Implementing a Digital Percussion and Entertainment System

David Levenson
Danny Malconian

December 14, 2005
Prof. Ike Chuang
Prof. Chris Terman

Abstract:

This report outlines the details and design methodology taken in creating digital percussion and entertainment system on a field programmable gate array. The system allows for three modes of operation. The system can be operated in a drum kit mode, a drum kit and visualization mode, or a video game entertainment mode controlled by the drum kit. Drum pad inputs with piezo transducers are used as inputs for this system and there is audio and video output. The steps taken to design each portion of the entertainment system are described as well as the process of integrating the subsystems present in the design. The testing and debugging process, both during computer simulation and actual implementation on the FPGA are discussed. The results of the design and construction are also analyzed. It was found to be possible to implement the digital percussion and entertainment system using an FPGA, three drum-pad inputs, three analog to digital converters, several switches to select modes, a reset button, and speakers and a video monitor to handle the outputs of the system.

Table of Contents

List of Figures

Introduction

The goal of this project is to implement a multi-functional digital percussion and entertainment system on a Xilinx Field Programmable Gate Array (FPGA). This system will combine both audio and video components. It is useful for a system designed to entertain a user to have many functions with which the user can interact. These functions should have both audio and video outputs to provide maximum entertainment value for the user. This is why we chose to combine a digital drum kit, visualization techniques, and a video game playable using the drum kit. For users who are interested in music, specifically drumming, this system provides a functional drum kit with visual feedback and multiple settings that a normal drum kit cannot produce. For the user who is into gaming this system provides a novel way to interact with a video game and control a character on a level. We chose to emulate the Super Mario Bros. video game since this has proven to be a successful and entertaining game through the years.

The user of this entertainment system sees a drum kit with practice pads. Coming from these pads are RCA cables which connect to the major digital electronics of the system. The user will also see several switches and buttons with which to control the settings of the system, as well as monitor to view the visual components of our project. When the user turns the system on, they will have the option, through switches, of using the drum kit as a drum kit only, providing visual feedback with the drum kit, or using the drum kit as a controller for a video game. The user will interact mostly with the drum kit as all of the options are controlled solely through hitting the pads. Aside from the switches used to set the desired function of the kit, there will be another set of switches to select the sounds of the kit, and a reset button to reset the system to its initial state. If the user selects the “drum kit only” function they will simply here the sounds of the kit they select as they drum on the pads. If they choose to show visual feedback they will see waveforms displaying the intensity of their hit, the signal entering the controller from each pad, and a waveform of the sound being sent through the speakers. Finally, if the user selects to play the video game they will see a character displayed on the screen and a timer. They must move the character across the screen as fast as possible while avoiding all obstacles in the character’s way.

This project is divided into several major sections. The top level modules are the hit recognition module, the audio playback module, the drum visualization module, the game logic module, and the game video module.The following figure shows the high level block diagram for our system. All of the major modules and their basic connections are shown and will be explained in further detail in the implementation section.

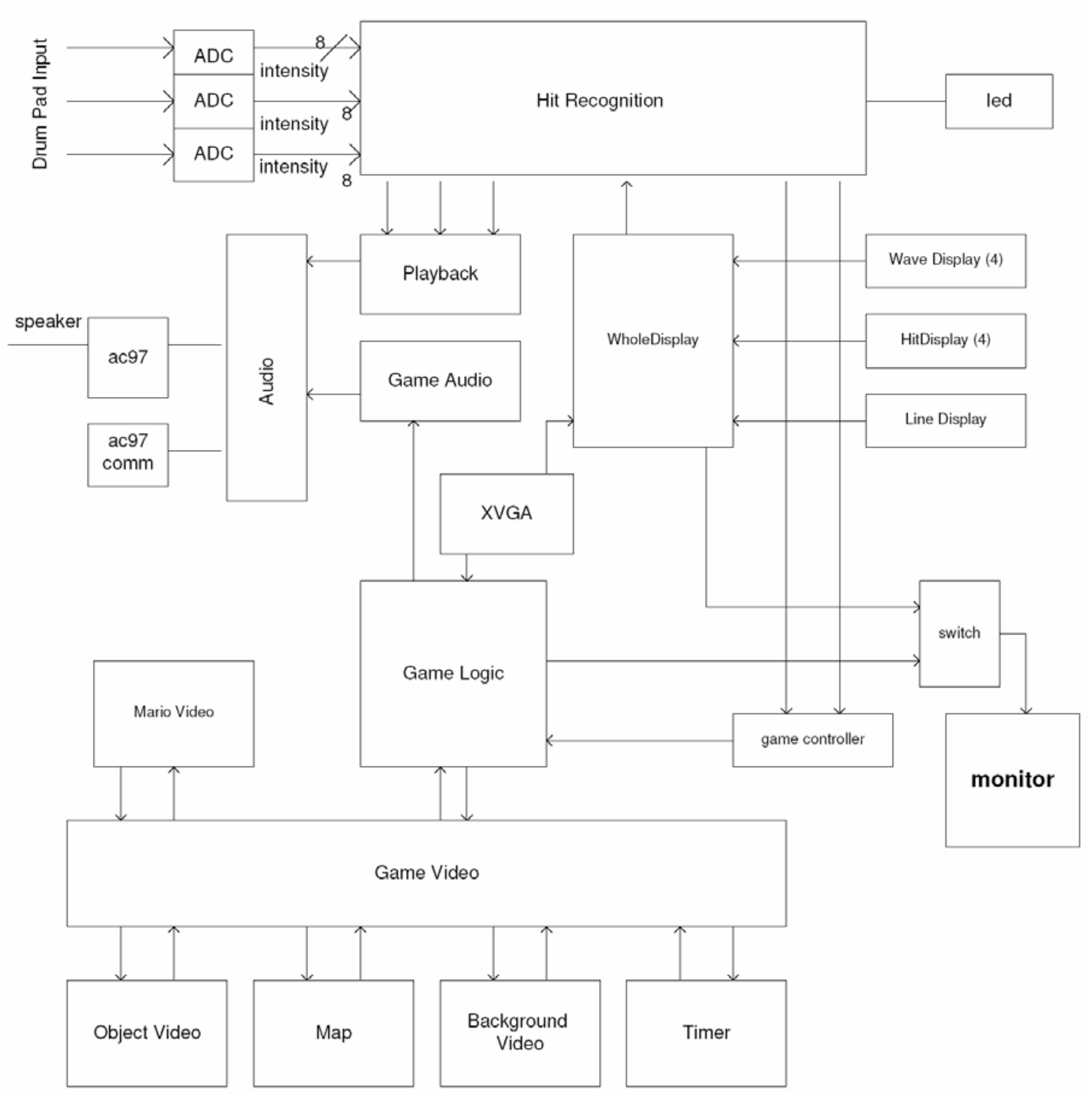

Figure 1: Top-Level Block diagram of all modules and their basic connections contained in our system.

The hit recognition module handles all of the inputs from the pads of the drum kits. The audio playback module plays the appropriate sound at the proper level determined by the intensity of the hit on the pad. The waveforms and visual drum feedback displayed are controlled and output to the screen with the drum visualization module. The positions of all objects on the screen is controlled through the game logic module, and all of the objects on the screen are displayed with the game video module. For example, if the user is in the video game mode and strikes a pad the game logic module will determine where to move the character depending on the objects impeding its motion. It will then send the appropriate coordinates to the appropriate game video module (there will be several of these modules as there will be several different types of objects) which will display the correct images from memory.

The drum kit and drum visualization modules are independent of the video game and can, thus, be developed and tested independently. Integration of the drum kit and the video game will be the last step in the process.

This document provides an outline of the design, implementation, testing and debugging of this digital percussion and entertainment system. Major problems and their solutions are discussed and recommendations for further improvement are given.

Implementing the Drum Kit

To design our drum set, we used piezo transducers in the middle of a tunable Remo practice pads. The practice pads are a plastic casing with a foam core and regular drum head pulled across and screwed in at 10 points like a regular snare drum. The practice pads are very responsive and similar to the feel of a regular drum and there are special stands made to accommodate 5 or 6 pads to set up like regular drum set. To pick up a signal, we took apart the pads, split the foam in half, and inserted a metal circular sheet with an attached transducer in the middle. Connecting the transducer to an RCA jack, the pads are connected to the ADC simply by running RCA cables from the set. The following figure shows the implementation of the analog circuitry for our drum pad.

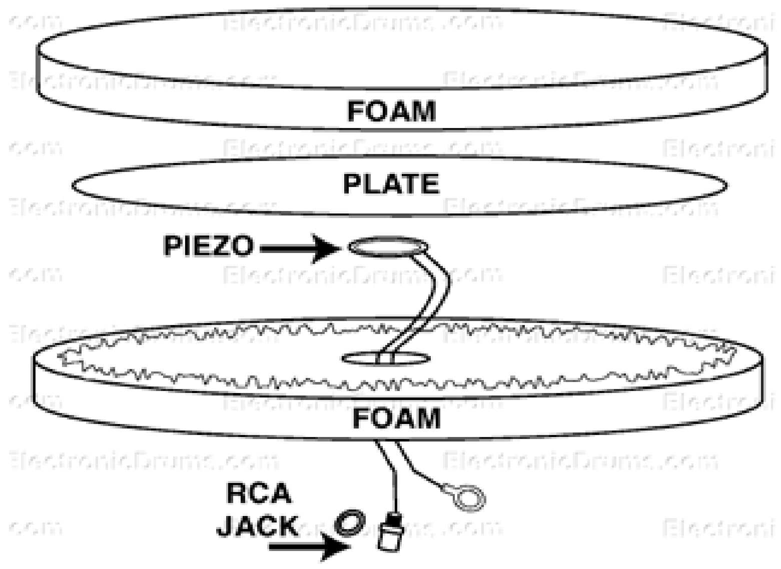

Figure 2: Diagram of how the piezo transducer is placed in the drum pad to pickup a hit

All of our audio samples for the drum sounds were simple wav files each lasting around a second. To get the wav files from the computer to the FPGA, we utilized several matlab functions. Wavread() writes all of the PCM data of a wav file into a vector from -1 to1. Given the data was 16 its, multiplying the -1 to 1 data by 2^15 (2^16 minutes the sign bit). A twos complement function was then used to get the numerical data into a binary representation. Using the csvwrite() and some filtering for commas, the data was in a proper .coe file.

The following figure shows an in-depth block diagram of the drum input, hit recognition and playback modules.

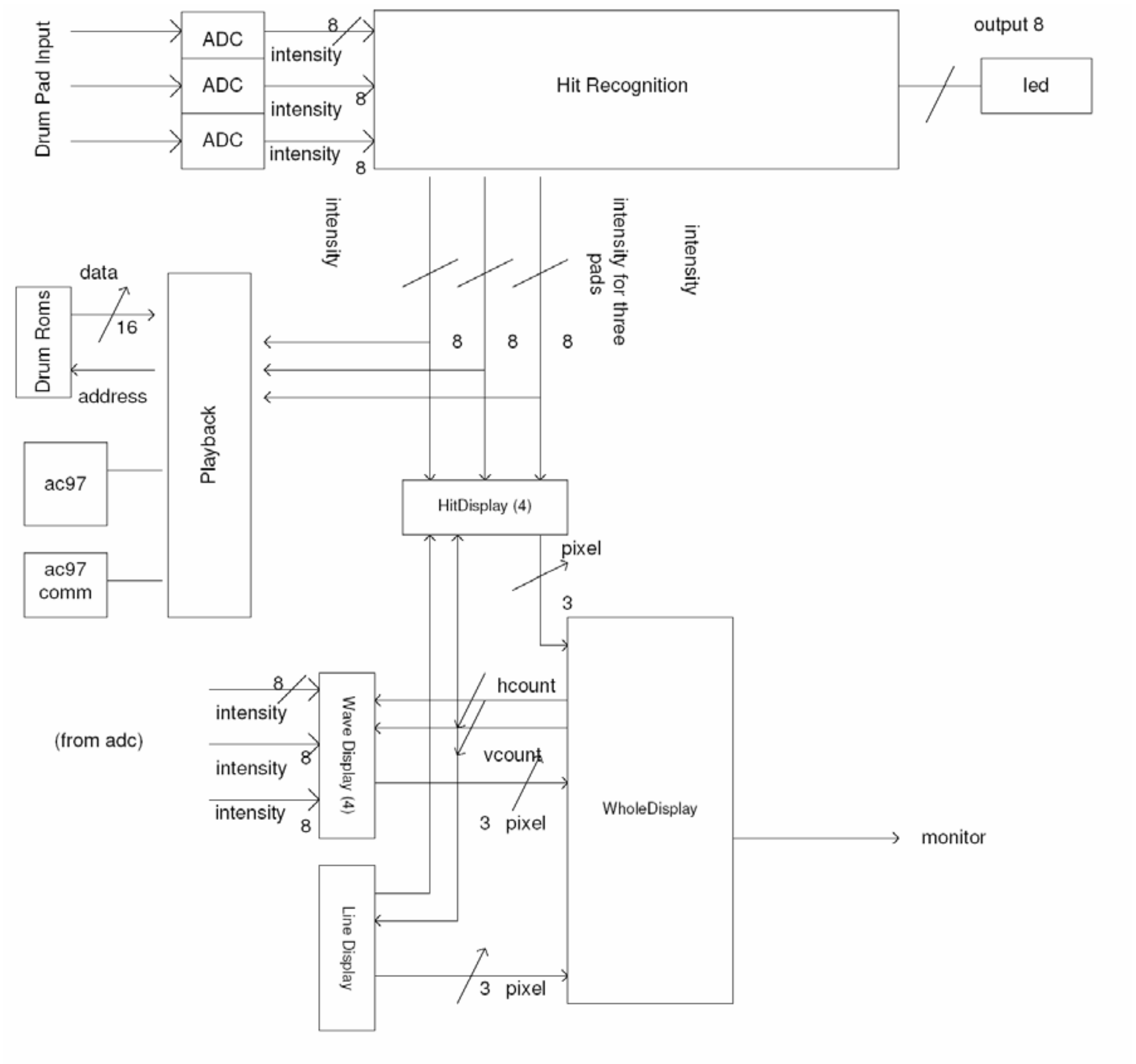

Figure 3: In-Depth Block Diagram of drum input, hit recognition and playback modules

The first part of the drum pad system is the interface of to the ADC. Our system uses the analog devices ad-670, a parallel ADC. The ADCInterface module is responsible for driving the ADCs and outputting the data to the rest of our system. This module has a clock input, reset input, data input, and status inputs. The outputs are a rw signal, the ADC’s last output data, and a new sample. The data output is the ADC’s last sampled value and new sample is a one clock cycle pulse signifying a new sample is ready. The rw is used to connect with the ADC. When rw is held low, the ADC is in write mode and will continuously sample and write the values to internal registers. After writing a new value and the ADC’s output is valid, the ADC will set its status signal high for a short period at which point at which point the data can be read. To initiate a valid read, the rw signal must be brought low while status is high. The ADCInterface’s functionality is implemented with a 3 state FSM. The first state sets the rw signal low and the newsample signal low. The next state waits until the status signal from the ADC is high, at which point it sets rw to high and initializes a counting used in the next state to account for the propagation delay for the ADC’s output to be valid after changing from a write to read. The final and third state waits until the counter has waited long enough to ensure the data is valid, it writes the ADC’s data output to the registered data output of the module, sets rw to 0 and the cycle repeats.

The HitRecognition module is responsible for taking the ADC’s output waveform and detecting when the drum pad has been ‘hit.’ The module has three parameters a minimum intensity (minRec), a time to output (outTime), and a time to wait (maxCount). When the input to the module crosses over the minRec threshold, the module starts a counter that runs until outTime. During this time, the module records the highest intensity from the ADC and when the counter reaches outTime, a one cycle pulse of the maximum intensity is sent. The counter continues beyond outTime and when it reaches maxCount, the module again waits for an input crossing the minRec threshold. Balancing these parameters can be somewhat tricky as there is give-take in the ending performance. Since the transducer's output on a hit can last up to ten or fifteen milliseconds, improperly setting these parameters can result in reading a "double-hit" of a single stroke. Setting a high minRec means the maxCount can be very low. Although the waveform from a stroke lasts a long time, its intensity decreases so the output from a hit 10 milliseconds ago would not trigger a hit if the minRec is substantially high. The other possible solution is a low minRec but high maxCount. In this way, waiting long enough after a hit will ensure the output of the transducer is below even a low threshold. The first solution leads to a less responsive pad, but one you can hit very fast. The second leads to a more responsive pad, but one that may not perform well on a drum roll or quick hits.

The drum_audio module is responsible for taking all of the hit recognition pulses and playing the appropriate drum samples. The module takes a ready input, a signal output from the ac97 every 48 kHz to notify the module when to update its PCM data output. The module has six ROMS. There are two drum samples for each type of drum, snare, bass, and high hat. These ROMS store that sample in 16 bit mono PCM data sample at 48 kHz. Whenever a pulse is received, the address for the ROM of the corresponding pad is reset and an intensity register for that pad is set to the pulse's value. For each pad, there is an intermediate wire which the output of the first ROM if the drum kit switch is 0, and the output of the second ROM if the drum kit switch is 1. These three wires are multiplied by the high four bits of the corresponding intensity register (so 0 if the sound is not playing). These multiplied signals are 22 bits. To get the left channel of the output sound, the high order bits of the bass multiplication and the snare multiplication are added together. For the right channel of output sound, the high order bits of the high hat multiplication and the bass multiplication are added together.

A nearly identical module called GameAudio has the same core as DrumsAudio. Instead of taking an 8 bit pulse to trigger playing audio from a ROM, it takes a one bit pulse from the MarioGameLogic module to trigger when to play the audio. A ROM is loaded with the sound of Mario jumping and played whenever the play_jump signal is set high. This module will be discussed below in conjunction with the video game.

Implementing the Video Game

To implement the game we first thought about how to minimize the amount of logic needed. We were well aware that the logic of the video game would be the limiting factor for the speed of our entire system. The game itself is a race for the best time. A character moves across the screen to the right to the beat of the drum. The character is able to jump to maximum height that is determined by the intensity of the hit on the drum kit. There are several obstacles in the path of the character impeding his motion forward that he must clear and avoid.

To minimize the amount of logic used we decided to divide the screen into 32, 32 pixel vertical sections. The character would only ever be in one section at any time. Therefore, we would only need to check the logic of the vertical section the character is in and the one he will move to. This method was chosen over checking for collisions with every object in the level which would require countless “if” statements, inevitably slowing down our entire system. The figure below shows how the screen was divided.

The size of the sections is not to the correct scale and the number of sections are not correct in the figure to allow for easy viewing.

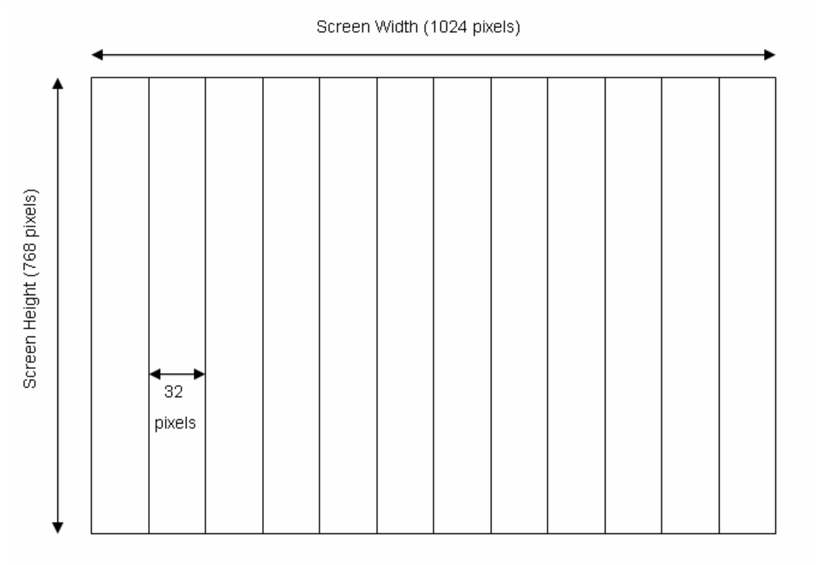

Figure 4: Diagram of how screen is divided up into several 32 pixel vertical sections.

There ended up being four layers to the game. First there was a simple plain background color upon which all of the objects and backgrounds of the game would be placed upon. Next, there is a background layer which contains all background images which the character will not interact with (i.e. clouds and hills). On top of the background layer is the object layer which contains all objects the character can interact with including walls and pits. And finally, the character itself is placed on the top layer. The background and object layers are broken up into the aforementioned vertical sections.

Dividing up the screen into vertical sections also makes level design quite easy. Instead of entering countless numbers into the actual verilog code as parameters a file is simply loaded in. Each word of the file signifies one vertical section and the files can be created and updated quite easily.

The following figure shows an in detail diagram of how all of the modules in the video game section of the block diagram are connected.

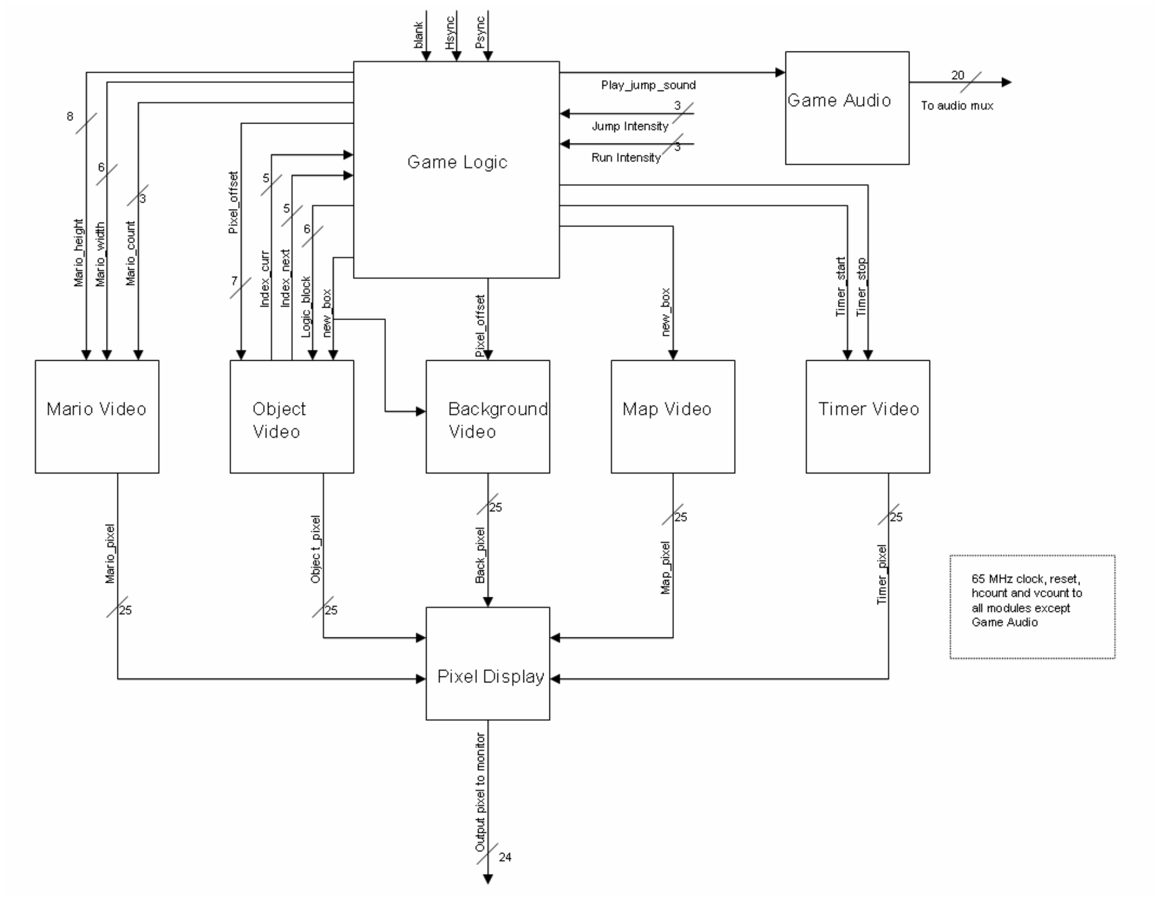

Figure 5: In-Depth Block Diagram of the Video Game portion of the system.

Logic Module

The Logic Module is the heart of the video game. It controls the position of the character on the screen, tests for all collisions and interactions with objects, and updates all of the other video modules. It receives inputs from the object video module to determine what objects are in the section the character is currently in, and what objects are in the next section the character will enter. This module takes in all of the sync and blank information from the xvga module to determine timing for the system. We delay the sync and blank signals by one cycle to help avoid glitching since this module will be the major factor in slowing down our entire system. This module controls the state of the character using a simple FSM. The character can be in 1 of 5 modes that include jumping, standing and 3 modes for running. It outputs the vertical section the character is located in and receives the index value of that vertical section and the next one after it to determine correct collisions. This module sends out a new_box signal to several modules to alert the system when the character has entered a new vertical section so that the screen can be updated. It also sends out the pixel_offset value as discussed above. It interfaces with all video modules and the sound module, alerting it when to play the appropriate sound. The timer is updated with a timer_start and timer_stop signal that is sent from this module

One drum pad is used as a button input to assign the character an x-velocity. To simulate acceleration and deceleration the character's x-velocity increased in steps to a certain final velocity depending on if the drumming input was continuous. This works only when the character has no y-velocity (i.e. he is on the ground). To make the character jump a different drum pad is hit. The intensity of that hit determines the maximum jump height for the character. The y-velocity is assigned using this intensity and the character jumps in steps moving to the maximum height. The maximum height of the jump is assigned with the objects located in the current vertical section in mind. For instance, when there is only ground in a section the total distance traveled is the same as if there were a raised block, but the height on the screen of the character at the top of his jump is different.

The logic for the game is only updated at the negative edge of every other vsync signal. The vsync signal is set low at a frequency of 60 Hz. We chose to update the logic every other cycle to make the game look smoother. It will not run too fast for the user to see all that is happening.

The logic module divides the screen up into 32 vertical sections and assigns each a logic block number depending on the position of the section. This module keeps track of which logic block the character is located in so that it can run the appropriate tests for that section. It sends the number of the logic block that the character is in to the object video module. The object video module then returns an index value for the section the character is currently in and for the next section the character will enter.

The character starts out on the left side of the screen and his x position is updated until he hits the center of the screen. At this point the background begins moving towards him. This was done to keep continuity in the game. To move the background smoothly to the left we implemented a register that holds the pixel_offset. The pixel_offset holds how much the background has shifted from its original position. It can hold values from 0 to 31 since there are 32 pixels per vertical section. When the character is centered the pixel_offset moves by the character's assigned x-velocity instead of the character. If the pixel_offset plus the character's x-velocity is greater than or equal to 32 we have entered a new vertical section and the new_box output is held high to alert all of the video modules that the character is entering a new vertical section. The figure below shows the character centered on the screen with the vertical sections shown. There is a non-zero pixel_offset value shown because the character has moved, but not through a full vertical section.

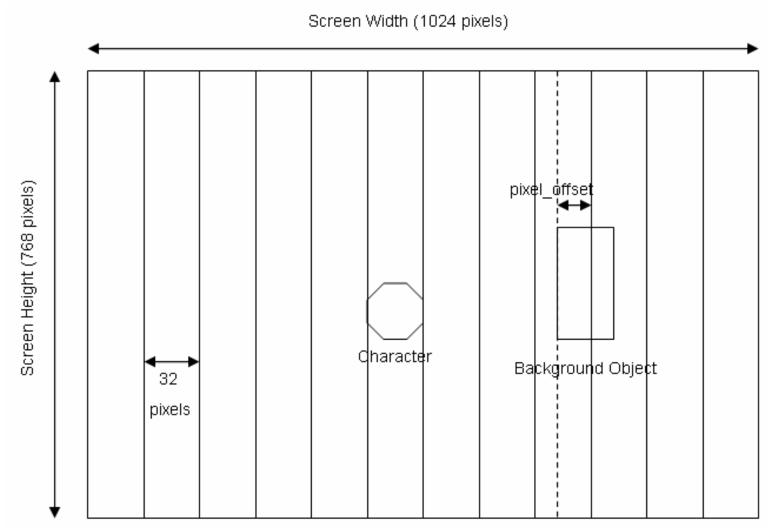

Figure 6: Diagram of position of character in vertical sections when centered and definition of pixel_offset value.

The character can interact with several different types of objects on the screen. There is specific logic written for each object which will be explained here.

Flat Ground: This is the most basic type of object the character interacts with. The character walks along the ground level only and returns to this level after jumping. The only tests necessary are to see if the character has reached his maximum jump height or if he has reached the ground.

Pit: A pit is essentially a hole that the character can fall into and not jump out of. A test is needed to see if the bottom of the character is below the ground level. If this is the case then the character becomes “trapped” in the pit and falls off the bottom of the screen. As a time penalty for falling in a pit the user must wait for the background to move left one pixel at a time until the center vertical section no longer is a pit. Then the character is dropped from the top of the screen with its normal y-velocity and the user must wait until he hits an object, before the user can again move the character forward.

Blocks: Blocks act to impede the forward motion of the character. They are 32 pixels high and can be stacked upon one another to create a larger wall-type structure. When a vertical section contains blocks the logic module must test collisions with the sides of the blocks and the top. The character has the ability to walk on top of the blocks and multiple vertical sections of blocks can be placed next to each other to create a "raised ground" effect.

Flags: Flags signify the starting and ending points of a level. The "start" flag is located in the center of the opening screen. In the image file of the flag, the flagpole is cemented into a block so the collision logic of the flag section is much the same as that for a section with just blocks. After the first flagpole is reached a start signal is sent to the timer module to begin the timer. Upon reaching the second flagpole a stop signal is sent to the timer and the character enters a different mode. The character now remains stationary in the x direction and jumps up and down to signify finishing the level. The user must reset the game to start the level over and play again.

Tubes: Tubes act much like blocks. They impede the forward progress of the character. However, they cannot be stacked upon one another like blocks and have only one specified height. The character can jump and walk on top of tubes.

Background Video Module

This module takes in hcount and vcount from the xvga module, pixel_offset and new_box from the logic module and outputs a 25 bit pixel to display on the screen. To implement this module we created a queue that had 33 elements, representing the 33 possible bars on the screen at any point. There can be 33 bars displayed because the first section may be partly off the screen allowing the 33rd bar to also be displayed. The queue imported the data it contained from a ROM that holds index values for the different background types to be displayed. Each index value corresponded to a different background to display. For example a “0” index value represented no background, a “1” represented the left side of a cloud and a “2” represented the right side of the cloud. Based on the position of hcount from the xvga module and the pixel_offset from the logic module, this module could determine which vertical section it was in and assign the correct value to the output pixel based on vcount. There were several ROMs instantiated inside this module, which were read from when needed. The address was updated based upon the position of hcount and vcount. If a new_box signal comes in from the logic module it means that one full vertical section has passed off the screen and we must update the queue. This is exactly what happens and the queue shifts all of its index values down and reads a new one from its memory file.

Object Video Module

This module takes in the same inputs as the background video module and outputs its own 25 bit pixel to display to the screen. The one major difference between this module and the background video module is that this module takes in the vertical section which the character is in and returns the index value corresponding to the objects in a vertical section to the game logic. This value is called the logic_block. The video it displays correspond to anything the character can run into or interact with. This includes the ground, pits, blocks and so on. There is no logic calculated within this module. It also uses a 33 element queue to divide the screen up into 32, 32 pixel vertical sections. It returns to the logic module the index value from the queue associated with the section the character is currently in and the next one it will enter. This module also reads from its own ROM which stores index values which correspond to objects on the screen.

Character Display Module

This module takes in the height, width, x and y postion of the character to display, the index of which character to display, as well as hcount and vcount and outputs a 25 bit character pixel to be displayed on the screen. This module contains 5 different ROMs as there are 5 different positions that the character can be displayed in. There are 3 positions corresponding to running, one corresponding to standing and one corresponding to jumping. The address to the ROMs was updated based on the positions of hcount and vcount and the x and y coordinates coming in from the logic.

Map Display Module

This module draws a 256 pixel long map of the position of the character in the level. It takes in a new_box signal from the logic, hcount and vcount from the xvga module and outputs a 25 bit map pixel in the top right corner of the screen. We designed the game so that each level is 256 vertical sections long after Mario is centered on the screen. Therefore every time the character enters a new section the logic will send a new_box signal to this module and the character’s representation on the map will be incremented.

The map is then drawn based on the position of hcount, vcount and the character's position within the map.

Timer Module

This module takes in an start and stop signal from the logic module, hcount and vcount from the xvga module and output a 25 bit timer pixel that is used to display the time the character has been active in the level. The time is displayed in tens of minutes, minutes, tens of seconds and seconds in the form "xx xx." There is a different 4 bit register for each of the minutes and seconds types to be displayed. This module contains a ROM that contains all of the fonts to be displayed on the screen. The numbers are 7 pixels long each, so to find the correct address to display we simply multiply the value in each register by 7 to get the correct section of the ROM. Upon receiving a start signal from the logic the timer begins incrementing and does so until a stop signal is received from the logic module.

Pixel Display Module

This module takes in all of the output pixels assigned in the other video modules, selects the appropriate RGB value to display and outputs the correct 24 bit pixel to the screen. The reason all of the video modules output a 25 bit pixel is because there are 24 pixels of color information (8 for red, green and blue each) and 1 pixel of transparency information. The pixels outputted from the video modules contain information for every location of the screen, however the character, for example, takes up only a small portion. Thus, the transparency bit for the character is high for all locations where there is no character information. This module looks at this bit and selects what to display. This module also stores the background color to display if every video module pixel output is transparent.

Game Sound Module

This module received a signal from the logic module indicated a jump move had been made and outputted an 8 bit sound to be played by the ac97 sound module. This module took in the ready signal from the audio module and only sent sound data at the negative edge of the ready signal. When the ready signal goes low it means the ac97 is ready to receive the next sound. This occurred at a frequency of 48 kHz. The sound was originally a .wav file that was converted to a .coe file through a matlab function. We wanted to include more sound in the game other than just a jumping sound, but the sound files took up too much memory on the FPGA so we were unable to do so.

All modules take in a 65 MH clock and a reset signal that is tied to a button on the labkit. If the reset button is hit, any registers in the modules (i.e. position, address, setup registers, timers, trapped registers, start and finish signals, etc) are reset to their initial state to allow for proper gameplay.

Storing Bitmaps in ROMs

Virtually every image displayed on the screen in the video game is an image that is stored in a ROM. There are 2 ways to get the images into the ROM, both of which were utilized for our project. Both methods store the image in a coefficients (.coe) file. This file can be loaded into the ROM upon creation with the IP wizard. The .coe file stores binary data in a vector which is read by our video modules. The first, and more tedious method for converting an image file to a .coe file is to “hand-draw” it pixel by pixel into the file. This is done by looking at the image, assigning color values to each shade in the image, and writing the .coe file by hand in binary. The second, and more efficient way, is to use a function in Matlab called “imread.” This function looks at a bitmap file and converts the colors into a specified number of binary bits. Matlab also removes the headers and extra information stored in a bitmap which is unnecessary to store in the ROM. Matlab can even assign a transparency bit to specified colors in the original bitmap which greatly aided us in converting these images to .coe files. All the images for our game were taken from http://sprites.fireball20xl.com/NSA/HTML/Mario/smb.htm.

Implementation: Integration

Originally, the drum kit and the video game were created separately. The video game was designed using button inputs to signify a run or a jump and switches to assign an intensity value to the jump. The drum kit is a self-sufficient kit. It needs no inputs from the video game to run properly. Upon integration of the game and the kit there was, therefore, no need to worry about the kit. A game controller module was created to interface between the drum kit and the video game. This module takes in the 8 bit intensities from the appropriate drum pads and outputs two 3 bit intensities to the game logic module. At every negative edge of the vsync signal the game logic module sends a reset signal to this controller module to alert it that the logic module is ready to receive a new intensity.

Testing and Debugging: Drum Kit

To debug and test the input from the drum pad and ADC, a set of modules are used to display the waveforms. The parent module, WholeDisplay, has four instances of WaveDisplay four instances of HitDisplay, one instance of sound_display, and one instance of lines. These four modules each take in an hcount and vcount and output a color based on that position.

The Lines module is small and straightforward. It outputs a while pixel if the current position is on the boarder of the screen, if vcount is equal to 576 or if vcount is greater than 576 and hcount is 192. In other words, it creates a white board and divides the screen into three pieces.

The first piece and largest piece consists of the four waveforms from the ADC's. These waves are all drawn with the WaveDisplay module. The wave display module takes the 8-bit ADC output in as well as a clock and screen position. This module also contains two 8-bit 1024 width RAMS. Since the width of the screen is refreshed every 60 Hz, and the screen is 1024 pixels long, the ADC had to be sample at 61440 Hz. Thus every 1058 clock cycles (65 mHz / 60*1024), the ADCs output is written to a ram. The two rams alternate between being read and being written every new frame. The data from the ADC stored in the ram is in twos complement form. The ram's address is tied to hcount. The waveform is drawn by setting the pixel register to a color if the vcount is within 2 pixels of the RAMs output added with the top offset parameter.

The Sound_Display parameter is essentially the same as the wave display except its rams are only 768 wide. The sound display module also takes 8 bits of data only this data is the 8 high-order bits of the data sent to the ac97 chip. It also samples at a slower rate since it only needs 768 samples every 60 Hz.

The final module used in the whole display is HitDisplay. This module outputs bar 15 pixels high and from 0 to 256 pixels wide for a short period displaying the intensity with which a user just hit a drum pad. Each drum pad had a hit display bar and a wave display of the same color. The HitDisplay used two other modules in outputting the bar. The HitDisplay was used to recognize the maximum intensity of a hit. A module named Turn_On was used to simply take a pulse from the HitDisplay and prolong it to hold for about a tenth of second.

The first modules build were the ADC interface, the hit recognition, and module to hold the value of a pulse for a short period. All three of these were easily tested with the Xilinx testbench. After writing the code, we had to make only minor changes until they functioned properly. After verifying those modules worked, we hooked up the ADC interface to the hit recognition system and the output of the hit recognition and the pulse hold to the labkits LEDS. At first the module would function exactly as desired but the output to the LED’s would freeze after a certain amount of time. Using an ocislloscopre, and testing the rw and status signals on the FPGA, we found that in face the FSM controlling the ADCs was freezing. Going back onto the testbench, we found that indeed it was functioning as it should. After spending a good deal of time trying to figure out what could be causing this, we decided to simply implement logic in the FSM to reset to its initial state if its been frozen or in the same state too long. This had no effect. We then tried clocking the ADC interface much slower with no change.

We initially used the lab4.v as base to start our project. In it, there were several modules clocked at 65 mHz. On a whim, even though our modules were not all connected to the graphical components of lab4.v, we decided to clock the ADC at 65 mHz and it worked fine.

After the ADC interface and the hit recognition system was functional, our next step was to build displays so we could easily view the waveforms. The displays were actually fairly straightforward and easily constructed. First step was building the individual wave display. Once that had worked combined four of them to display on one screen. After that was constructed, we had a visual way to see what was going on with the ADC output and the hit recognition. After this was constructed, we could see our system working as desired.

The drum system involved taking the pulses from the hit rec and playing back corresponding sounds. Since we are playing the sounds back at a volume proportional to the hit intensity, we had to multiply, shift, and finally add the three sets of PCM data from the ram. Although we did not have any conceptual or systematic errors in our design, this process involved a significant about of debugging and fixing glitches. Having the waveform displays of the high order bits of the ac97 output made the debugging significantly easier. We could tell when the data was clipping or a sign bit was not being assigned.

When the system was functional from ADC conversion to outputting sound, we had adjust threshold and timing parameters to make the drum set responsive and realistic. Since the transducer's output was a diminishing waveform lasting on the magnitude of ten to fifteen milliseconds, we had to adjust two parameters to stop the drums from "double hitting." If there was no minimum threshold voltage to trigger a hit and if there was not a long enough delay before allowing the hit recognition to output another hit, one strike to the pad would result in a high-intensity pulse followed by a lower-intensity pulse very shortly after. The problem was that adjusting the parameter was a give-and-take situation. A high minimum voltage results in less responsive pad and a high count delay resulted in not recognizing very fast hits on a drum roll. The process of finding an adequate balance in the parameters was somewhat cumbersome. The characteristics of a bass pedal hit and a drum stick hit were different enough that we had to go through the process twice. Although we could scale parameters post-compiling by multiplying parameters by switch values, having to balance the two properties was time consuming. Even after the initial process we fixed the parameters many times.

Testing and Debugging: Video Game

The testing and debugging process for the implementation of the video game was a long and arduous one. Once we had the game working basically how we wanted it, we had to integrate which began another whole testing and debugging process. Throughout this process many tools were used ranging from Xilinx testbenches to the waveform data inputted in from the analog to digital converters.

The approach we took to testing and debugging our video system was modular. We intended to create small modules that implemented a specific function. Doing this, rather than having large modules performing multiple and different calculations, allows for a much easier and smoother debugging process. For example, none of the video ROMs dealt with any of the logic for the game. This allowed us to limit the possible problems associated with each module. We would test each module individually to determine its effectiveness.

For the video game, most of the modules were tested simply by looking at the output. By looking at what was displayed on the screen it was easy to decipher if there was a problem or not. For example, when we were building the character video display, the character is displayed in a box that is the character's width wide and his height long. If the character was displayed in the correct box, but there was movement of the pixels within the box the problem was usually an addressing problem. If the heights and widths we defined for the character were off, or the number of addresses in the character's ROM was off then the appropriate pixel would not be written in the same place every time. For example, if we were drawing a character that had 25 bits (a 5 pixel by 5 pixel character) in a box that was 4 pixels by 4 pixels then the 17th pixel would be written over the 1st pixel after the object was drawn once and the character would appear to shift diagonally down. We became quite good at spotting these errors since many of our image ROMs were all different sizes.

Some tests were run using the test-benches provided in the Xilinx toolchain. The modules tested with test benches were usually the ones with queues. This was so because we needed to test that the correct values were going into and coming out of the queues at the correct times. Often times, the images displayed to the screen were glitchy and moved all over the screen. The test benches were used to check the timing of the information going into and out from the queue. This greatly helped us debug the modules using queues to read from memory files.

By testing small modules and discovering small errors early before they became large errors, combining video modules turned out to be a relatively smooth process. One by one, we would connect a video module to the logic module and look at the display. After making sure the correct image was displayed in the correct location we would connect another module. We started with the four main layers before creating and connecting the level map and timer. By doing this piece by piece, we could ensure that any new errors found by connecting a new module were due only to the new module and not previously connected and tested modules.

The first module created was the logic module. This was done for several reason. Firstly, it was the most complicated and therefore we wanted to put most of our time and effort into making sure it worked properly. Secondly, all of the other video modules connected to it and drew most of their information from this module. Therefore, to test the correctness of the other modules, this one was necessary. We originally wrote only what was necessary to move the character straight across the screen with a basic, solid color background. After testing this, we moved onto adding basic objects. Most of our problems came with collision testing when a character was exiting one vertical section and entering the next one. This was due to problems with timing and reading from the queue in the background and object video network.

There most difficult video errors to debug were involved with the background and object video modules. These modules were more complicated than the map or character video modules because they contained multiple ROMs, multiple types of ROMs (image information and index value information), and they contained the queues that had to be dealt with. The character module and map module simply took in x and y coordinates from the logic module and displayed images within this coordinates. The background and object video modules had to look up information and also, in some cases, send data back to the logic module. Several timing issues came up when dealing with the queues. Both of these modules used pixel_offset from the logic module to determine what vertical section hcount was currently located in. The current vertical section is stored in a register called block_num. Block_num is incremented sequentially when hcount reaches the edge of a section. There is an index value in each of these module that stores the value of the type of background to be displayed from the queue. The index value is assigned combinatorially using the current block_num. However, the addresses to the ROMs were updated sequentially in a “case” statement. This led to timing issues. What ended up happening was that the right edge of one object would get delayed and end up being displayed as the left end of the next object of similar type.

Many hours were spent going over and modifying the code within the background and object video modules. At first, we simply displayed different colors in the vertical bars to make sure each bar was displayed correctly. Our first test of this produced inconsistent results. To help debug this problem we created a module that displayed vertical bars spaced 32 pixels apart to trace out each vertical section. This allowed us to see which sections were being incorrectly displayed. We changed some settings and discovered we had made a mistake by including arithmetic in the variable check of the case statement. We were running a case statement on hcount and were checking when hcount was equal to a multiple of 32 (the number of pixels in each vertical section) minus the pixel offset. We found that this led to glitches in the system so we moved all the arithmetic outside of the case statement. We then dealt with the timing issues that delayed the edges of images being displayed on the screen as stated above. We realized we had to update the address one cycle sooner to put the correct pixels in the correct locations.

Fixing the bugs with the queues allowed us to increase the complexity of the logic used. We began testing moving the characters into different objects. There were, of course, problems with the characters not hitting an object at the appropriate place or not hitting at all. These sorts of errors were expected and were dealt with as they came up. Most of the errors occurred in the transition from one vertical section to the next. To deal with these errors we output the pixel_offset to the hex display. This allowed us to see if the character was hitting objects at the right pixel_offset value and if transferring to a new vertical section updated the pixel_offset correctly.

Another method we used to debug the collisions was by display vertical bars on the screen just as we had in the beginning of our design process. These vertical bars were the top layer on the screen and were mapped directly onto the individual vertical sections. This made it easy to see which section a character was located in. It also showed us exactly when a character would cross over into a new vertical section.

Once we had the logic for the collisions working properly, we loaded all of the images into ROMs to be displayed in place of the simple cubes we had been using thus far. This operation went smoothly and there were only a few address errors to deal with. At this point we had the character interacting with the major objects we would end up using and the movement across the screen was relatively smooth.

The next thing task we undertook was to write and test the code that allowed the character to move across the level very smoothly. These tasks included dealing with the character after falling down a pit, making the level look aesthetically pleasing, and handling the game at the end of the level. These additions were small, and were tested simply by compiling the code and looking at the errors in the game. After adding all of the additions we had a working game interface in which a character could move forward until hitting the center of the screen. At this point the screen begin moving towards the character and the character was able to correctly interact with any objects impeding his path.

Testing and Debugging: Integration

Our next major task was integrating the drum kit inputs into the video game. We created the game controller module described above to act as the interface between the drum kit and the video game. We tested this module quickly using a test bench since it simply sent two intensity values to the game logic module at the request of the module (at every negative edge of vsync). Hooking up the drum kit turned out to run rather smoothly. We attribute this to good initial testing of each system. Each system was tested and ready to either output the appropriate values or receive the appropriate values. For example, in the video game the height of the character's jump was based on the intensity from one of the drum pads. To simulate the intensity prior to the integration switches from the labkit were used to test this process. Thus, when we finally hooked up the drum kit, the video game was already prepared for a 3 bit intensity as an input. The video game didn't discriminate between the switch inputs or the drum inputs, so the integration ran rather smoothly.

To test the integration we mainly used the visual feedback from playing the game. There were too many modules at this point to have a test bench be usable (in addition there were ROMs which are unusable in a test bench). We would compile the code and look at how the character and backgrounds moved to determine our errors. We would run the entirety of the level to ensure there were no glitches. We would also make sure to have the character interact with every object in the level to ensure proper collisions and proper reactions. This testing process took roughly a day to complete and we had a fully integrated system.

Following the integration we added sound to the game to make the game more enjoyable to play. This was relatively easy to do except for some obvious syntax errors made while writing the code. The one problem we discovered was that our block RAM space was limited. We had several ROMs in our system due to the drum audio and all of the images being displayed in the level of the video game. This limited us to just using the jump sound, as any other sound took up to much memory. If we had more time to develop the system we would consider using the flash ROM to store large sound files.

Conclusion

The purpose of our system was to develop a fully functioning drum kit. This drum kit should be able to function as a normal kit would except with several preprogrammed sounds to choose from. The system should also be able to display several waveforms on a video monitor corresponding to the input signals from the drum pad, the intensity of the hit on the drum pad, and the outgoing sound date. Finally, there should be an option in the system to load a video game where a character moves across the screen based on input from the drum pads. The character will traverse a level with several different types of objects impeding the character's motion. We feel that we have met all of the goals of our system and that the functionality meets and exceeds our initial goals for the project.

By using a fast hit recognition and playback module for the drum kit section, we were able to recover from a struck pad quickly so we can input hits at a quick rate. This module also allows us to assign an appropriate intensity value to modulate the volume of the audio coming out of the speakers. Displaying waveforms of the data coming in is also made easy by using this method of hit recognition. It provides a great interface between the analog to digital converters and the rest of the digital logic in the FPGA.

Our design for the video game allows for maximum clock speed and efficient level design and video display. By dividing the screen into vertical sections we need to only worry about the section that the character is currently in and the section he will next be entered. This limits the amount of collision tests that need to be performed, which are the main factor in slowing the system down. Level design is easy because it is done in a separate file from the code. The designer can create a level by simply writing the index values of the objects they wish to use into a text file and loading it into the appropriate ROM. We also delay the sync and blank signals by a clock cycle, which increases the speed of our system. We feel that our video implementation was fully satisfactory and worked as planned. Our video was true to the original Mario Bros. game video, and was displayed glitch-free on the monitor.

This project forced us to spend a large amount of time on the overall design of the system. There are too many modules and functions to perform to just begin coding without creating a thorough block diagram, milestones, and a design process to create our final project. We were required to be very organized and manage our time with great care. We feel that we performed very well under the time constraints and produced a final project that demonstrates not only our understanding of digital systems and logic, but also a good feel for proper design and documentation techniques.

The main problems with the hit recognition and playback sections were interfacing with the analog to digital converters and calibrating parameters to ensure accurate hit recognition. It is important to understand the specifications of any chips that a system interfaces with fully. The parameters were found through a system of trial and error to find the maximum tradeoff between response time and hit threshold. Most of the problems with our video system came about due to timing issues between memory and logic. It is important to clearly understand the interface with any memory system, and the timing requirements needed to pull data out of a ROM.

There are two major recommendations we would make for further improvement of the system in the future. First, instead of using just vertical sections for the video game we would recommend implementing a design that divided up the screen into a grid of 32 pixel by 32 pixel boxes. This would make level design even more open than it already is, and would allow for the easy implementation of the logic for placing objects at different vertical locations on the screen. Second, we would recommend using the Flash ROM, or some other type of large storage device for storing all of the image and sound files. This will allow for the use of a larger amount of bigger files and will free up space on the FPGA. The amount of space we took up on the FPGA was much larger because of all of the block RAMs and also because we needed lots of wires and registers to connect to them. This problem could be alleviated by putting all of the memory files into one large ROM and reading from that. The timing of the memory would become more of an issue, but would allow for more exciting gameplay.

```
module lab4   (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;

       input [31:0] user1;
       output[31:0] user2;
       input [31:0] 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'b0;
 assign tv_in_fifo_clock = 1'b0;
 assign tv_in_iso = 1'b0;
 assign tv_in_reset_b = 1'b0;
 assign tv_in_clock = 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
 assign ram0_data = 36'hZ;
 assign ram0_address = 19'h0;
 assign ram0_adv_ld = 1'b0;
 assign ram0_clk = 1'b0;
 assign ram0_cen_b = 1'b1;
 assign ram0_ce_b = 1'b1;
 assign ram0_oe_b = 1'b1;
 assign ram0_we_b = 1'b1;
 assign ram0_bwe_b = 4'hF;
 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
 //sign user1 = 32'hZ;
 //sign 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;

   ////////////////////////////////////////////////////////////////////////////
   //
   // lab4 : a simple pong game
   //
   ////////////////////////////////////////////////////////////////////////////

   // 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 clock_slow , clock_slow_unbuf;

    DCM vclk3(.CLKIN(clock_27mhz),.CLKFX(clock_slow_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk3 is 20
   // synthesis attribute CLKFX_MULTIPLY of vclk3 is 2
   // synthesis attribute CLK_FEEDBACK of vclk3 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk3 is 37
   BUFG vclk4(.O(clock_slow),.I(clock_slow_unbuf));

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .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, clock_65mhz, ~button_enter, user_reset);
              assign reset = user_reset | power_on_reset;

              // UP and DOWN buttons for pong paddle
              wire up,down;
              debounce db2(reset, clock_65mhz, ~button_up, up);
              debounce db3(reset, clock_65mhz, ~button_down, down);

              // generate basic XVGA video signals
              wire [10:0] hcount;
              wire [9:0]  vcount;
              wire hsync,vsync,blank;
              xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

              assign led[7] = 0;




/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////////
 //           ADC_Interface adc2( user1[22:15]  , user1[14] , user2[29] , clock_65mhz , adc2_out , new_sample2 , reset );
//            ADC_Interface adc3( user1[13:6]   , user1[5]  , user2[28] , clock_65mhz , adc3_out , new_sample3 , reset );
              //assign {phsync , pvsync, pblank , pixel } = {phsync2, pvsync2 , pblank2, pixel2};
```

```
// feed XVGA signals to user's pong game

wire [2:0] pixel;
wire [23:0] pixel_game;
wire phsync,pvsync,pblank;
wire phsync_game,pvsync_game,pblank_game;
wire signed [19:0] to_ac97_data_left , to_ac97_data_right , to_ac97_data_left_drums , to_ac97_data_right_drums , to_ac97_data_game;
wire signed [7:0]  from_ac97_data;
wire ready;
wire [24:0] timer_pixel;
wire timer_start, timer_stop;
wire signed [7:0] adc1_out , adc2_out , adc3_out , adc4_out ,adc5_out;
wire [3:0] recOut;
wire new_sample1 , new_sample2 , new_sample3 ,new_sample4 ,new_sample5;
wire hitreset;
wire tmp_nothing;



wire [7:0] tmpL;

//DisplayHit LedHitDisplay( adc1_out , clock_65mhz , tmpL, reset );
//assign led[6:0] = tmpL[6:0];
ADC_Interface adc1( user1[31:24]  , user1[23] , user2[30]   , clock_65mhz , adc1_out , new_sample1 , reset );
ADC_Interface adc2( 8'b0          , 1'b0      , tmp_nothing , clock_65mhz , adc2_out , new_sample2 , reset );
ADC_Interface adc3( user1[13:6]   , user1[5]  , user2[28]   , clock_65mhz , adc3_out , new_sample3 , reset );
ADC_Interface adc4( user3[31:24]  , user3[23] , user2[27]   , clock_65mhz , adc4_out , new_sample4 , reset );

wire signed[19:0] to_ac97_data_net;

assign to_ac97_data_net = to_ac97_data_left_drums + to_ac97_data_right_drums;



whole_display display_for_drums( clock_65mhz ,hcount,vcount,hsync,vsync,blank, phsync,pvsync,pblank,pixel ,
                                 new_sample1, adc1_out,
                                 new_sample2, adc2_out,
                                 new_sample3, adc3_out,
                                 new_sample4, adc4_out,
                                 to_ac97_data_net[19:12], ready , reset );


 wire[23:0] tmp_pixel;
 assign tmp_pixel = {{8{pixel[2]}},{8{pixel[1]}},{8{pixel[0]}}};




   reg [23:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      if (switch[1:0] == 2'b01) begin
             // 1 pixel outline of visible area (white)
             hs <= hsync;
             vs <= vsync;
             b <= blank;
             rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
      end else if (switch[1:0] == 2'b10) begin
             // color bars
             hs <= hsync;
             vs <= vsync;
             b <= blank;
             rgb <= pixel_game;//hcount[8:6];
      end else begin
      //default: pong
             hs <= phsync;
             vs <= pvsync;
             b <= pblank;
             rgb <= tmp_pixel;//pixel[2]*;
```

```
    end
  end

  assign vga_out_red = rgb[23:16];//{8{rgb[2]}};
  assign vga_out_green = rgb[15:8];//{8{rgb[1]}};
  assign vga_out_blue = rgb[7:0];//{8{rgb[0]}};
  assign vga_out_sync_b = 1'b1;    // not used
  assign vga_out_blank_b = ~b;
  assign vga_out_pixel_clock = ~clock_65mhz;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;

  //assign led = ~{3'b000,up,down,reset,switch[1:0]};




        ///////////////////////////////////////////////////////////////////////////////
        //audio
        wire [7:0] snare_pulse, tom_pulse , hh_pulse, bass_pulse;




        assign to_ac97_data_left  = (switch[1:0] == 2'b10) ? to_ac97_data_game  : to_ac97_data_left_drums;
        assign to_ac97_data_right = (switch[1:0] == 2'b10) ? to_ac97_data_game : to_ac97_data_right_drums;

        audio audioMod(clock_65mhz, reset, from_ac97_data, to_ac97_data_left , to_ac97_data_right , ready,
                audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                 ac97_synch, ac97_bit_clock);

        wire[7:0] toggle;
        debounce debounceSwitch1(reset, clock_27mhz, switch[0], toggle[0]);
        debounce debounceSwitch2(reset, clock_27mhz, switch[1], toggle[1]);
        debounce debounceSwitch3(reset, clock_27mhz, switch[2], toggle[2]);
        debounce debounceSwitch4(reset, clock_27mhz, switch[3], toggle[3]);
        debounce debounceSwitch5(reset, clock_27mhz, switch[4], toggle[4]);
        debounce debounceSwitch6(reset, clock_27mhz, switch[5], toggle[5]);
        debounce debounceSwitch7(reset, clock_27mhz, switch[6], toggle[6]);
        debounce debounceSwitch8(reset, clock_27mhz, switch[7], toggle[7]);


        assign led[7] = 0;

        HitRec HitRecSnare( adc1_out , clock_65mhz , snare_pulse , reset  );
                defparam HitRecSnare.maxCount = 5000000;
                defparam HitRecSnare.outTime = 500000;
                defparam HitRecSnare.minRec = 5;
        HitRec HitRecTom(   adc2_out , clock_65mhz , tom_pulse , reset  );
                defparam HitRecTom.maxCount = 5000000;
                defparam HitRecTom.outTime = 500000;
                defparam HitRecTom.minRec = 5;

        HitRec HitRecBass(  adc3_out , clock_65mhz , bass_pulse , reset );
                defparam HitRecBass.maxCount = 5000000;
                defparam HitRecBass.outTime = 500000;
                defparam HitRecBass.minRec = 20;
        HitRec HitRecHh(    adc4_out , clock_65mhz , hh_pulse , reset );
                defparam HitRecHh.maxCount = 5000000;
                defparam HitRecHh.outTime = 500000;
                defparam HitRecHh.minRec = 5;



//      recorder drumAudio( clock_65mhz , reset, ready , from_ac97_data , to_ac97_data_left_drums , to_ac97_data_right_drums , snare_pulse ,
bass_pulse , hh_pulse , led[6:0] , toggle);

        wire[2:0] jump, move_f , move_b;

        gameController gc( clock_65mhz , reset , bass_pulse , snare_pulse, hh_pulse , jump , move_f , move_b , hitreset );
//      drumaudio drumAud ( clock_65mhz, reset , ready  , to_ac97_data_left_drums , to_ac97_data_right_drums ,  snare_pulse, bass_pulse ,
hh_pulse , led[6:0] , toggle);
```

```
        //MARIO VID GAME
                        wire [9:0] mario_y;
                        wire [10:0] mario_x;
                        wire [2:0] mario_count;
                        wire [16:0] mario_width;
                        wire [16:0] mario_height;
                        wire [24:0] mario_pixel;

                        //wire [24:0] line_pixel;
                        wire [24:0] map_pixel;

                        wire setup;
                        wire new_box;
                        wire [7:0] pixel_offset;
                        wire [24:0] bg_pixel;
                        wire [6:0] index_curr;
                        wire [6:0] index_next;
                        wire [6:0] logic_block;

                        wire[7:0] leda;
                        wire [24:0] line_pixel;
                        wire [24:0] back_pixel;

                        wire [63:0] hex_display;
                        wire play_jump_sound;
                        wire play_music;
                        assign play_music = 0;
                        game_sounds gs(clock_65mhz, reset,  ready, to_ac97_data_game, play_jump_sound , play_music);
                //      assign to_ac97_data_right_game = 0;
                //      assign to_ac97_data_left_game = 0;
                        mario_game_vid mgv(clock_65mhz,reset,
                             hcount,vcount,hsync,vsync,blank,
                             mario_pixel,mario_x,mario_y,mario_count,mario_height, mario_width);

                        mario_game_logic mgl(clock_65mhz, move_f, jump, 1'b0, 1'b0, reset, mario_x, mario_y, mario_count, mario_height,
                                                        mario_width,switch[7:4],hitreset,timer_start,timer_stop,
                                                        setup,pixel_offset,new_box,index_curr,index_next,logic_block,
                                                        hcount,vcount,hsync,vsync,blank,physnc_game,pvsync_game,pblank_game, leda[7], leda[6],
leda[5],hex_display , play_jump_sound, play_music);


                        map_vid mmv(clock_65mhz ,reset, new_box,
                              hcount, vcount, map_pixel);

                        back_vid mbackv(clock_65mhz,reset,
                              hcount,vcount,new_box,
                              pixel_offset,back_pixel);

                        line_vid mlv(clock_65mhz,reset,
          hcount,vcount,hsync,vsync,blank,
          line_pixel);

                        obj_vid mbv(clock_65mhz,reset,
                              hcount,vcount,new_box,
                              pixel_offset,bg_pixel,setup,logic_block,index_curr,index_next);
         pixel_draw mypd(clock_65mhz,reset,
            hcount,vcount,
            mario_pixel, bg_pixel,map_pixel, pixel_game,back_pixel,timer_pixel,line_pixel);

         timer mytimer(clock_65mhz , hcount, vcount , timer_pixel , timer_start , timer_stop, reset);

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

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

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


         /*wire [63:0] d;
         assign d = (switch[4] == 1) ? 1 : 0;
         wire disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b , disp_data_out;
         display_16hex hexDisplay(reset, clock_65mhz, d ,
                 disp_blank, disp_clock, disp_rs, disp_ce_b,
                 disp_reset_b, disp_data_out); */
```

```
endmodule
```

```
module DisplayHit( data , clk , out , reset);
        input reset;
        input signed [7:0] data;
        input clk;
        output [7:0] out;

        wire [7:0] mid;
        HitRec h( data , clk , mid , reset);
                defparam h.maxCount = 5000000;
                defparam h.outTime = 500000;
        turnOn t( mid , clk , out , reset );
endmodule

module HitRec( data , clk , out , reset);
        input signed [7:0] data;
        input clk;
        input reset;

        output [7:0] out;
        reg [7:0] out;

        reg signed [7:0] oldMax = 0;
        reg [24:0] timeCount = 0;

        reg runCount = 0;

        parameter maxCount = 325000;
        parameter outTime = 1000000;
        parameter minRec = 5;

        always @(posedge clk) begin
                        if ( reset == 1) begin
                                runCount <=0 ;
                                timeCount <=0;
                                oldMax <= 0;
                        end

                        if ( data > minRec && runCount == 0 ) begin
                                oldMax <= data;
                                runCount <= 1;
                                timeCount <= 1;
                        end

                        if ( runCount == 1 ) begin
                                timeCount <= timeCount + 1;
                                if ( data > oldMax )
                                        oldMax <= data;

                        /*      if ( timeCount == outTime ) begin
                                        out <= oldMax;
                                        runCount <= 0;
                                        timeCount <= 0;
                                end*/
                                if ( timeCount == outTime ) begin
                                        out <= oldMax;
                                end
                                else
                                        out <= 0;

                                if ( timeCount == maxCount ) begin
                                        runCount <= 0;
                                        timeCount <= 0;
                                end

                        end
                        else
                                out <= 0;
                end

endmodule


module gameController( clk , reset , bass_hit , snare_hit , hh_hit , jump , move_f , move_b , new_frame );
        input [7:0] bass_hit , snare_hit , hh_hit;
        input reset;
        output [2:0] jump , move_f , move_b;
```

```
	input new_frame , clk;
		reg[2:0] move_f , move_b , jump;


	always @(posedge clk) begin
			if (new_frame) begin
				move_f <=0;
				move_b <=0;
				jump <= 0;
			end
			else	begin
				if ( snare_hit > 5 )
					move_f <= 5;//snare_hit[6:4];
				if ( hh_hit > 5)
					move_b <= hh_hit[6:4];
				if ( bass_hit > 5)
					jump <= bass_hit[6:4];
			end

	end

endmodule
/*
module HitRecTest( data , clk , out , reset , switch );
	input signed [7:0] data;
	input clk;
	input reset;
	input[7:0] switch;
	output [7:0] out;
	reg [7:0] out;

	reg signed [7:0] oldMax = 0;
	reg [20:0] timeCount = 0;

	reg runCount = 0;
	parameter maxCount = 50000;
	wire m;
	assign m = switch[7:4];
	always @(posedge clk) begin

			if ( reset == 1) begin
				runCount <=0 ;
				timeCount <=0;
				oldMax <= 0;
			end

			if ( data > 5 && runCount == 0 ) begin
				oldMax <= data;
				runCount <= 1;
				timeCount <= 1;
			end

			if ( runCount == 1 ) begin
				timeCount <= timeCount + 1;
				if ( data > oldMax )
					oldMax <= data;

				if ( timeCount > maxCount * m ) begin
					out <= oldMax;
					runCount <= 0;
					timeCount <= 0;
				end
			end
			else
				out <= 0;
		end

endmodule			*/
module turnOn( data , clk , out , reset);
	input [7:0] data;
	input reset;
	input clk;

	output [7:0] out;
	reg [7:0] out;
```

```
    reg[30:0] count = 0;

    reg go = 0;

    always @ (posedge clk) begin
        if ( reset == 1) begin
            go <= 0;
            count <= 0;
            out <=0;
        end

        if ( data > 0 && go == 0) begin
            out <= data;
            go <=1;
            count <= 1;
        end

        if ( go == 1)
            count <= count + 1;

        if ( count == 5000000 ) begin
            count <= 0;
            go <= 0;
            out <= 0;
        end
    end
endmodule


//          ADC_Interface a1( user1[31:24]  , user1[23] , user2[30] , clock_65mhz , out , new_samp );

module ADC_Interface( dataIn, status, rw , clk , dataOut , new_sample , reset  );
    input signed [7:0]      dataIn;
    input                       status;
    input                       reset;
    input                       clk;

    output                      rw , new_sample;
    output[7:0]                 dataOut;

    reg signed [7:0]        dataOut;
    reg [30:0]                  count = 0;
    reg                             state = 0;
    reg                             rw;
    reg                             new_sample;

    reg[15:0] frz_count=0;
    parameter State_Tell_Read = 0;
    parameter State_Wait_Read = 1;

    reg[1:0] split = 0;
    always @(posedge clk) begin

        split <= split + 1;
        if (split == 0) begin
            if ( reset == 1) begin
                state <= State_Tell_Read;
                count <=0;
            end
            else begin
                case(state)
                    State_Tell_Read:    begin
                                                    if (status == 1) begin
                                                        state <= State_Wait_Read;
                                                        frz_count <= 0;
                                                        rw <= 1;
                                                        count <= 0;
                                                    end
                                                    else    begin
                                                        new_sample <= 0;
                                                        rw <= 0;
                                                    end

                                                end

                    State_Wait_Read:   begin
```

```
                                                                                                    count <= count + 1;
                                                                                                    if ( count > 350) begin
                                                                                                              dataOut <= dataIn;
                                                                                                              new_sample <= 1;
                                                                                                              rw<=0;
                                                                                                              state <= State_Tell_Read;
                                                                                                              frz_count <= 0;
                                                                                                    end
                                                                                          end

                                                  default:                       state <= State_Tell_Read;
                                        endcase

                              end
                    end
          end
endmodule



//        ADC_Interface a1( user1[31:24]  , user1[23] , user2[30] , clock_65mhz , out , new_samp );

module ADC_Interface2( dataIn, status, rw , clk , dataOut , new_sample , reset );
          input signed [7:0]    dataIn;
          input                           status;
          input                           reset;
          input                           clk;

          output                          rw , new_sample;
          output[7:0]                     dataOut;

          reg signed [7:0]      dataOut;
          reg [30:0]                      count = 0;
          reg [3:0]             state = 0;
          reg                                       rw;
          reg                                       new_sample;

          parameter State_Tell_Write = 0;
          parameter State_Tell_Read = 1;
          parameter State_Wait_Read = 2;

          reg[1:0] split = 0;
          always @(posedge clk) begin

                    split <= split + 1;
                    if (split == 0) begin
                              if ( reset == 1) begin
                                        state <= State_Tell_Write;
                                        count <=0;
                              end
                              else begin
                                        case(state)
                                                  State_Tell_Write:   begin
                                                                                                    new_sample <= 0;
                                                                                                    rw <= 0;
                                                                                                    state <= State_Tell_Read;
                                                                                          end

                                                  State_Tell_Read:    begin
                                                                                                    if (status == 1) begin
                                                                                                              state <= State_Wait_Read;
                                                                                                              rw <= 1;
                                                                                                              count <= 0;
                                                                                                    end
                                                                                          end

                                                  State_Wait_Read:    begin
                                                                                                    count <= count + 1;
                                                                                                    if ( count > 350) begin
                                                                                                              dataOut <= dataIn;
                                                                                                              new_sample <= 1;
                                                                                                              rw<=0;
                                                                                                              state <= State_Tell_Write;
                                                                                                    end
                                                                                          end
                                        default:                        state <= State_Tell_Write;
                                endcase

                        end
                end
        end
endmodule
```

```
module whole_display(           vclock,hcount,vcount,hsync,vsync,blank, phsync,pvsync,pblank,pixel ,
                                                    new_sample1, sample_data1,
                                                    new_sample2, sample_data2,
                                                    new_sample3, sample_data3,
                                                    new_sample4, sample_data4,
                                                     sound_data , new_sound_sample , reset );

          input new_sample1, new_sample2 , new_sample3 , new_sample4;
          input signed [7:0] sample_data1 , sample_data2 , sample_data3, sample_data4 ;
          input reset;
          input new_sound_sample;
          input vclock;                                        // 65MHz clock
          input [10:0] hcount;  // horizontal index of current pixel (0..1023)
          input [9:0]          vcount;    // vertical index of current pixel (0..767)
          input hsync;                                         // XVGA horizontal sync signal (active low)
          input vsync;                                         // XVGA vertical sync signal (active low)
          input blank;                                         // XVGA blanking (1 means output black pixel)

          output phsync;                             // pong game's horizontal sync
          output pvsync;                             // pong game's vertical sync
          output pblank;                             // pong game's blanking
          output [2:0] pixel;              // pong game's pixel

          input signed [7:0] sound_data;

//        reg phsync, pvsync, pblank ;   //declare as registers
          wire do_stuff;


          reg                              tmp , tmp_ph, tmp_pv, tmp_pb , old_vsync;           //for storing previous clock cycle's values
          always @ (posedge vclock) begin
          //         {phsync , tmp_ph} <= {tmp_ph, hsync};   //delay phsync, pvsync, pblank by 1 clock cycle
          //         {pvsync , tmp_pv} <= {tmp_pv, vsync};
          //         {pblank , tmp_pb} <= {tmp_pb, blank};
                     {old_vsync , tmp } <= { tmp , vsync};       //store old vsync to detect high->low transition
          end
          assign {phsync, pvsync, pblank} = {hsync, vsync, blank};

          assign do_stuff = old_vsync & ~vsync;    //true on vsync high->low transition
          wire[2:0] pixel_line , pixel_w1, pixel_w2 , pixel_w3 , pixel_w4 , pixel_w5;
          wire[2:0] pixel_hd1 , pixel_hd2 , pixel_hd3 , pixel_hd4 , pixel_hd5;
          wire[2:0] pixel_sd;
          wire[19:0] none;

          wd2 wave1 (vclock , new_sample1 , sample_data1 , hcount , vcount , hsync , vsync , blank  , pixel_w1 , reset );
                     defparam wave1.top_offset = 100;
                     defparam wave1.color = 3'b100;
          wd2 wave2 (vclock , new_sample2 , sample_data2 , hcount , vcount , hsync , vsync , blank , pixel_w2 , reset);
                     defparam wave2.top_offset = 200;
                     defparam wave2.color = 3'b010;
          wd2 wave3 (vclock , new_sample3 , sample_data3 , hcount , vcount , hsync , vsync , blank  , pixel_w3 , reset);
                     defparam wave3.top_offset = 300;
                     defparam wave3.color = 3'b001;
          wd2 wave4 (vclock , new_sample4 , sample_data4 , hcount , vcount , hsync , vsync , blank  , pixel_w4 , reset);
                     defparam wave4.top_offset = 400;
                     defparam wave4.color = 3'b110;

          sd2 sound_wave (vclock , new_sound_sample , sound_data , hcount , vcount , hsync , vsync , blank , none[15] , none[16] , none[17] , pixel_sd);
                     defparam sound_wave.top_offset = 672;
                     defparam sound_wave.color = 3'b111;


          lines l( vcount, hcount , pixel_line);

          wire [7:0] hit_out1 , hit_out2 , hit_out3 , hit_out4 , hit_out5;

          DisplayHit pad1_hit( sample_data1 , vclock , hit_out1 , reset );
          DisplayHit pad2_hit( sample_data2 , vclock , hit_out2 , reset );
          DisplayHit pad3_hit( sample_data3 , vclock , hit_out3 , reset );
          DisplayHit pad4_hit( sample_data4 , vclock , hit_out4 , reset );


          hit_display pad1_disp ( vcount, hcount , pixel_hd1 , hit_out1 );
                     defparam pad1_disp.color = 3'b100;
                     defparam pad1_disp.top_offset = 590;
```

```
        hit_display pad2_disp ( vcount, hcount , pixel_hd2 , hit_out2 );
                defparam pad2_disp.color = 3'b010;
                defparam pad2_disp.top_offset = 630;

        hit_display pad3_disp ( vcount, hcount , pixel_hd3 , hit_out3 );
                defparam pad3_disp.color = 3'b001;
                defparam pad3_disp.top_offset = 670;

        hit_display pad4_disp ( vcount, hcount , pixel_hd4 , hit_out4 );
                defparam pad4_disp.color = 3'b110;
                defparam pad4_disp.top_offset = 710;

/*      hit_display_circ pad1_disp ( vcount, hcount , pixel_hd1 , hit_out1 );
                defparam pad1_disp.color = 3'b100;
                defparam pad1_disp.top_offset = 620;
                defparam pad1_disp.left_offset = 70;
                defparam pad1_disp.low_bit = 2;
        hit_display_circ pad2_disp ( vcount, hcount , pixel_hd2 , hit_out2 );
                defparam pad2_disp.color = 3'b010;
                defparam pad2_disp.top_offset = 620;
                defparam pad2_disp.left_offset = 140;
                defparam pad2_disp.low_bit = 2;
        hit_display_circ pad3_disp ( vcount, hcount , pixel_hd3 , hit_out4 );
                defparam pad3_disp.color = 3'b001;
                defparam pad3_disp.top_offset = 700;
                defparam pad3_disp.left_offset = 105;
                defparam pad3_disp.low_bit = 1;
                 */



        assign pixel = ( pixel_line != 0) ? pixel_line :
                                        ( pixel_w1 != 0 ) ? pixel_w1 :
                                                (pixel_w2 != 0 ) ? pixel_w2 :
                                                        (pixel_w3 != 0 ) ? pixel_w3 :
                                                                (pixel_w4 != 0 ) ? pixel_w4 :
                                                                        (pixel_w5 != 0 ) ? pixel_w5 :
                                                                                ( pixel_hd1 != 0 ) ? pixel_hd1 :
                                                                                        ( pixel_hd2 != 0 ) ? pixel_hd2 :
                                                                                                ( pixel_hd3 != 0 ) ? pixel_hd3 :
                                                                                                        ( pixel_hd4 != 0 ) ? pixel_hd4 :
                                                                                                                ( pixel_sd != 0 ) ?
pixel_sd : 3'b000;

endmodule

module lines( vcount , hcount, pixel);
        input [10:0] hcount;
        input [9:0] vcount;
        output [2:0] pixel;
        assign pixel = ((hcount == 256 && vcount >  575) ||vcount ==  0 || vcount == 575 || vcount == 767 || hcount == 0 || hcount == 1023) ? 3'b111 : 3'b000;
endmodule

module hit_display( vcount , hcount, pixel , data );
        parameter color = 3'b111;
        parameter top_offset = 0;
        input [10:0] hcount;
        input [9:0] vcount;
        output [2:0] pixel;
        input [7:0] data;
        assign pixel = ( vcount > top_offset && vcount < top_offset + 20 && hcount > 0 && hcount < data * 2) ? color :
                                ( vcount > top_offset && vcount < top_offset + 20 && hcount == 256 ) ? 3'b111 : 3'b000;
endmodule

module hit_display_circ( vcount , hcount, pixel , data );
        parameter color = 3'b111;
        parameter top_offset = 0;
        parameter left_offset = 0;
        parameter low_bit = 2;
        input [10:0] hcount;
        input [9:0] vcount;
        output [2:0] pixel;
        input [7:0] data;
        wire [40:0] dsq;
```

```
		assign dsq = (hcount - left_offset)*(hcount - left_offset) + (vcount - top_offset) * ( vcount-top_offset);
		assign pixel = ( dsq > (data[6:low_bit] * data[6:low_bit]) ) ? 3'b000 : color;

		//assign pixel = ( vcount > top_offset && vcount < top_offset + 20 && hcount > 0 && hcount < data * 2) ? color :
		//					( vcount > top_offset && vcount < top_offset + 20 && hcount == 256 ) ? 3'b111 : 3'b000;
endmodule

 /*
module sd (vclock, new_sample, sample_data,
			hcount,vcount,hsync,vsync,blank,
			phsync,pvsync,pblank,pixel);

		parameter top_offset = 384;
		parameter color = 3'b111;
		input new_sample;

		input signed [7:0] sample_data;
		input vclock;						// 65MHz clock
		input [10:0] hcount;	// horizontal index of current pixel (0..1023)
		input [9:0]		vcount;	// vertical index of current pixel (0..767)
		input hsync;						// XVGA horizontal sync signal (active low)
		input vsync;						// XVGA vertical sync signal (active low)
		input blank;						// XVGA blanking (1 means output black pixel)

		output phsync;					// pong game's horizontal sync
		output pvsync;					// pong game's vertical sync
		output pblank;					// pong game's blanking
		output [2:0] pixel;			// pong game's pixel

		reg				old_vsync, tmp , tmp_ph, tmp_pv, tmp_pb;		//for storing previous clock cycle's values
		reg				game_over;					//1 if game should halt, 0 if it should play
		reg[9:0]		address , index;
		wire signed[7:0]		ram_out;
		wire do_stuff;
		reg				write;

		reg[9:0]				offset;
		sample_ram r( address, vclock , sample_data , ram_out, write);

		always @ (posedge vclock) begin
			{old_vsync , tmp } <= { tmp , vsync};	//store old vsync to detect high->low transition
		end

		assign {phsync, pvsync, pblank} = {hsync, vsync, blank};

		assign do_stuff = old_vsync & ~vsync;	//true on vsync high->low transition
		always@ (posedge vclock) begin

			if (do_stuff == 1)
				offset <= 0;
			if ( hcount == 1024) begin
				write <= 1;
				address <= index;
				index <= index + 1;
				offset <= offset+1;
			end
			else begin
				write <= 0;
				address <= index + hcount - offset -256;
			end
		end

		//assign output to monitor
		assign pixel =  ( hcount > 256 && vcount - top_offset -  ram_out  < 4 && vcount - top_offset - ram_out  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 :
3'b000 ;
endmodule


   /*

module wd (vclock, new_sample, sample_data,
			hcount,vcount,hsync,vsync,blank,
			pixel,reset);

		parameter top_offset = 384;
		parameter color = 3'b111;
	input new_sample;
	input reset;
	input signed [7:0] sample_data;
	input vclock;					// 65MHz clock
	input [10:0] hcount;	// horizontal index of current pixel (0..1023)
	input [9:0]		vcount;	// vertical index of current pixel (0..767)
	input hsync;					// XVGA horizontal sync signal (active low)
	input vsync;					// XVGA vertical sync signal (active low)
	input blank;					// XVGA blanking (1 means output black pixel)

	output [2:0] pixel;		// pong game's pixel
	reg [2:0] col;

	reg				old_vsync, tmp , tmp_ph, tmp_pv, tmp_pb;		//for storing previous clock cycle's values
	reg				game_over;				//1 if game should halt, 0 if it should play
	reg[9:0]		address, index ;

	wire signed[7:0]		ram_out ;
	wire do_stuff;
	reg					write;
	reg[9:0]			offset ;
    reg[10:0] x =0 ;
	reg read_ram1 = 0;

	sample_ram r(  address  , vclock , sample_data , ram_out, write);//write);



	always @ (posedge vclock) begin
		{old_vsync , tmp } <= { tmp , vsync};	//store old vsync to detect high->low transition
	end



	assign do_stuff = old_vsync & ~vsync;	//true on vsync high->low transition
	always@ (posedge vclock) begin

		if (do_stuff == 1)
			offset <= 0;

		if ( hcount == 1024) begin
			write <= 1;
			address <= index;
			index <= index + 1;
			offset <= offset+1;
		end
		else begin
			write <= 0;
			address <= index + hcount - offset;
		end



	end

	//assign output to monitor
	assign pixel =  ( vcount - top_offset -  ram_out  < 4 && vcount - top_offset - ram_out  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 : 3'b000 ;

endmodule

   */

module sd2 (vclock, new_sample, sample_data,
			hcount,vcount,hsync,vsync,blank,
			phsync,pvsync,pblank,pixel);

	parameter top_offset = 384;
	parameter color = 3'b111;
	input new_sample;

	input signed [7:0] sample_data;
	input vclock;					// 65MHz clock
	input [10:0] hcount;	// horizontal index of current pixel (0..1023)
	input [9:0]		vcount;	// vertical index of current pixel (0..767)
	input hsync;					// XVGA horizontal sync signal (active low)
	input vsync;					// XVGA vertical sync signal (active low)
	input blank;						// XVGA blanking (1 means output black pixel)

	output phsync;				// pong game's horizontal sync
	output pvsync;				// pong game's vertical sync
	output pblank;				// pong game's blanking
	output [2:0] pixel;			// pong game's pixel

	reg[9:0]		address_1 , address_2 , index;
	wire signed[7:0]		ram_1_out , ram_2_out , data_out;

	reg				read_ram_1;
	reg [12:0 ] sample_count;
	reg[9:0]			offset;
	sample_ram r( address_1, vclock , sample_data , ram_1_out, ~read_ram_1);
	sample_ram r2( address_2, vclock , sample_data , ram_2_out, read_ram_1);

	assign data_out = read_ram_1? ram_1_out : ram_2_out;


	assign {phsync, pvsync, pblank} = {hsync, vsync, blank};

	always@ (posedge vclock) begin
		sample_count <= sample_count + 1;

		if (read_ram_1 == 1) begin
			address_1 <= hcount;
			if ( sample_count == 1024) begin
				address_2 <= address_2 + 1;
				sample_count <= 0;
			end
		end
		else if (read_ram_1 == 0 ) begin
			address_2 <= hcount;
			if (sample_count == 1024 ) begin
				address_1 <= address_1 + 1;
				sample_count <= 0;
			end
		end

		if (hcount == 0 && vcount == 30)		begin
			address_1 <= 0;
			address_2 <= 0;
			read_ram_1 <= ~read_ram_1;

		end
	end

	//assign output to monitor
	assign pixel =  ( hcount > 256 && vcount - top_offset -  data_out  < 4 && vcount - top_offset - data_out  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 :
3'b000 ;
endmodule

module wd2 (vclock, new_sample, sample_data,
			hcount,vcount,hsync,vsync,blank,
			pixel,reset);

	parameter top_offset = 384;
	parameter color = 3'b111;
	input new_sample;
	input reset;
	input signed [7:0] sample_data;
	input vclock;						// 65MHz clock
	input [10:0]   hcount;		// horizontal index of current pixel (0..1023)
	input [9:0]		vcount;	// vertical index of current pixel (0..767)
	input hsync;						// XVGA horizontal sync signal (active low)
	input vsync;						// XVGA vertical sync signal (active low)
	input blank;						// XVGA blanking (1 means output black pixel)

	output [2:0] pixel;			// pong game's pixel

	reg[9:0]			address_1 , address_2;
	wire signed [7:0]	ram_1_out , ram_2_out , draw_data;

	reg[10:0]			sample_count = 0;

	reg read_ram_1 = 0;
```

```
        sample_ram r1(  address_1  , vclock , sample_data , ram_1_out, ~read_ram_1 );//write);
        sample_ram r2(  address_2  , vclock , sample_data , ram_2_out, read_ram_1 );//write);

        assign draw_data = read_ram_1 ==1  ? ram_1_out : ram_2_out;


        always@ (posedge vclock) begin
                sample_count <= sample_count + 1;

                if (read_ram_1 == 1) begin
                        address_1 <= hcount;
                        if ( sample_count == 1057) begin
                                address_2 <= address_2 + 1;
                                sample_count <= 0;
                        end
                end
                else if (read_ram_1 == 0 ) begin
                        address_2 <= hcount;
                        if (sample_count == 1057 ) begin
                                address_1 <= address_1 + 1;
                                sample_count <= 0;
                        end
                end

                if (hcount == 0 && vcount == 30)        begin
                        address_1 <= 0;
                        address_2 <= 0;
                        read_ram_1 <= ~read_ram_1;

                end

        /*      if (reset == 1 )
                        read_ram_1 <= 0;

                else if (hcount == 0 && vcount == top_offset - 128 ) begin
                        read_ram_1 <= ~read_ram_1;
                        address_1 <= 0;
                        address_2 <= 0;
                end

                else if ( sample_count == 439 ) begin
                        sample_count <= 0;
                        if ( read_ram_1 == 1)
                                address_2 <= address_2 + 1;
                        else
                                address_1 <= address_1 + 1;
                end
                else
                        sample_count <= sample_count + 1;


                if ( read_ram_1 == 1 )
                        address_1 <= hcount;
                else
                        address_2 <= hcount;
                                                                        */
        end

        //assign output to monitor
        assign pixel =  ( vcount - top_offset -  draw_data  < 4 && vcount - top_offset - draw_data  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 : 3'b000 ;

endmodule







module blob(x,y,hcount,vcount,pixel);
        parameter WIDTH = 64; // default width: 64 pixels
        parameter HEIGHT = 64; // default height: 64 pixels
        parameter COLOR = 3'b111; // default color: white
        input [10:0] x,hcount;
```

```
                input [9:0] y,vcount;
                output [2:0] pixel;
                reg [2:0] 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 = 0;
                end
endmodule

module blob_rot(x,y, , rot , hcount,vcount,pixel);
                parameter WIDTH = 64; // default width: 64 pixels
                parameter HEIGHT = 64; // default height: 64 pixels
                parameter COLOR = 3'b111; // default color: white
                input[10:0] rot;
                input [10:0] x,hcount;
                input [9:0] y,vcount;
                output [2:0] pixel;
                reg [2:0] 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 = 0;
                end
endmodule



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
```

```
/*module whole_display(        vclock,hcount,vcount,hsync,vsync,blank, phsync,pvsync,pblank,pixel ,
                                        new_sample1, sample_data1,
                                        new_sample2, sample_data2,
                                        new_sample3, sample_data3,
                                        new_sample4, sample_data4,
                                        sound_data , new_sound_sample , reset );

        input new_sample1, new_sample2 , new_sample3 , new_sample4;
        input signed [7:0] sample_data1 , sample_data2 , sample_data3, sample_data4 ;
        input reset;
        input new_sound_sample;
        input vclock;                                   // 65MHz clock
        input [10:0] hcount;  // horizontal index of current pixel (0..1023)
        input [9:0]           vcount;    // vertical index of current pixel (0..767)
        input hsync;                                    // XVGA horizontal sync signal (active low)
        input vsync;                                    // XVGA vertical sync signal (active low)
        input blank;                                    // XVGA blanking (1 means output black pixel)

        output phsync;                          // pong game's horizontal sync
        output pvsync;                          // pong game's vertical sync
        output pblank;                          // pong game's blanking
        output [2:0] pixel;             // pong game's pixel

        input signed [7:0] sound_data;

//      reg phsync, pvsync, pblank ;    //declare as registers
        wire do_stuff;


        reg                             tmp , tmp_pb , old_vsync;      //for storing previous clock cycle's values
        always @ (posedge vclock) begin
                {old_vsync , tmp } <= { tmp , vsync};   //store old vsync to detect high->low transition
        end
        assign {phsync, pvsync, pblank} = {hsync, vsync, blank};

        assign do_stuff = old_vsync & ~vsync;   //true on vsync high->low transition
        wire[2:0] pixel_line , pixel_w1, pixel_w2 , pixel_w3 , pixel_w4 , pixel_w5;
        wire[2:0] pixel_hd1 , pixel_hd2 , pixel_hd3 , pixel_hd4 , pixel_hd5;
        wire[2:0] pixel_sd;


        wd2 wave1 (vclock , new_sample1 , sample_data1 , hcount , vcount , hsync , vsync , blank  , pixel_w1 , reset );
                defparam wave1.top_offset = 100;
                defparam wave1.color = 3'b100;
        wd2 wave2 (vclock , new_sample2 , sample_data2 , hcount , vcount , hsync , vsync , blank , pixel_w2 , reset);
                defparam wave2.top_offset = 200;
                defparam wave2.color = 3'b010;
        wd2 wave3 (vclock , new_sample3 , sample_data3 , hcount , vcount , hsync , vsync , blank  , pixel_w3 , reset);
                defparam wave3.top_offset = 300;
                defparam wave3.color = 3'b001;
        wd2 wave4 (vclock , new_sample4 , sample_data4 , hcount , vcount , hsync , vsync , blank  , pixel_w4 , reset);
                defparam wave4.top_offset = 400;
                defparam wave4.color = 3'b110;

//      sd2 sound_wave (vclock , new_sound_sample , sound_data , hcount , vcount , hsync , vsync , blank , pixel_sd);
//              defparam sound_wave.top_offset = 672;
        //      defparam sound_wave.color = 3'b111;


        lines l( vcount, hcount , pixel_line);

        wire [7:0] hit_out1 , hit_out2 , hit_out3 , hit_out4 , hit_out5;

        DisplayHit pad1_hit( sample_data1 , vclock , hit_out1 , reset );
        DisplayHit pad2_hit( sample_data2 , vclock , hit_out2 , reset );
        DisplayHit pad3_hit( sample_data3 , vclock , hit_out3 , reset );
```

```
		DisplayHit pad4_hit( sample_data4 , vclock , hit_out4 , reset );


		hit_display pad1_disp ( vcount, hcount , pixel_hd1 , hit_out1 );
			defparam pad1_disp.color = 3'b100;
			defparam pad1_disp.top_offset = 590;

		hit_display pad2_disp ( vcount, hcount , pixel_hd2 , hit_out2 );
			defparam pad2_disp.color = 3'b010;
			defparam pad2_disp.top_offset = 630;

		hit_display pad3_disp ( vcount, hcount , pixel_hd3 , hit_out3 );
			defparam pad3_disp.color = 3'b001;
			defparam pad3_disp.top_offset = 670;

		hit_display pad4_disp ( vcount, hcount , pixel_hd4 , hit_out4 );
			defparam pad4_disp.color = 3'b110;
			defparam pad4_disp.top_offset = 710;




		assign pixel = ( pixel_line != 0) ? pixel_line :
								( pixel_w1 != 0 ) ? pixel_w1 :
									(pixel_w2 != 0 ) ? pixel_w2 :
										(pixel_w3 != 0 ) ? pixel_w3 :
											(pixel_w4 != 0 ) ? pixel_w4 :
												(pixel_w5 != 0 ) ? pixel_w5 :
													( pixel_hd1 != 0 ) ? pixel_hd1 :
														( pixel_hd2 != 0 ) ? pixel_hd2 :
															( pixel_hd3 != 0 ) ? pixel_hd3 :
																( pixel_hd4 != 0 ) ? pixel_hd4 :
																	( pixel_sd != 0 ) ?
pixel_sd : 3'b000;

endmodule

module lines( vcount , hcount, pixel);
		input [10:0] hcount;
		input [9:0] vcount;
		output [2:0] pixel;
		assign pixel = ((hcount == 256 && vcount >  575) ||vcount ==  0 || vcount == 575 || vcount == 767 || hcount == 0 || hcount == 1023) ? 3'b111 : 3'b000;
endmodule

module hit_display( vcount , hcount, pixel , data );
		parameter color = 3'b111;
		parameter top_offset = 0;
		input [10:0] hcount;
		input [9:0] vcount;
		output [2:0] pixel;
		input [7:0] data;
		assign pixel = ( vcount > top_offset && vcount < top_offset + 20 && hcount > 0 && hcount < data * 2) ? color :
						( vcount > top_offset && vcount < top_offset + 20 && hcount == 256 ) ? 3'b111 : 3'b000;
endmodule

module hit_display_circ( vcount , hcount, pixel , data );
		parameter color = 3'b111;
		parameter top_offset = 0;
		parameter left_offset = 0;
		parameter low_bit = 2;
		input [10:0] hcount;
		input [9:0] vcount;
		output [2:0] pixel;
		input [7:0] data;
		wire [40:0] dsq;

		assign dsq = (hcount - left_offset)*(hcount - left_offset) + (vcount - top_offset) * ( vcount-top_offset);
		assign pixel = ( dsq > (data[6:low_bit] * data[6:low_bit]) ) ? 3'b000 : color;

		//assign pixel = ( vcount > top_offset && vcount < top_offset + 20 && hcount > 0 && hcount < data * 2) ? color :
		//				( vcount > top_offset && vcount < top_offset + 20 && hcount == 256 ) ? 3'b111 : 3'b000;
endmodule

 /*
module sd (vclock, new_sample, sample_data,
                    hcount,vcount,hsync,vsync,blank,
                    phsync,pvsync,pblank,pixel);

        parameter top_offset = 384;
        parameter color = 3'b111;
        input new_sample;

        input signed [7:0] sample_data;
        input vclock;                                   // 65MHz clock
        input [10:0] hcount;  // horizontal index of current pixel (0..1023)
        input [9:0]         vcount;     // vertical index of current pixel (0..767)
        input hsync;                                    // XVGA horizontal sync signal (active low)
        input vsync;                                    // XVGA vertical sync signal (active low)
        input blank;                                    // XVGA blanking (1 means output black pixel)

        output phsync;                          // pong game's horizontal sync
        output pvsync;                          // pong game's vertical sync
        output pblank;                          // pong game's blanking
        output [2:0] pixel;             // pong game's pixel

        reg                             old_vsync, tmp , tmp_ph, tmp_pv, tmp_pb;        //for storing previous clock cycle's values
        reg                             game_over;                      //1 if game should halt, 0 if it should play
        reg[9:0]            address , index;
        wire signed[7:0]                ram_out;
        wire do_stuff;
        reg                             write;

        reg[9:0]                        offset;
        sample_ram r( address, vclock , sample_data , ram_out, write);

        always @ (posedge vclock) begin
                {old_vsync , tmp } <= { tmp , vsync};   //store old vsync to detect high->low transition
        end

        assign {phsync, pvsync, pblank} = {hsync, vsync, blank};

        assign do_stuff = old_vsync & ~vsync;    //true on vsync high->low transition
        always@ (posedge vclock) begin

                if (do_stuff == 1)
                        offset <= 0;
                if ( hcount == 1024) begin
                        write <= 1;
                        address <= index;
                        index <= index + 1;
                        offset <= offset+1;
                end
                else begin
                        write <= 0;
                        address <= index + hcount - offset -256;
                end
        end

        //assign output to monitor
        assign pixel =  ( hcount > 256 && vcount - top_offset -  ram_out  < 4 && vcount - top_offset - ram_out  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 :
3'b000 ;
endmodule


  /*

module wd (vclock, new_sample, sample_data,
                    hcount,vcount,hsync,vsync,blank,
                    pixel,reset);

        parameter top_offset = 384;
        parameter color = 3'b111;
        input new_sample;
        input reset;
        input signed [7:0] sample_data;
        input vclock;                                   // 65MHz clock
        input [10:0] hcount;  // horizontal index of current pixel (0..1023)
        input [9:0]         vcount;     // vertical index of current pixel (0..767)
        input hsync;                                    // XVGA horizontal sync signal (active low)
        input vsync;                                    // XVGA vertical sync signal (active low)
        input blank;                                    // XVGA blanking (1 means output black pixel)
```

```
	output [2:0] pixel;			// pong game's pixel
	reg [2:0] col;

	reg				old_vsync, tmp , tmp_ph, tmp_pv, tmp_pb;		//for storing previous clock cycle's values
	reg				game_over;					//1 if game should halt, 0 if it should play
	reg[9:0]			address, index ;

	wire signed[7:0]		ram_out ;
	wire do_stuff;
	reg						write;
	reg[9:0]			offset ;
   reg[10:0] x =0 ;
	reg read_ram1 = 0;

	sample_ram r(  address  , vclock , sample_data , ram_out, write);//write);



	always @ (posedge vclock) begin
		{old_vsync , tmp } <= { tmp , vsync};	//store old vsync to detect high->low transition
	end



	assign do_stuff = old_vsync & ~vsync;	//true on vsync high->low transition
	always@ (posedge vclock) begin

		if (do_stuff == 1)
			offset <= 0;

		if ( hcount == 1024) begin
			write <= 1;
			address <= index;
			index <= index + 1;
			offset <= offset+1;
		end
		else begin
			write <= 0;
			address <= index + hcount - offset;
		end



	end

	//assign output to monitor
	assign pixel =  ( vcount - top_offset -  ram_out  < 4 && vcount - top_offset - ram_out  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 : 3'b000 ;

endmodule

  */
 /*
module sd2 (vclock, new_sample, sample_data,
				hcount,vcount,hsync,vsync,blank,
				pixel);

	parameter top_offset = 384;
	parameter color = 3'b111;
	input new_sample;

	input signed [7:0] sample_data;
	input vclock;					// 65MHz clock
	input [10:0] hcount;  // horizontal index of current pixel (0..1023)
	input [9:0]		vcount;	// vertical index of current pixel (0..767)
	input hsync;					// XVGA horizontal sync signal (active low)
	input vsync;					// XVGA vertical sync signal (active low)
	input blank;					// XVGA blanking (1 means output black pixel)

	output [2:0] pixel;			// pong game's pixel

	reg[9:0]		address_1 , address_2 , index;
	wire signed[7:0]		ram_1_out , ram_2_out , data_out;

	reg				read_ram_1;
	reg [12:0 ] sample_count;
	reg[9:0]			offset;
	sample_ram r( address_1, vclock , sample_data , ram_1_out, ~read_ram_1);
	sample_ram r2( address_2, vclock , sample_data , ram_2_out, read_ram_1);

	assign data_out = read_ram_1? ram_1_out : ram_2_out;


	assign {phsync, pvsync, pblank} = {hsync, vsync, blank};

	always@ (posedge vclock) begin
		sample_count <= sample_count + 1;

		if (read_ram_1 == 1) begin
			address_1 <= hcount;
			if ( sample_count == 1024) begin
				address_2 <= address_2 + 1;
				sample_count <= 0;
			end
		end
		else if (read_ram_1 == 0 ) begin
			address_2 <= hcount;
			if (sample_count == 1024 ) begin
				address_1 <= address_1 + 1;
				sample_count <= 0;
			end
		end

		if (hcount == 0 && vcount == 30)		begin
			address_1 <= 0;
			address_2 <= 0;
			read_ram_1 <= ~read_ram_1;

		end
	end

	//assign output to monitor
	assign pixel =  ( hcount > 256 && vcount - top_offset -  data_out  < 4 && vcount - top_offset - data_out  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 :
3'b000 ;
endmodule

module wd2 (vclock, new_sample, sample_data,
			hcount,vcount,hsync,vsync,blank,
			pixel,reset);

	parameter top_offset = 384;
	parameter color = 3'b111;
	input new_sample;
	input reset;
	input signed [7:0] sample_data;
	input vclock;					// 65MHz clock
	input [10:0]   hcount;		// horizontal index of current pixel (0..1023)
	input [9:0]		vcount;	// vertical index of current pixel (0..767)
	input hsync;					// XVGA horizontal sync signal (active low)
	input vsync;					// XVGA vertical sync signal (active low)
	input blank;					// XVGA blanking (1 means output black pixel)

	output [2:0] pixel;			// pong game's pixel

	reg[9:0]			address_1 , address_2;
	wire signed [7:0]	ram_1_out , ram_2_out , draw_data;

	reg[10:0]			sample_count = 0;

	reg read_ram_1 = 0;

	sample_ram r1(  address_1  , vclock , sample_data , ram_1_out, ~read_ram_1 );//write);
	sample_ram r2(  address_2  , vclock , sample_data , ram_2_out, read_ram_1 );//write);

	assign draw_data = read_ram_1 ==1  ? ram_1_out : ram_2_out;


	always@ (posedge vclock) begin
		sample_count <= sample_count + 1;

		if (read_ram_1 == 1) begin
			address_1 <= hcount;
```

```
                        if ( sample_count == 1057) begin
                                address_2 <= address_2 + 1;
                                sample_count <= 0;
                        end
                end
                else if (read_ram_1 == 0 ) begin
                        address_2 <= hcount;
                        if (sample_count == 1057 ) begin
                                address_1 <= address_1 + 1;
                                sample_count <= 0;
                        end
                end

                if (hcount == 0 && vcount == 30)        begin
                        address_1 <= 0;
                        address_2 <= 0;
                        read_ram_1 <= ~read_ram_1;

                end

        /*      if (reset == 1 )
                        read_ram_1 <= 0;

                else if (hcount == 0 && vcount == top_offset - 128 ) begin
                        read_ram_1 <= ~read_ram_1;
                        address_1 <= 0;
                        address_2 <= 0;
                end

                else if ( sample_count == 439 ) begin
                        sample_count <= 0;
                        if ( read_ram_1 == 1)
                                address_2 <= address_2 + 1;
                        else
                                address_1 <= address_1 + 1;
                end
                else
                        sample_count <= sample_count + 1;


                if ( read_ram_1 == 1 )
                        address_1 <= hcount;
                else
                        address_2 <= hcount;
                                                                */
/*      end

        //assign output to monitor
        assign pixel =  ( vcount - top_offset -  draw_data  < 4 && vcount - top_offset - draw_data  > -1 )? color : 3'b000;//== ram_out ) ? 3'b111 : 3'b000 ;

endmodule
                 */



                                                                /*




module blob(x,y,hcount,vcount,pixel);
        parameter WIDTH = 64; // default width: 64 pixels
        parameter HEIGHT = 64; // default height: 64 pixels
        parameter COLOR = 3'b111; // default color: white
        input [10:0] x,hcount;
        input [9:0] y,vcount;
        output [2:0] pixel;
        reg [2:0] 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 = 0;
        end
endmodule
```

```
module blob_rot(x,y, , rot , hcount,vcount,pixel);
           parameter WIDTH = 64; // default width: 64 pixels
           parameter HEIGHT = 64; // default height: 64 pixels
           parameter COLOR = 3'b111; // default color: white
           input[10:0] rot;
           input [10:0] x,hcount;
           input [9:0] y,vcount;
           output [2:0] pixel;
           reg [2:0] 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 = 0;
           end
endmodule



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
                        */


module game_sounds(clock, reset,  ready, to_ac97_data , play_jump , play_music);
           input clock;            // 27mhz system clock
           input reset;               // 1 for playback, 0 for record
           input ready;                 // 1 when AC97 data is available     // 8-bit PCM data from mic
           output signed [19:0] to_ac97_data;
           input play_jump;
           input play_music;
           reg[16:0] maddress;
           reg[14:0] address;
           wire signed [7:0] data;
```

```
	wire signed [7:0] mdata;
	wire signed [18:0] sound;

	assign sound[10:0] = 11'b00000000000;
	assign sound[18:11] = data;

	wire signed [18:0] msound;

	assign msound[10:0] = 11'b00000000000;
	assign msound[18:11] = mdata;

	// detect clock cycle when READY goes 0 -> 1
	// f(READY) = 48khz
	wire new_frame;
	reg old_ready;

	reg [2:0] frame_count = 0;

	always @ (posedge clock) old_ready <= reset ? 0 : ready;
	assign new_frame = ready & ~old_ready;
	reg signed [19:0] to_ac97_data;

	jump_rom romjump( address, clock , data);
	//music_rom musicr(maddress,clock,mdata);

	always @ (posedge clock) begin
		if (reset == 1) begin
			address <= 0; maddress <= 1;
		end
		else if ( play_jump == 1)
			address <= 0;
		else	if ( address < 26260 ) begin
			if (new_frame == 1)
				address <= address + 1;
		end
		if ( maddress < 72659 ) begin
			if (new_frame == 1) begin
				if (frame_count == 0)
					maddress <= maddress + 1;
				frame_count <= frame_count + 1;
			end
		end
		else maddress <= 0;
		to_ac97_data  <= sound + msound;
	end

endmodule

module recorder(clock_27mhz, reset,  ready, from_ac97_data, to_ac97_data_left , to_ac97_data_right , snare_pad , bass_pad , hh_pad, led , switch);

	parameter SNARE_MAX = 19999;
	parameter BASS_MAX = 19999;
	parameter HH_MAX = 19999;

	input[7:0] switch;
	input clock_27mhz;		// 27mhz system clock
	input reset;			// 1 for playback, 0 for record
	input ready;			// 1 when AC97 data is available
	input [7:0] from_ac97_data;	// 8-bit PCM data from mic
	output [19:0] to_ac97_data_left , to_ac97_data_right;
	input [7:0] snare_pad , bass_pad , hh_pad;
	output [6:0] led;
	reg [6:0] led;
	reg play_snare , play_bass , play_hh;


	reg [7:0] snare_int , hh_int , bass_int;


	// detect clock cycle when READY goes 0 -> 1
	// f(READY) = 48khz
	wire new_frame;
	reg old_ready;
	always @ (posedge clock_27mhz) old_ready <= reset ? 0 : ready;
	assign new_frame = ready & ~old_ready;
```

```
		reg [19:0] to_ac97_data_left , to_ac97_data_right;

		wire signed [15:0] snare_data , bass_data , hh_data;
		wire signed [15:0] snare_data2 , bass_data2 , hh_data2;

		reg[14:0] snare_address , bass_address , hh_address;


//		snare_rom snr ( snare_address , clock_27mhz , snare_data );
//		bass_rom br   ( bass_address  , clock_27mhz , bass_data );
  //   hh_rom h      ( hh_address    , clock_27mhz , hh_data );
		assign snare_data = 0;
		assign bass_data = 0;
		assign hh_data = 0;
		snare_rom2 snr2 ( snare_address , clock_27mhz , snare_data2 );
		bass_rom2 br2   ( bass_address  , clock_27mhz , bass_data2 );
     hh_rom2 h2      ( hh_address    , clock_27mhz , hh_data2 );


//		wire signed [19:0] tmp_snare1 , tmp_bass1 , tmp_hh1;
//		wire signed [19:0] tmp_snare2 , tmp_bass2 , tmp_hh2;
///		assign tmp_snare1 = play_snare ?  {snare_data < 0 ? 2'b11 : 2'b00 , ( switch[7] ? snare_data : snare_data2 ), 2'b00} : 0;
//		assign tmp_hh1 = play_hh       ?   {  hh_data < 0 ? 3'b111 : 2'b00 ,( switch[7] ? hh_data :  hh_data2 )  , 1'b0} : 0;
//		assign tmp_bass1 = play_bass   ? {  bass_data  < 0 ? 2'b11 : 2'b00 ,( switch[7] ? bass_data :  bass_data2 ), 2'b00} : 0;



		wire signed [21:0] dsnare , dhh , dbass;
		wire zero;
		assign zero = 16'h0000;
		multd md1( clock_27mhz , (play_snare ? ( switch[7] ? snare_data : snare_data2 ) : zero ) , snare_int[6:3] , dsnare );
		multd md2( clock_27mhz , (play_hh    ? ( switch[7] ? hh_data    :  hh_data2 )   : zero )   , hh_int[6:3]    , dhh );
		multd md3( clock_27mhz , (play_bass  ? ( switch[7] ? bass_data  :  bass_data2 ) : zero ) , bass_int[6:3]  , dbass);

//		assign tmp_snare1 = play_snare ?  {snare_data < 0 ? 2'b11 : 2'b00 , snare_data, 2'b00} : 0;
//		assign tmp_hh1 = play_hh       ?   {  hh_data < 0 ? 3'b111 : 2'b00 , hh_data   , 1'b0} : 0;
//		assign tmp_bass1 = play_bass   ? {  bass_data  < 0 ? 2'b11 : 2'b00 , bass_data , 2'b00} : 0;

//		assign tmp_snare2 = ( snare_int * tmp_snare1 ) >> 3;

//		wire signed [19:0] tmp2 , tmp3;

//		assign tmp3 = tmp_snare2+ tmp_hh1 + tmp_bass1;

//		assign tmp2 = {tmp_snare , 4'b0000};
//		wire signed [23:0] mult_out_snare , mult_out_bass , mult_out_hh;

//		smult s( clock_27mhz ,  tmp_snare1 , snare_int[6:3] , mult_out_snare );
//		smult s2( clock_27mhz , tmp_bass1 , bass_int[6:3] , mult_out_bass );
//		smult s3( clock_27mhz , tmp_hh1 , hh_int[6:3] , mult_out_hh );



		always @ (posedge clock_27mhz) begin
			//led[3] <= ~switch[0];
			//led[4] <= 1;
			if (switch[0] == 0 && switch[1] == 0)
				led[6:0] <= ~snare_int[6:0];
			else if (switch[1] == 0 && switch[0] ==1 )
				led[6:0] <= ~bass_int[7:0];
			else
				led[6:0] <= ~hh_int[7:0];


			if (new_frame) begin
				if ( play_snare == 1 )
					snare_address <= snare_address + 1;
				if ( play_bass == 1)
					bass_address <= bass_address + 1;
				if ( play_hh == 1 )
					hh_address <= hh_address + 1;
					// just received new data from the AC97

			end
		//		if ( switch[3] ==1 )
		//				to_ac97_data <= mult_out_snare[22:3] + mult_out_bass[21:2] + mult_out_hh[22:3];
```

```
//		else
		to_ac97_data_left  <= dsnare[21:2] + dbass[21:2];
		to_ac97_data_right <= dbass[21:2]  + dhh[21:2];
			//to_ac97_data <= (( tmp_snare1 * snare_int[6:4] ) >> 3) + ( (tmp_bass1 * bass_int[6:4]) >> 3 ) + ( (tmp_hh1 * hh_int[6:4]) >> 3);
			//to_ac97_data <= tmp_snare1 + tmp_hh1 + tmp_bass1;

		if ( reset == 1) begin
			play_snare <= 0;
			play_bass <= 0;
			play_hh <= 0;
			snare_int <= 0;
			bass_int <= 0;
			hh_int <= 0;
			led[3:0] <= 4'b1111;
		end
		else begin
			if ( snare_pad > 0 ) begin
				snare_address <= 0;
				play_snare <= 1;
				snare_int <= snare_pad;
			//	led[0] <= 0;
			end
			else if ( snare_address > SNARE_MAX ) begin
				play_snare <= 0;
			//	led[0] <= 1;
			end

			if ( bass_pad > 0 ) begin
				bass_address <= 0;
				play_bass <= 1;
				bass_int <= bass_pad;
			//	led[1] <= 0;
			end
			else if ( bass_address > BASS_MAX ) begin
				play_bass <= 0;
			//	led[1] <= 1;
			end

			if ( hh_pad > 0 ) begin
				hh_address <= 0;
				play_hh <= 1;
				hh_int <= hh_pad;
			//	led[2] <= 0;
			end
			else if ( hh_address > HH_MAX ) begin
				play_hh <= 0;
		//			led[2] <= 1;
			end
		end

	end


endmodule





module drumaudio(clock_27mhz, reset,  ready, to_ac97_data_left , to_ac97_data_right , snare_pad , bass_pad , hh_pad, led , switch);

	parameter SNARE_MAX = 19999;
	parameter BASS_MAX = 19999;
	parameter HH_MAX = 19999;

	input[7:0] switch;
	input clock_27mhz;		// 27mhz system clock
	input reset;		// 1 for playback, 0 for record
	input ready;		// 1 when AC97 data is available
	output [19:0] to_ac97_data_left , to_ac97_data_right;
	input [7:0] snare_pad , bass_pad , hh_pad;
	output [6:0] led;
	reg [6:0] led;
	reg play_snare , play_bass , play_hh;
```

```
            reg [7:0] snare_int , hh_int , bass_int;


            // detect clock cycle when READY goes 0 -> 1
            // f(READY) = 48khz
            wire new_frame;
            reg old_ready;
            always @ (posedge clock_27mhz) old_ready <= reset ? 0 : ready;
            assign new_frame = ready & ~old_ready;

            reg [19:0] to_ac97_data_left , to_ac97_data_right;

            wire signed [15:0] snare_data , bass_data , hh_data;
            wire signed [15:0] snare_data2 , bass_data2 , hh_data2;

            reg[14:0] snare_address , bass_address , hh_address;


            snare_rom snr ( snare_address , clock_27mhz , snare_data );
            bass_rom br   ( bass_address  , clock_27mhz , bass_data );
    hh_rom h      ( hh_address    , clock_27mhz , hh_data );
//          assign snare_data    = 0;
//          assign bass_data  = 0;
//          assign hh_data = 0;
            snare_rom2 snr2 ( snare_address , clock_27mhz , snare_data2 );
            bass_rom2 br2   ( bass_address  , clock_27mhz , bass_data2 );
    hh_rom2 h2      ( hh_address    , clock_27mhz , hh_data2 );


//          wire signed [19:0] tmp_snare1 , tmp_bass1 , tmp_hh1;
//          wire signed [19:0] tmp_snare2 , tmp_bass2 , tmp_hh2;
///         assign tmp_snare1 = play_snare ?  {snare_data < 0 ? 2'b11 : 2'b00 , ( switch[7] ? snare_data : snare_data2 ), 2'b00} : 0;
//          assign tmp_hh1 = play_hh       ?   {  hh_data < 0 ? 3'b111 : 2'b00 ,( switch[7] ? hh_data :  hh_data2 )   , 1'b0} : 0;
//          assign tmp_bass1 = play_bass   ? {  bass_data  < 0 ? 2'b11 : 2'b00 ,( switch[7] ? bass_data :  bass_data2 ), 2'b00} : 0;



            wire signed [21:0] dsnare , dhh , dbass;
            wire zero;
            assign zero = 16'h0000;
            multd md1( clock_27mhz , (play_snare ? ( switch[7] ? snare_data : snare_data2 ) : zero ) , snare_int[6:3] , dsnare );
            multd md2( clock_27mhz , (play_hh    ? ( switch[7] ? hh_data    :  hh_data2 )   : zero )   , hh_int[6:3]    , dhh );
            multd md3( clock_27mhz , (play_bass  ? ( switch[7] ? bass_data  :  bass_data2 ) : zero ) , bass_int[6:3]  , dbass);

//          assign tmp_snare1 = play_snare ?  {snare_data < 0 ? 2'b11 : 2'b00 , snare_data, 2'b00} : 0;
//          assign tmp_hh1 = play_hh       ?   {  hh_data < 0 ? 3'b111 : 2'b00 , hh_data   , 1'b0} : 0;
//          assign tmp_bass1 = play_bass   ? {  bass_data  < 0 ? 2'b11 : 2'b00 , bass_data , 2'b00} : 0;

//          assign tmp_snare2 = ( snare_int * tmp_snare1 ) >> 3;

//          wire signed [19:0] tmp2 , tmp3;

//          assign tmp3 = tmp_snare2+ tmp_hh1 + tmp_bass1;

//          assign tmp2 = {tmp_snare , 4'b0000};
//          wire signed [23:0] mult_out_snare , mult_out_bass , mult_out_hh;

//          smult s( clock_27mhz ,  tmp_snare1 , snare_int[6:3] , mult_out_snare );
//          smult s2( clock_27mhz , tmp_bass1 , bass_int[6:3] , mult_out_bass );
//          smult s3( clock_27mhz , tmp_hh1 , hh_int[6:3] , mult_out_hh );



            always @ (posedge clock_27mhz) begin
                        //led[3] <= ~switch[0];
                        //led[4] <= 1;
                        if (switch[0] == 0 && switch[1] == 0)
                                    led[6:0] <= ~snare_int[6:0];
                        else if (switch[1] == 0 && switch[0] ==1 )
                                    led[6:0] <= ~bass_int[7:0];
                        else
                                    led[6:0] <= ~hh_int[7:0];
```

```
		if (new_frame) begin
			if ( play_snare == 1 )
				snare_address <= snare_address + 1;
			if ( play_bass == 1)
				bass_address <= bass_address + 1;
			if ( play_hh == 1 )
				hh_address <= hh_address + 1;
				// just received new data from the AC97

		end
//		if ( switch[3] ==1 )
//			to_ac97_data <= mult_out_snare[22:3] + mult_out_bass[21:2] + mult_out_hh[22:3];
//		else
		to_ac97_data_left  <= dsnare[21:2] + dbass[21:2];
		to_ac97_data_right <= dbass[21:2]  + dhh[21:2];
			//to_ac97_data <= (( tmp_snare1 * snare_int[6:4] ) >> 3) + ( (tmp_bass1 * bass_int[6:4]) >> 3 ) + ( (tmp_hh1 * hh_int[6:4]) >> 3);
			//to_ac97_data <= tmp_snare1 + tmp_hh1 + tmp_bass1;

		if ( reset == 1) begin
			play_snare <= 0;
			play_bass <= 0;
			play_hh <= 0;
			snare_int <= 0;
			bass_int <= 0;
			hh_int <= 0;
			led[3:0] <= 4'b1111;
		end
		else begin
			if ( snare_pad > 0 ) begin
				snare_address <= 0;
				play_snare <= 1;
				snare_int <= snare_pad;
			//	led[0] <= 0;
			end
			else if ( snare_address > SNARE_MAX ) begin
				play_snare <= 0;
			//	led[0] <= 1;
			end

			if ( bass_pad > 0 ) begin
				bass_address <= 0;
				play_bass <= 1;
				bass_int <= bass_pad;
			//	led[1] <= 0;
			end
			else if ( bass_address > BASS_MAX ) begin
				play_bass <= 0;
			//	led[1] <= 1;
			end

			if ( hh_pad > 0 ) begin
				hh_address <= 0;
				play_hh <= 1;
				hh_int <= hh_pad;
			//	led[2] <= 0;
			end
			else if ( hh_address > HH_MAX ) begin
				play_hh <= 0;
		//			led[2] <= 1;
			end
		end

	end


endmodule
```

```
module audio (clock_27mhz, reset, audio_in_data, audio_out_data_left , audio_out_data_right, ready,
                      audio_reset_b, ac97_sdata_out, ac97_sdata_in,
              ac97_synch, ac97_bit_clock);

   input clock_27mhz;
   input reset;
   output [7:0] audio_in_data;
   input [19:0] audio_out_data_left, audio_out_data_right;
   output ready;

   //ac97 interface signals
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

   wire [4:0] volume;
   wire source;
   assign volume = 4'd22;  //a reasonable volume value
   assign source = 1;   //mic

   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   reg audio_reset_b;
   reg [9:0] reset_count;
   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
   end

   ac97 ac97(ready, command_address, command_data, command_valid,
             left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
             right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
             ac97_bit_clock);

   ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                     command_valid, volume, source);


   assign left_out_data = audio_out_data_left;
   assign right_out_data = audio_out_data_right;//left_out_data;

   //arbitrarily choose left input, get highest-order bits
   assign audio_in_data = left_in_data[19:12];

endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

   reg [19:0] l_cmd_addr;
   reg [19:0] l_cmd_data;
   reg [19:0] l_left_data, l_right_data;
   reg l_cmd_v, l_left_v, l_right_v;
   reg [19:0] left_in_data, right_in_data;

   initial begin
      ready <= 1'b0;
      // synthesis attribute init of ready is "0";
      ac97_sdata_out <= 1'b0;
      // synthesis attribute init of ac97_sdata_out is "0";
      ac97_synch <= 1'b0;
      // synthesis attribute init of ac97_synch is "0";

      bit_count <= 8'h00;
      // synthesis attribute init of bit_count is "0000";
      l_cmd_v <= 1'b0;
      // synthesis attribute init of l_cmd_v is "0";
      l_left_v <= 1'b0;
      // synthesis attribute init of l_left_v is "0";
      l_right_v <= 1'b0;
      // synthesis attribute init of l_right_v is "0";

      left_in_data <= 20'h00000;
      // synthesis attribute init of left_in_data is "00000";
      right_in_data <= 20'h00000;
      // synthesis attribute init of right_in_data is "00000";
   end

   always @(posedge ac97_bit_clock) begin
      // Generate the sync signal
      if (bit_count == 255)
        ac97_synch <= 1'b1;
      if (bit_count == 15)
        ac97_synch <= 1'b0;

      // Generate the ready signal
      if (bit_count == 128)
        ready <= 1'b1;
      if (bit_count == 2)
        ready <= 1'b0;

      // Latch user data at the end of each frame. This ensures that the
      // first frame after reset will be empty.
      if (bit_count == 255)
        begin
           l_cmd_addr <= {command_address, 12'h000};
           l_cmd_data <= {command_data, 4'h0};
           l_cmd_v <= command_valid;
           l_left_data <= left_data;
           l_left_v <= left_valid;
           l_right_data <= right_data;
           l_right_v <= right_valid;
        end

      if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
          4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
          4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
          4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
                  4'h4: ac97_sdata_out <= l_right_v; // Right data valid
          default: ac97_sdata_out <= 1'b0;
        endcase

      else if ((bit_count >= 16) && (bit_count <= 35))
```

```
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

    else if ((bit_count >= 56) && (bit_count <= 75))
      begin
        // Slot 3: Left channel
        ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
        l_left_data <= { l_left_data[18:0], l_left_data[19] };
      end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
        ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;

  end // always @ (posedge ac97_bit_clock)

  always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
  end

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

  input clock;
  input ready;
  output [7:0] command_address;
  output [15:0] command_data;
  output command_valid;
  input [4:0] volume;
  input source;

  reg [23:0] command;
  reg command_valid;

  reg old_ready;
  reg done;
  reg [3:0] state;

  initial begin
    command <= 4'h0;
    // synthesis attribute init of command is "0";
    command_valid <= 1'b0;
    // synthesis attribute init of command_valid is "0";
    done <= 1'b0;
    // synthesis attribute init of done is "0";
    old_ready <= 1'b0;
    // synthesis attribute init of old_ready is "0";
    state <= 16'h0000;
    // synthesis attribute init of state is "0000";
  end

  assign command_address = command[23:16];
  assign command_data = command[15:0];

  //wire [4:0] vol;
  //assign vol = 31-volume;

  always @(posedge clock) begin
    if (ready && (!old_ready))
      state <= state+1;
    case (state)
      4'h0: // Read ID
        begin
           command <= 24'h80_0000;
           command_valid <= 1'b1;
        end
      4'h1: // Read ID
        command <= 24'h80_0000;
      4'h3: // headphone volume
        command <= { 8'h04, 3'b000, 5'b00100, 3'b000, 5'b00100 };
      4'h5: // PCM volume
        command <= 24'h18_0808;
      4'h6: // Record source select
        command <= 24'h1A_0000; // microphone
      4'h7: // Record gain = max
                command <= 24'h1C_0F0F;
      4'h9: // set +20db mic gain
        command <= 24'h0E_8048;
      4'hA: // Set beep volume
        command <= 24'h0A_0000;
      4'hB: // PCM out bypass mix1
        command <= 24'h20_8000;
      default:
        command <= 24'h80_0000;
    endcase // case(state)

    old_ready <= ready;

  end // always @ (posedge clock)

endmodule // ac97commands

/*

module play_sounds( clk , reset , ready , from_ac97_data , to_ac97_data , snare_pad , tom_pad , bass_pad ,  hh_pad , led);

                input [7:0] snare_pad , tom_pad , bass_pad , hh_pad;
                input clk;              // 27mhz system clock
                input reset;                    // 1 to reset to initial state
                input ready;                    // 1 when AC97 data is available
                input [19:0] from_ac97_data;     // 8-bit PCM data from mic
                output [19:0] to_ac97_data;      // 8-bit PCM data to headphone


                output[7:0] led;
                reg [7:0] led;
                reg [19:0] to_ac97_data;
                // detect clock cycle when READY goes 0 -> 1
                // f(READY) = 48khz
                wire new_frame;
                reg old_ready;
                always @ (posedge clk) old_ready <= reset ? 0 : ready;
                assign new_frame = ready & ~old_ready;

                parameter SNARE_MAX = 27000;
                parameter TOM_MAX = 27000;
                parameter BASS_MAX = 27000;
                parameter HH_MAX = 27000;

                wire signed [15:0] snare_data , tom_data , base_data , hh_data;

                reg[14:0] snare_address , tom_address , bass_address , hh_address;
                snare_rom snr ( snare_address , clock_27mhz , snare_data );

                wire[19:0] tone_out;
                tone750hz t( clk , ready , tone_out);

                reg play_snare , play_tom , play_bass , play_hh;
                always @ (posedge clk) begin
                                led[7] <= ~snare_pad;

                                if ( reset == 1) begin
                                                play_snare <= 0;
                                                play_tom <= 0;
                                                play_bass <= 0;
                                                play_hh <= 0;
```

```
				led[3:0] <= 4'b1111;
			end

			if ( snare_pad > 5 ) begin
				snare_address <= 0;
				play_snare <= 1;
				led[0] <= 0;
			end
			if ( hh_pad > 4 ) begin
				hh_address <= 0;
				play_hh <= 1;
			end
			if ( tom_pad > 4 ) begin
				tom_address <= 0;
				play_tom <= 1;
			end
			if ( bass_pad > 4 ) begin
				bass_address <= 0;
				play_bass <= 1;
			end


			if ( snare_address > SNARE_MAX ) begin
				play_snare <= 0;
				led[0] <= 1;
			end
			if ( tom_address > TOM_MAX )
				play_tom <= 0;
			if ( bass_address > BASS_MAX )
				play_bass <= 0;
			if ( hh_address > HH_MAX )
				play_hh <= 0;



			if (new_frame) begin

				if ( play_snare == 1 )
					snare_address <= snare_address + 1;
				if ( play_bass == 1 )
					bass_address <= bass_address + 1;
				if ( play_tom == 1 )
					tom_address <= tom_address + 1;
				if ( play_hh == 1 )
					hh_address <= hh_address + 1;
				if ( reset == 0 )
					to_ac97_data <= snare_data;
				else
					to_ac97_data <= tone_out;

			end
		end

endmodule
```

```
///////////////////
//
// mario_game: the game itself!
//
////////////////////////////////////////////////////////////////////////////////


module timer(vclock , hcount, vcount , timer_pixel , start , finish, reset);
        parameter left=100;
        parameter top=50;
        input vclock;
        input [10:0] hcount;
        input [9:0] vcount;
        input start;
        input finish;
        input reset;

        output [24:0] timer_pixel;

        reg [3:0] seconds_one, seconds_ten, minutes_one, minutes_ten;
        reg [25:0] counter = 0;
        reg start_hold;
        reg finish_hold;
        reg toolong = 0;
        reg [24:0] timer_pixel;
        reg [8:0] address;
        wire[10:0] xcount , ycount;
        assign xcount = hcount - left+2;
        assign ycount = vcount - top;
        wire [24:0] d_out;
        timer_rom tr( address, vclock , d_out);
        reg [2:0] mult;
        wire t1,t2,t3,t4,t5,t6;
        assign t1 = (ycount >= 2);
        assign t2 = (ycount >= 4);
        assign t3 = (ycount >= 6);
        assign t4 = (ycount >= 8);
        assign t5 = (ycount >= 10);
        assign t6 = (ycount >= 12);
        always @(posedge vclock) begin
                case(ycount)
                        (t6) : mult <= 6;
                        (t5) : mult <= 5;
                        (t4) : mult <= 4;
                        (t3) : mult <= 3;
                        (t2) : mult <= 2;
                        (t1) : mult <= 1;
                        default : mult <= 0;
                endcase
        end

        always @(posedge vclock) begin
                if (reset == 1) begin
                        seconds_one <= 0;
                        seconds_ten <= 0;
                        minutes_one <= 0;
                        minutes_ten <= 0;
                        start_hold <= 0;
                        finish_hold <= 0;
                end
                else begin
                        if (xcount < 7)
                                address = minutes_ten*7 + xcount + 70 * ycount;
                        else if ( xcount < 14)
                                address = minutes_one*7 + xcount - 7 + 70 * ycount;
                        else if ( xcount < 25 )
                                address = seconds_ten*7 + xcount -18 + 70 * ycount;
                        else if ( xcount < 32 )
                                address = seconds_one *7 + xcount -25+ 70*ycount;

                        if ( xcount > 15 && xcount < 20 )
                                timer_pixel <= 25'b10000000000000000000000000;
                        else if (vcount >= top && hcount >= left && vcount < top + 7 && hcount < left+32)
                                timer_pixel <= d_out;
                        else
                                timer_pixel <= 25'b10000000000000000000000000;
                                if (start == 1)
                                        start_hold <= 1;
                                if (finish == 1 )
                                        finish_hold <= 1;
                                if ( counter == 65000000 && start && finish==0 ) begin
                                        counter <= 0;
                                        if ( seconds_one == 9 ) begin
                                                seconds_one <= 0;
                                                if ( seconds_ten == 5 ) begin
                                                        seconds_ten <= 0;
                                                        if (minutes_one == 9) begin
                                                                minutes_one <= 0;
                                                                if ( minutes_ten == 5 )
                                                                        toolong <= 1;
                                                                else
                                                                        minutes_ten <= minutes_ten + 1;

                                                        end
                                                        else
                                                                minutes_one <= minutes_one + 1;
                                                end
                                                else
                                                        seconds_ten <= seconds_ten + 1;
                                        end
                                        else    begin
                                                seconds_one <= seconds_one + 1;
                                        end
                                end
                                else counter <= counter + 1;
                        end
                end
endmodule

module mario_game_logic(vclock, button , jump_button, game_select, mode_select, reset, mario_x, mario_y, mario_count,
                                                mario_height, mario_width,intensity,hitreset,timer_start,timer_stop,
                                                setup,pixel_offset,new_box,index_curr, index_next,logic_block,
                                                hcount,vcount,hsync,vsync,
                                                blank,phsync,pvsync,pblank, l1, l2, l3,hex_display, play_jump_sound , play_music);
        input[2:0] button;
        input[2:0] jump_button;

        input game_select;
        input mode_select;
        input vclock;       // 65MHz clock
        input reset;            // 1 to initialize module
        input [10:0] hcount;      // horizontal index of current pixel (0..1023)
        input [9:0] vcount; // vertical index of current pixel (0..767)
        input hsync;            // XVGA horizontal sync signal (active low)
        input vsync;            // XVGA vertical sync signal (active low)
        input blank;            // XVGA blanking (1 means output black pixel)
        input [3:0] intensity;
        input setup;
        input [6:0] index_curr;
        input [6:0] index_next;

        output play_music;
        output [9:0] mario_y;
   output [10:0] mario_x;
        output [2:0] mario_count;
        output [16:0] mario_height;
        output [16:0] mario_width;
        output phsync,pvsync,pblank;
        output l1, l2, l3;
        output [7:0] pixel_offset;
        output new_box;
        output [6:0] logic_block;
        output [63:0] hex_display;
        output hitreset;
        output timer_start,timer_stop;
        output play_jump_sound;

        //parameter intensity = 10;
        reg [7:0] pixel_offset = 0;
        wire [6:0] nextpoff;
        reg new_box = 0;
```

```
	parameter pix_num = 32;
	wire [6:0] index;
	reg [6:0] logic_block=0;
	reg end_game = 0;

	//Ground Parameters---------------------
	parameter ground_y = 680;
	//------------------------------------

	//WALL parameters-----------------
	parameter wallheight = 32;
	parameter pipeheight = 138;
	reg[10:0] wall1_height = ground_y-wallheight;
	reg[10:0] wall2_height = ground_y-2*wallheight;
	reg[10:0] wall3_height = ground_y-3*wallheight;
	reg[10:0] wall4_height = ground_y-4*wallheight;
	reg[10:0] wall5_height = ground_y-5*wallheight;
	reg[10:0] pipe_height = ground_y-pipeheight;
	//------------------------------------

	//Frame Counters-------------------------
	reg [5:0] frame_count = 0;  //run frame count
	//-----------------------------------

	reg timer_start = 0;
	reg timer_stop = 0;

	//---MARIO PARAMETERS and REGISTERS---------
	parameter MARIO_WIDTH = 32;
parameter MARIO_HEIGHT = 64;
	parameter mario_ys = ground_y; //starting y value
	parameter mario_xs = 35;   //starting x value
	parameter mario_step = 6;
	parameter mjump = 12;

	reg[16:0] mario_width = MARIO_WIDTH;
	reg [2:0] mario_count = 0;
	reg [9:0] mario_y = mario_ys;
	reg [10:0] mario_x = mario_xs;
reg [16:0] mario_height = MARIO_HEIGHT;
	reg mario_centered = 0;
	reg [4:0] mario_xvel = 0;
	reg [4:0] mario_yvel = 0;
	reg trapped = 0, timeout = 0;
//------------------------------------

	reg hitreset=0;

	reg phsync, pvsync, pblank;
reg temp1, temp2, temp3;

	wire new_frame;
assign new_frame = pvsync & ~vsync; //new_frame is available at negative edge

	reg jump_dir=1;  //jump_dir: 0 for up, 1 for down
	reg [20:0] max_jump = 0;

	//delay phsync, pvsync and pblank by vclock
always @ (posedge vclock) begin
			phsync <=  hsync;
			pvsync <= vsync;
			pblank <=  blank;
	end

parameter S_height = 768;   //screen height in pixels starting at pixel 0
parameter S_width = 1024;   //screen width in pixels starting at pixel 0

	wire [6:0] poff;
	assign poff = pixel_offset;
	reg play_jump_sound=0;
	assign hex_display = {56'b00000000000000000000000000000000000000000000000000000000, poff};
	assign l1 = ~(setup == 1);
	assign l2 = 1; assign l3 = 1;
	assign nextpoff = pixel_offset + mario_xvel;
	assign index = (nextpoff >= pix_num && trapped == 0) ? index_next : index_curr;
	reg  update_count = 0;
reg [10:0] run_count = 0;
reg [3:0] acc_count = 0;
always @ (posedge vclock) begin
        if (hitreset == 1) hitreset <= 0;

        if (new_box == 1) new_box <= 0;

        if (setup == 1) begin
                if (new_frame)
                        hitreset <= 1;

                if (reset == 1) begin
                                mario_x <= mario_xs;
                                mario_y <= mario_ys;
                                frame_count <= 0;
                                mario_count <= 0;
                                mario_centered <= 0;
                                max_jump <= 0;
                                jump_dir<=1;
                                pixel_offset <= 0;
                                logic_block <= 0;
                                mario_xvel <= 0;
                                mario_yvel<=0;
                                trapped <= 0;
                                timeout <= 0;
                                timer_start <= 0;
                                timer_stop <= 0;
                                play_jump_sound <= 0;
                                end_game <= 0;
                end

                        if (timeout == 0 && new_frame == 1 && (mario_yvel == 0 || mario_xvel < 12 )) begin
                        if (button > 0 && run_count == 0) begin
                                mario_xvel <= 6;
                                run_count <= 0;
                        end
                        else if ( button > 0 && run_count > 0 ) begin
                                if ( mario_xvel < 12 )
                                        mario_xvel <= mario_xvel + 6;
                                else
                                        mario_xvel <= 18;
                                        run_count <= 0;
                        end
                        else if ( button == 0 && run_count < 60) begin
                                run_count <= run_count + 1;
                        end
                        else if ( button == 0 && run_count == 60 ) begin
                                if ( mario_xvel > 6 )
                                        mario_xvel <= mario_xvel - 6;
                                else
                                        mario_xvel <= 0;
                                run_count <= 0;

                        end
                end


                //check for jump and assign max height to jump to
                if (jump_button > 0 && mario_yvel == 0) begin
                        play_jump_sound <= 1;
       jump_dir <= 0;
                        max_jump <= mario_y - jump_button*30;
                        mario_yvel <= mjump;
                end
                else
                        play_jump_sound <= 0;


                if (new_frame) begin
                        if (frame_count == 1) begin
                          frame_count <= 0;

                          if ( timer_stop == 1) begin
                                        mario_count <= 4;
                                        update_count <=0;
                                        if (mario_y+mario_height+mario_yvel >= ground_y && jump_dir == 1) begin
                        mario_y <= ground_y-mario_height; mario_yvel <= mjump;
                        timeout<= 0; jump_dir <= 0;

                end
                else if (mario_y - mario_yvel < (ground_y - 200) && jump_dir == 0) begin
                        mario_y <= ground_y - 200;
                        jump_dir <= 1;
                end
                else begin
                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel : mario_y+mario_yvel;
                end

                if (pixel_offset + 1 == pix_num) begin
                                new_box <= 1; pixel_offset <= 0;
                end
                else if (end_game == 0) begin
                        if (pixel_offset + 1 != pix_num - 10 || index_curr == 3)
                                pixel_offset <= pixel_offset + 1;
                        else begin
                                pixel_offset <= pix_num - 10;
                                mario_xvel <= 0; end_game <= 1;
                        end
                end
end
else begin
        if (mario_xvel == 0 && mario_yvel == 0 ) begin
                mario_count <=0;
                update_count <= 0;
        end
        else if (mario_yvel != 0) begin
                mario_count <=4;
                update_count <= 0;
        end

          else    if (mario_xvel != 0 && mario_yvel==0) begin
          update_count <= update_count + 1;
          if (mario_count == 5 && button == 0) begin
                mario_xvel <= 0;
                        mario_count <= 0;
                end

                else if (update_count == 0) begin
                        case (mario_count)
                          0: mario_count <= 1;
                          1: mario_count <= 2;
                          2: mario_count <= 3;
                          3: mario_count <= 5;
                          5: mario_count <= 0;
                          4: mario_count <= 1;
                          //2: mario_count <= 0;
                          //4: mario_count <= 1;
                        endcase
                end

          end



        //if mario is before the center of the screen
        if (mario_centered == 0) begin
                if (mario_x + mario_width + mario_xvel >= S_width/2) begin
                        mario_centered <= 1;
                        mario_x <= S_width/2-mario_width+7;
                        logic_block <= 16;
                end
                else        mario_x <= mario_x + mario_xvel;

                if (mario_y+mario_height+mario_yvel >= ground_y && jump_dir == 1) begin
                                mario_y <= ground_y-mario_height; mario_yvel <= 0;

                end
                else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                mario_y <= max_jump;
                                jump_dir <= 1;
                end
                else begin
```

```
                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel : mario_y+mario_yvel;
                                end
                        end

                        //mario is centered
                        else begin
                                //if mario will enter a new block


                        //determine where to put mario based on index value
                          //this is the meat of the logic
                                if ((timeout == 1) &&
                                 ((pixel_offset < pix_num - 10)||(index == 2))) begin
                                        if (pixel_offset + 1 == pix_num) begin
                                                new_box <= 1; pixel_offset <= 0;
                                        end
                                        else begin
                                                if (pixel_offset + 1 != pix_num - 10 || index == 2)
                                                        pixel_offset <= pixel_offset + 1;
                                                else begin
                                                        pixel_offset <= pix_num - 10; mario_yvel <= mjump;
                                                        mario_xvel <= 0; mario_y <= 0;
                                                end
                                        end
                                end
                                else begin
                                case (index)
                                        0: begin   //flat ground
                                          if (mario_y < ground_y && mario_y > max_jump)
                                                        mario_yvel <= mjump;
                                                if (nextpoff >= pix_num) begin
                                                        pixel_offset <= nextpoff - pix_num;
                                                        new_box <= 1;
                                                end
                                                else  pixel_offset <= nextpoff;

                                                if (mario_y+mario_height+mario_yvel >= ground_y && jump_dir == 1) begin
                                                        mario_y <= ground_y-mario_height; mario_yvel <= 0;
                                                        timeout<= 0;

                                                end
                                                else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                        mario_y <= max_jump;
                                                        jump_dir <= 1;
                                                end
                                                else begin
                                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
                                                end
                                  end
                                        1: begin //one brick
/*just entering box */                  //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                                if (nextpoff >= pix_num) begin
                                                        if (jump_dir == 1 && mario_yvel != 0) begin
                                                                if (mario_y+mario_height+mario_yvel > wall1_height) begin
                                                                        pixel_offset<=pix_num-1;
                                                                        mario_xvel <= 0;
                                                                        mario_y <= mario_y + mario_yvel;
                                                                        timeout <= 0;
                                                                end
                                                                else begin
                                                                        new_box <= 1;
                                                                        pixel_offset<=nextpoff-pix_num;
                                                                        mario_y <= mario_y + mario_yvel;
                                                                end
                                                        end
                                                        else begin
                                                    if (mario_y+mario_height-mario_yvel > wall1_height) begin
                                                                pixel_offset <= pix_num-1;
                                                                if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                         else begin
                                                                new_box <= 1;
                                                                pixel_offset<=nextpoff-pix_num;
                                                                if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
```

```
						end
					end
					else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
						mario_y <= max_jump;
						pixel_offset <= nextpoff;
						jump_dir <= 1;
					end
					else if (mario_y+mario_height+mario_yvel > wall1_height && jump_dir == 1) begin
						mario_y <= wall1_height-mario_height;
						mario_yvel <= 0;
						mario_xvel <= 0;
						pixel_offset <= nextpoff;
						timeout <= 0;

					end
					else begin
						mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
						pixel_offset <= nextpoff;
					end
				end
				2: begin	//PIT
					if (mario_y < ground_y && mario_y > max_jump)
						mario_yvel <= mjump;
					if (nextpoff >= pix_num) begin
							if (trapped == 0) begin
								pixel_offset <= nextpoff - pix_num;
								new_box <= 1;
							end
							else pixel_offset <= pix_num - 1;
					end
					else  pixel_offset <= nextpoff;

					if (mario_y +mario_height > ground_y)
						trapped <= 1;
					if (mario_y+mario_yvel >= S_height && jump_dir == 1) begin
						mario_y <= 770; mario_yvel <= mjump;	//new_box <= 1;
						mario_xvel <= 0;
						trapped <= 0;
						timeout <= 1;

					end
					else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
						mario_y <= max_jump;
						jump_dir <= 1;
					end
					else begin
						mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
					end
				end
				3: begin	//FLAGPOLE
/*just entering box */			//if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
					if (nextpoff >= pix_num) begin
						if (jump_dir == 1 && mario_yvel != 0) begin
							if (mario_y+mario_height+mario_yvel > wall1_height) begin
								pixel_offset<=pix_num-1;
								mario_xvel <= 0;
								mario_y <= mario_y + mario_yvel;
								timeout <= 0;

							end
							else begin
								new_box <= 1;
								pixel_offset<=nextpoff-pix_num;
								mario_y <= mario_y + mario_yvel;
								if (timer_start == 1) timer_stop <= 1;
									timer_start <= 1;
							end
						end
						else begin
					    if (mario_y+mario_height-mario_yvel > wall1_height) begin
							pixel_offset <= pix_num-1;
							if (mario_yvel != 0)
								mario_y <= mario_y - mario_yvel;
						  end
```

```
                                                                                else begin
                                                                                        new_box <= 1;
                                                                                        pixel_offset<=nextpoff-pix_num;
                                                                                        if (mario_yvel != 0)
                                                                                                mario_y <= mario_y - mario_yvel;
                                                                                        if (timer_start == 1)
                                                                                                timer_stop <= 1;
                                                                                        timer_start <= 1;
                                                                                end
                                                                              end
                                                                      end
                                                                      else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                                              mario_y <= max_jump;
                                                                              pixel_offset <= nextpoff;
                                                                              jump_dir <= 1;
                                                                      end
                                                                      else if (mario_y+mario_height+mario_yvel > wall1_height && jump_dir == 1) begin
                                                                              mario_y <= wall1_height-mario_height;
                                                                              mario_yvel <= 0;
                                                                              mario_xvel <= 0;
                                                                              pixel_offset <= nextpoff;
                                                                              timeout <= 0;

                                                                      end
                                                                      else begin
                                                                              mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;

                                                                              pixel_offset <= nextpoff;
                                                                      end
                                                       end
                                                             4: begin //2 bricks
/*just entering box */                        //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                                                      if (nextpoff >= pix_num) begin
                                                                              if (jump_dir == 1 && mario_yvel != 0) begin
                                                                                      if (mario_y+mario_height+mario_yvel > wall2_height) begin
                                                                                              pixel_offset<=pix_num-1;
                                                                                              mario_xvel <= 0;
                                                                                              mario_y <= mario_y + mario_yvel;
                                                                                              timeout <= 0;
                                                                                      end
                                                                                      else begin
                                                                                              new_box <= 1;
                                                                                              pixel_offset<=nextpoff-pix_num;
                                                                                              mario_y <= mario_y + mario_yvel;
                                                                                      end
                                                                              end
                                                                              else begin
                                                                          if (mario_y+mario_height-mario_yvel > wall2_height) begin
                                                                                      pixel_offset <= pix_num-1;
                                                                                      if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                                                end
                                                                                else begin
                                                                                      new_box <= 1;
                                                                                      pixel_offset<=nextpoff-pix_num;
                                                                                      if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                                                end
                                                                              end
                                                                      end
                                                                      else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                                              mario_y <= max_jump;
                                                                              pixel_offset <= nextpoff;
                                                                              jump_dir <= 1;
                                                                      end
                                                                      else if (mario_y+mario_height+mario_yvel > wall2_height && jump_dir == 1) begin
                                                                              mario_y <= wall2_height-mario_height;
                                                                              mario_yvel <= 0;
                                                                              mario_xvel <= 0;
                                                                              pixel_offset <= nextpoff;
                                                                              timeout <= 0;

                                                                      end
                                                                      else begin
                                                                              mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;

                                                                              pixel_offset <= nextpoff;
                                                                      end
```

```
                                        end
                                            5: begin //3 brick
/*just entering box */                  //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                                    if (nextpoff >= pix_num) begin
                                                        if (jump_dir == 1 && mario_yvel != 0) begin
                                                            if (mario_y+mario_height+mario_yvel > wall3_height) begin
                                                                pixel_offset<=pix_num-1;
                                                                mario_xvel <= 0;
                                                                mario_y <= mario_y + mario_yvel;
                                                                timeout <= 0;
                                                            end
                                                            else begin
                                                                new_box <= 1;
                                                                pixel_offset<=nextpoff-pix_num;
                                                                mario_y <= mario_y + mario_yvel;
                                                            end
                                                        end
                                                        else begin
                                                      if (mario_y+mario_height-mario_yvel > wall3_height) begin
                                                            pixel_offset <= pix_num-1;
                                                            if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                         else begin
                                                            new_box <= 1;
                                                            pixel_offset<=nextpoff-pix_num;
                                                            if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                        end
                                                    end
                                                    else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                        mario_y <= max_jump;
                                                        pixel_offset <= nextpoff;
                                                        jump_dir <= 1;
                                                    end
                                                    else if (mario_y+mario_height+mario_yvel > wall3_height && jump_dir == 1) begin
                                                        mario_y <= wall3_height-mario_height;
                                                        mario_yvel <= 0;
                                                        mario_xvel <= 0;
                                                        pixel_offset <= nextpoff;
                                                        timeout <= 0;

                                                    end
                                                    else begin
                                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
                                                        pixel_offset <= nextpoff;
                                                    end
                                        end
                                            6: begin //4 bricks
/*just entering box */                  //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                                    if (nextpoff >= pix_num) begin
                                                        if (jump_dir == 1 && mario_yvel != 0) begin
                                                            if (mario_y+mario_height+mario_yvel > wall4_height) begin
                                                                pixel_offset<=pix_num-1;
                                                                mario_xvel <= 0;
                                                                mario_y <= mario_y + mario_yvel;
                                                                timeout <= 0;
                                                            end
                                                            else begin
                                                                new_box <= 1;
                                                                pixel_offset<=nextpoff-pix_num;
                                                                mario_y <= mario_y + mario_yvel;
                                                            end
                                                        end
                                                        else begin
                                                      if (mario_y+mario_height-mario_yvel > wall4_height) begin
                                                            pixel_offset <= pix_num-1;
                                                            if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                         else begin
                                                            new_box <= 1;
                                                            pixel_offset<=nextpoff-pix_num;
                                                            if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                        end
                                                    end
                                                    else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                        mario_y <= max_jump;
                                                        pixel_offset <= nextpoff;
                                                        jump_dir <= 1;
                                                    end
                                                    else if (mario_y+mario_height+mario_yvel > wall4_height && jump_dir == 1) begin
                                                        mario_y <= wall4_height-mario_height;
                                                        mario_yvel <= 0;
                                                        mario_xvel <= 0;
                                                        pixel_offset <= nextpoff;
                                                        timeout <= 0;

                                                    end
                                                    else begin
                                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
                                                        pixel_offset <= nextpoff;
                                                    end
                                        end
                                            7: begin //5 bricks
/*just entering box */                //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                                    if (nextpoff >= pix_num) begin
                                                        if (jump_dir == 1 && mario_yvel != 0) begin

                                                            if (mario_y+mario_height+mario_yvel > wall5_height) begin
                                                                pixel_offset<=pix_num-1;
                                                                mario_xvel <= 0;
                                                                mario_y <= mario_y + mario_yvel;
                                                                timeout <= 0;
                                                            end
                                                            else begin
                                                              new_box <= 1;
                                                                pixel_offset<=nextpoff-pix_num;
                                                                mario_y <= mario_y + mario_yvel;
                                                            end
                                                        end
                                                        else begin
                                                      if (mario_y+mario_height-mario_yvel > wall5_height) begin
                                                            pixel_offset <= pix_num-1;
                                                            if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                         else begin
                                                            new_box <= 1;
                                                            pixel_offset<=nextpoff-pix_num;
                                                            if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                                         end
                                                        end
                                                    end
                                                    else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                        mario_y <= max_jump;
                                                        pixel_offset <= nextpoff;
                                                        jump_dir <= 1;
                                                    end
                                                    else if (mario_y+mario_height+mario_yvel > wall5_height && jump_dir == 1) begin
                                                        mario_y <= wall5_height-mario_height;
                                                        mario_yvel <= 0;
                                                        mario_xvel <= 0;
                                                        pixel_offset <= nextpoff;
                                                        timeout <= 0;

                                                    end
                                                    else begin
                                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
                                                        pixel_offset <= nextpoff;
                                                    end
                                        end
                                            8: begin //pipe left
/*just entering box */                //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                                    if (nextpoff >= pix_num) begin
                                                        if (jump_dir == 1 && mario_yvel != 0) begin
                                                            //new_box <= 1;
                                                            if (mario_y+mario_height+mario_yvel > pipe_height) begin
                                                                pixel_offset<=pix_num-1;
                                                                mario_xvel <= 0;
                                                                mario_y <= mario_y + mario_yvel;
```

```
                                                        timeout <= 0;
                                                end
                                                else begin
                                                        new_box <= 1;
                                                        pixel_offset<=nextpoff-pix_num;
                                                        mario_y <= mario_y + mario_yvel;
                                                end
                                        end
                                        else begin
                                    if (mario_y+mario_height-mario_yvel-6 > pipe_height) begin
                                                pixel_offset <= pix_num-1;
                                                if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                          end
                                          else begin
                                                new_box <= 1;
                                                pixel_offset<=nextpoff-pix_num;
                                                if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                          end
                                        end
                                end
                                else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                        mario_y <= max_jump;
                                        pixel_offset <= nextpoff;
                                        jump_dir <= 1;
                                end
                                else if (mario_y+mario_height+mario_yvel > pipe_height && jump_dir == 1) begin
                                        mario_y <= pipe_height-mario_height;
                                        mario_yvel <= 0;
                                        mario_xvel <= 0;
                                        pixel_offset <= nextpoff;
                                        timeout <= 0;

                                end
                                else begin
                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;

                                        pixel_offset <= nextpoff;
                                end
                        end
                            9: begin //pipe right
/*just entering box */  //if (mario_y < wall1_height && mario_y > max_jump) mario_yvel <= mjump;
                                if (nextpoff >= pix_num) begin
                                        if (jump_dir == 1 && mario_yvel != 0) begin
                                                new_box <= 1;
                                                if (mario_y+mario_height+mario_yvel > pipe_height) begin
                                                        pixel_offset<=nextpoff-pix_num;
                                                        mario_xvel <= 0;
                                                        mario_y <= pipe_height - mario_height;
                                                        timeout <= 0;
                                                end
                                                else begin
                                                        pixel_offset<=nextpoff-pix_num;
                                                        mario_y <= mario_y + mario_yvel;
                                                end
                                        end
                                        else begin
                                    if (mario_y+mario_height-mario_yvel > pipe_height) begin
                                                pixel_offset <= pix_num-1;
                                                if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                          end
                                          else begin
                                                new_box <= 1;
                                                pixel_offset<=nextpoff-pix_num;
                                                if (mario_yvel != 0) mario_y <= mario_y - mario_yvel;
                                          end
                                        end
                                end
                                else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                        mario_y <= max_jump;
                                        pixel_offset <= nextpoff;
                                        jump_dir <= 1;
                                end
                                else if (mario_y+mario_height+mario_yvel > pipe_height && jump_dir == 1) begin
                                        mario_y <= pipe_height-mario_height;
                                        mario_yvel <= 0;
                                        mario_xvel <= 0;
```

```
                                                                                                        pixel_offset <= nextpoff;
                                                                                                        timeout <= 0;

                                                                                            end
                                                                                            else begin
                                                                                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;
                                                                                                        pixel_offset <= nextpoff;
                                                                                            end
                                                                          end
                                                                                 default: begin
                                                                                            if (nextpoff >= pix_num) begin
                                                                                                                    pixel_offset <= nextpoff - pix_num;
                                                                                                                    new_box <= 1;
                                                                                            end
                                                                                            else  pixel_offset <= nextpoff;
                                                                                            if (mario_y < ground_y && mario_y > max_jump)
                                                                                                        mario_yvel <= mjump;
                                                                                            if (mario_y+mario_height+mario_yvel >= ground_y && jump_dir == 1) begin
                                                                                                        mario_y <= ground_y-mario_height;
                                                                                                        mario_yvel <= 0; mario_xvel <= 0; timeout<=0;

                                                                                            end
                                                                                            else if (mario_y - mario_yvel < max_jump && jump_dir == 0) begin
                                                                                                        mario_y <= max_jump;
                                                                                                        jump_dir <= 1;
                                                                                            end
                                                                                            else begin
                                                                                                        mario_y <= (jump_dir == 0) ? mario_y - mario_yvel :
mario_y+mario_yvel;

                                                                                            end
                                                                          end
                                                                        endcase
                                                                        end
                                                            end
                                                   end
                                                  end
                                                  else frame_count <= frame_count + 1;
                                        end
                              end
                    end
endmodule



module mario_game_vid (vclock,reset,
            hcount,vcount,hsync,vsync,blank,
            mario_pixel,mario_x,mario_y, mario_count, mario_height, mario_width);
   input vclock;      // 65MHz clock
   input reset;              // 1 to initialize module
   input [10:0] hcount;      // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   input hsync;              // XVGA horizontal sync signal (active low)
   input vsync;              // XVGA vertical sync signal (active low)
   input blank;              // XVGA blanking (1 means output black pixel)
   input [2:0] mario_count;
   input [9:0] mario_y;
   input [10:0] mario_x;
   input [16:0] mario_height;
   input [16:0] mario_width;

   output [24:0] mario_pixel;      // mario pixel

   reg[10:0] mario_add;

   wire [24:0] mario0_pix;
   mario0 mymario0(mario_add, vclock, mario0_pix);

   wire [24:0] mario1_pix;
   mario1 mymario1(mario_add, vclock, mario1_pix);

   wire [24:0] mario2_pix;
   mario2 mymario2(mario_add, vclock, mario2_pix);

   wire [24:0] mario3_pix;
   mario3 mymario3(mario_add, vclock, mario3_pix);
   wire [24:0] mario4_pix;
   mario4 mymario4(mario_add, vclock, mario4_pix);

   reg [24:0] mario_pixel;

   always @ (posedge vclock) begin

                    if (vcount >= mario_y && vcount < mario_y+64 && hcount >= mario_x && hcount < mario_x + 32) begin

                                                            mario_add = (vcount - mario_y) * 32 + hcount +1- mario_x;

                                                            case (mario_count)
                                                            0: mario_pixel <=mario0_pix;
                                                            1: mario_pixel <=mario1_pix;
                                                            5: mario_pixel <=mario2_pix;
                                                            2: mario_pixel <=mario2_pix;
                                                            3: mario_pixel <=mario3_pix;
                                                            4: mario_pixel <=mario4_pix;
                                                            endcase

                                                  end
                     else mario_pixel <= 25'b1111111111111111111111111;
   end
endmodule


module line_vid (vclock,reset,
            hcount,vcount,hsync,vsync,blank,
            line_pixel);
   input vclock;              // 65MHz clock
   input reset;               // 1 to initialize module
   input [10:0] hcount;       // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   input hsync;               // XVGA horizontal sync signal (active low)
   input vsync;               // XVGA vertical sync signal (active low)
   input blank;               // XVGA blanking (1 means output black pixel)


   output [24:0] line_pixel;       // wall pixel

   reg [24:0] line_pixel;

   always @ (posedge vclock) begin
                     if (hcount < 32 || hcount > 992) line_pixel <= 25'b0000000000000000000000000;
                     else line_pixel <= 25'b1000000000000000000000000;
   end
endmodule

module map_vid (vclock,reset, new_box,
            hcount, vcount, map_pixel);
   input vclock;              // 65MHz clock
   input reset;               // 1 to initialize module
   input [10:0] hcount;       // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   input new_box;

   output [24:0] map_pixel;       // wall pixel
   reg [24:0] map_pixel;

   reg [11:0] mario_mapx = 650; //marios start position in map


   always @ (posedge vclock) begin
                     if (reset == 1) mario_mapx <= 650;
                     if (new_box == 1) mario_mapx <= mario_mapx + 1;

                     if (vcount > 30 && vcount <= 70 && hcount >= 649 && hcount < 908) begin
                              if (hcount == 650 || hcount == 906) map_pixel <= 25'b0111010110100010000010100;
                              else if (hcount == mario_mapx) map_pixel <= 25'b0000000000000000000000000;
                              else map_pixel <= 25'b0101101100100100110001000;
                     end
                     else map_pixel <= 25'b1000000000000000000000000;
   end
endmodule
```

```
module obj_vid (vclock,reset,
                hcount,vcount,new_box,
                pixel_offset,bg_pixel,setup,logic_block,index_curr,index_next);
   input vclock;              // 65MHz clock
   input reset;               // 1 to initialize module
   input [10:0] hcount;       // horizontal index of current pixel (0..1023)
   input [9:0] vcount; // vertical index of current pixel (0..767)
   input [7:0] pixel_offset;
   input new_box;
   input [6:0] logic_block;

   output [24:0] bg_pixel;       // pixel out
   output setup;
   output [6:0] index_curr;
   output [6:0] index_next;

   reg[5:0] queue [32:0];

   wire[3:0] data_out;
   reg [8:0] address=0;

   reg [24:0] bg_pixel;
   reg [6:0] block_num;
   wire [6:0] index;

   reg [7:0] count; //count for for loop

   reg setup = 0;
   reg newb = 0;

   reg [10:0] block1add;
   wire[23:0] block1data;
   reg [9:0] block2add;
   wire[24:0] block2data;
   reg [13:0] flagadd;
   wire[24:0] flagdata;
   reg [12:0] piperightadd;
   wire[24:0] piperightdata;
   reg [12:0] pipeleftadd;
   wire[24:0] pipeleftdata;

   bg mymem(address,vclock,data_out);
   block1mem mymem2(block1add,vclock, block1data);
   block2mem mymem3(block2add,vclock,block2data);
   flagmem mymem4(flagadd,vclock,flagdata);
   piperightmem mymem5(piperightadd,vclock,piperightdata);
   pipeleftmem mymem6(pipeleftadd,vclock,pipeleftdata);

   parameter num_blocks = 32;
   parameter pix_num = 32;
   assign index_curr = (setup == 1) ? queue[logic_block] : 0;
   assign index_next = (setup == 1) ? (logic_block+1==33) ? queue[0] : queue[logic_block+1] : 0;
   assign index = (setup == 1) ? queue[block_num] : 0;

   //Ground Parameters---------------------
   parameter ground_y = 680;
   //------------------------------------

   //WALL parameters-----------------
   parameter block1_height = 32;
   parameter flagpoleheight = 359;
   parameter pipeheight = 138;
   //------------------------------------

   wire [11:0] hloc1,hloc2,hloc3,hloc4,hloc5,hloc6,hloc7,hloc8,hloc9,hloc10,hloc11,hloc12;
   wire [11:0] hloc13,hloc14,hloc15,hloc16,hloc17,hloc18,hloc19,hloc20,hloc21,hloc22,hloc23,hloc24;
   wire [11:0] hloc25,hloc26,hloc27,hloc28,hloc29,hloc30,hloc31,hloc32, hloc33;

   assign hloc1 = pix_num - pixel_offset;
   assign hloc2 = 2*pix_num - pixel_offset;
   assign hloc3 = 3*pix_num - pixel_offset;
   assign hloc4 = 4*pix_num - pixel_offset;
   assign hloc5 = 5*pix_num - pixel_offset;
   assign hloc6 = 6*pix_num - pixel_offset;
   assign hloc7 = 7*pix_num - pixel_offset;
   assign hloc8 = 8*pix_num - pixel_offset;
   assign hloc9 = 9*pix_num - pixel_offset;
   assign hloc10 = 10*pix_num - pixel_offset;
   assign hloc11 = 11*pix_num - pixel_offset;
   assign hloc12 = 12*pix_num - pixel_offset;
   assign hloc13 = 13*pix_num - pixel_offset;
   assign hloc14 = 14*pix_num - pixel_offset;
   assign hloc15 = 15*pix_num - pixel_offset;
   assign hloc16 = 16*pix_num - pixel_offset;
   assign hloc17 = 17*pix_num - pixel_offset;
   assign hloc18 = 18*pix_num - pixel_offset;
   assign hloc19 = 19*pix_num - pixel_offset;
   assign hloc20 = 20*pix_num - pixel_offset;
   assign hloc21 = 21*pix_num - pixel_offset;
   assign hloc22 = 22*pix_num - pixel_offset;
   assign hloc23 = 23*pix_num - pixel_offset;
   assign hloc24 = 24*pix_num - pixel_offset;
   assign hloc25 = 25*pix_num - pixel_offset;
   assign hloc26 = 26*pix_num - pixel_offset;
   assign hloc27 = 27*pix_num - pixel_offset;
   assign hloc28 = 28*pix_num - pixel_offset;
   assign hloc29 = 29*pix_num - pixel_offset;
   assign hloc30 = 30*pix_num - pixel_offset;
   assign hloc31 = 31*pix_num - pixel_offset;
   assign hloc32 = 32*pix_num - pixel_offset;
   assign hloc33 = 33*pix_num - pixel_offset;

   always @ (posedge vclock) begin
           if (reset == 1) begin
                   setup <= 0; address <= 0;
           end
           else if (setup == 0) begin
                   queue[address]<= data_out;
                   address <= address + 1;
                   if (address >= num_blocks) setup <= 1;
           end
           else if (new_box == 1) newb <= 1;
           else if (newb == 1 && vcount > 775) begin
                   address <= address + 1;
                   for (count = 0; count <= num_blocks; count = count + 1) begin
                           if (count == num_blocks) queue[count] <= data_out;
                           else queue[count] <= queue[count+1];
                   end

                   newb <=0;
           end
           else begin
               case (hcount)
                0: block_num <= 0;
                   (hloc1): block_num <= block_num + 1;
                   (hloc2): block_num <= block_num + 1;
                   (hloc3): block_num <= block_num + 1;
                   (hloc4): block_num <= block_num + 1;
                   (hloc5): block_num <= block_num + 1;
                   (hloc6): block_num <= block_num + 1;
                   (hloc7): block_num <= block_num + 1;
                   (hloc8): block_num <= block_num + 1;
                   (hloc9): block_num <= block_num + 1;
                   (hloc10): block_num <= block_num + 1;
                   (hloc11): block_num <= block_num + 1;
                   (hloc12): block_num <= block_num + 1;
                   (hloc13): block_num <= block_num + 1;
                   (hloc14): block_num <= block_num + 1;
                   (hloc15): block_num <= block_num + 1;
                   (hloc16): block_num <= block_num + 1;
                   (hloc17): block_num <= block_num + 1;
                   (hloc18): block_num <= block_num + 1;
                   (hloc19): block_num <= block_num + 1;
                   (hloc20): block_num <= block_num + 1;
                   (hloc21): block_num <= block_num + 1;
                   (hloc22): block_num <= block_num + 1;
                   (hloc23): block_num <= block_num + 1;
                   (hloc24): block_num <= block_num + 1;
                   (hloc25): block_num <= block_num + 1;
                   (hloc26): block_num <= block_num + 1;
                   (hloc27): block_num <= block_num + 1;
                   (hloc28): block_num <= block_num + 1;
```

```
                    (hloc29): block_num <= block_num + 1;
                    (hloc30): block_num <= block_num + 1;
                    (hloc31): block_num <= block_num + 1;
                    (hloc32): block_num <= block_num + 1;
                    (hloc33): block_num <= 0;
                    default: block_num <= block_num;
               endcase

                    if ( vcount >= ground_y && index != 2) begin

                              block2add = ((vcount - ground_y)%32) * 32 + hcount - (block_num*32-pixel_offset);
                              bg_pixel[24] <= 0;
                              bg_pixel[23:0] <= block2data;
               end
               else begin
                case (index)
                    0: begin  //REGULAR GROUND
                              bg_pixel <= 25'b1000000000000000000000000;
end
                    1: begin //1 BLOCK
                              if (vcount >= (ground_y-block1_height) && vcount < ground_y) begin
                                        block1add = (vcount - (ground_y-block1_height)) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= block1data;
                              end
              end
                    2: begin //PIT
                              bg_pixel <= 25'b1000000000000000000000000;
                     end
     3: begin //FLAGPOLE
                              if (vcount >= ground_y - flagpoleheight && vcount < ground_y) begin
                                        flagadd = (vcount - (ground_y-flagpoleheight)) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0] <= flagdata[23:0];
                              end
                    end
                    4: begin //WALL OF 2 BLOCKS
                              if (vcount >= (ground_y-2*block1_height) && vcount < ground_y) begin
                                        block1add = ((vcount - (ground_y-2*block1_height))%32) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= block1data;
                              end
          end
                    5: begin //WALL OF 3 BLOCKS
                              if (vcount >= (ground_y-3*block1_height) && vcount < ground_y) begin
                                        block1add = ((vcount - (ground_y-3*block1_height))%32) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= block1data;
                              end
          end
                    6: begin //WALL OF 4 BLOCKS
                              if (vcount >= (ground_y-4*block1_height) && vcount < ground_y) begin
                                        block1add = ((vcount - (ground_y-4*block1_height))%32) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= block1data;
                              end
          end
                    7: begin //WALL OF 5 BLOCKS
                              if (vcount >= (ground_y-5*block1_height) && vcount < ground_y) begin
                                        block1add = ((vcount - (ground_y-5*block1_height))%32) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= block1data;
                              end
          end
                    8: begin //PIPE LEFT
                              if (vcount >= (ground_y-pipeheight) && vcount < ground_y) begin
                                        pipeleftadd = (vcount - (ground_y-pipeheight)) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= pipeleftdata;
                              end
          end
                    9: begin //PIPE RIGHT
                              if (vcount >= (ground_y-pipeheight) && vcount < ground_y) begin
                                        piperightadd = (vcount - (ground_y-pipeheight)) * 32 + hcount - (block_num*32-pixel_offset);
                                        bg_pixel[24] <= 0;
                                        bg_pixel[23:0]<= piperightdata;
```

```
                                                    end
                        end
                                            default: begin
                                                    if (vcount < ground_y)
                                                            bg_pixel <= 25'b0000000000000000011111111;
                                                    else
                                                            bg_pixel <= 25'b0000000001111111100000000;
            end
                            endcase
                                    end
                end
  end
endmodule

module back_vid (vclock,reset, hcount, vcount,new_box, pixel_offset, back_pixel);
  input vclock;              // 65MHz clock
  input reset;               // 1 to initialize module
  input [10:0] hcount;       // horizontal index of current pixel (0..1023)
  input [9:0] vcount; // vertical index of current pixel (0..767)
  input [7:0] pixel_offset;
  input new_box;

  output [24:0] back_pixel;       // pixel out

  reg[5:0] queue [32:0];

  wire[4:0] data_out;
  reg [8:0] address=0;

  reg [8:0] count = 0;

  reg [24:0] back_pixel;
  reg [6:0] block_num;

  reg setup = 0;
  reg newb = 0;

  wire [6:0] index;

  parameter num_blocks = 32;
  parameter pix_num = 32;


  //Ground Parameters---------------------
  parameter ground_y = 680;
  //------------------------------------

  //CLOUD parameters-----------------
  parameter cloudheight = 350;
  //------------------------------------

  //BUSH AND HILL-------------------------
  parameter bushheight = 34;
  parameter hillheight = 75;
  reg [10:0] hill_height = ground_y - hillheight;
  reg [10:0] bush_height = ground_y - bushheight;


  //MEMORIES---------------------------------------------
  reg [10:0] cloudleftadd;
  wire[24:0] cloudleftdata;
  reg [10:0] cloudrightadd;
  wire[24:0] cloudrightdata;
  reg [10:0] bush1add;
  wire[24:0] bush1data;
  reg [10:0] bush2add;
  wire[24:0] bush2data;
  reg [10:0] bush3add;
  wire[24:0] bush3data;
  reg [10:0] bush4add;
  wire[24:0] bush4data;
  reg [11:0] hill1add;
  wire[24:0] hill1data;
  reg [11:0] hill2add;
  wire[24:0] hill2data;
  reg [11:0] hill3add;
```

```
   wire[24:0] hill3data;
   reg [11:0] hill4add;
   wire[24:0] hill4data;
   reg [11:0] hill5add;
   wire[24:0] hill5data;


   backmem mymem(address,vclock,data_out);
   cloudleftmem mymem3(cloudleftadd,vclock, cloudleftdata);
   cloudrightmem mymem2(cloudrightadd,vclock,cloudrightdata);
   bush1rom mymem4(bush1add,vclock,bush1data);
   bush2rom mymem5(bush2add,vclock,bush2data);
   bush3rom mymem6(bush3add,vclock,bush3data);
   bush4rom mymem7(bush4add,vclock,bush4data);
//   hill1rom mymem8(hill1add,vclock,hill1data);
//   hill2rom mymem9(hill2add,vclock,hill2data);
//   hill3rom mymem10(hill3add,vclock,hill3data);
//   hill4rom mymem11(hill4add,vclock,hill4data);
//   hill5rom mymem12(hill5add,vclock,hill5data);
   //---------------------------------------------------------------------------


   wire [11:0] hloc1,hloc2,hloc3,hloc4,hloc5,hloc6,hloc7,hloc8,hloc9,hloc10,hloc11,hloc12;
   wire [11:0] hloc13,hloc14,hloc15,hloc16,hloc17,hloc18,hloc19,hloc20,hloc21,hloc22,hloc23,hloc24;
   wire [11:0] hloc25,hloc26,hloc27,hloc28,hloc29,hloc30,hloc31,hloc32, hloc33;

   assign hloc1 = pix_num - pixel_offset;
   assign hloc2 = 2*pix_num - pixel_offset;
   assign hloc3 = 3*pix_num - pixel_offset;
   assign hloc4 = 4*pix_num - pixel_offset;
   assign hloc5 = 5*pix_num - pixel_offset;
   assign hloc6 = 6*pix_num - pixel_offset;
   assign hloc7 = 7*pix_num - pixel_offset;
   assign hloc8 = 8*pix_num - pixel_offset;
   assign hloc9 = 9*pix_num - pixel_offset;
   assign hloc10 = 10*pix_num - pixel_offset;
   assign hloc11 = 11*pix_num - pixel_offset;
   assign hloc12 = 12*pix_num - pixel_offset;
   assign hloc13 = 13*pix_num - pixel_offset;
   assign hloc14 = 14*pix_num - pixel_offset;
   assign hloc15 = 15*pix_num - pixel_offset;
   assign hloc16 = 16*pix_num - pixel_offset;
   assign hloc17 = 17*pix_num - pixel_offset;
   assign hloc18 = 18*pix_num - pixel_offset;
   assign hloc19 = 19*pix_num - pixel_offset;
   assign hloc20 = 20*pix_num - pixel_offset;
   assign hloc21 = 21*pix_num - pixel_offset;
   assign hloc22 = 22*pix_num - pixel_offset;
   assign hloc23 = 23*pix_num - pixel_offset;
   assign hloc24 = 24*pix_num - pixel_offset;
   assign hloc25 = 25*pix_num - pixel_offset;
   assign hloc26 = 26*pix_num - pixel_offset;
   assign hloc27 = 27*pix_num - pixel_offset;
   assign hloc28 = 28*pix_num - pixel_offset;
   assign hloc29 = 29*pix_num - pixel_offset;
   assign hloc30 = 30*pix_num - pixel_offset;
   assign hloc31 = 31*pix_num - pixel_offset;
   assign hloc32 = 32*pix_num - pixel_offset;
   assign hloc33 = 33*pix_num - pixel_offset;

   assign index = (setup == 1) ? queue[block_num] : 0;

   always @ (posedge vclock) begin
                  if (reset == 1) begin
                                 setup <= 0; address <= 0;
                  end
                  else if (setup == 0) begin
                                 queue[address]<= data_out;
                                 address <= address + 1;
                                 if (address >= num_blocks) setup <= 1;
                  end
                  else if (new_box == 1) newb <= 1;
                  else if (newb == 1 && vcount > 775) begin
                                 address <= address + 1;
                                 for (count = 0; count <= num_blocks; count = count + 1) begin
                                          if (count == num_blocks) queue[count] <= data_out;
```

```
                    else queue[count] <= queue[count+1];
                end
                newb <=0;
        end
        else begin
            case (hcount)
             0: block_num <= 0;
                (hloc1): block_num <= block_num + 1;
                (hloc2): block_num <= block_num + 1;
                (hloc3): block_num <= block_num + 1;
                (hloc4): block_num <= block_num + 1;
                (hloc5): block_num <= block_num + 1;
                (hloc6): block_num <= block_num + 1;
                (hloc7): block_num <= block_num + 1;
                (hloc8): block_num <= block_num + 1;
                (hloc9): block_num <= block_num + 1;
                (hloc10): block_num <= block_num + 1;
                (hloc11): block_num <= block_num + 1;
                (hloc12): block_num <= block_num + 1;
                (hloc13): block_num <= block_num + 1;
                (hloc14): block_num <= block_num + 1;
                (hloc15): block_num <= block_num + 1;
                (hloc16): block_num <= block_num + 1;
                (hloc17): block_num <= block_num + 1;
                (hloc18): block_num <= block_num + 1;
                (hloc19): block_num <= block_num + 1;
                (hloc20): block_num <= block_num + 1;
                (hloc21): block_num <= block_num + 1;
                (hloc22): block_num <= block_num + 1;
                (hloc23): block_num <= block_num + 1;
                (hloc24): block_num <= block_num + 1;
                (hloc25): block_num <= block_num + 1;
                (hloc26): block_num <= block_num + 1;
                (hloc27): block_num <= block_num + 1;
                (hloc28): block_num <= block_num + 1;
                (hloc29): block_num <= block_num + 1;
                (hloc30): block_num <= block_num + 1;
                (hloc31): block_num <= block_num + 1;
                (hloc32): block_num <= block_num + 1;
                (hloc33): block_num <= 0;
                default: block_num <= block_num;
            endcase

            case (index)
             0: begin back_pixel <= 25'b1000000000000000000000000; end
                1: begin  //CLOUDLEFT at cloudheight
                    if (vcount >= cloudheight && vcount < cloudheight + 51) begin
                        cloudleftadd = (vcount - cloudheight) * 32 + hcount - (block_num*32-pixel_offset);
                        back_pixel[24] <= 0;
                        back_pixel[23:0] <= cloudleftdata;
                    end
                end
                2: begin //CLOUDRIGHT at cloudheight
                    if (vcount >= cloudheight && vcount < cloudheight + 51) begin
                        cloudrightadd = (vcount - cloudheight) * 32 + hcount - (block_num*32-pixel_offset);
                        back_pixel[24] <= 0;
                        back_pixel[23:0] <= cloudrightdata;
                    end
          end
/*              3: begin  //CLOUDLEFT at cloudheight -15
                    if (vcount >= cloudheight - 15 && vcount < cloudheight + 51 - 15) begin
                        cloudleftadd = (vcount - (cloudheight-15)) * 32 + hcount - (block_num*32-pixel_offset);
                        back_pixel[24] <= 0;
                        back_pixel[23:0] <= cloudleftdata;
                    end
                end
                4: begin //CLOUDRIGHT at cloudheight - 15
                    if (vcount >= cloudheight - 15 && vcount < cloudheight + 51 - 15) begin
                        cloudrightadd = (vcount - (cloudheight-15)) * 32 + hcount - (block_num*32-pixel_offset);
                        back_pixel[24] <= 0;
                        back_pixel[23:0] <= cloudrightdata;
                    end
          end
                5: begin  //CLOUDLEFT at cloudheight + 15
                    if (vcount >= cloudheight + 15 && vcount < cloudheight + 51 + 15) begin
                        cloudleftadd = (vcount - (cloudheight+15)) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= cloudleftdata;
						end
				end
				6: begin //CLOUDRIGHT at cloudheight + 15
						if (vcount >= cloudheight + 15 && vcount < cloudheight + 51 + 15) begin
								cloudrightadd = (vcount - (cloudheight+15)) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= cloudrightdata;
						end
		end
				7: begin  //CLOUDLEFT at cloudheight - 25
						if (vcount >= cloudheight - 25 && vcount < cloudheight + 51 - 25) begin
								cloudleftadd = (vcount - (cloudheight-25)) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= cloudleftdata;
						end
				end
				8: begin //CLOUDRIGHT at cloudheight - 25
						if (vcount >= cloudheight - 25 && vcount < cloudheight + 51 - 25) begin
								cloudrightadd = (vcount - (cloudheight-25)) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= cloudrightdata;
						end
		end */
/*				9: begin  //Hill1
						if (vcount >= hill_height && vcount < ground_y) begin
								hill1add = (vcount - hill_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= hill1data;
						end
				end
				10: begin //Hill2
						if (vcount >= hill_height && vcount < ground_y) begin
								hill2add = (vcount - hill_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= hill2data;
						end
		end
				11: begin  //Hill3
						if (vcount >= hill_height && vcount < ground_y) begin
								hill3add = (vcount - hill_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= hill3data;
						end
				end
				12: begin //Hill4
						if (vcount >= hill_height && vcount < ground_y) begin
								hill4add = (vcount - hill_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= hill4data;
						end
		end
				13: begin  //Hill5
						if (vcount >= hill_height && vcount < ground_y) begin
								hill5add = (vcount - hill_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= hill5data;
						end
				end*/
				14: begin //Bush1
						if (vcount >= bush_height && vcount < ground_y) begin
								bush1add = (vcount - bush_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= bush1data;
						end
		end
				15: begin  //Bush2
						if (vcount >= bush_height && vcount < ground_y) begin
								bush2add = (vcount - bush_height) * 32 + hcount - (block_num*32-pixel_offset);
								back_pixel[24] <= 0;
								back_pixel[23:0] <= bush2data;
						end
				end
				16: begin //Bush3
						if (vcount >= bush_height && vcount < ground_y) begin
```

```
                                                        bush3add = (vcount - bush_height) * 32 + hcount - (block_num*32-pixel_offset);
                                                        back_pixel[24] <= 0;
                                                        back_pixel[23:0] <= bush3data;
                                              end
                    end
                                    17: begin //Bush4
                                              if (vcount >= bush_height && vcount < ground_y) begin
                                                        bush4add = (vcount - bush_height) * 32 + hcount - (block_num*32-pixel_offset);
                                                        back_pixel[24] <= 0;
                                                        back_pixel[23:0] <= bush4data;
                                              end
                    end
                                    default: begin back_pixel <= 25'b1000000000000000000000000; end
                          endcase
                end
  end
endmodule


module pixel_draw (vclock,reset,
           hcount,vcount,
           mario_pixel, bg_pixel,map_pixel, pixel,back_pixel,timer_pixel,line_pixel);

  input vclock;              // 65MHz clock
  input reset;               // 1 to initialize module
  input [10:0] hcount;       // horizontal index of current pixel (0..1023)
  input [9:0] vcount; // vertical index of current pixel (0..767)
  input [24:0] mario_pixel;
  input [24:0] line_pixel;
  input [24:0] bg_pixel;
  input [24:0] map_pixel;
  input [24:0] back_pixel;
  input [24:0] timer_pixel;


  output [23:0] pixel;       // pixel to screen

  reg [23:0] background_color = 24'b100001001000010011111111;


  assign pixel = (line_pixel[24]==1) ? (mario_pixel[24] == 1) ? (map_pixel[24]==1) ? (timer_pixel[24]==1) ? (bg_pixel[24]==1) ?
                                              (back_pixel[24]==1) ? background_color : back_pixel : bg_pixel : timer_pixel :
                                               map_pixel : mario_pixel : line_pixel;
endmodule
```