),.state(state));
                  pattern_analyzer
              pata(.ready(seg_done),.reset(reset),.clock(clock),.speed_in(speed),.segment(state)
              ,.done(done),.speed_out(speed_out),.pattern(pattern),.max_score(max_score));
              endmodule
```

Appendix VII – Segment Analyzer Verilog

```
//segamodule
module segment_analyzer(
                                              input ready, reset, clock,
                                              input [9:0] pos_x, pos_y,
                                              output reg done,
                                              output reg [1:0] speed,
                                              output reg [3:0] state);

                        // parameters to describe the segments
                        parameter s_horizontal_r = 4'b0000;
                        parameter s_horizontal_l = 4'b1111;
                        parameter s_vertical_t = 4'b0011;
                        parameter s_vertical_b = 4'b1100;
                        parameter s_diagonal_tr = 4'b0001;
                        parameter s_diagonal_tl = 4'b0111;
                        parameter s_diagonal_br = 4'b1000;
                        parameter s_diagonal_bl = 4'b1110;
                        parameter s_stop = 4'b0010;
                        parameter s_start = 4'b0100;
```

```
        // parameters to control the detection mechanisms
        parameter num_points = 4;
        parameter num_segs = 2;
        parameter move_cutoff = 30;
        parameter move_dist = 2;


        // registers for storing x and y coordinates of the previous point, current
point
        // and the start of the segment
        reg signed [10:0] prev_x, prev_y, start_x, start_y, new_x, new_y;

        // signed registers for storing the difference between points
        reg signed [10:0] x_diff, y_diff;

        // registers for storing the segments
        reg [3:0] prev_state;
        //reg [3:0] segments [31:0];

        // various counters used to keep track of points, similar segments etc
        reg [4:0] seg_count, state_count, point_count;
        reg [5:0] not_move;
    reg [30:0] length_count;

    // registers used to help pipeline
    reg [3:0] compute_stage;
    reg comparison;

    always @(posedge clock) begin
        if (reset) begin
            // intialize all the registers
            seg_count <= 0;
            state_count <= 0;
            point_count <= 0;
            length_count <= 0;
            not_move <= 0;
            state <= s_stop;
            prev_state <= s_stop;
            prev_x <= pos_x;
            prev_y <= pos_y;
            start_x <= pos_x;
            start_y <= pos_y;
            new_x <= pos_x;
            new_y <= pos_y;
            diff_x <= 0;
            diff_y <= 0;
            compute_stage <= 0;
        end
        else if(ready || compute_stage != 0) begin
            case (compute_stage)
                0: // get the new x and y coords and compute the movement
                   speed <= length_count > 810000000 ? 2'b11 :
                            length_count > 540000000 ? 2'b10 :
                            length_count > 270000000 ? 2'b01 : 2'b00;

                   length_count <= length_count + 1;
                   compute_stage <= 1;
                   new_x <= pos_x;
                   new_y <= pos_y;
                   diff_x <= pos_x - prev_x;
                   diff_y <= pos_y - prev_y;
                   done <= 0;
                1: // determine if we have moved enough
                   compute_stage <= 2;
                   comparison <= (diff_x * diff_x + diff_y * diff_y) > move_dist;
                2: // if we have moved enough, check for start of pattern,
otherwise increment non-movement
                   if (comparison) begin
                       not_move <= 0;
```

```
                                if(state == s_stop) begin
                                        length_count <= 0;
                                        count <= 0;
                                        state <= s_start;
                                        done <= 1;
                                        compute_stage <= 7;
                                        segCount <= 0;
                                        stateCount <= 0;
                                        start_x <= new_x;
                                        start_y <= new_y;
                                end
                                else begin
                                    compute_stage <= 3;
                                end
                                count <= count + 1;
                                prev_x <= new_x;
                               prev_y <= new_y;
                   end
                   else begin
                       not_move <= not_move + 1;
                       compute_stage <= 4;
                   end
               3: // for the case where we have moved, check if we have enough
points
                         if (count == (num_points - 1)) begin
                                 x_diff <= new_x - start_x;
                                 y_diff <= new_y - start_y;
                                 count <= 0;
                                 start_x <= new_x;
                                 start_y <= new_y;
                                 compute_stage <= 5;
                               end else begin
                                 compute_stage <= 0;
                               end
               4: // if we haven't moved, see if we have been sitting to long
                            if (not_move > move_cutoff) begin
                                 not_move <= 0;
                                 state <= s_stop;
                                 //segments[seg_count] = s_stop;
                                 done <= 1;
                                 compute_stage <= 7;
                             end
               5: // if we have enough points, determine the segment
                            if (x_diff > 0) begin
                                       if (y_diff > 0) begin
                                              if (x_diff > (y_diff >> 1) begin
                                                     if (x_diff > (y_diff << 1))
begin
                                                            state <=
s_horizontal_r;
                                                     end else begin
                                                            state <=
s_diagonal_br;
                                                     end
                                              end else begin
                                                        state <= s_vertical_b;
                                              end
                                       end else begin
                                              if (x_diff > (y_diff >> 1)) begin
                                                     if (x_diff > (y_diff << 1))
begin
                                                            state <=
s_horizontal_r;
                                                     end else begin
                                                            state <=
s_diagonal_tr;
                                                     end
                                              end else begin
                                                     state <= s_vertical_t;
                                              end
```

```
                                        end
                                    end else begin
                                        if (y_diff > 0) begin
                                            if (x_diff + (y_diff >> 1) < 0)
begin
                                                if (x_diff + (y_diff << 1) <
0) begin
                                                    state <=
s_horizontal_l;
                                                end else begin
                                                    state <=
s_diagonal_bl;
                                                end
                                            end else begin
                                                state <= s_vertical_b;
                                            end
                                        end else begin
                                            if (x_diff < (y_diff >>> 1)) begin
                                                if (x_diff <  (y_diff <<< 1))
begin
                                                    state <=
s_horizontal_l;
                                                end else begin
                                                    state <=
s_diagonal_tl;
                                                end
                                            end else begin
                                                state <= s_vertical_t;
                                            end
                                        end
                                    end
                                    compute_stage <= 6;
                                6: // after determining the segment, check if we have
had enough of the same segment
                                    if(prev_state == state) begin
                                        state_count <= state_count + 1;
                                        if ((state_count + 1) == num_segs) begin
                                           if (seg_count == 31) begin
                                                state <= s_stop;
                                                //segments[seg_count] = s_stop;
                                                done <= 1;
                                                compute_stage <= 7;
                                            end else begin
                                              //segments[seg_count] <= state;
                                              seg_count <= seg_count + 1;
                                              done <= 1;
                                              compute_stage <= 7;
                                           end
                                        end
                                    end else begin
                                        stateCount <= 0;
                                        compute_stage <= 0;
                                    end
                                    prev_state <= state;
                        7: // set done back to 0 and prepare for next point
                           done <= 0;
                           compute_stage <= 0;
                        default:
                           compute_stage <= 0;
                           done <= 0;
                    endcase
                end
            end

endmodule
```

Appendix VIII – Pattern Analyzer Verilog

```
//patamodule
module pattern_analyzer(
                     input wire ready, reset, clock,
                     input wire [1:0] speed_in,
                     input wire [3:0] segment,
                     output reg done,
                     output reg [1:0] speed_out,
                     output reg [6:0] pattern,
                     output wire signed [9:0] max_score);

      // parameters to describe the segments
      parameter s_horizontal_r = 4'b0000;
      parameter s_horizontal_l = 4'b1111;
      parameter s_vertical_t = 4'b0011;
      parameter s_vertical_b = 4'b1100;
      parameter s_diagonal_tr = 4'b0001;
      parameter s_diagonal_tl = 4'b0111;
      parameter s_diagonal_br = 4'b1000;
      parameter s_diagonal_bl = 4'b1110;
      parameter s_stop = 4'b0010;
      parameter s_start = 4'b0100;

      // parameters to control parameter detection
      parameter min_score = 0;
      parameter pause = 2;

      //wire signed [9:0] max_score;
      wire [6:0] max_pattern;
   wire signed [9:0] p_scores [63:0];
   wire [6:0] ps [63:0];

   reg started, find_scores, p_reset;
   reg [2:0] pause_count;
   reg signed [9:0] score;

   //patterns

   // melee 1
   pattern #(.ID(1),.LENGTH(3),
                     .pattern({s_start,s_horizontal_r,s_horizontal_l,s_stop,s_stop,s_stop,s_stop,s_
               stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
               s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_sto
               p,s_stop}))
           p1 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
                .score(p_scores[0]), .id(ps[0]));
   pattern #(.ID(1),.LENGTH(3),
                     .pattern({s_start,s_horizontal_l,s_horizontal_r,s_stop,s_stop,s_stop,s_stop,s_
               stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
               s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_sto
               p,s_stop}))
           p2 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
                .score(p_scores[1]), .id(ps[1]));
   // melee 2
   pattern #(.ID(2),.LENGTH(3),
                     .pattern({s_start,s_vertical_t,s_vertical_b,s_stop,s_stop,s_stop,s_stop,s_stop
               ,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_st
               op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
               stop}))
           p3 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
                .score(p_scores[2]), .id(ps[2]));
   pattern #(.ID(2),.LENGTH(3),
                     .pattern({s_start,s_vertical_b,s_vertical_t,s_stop,s_stop,s_stop,s_stop,s_stop
               ,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_st
               op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
               stop}))
           p4 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
                .score(p_scores[3]), .id(ps[3]));
```

```
    // ranged 1
pattern #(.ID(3),.LENGTH(3),
                .pattern({s_start,s_horizontal_r,s_vertical_t,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p5 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[4]), .id(ps[4]));
pattern #(.ID(3),.LENGTH(3),
                .pattern({s_start,s_horizontal_l,s_vertical_t,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p6 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[5]), .id(ps[5]));

    // ranged 2
pattern #(.ID(4),.LENGTH(3),
                .pattern({s_start,s_horizontal_r,s_vertical_b,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p7 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[6]), .id(ps[6]));
pattern #(.ID(4),.LENGTH(3),
                .pattern({s_start,s_horizontal_l,s_vertical_b,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p8 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[7]), .id(ps[7]));

// defense
pattern #(.ID(5),.LENGTH(3),
                .pattern({s_start,s_vertical_b,s_horizontal_r,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p9 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[8]), .id(ps[8]));
pattern #(.ID(5),.LENGTH(3),
                .pattern({s_start,s_vertical_b,s_horizontal_l,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p10 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[9]), .id(ps[9]));
// ranged 3
pattern #(.ID(6),.LENGTH(3),
                .pattern({s_start,s_vertical_t,s_horizontal_r,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p11 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[10]), .id(ps[10]));
pattern #(.ID(6),.LENGTH(3),
                .pattern({s_start,s_vertical_t,s_horizontal_l,s_stop,s_stop,s_stop,s_stop,s_st
            op,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_
            stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,s_stop,
            s_stop}))
        p12 (.ready(ready), .reset(p_reset), .clock(clock), .segment(segment),
             .score(p_scores[11]), .id(ps[11]));



max64
            bigm(.val({520'd0,p_scores[11],p_scores[10],p_scores[9],p_scores[8],p_scores[7],p_
            scores[6],p_scores[5],p_scores[4],p_scores[3],p_scores[2],p_scores[1],p_scores[0]}
            ),
```

```
            .pattern({364'd0,ps[11],ps[10],ps[9],ps[8],ps[7],ps[6],ps[5],ps[4],ps[3],ps[2],ps[1
            ],ps[0]}), .max(max_score), .max_pattern(max_pattern));


   always @(posedge clock) begin
        if (reset) begin // reset the registers
            done <= 0;
            pattern <= 0;
            speed_out <= 0;
            started <= 0;
            p_reset <= 1;
            pause_count <= 0;
        end else if (ready) begin
                    // when we recieve a segment check to see if it is the start of a gesture, if
           so reset the patterns
              // if it is the end of a gesture get the score
          if (segment == s_start || started) begin
              if (segment == s_stop) begin
                  started <= 0;
                  pause_count <= 0;
                  find_scores <= 1;
              end else begin
                  started <= 1;
                                    p_reset <= 0;
              end
          end
        end

                  // after a gesture ends wait clock cycle then output the pattern that scored the
              highest if it counts as a match
                  // it only counts as a match if the score is greater than min_score
        if (find_scores) begin
            if (pause_count == pause) begin
                done <= 1;
                speed_out <= speed_in;
                pattern <= (max_score > min_score) ? max_pattern : 0;
                find_scores <= 0;
                p_reset <= 1;
            end
            pause_count <= pause_count + 1;
        end else begin
            done <= 0;
        end

   end

endmodule

module pattern
       #(parameter ID = 7'b0000000,   // pattern identifier
                   LENGTH = 0,        // pattern length (1-32)
                   pattern = 128'd0)  // list of pattern segments
        (input wire ready, reset, clock,
                input wire [3:0] segment,
                output reg signed [9:0] score,
                output wire [6:0] id);

   // parameters to describe the segments
        parameter s_horizontal_r = 4'b0000;
        parameter s_horizontal_l = 4'b1111;
        parameter s_vertical_t = 4'b0011;
        parameter s_vertical_b = 4'b1100;
        parameter s_diagonal_tr = 4'b0001;
        parameter s_diagonal_tl = 4'b0111;
        parameter s_diagonal_br = 4'b1000;
        parameter s_diagonal_bl = 4'b1110;
        parameter s_stop = 4'b0010;
        parameter s_start = 4'b0100;

    assign id = ID;
```

```
//register to keep track of the number of segments seen
reg signed [5:0] seg_count;
wire [3:0] cur_seg, next_seg, prev_seg, last_seg;
wire signed [3:0] ones, next_ones, prev_ones, last_ones;
reg [3:0] pattern_data [31:0];

assign cur_seg = pattern_data[seg_count];
assign next_seg = pattern_data[seg_count + 1];
assign prev_seg = pattern_data[seg_count - 1];
assign last_seg = pattern_data[LENGTH - 2];

// find the hamming distance between the currently recieved segment and the current segments
          in the pattern
assign ones = ((~(cur_seg^segment))&4'b0001) + (((~(cur_seg^segment))&4'b0010)>>1) +
          (((~(cur_seg^segment))&4'b0100)>>2) + (((~(cur_seg^segment))&4'b1000)>>3);
          assign next_ones = ((~(next_seg^segment))&4'b0001) +
          (((~(next_seg^segment))&4'b0010)>>1) + (((~(next_seg^segment))&4'b0100)>>2) +
          (((~(next_seg^segment))&4'b1000)>>3);
assign prev_ones = ((~(prev_seg^segment))&4'b0001) + (((~(prev_seg^segment))&4'b0010)>>1) +
          (((~(prev_seg^segment))&4'b0100)>>2) + (((~(prev_seg^segment))&4'b1000)>>3);
assign last_ones = ((~(last_seg^segment))&4'b0001) + (((~(last_seg^segment))&4'b0010)>>1) +
          (((~(last_seg^segment))&4'b0100)>>2) + (((~(last_seg^segment))&4'b1000)>>3);


always @(posedge clock) begin
    if (reset) begin // reset the registers
        score <= 0;
        seg_count <= 1;
        pattern_data[0] = pattern[127:124];
        pattern_data[1] = pattern[123:120];
        pattern_data[2] = pattern[119:116];
        pattern_data[3] = pattern[115:112];
        pattern_data[4] = pattern[111:108];
        pattern_data[5] = pattern[107:104];
        pattern_data[6] = pattern[103:100];
        pattern_data[7] = pattern[99:96];
        pattern_data[8] = pattern[95:92];
        pattern_data[9] = pattern[91:88];
        pattern_data[10] = pattern[87:84];
        pattern_data[11] = pattern[83:80];
        pattern_data[12] = pattern[79:76];
        pattern_data[13] = pattern[75:72];
        pattern_data[14] = pattern[71:68];
        pattern_data[15] = pattern[67:64];
        pattern_data[16] = pattern[63:60];
        pattern_data[17] = pattern[59:56];
        pattern_data[18] = pattern[55:52];
        pattern_data[19] = pattern[51:48];
        pattern_data[20] = pattern[47:44];
        pattern_data[21] = pattern[43:40];
        pattern_data[22] = pattern[39:36];
        pattern_data[23] = pattern[35:32];
        pattern_data[24] = pattern[31:28];
        pattern_data[25] = pattern[27:24];
        pattern_data[26] = pattern[23:20];
        pattern_data[27] = pattern[19:16];
        pattern_data[28] = pattern[15:12];
        pattern_data[29] = pattern[11:8];
        pattern_data[30] = pattern[7:4];
        pattern_data[31] = pattern[3:0];
    end else if (ready) begin // if we get a segment then
           if (segment == s_stop) begin  // if it is an end to the gesture add length
           penalties
                   score <= score - ((seg_count > LENGTH)? seg_count - LENGTH : ((seg_count
           == LENGTH) ? -10 : LENGTH - seg_count));
             end else begin
                   if (seg_count >= LENGTH) begin // if we have recieved more segments than
           are in the pattern add length penalties
                   score <= score -((seg_count > LENGTH)? seg_count - LENGTH : LENGTH -
```

```
                seg_count);
                        end else begin // if it is the first or last segment just compare to the
                current segment
                        if (seg_count == 1 || seg_count == (LENGTH - 1)) begin
                                score <= score + (ones<<2) - 8;
                        end else begin // otherwise also look at neighboring segments
                                score <= score + ((ones > next_ones) ?

                       ((ones > prev_ones) ? ((ones<<3) - 16) : ((prev_ones<<1) - 4)) :

                       ((next_ones > prev_ones) ? ((next_ones<<1) - 4) : ((prev_ones<<1) - 4)));
                             end
                        end
                 end
                 seg_count <= seg_count + 1;
        end
    end

endmodule


// a bunch of maximizers to find the maximu score and corresponding pattern id
module max2 (input wire signed [9:0] val1, val2, input wire [13:0] pattern, output wire signed
                [9:0] max, output wire [6:0] max_pattern);
    wire bigger;
    assign bigger = val1 > val2;
    assign max = bigger ? val1 : val2;
    assign max_pattern = bigger ? pattern[13:7] : pattern[6:0];
endmodule

module max4 (input wire [39:0] val, input wire [27:0] pattern, output wire signed [9:0] max,
                output wire [6:0] max_pattern);
    wire signed [9:0] m[1:0];
    wire [6:0] p[1:0];
    max2
                ma1(.val1(val[19:10]),.val2(val[9:0]),.pattern(pattern[13:0]),.max(m[1]),.max_patt
                ern(p[1]));
    max2
                ma2(.val1(val[39:30]),.val2(val[29:20]),.pattern(pattern[27:14]),.max(m[0]),.max_p
                attern(p[0]));
    max2 ma3(.val1(m[1]),.val2(m[0]),.pattern({p[1],p[0]}),.max(max),.max_pattern(max_pattern));
endmodule

module max16 (input wire [159:0] val, input wire [111:0] pattern, output wire signed [9:0] max,
                output wire [6:0] max_pattern);
    wire signed [9:0] m[4:0];
    wire [6:0] p[4:0];
    max4 ma1(.val(val[39:0]),.max(m[0]),.pattern(pattern[27:0]),.max_pattern(p[0]));
    max4 ma2(.val(val[79:40]),.max(m[1]),.pattern(pattern[55:28]),.max_pattern(p[1]));
    max4 ma3(.val(val[119:80]),.max(m[2]),.pattern(pattern[83:56]),.max_pattern(p[2]));
    max4 ma4(.val(val[159:120]),.max(m[3]),.pattern(pattern[111:84]),.max_pattern(p[3]));
    max4
                ma5(.val({m[3],m[2],m[1],m[0]}),.max(max),.pattern({p[3],p[2],p[1],p[0]}),.max_pat
                tern(max_pattern));
endmodule

module max64 (input wire [639:0] val, input wire [447:0] pattern, output wire signed [9:0] max,
                output wire [6:0] max_pattern);
    wire signed [9:0] m[4:0];
    wire [6:0] p[4:0];
    max16 ma1(.val(val[159:0]),.max(m[0]),.pattern(pattern[111:0]),.max_pattern(p[0]));
    max16 ma2(.val(val[319:160]),.max(m[1]),.pattern(pattern[223:112]),.max_pattern(p[1]));
    max16 ma3(.val(val[479:320]),.max(m[2]),.pattern(pattern[335:224]),.max_pattern(p[2]));
    max16 ma4(.val(val[639:480]),.max(m[3]),.pattern(pattern[447:336]),.max_pattern(p[3]));
    max4
                ma5(.val({m[3],m[2],m[1],m[0]}),.max(max),.pattern({p[3],p[2],p[1],p[0]}),.max_pat
                tern(max_pattern));
endmodule
```

Appendix IX – Keyboard Module

```
//
// File:   ps2_kbd.v
// Date:   24-Oct-05
// Author: C. Terman / I. Chuang
//     Modified for this project by dgerb

//
// PS2 keyboard input for 6.111 labkit
//
// INPUTS:
//
//   clock_27mhz   - master clock
//   reset         - active high
//   clock         - ps2 interface clock
//   data          - ps2 interface data
//
// OUTPUTS:
//
//   ascii         - 8 bit ascii code for current character
//   ascii_ready   - one clock cycle pulse indicating new char received


//////////////////////////////////////////////////////////////////////////////

module ps2_ascii_input(clock_27mhz, reset, clock, data, ascii, ascii_ready, up, down, left,
right, enter);

   // module to generate ascii code for keyboard input
   // this is module works synchronously with the system clock

   input clock_27mhz;
   input reset;        // Active high asynchronous reset
   input clock;        // PS/2 clock
   input data;         // PS/2 data
   output [7:0] ascii;   // ascii code (1 character)
   output ascii_ready; // ascii ready (one clock_27mhz cycle active high)
       // dgerb added outputs for up, down, left, right, and enter signals

       // which are high if the key is pressed and low when not pressed

       output reg up, down, left, right, enter;


   reg [7:0]   ascii_val;     // internal combinatorial ascii decoded value
   reg [7:0]   lastkey;              // last keycode
   reg [7:0]   curkey;        // current keycode
   reg [7:0]   ascii;         // ascii output (latched & synchronous)
   reg        ascii_ready;    // synchronous one-cycle ready flag

   // get keycodes
   wire        fifo_rd;              // keyboard read request
   wire [7:0]  fifo_data;     // keyboard data
   wire        fifo_empty;    // flag: no keyboard data
   wire        fifo_overflow; // keyboard data overflow

   ps2 myps2(reset, clock_27mhz, clock, data, fifo_rd, fifo_data,
             fifo_empty,fifo_overflow);

   assign      fifo_rd = ~fifo_empty;       // continous read
   reg        key_ready;

   always @(posedge clock_27mhz)
     begin
       // get key if ready

       curkey <= ~fifo_empty ? fifo_data : curkey;
       lastkey <= ~fifo_empty ? curkey : lastkey;
       key_ready  <= ~fifo_empty;
```

```
    // raise ascii_ready for last key which was read

    ascii_ready <= key_ready & ~(curkey[7]|lastkey[7]);
    ascii <=  (key_ready & ~(curkey[7]|lastkey[7])) ? ascii_val : ascii;
  end



    // dgerb changed this block from logic to clocked in order to reduce glitches
always @(posedge clock_27mhz) begin //convert PS/2 keyboard make code ==> ascii code
  case (curkey)
    8'h1C: ascii_val <= 8'h41;           //A
    8'h32: ascii_val <= 8'h42;           //B
    8'h21: ascii_val <= 8'h43;           //C
    8'h23: ascii_val <= 8'h44;           //D
    8'h24: ascii_val <= 8'h45;           //E
    8'h2B: ascii_val <= 8'h46;           //F
    8'h34: ascii_val <= 8'h47;           //G
    8'h33: ascii_val <= 8'h48;           //H
    8'h43: ascii_val <= 8'h49;           //I
    8'h3B: ascii_val <= 8'h4A;           //J
    8'h42: ascii_val <= 8'h4B;           //K
    8'h4B: ascii_val <= 8'h4C;           //L
    8'h3A: ascii_val <= 8'h4D;           //M
    8'h31: ascii_val <= 8'h4E;           //N
    8'h44: ascii_val <= 8'h4F;           //O
    8'h4D: ascii_val <= 8'h50;           //P
    8'h15: ascii_val <= 8'h51;           //Q
    8'h2D: ascii_val <= 8'h52;           //R
    8'h1B: ascii_val <= 8'h53;           //S
    8'h2C: ascii_val <= 8'h54;           //T
    8'h3C: ascii_val <= 8'h55;           //U
    8'h2A: ascii_val <= 8'h56;           //V
    8'h1D: ascii_val <= 8'h57;           //W
             8'h22: ascii_val <= 8'h58;          //X
    8'h35: ascii_val <= 8'h59;           //Y
    8'h1A: ascii_val <= 8'h5A;           //Z

    8'h45: ascii_val <= 8'h30;           //0
    8'h16: ascii_val <= 8'h31;           //1
    8'h1E: ascii_val <= 8'h32;           //2
    8'h26: ascii_val <= 8'h33;           //3
    8'h25: ascii_val <= 8'h34;           //4
    8'h2E: ascii_val <= 8'h35;           //5
    8'h36: ascii_val <= 8'h36;           //6
    8'h3D: ascii_val <= 8'h37;           //7
    8'h3E: ascii_val <= 8'h38;           //8
    8'h46: ascii_val <= 8'h39;           //9

    8'h0E: ascii_val <= 8'h60;           // `
    8'h4E: ascii_val <= 8'h2D;           // -
    8'h55: ascii_val <= 8'h3D;           // =
    8'h5C: ascii_val <= 8'h5C;           // \
    8'h29: ascii_val <= 8'h20;           // (space)
    8'h54: ascii_val <= 8'h5B;           // [
    8'h5B: ascii_val <= 8'h5D;           // ]
    8'h4C: ascii_val <= 8'h3B;           // ;
    8'h52: ascii_val <= 8'h27;           // '
    8'h41: ascii_val <= 8'h2C;           // ,
    8'h49: ascii_val <= 8'h2E;           // .
    8'h4A: ascii_val <= 8'h2F;           // /

    8'h5A: begin

                ascii_val <= 8'h0D;          // enter (CR)
                // dgerb added

                // assert enter signal depending on whether or not the previous signal was
```

```
the break code

                    enter <= (lastkey == 8'hF0) ? 0 : 1;

               end

       8'h66: ascii_val <= 8'h08;           // backspace


               // dgerb added

               // assert directional signal depending on whether or not the previous signal was
the break code
               8'h75: up <= (lastkey == 8'hF0) ? 0 : 1;

               8'h6B: left <= (lastkey == 8'hF0) ? 0 : 1;

               8'h72: down <= (lastkey == 8'hF0) ? 0 : 1;

               8'h74: right <= (lastkey == 8'hF0) ? 0 : 1;


       8'hF0: ascii_val <= 8'hF0;           // BREAK CODE


       default: ascii_val <= 8'h23;         // #
     endcase

   end
endmodule // ps2toascii

//////////////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clock_27mhz, ps2c, ps2d, fifo_rd, fifo_data,
           fifo_empty,fifo_overflow);

   input clock_27mhz,reset;
   input ps2c;                // ps2 clock
   input ps2d;                // ps2 data
   input fifo_rd;             // fifo read request (active high)
   output [7:0] fifo_data;    // fifo data output
   output      fifo_empty;    // fifo empty (active high)
   output      fifo_overflow; // fifo overflow - too much kbd input

  reg [3:0] count;      // count incoming data bits
  reg [9:0] shift;      // accumulate incoming data bits

  reg [7:0] fifo[7:0];   // 8 element data fifo
  reg fifo_overflow;
  reg [2:0] wptr,rptr;   // fifo write and read pointers

  wire [2:0] wptr_inc = wptr + 1;

  assign fifo_empty = (wptr == rptr);
  assign fifo_data = fifo[rptr];

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clock_27mhz) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire sample = ps2c_sync[2] & ~ps2c_sync[1];

  always @ (posedge clock_27mhz) begin
    if (reset) begin
      count <= 0;
      wptr <= 0;
      rptr <= 0;
      fifo_overflow <= 0;
    end
    else if (sample) begin
```

```
       // order of arrival: 0,8 bits of data (LSB first),odd parity,1
       if (count==10) begin
          // just received what should be the stop bit
          if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
            fifo[wptr] <= shift[8:1];
            wptr <= wptr_inc;
            fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
          end
          count <= 0;
       end else begin
          shift <= {ps2d,shift[9:1]};
          count <= count + 1;
       end
     end
// bump read pointer if we're done with current value.
// Read also resets the overflow indicator
if (fifo_rd && !fifo_empty) begin
  rptr <= rptr + 1;
  fifo_overflow <= 0;
end
```

Appendix X – Game Logic Verilog

```
//glmodule
module game_logic
               #(parameter SPRITES = 26,
                           COLLISIONS = 8,
                           INIT_SPRITES = 26'b0,
                           INIT_ORIENT = 26'b0,
                           INIT_POS_X = 260'b0,
                           INIT_POS_Y = 260'b0,
                           INIT_DATA = 104'b0,
                           SPRITE_WIDTH = 208'b0,
                           SPRITE_HEIGHT = 208'b0,
                           KEYS = 8,
                           ATTACKS = 5)
               (input wire reset, clock, green_gesture_ready, red_gesture_ready, vsync,
                                   input [KEYS-1:0] keyboard,
                                   // 1 bit for new value and 8 bits for key
                                   input [6:0] green_gesture, red_gesture,
                                   input [1:0] green_speed, red_speed,
                                   input [COLLISIONS-1:0] collisions,
                                   // 1 bit for each possible collision we care about
                                   indicating one happened
                                   input [9:0] green_x, green_y, red_x, red_y,
                                   input enable_fatal, pause,
                                   input [6:0] buttons,
                  output reg [SPRITES-1:0] sprite_display, sprite_orientation,
                  // orientation is 1 for right and 0 for left (facing)
                  output reg [SPRITES*10-1:0] sprite_x, sprite_y,
                  output reg [SPRITES*4-1:0] sprite_data,
                  output reg signed [8:0] kirby_health, dedede_health);

   // level and general parameters
        parameter start_health = 100;
        parameter fatal_cutoff = 10;
        parameter collide_count = 20;

        parameter gravity = 2;
        parameter g_frames = 30;

        parameter p_h_move = 3;
        parameter p_v_move = 3;

        parameter p_frames = 2;

        parameter p1_xmax = 432;
        parameter p1_xmin = 10;
        parameter p3_ymin = 150;
        parameter p3_ymax = 550;

        reg [5:0] g_count, p_count;

   // movement parameters
        parameter k_jump_impulse = -5;
        parameter k_flap_impulse  = -3;
        parameter k_move_vel = 5;
        parameter k_move_vel_air = 2;
        parameter k_float_max = 3;

        parameter d_jump_impulse = -5;
        parameter d_flap_impulse  = -3;
        parameter d_move_vel = 5;
        parameter d_move_vel_air = 2;
        parameter d_float_max = 3;

        parameter v_collide = 0;

   //attack parameters
        parameter melee_gesture_1 = 1;
```

```
parameter melee_gesture_2 = 2;
parameter ranged_gesture_1 = 3;
parameter ranged_gesture_2 = 4;
parameter ranged_gesture_3 = 5;
parameter defense_gesture = 6;

parameter k_ammo_x_offset = 30;
parameter k_ammo_y_offset = 0;
parameter k_melee_x_offset_1 = -20;
parameter k_melee_y_offset_1 = -15;
parameter k_melee_x_offset_2 = -30;
parameter k_melee_y_offset_2 = -20;
parameter k_base_shot_vel = 15;
parameter k_bonus_vel = 5;
parameter k_lobbed_x_vel = 3;
parameter k_lobbed_y_vel = -5;
parameter k_bomb_y_vel = 5;
parameter k_attack_dmg = {7'd15, 7'd100, 7'd20, 7'd20, 7'd10, 7'd3};
       // ranged2, fatality, melee1, melee2, ranged, defense
parameter k_attack_hit_time = {10'd10, 10'd100, 10'd30, 10'd30, 10'd10, 10'd0};
parameter k_attack_x_speed = {5'd2, 5'd0, 5'd5, 5'd8, 5'd5, 5'd0};
parameter k_attack_y_speed = {5'd0, 5'd0, 5'd3, 5'd3, 5'd3, 5'd0};
parameter k_move_time = {10'd20, 10'd100, 10'd75, 10'd75, 10'd20, 10'd100};
parameter k_ammo_life_1 = 500;
parameter k_ammo_life_2 = 500;
parameter k_ranged_cooldown_1 = 600;
parameter k_ranged_cooldown_2 = 600;

parameter d_ammo_x_offset = 0;
parameter d_ammo_y_offset = 0;
parameter d_melee_x_offset_1 = -20;
parameter d_melee_y_offset_1 = -20;
parameter d_melee_x_offset_2 = -30;
parameter d_melee_y_offset_2 = -20;
parameter d_base_shot_vel = 15;
parameter d_bonus_vel = 5;
parameter d_lobbed_x_vel = 3;
parameter d_lobbed_y_vel = -5;
parameter d_bomb_y_vel = 5;
parameter d_attack_dmg = {7'd15, 7'd100, 7'd20, 7'd20, 7'd10, 7'd3};
       // ranged2, fatality, melee1, melee2, ranged, defense
parameter d_attack_hit_time = {10'd10, 10'd100, 10'd30, 10'd30, 10'd10, 10'd0};
parameter d_attack_x_speed = {5'd2, 5'd0, 5'd5, 5'd8, 5'd5, 5'd0};
parameter d_attack_y_speed = {5'd0, 5'd0, 5'd3, 5'd3, 5'd3, 5'd0};
parameter d_move_time = {10'd20, 10'd100, 10'd75, 10'd75, 10'd20, 10'd100};
parameter d_ammo_life_1 = 500;
parameter d_ammo_life_2 = 500;
parameter d_ranged_cooldown_1 = 600;
parameter d_ranged_cooldown_2 = 600;

// animation parameters
parameter blood_life = 15;
parameter blood_life_2 = 45;
parameter blood_count = 5;

parameter k_walk_frame_count = 10;
parameter k_melee_count_1 = 25;
parameter k_melee_count_2 = 25;
parameter k_ranged_count_1 = 10;
parameter k_ranged_count_2 = 10;
parameter k_blood_offset_x = 10;
parameter k_blood_offset_y = -10;
parameter k_blood_offset_x_1 = 20;
parameter k_blood_offset_y_1 = 10;
parameter k_blood_offset_x_2 = -30;
parameter k_blood_offset_y_2 = -30;


parameter d_walk_frame_count = 10;
parameter d_melee_count_1 = 25;
```

```
    parameter d_melee_count_2 = 25;
    parameter d_ranged_count_1 = 10;
    parameter d_ranged_count_2 = 10;
    parameter d_blood_offset_x = 10;
    parameter d_blood_offset_y = -10;
    parameter d_blood_offset_x_1 = 20;
    parameter d_blood_offset_y_1 = 10;
    parameter d_blood_offset_x_2 = -30;
    parameter d_blood_offset_y_2 = -30;

//sprite params
    parameter KIRBY = 0;
    parameter DEDEDE = 1;
    parameter SWORD = 2;
    parameter HAMMER = 3;
    parameter PLATFORM1 = 4;
    parameter PLATFORM2 = 5;
    parameter PLATFORM3 = 6;
    parameter PLATFORM4 = 7;
    parameter PLATFORM5 = 8;
    parameter PLATFORM6 = 9;
    parameter HEALTH1 = 10;
    parameter HEALTH2 = 11;
    parameter KAMMO1 = 12;
    parameter KAMMO2 = 13;
    parameter DAMMO1 = 14;
    parameter DAMMO2 = 15;
    parameter BLOOD1 = 16;
    parameter BLOOD2 = 17;
    parameter BLOOD3 = 18;
    parameter BLOOD4 = 19;
    parameter BLOOD5 = 20;
    parameter BLOOD6 = 21;
    parameter BLOOD7 = 22;
    parameter BLOOD8 = 23;
    parameter HURT1 = 24;
    parameter HURT2 = 25;

// collision params
    parameter KIRBY_HAMMER = 0;
    parameter DEDEDE_SWORD = 1;
    parameter KIRBY_DAMMO1 = 2;
    parameter KIRBY_DAMMO2 = 3;
    parameter DEDEDE_KAMMO1 = 4;
    parameter DEDEDE_KAMMO2 = 5;
    parameter HAMMER_SWORD = 6;
    parameter KIRBY_DEDEDE = 7;

//state params
    parameter walk_left = 2'b01;
    parameter walk_right = 2'b10;
    parameter sit_still = 2'b00;
    parameter float = 2'b01;
    parameter fall = 2'b10;
    parameter attack = 2'b01;
    parameter defend = 2'b10;

// level parameters
    parameter bottom = 650;
    parameter blood_offset = 150;
    parameter top = 0;
    parameter posbits = 10;


// internal registers for sprite data
reg signed [5:0] sprite_vel_x[SPRITES-1:0], sprite_vel_y[SPRITES-1:0];

// registers for kirby and dedede's states
reg [1:0] k_h_move, d_h_move, k_v_move, d_v_move, k_do, d_do;
reg [3:0] k_frame, d_frame, k_prev_frame, d_prev_frame;
```

```
reg [7:0] k_do_timer, d_do_timer, k_hit_timer, d_hit_timer, k_collide_counter,
          d_collide_counter, k_frame_count, d_frame_count;
reg [1:0] k_ammo_active, d_ammo_active;
reg [4:0] k_a_move, d_a_move, k_d_move, d_d_move;
reg k_flapped,d_flapped,v_move,h_move;
reg [2:0] k_up_count, d_up_count;
reg [7:0] k_ammo_time[1:0], d_ammo_time[1:0], blood_counter[1:0], b_frame_count[1:0],
          k_ammo_frame_count[1:0],  d_ammo_frame_count[1:0];
reg [12:0] k_cooldown[1:0], d_cooldown[1:0];

//keyboard setup
wire k_up = keyboard[0];
wire k_down = keyboard[1];
wire k_left = keyboard[2];
wire k_right = keyboard[3];
wire d_up = keyboard[4];
wire d_down = keyboard[5];
wire d_left = keyboard[6];
wire d_right = keyboard[7];

// gesture registers so that we hold incoming gestures for a frame to correctly respond to
          them
reg [6:0] f_green_gesture, f_red_gesture;
reg [1:0] f_green_speed, f_red_speed;
reg f_green_ready, f_red_ready;

reg [2:0] green_count, red_count;
    reg prev_vsync;
// grab the gesture
always @(posedge clock) begin
    if (reset) begin // reset gesture related registers
        f_green_gesture <= 0;
        f_red_gesture <= 0;
        f_green_speed <= 0;
        f_red_speed <= 0;
        f_green_ready <= 0;
        f_red_ready <= 0;
        green_count <= 0;
        red_count <= 0;
             prev_vsync <= vsync;
         end else begin
            if (green_gesture_ready) begin // if there is an incoming gesture store it in
            registers
                   f_green_gesture <= green_gesture;
                   f_green_speed <= green_speed;
                   f_green_ready <= 1;
                   green_count <= 0;
            end else if (f_green_ready && prev_vsync && !vsync) begin
                   // if the frame changes increase the count of how long we have been holding
            the gesutre
                   green_count <= green_count + 1;
                   if (green_count == 5) begin
                          // when we have held gesture for 5 frames stop
                          green_count <= 0;
                          f_green_ready <= 0;
                          f_green_gesture <= 0;
                   end
            end

            if (red_gesture_ready) begin
                   f_red_gesture <= red_gesture;
                   f_red_speed <= red_speed;
                   f_red_ready <= 1;
                   red_count <= 0;
            end else if (f_red_ready && prev_vsync && !vsync) begin
                   red_count <= red_count + 1;
                   if (red_count == 5) begin
                          red_count <= 0;
                          f_red_ready <= 0;
                          f_red_gesture <= 0;
```

```
                end
            end
            prev_vsync <= vsync;
             end
end

// combinational logic for actions
wire k_attack, k_defend, d_attack, d_defend;
assign k_attack = f_red_ready && (f_red_gesture <= ATTACKS) && f_red_gesture != 0;
assign k_defend = f_red_ready && (f_red_gesture > ATTACKS);
assign d_attack = f_green_ready && (f_green_gesture <= ATTACKS) && f_green_gesture != 0;
assign d_defend = f_green_ready && (f_green_gesture > ATTACKS);

// combinational logic to detect floor and platform collisions
    wire [6:0] d_on_floor, k_on_floor;
    wire [7:0] blood_on_floor;

    assign blood_on_floor = {((sprite_y[BLOOD8*posbits + posbits - 1 : BLOOD8 * posbits] +
           {2'b00,SPRITE_HEIGHT[BLOOD8*8 + 7: BLOOD8 * 8]} + {4'b0000,sprite_vel_y[BLOOD8]})
           > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD7*posbits +
           posbits - 1 : BLOOD7 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD7*8 + 7: BLOOD7 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD7]}) > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD6*posbits +
           posbits - 1 : BLOOD6 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD6*8 + 7: BLOOD6 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD6]}) > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD5*posbits +
           posbits - 1 : BLOOD5 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD5*8 + 7: BLOOD5 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD5]}) > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD4*posbits +
           posbits - 1 : BLOOD4 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD4*8 + 7: BLOOD4 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD4]}) > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD3*posbits +
           posbits - 1 : BLOOD3 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD3*8 + 7: BLOOD3 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD3]}) > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD2*posbits +
           posbits - 1 : BLOOD2 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD2*8 + 7: BLOOD2 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD2]}) > (bottom + blood_offset)),
                                                    ((sprite_y[BLOOD1*posbits +
           posbits - 1 : BLOOD1 * posbits] + {2'b00,SPRITE_HEIGHT[BLOOD1*8 + 7: BLOOD1 * 8]}
           + {4'b0000,sprite_vel_y[BLOOD1]}) > (bottom + blood_offset))};

    // a string of bits which are 1 if Kirby is colliding with certain platforms or the floor
    // uses simple bounding boxes
    assign k_on_floor = {(((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
           {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) <
           (sprite_y[PLATFORM1*posbits + posbits - 1 : PLATFORM1 * posbits] +
           {2'b00,SPRITE_HEIGHT[PLATFORM1*8 + 7: PLATFORM1 * 8]})) &&
           ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
           {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) >=
           (sprite_y[PLATFORM1*posbits + posbits - 1 : PLATFORM1 * posbits])) &&
           ((sprite_x[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
           {2'b00,SPRITE_WIDTH[KIRBY*8 + 7: KIRBY * 8]}) > sprite_x[PLATFORM1*posbits +
           posbits - 1 : PLATFORM1 * posbits]) && ((sprite_x[KIRBY*posbits + posbits - 1 :
           KIRBY * posbits]) < (sprite_x[PLATFORM1*posbits + posbits - 1 : PLATFORM1 *
           posbits] + {2'b00,SPRITE_WIDTH[PLATFORM1*8 + 7: PLATFORM1 * 8]}))),
                                    (((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY *
           posbits] + {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} +
           {4'b0000,sprite_vel_y[KIRBY]}) < (sprite_y[PLATFORM2*posbits + posbits - 1 :
           PLATFORM2 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM2*8 + 7: PLATFORM2 * 8]})) &&
           ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
           {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) >=
           (sprite_y[PLATFORM2*posbits + posbits - 1 : PLATFORM2 * posbits])) &&
           ((sprite_x[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
           {2'b00,SPRITE_WIDTH[KIRBY*8 + 7: KIRBY * 8]}) > sprite_x[PLATFORM2*posbits +
           posbits - 1 : PLATFORM2 * posbits]) && ((sprite_x[KIRBY*posbits + posbits - 1 :
           KIRBY * posbits]) < (sprite_x[PLATFORM2*posbits + posbits - 1 : PLATFORM2 *
           posbits] + {2'b00,SPRITE_WIDTH[PLATFORM2*8 + 7: PLATFORM2 * 8]}))),
                                    (((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY *
           posbits] + {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} +
```

```
        {4'b0000,sprite_vel_y[KIRBY]}) < (sprite_y[PLATFORM3*posbits + posbits - 1 :
        PLATFORM3 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM3*8 + 7: PLATFORM3 * 8]})) &&
        ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) >=
        (sprite_y[PLATFORM3*posbits + posbits - 1 : PLATFORM3 * posbits])) &&
        ((sprite_x[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_WIDTH[KIRBY*8 + 7: KIRBY * 8]}) > sprite_x[PLATFORM3*posbits +
        posbits - 1 : PLATFORM3 * posbits]) && ((sprite_x[KIRBY*posbits + posbits - 1 :
        KIRBY * posbits]) < (sprite_x[PLATFORM3*posbits + posbits - 1 : PLATFORM3 *
        posbits] + {2'b00,SPRITE_WIDTH[PLATFORM3*8 + 7: PLATFORM3 * 8]}))),
                                    (((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY *
        posbits] + {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} +
        {4'b0000,sprite_vel_y[KIRBY]}) < (sprite_y[PLATFORM4*posbits + posbits - 1 :
        PLATFORM4 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM4*8 + 7: PLATFORM4 * 8]})) &&
        ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) >=
        (sprite_y[PLATFORM4*posbits + posbits - 1 : PLATFORM4 * posbits])) &&
        ((sprite_x[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_WIDTH[KIRBY*8 + 7: KIRBY * 8]}) > sprite_x[PLATFORM4*posbits +
        posbits - 1 : PLATFORM4 * posbits]) && ((sprite_x[KIRBY*posbits + posbits - 1 :
        KIRBY * posbits]) < (sprite_x[PLATFORM4*posbits + posbits - 1 : PLATFORM4 *
        posbits] + {2'b00,SPRITE_WIDTH[PLATFORM4*8 + 7: PLATFORM4 * 8]}))),
                                    (((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY *
        posbits] + {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} +
        {4'b0000,sprite_vel_y[KIRBY]}) < (sprite_y[PLATFORM5*posbits + posbits - 1 :
        PLATFORM5 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM5*8 + 7: PLATFORM5 * 8]})) &&
        ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) >=
        (sprite_y[PLATFORM5*posbits + posbits - 1 : PLATFORM5 * posbits])) &&
        ((sprite_x[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_WIDTH[KIRBY*8 + 7: KIRBY * 8]}) > sprite_x[PLATFORM5*posbits +
        posbits - 1 : PLATFORM5 * posbits]) && ((sprite_x[KIRBY*posbits + posbits - 1 :
        KIRBY * posbits]) < (sprite_x[PLATFORM5*posbits + posbits - 1 : PLATFORM5 *
        posbits] + {2'b00,SPRITE_WIDTH[PLATFORM5*8 + 7: PLATFORM5 * 8]}))),
                                    (((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY *
        posbits] + {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} +
        {4'b0000,sprite_vel_y[KIRBY]}) < (sprite_y[PLATFORM6*posbits + posbits - 1 :
        PLATFORM6 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM6*8 + 7: PLATFORM6 * 8]})) &&
        ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} + {4'b0000,sprite_vel_y[KIRBY]}) >=
        (sprite_y[PLATFORM6*posbits + posbits - 1 : PLATFORM6 * posbits])) &&
        ((sprite_x[KIRBY*posbits + posbits - 1 : KIRBY * posbits] +
        {2'b00,SPRITE_WIDTH[KIRBY*8 + 7: KIRBY * 8]}) > sprite_x[PLATFORM6*posbits +
        posbits - 1 : PLATFORM6 * posbits]) && ((sprite_x[KIRBY*posbits + posbits - 1 :
        KIRBY * posbits]) < (sprite_x[PLATFORM6*posbits + posbits - 1 : PLATFORM6 *
        posbits] + {2'b00,SPRITE_WIDTH[PLATFORM6*8 + 7: PLATFORM6 * 8]}))),
                                    ((sprite_y[KIRBY*posbits + posbits - 1 : KIRBY *
        posbits] + {2'b00,SPRITE_HEIGHT[KIRBY*8 + 7: KIRBY * 8]} +
        {4'b0000,sprite_vel_y[KIRBY]}) > (bottom - v_collide))};


assign d_on_floor = {(((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
        {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
        < (sprite_y[PLATFORM1*posbits + posbits - 1 : PLATFORM1 * posbits] +
        {2'b00,SPRITE_HEIGHT[PLATFORM1*8 + 7: PLATFORM1 * 8]})) &&
        ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
        {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
        >= (sprite_y[PLATFORM1*posbits + posbits - 1 : PLATFORM1 * posbits])) &&
        ((sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
        {2'b00,SPRITE_WIDTH[DEDEDE*8 + 7: DEDEDE * 8]}) > sprite_x[PLATFORM1*posbits +
        posbits - 1 : PLATFORM1 * posbits]) && ((sprite_x[DEDEDE*posbits + posbits - 1 :
        DEDEDE * posbits]) < (sprite_x[PLATFORM1*posbits + posbits - 1 : PLATFORM1 *
        posbits] + {2'b00,SPRITE_WIDTH[PLATFORM1*8 + 7: PLATFORM1 * 8]}))),
                                    (((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE *
        posbits] + {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} +
        {4'b0000,sprite_vel_y[DEDEDE]}) < (sprite_y[PLATFORM2*posbits + posbits - 1 :
        PLATFORM2 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM2*8 + 7: PLATFORM2 * 8]})) &&
        ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
        {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
        >= (sprite_y[PLATFORM2*posbits + posbits - 1 : PLATFORM2 * posbits])) &&
```

```
            ((sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_WIDTH[DEDEDE*8 + 7: DEDEDE * 8]}) > sprite_x[PLATFORM2*posbits +
            posbits - 1 : PLATFORM2 * posbits]) && ((sprite_x[DEDEDE*posbits + posbits - 1 :
            DEDEDE * posbits]) < (sprite_x[PLATFORM2*posbits + posbits - 1 : PLATFORM2 *
            posbits] + {2'b00,SPRITE_WIDTH[PLATFORM2*8 + 7: PLATFORM2 * 8]}))),
                                    (((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE *
            posbits] + {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} +
            {4'b0000,sprite_vel_y[DEDEDE]}) < (sprite_y[PLATFORM3*posbits + posbits - 1 :
            PLATFORM3 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM3*8 + 7: PLATFORM3 * 8]})) &&
            ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
            >= (sprite_y[PLATFORM3*posbits + posbits - 1 : PLATFORM3 * posbits])) &&
            ((sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_WIDTH[DEDEDE*8 + 7: DEDEDE * 8]}) > sprite_x[PLATFORM3*posbits +
            posbits - 1 : PLATFORM3 * posbits]) && ((sprite_x[DEDEDE*posbits + posbits - 1 :
            DEDEDE * posbits]) < (sprite_x[PLATFORM3*posbits + posbits - 1 : PLATFORM3 *
            posbits] + {2'b00,SPRITE_WIDTH[PLATFORM3*8 + 7: PLATFORM3 * 8]}))),
                                    (((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE *
            posbits] + {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} +
            {4'b0000,sprite_vel_y[DEDEDE]}) < (sprite_y[PLATFORM4*posbits + posbits - 1 :
            PLATFORM4 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM4*8 + 7: PLATFORM4 * 8]})) &&
            ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
            >= (sprite_y[PLATFORM4*posbits + posbits - 1 : PLATFORM4 * posbits])) &&
            ((sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_WIDTH[DEDEDE*8 + 7: DEDEDE * 8]}) > sprite_x[PLATFORM4*posbits +
            posbits - 1 : PLATFORM4 * posbits]) && ((sprite_x[DEDEDE*posbits + posbits - 1 :
            DEDEDE * posbits]) < (sprite_x[PLATFORM4*posbits + posbits - 1 : PLATFORM4 *
            posbits] + {2'b00,SPRITE_WIDTH[PLATFORM4*8 + 7: PLATFORM4 * 8]}))),
                                    (((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE *
            posbits] + {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} +
            {4'b0000,sprite_vel_y[DEDEDE]}) < (sprite_y[PLATFORM5*posbits + posbits - 1 :
            PLATFORM5 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM5*8 + 7: PLATFORM5 * 8]})) &&
            ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
            >= (sprite_y[PLATFORM5*posbits + posbits - 1 : PLATFORM5 * posbits])) &&
            ((sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_WIDTH[DEDEDE*8 + 7: DEDEDE * 8]}) > sprite_x[PLATFORM5*posbits +
            posbits - 1 : PLATFORM5 * posbits]) && ((sprite_x[DEDEDE*posbits + posbits - 1 :
            DEDEDE * posbits]) < (sprite_x[PLATFORM5*posbits + posbits - 1 : PLATFORM5 *
            posbits] + {2'b00,SPRITE_WIDTH[PLATFORM5*8 + 7: PLATFORM5 * 8]}))),
                                    (((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE *
            posbits] + {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} +
            {4'b0000,sprite_vel_y[DEDEDE]}) < (sprite_y[PLATFORM6*posbits + posbits - 1 :
            PLATFORM6 * posbits] + {2'b00,SPRITE_HEIGHT[PLATFORM6*8 + 7: PLATFORM6 * 8]})) &&
            ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} + {4'b0000,sprite_vel_y[DEDEDE]})
            >= (sprite_y[PLATFORM6*posbits + posbits - 1 : PLATFORM6 * posbits])) &&
            ((sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE * posbits] +
            {2'b00,SPRITE_WIDTH[DEDEDE*8 + 7: DEDEDE * 8]}) > sprite_x[PLATFORM6*posbits +
            posbits - 1 : PLATFORM6 * posbits]) && ((sprite_x[DEDEDE*posbits + posbits - 1 :
            DEDEDE * posbits]) < (sprite_x[PLATFORM6*posbits + posbits - 1 : PLATFORM6 *
            posbits] + {2'b00,SPRITE_WIDTH[PLATFORM6*8 + 7: PLATFORM6 * 8]}))),
                                    ((sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE *
            posbits] + {2'b00,SPRITE_HEIGHT[DEDEDE*8 + 7: DEDEDE * 8]} +
            {4'b0000,sprite_vel_y[DEDEDE]}) > (bottom - v_collide))};

// combinational logic to detect when fatalaties can be done
    wire k_fatal = dedede_health <= fatal_cutoff && collisions[KIRBY_DEDEDE];
    wire d_fatal = kirby_health <= fatal_cutoff && collisions[KIRBY_DEDEDE];

    // register to control fatality animations
    reg fataled;

// used in for loops
integer i;

// calculate the next frame while the current one is being displayed
always @(posedge vsync) begin
    if (reset) begin
```

```
      // reset all the registers
sprite_display <= INIT_SPRITES;
      for(i = 0; i < SPRITES; i=i+1) begin
                  sprite_vel_x[i] <= 6'b0;
                  sprite_vel_y[i] <= 6'b0;
      end
sprite_x <= INIT_POS_X;
sprite_y <= INIT_POS_Y;
sprite_data <= INIT_DATA;
sprite_orientation <= INIT_ORIENT;

      v_move <= 1;
      h_move <= 1;

k_h_move <= 0;
d_h_move <= 0;
k_v_move <= 0;
d_v_move <= 0;
k_do <= 0;
d_do <= 0;

      k_a_move <= 0;
      k_d_move <= 0;
      d_a_move <= 0;
      d_d_move <= 0;

k_do_timer <= 0;
d_do_timer <= 0;
k_hit_timer <= 0;
d_hit_timer <= 0;

k_flapped <= 0;
d_flapped <= 0;

kirby_health <= start_health;
dedede_health <= start_health;

      k_frame_count <= 0;
      d_frame_count <= 0;
      k_ammo_frame_count[1] <= 0;
      k_ammo_frame_count[0] <= 0;
      d_ammo_frame_count[1] <= 0;
      d_ammo_frame_count[0] <= 0;
k_frame <= 0;
d_frame <= 0;

g_count <= 0;

k_ammo_time[0] <= 0;
k_ammo_time[1] <= 0;
d_ammo_time[0] <= 0;
d_ammo_time[1] <= 0;

      k_up_count <= 0;
      d_up_count <= 0;

      blood_counter[1] <= 0;
      blood_counter[0] <= 0;
      b_frame_count[1] <= 0;
      b_frame_count[0] <= 0;

      k_ammo_active <= 0;
      d_ammo_active <= 0;

      fataled <= 0;

      k_cooldown[1] <= 0;
      k_cooldown[0] <= 0;
      d_cooldown[1] <= 0;
      d_cooldown[0] <= 0;
```

```
        // easter egg buttons
 end else if(|buttons[4:0]) begin
          if (buttons[0]) begin
                      kirby_health <= 5;
          end else if (buttons[1]) begin
                      kirby_health <= 100;
          end else if (buttons[2]) begin
                      dedede_health <= 5;
          end else if (buttons[3]) begin
                      dedede_health <= 100;
          end else if (buttons[4]) begin
                      fataled <= 0;
          end
  // normal gameplay - both characters still alive and game not paused
end else if(dedede_health > 0 && kirby_health > 0 && !pause) begin

     // --------------- decrement timers used in the game

          // decrement blood animation frame counters
          b_frame_count[0] <= (b_frame_count[0] == 0) ? 0 : b_frame_count[0] - 1;
          b_frame_count[1] <= (b_frame_count[1] == 0) ? 0 : b_frame_count[1] - 1;

          // decrement weapon cooldown counters
          k_cooldown[0] <= (k_cooldown[0] == 0) ? 0 : k_cooldown[0] - 1;
          k_cooldown[1] <= (k_cooldown[1] == 0) ? 0 : k_cooldown[1] - 1;
          d_cooldown[0] <= (d_cooldown[0] == 0) ? 0 : d_cooldown[0] - 1;
          d_cooldown[1] <= (d_cooldown[1] == 0) ? 0 : d_cooldown[1] - 1;

          // decrement animation timer, collision timer (so we don't count same collision
        multiple times)
          //           action timer (can't perform another attack until 1st completes),
        and
          //                          hit timer (can't move if just got hit)
     k_hit_timer <= (k_hit_timer == 0) ? 0 : k_hit_timer - 1;
     k_do_timer <= (k_do_timer == 0) ? 0 : k_do_timer - 1;
          k_collide_counter <= (k_collide_counter == 0) ? 0 : k_collide_counter - 1;
          k_frame_count <= (k_frame_count == 0) ? 0 : k_frame_count - 1;

     d_hit_timer <= (d_hit_timer == 0) ? 0 : d_hit_timer - 1;
     d_do_timer <= (d_do_timer == 0) ? 0 : d_do_timer - 1;
          d_collide_counter <= (d_collide_counter == 0) ? 0 : d_collide_counter - 1;
          d_frame_count <= (d_frame_count == 0) ? 0 : d_frame_count - 1;

     // decrement gravity counter (accelleration doesn't happen every frame because without
        //floating point it would go to fast otherwise)
     g_count <= (g_count == (g_frames + 1)) ? 0 : g_count + 1;

          // perform gravitational accelleration on all affected sprites every g_frames,
        // but don't accelerate if on floor
     if (g_count == g_frames) begin
         sprite_vel_y[KIRBY] <= (k_v_move == sit_still) ? 0 : (sprite_vel_y[KIRBY] +
         (k_v_move == float ? gravity>>1 : gravity<<1));
         sprite_vel_y[DEDEDE] <= (d_v_move == sit_still) ? 0 : (sprite_vel_y[DEDEDE] +
         (d_v_move == float ? gravity>>1 : gravity<<1));

                      sprite_vel_y[BLOOD1] <= blood_on_floor[0] ? 0 :
         sprite_vel_y[BLOOD1] + gravity;
                      sprite_vel_y[BLOOD2] <= blood_on_floor[1] ? 0 :
         sprite_vel_y[BLOOD2] + gravity;
                      sprite_vel_y[BLOOD3] <= blood_on_floor[2] ? 0 :
         sprite_vel_y[BLOOD3] + gravity;
                      sprite_vel_y[BLOOD4] <= blood_on_floor[3] ? 0 :
         sprite_vel_y[BLOOD4] + gravity;
                      sprite_vel_y[BLOOD5] <= blood_on_floor[4] ? 0 :
         sprite_vel_y[BLOOD5] + gravity;
                      sprite_vel_y[BLOOD6] <= blood_on_floor[5] ? 0 :
         sprite_vel_y[BLOOD6] + gravity;
                      sprite_vel_y[BLOOD7] <= blood_on_floor[6] ? 0 :
         sprite_vel_y[BLOOD7] + gravity;
```

```
                sprite_vel_y[BLOOD8] <= blood_on_floor[7] ? 0 :
    sprite_vel_y[BLOOD8] + gravity;

                sprite_vel_y[KAMMO2] <= sprite_y[KAMMO2] >=  bottom ? 0 :
    sprite_vel_y[KAMMO2] + gravity;
                sprite_vel_y[DAMMO2] <= sprite_y[DAMMO2] >=  bottom ? 0 :
    sprite_vel_y[DAMMO2] + gravity;
end

      // stop blood sprites and make them invisible when they hit the floor
      if (blood_on_floor[0]) begin
                sprite_vel_y[BLOOD1] <= 0;
                sprite_display[BLOOD1] <= 0;
      end if (blood_on_floor[1]) begin
                sprite_vel_y[BLOOD2] <= 0;
                sprite_display[BLOOD2] <= 0;
      end if (blood_on_floor[2]) begin
                sprite_vel_y[BLOOD3] <= 0;
                sprite_display[BLOOD3] <= 0;
      end if (blood_on_floor[3]) begin
                sprite_vel_y[BLOOD4] <= 0;
                sprite_display[BLOOD4] <= 0;
      end if (blood_on_floor[4]) begin
                sprite_vel_y[BLOOD5] <= 0;
                sprite_display[BLOOD5] <= 0;
      end if (blood_on_floor[5]) begin
                sprite_vel_y[BLOOD6] <= 0;
                sprite_display[BLOOD6] <= 0;
      end if (blood_on_floor[6]) begin
                sprite_vel_y[BLOOD7] <= 0;
                sprite_display[BLOOD7] <= 0;
      end if (blood_on_floor[7]) begin
                sprite_vel_y[BLOOD8] <= 0;
                sprite_display[BLOOD8] <= 0;
      end

      // move the blood sprites based on their current velocity
            sprite_x[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] <=
    sprite_x[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] + sprite_vel_x[BLOOD1];
            sprite_y[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] <=
    sprite_y[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] + sprite_vel_y[BLOOD1];
            sprite_x[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] <=
    sprite_x[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] + sprite_vel_x[BLOOD2];
            sprite_y[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] <=
    sprite_y[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] + sprite_vel_y[BLOOD2];
            sprite_x[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] <=
    sprite_x[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] + sprite_vel_x[BLOOD3];
            sprite_y[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] <=
    sprite_y[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] + sprite_vel_y[BLOOD3];
            sprite_x[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] <=
    sprite_x[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] + sprite_vel_x[BLOOD4];
            sprite_y[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] <=
    sprite_y[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] + sprite_vel_y[BLOOD4];
            sprite_x[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] <=
    sprite_x[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] + sprite_vel_x[BLOOD5];
            sprite_y[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] <=
    sprite_y[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] + sprite_vel_y[BLOOD5];
            sprite_x[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] <=
    sprite_x[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] + sprite_vel_x[BLOOD6];
            sprite_y[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] <=
    sprite_y[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] + sprite_vel_y[BLOOD6];
            sprite_x[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] <=
    sprite_x[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] + sprite_vel_x[BLOOD7];
            sprite_y[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] <=
    sprite_y[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] + sprite_vel_y[BLOOD7];
            sprite_x[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] <=
    sprite_x[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] + sprite_vel_x[BLOOD8];
            sprite_y[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] <=
    sprite_y[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] + sprite_vel_y[BLOOD8];
```

```
    // ------------------------------------------------------------------------------
    // ---------------------------------------- KIRBY ------------------------------
          // ------------------------------------------------------------------------



    // character state handling (movement, attacks, defenses)
    k_h_move <= k_left ? walk_left : k_right ? walk_right : sit_still;   // horizontal
        movement

// vertical movement (flapping, falling and jumping)
    k_v_move <= k_v_move == sit_still ?
                   (k_up || !(|k_on_floor)) ? fall : sit_still :
                k_v_move == fall ?
                   (|k_on_floor && (sprite_vel_y[KIRBY] > 0)) ? sit_still :
                   (k_up && !k_flapped) ? float : fall
                                        :    // float
                   (|k_on_floor && (sprite_vel_y[KIRBY] > 0)) ? sit_still :
                                           k_down ? fall : float;

          // handle actions being performed
    if (k_attack && k_hit_timer == 0 && k_do_timer == 0) begin
                              k_do_timer <=
                f_red_gesture == melee_gesture_2 ? k_move_time[29:20] :
                f_red_gesture == melee_gesture_1 ? k_move_time[39:30] : k_move_time[49:40];
                              k_do <= attack;
                              k_a_move <= f_red_gesture;

        //handle ranged attacks
        if (f_red_gesture == ranged_gesture_1 && k_cooldown[0] == 0) begin
               if (~(k_ammo_active & 2'b01)) begin
                   k_ammo_active <= k_ammo_active | 2'b01;
                   sprite_display[KAMMO1] <= 1;
                                           sprite_x[KAMMO1*posbits + posbits -1 :
        KAMMO1*posbits] <= sprite_orientation[KIRBY] ? (sprite_x[posbits*KIRBY+posbits-1 :
        posbits*KIRBY] + SPRITE_WIDTH[8*KIRBY+7 : 8*KIRBY] - k_ammo_x_offset) :
        (sprite_x[posbits*KIRBY+posbits-1 : posbits*KIRBY] + k_ammo_x_offset);
                                           sprite_y[KAMMO1*posbits + posbits -1 :
        KAMMO1*posbits] <= sprite_y[posbits*KIRBY+posbits-1 : posbits*KIRBY] +
        k_ammo_y_offset;
                                    sprite_vel_x[KAMMO1] <= sprite_orientation[KIRBY] ?
                                      k_base_shot_vel + k_bonus_vel*(~f_red_speed) :
                                      -k_base_shot_vel - k_bonus_vel*(~f_red_speed);
                   k_ammo_time[0] <= k_ammo_life_1;
                                           sprite_orientation[KAMMO1] <=
        ~sprite_orientation[KIRBY];
                                           k_do_timer <= k_move_time[19:10];
                                           k_ammo_frame_count[0] <= k_ranged_count_1;
                                           k_cooldown[0] <= k_ranged_cooldown_1;
               end
        end else if (f_red_gesture == ranged_gesture_2 && k_cooldown[1] == 0) begin
               if (~(k_ammo_active & 2'b10)) begin
                   k_ammo_active <= k_ammo_active | 2'b10;
                   sprite_display[KAMMO2] <= 1;
                                           sprite_x[KAMMO2*posbits + posbits -1 :
        KAMMO2*posbits] <= sprite_orientation[KIRBY] ? (sprite_x[posbits*KIRBY+posbits-1 :
        posbits*KIRBY] + SPRITE_WIDTH[8*KIRBY+7 : 8*KIRBY] - k_ammo_x_offset) :
        (sprite_x[posbits*KIRBY+posbits-1 : posbits*KIRBY] + k_ammo_x_offset);
                                           sprite_y[KAMMO2*posbits + posbits -1 :
        KAMMO2*posbits] <= sprite_y[posbits*KIRBY+posbits-1 : posbits*KIRBY] +
        k_ammo_y_offset;
                                    sprite_vel_x[KAMMO2] <= sprite_orientation[KIRBY] ?
                                      k_lobbed_x_vel:
                                      -k_lobbed_x_vel;
                                           sprite_vel_y[KAMMO2] <= k_lobbed_y_vel;
                   k_ammo_time[1] <= k_ammo_life_2;
```

```
                                        sprite_orientation[KAMMO2] <=
    ~sprite_orientation[KIRBY];
                                        k_do_timer <= k_move_time[59:50];
                                        k_ammo_frame_count[1] <= k_ranged_count_2;
                                        k_cooldown[1] <= k_ranged_cooldown_2;
           end
    end else if (f_red_gesture == ranged_gesture_3 && k_cooldown[1] == 0) begin
           if (~(k_ammo_active & 2'b10)) begin
               k_ammo_active <= k_ammo_active | 2'b10;
               sprite_display[KAMMO2] <= 1;
                                        sprite_x[KAMMO2*posbits + posbits -1 :
    KAMMO2*posbits] <= sprite_orientation[KIRBY] ? (sprite_x[posbits*KIRBY+posbits-1 :
    posbits*KIRBY] + SPRITE_WIDTH[8*KIRBY+7 : 8*KIRBY] - k_ammo_x_offset) :
    (sprite_x[posbits*KIRBY+posbits-1 : posbits*KIRBY] + k_ammo_x_offset);
                                        sprite_y[KAMMO2*posbits + posbits -1 :
    KAMMO2*posbits] <= sprite_y[posbits*KIRBY+posbits-1 : posbits*KIRBY] +
    k_ammo_y_offset;
                                sprite_vel_x[KAMMO2] <= sprite_vel_x[KIRBY];
                                        sprite_vel_y[KAMMO2] <= k_bomb_y_vel;
               k_ammo_time[1] <= k_ammo_life_2;
                                        sprite_orientation[KAMMO2] <=
    ~sprite_orientation[KIRBY];
                                        k_do_timer <= k_move_time[59:50];
                                        k_ammo_frame_count[1] <= k_ranged_count_2;
                                        k_cooldown[1] <= k_ranged_cooldown_2;
           end
    end else begin
        sprite_display[SWORD] <= 1;
           end
end else if(k_defend && k_hit_timer == 0 && k_do_timer == 0) begin
    k_do_timer <= k_move_time[9:0];
    k_do <= defend;
    k_d_move <= f_red_gesture;
end
if ((k_do == attack || k_do == defend) && k_do_timer == 0) begin
    k_do <= sit_still;
    sprite_display[SWORD] <= 0;
end



// collision handling - for picking up, attacking and defending
if (collisions[KIRBY_HAMMER] && k_collide_counter == 0) begin
       if (k_do == defend) begin
           kirby_health <= (k_attack_dmg[6:0]>>1) > (d_a_move == melee_gesture_2 ?
    d_attack_dmg[20:14] : d_attack_dmg[27:21]) ? kirby_health : kirby_health -
    ((d_a_move == melee_gesture_2 ? d_attack_dmg[20:14] : d_attack_dmg[27:21])>>1) +
    k_attack_dmg[6:0];

                                // blood effects
                                sprite_display[BLOOD2:BLOOD1] <= 2'b11;
                                sprite_vel_x[BLOOD1] <= sprite_orientation[DEDEDE] ?
    5 : -5;
                                sprite_vel_y[BLOOD1] <= -2;
                                sprite_vel_x[BLOOD2] <= sprite_orientation[DEDEDE] ?
    3 : -3;
                                sprite_vel_y[BLOOD2] <= -3;
                                sprite_x[BLOOD1*posbits + posbits - 1 :
    BLOOD1*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_x;
                                sprite_y[BLOOD1*posbits + posbits - 1 :
    BLOOD1*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_y;
                                sprite_x[BLOOD2*posbits + posbits - 1 :
    BLOOD2*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_x;
                                sprite_y[BLOOD2*posbits + posbits - 1 :
    BLOOD2*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_y;
```

```
                                sprite_x[HURT1*posbits + posbits - 1 :
HURT1*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x_1;
                                sprite_y[HURT1*posbits + posbits - 1 :
HURT1*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y_1;
                                sprite_data[HURT1*4 + 3: HURT1*4] <= 1;
                                sprite_orientation[HURT1] <=
sprite_orientation[DEDEDE];
                                blood_counter[0] <= 0;
                                b_frame_count[0] <= blood_count;
   end else begin
       kirby_health <= kirby_health - (d_a_move == melee_gesture_2 ?
d_attack_dmg[20:14] : d_attack_dmg[27:21]);
       sprite_vel_x[KIRBY] <= sprite_orientation[DEDEDE] ? (d_a_move ==
melee_gesture_2 ? d_attack_x_speed[14:10] : d_attack_x_speed[19:15]) : -(d_a_move
== melee_gesture_2 ? d_attack_x_speed[14:10] : d_attack_x_speed[19:15]);
       sprite_vel_y[KIRBY] <= (d_a_move == melee_gesture_2 ?
-d_attack_y_speed[14:10] : -d_attack_y_speed[19:15]);
       k_hit_timer <= (d_a_move == melee_gesture_2 ? d_attack_hit_time[29:20] :
d_attack_hit_time[39:30]);

                                //blood effects
                                sprite_display[BLOOD4:BLOOD1] <= 4'b1111;
                                sprite_vel_x[BLOOD1] <= sprite_orientation[DEDEDE] ?
5 : -5;
                                sprite_vel_y[BLOOD1] <= -2;
                                sprite_vel_x[BLOOD2] <= sprite_orientation[DEDEDE] ?
3 : -3;
                                sprite_vel_y[BLOOD2] <= -3;
                                sprite_vel_x[BLOOD3] <= sprite_orientation[DEDEDE] ?
-5 : 5;
                                sprite_vel_y[BLOOD3] <= -1;
                                sprite_vel_x[BLOOD4] <= sprite_orientation[DEDEDE] ?
-3 : 3;
                                sprite_vel_y[BLOOD4] <= -2;
                                sprite_x[BLOOD1*posbits + posbits - 1 :
BLOOD1*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                                sprite_y[BLOOD1*posbits + posbits - 1 :
BLOOD1*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                                sprite_x[BLOOD2*posbits + posbits - 1 :
BLOOD2*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                                sprite_y[BLOOD2*posbits + posbits - 1 :
BLOOD2*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                                sprite_x[BLOOD3*posbits + posbits - 1 :
BLOOD3*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                                sprite_y[BLOOD3*posbits + posbits - 1 :
BLOOD3*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                                sprite_x[BLOOD4*posbits + posbits - 1 :
BLOOD4*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                                sprite_y[BLOOD4*posbits + posbits - 1 :
BLOOD4*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;

                                sprite_x[HURT1*posbits + posbits - 1 :
HURT1*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x_1;
                                sprite_y[HURT1*posbits + posbits - 1 :
HURT1*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y_1;
                                sprite_orientation[HURT1] <=
sprite_orientation[DEDEDE];
                                sprite_data[HURT1*4 + 3: HURT1*4] <= 1;
```

```
                                    blood_counter[0] <= 0;
                                    b_frame_count[0] <= blood_count;
        end
                            k_collide_counter <= collide_count;
 end

 for(i = 0; i <= 1; i=i+1) begin
    if (collisions[KIRBY_DAMMO1 + i] && k_collide_counter == 0) begin
        if (k_do == defend) begin
            kirby_health <= (k_attack_dmg[6:0]>>1) > (i == 1 ? d_attack_dmg[41:35] :
     d_attack_dmg[13:7]) ? kirby_health : kirby_health - ((i == 1 ? d_attack_dmg[34:28]
     : d_attack_dmg[13:7])>>1)  + k_attack_dmg[6:0];
        end else begin
            kirby_health <= kirby_health - (i == 1 ? d_attack_dmg[41:35] :
     d_attack_dmg[13:7]);
            sprite_vel_x[KIRBY] <= sprite_vel_x[KIRBY] + (sprite_vel_x[DAMMO1 +
     i]>>>1);
            k_hit_timer <= (i == 1 ? d_attack_hit_time[59:50] :
     d_attack_hit_time[19:10]);
            sprite_display[DAMMO1 + i] <= 0;
            d_ammo_active <= d_ammo_active ^ (4'b0001<<i);

                                    if (i == 1) begin
                                        //sprite_vel_y[KIRBY] <= gravity;
                                        k_up_count <= k_float_max;
                                    end

                                    // blood effects
                                    sprite_display[BLOOD2:BLOOD1] <= 2'b11;
                                    sprite_vel_x[BLOOD1] <= (sprite_vel_x[DAMMO1 +
     i]>>>1) + 1;
                                    sprite_vel_y[BLOOD1] <= -2;
                                    sprite_vel_x[BLOOD2] <= (sprite_vel_x[DAMMO1 +
     i]>>>1);
                                    sprite_vel_y[BLOOD2] <= -3;
                                    sprite_x[BLOOD1*posbits + posbits - 1 :
     BLOOD1*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
     k_blood_offset_x;
                                    sprite_y[BLOOD1*posbits + posbits - 1 :
     BLOOD1*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
     k_blood_offset_y;
                                    sprite_x[BLOOD2*posbits + posbits - 1 :
     BLOOD2*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
     k_blood_offset_x;
                                    sprite_y[BLOOD2*posbits + posbits - 1 :
     BLOOD2*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
     k_blood_offset_y;
        end
                            k_collide_counter <= collide_count;
    end
 end


 // handle side-to-side movement
       if (k_hit_timer == 0 && !(|
     {collisions[KIRBY_DAMMO2:KIRBY_DAMMO1],collisions[KIRBY_HAMMER]}))
                    sprite_vel_x[KIRBY] <= k_v_move == sit_still ? k_h_move ==
     walk_right ? k_move_vel : k_h_move == walk_left ? -k_move_vel : 0:

                                                        k_h_move == walk_right ?
     k_move_vel_air : k_h_move == walk_left ? -k_move_vel_air : 0;
       if (k_v_move == sit_still)
                    sprite_vel_y[KIRBY] <= 0;




 // handle jumping and flapping
 if (k_up && !k_flapped && k_up_count != k_float_max) begin
                    k_up_count <= k_up_count + 1;
```

```
    k_flapped <= 1;
    sprite_vel_y[KIRBY] <= k_v_move == sit_still ? k_jump_impulse : k_flap_impulse;
end else begin
                k_flapped <= k_flapped ? k_up : 0;
                if (k_v_move == sit_still)
                        k_up_count <= 0;
      end


//animation - frame transitioning
      if(k_ammo_frame_count[0] == 0) begin
                sprite_data[KAMMO1*4 + 3: KAMMO1*4] <= sprite_data[KAMMO1*4 + 3:
    KAMMO1*4] == 1 ? 2 : 1;
                k_ammo_frame_count[0] <= k_ranged_count_1;
      end else
              k_ammo_frame_count[0] <= k_ammo_frame_count[0] - 1;

      if(k_ammo_frame_count[1] == 0) begin
                sprite_data[KAMMO2*4 + 3: KAMMO2*4] <= sprite_data[KAMMO2*4 + 3:
    KAMMO2*4] == 1 ? 2 : 1;
                k_ammo_frame_count[1] <= k_ranged_count_2;
      end else
              k_ammo_frame_count[1] <= k_ammo_frame_count[1] - 1;

      if (blood_counter[0] != blood_life) begin
                sprite_display[HURT1] <= 1;
                blood_counter[0] <= blood_counter[0] + 1;
                if(sprite_data[HURT1*4 + 3: HURT1*4] == 2 && b_frame_count[0] == 0)
                        sprite_data[HURT1*4 + 3: HURT1*4] <= 3;
                else if(sprite_data[HURT1*4 + 3: HURT1*4] == 1 && b_frame_count[0]
    == 0) begin
                        sprite_data[HURT1*4 + 3: HURT1*4] <= 2;
                        b_frame_count[0] <= blood_count;
              end
      end else
                sprite_display[HURT1] <= 0;



if (k_h_move == walk_right) begin
    sprite_orientation[KIRBY] <= 1;
    sprite_orientation[SWORD] <= 1;
end else if (k_h_move == walk_left) begin
    sprite_orientation[KIRBY] <= 0;
    sprite_orientation[SWORD] <= 0;
      end

      k_prev_frame <= k_frame;
      if (k_hit_timer != 0) begin
                k_frame <= 12;
      end else if (k_do_timer != 0) begin
                if ((k_a_move == ranged_gesture_1 || k_a_move == ranged_gesture_2)
    && k_do == attack)
                        k_frame <= k_a_move == ranged_gesture_1  ? 10 : 8;
              else if (k_a_move == melee_gesture_1 && k_do == attack) begin
                        // animated melee animation
                        if(sprite_data[SWORD*4 + 3: SWORD*4] == 2 && k_frame_count
    == 0) begin
                                k_frame <= 8;
                                sprite_data[SWORD*4 + 3: SWORD*4] <= 3;
                        end else if(sprite_data[SWORD*4 + 3: SWORD*4] == 1 &&
    k_frame_count == 0) begin
                                k_frame_count <= k_melee_count_2;
                                sprite_data[SWORD*4 + 3: SWORD*4] <= 2;
                        end else if(k_frame != 9) begin
                                k_frame <= 9;
                                k_frame_count <= k_melee_count_1;
                                sprite_data[SWORD*4 + 3: SWORD*4] <= 1;
                        end
                end else if (k_do == attack && k_a_move == melee_gesture_2) begin
```

```
                        // animated melee animation
                        if(sprite_data[SWORD*4 + 3: SWORD*4] == 5 && k_frame_count
    == 0) begin
                                k_frame <= 9;
                                sprite_data[SWORD*4 + 3: SWORD*4] <= 6;
                        end else if(sprite_data[SWORD*4 + 3: SWORD*4] == 4 &&
    k_frame_count == 0) begin
                                k_frame_count <= k_melee_count_2;
                                sprite_data[SWORD*4 + 3: SWORD*4] <= 5;
                        end else if(k_frame != 8) begin
                                k_frame <= 8;
                                k_frame_count <= k_melee_count_1;
                                sprite_data[SWORD*4 + 3: SWORD*4] <= 4;
                        end
                end else
                        k_frame <= 7;
      end else begin
                sprite_data[SWORD*4 + 3: SWORD*4] <= 0;
                if (k_v_move == sit_still) begin
                        if (k_h_move == sit_still)
                                k_frame <= 1;
                        else begin
                                if ((k_frame == 2 || k_frame == 4) && k_frame_count
    == 0) begin
                                        k_frame <= 3;
                                        k_frame_count <= k_walk_frame_count;
                                end else if (k_frame != 2 && k_frame != 4 &&
    k_frame_count == 0) begin
                                        k_frame <= k_prev_frame == 2 ? 4 : 2;
                                        k_frame_count <= k_walk_frame_count;
                                end
                        end
                end else begin
                        if (sprite_vel_y[KIRBY] > 0)
                                k_frame <= 6;
                        else
                                k_frame <= 5;
                end
           end

      sprite_data[KIRBY*4 + 3: KIRBY*4] <= k_frame;

if(k_ammo_time[0] != 0)
   k_ammo_time[0] <= k_ammo_time[0] - 1;
      if(k_ammo_time[1] != 0)
   k_ammo_time[1] <= k_ammo_time[1] - 1;

// handle death of projectiles after time
for(i = KAMMO1; i <= KAMMO2; i=i+1) begin
   if ((k_ammo_active>>(i-KAMMO1)) & 4'b0001) begin
      if (k_ammo_time[i-KAMMO1] == 0) begin
          sprite_display[i] <= 0;
          k_ammo_active <= k_ammo_active ^ (4'b0001<<(i-KAMMO1));
      end
   end
end




        // ---------------------------------------- DEDEDE --------------------------


// character state handling (movement, attacks, defenses)
d_h_move <= d_left ? walk_left : d_right ? walk_right : sit_still;   // horizontal
    movement
```

```
d_v_move <= d_v_move == sit_still ?    // vertical movement (flapping, falling and
    jumping)
             (d_up || !(|d_on_floor)) ? fall : sit_still :
          d_v_move == fall ?
             (|d_on_floor && (sprite_vel_y[DEDEDE] > 0)) ? sit_still :
             (d_up && !d_flapped) ? float : fall
                                  :    // float
             (|d_on_floor && (sprite_vel_y[DEDEDE] > 0)) ? sit_still :
                                        d_down ? fall : float;

if (d_attack && d_hit_timer == 0 && d_do_timer == 0) begin
                          d_do_timer <= f_green_gesture == melee_gesture_2 ?
    d_move_time[29:20] :
                                                              f_green_gesture ==
    melee_gesture_1 ? d_move_time[39:30] : d_move_time[49:40];
                          d_do <= attack;
                          d_a_move <= f_green_gesture;

    //handle action specific things
    if (f_green_gesture == ranged_gesture_1  && d_cooldown[0] == 0) begin
           if (~(d_ammo_active & 2'b01)) begin
               d_ammo_active <= d_ammo_active | 2'b01;
               sprite_display[DAMMO1] <= 1;
                                        sprite_x[DAMMO1*posbits + posbits -1 :
    DAMMO1*posbits] <= sprite_orientation[DEDEDE] ? (sprite_x[posbits*DEDEDE+posbits-1
    : posbits*DEDEDE] + SPRITE_WIDTH[8*DEDEDE+7 : 8*DEDEDE] - d_ammo_x_offset) :
    (sprite_x[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + d_ammo_x_offset);
                                        sprite_y[DAMMO1*posbits + posbits -1 :
    DAMMO1*posbits] <= sprite_y[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] +
    d_ammo_y_offset;
                                  sprite_vel_x[DAMMO1] <= sprite_orientation[DEDEDE] ?
                                    d_base_shot_vel + d_bonus_vel*(~f_green_speed) :
                                    -d_base_shot_vel - d_bonus_vel*(~f_green_speed);
               d_ammo_time[0] <= d_ammo_life_1;
                                        sprite_orientation[DAMMO1] <=
    ~sprite_orientation[DEDEDE];
                                        d_do_timer <= k_move_time[19:10];
                                        d_ammo_frame_count[0] <= d_ranged_count_1;
                                        d_cooldown[0] <= d_ranged_cooldown_1;
           end
    end else if (f_green_gesture == ranged_gesture_2  && d_cooldown[1] == 0) begin
           if (~(d_ammo_active & 2'b10)) begin
               d_ammo_active <= d_ammo_active | 2'b10;
               sprite_display[DAMMO2] <= 1;
                                        sprite_x[DAMMO2*posbits + posbits -1 :
    DAMMO2*posbits] <= sprite_orientation[DEDEDE] ? (sprite_x[posbits*DEDEDE+posbits-1
    : posbits*DEDEDE] + SPRITE_WIDTH[8*DEDEDE+7 : 8*DEDEDE] - d_ammo_x_offset) :
    (sprite_x[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + d_ammo_x_offset);
                                        sprite_y[DAMMO2*posbits + posbits -1 :
    DAMMO2*posbits] <= sprite_y[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] +
    d_ammo_y_offset;
                                  sprite_vel_x[DAMMO2] <= sprite_orientation[DEDEDE] ?
                                    d_lobbed_x_vel:
                                    -d_lobbed_x_vel;
                                        sprite_vel_y[DAMMO2] <= d_lobbed_y_vel;
               d_ammo_time[1] <= d_ammo_life_2;
                                        sprite_orientation[DAMMO1] <=
    ~sprite_orientation[DEDEDE];
                                        d_do_timer <= d_move_time[59:50];
                                        d_ammo_frame_count[1] <= d_ranged_count_2;
                                        d_cooldown[1] <= d_ranged_cooldown_2;
           end
    end else if (f_green_gesture == ranged_gesture_3  && d_cooldown[1] == 0) begin
           if (~(d_ammo_active & 2'b10)) begin
               d_ammo_active <= d_ammo_active | 2'b10;
               sprite_display[DAMMO2] <= 1;
                                        sprite_x[DAMMO2*posbits + posbits -1 :
    DAMMO2*posbits] <= sprite_orientation[DEDEDE] ? (sprite_x[posbits*DEDEDE+posbits-1
    : posbits*DEDEDE] + SPRITE_WIDTH[8*DEDEDE+7 : 8*DEDEDE] - d_ammo_x_offset) :
    (sprite_x[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + d_ammo_x_offset);
```

```
                                        sprite_y[DAMMO2*posbits + posbits -1 :
    DAMMO2*posbits] <= sprite_y[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] +
    d_ammo_y_offset;
                                sprite_vel_x[DAMMO2] <= sprite_vel_x[DEDEDE];
                                        sprite_vel_y[DAMMO2] <= d_bomb_y_vel;
              d_ammo_time[1] <= d_ammo_life_2;
                                        sprite_orientation[DAMMO1] <=
    ~sprite_orientation[DEDEDE];
                                        d_do_timer <= d_move_time[59:50];
                                        d_ammo_frame_count[1] <= d_ranged_count_2;
                                        d_cooldown[1] <= d_ranged_cooldown_2;
           end
    end else begin
        sprite_display[HAMMER] <= 1;
           end
end else if(d_defend && d_hit_timer == 0 && d_do_timer == 0) begin
    d_do_timer <= d_move_time[9:0];
    d_do <= defend;
    d_d_move <= f_green_gesture;
end
if ((d_do == attack || d_do == defend) && d_do_timer == 0) begin
    d_do <= sit_still;
    sprite_display[HAMMER] <= 0;
end



// collision handling - for picking up, attacking and defending
if (collisions[DEDEDE_SWORD] && d_collide_counter == 0) begin
       if (d_do == defend) begin
           dedede_health <= (d_attack_dmg[6:0]>>1) > (k_a_move == melee_gesture_2 ?
    k_attack_dmg[20:14] : k_attack_dmg[27:21]) ? dedede_health : dedede_health -
    ((k_a_move == melee_gesture_2 ? k_attack_dmg[20:14] : k_attack_dmg[27:21])>>1) +
    d_attack_dmg[6:0];

                                // blood effects
                                sprite_display[BLOOD6:BLOOD5] <= 2'b11;
                                sprite_vel_x[BLOOD5] <= sprite_orientation[KIRBY] ?
    5 : -5;
                                sprite_vel_y[BLOOD5] <= -2;
                                sprite_vel_x[BLOOD6] <= sprite_orientation[KIRBY] ?
    3 : -3;
                                sprite_vel_y[BLOOD6] <= -3;
                                sprite_x[BLOOD5*posbits + posbits - 1 :
    BLOOD5*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_x;
                                sprite_y[BLOOD5*posbits + posbits - 1 :
    BLOOD5*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_y;
                                sprite_x[BLOOD6*posbits + posbits - 1 :
    BLOOD6*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_x;
                                sprite_y[BLOOD6*posbits + posbits - 1 :
    BLOOD6*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_y;

                                sprite_x[HURT1*posbits + posbits - 1 :
    HURT1*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_x_1;
                                sprite_y[HURT1*posbits + posbits - 1 :
    HURT1*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_y_1;
                                sprite_orientation[HURT1] <=
    sprite_orientation[KIRBY];
                                sprite_data[HURT1*4 + 3: HURT1*4] <= 1;
                                blood_counter[0] <= 0;
                                b_frame_count[0] <= blood_count;
       end else begin
           dedede_health <= dedede_health - (k_a_move == melee_gesture_2 ?
    k_attack_dmg[20:14] : k_attack_dmg[27:21]);
```

```
            sprite_vel_x[DEDEDE] <= sprite_orientation[KIRBY] ? (k_a_move ==
        melee_gesture_2 ? k_attack_x_speed[14:10] : k_attack_x_speed[19:15]) : -(k_a_move
        == melee_gesture_2 ? k_attack_x_speed[14:10] : k_attack_x_speed[19:15]);
            sprite_vel_y[DEDEDE] <= (k_a_move == melee_gesture_2 ?
        -k_attack_y_speed[14:10] : -k_attack_y_speed[19:15]);
            d_hit_timer <= (k_a_move == melee_gesture_2 ? k_attack_hit_time[29:20] :
        k_attack_hit_time[39:30]);

                                    //blood effects
                                    sprite_display[BLOOD8:BLOOD5] <= 4'b1111;
                                    sprite_vel_x[BLOOD5] <= sprite_orientation[DEDEDE] ?
        5 : -5;
                                    sprite_vel_y[BLOOD5] <= -2;
                                    sprite_vel_x[BLOOD6] <= sprite_orientation[DEDEDE] ?
        3 : -3;
                                    sprite_vel_y[BLOOD6] <= -3;
                                    sprite_vel_x[BLOOD7] <= sprite_orientation[DEDEDE] ?
        -5 : 5;
                                    sprite_vel_y[BLOOD7] <= -1;
                                    sprite_vel_x[BLOOD8] <= sprite_orientation[DEDEDE] ?
        -3 : 3;
                                    sprite_vel_y[BLOOD8] <= -2;
                                    sprite_x[BLOOD5*posbits + posbits - 1 :
        BLOOD5*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_x;
                                    sprite_y[BLOOD5*posbits + posbits - 1 :
        BLOOD5*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_y;
                                    sprite_x[BLOOD6*posbits + posbits - 1 :
        BLOOD6*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_x;
                                    sprite_y[BLOOD6*posbits + posbits - 1 :
        BLOOD6*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_y;
                                    sprite_x[BLOOD7*posbits + posbits - 1 :
        BLOOD7*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_x;
                                    sprite_y[BLOOD7*posbits + posbits - 1 :
        BLOOD7*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_y;
                                    sprite_x[BLOOD8*posbits + posbits - 1 :
        BLOOD8*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_x;
                                    sprite_y[BLOOD8*posbits + posbits - 1 :
        BLOOD8*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_y;

                                    sprite_x[HURT1*posbits + posbits - 1 :
        HURT1*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_x_1;
                                    sprite_y[HURT1*posbits + posbits - 1 :
        HURT1*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        d_blood_offset_y_1;
                                    sprite_orientation[HURT1] <=
        sprite_orientation[KIRBY];
                                    sprite_data[HURT1*4 + 3: HURT1*4] <= 1;
                                    blood_counter[0] <= 0;
                                    b_frame_count[0] <= blood_count;
           end
                               d_collide_counter <= collide_count;
    end

    for(i = 0; i <= 1; i=i+1) begin
       if (collisions[DEDEDE_KAMMO1 + i] && d_collide_counter == 0) begin
           if (d_do == defend) begin
               dedede_health <= (d_attack_dmg[6:0]>>1) > (i == 1 ? k_attack_dmg[41:35] :
        k_attack_dmg[13:7]) ? dedede_health : dedede_health - ((i == 1 ?
        k_attack_dmg[41:35] : k_attack_dmg[13:7])>>1)  + d_attack_dmg[6:0];
           end else begin
               dedede_health <= dedede_health - (i == 1 ? k_attack_dmg[41:35] :
```

```
    k_attack_dmg[13:7]) ;
           sprite_vel_x[DEDEDE] <= sprite_vel_x[DEDEDE] + (sprite_vel_x[KAMMO1 +
    i]>>>1) + 1;
           d_hit_timer <= (i == 1 ? k_attack_hit_time[59:50] :
    k_attack_hit_time[19:10]);
           sprite_display[KAMMO1 + i] <= 0;
           k_ammo_active <= k_ammo_active ^ (2'b01<<i);

                                  if (i == 1) begin
                                        //sprite_vel_y[DEDEDE] <= gravity;
                                        d_up_count <= d_float_max;
                                  end

                                  // blood effects
                                  sprite_display[BLOOD6:BLOOD5] <= 2'b11;
                                  sprite_vel_x[BLOOD5] <= (sprite_vel_x[KAMMO1 +
    i]>>>1);
                                  sprite_vel_y[BLOOD5] <= -2;
                                  sprite_vel_x[BLOOD6] <= (sprite_vel_x[KAMMO1 +
    i]>>>1);
                                  sprite_vel_y[BLOOD6] <= -3;
                                  sprite_x[BLOOD5*posbits + posbits - 1 :
    BLOOD5*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_x;
                                  sprite_y[BLOOD5*posbits + posbits - 1 :
    BLOOD5*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_y;
                                  sprite_x[BLOOD6*posbits + posbits - 1 :
    BLOOD6*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_x;
                                  sprite_y[BLOOD6*posbits + posbits - 1 :
    BLOOD6*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
    d_blood_offset_y;

       end
                        d_collide_counter <= collide_count;
   end
end


 // handle side-to-side movement
      if (d_hit_timer == 0 && !(|
    {collisions[DEDEDE_KAMMO2:DEDEDE_KAMMO1],collisions[DEDEDE_SWORD]}))
                  sprite_vel_x[DEDEDE] <= d_v_move == sit_still ? d_h_move ==
    walk_right ? d_move_vel : d_h_move == walk_left ? -d_move_vel : 0:

                                                  d_h_move == walk_right ?
    d_move_vel_air : d_h_move == walk_left ? -d_move_vel_air : 0;
      if (d_v_move == sit_still)
                  sprite_vel_y[DEDEDE] <= 0;



 // handle jumping and flapping
 if (d_up && !d_flapped && d_up_count != d_float_max) begin
                  d_up_count <= d_up_count + 1;
    d_flapped <= 1;
    sprite_vel_y[DEDEDE] <= d_v_move == sit_still ? d_jump_impulse : d_flap_impulse;
 end else begin
                  d_flapped <= d_flapped ? d_up : 0;
                  if (d_v_move == sit_still)
                        d_up_count <= 0;
      end


 //animation - frame transitioning
      if(d_ammo_frame_count[0] == 0) begin
                  sprite_data[DAMMO1*4 + 3: DAMMO1*4] <= sprite_data[DAMMO1*4 + 3:
    DAMMO1*4] == 1 ? 2 : 1;
```

```
                d_ammo_frame_count[0] <= d_ranged_count_1;
        end else
                d_ammo_frame_count[0] <= d_ammo_frame_count[0] - 1;

        if(d_ammo_frame_count[1] == 0) begin
                sprite_data[DAMMO2*4 + 3: DAMMO2*4] <= sprite_data[DAMMO2*4 + 3:
    DAMMO2*4] == 1 ? 2 : 1;
                d_ammo_frame_count[1] <= d_ranged_count_2;
        end else
                d_ammo_frame_count[1] <= d_ammo_frame_count[1] - 1;

if (d_h_move == walk_right) begin
    sprite_orientation[DEDEDE] <= 1;
    sprite_orientation[HAMMER] <= 1;
end else if (d_h_move == walk_left) begin
    sprite_orientation[DEDEDE] <= 0;
    sprite_orientation[HAMMER] <= 0;
      end

      d_prev_frame <= d_frame;
      if (d_hit_timer != 0) begin
                d_frame <= 12;
      end else if (d_do_timer != 0) begin
                if ((d_a_move == ranged_gesture_1 || d_a_move == ranged_gesture_2)
    && d_do == attack)
                        d_frame <= d_a_move == ranged_gesture_1  ? 10 : 8;
                else if (d_a_move == melee_gesture_1 && d_do == attack) begin
                        // animated melee animation
                        if(sprite_data[HAMMER*4 + 3: HAMMER*4] == 2 && d_frame_count
    == 0) begin
                                d_frame <= 8;
                                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 3;
                        end else if(sprite_data[HAMMER*4 + 3: HAMMER*4] == 1 &&
    d_frame_count == 0) begin
                                d_frame_count <= d_melee_count_2;
                                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 2;
                        end else if(d_frame != 9) begin
                                d_frame <= 9;
                                d_frame_count <= d_melee_count_1;
                                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 1;
                        end
                end else if (d_do == attack && d_a_move == melee_gesture_2) begin
                        // animated melee animation
                        if(sprite_data[HAMMER*4 + 3: HAMMER*4] == 5 && d_frame_count
    == 0) begin
                                d_frame <= 9;
                                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 6;
                        end else if(sprite_data[HAMMER*4 + 3: HAMMER*4] == 4 &&
    d_frame_count == 0) begin
                                d_frame_count <= d_melee_count_2;
                                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 5;
                        end else if(d_frame != 8) begin
                                d_frame <= 8;
                                d_frame_count <= d_melee_count_1;
                                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 4;
                        end
                end else
                        d_frame <= 7;
      end else begin
                sprite_data[HAMMER*4 + 3: HAMMER*4] <= 0;
                if (d_v_move == sit_still) begin
                        if (d_h_move == sit_still)
                                d_frame <= 1;
                        else begin
                                if ((d_frame == 2 || d_frame == 4) && d_frame_count
    == 0) begin
                                        d_frame <= 3;
                                        d_frame_count <= d_walk_frame_count;
                                end else if (k_frame != 2 && k_frame != 4 &&
    d_frame_count == 0) begin
```

```
                                        d_frame <= d_prev_frame == 2 ? 4 : 2;
                                        d_frame_count <= d_walk_frame_count;
                                end
                        end
                end else begin
                        if (sprite_vel_y[DEDEDE] > 0)
                                d_frame <= 6;
                        else
                                d_frame <= 5;
                end
        end

      sprite_data[DEDEDE*4 + 3: DEDEDE*4] <= d_frame;

   if(d_ammo_time[0] != 0)
      d_ammo_time[0] <= d_ammo_time[0] - 1;
        if(d_ammo_time[1] != 0)
      d_ammo_time[1] <= d_ammo_time[1] - 1;

   // handle death of projectiles after time
   for(i = DAMMO1; i <= DAMMO2; i=i+1) begin
      if ((d_ammo_active>>(i-DAMMO1)) & 2'b01) begin
         if (d_ammo_time[i-DAMMO1] == 0) begin
             sprite_display[i] <= 0;
             d_ammo_active <= d_ammo_active ^ (2'b01<<(i-DAMMO1));
         end
      end
   end



// ---------------------------------- General ----------------------------------


   // handle position updating
                    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + sprite_vel_x[KIRBY];
                    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] <=
    (sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + sprite_vel_y[KIRBY]) <
    top ? sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] :
    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + sprite_vel_y[KIRBY];
    sprite_x[posbits*SWORD + posbits-1 : posbits*SWORD] <= k_a_move == melee_gesture_1
    ?


           sprite_orientation[KIRBY] ? (sprite_x[posbits*KIRBY+posbits-1 :
    posbits*KIRBY] + SPRITE_WIDTH[8*KIRBY+7 : 8*KIRBY] + sprite_vel_x[KIRBY] +
    k_melee_x_offset_1) : (sprite_x[posbits*KIRBY+posbits-1 : posbits*KIRBY] +
    sprite_vel_x[KIRBY] - k_melee_x_offset_1 - SPRITE_WIDTH[8*SWORD+7 : 8*SWORD]):


           sprite_x[posbits*KIRBY+posbits-1 : posbits*KIRBY] + sprite_vel_x[KIRBY] -
    k_melee_x_offset_2 - SPRITE_WIDTH[8*SWORD+7 : 8*SWORD];
    sprite_y[posbits*SWORD+posbits-1 : posbits*SWORD] <= k_a_move == melee_gesture_1 ?


           sprite_y[posbits*KIRBY+posbits-1 : posbits*KIRBY] + k_melee_y_offset_1 +
    sprite_vel_y[KIRBY] :


           sprite_y[posbits*KIRBY+posbits-1 : posbits*KIRBY] + k_melee_y_offset_2 +
    sprite_vel_y[KIRBY];

                    if (k_ammo_time[0] != 0) begin
                        sprite_x[KAMMO1*posbits + posbits - 1 : KAMMO1*posbits] <=
    sprite_x[KAMMO1*posbits + posbits - 1 : KAMMO1*posbits] + sprite_vel_x[KAMMO1];
                        sprite_y[KAMMO1*posbits + posbits - 1 : KAMMO1*posbits] <=
    sprite_y[KAMMO1*posbits + posbits - 1 : KAMMO1*posbits] + sprite_vel_y[KAMMO1];
                    end if (k_ammo_time[1] != 0 && sprite_y[KAMMO2*posbits + posbits -
```

```
    1 : KAMMO2*posbits] < bottom) begin
                        sprite_x[KAMMO2*posbits + posbits - 1 : KAMMO2*posbits] <=
    sprite_x[KAMMO2*posbits + posbits - 1 : KAMMO2*posbits] + sprite_vel_x[KAMMO2];
                        sprite_y[KAMMO2*posbits + posbits - 1 : KAMMO2*posbits] <=
    (sprite_y[KAMMO2*posbits + posbits - 1 : KAMMO2*posbits] + sprite_vel_y[KAMMO2]) <
    top ? sprite_y[KAMMO2*posbits + posbits - 1 : KAMMO2*posbits] :
    sprite_y[KAMMO2*posbits + posbits - 1 : KAMMO2*posbits] + sprite_vel_y[KAMMO2];
                end



           sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] <=
    sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + sprite_vel_x[DEDEDE];
                sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] <=
    (sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + sprite_vel_y[DEDEDE]) <
    top ? sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] :
    sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + sprite_vel_y[DEDEDE];
    sprite_x[posbits*HAMMER + posbits-1 : posbits*HAMMER] <= d_a_move ==
    melee_gesture_1 ?


           sprite_orientation[DEDEDE] ? (sprite_x[posbits*DEDEDE+posbits-1 :
    posbits*DEDEDE] + SPRITE_WIDTH[8*DEDEDE+7 : 8*DEDEDE] + sprite_vel_x[DEDEDE] +
    d_melee_x_offset_1) : (sprite_x[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] +
    sprite_vel_x[DEDEDE] - d_melee_x_offset_1 - SPRITE_WIDTH[8*HAMMER+7 : 8*HAMMER]):


           sprite_x[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + sprite_vel_x[DEDEDE]
    - k_melee_x_offset_2 - SPRITE_WIDTH[8*HAMMER+7 : 8*HAMMER];
    sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER] <= d_a_move == melee_gesture_1
    ?


           sprite_y[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + d_melee_y_offset_1 +
    sprite_vel_y[DEDEDE] :


           sprite_y[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + d_melee_y_offset_2 +
    sprite_vel_y[DEDEDE];

                if (d_ammo_time[0] != 0) begin
                        sprite_x[DAMMO1*posbits + posbits - 1 : DAMMO1*posbits] <=
    sprite_x[DAMMO1*posbits + posbits - 1 : DAMMO1*posbits] + sprite_vel_x[DAMMO1];
                        sprite_y[DAMMO1*posbits + posbits - 1 : DAMMO1*posbits] <=
    sprite_y[DAMMO1*posbits + posbits - 1 : DAMMO1*posbits] + sprite_vel_y[DAMMO1];
                end if (d_ammo_time[1] != 0 && sprite_y[DAMMO2*posbits + posbits -
    1 : DAMMO2*posbits] < bottom) begin
                        sprite_x[DAMMO2*posbits + posbits - 1 : DAMMO2*posbits] <=
    sprite_x[DAMMO2*posbits + posbits - 1 : DAMMO2*posbits] + sprite_vel_x[DAMMO2];
                        sprite_y[DAMMO2*posbits + posbits - 1 : DAMMO2*posbits] <=
    (sprite_y[DAMMO2*posbits + posbits - 1 : DAMMO2*posbits] + sprite_vel_y[DAMMO2]) <
    top ? sprite_y[DAMMO2*posbits + posbits - 1 : DAMMO2*posbits] :
    sprite_y[DAMMO2*posbits + posbits - 1 : DAMMO2*posbits] + sprite_vel_y[DAMMO2];
                end


p_count <= (p_count == (p_frames + 1)) ? 0 : p_count + 1;
if (p_count == p_frames) begin
    if (h_move) begin
                        h_move <= sprite_x[PLATFORM1*posbits + posbits - 1 :
    PLATFORM1*posbits] >= p1_xmax ? 0 : 1;
                        sprite_x[PLATFORM1*posbits + posbits - 1 :
    PLATFORM1*posbits] <= sprite_x[PLATFORM1*posbits + posbits - 1 :
    PLATFORM1*posbits] + p_h_move;
                        sprite_x[PLATFORM2*posbits + posbits - 1 :
    PLATFORM2*posbits] <= sprite_x[PLATFORM2*posbits + posbits - 1 :
    PLATFORM2*posbits] + p_h_move;
                end else begin
                        h_move <= sprite_x[PLATFORM1*posbits + posbits - 1 :
```

```
PLATFORM1*posbits] <= p1_xmin ? 1 : 0;
                        sprite_x[PLATFORM1*posbits + posbits - 1 :
PLATFORM1*posbits] <= sprite_x[PLATFORM1*posbits + posbits - 1 :
PLATFORM1*posbits] - p_h_move;
                        sprite_x[PLATFORM2*posbits + posbits - 1 :
PLATFORM2*posbits] <= sprite_x[PLATFORM2*posbits + posbits - 1 :
PLATFORM2*posbits] - p_h_move;
                end
if (v_move) begin // 4 for 3 and 3 for 4 on_floor
                        v_move <= sprite_y[PLATFORM3*posbits + posbits - 1 :
PLATFORM3*posbits] >= p3_ymax ? 0 : 1;
                        sprite_y[PLATFORM3*posbits + posbits - 1 :
PLATFORM3*posbits] <= sprite_y[PLATFORM3*posbits + posbits - 1 :
PLATFORM3*posbits] + p_v_move;
                        sprite_y[PLATFORM4*posbits + posbits - 1 :
PLATFORM4*posbits] <= sprite_y[PLATFORM4*posbits + posbits - 1 :
PLATFORM4*posbits] - p_v_move;
                        if(k_on_floor[4]) begin
                                sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits]
<= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + p_v_move;
                                sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits]
<= sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits] + p_v_move;
                        end else if(k_on_floor[3]) begin
                                sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits]
<= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] - p_v_move;
                                sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits]
<= sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits] - p_v_move;
                        end
                        if(d_on_floor[4]) begin
                                sprite_y[DEDEDE*posbits + posbits - 1 :
DEDEDE*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
p_v_move;
                                sprite_y[HAMMER*posbits + posbits - 1 :
HAMMER*posbits] <= sprite_y[HAMMER*posbits + posbits - 1 : HAMMER*posbits] +
p_v_move;
                        end else if(d_on_floor[3]) begin
                                sprite_y[DEDEDE*posbits + posbits - 1 :
DEDEDE*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] -
p_v_move;
                                sprite_y[HAMMER*posbits + posbits - 1 :
HAMMER*posbits] <= sprite_y[HAMMER*posbits + posbits - 1 : HAMMER*posbits] -
p_v_move;
                        end
                end else begin
                        v_move <= sprite_y[PLATFORM3*posbits + posbits - 1 :
PLATFORM3*posbits] <= p3_ymin ? 1 : 0;
                        sprite_y[PLATFORM3*posbits + posbits - 1 :
PLATFORM3*posbits] <= sprite_y[PLATFORM3*posbits + posbits - 1 :
PLATFORM3*posbits] - p_v_move;
                        sprite_y[PLATFORM4*posbits + posbits - 1 :
PLATFORM4*posbits] <= sprite_y[PLATFORM4*posbits + posbits - 1 :
PLATFORM4*posbits] + p_v_move;
                        if(k_on_floor[4]) begin
                                sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits]
<= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] - p_v_move;
                                sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits]
<= sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits] - p_v_move;
                        end else if(k_on_floor[3]) begin
                                sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits]
<= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + p_v_move;
                                sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits]
<= sprite_y[SWORD*posbits + posbits - 1 : SWORD*posbits] + p_v_move;
                        end
                        if(d_on_floor[4]) begin
                                sprite_y[DEDEDE*posbits + posbits - 1 :
DEDEDE*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] -
p_v_move;
                                sprite_y[HAMMER*posbits + posbits - 1 :
HAMMER*posbits] <= sprite_y[HAMMER*posbits + posbits - 1 : HAMMER*posbits] -
p_v_move;
```

```
                                end else if(d_on_floor[3]) begin
                                        sprite_y[DEDEDE*posbits + posbits - 1 :
        DEDEDE*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
        p_v_move;
                                        sprite_y[HAMMER*posbits + posbits - 1 :
        HAMMER*posbits] <= sprite_y[HAMMER*posbits + posbits - 1 : HAMMER*posbits] +
        p_v_move;
                                end
                        end
    end


          if (k_fatal)
                        dedede_health <= -100;
          if (d_fatal)
                        kirby_health <= -100;

end else if (!pause) begin

          g_count <= (g_count == (g_frames + 1)) ? 0 : g_count + 1;
    if (g_count == g_frames) begin
                        sprite_vel_y[BLOOD1] <= blood_on_floor[0] ? 0 :
        sprite_vel_y[BLOOD1] + gravity;
                        sprite_vel_y[BLOOD2] <= blood_on_floor[1] ? 0 :
        sprite_vel_y[BLOOD2] + gravity;
                        sprite_vel_y[BLOOD3] <= blood_on_floor[2] ? 0 :
        sprite_vel_y[BLOOD3] + gravity;
                        sprite_vel_y[BLOOD4] <= blood_on_floor[3] ? 0 :
        sprite_vel_y[BLOOD4] + gravity;
                        sprite_vel_y[BLOOD5] <= blood_on_floor[4] ? 0 :
        sprite_vel_y[BLOOD5] + gravity;
                        sprite_vel_y[BLOOD6] <= blood_on_floor[5] ? 0 :
        sprite_vel_y[BLOOD6] + gravity;
                        sprite_vel_y[BLOOD7] <= blood_on_floor[6] ? 0 :
        sprite_vel_y[BLOOD7] + gravity;
                        sprite_vel_y[BLOOD8] <= blood_on_floor[7] ? 0 :
        sprite_vel_y[BLOOD8] + gravity;
    end

          if (blood_on_floor[0]) begin
                        sprite_vel_y[BLOOD1] <= 0;
          end if (blood_on_floor[1]) begin
                        sprite_vel_y[BLOOD2] <= 0;
          end if (blood_on_floor[2]) begin
                        sprite_vel_y[BLOOD3] <= 0;
          end if (blood_on_floor[3]) begin
                        sprite_vel_y[BLOOD4] <= 0;
          end if (blood_on_floor[4]) begin
                        sprite_vel_y[BLOOD5] <= 0;
          end if (blood_on_floor[5]) begin
                        sprite_vel_y[BLOOD6] <= 0;
          end if (blood_on_floor[6]) begin
                        sprite_vel_y[BLOOD7] <= 0;
          end if (blood_on_floor[7]) begin
                        sprite_vel_y[BLOOD8] <= 0;
          end

                        sprite_x[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] <=
        sprite_x[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] + sprite_vel_x[BLOOD1];
                        sprite_y[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] <=
        sprite_y[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] + sprite_vel_y[BLOOD1];
                        sprite_x[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] <=
        sprite_x[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] + sprite_vel_x[BLOOD2];
                        sprite_y[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] <=
        sprite_y[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] + sprite_vel_y[BLOOD2];
                        sprite_x[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] <=
        sprite_x[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] + sprite_vel_x[BLOOD3];
                        sprite_y[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] <=
        sprite_y[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] + sprite_vel_y[BLOOD3];
                        sprite_x[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] <=
```

```
sprite_x[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] + sprite_vel_x[BLOOD4];
                sprite_y[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] <=
sprite_y[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] + sprite_vel_y[BLOOD4];
                sprite_x[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] <=
sprite_x[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] + sprite_vel_x[BLOOD5];
                sprite_y[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] <=
sprite_y[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] + sprite_vel_y[BLOOD5];
                sprite_x[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] <=
sprite_x[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] + sprite_vel_x[BLOOD6];
                sprite_y[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] <=
sprite_y[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] + sprite_vel_y[BLOOD6];
                sprite_x[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] <=
sprite_x[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] + sprite_vel_x[BLOOD7];
                sprite_y[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] <=
sprite_y[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] + sprite_vel_y[BLOOD7];
                sprite_x[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] <=
sprite_x[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] + sprite_vel_x[BLOOD8];
                sprite_y[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] <=
sprite_y[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] + sprite_vel_y[BLOOD8];



    if (kirby_health == -100 && dedede_health == -100) begin
                // ----------------------------- both explode ------------
                b_frame_count[1] <= (b_frame_count[1] == 0) ? 0 : b_frame_count[1]
- 1;
                if (!fataled) begin
                        fataled <= 1;

                   sprite_display[KIRBY] <= 0;
                        sprite_display[DEDEDE] <= 0;

                        sprite_x[HURT2*posbits + posbits - 1 : HURT2*posbits] <=
sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_x_2;
                        sprite_y[HURT2*posbits + posbits - 1 : HURT2*posbits] <=
sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_y_2;
                        sprite_data[HURT2*4 + 3: HURT2*4] <= 1;
                        sprite_orientation[HURT2] <= sprite_orientation[KIRBY];
                        blood_counter[1] <= 0;
                        b_frame_count[1] <= 8'd15;

                        sprite_display[BLOOD8:BLOOD1] <= 8'b1111_1111;
                        sprite_data[BLOOD5] <= 4'b1101;
                        sprite_data[BLOOD6] <= 4'b1101;
                        sprite_data[BLOOD7] <= 4'b1101;
                        sprite_data[BLOOD8] <= 4'b1101;
                        sprite_vel_x[BLOOD5] <= -4;
                        sprite_vel_y[BLOOD5] <= -2;
                        sprite_vel_x[BLOOD6] <= -3;
                        sprite_vel_y[BLOOD6] <= -3;
                        sprite_vel_x[BLOOD7] <= 5;
                        sprite_vel_y[BLOOD7] <= -3;
                        sprite_vel_x[BLOOD8] <= 2;
                        sprite_vel_y[BLOOD8] <= -4;
                        sprite_x[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] <=
sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_x;
                        sprite_y[BLOOD5*posbits + posbits - 1 : BLOOD5*posbits] <=
sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_y;
                    sprite_x[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] <=
sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_x;
                    sprite_y[BLOOD6*posbits + posbits - 1 : BLOOD6*posbits] <=
sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_y;
                    sprite_x[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] <=
sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_x;
                    sprite_y[BLOOD7*posbits + posbits - 1 : BLOOD7*posbits] <=
sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_y;
                    sprite_x[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] <=
sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_x;
                    sprite_y[BLOOD8*posbits + posbits - 1 : BLOOD8*posbits] <=
sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_y;
```

```
                        sprite_data[BLOOD1] <= 4'b1001;
                        sprite_data[BLOOD2] <= 4'b1001;
                        sprite_data[BLOOD3] <= 4'b1001;
                        sprite_data[BLOOD4] <= 4'b1001;
                        sprite_vel_x[BLOOD1] <= -5;
                        sprite_vel_y[BLOOD1] <= -3;
                        sprite_vel_x[BLOOD2] <= -3;
                        sprite_vel_y[BLOOD2] <= -3;
                        sprite_vel_x[BLOOD3] <= 5;
                        sprite_vel_y[BLOOD3] <= -1;
                        sprite_vel_x[BLOOD4] <= 2;
                        sprite_vel_y[BLOOD4] <= -4;
                        sprite_x[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_x;
                        sprite_y[BLOOD1*posbits + posbits - 1 : BLOOD1*posbits] <=
    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_y;
                    sprite_x[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_x;
                    sprite_y[BLOOD2*posbits + posbits - 1 : BLOOD2*posbits] <=
    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_y;
                    sprite_x[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_x;
                    sprite_y[BLOOD3*posbits + posbits - 1 : BLOOD3*posbits] <=
    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_y;
                    sprite_x[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_x;
                    sprite_y[BLOOD4*posbits + posbits - 1 : BLOOD4*posbits] <=
    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_y;
              end else begin

                        if (blood_counter[1] != blood_life_2) begin
                                sprite_display[HURT2] <= 1;
                                blood_counter[1] <= blood_counter[1] + 1;
                                if(sprite_data[HURT2*4 + 3: HURT2*4] == 2 &&
    b_frame_count[1] == 0)
                                        sprite_data[HURT2*4 + 3: HURT2*4] <= 3;
                                else if(sprite_data[HURT2*4 + 3: HURT2*4] == 1 &&
    b_frame_count[1] == 0) begin
                                        sprite_data[HURT2*4 + 3: HURT2*4] <= 2;
                                        b_frame_count[1] <= 8'd15;
                                end
                        end else
                                sprite_display[HURT2] <= 0;
                    end
end else if (kirby_health == -100 || (kirby_health < dedede_health && enable_fatal))
    begin
              // ------------------------------ dedede fatals kirby ----------
                    b_frame_count[1] <= (b_frame_count[1] == 0) ? 0 : b_frame_count[1]
    - 1;
                    d_frame_count <= (d_frame_count == 0) ? 0 : d_frame_count - 1;
                    if (!fataled) begin
                            fataled <= 1;
                            sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] - 50;
                            sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] <=
    sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits];
                            sprite_x[posbits*HAMMER + posbits-1 : posbits*HAMMER] <=
    sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] - 50 +
    SPRITE_WIDTH[8*DEDEDE+7 : 8*DEDEDE] + d_melee_x_offset_1;


       sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER] <= sprite_y[KIRBY*posbits +
    posbits - 1 : KIRBY*posbits] + d_melee_y_offset_1 - 20;

                            sprite_orientation[DEDEDE] <= 1;
                            sprite_orientation[HAMMER] <= 1;
                            sprite_orientation[KIRBY] <= 0;
                            sprite_orientation[HURT2] <= 0;
                            sprite_display[HAMMER] <= 1;
```

```
                    sprite_data[DEDEDE*4 + 3: DEDEDE*4] <= 8;
                    sprite_data[HAMMER*4 + 3: HAMMER*4] <= 1;
                    sprite_data[KIRBY*4 + 3: KIRBY*4] <= 12;
                    d_frame_count <= 8'd35;
                end else begin
                    if (sprite_data[HAMMER*4 + 3: HAMMER*4] == 1 &&
d_frame_count == 0) begin
                        sprite_data[HAMMER*4 + 3: HAMMER*4] <= 7;
                        sprite_data[KIRBY*4 + 3: KIRBY*4] <= 13;
                        sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER]
<= sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER] + 10;
                        d_frame_count <= 8'd35;
                    end else if (sprite_data[HAMMER*4 + 3: HAMMER*4] == 7 &&
sprite_display[KIRBY] && d_frame_count == 0) begin
                        sprite_display[KIRBY] <= 0;
                        sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER]
<= sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER] + 10;

                        sprite_x[HURT2*posbits + posbits - 1 : HURT2*posbits]
<= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_x_2;
                        sprite_y[HURT2*posbits + posbits - 1 : HURT2*posbits]
<= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] + k_blood_offset_y_2;
                        sprite_data[HURT2*4 + 3: HURT2*4] <= 1;
                        sprite_orientation[HURT2] <=
sprite_orientation[KIRBY];
                        blood_counter[1] <= 0;
                        b_frame_count[1] <= 8'd15;

                    sprite_display[BLOOD8:BLOOD1] <= 8'b1111_1111;
                        sprite_data[BLOOD1] <= 4'b1001;
                        sprite_data[BLOOD2] <= 4'b1001;
                        sprite_data[BLOOD3] <= 4'b1001;
                        sprite_data[BLOOD4] <= 4'b1001;
                        sprite_vel_x[BLOOD1] <= -5;
                        sprite_vel_y[BLOOD1] <= -3;
                        sprite_vel_x[BLOOD2] <= -3;
                        sprite_vel_y[BLOOD2] <= -3;
                        sprite_vel_x[BLOOD3] <= 5;
                        sprite_vel_y[BLOOD3] <= -1;
                        sprite_vel_x[BLOOD4] <= 2;
                        sprite_vel_y[BLOOD4] <= -4;
                        sprite_x[BLOOD1*posbits + posbits - 1 :
BLOOD1*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                        sprite_y[BLOOD1*posbits + posbits - 1 :
BLOOD1*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                        sprite_x[BLOOD2*posbits + posbits - 1 :
BLOOD2*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                        sprite_y[BLOOD2*posbits + posbits - 1 :
BLOOD2*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                        sprite_x[BLOOD3*posbits + posbits - 1 :
BLOOD3*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                        sprite_y[BLOOD3*posbits + posbits - 1 :
BLOOD3*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                        sprite_x[BLOOD4*posbits + posbits - 1 :
BLOOD4*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_x;
                        sprite_y[BLOOD4*posbits + posbits - 1 :
BLOOD4*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
k_blood_offset_y;
                        sprite_data[BLOOD5] <= 4'b0101;
                        sprite_data[BLOOD6] <= 4'b0101;
                        sprite_data[BLOOD7] <= 4'b0101;
                        sprite_data[BLOOD8] <= 4'b0101;
                        sprite_vel_x[BLOOD5] <= -4;
```

```
                                sprite_vel_y[BLOOD5] <= -2;
                                sprite_vel_x[BLOOD6] <= -3;
                                sprite_vel_y[BLOOD6] <= -3;
                                sprite_vel_x[BLOOD7] <= 5;
                                sprite_vel_y[BLOOD7] <= -3;
                                sprite_vel_x[BLOOD8] <= 2;
                                sprite_vel_y[BLOOD8] <= -4;
                                sprite_x[BLOOD5*posbits + posbits - 1 :
    BLOOD5*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_x;
                                sprite_y[BLOOD5*posbits + posbits - 1 :
    BLOOD5*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_y;
                                sprite_x[BLOOD6*posbits + posbits - 1 :
    BLOOD6*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_x;
                                sprite_y[BLOOD6*posbits + posbits - 1 :
    BLOOD6*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_y;
                                sprite_x[BLOOD7*posbits + posbits - 1 :
    BLOOD7*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_x;
                                sprite_y[BLOOD7*posbits + posbits - 1 :
    BLOOD7*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_y;
                                sprite_x[BLOOD8*posbits + posbits - 1 :
    BLOOD8*posbits] <= sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_x;
                                sprite_y[BLOOD8*posbits + posbits - 1 :
    BLOOD8*posbits] <= sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] +
    k_blood_offset_y;

                        end else if (d_frame_count == 0) begin

                                if (blood_counter[1] != blood_life_2) begin
                                        sprite_display[HURT2] <= 1;
                                        blood_counter[1] <= blood_counter[1] + 1;
                                        if(sprite_data[HURT2*4 + 3: HURT2*4] == 2 &&
    b_frame_count[1] == 0) begin
                                                sprite_data[HURT2*4 + 3: HURT2*4] <=
    3;
                                                sprite_y[posbits*HAMMER+posbits-1 :
    posbits*HAMMER] <= sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER] + 10;
                                        end else if(sprite_data[HURT2*4 + 3: HURT2*4]
    == 1 && b_frame_count[1] == 0) begin
                                                sprite_data[HURT2*4 + 3: HURT2*4] <=
    2;
                                                sprite_y[posbits*HAMMER+posbits-1 :
    posbits*HAMMER] <= sprite_y[posbits*HAMMER+posbits-1 : posbits*HAMMER] + 10;
                                                sprite_y[posbits*DEDEDE+posbits-1 :
    posbits*DEDEDE] <= sprite_y[posbits*DEDEDE+posbits-1 : posbits*DEDEDE] + 15;
                                                b_frame_count[1] <= 8'd15;
                                        end
                                end else
                                        sprite_display[HURT2] <= 0;
                        end
                end
end else if (dedede_health == -100 || (kirby_health > dedede_health && enable_fatal))
    begin
                d_frame_count <= (d_frame_count == 0) ? 0 : d_frame_count - 1;
            // ----------------------------- kirby fatals dedede --------
                b_frame_count[1] <= (b_frame_count[1] == 0) ? 0 : b_frame_count[1]
    - 1;
                if (!fataled) begin
                        fataled <= 1;
                        sprite_x[KIRBY*posbits + posbits - 1 : KIRBY*posbits] <=
    sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] - 100;
                        sprite_y[KIRBY*posbits + posbits - 1 : KIRBY*posbits] <=
    sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits];
```

```
                    sprite_orientation[DEDEDE] <= 0;
                    sprite_orientation[KIRBY] <= 1;
                    sprite_data[DEDEDE*4 + 3: DEDEDE*4] <= 12;
                    sprite_data[KIRBY*4 + 3: KIRBY*4] <= 10;
                    d_frame_count <= 8'd100;
                end else begin
                    if (sprite_data[KIRBY*4 + 3: KIRBY*4] == 10 && d_frame_count
!= 0) begin
                        sprite_x[DEDEDE*posbits + posbits - 1 :
DEDEDE*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] - 1;
                    end else if (sprite_data[KIRBY*4 + 3: KIRBY*4] == 10 &&
d_frame_count == 0) begin
                        sprite_data[KIRBY*4 + 3: KIRBY*4] <= 14;
                        sprite_display[DEDEDE] <= 0;

                        sprite_x[HURT2*posbits + posbits - 1 : HURT2*posbits]
<= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_x_2;
                        sprite_y[HURT2*posbits + posbits - 1 : HURT2*posbits]
<= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] + d_blood_offset_y_2;
                        sprite_data[HURT2*4 + 3: HURT2*4] <= 1;
                        sprite_orientation[HURT2] <=
sprite_orientation[DEDEDE];
                        blood_counter[1] <= 0;
                        b_frame_count[1] <= 8'd15;

                    sprite_display[BLOOD8:BLOOD1] <= 8'b1111_1111;
                        sprite_data[BLOOD1] <= 4'b1101;
                        sprite_data[BLOOD2] <= 4'b1101;
                        sprite_data[BLOOD3] <= 4'b1101;
                        sprite_data[BLOOD4] <= 4'b1101;
                        sprite_vel_x[BLOOD1] <= -5;
                        sprite_vel_y[BLOOD1] <= -3;
                        sprite_vel_x[BLOOD2] <= -3;
                        sprite_vel_y[BLOOD2] <= -3;
                        sprite_vel_x[BLOOD3] <= 5;
                        sprite_vel_y[BLOOD3] <= -1;
                        sprite_vel_x[BLOOD4] <= 2;
                        sprite_vel_y[BLOOD4] <= -4;
                        sprite_x[BLOOD1*posbits + posbits - 1 :
BLOOD1*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_x;
                        sprite_y[BLOOD1*posbits + posbits - 1 :
BLOOD1*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_y;
                        sprite_x[BLOOD2*posbits + posbits - 1 :
BLOOD2*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_x;
                        sprite_y[BLOOD2*posbits + posbits - 1 :
BLOOD2*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_y;
                        sprite_x[BLOOD3*posbits + posbits - 1 :
BLOOD3*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_x;
                        sprite_y[BLOOD3*posbits + posbits - 1 :
BLOOD3*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_y;
                        sprite_x[BLOOD4*posbits + posbits - 1 :
BLOOD4*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_x;
                        sprite_y[BLOOD4*posbits + posbits - 1 :
BLOOD4*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
d_blood_offset_y;
                        sprite_data[BLOOD5] <= 4'b0101;
                        sprite_data[BLOOD6] <= 4'b0101;
                        sprite_data[BLOOD7] <= 4'b0101;
                        sprite_data[BLOOD8] <= 4'b0101;
                        sprite_vel_x[BLOOD5] <= -4;
                        sprite_vel_y[BLOOD5] <= -2;
                        sprite_vel_x[BLOOD6] <= -3;
                        sprite_vel_y[BLOOD6] <= -3;
```

```
                                        sprite_vel_x[BLOOD7] <= 5;
                                        sprite_vel_y[BLOOD7] <= -3;
                                        sprite_vel_x[BLOOD8] <= 2;
                                        sprite_vel_y[BLOOD8] <= -4;
                                        sprite_x[BLOOD5*posbits + posbits - 1 :
            BLOOD5*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_x;
                                        sprite_y[BLOOD5*posbits + posbits - 1 :
            BLOOD5*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_y;
                                        sprite_x[BLOOD6*posbits + posbits - 1 :
            BLOOD6*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_x;
                                        sprite_y[BLOOD6*posbits + posbits - 1 :
            BLOOD6*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_y;
                                        sprite_x[BLOOD7*posbits + posbits - 1 :
            BLOOD7*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_x;
                                        sprite_y[BLOOD7*posbits + posbits - 1 :
            BLOOD7*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_y;
                                        sprite_x[BLOOD8*posbits + posbits - 1 :
            BLOOD8*posbits] <= sprite_x[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_x;
                                        sprite_y[BLOOD8*posbits + posbits - 1 :
            BLOOD8*posbits] <= sprite_y[DEDEDE*posbits + posbits - 1 : DEDEDE*posbits] +
            d_blood_offset_y;
                                    end else begin
                                        if (blood_counter[1] != blood_life_2) begin
                                            sprite_display[HURT2] <= 1;
                                            blood_counter[1] <= blood_counter[1] + 1;
                                            if(sprite_data[HURT2*4 + 3: HURT2*4] == 2 &&
            b_frame_count[1] == 0)
                                                sprite_data[HURT2*4 + 3: HURT2*4] <=
            3;
                                            else if(sprite_data[HURT2*4 + 3: HURT2*4] ==
            1 && b_frame_count[1] == 0) begin
                                                sprite_data[HURT2*4 + 3: HURT2*4] <=
            2;
                                                b_frame_count[1] <= 8'd15;
                                            end
                                        end else
                                            sprite_display[HURT2] <= 0;
                                    end
                                end
            end else if (kirby_health < dedede_health) begin
                                    sprite_data[KIRBY*4 + 3: KIRBY*4] <= 13;
            end else begin
                                    sprite_data[DEDEDE*4 + 3: DEDEDE*4] <= 13;
            end

        end

    end


endmodule
```

Appendix XI – Sprite Module

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

//
// File:   sprite_module.v
// Date:   12/9/08
// Author: Daniel Gerber
//
// Sprite Module for Kirby Laser Attack
//
//////////////////////////////////////////////////////////////////////////////////

module sprite_module(clk, reset, hcount, vcount,

        platform1x, platform1y, platform2x, platform2y, platform3x, platform3y,

                platform4x, platform4y, platform5x, platform5y, platform6x, platform6y,

                kirbyx, kirbyy, dededex, dededey,

                smallbloodx, smallbloody, bigbloodx, bigbloody,

                chunk1x, chunk1y, chunk2x, chunk2y, chunk3x, chunk3y, chunk4x, chunk4y,

                chunk5x, chunk5y, chunk6x, chunk6y, chunk7x, chunk7y, chunk8x, chunk8y,
                swordx, swordy, hammerx, hammery,

                kproj1x, kproj1y, kproj2x, kproj2y,

                dproj1x, dproj1y, dproj2x, dproj2y,

                kirbyid, dededeid, swordid, hammerid,

                smallbloodid, bigbloodid,

                chunk1id, chunk2id, chunk3id, chunk4id, chunk5id, chunk6id, chunk7id, chunk8id,

                kproj1id, kproj2id, dproj1id, dproj2id,

                kirbyhealth, dededehealth,

        color,

                kvsd, kvsh, dvss, svsh, kvsb, kvsg, dvssh, dvsh);



   input clk, reset; //clock and reset

   input [10:0] hcount, vcount; // hcount and vcount signals
   input [9:0] platform1x, platform1y, platform2x, platform2y, platform3x, platform3y,

               platform4x, platform4y, platform5x, platform5y, platform6x, platform6y; //
platform coordinate inputs

   input [9:0] kirbyx, kirbyy, dededex, dededey; // kirby and dedede coordinate inputs

        input [9:0] smallbloodx, smallbloody, bigbloodx, bigbloody; // blood animation coordinate
inputs

        input [9:0] chunk1x, chunk1y, chunk2x, chunk2y, chunk3x, chunk3y, chunk4x, chunk4y,
                chunk5x, chunk5y, chunk6x, chunk6y, chunk7x, chunk7y, chunk8x, chunk8y; // bloody
chunk coordinate inputs
   input [9:0] swordx, swordy, hammerx, hammery; // weapon coordinate inputs

   input [9:0] kproj1x, kproj1y, kproj2x, kproj2y, dproj1x, dproj1y, dproj2x, dproj2y; //
projectile coordinate inputs
```

```
   input [4:0] kirbyid, dededeid, swordid, hammerid; // character and weapon id inputs

       input [2:0] smallbloodid, bigbloodid, dproj1id, dproj2id, kproj1id, kproj2id; // blood and
projectile id inputs

       input [3:0] chunk1id, chunk2id, chunk3id, chunk4id, chunk5id, chunk6id, chunk7id,
chunk8id; // bloody chunk id inputs

       input [6:0] dededehealth, kirbyhealth; // health inputs for the health meter

   output [7:0] color; // output of what color the pixel given by hcount and vcount should be

       output kvsd, kvsh, dvss, svsh, kvsb, kvsg, dvssh, dvsh; // outputs for pixel based
collisions



       // color submodule outputs

       wire [8:0] dededecolor, kirbycolor, swordcolor, hammercolor, bigbloodcolor,
smallbloodcolor,

               boomerangcolor, gordocolor, hadokencolor, shurikencolor, kirbyhealthcolor,
dededehealthcolor,

               chunk1color, chunk2color, chunk3color, chunk4color, chunk5color, chunk6color,
chunk7color, chunk8color,

               platform1color, platform2color, platform3color, platform4color, platform5color,
platform6color, floorcolor;

       // registers for counters for collision detection

       reg [1:0] kvsdcount, dvsscount, kvshcount, svshcount, kvsbcount, kvsgcount, dvsshcount,
dvshcount;

       reg vcountflag;



       // instances of each sprite,

       // each sprite has multiple brams in it and the sprite submodule decides which bram to use

       // bram to use is specified by the id input

       kirby cutie(clk, reset, kirbyx, kirbyy, hcount, vcount, kirbyid, kirbycolor);

       dedede fatty(clk, reset, dededex, dededey, hcount, vcount, dededeid, dededecolor);
       sword slicer(clk, reset, swordx, swordy, hcount, vcount, swordid, swordcolor);

       hammer pounder(clk, reset, hammerx, hammery, hcount, vcount, hammerid, hammercolor);

       bigblood corpseexplode(clk, reset, bigbloodx, bigbloody, hcount, vcount, bigbloodid,
bigbloodcolor);

       smallblood pinprick(clk, reset, smallbloodx, smallbloody, hcount, vcount, smallbloodid,
smallbloodcolor);
       boomerang zora(clk, reset, dproj2x, dproj2y, hcount, vcount, dproj2id, boomerangcolor);
       gordo pointy(clk, reset, dproj1x, dproj1y, hcount, vcount, dproj1id, gordocolor);
       hadoken douglass(clk, reset, kproj1x, kproj1y, hcount, vcount, kproj1id, hadokencolor);
       shuriken naruto(clk, reset, kproj2x, kproj2y, hcount, vcount, kproj2id, shurikencolor);

       bloodchunk gore1(clk, reset, chunk1x, chunk1y, hcount, vcount, chunk1id, chunk1color);
       bloodchunk gore2(clk, reset, chunk2x, chunk2y, hcount, vcount, chunk2id, chunk2color);
       bloodchunk gore3(clk, reset, chunk3x, chunk3y, hcount, vcount, chunk3id, chunk3color);
       bloodchunk gore4(clk, reset, chunk4x, chunk4y, hcount, vcount, chunk4id, chunk4color);
       bloodchunk gore5(clk, reset, chunk5x, chunk5y, hcount, vcount, chunk5id, chunk5color);
       bloodchunk gore6(clk, reset, chunk6x, chunk6y, hcount, vcount, chunk6id, chunk6color);
```

```
        bloodchunk gore7(clk, reset, chunk7x, chunk7y, hcount, vcount, chunk7id, chunk7color);
        bloodchunk gore8(clk, reset, chunk8x, chunk8y, hcount, vcount, chunk8id, chunk8color);
        kirbyhealthmeter redmeter(clk, reset, 10, 660, hcount, vcount, kirbyhealth,
kirbyhealthcolor);
        dededehealthmeter bluemeter(clk, reset, 900, 660, hcount, vcount, dededehealth,
dededehealthcolor);
        platforms column1(clk, reset, platform1x, platform1y, hcount, vcount, 1, platform1color);
        platforms column2(clk, reset, platform2x, platform2y, hcount, vcount, 2, platform2color);
        platforms column3(clk, reset, platform3x, platform3y, hcount, vcount, 3, platform3color);
        platforms column4(clk, reset, platform4x, platform4y, hcount, vcount, 4, platform4color);
        platforms column5(clk, reset, platform5x, platform5y, hcount, vcount, 5, platform5color);
        platforms column6(clk, reset, platform6x, platform6y, hcount, vcount, 6, platform6color);

        rectangle #(.WIDTH(780),.HEIGHT(105),.COLOR(9'b011110000))
floor(.x(125),.hcount(hcount),.y(660),.vcount(vcount),.pixel(floorcolor));

        // logic to determine which color to be displayed should there be overlapping of sprites

        // ordered from top to bottom in priority

        assign color =

                ~chunk1color[8] ? chunk1color[7:0] :
                ~chunk2color[8] ? chunk2color[7:0] :
                ~chunk3color[8] ? chunk3color[7:0] :
                ~chunk4color[8] ? chunk4color[7:0] :
                ~chunk5color[8] ? chunk5color[7:0] :
                ~chunk6color[8] ? chunk6color[7:0] :
                ~chunk7color[8] ? chunk7color[7:0] :
                ~chunk8color[8] ? chunk8color[7:0] :
                ~bigbloodcolor[8] ? bigbloodcolor[7:0] :
                ~smallbloodcolor[8] ? smallbloodcolor[7:0] :
                ~kirbyhealthcolor[8] ? kirbyhealthcolor[7:0] :
                ~dededehealthcolor[8] ? dededehealthcolor[7:0] :

                ~kirbycolor[8] ? kirbycolor[7:0] :

                ~dededecolor[8] ? dededecolor[7:0] :

                ~swordcolor[8] ? swordcolor[7:0] :
                ~hammercolor[8] ? hammercolor[7:0] :

                ~hadokencolor[8] ? hadokencolor[7:0] :
                ~gordocolor[8] ? gordocolor[7:0] :
                ~shurikencolor[8] ? shurikencolor[7:0] :
                ~boomerangcolor[8] ? boomerangcolor[7:0] :
                ~platform1color[8] ? platform1color[7:0] :
                ~platform2color[8] ? platform2color[7:0] :
                ~platform3color[8] ? platform3color[7:0] :
                ~platform4color[8] ? platform4color[7:0] :
                ~platform5color[8] ? platform5color[7:0] :
                ~platform6color[8] ? platform6color[7:0] :

                ~floorcolor[8] ? floorcolor[7:0] :

                8'b00000000;


        // logic for collision outputs

        assign kvsd = (kvsdcount == 0) ? 0 : 1;

        assign kvsh = (kvshcount == 0) ? 0 : 1;
        assign dvss = (dvsscount == 0) ? 0 : 1;
        assign svsh = (svshcount == 0) ? 0 : 1;
        assign kvsb = (kvsbcount == 0) ? 0 : 1;
        assign kvsg = (kvsgcount == 0) ? 0 : 1;
        assign dvssh = (dvsshcount == 0) ? 0 : 1;
        assign dvsh = (dvshcount == 0) ? 0 : 1;
```

```
	// counters to allow collision signal to remain valid for several screen refresh cycles
	always @ (posedge clk) begin
		if(vcount == 0 && vcountflag == 0)
			vcountflag <= 1;
		else
			vcountflag <= 0;
		if(vcountflag) begin
			kvsdcount <= (kvsdcount == 0) ? kvsdcount : kvsdcount + 1;
			kvshcount <= (kvshcount == 0) ? kvshcount : kvshcount + 1;
			dvsscount <= (dvsscount == 0) ? dvsscount : dvsscount + 1;
			svshcount <= (svshcount == 0) ? svshcount : svshcount + 1;
			kvsbcount <= (kvsbcount == 0) ? kvsbcount : kvsbcount + 1;
			kvsgcount <= (kvsgcount == 0) ? kvsgcount : kvsgcount + 1;
			dvsshcount <= (dvsshcount == 0) ? dvsshcount : dvsshcount + 1;
			dvshcount <= (dvshcount == 0) ? dvshcount : dvshcount + 1;
		end
		else begin
			kvsdcount <= (~dededecolor[8] && ~kirbycolor[8]) ? 1 : kvsdcount;
			kvshcount <= (~hammercolor[8] && ~kirbycolor[8]) ? 1 : kvshcount;
			dvsscount <= (~dededecolor[8] && ~swordcolor[8]) ? 1 : dvsscount;
			svshcount <= (~hammercolor[8] && ~swordcolor[8]) ? 1 : svshcount;
			kvsbcount <= (~boomerangcolor[8] && ~kirbycolor[8]) ? 1 : kvsbcount;
			kvsgcount <= (~gordocolor[8] && ~kirbycolor[8]) ? 1 : kvsgcount;
			dvsshcount <= (~dededecolor[8] && ~shurikencolor[8]) ? 1 : dvsshcount;
			dvshcount <= (~dededecolor[8] && ~hadokencolor[8]) ? 1 : dvshcount;
		end
	end


endmodule


// dedede sprite module
//----------------------------------------------------------------------------------

// this module will be fully commented
// but the other sprite modules will only be commented for big differences in code
module dedede(clk, reset, x, y, hcount, vcount, id, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [4:0] id;
	output reg [8:0] color;

	// sprite dimensions
```

```
        parameter WIDTH = 60;

        parameter HEIGHT = 60;



        // input and outputs for image brams
        reg [11:0] addr;
        wire [7:0] dout1, dout2, dout3, dout4, dout5, dout6, dout7, dout8,

                dout9, dout10, dout12, dout13; // dout11,

        // helper regs for processing output

        reg [8:0] precolor;

        reg ready;

        // image bram instances
        dedede1 image1(addr, clk, dout1);

        dedede2 image2(addr, clk, dout2);

        dedede3 image3(addr, clk, dout3);
        dedede4 image4(addr, clk, dout4);
        dedede5 image5(addr, clk, dout5);
        dedede6 image6(addr, clk, dout6);
        dedede7 image7(addr, clk, dout7);
        dedede8 image8(addr, clk, dout8);
        dedede9 image9(addr, clk, dout9);
        dedede10 image10(addr, clk, dout10);
        //dedede11 image11(addr, clk, dout11);
        dedede12 image12(addr, clk, dout12);
        dedede13 image13(addr, clk, dout13);

        always @ (x or y or hcount or vcount) begin
                // check if hcount and vcount are within sprite's box

                if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin

                        if(id[4:4] == 0) // sprite orientation set by highest order id bit

                                // read sprite forwards

                                addr <= (vcount - y) * WIDTH + (hcount - x);

                        else

                                // read sprite backwards

                                addr <= (vcount - y + 1) * WIDTH - (hcount - x);

                        ready <= 1; // set signal for reading output

                end

                else begin

                        addr <= 0; // reset address line

                        ready <= 0;

                end
        end



        always @ (posedge clk) begin
```

```
		if(ready == 1) begin

			// decide which bram to read from to determine color

			case(id[3:0])

				1: precolor <= {0, dout1};

				2: precolor <= {0, dout2};

				3: precolor <= {0, dout3};

				4: precolor <= {0, dout4};

				5: precolor <= {0, dout5};
				6: precolor <= {0, dout6};
				7: precolor <= {0, dout7};
				8: precolor <= {0, dout8};
				9: precolor <= {0, dout9};
				10: precolor <= {0, dout10};
				//11: precolor <= {0, dout11};
				12: precolor <= {0, dout12};
				13: precolor <= {0, dout13};
				default: precolor <= {1, 8'b00000000};

			endcase

		end

		else precolor <= {1, 8'b00000000};


		// process image, turn all pure green into invisible

		// all of the images were processed using GIMP and set so that pure green should
be invisible

		if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))

			color <= {1, 8'b00000000};

		else color <= precolor;

	end
endmodule


// kirby sprite module
----------------------------------------------------------------------------------

module kirby(clk, reset, x, y, hcount, vcount, id, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [4:0] id;
	output reg [8:0] color;

	parameter WIDTH = 60;
	parameter HEIGHT = 60;

	reg [11:0] addr;
	wire [7:0] dout1, dout2, dout3, dout4, dout5, dout6, dout7, dout8,

		dout9, dout10, dout12, dout13, dout14; // dout11,
	reg [8:0] precolor;
	reg ready;
	kirby1 image1(addr, clk, dout1);
	kirby2 image2(addr, clk, dout2);
```

```
        kirby3 image3(addr, clk, dout3);
        kirby4 image4(addr, clk, dout4);
        kirby5 image5(addr, clk, dout5);
        kirby6 image6(addr, clk, dout6);
        kirby7 image7(addr, clk, dout7);
        kirby8 image8(addr, clk, dout8);
        kirby9 image9(addr, clk, dout9);
        kirby10 image10(addr, clk, dout10);
        //kirby11 image11(addr, clk, dout11);
        kirby12 image12(addr, clk, dout12);
        kirby13 image13(addr, clk, dout13);

        kirby14 image14(addr, clk, dout14);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount < x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        if(id[4:4] == 0)
                                addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                        else
                                addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
                        ready <= 1;
                end
                else begin
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1) begin
                        case(id[3:0])
                                1: precolor <= {0, dout1};
                                2: precolor <= {0, dout2};
                                3: precolor <= {0, dout3};
                                4: precolor <= {0, dout4};
                                5: precolor <= {0, dout5};
                                6: precolor <= {0, dout6};
                                7: precolor <= {0, dout7};
                                8: precolor <= {0, dout8};
                                9: precolor <= {0, dout9};
                                10: precolor <= {0, dout10};
                                //11: precolor <= {0, dout11};
                                12: precolor <= {0, dout12};
                                13: precolor <= {0, dout13};

                                14: precolor <= {0, dout14};
                                default: precolor <= {1, 8'b00000000};
                        endcase
                end
                else precolor <= {1, 8'b00000000};

                if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
                else color <= precolor;
        end
endmodule



// sword sprite module
--------------------------------------------------------------------------------

module sword(clk, reset, x, y, hcount, vcount, id, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [4:0] id;
        output reg [8:0] color;
```

```
	parameter WIDTH = 45;
	parameter HEIGHT = 90;

	reg [11:0] addr;
	wire [7:0] dout1, dout2, dout3, dout4, dout5, dout6;
	reg [8:0] precolor;
	reg ready;
	sword1 image1(addr, clk, dout1);
	sword2 image2(addr, clk, dout2);
	sword3 image3(addr, clk, dout3);
	sword4 image4(addr, clk, dout4);
	sword5 image5(addr, clk, dout5);
	sword6 image6(addr, clk, dout6);

	always @ (x or y or hcount or vcount) begin
		// sword id 1 to 3 has width 45 and height 90

		if((id[3:0] <= 3) &&
			(hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y +
HEIGHT)) begin
			if(id[4:4] == 0)
				addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
			else
				addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
			ready <= 1;
		end

		// sword id 4 to 6 has width 90 and height 45

		// therefore, all that must be done is to switch WIDTH and HEIGHT
		else if((id[3:0] > 3 && id[3:0] < 7) &&
			(hcount >= x && hcount <= x + HEIGHT) && (vcount >= y && vcount < y +
WIDTH)) begin
			if(id[4:4] == 0)
				addr <= (vcount - y) * HEIGHT + (hcount - x); //forwards
			else
				addr <= (vcount - y + 1) * HEIGHT - (hcount - x); //backwards
			ready <= 1;
		end
		else begin
			addr <= 0;
			ready <= 0;
		end

	end

	always @ (posedge clk) begin
		if(ready == 1) begin
			case(id[3:0])
				1: precolor <= {0, dout1};
				2: precolor <= {0, dout2};
				3: precolor <= {0, dout3};
				4: precolor <= {0, dout4};
				5: precolor <= {0, dout5};
				6: precolor <= {0, dout6};
				default: precolor <= {1, 8'b00000000};
			endcase
		end
		else precolor <= {1, 8'b00000000};

		if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
			color <= {1, 8'b00000000};
		else color <= precolor;
	end
endmodule


// hammer sprite module
--------------------------------------------------------------------------------
```

```
module hammer(clk, reset, x, y, hcount, vcount, id, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [4:0] id;
	output reg [8:0] color;

	parameter WIDTH = 45;
	parameter HEIGHT = 90;

	reg [11:0] addr;
	wire [7:0] dout1, dout2, dout3, dout4, dout5, dout6, dout7;
	reg [8:0] precolor;
	reg ready;
	hammer1 image1(addr, clk, dout1);
	hammer2 image2(addr, clk, dout2);
	hammer3 image3(addr, clk, dout3);
	hammer4 image4(addr, clk, dout4);
	hammer5 image5(addr, clk, dout5);
	hammer6 image6(addr, clk, dout6);

	hammer7 image7(addr, clk, dout7);

	always @ (x or y or hcount or vcount) begin
		if((id[3:0] <= 3 || id[3:0] == 7) &&
			(hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y +
HEIGHT)) begin
			if(id[4:4] == 0)
				addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
			else
				addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
			ready <= 1;
		end
		else if((id[3:0] > 3 && id[3:0] < 7) &&

			(hcount >= x && hcount <= x + HEIGHT) && (vcount >= y && vcount < y +
WIDTH)) begin
			if(id[4:4] == 0)
				addr <= (vcount - y) * HEIGHT + (hcount - x); //forwards
			else
				addr <= (vcount - y + 1) * HEIGHT - (hcount - x); //backwards
			ready <= 1;
		end
		else begin
			addr <= 0;
			ready <= 0;
		end
	end

	always @ (posedge clk) begin
		if(ready == 1) begin
			case(id[3:0])
				1: precolor <= {0, dout1};
				2: precolor <= {0, dout2};
				3: precolor <= {0, dout3};
				4: precolor <= {0, dout4};
				5: precolor <= {0, dout5};
				6: precolor <= {0, dout6};

				7: precolor <= {0, dout7};
				default: precolor <= {1, 8'b00000000};
			endcase
		end
		else precolor <= {1, 8'b00000000};

		if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
			color <= {1, 8'b00000000};
		else color <= precolor;
	end
endmodule
```

```
// smallblood sprite module
--------------------------------------------------------------------------------
module smallblood(clk, reset, x, y, hcount, vcount, id, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [2:0] id;
	output reg [8:0] color;

	parameter WIDTH = 30;
	parameter HEIGHT = 30;

	reg [9:0] addr;
	wire [7:0] dout1, dout2, dout3;
	reg [8:0] precolor;
	reg ready;
	smallblood1 image1(addr, clk, dout1);
	smallblood2 image2(addr, clk, dout2);
	smallblood3 image3(addr, clk, dout3);

	always @ (x or y or hcount or vcount) begin
		if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
			if(id[2:2] == 0)
				addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
			else
				addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
			ready <= 1;
		end
		else begin
			addr <= 0;
			ready <= 0;
		end
	end

	always @ (posedge clk) begin
		if(ready == 1) begin
			case(id[1:0])
				1: precolor <= {0, dout1};
				2: precolor <= {0, dout2};
				3: precolor <= {0, dout3};
				default: precolor <= {1, 8'b00000000};
			endcase
		end
		else precolor <= {1, 8'b00000000};

		if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
			color <= {1, 8'b00000000};
		else color <= precolor;
	end
endmodule



// bigblood sprite module
--------------------------------------------------------------------------------

module bigblood(clk, reset, x, y, hcount, vcount, id, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [2:0] id;
	output reg [8:0] color;

	parameter WIDTH = 120;
	parameter HEIGHT = 120;
```

```
        reg [13:0] addr;
        wire [7:0] dout1, dout2;//, dout3;
        reg [8:0] precolor;
        reg ready;
        bigblood1 image1(addr, clk, dout1);
        bigblood2 image2(addr, clk, dout2);
//      bigblood3 image3(addr, clk, dout3);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        if(id[2:2] == 0)
                                addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                        else
                                addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
                        ready <= 1;
                end
                else begin
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1) begin
                        case(id[1:0])
                                1: precolor <= {0, dout1};
                                2: precolor <= {0, dout2};
//                              3: precolor <= {0, dout3};
                                default: precolor <= {1, 8'b00000000};
                        endcase
                end
                else precolor <= {1, 8'b00000000};

                if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
                else color <= precolor;
        end
endmodule



// bloodchunk sprite module
//------------------------------------------------------------------------------

module bloodchunk(clk, reset, x, y, hcount, vcount, id, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [3:0] id;
        output reg [8:0] color;

        parameter WIDTH = 15;
        parameter HEIGHT = 15;

        reg [9:0] addr;
        wire [7:0] dout1;//, dout2, dout3;
        reg [8:0] precolor;
        reg ready;
        bloodchunk1 image1(addr, clk, dout1);
        //bloodchunk2 image2(addr, clk, dout2);
        //bloodchunk3 image3(addr, clk, dout3);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                ready <= 1;
                end
                else begin
```

```
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1) begin
                        case(id[1:0])
                                1: precolor <= {0, dout1};
                                //2: precolor <= {0, dout2};
                                //3: precolor <= {0, dout3};
                                default: precolor <= {1, 8'b00000000};
                        endcase
                end
                else precolor <= {1, 8'b00000000};


                if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};

                // in order to use the bloodchunk sprite for blood, chunks of kirby, or chunks of
dedede,

                // must process the id's top two bits which represent color

                // default color is dedede's chunks, but those colors can be set to red or pink

                else if(precolor[7:0] == 8'b01010011) begin

                        if(id[3:2] == 2'b01) color[7:0] <= 8'b11100000;

                        else if(id[3:2] == 2'b10) color[7:0] <= 8'b11110010;

                end
                else color <= precolor;
        end
endmodule



// hadoken sprite module
----------------------------------------------------------------------------------

module hadoken(clk, reset, x, y, hcount, vcount, id, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [2:0] id;
        output reg [8:0] color;

        parameter WIDTH = 60;
        parameter HEIGHT = 60;

        reg [11:0] addr;
        wire [7:0] dout1, dout2;
        reg [8:0] precolor;
        reg ready;
        hadoken1 image1(addr, clk, dout1);
        hadoken2 image2(addr, clk, dout2);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        if(id[2:2] == 0)
                                addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                        else
                                addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
                        ready <= 1;
                end
                else begin
```

```
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1) begin
                        case(id[1:0])
                                1: precolor <= {0, dout1};
                                2: precolor <= {0, dout2};
                                default: precolor <= {1, 8'b00000000};
                        endcase
                end
                else precolor <= {1, 8'b00000000};

                if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
                else color <= precolor;
        end
endmodule



// shuriken sprite module
----------------------------------------------------------------------------------

module shuriken(clk, reset, x, y, hcount, vcount, id, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [2:0] id;
        output reg [8:0] color;

        parameter WIDTH = 60;
        parameter HEIGHT = 60;

        reg [11:0] addr;
        wire [7:0] dout1, dout2;
        reg [8:0] precolor;
        reg ready;
        shuriken1 image1(addr, clk, dout1);
        shuriken2 image2(addr, clk, dout2);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        if(id[2:2] == 0)
                                addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                        else
                                addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
                        ready <= 1;
                end
                else begin
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1) begin
                        case(id[1:0])
                                1: precolor <= {0, dout1};
                                2: precolor <= {0, dout2};
                                default: precolor <= {1, 8'b00000000};
                        endcase
                end
                else precolor <= {1, 8'b00000000};

                if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
```

```
                else color <= precolor;
        end
endmodule



// gordo sprite module
//------------------------------------------------------------------------------

module gordo(clk, reset, x, y, hcount, vcount, id, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [2:0] id;
        output reg [8:0] color;

        parameter WIDTH = 60;
        parameter HEIGHT = 60;

        reg [11:0] addr;
        wire [7:0] dout1, dout2;
        reg [8:0] precolor;
        reg ready;
        gordo1 image1(addr, clk, dout1);
        gordo2 image2(addr, clk, dout2);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        if(id[2:2] == 0)
                                addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                        else
                                addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
                        ready <= 1;
                end
                else begin
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1) begin
                        case(id[1:0])
                                1: precolor <= {0, dout1};
                                2: precolor <= {0, dout2};
                                default: precolor <= {1, 8'b00000000};
                        endcase
                end
                else precolor <= {1, 8'b00000000};

                if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
                else color <= precolor;
        end
endmodule



// boomerang sprite module
//------------------------------------------------------------------------------

module boomerang(clk, reset, x, y, hcount, vcount, id, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [2:0] id;
        output reg [8:0] color;

        parameter WIDTH = 60;
```

```
	parameter HEIGHT = 60;

	reg [11:0] addr;
	wire [7:0] dout1, dout2;
	reg [8:0] precolor;
	reg ready;
	boomerang1 image1(addr, clk, dout1);
	boomerang2 image2(addr, clk, dout2);

	always @ (x or y or hcount or vcount) begin
		if((hcount >= x && hcount <= x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
			if(id[2:2] == 0)
				addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
			else
				addr <= (vcount - y + 1) * WIDTH - (hcount - x); //backwards
			ready <= 1;
		end
		else begin
			addr <= 0;
			ready <= 0;
		end
	end

	always @ (posedge clk) begin
		if(ready == 1) begin
			case(id[1:0])
				1: precolor <= {0, dout1};
				2: precolor <= {0, dout2};
				default: precolor <= {1, 8'b00000000};
			endcase
		end
		else precolor <= {1, 8'b00000000};

		if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
			color <= {1, 8'b00000000};
		else color <= precolor;
	end
endmodule



// kirbyhealthmeter sprite module
// ----------------------------------------------------------------------------------

module kirbyhealthmeter(clk, reset, x, y, hcount, vcount, health, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [6:0] health;
	output reg [8:0] color;

	parameter WIDTH = 115;
	parameter HEIGHT = 105;

	reg [13:0] addr;
	wire [7:0] dout1;
	reg [8:0] precolor;
	reg ready;
	kirbyhealth image1(addr, clk, dout1);

	always @ (x or y or hcount or vcount) begin
		if((hcount >= x && hcount < x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
			addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
			ready <= 1;
		end
		else begin
			addr <= 0;
			ready <= 0;
```

```
                end
        end

        always @ (posedge clk) begin
                if(ready == 1)
                        precolor <= {0, dout1};
                else precolor <= {1, 8'b00000000};


                // turn pure green into red to indicate amount of health within the capsule
                if((precolor[7:0] == 8'h1C) && (y + HEIGHT - vcount < health))

                        color <= 9'b011100000;
                else if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
                else color <= precolor;
        end
endmodule



// dededehealthmeter sprite module
----------------------------------------------------------------------------------

module dededehealthmeter(clk, reset, x, y, hcount, vcount, health, color);
        input clk, reset;
        input [9:0] x, y, vcount;
        input [10:0] hcount;
        input [6:0] health;
        output reg [8:0] color;

        parameter WIDTH = 115;
        parameter HEIGHT = 105;

        reg [13:0] addr;
        wire [7:0] dout1;
        reg [8:0] precolor;
        reg ready;
        dededehealth image1(addr, clk, dout1);

        always @ (x or y or hcount or vcount) begin
                if((hcount >= x && hcount < x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
                        addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
                        ready <= 1;
                end
                else begin
                        addr <= 0;
                        ready <= 0;
                end
        end

        always @ (posedge clk) begin
                if(ready == 1)
                        precolor <= {0, dout1};
                else precolor <= {1, 8'b00000000};

                if((precolor[7:0] == 8'h1C) && (y + HEIGHT - vcount < health))
                        color <= 9'b000000011;
                else if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
                        color <= {1, 8'b00000000};
                else color <= precolor;
        end
endmodule



// platform sprite module
----------------------------------------------------------------------------------
```

```
module platforms(clk, reset, x, y, hcount, vcount, id, color);
	input clk, reset;
	input [9:0] x, y, vcount;
	input [10:0] hcount;
	input [3:0] id;
	output reg [8:0] color;

	parameter WIDTH = 90;
	parameter HEIGHT = 30;

	reg [11:0] addr;
	wire [7:0] dout1;
	reg [8:0] precolor;
	reg ready;
	platform image1(addr, clk, dout1);

	always @ (x or y or hcount or vcount) begin
		if((hcount >= x && hcount < x + WIDTH) && (vcount >= y && vcount < y + HEIGHT))
begin
			addr <= (vcount - y) * WIDTH + (hcount - x); //forwards
			ready <= 1;
		end
		else begin
			addr <= 0;
			ready <= 0;
		end
	end

	always @ (posedge clk) begin
		if(ready == 1)
			precolor <= {0, dout1};
		else precolor <= {1, 8'b00000000};

		if((precolor[8:8] == 1) || (precolor[7:0] == 8'h1C))
			color <= {1, 8'b00000000};

		// platform id cooresponds with what color it should be
		else if(precolor[7:0] == 8'b11100000) begin
			case(id)

				1: color <= 9'b000011100;

				2: color <= 9'b000000011;

				3: color <= 9'b011100011;

				4: color <= 9'b011111100;

				5: color <= 9'b000011111;

				6: color <= 9'b011111111;

				default: color <= 9'b100000000;

			endcase

		end
		else color <= precolor;
	end
endmodule


// rectangle sprite module
----------------------------------------------------------------------------------

module rectangle

   #(parameter WIDTH = 64,      // default width: 64 pixels
```

```
                HEIGHT = 64,      // default height: 64 pixels
                COLOR = 9'b000011100)  // default color: green
    (input [10:0] x,hcount,
     input [9:0] y,vcount,
     output reg [8:0] pixel);

    always @ (x or y or hcount or vcount) begin
       if ((hcount >= x && hcount < (x+WIDTH)) &&
                                (vcount >= y && vcount < (y+HEIGHT)))
                         pixel = COLOR;
       else pixel = 9'b100000000;
    end
endmodule
```